

O'REILLY®

3-е издание

JavaScript

рецепты
для разработчиков



Адам Д. Скотт
Мэтью Макдоналд
Шелли Пауэрс



THIRD EDITION

JavaScript Cookbook

*Adam D. Scott, Matthew MacDonald,
and Shelley Powers*

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

JavaScript

Рецепты для разработчиков

3-е издание

Адам Д. Скотт, Мэтью Макдоналд,
Шелли Пауэрс



Санкт-Петербург • Москва • Минск

2023

ББК 32.988.02-018
УДК 004.738.5
С44

Скотт Адам Д., Макдоналд Мэтью, Пауэрс Шелли

C44 JavaScript. Рецепты для разработчиков. 3-е изд. — СПб.: Питер, 2023. — 528 с.: ил. — (Серия «Бестселлеры O'Reilly»).

ISBN 978-5-4461-2001-7

Зачем изобретать велосипед, сталкиваясь с очередной проблемой в JavaScript? Здесь вы найдете множество рецептов кода для типовых задач программирования, а также познакомитесь с методами создания приложений, работающих в любом браузере. Адаптируемые примеры кода можно вставить практически в любой проект, а заодно поглубже изучить JS.

С помощью этой книги вы научитесь настраивать эффективную среду разработки с редактором кода, статическим анализатором и тестовым сервером; станете лучше понимать функции JS, включая замыкания и генераторы; узнаете, как использовать классы и наследование — основные концепции ООП; освоите работу с мультимедиа, включая аудио, видео и SVG; научитесь управлять HTML и CSS; благодаря Node.js сможете использовать JavaScript где угодно; узнаете, как получать доступ к удаленным данным и управлять ими с помощью технологий REST, GraphQL и Fetch; освоите популярную среду разработки приложений Express; научитесь выполнять асинхронные операции с промисами, `async/await` и веб-процессами.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.988.02-018
УДК 004.738.5

Права на издание получены по соглашению с O'Reilly. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги. В книге возможны упоминания организаций, деятельность которых запрещена на территории Российской Федерации, таких как Meta Platforms Inc., Facebook, Instagram и др. Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими.

ISBN 978-1492055754 англ.

Authorized Russian translation of the English edition JavaScript Cookbook,
3E ISBN 9781492055754 © 2021

Adam D. Scott and Matthew MacDonald

This translation is published and sold by permission of O'Reilly Media, Inc.,
which owns or controls all rights to publish and sell the same.

ISBN 978-5-4461-2001-7

© Перевод на русский язык ООО «Прогресс книга», 2022

© Издание на русском языке, оформление ООО «Прогресс книга», 2023

© Серия «Бестселлеры O'Reilly», 2023

Краткое содержание

Предисловие	15
-------------------	----

Часть I ЯЗЫК JAVASCRIPT

Глава 1. Настройка среды разработки	20
Глава 2. Строки и регулярные выражения	56
Глава 3. Числа.....	84
Глава 4. Даты.....	98
Глава 5. Массивы	110
Глава 6. Функции	141
Глава 7. Объекты.....	169
Глава 8. Классы	198
Глава 9. Асинхронное программирование	227
Глава 10. Ошибки и тестирование.....	254

Часть II JAVASCRIPT В БРАУЗЕРЕ

Глава 11. Инструментарий браузера.....	280
Глава 12. Работа с HTML	289
Глава 13. Получение удаленных данных.....	327
Глава 14. Сохранение данных.....	351
Глава 15. Работа с мультимедиа.....	371
Глава 16. Создание веб-приложений.....	387

Часть III NODE.JS

Глава 17. Основы Node	420
Глава 18. Модули Node	442
Глава 19. Управление экосистемой Node	462
Глава 20. Удаленные данные.....	481
Глава 21. Построение веб-приложений с помощью Express.....	488
Об авторах	524
Иллюстрация на обложке.....	525

Оглавление

Предисловие	15
Аудитория этой книги	15
Структура издания.....	16
Условные обозначения.....	16
Примеры кода.....	17
Благодарности	18
От издательства.....	18

Часть I ЯЗЫК JAVASCRIPT

Глава 1. Настройка среды разработки	20
1.1. Выбираем редактор кода	21
1.2. Использование консоли разработки в браузере	23
1.3. Выполнение фрагментов кода в консоли разработчика	27
1.4. Использование строгого режима для выявления типичных ошибок.....	29
1.5. Вставка блоков HTML с помощью сокращенных команд Emmet	31
1.6. Установка менеджера пакетов npm (с Node.js).....	33
1.7. Загрузка пакета с помощью npm	36
1.8. Обновление пакета с помощью npm	40
1.9. Настройка локального тестового сервера	42
1.10. Соблюдение стандартов кодирования с помощью статического анализатора	45
1.11. Согласованное оформление кода с помощью форматировщика.....	50
1.12. Эксперименты в интерактивной среде JavaScript.....	52
Глава 2. Строки и регулярные выражения	56
2.1. Проверка того, что строка существует и она не пустая.....	56
2.2. Преобразование числового значения в форматированную строку.....	59

2.3. Вставка специальных символов	61
2.4. Использование эмодзи.....	63
2.5. Использование шаблонных литералов для более наглядной конкатенации строк	65
2.6. Сравнение строк без учета регистра	67
2.7. Проверка того, содержит ли строка заданную подстроку.....	68
2.8. Замена всех вхождений строки	69
2.9. Замена тегов HTML на именованные сущности	70
2.10. Использование регулярных выражений для создания шаблонов при замене строк.....	71
2.11. Извлечение списка из строки	74
2.12. Поиск по шаблону.....	76
2.13. Удаление пробелов в начале и в конце строки	80
2.14. Замена первой буквы строки на прописную	81
2.15. Валидация адреса электронной почты.....	82
Глава 3. Числа	84
3.1. Генерирование случайных чисел	84
3.2. Генерирование криптографически надежных случайных чисел.....	86
3.3. Округление до заданного десятичного разряда	88
3.4. Сохранение точности в дробных числах.....	89
3.5. Преобразование строки в число	91
3.6. Преобразование десятичных значений в шестнадцатеричные.....	93
3.7. Преобразование градусов в радианы	94
3.8. Вычисление длины дуги окружности.....	94
3.9. Манипуляции с очень большими целыми числами в формате BigInt	95
Глава 4. Даты.....	98
4.1. Получение текущих даты и времени.....	98
4.2. Преобразование строки в дату.....	100
4.3. Добавляем дни к дате.....	102
4.4. Сравнение дат и проверка двух дат на равенство.....	103
4.5. Вычисление времени, прошедшего между двумя датами	105
4.6. Представление даты в виде форматированной строки	107

Глава 5. Массивы	110
5.1. Проверка того, является ли объект массивом.....	110
5.2. Перебор всех элементов массива	111
5.3. Проверка равенства двух массивов.....	113
5.4. Разбиение массива на отдельные переменные	116
5.5. Передача массива в функцию, которая принимает список значений.....	117
5.6. Клонирование массива.....	118
5.7. Слияние двух массивов	120
5.8. Копирование части массива, выбранной по положению элемента	122
5.9. Извлечение из массива элементов, удовлетворяющих заданному условию	123
5.10. Очистка массива.....	124
5.11. Удаление дубликатов	125
5.12. Сведение двумерного массива.....	126
5.13. Точный поиск элементов массива.....	128
5.14. Поиск элементов массива, удовлетворяющих заданному критерию.....	129
5.15. Удаление и замена элементов массива.....	131
5.16. Сортировка массива объектов по заданному свойству	132
5.17. Преобразование элементов массива	134
5.18. Использование всех элементов массива в одном вычислении	134
5.19. Проверка содержимого массива.....	136
5.20. Построение коллекции недублирующихся значений.....	137
5.21. Создание коллекции элементов, индексируемой по ключу.....	139
Глава 6. Функции	141
6.1. Передача одной функции в другую в качестве аргумента.....	141
6.2. Использование стрелочных функций.....	145
6.3. Предоставление значения параметра по умолчанию	148
6.4. Создание функции, принимающей неограниченное число аргументов ...	149
6.5. Использование именованных параметров функции	150
6.6. Создание функции с сохранением состояния посредством замыкания....	153
6.7. Создание функции-генератора, которая возвращает несколько значений.....	155
6.8. Уменьшение избыточности за счет частичного применения.....	160

6.9. Фиксация this посредством привязки функций	163
6.10. Реализация рекурсивного алгоритма	166

Глава 7. Объекты..... 169

7.1. Проверка того, относится ли объект к заданному типу	169
7.2. Объединение данных с помощью объектных литералов	171
7.3. Проверка существования свойства у объекта	174
7.4. Перебор всех свойств объекта	176
7.5. Проверка того, является ли объект пустым	178
7.6. Объединение свойств двух объектов.....	180
7.7. Выбор способа определения свойств	181
7.8. Запрет любых изменений объекта	184
7.9. Перехват и изменение объектов с помощью прокси-объектов.....	185
7.10. Клонирование объектов	189
7.11. Создание глубокой копии объекта.....	191
7.12. Создание абсолютно уникальных ключей для свойств объекта	193
7.13. Создание перечислений с помощью Symbol.....	195

Глава 8. Классы 198

8.1. Создание класса для многократного использования	198
8.2. Добавление в класс новых свойств	202
8.3. Улучшенное строковое представление класса	207
8.4. Создание произвольного класса посредством шаблона «Конструктор»	208
8.5. Создание возможности для объединения методов класса в цепочку	211
8.6. Создание статических методов класса.....	213
8.7. Создание объектов посредством статических методов	216
8.8. Наследование функционала другого класса	218
8.9. Объединение классов JavaScript в модули.....	223

Глава 9. Асинхронное программирование..... 227

9.1. Обновление страницы в цикле	228
9.2. Использование функции, которая возвращает промис.....	230
9.3. Замена асинхронной функции с обратным вызовом на промис	234
9.4. Конкурентное выполнение нескольких промисов	237
9.5. Ожидание выполнения промиса с помощью await и async.....	240

9.6. Создание асинхронной функции-генератора	244
9.7. Выполнение фоновых задач с помощью Web Worker	247
9.8. Поддержка сообщений о ходе выполнения задач в Web Worker	251
Глава 10. Ошибки и тестирование	254
10.1. Обнаружение и обезвреживание ошибок	254
10.2. Перехват различных типов ошибок	257
10.3. Перехват асинхронных ошибок	259
10.4. Обнаружение необработанных ошибок	261
10.5. Выдача обычной ошибки	264
10.6. Выдача нестандартных ошибок	266
10.7. Написание модульных тестов для кода	268
10.8. Отслеживание покрытия кода тестами	276

Часть II

JAVASCRIPT В БРАУЗЕРЕ

Глава 11. Инструментарий браузера	280
11.1. Отладка кода JavaScript	280
11.2. Анализ производительности во время выполнения кода	282
11.3. Обнаружение неиспользуемого кода JavaScript	284
11.4. Выдача наилучших рекомендаций посредством Lighthouse	286
Глава 12. Работа с HTML	289
12.1. Доступ к определенному элементу, поиск его родительского и дочерних элементов	289
12.2. Перебор результатов, полученных от querySelectorAll(), с помощью forEach()	292
12.3. Привязка к элементу действия в ответ на щелчок	293
12.4. Поиск всех элементов с данным атрибутом	295
12.5. Выбор всех элементов определенного типа	296
12.6. Исследование дочерних элементов с помощью Selectors API	298
12.7. Изменение класса элемента	300
12.8. Присвоение элементу атрибута style	300
12.9. Создание абзаца и вставка в него текста	303
12.10. Вставка нового элемента в определенной точке DOM	304

12.11. Проверка того, установлен ли флажок.....	305
12.12. Вставка значений в таблицу HTML.....	306
12.13. Удаление строк из таблицы HTML.....	309
12.14. Скрытие частей страницы	312
12.15. Создание окон, всплывающих по наведению указателя мыши.....	313
12.16. Валидация данных формы.....	315
12.17. Выделение ошибочно заполненных полей форм и реализация специальных возможностей	319
12.18. Создание автоматически обновляемой области, доступной на специализированных устройствах	325
Глава 13. Получение удаленных данных	327
13.1. Запрос удаленных данных с помощью Fetch API.....	327
13.2. Использование метода XMLHttpRequest.....	331
13.3. Отправка данных формы.....	332
13.4. Заполнение списка выбора данными, полученными с сервера	336
13.5. Синтаксический анализ данных, полученных в формате JSON.....	340
13.6. Получение и синтаксический анализ данных в формате XML	342
13.7. Передача двоичных данных и загрузка изображения	344
13.8. Обмен HTTP cookies между несколькими доменами	345
13.9. Двухнаправленный обмен данными между клиентом и сервером посредством WebSockets.....	346
13.10. Длинный опрос удаленного источника данных.....	348
Глава 14. Сохранение данных.....	351
14.1. Сохранение информации в cookies.....	351
14.2. Хранение данных на стороне клиента с помощью sessionStorage	354
14.3. Создание хранилища данных на стороне клиента на основе localStorage.....	360
14.4. Сохранение больших объемов данных на стороне клиента с помощью IndexedDB	364
14.5. Упрощение IndexedDB с помощью библиотеки	367
Глава 15. Работа с мультимедиа	371
15.1. JavaScript для SVG	371
15.2. Доступ к SVG из скрипта веб-страницы.....	374
15.3. Построение столбчатой диаграммы в формате SVG с помощью библиотеки D3	376

15.4. Интеграция элементов SVG и Canvas в HTML.....	379
15.5. Выполнение процедуры в начале воспроизведения аудиофайла	382
15.6. Управление отображением видео с помощью элемента video и JavaScript	383
Глава 16. Создание веб-приложений.....	387
16.1. Создание пакетов JavaScript	387
16.2. JavaScript для мобильного интернета	389
16.3. Создание прогрессивного веб-приложения.....	392
16.4. Тестирование и профилирование прогрессивных веб-приложений	399
16.5. Получение текущего URL.....	403
16.6. Перенаправление на другой URL	404
16.7. Копирование текста в буфер обмена.....	405
16.8. Вывод на стационарном компьютере таких же уведомлений, как на мобильном устройстве.....	407
16.9. Открытие в браузере файла с локального устройства.....	410
16.10. Расширение возможностей с помощью Web Components	413
16.11. Выбор фреймворка для разработки на стороне клиента	416

Часть III

NODE.JS

Глава 17. Основы Node.....	420
17.1. Управление версиями Node с помощью Node Version Manager	420
17.2. Ответ на простой запрос браузера.....	423
17.3. Интерактивная проверка кода Node с помощью REPL.....	425
17.4. Чтение данных из файла и запись данных в файл	428
17.5. Получение данных из терминала	433
17.6. Получение пути к выполняемому скрипту	435
17.7. Работа с таймерами и циклом событий Node	436
Глава 18. Модули Node	442
18.1. Поиск нужного модуля Node через npm	442
18.2. Преобразование библиотеки в модуль Node.....	444
18.3. Перенос кода в модульную среду	445
18.4. Создание устанавливаемого модуля Node	448

18.5. Создание мультиплатформенных библиотек	454
18.6. Тестирование модулей	458
Глава 19. Управление экосистемой Node.....	462
19.1. Использование переменных среды	462
19.2. Что делать с адом обратных вызовов.....	464
19.3. Доступ к функциям командной строки из приложений Node	467
19.4. Передача аргументов в командную строку	470
19.5. Создание утилиты командной строки с подсказкой с помощью Commander	471
19.6. Обеспечение работоспособности экземпляра Node.....	474
19.7. Отслеживание изменений и перезапуск приложения в процессе разработки на локальном компьютере	475
19.8. Многократное выполнение задач по расписанию.....	477
19.9. Тестирование производительности и возможностей приложения WebSockets.....	478
Глава 20. Удаленные данные	481
20.1. Получение удаленных данных.....	481
20.2. Анализ экранных данных	483
20.3. Доступ к данным в формате JSON посредством RESTful API.....	485
Глава 21. Построение веб-приложений с помощью Express	488
21.1. Использование Express для ответов на запросы.....	488
21.2. Использование Express-Generator	492
21.3. Задача маршрутизации.....	497
21.4. Работа с OAuth.....	499
21.5. Аутентификация пользователей в OAuth 2 с помощью Passport.js.....	510
21.6. Обработка форматированных данных	514
21.7. Построение RESTful API.....	516
21.8. Построение API GraphQL.....	519
Об авторах	524
Иллюстрация на обложке.....	525

Предисловие

Садясь за работу над последним изданием книги «JavaScript. Рецепты для разработчиков», я решил, что «рецепты» — точная метафора. Что отличает хороший сборник кулинарных рецептов? Перебрав несколько кулинарных справочников, стоящих на полке в кухне, я обнаружил, что мои любимые — это те, в которых не только описаны самые вкусные блюда, но и дается много советов, основанных на богатом опыте знающих людей. Хорошая кулинарная книга — не та, где вы найдете все варианты приготовления мяса по-бургундски, а та, где описаны наилучшие, по мнению автора, методики и рецепты, к которым обычно прилагается несколько советов. Именно такой концепцией мы и решили руководствоваться, собирая коллекцию рецептов по JavaScript. Советы, представленные в этой книге, дают трое маститых профессионалов, но это кульминация главным образом *нашего* уникального опыта. Окажись на нашем месте другие разработчики, они бы написали похожую, но другую книгу.

JavaScript вырос в невероятный, высокоэффективный, многоцелевой язык программирования. С нашей подборкой рецептов в руках вы сможете не только решать всевозможные проблемы, которые вам встретятся, но и придумывать собственные рецепты.

Аудитория этой книги

Предупреждаем сразу: стремясь охватить все разнообразие тем и областей, в которых сегодня используется JavaScript, мы не рассчитывали на новичков в программировании. Для тех, кто начинает изучать программирование на JavaScript, есть так много хороших книг и учебных пособий, что мы со спокойной совестью посвятили свою *практикующим разработчикам* — тем, кто ищет решение конкретных проблем и задач на JavaScript.

Если вы успели попрактиковаться с JavaScript лишь в течение пары месяцев — возможно, попробовали силы в разработке для Node или веб-разработке, то изучение материала этой книги не должно вызвать у вас затруднений. Издание станет полезным руководством и для тех разработчиков, кто в основном имеет дело с другим языком программирования, но время от времени испытывает потребность в JavaScript. Наконец, если вы действующий разработчик на JavaScript, но вас иногда ставят в тупик некоторые особенности языка, то эта книга также станет для вас полезным ресурсом.

Структура издания

У книги есть два типа читателей. Первые — те, кто прочитает ее от корки до корки, подобрав по дороге каждую крошку полезных знаний. Вторые будут углубляться в чтение лишь по мере необходимости в поисках решения определенной задачи или категории стоящих перед ними проблем. Мы постарались структурировать книгу таким образом, чтобы она была полезной обеим категориям читателей. Для этого разбили ее на три части.

- Часть I. Язык JavaScript. Сюда вошли рецепты для JavaScript как языка программирования.
- Часть II. JavaScript в браузере. Здесь описан JavaScript в его естественной среде обитания — в браузере.
- Часть III. Node.js. Здесь JavaScript рассматривается сквозь призму Node.js.

Все главы книги разделены на несколько самостоятельных рецептов. Каждый из них состоит из следующих частей.

- *Задача* — описание типичного сценария разработки, в котором может быть использован JavaScript.
- *Решение* — решение задачи с примером кода и минимальным описанием.
- *Обсуждение* — подробное обсуждение примера кода и применяемых методик.

Кроме того, в рецепте могут присутствовать разделы «Читайте также» со списком литературы, рекомендованной для дальнейшего чтения, и «Дополнительно» с описанием других методик.

Условные обозначения

В этой книге использованы следующие условные обозначения.

Курсив

Курсивом выделены новые термины.

Рубленый шрифт

Этим шрифтом выделены элементы пользовательского интерфейса, URL, адреса электронной почты и названия клавиш.

Моноширинный шрифт

Любой компьютерный код — команды, массивы, элементы, операторы, опции, переключатели, переменные, атрибуты, ключи, функции, типы, классы, пространства имен, методы, модули, свойства, параметры, значения, объекты,

события, обработчики событий, XML- и HTML-теги, макросы, названия и содержимое файлов, результаты работы команд.

Моноширинный жирный шрифт

Команды и другой текст, который должен вводить пользователь.



Этот значок обозначает обычное примечание.



Этот значок обозначает совет или рекомендацию.



Этот значок обозначает предупреждение или предостережение.

Вам также могут пригодиться упоминаемые в книге сайты и веб-страницы — это упростит поиск информации онлайн. Обычно приводится адрес (URL) и название (заглавие или другой подходящий заголовок) этих ресурсов. Некоторые адреса довольно длинные, так что вам, возможно, будет проще найти такие страницы, введя их название в той поисковой системе, которой вы обычно пользуетесь. Названия помогут и в том случае, если страницу не удастся найти по адресу: URL может измениться, но название страницы останется прежним.

Примеры кода

Вспомогательные материалы (примеры кода, упражнения и т. п.) доступны для скачивания по адресу <https://github.com/javascripteverywhere/cookbook>.

В общем случае все примеры кода из книги вы можете использовать в своих программах и в документации. Вам не нужно обращаться в издательство за разрешением, если вы не собираетесь воспроизводить существенные части программного кода. Если вы разрабатываете программу и используете в ней несколько фрагментов кода из книги, вам не нужно обращаться за разрешением. Но для продажи или распространения примеров из книги вам потребуется разрешение от издательства O'Reilly. Вы можете отвечать на вопросы, цитируя данную книгу или примеры из нее, но для включения существенных объемов программного кода из книги в документацию вашего продукта потребуется разрешение.

Мы рекомендуем, но не требуем добавлять ссылку на первоисточник при цитировании. Под ссылкой на первоисточник мы подразумеваем указание авторов, издательства и ISBN.

За получением разрешения на использование значительных объемов программного кода из книги обращайтесь по адресу permissions@oreilly.com.

Благодарности

Это третье издание книги «JavaScript. Рецепты для разработчиков». Первые два были написаны Шелли Пауэрс. Третье издание написали и обновили Адам Скотт и Мэтью Макдоналд. Адам и Мэтью благодарят своих редакторов Анджелу Руфино и Дженнифер Поллок, которые сопровождали проект на протяжении всего периода его трудного взросления; Сару Вакс, Шайка Нитлинга и Элизабет Робсон, которые сделали множество тонких замечаний и полезных предложений. Адам также благодарен Джону Пакстону за поддержку и общение на ранних стадиях работы над книгой.

Шелли благодарит своих редакторов Симона Сен-Лорана и Брайана Макдоналда, а также доктора Акселя Раушмайера и Семми Пьюруол.

Все вместе мы благодарим производственный персонал O'Reilly за постоянную помощь и поддержку.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

ЧАСТЬ I

Язык JavaScript

ГЛАВА 1

Настройка среды разработки

Возможно, вам приходилось слышать, что «разработчики делают инструменты разработки». Это, конечно же, преувеличение, но никто не захочет остаться один на один с кодом JavaScript без любимых инструментов для редактирования, анализа и отладки.

При настройке среды разработки первое, чем вы озаботитесь, — это редактор кода. Даже в самом простом редакторе есть такие базовые вещи, как автодополнение и подсветка синтаксиса — две простые функции, предотвращающие кучу потенциальных ошибок. В современных редакторах кода функций гораздо больше. В частности, это интеграция с системами управления версиями, такими как GitHub, построчная отладка и интеллектуальный рефакторинг. Некоторые из этих функций подключаются к редактору в виде плагинов. Иногда функции запускаются из терминала или являются частью процесса сборки. Но как бы вы ни использовали свой инструментарий, подобрать удачное сочетание инструментов, соответствующее вашему стилю программирования, среде разработки и типам проектов, — обязательная часть общего удовольствия. Это примерно такая же обязательная процедура, как приобретение инструментов для профессионала, занимающегося ремонтом квартир, или инвестиции в правильное кухонное оборудование для амбициозного шеф-повара.

Инструменты не выбирают раз и навсегда. Ваши предпочтения как разработчика могут меняться. По мере профессионального роста, а также при появлении новых полезных инструментов ваш инструментарий будет расширяться. В этой главе описан минимальный комплект, которым необходимо обзавестись любому разработчику JavaScript, прежде чем браться за проект. Однако у вас остается широчайший выбор между различными, но в целом эквивалентными вариантами. И как отмечают многие мудрые люди, о вкусах не спорят!



В этой главе мы в каком-то смысле поработаем адвокатами. Мы познакомим вас с некоторыми из наших любимых инструментов и дадим ссылки на другие, ничем не худшие варианты. Не будем пытаться познакомить вас со всеми возможными инструментами, а лишь предложим несколько отличных общепринятых вариантов, которые можно задействовать на начальном этапе.

1.1. Выбираем редактор кода

Задача

Использовать для написания кода редактор, который понимает синтаксис JavaScript.

Решение

Если вы спешите, то не ошибетесь, выбрав наш любимый Visual Studio Code, который для краткости часто называют VS Code. Этот бесплатный редактор с открытым кодом существует в версиях для Windows, Macintosh и Linux.

Если у вас есть время на поиски, то можете обратить внимание на несколько других редакторов. Их список, приведенный в табл. 1.1, далеко не полон, но в нем перечислены самые популярные редакторы.

Таблица 1.1. Автономные редакторы кода

Редактор	Поддерживаемые платформы	Открытый код	Стоимость	Примечания
Visual Studio Code (https://code.visualstudio.com)	Windows, Macintosh, Linux	Да	Бесплатно	Отличный редактор для любого языка и наш выбор номер один для разработки на JavaScript
Atom (https://atom.io)	Windows, Macintosh, Linux	Да	Бесплатно	Большинство глав этой книги написаны с использованием Atom с плагинами для поддержки AsciiDoc
WebStorm (https://jetbrains.com/webstorm)	Windows, Macintosh, Linux	Нет	Бесплатно для разработки ПО с открытым кодом и для образовательных целей, иначе примерно 60 долларов в год на пользователя	Более сложная среда — скорее традиционная IDE, чем редактор кода
Sublime Text (https://sublimetext.com)	Windows, Macintosh, Linux	Нет	Однократный платеж 80 долларов на пользователя без контроля за лицензированием или ограничений по времени	Популярный редактор, известный быстрой обработкой больших текстовых файлов
Brackets (http://brackets.io)	Windows, Macintosh	Да	Бесплатно	Спонсируемый Adobe проект, ориентированный на веб-разработку

Какой бы редактор кода вы ни выбрали, для того чтобы начать новый проект, вам потребуется выполнить примерно одни и те же операции. Вначале создайте папку проекта (например, `test-site`). Затем откройте редактор кода, найдите команду наподобие **File ► Open Folder** и выберите созданную папку проекта. В большинстве редакторов кода сразу откроется панель с содержимым этой папки, представленным в виде удобного списка или дерева, чтобы можно было быстро переключаться между файлами.

В папке проекта вы также будете размещать используемые пакеты (рецепт 1.7), сохранять файлы конфигурации приложения и правила кодирования (рецепт 1.10). Кроме того, у редактора есть встроенный терминал (см. подраздел «Дополнительно: терминал и оболочка» далее), который всегда запускается из папки текущего проекта.

Обсуждение

Рекомендовать лучший редактор — все равно что выбирать десерт другому человеку. Подходящих вариантов не менее дюжины, и при выборе не последнюю роль играют личные предпочтения. Большинство предложений, перечисленных в табл. 1.1, соответствуют всем основным требованиям.

- Кроссплатформенность — не имеет значения, какую операционную систему вы используете.
- Поддержка плагинов — можно подключить все необходимые функции. Многие инструменты, упоминаемые в книге (такие как форматировщик кода Prettier, описанный в рецепте 1.10), реализованы в виде плагинов, которые подключаются к различным редакторам кода.
- Мультиязычность — это позволит писать код не только на HTML, CSS и JavaScript, но и на других языках программирования.
- Поддержка сообщества — благодаря этому вы можете быть уверены, что выбранный редактор будет еще долго поддерживаться и совершенствоваться.
- Свободное распространение или разумная цена.

VS Code, который мы посчитали наилучшим вариантом, — это редактор кода от Microsoft со встроенной поддержкой JavaScript. Сам редактор тоже написан на JavaScript и размещен на платформе Electron. (Точнее, редактор написан на TypeScript — более строгом варианте JavaScript, который компилируется в JavaScript перед дистрибуцией или выполнением.)

VS Code во многих отношениях является стильным младшим братом массивной IDE Visual Studio, которая также существует в бесплатной редакции Community и тоже позволяет писать код на JavaScript. Но VS Code лучше подходит для тех разработчиков, которые уже не работают со стекom Microsoft.NET. Такого удачного баланса удалось достичь благодаря тому, что изначально VS Code очень прост, но его можно до бесконечности настраивать благодаря библиотеке, насчитывающей

сотни разрабатываемых сообществом плагинов (<https://oreil.ly/RvMZ9>). В обзорах для разработчиков, публикуемых на Stack Overflow, VS Code постоянно упоминается как самый популярный редактор кода для разных языков программирования.

Читайте также

Для того чтобы познакомиться с основными функциями и общей структурой VS Code, советуем посмотреть отличный набор начальных видеоуроков (<https://oreil.ly/iiRhA>). В этой главе вы научитесь применять сокращенные команды Emmet для VS Code (рецепт 1.5), а также подключать плагины ESLint (рецепт 1.10) и Prettier (рецепт 1.11).

1.2. Использование консоли разработки в браузере

Задача

Иметь возможность просматривать сообщения об ошибках на веб-странице и информацию, которую мы сами выводим в консоль.

Решение

Для этого мы будем использовать консоль разработки в браузере. В табл. 1.2 показано, как открыть инструменты разработки во всех современных браузерах.

Таблица 1.2. Сочетания клавиш, чтобы открыть консоль разработки

Браузер	Операционная система	Сочетание клавиш
Chrome	Windows или Linux	F12 или Ctrl+Shift+J
Chrome	Macintosh	Cmd-Option-J
Edge	Windows или Linux	F12 или Ctrl+Shift+J
Firefox	Windows или Linux	F12 или Ctrl+Shift+J
Firefox	Macintosh	Cmd-Shift-J
Safari ¹	Macintosh	Cmd-Option-C
Opera	Windows	Ctrl+Shift+J
Opera	Macintosh	Cmd-Option-J

¹ Чтобы использовать консоль разработки в Safari, ее нужно вначале активировать. Для этого выберите в меню команду Safari Menu ► Preferences, перейдите на вкладку Advanced и установите флажок Show Develop menu in the menu bar.

Как правило, инструменты разработки представляют собой группу панелей со вкладками и размещаются в правой или нижней части окна браузера. Сообщения, которые выводятся с помощью `console.log()`, а также сообщения об ошибках появляются на панели **Console**.

Вот полный код страницы, которая выводит текст в консоль и затем завершается с ошибкой:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <meta http-equiv="X-UA-Compatible" content="ie=edge" />
    <title>Log and Error Test</title>
  </head>
  <body>
    <h1>Log and Error Test</h1>

    <script>
      console.log('This appears in the developer console');
    </script>

    <script>
      // Этот код вызовет ошибку, сообщение о которой
      // появится в консоли
      const myNumber =
    </script>
  </body>
</html>
```

На рис. 1.1 показано, что появится в консоли разработчика при отображении этой страницы. Первым будет показано выводимое сообщение, а затем — сообщение

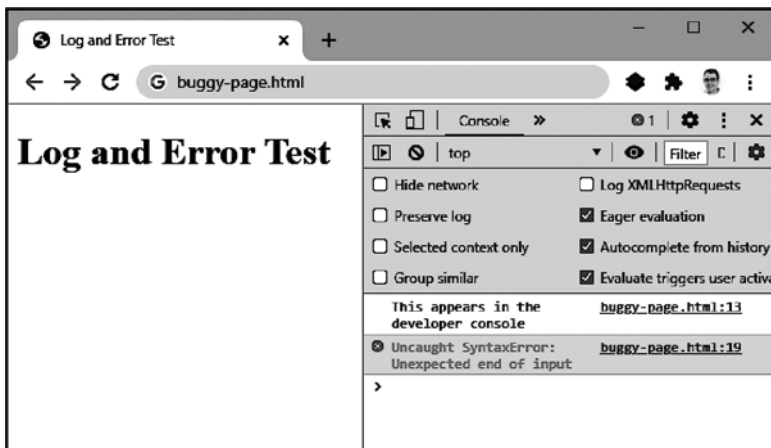


Рис. 1.1. Просмотр содержимого консоли разработчика в Chrome

об ошибке (`SyntaxError` с текстом `Unexpected end of input`). Ошибки выводятся красным цветом, каждое такое сообщение Chrome любезно сопровождает ссылкой, благодаря которой можно быстро перейти к исходному коду и увидеть, что именно стало причиной ошибки. Строки веб-страниц и файлов со сценариями автоматически нумеруются. В данном примере легко отличить код с выводом сообщения (строка 13) от кода, который вызвал ошибку (закрывающий тег `</script>` в строке 19).

Обсуждение

В книге мы будем постоянно использовать конструкцию `console.log()` для быстрого вывода тестовых сообщений. Но у объекта `console` есть и другие методы, которыми можно пользоваться. Самые полезные методы приводятся в табл. 1.3.

Таблица 1.3. Методы объекта `console`

Метод	Описание
<code>console.warn(object)</code>	Аналог <code>console.log()</code> , но текст выводится на желтом фоне
<code>console.error(object)</code>	Аналог <code>console.log()</code> , но текст выводится на красном фоне. Обычно используется для вывода объектов с сообщениями об ошибках
<code>console.assert(expression, object)</code>	Если значением выражения (<code>expression</code>) является <code>false</code> , то сообщение (<code>message</code>) выводится в консоли с трассировкой стека
<code>console.trace()</code>	Вывод трассировки стека
<code>console.count(label)</code>	Вывод количества вызовов метода с данным именем (<code>label</code>)
<code>console.dir(object)</code>	Вывод всех свойств объекта в виде разворачивающегося древовидного списка
<code>console.group()</code>	Создает новую группу с заданным именем. Все последующие сообщения в консоли выводятся под этим заголовком и с отступом, так что кажутся логически связанными частями одного раздела. Чтобы закрыть группу, используется метод <code>console.groupEnd()</code>
<code>console.time(label)</code>	Запускает таймер с заданным названием (<code>label</code>)
<code>console.timeEnd(label)</code>	Останавливает таймер с заданным названием (<code>label</code>) и выводит истекшее время



В консолях современных браузеров часто используется отложенное вычисление объектов и массивов. Такая ситуация может возникать, если вывести объект с помощью `console.log()`, изменить его, а потом вывести этот объект еще раз. Если делать это в коде сценария на веб-странице, то зачастую можно обнаружить, что `console.log()` выводит один и тот же уже измененный объект, несмотря на то что первый вызов произошел до того, как объект был изменен!

Чтобы избежать этой причуды браузера, нужно перед выводом явно преобразовать объект в строку. Этот трюк сработает, поскольку консоль не выполняет отложенных вычислений для строк. Так делать не всегда удобно (например, это не поможет, если нужно вывести массив, в состав которого входят объекты), но в большинстве случаев это решает проблему.

Разумеется, консоль — это лишь одна из панелей (или вкладок) раздела с инструментами разработки. Осмотревшись, вы обнаружите еще несколько полезных функций, расположенных на других панелях. Их точное расположение и названия зависят от браузера. Далее перечислены некоторые из основных таких функций для Chrome.

- *Elements*. На этой панели можно увидеть разметку HTML для тех или иных частей страницы и проверить, какие стили CSS применяются к отдельным элементам. Можно даже изменить эту разметку и стили (временно), чтобы быстро увидеть результат изменений, которые вы намерены внести.
- *Sources*. На данной панели можно просмотреть содержимое всех файлов, которые используются для данной страницы, включая библиотеки JavaScript, изображения и таблицы стилей.
- *Network*. На этой панели можно увидеть размер и время загрузки страницы и ее ресурсов, а также прочитать асинхронные сообщения, передаваемые по сети (например, при запросе `fetch`).
- *Performance*. Эту панель используют, чтобы отслеживать время выполнения кода (см. рецепт 11.2).
- *Application*. Здесь можно увидеть все данные, которые этот сайт хранит в cookie — как в локальном хранилище, так и с помощью API IndexedDB.

Советуем поэкспериментировать с этими панелями, чтобы получить представление о том, как они работают, или ознакомиться с документацией, предоставляемой Google (<https://oreil.ly/cZ6AP>).

Читайте также

В рецепте 1.3 показано, как быстро выполнить фрагмент кода в консоли разработчика.

1.3. Выполнение фрагментов кода в консоли разработчика

Задача

Иметь возможность выполнить фрагмент кода, не открывая его в редакторе и не создавая файлы для HTML и JavaScript.

Решение

Воспользуемся консолью разработки в браузере. Откройте средства разработки (как описано в рецепте 1.2). Выберите панель **Console**. Вставьте или введите в ней код JavaScript.

Чтобы сразу выполнить код, нажмите клавишу **Enter**. Если хотите выполнить несколько строк кода, то нажимайте в конце каждой строки **Shift+Enter**, чтобы вставить мягкий перевод строки. Нажимайте **Enter** только после того, как закончите ввод кода и захотите выполнить весь код.

Часто бывает нужно изменить прежний фрагмент кода и заново его выполнить. Во всех современных браузерах в консоли разработки есть функция истории, которая очень упрощает эту задачу. Для того чтобы вывести фрагмент кода, выполнявшийся последним, просто нажмите клавишу со стрелкой вверх. Чтобы вывести код, который выполнялся *до этого*, нажмите стрелку вверх несколько раз.

На рис. 1.2 показан пример, где фрагмент кода в первый раз не выполнялся из-за синтаксической ошибки. Затем этот код был извлечен из истории, изменен и снова выполнен. Результат выполнения (число 15) появился в консоли под кодом.

Функция истории работает только до тех пор, пока вы *не начнете* вводить новый код. Если командная строка консоли не пустая, то при нажатии клавиши со стрелкой вверх вы не вернете предыдущий фрагмент, а будете перемещаться по текущему блоку кода.

Обсуждение

В консоли разработчика можно вводить код JavaScript точно в том же виде, что и в блоке сценария. Другими словами, вводить функции и вызывать их или определить класс и создать его экземпляр. Можно также обращаться к объекту **document**, взаимодействовать с элементами HTML, размещенными на текущей странице, выводить предупреждения и текст в консоль. (Сообщения будут появляться сразу под кодом.)

Есть одна загвоздка: при выполнении в консоли больших фрагментов кода возможен конфликт имен, так как JavaScript не позволяет создавать в одной и той же

области видимости несколько переменных или функций с одинаковыми именами. Например, рассмотрим следующий простой блок кода:

```
const testValue = 40+12;
console.log(testValue);
```

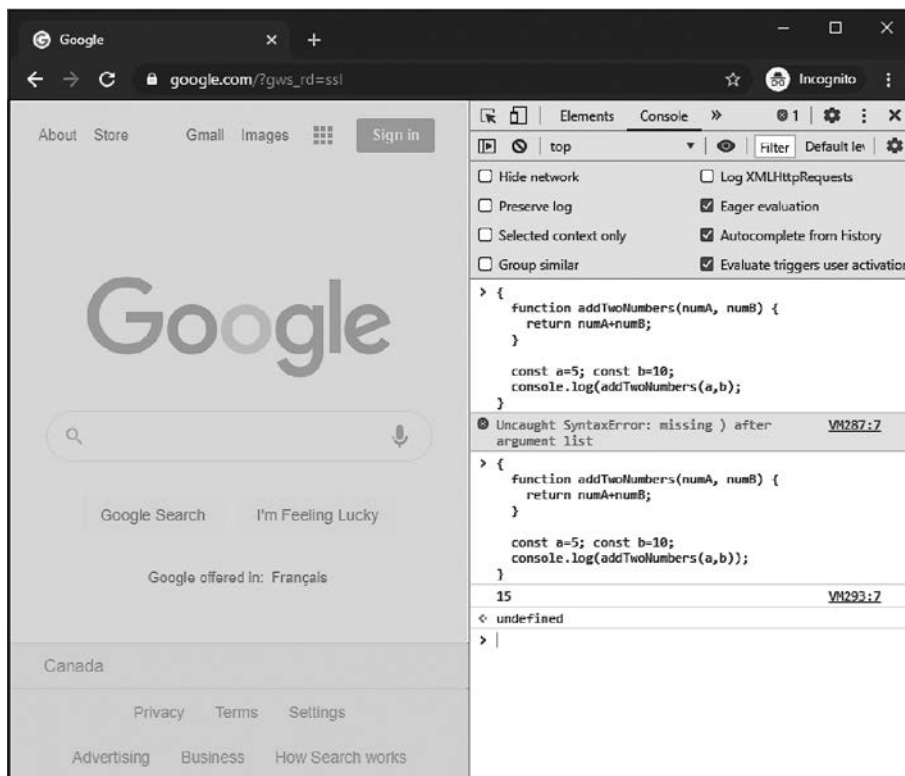


Рис. 1.2. Выполнение кода в консоли

Если выполнить его один раз, все работает хорошо. Но если вызвать его из истории (нажав клавишу со стрелкой вверх), отредактировать и попробовать запустить снова, то получим ошибку с сообщением о том, что переменная `testValue` уже объявлена. Конечно, эту переменную можно переименовать, но если вы пытаетесь усовершенствовать фрагмент кода с несколькими переменными и функциями, то такое переименование быстро станет неудобным. Вместо этого можно выполнить команду `location.reload()`, чтобы обновить страницу, но обновление сложных страниц может выполняться медленно. Кроме того, при обновлении можно потерять какое-то состояние страницы, которое вы хотели бы сохранить.

К счастью, у этой проблемы есть более простое решение. Заключите весь блок в пару скобок, чтобы получилась новая область видимости имен. Теперь можно

спокойно выполнять этот код сколько угодно раз, поскольку каждый раз будет создаваться новый контекст (и удаляться по окончании выполнения):

```
{  
  const testValue = 40+12;  
  console.log(testValue);  
}
```

Читайте также

В рецепте 11.1 продемонстрировано искусство отладки в консоли разработки. В рецепте 11.2 показано, как можно применять консоль разработки для анализа производительности.

1.4. Использование строгого режима для выявления типичных ошибок

Задача

Запретить использование потенциально опасных операций, таких как автоматическое создание переменных или операторы, которые могут не выполняться и никак не сообщать об этом.

Решение

Поставьте в начале файла с кодом JavaScript директиву `strict`:

```
'use strict';
```

Или можно оформить код JavaScript в виде модуля, который всегда запускается в строгом режиме (рецепт 8.9).

Обсуждение

JavaScript пользуется в некоторой степени заслуженной репутацией языка, терпимого к небрежно написанному коду. Проблема в том, что языки, игнорирующие мелкие нарушения правил, ставят разработчиков в невыгодное положение. В конце концов, вы не сможете исправить ошибку, если не узнаете о ней.

Далее показан пример плохого кода на JavaScript. Сможете ли вы найти в нем ошибку?

```
// Эта функция суммирует последовательные числа  
function addRange(start, end) {  
  let sum = 0;  
  for (let i = start; i < end+1; i++) {  
    sum += i;  
  }  
}
```

```
    }  
    return sum;  
}  
  
// Суммируем числа от 10 до 15  
let startNumber = 10;  
let endNumber = 15;  
console.log(addRange(startNumber, endNumber)); // Выведет 75  
  
// Теперь суммируем числа от 1 до 5  
startnumber = 1;  
endNumber = 5;  
console.log(addRange(startNumber, endNumber)); // Выведет 0, а не 15,  
// как ожидалось
```

Код выполнится без ошибок, но результат будет не тот, которого мы ожидали. Проблема кроется в этой строке:

```
startnumber = 1;
```

Дело в том, что если вы пытаетесь присвоить значение переменной, которую не объявили явно, то JavaScript создает эту переменную. Поэтому, если присвоить значение переменной `startnumber`, в то время как имелась в виду `startNumber`, JavaScript тихо создаст переменную `startnumber`. В результате значение, которое мы хотели присвоить `startNumber`, попадет в другую переменную, и мы его больше никогда не увидим и не сможем использовать.

Чтобы устранить эту проблему, нужно поставить в начале файла, перед кодом функции, директиву строгого режима:

```
'use strict';
```

Теперь, когда JavaScript дойдет до присвоения значения переменной `startnumber`, появится сообщение об ошибке `ReferenceError`. Выполнение кода будет прервано, работа сценария завершится. При этом в консоли появится набранное красными буквами сообщение об ошибке. В нем будет описана проблема и указан номер строки, в которой произошла ошибка. Теперь исправить ошибку очень просто.

Строгий режим позволяет отловить множество мелких, но зловредных ошибок. Вот лишь некоторые из них.

- Присвоение значений необъявленным переменным.
- Дублирование имен параметров (например, `function(a, b, a)`) или имен свойств объектных литералов (например, `{a: 5, a: 0}`).
- Попытки присвоить значения зарезервированным ключевым словам, таким как `Infinity` или `undefined`.
- Попытки изменить неизменяемые свойства (рецепт 7.7) или заблокированные объекты (рецепт 7.8).

Многие из таких действий не удалось бы выполнить и без использования строгого режима. Но это произошло бы *тихо* и, возможно, привело бы к печальной ситуации, когда код работает не так, как нужно, а вы понятия не имеете, почему это происходит.



Скорее всего, ваш редактор кода можно настроить так, чтобы директива `use strict` по умолчанию вставлялась во все создаваемые файлы. Например, у Visual Studio Code есть как минимум три небольших расширения (<https://oreil.ly/ye0o7>), которые это делают.

Строгий режим позволяет отловить не так уж много ошибок. Большинство разработчиков используют также инструмент статического анализа кода (рецепт 1.10), способный обнаруживать гораздо более широкий спектр ошибок и потенциально рискованных операций. Собственно говоря, разработчики так сильно полагаются на анализаторы кода, что иногда не утруждают себя включением строгого режима. Однако его рекомендуется применять всегда как простейшее средство гарантировать, что вы не выстрелите сами себе в ногу.

Читайте также

Подробнее о том, что невозможно сделать при активированном строгом режиме, читайте в документации строгого режима (<https://oreil.ly/Z7QhF>). Использование модулей, которые всегда выполняются в строгом режиме, описано в рецепте 8.9.

1.5. Вставка блоков HTML с помощью сокращенных команд Emmet

Задача

Иметь возможность вставить в файл большой кусок стандартного HTML, не вбивая руками каждый открывающий и закрывающий тег.

Решение

Emmet — это функция редактора, которая автоматически заменяет predefined текстовые сокращения на стандартные блоки HTML. В некоторых редакторах кода, таких как Visual Studio и WebStorm, Emmet поддерживается изначально. В других редакторах, таких как Atom и Sublime Text, для этого нужно подключать плагины. Чтобы найти нужный плагин, обычно достаточно открыть библиотеку плагинов и ввести **Emmet** в строке поиска. В случае сомнений загляните в список плагинов для поддержки Emmet (<https://emmet.io/download>).

Для того чтобы использовать Emmet, создайте пустой файл и сохраните его с расширением `.html` или `.htm`, чтобы редактор кода распознал его как документ HTML.

Затем введите одно из сокращений Emmet и нажмите клавишу табуляции. (В некоторых редакторах нужна другая клавиша или комбинация клавиш — это может быть **Enter** или **Ctrl+E**, но обычно применяется табуляция.) Введенный текст будет автоматически преобразован в соответствующий блок или элемент разметки.

Например, сокращение Emmet `input:time` преобразуется в следующий элемент:

```
<input type="time" name="" id="" />
```

На рис. 1.3 показано, как VS Code распознает сокращение Emmet по мере ввода. В VS Code поддерживается автодополнение сокращенных команд Emmet, так что вы увидите возможные варианты. В меню автодополнения выводится примечание **Emmet Abbreviation**, которое показывает, что сейчас вы набираете не HTML-код, а сокращенную команду Emmet, которая будет *преобразована* в HTML.

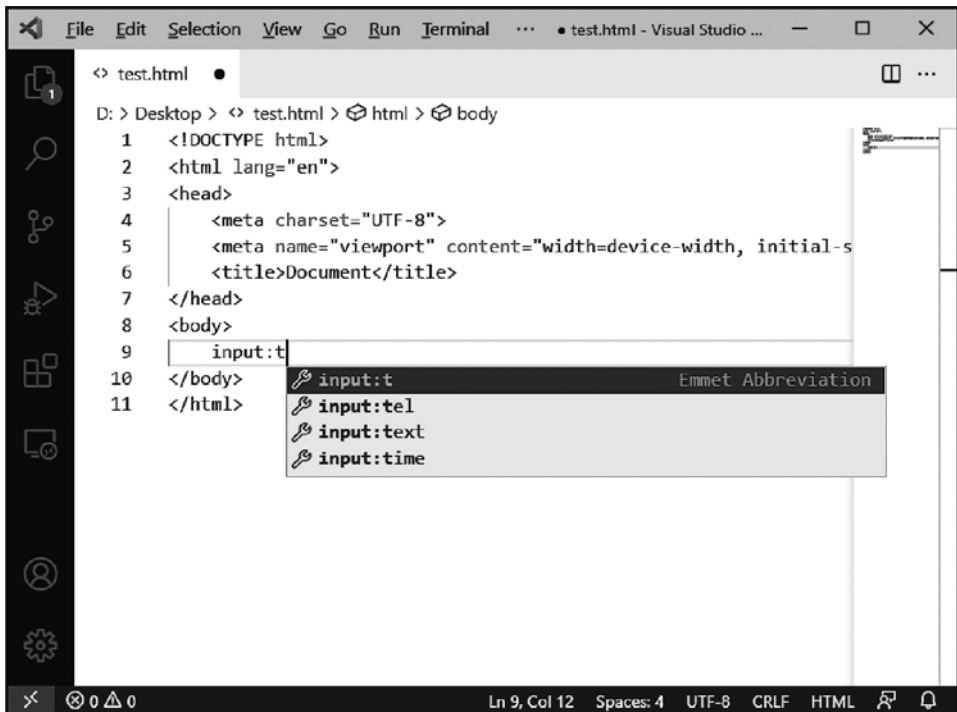


Рис. 1.3. Использование Emmet в VS Code

Обсуждение

Несмотря на простой синтаксис, Emmet удивительно гибок. Более сложные выражения позволяют создавать вложенные сочетания элементов, назначать атрибуты и вставлять в имена элементов последовательную нумерацию. Например, для того чтобы создать маркированный список из пяти элементов, используется

сокращение `ul>li*5`, в результате чего в разметку HTML будет вставлен следующий блок:

```
<ul>
  <li></li>
  <li></li>
  <li></li>
  <li></li>
  <li></li>
</ul>
```

А с помощью сокращенной команды `html:5` можно создать стандартный скелет веб-страницы, отвечающий современному стандарту HTML5:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>

</body>
</html>
```

Все эти функции описаны в документации Emmet (<https://docs.emmet.io>). Если у вас мало времени, начните с шаблонов, представленных в виде удобной шпаргалки.

1.6. Установка менеджера пакетов npm (с Node.js)

Задача

Установить npm, чтобы быстро загружать библиотеки JavaScript из реестра npm и подключать их к веб-проектам.

Решение

В Node Package Manager (npm) хранится самый большой в мире и самый популярный на сегодняшний день реестр программного обеспечения. Самый простой способ получить программу из npm-реестра — использовать утилиту npm, которая входит в комплект Node.js. Чтобы установить Node, загрузите с сайта Node (<https://nodejs.org>) установочный пакет для своей операционной системы (Windows, MacOS или Linux).

Закончив установку Node, можете проверить, доступен ли он из командной строки. Для этого откройте окно терминала и введите команду `node -v`. Чтобы

убедиться, что утилита `npm` тоже установилась, введите `npm -v`. В обоих случаях вы увидите версии этих пакетов:

```
$ node -v
v14.15.4
$ npm -v
6.14.10
```

Обсуждение

Утилита `npm` входит в состав Node.js — операционной среды JavaScript и веб-сервера. Node можно использовать для выполнения серверных фреймворков JavaScript, таких как Express, или для сборки автономных приложений на JavaScript с помощью Electron. Но даже если вы не собираетесь работать с Node, вам почти наверняка придется его установить хотя бы для того, чтобы получить доступ к менеджеру пакетов `npm`.

Node Package Manager — это инструмент, позволяющий загружать пакеты из реестра `npm`: бесплатного каталога, который отслеживает десятки тысяч библиотек JavaScript. Собственно, сегодня едва ли найдется компьютер, который используется для разработки на JavaScript и на котором при этом *не установлены* Node и `npm`.

Возможности менеджера пакетов далеко не ограничиваются простой загрузкой полезных библиотек. Менеджер пакетов также отслеживает, какие библиотеки использует проект (они называются *зависимостями*), загружает пакеты, от которых *в свою очередь* зависят эти библиотеки (эти пакеты иногда называют подзависимостями), сохраняет информацию о версиях, различает тестовые и эксплуатационные сборки. Благодаря `npm` можно полностью перенести приложение с одного компьютера и на другой и установить все необходимые зависимости одной командой, как описано в рецепте 1.7.

В настоящее время `npm` — самый популярный, но далеко не единственный менеджер пакетов для JavaScript. Некоторые разработчики предпочитают Yarn (<https://yarnpkg.com>), так как считают, что он быстрее устанавливает пакеты. Еще один вариант — Pnpm (<https://pnpm.io>). Он почти полностью совместим с командной строкой `npm`, но занимает гораздо меньше места и быстрее устанавливается.

Читайте также

О том, как установить пакет из `npm`, читайте в рецепте 1.7.

Если вы используете Node (а не только `npm`) для разработки, то, возможно, стоит установить его с `nvm` — менеджером версий Node. Тогда вы сможете легко переходить с одной версии Node на другую и быстро обновлять установленный Node по мере появления новых релизов, что происходит довольно часто. Подробнее об этом читайте в рецепте 17.1. Если же вы не знаете, как запустить на

выполнение код в среде Node, то загляните в главу 17, где найдете множество примеров.

Дополнительно: терминал и оболочка

Node и npm запускаются из *терминала*. Технически терминал — это текстовый интерфейс, который взаимодействует с *оболочкой*, чтобы выполнять команды. Существует множество программ, реализующих функцию терминала, а также множество оболочек. Выбор терминала и оболочки зависит от используемой операционной системы (и ваших личных предпочтений, поскольку есть множество сторонних альтернатив).

Вот некоторые из наиболее популярных сочетаний терминалов и оболочек, которые могут вам встретиться.

- На компьютере Macintosh перейдите в **Applications**, откройте папку **Utilities** и выберите **Terminal**. Запустится стандартная программа терминала, оболочкой которой служит **bash**.
- Какая программа терминала будет установлена на компьютере с Linux, зависит от дистрибутива. Обычно она называется **Terminal**, а в качестве оболочки почти всегда используется **bash**.
- На компьютере с Windows можно запустить PowerShell из меню Start. Технически PowerShell — это оболочка, заключенная в процесс терминала, который называется **conhost**. В настоящее время Microsoft разрабатывает современную замену **conhost** под названием **Windows Terminal**, которую первые пользователи уже могут установить с Windows Store или загрузить с GitHub (<https://github.com/microsoft/terminal>). Microsoft также включила оболочку **bash** в состав Windows Subsystem for Linux (<https://oreil.ly/N7EWS>), хотя это относительно недавнее дополнение к операционной системе.
- В некоторых редакторах кода есть свои встроенные терминалы. Например, если открыть окно терминала в VS Code, нажав комбинацию клавиш **Ctrl+`** (это не одиночная кавычка, а обратный апостроф) или выбрав из меню команду **View ► Terminal**, то откроется встроенный терминал VS Code. По умолчанию он взаимодействует с PowerShell в Windows и с **bash** в остальных операционных системах, но это можно изменить.

Когда мы будем предлагать вам ввести команду терминала, можете использовать окно терминала в редакторе кода, программу терминала, установленную на вашем компьютере, или же любое из многочисленных сторонних приложений терминала и оболочки. Все они принимают одни и те же переменные окружения (то есть имеют доступ к установленным на компьютере Node и npm) и позволяют запускать программы из текущей папки. Вы также можете с помощью терминала решать обычные задачи управления файловой системой, такие как создание папок и файлов.



В книге перед командами, которые нужно вводить в терминале (как в рецепте 1.6), ставится знак \$. Это традиционное приглашение ввода для `bash`. Но в других оболочках могут быть другие соглашения. Так, в PowerShell вместо знака \$ вы увидите имя папки, после которого стоит знак > (например, `C:\Projects\Sites\WebTest>`). Как бы то ни было, команды для запуска утилит, таких как `npm`, от этого не изменятся.

1.7. Загрузка пакета с помощью `npm`

Задача

Установить определенный программный пакет из реестра `npm`.

Решение

Прежде всего необходимо установить на компьютере утилиту `npm` (о том, как это сделать, см. рецепт 1.6). Если вы это сделали, откройте окно терминала (см. подраздел «Дополнительно: терминал и оболочка» ранее) и перейдите в папку проекта вашего сайта.

Затем создайте файл `package.json`, если у вашего приложения его еще нет. На самом деле этот файл не является необходимым для установки пакетов, но он вам понадобится для некоторых других задач, таких как восстановление пакетов при переносе разрабатываемого приложения на другой компьютер. Самый простой способ создать файл `package.json` — воспользоваться командой `npm init`:

```
$ npm init -y
```

Параметр `-y` (сокращенное от *yes*) означает, что `npm` не будет предлагать вам ввести различную информацию о приложении, а просто выберет стандартные значения. Утилита, запущенная без параметра `-y`, задаст множество вопросов о приложении: имя пакета, описание, версия, лицензия и т. п. Но вам поначалу не нужно указывать все эти подробности (и не факт, что когда-либо понадобится), так что можно спокойно нажимать `Enter`, чтобы оставить все эти поля пустыми и создать простейший шаблонный файл `package.json`. Подробнее об описаниях, которые вносятся в файл `package.json`, читайте в подразделе «Дополнительно: основные сведения о `package.json`» далее.

После инициализации приложения все готово к установке пакета. Нужно знать точное имя пакета, который вы хотите установить. В соответствии с соглашением имени пакетов `npm` состоят из слов, набранных строчными буквами и разделенных дефисами, например `fs-extra` или `react-dom`. Для того чтобы установить выбранный пакет, нужно выполнить команду `npm`, указав его имя. Например, для установки популярной библиотеки `Lodash` требуется выполнить следующую команду:

```
$ npm install Lodash
```

Утилита `npm` заносит имена установленных пакетов в файл `package.json`, а также записывает подробную информацию о версиях каждого пакета в файл `package-lock.json`.

При установке пакета `npm` загружает его файлы в папку `node_modules`. Например, при установке пакета `Lodash` в папку проекта `test-site` файлы сценариев будут размещены в папке `test-site/node_modules/Lodash`.

Для того чтобы удалить пакет, используется команда `npm uninstall`:

```
$ npm uninstall Lodash
```

Обсуждение

Вся гениальность `npm` и других менеджеров пакетов станет очевидной, когда вы начнете работать с типичными веб-проектами, насчитывающими полдесятка или более пакетов, каждый из которых зависит от других пакетов. Благодаря тому что все эти зависимости отслеживаются в файле `package-lock.json`, можно легко понять, что именно нужно данному веб-приложению. Для того чтобы получить полный отчет, требуется перейти в папку проекта и выполнить следующую команду:

```
$ npm list
```

Эти пакеты так же легко заново скачиваются при переносе проекта на новый компьютер. Например, если скопировать на другой компьютер веб-сайт с файлами `package.json` и `package-lock.json`, но без папки `node_modules`, то для установки всех зависимых пакетов нужно выполнить следующую команду:

```
$ npm install
```

До сих пор мы рассматривали *локальную* установку пакетов (в составе данного веб-приложения). Но `npm` также позволяет устанавливать пакеты *глобально* (в соответствующей системной папке, чтобы все приложения, установленные на компьютере, использовали одни и те же версии пакетов). Большинство программных пакетов предпочтительнее устанавливать локально. Это позволяет гибко выбирать версию пакета: можно задействовать разные версии одного и того же пакета для разных приложений, что гарантирует совместимость. (Эта потенциальная проблема становится еще серьезней, если один пакет зависит от определенной версии *другого* пакета.) Однако глобальная установка бывает полезной для определенных типов пакетов, в особенности для средств разработки, имеющих утилиты командной строки. В число пакетов, которые иногда устанавливаются глобально, входят `create-react-app` (используемый для создания проектов React), `http-server` (для запуска тестового веб-сервера), `typescript` (для компиляции кода TypeScript в JavaScript) и `jest` (для автоматического тестирования кода).

Для того чтобы увидеть список всех глобальных пакетов `npm`, установленных на компьютере, выполните следующую команду:

```
`npm list -g --depth 0`
```

Использование параметра `--depth` гарантирует, что мы увидим только глобальные пакеты верхнего уровня, а не все остальные применяемые ими пакеты. У `npm` есть еще несколько функций, которые мы не будем здесь рассматривать, в том числе следующие.

- Назначение определенных зависимостей *зависимостями разработки* — они используются при разработке, но не при развертывании продукта (например, инструмент модульного тестирования). Мы применим эту методику в рецептах 1.9 и 1.10.
- Ревизия используемых зависимостей путем поиска в реестре `npm` отчета об обнаруженных уязвимостях — чтобы устранить эти зависимости, нужно установить новые версии пакетов (<https://oreil.ly/XjkEM>).
- Выполнение операций, обычно запускаемых из командной строки, с помощью утилиты `npm`, которая поставляется в комплекте с `npm`. Эти операции даже можно выполнять автоматически, внося их в файл `package.json`. В число таких операций входит, в частности, подготовка веб-сайта к развертыванию в среде эксплуатации или запуск веб-сервера для тестирования в процессе разработки. Методику использования тестового сервера рассмотрим в рецепте 1.9.

Утилита `npm` — не единственный менеджер пакетов, применяемый разработчиками JavaScript. Есть еще аналогичный менеджер пакетов `Yarn`, первоначально разработанный компанией Facebook. В некоторых случаях он обеспечивает лучшую производительность, так как загружает пакеты параллельно и использует кэширование. Исторически сложилось так, что в нем применяются более строгие методы проверки безопасности. Нет никаких причин *отказываться* от `Yarn`, но в сообществе разработчиков JavaScript `npm` по-прежнему остается значительно более популярной утилитой.

Если захотите изучить все, что стоит знать об `npm`, потратьте некоторое время на чтение документации по ней для разработчиков (<https://docs.npmjs.com>). Можете также взглянуть на `Yarn` (<https://yarnpkg.com>).

Дополнительно: основные сведения о `package.json`

Файл `package.json` — это файл конфигурации приложения, который изначально создавался Node, но сейчас используется для достижения множества других целей. В нем хранятся описание проекта, имя автора, лицензия — все это станет важно, когда вы захотите опубликовать свой проект в виде пакета в реестре `npm` (о том, как это сделать, читайте в рецепте 18.2). Файл `package.json` позволяет также отслеживать зависимости (пакеты, применяемые приложением): в нем можно хранить дополнительные команды конфигурации, выполняемые при отладке и развертывании приложения.

Файл `package.json` рекомендуется создать сразу, в самом начале работы над проектом. Это можно сделать вручную или с помощью команды `npm init -y`, которую

мы использовали в примерах этой главы. Сгенерированный файл `package.json` выглядит так (если папка проекта называется `test_site`):

```
{
  "name": "test_site",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

Легко заметить, что в `package.json` используется формат JSON (JavaScript Object Notation). Этот файл состоит из заключенного в фигурные скобки списка параметров, разделенных запятыми. Файл `package.json` всегда можно изменить с помощью редактора кода.

При установке пакета с помощью `npm` эта зависимость заносится в файл `package.json` как значение свойства `dependencies`. Например, после установки пакета `Lodash` файл `package.json` будет выглядеть так:

```
{
  "name": "test_site",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "lodash": "^4.17.20"
  }
}
```

Не следует путать файлы `package.json` и `package-lock.json`. В `package.json` хранятся основные параметры проекта и список применяемых пакетов. В `package-lock.json` указываются версии и контрольные суммы для всех подключенных вами пакетов, а также версии и контрольные суммы всех используемых ими пакетов. Например, автоматически сгенерированный файл `package-lock.json` после установки пакета `Lodash` выглядит так:

```
{
  "name": "test-site",
  "version": "1.0.0",
  "lockfileVersion": 1,
```

```
"requires": true,
"dependencies": {
  "lodash": {
    "version": "4.17.20",
    "resolved": "https://registry.npmjs.org/Lodash/-/Lodash-4.17.20.tgz",
    "integrity": "sha512-PlhdFcill0INfeV7Ni6oF1TAEayyZBoZ8bcshTHqOYJY1r
qzRK5hagpagky5o4HfCzzd1TRkXPMFq6cKk9rGmA=="
  }
}
```

Другими словами, файл `package-lock.json` привязывает пакеты к определенной версии. Это пригодится при развертывании проекта на другом компьютере, если вы захотите установить там те же самые версии всех пакетов, ранее использованных при разработке.

Есть две причины, по которым чаще всего приходится изменять файл `package.json`. Во-первых, чтобы внести туда дополнительные описания, обеспечивающие полноту проекта, прежде чем передавать его кому-то другому. Если вы пожелаете распространять пакет через реестр npm (см. рецепт 18.2), то наверняка захотите гарантировать правильность этой информации. Во-вторых, чтобы внести в файл `package.json` операции командной строки, выполняемые при отладке приложения, такие как запуск тестового сервера (рецепт 1.9). Полное описание всех свойств, которые могут использоваться в файле `package.json`, вы найдете в документации по npm (<https://oreil.ly/n9PkO>).

1.8. Обновление пакета с помощью npm

Задача

Установить свежую версию npm-пакета.

Решение

Для незначительных обновлений можно взять команду `npm update`. В ней нужно указать имя пакета, который требуется обновить. Или же с помощью утилиты npm можно проверить наличие новых версий для *всех* пакетов, используемых сайтом, и обновить их все одним махом:

```
$ npm update
```

Утилита npm прочитает файл `package.json`, обновит все зависимости и подзависимости указанных в нем пакетов, загрузит все недостающие пакеты и внесет новые версии в файл `package-lock.json`.

Обсуждение

Рекомендуется регулярно обновлять используемые пакеты. Однако не все обновления выполняются автоматически. Утилита `npm` выполняет обновления по правилам *семантической верификации* (semantic versioning, `semver`). Она устанавливает обновления, которые имеют больший номер патча (например, 2.1.3 вместо 2.1.2) или дополнительный номер версии (2.2.0 вместо 2.1.2), но не обновляет зависимость в том случае, если у нового релиза изменился основной номер версии (3.0.0 вместо 2.1.2). Благодаря такому поведению работа приложения не нарушится при обновлении или развертывании.

Чтобы просмотреть доступные обновления для всех применяемых зависимостей, можно воспользоваться командой `npm outdated`:

```
$ npm outdated
```

Результат будет выглядеть примерно так:

Package	Current	Wanted	Latest	Location
-----	-----	-----	-----	-----
eslint	7.18.0	7.25.0	7.25.0	my-site
eslint-plugin-promise	4.2.1	4.3.1	5.1.0	my-site
Lodash	4.17.20	4.17.21	4.17.21	npm-test

В столбце **Wanted** показаны доступные обновления, которые будут установлены при следующем выполнении команды `npm update`. В столбце **Latest** представлены последние версии пакетов. В данном примере пакеты `Lodash` и `eslint` могут быть обновлены до последних версий, а пакет `eslint-plugin-promise` — только до версии 4.3.1. Последняя версия этого пакета — 5.1.0, то есть изменился главный номер версии, поэтому в соответствии с правилами семантической верификации он не может быть обновлен автоматически.



Это немного упрощенное представление. Утилита `npm` позволяет настраивать политику управления версиями с помощью файла `package.json`. Но на практике режим обновлений, предлагаемый в `npm` по умолчанию, почти всегда работает хорошо. Подробнее об управлении версиями в `npm` читайте в документации по `npm` (<https://oreil.ly/NX8js>).

Обновлять зависимости до новой главной версии необходимо специально. Для этого нужно либо вручную внести изменения в файл `package.json` (что несколько хлопотно), либо воспользоваться инструментом, который сделает это за вас, таким как `npm-check-updates` (<https://oreil.ly/0JcMt>). Утилита `npm-check-updates` позволяет просмотреть список зависимостей, найти те, для которых есть обновления, и внести изменения в файл `package.json`, чтобы разрешить установку новых главных версий для этих пакетов. Затем, чтобы установить новые версии, нужно вызвать команду `npm update`.

1.9. Настройка локального тестового сервера

Задача

Иметь возможность тестировать веб-страницы в процессе разработки без локальных ограничений в сфере безопасности и без развертывания на рабочем веб-сервере.

Решение

Установите на компьютере локальный тестовый сервер. Он будет обрабатывать запросы и передавать веб-страницы в браузер так же, как это сделал бы обычный веб-сервер. Единственное отличие состоит в том, что тестовый сервер не будет устанавливать удаленные соединения с другими компьютерами.

Есть множество вариантов тестовых серверов (см. раздел «Обсуждение»). Из них два самых простых и надежных — это пакеты `http-server` и `lite-server`, которые устанавливаются посредством `npm`. В этой книге мы будем использовать пакет `lite-server`, поскольку у него есть функция автоматического обновления, которая обновляет страницы в браузере сразу после сохранения измененного в редакторе кода.

Прежде чем устанавливать пакет `lite-server`, стоит создать простую веб-страницу, чтобы было на чем проверить его работу. Если у вас еще нет такой страницы, создайте папку проекта и настройте его конфигурацию с помощью команды `npm init -y` (см. рецепт 1.7). Затем создайте файл `index.html` с простейшим содержанием. Если вы торопитесь, то вот минимальный корректный HTML-документ, с помощью которого можно проверить, работает ли ваш код:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Test Page</title>
  </head>
  <body>
    <p>This is the index page</p>
    <script>
if (window.location.protocol === 'file:') {
  console.log('Running as local file!');
}
else if (window.location.host.startsWith('localhost')) {
  console.log('Running on a local server');
}
else {
  console.log('Running on a remote web server');
}
    </script>
  </body>
</html>
```

Теперь можно сделать так, чтобы этот документ открывался в браузере с помощью тестового сервера.

Для установки `lite-server` нужно вызвать команду `npm` с параметром `--save-dev`. Таким образом мы указываем, что данная зависимость является *зависимостью разработки* и не должна разворачиваться при сборке готового продукта:

```
npm install lite-server --save-dev
```

Теперь запустим `lite-server` непосредственно из окна терминала с помощью утилиты запуска пакетов `npx`:

```
npx lite-server
```

Эта утилита запустит `lite-server`, откроет новую вкладку в браузере и выполнит запрос `http://localhost:3000`, где `3000` — порт, который динамически назначается `lite-server`. Далее `lite-server` попытается вернуть `index.html` либо, если не найдет файл с таким именем, выведет сообщение `Cannot GET /`. Если вы использовали образец страницы, представленный в этом разделе, то на ней появится текст `This is the index page`, а в консоли разработки — сообщение `Running on a local server`. Если на вашем тестовом сайте нет страницы `index.html`, можете загрузить другую страницу, изменив URL в строке адреса (например, `http://localhost:3000/someOtherPage.html`).

Теперь давайте что-нибудь изменим. Экземпляр `lite-server` отслеживает папку проекта. Когда вы вносите изменения в файл, сервер автоматически обновляет соответствующую страницу в браузере. В терминале при этом каждый раз выводится сообщение `Reloading Browsers`.

Чтобы остановить сервер, перейдите в терминал, нажмите `Ctrl+C` (на Macintosh — `Command-C`) и введите `Y` в ответ на последующий вопрос. Или же закройте окно терминала (в VS Code можно воспользоваться значком `Kill Terminal` с изображением корзины для мусора).



Внутри `lite-server` применяется популярный инструмент автоматизации браузера `BrowserSync` (<https://oreil.ly/tAwyk>) — именно он обеспечивает автоматическую перезагрузку страниц. Для этого на веб-странице обязательно должен быть раздел `<body>` (Создайте простейшую тестовую страницу без этого элемента — и увидите, что автоматического обновления не происходит.)

Обсуждение

Мы и без тестового сервера можем сохранить веб-страницу на локальном компьютере, открыть ее в веб-браузере и выполнить размещенный на ней код. Но веб-браузеры сильно ограничивают возможности страниц, открываемых из локальной файловой системы. Целые функции, такие как веб-процессы, ES-модули и некоторые операции с `Canvas`, окажутся недоступными и не будут работать, никак об этом не сообщая. Чтобы не наткнуться на эти барьеры системы безопасности

или, что еще хуже, не понимать, почему приложение не работает так, как нужно, стоит всегда открывать веб-страницы с тестового веб-сервера.

При тестировании обычно используется сервер разработки. Таких серверов много, и ваш выбор до определенной степени будет зависеть от серверных технологий, которые вы планируете задействовать. Например, если на страницах будет выполняться код РНР, то вам понадобится сервер, поддерживающий этот язык. Если планируете собрать серверную часть своего приложения с помощью JavaScript или серверных фреймворков на базе JavaScript, таких как Express, то нужен Node.js. Но для обычных веб-страниц с клиентским JavaScript подойдет простой сервер, способный передавать статические файлы, например `http-server` или `lite-server`. Таких серверов очень много, и у многих редакторов кода есть собственные плагины тестовых серверов. Например, в библиотеке расширений для Visual Studio Code вы найдете популярный плагин Live Server (<https://oreil.ly/NlrRK>).

В разделе «Решение» было показано, как запустить `lite-server` с помощью команды `prx`. Но удобнее создать *задачу разработки*, которая будет запускать сервер автоматически. Для этого нужно открыть файл `package.json` и добавить в раздел `scripts` следующее:

```
{
  ...
  "scripts": {
    "dev": "lite-server"
  }
}
```

В разделе `scripts` содержатся задачи, которые должны выполняться регулярно. Это может быть проверка кода синтаксическим анализатором и системой управления версиями, упаковка файлов для развертывания или выполнение модульного тестирования. Этих задач может быть сколько угодно. Например, типичный набор — одна задача для запуска приложения, еще одна — для автоматизированного тестирования с помощью соответствующего инструмента (см. рецепт 10.7), еще одна — для подготовки к развертыванию и т. д. В нашем примере сценарий называется `dev`, что в соответствии с соглашением означает задачу, которая выполняется при разработке приложения.

После того как сценарий занесен в файл `package.json`, его можно выполнить в терминале с помощью команды `npm run`:

```
npm run dev
```

В результате будет вызвана утилита `prx`, с помощью которой будет запущен `lite-server`.

Некоторые редакторы кода поддерживают такую возможность дополнительной настройки. Например, если открыть файл `package.json` в VS Code, то непосредственно над параметром `dev` появится ссылка `Debug`. Если щелкнуть на ней,

то в VS Code откроется новое окно терминала, в котором автоматически запустится `lite-server`.

Читайте также

Для получения более подробной информации об использовании Node в качестве тестового сервера читайте рецепты из главы 17. Подробнее о выполнении задач с помощью `npm` можно узнать из обзора, размещенного по адресу <https://oreil.ly/nq31H>.

1.10. Соблюдение стандартов кодирования с помощью статического анализатора

Задача

Стандартизировать код JavaScript, обеспечить соблюдение рекомендаций и избегать типичных подводных камней, которые приводят к появлению ошибок.

Решение

Проверьте код с помощью *статического анализатора* (linter). Он предупредит вас, если вы отступите от правил, которым решили следовать. Самым популярным статическим анализатором JavaScript является ESLint.

Для того чтобы использовать ESLint, его вначале нужно установить с помощью `npm` (см. рецепт 1.6). Перейдите в папку проекта и откройте окно терминала. Если вы еще не создали файл `package.json`, то сделайте это с помощью следующей команды `npm`:

```
$ npm init -y
```

Затем установите пакет `eslint` с параметром `--save-dev` — нам нужно, чтобы ESLint был зависимостью разработчика и устанавливался только на компьютер, на котором разрабатывается проект, но не на рабочий сервер:

```
$ npm install eslint --save-dev
```

Если у вас еще нет конфигурационного файла ESLint, то нужно его создать. Для настройки ESLint используется следующая команда `npm`:

```
$ npx eslint --init
```

ESLint задаст несколько вопросов, чтобы определить тип правил, за соблюдением которых нужно следить. Часто предлагается меню с вариантами выбора, между которыми можно переключаться при помощи клавиш со стрелками.

Первый вопрос: How would you like to use ESLint? (Как вы хотите использовать ESLint?). На него есть три варианта ответа, от менее жесткого до самого жесткого.

- *Только проверка синтаксиса.* В режиме **Check syntax only** ESLint просто находит ошибки и не делает ничего более серьезного, чем обычная функция проверки ошибок, которая есть в большинстве редакторов кода.
- *Проверка синтаксиса и поиск проблем.* В режиме **Check syntax and find problems** активируются рекомендованные правила ESLint (<https://eslint.org/docs/rules>) — те, что отмечены флажками. Этот режим — отличная начальная точка, впоследствии при желании вы сможете переопределить некоторые из этих правил.
- *Проверка синтаксиса, поиск проблем, проверка стиля программирования.* Режим **Check syntax, find problems, and enforce code style** хорошо подходит для тех случаев, когда нужно использовать определенный стиль программирования на JavaScript, например стандарт Airbnb (<https://github.com/airbnb/javascript>), чтобы обеспечить соблюдение дополнительных соглашений по стилю программирования. Если выбрать этот вариант, то вам будет предложено указать соответствующее руководство по оформлению кода.

За этим последует ряд технических вопросов: используете ли вы модули, фреймворк React или Vue либо язык TypeScript? Если нужна поддержка стандартов модулей ES6 (они описываются в рецепте 8.9), то выберите **JavaScript modules**. В ответ на остальные вопросы выберите вариант **No**, если только не применяете технологию, указанную в вопросе.

Затем появится вопрос: Where does your code run? (Где выполняется ваш код?). Выберите вариант **Browser** в случае традиционного веб-сайта с кодом JavaScript, выполняемым на стороне клиента (как обычно), или **Node**, если разрабатываете серверное приложение, которое выполняется на сервере Node.js.

Если вы решите использовать руководство по оформлению кода, то ESLint предложит небольшой список вариантов, и нужно будет выбрать один из них. Затем будут автоматически установлены соответствующие правила с применением одного или нескольких отдельных пакетов — при условии что вы разрешите их установку.

В конце ESLint задаст вопрос: What format do you want your config file to be in? (В каком формате должен быть представлен файл конфигурации?). Все предлагаемые варианты формата подходят одинаково хорошо. Мы предпочитаем использовать JSON из соображений симметрии с файлом `package.json`, и тогда ESLint сохранит свою конфигурацию в файле `.eslintrc.json`. Если вы выберете конфигурацию в формате JavaScript, то расширением файла будет `.js`, а в случае конфигурационного файла в формате YAML — `.yaml`.

Если выбрать для ESLint режим **Check syntax and find problems** без подключения дополнительного руководства по оформлению кода, то файл `.eslintrc.json` будет выглядеть так:

```
{
  "env": {
    "browser": true,
    "es2021": true
  },
  "extends": "eslint:recommended",
  "parserOptions": {
    "ecmaVersion": 12,
    "sourceType": "module"
  },
  "rules": {
  }
}
```

Теперь можно запустить ESLint в терминале, чтобы проверить файлы проекта:

```
npx eslint my-script.js
```

Но будет гораздо практичнее использовать плагин, который встраивает ESLint в редактор кода. Все редакторы кода, описанные в рецепте 1.1, поддерживают ESLint, а полный список плагинов для поддержки ESLint вы найдете по адресу <https://oreil.ly/isQMA>.

Для того чтобы подключить ESLint к редактору кода, нужно перейти в библиотеку плагинов этого редактора. Например, в Visual Studio Code для этого нужно щелкнуть на надписи **Extensions**, расположенной на левой панели, найти в библиотеке плагин **eslint** и нажать кнопку **Install**. После того как ESLint будет установлен, следует явно активировать его на странице параметров плагинов (или щелкнуть на значке в виде лампочки, который появится, когда вы откроете в редакторе файл с кодом, и затем выбрать вариант **Allow**). Иногда имеет смысл установить ESLint глобально, чтобы данный инструмент был доступен для всех проектов на этом компьютере и плагин всегда мог его найти:

```
$ npm install -g eslint
```

После того как ESLint будет активирован, вы увидите, что некоторые части кода подчеркнуты волнистой линией. Так ESLint обозначает ошибки и предупреждения. На рис. 1.4 показан пример того, что ESLint обнаружил проблему с оператором **switch**: выполнение программы «проваливается» к следующему **case**, чего по умолчанию ESLint не допускает. Метка **eslint** во всплывающем окне означает, что данное сообщение исходит от плагина ESLint, а не от стандартной системы проверки ошибок VS Code.



Если ESLint не обнаруживает проблем, которые, по идее, должен обнаруживать, возможно, дело в том, что в файле есть другая ошибка — вероятно, даже в другом разделе кода. Попробуйте устранить все остальные проблемы и проверить файл снова.

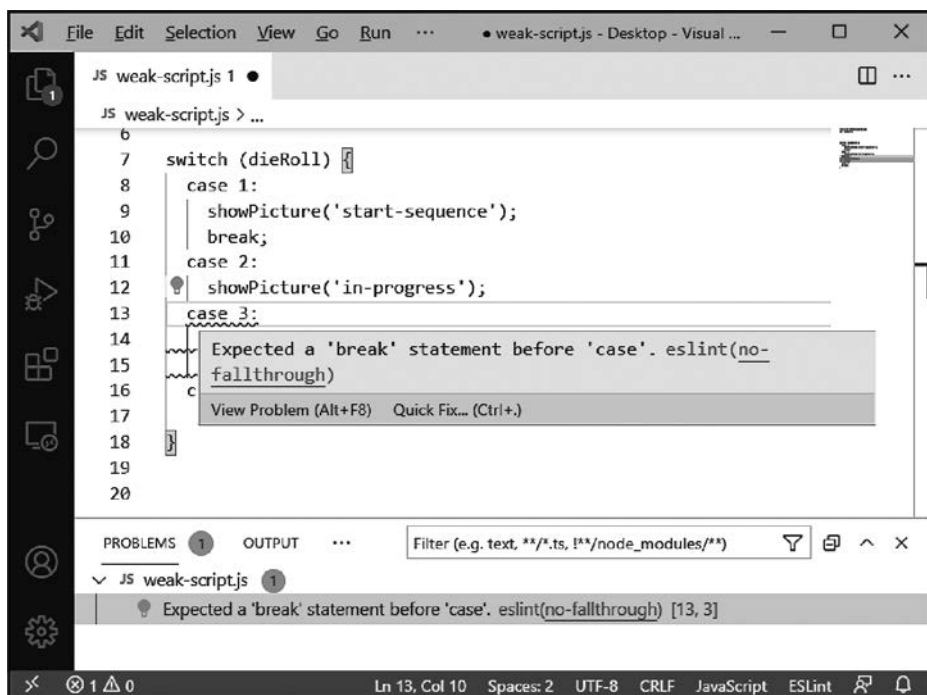


Рис. 1.4. ESLint обнаружил ошибку в VS Code

Для того чтобы узнать больше о проблеме или попытаться ее исправить, если это возможно, щелкните на надписи **Quick Fix** или на значке в виде лампочки на рамке окна. Можно также отключить проверку этой проблемы в данной строке или в данном файле, и тогда исправление будет записано в виде отдельного комментария. Например, следующий комментарий отменяет запрет на объявление переменных, которые потом не используются:

```
/* eslint-disable no-unused-vars */
```

Если уж пришлось переопределять правила ESLint посредством комментариев, постарайтесь сделать это максимально точно и рационально. Вместо того чтобы отменять проверку всего файла, переопределите правило только для отдельной строки, например, так:

```
// eslint-disable-next-line no-unused-vars
let futureUseVariable;
```

или так (заменяв `eslint-disable-next-line` на `eslint-disable-line`):

```
let futureUseVariable;    // eslint-disable-line no-unused-vars
```


Если захотите восстановить проверку данной проблемы, просто удалите комментарий.

Обсуждение

JavaScript — весьма снисходительный язык, обеспечивающий значительную гибкость для разработчиков. Но иногда это вызывает проблемы, например язык скрывает ошибки или допускает неясности, из-за которых код становится трудно понимать. Назначение статического анализатора — предотвращать подобные проблемы, требуя соблюдения ряда стандартов, даже если их нарушение не приводит к прямым ошибкам. Статический анализатор указывает на потенциальные проблемы на этапе написания кода и на подозрительные приемы, которые, хоть и не фиксируются при проверке ошибок в редакторе кода, впоследствии могут вызвать неприятности.

ESLint — *весьма нетерпимый* анализатор: он отмечает даже те проблемы, которые вы можете и не считать таковыми. В число таких проблем входят объявленные, но не используемые переменные; значения параметров, изменяемые внутри функции; пустые блоки условий; регулярные выражения, включающие в себя символы пробелов (и это лишь несколько примеров). Если вы захотите допустить некоторые из этих приемов, то у вас есть возможность переопределить соответствующие параметры в файле конфигурации ESLint или в комментариях к каждой соответствующей строке в каждом файле. Но скорее всего, вы просто измените свой стиль разработки, согласившись с тем, что выбор ESLint позволит избежать проблем в будущем.

ESLint также позволяет автоматически исправлять некоторые виды ошибок и обеспечивать соблюдение стиля оформления, заменяя табуляции пробелами, одиночные кавычки — двойными, поддерживая стили использования скобок, отступов и т. п. Подключив плагин ESLint к редактору кода, такому как VS Code, можно затем настроить этот плагин на автоматическое внесение данных изменений при сохранении файла. Либо можно использовать ESLint только для выявления потенциальных проблем, а для выполнения соглашений об оформлении кода задействовать форматировщик (см. рецепт 1.11).

Если вы работаете в команде, то вам могут просто предоставить готовый файл конфигурации ESLint. Если же нет, то придется самостоятельно решать, какого варианта стандартных правил ESLint придерживаться. Подробнее о рекомендованном наборе правил ESLint (использованном в этом рецепте) читайте здесь: <https://eslint.org/docs/rules>. В этом документе для каждого условия, проверяемого ESLint, приводятся примеры недопустимого кода. Если же вы захотите применить более строгое руководство по оформлению кода JavaScript, советуем популярный Airbnb JavaScript Style Guide (<https://github.com/airbnb/javascript>), который устанавливается автоматически с помощью команды `eslint -init`.

1.11. Согласованное оформление кода с помощью форматировщика

Задача

Соблюдение единого формата для всего кода JavaScript, чтобы сделать его более удобным для чтения и исключить неясности.

Решение

Для автоматического форматирования кода в соответствии с установленными правилами воспользуйтесь форматировщиком кода Prettier. Он обеспечивает согласованность таких деталей оформления, как отступы, одинарные и двойные кавычки, пробелы внутри скобок и между параметрами функций, а также перенос длинных строк кода. Но, в отличие от статического анализатора (см. рецепт 1.10), Prettier не отмечает эти проблемы, чтобы вы их исправили. Вместо этого он применяет правила форматирования автоматически при каждом сохранении кода JavaScript, HTML-страницы или таблицы стилей CSS.

Prettier существует в виде пакета, его можно установить с помощью npm и затем использовать из командной строки. Но гораздо удобнее применять Prettier в виде плагина, подключенного к редактору кода. Плагины Prettier существуют для всех редакторов кода, упомянутых в рецепте 1.1. Большинство этих плагинов размещено на сайте Prettier (<https://oreil.ly/weRb5>).

Для того чтобы подключить Prettier к редактору кода, перейдите в библиотеку плагинов. Например, в Visual Studio Code для этого нужно нажать кнопку **Extensions**, расположенную на левой панели, найти в библиотеке плагин **prettier** и нажать кнопку **Install**.

Установив Prettier, вы сможете с его помощью редактировать файлы с кодом. Для этого щелкните правой кнопкой мыши в окне редактора рядом с кодом и выберите команду **Format Document**. Можно настроить свойства плагина, изменив несколько его параметров, таких как максимально допустимая ширина неразделяемой строки кода и выбор символов для отступов — пробелы или табуляция.



В VS Code можно также настроить автоматический запуск Prettier при каждом сохранении файла. Для активации такого поведения выберите команду **File** ▶ **Preference** ▶ **Settings**, перейдите к **Text Editor** ▶ **Formatting section** и выберите **Format On Save**.

Обсуждение

Во многих редакторах кода есть встроенные средства автоматического форматирования, но у отдельного форматировщика кода гораздо больше возможностей.

Например, форматировщик Prettier удаляет любое нестандартное форматирование. Он анализирует код и переформатирует его в соответствии с заданными соглашениями, почти не обращая внимания на то, как он был написан изначально. (Из этого правила есть всего два исключения: пустые строки и литералы объектов.) Такой подход гарантирует, что одинаковый код всегда будет представлен одним и тем же способом и код, написанный разными разработчиками, окажется полностью согласованным. Как и для статического анализатора, правила для форматировщика кода определяются в файле конфигурации, а следовательно, их можно легко передать всем членам команды, даже если они используют разные редакторы кода.

Форматировщик Prettier обращает особое внимание на разрывы строк. По умолчанию максимальная длина строки — 80 символов, но Prettier позволяет делать некоторые строки немного длиннее — в том случае, если бы разрыв строки привел к путанице. Если же разрыв строки необходим, то Prettier делает это разумно. Например, он старается, чтобы вызов функции целиком помещался в одной строке:

```
myFunction(argA(), argB(), argC());
```

Но если это нецелесообразно, Prettier не просто разорвет код где попало, а выберет наиболее красивое, по его мнению, размещение строк:

```
myFunction(  
  reallyLongArg(),  
  omgSoManyParameters(),  
  IShouldRefactorThis(),  
  isThereSeriouslyAnotherOne()  
);
```

Разумеется, каким бы разумным ни был форматировщик наподобие Prettier, у вас могут быть собственные специфические правила оформления кода. Как говорится, «никому не нравится то, что Prettier делает с его кодом, но всем нравится то, что он делает с кодом его коллег». Другими словами, ценность столь агрессивного и бескомпромиссного форматировщика, как Prettier, состоит в том, что он унифицирует код, написанный разными людьми, чистит код, написанный вашими предшественниками, и сглаживает странные привычки кодирования. Если вы решите использовать Prettier, то получите неограниченную свободу писать код, не задумываясь о расстановке пробелов, разрывах строк или общем виде кода. В итоге он все равно будет приведен к одному и тому же каноническому виду.



Если вы еще не определились, хотите ли задействовать форматировщик кода, или не знаете, как именно настроить его параметры, потратьте некоторое время на эксперименты в интерактивной среде Prettier (<https://oreil.ly/TKam1>), чтобы понять, как это работает.

Функции статических анализаторов, таких как ESLint, и форматировщиков, таких как Prettier, отчасти перекрываются. Но назначение этих утилит различно,

они дополняют друг друга. Если вы решите использовать и ESLint, и Prettier, то вам стоит оставить те правила ESLint, которые выявляют сомнительные стили программирования, но отключить те, которые применяют соглашения о форматировании, такие как правила расстановки отступов, кавычек и пробелов. К счастью, это легко сделать, внеся в ESLint дополнительное правило конфигурации, отменяющее параметры настройки, которые могут вступать в конфликт с правилами Prettier. Самый простой способ сделать это — подключить к проекту пакет `eslint-config-prettier`:

```
$ npm install --save-dev eslint-config-prettier
```

И наконец, нужно внести пакет `prettier` в раздел `extends` файла `.eslintrc.json`. В итоге раздел `extends` будет представлять собой список, заключенный в квадратные скобки, в конце которого появится `prettier`, например:

```
{
  "env": {
    "browser": true,
    "es2021": true
  },
  "extends": ["eslint:recommended", "prettier"],
  "parserOptions": {
    "ecmaVersion": 12,
    "sourceType": "module"
  },
  "rules": {
  }
}
```

Чтобы ознакомиться с самыми свежими инструкциями по установке пакета `eslint-config-prettier`, загляните в его документацию (<https://oreil.ly/AgxiF>).

1.12. Эксперименты в интерактивной среде JavaScript

Задача

Быстро протестировать возможность кодирования или поделиться этой идеей с другими разработчиками, не создавая проект и не запуская автономный редактор кода.

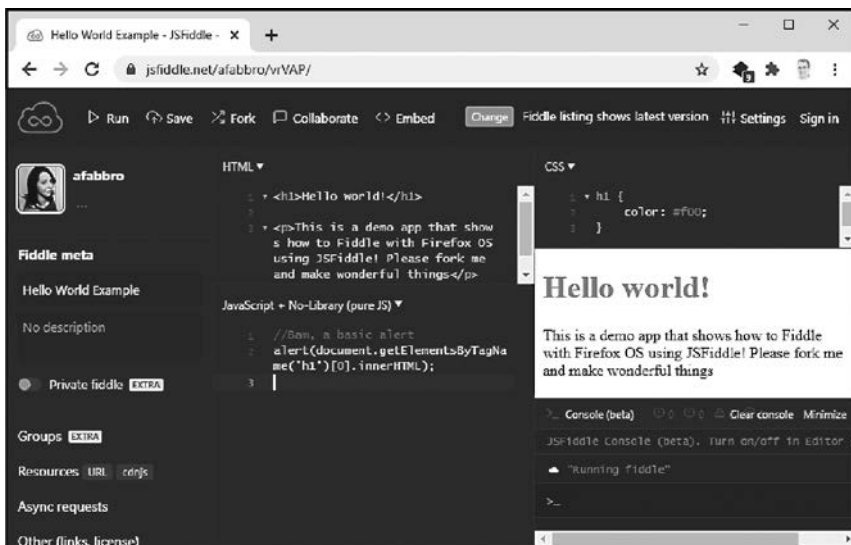
Решение

Нужно воспользоваться *интерактивной средой* JavaScript. Это сайт, на котором можно редактировать и запускать код JavaScript. Таких интерактивных сред JavaScript существует больше десятка, но в табл. 1.4 приведены пять самых популярных из них.

Таблица 1.4. Интерактивные среды JavaScript

Веб-сайт	Примечания
JSFiddle (https://jsfiddle.net)	Несмотря на то что JSFiddle — едва ли не первая интерактивная среда JavaScript, она по-прежнему сохраняет первенство, обеспечивая симуляцию асинхронных вызовов и интеграцию с GitHub
JS Bin (https://jsbin.com)	Классическая интерактивная среда с простым интерфейсом, состоящим из отдельных вкладок для JavaScript, HTML и CSS, которые можно просматривать поочередно. JS Bin доступна также в виде проекта с открытым кодом
CodePen (https://codepen.io)	Одна из наиболее привлекательно сконструированных интерактивных сред, в которой особое внимание уделено сообществу (самые популярные примеры предлагаются в сообществе CodePen). Ее отшлифованный интерфейс особенно оценят начинающие пользователи
CodeSandbox (http://codesandbox.io)	Одна из самых новых интерактивных сред. Применяет IDE-подобную структуру и очень похожа на онлайн-версию Visual Studio Code
Glitch (https://glitch.com)	Еще одна «IDE в браузере». Интерактивная среда Glitch известна своим плагином для VS Code, который позволяет переключаться между редактированием одного и того же проекта в интерактивной среде в окне браузера и в автономном редакторе

Все эти интерактивные среды JavaScript довольно удобны и функциональны. Все они работают приблизительно одинаково, хотя и выглядят на удивление по-разному. Например, сравним компактный кабинет разработчика в JSFiddle (рис. 1.5) и более свободный редактор в CodePen (рис. 1.6).

**Рис. 1.5.** Интерактивная среда JavaScript в JSFiddle

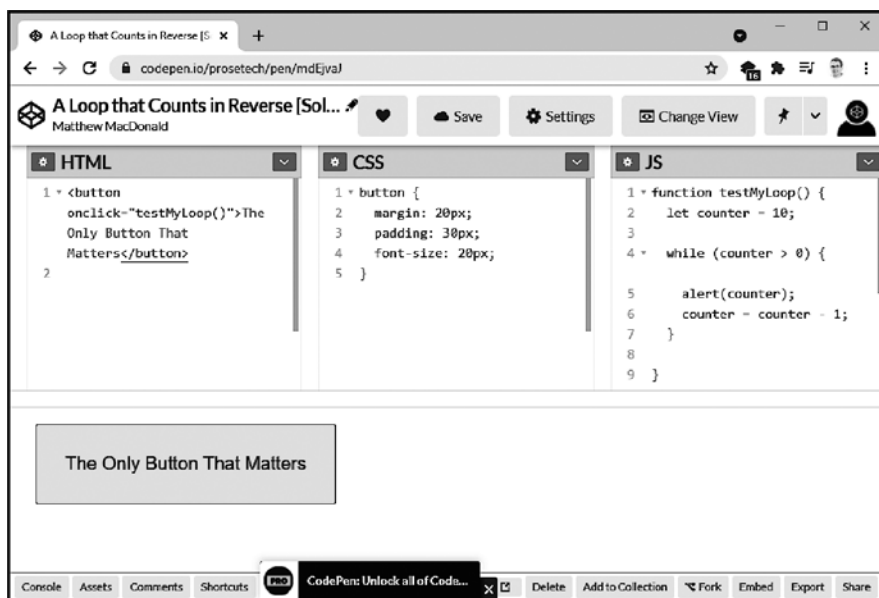


Рис. 1.6. То же самое в CodePen

Интерактивную среду JavaScript используют следующим образом. Открыв сайт, можно сразу начать писать код на чистой странице. Несмотря на то что код JavaScript, HTML и CSS находится в отдельных окнах, вам не нужно явно добавлять элемент `<script>`, чтобы подключить JavaScript, или `<link>`, чтобы установить связь с CSS. Эти детали уже вставлены в разметку страницы или, что более вероятно, являются неявной частью скрытого шаблона.

Любая интерактивная среда JavaScript позволяет увидеть страницу, над которой вы работаете, рядом с окном кода. Иногда (как в CodePen) окно предварительного просмотра обновляется автоматически, когда вы вносите изменения. В других случаях (как в JSFiddle), чтобы перезагрузить страницу, нужно нажать кнопку **Play** или **Run**. Сообщения, которые выводятся с помощью `console.log()`, в некоторых интерактивных средах JavaScript будут появляться непосредственно в консоли браузера (как в CodePen), а в других — на специальной панели, отображаемой на странице (как в JSFiddle).

Закончив, можете сохранить свою работу. Для этого вы получите специально сгенерированную ссылку, которой можно делиться с другими разработчиками. Но лучше сначала создать учетную запись, чтобы можно было возвращаться к этой среде разработки JavaScript, хранить в ней все образцы кода и редактировать их. (Если сохранить образец кода анонимно, то его потом нельзя будет изменить, хотя можно будет использовать как основу для создания другого образца кода.) Все интерактивные среды, перечисленные в табл. 1.4, позволяют бесплатно создавать учетные записи и сохранять результаты работы.



Точное название образцов кода, которые создаются в интерактивной среде JavaScript, зависит от конкретного сайта. Это может быть `fiddle` (поделка), `rep` (набросок), `snippet` (фрагмент) или еще что-то.

Обсуждение

Интерактивная среда JavaScript — полезная идея, которую подхватили более десятка веб-сайтов. Почти все они имеют следующие общие характеристики.

- Бесплатность. Однако на многих таких сайтах можно подписаться на платные функции, например возможность сохранить работу, но запретить публичный доступ к ней.
- Возможность сохранять работу сколько угодно раз. Это особенно удобно, если вы хотите быстро создать макет с общим доступом или провести совместный эксперимент с другими разработчиками.
- Поддержка множества популярных библиотек и фреймворков JavaScript. Например, к создаваемым примерам можно быстро подключить `Lodash`, `React` или `jQuery`, просто выбрав соответствующий вариант из списка.
- Возможность редактировать HTML, JavaScript и CSS в одном окне. В зависимости от конкретной интерактивной среды оно может быть разделено на несколько одновременно видимых панелей (как в `JSFiddle`) или между этими панелями нужно переключаться (как в `JS Bin`). Либо можно выбирать между разными вариантами (как в `CodePen`).
- Наличие базовых функций автодополнения, проверки ошибок и подсветки синтаксиса (выделение разных элементов кода разными цветами) — пускай и не в том объеме, как в автономных редакторах кода.
- Предварительный просмотр страницы, так что можно легко переключаться между написанием кода и тестированием.

Впрочем, у интерактивных сред JavaScript есть свои ограничения. Например, в них нельзя размещать другие ресурсы, такие как изображения, нельзя взаимодействовать с серверными службами, такими как базы данных, или использовать асинхронные запросы посредством `fetch`.

Следует также отличать интерактивные среды JavaScript от полноценных облачных сред программирования. Например, с `VS Code` можно работать онлайн, в полностью размещенной в облаке среде `GitHub Codespaces` (<https://oreil.ly/Vo95d>), или `AWS Cloud9` от Amazon (<https://oreil.ly/tvTZq>), или `Google Cloud` (<https://oreil.ly/fqWuW>). Ни один из этих продуктов не бесплатен, но все они весьма привлекательны, если вы захотите сформировать специфическую среду разработки, которой можно будет пользоваться через браузер на разных устройствах и не испытывать проблем с настройкой и производительностью.

ГЛАВА 2

Строки и регулярные выражения

Продолжая нашу JavaScript-вечеринку, зададим простой вопрос: сколько типов данных поддерживает этот самый популярный в мире язык программирования?

Правильный ответ — *восемь*. Но, возможно, не те, что вы ожидали. Вот какие восемь типов данных существуют в JavaScript:

- `Number`;
- `String`;
- `Boolean`;
- `BigInt` (для очень больших целых чисел);
- `Symbol` (для уникальных идентификаторов);
- `Object` (корневой тип данных для всех остальных типов данных JavaScript);
- `undefined` (переменная, которой не присвоено значение);
- `null` (отсутствующий объект).

Все рецепты в книге готовятся только из этих ингредиентов. В этой главе мы рассмотрим возможности обработки текста, предоставляемые строками.

2.1. Проверка того, что строка существует и она не пустая

Задача

Перед использованием переменной убедиться, что она определена, является строкой и непустая.

Решение

Прежде чем начинать работу со строкой, часто бывает необходимо убедиться, что применять ее безопасно. Для этого нужно ответить на ряд вопросов.

Чтобы убедиться, что данная переменная является строкой (а не просто переменной, которую можно *преобразовать* в строку), нужно выполнить следующую проверку:

```
if (typeof unknownVariable === 'string') {  
    // unknownVariable – это строка  
}
```

Для того чтобы убедиться, что строка непустая (не строка вида '', длина которой равна нулю), нужно уточнить нашу проверку:

```
if (typeof unknownVariable === 'string' && unknownVariable.length > 0) {  
    // Это именно строка, и она содержит какие-то символы либо пробелы  
}
```

Для того чтобы отсеять строки, состоящие только из пробелов, можно воспользоваться методом `String.trim()`:

```
if (typeof unknownVariable === 'string' && unknownVariable.trim().length > 0) {  
    // Это именно строка, она непустая и состоит не только из пробелов  
}
```

Порядок проверки условий имеет значение, так как в JavaScript используется *сокращенная оценка логических выражений*. Это значит, что второе условие (проверка длины) вычисляется только в том случае, если первое условие (проверка типа) истинно. Это важно, поскольку, если `unknownVariable` является переменной другого типа, например числом, проверка длины закончится неудачей:

```
// Этот тест безопасен только в том случае, если мы уже знаем,  
// что unknownVariable является строкой  
if (unknownVariable.length > 0)
```

У оператора `typeof` есть один потенциальный недостаток. Проверку на строковый тип можно обойти, если вместо строкового литерала будет использован объект `String`:

```
const unknownVariable = new String('test');
```

Теперь оператор `typeof` будет возвращать не `string`, а `object`, так как примитив типа `string` обернут в объект `String`.

Именно по причинам, подобным этой, в современном JavaScript не рекомендуется создавать экземпляры объекта `String`. Вместо того чтобы как-то учитывать этот код, лучше исключить подобную практику в любом коде, с которым вы имеете дело. Но если необходимо допустить возможность использования в коде объектов `String`, проверку нужно усложнить:

```
if (typeof unknownVariable === 'string' ||  
    String.prototype.isPrototypeOf(unknownVariable)) {  
    // Это строковый примитив или строка, обернутая в объект  
}
```

В этом коде проверяется выполнение одного из двух условий: является ли переменная строковым примитивом или же объектом, имеющим тот же прототип, что и `String`¹.

Обсуждение

При проверке типов в этом рецепте используется оператор `typeof`. Он возвращает имя типа переменной в виде строки, состоящей только из строчных букв. Возможны следующие значения:

- `undefined`;
- `boolean`;
- `number`;
- `bigint`;
- `string`;
- `symbol`;
- `function`;
- `object`.

Эти значения соответствуют списку, приведенному в начале главы, но с двумя незначительными различиями. Во-первых, здесь отсутствует `null`, поскольку вместо строки `null` такие значения возвращают строку `object`. (Это часто считают ошибкой языка, но так сложилось по историческим причинам.) Во-вторых, здесь появился еще тип данных `function`, хотя технически функция — это особый вид объекта.

Время от времени вам будет попадаться следующий старомодный способ валидации строк. Он не требует, чтобы переменная действительно была строкой, а лишь проверяет, что значение может трактоваться как строка и она не будет пустой:

```
if (unknownVariable) {  
    /* Мы попадем сюда, если:  
       unknownVariable объявлена  
       unknownVariable не равна null  
       unknownVariable не является пустой строкой (='')  
    */  
}
```

Это работает, поскольку значения `null` и `undefined`, а также пустые строки (``) в JavaScript считаются ложными. При вычислении любого из них в логическом выражении они принимают значение `false`.

¹ В JavaScript прототипом называется шаблон для определенного типа объектов. В более традиционных объектно-ориентированных языках объекты с одним и тем же прототипом принято называть экземплярами одного класса. В главе 8 вы найдете много рецептов, в которых раскрываются возможности прототипов JavaScript.

У этого приема есть потенциальное «слепое пятно» — число 0. В логических выражениях оно всегда равно `false`, из-за чего блок `if` не будет выполнен. Для гарантии лучше явно преобразовать числовые значения в строковые, как показано в рецепте 2.2.

2.2. Преобразование числового значения в форматированную строку

Задача

Создать строковое представление числа.

Решение

JavaScript — свободно типизированный язык: при необходимости он автоматически преобразует любое значение в строку — например, при сравнении числа и строки или присоединении числа к строке с помощью оператора `+`. В сущности, одна из простейших уловок, с помощью которой разработчики JavaScript преобразуют число в строку, — присоединить пустую строку к числу, поставив ее перед числовым значением или после него:

```
const someNumber = 42;
const someString = someNumber + '';
```

Однако современный стиль программирования предпочитает явные преобразования типов переменных. У всех объектов JavaScript, включая `Number`, есть встроенный метод `toString()`. Он вызывается так:

```
const someNumber = 42;
const someString = someNumber.toString();
```

Часто возникает необходимость определить строковое представление числа — например, с фиксированным количеством знаков после точки (30.00 вместо 30). Для этого некоторые числа приходится округлять, например, с 30.009 до 30.01.

Для этого в JavaScript есть три вспомогательных метода, встроенных в числовой тип данных. Все они создают строковое представление числа.

- `Number.toFixed()` — позволяет задать число цифр после точки.
- `Number.toExponential()` — представляет числа в экспоненциальном формате, позволяет задать количество цифр после десятичной точки.
- `Number.toPrecision()` — позволяет задать количество значащих цифр независимо от того, насколько велико или мало число.



На случай, если вы не знакомы с понятием значащих цифр, — это научное понятие, используемое для того, чтобы гарантировать, что вычисления будут вестись с заданной точностью. Значащие цифры также позволяют гарантировать, что результаты измерений будут представлены с точностью, не превышающей возможности измерительных приборов. (Например, ваш средний вес может составлять 76,5 килограмма, но, пожалуй, бессмысленно утверждать, что он равен 76,503018 килограмма, также не имеет смысла округлять его до 80 килограммов.) Подробнее об этой концепции можно почитать в «Википедии» (<https://oreil.ly/vrrPr>).

Вот пример, демонстрирующий все три метода преобразования строк:

```
const someNumber = 1242.0055;

// Требование ровно двух цифр после точки.
// При необходимости числа будут округляться.
const fixedString = someNumber.toFixed(2);
// fixedString = '1242.01'

// Требование пяти значащих цифр. При необходимости
// будет использован экспоненциальный формат.
const precisionString = someNumber.toPrecision(5);
// precisionString = '1242.0'

// Требование представления чисел в экспоненциальном формате
// с двумя цифрами после десятичной точки.
const scientificString = someNumber.toExponential(2);
// scientificString = '1.24e+3'
```

Для того чтобы использовать такие элементы форматирования, как точки, знак валюты или детали, определяемые региональным стандартом, необходимо задействовать объект `Intl.NumberFormat`. Создав экземпляр этого объекта с соответствующей конфигурацией, можно использовать `Intl.NumberFormat` для преобразования чисел в строки.

Например, для того чтобы представить число в виде денежной суммы в валюте США, нужно написать такой код:

```
const formatter =
  new Intl.NumberFormat('en-US', { style: 'currency', currency: 'USD' });

const someNumber = 1242.0005;
const moneyString = formatter.format(someNumber);
// moneyString = '$1,242.00'
```

Обсуждение

Региональный стандарт (локаль) представляет определенный географический или культурный регион. Локаль — это сочетание кода языка и строки региона. Так, локаль *en-US* соответствует английскому языку в США, локаль *en-CA* — английскому в Канаде, *fr-CA* — французскому в Канаде, *ja-JP* — японскому в Японии и т. д.

Есть несколько стандартных правил представления чисел, применяемых в зависимости от локали. Например, в англоговорящих странах для разделения рядов в числах часто используются запятые (например: 1,200.00), в то время как в странах, где говорят по-французски, запятые обычно ставят вместо десятичной точки (1200,00). Если создать объект `Intl.NumberFormat`, не передавая в конструктор никаких аргументов, то будут применены параметры локали, установленные на данном компьютере:

```
const formatter = new Intl.NumberFormat();
```

Можно также создать объект `Intl.NumberFormat` для выбранной локали без возможности ее изменения:

```
const formatter = new Intl.NumberFormat('en-US');
```

Для региона *en-US* будет создан объект с разделителями в виде запятых, но в нем не будет зафиксировано количество цифр после запятой и не будет добавлен символ валюты.

Объект `Intl.NumberFormat` поддерживает ряд параметров. Можно изменить способ отображения отрицательных чисел, задать минимальное и максимальное количество цифр, вывод знака процента и выбрать одну из систем отображения чисел, принятых в разных языках. Исчерпывающую информацию об этом вы найдете в справочном руководстве Mozilla Developer Network (<https://oreil.ly/JEF4Q>).

Возможно, вам встречался такой прием с использованием метода `Number.toLocaleString()`:

```
const someNumber = 1242.0005;  
const moneyString = someNumber.toLocaleString(  
  'en-US', { style: 'currency', currency: 'USD' });
```

Это вполне действенный метод, но если нужно представить в определенном формате большое количество чисел, то лучше создать и многократно применить один объект `Intl.NumberFormat` — это обеспечит более высокую производительность.

Читайте также

Если нужны дополнительные параметры форматирования, которые не поддерживаются `Intl.NumberFormat`, то можно воспользоваться сторонней библиотекой, такой как `N numeral.js` (<https://github.com/adamwdraper/Numeral-js>).

2.3. Вставка специальных символов

Задача

Вставить в строку специальный символ, такой как разрыв строки.

Решение

В случае многих специальных символов простейшее решение выглядит элементарно: просто вставьте нужный символ в редакторе кода. Например, если нужен символ авторского права (©), найдите его в системной утилите, такой как `charmap` (для Windows), или в Google по запросу `copyright symbol`. Выделите символ, скопируйте его и вставьте в код.

Для того чтобы вставить символ, который обычно недопустим в коде (согласно синтаксическим правилам JavaScript), нужно воспользоваться одной из экранирующих последовательностей (или `escape`-последовательностей) — специальных кодовых комбинаций символов, которые не интерпретируются буквально.

Например, если строки заключаются в апострофы, то мы не можем вставить символ апострофа непосредственно в строку. Вместо этого нужно применить экранирующую последовательность `\'`:

```
const favoriteMovie = 'My favorite movie is \'The Seventh Seal\'';
```

Теперь в `favoriteMovie` содержится текст *My favorite movie is 'The Seventh Seal'*.

Обсуждение

Все экранирующие последовательности в JavaScript начинаются с *обратного слеша* (`\`). Этот символ означает, что дальше идет последовательность символов, которая нуждается в специальной обработке. В табл. 2.1 перечислены остальные экранирующие последовательности, распознаваемые в JavaScript.

Последние три экранирующие последовательности в табл. 2.1 — это шаблоны, в которые нужно вставить число. Например, если вы не хотите использовать копирование и вставку для символа авторского права, то можете поместить его в код, используя экранирующую последовательность `\u` и указав в ней значение Unicode для символа авторского права:

```
const copyrightNotice = 'This page \u00A9 Shelley Powers.';
```

Теперь в строке `copyrightNotice` содержится текст *This page © Shelley Powers*.

Таблица 2.1. Экранирующие последовательности

Последовательность	Символ
<code>\'</code>	Одинарная кавычка
<code>\"</code>	Двойная кавычка
<code>\\</code>	Обратный слеш
<code>\n</code>	Новая строка
<code>\t</code>	Горизонтальная табуляция

Последовательность	Символ
<code>\b</code>	Возврат без удаления*
<code>\f</code>	Перевод страницы*
<code>\r</code>	Возврат каретки*
<code>\ddd</code>	Восьмеричное число (трехзначное — <code>ddd</code>)
<code>\xdd</code>	Шестнадцатеричное число (двузначное — <code>dd</code>)
<code>\udddd</code>	Unicode-последовательность (четырёхзначное шестнадцатеричное число — <code>dddd</code>)

* Некоторые экранирующие последовательности, такие как возврат на один символ и перевод страницы, — это пережитки первоначального стандарта символов ASCII и языка C. Если только вы не имеете дела с устаревшими задачами, такими как получение вводимых данных с терминала, эти экранирующие последовательности едва ли будут полезны в JavaScript.

Читайте также

Подробнее о вставке в строку более специализированных символов читайте в рецепте 2.4. Альтернативный способ работы с разрывами строк без `\n` описан в рецепте 2.5.

2.4. Использование эмодзи

Задача

Вставить из расширенного набора Unicode символ с четырехзначным кодом, такой как эмодзи или некоторые буквы, не входящие в английский алфавит.

Решение

Если вы всего лишь хотите создать строку с эмодзи, то, как правило, вполне достаточно приема с копированием и вставкой, описанного в рецепте 2.3. В современных редакторах кода можно написать код, подобный следующему:

```
const hamburger = ' 🍔 ';
const hamburgerStory = 'I like hamburgers' + hamburger;
```

Шрифт, которым вы набираете код, не обязан поддерживать эмодзи — для их вывода редактор кода будет обращаться к возможностям операционной системы. (Конечно, проблемы все равно возможны. Например, в более старых операционных системах, где нет поддержки эмодзи, вы можете увидеть пиктограмму в виде квадрата, обозначающую отсутствующий символ.)

Еще один вариант — использовать для эмодзи значение Unicode. Проблема состоит в том, что для эмодзи нельзя применять стандартную экранирующую последовательность, начинающуюся с `\u`, так как все они представлены в виде четырехбайтных значений. (Для сравнения: коды символов Unicode, соответствующих клавишам на клавиатуре, обычно занимают 2 байта.)

Решение проблемы состоит в использовании метода `String.fromCodePoint()`:

```
const hamburgerStory = 'I like hamburgers' + String.fromCodePoint(0x1F354);
```

Шестнадцатеричный код эмодзи в виде гамбургера — `U+1F354`. Чтобы задействовать его в `fromCodePoint()`, нужно заменить префикс `U+` на `0x`.

После того как вы создадите строку с эмодзи, ее, как и обычную строку, состоящую из обычных символов, можно вывести в консоли разработки или показать на веб-странице.

Обсуждение

К 2020 году в мире насчитывалось чуть больше 3000 эмодзи. Все они вместе с соответствующими шестнадцатеричными кодами представлены в списке *Full Emoji List* (<https://oreil.ly/IIGuA>). Но один лишь факт существования эмодзи не гарантирует, что он будет отображаться на тех устройствах, на которых вы планируете его использовать, поэтому необходимо сразу проверять совместимость.

