O'REILLY®

«Кайл распутывает клубок асинхронных возможностей JavaScript и показывает, как пользоваться ими при помощи обещаний и генераторов».

— Марк Грабански, исполнительный директор и UI-разработчик, Frontend Masters

КАЙЛ СИМПСОН

АСИНХРОННАЯ ОБРАБОТКА & ОПТИМИЗАЦИЯ

ВЫ НЕ ЗНАЕТЕ JS

Async & Performance

Kyle Simpson

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo O'REILLY®

АСИНХРОННАЯ ОБРАБОТКА & ОПТИМИЗАЦИЯ

КАЙЛ СИМПСОН

ПИТЕР®

Санкт-Петербург • Москва • Екатеринбург • Воронеж
Нижний Новгород • Ростов-на-Дону
Самара • Минск

2019

ББК 32.988.02-018
УДК 004.738.5
С37

Симпсон К.

С37 {Вы не знаете JS} Асинхронная обработка и оптимизация. — СПб.: Питер, 2019. — 352 с. — (Серия «Бестселлеры O'Reilly»).

ISBN 978-5-4461-1313-2

Каким бы опытом программирования на JavaScript вы ни обладали, скорее всего, вы не понимаете язык в полной мере. Это лаконичное, но при этом глубоко продуманное руководство посвящено новым асинхронным возможностям и средствам повышения производительности, которые позволяют создавать сложные одностраничные веб-приложения и избежать при этом «кошмара обратных вызовов».

Как и в других книгах серии «Вы не знаете JS», вы познакомитесь с нетривиальными особенностями языка, которых так боятся программисты. Только вооружившись знаниями, можно достичь истинного мастерства.

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

ББК 32.988.02-018
УДК 004.738.5

ISBN 978-1491904220 англ.

Authorized Russian translation of the English edition of You Don't Know JS: Async & Performance (ISBN 9781491904220)

ISBN 978-5-4461-1313-2

Оглавление

Предисловие

За годы работы мой руководитель достаточно стал доверять мне, чтобы поручить мне проведение собеседований. Если вы ищете кандидата, умеющего программировать на JavaScript, первые вопросы должны... вообще-то они должны выявить, не нужно ли кандидату в туалет и не хочет ли он пить, потому что комфорт — это важно. Разобравшись с физиологией, я начинаю выяснять, действительно ли кандидат знает JavaScript или он знает только jQuery.

Я ничего не имею против jQuery. jQuery позволяет сделать много полезного без реальных знаний JavaScript, и это достоинство, а не недостаток. Но если должность требует высоких познаний в области быстродействия JavaScript и сопровождения кода, вам нужен человек, который знает, как устроены библиотеки (такие, как jQuery). Он должен уметь пользоваться базовыми возможностями JavaScript на том же уровне, что и разработчики библиотек.

Чтобы оценить навыки владения базовыми средствами JavaScript, я в первую очередь поинтересуюсь, что кандидат знает о замыканиях и как он использует асинхронные средства с максимальной эффективностью — именно этой теме посвящена книга.

Для начала будут рассмотрены обратные вызовы — «хлеб с маслом» асинхронного программирования. Конечно, питаться одним хлебом с маслом не очень весело, но следующее блюдо будет полно вкусных-превкусных «обещаний» (`Promise`)!

Если вы еще не знакомы с обещаниями, или промисами, — самое время познакомиться. Обещания стали официальным способом

передачи асинхронных возвращаемых значений в JavaScript и DOM. Они будут использоваться всеми будущими асинхронными DOM API и уже используются многими существующими — так что приготовьтесь! В момент написания книги обещания поддерживались большинством основных браузеров, а скоро будут поддерживаться в IE. А после того, как вы закончите смаковать обещания, надеюсь, у вас еще хватит места для следующего блюда — генераторов (`Generator`).

Генераторы обосновались в стабильных версиях Chrome и Firefox без особой помпы и церемоний, потому что, откровенно говоря, они создают больше сложностей, чем открывают интересных возможностей. По крайней мере, я так думал, пока не увидел их в связке с обещаниями. Здесь они стали важным инструментом, обеспечивающим удобочитаемость и простоту сопровождения кода.

А на десерт... Не стану портить сюрприз, но приготовьтесь заглянуть в будущее JavaScript! В книге будут рассмотрены средства, расширяющие контроль над параллелизмом и асинхронностью.

Что ж, не буду вам мешать получать удовольствие от книги, пусть занавес поднимется. А если вы уже прочитали часть книги перед тем, как взяться за предисловие, то получите заслуженные 10 очков за асинхронность!

Джейк Арчибальд (Jake Archibald) (http://jakearchibald.com, @jafathecake), разработчик Google Chrome

Введение

С самых первых дней существования Всемирной паутины язык JavaScript стал фундаментальной технологией для управления интерактивностью контента, потребляемого пользователями. Хотя история JavaScript начиналась с мерцающих следов от указателя мыши и раздражающих всплывающих подсказок, через два десятилетия технология и возможности JavaScript выросли на несколько порядков, и лишь немногие сомневаются в его важности как ядра самой распространенной программной платформы в мире — веб-технологий.

Но как язык JavaScript постоянно подвергался серьезной критике — отчасти из-за своего происхождения, еще больше из-за своей философии проектирования. Даже само его название наводит на мысли, как однажды выразился Брендан Эйх (Brendan Eich), о «недоразвитом младшем брате», который стоит рядом со своим старшим и умным братом Java. Тем не менее такое название возникло исключительно по соображениям политики и маркетинга. Между этими двумя языками существуют колоссальные различия. У JavaScript с Java общего не больше, чем у луна-парка с Луной.

Так как JavaScript заимствует концепции и синтаксические идиомы из нескольких языков, включая процедурные корни в стиле C и менее очевидные функциональные корни в стиле Scheme/Lisp, он в высшей степени доступен для широкого спектра разработчиков — даже обладающих минимальным опытом программирова-

ния. Программа «Hello World» на JavaScript настолько проста, что язык располагает к себе и кажется удобным с самых первых минут знакомства.

Пожалуй, JavaScript — один из самых простых языков для изучения и начала работы, но из-за его странностей хорошее знание этого языка встречается намного реже, чем во многих других языках. Если для написания полноценной программы на C или C++ требуется достаточно глубокое знание языка, полномасштабное коммерческое программирование на JavaScript порой (и достаточно часто) едва затрагивает то, на что способен этот язык.

Хитроумные концепции, глубоко интегрированные в язык, проявляются в простых на первый взгляд аспектах, например, передачи функций в форме обратных вызовов. У разработчика JavaScript появляется соблазн просто использовать язык «как есть» и не беспокоиться о том, что происходит «внутри».

Это одновременно простой и удобный язык, находящий повсеместное применение, и сложный, многогранный набор языковых механик, истинный смысл которых без тщательного изучения останется непонятным даже для самого опытного разработчика JavaScript.

В этом заключается парадокс JavaScript — ахиллесова пята языка, — проблема, которой мы сейчас займемся. Так как JavaScript можно использовать без полноценного понимания, очень часто это понимание языка так и не приходит к разработчику.

Задача

Если каждый раз, сталкиваясь с каким-то сюрпризом или неприятностью в JavaScript, вы заносите их в «черный список» (как многие привыкли делать), вскоре от всей мощи JavaScript у вас останется пустая скорлупа.

Хотя это подмножество принято называть «Хорошими Частями», я призываю вас, дорогой читатель, рассматривать его как «Простые Части», «Безопасные Части» и даже «Неполные Части».

Серия «Вы не знаете JS» идет в прямо противоположном направлении: изучить и глубоко понять весь язык JavaScript и особенно «Сложные Части».

Здесь мы прямо говорим о существующей среди разработчиков JS тенденции изучать «ровно столько, сколько нужно» для работы, не заставляя себя разбираться в том, что именно происходит и почему язык работает именно так. Более того, мы воздерживаемся от распространенной тактики отступить, когда двигаться дальше становится слишком трудно.

Я не привык останавливаться в тот момент, когда что-то просто работает, а я толком сам не знаю почему, и вам не советую. Приглашаю вас в путешествие по этим неровным, непростым дорогам; здесь вы узнаете, что собой представляет язык JavaScript и что он может сделать. С этими знаниями ни один метод, ни один фреймворк, ни одно модное сокращение или термин недели не останутся за пределами вашего понимания.

В каждой книге серии мы возьмем одну из конкретных базовых частей языка, которые часто понимаются неправильно или недостаточно глубоко, и рассмотрим ее вдумчиво и подробно. После чтения у вас должна сформироваться твердая уверенность в том, что вы понимаете не только теорию, но и практические аспекты «того, что нужно знать для работы».

Скорее всего, то, что вы сейчас знаете о JavaScript, вы узнавали по частям от других людей, которые тоже недостаточно хорошо разбирались в теме. Такой JavaScript — не более чем тень настоящего языка. На самом деле вы пока JavaScript не знаете, но будете знать, если как следует ознакомитесь с этой серией книг. Читайте дальше, друзья мои. JavaScript ждет вас.

О книге

JavaScript — замечательный язык. Его легко изучать частично и намного сложнее — изучать полностью (или хотя бы в достаточной мере). Когда разработчики сталкиваются с трудностями, они обычно винят в этом язык вместо своего ограниченного понимания. В этих книгах я постараюсь исправить такое положение дел и помогу оценить по достоинству язык, который вы можете (и должны!) знать достаточно глубоко.

Многие примеры, приведенные в книге, рассчитаны на современные (и обращенные в будущее) среды движка JavaScript, такие как ES6. При запуске в более старых версиях движка (предшествующих ES6) поведение некоторых примеров может отличаться от описанного в тексте.

Типографские соглашения

В этой книге приняты следующие типографские соглашения:

Курсив

Используется для обозначения новых терминов.

`Моноширинный шрифт`

Применяется для оформления листингов программ и программных элементов внутри обычного текста, таких как имена переменных и функций, баз данных, типов данных, переменных окружения, инструкций и ключевых слов.

`Моноширинный курсив`

Обозначает текст, который должен замещаться фактическими значениями, вводимыми пользователем или определяемыми из контекста.

Так выделяются советы и предложения.

Так обозначаются советы, предложения и примечания общего характера.

Так обозначаются предупреждения и предостережения.

Использование программного кода примеров

Вспомогательные материалы (примеры кода, упражнения и т. д.) доступны для загрузки по адресу http://bit.ly/ydkjs-async-code.

Эта книга призвана оказать вам помощь в решении ваших задач. В общем случае все примеры кода из этой 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. Например, если вы разрабатываете программу и используете в ней несколько отрывков программного кода из книги, вам не нужно обращаться за разрешением. Однако в случае продажи или распространения компакт-дисков с примерами из этой книги вам необходимо получить разрешение от издательства O'Reilly. Если вы отвечаете на вопросы, цитируя данную книгу или примеры из нее, получение разрешения не требуется. Но при включении существенных объемов программного кода примеров из этой книги

в вашу документацию вам необходимо будет получить разрешение издательства.

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

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.

1 Асинхронность: сейчас и потом

Одна из важнейших, но при этом часто недопонимаемых частей программирования в таких языках, как JavaScript, — это средства выражения и управления поведением программы во времени. Конечно, я не имею в виду то, что происходит от начала цикла `for` до конца цикла `for`, что, безусловно, занимает некоторое время (микросекунды или миллисекунды). Речь о том, какая часть программы выполняется сейчас, а какая часть программы будет выполняться позднее, а между «*сейчас*» и «*потом*» существует промежуток, в котором ваша программа не выполняется активно.

Практически всем нетривиальным программам, когда-либо написанным (особенно на JS), в том или ином отношении приходится управлять этим промежутком, будь то ожидание пользовательского ввода, запрос информации из базы данных или файловой системы, передача данных по сети и ожидание ответа или периодическое выполнение задачи с фиксированным интервалом времени (скажем, анимация). Во всех этих ситуациях ваша программа должна управлять состоянием на протяжении промежутка времени.

Собственно, связь между частями программы, выполняющимися *сейчас* и *потом*, занимает центральное место в асинхронном программировании.

Безусловно, асинхронное программирование было доступно с первых дней JS. Но многие разработчики JS никогда не задумывались над тем, как именно и почему оно возникает в их программах, и не использовали другие способы его организации. Всегда существовал *достаточно хороший* вариант — обычная функция обратного вызова (callback). И сегодня многие настаивают на том, что обратных вызовов более чем достаточно.

Но с ростом масштаба и сложности JS, которые необходимы для удовлетворения вечно расширяющихся требований полноценного языка программирования, работающего в браузерах и на серверах, а также на всех промежуточных мыслимых устройствах, проблемы с управлением асинхронностью становятся непосильными. Возникает острая необходимость в решениях, которые были бы одновременно более действенными и разумными.

Хотя все эти рассуждения могут показаться довольно абстрактными, уверяю вас, что тема будет более полно и конкретно рассмотрена в книге. В нескольких ближайших главах представлены перспективные методы асинхронного программирования JavaScript.

Но сначала необходимо гораздо глубже разобраться, что такое асинхронность и как она работает в JS.

Блочное строение программы

Программу можно написать в одном файле .js, но ваша программа почти наверняка состоит из нескольких блоков, только один из которых будет выполняться в данный момент, а остальные будут выполняться *позднее*. Самой распространенной структурной единицей *блоков* является функция.

Проблема, с которой сталкивается большинство начинающих разработчиков JS, заключается в том, что «*потом*» не наступает четко и немедленно после «*сейчас*». Иначе говоря, задачи, которые

не могут быть завершены прямо сейчас, по определению будут завершаться асинхронно, а следовательно, вы не будете наблюдать блокирующее поведение, как вы могли бы интуитивно предположить или желать.

Пример:

```
// ajax(..) - произвольная функция Ajax из библиотеки
var data = ajax( "http://some.url.1" );

console.log( data );
// Ой! В общем случае `data` не содержит результатов Ajax
```

Вероятно, вы знаете, что стандартные запросы Ajax не завершаются синхронно. Это означает, что у функции `ajax(..)` еще нет значения, которое можно было бы вернуть для присваивания переменной `data`. Если бы функция `ajax(..)` могла блокироваться до возвращения ответа, то присваивание `data = ..` работало бы нормально.

Но Ajax-взаимодействия происходят не так. Вы выдаете асинхронный запрос Ajax *сейчас*, а результаты получаете только *через какое-то время*. Простейший (но определенно не единственный и даже не всегда лучший!) способ «ожидания» основан на использовании функции, обычно называемой *функцией обратного вызова* (callback function):

```
// ajax(..) - произвольная функция Ajax из библиотеки
ajax( "http://some.url.1", function myCallbackFunction(data){

    console.log( data ); // Отлично, данные `data` получены!

} );
```

Возможно, вы слышали о синхронных Ajax-запросах. Хотя такая техническая возможность существует, вы никогда, ни при каких условиях не должны пользоваться ею, потому что она полностью блокирует пользовательский интерфейс браузера (кнопки, меню, прокрутку и т. д.) и перекрывает любые взаимодействия с пользователем. Это очень плохая мысль, избегайте ее в любых случаях.

Прежде чем вы начнете протестовать — нет, ваше желание избежать путаницы с обратными вызовами не оправдывает блокирующие синхронные операции Ajax. Возьмем для примера следующий код:

```
function now() {
    return 21;
}

function later() {
    answer = answer * 2;
    console.log( "Meaning of life:", answer );
}

var answer = now();

setTimeout( later, 1000 ); // Meaning of life: 42
```

Программа состоит из двух блоков: того, что будет выполняться *сейчас*, и того, что будет выполняться *потом*. Наверное, вы и сами понимаете, что это за блоки, но чтобы не оставалось ни малейших сомнений:

Сейчас:

```
function now() {
    return 21;
}
function later() { .. }

var answer = now();

setTimeout( later, 1000 );
```

Потом:

```
answer = answer * 2;
console.log( "Meaning of life:", answer );
```

Блок «*сейчас*» выполняется немедленно, как только вы запускаете свою программу. Но `setTimeout(..)` также настраивает событие (тайм-аут), которое должно произойти в будущем, так что содер-

жимое функции `later()` будет выполнено позднее (через 1000 миллисекунд).

Каждый раз, когда вы упаковываете фрагмент кода в функцию и указываете, что она должна быть выполнена по некоторому событию (таймер, щелчок мышью, ответ Ajax и т. д.), вы создаете в своем коде блок «*потом*», а следовательно, вводите асинхронность в свою программу.

Асинхронный вывод в консоль

Не существует никаких спецификаций или наборов требований, определяющих, как работают методы `console.*`, — официально они не являются частью JavaScript, а добавляются в JS управляющей средой.

А значит, разные браузеры и среды JS действуют так, как считают нужным, что иногда приводит к неожиданному поведению.

В частности, для некоторых браузеров и некоторых условий `console.log(..)` не выводит полученные данные немедленно. Прежде всего, это может произойти из-за того, что ввод/вывод — очень медленная и блокирующая часть многих программ (не только JS). Для браузера может быть более производительно (с точки зрения страниц/пользовательского интерфейса) выполнять консольный ввод/вывод в фоновом режиме, и вы, возможно, даже не будете подозревать о нем.

Не слишком распространенная, но возможная ситуация, в которой можно понаблюдать за этим явлением (не из самого кода, а снаружи):

```
var a = {
    index: 1
};

// потом
console.log( a ); // ??

// еще позднее
a.index++;
```

Обычно мы ожидаем, что объект `a` будет зафиксирован в точный момент выполнения команды `console.log(..)` и будет выведен результат `{ index: 1 }`, чтобы в следующей команде при выполнении `a.index++` изменялось нечто иное, чем выводимое значение `a`.

В большинстве случаев приведенный выше код, скорее всего, выдаст в консоль инструментария разработчика представление объекта, чего вы и ожидаете. Но может оказаться, что тот же код будет выполняться в ситуации, в которой браузер сочтет нужным перевести консольный ввод/вывод в фоновый режим. И тогда может оказаться, что к тому времени, когда представление объекта будет выводиться в консоль браузера, увеличение `a.index++` уже произошло, и будет выведен результат `{ index: 2 }`.

На вопрос о том, при каких именно условиях консольный ввод/вывод будет отложен и будет ли вообще наблюдаться данный эффект, невозможно дать четкий ответ. Просто учитывайте возможную асинхронность при вводе/выводе в том случае, если у вас когда-нибудь возникнут проблемы с отладкой, когда объекты были изменены уже *после* команды `console.log(..)`, но эти изменения совершенно неожиданно проявляются при выводе.

Если вы столкнетесь с этой нечастой ситуацией, лучше всего использовать точки прерывания в отладчике JS, вместо того чтобы полагаться на консольный вывод. Следующий вариант — принудительно «зафиксировать» представление интересующего вас объекта, преобразовав его в строку (например, `JSON.stringify(..)`).

Цикл событий

Начну с утверждения (возможно, даже удивительного): несмотря на то что вы очевидным образом имели возможность писать асинхронный код JS (как в примере с тайм-аутом, рассмотренным выше), до последнего времени (ES6) в самом языке JavaScript никогда не было никакого встроенного понятия асинхронности.

Что?! Но это же полный бред? На самом деле это чистая правда. Сам движок JS никогда не делало ничего, кроме выполнения одного блока программы в любой конкретный момент времени, когда ему это приказывали. «Ему приказывали»... Кто приказывал? Это очень важный момент!

Движок JS не работает в изоляции. Он работает внутри управляющей среды, которой для большинства разработчиков становится обычный веб-браузер. За последние несколько лет (впрочем, не только за этот период) язык JS вышел за границы браузеров в другие среды — например, на серверы — благодаря таким технологиям, как Node.js. Более того, JavaScript сейчас встраивается в самые разные устройства, от роботов до лампочек.

Но у всех этих сред есть одна характерная особенность: у них существует механизм, который обеспечивает выполнение нескольких фрагментов вашей программы, обращаясь с вызовами к движку JS *в разные моменты времени*. Этот механизм называется *циклом событий*.

Иначе говоря, движок JS сам по себе не обладает внутренним чувством времени, но он становится средой исполнения для любого произвольного фрагмента JS. *Планирование* «событий» (выполнений фрагментов кода JS) всегда осуществляется окружающей средой.

Итак, например, когда ваша программа JS выдает запрос Ajax для получения данных с сервера, вы определяете код реакции в функции (обычно называемой функцией обратного вызова, или просто обратным вызовом), а движок JS говорит управляющей среде: «Так, я собираюсь ненадолго приостановить выполнение, но когда ты завершишь обработку этого сетевого запроса и получишь данные, пожалуйста, вызови *вот эту* функцию».

Браузер настраивается для прослушивания ответа от сети, и когда у него появятся данные, чтобы передать их программе, планирует выполнение функции обратного вызова, вставляя ее в цикл событий. Так что же такое «цикл событий»?

Следующий фиктивный код поможет представить суть цикла событий на концептуальном уровне:

```
// `eventLoop` - массив, работающий по принципу очереди
// (первым пришел, первым вышел)
var eventLoop = [ ];
var event;

// продолжать "бесконечно"
while (true) {
    // отработать "квант"
    if (eventLoop.length > 0) {
        // получить следующее событие в очереди
        event = eventLoop.shift();

        // выполнить следующее событие
        try {
            event();
        }
        catch (err) {
            reportError(err);
        }
    }
}
```

Конечно, этот сильно упрощенный псевдокод только демонстрирует основные концепции. Тем не менее и его должно быть достаточно для понимания сути.

Как видите, имеется непрерывно работающий цикл, представленный циклом `while`; каждая итерация цикла называется *тиком*. В каждом тике, если в очереди ожидает событие, оно, событие, извлекается и выполняется. Этими событиями становятся ваши функции обратного вызова.

Важно заметить, что функция `setTimeout(..)` не ставит обратный вызов в очередь цикла событий. Вместо этого она запускает таймер; по истечении таймера среда помещает ваш обратный вызов в цикл событий, чтобы некий тик в будущем подобрал его для выполнения.

А если в очереди в данный момент уже находятся 20 элементов? Вашему обратному вызову придется подождать. Он ставится в очередь после всех остальных — обычно не существует способа выйти вперед и обогнать других в очереди. Это объясняет, почему таймеры `setTimeout(..)` не гарантируют идеальной точности. Функция гарантирует (упрощенно говоря), что обратный вызов не сработает *до* истечения заданного вами интервала, но это может произойти в заданный момент или после него в зависимости от состояния очереди событий.

Другими словами, ваша программа обычно разбивается на множество мелких блоков, которые выполняются друг за другом в очереди цикла событий. И с технической точки зрения в эту очередь также могут попасть другие события, не имеющие прямого отношения к вашей программе.

Мы упомянули «до недавнего времени», говоря о том, как спецификация ES6 изменила природу управления очередью цикла событий. В основном это формальная техническая деталь, но ES6 теперь точно указывает, как работает цикл событий; это означает, что формально он попадает в сферу действия движка JS, а не только управляющей среды. Одна из главных причин для такого изменения — появление в ES6 обещаний (см. главу 3), для которых необходим прямой, точный контроль за операциями планирования очереди цикла событий (вызов `setTimeout(..0)` рассматривается в разделе «Кооперация»).

Параллельные потоки

Термины «асинхронный» и «параллельный» очень часто используются как синонимы, но в действительности они имеют разный смысл. Напомню, что суть асинхронности — управление промежутком между «сейчас» и «потом». Параллелизм обозначает возможность одновременного выполнения операций.

Самые распространенные средства организации параллельных вычислений — *процессы* и *потоки* (threads). Процессы и потоки

выполняются независимо и могут выполняться одновременно на разных процессорах и даже на разных компьютерах, но несколько потоков могут совместно использовать общую память одного процесса.

С другой стороны, цикл событий разбивает свою работу на задачи и выполняет их последовательно, что делает невозможным параллельный доступ и изменения в общей памяти. Параллелизм и последовательность не могут совместно существовать в форме взаимодействующих циклов событий в разных потоках.

Чередование параллельных потоков выполнения и чередование асинхронных событий происходит на совершенно разных уровнях детализации.

Пример:

```
function later() {
    answer = answer * 2;
    console.log( "Meaning of life:", answer );
}
```

Хотя все содержимое `later()` будет рассматриваться как один элемент очереди цикла событий, в потоке, в котором будет выполняться этот код, произойдет более десятка низкоуровневых операций. Например, для команды `answer = answer * 2` нужно будет сначала загрузить текущее значение `answer`, потом поместить куда-то `2`, затем выполнить умножение, затем взять результат и снова сохранить его в `answer`.

В однопоточной среде неважно, что элементы очереди потока являются низкоуровневыми операциями, потому что ничто не может прервать поток. Но в параллельной системе, в которой в одной программе могут выполняться два разных потока, легко могут возникнуть непредсказуемые последствия.

Пример:

```
var a = 20;

function foo() {
```

```
    a = a + 1;
}

function bar() {
    a = a * 2;
}

// ajax(..) - произвольная функция Ajax из библиотеки
ajax( "http://some.url.1", foo );
ajax( "http://some.url.2", bar );
```

При однопоточном поведении в JavaScript, если `foo()` выполняется перед `bar()`, в результате `a` будет содержать `42`, а если `bar()` выполняется перед `foo()`, то результат будет равен `41`.

Но если события JS, совместно использующие одни данные, выполняются параллельно, проблемы становятся намного более коварными. Возьмем два списка задач из псевдокода как потоки, которые могут выполнять код `foo()` и `bar()` соответственно, и посмотрим, что произойдет, если они выполняются ровно в одно и то же время.

Поток 1 (X и Y — временные области памяти):

```
foo():
  a. загрузить значение `a` в `X`
  b. сохранить `1` в `Y`
  c. сложить `X` и `Y`, сохранить результат в `X`
  d. сохранить значение `X` в `a`
```

Поток 2 (X и Y — временные области памяти):

```
bar():
  a. загрузить значение `a` в `X`
  b. сохранить `2` в `Y`
  c. перемножить `X` и `Y`, сохранить результат в `X`
  d. сохранить значение `X` в `a`
```

Теперь предположим, что два потока выполняются действительно параллельно. Вы уже заметили проблему, верно? Они ис-

пользуют области общей памяти X и Y для своих промежуточных операций.

Какой результат будет сохранен в a, если операции будут выполняться в следующем порядке?

```
1a  (загрузить значение `a` в `X`   ==> `20`)
2a  (загрузить значение `a` в `X`   ==> `20`)
1b  (сохранить `1` в `Y`   ==> `1`)
2b  (сохранить `2` в `Y`   ==> `2`)
1c  (сложить `X` и `Y`, сохранить результат в `X`   ==> `22`)
1d  (сохранить значение `X` в `a`   ==> `22`)
2c  (перемножить `X` и `Y`, сохранить результат в `X`   ==> `44`)
2d  (сохранить значение `X` в `a`   ==> `44`)
```

Значение a будет равно 44. А как насчет такого порядка?

```
1a  (загрузить значение `a` в `X`   ==> `20`)
2a  (загрузить значение `a` в `X`   ==> `20`)
2b  (сохранить `2` в `Y`   ==> `2`)
1b  (сохранить `1` в `Y`   ==> `1`)
2c  (перемножить `X` и `Y`, сохранить результат в `X`   ==> `20`)
1c  (сложить `X` и `Y`, сохранить результат в `X`   ==> `21`)
1d  (сохранить значение `X` в `a`   ==> `21`)
2d  (сохранить значение `X` в `a`   ==> `21`)
```

На этот раз значение a будет равно 21.

Итак, многопоточное программирование может быть очень сложным, потому что если вы не предпримете специальных мер для предотвращения подобных прерываний/чередований, возможно очень странное недетерминированное поведение, которое часто создает проблемы.

В JavaScript совместное использование данных разными потоками невозможно, что означает, что этот уровень недетерминизма не создаст проблем. Но это не означает, что поведение JS всегда детерминировано. Вспомните предыдущий пример, в котором относительный порядок foo() и bar() мог приводить к двум разным результатам (41 или 42).

Возможно, это еще неочевидно, но недетерминизм не всегда плох. Иногда он обходится без последствий, иногда вводится намеренно. В этой и следующих главах вам встретятся другие примеры.

Выполнение до завершения

Из-за однопоточного характера JavaScript код внутри функций `foo()` (и `bar()`) выполняется атомарно; это означает, что после того, как функция `foo()` начнет выполняться, весь ее код будет завершен до того, как будет выполнен какой-либо код `bar()`, и наоборот. Это поведение называется *выполнением до завершения* (run-to-completion). Собственно, семантика выполнения до завершения становится более очевидной, когда `foo()` и `bar()` содержат больше кода:

```
var a = 1;
var b = 2;

function foo() {
    a++;
    b = b * a;
    a = b + 3;
}

function bar() {
    b--;
    a = 8 + b;
    b = a * 2;
}

// ajax(..) - произвольная функция Ajax из библиотеки
ajax( "http://some.url.1", foo );
ajax( "http://some.url.2", bar );
```

Поскольку `foo()` не может прерываться `bar()`, а `bar()` не может прерываться `foo()`, у этой программы есть только два возможных результата в зависимости от того, какая из функций запустится первой. Если присутствует многопоточное выполнение, а отдель-

ные команды в `foo()` и `bar()` чередуются, количество возможных результатов значительно возрастает!

Блок 1 является синхронным (выполняется *сейчас*), но блоки 2 и 3 асинхронны (выполняются *потом*); это означает, что их выполнение будет разделено временным интервалом.

Блок 1:

```
var a = 1;
var b = 2;
```

Блок 2 (`foo()`):

```
a++;
b = b * a;
a = b + 3;
```

Блок 3 (`bar()`):

```
b--;
a = 8 + b;
b = a * 2;
```

При выполнении блоков 2 и 3 первым может оказаться любой из этих блоков, поэтому у программы есть два возможных результата:

Результат 1:

```
var a = 1;
var b = 2;

// foo()
a++;
b = b * a;
a = b + 3;

// bar()
b--;
a = 8 + b;
b = a * 2;
```

```
a; // 11
b; // 22
```

Результат 2:

```
var a = 1;
var b = 2;

// bar()
b--;
a = 8 + b;
b = a * 2;

// foo()
a++;
b = b * a;
a = b + 3;

a; // 183
b; // 180
```

Два результата при выполнении одного кода? Недетерминизм! Но он проявляется на уровне упорядочения функций (событий), а не на уровне упорядочения команд (или даже на уровне упорядочения выражений), как в случае потоков. Иначе говоря, выполнение более детерминировано, чем в случае с потоками.

Применительно к поведению JavaScript этот недетерминизм упорядочения функций относится к стандартному *состоянию гонки* (функции `foo()` и `bar()` словно соревнуются друг с другом за право запуститься первой). А конкретно состояние гонки проявляется в том, что вы не можете заранее надежно предсказать значения `a` и `b`.

Если бы в JS существовала функция, не обладающая поведением выполнения до завершения, то возможных результатов было бы намного больше, не так ли? Оказывается, в ES6 такая функция появилась (см. главу 4), но пока не беспокойтесь — мы еще вернемся к этой теме!

Параллельное выполнение

Представьте сайт со списком обновлений (например, поставки новостей социальной сети), которые последовательно загружаются при прокрутке списка пользователем. Чтобы такая функциональность работала правильно, по крайней мере два разных «процесса» должны выполняться *одновременно* (то есть в пределах одного временного окна, но не обязательно в один момент времени).

Термин «процесс» заключен в кавычки, потому что речь идет не о настоящих процессах уровня операционной системы в понимании компьютерных наук. Это виртуальные процессы (или задачи), представляющие логически связанные, последовательные серии операций. Я использую термин «процесс» вместо «задача», потому что с точки зрения терминологии он подходит под определения тех концепций, которые рассматриваются в этой главе.

Первый «процесс» реагирует на события `onscroll` (выдавая запросы Ajax на получение нового контента), срабатывающие при прокрутке страницы. Второй «процесс» получает ответы Ajax (для рендера контента на странице).

Очевидно, если пользователь прокручивает страницу достаточно быстро, за время, необходимое для получения и обработки первого ответа, могут сработать два и более события `onscroll`. Таким образом, события `onscroll` и события ответов Ajax будут срабатывать достаточно часто и чередоваться друг с другом.

Параллельное выполнение происходит тогда, когда два и более «процесса» выполняются одновременно в течение одного периода времени независимо от того, выполняются ли параллельно составляющие их отдельные операции (в один момент времени на разных процессорах или ядрах). Таким образом, речь идет о параллелизме уровня «процессов» (или уровня задач), в отличие от параллелизма уровня операций (потоков на разных процессорах).

Параллельное выполнение также открывает дополнительный вопрос о взаимодействии этих «процессов» друг с другом. Мы вернемся к этой теме позднее.

Давайте представим каждый независимый «процесс» как серию событий/операций в заданном временном окне (прокрутки страницы пользователем в течение нескольких секунд):

«Процесс» 1 (события `onscroll`):

```
onscroll, запрос 1
onscroll, запрос 2
onscroll, запрос 3
onscroll, запрос 4
onscroll, запрос 5
onscroll, запрос 6
onscroll, запрос 7
```

«Процесс» 2 (события ответов Ajax):

```
ответ 1
ответ 2
ответ 3
ответ 4
ответ 5
ответ 6
ответ 7
```

Вполне возможно, что событие `onscroll` и событие ответа Ajax окажутся готовыми к обработке точно *в один момент*. Возьмем наглядное представление этих событий на временной шкале:

```
onscroll, запрос 1
onscroll, запрос 2          ответ 1
onscroll, запрос 3          ответ 2
ответ 3
onscroll, запрос 4
onscroll, запрос 5
onscroll, запрос 6          ответ 4
```

```
onscroll, запрос 7
ответ 6
ответ 5
ответ 7
```

Вспомните концепцию цикла событий, представленную ранее в этой главе: JS может обрабатывать только одно событие за раз, так что первым произойдет либо `onscroll, запрос 2`, либо `ответ 1`, но они не могут произойти буквально в один момент. Представьте детей в школьной столовой — как бы они ни толпились у дверей, им придется выстроиться в очередь по одному, чтобы получить свой обед!

В очереди цикла событий чередование событий будет выглядеть примерно так:

```
onscroll, запрос 1   <--- Процесс 1 запустился
onscroll, запрос 2
ответ 1              <--- Процесс 2 запустился
onscroll, запрос 3
ответ 2
ответ 3
onscroll, запрос 4
onscroll, запрос 5
onscroll, запрос 6
ответ 4
onscroll, запрос 7   <--- Процесс 1 завершился
ответ 6
ответ 5
ответ 7              <--- Процесс 2 завершился
```

«Процесс» 1 и «процесс» 2 выполняются параллельно (на уровне задач), но их отдельные события в очереди цикла задач отрабатываются последовательно. А вы заметили, что ответ 6 и ответ 5 выбиваются из ожидаемого порядка?

Однопоточный цикл событий — одно из выражений концепции параллельного выполнения (конечно, существуют и другие; мы вернемся к этой теме позднее).

Отсутствие взаимодействий

При параллельном чередовании операций/событий двух или более «процессов» в одной программе они могут и не взаимодействовать друг с другом, если задачи не связаны. *Если они не взаимодействуют, то недетерминизм вполне приемлем.*

Пример:

```
var res = {};

function foo(results) {
    res.foo = results;
}

function bar(results) {
    res.bar = results;
}

// ajax(..) - произвольная функция Ajax из библиотеки
ajax( "http://some.url.1", foo );
ajax( "http://some.url.2", bar );
```

`foo()` и `bar()` — два параллельных «процесса», и порядок их запуска не детерминирован. Но программа спроектирована так, что порядок их запуска неважен, потому что они работают независимо, а следовательно, им незачем взаимодействовать друг с другом.