При обработке строк, содержащих эмодзи, могут вылезти и другие подводные камни. Например, как вы думаете, каким будет результат выполнения этого кода?

```
const hamburger = ' 🍔 ';  
const hamburgerLength = hamburger.length;
```

Даже если считать, что строка `hamburger` состоит всего лишь из одного символа, с точки зрения нашего кода длина этой строки равна 2, так как эмодзи «гамбургер» занимает в памяти вдвое больше места, чем обычный символ. К сожалению, это «дырявая» абстракция (<https://oreil.ly/nlmvi>), ограничивающая поддержку Unicode в JavaScript.

Для решения возникающих в ходе работы с эмодзи проблем, таких как некорректный размер строки, посимвольная обработка и разбиение строк, были придуманы обходные пути. Но любые доморощенные решения несут в себе риски, поскольку часто порождают странные побочные эффекты. Вместо этого при манипуляциях с текстом, содержащим эмодзи, стоит применить библиотеку JavaScript для поддержки эмодзи, такую как *Grapheme Splitter* (<https://github.com/orling/grapheme-splitter>).

2.5. Использование шаблонных литералов для более наглядной конкатенации строк

Задача

Найти более простой и ясный способ писать длинные операции конкатенации строк.

Решение

Одна из типичных задач в программировании — объединение статического текста и переменных в одну длинную строку. Обычно для сборки такой строки используется оператор конкатенации `+`:

```
const employeeDetail = 'Our team includes ' + firstName + ' ' + lastName +  
  ' who works on the ' + team + " team. They've been a team member since "  
  + hireDate + '!';
```

Это не выглядит особенно ужасно, но все может измениться, когда дело дойдет до поиска ошибок или если текст станет длиннее. Кроме того, в такой записи необыкновенно легко забыть поставить пробелы перед переменными и после них.

Есть и другое решение — использовать *шаблонные литералы* — тип строковых литералов, допускающих встроенные выражения. Для того чтобы создать шаблонный литерал, просто замените обычные ограничители строк (апострофы или двойные кавычки) на наклоненные влево кавычки (```):

```
const greeting = `Hello world from a template literal!`;
```

Теперь можно вставлять переменные прямо в шаблонный литерал. Для этого нужно лишь заключить каждую переменную в фигурные скобки и поставить впереди знак доллара, например: `${firstName}`. Такая запись называется *выражением*.

Преимущество шаблонных литералов станет понятнее, если посмотреть на пример целиком:

```
employeeDetail = `Our team includes ${firstName} ${lastName} who works on the  
${team} team. They've been a team member since ${hireDate}!`;
```

А если использовать современный редактор кода, в котором подсвечиваются выражения с фигурными скобками, так что переменные выделяются на фоне остального текста, то все станет еще нагляднее.

Шаблонные литералы позволяют также сохранять разрывы строк. В показанных примерах это незаметно, так как мы разорвали строку кода, чтобы она поместилась

на странице. Но если вы специально нажмете **Enter**, чтобы вставить разрыв строки в шаблонный литерал, то этот разрыв сохранится при ее отображении, как если бы вы поставили в этом месте экранирующую последовательность `\n`, означающую переход на новую строку (см. рецепт 2.3).



Во многих руководствах по стилю программирования на JavaScript, включая Airbnb (<https://github.com/airbnb/javascript>), встречаются рекомендации не использовать конкатенацию строк, заменяя ее шаблонными литералами. Для того чтобы внедрить эту практику в своем коде, можете задействовать статический анализатор, такой как ESLint (рецепт 1.10).

Обсуждение

Используя выражения в шаблонных литералах, мы не ограничены только переменными. В сущности, это могут быть любые выражения, которые возможно вычислить в JavaScript. Например, рассмотрим следующий код:

```
const calculation = `The sum of 5 + 3 is ${5+3}`;
```

Здесь JavaScript выполняет суммирование `{5+3}`, получает результат и создает строку *The sum of 5 + 3 is 8*.

Для того чтобы сделать что-то более сложное, такое как форматирование строк или манипуляции с объектами, можно использовать выражения с вызовом функций. Например, если ранее вы создали функцию `getDaysSince()` для вычисления разности дат (см. рецепт 4.5), то можете задействовать ее в шаблонном литерале:

```
function getDaysSince(date) {
  const today = new Date();
  const oneDay = 24 * 60 * 60 * 1000; // часы*минуты*секунды*миллисекунды
  return Math.round(Math.abs((today - date) / oneDay));
}
```

```
employeeDetail = `Our team includes ${firstName} ${lastName}. They've been
a team member since ${hireDate}! That's ${getDaysSince(hireDate)} days.`;
```

Единственное ограничение здесь — практичность, другими словами, не стоит делать выражения слишком сложными, чтобы полученный в результате шаблонный литерал не оказался более сложным для понимания, чем код с традиционной конкатенацией строк.

В настоящее время для выражений, используемых в шаблонных литералах JavaScript, не существует встроенных способов форматирования чисел, дат и валют. Многие предполагают, что в будущих версиях JavaScript такая возможность появится. Чтобы это исправить, есть даже библиотека JavaScript, в которой реализовано неудобное расширение, названное *теговыми шаблонами* (<https://github.com/skolmer/es2015-i18n-tag>).

2.6. Сравнение строк без учета регистра

Задача

Определить, являются ли строки одинаковыми, если считать, что прописные и строчные буквы — это одно и то же.

Решение

Простой способ решения этой задачи состоит в том, чтобы применить к обеим строкам метод `String.toLowerCase()` и сравнить результат:

```
const a = "hello";
const b = "HELLO";

if (a.toLowerCase() === b.toLowerCase()) {
  // Мы здесь, так как после замены всех букв
  // на строчные получились одинаковые строки
}
```

Этот метод вполне надежен, но у него возможны проблемы в пограничных случаях для разных языков, в которых есть буквы с диакритическими знаками и специальные символы (например, вот одна из таких потенциальных проблем с турецким языком — <https://oreil.ly/CiALB>).

Другой способ состоит в использовании метода `String.localeCompare()` с параметром `sensitivity`, которому присвоено значение `accent`:

```
const a = "hello";
const b = "HELLO";

if (a.localeCompare(b, undefined, { sensitivity: 'accent' }) === 0) {
  // Мы попадаем сюда, потому что без учета
  // регистра эти строки одинаковые
}
```

Обсуждение

Если метод `localeCompare()` решает, что две строки одинаковы, то возвращает 0. В противном случае он возвращает положительное либо отрицательное целое число, показывающее, стоит ли вторая строка перед первой или после нее при сортировке. (Поскольку мы применяем `localeCompare()` в тесте на идентичность, порядок сортировки не имеет значения и его можно игнорировать.)

Второй параметр метода `localeCompare()` представляет собой строку, определяющую локаль (о которой говорилось в рецепте 2.2). Если передать значение `undefined`, то `localeCompare()` будет использовать локаль текущего компьютера — почти всегда это именно то, чего мы хотели.

Для сравнения с учетом регистра нужно указать значение свойства `sensitivity`. Здесь нам пригодятся два значения: если присвоить `sensitivity` значение *accent*, то буквы с диакритическими знаками и без них, такие как *a* и *á*, будут различаться. Но если присвоить `sensitivity` значение *base*, то получим более либеральное сравнение не только без учета регистра, но и без учета диакритических знаков.

2.7. Проверка того, содержит ли строка заданную подстроку

Задача

Проверить, содержит ли строка другую строку.

Решение

Если нам нужна простая проверка типа «да — нет», то можно воспользоваться методом `String.includes()`:

```
const searchString = 'infinitely';
const fullText = 'I know not where I was born, save that the castle was' +
  ' infinitely old and infinitely horrible.';

if (fullText.includes(searchString)) {
  // Подстрока найдена
}
```

При желании можно указать методу `includes()` позицию символа, с которого нужно начинать поиск. Например, если передать в метод значение 5, то будут пропущены первые пять символов, поиск начнется с шестого и продолжится до конца строки:

```
const searchString = 'infinitely';
const fullText = 'I know not where I was born, save that the castle was' +
  ' infinitely old and infinitely horrible.';

if (fullText.includes(searchString, 70)) {
  // Метод все равно возвращает true: он пропускает
  // первое слово 'infinitely', но находит второе.
}
```

Обсуждение

Поиск, выполняемый методом `includes()`, ведется с учетом регистра. Для поиска без учета регистра нужно вначале применить к обеим строкам метод `toLowerCase()`:

```
const searchString = 'INFINITELY';
const fullText = 'I know not where I was born, save that the castle was' +
```

```
    ' infinitely old and infinitely horrible.';

if (fullText.toLowerCase().includes(searchString.toLowerCase())) {
    // Подстрока найдена
}
```

Метод `includes()` не предоставляет информацию о том, где именно найдено совпадение. Для получения этой информации можно вместо `includes()` использовать метод `String.indexOf()`, описанный в рецепте 2.11.

2.8. Замена всех вхождений строки

Задача

Найти в строке все вхождения заданной подстроки и заменить их на что-то другое.

Решение

Для того чтобы изменить все сразу, можно использовать метод `String.replaceAll()`. Нужно лишь указать подстроку для поиска и другую строку, которую нужно поставить на место искомой:

```
const storyText = 'I know not where I was born, save that the castle was' +
    ' infinitely old and infinitely horrible.';

const changedStory = storyText.replaceAll('infinitely', 'somewhat');

console.log(changedStory);
```

Если выполнить этот код, то в консоли разработки появится измененная строка: `I know not where I was born, save that the castle was somewhat old and somewhat horrible1.`

Обсуждение

Метод `replaceAll()` позволяет использовать для поиска не только обычные строки, но и регулярные выражения. В рецепте 2.10 показано, как это работает.

Читайте также

В рецептах 2.11 и 2.12 показано, как найти совпадения в строке и, вместо того чтобы сразу их заменить, сначала проверить каждое из них.

¹ Цитата из рассказа «Изгой» Г. Ф. Лавкрафта. — *Примеч. пер.*

2.9. Замена тегов HTML на именованные сущности

Задача

Разместить на веб-странице фрагмент HTML-кода, для чего понадобится экранировать разметку (чтобы браузер *вывел* угловые скобки, а не пытался интерпретировать их как теги HTML). Это бывает нужно, например, чтобы вывести кусок HTML-кода в учебной статье. Или для надежной очистки внешних данных, таких как текст, переданный пользователем, который затем должен быть помещен в базу данных.

Решение

С помощью метода `String.replaceAll()` преобразовать угловые скобки (`<` `>`) в именованные сущности HTML `<` и `>`. Это делается в два этапа, по одному для каждой замены:

```
const originalPieceOfHtml = '<p>This is a <span>paragraph</span></p>';
// Получаем строку без символов <
let safePieceOfHtml = originalPieceOfHtml.replaceAll('<', '&lt;');
// Получаем строку без символов >
safePieceOfHtml = safePieceOfHtml.replaceAll('>', '&gt;');
// Выводим строку на странице
document.getElementById('placeholder').innerHTML = safePieceOfHtml;
```

Если теперь проверим полученную строку, то обнаружим, что она содержит текст `<p>This is a paragraph</p>`, который выводится на веб-странице именно так, как мы хотели, — с угловыми скобками.

Можно выполнить обе подстановки за один шаг, при этом сохранив читаемость кода:

```
const safePieceOfHtml =
  originalPieceOfHtml.replaceAll('<', '&lt;').replaceAll('>', '&gt;');
```

Первый вызов `replaceAll()` возвращает новую строку, для которой выполняется следующий вызов `replaceAll()`, и эта третья строка присваивается переменной. Такой способ вызова метода для значения, возвращаемого другим методом, называется *цеплением методов*.

Обсуждение

Экранировать HTML-теги критически важно, когда нужно разместить на веб-странице необработанный текст. Если этого не сделать, то в безопасности приложения появится зияющая дыра. В сущности, необходимо обеспечить экранирование любого текста, прежде чем он попадет на веб-страницу, даже если в нем, по

идее, не должно быть HTML-сущностей (например, это просто литерал, жестко заданный в коде). Нет никакой гарантии, что кто-нибудь однажды не изменит этот код и не поставит вместо этого текста что-нибудь другое.

Однако выполнять HTML-экранирование своими силами — не лучший вариант. Это приходится делать, если вы специально создаете строку, в которой HTML-теги комбинируются с внешним содержимым. Но в идеале, размещая текст на веб-странице, следует вместо свойства элемента `innerHTML` задействовать свойство `textContent`. При использовании `textContent` браузер экранирует содержимое автоматически, так что нет необходимости применять `String.replaceAll()`.

Читайте также

Подробнее о применении HTML Document Object Model (DOM) при размещении текстового содержимого на веб-странице читайте в главе 12.

2.10. Использование регулярных выражений для создания шаблонов при замене строк

Задача

Найти в строке не точную последовательность символов, а фрагмент, соответствующий заданному шаблону. Затем создать новую строку, заменив этот фрагмент.

Решение

Применить метод `String.replace()` или `String.replaceAll()` — оба они поддерживают регулярные выражения.



Регулярное выражение — это последовательность символов, которая определяет текстовый шаблон. Регулярные выражения — это стандарт, который внедрен в JavaScript и многие другие языки программирования. В табл. 2.2 содержится краткое введение в синтаксис регулярных выражений.

Рассмотрим, например, шаблон регулярного выражения `t\w{2}e`. Он расшифровывается так: «Найти все последовательности символов, которые начинаются с `t`, заканчиваются на `e` и содержат еще две буквы или цифры». Этому условию соответствует и *time*, и *tame*.

Вот код, в котором используется регулярное выражение:

```
const originalString = 'Now is the time, this is the tame';
const regex = /t\w{2}e/g;
const newString = originalString.replaceAll(regex, 'place');

// newString = 'Now is the place, this is the place'
```

Обратите внимание на то, что регулярное выражение — это не строка. Это литерал, который начинается и заканчивается косой чертой (/). JavaScript распознает этот синтаксис и создает объект `RegExp`, который применяет это регулярное выражение.

Буква `g` в конце регулярного выражения — это дополнительный элемент, называемый *глобальным флагом*. Он показывает, что нужно искать совпадения с шаблоном во всей строке. Если не поставить флаг `g`, то при вызове `replaceAll()` получим ошибку. При вызове метода `replace()` можно использовать регулярное выражение без глобального флага, но тогда будет заменен только первый фрагмент, соответствующий шаблону.

Обсуждение

Если вы предпочитаете писать регулярные выражения, не применяя разделитель `/`, то есть другой вариант. Вместо того чтобы создавать литерал регулярного выражения, можно явно создать объект `RegExp`:

```
const regex = new RegExp('t\\w{2}e', 'g');
const newString = originalString.replaceAll(regex, 'place');
```

Здесь не нужно ставить регулярное выражение между двумя косыми, но необходимо экранировать все обратные косые черты, находящиеся внутри шаблона, заменяя `\` на `\\`. Кроме того, глобальный флаг не ставится в конце регулярного выражения, а становится вторым аргументом конструктора `RegExp`. Если вам покажется, что в длинных и сложных регулярных выражениях экранирование косых выглядит неуклюже или сбивает с толку, то это требование экранирования можно обойти, воспользовавшись шаблонным литералом, описанным в рецепте 2.5. Хитрость состоит в сочетании шаблонного литерала и метода `String.raw()`. Напомним, что вместо апострофов или кавычек строковое выражение должно быть заключено в обратные кавычки (```):

```
// String.raw является методом, но после него не ставятся скобки
// Здесь использован специальный синтаксис с обратными кавычками
const regex = new RegExp(String.raw`t\\w{2}e`, 'g');
```

Дополнительно: регулярные выражения

Регулярные выражения состоят из обычных символов, которые применяются сами по себе или в сочетании со специальными символами. Например, так выглядит регулярное выражение для шаблона, соответствующего строке, которая содержит слова *technology* и *book*, именно в этом порядке, разделенные одним или несколькими пробельными символами:

```
const regex = /technology\s+book/;
```

Символ обратной косой черты (`\`) имеет двойное назначение: он может использоваться либо в сочетании с обычным символом, чтобы показать, что это специальный символ, или со специальным символом, таким как знак «плюс» (`+`),

чтобы показать, что этот символ должен трактоваться буквально. В данном случае обратный слеш применяется с буквой *s*, так что буква *s* преобразуется в специальный символ, обозначающий любой пробельный знак — пробел, табуляцию, новую строку или новую страницу. Знак *+*, стоящий после `\s`, означает, что символ, стоящий перед знаком «плюс» (в данном случае `\s`), может повторяться один или несколько раз. Этому регулярному выражению соответствует строка

```
technology book
```

Следующая строка тоже подходит:

```
technology    book
```

А эта строка не годится, так как между словами нет пробельных символов:

```
technologybook
```

Поскольку мы использовали `\s+`, не имеет значения, сколько пробелов стоит между словами *technology* и *book*. Но знак «плюс» требует, чтобы там был как минимум один пробел.

В табл. 2.2 показаны специальные символы, наиболее часто встречающиеся в приложениях JavaScript.

Таблица 2.2. Специальные символы, используемые в регулярных выражениях

Символ	Означает	Пример
<code>^</code>	Фрагмент в начале строки	<code>/^This/</code> найдет <i>This is...</i>
<code>\$</code>	Фрагмент в конце строки	<code>/end\$/</code> найдет <i>This is the end</i>
<code>*</code>	Фрагмент, повторяющийся ноль и более раз	<code>/se*/</code> найдет <i>seeee</i> и <i>se</i>
<code>?</code>	Фрагмент, повторяющийся ноль или один раз	<code>/ap?/</code> найдет <i>apple</i> и <i>and</i>
<code>+</code>	Фрагмент, повторяющийся не менее одного раза	<code>/ap+/</code> найдет <i>apple</i> , но не <i>and</i>
<code>{n}</code>	Фрагмент, повторяющийся ровно <i>n</i> раз	<code>/ap{2}/</code> найдет <i>apple</i> , но не <i>apie</i>
<code>\{n,\}</code>	Фрагмент, повторяющийся не менее <i>n</i> раз	<code>/ap{2,}/</code> найдет все буквы <i>p</i> в словах <i>apple</i> и <i>apple</i> , но не в <i>apie</i>
<code>\{n,m\}</code>	Фрагмент, повторяющийся не менее <i>n</i> , но не более <i>m</i> раз	<code>/ap{2,4}/</code> найдет четыре буквы <i>p</i> в слове <i>apple</i>
<code>.</code>	Любой символ, кроме новой строки	<code>/a.e/</code> найдет <i>ape</i> и <i>axe</i>
<code>[...]</code>	Любой символ из перечисленных в квадратных скобках	<code>/a[px]e/</code> найдет <i>ape</i> и <i>axe</i> , но не <i>ale</i>

Таблица 2.2 (Окончание)

Символ	Означает	Пример
[^ ...]	Любой символ, кроме перечисленных в квадратных скобках	/a[^rx]/ найдет ale, но не ахе и не аре
\b	Фрагмент в начале слова	/\bno/ найдет первое no в слове попо
\B	Фрагмент в начале того, что не является словом	/\Bno/ найдет второе no в слове попо
\d	Цифры от 0 до 9	/\d{3}/ найдет 123 в Now in 123
\D	Любой нецифровой символ	/\D{2,4}/ найдет 'Now в 'Now in 123;
\w	Символ, допустимый в слове (буква, цифра, знак подчеркивания)	/\w/ найдет j в javascript
\W	Символ, не допустимый в слове (любой, кроме букв, цифр и знаков подчеркивания)	/\W/ найдет % в 100%
\n	Перенос строки	
\s	Одиночный пробел	
\S	Один символ, не являющийся пробелом	
\t	Табуляция	
(x)	Захват выражения	Запоминает подходящие символы



Регулярные выражения — мощное, но временами каверзное средство. В нашей книге мы лишь слегка коснемся этой темы. Если вы хотите более глубоко изучить регулярные выражения, советуем почитать отличную книгу *Regular Expressions Cookbook* Яна Гойвартса (Jan Goyvaerts) и Стивена Левитана (Steven Levithan) (O'Reilly) или свериться с онлайн-руководством (<https://github.com/ziishaned/learn-regex>).

2.11. Извлечение списка из строки

Задача

Есть строка, состоящая из нескольких предложений. В одном из них есть список элементов. Он начинается с двоеточия (:) и заканчивается точкой (.), а элементы разделены запятыми. Мы хотим извлечь из строки только этот список.

Дано:

```
This is a list of items: cherries, limes, oranges, apples.
```

Нужно получить:

```
['cherries', 'limes', 'oranges', 'apples']
```

Решение

Эта задача решается в два действия: сначала нужно выделить из строки перечисление элементов, а затем преобразовать эту подстроку в список.

Для этого мы дважды воспользуемся методом `String.indexOf()`: сначала найдем двоеточие, а затем первую точку после двоеточия:

```
const sentence = 'This is one sentence. This is a sentence with a list of items: ' +  
'cherries, oranges, apples, bananas. That was the list of items.';
```

```
const start = sentence.indexOf(':');  
const end = sentence.indexOf('.', start + 1);
```

Зная эти две позиции, применим метод `String.slice()`, чтобы выделить нужную подстроку:

```
const list = sentence.slice(start + 1, end);  
// list = 'cherries, oranges, apples, bananas'
```

Далее мы могли бы написать цикл, в котором с помощью `indexOf()` нашли бы все запятые, и с помощью метода `slice()` разделить строку на меньшие куски, по одному на каждый элемент списка. Но есть более простой способ — преобразовать строку в массив с помощью метода `String.split()`:

```
let fruits = list.split(',');  
// Теперь fruits состоит из следующих элементов:  
// ['cherries', ' oranges', ' apples', ' bananas']
```

При вызове `split()` нужно указать разделитель. Это может быть пробел, несколько тире или что-то еще. Разделитель используется для того, чтобы разрезать строку на фрагменты, но сам он в эти фрагменты не попадет.

Обсуждение

Результатом разделения найденного фрагмента строки является массив элементов. Но в этих элементах могут присутствовать дефекты (в данном случае во всех, кроме первого, есть ведущий пробел). К счастью, есть простой способ их убрать.

Самое очевидное решение — перебрать все элементы массива строк и почистить каждый отдельно с помощью функции `trim()`, используя методику, описанную в рецепте 2.13. Это сработает, но есть более простой способ.

Хитрость состоит в том, чтобы взять метод `Array.map()`, который применяет переданный ему фрагмент кода к каждому элементу массива. Таким образом, для вызова функции `trim()` достаточно всего одной строки:

```
fruits = fruits.map(s => s.trim());  
// Теперь fruits состоит из следующих элементов:  
// ['cherries', 'oranges', 'apples', 'bananas']
```

Если вы не знакомы с синтаксисом массивов, использованным для применения функции `trim()` в этом примере, то найдете более подробное описание данной методики в рецепте 6.2.

Читайте также

Еще один способ найти нужные фрагменты в строке состоит в применении регулярных выражений. Например, можно построить регулярное выражение, которое в соответствии со структурой списка будет выбирать слова, заключенные между двумя запятыми. Регулярные выражения кратко описаны в рецепте 2.10, а в рецепте 2.12 рассматривается использование регулярных выражений для поиска.

2.12. Поиск по шаблону

Задача

Найти в строке все фрагменты, соответствующие заданному шаблону, и выполнить с каждым из них некую операцию.

Решение

Воспользуемся регулярным выражением и методом `String.matchAll()`. Метод `matchAll()` возвращает итератор, который позволяет перебрать все найденные совпадения.

В следующем примере применено регулярное выражение для поиска всех слов, начинающихся на *t* и заканчивающихся на *e*, с любым количеством букв между ними. Для того чтобы сформировать строку результатов, воспользуемся шаблонным литералом, синтаксис которого был описан в рецепте 2.5:

```
const searchString = 'Now is the time and this is the time and that is the time';  
const regex = /t\\w*e/g;  
  
const matches = searchString.matchAll(regex);  
for (const match of matches) {  
    console.log(`at ${match.index} we found ${match[0]}`);  
}
```

Результат выполнения этого кода выглядит так:

```
at 7 we found the  
at 11 we found time  
at 28 we found the
```

```
at 32 we found time
at 49 we found the
at 53 we found time
```

Обсуждение

Все результаты поиска, которые возвращает `matchAll()`, представляют собой объекты. При переборе этих результатов можно узнать текст (`match[0]`) и позицию, в которой этот текст был найден (`match.index`).

Но вот что немного странно в этом примере: даже перебирая результаты по одному, мы все равно используем `match[0]`, чтобы получить первый элемент массива. Этот массив существует, потому что благодаря круглым скобкам регулярное выражение позволяет запоминать несколько фрагментов, удовлетворяющих шаблону. Впоследствии на эти скобочные группы можно ссылаться. Предположим, например, что мы написали регулярное выражение для строки с информацией о человеке. Благодаря такому запоминанию можно легко выделить из найденных кусков отдельные фрагменты информации, такие как имя и дата рождения. При использовании такой методики в `matchAll()` найденные подстроки предоставляются в виде `match[1]`, `match[2]` и т. д.

Если вы не собираетесь сразу обрабатывать результаты поиска, то их можно объединить в массив с помощью `spread`-оператора:

```
const searchString = 'Now is the time and this is the time and that is the time';
const regex = /t\\w*e/g;
```

```
// Помещаем шесть найденных объектов в массив
const matches = [...searchString.matchAll(regex)];
```

Теперь можно в любой момент перебрать результаты поиска в цикле `foreach`. Но, напомним, результаты поиска — это не просто массив соответствующих фрагментов текста. Как мы видели в исходном примере, каждый такой объект содержит в себе позицию (`match.index`) и массив, состоящий из одного или нескольких фрагментов текста, начиная с `match[0]`.

Дополнительно: выделение результатов поиска

Рассмотрим более подробный пример, в котором показано, как найти и выделить фрагменты на веб-странице. На рис. 2.1 показано, как такое приложение работает для стихотворения Уильяма Вордсворта «Котенок и падающие листья».

На этой странице размещены поля `textarea` и `input` для ввода строки поиска и регулярного выражения. На основе этого регулярного выражения создается объект `RegExp`, который затем применяется к тексту, введенному в `textarea`, с помощью метода `matchAll()` — точно так же, как в предыдущем более коротком примере.

В процессе поиска соответствий код формирует строку, состоящую из текстовых фрагментов, как соответствующих, так и не соответствующих заданному шаблону.

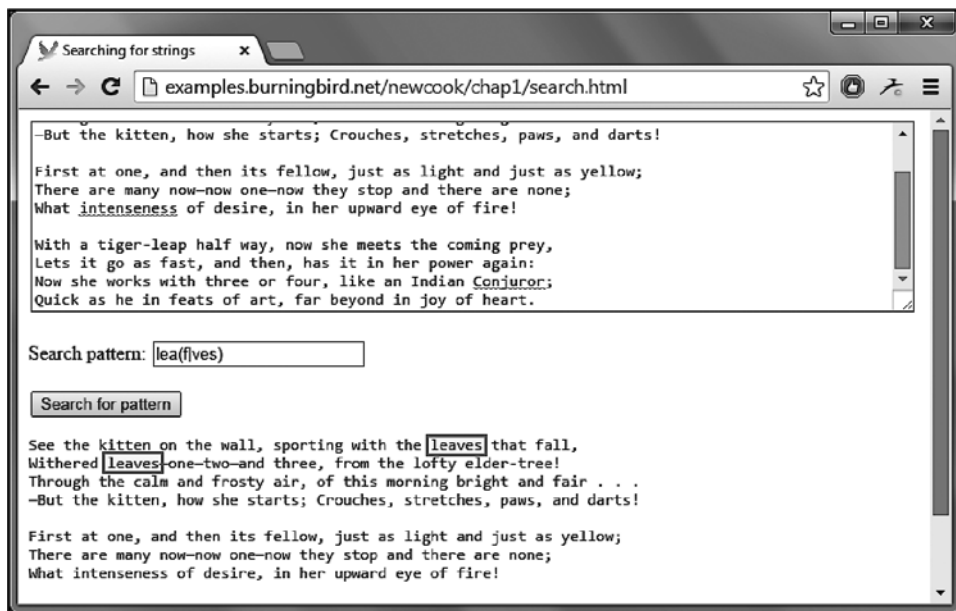


Рис. 2.1. Приложение, которое находит и выделяет все найденные строки

Фрагменты, соответствующие шаблону, заключаются в элементы `` с CSS-классом, используемым для выделения текста. Затем полученная строка вставляется в страницу с помощью свойства `innerHTML` элемента `<div>` (см. пример 2.1).

Пример 2.1. Выделение всех найденных фрагментов в текстовой строке

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <meta http-equiv="X-UA-Compatible" content="ie=edge" />
    <title>Finding All Instances of a Pattern</title>

    <style>
      .found {
        background-color: #ff0;
      }
      body {
        margin: 15px;
      }
      textarea {
        width: 100%;
        height: 350px;
      }
    </style>
```

```
</head>
<body>
  <h1>Finding All Instances of a Pattern</h1>

  <form id="textsearch">
    <textarea id="incoming">
    </textarea>
    <p>
      Search pattern: <input id="pattern" type="text">
    </p>
  </form>
  <button id="searchSubmit">Search for pattern</button>
  <div id="searchResult"></div>

  <script>
document.getElementById("searchSubmit").onclick = function() {
  // Получаем шаблон для поиска
  const pattern = document.getElementById('pattern').value;
  const regex = new RegExp(pattern, 'g');

  // Получаем текст для поиска
  const searchText = document.getElementById('incoming').value;

  let highlightedResult = "<pre>";
  let startPosition = 0;
  let endPosition = 0;

  // Находим все совпадения, формируем результат
  const matches = searchText.matchAll(regex);
  for (const match of matches) {
    endPosition = match.index;

    // Берем всю строку до этого совпадения,
    // присоединяем к предыдущему фрагменту
    highlightedResult += searchText.slice(startPosition, endPosition);

    // Добавляем найденный фрагмент
    // с CSS-классом для форматирования
    highlightedResult += "<span class='found'>" + match[0] + "</span>";
    startPosition = endPosition + match[0].length;
  }

  // Завершаем результирующую строку
  highlightedResult += searchText.slice(startPosition);
  highlightedResult += "</pre>";

  // Выводим текст с выделенными результатами поиска на страницу
  document.getElementById("searchResult").innerHTML =
    highlightedResult;
}
</script>
</body>
</html>
```

На рис. 2.1 показаны результаты поиска на этой странице по следующему регулярному выражению:

```
lea(f|ves)
```

Вертикальная черта (|) — это условие. Оно считается выполненным, если фрагмент соответствует любой из частей шаблона, стоящей справа или слева от вертикальной черты. Таким образом, условию соответствуют слова *leaf* и *leaves*, но не *leap*.

2.13. Удаление пробелов в начале и в конце строки

Задача

Удалить лишние пробелы, заполняющие начало и конец строки.

Решение

Воспользуемся методом `String.trim()`. Он удаляет все пробельные символы с обеих концов строки, включая пробелы, табуляции, неразрывные пробелы и символы конца строки:

```
const paddedString = '    The road is long, with many a winding turn.    ';\nconst trimmedString = paddedString.trim();\n\n// trimmedString = 'The road is long, with many a winding turn.'
```

Обсуждение

Метод `trim()` простой, но не настраиваемый. При наличии хоть немного более сложных требований к изменению строки придется задействовать регулярные выражения.

Одна из типичных проблем `trim()` состоит в том, что этот метод не позволяет удалить лишние пробелы внутри строки. Эта задача относительно легко решается с помощью метода `replaceAll()` при использовании регулярного выражения с символом `\s`, которое находит пробелы:

```
const paddedString = 'The road is long,    with many a    winding turn.';\nconst trimmedString = paddedString.replaceAll(/\\s\\s+/g, ' ');\n\n// trimmedString = 'The road is long, with many a winding turn.'
```

Разумеется, даже после обработки данных, засоренных лишними пробелами, могут остаться нежелательные артефакты. Например, если несколько пробелов стоят там, где они *вообще* не нужны (*is long*, *with*), в этом месте все равно останется

один пробел (*is long, with*). Единственный способ решить подобные проблемы — вручную перебрать все такие случаи, как показано в рецепте 2.12.

Читайте также

Синтаксис регулярных выражений описан в рецепте 2.10.

2.14. Замена первой буквы строки на прописную

Задача

Сделать первую букву строки прописной, не затрагивая всю остальную строку.

Решение

Отделить от строки первую букву и заменить ее прописной с помощью метода `String.toUpperCase()`. Затем присоединить прописную букву к остальной строке, используя метод `String.slice()`:

```
const original = 'if you cut an orange, there is a risk it will orbisculate.';
const fixed = original[0].toUpperCase() + original.slice(1);

// fixed = 'If you cut an orange, there is a risk it will orbisculate.';
```

Обсуждение

Для того чтобы отделить от строки один символ, можно воспользоваться индексом: `original[0]`. В результате получим символ, стоящий в позиции 0, то есть первый символ:

```
const firstLetter = original[0];
```

Или можно задействовать метод `String.charAt()`, который работает точно так же.

Для того чтобы получить фрагмент строки, применяется метод `slice()`. При вызове `slice()` нужно обязательно указать индекс, с которого начинается фрагмент. Например, `text.slice(5)` берет фрагмент строки, который начинается с позиции 5 и продолжается до конца строки, и возвращает его как новую строку.

Если `slice()` не должен доходить до конца строки, можно передать в метод необязательный второй параметр — индекс, на котором копирование строки должно остановиться:

```
// Получить строку от символа в позиции 5 до символа в позиции 10
const substring = original.slice(5, 10);
```

В этом рецепте показано, как заменить одну букву на прописную. Задача становится сложнее, если нужно заменить на прописные (верхний регистр) первые буквы слов во всем предложении. Для этого можно, например, разделить строку на отдельные слова, убрать пробелы и затем объединить результаты, используя прием, подобный описанному в рецепте 2.11.

Читайте также

Для поиска определенных фрагментов текста, которые нужно извлечь, можно применить метод `slice()` в сочетании с `indexOf()`. Пример этого приема рассматривается в рецепте 2.11.

2.15. Валидация адреса электронной почты

Задача

Обнаруживать типичные ошибки в адресах электронной почты и отклонять такие адреса.

Решение

Регулярные выражения полезны не только при поиске. Их можно применять и при валидации строк, проверяя эти строки на соответствие заданным шаблонам. В JavaScript, для того чтобы проверить, соответствует ли строка регулярному выражению, используется метод `RegExp.test()`:

```
const emailValid = "abelincoln@gmail.com";
const emailInvalid = "abelincoln@gmail .com";
const regex = /\S+@\S+\.\S+/;

if (regex.test(emailValid)) {
    // Этот код выполняется, так как адрес emailValid прошел проверку
}
if (regex.test(emailInvalid)) {
    // Этот код не выполняется, так как адрес emailInvalid не прошел проверку
}
```

Обсуждение

Программисты используют различные регулярные выражения для валидации электронных адресов. Лучшие из этих выражений находят очевидные ошибки и поддельные значения, но при этом не слишком сложны. Переусложненные регулярные выражения время от времени отклоняют правильные адреса электронной почты как недопустимые. Но даже если адрес электронной почты прошел самый строгий из возможных тестов, это еще не значит, что он действительно

корректный (по крайней мере, пока вы не отправите по нему письмо и не получите подтверждение).

Регулярное выражение, использованное в этом рецепте, требует, чтобы адрес электронной почты состоял как минимум из одного непробельного символа, после которого стоит знак @, затем еще один или более непробельных символов, точка (.) и снова один или несколько непробельных символов. Такой шаблон не пропустит очевидно некорректные адреса, такие как `tomkhangmail.com` или `tomkhan@gmail`.

Вместо того чтобы писать регулярные выражения для валидации самостоятельно, как правило, лучше применять готовые выражения для соответствующего типа данных. Обширная коллекция регулярных выражений размещена на странице Awesome Regex (<https://github.com/alaisdg/.awesome-regex>).

Читайте также

Синтаксис регулярных выражений описан в рецепте 2.10.

ГЛАВА 3

Числа

Немного найдется ингредиентов для рецептов повседневного программирования, которые были бы важнее чисел. Во многих современных языках программирования есть целый набор различных цифровых типов данных для использования в разных сценариях, таких как целые числа, десятичные дроби, числа с плавающей точкой и т. п. Но JavaScript в отношении чисел проявляет себя как скороспелое, несколько импровизированное творение, известное как язык сценариев со слабой типизацией.

До последнего времени в JavaScript был всего один тип цифровых данных на все случаи жизни — `Number`. Сейчас таких типов стало два: стандартный `Number`, который можно применять почти всегда, и очень узкоспециализированный `BigInt`, который стоит использовать только в работе с очень большими целыми числами. В этой главе мы задействуем оба типа, а также вспомогательные методы объекта `Math`.

3.1. Генерирование случайных чисел

Задача

Генерировать случайные целые числа, попадающие в определенный диапазон (например, от 1 до 6).

Решение

Метод `Math.random()` позволяет генерировать значения с плавающей точкой в диапазоне от 0 до 1. Обычно, чтобы получить целое число в заданном диапазоне, увеличивают и округляют дробное число, полученное от `Math.random()`. Если считать, что диапазон ограничен неким минимальным числом `min` и максимальным `max`, то для этого нужно сделать следующее:

```
randomNumber = Math.floor(Math.random() * (max - min + 1) ) + min;
```

Например, для того чтобы получить случайное число в диапазоне от 1 до 6, код следует сделать таким:

```
const randomNumber = Math.floor(Math.random()*6) + 1;
```

Теперь возможными значениями `randomNumber` являются 1, 2, 3, 4, 5 или 6.

Обсуждение

Объект `Math` состоит из статических вспомогательных методов, которые можно задействовать в любой момент. В данном рецепте использован метод `Math.random()` для получения случайного дробного числа и метод `Math.floor()`, который отсекает дробную часть, оставляя лишь целое число.

Чтобы понять, как это работает, рассмотрим этот пример пошагово. Вначале метод `Math.random()` выдает значение от 0 до 1, допустим, 0,374324823:

```
const randomNumber = Math.floor(0.374324823*6) + 1;
```

Затем это число умножается на количество значений в диапазоне (в данном примере 6), и получается 2,245948938:

```
const randomNumber = Math.floor(2.245948938) + 1;
```

Затем функция `Math.floor()` обрезает это число до 2:

```
const randomNumber = 2 + 1;
```

И наконец, прибавляется начальное число диапазона, что дает окончательный результат 3. Повторяя это вычисление, мы получим другие числа, но они всегда окажутся целыми и будут попадать в диапазон от 1 до 6.

Читайте также

Метод `Math.floor()` — лишь один из возможных вариантов округления чисел. О других вы можете узнать в рецепте 3.3.

Важно понимать, что числа, генерируемые `Math.random()`, являются *псевдослучайными* — другими словами, их можно угадать либо вычислить и воспроизвести закономерность. Эти числа недостаточно случайны для криптографии, лотерей или сложного моделирования. Подробнее о других методах генерирования случайных чисел читайте в рецепте 3.2. Если же вам нужен способ генерировать повторяемую последовательность псевдослучайных чисел, то прочитайте подраздел «Дополнительно: построение многократно вызываемого генератора псевдослучайных чисел» в главе 6.

3.2. Генерирование криптографически надежных случайных чисел

Задача

Генерировать случайные числа, для которых нельзя легко вычислить (предсказать) закономерность.

Решение

С помощью свойства `window.crypto` получить экземпляр объекта `Crypto`. Использовать метод `Crypto.getRandomValues()` для генерирования случайных чисел, *энтропия* которых выше, чем у тех, которые генерирует `Math.random()`. (Другими словами, эти числа гораздо труднее повторить или предсказать — подробнее об этом читайте в подразделе «Обсуждение».)

Метод `Crypto.getRandomValues()` работает не так, как `Math.random()`. Вместо того чтобы возвращать числа с плавающей точкой в диапазоне от 0 до 1, `getRandomValues()` заполняет массив случайными целыми числами. Они могут быть 8-, 16- или 32-битными, со знаком или без. (Числа со знаком могут быть положительными или отрицательными, в то время как числа без знака всегда положительные.)

Существует общепринятое обходное решение, позволяющее получить от `getRandomValues()` традиционные значения в диапазоне от 0 до 1. Суть его в том, чтобы разделить случайное значение на максимально возможное число, которое может быть представлено этим типом данных:

```
const randomBuffer = new Uint32Array(1);
window.crypto.getRandomValues(randomBuffer);
const randomFraction = randomBuffer[0] / (0xffffffff + 1);
```

Теперь можно работать с `randomFraction` точно так же, как с числами, полученными от `Math.random()`. Например, преобразовать в целые случайные числа в заданном диапазоне, как показано в рецепте 3.1:

```
// Преобразование случайных дробных чисел
// в случайные целые числа от 1 до 6
const randomNumber = Math.floor(randomFraction*6) + 1;
console.log(randomNumber);
```

При выполнении кода в среде `Node.js` объект `window` недоступен. Зато можно получить доступ к очень похожей реализации Web Crypto API:

```
const crypto = require('crypto').webcrypto;
```

Обсуждение

В этом примере скрыто много интересного. Прежде всего, даже если вы не хотите слишком углубляться в работу этого кода, следует знать несколько важных моментов о реализации метода `Crypto.getRandomValues()`.

- Технически `Crypto` генерирует псевдослучайные числа по математической формуле, подобной той, что реализована в `Math.random()`. Однако, в отличие от `Math.random()`, эти числа считаются *криптографически стойкими*, так как исходным значением для генератора случайных чисел является действительно случайное число. Преимущество такого компромисса заключается в том, что метод `getRandomValues()` имеет примерно такую же производительность, как и `Math.random()`. (Он быстрый.)
- Не существует способа узнать, какое именно число служит начальным значением для `Crypto`, так как это зависит от реализации (для кода веб-страницы — от производителя браузера), а реализация, в свою очередь, опирается на функционал операционной системы. Как правило, начальное значение создается как сочетание последних записанных данных о времени срабатывания клавиатуры, движениях мыши и операциях чтения с запоминающих устройств.
- Сколь бы хороши ни были случайные числа, но если код JavaScript выполняется в браузере, он будет подвергаться многочисленным атакам. В конце концов, ничто не мешает злоумышленникам увидеть ваш код и написать вместо него другой, который полностью обойдет генерацию случайных чисел. Но если код работает на сервере — это совсем другое дело, к нему это предупреждение не относится.

Теперь давайте внимательнее присмотримся к тому, как работает `getRandomValues()`. Перед вызовом `getRandomValues()` нужно создать *типизированный массив* — массивоподобный объект, в котором могут храниться только значения определенного типа. (Мы называем этот объект *массивоподобным*, так как он ведет себя как массив, но формально не является экземпляром типа `Array`.) В JavaScript есть несколько объектов строго типизированных массивов: `Uint32Array` (массив 32-разрядных целых чисел без знака), `Uint16Array`, `Uint8Array` и их аналоги со знаком `Int32Array`, `Int16Array` и `Int8Array`. Сколь бы большим ни был созданный массив, `getRandomValues()` все равно заполнит весь выделенный ему буфер.

В этом рецепте мы выделяем место только для одного значения в `Uint32Array`:

```
const randomBuffer = new Uint32Array(1);
window.crypto.getRandomValues(randomBuffer);
```

Заключительный этап — разделить это случайное число на максимально возможное 32-разрядное целое число, то есть на 4 294 967 295. Будет понятнее, если представить это число в шестнадцатеричной записи — `0xffffffff`:

```
const randomFraction = randomBuffer[0] / (0xffffffff + 1);
```

Как видно из кода, к этому максимальному значению нужно прибавить 1. Так сделано, потому что, теоретически, случайное значение может быть равно максимальному целому числу. Если это произойдет, то `randomFraction` будет равно 1, что отличается от результатов, возвращаемых `Math.random()` и большинством других генераторов случайных чисел. (Это крошечное отклонение от нормы может привести сначала к некорректному предположению, а затем, однажды в будущем, — к ошибке.)

3.3. Округление до заданного десятичного разряда

Задача

Округлить число до определенной точности (например, 124,793 до 124,80 или до 120).

Решение

Для округления числа до ближайшего целого можно использовать метод `Math.round()`:

```
const fractionalNumber = 19.48938;
const roundedNumber = Math.round(fractionalNumber);

// Теперь roundedNumber равно 19
```

Как ни странно, но метод `round()` не принимает аргумент с числом десятичных разрядов, которые следовало бы сохранить. Если нужна другая степень точности, приходится самостоятельно умножить число на 10 в соответствующей степени, округлить его и затем разделить на 10 в той же степени. Вот общая формула этой операции:

```
const numberToRound = fractionalNumber * (10**numberOfDecimalPlaces);
let roundedNumber = Math.round(numberToRound);
roundedNumber = roundedNumber / (10**numberOfDecimalPlaces);
```

Например, код для округления до двух знаков после точки выглядит так:

```
const fractionalNumber = 19.48938;
const numberToRound = fractionalNumber * (10**2);
let roundedNumber = Math.round(numberToRound);
roundedNumber = roundedNumber / (10**2);

// Теперь roundedNumber равно 19.49
```

Если нужно округлить до определенного разряда *слева* (до ближайших десятков, сотен и т. д.), присвойте `numberOfDecimalPlaces` соответствующее отрицательное значение, например `-1` для округления до десятков, `-2` — до сотен и т. д.

Обсуждение

У объекта `Math` есть несколько статических методов для преобразования дробных чисел в целые. Метод `floor()` удаляет все цифры после точки, округляя число до ближайшего целого в меньшую сторону. Метод `ceil()`, наоборот, всегда округляет дробное число в большую сторону. Метод `round()` округляет число до ближайшего целого.

В работе метода `round()` есть два важных момента, о которых следует знать.

- Числа, в которых после точки стоит ровно 5, всегда округляются в большую сторону, несмотря на то что они находятся точно посередине между ближайшими меньшим и большим целыми числами. В финансовой и научной сфере часто применяются различные методы преодоления этого перекоса, такие как округление одних значений с 5 после точки в большую сторону, а других — в меньшую. Но для реализации такого поведения в JavaScript нужно либо писать код самостоятельно, либо использовать стороннюю библиотеку.
- При округлении отрицательных чисел JavaScript округляет $-0,5$ до нуля. Соответственно, $-4,5$ будет округлено до -4 , что отличается от правил округления, принятых во многих других языках программирования.

Читайте также

Округление чисел — один из способов привести числовое значение к виду, удобному для визуализации. Если вы выполняете округление при подготовке чисел, перед тем как показать их пользователю, вас могут заинтересовать методы форматирования значений типа `Number`, описанные в рецепте 2.2.

3.4. Сохранение точности в дробных числах

Задача

Все числа в JavaScript представляют собой значения с плавающей точкой, что в случае очень малых величин вызывает ошибки округления при выполнении определенных операций. В некоторых приложениях (например, в тех, которые оперируют денежными суммами) такие ошибки неприемлемы.

Решение

Ошибки округления чисел с плавающей точкой — хорошо известное явление, которое существует практически в любом языке программирования. Для того чтобы понаблюдать его в JavaScript, выполните следующий код:

```
const sum = 0.1 + 0.2;  
console.log(sum);           // выведет 0.30000000000000004
```

Мы не можем избежать ошибки округления, но можем свести ее к минимуму. При работе с денежными суммами в определенной валюте (например, в долларах) можно умножать все значения на 100, чтобы не иметь дела с дробными величинами. Вместо того чтобы писать такой код:

```
const currentBalance = 5382.23;
const transactionAmount = 14.02;

const updatedBalance = currentBalance - transactionAmount;

// Теперь updatedBalance = 5368.209999999999
```

лучше использовать следующие переменные для денежных сумм:

```
const currentBalanceInCents = 538223;
const transactionAmountInCents = 1402;

const updatedBalanceInCents = currentBalanceInCents - transactionAmountInCents;

// Теперь updatedBalanceInCents = 536821
```

Этот прием решит проблему для тех операций, которые работают с целыми числами без потери точности, такими как сложение и вычитание количества центов. Но что будет, если понадобится вычислить налог или доход в процентах? Результатом этих операций в любом случае будут дробные значения. Придется делать то же самое, что делают в таких случаях предприятия и банки, — округлять значения сразу после транзакции:

```
const costInCents = 4899;

// Вычислить налог в 11 % и округлить результат
// до ближайшего целого числа центов
const costWithTax = Math.round(costInCents*1.11);
```

Обсуждение

Проблема округления чисел с плавающей точкой состоит в том, что некоторые дробные значения невозможно хранить в двоичном представлении без округления. То же самое иногда происходит и в десятичной системе счисления (например, попробуйте записать результат деления $1/3$). Но в случае чисел с плавающей точкой результат еще и нелогичен. Мы не ожидаем проблем, суммируя 0,1 и 0,2, так как в десятичной записи обе эти дроби могут быть представлены точно.

Тот же феномен наблюдается и в других языках программирования, но во многих из них есть альтернативный тип данных для представления дробных чисел и денежных сумм. В JavaScript такого типа данных нет. Правда, есть предложение ввести новый тип данных `Decimal` в следующих версиях языка JavaScript (<https://github.com/tc39/proposal-decimal>).

Читайте также

Если приходится выполнять много финансовых вычислений, то можно упростить себе жизнь, воспользовавшись сторонней библиотекой, такой как `bignumber.js` (<https://github.com/MikeMcl/bignumber.js>). В ней есть специализированный числовой тип данных, который во многом подобен обычному `Number`, но обеспечивает дополнительную точность для чисел с определенным количеством знаков после точки.

3.5. Преобразование строки в число

Задача

Проанализировать число, представленное в виде строки, и преобразовать эту строку в числовой тип данных.

Решение

Преобразовать число в строку всегда безопасно — эта операция не может завершиться неудачно. Обратная задача — преобразовать строку в число, чтобы потом использовать его в вычислениях, — дело более тонкое.

Общепринятый способ состоит в применении функции `Number()`:

```
const stringData = '42';
const numberData = Number(stringData);
```

Функция `Number()` не принимает такие элементы форматирования, как символы валют и запятые, в качестве разделителей. Она лишь допускает лишние пробелы в начале и конце строки. Зато принимает пустые строки или строки, состоящие только из пробелов, которые преобразует в число 0. Иногда (например, при получении значения из текстовой строки ввода) это можно считать разумным значением по умолчанию, но оно не всегда приемлемо. Во избежание таких случаев можно перед вызовом `Number()` выполнить следующую проверку:

```
if (stringData.trim() === '') {
  // Это пустая строка либо строка, состоящая только из пробелов
}
```

Если преобразование не удастся, то функция `Number()` возвращает значение `NaN` (*Not a Number* — не число). Чтобы проверить, не случился ли сбой при выполнении `Number()`, можно сразу после ее использования вызвать метод `Number.isNaN()`:

```
const numberData = Number(stringData);

if (Number.isNaN(numberData)) {
  // Эти данные можно обрабатывать как число
}
```



Метод `isFinite()` практически не отличается от `isNaN()`, за исключением того что обрабатывает странные граничные случаи, такие как $1/0$, в которых возвращает значение `infinity`. Если применить метод `isNaN()` для `infinity`, то он вернет `false`, что выглядит несколько неоднозначно.

Альтернативный вариант — использовать метод `parseFloat()`. Это несколько более свободный вариант, допускающий текст после числа. Но зато `parseFloat()` строже относится к пустым строкам — он их не принимает:

```
console.log(Number('42'));           // 42
console.log(parseFloat('42'));       // 42

console.log(Number('12 goats'));      // NaN
console.log(parseFloat('12 goats'));  // 12

console.log(Number('goats 12'));      // NaN
console.log(parseFloat('goats 12'));  // NaN

console.log(Number('2001/01/01'));    // NaN
console.log(parseFloat('2001/01/01')); // 2001

console.log(Number(' '));              // 0
console.log(parseFloat(' '));          // NaN
```

Обсуждение

У разработчиков есть несколько уловок для преобразования строк в числа, таких как умножение строки на 1 (`numberInString*1`) или использование унарного плюса (`+numberInString`). Функционально эти приемы эквивалентны `Number()`. Но чтобы код лучше читался, стоит применять `Number()` или `parseFloat()`.

Для преобразования форматированных чисел, таких как 2,300¹, необходимо выполнить дополнительную работу. Метод `Number()` вернет `NaN`, а `parseFloat()` остановится на запятой и вернет 2. К сожалению, несмотря на то что в JavaScript есть объект `Intl.NumberFormat`, позволяющий преобразовывать числа в форматированные строки (см. рецепт 2.2), в нем нет функции синтаксического анализа, которая позволила бы выполнить обратную операцию.

Для решения таких задач, как удаление запятых из строки, можно было бы задействовать регулярные выражения (см. рецепт 2.8). Но подобные доморощенные приемы рискованны, так как в некоторых локалях запятые применяются для разделения на разряды, а в других — для отделения дробной части. В такой ситуации лучше подойдет широко используемая и хорошо протестированная библиотека JavaScript, такая как `Numeral` (<http://numeraljs.com>).

¹ Значения с запятыми на месте разделения разрядов используются в англоязычных странах. — *Примеч. пер.*

3.6. Преобразование десятичных значений в шестнадцатеричные

Задача

Есть десятичное число, нужно вычислить его шестнадцатеричный эквивалент.

Решение

Использовать метод `Number.toString()` с аргументом, определяющим основание системы счисления, *в которую* выполняется преобразование:

```
const num = 255;

// выведет ff — шестнадцатеричный эквивалент 255
console.log(num.toString(16));
```

Обсуждение

По умолчанию числа в JavaScript представляются в системе счисления с *основанием* 10, то есть в десятичной. Но их можно перевести в систему счисления с другим основанием, такую как шестнадцатеричная (16) или восьмеричная (8). Запись шестнадцатеричных чисел начинается с 0x (ноль, после которого стоит буква x в нижнем регистре). Перед восьмеричными числами раньше ставился ноль (0), но теперь перед ними должен стоять ноль и латинская буква O (в верхнем или нижнем регистре):

```
const octalNumber = 0o255;    // 173 в десятичной системе счисления
const hexaNumber = 0xad;     // 173 в десятичной системе счисления
```

Десятичное число можно преобразовать в другую систему счисления с основанием от 2 до 36:

```
const decNum = 55;
const octNum = decNum.toString(8);    // 67 в восьмеричной системе
const hexNum = decNum.toString(16);   // 37 в шестнадцатеричной системе
const binNum = decNum.toString(2);    // 110111 в двоичной системе
```

Для окончательного представления числа в восьмеричной или шестнадцатеричной системе счисления нужно присоединить к восьмеричному числу строку 0o, а к шестнадцатеричному — 0x. Но учтите, что после преобразования числа в строку его не следует использовать в сортировке или операциях с числами независимо от того, как оно будет отформатировано.

Несмотря на то что десятичные числа могут быть преобразованы в любую систему счисления от 2 до 36, числовые операции возможны только для восьмеричных, шестнадцатеричных и десятичных значений.

3.7. Преобразование градусов в радианы

Задача

Задан угол в градусах. Для использования этого значения в тригонометрических функциях объекта `Math` необходимо преобразовать его из градусов в радианы.

Решение

Для того чтобы преобразовать градусы в радианы, нужно умножить значение в градусах на $(\text{Math.PI}/180)$:

```
const radians = degrees * (Math.PI / 180);
```

Таким образом, для угла 90° получаем следующее вычисление:

```
const radians = 90 * (Math.PI / 180);  
console.log(radians); // 1.5707963267948966
```

Для того чтобы преобразовать радианы в градусы, нужно умножить значение в радианах на $(180/\text{Math.PI})$:

```
const degrees = radians * (180 / Math.PI);
```

Обсуждение

Все тригонометрические методы объекта `Math` — `sin()`, `cos()`, `tan()`, `asin()`, `acos()`, `atan()` и `atan2()` — принимают значения в радианах и возвращают результат также в радианах. Однако нет ничего необычного в том, что чаще предоставляют значения в градусах, чем в радианах, так как градусы — более известная единица измерения.

3.8. Вычисление длины дуги окружности

Задача

Заданы радиус окружности и угол дуги в градусах. Нужно найти длину дуги.

Решение

С помощью `Math.PI` преобразовать градусы в радианы и использовать этот результат в формуле вычисления длины дуги:

```
// угол дуги — 120 градусов, радиус окружности — 2  
const radians = degrees * (Math.PI / 180);  
const arclength = radians * radius; // результат: 4.18879020478...
```

Обсуждение

Длина дуги окружности вычисляется умножением радиуса окружности на угол дуги в радианах.

Если угол представлен в градусах, то перед его умножением на радиус нужно преобразовать градусы в радианы. Как будет рассказано в главе 15, эта операция часто выполняется при рисовании фигур в формате SVG.

3.9. Манипуляции с очень большими целыми числами в формате BigInt

Задача

Обрабатывать очень большие целые числа (свыше 2^{53}) без потери точности.

Решение

Использовать тип данных **BigInt**, позволяющий хранить целые числа любого размера и ограниченный только памятью операционной системы (или реализацией **BigInt** в движке JavaScript).

Есть два способа создания числа в формате **BigInt**. Первый — с помощью функции **BigInt()**:

```
// Создаем число BigInt и присваиваем ему значение 10
const bigInteger = BigInt(10);
```

Второй вариант — поставить в конце числа букву *n*:

```
const bigInteger = 10n;
```

В следующем примере показана разница между обычным типом **Number** и **BigInt** в случае очень больших чисел:

```
// При обработке больших чисел в обычном формате
// Number возникают неточности
const maxInt = Number.MAX_SAFE_INTEGER // Что-то около 9007199254740991
console.log(maxInt + 1); // 9007199254740992 (разумно)
console.log(maxInt + 2); // 9007199254740992 (это не опечатка, а ошибка)
console.log(maxInt + 3); // 9007199254740994 (точно)
console.log(maxInt + 4); // 9007199254740996 (а теперь-то почему неправильно?)

// В этом случае надежнее использовать тип BigInt
const bigInt = BigInt(maxInt);
console.log(bigInt + 1n); // 9007199254740992 (как и раньше)
console.log(bigInt + 2n); // 9007199254740993 (уже лучше)
console.log(bigInt + 3n); // 9007199254740994 (по-прежнему хорошо)
console.log(bigInt + 4n); // 9007199254740995 (отлично!)
```



При выводе числа `BigInt` в консоль в конце значения появляется буква `n` (например, `9007199254740992n`). Благодаря этому соглашению значения `BigInt` легко распознаются. Если же нужно вывести только числовое значение `BigInt`, сначала преобразуйте его в текст с помощью `BigInt.toString()`.

Обсуждение

Собственный тип данных `Number` в JavaScript соответствует спецификации IEEE-754 для 64-разрядных чисел двойной точности с плавающей точкой. Этот стандарт имеет приемлемые известные ограничения и неточности. Одно из разумных ограничений — то, что в нем не могут быть точно представлены целые числа свыше 2^{53} . Если до достижения этой черты все неточности представления начинались справа от десятичной точки (см. рецепт 3.4), то за ее пределами они перемещаются влево от десятичной точки. Другими словами, чем больше числа, с которыми имеет дело движок JavaScript, тем выше вероятность возникновения неточностей. За пределами 2^{53} эта неточность превышает 1 и становится заметной в операциях не только с дробными, но и с целыми числами.

Частично эта проблема JavaScript решается с помощью типа `BigInt`, появившегося в спецификации ECMAScript 2020. `BigInt` — это целочисленные значения произвольного размера, что позволяет представлять значительно бóльшие числа. На практике значения типа `BigInt` могут быть произвольной битовой длины.

К `BigInt` применимы практически все операции, которые выполняются с обычными числами, в том числе сложение (+), вычитание (-), умножение (*), деление (/) и возведение в степень (**). Но `BigInt` является целочисленным типом и не может хранить дробные значения. При выполнении операции деления `BigInt` без предупреждения отбрасывает дробную часть:

```
const result = 10n / 6n; // Результат: 1
```

Значения в формате `BigInt` и `Number` невазимоаменяемы и несовместимы. Но их можно преобразовывать одно в другое с помощью функций `Number()` и `BigInt()`:

```
let bigInteger = 10n;
let integer = Number(bigInteger); // integer равно 10

integer = 20;
bigInteger = BigInt(integer);    // bigInteger равно 20n
```

Такое преобразование требуется выполнять в тех случаях, когда нужно использовать значения типа `BigInt` в методах, которые принимают `Number`, таких как методы объекта `Math`. Аналогичным образом нужно преобразовывать `Number` в `BigInt` в вычислениях, где присутствуют другие значения типа `BigInt`.

При попытке преобразовать в `BigInt` переменную типа `Number`, в которой хранится дробное значение, получим ошибку `RangeError`. Чтобы этого избежать, вначале выполним округление:


```
const decimal = 10.8;
const bigInteger = BigInt(Math.round(decimal)); // bigInteger равно 11n
```

Не забывайте, что операции должны соответствовать типу. Иногда простая на вид операция может не выполняться из-за того, что в ней случайно оказались и значения `BigInt`, и обычные числа:

```
let x = 10n;
x = x * 2;    // выбрасывает исключение TypeError, так как
              // x имеет тип BigInt, а 2 — Number
x += 1;       // тоже выбрасывает исключение TypeError

x = x * 2;    // Теперь x равно 20n, как и ожидалось
x += 1n;      // x равно 21
```

Значения `BigInt` можно сравнивать с `Number` при помощи обычных операторов сравнения `<`, `>`, `<=`, `>=`. Для того чтобы проверить равенство значений типа `BigInt` и `Number`, можно использовать операторы нестрогого равенства (`==` и `!=`). Строгое равенство (`===`) всегда будет возвращать `false`, так как `BigInt` и `Number` — разные типы данных. Но еще лучше явно преобразовать `Number` в `BigInt` и сравнить результаты с помощью оператора `===`.

И последнее, что нужно учитывать, задействуя `BigInt`, — этот тип данных (на момент написания книги) не сериализуется в JSON. Попытка вызвать `JSON.stringify()` для `BigInt` приведет к синтаксической ошибке. Эту проблему можно решить при помощи самодельного патча, написав для `BigInt` следующий метод `toJSON()`:

```
BigInt.prototype.toJSON = function() { return this.toString() }
```

Также можно использовать библиотеку, такую как `granola` (<https://github.com/kanongil/granola>), в которой есть JSON-совместимые функции преобразования в строку для многих типов данных, в том числе `BigInt`.

ГЛАВА 4

Даты

На удивление, в JavaScript реализованы очень мощные механизмы обработки дат, упакованные в несколько старомодный объект `Date`. Как вы увидите, у объекта `Date` есть свои причуды и скрытые ловушки, вроде того что отсчет месяцев начинается с 0, а синтаксический анализ года выполняется в зависимости от регионального стандарта, выбранного на данном компьютере. Но научившись лавировать между этими подводными камнями, вы сможете выполнять множество полезных типовых операций, таких как подсчет количества дней между двумя датами, форматирование дат для вывода на экран, а также установка времени выполнения событий.

4.1. Получение текущих даты и времени

Задача

Получить текущие дату и время.

Решение

В JavaScript есть объект `Date` с хорошими возможностями для управления информацией о дате (и более скромными — для вычислительных операций с датами). При создании нового объекта `Date` в него автоматически заносятся текущие дата и время с округлением до ближайшей миллисекунды в меньшую сторону:

```
const today = new Date();
```

Теперь остается просто извлечь из объекта `Date` нужную информацию. Для решения этой задачи в объекте `Date` есть длинный список методов. Самые важные из них перечислены в табл. 4.1. Обратите внимание на то, что начало отсчета в разных методах может различаться: месяцы и дни недели отсчитываются от 0, а дни нумеруются начиная с 1.

Таблица 4.1. Методы обработки дат для получения различной информации о дате

Метод	Возвращает	Возможные значения
<code>getFullYear()</code>	Год	Четырехзначное число, например 2021
<code>getMonth()</code>	Номер месяца	Число от 0 до 11, где 0 соответствует январю
<code>getDate()</code>	День месяца	Число от 1 до 31
<code>getDay()</code>	День недели	Число от 0 до 6, где 6 соответствует воскресенью
<code>getHours()</code>	Время суток в часах	Число от 0 до 23
<code>getMinutes()</code>	Минуты	Число от 0 до 59
<code>getSeconds()</code>	Секунды	Число от 0 до 59
<code>getMilliseconds()</code>	Миллисекунды (тысячные доли секунды)	Число от 0 до 999

Вот пример вывода основной информации о текущей дате:

```
const today = new Date();

console.log(today.getFullYear()); // например, 2021
console.log(today.getMonth());   // например, 02 (март)
console.log(today.getDay());     // например, 0 (понедельник)

// Добавим небольшую обработку, чтобы при необходимости
// перед минутами выводились лидирующие нули и получалось
// двузначное число, как "05" в значении времени 4:05
const hours = today.getHours();
const minutes = today.getMinutes().toString().padStart(2, '0');
console.log('Time ' + hours + ':' + minutes); // например, 15:32
```



Методы `Date`, перечисленные в табл. 4.1, существуют в двух версиях. В тех, которые представлены в таблице, используются локальные параметры времени. Второй набор методов имеет префикс UTC, например `getUTCMonth()` и `getUTCSeconds()`. В них применяется общемировой стандарт времени — всемирное координированное время (Coordinated Universal Time). Если нужно сравнить даты для разных часовых поясов или мест, где приняты разные соглашения о переходе на летнее время, необходимо задействовать UTC-методы. Внутри объекта `Date` всегда используется стандарт UTC.

Обсуждение

У объекта `Date()` есть несколько конструкторов. Как мы только что выяснили, пустой конструктор создает объект `Date` с текущими датой и временем. Но можно создать объект `Date` и для другой даты, указав год, месяц и день:

```
// 10 февраля 2021 года:
const anotherDay = new Date(2021, 1, 10);
```

Подчеркнем еще раз: помните о разных точках отсчета (месяцы начинаются с 0, а дни — с 1). Это значит, что переменная `anotherDay` в предыдущем примере соответствует 10 февраля, а не 10 января.

При желании можно указать в конструкторе `Date` до четырех дополнительных параметров — часы, минуты, секунды и миллисекунды:

```
// 1 февраля 2021 года, 9:30:  
const anotherDay = new Date(2021, 1, 1, 9, 30);
```

Как вы узнаете далее в этой главе, у встроенного в JavaScript объекта `Date` есть ряд известных ограничений и несколько странностей. Если ваш код требует большого количества операций с датами, таких как вычисление диапазона дат, синтаксический анализ строк с датами или перенос дат между часовыми поясами, стоит использовать хорошо протестированную стороннюю библиотеку, такую как `day.js` (<https://github.com/iamkun/dayjs>) или `date-fns` (<https://date-fns.org>).

Читайте также

Если в вашей задаче есть даты, то вы, вероятно, захотите использовать их в вычислениях, как показано в рецепте 4.4. Также вас, возможно, заинтересует способ преобразования даты в форматированную строку (рецепт 4.6) или же преобразование строки с датой в соответствующий объект `Date` (рецепт 4.2).

4.2. Преобразование строки в дату

Задача

Есть информация о дате, представленная в виде строки. Мы хотим преобразовать ее в объект `Date`, чтобы потом манипулировать им в коде или вычислять даты.

Решение

Если повезет, то вам достанется строка с датой в стандартном формате метки времени ISO 8601 (вида `2021-12-17T03:24:00Z`), которую можно передать в конструктор `Date` напрямую:

```
const eventDate = new Date('2021-12-17T03:24:00Z');
```

Буква `T` в этой строке отделяет дату от времени, а буква `Z` в конце строки показывает, что это универсальное время, заданное с использованием часового пояса UTC, что обеспечивает наилучшую совместимость на разных компьютерах.

Существуют и другие форматы, распознаваемые конструктором `Date` и методом `Date.parse()`. Но применять их настоятельно не рекомендуется, поскольку реализация этих форматов неодинаковая в разных браузерах. При рассмотрении

тестовых примеров может показаться, что они работают, однако в разных браузерах задействуются разные параметры, зависящие от региональных стандартов, такие как переход на летнее время, поэтому использование таких форматов вызовет проблемы.

Если дата представлена не в формате ISO 8601, необходимо исправить это вручную. Нужно извлечь из строки отдельные компоненты даты и передать их в конструктор `Date`. Для этого хорошо подойдут такие методы `String`, как `split()`, `slice()` и `indexOf()`, которые были подробно описаны в рецептах главы 2.

Например, если есть строка в формате `mm/dd/yyyy`, то можно использовать следующий код:

```
const stringDate = '12/30/2021';

// разбить строку на части по косым
const dateArray = stringDate.split('/');

// получить отдельные части даты
const year = dateArray[2];
const month = dateArray[0];
const day = dateArray[1];

// скорректировать номер месяца с учетом отсчета от 0
const eventDate = new Date(year, month-1, day);
```

Обсуждение

Конструктор объекта `Date` не особенно тщательно проверяет входные данные. Проверяйте их сами перед созданием объекта `Date`, поскольку он может посчитать приемлемыми значения, которые для вас неприемлемы. Например, он допускает перенос дней месяца (если передать конструктору день номер 40, то JavaScript просто перенесет эту дату на следующий месяц). Конструктор `Date` также принимает строки, которые могут по-разному преобразовываться в даты на разных компьютерах.

При попытке создать объект `Date` из строки, содержащей нецифровые данные, получим объект `Invalid Date`. Для проверки этого условия можно использовать функцию `isNaN()`:

```
const badDate = '12 bananas';

const convertedDate = new Date(badDate);

if (Number.isNaN(convertedDate)) {
    // Мы окажемся здесь, так как объект Date не создан
} else {
    // Если экземпляр объекта Date корректен, то мы окажемся здесь
}
```

Эта методика работает, поскольку по своей внутренней реализации объекты `Date` являются числами — этот факт подробно рассматривается в рецепте 4.4.

Читайте также

В рецепте 4.6 описывается обратная операция — преобразование объекта `Date` в строку.

4.3. Добавляем дни к дате

Задача

Вычислить дату, которая отстоит от другой даты на заданное число дней вперед или назад.

Решение

Получить номер сегодняшнего дня с помощью `Date.getDate()`, затем изменить его, применив `Date.setDate()`. Объект `Date` достаточно сообразителен, для того чтобы при необходимости перейти на следующий месяц или год:

```
const today = new Date();
const currentDay = today.getDate();

// Какой день наступит через три недели?
today.setDate(currentDay + 21);
console.log(`Three weeks from today is ${today}`);
```

Обсуждение

Метод `setDate()` не ограничен положительными числами. Если использовать отрицательное число, то можно получить дату в прошлом. Есть и другие методы типа `setXxx()` для изменения даты: например, `setMonth()` позволяет перейти на несколько месяцев вперед или назад, а `setHours()` — на несколько часов. Все эти методы, как и `setDate()`, позволяют перейти на следующий день, месяц или год. Например, если добавить к текущему времени 48 часов, то получим дату точно на двое суток позже.

Объект `Date` *изменяемый*, из-за чего его поведение выглядит весьма старомодным. Возможно, в будущих библиотеках JavaScript методы, подобные `setDate()`, станут возвращать новый объект `Date`. Но сейчас они изменяют *текущий* объект `Date`. Это происходит даже в том случае, если дата объявлена как константа. (Ключевое слово `const` не позволяет переменной ссылаться на другой объект `Date`, но не мешает изменить объект `Date`, на который указывает данная ссылка.) Для того чтобы гарантированно избежать потенциальных проблем, необходимо клонировать дату, прежде чем что-то с ней делать. Для этого воспользуйтесь методом `Date.getTime()`, чтобы получить количество миллисекунд, которое является внутренним представлением даты, и создайте на его основе новый объект:

```
const originalDate = new Date();

// Клонировем дату
const futureDate = new Date(originalDate.getTime());

// Изменяем клонированную дату
futureDate.setDate(originalDate.getDate()+21);
console.log(`Three weeks from ${originalDate} is ${futureDate}`);
```

Читайте также

В рецепте 4.5 показано, как вычислить временной интервал между двумя датами.

4.4. Сравнение дат и проверка двух дат на равенство

Задача

Убедиться, что два объекта `Date` описывают одну и ту же календарную дату, или определить, что одна дата является более ранней, чем другая.

Решение

Объекты `Date` можно сравнивать между собой точно так же, как и числа, с помощью операторов `<` и `>`:

```
const oldDay = new Date(1999, 10, 20);
const newerDay = new Date(2021, 1, 1);

if (newerDay > oldDay) {
    // Выражение истинно, так как newerDay идет после oldDay
}
```

Внутри объекта `Date` даты представлены в виде целых чисел. При использовании оператора `<` или `>` даты автоматически преобразуются в числа и сравниваются. При выполнении этого кода мы сравниваем значение `oldDay` в миллисекундах (943 074 000 000) со значением `newerDay` в миллисекундах (1 612 155 600 000).

Оператор равенства (`=`) работает по-другому. Он сравнивает не содержимое объектов, а ссылки на эти объекты. (Другими словами, два объекта `Date` равны между собой только в том случае, если мы сравниваем две переменные, которые ссылаются на один и тот же экземпляр.)

Для того чтобы убедиться, что два объекта `Date` описывают один и тот же момент времени, необходимо вручную преобразовать их в числа. Нагляднее всего это делается с помощью метода `Date.getTime()`, который возвращает число миллисекунд для даты:

```
const date1 = new Date(2021, 1, 1);
const date2 = new Date(2021, 1, 1);

// Результат равен false, так как это разные объекты
console.log(date1 === date2);

// Результат равен true, так как это одна и та же дата
console.log(date1.getTime() === date2.getTime());
```



Несмотря на название метода, `getTime()` возвращает не просто время, а число миллисекунд, точно соответствующее дате и времени объекта `Date`.

Обсуждение

По своему внутреннему представлению объект `Date` — это просто целое число. Точнее, это число миллисекунд, прошедших с 1 января 1970 года. Оно может быть положительным или отрицательным. Другими словами, с помощью объекта `Date` можно представлять даты, относящиеся как к глубокому прошлому (примерно с 271 821 года до н. э.), так и к далекому будущему (до 275 760 года). Это количество миллисекунд можно получить с помощью метода `Date.getTime()`.

Два объекта `Date` равны только в том случае, если они совпадают вплоть до миллисекунды. Если два объекта `Date` соответствуют одной и той же дате, но различаются по времени, то они не равны. Это может стать проблемой, так как не все учитывают, что в объекте `Date` хранится также информация о времени. Такая ошибка часто возникает при создании объекта `Date` для текущего дня (см. рецепт 4.1).

Чтобы избежать данной ошибки, нужно удалить информацию о времени с помощью метода `Date.setHours()`. Несмотря на свое название, метод `setHours()` принимает до четырех параметров, позволяя задавать часы, минуты, секунды и миллисекунды. Для того чтобы создать объект `Date`, содержащий только дату, все эти компоненты нужно установить равными 0:

```
const today = new Date();

// Создаем еще один объект Date с текущей датой
// День остался тот же, но время, возможно, изменилось на миллисекунду
const todayDifferent = new Date();

// Это может быть true или false в зависимости
// от факторов времени, которые вы не контролируете
console.log(today.getTime() === todayDifferent.getTime());

// Удаляем всю информацию о времени
todayDifferent.setHours(0,0,0,0);
today.setHours(0,0,0,0);

// Это всегда равно true, поскольку мы удалили время из обоих экземпляров
console.log(today.getTime() === todayDifferent.getTime());
```


Читайте также

Подробнее о вычислениях дат читайте в рецептах 4.5 и 4.3.

4.5. Вычисление времени, прошедшего между двумя датами

Задача

Вычислить, сколько дней, часов или минут разделяют две даты.

Решение

Поскольку даты представляют собой числа в миллисекундах (см. рецепт 4.4), выполнять с ними вычисления довольно просто. При вычитании одной даты из другой получим разделяющее их количество миллисекунд:

```
const oldDate = new Date(2021, 1, 1);
const newerDate = new Date(2021, 10, 1);

const differenceInMilliseconds = newerDate - oldDate;
```

За исключением случаев тестирования производительности путем измерения времени выполнения коротких операций, миллисекунды — не особенно удобная единица измерения. Вы можете сами преобразовать это значение в более наглядное количество минут, часов или дней:

```
const millisecondsPerDay = 1000*60*60*24;
let differenceInDays = differenceInMilliseconds / millisecondsPerDay;

// Считаем только целые дни
differenceInDays = Math.trunc(differenceInDays);

console.log(differenceInDays);
```

Несмотря на то что эти вычисления дают только целое число дней (поскольку ни в одной дате нет информации о времени), нам все равно нужно применять к результату метод `Math.round()`, чтобы исключить ошибки округления, свойственные математическим вычислениям с плавающей точкой (см. рецепт 3.4).

Обсуждение

При вычислении дат следует помнить о двух возможных ловушках.

- В датах может указываться информация о времени. (Например, в новом объекте `Date`, созданном для текущего дня, содержится время создания этого объекта с точностью до миллисекунд.) Прежде чем вычислять количество дней, удалите данные о времени с помощью метода `setHours()`, как показано в рецепте 4.4.

- Вычисления с двумя датами имеют смысл только в том случае, если они относятся к одному и тому же часовому поясу. В идеале это означает сравнение двух локальных дат или двух дат, представленных в стандарте UTC. Может показаться, что достаточно всего лишь пересчитать даты из одного часового пояса в другой, но здесь часто возникают неожиданные пограничные случаи с переходом на летнее время.

Сейчас появилась предположительная замена устаревшему объекту `Date` — объект `Temporal` (<https://oreil.ly/BAbB2>), целью создания которого является улучшение вычислительных операций с локальными датами и разными часовыми поясами. В настоящее время для операций с датами, выходящими за пределы возможностей объекта `Date`, можно попробовать использовать какую-либо стороннюю библиотеку для обработки дат. Наиболее популярные — *day.js* (<https://github.com/iamkun/dayjs>) и *date-fns* (<https://date-fns.org>).

Однако для вычислительных операций с микроскопическими временными промежутками, которые применяются для профилирования производительности, объект `Date` не подходит. Лучше взять объект `Performance`, который доступен в среде браузера через встроенное свойство `window.performance`. Этот объект позволяет получать метки времени высокого разрешения, с точностью до долей миллисекунд, насколько это поддерживает система, например:

```
// Создаем объект DOMHighResTimeStamp, соответствующий начальному времени
const startTime = window.performance.now();

// (Выполняем какую-либо затратную по времени задачу.)

// Создаем объект DOMHighResTimeStamp, соответствующий времени окончания
const endTime = window.performance.now();

// Вычисляем затраченное время в миллисекундах
const elapsedMilliseconds = endTime - startTime;
```

Результат (`elapsedMilliseconds`) — это не ближайшее целое значение в миллисекундах, а настолько точное число долей миллисекунд, насколько позволяет вычислить данное оборудование.



В Node нет объекта `Performance`, зато есть собственный механизм для извлечения точной информации о времени. Для этого используется глобальный объект `process`, у которого есть метод `process.hrtime.bigint()`. Данный метод считывает время в наносекундах (1 мс = 1 000 000 нс) и возвращает это значение. Для того чтобы получить разницу по времени в наносекундах, достаточно вычесть одно значение, полученное с помощью `process.hrtime.bigint()`, из другого такого значения.

Поскольку очевидно, что счетчик наносекунд будет выдавать очень большие числа, для их хранения следует применять тип данных `BigInt`, описанный в рецепте 3.9.

Читайте также

В рецепте 4.3 показано, как перенести дату вперед или назад, прибавляя к ней или вычитая из нее временной интервал.

4.6. Представление даты в виде форматированной строки

Задача

Преобразовать объект `Date` в форматированную строку.

Решение

Если вывести дату в `console.log()`, то получим ее красиво форматированное строковое представление наподобие *Wed Oct 21 2020 22:17:03 GMT-0400 (Eastern Daylight Time)*. Оно создается с помощью метода `DateTime.toString()`. Это стандартизованное, не привязанное к региональным стандартам строковое представление даты, определяемое стандартом JavaScript (<https://oreil.ly/S0IMb>).



Внутри объекта `Date` информация представлена в виде времени UTC без дополнительных данных о часовом поясе. Когда объект `Date` преобразуется в строку, время UTC преобразуется в локальное время для данного часового пояса, выбранного на компьютере или другом устройстве, на котором выполняется код.

Для того чтобы представить строку с датой в другом формате, можно вызывать один из следующих готовых методов `Date`:

```
const date = new Date(2021, 0, 1, 10, 30);

let dateString;
dateString = date.toString();
// 'Fri Jan 01 2021 10:30:00 GMT-0500 (Eastern Standard Time)'

dateString = date.toTimeString();
// '10:30:00 GMT-0500 (Eastern Standard Time)'

dateString = date.toUTCString();
// 'Fri, 01 Jan 2021 15:30:00 GMT'

dateString = date.toDateString();
// 'Fri Jan 01 2021'

dateString = date.toISOString();
// '2021-01-01T15:30:00.000Z'
```

```
dateString = date.toLocaleDateString();  
    // '1/1/2021, 10:30:00 AM'  
  
dateString = date.toLocaleTimeString();  
    // '10:30:00 AM'
```

Следует учитывать, что при использовании методов `toLocaleString()` и `toLocaleTime()` представление строки зависит от реализации браузера и от параметров, выбранных на данном компьютере. Не стоит рассчитывать, что это представление везде будет одинаковым!

Обсуждение

Есть множество способов преобразовать информацию о дате в строку. Для наглядного представления вполне подходят методы `toXxxString()`. Но если нужно что-то более специфическое или тонко настраиваемое, то, возможно, стоит взять управление объектом `Date` на себя.

Если нужно нечто большее, чем стандартные методы форматирования, то у вас есть два пути. Можно извлечь из даты отдельные временные компоненты, воспользовавшись методами `getXxx()`, описанными в рецепте 4.1, а затем объединить их в строку того формата, который вам нужен, например:

```
const date = new Date(2021, 10, 1);  
  
// Если день месяца меньше 10, то добавляем ведущий 0  
const day = date.getDate().toString().padStart(2, '0');  
  
// Прибавляем к номеру месяца 1, так как в JavaScript  
// их отсчет начинается с 0, и при необходимости добавляем ведущий 0  
const month = (date.getMonth()+1).toString().padStart(2, '0');  
  
// Год всегда четырехзначный  
const year = date.getFullYear();  
  
const customDateString = `${year}.${month}.${day}`;  
// теперь customDateString = '2021.11.01'
```

Это очень гибкий подход, но он вынуждает писать собственный шаблон представления даты. А такое решение неидеально, поскольку усложняет код и оставляет место для новых ошибок.

Если вы хотите применять стандартный формат для соответствующего регионального стандарта, это сильно упростит дело. Для такого преобразования можно взять объект `Intl.DateTimeFormat`. Вот три примера использования строк для представления дат в региональных стандартах США, Великобритании и Японии:

```
const date = new Date(2020, 11, 20, 3, 0, 0);  
  
// Стандартное представление даты в США  
console.log(new Intl.DateTimeFormat('en-US').format(date)); // '12/20/2020'
```

```
// Стандартное представление даты в Великобритании
console.log(new Intl.DateTimeFormat('en-GB').format(date)); // '20/12/2020'
```

```
// Стандартное представление даты в Японии
console.log(new Intl.DateTimeFormat('ja-JP').format(date)); // '2020/12/20'
```

Все эти строки представляют собой только даты, но при создании объекта `Intl.DateTimeFormat()` можно задать множество других параметров. Вот лишь один пример того, как добавить в строку день недели и месяц для Германии:

```
const date = new Date(2020, 11, 20);

const formatter = new Intl.DateTimeFormat('de-DE',
  { weekday: 'long', year: 'numeric', month: 'long', day: 'numeric' });

const dateString = formatter.format(date);
// теперь dateString = 'Sonntag, 20. Dezember 2020'
```

Эти параметры позволяют также добавить в строку информацию о времени, указав часы, минуты и секунды:

```
const date = new Date(2022, 11, 20, 9, 30);

const formatter = new Intl.DateTimeFormat('en-US',
  { year: 'numeric', month: 'numeric', day: 'numeric',
    hour: 'numeric', minute: 'numeric' });

const dateString = formatter.format(date);
// теперь dateString = '12/20/2022, 9:30 AM'
```

Читайте также

В рецепте 2.2 были представлены объект `Intl` и концепция строк языкового стандарта, которые соответствуют различным географическим и культурным регионам. Более подробное описание всех параметров, поддерживаемых объектом `Intl.DateTimeFormat` (всего их 21), читайте в документации MDN (<https://oreil.ly/at36f>). Следует отметить, что реализация некоторых из этих параметров зависит от системы и в некоторых браузерах может не поддерживаться. (Примерами таких параметров являются `timeStyle`, `dateStyle` и `timeZone`, которые мы здесь не рассматривали.) Как обычно, для сложных манипуляций с объектом `Date` стоит поискать стороннюю библиотеку.

ГЛАВА 5

Массивы

Изначально массивы в JavaScript представляли собой отдельный, самостоятельный тип данных. Однако со временем способы взаимодействия с массивами значительно изменились.

Прежде операции с массивами подразумевали большое количество циклов и логики выполнения итераций, а также несколько примитивных методов. Сегодня объект `Array` имеет гораздо больший функционал, включая методы, рассчитанные на функциональный стиль программирования. С помощью этих методов можно фильтровать, сортировать, копировать и преобразовывать данные, не перебирая элементы массива по одному.

В этой главе вы научитесь использовать эти функциональные принципы и узнаете, в каких случаях их стоит избегать. Мы обратим основное внимание на решение задач с помощью самых современных подходов из тех, что доступны в настоящее время.



Когда вы будете выполнять эти примеры в консоли разработчика в браузере, имейте в виду, что вас могут ввести в заблуждение «ленивые» вычисления. Например, как вы думаете, что произойдет, если вывести массив посредством `console.log()`, затем отсортировать его и вывести снова? По идее, мы должны увидеть два по-разному отсортированных массива. Однако на самом деле мы увидим один и тот же массив, одинаково отсортированный дважды. Так происходит потому, что большинство браузеров не трогают элементы массива, пока вы не откроете консоль и не щелкнете на массиве, чтобы просмотреть его содержимое. Один из способов решить эту проблему — обойти весь массив в цикле и вывести в консоль каждый элемент в отдельности. Подробнее об этом читайте в статье «Почему консоль Chrome иногда лжет» (<https://oreil.ly/VDHtm>).

5.1. Проверка того, является ли объект массивом

Задача

Перед выполнением операции с массивом имеет смысл убедиться, что данный объект действительно является массивом.

Решение

Воспользоваться статическим методом `Array.isArray()`:

```
const browserNames = ['Firefox', 'Edge', 'Chrome', 'IE', 'Safari'];

if (Array.isArray(browserNames)) {
    // Мы окажемся здесь, так как browserNames действительно массив
}
```

Обсуждение

Выбор метода `Array.isArray()` очевиден. Проблемы возникают в тех случаях, когда у разработчиков появляется соблазн использовать старый оператор `instanceOf`. По ряду исторических причин у этого оператора есть ряд странных пограничных случаев при работе с массивами (в частности, он возвращает `false` для массива, созданного в другом контексте выполнения, например в другом окне). Метод `isArray()` был создан именно для того, чтобы исправить эту ошибку.

Важно также понимать, что метод `isArray()` проверяет именно то, является ли данный экземпляр объектом `Array`. Если вызвать его для другого типа коллекции (например, для `Map` или `Set`), то `isArray()` вернет `false`. Он вернет `false` даже в том случае, если у этих коллекций такая же семантика, как и у массивов, и даже если в названии коллекции есть слово `Array`, как у `TypedArray` (низкоуровневая оболочка для буфера двоичных данных).

5.2. Перебор всех элементов массива

Задача

Найти наилучший способ перебрать в цикле все элементы массива по порядку.

Решение

Традиционным способом является цикл `for ... of`, который автоматически перебирает все элементы:

```
const animals = ['elephant', 'tiger', 'lion', 'zebra', 'cat', 'dog', 'rabbit'];

for (const animal of animals) {
    console.log(animal);
}
```

В современном JavaScript при обработке массивов все чаще отдают предпочтение функциональным принципам программирования. Для обхода массива в функциональном стиле используется метод `Array.forEach()`. В него передается функция, которая затем вызывается для каждого элемента массива. Также в метод

могут быть переданы еще три параметра, которые могут пригодиться: элемент, индекс элемента и исходный массив, например:

```
const animals = ['elephant', 'tiger', 'lion', 'zebra', 'cat', 'dog', 'rabbit'];

animals.forEach(function(animal, index, array) {
  console.log(animal);
});
```

Этот код можно еще сильнее сократить, используя синтаксис стрелочных функций (см. рецепт 6.2):

```
animals.forEach(animal => console.log(animal));
```

Обсуждение

В таких языках, как JavaScript, часто существует несколько способов выполнения одних и тех же действий. Цикл `for ... of` предоставляет простой синтаксис для обхода массива. Он не позволяет изменять элементы просматриваемого массива, что является безопасным и разумным подходом.

Но иногда бывает нужно нечто другое. Один из самых гибких вариантов — простейший цикл со счетчиком:

```
const animals = ['elephant', 'tiger', 'lion', 'zebra', 'cat', 'dog', 'rabbit'];

for (let i = 0; i < animals.length; ++i) {
  console.log(animals[i]);
}
```

Такой подход открывает лазейку для ошибок выхода за пределы диапазона, которые, оставаясь незамеченными, часто оказываются причиной более серьезных ошибок в современном программировании. Однако в некоторых ситуациях этот цикл все же приходится задействовать — например, при одновременном обходе нескольких массивов (см. рецепт 5.3).

Для обхода массива можно использовать также метод `Array.forEach()`, передав в него функцию, которая будет вызываться для каждого элемента массива. Эта функция может принимать три параметра: текущий элемент массива, индекс текущего элемента и ссылку на исходный массив. Как правило, нужен только один параметр — элемент массива. (С помощью индекса можно изменить значение элемента исходного массива, но это считается плохим стилем программирования.) Если же вы хотите применить функциональный принцип программирования для обхода или изменения массива, обратите внимание на более специализированные методы. Наиболее полезные из них перечислены в табл. 5.1.

Современные практики программирования отдают предпочтение не *итеративным*, а *функциональным подходам*. Преимущество такого подхода состоит в том, что код получается более кратким, обычно лучше читается и меньше подвержен ошибкам. Как правило, функциональный подход обеспечивает также

неизменяемость массива, поскольку подразумевает создание новой копии массива всякий раз, когда вносятся изменения, вместо того чтобы вносить изменения непосредственно в исходный массив. Благодаря такому подходу снижается и вероятность некоторых типов ошибок.

Таблица 5.1. Методы функциональной обработки массивов

Задача	Метод обработки массива	Описан в рецепте
Изменить каждый элемент массива	<code>map()</code>	5.17
Проверить, что все элементы массива удовлетворяют заданному условию	<code>every()</code>	5.19
Проверить, что хотя бы один из элементов массива удовлетворяет заданному условию	<code>some()</code>	5.19
Отобрать элементы массива, которые удовлетворяют заданному условию	<code>filter()</code>	5.9
Отсортировать массив	<code>sort()</code>	5.16
Использовать все элементы массива в некотором вычислении	<code>reduce()</code>	5.18



Как правило, стоит по возможности использовать функциональные методы работы с массивами. Если же такие методы лишь усложняют задачу (так может случиться, если нужно обработать параллельно несколько массивов или выполнить сразу несколько операций с массивом), то можно вернуться к итеративному подходу. Если вы пишете ресурсоемкий код (например, процедуры, которые оперируют сверхбольшими массивами), то лучше применить итеративный подход, так как он обычно работает быстрее. Только не забудьте сначала профилировать оба варианта, чтобы убедиться в том, что разница действительно существенна.

5.3. Проверка равенства двух массивов

Задача

Найти простой способ проверки того, равны ли два массива между собой, то есть одинаково ли их содержимое.

Решение

Самое простое решение этой задачи, оно же самое старое — использовать простейший цикл `for` со счетчиком, обойти одновременно оба массива и сравнить каждую пару элементов. Разумеется, перед началом цикла нужно будет выполнить

несколько проверок — в частности, убедиться, что оба объекта являются массивами, не равны `null` и т. п. Вот короткий код, в котором все эти критерии упакованы в полезную функцию:

```
function areArraysEqual(arrayA, arrayB) {
  if (!Array.isArray(arrayA) || !Array.isArray(arrayB)) {
    // Эти объекты равны null, либо не объявлены,
    // либо не являются массивами
    return false;
  }
  else if (arrayA === arrayB) {
    // Короткий путь: это две ссылки на один и тот же массив
    return true;
  }
  else if (arrayA.length !== arrayB.length) {
    // Массивы разного размера не могут быть равны
    return false;
  }
  else {
    // Теперь можно рассмотреть массивы поэлементно
    for (let i = 0; i < arrayA.length; ++i) {
      // Элементы массива должны быть одного типа
      // и иметь одинаковое содержимое, но если
      // позволяет задача, можно использовать
      // слаботипизированное равенство
      if (arrayA[i] !== arrayB[i]) return false;
    }
    return true;
  }
}
```

Теперь можно сравнивать два массива так:

```
const fruitNamesA = ['apple', 'kumquat', 'grapefruit', 'kiwi'];
const fruitNamesB = ['apple', 'kumquat', 'grapefruit', 'kiwi'];
const fruitNamesC = ['avocado', 'squash', 'red pepper', 'cucumber'];

console.log(areArraysEqual(fruitNamesA, fruitNamesB)); // true
console.log(areArraysEqual(fruitNamesA, fruitNamesC)); // false
```

В этой версии `areArraysEqual()` считается, что если массивы имеют одинаковые элементы, но расположенные в разной последовательности, то они не равны. Массивы строк или чисел можно легко упорядочить с помощью метода `Array.sort()`. Однако не имеет смысла вставлять этот код в метод `areArrayEquals()`, поскольку данный способ подходит не для всех типов данных. Кроме того, он может слишком медленно работать при сравнении очень больших массивов. Вместо этого можно упорядочить массивы перед сравнением:

```
const fruitNamesA = ['apple', 'kumquat', 'grapefruit', 'kiwi'];
const fruitNamesB = ['kumquat', 'kiwi', 'grapefruit', 'apple'];

console.log(areArraysEqual(fruitNamesA.sort(), fruitNamesB.sort())); // true
```

Обсуждение

В программировании смысл равенства часто определяет сам программист. В данном примере `areArraysEqual()` выполняет *нестрогое сравнение*. Если два массива состоят из одних и тех же примитивов или ссылок на одни и те же объекты, и элементы этих массивов расположены в одном и том же порядке, то будет считаться, что они равны. Но при сравнении более сложных *объектов* возникнут неясности.

Например, рассмотрим сравнение следующих двух массивов, в каждом из которых хранится один объект `Date` с одной и той же датой:

```
const datesA = [new Date(2021,1,1)];
const datesB = [new Date(2021,1,1)];

console.log(areArraysEqual(datesA, datesB)); // false
```

Эти массивы не считаются равными, поскольку, несмотря на то что в них хранится одна и та же дата, это разные *экземпляры* `Date`. (Другими словами, это два разных объекта `Date`, в которых просто хранится одна и та же информация.)

Конечно, мы вполне могли бы сравнить содержимое двух объектов `Date`, просто вызвав метод `getTime()`, чтобы преобразовать их временное представление в миллисекунды, как было показано в рецепте 4.4. Но если вы захотите делать это при сравнении массивов, то вам придется написать другую функцию, в которой идентифицировать объекты `Date` с помощью `instanceOf` и затем вызывать для них метод `getTime()`:

```
function areArraysEqual(arrayA, arrayB) {
  if (!Array.isArray(arrayA) || !Array.isArray(arrayB)) {
    return false;
  }
  else if (arrayA === arrayB) {
    return true;
  }
  else if (arrayA.length !== arrayB.length) {
    return false;
  }
  else {
    for (let i = 0; i < arrayA.length; ++i) {
      // Проверяем даты на равенство
      if (arrayA[i] instanceof Date && arrayB[i] instanceof Date) {
        if (arrayA[i].getTime() !== arrayB[i].getTime()) return false;
      }
      else {
        // Используем обычную строгую проверку на равенство
        if (arrayA[i] !== arrayB[i]) return false;
      }
    }
    return true;
  }
}
```

Проблема, рассмотренная в этом примере, касается массивов, в которых могут храниться любые объекты JavaScript. Она существует даже в случае вложенных массивов, поскольку `Array` — это тоже объект. Однако ваши решения этой проблемы будут различаться, поскольку разные объекты требуют разных тестов на равенство.

В заключение стоит отметить, что во многих популярных библиотеках JavaScript реализованы собственные универсальные способы глубокого сравнения массивов. Они могут подходить для ваших данных, а могут и не подходить. Если вы уже используете библиотеку вроде `Lodash` или `Underscore.js`, исследуйте их метод `isEqual()`.

5.4. Разбиение массива на отдельные переменные

Задача

Найти удобный способ одновременно присвоить значения элементов массива нескольким переменным, чтобы не приходилось присваивать значение каждой переменной в отдельности.

Решение

Для одновременного присвоения значений нескольким переменным можно использовать *синтаксис деструктурирования* массива. Нужно написать выражение, которое бы объявляло несколько переменных (слева) и извлекало значения из массива (справа), например:

```
const stateValues = [459, 144, 96, 34, 0, 14];
const [arizona, missouri, idaho, nebraska, texas, minnesota] = stateValues;
console.log(missouri); // 144
```

При деструктурировании массива значения копируются в соответствии с их позицией в массиве. В данном примере это означает, что переменной `arizona` будет присвоено первое значение массива, `missouri` — второе и т. д. Если переменных больше, чем элементов массива, то лишние переменные получат значение `undefined`.

Обсуждение

При деструктурировании массива нет необходимости копировать каждое его значение. Чтобы пропустить ненужные значения, следует поставить дополнительные запятые без имени переменных:

```
const stateValues = [459, 144, 96, 34, 0, 14];

// Получить только три значения из массива
const [arizona, , , nebraska, texas] = stateValues;
console.log(nebraska); // 34
```

Также можно использовать *rest-оператор*, чтобы поместить все оставшиеся значения в новый массив, вместо того чтобы явно присваивать их переменным. Вот пример копирования трех последних элементов в новый массив, который называется `others`:

```
const stateValues = [459, 144, 96, 34, 0, 14];
const [arizona, missouri, idaho, ...others] = stateValues;
console.log(others); // 34, 0, 14
```



Rest-оператор в JavaScript очень похож на spread-оператор (три точки перед переменной). Эти операторы даже в коде выглядят практически одинаково, хотя на самом деле они дополняют друг друга. Rest-оператор отделяет лишние значения и объединяет их в один массив. Spread-оператор распределяет массив или другой итерируемый объект по отдельным переменным.

До сих пор мы видели объявления переменных и присвоение им значений в одном операторе. Но эти действия можно разделить — точно так же, как при создании обычных переменных. Только не забудьте поставить квадратные скобки: они показывают, что мы используем деструктурирование массива:

```
let arizona, missouri, idaho, nebraska, texas, minnesota;
[arizona, missouri, idaho, nebraska, texas, minnesota] = stateValues;
```

Читайте также

Для преобразования массива в список значений *без* присвоения этих значений переменным применяется spread-оператор, описанный в рецепте 5.5.

5.5. Передача массива в функцию, которая принимает список значений

Задача

Есть массив со списком значений, которые нужно передать функции. Но функция принимает не объект массива, а список аргументов.

Решение

Разбить массив на отдельные значения с помощью spread-оператора. Вот пример для метода `Math.max()`:

```
const numbers = [2, 42, 5, 304, 1, 13];

// Так писать нельзя. В результате получим NaN
const maximumFail = Math.max(numbers);
```

```
// А это работает – благодаря spread-оператору
// (Результат – 304)
const maximum = Math.max(...numbers);
```

Обсуждение

Spread-оператор разворачивает массив в список элементов. Технически spread-оператор способен работать с любым итерируемым объектом, включая другие типы коллекций. Вы еще встретите примеры работы spread-оператора в других рецептах этой главы.

Значения, полученные при помощи spread-оператора, могут быть не единственными аргументами, передаваемыми в функцию, — они даже не обязательно должны быть последними аргументами. Spread-оператор вполне можно использовать и так:

```
const numbers = [2, 42, 5, 304, 1, 13];

// Вызываем метод max() для значений массива и еще трех аргументов
const maximum = Math.max(24, ...numbers, 96, 7);
```

Скорее всего, вам не захочется применять такой подход, если последовательность аргументов имеет хоть какое-то значение. Слишком легко столкнуться с ситуацией, когда массив окажется чуть больше или меньше, чем вы ожидали, из-за чего остальные аргументы сместятся со своих позиций и их смысл изменится.

Читайте также

В рецепте 5.7 показан пример использования spread-оператора для слияния двух массивов. В рецепте 5.15 показано, как задействовать spread-оператор при удалении элементов. В рецепте 5.6 демонстрируется копирование массива посредством spread-оператора.

5.6. Клонирование массива

Задача

Создать копию существующего массива.

Решение

С помощью spread-оператора разобьем массив на элементы и затем заполним ими новый массив:

```
const numbers = [2, 42, 5, 304, 1, 13];
const numbersCopy = [...numbers];
```

Еще один хороший способ — использовать метод `Array.slice()` без аргументов, в результате чего массив будет разбит на отдельные элементы:

```
const numbers = [2, 42, 5, 304, 1, 13];
const numbersCopy = numbers.slice();
```

Оба эти способа лучше, чем перебирать элементы массива в цикле и строить новый массив вручную.

Обсуждение

Создание копий массивов — важная операция, так как она позволяет выполнять *недеструктивные изменения*. Например, можно внести изменения в копию массива, не затрагивая исходный массив. Таким образом мы уменьшаем риск нежелательных побочных эффектов, например если тем временем исходный массив используется в других частях кода.

Как и другие ссылочные объекты, массивы не копируются путем присвоения. Например, в следующем коде в итоге получаем две переменные, которые ссылаются на один и тот же участок памяти, в котором размещен объект `Array`:

```
const numbers = [2, 42, 5, 304, 1, 13];
const numbersCopy = numbers;
```

Чтобы корректно скопировать массив, нужно создать дубликаты всех его элементов. Проще всего сделать это посредством `spread`-оператора — впрочем, метод `Array.slice()` работает не хуже.

Оба описанных способа создают неполные копии. Если массив состоит из примитивов — чисел, строк или булевых значений, то полученный массив будет точной копией. Но если в массиве есть объекты, то таким образом будут скопированы не они сами, а лишь ссылки на них. В результате в новом массиве будут храниться ссылки, указывающие на те же объекты, что и в старом массиве. В случае изменения одного из таких объектов в скопированном массиве исходный массив тоже изменится:

```
const objectsOriginal = [{name: 'Sadie', age: 12}, {name: 'Patrick', age: 18}];
const objectsCopy = [...objectsOriginal];
```

```
// Изменяем один из объектов в objectsCopy
objectsCopy[0].age = 14;
```

```
// Наблюдаем те же изменения объекта в objectsOriginal
console.log(objectsOriginal[0].age); // 14
```

Это может быть проблемой, а может и не быть в зависимости от того, как вы собираетесь использовать эти массивы. Если вам нужны несколько копий объектов, чтобы манипулировать ими независимо друг от друга, то это можно сделать несколькими способами.

- Обойти весь массив в цикле `for`, явно создавая новые объекты там, где это необходимо, и добавляя их в новый массив.
- Использовать функцию `Array.map()`. Она хорошо работает для простых объектов, но не выполняет глубокое вложенное клонирование. (Например, если в массиве есть объекты, которые ссылаются на другие объекты, то будут созданы настоящие дубликаты только первого уровня объектов.)
- Использовать вспомогательную функцию из сторонней библиотеки JavaScript, такую как `cloneDeep()` из `Lodash` или `clone()` из `Ramda`.

Вот пример, демонстрирующий возможности метода `Array.map()`. Это немного похоже на магию: сначала с помощью `spread`-оператора разбиваем элемент массива на отдельные свойства (`...element`), а затем создаем из этих элементов новый объект (`{...element}`), который заносится в новый массив:

```
const objectsOriginal = [{name: 'Sadie', age: 12}, {name: 'Patrick', age: 18}];

// Создаем новый массив из скопированных объектов
const objectsCopy = objectsOriginal.map( element => ({...element}) );

// Изменяем один из объектов в objectsCopy
objectsCopy[0].age = 14;

// Проверяем этот объект в objectsOriginal
console.log(objectsOriginal[0].age); // 12
```

Для того чтобы более детально изучить метод `map()`, прочитайте подробное объяснение в рецепте 5.17.



`Spread`-оператор (`...`) выполняет двойную функцию. В первом примере мы видели, как с помощью `spread`-оператора можно разбить массив на отдельные элементы. В примере с методом `Array.map()` `spread`-оператор разбивает объект на отдельные свойства. Подробнее о том, как `spread`-оператор работает с объектами, читайте в рецепте 7.6.

Читайте также

О том, как скопировать *только некоторые* элементы массива, читайте в рецепте 5.8. Чтобы познакомиться с другими способами создания глубоких копий объекта, изучите рецепт 7.11.

5.7. Слияние двух массивов

Задача

Объединить два массива в новый массив.

Решение

Есть два общепринятых способа объединения двух массивов. Проверенный временем (и, скорее всего, самый быстрый) вариант состоит в использовании метода `Array.concat()`. Метод `concat()` применяется к первому массиву, а второй массив передается в этот метод в качестве аргумента. Результатом является третий массив, в котором содержатся все элементы первых двух:

```
const evens = [2, 4, 6, 8];
const odds = [1, 3, 5, 7, 9];

const evensAndOdds = evens.concat(odds);
// теперь evensAddOdds содержит [2, 4, 6, 8, 1, 3, 5, 7, 9]
```

В результате получится массив, в котором сначала идут элементы первого массива (в данном случае `evens`), а затем — элементы второго массива (`odds`). Разумеется, после метода `concat()` можно использовать метод `Array.sort()` (см. рецепт 5.16).

Вместо этого можно применить `spread`-оператор (описанный в рецепте 5.5):

```
const evens = [2, 4, 6, 8];
const odds = [1, 3, 5, 7, 9];

const evensAndOdds = [...evens, ...odds];
```

Преимущество этого способа состоит в том, что код (возможно) получается более интуитивно понятным и легкочитаемым. `Spread`-оператор также отлично подходит для тех случаев, когда нужно объединить более двух массивов или массивы и литеральные значения:

```
const evens = [2, 4, 6, 8];
const odds = [1, 3, 5, 7, 9];

const evensAndOdds = [...evens, 10, 12, ...odds, 11];
```

Тесты производительности показывают, что в данной реализации объединение больших массивов лучше выполнять с помощью `concat()`. Но в большинстве сценариев использования различия в производительности будут незначительными, а иногда и вовсе незаметными.

Обсуждение

После слияния массивов одним из описанных способов в нашем распоряжении оказываются три массива: два исходных и третий, полученный в результате слияния. Если массивы содержат примитивные значения — числа, строки, булевы значения, то в новом массиве будут созданы дубликаты их всех. Но если в исходных массивах содержатся объекты, то будут скопированы только *ссылки* на них. Например, при слиянии массивов, которые содержат объекты `Date`, новые объекты `Date` не создаются. Вместо этого в новом, объединенном массиве появятся

ссылки, указывающие на *уже существующие* объекты `Date`. Если изменить объект `Date` в объединенном массиве, то эти изменения отразятся на исходном массиве:

```
const dates2020 = [new Date(2020,1,10), new Date(2020,2,10)];
const dates2021 = [new Date(2021,1,10), new Date(2021,2,10)];

const datesCombined = [...dates2020, ...dates2021];

// Изменяем дату в новом массиве
datesCombined[0].setYear(2022);

// В первом массиве эта дата тоже изменится
console.log(dates2020[0]); // 2022/02/10
```

Подробнее о различиях между поверхностными и глубокими копиями читайте в рецепте 7.11.

Читайте также

При слиянии массивов нет возможности контролировать способ объединения элементов. Если вы хотите скопировать только часть массива или поместить один массив в другой, используйте метод `slice()` из рецепта 5.8.

5.8. Копирование части массива, выбранной по положению элемента

Задача

Скопировать часть массива, не изменив исходный массив.

Решение

Воспользоваться методом `Array.slice()`, который создает поверхностную копию части существующего массива и возвращает ее в виде нового массива:

```
const animals = ['elephant', 'tiger', 'lion', 'zebra', 'cat', 'dog', 'rabbit', 'goose'];

// Получить часть массива с элемента 4 по элемент 7
const domestic = animals.slice(4, 7);

console.log(domestic); // ['cat', 'dog', 'rabbit']
```

Обсуждение

Метод `slice()` принимает два параметра, соответствующие начальной и конечной позициям. Если пропустить второй параметр, то будет выбран фрагмент

от начальной позиции до конца массива. При вызове `slice(0)` копируется весь массив.

Например, в следующем коде с помощью `slice()` создаются два фрагмента исходного массива, которые затем используются для формирования нового массива:

```
const animals = ['elephant', 'tiger', 'lion', 'zebra', 'cat', 'dog', 'rabbit', 'goose'];

const firstHalf = animals.slice(0, 3);
const secondHalf = animals.slice(4, 7);

// Помещаем в середину еще двух животных
const extraAnimals = [...firstHalf, 'emu', 'platypus', ...secondHalf];
```

Этот пример может показаться искусственным, поскольку номера индексов заданы жестко. Но данный метод можно применять в сочетании с поиском по массиву и методом `findIndex()` (см. рецепт 5.13), чтобы найти место, в котором нужно разорвать массив.



Метод `slice()` легко спутать с методом `splice()`, который используется для замены или удаления фрагмента массива. В отличие от `slice()`, `splice()` вносит изменения в исходный массив. В современном программировании считается, что лучше заблокировать объекты, по возможности сделать их неизменяемыми (для чего задействуется `const`) и создавать новые объекты, содержащие изменения. Поэтому лучше придерживаться `slice()`, и использовать `splice()` только в тех случаях, когда на то есть веские причины (например, разница в производительности, которая важна для данного сценария применения).

Читайте также

В рецепте 5.15 показан способ удаления фрагментов массива с помощью `slice()`.

5.9. Извлечение из массива элементов, удовлетворяющих заданному условию

Задача

Найти все элементы массива, удовлетворяющие некоторому условию, и скопировать их в новый массив.

Решение

Проверить все элементы массива с помощью метода `Array.filter()`:

```
function startsWithE(animal) {
  return animal[0].toLowerCase() === 'e';
}
```

```
const animals = ['elephant', 'tiger', 'emu', 'zebra', 'cat', 'dog', 'eel',  
  'rabbit', 'goose', 'earwig'];  
const animalsE = animals.filter(startsWithE);  
console.log(animalsE); // ["elephant", "emu", "eel", "earwig"]
```

Этот пример специально сделан таким многословным, чтобы вы могли рассмотреть отдельные части решения. Для каждого элемента массива вызывается функция, переданная в `filter()`. В данном случае это означает, что функция `startsWithE()` будет вызвана 10 раз — всякий раз для другой строки. Каждый элемент, для которого `filter()` вернет `true`, будет добавлен в новый массив.

Вот тот же пример, но записанный более компактно, с помощью стрелочной функции. Теперь логика фильтра используется в том же месте кода, где она определена:

```
const animals = ['elephant', 'tiger', 'emu', 'zebra', 'cat', 'dog', 'eel',  
  'rabbit', 'goose', 'earwig'];  
const animalsE = animals.filter(animal => animal[0].toLowerCase() === 'e');
```

Обсуждение

В этом примере `filter()` проверяет каждый элемент массива на то, начинается ли он с буквы *e*. Точно так же можно отобрать числа, попадающие в заданный диапазон, или объекты, имеющие определенные значения свойств.

Метод `filter()` относится к новым методам работы с массивами, которые пришли на смену старому итеративному коду и соответствуют функциональному принципу программирования. Ничто не мешает обойти массив в цикле `for`, проверить каждый элемент и, если он соответствует критерию, вставить его в новый массив посредством `Array.push()`. Но решив эту задачу с помощью метода `filter()`, вы, скорее всего, будете вознаграждены более компактным кодом и более простым тестированием.

Читайте также

В нескольких рецептах этой главы описаны сходные методы функциональной обработки массивов. Так, в рецепте 5.17 показано, как преобразовать все элементы массива, а в рецепте 5.18 — как выполнить вычисление, результат которого основан на всех значениях массива.

5.10. Очистка массива

Задача

Удалить из массива все элементы — либо чтобы освободить память, либо для повторного использования массива.

Решение

Присвоить свойству массива `length` значение `0`:

```
const numbers = [2, 42, 5, 304, 1, 13];  
numbers.length = 0;
```

Обсуждение

Один из самых простых способов получить новый массив — создать пустой массив и присвоить его переменной:

```
myArray = [];
```

Но у такого варианта есть несколько ограничений. Во-первых, поскольку здесь создается новый объект массива, этот вариант не подходит для определения массива с ключевым словом `const`. Это мелкий нюанс, но в современном программировании предпочтительнее использовать `const`, а не `let`, чтобы уменьшить вероятность появления ошибок в коде. Во-вторых, при таком присвоении старый массив не уничтожается. Если есть другая переменная, которая ссылается на него, то он продолжит существовать и занимать место в памяти.

Альтернативный вариант — многократный вызов метода `Array.pop()`. При каждом вызове `pop()` удаляется последний элемент массива, так что можно очистить массив, просто вызывая `pop()` в цикле, до тех пор пока он не опустеет. Однако трюк с изменением значения `length` делает то же самое и требует всего одного оператора. Иногда разработчики упускают из виду этот прием, так как считают, что свойство `length` неизменяемое, как во многих других языках. Но в JavaScript изменение свойства массива `length` позволяет уменьшить размер массива и отбросить лишние элементы.

Есть и другие интересные способы использования `length`. Например, уменьшив значение `length` (но не до нуля), можно отрезать лишь часть массива, но не очистить его полностью. А увеличив значение `length`, можно добавить в конец массива пустые элементы:

```
const numbers = [2, 42, 5, 304, 1, 13];  
numbers.length = 3;  
  
console.log(numbers); // [2, 42, 5]  
numbers.length = 5;  
console.log(numbers); // [2, 42, 5, undefined, undefined]
```

5.11. Удаление дубликатов

Задача

Гарантировать уникальность всех значений массива, удалив дубликаты.

Решение

Создать новый объект `Set` и заполнить его элементами массива. В объекте `Set` дубликаты удаляются автоматически. Затем преобразовать `Set` обратно в массив:

```
const numbersWithDuplicats = [2, 42, 5, 42, 304, 1, 13, 2, 13];

// Создаем объект Set с уникальными значениями
// (дубликаты 42, 2 и 13 удаляются)
const uniqueNumbersSet = new Set(numbersWithDuplicats);

// Преобразуем Set снова в массив (теперь в нем шесть элементов)
const uniqueNumbersArray = Array.from(uniqueNumbersSet);
```

Теперь, когда вы поняли принцип, можно сжать этот код до одного выражения со `spread`-оператором:

```
const numbersWithDuplicats = [2, 42, 5, 42, 304, 1, 13, 2, 13];

const uniqueNumbers = [...new Set(numbersWithDuplicats)];
```

Обсуждение

Объект `Set` — это специальный тип коллекции, в которой игнорируются повторяющиеся значения. Объект `Set` также является быстрым и эффективным способом удаления дубликатов из массива. Этот способ (преобразовать массив в `Set`, а потом `Set` — снова в массив) гораздо эффективнее, чем обходить массив в поисках дубликатов с помощью `findIndex()`.

При поиске дубликатов `Set` использует проверку, подобную строгому равенству, так что `3` и `'3'` не считаются дубликатами. Исключением из этого поведения `Set` является `NaN`: два значения `NaN` считаются дубликатами, несмотря на то что обычно результат сравнения `NaN === NaN` равен `false`.

Читайте также

В этом примере задействован `spread`-оператор, описанный в рецепте 5.5. Подробнее об объекте `Set` читайте в рецепте 5.20.

5.12. Сведение двумерного массива

Задача

Свести двумерный массив к одномерному списку.

Решение

Воспользоваться методом `Array.flat()`:

```
const fruitArray = [];  
  
// Добавляем в fruitArray три элемента.  
// Каждый из этих элементов – массив строк  
fruitArray[0] = ['strawberry', 'blueberry', 'raspberry'];  
fruitArray[1] = ['lime', 'lemon', 'orange', 'grapefruit'];  
fruitArray[2] = ['tangerine', 'apricot', 'peach', 'plum'];  
  
const fruitList = fruitArray.flat();  
// Теперь fruitList состоит из 11 элементов,  
// каждый из которых представляет собой строку
```

Обсуждение

Рассмотрим двумерный массив, подобный следующему:

```
const fruitArray = [];  
fruitArray[0] = ['strawberry', 'blueberry', 'raspberry'];  
fruitArray[1] = ['lime', 'lemon', 'orange', 'grapefruit'];  
fruitArray[2] = ['tangerine', 'apricot', 'peach', 'plum'];
```

Каждый элемент `fruitArray` в свою очередь является массивом. Например, `fruitArray[0]` содержит три строки с названиями разных ягод, в `fruitArray[1]` хранятся цитрусовые, а в `fruitArray[2]` — косточковые.

Для преобразования `fruitArray` можно было бы воспользоваться методом `concat()` — вызывать `concat()` для первого вложенного массива, передав методу следующий вложенный массив, и т. д.:

```
const fruitList = fruitArray[0].concat(fruitArray[1], fruitArray[2], fruitArray[3]);
```

Если в массиве много элементов, то такой подход утомителен и чреват ошибками. Вместо этого можно было бы использовать цикл или рекурсию, но эти варианты не менее трудоемки. В методе `flat()` реализована та же логика, и он автоматически выполняет слияние всех элементов массива. В качестве необязательного аргумента `depth` метод `flat()` принимает глубину сведения, которая по умолчанию равна 1. Если увеличить это значение, то можно выполнять сведение массивов с более глубоким вложением. Предположим, что у нас есть массив, в котором содержатся вложенные массивы, которые, *в свою очередь*, содержат вложенные массивы. В этом случае если присвоить аргументу `depth` значение 2, то будет выполнено сведение обоих уровней и мы получим общий список:

```
// Массив, внутри которого находится несколько  
// уровней вложенных массивов  
const threeDimensionalNumbers = [1, [2, [3, 4, 5], 6], 7];  
  
// Сведение по умолчанию  
const flat2D = threeDimensionalNumbers.flat(1);  
// теперь flat2D = [1, 2, [3, 4, 5], 6, 7]  
  
// Сведение двух уровней  
const flat1D = threeDimensionalNumbers.flat(2);
```

```
// теперь flat1D = [1, 2, 3, 4, 5, 6, 7]

// Сведение всех уровней, сколько бы их ни было
const flattest = threeDimensionalNumbers.flat(Infinity);
```

Аргумент `depth` задает максимальный уровень, который может быть использован при сведении массивов. Вы ничем не рискуете, если значение `depth` окажется больше, чем степень вложенности данного массива.

5.13. Точный поиск элементов массива

Задача

Найти в массиве определенное значение. Иногда достаточно лишь узнать, есть ли оно в массиве, а иногда нужно получить его позицию.

Решение

Воспользоваться одним из методов поиска в массиве — `indexOf()`, `lastIndexOf()` или `includes()`:

```
const animals = ['dog', 'cat', 'seal', 'elephant', 'walrus', 'lion'];
console.log(animals.indexOf('elephant'));      // 3
console.log(animals.lastIndexOf('walrus'));    // 4
console.log(animals.includes('dog'));           // true
```

Этот способ работает только в случае примитивных значений (обычно это числа, строки и булевы значения). Для поиска объектов нужно использовать метод `Array.find()` (рецепт 5.14).

Обсуждение

В оба метода — и `indexOf()`, и `lastIndexOf()` — передается искомое значение, которое затем сравнивается с каждым элементом массива. Если совпадение найдено, то возвращается индекс найденного элемента массива. Если же не найдено, то оба метода возвращают `-1`.

Метод `indexOf()` просматривает массив в порядке возрастания индексов и возвращает первый найденный элемент (другими словами, начинает поиск с начала массива и двигается вперед). Метод `lastIndexOf()`, наоборот, начинает поиск с конца массива. Различие между методами становится заметным, если в массиве одно и то же значение встречается более одного раза:

```
const animals = ['dog', 'cat', 'seal', 'walrus', 'lion', 'cat'];

console.log(animals.indexOf('cat'));           // 1
console.log(animals.lastIndexOf('cat'));       // 5
```


И `indexOf()`, и `lastIndexOf()` в качестве необязательного аргумента принимают начальный индекс. Он определяет позицию, с которой должен начинаться поиск:

```
const animals = ['dog', 'cat', 'seal', 'walrus', 'lion', 'cat'];

console.log(animals.indexOf('cat', 2));      // 5
console.log(animals.lastIndexOf('cat', 4));  // 1
```

Может показаться, что если перебрать в цикле все индексы, возвращаемые `indexOf()`, в порядке возрастания, то получим все искомые элементы массива. Но вместо того чтобы писать подобный шаблонный код, лучше воспользоваться методом `filter()` — он быстро и безболезненно создает массив, в который входят все элементы исходного массива, соответствующие заданному критерию поиска (см. рецепт 5.9).

И еще. Важно понимать, что в методах `indexOf()`, `lastIndexOf()` и `includes()` для проверки совпадений используется оператор `===`. Это значит, что там не выполняется преобразование типов (3 не равно '3'). Кроме того, если в массиве содержатся объекты, то сравнивается не их содержимое, а ссылки на них. Если нужно изменить способ сравнения или задействовать другую проверку при поиске, то следует взять метод `findIndex()` (см. рецепт 5.14).

Читайте также

Для настраиваемого поиска используйте методы `find()` и `findIndex()`, описанные в рецепте 5.14.

5.14. Поиск элементов массива, удовлетворяющих заданному критерию

Задача

Найти элемент массива, удовлетворяющий заданному критерию. Например, это может быть объект с заданным свойством.

Решение

Воспользоваться одним из функциональных методов поиска в массиве — `find()` или `findIndex()`. Оба они принимают функцию, которая проверяет элемент массива на соответствие критерию поиска. Поиск продолжается до тех пор, пока не будет найден соответствующий элемент.

В следующем примере будет найдено первое число, большее 10:

```
const nums = [2, 4, 19, 15, 183, 6, 7, 1, 1];

// Находим первое число, большее 10
```

```
const bigNum = nums.find(element => element > 10);  
console.log(bigNum); // 19 (первое подходящее)
```

Если нужно не просто найти подходящий элемент, но и узнать его положение, то следует использовать метод `findIndex()`:

```
const nums = [2, 4, 19, 15, 183, 6, 7, 1, 1];  
const bigNumIndex = nums.findIndex(element => element > 100);  
console.log(bigNumIndex); // 4 (индекс первого подходящего элемента)
```

Если подходящий элемент не найден, то `find()` возвращает `undefined`, а `findIndex()` возвращает `-1`.

Обсуждение

В методы `find()` и `findIndex()` передается имя функции, которая получает до трех параметров — элемент массива на текущей итерации, индекс этого элемента и сам массив. Синтаксис стрелочных функций позволяет писать более рациональный код, поместив определение функции обратного вызова в том месте, где она используется.

Но главное достоинство методов `find()` и `findIndex()` проявляется там, где необходимо описать более сложные условия. Представьте себе, что нужно написать код, чтобы найти первую дату в определенном году:

```
// Напомним: конструктор Date принимает номер месяца  
// с отсчетом от нуля, так что месяцу номер 10  
// соответствует 11-й месяц года — ноябрь  
const dates = [new Date(2021, 10, 20), new Date(2020, 3, 12),  
  new Date(2020, 5, 23), new Date(2022, 3, 18)];  
  
// Найти первую дату 2020 года  
const matchingDate = dates.find(date => date.getFullYear() === 2020);  
  
console.log(matchingDate); // 'Sun Apr 12 2020 ...'
```

При таком подходе использовать метод `indexOf()` невозможно, потому что нам нужно проверить *свойства* элемента массива. (Вообще говоря, метод `indexOf()` не позволяет даже проверять объекты `Date` на равенство, так как сравнивает лишь ссылки на них.)

Читайте также

Если вы захотите написать функцию поиска, которая выдавала бы несколько результатов, то вам, скорее всего, подойдет функция `filter()`, описанная в рецепте 5.9. Подробнее о синтаксисе стрелочных функций читайте в рецепте 6.2.

5.15. Удаление и замена элементов массива

Задача

Найти все элементы массива, равные заданному значению, и удалить их либо заменить на что-то другое.

Решение

Вначале с помощью `indexOf()` выяснить, где расположен элемент, который вы хотите удалить. Далее возможны два подхода.

В небольших задачах самым наглядным решением будет создать новый массив, не включающий в себя нежелательный элемент. Чтобы построить такой массив, можно воспользоваться методом `slice()` и `spread`-оператором:

```
const animals = ['dog', 'cat', 'seal', 'walrus', 'lion', 'cat'];

// Находим позицию элемента 'walrus'
const walrusIndex = animals.indexOf('walrus');

// Создаем новый массив, состоящий из двух частей
// старого: до элемента 'walrus' и после него
const animalsSliced =
  [...animals.slice(0, walrusIndex), ...animals.slice(walrusIndex+1)];

// теперь animalsSliced выглядит так: ['dog', 'cat', 'seal', 'lion', 'cat']
```

Обсуждение

Другой подход состоит в том, чтобы изменять старый массив, а не создавать копию и вносить изменения в нее. Для больших массивов это может работать быстрее. Однако чем больше изменений вы будете допускать, тем сложнее станет код и тем труднее впоследствии будет его обслуживать и отлаживать.

Для того чтобы внести изменения в существующий массив, нужно воспользоваться методом с похожим названием, но совершенно другим назначением — `splice()`. Он позволяет удалить из массива произвольное число элементов, начиная с заданной позиции:

```
const animals = ['dog', 'cat', 'seal', 'walrus', 'lion', 'cat'];

// Находим позицию элемента 'walrus'
const walrusIndex = animals.indexOf('walrus');

// Начиная с walrusIndex, удаляем один элемент
animals.splice(walrusIndex, 1);

// Теперь animals = ['dog', 'cat', 'seal', 'lion', 'cat']
```

Первым аргументом метода `splice()` является индекс, начиная с которого будут удаляться элементы. Это единственный обязательный аргумент. Если больше

ничего не передавать, то будут удалены все элементы с заданного индекса до конца массива:

```
const animals = ['cat', 'walrus', 'lion', 'cat'];  
  
// Удаляем все элементы массива, начиная с 'lion'  
animals.splice(2);  
// Теперь animals = ['cat', 'walrus']
```

Необязательный второй аргумент метода `splice()` — это количество удаляемых элементов. Третий, также необязательный, аргумент — это набор элементов, которые нужно *вставить* на место удаленных:

```
const animals = ['cat', 'walrus', 'lion', 'cat'];  
  
// Удаляем один элемент и вставляем на его место два других  
animals.splice(2, 1, 'zebra', 'elephant');  
// Теперь animals = ['cat', 'walrus', 'zebra', 'elephant', 'cat']
```

Если использовать метод `indexOf()` в цикле, то можно удалить и заменить несколько элементов, удовлетворяющих критерию поиска. Но для этой цели лучше подходит метод `filter()` — он, как правило, обеспечивает более понятный код, так как позволяет определить функцию, отбирающую элементы, которые следует сохранить (см. рецепт 5.9).

5.16. Сортировка массива объектов по заданному свойству

Задача

Отсортировать массив, в котором содержатся объекты, по одному из свойств этих объектов.

Решение

Для упорядочения массива используется метод `Array.sort()`. Например, он позволяет упорядочить массив чисел в порядке возрастания или массив строк — по алфавиту. Но мы не обязаны ограничиваться только стандартной системой сортировки массивов. Вместо этого можно передать методу `sort()` свою функцию сравнения, которая будет применяться для упорядочения элементов массива.

Функция сравнения получает два аргумента, соответствующие двум элементам массива, сравнивает их и возвращает число, которое определяет результат. Если значения признаны равными, то возвращается `0`, если первое значение меньше второго, то `-1`, а если первое значение больше второго, то `1`.

Вот простая реализация сортировки массива объектов, которые содержат персональные данные:

```
const people = [
  { firstName: 'Joe', lastName: 'Khan', age: 21 },
  { firstName: 'Dorian', lastName: 'Khan', age: 15 },
  { firstName: 'Tammy', lastName: 'Smith', age: 41 },
  { firstName: 'Noor', lastName: 'Biles', age: 33 },
  { firstName: 'Sumatva', lastName: 'Chen', age: 19 }
];

// Отсортировать по возрасту от самых молодых до самых старых
people.sort( function(a, b) {
  if (a.age < b.age) {
    return -1;
  } else if (a.age > b.age) {
    return 1;
  } else {
    return 0;
  }
});
console.log(people);
// Теперь объекты расположены в таком порядке:
// Dorian, Sumatva, Joe, Noor, Tammy
```

Здесь кое-что можно написать короче. Технически мы можем возвращать вместо -1 любое отрицательное число, а вместо 1 — любое положительное. Это позволяет сделать функцию сравнения гораздо короче:

```
people.sort(function(a, b) {
  // Чтобы упорядочить от младших к старшим,
  // вычитаем один возраст из другого
  return a.age - b.age;
});
```

В сочетании с синтаксисом стрелочных функций получается еще короче:

```
people.sort((a,b) => a.age - b.age);
```

Иногда при выполнении сортировки можно применять уже существующие методы сравнения. Например, для того чтобы отсортировать массив персональных данных по фамилиям, не нужно изобретать велосипед — достаточно воспользоваться методом `String.localeCompare()`:

```
people.sort((a,b) => a.lastName.localeCompare(b.lastName));
console.log(people);
// Теперь массив отсортирован так: Noor, Sumatva, Joe, Dorian, Tammy
```

Обсуждение

Метод `sort()` изменяет *уже имеющийся* массив. Этим он отличается от большинства других методов работы с массивами, которые мы используем, — они возвращают измененные копии, не затрагивая исходный массив. Если такое поведение вам не подходит, можно клонировать массив перед сортировкой, как описано в рецепте 5.6.

5.17. Преобразование элементов массива

Задача

Преобразовать каждый элемент массива, используя одну и ту же информацию, и объединить полученные значения в новый массив.

Решение

Воспользоваться методом `Array.map()`, передав ему функцию, которая будет осуществлять изменения. Метод `map()` обходит весь массив, применяя заданную функцию к каждому элементу, и объединяет возвращаемые функцией значения в новый массив.

Вот пример, в котором с помощью данного подхода массив десятичных чисел преобразуется в массив их шестнадцатеричных эквивалентов (методом, описанным в рецепте 3.6):

```
const decArray = [23, 255, 122, 5, 16, 99];  
  
// С помощью метода toString() преобразуем значения в шестнадцатеричные  
const hexArray = decArray.map( element => element.toString(16) );  
  
console.log(hexArray); // ['17', 'ff', '7a', '5', '10', '63']
```

Описание

Как правило, функция `map()` применяется только к элементам массива. Но функция обратного вызова может принимать еще два элемента: индекс элемента и исходный массив. Используя эти детали, технически возможно применять `map()` для изменения *исходного* массива. Однако такая методика считается антишаблоном проектирования — другими словами, если вы не планируете применять новый массив, возвращаемый методом `map()`, то не следует задействовать `map()`. Вместо этого можно взять метод `forEach()` (рецепт 5.2) или же просто обойти массив в цикле.

5.18. Использование всех элементов массива в одном вычислении

Задача

Задействовать все значения массива в некотором итоговом вычислении, таком как вычисление суммы или среднего значения.

Решение

Конечно, можно было бы обойти массив в цикле. Но есть более наглядное решение — использовать метод `Array.reduce()` с функцией обратного вызова. Эта

функция, называемая *функцией свертки* (reducer function), вызывается для каждого элемента массива. Для вычисления промежуточной суммы нам понадобится *аккумулятор* — значение, которое метод `reduce()` будет изменять каждый раз до завершения вычислений.

Предположим, например, что нужно вычислить сумму массива чисел. При каждом вызове функции свертки в нее будет передаваться аккумулятор с промежуточной суммой. Функция будет прибавлять к этой сумме значение текущего элемента и возвращать новую сумму:

```
const reducerFunction = function (accumulator, element) {
  // Прибавляем текущее значение к промежуточной сумме,
  // которая хранится в аккумуляторе
  const newTotal = accumulator + element;
  return newTotal;
}
```

Когда функция свертки будет вызвана для *следующего* элемента, в аккумуляторе будет храниться вычисленная ранее промежуточная сумма.

Теперь можно использовать эту функцию для суммирования всех элементов массива:

```
const numbers = [23, 255, 122, 5, 16, 99];

// Второй аргумент (0) задает начальное значение аккумулятора.
// Если не задать начальное значение,
// то аккумулятору автоматически присваивается
// значение первого элемента массива.
const total = numbers.reduce(reducerFunction, 0);
console.log(total); // 520
```

Когда функция свертки вызывается для последнего элемента, она делает последнее вычисление. Возвращаемое значение является результатом, полученным от `reduce()`.

Если вам понятно, как работает `reduce()`, то можно сделать код короче и лаконичнее при помощи встроенной стрелочной функции. Вот пример применения `reduce()` для вычисления суммы квадратов, среднего арифметического и максимального значения:

```
const numbers = [23, 255, 122, 5, 16, 99];

// Функция свертки прибавляет к аккумулятору
// квадрат значения текущего элемента
const totalSquares = numbers.reduce( (acc, val) => acc + val**2, 0);
// totalSquares = 90520

// Функция свертки прибавляет к аккумулятору
// значение текущего элемента
const average = numbers.reduce( (acc, val) => acc + val, 0) / numbers.length;
// average = 86.66...

// Функция свертки возвращает большее из двух значений:
```

```
// аккумулятора и текущего элемента
const max = numbers.reduce( (acc, val) => acc > val ? acc: val);
// max = 255
```

Обсуждение

Метод `reduce()` может показаться более сложным, чем другие функциональные методы работы с массивами, такие как `map()` (см. рецепт 5.9), `filter()` (см. рецепт 5.9) или `sort()` (см. рецепт 5.16). Разница состоит в том, что здесь нужно тщательно продумать, какие данные будут сохраняться после каждого вызова функции. Следует учитывать, что в аккумуляторе можно хранить произвольный объект с несколькими свойствами — это позволяет отслеживать произвольное количество информации. Также в функцию свертки можно передавать еще два параметра: `index` (индекс текущего элемента массива) и `array` (весь обрабатываемый массив). Однако будьте осторожны: если переборщить с использованием `reduce()`, можно легко получить код, в котором будет трудно разобраться другим программистам.

Читайте также

Еще один способ узнать максимальное значение в массиве чисел — использовать `Math.max()` в сочетании со `spread`-оператором, который превратит массив в список аргументов (см. рецепт 5.5).

5.19. Проверка содержимого массива

Задача

Убедиться, что содержимое массива удовлетворяет заданному критерию.

Решение

Использовать метод `Array.every()`, чтобы убедиться, что каждый элемент массива проходит заданную проверку. Например, следующий код позволяет убедиться (при помощи регулярного выражения), что каждый элемент массива состоит только из букв латинского алфавита:

```
// Проверяющая функция
function containsLettersOnly(element) {
    const textExp = /^[a-zA-Z]+$/;
    return textExp.test(element);
}

// Проверка массива
const mysteryItems = ['**', 123, 'aaa', 'abc', '-', 46, 'AAA'];
let result = mysteryItems.every(containsLettersOnly);
console.log(result); // false

// Проверка другого массива
const mysteryItems2 = ['elephant', 'lion', 'cat', 'dog'];
```



```
result = mysteryItems2.every(containsLettersOnly);  
console.log(result); // true
```

Если же проверку должен проходить хотя бы один элемент массива, то можно использовать другой метод — `Array.some()`. Например, следующий код позволяет убедиться, что как минимум один элемент массива представляет собой строку, состоящую из букв латинского алфавита:

```
const mysteryItems = new Array('**', 123, 'aaa', 'abc', '-', 46, 'AAA');  
  
// Проверяющая функция  
function testValue (element) {  
    const textExp = /^[a-zA-Z]+$/;  
    return textExp.test(element);  
}  
  
// Выполняем проверку  
const result = mysteryItems.some(testValue);  
console.log(result); // true
```

Обсуждение

В отличие от многих других методов работы с массивами, использующих функции обратного вызова, методы `every()` и `some()` не всегда обрабатывают все элементы массива. Они обходят массив лишь до тех пор, пока это необходимо в соответствии с их назначением.

Как видно из представленных примеров, в методах `every()` и `some()` может быть использована одна и та же функция обратного вызова. Разница состоит только в том, что в случае с `every()` обработка массива завершается после того, как функция вернет `false`, после чего метод также возвращает `false`. Метод же `some()` продолжает проверку элементов массива, до тех пор пока функция обратного вызова не вернет `true`. После этого перебор элементов прекращается, и метод возвращает `true`. Если же функция обратного вызова будет применена ко всем элементам массива и ни разу не вернет `true`, то `some()` вернет `false`.

Читайте также

Подробнее о синтаксисе регулярных выражений, использованном для построения шаблона проверки строк в этом примере, читайте в рецепте 2.10.

5.20. Построение коллекции недублирующихся значений

Задача

Построить массивоподобный объект, каждое значение в котором существует только в одном экземпляре.

Решение

Создать объект `Set`. Он тихо, не сообщая об ошибке, игнорирует попытки добавить в него элемент, если уже содержит элемент с таким значением.

Объект `Set` — это не массив, но, подобно массиву, является итерируемой коллекцией элементов. В `Set` можно добавлять элементы по одному с помощью метода `add()` либо сразу несколько, передав в конструктор `Set` целый массив:

```
// Начинаем с шести элементов
const animals = new Set(['elephant', 'tiger', 'lion', 'zebra', 'cat', 'dog']);

// Добавляем еще два
animals.add('rabbit');
animals.add('goose');

// Ничего не происходит, так как такой элемент уже есть в Set
animals.add('tiger');

// Обходим Set точно так, как если бы это был массив
for (const animal of animals) {
  console.log(animal);
}
```

Обсуждение

Объекты `Set` — это не массивы. В отличие от класса `Array` с его тремя десятками полезных методов, возможности `Set` значительно скромнее. В `Set` можно вставить элемент с помощью метода `add()`, удалить с помощью `delete()`, проверить существование элемента с помощью `has()` и удалить все элементы с помощью `clear()`. В `Set` нет методов сортировки, фильтров, преобразования и копирования.

Но при желании объект `Set` можно обработать как массив — это нетрудно сделать, выполнив соответствующее преобразование, достаточно лишь передать объект `Set` в статический метод `Array.from()`:

```
// Преобразование массива в Set
const animalSet = new Set(['elephant', 'tiger', 'zebra', 'cat', 'dog']);

// Преобразование Set в массив
const animalArray = Array.from(animalSet);
```

В сущности, можно преобразовывать `Set` в `Array` и наоборот сколько угодно раз без потерь, за исключением возможного снижения производительности, если список элементов очень длинный.



Для подсчета количества элементов в коллекции `Set` или `Map` используется свойство `size` — в отличие от массивов, где для этого существует свойство `length`.

5.21. Создание коллекции элементов, индексируемой по ключу

Задача

Создать коллекцию, каждый элемент которой отмечен уникальной строкой-ключом.

Решение

Использовать объект `Map`. Каждый элемент в нем имеет индекс в виде уникального ключа (как правило — но не обязательно — в виде строки). Для того чтобы добавить в `Map` новый элемент, применяется метод `set()`. Чтобы получить определенный элемент, требуется однозначно идентифицировать его по ключу:

```
const products = new Map();

// Добавляем три элемента
products.set('RU007', {name: 'Rain Racer 2000', price: 1499.99});
products.set('STKY1', {name: 'Edible Tape', price: 3.99});
products.set('P38', {name: 'Escape Vehicle (Air)', price: 2999.00});

// Проверяем, есть ли в коллекции элементы с такими ключами
console.log(products.has('RU007')); // true
console.log(products.has('RU494')); // false

// Получаем элемент
const product = products.get('P38');
if (typeof product !== 'undefined') {
  console.log(product.price); // 2999
}

// Удаляем элемент Edible Tape
products.delete('STKY1');

console.log(products.size); // 2
```

Обсуждение

При добавлении элементов в объект `Map` всегда используйте метод `set()`. Не попадитесь в такую ловушку:

```
const products = new Map();

// Не делайте так!
products['RU007'] = {name: 'Rain Racer 2000', price: 1499.99};
```

На первый взгляд может показаться, что это работает (подобный синтаксис используется во многих других языках для создания коллекций типа «имя — значение»),

однако на самом деле вместо коллекции `Map` будет создан обычный объект `Map` со свойством `RU007`. Такие свойства не будут появляться при обходе `Map` в цикле `for ... of` и не будут видимы методам `has()` и `get()`.

У объекта `Map` довольно скромный набор методов управления содержимым: `set()`, `get()`, `has()` и `delete()`. Если вы захотите использовать функционал объекта `Array`, можно без труда преобразовать `Map` в массив с помощью статического метода `Array.from()`:

```
const productArray = Array.from(products);

console.log(productArray[0]);
// ['RU007', {name: 'Rain Racer 2000', price: 1499.99}]
```

Можно предположить, что `productArray` в этом примере будет содержать коллекцию объектов с информацией о товаре, но это не совсем так. На самом деле каждый элемент `productsArray` — это *отдельный* массив, состоящий из двух элементов, первый из которых — ключ (в данном случае `RU007`), а второй — значение (объект с информацией о товаре).

Иногда при преобразовании объекта `Map` в массив не нужно сохранять значения ключей — эти ключи либо не важны, либо дублируют одно из свойств элементов. В таких случаях при копировании данных из `Map` нужно преобразовать коллекцию, отбросив значения ключей. Вот как это работает:

```
const productArray = Array.from(products, ([name, value]) => value);

console.log(productArray[0]);
// {name: 'Rain Racer 2000', price: 1499.99}
```

Функции

Функции — это строительные блоки, состоящие из дискретных фрагментов кода. Эти блоки можно использовать многократно, именно из них в итоге строится программа. Но в случае с JavaScript это лишь одна часть истории.

Функции JavaScript являются также полноценными *объектами* — экземплярами типа `Function`. Функции можно присваивать переменным и передавать в качестве аргументов. Их можно объявлять как выражения, без имени, при желании используя упрощенный *стрелочный синтаксис*. Можно даже обернуть одну функцию в другую, создавая закрытый пакет, внутри которого хранится состояние функции, — так называемое *замыкание*.

Кроме того, функции являются основой, обеспечивающей поддержку объектно-ориентированного программирования в JavaScript. Так происходит потому, что создаваемые пользователем классы — это в действительности лишь особый тип функции-конструктора, как вы увидите в главе 8. Рано или поздно все в JavaScript сводится к функциям.

6.1. Передача одной функции в другую в качестве аргумента

Задача

При вызове функции необходимо передать другую функцию. Как лучше это сделать?

Решение

Многие функции JavaScript допускают или даже требуют передачи в них другой функции в качестве аргумента. Одни операции запрашивают функцию обратного вызова, которая будет выполнена после их завершения, другие требуют функцию для выполнения более широкого круга задач. Например, многим методам объекта

Array необходимо предоставить функцию для сортировки, преобразования, объединения или выбора данных. Затем эти функции применяются к массиву многократно — до тех пор пока не будут обработаны все элементы.

Вот некоторые подходы, которые можно использовать при передаче функции в качестве аргумента. Существуют следующие три основных шаблона разработки.

- Передать ссылку на функцию, которая уже объявлена где-то в другом месте кода. Так стоит поступать, когда вы намерены применять функцию и в других частях приложения либо она слишком длинная или сложная.
- Объявить функцию как *функцию-выражение* и передать ее как аргумент. Такой способ хорошо подходит для простых задач, при условии что вы не собираетесь использовать ее где-то еще.
- Объявить встроенную функцию в том месте, где она используется, при передаче в качестве аргумента в другую функцию. Это похоже на предыдущий способ, но позволяет сделать код еще более компактным. Такой способ лучше всего подходит для коротких и простых, особенно однострочных, функций.

Рассмотрим для начала простую страницу со следующей кнопкой:

```
<button id="runTest">Выполнить тест</button>
```

Привяжем к этой кнопке следующий обработчик события:

```
// Привязываем к кнопке обработчик события
document.getElementById('runTest').addEventListener("click", buttonClicked);
```

Теперь рассмотрим функцию `setTimeout()`, которая выполняет другую функцию (она передается в `setTimeout()` как аргумент) после некоторой задержки. Вот как выглядит первый способ передачи функции в `setTimeout()` — это отдельно объявленная функция с именем `showMessage()`:

```
// Эта функция выполняется, если нажать кнопку
function buttonClicked() {
    // Выполнить функцию через 2000 мс (2 с)
    setTimeout(showMessage, 2000);
}

// Эта функция выполняется, когда ее запускает setTimeout()
function showMessage() {
    alert('Вы нажали на эту кнопку 2 с назад');
}
```



При передаче функции в качестве аргумента не надо ставить после ее имени пару круглых скобок. В данном примере в функцию `setTimeout()` передается функция `showMessage`. Если случайно написать вместо этого `showMessage()`, то JavaScript, вместо того чтобы передать в `setTimeout()` ссылку на функцию `showMessage`, немедленно выполнит ее и вернет результат выполнения в `setTimeout()`.

А так выглядит второй способ, где объявление функции представляет собой функцию-выражение и располагается ближе к тому месту, где она нужна:

```
function buttonClicked() {  
    // Объявляем функцию-выражение для использования в setTimeout()  
    const timeoutCallback = function showMessage() {  
        alert('Вы нажали эту кнопку 2 с назад');  
    }  
  
    // Активируем функцию через 2000 мс (2 с)  
    setTimeout(timeoutCallback, 2000);  
}
```

В этом случае область видимости `showMessage()` ограничена функцией `buttonClicked()`. Функцию `showMessage()` нельзя вызвать из другой функции, в другом месте кода. При желании здесь можно обойтись без имени функции (`showMessage`), сделав ее *анонимной*. Это никак не повлияет на работу `timeoutCallback`, но имя функции может пригодиться при отладке, так как в случае ошибок оно будет появляться при трассировке стека.

А так выглядит третий способ, когда функция объявляется как встроенная при вызове `setTimeout()`:

```
function buttonClicked() {  
    // Активировать функцию через 2000 мс (2 с)  
    setTimeout(function showMessage() {  
        alert('Вы нажали эту кнопку 2 с назад');  
    }, 2000);  
}
```

Теперь функция `showMessage()` объявляется и передается в `setTimeout()` в одном и том же выражении. К функции `showMessage()` невозможно обратиться из другой части кода, даже внутри `buttonClicked()`. При желании можно убрать имя `showMessage()` и сделать функцию анонимной:

```
setTimeout(function() {  
    alert('Вы нажали на эту кнопку 2 с назад');  
}, 2000);
```

Эту запись можно еще больше упростить, применив стрелочный синтаксис, как показано в рецепте 6.2. Но использование имени функции является хорошим стилем программирования в случае длинного или сложного кода. Дело в том, что если в функции возникнут ошибки, вы увидите ее имя при трассировке стека.



При использовании анонимных функций обратите внимание на то, какие соглашения о стилях программирования приняты в вашей организации. Один из общепринятых шаблонов программирования состоит в том, чтобы помещать объявление `function()` и открывающую фигурную скобку `{` в одной строке. Ниже с отступом располагается весь код анонимной функции. Наконец, в следующей за кодом строке ставится закрывающая фигурная скобка `}`, после которой сразу идут остальные аргументы для вызова функции.

Обсуждение

Рассмотренные три варианта передачи функции как аргумента показывают, как постепенно сужается ее область видимости от наиболее доступной (в первом примере) до наименее доступной (в последнем). Как правило, чем меньше область видимости функции, тем лучше. Благодаря этому уменьшается неопределенность кода (он становится более понятным для других разработчиков, которые будут с ним работать после вас) и снижается вероятность появления неожиданных побочных эффектов. Но за все приходится платить. Если позже функция станет более длинной и сложной, ее встроенное объявление будет труднее читать. А если функцию предполагается задействовать еще где-то или вы собираетесь писать для нее тесты, то ее необходимо определить как отдельную.

Если вы все еще не очень хорошо понимаете, как одна функция может использовать ссылку на другую функцию, вот простой пример: есть функция `callYouBack()`, которая принимает другую функцию в виде аргумента и затем вызывает ее. Внутри `callYouBack()` ссылка на функцию трактуется точно так же, как обычная функция, которую можно вызвать по имени и которой, если нужно, передать параметры:

```
function buttonClicked() {
    // Создаем функцию, которая будет выполнять обратный вызов
    function logTime(time) {
        console.log('Время записи: ' + time.toLocaleTimeString());
    }

    console.log('Сейчас выполнится callYouBack()');
    callYouBack(logTime);
    console.log('Готово');
}

function callYouBack(callbackFunction) {
    console.log('Начало работы callYouBack()');

    // Вызываем переданную функцию с аргументом
    // callbackFunction(new Date());

    console.log('Завершение callYouBack()');
}
```

Если выполнить этот код и нажать кнопку, то получим следующее:

```
Сейчас выполнится callYouBack()
Начало работы callYouBack()
Время записи: 2:20:59 PM
Завершение callYouBack()
Готово
```

Читайте также

В рецепте 6.2 описан синтаксис, позволяющий упростить объявление анонимных функций. Он особенно полезен для однострочных функций, возвращающих

переменную. В табл. 5.1 перечислены основные методы объекта `Array`, принимающие функцию в качестве параметра.

6.2. Использование стрелочных функций

Задача

С помощью синтаксиса стрелочных функций JavaScript объявить встроенную функцию максимально компактным способом.

Решение

За последние годы в JavaScript существенно сместились акценты в сторону функциональных принципов разработки. Яркими примерами служат работа с массивами и возможности асинхронного программирования. Для этого в языке появился упрощенный синтаксис написания встроенных функций, которые называют *стрелочными*.

Вот пример использования метода `Array.map()` для преобразования содержимого массива с помощью именованной функции. Исходный массив представляет собой список чисел, преобразованный массив — список квадратов этих чисел:

```
const numbers = [1,2,3,4,5,6,7,8,9,10];

function squareNumber(number) {
    return number**2;
}
const squares = numbers.map(squareNumber);

console.log(squares);
// Выводится [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

А вот тот же пример, но теперь функция `squareNumber()` определена как встроенная, с использованием стрелочного синтаксиса:

```
const numbers = [1,2,3,4,5,6,7,8,9,10];
const squares = numbers.map( number => number**2 );

console.log(squares);
```

Обсуждение

В этом примере использована наиболее компактная форма стрелочного синтаксиса. Она подходит для функций, принимающих единственный параметр и состоящих из единственного выражения. Для остальных функций упрощения, предоставляемые стрелочным синтаксисом, менее доступны. Чтобы понять, почему так происходит, рассмотрим пошагово, как именованная функция преобразуется в функцию-выражение со стрелочным синтаксисом.

1. Сначала ставим список параметров, после него — символ `=>`. Если параметров нет, то перед символом `=>` ставим пустые скобки:

```
(number) =>
```

2. Если параметр только один, как в данном примере, то скобки, в которые заключен список параметров, можно убрать:

```
number =>
```

3. Справа от стрелки ставим фигурные скобки, в которые заключено тело функции:

```
number => {  
    return number**2;  
}
```

4. Если функция состоит из единственного выражения, то фигурные скобки и ключевое слово `return` можно убрать. Но если выражений несколько, то и скобки, и `return` нужно оставить:

```
number => number**2;
```

Учтите, что стрелочные функции предназначены для объявления встроенных функций, так что вы всегда будете передавать их результат куда-то дальше или присваивать его переменной в выражении:

```
const myFunc = number => number**2;
```

```
const squaredNumber = myFunc(10);  
// squaredNumber = 100
```

Теперь рассмотрим преобразование чуть более сложной функции:

```
function raiseToPower(number, power) {  
    return number**power;  
}
```

Мы можем выполнить п. 1, 3 и 4, но п. 2 здесь не применим, поскольку у функции два аргумента:

```
const myFunc = (number, power) => number**power;
```

Рассмотрим еще один пример — функцию более детальной обработки строк:

```
function applyTitleCase(inputString) {  
    // Разбиваем строку на массив слов  
    const wordArray = inputString.split(' ');  
  
    // Создаем новый массив для обработанных слов  
    const processedWordArray = [];  
  
    for (const word of wordArray) {  
        // Делаем первую букву каждого слова прописной
```

```
        processedWordArray.push(word[0].toUpperCase() + word.slice(1));
    }

    // Объединяем все слова обратно в одну строку
    return processedWordArray.join(' ');
}
```

Здесь можно применить п. 1, 2 и 3, но не п. 4. Необходимо оставить фигурные скобки и ключевое слово `return`:

```
const myFunc = inputString => {
    // Разбиваем строку на массив слов
    const wordArray = inputString.split(' ');

    // Создаем новый массив для обработанных слов
    const processedWordArray = [];

    for (const word of wordArray) {
        // Делаем первую букву каждого слова прописной
        processedWordArray.push(word[0].toUpperCase() + word.slice(1));
    }

    // Объединяем все слова обратно в одну строку
    return processedWordArray.join(' ');
}
```

Здесь различий между традиционной и стрелочной записью гораздо меньше. Изменилось только объявление функции в самом начале, а общая экономия кода минимальна.



Именно в таких случаях трудно сделать выбор между стрелочным и обычным синтаксисом. Часто удается сжать несколько выражений, из которых состоит функция, в одно. В примере с обработкой строки можно объединить несколько методов в цепочку (как показано в рецепте 5.8) и заменить цикл на функцию `Array.map()` (рецепт 5.17). Если как следует постараться, то эти изменения позволяют сжать `applyTitleCase()` в одно длинное выражение, после чего можно использовать все сокращения стрелочного синтаксиса. Но в этом случае платой за более лаконичный код будет то, что его станет трудно читать. Как правило, стрелочный синтаксис полезен только в том случае, если он помогает писать более понятный код.

В стрелочных функциях ключевое слово `this` имеет другой смысл. В объявленной функции оно указывает на объект, вызвавший функцию, — это может быть текущее окно, кнопка и т. п. В стрелочной функции `this` просто ссылается на код, в котором эта функция объявлена. (Другими словами, на что бы ни ссылался `this` при создании стрелочной функции, при ее выполнении это будет одна и та же ссылка.) Такое поведение многое упрощает, но за него приходится платить: стрелочный синтаксис не применим к методам и конструкторам объектов, так как стрелочные функции не привязываются к объекту, для которого вызываются. Это поведение не меняется даже при использовании `Function.bind()`.

Есть еще несколько мелких ограничений. Стрелочные функции нельзя применять с ключевым словом `yield` для создания функций-генераторов. Они также не поддерживают объект `arguments`.

Читайте также

В главе 5 приводятся многочисленные примеры использования стрелочного синтаксиса для передачи коротких функций в методы обработки массивов — см., например, рецепты 5.9, 5.14 и 5.16.

6.3. Предоставление значения параметра по умолчанию

Задача

Определить для параметра значение по умолчанию, которое будет использоваться, если при вызове функции не будет передан соответствующий аргумент.

Решение

Значения параметров по умолчанию можно определить сразу же при объявлении функции. Вот пример, где предусмотрено значение по умолчанию для третьего параметра — `thirdNum`:

```
function addNumbers(firstNum, secondNum, thirdNum=0) {  
    return firstNum+secondNum+thirdNum;  
}
```

Теперь эту функцию можно вызвать, не передавая ей все три параметра:

```
console.log(addNumbers(42, 6, 10)); // выведет 58  
console.log(addNumbers(42, 6));     // выведет 48
```

Обсуждение

Параметры по умолчанию — относительно свежее изобретение. Но в JavaScript при вызове функций никогда не было жесткого требования передавать все параметры. В отдаленном прошлом функции просто проверяли, равен ли параметр `undefined` (с помощью оператора `typeof`, как показано в рецепте 7.1).

Значения по умолчанию можно присвоить произвольному числу параметров. Хорошим стилем программирования считается сначала ставить обязательные параметры, а затем — те, для которых предусмотрены значения по умолчанию. Другими словами, если вы назначаете параметру значение по умолчанию, то все

остальные параметры после него также становятся необязательными и должны иметь свои значения по умолчанию. Это не обязательное соглашение, но оно позволяет сделать код более понятным.

При вызове функции, для которой определены несколько параметров по умолчанию, можно выбирать, для каких из них вы передаете значения, а для каких — нет. Рассмотрим следующий пример.

```
function addNumbers(firstNum=10, secondNum=20, thirdNum=30, multiplier=1) {
    return multiplier*(firstNum+secondNum+thirdNum);
}
```

Если мы хотим задать значения для параметров `firstNum`, `secondNum` и `multiplier`, но не для `thirdNum`, необходимо использовать `undefined` в качестве заглушки. Это позволяет передать все параметры в правильном порядке:

```
const sum = addNumbers(42, 10, undefined, 1);
// sum = 82
```

Но `null` не работает как заглушка. В этом примере `null` просто будет преобразован в число 0, что повлияет на результат:

```
const sum = addNumbers(42, 10, null, 1);
// sum = 52
```

Во многих других языках есть более удобные способы обозначить параметры по умолчанию, такие как использование запятых, чтобы показать последовательность параметров без указания значения-заглушки, или присвоение значений параметрам с соответствующими именами. В JavaScript это не работает, хотя можно имитировать именованные параметры посредством литерала объекта (рецепт 6.5).

6.4. Создание функции, принимающей неограниченное число аргументов

Задача

Создать функцию, которая принимает столько аргументов, сколько надо вызывающему ее объекту, без создания массива.

Решение

При объявлении функции использовать *rest-параметр*. Для того чтобы определить *rest-параметр*, нужно поставить перед его именем три точки:

```
function sumRounds(...numbers) {
    let sum = 0;
```

```
for(let i = 0; i < numbers.length; i+=1) {  
    sum += Math.round(numbers[i]);  
}  
return sum;  
}  
  
console.log(sumRounds(2.3, 4, 5, 16, 18.1)); // 45
```

Обсуждение

Rest-параметр не обязан быть единственным, но должен быть последним параметром функции. Он объединяет в себе все остальные аргументы, которые передаются в функцию, и создает из них массив.

В прежние времена разработчики JavaScript для реализации подобного функционала использовали объект `arguments`. Он доступен для любой функции (технически это свойство `Function.prototype.arguments`). Объект `arguments` предоставляет доступ ко всем своим параметрам по принципу массива. Однако `arguments` — не настоящий массив, и разработчики часто задеиствуют шаблонный код, чтобы преобразовать его в массив. Такой подход встречается в коде до сих пор, но современные rest-параметры позволяют избежать этой проблемы.



Rest-параметр выглядит точно так же, как `spread`-оператор (см. рецепт 5.4), но в действительности rest-параметры и `spread`-операторы дополняют друг друга. `Spread`-оператор разбивает массив или свойства объекта на отдельные значения, а `rest`-оператор объединяет отдельные значения, создавая из них единый объект массива.

Читайте также

Если вы хотите передать в функцию массив значений, а функция принимает `rest`-параметр, то можно выполнить преобразование посредством `spread`-оператора (см. рецепт 5.4).

В этом примере для обработки массива значений задеиствован цикл. Но можно получить тот же результат, написав более понятный код с помощью функции `Array.reduce()`, как показано в рецепте 5.18.

6.5. Использование именованных параметров функции

Задача

Найти более простой способ выбирать, какие именно необязательные параметры будут переданы в функцию.

Решение

Объединить все необязательные параметры в один литерал объекта (рецепт 7.2). Тогда при вызове можно будет выбирать, какие из них указывать при создании литерала. Вот пример вызова функции с помощью этого шаблона проектирования:

```
someFunction(arg1, arg2, {optionalArg1: val1, optionalArg2: val2});
```

Чтобы внутри функции быстро скопировать значения из литерала объекта в отдельные переменные, можно использовать *деструктурирующее присваивание*. Вот пример функции, принимающей три аргумента. Из них первые два (`newerDate` и `olderDate`) — обязательные, а третий представляет собой литерал объекта, в котором могут храниться три необязательных значения (`discardTime`, `discardYears` и `precision`):

```
function dateDifferenceInSeconds(
  newerDate, olderDate, {discardTime, discardYears, precision} = {}) {

  if (discardTime) {
    newerDate = newerDate.setHours(0,0,0,0);
    olderDate = olderDate.setHours(0,0,0,0);
  }
  if (discardYears) {
    newerDate.setYear(0);
    olderDate.setYear(0);
  }

  const differenceInSeconds = (newerDate.getTime() -
    olderDate.getTime())/1000;
  return differenceInSeconds.toFixed(precision);
}
```

Функцию `dateDifferenceInSeconds()` можно вызывать как с литералом объекта, так и без него:

```
// Сравниваем текущую дату с более ранней
const newerDate = new Date();
const oldDate = new Date(2010, 1, 10);

// Вызываем функцию без литерала объекта
let difference = dateDifferenceInSeconds(newerDate, oldDate);
console.log(difference); // Увидим что-то вроде 354378086

// Вызываем функцию с литералом объекта и задаем два свойства
difference = dateDifferenceInSeconds(
  newerDate, oldDate, {discardYears:true, precision:2});
console.log(difference); // Увидим что-то вроде 7226485.90
```

Обсуждение

Использование литерала объекта для передачи необязательных значений — это типичный шаблон JavaScript. Он позволяет задавать только необходимые свойства, не заботясь об их последовательности:

```
// Это работает
dateDifferenceInSeconds(newDate, oldDate, {precision:2});

// И это работает
dateDifferenceInSeconds(newDate, oldDate, {discardYears:true, precision:2});

// И так тоже работает
dateDifferenceInSeconds(newDate, oldDate, {precision:2, discardYears:true});
```

Внутри функции можно извлекать свойства из объекта индивидуально — например, так:

```
function dateDifferenceInSeconds(newerDate, olderDate, options) {
    const precision = options.precision;
```

Но у этого решения в данном рецепте есть более сокращенный вариант — распаковать литерал объекта, разместив свойства объекта в отдельных именованных переменных, посредством деструктурирования. Можно использовать деструктурирующее присваивание в отдельном выражении:

```
function dateDifferenceInSeconds(newerDate, olderDate, options) {
    const {discardTime, discardYears, precision} = options;
```

Или сразу в объявлении функции:

```
function dateDifferenceInSeconds(
    newerDate, olderDate, {discardTime, discardYears, precision})
```

Считается хорошим тоном применять пустой литерал объекта в качестве значения по умолчанию (рецепт 6.3). Такой объект будет использован, если литерал объекта не определен при вызове функции:

```
function dateDifferenceInSeconds(
    newerDate, olderDate, {discardTime, discardYears, precision} = {})
```

Какие именно свойства литерала объекта будут указаны — все, лишь некоторые или вообще никаких, — определяется в момент вызова функции. Всем параметрам, которые не были определены, присваивается специальное значение `undefined` — и соответствующую проверку стоит включить в код. Вот менее оптимизированный пример:

```
if (discardTime !== undefined || discardTime === true) {
```

Но часто в явной проверке на `undefined` нет необходимости. Например, при проверке условий `undefined` расценивается как `false`. В функции `dateDifferenceInSeconds()` это поведение использовано при определении значений `discardYears` и `discardTime`, что позволяет сократить код:

```
if (discardTime) {
```

Аналогичное сокращение применяется и для свойства `precision`. Можно спокойно вызывать `Number.toPrecision(undefined)`, поскольку это то же самое, что

вызвать `toPrecision()` без аргументов. В любом случае числа будут округляться до ближайшего целого.

Единственный недостаток шаблона проектирования с применением литералов объектов — то, что при таком подходе невозможно предотвратить ошибки в именах свойств, например:

```
// Нам нужна переменная discardYears, но мы случайно
// задали ее как discardYear
dateDifferenceInSeconds(newDate, oldDate, {discardYear:true});
```

Читайте также

Литералы объектов описаны в рецепте 7.2, а в рецепте 5.4 показано, как использовать синтаксис деструктурирования массива. Этот синтаксис подобен деструктурированию объекта, описанному в данном рецепте, за тем исключением, что он оперирует не объектами, а массивами (и в нем применяются квадратные скобки вместо фигурных).

6.6. Создание функции с сохранением состояния посредством замыкания

Задача

Создать функцию, способную запоминать данные, не задействуя глобальные переменные и не пересылая постоянно одни и те же данные при каждом вызове функции.

Решение

Обернуть функцию, состояние которой мы хотим сохранить, в *другую* функцию. Внешняя функция будет возвращать внутреннюю в соответствии со следующей структурой:

```
function outerFunction() {
  function innerFunction() {
    ...
  }
  return innerFunction;
}
```

Обе эти функции принимают параметры. Но есть одна тонкость: параметры внешней функции доступны до тех пор, пока доступна ссылка на внутреннюю. Можно вызывать внутреннюю функцию сколько угодно раз, и данные из внешней будут сохраняться. (На уровне концепции это выглядит так, как будто внешняя функция — это метод создания объекта, а внутренняя — объект с состоянием.)

Полностью пример выглядит так:

```
function greetingMaker(greeting) {
  function addName(name) {
    return `${greeting} ${name}`;
  }
  return addName;
}

// С помощью внешней функции создаем две копии
// внутренней функции, каждая — со своим текстом приветствия
const daytimeGreeting = greetingMaker('Good Day to you');
const nightGreeting = greetingMaker('Good Evening');

console.log(daytimeGreeting('Peter')); // Выводит 'Good Day to you Peter'
console.log(nightGreeting('Sally'));   // Выводит 'Good Evening Sally'
```

Обсуждение

Часто возникает необходимость сохранить данные, которые используются при нескольких вызовах функции. Для этого можно задействовать глобальные переменные, но это средство — на крайний случай. Глобальные переменные приводят к конфликтам имен и усложнению кода: из-за них возникают скрытые взаимозависимости между функциями, ограничивается возможность повторного применения кода. Под прикрытием глобальных переменных часто скрываются мелкие ошибки кодирования.

Можно было бы сделать так, чтобы информация сохранялась в объекте, вызывающем функцию, и передавалась при каждом вызове, но это тоже не особенно красиво. В данном примере показано другое решение — создание пакета функций с сохранением состояния, именуемого *замыканием*. В нем внешняя функция `greetingMaker()` принимает один аргумент, который определяет текст приветствия. Функция `greetingMaker()` возвращает внутреннюю функцию, `addName()`, которая, в свою очередь, тоже принимает аргумент — имя человека. Замыкание включает в себя функцию `addName()` и окружающий ее контекст, в том числе параметр, переданный в функцию `greetingMaker()`. Чтобы продемонстрировать это, мы создали две копии `addName()` в двух разных контекстах: одна копия находилась в замыкании с приветствием «Добрый день», переданным в `greetingMaker()`, а вторая — в замыкании с функцией `greetingMaker()`, в которую было передано приветствие «Добрый вечер». В обоих случаях при вызове `addName()` для построения сообщения использовался текущий контекст.

Следует отметить, что состояние не ограничивается значениями параметров. Пока существует ссылка на внутреннюю функцию, существуют и все переменные, принадлежащие внешней функции. Вот пример, в котором с помощью простой переменной-счетчика ведется подсчет вызовов функции:

```
function createCounter() {
  // Эта переменная существует до тех пор, пока
```

```
// существует ссылка на функцию createCounter
let count = 0;

function counter() {
    count += 1;
    console.log(count);
}
return counter;
}

const counterFunction = createCounter();
counterFunction(); // выводится 1
counterFunction(); // выводится 2
counterFunction(); // выводится 3
```

Читайте также

Еще один пример функции, в которой используется замыкание для сохранения состояния, вы найдете в подразделе «Дополнительно: построение многократно вызываемого генератора псевдослучайных чисел» далее в этой главе.

Замыкания и обернутые функции кажутся отголосками объектно-ориентированного программирования — и не случайно. В прежние времена разработчики JavaScript использовали функции, чтобы имитировать создание классов (см. рецепт 8.4), и ключевое слово `class` в JavaScript — продолжение этой методики (см. рецепт 8.1).

6.7. Создание функции-генератора, которая возвращает несколько значений

Задача

Создать *генератор* — функцию, способную предоставлять несколько значений по требованию. Каждый раз, вернув значение, генератор приостанавливает выполнение, до тех пор пока вызывающий объект не запросит следующее значение.

Решение

Для того чтобы объявить функцию-генератор, необходимо в начале объявления заменить ключевое слово `function` на `function*`:

```
function* generateValues() {
}
```

Каждый раз, когда нужно вернуть результат, внутри функции-генератора используется ключевое слово `yield`. Помните, что после каждого `yield` (которое

почти аналогично `return`) выполнение функции приостанавливается. Но после того как объект, вызвавший функцию, запросит следующее значение, выполнение функции *возобновляется*. Этот процесс продолжается до тех пор, пока не закончится код функции или пока она не вернет окончательное значение с помощью ключевого слова `return`.

Далее показана наивная реализация генератора. (Она работает, но не делает ничего полезного.) Эта функция возвращает три значения с помощью ключевого слова `yield`, а в конце возвращает значение с помощью `return`:

```
function* generateValues() {
  yield 895498;
  yield 'Это второе значение';
  yield 5;
  return 'Это конец';
}
```

При вызове функция-генератор возвращает объект `Generator`. Это происходит сразу же, прежде чем начнет выполняться код функции-генератора. Объект `Generator` используется для запуска функции и получения значений, которые она возвращает. Также с помощью объекта `Generator` можно определить условия завершения функции.

При каждом вызове `Generator.next()` функция-генератор будет выполняться до следующего ключевого слова `yield` или завершающего `return`. Метод `next()` возвращает объект, состоящий из двух значений. В свойстве `value` хранится значение, возвращаемое функцией посредством `yield` или `return`. Свойство `done` — это значение типа `Boolean`, которое равно `false` до тех пор, пока функция-генератор не завершит работу:

```
const generator = generateValues();

// Запускаем генератор (он будет выполняться от начала до первого yield)
console.log(generator.next().value); // 895498

// Снова запускаем генератор (до следующего yield)
console.log(generator.next().value); // 'Это второе значение'

// Получаем оставшиеся два значения
console.log(generator.next().value); // 5
console.log(generator.next().value); // 'Это конец'
```

Обсуждение

Генераторы позволяют создавать функции, выполнение которых можно приостановить и потом продолжить. Самое главное здесь то, что JavaScript управляет состоянием этих функций автоматически — другими словами, вы не обязаны писать специальный код для хранения значений между вызовами `next()` (в отличие, например, от самостоятельной разработки итератора).

Поскольку в генераторах используется модель ленивых вычислений, то они хорошо подходят для выполнения затратных по времени операций по созданию или получению данных. Например, генератор можно применять для вычисления сложной последовательности чисел или извлечения фрагментов информации из потока данных.

Как правило, нас не интересует, сколько именно значений возвращает генератор. Можно написать цикл `while`, который будет проверять свойство `Generator.done` и вызывать `next()`, до тех пор пока функция не будет завершена. Однако, поскольку объект `Generator` является итерируемым, для него лучше подходит цикл `for ... of`:

```
// Получаем все значения генератора
for (const value of generateValues()) {
  console.log(value);
}

// Благодаря синтаксису spread можно одной строкой
// объединить все значения в массив
const values = [...generateValues()];
```

В любом случае такой подход позволяет получить только значения, возвращаемые посредством `yield`. Если генератор также возвращает что-то в завершающем `return`, то это значение игнорируется.

Некоторые функции-генераторы бесконечны по своей природе. Вы будете получать значения, возвращаемые посредством `yield`, столько раз, сколько вызовете `next()`. При вызове бесконечного генератора невозможно собрать все значения в массив (программа зависнет). Вместо этого стоит использовать цикл `while` с условием, которое становится равным `false`, после того как будут получены все необходимые значения.

Читайте также

В рецепте 9.6 показано, как создавать генераторы, которые выполняются асинхронно.

Дополнительно: построение многократно вызываемого генератора псевдослучайных чисел

Вы уже изучили основной синтаксис функций-генераторов, но еще не видели настоящих практических примеров. Вот один такой пример, в котором видно, как бесконечная функция-генератор может выдавать полезную последовательность значений.

Как было показано в рецепте 3.1, метод `Math.random()` генерирует псевдослучайные числа, но не позволяет контролировать начальное значение. (В качестве начального значения в генераторе псевдослучайных чисел `Math.random()`

используется непрозрачный, некриптографически безопасный метод, зависящий от реализации JavaScript.) Это вполне подходит для большинства приложений, но в некоторых сценариях необходимо генерировать повторяющуюся последовательность псевдослучайных чисел. Распределение этих чисел все равно должно быть статистически случайным, единственное различие состоит в том, что необходима возможность получить от генератора псевдослучайных чисел одну и ту же последовательность несколько раз. Такие повторяющиеся псевдослучайные числовые последовательности нужны, например, в некоторых симуляциях и тестах, которые должны быть полностью воспроизводимыми.

Существует несколько сторонних библиотек JavaScript, в которых реализованы генераторы псевдослучайных чисел с возможностью задания начального значения (и, следовательно, генерирующие повторяемые последовательности). Длинный список таких генераторов есть на GitHub (<https://github.com/bryc/code/blob/master/jshash/PRNGs.md>). Один из простейших генераторов называется Mulberry32. Его реализация на JavaScript состоит всего из нескольких плотных строк кода:

```
function mulberry32(seed) {
  return function random() {
    let t = seed += 0x6D2B79F5;
    t = Math.imul(t ^ t >> 15, t | 1);
    t ^= t + Math.imul(t ^ t >> 7, t | 61);
    return ((t ^ t >> 14) >> 0) / 4294967296;
  }
}

// Выбираем начальное значение
const seed = 98345;

// Получаем версию mulberry32(), в которой используется
// это начальное значение:
const randomFunction = mulberry32(seed);

// Генерируем несколько случайных чисел
console.log(randomFunction()); // 0.9057375795673579
console.log(randomFunction()); // 0.44091642647981644
console.log(randomFunction()); // 0.7662326360587031
```



Подобно большинству генераторов псевдослучайных чисел, Mulberry32 возвращает дробные числа в диапазоне от 0 до 1. Для того чтобы преобразовать их в целые числа заданного диапазона, можно воспользоваться методом, описанным в рецепте 3.1.

В функции `mulberry32()` использована технология замыкания, описанная в рецепте 6.6. Эта функция принимает начальное значение, которое затем блокируется в контексте внутренней функции `random()`. Это означает, что когда бы мы ни вызвали функцию `random()`, исходное начальное значение будет доступно во

внешней функции. Это важно, поскольку другое начальное значение даст другую последовательность случайных чисел. Если вызывать `mulberry32()` с одним и тем же начальным значением, то мы гарантированно получим от `random()` одну и ту же последовательность псевдослучайных чисел.

Замыкания стали частью JavaScript в незапамятные времена, но генераторы — гораздо более новое приобретение. Этот пример можно переписать с применением функции-генератора, благодаря чему назначение кода станет гораздо понятнее:

```
function* mulberry32(seed) {
  let t = seed += 0x6D2B79F5;

  // Генерируем бесконечную последовательность чисел
  while(true) {
    t = Math.imul(t ^ t >>> 15, t | 1);
    t ^= t + Math.imul(t ^ t >>> 7, t | 61);
    yield ((t ^ t >>> 14) >>> 0) / 4294967296;
  }
}

// Используем одно и то же начальное значение, чтобы получить
// одну и ту же последовательность
const seed = 98345;

const generator = mulberry32(seed);
console.log(generator.next().value); // 0.9057375795673579
console.log(generator.next().value); // 0.7620641703251749
console.log(generator.next().value); // 0.0211441791616380
```

Поскольку функция `mulberry32()` объявлена посредством `function*`, сразу становится очевидно, что она возвращает несколько значений. Внутри функции находится бесконечный цикл, который гарантирует, что генератор всегда будет готов выдать новое число. При каждом прохождении цикла функция `random()` возвращает новое случайное число и приостанавливает работу, до тех пор пока `next()` не запросит следующее значение. В целом, это решение выполняется примерно так же, как первоначальная версия, но только данный вариант соответствует известному шаблону проектирования, благодаря чему проще понять, как это работает. (Однако, как всегда, полезность подобного рефакторинга зависит от соглашений, принятых в вашей организации, ожиданий тех, кто будет читать код, и ваших собственных предпочтений.)



В том, чтобы построить генератор с бесконечным циклом, возвращающим значения, нет ничего опасного. Выдав очередное значение, генератор приостанавливает работу, так что он не «подвесит» цикл событий JavaScript. В отличие от обычных функций, от функции-генератора не ожидается, что она будет выполнена до последней закрывающей фигурной скобки. Когда объект `Generator` выйдет за пределы своей области видимости, функция и ее контекст станут доступными для сборщика мусора.

6.8. Уменьшение избыточности за счет частичного применения

Задача

Есть функция, принимающая несколько аргументов. Мы хотим обернуть ее в одну или несколько специализированных версий, которые требуют меньшего числа аргументов.

Решение

Следующая функция `makeString()` принимает три параметра (другими словами, ее *арность* равна 3):

```
function makeString(prefix, str, suffix) {  
    return prefix + str + suffix;  
}
```

Но первый и последний аргументы часто остаются неизменными для конкретного сценария использования. Хотелось бы по возможности избежать повторения одних и тех же аргументов.

Чтобы решить эту задачу, можно создать для уже существующей функции `makeString()` несколько функций-оберток с зафиксированными внутри них известными значениями аргументов:

```
function quoteString(str) {  
    return makeString('"', str, '"');  
}  
  
function barString(str) {  
    return makeString('-', str, '-');  
}  
  
function namedEntity(str) {  
    return makeString('&#', str, ';');  
}
```

Теперь для вызова одной из этих функций достаточно указать только один аргумент:

```
console.log(quoteString('apple'));    // "apple"  
console.log(barString('apple'));      // -apple-  
console.log(namedEntity(169));        // "&#169; (символ авторского права в HTML)
```

Обсуждение

Технология обертывания одной функции вокруг другой, чтобы зафиксировать во внешней функции один или несколько аргументов, называется *частичным*

применением (поскольку новые функции *применяют часть аргументов* к исходной функции). Разумеется, за это приходится платить: дополнительные функции, которые вы создаете, засоряют код, так что не стоит создавать обертки, которые не собираетесь использовать многократно.

Дополнительно: фабрика частичных функций

Для того чтобы еще сильнее уменьшить избыточность кода посредством этой методики, можно создать функцию, которая будет парциализировать (переводить в режим частичного применения) *любую* другую функцию. В сущности, такой подход представляет собой широко используемый шаблон проектирования JavaScript. В прежние времена при манипулировании объектами и массивами JavaScript приходилось полагаться на список аргументов. В современном JavaScript эта задача значительно упростилась благодаря `rest`- и `spread`-операторам.

В следующей реализации показана функция парциализации `partial()`. Она позволяет сократить любое количество аргументов для любой функции:

```
function partial(fn, ...argsToApply) {  
  return function(...restArgsToApply) {  
    return fn(...argsToApply, ...restArgsToApply);  
  }  
}
```

Эта функция требует разъяснения. Но прежде давайте рассмотрим простой пример ее использования. В данном случае функция `partial()` применяется для создания новой функции `cubeIt()`, которая является оберткой для более общей функции `raiseToPower()`. Другими словами, в `cubeIt()` с помощью частичного применения фиксируется один из аргументов `raiseToPower()` (показатель степени, который устанавливается равным 3):

```
// Функция, которую мы хотим парциализировать  
function raiseToPower(exponent, number) {  
  return number**exponent;  
}  
  
// Используя partial(), создаем специализированную функцию  
const cubeIt = partial(raiseToPower, 3);  
  
// Вычисляем 9 в кубе (9**3)  
console.log(cubeIt(9)); // 729
```

Теперь при вызове `cubeIt(9)` вызов будет преобразовываться в `raiseToPower(3, 9)`.

Как же это работает? Функция принимает два аргумента. Первый из них — функция, которую нужно парциализировать (`fn`), второй — список всех аргументов, которые мы хотим зафиксировать (`argsToApply`), преобразуемый в массив посредством `rest`-оператора (`...`), как показано в рецепте 6.4:

```
function partial(fn, ...argsToApply) {
```

Дальше становится интереснее. Функция `partial` возвращает вложенную внутреннюю функцию (эта методика описана в рецепте 6.6). Та получает все аргументы, которые не были зафиксированы. Эти аргументы также объединены в массив посредством `rest`-оператора (`...restToApply`):

```
// Создаем и возвращаем анонимную функцию
return function(...restArgsToApply) {
```

Новая функция получает три вида информации: исходную функцию (`fn`), зафиксированные аргументы (`argsToApply`) и аргументы, передаваемые при каждом вызове функции (`restArgsToApply`).

Сама функция состоит из единственной, но очень важной строки кода. Здесь два массива преобразуются в списки аргументов посредством `spread`-оператора, который, что несколько сбивает с толку, выглядит точно как `rest`-оператор. Другими словами, `argsToApply` превращается в список аргументов, после которого идут аргументы `restToApply`:

```
// Здесь вызывается обернутая функция
return fn(...argsToApply, ...restArgsToApply);
```



В функциональном программировании часто используются функции высшего порядка (функции, которые управляют другими функциями). Функция `partial()` — это функция высшего порядка, которая создает обертку для другой функции.

У этой реализации функции `partial()` есть еще одно ограничение. Поскольку в нее первыми передаются фиксированные аргументы, мы не можем зафиксировать более поздний аргумент, не зафиксировав предварительно все аргументы, идущие перед ним. Если бы мы захотели использовать `partial()` для создания обертки функции `makeString()` из первоначального примера, то нам понадобилось бы сначала поменять последовательность аргументов:

```
function makeString(prefix, suffix, str) {
  return prefix + str + suffix;
}
```

```
const namedEntity = partial(makeString, "&#", ";");
```

```
console.log(namedEntity(169));
```

Дополнительно: частичная передача аргументов с помощью `bind()`

Для частичного применения аргументов можно также использовать метод `Function.bind()`. Метод `bind()` возвращает новую функцию, при этом `this` будет указывать на то, что является первым аргументом этой функции. Все остальные аргументы добавляются в начало списка новой функции.

Вместо того чтобы создавать именованную функцию посредством `partial()`, теперь мы можем предоставить тот же функционал с помощью `bind()`, передав `undefined` в качестве первого аргумента:

```
function makeString(prefix, suffix, str) {
    return prefix + str + suffix;
}

const named = makeString.bind(undefined, "&#", ";");
console.log(named(169)); // "&#169;"
```

Теперь у нас есть два хороших способа создания нескольких версий функции, в которых используются разные параметры.

6.9. Фиксация `this` посредством привязки функций

Задача

Функция пытается применить ключевое слово `this`, но оно привязано не к тому объекту.

Решение

Используя метод `Function.bind()`, изменить контекст функции и, соответственно, то, на что ссылается `this`:

```
window.onload = function() {
    window.name = 'window';

    const newObject = {
        name: 'object',

        sayGreeting: function() {
            console.log(`Now this is easy, ${this.name}`);

            const nestedGreeting = function(greeting) {
                console.log(`${greeting} ${this.name}`);
            }.bind(this);

            nestedGreeting('hello');
        }
    };

    newObject.sayGreeting();
};
```

Обсуждение

Ключевое слово `this` указывает на владельца, или родителя, функции. В JavaScript из-за этого возникает проблема: мы не всегда точно знаем, какой именно родительский объект будет применяться к функции.

В приведенном коде у объекта есть метод `sayGreeting()`, который выводит сообщение и присваивает еще одну, вложенную, функцию свойству объекта `nestedGreeting`. Такой подход используется в шаблоне проектирования «конструктор» (рецепт 8.4) для создания классоподобных объектов-функций.

Без использования метода `Function.bind()` первое сообщение выглядело бы так: *Now this is easy, object*, а второе — *hello window*. Во втором сообщении выводится другое имя: поскольку функция вложенная, внешний объект больше не ссылается на внутреннюю функцию, а все функции, находящиеся *вне области видимости*, автоматически становятся свойствами объекта `window`.

Метод `bind()` решает эту проблему, связывая функцию с выбранным объектом. В данном примере метод `bind()` вызывается вложенной функцией и ему передается ссылка на родительский объект. Теперь, когда во внутреннем коде функции `nestedGreeting()` используется ключевое слово `this`, оно указывает на назначенный нами ранее родительский объект.

Метод `bind()` особенно полезен для функций-таймеров `setTimeout()` и `setInterval()`. Обычно, когда они запускают переданную им функцию обратного вызова, ссылка `this` теряется (`this` становится равным `undefined`). Но благодаря `bind()` можно гарантировать, что нужная ссылка сохранится в функции обратного вызова.

В примере 6.1 показана веб-страница, на которой с помощью `setTimeout()` выполняется обратный отсчет от 10 до 0. Когда счетчик уменьшается, его значение размещается на веб-странице. Также в этом примере для создания объекта использован шаблон проектирования «конструктор» (как описано в рецепте 8.4), который создает классоподобную функцию `Counter`.

Пример 6.1. Демонстрация полезности метода `bind()`

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <meta http-equiv="X-UA-Compatible" content="ie=edge" />
    <title>Using Bind with Timers</title>
  </head>
  <body>
    <div id="counterDiv"></div>

    <script>
      // Функция-конструктор для объекта Counter
      function Counter(from, to, divElement) {
        this.currentCount = from;
        this.finishCount = to;
        this.element = divElement;

        // Метод incrementCounter() обновляет страницу
```

```
this.incrementCounter = function() {
    this.currentCount -= 1;
    this.element.textContent = this.currentCount;
    if (this.currentCount > this.finishCount) {
        // Запланируем повторный запуск этой функции
        // через 1 секунду
        setTimeout(this.incrementCounter.bind(this), 1000);
    }
};

this.startCounter = function() {
    this.incrementCounter();
}

// Создаем счетчик для этой страницы
const counter = new Counter(10, 0,
    document.getElementById('counterDiv'));

// Запускаем счетчик при загрузке страницы
window.onload = function() {
    counter.startCounter();
}
</script>
</body>
</html>
```

Если бы функция `setTimeout()` в этом коде выглядела так:

```
setTimeout(this.incrementCounter, 1000);
```

то ссылка `this` терялась бы, и у функции обратного вызова не было бы доступа к таким переменным, как `currentCount`, несмотря на то что метод `incrementCounter()` относится к тому же объекту.

Дополнительно: `self = this`

В прежние времена, а иногда и сейчас, вместо того чтобы использовать `bind()`, присваивали `this` переменной во внешней функции, и потом эта переменная была доступна во внутренней функции. Обычно `this` присваивают переменной с именем `that` или `self`:

```
window.onload = function() {
    window.name = 'window';

    const newObject = {
        name: 'object',

        sayGreeting: function() {
            const self = this;
            alert('Now this is easy, ' + this.name);
            nestedGreeting = function(greeting) {
                alert(greeting + ' ' + self.name);
            };
        }
    };
};
```

```
        };  
        nestedGreeting('hello');  
    }  
};  
newObject.sayGreeting('hello');  
};
```

Без этого присваивания во втором сообщении снова стояло бы `window`, а не `object`.

6.10. Реализация рекурсивного алгоритма

Задача

Создать функцию, которая для выполнения некоторой задачи *вызывала бы сама себя*. Такая технология называется рекурсией. Она бывает полезна в работе с иерархическими структурами данных (например, деревьями узлов или встроеными массивами), в некоторых типах алгоритмов (таких как сортировка) и для некоторых математических вычислений (таких как последовательность Фибоначчи).

Решение

Рекурсия — хорошо известная концепция из области математики и информатики. Ее примером в математике является *последовательность Фибоначчи*. Каждое число последовательности Фибоначчи — это сумма двух предыдущих ее чисел:

$$\begin{aligned} f(n) &= f(n-1) + f(n-2), \\ &\text{for } n = 2, 3, 4, \dots, n \text{ and} \\ f(0) &= 0 \text{ and } f(1) = 1 \end{aligned}$$

Еще один пример математической рекурсии — это *факториал*, обычно обозначаемый восклицательным знаком ($4!$). Факториал представляет собой произведение всех целых чисел от 1 до заданного числа n . Если n равно 4, то факториал

$$4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24.$$

Такие рекурсии можно запрограммировать на JavaScript, используя несколько циклов и условий, а можно применить функциональную рекурсию. Рекурсивная функция, которая вычисляет число в последовательности Фибоначчи, выглядит так:

```
function fibonacci(n) {  
    return n < 2 ? n : fibonacci(n - 1) + fibonacci(n - 2);  
}
```

А эта функция вычисляет факториал:

```
function factorial(n) {  
    return n <= 1 ? 1 : n * factorial(n - 1);  
}
```

Обсуждение

Характерной чертой рекурсивных функций является *условие завершения*, также называемое *базовым случаем*. Рекурсивная функция не может вызывать сама себя бесконтрольно, поскольку это привело бы к бесконечному циклу до переполнения стека и аварийного завершения программы. Поэтому в рекурсивных функциях всегда проверяется некое условие, после чего принимается решение, должна ли функция снова вызвать сама себя (и перейти на более глубокий уровень рекурсии) или вернуть значение (возвращаясь на предыдущий уровень, к вызвавшей ее функции). Значение, которое будет возвращено функцией самого верхнего уровня, становится окончательным результатом, после чего операция рекурсии считается завершенной.

В примере с числами Фибоначчи проверяется значение n . Если $n < 2$, то n возвращается, если же нет, то функция `fibonacci()` вызывается снова для $(n - 1)$ и $(n - 2)$ и возвращается сумма полученных значений.

В примере с вычислением факториала при первом вызове функции значение, переданное как аргумент, сравнивается с 1. Если $n \leq 1$ (в этой простой реализации отрицательные числа не поддерживаются), то функция прекращает работу и возвращает 1. Если же $n > 1$, то возвращается n , умноженное на повторный вызов функции `factorial()` — на этот раз для $n - 1$. При каждом последующем вызове функции значение n уменьшается на единицу — и так до тех пор, пока не будет выполнено условие завершения.

При вычислении факториала все промежуточные значения, возвращаемые при каждом вызове функции, помещаются в стек оперативной памяти и хранятся там до тех пор, пока не будет выполнено условие завершения рекурсии. Затем эти значения извлекаются из памяти и возвращаются, так что состояние стека выглядит примерно так:

```
return 1;           // 0!  
return 1;           // 1!  
return 1 * 2;       // 2!  
return 1 * 2 * 3;   // 3!  
return 1 * 2 * 3 * 4; // 4!
```

Большинство рекурсивных функций может быть заменено кодом, выполняющим ту же задачу линейно — посредством того или иного цикла. Циклы могут работать быстрее, хотя разница обычно незначительна. Преимуществом рекурсии является то, что рекурсивные функции обычно бывают очень краткими и минималистичными. Вопрос о том, насколько рекурсивный код понятнее цикла, остается

спорным. (Рекурсивные функции определенно короче, благодаря чему их легче читать, но из-за того, что они ссылаются на самих себя, их логику бывает трудно понять с первого взгляда, особенно если программист прежде не использовал рекурсивные функции.)

Если рекурсивная функция вызывает сама себя снова и снова, то может возникнуть переполнение стека вызовов. В результате появляется ошибка с сообщением вроде **Out of stack space** («Выход за пределы стека»), **Too much recursion** («Слишком глубокая рекурсия») или **Maximum call stack size exceeded** («Превышен максимальный размер стека вызовов»). Точный текст сообщения и количество допустимых открытых вызовов функции зависят от конкретной реализации движка JavaScript. Но, как правило, сообщения об ошибке свидетельствуют о неправильном построении рекурсивной функции, из-за чего не удастся выполнить условие завершения и функция вызывает сама себя до бесконечности.

Объекты

Типы данных в JavaScript делятся на две категории. Одна из них — это небольшое количество *примитивных* типов, таких как строки и числа. Другая — истинные объекты, которые в JavaScript происходят от типа `Object`.

Распознать встроенные объекты JavaScript легко. У них есть конструкторы, а экземпляры таких объектов, как правило, создаются с помощью ключевого слова `new`. В виде объектов реализованы такие базовые вещи, как массивы, даты (`Date`), объекты ошибок, коллекции `Map` и `Set`, а также регулярные выражения `RegExp`.

У объектов JavaScript есть важные особенности, отличающие их от объектов в традиционных языках объектно-ориентированного программирования. Например, в JavaScript можно создавать экземпляры базового типа `Object`, а также добавлять экземплярам объектов новые свойства и функции в процессе выполнения. В сущности, можно взять уже существующий объект — вообще любой объект — и изменять его компоненты, не заботясь об определении класса.

В этой главе мы ближе познакомимся с функциональностью и особенностями типа `Object` в JavaScript. Вы научитесь использовать основные свойства `Object` для проверки, расширения и копирования объектов любого типа. А в следующей главе пойдем дальше, и вы познакомитесь с рекомендованными методами формализации собственных объектов.

7.1. Проверка того, относится ли объект к заданному типу

Задача

Есть некий таинственный объект, и мы хотим определить, к какому типу он относится.

Решение

Воспользоваться оператором `instanceof`:

```
const mysteryObject = new Date(2021, 2, 1);

if (mysteryObject instanceof Date) {
    // Мы попадем сюда, поскольку mysteryObject – это объект Date
}
```

Для того чтобы убедиться, что объект *не является* экземпляром определенного типа, можно воспользоваться оператором отрицания (`!`). Только не забудьте поставить скобки, чтобы этот оператор применялся ко всему условию `instanceof`:

```
if (!(mysteryObject instanceof Date)) {
    // Мы попадем сюда, поскольку mysteryObject – это не Date
}

// Не делайте таких ошибок!
if (!mysteryObject instanceof Date) {
    // Этот код никогда не будет выполнен
}
```

У оператора `instanceof` есть один недостаток: он не работает для примитивных значений, таких как числа, строки, значения типа `Boolean` и `BigInt`, а также для `null` и `undefined`. Продемонстрируем эту проблему:

```
const testNumber = 42;
if (testNumber instanceof Number) {
    // Этот код никогда не будет выполнен
}

const testString = 'Hello';
if (testString instanceof String) {
    // Этот код никогда не будет выполнен
}

// Следующие две проверки будут работать, так как примитивы обернуты
// в объекты, но в современном JavaScript так делать не принято
const numberObject = new Number(42);
if (numberObject instanceof Number) {
    // Этот код будет работать
}

const stringObject = new String('Hello');
if (stringObject instanceof String) {
    // Этот код будет работать
}
```

Чтобы решить эту проблему, при проверке переменной, в которой могут храниться данные одного из примитивных типов, следует использовать оператор `typeof`. В отличие от `instanceof`, в `typeof` можно задействовать одно из девяти предопределенных строковых значений (см. рецепт 2.1). Если результатом будет `object`, то затем можно взять для уточнения оператор `instanceof`:

```
const mysteryPrimitive = 42;
const mysteryObject = new Date();

if (typeof mysteryPrimitive === 'number') {
  // Этот код работает
}

if (typeof mysteryObject === 'object') {
  // Этот код работает, так как Date — это объект, а не примитив

  if (mysteryObject instanceof Date) {
    // Этот код тоже работает
  }
}
```

Обсуждение

Принцип работы оператора `instanceof` состоит в проверке *цепочки прототипов* объекта. Эта концепция разъясняется в разделе «Дополнительно: цепочки прототипов» главы 8. Цепочки прототипов могут быть разными в зависимости от того, как был сконструирован объект (подобно тому как в традиционных языках ООП объект может быть унаследован от цепочки классов). Например, в основе этой цепочки для любого объекта лежит прототип `Object`, так что следующее условие всегда выполняется:

```
if (mysteryObject instanceof Object) {
  // Это условие всегда выполняется, если только mysteryObject
  // не относится к одному из примитивных типов
}
```

Следует помнить, что примитивами являются не только числа, строки и булевы значения. Это также значения `BigInt` и `Symbol` и специальные значения `null` и `undefined`. При проверке на `instanceof Object` для всех них получим `false`.

7.2. Объединение данных с помощью объектных литералов

Задача

Объединить несколько значений в группу, чтобы создать простейший пакет данных.

Решение

Создать экземпляр типа `Object`, используя синтаксис *объектного литерала*. Для этого не понадобится ни ключевое слово `new`, ни даже имя типа `Object`. Достаточно просто написать в фигурных скобках `{ }` список свойств через запятую. Каждое свойство состоит из имени свойства, двоеточия и значения этого свойства:

```
const employee = {
  employeeId: 402,
  firstName: 'Lisa',
  lastName: 'Stanecki',
  birthDate: new Date(1995, 8, 15)
};

console.log(employee.firstName); // 'Lisa'
```

Разумеется, после создания объекта к нему, как к любому объекту JavaScript, можно добавлять новые свойства:

```
employee.role = 'Manager';
```

Этот прием работает даже в том случае, если объект объявлен как `const`, поскольку объектные литералы — это не значения (как структуры в других языках), а ссылочные типы. При добавлении свойства изменяется сам объект, а не ссылка на него. (А вот добавить переменную `employee` в новый объект в этом примере было бы нельзя, поскольку такая операция изменила бы ссылку на объект.)

Обсуждение

Благодаря синтаксису объектных литералов появляется более понятный и компактный способ быстрого создания простого объекта. Однако это лишь сокращенная запись, которая используется вместо явного создания экземпляра `Object` и присвоения ему свойств, например:

```
const employee = new Object();
employee.employeeId = 402;
employee.firstName = 'Lisa';
employee.lastName = 'Stanecki';
employee.birthDate = new Date(1995, 8, 15);
```

Или можно использовать синтаксис «ключ — значение»:

```
const employee = new Object();
employee['employeeId'] = 402;
employee['firstName'] = 'Lisa';
employee['lastName'] = 'Stanecki';
employee['birthDate'] = new Date(1995, 8, 15);
```

Одно из самых приятных свойств синтаксиса объектных литералов — способ записи вложенных объектов, таких как `birthPlace`, в следующем примере:

```
const employee = {
  employeeId: 402,
  firstName: 'Lisa',
  lastName: 'Stanecki',
  birthPlace: {country: 'Canada', city: 'Toronto'}
};

console.log(employee.birthPlace.city); // 'Toronto'
```

С точки зрения JavaScript объектный литерал — это экземпляр базового типа `Object`. Такое упрощение позволяет легко создавать объекты из любых произвольных групп данных. Однако за все приходится платить: у таких объектов нет осмысленной *идентичности*.

Мы легко можем проверить, есть ли у такого объекта определенное свойство (рецепт 7.3), или перечислить все его свойства (рецепт 7.4), но не можем с помощью `instanceof` проверить, относится ли данный объект к определенному нами типу. Другими словами, не существует каких-либо соглашений по программированию и нет простого способа проверить, являются ли созданные нами объекты теми, которых мы ожидаем. Если в коде нужно передавать более надежные объекты или моделировать более сложные сущности с собственными методами, лучше использовать формальные классы (рецепт 8.1).



Может показаться, что удастся упростить процесс создания объекта, если написать функцию-фабрику, которая принимает параметры и строит соответствующий объект. Сам по себе такой подход неплох, но у него есть более мощная общепринятая альтернатива. Если вам понадобится создавать много объектов одинаковой структуры, обратите внимание на классы (рецепт 8.1).

Читайте также

О том, как получить все свойства объектного литерала, читайте в рецепте 7.4. Если захотите перейти к формальному определению класса, загляните в рецепт 8.1.

Дополнительно: вычисляемые имена свойств

Как вы уже знаете, есть два способа добавить свойство в объект JavaScript. Можно использовать запись через точку:

```
employee.employeeId = 402;
```

или запись «ключ — значение»:

```
employee['employeeId'] = 402;
```

Эти два варианта не равнозначны. При записи «ключ — значение» имя свойства сохраняется в виде строки. А это означает, что у нас есть возможность генерировать имена свойств при выполнении программы. Такие имена называются *вычисляемыми именами свойств*, и они очень полезны в ряде сценариев (представьте, что вы получаете какие-то внешние данные и используете их для создания соответствующего объекта):

```
const dynamicProperty = 'nickname';
const dynamicPropertyValue = 'The Izz';

employee[dynamicProperty] = dynamicPropertyValue;
// Теперь employee.nickname = 'The Izz'
```

```
const i = 10;
employee['sequence' + i] = 1;
// Теперь employee.sequence10 = 1
```

Вычисляемые имена свойств всегда преобразуются в строки. В них можно применять символы, недопустимые для обычных имен переменных, такие как пробелы. Например, так делать можно (хотя и настоятельно не рекомендуется):

```
const employee = {};
const today = new Date();

employee[today] = 42;

// Мы увидим, что 42 хранится в свойстве с именем в виде длинной строки
// наподобие "Tue May 04 2021 08:18:16 GMT-0400 (Eastern Daylight Time)"
console.log(employee);
```

Синтаксис объектных литералов тоже позволяет создавать вычисляемые свойства. Однако, поскольку в нем не используется формат со строковыми именами ключей, все вычисляемые имена свойств нужно помещать в квадратные скобки. Вот как это выглядит:

```
const dynamicProperty = 'nickname';
const dynamicPropertyValue = 'The Izz';
const i = 10;

const employee = {
  employeeId: 402,
  firstName: 'Lisa',
  lastName: 'Stanecki',
  [dynamicProperty]: dynamicPropertyValue,
  ['sequence' + i]: 1
};
```



Если имена свойств создаются динамически, то может возникнуть ситуация, когда нужно убедиться в уникальности имени свойства. Для этого используют различные самодельные обходные пути: проверяют существование свойства и добавляют к его имени порядковый номер, до тех пор пока не получится нечто уникальное, или же просто применяют GUID (globally unique identifier — глобальный уникальный идентификатор). Но в JavaScript для этого есть встроенное решение — тип `Symbol`, который в данной ситуации является наилучшим выбором (см. рецепт 7.12).

7.3. Проверка существования свойства у объекта

Задача

В процессе выполнения программы проверить, есть ли у объекта данное свойство.

Решение

Попробовать найти свойство по имени с помощью оператора `in`:

```
const address = {
  country: 'Australia',
  city: 'Sydney',
  streetNum: '412',
  streetName: 'Worcestire Blvd'
};

if ('country' in address) {
  // Этот код будет выполнен, так как
  // свойство address.country существует
}

if ('zipCode' in address) {
  // Этот код не будет выполнен, так как
  // свойства address.zipCode не существует
}
```

Обсуждение

Если мы попытаемся прочитать несуществующее свойство, то в результате получим `undefined`. Мы могли бы использовать проверку на `undefined`, но сама по себе она не является железной гарантией того, что данного свойства не существует. (Технически возможна ситуация, что такое свойство есть и его значение равно `undefined`. В этом случае проверка покажет, что свойства не существует, тогда как на самом деле это не так.) Для поиска свойства лучше задействовать оператор `in`.

Оператор `in` выполняет поиск по объекту и его цепочке прототипов. Это означает, что если создать объект `Dog`, который является наследником другого объекта, `Animal`, то `in` вернет `true`, если свойство определено в `Dog` или в `Animal`. Другой вариант — воспользоваться методом `hasOwnProperty()`, который выполняет поиск только для текущего объекта, игнорируя унаследованные свойства:

```
const address = {
  country: 'Australia',
  city: 'Sydney',
  streetNum: '412',
  streetName: 'Worcestire Blvd'
};

console.log(address.hasOwnProperty('country')); // true
console.log(address.hasOwnProperty('zipCode')); // false
```

Подробнее о наследовании читайте в рецепте 8.8.

Читайте также

В рецепте 7.4 показано, как извлечь все свойства объекта в виде массива, а в рецепте 7.5 — как проверить, является ли объект совершенно пустым.

7.4. Перебор всех свойств объекта

Задача

Исследовать все свойства объекта.

Решение

Получить массив имен свойств для данного объекта с помощью статического метода `Object.keys()`. Например, если запустить следующий код:

```
const address = {
  country: 'Australia', city: 'Sydney', streetNum: '412',
  streetName: 'Worcestire Blvd'
};

const properties = Object.keys(address);

// Выводим все свойства и их значения
for (const property of properties) {
  console.log(`Property: ${property}, Value: ${address[property]}`);
}
```

то в консоли появится следующая информация:

```
Property: country, Value: Australia
Property: city, Value: Sydney
Property: streetNum, Value: 412
Property: streetName, Value: Worcestire Blvd
```

Такая методика — исследовать объект, получить все его свойства и вывести их — подобна тому, что делает метод `console.log()`, если передать в него объект.

Обсуждение

При использовании `Object.keys()` мы получаем имена всех свойств (они также называются ключами). Но если после этого нужно получить значение соответствующего свойства объекта, то мы не можем применить для этого запись через точку (`object.propertyName`), так как имя свойства представляет собой строку. Вместо этого требуется запись, как для индексов массива, — `object['propertyName']`.

Обычно свойства появляются в той последовательности, в которой были определены, но JavaScript не гарантирует, что так будет всегда.

Метод `Object.keys()` широко применяется для подсчета свойств (вычисления *размера*) объекта:

```
const address = {
  country: 'Australia', city: 'Sydney', streetNum: '412',
  streetName: 'Worcestire Blvd'
};

properties = Object.keys(address);
console.log(`The address object has a length of ${properties.length}`);
// В данном случае размер объекта равен 4
```

Метод `Object.keys()` — лишь одно из возможных решений для исследования объектов JavaScript. Как правило, с него удобно начинать, так как в нем игнорируются унаследованные и перечисляемые свойства — именно такое поведение желательно в большинстве сценариев.

Еще один вариант — использовать цикл `for...in`, например:

```
for (const property in address) {
  console.log(`Property: ${property}, Value: ${address[property]}`);
}
```

Цикл `for...in` проходит по всей цепочке прототипов, обнаруживая все свойства, унаследованные объектом. В случае с объектным литералом `address` отличий от предыдущего примера не будет. Но если нужно часто исследовать объекты, то неоправданное использование циклов `for...in`, когда было бы достаточно обойтись `Object.keys()`, может привести к снижению производительности.



Вопреки ожиданиям, у цикла `for...in` и оператора `in` несколько разные области применения. Оператор `in` проверяет все свойства, включая перечисляемые, символьные и унаследованные. Цикл `for...in` находит унаследованные свойства, но игнорирует перечисляемые и символьные.

В JavaScript есть и другие, более специализированные функции, позволяющие находить различные подмножества свойств. Например, функция `getOwnPropertyNames()` игнорирует унаследованные свойства, а функция `getOwnPropertyDescriptors()` игнорирует унаследованные, но находит перечисляемые и символьные свойства, которые часто используются для расширяемости (см. рецепт 7.12). Эти различные методики описаны в табл. 7.1. Для получения более подробной информации обратитесь в Mozilla Developer Network — там есть полное описание всевозможных функций поиска свойств (<https://oreil.ly/rbd7z>).

Таблица 7.1. Различные способы поиска свойств объекта

Метод	Возвращает	Учитывает свойства			
		Пере- числе- мые	Непере- числе- мые	Сим- воль- ные	Унасле- дован- ные
<code>Object.keys()</code>	Массив имен свойств	Да	Нет	Нет	Нет
<code>Object.values()</code>	Массив значений свойств	Да	Нет	Нет	Нет
<code>Object.entries()</code>	Массив массивов свойств, в каждом элементе которо- го хранятся имя и значение свойства	Да	Нет	Нет	Нет
<code>Object.getOwnPropertyNames()</code>	Массив имен свойств	Да	Да	Нет	Нет
<code>Object.getOwnPropertySymbols()</code>	Массив имен свойств	Нет	Нет	Да	Нет
<code>Object.getOwnPropertyDescriptors()</code>	Массив объектов-де- скрипторов свойств, как при использовании <code>defineProperty()</code> (рецепт 7.7)	Да	Да	Да	Нет
<code>Reflect.ownKeys()</code>	Массив имен свойств	Да	Да	Да	Нет
Цикл <code>for...in</code>	Имя каждого свойства	Да	Нет	Нет	Да

Читайте также

В рецепте 7.3 показано, как проверить наличие определенного свойства с помощью оператора `in`.

7.5. Проверка того, является ли объект пустым

Задача

Определить, пуст ли объект (данный объект не имеет свойств).

Решение

С помощью `Object.keys()` получить массив свойств объекта, затем проверить его свойство `length` на равенство нулю:

```
const blankObject = {};  
if (Object.keys(blankObject).length === 0) {  
    // Этот код будет выполнен, так как в данном объекте нет свойств  
}  
  
const objectWithProperty = {price: 47.99};  
if (Object.keys(objectWithProperty).length === 0) {  
    // Этот код не будет выполнен, так как objectWithProperty не пустой  
}
```

Обсуждение

Чтобы создать пустой объект, можно воспользоваться синтаксисом объектных литералов:

```
const blankObject = {};
```

Или создать экземпляр `Object` посредством ключевого слова `new`:

```
const blankObject = new Object();
```

Пустые объекты можно получить и другими, менее распространенными методами. Например, можно взять существующий объект и удалить его свойства посредством оператора `delete`:

```
const objectWithProperty = {price: 47.99};  
delete objectWithProperty.price;  
  
if (Object.keys(objectWithProperty).length === 0) {  
    // Этот код будет выполнен, так как у objectWithProperty  
    // было только одно свойство и мы его удалили  
}
```

Поскольку объекты — это ссылочные типы, нельзя просто сравнить один пустой объект с другим пустым объектом. Например, при следующей проверке `unknownObject` не будет распознан как пустой:

```
const blankObject = {};  
const unknownObject = {};  
  
if (unknownObject === blankObject) {  
    // Мы никогда сюда не попадем  
    // Даже если unknownObject пустой, как и blankObject,  
    // эти переменные хранят ссылки на разные участки памяти  
}
```

Во многих библиотеках JavaScript, таких как `Underscore` и `Lodash`, есть метод `isEmpty()` для проверки объектов. Но проверка с `Object.keys()` ничуть не сложнее.

7.6. Объединение свойств двух объектов

Задача

Есть два простых объекта, каждый — со своими свойствами. Мы хотим объединить их данные в один объект.

Решение

Развернуть объекты с помощью spread-оператора (...) и создать из результата новый объект:

```
const address = {
  country: 'Australia', city: 'Sydney', streetNum: '412',
  streetName: 'Worcestire Blvd'
};

const customer = {
  firstName: 'Lisa', lastName: 'Stanecki'
};

const customerWithAddress = {...customer, ...address};
console.log(customerWithAddress);
// Теперь customerWithAddress содержит все три свойства
```

Обсуждение

Объединение двух объектов — простая операция, но не без потенциальных проблем. Если у обоих объектов есть свойства с одинаковыми именами, то на место свойств второго объекта (в предыдущем примере это `address`) будут тихо записаны свойства первого объекта. Вот измененная версия примера, в котором демонстрируется эта проблема:

```
const address = {
  country: 'Australia', city: 'Sydney', streetNum: '412',
  streetName: 'Worcestire Blvd'
};

const customer = {
  firstName: 'Lisa', lastName: 'Stanecki', country: 'South Korea'
};

const customerWithAddress = {...customer, ...address};
console.log(customerWithAddress.country); // Выводится 'Australia'
```

В этом примере есть два объекта со свойством `country`. При объединении объектов первым разворачивается объект `customer`, а затем — объект `address`. В результате вместо свойства `customer.country` будет записано свойство `address.country`.

7.7. Выбор способа определения свойств

Задача

Добавить новое свойство в объект легко. Но иногда нужно явно изменить свойство таким образом, чтобы лучше контролировать способы его применения.

Решение

Вместо того чтобы создавать свойство путем присваивания ему значения, можно определить свойство с помощью метода `Object.defineProperty()`. Например, рассмотрим следующий объект:

```
const data = {};
```

Предположим, что мы хотим добавить в него два свойства со следующими характеристиками:

- `type` — начальное значение, будучи установленным, не может быть изменено, удалено или модифицировано, но является перечисляемым;
- `id` — начальное значение, будучи установленным, может быть изменено, но не может быть удалено или модифицировано и не является перечисляемым.

Воспользуемся следующим кодом JavaScript:

```
const data = {};
```

```
Object.defineProperty(data, 'type', {
  value: 'primary',
  enumerable: true
});
```

```
// Пытаемся изменить неизменяемое свойство
console.log(data.type); // primary
data.type = 'secondary';
console.log(data.type); // не вышло, все равно primary
```

```
Object.defineProperty(data, 'id', {
  value: 1,
  writable: true
});
```

```
// Изменяем модифицируемое свойство
console.log(data.id); // 1
data.id = 300;
console.log(data.id); // 300
```

```
// Посмотрим, какие свойства появляются при перечислении в цикле
```

```
for (prop in data) {  
    console.log(prop); // только type  
}
```

В этом примере попытка изменить неизменяемое свойство тихо заканчивается неудачей. Но вы, скорее всего, будете работать в строгом режиме — либо потому, что код является модулем (см. рецепт 8.9), либо потому, что вы поставили директиву `'use strict'`; в начале файла JavaScript. В строгом режиме при попытке изменить неизменяемое свойство выполнение кода прекращается и выводится ошибка `TypeError`.

Обсуждение

Метод `defineProperty()` — еще один способ добавить свойство в объект. В отличие от прямого присваивания, он позволяет до некоторой степени контролировать поведение и состояние свойства. Но даже если с помощью `defineProperty()` назначить только имя и значение свойства, это не то же самое, что просто создать новое свойство. Дело в том, что свойства, созданные с помощью `defineProperty()`, являются неизменяемыми и неперечисляемыми по умолчанию.

Метод `defineProperty()` принимает три аргумента: объект, для которого назначается свойство, имя свойства и объект-дескриптор, который определяет его параметры. С этого момента все становится интереснее. Существует два типа дескрипторов. В рассмотренном примере использован *дескриптор данных*, который позволяет настраивать следующие четыре параметра:

- **configurable** — определяет, является ли дескриптор свойства изменяемым. По умолчанию равен `false`;
- **enumerable** — определяет, является ли дескриптор свойства перечисляемым. По умолчанию равен `false`;
- **value** — начальное значение свойства;
- **writable** — определяет, является ли значение свойства изменяемым. По умолчанию равен `false`.

Вместо дескриптора данных можно использовать *дескриптор доступа*, который поддерживает несколько иной набор параметров:

- **configurable** — то же самое, что для дескриптора данных;
- **enumerable** — то же самое, что для дескриптора данных;
- **get** — функция, которая будет применяться в качестве геттера свойства и станет возвращать значение этого свойства;
- **set** — функция, которая будет использоваться в качестве сеттера свойства и станет изменять значение этого свойства.

Вот пример применения `defineProperty()` для дескриптора доступа:

```
const person = {
  firstName: 'Joe',
  lastName: 'Khan',
  dateOfBirth: new Date(1996, 6, 12)
};

Object.defineProperty(person, 'age', {
  configurable: true,
  enumerable: true,
  get: function() {
    // Вычисляем возраст в годах
    const today = new Date();
    let age = today.getFullYear() - this.dateOfBirth.getFullYear();

    // Вносим поправку, если в этом году день рождения еще не наступил
    const monthDiff = today.getMonth() - this.dateOfBirth.getMonth();
    if (monthDiff < 0 ||
        (monthDiff === 0 && today.getDate() < this.dateOfBirth.getDate())) {
      age -= 1;
    }

    return age;
  }
});

console.log(person.age);
```

Здесь `defineProperty()` создает вычисляемое свойство (`age`), для которого выполняются вычисления с использованием другого свойства (`birthdate`). (Как вы могли заметить, можно обращаться к другим свойствам этого же экземпляра в геттере или сеттере посредством `this`.) Теперь строение объекта становится слишком сложным, чтобы его можно было создавать посредством синтаксиса объектных литералов. Для таких объектов лучше задействовать формальные классы, которые являются более естественным способом описания таких свойств, как геттеры и сеттеры (рецепт 8.2).

Метод `defineProperty()` позволяет *изменить* существующее свойство, вместо того чтобы создавать новое. По сути, синтаксис остался прежним — единственное отличие состоит в том, что у объекта уже есть свойство с указанным именем. Но здесь есть одно ограничение. Если при создании свойства было запрещено изменять его параметры, то при попытке вызвать для него метод `defineProperty()` получим ошибку `TypeError`.

Читайте также

В рецепте 8.2 показано, как определять свойства классов, — отчасти то же самое делает метод `defineProperty()`. В рецепте 7.8 описывается, как заморозить объект во избежание изменений его свойств.

7.8. Запрет любых изменений объекта

Задача

У нас есть некий объект, и мы хотим сделать так, чтобы его свойства нельзя было переопределить или изменить в другом месте кода.

Решение

С помощью метода `Object.freeze()` заморозить объект, запретив любые изменения:

```
const customer = {
  firstName: 'Josephine',
  lastName: 'Stanecki'
};

// Замораживаем объект
Object.freeze(customer);

// В строгом режиме этот оператор приведет к ошибке
customer.firstName = 'Joe';

// При попытке добавить свойство получим то же самое
customer.middleInitial = 'P';

// И при попытке удалить — тоже
delete customer.lastName;
```

При попытке изменить замороженный объект может произойти одно из двух. В строгом режиме будет выброшено исключение `TypeError`. При отключенном строгом режиме операция просто не будет выполнена — объект не изменится, но выполнение кода продолжится. Строгий режим всегда включен в модулях (см. рецепт 8.9) или если в начале файла JavaScript стоит директива `'use strict'`;

Обсуждение

Как вы уже знаете, объекты — это ссылочные типы. JavaScript позволяет изменять, добавлять и удалять их свойства, даже если переменная объекта объявлена как `const`.

Однако в JavaScript есть несколько статических методов класса `Object`, с помощью которых можно зафиксировать объект. Существует три способа это сделать, которые мы перечислим здесь в порядке от наименее до наиболее ограничивающего.

- `Object.preventExtensions()` — запрещает добавлять новые свойства, но оставляет возможность изменять значения свойств. Также можно удалять свойства и изменять их параметры с помощью `Object.getOwnPropertyDescriptor()`.

- `Object.seal()` — запрещает добавление, удаление и изменение параметров свойств, но позволяет изменять значения свойств. Иногда так поступают, чтобы заметить присвоение значений несуществующим свойствам, поскольку это приводит к скрытым ошибкам.
- `Object.freeze()` — запрещает любое изменение свойств. Нельзя изменять параметры свойств, добавлять новые свойства или присваивать свойствам новые значения. Объект становится неизменяемым.

При использовании строгого режима (как обычно и следует делать, если только это не тестовый код в консоли) попытка изменить замороженный объект приводит к исключению `TypeError`. Если строгий режим не применяется, то изменить свойство просто не удастся — оно сохранит исходное значение, а выполнение кода продолжится.

Для проверки того, является ли объект замороженным, можно воспользоваться дополнительным методом `Object.isFrozen()`:

```
if (Object.isFrozen(obj)) ...
```

7.9. Перехват и изменение объектов с помощью прокси-объектов

Задача

Мы хотим написать код, в котором с объектом выполняются некие действия, но не хотим размещать этот код *внутри* объекта.

Решение

Класс `Proxy` позволяет перехватывать некоторые действия с объектами. В следующем примере прокси-объект позволяет выполнить валидацию объекта `product`. Благодаря использованию прокси-объекта в коде можно применить несуществующее свойство или присвоить число свойству, которое относится к нечисловому типу данных:

```
// Это объект, который мы будем отслеживать с помощью прокси-объекта
const product = {name: 'banana'};
```

```
// Это обработчик, который используется прокси-объектом для перехвата ошибок
const propertyChecker = {
  set: function(target, property, value) {
    if (property === 'price') {
      if (typeof value !== 'number') {
        throw new TypeError('price is not a number');
      }
    }
  }
}
```

```

        else if (value <= 0) {
            throw new RangeError('price must be greater than zero');
        }
    }
    else if (property !== 'name') {
        throw new ReferenceError(`property '${property}' not valid`);
    }
    target[property] = value;
}
};

// Создаем прокси-объект
const proxy = new Proxy(product, propertyChecker);

// А теперь изменяем объект product посредством прокси-объекта
proxy.name = 'apple';

// Здесь получим ReferenceError
proxy.type = 'red delicious';

// Здесь получим TypeError
proxy.price = 'three dollars';

// Здесь получим RangeError
proxy.price = -1.00;

// Этот код пройдет проверку прокси-объекта и будет выполнен
product.price = -1.00;

```



Создав удобный прокси-объект для одного свойства, вы сможете использовать его для перехвата других действий над другими свойствами и другими объектами.

Обсуждение

Объект `Proxy` является оберткой объекта и может быть использован, для того чтобы перехватывать определенные действия, а затем выполнять дополнительные или альтернативные операции в зависимости от конкретного действия и данных объекта в момент его выполнения.

При создании объекта `Proxy` необходимо указать два параметра: объект, который будет отслеживаться, и обработчик, который станет перехватывать выбранные операции. В показанном примере обработчик перехватывает только операции изменения значений свойств. При каждом перехвате такой операции обработчик получает сам объект, изменяемое свойство и его новое значение. Затем функция проверяет имя изменяемого свойства. Если это `price`, то затем проверяется, является ли новое значение числом. Если не является, то выбрасывается исключение `TypeError`. Если это число, то проверяется, положительное ли оно. Если нет, то выбрасывается исключение `RangeError`. Наконец, обработчик снова проверяет имя свойства, и если это не `name`, то выбрасывается последнее

исключение — `ReferenceError`. Если ни одна ошибка не найдена, то свойству присваивается новое значение, как обычно.

Объект `Proxy` поддерживает значительное число проверок, полный список которых приводится в табл. 7.2. Здесь перечислены все проверки, а также параметры, принимаемые функцией-обработчиком, ожидаемое возвращаемое значение и то, как активируется данный обработчик.

Таблица 7.2. Проверки `Proxy`

Проверка <code>Proxy</code>	Параметры функции	Ожидаемое возвращаемое значение	Как активируется обработчик
<code>getOwnPropertyDescriptor</code>	Объект, имя	Описание или <code>undefined</code>	<code>Object.getOwnPropertyDescriptor(proxy, name)</code>
<code>getOwnPropertyNames</code>	Объект	Строка	<code>Object.getOwnPropertyNames(proxy)</code>
<code>getPrototypeOf</code>	Объект	Любое	<code>Object.getPrototypeOf(proxy)</code>
<code>defineProperty</code>	Объект, имя, описание	<code>Boolean</code>	<code>Object.defineProperty(proxy, name, desc)</code>
<code>deleteProperty</code>	Объект, имя	<code>Boolean</code>	<code>Object.deleteProperty(proxy, name)</code>
<code>freeze</code>	Объект	<code>Boolean</code>	<code>Object.freeze(target)</code>
<code>seal</code>	Объект	<code>Boolean</code>	<code>Object.seal(target)</code>
<code>preventExtensions</code>	Объект	<code>Boolean</code>	<code>Object.preventExtensions(proxy)</code>
<code>isFrozen</code>	Объект	<code>Boolean</code>	<code>Object.isFrozen(proxy)</code>
<code>isSealed</code>	Объект	<code>Boolean</code>	<code>Object.isSealed(proxy)</code>
<code>isExtensible</code>	Объект	<code>Boolean</code>	<code>Object.isExtensible(proxy)</code>
<code>has</code>	Объект, имя	<code>Boolean</code>	Имя в прокси-объекте
<code>hasOwn</code>	Объект, имя	<code>Boolean</code>	<code>({}).hasOwnProperty.call(proxy, name)</code>
<code>get</code>	Объект, имя, получатель	Любое	<code>receiver[имя]</code>
<code>set</code>	Объект, имя, значение, получатель	<code>Boolean</code>	<code>receiver[имя] = val</code>

Таблица 7.2 (окончание)

Проверка Proxy	Параметры функции	Ожидаемое возвращаемое значение	Как активируется обработчик
enumerator	Объект	Итератор	for (имя в прокси-объекте) (итератор должен возвращать все перечисляемые свойства объекта — как собственные, так и унаследованные)
keys	Объект	Строка	Object.keys(proxy) (возвращает массив перечисляемых собственных свойств объекта)
apply	Объект, thisArg, args	Любое	proxy(...args)
construct	Объект, args	Любое	new proxy(...args)

Прокси-объекты могут служить также оберткой для стандартных объектов JavaScript, таких как `Array` или `Date`. В следующем коде с помощью прокси-объекта переопределяется семантика того, что происходит при обращении к массиву. В случае операции `get` обработчик проверяет значение элемента массива с заданным индексом и, если это 0, возвращает `false`:

```
const handler = {
  get: function(array, index) {
    if (array[index] === 0) {
      return false;
    }
    else {
      return true;
    }
  }
};

const numbers = [1,0,6,1,1,0];
const proxy = new Proxy(numbers, handler);

console.log(proxy[2]); // true
console.log(proxy[0]); // true
console.log(proxy[1]); // false
```

Значение элемента массива с индексом 2 не равно нулю, поэтому возвращается `true`. То же самое происходит при обращении к элементу массива с индексом 0. Но значение элемента с индексом 1 равно нулю, поэтому возвращается `false`. Так происходит всякий раз при обращении к данному прокси-объекту массива.

7.10. Клонирование объектов

Задача

Создать точную копию произвольного объекта.

Решение

С помощью `spread`-оператора (`...`) распаковать объект, преобразовав его в набор свойств. Затем построить новый объект, заключив эти свойства в фигурные скобки (`{}`):

```
const animal = {
  name: 'Red Fox', class: 'Mammalia', order: 'Carnivora',
  family: 'Canidae', genus: 'Vulpes', species: 'Vulpes vulpes'
};

const animalCopy = {...animal};
console.log(animalCopy.species); // 'Vulpes vulpes'
```

Обсуждение

Может показаться, что для копирования объекта достаточно использовать следующий оператор:

```
const animalCopy = animal;
```

Для примитивных типов, таких как строки, числа или `BigInt`, это работает. Но объекты являются ссылочными типами, поэтому оператор присваивания копирует только ссылку на объект. В итоге получим две переменные, `animal` и `animalCopy`, которые ссылаются на один и тот же объект, размещенный в памяти.

Для того чтобы правильно скопировать произвольный объект, нужно создать новый объект, а затем пройтись по всем свойствам старого объекта и скопировать их все в новый. Для этого можно было бы пойти длинной дорогой, применив оператор `in` (рецепт 7.4). Но `spread`-оператор предполагает более короткий путь, позволяя сжать все действия в одну простую строку кода.

При использовании `spread`-оператора мы получаем все перечисляемые свойства объекта. Это все свойства, созданные посредством синтаксиса объектных литералов, а также новые свойства, присвоенные объекту после его создания. Однако с помощью метода `Object.defineProperty()` в объекте могли быть намеренно созданы неперечисляемые свойства (см. рецепт 7.7). Как правило, неперечисляемые свойства являются чем-то дополнительным — например, это могут быть данные, добавляемые другим сервисом как часть некой расширенной системы.



Обычно перечисляемые свойства не нужно копировать в новые объекты, поэтому вполне логично, что `spread`-оператор их игнорирует. Но есть и другие варианты. У объектов JavaScript для этого предусмотрена встроенная «сантехника», такая как метод `Object.getOwnPropertyDescriptors()`, позволяющий находить перечисляемые свойства. Подробнее о перечисляемых свойствах читайте в рецепте 7.4.

Возможно, вам встречался более старый способ клонирования объектов с помощью метода `Object.assign()`. Он эквивалентен использованию `spread`-оператора:

```
const animalCopy = Object.assign({}, animal);
```

В любом случае, обе эти операции выполняют поверхностное копирование. Если в состав объекта входят свойства, значениями которых являются массивы или другие объекты, то их содержимое не копируется. Вместо этого будут созданы ссылки, по которым эти массивы или объекты будут доступны из старого и нового объектов. Вот демонстрация этой проблемы:

```
const student = {
  firstName: 'Tazie', lastName: 'Yang',
  testScores: [78, 88, 94, 91, 88, 96]
};

const studentCopy = {...student};

// Теперь у нас есть два объекта с доступом к одному и тому же
// массиву testScores.
// Мы можем в этом убедиться, если изменим этот массив.
// Это повлияет только на копию:
studentCopy.firstName = 'Dori';
// А это — на оба объекта:
studentCopy.testScores[0] = 56;

console.log(student);
// {firstName: "Tazie", lastName: "Yang", testScores: [56, 88, 94, 91, 88, 96]}
console.log(studentCopy);
// {firstName: "Dori", lastName: "Yang", testScores: [56, 88, 94, 91, 88, 96]}
```

Такое поведение не всегда оказывается проблемой — все зависит от того, что вам требуется получить. Но если вы хотите скопировать не только верхний уровень, то нужно выбрать другую методику клонирования, которая позволяла бы создавать глубокие копии (рецепт 7.11).

Читайте также

В рецепте 7.11 показано, как получить *глубокую копию* для базовой структуры данных, подобной объекту `student`, рассмотренному в примере.

7.11. Создание глубокой копии объекта

Задача

Создать точную копию произвольного объекта. Скопировать не только верхний уровень объекта, но и все объекты, на которые он ссылается.

Решение

Не существует единого способа для глубокого копирования объектов. Разработчики используют несколько различных методов, у каждого из которых есть свои недостатки.

Самый надежный вариант состоит в том, чтобы написать собственную логику клонирования для вашего объекта. Вот пример создания глубокой копии объекта `student`, представленного в рецепте 7.10:

```
const student = {
  firstName: 'Tazie', lastName: 'Yang',
  testScores: [78, 88, 94, 91, 88, 96]
};

function cloneStudent(student) {
  // Начинаем с поверхностной копии
  const studentCopy = {...student};

  // Теперь создаем дубликат массива (разворачивая
  // его с помощью spread-оператора)
  studentCopy.testScores = [...studentCopy.testScores];

  return studentCopy;
}

// Создаем полностью независимую копию объекта student
const studentCopy = cloneStudent(student);

// Убеждаемся в том, что это разные массивы
studentCopy.testScores[0] = 56;

console.log(student.testScores[0]);    // 78
console.log(studentCopy.testScores[0]); // 56
```

Красота этого решения состоит в следующем: поскольку объект известен, то мы знаем, насколько глубокую копию нужно создать. В данном примере известно, что в массиве `testScores` содержатся числа. Таким образом, мы знаем, что для дублирования этого массива достаточно будет простого клонирования с помощью `spread`-оператора. Но если бы массив содержал объекты, то нам бы пришлось решать, нужно ли дублировать все эти объекты посредством технологии, описанной в рецепте 5.6. А если бы в `testScores` содержались какие-то другие коллекции,

такие как `Set` или `Map`, то нам пришлось бы корректно создавать новые коллекции соответствующего типа.

Если же вам нужен универсальный способ глубокого копирования произвольных объектов, то наилучшим решением на данный момент будет использование готовых, протестированных программ из известных библиотек JavaScript, таких как функция `cloneDeep()` из `Lodash`, которую можно импортировать отдельно из модуля `lodash.cloneDeep`.

Обсуждение

Сейчас много спорят о том, стоит ли встроить в будущие версии JavaScript поддержку сериализации и глубокого копирования. Однако пока что глубокое копирование остается пробелом, который приходится восполнять самостоятельно.

При разработке полноценного класса (рецепт 8.1), возможно, имеет смысл предусмотреть функцию клонирования как собственный метод этого класса:

```
class Student {
  constructor(firstName, lastName, testScores) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.testScores = testScores;
  }

  clone() {
    return new Student(this.firstName, this.lastName,
      [...this.testScores]);
  }
}

const student = new Student('Tazie', 'Yang', [78, 88, 94, 91, 88, 96]);
const studentCopy = student.clone();

// Убеждаемся в том, что это разные массивы
studentCopy.testScores[0] = 56;

console.log(student.testScores[0]);    // 78
console.log(studentCopy.testScores[0]); // 56
```

В данном примере не применяется `spread`-оператор. Вместо этого создается новый объект `Student` посредством конструктора. При использовании `spread`-оператора копия будет представлять собой не экземпляр `Student`, а экземпляр базового класса `Object`. Она будет иметь те же свойства, что и оригинал, но при проверке посредством `instanceof` (см. рецепт 7.1) не будет распознаваться как `Student`. К такой копии также будут неприменимы все методы, созданные для класса `Student`. Во избежание таких проблем необходимо при копировании всегда создавать объекты правильного типа.

Вы можете поинтересоваться: почему бы не создать универсальную функцию копирования объектов? Такие задачи в действительности сложнее, чем кажутся на первый взгляд. По интернету гуляет множество антишаблонов проектирования, способных привести к серьезным проблемам.

Наивный подход с рекурсивной логикой закончится катастрофическим сбоем (с переполнением стека) в случае цепочек объектов, ссылающихся на самих себя. Простой пример: объект ссылается на другой объект, который, в свою очередь, ссылается на первоначальный объект. Однако более тонкие версии этого подхода встречаются на удивление часто.

Еще один вариант той же проблемы — случай, когда в одном объекте есть две ссылки на один и тот же объект. Например, представим себе объект `ProductCatalog`, в котором есть массив объектов `Product`, часть из которых ссылаются на один и тот же объект `Supplier`. При наивном подходе были бы созданы несколько копий `Supplier`, по одной для каждого `Product`. В более сложной реализации, такой как `cloneDeep()` или библиотеки `Lodash`, ссылки отслеживаются по мере их поступления таким образом, чтобы один и тот же объект не создавался более одного раза. (Исходный код этой реализации клонирования — <https://github.com/lodash/lodash/blob/master/.internal/baseClone.js> — будет хорошим антидотом для всех желающих изобрести велосипед.)

Еще одна широко рекомендуемая методика предполагает использовать сериализацию JSON, чтобы преобразовать объект в строку и обратно. Здесь возможны проблемы с объектами `Date` (которые превращаются в строки), специальными значениями вроде `Infinity` и объектами, включающими в себя функции (последние в этом случае отбрасываются). Хуже всего здесь то, что вы не будете предупреждены о потере информации.



То же самое приходится учитывать и при сравнении объектов. Оператор `===` покажет только то, ссылаются ли две переменные на один и тот же объект. Если же это два разных объекта с одинаковыми данными, то оператор вернет `false`. Можно было бы написать универсальную функцию для сравнения всех свойств двух объектов, но смысл равенства зависит от типа сравниваемых данных, поэтому надежнее всего будет написать собственную функцию `isEqual()`.

7.12. Создание абсолютно уникальных ключей для свойств объекта

Задача

Добавить в объект свойство с уникальным именем и гарантировать, что никакое имя другого свойства не будет с ним конфликтовать.

Решение

Создать имя нового свойства с использованием типа `Symbol`. Затем присвоить значение свойству с этим именем посредством синтаксиса «ключ — значение»:

```
const newObj = {};  
  
// Создаем уникальное свойство, имя которого  
// не будет конфликтовать ни с каким другим свойством  
const uniqueId = Symbol();  
newObj[uniqueId] = 'No two alike';  
  
// Создаем другое свойство  
const anotherUniqueId = Symbol();  
newObj[anotherUniqueId] = 'This will not clash, eithera';  
  
console.log(newObj);
```

Любопытно, что вы никогда не увидите уникальный идентификатор, который в действительности используется типом `Symbol`. В данном примере в консоль будет выведено следующее:

```
{Symbol(): 'No two alike', Symbol(): 'This will not clash, either'}
```

Для доступа к свойству, имя которого создано с помощью `Symbol`, понадобится переменная, в которой будет храниться имя данного свойства. Вы будете использовать эту переменную всякий раз, когда потребуется получить имя свойства:

```
console.log(newObj[uniqueID]); // 'No two alike'
```

Обсуждение

Коллизии имен свойств возникают нечасто, но в JavaScript это случается чаще, чем в других языках. Отчасти причиной проблемы является то, что все свойства являются открытыми. Это значит, что при наследовании от другого класса (см. рецепт 8.8) необходимо помнить обо всех наследуемых свойствах и не использовать их имена при создании новых свойств. Однако наиболее частая причина конфликтов имен — создание некоторой системы расширений или сервиса, который требует добавлять свойства к объектам, созданным другими программистами. В этом случае вы не знаете, будут ли создаваемые вами свойства конфликтовать с теми, которые уже есть в данном объекте, так как не вы создавали структуру этого объекта.

Есть и другие обходные пути, позволяющие проверять существующие свойства и генерировать случайные имена. Но использование типа `Symbol` — это быстрое и эффективное решение. Каждое значение `Symbol` гарантированно является уникальным. Для создания такого значения нужно вызвать метод `Symbol()`.

(Поскольку `Symbol` — это не объект, а примитивный тип, то вызывать конструктор с помощью ключевого слова `new` не нужно.)

При желании можно снабдить символ описанием, которое может пригодиться при отладке:

```
newObj = {};  
const propertyName = Symbol('Log Status');  
newObj[propertyName] = 'logged';
```

Однако описание не используется при создании значений `Symbol`. Если создать два экземпляра `Symbol` с одинаковыми описаниями, то это будут два совершенно разных, уникальных идентификатора, хранящихся в глобальном реестре значений `Symbol`.

7.13. Создание перечислений с помощью `Symbol`

Задача

Создать и сохранить небольшую группу взаимосвязанных констант, на которые можно ссылаться в коде по их именам.

Решение

Присвоить значение каждой константе с помощью `Symbol()`:

```
// Создаем три константы, чтобы использовать их как перечисление  
const TrafficLight = {  
  Green: Symbol('green'),  
  Red: Symbol('red'),  
  Yellow: Symbol('yellow')  
}  
  
// В этой функции используется перечисление TrafficLight  
function switchLight(newLight) {  
  if (newLight === TrafficLight.Green) {  
    console.log('Turning light green');  
  }  
  else if (newLight === TrafficLight.Yellow) {  
    console.log('Get ready to stop');  
  }  
  else {  
    console.log('Turning light red');  
  }  
  return newLight;  
}
```

```
}  
  
let light = TrafficLight.Green;  
light = switchLight(TrafficLight.Yellow);  
light = switchLight(TrafficLight.Red);  
  
console.log(light);    // выводится "Symbol('red')"
```

Обсуждение

Перечисление (или *перечисляемый идентификатор*) — это группа именованных констант. Перечисления удобны в тех случаях, когда имеется переменная, которая может принимать только одно из небольшого множества значений. Благодаря перечислениям можно сделать код более простым и уменьшить вероятность ошибок (по сравнению с помощью «магических» чисел), так как теперь вы не забудете, что означает каждое число, и не сможете случайно использовать число, для которого не определена константа.



Вопрос о правильном соглашении относительно применения прописных букв в именах констант все еще остается открытым. В классе `Math` неизменяемые свойства, такие как `Math.PI` или `Math.E`, записываются прописными буквами. В приведенном примере имена констант перечисления начинаются с прописной буквы, как и объект, в который они обернуты, так что в итоге запись выглядит как `TrafficLight.Red`.

Константы часто создаются с числовыми или строковыми значениями. Это особенно удобно, если константа связана с еще какой-то полезной информацией, как в показанном далее преобразовании единиц измерения:

```
const Units = {  
  Meters: 100,  
  Centimeters: 1,  
  Kilometers: 100000,  
  Yards: 91.44,  
  Feet: 30.48,  
  Miles: 160934,  
  Furlongs: 20116.8,  
  Elephants: 625,  
  Boeing747s: 7100  
};
```

Если для константы перечисления не существует естественного уникального значения, то можно использовать `Symbol`. Это избавит вас от необходимости придумывать случайные числа самостоятельно. Кроме того, применение `Symbol` гарантирует уникальность — в отличие от подстановки собственных значений. (Это также исключает возможность случайно использовать одно и то же значение как жестко запрограммированное число в одних местах кода и как константу — в других: впоследствии при внесении изменений в код такие несоответствия могут

вызывать ошибки.) В приведенном перечислении `TrafficLight` значения `Symbol` применены для всех трех значений.

Недостатком использования `Symbol` является то, что значения, лежащие в основе данных этого типа, невидимы. Именно поэтому в решении, приведенном в данном рецепте, у каждого `Symbol` есть описательное имя, такое как `Symbol('red')`. Этот текст вы увидите при выводе `Symbol` в консоль или при преобразовании в строку. Если не указать описательное имя при создании `Symbol`, то в этих случаях мы увидим только стандартный текст `"Symbol()"`.

Читайте также

Для того чтобы ближе познакомиться с типом данных `Symbol`, читайте рецепт 7.12.

ГЛАВА 8

Классы

Является ли JavaScript объектно-ориентированным языком программирования? Ответ на этот вопрос зависит от того, кому вы его задаете и как формулируете. Но, по общему мнению, — *да, является*, с некоторыми оговорками.

Вне университетских кругов считается, что представление об объектно-ориентированных языках программирования обычно связано с такими концепциями, как классы, интерфейсы и наследование. До последнего времени JavaScript — язык объектно-ориентированного программирования, построенный на функциях и *прототипах*, — был парией в этом обществе. Но потом появился ES6 и все смешалось: внезапно оказалось, что классы и все остальное — это естественные конструкции языка. Что это, очередное синтаксическое украшение или коренное преобразование языка?

Ответ лежит где-то посередине. Вообще говоря, классы ES6 — более высокоуровневая конструкция языка, построенная на основе уже знакомых нам прототипов JavaScript. Но это не точное соответствие: модель классов привносит несколько новых особенностей, которые не вполне описываются моделью прототипов. Более того, похоже, что в будущем классы будут поддерживать новые объектно-ориентированные свойства, так что взаимное перекрытие этих моделей станет еще меньше.

В общем случае сейчас при разработке новых продуктов предпочитают использовать классы, однако код на основе прототипов все еще встречается довольно часто и далеко не устарел. В этой главе мы изучим основные шаблоны проектирования с помощью классов, рассмотрим и прототипы.

8.1. Создание класса для многократного использования

Задача

Создать многократно применяемый шаблон для произвольных объектов.

Решение

Воспользоваться ключевым словом `class` и присвоить классу имя. Внутри класса создать функцию конструктора, которая будет инициализировать объект. Вот как выглядит полная реализация класса `Person`:

```
class Person {
  constructor(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
  }
}

// Проверим работу класса Person — создадим объект
// Чтобы вызвать конструктор, нужно применить к классу ключевое слово new
const newPerson = new Person('Luke', 'Takei');
console.log(newPerson.firstName); // 'Luke'
```

В этом примере класс `Person` — это простой набор из двух открытых полей, `firstName` и `lastName`. Но к этому классу нетрудно добавить методы — они будут работать как функции, но без ключевого слова `function`. Вот как можно создать метод `Person.swapNames()`:

```
class Person {
  constructor(firstName, lastName, dateOfBirth) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.dateOfBirth = dateOfBirth;
  }

  // Это метод
  swapNames() {
    // Используем удобное сокращение (деструктурирующее присваивание),
    // чтобы присвоить значения сразу обоим свойствам
    [this.firstName, this.lastName] = [this.lastName, this.firstName];
  }
}

// Проверяем работу класса Person
const newPerson = new Person('Luke', 'Takei', new Date(1990, 5, 22));
newPerson.swapNames();
console.log(newPerson.firstName); // 'Takei'
```

Обсуждение

Главный элемент класса JavaScript — это функция конструктора. В сущности, класс JavaScript — это и есть функция конструктора, а также остальные методы, связанные с ее *прототипом*. Это означает, что такие методы, как `Person.swapNames()`, доступны всем экземплярам класса `Person`, поскольку все эти экземпляры происходят от одного и того же прототипа. (Если хотите глубже покопаться в этой закулисной реальности, обратите внимание на шаблон проектирования «Конструктор» в рецепте 8.4.)

У классов есть следующие специальные требования к синтаксису, которые вам придется выполнять.

- Имя функции-конструктора — всегда `constructor`.
- Ни в конструкторах, ни в методах не используется ключевое слово `function`, несмотря на то что во всех остальных отношениях те и другие объявляются именно как функции.

В конструкторе с помощью ключевого слова `this` создаются открытые поля объекта. Затем на них можно ссылаться в других методах класса — только не забывайте ставить `this` перед именами таких переменных. Эти поля можно применять и за пределами кода класса посредством уже знакомой нам записи через точку.

Вы спросите: можно ли изменить этот способ доступа? Например, можно ли сделать поля закрытыми и обернуть их в открытые свойства? На сегодняшний день — нет, нельзя. Разве что с применением самодельных решений, которые дополнительно все усложняют. Подробное обсуждение этого вопроса вы найдете в рецепте 8.2.

Как и в случае с функциями, JavaScript позволяет создавать классы в виде *выражений*, например:

```
const personExpression = class Person {
  constructor(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
  }
}

// Так не работает, потому что в данной области видимости нет класса Person
const newPerson = new Person('Luke', 'Takei');

// Так работает, потому что мы создаем новый экземпляр переменной,
// в которой хранится выражение класса
const newPerson = new personExpression('Luke', 'Takei');
```

Это специализированная, но не такая уж редкая методика. Она позволяет добавлять класс в данную область видимости. Например, эта методика может пригодиться, если мы не уверены, что класс `Person` из нашего примера не определен где-нибудь еще. (Другой способ решить проблему с коллизиями имен — использование модулей, как описано в рецепте 8.9.)

Читайте также

В рецепте 8.4 показан старомодный шаблон проектирования «Конструктор» для создания объекта. О том, как создавать свойства класса, читайте в рецепте 8.2. В рецепте 8.8 вы узнаете, как устанавливать отношения наследования между классами.

Дополнительно: класс с несколькими конструкторами

В большинстве объектно-ориентированных языков допускается, чтобы у класса было несколько конструкторов, так что при создании класса можно выбирать, какие параметры указывать. Но в JavaScript не поддерживается перегрузка конструкторов и методов.

Однако это не такое уж серьезное ограничение, как кажется: JavaScript известен свободным отношением к аргументам функций и тем, что вы не обязаны их предоставлять. Поэтому, несмотря на то что у класса `Person` всего один конструктор, который принимает три аргумента, следующие способы создания экземпляров без указания всех трех аргументов корректны:

```
const noDatePerson = new Person('Luke', 'Takei');
const firstNamePerson = new Person('Luke');
const noDataPerson = new Person();
```

У каждого класса должен быть только один конструктор, и он всегда выполняется. Даже если при создании объекта `Person` не указать ни одного аргумента, этот трехаргументный конструктор все равно будет выполнен и присвоит значения переменным `this.firstName`, `this.lastName` и `this.birthDate` (все они станут равны `undefined`). Если это неприемлемо, необходимо указать значения параметров по умолчанию — точно так же, как для обычных функций (см. рецепт 6.3).



Если создать класс без конструктора, то JavaScript автоматически сгенерирует для него пустой конструктор без аргументов. Это важно учитывать, если вы планируете использовать наследование классов (рецепт 8.8).

Другой способ передачи необязательных аргументов — передавать в конструктор объектный литерал. Таким образом при вызове конструктора можно передавать только выбранные свойства, указывая их имена:

```
const partialInfoPerson1 = new Person({
  lastName: "Takei",
  birthDate: new Date(1990, 04, 23)
});
const partialInfoPerson2 = new Person({firstName: 'Luke', lastName: 'Takei'});
```

Это распространенный шаблон разработки для JavaScript, он подробно описан в рецепте 6.5. Одним из его преимуществ является то, что в объектном литерале не обязательно соблюдать точный порядок свойств, а недостатком — то, что ничто не мешает сделать ошибку в имени параметра и такие параметры будут просто проигнорированы:

```
// Класс Person станет искать в этом объектном литерале
// свойство firstName, а свойство firstname будет
// тихо проигнорировано
const partialInfoPerson2 = new Person({firstname: 'Luke'});
```

Еще один вариант — создать класс с одним конструктором, но с несколькими статическими методами, которые будут создавать экземпляры данного объекта с разными конфигурациями. В зависимости от реализации такой подход иногда называют шаблоном разработки «Строитель» или «Фабрика». Он описан в рецепте 8.7.

8.2. Добавление в класс новых свойств

Задача

Создать в классе геттеры и сеттеры свойств в качестве оберток для содержимого класса.

Решение

Прежде всего подумайте, станет ли создание свойств класса наилучшим решением в вашем случае. (Как было показано в обсуждении ранее, у такого подхода есть ряд ограничений и он несколько противоречив.) Если вы все же решите использовать свойства, то можно для каждого из них создать методы `get` и `set`. Вот пример вычисляемого свойства `age` — оно вычисляется на основе даты, хранящейся в свойстве `this.dateOfBirth`:

```
class Person {
  constructor(firstName, lastName, dateOfBirth) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.dateOfBirth = dateOfBirth;
  }

  // Это геттер для свойства age
  get age() {
    if (this.dateOfBirth instanceof Date) {
      // Вычисляем разницу в возрасте в годах
      const today = new Date();
      let age = today.getFullYear() - this.dateOfBirth.getFullYear();

      // Вносим поправку, если в этом году еще не было дня рождения
      const monthDiff = today.getMonth() - this.dateOfBirth.getMonth();
      if (monthDiff < 0 ||
          (monthDiff === 0 && today.getDate() < this.dateOfBirth.getDate())) {
        age -= 1;
      }

      return age;
    }
  }
}

// Проверяем работу класса Person
const newPerson = new Person('Luke', 'Takei', new Date(1990, 5, 22));
console.log(newPerson.age);
```

Создавать ли только геттер, только сеттер или то и другое, решать вам. Вот пример, в котором с помощью шаблона разработки «Свойство» реализована простейшая валидация даты рождения:

```
class Person {
  constructor(firstName, lastName, date) {
    this.firstName = firstName;
    this.lastName = lastName;

    // Присваиваем date с использованием сеттера свойства,
    // чтобы нельзя было создать объект Person с неправильным состоянием
    this.dateOfBirth = date;
  }

  // Этот геттер просто возвращает дату без какой-либо обработки
  get dateOfBirth() {
    return this._dateOfBirth;
  }

  // Этот сеттер не позволит присваивать даты из будущего
  set dateOfBirth(value) {
    if (value instanceof Date && value < Date.now()) {
      // Это корректная дата
      this._dateOfBirth = value;
    }
    else {
      throw new TypeError('Birthdate needs to be a valid date in the past');
    }
  }
}

// Проверяем ограничение дат
const newPerson = new Person('Luke', 'Takei', new Date(1990, 5, 22));
console.log(newPerson.dateOfBirth);

// Это изменение допустимо
newPerson.dateOfBirth = new Date(2010, 10, 10);
console.log(newPerson.dateOfBirth);

// А это изменение вызовет ошибку
newPerson.dateOfBirth = new Date(2035, 10, 10);
```



В данном примере выбрасывается исключение (см. рецепт 10.5), которое сообщает вызывающему объекту о попытке присвоить некорректное значение. Такое решение при разработке разумное, но не всегда наилучшее. Получение ошибки при попытке присвоить значение свойству (или, что еще хуже, при попытке создать экземпляр Person с некорректной датой) не является ожидаемым поведением в JavaScript, и в вызывающем коде может быть не предусмотрена вероятность ошибки. (Альтернатива — тихо проигнорировать досадную ошибку — тоже весьма рискованна.) Поэтому вместо свойств лучше использовать методы, которые будут предоставлять потенциально проблемные данные.

Обсуждение

Есть много причин, по которым вы можете принять решение о создании процедур свойств. В их число входят следующие:

- для вычисления значения (как `Person.age`);
- для преобразования представления поля;
- для валидации перед изменением значения поля;
- для привязки каких-то других сервисов (таких как ведение журнала или тестирование), которые должны выполняться при каждом чтении или записи поля;
- для отложенной инициализации, при которой значение свойства создается или вычисляется только при обращении к этому свойству;
- для доступа к определенному свойству объекта, который хранится в данном поле.

В этом рецепте рассматриваются два примера: `Person.age` является защищенным от записи вычисляемым свойством, `Person.dateOfBirth` представляет собой изменяемое свойство с валидацией.

При использовании свойств необходимо следить, чтобы не возникли коллизии имен. Имя поля, в котором хранится значение, не должно совпадать с именем параметра свойства или конструктора. Чтобы понять причину этого, давайте подробнее рассмотрим пример с `dateOfBirth`. Конструктор принимает параметр `date` и задействует его следующим образом:

```
this.dateOfBirth = date;
```

На первый взгляд может показаться, что в этом операторе значение `date` присваивается открытому полю `this.dateOfBirth` (как обычно и происходит). Однако в данном случае `this.dateOfBirth` — это свойство `dateOfBirth`. Поэтому управление переходит к сеттеру этого свойства:

```
set dateOfBirth(value) {  
  if (value instanceof Date && value < Date.now()) {  
    // Это корректная дата  
    this._dateOfBirth = value;  
  }  
  else {  
    throw new TypeError('Birthdate needs to be a valid date in the past');  
  }  
}
```

Если новое значение проходит проверку, то оно сохраняется в открытом поле с именем `this._dateOfBirth`. Такое неуклюжее именование необходимо, поскольку `this.dateOfBirth` (свойство) и `this._dateOfBirth` (поле) находятся в одной и той же области видимости. Если использовать для обоих одно и то же имя, то

мы будем обращаться не к тому, к чему хотели (а бесконечная последовательность вызовов метода в итоге переполнит стек).

У ведущего знака подчеркивания в переменной вроде `_dateOfBirth` есть еще одно назначение. В настоящее время в JavaScript нет возможности создавать закрытые поля. Но ведущее подчеркивание говорит о том, что это поле класса *должно считаться* закрытым. После этого остается уповать на то, что в коде, применяющем данный класс, разработчики постараются не обращаться к этому полю. Если не выполнять это соглашение, то вас наверняка ждут проблемы, так как при использовании класса рано или поздно будет вызвано поле, а не свойство. И даже если вы следуете данной рекомендации, все равно нет никакой гарантии, что в вызывающем коде это тоже будут делать.

Многие разработчики JavaScript утверждают, что для языка более естественно использовать методы типа `setXxx()` и `getXxx()`:

```
class Person {
  constructor(firstName, lastName, date) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.setDateOfBirth(date);
  }

  getDateOfBirth() {
    return this._dateOfBirth;
  }

  setDateOfBirth(value) {
    if (value instanceof Date && value < Date.now()) {
      // Это корректная дата
      this._dateOfBirth = value;
    }
    else {
      throw new TypeError('Birthdate cannot be in the future');
    }
  }
}

const newPerson = new Person('Luke', 'Takei', new Date(1990, 5, 22));
console.log(newPerson.getDateOfBirth());

// Это допустимое изменение
newPerson.setDateOfBirth (new Date(2010, 10, 10));
console.log(newPerson.getDateOfBirth());

// А такое изменение приведет к ошибке
newPerson.setDateOfBirth (new Date(2035, 10, 10));
```

Это более громоздкий подход, но у него есть свои преимущества. Здесь очевидно, что вызывается метод и выполняется код, а не просто изменяется значение переменной. А при вызове кода естественно ожидать исключений при проверке типов

и других побочных эффектов. Вызов методов также предотвращает проблемы, подобные следующей:

```
// Это не то свойство, которое нам нужно (не dateOfBirth), но JavaScript
// все равно его создает, и мы не заметим эту ошибку
person.DateOfBirth = new Date(2035, 10, 10);

// Нельзя вызвать несуществующую функцию, поэтому
// опечатка ("Data" вместо "Date") приведет к ошибке,
// которую невозможно проигнорировать
person.setDataOfBirth(new Date(2035, 10, 10));
```



И в Google JavaScript Style Guide (<https://google.github.io/styleguide/jsguide.html>), и в популярном руководстве Airbnb JavaScript Style Guide (<https://github.com/airbnb/javascript>) не рекомендуется использовать геттеры и сеттеры свойств, но допускается применение методов `setXxx()` и `getXxx()`.

Со свойствами связана еще одна проблема. Внутри JavaScript геттеры и сеттеры свойств реализуются с помощью метода `Object.defineProperty()`, и это, как правило, отлично работает. Но есть ряд особых случаев, когда предпочтительнее использовать `defineProperty()` явно, так как этот метод позволяет настраивать метаданные, которые нельзя задать другим способом. Например, если нужно сделать свойство неконфигурируемым (чтобы невозможно было изменить реализацию свойства) или перечисляемым (чтобы свойство не появлялось в цикле `for...in`), следует явно вызвать `defineProperty()`. В этом случае метод `defineProperty()` обычно вызывают в конструкторе.

Читайте также

Если вы хотите использовать процедуры свойств, для того чтобы в ответ на изменение свойства вызывать какие-то другие действия (такие как запись в журнал), обратите внимание на прокси-объекты (см. рецепт 7.9). Подробнее о создании свойств с помощью метода `Object.defineProperty()` читайте в рецепте 7.7.

Дополнительно: закрытые поля

Пока что в JavaScript нет возможности сделать переменную — член класса, созданную с помощью `this`, закрытой. Зато есть множество обходных путей, и многие из них опасно креативны. В наиболее популярной реализации используется `WeakMap` для хранения внутренних данных. Этот метод работает, но предусматривает добавление самодельного слоя, который опасно усложняет все решение.

Лучше применять соглашение о подчеркивании, такое как `_firstName`, при именовании полей, к которым не следует обращаться вне класса. Когда-нибудь JavaScript восполнит этот пробел, и будет предложен некий вариант *закрытых полей класса* (<https://github.com/tc39/proposal-class-fields>). Пока что для обозначения закрытых полей предлагается использовать синтаксис со знаком `#` — такие поля

можно объявить в начале блока класса, благодаря чему класс становится само-документируемым. Вот как это выглядит:

```
// Возможно, так скоро будет выглядеть синтаксис закрытых полей класса
class Person {
  #firstName;
  #lastName;

  constructor(firstName, lastName) {
    this.#firstName = firstName;
    this.#lastName = lastName;
  }

  // Обертываем поля в свойства
  get firstName() {
    return this.#firstName;
  }
  set firstName(name) {
    this.#firstName = name;
  }

  get lastName() {
    return this.#lastName;
  }
  set lastName(name) {
    this.#lastName = name;
  }
}
```

Если вы хотите уже сейчас поэкспериментировать с этими свойствами, попробуйте скомпилировать код при помощи Babel (<https://babeljs.io>) — только учтите, что синтаксис может меняться. Любопытно, что это тот редкий случай, когда функциональность классов JavaScript *меньше*, чем у старого шаблона проектирования «Конструктор» (рецепт 8.4). Дело в том, что в шаблоне «Конструктор» для хранения закрытых переменных используются замыкания (см. рецепт 6.6).

8.3. Улучшенное строковое представление класса

Задача

Подобрать подходящее текстовое представление, которое будет использоваться в ходе преобразования объекта в строку.

Решение

Добавить в класс метод `toString()`, который будет возвращать желаемую строку:

```
class Person {
  constructor(firstName, lastName) {
    this.firstName = firstName;
```

```
        this.lastName = lastName;
    }

    toString() {
        return `${this.lastName}, ${this.firstName}`;
    }
}

const newPerson = new Person('Luke', 'Takei');
console.log(newPerson.toString()); // 'Takei, Luke'
```

Обсуждение

Предлагаемая по умолчанию для всех объектов реализация метода `toString()` выводит бесполезный текст `[object Object]`. Чтобы выводить другой текст, нужно добавить в класс метод `toString()`. Он может вызываться явно, как в этом примере, или неявно, когда объект преобразуется в строку. Например, при объединении объекта со строкой метод `toString()` вызывается автоматически:

```
const newPerson = new Person('Luke', 'Takei');
const message = 'The name is ' + newPerson;

// Теперь message = 'The name is Takei, Luke'
// Так гораздо лучше, чем 'The name is [object Object]'
```

Однако вызов для объекта `console.log()` сам по себе не приводит к вызову `toString()`. Дело в том, что у `console.log()` есть дополнительная логика, которая перебирает все свойства объекта и использует результат для построения собственной строки. Для того чтобы обойти такое поведение, нужно явно вызвать метод `toString()` или задействовать шаблонный литерал (см. рецепт 2.5). Сравним:

```
const newPerson = new Person('Luke', 'Takei');

console.log(newPerson);           // 'Person {firstName: "Luke", lastName:
'Takei'}'
console.log(`${newPerson}`);      // 'Takei, Luke'
console.log(newPerson+ '');       // 'Takei, Luke'
```

8.4. Создание произвольного класса посредством шаблона «Конструктор»

Задача

Создать классоподобную сущность, пригодную для многократного использования в коде. Для этого имеет смысл применить традиционный шаблон разработки «Конструктор», так как он подходит к уже существующему коду.

Решение

Шаблон «Конструктор» несколько устарел, однако все еще годится для создания объектов. Даже если вы планируете использовать формальные классы (рецепт 8.1), все равно стоит иметь представление об этом шаблоне, поскольку он наверняка встретится вам на практике. Он также поможет вам понять, как работают классы JavaScript.

Далее показан один из примеров класса `Person` из рецепта 8.1, написанный в виде функции с помощью шаблона «Конструктор»:

```
function Person(firstName, lastName) {  
    // Сохраняем открытые данные, используя 'this'  
    this.firstName = firstName;  
    this.lastName = lastName;  
  
    // Добавляем вложенную функцию, представляющую метод  
    this.swapNames = function() {  
        [this.firstName, this.lastName] = [this.lastName, this.firstName];  
    }  
}  
  
// Создаем объект Person  
const newPerson = new Person('Luke', 'Takei');  
console.log(newPerson.firstName); // 'Luke'  
  
newPerson.swapNames();  
console.log(newPerson.firstName); // 'Takei'
```

Обратите внимание: код, в котором используется объект, построенный на функциях, ничем не отличается от кода, в котором применяется объект на основе класса с идентичным конструктором. Таким образом, при переносе кода с шаблона «Конструктор» на формальные классы остальная часть приложения, скорее всего, не будет затронута.

Обсуждение

Классы появились в JavaScript относительно недавно. До этого разработчики вместо классов использовали функции. Это работало, так как JavaScript позволяет создавать новые экземпляры функции (объекты-функции) с помощью ключевого слова `new`. У каждой функции есть собственная область видимости и свои локальные данные.

У шаблона «Конструктор» есть несколько вариантов. Наиболее часто встречается такой: создать функцию с именем класса, которая принимает все параметры конструктора, необходимые для создания экземпляра. Внутри нее с помощью `this` создаются открытые поля. Можно создавать и обычные переменные, невидимые во внешнем коде, которые будут использоваться только конструктором и его вложенными функциями.

Есть два основных способа создания функций, имитирующих методы. В рассмотренном примере каждый метод создается с помощью функционального выражения и доступен извне посредством `this`. Поскольку функции-методы обернуты в функцию-конструктор, у них та же область видимости, что и у конструктора, и есть доступ к тем же обычным и локальным переменным, что и у конструктора. (Технически функция-конструктор создает замыкание, как показано в рецепте 6.6.)

Другой способ создания методов состоит в том, чтобы явно добавить методы в прототип функции-конструктора. На тот случай, если вам еще не приходилось иметь дело с прототипами, — это фундаментальный (но обычно скрытый) элемент языка, позволяющий объектам использовать одни и те же функции. Когда вы вызываете какой-нибудь метод, например `Person.swapNames()`, JavaScript ищет функцию `swapNames()` в конструкторе `Person` и если не находит, то ищет ее в прототипе. Если имеет место наследование, то процесс несколько усложняется, так как JavaScript выполняет поиск функции по всей *цепочке прототипов*, как показано в рецепте 8.8.

Так как же добавить функцию в прототип? Это можно сделать напрямую с помощью свойства `prototype`:

```
function Person(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
}

// Добавляем в прототип Person функцию, представляющую собой метод
Person.prototype.swapNames = function() {
    [this.firstName, this.lastName] = [this.lastName, this.firstName];
}

const newPerson = new Person('Luke', 'Takei');
newPerson.swapNames();
console.log(newPerson.firstName); // 'Takei'
```

Этот пример работает практически так же, как версия с функциями, вложенными в конструктор. За единственным исключением: в предыдущем примере в каждом объекте `Person` существовала своя функция `swapNames()`. Теперь есть только одна функция `swapNames()` — она находится в прототипе и используется всеми экземплярами `Person`. Это важно учитывать, если вы планируете устанавливать отношения наследования между прототипами (см. подраздел «Дополнительно: цепочки прототипов» далее в этой главе). Это также станет заметно, если вы попытаетесь задействовать закрытые переменные в замыкании (рецепт 6.6), поскольку функции, привязанные к прототипу, не существуют в контексте функции-конструктора и не будут иметь доступ к определенным в конструкторе закрытым переменным.



С помощью прототипов можно изменять поведение встроенных объектов JavaScript. Например, можно расширить функциональность базовых типов `Array` и `String`. На первый взгляд может показаться, что это отличная возможность, однако в действительности она изобилует сложностями, и так поступать настоятельно не рекомендуется, за исключением разве что разработки фреймворков. Стирание различий между стандартным и разработанным вами кодом приводит к путанице: возможно создание нестандартных шаблонов, плохо оптимизированного кода и появление скрытых ошибок. Если же несколько человек попытаются расширить функционал одного и того же встроенного объекта, то такая затея вообще закончится неудачей.

Любопытно сравнить шаблон «Конструктор» и использование ключевого слова `class`, описанное в рецепте 8.1. В обоих примерах большая часть кода выглядит одинаково.

- Пишем функцию-конструктор, которая принимает параметры и инициализирует объект.
- С помощью ключевого слова `this` создаем открытые поля.
- Для создания объекта используем ключевое слово `new` (с той разницей, что технически это не класс, а экземпляр функции).

Но есть и несколько тонких различий, в основном синтаксических. В шаблоне «Конструктор» нет выделенных свойств — методы объявляются отдельно, а не внутри конструктора и не как явно прикрепленные к прототипу конструктора (хотя при выполнении кода именно это и происходит).

Читайте также

В рецепте 8.1 показан предпочтительный способ создания шаблона произвольного объекта в современном JavaScript — с помощью ключевого слова `class`.

8.5. Создание возможности для объединения методов класса в цепочку

Задача

Определить методы класса таким образом, чтобы можно было вызывать их цепочкой, в одном выражении.

Решение

Проследить, чтобы в конце всех методов, из которых может строиться цепочка, возвращался текущий объект. Для этого при разработке класса достаточно добавить в эти методы оператор `return this`.

Вот пример разработанного нами объекта `Book`, у которого есть два метода: `raisePrice()` и `releaseNewEdition()`. Оба их можно использовать при построении цепочек:

```
class Book {
  constructor(title, author, price, publishedDate) {
    this.title = title;
    this.author = author;
    this.price = price;
    this.publishedDate = publishedDate;
  }

  raisePrice(percent) {
    const increase = this.price*percent;
    this.price += Math.round(increase)/100;
    return this;
  }

  releaseNewEdition() {
    // Записываем в publishedDate сегодняшнюю дату
    this.publishedDate = new Date();
    return this;
  }
}

const book = new Book('I Love Mathematics', 'Adam Up', 15.99,
  new Date(2010, 2, 2));

// Поднимаем цену на 15 % и меняем дату выхода книги,
// используя цепочку методов
console.log(book.raisePrice(15).releaseNewEdition());
```

Обсуждение

Непосредственный вызов одного метода для результата другого метода в одном выражении называется *цепочкой методов*. Далее в качестве примера показан вызов метода `replaceAll()` для строки. Поскольку `replaceAll()` возвращает новую строку, этот метод можно вызвать еще раз, для этой новой строки, и получить третью строку:

```
const safePieceOfHtml =
  originalPieceOfHtml.replaceAll('<', '&lt;').replaceAll('>', '&gt;');
```

Цепочка методов не обязательно должна состоять из одного и того же метода. Этот прием работает для любых методов, возвращающих данный объект. Рассмотрим, например, следующий код, в котором последовательным применением `concat()` и `sort()` сначала объединяются два массива, а затем результат сортируется:

```
const evens = [2, 4, 6, 8];
const odds = [1, 3, 5, 7, 9];

const evensAndOdds = evens.concat(odds).sort();
console.log(evensAndOdds); // [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Цепочки методов широко применяются для встроенных объектов JavaScript, а также во многих библиотеках и фреймворках языка. Для использования этого шаблона проектирования при разработке классов достаточно просто возвращать ссылку на `this` в конце методов этого класса. Вызывающий код будет игнорировать эту ссылку либо применять ее при выполнении цепочки методов.

В данном примере метод, вызванный для `Book`, изменяет объект и возвращает ссылку на измененный объект. В функции, вызвавшей этот метод, возвращаемое значение может быть проигнорировано, так как там уже есть ссылка на объект `Book`. Но многие пуристы от функционального программирования поступают иначе. Они пишут методы, которые возвращают измененную *копию* объекта, не меняя исходный объект. Вот как можно реализовать этот шаблон проектирования:

```
class Book {
  constructor(title, author, price, publishedDate) {
    this.title = title;
    this.author = author;
    this.price = price;
    this.publishedDate = publishedDate;
  }

  getRaisedPriceBook(percent) {
    const increase = this.price*percent;
    return new Book(this.title, this.author, Math.round(increase)/100,
      this.publishedDate);
  }

  getNewEdition() {
    return new Book(this.title, this.author, this.price, new Date());
  }
}
```

Такой шаблон проектирования не влияет на то, как работают цепочки методов, однако теперь вызывающая функция обязана получить возвращаемое методом значение, иначе изменения не будут получены.

8.6. Создание статических методов класса

Задача

Написать привязанную к классу утилиту, которую можно было бы вызывать, не создавая объект этого класса.

Решение

Поставить перед методом ключевое слово `static`. Проследить, чтобы в методе не было попыток использовать какие-либо поля, свойства или методы экземпляра класса. Вот пример статического метода `Book.isEqual()`:

```
class Book {
  constructor(isbn, title, author, publishedDate) {
    this.isbn = isbn;
    this.title = title;
    this.author = author;
    this.publishedDate = publishedDate;
  }

  static isEqual(book, otherBook) {
    if (book instanceof Book && otherBook instanceof Book) {
      // Объекты Book считаются равными, если у них
      // одинаковые ISBN независимо от дефисов
      return (book.isbn.replaceAll('-', '') ===
              otherBook.isbn.replaceAll('-', ''));
    }
    else {
      return false;
    }
  }
}
```

Для доступа к статическому методу используется имя класса, например `Book.isEqual()`. Статические методы нельзя вызывать для переменной объекта:

```
const firstPrinting = new Book('978-3-16-148410-0', 'A.I. Is Not a Threat',
  'Anne Droid', new Date(2019, 2, 2));

const secondPrinting = new Book('978-3-16-148410-0', 'A.I. Is Not a Threat',
  'A. Droid', new Date(2021, 2, 10));

// Сравниваем две книги посредством статического метода
const sameBook = Book.isEqual(firstPrinting, secondPrinting);
// sameBook = true

// А так не работает, потому что метод isEqual
// недоступен для экземпляров Book
sameBook = firstPrinting.isEqual(firstPrinting, secondPrinting);
```

Обсуждение

Функциональность статических методов логически связана с классом, а не с конкретным экземпляром. Хороший пример — метод `Array.isArray()`, он позволяет проверить, является ли объект массивом, при этом пользователь не обязан вначале создать массив. Некоторые классы состоят только из статических методов. Хороший пример такого класса — стандартный класс JavaScript `Math`.

В рассмотренном примере можно добавить в класс `Book` статические методы для обработки или верификации ISBN. Статические методы можно применять также для сравнения и копирования объектов класса. Этот принцип продемонстрирован в следующем решении на примере статического метода `isEqual()`. Вы также можете написать метод `compare()`, который позволит сортировать объекты класса, объединенные в массив (см. рецепт 5.16).

В статическом методе `this` ссылается не на экземпляр объекта, а на класс. Иногда это создает проблемы, так как в коде все равно допускается сохранение данных в `this` (и извлечение данных из `this`). В результате можно получить не тот эффект, которого мы ожидали. В сущности, `this` в статических методах играет роль глобальной переменной с областью видимости в пределах данного класса, и такого использования `this` лучше избегать.



Ключевое слово `this` можно применять, для того чтобы вызвать один статический метод из другого статического метода. Например, если мы хотим вызвать статический метод `isEqual()` из другого статического метода класса `Book`, то можем написать `Book.isEqual()` или, чтобы было понятнее, `this.isEqual()`.

Методы `set` и `get` для свойств класса также могут быть статическими, хотя такое их использование вызывает споры. Например, можно задействовать статический геттер для хранения константы:

```
class Book {
  constructor(isbn, title, author, publishedDate) {
    this.isbn = isbn;
    this.title = title;
    this.author = author;
    this.publishedDate = publishedDate;
  }

  // Создаем статическое, предназначенное только
  // для чтения свойство Books.isbnPrefix
  static get isbnPrefix() {
    return '978-1';
  }
}
```

Можно написать и статический сеттер, который будет играть роль глобальной переменной приложения. Однако, поскольку не существует статических конструкторов, вам придется где-то выполнить этот код, чтобы присвоить данной переменной начальное значение. Это не очень понятно, поэтому в JavaScript разрабатывается новый синтаксис свойств (<https://oreil.ly/7O28H>), который сейчас поддерживается все большим количеством последних версий браузеров. Этот синтаксис позволяет присваивать значения открытым статическим свойствам с использованием примерно такого же синтаксиса, что и для обычных переменных:

```
class Book {
  // Создаем статическое свойство Book.isbnPrefix
  static isbnPrefix = '978-1';

  constructor(isbn, title, author, publishedDate) {
    this.isbn = isbn;
    this.title = title;
    this.author = author;
    this.publishedDate = publishedDate;
  }
}
```

Лучше все-таки вообще не использовать это свойство языка — или хотя бы подождать, пока оно не станет более привычной нормой для JavaScript.

8.7. Создание объектов посредством статических методов

Задача

Создать метод, который генерировал бы объект определенной конфигурации, — возможно, чтобы обойти таким образом ограничение JavaScript на единственный конструктор класса.

Решение

Написать статический метод класса, который будет создавать и возвращать желаемый объект. Далее показан пример для класса `Book` — объект можно создать как с помощью конструктора, так и с помощью метода `Book.createSequel()`:

```
class Book {
  constructor(title, firstName, lastName) {
    this.title = title;
    this.firstName = firstName;
    this.lastName = lastName;
  }

  static createSequel(prevBook, title) {
    return new Book(title, prevBook.firstName, prevBook.lastName);
  }
}
```

Этот статический метод можно использовать так:

```
// Создаем объект Book с помощью обычного конструктора
const book = new Book('Good Design', 'Polly', 'Morfissim');

// Создаем объект sequel с помощью статического метода
const sequel = Book.createSequel(book, 'Even Gooder Design');
console.log(sequel);
```


Обсуждение

Статические методы позволяют реализовать различные порождающие шаблоны — прежде всего те, с помощью которых можно создавать экземпляры класса с заданной конфигурацией. Например, у класса `Date` в JavaScript есть свойство `now()`, которое возвращает новый объект `Date` с автоматически присвоенными ему текущей датой и временем.

Такой подход особенно полезен для создания более сложных комбинаций объектов. Например, можно дополнить предыдущий пример методом `Book.createTrilogy()`, который будет возвращать массив из трех объектов `Book`. В данном примере у этих объектов `Book` общий объект `Author` — это означает, что изменение объекта `Author` отразится на всех экземплярах `Book` со ссылкой на данный объект `Author`:

```
class Author {
  constructor(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
  }
}

class Book {
  constructor(title, author) {
    this.title = title;
    this.author = author;
  }

  static createSequel(prevBook, title) {
    return new Book(title, prevBook.author);
  }

  static createTrilogy(author, title1, title2, title3) {
    return [new Book(title1, author),
            new Book(title2, author),
            new Book(title3, author)];
  }
}

// Создаем трилогию — массив из трех книг —
// посредством метода-фабрики
const author = new Author('Koh', 'Der');
const books = Book.createTrilogy(author, 'A Sea of Fire', 'A Sea of Ice',
  'A Sea of Water');
console.log(books);
```

В отличие от конструкторов, количество статических методов, которые можно написать для реализации различных сценариев создания объектов, неограниченно.



Иногда эти статические методы называют методами-фабриками, хотя технически это не совсем точное определение. В теории объектно-ориентированного проектирования шаблон «Фабрика» используется тогда, когда точный тип создаваемого объекта заранее не известен. Например, мы могли бы написать метод `createBook()`, который проверял бы передаваемые ему аргументы и возвращал либо экземпляр класса `TechBook`, либо экземпляр класса `FictionBook`, причем оба эти класса были бы наследниками базового класса `Book`. В JavaScript можно сделать и так, однако не существует единого мнения по поводу того, хорошо ли JavaScript справляется с подобными тяжеловесными абстракциями классического ООП.

8.8. Наследование функционала другого класса

Задача

Создать произвольный класс, который бы наследовал функции другого класса.

Решение

Наследование позволяет построить один или несколько дочерних классов на базе одного родительского класса. Для того чтобы представить это в коде, в объявлении дочернего класса используется ключевое слово `extends`:

```
public class SomeChild extends SomeParent {
}
```

Например, вот класс `Triangle`, унаследованный от более примитивного родительского класса `Shape`:

```
// Это родительский класс
class Shape {
    getArea() {
        return null;
    }
}

// Это дочерний класс
class Triangle extends Shape {
    constructor(base, height) {
        // Вызываем конструктор базового класса
        super();
        this.base = base;
        this.height = height;
    }

    getArea() {
        return this.base * this.height/2;
    }
}
```

В этом примере у родительского класса (**Shape**) нет полезного функционала. Метод `getArea()` — это всего лишь заглушка. Но в других случаях базовые классы могут быть полезны и сами по себе. Например, с помощью наследования можно создать на основе класса **Book** дочерний класс **EBook**, а на основе класса **Person** — класс **Customer**.

Может показаться, что нет смысла создавать класс **Triangle** на базе **Shape**, если мы планируем применять только **Triangle**. И в слаботипизированных языках, таких как JavaScript, это зачастую действительно так! Но главная польза наследования проявляется тогда, когда с помощью одного родительского класса удается стандартизировать несколько дочерних классов:

```
class Circle extends Shape {
  constructor(radius) {
    super();
    this.radius = radius;
  }

  getArea() {
    return Math.PI * this.radius**2;
  }
}

class Square extends Shape {
  constructor(length) {
    super();
    this.length = length;
  }

  getArea() {
    return this.length**2;
  }
}
```

Теперь можно написать такой код:

```
// Создаем массив из разных фигур
const shapes = [new Triangle(15, 8), new Circle(8), new Square(7)];

// Сортируем их в порядке возрастания площади
shapes.sort( (a,b) => a.getArea()-b.getArea() );

console.log(shapes);
// Получаем такой порядок: Square, Triangle, Circle
```

Конечно, в слаботипизированном языке, таком как JavaScript, можно вызывать метод `getArea()` для объектов **Triangle**, **Circle** и **Square**, даже если у них нет общего родительского класса, в котором определен этот метод. Но формализация интерфейса посредством наследования позволяет сделать эти требования явными. Это важно и в том случае, если мы хотим выполнять проверку объектов посредством `instanceof` (см. рецепт 7.1):

```
const triangle = new Triangle(15, 8);

if (triangle instanceof Shape) {
  // Мы попадем сюда, так как triangle относится к классу Triangle,
  // который, в свою очередь, является Shape
}
```

Обсуждение

Если не написать конструктор для дочернего класса, то JavaScript создаст его автоматически. Этот конструктор станет вызывать конструктор базового класса, но не будет передавать ему аргументов.

При создании конструктора для дочернего класса *необходимо* вызвать в нем конструктор родительского класса, иначе при попытке создать экземпляр дочернего класса получим ошибку `ReferenceError`. Для того чтобы вызвать конструктор родительского класса, используется ключевое слово `super()`:

```
constructor(length) {
  super();
}
```

Если конструктор родительского класса принимает аргументы, то их необходимо передать в `super()` — так же, как при создании объекта. Вот пример для класса `EBook`, который является наследником `Book`:

```
class Book {
  constructor(title, author, publishedDate) {
    this.title = title;
    this.author = author;
    this.publishedDate = publishedDate;
  }
}

class EBook extends Book {
  constructor(title, author, publishedDate, format) {
    super(title, author, publishedDate);
    this.format = format;
  }
}
```

С помощью функции `super()` можно вызывать другие методы и свойства родительского класса. Например, если нужно вызвать из дочернего класса реализацию `formatString()`, то нужно написать `super.formatString()`.

Классы появились в JavaScript относительно недавно. Они поддерживают наследование, но многие другие свойства, присущие классам в традиционных объектно-ориентированных языках, такие как абстрактные базовые классы, виртуальные методы и интерфейсы, не имеют аналогов в JavaScript. Некоторым разработчикам нравится легковесная природа JavaScript и акцент на прототипах, другим не хватает инструментов, необходимых для построения больших

и сложных приложений. (Если вы относитесь ко вторым, то обратите внимание на TypeScript — более строгую надстройку над JavaScript.)

Однако наследование тоже требует компромиссов. Оно может подтолкнуть к написанию тесно связанных взаимозависимых классов, которые плохо адаптируются к изменениям. Хуже того, эти зависимости зачастую трудно идентифицировать, из-за чего разработчики неохотно вносят изменения в родительский класс (так называемая *проблема хрупкого базового класса*). Из-за подобных проблем в современной разработке часто отдают предпочтение не отношениям наследования, а объектам. Например, вместо того чтобы строить класс `Employee` как расширение `Person`, можно было бы создать объект `Employee`, включающий в себя свойство `Person` и прочие необходимые детали. Такой шаблон называется *композицией*:

```
class Person {
  constructor(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
  }
}

class Employee {
  constructor(person, department, hireDate) {
    // person — это полноценный объект Person
    this.person = person;

    // В этих свойствах хранится дополнительная
    // информация, не входящая в person
    this.department = department;
    this.hireDate = hireDate;
  }
}

// Создаем объект Employee, состоящий из объекта Person
// и некоторых дополнительных элементов
const employee = new Employee(new Person('Mike', 'Scott'), 'Sales', new Date());
```

Дополнительно: цепочки прототипов

Как мы помним, при создании классов в JavaScript создается прототип для всех объектов. В этом прототипе содержится реализация всех методов и свойств класса, он является общим для всех экземпляров класса. Именно прототипы — тот секрет, благодаря которому возможно наследование. Когда один класс становится наследником другого, эти классы объединяются в *цепочку прототипов*.

Рассмотрим, например, взаимоотношения между классами `Shape` и `Triangle`. У класса `Triangle` есть прототип, в котором хранится все, что было определено для дочернего класса. Но у этого прототипа, в свою очередь, есть собственный прототип — прототип класса `Shape`, в который входят все члены класса `Shape`. У прототипа `Shape`, в свою очередь, тоже есть прототип — это `Object.prototype`, на котором цепочка прототипов заканчивается.

Наследование может быть сколь угодно многоуровневым, а цепочка прототипов — гораздо длиннее, чем в этом примере. При вызове метода, такого как `Triangle.getArea()`, JavaScript ищет его во всей цепочке прототипов: сначала в прототипе `Triangle`, затем в прототипе `Shape` и, наконец, в прототипе `Object` (если и здесь подходящий метод не будет найден, то поиск заканчивается ошибкой).

Разумеется, классы появились в JavaScript относительно недавно, а прототипы существовали еще в первой версии языка. Поэтому неудивительно, что с помощью них можно построить взаимосвязи по принципу наследования даже без использования классов JavaScript. Иногда так и поступают, применяя старый шаблон проектирования «Конструктор» (рецепт 8.4), что приводит к появлению совсем уж неизящного кода:

```
// Этот класс будет родительским
function Person(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
}

// Добавляем методы в класс Person
Person.prototype.greet = function() {
    console.log('I am ' + this.firstName + ' ' + this.lastName);
}

// Это будет дочерний класс
function Employee(firstName, lastName, department) {
    // Метод Object.call() позволяет объединять в цепочки
    // функции-конструкторы
    // Привязываем конструктор Person к контексту данного объекта
    Person.call(this, firstName, lastName);

    // Добавляем дополнительные свойства
    this.department = department;
}

// Привязываем прототип Person к функции Employee
// Таким образом устанавливаем взаимоотношения наследования
Employee.prototype = Object.create(Person.prototype);
Employee.prototype.constructor = Employee;

// Теперь добавим методы в класс Employee
Employee.prototype.introduceJob = function() {
    console.log('I work in ' + this.department);
}

// При создании экземпляра функции Employee
// ее прототип будет объединен в цепочку с прототипом Person
const newEmployee = new Employee('Luke', 'Takei', 'Tech Support');

// Так можно вызывать методы Person и Employee
newEmployee.greet();           // 'I am Luke Takei'
newEmployee.introduceJob();    // 'I work in Tech Support'
```

Сейчас этот шаблон *следует считать* практически устаревшим, так как благодаря классам появился более понятный способ построения взаимоотношений наследования. Но он все еще встречается во многих долгоживущих массивах кода.

8.9. Объединение классов JavaScript в модули

Задача

Инкапсулировать классы внутри отдельного пространства имен, чтобы их можно было использовать многократно совместно с другими библиотеками, избегая конфликтов имен.

Решение

Воспользоваться появившейся в ES6 системой модулей. Для этого сделать следующее.

1. Решить, какой именно функционал должен представлять данный модуль. Разместить код соответствующих классов и функций, а также глобальные переменные в отдельном файле.
2. Решить, какие детали кода должны *экспортироваться* (быть доступными для других скриптов и в других файлах).
3. *Импортировать* в другие скрипты функции, которые вы хотите там использовать.

Далее представлен пример модуля (мы сохраним его в отдельном файле с именем `lengthConverterModule.js`):

```
const Units = {
  Meters: 100,
  Centimeters: 1,
  Kilometers: 100000,
  Yards: 91.44,
  Feet: 30.48,
  Miles: 160934,
  Furlongs: 20116.8,
  Elephants: 625,
  Boeing747s: 7100
};

class InvisibleLogger {
  static log() {
    console.log('Greetings from the invisible logger');
  }
}

class LengthConverter {
```

```

    static Convert(value, fromUnit, toUnit) {
        InvisibleLogger.log();
        return value*fromUnit/toUnit;
    }
}

export {Units, LengthConverter}

```

Здесь важная деталь — оператор `export` в конце файла. В нем перечислены все функции, переменные и классы, которые должны быть доступны в других файлах кода. В данном примере доступными сделаны константа `Units` (в действительности это перечисление) и класс `LengthConverter`, а класс `InvisibleLogger` доступным не будет.



При создании файлов модулей иногда рекомендуют использовать расширение `.mjs`. Оно явно сигнализирует о том, что данный файл является модулем ES6, благодаря чему такие инструменты, как Node и Babel, распознают эти файлы автоматически. Но если у веб-сервера `.mjs`-файлы не связаны с тем же типом MIME (`text/javascript`), что и обычные `.js`-файлы, то расширение `.mjs` может вызвать проблемы. Именно поэтому мы не применяем расширение `.mjs` в данном примере.

Теперь можно импортировать нужный нам функционал в другой модуль. Его можно разместить в отдельном файле либо в блоке `<script>` на веб-странице — именно так мы здесь и поступили. В любом случае в теге `<script>` необходимо использовать атрибут `type="module"`.

Вот как выглядит готовая страница с кнопкой, которая позволяет проверить работу модуля с помощью функции `doSampleConversion()`:

```

<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <meta http-equiv="X-UA-Compatible" content="ie=edge" />
    <title>Module Test</title>
  </head>
  <body>
    <h1>Module Test</h1>
    <button id="convertButton">Do Sample Conversion</button>

    <script type="module">
      import {Units, LengthConverter} from './lengthConverterModule.js';

      function doSampleConversion() {
        const lengthInMiles = 495;

        // Это работает, так как у нас есть доступ
        // к LengthConverter и Units
        const lengthInElephants =

```



```
        LengthConverter.Convert(lengthInMiles, Units.Feet, Units.Yards);
        alert(lengthInElephants);
        // А это не будет работать, так как у нас нет
        // доступа к InvisibleLogger
        // InvisibleLogger.log();
    }

    // Подключаем функцию к кнопке
    document.getElementById('convertButton').addEventListener('click',
        doSampleConversion);
</script>
</body>
</html>
```

Обсуждение

За годы существования JavaScript появился ряд модульных систем, самые известные из которых Node и npm. Но после выхода ES6 у JavaScript сформировался собственный модульный стандарт, который изначально поддерживается всеми современными браузерами.

Перед тем как создавать собственное модульное решение, стоит учесть следующие моменты.

- Ограничения по безопасности браузеров таковы, что вы не сможете запустить пример с модулем на локальном компьютере. Необходимо разместить его на веб-сервере, предназначенном для разработки (как описано в рецепте 1.9).
- Модули ограничены собственной специальной модульной областью видимости. Нельзя получить доступ к модулю из обычного, немодульного скрипта. Также нельзя получить доступ к модулям из консоли разработчика.
- Нельзя получить доступ к модулям на веб-странице из HTML. Это означает, что вы не сможете связать HTML-элемент с обработчиком события посредством HTML-атрибута, такого как `onclick`, поскольку страница не имеет доступа к обработчику события, который находится в модуле. Необходимо, наоборот, обращаться *изнутри* модуля к окружающему контексту браузера через `window` или `document`.
- Модули автоматически выполняются в строгом режиме (см. рецепт 1.4).

Функции модулей могут импортироваться только в другой модуль. Если вы хотите создать блок модуля `<script>` на веб-странице, не забудьте присвоить атрибуту `type` значение `module`, иначе импортирование модуля не будет работать:

```
<script type="module">
```

При импортировании функционала из модуля необходимо указать путь к файлу модуля в разделе `from` оператора `import`. Для модулей действует удобная сокращенная запись, позволяющая указывать относительные пути,

начинающиеся с `./`, — например, `./lengthConverterModule.js` означает, что файл `lengthConverterModule.js` находится в текущей папке:

```
import {Units, LengthConverter} from './lengthConverterModule.js';
```

При импортировании функционала из модуля действует очень гибкая система именования. Импортируемые элементы могут быть обернуты в *модульный объект*, который является особым видом контейнера с собственным пространством имен. Вот пример импорта всех экспортируемых типов в модуль с именем `LConvert`:

```
import * as LConvert from './lengthConverterModule.js';

// Теперь мы можем обращаться к LengthConverter как
// к LConvert.LengthConverter
```

Обратите внимание: при использовании модульных объектов фигурные скобки не ставятся.

Также можно применить в модуле экспорт *по умолчанию*:

```
export default LengthConverter
```

После чего импортировать, взяв любое имя:

```
import LConvert from './lengthConverterModule.js';
```

Импортирование по умолчанию соответствует аналогичному функционалу в других модульных системах. Это упрощает перенос модулей на стандарты ES6. Скорее всего, модули ES6 в итоге станут доминирующим модульным стандартом в JavaScript. Однако пока реализация ES-модулей в `npm` все еще сыровата. Это значит, что в обозримом будущем разработчики будут балансировать как минимум между двумя модульными стандартами: ES6, который по умолчанию распознаётся современными браузерами, и более старым и зрелым стандартом CommonJS, который хорошо зарекомендовал себя в Node и экосистеме `npm`.

Читайте также

Об использовании модулей CommonJS в Node и `npm` читайте в главе 18.

Асинхронное программирование

Изначально JavaScript создавался как язык однопоточного программирования с одним стеком вызовов, одной кучей в памяти и способностью выполнять процедуры кода исключительно по очереди, одну за другой. Но за последние годы он значительно вырос. В нем появилась возможность передавать сообщения по сети, читать файлы, ожидать подтверждение от пользователя, то есть выполнять операции, требующие некоторого времени и способные заблокировать пользовательский интерфейс. Для безопасного их выполнения в JavaScript появились собственные шаблоны асинхронного программирования.

Поначалу поддержка асинхронного программирования ограничивалась *обратными вызовами*. При обратном вызове мы запрашиваем некую операцию, например получение изображения из интернета, и браузер выполняет ее в другом потоке, отдельно от кода приложения. Когда загрузка изображения завершается и приложение освобождается, JavaScript выполняет обратный вызов и передает данные в код. В итоге код приложения остается однопоточным, но у нас появляется возможность выполнять асинхронные действия через набор стандартизированных веб-API.

Обратные вызовы до сих пор широко используются в JavaScript, но за последние годы в дополнение к ним появились более совершенные языковые функции, такие как *промисы*, а также ключевые слова `async` и `await`. Лежащая в основе этих функций внутренняя начинка остается прежней, но теперь стало возможно строить сложные приложения, способные управлять конкурентными асинхронными задачами, поддерживать последовательности асинхронных вызовов и корректно обрабатывать неожиданные ошибки.

В этой главе мы будем применять обратные вызовы и промисы для управления асинхронными задачами. Вы также узнаете, как выйти за пределы однопоточной модели JavaScript и выполнять непрерывную фоновую обработку посредством Web Worker API.

9.1. Обновление страницы в цикле

Задача

Обновить страницу во время выполнения длительной операции с высокой нагрузкой на процессор, так чтобы браузер не перерисовывал окно, пока оно занято.

Решение

Поставить задачу в очередь, периодически задействуя функцию `setTimeout()`. Вопреки названию функции, в `setTimeout()` не нужно указывать задержку. Наоборот, задать нулевую задержку, чтобы следующий этап операции выполнялся сразу же, как только освободится пользовательский интерфейс.

Например, рассмотрим цикл, в котором на протяжении 10 с (10 000 мс) увеличивается счетчик. После каждой итерации цикла выполняется попытка изменить текст в элементе `<p>` с именем `status`:

```
function doWork() {
    // Получаем элемент <p>, который будем изменять
    const statusElement = document.getElementById('status');

    // Отслеживаем время и количество итераций цикла
    const startTime = Date.now();
    let counter = 0;

    statusElement.innerText = 'Processing started';

    while ((Date.now() - startTime < 10000)) {
        counter += 1;
        statusElement.innerText = `Just generated number ${counter}`;
    }

    statusElement.innerText = 'Processing completed';
}
```

Если запустить этот код, то мы не увидим ни одного сообщения **Just generated number**. Вместо этого страница будет недоступна в течение 10 с, после чего появится надпись **Processing completed**.

Для того чтобы это исправить, перенесем выполнение задачи (в данном случае приращение счетчика и вывод сообщения) в отдельную функцию. Затем, вместо того чтобы многократно вызывать эту функцию в цикле, станем вызывать ее из `setTimeout()`. Каждый раз, когда в функции увеличивается счетчик, страница будет обновляться, после чего снова вызываться `setTimeout()` для следующего выполнения функции. Так продолжается до тех пор, пока не истечет 10-секундный временной интервал:

```
function doWorkInChunks() {
    // Получаем элемент <p>, который будем изменять
```

```
const statusElement = document.getElementById("status");

// Отслеживаем время и количество итераций цикла
const startTime = Date.now();
let counter = 0;

statusElement.innerText = 'Processing started';

// Создаем анонимную функцию, которая выполняет один этап задачи
const doChunkedTask = () => {
  if (Date.now() - startTime < 10000) {
    counter += 1;
    statusElement.innerText = `Just generated number ${counter}`;

    // Снова вызываем функцию, для следующего этапа
    setTimeout(doChunkedTask, 0);
  }
  else {
    statusElement.innerText = 'Processing completed';
  }
};

// Запускаем процесс, вызывая функцию в первый раз
doChunkedTask();
}
```

Здесь в переменной `doChunkedTask` хранится анонимная функция, определенная посредством стрелочного синтаксиса (см. рецепт 6.2). Использовать анонимную функцию или стрелочный синтаксис не обязательно, но это упрощает код. Функция `doChunkedTask` получает доступ ко всем переменным, которые находятся в той же области видимости, в которой она создана, в том числе к `startTime` и `statusElement`. В результате можно не беспокоиться о передаче информации в функцию — в отличие от ситуации, когда функция объявлена отдельно.

Запустив этот код, мы увидим на веб-странице быстро сменяющие друг друга цифры, а через 10 с появится сообщение о завершении работы.

Обсуждение

В JavaScript есть полноценное решение для асинхронной работы с помощью технологии *Web Workers* (рецепт 9.7). Но такой уровень сложности нужен далеко не всегда. *Web Workers* — хороший инструмент для длительных задач, асинхронных операций, которые должны получать фрагменты данных для обработки, или же асинхронных операций, для которых нужно предусмотреть возможность отмены. Но если речь идет об относительно короткой задаче с более скромными требованиями — например, нужно всего лишь обновить страницу в течение краткой, но интенсивной нагрузки на процессор, — то прием с `setTimeout()` отлично подойдет.

В рассмотренном примере метод `setTimeout()` вызывается многократно. При этом страница каждый раз теряет управление и ждет, пока браузер запланирует

выполнение запрошенной функции, а браузер это сделает, как только освободится основной поток приложения (в данном случае практически немедленно). Для того чтобы понять, как все это работает, важно осознать, что `setTimeout()` не устанавливает *точное* время запуска функции. Вместо этого задается *минимальный* временной интервал. Когда таймер останавливается, `setTimeout()` дает браузеру команду выполнить функцию. Но когда именно будет выполнен этот запрос, зависит от браузера. Если он занят, то запрос будет отложен. (И даже если не занят, тоже — в современных браузерах частота запросов ограничена, чтобы один и тот же запрос выполнялся не чаще одного раза в 4 мс.) Но это, по сути, очень маленькие задержки, и вызов `setTimeout()` со значением 0 мс приводит к тому, что код выполняется практически немедленно.

Метод `setTimeout()` — не единственный метод JavaScript для выполнения заданий по таймеру. Еще есть метод `window.setInterval()`, который вызывает функцию постоянно, выдерживая заданный интервал перед каждым последующим вызовом. Если же вы захотите применить таймер для создания анимации (например, для перерисовывания объектов в `<canvas>`), то лучше воспользоваться методом `requestAnimationFrame()`, который синхронизируется с операциями перерисовки в браузере. Таким образом, вы не будете зря тратить ресурсы на вычисление анимации чаще, чем она может быть показана.



И `setTimeout()`, и `setInterval()` относятся к древней части JavaScript. Но оба эти метода ничуть не устарели. Для более сложных сценариев следует использовать Web Workers, вместо того чтобы писать собственные решения на основе `setTimeout()` и `setInterval()`. Однако оба эти метода все еще применимы.

Читайте также

В рецепте 9.7 показано, как выполнять более сложные операции в фоновом режиме посредством Web Workers.

9.2. Использование функции, которая возвращает промис

Задача

Выполнить код после завершения асинхронной задачи (успешного или неудачного). Получить сообщение о завершении задачи с помощью объекта `Promise`.

Решение

`Promise` — это объект, который помогает управлять асинхронными задачами. Он следит за состоянием задачи и, что самое главное, выполняет обратные вызовы,

которые сообщают остальному коду, успешно или нет завершилось выполнение задачи. Технически промисы не привносят в JavaScript новый функционал, но благодаря им гораздо проще наладить четкое взаимодействие в последовательности асинхронных операций.

Для того чтобы можно было применять промисы, они должны поддерживаться тем API, к которому вы обращаетесь. Выяснить это легко: если в API поддерживаются промисы, то там есть методы, возвращающие объекты `Promise`. В более старых API, где *не используются* промисы, предлагается предоставить одну или несколько функций обратного вызова или создать обработчик определенного события. (О том, как применять промисы для API на основе обратного вызова, читайте в рецепте 9.3.)

Чтобы определить, что должно происходить после завершения промиса, нужно вызвать метод `Promise.then()` и передать в него функцию. Чтобы определить, что должно происходить в случае ошибки, нужно вызвать `Promise.catch()` и передать туда другую функцию. Для добавления некоего кода очистки, который будет выполняться после успешного или неудачного завершения промиса, нужно вызвать `Promise.finally()` и передать туда третью функцию.

Вот как выглядит наивная реализация промисов с использованием Fetch API:

```
// Создаем промис
const promise = fetch(
  'https://upload.wikimedia.org/wikipedia/commons/b/b2/Eagle_nebula_pillars.jpg');

// Предоставляем функцию, записывающую в журнал успешные запросы
promise.then( function onSuccess(response) {
  console.log(`HTTP status: ${response.status}`);
});

// Предоставляем функцию, которая записывает ошибки
promise.catch( function onError(error) {
  console.error(`Error: ${error}`);
});

// Предоставляем функцию, которая выполняется в любом случае
promise.finally( function onFinally() {
  console.log('All done');
});
```

Если вызов завершится успешно, то в окне консоли появится статус HTTP и сообщение «All done». В этом примере показана структура простейшего вызова промиса, но обычный код на основе промисов выглядит иначе, на что есть две причины. Во-первых, чтобы код был компактнее и его было удобнее читать, предпочтительнее объявлять функции с помощью стрелочного синтаксиса (см. рецепт 6.2). Во-вторых, методы `then()`, `catch()` и `finally()` обычно объединяются в цепочку и используются в одном операторе. Так можно делать, поскольку все эти методы возвращают один и тот же объект `Promise`.

Вот так выглядит более компактный типичный способ записи этого кода:

```
fetch(
  'https://upload.wikimedia.org/wikipedia/commons/b/b2/Eagle_nebula_pillars.jpg')
.then(response => {
  console.log(`HTTP status: ${response.status}`);
})
.catch(error => {
  console.error(`Error: ${error}`);
})
.finally(() => {
  console.log('All done');
});
```



Этот пример использования промисов состоит из единственного оператора, в любом месте которого можно разорвать строку. Мы применили здесь одно из общепринятых соглашений: разрывать оператор непосредственно перед точкой, чтобы следующая строка начиналась с `.then` или `.catch`. Благодаря этому код легко читать, а его схема обработки ошибок подобна той, что используется в синхронном коде. Эта структура задействуется также в форматировщике кода Prettier (см. рецепт 1.11).

Обсуждение

Объект **Promise** — это не результат, а *заменитель* результата, который появится в будущем.

Как только вы создадите объект **Promise**, начнет выполняться его код. Может даже случиться так, что **Promise** завершит работу прежде, чем будет вызван метод `then()` или `catch()`. Это не повлияет на работу кода в целом. Если вызвать `then()` для промиса, который уже завершился (успешно), или `catch()` для промиса, который был отклонен (с ошибкой), то код все равно будет выполнен.

Далее показано простое решение с цепочкой методов, в которую объединены функции обработки успешного (`then()`) и неудачного (`catch()`) выполнения задачи. В цепочки часто объединяют также несколько асинхронных задач, чтобы они выполнялись одна за другой. Хороший пример такого объединения — функция `fetch()`. Она возвращает промис, который принимает ответ сервера, когда он поступит. Но если нужно почитать тело сообщения, то необходимо запустить следующую асинхронную операцию. (Нет, это не лишние проблемы, как может показаться, — это совершенно осмысленная операция, поскольку объем передаваемых данных может быть огромным и вряд ли вы захотите рисковать блокировкой кода, пока будете получать их. В JavaScript операции ввода/вывода всегда выполняются асинхронно.)

Вот пример выполнения асинхронного запроса `fetch`, после которого выполняется чтение результата как бинарного потока с помощью метода `response.blob()`, возвращающего еще один объект **Promise**. Затем для этого объекта вызывается метод `then()`, который выполняет третий этап — преобразование бинарного

потока в строку в формате Base64, которую можно вывести на экран в виде элемента ``:

```
fetch(
  'https://upload.wikimedia.org/wikipedia/commons/b/b2/Eagle_nebula_pillars.jpg')
.then(response => response.blob())
.then(blob => {
  const img = document.getElementById('imgDownload');
  img.src = URL.createObjectURL(blob);
}));
```

Хорошее форматирование кода очень важно, так как цепочки промисов бывают довольно длинными. Однако если разместить асинхронные вызовы последовательно, то они будут выглядеть как обычный линейный код. Это гораздо лучше, чем прежние пирамиды вложенных функций обратного вызова — разработчики называли их *адом обратных вызовов*.

Когда в цепочку объединены несколько промисов, методы `catch()` и `finally()`, если вы решите их использовать, можно вызывать в конце всей цепочки. Таким образом можно собрать в одном месте все необработанные ошибки, которые могут появиться на любом этапе цепочки промисов. В функции `then()` можно даже выбрасывать собственные исключения, которые будут означать неудачу и завершать выполнение всей цепочки:

```
fetch(
  'https://upload.wikimedia.org/wikipedia/commons/b/b2/Eagle_nebula_pillars.jpg')
.then(response => {
  if (!response.ok) {
    // Как правило, если сервер ответил на запрос, это не ошибка
    // Мы будем считать ошибкой любой ответ, кроме HTTP 200 OK
    throw new Error(`HTTP code: ${response.status}`);
  }
  else {
    return response.blob();
  }
})
.then(blob => {
  const img = document.getElementById('imgDownload');
  img.src = URL.createObjectURL(blob);
})
.catch(error => {
  console.log('An error occurred in the first or second promise');
}));
```

Если возникает необработанная ошибка, то вся цепочка промисов прерывается. В ответ на такую ошибку можно внести запись в журнал или выполнить какую-либо диагностику, но нельзя продолжить выполнение остальных промисов в цепочке. Если пропустить ошибку в промисе, то это в итоге приведет к появлению события `window.unhandledrejection`, а если и в этот момент выполнение не будет прервано, то в консоль будет выведено сообщение об ошибке.

Читайте также

Fetch API более подробно рассматривается в главе 13. В рецепте 9.4 описывается, как с помощью промисов связывать между собой конкурентные задачи. В рецепте 9.5 показано, как можно использовать метод `fetch()` с ключевым словом `await`.

9.3. Замена асинхронной функции с обратным вызовом на промис

Задача

Заменить асинхронную функцию с обратным вызовом на промис.

Решение

Создать еще одну функцию и обернуть в нее асинхронную функцию. Эта новая функция будет создавать и возвращать объект `Promise`. Когда асинхронная задача будет выполнена, эта функция вызовет либо `Promise.resolve()` в случае успешного завершения, либо `Promise.reject()` в случае неудачного завершения задачи.

Вот пример функции, которая работает точно так же, как обычная, асинхронная функция с обратным вызовом. Асинхронное выполнение функции обеспечивается за счет таймера:

```
function factorializeNumber(number, successCallback, failureCallback) {
  if (number < 0) {
    failureCallback(
      new Error('Factorials are only defined for positive numbers'));
  }
  else if (number !== Math.trunc(number)) {
    failureCallback(new Error('Factorials are only defined for integers'));
  }
  else {
    setTimeout( () => {
      if (number === 0 || number === 1) {
        successCallback(1);
      }
      else {
        let result = number;
        while (number > 1) {
          number -= 1;
          result *= number;
        }
        successCallback(result);
      }
    }, 5000); // Жестко закодированная 5-секундная задержка
              // имитирует длительный асинхронный процесс
  }
}
```

Нет никакой пользы в том, чтобы вычислять факториалы асинхронно или с помощью таймера. Это просто демонстрация того, как работают старые API с обратными вызовами.

В настоящее время вместо этого можно использовать такие функции:

```
function logResult(result) {
    console.log(`5! = ${result}`);
}

function logError(error) {
    console.log(`Error: ${error.message}`);
}

factorializeNumber(5, logResult, logError);
```

Простейший способ сделать функцию `factorializeNumber()` промисом — создать новую функцию, которая будет служить оберткой для `factorializeNumber()`:

```
function factorializeNumberPromise(number) {
    return new Promise((resolve, reject) => {
        factorializeNumber(number,
            result => {
                resolve(result);
            },
            error => {
                reject(error);
            }
        ));
    });
}
```

Теперь можно вызвать `factorializeNumberPromise()`, получить объект `Promise` и обработать результат посредством `Promise.then()`:

```
factorializeNumberPromise(5)
    .then( result => {
        console.log(`5! = ${result}`);
    });
```

Здесь также можно отслеживать возможные ошибки и даже строить цепочки асинхронных операций.

```
factorializeNumberPromise('Bad value')
    .then( result => {
        console.log(`6! = ${result}`);
    })
    .catch( error => {
        console.log(error);
    });
```

Обсуждение

Прежде чем рассмотреть это решение более подробно, важно сразу же устранить одно возможное заблуждение. Создать функцию, возвращающую объект `Promise`,

нетрудно. Но само по себе это не делает код асинхронным. Обычно код потока пользовательского интерфейса выполняется синхронно. (Это все равно что вызвать `setTimeout()` с нулевой задержкой.)

Для того чтобы обойти это ограничение, в данном примере в функции `factorializeNumber()` применен таймер, имитирующий асинхронный API. Для того чтобы код действительно выполнялся в фоновом режиме в другом потоке, необходимо использовать API Web Workers (рецепт 9.7).



В JavaScript промисы задействуются часто, но создавать их приходится очень редко. Обычно объекты `Promise` создают, для того чтобы обернуть в них старый код с обратными вызовами, как показано в этом примере.

Для того чтобы создать версию функции с промисами, нам понадобится функция, которая будет создавать объект и возвращать `Promise`. Именно это и делает функция `factorializeNumberPromise()`. Но несмотря на то, что создать объект `Promise` очень легко, функция на первый взгляд кажется сложной, так как внутри нее находятся две вложенные функции. Ее главным элементом является объект `Promise`, который служит оберткой для функции со следующей структурой:

```
function(resolve, reject) {
  ...
}
```

Функция промиса принимает два параметра, которые, в сущности, являются функциями обратного вызова. Они будут сигнализировать о завершении промиса: `resolve()` (с возвращенным значением) станет вызываться в случае успешного завершения промиса, а `reject()` (с объектом `error`) — в случае неудачи. Если же возникнет ошибка, которая не будет обрабатываться в функции промиса, то объект `Promise` перехватит ее и автоматически вызовет `reject()`, передавая ошибку дальше.

Асинхронная задача загружается в функции промиса. Либо же, как в примере с `factorializeNumberPromise()`, можно вызвать уже существующую функцию `factorializeNumber()`, которая запустит таймер. Для взаимодействия со старой функцией `factorializeNumber()` нам по-прежнему нужны функции обратного вызова. Разница состоит в том, что теперь эти функции передаются через промис, при вызове `resolve()` или `reject()`. Например, так выглядит функция `successCallback`, которая вызывает `resolve()`:

```
function(resolve, reject) {
  factorializeNumber(number,
    function successCallback(result) {
      resolve(result);
    },
    ...
  );
}
```

А так выглядит вызов функции `reject()` в случае неудачи:

```
function(resolve, reject) {  
  factorializeNumber(number,  
    function successCallback(result) {  
      resolve(result);  
    },  
    function failureCallback(error) {  
      reject(error);  
    });  
}
```



Метод `Promise.reject()` принимает один аргумент, в котором описана причина неудачи. Эта причина может быть представлена в виде объекта любого типа, однако настоятельно рекомендуется использовать экземпляр объекта `Error` либо созданный вами объект, который унаследован от `Error` (рецепт 10.6). В данном примере обратный вызов в случае неудачи уже передает объект `Error`, так что достаточно просто передать его в `reject()`.

В целом, такое решение делает код более компактным благодаря объявлению функций `successCallback`, `failureCallback` и функции промиса, в которой первые две функции описаны посредством стрелочного синтаксиса (см. рецепт 6.2).

Можно написать и обобщенную функцию, которая бы преобразовывала в промис любую функцию с обратным вызовом. В сущности, как раз такой функционал и предлагается в некоторых библиотеках, таких как `BlueBird.js`. Однако в большинстве случаев будет проще — и меньше путаницы, — если преобразовывать функции в промисы индивидуально, вместо того чтобы пытаться обернуть в промисы все старые синхронные API.

Читайте также

При разработке продуктов для среды выполнения Node можно использовать утилиту преобразования в промисы — она будет обертывать функции в промисы, как показано в рецепте 19.2.

9.4. Конкурентное выполнение нескольких промисов

Задача

Выполнить несколько промисов одновременно, а когда все они завершат работу, сообщить об этом.

Решение

С помощью статического метода `Promise.all()` объединить несколько промисов в один, после чего дождаться момента, когда все они будут успешно выполнены или один из них завершится неудачей.

Чтобы показать, как это работает, предположим, что у нас есть функция, возвращающая промис, который бездействует в течение 0–10 с, после чего завершает работу. Далее показана функция `randomWaitPromise()`, которая именно это и делает посредством `setTimeout()`. Эту функцию можно рассматривать как заменитель произвольной асинхронной операции:

```
function randomWaitPromise() {
  return new Promise((resolve, reject) => {

    // Выбираем период ожидания
    const waitMilliseconds = Math.round(Math.random() * 10000);

    // Имитируем асинхронную задачу с помощью setTimeout()
    setTimeout(() => {
      console.log(`Resolved after ${waitMilliseconds}`);

      // Возвращаем период ожидания в миллисекундах
      resolve(waitMilliseconds);
    }, waitMilliseconds);
  });
}
```

Теперь с помощью `randomWaitPromise()` можно быстро создать любое количество промисов. Для того чтобы дождаться, пока все промисы завершат работу, нужно поместить все объекты `Promise` в массив и передать его в метод `Promise.all()`. `Promise.all()` возвращает новый промис, представляющий собой результат выполнения всех созданных нами промисов. Для возвращаемого промиса можно вызвать `then()` и `catch()`, которые образуют уже знакомую нам цепочку:

```
// Создаем три промиса
const promise1 = randomWaitPromise();
const promise2 = randomWaitPromise();
const promise3 = randomWaitPromise();
const promises = [promise1, promise2, promise3];

// Ждем окончания всех промисов, затем выводим результат
Promise.all(promises).then(values => {
  console.log(`All done with: ${values}`);
});
```

В этой цепочке отсутствует `Promise.catch()`, так как данный код не может закончиться неудачей. При выполнении этого примера каждый промис, завершая работу, выводит сообщение в консоль. После того как завершится последний, самый медленный промис, будет выведено сообщение `All done`:

```
Resolved after 790  
Resolved after 4329  
Resolved after 6238  
All done with: 790,6238,4329
```



При выполнении сразу нескольких промисов обычно в каждый из них передается объект с каким-нибудь идентификатором, таким как URL или ID. Затем, когда промис заканчивает работу, он может вернуть объект, в котором хранится эта идентифицирующая информация. Таким образом можно определить, какой результат относится к какому промису. Это удобный способ отслеживания, но он не является обязательным, так как мы можем идентифицировать результаты промисов по их последовательности. Получаемые результаты размещаются в массиве в той же последовательности, что и промисы в переданном исходном массиве.

Обсуждение

Одно из преимуществ асинхронного программирования состоит в возможности сократить время ожидания. Другими словами, вместо того чтобы ждать, пока закончится одна задача, прежде чем запустить следующую, и потом снова ждать, прежде чем запустить еще одну, можно запустить все три задачи одновременно. В реальной жизни такой сценарий несколько необычен. Нам гораздо чаще встречаются асинхронные задачи, которые зависят от результатов других асинхронных задач, так что приходится строить цепочки, в которых выполняется одна задача за другой. Но если это не так, то можно сэкономить немало времени, выполняя сразу несколько промисов и ожидая их результатов с помощью `Promise.all()`.

В `Promise.all()` реализован принцип *немедленного отключения* (fail-fast behavior). Если один из промисов будет отклонен (намеренно, путем вызова `Promise.reject()`, либо вследствие необработанной ошибки), то весь составной промис, созданный с помощью `Promise.all()`, также будет отклонен, после чего будет вызвана функция, прикрепленная к цепочке промисов с помощью `Promise.catch()`. Остальные промисы все равно будут выполняться, и их результаты можно будет получить из соответствующих объектов `Promise`. Например, если `promise1` будет отклонен, то ничто не мешает вызвать `promise2.then()`, чтобы получить результат `promise2`. Однако на практике при использовании `Promise.all()` неудачное завершение одного из промисов, как правило, рассматривается как причина для прекращения всей составной операции. Иначе было бы проще выполнять промисы по отдельности либо применить один из альтернативных методов `Promise`, которые будут описаны далее.

Кроме `all()`, у `Promise` есть и другие статические методы, принимающие несколько промисов и возвращающие один комбинированный промис. Их поведение немного различается.

- `Promise.allSettled()` — завершается успешно после того, как все промисы будут либо завершены, либо отклонены. (В отличие от `Promise.all()`, ко-

торый завершает работу успешно, только если все промисы завершились успешно.) Функция, которая прикрепляется к `Promise.then()`, получает массив результирующих объектов, по одному для каждого промиса. Каждый результирующий объект состоит из двух свойств: `status` показывает, был ли промис выполнен или отклонен, а в `value` содержится возвращаемое значение либо объект `Error`.

- `Promise.any()` — завершается успешно, если хотя бы один промис завершился успешно. Возвращает только значение этого промиса.
- `Promise.race()` — завершается успешно, если хотя бы один промис завершился успешно либо был отклонен. Это самый специализированный из методов `Promise`, но с его помощью можно строить собственные системы планирования, которые ставят в очередь новые запросы по мере выполнения уже существующих.

9.5. Ожидание выполнения промиса с помощью `await` и `async`

Задача

Вместо того чтобы строить цепочку промисов, написать линейную логику, которую было бы проще читать и которая была бы больше похожа на синхронный код.

Решение

Вместо того чтобы вызывать `Promise.then()`, применить к промису ключевое слово `await`:

```
console.log('taskPromise is working asynchronously');
await taskPromise;
console.log('taskPromise has finished');
```

Код, идущий после `await`, не будет выполняться, до тех пор пока промис, указанный после `await`, не завершится успешно либо не будет отклонен. Выполнение кода приостанавливается, но поток не блокируется, пользовательский интерфейс по-прежнему доступен, остальные таймеры и события по-прежнему могут срабатывать.

Но здесь есть одна загвоздка. Ключевое слово `await` можно применять только внутри `async`-функции. Это означает, что для использования `await` нужно немного переписать код. Рассмотрим для примера функцию `fetch()` из рецепта 9.2. С промисами она выглядит так:

```
const url =
  'https://upload.wikimedia.org/wikipedia/commons/b/b2/Eagle_nebula_pillars.jpg';
```



```
fetch(url)
  .then(response => {
    // Функция fetch завершена
    console.log(`HTTP status: ${response.status}`);
    console.log('All asynchronous steps completed');
  })
```

Для использования ключевых слов `async` и `await` этот код нужно структурировать так:

```
async function getImage() {
  const url =
    'https://upload.wikimedia.org/wikipedia/commons/b/b2/Eagle_nebula_pillars.jpg';

  const response = await fetch(url);

  // Операция fetch завершена, промис выполнен либо отклонен
  console.log(`HTTP status: ${response.status}`);
}

getImage().then(() => {
  console.log('All asynchronous steps completed');
});
```

Вместо метода `Promise.catch()` для операций `await` можно использовать традиционные блоки отслеживания исключений:

```
async function getImage() {
  const url =
    'https://upload.wikimedia.org/wikipedia/commons/b/b2/Eagle_nebula_pillars.jpg';

  try {
    const response = await fetch(url);
    console.log(`HTTP status: ${response.status}`);
  }
  catch(err) {
    console.error(`Error: ${error}`);
  }
  finally {
    console.log('All done');
  }
}
```

Если вызов всего один, то преимущество `await` относительно невелико. Однако в случае последовательности асинхронных операций, которые должны выполняться одна за другой, `await` позволяет значительно сократить код. Обычно эта задача решается посредством цепочки промисов с многократным вызовом `Promise.then()`. Но благодаря `await` удастся получить код, гораздо более похожий на обычный код синхронных операций. Вот пример чтения изображения, который повторяет пример из рецепта 9.2, с отправкой асинхронного веб-запроса и последующим асинхронным чтением возвращаемых данных изображения:

```
async function getImage() {
  const url =
    'https://upload.wikimedia.org/wikipedia/commons/b/b2
                                     /Eagle_nebula_pillars.jpg';

  // Ожидаем ответа (асинхронно)
  const response = await fetch(url);

  if (response.ok) {
    // Ожидаем окончания чтения blob (асинхронного)
    const blob = await response.blob();

    // А теперь выводим изображение
    const img = document.getElementById('imgDownload');
    img.src = URL.createObjectURL(blob);
  }
}
```

Обсуждение

Ключевое слово `await` обрабатывает промисы способом, который выглядит как синхронный код, но не блокирует работу приложения. Рассмотрим следующее выражение:

```
const response = await fetch(url);
```

Если посмотреть на код, то может показаться, что выполнение останавливается и функция `fetch()` становится синхронной. Но на самом деле JavaScript берет оставшуюся часть функции и присоединяет ее к промису, возвращаемому `fetch()`, как если бы мы передали функцию в `Promise.then()`. В результате остаток кода ставится в очередь, а поток пользовательского интерфейса не блокируется. Приложение по-прежнему может обрабатывать остальные события и таймеры, ожидая, пока завершится выполнение `fetch()`.

Ключевое слово `await` работает только в функции, объявленной как `async`. Нельзя использовать `await` на верхнем уровне кода веб-страницы. Вместо этого необходимо создать `async`-функцию и выполнять код внутри нее, как сделано для функции `getImage()` в следующем примере:

```
async function getImage() {
  ...
}
```

Теперь `getImage()` — асинхронная функция. Она будет автоматически возвращать объект `Promise`. Код, который станет выполняться после окончания `getImage()`, прикрепляется с помощью `Promise.then()`, как в обычной цепочке промисов.

Если забыть, что `getImage()` асинхронная, то можно вызвать ее, забыв использовать промис. Это типичная ошибка, которую допускают программисты, не знакомые с `async` и `await`:

```
// Так, конечно же, делать нельзя, поскольку мы теряем объект Promise  
getImage();
```

Вместо этого нужно получить объект `Promise`, возвращаемый `getImage()`, и вызвать `then()` и `catch()`, чтобы прикрепить к промису код, который будет выполняться далее, и код обработки ошибок соответственно:

```
getImage()  
.then(response => {  
  console.log('Image download finished');  
})  
.catch(error => {  
  console.error(`Error: ${error}`);  
});
```

Вы спросите: зачем мы продолжаем использовать промисы, в то время как ключевые слова `async` и `await`, по идее, должны нас от этого избавить? Ответ состоит в том, что всегда необходимо управлять корневым объектом `Promise`, с которого начинается асинхронная операция.



Относительно недавно появилось исключение из этого правила. Применение `await` в коде верхнего уровня допускается внутри модуля (см. рецепт 8.9). В этом случае оператор с `await` обязательно должен находиться внутри блока обработки исключений `try...catch`, чтобы была возможность перехватывать необработанные ошибки.

Ключевое слово `await` становится более полезным, если нужно выполнить несколько асинхронных операций, принимая решения в процессе этого. Предположим, что нам нужно написать код, который ожидает завершения асинхронной задачи, оценивает результат, а затем выбирает, какую задачу выполнять следующей. С помощью `await` для этого можно построить код, похожий на традиционный синхронный:

```
const step1 = await someAsyncTask();  
  
if (step1 === someResult) {  
  const step2 = await differentAsyncTask();  
  ...  
}  
else {  
  const step2 = await anotherAsyncTask();  
  ...  
}
```

Посмотрев на такой простой и понятный код, вы можете спросить: почему бы не использовать `await` всегда? Подобно любым абстракциям, `await` скрывает часть деталей основного объекта `Promise`, что усложняет некоторые ситуации. Например, типичная ошибка применения `await` состоит в том, чтобы ожидать завершения ряда последовательных действий, каждое из которых выполняется

со своим `await`, в то время как на самом деле нужно выполнить все эти действия одновременно. Вот пример этой проблемы:

```
const response1 = await slowFunction(dataObject1);
const response2 = await slowFunction(dataObject2);
const response3 = await slowFunction(dataObject3);
```

Чтобы решить ее, можно было бы использовать `Promise.all()` (см. рецепт 9.4). Но в этом нет необходимости. Мы можем задействовать `await`, нужно лишь запустить все промисы заранее. Вот как следует исправить этот код:

```
const promise1 = slowFunction(dataObject1);
const promise2 = slowFunction(dataObject2);
const promise3 = slowFunction(dataObject3);

const response1 = await promise1;
const response2 = await promise2;
const response3 = await promise3;
```

Это будет работать, так как промисы начинают выполняться сразу же после их создания. К тому времени, как будут присвоены значения константам `promise1`, `promise2` и `promise3`, все три асинхронных процесса уже будут запущены. А поскольку `await` часто используется для функций, которые возвращают промис, этот код будет работать так же, как и любой объект `Promise`.

Не имеет значения, какой промис будет выполнен первым, — `await` можно безопасно применять для уже завершенных промисов. Что бы вы ни делали, этот участок кода не будет пройден до тех пор, пока не будут выполнены либо отклонены все промисы. (Технически это значит, что код работает не как `Promise.all()`, а как `Promise.allSettled()`, поскольку ожидает завершения *всех* переданных промисов, даже если какой-то из них и завершится неудачно.)

9.6. Создание асинхронной функции-генератора

Задача

Создать генератор для операции, которая бы асинхронно возвращала значения.

Решение

Использовать ключевое слово `async` и специализированную функцию-генератор, синтаксис которой описан в рецепте 6.7.

Рассмотрим следующий простейший генератор, который выдает бесконечную последовательность случайных чисел:

```
function* getRandomIntegers(max) {
  while (true) {
```

```

        yield Math.floor(Math.random() * Math.floor(max) + 1);
    }
}

```

Этот генератор вызывается так:

```

const randomGenerator = getRandomIntegers(6);

// Получаем 10 случайных значений в диапазоне от 1 до 6
for (let i=0; i<10; i++) {
    console.log(randomGenerator.next());
}

```

Для того чтобы генератор стал асинхронным, нужно просто добавить ключевое слово `async` — точно так же, как и для обычной функции:

```

async function* getRandomIntegers(max) {
    while (true) {
        yield Math.floor(Math.random() * Math.floor(max) + 1);
    }
}

```

Подобно любой асинхронной функции, асинхронный генератор возвращает не сами результаты, а объекты `Promise`, в которые обернуты результаты. Чтобы получить результат, когда он будет готов, нужно вызвать `Promise.then()`. Вот пример того, как это происходит:

```

const randomGenerator = getRandomIntegers(6);

// Получаем 10 случайных значений в диапазоне от 1 до 6
for (let i=0; i<10; i++) {
    const promise = randomGenerator.next();
    console.log('Received promise.');
```

promise.then(result => console.log(`Received result: \${result.value}`));

```

}

```

Когда вы запустите этот код, то увидите список сообщений `Received promise`, а сразу за ним — список результатов.

Синхронные генераторы часто применяются в сочетании с ключевым словом `await`. Обычно это происходит в `await`-цикле, в котором ожидается получение новых значений от генератора, пока выполняется предыдущий промис. Вот пример, в котором с помощью этой методики выполняется поиск случайных чисел, по одному числу за один раз:

```

// В этой функции используется await-цикл for
// для ожидания последовательных операций
async function searchRandomNumbers(searchNumber, generator) {
    for await (const value of generator) {
        console.log(value);
        if (value === searchNumber) return;
    }
}

```

```
// С помощью функции searchRandomNumbers() асинхронно
// генерируем случайные числа в диапазоне от 1 до 100,
// до тех пор пока не выпадет 42
const randomGenerator = getRandomIntegers(100);
searchRandomNumbers(42, randomGenerator).then(result => {
  console.log('Number found');
});
```

Как можно заметить, код, в котором задействован асинхронный итератор, в свою очередь тоже обернут в асинхронную функцию. Это сделано потому, что нельзя использовать `await` в коде верхнего уровня, как говорилось в рецепте 9.5.

Обсуждение

Функции-генераторы значительно упрощают предоставление значений по требованию. После каждого оператора `yield` JavaScript приостанавливает работу генератора. Но окружающий контекст (все локальные переменные и передаваемые аргументы) сохраняется до тех пор, пока код, вызывающий генератор, не запросит следующее значение.

На самом деле в представленном здесь примере не выполняются какие-либо асинхронные действия — случайные числа становятся доступны немедленно. Для того чтобы имитировать асинхронный процесс, можно было бы добавить в него задержку. Но гораздо интереснее рассмотреть пример, в котором показаны асинхронные генераторы, которые работают через настоящий асинхронный API.

Асинхронные генераторы приносят наибольшую пользу в тех задачах, где необходим доступ к внешнему ресурсу, отчего возникают некоторые задержки. Например, так происходит в случае веб-запросов или в API для обработки файловых потоков. Далее показан генератор, который с помощью Fetch API получает список случайных чисел от веб-сервиса:

```
async function* getRandomWebIntegers(max) {
  // Конструируем URL, чтобы получать случайные числа в заданном диапазоне
  const url = 'https://www.random.org/integers/?num=1&min=1&max=' + max +
    '&col=1&base=10&format=plain&rnd=new';

  while (true) {
    // Иницилируем запрос (и асинхронно ожидаем ответа)
    const response = await fetch(url);

    // Начинаем асинхронное чтение текста
    const text = await response.text();

    // Выдаем результат и ожидаем следующего запроса
    yield Number(text);
  }
}
```

Теперь каждый раз, когда вызывающий код будет запрашивать значение, генератор станет запускать асинхронную операцию `fetch()` и возвращать это значение. Вместе с завершением `fetch()` будет завершаться и промис. Вызывающий код сможет инициировать сразу несколько асинхронных вызовов, многократно вызывая `next()` для генератора. Но на практике для последовательного выполнения гораздо чаще используется цикл `for await`. В любом случае нет необходимости изменять код исходного решения. Если запустить эту версию примера, то мы увидим, что каждое случайное число появляется в консоли с небольшой, но заметной задержкой.

Читайте также

В рецепте 6.7 описано создание неасинхронных генераторов, а в рецепте 9.5 — создание обычных асинхронных функций.

9.7. Выполнение фоновых задач с помощью Web Worker

Задача

Выполнить код, требующий значительного времени, в отдельном потоке, чтобы он не блокировал пользовательский интерфейс.

Решение

Применить Web Worker API. Создать объект `Worker`, весь код которого будет выполняться в фоновом потоке. Объект `Worker` изолирован от остального кода (в частности, у него нет доступа к DOM, странице и глобальным переменным), однако с ним можно коммуницировать посредством обмена сообщениями.

На рис. 9.1 показан пример страницы, на которой вычисляются все простые числа в заданном диапазоне. Поскольку на ней используются Web Workers, ее интерфейс остается доступным, в то время как задача выполняется в фоновом режиме. В частности, в любой момент можно ввести текст в текстовое поле или нажать кнопку **Cancel**.

Кнопка **Start** активирует функцию `startSearch()`. Эта функция создает объект `Worker`, прикрепляет к нему функции, которые будут обрабатывать события `Worker.error` и `Worker.message`, а потом запускает выполнение, вызывая `Worker.postMessage()`. Соответствующий код скрипта на веб-странице выглядит так:

```
// Сохраняем ссылку на объект Worker, чтобы остановить его при необходимости
let worker;
```

```
function startSearch() {
    // Создаем объект Worker
    worker = new Worker('prime-worker.js');
```

```

const statusDisplay = document.getElementById('status');
statusDisplay.textContent = 'Search started.';

// В случае ошибки выводим на страницу сообщение
worker.onerror = error => {
    statusDisplay.textContent = error.message;
};

// Реагируем на сообщения от Worker и выводим на страницу
// окончательный результат (список простых чисел),
// когда он будет получен
worker.onmessage = event => {
    const primes = event.data;
    document.getElementById('primeContainer').textContent =
        primes.join(', ');
};

// Получаем диапазон поиска и запускаем Worker
const fromNumber = document.getElementById('from').value;
const toNumber = document.getElementById('to').value;
worker.postMessage({from: fromNumber, to: toNumber});
}

```

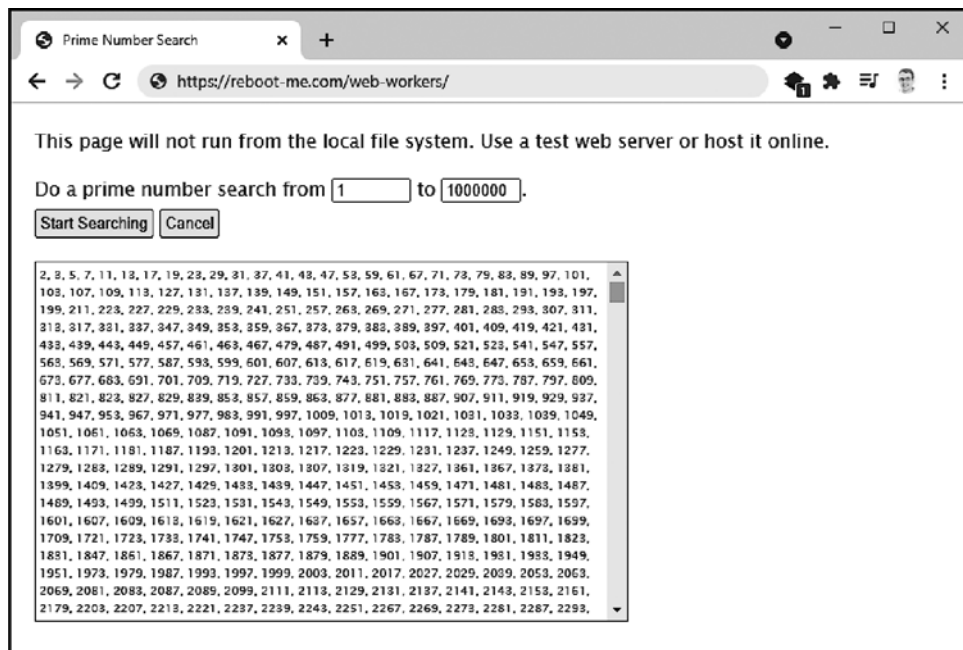


Рис. 9.1. Web Worker вычисляет простые числа

Код, выполняемый Web Worker, находится в файле `prime-worker.js`. В него входит и не показанная здесь функция `findPrimes()`, в которой реализована логика поиска простых чисел методом решета Эратосфена (<https://oreil.ly/6CyO9>). В файле `prime-worker.js` также находится обработчик события `Worker.message`, которое возникает всякий раз, когда на странице вызывается метод `Worker.postMessage()`. В данном примере функция `postMessage()` вызывается на странице, для того чтобы передать ряд чисел в `Worker` и начать поиск:

```
// С помощью этого кода Worker обрабатывает сообщения,  
// поступающие со страницы  
onmessage = (event) => {  
  // Получаем объект, переданный event.data, и вызываем  
  // ресурсоемкий метод findPrimes(), который выполняет поиск  
  const primes = findPrimes(Number(event.data.from),  
    Number(event.data.to));  
  
  // Передаем обратно результат  
  postMessage(primes);  
};
```

Остался последний ингредиент — обработчик события для кнопки `Cancel`. Она отключает Web Worker, даже если поиск еще не завершен:

```
function cancelSearch() {  
  // Отключаем Worker, при условии что ранее он был создан на странице  
  if (worker) worker.terminate();  
}
```

Обсуждение

Обычно код JavaScript выполняется в виде однопоточного приложения. В JavaScript используется система планирования, основанная на цикле событий. Она постоянно отслеживает события, контролирует такты таймера и ожидает обратных вызовов от асинхронных API. Получив функции, которые должны быть выполнены, она ставит их в очередь в порядке поступления. Если написать код с сильной нагрузкой на процессор, такой как выполнение интенсивных вычислений, то это свяжет главный поток и не позволит выполнять остальные функции, пока работа не будет завершена.



То обстоятельство, что обычный код JavaScript однопоточный, может сбить с толку. Но в JavaScript есть ряд API, таких как `Fetch`, способных работать асинхронно. Это возможно благодаря тому, что их работа обеспечивается сервисами браузера и в итоге — операционной системы. Эти API выходят за пределы среды JavaScript. Например, веб-запросы, которые делаются с помощью `fetch()`, выполняются не в главном потоке приложения, а в отдельном потоке.

Благодаря Web Worker API в JavaScript появляется возможность избежать модели однопоточного выполнения. Web Workers позволяют выполнять код конкурентно, в особом потоке, отделенном от основного пользовательского интерфейса приложения. Чтобы вам не пришлось иметь дело с такими сложными проблемами, как безопасность потоков, состояние гонки и блокировки, Web Workers размещаются в отдельном контексте выполнения. У Web Workers нет возможности взаимодействовать с веб-страницей, окном браузера или остальной частью кода. Чтобы подчеркнуть это обстоятельство, объект `Worker` требует, чтобы код Web Worker был размещен в отдельном файле, который подключается при создании `Worker`:

```
worker = new Worker('prime-worker.js');
```

После того как вы осознаете это ограничение, остальную часть модели Web Worker понять несложно. Вся коммуникация между приложением и `Worker` осуществляется посредством передачи сообщений. Для того чтобы отправить сообщение, нужно вызвать функцию `postMessage()`. В примере с поиском простых чисел страница передает литерал объекта с двумя свойствами, `to` и `from`, которые определяют диапазон поиска:

```
worker.postMessage({from: fromNumber, to: toNumber});
```

В ответ `Worker` также вызывает `postMessage()`, чтобы передать массив простых чисел:

```
postMessage(primes);
```

Количество передаваемых сообщений неограниченно. Например, можно создать объект `Worker`, вызвать `postMessage()`, чтобы передать ему какую-то работу, потом на некоторое время оставить его без дела, а затем снова вызвать `postMessage()` и передать еще кусок работы. В Web Workers также можно использовать функции `setTimeout()` и `setInterval()` для выполнения регулярных задач по расписанию.

Есть два способа остановить `Worker`. Первый — `Worker` может остановиться сам, вызвав функцию `close()`. Но чаще страница, которая создала `Worker`, закрывает его, вызывая метод `worker.terminate()`. `Worker`, остановленный таким способом, не может быть возобновлен.

Читайте также

Полный код, включая процедуру поиска случайного числа, находится в сборнике кодов к этой книге (<https://github.com/javascripteverywhere/cookbook>). Пересмотренный вариант этого примера с использованием более сложной схемы передачи сообщений находится в рецепте 9.8.

9.8. Поддержка сообщений о ходе выполнения задач в Web Worker

Задача

Сделать так, чтобы Web Worker сообщал о ходе выполнения задач.

Решение

Можно задействовать стандартную схему передачи сообщений в Worker и воспользоваться свойством объектов сообщений, чтобы различать типы сообщений.

Например, рассмотрим вариант примера с простыми числами (из рецепта 9.8), в котором передаются два типа сообщений: уведомления о ходе выполнения задач (в процессе работы) и список простых чисел (когда работа завершена). Для того чтобы сообщить приложению о разнице между этими двумя типами сообщений, добавим строковое свойство `messageType`, которое принимает значения "Progress" или "PrimestList" соответственно. Вот как выглядит переписанный код:

```
onmessage = function(event) {
    // Выполняем поиск простых чисел
    const primes = findPrimes(Number(event.data.from),
        Number(event.data.to));

    // Возвращаем результаты
    postMessage(
        {messageType: "PrimestList", data: primes}
    );
};
```

Теперь в коде, который вычисляет простое число, также нужно задействовать `postMessage()`, чтобы сообщать о ходе выполнения. Для этого используется проверка ограничения скорости, чтобы округлить данные о прогрессе до ближайшего целого процента и гарантировать, что мы не сообщаем об одном и том же состоянии дважды:

```
function findPrimes(fromNumber, toNumber) {
    // Формируем диапазон для поиска простых чисел
    ...

    // Это цикл поиска простых чисел
    for (let i = 0; i < list.length; i+=1) {

        // Проверяем, является ли данное число простым
        ...

        // Вычисляем и передаем данные о ходе выполнения
        var progress = Math.round(i/list.length*100);
```

```

    // Передаем новые данные о ходе выполнения,
    // только если они изменились хотя бы на 1 %
    if (progress !== previousProgress) {
        postMessage(
            {messageType: 'Progress', data: progress}
        );
        previousProgress = progress;
    }
}

// Очищаем и возвращаем список простых чисел
...
}

```

Когда страница получает сообщение, она проверяет свойство `messageType`, чтобы определить тип сообщения, и затем действует в соответствии с ним. Если это список простых чисел, то он выводится на странице, если же это уведомление о ходе выполнения задачи, то обновляется текст о ходе выполнения, как показано на рис. 9.2:

```

worker.onmessage = event => {
    const message = event.data;

    if (message.messageType === 'PrimeList') {
        const primes = message.data;
        document.getElementById('primeContainer').textContent =
            primes.join(', ');
    }
    else if (message.messageType === 'Progress') {
        statusDisplay.textContent = `${message.data} % done ...`;
    }
};

```

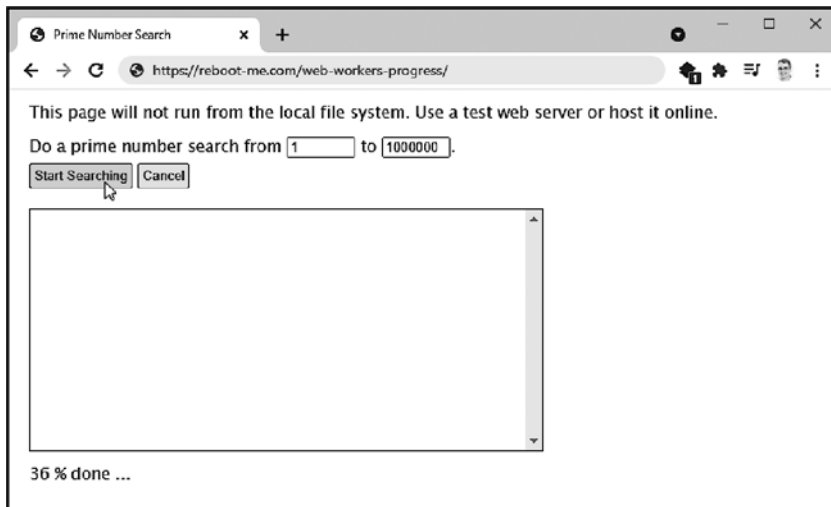


Рис. 9.2. Web Worker сообщает о ходе выполнения задачи

Обсуждение

Для того чтобы гарантировать безопасность потока, приложение и `Web Worker` могут взаимодействовать только посредством обмена сообщениями и никак иначе. В сообщении можно передать любой объект, который может быть представлен в формате JSON — примерно как при передаче сообщения на удаленный сайт.

Возможно, вы захотите создать собственный класс сообщений, чтобы формализовать используемую структуру. Но следует помнить, что объект, передаваемый между потоками, после передачи не будет отличаться от обычного объектного литерала. У него не будет собственного прототипа или методов, и вы не сможете проверить его тип с помощью `instanceof`. Аналогично, вы можете захотеть применить прием с перечисляемыми значениями из рецепта 7.13, но он не будет работать, так как приложение и `Worker` не смогут использовать одни и те же символы.

Читайте также

В JavaScript также есть два специализированных API, построенных на основе `Web Worker API`. Общие `Web Workers` (<https://oreil.ly/jGV06>) позволяют применять один и тот же `Web Worker` из разных окон. А с помощью более сложных сервисных `Web Workers` (<https://oreil.ly/vh3L3>) можно создавать `Web Workers`, которые продолжают действовать даже после того, как страница будет закрыта. Идея этого API — помочь в построении кэширования, синхронизации и сервисов отправки уведомлений, благодаря которым веб-сайт становится еще более похожим на обычное приложение.

Ошибки и тестирование

Писать код — значит делать ошибки. Многие из них можно предвидеть. В число рискованных действий входят операции, подразумевающие взаимодействие с другими ресурсами, такими как файлы, базы данных и API веб-серверов. Информация, поступающая в код извне — прочитанная с веб-страницы или полученная из другой библиотеки, — может содержать ошибки либо быть представленной в форме, отличной от той, которую вы ожидали. Но, перефразируя известную поговорку, не так важны ошибки, как то, что вы с ними будете делать потом.

Что же делать с ошибками? По умолчанию JavaScript в случае ошибки прекращает работу, записывая в консоль трассировку стека. Но есть варианты получше. Можно перехватить ошибку, отреагировать на нее, внести в нее изменения, передать ее дальше и даже при желании вообще скрыть. По сравнению со многими другими языками в JavaScript средства обработки ошибок развиты довольно слабо. Но базовая обработка ошибок все равно важна, поэтому многие рецепты главы посвящены решению этой задачи.

Защита от ошибок очень важна, однако не менее важно предотвращать их по мере возможности. Для этого существует множество фреймворков для тестирования, работающих с JavaScript, таких как Jest, Mocha, Jasmine и Karma. С помощью этих фреймворков можно писать модульные тесты, которые гарантируют, что код работает так, как должен. В этой главе мы вкратце познакомимся с фреймворком Jest.

10.1. Обнаружение и обезвреживание ошибок

Задача

Мы выполняем задачу, которая может завершиться неудачно, и не хотим, чтобы ошибка помешала дальнейшей работе кода или появилась в консоли.

Решение

Обернуть раздел кода в блок `try...catch` — например, так:

```
try {  
  // Эта строка гарантированно завершится ошибкой URIError  
  const uri = decodeURI('http%test');  
  
  // Этот код никогда не будет выполнен  
  console.log('Success!');  
}  
catch (error) {  
  console.log(error);  
}
```

Когда функция `decodeURI()` не выполнится и возникнет ошибка, программа перейдет к блоку `catch`. Он получает объект `error`, также именуемый *исключением*. Этот объект имеет следующие свойства:

- `name` — строка, обычно отражающая подтип ошибки (в данном случае `URIError`). Но это может быть и просто `Error`;
- `message` — строка, в которой содержится описание проблемы на обычном языке, например: `URI malformed` («Некорректный URI»);
- `stack` — строка, в которой перечисляются функции, открытые в данный момент в стеке, от самых последних вызовов до самых первых. В зависимости от браузера в свойстве `stack` может храниться информация о месте расположения функции (такая как номер строки и имя файла), а также аргументы, с которыми эта функция вызвана.



Будьте осторожны. У объекта `Error` есть еще несколько свойств, таких как `description` и `lineNumber`, которые работают только в определенных браузерах. Не рассчитывайте на эти нестандартные свойства при написании кода для обработки ошибок, так как они будут работать не во всех браузерах.

Если передать объект `Error` непосредственно в метод `console.log()` (как в данном примере), то получим информацию, извлеченную из всех трех свойств. Результат будет выглядеть примерно так (может немного отличаться в зависимости от конкретного браузера):

```
URIError: URI malformed  
    at decodeURI (<anonymous>)  
    at runTest (<anonymous>):14:15  
    at <anonymous>:20:1
```

Здесь в консоль выведен фрагмент кода верхнего уровня (он представлен функцией `<anonymous>` в нижней строке списка вызовов стека). Этот код вызывает функцию `runTest()`, которая затем (строкой выше) вызывает функцию `decodeURI()` с некорректным URI. Именно в `decodeURI()` и возникает ошибка, которая затем выводится в консоль.

Решение

Прежде чем тестировать код с обработкой ошибки, необходимо написать процедуру, в которой эта ошибка будет возникать. В данном примере мы не рассматриваем синтаксические или логические ошибки, которые в реальных условиях должны обнаруживаться на стадии написания кода (например, с помощью статического анализатора кода, как описано в рецепте 1.10). Поэтому сейчас речь идет о потенциально рискованных операциях, которые зависят от внешних ресурсов и из-за этого могут завершаться неудачно. Мы хотим, чтобы такие операции не нарушали выполнение кода.

JavaScript необычайно толерантен к действиям, которые во многих других языках программирования считаются ошибками. При попытке доступа к несуществующему свойству вместо сообщения об ошибке возвращается значение `undefined`, которое не считается ошибочным. То же самое происходит при выходе за границы массива. Терпимость к ошибкам в JavaScript становится особенно заметной при выполнении математических операций, когда бессмысленные вычисления, такие как умножение числа на строку, возвращают не ошибку, а допустимое значение `NaN` (not a number — «не число»), а при делении на ноль возвращается специальное значение `Infinity`. Попытка использовать функцию `decodeURI()` как раз и является примером неудачной операции, которая в данном случае возвращает `UriError`.



Методы `decodeURI()` и `encodeURIComponent()` созданы, для того чтобы заменять недопустимые в URL символы на допустимые escape-последовательности. Эта технология нужна в тех случаях, когда нужно сохранить произвольные данные в строке запроса (в той части URL, которая идет после вопросительного знака). Если исходная строка была неправильно закодирована — например, там есть лишний знак `%`, с которого должна начинаться escape-последовательность, — то попытка выполнить обратное преобразование закончится неудачей.

Благодаря перехвату ошибок мы предотвращаем появление необработанных ошибок. Это значит, что код продолжит выполняться и в случае Node ошибка не приведет к отключению всего приложения. Однако следует перехватывать только те ошибки, причины которых вы понимаете, так что эти ошибки можно обработать. Никогда не следует перехватывать ошибки, просто чтобы подавить их и проигнорировать возможные проблемы. О том, к чему могут привести перехваченные ошибки, рассказывается в рецепте 10.4.

Блок `try...catch` — наиболее распространенная структура для перехвата ошибок, но иногда в конце него ставится раздел `finally`. Код, размещенный в блоке `finally`, выполняется всегда. Если ошибок не возникло, то этот код будет выполнен после блока `try`, а в случае ошибки — после блока `catch`. Чаще всего в блоке `finally` размещается код очистки, который должен выполняться независимо от того, завершился ли предыдущий код успешно или с ошибкой:


```
try {
  const uri = decodeURI('http%test');

  // Этот код никогда не выполнится
  console.log('Success!');
}
catch (error) {
  console.log(error);
}
finally {
  console.log('The operation (and any error handling) is complete');
}
```

Читайте также

В рецепте 10.2 показано, как можно избирательно перехватывать разные типы ошибок, а в рецепте 10.3 — как перехватывать ошибки, которые возникают при выполнении асинхронных операций.

10.2. Перехват различных типов ошибок

Задача

Распознавать разные типы ошибок и обрабатывать их соответственно либо же перехватывать только ошибки определенных типов.

Решение

В отличие от многих других языков, JavaScript не позволяет перехватывать ошибки в зависимости от их типов. Вместо этого приходится перехватывать все ошибки (как обычно), а затем определять их тип с помощью оператора `instanceof`:

```
try {
  // Код, который может приводить к ошибкам
}
catch (error) {
  if (error instanceof RangeError) {
    // Выполнить это, если значение выходит за границы диапазона
  }
  else if (error instanceof TypeError) {
    // Выполнить это, если значение неправильного типа
  }
  else {
    // Передать ошибку дальше
    throw error;
  }
}
```

Наконец, если ошибка не соответствует ни одному из обрабатываемых типов, ее следует передать дальше.