Ошибки ситуации гонки не возникает, потому что код всегда будет работать правильно независимо от порядка запуска.

Взаимодействия

Чаще параллельные «процессы» все же взаимодействуют вследствие необходимости, косвенно через область видимости и/или DOM. Если же такие взаимодействия возникают, необходимо координировать их для предотвращения состояния гонки, как упоминалось выше.

Простой пример двух параллельных «процессов», взаимодействующих из-за неявного порядка активизации, который нарушается *в отдельных случаях*:

```
var res = [];

function response(data) {
    res.push( data );
}

// ajax(..) - произвольная функция Ajax из библиотеки
ajax( "http://some.url.1", response );
ajax( "http://some.url.2", response );
```

Параллельные «процессы» — два вызова `response()`, которые будут выданы для обработки ответов Ajax. Они могут происходить в любом порядке.

Предположим, что `res[0]` содержит результаты вызова `"http://some.url.1"`, а `res[1]` — результаты вызова `"http://some.url.2"`. Иногда это так, но иногда они меняются местами в зависимости от того, какой вызов завершится первым. Существует достаточно высокая вероятность того, что недетерминизм создаст ошибку из-за состояния гонки.

Будьте исключительно осторожны с предположениями, которые могут делаться в подобных ситуациях. Например, разработчик вполне может заметить, что `"http://some.url.2"` всегда отвечает намного медленнее, чем `"http://some.url.1"`, — возможно, из-за выполняемых задач (например, одна обращается к базе данных, а другая просто читает статический файл), так что наблюдаемый порядок на первый взгляд всегда совпадает с ожидаемым. Даже если оба запроса поступают на один сервер и сервер намеренно выдает ответы в определенном порядке, нет никаких реальных гарантий того, что ответы поступят браузеру в том же порядке.

Итак, для решения проблемы с состоянием гонки можно координировать порядок взаимодействий:

```
var res = [];

function response(data) {
    if (data.url == "http://some.url.1") {
        res[0] = data;
    }
    else if (data.url == "http://some.url.2") {
        res[1] = data;
    }
}

// ajax(..) - произвольная функция Ajax из библиотеки
ajax( "http://some.url.1", response );
ajax( "http://some.url.2", response );
```

Независимо от того, какой ответ Ajax вернется первым, мы анализируем значение `data.url` (если, конечно, он был возвращен сервером!) для определения того, какую позицию данные ответа должны занимать в массиве `res`. `res[0]` всегда содержит результаты `"http://some.url.1"`, а `res[1]` всегда содержит результаты `"http://some.url.2"`. Простая координация устраняет недетерминизм состояния гонки.

Рассуждения в этой ситуации применимы в том случае, если несколько параллельных вызовов функций взаимодействуют друг с другом через общую модель DOM — например, один обновляет содержимое `<div>`, а другой обновляет стиль или атрибуты `<div>` (например, чтобы элемент DOM становился видимым при появлении в нем контента). Вероятно, элемент DOM не должен быть видимым, пока в нем появится контент, так что координация должна гарантировать правильный порядок взаимодействий.

Некоторые сценарии параллелизма всегда (а не только иногда!) ломаются без координации взаимодействий. Пример:

```
var a, b;

function foo(x) {
    a = x * 2;
    baz();
```

```
}

function bar(y) {
    b = y * 2;
    baz();
}

function baz() {
    console.log(a + b);
}

// ajax(..) - произвольная функция Ajax из библиотеки
ajax( "http://some.url.1", foo );
ajax( "http://some.url.2", bar );
```

В этом примере независимо от того, сработает ли первой функция `foo()` или `bar()`, это всегда приведет к слишком раннему срабатыванию `baz()` (поскольку `a` или `b` все еще будет содержать `undefined`), но второй вызов `baz()` сработает, поскольку оба значения `a` и `b` будут доступны.

Существуют разные способы разрешения таких ситуаций. Один простой вариант выглядит так:

```
var a, b;

function foo(x) {
    a = x * 2;
    if (a && b) {
        baz();
    }
}

function bar(y) {
    b = y * 2;
    if (a && b) {
        baz();
    }
}

function baz() {
    console.log( a + b );
}
```

```
// ajax(..) - произвольная функция Ajax из библиотеки
ajax( "http://some.url.1", foo );
ajax( "http://some.url.2", bar );
```

Условная конструкция `if (a && b)`, в которую заключен вызов `baz()`, традиционно называется *шлюзом*: мы не уверены, в каком порядке поступят значения `a` и `b`, но ожидаем появления обоих значений, прежде чем открывать шлюз (вызывать `baz()`).

Другая ситуация параллельных взаимодействий, которая тоже встречается на практике, иногда называется *гонкой* (race), хотя правильнее было бы назвать ее *защелкой* (latch). Для нее характерно поведение «побеждает только тот, кто пришел первым». Здесь недетерминизм приемлем; вы явно указываете, что «гонка» к финишной прямой и единственный победитель — нормальное явление.

Рассмотрим следующий (неработающий) код:

```
var a;

function foo(x) {
    a = x * 2;
    baz();
}

function bar(x) {
    a = x / 2;
    baz();
}

function baz() {
    console.log( a );
}

// ajax(..) - произвольная функция Ajax из библиотеки
ajax( "http://some.url.1", foo );
ajax( "http://some.url.2", bar );
```

Какой бы из вызовов (`foo()` или `bar()`) ни сработал последним, он заменит значение `a`, присвоенное другим вызовом. Но при этом

он также продублирует вызов `baz()` (что, скорее всего, нежелательно).

Таким образом, взаимодействие можно координировать простой защелкой, чтобы проходил только первый вызов:

```
var a;

function foo(x) {
    if (!a) {
        a = x * 2;
        baz();
    }
}

function bar(x) {
    if (!a) {
        a = x / 2;
        baz();
    }
}

function baz() {
    console.log( a );
}

// ajax(..) - произвольная функция Ajax из библиотеки
ajax( "http://some.url.1", foo );
ajax( "http://some.url.2", bar );
```

Условие `if (!a)` пропускает только первый из вызовов `foo()` или `bar()`, а второй вызов (а на самом деле и все последующие) будет проигнорирован. За второе место награды не полагается!

Во всех перечисленных сценариях мы используем глобальные переменные просто для демонстрации, но в обосновании нет ничего, что бы требовало их применения. Пока функции, о которых идет речь, могут обратиться к переменным (через область видимости), они будут работать так, как положено. Единственный очевидный недостаток этой формы координации параллельного выполнения — зависимость от переменных

с лексической областью видимости (см. книгу «Область видимости и замыкания» этой серии) или глобальных переменных, как в этих примерах. В нескольких ближайших главах будут представлены другие способы координации, намного более «чистые» в этом отношении.

Кооперация

Другое выражение координации параллельного выполнения называется *кооперативным параллельным выполнением*. В нем основное внимание обращено не на организацию взаимодействий через совместный доступ к значениям в областях видимости (хотя, разумеется, такая возможность остается). Основная цель — взять продолжительный «процесс» и разбить его на фазы, или пакеты, чтобы у других параллельных «процессов» была возможность вставить свои операции в очередь цикла событий.

В качестве примера возьмем обработчик ответов Ajax, который должен обработать длинный список результатов для преобразования значений. Мы используем `Array#map(..)`, чтобы код был короче:

```
var res = [];

// `response(..)` получает массив результатов от вызова Ajax
function response(data) {
    // добавить в существующий массив `res`
    res = res.concat(
        // создать новый преобразованный массив
        // с удвоением всех значений `data`
        data.map( function(val){
            return val * 2;
        } )
    );
}

// ajax(..) - произвольная функция Ajax из библиотеки
ajax( "http://some.url.1", response );
ajax( "http://some.url.2", response );
```

Если "http://some.url.1" получит свои результаты первым, то весь список будет отображен на res одновременно. Если размер списка не превышает нескольких тысяч записей, обычно это не создает проблем. Но для списка, допустим, с 10 миллионами записей выполнение может занять некоторое время (несколько секунд на мощном ноутбуке, намного больше на мобильном устройстве и т. д.).

Пока такие «процессы» выполняются, на странице не может происходить ничего другого: ни других вызовов response(..), ни обновлений пользовательского интерфейса, ни даже пользовательских событий вроде прокрутки, ввода с клавиатуры, нажатий кнопок и т. д. Все это довольно неприятно.

Чтобы создать параллельную систему с более высоким уровнем кооперации, которая ведет себя не столь эгоистично и не забивает очередь цикла событий, вы можете обрабатывать эти результаты асинхронными пакетами. После каждого пакета управление будет возвращаться циклу событий, чтобы могли произойти другие ожидающие события.

Очень простое решение:

```
var res = [];

// `response(..)` получает массив результатов от вызова Ajax
function response(data) {
    // Данные будут обрабатываться по 1000 записей
    var chunk = data.splice( 0, 1000 );

    // добавление в существующий массив `res`
    res = res.concat(
        // создание нового преобразованного массива
        // с удвоением всех значений `chunk`
        chunk.map( function(val){
            return val * 2;
        } )
    );

    // остались данные для обработки?
    if (data.length > 0) {
        // асинхронное планирование следующего пакета
```

```
        setTimeout( function(){
            response( data );
        }, 0 );
    }
}

// ajax(..) - произвольная функция Ajax из библиотеки
ajax( "http://some.url.1", response );
ajax( "http://some.url.2", response );
```

Набор данных обрабатывается блоками с максимальным размером 1000 элементов. Таким образом создается «процесс» с коротким временем выполнения (пусть даже это означает много последующих «процессов»), так как чередование событий в очереди цикла событий существенно улучшит скорость отклика (быстродействие) сайта/приложения.

Конечно, порядок всех этих «процессов» не координируется на уровне взаимодействий, так что порядок результатов в `res` будет непредсказуемым. Если вам потребуется упорядочить данные, необходимо использовать приемы взаимодействия вроде тех, что были описаны выше, — или те, которые будут рассмотрены в последующих главах книги.

Для асинхронного планирования используется вызов `setTimeout(..0)` (`hack`), который, по сути, означает «поместить эту функцию в конец текущей очереди цикла событий».

`setTimeout(..0)` формально не вставляет элемент прямо в очередь цикла событий. Таймер вставит событие при следующей возможности. Например, для двух последовательных вызовов `setTimeout(..0)` обработка строго в порядке вызова не гарантируется, поэтому в некоторых ситуациях (например, при замедлении таймера) порядок таких событий непредсказуем. В Node.js существует аналогичное решение `process.nextTick(..)`. Каким бы удобным (а обычно и более производительным) оно ни было, не существует (по крайней мере, пока) прямолинейного механизма, обеспечивающего упорядочение асинхронных событий для всех сред. Эта тема гораздо более подробно рассматривается в следующем разделе.

Задания

В ES6 появилась новая концепция, созданная на базе очереди цикла событий — так называемая *очередь заданий* (job queue). Скорее всего, вы столкнетесь с ней при использовании асинхронного поведения обещаний (см. главу 3).

К сожалению, на данный момент этот механизм не имеет официального API, что несколько усложняет его демонстрацию. Поэтому мы опишем его концептуально, чтобы при рассмотрении асинхронного поведения с обещаниями в главе 3 вы понимали, как происходит планирование и обработка этих действий.

Пожалуй, самое лучшее объяснение из всех, которые мне удалось найти: что очередь заданий присоединяется к концу каждого тика в очереди цикла событий. Некоторые операции, подразумевающие асинхронный характер выполнения, которые могут происходить во время тика цикла событий, не приводят к добавлению в очередь цикла событий нового события, вместо этого в конец очереди заданий текущего тика добавляется новый элемент (то есть задание).

По сути, это означает примерно следующее: «Да, и вот еще одна штука, которую нужно выполнить *потом*, — но только это должно произойти до того, как произойдет *что-нибудь еще*».

Очередь цикла событий напоминает аттракцион в парке развлечений: покатавшись на нем, вы должны снова встать в конец очереди. Очередь заданий выглядит так, словно после аттракциона вы обходите очередь и сразу проходите на аттракцион еще раз.

Задание может привести к добавлению новых заданий в конец той же очереди. Теоретически возможно, что цикл заданий (задание, которое добавляет новое задание и т. д.) будет работать бесконечно долго, не позволяя вашей программе перейти к следующему тику цикла событий. На концептуальном уровне это будет почти эквивалентно включению очень долгого или бесконечного цикла (например, `while (true) ..`) в ваш код.

Задания отдаленно напоминают трюк с `setTimeout(..0)`, но реализуются так, чтобы обеспечивать гораздо более четко определенное и гарантированное упорядочение: *потом*, но как можно скорее.

Попробуем представить API для планирования заданий (напрямую, без всяких трюков); назовем его `schedule(..)`:

```
console.log( "A" );

setTimeout( function(){
    console.log( "B" );
}, 0 );

// теоретический "API задания"
schedule( function(){
    console.log( "C" );

    schedule( function(){
        console.log( "D" );
    } );
} );
```

Можно было бы ожидать, что этот фрагмент выведет `A B C D`, но вместо этого он выведет `A C D B`, потому что задания отрабатывают в конце текущего тика цикла событий, а таймер будет запланирован на следующий тик цикла событий.

В главе 3 будет показано, что асинхронное поведение обещаний основано на заданиях, поэтому очень важно четко представлять, как оно связано с поведением цикла событий.

Упорядочение команд

Порядок записи команд в вашем коде не обязательно совпадает с порядком их выполнения движком JS. На первый взгляд это утверждение выглядит довольно странно, так что мы вкратце рассмотрим его.

Но сначала необходимо абсолютно четко прояснить один момент: правила/грамматика языка (см. книгу «Типы и грамматические конструкции» этой серии) диктуют предельно предсказуемое и надежное поведение упорядочения команд с точки зрения программы. Поэтому сейчас мы рассмотрим *то, что никогда не должно наблюдаться* в ваших программах JS.

Если вы когда-нибудь сможете наблюдать переупорядочение команд компилятором, которое мы сейчас собираемся представить, это будет очевидным нарушением спецификации, однозначно указывающим на ошибку в движке JS — ошибку, о которой следует как можно быстрее сообщить для исправления! Тем не менее намного чаще встречается другая ситуация: вы подозреваете, что в движке JS происходит нечто немыслимое, хотя на самом деле это всего лишь ошибка (возможно, состояние гонки!) в вашем собственном коде, поэтому сначала поищите ошибку у себя... а потом еще и еще раз. Отладчик JS с возможностью расстановки точек прерывания и пошагового выполнения кода станет самым мощным инструментом для выявления таких ошибок в вашем коде.

Пример:

```
var a, b;

a = 10;
b = 30;

a = a + 1;
b = b + 1;

console.log( a + b ); // 42
```

В этом коде нет четко выраженной асинхронности (если не считать редко встречающейся асинхронности консольного ввода/вывода, о которой говорилось выше!), так что разумнее всего предположить, что программа будет выполняться строка за строкой сверху вниз.

Тем не менее может оказаться, что движок JS после компиляции кода (да, код JS *компилируется* — см. книгу «Область видимости и замыкания» этой серии) может обнаружить возможность ускорить выполнение кода за счет (безопасного) переупорядочения команд. И в общем-то пока вы не замечаете изменившегося порядка, это вполне нормально.

Например, движок может обнаружить, что следующий код будет выполняться быстрее:

```
var a, b;

a = 10;
a++;
b = 30;
b++;

console.log( a + b ); // 42
```

Или такой:

```
var a, b;

a = 11;
b = 31;

console.log( a + b ); // 42
```

Или даже такой:

```
// так как `a` и `b` больше не используются,
// можно просто подставить нужное значение!
console.log( 42 ); // 42
```

Во всех этих случаях движок JS выполняет безопасные оптимизации во время компиляции, и конечный наблюдаемый результат остается неизменным. Однако существует ситуация, в которой эти конкретные оптимизации ненадежны и должны быть запрещены (конечно, это не означает, что любая оптимизация вообще невозможна):

```
var a, b;

a = 10;
b = 30;

// нужны `a` и `b` в состоянии до увеличения!
console.log( a * b ); // 300

a = a + 1;
b = b + 1;

console.log( a + b ); // 42
```

Есть и другие примеры, в которых переупорядочение команд компилятором создаст наблюдаемые побочные эффекты (а следовательно, должны быть запрещены): это вызовы функций с побочными эффектами (прежде всего геттеры) и объекты ES6 `Proxy`.

Пример:

```
function foo() {
    console.log( b );
    return 1;
}

var a, b, c;

// синтаксис литералов геттеров ES5.1
c = {
    get bar() {
        console.log( a );
        return 1;
    }
};

a = 10;
b = 30;

a += foo();             // 30
b += c.bar;             // 11

console.log( a + b );   // 42
```

Если бы не команды `console.log(..)` в этом фрагменте (которые используются только как удобная форма наблюдаемого побочного эффекта для иллюстрации), движок JS мог бы свободно, по своему желанию (а кто знает, чего он пожелает?!), переупорядочить код в следующем виде:

```
// ...
a = 10 + foo();
b = 30 + c.bar;
// ...
```

К счастью, семантика JS защищает нас от *наблюдаемых* (observable) кошмаров, к которым могло бы привести переупорядочение команд компилятором, важно понимать, насколько непрочна связь между тем, как пишется исходный код, и тем, как он выполняется после компиляции.

Переупорядочение команд компилятором почти что является микрометафорой параллельного выполнения и взаимодействий. В общем случае понимание этой темы поможет вам лучше разобраться в проблемах асинхронного выполнения кода JS.

Итоги

Программа JavaScript (практически) всегда разбивается на два и более блока: первый блок выполняется *сейчас*, а следующий блок будет выполняться *потом* в ответ на событие. Хотя программа выполняется по блокам, все они совместно используют одинаковый доступ к области видимости и состоянию программы, так что каждое изменение состояния осуществляется поверх предыдущего состояния.

Каждый раз, когда в системе имеются события для обработки, цикл событий выполняется до тех пор, пока очередь не опустеет. Каждая итерация цикла событий называется тиком. Действия

пользователя, операции ввода/вывода и таймеры помещают события в очередь событий.

В любой момент времени может обрабатываться только одно событие из очереди. В то время, пока событие выполняется, оно может прямо или косвенно породить одно или несколько последующих событий.

Под *параллельным выполнением* понимается чередование двух и более цепочек событий во времени, так что с высокоуровневой точки зрения они вроде бы выполняются одновременно (хотя в любой момент времени обрабатывается только одно событие).

Часто бывает необходимо в той или иной форме координировать выполнение параллельных «процессов» (не путать с процессами операционной системы!), например, чтобы гарантировать упорядочение или предотвратить состояние гонки. Эти «процессы» также могут выполняться в кооперативном режиме, для чего они разбиваются на меньшие блоки и дают возможность выполняться другим чередующимся «процессам».

2 Обратные вызовы

В главе 1 была представлена терминология и основные концепции асинхронного программирования в JavaScript. При этом нас прежде всего интересовала однопоточная очередь цикла событий, управляющая всеми событиями (асинхронные вызовы функций). Также рассматривались различные способы объяснения паттернами параллельного выполнения отношений (если они существуют!) между одновременно выполняемыми цепочками событий, или «процессами» (задачи, вызовы функций и т. д.).

Во всех примерах главы 1 функции использовались как изолированные неделимые блоки вычислений: внутри функции команды выполняются в предсказуемом порядке (над уровнем компилятора), но на уровне упорядочения функций события (или асинхронные вызовы функций) могут происходить в разном порядке.

Во всех таких случаях функция может рассматриваться как *обратный вызов*, потому что через нее цикл событий обращается с «обратным вызовом» к программе при обработке соответствующего элемента в очереди.

Несомненно, вы уже заметили, что обратные вызовы чаще всего используются для выражения асинхронности и управления ею в программах JS. И действительно, паттерн «обратный вызов»

является самым фундаментальным асинхронным паттерном в языке.

Бесчисленные программы JS, даже очень сложные и хитроумные, были написаны на базе асинхронных средств, не выходящих за рамки обратного вызова (конечно, с паттернами параллельного выполнения, расмотренными в главе 1). *Функция обратного вызова* — это «рабочая лошадка» асинхронности в JavaScript, и она вполне достойно справляется со своей работой.

Вот только... у обратных вызовов тоже есть недостатки. Многие разработчики соблазняются обещаниями (намеренный каламбур!) более совершенных асинхронных паттернов. Тем не менее любая абстракция может эффективно использоваться только в том случае, если вы понимаете, что она абстрагирует и зачем.

В этой главе мы подробно рассмотрим пару таких недостатков. Они объясняют, почему возникла необходимость в более сложных асинхронных паттернах (которые рассматриваются в следующих главах книги и в приложении Б).

Продолжения

Вернемся к примеру с асинхронными обратными вызовами, который был начат в главе 1. Я слегка изменил его, чтобы более наглядно выделить суть вопроса:

```
// A
ajax( "..", function(..){
    // C
} );
// B
```

`// A` и `// B` представляют первую половину программы («*сейчас*»), а `//C` — вторую половину программы («*потом*»). Первая половина выполняется в данный момент, после нее следует пауза неопределенной длины. Затем в какой-то момент при завершении вы-

зова Ajax программа продолжает работу с того момента, на котором она была приостановлена, и продолжает на второй половине.

Иначе говоря, функция обратного вызова заключает (или *инкапсулирует*) продолжение программы.

Давайте еще сильнее упростим код:

```
// A
setTimeout( function(){
    // C
}, 1000 );
// B
```

Остановитесь и спросите себя, как бы вы описали (человеку, менее знакомому с тем, как работает JS) поведение программы. Давайте, расскажите вслух. Это хорошее упражнение, с которым мои следующие рассуждения покажутся более осмысленными.

Вероятно, многие читатели подумали или сказали что-то вроде «Выполнить A, затем установить таймер на ожидание 1000 миллисекунд. Когда таймер сработает, выполнить C». Насколько близко это описание к вашему варианту?

А может быть, здесь вы спохватились и поправились: «Выполнить A, затем установить таймер на 1000 миллисекунд, потом выполнить B. Когда таймер сработает, выполнить C». Такое описание точнее первой версии. Заметили отличие?

Хотя вторая версия точнее первой, оба варианта не могут объяснить код так, чтобы ваши рассуждения соответствовали логике кода, а код — движку JS. Различия одновременно неочевидны и огромны, в них заключается суть понимания недостатков обратных вызовов как средства выражения и управления асинхронностью.

Стоит добавить одно продолжение (или несколько десятков, как во многих программах!) в форме функции обратного вызова, как возникает расхождение между работой вашего мозга и способом работы кода. Каждый раз, когда возникает такое расхождение (а как вам наверняка известно, это далеко не единственное место, где

случается нечто подобное!), вы сталкиваетесь с неизбежным фактом: ваш код становится труднее понять и проследить за его логикой, он создает больше проблем с отладкой и сопровождением.

Последовательное мышление

Уверен, что многие читатели слышали от других, а то и заявляли сами: «Я работаю в многозадачном режиме». Попытки работать в многозадачном режиме лежат в широком диапазоне, от повседневных (жевать резинку во время ходьбы) до откровенно опасных (отправлять SMS во время управления машиной).

Но насколько мы многозадачны? Действительно ли мы способны выполнять два сознательных, намеренных действия одновременно и думать о них в один момент времени? Работает ли параллельная многозадачность на верхнем уровне функциональности нашего мозга?

Ответ вас может удивить: по всей видимости, нет.

Дело даже не в том, как устроен наш мозг. Мы по своей природе куда более однозадачны, чем многие из нас готовы признать. В любой момент времени мы можем думать только о чем-то одном.

Я не говорю о непроизвольных, подсознательных, автоматических функциях мозга, таких как сердцебиение, дыхание и моргание. Все эти задачи критичны для поддержания жизни, но мы не выделяем на них ресурсы мозга. К счастью, пока вы в пятнадцатый раз за три минуты проверяете новости социальных сетей, ваш мозг занимается этими важными задачами в фоновом режиме.

Нет, я имею в виду те задачи, которые находятся на переднем плане вашей умственной деятельности в данный момент. Скажем, для меня сейчас это написание текста книги. Занимаюсь ли я другой высокоуровневой умственной деятельностью именно в текущий момент? Пожалуй, нет. Я быстро и легко отвлекаюсь, несколько десятков раз за последнюю пару абзацев!

Когда мы имитируем многозадачность, например пытаемся что-нибудь набрать на клавиатуре и в то же время говорим с другом по телефону, то происходит нечто вроде быстрого переключения контекста. Иначе говоря, мы очень быстро переключаемся между двумя и более задачами, а каждая задача продвигается вперед крошечными, но очень быстрыми шагами. Все это происходит так быстро, что кажется, будто задачи выполняются параллельно.

А вам не кажется, что такое описание подозрительно смахивает на асинхронное парралельное выполнение с событиями (вроде того, что происходит в JS)?! Если не кажется, то вернитесь к главе 1 и перечитайте ее.

Получается, что наш мозг работает примерно по тому же принципу, что и очередь цикла событий (если уж я пытаюсь упростить неимоверно сложный мир нейрологии до чего-то, что можно хотя бы отдаленно обсудить в этой книге).

Если рассматривать каждую отдельную букву (или слово), которую я набираю на клавиатуре, как отдельное асинхронное событие, то только в этом предложении у моего мозга возникает несколько десятков возможностей переключиться на другое событие, например, от моих органов чувств или просто от случайных размышлений.

Я не прерываюсь и не перехожу на другой «процесс» при каждой представившейся возможности (и это хорошо, иначе эта книга никогда бы не была закончена!). Но это происходит достаточно часто, чтобы мне казалось, что мой мозг почти моментально переключается на другие контексты (то есть «процессы»). И наверное, мои ощущения очень близки к тому, как мог бы воспринимать происходящее движок JS.

Работа и планирование

Итак, можно представить, что наш мозг работает в условиях однопоточной очереди цикла событий, как и движок JS. Такое сравнение выглядит вполне разумно.

Тем не менее анализ не должен ограничиться поверхностным сходством. Между тем, как мы планируем различные задачи, и тем, как наш мозг выполняет эти задачи, существует большое и заметное различие.

И снова вернемся к написанию этого текста как метафоре. Мой приблизительный творческий план заключается в том, чтобы писать и писать, последовательно проходя через определенные точки, которые я упорядочил в своих размышлениях. Я не планирую никакие прерывания или нелинейные операции в процессе работы над книгой. Тем не менее мой мозг постоянно на что-нибудь переключается.

И хотя на оперативном уровне наш мозг работает по принципу асинхронных событий, мы склонны планировать задачи последовательно и синхронно: «нужно сходить в магазин, купить молока и отнести одежду в химчистку». Заметьте, что это высокоуровневое мышление (планирование) в своей формулировке имеет мало общего с асинхронными событиями. Собственно, мы довольно редко мыслим понятиями событий. Вместо этого мы планируем все тщательно и последовательно (A, затем B, затем C) и предполагаем своего рода временную блокировку, которая заставляет B ожидать завершения A, а C — ожидать завершения B.

Когда разработчик пишет код, он планирует некий набор действий. Если он чего-то стоит как разработчик, эти действия тщательно планируются. «Я должен присвоить `z` значение `x`, а потом присвоить `x` значение `y`» и т. д.

Когда мы пишем синхронный код, команда за командой, он работает по тому же принципу, что и наш список дел:

```
// поменять местами `x` и `y` (через временную переменную `z`)
z = x;
x = y;
y = z;
```

Эти три команды присваивания синхронны; `x = y` ожидает завершения `z = x`, а `y = z` ожидает завершения `x = y`.

То же самое сказать иначе: эти три команды связаны по времени; они должны выполняться в определенном порядке, одна за другой. К счастью, нам не нужно беспокоиться о подробностях асинхронных событий (иначе код бы серьезно усложнился).

Итак, если синхронное планирование в мозге хорошо соответствует синхронным командам в коде, насколько хорошо наш мозг справляется с планированием асинхронного кода?

Оказывается, выражение асинхронности (с обратными вызовами) в коде не так уж хорошо соответствует поведению мозга по синхронному планированию.

Можете ли вы представить примерно такой ход мыслей при планировании текущих дел?

«Я должен сходить в магазин, но по пути наверняка мне позвонят, так что “Привет, мама”, и пока она говорит, я буду искать адрес магазина на GPS, но на загрузку потребуется около секунды, поэтому я выключу радио, чтобы лучше слышать телефон, но потом я пойму, что забыл надеть куртку, а снаружи холодно, но я все равно буду вести машину и говорить с мамой, а потом сигнал напомнит мне, что нужно пристегнуть ремень безопасности, и “Да, мама, я пристегнулся, как всегда!” Наконец-то на GPS появился маршрут, теперь...»

Как бы смешно ни звучало такое описание наших планов на день и порядка выполняемых дел, на функциональном уровне наш мозг работает именно так. Помните: это не многозадачность, а всего лишь быстрое переключение контекста.

Причина, по которой разработчикам трудно писать код с асинхронными событиями, а особенно при использовании обратных вызовов для этой цели, состоит в том, что такой стиль мышления/планирования, больше смахивающий на поток сознания, не является естественным для большинства из нас.

Мы мыслим пошаговыми категориями, но инструменты (обратные вызовы), доступные для нас в коде, не выражаются в пошаговом

виде при переходе от синхронного программирования к асинхронному.

Вот почему так трудно писать асинхронный код с обратными вызовами и анализировать его: в нашем мозге планирование осуществляется на другом уровне.

Вы не понимаете, почему ваш код перестал работать… Что может быть хуже? Только одно: если вы не понимаете, почему он работал до этого! Классический менталитет «карточного домика»: «Это работает, но я не знаю, почему и как, поэтому — ничего не трогать!» Возможно, вы слышали фразу Сартра «Ад — это другие» или ее программистский аналог «Ад — это код других людей». Я свято верю в принцип: «Ад — это непонимание собственного кода». И обратные вызовы часто становятся главным виновником таких проблем.

Вложенные/сцепленные обратные вызовы

Пример:

```
listen( "click", function handler(evt){
    setTimeout( function request(){
        ajax( "http://some.url.1", function response(text){
            if (text == "hello") {
                handler();
            }
            else if (text == "world") {
                request();
            }
        } );
    }, 500) ;
} );
```

Скорее всего, этот код покажется вам знакомым. В нем используется цепочка из трех функций, каждая из которых представляет этап асинхронной серии (задачу, «процесс»).

Такой код часто называется «кошмаром обратных вызовов» (callback hell); также его именуют «роковой пирамидой» (pyramid

of doom) (вложенные отступы напоминают треугольник, лежащий на боку).

Но «кошмар обратных вызовов» не имеет почти ничего общего с вложением/отступами. Проблема лежит гораздо глубже. В этой главе все «как» и «почему» будут рассмотрены более подробно.

Сначала мы ожидаем события `click`, затем ожидаем срабатывания таймера, затем ожидаем возвращения ответа Ajax — после чего все может повториться снова.

На первый взгляд асинхронность этого кода вроде бы естественно соответствует последовательному стилю мышления.

Сначала (*сейчас*):

```
listen( "..", function handler(..){
    // ..
} );
```

Потом:

```
setTimeout( function request(..){
    // ..
}, 500) ;
```

Потом еще позднее:

```
ajax( "..", function response(..){
    // ..
} );
```

И наконец (*потом* в самом конце):

```
if ( .. ) {
    // ..
}
else ..
```

Однако при таком линейном подходе к анализу кода возникает ряд проблем.

Во-первых, тот факт, что разные этапы располагаются в последовательных строках (1, 2, 3 и 4...), — чистая случайность. В реальных асинхронных программах код загромождается посторонним шумом, который нам придется мысленно обходить при переходе от одной функции к другой. Понять асинхронную последовательность выполнения в коде с большим количеством обратных вызовов возможно, однако сделать это не просто и не очевидно даже при изрядном опыте.

Однако существуют и другие, более глубокие проблемы, невидимые в этом примере. Представим другую ситуацию (на псевдокоде) для демонстрации:

```
doA( function(){
    doB();

    doC( function(){
        doD();
    } )

    doE();
} );

doF();
```

Хотя опытный читатель правильно определит истинную последовательность операций, я готов поспорить, что ситуация на первый взгляд кажется довольно запутанной, и чтобы прийти к правильному ответу, придется основательно поразмыслить. Операции выполняются в следующем порядке:

- `doA()`
- `doF()`
- `doB()`
- `doC()`
- `doE()`
- `doD()`

А вам удалось дать правильный ответ с первого раза?

Допустим, некоторые читатели полагают, что я задал имена функций, чтобы намеренно увести вас в сторону. Честное слово, я просто расставлял имена сверху вниз. Но попробуем еще раз:

```
doA( function(){
    doC();

    doD( function(){
        doF();
    } )

    doE();
} );

doB();
```

Теперь я присвоил имена по алфавиту в порядке фактического выполнения. Но я уверен, что даже при наличии опыта в этой ситуации отслеживание порядка A → B → C → D → E → F не покажется вам чем-то естественным. Приходится постоянно бегать взглядом вверх-вниз, верно?

Но даже если все это у вас получается само собой, осталась еще одна опасность, которая может вызвать хаос. Догадаетесь, о чем я?

Что, если `doA(..)` или `doD(..)` на самом деле не асинхронны, как мы предполагали? Порядок изменяется. Если эти функции синхронны (и, возможно, только в отдельных случаях в зависимости от условий в программе), получается порядок A → C → D → F → E → B.

Тихий шорох, который вы услышали, это вздох тысяч JS-разработчиков, в отчаянии обхвативших голову руками. Так вложение — это проблема? Это оно так усложняет отслеживание асинхронного порядка выполнения? Безусловно, это часть проблемы.

Но позвольте мне переписать предыдущий пример с вложенными событиями/тайм-аутом/Ajax без использования вложения:

```
listen( "click", handler );

function handler() {
    setTimeout( request, 500 );
}

function request(){
    ajax( "http://some.url.1", response );
}

function response(text){
    if (text == "hello") {
        handler();
    }
    else if (text == "world") {
        request();
    }
}
```

В такой формулировке кода беды вложения уже не столь очевидны, как в предыдущей форме, и все же он в той же степени подвержен «кошмару обратных вызовов». Почему?

При линейном (последовательном) анализе этого кода нам приходится переходить от одной функции к другой, потом к следующей и т. д., прыгая по всей кодовой базе, чтобы «разглядеть» последовательность выполнения. Напомню, что это упрощенный код, написанный исключительно для демонстрации. Все мы знаем, что реальные кодовые базы асинхронных программ JS часто бывают куда более навороченными, из-за чего подобный анализ усложняется на порядок.

Еще одно обстоятельство: чтобы этапы 2, 3 и 4 были сцеплены для выполнения в строго определенном порядке, сами по себе обратные вызовы предоставляют только одну возможность: жестко запрограммировать этап 2 в этапе 1, этап 3 в этапе 2, этап 4 в этапе 3 и т. д. Такое жесткое программирование не всегда плохо; в действительности оно означает фиксированное условие, что этап 2 всегда должен приводить к этапу 3. Однако жесткое программирование определенно делает код менее надежным, поскольку он не учитывает возможных проблем, которые могут вызвать

отклонение в последовательности этапов. Например, если на этапе 2 произойдет сбой, этап 3 никогда не будет достигнут, программа не попытается повторно выполнить этап 2, не произойдет переход на альтернативную ветвь обработки ошибок и т. д.