Обсуждение

В JavaScript поддерживается восемь типов ошибок, представленных в виде соответствующих объектов `Error` (табл. 10.1). Чтобы понять, что случилось, нужно определить тип ошибки. Он может указать на то, какие действия следует предпринять: выполнить альтернативный код, повторить операцию или вернуться к исходному состоянию. Из типа ошибки также можно извлечь дополнительную информацию о том, что именно пошло не так.

Таблица 10.1. Объекты `Error`

Тип ошибки	Описание
<code>RangeError</code>	Возникает, когда числовое значение выходит за пределы допустимого диапазона
<code>ReferenceError</code>	Возникает при попытке присвоить переменной несуществующий объект
<code>SyntaxError</code>	Возникает при наличии в коде явной синтаксической ошибки, такой как лишняя скобка или отсутствие фигурной скобки
<code>TypeError</code>	Возникает, если значение имеет недопустимый тип для данной операции
<code>URIError</code>	Возникает при проблемах экранирования URL в <code>decodeURI()</code> и подобных функциях
<code>AggregateError</code>	Обертка для нескольких ошибок. Удобна для ошибок, возникающих при асинхронных операциях. Массив объектов <code>Error</code> размещается в свойстве <code>errors</code>
<code>EvalError</code>	Изначально этот тип предназначался для описания проблем, возникающих при использовании встроенной функции <code>eval()</code> , но в настоящее время он не применяется. Сейчас <code>eval()</code> или синтаксически некорректный код приводят к появлению ошибок <code>SyntaxError</code>
<code>InternalError</code>	Возникает в различных нестандартных ситуациях и зависит от конкретного браузера. Например, в Firefox <code>InternalError</code> возникает в случае превышения глубины рекурсии (когда функция снова и снова вызывает сама себя), а в Chrome в той же ситуации появляется ошибка <code>RangeError</code>

В дополнение к этим типам ошибок можно создавать, а затем выбрасывать и перехватывать собственные объекты `Error`, как описано в рецепте 10.6.

В JavaScript у каждого блока `try` может быть только один блок `catch`, что не позволяет перехватывать ошибки в зависимости от их типа. Но мы можем перехватить стандартный объект `Error`, определить его тип с помощью `instanceof` и написать код для обработки ошибок в соответствии с их типом. Главное при использовании этой методики — сделать так, чтобы случайно не подавлять ошибки, которые вы не можете обработать.

В данном примере код явно обрабатывает ошибки `RangeError` и `TypeError`. Если ошибка относится к другому типу, то предполагается, что мы ничего не можем сделать, чтобы решить эту проблему. Такие ошибки передаются дальше с помощью оператора `throw`. В ветви `throw` ошибка как бы возникает снова. Если код находится внутри функции, то ошибка будет подниматься по стеку вызовов до тех пор, пока не попадет на код обработки ошибок, способный ее корректно обработать. Если кода для перехвата таких ошибок нет, то получим необработанную ошибку — такую же, как если бы мы ее не перехватили в самом начале. (Подробнее об этом читайте в рецепте 10.4.)

Другими словами, передача неизвестных ошибок дальше по стеку вызовов обеспечивает то же поведение программы, каким оно было бы, если бы мы перехватывали только ошибки определенных типов. Именно такой подход мы бы использовали, если бы JavaScript позволял это делать.

Читайте также

В рецепте 10.6 показано, как создать собственный класс ошибок, который бы описывал нестандартные условия появления ошибки и передавал дополнительную информацию о ней.

10.3. Перехват асинхронных ошибок

Задача

Перехватывать ошибки, возникающие при выполнении операции в фоновом потоке.

Решение

В разных API JavaScript есть несколько моделей асинхронного выполнения операций, поэтому способ обработки ошибок зависит от применяемой функции.

При использовании более старых API может потребоваться создать функцию обратного вызова, которая будет вызываться при возникновении ошибки, или прикрепить обработчик событий. В объекте `XMLHttpRequest` есть событие `error`, позволяющее сообщать о неудачных запросах, например:

```
const request = new XMLHttpRequest();

request.onerror = function errorHandler(error) {
  console.log(error);
}

request.open('GET', 'http://noserver');
request.send();
```

Здесь вызов функции `send()` запускает асинхронную операцию, которая приводит к ошибке. Но она возникает в отдельном потоке, поэтому, если поместить оператор внутрь блока `try...catch`, проблема не решится. Лучшее, что здесь можно сделать, — это получить уведомление, которое генерируется при событии `error`.

При использовании API на базе промисов можно подключить функцию обработки ошибок с помощью `Promise.catch()`. Вот пример для Fetch API:

```
fetch('http://noserver')
  .then((response) => {
    console.log('We did it, fam.');
```

```
  })
  .catch((error) => {
    console.log(error);
  });
```

Написанный нами код будет активироваться в случае необработанной ошибки или отклоненного промиса. Если не перехватить ошибку, возникшую в промисе, то она будет передана вверх по стеку вызовов до потока главного приложения и приведет к событию `window.unhandledrejection` — эквиваленту события `window.error` для промисов (см. рецепт 10.4).

В целом, если использовать промисы в сочетании с высокоуровневой моделью `async/await`, то можно задействовать обычный блок обработки ошибок. Раздел `catch` будет автоматически подключен к промису посредством `Promise.catch()`, например:

```
async function doWork() {
  try {
    const response = await fetch('http://noserver');
  }
  catch (error) {
    console.log(error);
  }
}

doWork().then(() => {
  console.log('All done');
});
```

Обсуждение

Распространенная ошибка состоит в размещении кода обработки ошибок не там, где нужно. К сожалению, не всегда удастся заметить, что код обработки ошибок неэффективен или вообще никогда не работает, хотя статический анализатор кода (см. рецепт 1.10) может предупредить о проблеме. Наилучшим решением будет протестировать реальные условия возникновения ошибки в приложении и убедиться, что код обработки ошибок работает и перехватывает ошибки.

Читайте также

В рецепте 9.2 показан полный пример обработки ошибок в промисах для Fetch API. В рецепте 9.5 представлен полный пример для обработки ошибок в Fetch API с использованием ключевых слов `async` и `await`.

10.4. Обнаружение необработанных ошибок

Задача

Перехватывать ошибки, которые не были обработаны в коде, — возможно, для записи в журнал диагностики.

Решение

Перехватывать событие `window.error`. Функция обработки ошибок принимает пять параметров с информацией об ошибке. Кроме объекта `error`, который соответствует возникшей ошибке, передаются также отдельный параметр `message` и информация о месте возникновения ошибки (параметр `source` с URL файла с кодом, параметр `lineno` с номером строки, в которой находится ошибка, и параметр `colno` с номером столбца).

Вот пример, в котором проверяется это событие:

```
// Прикрепляем обработчик ошибок
window.onerror = (message, url, lineno, columnNo, error) => {
  console.log(`An unhandled error occurred in ${url}`);
}

// Вызываем необработанную ошибку
console.log(null.length);
```

Обратите внимание: для того чтобы протестировать это пример, понадобится тестовая страница. Мы не можем прикрепить функцию к обработчику ошибок `window.error` через консоль разработчика.



В некоторых случаях политика безопасности браузеров относительно информации разного происхождения закрывает доступ к подробной информации об ошибках для кода JavaScript. Например, так происходит, если страница находится не на тестовом сервере, а в локальной файловой системе. В таких ситуациях параметр `message` содержит общий текст `Script error`, а свойства `url`, `lineno`, `columnNo` и `error` пусты. Подробнее об этом читайте в разделе документации о событии `onerror` (<https://oreil.ly/9MbGP>).

Обсуждение

Необработанные ошибки, возникающие в главном потоке приложения, поднимаются вверх по стеку вызовов до тех пор, пока не достигнут верхнего уровня

кода, и если они и там не будут обработаны, то вызовут в браузере событие `window.error`.

Событие `window.error` необычно в том смысле, что позволяет отменить ошибку, просто подавив ее. Для этого нужно, чтобы функция обработки события вернула `true`. Если не подавить ошибку, то в дело включится стандартный обработчик ошибок браузера. Он выведет информацию об ошибке в консоль разработчика, выделив сообщение красным цветом, — точно так же, как если бы мы вывели это сообщение с помощью метода `console.error()`. Но если вернуть `true` из обработчика `window.error`, то ошибка исчезнет и сообщение о ней не появится в консоли. За исключением этого, нет никаких практических различий между тем, чтобы подавить ошибку или позволить ей достичь обработчика `window.error`. К тому времени, как ошибка активирует событие `window.error`, код уже будет остановлен, а стек очищен. Однако веб-страница при этом продолжит функционировать. Если возникнет другое событие (например, вы нажмете кнопку), то JavaScript продолжит выполнение кода.



Современные подходы не рекомендуют скрывать ошибки даже из консоли разработчика, если только на то нет очень веских причин. Одной из них может быть замена стандартного сообщения об ошибке чем-то другим, более соответствующим конкретному приложению, где вы хотите предоставить пользователям больше полезной информации.

В обработчике событий `window.error` можно выполнять любой код JavaScript. Например, можно сохранить сообщение об ошибке на локальном носителе данных или даже переслать сообщение на веб-сервер посредством Fetch API. Если в *процессе* обработки события `window.error` тоже возникнет ошибка, то обработчик ошибки не будет вызван снова. Он просто передаст ошибку непосредственно в стандартный обработчик ошибок браузера, и она появится в консоли.

В случае асинхронного кода ошибки обрабатываются иначе. В старых API, построенных на функциях обратного вызова, ошибки обычно не генерируются. В более новых API условия ошибки передаются в основной код посредством обратных вызовов (см. рецепт 10.3). В API на основе промисов необработанные ошибки поднимаются вверх по стеку вызовов и активируют событие `window.unhandledrejection`:

```
// Прикрепляем обработчик ошибок
window.onunhandledrejection = (e) => {
  console.log(e.reason);
}

// Создаем промис, который приводит к необрабатываемой асинхронной ошибке
const faultyPromise = new Promise(() => {
  throw new Error('Disaster strikes!');
});

// Создаем промис, который отклоняет ошибку
```

```
// (и активирует событие window.onunhandledrejection)
const rejectedPromise = new Promise((resolve, reject) => {
  reject(new Error('Another disaster strikes!'));
});
```

Событие `unhandledrejection` передает в обработчик событий один параметр — объект со свойствами события. В свойстве `reason` (использовано в предыдущем примере) хранится объект необработанной ошибки, а если промис был отклонен вручную, — то любой другой объект, переданный в `Promise.reject()`. Доступ к основному объекту `Promise` также можно получить через свойство `promise`.

Событие `window.unhandledrejection`, как и `window.error`, отменяемое. Но в `window.unhandledrejection` задействуется другое, более современное условие отмены. Вместо того чтобы возвращать `true`, можно воспользоваться методом объекта `preventDefault()`, передав ему аргументы события. Вот пример, где в случае необработанной ошибки промиса выводится сообщение, но не выполняется автоматическая запись в консоль:

```
window.onunhandledrejection = (e) => {
  console.log('An error occurred, but we won\'t tell you what it was');

  // Отменяем обработку ошибок по умолчанию
  e.preventDefault();
}
```



Может показаться, что события необработанных исключений — подходящий момент для вывода информации в журнал. Иногда это действительно так. Но обычно лучше перехватить ошибку как можно ближе к тому месту, где она произошла, сразу внести запись в журнал и при необходимости передать ошибку дальше. Однако события необработанных ошибок всегда хороши для поиска рискованных участков кода, которые нуждаются в логике обработки исключений и в которых эта логика почему-либо отсутствует.

Дополнительно: средства ведения журнала

Вообще говоря, существуют два основных момента для обнаружения ошибок: при тестировании кода, когда их можно исправить, и во время эксплуатации приложения, когда мы хотим понять, что именно пошло не так. В первом случае нашей целью является обнаружение и исправление проблемы. И тогда запись в журнал зачастую ограничивается обычным вызовом `console.log()`. Во втором случае необходимо исследовать проблему, которая может возникать нерегулярно, в определенном окружении и перед конечным пользователем. Здесь нужен способ распознать проблему и сообщить разработчику подробные сведения о ней.

Мы могли бы перехватывать события `window.error` и `window.unhandledrejection`, а затем записывать детали на какой-либо носитель. Например, можно сохранить информацию об ошибке в объекте `localStorage`, где она будет храниться дольше, чем существует текущая сессия браузера. Можно применить функцию `fetch()`,

чтобы передать информацию об ошибке на сервер посредством веб-API. В случае Node-приложения можно сохранить данные на сервере — в файле или в базе данных. К ним можно добавить дополнительную информацию о контексте, такую как системные данные, уровень приоритета и метка времени. Но по мере того как система записи в журнал будет разрастаться, вы, вероятно, пожелаете использовать инструменты ведения журнала с открытым кодом, вместо того чтобы развивать собственное решение.

Хороший инструмент ведения журнала обеспечивает *уровень абстракции* поверх журнала. Это значит, что при записи в журнал, которая делается примерно так же, как при обычном вызове `console.log()`, вы не должны знать, где этот журнал находится или каким образом он построен. При тестировании достаточно, чтобы уровень журнала просто выводил сообщения в консоль. Но после того как приложение будет развернуто, уровень журнала должен полностью игнорировать низкоуровневые сообщения и передавать только самые важные из них куда-то еще — например, на удаленный веб-сервер. Инструмент ведения журнала должен поддерживать такие расширенные функции, как пакетная передача. Благодаря этому повышается производительность, так как несколько сообщений могут передаваться на удаленный сайт ускоренно, в виде одной последовательности.

Есть великое множество библиотек для ведения журнала в приложениях JavaScript — Winston, Bunyan, Log4js, Loglevel, Debug, Pino и многие другие. Некоторые из них разрабатывались специально для приложений Node, но многие другие могут работать и с обычным кодом веб-страниц в браузере.

10.5. Выдача обычной ошибки

Задача

Сообщить о состоянии ошибки, выбросив объект `Error`.

Решение

Создать экземпляр объекта `Error`, передав в конструктор краткое описание проблемы — оно будет сохранено в свойстве `message`. Выбросить объект `Error` посредством оператора `throw`. Затем другой код перехватит этот объект `Error` так же, как перехватываются другие типы ошибок в JavaScript:

```
function strictDivision(number, divisor) {
  if (divisor == 0) {
    throw new Error('Dividing by zero is not allowed');
  }
  else {
    return number/divisor;
  }
}
```



```
// Перехватываем ошибку
try {
  const result = strictDivision(42, 0);
}
catch (error) {
  // Выводим собственное сообщение об ошибке
  console.log(`Error: ${error.message}`);
}
```

Обсуждение

Есть два способа создать объект `Error`: с помощью ключевого слова `new`, как показано ранее, или (что встречается реже) путем вызова `Error()` как функции. Результат получается один и тот же:

```
// Стандартная выдача ошибки
throw new Error(`Dividing by zero is not allowed`);

// Другой вариант, с тем же результатом
throw Error(`Dividing by zero is not allowed`);
```

У объекта `Error` есть стандартные свойства ошибки, в том числе `message`, которое мы задаем, `name`, которому присваивается бесполезное значение `Error`, и `stack` — трассировка стека, которая указывает место, где произошла ошибка.



В коде JavaScript также можно использовать оператор `throw` для объектов, не являющихся ошибками (например, для строк). Это нестандартное применение, и оно может приводить к проблемам в коде обработки исключений, где ожидаются такие свойства, как `name` и `message`. Как показывает практика, не стоит выбрасывать вместо исключений объекты, которые не являются исключениями.

Иногда вместо стандартной ошибки можно выбросить ошибку более конкретного подтипа. Большинство встроенных типов ошибок JavaScript (перечисленных в табл. 10.1) предназначены для специальных случаев и не подходят для разработанного вами кода. Однако среди них есть несколько потенциально полезных. Если функция получает числовое значение, которое выходит за пределы приемлемого диапазона, то можно использовать ошибку `RangeError`. Главное — не забыть включить в нее информативное сообщение об ошибке с указанием полученного значения и ожидаемого диапазона:

```
function setAge(age) {
  const upper = 125;
  const lower = 18;
  if (age > 125 || age < 18) {
    throw new RangeError(
      `Age [${age}] is out of the acceptable range of ${lower} to ${upper}.`);
  }
}
```

Ошибки `RangeError` предназначены специально для числовых значений. Если полученное значение не соответствует заданному типу, то можно использовать ошибку `TypeError`. В чем именно состоит неправильность типа, решать вам: это может быть строка там, где ожидается число (что проверяется с помощью `typeof`), или неправильный тип объекта (проверяется с помощью `instanceof`):

```
function calculateValue(num) {  
  if (typeof num !== 'number') {  
    throw new TypeError(`Value [${num}] is not a number.`);  
  }  
}
```

К менее полезным типам ошибок, которые, однако, тоже можно использовать, относятся `ReferenceError` (выдается при получении ссылки со значением `null` или `undefined` там, где ожидался объект) и `SyntaxError` (выдается, например, при анализе чего-то, представленного в виде строки, формат которой не соответствует установленным правилам). Чтобы передать более подробную информацию об условиях ошибки, можно создать собственный класс ошибок (рецепт 10.6).

По сравнению со многими более строгими языками JavaScript использует ошибки весьма умеренно. При разработке библиотек обычно придерживаются следующего соглашения: не применяют исключения для тех случаев, которые не вызывают ошибок в обычном JavaScript, например при неявном преобразовании типов. Не стоит с помощью ошибок уведомлять вызывающий код о неожиданных ситуациях — другими словами, о ситуациях, которые вполне могут случиться в ходе обычной работы приложения, таких как ввод пользователем некорректной информации. Задействуйте исключения для того, чтобы предотвратить выполнение операций, которые закончатся неудачей из-за неверно заданных начальных условий.

Читайте также

В рецепте 10.6 показано, как создать нестандартный объект ошибки.

10.6. Выдача нестандартных ошибок

Задача

Указать на специфические условия ошибки, выбросив собственный объект ошибки.

Решение

Создать класс, унаследованный от стандартного класса `Error`. Конструктор этого класса должен принимать текст с описанием ошибки, который будет

присваиваться свойству `message`, и использовать `super()`, чтобы вызывать конструктор базового класса `Error` с этим сообщением. Вот как выглядят минимальная нестандартная ошибка и код, который ее перехватывает:

```
class CustomError extends Error {
  constructor(message) {
    super(message);
    this.name = 'CustomError';

    // Необязательное улучшение: очищаем
    // трассировку стека, если это возможно
    if (Error.captureStackTrace) {
      Error.captureStackTrace(this, CustomError);
    }
  }
}

// Пытаемся выбросить нашу ошибку
throw new CustomError('An application-specific problem occurred');
```

Еще одно необязательное, но рекомендуемое улучшение: с помощью метода `Error.captureStackTrace()` можно немного почистить трассировку стека. (Технически `captureStackTrace()` гарантирует, что вызов конструктора ошибки не появится в трассировке стека, которая сохраняется в свойстве `Error.stack`.)

В новый объект можно добавить и другие свойства, чтобы передавать дополнительную информацию об условиях возникновения ошибки. В следующем примере сохраняется `productID` после неудачного поиска товара:

```
class ProductNotFound extends Error {
  constructor(missingProductID) {
    super(`Product ${missingProductID} does not exist in the catalog`);

    this.name = 'ProductNotFound';
    this.productID = missingProductID;

    if (Error.captureStackTrace) {
      Error.captureStackTrace(this, ProductNotFound);
    }
  }
}

try {
  throw new ProductNotFound(420);
} catch (error) {
  console.log(`An error occured with the message: ${error.message}`);

  if (error instanceof ProductNotFound) {
    console.log(`Missing: ${error.productID}`);
  }
}
```

Обсуждение

При создании нестандартных классов `Error` необходимо учитывать две задачи, которые могут противоречить друг другу: остаться в пределах стандартной ошибки JavaScript и донести достаточно информации о нестандартных условиях ошибки. Возвращаясь к предыдущему примеру: не стоит пытаться восстановить ошибки и исключения из вашего второго любимого языка программирования. Не следует перегружать тип `Error` в JavaScript излишними методами и дополнительным функционалом.

При создании нестандартной ошибки учитывайте следующие соглашения.

- Выберите такое имя класса, которое бы указывало на тип ошибки, и присвойте соответствующее значение свойству `name`. Это важно, так как для определения типа ошибки любой код будет проверять значение `name`, вместо того чтобы использовать `instanceof`. Это значение сохранится и в том случае, если объект ошибки будет сериализован в формат JSON, значение `name` появится и в строковом представлении ошибки при выводе в консоль разработчика.
- В конструкторе дополнительные свойства должны идти первыми в списке параметров. Параметр `message`, если он используется, должен стоять последним.
- В конструкторе обязательно должен быть вызов `super()` с передачей свойства `message` в конструктор базового класса.
- Еще одна тонкость — правильное значение для трассировки стека. Проверьте, доступен ли метод `captureStackTrace()`, и если да, вызовите его, передавая туда ссылку на текущий экземпляр (`this`) и созданный вами класс ошибок.

Читайте также

Подробнее о наследовании и ключевом слове `extends` читайте в рецепте 8.8.

10.7. Написание модульных тестов для кода

Задача

Написать автоматизированные тесты, которые бы гарантировали, что код соответствует критериям разработки сейчас и будет им соответствовать в дальнейшем.

Решение

Воспользоваться инструментом для написания модульных тестов, таким как Jest, и написать тесты на как можно более ранней стадии разработки кода.

Проще всего установить Jest с помощью `npm` (см. рецепт 1.7). Откройте окно терминала в папке проекта и создайте файл конфигурации `package.json`, если у вас его еще нет, с помощью команды `npm init`:

```
$ npm init -y
```

Затем установите Jest, используя параметр `--save-dev`, чтобы этот инструмент-рий подключался к сборкам только на этапе разработки:

```
$ npm install --save-dev jest
```

Теперь нам нужно найти код для тестирования. Предположим, у нас есть файл с именем `factorialize.js`, а в нем — следующая функция `factorialize()`:

```
function factorialize(number) {
  if (number < 0) {
    throw new RangeError('Factorials are only defined for positive numbers');
  }
  else if (number !== Math.trunc(number)) {
    throw new RangeError('Factorials are only defined for integers');
  }
  else {
    if (number == 0 || number == 1) {
      return 1;
    }
    else {
      let result = number;
      while (number > 1) {
        number--;
        result *= number;
      }
      return result;
    }
  }
}
```

Для того чтобы она стала доступной для Jest, нужно экспортировать функцию `factorialize()`, добавив в конец файла следующую строку:

```
export {factorialize}
```



Jest предполагает, что мы используем стандарт модулей Node (CommonJS). Если вы уже перешли на более новый стандарт ES6, то необходимо задействовать Babel — инструмент компиляции JavaScript, который преобразует ссылки в вашем модуле, перед тем как Jest начнет обрабатывать код. Это может показаться сложным, но модуль `plugin-transform-modules-commonjs` сделает за вас почти всю работу. Чтобы увидеть полностью сконфигурированный результат для обоих стандартов (с модулями CommonJS и ES6), просмотрите исходный код. Подробнее о модулях CommonJS читайте в рецепте 18.2, а о модулях ES6 — в рецепте 8.9.

Теперь нужно создать тестовый файл. В Jest тестовым файлам присваивается расширение `.test.js`. Соответственно, в данном случае нужно создать файл

с именем `factorialize.test.js`. Затем мы импортируем в него функцию, которую хотим протестировать:

```
import {factorialize} from './factorialize.js';
```

Остальную часть тестового файла займут тесты, которые мы хотим выполнять. Самый простой способ тестирования состоит в том, чтобы вначале убедиться: функция работает так, как ожидается. Например, можно написать Jest-тест, проверяющий, что `factorialize()` возвращает корректную информацию для нескольких репрезентативных случаев. Вот пример проверки того, что `10!` равно `3 628 800`:

```
test('10! is 3628800', () => {
  expect(factorialize(10)).toBe(3628800);
});
```

Функция `test()` из инструментария Jest создает именованный тест. Имена позволяют различать тесты в отчетах о тестировании, благодаря чему вы всегда будете точно знать, какие тесты пройдены успешно, а какие нет. В данном примере в тесте использована функция Jest `expect()`, которая вызывает тестируемый код (в данном случае функцию `factorialize()`) и оценивает результат с помощью функции `toBe()`. Технически `toBe()` — это одна из нескольких *функций соответствия* Jest. Они определяют, проходит код данный тест или нет.

Для того чтобы выполнить тест, необходимо задействовать Jest. Это можно сделать из командной строки, указав файл с тестом и воспользовавшись `prx` — средством запуска `prn`-пакетов. В данном случае команда в терминале должна выглядеть так:

```
$ npx jest factorialize.test.js
```

Она запускает единственный написанный нами тест и генерирует такой отчет:

```
PASS ./factorialize.test.js
  ✓ 10! is 3628800 (4 ms)

Test Suites:   1 passed, 1 total
Tests:         1 passed, 1 total
Snapshots:     0 total
Time:          2.725 s, estimated 3 s
```

```
Ran all test suites matching /factorialize.test.js
```

Но чаще Jest добавляют в раздел `scripts` файла `package.json`, благодаря чему все тесты выполняются автоматически:

```
{
  "scripts": {
    "test": "jest"
  }
}
```

Теперь можно дать Jest команду выполнить все тесты (все файлы с расширением `.test.js`), размещенные в папке проекта.

Обсуждение

Существует несколько типов тестов, в том числе тесты безопасности, производительности, удобства использования. Но простейшей формой тестирования являются *модульные тесты*. Модульное тестирование заключается в выполнении тестов для дискретных модулей исходного кода с проверкой того, что они ведут себя в соответствии с ожиданиями. В JavaScript наиболее распространенным модулем для тестирования является функция.

Несмотря на то что доступных фреймворков тестирования множество (Jest, Mocha, Jasmine, Karma и др.), большинство из них имеет сходный синтаксис. В Jest все вращается вокруг функции `test()`, которая принимает два аргумента. Первый аргумент — это заголовок теста, который появляется в отчете о тестировании, а второй — функция, включающая в себя одно или несколько тестовых утверждений. Эти утверждения будут либо успешно подтверждены (тест пройден), либо опровергнуты (тест не пройден):

```
test('Some test name', () => {  
  // Здесь проверяются тестовые утверждения  
});
```

Для создания тестовых утверждений применяется функция `expect()`, которая является центральным стержнем Jest. В сочетании с функциями соответствия, такими как `toBe()`, функция `expect()` позволяет оценить результаты тестирования:

```
test('10! is 3628800', () => {  
  expect(factorialize(10)).toBe(3628800);  
});
```

В этом примере показан единичный тест функции `factorialize()`. Но цели написания тестов гораздо шире. Нам необходимо создать репрезентативную группу тестов, которые проверяли бы различные значения, по возможности захватывая граничные условия. Например, при тестировании функции `factorialize()` имеет смысл проверить, как она ведет себя, если подать на вход нецифровые значения, отрицательные числа, ноль, очень большие числа и т. п. Следующий код представляет собой более полный набор тестов. Он проверяет результаты пяти вызовов `factorialize()` с разными входными значениями. Эти вызовы сгруппированы в один тестовый набор с помощью функции `describe()`. Она просто назначает общий заголовок для набора связанных тестовых вызовов. В данном примере `describe()` группирует вызовы одной и той же функции, однако с помощью нее можно группировать вызовы и по принципу использования одного и того же набора входных данных:

```
describe('factorialize() function tests', () => {  
  test('0! is 1', () => {
```

```

    expect(factorialize(0)).toBe(1);
  });
  test('1! is 1', () => {
    expect(factorialize(1)).toBe(1);
  });
  test('10! is 3628800', () => {
    expect(factorialize(10)).toBe(3628800);
  });
  test('"5"! is 120', () => {
    expect(factorialize('5')).toBe(120);
  });
  test('NaN is 0', () => {
    expect(factorialize(NaN)).toBe(0);
  });
});

```

При выполнении этих тестов мы обнаружим, что последний из них не проходит. В нем ожидается, что вызов `factorialize(NaN)` вернет 0, однако функция выбрасывает ошибку, что становится ясно из журнала тестирования:

```

FAIL ./factorialize.test.js
  factorialize() function tests
    ✓ 0! is 1 (3 ms)
    ✓ 1! is 1
    ✓ 10! is 3628800
    ✓ "5"! is 120
    ✗ NaN is 0 (3 ms)

    • factorialize() function tests › NaN is 0

      RangeError: Factorials are only defined for integers

      4 |   }
      5 |   if (number !== Math.trunc(number)) {
>  6 |     throw new RangeError('Factorials are only defined for integers');
        |           ^
      7 |   }
      8 |   else {
      9 |     if (number == 0 || number == 1) {

      at factorialize (factorialize.js:6:11)
      at Object.<anonymous> (factorialize.test.js:17:12)

Test Suites: 1 failed, 1 total
Tests:       1 failed, 4 passed, 5 total
Snapshots:   0 total
Time:        2.833 s
Ran all test suites.

```

Пока что во всех тестах, которые вам встречались, использовалась только функция соответствия `toBe()`, проверяющая точное значение. Однако в Jest, как и в остальных фреймворках тестирования, есть разные типы правил. Например, можно проверить, попадает ли значение в определенный диапазон, соответствует

ли текст заданному шаблону или является ли значение ненулевым. В табл. 10.2 приведены некоторые из самых полезных функций соответствия, которые можно применять в `expect()`. Полный список таких функций вы найдете в документации Jest для метода `expect()` (<https://oreil.ly/hnbiy>).

Таблица 10.2. Функции соответствия в Jest

Функция	Описание
<code>arrayContaining()</code>	Ищет в массиве заданное значение
<code>not()</code>	Инвертирует условие. Например, тест <code>expect(...).not.toBe(5)</code> будет пройден, если значение не равно 5
<code>stringContaining()</code>	Ищет подстроку в строке
<code>stringMatching()</code>	Проверяет соответствие строки регулярному выражению
<code>toBe()</code>	Проверяет значение на стандартное равенство JavaScript, как при использовании оператора <code>==</code>
<code>toBeCloseTo()</code>	Проверяет два числа на равенство или почти равенство. Функция предназначена, для того чтобы игнорировать мелкие ошибки округления при сравнении чисел с плавающей точкой (эта проблема подробно описана в рецепте 3.4)
<code>toBeGreaterThan()</code>	Проверяет, оказывается ли числовое значение больше заданного. Есть небольшая группа подобных функций соответствия для различных сравнений, в том числе <code>toBeGreaterThanOrEqual()</code> , <code>toBeLessThan()</code> и <code>toBeLessThanOrEqual()</code>
<code>toBeInstanceOf()</code>	Проверяет, является ли возвращаемый объект экземпляром заданного класса — так же, как при использовании оператора <code>instanceof</code>
<code>toBeNull()</code>	Проверяет, равно ли значение <code>null</code> . Аналогично можно проверить значение на равенство <code>NaN</code> с помощью функции <code>toBeNaN()</code> и на равенство <code>undefined</code> с помощью <code>toBeUndefined()</code>
<code>toBeTruthy()</code>	Проверяет, соответствует ли число истине, то есть будет ли оно приравнено к <code>true</code> в операторе <code>if</code> . В JavaScript истине соответствует любое значение, кроме <code>null</code> , <code>undefined</code> , пустой строки, <code>NaN</code> , <code>0</code> и <code>false</code>
<code>toEqual()</code>	Выполняет глубокое сравнение, при котором сравнивается содержимое двух объектов — в отличие от <code>toBe()</code> , где в случае объектов сравниваются ссылки. Как правило, <code>toBe()</code> применяется к примитивным типам, а <code>toEqual()</code> — к экземплярам объектов
<code>toHaveProperty()</code>	Проверяет, есть ли у возвращаемого объекта определенное свойство и (возможно) равно ли оно заданному значению

Таблица 10.2 (окончание)

Функция	Описание
<code>toStrictEqual()</code>	Делает то же самое, что и <code>toEqual()</code> , но проверяет объекты на точное равенство. Например, объекты с одинаковыми свойствами и их значениями не будут считаться равными, если они являются экземплярами разных классов или же один из них — экземпляр класса, а второй — объектный литерал
<code>toThrow()</code>	Проверяет, выбрасывает ли функция исключение. Также можно потребовать, чтобы это исключение было определенным объектом <code>Error</code>

Для того чтобы исправить предыдущий пример, нужно показать, что мы ожидаем, что в случае значения `NaN` будет выбрасываться исключение, а для этого следует воспользоваться функцией `toThrow()`. Однако для применения `toThrow()` нужно выполнить дополнительную операцию — обернуть код, который находится внутри `expect()`, в *еще одну* анонимную функцию. Иначе исключение не будет перехвачено и тест не будет пройден. В итоге код должен выглядеть так:

```
test('NaN causes error', () => {  
  expect(() => {  
    factorialize(NaN);  
  }).toThrow();  
});
```

Читайте также

Этот пример представляет собой хороший обзор основных возможностей Jest. Но есть множество дополнительных функций, на которые стоит обратить внимание. Например, в Jest есть дополнительные возможности для использования фиктивных данных, обработки асинхронных результатов, полученных от промисов, эмуляции таймеров и тестирования копий экрана, при котором проверяется, что интерфейс страницы не изменился. Подробнее обо всех этих возможностях читайте в документации Jest (<https://oreil.ly/aeu1l>).

Дополнительно: сначала пишем тесты

В современной разработке принято сначала писать тесты и только потом — большую часть функционала приложения (и библиотек). Такая разработка через тестирование (test-driven development, TDD) является компонентом парадигмы гибкой разработки Agile.

К TDD нужно привыкнуть. Вместо более формального структурного программирования или «водопадного» проектирования, при котором тестирование откладывается, до тех пор пока не будет получен более или менее готовый код,

TDD требует написания тестов прежде, чем будет создано что-либо еще. Вот как это происходит.

1. *Определяем тесты.* Например, если бы мы хотели написать функцию `factorialize()`, которая была показана в предыдущем примере, то нам следовало бы начать с определения репрезентативного набора тестов, который покрывал бы все возможные варианты входных данных — в частности, максимальное число, факториал которого можно получить, граничные значения вроде 0 и возможные краевые условия (такие как строка, неявно преобразованная в число, или значение `BigInt`). Также можно написать тесты для проверки того, что функция корректно обрабатывает ошибки — в данном случае выбрасывает соответствующее исключение.
2. *Делаем так, чтобы тесты не выполнялись.* После того как тесты написаны, создаем код. Некоторые специалисты по TDD требуют сделать так, чтобы на первом этапе скомпилированный код провалил тесты. Добившись этого, мы убедимся, что тесты работают, тестовые условия являются осмысленными и мы не сможем случайно передать в эксплуатацию неготовый код.
3. *Делаем так, чтобы тесты выполнялись.* Этот этап можно описать как «сделать так, чтобы код проходил тесты всеми возможными способами». Другими словами, мы не стараемся создать наилучшее из возможных решений, а лишь стремимся сделать так, чтобы код проходил все тесты. Не следует писать больше кода, чем требуется для выполнения условий тестов.
4. *Выполняем рефакторинг.* После того как все тесты успешно проходятся, можно поработать над улучшением кода. Теперь можно выполнить рефакторинг, удалить дублирующий код, внести улучшения и снова повторить все тесты, чтобы убедиться, что они по-прежнему выполняются. Возможно, при этом обнаружатся случаи, еще не покрытые тестами, и придется написать дополнительные тесты.

Одно из очевидных преимуществ TDD состоит в том, что этот метод заставляет сосредоточиться на текущей задаче. Вы не обязаны интерпретировать требования к структуре приложения, чтобы определить, как они должны быть реализованы в коде. Вместо этого достаточно просто закодировать точные спецификации, формально представленные в виде тестов. Кроме того, TDD-разработка упрощает развитие приложения, так как сокращает риски, возникающие при изменениях. До тех пор пока код проходит заранее заданные тесты и эти тесты репрезентативны (действительно ли это так — большой вопрос), можно спокойно размещать новые версии в базе кода.

Ценой такой защиты является необходимость создания соответствующих тестов. Это занимает гораздо больше времени, чем просто написание кода, а чтобы сделать все правильно, нужен большой опыт. Один из параметров, помогающих оценить схему тестирования, называется *покрытием кода тестами* (рецепт 10.8).

10.8. Отслеживание покрытия кода тестами

Задача

Понять, насколько хорошо тестовые случаи охватывают все функциональные возможности кода.

Решение

С помощью инструмента тестирования получить отчет о покрытии кода тестами. В Jest для этого используется параметр `--collect-coverage`:

```
$ npx jest --collect-coverage
```

Теперь Jest будет выполнять все тесты для всех файлов `test.js` (как обычно), но в конце отчета будет размещаться более подробный анализ покрытия кода тестами. Вот как выглядит отчет с тестами для функции `factorialize()` из рецепта 10.7:

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	82.61	66.67	100	82.61	
factorialize.js	82.61	66.67	100	82.61	3-4,6-7

Test Suites: 1 passed, 1 total
 Tests: 4 passed, 4 total
 Snapshots: 0 total
 Time: 2.741 s

Обсуждение

Для того чтобы определить, насколько хорошо код покрывается тестами, необходим многосторонний подход. Чтобы ответить на этот вопрос, необходимо использовать такие методики, как проверка кода и его пошаговый анализ совместно с коллегами. Однако все инструменты тестирования включают в себя средства автоматизированного анализа кода, помогающие приблизительно определить, насколько успешно тесты оценивают данный код.

В Jest этот анализ активируется посредством параметра `--collect-coverage`. Он указывается в командной строке либо в команде `jest` в конфигурационном файле `package.json` для вашего приложения.

Отчет о покрытии кода помогает понять, насколько полно тот тестируется. Для этого используются несколько параметров, измеряемых в процентах, которые указаны в отдельных столбцах.

- **Funcs** — здесь показано, сколько функций было протестировано. Это хорошая отправная точка для определения качества покрытия тестами, но это

и наименее детализированная статистика. При тестировании `factorialize()` все функции покрыты тестами. Но это не означает, что выполняется весь код этих функций!

- **Stmts** — показывает, сколько процентов операторов в коде выполняется в процессе тестирования. При тестировании `factorialize()` 83 % написанного кода покрывается хотя бы одним тестом.
- **Branch** — показывает, сколько различных ветвей (созданных условной логикой, такой как операторы `if`) выполняются при тестировании. Тесты `factorialize()` проходят по 67 % отдельных условных ветвей.

Кроме того, в отчете о выполнении кода могут указываться строки, не покрытые тестами. Например, в случае с `factorialize()` это строки 3 и 4 в файле с исходным кодом, где отбрасываются отрицательные числа, и строки 6 и 7, где отбрасываются нецелые числа. Для того чтобы улучшить тестовое покрытие кода, необходимо написать тестовое утверждение, в котором бы с помощью функции `toThrow()` проверялось, что оба эти варианта корректно отбрасываются.

Отчет, полученный с помощью командной строки, дает краткий обзор тестового покрытия кода. Jest позволяет также генерировать более полный отчет в формате HTML, который сохраняется в папке *покрытия*. Список всех протестированных файлов с более подробными основными статистическими показателями находится в файле `index.html` (рис. 10.1). Например, вместо общих значений в процентах вы найдете в этом отчете точное число операторов, ветвей и функций. Если щелкнуть на любом файле в списке, то вы попадете на другую страницу, где показан код, с одной особенностью: не покрытые тестами операторы выделены, так что их легко заметить (рис. 10.2).

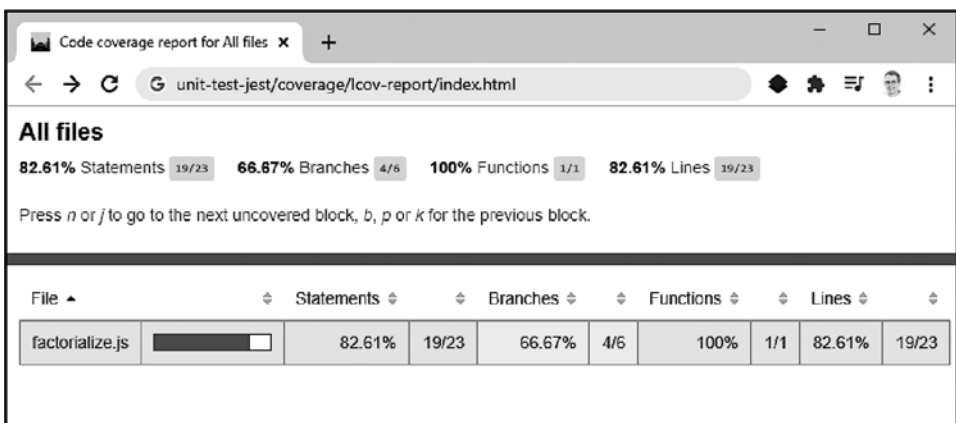


Рис. 10.1. Отчет о покрытии кода тестами

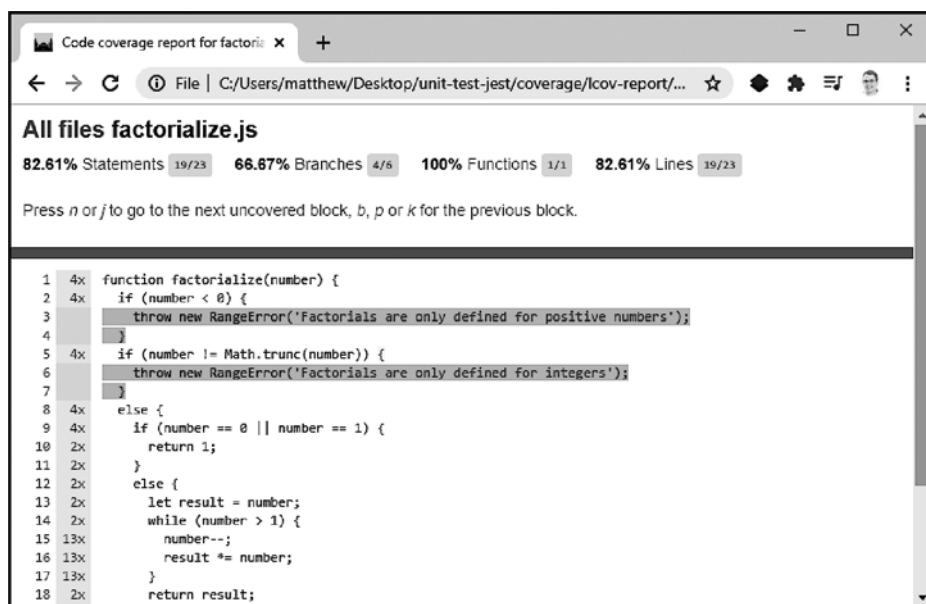


Рис. 10.2. Выделение строк кода, не покрытых тестами



Сейчас много спорят о том, какова цель правильного покрытия кода тестами. Одни разработчики считают, что покрытие должно быть как можно ближе к 100 %, в то время как другие настаивают, что более реалистична цифра 70–80 % и она обеспечивает наилучший возврат инвестиций, вложенных в написание тестов. Однако, честно говоря, покрытие тестами не является определяющим показателем. Дело не только в том, что процентная доля покрытия зависит от способа измерения (функции, операторы, ветви), — у средств тестирования нет способа определить наиболее рискованные или уязвимые участки кодовой базы.

Часть II

JavaScript в браузере

Инструментарий браузера

С точки зрения веб-разработчика браузер — это окно, через которое внешний мир видит труд программиста. Браузер также предоставляет ему полезный инструментарий для разработки и тестирования сайтов. Поэтому имеет смысл уделить некоторое время изучению того, как использовать имеющиеся в браузере средства разработки, чтобы затем было лучше и легче отлаживать код. В этой главе мы рассмотрим несколько полезных средств для отладки, профилирования и анализа кода JavaScript.

Для простоты во всех примерах будет задействоваться Developer Tools (DevTools) из Google Chrome. На момент написания этой книги Chrome применяли более 65 % всех пользователей (<https://oreil.ly/QFZD9>). В большинстве остальных браузеров есть аналогичный функционал. Так, отличная альтернатива с полезными средствами разработки — Mozilla Firefox Developer Edition (<https://oreil.ly/IJSel>).

11.1. Отладка кода JavaScript

Задача

Узнать значение переменной в определенный момент выполнения кода JavaScript.

Решение

Проинспектировать значения в коде посредством контрольных точек. Если создать контрольную точку, то отладчик будет останавливать в ней выполнение кода и выводить все текущие значения из данной области видимости. Затем можно будет выполнить дальнейший код пошагово либо позволить JavaScript завершить выполнение. На рис. 11.1 показана копия экрана с кодом, выполнение которого было остановлено в контрольной точке.

Для того чтобы в Chrome Developer Tools создать контрольную точку в определенной строке кода JavaScript, сделайте следующее.

1. Откройте Chrome Developer Tools с помощью клавиш **Command-Option-C** (Macintosh) или **Control+Shift+C** (Windows или Linux).

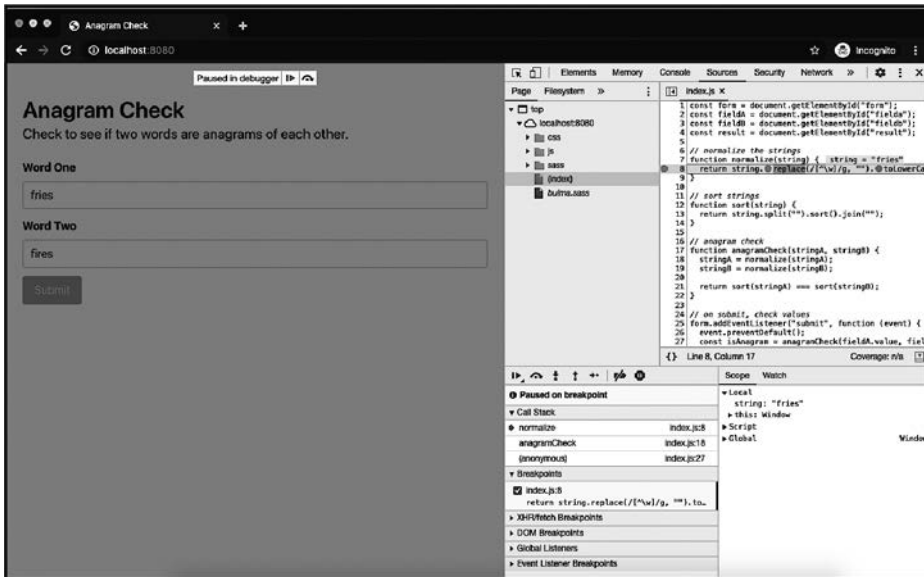


Рис. 11.1. Копия экрана с отладчиком Chrome, в котором выполнение кода остановлено в контрольной точке

2. Перейдите на вкладку DevTools Sources.
3. Выберите из списка файлов нужный файл с кодом JavaScript.
4. Щелкните на номере строки, в которой хотите создать контрольную точку.
5. Выполните код, либо реализовав нужные действия на странице, либо обновив окно браузера.

Обсуждение

Обычно для того, чтобы узнать нужные значения в определенных точках кода, используют функцию `console.log()`. Однако контрольные точки обеспечивают большую гибкость и позволяют получить больше информации. Когда вы лучше освоите этот способ отладки, вам будет гораздо проще устранять неполадки в коде JavaScript, написанном для браузеров.

Контрольные точки можно создавать не только в интерфейсе браузера, но и в коде, добавляя туда оператор `debugger`. В этом случае выполнение кода будет останавливаться на операторе `debugger`:

```
function normalize(string) {
  const normalized = string.replace(/^[^w]/g, "").toLowerCase();
  debugger;
  return normalized;
}
```

При достижении контрольной точки возможны несколько вариантов дальнейшего выполнения кода JavaScript.

- *Возобновить выполнение кода.* Выполнить код далее до конца.
- *Переступить через функцию.* Выполнить функцию, не заходя внутрь нее для отладки.
- *Войти в функцию.* Войти в функцию, чтобы отладить ее изнутри.
- *Выйти из функции.* Выполнить оставшуюся часть кода функции и выйти из нее.
- *Сделать шаг.* Перейти к следующей строке кода.

Эти контрольные точки, устанавливаемые в строках кода, — лишь один из возможных вариантов контрольных точек. Также можно устанавливать контрольные точки в зависимости от изменений DOM, значений условий, обработчиков ошибок, исключений, а также запросов `fetch/XHR`. Благодаря контрольным точкам удастся лучше контролировать выполнение кода JavaScript при отладке.

11.2. Анализ производительности во время выполнения кода

Задача

Вам кажется, что код JavaScript выполняется медленно или с ошибками, но вы не можете с уверенностью назвать причину проблемы.

Решение

С помощью инструмента браузера под названием Performance analysis поискать в коде узкие места и задачи, создающие значительную нагрузку на процессор (рис. 11.2).

Для того чтобы исследовать производительность кода JavaScript с помощью Chrome Developer Tools, сделайте следующее.

1. Откройте Chrome Developer Tools с помощью клавиш **Command-Option-C** (Macintosh) или **Control+Shift+C** (Windows или Linux).
2. Перейдите на вкладку DevTools Performance.
3. Нажмите кнопку **Record** для взаимодействия со страницей или **Reload** — для того чтобы проследить за параметрами производительности при загрузке новой страницы.

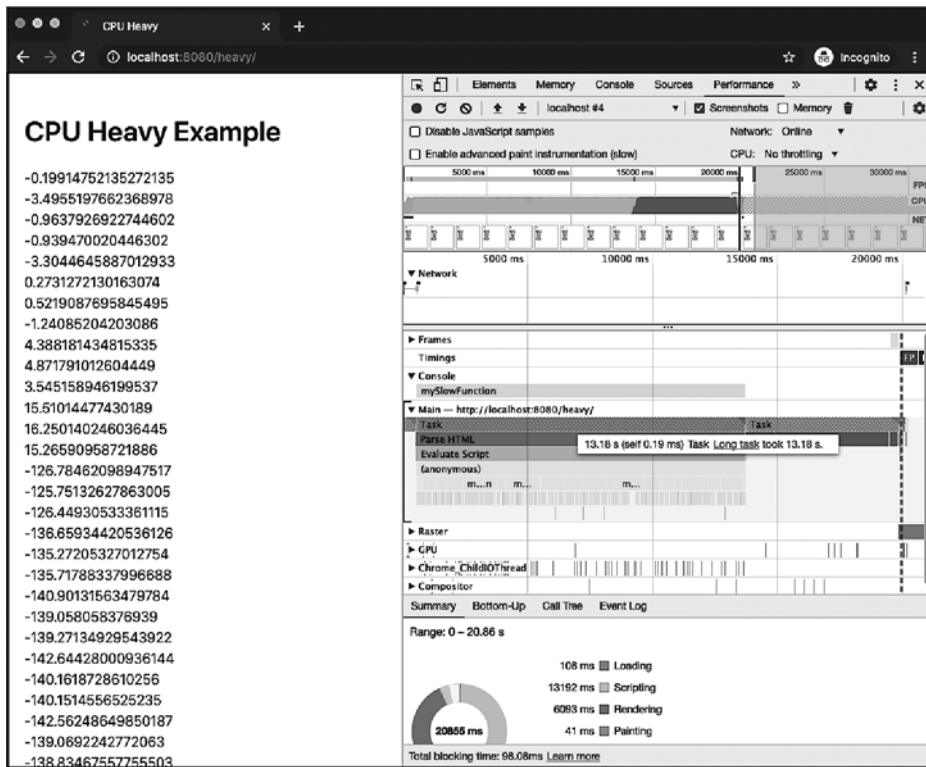


Рис. 11.2. Вкладка Performance в Chrome

Когда Chrome закончит профилирование страницы, вам будет представлена информация, позволяющая отследить потенциальные узкие места по производительности.

Обсуждение

Инструмент Chrome Performance разбивает процесс отображения страницы в браузере на части и представляет их в виде последовательности на шкале времени, с копиями экрана и сводной диаграммой (рис. 11.3). С помощью этой информации можно найти места в коде, которые снижают производительность.

У вас, как разработчика, скорее всего, высокопроизводительный компьютер и скоростное подключение к интернету. Один из самых полезных инструментов отслеживания производительности в браузере — это возможность имитировать слабый процессор или медленное соединение с Сетью. Это позволяет обнаружить проблемы с производительностью, с которыми могут столкнуться пользователи, хотя для вас они неочевидны.

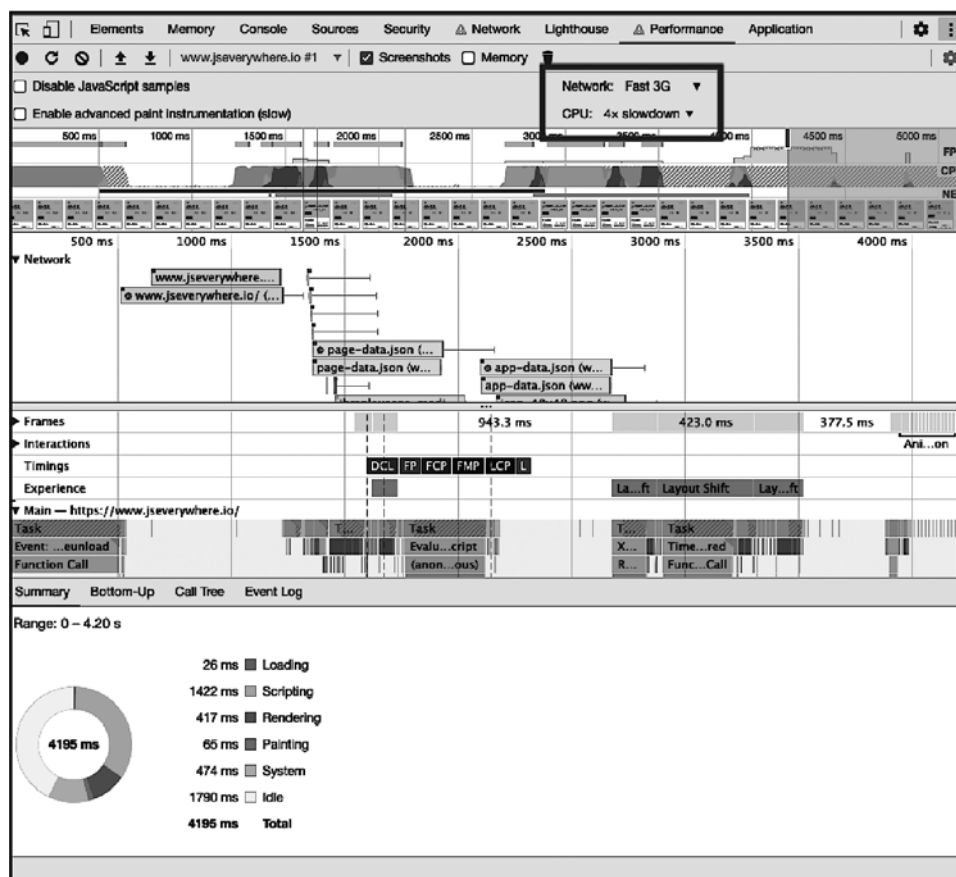


Рис. 11.3. Инструменты Chrome Developer Performance позволяют имитировать слабый процессор и медленное соединение с интернетом

Исследование данных о производительности — важный этап, позволяющий гарантировать положительные впечатления пользователей. Доказано, что хорошая производительность сайта повышает коэффициент удержания пользователей и конверсию продаж. В рецепте 11.4 мы покажем, как еще можно проанализировать возможные проблемы с производительностью.

11.3. Обнаружение неиспользуемого кода JavaScript

Задача

Производительность приложения снижена из-за большого размера файлов JavaScript.

Решение

Найти неиспользуемые фрагменты кода JavaScript с помощью инструмента Chrome Developer Tool Coverage (рис. 11.4).

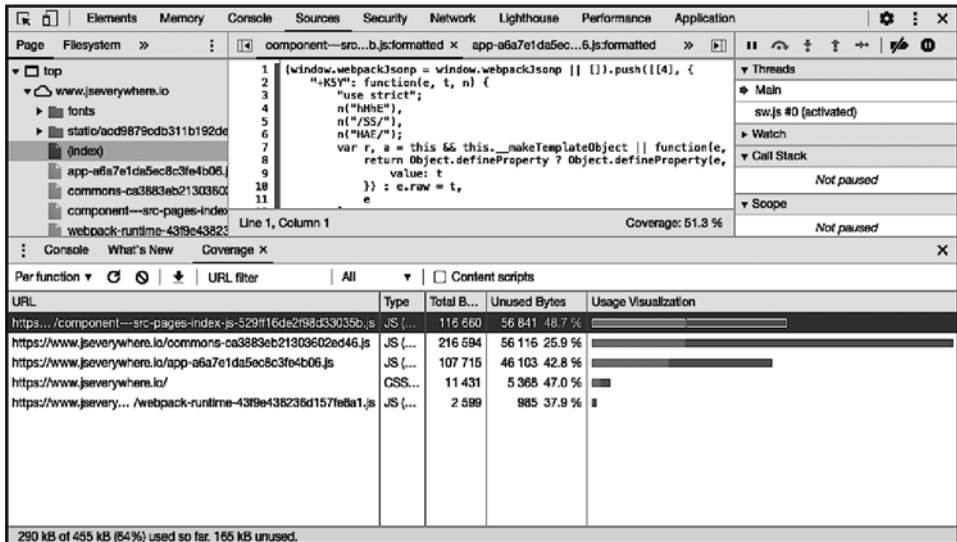


Рис. 11.4. Инструмент Chrome Coverage

Чтобы увидеть, какие участки кода JavaScript не используются, нужно перейти на вкладку Coverage. Для этого выполните следующие действия.

1. Откройте Chrome Developer Tools с помощью клавиш Command-Option-C (Macintosh) или Control+Shift+C (Windows или Linux).
2. Откройте Command Menu с помощью клавиш Command-Shift-P (Macintosh) или Control+Shift+P (Windows или Linux) и введите команду coverage.
3. Выберите Show Coverage и нажмите Enter.
4. Нажмите кнопку Record для взаимодействия со страницей или Reload — для того чтобы проследить за параметрами производительности при загрузке новой страницы.
5. Когда захотите остановить запись результатов, нажмите Stop Instrumenting Coverage And Show Results.

Результаты будут представлены в виде отчета, который содержит следующую информацию:

- URL файла;
- тип файла;

- размер файла в байтах;
- количество неиспользуемых байтов;
- визуализация применения файла.

Эту информацию можно в дальнейшем задействовать при рефакторинге кода, чтобы сократить общее количество неиспользуемых байтов на странице.

Обсуждение

Исследовать степень используемости кода полезно — это позволяет определить, сколько процентов кода, который вы передаете пользователям, никогда не будут задействованы. Такой код часто сокращают вручную при рефакторинге. Но можно использовать и такие инструменты связывания кода JavaScript, как Webpack, чтобы разделить код на несколько пакетов и автоматически устранить «мертвый» код путем «встряски» дерева вызовов функций. Эти методы подробно описаны в рецепте 16.2.

11.4. Выдача наилучших рекомендаций посредством Lighthouse

Задача

Определить, насколько ваше веб-приложение соответствует наилучшим рекомендациям.

Решение

Использовать инструмент Google Lighthouse, встроенный в Chrome Developer Tools (рис. 11.5). Для этого сделайте следующее.

1. Откройте Chrome Developer Tools нажатием **Command-Option-C** (Macintosh) или **Control+Shift+C** (Windows или Linux).
2. Перейдите на вкладку **DevTools Lighthouse**.
3. Выберите категории, которые вы хотели бы профилировать, и тип устройства (мобильное или настольное).
4. Нажмите кнопку **Generate Report**.

Lighthouse сформирует отчет с оценками по каждой категории и рекомендуемыми улучшениями.

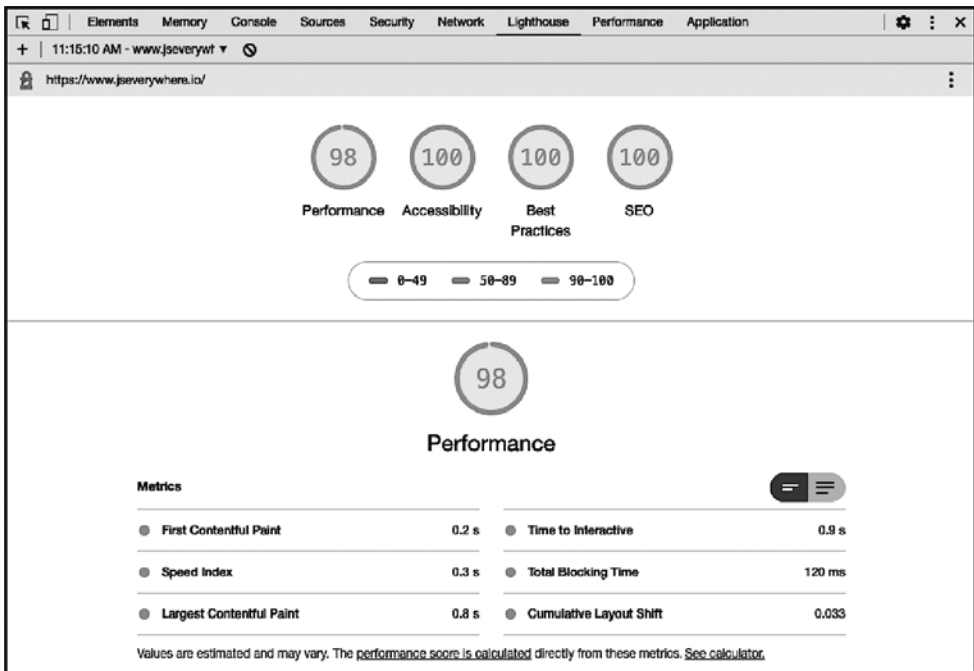


Рис. 11.5. Отчет с результатами работы Google Lighthouse в Chrome Developer Tools

Обсуждение

Lighthouse — это инструмент с открытым кодом, созданный Google для измерения производительности и выдачи наилучших рекомендаций при создании веб-сайта. Он встроен в Chrome Developer Tools, но может работать и как самостоятельное расширение браузера, модуль Node или утилита командной строки. Отчеты Lighthouse могут формироваться на случай просмотра приложения на настольном компьютере или на мобильном устройстве, что позволяет быстро получить представление о производительности на мобильном устройстве. Lighthouse генерирует отчеты и рекомендации в следующих областях:

- производительность;
- прогрессивные веб-приложения;
- наилучшие рекомендации;
- доступность для лиц с ограниченными физическими возможностями;
- SEO.

Итоговый отчет предоставляет эффективную обратную связь с описанием конкретных проблем, ссылками на документацию и списком рекомендуемых улучшений. На рис. 11.6 показаны рекомендации по повышению производительности для профилируемого веб-сайта, включая удаление неиспользуемого кода JavaScript и уменьшение влияния стороннего кода. Если развернуть каждый из этих диагностических пунктов, то увидим дополнительные сведения и ссылки на файлы.

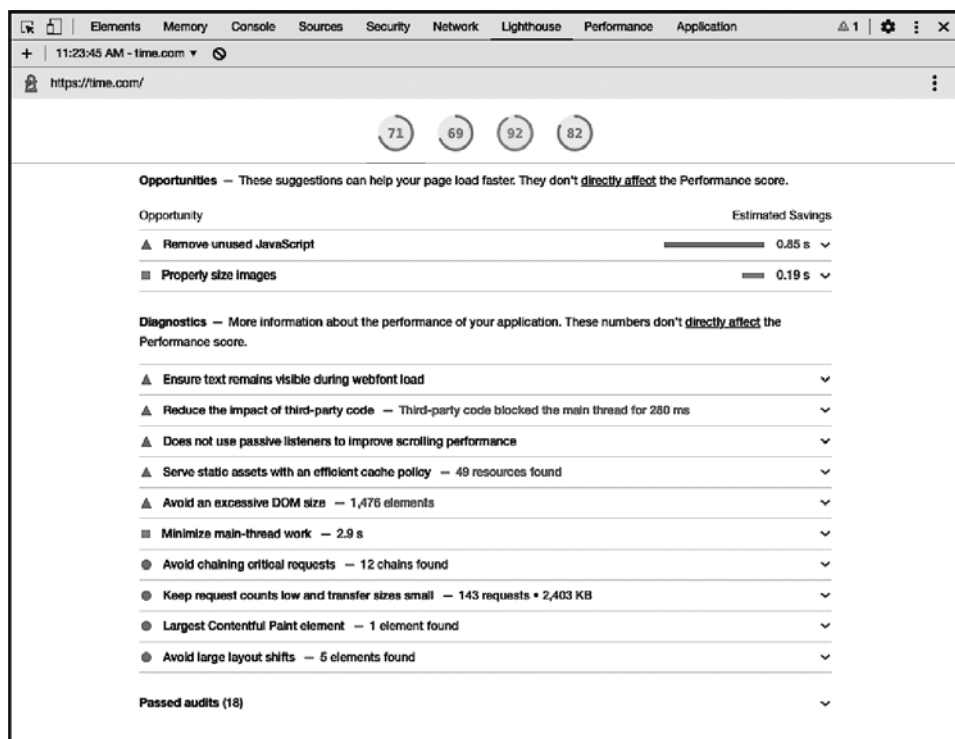


Рис. 11.6. Рекомендации по производительности Lighthouse

Google Lighthouse — полезный инструмент для оценки общего состояния и производительности веб-сайтов и приложений, над которыми вы работаете. Доступ к Lighthouse из средств разработки браузера обеспечивает быстрый и простой способ профилирования сайта в ходе его создания. Кроме средств разработки в пользовательском интерфейсе есть еще утилита командной строки с открытым кодом (<https://github.com/GoogleChrome/lighthouse>) и модуль Node, благодаря которым можно встраивать отчеты Lighthouse в процесс непрерывной интеграции и доставки приложений.

Работа с HTML

В 1995 году компания Netscape поручила разработчику программного обеспечения, компании Brendan Eich, создать язык программирования, который позволил бы строить интерактивные веб-страницы в браузере Netscape Navigator. В результате Eich за десять дней создала печально известную первую версию JavaScript. Через несколько лет JavaScript благодаря стандартизации ECMAScript стал кросс-браузерным стандартом.

Несмотря на то что этот язык пытались стандартизировать почти сразу же после его появления, веб-разработчики еще долгие годы сражались с различными интерпретациями движка и отдельных функций JavaScript в разных браузерах. В итоге появились популярные библиотеки, такие как jQuery, которые позволили писать простой кросс-браузерный код на JavaScript. К счастью, во всех современных браузерах используется почти одна и та же универсальная реализация JavaScript, что позволяет веб-программистам писать код для взаимодействия с HTML-страницами на чистом JavaScript (без подключения библиотек).

Имея дело с HTML, мы взаимодействуем с объектной моделью документа (Document Object Model, DOM) — именно в ней представлены данные на HTML-странице. В рецептах, приведенных в этой главе, будет показано, как взаимодействовать с DOM HTML-страницы, выбирая, изменяя и удаляя отдельные элементы страницы.

12.1. Доступ к определенному элементу, поиск его родительского и дочерних элементов

Задача

Получить доступ к определенному элементу веб-страницы, найти его родительский и дочерние элементы.

Решение

Присвоить элементу уникальный идентификатор:

```
<div id="demodiv">
  <p>
    This is text.
  </p>
</div>
```

Затем с помощью метода `document.getElementById()` получить ссылку на этот элемент:

```
const demodiv = document.getElementById("demodiv");
```

Родительский узел находится в свойстве `parentNode`:

```
const parent = demodiv.parentNode;
```

А дочерние узлы — в свойстве `childNodes`:

```
const children = demodiv.childNodes;
```

Обсуждение

Структура веб-документа представляет собой перевернутое дерево, верхний элемент которого является корневым, а все остальные отходят от него вниз, как ветви. У каждого элемента этого дерева, за исключением корневого (HTML), есть родительский узел, все элементы дерева доступны через элемент `document`.

Есть несколько способов доступа к элементам веб-документа, или *узлам*, как они называются в DOM. В настоящее время доступ к узлам осуществляется посредством стандартизованных версий DOM, таких как DOM Level 2 и DOM Level 3. Однако изначально доступ к элементам страницы де-факто происходил через объектную модель браузера, которую иногда называют DOM Level 0. Модель DOM Level 0 была изобретена компанией Netscape, которая в то время была лидером по производству браузеров. С тех пор она в большей или меньшей степени поддерживается большинством существующих браузеров. Главным объектом для доступа к элементам веб-страницы в DOM Level 0 является объект `document`.

Из всех методов DOM наиболее часто используется `document.getElementById()`. Он принимает один параметр — строку с идентификатором элемента (с учетом регистра). Этот метод возвращает объект `element`, соответствующий элементу документа, если таковой существует, и `null`, если такого элемента нет.



Есть множество способов получить определенный элемент веб-страницы, в том числе с помощью селекторов, описанных в этой главе. Но всегда желательно использовать наиболее ограничивающий из возможных методов, а самый ограничивающий из них — это `document.getElementById()`.

У возвращаемого объекта `element` есть несколько методов и свойств, часть из которых унаследованы от объекта `node`. Методы `node` предназначены в основном для обхода дерева документа. В частности, чтобы найти родительский узел элемента, нужно сделать следующее:

```
const parent = document.getElementById("demodiv").parentNode;
```

Тип элемента, соответствующего узлу, находится в свойстве `nodeName`:

```
const type = parent.nodeName;
```

Для того чтобы узнать, какие дочерние элементы есть у данного элемента, нужно перебрать коллекцию элементов `NodeList`, полученную с помощью свойства `childNodes`:

```
let outputString = '';

if (demodiv.hasChildNodes()) {
  const children = demodiv.childNodes;
  children.forEach(child => {
    outputString += `has child ${child.nodeName} `;
  });
}
console.log(outputString);
```

Для элемента из нашего примера будет выведено следующее:

```
"has child #text has child P has child #text "
```

Возможно, вас удивит перечень дочерних узлов. Здесь пробелы перед элементом абзаца и после него тоже являются дочерними узлами, для которых значение `nodeName` равно `#text`. В следующем примере:

```
<div id="demodiv" class="demo">
  <p>Some text</p>
  <p>Some more text</p>
</div>
```

у элемента (узла) `demodiv` не два, а пять дочерних узлов:

```
has child #text
has child P
has child #text
has child P
has child #text
```

Наилучший способ убедиться в том, насколько запутанным может быть DOM, — открыть страницу в отладчике, который входит в инструментарий разработчика Firefox и Chrome, а затем задействовать одно из средств просмотра DOM, которые предлагает отладчик. На рис. 12.1 показана простая страница, я открыл ее в Chrome и воспользовался средствами разработки, чтобы продемонстрировать дерево элементов.

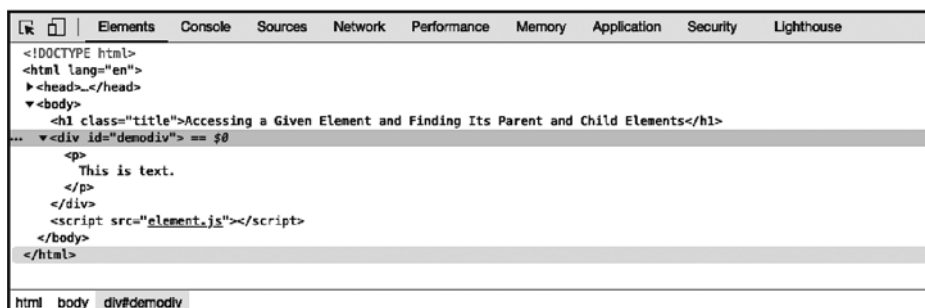


Рис. 12.1. Исследование дерева элементов веб-страницы с помощью Chrome Developer Tools

12.2. Перебор результатов, полученных от `querySelectorAll()`, с помощью `forEach()`

Задача

Перебрать в цикле содержимое объекта `nodeList`, полученного при вызове `querySelectorAll()`.

Решение

В современных браузерах к объекту `NodeList` — коллекции, возвращаемой после вызова `querySelectorAll()`, — можно применять метод `forEach()`:

```
// С помощью querySelectorAll находим все элементы списка на странице
const items = document.querySelectorAll('li');

items.forEach(item => {
  console.log(item.firstChild.data);
});
```

Обсуждение

Метод `forEach()` предназначен для массивов, но `querySelectorAll()` возвращает не массив, а объект `NodeList`. К счастью, в современных браузерах есть встроенная поддержка `forEach`, позволяющая перебирать содержимое объекта `NodeList` так, как будто это массив. Но, к сожалению, в Internet Explorer (IE) такой поддержки `forEach` нет. Если вы хотите, чтобы код работал в IE, необходимо использовать полизаполнение (polyfill), в основе которого лежит стандартный цикл:

```
if (window.NodeList && !NodeList.prototype.forEach) {
  NodeList.prototype.forEach = function(callback, thisArg) {
    thisArg = thisArg || window;
    for (var i = 0; i < this.length; i++) {
```

```
        callback.call(thisArg, this[i], i, this);
    }
};
}
```

В полизаполнении мы проверяем, существует ли метод `NodeList.prototype.forEach`, и если нет, то добавляем в прототип `NodeList` метод `forEach`, в котором результаты запроса к DOM перебираются в цикле. После этого можно без проблем использовать синтаксис `forEach` в любом месте кода.

12.3. Привязка к элементу действия в ответ на щелчок

Задача

Сделать так, чтобы при щелчке на кнопке, ссылке или другом элементе веб-страницы выполнялась определенная функция JavaScript.

Решение

Привязать к элементу обработчик события `click`:

```
// Создаем функцию – обработчик события
const clickHandler = (event) => {
    window.alert('The element has been clicked!');
};

// Выбираем элемент
const btn = document.getElementById('click-button');
// Привязываем к элементу обработчик события и вызов функции clickHandler
btn.addEventListener('click', clickHandler);
```

Обсуждение

Метод `addEventListener()` позволяет коду JavaScript перехватывать определенный тип событий и определять функцию, которая должна вызываться при их возникновении. В предыдущем примере я привязал к кнопке обработчик события `click`. Теперь при нажатии кнопки будет вызываться функция `clickHandler`, которая выводит предупреждение.

По умолчанию обработчики щелчков должны привязываться к кнопкам (элементам `button`), так как это наиболее очевидные элементы, которые должны реагировать на события `click`. Если того требует дизайн приложения, элемент `button` может быть оформлен при помощи стилей таким образом, чтобы выглядеть как ссылка. Вместо кнопки можно использовать элемент, стандартное поведение которого является переходом по ссылке, но тогда нужно сделать так, чтобы по умолчанию JavaScript не загружал страницу. Для того чтобы

переопределить поведение ссылки по умолчанию, применяется метод события `preventDefault`:

```
const clickHandler = (event) => {
  event.preventDefault();
  window.alert(`The ${event.currentTarget.nodeName} element has been
  clicked!`);
};

const href = document.getElementById('click-link');
href.addEventListener('click', clickHandler);
```



В обычных функциях JavaScript ключевое слово `this` указывает на элемент, на котором щелкнули клавишей мыши. Но при использовании нового синтаксиса стрелочных функций JavaScript, как в данном примере, значение `this` наследуется от родительской функции, которой по умолчанию является `window`. Если вы привыкли к традиционному синтаксису без стрелок, это может сбить с толку. Если хотите ближе познакомиться с этой темой, советую изучить статью Джо Кардилло (<https://oreil.ly/wK7Ik>).

Иногда возникает необходимость сделать так, чтобы на щелчок реагировал обычный блок, такой как элемент `div`. Я бы рекомендовал делать это как можно реже, по возможности с помощью элементов `button`. Но если все же придется так поступить, то необходимо гарантировать, что эта функциональность будет доступна при использовании экранных дикторов и навигации с клавиатуры. Поэтому прежде всего нужно указать в разметке свойство `role` со значением `button` и значение `tabindex`. Свойство `role` будет сообщать экранным дикторам, что данный элемент реагирует на нажатия, а благодаря `tabindex` этот элемент станет доступным при управлении с клавиатуры:

```
<div tabindex="0" role="button" id="click-div">Click me</div>
```

В данном случае мы используем обработчик события `keydown`. Благодаря этому элемент будет доступен при управлении с клавиатуры:

```
const clickHandler = (event) => {
  window.alert(`The ${event.currentTarget.nodeName} element has been clicked!`);
};

const clickableDiv = document.getElementById('click-link');
clickableDiv.addEventListener('click', clickHandler);

// При использовании элемента div добавляем
// обработчик события keydown, чтобы элемент был
// доступен при управлении с клавиатуры
clickableDiv.addEventListener('keydown', (event) => {
  if (event.code === 'Space' || event.code === 'Enter') {
    clickableDiv.click();
  }
});
```

12.4. Поиск всех элементов с данным атрибутом

Задача

Найти в веб-документе все элементы с данным атрибутом.

Решение

Воспользоваться *универсальным селектором* (*) в сочетании с селектором атрибута — таким образом мы найдем все элементы с данным атрибутом независимо от его значения:

```
const elems = document.querySelectorAll('*[class]');
```

С помощью универсального селектора можно найти также все элементы, у которых этот атрибут имеет определенное значение:

```
const reds = document.querySelectorAll('*[class="red"]');
```

Обсуждение

В этом решении продемонстрирован весьма изящный селектор запроса — *универсальный селектор* (*). Он выбирает все элементы. Поэтому его следует использовать, если нужно далее задать *что-то общее* для всех элементов. В данном случае мы таким образом выбрали все элементы с заданным атрибутом.

Для того чтобы проверить, есть ли у элемента данный атрибут, достаточно просто указать его имя в квадратных скобках ([attrname]). В данном примере мы вначале проверяли, есть ли у элемента атрибут `class`. Если он существует, то элемент возвращается в составе коллекции:

```
var elems = document.querySelectorAll('*[class]');
```

Затем мы получили все элементы, у которых атрибут `class` имеет значение `red`. Если вы не уверены в имени класса, можно использовать селектор запроса на соответствие подстроке:

```
const reds = document.querySelectorAll('*[class="red"]');
```

Теперь будут выбраны все элементы, в имени класса которых есть подстрока `red`.

Синтаксис можно изменить и так, чтобы выбирать все элементы, которые не содержат данного значения. Например, для того чтобы найти все элементы `div`, у которых нет заданного имени класса, нужно использовать оператор `:not` следующим образом:

```
const notRed = document.querySelectorAll('div:not(.red)');
```

12.5. Выбор всех элементов определенного типа

Задача

Выбрать все элементы `img` в документе.

Решение

Воспользоваться методом `document.getElementsByTagName()`, передав в него тег `img`:

```
const imgElements = document.getElementsByTagName('img');
```

Обсуждение

Метод `document.getElementsByTagName()` возвращает коллекцию узлов (`NodeList`) для заданного типа элементов — в данном примере это `img`. Полученную коллекцию можно перебрать в цикле как массив, а последовательность узлов совпадает с последовательностью соответствующих элементов в документе (первый элемент `img` на странице доступен по индексу 0 и т. д.):

```
const imgElements = document.getElementsByTagName('img');
for (let i = 0; i < imgElements.length; i += 1) {
  const img = imgElements[i];
  ...
}
```

Как говорилось в рецепте 12.2, коллекцию `NodeList` можно перебрать в цикле, подобно массиву, однако сама коллекция не является объектом `Array`. К `NodeList` нельзя применять такие методы объекта `Array`, как `push()` или `reverse()`. Единственным свойством `NodeList` является `length`, а единственным методом — `item()`. Метод `item()` возвращает элемент, который находится в позиции, заданной переданным в метод индексом:

```
const img = imgElements.item(1); // вторая картинка
```

`NodeList` — очень любопытный объект. Это «живая» коллекция — другими словами, в ней отражаются изменения, внесенные в документ после того, как был создан объект `NodeList`. В примере 12.1 продемонстрируем функциональность «живой» коллекции `NodeList` и использование метода `getElementsByTagName`.

В нем мы с помощью метода `getElementsByTagName` получаем коллекцию `NodeList`, которая состоит из трех изображений, размещенных на веб-странице. В консоль выводится значение свойства `length` этой коллекции, равное 3. Сразу после этого создаются абзац и элемент `img`, который размещается в новом абзаце. Для того чтобы поместить на страницу новый абзац после уже существующих, снова используется метод `getElementsByTagName` — на этот раз ему передается тег `p`.

Но на самом деле нас интересуют не абзацы, а их родительские элементы, которые мы находим в свойстве `parentNode` каждого абзаца.

Новый элемент абзаца размещается в конце родительского элемента существующего абзаца. После этого мы снова выводим свойство `length` созданной ранее коллекции `NodeList` и видим, что его значение равно 4, поскольку в коллекцию добавился новый элемент `img`.

Пример 12.1. Демонстрация применения метода `getElementsByName` и свойств «живой» коллекции `NodeList`

```
<!DOCTYPE html>
<html>
<head>
<title>NodeList</title>
</head>
<body>
  <p></p>
  <p></p>
  <p></p>

  <script>
    const imgs = document.getElementsByTagName('img');
    console.log(imgs.length);
    const p = document.createElement('p');
    const img = document.createElement('img');
    img.src = './img/someimg.jpg';
    p.appendChild(img);

    const paras = document.getElementsByTagName('p');
    paras[0].parentNode.appendChild(p);

    console.log(imgs.length);
  </script>
</body>
</html>
```

При выполнении примера 12.1 в консоль будет выведено следующее:

```



3
4
```

В `getElementsByTagName()` можно передавать в качестве параметра не только определенный тип элемента, но и универсальный селектор (*), чтобы получить все элементы:

```
const allElems = document.getElementsByTagName('*');
```

Читайте также

В коде, продемонстрированном в обсуждении этого рецепта, дочерние узлы перебираются в обычном цикле. В современных браузерах к `NodeList` можно применять непосредственно метод `forEach()`, как показано в рецепте 12.2.

12.6. Исследование дочерних элементов с помощью Selectors API

Задача

Получить список всех дочерних элементов, таких как элементы `img`, для родительских элементов данного типа, таких как `article`, не перебирая всю коллекцию элементов.

Решение

С помощью Selectors API получить доступ только к тем элементам `img`, которые находятся внутри элементов `article`, используя следующий CSS-селектор типа:

```
const imgs = document.querySelectorAll('article img');
```

Обсуждение

В этом API есть два селекторных запроса. Первый из них, `querySelectorAll()`, продемонстрирован в предыдущем примере, второй — это метод `querySelector()`. Разница между этими двумя методами состоит в том, что `querySelectorAll()` возвращает все элементы, соответствующие критерию селектора, а `querySelector()` — только первый найденный результат.

Синтаксис селекторов соответствует синтаксису селекторов CSS — разница лишь в том, что отображенным элементам не назначается определенный стиль. Вместо этого отображенные элементы возвращаются в приложение. В данном примере возвращаются все элементы `img`, наследующие элементам `article`. Для того чтобы получить все элементы `img` независимо от их родительского элемента, нужно использовать следующий код:

```
const imgs = document.querySelectorAll('img');
```

В этом варианте мы получаем все элементы `img`, которые являются прямыми или непрямыми наследниками элемента `article`. Другими словами, если элемент `img` находится внутри элемента `div`, который, в свою очередь, находится внутри `article`, то он попадет в результаты запроса:

```
<article>
  <div>
    
  </div>
</article>
```

Если же мы хотим получить только элементы `img` — прямые наследники элемента `article`, то следует использовать такой код:

```
const imgs = document.querySelectorAll('article > img');
```

Если нас интересуют все элементы `img`, сразу после которых идет абзац, то нужно применить такой запрос:

```
const imgs = document.querySelectorAll('img + p');
```

А если хотим получить элемент `img` с пустым атрибутом `alt`, то нужно написать так:

```
const imgs = document.querySelectorAll('img[alt=""]');
```

Если же нам нужны только те элементы `img`, у которых атрибут `alt` непустой, то используем следующий код:

```
const imgs = document.querySelectorAll('img:not([alt=""])');
```

Для того чтобы найти все элементы `img` с непустым атрибутом `alt`, задействуется псевдоселектор отрицания (`:not`).

В отличие от коллекции, возвращаемой описанным ранее методом `getElementsByName()`, коллекция элементов, возвращаемых методом `querySelectorAll()`, не является «живой». Изменения на странице, которые происходят после ее создания, никак на нее не влияют.



Selectors API, безусловно, прекрасен, однако не следует использовать его для любых запросов к документу. Для того чтобы сохранить производительность приложения, я рекомендую всегда применять для доступа к элементам самый ограничивающий из возможных запросов. Например, если у элемента есть идентификатор, то, чтобы получить такой элемент, гораздо эффективнее (другими словами, быстрее в браузере) взять не `querySelectorAll()`, а `getElementById()`.

Читайте также

Существует три уровня спецификаций селекторов CSS, обозначаемых соответственно Selectors Level 1, Selectors Level 2 и Selectors Level 3. В документации CSS Selectors Level 3 (<https://oreil.ly/rGfxD>) содержатся ссылки на документы, описывающие остальные уровни. В них имеются определения и примеры для разных типов селекторов.

12.7. Изменение класса элемента

Задача

Изменить применяемые к элементу правила CSS, изменив его класс.

Решение

Добавить, удалить или переключить класс элемента, используя свойство `classList`:

```
const element = document.getElementById('example-element');
// Добавляем новый класс
element.classList.add('new-class');
// Удаляем существующий класс
element.classList.remove('existing-class');
// Если у элемента есть класс toggle-me, то удаляем его, если нет — добавляем
element.classList.toggle('toggle-me');
```

Обсуждение

Свойство `classList` позволяет легко манипулировать свойством `class` выбранного элемента. Это удобно, когда нужно обновить или заменить стили элемента без помощи встроенных стилей CSS. Иногда этот прием помогает проверить, есть ли у элемента данный класс, — для этого используется метод `contains`:

```
if (element.classList.contains('new-class')) {
    element.classList.remove('new-class');
}
```

С помощью свойства `classList` можно также добавлять, удалять или переключать несколько классов, передав их как отдельные значения либо посредством оператора `spread`:

```
// Добавляем сразу несколько классов
.classList.add("my-class", "another-class");

// удаляем несколько классов, объединив их с помощью оператора spread
const classes = ["my-class", "another-class"];
div.classList.remove(...classes);
```

12.8. Присвоение элементу атрибута `style`

Задача

Добавить или удалить встроенный стиль для заданного элемента.

Решение

Для того чтобы изменить одно свойство CSS во встроенном стиле, нужно изменить значение этого свойства в атрибуте `style`:

```
elem.style.backgroundColor = 'red';
```

Для того чтобы изменить одно или несколько свойств CSS отдельного элемента, можно использовать метод `setAttribute()`, передав в него правило CSS:

```
elem.setAttribute('style', 'background-color: red; color: white; border: 1px solid black');
```

Эти методики позволяют создать значение встроенного стиля для элемента HTML, которое затем появится в разметке HTML. Чтобы это продемонстрировать, рассмотрим следующий код JavaScript, в котором элементу с идентификатором `card` присваивается значение атрибута `style`:

```
const card = document.getElementById('card');
card.setAttribute(
  'style',
  'background-color: #ecf0f1; color: #2c3e50;'
);
```

В результате получим следующий HTML со встроенным стилем:

```
<div id="card" style="background-color: #ecf0f1; color: #2c3e50;">
...
</div>
```

Обсуждение

В JavaScript есть три способа изменить свойство CSS для заданного элемента. Как показано в данном примере, самый простой из них состоит в том, чтобы напрямую изменить значение стиля, используя свойство `style`:

```
elem.style.width = '500px';
```

Для свойств CSS, в названии которых есть дефис, таких как `font-family` или `background-color`, применяется запись в формате CamelCase:

```
elem.style.fontFamily = 'Courier';
elem.style.backgroundColor = 'rgb(255,0,0)';
```

В этом случае дефисы удаляются, а первые буквы, стоящие после них, заменяются на прописные.

Для изменения свойства стиля можно также применять метод `setAttribute()` или свойство `cssText`. Это удобно, если стилей несколько:

```
// Используем setAttribute
elem.setAttribute('style', 'font-family: Courier; background-color: yellow');
```

```
// или изменяем значение style.cssText
elem.style.cssText = 'font-family: Courier; background-color: yellow';
```

Метод `setAttribute()` позволяет добавить атрибут или изменить значение уже существующего атрибута для элемента веб-страницы. Первый аргумент метода — это имя атрибута (если это элемент HTML, все буквы автоматически заменяются на строчные), а второй — новое значение атрибута.

При задании значения для атрибута `style` должны быть перечислены сразу все свойства CSS, которые нужно изменить, так как при изменении значения этого атрибута все предыдущие значения будут удалены. При изменении стиля посредством метода `setAttribute()`, наоборот, никакие значения, назначенные в таблице стилей или по умолчанию в браузере, не изменяются.

Дополнительно: получение значения существующего стиля

Как правило, получить значение существующего атрибута так же легко, как и присвоить значение, — нужно лишь вместо метода `setAttribute()` использовать `getAttribute()`. Например, для того чтобы узнать класс элемента, нужно сделать следующее:

```
const className = elem.getAttribute('class');
```

Однако получить значение стиля несколько сложнее, так как обычно стиль элемента состоит из нескольких параметров, объединенных в один. Именно такой *вычисленный стиль* элемента мы обычно и хотим получить, чтобы узнать, какие стили действуют для этого элемента в данный момент. К счастью, для этого есть метод `window.getComputedStyle()`, который возвращает вычисленные текущие стили элемента:

```
const style = window.getComputedStyle(elem);
```

Дополнительные возможности

Если нужно добавить или изменить атрибут, то вместо использования метода `setAttribute()` можно создать атрибут и прикрепить его к элементу. Для этого следует создать узел `Attr` с помощью метода `createAttribute()`, присвоить ему значение с помощью свойства `nodeValue`, а затем присвоить атрибут элементу посредством `setAttribute()`:

```
const styleAttr = document.createAttribute('style');
styleAttr.nodeValue = 'background-color: red';
someElement.setAttribute(styleAttr);
```

Используя `createAttribute()` и `setAttribute()` либо непосредственно `setAttribute()`, можно присвоить элементу любое количество атрибутов. Оба эти варианта одинаково эффективны, поэтому, если нет особых соображений, предпочтителен более простой подход — сразу присвоить имя и значение атрибута с помощью `setAttribute()`.

Когда имеет смысл применять `createAttribute()`? Если значением атрибута является ссылка на другую сущность, как бывает в XML. Тогда нужно создать узел `Attr` с помощью `createAttribute()`, так как `setAttribute()` поддерживает только обычные строки.

12.9. Создание абзаца и вставка в него текста

Задача

Создать новый абзац с текстом и вставить его в документ.

Решение

Для вставки текста в элемент нужно воспользоваться методом `createTextNode()`:

```
const newPara = document.createElement('p');
const text = document.createTextNode('New paragraph content');
newPara.appendChild(text);
```

Обсуждение

Текст внутри элемента тоже является объектом в составе DOM. Это узел типа `Text`, и он создается при помощи специального метода `createTextNode()`, который принимает один параметр — строку с текстом.

В примере 12.2 показана веб-страница с элементом `div`, внутри которого находятся четыре абзаца. В коде JavaScript создается еще один абзац с текстом, полученным от пользователя через экранную подсказку. С тем же успехом этот текст может быть получен от другого сервера или каким-то иным способом. Из него создается текстовый узел, который затем прикрепляется к новому абзацу в качестве дочернего узла. Элемент абзаца вставляется в веб-страницу перед первым абзацем.

Пример 12.2. Демонстрация различных методов размещения содержимого на веб-странице

```
<!DOCTYPE html>
<html>
<head>
<title>Adding Paragraphs</title>
</head>
<body>
<div id="target">
  <p>
    There is a language 'little known,'<br />
    Lovers claim it as their own.
  </p>
  <p>
    Its symbols smile upon the land, <br />
    Wrought by nature's wondrous hand;
  </p>
```

```

    <p>
      And in their silent beauty speak,<br />
      Of life and joy, to those who seek.
    </p>
    <p>
      For Love Divine and sunny hours <br />
      In the language of the flowers.
    </p>
  </div>
  <script>
    // получаем элемент div посредством getElementById
    const div = document.getElementById('target');

    // получаем текст абзаца
    const txt = prompt('Enter new paragraph text', '');

    // с помощью getElementsByTagName и индекса коллекции
    // получаем доступ к первому абзацу
    const oldPara = div.getElementsByTagName('p')[0];

    // создаем текстовый узел
    const txtNode = document.createTextNode(txt);

    // создаем новый абзац
    const para = document.createElement('p');

    // вставляем текст в абзац, а абзац — в документ
    para.appendChild(txtNode);
    div.insertBefore(para, oldPara);
  </script>
</body>
</html>

```



Вставлять полученный от пользователя текст на веб-страницу непосредственно, без предварительной очистки — плохая идея. Это все равно что распахнуть дверь, через которую в дом может заползти всякая гадость. Пример 12.9 предназначен исключительно для демонстрационных целей.

12.10. Вставка нового элемента в определенной точке DOM

Задача

Вставить новый абзац перед третьим абзацем внутри элемента `div`.

Решение

Воспользоваться одним из методов, позволяющих получить доступ к третьему абзацу, — например, `getElementsByTagName()`, который возвратит все

абзацы, расположенные внутри элемента `div`. Затем с помощью DOM-методов `createElement()` и `insertBefore()` создать абзац и вставить его перед уже существующим третьим абзацем:

```
// Получаем нужный div
const div = document.getElementById('target');

// Получаем коллекцию абзацев
const paras = div.getElementsByTagName('p');

// Создаем элемент и вставляем в него текст
const newPara = document.createElement('p');
const text = document.createTextNode('New paragraph content');
newPara.appendChild(text);

// Если третий абзац существует, вставляем перед ним новый элемент
// Если не существует, то размещаем новый абзац в конце div
if (paras[2]) {
    div.insertBefore(newPara, paras[2]);
} else {
    div.appendChild(newPara);
}
```

Обсуждение

Метод `document.createElement()` создает элемент HTML, который затем можно вставить в середину или добавить в конец страницы. В данном случае созданный абзац вставляется перед уже существующим с помощью метода `insertBefore()`.

Поскольку мы хотим вставить новый абзац перед уже существующим, нам нужно получить коллекцию абзацев, расположенных внутри элемента `div`, убедиться, что там есть третий абзац, и поместить перед ним новый абзац с помощью метода `insertBefore()`. Если же третьего абзаца не существует, то можем вставить новый элемент в конце элемента `div` с помощью метода `appendChild()`.

12.11. Проверка того, установлен ли флажок

Задача

Проверить, установил ли пользователь флажок в приложении.

Решение

Выбрать элемент флажка (`checkbox`) и проверить состояние свойства `checked`. В данном примере я буду выбирать элемент HTML `input` типа «флажок» по его `id` со значением `check` и отслеживать событие `click`. В ответ на это событие будет вызываться функция `validate`, которая станет проверять свойство `checked` элемента и выводить состояние свойства в консоль:

```
const checkBox = document.getElementById('check');

const validate = () => {
  if (checkBox.checked) {
    console.log('Checkbox is checked')
  } else {
    console.log('Checkbox is not checked')
  }
}

checkBox.addEventListener('click', validate);
```

Обсуждение

Предложить пользователю установить флажок, чтобы что-то подтвердить — например, согласиться с условиями обслуживания, — обычная практика. В таких случаях, если флажок не установлен, кнопка закрытия формы остается недоступной. Чтобы добавить этот функционал в предыдущий пример, нужно сделать следующее:

```
const checkBox = document.getElementById('check');
const acceptButton = document.getElementById('accept');

const validate = () => {
  if (checkBox.checked) {
    acceptButton.disabled = false;
  } else {
    acceptButton.disabled = true;
  }
}

checkBox.addEventListener('click', validate);
```

12.12. Вставка значений в таблицу HTML

Задача

Вычислить сумму всех чисел в столбце таблицы.

Решение

Перебрать в столбце таблицы все ячейки, которые содержат строковые значения, преобразуемые в числа. Преобразовать эти значения в числа и найти сумму:

```
let sum = 0;

// С помощью querySelectorAll получаем все вторые ячейки таблицы
const cells = document.querySelectorAll('td:nth-of-type(2)');

// Перебираем полученные ячейки
```

```
cells.forEach(cell => {  
    sum += Number.parseFloat(cell.firstChild.data);  
});
```

Обсуждение

Селектор `:nth-of-type(n)` позволяет выбрать определенный (*n*-й) дочерний элемент. С помощью селектора `td:nth-of-type(2)` мы выбираем каждый второй дочерний элемент типа `td`. В следующем примере разметки HTML все вторые элементы `td` в таблице содержат числовые значения:

```
<td>Washington</td><td>145</td>
```

Методы `parseInt()` и `parseFloat()` позволяют преобразовать строку в число, но при обработке чисел из таблицы HTML лучше использовать `parseFloat()` — разве что вы абсолютно уверены, что все числа в таблице целые. Метод `parseFloat()` работает и с целыми, и с вещественными числами.

В примере 12.3 показано, как преобразовать в числа значения из таблицы HTML и просуммировать их, а затем вставить в таблицу строку с полученной суммой. Здесь для получения данных в коде используется метод `document.querySelectorAll()` с другим вариантом селектора CSS — `td + td`. Он выбирает все ячейки таблицы, предыдущим элементом которых также являются ячейки таблицы.

Пример 12.3. Преобразование значений таблицы в числа и суммирование результатов

```
<!DOCTYPE html>  
<html lang="en">  
<head>  
    <meta charset="UTF-8">  
    <meta name="viewport" content="width=device-width, initial-scale=1.0">  
    <meta http-equiv="X-UA-Compatible" content="ie=edge">  
    <title>Adding Up Values in an HTML Table</title>  
</head>  
<body>  
    <h1>Adding Up Values in an HTML Table</h1>  
    <table>  
        <tbody id="table1">  
            <tr>  
                <td>Washington</td><td>145</td>  
            </tr>  
            <tr>  
                <td>Oregon</td><td>233</td>  
            </tr>  
            <tr>  
                <td>Missouri</td><td>833</td>  
            </tr>  
        </tbody>  
    </table>  
  
    <script>
```

```
let sum = 0;

// С помощью querySelector выбираем все вторые ячейки таблицы
const cells = document.querySelectorAll('td:nth-of-type(2)');

// Перебираем полученные ячейки
cells.forEach(cell => {
    sum += Number.parseFloat(cell.firstChild.data);
});

// Добавляем сумму в конец таблицы
const newRow = document.createElement('tr');

// Первая ячейка
const firstCell = document.createElement('td');
const firstCellText = document.createTextNode('Sum:');
firstCell.appendChild(firstCellText);
newRow.appendChild(firstCell);

// Вторая ячейка с суммой
const secondCell = document.createElement('td');
const secondCellText = document.createTextNode(sum);
secondCell.appendChild(secondCellText);
newRow.appendChild(secondCell);

// Добавляем в таблицу еще одну строку
document.getElementById('table1').appendChild(newRow);
</script>
</body>
</html>
```

Возможность вычислить сумму или выполнить какую-либо другую операцию с табличными данными очень полезна в случае динамических обновлений, например из записей базы данных. Извлеченные оттуда данные могут не содержать сумм или, возможно, вы не хотите выводить суммы, пока этого не потребует пользователь. Иногда нужно, чтобы у него была возможность управлять видом результатов в таблице, и для вычисления суммы следует нажать на определенную кнопку.

Добавить строки в таблицу очень легко — нужно лишь запомнить последовательность действий.

1. Создать новую строку таблицы с помощью `document.createElement("tr")`.
2. Создать в новой строке все необходимые ячейки с помощью `document.createElement("td")`.
3. Вставить в каждую ячейку данные с помощью `document.createTextNode()`, передав в этот метод текст узла (в частности, числа, которые автоматически преобразуются в строки).
4. Прикрепить текстовый узел к ячейке таблицы.
5. Прикрепить ячейку таблицы к строке таблицы.
6. Прикрепить строку таблицы к таблице. Очистить, повторить.

Дополнительно: `forEach` и `querySelectorAll`

В предыдущем примере для перебора результатов `querySelectorAll()` применялся метод `forEach()`, несмотря на то что `querySelectorAll()` возвращает не массив, а `NodeList`. Метод `forEach()` предназначен для обработки массивов, однако, как говорилось в рецепте 12.2, в современных браузерах реализован `NodeList.prototype.forEach()`, что позволяет перебирать `NodeList`, используя синтаксис `forEach()`. Вместо этого можно было бы задействовать цикл:

```
let sum = 0;

// С помощью querySelector выбираем все вторые ячейки
// таблицы
let cells = document.querySelectorAll("td:nth-of-type(2)");

for (var i = 0; i < cells.length; i++) {
    sum += parseFloat(cells[i].firstChild.data);
}
```

Дополнительно: модуляризация глобальных методов

Сейчас прилагается все больше усилий для *модуляризации* JavaScript, поэтому в стандарте ECMAScript 2015 методы `parseFloat()` и `parseInt()` прикреплены к `Number` в качестве статических методов объекта:

```
// модульный метод
const modular = Number.parseInt('123');
// глобальный метод
const global = parseInt('123');
```

Модули получили широкое распространение в современных браузерах, для поддержки старыми браузерами их можно заменить на полизаполнения, используя инструменты вроде Babel или собственные средства:

```
if (Number.parseInt === undefined) {
    Number.parseInt = window.parseInt
}
```

12.13. Удаление строк из таблицы HTML

Задача

Удалить из таблицы HTML одну или несколько строк.

Решение

Применить к строке таблицы HTML метод `removeChild()`. В результате и строка, и все ее дочерние элементы, такие как ячейки, будут удалены:

```
const parent = row.parentNode;  
const oldrow = parent.removeChild(parent);
```

Обсуждение

Удаляя из веб-документа элемент, мы удаляем также все его дочерние элементы. При такой *обрезке* дерева DOM возвращается ссылка на удаленный элемент, чтобы можно было обработать его содержимое, прежде чем оно окончательно исчезнет. Это полезно, если мы хотим обеспечить возможность отмены данной операции, когда пользователь случайно выбрал не ту строку таблицы.

Чтобы продемонстрировать, в чем заключается обрезка DOM, в примере 12.4 с помощью методов DOM `createElement()` и `createTextNode()` создаются строки и ячейки таблицы и в эти ячейки вставляется текст. Когда создается очередная строка таблицы, к ней прикрепляется обработчик события `click`. Теперь, если пользователь щелкает на строке, вызывается функция, которая удаляет ее из таблицы. Затем ячейки удаленной строки перебираются, из них извлекаются данные, которые объединяются в строку, и она выводится на странице.

Пример 12.4. Вставка и удаление строк таблицы вместе с их ячейками и данными в ячейках

```
<!DOCTYPE html>  
<html lang="en">  
  <head>  
    <meta charset="UTF-8" />  
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />  
    <meta http-equiv="X-UA-Compatible" content="ie=edge" />  
    <title>Deleting Rows from an HTML Table</title>  
    <style>  
      table {  
        border-collapse: collapse;  
      }  
      td,  
      th {  
        padding: 5px;  
        border: 1px solid #ccc;  
      }  
      tr:nth-child(2n + 1) {  
        background-color: #eeffee;  
      }  
    </style>  
  </head>  
  <body>  
    <h1>Deleting Rows from an HTML Table</h1>  
    <table id="mixed">  
      <tr>  
        <th>Value One</th>  
        <th>Value two</th>  
        <th>Value three</th>  
      </tr>  
    </table>
```

```
<div id="result"></div>
<script>
// Содержимое таблицы
const values = new Array(3);
values[0] = [123.45, 'apple', true];
values[1] = [65, 'banana', false];
values[2] = [1034.99, 'cherry', false];

const mixed = document.getElementById('mixed');
const tbody = document.createElement('tbody');

function pruneRow() {
// Удаляем строку
const parent = this.parentNode;
const oldRow = parent.removeChild(this);

// Помещаем данные из удаленной строки в dataString
let dataString = '';
oldRow.childNodes.forEach(row => {
    dataString += `${row.firstChild.data} `;
});

// Выводим сообщение
const msg = document.createTextNode(`removed ${dataString}`);
const p = document.createElement('p');
p.appendChild(msg);
document.getElementById('result').appendChild(p);
}

// Для каждой строки внешнего массива
values.forEach(value => {
    const tr = document.createElement('tr');

    // Для каждой ячейки внешнего массива
    // создаем элементы td и text,
    // потом прикрепляем text к td, а td — к tr
    value.forEach(cell => {
        const td = document.createElement('td');
        const txt = document.createTextNode(cell);
        td.appendChild(txt);
        tr.appendChild(td);
    });

    // Прикрепляем обработчик события
    tr.onclick = pruneRow;

    // Прикрепляем строку к таблице
    tbody.appendChild(tr);
    mixed.appendChild(tbody);
});
</script>
</body>
</html>
```

12.14. Скрытие частей страницы

Задача

Временно скрыть существующий элемент страницы и его дочерние элементы.

Решение

Для того чтобы скрыть, а затем отобразить элемент, можно использовать свойство CSS `visibility`:

```
msg.style.hidden = 'visible'; // отобразить элемент
msg.style.hidden = 'hidden';  // скрыть элемент
```

Либо использовать свойство CSS `display`:

```
msg.style.display = 'block'; // отобразить элемент
msg.style.display = 'none';  // скрыть элемент
```

Обсуждение

Для отображения и скрытия элементов можно задействовать оба эти свойства CSS — и `visibility`, и `display`. Но между ними есть важное различие, от которого будет зависеть то, какое из этих двух свойств вы выберете.

Свойство `visibility` определяет визуальное отображение элемента, но наличие этого свойства влияет и на другие элементы. Даже когда элемент скрыт, он все равно занимает место на странице. Наличие свойства `display`, наоборот, означает, что элемент будет полностью удален из разметки страницы.

Свойство `display` может принимать несколько значений, из которых нам наиболее интересны следующие четыре:

- `none` — когда значение `display` равно `none`, элемент полностью удаляется из отображения;
- `block` — когда значение `display` равно `block`, элемент считается блоком, то есть перед ним и после него стоят символы разрыва строки;
- `inline-block` — когда значение `display` равно `inline-block`, содержимое элемента форматируется как блок, но затем элемент передается в разметку как строковый;
- `inherit` — это отображение элемента по умолчанию. Свойство `display` наследуется от родительского элемента.

Есть и другие значения `display`, но в приложениях JavaScript обычно используются те, что перечислены здесь.

Скорее всего, вы будете применять свойство CSS `display` — за исключением случаев абсолютного позиционирования скрытого элемента. Иначе скрытый

элемент будет нарушать разметку страницы, сдвигая остальные элементы вниз или вправо в зависимости от их типа.

Еще один способ удалить элемент из отображения страницы состоит в том, чтобы полностью вынести его за пределы экрана, используя сдвиг влево на отрицательное число. Это иногда работает, особенно при создании слайдеров, которые должны сдвигаться слева направо. Такой подход рекомендуется применять и с целью приспособить интерфейс для людей с ограниченными возможностями, чтобы скрыть элементы страницы, которые должны отображаться только на специализированных устройствах (assistive technology devices, AT), но не на обычных экранах.

12.15. Создание окон, всплывающих по наведению указателя мыши

Задача

Сделать так, чтобы миниатюра изображения реагировала на наведение указателя мыши и чтобы в этот момент выводилась дополнительная информация.

Решение

Такое взаимодействие основано на четырех отдельных действиях.

Во-первых, чтобы отображать и удалять всплывающее окно, необходимо, чтобы все миниатюры перехватывали события `mouseover` и `mouseout` соответственно. В следующем фрагменте кода показан кросс-браузерный обработчик событий, привязанный ко всем изображениям на странице:

```
window.onload = () => {
  const imgs = document.querySelectorAll('img');
  imgs.forEach(img => {
    img.addEventListener(
      'mouseover',
      () => {
        getInfo(img.id);
      },
      false
    );

    img.addEventListener(
      'mouseout',
      () => {
        removeWindow();
      },
      false
    );
  });
};
```

Во-вторых, нам нужно каким-то образом получать сведения об элементе, на который наводится указатель мыши, чтобы знать, какую информацию следует вставить во всплывающее окно. Это может быть информация, уже имеющаяся на странице либо получаемая от веб-сервера:

```
function getInfo(id) {
    // получаем данные
}
```

В-третьих, нужно либо сделать всплывающее окно видимым (если оно уже есть, но не отображается), либо создать его. В следующем коде всплывающее окно создается непосредственно под объектом и чуть правее, после того как в ответ на обращение к веб-серверу выдается информация об элементе. Положение всплывающего окна определяется с помощью метода `getBoundingClientRect()`, а для его создания используются методы `createElement()` и `createTextNode()`:

```
// Вычисляем положение всплывающего окна
function compPos(obj) {
    const rect = obj.getBoundingClientRect();
    let height;
    if (rect.height) {
        height = rect.height;
    } else {
        height = rect.bottom - rect.top;
    }
    const top = rect.top + height + 10;
    return [rect.left, top];
}

function showWindow(id, response) {
    const img = document.getElementById(id);

    console.log(img);
    // Получаем положение всплывающего окна
    const loc = compPos(img);
    const left = `${loc[0]}px`;
    const top = `${loc[1]}px`;

    // Создаем всплывающее окно
    const div = document.createElement('popup');
    div.id = 'popup';
    const txt = document.createTextNode(response);
    div.appendChild(txt);

    // Применяем стили к всплывающему окну
    div.setAttribute('class', 'popup');
    div.setAttribute('style', `position: fixed; left: ${left}; top: ${top}`);
    document.body.appendChild(div);
}
```

Наконец, когда возникает событие `mouseout`, необходимо либо скрыть всплывающее окно, либо удалить его — в зависимости от ситуации. Поскольку при событии

`mouseover` наше приложение создает новое всплывающее окно, то в обработчике события `mouseout` имеет смысл его удалить:

```
function removeWindow() {  
    const popup = document.getElementById('popup');  
    if (popup) popup.parentNode.removeChild(popup);  
}
```

Обсуждение

Создание информационных или вспомогательных всплывающих окон не должно вызывать сложностей, если сделать все действия простыми и выполнять четыре операции, описанные в этом рецепте. Если всплывающее окно предоставляет справочную информацию для элементов формы, то ее можно оставить на странице в скрытом виде и просто делать всплывающие окна видимыми по мере необходимости. Но если страница насчитывает сотни элементов, то чтобы повысить производительность, лучше получать информацию для всплывающих окон, обращаясь к веб-сервису в нужный момент.

Выбирая в данном примере положение всплывающего окна, я не стал размещать его прямо над объектом. Причина в том, что мы не получали координаты мыши, чтобы разместить всплывающее окно прямо над указателем, — это гарантировало бы, что мы никогда не наведем его непосредственно на всплывающее окно. Но если положение всплывающего окна статично и окно немного перекрывает объект, то читатель веб-страницы может навести указатель мыши на всплывающее окно, и тогда активируется событие, скрывающее это окно... что, в свою очередь, активирует событие, отображающее окно, и т. д. Это вызовет эффект мерцания, не говоря уже о повышенной нагрузке на сеть.

Если же разрешить дальнейшую обработку событий мыши, возвращая `true` из каждой функции обработки событий, то, когда читатель веб-страницы переместит указатель мыши на всплывающее окно, оно не исчезнет. Но если сдвинуть указатель с изображения на всплывающее окно, а затем на остальную часть страницы, то событие, скрывающее окно, не активируется и всплывающее окно останется видимым.

Лучше всего размещать всплывающее окно не прямо над объектом, а под ним (или сбоку, или в специальном месте страницы).

12.16. Валидация данных формы

Задача

Веб-приложение получает данные от пользователей через формы HTML. Но прежде чем передавать данные на сервер, нужно убедиться, что они верны, полны и валидны, после чего предоставить пользователю обратную связь.

Решение

Задействовать атрибуты валидации форм из HTML5, при необходимости дополнив их средствами валидации строк из внешней библиотеки:

```
<form id="example" name="example" action="" method="post">
  <fieldset>
    <legend>Example Form</legend>
    <div>
      <label for="email">Email (required):</label>
      <input type="email" id="email" name="email" value="" required />
    </div>
    <div>
      <label for="postal">Postal Code:</label>
      <input type="text" pattern="[0-9]*" id="postal" name="url" value="" />
    </div>
    <div id="error"></div>
    <div>
      <input type="submit" value="Submit" />
    </div>
  </fieldset>
</form>
```

Для валидации по мере ввода данных пользователем можно задействовать одну из стандартных библиотек, например validator.js (<https://github.com/validatorjs/validator.js>):

```
<script type="text/javascript">
  function inputValidator(id, value) {
    // Проверяем корректность адреса электронной почты
    if (id === 'email') {
      return validator.isEmail(value);
    }

    // Проверяем корректность почтового индекса США
    if (id === 'postal') {
      return validator.isPostalCode(value, 'US');
    }

    return false;
  }

  const inputs = document.querySelectorAll('#example input');

  inputs.forEach(input => {
    // Активируем событие при каждом изменении в поле input
    input.addEventListener('input', () => {
      // Передаем значение из поля input в функцию валидации
      const valid = inputValidator(input.id, input.value);
      // Если результаты ввода некорректны,
      // присваиваем атрибуту aria-invalid значение true
    });
  });
</script>
```

```
        if (!valid && input.value.length > 0) {  
            this.setAttribute('aria-invalid', 'true');  
        }  
    });  
});  
</script>
```

Обсуждение

Сегодня у нас нет необходимости писать собственные средства валидации форм, разве что мы имеем дело с каким-то уж очень эксцентричным поведением формы и/или данных. Под эксцентричным поведением я понимаю что-то совершенно из ряда вон выходящее, когда проще написать код самому, чем внедрить библиотеку JavaScript, — что-то вроде валидации правила «значение в поле формы должно быть строкой во все дни недели, кроме четверга, когда оно должно быть числом, а в четные месяцы — наоборот».

Существует множество библиотек; я показал лишь одну из них. Validator.js — хорошая, простая и удобная библиотека, которая позволяет выполнять валидацию различных типов строк. Она не требует изменять поля форм, так что ее можно легко отключить, не переделывая форму. Все стили и размещение сообщений об ошибках также определяет разработчик.

В данном примере кода обработчик события привязывается к каждому элементу `input`. Когда пользователь изменяет поле формы, активируется обработчик события `input`. Он вызывает функцию `inputValidator`, которая проверяет введенное значение средствами библиотеки `validator.js`. Если значение некорректно, то с помощью простых стилей CSS у поля ввода появляется красная рамка. Если значение корректно, то эти стили не применяются.

Иногда возникает необходимость в библиотеке меньшего размера, которая бы проверяла только один тип данных. К таким данным со сложной проверкой относятся, например, кредитные карты. Вы можете ввести значение в нужном формате, однако для того, чтобы оно было признано корректным номером кредитной карты, оно должно удовлетворять особым правилам.

В дополнение к другим библиотекам валидации можно подключить библиотеку валидации кредитных карт, такую как `Payment` (<https://github.com/jessepollak/payment>), с простым API валидации. Вот пример того, как после загрузки формы указывается, что данное поле предназначено для ввода номера кредитной карты:

```
const cardInput = document.querySelector('input.cc-num');  
  
Payment.formatCardNumber(cardInput);
```

Затем, после отправки формы, выполняется валидация номера кредитной карты:

```
var valid = Payment.fns.validateCardNumber(cardInput.value);

if (!valid) {
  message.innerHTML = 'You entered an invalid credit card number';
  return false;
}
```

Библиотека не просто проверяет формат — она также гарантирует, что данное значение является корректным номером карты для всех основных платежных систем. В зависимости от способа обработки кредитных карт обработчик платежа предоставляет аналогичный функционал на стороне клиента. Например, API валидации кредитных карт включен в обработчик платежей Stripe.js (<https://oreil.ly/GqPVh>) для платежной системы Stripe.

Наконец, можно связать валидацию на стороне клиента и на стороне сервера, действуя для этого ту же или какую-либо другую библиотеку. В данном примере мы использовали `validator.js` в браузере, но эту библиотеку можно применять и для валидации входных данных на стороне сервера в приложениях Node.

Дополнительно: способы валидации форм в HTML5

В HTML5 есть множество средств валидации форм, которые не требуют применения JavaScript. В их число входят следующие:

- `min` и `max` — минимальное и максимальное значения числовых входных данных;
- `minlength` и `maxlength` — минимальная и максимальная длина вводимой строки;
- `pattern` — регулярное выражение, которому должны соответствовать входные данные;
- `required` — данные, которые обязательно следует ввести, прежде чем можно будет отправить форму;
- `type` — разработчик может указать конкретный тип вводимых данных, такой как дата, адрес электронной почты, число, пароль, URL и некоторые другие специальные типы данных.

Кроме этого, для выбора корректно и некорректно введенных данных могут использоваться псевдоселекторы CSS `:valid` и `:invalid`.

По этой причине для валидации простых форм JavaScript может вообще не понадобиться. Но если вам нужен полный контроль над поведением приложения при валидации формы, то лучше задействовать библиотеку JavaScript, чем полагаться на спецификации валидации форм, предоставляемые HTML5 и CSS. Однако при этом не забудьте реализовать в форме функции для специализированных устройств. Подробнее об этом рекомендую почитать в статье *WebAIM: Creating Accessible Forms* (<https://oreil.ly/5oL3E>).

12.17. Выделение ошибочно заполненных полей форм и реализация специальных возможностей

Задача

Выделять поля формы, в которые введены некорректные данные, и гарантировать, что это будет заметно всем пользователям веб-страницы.

Решение

Выделять поля формы с некорректно введенными данными средствами CSS. Применить разметку WAI-ARIA (Web Accessibility Initiative-Accessible Rich Internet Applications), чтобы гарантировать доступность интерфейса для всех пользователей:

```
[aria-invalid] {
    background-color: #f5b2b2;
}
```

К тем полям, которые нужно валидировать, привязывается обработчик события `oninput`. Он вызывает функцию, которая проверяет корректность данных, введенных в поле. Если введенное значение некорректно, то пользователю выводится информация об ошибке, а поле выделяется цветом:

```
function validateField() {
    // проверяем, число ли это
    if (typeof this.value !== 'number') {
        this.setAttribute('aria-invalid', 'true');
        generateAlert(
            'You entered an invalid value. Only numeric values are allowed'
        );
    }
}
```

```
document.getElementById('number').oninput = validateField;
```

К тем полям, которые обязательно должны быть заполнены, привязывается обработчик события `onblur`, который вызывает функцию, проверяющую, было ли введено значение:

```
function checkMandatory() {
    // проверяем наличие данных
    if (this.value.length === 0) {
        this.setAttribute('aria-invalid', 'true');
        generateAlert('A value is required in this field');
    }
}
```

```
document.getElementById('required-field').onblur = checkMandatory;
```

Если валидация выполняется при отправке формы, то нужно предусмотреть отмену отправки данных в случае неудачной валидации.

Обсуждение

WAI-ARIA позволяет пометить определенные поля и действия, так чтобы специализированные устройства реагировали на эти поля и действия соответствующим образом и люди, нуждающиеся в таких устройствах, тоже могли заполнить форму.

Для экранных дикторов необходимо использовать элементы формы с атрибутом `aria-invalid` со значением `true` (или добавить этот атрибут в коде JavaScript). В результате экранный диктор будет выдавать звуковые предупреждения — они заменяют цветовое выделение элементов, применяемое на обычных устройствах.



Подробнее о WAI-ARIA можно узнать в разделе Web Accessibility Initiative на сайте W3C (<https://oreil.ly/8wGnc>). Чтобы проверить, правильно ли работает ваш код для экранного диктора в Windows, я рекомендую использовать NVDA (<http://www.nvaccess.org>) — бесплатный экранный диктор с открытым кодом. Для macOS советую применять инструмент VoiceOver, встроенный в браузер Safari.

Кроме того, есть еще атрибут `role`. Он может принимать несколько значений, одно из которых, `alert`, активирует похожее поведение экранных дикторов (обычно это произношение содержимого поля). При валидации элементов форм необходимо давать такие подсказки. Можно выполнить валидацию формы перед отправкой данных и предоставить пользователю текстовое описание всех ошибок. Но лучше проводить валидацию данных для каждого поля в отдельности, после того как пользователь заполнит его, чтобы не выдавать в конце сразу много раздражающих сообщений об ошибках.

При валидации следует сделать так, чтобы пользователь всегда точно знал, какое именно поле заполнено неверно. Для этого применяются визуальные индикаторы. Такой способ обозначения ошибок не обязателен — это просто элемент вежливости.

При выделении цветом некорректно заполненных элементов формы следует избегать цветов, сливающихся с фоном. Если фон страницы белый, а вы используете темно-желтый, серый, красный, синий, зеленый или другой подобный цвет, то выделение будет достаточно контрастным даже в том случае, если страницу просматривает дальтоник. В примере для выделения полей формы я использовал темно-розовый.

Цвет можно задавать непосредственно, но имеет смысл привязать оба значения — цвет и значение атрибута `aria-invalid` — к одному параметру CSS, чтобы

их было проще обновлять. К счастью, *селекторы атрибутов* CSS значительно упрощают эту задачу.

Кроме выделения цветом нужно дать текстовое описание ошибки, чтобы пользователь точно знал, в чем состоит проблема.

Важно учесть, как именно будет выводиться информация. Никто не любит окна предупреждений браузера, и их следует по возможности избегать. Окна предупреждений закрывают форму, и единственный способ добраться до элемента формы — закрыть окно с сообщением об ошибке. Лучше размещать информацию на странице, рядом с формой. Нам также хотелось бы, чтобы сообщение об ошибке было доступно всем, кто использует вспомогательные технологии, такие как экранный диктор. Это легко сделать, присвоив атрибут ARIA `alert` со значением `role` тем элементам, для которых должны срабатывать предупреждения в экранных дикторах и других специализированных устройствах.

Наконец, последняя выгода атрибута `aria-invalid` состоит в том, что при отправке формы по нему можно получить все некорректно заполненные поля. Достаточно выбрать все поля, у которых есть этот атрибут. Если будет найдено хотя бы одно такое поле, значит, в форме все еще остаются некорректно заполненные поля и их нужно исправить.

В примере 12.5 показано, как выделить некорректно введенные данные в одном из элементов формы и отсутствующие данные — в другом. Также здесь перед отправкой формы выполняется проверка, не осталось ли в ней некорректно заполненных полей. Отправлять форму разрешается только в том случае, если все в порядке.

Пример 12.5. Предоставление визуальных и других подсказок при валидации полей формы

```
<!DOCTYPE html>
<head>
<title>Validating Forms</title>
<style>
[aria-invalid] {
    background-color: #ffeeee;
}
[role="alert"] {
    background-color: #ffcccc;
    font-weight: bold;
    padding: 5px;
    border: 1px dashed #000;
}

div {
    margin: 10px 0;
    padding: 5px;
    width: 400px;
    background-color: #ffffff;
}
```

```
</style>
</head>
<body>

<form id="testform">
  <div><label for="firstfield">*First Field:</label><br />
    <input id="firstfield" name="firstfield" type="text" aria-required="true"
      required />
  </div>
  <div><label for="secondfield">Second Field:</label><br />
    <input id="secondfield" name="secondfield" type="text" />
  </div>
  <div><label for="thirdfield">Third Field (numeric):</label><br />
    <input id="thirdfield" name="thirdfield" type="text" />
  </div>
  <div><label for="fourthfield">Fourth Field:</label><br />
    <input id="fourthfield" name="fourthfield" type="text" />
  </div>

  <input type="submit" value="Send Data" />
</form>

<script>

document.getElementById("thirdfield").onchange=validateField;
document.getElementById("firstfield").onblur=mandatoryField;
document.getElementById("testform").onsubmit=finalCheck;

function removeAlert() {

  var msg = document.getElementById("msg");
  if (msg) {
    document.body.removeChild(msg);
  }
}

function resetField(elem) {
  elem.parentNode.setAttribute("style","background-color: #ffffff");
  var valid = elem.getAttribute("aria-invalid");
  if (valid) elem.removeAttribute("aria-invalid");
}

function badField(elem) {
  elem.parentNode.setAttribute("style", "background-color: #ffeeee");
  elem.setAttribute("aria-invalid","true");
}

function generateAlert(txt) {

  // Создаем новые текстовые элементы и элементы div,
  // присваиваем им значения атрибутов role, class и id
  var txtNd = document.createTextNode(txt);
  msg = document.createElement("div");
  msg.setAttribute("role","alert");
  msg.setAttribute("id","msg");
  msg.setAttribute("class","alert");
```

```
// Прикрепляем текстовый элемент к div, а div — к document
msg.appendChild(txtNd);
document.body.appendChild(msg);
}

function validateField() {

    // Удаляем все предупреждения независимо от их значения
    removeAlert();

    // Проверяем, является ли введенное значение числом
    if (!isNaN(this.value)) {
        resetField(this);
    } else {
        badField(this);
        generateAlert("You entered an invalid value in Third Field. " +
            "Only numeric values such as 105 or 3.54 are allowed");
    }
}

function mandatoryField() {

    // Удаляем все предупреждения
    removeAlert();

    // Проверяем, введено ли значение
    if (this.value.length > 0) {
        resetField(this);
    } else {
        badField(this);
        generateAlert("You must enter a value into First Field");
    }
}

function finalCheck() {
    removeAlert();
    var fields = document.querySelectorAll("[aria-invalid='true']");
    if (fields.length > 0) {
        generateAlert("You have incorrect field entries that must be fixed " +
            "before you can submit this form");
        return false;
    }
}

</script>

</body>
```

Если какое-либо из валидируемых полей приложения окажется некорректным, то атрибут `aria-invalid` станет равным `true`, а атрибут `ARIA role` примет значение `alert`, чтобы выводилось сообщение об ошибке, как показано на рис. 12.2. Когда ошибка будет исправлена, атрибут `aria-invalid` и сообщение об ошибке будут удалены. Оба эти атрибута приводят к изменению фонового цвета в поле формы.

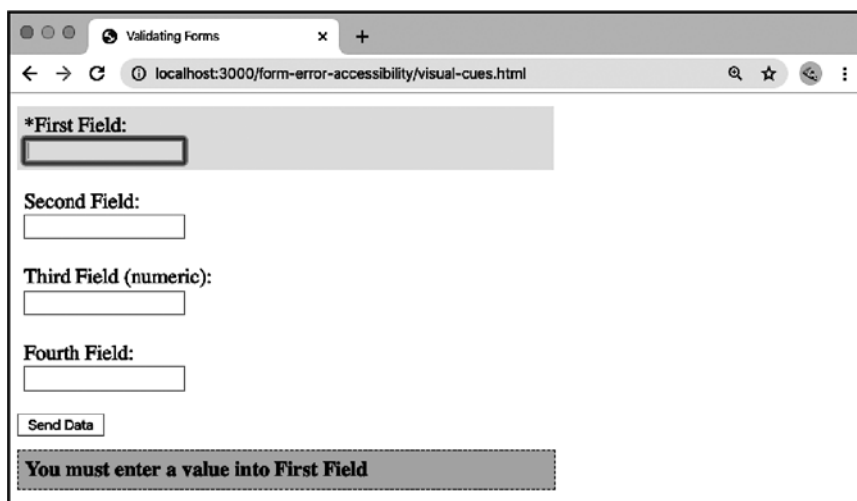


Рис. 12.2. Выделение некорректно заполненного поля формы

Обратите внимание: когда вводятся правильные данные, код элемента, в который обернуто соответствующее поле формы, возвращается в свое нормальное состояние. Таким образом, после того как данные в поле исправлены, поле больше не отображается как требующее ввода данных или содержащее ошибочные данные. Я удаляю старые сообщения об ошибке независимо от предыдущего события, так как после нового события эти сообщения больше не актуальны. Также можно было бы отключить или даже скрыть правильно заполненные элементы формы, чтобы выделить те элементы, где данные отсутствуют или содержатся некорректные данные. Однако я не советую так поступать. Вводя отсутствующую информацию, пользователь может обнаружить, что данные, введенные им в других полях, неверны. Если усложнить пользователям исправление информации в других полях, им это не понравится и они будут недовольны компанией, человеком или организацией, предоставившими такую форму.

Можно поступить иначе: выполнять валидацию только при отправке формы. Именно так сделано в большинстве встроенных библиотек. Вместо того чтобы проверять каждое поле — введены ли там обязательные значения и корректны ли они — по мере заполнения формы, все правила валидации применяются только при отправке формы. Таким образом, пользователь может заполнять форму в произвольной последовательности, не получая раздражающих сообщений об ошибках валидации при переходе от одного поля к другому.

Выделение с помощью JavaScript тех полей формы, в которых отсутствуют данные либо они некорректны, — лишь часть процедуры отправки формы. Следует предусмотреть и вариант, когда JavaScript будет отключен, — на этот случай необходимо предоставить пользователю такую же обратную связь после обработки данных из формы на сервере с выводом результата на отдельной странице.

Также важно заранее отметить поля формы, заполнение которых обязательно. Для этого рядом с именем поля ставят звездочку с пометкой, что все поля формы, отмеченные таким образом, обязательны для заполнения. Чтобы гарантировать получение этой информации на специализированных устройствах, используйте атрибуты `aria-required` и `role`. Вместе с `aria-required` я советую применять атрибут HTML5 `required`, который активирует встроенную валидацию браузера.

Читайте также

В рецепте 12.16 описываются библиотеки для валидации форм и модули, упрощающие валидацию форм. Мы также кратко рассмотрим использование методов декларативной валидации форм в HTML5.

12.18. Создание автоматически обновляемой области, доступной на специализированных устройствах

Задача

На веб-странице есть периодически обновляемый раздел — например, со списком последних обновлений некоего файла или с последними записями в Twitter по какой-то теме. Нужно гарантировать, что при обновлении страницы пользователи экранных дикторов тоже получают новую информацию.

Решение

Присвоить обновляемому элементу атрибуты области WAI-ARIA:

```
<div id="update" role="log" aria-live="polite" aria-atomic="true"
      aria-relevant="additions">
</div>
```

Обсуждение

В разделе веб-страницы, обновляемом после ее загрузки без непосредственного участия пользователя, нужно задействовать динамические области (Live Regions) WAI-ARIA. Это, пожалуй, самый простой для реализации функционал ARIA, и польза от него видна сразу же. Для этого не нужен никакой дополнительный код, кроме кода JavaScript, выполняющего обновление страницы.

```
<div id="update" role="log" aria-live="polite" aria-atomic="true"
      aria-relevant="additions"></div>
```

Перечислим использованные атрибуты (слева направо). Первым идет `role` со значением `log`, который будет применяться для запроса данных об обновлениях

из файла журнала. Этот атрибут также может принимать значение `status` для обновления состояния или более общее значение `region`, если цель обновления не определена.

Атрибуту области `aria-live` присвоено значение `polite`, так как обновленные данные не являются критически важными. Для экранного диктора значение `polite` означает, что обновление нужно озвучить, но не прерывать для этого текущую задачу. Если бы я использовал здесь значение `assertive`, то экранный диктор прочитал бы содержимое элемента, прервав то, что выполнялось в этот момент. Если задача не является критически важной, всегда следует брать значение `polite`.

Атрибуту `aria-atomic` присвоено значение `false`, чтобы экранный диктор озвучивал только новые данные в зависимости от значения `aria-relevant`. Очень раздражает, когда при каждом обновлении экранный диктор заново озвучивает весь список — именно так и будет, если присвоить атрибуту `aria-atomic` значение `true`.

Наконец, последнему атрибуту, `aria-relevant`, присвоено значение `additions`, поскольку нас не интересуют верхние записи, даже если при обновлении они будут удалены. Вообще-то это значение присваивается данному атрибуту `aria-relevant` по умолчанию, поэтому, строго говоря, его можно было бы и не писать. Кроме того, не все вспомогательные устройства поддерживают этот атрибут. Но я считаю, что лучше указать его явно. Также атрибут `aria-relevant` может принимать значения `removals`, `text` и `all` (для всех событий). Атрибуту можно присваивать несколько значений, разделяя их пробелом.

Пожалуй, этот функционал WAI-ARIA впечатлил меня больше всего. Одна из первых задач по получению удаленных данных, выполняемая мною несколько лет назад, требовала размещать на веб-странице новую информацию. Было сущим мучением тестировать страницу для экранного диктора (на тот момент это был JAWS) и каждый раз при ее обновлении снова слушать тишину. Могу представить, каково приходилось тем, кому этот функционал действительно нужен.

Теперь этот функционал у нас есть, и он простой. Все в выигрыше.

Получение удаленных данных

Одна из суперспособностей JavaScript — возможность получать данные и обрабатывать их в браузере без обновления страницы. Отслеживание данных в реальном времени, чаты, обновления лент в соцсетях и многое другое — все это стало возможным благодаря тому, что JavaScript может отправлять запросы на сервер и обновлять содержимое страницы. В этой главе вы узнаете, как создаются и обрабатываются эти запросы.



Возможно, вам также встречалась аббревиатура AJAX, которая расшифровывается как Asynchronous JavaScript and XML — асинхронный JavaScript и XML. Изначально эта технология действительно предназначалась для получения XML, но сейчас AJAX используется как обобщенный термин для разных способов обмена данными между удаленным сервером и браузером.

13.1. Запрос удаленных данных с помощью Fetch API

Задача

Запросить данные с удаленного сервера.

Решение

Использовать Fetch API, который позволяет создавать запросы и управлять ответами на них. Для того чтобы создать простой запрос, достаточно передать URL в функцию `fetch`, которая возвращает ответ сервера в виде промиса. В следующем примере на сервер передается URL, затем выполняется синтаксический анализ ответа, полученного в формате JSON, и результат выводится в консоль:

```
const url = 'https://api.nasa.gov/planetary/apod?api_key=DEMO_KEY';
fetch(url)
  .then(response => response.json())
  .then(data => console.log(data));
```

Или можно использовать `fetch` в синтаксисе `async/await`:

```
const url = 'https://api.nasa.gov/planetary/apod?api_key=DEMO_KEY';

async function fetchRequest() {
    const response = await fetch(url);
    const data = await response.json();
    console.log(data);
}

fetchRequest();
```

Обсуждение

Fetch API позволяет отправлять данные на удаленный сервер и получать их оттуда. При работе в среде веб-браузера это означает возможность получения данных без обновления страницы. Вам, как пользователю, наверняка часто встречались такие запросы. С помощью Fetch API можно делать следующее:

- загружать новые записи в ленте соцсети;
- выводить подсказки при автозаполнении формы;
- ставить лайки под постами в соцсети;
- изменять значения в полях формы в зависимости от ранее полученных данных;
- отправлять данные формы, не покидая страницы;
- помещать товары в корзину онлайн-магазина.

Как вы, вероятно, догадываетесь, этот список можно продолжать долго.

Метод `fetch()` принимает следующие два параметра:

- `url` (обязательный) — URL, на который отправляется запрос;
- `options` — объект с параметрами запроса.

В число возможных параметров входят следующие:

- `body` — тело запроса;
- `cache` — режим кэширования запроса (принимает значения `default`, `no-store`, `reload`, `no-cache`, `force-cache` или `only-if-cached`);
- `credentials` — требование учетных данных запроса (принимает значения `omit`, `same-origin` или `include`);
- `headers` — заголовки запроса;
- `integrity` — значение целостности дочернего ресурса, используемое для верификации ресурсов;

- `keepalive` — если нужно, чтобы запрос действовал после закрытия страницы, этому параметру присваивается значение `true`;
- `method` — метод запроса (`GET`, `POST`, `PUT` или `DELETE`);
- `mode` — режим запроса (`cors`, `no-cors` или `same-origin`);
- `redirect` — правила перенаправления (`follow`, `error` или `manual`);
- `referrer` — значение заголовка `referrer` (`about:client`, текущий URL или пустая строка);
- `referrerPolicy` — политика по отношению к заголовку `referrer` (`no-referrer`, `no-referrer-when-downgrade`, `same-origin`, `origin`, `strict-origin`, `origin-when-cross-origin`, `strict-origin-when-cross-origin` или `unsafe-url`);
- `signal` — объект `AbortController`, позволяющий прервать обработку запроса.

Как показано в предыдущем примере, обязательным является только параметр `url`. Если в метод `fetch` передан только URL, то выполняется запрос `GET`. В следующем примере показано использование объекта `options`:

```
const response = await fetch(url, {  
  method: 'GET',  
  mode: 'cors',  
  credentials: 'omit',  
  redirect: 'follow',  
  referrerPolicy: 'no-referrer'  
});
```

В методе `fetch` используются промисы JavaScript. Вначале промис возвращает объект `Response`. Он содержит HTTP-ответ с телом, заголовками, кодом состояния, информацией о перенаправлении, типом `CORS` и URL. Когда метод вернет запрос, можно будет применить к телу запроса дополнительные методы синтаксического анализа. В данном примере я использовал метод `json()` для синтаксического анализа тела запроса в формате JSON. Есть и другие методы:

- `arrayBuffer()` — синтаксический анализ тела запроса, представленного в виде `ArrayBuffer`;
- `blob()` — синтаксический анализ тела запроса, представленного как `Blob`;
- `json()` — синтаксический анализ тела запроса, представленного в формате JSON;
- `text()` — синтаксический анализ тела запроса, представленного в виде строки в формате UTF-8;
- `formData()` — синтаксический анализ тела запроса, представленного в виде объекта `FormData()`.

С помощью метода `fetch()` можно построить обработку ошибок на основе состояния ответа, возвращенного сервером. При использовании синтаксиса `async/await` это выглядит так:

```
async function fetchRequestWithError() {
  const response = await fetch(url);
  if (response.status >= 200 && response.status < 400) {
    const data = await response.json();
    console.log(data);
  } else {
    // Обрабатываем ошибку, полученную от сервера
    // Например: INTERNAL SERVER ERROR: 500 error
    console.log(`${response.statusText}: ${response.status} error`);
  }
}
```

Для более надежной обработки ошибок можно обернуть запрос `fetch` в блок `try/catch`, который позволяет обрабатывать дополнительные ошибки:

```
async function fetchRequestWithError() {
  try {
    const response = await fetch(url);
    if (response.status >= 200 && response.status < 400) {
      const data = await response.json();
      console.log(data);
    } else {
      // Обрабатываем ошибку, полученную от сервера
      // Например: INTERNAL SERVER ERROR: 500 error
      console.log(`${response.statusText}: ${response.status} error`);
    }
  } catch (error) {
    // Обработчик остальных ошибок
    console.log(error);
  }
}
```

Аналогичным образом можно обрабатывать ошибки, используя синтаксис промисов JavaScript:

```
fetch(url)
  .then((response) => {
    if (response.status >= 200 && response.status < 400) {
      return response.json();
    } else {
      // Обрабатываем ошибку, полученную от сервера
      // Например: INTERNAL SERVER ERROR: 500 error
      console.log(`${response.statusText}: ${response.status} error`);
    }
  })
  .then((data) => {
    console.log(data)
  })
  .catch(error) => {
    // Обработчик остальных ошибок
    console.log(error);
  };
};
```

Если вам уже приходилось иметь дело с AJAX, то вы могли использовать метод XMLHttpRequest (XHR), описанный в рецепте 13.2. Сейчас для таких запросов рекомендуется задействовать Fetch API из-за его более простого синтаксиса на основе промисов и широкой поддержки в браузерах (Chrome, Edge, Firefox, Safari, но не в Internet Explorer). Если приложение должно работать в старых версиях Internet Explorer, то можно применять XHR (XMLHttpRequest) или полизаполнение для fetch (<https://github.com/github/fetch>) и промисов (<https://github.com/taylorhakes/promise-polyfill>).

13.2. Использование метода XMLHttpRequest

Задача

Приложение должно запрашивать удаленные данные и поддерживать старые браузеры.

Решение

Вместо fetch использовать XMLHttpRequest (XHR). Далее показан XHR-запрос GET, соответствующий примеру, описанному в рецепте 13.1:

```
const url = 'https://api.nasa.gov/planetary/apod?api_key=DEMO_KEY';
const request = new XMLHttpRequest();
request.open('GET', url);
request.send();

request.onload = () => {
  if (request.status >= 200 && request.status < 400) {
    // Запрос успешный, выводим в консоль возвращенные данные JSON
    const data = JSON.parse(request.response);
    console.log(data);
  } else {
    // Ошибка сервера
    // Пример: INTERNAL SERVER ERROR: 500 error
    console.log(`${request.statusText}: ${request.status} error`);
  }
};

// Ошибка запроса
request.onerror = () => console.log(request.statusText);
```

Обсуждение

Изначально для удаленных запросов задействовался синтаксис XMLHttpRequest. Несмотря на наличие в его названии аббревиатуры XML, его можно применять ко всем видам данных. В предыдущем примере показан запрос данных JSON. Так чем же XMLHttpRequest отличается от fetch?

- В методе `fetch` активно используются промисы JavaScript, а `XMLHttpRequest` основан на конструкторе `XMLHttpRequest()`.
- `XMLHttpRequest()` поддерживается всеми браузерами, включая старые версии Internet Explorer. Метод `fetch` не работает с полизаполнениями (в основе которых лежит `XMLHttpRequest`) в Internet Explorer 11 и более ранних версиях, а также с некоторыми версиями современных автоматически обновляемых браузеров 2017 года и более ранних.
- `XMLHttpRequest` по умолчанию при каждом запросе отправляет на сервер файлы `cookie`, а `fetch` требует явно устанавливать все данные учетных записей.
- `XMLHttpRequest` может отслеживать процесс загрузки на сервер, в то время как метод `fetch` на момент написания этой книги поддерживал только процесс скачивания данных.
- Метод `fetch` не поддерживает задержки, так что размер запроса определяется браузером пользователя.

В остальной части этой главы будет использоваться современный синтаксис `fetch`, но `XMLHttpRequest` по-прежнему имеет смысл применять, особенно в старых приложениях, так как он поддерживается всеми браузерами.

13.3. Отправка данных формы

Задача

Передать данные формы от клиента на сервер.

Решение

Создать запрос POST для объекта `FormData`, используя `fetch`:

```
const myForm = document.getElementById('my-form');
const url = 'http://localhost:8080/';

myForm.addEventListener('submit', async event => {
  event.preventDefault();

  const formData = new FormData(myForm);
  const response = await fetch(url, {
    method: 'post',
    body: formData
  });

  const result = await response.text();
  alert(result);
});
```

Обсуждение

В приведенном коде я выбрал HTML-элемент формы с помощью метода `getElementById`, а URL для запроса POST сохранил в переменной. В данном случае запрос POST с данными формы передается на локальный сервер разработки, как показано в примере 13.1. Затем я привязал к форме обработчик события и отключил стандартную отправку формы, чтобы вместо нее выполнялся запрос POST на JavaScript с использованием `fetch`.

Полностью разметка HTML и код JavaScript выглядят так:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <meta http-equiv="X-UA-Compatible" content="ie=edge" />
    <title>Form POST</title>
  </head>
  <body>
    <h1>Form POST HTML</h1>

    <form id="my-form">
      <label for="name">Name:</label>
      <input type="text" id="name" name="name" />

      <label for="mail">E-mail:</label>
      <input type="email" id="mail" name="email" />

      <label for="msg">Message:</label>
      <textarea id="message" name="message"></textarea>

      <button>Submit</button>
    </form>

    <script>
      const myForm = document.getElementById('my-form');
      const url = 'http://localhost:8080/';

      myForm.addEventListener('submit', async event => {
        event.preventDefault();

        const formData = new FormData(myForm);
        const response = await fetch(url, {
          method: 'post',
          body: formData
        });

        const result = await response.text();
        alert(result);
      });
    </script>
  </body>
</html>
```

Объект `FormData` позволяет легко создавать на JavaScript пары «ключ — значение» для любых данных формы. Он обрабатывает не только текстовые элементы, как показано в примере, но и поля загрузки файлов. Для этого мы вначале использовали конструктор `FormData`:

```
const myForm = document.getElementById('my-form');
const formData = new FormData(myForm);
```

Есть еще несколько полезных методов для манипулирования данными формы, которые содержатся в `FormData`:

- `FormData.append(key, value)` или `FormData.append(key, blob, filename)` — добавляет данные в форму;
- `FormData.delete(key)` — удаляет поле формы;
- `FormData.set(key, value)` — добавляет данные, при необходимости удаляя дублирующиеся ключи.

Вот как можно было бы добавить новое поле в форму из предыдущего примера:

```
const myForm = document.getElementById('my-form');
const url = 'http://localhost:8080/';

myForm.addEventListener('submit', async event => {
  event.preventDefault();

  const formData = new FormData(myForm);
  // С помощью FormData.append добавляем новое поле
  formData.append('user', true);

  const response = await fetch(url, {
    method: 'post',
    body: formData
  });

  const result = await response.text();
  console.log(result);
});
```

Теперь тело POST-запроса будет выглядеть так:

```
{
  name: 'Adam',
  email: 'adam@example.com',
  message: 'Hello',
  user: 'true'
}
```

Для работы со значениями полей форм можно использовать также методы `get` и `has`:

- `FormData.get(key)` — получает значение для заданного ключа;

- `FormData.has(key)` — проверяет, есть ли значение у заданного ключа, и возвращает `true` или `false`.

Объект `FormData` очень полезен, но это не единственный тип значений для тела POST-запросов. В POST-запросе можно также передавать следующие типы данных:

- строки;
- строки определенного формата, такие как JSON и XML;
- объекты `URLSearchParams`;
- двоичные данные `Blob` и `BufferSource`.

В рецепте 13.4 будет показано, как передать данные JSON в POST-запросе с использованием `fetch`.

Наконец, в примере 13.1 показана обработка запроса сервером Express на основе Node.js.

Пример 13.1. Обработка формы на сервере Express

```
const express = require('express');
const formidable = require('formidable');
const cors = require('cors');

const app = express();
const port = 8080;

app.use(cors());

app.get('/', (req, res) => {
  res.send('Example server for receiving JS POST requests')
});

app.post('/', (req, res) => {
  const form = formidable();

  form.parse(req, (err, fields) => {
    if (err) {
      return;
    }
    console.log('POST body:', fields);
    res.sendStatus(200);
  });
});

app.listen(port, () => {
  console.log(`Example app listening at http://localhost:${port}`)
});
```



Сервер Express подробно рассматривается в главе 21.

13.4. Заполнение списка выбора данными, полученными с сервера

Задача

Заполнить список выбора значениями, которые зависят от действий пользователя, выполненных с другим элементом формы.

Решение

Перехватить событие `change` от элемента формы:

```
const niceThings = document.getElementById('nice-thing');
niceThings.addEventListener('change', async () => {
  // GET-запрос и соответствующие дальнейшие действия
});
```

В функции обработки запроса нужно сделать POST-запрос посредством `fetch`, передав данные формы в формате JSON:

```
const niceThings = document.getElementById('nice-thing');
const url = 'http://localhost:8080/select';

// При изменении значения элемента select выполняем GET-запрос
niceThings.addEventListener('change', async () => {
  // Объект, в котром содержится значение select
  const selection = {
    niceThing: niceThings.value
  };

  // GET-запрос к серверу
  const response = await fetch(url, {
    method: 'post',
    headers: {
      'Content-Type': 'application/json;charset=utf-8'
    },
    body: JSON.stringify(selection)
  });
});
```

Затем получаем результат и заполняем список с вариантами выбора:

```
const select = document.getElementById('nicestuff');

if (response.ok) {
  const result = await response.json();
  // Удаляем содержимое элемента select
  select.length = 0;
  // Вставляем вариант, отображаемый
  // по умолчанию, — с текстом, но без значения
  select.options[0] = new Option('--Please choose an option--', '');
}
```



```
// Заполняем элемент select значениями, полученными с сервера
for (let i = 0; i < result.length; i += 1) {
    select.options[select.length] = new Option(result[i], result[i]);
}
// Выводим элемент select
select.style.display = 'block';
} else {
    // В случае проблем с получением данных выводим сообщение об ошибке
    alert('Error');
}
```

```
    }
  </style>
</head>
<body>
  <h1>Select List</h1>

  <form id="my-form">
    <label for="pet-select">Select a nice thing:</label>

    <select name="nicething" id="nice-thing">
      <option value="">--Please choose an option--</option>
      <option value="birds">Birds</option>
      <option value="flowers">Flowers</option>
      <option value="sweets">Sweets</option>
      <option value="critters">Cute Critters</option>
    </select>
    <select id="nicestuff">
      <option value="">--Please choose an option--</option>
    </select>
  </form>
  <script>
const niceThings = document.getElementById('nice-thing');
const select = document.getElementById('nicestuff');
const url = 'http://localhost:8080/select';

// При изменении значения select выполняем GET-запрос
niceThings.addEventListener('change', async () => {
// Объект, в котором содержится значение select
const selection = {
  niceThing: niceThings.value
};
// GET-запрос на сервер
const response = await fetch(url, {
  method: 'post',
  headers: {
    'Content-Type': 'application/json;charset=utf-8'
  },
  body: JSON.stringify(selection)
});

// Если fetch-запрос выполнен успешно
if (response.ok) {
  const result = await response.json();
  // Удаляем старое содержимое элемента select
  select.length = 0;
  // Вставляем вариант выбора, который выводится
  // по умолчанию, — с текстом, но без значения
  select.options[0] = new Option('--Please choose an option--', '');
  // Заполняем элемент select данными,
  // полученными в ответ на запрос
  for (let i = 0; i < result.length; i += 1) {
    select.options[select.length] = new Option(result[i], result[i]);
  }
  // Выводим элемент select
```

```

        select.style.display = 'block';
    } else {
        // В случае проблем с получением данных
        // выводим сообщение об ошибке
        alert('Error');
    }
});

</script>
</body>
</html>

```

В этом примере для заполнения элемента `select` использовано приложение Node, но серверная часть может быть написана на любом другом языке программирования (подробнее о Node читайте в части III):

```

const express = require('express');
const formidable = require('formidable');
const cors = require('cors');

const app = express();
const port = 8080;

app.use(cors());

app.get('/', (req, res) =>
    res.send('Example server for receiving JS POST requests')
);

app.post('/select', (req, res) => {
    const form = formidable();
    form.parse(req, (err, fields) => {
        if (err) {
            return;
        }
        if (fields.niceThing === 'critters') {
            res.send(['puppies', 'kittens', 'guinea pigs']);
        } else if (fields.niceThing === 'sweets') {
            res.send(['licorice', 'cake', 'cookies', 'custard']);
        } else if (fields.niceThing === 'birds') {
            res.send(['robin', 'mockingbird', 'finch', 'dove']);
        } else if (fields.niceThing === 'flowers') {
            res.send(['roses', 'lilys', 'daffodils', 'pansies']);
        } else {
            res.send(['No Nice Things Found']);
        }
    });
});

app.listen(port, () =>
    console.log(`Example app listening at http://localhost:${port}`)
);

```

Не все приложения нуждаются в последовательном построении элементов форм. Но это отличный способ создавать более эффективные формы, когда данные могут изменяться по мере заполнения формы или она слишком сложна.

13.5. Синтаксический анализ данных, полученных в формате JSON

Задача

Надежно преобразовывать данные в формате JSON в объекты JavaScript. При этом заменять числовое представление `true` и `false` (1 и 0 соответственно) на аналоги типа `Boolean` (`true` и `false`).

Решение

Выполнить синтаксический анализ объекта с помощью `JSON.parse`. Для преобразования числовых значений в тип `Boolean` использовать следующую функцию восстановления:

```
const jsonobj = '{"test" : "value1", "test2" : 3.44, "test3" : 0}';
const obj = JSON.parse(jsonobj, (key, value) => {
  if (typeof value === 'number') {
    if (value === 0) {
      value = false;
    } else if (value === 1) {
      value = true;
    }
  }
  return value;
});
console.log(obj.test3); // false
```

Обсуждение

Чтобы понять, из чего состоит запись в формате JSON, представьте себе литеральный объект, который просто преобразовали в строку (с некоторыми оговорками).

Предположим, что это массив:

```
const arr = new Array("one", "two", "three");
```

Тогда запись в формате JSON будет эквивалентна литеральному представлению массива:

```
["one", "two", "three"];
```

Обратите внимание на двойные кавычки — одинарные в JSON недопустимы.

Для объекта

```
const obj3 = {
  prop1 : "test",
  result : true,
  num : 5.44,
  name : "Joe",
  cts : [45,62,13]
};
```

запись JSON будет выглядеть так:

```
{"prop1":"test","result":true,"num":5.44,"name":"Joe","cts":[45,62,13]}
```

Обратите внимание на то, что в JSON имена свойств всегда заключены в кавычки, а значения заключаются в кавычки только в том случае, если это строки. Кроме того, если внутри объекта содержатся другие объекты, такие как массивы, то они также преобразуются в их эквиваленты формата JSON. Но методы в JSON недопустимы. Если в состав объекта входят методы, то возникает ошибка — JSON работает только с данными.

Статический объект JSON несложен: он содержит только два метода — `stringify()` и `parse()`. Метод `parse()` принимает два аргумента: строку в формате JSON и при необходимости функцию восстановления. Последняя принимает в качестве параметров пару «ключ — значение» и возвращает либо исходное значение, либо его модифицированный вариант.

В нашем примере строка в формате JSON описывает объект с тремя свойствами — строкой, числом и еще одним свойством, которое содержит числовое значение, однако на самом деле представляет собой значение типа `Boolean` в числовом представлении: `0` соответствует `false`, а `1` — `true`.

Для того чтобы преобразовать все нули и единицы в `false` и `true`, мы создали специальную функцию, которая передается в `JSON.parse()` в качестве второго аргумента. Она проверяет каждое свойство объекта. Если это число, то функция проверяет, чему оно равно — `0` или `1`. Если это `0`, то возвращается `false`, если `1` — `true`. В остальных случаях возвращается исходное значение.

Возможность преобразовывать входящие данные из формата JSON в объекты очень важна, особенно при обработке результата запросов AJAX и JSONP. Мы не всегда можем управлять тем, в каком виде внешний сервис возвращает данные.



У JSON есть ряд ограничений: строки должны быть заключены в двойные кавычки, в строках недопустимы шестнадцатеричные значения и символы табуляции.

13.6. Получение и синтаксический анализ данных в формате XML

Задача

Получить из удаленного источника XML-файл и выполнить синтаксический анализ его содержимого.

Решение

Использовать `fetch` в сочетании с `DomParser` API, который позволяет преобразовывать строку в формат XML.

Вначале нужно с помощью `fetch` создать запрос и получить XML-файл. В следующем примере я запрашиваю новостную ленту в формате XML с начальной страницы *New York Times*:

```
const url = 'https://rss.nytimes.com/services/xml/rss/nyt/HomePage.xml';

async function fetchAndParse() {
  const response = await fetch(url);
  const data = await response.text();
  console.log(data);
}

fetchAndParse();
```

Затем с помощью `DOMParser` выполняем синтаксический анализ полученной строки в формате XML и с помощью методов DOM получаем данные из документа:

```
const url = 'https://rss.nytimes.com/services/xml/rss/nyt/HomePage.xml';

async function fetchAndParse() {
  const response = await fetch(url);
  const data = await response.text();
  const parser = new DOMParser();
  const XMLDocument = parser.parseFromString(data, 'text/xml');
  console.log(XMLDocument);
}

fetchAndParse();
```

Обсуждение

При получении XML с помощью `fetch` возвращается документ в виде простого текста. Затем нужно использовать `DOMParser` API, чтобы можно было обращаться к этому документу и обрабатывать результаты.

`DOMParser` позволяет обращаться к данным в формате XML, задействуя методы запросов к DOM, такие как `getElementsByTagName`. `DOMParser` принимает два аргумента. Первый из них — это строка, которую нужно проанализировать, а второй — `mimeType`, который определяет тип документа. Параметр `mimeType` может принимать следующие значения:

- `text/html`;
- `text/xml`;
- `application/xml`;
- `application/xhtml+xml`;
- `image/svg+xml`.

В следующем примере XML-анализатор дополнен таким образом, чтобы с помощью селекторов DOM-запросов можно было выводить заголовки новых статей на веб-странице:

```
(async () => {
  const url = 'https://rss.nytimes.com/services/xml/rss/nyt/HomePage.xml';

  // Получаем и анализируем XML-документ
  async function fetchAndParse() {
    const response = await fetch(url);
    const data = await response.text();
    const parser = new DOMParser();
    const XMLDocument = parser.parseFromString(data, 'text/xml');
    return XMLDocument;
  }

  function displayTitles(xml) {
    // HTML-элемент, в котором будут выводиться результаты
    // В разметке содержится элемент ul с id, равным "results"
    const listElem = document.getElementById('results');
    // Получаем заголовки статей
    // Каждый заголовок обернут в тег <item>, внутри
    // которого находится тег <title>
    const titles = xml.querySelectorAll('item title');
    // Перебираем все заголовки, полученные из XML;
    // вставляем текст каждого в список HTML
    titles.forEach(title => {
      const listItem = document.createElement('li');
      listItem.innerText = title.textContent;
      listElem.appendChild(listItem);
    });
  }

  const xml = await fetchAndParse();
  displayTitles(xml);
})();
```

13.7. Передача двоичных данных и загрузка изображения

Задача

Получить с сервера изображение в виде двоичных данных.

Решение

Чтобы получить двоичные данные посредством запроса `fetch`, нужно присвоить ответу тип `blob` и затем обрабатывать полученные данные соответствующим образом. Вот как можно преобразовать данные в изображение и загрузить их в элемент `img`:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <meta http-equiv="X-UA-Compatible" content="ie=edge" />
    <title>Binary Data</title>
  </head>
  <body>
    <h1>Binary Data</h1>

    <img id="result" />
    <script>
      async function fetchImage() {
        const url = 'logo.png';
        const response = await fetch(url);
        const blob = await response.blob();

        // Вставляем возвращенный url в элемент img
        const img = document.getElementById('result');
        img.src = URL.createObjectURL(blob);
      }
      fetchImage();
    </script>
  </body>
</html>
```

Обсуждение

Одним из преимуществ спецификации CORS является поддержка `fetch`-запросов для двоичных данных, которые называют также *типизированными массивами*. Главное требование, которое предъявляется к запросам двоичных данных, — определить тип запроса. Он может быть следующим:

- `arraybuffer` — буфер фиксированной длины для сырых двоичных данных;
- `blob` — файлоподобные неизменяемые сырые данные.

В нашем примере для преобразования данных типа `blob` в `DOMString` (которому обычно соответствует тип `String` в JavaScript) я использовал метод `URL.createObjectURL()`, передав туда URL объекта. Этот URL затем присваивается свойству `src` элемента `img`.

Разумеется, было бы гораздо проще взять URL PNG-файла и сразу присвоить его атрибуту `src`. Но в некоторых технологиях, таких как Web Workers и WebGL, необходимо иметь возможность оперировать двоичными данными.

13.8. Обмен HTTP cookies между несколькими доменами

Задача

Получить доступ к ресурсу из другого домена по запросу с *учетными данными*, включая данные HTTP cookies и остальную информацию для аутентификации.

Решение

Для обработки запросов с учетными данными необходимо внести изменения и в клиентскую, и в серверную части приложения. В следующем примере клиентское приложение обслуживается по адресу `somedomain.com`, а серверное — по адресу `api.example.com`. Поскольку это разные домены, то по умолчанию запросы с учетными данными не будут передаваться от клиента серверу.

На стороне клиента нужно указать в запросе `fetch` свойство `credentials`:

```
fetch('https://api.example.com', {
  credentials: "include"
})
```

На стороне сервера нужно присвоить заголовку `Access-Control-Allow-Credentials` значение `true`:

```
const http = require('http');
const Cookies = require('cookies');

const server = http.createServer((req, res) => {
  // Присваиваем значения заголовкам CORS
  res.setHeader('Content-type', 'text/plain');
  res.setHeader('Access-Control-Allow-Origin', 'https://somedomain.com');
  res.setHeader('Access-Control-Allow-Credentials', true);

  const cookies = new Cookies (req, res);
  cookies.set("apple", "red");

  res.writeHead(200);
```

```
    res.end("Hello cross-domain");  
  });  
  server.listen(8080);
```



Для сервера Express я рекомендую использовать промежуточное программное обеспечение CORS (<https://oreil.ly/vNPPC>). Express подробно рассматривается в главе 21.

Обсуждение

Технология совместного доступа к информации для нескольких доменов называется CORS (Cross-Origin Resource Sharing — совместное использование ресурсов разными источниками). По соображениям безопасности браузеры ограничивают возможность обмена данными, такими как файлы cookie и заголовки с учетными данными, между разными доменами. Для передачи HTTP cookies и заголовков аутентификации между доменами необходимо настроить расширение CORS — при условии, что и клиент, и сервер согласятся на передачу этих данных.

Если на стороне клиента применяется не `fetch`, а `XMLHttpRequest`, то необходимо указать значение свойства `withCredentials`:

```
const request = new XMLHttpRequest();  
  
request.onreadystatechange = function() {  
  if (this.readyState == 4) {  
    console.log(this.status);  
    if (this.status == 200) {  
      document.getElementById('result').innerHTML = this.responseText;  
    }  
  }  
};  
request.open('GET', 'http://localhost:8080/');  
request.withCredentials = true;  
request.send(null);
```

13.9. Двухнаправленный обмен данными между клиентом и сервером посредством WebSockets

Задача

Организовать двухнаправленный обмен данными между сервером и клиентской веб-страницей в режиме реального времени.

Решение

WebSockets позволяет установить двухнаправленный обмен данными между клиентом и сервером. Для этого клиент создает объект `WebSockets`, который

передает URI на сервер WebSockets. Обратите внимание на то, что при этом вместо протокола `http` или `https` используется протокол `ws`:. Когда клиент получает сообщение, он преобразует его текст в объект, извлекает оттуда числовой счетчик, увеличивает его на единицу и затем применяет счетчик в строковом значении объекта.

В следующем примере клиент выводит все числа, начиная с 2. Для того чтобы и на стороне клиента, и на стороне сервера сохранялось одно и то же состояние, в сообщении передается строка, которую нужно вывести:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <meta http-equiv="X-UA-Compatible" content="ie=edge" />
    <title>Using Websockets</title>
  </head>
  <body>
    <h1>Using Websockets</h1>

    <div id="output"></div>
    <script type="text/javascript">
      const socket = new WebSocket('ws://localhost:8080');
      socket.onmessage = event => {
        const msg = JSON.parse(event.data);
        msg.counter = Number(msg.counter) + 1;
        msg.strng += `${msg.counter}-`;
        const html = `<p> ${msg.strng} </p>`;
        document.getElementById('output').innerHTML = html;
        socket.send(JSON.stringify(msg));
      };
    </script>
  </body>
</html>
```

В качестве сервера я использовал модуль Node. Сразу после создания сервер начинает обмен данными с клиентом, отправляя туда объект JavaScript, состоящий из двух свойств: числового счетчика и строки. Перед отправкой объект должен быть преобразован в строку. Код перехватывает входящие сообщения и событие `close`. При поступлении входящего сообщения счетчик увеличивается на единицу и объект передается на сервер:

```
var wsServer = require('ws').Server;
var wss = new wsServer({port:8001});
wss.on('connection', (function (conn) {

  // Объект, который передается от сервера к клиенту и обратно
  var counter = {counter: 1, strng: ''};

  // Передаем первые данные клиенту
  conn.send(JSON.stringify(counter));
```

```
// В ответ возвращаем данные на сервер
conn.on('message', function(message) {
    var ct = JSON.parse(message);
    ct.counter = parseInt(ct.counter) + 1;
    if (ct.counter < 100) {
        conn.send(JSON.stringify(ct));
    }
});
});
```

Обсуждение

Двунаправленный обмен данными, также называемый *полнодуплексным*, — это такой обмен данными, при котором передача может осуществляться одновременно в обоих направлениях. Такой обмен данными можно представить в виде двухполосной дороги, по которой в обоих направлениях движутся машины. Все современные браузеры поддерживают спецификацию WebSockets, которой, как вы увидите, исключительно легко пользоваться.

Кроме простоты применения в браузерах, у WebSockets есть еще одно преимущество — способность преодолевать прокси-серверы и брандмауэры. С этой задачей другие двунаправленные технологии обмена данными, такие как длинный опрос (long polling), справляются с трудом либо не справляются вовсе. А чтобы гарантировать безопасность приложения, программы на стороне клиента, такие как Chrome и Firefox, запрещают передавать содержимое смешанного типа, то есть использовать и HTTP, и HTTPS.

WebSockets позволяет передавать как текст, так и двоичные данные. И как показано в примерах, можно передавать данные в формате JSON, вызвав для объекта перед передачей метод `JSON.stringify()`, а потом на стороне приемника вызвав `JSON.parse()` для анализа полученной строки.

Читайте также

Подробнее о WebSockets читайте на соответствующем сайте — <https://www.websocket.org>.

13.10. Длинный опрос удаленного источника данных

Задача

Сделать так, чтобы соединение с сервером оставалось открытым и клиент мог сразу получать новую информацию, — но без WebSockets.

Решение

Использовать длинный опрос — технологию, при которой клиент сохраняет соединение с сервером с помощью асинхронной функции `fetch`, которая, получив ответ на запрос, вызывает сама себя. В простейшем случае длинный опрос на стороне клиента выглядит так:

```
const url = 'http://localhost:8080/';

async function longPoll() {
  const response = await fetch(url);
  // При получении сообщения ответ выводится
  // в консоль и вызывается функция опроса
  const message = await response.text();
  console.log(message);
  await longPoll();
}

longPoll();
```

Этот код можно улучшить, добавив обработку ошибок, чтобы при получении сообщения об ошибке приложение подождало какое-то время и повторило попытку опроса сервера:

```
const url = 'http://localhost:8080/';

async function longPoll() {
  try {
    // Если сообщение получено, то выводим ответ
    // в консоль и вызываем функцию опроса
    const response = await fetch(url);
    const message = await response.text();
    console.log(message);
    await longPoll();
  } catch (error) {
    // Если fetch возвращает ошибку, то ждем 1 с и повторяем попытку
    console.log(`Request failed ${error}`);
    await new Promise(resolve => setTimeout(resolve, 1000));
    await longPoll();
  }
}

longPoll();
```

Обсуждение

Для длинного опроса сервера нужно отправить запрос на сервер и поддерживать соединение до тех пор, пока он не вернет ответ. Получив ответ, клиент сразу устанавливает новое соединение с сервером и ожидает нового запроса. Этот процесс можно разделить на следующие этапы.

1. Клиент отправляет запрос на сервер.
2. Клиент поддерживает соединение, ожидая ответа от сервера.
3. Сервер отправляет ответ клиенту.
4. Клиент снова устанавливает соединение с сервером, и все повторяется сначала.

Принцип длинного опроса легко проиллюстрировать на примере чата. Представьте себе программу чата, в котором два пользователя обмениваются сообщениями. Пусть их зовут Райли и Харлоу. Оба они подключены к серверу. Когда Райли отправляет сообщение, сервер передает ответ браузеру Харлоу. В ответ браузер сразу подключается к серверу и ожидает следующего сообщения.

Ограничение длинных опросов состоит в количестве открытых соединений, которые может поддерживать сервер. Node способен поддерживать много конкурентных соединений, но у других языков есть ограничения. И все языки ограничены возможностями аппаратных средств самого сервера. Таким образом, несмотря на то что длинные опросы — простой и эффективный способ установки соединения, WebSockets, описанные в рецепте 13.9, лучше подходят для двунаправленного обмена данными между клиентом и сервером.

Сохранение данных

Что бы мы ни программировали — анимацию, взаимодействие, трансляцию, игру или рендеринг, — мы всегда имеем дело с данными. Это тот фундамент, на котором строится большинство приложений JavaScript. В первой части книги мы изучили стандартные типы данных для языка JavaScript. В главе 13 научились получать данные из удаленного источника, в главе 20 будем работать с данными, размещенными на сервере, и с источниками данных, а также научимся управлять данными посредством API. В общем, данные и JavaScript — друзья навеки.

В этой главе мы рассмотрим способы сохранения данных в браузере средствами JavaScript, используя cookies, а также объекты `sessionStorage`, `localStorage` и `IndexedDB`.

14.1. Сохранение информации в cookies

Задача

Прочитать из cookie браузера несколько значений или записать значения в cookie.

Решение

Для чтения и записи значений cookie используется объект `document.cookie`:

```
document.cookie = 'author=Adam';  
console.log(document.cookie);
```

Для того чтобы закодировать строку, применяется функция `encodeURIComponent` — она удаляет из строки все запятые, точки с запятой и пробелы:

```
const book = encodeURIComponent('JavaScript Cookbook');  
document.cookie = `title=${book}`;  
console.log(document.cookie);
```

```
// В консоль будет выведено: title=JavaScript%20Cookbook
```

В конце значения cookie можно добавить необязательные параметры, которые должны разделяться точкой с запятой:

```
document.cookie = 'user=Abigail; max-age=86400; path=/';
```

Чтобы удалить cookie, нужно присвоить уже созданному cookie срок окончания действия:

```
function eraseCookie(key) {  
    const cookie = `${key}=;expires=Thu, 01 Jan 1970 00:00:00 UTC`;  
    document.cookie = cookie;  
}
```

Обсуждение

Значения cookies — это маленькие элементы данных, которые хранятся в браузере. Cookies часто передаются из серверного приложения и отправляются на сервер практически с каждым запросом. В браузере доступ к cookies получают с помощью объекта `document.cookie`.

Значениям cookies можно присваивать следующие дополнительные параметры, разделяя их точкой с запятой:

- **domain** — домен, в котором доступно данное значение cookie. Если этот параметр не определен, то по умолчанию он равен домену, в котором размещен текущий хост. Если задать конкретный домен, то значение cookie будет доступно также в поддоменах этого домена;
- **expires** — срок истечения действия данного cookie. Принимает значения в формате даты GMTString;
- **max-age** — период времени в секундах, в течение которого действительно данное значение cookie;
- **path** — путь, по которому доступно значение cookie (например, / или /app). Если этот параметр не определен, то по умолчанию данное значение cookie доступно по текущему пути;
- **secure** — если этот параметр равен `true`, то данное значение cookie будет передаваться только по протоколу `https`;
- **samesite** — по умолчанию данный параметр равен `strict`. В этом случае cookie не передается между разными сайтами. Если же этот параметр равен `lax`, то cookie будет передаваться при запросах GET верхнего уровня.

В следующем примере пользователь может ввести значение, которое сохраняется в cookie. Затем это значение можно получить, указав его ключ, или удалить.

HTML-файл выглядит так:


```

<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <meta http-equiv="X-UA-Compatible" content="ie=edge" />
    <style>
      div {
        margin: 10px;
      }

      .data {
        width: 200px;
        background-color: yellow;
        padding: 5px;
      }
    </style>
    <title>Store, retrieve, and delete a cookie</title>
  </head>
  <body>
    <h1>Store, retrieve, and delete a cookie</h1>

    <form>
      <div>
        <label for="key"> Enter key:</label>
        <input type="text" id="key" />
      </div>
      <div>
        <label for="value">Enter value:</label>
        <input type="text" id="value" />
      </div>
    </form>
    <button id="set">Set data</button>
    <button id="get">Get data</button>
    <button id="erase">Erase data</button>

    <p>Cookie value:</p>
    <div id="cookiestr" class="data"></div>

    <script src="cookie.js"></script>
  </body>
</html>

```

А так выглядит привязанный к HTML-странице файл `cookie.js`:

```

// Присваиваем значение cookie
function setData() {
  const formKey = document.getElementById('key').value;
  const formValue = document.getElementById('value').value;

  const cookieVal = `${formKey}=${encodeURIComponent(formValue)}`;
  document.cookie = cookieVal;
}

```

```
// Получаем значение cookie по заданному ключу
function getData() {
    const key = document.getElementById('key').value;
    const cookie = document.getElementById('cookiestr');
    cookie.innerHTML = '';

    const keyValue = key.replace(/([.*+?^=!:${}()|\[\]\/\\])/g, '\\$1');
    const regex = new RegExp(`(?:^|;)\s?${keyValue}=(.*?)(?:;|$)` , 'i');
    const match = document.cookie.match(regex);
    const value = (match && decodeURIComponent(match[1])) || '';
    cookie.innerHTML = `<p>${value}</p>`;
}

// Удаляем cookie по заданному ключу
function removeData() {
    const key = document.getElementById('key').value;
    document.getElementById('cookiestr').innerHTML = '';

    const cookie = `${key}=; expires=Thu, 01 Jan 1970 00:00:00 UTC`;
    document.cookie = cookie;
}

document.getElementById('set').onclick = setData;
document.getElementById('get').onclick = getData;
document.getElementById('erase').onclick = removeData;
```

Обратите внимание на то, что для выбора значений cookie я использовал регулярные выражения, закодированные с помощью функции `encodeURIComponent`. Это сделано потому, что `document.cookie` возвращает строку, содержащую все cookies. Благодаря регулярным выражениям я смог вычлениить оттуда нужную информацию. Регулярные выражения подробно рассматриваются в главе 2.

14.2. Хранение данных на стороне клиента с помощью `sessionStorage`

Задача

Нам нужен удобный способ сохранять информацию в течение всей сессии без ограничения размера и межстраничного загрязнения, свойственного cookies.

Решение

Использовать функцию `sessionStorage` из DOM Storage:

```
sessionStorage.setItem('name', 'Franco');
sessionStorage.city = 'Pittsburgh';

// Выводится 2
console.log(sessionStorage.length);
```

```
// Получаем отдельные значения
const name = sessionStorage.getItem('name');
const city = sessionStorage.getItem('city');
console.log(`The stored name is ${name}`);
console.log(`The stored city is ${city}`);

// Удаляем элемент из хранилища
sessionStorage.removeItem('name');

// Удаляем все элементы из хранилища
sessionStorage.clear();

// Выводится 0
console.log(sessionStorage.length);
```

Обсуждение

Объект `sessionStorage` позволяет легко сохранять информацию в браузере пользователя, где она будет находиться в течение сессии. Сессия длится до тех пор, пока открыта данная вкладка браузера. Когда пользователь закрывает эту вкладку или окно браузера, сессия завершается, а при открытии той же страницы на новой вкладке браузер создает новую сессию.

В отличие от `sessionStorage`, `cookies` и объект `localStorage` (описан в рецепте 14.3) по умолчанию предназначены для сохранения данных между сессиями. Для сравнения этих методов хранения данных рассмотрим пример 14.1, в котором показано сохранение информации из формы в `cookie`, `localStorage` и `sessionStorage`.

Пример 14.1. Сравнение `sessionStorage` и `cookies`

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <meta http-equiv="X-UA-Compatible" content="ie=edge" />
    <style>
      div {
        margin: 10px;
      }

      .data {
        width: 100px;
        background-color: yellow;
        padding: 5px;
      }
    </style>
    <title>Comparing Cookies, localStorage, and sessionStorage</title>
  </head>
  <body>
    <h1>Comparing Cookies, localStorage, and sessionStorage</h1>
```

```

    <form>
      <div>
        <label for="key"> Enter key:</label>
        <input type="text" id="key" />
      </div>
      <div>
        <label for="value">Enter value:</label>
        <input type="text" id="value" />
      </div>
    </form>
    <button id="set">Set data</button>
    <button id="get">Get data</button>
    <button id="erase">Erase data</button>

    <p>Session:</p>
    <div id="sessionstr" class="data"></div>
    <p>Local:</p>
    <div id="localstr" class="data"></div>
    <p>Cookie:</p>
    <div id="cookiestr" class="data"></div>

    <script src="cookie.js"></script>
    <script src="app.js"></script>
  </body>
</html>

```

В файле `cookies.js` содержится код, позволяющий присваивать, получать и удалять значение `cookie`:

```

// Создаем сессию cookie
function setCookie(cookie, value) {
  const cookieVal = `${cookie}=${encodeURIComponent(value)};path=/`;
  document.cookie = cookieVal;
  console.log(cookieVal);
}

// Все значения cookie разделяются точкой с запятой ;
function getCookie(key) {
  const keyValue = key.replace(/([.*+?^=!:${}()|\[\]\/\\])/g, '\\$1');
  const { cookie } = document;
  const regex = new RegExp(`(?:^|;)\s?${keyValue}=(.*?)(?:$|`), 'i');
  const match = cookie.match(regex);

  return match && decodeURIComponent(match[1]);
}

// Чтобы удалить значение cookie, присваиваем ему прошедшую дату
function eraseCookie(key) {
  const cookie = `${key}=;path=/; expires=Thu, 01 Jan 1970 00:00:00 UTC`;
  document.cookie = cookie;
  console.log(cookie);
}

```

Наконец, в файле `app.js` содержится остальной функционал приложения:

```
// Сохраняем данные в сессии и cookie
function setData() {
    const key = document.getElementById('key').value;
    const { value } = document.getElementById('value');

    // Сохраняем данные в sessionStorage
    sessionStorage.setItem(key, value);

    // Сохраняем данные в localStorage
    localStorage.setItem(key, value);

    // Сохраняем данные в cookie
    setCookie(key, value);
}

function getData() {
    try {
        const key = document.getElementById('key').value;
        const session = document.getElementById('sessionstr');
        const local = document.getElementById('localstr');
        const cookie = document.getElementById('cookiestr');

        // Обновляем отображение
        session.innerHTML = '';
        local.innerHTML = '';
        cookie.innerHTML = '';

        // sessionStorage
        let value = sessionStorage.getItem(key) || '';
        if (value) session.innerHTML = `<p>${value}</p>`;

        // localStorage
        value = localStorage.getItem(key) || '';
        if (value) local.innerHTML = `<p>${value}</p>`;

        // cookie
        value = getCookie(key) || '';
        if (value) cookie.innerHTML = `<p>${value}</p>`;
    } catch (e) {
        console.log(e);
    }
}

function removeData() {
    const key = document.getElementById('key').value;

    // sessionStorage
    sessionStorage.removeItem(key);

    // localStorage
    localStorage.removeItem(key);

    // cookie
    eraseCookie(key);

    // Обновляем отображение
```

```

    getData();
}

document.getElementById('set').onclick = setData;
document.getElementById('get').onclick = getData;
document.getElementById('erase').onclick = removeData;

```

Для того чтобы получать данные и заносить их в объект `sessionStorage`, можно обращаться к нему напрямую, как показано в примере, но лучше использовать функции `getItem()` и `setItem()`.

Теперь загрузите страницу примера, введите одно или несколько значений для одного и того же ключа и нажмите кнопку **Get data**. Результат показан на рис. 14.1 — все предсказуемо. Данные сохраняются в `cookies`, `localStorage` и `sessionStorage`. Теперь откройте ту же страницу в новой вкладке браузера, введите значение в поле формы `key` и снова нажмите кнопку **Get data**. В результате должна получиться страница, подобная показанной на рис. 14.2.

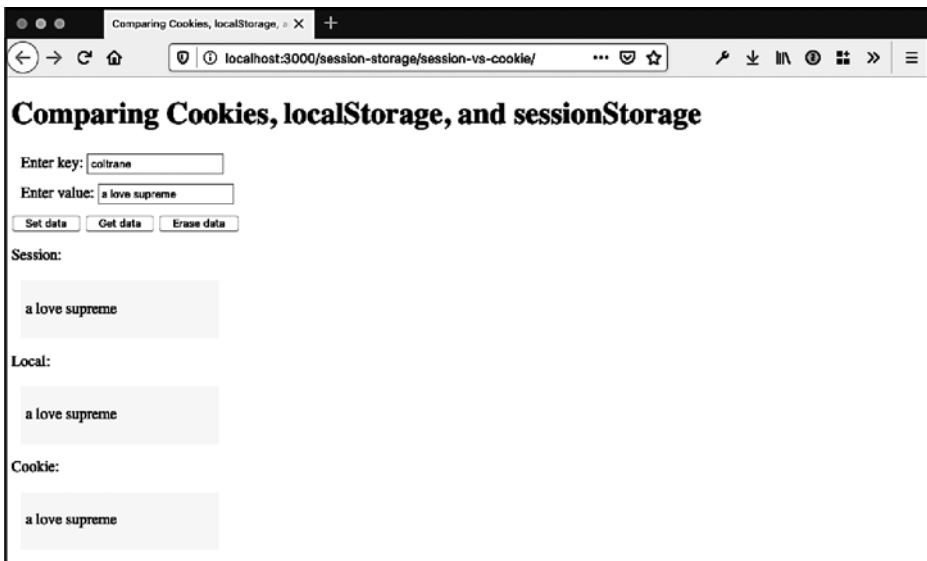


Рис. 14.1. Вывод данных, сохраненных в `sessionStorage` и `cookie`, на вкладке с формой

На новой вкладке значения `cookie` и `localStorage` сохраняются, так как `cookie` привязаны к сессии. Но значения `sessionStorage` не сохраняются, потому что они привязаны к вкладке браузера.

На этих снимках экрана, демонстрирующих разницу в сохранении данных на одной вкладке и на разных вкладках, видно одно из ключевых различий между `sessionStorage` и `cookies` помимо способа записи этих значений и доступа к ним в JavaScript. Я надеюсь, что на этих рисунках и в примере заметны также

потенциальные риски использования `sessionStorage`, особенно в тех случаях, когда обычно применяются `cookies`.

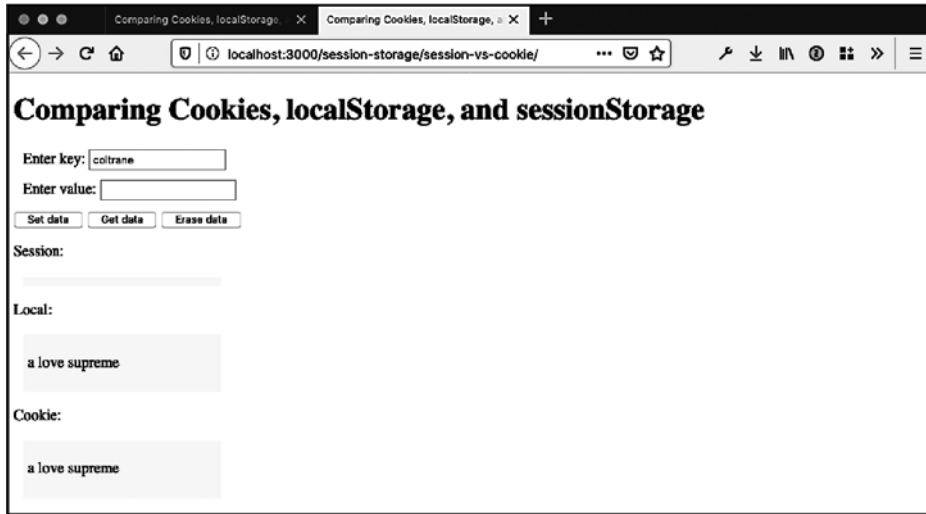


Рис. 14.2. Вывод данных, сохраненных в `sessionStorage` и `cookie`, на отдельной вкладке

Если пользователи сайта или приложения привыкли, что значения `cookie` сохраняются при переходе между вкладками, то `sessionStorage` может стать для них неприятным сюрпризом. Дело не только в изменившемся поведении — многие с неудовольствием обнаружат, что теперь эти значения нельзя удалить с помощью команды меню браузера, которая удаляет `cookies`, так как эта команда не влияет на `sessionStorage`. Зато объект `sessionStorage` необычайно прост и понятен — наконец-то у нас появился способ хранения данных, позволяющий привязать хранилище к одной определенной вкладке окна браузера.

Наконец, последнее замечание относительно объекта `sessionStorage` касается реализации: и `sessionStorage`, и описанный в следующем рецепте `localStorage` входят в спецификацию W3C DOM Storage. Оба они являются свойствами объекта `window`, а следовательно, доступны глобально. Оба они являются реализациями объекта `Storage`, так что изменение прототипа `Storage` приведет к изменению объектов `sessionStorage` и `localStorage`:

```
Storage.prototype.someMethod = function (param) { ...};
...
localStorage.someMethod(param);
...
sessionStorage.someMethod(param);
```

Помимо различий, описанных в этом и следующем рецептах, еще одна значительная особенность состоит в том, что объекты `Storage` не передаются на сервер

и обратно — эти технологии предназначены для хранения данных исключительно на стороне клиента.

Читайте также

Подробнее об объектах `Storage`, `sessionStorage`, `localStorage` и `Storage DOM` читайте в спецификации (<https://oreil.ly/PgBUt>). В рецепте 14.3 описан еще один способ записи и чтения `sessionStorage` и `localStorage`.

14.3. Создание хранилища данных на стороне клиента на основе `localStorage`

Задача

Сохранить данные, полученные при заполнении формы (или любые другие данные), таким образом, чтобы пользователи могли продолжать работать с ними с того места, на котором остановились в прошлый раз, в случае сбоя браузера, или если случайно закрыли окно, или прервалось соединение с интернетом.

Решение

Если данных немного, то мы могли бы использовать cookies. Но в случае разрыва соединения с интернетом эта стратегия не работает. Лучше поступить иначе — задействовать `localStorage`, особенно при сохранении большого объема данных или если нужно обеспечить функционирование приложения при отсутствии доступа к интернету:

```
const formValue = document.getElementById('formelem').value;
if (formValue) {
    localStorage.formelem = formValue;
}

// чтение данных
const storedValue = localStorage.formelem;
if (storedValue) {
    document.getElementById('formelem').value = storedValue;
}
```

Обсуждение

В рецепте 14.2 описывается `sessionStorage` — одна из технологий спецификации DOM Storage. У объекта `localStorage` практически такой же интерфейс, с теми же принципами записи данных:

```
// С помощью методов item
sessionStorage.setItem('key', 'value');
```



```
localStorage.setItem('key', 'value');

// Обращаясь напрямую к свойству по его имени
sessionStorage.keyName = 'value';
localStorage.keyName = 'value';

// С помощью метода key
sessionStorage.key(0) = 'value';
localStorage.key(0) = 'value';
```

Чтение данных выполняется так:

```
// С помощью методов item
value = sessionStorage.getItem('key');
value = localStorage.getItem('key');

// Обращаясь напрямую к свойству по его имени
value = sessionStorage.keyName;
value = localStorage.keyName;

// С помощью метода key
value = sessionStorage.key(0);
value = localStorage.key(0);
```

Как и `sessionStorage`, `localStorage` позволяет записывать и читать данные непосредственно, но лучше так не делать, а использовать методы `getItem()` и `setItem()`.

У обоих этих объектов хранения данных есть свойство `length`, в котором хранится количество сохраненных пар «ключ — значение», а также метод `clear()` (без параметров), который полностью освобождает хранилище. Кроме того, оба объекта привязаны к HTML5, а значит, сохраненные данные будут доступны на всех страницах домена, но только в пределах одного протокола (`http` или `https`) и порта.

Разница между этими двумя объектами состоит в длительности хранения данных. Объект `sessionStorage` позволяет сохранять данные только в пределах сессии, а в объекте `localStorage` они могут храниться сколь угодно долго, пока их специально не удалят.

Объекты `sessionStorage` и `localStorage` поддерживают одно и то же событие `storage`. Оно интересно тем, что возникает на всех страницах при изменении элемента `localStorage`. Это событие также относится к области низкой совместимости между браузерами: в Firefox его можно перехватывать в элементах `body` или `document`, в IE — только в `body`, а в Safari — только в `document`.

В примере 14.2 показана более полная реализация, чем вариант использования, описанный в рецепте. Здесь ко всем элементам небольшой формы привязан обработчик события `onchange`. Функция, назначенная ему, перехватывает имя и значение измененного элемента и сохраняет их в локальном хранилище с помощью `localStorage`. При отправке формы все сохраненные данные формы удаляются.

При загрузке страницы обработчик событий `onchange` для элементов формы привязывается к функции, которая сохраняет значения. Если значение с таким именем уже сохранено, то на его место записывается значение из элемента формы. Для того чтобы протестировать приложение, введите данные в несколько полей формы, но перед тем как нажимать кнопку **Submit**, обновите страницу. Если бы данные не сохранялись в `localStorage`, то при обновлении страницы они были бы потеряны. Но теперь при перезагрузке страницы форма возвращается к тому состоянию, в котором находилась перед обновлением страницы.

Пример 14.2. Сохранение состояния формы с помощью `localStorage` на случай перезагрузки страницы или сбоя браузера

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <meta http-equiv="X-UA-Compatible" content="ie=edge" />
    <title>Creating a localStorage Client-Side Data Storage Item</title>
  </head>
  <body>
    <h1>Creating a localStorage Client-Side Data Storage Item</h1>

    <form id="inputform">
      <div>
        <label for="field1">Enter field1:</label>
        <input type="text" id="field1" />
      </div>
      <div>
        <label for="field2">Enter field2:</label>
        <input type="text" id="field2" />
      </div>
      <div>
        <label for="field3">Enter field1:</label>
        <input type="text" id="field3" />
      </div>
      <div>
        <label for="field4">Enter field1:</label>
        <input type="text" id="field4" />
      </div>
      <input type="submit" value="Clear Storage" />
    </form>

    <script src="localStorage.js"></script>
  </body>
</html>
```

В файле JavaScript содержится следующий код:

```
// Сохраняем содержимое полей ввода формы в переменной
const elems = document.querySelectorAll('input');
```

```

// Сохраняем значения полей
function processField() {
    localStorage.setItem(window.location.href, 'true');
    localStorage.setItem(this.id, this.value);
}

// Очищаем поля по одному
function clearStored() {
    elems.forEach(elem => {
        if (elem.type === 'text') {
            localStorage.removeItem(elem.id);
        }
    });
}

// Перехватываем нажатие кнопки Submit и очищаем хранилище
document.getElementById('inputform').onsubmit = clearStored;

// При изменении элемента формы записываем его значение в localStorage
elems.forEach(elem => {
    if (elem.type === 'text') {
        const value = localStorage.getItem(elem.id);
        if (value) elem.value = value;

        // Событие change
        elem.onChange = processField;
    }
});

```

Объем памяти, выделяемый для `localStorage`, зависит от браузера, но обычно колеблется в диапазоне от 5 до 10 Мбайт. Чтобы проверить, не превышен ли лимит выделенной браузером памяти, можно использовать блок `try/catch`:

```

try {
    localStorage.setItem('key', 'value');
} catch (domException) {
    if (
        ['QuotaExceededError', 'NS_ERROR_DOM_QUOTA_REACHED'].includes(
            domException.name
        )
    ) {
        // Превышен размер файла; обрабатываем ошибку
    } else {
        // Какая-то другая ошибка; обрабатываем ее
    }
}

```

Объект `localStorage` можно задействовать в ходе работы без доступа к сети. В примере с формой можно сохранить данные в `localStorage` и предоставить пользователю кнопку, на которую он нажмет, когда появится доступ к интернету, чтобы синхронизировать данные, хранящиеся в `localStorage` и на сервере.

Читайте также

Подробнее об объекте `Storage`, а также объектах `sessionStorage` и `localStorage` читайте в рецепте 14.2.

14.4. Сохранение больших объемов данных на стороне клиента с помощью IndexedDB

Задача

Нам нужно более сложное хранилище данных на стороне клиента, чем то, что предоставляет `localStorage`.

Решение

В современных браузерах можно использовать IndexedDB.

В примере 14.3 показан файл с кодом JavaScript, в котором с помощью IndexedDB созданы база данных и объект данных. Здесь создается база данных, затем в нее записываются данные и считывается первый объект. Как именно это делается, подробно описано в разделе «Обсуждение».

Пример 14.3. Создание базы данных с помощью IndexedDB, запись в нее данных и считывание объекта данных

```
const data = [
  { name: 'Joe Brown', age: 53, experience: 5 },
  { name: 'Cindy Johnson', age: 44, experience: 5 },
  { name: 'Some Reader', age: 30, experience: 3 }
];

// Удаляем базу данных Cookbook, чтобы можно было
// запускать пример многократно
const delReq = indexedDB.deleteDatabase('Cookbook');
delReq.onerror = event => {
  console.log('delete error', event);
};

// Открываем базу данных Cookbook с номером версии 1
// Если такой базы не существует, создаем ее
const request = indexedDB.open('Cookbook', 1);

// При открытии базы данных, номер версии которой выше, чем у той,
// что была записана ранее, активируется событие upgradeneeded
// (в данном случае этого не происходит)
request.onupgradeneeded = event => {
  const db = event.target.result;
  const { transaction } = event.target;
```

```

// Создаем в базе данных новый объект-хранилище с именем reader
const objectStore = db.createObjectStore('reader', {
  keyPath: 'id',
  autoIncrement: true
});

// Создаем в объекте-хранилище новые ключи
objectStore.createIndex('experience', 'experience', { unique: false });
objectStore.createIndex('name', 'name', { unique: true });

// После того как будут загружены все данные,
// выводим сообщение в консоль
transaction.oncomplete = () => {
  console.log('data finished');
};

const readerObjectStore = transaction.objectStore('reader');

// Перебираем все значения объекта данных
// и записываем их в базу данных indexedDB
data.forEach(value => {
  const req = readerObjectStore.add(value);
  // После каждой успешной записи выводим сообщение в консоль
  req.onsuccess = () => {
    console.log('data added');
  };
});

// Если при запросе возникает ошибка, выводим в консоль сообщение о ней
request.onerror = () => {
  console.log(event.target.errorCode);
};

// После успешного создания хранилища данных выводим сообщение в консоль
request.onsuccess = () => {
  console.log('datastore created');
};

// Если пользователь щелкнет на странице, считываем
// из базы данных случайное число и выводим его в консоль
document.onclick = () => {
  const randomNum = Math.floor(Math.random() * 3) + 1;
  const dataRequest = db
    .transaction(['reader'])
    .objectStore('reader')
    .get(randomNum);
  dataRequest.onsuccess = () => {
    console.log(`Name : ${dataRequest.result.name}`);
  };
};
};

```

Обсуждение

IndexedDB — это спецификация, утвержденная W3C и другими заинтересованными сторонами для создания решений, позволяющих оперировать большими

объемами данных на стороне клиента. IndexedDB основана на принципе транзакций и поддерживает концепцию *курсора*, однако все же не является реляционной базой данных. IndexedDB работает с объектами JavaScript, каждый из которых индексируется по заданному *ключу*. Что именно представляет собой ключ, решаете вы.

База данных IndexedDB может быть как синхронной, так и асинхронной. Ее можно использовать для больших объемов данных на обычном сервере или в облачном приложении, она может быть полезной и при автономной работе веб-приложения.

В большинстве реализаций IndexedDB объем хранилища данных не ограничен. Но в Firefox, чтобы сохранить более 50 Мбайт, потребуется разрешение пользователя. В Chrome создается пул временного хранилища, из которого каждому приложению выделяется до 20 %. В остальных системах тоже есть похожие ограничения. IndexedDB поддерживается во всех основных браузерах, за исключением Opera Mini, хотя отдельные нюансы этой поддержки могут различаться.

Как показано в примере, методы IndexedDB API позволяют вызывать функции в случае успешного и неудачного завершения операции путем перехвата этих событий обычными обработчиками, или обратного вызова, или присвоения функции. В Mozilla приводится следующий алгоритм использования IndexedDB.

1. Открыть базу данных.
2. Создать хранилище объектов в обновляемой базе данных.
3. Открыть транзакцию и сделать запрос, чтобы выполнить некую операцию с базой данных, такую как запись или получение данных.
4. Дождаться завершения операции, перехватывая соответствующее событие DOM.
5. Сделать что-то с результатами, которые извлекаются из объекта, полученного в ответ на запрос.

В нашем примере сначала создается объект данных с тремя значениями. Впоследствии он будет записан в базу данных. Если база данных существовала ранее, то она удаляется, чтобы пример можно было выполнять многократно. Затем вызывается функция `open()`, которая открывает базу данных, если она существует, и создает ее, если такой базы еще нет. Поскольку перед выполнением примера база данных была удалена, то она создается заново. При этом важно указать и имя, и версию, так как база данных может быть изменена только в том случае, если открывается ее новая версия.

Из метода `open()` возвращается объект ответа на запрос (`IDBOpenDBRequest`). Именно он активирует события завершения операции — как успешного, так и неудачного. В нашем коде обработчик события `onsuccess` для данного объекта перехватывает это событие и выводит в консоль сообщение об успешном выполнении операции. В этом обработчике событий можно было бы присвоить ссылку на базу данных глобальной переменной, но в нашем коде это сделано в обработчике другого события — `upgradeneeded`.

Обработчик события `upgradeneeded` вызывается только в том случае, если не существует базы данных с такими именем и версией. Объект события также предоставляет нам доступ к ссылке на `IDBDatabase`, которая присваивается глобальной переменной `db`. Доступ к текущей транзакции тоже осуществляется через объект события, который передается в виде аргумента в обработчик события, где он становится доступным и присваивается локальной переменной.

Обработка этого события — единственный момент, когда можно создать объект-хранилище с ассоциированными индексами. В рассматриваемом примере создано хранилище данных с именем `reader` и ключом `id` с автоприращением. Еще два индекса — это поля хранилища данных `name` и `experience`. Запись данных в хранилище также происходит внутри события, хотя данные можно записывать и в определенный момент — например, когда пользователь отправляет заполненную HTML-форму.

После обработчика события `upgradeneeded` определены обработчики событий `success` и `error` — просто для предоставления обратной связи. И последним — по порядку, но не по важности — идет обработчик события `document.onclick`, который активирует доступ к базе данных. В примере обработчик базы данных предоставляет доступ к случайному экземпляру данных, к его транзакции, объекту-хранилищу и, наконец, к заданному ключу. При успешном выполнении запроса предоставляется доступ к полю `name`, и его значение выводится в консоль. Кроме предоставления доступа к отдельному значению можно использовать и курсор, но эти эксперименты я предоставляю вам провести самостоятельно.

В результате выполнения примера в консоль будут выведены сообщения в следующем порядке:

```
data added
data finished
datastore created
Name : Cindy Johnson
```

14.5. Упрощение IndexedDB с помощью библиотеки

Задача

Наладить асинхронное взаимодействие с IndexedDB при помощи промисов JavaScript.

Решение

Подключить библиотеку IDB (<https://github.com/jakearchibald/idb>), благодаря которой становится значительно удобнее пользоваться IndexedDB API. Кроме того, она может служить оберткой, позволяющей применять промисы.

В следующем файле импортируется библиотека IDB, затем создается хранилище данных IndexedDB и в него записываются данные:

```
import { openDB, deleteDB } from 'https://unpkg.com/idb?module';

const data = [
  { name: 'Riley Harrison', age: 57, experience: 1 },
  { name: 'Harlow Everly', age: 29, experience: 5 },
  { name: 'Abigail McCullough', age: 38, experience: 10 }
];

(async () => {
  // В целях демонстрации при загрузке страницы
  // удаляем существующую базу данных
  try {
    await deleteDB('CookbookIDB');
  } catch (err) {
    console.log('delete error', err);
  }

  // Открываем базу данных и создаем хранилище данных
  const database = await openDB('CookbookIDB', 1, {
    upgrade(db) {
      // Создаем хранилище объектов
      const store = db.createObjectStore('reader', {
        keyPath: 'id',
        autoIncrement: true
      });

      // Создаем ключи в хранилище объектов
      store.createIndex('experience', 'experience', { unique: false });
      store.createIndex('name', 'name', { unique: true });
    }
  });

  // Записываем в хранилище все данные из reader
  data.forEach(async value => {
    await database.add('reader', value);
  });
})();
```



В этом примере я загружаю модуль idb с сайта UNPKG (<https://unpkg.com>) — благодаря этому можно обращаться к нему напрямую по URL, вместо того чтобы устанавливать его локально. Это удобно в демонстрационных целях, но в рабочем приложении необходимо установить модуль через npm и подключить его к коду.

Обсуждение

IDB позиционирует себя как «крошечная библиотека, которая почти полностью соответствует IndexedDB API, но с маленькими улучшениями, имеющими большое значение для удобства использования». Благодаря IDB можно упростить некоторые синтаксические конструкции IndexedDB, кроме того, появляется возможность выполнять асинхронный код с промисами.

Метод `openDB` открывает базу данных и возвращает промис:

```
const db = await openDB(name, version, {
  // ...
});
```

В следующем примере пользователь может заносить значения в базу данных и получать из нее все данные, которые затем выводятся на страницу. HTML-файл выглядит так:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <meta http-equiv="X-UA-Compatible" content="ie=edge" />
    <title>IDB Discussion Example</title>
    <style>
      div {
        margin: 10px;
      }

      .data {
        width: 200px;
        background-color: yellow;
        padding: 5px;
      }
    </style>
  </head>
  <body>
    <h1>IDB Discussion Example</h1>

    <form>
      <div>
        <label for="name"> Enter name:</label>
        <input type="text" id="name" />
      </div>
      <div>
        <label for="age">Enter age:</label>
        <input type="text" id="age" />
      </div>
    </form>
    <button id="set">Set data</button>
    <button id="get">Get data</button>

    <p>Data:</p>
    <div class="data">
      <ul id="data-list"></ul>
    </div>

    <script type="module" src="idb-discussion.js"></script>
  </body>
</html>
```

А в файле `idb-discussion.js` содержится следующий код:

```
import { openDB } from 'https://unpkg.com/idb?module';

(async () => {
  // Открываем базу данных и создаем хранилище данных
  const database = await openDB('ReaderNames', 1, {
    upgrade(db) {
      // Создаем хранилище объектов
      const store = db.createObjectStore('reader', {
        keyPath: 'id',
        autoIncrement: true
      });

      // Создаем ключи в хранилище объектов
      store.createIndex('age', 'age', { unique: false });
      store.createIndex('name', 'name', { unique: true });
    }
  });

  async function setData() {
    const name = document.getElementById('name').value;
    const age = document.getElementById('age').value;

    await database.add('reader', {
      name,
      age
    });
  }

  async function getData() {
    // Получаем данные из базы reader
    const readers = await database.getAll('reader');

    const dataDisplay = document.getElementById('data-list');

    // Выводим на страницу значения name и age
    // для каждого объекта reader из базы данных
    readers.forEach(reader => {
      const value = `${reader.name}: ${reader.age}`;
      const li = document.createElement('li');
      li.appendChild(document.createTextNode(value));
      dataDisplay.appendChild(li);
    });
  }

  document.getElementById('set').onclick = setData;
  document.getElementById('get').onclick = getData;
})();
```

Я не буду приводить здесь полное описание API, но настоятельно рекомендую изучить документацию библиотеки (<https://github.com/jakearchibald/idb/blob/master/README.md>) и всегда использовать IDB при работе с IndexedDB.

Работа с мультимедиа

Красивые картинки. Анимация. Прикольные видео. Звук!

Интернет стал гораздо привлекательнее, когда обогатился всевозможными видами мультимедиа. Наши старые знакомые SVG и Canvas позволяют создавать сложную анимацию, диаграммы и графики. В HTML5 появились еще и видео- и аудиоэлементы, а в ближайшем будущем ожидаются элементы для 3D-графики.

И самое главное: ничто из этого не требует специальных плагинов — все эти возможности интегрированы во все браузеры для всех клиентов, включая смартфоны, планшеты и компьютеры.

15.1. JavaScript для SVG

Задача

Применить JavaScript к SVG-файлу или SVG-элементу.

Решение

Код JavaScript встраивается в SVG так же, как в обычный HTML, — в виде элементов `script`. Только, в отличие от HTML, в SVG код JavaScript должен располагаться внутри разметки CDATA (пример 15.1). Для работы с элементами SVG применяются те же методы DOM, что и для обычного HTML.

Пример 15.1. Использование JavaScript в SVG-файле

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<svg xmlns=http://www.w3.org/2000/svg
  xmlns:xlink="http://www.w3.org/1999/xlink" width="600" height="600">
  <script type="text/ecmascript">
    <![CDATA[
      // Создаем обработчик события onclick
      window.onload = function() {
        const square = document.getElementById('square');
```

```

        // Обработчик события onclick изменяет радиус окружности
        square.onclick = function click() {
            const color = this.getAttribute('fill');
            if (color === '#ff0000') {
                this.setAttribute('fill', '#0000ff');
            } else {
                this.setAttribute('fill', '#ff0000');
            }
        };
    }
]]>
</script>
<rect id="square" width="400" height="400" fill="#ff0000" x="10" y="10" />
</svg>

```

Обсуждение

Как видно из примера, SVG — это разновидность XML, поэтому в нем необходимо соблюдать правила встраивания кода в XML. Это значит, что в теге `script` следует указать тип скрипта и обернуть содержимое элемента `script` в блок CDATA. Без раздела CDATA появление в скрипте таких символов, как `<` или `&`, приведет к ошибкам на странице, поскольку синтаксический анализатор XML будет рассматривать эти символы как относящиеся к XML, а не к скрипту.



Иногда очень хочется рассматривать SVG как вариант HTML, особенно если SVG находится внутри HTML-документа. Именно так сделано в Chrome. Но лучше перестраховаться, чем потом жалеть, поэтому стоит выполнять требования XML.

Методы DOM, такие как `document.getElementById()`, можно использовать не только в HTML, но и в любом XML-документе, включая SVG. Но в SVG появляется атрибут `fill`, который применим только в SVG-элементах, таких как `rect`.



Если бы в одном из элементов в этом файле использовались пространства имен, то для методов DOM также нужно было бы указывать версию пространства имен.

В данном примере показан код, размещенный в отдельном SVG-файле с расширением `.svg`. Если встроить код SVG в HTML-файл, как показано в примере 15.2, то анимация с изменением цвета будет работать так же. Раздел CDATA можно удалить: все современные браузеры понимают, что данный SVG-код является частью HTML. Но в случае XHTML-файла раздел CDATA нужно сохранить.

Пример 15.2. SVG-элемент из примера 15.1, встроенный в HTML-страницу

```

<!DOCTYPE html>
<html>
<head>

```

```

<title>Accessing Inline SVG</title>
<meta charset="utf-8">
</head>
<body>
<svg width="600" height="600">
  <script>
    // Присваиваем элементу обработчик события onclick
    window.onload = function() {
      const square = document.getElementById('square');

      // Обработчик события onclick изменяет радиус окружности
      square.onclick = function click() {
        const color = this.getAttribute('fill');
        if (color === '#ff0000') {
          this.setAttribute('fill', '#0000ff');
        } else {
          this.setAttribute('fill', '#ff0000');
        }
      };
    };
  </script>
  <rect id="square" width="400" height="400" fill="#ff0000" x="10" y="10" />
</svg>
</body>
</html>

```

В этом примере код SVG встроен непосредственно в HTML-страницу. SVG-файл с кодом JavaScript можно также разместить на странице с помощью тега `<object>` с резервным тегом ``:

```

<object type="image/svg+xml" data="demo.svg">
  
</object>

```

SVG, в том числе встроенный в HTML, поддерживается всеми современными браузерами. IE поддерживает SVG, начиная с версии 9.



Если хотите узнать больше об SVG, советую почитать книгу Сары Драснер (Sarah Drasner) *SVG Animations*, выпущенную издательством O'Reilly.

Дополнительно: SVG-библиотеки

Библиотек для работы с SVG не так много, как для Canvas, но они очень удобны. Одна из самых популярных таких библиотек — D3, описанная в рецепте 15.3. В число популярных библиотек входят также Raphaël (<http://dmitrybaranovskiy.github.io/raphael/>), GreenSock (<https://greensock.com/>), Snap.svg (<http://snapsvg.io/>) и SVG.js (<https://svgjs.dev/docs/3.0>). Все они упрощают создание SVG-элементов и анимации. Следующий код представляет собой пример использования Raphaël:

```
// Создаем холст 320 x 400 в точке с координатами 10, 50
const paper = Raphael(10, 50, 320, 400);
// Рисуем круг радиусом 100 в точке с координатами x = 150, y = 140
const circle = paper.circle(150, 140, 100);
// Окрашиваем круг в красный цвет (#f00)
circle.attr("fill", "#f0f");
// Красим контур круга в белый цвет
circle.attr("stroke", "#ff0");
```

15.2. Доступ к SVG из скрипта веб-страницы

Задача

Изменить содержимое SVG-элемента из скрипта, размещенного на веб-странице.

Решение

Если код SVG размещен непосредственно на веб-странице, то для доступа к SVG-элементам и их атрибутам используется тот же функционал, что и для остальных элементов веб-страницы:

```
const square = document.getElementById("square");
square.setAttribute("width", "500");
```

Но если SVG-код находится во внешнем SVG-файле, который встроен в страницу с помощью элемента `object`, то, чтобы получить доступ к SVG-элементам, необходимо извлечь документ из внешнего SVG-файла. Для этого необходимо распознать объект, так как эта процедура в разных браузерах выполняется по-разному:

```
window.onload = function onLoad() {
    const object = document.getElementById('object');
    let svgdoc;

    try {
        svgdoc = object.contentDocument;
    } catch (e) {
        try {
            svgdoc = object.getSVGDocument();
        } catch (err) {
            console.log(err, 'SVG in object not supported in this
                environment');
        }
    }

    if (!svgdoc) return;

    const square = svgdoc.getElementById('square');
    square.setAttribute('width', '900');
};
```

Обсуждение

Первый вариант доступа к SVG, показанный в примере, — это случай, когда SVG-код размещен в HTML-файле. Доступ к SVG-элементу осуществляется теми же методами, что и в случае обычных HTML-элементов.

Второй вариант несколько сложнее, так как в нем объект документа извлекается из внешнего SVG-документа. Вначале приложение пытается получить доступ к свойству `contentDocument` этого объекта, а если это не получается, то пробует получить доступ к SVG-документу с помощью метода `getSVGDocument()`. Получив доступ к объекту SVG-документа, мы можем использовать те же методы DOM, что и для остальных элементов веб-страницы.

В примере 15.3 показаны второй способ размещения SVG-кода на веб-странице и то, как можно получить доступ к SVG-элементам из скрипта, вставленного в HTML-код.

Пример 15.3. Доступ к SVG в скрипте из элемента `object`

```
<!DOCTYPE html>
<head>
  <title>SVG in Object</title>
  <meta charset="utf-8" />
</head>
<body>
  <object id="object" type="image/svg+xml" data="../demo1.svg">
    <p>No SVG support</p>
  </object>
  <script type="text/javascript">
    const object = document.getElementById('object');
    object.onload = function() {
      let svgdoc;

      // Получаем доступ к объекту SVG-документа
      try {
        svgdoc = object.contentDocument;
      } catch (e) {
        try {
          svgdoc = object.getSVGDocument();
        } catch (err) {
          console.log(err, 'SVG in object not supported in this
            environment');
        }
      }

      if (!svgdoc) return;

      // Получаем SVG-элемент и изменяем его
      const square = svgdoc.getElementById('square');
      square.onclick = function() {
        let width = parseFloat(square.getAttribute('width'));
        width -= 50;
        square.setAttribute('width', width);
      }
    }
  </script>
</body>
```

```

        const color = square.getAttribute('fill');
        if (color == 'blue') {
            square.setAttribute('fill', 'yellow');
            square.setAttribute('stroke', 'green');
        } else {
            square.setAttribute('fill', 'blue');
            square.setAttribute('stroke', 'red');
        }
    };
};
</script>
</body>

```

В этом примере доступ к объекту осуществляется после того, как он будет загружен: в этот момент активируется обработчик события `object.onload`, который получает SVG-документ и назначает функцию для обработчика события `onclick`.

15.3. Построение столбчатой диаграммы в формате SVG с помощью библиотеки D3

Задача

Построить столбчатую диаграмму с возможностью изменения масштаба, не создавая каждый графический элемент по отдельности.

Решение

Построить диаграмму в виде SVG-кода средствами библиотеки D3, привязав ее к набору данных, предоставляемому приложением. В примере 15.4 показана вертикальная столбчатая диаграмма, созданная с использованием D3 для заданного набора данных, описывающего высоту каждого столбца.

Пример 15.4. Созданная с помощью D3 столбчатая диаграмма, представленная в виде SVG

```

<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>SVG Bar Chart using D3</title>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/d3/5.15.0/
      d3.min.js"></script>
  </head>
  <body>
    <script type="text/javascript">
      const data = [56, 99, 14, 12, 46, 33, 22, 100, 87, 6, 55,
        44, 27, 28, 34];

      const height = 400;

```



```
const barWidth = 25;

const x = d3
  .scaleLinear()
  .domain([0, d3.max(data)])
  .range([0, height]);

const svg = d3
  .select('body')
  .append('svg')
  .attr('width', data.length * (barWidth + 1))
  .attr('height', height);

svg
  .selectAll('rect')
  .data(data)
  .enter()
  .append('rect')
  .attr('fill', '#008b8b')
  .attr('x', function(d, i) {
    return i * (barWidth + 1);
  })
  .attr('y', function(d) {
    return height - x(d);
  })
  .attr('width', barWidth)
  .attr('height', x);
</script>
</body>
</html>
```

Обсуждение

D3 не относится к стандартным инструментам работы с графикой, которые создают геометрические фигуры по их размерам. В эту библиотеку можно передать набор данных и объекты для их визуализации, а она сделает все остальное. Это может показаться легким делом, но в действительности, чтобы воспользоваться всеми преимуществами такой визуализации, необходимо правильно составить исходный набор данных, что на первых порах работы с библиотекой бывает не просто.

Прежде всего необходимо учесть, что в D3 максимально используются *цепочки методов*. Конечно же, можно вызывать методы и по одному, но будет гораздо проще, понятнее и эффективнее, если задействовать предоставляемую библиотекой возможность объединения методов в цепочки.

В первой строке рассмотренного примера создается массив, который содержит набор данных. D3 ожидает, что точки диаграммы будут представлены в виде массива данных, но каждый элемент этого массива может быть не только простым значением, но и объектом, как показано в примере. Затем необходимо определить максимальную высоту столбца диаграммы, а также ширину каждого столбца. После этого наконец можно использовать D3.



Библиотека D3 (<http://d3js.org>), созданная Майком Бостоком, — это мощное средство визуализации данных, и оно слишком сложное, чтобы не напрягаясь освоить его за полдня. Но этот инструментарий стоит того, чтобы уделить ему время. Так что пример, приведенный в данном рецепте, стоит рассматривать не как подробную инструкцию, а скорее как анонс, призванный заинтересовать. Для более глубокого изучения D3 рекомендую книгу Филиппа Джанерта (Philipp Janert) *D3 for the Impatient* (издательство O'Reilly).

Я мог бы разместить на веб-странице статичный SVG-элемент, но мне хотелось показать, как D3 строит элементы. При создании SVG-элемента возвращается ссылка на этот элемент, чтобы потом с ним можно было работать, хотя D3 позволяет получить ссылку и на уже существующий элемент. В приведенном коде мы получаем ссылку на элемент `body` с помощью метода `select()` из библиотеки D3. После этого вставляем новый SVG-элемент в элемент `body` посредством метода `append()` и назначаем ему атрибуты с помощью метода `attr()`. Высота элемента задана изначально, а ширина вычисляется умножением количества элементов данных на ширину столбца (плюс 1, чтобы добавить отступы).

После того как SVG-элемент создан, мы с помощью функционала *масштабирования* D3 определяем высоту каждого столбца пропорционально высоте всего элемента таким образом, чтобы столбчатая диаграмма заполнила весь SVG-элемент, но при этом сохранялась пропорциональность всех столбцов. Это делается посредством метода `scale.linear()`, который строит линейную шкалу. Согласно документации D3, «отображение является линейным в том смысле, что значения на шкале результатов y могут быть выражены как линейная функция диапазона входных значений x : $y = mx + b$ ».

Функция `domain()` определяет масштабируемый диапазон входных значений, а функция `range()` — диапазон выходных значений. В рассмотренном примере входные значения находятся в диапазоне от нуля до максимального значения из набора данных, которое определяется вызовом функции `max()`. Нижнее значение диапазона высот SVG-элемента равно нулю. Затем возвращенная и присвоенная переменной функция будет нормализовывать все данные, передаваемые при вызове. Если передать в нее значение, равное самому большому из элементов данных, то возвращаемое значение будет равно высоте элемента (в данном случае самое большое значение из набора данных — 100, а возвращаемое масштабированное значение — 400).

В последней части кода создаются столбцы диаграммы. Нам нужно указать, с каким элементом следует работать, поэтому в коде вызывается функция `selectAll()` для элементов `rect`. В SVG-блоке пока что нет элементов `rect`, но мы их создадим. Данные передаются в D3 с помощью метода `data()`, после чего вызывается функция `enter()`. Функция `enter()` обрабатывает данные и возвращает заглушки для отсутствующих элементов. В данном примере создаются заглушки для всех 15 элементов `rect` — по одному для каждого столбца диаграммы.

Затем каждый элемент `rect` вставляется в SVG-элемент с помощью функции `append()`, и ему присваиваются атрибуты с помощью функции `attr()`. В данном

примере это атрибуты `fill` и `stroke`, хотя их можно было бы определить и в таблице стилей страницы. Затем в функцию передается значение атрибута `x` — координата нижнего левого угла столбца, которая вычисляется как функция от `d` (текущее значение данных) и `i` (текущий индекс). Для атрибута `x` индекс умножается на `barWidth` плюс 1, чтобы создать промежутки между столбцами диаграммы.

Вычислить значение атрибута `y` немного сложнее. Начальной точкой SVG-элемента является верхний левый угол, так что возрастание значений `y` идет вниз, а не вверх по диаграмме. Для того чтобы изменить это направление, нужно вычитать значение `y` из высоты элемента. Но это нельзя сделать напрямую. Если передать данные в код напрямую, то получим пропорциональную диаграмму с очень маленькими, сжатыми столбцами. Поэтому мы воспользовались созданной нами функцией `x` и передали данные туда.

Ширина всех столбцов — постоянная величина, она равна значению `barWidth`, а высота — это переменная, определяемая функцией масштабирования. Высота столбца определяется вызовом функции масштабирования, в которую передаются данные. В результате получается диаграмма, показанная на рис. 15.1.

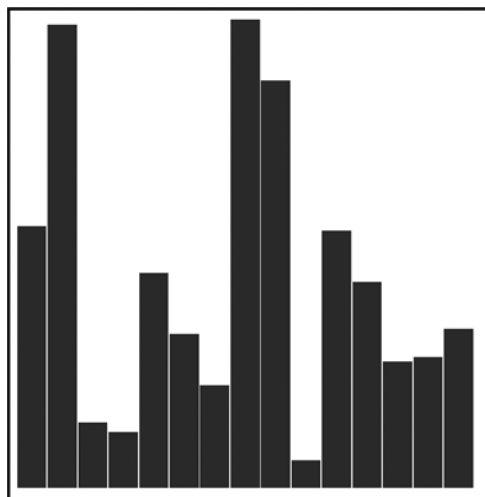


Рис. 15.1. Столбчатая диаграмма, высота столбцов которой нормализована так, чтобы заполнить заданное пространство

15.4. Интеграция элементов SVG и Canvas в HTML

Задача

Использовать на веб-странице элемент `canvas` в сочетании с SVG.

Решение

Один из вариантов — разместить элементы SVG и canvas непосредственно на HTML-странице, после чего обращаться к элементу canvas из скрипта, встроенного в SVG:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <meta http-equiv="X-UA-Compatible" content="ie=edge" />
    <title>Integrating SVG and the Canvas Element in HTML</title>
  </head>
  <body>
    <canvas id="myCanvas" width="400px" height="100px">
      <p>canvas item alternative content</p>
    </canvas>

    <svg id="svgelem" height="400">
      <title>SVG Circle</title>
      <script type="text/javascript">
        window.onload = function () {
          var context =
            document.getElementById("myCanvas").getContext('2d');
          context.fillStyle = 'rgba(0,200,0,0.7)';
          context.fillRect(0,0,100,100);
        }
      </script>
      <circle id="redcircle" cx="100" cy="100" r="100"
        fill="red" stroke="#000" />
    </svg>
  </body>
</html>
```

Или можно встроить элемент canvas непосредственно в SVG в виде внешнего объекта:

```
<!DOCTYPE html>
<html>
<head>
<title>Accessing Inline SVG</title>
<meta charset="utf-8">
</head>
<body>
<svg id="svgelem" height="400" width="600">
  <script type="text/javascript">
    window.onload = function () {
      var context2 =
        document.getElementById("thisCanvas").getContext('2d');
      context2.fillStyle = "#ff0000";
      context2.fillRect(0,0,200,200);
    };
  </script>
```

```
<foreignObject width="300" height="150">
  <canvas width="300" height="150" id="thisCanvas">
    alternate content for browsers that do not support Canvas
  </canvas>
</foreignObject>
<circle id="redcircle" cx="300" cy="100" r="100"
  fill="red" stroke="#000" />
</svg>
</body>
</html>
```

Обсуждение

При размещении SVG-элемента на текущей веб-странице можно обращаться к HTML-элементам из SVG. Можно также встроить элементы непосредственно в SVG, используя SVG-элемент `foreignObject`. Он позволяет встраивать в SVG код XHTML, MathML, RDF и другие варианты XML.

Метод `getElementById()` можно задействовать в обоих примерах. Однако если бы я захотел манипулировать элементами посредством других методов, таких как `getElementsByName()`, то мне пришлось бы внимательно следить за тем, с какой именно версией метода я работаю. Например, для внешнего элемента `canvas` можно применять `getElementsByName()`. Но если в `foreignObject` содержится XML-элемент, такой как RDF/XML, то нужно взять версию метода с учетом пространства имен — `getElementsByNameNS()`. Поскольку в данном случае встроенный элемент относится к HTML5, пространство имен можно не учитывать.

Получив доступ к контексту элемента `canvas`, можно использовать этот элемент в скрипте, встроенном в HTML-документ, как угодно: вставлять прямоугольники, рисовать линии, создавать дуги и т. п.

Дополнительно: Canvas или SVG?

Что лучше: размещать SVG внутри Canvas или Canvas внутри SVG? В случае покадровой анимации элемент `canvas` работает быстрее. При любой анимации браузеру не нужно заново перестраивать всю сцену — достаточно перерисовать изменившиеся пиксели. Но если приходится поддерживать различные размеры экрана, от смартфонов до широких мониторов, то преимущество элемента `canvas` при отображении анимации уменьшается, так как SVG лучше масштабируется.

Еще одно преимущество SVG состоит в более широких возможностях визуализации данных, которые достигаются благодаря поддержке мощных библиотек. Зато Canvas задействуется в 3D-системах, таких как WebGL.

Один из вариантов совместного использования SVG и Canvas — создание запасного варианта отображения для элемента `canvas`: в отличие от элемента `canvas`, SVG вставляется в DOM и остается там даже при отключенном JavaScript.

15.5. Выполнение процедуры в начале воспроизведения аудиофайла

Задача

Предоставить доступ к аудиофайлу и выводить дополнительную информацию в начале или в конце воспроизведения.

Решение

Использовать элемент HTML5 `audio`:

```
<audio id="meadow" controls>
  <source src="meadow.wav" type="audio/wav" />
  <p><a href="meadow.wav">Meadow sounds</a></p>
</audio>
```

И перехватывать его события `play` (начало воспроизведения) или `ended` (завершение воспроизведения):

```
const meadow = document.getElementById('meadow');
meadow.addEventListener('play', aboutAudio);
```

А затем выводить информацию:

```
function aboutAudio() {
  const info = 'A summer field near a lake in July.';
  const txt = document.createTextNode(info);
  const div = document.createElement('div');
  div.appendChild(txt);
  document.body.appendChild(div);
}
```

Обсуждение

В HTML5 появились два элемента поддержки мультимедиа: `audio` и `video`. Это простые элементы, позволяющие воспроизводить аудио- и видеофайлы.

В предлагаемом решении элементу `audio` присвоен атрибут `controls`, принимающий булевы значения, благодаря чему выводятся элементы управления воспроизведением. В атрибуте `src` указан WAV-файл, который должен воспроизводиться в браузере. В качестве запасного варианта приводится ссылка на этот же WAV-файл, чтобы пользователи, у которых браузер не поддерживает воспроизведение аудио, тоже могли получить доступ к нему. Вместо этого можно было бы использовать элемент `object` или какой-нибудь другой запасной вариант.



WAV — широко распространенный аудиоформат, однако в браузерах поддерживаются и другие форматы и типы файлов. В Mozilla Developer Network есть полная таблица поддержки аудио- и видеокодеков в разных браузерах (<http://mzl.la/1DS3rPL>), а в «Википедии» представлена более простая таблица поддержки браузерами форматов кодирования звука (<https://oreil.ly/55EwV>).

Для мультимедийных элементов есть ряд методов управления воспроизведением, а также события, которые активируются в различных ситуациях. В данном примере перехватывается событие `ended` — ему назначен обработчик `aboutAudio()`, который по окончании воспроизведения выводит сообщение о файле. Обратите внимание: несмотря на то что в коде использован обработчик события уровня DOM Level 0, который вызывается после загрузки окна, для элемента `audio` задействуется обработчик события уровня DOM Level 2. Он поддерживается не всеми браузерами, поэтому я настоятельно рекомендую применять `addEventListener()`. Однако обработка события `onended`, похоже, не вызывает проблем, если использовать это событие непосредственно в элементе:

```
<audio id="meadow" src="meadow.wav" controls onended="alert('All done')">
  <p><a href="meadow.wav">Meadow sounds</a></p>
</audio>
```

Любопытно понаблюдать, как выводятся мультимедийные элементы в тех браузерах, в которых они поддерживаются. Стандартного отображения не существует, поэтому каждый браузер предлагает собственную интерпретацию. Чтобы как-то контролировать отображение мультимедийных элементов, можно создавать свои элементы управления и применять собственные элементы/CSS/SVG/Canvas для придания им желаемого вида.

15.6. Управление отображением видео с помощью элемента `video` и JavaScript

Задача

Разместить видео на веб-странице и сделать так, чтобы элементы управления воспроизведением всегда выглядели одинаково независимо от браузера и операционной системы.

Решение

Использовать элемент HTML5 `video`:

```
<video id="meadow" poster="purples.jpg" >
  <source src="meadow.m4v" type="video/mp4"/>
```

```
    <source src="meadow.ogv" type="video/ogg" />
</video>
```

Как показано в примере 15.5, им можно управлять посредством JavaScript. Для управления воспроизведением используются кнопки, а текст в элементе `div` предназначен для вывода информации в процессе воспроизведения.