Все эти возможности можно вручную запрограммировать на каждом этапе, но такой код получится повторяющимся, и его нельзя будет повторно использовать в других этапах или других асинхронных ветвях вашей программы.

Несмотря на то что ваш мозг может планировать серию задач в последовательном виде (сначала это, потом то, затем другое), благодаря событийной природе мозга восстановление/повтор/ветвление реализуются почти без всяких усилий. Если вы пошли за покупками и обнаружили, что забыли список дома, это не станет трагедией только потому, что вы не учли эту возможность заранее. Мозг легко обойдет эту заминку: вы возвращаетесь домой, забираете список и снова идете в магазин.

Но хрупкость вручную запрограммированных обратных вызовов (даже с жестко запрограммированной обработкой ошибок) часто оказывается не столь элегантной. После того как вы закончите определять (или предварительно планировать) все возможности/пути, код становится настолько запутанным, что вам будет трудно сопровождать или обновлять его.

В *этом* проявляется суть «кошмара обратных вызовов». Вложение/отступы — всего лишь отвлечение внимания.

А если вас это не убеждает, мы даже не притронулись к тому, что происходит при одновременном выполнении двух и более цепочек этих продолжений обратных вызовов, или когда третий этап переходит в параллельные обратные вызовы с шлюзами или защелками, или... OMG, у меня уже голова раскалывается! А у вас?

Вы уловили суть: наше последовательное, блокирующее поведение планирования мозга плохо соответствует асинхронному коду, ориентированному на обратные вызовы. Это первый существенный недостаток обратных вызовов: они выражают асинхронность

в коде способами, для осознания которых нашему мозгу придется прилагать массу усилий.

Проблемы доверия

Несоответствие между последовательным характерам планирования мозга и асинхронным кодом JS на основе обратных вызовов — всего лишь одна часть проблемы с обратными вызовами. Существуют более глубокие проблемы, о которых нужно побеспокоиться.

Давайте вернемся к концепции функции обратного вызова как продолжения (или второй половины) вашей программы:

```
// A
ajax( "..", function(..){
    // C
} );
// B
```

`// A` и `// B` выполняются *сейчас*, под прямым управлением главной программы JS. Но `// C` откладывается для выполнения *потом*, под контролем другой стороны — в данном случае функции `ajax(..)`. В принципе подобная передача управления не обязательно создает проблемы для программ.

Но не стоит обманываться ее редкостью и предполагать, что переключение управления — обычное дело. На самом деле это одна из худших (и при этом самых коварных) проблем проектирования, ориентированного на обратные вызовы. Суть в том, что иногда `ajax(..)` (то есть сторона, которой вы передаете продолжение своего обратного вызова) не является функцией, написанной вами или находящейся под вашим прямым контролем. Достаточно часто это служебная функция, предоставленная некой третьей стороной.

Это явление называется *инверсией управления*: вы берете часть своей программы и предоставляете контроль над ее выполнением

другой стороне. Между вашим кодом и сторонней функцией существует необъявленный контракт — набор условий, на выполнение которых вы рассчитываете.

История о пяти обратных вызовах

Возможно, важность этого момента не совсем очевидна. Позвольте мне сконструировать преувеличенный сценарий, который показывает, чем опасно такое доверие.

Представьте, что вы — разработчик, которому поручено построить систему оформления заказов для сайта, продающего дорогие телевизоры. Все страницы системы уже успешно построены. На последней странице, когда пользователь щелкает на кнопке «Подтвердить» для покупки, необходимо вызвать стороннюю функцию (предоставленную, допустим, аналитической компанией) для отслеживания факта продажи.

Вы замечаете, что предоставленная функция отслеживания является асинхронной, возможно, по соображениям быстродействия. Это означает, что ей нужно передать функцию обратного вызова. В продолжении, которое вы передаете, содержится итоговый код, который снимает средства с кредитной карты покупателя и выводит страницу с благодарностью.

Код может выглядеть так:

```
analytics.trackPurchase( purchaseData, function(){
    chargeCreditCard();
    displayThankyouPage();
} );
```

Все просто, верно? Вы пишете код, тестируете его, все работает, код уходит в эксплуатацию. Все счастливы!

Проходит полгода, никаких проблем. Вы почти забыли, что когда-то писали этот код. Но вот однажды утром, когда вы заходите

в кафетерий перед работой и смакуете латте, вам звонит встревоженный начальник. Он требует, чтобы вы бросили кофе и срочно мчались на работу. Оказавшись на месте, вы узнаете, что у особо ценного клиента за один купленный телевизор оплата с кредитки была списана 5 раз. Понятное дело, клиенту это совершенно не понравилось. Служба поддержки уже принесла извинения и запустила возврат средств, но начальник желает знать, как это вообще могло произойти. «Разве у нас нет тестов для таких случаев?!» А вы даже не помните написанный вами код. Но вы снова закапываетесь в него и пытаетесь понять, что могло пойти не так.

После анализа журнальных данных вы приходите к выводу, что возможно только одно объяснение: аналитическая функция неизвестно почему вызвала вашу функцию обратного вызова пять раз вместо одного. В их документации об этом ничего не сказано.

Вы связываетесь с их службой поддержки, которая пребывает в таком же изумлении, как и вы. Специалисты обещают связаться со своими разработчиками и сообщить о результате. На следующий день вам приходит длинное сообщение с объяснениями, которое вы поспешно приносите начальству.

Оказывается, разработчики из аналитической компании работали над неким экспериментальным кодом, который в определенных условиях пытается повторять переданный обратный вызов один раз в секунду, и так пять раз, и только после этого происходит ошибка тайм-аута. Этот код не должен был попасть в готовый продукт, но все же как-то попал. Разработчики сильно смущены и извиняются. Они подробно рассказывают о том, как нашли проблему, и что предприняли, чтобы она никогда не повторилась... Бла-бла-бла.

Что дальше?

Вы докладываете начальнику, но его такое положение дел не устраивает. Он настаивает (и вы неохотно соглашаетесь), что им доверять больше нельзя, и теперь нужно каким-то образом защитить код оформления заказа от таких уязвимостей.

Через некоторое время вы пишете простую «заплатку», которая вроде бы всех устраивает:

```
var tracked = false;

analytics.trackPurchase( purchaseData, function(){
    if (!tracked) {
        tracked = true;
        chargeCreditCard();
        displayThankyouPage();
    }
} );
```

Этот код должен быть знаком вам по первой главе, так как мы фактически создаем задвижку для множественных параллельных активизаций нашего обратного вызова.

Но тут один из инженеров по тестированию спрашивает: «А что произойдет, если они вообще не вызовут нашу функцию?» Какая неприятность. Никто из нас не подумал об этом.

Вы начинаете расследование и продумываете все возможные проблемы, которые могут возникнуть при вызове вашей функции обратного вызова. Приблизительный список того, что только аналитическая функция может сделать не так:

- Вызвать функцию обратного вызова слишком рано (до того, как факт продажи был зарегистрирован).
- Вызвать функцию обратного вызова слишком поздно (или вообще никогда).
- Вызвать функцию обратного вызова слишком мало или слишком много раз (как в обнаруженной вами проблеме).
- Не передать необходимые переменные среды/параметры вашей функции обратного вызова.
- Поглотить любые возникшие ошибки/исключения.
- ...

Список производит гнетущее впечатление — так оно и есть. Вы понемногу начинаете сознавать, что вам придется писать специальную логику *для каждого без исключения* обратного вызова, который передается стороне, в надежности которой вы не уверены.

Теперь вы стали еще лучше понимать, что такое «кошмар обратных вызовов».

Не только в чужом коде

Некоторые читатели могут усомниться, а не преувеличиваю ли я серьезность проблемы? Может, вы редко взаимодействуете со сторонним кодом или не взаимодействуете вовсе. Может быть, используете API с контролем версий или размещаете такие библиотеки на собственном сервере, так что их поведение не может быть изменено незаметно для вас.

Итак, задумайтесь над вопросом: можете ли вы в полной мере доверять тем средствам, которые теоретически находятся под вашим контролем (то есть в вашей кодовой базе)?

Многие из нас согласятся с тем, что, по крайней мере, до какой-то степени в свои внутренние функции следует включать защитные проверки входных параметров для предотвращения/снижения непредвиденных проблем.

Чрезмерное доверие входным данным:

```
function addNumbers(x,y) {
    // + перегружается с преобразованием типа для выполнения
    // конкатенации строк, так что эта операция
    // не является строго безопасной в зависимости
    // от переданных параметров.
    return x + y;
}

addNumbers( 21, 21 );   // 42
addNumbers( 21, "21" ); // "2121"
```

Защита от ненадежного ввода:

```
function addNumbers(x,y) {
    // проверка ввода на числовые данные
    if (typeof x != "number" || typeof y != "number") {
        throw Error( "Bad parameters" );
    }

    // если управление передано сюда, + безопасно
    // выполнит сложение чисел
    return x + y;
}

addNumbers( 21, 21 );   // 42
addNumbers( 21, "21" ); // Error: "Bad parameters"
```

Тоже безопасно, но выглядит лучше:

```
function addNumbers(x,y) {
    // проверка ввода на числовые данные
    x = Number( x );
    y = Number( y );

    // + безопасно выполнит сложение чисел
    return x + y;
}

addNumbers( 21, 21 );   // 42
addNumbers( 21, "21" ); // 42
```

Какой бы вариант вы ни выбрали, такие проверки/нормализации достаточно часто встречаются при работе с входными данными функций, даже в коде, которому мы теоретически полностью доверяем. В каком-то примитивном смысле это программистский эквивалент геополитического принципа «доверяй, но проверяй».

Разве не логично, что такой же подход следует применять при создании асинхронных функций обратного вызова не только с внешним кодом, но и с кодом, который в общем случае находится под нашим контролем? Конечно, следует.

Однако обратные вызовы на самом деле никак не помогают нам в этом деле. Всю механику приходится строить самостоятельно,

и часто дело сводится к большим объемам шаблонного кода, который приходится повторять для каждого асинхронного обратного вызова.

Самая хлопотная проблема с обратными вызовами — инверсия управления, приводящая к полному нарушению по всем линиям доверия. Если у вас есть код, использующий эти обратные вызовы (особенно в сочетании со сторонними средствами, но не только) и вы еще не применяете никакую логику снижения рисков, связанных с проблемами доверия при инверсии управления, то ваш код уже содержит ошибки, хотя возможно, вы с ними еще не сталкивались. Незамеченные ошибки все равно остаются ошибками.

Кошмар, да и только!

Попытки спасти обратные вызовы

У архитектуры обратных вызовов есть несколько вариаций, которые пытаются решить некоторые (но не все!) проблемы доверия, рассмотренные выше. Это доблестные, но безнадежные попытки спасти паттерн «обратный вызов» от уничтожения самого себя.

Например, для более элегантной обработки ошибок некоторые API предоставляют возможность *расщепления обратных вызовов* (для оповещений об успехе и для оповещений об ошибке):

```
function success(data) {
    console.log( data );
}

function failure(err) {
    console.error( err );
}
ajax( "http://some.url.1", success, failure );
```

В API с такой архитектурой обработчик ошибок `failure()` часто не является обязательным. Если вы предпочитаете, чтобы ошибки бесследно поглощались (фу!), то он не предоставляется.

Архитектура расщепления обратных вызовов используется Promise APIв ES6. Обещания ES6 будут гораздо более подробно рассмотрены в главе 3.

Другой распространенный паттерн обратных вызовов называется «*ошибка на первом месте*» (иногда также встречается термин «*стиль Node*», поскольку это соглашение используется почти во всех API Node.js): первый аргумент единого обратного вызова резервируется для объекта ошибки (если он есть). В случае успеха этот аргумент будет пустым/ложным (а любые последующие аргументы содержат данные успешного вызова), но при получении сигнала об ошибке первый аргумент задан/истинен (а другие аргументы обычно не передаются):

```
function response(err,data) {
    // ошибка?
    if (err) {
        console.error( err );
    }
    // если нет - предполагается успех
    else {
        console.log( data );
    }
}
ajax( "http://some.url.1", response );
```

В обоих случаях следует обратить внимание на ряд обстоятельств.

Во-первых, этот паттерн не решает большинство проблем доверия, как могло бы показаться. Ни в одном варианте обратного вызова нет ничего, что предотвращало бы или фильтровало нежелательные повторные вызовы. Более того, ситуация даже ухудшилась, потому что вы можете получить сигналы об успехе и ошибке одновременно или не получить ни один из них, и вам все равно придется писать для этих условий дополнительный код.

Также не забывайте о том, что хотя это стандартный паттерн, который можно применять на практике, он занимает больше места и содержит больше шаблонного кода, который вряд ли можно

будет использовать повторно. Вам быстро надоест вводить весь этот код для каждого отдельного обратного вызова в приложении.

Как насчет проблемы, что функция никогда не будет вызвана? Если вас беспокоит такая возможность (как и должно быть!), то, скорее всего, следует назначить тайм-аут для отмены события. Для этого можно написать специальную функцию (далее приводится только заготовка):

```
function timeoutify(fn,delay) {
    var intv = setTimeout( function(){
            intv = null;
            fn( new Error( "Timeout!" ) );
        }, delay )
    ;

    return function() {
        // тайм-аут еще не случился?
        if (intv) {
            clearTimeout( intv );
            fn.apply( this, arguments );
        }
    };
}
```

А вот как она будет использоваться:

```
// использование паттерна обратного вызова "ошибка
// на первом месте"
function foo(err,data) {
    if (err) {
        console.error( err );
    }
    else {
        console.log( data );
    }
}

ajax( "http://some.url.1", timeoutify( foo, 500 ) );
```

Другая потенциальная проблема доверия — слишком ранний вызов. В контексте приложения это может означать вызов до завер-

шения некоторого критического условия. Но на более общем уровне проблема проявляется в утилитах, которые могут вызвать функцию обратного вызова переданную вами либо *сейчас* (синхронно), либо *потом* (асинхронно).

Недетерминизм, окружающий поведение «синхронно/асинхронно», почти всегда порождает ошибки, которые бывает очень трудно найти. В определенных кругах для описания кошмара «синхронно/асинхронно» используется вымышленный монстр по имени Залго, наводящий безумие. Распространенный призыв «Не выпускайте Залго!» ведет к очень разумному совету: всегда активизируйте обратные вызовы асинхронно, даже если они отработают прямо на следующей итерации цикла событий, чтобы все обратные вызовы были предсказуемо асинхронными.

За дополнительной информацией о Залго обращайтесь к публикациям Орена Голана (Oren Golan) «Don't Release Zalgo!» (https://github.com/oren/oren.github.io/blob/master/posts/zalgo.md) и Айзека З. Шлютера (Isaac Z. Schlueter) «Designing APIs for Asynchrony» (http://blog.izs.me/post/59142742143/designing-apis-for-asynchrony).

Пример:

```
function result(data) {
    console.log( a );
}

var a = 0;

ajax( "..pre-cached-url..", result );
a++;
```

Что выведет этот код — 0 (синхронный обратный вызов) или 1 (асинхронный обратный вызов)? Это зависит от обстоятельств.

Вы видите, как быстро непредсказуемость Залго начинает угрожать любой программе JS. Таким образом, глупо звучащая фраза «Не выпускайте Залго!» на самом деле оборачивается в высшей

степени разумным и универсальным советом. Всегда используйте асинхронность.

А если вы не знаете, будет ли API всегда использовать асинхронный вызов? Можно придумать специальную функцию — что-то вроде следующей заготовки `asyncify(..)`:

```
function asyncify(fn) {
    var orig_fn = fn,
        intv = setTimeout( function(){
            intv = null;
            if (fn) fn();
        }, 0 )
    ;

    fn = null;

    return function() {
        // активируется слишком быстро, до срабатывания
        // таймера `intv`,
        // обозначающего прохождение асинхронного периода?
        if (intv) {
            fn = orig_fn.bind.apply(
                orig_fn,
                // добавить `this` обертки в параметры вызова
                // `bind(..)` и каррировать все переданные
                // параметры.
                [this].concat( [].slice.call( arguments ) )
            );
        }
        // асинхронно
        else {
            // вызвать исходную функцию
            orig_fn.apply( this, arguments );
        }
    };
}
```

Функция `asyncify(..)` используется примерно так:

```
function result(data) {
    console.log( a );
}
```

```
var a = 0;

ajax( "..pre-cached-url..", asyncify( result ) );
a++;
```

Неважно, находится ли запрос Ajax в кэше и разрешение (resolve) приводит к немедленной активизации обратного вызова или же данные передаются по каналу связи, а обратный вызов завершается позднее асинхронно, этот код всегда будет выводить 1 вместо 0. `result(..)` в любом случае будет вызываться асинхронно, это означает, что `a++` сможет отработать до выполнения `result(..)`.

Ура, еще одна проблема доверия «решена»! Однако это решение неэффективно, а в проект добавляется еще больше раздутого шаблонного кода, который становится лишним бременем.

С обратными вызовами эта история повторяется снова и снова. Обратные вызовы позволяют сделать практически все, что вам только потребуется, но вы должны быть готовы основательно поработать для достижения результата. Достаточно часто этих усилий требуется намного больше, чем вам захочется потратить на такой код.

Возможно, у вас возникнет мысль об использовании встроенных API или других языковых механик для подобных проблем. ES6 наконец-то предлагает некоторые достойные решения, так что продолжайте читать!

Итоги

Обратные вызовы — фундаментальная структурная единица асинхронного выполнения в JS. Тем не менее для развивающегося по мере становления JS ландшафта асинхронного программирования их становится недостаточно.

Прежде всего, наш мозг все планирует последовательно, с блокировкой и с однопоточной семантикой, но обратные вызовы вы-

ражают асинхронную последовательность выполнения в нелинейном и непоследовательном виде, что сильно затрудняет правильный анализ такого кода. Код, который трудно анализировать, — это плохой код, который приводит к ошибкам.

Нам необходим способ выражения асинхронности в более синхронном, последовательном, блокирующем виде, то есть так, как работает наш мозг.

Второй, более важный недостаток — обратные вызовы подвержены инверсии управления. Иначе говоря, они неявно передают управление другой стороне (часто стороннему коду, который вам неподконтролен), которая должна активизировать продолжение вашей программы. Передача управления открывает целый список проблем доверия: например, не будет ли функция обратного вызова вызываться чаще, чем вы ожидаете.

Вообще говоря, вы можете изобрести специализированную логику для решения подобных проблем, но сделать сложнее, чем хотелось бы, а полученный код оказывается более неуклюжим, сложным в сопровождении и, скорее всего, недостаточно защищенным от этих рисков, пока вы не столкнетесь с этими ошибками на практике.

Нам хотелось бы иметь обобщенное решение для всех проблем доверия — такое, которое можно было бы повторно использовать для любого количества обратных вызовов без избыточного балласта в виде лишнего шаблонного кода.

Придется найти что-то получше обратных вызовов. Да, они неплохо служили до сих пор, но будущее JavaScript требует более совершенных и мощных асинхронных паттернов. Такие появляющиеся решения будут подробно рассмотрены в следующих главах.

3 Обещания

В главе 2 были выявлены две основные категории недостатков использования обратных вызовов для выражения асинхронности в программе и управления параллельным выполнением: это *отсутствие последовательности* и *отсутствие доверия*. Теперь, когда вы лучше понимаете проблемы, пора обратиться к паттернам, которые могут применяться для их решения.

Начнем с проблемы *инверсии управления* — того самого доверия, которое так непрочно и так легко теряется.

Вспомните, что продолжение программы было упаковано в функцию обратного вызова, которая передавалась другой стороне (а возможно, даже внешнему коду). Вам оставалось только надеяться, что эта сторона правильно поступит при активизации обратного вызова.

Это делалось для того, чтобы выразить идею: «Вот это должно произойти *потом*, после завершения текущей фазы».

Но что, если удалось бы инвертировать инверсию управления? Что, если бы вместо передачи продолжения программы другой стороне можно было бы ожидать, что она вернет нам возможность узнать о завершении задачи, чтобы наш код потом мог решить, что делать дальше?

Эта парадигма называется *обещанием* (promise).

Обещания постепенно захватывают мир JS, поскольку и разработчики и авторы спецификаций отчаянно стремятся распутать безумный клубок «кошмара обратных вызовов» в своем коде/архитектуре. Большинство новых асинхронных API, добавляемых на платформу JS/DOM, строится на обещаниях. Наверное, обещания заслуживают того, чтобы познакомиться с ними поближе, как вы думаете?

В этой главе часто используется слово «немедленно» — обычно для обозначения некоторого действия по разрешению обещания. Однако практически во всех случаях «немедленно» следует понимать в контексте поведения очереди заданий (см. главу 1), а не как «сейчас» в контексте строгой синхронности.

Что такое обещание?

Когда разработчик решает освоить новую технологию или паттерн, обычно все начинается со слов: «Покажите мне код!» Для нас вполне естественно просто прыгнуть в воду и учиться плавать по ходу дела.

Но как выясняется, в одних лишь API теряются некоторые абстракции. Обещания — один из инструментов, по использованию которых становится совершенно ясно, понимает ли разработчик, для чего они нужны и на какой стадии — изучения или использования API — он находится.

Итак, прежде чем показывать код обещаний, я хочу полностью объяснить, что же собой представляют обещания на концептуальном уровне. Надеюсь, после этого вам будет проще интегрировать теорию обещаний в асинхронный поток выполнения вашей программы. Запомним это и рассмотрим две разные аналогии, поясняющие суть обещаний.

Будущее значение

Представьте ситуацию: я подхожу к стойке бистро и заказываю чизбургер. Я отдаю кассиру 1,47 доллара. Разместив свой заказ и оплатив его, я желаю получить нечто в будущем (чизбургер). По сути, я создаю транзакцию.

Но часто оказывается, что чизбургер нужно еще приготовить. Кассир подает мне что-то взамен моего чизбургера: чек с номером заказа. Номер заказа — обязательство, которое гарантирует, что когда-нибудь я получу свой чизбургер.

Итак, я держу чек с номером заказа. Я знаю, что он представляет мой *будущий чизбургер*, так что беспокоиться больше не о чем, если не считать чувства голода!

Во время ожидания я могу заниматься другими вещами, скажем, отправить текстовое сообщение другу: «Эй, не хочешь пообедать со мной? Я как раз заказал чизбургер».

Я уже рассуждаю о своем будущем чизбургере, хотя еще и не держу его в руках. Мозг позволяет это делать, потому что номер заказа как бы заменяет чизбургер. По сути, заказанный чизбургер становится *независимым от времени*.

Через некоторое время я слышу: «Заказ 113!» — и радостно подхожу к стойке с чеком в руке. Я отдаю чек кассиру и забираю вместо него чизбургер.

Другими словами, когда мое «будущее нечто» готово, я обмениваю «обещание» на него.

Однако возможен и другой исход. Я слышу номер своего заказа, но когда подхожу к стойке за чизбургером, кассир с сожалением сообщает: «Извините, но чизбургеры кончились». Забыв на время о недовольстве клиента, мы видим одну важную особенность *будущих значений*: они могут служить признаком успеха или неудачи.

Каждый раз, когда я заказываю чизбургер, я знаю, что со временем получу либо чизбургер, либо печальные новости об отсутствии

чизбургеров, и тогда мне придется выбрать себе на обед что-то другое.

В программе все не так просто, потому что (образно выражаясь) ваш номер заказа могут никогда не назвать, и тогда ситуация бесконечно долго остается в неразрешенном состоянии. Позднее мы вернемся к тому, что делать в таких случаях.

Значения: сейчас и потом

Возможно, пока все это кажется слишком абстрактным для применения в коде. Давайте рассуждать более конкретно.

Но перед тем как объяснять принцип работы обещаний, мы реализуем обработку этих будущих значений в коде, который вам уже понятен, то есть в коде с обратными вызовами!

Когда вы пишете код для выполнения некоторых действий со значением (например, математической операции с числом), вы (сознательно или нет) предполагаете одну фундаментальную характеристику этого значения, а именно то, что оно уже является конкретным, существующим «*сейчас*» значением:

```
var x, y = 2;
console.log( x + y ); // NaN  <-- потому что значение `x`
                      // еще не задано
```

Операция `x + y` предполагает, что значения `x` и `y` уже заданы. На условиях, которые будут пояснены позднее, предполагается, что значения `x` и `y` уже готовы к использованию.

Было бы глупо предполагать, что оператор `+` сам по себе каким-то волшебным образом способен определить, когда будут готовы к использованию оба значения `x` и `y`, и только после того выполнить свою операцию. Если одни команды будут завершаться *сейчас*, а другие — *потом*, это вызовет сущий хаос в программе, не так ли?

Как можно сделать какие-то разумные выводы об отношениях между двумя командами, если может оказаться, что хотя бы одна (или обе) из них еще не завершена? Если команда 2 зависит от завершения команды 1, возможны всего два варианта: либо команда 1 уже завершилась и все пойдет нормально, либо команда 1 еще не завершилась и попытка выполнения команды 2 завершится неудачей.

Если такие рассуждения показались вам отдаленно знакомыми по главе 1 — хорошо!

А теперь вернемся к математической операции `x + y`. Представьте, что у вас есть возможность сказать: «Сложить `x` и `y`, но если хотя бы одно из значений еще не готово, подождать, пока будут готовы. Сложить, как только это будет возможно».

Возможно, у вас промелькнула мысль об обратных вызовах. Тогда...

```
function add(getX,getY,cb) {
    var x, y;
    getX( function(xVal){
        x = xVal;
        // оба значения готовы?
        if (y != undefined) {
            cb( x + y );    // вычислить сумму
        }
    } );
    getY( function(yVal){
        y = yVal;
        // оба значения готовы?
        if (x != undefined) {
            cb( x + y );    // вычислить сумму
        }
    } );
}
// `fetchX()` и `fetchY()` - синхронные или асинхронные
// функции
add( fetchX, fetchY, function(sum){
    console.log( sum ); // все просто, не так ли?
} );
```

Не пожалейте времени и оцените красоту этого фрагмента (или отсутствие оной).

Да, код выглядит уродливо, но в этом асинхронном паттерне есть нечто очень важное.

В этом фрагменте `x` и `y` интерпретируются как будущие значения, а в коде выражается операция `add(..)`, которая (на первый взгляд) не интересуется тем, доступны ли значения `x` и/или `y` в данный момент. Иначе говоря, она нормализует *сейчас* и *потом*, чтобы вы могли твердо рассчитывать на предсказуемость результата операции `add(..)`. Использование функции `add(..)`, согласованной по времени (такая функция ведет себя одинаково *сейчас* и *потом*), существенно упрощает понимание асинхронного кода.

Попросту говоря, чтобы все операции, когда бы они ни выполнялись (*сейчас* или *потом*), обрабатывались по единым правилам, следует перевести их все на «*потом*»: все операции становятся асинхронными.

Конечно, это примитивное решение на базе обратных вызовов далеко не идеально. Это всего лишь первый крошечный шаг к осознанию преимуществ мышления, ориентированного на будущие значения, когда вы не беспокоитесь о том, доступно в данный момент значение или нет.

Значение обещания

Обещания будут более подробно описаны позднее в этой главе, не беспокойтесь, если что-то покажется непонятным. А пока давайте посмотрим, как выразить пример `x + y` при помощи функций `Promise`:

```
function add(xPromise,yPromise) {
    // `Promise.all([ .. ])` получает массив обещаний
    // и возвращает новое обещание, ожидающее завершения
    // всех обещаний в массиве
    return Promise.all( [xPromise, yPromise] )
```

```
    // при разрешении этого обещания можно взять
    // полученные значения `X` и `Y` и просуммировать их.
    .then( function(values){
        // `values` - массив сообщений из
        // ранее разрешенных обещаний
        return values[0] + values[1];
    } );
}

// `fetchX()` и `fetchY()` возвращают обещания
// для соответствующих значений, которые могут быть готовы
// сейчас или позднее.
add( fetchX(), fetchY() )
// мы получаем обещание для суммы этих двух чисел.
// теперь сцепленный вызов `then(..)` используется для ожидания
// момента разрешения возвращенного обещания.
.then( function(sum){
    console.log( sum ); // так гораздо проще!
} );
```

В этом фрагменте существуют два уровня обещаний.

`fetchX()` и `fetchY()` вызываются напрямую, а возвращаемые ими значения (обещания!) передаются `add(..)`. Значения, представляемые этими обещаниями, могут быть готовы *сейчас* или *потом*, но каждое обещание нормализует поведение и приводит его к «общему знаменателю». Значения `X` и `Y` рассматриваются как *независимые от времени*. Это *будущие значения*.

Второй уровень — обещание, создаваемое и возвращаемое функцией `add(..)` (посредством вызова `Promise.all([ .. ])`). В коде мы ожидаем его вызовом `then(..)`. При завершении операции `add(..)` будущее значение `sum` готово, и его можно вывести в консоль. Вся логика ожидания будущих значений `X` и `Y` скрывается внутри `add(..)`.

Внутри `add(..)` вызов `Promise.all([ .. ])` создает обещание (которое ожидает разрешения promiseX и promiseY). Сцепленный вызов `.then(..)` создает другое обещание, которое немедленно разрешается в строке `return values[0] + values[1]` (с результатом сложения). Таким образом, вызов `then(..)`,

> присоединенный после вызова `add(..)` — в конце фрагмента, — на самом деле применяется ко второму возвращенному обещанию, а не к первому, созданному вызовом `Promise.all([ .. ])`. Кроме того, хотя ко второму вызову `then(..)` ничего не присоединяется, он тоже создает другое обещание, как если бы мы вдруг решили отслеживать/использовать его. Сцепление ожиданий намного подробнее рассматривается позднее в этой главе.

Как и в случае с заказами чизбургера, может оказаться, что разрешение обещания закончится отказом вместо выполнения. В отличие от выполненного обещания, где значение всегда генерируется программой, значение отказа — чаще называемое *причиной отказа* — может задаваться напрямую программной логикой или же определяться косвенно на основании исключения времени выполнения.

С обещаниями вызов `then(..)` может получать две функции: для успешного выполнения (как показано выше) и для отказа:

```
add( fetchX(), fetchY() )
.then(
    // обработка успешного выполнения
    function(sum) {
        console.log( sum );
    },
    // обработка отказа
    function(err) {
        console.error( err ); // облом!
    }
);
```

Если с получением `X` или `Y` что-то пойдет не так или при сложении произойдет какая-то непредвиденная ошибка, обещание, которое возвращает `add(..)`, отклоняется, а второй обратный вызов (обработчик ошибки), переданный `then(..)`, получит значение отказа из обещания.

Поскольку обещания инкапсулируют состояние, зависящее от времени (ожидание выполнения или отказа значения), для внеш-

него наблюдателя само обещание не зависит от времени; следовательно, обещания могут строиться (объединяться) предсказуемым образом независимо от хронометража или результата внутренней операции.

Более того, после того как обещание будет разрешено, оно остается в этом состоянии навсегда (то есть превращается в *неизменяемое значение*), после чего его можно проверять столько раз, сколько потребуется.

Поскольку обещание после разрешения становится неизменяемым извне, его можно безопасно передать любой стороне, зная, что оно может быть изменено (случайно или злонамеренно). Это особенно актуально в ситуации, в которой несколько сторон наблюдают за разрешением обещания. Одна сторона не может повлиять на способность другой стороны к наблюдению за разрешением обещания. Может показаться, что неизменяемость — чисто теоретический аспект, но в действительности это один из самых фундаментальных и важных аспектов архитектуры обещаний, который определенно не стоит упускать из виду.

Это одна из самых мощных и самых важных концепций, относящихся к обещаниям. Изрядно потрудившись, вы сможете вручную добиться того же эффекта, используя только уродливую конструкцию из обратных вызовов, но такая стратегия не особенно эффективна, потому что придется повторять ее снова и снова.

Обещания предоставляют механизм инкапсуляции и конструирования будущих значений, который может быть легко повторен по мере необходимости.

Событие завершения

Как вы уже видели, отдельное обещание ведет себя как будущее значение. Но существует и другой способ разрешения обещаний:

как механизма управления программной логикой для двух и более шагов в асинхронной задаче.

Допустим, имеется функция `foo(..)` для выполнения некоторой задачи. Подробности неизвестны, да они нас и не интересуют. Функция может завершить свою задачу немедленно или на ее выполнение может потребоваться некоторое время. Все, что нам нужно, — это узнать, когда `foo(..)` завершится, чтобы можно было перейти к следующей задаче. Другими словами, нам хотелось бы получить уведомление о завершении `foo(..)`, чтобы продолжить выполнение.

Как правило, если вам потребуется прослушивать уведомления в JavaScript, вы, скорее всего, будете мыслить понятиями событий. Таким образом, необходимость в уведомлении можно переформулировать как необходимость прослушивания *события завершения* (или *события продолжения*), выдаваемого `foo(..)`.

Какой термин вы будете использовать — событие завершения или событие продолжения — зависит от точки зрения. Что вас интересует в первую очередь: то, что происходит с `foo(..)`, или то, что происходит после завершения `foo(..)`? Обе точки зрения достаточно точны и полезны. Уведомление о событии сообщает о том, что вызов `foo(..)` завершился, но еще и о том, что можно переходить к следующему шагу. В самом деле, обратный вызов, который вы передаете для вызова с целью уведомления, сам по себе является тем, что мы прежде называли продолжением. Событие завершения в большей степени концентрируется на функции foo(..), которая представляет для нас интерес в данный момент, так что мы будем использовать термин «событие завершения» в оставшейся части книги.

С обратными вызовами уведомлением станет наш обратный вызов, который активизируется задачей `foo(..)`. Но с обещаниями эта связь разворачивается в другую сторону; предполагается возможность прослушивания события от `foo(..)`, и при получении уведомления мы можем продолжать действовать соответствующим образом.

Сначала рассмотрим псевдокод:

```
foo(x) {
    // начать выполнение операции,
    // которая может занять некоторое время
}
foo( 42 )
on (foo "completion") {
    // теперь можно переходить к следующему шагу!
}
on (foo "error") {
    // в `foo(..)` что-то пошло не так
}
```

Мы вызываем `foo(..)`, а затем назначаем двух слушателей событий, для «завершения» и для «ошибки» — двух возможных результатов вызова `foo(..)`. Фактически функция `foo(..)` даже не знает, что вызывающий код подписался на эти события; таким образом обеспечивается очень качественное *разделение обязанностей*.

К сожалению, такой код потребует волшебства со стороны среды JS, которого пока не существует (и скорее всего, будет не особо практичным). Более естественный способ выражения этого кода в JS выглядит так:

```
function foo(x) {
    // начать выполнение операции,
    // которая может занять некоторое время

    // вернуть объект уведомления для события `listener`

    return listener;
}

var evt = foo( 42 );

evt.on( "completion", function(){
    // теперь можно переходить к следующему шагу!
} );

evt.on( "failure", function(err){
    // в `foo(..)` что-то пошло не так
} );
```

`foo(..)` явно создает и возвращает объект подписки на событие, а вызывающий код получает его и использует для регистрации двух обработчиков событий.

Инверсия обычного кода, ориентированного на обратные вызовы, должна быть очевидной; это делается намеренно. Вместо передачи обратных вызовов `foo(..)` используется возвращение объекта события `evt`, получающего обратные вызовы.