Пример 15.5. Создание элементов управления для элемента HTML5 video

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <meta http-equiv="X-UA-Compatible" content="ie=edge" />
    <title>Controlling Video from JavaScript with the video
      Element</title>
    <style>
      video {
        border: 1px solid black;
        max-width: 600px;
      }
    </style>
  </head>
  <body>
    <h1>Controlling Video from JavaScript with the video Element</h1>

    <video id="meadow" controls>
      <source src="meadow.mp4" type="video/mp4" />
      <source src="meadow.webm" type="video/webm" />
    </video>
    <div id="feedback"></div>
    <div id="controls">
      <button id="start">Play</button>
      <button id="stop">Stop</button>
      <button id="pause">Pause</button>
    </div>

    <script src="video.js"></script>
  </body>
</html>
```

В файле `video.js` содержится следующий код:

```
// Элементы DOM
const meadow = document.getElementById('meadow');
const start = document.getElementById('start');
const pause = document.getElementById('pause');
const stop = document.getElementById('stop');

// Запуск видео, активация кнопок stop и pause
// Отключение кнопки play
function startPlayback() {
  meadow.play();
```



```
    pause.disabled = false;
    stop.disabled = false;
    this.disabled = true;
}

// Приостановка воспроизведения video, активация
// кнопки start, отключение кнопок stop и pause
function pausePlayback() {
    meadow.pause();
    pause.disabled = true;
    start.disabled = false;
    stop.disabled = true;
}

// Остановка воспроизведения видео, возврат к началу
// Активация кнопки play, отключение кнопок pause и stop
function stopPlayback() {
    meadow.pause();
    meadow.currentTime = 0;
    start.disabled = false;
    pause.disabled = true;
    this.disabled = true;
}

// Для каждого момента времени, кратного 5, выводим информацию
function reportProgress() {
    const time = Math.round(this.currentTime);
    const div = document.getElementById('feedback');
    div.innerHTML = `${time} seconds`;
}

// Обработчики событий
document.getElementById('start').addEventListener('click', startPlayback);
document.getElementById('stop').addEventListener('click', stopPlayback);
document.getElementById('pause').addEventListener('click', pausePlayback);
meadow.addEventListener('timeupdate', reportProgress);
```

Обсуждение

Элемент HTML5 `video`, как и HTML5 `audio`, может иметь стандартные, встроенные элементы управления, либо же мы можем создать собственные. Элементы воспроизведения мультимедиа поддерживают следующие методы:

- `play` — запустить воспроизведение видео;
- `pause` — приостановить воспроизведение видео;
- `load` — загрузить видео перед началом воспроизведения;
- `canPlayType` — проверить, поддерживает ли браузер данный видеоформат.

У элементов мультимедиа нет метода для остановки воспроизведения — оно имитируется приостановкой воспроизведения и присвоением значения 0 атрибуту `currentTime` элемента `video`, в результате чего время воспроизведения

сбрасывается. Я также использовал атрибут `currentTime` для вывода времени воспроизведения видео, округляя его до ближайших секунд с помощью метода `Math.round`.

Элементы воспроизведения видео позволяют применять один из двух видеокodeков: H.264 (`.mp4`) и VP8 (`.webm`). Формат WebM поддерживается практически во всех современных браузерах, а для более старых в качестве запасного варианта можно задействовать элемент `video` с MP4.

Стандартные элементы воспроизведения видео и аудио реагируют на нажатия клавиш клавиатуры. При их замене необходимо позаботиться о том, чтобы новые элементы управления также были доступны с клавиатуры.



Представленные в данном примере функции воспроизведения видео работают с исходным, незашифрованным видео. Если видео- или аудиофайл зашифрован, то для его воспроизведения необходимо приложить значительно больше усилий и использовать HTML 5.1 W3C Encrypted Media Extensions (EME).

Рабочая версия проекта W3C EME (<https://oreil.ly/mMu7q>) реализована в Internet Explorer 11 (<http://bit.ly/1DS5umQ>), Chrome, Firefox, Microsoft Edge и Safari.

Создание веб-приложений

Было время, когда JavaScript использовался для простых интерактивных веб-страниц, а сейчас на этом языке можно писать сложные полнофункциональные приложения, которые работают в веб-браузере. JavaScript позволяет создавать электронные карты, почтовые клиенты, сайты потоковой передачи видео, чаты реального времени и многое другое. Грань между сайтом и приложением очень нечеткая, но одно можно сказать точно: приложение — это сайт, который принимает данные от пользователя и что-то возвращает в ответ.

Вы, как разработчик, можете создавать такие приложения и сразу же развертывать их по всему миру, но такая возможность сопряжена с некоторыми уникальными сложностями. По мере роста кодовой базы приложения вам придется разбить ее на модули меньшего размера и обеспечить передачу пользователям пакетов с оптимизированным кодом. Создаваемые вами функции и способы взаимодействия будут конкурировать с аналогичными возможностями уже существующих мобильных приложений, такими как автономная работа, отправка уведомлений и значки приложений. К счастью, современный JavaScript и API браузеров обеспечивают эти широкие возможности.

16.1. Создание пакетов JavaScript

Задача

Использовать в браузере модули JavaScript.

Решение

Задействовать стандартную поддержку модулей JavaScript или сборщик пакетов, такой как Webpack (<https://webpack.js.org>). Поддержка модулей JavaScript реализована во всех современных браузерах (<https://oreil.ly/FhPq9>). Предположим, у нас есть простой модуль `mod.js`, который экспортирует значение:

```
export const name = 'Riley';
```

Мы можем использовать его непосредственно в HTML-файле:

```
<script type='module'>
  import {name} from './mod.js';
  console.log(name);
</script>
```

Для более сложных приложений и сайтов лучше взять сборщик пакетов, способный оптимизировать модули. Для построения пакетов с помощью Webpack вначале нужно установить зависимости этого сборщика посредством npm:

```
$ npm install webpack webpack-cli --save-dev
```



Прежде чем устанавливать пакеты из npm, необходимо создать в проекте файл `package.json`. Для того чтобы сгенерировать его, перейдите в корневой каталог проекта, введите команду `npm init` и ответьте на несколько уточняющих вопросов в интерфейсе командной строки. Подробнее об установке и использовании npm читайте в главе 1.

После этого можно сгенерировать в корневом каталоге проекта файл `webpack.config.js` — в нем указываются исходный файл модуля и каталог, в котором будет размещен готовый файл модуля:

```
const path = require('path');

module.exports = {
  entry: './src/index.js',
  output: {
    filename: 'bundle.js',
    path: path.resolve(__dirname, 'dist')
  }
};
```

После этого, чтобы запускать сборщик Webpack, внесите скрипт в файл `package.json`:

```
"scripts": {
  ...
  "build": "webpack"
}
```

Обсуждение

Сейчас модули JavaScript легко создаются и широко поддерживаются браузерами. Это позволяет разбивать код на куски меньшего размера, которыми проще управлять.

Webpack — популярный сборщик модулей JavaScript, который работает на основе конфигурационного файла.

Согласно приведенному ранее конфигурационному файлу, Webpack должен найти в каталоге `src` файл `index.js`. Это и есть исходный файл нашего проекта JavaScript:

```
import foo from './foo.js';
import bar from './bar.js';

foo();
bar();
```

В файле `index.js` импортируются еще два файла — `foo.js` и `bar.js`.

Выполняя скрипт `build`, Webpack создаст в каталоге `dist` новый, сжатый файл с именем `bundle.js`.

Компиляция простых операторов импорта — это лишь верхушка айсберга. Webpack позволяет выполнять динамическую перезагрузку модулей, разделение кода, преобразование кода для специфических браузеров и даже может использоваться как сервер разработки. В рецепте 16.2 будет показано, как с помощью Webpack уменьшить размер пакета JavaScript.

Дополнительно: использование модулей npm

Кроме тех модулей, которые вы разработали сами, Webpack позволяет загружать и применять готовые модули непосредственно из npm (<https://www.npmjs.com>). Для этого нужно сначала установить модуль и сохранить его в проекте в качестве зависимости:

```
$ npm install some-module --save
```

После этого можно обращаться к нему непосредственно из кода, причем указывать путь не обязательно:

```
import some-code from 'some-module'
```

16.2. JavaScript для мобильного интернета

Задача

Применение JavaScript значительно замедляет загрузку сайта или приложения на мобильных устройствах и при медленном соединении с интернетом.

Решение

Для сайтов с небольшим количеством кода JavaScript, который размещен в одном файле, можно использовать утилиту сжатия кода, такую как UglifyJS (<https://github.com/mishoo/UglifyJS>). Размер файла JavaScript уменьшается за счет удаления лишних символов, например пробелов.

Для того чтобы применить утилиту UglifyJS, ее необходимо сначала установить с помощью npm:

```
$ npm install uglify-js
```

Затем вставить ссылку на код в файл `package.json`, указав имена входного и выходного файлов JavaScript:

```
"scripts": {  
  "minify": "uglifyjs index.js --output index.min.js"  
}
```

Для более крупных сайтов и приложений, в которых применяются несколько файлов JavaScript, можно воспользоваться сборщиком, таким как Webpack (<https://webpack.js.org>). Сборщик выполнит сжатие и разделение кода, а также «встряску» дерева вызовов функций и обеспечит отложенную загрузку. Webpack автоматически сжимает код в режиме эксплуатации приложения, не требуя для этого ни специальной настройки, ни дополнительных средств сжатия.

Разделение кода — это процесс, в ходе которого генерируются несколько сжатых пакетов, так что на каждой HTML-странице или в шаблоне в итоге загружается только тот код, который там нужен. Следующий файл `webpack.config.js` приведет к созданию в каталоге `dist` двух файлов JavaScript — `index.bundle.js` и `secondary.bundle.js`:

```
const path = require('path');  
  
module.exports = {  
  entry: {  
    index: './src/index.js',  
    secondary: './src/secondary.js',  
  },  
  output: {  
    filename: '[name].bundle.js',  
    path: path.resolve(__dirname, 'dist'),  
  },  
};
```

Пакеты имеют свойство стремительно разбухать, особенно при импортировании сторонних библиотек, часть функционала которых не нужна. Чтобы убрать «мертвый» (неиспользуемый) код, применяется концепция *«встряски» дерева вызовов функций* (tree shaking). Чтобы настроить Webpack на удаление «мертвого» кода, нужно добавить следующий параметр оптимизации:

```
module.exports = {  
  mode: 'development',  
  entry: {  
    index: './src/index.js',  
    secondary: './src/secondary.js'  
  },  
  output: {  
    filename: '[name].bundle.js',  
    path: path.resolve(__dirname, 'dist')  
  },  
  optimization: {
```

```
    usedExports: true
  }
};
```

На последнем этапе разделения кода в файл проекта `package.json` добавляется поле побочных эффектов `sideEffects`. В соответствии с документацией Webpack, *побочный эффект* — это код, который при импортировании выполняет какие-то действия помимо одной или нескольких операций `export`. Примером побочного эффекта является глобальное полизаполнение, в котором вообще нет операторов `export`.

Если такого файла нет, то в `package.json` нужно вставить следующую строку:

```
"sideEffects": false
```

Если же в проекте есть файлы JavaScript, попадающие в категорию побочных эффектов, то нужно перечислить их в виде массива:

```
"sideEffects": [
  "./src/file-with-side-effect.js"
]
```

Наконец, Webpack можно использовать для отложенной загрузки модулей JavaScript — режима, в котором модули загружаются только тогда, когда они необходимы для определенного взаимодействия с браузером. Благодаря Webpack это делается просто — с помощью динамических операторов `import`. Так, содержимое файла `button.js` из каталога `src` будет загружено только тогда, когда пользователь нажмет кнопку. Код в `index.js` выглядит так:

```
const buttonElement = document.getElementById('button');
buttonElement.onclick = e => {
  import(/* webpackChunkName: "button" */ './button').then(module => {
    const button = module.default;
    button();
  });
};
```

Обсуждение

Самый быстрый JavaScript — тот, где нет JavaScript. Однако в основе интерактивности, которую требуют современные веб-приложения, часто лежит клиентская часть, написанная на JavaScript. С учетом этого наша цель — ограничить количество и размер файлов JavaScript, загружаемых клиентским браузером. Такие стратегии, как сжатие, разделение кода, «встряска» дерева вызовов функций и отложенная загрузка, позволяют лучше контролировать объем кода и количество файлов JavaScript, загружаемых браузером на стороне пользователя.

Читайте также

Руководство Webpack's Getting Started (<https://oreil.ly/TAnYG>) представляет собой полезное начальное пособие по сборке кода и файлам конфигурации Webpack.

16.3. Создание прогрессивного веб-приложения

Задача

Использовать в веб-приложении функции, свойственные обычным приложениям, такие как быстрая загрузка, автономная работа и загрузка значков.

Решение

Превратить веб-приложение в прогрессивное веб-приложение (Progressive Web Application, PWA). Этим термином описывают набор технологий, сочетание которых позволяет реализовывать в веб-приложениях такие типичные функции обычных приложений, как автономная работа и устанавливаемые пользователем значки приложений. При этом приложение по-прежнему строится на основе стандартных веб-технологий и распространяется через интернет.

У PWA помимо возможностей обычной веб-страницы должны иметься следующие два компонента:

- *манифест приложения* — свойства приложения, о которых нужно сообщить браузеру;
- *Service Worker* — функциональность, обеспечивающая автономную работу приложения.

Первый этап построения прогрессивного веб-приложения — создание файла с манифестом приложения. Благодаря этому файлу у разработчиков появляется доступ к такому функционалу, как значки приложений, всплывающие окна, стиль отображения приложения в браузере и ориентация веб-страницы. Содержимое файла `manifest.json` выглядит так:

```
{
  "name": "JavaScript Everywhere",
  "short_name": "JavaScript",
  "start_url": "/index.html",
  "display": "standalone",
  "background_color": "#ffc40d",
  "theme_color": "#ffc40d",
  "icons": [
    {
      "src": "/images/icons/icon-192x192.png",
      "sizes": "192x192",
      "type": "image/png"
    },
    {
      "src": "/images/icons/icon-512x512.png",
      "sizes": "512x512",
      "type": "image/png"
    }
  ]
}
```


Теперь остается добавить в HTML-файлы и шаблоны ссылку на файл манифеста, а в раздел `<head>` — соответствующие значки (пример 16.1).

Пример 16.1. Метатеги PWA

```
<!-- Ссылка на файл manifest.json -->
<link rel="manifest" href="manifest.json" />
<!-- Ссылка на значки iOS -->
<link rel="apple-touch-icon" sizes="180x180" href="images/icons/
      apple-touch-icon.png" />
<!-- Значки и цвета для плиток приложений Microsoft -->
<meta name="msapplication-TileColor" content="#ffc40d" />
<meta name="msapplication-TileImage" content="/img/icons/mstile-310x310.png" />
<!-- Цвет темы -->
<meta name="theme-color" content="#ffc40d" />
```

Если веб-сайт соответствует критериям PWA, то автоматически запускается система подсказок для установки PWA (рис. 16.1). После установки PWA на устройстве пользователя появляется значок PWA, как и при установке обычного приложения (рис. 16.2).

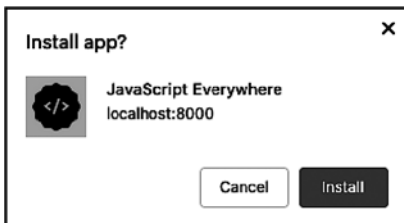


Рис. 16.1. Система подсказок для установки PWA



Рис. 16.2. Приложение может быть сохранено на мобильном устройстве

На втором этапе создается Service Worker — скрипт, который выполняется независимо от страницы. Он обеспечивает автономную работу сайта, ускоряет работу приложения и позволяет делать резервные копии. С учетом ограниченной пропускной способности мобильных соединений благодаря Service Worker появляется

возможность запуска приложений в автономном режиме, когда содержимое загружается после первого посещения сайта пользователем независимо от состояния сети. Но самое главное свойство Service Worker — то, что это по-настоящему прогрессивное усовершенствование, позволяющее выделить функционал для браузеров, поддерживающих PWA, в отдельный уровень, не изменяя функциональность сайта для пользователей браузеров, которые не поддерживают PWA.

Первый шаг при создании Service Worker — регистрация скрипта, который будет содержать код Service Worker, в браузере пользователя. Для этого нужно разместить скрипт регистрации внизу страницы, непосредственно перед закрывающим тегом `</body>`:

```
<!-- Инициализируем Service Worker -->
<script>
  if ('serviceWorker' in navigator) {
    window.addEventListener('load', function() {
      navigator.serviceWorker
        .register('service-worker.js')
        .then(reg => {
          console.log('Service worker registered!', reg);
        })
        .catch(err => {
          console.log('Service worker registration failed: ', err);
        });
    });
  }
</script>
```

В нем мы проверяем, поддерживает ли данный браузер технологию Service Worker, и, если поддерживает, даем браузеру ссылку на скрипт Service Worker (в данном случае это файл `service-worker.js`). Для удобства отладки в скрипте реализованы также перехват ошибок и вывод в консоль сообщений о них.

В файле `service-worker.js` прежде всего определяется версия кэша и приводится список файлов, которые должны кэшироваться в браузере:

```
var cacheVersion = 'v1';

filesToCache = [
  'index.html',
  '/styles/main.css',
  '/js/main.js',
  '/images/logo.svg'
]
```



При внесении изменений на сайте значение `cacheVersion` также необходимо изменить, иначе пользователи могут получать неизменное содержимое сайта, сохраненное в кэше.

Затем в файле `service-worker.js` создаются обработчики событий `install`, `fetch` и `activate`. При возникновении события `install` браузеру передаются инструкции

по установке файлов, сохраненных в кэше. При возникновении события `fetch` браузер получает инструкции по обработке таких событий: это может быть загрузка файлов — либо из кэша, либо полученных по сети. Наконец, событие `activate`, которое возникает при активации Service Worker, может использоваться для проверки объектов, хранящихся в кэше, и их удаления в случае, если изменилось значение `cacheVersion` или если данного файла больше нет в списке `filestoCache` (рис. 16.3):

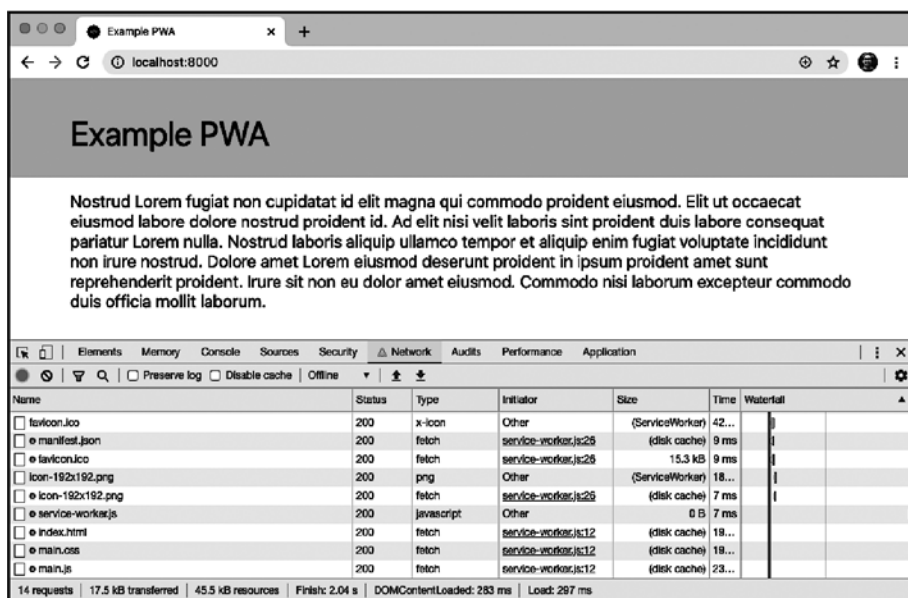


Рис. 16.3. После установки Service Worker приложение может загружать файлы в автономном режиме

```
const cacheVersion = 'v1';

const filesToCache = ['index.html', '/styles/main.css', '/js/main.js'];

self.addEventListener('install', event => {
  console.log('Service worker install event fired');
  event.waitUntil(
    caches.open(cacheVersion).then(cache => {
      return cache.addAll(filesToCache);
    })
  );
});

self.addEventListener('fetch', event => {
  console.log('Fetch intercepted for:', event.request.url);
  event.respondWith(
    caches.match(event.request).then(cachedResponse => {
      if (cachedResponse) {
        return cachedResponse;
      }
    })
  );
});
```

```
        return fetch(event.request);
    })
  );
});

self.addEventListener('activate', event => {
  event.waitUntil(
    caches.keys().then(keyList => {
      return Promise.all(
        keyList.map(key => {
          if (key !== cacheVersion) {
            return caches.delete(key);
          }
        })
      );
    })
  );
});
});
```

Обсуждение

Приложения Progressive Web Application устанавливаются пользователем и имеют определенный функционал для автономной работы. Он позволяет веб-приложениям максимально точно имитировать лучшие свойства обычных приложений, но в то же время пользоваться преимуществами открытой сети.

Манифест веб-приложения представляет собой файл в формате JSON, в котором содержится информация о приложении. Вот полный список основных значений, которые могут содержаться в этом файле:

- **background_color** — код фоновой цвета для экрана запуска;
- **categories** — категории, к которым относится приложение, представленные в виде массива строк;
- **description** — строка с описанием приложения;
- **dir** — направление отображения символов: **auto**, **ltr** (слева направо) или **rtl** (справа налево);
- **display** — предпочтительный режим отображения: **browser** (стандартное поведение браузера) или **fullscreen** (на некоторых устройствах браузер сворачивается);
- **iarc_rating_id** — возрастной рейтинг (International Age Rating);
- **icons** — массив объектов со ссылками на изображения и описания значков;
- **lang** — основной язык приложения;
- **name** — название приложения;
- **orientation** — позволяет разработчику определить для приложения ориентацию экрана по умолчанию;

- `prefer_related_applications` — если этот параметр равен `true`, то разработчик может указать аналогичные приложения, которые можно установить вместо данного веб-приложения;
- `related_applications` — массив объектов, в котором содержится список аналогичных обычных приложений;
- `scope` — строка, в которой обозначена область навигации приложения. Если она задана, то навигация внутри приложения ограничивается указанным здесь каталогом;
- `screenshots` — массив копий экрана приложения;
- `short_name` — сокращенное название приложения, которое можно использовать там, где полное название не помещается;
- `start_url` — URL страницы, которая должна открываться при запуске приложения;
- `theme_color` — строка, в которой указан цвет темы приложения, используемый по умолчанию.

В документации W3C приводится образец правильного файла манифеста для онлайн-игры (<https://oreil.ly/zlk9P>):

```
{
  "lang": "en",
  "dir": "ltr",
  "name": "Super Racer 3000",
  "description": "The ultimate futuristic racing game from the future!",
  "short_name": "Racer3K",
  "icons": [{
    "src": "icon/lowres.webp",
    "sizes": "64x64",
    "type": "image/webp"
  }, {
    "src": "icon/lowres.png",
    "sizes": "64x64"
  }, {
    "src": "icon/hd_hi",
    "sizes": "128x128"
  }],
  "scope": "/racer/",
  "start_url": "/racer/start.html",
  "display": "fullscreen",
  "orientation": "landscape",
  "theme_color": "aliceblue",
  "background_color": "red",
  "screenshots": [{
    "src": "screenshots/in-game-1x.jpg",
    "sizes": "640x480",
    "type": "image/jpeg"
  }, {
    "src": "screenshots/in-game-2x.jpg",
```

```
    "sizes": "1280x920",  
    "type": "image/jpeg"  
  }  
}
```

На некоторых платформах, таких как iOS и Windows, для веб-приложения помимо файла манифеста нужно указать дополнительную информацию в виде метатегов HTML. В примере 16.1 с помощью метатегов определены цвет темы, значок iOS и параметры плиток Windows.



Создание значков для всех возможных типов устройств и всех разрешений экрана может оказаться весьма утомительным занятием. Советую использовать RealFaviconGenerator (<https://oreil.ly/AlsQe>).

Service Worker — это скрипт, который выполняется в браузере в фоновом режиме параллельно с отображением и выполнением кода страницы. Поскольку это рабочий процесс (worker), он не имеет прямого доступа к DOM. Зато, поскольку Service Worker выполняется параллельно, благодаря ему у приложения появляется множество новых вариантов использования. Один из самых впечатляющих — возможность кэшировать куски приложения, чтобы потом оно могло работать автономно. В приведенном ранее примере я выполнил кэширование файлов HTML, JavaScript и CSS, чтобы обеспечить полноценную (хотя и по минимуму) автономную работу сайта.

Другие варианты применения включают в себя создание отдельного автономного интерфейса либо кэширование общих шаблонов разметки и стилей, которые часто называют оболочкой приложения.

Следует учитывать, что использование Service Worker накладывает следующие ограничения.

- Сайты, на которых применяются скрипты Service Worker, обслуживаются только по протоколу HTTPS.
- Скрипты Service Worker не работают, если пользователь включил в браузере режим приватного просмотра.
- Поскольку браузер выполняет Service Worker в отдельном потоке, у Service Worker нет доступа к DOM.
- Скрипты Service Worker ограничены своей областью видимости — это значит, что их необходимо размещать в корневом каталоге приложения.
- Объем хранилища для кэша зависит от браузера и свободного места на жестком диске у пользователя.

В предыдущем примере я создал Service Worker вручную, однако при таком подходе большое приложение быстро станет неуправляемым. Для управления скриптами Service Worker и автономной работы веб-приложений можно использовать

созданную Google библиотеку Workbox (<https://oreil.ly/Gu3Z6>). Она берет на себя большую часть забот по управлению версиями и кэшем и выполняет более сложные операции, такие как фоновая синхронизация и предварительное кэширование.

Прогрессивные веб-приложения — важный шаг в сторону интернет-приложений, не зависящих от фреймворка: они могут быть построены как на базе обычных HTML, CSS и JavaScript, так и с использованием одного из новых фреймворков JavaScript. В этом разделе мы лишь немного коснулись возможностей этих технологий. Подробное описание свойств и функциональных возможностей прогрессивных веб-приложений можно найти в книге Тала Альтера (Tal Alter) *Building Progressive Web Apps*, выпущенной издательством O'Reilly.

16.4. Тестирование и профилирование прогрессивных веб-приложений

Задача

Протестировать прогрессивное веб-приложение, чтобы убедиться, что оно удовлетворяет всем требованиям.

Решение

С помощью Lighthouse (<https://oreil.ly/hEdHB>) можно выполнить аудит производительности, доступности, соответствия лучшим рекомендациям, SEO и критериям прогрессивных веб-приложений. Чтобы получить доступ к Lighthouse, проще всего перейти на вкладку Lighthouse на панели инструментов разработки в Google Chrome. Откройте сайт (на рабочем или на локальном веб-сервере) и нажмите кнопку **Generate Report** (рис. 16.4).

Lighthouse сгенерирует отчет, включив в него рекомендации по улучшению всех показателей (рис. 16.5 и 16.6).

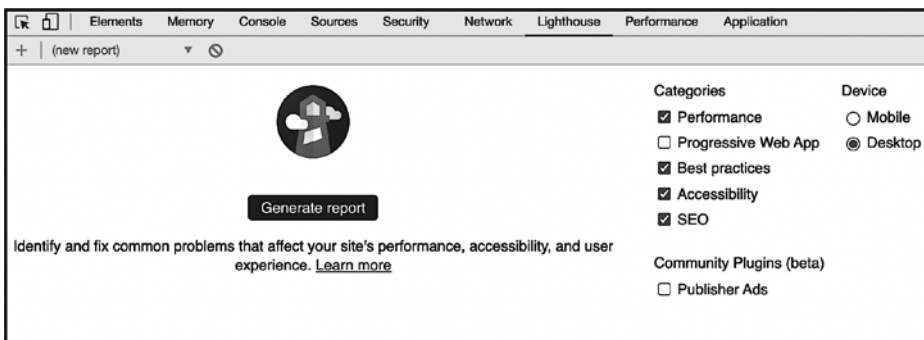


Рис. 16.4. Вкладка Lighthouse на панели инструментов разработки в Chrome



Рис. 16.5. Высокие баллы означают, что это эффективное приложение, соответствующее требованиям прогрессивных веб-приложений

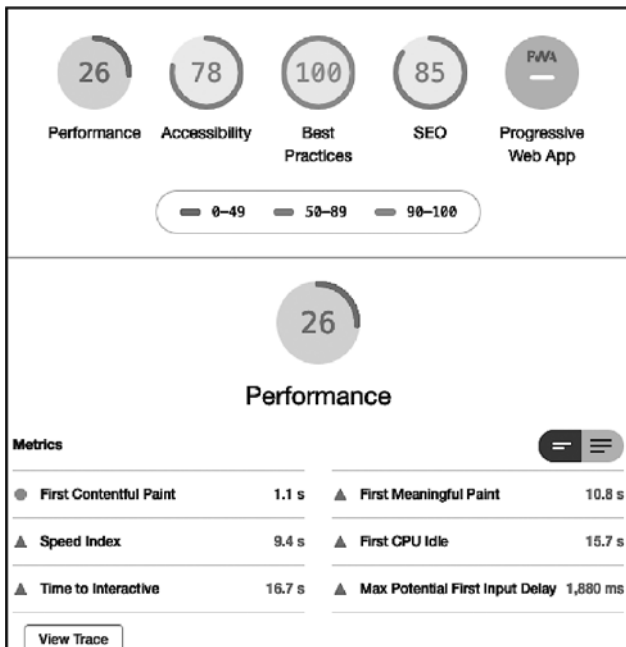


Рис. 16.6. Если сайт получает низкие оценки, то Lighthouse дает рекомендации по улучшению показателей



Подробнее об использовании Lighthouse на панели инструментов разработки Chrome для обычных сайтов, а не для прогрессивных веб-приложений, читайте в рецепте 11.4.

Обсуждение

Lighthouse — это инструмент, позволяющий проверить сайт на соответствие рекомендациям для веб-приложений, включая производительность и соответствие критериям прогрессивных веб-приложений. Инструмент Lighthouse встроен в панель инструментов разработки Chrome, его также можно установить в Firefox в виде расширения.

Lighthouse доступен не только в браузерах — его можно установить через npm и использовать в командной строке или как модуль Node. Lighthouse устанавливается так же, как и другие модули Node:

```
$ npm install -g lighthouse
```

Затем можно запустить Lighthouse, передав URL в качестве аргумента:

```
$ lighthouse https://www.oreilly.com/
```

Если вдобавок передать аргумент `--view`, то результаты откроются в браузере:

```
$ lighthouse https://www.oreilly.com/ --view
```

Можно также указать имя и расположение выходного файла, чтобы сохранить там результаты проверки:

```
$ lighthouse https://www.oreilly.com/ --view --output html
--output-path ./report.html
```

С помощью файла `budget.json` можно проверить производительность с учетом заданных бюджетных ограничений. В файле `budget.json` задаются следующие ограничения тестирования:

```
[
  {
    "path": "/*",
    "timings": [
      {
        "metric": "interactive",
        "budget": 3000
      },
      {
        "metric": "first-meaningful-paint",
        "budget": 1000
      }
    ]
  }
],
```

```

    "resourceSizes": [
      {
        "resourceType": "script",
        "budget": 125
      },
      {
        "resourceType": "total",
        "budget": 300
      }
    ],
    "resourceCounts": [
      {
        "resourceType": "third-party",
        "budget": 10
      }
    ]
  }
]

```



Команда Google Chrome создала репозиторий (<https://github.com/GoogleChrome/budget.json>), в котором хранится документация с описанием всех параметров budget.json.

Локальное тестирование из командной строки хорошо подходит для разработки на локальном сервере. Но главные возможности Lighthouse как модуля кода раскрываются при использовании средств непрерывной интеграции, таких как GitHub Actions, Circle CI, Jenkins и Travis CI. Модуль Lighthouse CI (<https://github.com/GoogleChrome/lighthouse-ci>) позволяет выполнять тестирование с помощью Lighthouse в конвейере непрерывной интеграции — например, при каждом запросе на внесение изменений в репозиторий GitHub.

Вот пример конфигурации Circle CI:

```

version: 2.1
jobs:
  build:
    docker:
      - image: circleci/node:10.16-browsers
    working_directory: ~/your-project
    steps:
      - checkout
      - run: npm install
      - run: npm run build
      - run: sudo npm install -g @lhci/cli@0.3.x
      - run: lhci autorun

```

Полное описание использования Lighthouse в разных средах непрерывной интеграции содержится в руководстве Google Getting Started (<https://oreil.ly/7jnwX>).

16.5. Получение текущего URL

Задача

Для работы приложения нужно прочесть значение текущего URL.

Решение

Получить полный текущий URL, прочитав свойство `href` объекта `window.location`:

```
const URL = window.location.href;
```

Обсуждение

В объекте `window.location` хранится неизменяемая информация о текущем URL, или месте размещения документа. В свойстве `href` записан полный URL, включая протокол (например, HTTPS), имя домена, путь к документу и строки запроса — все то, что обычно отображается на панели URL в браузере пользователя:

```
const URL = window.location.href;  
// Вывести https://www.jseverywhere.io/example  
console.log(`The current URL is ${URL}`);
```



Глобальная переменная `location` — то же самое, что `window.location`, но я предпочитаю явно использовать API `window`.

Кроме `href` у объекта `window.location` есть и другие полезные свойства. Если известно, что пользователь уже зашел на сайт, то удобнее сразу получить доступ к пути и параметрам запроса:

```
// Пользователь открыл страницу https://www.jseverywhere.io/example?page=2
```

```
const PATH = window.location.pathname;  
// Вывести /example/  
console.log(`The current path is ${PATH}`);
```

```
const QUERY = window.location.search;  
// Вывести ?page=2  
console.log(`The current query parameter is ${QUERY}`)
```

Вот полный список неизменяемых свойств `window.location`:

- `hash` — значение в URL, отмеченное «решеткой», такое как `#id`;
- `host` — имя домена и порт;
- `hostname` — имя домена;

- `href` — полный URL;
- `origin` — протокол, имя домена и порт;
- `pathname` — путь к документу;
- `port` — номер порта сервера;
- `protocol` — протокол (HTTP или HTTPS);
- `search` — значения строки запроса.

16.6. Перенаправление на другой URL

Задача

Перенаправить пользователя на другую страницу средствами JavaScript.

Решение

Применить метод объекта `window.location`, соответствующий целям перенаправления, — `assign` или `replace`:

```
// Перенаправляем пользователя на новую страницу
// с сохранением истории браузера
window.location.assign('https://www.example.com');
// Перенаправляем пользователя на новую страницу,
// не сохраняя текущую страницу в истории браузера
window.location.replace('https://www.example.com');
```

Метод `window.location.assign` перенаправляет пользователя на другой URL, сохраняя текущую страницу в истории браузера, так что он может вернуться на исходную страницу с помощью кнопки **Назад**. Метод `window.location.replace`, напротив, заменяет текущий URL в истории браузера, тем самым лишая пользователя возможности вернуться на исходную страницу.

Обсуждение

Методы `window.location` позволяют перейти на другой URL средствами JavaScript. Это дает возможность перенаправить пользователя на другую страницу, если того требуют его действия на текущей странице. Наряду с `assign` и `replace` в нашем распоряжении есть и другие методы объекта `window.location`. Вот полный список:

- `.assign()` — перенаправляет браузер пользователя по данному URL;
- `.reload()` — перезагружает страницу;
- `.replace()` — перенаправляет браузер пользователя по данному URL и удаляет текущий документ из истории браузера;
- `toString()` — возвращает текущий URL, представленный в виде строки.

С помощью этих методов вы сможете, применяя JavaScript, управлять переходами со страницы и таким образом обогатить пользовательский интерфейс приложений интерактивной маршрутизацией и другим полезным функционалом. Эти функции полезны не только в процессе разработки — полное перенаправление страницы должно выполняться всякий раз при получении кода статуса 301 (страница перенесена навсегда) и 302 (страница перенесена временно).



В состав популярных фреймворков JavaScript входят библиотеки маршрутизации, либо эти фреймворки могут быть расширены за счет сторонних библиотек маршрутизации, чтобы обеспечить более надежную маршрутизацию на стороне клиента.

16.7. Копирование текста в буфер обмена

Задача

Сделать так, чтобы приложение позволяло копировать в буфер обмена текст, например общедоступную ссылку.

Решение

Чтобы скопировать текст в буфер обмена на устройстве пользователя, нужно поместить его в элемент `input` или `textarea`, а затем скопировать оттуда с помощью метода `navigator.clipboard.writeText`.

Поместите в HTML-код форму с кнопкой отправки. В следующем примере я явно присваиваю значение элементу `input`. Однако это значение может быть присвоено пользователем либо динамически определено в коде:

```
<input type="text" id="copy-text" value="https://example.com/share/12345">
<button id="copy-button">Copy To Clipboard</button>
```

В сопровождающем коде JavaScript нужно создать обработчик события для элемента `button`. При нажатии кнопки мы с помощью метода `select` получаем текст из элемента `input`, после чего с помощью метода `navigator.clipboard.writeText()` копируем его в буфер обмена, как показано в примере 16.2.

Пример 16.2. Копирование текста в буфер обмена

```
const copyText = document.getElementById('copy-text');
const copyButton = document.getElementById('copy-button');

const copyToClipboard = () => {
  copyText.select();
  navigator.clipboard.writeText(copyText.value);
};

copyButton.addEventListener('click', copyToClipboard);
```

Обсуждение

Вставка текста из поля ввода в буфер обмена на устройстве клиента — одна из типичных функций пользовательского интерфейса. Она встречается в таких веб-приложениях, как GitHub и Google Docs. Данное полезное свойство позволяет упростить обмен URL и другой информацией между пользователями. Наиболее часто реализуется сценарий применения с полем ввода и кнопкой, продемонстрированный в предыдущем примере, но бывают случаи, когда нужно скопировать выделенный пользователем фрагмент страницы. В этой ситуации может быть удобно скрыть элементы управления формой. Для этого задействуется разметка страницы в сочетании с элементом `textarea` или `input`. В следующем примере использован элемент `textarea`, который благодаря атрибуту `tabindex` исключен из последовательности перебора элементов формы. А благодаря атрибуту `aria-hidden` со значением `true` его будут игнорировать программы чтения с экрана:

```
<p>Some example text<p>

<textarea id="copy-text" tabindex="-1" aria-hidden="true"></textarea>
<button id="copy-button">Copy the Highlighted Text</button>
```

В таблице стилей CSS я скрыл элемент `textarea`, разместив его за пределами экрана, а также присвоив ему нулевую высоту и ширину:

```
#copy-text {
  position: absolute;
  left: -9999px;
  height: 0;
  width: 0;
}
```

Наконец, в коде JavaScript я реализовал сценарий, подобный показанному в примере 16.2, но добавил метод `document.getSelection()`, чтобы получить значение любого текста, выделенного пользователем на странице:

```
const copyText = document.getElementById('copy-text');
const copyButton = document.getElementById('copy-button');

const copyToClipboard = () => {
  const selection = document.getSelection();
  copyText.value = `${selection} — Check out my highlight
    at https://example.com`;
  copyText.select();
  navigator.clipboard.writeText(copyText.value);
}

copyButton.addEventListener('click', copyToClipboard);
```

Возможность легко передавать содержимое веб-приложений стала особенно востребованной в эпоху социальных сетей. Применение этих технологий значительно упрощает данный вид взаимодействия.

16.8. Вывод на стационарном компьютере таких же уведомлений, как на мобильном устройстве

Задача

Сообщать пользователю о событиях, а также о завершении длительных процессов, даже если вкладка с сайтом неактивна.

Решение

Задействовать Web Notifications API.

Этот API предоставляет относительно простой способ выводить всплывающее окно с уведомлением вне браузера, так что человек, просматривающий веб-страницу на другой вкладке, все равно увидит это уведомление.

Для применения Web Notification необходимо получить разрешение. В следующем коде пользователь запрашивает разрешение для Web Notification, нажимая кнопку. Если разрешение дано, то выводится уведомление:

```
const notificationButton = document.getElementById('notification-button');

const showNotification = permission => {
  // Если пользователь не предоставил разрешение, выходим из функции
  if (permission !== 'granted') return;

  // Содержимое уведомления
  const notification = new Notification('Title', {
    body: 'Check out this super cool thing'
  });

  // Дополнительно можно указать действие, которое
  // должно выполняться при нажатии на уведомлении
  notification.onclick = () => {
    window.open('https://example.com');
  };
};

const notificationCheck = () => {
  // Если уведомления не поддерживаются, — выйти
  // Если поддерживаются, то можно выполнить дополнительное действие
  // Например, перенаправить пользователя
  // на страницу регистрации по электронной почте
  if (!window.Notification) return;

  // Запрашиваем разрешение у пользователя
  Notification.requestPermission().then(showNotification);
};

// В ответ на щелчок кнопкой мыши вызываем функцию `notificationCheck`
notificationButton.addEventListener('click', notificationCheck);
```

Обсуждение

На мобильных устройствах можно получать уведомления о том, что кто-то поставил лайк под вашим постом в Facebook или что вам пришло письмо по электронной почте. На стационарных компьютерах такой возможности обычно нет, хотя, возможно, она бы не помешала.

Поскольку мы создаем все более сложные веб-приложения, такая функциональность была бы очень кстати, особенно для тех операций, выполнение которых занимает значительное время. Это гораздо лучше, чем заставлять пользователя зря сидеть у экрана и пялиться на значок «в процессе», — он мог бы в это время просмотреть другие страницы на других вкладках, а потом вернуться, получив уведомление о завершении длительной операции.

В предлагаемом решении при создании первого уведомления мы получаем разрешение от посетителя веб-страницы. Если наше приложение — это самостоятельное веб-приложение, то можно определить разрешения в файле манифеста, но в случае веб-страниц необходимо запрашивать разрешение у пользователя.

Перед запросом разрешения на вывод уведомлений можно также проверить, поддерживаются ли уведомления в принципе, и если нет, выдать сообщение об ошибке:

```
if (window.Notification) {
  Notification.requestPermission(() => {
    setTimeout(() => {
      const notification = new Notification('hey wake up', {
        body: 'your process is done',
        tag: 'loader',
        icon: 'favicon.ico'
      });
      notification();
    }, 5000);
  });
}
```

Notification принимает два аргумента — строку заголовка и объект со следующими параметрами:

- **body** — текст сообщения, которое выводится в теле уведомления;
- **tag** — тег, упрощающий идентификацию уведомлений в случае глобальных изменений;
- **icon** — нестандартный значок;
- **lang** — язык уведомления;
- **dir** — каталог для параметров языка.

Для Notification можно создать также обработчики следующих событий:

- onerror;
- onclick;
- onshow;
- onclose.

Наконец, можно программно закрыть уведомление с помощью метода `Notification.close()`, хотя в Safari и Firefox уведомления закрываются сами через несколько секунд. Во всех браузерах окно уведомления можно закрыть с помощью размещенного в окне уведомления значка «×».

Дополнительно: совместное использование Web Notifications и Page Visibility API

Чтобы уведомления выводились только тогда, когда посетитель находится на странице, к Web Notifications можно добавить Page Visibility API. Page Visibility API широко поддерживается в современных браузерах. В этом API определено только одно событие — `visibilitychange`, которое возникает, когда изменяется видимость вкладки со страницей. В нем определены также несколько новых свойств — `document.hidden` возвращает `true`, если вкладка страницы закрыта, а `document.visibilityState` принимает одно из следующих четырех значений:

- `visible` — когда вкладка со страницей открыта;
- `hidden` — когда вкладка со страницей скрыта;
- `prerender` — страница формируется, но еще невидима (поддерживается не всеми браузерами);
- `unloaded` — страница выгружается из памяти (поддерживается не всеми браузерами).

Для того чтобы уведомление появлялось только в том случае, если вкладка со страницей скрыта, нужно вставить в код приведенного ранее решения проверку значения `visibilityState`:

```
if (window.Notification) {
  Notification.requestPermission(() => {
    setTimeout(() => {
      if (document.visibilityState === 'hidden') {
        const notification = new Notification('hey wake up', {
          body: 'your process is done',
          icon: 'favicon.ico'
        });
        notification();
      } else {
```

```

        document.getElementById('result').innerHTML = 'your process
                                                    is done';
    }
    }, 5000);
  });
}

```

Теперь, прежде чем создавать уведомление, программа будет проверять, скрыта ли страница, и создавать уведомление, только если скрыта. Если же страница активна, то сообщение будет выводиться на самой странице.

16.9. Открытие в браузере файла с локального устройства

Задача

Открыть в браузере файл с изображением и вывести метаданные этого файла.

Решение

Использовать File API:

```

const inputElement = document.getElementById('file');

function handleFile() {
  // Читаем содержимое файла
  const file = this.files[0];
  const reader = new FileReader();
  // Создаем обработчик события load
  reader.addEventListener('load', event => {
    // После того как файл загружен, делаем что-то с его содержимым
  });
  reader.readAsDataURL(file);
}

inputElement.addEventListener('change', handleFile, false);

```

Обсуждение

File API привязывается к типу файлов, указанному в элементе формы, который отвечает за загрузку файлов. File API позволяет не только загружать файлы на сервер из размещенной на странице формы, но и обеспечивает доступ к загруженным файлам непосредственно в коде JavaScript — либо для локальной обработки, либо для последующей передачи данных на сервер.



Подробнее о FileReader читайте на веб-странице MDN, посвященной File API (<http://mzl.la/1ya0o1k>), и в соответствующем учебном пособии (<http://mzl.la/1ya0qGs>).

В File API определены три объекта:

- **FileList** — список файлов, загруженных с помощью элемента формы `input type="file"`;
- **File** — информация о конкретном файле;
- **FileReader** — объект, который асинхронно загружает файл для доступа на стороне клиента.

У каждого из них есть свои свойства и события, включая возможность отслеживать ход загрузки файла, создавать индикатор загрузки и сообщать об окончании процесса. Объект **File** предоставляет информацию о файле, включая имя, размер и тип MIME. В объекте **FileList** хранится список объектов **File**, поскольку, если у элемента `input` есть атрибут `multiple`, могут загружаться несколько файлов. Объект **FileReader** выполняет собственно загрузку файла.

В примере 16.3 показано приложение, которое загружает изображение, размещает его на веб-странице и выводит некую информацию о нем. Результат работы приложения показан на рис. 16.7.

Пример 16.3. Загрузка изображения с метаданными

```
<!DOCTYPE html>
<head>
  <title>Image Reader</title>
  <meta charset="utf-8" />
  <style>
    #result {
      width: 500px;
      margin: 30px;
    }
  </style>
</head>
<body>
  <h1>Image Reader</h1>
  <form>
    <label for="file">File:</label> <br />
    <input type="file" id="file" accept=".jpg, .jpeg, .png" />
  </form>
  <div id="result">
    <ul>
      <li>Image name: <span id="name"></span></li>
      <li>Image type: <span id="type"></span></li>
    </ul>
  </div>

  <script>
    const inputElement = document.getElementById('file');
    const result = document.getElementById('result');
    const nameEl = document.getElementById('name');
    const typeEl = document.getElementById('type');
    function handleFile() {
      // Читаем содержимое файла
```

```

const file = this.files[0];
const reader = new FileReader();
// Создаем обработчик события load
reader.addEventListener('load', event => {
    // Создаем в элементе div с id="result" элемент img
    // и выводим в нем изображение
    const img = document.createElement('img');
    img.setAttribute('src', event.target.result);
    img.setAttribute('width', '250');
    result.appendChild(img);
    // Выводим имя и тип файла изображения
    const name = document.createTextNode(file.name);
    const type = document.createTextNode(file.type);
    nameEl.appendChild(name);
    typeEl.appendChild(type);
});
reader.readAsDataURL(file);
}

inputElement.addEventListener('change', handleFile, false);
</script>
</body>

```

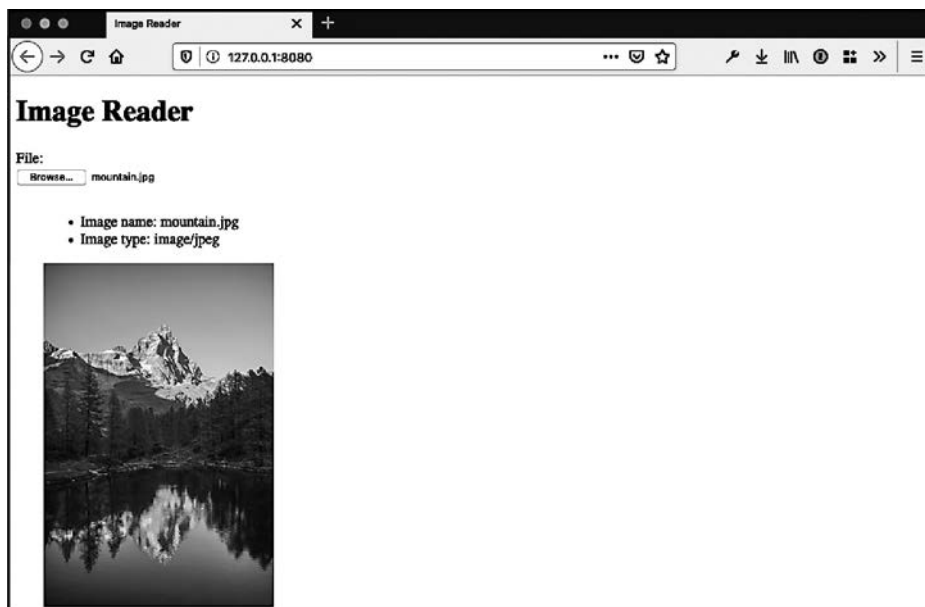


Рис. 16.7. Чтение изображения с помощью File API



File API создан усилиями W3C. Подробнее об этом API читайте в последней версии проекта (<http://w3.org/TR/FileAPI>) или в документации Mozilla (<http://mzl.la/1ya0qGs>).

16.10. Расширение возможностей с помощью Web Components

Задача

Нам нужен компонент с определенным внешним видом и поведением, который бы размещался на странице как обычный HTML-элемент, но без применения веб-фреймворка.

Решение

Обратите внимание на технологии Web Components, которые позволяют создавать и многократно использовать собственные HTML-элементы. В состав Web Components входят шаблон (Template), новые элементы и теневой DOM. Мы подробно обсудим каждый из них в подразделе «Обсуждение».

Обсуждение

Рассмотрим виджет страницы, который должен быть полностью автономным и который если и может иметь некоторое сходство с Web Components, то разве что поверхностное. Понятие Web Components включает в себя несколько различных конструкций. В следующих разделах я опишу каждую из них и приведу примеры, мы рассмотрим полизаполнения и обсудим, чего стоит ожидать от Web Components в будущем.

HTML-шаблоны

Элемент `template` включен в спецификацию HTML5. Он уже поддерживается большинством современных браузеров (<https://oreil.ly/SJZDC>). Внутри элемента `template` можно размещать HTML-код, который мы хотим сгруппировать и затем использовать как единое целое. Этот элемент не создается до тех пор, пока не будет *клонирован*. При загрузке браузер анализирует данный элемент, чтобы убедиться в его корректности, но сам элемент еще не существует — пока что.

Работать с шаблонами легко — они интуитивно понятны. Рассмотрим типичную для современных одностраничных приложений JavaScript задачу: получить данные от веб-сервиса и представить их в виде нумерованного списка (`ul`) (или абзаца, или таблицы, или еще чего-то). Обычно для этого используются методы DOM: мы обращаемся к существующему элементу `ul`, создаем элементы списка (`li`), вставляем в них текст, а сами элементы — в список.

Нельзя ли пропустить некоторые из этих операций? Можно — если применить шаблон. Рассмотрим следующий HTML-код:

```
<template id="hello-world">
  <p>Hello world!</p>
</template>
```

А вот код JavaScript, позволяющий разместить на странице шаблон **Hello World**:

```
const template = document.getElementById('hello-world');
const templateContent = template.content;
document.body.appendChild(templateContent);
```

В этом примере мы обращаемся к элементу `template`, получаем доступ к его содержимому и вставляем это содержимое в HTML-документ с помощью метода `appendChild()`. Как я уже упоминал, шаблоны интуитивно понятны. Вы спросите: в чем тогда смысл? Мы лишь увеличили количество обрабатываемого кода в изначально довольно простом процессе. Однако шаблоны важны при использовании специальных элементов и технологии теневой модели DOM, которые обсуждаются далее.

Специальные элементы

Из всех концепций Web Components самый большой интерес вызывают специальные элементы. Вместо того чтобы оперировать уже существующими HTML-элементами со стандартным поведением и внешним видом, можно создать специальный элемент с собственными стилями и поведением и разместить его на веб-странице. Специальный элемент может либо расширять возможности уже существующего элемента, либо быть анонимным, то есть совершенно новым. В следующем примере я расширил возможности HTML-элемента `<p>`, создав новый элемент `<hello-world>`. Для него потребовалось определить класс со специальными методами:

```
class CustomGreeting extends HTMLParagraphElement {
  constructor() {
    // В конструкторе всегда сначала вызываем super()
    super();

    // Сюда дописываем любой дополнительный функционал элемента
  }
}
```

После того как класс определен, я могу зарегистрировать свой элемент. Обратите внимание: имя элемента должно содержать дефис, чтобы не конфликтовать с уже существующими HTML-элементами:

```
customElements.define("custom-greeting", CustomGreeting);
```

Теперь можно использовать созданный элемент на HTML-странице:

```
<custom-greeting>Hello world!</custom-greeting>
```

Теневой DOM

Каждый раз, говоря о *теневом* DOM, я вспоминаю супергероя Тень. Замечательный персонаж, к тому же очень подходящий к теме. Только Тень знал, какое зло

кроется в умах людей, и только теневой DOM знает, что скрывается в DOM-структуре его элементов.

Но — даже если не отвлекаться на супергеройские истории — теневой DOM остается самым неуловимым из Web Components. Зато и самым интригующим.

Начнем с неволебного. Теневой DOM — это все равно DOM, дерево узлов, подобное тому, которое мы использовали для получения доступа к дочерним элементам элемента `document`. Главное отличие состоит в том, что теневого DOM не существует в том смысле, в котором мы привыкли думать о существовании DOM. Теневой DOM начинает существовать тогда, когда мы создаем теневой корень элемента. Но с этого момента у элемента пропадает все, что было раньше. Вот главное, что следует помнить о теневом DOM: в момент своего создания теневой DOM заменяет существовавший ранее DOM элемента.

Для того чтобы прикрепить к элементу теневой корень, используется метод `attachShadow`:

```
const shadow = element.attachShadow({mode: 'open'});
```

Метод `attachShadow` принимает параметр (`mode`), который принимает одно из двух значений: `open` или `closed`. Если `mode` равен `open`, то теневой DOM доступен из контекста страницы, как любой другой элемент. Чаще всего теневой DOM прикрепляется к специальному элементу в его конструкторе:

```
class CustomGreeting extends HTMLElement {
  constructor() {
    super();
    const shadow = this.attachShadow({mode: 'open'});
    const greeting = this.getAttribute('greeting') || 'world'
    shadow.innerHTML = `<p>
      Hello, <span class="greeting">${greeting}</span>
    </p>`;
  }
}
```

В приведенном примере содержатся два HTML-элемента, но глобальные стили CSS к ним применяться не будут. Для того чтобы создать стили для специального элемента с теневым DOM, необходимо создать в классе специального элемента элемент `style` и применить определенные в нем стили:

```
class CustomGreeting extends HTMLElement {
  constructor() {
    super();
    const shadow = this.attachShadow({mode: 'open'});
    const greeting = this.getAttribute('greeting') || 'world'
    shadow.innerHTML = `<p class="wrapper">
      Hello, <span class="greeting">${greeting}</span>
    </p>`;

    // Добавляем стили CSS
  }
}
```

```

const style = document.createElement('style');

style.textContent = `
  .wrapper {
    color: pink;
  }
  .greeting {
    color: green;
    font-weight: bold;
  }
`;

```



Для работы с веб-компонентами существует подборка библиотек и утилит Polymer Project (<https://oreil.ly/874AX>).

Веб-компоненты — очень интересная часть экосистемы веб-стандартов, у нее огромный потенциал. HTML-шаблоны, специальные HTML-элементы и теневого DOM позволяют создавать небольшие компоненты пользовательского интерфейса для многократного применения. Данная концепция облегченных компонентов нашла отражение в таких библиотеках JavaScript, как React и Vue.

16.11. Выбор фреймворка для разработки на стороне клиента

Задача

Мы строим сложное веб-приложение, для которого нужен фреймворк JavaScript. Как правильно выбрать его?

Решение

Было время, когда фреймворки JavaScript появлялись и выходили из моды быстрее, чем модели на подиуме. К счастью, в последние несколько лет война фреймворков несколько утихла, и в нашем распоряжении оказалось несколько отличных вариантов. Несмотря на замедление новых разработок, все равно бывает трудно выбрать фреймворк, который лучше всего подходил бы для вас и вашего проекта. При выборе фреймворка для проекта я рекомендую задать себе следующие вопросы.

- *Действительно ли мне нужен фреймворк JavaScript?* Не хватайтесь сразу за фреймворк. Очень часто простые сайты и приложения проще написать

и поддерживать без всякого фреймворка, при этом с точки зрения пользователя они будут работать быстрее.

- *Проект какого типа я разрабатываю?* Это ваш личный проект? Проект для клиента? Корпоративный проект, который потом будет долго поддерживаться? Проект с открытым кодом? Подумайте о том, что выбранное вами должно быть наиболее удобным для тех, кто будет позже поддерживать ваш проект.
- *Насколько обширно сообщество фреймворка и как долго оно существует?* Обратите внимание на то, давно ли поддерживается фреймворк. Эта поддержка все еще активна? У фреймворка большое сообщество, которое его поддерживает, отвечает на вопросы и исправляет ошибки?
- *Хорошо ли документирован фреймворк?* Убедитесь, что документация фреймворка полна и написана понятным языком.
- *Что представляет собой экосистема разработки фреймворка?* Посмотрите, какие существуют утилиты, плагины и метафреймворки.
- *Мне знаком этот фреймворк?* Вы уже имели дело с этим фреймворком или вам только предстоит его изучить?
- *Как это повлияет на пользователей?* Это, вероятно, самый важный из всех вопросов. Исследуйте, как фреймворк повлияет на производительность, доступность и удобство применения продукта.

Следующий список далеко не полон, но мы рекомендуем обратить внимание на фреймворки React, Vue, Svelte и Angular.

React

React (<https://reactjs.org>) — это JavaScript-фреймворк для пользовательского интерфейса, разработанный и выпускаемый Facebook. В React основной упор сделан на мелкие визуальные компоненты. Для визуализации HTML-компонентов здесь в основном используется XML-синтаксис `jsx` и JavaScript. Благодаря React обновление страниц становится более эффективным, так как задействуется представление DOM, известное как виртуальный DOM (<https://oreil.ly/oK21x>).

Vue

Vue (<https://vuejs.org>) — это фреймворк для пользовательского интерфейса, имеющий обширное сообщество поддержки. Подобно React, Vue задействует виртуальный DOM, благодаря чему страницы обновляются практически мгновенно. Многие рассматривают Vue как альтернативу React. Оба фреймворка имеют похожий функционал, но в Vue применяется более близкий к HTML синтаксис шаблонов. Кроме того, Vue поддерживается не командой Facebook, а сообществом разработчиков. Я бы рекомендовал попробовать и React, и Vue и лишь потом решить, какой из них больше соответствует вашему стилю и стилю вашей команды.

Svelte

В отличие от остальных рассмотренных здесь JS-фреймворков, Svelte (<https://svelte.dev>) построен на других принципах. Подобно React и Vue, это библиотека с упором на пользовательский интерфейс, однако здесь основная работа выполняется не в браузере. В Svelte основное внимание уделяется компиляции кода в процессе разработки. Цель фреймворка — сократить нагрузку на браузер, что позволяет разработчикам создавать высокопроизводительные приложения.

Angular

Angular (<https://angular.io>) — это полнофункциональный фреймворк JavaScript, разработанный и выпущенный Google. Angular пережил первую волну «войны фреймворков» и перестроился на компонентно ориентированную архитектуру, свойственную большинству современных библиотек. Для многих команд разработчиков, особенно корпоративных, этот фреймворк может оказаться полезным инструментом, так как он сужает круг принимаемых решений при построении новых приложений и подключении новых функций.

ЧАСТЬ III

Node.js

ГЛАВА 17

Основы Node

Разделение JavaScript на «старый» и «новый» произошло благодаря появлению на свет Node.js (далее будем называть его просто Node). Конечно, здесь важную роль сыграли и возможность динамического изменения элементов страницы, и усилия по выпуску новых версий ECMAScript. Но именно Node заставил нас посмотреть на JavaScript под совершенно иным углом. И мне этот новый угол нравится — я большой поклонник Node и разработки серверных продуктов на JavaScript.

В этой главе мы изучим основы Node. Для этого вам понадобится как минимум установить Node, как описано в рецептах 1.6 и 17.1.

17.1. Управление версиями Node с помощью Node Version Manager

Задача

Установить на компьютере для разработки несколько версий Node и управлять ими.

Решение

Воспользоваться Node Version Manager (NVM) (<https://github.com/nvm-sh/nvm>), который позволяет установить и применять любую распространяемую версию Node — по одной для каждой оболочки. NVM совместим с Linux, macOS и Windows Subsystem для Linux.

Для того чтобы установить NVM, нужно запустить установочный скрипт из `curl` или `wget` в системном терминале:

```
## с помощью curl:
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.37.2/install.sh | bash
```

```
## с помощью wget:
wget -qO- https://raw.githubusercontent.com/nvm-sh/nvm/v0.37.2/install.sh | bash
```



При разработке в Windows рекомендуется задействовать утилиту `nvm-windows` (<https://github.com/coreybutler/nvm-windows>). Она является самостоятельным проектом, не связанным с NVM, но предоставляет аналогичный функционал для операционной системы Windows. Инструкции по применению `nvm-windows` вы найдете в документации проекта.

После установки NVM нужно поставить версию Node. Для того чтобы установить последнюю версию Node, необходимо выполнить следующую команду:

```
$ nvm install node
```

Или можно установить определенную версию Node:

```
# Установить последний основной выпуск главной версии
$ nvm install 15
```

```
# Установить определенный главный/промежуточный выпуск или патч
$ nvm install 15.6.0
```

После установки Node нам понадобится указать версию, которая будет использоваться по умолчанию для последующих версий оболочки. Это может быть либо последняя установленная версия Node, либо версия с определенным номером:

```
# По умолчанию новые сессии оболочки будут открываться
# для последней версии Node
nvm alias default node
# По умолчанию новые сессии оболочки будут открываться
# для определенной версии Node
nvm alias default 14
```

Для того чтобы перейти на другую версию в той же сессии оболочки, нужно воспользоваться командой `nvm use`, указав номер установленной версии:

```
$ nvm use 15
```

Обсуждение

NVM позволяет легко загружать версии Node и переключаться между разными версиями в рамках одной операционной системы. Это необычайно удобно во время работы с библиотеками, которые поддерживают разные версии Node, и устаревшими кодовыми базами. При этом также упрощается управление Node в среде разработки. Список всех выпусков и сроки их поддержки вы найдете по ссылке <https://oreil.ly/9IY83>.

Благодаря NVM (с помощью команды `nvm ls`) можно вывести список всех версий Node, установленных на данном компьютере. В него войдут все установленные версии, версия, используемая по умолчанию для всех новых сессий, а также все версии LTS, которые не были установлены:

```
$ nvm ls
   v8.1.2
   v8.11.3
   v10.13.0
->  v10.23.1
   v12.8.0
   v12.20.0
   v12.20.1
   v13.5.0
   v14.14.0
   v14.15.1
   v14.15.4
   v15.6.0
   system
default -> 14 (-> v14.15.4)
node -> stable (-> v15.6.0) (default)
stable -> 15.6 (-> v15.6.0) (default)
iojs -> N/A (default)
unstable -> N/A (default)
lts/* -> lts/fermium (-> v14.15.4)
lts/argon -> v4.9.1 (-> N/A)
lts/boron -> v6.17.1 (-> N/A)
lts/carbon -> v8.17.0 (-> N/A)
lts/dubnium -> v10.23.1
lts/erbium -> v12.20.1
lts/fermium -> v14.15.4
```

Как видите, на моем компьютере установлены несколько устаревших патчей основных выпусков. Для того чтобы деинсталлировать и удалить определенную версию, можно воспользоваться командой `nvm uninstall`:

```
nvm uninstall 14.14
```

Отследить, для какой версии Node написан проект, бывает непросто. Для того чтобы упростить эту задачу, можно создать в корневом каталоге проекта файл `.nvmrc`. В нем хранится номер версии Node, которую поддерживает проект, например:

```
# По умолчанию используется последняя версия LTS
$ lts/*

# Для того чтобы использовать определенную версию
$ 14.15.4
```

Для того чтобы задействовать версию, указанную в файле проекта `.nvmrc`, нужно перейти в корневой каталог проекта и выполнить оттуда команду `nvm use`.



Для крупных проектов, в которых используются контейнерные технологии, такие как Docker, чрезвычайно важно обеспечить соответствие версий в рамках всей среды, в том числе при развертывании. В документации Node есть полезное руководство по управлению веб-приложениями Node.js в Docker (<https://oreil.ly/phXQZ>).

17.2. Ответ на простой запрос браузера

Задача

Создать приложение Node, которое отвечало бы на простейший запрос браузера.

Решение

Для ответа на запросы можно использовать HTTP-сервер, встроенный в Node:

```
// Загружаем модуль http
const http = require('http');

// Создаем http-сервер
http
  .createServer((req, res) => {
    // Заголовок content
    res.writeHead(200, { 'content-type': 'text/plain' });

    // Выводим сообщение и заканчиваем обмен сигналами
    res.end('Hello, World!');
  })
  .listen(8124);

console.log('Server running on port 8124');
```

Обсуждение

Данное приложение Node выдает на запрос браузера ответ веб-сервера «Hello, World». В этом примере видно не только то, как работает приложение Node, но и то, что для обмена данными с ним используется традиционный метод коммуникации — отправка запроса на веб-ресурс.

В первой строке с помощью функции Node `require()` загружается модуль `http`. Эта функция дает системе модулей Node команду загрузить определенную библиотеку, которая будет использоваться в приложении. Модуль `http` — один из многих модулей, которые входят в стандартный инсталляционный пакет Node.

Затем с помощью метода `http.createServer()` создается HTTP-сервер. В этот метод передается анонимная функция, известная как `RequestListener`, с двумя параметрами. Node подключает ее к обработчику событий, перехватывающему запросы сервера. Два параметра функции — это *запрос* и *ответ*. Запрос — это экземпляр объекта `http.IncomingMessage`, а ответ — экземпляр объекта `http.ServerResponse`.

Объект `http.ServerResponse` используется для ответа на веб-запросы. В объекте `http.IncomingMessage` содержится информация о запросе, такая как его URL. Если нужно извлечь из URL определенные элементы информации, например параметры строки запроса, то можно выполнить синтаксический анализ строки URL с помощью вспомогательного модуля Node `url`.

В примере 17.1 показано, как с помощью строки запроса можно возвращать в браузер более индивидуализированные сообщения.

Пример 17.1. Синтаксический анализ строки с данными запроса

```
// Загружаем модуль http
const http = require('http');
const url = require('url');

// Создаем http-сервер
http
  .createServer((req, res) => {
    // Получаем строку запроса с его параметрами
    const { query } = url.parse(req.url, true);

    // Заголовок content
    res.writeHead(200, { 'content-type': 'text/plain' });

    // Передача сообщений и сигналов завершена
    const name = query.first ? query.first : 'World';

    // Передаем сообщение и информируем, что обмен данными завершен
    res.end(`Hello, ${name}!`);
  })
  .listen(8124);

console.log('Server running on port 8124');
```

Если передать такой URL:

```
http://localhost:8124/?first=Reader
```

то на веб-странице появится текст **Hello, Reader!**.

В этом коде в объекте модуля `url` определен метод `parse()`, который анализирует URL и возвращает отдельные его компоненты (`href`, `protocol`, `host` и т. д.). Если в качестве второго аргумента передать `true`, то строка будет проанализирована еще одним модулем — `querystring`. Вместо обычной строки `querystring` возвращает объект, свойствами которого являются отдельные параметры URL.

И в предложенном решении, и в примере 17.1 текстовое сообщение выводится на страницу с помощью метода `end()` объекта `http.ServerResponse`. Я мог бы вывести его, используя `write()`, и затем вызвать `end()`:

```
res.write(`Hello, ${name}!`);
res.end();
```

Важным выводом из обоих примеров является то, что после формирования всех заголовков и тела ответа сервера всегда необходимо вызывать метод `end()`.

К функции `createServer()` может прикрепляться цепочка других функций — в данном случае это вызов функции `listen()`. В нее передается номер порта сервера, по которому должны поступать запросы. Этот номер порта — еще один чрезвычайно важный компонент приложения.

Традиционно по умолчанию для большинства веб-сервисов используется порт 80 (но не для HTTPS — в этом случае порт по умолчанию 443). Если задается порт 80, то в веб-запросах, передаваемых на определенный URL-адрес сервера, указывать порт не нужно. Но есть более традиционный сервер, который тоже использует порт 80 по умолчанию, — Apache. При попытке запустить сервис Node на том же порте, что и Apache, приложение не будет работать. Нужно либо установить приложение Node на отдельном сервере, либо выделить для него другой порт.

Кроме порта можно также определить IP-адрес (хост), и тогда пользователи будут направлять запросы не только на порт, но и на определенный хост. Если не указать хост, то приложение будет отслеживать запросы на все IP-адреса, связанные с данным сервером. Либо можно указать имя домена, которое Node станет преобразовывать в IP-адрес.

У продемонстрированных методов есть и другие аргументы, а у объектов — множество иных методов, но эти примеры помогут вам начать работу. Более подробную информацию вы найдете в документации Node (<http://nodejs.org/api>).

17.3. Интерактивная проверка кода Node с помощью REPL

Задача

Выполнять фрагменты серверного кода, написанного для Node.

Решение

Воспользоваться Node REPL (Read-Evalute-Print-Loop) — интерактивной версией Node с интерфейсом командной строки, из которой можно выполнять любые фрагменты кода.

Для применения REPL введите в командной строке слово `node`, не указывая запускаемое приложение:

```
$ node
```

Затем можно ввести код JavaScript в упрощенном редакторе строк Emacs (к сожалению, vi здесь нет). Можно импортировать библиотеки, создавать функции — делать все то же, что и в статических приложениях. Главное отличие состоит в том, что каждая строка кода интерпретируется немедленно:

```
> const add = (x, y) => { return x + y };
undefined
> add(2, 2);
4
```

Когда закончите, выйдите из программы, введя команду `.exit`:

```
> .exit
```

Обсуждение

REPL можно запускать как отдельно, так и из другого приложения, чтобы снабдить последнее определенными функциями. Код JavaScript вводится так же, как если бы вы набирали скрипт в текстовом файле. Главное поведенческое отличие состоит в том, что REPL позволяет увидеть результаты, такие как значение `undefined`, сразу после ввода строки.

При этом можно импортировать модули:

```
> const fs = require('fs');
```

Можно также обращаться к глобальным объектам — мы только что так и сделали, когда использовали `require()`.

Значение `undefined`, которое появляется при вводе кода, — это значение, возвращаемое после выполнения предыдущей строки кода. JavaScript возвращает `undefined`, в частности, после создания новой переменной или функции. Однако такое поведение вскоре начинает раздражать. Чтобы этого избежать, а также для внесения еще ряда изменений в небольших приложениях Node можно использовать функцию `REPL.start()`. Она активирует REPL (но только с заранее заданными параметрами).

Вот эти параметры:

- `prompt` — изменяет экранную подсказку (по умолчанию это `>`);
- `input` — изменяет входной поток для чтения данных (по умолчанию это стандартный входной поток `process.stdin`);
- `output` — изменяет выходной поток для записи результатов (по умолчанию это стандартный выходной поток `process.stdout`);
- `terminal` — принимает значение `true`, если поток должен обрабатываться как TTY и в него должны записываться управляющие коды ANSI/VT100;
- `eval` — функция, используемая вместо асинхронной функции `eval()` для выполнения кода JavaScript;
- `useColors` — принимает значение `true`, если нужно задать цвет вывода результатов для функции, указанной в параметре `writer` (по умолчанию это стандартные цвета терминала);
- `useGlobal` — принимает значение `true`, если нужно использовать глобальный объект, а не выполнять все скрипты в выделенном контексте;
- `ignoreUndefined` — принимает значение `true`, чтобы запретить возврат значений `undefined`;

- **writer** — функция, которая возвращает и выводит на экран отформатированный результат выполненного кода (по умолчанию это `util.inspect`).

Далее показан пример приложения, в котором REPL запускается с измененной экранной подсказкой, игнорируются значения `undefined` и изменены цвета:

```
const repl = require('repl');

const options = {
  prompt: '-> ',
  useColors: true,
  ignoreUndefined: true
};

repl.start(options);
```

Нужные нам параметры определяются в объекте `options` и затем передаются в функцию `repl.start()`. Одновременно с приложением запускается и REPL, но теперь не приходится обрабатывать значения `undefined`:

```
-> const add = (x, y) => { return x + y };
-> add(2, 2);
4
```

Как видим, результат получается более простым, без всех этих `undefined`, засоряющих вывод.

Дополнительно: постойте, что за глобальные объекты?

Вы это заметили?

Различие между JavaScript в Node и JavaScript в браузере состоит в глобальной области видимости. В браузере переменная, созданная внутри функции с помощью ключевого слова `var`, традиционно относится к глобальному объекту высшего уровня, известному как `window`:

```
var test = 'this is a test';
console.log(window.test); // 'this is a test'
```

Аналогично переменные, созданные в браузере с помощью ключевого слова `let` или `const`, также имеют глобальную область видимости, хотя и не привязаны к объекту `window`.

В Node каждый модуль действует в рамках собственного контекста, так что одни и те же переменные, объявленные в разных модулях и используемые в одном и том же приложении, не конфликтуют между собой.

Однако существуют объекты, доступные в Node посредством глобального объекта `global`. В предыдущих примерах мы уже применяли некоторые из них, например `console`, объект `Buffer` и `require()`. Это также наши старые знакомые `setTimeout()`, `clearTimeout()`, `setInterval()` и `clearInterval()`.

17.4. Чтение данных из файла и запись данных в файл

Задача

Прочитать данные из файла и записать данные в файл, который хранится на локальном устройстве.

Решение

В ядре Node реализован функционал управления файловой системой Node. Он находится в модуле `fs`:

```
const fs = require('fs');
```

Для того чтобы прочитать содержимое файла, нужно воспользоваться функцией `readFile()`:

```
const fs = require('fs');

fs.readFile('main.txt', 'utf8', (err, data) => {
  if (err) throw err;
  console.log(data);
});
```

Для записи данных в файл применяется функция `writeFile()`:

```
const fs = require('fs');

const buf = "I'm going to write this text to a file";
fs.writeFile('main2.txt', buf, err => {
  if (err) throw err;
  console.log('wrote text to file');
});
```

Функция `writeFile()` перезаписывает существующий файл. Для того чтобы дописать текст в конец файла, используется функция `appendText()`:

```
const fs = require('fs');

const buf = "\nI'm going to add this text to a file";
fs.appendFile('main.txt', buf, err => {
  if (err) throw err;
  console.log('appended text to file');
});
```

Обсуждение

Поддержка файловой системы в Node одновременно полнофункциональная и простая в применении. Для того чтобы прочитать данные из файла, используется функция `readFile()`, которая принимает следующие параметры:

- имя файла, включая путь к нему в операционной системе, если этот файл не находится в каталоге приложения;
- объект параметров, включая кодировку (**encoding**), использованную в примере, и **flag**, который по умолчанию равен **r** (от **reading** — «для чтения»);
- функция обратного вызова с параметрами на случай ошибки чтения данных.

В рассмотренном примере, если бы я не указал кодировку приложения, то Node вернул бы содержимое файла в виде буфера с сырыми данными. Но поскольку я указал кодировку, содержимое файла вернулось в виде строки.

Функции **writeFile()** и **appendFile()**, предназначенные соответственно для перезаписи файла и добавления данных в конец файла, принимают те же аргументы, что и **readFile()**:

- имя файла и путь к нему;
- строку или буфер с данными, которые нужно записать в файл;
- объект с параметрами **encoding** (по умолчанию равен **w** для **writeFile()** и **a** для **appendFile()**) и **mode**, который по умолчанию равен **438** (**0666** в восьмеричной записи);
- функцию обратного вызова с единственным аргументом **error**.

Значение **mode** может использоваться для назначения прав доступа к файлу в том случае, если он был создан функцией **writeFile()** или **appendFile()**. По умолчанию создается файл с правом чтения и записи для владельца и только чтения — для всех остальных.

Как я уже упоминал, данные для записи могут быть представлены в виде буфера или строки. Последняя не может содержать двоичные данные, поэтому Node позволяет использовать буфер, который способен оперировать как строками, так и двоичными данными. Во всех описанных в этом разделе функциях для работы с файловой системой можно взять как строки, так и буферы, но если вы хотите применять и то и другое, необходимо явно преобразовывать эти типы данных один в другой.

Например, вместо того чтобы указывать кодировку **utf8** в методе **writeFile()**, можно преобразовать строку в буфер, указав желаемую кодировку:

```
const fs = require('fs');

const str = "I'm going to write this text to a file";
const buf = Buffer.from(str, 'utf8');
fs.writeFile('mainbuf.txt', buf, err => {
  if (err) throw err;
  console.log('wrote text to file');
});
```

Обратное преобразование — из буфера в строку — выполняется так же легко:

```
const fs = require('fs');

fs.readFile('main.txt', (err, data) => {
  if (err) throw err;
  const str = data.toString();
  console.log(str);
});
```

Функция преобразования буфера в строку `toString()` принимает три необязательных параметра: кодировку, в которую нужно выполнить преобразование, с какого места буфера начинать преобразование и где его нужно закончить. По умолчанию выполняется преобразование всего буфера в кодировку UTF8.

Функции `readFile()`, `writeFile()` и `appendFile()` асинхронные — другими словами, скрипт не ожидает их завершения, прежде чем продолжить выполнять код. Это особенно важно, когда речь идет о сравнительно медленных операциях, таких как доступ к файлу. Существуют и синхронные аналоги этих функций — `readFileSync()`, `writeFileSync()` и `appendFileSync()`. Не устану повторять: вам не следует использовать эти синхронные функции. Я упоминаю их здесь лишь для полноты картины.

Дополнительно

Еще один способ чтения файла и записи в него — использовать функцию `open()`, а затем `read()` для чтения содержимого файла или `write()` — для записи. Преимущество такого подхода состоит в более ограниченных возможностях влияния на процесс, а недостаток — в усложнении кода из-за применения всех этих функций, включая возможность использования буфера для чтения и записи.

Функция `open()` принимает следующие аргументы:

- имя файла и путь к нему;
- флаг;
- дополнительный режим;
- функцию обратного вызова.

Для выполнения всех операций используется одна и та же функция `open()`, а конкретная операция указывается посредством флага. Он может задавать довольно много значений, но нас в данный момент больше всего интересуют следующие:

- `r` — открыть файл для чтения (файл должен быть создан ранее);
- `r+` — открыть файл для чтения и записи. Если файла не существует, то выбрасывается исключение;

- `w` — открыть файл для записи, обрезать файл либо создать его, если такого файла не существует;
- `wx` — открыть файл для записи. Если файла не существует, возникает ошибка;
- `w+` — открыть файл для чтения и записи. Если файла не существует, то он создается, если существует — обрезается;
- `wx+` — то же самое, что и `w+`, но выдать ошибку, если файл существует;
- `a` — открыть файл для дополнения, если файла не существует — создать;
- `ax` — открыть файл для дополнения, если файла не существует — выдать ошибку;
- `a+` — открыть файл для чтения и дополнения; если файла не существует — создать;
- `ax+` — то же, что и `a+`, но ошибка выдается, если файл существует.

Параметр `mode` имеет тот же смысл, что и в предыдущем случае для функций `writeFile()` и `appendFile()`, — это значение, в котором устанавливаются биты *закрепления* и *полномочий* для создаваемого файла. По умолчанию параметр `mode` равен 0666. Функция обратного вызова принимает два аргумента: объект `Error`, если возникает ошибка, и *дескриптор файла*, который может использоваться в последующих операциях с ним.

Функции `read()` и `write()` принимают одни и те же базовые аргументы:

- дескриптор файла, возвращенный функцией `open()`;
- буфер, используемый для хранения данных — либо тех, которые нужно записать или дописать в файл, либо прочитанных из файла;
- позиция, с которой начинается чтение/запись;
- длина буфера (назначается при чтении, отслеживается при записи);
- позиция в файле, в которой выполняется операция. Если это текущая позиция, то `null`.

Функции обратного вызова для обоих методов принимают три аргумента: объект ошибки, прочтенные (или записанные) байты и буфер.

Есть еще множество параметров настройки. Лучший способ продемонстрировать, как это все работает, — создать законченное приложение Node, которое создает файл для чтения, записывает туда какой-то текст, дописывает другой текст, после чего читает весь этот текст и выводит прочитанное в консоль. Поскольку функция `open()` асинхронная, операции чтения и записи могут выполняться как функции обратного вызова. Обратите на это внимание, когда будете изучать пример 17.2, — это первый шаг к тому, что известно как *ад обратных вызовов*.

Пример 17.2. Демонстрация функций `open`, `read` и `write`

```
const fs = require('fs');

fs.open('newfile.txt', 'a+', (err, fd) => {
  if (err) {
    throw err;
  } else {
    const buf = Buffer.from('The first string\n');
    fs.write(fd, buf, 0, buf.length, 0, (err, written) => {
      if (err) {
        throw err;
      } else {
        const buf2 = Buffer.from('The second string\n');
        fs.write(fd, buf2, 0, buf2.length, buf.length, (err, written2) =>
{
          if (err) {
            throw err;
          } else {
            const length = written + written2;
            const buf3 = Buffer.alloc(length);
            fs.read(fd, buf3, 0, length, 0, err => {
              if (err) {
                throw err;
              } else {
                console.log(buf3.toString());
              }
            });
          }
        });
      }
    });
  }
});
});
```



О том, как справиться с обратными вызовами, рассказывается в рецепте 19.2.

Для того чтобы узнать размеры буферов, я использовал свойство `length` — для буфера оно возвращает количество байтов. Это значение не всегда равно длине хранящейся в буфере строки, но в данном случае такое решение нам подходит.

От бесконечной лесенки отступов мурашки бегут по коже, зато на этом примере видно, как работают функции `open()`, `read()` и `write()`. Именно их сочетание используется внутри методов доступа к файлам `readFile()`, `writeFile()` и `appendFile()`. Функции более высокого уровня лишь упрощают типичные операции, выполняемые с файлами.



О том, что можно сделать с этими неприятными отступами, читайте в рецепте 19.2.

17.5. Получение данных из терминала

Задача

Получить входные данные от пользователя приложения через терминал.

Решение

Применить модуль Node Readline.

Для того чтобы получить данные из стандартного потока ввода, нужно написать примерно такой код:

```
const readline = require('readline');

const rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout
});

rl.question(">>What's your name? ", answer => {
  console.log(`Hello ${answer}`);
  rl.close();
});
```

Обсуждение

Модуль Readline позволяет читать текстовые строки из потока, открытого для чтения. Для этого нужно вначале создать экземпляр интерфейса Readline с помощью функции `createInterface()`, передав ей как минимум потоки для чтения и записи. Нужны оба потока, так как мы будем не только читать текст, но и выводить экранные подсказки. В предложенном решении входной поток — это стандартный входной поток `process.stdin`, а выходной поток — `process.stdout`. Другими словами, мы получаем данные из командной строки и выводим данные тоже в командную строку.

Использованная в этом решении функция `question()` выводит вопрос, а переданная ей в виде аргумента функция обратного вызова обрабатывает ответ. При вызове функции `close()` интерфейс закрывается, а управление входным и выходным потоками возвращается системе.

Также можно создать приложение, которое постоянно отслеживало бы входной поток и выполняло какие-то действия с поступающими данными, до тех пор, пока

не поступит сигнал о завершении работы. Обычно это некая последовательность букв, сигнализирующая о том, что пользователь закончил работу, — например, слово *exit*.

В приложениях этого типа применяются и другие функции `Readline`. Например, `setPrompt()` позволяет изменить экранную подсказку, которая выводится в начале каждой строки, где пользователь должен ввести текст. Посредством `prompt()` подготавливается область ввода, включая изменение экранной подсказки на ту, которая указана в `setPrompt()`. Наконец, `write()` применяется для вывода экранной подсказки. Нам также понадобятся обработчики события `line` — оно активируется при вводе очередной строки текста.

В примере 17.3 представлено полноценное приложение Node, в котором обработка данных, вводимых пользователем, продолжается до тех пор, пока не будет введено слово `exit`. Обратите внимание на то, что в приложении применяется метод `process.exit()`. Эта функция явно прерывает работу приложения Node.

Пример 17.3. Считывание чисел из потока ввода, до тех пор пока пользователь не введет слово `exit`

```
const readline = require('readline');

let sum = 0;

const rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout
});

console.log("Enter numbers, one to a line. Enter 'exit' to quit.");

rl.setPrompt('>> ');
rl.prompt();

rl.on('line', input => {
  const userInput = input.trim();
  if (userInput === 'exit') {
    rl.close();
    return;
  }
  sum += Number(userInput);
  rl.prompt();
});

// Пользователь ввел 'exit'
rl.on('close', () => {
  console.log(`Total is ${sum}`);
  process.exit(0);
});
```

Выполнение этого приложения для нескольких чисел приведет к выводу следующей информации:

```
Enter numbers, one to a line. Enter 'exit' to quit.  
>> 55  
>> 209  
>> 23.44  
>> 0  
>> 1  
>> 6  
>> exit  
Total is 294.44
```

Для вывода экранной подсказки вместо интерфейса `Readline write()` использовал `console.log()`, после чего выведена пустая строка, чтобы отличать вывод данных от ввода.

Читайте также

В главе 19 показано, как передавать и получать аргументы командной строки в приложениях Node.

17.6. Получение пути к выполняемому скрипту

Задача

Приложение должно получить путь к скрипту, выполняемому в данный момент.

Решение

Использовать переменную `__dirname` или `__filename`. Они находятся в области видимости выполняемого модуля:

```
// Выводим имя каталога, в котором находится выполняемый файл  
// пример: /Users/Adam/Projects/js-cookbook/node  
console.log(__dirname);  
  
// Выводим имя выполняемого файла и его каталог  
// пример: /Users/Adam/Projects/js-cookbook/node/example.js  
console.log(__filename);
```

Обсуждение

На первый взгляд кажется, что переменные `__dirname` и `__filename` глобальные, но в действительности они относятся к области видимости своего модуля. Предположим, у нас есть проект со следующей структурой каталогов:

```
example-app
├── index.js
├── dir1
│   ├── example.js
│   └── dir3
│       └── nested.js
```

Если прочитаем переменную `__dirname` из файла `index.js`, то получим путь к корневому каталогу проекта. Но если прочитаем переменную `__dirname` из скрипта в файле `nested.js`, то получим путь к каталогу `dir3`. Это позволяет прочитать путь к выполняемому модулю, а не ограничиваться только родительским каталогом.

Пример полезного практического применения `__dirname` — при создании файла или каталога внутри текущего каталога. В следующем примере скрипт создает в текущем каталоге подкаталог `cache`:

```
const fs = require('fs');
const path = require('path');
const newDirectoryPath = path.join(__dirname, '/cache');

fs.mkdirSync(newDirectoryPath);
```

17.7. Работа с таймерами и циклом событий Node

Задача

В приложении Node нужен таймер, но мы не знаем, какой из таймеров Node использовать и насколько они точны.

Решение

Если точность таймера не особенно важна, то можно создать одиночное событие таймера с помощью `setTimeout()` или повторяющийся таймер с помощью `setInterval()`:

```
setTimeout(() => {}, 3000);

setInterval(() => {}, 3000);
```

При необходимости оба таймера можно отменить:

```
const timer1 = setTimeout(() => {}, 3000);
clearTimeout(timer1);

const timer2 = setInterval(() => {}, 3000);
clearInterval(timer2);
```

Если же требуется таймер с более ограниченным управлением и немедленными результатами, то можно использовать `setImmediate()`. Для этой функции не нужно определять задержку — здесь обратный вызов срабатывает сразу после всех обратных вызовов ввода/вывода, но перед обратными вызовами `setTimeout()` и `setInterval()`:

```
setImmediate(() => {});
```

Этот таймер также может быть остановлен — с помощью функции `clearImmediate()`.

Обсуждение

Поскольку Node написан на JavaScript, код приложения выполняется в одном потоке — он *синхронный*. Однако операции ввода/вывода (input/output, I/O) и доступ к остальным собственным API выполняется *асинхронно*, то есть в отдельном потоке. Способ, которым Node справляется с этим разрывом по времени, называется *циклом событий*.

Когда в коде выполняется операция ввода/вывода, такая как запись фрагмента текста в файл, мы определяем функцию обратного вызова, которая выполнит все, что должно следовать после записи. Тем временем остальная часть кода приложения продолжит выполняться. Приложение не будет дожидаться окончания записи в файл. Когда запись в файл завершится, в Node будет передано событие, сигнализирующее об этом. Это событие будет помещено в очередь на обработку. Node обрабатывает эту очередь событий и, когда доходит до события, сообщаящего о завершении записи в файл, сопоставляет его с функцией обратного вызова и выполняет ее.

Все это напоминает обед в закусочной: мы становимся в очередь, чтобы сделать заказ, и получаем талончик. Затем садимся за столик и ждем, читая газету или листая Twitter. Тем временем заказы на обед попадают в другую очередь, где их обрабатывает работник закусочной. Но это не значит, что все заказы выполняются в той последовательности, в какой поступили. Некоторые из них выполняются дольше других, так как там нужно что-то испечь или поджарить на гриле, а это занимает много времени. Поэтому работник закусочной обрабатывает ваш заказ — подготавливает блюдо, затем ставит его в духовку и устанавливает таймер на то время, за которое еда должна приготовиться, а тем временем занимается другими задачами.

Услышав сигнал таймера, работник закусочной быстро заканчивает текущую задачу и вынимает заказанное вами блюдо из духовки. Затем вы получаете сообщение о том, что обед готов, — услышите объявление с номером своего заказа. Если одновременно поступили несколько заказов, требующих значительного времени, работник закусочной будет выполнять их один за другим, переходя к следующему заказу после очередного сигнала таймера.

Процессы Node выполняются по такой же схеме, что и обработка заказов в закусочной: заказы попадают к работникам закусочной (в поток) в той

последовательности, в которой поступают. Однако некоторые операции, такие как ввод/вывод, подобны заказам, для выполнения которых нужно больше времени (что-то печется в духовке или жарится на гриле). При этом работник закуской не должен останавливать остальные операции и ждать, пока блюдо испечется или поджарится. Таймеры гриля и духовки подобны сообщениям, которые появляются в цикле событий Node, активируя завершающее действие, в соответствии с выполняемой операцией.

Теперь мы получили некую работающую смесь из синхронных и асинхронных процессов. Но что происходит с таймером?

Функции `setTimeout()` и `setInterval()` срабатывают после заданной задержки. В момент срабатывания в цикл событий помещается сообщение, которое будет обработано в порядке очереди. Поэтому, если цикл событий основательно заполнен, то между срабатыванием таймера и вызовом соответствующей функции пройдет заметное время: *«Важно отметить, что функция обратного вызова, скорее всего, не будет вызвана через точный интервал в миллисекундах. Node.js не гарантирует ни точного времени срабатывания функции обратного вызова, ни последовательности выполнения операций. Функция обратного вызова сработает по истечении указанного времени настолько быстро, насколько возможно»* (документация по Node Timers).

По большей части задержка срабатывания незаметна для человека, однако она может вызывать недостаточно плавное воспроизведение анимации и другие странные эффекты.

В примере 17.4 я создал временную шкалу с прокруткой в формате SVG, на которой отображался процесс передачи данных клиенту через WebSockets. Для имитации передачи реальных данных использован таймер с трехсекундной задержкой, а данные представляют собой случайно сгенерированные числа. Поскольку таймер должен запускаться многократно, то в коде серверной части я задействовал `setInterval()`.

Пример 17.4. Временная шкала с прокруткой

```
const app = require('http');
const fs = require('fs');
const ws = require('nodejs-websocket');

let server;

// Статическая страница
const handler = (req, res) => {
  fs.readFile(`${__dirname}/drawline.html`, (err, data) => {
    if (err) {
      res.writeHead(500);
      return res.end('Error loading drawline.html');
    }
    res.writeHead(200);
    res.end(data);
  });
};
```

```

        return data;
    });
};

// Запускаем веб-сервер
// Обработчик будет перехватывать соединения с портом 8124
app.listen(8124);
app.createServer(handler);

// Таймер для данных
const startTimer = () => {
    setInterval(() => {
        const newval = Math.floor(Math.random() * 100) + 1;
        if (server.connections.length > 0) {
            console.log(`sending ${newval}`);
            const counter = { counter: newval };
            server.connections.forEach(conn => {
                conn.sendText(JSON.stringify(counter), () => {
                    console.log('conn sent');
                });
            });
        }
    }, 3000);
};

// Создаем на другом порте обработчик соединения с WebSocket
server = ws
    .createServer(conn => {
        console.log('connected');
        conn.on('close', () => {
            console.log('Connection closed');
        });
    })
    .listen(8001, () => {
        startTimer();
    });
};

```

Я вставил в код `console.log()`, чтобы можно было отследить, когда появляются события таймера, а когда — сообщения коммуникации. При вызове функции `setInterval()` таймер поступает в очередь. При обработке обратного вызова обмен данными WebSocket также вносится в очередь.

В данном примере использована функция `setInterval()` — один из трех таймеров Node. Формат у `setInterval()` такой же, как у одноименной функции, используемой в браузере. В нее передаются функция обратного вызова, интервал задержки (в миллисекундах) и возможные аргументы. По идее, таймер должен сработать через 3 с, но, как мы уже знаем, функция обратного вызова может начать действовать не сразу.

То же самое касается функций обратного вызова, передаваемых в функцию `sendText()` WebSocket. В их основе лежат сокеты Node Net (или TLS, если нужна повышенная безопасность). В документации к методу `socket.write()`,

используемому в `sendText()`, написано следующее: «Функция обратного вызова, передаваемая как необязательный параметр, будет выполнена после окончания записи данных — это может произойти не сразу» (документация Node).

Если настроить таймер на немедленное срабатывание, задав нулевую задержку, то увидим, что сообщения о передаче данных будут поступать вперемешку с сообщениями о коммуникациях (и так до тех пор, пока браузер клиента не зависнет, перегруженный операциями связи через сокет, так что вам больше не захочется использовать нулевую задержку в приложениях).

Однако время выполнения операций для всех клиентов будет одинаковым, поскольку обмен данными, осуществляемый с помощью функции обратного вызова из таймера, остается *синхронным*, так что данные для всех коммуникаций одинаковы — обрабатываются только обратные вызовы, которые кажутся неупорядоченными.

Ранее я уже писал об использовании `setInterval()` с нулевой задержкой. В действительности задержка не равна в точности нулю — в соответствии со спецификацией HTML5, которой следуют браузеры, Node фиксирует временной интервал на минимальном значении 4 мс. На первый взгляд может показаться, что это слишком мало, чтобы вызвать какие-либо проблемы, однако, когда речь заходит об анимации и критичных по времени процессах, временная задержка может повлиять на внешний вид приложения или ход выполнения функции.

Для того чтобы обойти эти ограничения, разработчики Node используют вместо `setInterval()` метод Node `process.nextTick()`. Функция обратного вызова, передаваемая в `process.nextTick()`, срабатывает на следующем проходе цикла событий, обычно до всех обратных вызовов, касающихся ввода/вывода (хотя здесь тоже есть свои ограничения, о которых я скоро расскажу). Никаких проволочек в 4 мс! Но что будет в случае ненормально большого количества рекурсивных вызовов `process.nextTick()`?

Возвращаясь к аналогии с закусочной: в напряженное обеденное время работники могут быть перегружены заказами. Они могут быть слишком заняты поступающими заказами и в результате реагировать на сигналы духовки и гриля с некоторой задержкой. При этом что-то может подгореть. Если вам случалось бывать в перегруженной закусочной, то вы могли заметить, что работник, стоящий за прилавком, сначала оценивает ситуацию на кухне и лишь потом принимает заказ. При этом он делает небольшую задержку или даже берет на себя часть кухонных обязанностей, так что посетителям приходится ждать своей очереди лишь немногим дольше, чем обычно.

То же самое происходит и в Node. Если позволить `process.nextTick()` выполняться по первому капризу, то операции ввода/вывода окажутся на голодном пайке. Поэтому в Node используется еще одна переменная, `process.maxTickDepth`, по умолчанию равная 1000. Это позволяет ограничить количество обратных вызовов `process.next()`, которые обрабатываются перед тем, как будет разрешено

выполнить обратный вызов для операций ввода/вывода. Это аналог работника закусочной, стоящего за прилавком.

В последних выпусках Node появилась функция `setImmediate()`. Она пытается решить все проблемы, связанные с операциями, выполняемыми по таймеру, и достичь наилучшего компромисса, который устроил бы все заинтересованные стороны. При активации функции `setImmediate()` ее обратный вызов добавляется в очередь после вызовов операций ввода/вывода, но перед обратными вызовами `setTimeout()` и `setInterval()`. Нам не приходится платить налог в 4 мс, как при вызове обычных таймеров, и у нас в коде нет вечной непогоды вроде `process.nextTick()`.

Снова — в последний раз — обратимся к аналогии с закусочной: `setImmediate()` играет роль посетителя, который, стоя в очереди, чтобы сделать заказ, и заметив, что работники закусочной не успевают реагировать на сигналы таймера духовки, вежливо заявляет, что подождет и сделает заказ позднее.



Однако вам наверняка не понравится использовать `setImmediate()` для временной шкалы с прокруткой, которая была рассмотрена в примере: вы и моргнуть не успеете, как браузер зависнет.

ГЛАВА 18

Модули Node

Один из самых приятных аспектов разработки приложений на Node.js — это встроенная модульность, которую обеспечивает среда Node. Модули Node легко загружаются и устанавливаются в любом количестве. Использовать их тоже легко: просто вставьте в код строку с функцией `require()` и вызовом модуля — и дело сделано.

Легкость внедрения модулей — одно из преимуществ *модульности* JavaScript. Она позволяет создавать внешние функции, которые зависят от других внешних функций, — данный принцип называется *слабым связыванием* (loose coupling). Это означает, что я могу использовать модуль `Foo`, не подключая модуль `Bar`, так как `Foo` сильно зависит от `Bar`.

Модульность JavaScript — это одновременно дисциплина и соглашение. Дисциплина состоит в необходимости выполнять некоторые обязательные критерии, чтобы внешний код стал частью модульной системы. Соглашение заключается между вами и остальными разработчиками на JavaScript: производя (или потребляя) внешний функционал в модульной системе, мы следуем определенным договоренностям и у всех нас есть определенные ожидания от модульной системы.



Одна из главных зависимостей, с которой связаны практически все аспекты управления приложением и библиотеками, — это необходимость применения системы управления версиями Git, а также GitHub — мегапопулярного хранилища для Git. Работа с Git и использование Git для GitHub выходят за рамки этой книги. Для ознакомления с Git я рекомендую книгу Ричарда Силвермана (Richard Silverman) *Git Pocket Guide* (издательство O'Reilly); о GitHub можно узнать из документации этого сервиса (<https://github.com>).

18.1. Поиск нужного модуля Node через npm

Задача

Мы пишем приложение Node и хотим использовать уже существующие модули. Но как их найти?

Решение

В рецепте 1.7 показано, как устанавливать пакеты с помощью `npm` — популярного менеджера пакетов Node (это тот самый «клей», на котором держится вся вселенная Node). Но вы еще не умеете *искать* нужные пакеты в обширном реестре `npm`.

В большинстве случаев мы находим модули по рекомендациям друзей и коллег, но иногда бывает нужно что-то новенькое. Новые модули можно искать непосредственно на веб-сайте `npm` (<https://www.npmjs.org>). Или делать это непосредственно из командной строки `npm`. Например, если мы хотим найти модули, которые что-то делают с документами в формате PDF, нужно ввести в командной строке следующий поисковый запрос:

```
$ npm search pdf
```

Обсуждение

Документации по использованию `npm` на веб-сайте `npm` предостаточно, есть там и интерфейс для поиска модулей. На странице каждого модуля указано, насколько он популярен, какие модули от него зависят, есть ли лицензия, имеется и другая полезная информация.

Модули можно искать и непосредственно через интерфейс `npm`. На это может уйти немало времени, а в итоге вы, скорее всего, получите огромное количество модулей, особенно по такому широкому критерию поиска, как работа с PDF.

Чтобы получить более точные результаты, можно указать в критерии поиска дополнительные термины:

```
$ npm search PDF generation
```

Список модулей, полученный в ответ на этот запрос, будет значительно короче, так как перечисленные модули будут относиться к созданию документов в формате PDF.

Когда вам наконец встретится модуль, который вас заинтересует, можно получить дополнительную информацию о нем с помощью следующей команды:

```
$ npm view electron
```

Полезную информацию можно извлечь из файла `package.json`, имеющегося в каждом модуле. В этом файле указано, от каких модулей зависит данный, кто его автор, когда он был создан. Я также советую посетить веб-страницу модуля на сайте `npm` и его репозиторий на сайте GitHub. Так вы поймете, активно ли поддерживается данный модуль, получите представление о том, насколько он популярен, какие есть нерешенные проблемы, а также заглянете в его код.

18.2. Преобразование библиотеки в модуль Node

Задача

Использовать в Node одну из разработанных вами библиотек.

Решение

Преобразовать библиотеку в модуль Node. В Node любой файл считается модулем. Предположим, что библиотека представляет собой файл, который находится в каталоге `/lib/hello.js` и содержит следующую функцию:

```
const hello = val => {  
  return console.log(`Hello ${val}`);  
};
```

Мы можем преобразовать этот код в модуль Node с помощью ключевого слова `exports`:

```
const hello = val => {  
  return console.log(`Hello ${val}`);  
};  
  
module.exports = hello;
```

Или можно экспортировать саму функцию:

```
module.exports = val => {  
  return console.log(`Hello ${val}`);  
};
```

После этого модуль можно использовать в приложении:

```
var hello = require('./lib/hello.js');  
  
// Выводим 'Hello world'  
hello('world');
```

Обсуждение

В основе стандартной модульной системы Node лежит CommonJS. В CommonJS используются три конструкции: ключевое слово `exports`, позволяющее определить, что именно экспортируется из библиотеки, функция `require()`, позволяющая подключить модуль к приложению, и `module`, который включает в себя информацию о модуле, а также может быть применен для того, чтобы экспортировать функцию напрямую.

Если библиотека возвращает объект, который содержит несколько функций и объектов данных, можно присвоить каждый из этих компонентов свойству `module.exports` с соответствующим именем или же вернуть следующий объект:

```
const greeting = {
  hello: val => {
    return console.log(`Hello ${val}`);
  },
  ciao: val => {
    return console.log(`Ciao ${val}`);
  }
};
```

```
module.exports = greeting;
```

или:

```
const hello = val => {
  return console.log(`Hello ${val}`);
};

const ciao = val => {
  return console.log(`Ciao ${val}`);
};

module.exports = { hello, ciao };
```

После этого можно обращаться к свойствам объекта напрямую:

```
const greeting = require('./lib/greeting.js')

// Выводим 'Hello world'
greeting.hello('world');
// Выводим 'Ciao mondo'
greeting.ciao('mondo');
```

Поскольку модуль не был установлен посредством `npm` — он просто находится в том же каталоге, что и приложение, то при обращении к модулю нужно указывать не только имя файла, но и путь.

Читайте также

В рецепте 18.3 показано, как убедиться, что код библиотеки будет работать в модульных средах `CommonJS` и `ECMAScript`.

В рецепте 18.4 показано, как создать отдельный модуль.

18.3. Перенос кода в модульную среду

Задача

Вы написали библиотеку и хотите поделиться ею с другими разработчиками. Но они применяют разные версии Node с модулями `CommonJS` и `ECMAScript`. Как гарантировать, что ваша библиотека будет работать во всех этих средах?

Решение

Использовать модули CommonJS, помещенные в модульную обертку ECMAScript.

Сначала пишем библиотеку в виде модуля CommonJS и сохраняем ее в файле с расширением `.cjs`:

```
const bbararray = {
  concatArray: (str, array) => {
    return array.map(element => {
      return `${str} ${element}`;
    });
  },
  splitArray: (str, array) => {
    return array.map(element => {
      return element.substring(str.length + 1);
    });
  }
};

module.exports = bbararray;
exports.concatArray = bbararray.concatArray;
exports.splitArray = bbararray.splitArray;
```

Затем создаем модульную обертку ECMAScript в файле с расширением `.mjs`:

```
import bbararray from './index.cjs';

export const { concatArray, splitArray } = bbararray;
export default bbararray;
```

Наконец, в файле `package.json` указываем данные для полей `type`, `main` и `exports`:

```
"type": "module",
"main": "./index.cjs",
"exports": {
  ".": "./index.cjs",
  "./module": "./wrapper.mjs"
},
```

Потребители модуля, использующие синтаксис CommonJS, смогут импортировать его с помощью функции `require`:

```
const bbararray = require('bbararray');

bbararray.concatArray('is', ['test', 'three']);
bbararray.splitArray('is', ['is test', 'is three']);
```

или так:

```
const { concatArray, splitArray } = require('bbararray');

concatArray('is', ['test', 'three']);
splitArray('is', ['is test', 'is three']);
```

А те, кто задействует модули ECMAScript, смогут указать версию модуля для нашей библиотеки посредством принятого в ES синтаксиса `import`:

```
import bbarray from 'bbarray/module';

bbarray.concatArray('is', ['test', 'three']);
bbarray.splitArray('is', ['is test', 'is three']);
```

или так:

```
import { concatArray, splitArray } from 'bbarray/module';

concatArray('is', ['test', 'three']);
splitArray('is', ['is test', 'is three']);
```



На момент написания книги, чтобы избежать принятого для модулей ECMAScript соглашения об именовании `/module`, можно было использовать флаг `--experimental-conditional-exports`. Но поскольку этот синтаксис сейчас является экспериментальным и, вероятно, вскоре изменится, мы пока не рекомендуем его применять. В последующих версиях Node он, скорее всего, станет стандартным. Подробнее об этом подходе можно прочитать в документации Node (<https://oreil.ly/Xzkid>).

Обсуждение

Модули CommonJS с самого начала были стандартом для Node. Такие инструменты, как Browserify, вывели синтаксис CommonJS за пределы экосистемы Node, что позволило разработчикам использовать модули в стиле Node и в браузерах. В стандарте ECMAScript 2015 (также известном как ES6) у JavaScript появился собственный синтаксис создания модулей: в Node 8.5.0 он применялся с флагом `--experimental-module`. Начиная с Node 13.2.0 в Node имеется встроенная поддержка модулей ECMAScript.

Обычно сначала пишут модуль, используя синтаксис CommonJS или ECMAScript, а затем с помощью компилятора передают для распространения либо отдельные точки входа в модуль, либо экспортированные пути. Однако здесь есть риск повторной загрузки модуля, если он был загружен непосредственно из приложения с применением одного синтаксиса, а затем тоже непосредственно или посредством зависимости, но с помощью другого синтаксиса.

В файле `package.json` есть три ключевых поля:

- **"type"**. Значение `module` говорит о том, что в данной библиотеке используется модульный синтаксис ECMAScript. Для библиотек, в которых задействуется только CommonJS, значение `type` равно `commonjs`.
- **"main"**. Определяет главную точку входа в приложение — в данном случае это ссылка на файл CommonJS.
- **"exports"**. Экспортируемые пути для наших модулей. В данном случае потребители пакета по умолчанию сразу получают модуль CommonJS, а те, кто

использует `package/module`, будут импортировать файл из модульной обертки ECMAScript.

```
"type": "module",
"main": "./index.cjs",
"exports": {
  ".": "./index.cjs",
  "./module": "./wrapper.mjs"
},
```

Если мы хотим избавиться от файловых расширений `.cjs` и `.mjs`, то можем сделать так:

```
"type": "module",
"main": "./index.js",
"exports": {
  ".": "./index.js",
  "./module": "./wrapper.js"
},
```

Читайте также

В рецепте 18.5 было показано, как с помощью сборщика Webpack обеспечить работу библиотеки в разных модульных средах в Node и браузере.

18.4. Создание устанавливаемого модуля Node

Задача

Вы создали модуль Node с нуля или преобразовали в модуль уже существующую библиотеку, и вам нужно, чтобы он работал и в Node, и в браузере. Теперь вы хотите сделать так, чтобы этот модуль устанавливался посредством npm.

Решение

Можно создать пакет из всего каталога, в котором находится и созданный вами модуль Node, и весь сопутствующий функционал, включая тесты модуля. Главным условием создания пакета и публикации модуля Node является создание файла `package.json` — в нем описываются модуль, все зависимости, структура каталога, что должно игнорироваться и т. д. Для того чтобы сгенерировать файл `package.json`, нужно выполнить команду `npm init` из корневого каталога проекта, а затем — все остальные действия в соответствии с подсказками системы.

Так выглядит сравнительно простой файл `package.json`:

```
{
  "name": "bbArray",
  "version": "0.1.0",
```



```

"description": "A description of what my module is about",
"main": "./lib/bbArray",
"author": {
  "name": "Shelley Powers"
},
"keywords": [
  "array",
  "utility"
],
"repository": {
  "type": "git",
  "url": "https://github.com/accountname/bbarray.git"
},
"engines" : {
  "node" : ">=0.10.0"
},
"bugs": {
  "url": "https://github.com/accountname/bbarray/issues"
},
"licenses": [
  {
    "type": "MIT",
    "url": "https://github.com/accountname/bbarray/raw/master/LICENSE"
  }
],
"dependencies": {
  "some-module": "~0.1.0"
},
"directories":{
  "doc":"./doc",
  "man":"./man",
  "lib":"./lib",
  "bin":"./bin"
},
"scripts": {
  "test": "nodeunit test/test-bbarray.js"
}
}

```

Когда файл `package.json` будет создан, упакуйте все каталоги с исходным кодом и файлом `package.json` в архив `gzip`. Затем установите пакет локально или передайте его в `npm`, чтобы сделать общедоступным.

Обсуждение

Файл `package.json` необходим для того, чтобы создать пакет из модуля Node и затем устанавливать этот пакет локально либо загружать через `npm`. В этом файле нужно указать как минимум имя и версию пакета. В примере также используются следующие поля:

- **description** — описание того, что собой представляет модуль и что он делает;

- `main` — начальный файл модуля;
- `author` — автор(-ы) модуля;
- `keywords` — список ключевых слов, по которым будет легче искать модуль;
- `repository` — место, где находится код, — обычно это GitHub;
- `engines` — версии Node, в которых гарантированно работает модуль;
- `bugs` — куда сообщать об ошибках;
- `licenses` — лицензия модуля;
- `dependencies` — список зависимостей, необходимых для работы модуля;
- `directories` — хеш, описывающий структуру каталогов модуля;
- `scripts` — хеш команд объекта, которые выполняются в течение жизненного цикла модуля.

Есть еще множество других параметров — все они описаны на веб-сайте npm (<https://oreil.ly/iXynV>). Существует специальная утилита, которая упрощает заполнение многих из этих полей. Для того чтобы ее выполнить, нужно ввести в командную строку следующую команду:

```
$ npm init
```

Утилита задаст вам ряд вопросов и сгенерирует базовый файл `package.json`.

Когда весь исходный код и файл `package.json` будут готовы, можно проверить, все ли работает. Для этого нужно перейти в корневой каталог модуля и выполнить следующую команду:

```
$ npm install . -g
```

Если не возникнет ошибок, то можно упаковать модуль в архив `gzip`. Теперь, если вы захотите опубликовать модуль, нужно сначала зарегистрироваться как пользователь в реестре npm:

```
$ npm add-user
```

Для того чтобы опубликовать модуль Node в реестре npm, нужно выполнить из корневого каталога модуля следующую команду, указав URL архива, имя файла с архивом или путь к нему:

```
$ npm publish ./
```

Если у модуля есть зависимости разработки — например, используется фреймворк тестирования, такой как Jest, то следующая короткая команда позволяет включить эти зависимости в файл `package.json`, чтобы применять их впоследствии.

Команда выполняется при установке зависимого модуля — из того же каталога, в котором находится `package.json`:

```
$ npm install jest --save-dev
```

При выполнении этой команды будет не только установлен фреймворк Jest (его мы обсудим в рецепте 18.6), но и внесены изменения в файл `package.json`, в который добавится следующий параметр:

```
"devDependencies": {  
  "jest": "^24.9.0"  
}
```

С помощью этой команды можно также добавить модуль в раздел `dependencies` файла `package.json`. Так, при выполнении команды

```
$ npm install express --save
```

в файл `package.json` вносится следующий параметр:

```
"dependencies": {  
  "express": "^3.4.11"  
}
```

Если модуль больше не нужен и не должен упоминаться в `package.json`, то, чтобы удалить его из раздела `devDependencies`, можно воспользоваться следующей командой:

```
$ npm remove jest
```

А эта команда удаляет модуль из раздела `dependencies`:

```
$ npm remove express
```

Если в разделе `dependencies` или `devDependencies` нет других модулей, то раздел не удаляется полностью, а остается в виде пустого значения:

```
"dependencies": {}
```



В npm есть солидное руководство для разработчиков по созданию и установке модулей Node (<https://oreil.ly/ifa4e>). Элементы, которые не должны войти в модуль, стоит указать в файле `.npmignore` или `.gitignore`. Также, хотя эта тема и выходит за рамки данной книги, вам стоит ближе познакомиться с возможностями Git и GitHub и использовать их в ваших приложениях и модулях.

Дополнительно: файл README и синтаксис Markdown

При упаковке модуля или библиотеки для многократного применения и загрузки кода в репозиторий, такой как GitHub, необходимо приложить к пакету

инструкции по установке этого модуля или библиотеки, а также базовую информацию об их использовании. Для этого вам понадобится файл README.

Вам, скорее всего, встречались файлы README.md в приложениях и модулях Node. Это текстовые файлы с несколько странной, непримечательной разметкой. Она не выглядит особенно полезной, пока вы не увидите этот файл на сайте, таком как GitHub, — именно из README извлекается представленная на странице пакета информация об установке и применении этого пакета. Разметка преобразуется в HTML, благодаря чему справочную информацию удобно читать.

Синтаксис разметки файлов README называется Markdown. На популярном веб-сайте Daring Fireball написано, что разметку Markdown удобно читать и писать, однако «удобство чтения превышает всего». В отличие от HTML, Markdown не мешает читать размеченный текст.



На сайте Daring Fireball также есть обзор основных элементов Markdown (<https://oreil.ly/qkKRT>). Однако при подготовке файлов для GitHub стоит также свериться с руководством GitHub's Flavored Markdown (<https://help.github.com/en/github/writing-on-github>).

Вот пример файла README.md:

```
# Название проекта
```

Краткое описание проекта: что он собой представляет и что делает. Если у проекта есть пользовательский интерфейс, приведите копию экрана.

Если есть более полная документация — дайте ссылку на нее.

```
## Функции
```

Опишите основные функции проекта (что он делает?) в виде маркированного списка:

- Функция #1
- Функция #2
- Функция #3

```
## Начало работы
```

Инструкции по установке, общее руководство по применению, примеры API, информация по сборке и развертыванию. Опишите все операции четко и последовательно, исходя из предположения, что пользователь ничего или почти ничего не знает о проекте.

```
### Установка и зависимости
```

Как привести проект в рабочее состояние? Какие зависимости есть у проекта? Постарайтесь описать все в виде простых и понятных шагов. Предоставьте внешние ссылки.

Использование

Приведите примеры использования проекта. Для больших проектов с обширной документацией приведите несколько примеров и дайте ссылку на полную документацию.

Сборка и развертывание

Если пользователь должен собирать или развертывать проект, предоставьте здесь соответствующие полезные рекомендации.

Получение помощи

Что делать и чего следует ожидать пользователю, если в проекте обнаружатся ошибки или пользователь попадет в тупиковую ситуацию? Опишите, на какую поддержку он может рассчитывать, дайте ссылку на систему отслеживания ошибок и на планы развития проекта, если они есть.

Куда пользователю следует обратиться, чтобы задать вопрос? (Stack Overflow, Gitter, IRC, список рассылки и т. п.)

При желании также можно предоставить адреса электронной почты основных участников проекта.

Как помочь проекту

Инструкции по настройке и развитию среды, стандарты по написанию кода, тестированию и отправке кода в репозиторий. Здесь будет уместной ссылка на отдельный файл CONTRIBUTING.md. Вот пример такого файла для проекта Hoodie: <https://github.com/hoodiehq/hoodie/blob/master/CONTRIBUTING.md>

Нормы поведения

Предоставьте ссылку на раздел "Нормы поведения" для вашего проекта. Советую использовать "Соглашение участника": <http://contributor-covenant.org/>

Лицензия

Укажите лицензию проекта. Если вам нужна помощь в выборе лицензии, воспользуйтесь этим руководством: <https://choosealicense.com>

В большинстве распространенных текстовых редакторов реализованы подсветка синтаксиса и предварительный просмотр документов с разметкой Markdown. Для всех платформ стационарных компьютеров существуют специализированные редакторы Markdown. Также есть утилиты командной строки для преобразования файлов README.md в удобочитаемый HTML, такие как Pandoc (<https://oreil.ly/Cc4GX>):

```
$ pandoc README.md -o readme.html
```

На рис. 18.1 показана HTML-страница, сгенерированная из README.md, — не особенно красивая, но вполне читаемая.

При размещении исходного кода на сайте, таком как GitHub, с помощью файла README.md будет создана титульная страница репозитория.

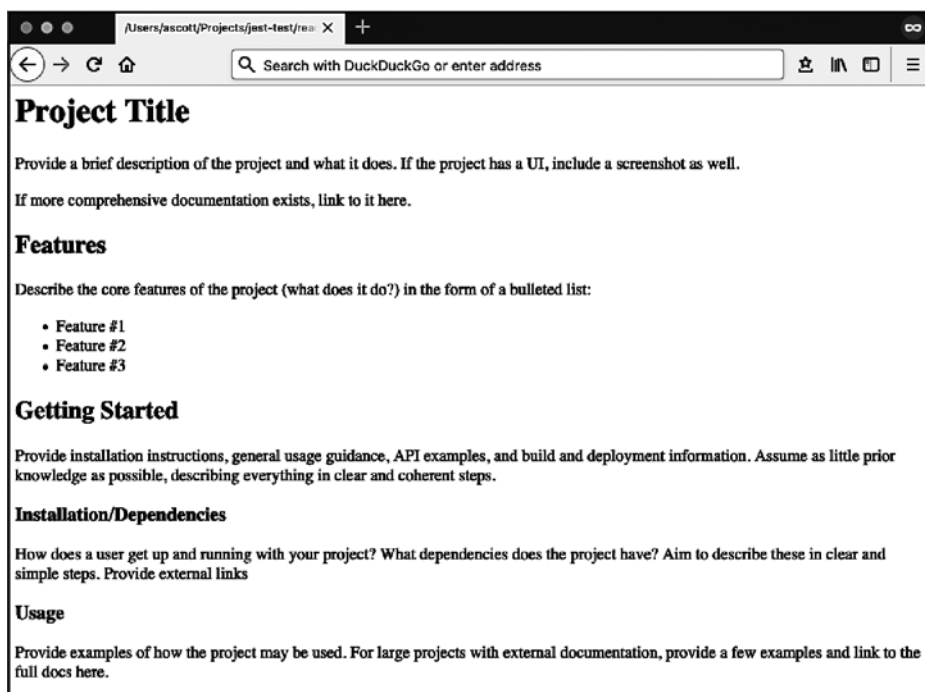


Рис. 18.1. HTML-страница, сгенерированная из файла README.md, с текстом и разметкой Markdown

18.5. Создание мультиплатформенных библиотек

Задача

Вы написали библиотеку, которая может быть полезна как в браузере, так и в Node.js, и хотите, чтобы она была доступна в обеих этих средах.

Решение

Использовать сборщик, например Webpack, и собрать библиотеку таким образом, чтобы она работала в качестве модулей ES2015, CommonJS и AMD и загружалась в браузере в теге `script`.

Для этого пропишите в файле Webpack `webpack.config.js` поля `library` и `libraryTarget`. Они сигнализируют о том, что данный модуль должен собираться как библиотека и предназначен для работы в разных средах:

```
const path = require('path');

module.exports = {
  entry: './src/index.js',
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'my-library.js',
    library: 'myLibrary',
    libraryTarget: 'umd',
    globalObject: 'this'
  },
};
```

В поле `library` указывается имя библиотеки, которая будет использоваться в модульных средах ECMAScript, CommonJS и AMD. В поле `libraryTarget` можно указать, как именно будет представлен модуль. По умолчанию значение этого поля равно `var`, то есть модуль будет представлен как переменная. Если поле равно `umd`, то будет применено определение универсального модуля JavaScript — JavaScript Universal Module Definition (UMD) (<https://oreil.ly/VSpd0>), что позволит задействовать библиотеку в виде модулей разных типов. Для того чтобы сборка UMD была доступна и в браузере, и в среде Node.js, нужно присвоить свойству `output.globalObject` значение `this`.



Подробнее о сборке кода с помощью Webpack читайте в главе 17.

Обсуждение

В следующем примере я создал простую математическую библиотеку. Пока что она состоит из единственной функции `squareIt`, которая принимает в виде параметра число и возвращает значение, равное этому числу, умноженному само на себя. Функция находится в файле `src/index.js`:

```
export function squareIt(num) {
  return num * num;
};
```

В файле `package.json` указаны зависимости разработки: Webpack и интерфейс командной строки (command-line interface, CLI) для Webpack. Там также дана ссылка на собранную версию библиотеки `main`, которую Webpack разместит в каталоге `dist`. Я также добавил скрипт сборки с очень подходящим названием `build` — он будет запускать сборщик Webpack. Это позволит мне генерировать сборки, просто вводя команду `npm run build` (или `yarn run build`, если использовать Yarn):

```
{
  "name": "my-library",
  "version": "1.0.0",
  "description": "An example library bundled by Webpack",
  "main": "dist/my-library.js",
  "scripts": {
    "build": "webpack"
  },
  "keywords": ["example"],
  "author": "Adam Scott <adam@jseverywhere.io>",
  "license": "MIT",
  "devDependencies": {
    "webpack": "4.44.1",
    "webpack-cli": "3.3.12"
  }
}
```

Наконец, в соответствии с указаниями рецепта в моем проекте есть файл `webpack.config.js`:

```
const path = require('path');

module.exports = {
  entry: './src/index.js',
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'my-library.js',
    library: 'myLibrary',
    libraryTarget: 'umd',
    globalObject: 'this'
  },
};
```

При такой конфигурации команда `npm run build` упакует библиотеку и разместит ее в подкаталоге `dist` проекта. Именно этот упакованный файл будут использовать потребители библиотеки.



Для того чтобы перед публикацией в npm протестировать пакет локально, выполните из корневого каталога проекта команду `npm link`. Затем создайте отдельный проект, в котором можно использовать этот модуль, и выполните для проекта команду `npm link <имя библиотеки>`. В результате будет создана символическая ссылка на пакет — такая, как будто он был установлен через интернет.

Публикация библиотеки

Когда библиотека будет готова, вы, скорее всего, захотите опубликовать ее в npm для дальнейшего распространения. Убедитесь, что в проекте действует система управления версиями Git и проект размещен в открытом удаленном репозитории, таком как GitHub или GitLab. Затем выполните из корневого каталога проекта следующие команды:


```
$ git init
$ git remote add origin git://git-remote-url
$ npm publish
```

После размещения в репозитории Git и в реестре npm библиотека становится доступной для потребителей — для этого им нужно выполнить команду `npm install`, загрузить или клонировать репозиторий Git либо поставить на веб-странице явную ссылку на библиотеку в формате <https://unpkg.com/<имя-библиотеки>>. Библиотека будет доступна в различных библиотечных форматах JavaScript, например в виде модуля ES 2015:

```
import * as myLibrary from 'my-library';

myLibrary.squareIt(4);
```

в виде модуля CommonJS:

```
const myLibrary = require('my-library');

myLibrary.squareIt(4);
```

в виде модуля AMD:

```
require(['myLibrary'], function (myLibrary) {
  myLibrary.squareIt(4);
});
```

А так можно подключить библиотеку в теге `script` на веб-странице:

```
<!doctype html>
<html>
  <script src="https://unpkg.com/my-library"></script>
  <script>
    myLibrary.squareIt(4);
  </script>
</html>
```

Подключение библиотечных зависимостей

Обычно библиотека содержит зависимости. В предложенной конфигурации все они будут упакованы в одну сборку вместе с библиотекой. Для того чтобы сократить размер сборки и избавить потребителей библиотеки от необходимости устанавливать несколько экземпляров одной и той же зависимости, лучше представить зависимости как равноправные, то есть такие, относительно которых пользователь решает сам: установить или дать ссылку. Для этого нужно прописать свойство `externals` со значением `webpack.config.js`. В следующем примере `moment` является равноправной зависимостью:

```
const path = require('path');

module.exports = {
  entry: './src/index.js',
  output: {
```

```
    path: path.resolve(__dirname, 'dist'),
    filename: 'my-library.js',
    library: 'myLibrary',
    libraryTarget: 'umd',
    globalObject: 'this'
  },
  externals: {
    moment: {
      commonjs: 'moment',
      commonjs2: 'moment',
      amd: 'moment',
      root: 'moment',
    }
  }
};
```

В этой конфигурации библиотека будет считать `moment` глобальной переменной.

18.6. Тестирование модулей

Задача

Убедиться, что модуль функционирует корректно и может использоваться потребителями.

Решение

Сделать модульное тестирование частью процесса разработки.

Рассмотрим следующий модуль с именем `bbararray`, который находится в файле `index.js`:

```
const util = require('util');

const bbararray = {
  concatArray: (str, array) => {
    if (!util.isArray(array) || array.length === 0) {
      return -1;
    }

    if (typeof str !== 'string') {
      return -1;
    }

    return array.map(element => {
      return `${str} ${element}`;
    });
  },
  splitArray: (str, array) => {
    if (!util.isArray(array) || array.length === 0) {
      return -1;
    }
  }
};
```

```

    }

    if (typeof str !== 'string') {
      return -1;
    }

    return array.map(element => {
      return element.substring(str.length + 1);
    });
  }
};

```

```
module.exports = bbararray;
```

При использовании фреймворка Jest (<https://jestjs.io>) для тестирования кода JavaScript следующий модульный тест (размещен в файле `index.js` в подкаталоге `test` проекта) должен привести к успешному прохождению шести тестов:

```

const bbararray = require('../index.js');

describe('concatArray()', () => {
  test('should return -1 when not using array', () => {
    expect(bbararray.concatArray(9, 'str')).toBe(-1);
  });

  test('should return -1 when not using string', () => {
    expect(bbararray.concatArray(9, ['test', 'two'])).toBe(-1);
  });

  test('should return an array with proper args', () => {
    expect(bbararray.concatArray('is', ['test', 'three'])).toStrictEqual([
      'is test',
      'is three'
    ]);
  });
});

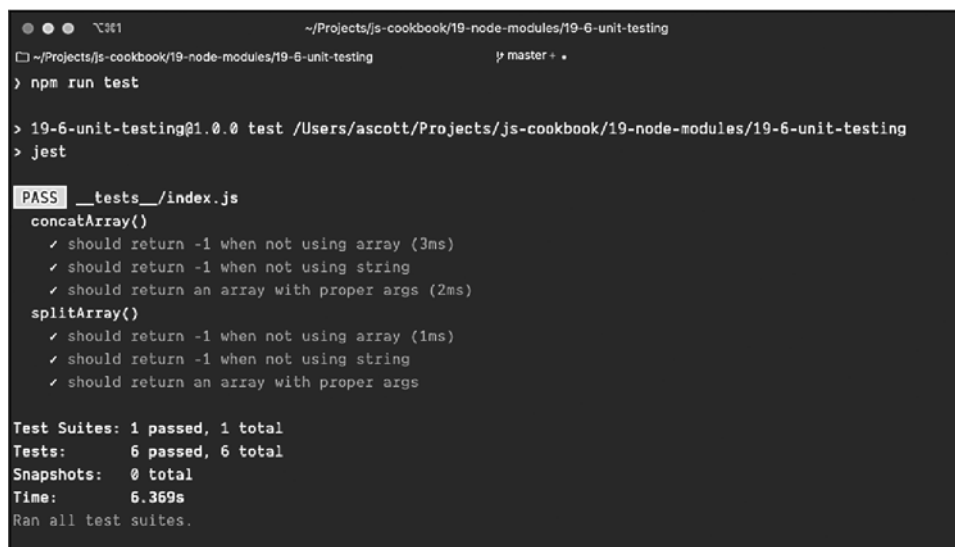
describe('splitArray()', () => {
  test('should return -1 when not using array', () => {
    expect(bbararray.splitArray(9, 'str')).toBe(-1);
  });

  test('should return -1 when not using string', () => {
    expect(bbararray.splitArray(9, ['test', 'two'])).toBe(-1);
  });

  test('should return an array with proper args', () => {
    expect(bbararray.splitArray('is', ['is test',
      'is three'])).toStrictEqual([
      'test',
      'three'
    ]);
  });
});

```

Результат тестирования, выполненного при помощи команды `npm test`, показан на рис. 18.2.



```
~/Projects/js-cookbook/19-node-modules/19-6-unit-testing
> npm run test

> 19-6-unit-testing@1.0.0 test /Users/ascott/Projects/js-cookbook/19-node-modules/19-6-unit-testing
> jest

PASS __tests__/index.js
  concatArray()
    ✓ should return -1 when not using array (3ms)
    ✓ should return -1 when not using string
    ✓ should return an array with proper args (2ms)
  splitArray()
    ✓ should return -1 when not using array (1ms)
    ✓ should return -1 when not using string
    ✓ should return an array with proper args

Test Suites: 1 passed, 1 total
Tests:      6 passed, 6 total
Snapshots:  0 total
Time:       6.369s
Ran all test suites.
```

Рис. 18.2. Выполнение модульных тестов с помощью фреймворка Jest

Обсуждение

Модульные тесты — это способ, используемый разработчиками, для того чтобы убедиться, что код соответствует спецификациям. Модульное тестирование подразумевает проверку функционального поведения и того, что происходит, если передать некорректные аргументы или не передать их вообще. Тестирование называется модульным, так как применяется не ко всему приложению, а к отдельным участкам кода — например, к отдельным модулям в приложениях Node. Модульные тесты — это составная часть *комплексного тестирования*, при котором все части программного обеспечения собираются воедино. После этого выполняется *тестирование на приемлемость для пользователя*: действительно ли приложение соответствует пользовательским ожиданиям (и не слишком сильно раздражает — хотя бы по большей части).

Модульное тестирование — один из этапов разработки, который поначалу может показаться проблемным, но впоследствии становится второй натурой. Хорошая идея — разрабатывать тесты параллельно с написанием кода. Многие разработчики даже практикуют *разработку через тестирование*, когда сначала пишутся модульные тесты, а уж потом — собственно код.

В предложенном решении мы задействовали Jest — многофункциональный фреймворк для тестирования. Наш модуль простой, поэтому мы не применяли некоторые из сложных механизмов тестирования, реализованных в Jest. Но это

хороший пример строительных блоков, используемых при написании модульных тестов.

Для того чтобы установить Jest, нужно ввести следующую команду:

```
$ npm install jest --save-dev
```

Я использовал флаг `--save-dev`, так как обычно прописываю Jest в разделе зависимостей разработки модуля. Кроме того, я добавил в файл `package.json` следующий раздел:

```
"scripts": {  
  "test": "jest"  
},
```

Скрипт тестирования находится в файле `index.js`, в подкаталоге проекта `tests`. Jest автоматически будет находить файлы в каталоге `tests`, а также файлы, соответствующие шаблону именования *имя-файла.test.js*. Для выполнения тестов используется следующая команда:

```
$ npm test
```

Для тестирования возвращаемых значений в модульных тестах Jest выполняется проверка на *соответствие ожиданиям* (<https://oreil.ly/E7RnY>).

Управление экосистемой Node

Экосистема Node становится все более обширной и разветвленной — от скриптов, работающих на ноутбуке, до управления данными на удаленных серверах. Разнообразие основных функций Node в сочетании с тысячами модулей, созданных пользователями, образует богатую среду, в которой можно выполнить практически любую задачу программирования. Однако это разнообразие способно усложнить выбор параметров для обычных задач. В этой главе представлены типичные проблемы, с которыми могут столкнуться разработчики Node.

19.1. Использование переменных среды

Задача

В приложение Node нужно передавать разные значения переменных в зависимости от среды, такие как имя локального компьютера и состояние «в разработке».

Решение

Использовать переменные среды — присвоить им значения и считывать их в разных средах. В центральном модуле Node `process` есть свойство `env`, благодаря которому приложение получает доступ ко всем переменным среды. В следующем примере я читаю значение переменной среды с именем `NODE_ENV`:

```
process.env.NODE_ENV
```

Для того чтобы присвоить значение переменной среды, нужно вначале определить это значение, запустив приложение с помощью команды `node`. В следующей команде переменной `NODE_ENV` присваивается значение `development`, после чего запускается скрипт `index.js`:

```
$ NODE_ENV=development node index.js
```

Работая над проектами с несколькими переменными среды, обычно предпочитают хранить эти переменные локально, в файле `.env`. Для этого в Node используется пакет `dotenv`, который устанавливается с помощью `npm`:

```
$ npm install dotenv --save
```

Теперь нужно подключить модуль в коде приложения с помощью `require` и инициализировать конфигурацию модуля:

```
require('dotenv').config();
```

После того как модуль будет подключен, переменные среды можно будет читать из файла `.env`, вместо того чтобы передавать их в командной строке. В файле `.env` можно указать значения для нескольких переменных среды:

```
PORT=8080
DB_URI=mongodb://mongodb0.example.com:27017
KEY=12345
```

Обсуждение

Объект `process` не обязательно импортировать в качестве модуля с помощью оператора `require`, так как он доступен глобально во всех программах Node. В объекте `process` предоставляется информация о текущем процессе Node, включая информацию о среде, в которой он выполняется.

При чтении переменных среды обычно имеет смысл использовать оператор `||`, чтобы определить значение по умолчанию на тот случай, если среда не предоставляет запрашиваемое значение. В следующем примере переменной `port` присваивается значение переменной среды `PORT`, а если такой переменной не существует, то значение `8080`:

```
const port = process.env.PORT || 8080;
```

Пакет `dotenv` — это модуль `npm`, который позволяет загружать переменные среды из файла `.env`. Для этого достаточно установить пакет, подключить его с помощью оператора `require` и активировать конфигурацию:

```
require('dotenv').config();
```

После того как модуль установлен и сконфигурирован, он будет автоматически читать значения из файла `.env`, расположенного в корневом каталоге проекта. Можно также выбрать такую конфигурацию пакета, чтобы читать данные из файла, расположенного в другом месте:

```
require('dotenv').config({ path: '/alternate/file/path/.env' })
```

Если в проекте Node используются модули ECMAScript, то нужно вначале импортировать пакет в виде модуля, а затем в отдельной строке активировать его конфигурацию:

```
import dotenv from 'dotenv'
dotenv.config()
```

В среде эксплуатации переменные среды чаще всего устанавливаются на хосте. В этом случае считывать значения из файла `.env` не имеет смысла — в среде эксплуатации модуль `dotenv` обычно не используют:

```
if (process.env.NODE_ENV !== 'production') {
  require('dotenv').config();
}
```



Никогда не помещайте файл `.env` в репозиторий — обязательно внесите его в список ресурсов, игнорируемых системой управления версиями. В файлах `.env` часто хранится секретная информация о среде, такая как пароли или ключи, — эти данные не должны распространяться. Вместо этого рекомендуется сохранять в репозитории файлы типа `.env.example` — с пустыми значениями или заглушками.

19.2. Что делать с адом обратных вызовов

Задача

Сделать что-то с асинхронными операциями, такими как чтение файла и запись прочитанных данных в новый файл. В Node есть функционал, использующий функции обратного вызова, однако если задействовать эти функции асинхронно, в итоге получим код с большим количеством вложенных вызовов, легко распознаваемых по длинной лесенке отступов. Эти вложенные вызовы затрудняют чтение кода и обслуживание приложения.

Решение

Начиная с версии 8.0 в Node можно использовать синтаксис `async/await` и утилиту `promisify`:

```
const fs = require('fs');
const { promisify } = require('util');

const readFile = promisify(fs.readFile);
const appendFile = promisify(fs.appendFile);

const readAppend = async (originalFile, secondaryFile) => {
  const fileData = await readFile(originalFile);
  await appendFile(secondaryFile, fileData);
  console.log(
    `The data from ${originalFile} was appended to ${secondaryFile}!`
  );
};

readAppend('./files/main.txt', './files/secondary.txt');
```


Встроенная в Node утилита `promisify` необычайно полезна: благодаря ей любая функция, соответствующая общепринятому стилю «ошибка, значения, обратный вызов», может возвращать промис. В Node 10+ благодаря API `fs.promises` операции с файловой системой можно сразу использовать как промисы:

```
const fsp = require('fs').promises;

const readAppend = async (originalFile, secondaryFile) => {
  const fileData = await fsp.readFile(originalFile);
  await fsp.appendFile(secondaryFile, fileData);
  console.log(
    `The data from ${originalFile} was appended to ${secondaryFile}!`
  );
};

readAppend('./files/main.txt', './files/tertiary.txt');
```

Обсуждение

Код Node является асинхронным, или неблокирующим, по своей природе. Это значит, что пока код ожидает завершения операции, он может выполнять что-то еще. Однако зачастую нам нужно, чтобы операции выполнялись в определенной последовательности. В Node для этого традиционно используются функции обратного вызова. Это такие функции, которые вызываются после реализации некоторой задачи. В следующем примере выполняется чтение файла, а затем операция, запрограммированная как функция обратного вызова:

```
fs.readFile(file, (error, data) => {
  if (error) {
    // обрабатываем ошибку
  } else {
    // выполняем операцию после чтения файла
  }
});
```

Синтаксис `async/await` позволяет писать асинхронный код так же, как и синхронный. Синтаксис `async/await` подробно описан в главе 10:

```
const waitOne = () => {
  return new Promise(resolve => {
    setTimeout(() => {
      console.log('It has been one second');
      resolve();
    }, 1000);
  });
};

const callWait = async () => {
  await waitOne();
};

callWait();
```

Работая с функцией, которая соответствует общепринятому стилю «ошибка, значения, обратный вызов», можно вернуть промис с помощью встроенной в Node утилиты `promisify`:

```
const fs = require('fs');
const { promisify } = require('util');
const writeFile = promisify(fs.writeFile);
```

При использовании `async/await` ошибки обрабатываются внутри блоков `try/catch`:

```
try {
  await writeFile(file, buf);
} catch (error) {
  console.log(error);
  throw error;
}
```

В следующем примере показано, как можно улучшить существующий код. В этом коде мы с помощью функций обратного вызова записываем в файл две строки, читаем их и выводим содержимое в консоль:

```
const fs = require('fs');

const callbackHell = file => {
  const buf = Buffer.from('Callback hell first string\n');
  const buf2 = Buffer.from('Callback hell second string\n');

  // Записываем или дописываем в файл содержимое первого буфера
  fs.writeFile(file, buf, err => {
    if (err) {
      console.log(err);
      throw err;
    }
    // Дописываем в файл содержимое второго буфера
    fs.appendFile(file, buf2, err2 => {
      if (err2) {
        console.log(err2);
        throw err2;
      }
      // Выводим содержимое файла в консоль
      fs.readFile(file, 'utf-8', (err3, data) => {
        if (err3) {
          console.log(err3);
          throw err3;
        }
        console.log(data);
      });
    });
  });
};

callbackHell('./files/callback.txt');
```

И это еще сравнительно простая операция. Обратите внимание на то, как быстро увеличиваются отступы для вложенных обратных вызовов. При использовании `async/await` код становится значительно понятнее:

```
const fs = require('fs');
const { promisify } = require('util');

const writeFile = promisify(fs.writeFile);
const appendFile = promisify(fs.appendFile);
const readFile = promisify(fs.readFile);

const fileWriteRead2 = async file => {
  const buf = Buffer.from('The first string\n');
  const buf2 = Buffer.from('The second string\n');

  // Записываем или дописываем в файл содержимое первого буфера
  try {
    await writeFile(file, buf);
  } catch (error) {
    console.log(error);
    throw error;
  }

  // Дописываем в файл содержимое второго буфера
  try {
    await appendFile(file, buf2);
  } catch (error) {
    console.log(error);
    throw error;
  }

  // Выводим содержимое файла в консоль
  console.log(await readFile(file, 'utf8'));
};

fileWriteRead2('./files/async.txt');
```

Мы сделали код гораздо понятнее, при этом не отказавшись от асинхронного выполнения.

Во всех примерах я использую операции с файловой системой, однако синтаксис `async/await` необычайно полезен в самых разных сценариях применения Node, включая взаимодействие с базой данных, чтение данных с удаленных ресурсов, хеширование строк и многое другое.

19.3. Доступ к функциям командной строки из приложений Node

Задача

Запустить из приложения Node утилиту командной строки, такую как `ImageMagick`.

Решение

Использовать модуль `Node child_process`. Например, если мы хотим выполнить функцию `identify` из `ImageMagick`, а затем вывести данные в консоль, то нужно написать следующий код:

```
const { spawn } = require('child_process');
const identify = spawn('identify', ['-verbose', 'osprey.jpg']);
identify.stdout.on('data', data => {
  console.log(`stdout: ${data}`);
});
identify.stderr.on('data', data => {
  console.log(`stderr: ${data}`);
});
identify.on('exit', code => {
  console.log(`child process exited with code ${code}`);
});
```

Обсуждение

В модуле `child_process` предусмотрены следующие четыре метода для выполнения операций из командной строки и обработки возвращаемых данных.

- `spawn(command, [args], [options])`. Загружает заданный процесс. В него можно передавать дополнительные аргументы командной строки, а также объект `options` с дополнительными параметрами, такими как `cwd`, который позволяет изменить каталог, и `uid`, в котором хранится ID пользователя для данного процесса.
- `exec(command, [options], callback)`. Выполняет команду из оболочки и записывает результат в буфер.
- `execFile(file, [args],[options],[callback])`. Делает то же самое, что и `exec()`, но запускает файл напрямую.
- `fork(modulePath, [args],[options])`. Особый случай `spawn()` — порождает процессы Node и возвращает объект со встроенным каналом коммуникации. Каждый раз требует отдельного экземпляра V8, так что используйте этот метод экономно.

С методами `child_process` связаны три потока: `stdin`, `stdout` и `stderr`. Из всех методов `child_process` чаще всего используют `spawn()`, и в нашем примере тоже задействован именно он. Вначале из командной строки выполняется команда `ImageMagick identify`. Она возвращает разнообразную информацию об изображении. В массиве аргументов передаются флаг `--verbose` и имя файла с изображением. Когда в потоке `child_process.stdout` возникает событие `data`, приложение выводит данные в консоль. Они представляют собой буфер, в котором

для слияния с другой строкой неявно используется функция `toString()`. Ошибки в случае их возникновения тоже выводятся в консоль. Третий обработчик события просто сообщает о завершении дочернего процесса.

Если мы хотим представить результат в виде массива, то нужно изменить обработчик события для ввода данных:

```
identify.stdout.on('data', (data) => {  
  console.log(data.toString().split("\n"));  
});
```

Теперь обрабатываемые данные будут представляться в виде массива строк, разделенных символами новой строки, по которым определяется начало ввода данных.



Если у вас установлена утилита `GraphicsMagick` или `ImageMagick`, то вместо дочернего процесса для обработки изображений можно использовать модуль `Node gm` (<http://aheckmann.github.io/gm>).

Дополнительно: использование дочерних процессов в Windows

В предложенном решении показано, как задействовать дочерние процессы в macOS и Linux. В Windows это происходит примерно так же, как в Linux/Unix, но с некоторыми отличиями.

В Windows нельзя явно передать команду дочернему процессу — нужно вызвать программу `Windows cmd.exe` и выполнить процесс из нее. Кроме того, первым флагом команды является `/c` — он сообщает `cmd.exe`, что нужно выполнить команду и завершить работу.

Следующий пример позаимствован из книги Шелли Пауэрс (Shelley Powers) *Learning Node*¹ (издательство O'Reilly). В этом коде с помощью команды `cmd.exe` выводится список содержимого каталога, который возвращает команда `Windows dir`:

```
const { spawn } = require('child_process');  
  
const cmd = spawn('cmd', ['/c', 'dir\n']);  
  
cmd.stdout.on('data', data => {  
  console.log(`stdout: ${data}`);  
});  
  
cmd.stderr.on('data', data => {  
  console.log(`stderr: ${data}`);  
});  
  
cmd.on('exit', code => {  
  console.log(`child process exited with code ${code}`);  
});
```

¹ Пауэрс Ш. Изучаем Node.js. — СПб.: Питер, 2014.

19.4. Передача аргументов в командную строку

Задача

Нам бы хотелось иметь возможность передавать в командную строку аргументы из приложения Node и считывать их значения в приложении.

Решение

В простых случаях можно использовать свойство `process.argv` — оно возвращает массив, в котором содержатся все аргументы командной строки, переданные в программу при ее выполнении. Поскольку эти значения объединены в массив, мы можем перебрать их в цикле и прочитать значения (или, как в данном примере, вывести в консоль):

```
process.argv.forEach((value, index) => {  
    console.log(`${index}: ${value}`);  
});
```

Теперь при запуске нашего скрипта можно передать аргументы командной строки, и они будут выводиться в консоль:

```
$ node index.js --name=Adam --food=pizza
```

В консоли получим следующее:

```
0: /usr/local/bin/node  
1: /Users/ascott/Projects/command-line-args/index.js  
2: --name=Adam  
3: --food=pizza
```

Объект `process` в Node — это глобальный объект, благодаря которому сценарий получает доступ к информации о текущем процессе Node.js. В свойстве `argv` объекта `process` содержатся значения аргументов. Первый индекс всегда указывает на путь к выполняемому файлу в среде Node, второе значение массива — это всегда путь к самому скрипту. Остальные элементы массива представляют собой аргументы, стоящие в той последовательности, в которой они были переданы в скрипт.

Обсуждение

Благодаря доступности аргументов непосредственно из объекта Node `process` открывается возможность легко получать параметры командной строки. Однако при синтаксическом анализе и использовании этих значений могут возникнуть сложности. К счастью, популярный модуль Yargs (<https://oreil.ly/Ue9LF>) несколько упрощает работу с аргументами командной строки:

```
const yargs = require('yargs/yargs');  
const { hideBin } = require('yargs/helpers');
```

```
const {argv} = yargs(hideBin(process.argv));  
  
console.log(argv);
```

Вернемся к нашему скрипту. Сейчас при передаче в него аргументов командной строки выводятся следующие значения аргументов:

```
$ node index.js --name=Adam --food=pizza  
# будет выведено следующее:  
{ _: [], name: 'Adam', food: 'pizza', '$0': 'yargs/index.js' }
```

При использовании модуля Yargs можно легко прочитать отдельные значения и оперировать ими в скрипте:

```
const yargs = require('yargs/yargs');  
const { hideBin } = require('yargs/helpers');  
  
const {argv} = yargs(hideBin(process.argv));  
  
if (argv.food === 'pizza') {  
  console.log('mmm');  
}
```

Благодаря доступности аргументов командной строки можно использовать информацию, поступающую при выполнении приложения, и реагировать соответственно. Yargs способен на гораздо большее, нежели просто читать входные значения, — этот модуль позволяет настраивать параметры команд подсказки, применять входные значения типа Boolean, ограничивать значения множеством predefined вариантов и многое другое. Для получения дополнительной информации и ссылок на другие ресурсы советую ознакомиться с документацией Yargs (<https://github.com/yargs/yargs#documentation>).

19.5. Создание утилиты командной строки с подсказкой с помощью Commander

Задача

Превратить модуль Node в утилиту командной строки Linux, включая поддержку аргументов и дополнительных параметров командной строки.

Решение

Преобразовать модуль Node в утилиту командной строки Linux, добавив в начало модуля следующую строку:

```
#!/usr/bin/env node
```

Для того чтобы в командной строке можно было передавать в приложение аргументы и дополнительные параметры, в том числе самый полезный из них, `--help`, нужно задействовать модуль `Commander`:

```
#!/usr/bin/env node
const program = require('commander');

program
  .version('0.0.1')
  .option('-n, --number <value>', 'A number to square')
  .parse(process.argv);

const square = Math.pow(program.number, 2);

console.log(`The square of ${program.number} is ${square}`);
```



В рецепте 19.4 мы обсуждаем модуль `Yargs`, благодаря которому упрощается использование аргументов командной строки. `Yargs` — отличное средство для работы с аргументами командной строки, а `Commander` — полнофункциональный модуль, позволяющий строить приложения, управляемые из командной строки. Советую рассмотреть оба варианта, прежде чем решить, какой из них лучше подходит в вашем случае.

Обсуждение

Для того чтобы преобразовать модуль `Node` в утилиту командной строки, нужно вставить в начало модуля следующее:

```
#!/usr/bin/env node
```

Затем с помощью `CHMOD` преобразовать файл в исполняемый:

```
$ chmod a+x square.js
```

Для выполнения созданного ранее примера я перешел в папку проекта и ввел в командной строке:

```
$ ./square.js -n 4
```

Созданная утилита командной строки просто выводит в консоль квадрат заданного числа. Рассмотрим более сложный пример, в котором с помощью библиотеки `Puppeteer` (<https://github.com/puppeteer/puppeteer>) создается копия экрана с открытой веб-страницей. Для этого напишем следующий код и сохраним его в файле `snapshot.js`:

```
#!/usr/bin/env node
const program = require('commander');
const puppeteer = require('puppeteer');

program
  .version('0.0.1')
```



```

.option('-s, --source [website]', 'Source website')
.option('-f, --file [filename]', 'Filename')
.parse(process.argv);

(async () => {
  console.log('capturing screenshot...');
  const browser = await puppeteer.launch();
  const page = await browser.newPage();
  await page.goto(program.source);
  await page.screenshot({ path: program.file });
  await browser.close();
  console.log(`captured screenshot at ${program.file}`);
})();

```

Теперь можно внести изменения в файл `package.json`, чтобы у нашей команды было имя и ее можно было вызывать только по нему (без расширения `.js`):

```

"main": "snapshot.js",
"preferGlobal": true,
"bin": {
  "snapshot": "snapshot.js"
},

```

Если теперь выполнить команду `npm link`, то можно будет использовать команду `snapshot` на локальном компьютере непосредственно, не ссылаясь на сам файл:

```
$ snapshot -s http://oreilly.com -f test.png
```

Или можно задействовать более длинный вариант, состоящий из двойного дефиса (`--`) и полного имени:

```
$ snapshot --source http://oreilly.com --file test.png
```

При выполнении утилиты с флагом `-h` или `--help` получим следующее:

```
Usage: snapshot [options]
```

```

Options:
  -h, --help            output usage information
  -V, --version          output the version number
  -s, --source [website] Source website
  -f, --file [filename]  Filename

```

При выполнении следующей команды получим номер версии:

```
$ snapshot -V
```

Commander генерирует все это автоматически, так что можно сосредоточиться на функционале самого приложения.

Для публикации утилиты командной строки в реестре npm используется та же команда, что и для любого другого модуля:

```
$ npm publish
```

19.6. Обеспечение работоспособности экземпляра Node

Задача

Есть среда эксплуатации. Мы хотим запустить в ней приложение Node и сделать так, чтобы оно работало постоянно, а перезапуск приложения не приводил к простоям.

Решение

Использовать модуль `pm2`, который будет перезапускать приложение при каждом отключении:

```
$ pm2 start index.js
```

Обсуждение

Модуль `pm2` — это утилита командной строки, с помощью которой можно не только запустить приложение Node, но и обеспечить его перезапуск всякий раз, когда оно по какой-либо причине отключится.

Модуль `pm2` устанавливается посредством `npm`:

```
$ sudo npm install pm2 -g
```

После установки модуля `pm2` нужно запустить с его помощью приложение Node:

```
$ pm2 start index.js
```

Действие `start` запускает приложение Node в качестве *демона* Unix — процесса, который выполняется в фоновом режиме. У утилиты `pm2` есть и другие параметры, полный список которых можно получить с помощью команды `pm2 --help`. Вот некоторые наиболее полезные из них:

- `-l` — создать файл журнала;
- `-o` — выводить стандартный поток вывода скрипта в заданный выходной файл;
- `-e` — выводить стандартный поток ошибок скрипта в заданный файл ошибок;
- `-n` — задать имя приложения;
- `--watch` — отслеживать изменения и перезапускать приложение.

Для запуска приложения, у которого есть журналы, нужно указать выходные файлы и следующие флаги:

```
$ pm2 start -l forever.log -o out.log -e err.log -n app_name index.js --watch
```

Вот еще некоторые полезные действия `pm2`:

- `stop` — остановить скрипт демона;
- `restart` — перезапустить скрипт демона;
- `delete` — удалить скрипт демона;
- `describe` — вывести подробную информацию о данном приложении;
- `list` — вывести список всех работающих скриптов;
- `monitor` — отслеживать журналы, показатели и информацию о приложении.

Очень полезно бывает добавить скрипт `pm` в файл проекта `package.json`, чтобы команда `pm2` выполнялась автоматически:

```
"scripts": {  
  "start": "pm2 start index.js",  
}
```

В этом случае при выполнении команды `npm start` из корневого каталога проекта будет запускаться приложение через `pm2`. Дополнительный бонус: именно так обычно ведут себя многие приложения Node, размещенные на облачных платформах.

19.7. Отслеживание изменений и перезапуск приложения в процессе разработки на локальном компьютере

Задача

Разработка бывает весьма активной, и тогда перезапуск приложения после каждого изменения кода занимает много времени, а разработчики часто забывают перезапустить приложение.

Решение

Использовать утилиту `nodemon`, которая следит за исходным кодом и перезапускает приложение, когда код изменяется.

Для этого нужно сначала установить `nodemon`:

```
$ npm install -g nodemon
```

Вместо `node` для запуска приложения используем команду `nodemon`:

```
$ nodemon index.js
```

Обсуждение

Утилита `nodemon` следит за файлами, размещенными в каталоге, из которого она была запущена. Когда какой-то из них изменяется, приложение Node перезапускается. Это удобный способ гарантировать, что в выполняемом приложении Node учтены все последние изменения кода.

В режиме эксплуатации приложений `nodemon` обычно не используется. Вместо этого применяются менеджеры процессов, такие как `pm2`, описанный в рецепте 19.6.

Если при запуске приложения нужно передавать какие-то значения, то это делается в командной строке — так же, как в случае с `node`. Только теперь перед этими значениями нужно поставить флаг в виде двух дефисов (`--`). Он сообщает утилите `nodemon`, что все, что идет дальше, нужно игнорировать и передать в приложение:

```
$ nodemon index.js -- -param1 -param2
```

После запуска приложения получим примерно такой ответ:

```
[nodemon] 2.0.2
[nodemon] to restart at any time, enter `rs`
[nodemon] watching dir(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node index.js`
Listening on port 8124
```

Если код изменится, то получим примерно такие сообщения:

```
[nodemon] restarting due to changes...
[nodemon] starting `node index.js`
Server running on 8124/
```

Для того чтобы перезапустить приложение вручную, нужно при работающей утилите `nodemon` ввести в терминале команду `rs`. Также можно внести в конфигурационный файл или файл `package.json` инструкции, в соответствии с которыми утилита будет отслеживать только определенные файлы либо подкаталоги или даже запускать приложения, не относящиеся к среде Node.

Вот пример конфигурации `package.json`, согласно которой `nodemon` будет работать в режиме `verbose` и игнорировать указанные каталоги:

```
{
  "nodemonConfig": {
    "verbose": true,
    "ignore": ["__tests__/*", "docs/*"],
  }
}
```

19.8. Многократное выполнение задач по расписанию

Задача

Выполнять определенную задачу многократно через заданные интервалы.

Решение

Использовать утилиту `node-cron` (<https://oreil.ly/dYQHv>), которая позволяет выполнять задачи в Node по расписанию с применением синтаксиса GNU `crontab`.

При выполнении следующего кода каждую минуту в консоль будет выводиться сообщение:

```
const cron = require('node-cron');

cron.schedule('* * * * *', () => {
  console.log('Log to the console every minute');
});
```

Обсуждение

Для того чтобы использовать модуль `node-cron`, его нужно вначале установить с помощью следующей команды:

```
$ npm install node-cron
```

После этого можно задействовать метод `schedule` и синтаксис `crontab` для создания запланированных задач.

Если прежде вам не встречался синтаксис `crontab`, он может показаться странным. В предыдущем примере я обозначил все поля звездочками, что означает «раз и навсегда». Вместо звездочек можно указать следующие значения (именно в таком порядке):

- секунды (необязательный параметр), от 0 до 59;
- минуты, от 0 до 59;
- часы, от 0 до 23;
- дни месяца, от 0 до 31;
- месяцы, от 0 до 12 (или трехбуквенные сокращенные названия);
- дни недели, от 0 до 7, где 0 или 7 — это воскресенье (либо трехбуквенные сокращенные названия).

При выполнении следующего кода запись в консоль будет выводиться в полночь в первый день каждого месяца:

```
const cron = require('node-cron');

cron.schedule('5 0 1 * *', () => {
  console.log('It is the first of the month!');
});
```

Можно также использовать временные диапазоны. При выполнении следующего кода сообщение будет выводиться в консоль каждый будний день с июня по сентябрь:

```
const cron = require('node-cron');

cron.schedule('0 0 * 6-9 1-5', () => {
  console.log('Summer workdays');
});
```

У утилиты `node-cron` есть два дополнительных параметра: `scheduled` и `timezone`. При выполнении следующего кода текст в консоль будет выводиться в полночь по времени Нью-Йорка:

```
var cron = require('node-cron');

cron.schedule('0 0 * * *', () => {
  console.log('Running a job at midnight ');
}, {
  scheduled: true,
  timezone: "America/New_York"
});
```

Параметр `scheduled` — это параметр типа `Boolean`, по умолчанию он равен `true`. Если же он равен `false`, то запланированные задания выполняться не будут. Параметр `timezone` позволяет указать время выполнения задания для определенного часового пояса. Названия всех часовых поясов вы найдете на странице часовых поясов `Moment.js` (<https://oreil.ly/VhAkl>).

19.9. Тестирование производительности и возможностей приложения WebSockets

Задача

Есть приложение, которое передает часто обновляемую информацию всем подключенным к нему клиентам. Нас интересует производительность приложения и то, насколько хорошо оно способно справляться с высокой нагрузкой.

Решение

Провести *тестирование скорости (производительности)* и *нагрузочное тестирование*. Подробнее об этом читайте в разделе «Обсуждение».

Обсуждение

Благодаря Node, а также WebSockets и другим технологиям двунаправленной коммуникации мы больше не обязаны использовать на веб-страницах таймеры, чтобы обращаться к серверам и получать оттуда свежую информацию. Теперь сервер может сам рассылать всем подключенным клиентам данные по мере их поступления. Такой тип приложений продемонстрирован в примере 17.4, где была создана шкала времени с прокруткой и анимацией. Возникает вопрос: все это, конечно, хорошо, но какова плата? А вдруг при одновременном подключении 10 (100/1000/10 000) клиентов мой сервер рухнет и сгорит? Будут ли все клиенты получать от сервера один и тот же отклик? Ответ на эти вопросы можно получить в результате одного из следующих двух типов тестирования.

- Тестирование скорости или производительности. Проверяется, насколько быстро загружается страница, особенно если сервер находится под нагрузкой.
- Нагрузочное тестирование. Имитируется ситуация конкурентного обращения к странице большого количества клиентов.

Существуют сервисы, выполняющие оба вида тестирования. Если вы представляете большую коммерческую организацию и вам критически важны надежность и производительность приложения, то я настоятельно советую использовать эти сервисы. Некоторые из них, такие как Load Impact (<http://loadimpact.com>), даже предоставляют достойную пробную версию своих продуктов, которой можно пользоваться, прежде чем совершить покупку. Также существуют инструменты, которые можно применять для конкурентных обращений к странице и вывода (в том числе в виде графиков) нагрузочных откликов для каждого из них. Одно из самых популярных средств для тестирования производительности — Selenium (<http://seleniumhq.org>).

В мире Node есть и собственные инструменты, которые легко и быстро устанавливаются с помощью npm. Возможно, они не так красивы, как коммерческие аналоги, зато, безусловно, гораздо дешевле. Один из таких инструментов, на который стоит обратить внимание, — это loadtest — упрощенный вариант ApacheBench (ab). Эта утилита устанавливается глобально:

```
$ npm install -g loadtest
```

После установки утилита loadtest запускается из командной строки. Следующая команда выполняет 200 запросов в секунду (requests per second, rps) с конкурентностью, равной 10:

```
$ loadtest -c 10 --rps 200 http://mysite.com/
```

Есть и другие варианты. Одна из хороших альтернатив для тестирования производительности — ApacheBench. Однако в этих тестах не проверяется соединение WebSockets, так как код JavaScript, в котором содержится запрос к серверу WebSockets, никогда не выполняется.

Еще один вариант — утилита Thor, выполняющая нагрузочное тестирование. Она запускается непосредственно на сервере WebSocket:

```
$ npm install -g thor
$ thor --amount 5000 ws://shelleystoybox.com:8001
```

Есть один эффективный способ — завалить (хм-м, я имел в виду забросать) сервер WebSockets соединениями. Но мы и тогда не получим настоящую двустороннюю коммуникацию, чтобы протестировать все приложение, включая серверную и клиентскую части. Соединения устанавливаются и быстро разрываются, так что это нельзя считать тестированием коммуникации в том виде, в котором она осуществляется при доступе к приложению из клиентских браузеров. Однако в сочетании с другими тестами, которые действительно получают доступ к странице клиента, этот тест помогает определить, возникнут ли проблемы с производительностью при таком количестве запросов на установку соединения (примечание: приложение должно быть активно).

Удаленные данные

Данные окружают нас повсюду. Мы всю жизнь постоянно создаем данные и взаимодействуем с ними — зачастую забавными и неожиданными способами. При построении приложений Node мы тоже постоянно взаимодействуем с данными. Иногда сами создаем их для приложения, а иногда данные вводит в систему пользователь. Не менее часто приходится взаимодействовать с данными, которые поступают в приложение извне. В этой главе описываются наилучшие способы и методики работы с удаленными данными в приложениях Node.

20.1. Получение удаленных данных

Задача

Отправить из приложения Node запрос на удаленный сервер.

Решение

Использовать `node-fetch` — один из самых популярных и распространенных модулей, благодаря которому в Node можно задействовать `window.fetch`. Модуль `node-fetch` устанавливается с помощью `npm`:

```
$ npm install node-fetch
```

Применять `node-fetch` легко:

```
const fetch = require('node-fetch');

fetch('https://oreilly.com')
  .then(res => res.text())
  .then(body => console.log(body));
```

Обсуждение

У модуля `node-fetch` есть API, который довольно точно соответствует свойству браузера `window.fetch`, что позволяет программам Node обращаться к удаленным

ресурсам. Подобно `window.fetch`, этот API поддерживает HTTP-методы GET, POST, DELETE и PUT. В случае GET при получении успешного ответа (код статуса 200) можно обработать возвращенные данные (в этом случае представленные в формате HTML) желаемым способом.

Чтобы отправить запрос к ресурсу в формате JSON, нужно сделать следующее:

```
fetch('https://swapi.dev/api/people/1')
  .then(res => res.json())
  .then(json => console.log(json));
```

Можно также использовать синтаксис `async/await` с блоком `try/catch` для обработки ошибок:

```
(async () => {
  try {
    const response = await fetch('https://swapi.dev/api/people/3');
    const json = await response.json();
    console.log(json);
  } catch (error) {
    console.log(error);
  }
})();
```

С помощью модуля `filesystem` можно перенаправить результат в файл:

```
const fs = require('fs');
const fetch = require('node-fetch');

fetch('https://example.com/image.png')
  .then(res => {
    const dest = fs.createWriteStream('image.png');
    res.body.pipe(dest);
  });
```

Модуль `node-fetch` также поддерживает методы POST, DELETE и PUT, что позволяет передавать данные на сервер. Вот пример запроса POST:

```
// пример тела запроса
const body = {
  id: 1,
  title: "Example"
};

fetch('https://example.com/post', {
  method: 'post',
  body:
    JSON.stringify(body),
  headers: { 'Content-Type': 'application/json' },
})
  .then(res => res.json())
  .then(json => console.log(json));
```



Метод `node-fetch` — это повсеместно используемая и полезная, но не единственная библиотека для получения удаленных данных. Распространенные альтернативы — `Request` (до сих пор популярная, но больше не поддерживаемая), `Got`, `Axios` и `Superagent`.

20.2. Анализ экранных данных

Задача

Обеспечить доступ приложения Node к некоторому содержимому определенного веб-ресурса.

Решение

Выполнить *анализ экранных данных* веб-сайта с помощью модулей `node-fetch` и `Cheerio`.

Сначала установим эти модули:

```
$ npm install node-fetch cheerio
```

Для перехвата экранных данных страницы нужно получить ее содержимое с помощью `node-fetch` и затем с помощью `Cheerio` запросить нужную информацию:

```
const fetch = require('node-fetch');
const cheerio = require('cheerio');

fetch('https://example.com')
  .then(res => res.text())
  .then(body => {
    const $ = cheerio.load(body);
    $('h1').each((i, element) => {
      console.log(element.children[0].data);
    });
  });
```

Обсуждение

Одно из интересных применений Node состоит в *перехвате* экранных данных веб-сайта или другого ресурса, чтобы затем с помощью других функций извлечь из полученных материалов определенную информацию. Для извлечения информации часто используют модуль `Cheerio` — это микрореализация ядра `jQuery`, приспособленная для работы на сервере. В следующем примере создается простое приложение, которое загружает все заголовки постов со страницы блога O'Reilly Radar. Для того чтобы выбрать эти заголовки, мы задействовали `Cheerio`: он находит все ссылки (a), расположенные внутри заголовков (h2), которые, в свою очередь, содержатся в основном блоке `main`. Затем список ссылок выводится в отдельный поток вывода.

```

const fetch = require('node-fetch');
const cheerio = require('cheerio');

fetch('https://www.oreilly.com/radar/posts/')
  .then(res => res.text())
  .then(body => {
    const $ = cheerio.load(body);
    $('main h2 a').each((i, element) => {
      console.log(element.children[0].data);
    });
  });

```

После успешного запроса полученный HTML-код передается в Cheerio с помощью метода `load()`. Результат присваивается переменной с именем в виде знака доллара `$`, чтобы оттуда можно было выбирать элементы по тому же принципу, что и в библиотеке jQuery.

Затем выбираются все элементы, соответствующие шаблону `main h2`, а результат передается в метод `each`, где из каждого заголовка извлекается текст. В итоге в консоль выводятся заголовки всех статей, размещенных на главной странице блога.

Этот способ часто применяют для загрузки данных при отсутствии API. В следующем примере выбираются все определенные ссылки, размещенные на странице, а расположенные по этим ссылкам ресурсы передаются в локальный файл. Я также использовал синтаксис `async/await`, чтобы показать, как его можно применить в данном случае:

```

const path =
  'data-research/mortgage-performance-trends/mortgages-30-89-days-delinquent/';
const url = `https://www.consumerfinance.gov/${path}`;

(async () => {
  try {
    const response = await fetch(url);
    const body = await response.text();
    const $ = cheerio.load(body);
    $('a:contains('state')').each(async (i, element) => {
      const fetchFile = await fetch(element.attribs.href);
      const dest = fs.createWriteStream(`data-${i}.csv`);
      await fetchFile.body.pipe(dest);
    });
  } catch (error) {
    console.log(error);
  }
})();

```

Вначале получаем страницу, расположенную по заданному URL, — в данном случае веб-сайт правительства Соединенных Штатов, на странице которого размещены несколько ссылок на CSV-файлы. Затем с помощью Cheerio выбираем на странице все ссылки, в которых содержится слово *state*. И наконец, скачиваем размещенный по ссылке файл и сохраняем его в виде файла на локальном компьютере.



Инструменты анализа экранных данных стоит иметь под рукой, но применять их следует осмотрительно. Прежде чем анализировать содержимое веб-сайта в рабочем приложении, обязательно уточните правила его использования в разделе Terms of Service (ToS) или получите разрешение у владельца сайта. Кроме того, следите за тем, чтобы случайно не перегрузить серверы хоста, что может привести к атаке типа «отказ в обслуживании» (denial-of-service, DDoS).

20.3. Доступ к данным в формате JSON посредством RESTful API

Задача

Получить доступ к данным, представленным в формате JSON и размещенным на некотором сервисе, посредством API этого сервиса.

Решение

Простейший способ получить доступ к данным, представленным в формате JSON, из приложения Node посредством API — использование библиотеки HTTP-запросов.

В следующем примере я снова задействовал модуль `node-fetch` — примерно так же, как и в рецепте 20.1:

```
const fetch = require('node-fetch');

(async () => {
  try {
    const response = await fetch('https://swapi.dev/api/people/1/');
    const json = await response.json();
    console.log(json);
  } catch (error) {
    console.log(error);
  }
})();
```

У `node-fetch` есть популярная альтернатива — npm-модуль `got`:

```
const got = require('got');

(async () => {
  try {
    const response = await got('https://swapi.dev/api/people/2/');
    console.log(JSON.parse(response.body));
  } catch (error) {
    console.log(error.response.body);
  }
})();
```

Обсуждение

RESTful API — это API без сохранения состояния. Другими словами, в каждом клиентском запросе этого API содержится вся информация, необходимая серверу для того, чтобы вернуть ответ (сохранение каких-либо состояний между запросами не требуется). HTTP-методы используются в RESTful API явно. В RESTful API поддерживается структура URI, построенная по типу каталогов, а передача данных осуществляется определенным образом — как правило, в формате XML или JSON. К HTTP-методам относятся следующие:

- GET — получение данных ресурса;
- PUT — изменение ресурса;
- DELETE — удаление ресурса;
- POST — создание ресурса.

Поскольку мы будем заниматься главным образом получением данных, то сейчас нас интересует только метод GET. А поскольку мы решили ограничиться JSON, то будем использовать клиентские методы для доступа к данным в формате JSON и их преобразования в объекты, которыми затем можно манипулировать в приложениях JavaScript.

Рассмотрим еще один пример.

У Open Exchange Rate (<https://openexchangerates.org>) есть API, с помощью которого можно получать текущие курсы валют, полное и сокращенное название различных валют, а также обменные курсы на определенную дату. Для ограниченного бесплатного доступа к API можно использовать план Forever Free (<https://oreil.ly/TjhFo>).

Можно сделать два запроса к системе для получения текущего курса и полных и сокращенных названий валют, а затем, когда оба запроса будут выполнены, получить и вывести в консоль полные названия и курсы, применяя сокращенные названия как ключи:

```
const fetch = require('node-fetch');
require('dotenv').config();

const id = process.env.APP_ID;

(async () => {
  try {
    const moneyAPI1 = await fetch(
      `https://openexchangerates.org/api/latest.json?app_id=${id}`
    );
    const moneyAPI2 = await fetch(
      `http://openexchangerates.org/api/currencies.json?app_id=${id}`
    );

    const latest = await moneyAPI1.json();
```

```
const names = await moneyAPI2.json();
const keys = Object.keys(latest.rates);
keys.forEach((value, index) => {
  const rate = latest.rates[keys[index]];
  const name = names[keys[index]];
  console.log(`${name} ${rate}`);
});
} catch (error) {
  console.log(error);
}
})();
```



Обратите внимание: вместо значения `id` нужно подставить уникальный ID, который нам присвоит API провайдера при создании учетной записи. В данном примере я использовал модуль `dotenv`, чтобы загрузить значение, сохраненное в `.env`-файле.

Для базовой валюты USD, то есть доллара США, я получил следующие результаты:

```
"Malawian Kwacha 394.899498"
"Mexican Peso 13.15711"
"Malaysian Ringgit 3.194393"
"Mozambican Metical 30.3662"
"Namibian Dollar 10.64314"
"Nigerian Naira 162.163699"
"Nicaraguan Córdoba 26.03978"
"Norwegian Krone 6.186976"
"Nepalese Rupee 98.07189"
"New Zealand Dollar 1.185493"
```

Я использовал в примере синтаксис `async/await`, чтобы отправить запросы и обработать результаты после того, как оба запроса будут выполнены. В реальном приложении лучше кэшировать результаты так долго, как позволяет план (для бесплатного API кэш обновляется раз в час).

Читайте также

В этих примерах мы не применяли экранирование значений, передаваемых в API-запросах в качестве параметров. Если же такое экранирование понадобится, можно использовать встроенный метод `Node querystring.escape()`.

Построение веб-приложений с помощью Express

Express (<https://expressjs.com>) — это легкий веб-фреймворк, который долго был лидером в области разработки веб-приложений для Node. Подобно Sinatra для Ruby и Flask для Python, сам Express имеет минимум функций, но к нему есть множество расширений, благодаря которым можно строить приложения практически любого типа. Express также служит основой, на которую устанавливаются другие фреймворки для разработки веб-приложений, такие как Keystone.js (<https://keystonejs.com>), Sails (<https://sailsjs.com>) и Vulcan.js (<http://vulcanjs.org>). Если вы занимаетесь разработкой веб-приложений для Node, то вам, скорее всего, придется иметь дело с Express. В этой главе мы рассмотрим ряд полезных простых рецептов для работы с Express, которые, будучи расширены, позволяют строить веб-приложения самых разных типов.

21.1. Использование Express для ответов на запросы

Задача

Сделать так, чтобы приложение Node отвечало на HTTP-запросы.

Решение

Установить пакет Express:

```
$ npm install express
```

Для того чтобы использовать Express, нужно подключить этот модуль, вызвать его и указать порт для соединения — все это мы делаем в файле `index.js`:


```
const express = require('express');

const app = express();
const port = process.env.PORT || '3000';

app.listen(port, () => console.log(`Listening on port ${port}`));
```

Для ответа на запрос с применением Express нужно указать маршрут и выдать ответ с помощью метода `.get`:

```
const express = require('express');

const app = express();
const port = process.env.PORT || '3000';

app.get('/', (req, res) => res.send('Hello World'));

app.listen(port, () => console.log(`Listening on port ${port}`));
```

Для обработки статических файлов нужно указать каталог с помощью промежуточного ПО `express.static`:

```
const express = require('express');

const app = express();
const port = process.env.PORT || '3000';

// Промежуточное ПО для статических файлов
// обрабатывает статические файлы из каталога files
app.use(express.static('files'));

app.listen(port, () => console.log(`Listening on port ${port}`));
```

Для того чтобы в ответ на запрос выдать HTML-код, созданный на основе шаблона, нужно вначале установить движок обработки шаблонов:

```
$ npm install pug --save
```

Затем в файле `index.js` создать механизм представлений и указать маршрут, по которому мы будем получать содержимое шаблонов:

```
app.set('view engine', 'pug')

app.get('/template', (req, res) => {
  res.render('template');
});
```

После этого нужно создать файл с шаблоном в подкаталоге `views` проекта. Имя файла шаблона должно соответствовать указанному в `res.render`. В файле `views/template.pug` содержится следующий код:

```
html
  head
    title="Using Express"
  body
    h1="Hello World"
```

Теперь на все запросы по адресу `http://localhost:3000/template` будет возвращаться HTML-код.

Обсуждение

Express — это минималистичный, но очень гибко настраиваемый фреймворк, позволяющий отвечать на HTTP-запросы и строить веб-приложения. В приведенном примере мы выбрали порт `process.env.PORT`, он же порт 3000. Для режима разработки можно указать другой порт, используя переменную среды:

```
$ PORT=7777 node index.js
```

либо создав файл `.env` для модуля Node `dotenv`. При разворачивании приложения иногда нужно указать определенный номер порта для платформы, на которой будет размещаться приложение, или предоставить возможность выбирать номер порта для этой платформы вручную.

С помощью реализованного в Express метода `get` приложение может получать запросы на определенный URI и отвечать на них соответствующим образом. В нашем примере, когда приложение получает запрос к корневому URI (`/`), в ответ передается текст «Hello World»:

```
app.get('/', (req, res) => res.send('Hello World'));
```

Эти ответы могут представлять собой HTML-код, преобразованные в HTML шаблоны, статические файлы и форматированные данные (например, в формате JSON или XML).

Будучи минималистичным по своей сути, Express содержит минимум функционала, но может быть расширен за счет промежуточного ПО. В Express функции промежуточного ПО получают доступ к объектам `request` и `response`. Промежуточное ПО уровня приложения связано с экземпляром объекта `app` посредством `app.use(MIDDLEWARE)`. В нашем примере используется промежуточное ПО для встроенных статических файлов:

```
app.use(express.static('files'));
```

С помощью пакетов промежуточного ПО можно расширять функционал Express в различных направлениях. Так, пакет промежуточного ПО `helmet` позволяет улучшить изначальные параметры безопасности Express:

```
const express = require('express');
const helmet = require('helmet');
```

```
const app = express();  
app.use(helmet());
```

Механизмы обработки шаблонов упрощают написание HTML и позволяют передавать данные из приложения сразу на страницу.

В следующем коде я передаю данные из объекта `userData` в шаблон, который находится в файле `views/user.pug` и доступен по маршруту `/user`:

```
// Объект с данными о пользователе, которые передаются в шаблон  
const userData = {  
  name: 'Adam',  
  email: 'adam@jseverywhere.io',  
  avatar: 'https://s.gravatar.com/avatar/33aab819d1ffa11fc4b31a4eebaf0c5a?s=80'  
};  
  
// Наполняем шаблон данными о пользователе  
app.get('/user', (req, res) => {  
  res.render('user', { userData });  
});
```

Теперь можно перейти в шаблон и использовать помещенные туда данные:

```
html  
  head  
    title User Page  
  body  
    h1 #{userData.name} Profile  
    ul  
      li  
        image(src=userData.avatar)  
      li #{userData.name}  
      li #{userData.email}
```

Механизм шаблонов Pug поддерживается той же командой, которая разрабатывает ядро Express. Этот механизм широко используется в приложениях Express. Однако основанный на пробелах синтаксис этих шаблонов подходит не для всех задач. Отличная альтернатива Pug — шаблоны EJS (<https://ejs.co>), синтаксис которых ближе к HTML. Для того чтобы переделать предыдущий пример для EJS, сделаем следующее.

Вначале установим пакет `ejs`:

```
$ npm install ej
```

Затем укажем, что в нашем приложении Express используется механизм представлений EJS:

```
app.set('view engine', 'ejs');
```

И создадим файл `views/user.ejs`:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>User Page</title>
  </head>
  <body>
    <h1><%= userData.name %> Profile</h1>
    <ul>
      <li><img src=<%= userData.avatar %> /></li>
      <li><%= userData.name %></li>
      <li><%= userData.email %></li>
    </ul>
  </body>
</html>
```

21.2. Использование Express-Generator

Задача

Мы хотим применять Express для управления данными приложения на стороне сервера, но не хотим выполнять всю установку вручную.

Решение

Для быстрого запуска приложения Express используют Express-Generator — утилиту командной строки, которая генерирует базовую инфраструктуру типичного приложения Express. Для этого вначале нужно создать рабочий каталог, чтобы с помощью Express-Generator безопасно разместить в нем подкаталог нового приложения. Затем с помощью `npx` выполнить команду `express-generator`:

```
$ npx express-generator --pug --git
```

Я передал в команду два параметра: `--pug`, чтобы использовать механизм шаблонов Pug, и `--git`, чтобы в каталоге проекта был создан стандартный файл `.gitignore`. Для того чтобы получить полный список параметров, следует запустить генератор с флагом `-h`:

```
$ npx express-generator -h
```

Генератор создает каталог с несколькими подкаталогами, несколько базовых файлов, на основе которых можно начать проект, и файл `package.json` для всех зависимостей. Для установки зависимостей требуется перейти в созданный каталог и выполнить следующую команду:

```
$ npm install
```

После установки всех зависимостей можно запустить приложение:

```
$ npm start
```

Теперь созданное приложение Express доступно по IP-адресу или имени домена через порт 3000, который используется приложениями Express по умолчанию.

Обсуждение

Express представляет собой фреймворк веб-приложения на базе Node с поддержкой различных механизмов шаблонов и препроцессоров CSS. В предложенном для примера решении я создал приложение, в котором выбрал механизм шаблонов Pug (используемый по умолчанию) с простым CSS (без препроцессора — тоже вариант, предлагаемый по умолчанию). При построении приложения с нуля у нас есть более широкий выбор, однако Express поддерживает только следующие механизмы шаблонов:

- `--ejs` — поддержка механизма шаблонов EJS;
- `--pug` — поддержка механизма шаблонов Pug;
- `--hbs` — поддержка механизма шаблонов Handlebar;
- `--hogan` — поддержка механизма шаблонов Hogan.js.

Express также поддерживает следующие препроцессоры CSS:

- `express --css sass` — поддержка Sass;
- `express --css less` — поддержка Less;
- `express --css stylus` — поддержка Stylus;
- `express --css compass` — поддержка Compass.

Если не указать препроцессор CSS, то по умолчанию будет использоваться простой CSS.

Express также предполагает, что изначально каталог проекта пуст. Если это не так, то нужно использовать флаг `-f` или `--force`, чтобы Express все равно сгенерировал нужное содержимое.

Express создает систему каталогов со следующей структурой (без учета `node_modules`):

```
app.js
package-lock.json
package.json
/bin
  www
/node_modules
/public
  /images
  /javascripts
  /stylesheets
    style.css
    style.styl
```

```
/routes
  index.js
  users.js
/views
  error.pug
  index.pug
  layout.pug
```

Ядром приложения Express является файл `app.js`. В нем содержатся ссылки на все необходимые библиотеки:

```
var createError = require('http-errors');
var express = require('express');
var path = require('path');
var cookieParser = require('cookie-parser');
var logger = require('morgan');

var indexRouter = require('./routes/index');
var usersRouter = require('./routes/users');
```



Мы договорились использовать для объявления переменных ключевые слова `const` и `let`, однако на момент написания книги в генераторе Express все еще было указано ключевое слово `var`.

Приложение Express создается в следующей строке:

```
var app = express();
```

Затем путем объявления переменных `views` и `view engine` выбирается механизм представлений Pug:

```
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'pug');
```

Далее с помощью `app.use()` вызываются несколько промежуточных программ. *Промежуточное ПО* — это функционал, который выполняется между исходным запросом и маршрутизацией и обрабатывает определенные виды запросов. В промежуточном ПО действует правило: если в качестве первого параметра не передан путь, то по умолчанию выбирается путь `/` и по нему загружаются функции промежуточного ПО. Рассмотрим следующий автоматически сгенерированный код:

```
app.use(logger('dev'));
app.use(express.json());
app.use(express.urlencoded({ extended: false }));
app.use(cookieParser());
app.use(express.static(path.join(__dirname, 'public')));
```

В нескольких первых вызовах промежуточное ПО загружается при каждом запросе приложения. В частности, промежуточное ПО упрощает разработку системы регистрации пользователей, а также включает в себя синтаксические

анализаторы *URL-кодированных* тел запроса и тел запроса в формате JSON. Конкретный путь назначается только тогда, когда мы имеем дело с записью `static`: промежуточное ПО для запросов статических файлов загружается при запросе к *общедоступному* каталогу.

Маршрутизация выполняется следующим образом:

```
app.use('/', indexRouter);
app.use('/users', usersRouter);
```

Веб-запрос верхнего уровня (/) направляется в модуль `routes`, а все пользовательские запросы (/users) — в модуль `users`.



Подробнее о маршрутизации в Express читайте в рецепте 21.3.

Затем выполняется обработка ошибок. Прежде всего это ошибка 404, которая возникает при попытке запроса к несуществующему веб-ресурсу:

```
app.use(function(req, res, next) {
  next(createError(404));
});
```

Далее идут еще несколько обработчиков ошибок для режима разработки и эксплуатации:

```
app.use(function(err, req, res, next) {
  // Присваиваем значения локальным переменным
  // таким образом, чтобы ошибка выдавалась только на этапе разработки
  res.locals.message = err.message;
  res.locals.error = req.app.get('env') === 'development' ? err : {};

  // Генерируем страницу с сообщением об ошибке
  res.status(err.status || 500);
  res.render('error');
});
```

В последней строке созданного файла свойству `module.exports` присваивается значение `app`:

```
module.exports = app;
```

В подкаталоге `routes` указан маршрут по умолчанию — он ссылается на файл `routes/index.js`:

```
var express = require('express');
var router = express.Router();

/* GET-запрос к начальной странице */
router.get('/', function(req, res, next) {
```

```
res.render('index', { title: 'Express' });
});
```

```
module.exports = router;
```

В данном файле происходит следующее: маршрутизатор Express перенаправляет все HTTP-запросы GET по адресу /, где находится функция обратного вызова. Там в ответ на запрос передается представление, которое генерируется для определенной страницы ресурса — в отличие от файла `routes/users.js`, где в ответ на запрос передается не представление, а текстовое сообщение:

```
var express = require('express');
var router = express.Router();

/* Перехват событий GET, поступающих от пользователей */
router.get('/', function(req, res, next) {
  res.send('respond with a resource');
});

module.exports = router;
```

Что происходит при создании представления при первом запросе? В подкаталоге `views` есть три файла Pug: один для обработки ошибок, другой — со структурой страницы и еще один, `index.pug`, с помощью которого, собственно, и создается страница. В файле `index.pug` содержится следующий код:

```
extends layout

block content
  h1= title
  p Welcome to #{title}
```

Файл `index.pug` является расширением файла `layout.pug`, в котором содержится следующий код:

```
doctype html
html
  head
    title= title
    link(rel='stylesheet', href='/stylesheets/style.css')
  body
    block content
```

В файле `layout.pug` описана общая структура страницы независимо от содержимого, включая ссылку на автоматически генерируемый CSS-файл. Содержимое размещено в разделе `block content`. Формат содержимого, которое находится в файле `index.js`, указан в одноименном параметре раздела `block content`.

В двух файлах Pug определена простейшая веб-страница с элементом `h1`, в котором находится переменная заголовка, и абзацем с текстом приветствия. Стендериванная по умолчанию страница показана на рис. 21.1.



Механизм шаблонов Pug (раньше он назывался Jade) получил широкое распространение благодаря Express. Pug предлагает минималистичный принцип построения шаблонов, при котором вместо привычных HTML-тегов используются пробелы. Такой подход нравится не всем, поэтому у Pug есть ряд альтернатив (Handlebars, Hogan.js и EJS), синтаксис которых больше похож на HTML.

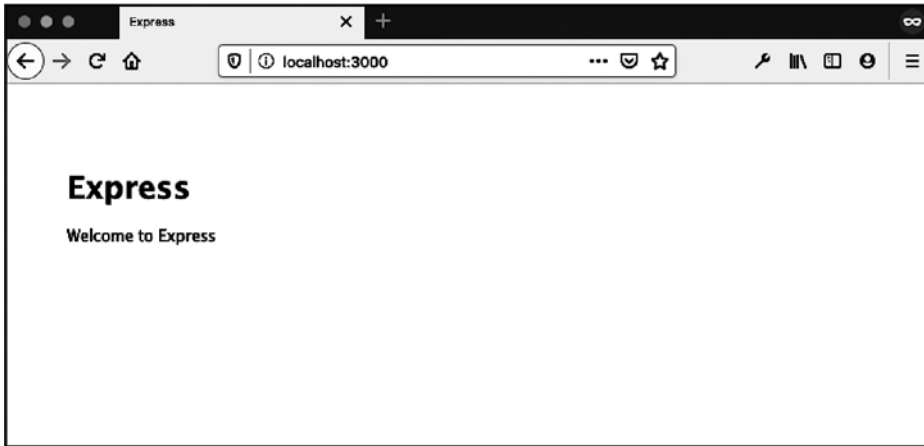


Рис. 21.1. Веб-страница, сгенерированная Express

Страница выглядит не особенно интересной. Однако на ней хорошо видно, как взаимодействуют разные части приложения: маршрутизатор направляет запрос в соответствующий модуль, тот передает ответ в созданное для этого представление, которое, в свою очередь, использует полученные данные для генерации веб-страницы. Если сделать следующий запрос:

```
http://yourdomain.com:3000/users
```

то вместо сгенерированного представления получим простое текстовое сообщение.

По умолчанию Express запускается в режиме разработки. Для того чтобы переключиться на режим эксплуатации, нужно присвоить *переменной среды* `NODE_ENV` значение `production`. В Linux и Unix для этого можно использовать следующий код:

```
$ export NODE_ENV=production
```

21.3. Задача маршрутизации

Задача

Перенаправлять пользователей на разные ресурсы приложения в зависимости от запроса.

Решение

Используя маршруты Express, переходить на определенные ресурсы в зависимости от пути и параметров запроса:

```
// Ответ для разных путей запроса
app.get('/', (req, res) => res.send('Hello World'));
app.get('/users', (req, res) => res.send('Hello users'));

// Параметры
app.get('/users/:userId', (req, res) => {
  res.send(`Hello user ${req.params.userId}`);
});
```

Обсуждение

В Express ответ пользователю возвращается как реакция на HTTP-запрос. В предыдущих примерах я применял запросы GET, но в Express поддерживаются еще несколько методов. Наиболее часто используются следующие:

- `app.get` — запрос данных;
- `app.post` — передача данных;
- `app.put` — передача или изменение данных;
- `app.delete` — удаление данных.

```
app.post('/new', (req, res) => {
  res.send('POST request to the `new` route');
});
```

Часто для одного и того же маршрута нужно использовать разные методы. Для этого можно построить следующую цепочку:

```
app
  .route('/record')
  .get((req, res) => {
    res.send('Get a record');
  })
  .post((req, res) => {
    res.send('Add a record');
  })
  .put((req, res) => {
    res.send('Update a record');
  });
```

В запросах часто встречаются параметры со специфическими значениями, которые используются в приложении. Для их определения в URL применяется двоеточие (:):

```
app.get('/users/:userId', (req, res) => {
  res.send(`Hello user ${req.params.userId}`);
});
```

В этом примере, когда пользователь переходит по URL `/users/adam123`, браузер возвращает в ответ текст `Hello user adam123`. Это очень простой пример, однако с помощью параметров URL можно получать данные из базы и передавать информацию в шаблон.

Мы также можем определять форматы параметров запроса. В следующем примере я с помощью регулярного выражения установил ограничение для параметра `noteId` — теперь это могут быть только шестизначные целые числа:

```
app.get('^/users/:userId/notes/:noteId([0-9]{6})', (req, res) => {
  res.send(`This is note ${req.params.noteId}`);
});
```

С помощью регулярных выражений можно также задать весь маршрут:

```
app.get(/.*day$/, (req, res) => {
  res.send(`Every day feels like ${req.path}`);
});
```

В этом примере будут маршрутизироваться все запросы, которые заканчиваются на `day`. Например, при разработке на локальном компьютере в ответ на запрос `http://localhost:3000/Sunday` на странице появится текст `Every day feels like Sunday`.

21.4. Работа с OAuth

Задача

Получить доступ из приложения Node к стороннему API, которое требует авторизации — а именно авторизации OAuth, — такому как GitHub, Facebook или Twitter.

Решение

Внедрить в приложение клиент OAuth. Для этого приложение должно соответствовать требованиям OAuth, которые предъявляет поставщик ресурсов. Подробнее об этом читайте в обсуждении.

Обсуждение

OAuth — это фреймворк авторизации, который используется в большинстве популярных социальных сетей и приложений для облачного хранения данных. Если вам когда-либо приходилось заходить на сайт, где требовалась авторизация для доступа к данным, размещенным на стороннем сервисе, таком как GitHub, — вы проходили *процесс* авторизации OAuth.

Существуют две версии OAuth, 1.0 и 2.0, и они не совместимы между собой. OAuth 1.0 основан на коммерческих API, разработанных Flickr и Google, и жестко

ориентирован на веб-страницы. Он так и не сумел преодолеть барьер между веб-страницами, мобильными приложениями и сервисами. Если нужен доступ к ресурсам из мобильного приложения, то сначала такое приложение должно зарегистрировать пользователя в мобильном браузере, а затем скопировать токены доступа в приложение. Другой недостаток OAuth 1.0 состоит в требовании, чтобы сервер авторизации был тем же сервером, что и сервер ресурсов, а это не допускает масштабирования, когда речь идет о таких провайдерах сервисов, как Twitter, Facebook и Amazon.

OAuth 2.0 предлагает более простой процесс авторизации, а также предусматривает разные типы (процессы) авторизации для разных обстоятельств. Впрочем, некоторые считают, что это сделано в ущерб безопасности, так как предъявляются разные требования к шифрованию токенов хеша и строк запроса.

Большинству разработчиков не придется создавать сервер OAuth 2.0. Это выходит за рамки книги, не говоря уже о данном рецепте. Однако в приложения часто встраивают клиент OAuth (1.0 или 2.0) для того или иного сервиса, поэтому я намерен продемонстрировать разные способы применения OAuth. Однако для начала рассмотрим различия между авторизацией и аутентификацией.

Авторизация — это не аутентификация

Авторизация говорит: «Я даю этому приложению доступ к моим ресурсам, расположенным на вашем сервере». Аутентификация — это процесс, цель которого — выяснить, действительно ли вы являетесь тем человеком, которому принадлежит данная учетная запись и который имеет право управлять данными ресурсами. Предположим, я хочу прокомментировать статью на веб-сайте газеты. Скорее всего, мне будет предложено войти в систему посредством какого-либо сервиса. Если для этого я воспользуюсь своей учетной записью Facebook, то новостной сайт, очевидно, запросит у Facebook какие-то данные обо мне.

Прежде всего новостной сайт должен аутентифицировать меня как пользователя Facebook, у которого есть учетная запись Facebook. Другими словами, новостной сайт должен убедиться, что я не случайный человек, который пришел, чтобы оставить анонимный комментарий. Затем новостной сайт захочет получить от меня что-то взамен возможности комментировать записи: он захочет данные обо мне. Возможно, он запросит разрешение предлагать мне какие-то публикации (если я размещу свой комментарий не только на новостном сайте, но и в Facebook). Это запрос и на аутентификацию, и на авторизацию.

Если я еще не вошел в Facebook, то мне понадобится это сделать. Если уже вошел, то Facebook задействует мое имя пользователя и пароль, чтобы меня аутентифицировать — подтвердить, что я действительно владелец данного аккаунта Facebook. После того как я вошел в Facebook, он спросит меня, согласен ли я предоставить новостному сайту авторизацию — право доступа к тем ресурсам, которые сайт хочет получить. Если я соглашусь (потому что очень уж хочу

оставить комментарий под той публикацией), Facebook предоставит новостному сайту авторизацию, после чего между новостным сайтом и моей учетной записью Facebook установится постоянное соединение, которое можно будет увидеть на странице параметров учетной записи Facebook. Теперь я могу оставить комментарий и комментировать остальные публикации, пока не выйду из Facebook или не отзову авторизацию Facebook.

Разумеется, это не означает, что Facebook или новостной сайт аутентифицируют именно меня, то есть знают, кто я такой. В данном случае аутентифицировать означает установить, что именно я являюсь владельцем данной учетной записи Facebook. Единственный случай, когда имеет место реальная аутентификация личности в социальных сетях, — это аутентифицированные учетные записи Twitter для знаменитостей.

Наша задача разработки упрощается, благодаря тому что программы авторизации — это зачастую те же самые программы, которые используются для идентификации личности. Таким образом, нам не придется иметь дело с двумя разными библиотеками/модулями/системами JavaScript. Также для приложений Node написано несколько отличных модулей OAuth (1.0 и 2.0). Один из самых популярных таких модулей — Passport (<http://www.passportjs.org>). Специально для системы Passport созданы расширения для различных сервисов авторизации. Однако существуют и очень простые клиенты OAuth, которые обеспечивают базовую авторизацию для большинства сервисов, и несколько модулей, созданных специально для конкретных сервисов.



Система Passport.js описана в рецепте 21.5. Подробнее о Passport и стратегиях, применяемых этой системой для различных серверов, читайте на ее веб-сайте.

А теперь перейдем к собственно технологии.

Предоставление учетных данных клиента

В настоящее время существует несколько веб-ресурсов, которые предоставляют API и не требуют каких-либо данных для авторизации. Это означает, что на стороне конечного пользователя нужно реализовать двустороннюю процедуру — запрашивать авторизацию для доступа сервиса к учетной записи пользователя. После авторизации приложение получает доступ к данным пользователя. Проблема состоит в том, что иногда требуется доступ только для чтения, без права изменять данные и без интерфейса входа в систему, так что пользователь не обязан предоставлять для этого специальное разрешение.

В OAuth 2.0 специально для такого типа авторизации предусмотрена процедура *Client Credentials Grant*. На рис. 21.2 представлена диаграмма этого упрощенного типа авторизации.

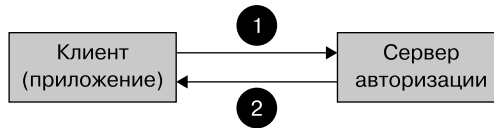


Рис. 21.2. Процедура авторизации Client Credentials Grant

В Twitter предоставляется так называемая авторизация только для приложений, в основе которой лежит OAuth 2.0 Client Credentials Grant. С помощью этого типа авторизации можно получить доступ к API Twitter Search.

В следующем примере я реализовал авторизацию с помощью модуля `Node oauth` — простейшего модуля авторизации, который поддерживает обе процедуры авторизации OAuth, и 1.0, и 2.0:

```

const OAuth = require('oauth');
const fetch = require('node-fetch');
const { promisify } = require('util');

// Читаем ключи Twitter из файла .env
require('dotenv').config();

// Конечная точка API для поиска в Twitter и запрос с нашим критерием поиска
const endpointUrl = 'https://api.twitter.com/2/tweets/search/recent';
const query = 'javascript';

async function getTweets() {
  // Открытый и секретный ключи пользователя, переданные
  // через переменные среды
  const oauth2 = new OAuth.OAuth2(
    process.env.TWITTER_CONSUMER_KEY,
    process.env.TWITTER_CONSUMER_SECRET,
    'https://api.twitter.com/',
    null,
    'oauth2/token',
    null
  );

  // Получаем учетные данные из Twitter
  const getOAuthAccessToken = promisify(
    oauth2.getOAuthAccessToken.bind(oauth2)
  );

  const token = await getOAuthAccessToken('', {
    grant_type: 'client_credentials'
  });

  // Запрашиваем данные, используя полученный токен
  const res = await fetch(`${endpointUrl}?query=${query}`, {
    headers: {
      authorization: `Bearer ${token}`
    }
  });
}
  
```

```

    });

    const json = await res.json();
    return json;
}

(async () => {
  try {
    // Делаем запрос
    const response = await getTweets();
    console.log(response);
  } catch (e) {
    console.log(e);
    process.exit(-1);
  }
  process.exit();
})();

```

Для того чтобы использовать API авторизации из Twitter, клиент должен зарегистрировать приложение в Twitter, который предоставляет клиенту *открытый* и *секретный* ключи.

Затем с помощью модуля `oauth` создается объект `OAuth2`, в который передаются следующие данные:

- открытый ключ;
- секретный ключ;
- базовый URI API (URI API без строки запроса);
- значение `null`, которое сообщает OAuth, что следует использовать путь по умолчанию `/oauth/authorize`;
- путь токена доступа;
- значение `null`, поскольку нестандартные заголовки не применяются.

Модуль `oauth` получает эти данные и формирует запрос POST, который передается в Twitter вместе с открытым и секретным ключами, а также областью видимости ответа. В документации Twitter приводится следующий пример запроса POST для получения токена доступа (разрывы строк вставлены для удобства чтения):

```

POST /oauth2/token HTTP/1.1
Host: api.twitter.com
User-Agent: My Twitter App v1.0.23
Authorization: Basic eHZ6MWV2RlM0d0VFUFRHRUZQSEJvZzpmOHFxOVBAeVJn
                NmllS0dFS2hab2xHQzB2SldMdzhpRUo4OERSZHlPZW==
                Content-Type: application/x-www-form-urlencoded; charset=UTF-8
Content-Length: 29
Accept-Encoding: gzip

grant_type=client_credentials

```

В ответе содержится токен доступа (здесь разрывы строк тоже вставлены для удобства чтения):

```
HTTP/1.1 200 OK
Status: 200 OK
Content-Type: application/json; charset=utf-8
...
Content-Encoding: gzip
Content-Length: 140

{"token_type": "bearer", "access_token": "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA%2FAAAAAAAAAAAAAAAAAAAAA%3DAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"} }
```

Этот токен доступа нужно указывать во всех последующих запросах к API. На этом авторизация закончена — процесс действительно очень простой. Кроме того, поскольку авторизация выполняется на уровне приложения, она не требует идентификации пользователя, так что для него процесс становится менее раздражающим.



Twitter предоставляет отличную документацию. Рекомендую прочитать раздел Application-only authentication overview (<https://oreil.ly/Mikyl>).

Авторизация для чтения и записи в OAuth 1.0

Аутентификация Application-Only отлично подходит для доступа к данным для чтения. Но что делать, если нужно предоставить доступ к данным определенного пользователя или даже доступ с правом изменения данных? Для этого нужна полная авторизация OAuth. В этом разделе мы снова воспользуемся Twitter для демонстрации, так как в нем задействуется OAuth 1.0. Рассмотрим OAuth 2.0 в следующем рецепте.



Я использую название OAuth 1.0, хотя сервис Twitter основан на OAuth Core 1.0 Revision A (<http://oauth.net/core/1.0a>). Однако гораздо проще писать кратко — OAuth 1.0.

OAuth 1.0 требует цифровую подпись. Этапы ее получения графически представлены на рис. 21.3. В случае с Twitter нужно сделать следующее.

1. Составить строку из HTTP-метода и базового URI, без строки запроса.
2. Собрать все параметры, включая открытый ключ, данные запроса, одноразовый номер (nonce), метод подписи и т. п.
3. Создать базовую строку подписи — она состоит из собранных нами данных, правильно объединенных в строку и правильно закодированных.

4. Создать ключ подписи, представляющий собой комбинацию открытого ключа и токена секрета OAuth, и снова правильно закодировать.
5. Передать базовую строку подписи и ключ подписи в алгоритм хеширования HMAC-SHA1. Он вернет двоичную строку для дальнейшего кодирования.



Рис. 21.3. Процесс авторизации OAuth 1.0

Этот процесс необходимо выполнять для *каждого* запроса. К счастью, есть модули и библиотеки, способные выполнить всю эту рутинную работу за нас. Не знаю, как вы, а я быстро потерял бы интерес к внедрению данных и сервисов Twitter в свои приложения, если бы мне пришлось все это делать самому.

Наш старый знакомый `oauth` обеспечивает базовую поддержку OAuth 1.0, но на этот раз нам не придется писать код самостоятельно. Для этого есть другой модуль, `node-twitter-api`, в котором содержатся все части OAuth. От нас требуется только создать объект `node-twitter-api`, передать в него открытый и секретный ключи клиента, а также URL для обратного вызова или перенаправления, который требуют сервисы ресурсов в процессе авторизации. После обработки объекта `request` по этому URL нам будут предоставлены токен и секретный код, необходимые для доступа к API. Их мы будем передавать при каждом запросе.

Модуль `twitter-node-api` — это тонкая надстройка над REST API: для того чтобы сделать запрос, мы экстраполируем функцию из API. Конечная точка REST API, из которой можно переслать изменение состояния, выглядит так:

```
https://api.twitter.com/1.1/statuses/update.json
```

Для этого нам понадобится функция экземпляра объекта `twitter-node-api` `statuses()`, первым параметром которой является `update`:

```
twitter.statuses('update', {
  "status": "Hi from Shelley's Toy Box. (Ignore--developing Node app)"
}, atoken, atokensec, function(err, data, response) {...});

twitter.statuses(
  'update',
  {
    status: 'Ignore learning OAuth with Node'
```

```
    },  
    tokenValues.atoken,  
    tokenValues.atokensec,  
    (err, data) => { ... });
```

В функцию обратного вызова передаются следующие аргументы: все возможные ошибки, запрошенные данные (если есть) и необработанный ответ.

Полный код показан в примере 21.1. В качестве сервера применен Express. Пользователю возвращается примитивная веб-страница, после чего задействуется еще один модуль.

Пример 21.1. Полная авторизация приложения в Twitter посредством OAuth 1.0

```
const express = require('express');  
const TwitterAPI = require('node-twitter-api');  
  
require('dotenv').config();  
  
const port = process.env.PORT || '8080';  
  
// Ключи и URL обратного вызова формируются в Twitter Dev Center  
const twitter = new TwitterAPI({  
  consumerKey: process.env.TWITTER_CONSUMER_KEY,  
  consumerSecret: process.env.TWITTER_CONSUMER_SECRET,  
  callback: 'http://127.0.0.1:8080/oauth/callback'  
});  
  
// Объект для хранения полученных токенов  
const tokenValues = {};  
  
// Twitter OAuth API URL  
const twitterAPI = 'https://api.twitter.com/oauth/authenticate';  
  
// Простой HTML-шаблон  
const menu =  
  '<a href="/post/status/">Say hello</a><br />' +  
  '<a href="/get/account/">Account Settings<br />';  
  
// Создаем приложение Express  
const app = express();  
  
// Запрашиваем у Twitter разрешение при обращении по маршруту /  
app.get('/', (req, res) => {  
  twitter.getRequestToken((error, requestToken, requestTokenSecret) => {  
    if (error) {  
      console.log(`Error getting OAuth request token : ${error}`);  
      res.writeHead(200);  
      res.end(`Error getting authorization${error}`);  
    } else {  
      tokenValues.token = requestToken;  
      tokenValues.tokensec = requestTokenSecret;
```

```

        res.writeHead(302, {
            Location: `${twitterAPI}?oauth_token=${requestToken}`
        });
        res.end();
    }
    });
});

// Обращаемся по URL обратного вызова,
// сформированному в Twitter Developer Center
app.get('/oauth/callback', (req, res) => {
    twitter.getAccessToken(
        tokenValues.token,
        tokenValues.tokensec,
        req.query.oauth_verifier,
        (err, accessToken, accessTokenSecret) => {
            res.writeHead(200);
            if (err) {
                res.end(`problems getting authorization with
                    Twitter${err}`);
            } else {
                tokenValues.atoken = accessToken;
                tokenValues.atokensec = accessTokenSecret;
                res.end(menu);
            }
        }
    );
});

// Передаем изменение состояния от аутентифицированных
// и авторизованных пользователей
app.get('/post/status/', (req, res) => {
    twitter.statuses(
        'update',
        {
            status: 'Ignore teaching OAuth with Node'
        },
        tokenValues.atoken,
        tokenValues.atokensec,
        (err, data) => {
            res.writeHead(200);
            if (err) {
                res.end(`problems posting ${JSON.stringify(err)}`);
            } else {
                res.end(`posting status: ${JSON.stringify(data)}<br />${menu}`);
            }
        }
    );
});

// Получаем информацию об учетной записи
// для аутентифицированного и авторизованного пользователя

```

```
app.get('/get/account/', (req, res) => {
  twitter.account(
    'settings',
    {},
    tokenValues.atoken,
    tokenValues.atokensec,
    (err, data) => {
      res.writeHead(200);
      if (err) {
        res.end(`problems getting account ${JSON.stringify(err)}`);
      } else {
        res.end(`<p>${JSON.stringify(data)}</p>${menu}`);
      }
    }
  );
});

app.listen(port, () => console.log(`Listening on port ${port}!`));
```

В этом приложении нас интересуют следующие маршруты:

- / — страница, с которой происходит перенаправление в Twitter для авторизации;
- /auth — зарегистрированный в приложении URL обратного вызова или перенаправления, который передается в запросе;
- /post/status/ — передача состояния в учетную запись Twitter;
- /get/account/ — получение информации об учетной записи пользователя.

Во всех случаях применяется соответствующая функция модуля `node-twitter-api`:

- / — с помощью функции `getRequestToken()` получить токен запроса и секретный код токена запроса;
- /auth/ — получить токен доступа к API и секретный код токена, сохранить их в локальном кэше, вывести меню;
- /post/status/ — вызвать функцию `status()`, передав в нее первым параметром слово `update`, а также состояние, токен и секретный код доступа, а также функцию обратного вызова;
- /get/account/ — вызвать функцию `account()`, передав в нее первым параметром слово `settings`, а затем пустой объект, так как для данного запроса данные не нужны, токен доступа, секретный код и функцию обратного вызова.

В ответ откроется страница авторизации Twitter, показанная на рис. 21.4. На рис. 21.5 приведена веб-страница с информацией об учетной записи вашего почтового слуги.

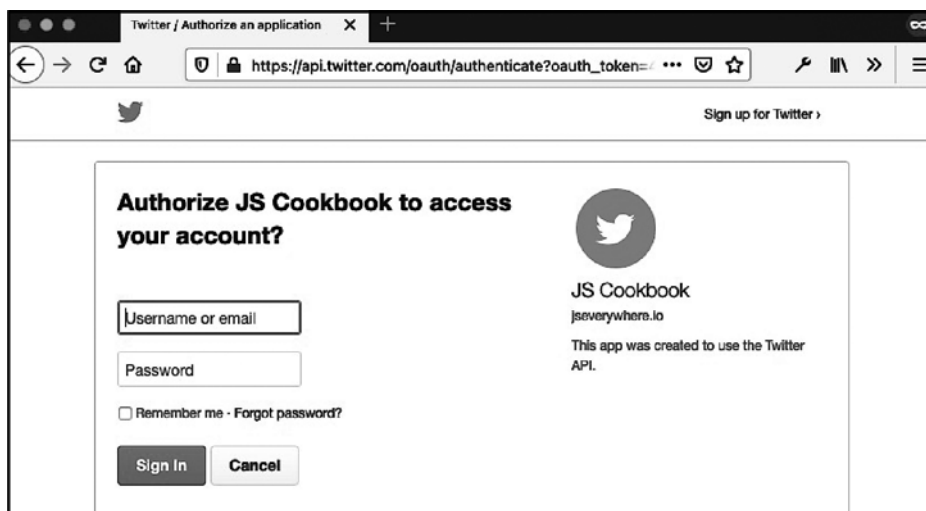


Рис. 21.4. Страница авторизации Twitter, на которую осуществляется переход из приложения, описанного в этом рецепте

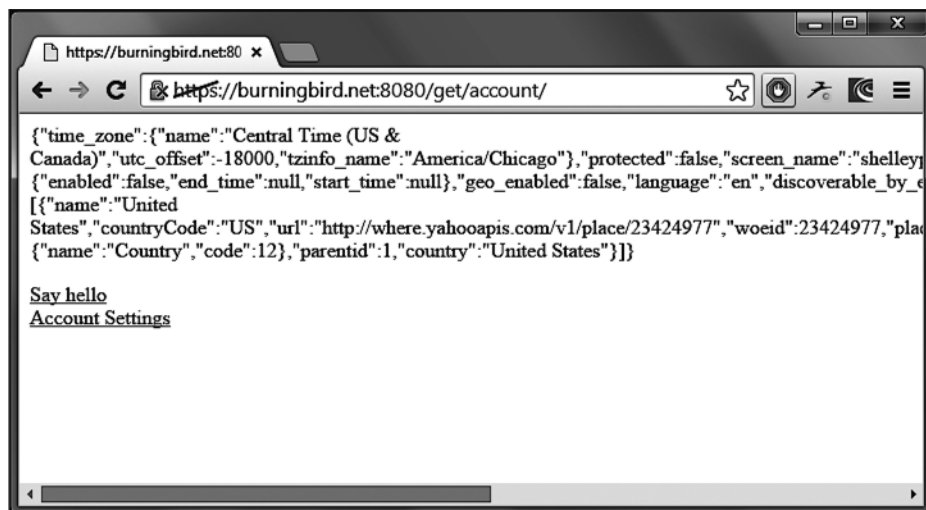


Рис. 21.5. Вывод в приложении данных о пользователе, полученных из учетной записи Twitter



Несмотря на то что модуль `node-twitter-api` уже не поддерживается столь активно, как раньше, вы все еще можете ближе с ним познакомиться на странице модуля в репозитории GitHub (<https://github.com/reneraab/node-twitter-api>). Другие библиотеки поддерживаются более энергично и предоставляют подобный функционал, однако я считаю, что `node-twitter-api` — простейший функциональный пример, который лучше всего подходит для демонстрации.

21.5. Аутентификация пользователей в OAuth 2 с помощью Passport.js

Задача

Организовать аутентификацию пользователей в приложении посредством стороннего сервиса.

Решение

Воспользоваться библиотекой Passport.js в сочетании со стратегией, соответствующей выбранному провайдеру аутентификации. В приведенном здесь примере я применил стратегию GitHub, но этот процесс будет примерно одинаковым для любого провайдера OAuth 2, включая Facebook, Google и Twitter.

Для того чтобы задействовать стратегию GitHub, нужно зайти на веб-сайт GitHub и зарегистрировать там новое приложение OAuth (<https://github.com/settings/applications/new>). После того как приложение будет зарегистрировано, в него можно встроить код OAuth для Passport.js.

Для этого нужно вначале настроить стратегию Passport, указав, в частности, предоставленные GitHub идентификатор и секретный код клиента, а также выбранный вами URL обратного вызова:

```
const express = require('express');
const passport = require('passport');
const { Strategy } = require('passport-github');

passport.use(
  new Strategy(
    {
      clientID: GITHUB_CLIENT_ID,
      clientSecret: GITHUB_CLIENT_SECRET,
      callbackURL: 'login/github/callback'
    },
    (accessToken, refreshToken, profile, cb) => {
      return cb(null, profile);
    }
  )
);
```

Для того чтобы вернуться к исходному состоянию аутентификации в процессе HTTP-запросов, Passport требует выполнять сериализацию и десериализацию данных о пользователе:

```
passport.serializeUser((user, cb) => {
  cb(null, user);
});

passport.deserializeUser((obj, cb) => {
```

```
    cb(null, obj);  
  });
```

Чтобы сохранить состояние регистрации пользователя между сессиями браузера, можно задействовать промежуточное ПО `express-session`:

```
app.use(  
  require('express-session')({  
    secret: SESSION_SECRET,  
    resave: true,  
    saveUninitialized: true  
  })  
);  
  
app.use(passport.session());
```

После этого можно аутентифицировать запросы посредством `passport.authenticate`:

```
app.use(passport.initialize());  
  
app.get('/login/github', passport.authenticate('github'));  
  
app.get(  
  '/login/github/callback',  
  passport.authenticate('github', { failureRedirect: '/login' } ),  
  (req, res) => {  
    res.redirect('/');  
  }  
);
```

А так можно ссылаться на объект `user` из запросов:

```
app.get('/', (req, res) => {  
  res.render('home', { user: req.user });  
});
```

Обсуждение

OAuth — открытый стандарт для аутентификации пользователей, который позволяет делать это с помощью сторонних приложений. Это полезно, так как позволяет пользователям легко создавать учетные записи и регистрироваться в приложениях, а также аутентифицироваться для применения данных из сторонних источников.

Запросы OAuth выполняются в такой последовательности.

1. Приложение отправляет стороннему сервису запрос на авторизацию.
2. Пользователь одобряет этот запрос.
3. Сервис перенаправляет пользователя снова в приложение, уже с кодом авторизации.

4. Приложение направляет стороннему сервису запрос с кодом авторизации.
5. Сервис выдает в ответ токен доступа (и иногда обновляет этот токен).
6. Приложение направляет сервису запрос с токеном доступа.
7. Сервис возвращает защищенный ресурс — в данном случае информацию об учетной записи пользователя.

Благодаря Passport.js и стратегии Passport.js для провайдера OAuth в Express.js этот процесс значительно упрощается. В данном примере мы построим небольшое приложение Express, которое выполняет аутентификацию в GitHub и сохраняет учетные данные пользователя между сессиями.

После того как приложение будет зарегистрировано у провайдера сервиса, можно приступать к разработке, установив соответствующие зависимости:

```
# устанавливаем общие зависимости приложения
npm install express pug dotenv
# устанавливаем зависимости passport
npm install passport passport-github
# устанавливаем зависимости для постоянной сессии пользователя
npm install connect-ensure-login express-session
```

Идентификатор и секретный код клиента OAuth мы будем хранить в файле `.env`. Эти данные можно хранить также в файле JavaScript (например, `config.js`). Этот файл ни в коем случае не должен попасть в общедоступную систему управления версиями, поэтому я советую внести его в файл `.gitignore`. Код в файле `.env` должен выглядеть так:

```
GITHUB_CLIENT_ID=<ID клиента>
GITHUB_CLIENT_SECRET=<Секретный код клиента>
SESSION_SECRET=<Секретный код сессии — это значение может быть любым, на ваш выбор>
```

Затем нужно указать в приложении Express параметры Passport.js. Для этого вносим в `index.js` следующий код:

```
const express = require('express');
const passport = require('passport');
const { Strategy } = require('passport-github');

require('dotenv').config();

const port = process.env.PORT || '3000';

// Настраиваем параметры стратегии Passport
passport.use(
  new Strategy(
    {
      clientID: process.env.GITHUB_CLIENT_ID,
      clientSecret: process.env.GITHUB_CLIENT_SECRET,
      callbackURL: `http://localhost:${port}/login/github/callback`
```



```

    },
    (accessToken, refreshToken, profile, cb) => {
      return cb(null, profile);
    }
  )
);

// Сериализуем и десериализуем данные пользователя
passport.serializeUser((user, cb) => {
  cb(null, user);
});

passport.deserializeUser((obj, cb) => {
  cb(null, obj);
});

// Создаем приложение Express
const app = express();
app.set('views', `${__dirname}/views`);
app.set('view engine', 'pug');

// Сохраняем данные сессии пользователя с помощью
// промежуточного ПО сессии Express
app.use(
  require('express-session')({
    secret: process.env.SESSION_SECRET,
    resave: true,
    saveUninitialized: true
  })
);

// Инициализируем passport и получаем из сессии состояние аутентификации
app.use(passport.initialize());
app.use(passport.session());

// Слушаем порт 3000 или порт, сохраненный в переменной среды PORT
app.listen(port, () => console.log(`Listening on port ${port}!`));

```

Теперь можно построить шаблоны представлений, которые будут иметь доступ к данным пользователя.

Файл `views/home.pug`:

```

if !user
  p Welcome! Please
    a(href='/login/github') Login with GitHub
else
  h1 Hello #{user.username}!
  p View your
    a(href='/profile') profile

```

Файл `views/login.pug`:

```

h1 Login
a(href='/login/github') Login with GitHub

```

Файл `views/profile.pug`:

```
h1 Profile
ul
  li ID: #{user.id}
  li Name: #{user.username}
  if user.emails
    li Email: #{user.emails[0].value}
```

Теперь наконец можно задать маршруты в файле `index.js`:

```
app.get('/', (req, res) => {
  res.render('home', { user: req.user });
});

app.get('/login', (req, res) => {
  res.render('login');
});

app.get('/login/github', passport.authenticate('github'));

app.get(
  '/login/github/callback',
  passport.authenticate('github', { failureRedirect: '/login' }),
  (req, res) => {
    res.redirect('/');
  }
);

app.get(
  '/profile',
  require('connect-ensure-login').ensureLoggedIn(),
  (req, res) => {
    res.render('profile', { user: req.user });
  }
);
```

Этот пример специально сделан так, чтобы в точности соответствовать примеру Express 4.x для Facebook (<https://github.com/passport/express-4.x-facebook-example>), который представляет собой хорошо документированный код для работы с Express и аутентификации в Facebook. Есть также сотни других стратегий использования Passport.js (<http://www.passportjs.org>).

21.6. Обработка форматированных данных

Задача

Вместо того чтобы размещать данные на веб-странице или передавать их в виде текста, мы хотим вернуть в браузер данные, представленные в определенном формате, например XML.

Решение

Использовать для форматирования данных один или несколько модулей Node. Например, если мы хотим вернуть код XML, то можно представить данные в этом формате с помощью следующего модуля:

```
const builder = require('xmlbuilder');

const xml = builder
  .create('resources')
  .ele('resource')
  .ele('title', 'Ecma-262 Edition 10')
  .up()
  .ele('url', 'https://www.ecma-international.org/ecma-262/10.0/index.html')
  .up()
  .end({ pretty: true });
```

Затем нужно создать соответствующий заголовок, который будет передаваться вместе с данными, и вернуть данные в браузер:

```
app.get('/', (req, res) => {
  res.setHeader('Content-Type', 'application/xml');
  res.end(xml.toString(), 'utf8');
});
```

Обсуждение

Веб-серверы часто обслуживают статические ресурсы или ресурсы, генерируемые на стороне сервера. Не менее часто браузер получает данные, представленные в каком-либо формате, — они сначала обрабатываются и лишь затем выводятся на веб-странице.

При создании и передаче форматированных данных есть два ключевых момента. Первый — задействовать любую библиотеку Node, которая упростит создание данных, а второй — гарантировать, что заголовок, передаваемый вместе с данными, будет соответствовать этим данным.

В предложенном решении для создания корректного XML-кода использован модуль `xmlbuilder`. Он не устанавливается вместе с Node по умолчанию, так что его нужно установить посредством `npm` — Node Package Manager:

```
npm install xmlbuilder
```

Затем нужно создать XML-документ, а в нем — корневой элемент и вставить в него по очереди все элементы ресурса, как показано в примере. Конечно, мы могли бы сформировать XML-строку самостоятельно, но это та еще морока. И здесь слишком легко наделать ошибок, которые потом будет трудно обнаружить. Одно из главных достоинств Node состоит в большом количестве модулей, с помощью которых можно реализовать практически все, что угодно. Это не просто избавляет нас от необходимости писать код

самостоятельно — большинство этих модулей тщательно протестированы и активно поддерживаются.

Когда форматированные данные будут готовы к отправке, нужно создать сопровождающий их заголовок. В данном примере, поскольку документ представлен в формате XML, следует присвоить параметру `Content-Type` в заголовке значение `application/xml`. Затем передаются данные, представленные в виде строки.

21.7. Построение RESTful API

Задача

С помощью Node.js построить REST API.

Решение

Воспользоваться методами Express: `app.get`, `app.post`, `app.put` и `app.delete`:

```
const express = require('express');

const app = express();
const port = process.env.PORT || 3000;

app.get('/', (req, res) => {
  return res.send('Received a GET HTTP method');
});
app.post('/', (req, res) => {
  return res.send('Received a POST HTTP method');
});
app.put('/', (req, res) => {
  return res.send('Received a PUT HTTP method');
});
app.delete('/', (req, res) => {
  return res.send('Received a DELETE HTTP method');
});
app.listen(port, () => console.log(`Listening on port ${port}!`));
```

Обсуждение

Аббревиатура REST означает Representational State Transfer — передача состояния представления. Это самая распространенная архитектура для построения API. REST позволяет взаимодействовать с удаленными источниками данных посредством HTTP, используя стандартные HTTP-методы — GET, POST, PUT и DELETE. Принимать такие запросы можно с помощью методов Express.

В следующем примере я создаю несколько маршрутов, которые играют роль конечных точек API. Каждая из них отвечает на свой HTTP-запрос:

- `/todos` — принимает запрос `get`, требующий вернуть список задач, и запрос `post` на создание новой задачи;
- `/todos/:todoId` — принимает запрос `get`, возвращающий конкретную задачу, запрос `put`, позволяющий пользователю изменить содержание задачи или присвоить ей состояние «готово», а также запрос `delete` для удаления задачи.

Теперь, определив маршруты, можно создать REST API, который будет отвечать на эти запросы:

```
const express = require('express');

const port = process.env.PORT || 3000;
const app = express();
app.use(express.json());
app.use(express.urlencoded({ extended: true }));

// Массив данных
let todos = [
  {
    id: '1',
    text: 'Order pizza',
    completed: true
  },
  {
    id: '2',
    text: 'Pick up pizza',
    completed: false
  }
];

// Получить список задач
app.get('/todos', (req, res) => {
  return res.send({ data: { todos } });
});

// Получить отдельную задачу
app.get('/todos/:todoId', (req, res) => {
  const foundTodo = todos.find(todo => todo.id === req.params.todoId);
  return res.send({ data: foundTodo });
});

// Создать задачу
app.post('/todos', (req, res) => {
  const todo = {
    id: String(todos.length + 1),
    text: req.body.text,
    completed: false
  };

  todos.push(todo);
  return res.send({ data: todo });
});
```

```
});

// Изменить задачу
app.put('/todos/:todoId', (req, res) => {
  const todoIndex = todos.findIndex(todo => todo.id ===
    req.params.todoId);
  const todo = {
    id: req.params.todoId,
    text: req.body.text || todos[todoIndex].text,
    completed: req.body.completed || todos[todoIndex].completed
  };

  todos[todoIndex] = todo;
  return res.send({ data: todo });
});

// Удалить задачу
app.delete('/todos/:todoId', (req, res) => {
  const deletedTodo = todos.find(todo => todo.id === req.params.todoId);
  todos = todos.filter(todo => todo.id !== req.params.todoId);
  return res.send({ data: deletedTodo });
});

// Прослушивать порт 3000 или порт, заданный в переменной среды PORT
app.listen(port, () => console.log(`Listening on port ${port}!`));
```

Мы можем протестировать получаемые ответы в терминале с помощью утилиты `curl`:

```
# Получить список задач
curl http://localhost:3000/todos

# Получить отдельную задачу
curl http://localhost:3000/todos/1

# Создать задачу
curl -X POST -H "Content-Type:application/json" /
  http://localhost:3000/todos -d '{"text":"Eat pizza"}'

# Изменить задачу
curl -X PUT -H "Content-Type:application/json" /
  http://localhost:3000/todos/2 -d '{"completed": true }

# Удалить задачу
curl -X DELETE http://localhost:3000/todos/3
```

Но тестировать API вручную с помощью `curl` быстро надоедает. При разработке API очень удобно применять REST-клиент с пользовательским интерфейсом, такой как *Insomnia* (<https://insomnia.rest>) или *Postman* (<https://postman.com>) (рис. 21.6).

В приведенном примере я разместил данные непосредственно в приложении. При построении API вам, скорее всего, понадобится подключение к базе данных. Для этого стоит обратиться к библиотекам, таким как Sequelize (<https://oreil.ly/NuXyR>) (для баз данных SQL) или Mongoose (<https://oreil.ly/zP8Fr>) (для MongoDB), либо разместить данные в онлайн-хранилище, таком как Firebase (<https://oreil.ly/iZSFB>).

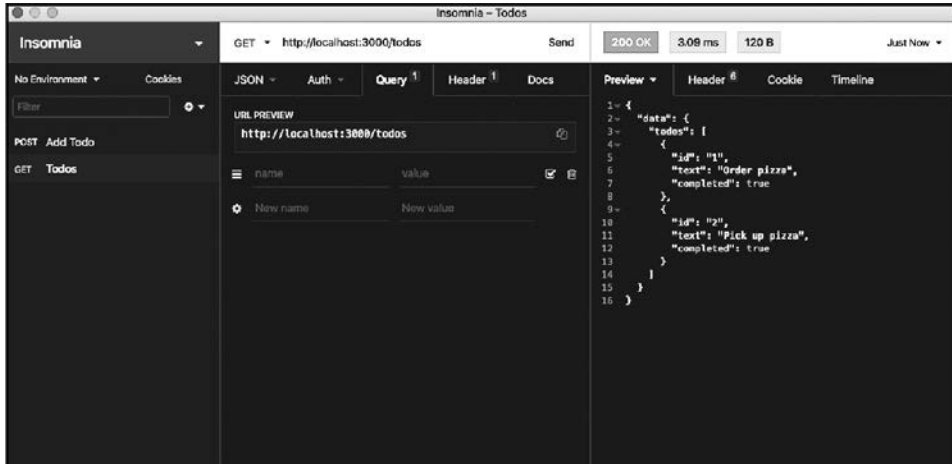


Рис. 21.6. GET-запрос из REST-клиента Insomnia

21.8. Построение API GraphQL

Задача

Построить серверное приложение с API GraphQL или добавить конечные точки GraphQL в уже существующее приложение.

Решение

Использовать пакет Apollo Server, позволяющий добавить в приложение определения типов и преобразователи GraphQL, а также GraphQL Playground:

```
const express = require('express');
const { ApolloServer, gql } = require('apollo-server-express');

const port = process.env.PORT || 3000;
const app = express();

const typeDefs = gql`
```

```
    type Query {  
      hello: String  
    }  
  `;  
  
const resolvers = {  
  Query: {  
    hello: () => 'Hello world!'  
  }  
};  
  
const server = new ApolloServer({ typeDefs, resolvers });  
server.applyMiddleware({ app, path: '/' });  
app.listen({ port }, () => console.log(`Listening on port ${port}!`));
```

Apollo Server предоставляет доступ к GraphQL Playground (рис. 21.7), что обеспечивает простой доступ к API в процессе разработки (а при необходимости и в ходе эксплуатации).

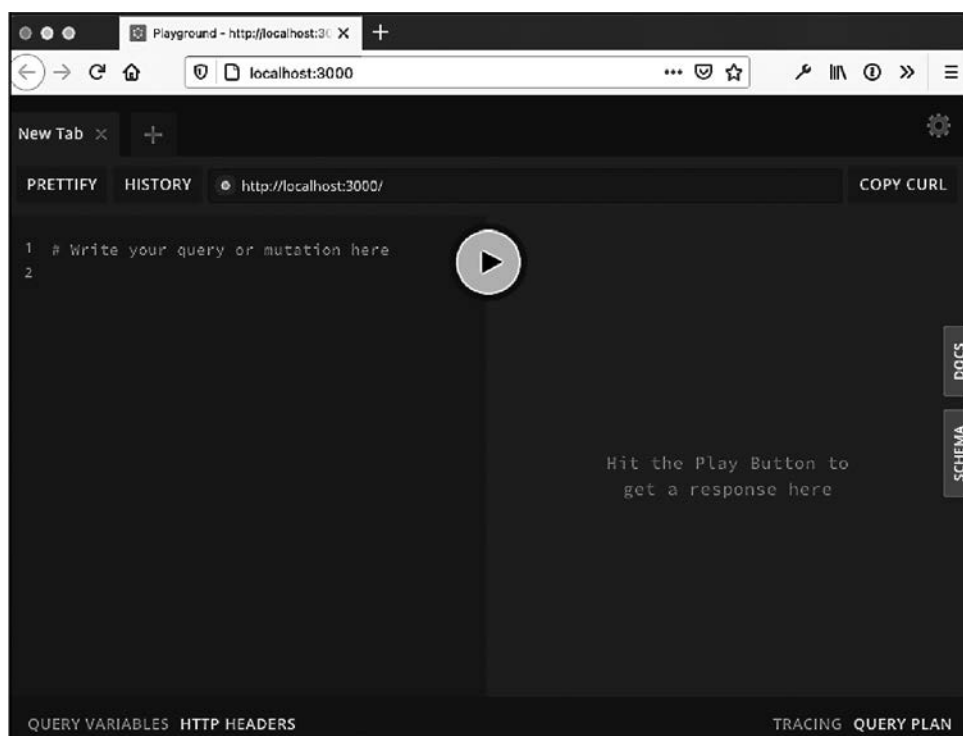


Рис. 21.7. Запрос GraphQL в GraphQL Playground

В GraphQL Playground также автоматически генерируется документация к API на основе предоставляемых разработчиком определений типов (рис. 21.8).

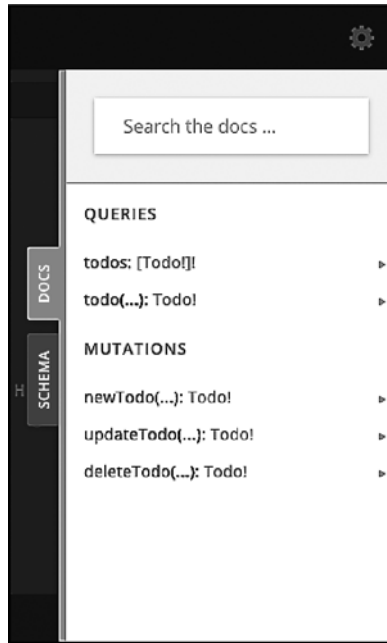


Рис. 21.8. Документация, автоматически генерируемая в GraphQL Playground

Обсуждение

GraphQL — это предназначенный для API язык запросов с открытым кодом. Его цель — предоставить единые конечные точки, позволяющие запрашивать из приложений необходимые данные. Apollo Server (<https://oreil.ly/toPLM>) можно использовать как самостоятельный пакет или как промежуточное ПО, интегрированное в одну из популярных библиотек для серверных приложений Node.js, таких как Express, Napi, Fastify и Koa.

В GraphQL реализована схема определения типов — создаваемое разработчиком представление данных и взаимодействий. Благодаря схеме GraphQL создает четкую структуру создаваемого API. В результате API может возвращать только те данные и обеспечивать только те взаимодействия, которые определены схемой. Фундаментальным компонентом схем GraphQL являются типы объектов. В GraphQL пять встроенных скалярных типов:

- **String** — строка в кодировке UTF-8;
- **Boolean** — принимает значение `true` или `false`;
- **Int** — 32-разрядное целое число;
- **Float** — число в формате с плавающей точкой;
- **ID** — уникальный идентификатор.

Написав схему, мы предоставляем API ряд преобразователей — функций, которые определяют, каким образом следует возвращать или изменять данные в соответствии с запросом.

В предыдущем примере мы использовали пакет `apollo-server-express`, который необходимо установить вместе с пакетами `express` и `graphql`:

```
$ npm install express apollo-server-express graphql
```

Для создания приложения с функциями CRUD нужно написать определения типов GraphQL и соответствующие преобразователи. Следующий код делает то же, что и пример из рецепта 21.7:

```
const express = require('express');
const { ApolloServer, graphql } = require('apollo-server-express');

const port = process.env.PORT || 3000;
const app = express();

// Массив данных
let todos = [
  {
    id: '1',
    text: 'Order pizza',
    completed: true
  },
  {
    id: '2',
    text: 'Pick up pizza',
    completed: false
  }
];

// Определения типов GraphQL
const typeDefs = graphql`
  type Query {
    todos: [Todo!]!
    todo(id: ID!): Todo!
  }

  type Mutation {
    newTodo(text: String!): Todo!
    updateTodo(id: ID!, text: String, completed: Boolean): Todo!
    deleteTodo(id: ID!): Todo!
  }

  type Todo {
    id: ID!
    text: String!
    completed: Boolean
  }
`;

// Преобразователи GraphQL
```

```
const resolvers = {
  Query: {
    todos: () => todos,
    todo: (parent, args) => {
      return todos.find(todo => todo.id === args.id);
    }
  },
  Mutation: {
    newTodo: (parent, args) => {
      const todo = {
        id: String(todos.length + 1),
        text: args.text,
        completed: false
      };
      todos.push(todo);
      return todo;
    },
    updateTodo: (parent, args) => {
      const todoIndex = todos.findIndex(todo => todo.id === args.id);
      const todo = {
        id: args.id,
        text: args.text || todos[todoIndex].text,
        completed: args.completed || todos[todoIndex].completed
      };
      todos[todoIndex] = todo;
      return todo;
    },
    deleteTodo: (parent, args) => {
      const deletedTodo = todos.find(todo => todo.id === args.id);
      todos = todos.filter(todo => todo.id !== args.id);
      return deletedTodo;
    }
  }
};

// Конфигурация сервера Apollo + Express
const server = new ApolloServer({ typeDefs, resolvers });
server.applyMiddleware({ app, path: '/' });
app.listen({ port }, () => console.log(`Listening on port ${port}!`));
```

В этом примере данные сохраняются непосредственно в приложении. При построении реального API вам, скорее всего, потребуется подключение к базе данных. Для этого можно воспользоваться одной из библиотек, такой как Sequelize (для баз данных SQL) или Mongoose (для MongoDB), либо онлайн-хранилищем данных, таким как Firebase.

Написанные нами запросы возвращают данные непосредственно через API, а функции изменения данных позволяют выполнять такие изменения, как создание записи, изменение или удаление элемента данных.

Об авторах

Адам Д. Скотт — ведущий инженер, веб-разработчик, преподаватель и художник из Коннектикута. Вот уже более десяти лет Адам работает на стыке технологий и образования, составляя учебные программы для ряда технических дисциплин. Это его седьмая книга.

Мэтью Макдоналд — технический писатель, обладатель статуса Microsoft MVP (Most Valuable Professional). Мэтью написал столько увесистых книг, что ими можно подпереть все двери в доме. На его сайте (<https://prosetech.com>) вы найдете бесплатную книгу «JavaScript для детей», а также можете ознакомиться с серией его публикаций по программированию для юных разработчиков, которая называется Young Coder.

Шелли Пауэрс работает в области веб-технологий и пишет о них уже более 12 лет. Ее последние книги, вышедшие в O'Reilly, были посвящены семантически структурированным сетям, Ajax, JavaScript и веб-графике. Шелли — заядлый фотограф и поклонница веб-разработки. Ей нравится применять результаты своих новейших экспериментов при разработке многочисленных сайтов.

Иллюстрация на обложке

Птица на обложке — малая белая цапля (*Egretta garzetta*). Эта белая птичка — самая маленькая и самая распространенная из цапель Сингапура; она очень похожа на снежную цаплю Нового Света. Гнездится возле обширных внутренних и прибрежных болотистых водоемов Европы, Азии, Африки, Тайваня и Австралии. В более теплых местах малые белые цапли живут постоянно, тогда как живущие на севере птицы мигрируют в Африку и Южную Азию.

Взрослые птицы имеют длину тела 55–65 см, размах крыльев 88–106 см и весят 350–550 граммов. Их оперение полностью белое. У цапель длинные черные ноги с желтой плюсной и тонкий черный клюв. В период размножения у взрослых особей вырастают два длинных пера на затылке, тонкие перья на спине и груди. Уздечка (пространство между клювом и глазом) неоперенная, в брачный период красноватая, в остальное время года голубовато-серая.

Малые белые цапли — бойкие охотники с самыми разнообразными приемами: они умеют терпеливо выслеживать добычу на мелководье; стоять на одной ноге, а другой размешивать грязь, чтобы напугать добычу; стоя на одной ноге, другой размахивать над поверхностью воды как приманкой. Они питаются рыбой, насекомыми, земноводными, ракообразными и рептилиями. Цапли гнездятся колониями на платформах, собранных из веток, на деревьях или кустарниках, в тростниковых зарослях или бамбуковых рощах, часто вместе с другими болотными птицами.

Многие животные, изображенные на обложках O'Reilly, находятся под угрозой исчезновения, но все они важны для мира.

Автор иллюстрации — Карен Монтгомери. Обложка выполнена на основе черно-белой гравюры из «Естественной истории» (*Natural History*) Касселла.

Адам Д. Скотт, Мэтью Макдоналд, Шелли Пауэрс
JavaScript. Рецепты для разработчиков
3-е издание

Перевела с английского Е. Сандицкая

Руководитель дивизиона	<i>Ю. Сергиенко</i>
Руководитель проекта	<i>Н. Михеева</i>
Ведущий редактор	<i>Н. Гринчик</i>
Литературный редактор	<i>Н. Рощина</i>
Художественный редактор	<i>В. Мостипан</i>
Корректоры	<i>С. Беляева, М. Молчанова</i>
Верстка	<i>Л. Егорова</i>

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 08.2023. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 20.06.23. Формат 70×100/16. Бумага офсетная. Усл. п. л. 42,570. Тираж 1000. Заказ 0000.