Но если вспомнить главу 2, сами обратные вызовы представляют собой инверсию управления. Таким образом, инверсия паттерна «обратный вызов» в действительности представляет собой инверсию инверсии, или, проще говоря, восстановление управления в вызывающем коде, где бы мы и хотели его видеть.

Одно важное преимущество такой схемы заключается в том, что нескольким отдельным частям кода может быть предоставлена возможность прослушивания событий и они могут независимо получать уведомления о завершении `foo(..)` для выполнения следующих шагов:

```
var evt = foo( 42 );

// приказать `bar(..)` прослушивать событие завершения `foo(..)`
bar( evt );

// также приказать `baz(..)` прослушивать событие завершения
// `foo(..)`
baz( evt );
```

Восстановление управления улучшает разделение обязанностей, поскольку `bar(..)` и `baz(..)` могут не использоваться в вызове `foo(..)`. Аналогичным образом `foo(..)` может не знать и не беспокоиться о том, существуют ли `bar(..)` и `baz(..)` и ожидают ли они уведомления о завершении `foo(..)`.

По сути, объект `evt` представляет собой нейтральную стороннюю точку согласования между разделенными обязанностями.

События обещаний

Как вы, возможно, уже догадались, объект прослушивания событий `evt` является аналогом обещания.

При использовании решения на базе обещаний в предыдущем фрагменте `foo(..)` будет создавать и возвращать экземпляр `Promise`, а это обещание будет передаваться `bar(..)` и `baz(..)`.

События разрешения обещаний, которые мы прослушиваем, формально не являются событиями (хотя и ведут себя как события для нужных нам целей) и обычно им не присваиваются имена `"completion"` или `"error"`. Вместо этого вызов `then(..)` используется для регистрации события «then». Хотя, возможно, точнее будет сказать, что `then(..)` регистрирует события выполнения `"fulfillment"` и/или отказа `"rejection"`, несмотря на то что эти термины не используются в коде в явном виде.

Пример:

```
function foo(x) {
    // начать выполнение действий, которые могут занять много
    // времени.
    // сконструировать и вернуть обещание
    return new Promise( function(resolve,reject){
        // в будущем вызвать `resolve(..)` или `reject(..)` -
        // обратные вызовы для выполнения или отказа обещания.
    } );
}

var p = foo( 42 );

bar( p );

baz( p );
```

Паттерн в строке `new Promise( function(..){ .. } )` обычно называется раскрывающим конструктором (revealing constructor). Переданная функция выполняется немедленно (без асинхронной задержки, как обратные вызовы `then(..)`), и ей передаются два параметра, которым в нашем случае были присвоены имена `resolve` и `reject`. Это функции разрешения

обещания. `resolve(..)` обычно сигнализирует о выполнении, а `reject(..)` сигнализирует об отказе.

Вероятно, вы уже догадались, как может выглядеть внутренняя реализация `bar(..)` и `baz(..)`:

```
function bar(fooPromise) {
    // прослушивать завершение `foo(..)`
    fooPromise.then(
        function(){
            // функция `foo(..)` завершена, перейти
            // к задаче `bar(..)`
        },
        function(){
            // в `foo(..)` возникли какие-то проблемы
        }
    );
}
// то же для `baz(..)`
```

Разрешение обещания не обязательно требует отправки сообщения, как в ситуации с использованием обещаний как будущих значений. Например, оно может служить сигналом управления программной логикой, как в показанном фрагменте.

Другое возможное решение выглядит так:

```
function bar() {
    // функция `foo(..)` определенно завершилась,
    // выполнить задачу `bar(..)`
}

function oopsBar() {
    // в `foo(..)` возникли какие-то проблемы,
    // поэтому `bar(..)` не запускается
}

// то же для `baz()` и `oopsBaz()`

var p = foo( 42 );

p.then( bar, oopsBar );

p.then( baz, oopsBaz );
```

Если вы уже видели код на основе обещаний, возможно, вам захочется записать две последние строки кода в виде `p.then( .. ).then( .. )`, то есть с применением сцепления вместо `p.then(..); p.then(..)`. Но это будет совершенно другое поведение, так что будьте осторожны! Возможно, различия для вас сейчас не очевидны, но это другой асинхронный паттерн, который уже встречался вам: расщепление/ветвление. Не беспокойтесь! Мы еще вернемся к этой теме.

Вместо того чтобы передавать обещание `p` функциям `bar(..)` и `baz(..)`, мы используем обещание для управления тем, когда будут выполняться `bar(..)` и `baz(..)` (и будут ли вообще). Главное отличие связано с обработкой ошибок. В решении из первого фрагмента `bar(..)` вызывается независимо от того, завершилась функция `foo(..)` успехом или неудачей, и обеспечивает собственную логику восстановления, получив уведомление об отказе `foo(..)`. Разумеется, то же относится и к `baz(..)`.

Во втором фрагменте `bar(..)` вызывается только в случае успешного выполнения `foo(..)`; в противном случае вызывается `oopsBar(..)` (аналогично для `baz(..)`).

Ни один подход не является единственно правильным. В разных ситуациях предпочтительными могут оказаться разные решения.

В любом случае обещание `p`, поступающее от `foo(..)`, используется для управления тем, что должно произойти следующим.

Кроме того, тот факт, что оба фрагмента в итоге дважды вызывают `then(..)` для одного обещания `p`, демонстрирует то, о чем говорилось выше: обещания (после разрешения) сохраняют свой статус разрешения (выполнение или отказ) навсегда и впоследствии могут анализироваться столько раз, сколько потребуется.

Когда бы ни происходило разрешение `p`, следующий шаг всегда остается неизменным — и *сейчас*, и *потом*.

Утиная типизация с методом then()(thenable)

В мире обещаний важно с полной уверенностью знать, является ли некоторое значение «настоящим» обещанием или нет. Или проще говоря, обладает ли это значение поведением, которым должно обладать обещание?

С учетом того, что обещания конструируются в синтаксисе `new Promise(..)`, можно подумать, что для этого можно воспользоваться проверкой `p instanceof Promise`. Но к сожалению, по целому ряду причин этого недостаточно.

Прежде всего вы можете получить значение обещания от другого окна браузера (`iframe` и т. д.), которое будет иметь собственные обещания, отличные от обещаний текущего окна/фрейма, и такая проверка не сможет идентифицировать экземпляр обещания.

Более того, библиотека или фреймворк могут выбрать собственную реализацию обещаний и не использовать реализацию обещаний ES6. Собственно, вы можете прекрасно пользоваться обещаниями с помощью библиотек в старых браузерах, в которых обещаний вообще нет.

Когда мы будем обсуждать процессы разрешения обещаний позднее в этой главе, вам станет более очевидно, почему так важно иметь возможность распознать и ассимилировать значение, которое похоже на обещание, но «настоящим» обещанием не является. А пока просто поверьте мне на слово, что это одна из важнейших частей головоломки.

Поэтому было решено, что для распознавания обещаний (или чего-то, что ведет себя как обещание) будет правильно определить понятие «thenable» как любого объекта или функции, содержащих метод `then(..)`. Предполагается, что любое такое значение является «thenable» и соответствующим требованиям к обещаниям.

Общий термин для проверки типа, которая делает предположения относительно типа значения на основании его строения (то есть наличествующих свойств), называется *утиной типизацией* (см. книгу «Типы и грамматические конструкции» этой серии). Таким образом, проверка утиной типизации для «thenable» выглядит примерно так:

```
if (
    p !== null &&
    (
        typeof p === "object" ||
        typeof p === "function"
    ) &&
    typeof p.then === "function"
) {
    //"thenable"!
}
else {
    // не "thenable"
}
```

Фу, гадость! Даже если забыть о том факте, что реализация этой логики в разных местах выглядит уродливо, здесь происходит нечто более глубокое и тревожное.

Если вы попытаетесь выполнить обещание с любым объектом/функцией, который по случайности содержит функцию `then(..)`, но при этом не предполагалось, что он будет использоваться как обещание/«thenable», вам не повезло. Объект будет автоматически распознан как «thenable» и будет обрабатываться по особым правилам (см. далее в этой главе).

Это произойдет даже в том случае, если вы не сознаете, что значение содержит `then(..)`. Пример:

```
var o = { then: function(){} };

// связать `v` через `[[Prototype]]` с `o`
var v = Object.create( o );

v.someStuff = "cool";
```

```
v.otherStuff = "not so cool";

v.hasOwnProperty( "then" );     // false
```

v совершенно не напоминает обещание или«thenable». Это простой объект с несколькими свойствами. Возможно, вы просто хотели передать это значение как любой другой объект.

Но незаметно для вас v также связывается через [[Prototype]] с другим объектом o, который содержит метод then(..). Таким образом, проверки утиной типизации на «thenable» будут считать, что значение v является «thenable». Ой-ой!

Проблема даже не обязана быть настолько явной и прямолинейной:

```
Object.prototype.then = function(){};
Array.prototype.then = function(){};

var v1 = { hello: "world" };
var v2 = [ "Hello", "World" ];
```

И v1 и v2 будут считаться «thenable». Вы не можете управлять тем, что другой код случайно или намеренно добавит then(..) в Object.prototype, Array.prototype или любой другой из встроенных прототипов, или предсказать такую возможность. А если задается функция, которая не вызывает никакие из своих параметров как обратные вызовы, то любое обещание, разрешенное с таким значением, попросту зависнет навсегда!

Кажется маловероятным? Возможно.

Но помните, что до появления ES6 в сообществе было несколько хорошо известных библиотек, не связанных с обещаниями, и в этих библиотеках уже присутствовали методы с именем then(..). Одни из этих библиотек потом переименовали свои методы, чтобы избежать конфликтов имен. Другие за свою неспособность измениться были попросту переведены в нехорошую категорию «несовместимых с программированием на базе обещаний».

Принятое в стандарте решение о захвате ранее не зарезервированного — и вполне нормально выглядящего — имени свойства `then` означает, что никакое значение (и никакие из его делегатов) в прошлом, настоящем и будущем не может содержать функцию с именем `then(..)` (намеренно или случайно). В противном случае в системах с обещаниями это значение будет сочтено «thenable», что, скорее всего, приведет к появлению крайне трудноуловимых ошибок.

Мне не нравится, что мы пришли к «утиной типизации» с методом `then()` для распознавания обещаний. Были и другие варианты; то, что получилось у нас, смахивает на худший из возможных компромиссов. Впрочем, все не так ужасно. Утиная типизация с методом `then()` может быть полезной, как вы вскоре убедитесь. Просто помните: утиная типизация с методом `then()` может быть опасна, если она ошибочно идентифицирует как обещание нечто, что обещанием не является.

Доверие Promise

Мы рассмотрели две сильные аналогии, поясняющие разные аспекты того, что обещания могут сделать для асинхронного кода. Но если остановиться на этом, мы упустим самую важную характеристику, которую создает паттерн обещаний, — доверие.

Хотя аналогии будущих значений и событий завершения достаточно явно проявляются в паттернах кода, которые мы исследовали, будет не совсем очевидно, почему или как обещания работают на решение всех проблем инверсии управления, изложенных в разделе «Проблемы доверия» главы 2. Но если немного разобраться, мы обнаружим некоторые важные гарантии, которые восстанавливают доверие к асинхронному программированию, подорванное в главе 2!

Начнем с обзора проблем доверия, возникающих при программировании, основанном только на обратных вызовах. Если вы передаете обратный вызов функции `foo(..)`, эта функция может:

- Активизировать обратный вызов слишком рано.
- Активизировать обратный вызов слишком поздно (или не активизировать вообще).
- Активизировать обратный вызов слишком мало или слишком много раз.
- Не передать необходимые переменные среды/параметры.
- Поглотить любые ошибки/исключения, которые могут произойти.

Характеристики обещаний сознательно проектировались с расчетом на то, чтобы дать полезные, пригодные для повторного использования решения для всех этих проблем.

Слишком ранний обратный вызов

Прежде всего, эта проблема связана с возможными Залго-эффектами (см. главу 2): иногда задача может завершаться синхронно, иногда — асинхронно, что может привести к состоянию гонки.

Обещания по определению избавлены от этой проблемы, потому что даже немедленно выполняемое обещание (типа `new Promise(function(resolve){ resolve(42); })`) не может наблюдаться синхронно.

Иначе говоря, когда вы вызываете `then(..)` для обещания, даже если это обещание уже разрешено, переданный `then(..)` обратный вызов всегда будет вызываться асинхронно (см. раздел «Задания» главы 1).

Вставлять свои трюки с `setTimeout(..,0)` не нужно. Обещания автоматически предотвращают проблемы Залго.

Слишком поздний обратный вызов

По аналогии с предыдущим пунктом, зарегистрированные в `then(..)` обратные вызовы автоматически планируются при вы-

зове `resolve(..)` или `reject(..)` функциональностью создания обещания. Эти запланированные обратные вызовы прогнозируемым образом сработают в следующий асинхронный момент (см. раздел «Задания» главы 1).

Для синхронного наблюдения это невозможно, поэтому синхронная цепочка задач не может выполняться таким образом, чтобы фактически отложить обратный вызов, не позволяя ему произойти в ожидаемый момент. То есть при разрешении обещания все зарегистрированные обратные вызовы `then(..)` будут вызваны по порядку немедленно при следующей асинхронной возможности (также см. раздел «Задания» главы 1), и ничего, что происходит внутри одного из этих обратных вызовов, не сможет повлиять/отложить активизацию других обратных вызовов.

Пример:

```
p.then( function(){
    p.then( function(){
        console.log( "C" );
    } );
    console.log( "A" );
} );
p.then( function(){
    console.log( "B" );
} );
// A B C
```

Здесь `"C"` не может прервать и опередить `"B"` вследствие самого определения механизма работы обещаний.

Странности планирования обещаний

Однако важно заметить, что существует множество нюансов планирования, и относительный порядок обратных вызовов, отходящих от двух разных обещаний, не может быть спрогнозирован надежным образом.

Если оба обещания, `p1` и `p2`, уже разрешены, тогда вроде бы при выполнении `p1.then(..); p2.then(..)` обратный вызов(-ы) для

p1 должен быть вызван ранее обратного вызова(-ов) для p2. Однако существуют нетривиальные случаи, в которых это не так. Пример:

```
var p3 = new Promise( function(resolve,reject){
    resolve( "B" );
} );

var p1 = new Promise( function(resolve,reject){
    resolve( p3 );
} );

p2 = new Promise( function(resolve,reject){
    resolve( "A" );
} );
p1.then( function(v){
    console.log( v );
} );

p2.then( function(v){
    console.log( v );
} );

// A B  <-- не  B A как можно было бы ожидать
```

Этот пример будет подробнее рассмотрен позднее, но как вы видите, p1 разрешается не непосредственным значением, но другим обещанием p3, которое, в свою очередь, разрешается значением "B". Согласно определенному поведению происходит распаковка p3 в p1, но асинхронно, так что обратный вызов(-ы) p1 оказывается позади обратного вызова(-ов) p2 в асинхронной очереди заданий (см. раздел «Задания»).

Чтобы избежать подобных неочевидных проблем, вам никогда не следует полагаться на любое поведение, связанное с упорядочением/планированием обратных вызовов между обещаниями. Вообще говоря, хорошая практика программирования не рекомендует программировать так, чтобы зависеть от порядка нескольких обратных вызовов. Старайтесь по возможности обходиться без этого.

Обратный вызов вообще не вызывается

Это очень распространенная проблема, которая решается несколькими способами с обещаниями.

Во-первых, ничего (даже ошибка JS) не может помешать обещанию уведомить вас о его разрешении. Если вы регистрируете для обещания обратные вызовы как выполнения, так и отказа и обещание будет разрешено, один из двух обратных вызовов всегда будет вызван.

Конечно, если ваши обратные вызовы сами порождают ошибки JS, вы можете не получить ожидаемый результат, но обратный вызов все равно будет вызван. Позднее мы расскажем, как получить уведомление об ошибке в вашем обратном вызове, потому что даже они не будут поглощаться.

Но что, если само обещание не будет разрешено тем или иным образом? Даже для такой ситуации обещания предоставляют ответ, используя абстракцию более высокого уровня, которая называется *гонкой* (race):

```
// функция определения тайм-аута для обещания
function timeoutPromise(delay) {
    return new Promise( function(resolve,reject){
        setTimeout( function(){
            reject( "Timeout!" );
        }, delay );
    } );
}

// настройка тайм-аута для `foo()`
Promise.race( [
    foo(),                  // попытаться выполнить `foo()`
    timeoutPromise( 3000 )  // выделить 3 секунды
] )
.then(
    function(){
        // функция `foo(..)` выполнена вовремя!
```

```
    },
    function(err){
        // либо функция `foo()` столкнулась с отказом, либо
        // просто не завершилась вовремя; проанализировать
        // `err` для определения причины
    }
);
```

С этим паттерном тайм-аута обещаний необходимо учитывать ряд подробностей, но мы вернемся к ним позднее.

Очень важно то, что мы можем обеспечить выдачу сигнала о результате `foo()`, чтобы предотвратить зависание программы.

Слишком малое или слишком большое количество вызовов

По определению правильное количество активизаций обратного вызова — один. «Слишком мало» — нуль вызовов, что эквивалентно только что рассмотренному случаю «никогда».

Случай «слишком много» легко объясняется. Обещания определяются так, чтобы они могли разрешаться только один раз. Если по какой-то причине код создания обещания попытается многократно вызвать `resolve(..)` или `reject(..)` или попытается вызвать обе функции, обещание примет только первое разрешение и незаметно проигнорирует все последующие попытки.

Поскольку обещание может быть разрешено только один раз, все зарегистрированные обратные вызовы `then(..)` будут вызваны однократно (каждый).

Конечно, если зарегистрировать один обратный вызов более одного раза (например, `p.then(f); p.then(f);`), он будет активизирован столько раз, сколько он был зарегистрирован. Гарантия того, что функция отклика будет вызвана только один раз, не помешает вам сознательно наступить на грабли.

Отсутствие параметров/переменных среды

Обещания могут иметь не более одного результата разрешения (выполнение или отказ).

Без явного разрешения используется значение `undefined`, как это обычно бывает в JS. Но каким бы ни было значение, оно всегда будет передано всем зарегистрированным (и подходящим по выполнению/отказу) обратным вызовам, сейчас или в будущем.

Необходимо учитывать один важный момент: если вызвать `resolve(..)` или `reject(..)` с несколькими параметрами, все последующие параметры, кроме первого, будут проигнорированы. На первый взгляд это нарушает только что описанную гарантию, но на самом деле это не совсем так, потому что механизм обещаний используется некорректно. Другие недопустимые сценарии использования API (скажем, многократный вызов `resolve(..)`) тоже имеют аналогичную защиту, так что поведение обещаний здесь вполне последовательно (хотя малость и раздражает).

Если вы хотите передать несколько значений, необходимо заключить их в одно передаваемое значение (например, массив или объект). Что касается переменных среды, функции в JS всегда сохраняют замыкание области видимости, в которой они определены, и конечно, они сохранят доступ к любому окружающему состоянию. Конечно, это относится и к архитектуре, основанной только на обратных вызовах, так что сказанное не относится к преимуществам именно обещаний, — и все же это гарантия, на которую можно положиться.

Поглощение ошибок/исключений

На каком-то базовом уровне этот пункт является переформулировкой предыдущего пункта. Если обещание отклоняется с *причиной* (то есть сообщением об ошибке), это значение передается обратному вызову(-ам) отказа.

Тем не менее здесь происходит нечто более серьезное. Если в какой-то момент времени при создании обещания или при отслеживании его разрешения происходит исключение JS (например, `TypeError` или `ReferenceError`), то это исключение будет перехвачено, а в соответствующем обещании будет отказано.

Пример:

```
var p = new Promise( function(resolve,reject){
    foo.bar();  // `foo` не определено, ошибка!
    resolve( 42 );  // в эту точку управление не передается :(
} );

p.then(
    function fulfilled(){
        // в эту точку управление не передается :(
    },
    function rejected(err){
        // здесь `err` будет объектом исключения `TypeError`
        // из строки `foo.bar()`.
    }
);
```

Исключение JS, которое возникает в `foo.bar()`, становится отказом обещания, который можно перехватить и обработать.

Это важная подробность, потому что она фактически решает еще одну «проблему Залго»: ошибки могут создать синхронную реакцию, тогда как нормальные продолжения могут быть асинхронными. Обещания преобразуют даже исключения JS в асинхронное поведение, что существенно сокращает риск ситуаций гонки.

Но что произойдет, если обещание будет выполнено, но при наблюдении произойдет исключение JS (в зарегистрированном обратном вызове `then(..)`)? Даже такие исключения не теряются, но способ их обработки может показаться немного странным (пока вы не разберетесь глубже):

```
var p = new Promise( function(resolve,reject){
    resolve( 42 );
} );
```

```
p.then(
    function fulfilled(msg){
        foo.bar();
        console.log( msg ); // в эту точку управление
                            // не передается :(
    },
    function rejected(err){
        // и сюда тоже :(
    }
);
```

Постойте-ка, разве исключение из `foo.bar()` не поглощается? Нет, не поглощается. Здесь кроется другая, более глубокая проблема: мы не прослушиваем его. Вызов `p.then(..)` возвращает другое обещание, и именно *это* обещание получит отказ с исключением `TypeError`.

Почему оно просто не может вызвать обработчик ошибок, который мы определили? На первый взгляд такое поведение кажется логичным. Но оно может нарушить фундаментальный принцип, согласно которому обещания становятся неизменяемыми после разрешения. Обещание `p` уже выполнено со значением `42`, и оно не может позднее измениться и превратиться в отказ только потому, что при наблюдении за разрешением `p` произошла ошибка.

Помимо нарушения принципа, такое поведение может привести к хаосу — например, если для обещания `p` зарегистрировано несколько обратных вызовов `then(..)`, потому что одни из них будут вызваны, а другие нет, а причины происходящего будут совершенно неочевидны.

Обещания, заслуживающие доверия?

Для установления доверия на основании паттерна обещаний осталось изучить одну последнюю подробность.

Несомненно, вы заметили, что обещания вовсе не избавляются от обратных вызовов. Они просто меняют сторону, которой передается обратный вызов. Вместо того чтобы передавать обратный вызов `foo(..)`, мы получаем нечто (вероятно, полноценное обещание) от `foo(..)` и передаем обратный вызов этому «нечто».

Но почему такая схема заслуживает большего доверия, чем одни обратные вызовы? Как можно быть уверенными, что то, что вы получите обратно, действительно является обещанием, которому можно доверять? Разве это не тот же «карточный домик», где можно доверять только тому, чему мы уже доверяем?

Один из самых важных аспектов обещаний, о котором часто забывают, состоит в том, что у них есть решение и для этой проблемы. Во встроенную реализацию обещаний ES6 включен вызов `Promise.resolve(..)`. Если передать `Promise.resolve(..)` обычное, не являющееся обещанием, не-«thenable» значение, вы получите обещание, выполняемое с этим значением. В следующем случае обещания `p1` и `p2` будут обладать одинаковым поведением:

```
var p1 = new Promise( function(resolve,reject){
    resolve( 42 );
} );

var p2 = Promise.resolve( 42 );
```

Но если передать `Promise.resolve(..)` полноценное обещание, вы получите обратно то же обещание:

```
var p1 = Promise.resolve( 42 );

var p2 = Promise.resolve( p1 );

p1 === p2; // true
```

Что еще важнее, при передаче `Promise.resolve(..)` «thenable» значения, которое не является обещанием, функция попытается распаковать это значение, и распаковка будет продолжаться до тех пор, пока не будет извлечено конкретное итоговое значение, не похожее на обещание.

Помните, что мы уже говорили о «thenable»?

Пример:

```
var p = {
    then: function(cb) {
        cb( 42 );
    }
};

// работает, но это чистая случайность
.then(
    function fulfilled(val){
        console.log( val ); // 42
    },
    function rejected(err){
        // сюда управление не передается
    }
);
```

Это значение p является «thenable», но не является полноценным обещанием. К счастью, это значение разумно, как и большинство значений. Но что, если вы получите обратно нечто иное?

```
var p = {
    then: function(cb,errcb) {
        cb( 42 );
        errcb( "evil laugh" );
    }
};

p
.then(
    function fulfilled(val){
        console.log( val ); // 42
    },
    function rejected(err){
        // выполняться не должно
        console.log( err ); // злобный смех
    }
);
```

Это значение `p` является «thenable», но для обещания оно ведет себя далеко не так добропорядочно. Оно делает это намеренно? Или просто не знает, как должны работать обещания? Честно говоря, это не так важно. В любом случае ему вряд ли можно доверять как таковому.

Тем не менее любую из этих версий `p` можно передать `Promise.resolve(..)`, и мы получим нормализованный безопасный результат, на который рассчитываем:

```
Promise.resolve( p )
.then(
    function fulfilled(val){
        console.log( val ); // 42
    },
    function rejected(err){
        // выполняться не должно
    }
);
```

`Promise.resolve(..)` принимает любой «thenable» объект и распаковывает его в не-«thenable» значение. Но вы получите от `Promise.resolve(..)` настоящее, подлинное обещание, которому можно доверять. Если то, что вы передали, уже является полноценным обещанием, вы просто получаете его обратно, так что фильтрация через `Promise.resolve(..)` для получения доверия не имеет недостатков.

Допустим, мы вызываем функцию `foo(..)`, но при этом не уверены, можно ли ей доверять и будет ли ее возвращаемое значение добропорядочным обещанием. Известно, что оно, по крайней мере, «thenable». `Promise.resolve(..)` предоставит обертку для обещания, которой можно доверять:

```
// так поступать не надо:
foo( 42 )
.then( function(v){
    console.log( v );
} );
```

```
// вместо этого нужно действовать так:
Promise.resolve( foo( 42 ) )
.then( function(v){
    console.log( v );
} );
```

У заключения возвращаемого значения любой функции («thenable» или нет) в `Promise.resolve(..)` есть еще один полезный побочный эффект: это простой способ нормализации этого вызова в добропорядочную асинхронную задачу. Если `foo(42)` в одних случаях возвращает непосредственное значение, а в других — обещание, `Promise.resolve( foo(42) )` гарантирует, что результатом всегда будет обещание. Держитесь подальше от Залго, от этого ваш код станет намного лучше!

Формирование доверия

Хочется верить, что предыдущее пояснение полностью убедило вас в том, почему обещания повышают уровень доверия, и что еще важнее, почему это доверие настолько критично для построения надежных программных продуктов, не создающих проблем с сопровождением.

Можно ли написать асинхронный код на JS без доверия? Конечно, можно. Мы, разработчики JS, почти два десятилетия писали асинхронный код, не пользуясь ничем, кроме обратных вызовов.

Но стоит вам поставить под вопрос предсказуемость и надежность механизмов, на основе которых работает ваш код, и вы осознаете, что обратные вызовы — довольно сомнительная основа для доверия.

Обещания — паттерн, дополняющий обратные вызовы семантикой, заслуживающей доверия, что делает его поведение более разумным и более надежным. За счет восстановления инверсии управления обратных вызовов мы передаем управление системе, пользующейся доверием (обещаниям), которая проектировалась специально для того, чтобы сделать асинхронность более разумной.

Сцепление

Я уже пару раз на это намекал: обещания — не просто механизм для одношаговых операций в стиле «одно-потом-другое». Конечно, это структурный элемент кода, но как выясняется, несколько обещаний можно сцепить воедино для представления серии асинхронных шагов.

Ключ к работоспособности этой схемы — два аспекта поведения, присущих обещаниям:

- Каждый раз, когда вы вызываете `then(..)` для обещания, вызов создает и возвращает новое обещание, которое может использоваться для *сцепления*.
- Значение, возвращенное из обратного вызова выполнения для вызова `then(..)` (первый параметр), автоматически назначается как значение выполнения сцепленного обещания (от первой точки).

Сначала я покажу, что все это значит, а потом мы разберемся, как это помогает создавать асинхронные последовательности выполнения. Возьмем следующий фрагмент:

```
var p = Promise.resolve( 21 );

var p2 = p.then( function(v){
    console.log( v );   // 21

    // выполнение `p2` со значением `42`
    return v * 2;
} );

// сцепление `p2`
p2.then( function(v){
    console.log( v );   // 42
} );
```

Возвращая `v * 2` (то есть `42`), мы выполняем обещание `p2`, которое создает и возвращает первый вызов `then(..)`. При выполнении

вызова `then(..)` для `p2` он получает результат выполнения от команды `return v * 2`. Конечно, `p2.then(..)` создает еще одно обещание, которое можно было бы сохранить в переменной `p3`.

Однако создавать промежуточную переменную `p2` (а также `p3` и т. д.) не хочется. К счастью, вызовы можно просто объединить в цепочку:

```
var p = Promise.resolve( 21 );

p
.then( function(v){
    console.log( v );   // 21

    // выполнение сцепленного обещания со значением `42`
    return v * 2;
} )
// сцепленное обещание
.then( function(v){
    console.log( v );   // 42
} );
```

Теперь первый вызов `then(..)` становится первым шагом асинхронной последовательности, а второй вызов `then(..)` — ее вторым шагом. Все это может продолжаться так долго, как потребуется. Просто продолжайте присоединять вызовы к предыдущим `then(..)` с каждым автоматически созданным обещанием.

Но здесь кое-чего не хватает. А если мы хотим, чтобы шаг 2 ожидал шага 1 для выполнения некой асинхронной операции? Мы используем команду `return`, которая немедленно выполняет сцепленное обещание.

Чтобы создать последовательность обещаний, которая была бы полностью асинхронной на каждом этапе, необходимо вспомнить, как работает вызов `Promise.resolve(..)`, если ему передается обещание или «thenable» вместо итогового значения.

`Promise.resolve(..)` напрямую возвращает полученное полноценное обещание или распаковывает значение полученного

«thenable» — и продолжает действовать рекурсивно, в то время как распаковывает «thenables».

Аналогичная распаковка происходит при возвращении «thenable» или обещания из обработчика выполнения (или отказа). Пример:

```
var p = Promise.resolve( 21 );
p.then( function(v){
    console.log( v );   // 21

    // создать обещание и вернуть его
    return new Promise( function(resolve,reject){
        // выполнить со значением `42`
        resolve( v * 2 );
    } );
} )
.then( function(v){
    console.log( v );   // 42
} );
```

И хотя мы упаковали значение `42` в возвращаемое обещание, оно все равно распаковывается и становится результатом разрешения сцепленного обещания, так что второй вызов `then(..)` все равно получает `42`. Если мы введем асинхронность в это обещание-оберт-ку, все будет нормально работать без всяких изменений:

```
var p = Promise.resolve( 21 );

p.then( function(v){
    console.log( v );   // 21

    // создать обещание для возвращения
    return new Promise( function(resolve,reject){
        // ввести асинхронность!
        setTimeout( function(){
            // выполнить со значением `42`
            resolve( v * 2 );
        }, 100 );
    } );
} )
.then( function(v){
    // выполняется после 100-миллисекундной задержки
```

```
    // на предыдущем шаге
    console.log( v );   // 42
} );
```

Это невероятно мощный механизм! Теперь вы можете сконструировать последовательность из любого количества асинхронных шагов, и каждый шаг может создавать задержку следующего шага (или не создавать!) по мере необходимости.

Конечно, передача значения между шагами в данном примере не обязательна. Если не возвращается явное значение, неявно предполагается `undefined`, и обещания могут сцепляться так же, как прежде. Таким образом, каждое разрешение обещания всего лишь является сигналом для перехода к следующему шагу.

Продолжая пример сцепления, обобщим создание отложенного обещания (без сообщений о разрешении) в функцию, пригодную для повторного использования на разных шагах:

```
function delay(time) {
    return new Promise( function(resolve,reject){
        setTimeout( resolve, time );
    } );
}

delay( 100 ) // шаг 1
.then( function STEP2(){
    console.log( "step 2 (after 100ms)" );
    return delay( 200 );
} )
.then( function STEP3(){
    console.log( "step 3 (after another 200ms)" );
} )
.then( function STEP4(){
    console.log( "step 4 (next Job)" );
    return delay( 50 );
} )
.then( function STEP5(){
    console.log( "step 5 (after another 50ms)" );
} )
...
```

Вызов `delay(200)` создает обещание, которое будет выполнено через 200 мс. Оно возвращается обратным вызовом выполнения первого `then(..)`, в результате чего обещание второго вызова `then(..)` будет ожидать этого 200-миллисекундного обещания.

На техническом уровне в описанной ситуации задействованы два обещания: с 200-миллисекундной задержкой и сцепленное обещание, от которого ответвляется второй вызов `then(..)`. Но возможно, для лучшего понимания вам будет проще мысленно объединить эти два обещания, потому что механизм обещаний автоматически объединяет их состояния за вас. В этом отношении `return delay(200)` можно рассматривать как создание обещания, которое замещает ранее возвращенное сцепленное обещание.

Откровенно говоря, последовательность задержек без передачи сообщений вряд ли можно назвать очень полезным примером управления программной логикой посредством обещаний. Рассмотрим чуть более практичный пример.

Вместо таймеров рассмотрим создание запросов Ajax:

```
// предполагает существование функции `ajax( {url},
// {обратный_вызов} )`

// ajax с поддержкой обещаний
function request(url) {
    return new Promise( function(resolve,reject){
        // Обратный вызов `ajax(..)` должен быть функцией
        // `resolve(..)` нашего обещания
        ajax( url, resolve );
    } );
}
```

Сначала мы определяем функцию `request(..)`, которая конструирует обещание, представляющее завершение вызова `ajax(..)`:

```
request( "http://some.url.1/" )
.then( function(response1){
    return request( "http://some.url.2/?v=" + response1 );
```

```
} )
.then( function(response2){
    console.log( response2 );
} );
```

Разработчики часто сталкиваются с ситуациями, в которых они хотят управлять асинхронной программной логикой с поддержкой обещаний с использованием средств, которые сами по себе не поддерживают обещания (например, как функция `ajax(..)` из предыдущего примера, которая рассчитывает получить обратный вызов). И хотя встроенный механизм обещаний ES6 не решает эту проблему автоматически за нас, практически все библиотеки обещаний это делают. Процесс может называться «подъемом», «обещанизацией» (promisifying) или как-нибудь в этом роде. Мы вернемся к этой теме позднее.

Используя вызов `request(..)`, возвращающий обещание, мы создаем первое звено цепочки неявно, вызывая его с первым URL-адресом, и отходим от возвращенного обещания первым вызовом `then(..)`.

После возвращения вернется `response1` — это значение используется для конструирования второго URL-адреса и создания второго вызова `request(..)`. Возвращается второе обещание `request(..)`, чтобы третий шаг асинхронного потока выполнения ожидал завершения этого вызова Ajax. Наконец, мы выводим `response2` после возвращения.

Сконструированная нами цепочка обещаний не только обеспечивает управление программной логикой, выражающей многошаговую асинхронную последовательность, но и работает как канал сообщений для передачи сообщений от шага к шагу.

Что, если на одном из шагов цепочки обещаний что-то пойдет не так? Ошибки/исключения происходят на уровне обещаний; это означает, что такая ошибка может быть перехвачена в любой точке цепочки, и перехват действует как своего рода «сброс» цепочки к нормальному режиму работы в этой точке:

```
// шаг 1:
request( "http://some.url.1/" )

// шаг 2:
.then( function(response1){
    foo.bar(); // undefined, ошибка!

    // сюда управление не передается
    return request( "http://some.url.2/?v=" + response1 );
} )

// шаг 3:
.then(
    function fulfilled(response2){
        // сюда управление не передается
    },

    // обработчик отказа для перехвата ошибки
    function rejected(err){
        console.log( err );
        // ошибка `TypeError` из `foo.bar()`
        return 42;
    }
)
// шаг 4:
.then( function(msg){
    console.log( msg );     // 42
} );
```

Когда на шаге 2 происходит ошибка, обработчик отказа на шаге 3 перехватывает его. Возвращаемое значение от этого обработчика отказа (`42` в данном фрагменте), если оно есть, выполняет обещание для следующего шага (4), так что цепочка снова находится в состоянии выполнения.

Как упоминалось ранее, при возвращении обещания из обработчика выполнения оно распаковывается и может задержать следующий шаг. Это также относится к возвращению обещаний из обработчиков отказа, так что если бы команда return 42 на шаге 3 вместо этого возвращала обещание, это обещание могло бы отложить шаг 4. Выданное исключение внутри обработчика выполнения или отказа вызова `then(..)`

инициирует немедленный отказ следующего (сцепленного) обещания с этим исключением.

Если вы вызываете `then(..)` для обещания и передаете ему только обработчик выполнения, то подставляется предполагаемый обработчик отказа:

```
var p = new Promise( function(resolve,reject){
    reject( "Oops" );
} );

var p2 = p.then(
    function fulfilled(){
        // управление сюда не передается
    }
    // предполагаемый обработчик отказа, если он опущен
    // или передано другое значение, не являющееся функцией
    // function(err) {
    //     throw err;
    // }
);
```

Как видите, предполагаемый обработчик отказа просто заново инициирует ошибку, которая приводит к отказу `p2` (сцепленного обещания) с той же причиной ошибки. Фактически это позволяет ошибке продолжить распространение по цепочке обещаний до тех пор, пока не будет обнаружен явно определенный обработчик отказа.

Обработка ошибок с обещаниями будет более подробно рассмотрена позднее, потому что существуют другие нюансы, о которых следует помнить. Если в параметре обработчика выполнения `then(..)` не будет передана действительная функция, также будет подставлен обработчик по умолчанию.

```
var p = Promise.resolve( 42 );

p.then(
    // предполагаемый обработчик выполнения, если он опущен
```

```
    // или передано другое значение, не являющееся функцией
    // function(v) {
    //     return v;
    // }
    null,
    function rejected(err){
        // управление сюда не передается
    }
);
```

Как вы видите, обработчик выполнения по умолчанию просто передает полученное значение следующему шагу (обещание).

У паттерна `then(null,function(err){ .. })`, который обрабатывает только отказы, но пропускает выполнения, существует сокращенная запись в API: `catch(function(err){ .. })`. Конструкция `catch(..)` будет более полно рассмотрена в следующем разделе.

А теперь кратко рассмотрим внутренние аспекты поведения обещаний, которые делают возможным сцепленное управление последовательностью выполнения:

- Вызов `then(..)` для одного обещания автоматически создает новое обещание, которое будет возвращено в результате вызова.
- Внутри обработчиков выполнения/отказа, если вы возвращаете значение или выдается исключение, новое возвращаемое (сцепляемое) обещание разрешается соответствующим образом.
- Если обработчик выполнения или отказа возвращает обещание, то оно распаковывается, чтобы результат его разрешения стал результатом разрешения сцепленного обещания, возвращаемого текущим вызовом `then(..)`.

Хотя сцепление средств управления программной логикой полезно, правильнее будет рассматривать его как побочное преимущество способа объединения обещаний, а не как основное пред-

назначение. Как мы уже подробно обсуждали несколько раз, обещания нормализуют асинхронность и инкапсулируют состояние значения, зависимое от времени, и *именно это* обстоятельство позволяет объединять их в цепочки с пользой.

Очевидно, последовательные выразительные возможности цепочки («одно-потом-другое-потом-третье») являются значительным усовершенствованием по сравнению с мешаниной обратных вызовов из главы 2. Тем не менее и в этом случае остается достаточно большой объем шаблонного кода (`then(..)` и `function(){ .. }`). В главе 4 будет представлен куда более удобный паттерн для выражения последовательного управления программной логикой с использованием генераторов.

Терминология: разрешение, выполнение и отказ

С терминами «разрешение» (resolve), «выполнение» (fulfill) и «отказ» (reject) связана некоторая путаница, с которой необходимо разобраться, пока вы не слишком глубоко зашли в изучении обещаний. Для начала рассмотрим конструктор `Promise(..)`:

```
var p = new Promise( function(X,Y){
    // X() для выполнения
    // Y() для отказа
} );
```

Как вы видите, здесь предоставляются два обратных вызова (с именами `X` и `Y`). Первый *обычно* используется для пометки обещания как выполненного, а второй всегда помечает обещание как отклоненное. Но что означает «*обычно*» и что из этого следует в отношении имен этих параметров?

В конечном итоге это ваш код, и с точки зрения движка имена ничего не значат, `foo(..)` и `bar(..)` в равной степени функциональны. Но выбор слов влияет не только на то, что вы думаете о своем коде, но и на то, что о нем будут думать другие разработчики из вашей

команды. Если кто-то неправильно думает о тщательно организованном асинхронном коде, результат почти наверняка будет хуже, чем альтернативное решение с клубком обратных вызовов.

Так что выбор имен все же в некотором смысле важен.

Со вторым параметром определиться несложно. Почти во всей литературе используется имя `reject(..)`, а поскольку оно точно описывает (то единственное!), что делает этот параметр, это очень хороший вариант для имени. Я настоятельно рекомендую всегда использовать имя `reject(..)`.

Однако с первым параметром все не столь однозначно; в литературе по промисам ему часто присваивается имя `resolve(..)`. Разумеется, это слово связано с «*разрешением*» (resolution), а оно, в свою очередь, используется в литературе (включая эту книгу) для описания присваивания итогового значения/состояния обещания. Мы уже неоднократно использовали выражение «разрешение обещания» для обозначения выполнения или отклонения обещания.

Но если этот параметр используется именно для выполнения обещания, почему бы не назвать его `fulfill(..)` вместо `resolve(..)` для большей точности? Чтобы ответить на этот вопрос, давайте также взглянем на два метода API обещаний:

```
var fulfilledPr = Promise.resolve( 42 );

var rejectedPr = Promise.reject( "Oops" );
```

`Promise.resolve(..)` создает обещание, которое разрешается в предоставленное значение. В данном примере `42` — нормальное, не являющееся обещанием, не-«thenable» значение, так что для значения `42` создается выполненное обещание `fulfilledPr`. `Promise.reject("Oops")` создает отклоненное обещание `rejectedPr` с причиной `"Oops"`.

А теперь покажем, почему слово «разрешение» (как в `Promise.resolve(..)`) однозначно и более точно при явном использовании

в контексте, который может привести как к выполнению, так и к отказу:

```
var rejectedTh = {
    then: function(resolved,rejected) {
        rejected( "Oops" );
    }
};

var rejectedPr = Promise.resolve( rejectedTh );
```

Как упоминалось ранее в этой главе, `Promise.resolve(..)` вернет полученное полноценное обещание напрямую или распакует полученный «thenable». Если в результате распаковки «thenable» будет обнаружено состояние отказа, то обещание, возвращенное от `Promise.resolve(..)`, находится в том же состоянии отказа.

Итак, `Promise.resolve(..)` хорошее, точное имя для метода API, потому что оно может привести либо к выполнению, либо к отказу.

В результате распаковки первого параметра обратного вызова конструктора `Promise(..)` будет получен либо «thenable» (по аналогии с `Promise.resolve(..)`), либо полноценное обещание:

```
var rejectedPr = new Promise( function(resolve,reject){
    // разрешить это обещание обещанием отказа
    resolve( Promise.reject( "Oops" ) );
} );

rejectedPr.then(
    function fulfilled(){
        // управление сюда не передается
    },
    function rejected(err){
        console.log( err ); // "Oops"
    }
);
```

Думаю, вы согласитесь с тем, что `resolve(..)` — подходящее имя для первого параметра обратного вызова конструктора `Promise(..)`.

Ранее упоминавшийся вызов `reject(..)` не выполняет распаковку, как это делает `resolve(..)`. Если передать `reject(..)` обещание/«thenable» значение, то это значение в неизменном виде будет задано как причина отказа. Последующий обработчик отказа получит фактическое обещание/«thenable», которое вы передали `reject(..)`, а не нижележащее непосредственное значение.

А теперь обратимся к обратным вызовам, передаваемым `then(..)`. Какие имена им должны быть присвоены (в литературе и в коде)? Я рекомендую использовать имена `fulfilled(..)` и `rejected(..)`:

```
function fulfilled(msg) {
    console.log( msg );
}

function rejected(err) {
    console.error( err );
}

p.then(
    fulfilled,
    rejected
);
```

В случае первого параметра `then(..)` речь всегда однозначно идет о случае выполнения, так что двойственность термина «разрешение» здесь не нужна. А заодно отметим, что в спецификации ES6 для пометки этих двух обратных вызовов используются имена `onFulfilled(..)` и `onRejected(..)`, поэтому эти термины можно признать точными.

Обработка ошибок

Мы уже рассмотрели несколько примеров того, как отказы обещаний — либо намеренные, посредством вызова `reject(..)`, либо случайные — из-за исключений JS — обеспечивают более разумную обработку ошибок в асинхронном программировании. А те-

перь вернемся назад и проясним подробности, о которых тогда ничего не было сказано.

Самая естественная форма обработки ошибок для большинства разработчиков — синхронная конструкция `try..catch`. К сожалению, она бывает только синхронной, так что в паттернах асинхронного программирования она бесполезна:

```
function foo() {
    setTimeout( function(){
        baz.bar();
    }, 100 );
}

try {
    foo();
    // позднее выдает глобальную ошибку из `baz.bar()`
}
catch (err) {
    // управление сюда не передается
}
```

Безусловно, конструкция `try..catch` была бы удобна, но она не работает для асинхронных операций — во всяком случае, без дополнительной поддержки со стороны среды, к которой мы вернемся при рассмотрении генераторов в главе 4.

В обратных вызовах были выработаны некоторые стандартные паттерны обработки ошибок, прежде всего стиль обратных вызовов «ошибка на первом месте»:

```
function foo(cb) {
    setTimeout( function(){
        try {
            var x = baz.bar();
            cb( null, x ); // успех!
        }
        catch (err) {
            cb( err );
        }
    }, 100 );
```

```
}

foo( function(err,val){
    if (err) {
        console.error( err ); // неудача :(
    }
    else {
        console.log( val );
    }
} );
```

Конструкция `try..catch` работает только при условии, что вызов `baz.bar()` немедленно и синхронно завершится успехом или неудачей. Если сама функция `baz.bar()` является асинхронно завершаемой, то никакие асинхронные ошибки в ней перехватываться не будут.

Обратный вызов, передаваемый `foo(..)`, ожидает получить сигнал об ошибке в зарезервированном первом параметре `err`. Если он присутствует, делается вывод об ошибке, если нет — предполагается успех.

Такой механизм обработки ошибок технически может поддерживать асинхронность, но она плохо работает в композиции. Множество уровней с обратными вызовами «ошибка на первом месте», сплетенные воедино с этими повсеместными проверками `if`, неизбежно повышают риск «кошмара обратных вызовов» (см. главу 2).

Итак, мы возвращаемся к обработке ошибок в обещаниях, с передачей `then(..)` обработчика отказа. Обещания не используют популярный стиль «ошибка на первом месте», а вместо этого используют стиль *расщепления обратных вызовов*: один обратный вызов предназначен для выполнения, а другой — для отказа:

```
var p = Promise.reject( "Oops" );

p.then(
    function fulfilled(){
```

```
        // управление сюда не передается
    },
    function rejected(err){
        console.log( err ); // "Oops"
    }
);
```

Хотя этот паттерн обработки ошибок на первый взгляд выглядит вполне логично, некоторые нюансы обработки ошибок обещаний часто бывает довольно трудно понять в полной мере.

Пример:

```
var p = Promise.resolve( 42 );

p.then(
    function fulfilled(msg){
        // у чисел нет строковых функций, поэтому
        // произойдет ошибка
        console.log( msg.toLowerCase() );
    },
    function rejected(err){
        // управление сюда не передается
    }
);
```

Если вызов `msg.toLowerCase()` выдаст совершенно законную ошибку (обязательно выдаст!), то почему наш обработчик ошибок не получает уведомления? Как объяснялось ранее, это происходит из-за того, что обработчик ошибок предназначен для обещания `p`, которое уже было выполнено со значением `42`. Обещание `p` неизменяемо, так что единственным обещанием, которое может быть уведомлено об ошибке, является обещание, возвращаемое из `p.then(..)`, которое в данном случае не перехватывается.

Все это должно создать четкую картину того, почему обработка ошибок с обещаниями чревата ошибками (простите за каламбур). Она слишком часто приводит к поглощению ошибок, а это редко бывает желательно.

Если вы используете Promise API неправильно и произойдет ошибка, препятствующая правильному конструированию обещаний, то результатом будет немедленно выданное исключение, а не отклоненное обещание. Несколько примеров некорректного использования, приводящего к ошибкам конструирования обещаний: `new Promise(null)`, `Promise.all()`, `Promise.race(42)` и т. д. Вы не получите отклоненное обещание, если не будете правильно использовать Promise API для его конструирования!

Бездна отчаяния

Джефф Этвуд (Jeff Atwood) несколько лет назад упомянул, что языки программирования часто строятся таким образом, что по умолчанию разработчики падают в «бездну отчаяния», где все случайности наказываются, а разработчик должен прикладывать массу усилий, чтобы сделать все правильно. Он предложил вместо этого создать «бездну успеха», где по умолчанию вы попадаете к ожидаемому (успешному) действию, а чтобы у вас что-то не получилось, придется основательно постараться.

Безусловно, обработка ошибок обещаний построена по принципу «бездны отчаяния». По умолчанию она предполагает, что все ошибки должны поглощаться состоянием обещания, и если вы забудете наблюдать за этим состоянием, ошибка незаметно пропадает/гибнет в неизвестности, что обычно приводит к отчаянию.

Чтобы ошибки не терялись из-за забытых/отброшенных обещаний, некоторые разработчики считают, что цепочки обещаний лучше всего завершать вызовом `catch(..)`, например:

```
var p = Promise.resolve( 42 );

p.then(
    function fulfilled(msg){
        // у чисел нет строковых функций,
        // поэтому произойдет ошибка
        console.log( msg.toLowerCase() );
```

```
    }
)
.catch( handleErrors );
```

Так как мы не передали обработчик отказа `then(..)`, был подставлен обработчик по умолчанию, который просто распространяет ошибку до следующего обещания в цепочке. Соответственно, как ошибки, приходящие в `p`, так и ошибки, приходящие *после* `p` в его разрешение (как в случае с `msg.toLowerCase()`), отфильтровываются в завершающий вызов `handleErrors(..)`.

Так значит, проблема решена? Не торопитесь!

Что произойдет, если в самой функции `handleErrors(..)` тоже произойдет ошибка? Кто перехватит ее? Существует еще одно необработанное обещание: то, которое возвращает `catch(..)`. Мы не перехватываем его и не регистрируем для него обработчик отказа.

Нельзя просто прицепить в конец цепочки еще один вызов `catch(..)`, потому что в нем тоже может произойти ошибка. На последнем шаге любой цепочки обещаний, каким бы он ни был, всегда существует опасность (пусть и убывающая) остаться с необработанной ошибкой внутри ненаблюдаемого обещания. Порочный круг?

Обработка неперехваченных ошибок

Полностью решить такую проблему будет непросто. Существуют и другие (по мнению многих — лучшие) возможные решения.

В некоторые библиотеки обещаний были добавлены методы для регистрации некоего «глобального обработчика необработанных отказов», который вызывался бы вместо глобально инициируемой ошибки. Но чтобы идентифицировать ошибку как неперехваченную, они устанавливают таймер произвольной продолжительности (допустим, 3 секунды) от момента отказа. Если обещание отклонено, но до срабатывания таймера обработчик ошибок так и не был зарегистрирован, то делается вывод о том, что вы уже не

зарегистрируете обработчик, так что ошибка считается необработанной.

На практике такой подход хорошо работает для многих библиотек, так как большинство паттернов использования обычно не требует значительной задержки между отклонением обещания и отслеживанием этого отклонения. Но этот паттерн создает определенный риск, потому что интервал в 3 секунды выбран произвольно (пусть и эмпирическим путем), а также потому, что в некоторых ситуациях нужно, чтобы обещание пребывало в состоянии отказа в течение неопределенного периода времени, а вы не хотите, чтобы ваш обработчик вообще реагировал на все эти ложные положительные срабатывания (не перехваченные и еще не обработанные ошибки).

Другая, более распространенная рекомендация — добавлять ко всем обещаниям метод `done(..)`, который фактически помечает цепочку обещаний как завершенную. `done(..)` не создает и не возвращает обещание, так что обратные вызовы, переданные `done(..)`, очевидно не подключаются для передачи информации о проблемах к сцепленному обещанию, которое не существует.

Что же происходит вместо этого? То, чего можно обычно ожидать при возникновении неперехваченной ошибки: любое исключение внутри обработчика отказа `done(..)` выдается как глобальная неперехваченная ошибка (в консоли разработчика):

```
var p = Promise.resolve( 42 );

p.then(
    function fulfilled(msg){
        // у чисел нет строковых функций, поэтому
        // произойдет ошибка
        console.log( msg.toLowerCase() );
    }
)
.done( null, handleErrors );

// если `handleErrors(..)` породит собственное исключение,
// оно будет глобально выдано здесь
```

Такое решение выглядит более привлекательно, чем бесконечная цепочка произвольных тайм-аутов. Но самая серьезная проблема заключается в том, что оно не является частью стандарта ES6, так что как бы хорошо оно ни выглядело, в лучшем случае ему очень далеко до положения надежного и универсального.

Безвыходное положение? Не совсем.

У браузеров есть уникальная возможность, которой нет у нашего кода: они могут отследить и точно узнать, когда любой объект будет выброшен и уничтожен сборщиком мусора. Таким образом браузеры могут отслеживать объекты обещаний, и каждый раз, когда уничтожаемый сборщиком мусора объект содержит отказ, браузер точно знает, что это была законная неперехваченная ошибка, и с полной уверенностью выводит информацию в консоль разработчика.

На момент написания книги и в Chrome, и в Firefox делались первые попытки функциональности неперехваченных отказов, но их поддержка в лучшем случае неполна.

Но если обещание не уничтожается сборщиком мусора — а это очень легко может произойти случайно при применении разных паттернов программирования, — отслеживание уборки мусора в браузере не поможет вам узнать о незаметно отклоненном обещании, которое продолжает где-то существовать.

Есть ли другие варианты? Да.

Бездна успеха

В этом разделе приведены чисто теоретические рассуждения о том, как поведение обещаний может когда-нибудь изменяться. Я считаю, оно будет намного лучше того, что мы имеем сейчас. И думаю, что это изменение будет возможно даже после ES6, потому что это приведет к нарушению веб-совместимости с обе-

щаниями ES6. Более того, если действовать осторожно и внимательно, для этой функциональности можно написать полифил. Итак:

- Обещания могут по умолчанию сообщать о любом отказе (в консоль разработчика) на следующем задании или тике цикла событий, если в этот конкретный момент для обещания не был зарегистрирован никакой обработчик событий.
- Если вы хотите, чтобы отклоненное обещание оставалось в своем состоянии отказа неопределенное время перед наблюдением, вы вызываете `defer()` для подавления автоматического сообщения об ошибках для этого обещания.

Если обещание отклоняется, по умолчанию информация об этом факте выводится в консоль разработчика (вместо того, чтобы игнорироваться по умолчанию). Вы можете выйти из этого режима либо неявно (зарегистрировав обработчик ошибки перед отказом), либо явно (вызовом `defer()`). В любом случае ложные положительные срабатывания находятся под *вашим* контролем.

Пример:

```
var p = Promise.reject( "Oops" ).defer();

// `foo(..)` с поддержкой обещаний
foo( 42 )
.then(
    function fulfilled(){
        return p;
    },
    function rejected(err){
        // обработать ошибку `foo(..)`
    }
);
...
```

При создании `p` мы знаем, что до использования/наблюдения за отказом пройдет некоторое время, поэтому мы вызываем `defer()` — отсюда отсутствие глобальных оповещений. `defer()` просто возвращает то же обещание для сцепления.

К обещанию, возвращенному из `foo(..)`, обработчик ошибки присоединяется немедленно, поэтому для него глобальное оповещение тоже не происходит.

Но обещание, возвращаемое из вызова `then(..)`, не имеет `defer()` или присоединенного обработчика ошибок, поэтому если для него выдается отказ (в одном из обработчиков разрешения), информация будет выведена в консоль разработчика в виде неперехваченной ошибки.

Такая архитектура является примером «бездны успеха». По умолчанию все ошибки либо обрабатываются, либо о них выводится информация — то, чего ожидают почти все разработчики почти во всех случаях. Вы должны либо зарегистрировать обработчик, либо намеренно отказаться и указать, что вы хотите отложить обработку ошибок *на потом*; фактически вы принимаете на себя лишнюю ответственность именно в этом конкретном случае.

Единственная реальная опасность в этом подходе возникает тогда, когда вы вызываете для обещания `defer()`, но не отслеживаете/не обрабатываете его отказ.

Но чтобы попасть в эту «бездну отчаяния», вы должны намеренно вызвать `defer()` (по умолчанию вы находитесь в «бездне успеха»). Вряд ли мы можем еще что-то сделать для того, чтобы спасти вас от ваших собственных ошибок.

Я думаю, что для обработки ошибок еще есть надежда (после ES6). Надеюсь, влиятельные фигуры смогут переосмыслить ситуацию и рассмотрят эту альтернативу. А пока вы можете реализовать ее самостоятельно (творческое упражнение для читателя!) или воспользоваться более совершенной библиотекой обещаний, которая сделает это за вас!

Именно эта модель обработки ошибок/оповещений реализована в моей библиотеке абстракции обещаний asynquence, описанной в приложении А этой книги.

Паттерны обещаний

Этот паттерн уже неявно встречался нам с цепочками обещаний («одно-потом-другое-потом-третье»), но существует много вариаций на тему асинхронных паттернов, которые могут строиться как абстракции поверх обещаний. Такие паттерны упрощают выражение асинхронной программной логики, что помогает сделать этот код более логичным и простым в сопровождении — даже в самых сложных частях программы.

Два таких паттерна напрямую запрограммированы во встроенной реализации `Promise` в ES6, так что они достаются вам автоматически для использования в качестве структурных элементов в других паттернах.

Promise.all([..])

В асинхронной последовательности (цепочка обещаний) в любой момент времени координируется только одна асинхронная задача — шаг 2 выполняется строго после шага 1, а шаг 3 выполняется строго после шага 2. Но как насчет двух и более «параллельно» выполняемых шагов?

В терминологии классического программирования *шлюзом* (gate) называется механизм, который ожидает завершения двух и более параллельных задач. Неважно, в каком порядке они будут завершаться, важно только то, что все они должны быть завершены, чтобы шлюз открылся и пропустил поток команд.

В API обещаний этот паттерн называется `all([ .. ])`.

Допустим, вы хотите выдать два запроса Ajax одновременно и дождаться завершения обоих, независимо от их порядка, прежде чем выдавать третий запрос Ajax. Пример:

```
// `request(..)` - функция Ajax с поддержкой обещаний
// вроде той, что мы определили ранее в этой главе
```

```
var p1 = request( "http://some.url.1/" );
var p2 = request( "http://some.url.2/" );

Promise.all( [p1,p2] )
.then( function(msgs){
    // обе переменные `p1` и `p2` выполняются
    // и передают свои сообщения
    return request(
        "http://some.url.3/?v=" + msgs.join(",")
    );
} )
.then( function(msg){
    console.log( msg );
} );
```

`Promise.all([ .. ])` ожидает получить один аргумент (массив), в общем случае состоящий из экземпляров обещаний. Обещание, возвращенное при вызове `Promise.all([ .. ])`, получит сообщение о выполнении (`msgs` в этом фрагменте), которое представляет собой массив всех сообщений о выполнении из переданных обещаний в порядке их задания (независимо от порядка выполнения).

С технической точки зрения массив значений, переданный `Promise.all([ .. ])`, может включать обещания, «thenable» и даже непосредственные значения. Каждое значение в списке фактически передается через `Promise.resolve(..)`, чтобы убедиться в том, что это полноценное обещание, пригодное для ожидания, так что непосредственное значение будет нормализовано в обещание для этого значения. Если массив пуст, то главное обещание немедленно выполняется.

Главное обещание, возвращаемое `Promise.all([ .. ])`, выполняется тогда и только тогда, когда выполнены все его составляющие обещания. Если хотя бы с одним из этих обещаний произойдет отказ, главное обещание `Promise.all([ .. ])` немедленно отклоняется с потерей всех результатов от любых других обещаний.

Не забывайте всегда присоединять обработчик отказа/ошибки для каждого обещания — даже для того (и особенно для того!), которое было получено от `Promise.all([ .. ])`.

Promise.race([..])

Хотя `Promise.all([ .. ])` координирует несколько параллельных обещаний и предполагает, что все они необходимы для выполнения, иногда бывает нужно отреагировать только на «первое обещание, пересекающее финишную черту», а все остальные обещания просто игнорируются.

Этот паттерн традиционно называется *защелкой* (latch), но в области обещаний он называется *гонкой* (race).

Хотя метафора «только первого обещания, пересекающего финишную черту», хорошо подходит для этого поведения, к сожалению, термин «гонка» в какой-то степени перегружен разными смыслами; обычно состоянием гонки считаются ошибки в программах (см. главу 1). Не путайте `Promise.race([..])` с состоянием гонки.

`Promise.race([ .. ])` также ожидает получить один аргумент-массив, содержащий одно или несколько обещаний, «thenable» или непосредственных значений. Гонка с непосредственными значениями не имеет особого практического смысла, потому что первое указанное значение очевидным образом выиграет — как соревнования по бегу, на котором один участник начинает на финишной черте!

По аналогии с `Promise.all([ .. ])`, `Promise.race([ .. ])` выполняется в том случае (и тогда), когда разрешение любого обещания приведет к выполнению, а отклоняется — в том случае (и тогда), когда разрешение любого обещания приведет к отказу.

Для гонки нужен хотя бы один «бегун», так что если вы передадите пустой массив, вместо того, чтобы разрешиться немедленно, главное обещание `race([..])` никогда не разрешится. Сюрприз! В ES6 следовало бы указать, что такой вызов должен либо выполняться, либо отклоняться, либо просто выдавать ту или иную синхронную ошибку. К сожалению, из-за появления первых библиотек обещаний до появления под-

держки обещаний в ES6 эту ловушку пришлось оставить. Будьте осторожны и никогда не передавайте пустой массив!

А теперь вернемся к предыдущему параллельному примеру Ajax, но в контексте гонки между `p1` и `p2`:

```
// `request(..)` - функция Ajax с поддержкой обещаний
// вроде той, что мы определили ранее в этой главе

var p1 = request( "http://some.url.1/" );
var p2 = request( "http://some.url.2/" );

Promise.race( [p1,p2] )
.then( function(msg){
    // либо `p1`, либо `p2` выиграет гонку
    return request(
        "http://some.url.3/?v=" + msg
    );
} )
.then( function(msg){
    console.log( msg );
} );
```

Поскольку победит только одно обещание, значение выполнения представляет собой отдельное сообщение, а не массив, как было для `Promise.all([ .. ])`.

Тайм-аут и race(..)

Этот пример уже приводился ранее. Он показывает, как `Promise.race([ .. ])` может использоваться для выражения паттерна «тайм-аут» с обещаниями:

```
// `foo()` - функция с поддержкой обещаний

// Функция `timeoutPromise(..)`, определенная ранее,
// возвращает обещание, которое разрешается отказом
// после заданной задержки.
// настройка тайм-аута для `foo()`
Promise.race( [
    foo(),                  // попытаться выполнить `foo()`
```

```
    timeoutPromise( 3000 )  // выделить 3 секунды
] )
.then(
    function(){
        // функция `foo(..)` выполнена вовремя!
    },
    function(err){
        // либо функция `foo()` столкнулась с отказом, либо
        // просто не завершилась вовремя; проанализировать
        // `err` для определения причины
    }
);
```

Эта разновидность тайм-аута хорошо работает в большинстве случаев. Тем не менее необходимо учитывать ряд нюансов; откровенно говоря, они в равной степени применимы как к `Promise.race([ .. ])`, так и к `Promise.all([ .. ])`.

finally

Ключевой вопрос, который необходимо задать: «Что происходит с обещаниями, которые теряются/игнорируются»? Этот вопрос важен не с точки зрения эффективности (обещания в конечном итоге станут доступными для уничтожения сборщиком мусора), но с точки зрения поведения (побочные эффекты и т. д.). Обещания не могут отменяться и не должны, потому что это уничтожило бы гарантию внешней неизменяемости, рассмотренную в разделе «Неотменяемость обещаний» этой главы, так что они могут только молча игнорироваться. Но что, если `foo()` в предыдущем примере резервирует некоторый ресурс для использования, но тайм-аут сработает первым, в результате чего обещание будет проигнорировано? Существует ли в этом паттерне возможность активного освобождения зарезервированного ресурса после тайм-аута или иного способа отмены возможных побочных эффектов? Что, если вам нужно только сохранить в журнале запись о тайм-ауте `foo()`?

Некоторые разработчики считают, что обещаниям необходима регистрация обратного вызова `finally(..)`, который всегда вы-

зывается при разрешении обещания и позволяет задать любые завершающие действия, которые могут потребоваться. На данный момент в спецификации такая возможность отсутствует, но она может появиться в ES7+. Подождем — увидим.

Это может выглядеть так:

```
var p = Promise.resolve( 42 );

p.then( something )
.finally( cleanup )
.then( another )
.finally( cleanup );
```

В разных библиотеках обещаний `finally(..)` создает и возвращает новое обещание (чтобы цепочка продолжалась). Если бы функция `cleanup(..)` возвращала обещание, оно было бы добавлено к цепочке, а это означает, что вы все равно столкнетесь с проблемами необработанных отказов, которые обсуждались ранее.

А пока мы можем создать статическую вспомогательную функцию, которая позволяет наблюдать за разрешением обещания (без вмешательства):

```
// защитная проверка, безопасная для полифилов
if (!Promise.observe) {
    Promise.observe = function(pr,cb) {
        // наблюдение за разрешением `pr`
        pr.then(
            function fulfilled (msg){
                // асинхронное планирование обратного вызова
                // (как задания)
                Promise.resolve( msg ).then( cb );
            },
            function rejected(err){
                // асинхронное планирование обратного вызова
                // (как задания)
                Promise.resolve( err ).then( cb );
            }
```

```
        );

        // возвращение исходного обещания
        return pr;
    };
}
```

Вот как она бы использовалась в приведенном выше примере с тайм-аутом:

```
Promise.race( [
    Promise.observe(
        foo(),                  // попытаться выполнить `foo()`
        function cleanup(msg){
            // прибрать за функцией `foo()`, даже если она
            // не завершилась до истечения тайм-аута
        }
    ),
    timeoutPromise( 3000 )  // выделить 3 секунды
] )
```

Вспомогательная функция `Promise.observe(..)` всего лишь демонстрирует, как можно наблюдать за завершением обещаний, не вмешиваясь в их обработку. Другие библиотеки обещаний предоставляют свои решения. Независимо от того, как вы это будете делать, скорее всего, в каких-то местах вы захотите удостовериться в том, что ваши обещания не будут просто игнорироваться по какой-то случайности.

Вариации на тему all([..]) и race([..])

Хотя у обещаний ES6 есть встроенные функции `Promise.all([ .. ])` и `Promise.race([ .. ])`, есть несколько других часто встречающихся паттернов с вариациями этой семантики:

- `none([ .. ])` — аналогичен `all([ .. ])`, но выполнения и отказы меняются местами. Все обещания должны быть отклонены — отказы становятся значениями выполнения, и наоборот.

- `any([ .. ])` — аналогичен `all([ .. ])`, но он игнорирует любые отказы, так что будет достаточно одного выполнения вместо *всех*.
- `first([ .. ])` — аналогичен гонке с `any([ .. ])`; это означает, что он игнорирует любые отказы и выполнения после первого выполнения обещания.
- `last([ .. ])` — аналогичен `first([ .. ])`, но побеждает только последнее выполнение.

Некоторые библиотеки, абстрагирующие обещания, предоставляют эти версии, но вы можете определить их самостоятельно с использованием механик обещаний, `race([ .. ])` и `all([ .. ])`.

Например, определение `first([ .. ])` могло бы выглядеть так:

```
// защитная проверка, безопасная для полифилов
if (!Promise.first) {
    Promise.first = function(prs) {
        return new Promise( function(resolve,reject){
            // перебрать все обещания
            prs.forEach( function(pr){
                // нормализовать значение
                Promise.resolve( pr )
                // обещание, которое выполняется первым,
                // побеждает и может разрешить главное
                // обещание
                .then( resolve );
            } );
        } );
    };
}
```

Эта реализация `first(..)` не выдает отказа в случае отказа всех ее обещаний; она просто зависает, как и `Promise.race([])`. При желании вы можете добавить дополнительную логику, чтобы отслеживать отказы обещаний, и при получении всех отказов вызвать `reject()` для главного обещания. Оставим эту возможность читателям в качестве самостоятельной работы.

Параллельно выполняемые итерации

Иногда требуется перебрать список обещаний и выполнить некоторую задачу со всеми элементами списка по аналогии с тем, как это делается с синхронными массивами (например, `forEach(..)`, `map(..)`, `some(..)` и `every(..)`). Если задача, выполняемая для каждого обещания, синхронна по своей сути, эти конструкции нормально работают (см. использование `forEach(..)` в предыдущем фрагменте).

Но если задачи асинхронны или могут/должны выполняться параллельно, можно воспользоваться асинхронными версиями этих функций, предоставляемыми многими библиотеками.

Например, рассмотрим асинхронную функцию `map(..)`, получающую массив значений (это могут быть обещания или что угодно другое) и функцию (задачу), которая должна быть применена к каждому элементу. Функция `map(..)` сама возвращает обещание, значение выполнения которого представляет собой массив, содержащий (в исходном порядке) асинхронное значение выполнения для каждой задачи:

```
if (!Promise.map) {
    Promise.map = function(vals,cb) {
        // новое обещание, которое ожидает все обещания
        // для элементов
        return Promise.all(
            // примечание: обычная функция `map(..)`,
            // преобразует
            // массив значений в массив обещаний
            vals.map( function(val){
                // заменить `val` новым обещанием, которое
                // разрешается после асинхронного
                // отображения `val`
                return new Promise( function(resolve){
                    cb( val, resolve );
                } );
            } )
        );
    };
}
```

В этой реализации `map(..)` невозможны сигналы об отказе асинхронных задач, но если синхронное исключение или ошибка произойдут в обратном вызове отображения (`cb(..)`), то главное обещание, возвращенное `Promise.map(..)`, будет отклонено.

Продемонстрируем использование `map(..)` со списком обещаний (вместо простых значений):

```
var p1 = Promise.resolve( 21 );
var p2 = Promise.resolve( 42 );
var p3 = Promise.reject( "Oops" );

// удвоить значения в списке, даже если они
// хранятся в обещаниях
Promise.map( [p1,p2,p3], function(pr,done){
    // убедиться в том, что сам элемент является обещанием
    Promise.resolve( pr )
    .then(
        // извлечь значение в `v`
        function(v){
            // отобразить выполнение `v` на новое значение
            done( v * 2 );
        },
        // или отобразить на сообщение об отказе
        done
    );
} )
.then( function(vals){
    console.log( vals );    // [42,84,"Oops"]
} );
```

Снова о Promise API

Приведу краткую сводку ES6 `Promise` API, который уже был представлен по частям в разных местах этой главы.

Следующий API встроен только в ES6, но существуют полифилы, соответствующие спецификации (не просто расширенные библиотеки обещаний!), которые могут определять Promise и все сопутствующее поведение так, что встроенные обещания могут использоваться даже в браузерах, не поддерживающих

ES6. Я даже написал один из таких полифилов — «Native Promise Only» (http://github.com/getify/native-promise-only).

Конструктор new Promise(..)

Раскрывающий конструктор `Promise(..)` должен использоваться с `new` и должен получать функцию обратного вызова, которая вызывается синхронно/немедленно. Эта функция получает два обратных вызова, используемые для разрешения обещания. Обычно этим обратным вызовам присваиваются имена `resolve(..)` и `reject(..)`:

```
var p = new Promise( function(resolve,reject){
    // `resolve(..)` для разрешения/выполнения обещания
    // `reject(..)` для отказа
} );
```

`reject(..)` просто отклоняет обещание, но `resolve(..)` может как выполнить обещание, так и отклонить его в зависимости от того, что было передано. Если `resolve(..)` передается непосредственное значение, не являющееся обещанием или then-совместимым объектом, то обещание выполняется с этим значением.

Но если `resolve(..)` передается полноценное обещание или «thenable» значение, это значение распаковывается рекурсивно, и каким бы ни было его итоговое разрешение/состояние, оно будет принято обещанием.

Promise.resolve(..) и Promise.reject(..)

Сокращенной записью для создания уже отклоненного обещания является `Promise.reject(..)`, так что следующие два обещания эквивалентны:

```
var p1 = new Promise( function(resolve,reject){
    reject( "Oops" );
} );

var p2 = Promise.reject( "Oops" );
```

`Promise.resolve(..)` обычно используется для создания уже выполненных обещаний (по аналогии с `Promise.reject(..)`). Тем не менее `Promise.resolve(..)` также распаковывает «thenable» значения (как неоднократно упоминалось выше). В этом случае возвращаемое обещание принимает результат итогового разрешения переданного «thenable» значения, которым может быть как выполнение, так и отказ:

```
var fulfilledTh = {
    then: function(cb) { cb( 42 ); }
};
var rejectedTh = {
    then: function(cb,errCb) {
        errCb( "Oops" );
    }
};

var p1 = Promise.resolve( fulfilledTh );
var p2 = Promise.resolve( rejectedTh );

// `p1` будет выполненным обещанием
// `p2` будет отклоненным обещанием
```

И помните: метод `Promise.resolve(..)` ничего не сделает, если переданное значение уже является полноценным обещанием, он просто вернет это значение. А значит, вызов `Promise.resolve(..)` для значений, природа которых вам неизвестна, не приведет к потере быстродействия, если переданное значение окажется полноценным обещанием.

then(..) и catch(..)

Каждый экземпляр обещания (*не* пространство имен `Promise` API) содержит методы `then(..)` и `catch(..)`, что позволяет регистрировать обработчики выполнения и отказа для обещания. После того как обещание будет разрешено, вызывается тот или иной из этих обработчиков (но не оба сразу!), и вызов всегда будет асинхронным (см. раздел «Задания» главы 1).

`then(..)` получает один или два параметра (первый для обратного вызова выполнения, второй для обратного вызова отказа). Если любой из этих параметров опущен или вместо него передается значение, не являющееся функцией, подставляется соответствующий обратный вызов по умолчанию. Для выполнения обратный вызов по умолчанию просто передает сообщение дальше, а для отказа обратный вызов по умолчанию просто заново выдает (распространяет) полученную причину ошибки.

`catch(..)` получает в параметре только обратный вызов отказа и автоматически заменяет обратный вызов по умолчанию для выполнения, как указано выше. Иначе говоря, вызов эквивалентен `then(null,..)`:

```
p.then( fulfilled );

p.then( fulfilled, rejected );

p.catch( rejected ); // или `p.then( null, rejected )`
```

`then(..)` и `catch(..)` также создают и возвращают новое обещание, которое может использоваться для управления цепочкой обещаний. Если в обратных вызовах выполнения или отказа происходит исключение, то возвращаемое обещание отклоняется. Если хотя бы один из обратных вызовов вернет непосредственное значение, не являющееся обещанием или «thenable» значением, то это значение задается как значение выполнения возвращенного обещания. Если обработчик выполнения возвращает обещание или «thenable» значение, это значение распаковывается и становится результатом разрешения возвращенного обещания.

Promise.all([..]) и Promise.race([..])

Статические вспомогательные методы `Promise.all([ .. ])` и `Promise.race([ .. ])` из `Promise` API в ES6 создают обещание как возвращаемое значение. Разрешение этого обещания определяется исключительно массивом обещаний, который вы передаете при вызове.

У `Promise.all([ .. ])` все передаваемые обещания должны быть выполнены, чтобы выполнялось возвращаемое обещание. Если какое-либо обещание отклоняется, то и главное возвращаемое обещание тоже будет немедленно отклонено (с потерей результатов всех остальных обещаний). В случае выполнения вы получаете массив значений выполнения всех переданных обещаний. В случае отказа вы получаете значение причины только для первого отклоненного обещания. Этот паттерн традиционно называется *шлюзом*: чтобы шлюз открылся, должны прибыть все желающие пройти.

У `Promise.race([ .. ])` побеждает только первое разрешенное обещание (выполнение или отказ), и результат этого разрешения становится результатом разрешения возвращенного обещания. Этот паттерн традиционно называется *защелкой*: первый, кто открывает защелку, проходит далее. Пример:

```
var p1 = Promise.resolve( 42 );
var p2 = Promise.resolve( "Hello World" );
var p3 = Promise.reject( "Oops" );

Promise.race( [p1,p2,p3] )
.then( function(msg){
    console.log( msg );     // 42
} );

Promise.all( [p1,p2,p3] )
.catch( function(err){
    console.error( err );   // "Oops"
} );

Promise.all( [p1,p2] )
.then( function(msgs){
    console.log( msgs );    // [42,"Hello World"]
} );
```

Будьте осторожны! Если передать пустой массив методу `Promise.all([ .. ])`, он будет выполнен немедленно, но метод `Promise.race([ .. ])` зависнет и никогда не разрешится.

`Promise` API в ES6 достаточно прост и прямолинеен. Его, по крайней мере, хватает для основных случаев асинхронности и с него хорошо начинать перевод вашего кода с «кошмара обратных вызовов» на нечто более совершенное.

Но приложения часто требуют функциональности, которую обещания сами по себе предоставить не могут. В следующем разделе будут рассмотрены ограничения обещаний как основная причина для использования библиотек обещаний.

Ограничения обещаний

Многие подробности, которые рассматриваются в этом разделе, уже упоминались в этой главе, но сейчас я хочу особо выделить эти ограничения.

Последовательность обработки ошибок

Обработка ошибок при использовании обещаний подробно рассматривалась ранее в этой главе. Ограничения в архитектуре обещаний, а конкретно их объединения в цепочки, создают ловушку, которая легко может привести к случайному игнорированию ошибки в цепочке обещаний.

Однако с ошибками обещаний необходимо учитывать и другие обстоятельства. Так как цепочка обещаний представляет собой не что иное, как набор обещаний, связанных воедино, нет никакой сущности, которая позволяла бы сослаться на всю цепочку как на *единое целое*, а это означает, что не существует внешнего механизма отслеживания возникающих ошибок.

Если вы сконструируете цепочку обещаний, в которой отсутствует обработка ошибок, любая ошибка в этой цепочке будет бесконечно распространяться вниз по цепочке до тех пор, пока не будет отслежена (регистрацией обработчика отказа на каком-то шаге).

Итак, в этом конкретном случае достаточно ссылки на последнее обещание в цепочке (p в предыдущем фрагменте), потому что вы можете зарегистрировать обработчик отказа, и он будет получать уведомления о любых распространяемых ошибках:

```
// foo(..), STEP2(..) и STEP3(..) -
// функции с поддержкой обещаний

var p = foo( 42 )
.then( STEP2 )
.then( STEP3 );
```

И хотя это не очевидно, p в этом фрагменте указывает не на первое обещание в цепочке (от вызова `foo(42)`), а на последнее — то, которое происходит от вызова `then(STEP3)`.

Кроме того, ни один шаг в цепочке обещаний внешне не выполняет собственную обработку ошибок. Это означает, что вы можете зарегистрировать обработчик ошибки отказа для p и он будет получать уведомления о любых ошибках, возникающих в цепочке:

```
p.catch( handleErrors );
```

Но если какой-то шаг цепочки реализует собственную обработку ошибок (возможно, скрытую/абстрагированную от того, что вы видите), то ваш обработчик `handleErrors(..)` уведомления не получит. Возможно, это именно то, что вам нужно (в конце концов, это был «обработанный отказ»), — но может быть и совсем не то. Полное отсутствие возможности уведомления (или «уже обработанных» ошибок отказа) ограничивает ваши возможности в некоторых ситуациях.

По сути, здесь действует то же ограничение, которое существует у конструкции `try..catch`, — эта конструкция может перехватить исключение и просто поглотить его. Получается, что это ограничение присуще не только обещаниям, но для него хотелось бы иметь обходное решение.

К сожалению, часто ссылки на промежуточные звенья цепочки обещаний нигде не хранятся, а без таких ссылок вы не сможете

присоединить обработчики ошибок для надежного отслеживания ошибок.

Единственное значение

Обещания по умолчанию имеют только одно значение выполнения или одну причину отказа. В простых ситуациях это не создает особых проблем, однако в более сложных оно может ограничить ваши возможности.

Как правило, в таких случаях рекомендуется сконструировать обертку для хранения набора значений (например, объект или массив). Такое решение работает, но оно может быть достаточно неудобным и рутинным: вам придется упаковывать и распаковывать ваши сообщения на каждом одиночном шаге цепочки обещаний.

Разбиение значений

Иногда в такой ситуации задачу лучше разбить на два и более обещания.

Представьте, что у вас имеется функция `foo(..)`, которая асинхронно производит два значения (`x` и `y`):

```
function getY(x) {
    return new Promise( function(resolve,reject){
        setTimeout( function(){
            resolve( (3 * x) - 1 );
        }, 100 );
    } );
}

function foo(bar,baz) {
    var x = bar * baz;
    return getY( x )
    .then( function(y){
        // упаковать оба значения в контейнер
        return [x,y];
```

```
    } );
}

foo( 10, 20 )
.then( function(msgs){
    var x = msgs[0];
    var y = msgs[1];
    console.log( x, y );    // 200 599
} );
```

Для начала переставим то, что возвращает `foo(..)`, чтобы `x` и `y` не приходилось упаковывать в один массив для передачи через одно обещание. Вместо этого каждое значение упаковывается в собственное обещание:

```
function foo(bar,baz) {
    var x = bar * baz;

    // вернуть оба обещания
    return [
        Promise.resolve( x ),
        getY( x )
    ];
}

Promise.all(
    foo( 10, 20 )
)
.then( function(msgs){
    var x = msgs[0];
    var y = msgs[1];

    console.log( x, y );
} );
```

Можно ли сказать, что массив обещаний действительно лучше массива значений, переданных через одно обещание? С точки зрения синтаксиса особых усовершенствований не видно.

Но этот подход более точно воплощает теорию проектирования обещаний. В будущем будет проще произвести рефакторинг и разбить вычисление `x` и `y` на раздельные функции. Вариант, при котором вызывающий код сам решает, как организовать обработку

двух обещаний, получается более гибким и чистым (в данном случае используется `Promise.all([ .. ])`, но, безусловно, это не единственный вариант вместо абстрагирования таких подробностей внутри `foo(..)`).

Распаковка/распределение аргументов

Присваивания `var x = ..` и `var y = ..` все еще остаются неудобным балластом. Можно воспользоваться трюком со вспомогательной функцией (за этот совет спасибо Реджинальду Брайтуэйту (Reginald Braithwaite), @raganwald на Twitter):

```
function spread(fn) {
    return Function.apply.bind( fn, null );
}

Promise.all(
    foo( 10, 20 )
)
.then(
    spread( function(x,y){
        console.log( x, y );    // 200 599
    } )
)
```

Так гораздо лучше! И конечно, можно определить код во встроенном виде, чтобы избежать создания лишней функции:

```
Promise.all(
    foo( 10, 20 )
)
.then( Function.apply.bind(
    function(x,y){
        console.log( x, y );    // 200 599
    },
    null
) );
```

Все эти трюки удобны, но в ES6 появилось еще более элегантное решение: *деструктуризация*. Деструктуризирующее присваивание массива выглядит так:

```
Promise.all(
    foo( 10, 20 )
)
.then( function(msgs){
    var [x,y] = msgs;

    console.log( x, y );    // 200 599
} );
```

Но и это не все: в ES6 также поддерживается деструктуризация параметра-массива:

```
Promise.all(
    foo( 10, 20 )
)
.then( function([x,y]){
    console.log( x, y );    // 200 599
} );
```

Принцип «одно значение на обещание» соблюден, но объем служебного шаблонного кода сведен к минимуму!

Однократное разрешение

Одна из самых фундаментальных особенностей поведения обещаний заключается в том, что обещание может быть разрешено только один раз (выполнение или отказ). Во многих асинхронных сценариях значение должно быть получено однократно, так что это не создает проблем.

Однако существует также много асинхронных сценариев, которые лучше представляются другой моделью, больше похожей на события и/или потоки данных. На первый взгляд неясно, насколько хорошо обещания соответствуют таким сценариям использования (и соответствуют ли вообще). Без построения значительных абстракций поверх обещаний они совершенно не подходят для обработки множественных значений разрешения.

Представьте, что вам нужно сгенерировать серию асинхронных действий по некоторой причине (скажем, событию), кото-

рая может происходить несколько раз (например, щелчок на кнопке).

Вероятно, такое решение будет работать не так, как вам нужно:

```
// `click(..)` связывает событие `"click"` с элементом DOM
// `request(..)` - ранее определенная функция Ajax с поддержкой
// обещаний

var p = new Promise( function(resolve,reject){
    click( "#mybtn", resolve );
} );

p.then( function(evt){
    var btnID = evt.currentTarget.id;
    return request( "http://some.url.1/?id=" + btnID );
} )
.then( function(text){
    console.log( text );
} );
```

Такое поведение работает в том случае, если приложение требует, чтобы щелчок на кнопке выполнялся только один раз. Если щелкнуть на кнопке повторно, второй вызов `resolve(..)` будет проигнорирован, так как обещание `p` уже было разрешено.

Вероятно, вместо этого следует сменить направление парадигмы и создавать новую цепочку обещаний для каждого срабатывающего события:

```
click( "#mybtn", function(evt){
    var btnID = evt.currentTarget.id;
    request( "http://some.url.1/?id=" + btnID )
    .then( function(text){
        console.log( text );
    } );
} );
```

При таком решении каждое новое событие `"click"` для кнопки будет порождать новую последовательность обещаний.

Но даже если не принимать во внимание уродство решения, которое требует определять всю цепочку обещаний внутри обработ-

чика событий, такая архитектура в некоторых отношениях нарушает принцип разделения обязанностей (SoC). Вполне возможно, что вам потребуется определить свой обработчик события совсем не в том месте кода, в каком определяется реакция на событие (цепочка обещаний). Сделать это в данном паттерне без каких-то вспомогательных механизмов довольно неудобно.

Это ограничение также можно выразить другим способом: было бы хорошо, если бы мы могли сконструировать некий «наблюдаемый объект», на который можно было подписать цепочку обещаний. Существуют библиотеки, которые создают такие абстракции (например, RxJS), но эти абстракции настолько тяжеловесны, что за ними уже не видна природа обещаний. Эти тяжеловесные абстракции поднимают важные вопросы — например, пользуются ли эти механизмы тем же уровнем доверия, с каким проектировались сами обещания. Паттерн «наблюдаемый объект» будет рассматриваться в приложении Б.

Инерция

Ощутимым препятствием к переходу на обещания в вашем коде становится весь существующий код, в котором обещания пока не поддерживаются. Если у вас имеется большой объем кода на базе обратных вызовов, гораздо проще просто продолжать программировать в том же стиле.

«Развивающаяся кодовая база (с обратными вызовами) так и продолжит развиваться (с обратными вызовами), если только на нее не воздействует умный разработчик, который умеет работать с обещаниями».

Обещания предлагают другую парадигму и, как следствие, могут потребовать нового подхода — от незначительных изменений до принципиального переосмысления. При этом необходимо действовать сознательно, потому что обещания не вырастут из старых подходов к программированию, так надежно служивших вам до сих пор.

Рассмотрим примерный сценарий на базе обратных вызовов:

```
function foo(x,y,cb) {
    ajax(
        "http://some.url.1/?x=" + x + "&y=" + y,
        cb
    );
}

foo( 11, 31, function(err,text) {
    if (err) {
        console.error( err );
    }
    else {
        console.log( text );
    }
} );
```

Сможете с ходу сказать, с чего нужно начать преобразование этого кода с обратными вызовами в код с поддержкой обещаний? Это зависит от опыта. Чем больше у вас будет опыта, тем естественнее будет все происходить. Но конечно, к обещаниям никто не пришивает ярлык с точными инструкциями — не существует единого решения на все случаи жизни, поэтому вся ответственность лежит на вас.

Как упоминалось ранее, нам определенно нужна функция Ajax, которая бы работала на базе обещаний вместо обратных вызовов; такая функция могла бы называться `request(..)`. Вы можете написать собственную версию, как мы уже сделали выше. Однако если вам придется вручную писать обертки, совместимые с обещаниями, для всех функций на базе обратных вызовов, то скорее всего, вы вообще не станете заниматься рефакторингом кода и переходом на программирование с обещаниями.

Обещания не предоставляют очевидного ответа на это ограничение. Впрочем, в большинстве библиотек обещаний имеются вспомогательные функции. Но даже без библиотеки можно представить себе вспомогательную функцию следующего вида:

```
// защитная проверка, безопасная для полифилов
if (!Promise.wrap) {
    Promise.wrap = function(fn) {
        return function() {
            var args = [].slice.call( arguments );
            return new Promise( function(resolve,reject){
                fn.apply(
                    null,
                    args.concat( function(err,v){
                        if (err) {
                            reject( err );
                        }
                        else {
                            resolve( v );
                        }
                    } )
                );
            } );
        };
    };
}
```

Что-то не очень похоже на «маленькую тривиальную функцию». А впрочем, несмотря на устрашающий вид, все не так плохо, как может показаться. Используется функция, которая ожидает получить в последнем параметре обратный вызов в стиле «ошибка на первом месте», и возвращает новую функцию, которая автоматически создает возвращаемое обещание и подставляет обратный вызов, связанный с выполнением/отклонением обещания.

Но не стоит тратить время на разговоры о том, *как* работает вспомогательная функция `Promise.wrap(..)`. Лучше просто посмотреть, как она используется:

```
var request = Promise.wrap( ajax );

request( "http://some.url.1/" )
.then( .. )
..
```

Как видите, проще простого!

Функция `Promise.wrap(..)` *не* создает обещания, она создает функцию, которая создает обещания. В каком-то смысле функция, производящая обещания, может рассматриваться как фабрика обещаний.

Таким образом, `Promise.wrap(ajax)` создает фабрику обещаний `ajax(..)`, которую мы называем `request(..)`, и эта фабрика обещаний производит обещания для ответов Ajax.

Если бы все функции уже были фабриками обещаний, нам не приходилось бы создавать их самостоятельно, так что этот дополнительный шаг немного огорчает. Но по крайней мере паттерн упаковки (обычно) мы повторяем, так что можем заключить его во вспомогательную функцию `Promise.wrap(..)` для упрощения программирования с обещаниями.

Вернемся к приведенному ранее примеру. Нам понадобится фабрика обещаний как для `ajax(..)`, так и для `foo(..)`:

```
// создание фабрики обещаний для `ajax(..)`
var request = Promise.wrap( ajax );

// выполнить рефакторинг `foo(..)`, но сохранить внешнюю
// поддержку обратных вызовов для обеспечения совместимости
// с другими частями кода -- обещание `request(..)` используется
// только во внутренней реализации.
function foo(x,y,cb) {
    request(
        "http://some.url.1/?x=" + x + "&y=" + y
    )
    .then(
        function fulfilled(text){
            cb( null, text );
        },
        cb
    );
}

// для целей этого кода создать
// фабрику обещаний для `foo(..)`
var betterFoo = Promise.wrap( foo );
```

```
// использовать фабрику обещаний
betterFoo( 11, 31 )
.then(
    function fulfilled(text){
        console.log( text );
    },
    function rejected(err){
        console.error( err );
    }
);
```

Конечно, хотя мы выполняем рефакторинг `foo(..)` для использования новой фабрики обещаний `request(..)`, с таким же успехом можно было сделать фабрикой обещаний `foo(..)`, вместо того, чтобы оставить функцию на базе обратных вызовов с необходимостью использования фабрики обещаний `betterFoo(..)`. Это решение просто зависит от того, нужно ли `foo(..)` сохранять совместимость на уровне обратных вызовов с другими частями кодовой базы.

Пример:

```
функция `foo(..)` теперь тоже является фабрикой обещаний,
потому что она делегирует фабрике обещаний `request(..)`
function foo(x,y) {
    return request(
        "http://some.url.1/?x=" + x + "&y=" + y
    );
}
foo( 11, 31 )
.then( .. )
..
```

Хотя обещания ES6 не имеют встроенных вспомогательных средств для такой упаковки фабрик обещаний, они предоставляются большинством библиотек (хотя вы также можете реализовать их самостоятельно). В любом случае, это конкретное ограничение обещаний решается без особых проблем (конечно, по сравнению с «кошмаром обратных вызовов»).

Неотменяемость обещаний

После того как вы создали обещание и зарегистрировали для него обработчик выполнения и/или отказа, нет никаких внешних возможностей для остановки этого состояния дел, если по какой-либо причине задача становится неактуальной.

Многие библиотеки абстракции обещаний предоставляют средства для отмены обещаний, но это крайне неудачная мысль! Многие разработчики хотели бы, чтобы обещания изначально проектировались с возможностью внешней отмены, но это позволило бы одному потребителю/наблюдателю обещания влиять на возможность другого потребителя наблюдать за этим же обещанием. Это нарушает надежность (внешнюю неизменяемость) будущего значения, а также является воплощением антипаттерна «действие на расстоянии». Какой бы полезной ни казалась такая возможность, она в конечном итоге приведет вас обратно к таким же кошмарам, как при использовании обратных вызовов.

Рассмотрим сценарий с тайм-аутом обещаний, представленный выше:

```
var p = foo( 42 );

Promise.race( [
    p,
    timeoutPromise( 3000 )
] )
.then(
    doSomething,
    handleError
);

p.then( function(){
    // все равно происходит даже при тайм-ауте :(
} );
```

«Тайм-аут» был внешним по отношению к обещанию `p`, так что `p` продолжит работать (чего мы, скорее всего, не хотим).

Один из вариантов — агрессивное определение собственных обратных вызовов разрешения:

```
var OK = true;

var p = foo( 42 );

Promise.race( [
    p,
    timeoutPromise( 3000 )
    .catch( function(err){
        OK = false;
        throw err;
    } )
] )
.then(
    doSomething,
    handleError
);

p.then( function(){
    if (OK) {
        // происходит только при отсутствии тайм-аута! :)
    }
} );
```

Такое решение уродливо. Оно работает, но от идеала далеко. В общем случае таких сценариев лучше избегать.

Но если это невозможно, уродство этого решения должно подсказать, что отмена — функциональность, которая относится к более высокому уровню абстракций на базе обещаний. Я рекомендую обратиться за помощью к библиотекам абстракции обещаний, вместо того чтобы сооружать решение самостоятельно.

Моя библиотека абстракции обещаний asynquence предоставляет как раз такую абстракцию и функцию отмены `abort()`; обе возможности будут рассмотрены в приложении А.

Одиночное обещание само по себе не может считаться механизмом управления программной логикой (по крайней мере, в содержательном смысле). Собственно, именно поэтому отмена обещаний является такой неудобной.

С другой стороны, цепочка обещаний, взятая как единое целое — то, что я обычно называю «последовательностью», — выражает управление программной логикой. Следовательно, отмену уместно определять именно на этом уровне абстракции.

Ни одно отдельное обещание не должно быть отменяемым, однако будет разумно иметь возможность отмены *последовательности*, потому что последовательность не передается как единое неизменяемое значение (в отличие от обещаний).

Эффективность обещаний

Это конкретное ограничение одновременно и просто, и сложно.

Если сравнить количество составляющих, задействованных в базовых асинхронных цепочках задач на базе обратных вызовов, с цепочками обещаний, становится ясно, что механизм обещаний устроен сложнее, а следовательно, они хотя бы в какой-то степени работают медленнее. Вспомните простой список гарантий доверия, предоставляемых обещаниями, и сравните его с кодом ситуационных решений, который вам приходится накладывать поверх обратных вызовов для достижения тех же гарантий защиты.

Больший объем работы и увеличение количества гарантий означают, что обещания работают *медленнее* простых обратных вызовов, не предоставляющих никакой защиты. Вроде бы все очевидно и понятно.

Но насколько медленнее? А вот это вопрос, на который невероятно сложно дать однозначный ответ.

Откровенно говоря, здесь сравниваются разные вещи, так что вопрос просто некорректен. Сравнивать следует ситуационное ре-

шение с обратными вызовами с теми же защитными мерами, наложенными вручную, — будет ли оно работать быстрее реализации обещаний?

Если у обещаний и есть законное ограничение эффективности, так это то, что они не позволяют выбрать нужные вам гарантии доверия, вы получаете их все и всегда.

Тем не менее, если согласиться с тем, что обещание обычно *немного* медленнее своего эквивалента с обратными вызовами без гарантий доверия (предполагая, что в некоторых ситуациях отсутствие доверия может быть оправданно), означает ли это, что обещаний нужно избегать повсеместно, потому что в вашем приложении всегда должен работать самый быстрый код из всех возможных?

Если вы можете положительно ответить на этот вопрос, то правильно ли выбран язык JavaScript для таких задач? JavaScript может оптимизироваться для крайне эффективного выполнения приложений (см. главы 5 и 6). Но насколько уместно навязчивое стремление к ничтожному приросту быстродействия с потерей всех преимуществ, предоставляемых обещаниями?

Другой неочевидный аспект заключается в том, что обещания делают *все* операции асинхронными; это означает, что выполнение некоторых немедленно (синхронно) завершаемых действий откладывается до следующего шага задания (см. главу 1). И последовательность задач обещаний может завершаться чуть медленнее той же последовательности, связанной обратными вызовами.

И теперь вопрос звучит так: стоит ли потенциальная потеря ничтожных процентов быстродействия всех потерянных достоинств обещаний, которые были описаны в этой главе?

Я считаю, что если снижение быстродействия обещаний заслуживает того, чтобы принимать его во внимание, практически всегда оно оказывается антипаттерном, который приводит к по-

тере преимуществ, связанных с гарантиями доверия и удобства композиции.

Вместо этого следует по умолчанию использовать обещание во всей кодовой базе, а потом провести профилирование и проанализировать критические ветви вашего приложения. Действительно ли обещания становятся «узким местом» вашей программы или это замедление просто теоретическое? И тогда, вооружившись достоверными эталонными тестами (см. главу 6), вы сможете ответственно и разумно исключить обещания только в этих выявленных критических областях.

Обещания работают чуть медленнее, однако взамен вы получаете значительную степень доверия, предсказуемость поведения и удобные встроенные средства композиции. А может, ограничена на самом деле не эффективность обещаний, а ваше понимание их преимуществ?

Итоги

Обещания прекрасны. Пользуйтесь ими. Они успешно решают проблемы инверсии управления, которые преследуют нас в коде, основанном только на обратных вызовах.

Обещания не избавляются от обратных вызовов — они просто перенаправляют организацию этих обратных вызовов доверенному промежуточному механизму, который располагается между вами и другим кодом.

Цепочки обещаний также лучше выражают (хотя, конечно, и не идеально) асинхронную программную логику в последовательном виде, который помогает нашему мозгу планировать и сопровождать асинхронный код JS. Более качественное решение этой задачи представлено в главе 4!

4 Генераторы

В главе 2 были выявлены два ключевых недостатка выражения асинхронной программной логики с использованием обратных вызовов:

- Асинхронность на базе обратных вызовов не соответствует тому, как наш мозг планирует этапы выполнения задачи.
- Обратные вызовы не пользуются доверием и не обладают средствами композиции из-за инверсии управления.

В главе 3 мы подробно разобрали, как обещания восстанавливают инверсию управления обратных вызовов и предоставляют гарантии доверия и средства композиции.

А теперь обратимся к выражению асинхронной программной логики в последовательном синхронном виде. «Волшебство», благодаря которому это становится возможным, — *генераторы* ES6.

Нарушение принципа выполнения до завершения

В главе 1 объяснялось одно предположение, на которое разработчики JS почти всегда полагаются в своем коде: после того, как

функция начнет выполняться, она отрабатывает до завершения, и никакой другой код не сможет прервать ее и вклиниться в процесс выполнения.

Как бы странно это ни выглядело, в ES6 появилась новая разновидность функций, поведение которых не соответствует принципу выполнения до завершения. Такие функции называются *генераторами*.

Чтобы понять суть происходящего, рассмотрим следующий пример:

```
var x = 1;

function foo() {
    x++;
    bar();              // <-- как насчет этой строки?
    console.log( "x:", x );
}

function bar() {
    x++;
}

foo();                  // x: 3
```

В этом примере мы точно знаем, что `bar()` отрабатывает между `x++` и `console.log(x)`. А если бы вызова `bar()` не было? Очевидно, результат будет равен 2 вместо 3.

А теперь проявим немного воображения. Что, если вызов `bar()` отсутствует, но она все равно каким-то образом сможет выполниться между командами `x++` и `console.log(x)`? Разве такое возможно?

В многопоточных языках с вытеснением может случиться, что `bar()` вмешается и отработает прямо в подходящий момент между двумя командами. Но в JS нет вытеснения, да и язык не является многопоточным (в настоящее время). Тем не менее кооперативная форма такого прерывания возможна, если функция `foo()` сможет как-то обозначить паузу в этой части кода.

Я использую термин «кооперативный» не только из-за связи с классической терминологией параллелизма (см. главу 1), но и потому, что, как будет показано в следующем фрагменте, в ES6 для обозначения точки приостановки используется ключевое слово `yield`, котрое предполагает добровольную кооперативную уступку управления.

Вот как выглядит код ES6 для реализации кооперативного параллельного выполнения:

```
var x = 1;

function *foo() {
    x++;
    yield; // приостановка!
    console.log( "x:", x );
}

function bar() {
    x++;
}
```

Возможно, вы увидите в другой документации/коде JS другой формат объявления генераторов в виде `function* foo() { .. }` (вместо `function *foo() { .. }`, как у меня) — отличается только стилистическое позиционирование *. Две формы функционально/синтаксически идентичны, как и третья форма `function*foo() { .. } ()`. В обоих стилях могут передаваться аргументы, но я обычно предпочитаю запись `function *foo..`, потому что она соответствует ссылке на генератор `*foo()`. Если ограничиться именем `foo()`, будет не столь очевидно, о чем идет речь — о генераторе или об обычной функции. Впрочем, это чисто стилистические предпочтения.

Как теперь запустить код из предыдущего фрагмента, чтобы функция `bar()` выполнялась в точке уступки управления внутри `*foo()`?

```
// сконструировать итератор `it` для управления генератором
var it = foo();
```

```
// здесь запускается `foo()`!
it.next();
x;                      // 2
bar();
x;                      // 3
it.next();              // x: 3
```

В этих двух фрагментах происходит немало нового и малопонятного, так что нам придется разбираться. Но перед тем, как объяснять использование разных механик/вариантов синтаксиса с генераторами ES6, рассмотрим порядок выполнения операций:

1. Операция `it = foo()` еще не выполняет генератор `*foo()` — она всего лишь конструирует *итератор*, который будет управлять его выполнением. Вскоре *итераторы* будут рассмотрены более подробно.
2. Первая команда `it.next()` запускает генератор `*foo()` и выполняет `x++` в первой строке `*foo()`.
3. `*foo()` приостанавливает выполнение на команде `yield`, и первый вызов `it.next()` завершается. В этот момент `*foo()` продолжает работать и сохраняет активность, но находится в приостановленном состоянии.
4. Проверяем значение `x`, оно сейчас равно 2.
5. Вызываем функцию `bar()`, которая снова увеличивает `x` командой `x++`.
6. Снова проверяем значение `x`, на этот раз оно равно 3.
7. Итоговый вызов `it.next()` возобновляет выполнение генератора `*foo()` с точки приостановки и выполняет команду `console.log(..)`, в которой используется текущее значение `x`, равное 3.

Очевидно, функция `foo()` запускается, но не выполняется до завершения, она приостанавливается в точке `yield`. Работа `foo()` была продолжена позднее, и функция отработала до конца, но это даже не было обязательно.

Итак, генератор — это особая разновидность функций, которые могут запускаться и останавливаться один или несколько раз, и даже не обязательно завершают свое выполнение.

Наверное, мощь генераторов на первый взгляд неочевидна, но по мере изложения материала этой главы они станут одним из основных структурных элементов, используемых для конструирования генераторов как средства управления асинхронной программной логикой.

Ввод и вывод

Функция-генератор — специальная функция с новой моделью обработки, которую мы только что описали. Тем не менее она остается функцией; это означает, что некоторые фундаментальные свойства остаются неизменными, а именно, что она все еще получает аргументы (ввод) и может возвращать значение (вывод):

```
function *foo(x,y) {
    return x * y;
}

var it = foo( 6, 7 );

var res = it.next();

res.value;      // 42
```

Аргументы `6` и `7` передаются `*foo(..)` в параметрах `x` и `y` соответственно. Кроме того, `*foo(..)` возвращает значение `42` на сторону вызова.

И здесь мы видим отличие в вызове генератора по сравнению с нормальной функцией. Конечно, запись `foo(6,7)` выглядит знакомо, но при этом генератор `*foo(..)` еще не запустился, как это было бы с обычной функцией.

Вместо этого мы создаем объект-*итератор* для управления генератором `*foo(..)` и присваиваем его переменной `it`. После этого

вызывается метод `it.next()`, который приказывает генератору `*foo(..)` переместиться от текущей позиции либо до следующей позиции `yield`, либо до конца генератора.

Результатом вызова `next(..)` является объект со свойством `value`, которое содержит значение, возвращенное `*foo(..)`. Иначе говоря, `yield` приводит к передаче значения генератором в середине его выполнения — что-то вроде промежуточного `return`.

И снова неочевидно, зачем нужен дополнительный объект-итератор для управления генератором. Мы доберемся до этого, я *обещаю*.

Передача сообщений при итерациях

Генераторы получают аргументы и могут возвращать значения, но существует еще более мощный и удобный механизм передачи сообщений ввода/вывода с использованием `yield` и `next(..)`.

Пример:

```
function *foo(x) {
    var y = x * (yield);
    return y;
}

var it = foo( 6 );

// запустить `foo(..)`
it.next();

var res = it.next( 7 );

res.value;      // 42
```

Сначала значение `6` передается в параметре `x`. Затем вызывается метод `it.next()`, который запускает `*foo(..)`.

Внутри `*foo(..)` начинается обработка команды `var y = x ..`, но тут же встречается выражение `yield`. В этой точке выполнение `*foo(..)`

приостанавливается (в середине команды присваивания!), а вызывающий код фактически должен предоставить результат для выражения `yield`. Вызов метода `it.next( 7 )` передает значение `7` обратно как результат приостановленного выражения `yield`.

Итак, на этой стадии команда присваивания фактически принимает вид `var y = 6 * 7`. Теперь `return y` возвращает значение `42` как результат вызова `it.next( 7 )`.

Обратите внимание на один очень важный момент, легко сбивающий с толку даже опытных разработчиков JS: в зависимости от точки зрения между вызовами `yield` и `next(..)` может существовать несоответствие. В общем случае количество вызовов `next(..)` на 1 больше количества команд `yield` — в предыдущем фрагменте содержится одна команда `yield` и два вызова `next(..)`.

Откуда берется расхождение?

Первый вызов `next(..)` всегда запускает генератор и выполняет его до первого `yield`. Второй вызов `next(..)` выполняет первое приостановленное выражение `yield`, третий вызов `next(..)` выполняет второе выражение `yield`, и т. д.

Два вопроса

На самом деле субъективное наличие или отсутствие расхождения зависит от того, о каком коде вы думаете в первую очередь.

Возьмем только код генератора

```
var y = x * (yield);
return y;
```

Первое вхождение `yield`, по сути, задает вопрос: «Какое значение я должен подставить сюда?»

Кто должен ответить на этот вопрос? Что ж, *первый* вызов `next()` уже был выполнен для того, чтобы привести генератор в эту точ-

ку, поэтому очевидно, что он не может дать ответ. Таким образом, на вопрос, поставленный первым `yield`, должен ответить *второй* вызов `next(..)`.

Но давайте поменяем точку зрения. Взглянем на происходящее не с точки зрения генератора, а с точки зрения итератора.

Чтобы должным образом продемонстрировать эту точку зрения, необходимо объяснить, что сообщения могут передаваться в обоих направлениях: `yield ..` как выражение может отправлять сообщения в ответ на вызовы `next(..)`, а `next(..)` может отправлять значения приостановленному выражению `yield`. Рассмотрим эту слегка модифицированную версию кода:

```
function *foo(x) {
    var y = x * (yield "Hello");    // <-- возвращает значение!
    return y;
}

var it = foo( 6 );

var res = it.next();    // первый вызов `next()`, ничего
                        // не передается
res.value;              // "Hello"

res = it.next( 7 );     // передать `7` ожидающему `yield`
res.value;              // 42
```

`yield ..` в сочетании с `next(..)` образует двустороннюю систему передачи сообщений *во время выполнения генератора*.

Итак, посмотрим только на код *итератора*:

```
var res = it.next();    // первый вызов `next()`, ничего
                        // не передается
res.value;              // "Hello"

res = it.next( 7 );     // передать `7` ожидающему `yield`
res.value;              // 42
```

При первом вызове `next()` значение не передается, и это делается намеренно. Только приостановленная команда `yield` может получить такое значение, переданное `next(..)`, а в начале выполнения генератора при первом вызове `next()` еще нет приостановленной команды `yield`, которая бы приняла такое значение. Спецификация и все поддерживающие ее браузеры просто отбрасывают все, что было передано первому вызову `next()`. Все равно передавать это значение не рекомендуется, так как вы фактически создаете код, который незаметно игнорирует ошибочное условие, а это усложняет его понимание. Всегда начинайте генератор с вызова `next()` без аргументов.

Первый вызов `next()` (которому ничего не передается) фактически *задает вопрос*: «Какое *следующее* значение генератор `*foo(..)` предоставит мне?» И кто ответит на этот вопрос? Первое выражение `yield "hello"`.

Видите? Здесь никаких расхождений нет.

В зависимости от того, кто, по вашему мнению, задает вопрос, расхождение между вызовами `yield` и `next(..)` может существовать, а может и не существовать.

Погодите! Все равно существует один лишний вызов `next()` по сравнению с количеством команд `yield`. Итак, последний вызов `it.next(7)` снова задает вопрос о том, какое следующее значение произведет генератор. Но команд `yield`, которые бы могли дать ответ, не осталось. Кто же ответит на этот вопрос?

Команда `return`!

А если в вашем генераторе отсутствует `return` (в генераторах эта команда является не более обязательной, чем в обычных функциях), всегда остается неявная/предполагаемая команда `return;` (то есть `return undefined;`). Она даст ответ по умолчанию на вопрос, поставленный последним вызовом `it.next(7)`.

Этот механизм вопросов и ответов — двусторонняя передача сообщений `yield` и `next(..)` — достаточно мощен, но на первый взгляд совершенно не очевидно, как эти механизмы связаны с асинхронным управлением программной логикой. Скоро вы это узнаете!

Множественные итераторы

Синтаксис может создать впечатление, что при использовании итератора для управления генератором вы управляете непосредственно объявленной функцией-генератором. Однако существует один нюанс, который легко упустить из виду: каждый раз, когда вы создаете итератор, вы неявно создаете экземпляр генератора, которым этот итератор будет управлять.

Несколько экземпляров одного генератора могут работать одновременно и даже взаимодействовать друг с другом:

```
function *foo() {
    var x = yield 2;
    z++;
    var y = yield (x * z);
    console.log( x, y, z );
}

var z = 1;

var it1 = foo();
var it2 = foo();

var val1 = it1.next().value;           // 2 <-- yield 2
var val2 = it2.next().value;           // 2 <-- yield 2

val1 = it1.next( val2 * 10 ).value;    // 40  <-- x:20,  z:2
val2 = it2.next( val1 * 5 ).value;     // 600 <-- x:200, z:3

it1.next( val2 / 2 );                  // y:300
                                       // 20 300 3
it2.next( val1 / 4 );                  // y:10
                                       // 200 10 3
```

Чаще всего несколько параллельно работающих экземпляров одного генератора используются не для таких взаимодействий, а для производства генератором собственных значений без ввода (возможно, из некоторого независимо подключенного ресурса). Производство значений более подробно рассматривается в следующем разделе.

Краткий обзор происходящих событий:

1. Оба экземпляра `*foo()` запускаются одновременно. Следующие вызовы `next()` получают значение 2 от команд `yield 2` соответственно.

2. `val2 * 10` дает `2 * 10`; значение передается первому экземпляру генератора `it1`, так что `x` получает значение `20`. `z` увеличивается от `1` до `2`, после чего `20 * 2` выдается командой `yield`, и в результате `val1` присваивается значение `40`.

3. `val1 * 5` дает `40 * 5`; значение передается второму экземпляру генератора `it2`, так что `x` получает значение `200`. `z` снова увеличивается от `2` до `3`, после чего `200 * 3` выдается командой `yield`, и в результате `val2` присваивается значение `600`.

4. `val2 / 2` дает `600 * 2`; значение передается первому экземпляру генератора `it1`, так что `y` получает значение `300`. После этого выводятся значения `20 300 3` для переменных `x y z` соответственно.

5. `val1 / 4` дает `40 * 2`; значение передается второму экземпляру генератора `it2`, так что `y` получает значение `10`. После этого выводятся значения `200 10 3` для переменных `x y z` соответственно.

Это довольно забавный пример, который стоит как следует обдумать. А вам удалось правильно понять, что здесь происходит?

Чередование

Вспомните следующий сценарий из раздела «Выполнение до завершения» главы 1:

```
var a = 1;
var b = 2;

function foo() {
    a++;
    b = b * a;
    a = b + 3;
}

function bar() {
    b--;
    a = 8 + b;
    b = a * 2;
}
```

Конечно, с нормальными функциями JS либо сначала полностью выполнится `foo()`, либо сначала полностью выполнится `bar()`, но отдельные команды `foo()` не смогут чередоваться с командами `bar()`. Таким образом, в этой программе возможны только два результата.

Однако с генераторами возможно чередование (и даже в середине команд!):

```
var a = 1;
var b = 2;

function *foo() {
    a++;
    yield;
    b = b * a;
    a = (yield b) + 3;
}

function *bar() {
    b--;
    yield;
    a = (yield 8) + b;
    b = a * (yield 2);
}
```

В зависимости от относительного порядка вызова *итераторов*, управляющих `*foo()` и `*bar()`, приведенная программа может вы-

дать несколько разных результатов. Иначе говоря, мы можем продемонстрировать (хотя и несколько искусственно) теоретическое многопоточное состояние гонки, рассмотренной в главе 1, посредством чередования итераций двух генераторов с одними общими переменными.

Начнем со вспомогательной функции `step(..)`, которая управляет итератором:

```
function step(gen) {
    var it = gen();
    var last;
    return function() {
        // какое бы значение ни было выдано `yield`,
        // просто вернуть его в следующий раз!
        last = it.next( last ).value;
    };
}
```

`step(..)` инициализирует генератор для создания его *итератора*, а затем возвращает функцию, которая при вызове продвигает *итератор* на один шаг. Кроме того, ранее переданное `yield` значение возвращается обратно на следующем шаге. Таким образом, `yield 8` дает просто `8`, а `yield b` дает `b` (каким бы оно ни было в момент `yield`).

А теперь просто для интереса немного поэкспериментируем, чтобы понаблюдать за эффектом чередования разных фрагментов `*foo()` и `*bar()`. Начнем со скучного базового случая, при котором `*foo()` гарантированно завершается до `*bar()` (как было в главе 1):

```
// обязательно сбросить `a` и `b`
a = 1;
b = 2;

var s1 = step( foo );
var s2 = step( bar );

// сначала полностью завершить `*foo()`
s1();
s1();
```

```
s1();

// теперь запустить `*bar()`
s2();
s2();
s2();
s2();

console.log( a, b );    // 11 22
```

В результате получаем 11 и 22, как и в версии из главы 1. Теперь изменим порядок чередования и посмотрим, как это повлияет на итоговые значения a и b:

```
// обязательно сбросить `a` и `b`
a = 1;
b = 2;

var s1 = step( foo );
var s2 = step( bar );

s2();       // b--;
s2();       // yield 8
s1();       // a++;
s2();       // a = 8 + b;
            // yield 2
s1();       // b = b * a;
            // yield b
s1();       // a = b + 3;
s2();       // b = a * 2;
```

Но прежде чем я сообщу результаты, сможете ли вы самостоятельно определить значения a и b после этой программы? И не подглядывать!

```
console.log( a, b );    // 12 18
```

Упражнение для читателя: попробуйте определить, сколько других комбинаций результатов можно получить при изменении порядка вызовов s1() и s2(). Не забудьте, что в программе всегда должны присутствовать три вызова s1() и четыре вызова s2(). Чтобы понять почему, вспомните приведенное выше обсуждение соответствия вызовов next() с yield.

Не следует намеренно создавать *такой* уровень путаницы с чередованием — понять такой код невероятно трудно. Но это интересное и поучительное упражнение поможет лучше понять, как несколько генераторов могут параллельно работать в одной общей области видимости, потому что в некоторых ситуациях эта возможность может оказаться весьма полезной.

Параллельное выполнение генераторов более подробно рассматривается в разделе «Параллельное выполнение генераторов» этой главы.

Генерирование значений

В предыдущем разделе упоминалось интересное применение генераторов для производства значений. Это не основная тема настоящей главы, но было бы неправильно не рассмотреть основные возможности, особенно если учесть, что от этого сценария использования происходит сам термин «*генератор*».

Сейчас мы ненадолго отвлечемся на тему *итераторов*, но потом вернемся к тому, как они связаны с генераторами и как использовать генератор для *генерирования* значений.

Производители и итераторы

Представьте, что вы создаете серию значений, в которой каждое значение связано определенным отношением с предыдущим значением. Для этого вам понадобится производитель с состоянием (stateful producer), который запоминает последнее выданное им значение.

Такие концепции можно тривиально выразить с использованием замыканий:

```
var gimmeSomething = (function(){
    var nextVal;
```

```
    return function(){
        if (nextVal === undefined) {
            nextVal = 1;
        }
        else {
            nextVal = (3 * nextVal) + 6;
        }

        return nextVal;
    };
})();

gimmeSomething();       // 1
gimmeSomething();       // 9
gimmeSomething();       // 33
gimmeSomething();       // 105
```

Возможно, логика вычисления `nextVal` здесь несколько упрощена, но на концептуальном уровне следующее значение (то есть `nextVal`) не должно вычисляться до следующего вызова `gimmeSomething()`, поскольку в общем случае такая архитектура может привести к утечке ресурсов для производителей значений, находящихся в долгосрочном хранении или более ограниченных по ресурсам, чем обычные числа.

Генерирование произвольных числовых рядов — не самый реалистичный пример. Но что, если вы генерируете записи по источнику данных? Код останется практически таким же.

На самом деле эта задача является чрезвычайно распространенным паттерном проектирования, который обычно решается при помощи итераторов. Итератор предоставляет четко определенный интерфейс для перебора серий значений от производителя. JS-интерфейс для итераторов, как и в большинстве языков, основан на вызове `next()` каждый раз, когда вы хотите получить следующее значение от производителя.

Мы можем реализовать стандартный интерфейс итераторов для производителя числовых рядов из приведенного примера:

```
var something = (function(){
    var nextVal;

    return {
        // необходимо для циклов `for..of`
        [Symbol.iterator]: function(){ return this; },

        // метод стандартного интерфейса итераторов
        next: function(){
            if (nextVal === undefined) {
                nextVal = 1;
            }
            else {
                nextVal = (3 * nextVal) + 6;
            }

            return { done:false, value:nextVal };
        }
    };
})();

something.next().value;     // 1
something.next().value;     // 9
something.next().value;     // 33
something.next().value;     // 105
```

Стоит объяснить, для чего нужна часть `[Symbol.iterator]: ..` во фрагменте кода из раздела «Итерируемые объекты» этой главы. С точки зрения синтаксиса здесь задействованы две возможности ES6. Во-первых, синтаксис `[ .. ]` называется «вычисляемым именем свойства». Этот механизм позволяет задать в определении объектного литерала выражение и использовать результат этого выражения как имя свойства. Во-вторых, `Symbol.iterator` — одно из предопределенных значений `Symbol` в ES6.

Вызов `next()` возвращает объект с двумя свойствами: `done` — значение `boolean`, обозначающее статус завершения итератора, а `value` содержит значение итерации.

В ES6 также был добавлен цикл `for..of`, который означает, что стандартный итератор может автоматически использоваться со встроенным синтаксисом цикла:

```
for (var v of something) {
    console.log( v );

    // цикл не должен работать бесконечно!
    if (v > 500) {
        break;
    }
}
// 1 9 33 105 321 969
```

Так как наш итератор `something` всегда возвращает `done:false`, этот цикл `for..of` будет выполняться бесконечно; по этой причине в него была включена условная команда break. Бесконечные итераторы — совершенно нормальное явление, но также возможны ситуации, в которых итератор перебирает конечный набор значений и в конечном итоге возвращает `done:true`.

Цикл `for..of` автоматически вызывает `next()` для каждой итерации (он не передает никакие значения `next()`) и автоматически завершается при получении `done:true`. Это достаточно удобно для перебора наборов данных.

Конечно, можно вручную организовать перебор по итераторам, вызывая `next()` и проверяя условие `done:true` для принятия решения о завершении:

```
for (
    var ret;
    (ret = something.next()) && !ret.done;
) {
    console.log( ret.value );

    // цикл не должен работать бесконечно!
    if (ret.value > 500) {
        break;
    }
}
// 1 9 33 105 321 969
```

Безусловно, ручная реализация `for` намного уродливее синтаксиса ES6 `for..of`, но у нее есть одно преимущество: она позволяет при необходимости передавать значения вызовам `next(..)`.

Кроме создания собственных *итераторов*, многие встроенные структуры данных JS (по состоянию ES6), например массивы, также имеют *итераторы* по умолчанию:

```
var a = [1,3,5,7,9];

for (var v of a) {
    console.log( v );
}
// 1 3 5 7 9
```

Цикл `for..of` запрашивает у массива `a` его *итератор* и автоматически использует его для перебора значений `a`.

Это может показаться странным упущением ES6, но обычные объекты намеренно не имеют итератора по умолчанию (который имеется у массивов). Причины такого решения глубже, чем можно было бы объяснить здесь. Если все, что вам нужно, — это перебор свойств объекта (без конкретных гарантий упорядочения), то метод `Object.keys(..)` возвращает массив, который может использоваться в форме `for (var k of Object.keys(obj)) { ...` Такой цикл `for..of` по ключам объекта будет аналогичен циклу `for..in`, не считая того, что `Object.keys(..)` не включает свойства из цепочки `[[Prototype]]`, а `for..in` их включает.

Итерируемые объекты

Объект `something` в нашем примере называется *итератором*, поскольку в его интерфейс входит метод `next()`. С этим понятием тесно связан термин *итерируемый объект* — это объект, содержащий *итератор*, который может использоваться для перебора его значений.

В ES6 для получения *итератора итерируемый объект* должен содержать функцию, имя которой задается специальным символическим значением ES6 `Symbol.iterator`. При вызове эта функция возвращает *итератор*. Обычно каждый вызов должен возвращать новый *итератор*, хотя это и не обязательно.

`a` в предыдущем фрагменте является *итерируемым объектом*. Цикл `for..of` автоматически вызывает свою функцию `Symbol.iterator` для конструирования *итератора*. Но конечно, вы также можете вызвать функцию вручную и использовать возвращенный ею *итератор*:

```
var a = [1,3,5,7,9];

var it = a[Symbol.iterator]();

it.next().value;    // 1
it.next().value;    // 3
it.next().value;    // 5
..
```

В предыдущем листинге с определением `something` вы могли заметить следующую строку:

```
[Symbol.iterator]: function(){ return this; }
```

Этот маленький, но запутанный фрагмент кода делает значение `something` — интерфейс *итератора* `something` — *итерируемым объектом*; в результате `something` становится как *итерируемым объектом*, так и *итератором*. После этого `something` передается циклу `for..of`:

```
for (var v of something) {
    ..
}
```

Цикл `for..of` ожидает, что `something` является итерируемым объектом, поэтому он находит и вызывает его функцию `Symbol.iterator`. Мы определили эту функцию, чтобы она просто воз-

вратила this, поэтому она возвращает сама себя, о чем цикл for.. of даже не подозревает.

Итераторы генераторов

А теперь обратимся к генераторам в контексте *итераторов*. Генератор может рассматриваться как производитель значений, которые извлекаются по одному через вызовы next() интерфейса итератора. Таким образом, сам генератор формально не является итерируемым объектом, хотя и очень похож на него — при выполнении генератора вы получите итератор:

```
function *foo(){ .. }

var it = foo();
```

Мы можем реализовать производителя бесконечных числовых рядов something (см. выше) на базе генератора:

```
function *something() {
    var nextVal;

    while (true) {
        if (nextVal === undefined) {
            nextVal = 1;
        }
        else {
            nextVal = (3 * nextVal) + 6;
        }

        yield nextVal;
    }
}
```

Обычно циклы while..true крайне не рекомендуется включать в реальные программы, по крайней мере если они не содержат команды break или return, так как этот цикл будет выполняться синхронно и полностью заблокирует пользовательский интерфейс браузера. Однако в генераторе такой цикл вполне допустим, если он содержит yield, поскольку генера-

> тор будет приостанавливаться при каждой итерации и возвращать управление основной программе и/или очереди цикла событий. Упрощенно говоря, «генераторы возвращают конструкцию `while..true` в программирование JS!»

Так намного элегантнее и проще, верно? Так как генератор приостанавливается при каждом `yield`, состояние (область видимости) функции `*something()` сохраняется, а следовательно, отпадает необходимость в шаблонном коде замыканий для сохранения состояния переменной между вызовами.

Код не только упрощается (нам не нужно создавать собственный интерфейс итератора), он становится более логичным, потому что он более четко выражает намерения программиста. Например, цикл `while..true` сообщает, что генератор должен работать бесконечно — продолжать генерировать значения, пока мы продолжаем их запрашивать.

А теперь новый генератор `*something()` можно использовать с циклом `for..of`; как вы увидите, он работает практически идентично:

```
for (var v of something()) {
    console.log( v );

    // цикл не должен работать бесконечно!
    if (v > 500) {
        break;
    }
}
// 1 9 33 105 321 969
```

Обратите особое внимание на `for (var v of something())`! Мы не просто ссылаемся на `something` как на значение, как в предыдущих примерах, а вызываем генератор `*something()` для получения его итератора, чтобы использовать его в цикле `for..of`.

Если вы достаточно внимательно следите за происходящим, из этих взаимодействий между генератором и циклом могут возникнуть два вопроса:

- Почему нельзя использовать запись `for (var v of something)`..? Потому что `something` — генератор, а он не является *итерируемым объектом*. Необходимо вызвать `something()`, чтобы сконструировать производителя для перебора в цикле `for..of`.
- Вызов `something()` производит *итератор*, но цикл `for..of` должен получить *итерируемый объект*, верно? Да. *Итератор* генератора также содержит функцию `Symbol.iterator`, которая, по сути, возвращает `this`, как и итерируемый объект `something`, определенный ранее. Другими словами, *итератор* генератора также является *итерируемым объектом*!

Остановка генератора

В предыдущем примере все выглядит так, словно экземпляр итератора для генератора `*something()` фактически навсегда остается в приостановленном состоянии после выполнения `break` в цикле.

Однако существует скрытое поведение, которое решит эту проблему. «Аномальное завершение» (то есть «раннее прерывание») цикла `for..of`, обычно обусловленное `break`, `return` или неперехваченным исключением, отправляет итератору генератора сигнал завершения.

С технической точки зрения цикл for..of также отправляет этот сигнал итератору при нормальном завершении цикла. Для генератора это, по сути, пустая операция, так как для завершения цикла `for..of` сначала должен завершиться итератор генератора. Однако пользовательские итераторы могут захотеть получить этот дополнительный сигнал от сторон, использующих цикл `for..of`.

Хотя цикл `for..of` отправляет этот сигнал автоматически, возможно, вы захотите отправить этот сигнал *итератору* вручную; это делается вызовом `return(..)`.

Если внутри генератора находится секция `try..finally`, она будет выполняться всегда, даже при внешнем завершении генератора.

Например, это может быть полезно в тех случаях, если вам требуется освободить ресурсы (подключения к базам данных и т. д.):

```
function *something() {
    try {
        var nextVal;

        while (true) {
            if (nextVal === undefined) {
                nextVal = 1;
            }
            else {
                nextVal = (3 * nextVal) + 6;
            }

            yield nextVal;
        }
    }
    // завершающая секция
    finally {
        console.log( "cleaning up!" );
    }
}
```

В более раннем примере команда `break` в цикле `for..of` инициирует выполнение секции `finally`. Но вы также могли вручную завершить экземпляр итератора генератора извне вызовом `return(..)`:

```
var it = something();
for (var v of it) {
    console.log( v );

    // цикл не должен работать бесконечно!
    if (v > 500) {
        console.log(
            // завершить итератор генератора
            it.return( "Hello World" ).value
        );
        // команда `break` здесь не нужна
    }
}
```

```
// 1 9 33 105 321 969
// cleaning up!
// Hello World
```

Вызов `it.return(..)` немедленно завершает генератор, что, конечно, приводит к выполнению секции `finally`. Кроме того, он присваивает возвращенное значение тому, что было передано `return(..)`; так выводится строка `"Hello World"`. Также теперь не нужно включать команду `break`, потому что итератору генератора присваивается `done:true`, а цикл `for..of` завершается при следующей итерации.

Генераторы получили свое название в основном из-за *потребления произведенных значений*. Тем не менее это лишь один из вариантов использования генераторов, и откровенно говоря, даже не самый главный из интересующих нас в контексте этой книги.

Но теперь, когда вы лучше понимаете некоторые механизмы того, как работают генераторы, мы можем обратиться к теме применения генераторов в рамках асинхронного параллельного выполнения.

Асинхронный перебор итераторов

Какое отношение имеют генераторы к паттернам асинхронного программирования, решению проблем обратных вызовов и т. д.? Пора ответить на этот важный вопрос.

Вернемся к одному из сценариев из главы 3. Вспомним решение с обратными вызовами:

```
function foo(x,y,cb) {
    ajax(
        "http://some.url.1/?x=" + x + "&y=" + y,
        cb
    );
}

foo( 11, 31, function(err,text) {
    if (err) {
        console.error( err );
```

```
    }
    else {
        console.log( text );
    }
} );
```

Если вы захотите выразить аналогичное управление программной логикой с генератором, это можно сделать так:

```
function foo(x,y) {
    ajax(
        "http://some.url.1/?x=" + x + "&y=" + y,
        function(err,data){
            if (err) {
                // выдать ошибку в `*main()`
                it.throw( err );
            }
            else {
                // возобновить `*main()` с полученными `data`
                it.next( data );
            }
        }
    );
}

function *main() {
    try {
        var text = yield foo( 11, 31 );
        console.log( text );
    }
    catch (err) {
        console.error( err );
    }
}

var it = main();

// запустить все!
it.next();
```

На первый взгляд этот фрагмент длиннее приведенного ранее фрагмента с обратными вызовами и, возможно, сложнее выглядит.

Но это впечатление обманчиво. Фрагмент с генератором на самом деле гораздо лучше! Однако здесь происходит много всего, что необходимо объяснить.

Начнем со следующей, самой важной части кода:

```
var text = yield foo( 11, 31 );
console.log( text );
```

Подумайте, как работает этот код. Мы вызываем обычную функцию `foo(..)` и очевидно можем получить `text` от вызова Ajax, несмотря на то что он является асинхронным.

Как такое возможно? Если вспомнить начало главы 1, там встречался почти идентичный код:

```
var data = ajax( "..url 1.." );
console.log( data );
```

И этот код не работал! А вы видите, чем отличаются эти два случая? Командой `yield`, используемой в генераторе.

Вот она, магия! Вот что позволяет нам писать код, который выглядит блокирующим и синхронным, но при этом не блокирует всю программу, он только приостанавливает/блокирует код в самом генераторе.

В `yield foo(11,31)` сначала делается вызов `foo(11,31)`, который не возвращает ничего (то есть `undefined`), так что мы выдаем вызов для запроса данных, но потом фактически выполняем `yield undefined`. Это нормально, потому что код не делает ничего интересного со значением, выданным `yield`. Мы еще вернемся к этой теме позднее в этой главе.

Здесь `yield` используется не для передачи сообщений, а только в смысле управления программной логикой для приостановки/блокировки. На самом деле сообщения будут передаваться, но только в одном направлении после возобновления работы генератора.

Таким образом, генератор приостанавливается в точке `yield`, фактически задавая вопрос: «Какое значение мне следует вернуть для присваивания переменной `text`?» Кто будет отвечать на этот вопрос?

Взгляните на `foo(..)`. Если запрос Ajax был успешным, следует вызов:

```
it.next( data );
```

Выполнение генератора возобновляется с данными ответа; это означает, что приостановленное выражение `yield` получает это значение напрямую, а затем при перезапуске кода генератора это значение присваивается локальной переменной `text`.

Впечатляет, верно?

Отступим на шаг назад и подумаем о последствиях. Генератор содержит код, который выглядит совершенно синхронно (если не считать самого ключевого слова `yield`), но «за кулисами» внутри `foo(..)` операции могут быть полностью асинхронными.

Замечательно! Это почти идеальное решение для ранее описанной проблемы с обратными вызовами, не способными выражать асинхронность в последовательной, синхронной манере, привычной для нашего мозга.

По сути, асинхронность абстрагируется как подробность реализации, так что мы можем описывать программную логику синхронно/последовательно: «Выдать запрос Ajax, а после его завершения вывести ответ». И конечно, мы описали только два шага программной логики, но эту возможность можно неограниченно расширять для описания любого нужного количества шагов.

Это очень важный вывод. Вернитесь назад и перечитайте три последних абзаца, чтобы информация как следует закрепилась у вас в голове!

Синхронная обработка ошибок

Однако это не все, что вам может предложить приведенный код генератора. Обратите внимание на секцию `try..catch` внутри генератора:

```
try {
    var text = yield foo( 11, 31 );
    console.log( text );
}
catch (err) {
    console.error( err );
}
```

Как это работает? Вызов `foo(..)` завершается асинхронно, но разве `try..catch` умеет перехватывать асинхронные ошибки (см. главу 3)?

Вы уже видели, как с `yield` команда присваивания приостанавливается, ожидая завершения `foo(..)`, чтобы завершенный ответ мог быть присвоен `text`. Самое замечательное здесь то, что приостановка `yield` *также* позволяет генератору перехватить ошибку. Ошибка запускается в генератор следующей частью приведенного выше кода:

```
if (err) {
    // выдать ошибку в `*main()`
    it.throw( err );
}
```

Принцип работы генераторов «`yield`-приостановка» означает, что мы не только получаем синхронно выглядящие возвращаемые значения от асинхронных вызовов функций, но и можем синхронно перехватывать ошибки от этих асинхронных вызовов функций!

Вы увидели, как запустить ошибку *в* генератор, но как насчет запуска ошибок *из* генератора? Так, как и следовало ожидать:

```
function *main() {
    var x = yield "Hello World";
```

```
    yield x.toLowerCase();  // породить исключение!
}

var it = main();

it.next().value;            // Hello World
try {
    it.next( 42 );
}
catch (err) {
    console.error( err );   // TypeError
}
```

Конечно, можно было бы вручную инициировать ошибку командой `throw ..`, вместо того чтобы провоцировать исключение.

Мы даже можем перехватить ту же ошибку, которая была запущена в генератор вызовом `throw(..)`, фактически предоставляя генератору возможность обработать ее; если генератор этого не сделает, то обработкой должен заняться код *итератора*:

```
function *main() {
    var x = yield "Hello World";

    // управление сюда не передается
    console.log( x );
}

var it = main();

it.next();

try {
    // обработает ли эту ошибку `*main()`? посмотрим!
    it.throw( "Oops" );
}
catch (err) {
    // нет, не обработает!
    console.error( err );           // Неудача
}
```

Обработка ошибок синхронного вида (с `try..catch`) с асинхронным кодом существенно улучшает удобочитаемость и анализ кода.

Генераторы + обещания

В предыдущем обсуждении мы показали, как организовать итеративное выполнение генераторов, которое становится огромным шагом вперед для последовательного анализа по сравнению с месивом обратных вызовов. Но при этом теряется нечто очень важное: гарантии доверия и средства композиции обещаний (см. главу 3).

Не беспокойтесь, все это можно вернуть. ES6 позволяет объединить лучшие стороны обоих миров: генераторы (асинхронный код, который синхронно выглядит) с обещаниями (гарантии доверия и средства композиции).

Но как?

Вспомните наш подход к примеру Ajax на основе обещаний, представленный в главе 3:

```
function foo(x,y) {
    return request(
        "http://some.url.1/?x=" + x + "&y=" + y
    );
}

foo( 11, 31 )
.then(
    function(text){
        console.log( text );
    },
    function(err){
        console.error( err );
    }
);
```

В более раннем коде генератора из примера Ajax функция `foo(..)` возвращала ничто (`undefined`), а управляющий код *итератора* игнорировал значение, переданное `yield`.

Но здесь `foo(..)` с поддержкой обещаний возвращает обещание после вызова Ajax. Это наводит на мысль, что мы можем сконстру-

ировать обещание в `foo(..)`, а затем передать его через `yield` из генератора, и тогда управляющий код *итератора* получит это обещание.

Но что *итератор* должен сделать с обещанием?

Он должен прослушивать разрешение обещания (выполнение или отказ), после чего либо возобновить выполнение генератора с сообщением выполнения, либо запустить ошибку в генератор с причиной отказа.

Я еще раз повторю, потому что это очень важно. Естественный способ извлечь максимум пользы из обещаний и генераторов — выдать обещание через `yield` и подключить это обещание для управления итератором генератора.

Давайте попробуем! Начнем с объединения `foo(..)` с поддержкой обещаний с генератором `*main()`:

```
function foo(x,y) {
    return request(
        "http://some.url.1/?x=" + x + "&y=" + y
    );
}

function *main() {
    try {
        var text = yield foo( 11, 31 );
        console.log( text );
    }
    catch (err) {
        console.error( err );
    }
}
```

Самое замечательное, на что стоит обратить внимание в этом рефакторинге: код внутри `*main()` вообще не изменился! Внутри генератора выдаваемые `yield` значения являются непрозрачной подробностью реализации, так что мы даже не видим, что это происходит, да нам и не нужно об этом беспокоиться.

Но как теперь запустить `*main()`? Нам все еще нужно поработать над некоторыми служебными аспектами реализации: получением и связыванием обещания, переданного `yield`, чтобы работа генератора возобновлялась при разрешении. Для начала попробуем это сделать вручную:

```
var it = main();

var p = it.next().value;

// ожидать разрешения обещания `p`
p.then(
    function(text){
        it.next( text );
    },
    function(err){
        it.throw( err );
    }
);
```

Не так уж страшно, верно?

Этот фрагмент очень похож на то, что делалось ранее при ручном подключении генератора, управляемого обратным вызовом по принципу «ошибка на первом месте». Вместо `if (err) { it.throw..` обещание уже отделяет выполнение (успех) от отказа (неудача), но в остальном управление итератором не отличается.

Тем не менее такое решение замалчивает некоторые важные подробности.

Прежде всего мы воспользовались важным фактом: мы знали, что `*main()` содержит только один шаг с поддержкой обещаний. А что, если бы мы захотели управлять генератором с использованием обещаний независимо от числа шагов? Конечно, вам не захочется вручную писать разные цепочки обещаний для каждого генератора! Было бы намного удобнее, если бы существовал способ повторения (то есть циклического выполнения) управления

итерациями, и каждый раз, когда появляется обещание, он бы дождался его разрешения перед продолжением работы.

Кроме того, что должно происходить, если генератор выдаст ошибку (случайно или намеренно) при следующем вызове `it.next(..)`? Завершить работу или перехватить ошибку и вернуть ее обратно? Или что делать, если мы запускаем в генератор отказ через `it.throw(..)`, но он не будет обработан и просто вернется назад?

Выполнение генератора с поддержкой обещаний

Чем больше вы изучаете этот путь, тем больше вы осознаете: «А как было бы здорово, если бы были встроенные средства, которые делали бы это за меня!» И вы абсолютно правы. Это очень важный паттерн, и вам не хочется ошибиться (или устать от его бесконечных повторений), так что лучше всего воспользоваться какой-нибудь функцией, специально спроектированной для *выполнения* генераторов, выдающих обещания, описанным мной способом.

Такую возможность предоставляют некоторые библиотеки абстракции обещаний, включая мою библиотеку `asynquence` и ее функцию `runner(..)` (см. приложение А).

Тем не менее в учебных целях и для демонстрации давайте определим собственную функцию с именем `run(..)`:

```
// спасибо Бенджамину Грюнбауму (@benjamingr на GitHub)
// за существенные улучшения!
function run(gen) {
    var args = [].slice.call( arguments, 1), it;

    // инициализировать генератор в текущем контексте
    it = gen.apply( this, args );

    // вернуть обещание для завершающегося генератора
```

```
    return Promise.resolve()
        .then( function handleNext(value){
            // выполнить до следующего значения, переданного
            // yield
            var next = it.next( value );
            return (function handleResult(next){
                // генератор завершил выполнение?
                if (next.done) {
                    return next.value;
                }
                // в противном случае продолжить
                else {
                    return Promise.resolve( next.value )
                        .then(
                            // возобновить асинхронный цикл
                            // в случае успеха и отправить
                            // значение, полученное в результате
                            // разрешения, обратно генератору
                            handleNext,

                            // если `value` - отклоненное
                            // обещание, распространить ошибку
                            // обратно в генератор для
                            // собственной обработки ошибок
                            function handleErr(err) {
                                return Promise.resolve(
                                    it.throw( err )
                                )
                                .then( handleResult );
                            }
                        );
                }
            })(next);
        } );
}
```

Как видите, код получается слишком сложным, чтобы вам хотелось писать его самостоятельно и еще меньше — повторять его для каждого используемого вами генератора. Таким образом, вспомогательная функция/библиотека, безусловно, пригодилась бы. Тем не менее я рекомендую выделить несколько минут

на изучение этого кода, чтобы вы лучше разобрались в том, как управлять взаимодействием «генератор + обещания».

Как бы вы использовали `run(..)` с `*main()` в нашем примере с Ajax?

```
function *main() {
    // ..
}

run( main );
```

Вот и все! При таком способе подключения `run(..)` будет автоматически продвигать переданный генератор в асинхронном режиме до завершения.

Функция `run(..)`, определенная нами, возвращает обещание, которое разрешается при завершении генератора, или получает неперехваченное исключение, если оно не было обработано генератором. Здесь эта возможность не продемонстрирована, но мы еще вернемся к ней позднее в этой главе.

ES7: async и await?

Описанный паттерн — генераторы, выдающие обещания, которые затем управляют итератором генератора для продвижения его к завершению, — настолько мощен и полезен, что было бы удобно пользоваться им без балласта в виде библиотечной вспомогательной функции (то есть `run(..)`).

И возможно, по этой части есть хорошие новости. На момент написания книги существовала ранняя, но достаточно сильная поддержка предложения по добавлению нового синтаксиса после ES6 (вероятно, во временном окне ES7).

Очевидно, пока слишком рано давать какие-то гарантии подробностей, но по всей вероятности, это будет выглядеть примерно так:

```
function foo(x,y) {
    return request(
        "http://some.url.1/?x=" + x + "&y=" + y
    );
}

async function main() {
    try {
        var text = await foo( 11, 31 );
        console.log( text );
    }
    catch (err) {
        console.error( err );
    }
}

main();
```

Как видите, для запуска и управления `main()` не нужен вызов `run(..)` (а значит, не нужна и библиотечная поддержка!) — `main()` просто вызывается как обычная функция. Кроме того, `main()` более не объявляется как функция-генератор; это новая разновидность функций — асинхронная функция (`async function`). И наконец, вместо того чтобы передавать обещание через `yield`, мы просто ожидаем его завершения при помощи `await`.

Функция `async function` уже знает, что нужно делать при ожидании обещания, — она приостанавливает функцию (как в случае с генераторами) до разрешения обещания. В приведенном фрагменте это не показано, но вызов такой асинхронной функции, как `main()`, автоматически возвращает обещание, которое разрешается при полном завершении функции.

Синтаксис async/await должен быть хорошо знаком читателям с опытом программирования на C#. В этом языке он устроен практически идентично.

Предложение фактически закрепляет поддержку паттерна, который мы уже разработали, в синтаксический механизм: объедине-

ние обещаний с кодом управления программной логикой, который выглядит как синхронный. Таким образом лучшие стороны двух миров объединяются для решения практически всех основных проблем, присущих обратным вызовам.

Сам факт того, что предложение для ES7 уже существует и пользуется поддержкой и энтузиазмом на ранней стадии, — важный фактор уверенности в будущем этого асинхронного паттерна.

Параллелизм обещаний в генераторах

До настоящего момента мы продемонстрировали одношаговую асинхронную программную логику на базе связки «обещания + генераторы». Тем не менее реальный код часто содержит много асинхронных шагов.

Если вы будете недостаточно внимательны, генераторы в синхронном стиле могут породить беспечное отношение к структуре асинхронного параллелизма и привести к формированию паттернов, не оптимальных по эффективности. По этой причине выделим немного времени на анализ альтернатив.

Представьте ситуацию, в которой вам требуется получить данные из двух разных источников, затем объединить полученные ответы для создания третьего запроса и, наконец, вывести последний ответ. Мы исследовали аналогичный сценарий с обещаниями в главе 3, но давайте снова рассмотрим его в контексте генераторов.

Первое инстинктивное решение может выглядеть примерно так:

```
function *foo() {
    var r1 = yield request( "http://some.url.1" );
    var r2 = yield request( "http://some.url.2" );

    var r3 = yield request(
        "http://some.url.3/?v=" + r1 + "," + r2
    );

    console.log( r3 );
```

```
}

// использовать ранее определенную функцию `run(..)`
run( foo );
```

Этот код будет работать, но в специфике нашего сценария он не оптимален. А вы понимаете почему?

Потому что запросы `r1` и `r2` могут — и по соображениям быстродействия *должны* — выполняться параллельно, но в этом коде они будут выполняться последовательно; URL-адрес `"http://some.url.2"` не будет загружен средствами Ajax, пока не будет завершен запрос `"http://some.url.1"`. Эти два запроса не зависят друг от друга, так что для повышения эффективности, скорее всего, лучше выполнять их одновременно.

Но как именно это сделать с генератором и `yield`? Мы знаем, что `yield` — всего лишь одна точка приостановки в коде, так что вам не удастся выполнить две приостановки одновременно.

Самый естественный и эффективный ответ — реализовать асинхронную программную логику на базе обещаний, а именно на их способности управлять состоянием независимо от времени (см. раздел «Будущее значение» главы 3). Простейший вариант:

```
function *foo() {
    // выполнить оба запроса "параллельно"
    var p1 = request( "http://some.url.1" );
    var p2 = request( "http://some.url.2" );

    // ожидать разрешения обоих обещаний
    var r1 = yield p1;
    var r2 = yield p2;

    var r3 = yield request(
        "http://some.url.3/?v=" + r1 + "," + r2
    );

    console.log( r3 );
}
```

```
// использовать ранее определенную функцию `run(..)`
run( foo );
```

Чем это отличается от предыдущего фрагмента? Взгляните, где располагается (или не располагается) `yield`. `p1` и `p2` — обещания для запросов Ajax, выполняемых параллельно. Неважно, какой из них завершится первым, потому что обещания будут сохранять свое состояние разрешения столько, сколько понадобится.

Затем мы используем две последующие команды `yield` для ожидания и получения разрешений от обещаний (в `r1` и `r2` соответственно). Если `p1` разрешится первым, то `yield p1` сначала возобновляет работу, а затем ожидает возобновления `yield p2`. Если обещание `p2` разрешится первым, то оно просто терпеливо удерживает это значение разрешения до того, как оно будет востребовано, но первым будет удерживаться результат `yield p1` до разрешения `p1`.

В любом случае `p1` и `p2` выполняются параллельно и должны завершиться (в произвольном порядке), чтобы был выдан запрос Ajax `r3 = yield request...`

Такая модель управления программной логикой кажется знакомой — по сути, она не отличается от той, которая была представлена в главе 3 как паттерн «шлюз» (на базе `Promise.all([ .. ])`). Программную логику можно выразить так:

```
function *foo() {
    // выполнить оба запроса "параллельно"
    // и ожидать разрешения обоих обещаний
    var results = yield Promise.all( [
        request( "http://some.url.1" ),
        request( "http://some.url.2" )
    ] );

    var r1 = results[0];
    var r2 = results[1];

    var r3 = yield request(
        "http://some.url.3/?v=" + r1 + "," + r2
```

```
    );

    console.log( r3 );
}

// использовать ранее определенную функцию `run(..)`
run( foo );
```

Как обсуждалось в главе 3, деструктурирующее присваивание ES6 позволяет упростить присваивания `var r1 = .. var r2 = ..` и привести их к виду `var [r1,r2] = results`.

Другими словами, все возможности параллельного выполнения обещаний доступны в решениях «генератор+обещания». Таким образом, в любом месте, где вам потребуется нечто большее, чем последовательное асинхронное управление программной логикой «одно-потом-другое», обещания с большой вероятностью окажутся наилучшим вариантом.

Скрытые обещания

Небольшое стилистическое предупреждение: будьте внимательны к тому, какое количество логики обещаний вы размещаете *внутри своих генераторов*. Вся суть использования генераторов для асинхронного выполнения описанным образом — это создание простого, последовательного, синхронно выглядящего кода и максимально возможная изоляция подробностей асинхронности от кода:

Например, следующее решение может быть более элегантным:

```
// обратите внимание: обычная функция, не генератор
function bar(url1,url2) {
    return Promise.all( [
        request( url1 ),
        request( url2 )
    ] );
}

function *foo() {
```

```
    // скрыть подробности параллельного выполнения на базе
    // обещаний внутри `bar(..)`
    var results = yield bar(
        "http://some.url.1",
        "http://some.url.2"
    );

    var r1 = results[0];
    var r2 = results[1];

    var r3 = yield request(
        "http://some.url.3/?v=" + r1 + "," + r2
    );

    console.log( r3 );
}

// использовать ранее определенную функцию `run(..)`
run( foo );
```

Внутри `*foo()` все выглядит чище и яснее: вы всего лишь требуете у `bar(..)` предоставить результаты, а затем ожидаете с `yield`, когда это произойдет. Вам не нужно беспокоиться о том, что во внутренней реализации для этого будет использована композиция `Promise.all([ .. ])`.

И асинхронность, и обещания рассматриваются как подробности реализации.

Сокрытие логики обещаний внутри функции, которая просто вызывается из генератора, особенно полезно в том случае, если вы собираетесь управлять программной логикой сложной последовательности действий.

Пример:

```
function bar() {
    Promise.all( [
        baz( .. )
        .then( .. ),
        Promise.race( [ .. ] )
    ] )
    .then( .. )
}
```

Иногда такая логика бывает необходимой, а вынесение ее непосредственно в генератор(ы) противоречит многим причинам, по которым вы изначально используете генераторы. Такие подробности *должны* намеренно абстрагироваться из кода генератора, чтобы они не загромождали выражение задач более высокого уровня.

Кроме создания кода как функционального, так и эффективного, вы также должны стремиться к тому, чтобы ваш код был максимально простым для анализа и сопровождения.

Абстракция не *всегда* однозначно желательна в программировании, нередко она только повышает сложность в обмен на компактность. Но в данном случае я считаю, что это решение намного здоровее для вашего асинхронного кода «генератор + обещания», чем альтернативы. Впрочем, как и при всех таких рекомендациях, всегда обращайте внимание на конкретные ситуации и принимайте правильные решения для вас и вашей команды.

Делегирование

В предыдущем разделе мы показали, как вызывать обычные функции из генератора и какую пользу этот метод приносит для абстрагирования подробностей реализации (таких, как асинхронная программная логика обещаний). Но главный недостаток применения нормальных функций для таких задач заключается в том, что они должны вести себя по правилам обычных функций, а это означает, что они не могут приостанавливать себя командой `yield`, как это делают генераторы.

Возможно, вам покажется, что один генератор можно попытаться вызвать из другого генератора с использованием вспомогательной функции `run(..)`:

```
function *foo() {
    var r2 = yield request( "http://some.url.2" );
    var r3 = yield request( "http://some.url.3/?v=" + r2 );

    return r3;
}

function *bar() {
    var r1 = yield request( "http://some.url.1" );

    // "делегирование" `*foo()` с использованием `run(..)`
    var r3 = yield run( foo );
    console.log( r3 );
}

run( bar );
```

Мы запускаем `*foo()` внутри `*bar()`, для чего снова задействуется функция `run(..)`. Здесь используется тот факт, что функция `run(..)`, определенная ранее, возвращает обещание, которое разрешается при отработке генератора до завершения (или из-за ошибки), так что если выдать экземпляру `run(..)` командой `yield` обещание из другого вызова `run(..)`, оно автоматически приостановит `*bar()` до завершения `*foo()`.

Но существует лучший способ интеграции вызовов `*foo()` в `*bar()`, и он называется `yield`-*делегированием*. `yield`-делегирование имеет специальный синтаксис `yield * __` (обратите внимание на появление `*`). Прежде чем мы увидим, как работает объект в предыдущем примере, рассмотрим более простой сценарий:

```
function *foo() {
    console.log( "`*foo()` starting" );
    yield 3;
    yield 4;
    console.log( "`*foo()` finished" );
}

function *bar() {
    yield 1;
```

```
    yield 2;
    yield *foo();   // `yield`-делегирование!
    yield 5;
}

var it = bar();

it.next().value;    // 1
it.next().value;    // 2
it.next().value;    // `*foo()` запускается
                    // 3
it.next().value;    // 4
it.next().value;    // `*foo()` завершается
                    // 5
```

В одном из предыдущих примечаний я объяснял, почему я предпочитаю `function *foo() ..` вместо `function* foo() ...` Также я предпочитаю — в отличие от большинства документации по теме — использовать запись `yield *foo()` вместо `yield* foo()`. Расположение * — чисто стилистический аспект, и решение следует принимать, руководствуясь здравым смыслом. Лично мне нравится последовательный характер стилевого оформления.

Как работает делегирование `yield *foo()`?

Сначала вызов `foo()` создает *итератор* точно так же, как вы уже видели ранее. Затем `yield *` делегирует/передает управление экземпляром *итератора* (текущего генератора `*bar()`) другому *итератору* `*foo()`.

Таким образом, первые два вызова `it.next()` управляют `*bar()`, но при третьем вызове `it.next()` запускается `*foo()`, и теперь мы уже управляем `*foo()` вместо `*bar()`. Вот почему это называется делегированием — `*bar()` делегирует свое управление итерациями `*foo()`.

Как только управление итератором `it` исчерпывает весь *итератор* `*foo()`, он автоматически возвращается к управлению `*bar()`.

Вернемся к предыдущему примеру с тремя последовательными запросами Ajax:

```
function *foo() {
    var r2 = yield request( "http://some.url.2" );
    var r3 = yield request( "http://some.url.3/?v=" + r2 );
    return r3;
}

function *bar() {
    var r1 = yield request( "http://some.url.1" );

    // "делегирование" `*foo()` с использованием `run(..)`
    var r3 = yield *foo();

    console.log( r3 );
}

run( bar );
```

Единственное различие между этим фрагментом и версией, использовавшейся ранее, — применение `yield *foo()` вместо `yield run(foo)`.

`yield *` уступает управление итерациями, а не управление генератором; когда вы вызываете генератор `*foo()`, вы теперь `yield`-делегируете ее итератору. Но вы можете `yield`-делегировать любому итерируемому объекту; `yield *[1,2,3]` будет потреблять итератор по умолчанию для массива `[1,2,3]`.

Почему делегирование?

`Yield`-делегирование предназначено в основном для организации кода; в этом отношении оно симметрично с нормальным вызовом функций.

Представьте два модуля, которые предоставляют соответственно методы `foo()` и `bar()`, при этом `bar()` вызывает `foo()`. Они существуют раздельно прежде всего потому, что этого требует правиль-

ная организация кода программы. Например, в некоторых случаях `foo()` может вызываться автономно, а в других — когда `bar()` вызывает `foo()`.

По тем же самым причинам разделение генераторов упрощает чтение, сопровождение и отладку программы. В этом отношении `yield *` является синтаксическим сокращением для ручного перебора шагов `*foo()` во время нахождения внутри `*bar()`.

Такие ручные решения будут особенно сложными, если шаги в `*foo()` асинхронны; поэтому в подобной ситуации, вероятно, лучше воспользоваться `run(..)`. Как мы уже показали, `yield *foo()` избавляет от необходимости в подэкземпляре `run(..)` (например, `run(foo)`).

Делегирование сообщений

Возможно, вас интересует, как `yield`-делегирование работает не только с управлением итератором, но и с двусторонней передачей сообщений? Внимательно проследите за передачей сообщений входа/выхода с использованием `yield`-делегирования:

```
function *foo() {
    console.log( "inside `*foo()`:", yield "B" );

    console.log( "inside `*foo()`:", yield "C" );

    return "D";
}

function *bar() {
    console.log( "inside `*bar()`:", yield "A" );

    // `yield`-делегирование!
    console.log( "inside `*bar()`:", yield *foo() );

    console.log( "inside `*bar()`:", yield "E" );

    return "F";
```

```
}

var it = bar();

console.log( "outside:", it.next().value );
// outside: A

console.log( "outside:", it.next( 1 ).value );
// inside `*bar()`: 1
// outside: B

console.log( "outside:", it.next( 2 ).value );
// inside `*foo()`: 2
// outside: C

console.log( "outside:", it.next( 3 ).value );
// inside `*foo()`: 3
// inside `*bar()`: D
// outside: E

console.log( "outside:", it.next( 4 ).value );
// inside `*bar()`: 4
// outside: F
```

Обратите особое внимание на действия по обработке после вызова `it.next(3)`:

1. Значение `3` передается (через `yield`-делегирование в `*bar()`) ожидающему выражению `yield "C"` внутри `*foo()`.
2. Затем `*foo()` вызывает `return "D"`, но это значение не возвращается внешнему вызову `it.next(3)`.
3. Вместо этого значение `"D"` отправляется как результат ожидающего выражения `yield *foo()` внутри `*bar()` — это выражение `yield`-делегирования фактически было приостановлено до полного исчерпания `*foo()`. Таким образом, `"D"` попадает внутрь `*bar()` для вывода.
4. `yield "E"` вызывается внутри `*bar()`, и значение `"E"` передается наружу как результат вызова `it.next(3)`.

С точки зрения внешнего *итератора* (`it`) никаких различий между управлением исходным и делегированным генератором нет.

Собственно, `yield`-делегирование даже не обязательно направлять другому генератору; оно также может быть направлено обобщенному *итерируемому объекту*. Пример:

```
function *bar() {
    console.log( "inside `*bar()`:", yield "A" );

    // `yield`-делегирование не-генератору!
    console.log( "inside `*bar()`:", yield *[ "B", "C", "D" ] );

    console.log( "inside `*bar()`:", yield "E" );

    return "F";
}

var it = bar();

console.log( "outside:", it.next().value );
// outside: A

console.log( "outside:", it.next( 1 ).value );
// inside `*bar()`: 1
// outside: B

console.log( "outside:", it.next( 2 ).value );
// outside: C

console.log( "outside:", it.next( 3 ).value );
// outside: D

console.log( "outside:", it.next( 4 ).value );
// inside `*bar()`: undefined
// outside: E

console.log( "outside:", it.next( 5 ).value );
// inside `*bar()`: 5
// outside: F
```

Обратите внимание на различия в точках получения/выдачи сообщений между этим примером и предыдущим.

Самое замечательное, что итератор массива по умолчанию не обращает внимания на сообщения, отправляемые вызовами `next(..)`, так что значения 2, 3 и 4 фактически игнорируются. Кроме того, поскольку итератор не имеет явного возвращаемого значения (в отличие от ранее использованной версии `*foo()`), выражение `yield *` получает `undefined` при завершении.

Исключения тоже делегируются!

По аналогии с тем, как `yield`-делегирование прозрачно передает сообщения в обоих направлениях, ошибки/исключения тоже проходят в обоих направлениях:

```
function *foo() {
    try {
        yield "B";
    }
    catch (err) {
        console.log( "error caught inside `*foo()`:", err );
    }

    yield "C";

    throw "D";
}

function *bar() {
    yield "A";

    try {
        yield *foo();
    }
    catch (err) {
        console.log( "error caught inside `*bar()`:", err );
    }

    yield "E";
    yield *baz();

    // примечание: сюда управление не передается!
    yield "G";
```

```
}

function *baz() {
    throw "F";
}

var it = bar();

console.log( "outside:", it.next().value );
// outside: A

console.log( "outside:", it.next( 1 ).value );
// outside: B

console.log( "outside:", it.throw( 2 ).value );
// error caught inside `*foo()`: 2
// outside: C

console.log( "outside:", it.next( 3 ).value );
// error caught inside `*bar()`: D
// outside: E

try {
    console.log( "outside:", it.next( 4 ).value );
}
catch (err) {
    console.log( "error caught outside:", err );
}
// error caught outside: F
```

В этом коде следует обратить внимание на ряд обстоятельств:

1. Когда мы вызываем `it.throw(2)`, этот вызов отправляет сообщение об ошибке `2` в `*bar()`, который делегирует его `*foo()`, где оно корректно перехватывается и обрабатывается. Затем `yield "C"` отправляет `"C"` обратно как возвращаемое значение от вызова `it.throw(2)`.

2. Значение `"D"`, которое затем выдается из `*foo()`, распространяется в `*bar()`, где оно перехватывается и корректно обрабатывается. Затем `yield "E"` отправляет `"E"` обратно как возвращаемое значение от вызова `it.throw(3)`.

3. Далее исключение, выданное из `*baz()`, не перехватывается в `*bar()` (хотя оно было перехвачено снаружи), так что `*baz()` и `*bar()` переводятся в завершенное состояние. После этого фрагмента вы не сможете получить значение `"G"` от последующих вызовов `next(..)`, они будут просто возвращать для `value` значение `undefined`.

Делегирование асинхронности

А теперь наконец-то вернемся к примеру `yield`-делегирования с несколькими последовательными запросами Ajax:

```
function *foo() {
    var r2 = yield request( "http://some.url.2" );
    var r3 = yield request( "http://some.url.3/?v=" + r2 );

    return r3;
}

function *bar() {
    var r1 = yield request( "http://some.url.1" );

    var r3 = yield *foo();

    console.log( r3 );
}

run( bar );
```

Вместо вызова `yield run(foo)` внутри `*bar()` мы просто вызываем `yield *foo()`.

В предыдущей версии этого примера механизм обещаний (управляемый `run(..)`) использовался для передачи значения из `return r3` в `*foo()` в локальную переменную `r3` внутри `*bar()`. Теперь это значение просто возвращается напрямую через механику `yield *`.

В остальном поведение остается почти неизменным.

Делегирование рекурсии

Конечно, `yield`-делегирование может пройти столько шагов делегирования, сколько вы подключите. Оно даже может использоваться для *рекурсии* генераторов с асинхронностью, то есть генераторов, которые выполняют `yield`-делегирование самим себе:

```
function *foo(val) {
    if (val > 1) {
        // рекурсия генератора
        val = yield *foo( val - 1 );
    }

    return yield request( "http://some.url/?v=" + val );
}

function *bar() {
    var r1 = yield *foo( 3 );
    console.log( r1 );
}

run( bar );
```

Нашу функцию `run(..)` можно было бы вызывать в виде `run( foo, 3 )`, потому что она поддерживает передачу дополнительных параметров при инициализации генератора. Тем не менее мы использовали `*bar()` без параметров, чтобы подчеркнуть гибкость `yield *`.

Какие основные действия происходят в этом коде? Приготовьтесь, подробное описание будет довольно длинным:

1. `run(bar)` запускает генератор `*bar()`.
2. `foo(3)` создает *итератор* для `*foo(..)` и передает `3` в параметре `val`.
3. Поскольку `3 > 1`, `foo(2)` создает другой *итератор* и передает `2` в параметре `val`.

4. Поскольку `2 > 1`, `foo(1)` создает другой *итератор* и передает `1` в параметре `val`.

5. Условие `1 > 1` не выполняется, поэтому мы затем вызываем `request(..)` со значением `1` и получаем обратно обещание для первого вызова Ajax.

6. Это обещание выдается `yield` и переходит к экземпляру генератора `*foo(2)`.

7. `Yield *` передает обещание обратно экземпляру генератора `*foo(3)`. Другая команда `yield *` передает обещание экземпляру генератора `*bar()`. После этого еще одна команда `yield *` передает обещание функции `run(..)`, которая ожидает по этому обещанию (для первого запроса Ajax) для продолжения работы.

8. Когда обещание разрешается, его сообщение выполнения отправляется для продолжения `*bar()`; через `yield *` оно передается экземпляру `*foo(3)`, после чего через `yield *` оно передается экземпляру генератора `*foo(2)`, после чего через `yield *` оно передается нормальной команде `yield`, ожидающей в экземпляре генератора `*foo(3)`.

9. Ответ Ajax первого вызова теперь немедленно возвращается из экземпляра генератора `*foo(3)`, который отправляет это значение как результат выражения `yield *` в экземпляре `*foo(2)`, и присваивается локальной переменной `val`.

10. Внутри `*foo(2)` выдается второй запрос Ajax вызовом `request(..)`; его обещание возвращается `yield` экземпляру `*foo(1)`, и затем `yield *` распространяется по всему пути до `run(..)` (снова шаг 7). Когда обещание разрешится, второй ответ Ajax распространяется по всему пути до экземпляра генератора `*foo(2)` и присваивается локальной переменной `val`.

11. Наконец, выдается третий запрос Ajax вызовом `request(..)`; его обещание проходит до `run(..)`, после чего значение разрешения проходит весь обратный путь. Затем оно возвращается

через `return`, чтобы дойти до ожидающего выражения `yield *` в `*bar()`.

Уф! Какая-то дикая мысленная эквилибристика! Возможно, вам стоит перечитать это несколько раз, а потом пойти и съесть что-нибудь, чтобы отвлечься.

Параллельное выполнение генераторов

Как обсуждалось в главе 1 и ранее в этой главе, два одновременно работающих «процесса» могут чередовать свои операции в кооперативном режиме, и достаточно часто это приводит к очень эффективным выражениям асинхронности.

Откровенно говоря, наши примеры чередования множественных генераторов показывали, каким запутанным может стать код. Но мы намекнули, что в некоторых местах такая возможность будет достаточно полезной.

Вспомните сценарий, рассмотренный в главе 1: два разных одновременно работающих обработчика ответов Ajax должны координировать свою работу друг с другом, чтобы предотвратить потенциальную ситуацию гонки при передаче данных. Ответы размещались в массиве `res`:

```
function response(data) {
    if (data.url == "http://some.url.1") {
        res[0] = data;
    }
    else if (data.url == "http://some.url.2") {
        res[1] = data;
    }
}
```

Но как организовать параллельное использование нескольких генераторов в этом сценарии?

```
// `request(..)` - функция Ajax с поддержкой обещаний
var res = [];
function *reqData(url) {
    res.push(
        yield request( url )
    );
}
```

Здесь используются два экземпляра генератора `*reqData(..)`, но с таким же успехом можно было использовать один экземпляр двух разных генераторов; анализ в обоих случаях остается неизменным. Координация двух разных генераторов будет рассмотрена ниже.

Вместо того чтобы разбираться с ручным присваиванием `res[0]` и `res[1]`, мы используем координированное упорядочение, чтобы операция `res.push(..)` сохраняла значения в ожидаемом и предсказуемом порядке. Это должно сделать выраженную логику немного чище.

Но как организовать это взаимодействие? Для начала попробуем сделать это вручную с использованием обещаний:

```
var it1 = reqData( "http://some.url.1" );
var it2 = reqData( "http://some.url.2" );

var p1 = it1.next();
var p2 = it2.next();

p1
.then( function(data){
    it1.next( data );
    return p2;
} )
.then( function(data){
    it2.next( data );
} );
```

Два экземпляра `*reqData(..)` запускаются для выдачи запросов Ajax, а затем приостанавливаются `yield`. После этого мы выбира-

ем возобновление первого экземпляра при разрешении `p1`, после чего разрешение `p2` перезапустит второй экземпляр. Таким образом мы используем организационные средства обещаний, чтобы в `res[0]` хранился первый ответ, а в `res[1]` — второй.

Но честно говоря, это решение остается до отвращения ручным и не позволяет генераторам организовать свою работу, в чем, возможно, заключается их истинная сила. Попробуем сделать это иначе:

```
// `request(..)` - функция Ajax с поддержкой обещаний

var res = [];

function *reqData(url) {
    var data = yield request( url );

    // передача управления
    yield;

    res.push( data );
}

var it1 = reqData( "http://some.url.1" );
var it2 = reqData( "http://some.url.2" );

var p1 = it.next();
var p2 = it.next();

p1.then( function(data){
    it1.next( data );
} );

p2.then( function(data){
    it2.next( data );
} );

Promise.all( [p1,p2] )
.then( function(){
    it1.next();
    it2.next();
} );
```

Так, уже немного лучше (хотя решение остается ручным!), потому что теперь два экземпляра `*reqData(..)` выполняются действительно параллельно и (по крайней мере, в первой части) независимо.

В предыдущем фрагменте второму экземпляру данные не передавались до тех пор, пока первый экземпляр не будет полностью завершен. Но здесь оба экземпляра получают свои данные сразу же после получения соответствующих ответов, после чего каждый экземпляр снова выполняет `yield` для целей передачи управления. Затем мы выбираем, в каком порядке возобновить их, в обработчике `Promise.all([ .. ])`.

В этом решении есть один неочевидный момент: оно подсказывает более простую форму функции, пригодной для повторного использования, из-за симметрии. Представьте функцию с именем `runAll(..)`:

```
// `request(..)` - функция Ajax с поддержкой обещаний

var res = [];

runAll(
    function*(){
        var p1 = request( "http://some.url.1" );

        // передача управления
        yield;

        res.push( yield p1 );
    },
    function*(){
        var p2 = request( "http://some.url.2" );

        // передача управления
        yield;
        res.push( yield p2 );
    }
);
```

Я не привожу код `runAll(..)`, потому что он не только затормозит темп изложения материала, но и является расширением логики, уже реализованной в `run(..)`. Таким образом, в качестве хорошего дополнительного упражнения попробуйте доработать код `run(..)`, чтобы он работал как вымышленная функция `runAll(..)`.

Кроме того, моя библиотека asynquence предоставляет ранее упомянутую функцию `runner(..)` со встроенной функциональностью такого рода (см. приложение А).

А вот как происходит обработка внутри `runAll(..)`:

1. Первый генератор получает обещание для первого ответа Ajax от `"http://some.url.1"`, а затем возвращает управление обратно функции `runAll(..)`.
2. Второй генератор запускается и делает то же самое для `"http://some.url.2"`, возвращая управление обратно функции `runAll(..)`.
3. Первый генератор возобновляет работу и выдает свое обещание `p1`. Функция `runAll(..)` делает в данном случае то же самое, что и предыдущая версия `run(..)`, — она ожидает разрешения этого обещания, а затем возобновляет выполнение того же генератора (без передачи управления!). Когда `p1` разрешится, `runAll(..)` снова возобновляет выполнение первого генератора с этим значением разрешения, после чего `res[0]` получает свое значение. Когда первый генератор затем завершает работу, происходит неявная передача управления.
4. Второй генератор возобновляет работу, выдает свое обещание `p2` и ожидает его разрешения. После этого `runAll(..)` возобновляет выполнение второго генератора с этим значением, после чего задается значение `res[1]`.

В нашем примере внешняя переменная `res` используется для хранения результатов двух разных ответов Ajax, это стало возможным благодаря координации параллельного выполнения.

Однако может быть полезно дополнительно расширить `runAll(..)` для того, чтобы предоставить внутреннее пространство, общее для нескольких экземпляров генераторов, — например, пустой объект, который далее будет называться `data`. Кроме того, эта реализация может получать переданные через `yield` значения, не являющиеся обещаниями, и передавать их следующему генератору.

Пример:

```
// `request(..)` - функция Ajax с поддержкой обещаний

runAll(
    function*(data){
        data.res = [];

        // передача управления (и передача сообщения)
        var url1 = yield "http://some.url.2";

        var p1 = request( url1 ); // "http://some.url.1"

        // передача управления
        yield;

        data.res.push( yield p1 );
    },
    function*(data){
        // передача управления (и передача сообщения)
        var url2 = yield "http://some.url.1";

        var p2 = request( url2 ); // "http://some.url.2"

        // передача управления
        yield;

        data.res.push( yield p2 );
    }
);
```

В этой формулировке два генератора не только координируют передачу управления, но и напрямую общаются друг с другом, как через `data.res`, так и через переданные через `yield` сообщения,

которые меняют местами значения `url1` и `url2`. Это невероятно мощная возможность!

Такая реализация также служит концептуальной базой для более сложного асинхронного метода CSP (Communicating Sequential Processes), который рассматривается в приложении Б этой книги.

Преобразователи

До настоящего момента предполагалось, что выдача обещания из генератора — и возобновление выполнения генератора этим обещанием через вспомогательную функцию вроде `run(..)` — была лучшим возможным способом управления асинхронностью при помощи генераторов. Не буду скрывать: так оно и есть.

Но мы пропустили другой паттерн, который получил более или менее широкое распространение, поэтому для полноты картины я кратко опишу его.

В общей теории обработки данных существует старая, появившаяся еще до JS концепция *преобразователей* (thunk). Не углубляясь в историю происхождения термина, узкоспециализированным выражением преобразователя в JS является функция, которая без каких-либо параметров подключается для вызова другой функции.

Иначе говоря, вызов функции (со всеми необходимыми параметрами) «заворачивается» в определение функции, которое создает промежуточное звено для выполнения внутренней функции. Внешняя функция-обертка и называется преобразователем. Позднее при выполнении преобразователя в конечном итоге будет вызвана исходная функция.

Пример:

```
function foo(x,y) {
    return x + y;
```

```
}
function fooThunk() {
    return foo( 3, 4 );
}

// позднее

console.log( fooThunk() );  // 7
```

Итак, синхронный преобразователь устроен достаточно прямолинейно. Но что насчет асинхронного преобразователя? Можно расширить узкое определение преобразования и включить в него получение обратного вызова.

Пример:

```
function foo(x,y,cb) {
    setTimeout( function(){
        cb( x + y );
    }, 1000 );
}

function fooThunk(cb) {
    foo( 3, 4, cb );
}

// позднее
fooThunk( function(sum){
    console.log( sum );     // 7
} );
```

Как видите, функция `fooThunk(..)` ожидает получить только параметр `cb(..)`, так как она уже имеет значения `3` и `4` (для `x` и `y` соответственно), заданные заранее и готовые к передаче `foo(..)`. Преобразователь просто терпеливо ожидает последнего компонента, необходимого ему для выполнения работы — обратного вызова.

Впрочем, создавать преобразователи вручную не хочется. Давайте изобретем функцию, которая выполняет упаковку за нас.

Пример:

```
function thunkify(fn) {
    var args = [].slice.call( arguments, 1 );
    return function(cb) {
        args.push( cb );
        return fn.apply( null, args );
    };
}

var fooThunk = thunkify( foo, 3, 4 );
// позднее

fooThunk( function(sum) {
    console.log( sum );     // 7
} );
```

Здесь предполагается, что исходная сигнатура функции (`foo(..)`) ожидает получить обратный вызов в последней позиции, а все остальные параметры предшествуют ему. Это почти повсеместно принятый стандарт для асинхронных функций JS, который можно назвать «обратный вызов на последнем месте». Если по какой-то причине у вас возникнет необходимость в обработке сигнатур в стиле «обратный вызов на первом месте», просто создайте функцию, использующую `args.unshift(..)` вместо `args.push(..)`.

Приведенная формулировка `thunkify(..)` получает ссылку на функцию `foo(..)` и все необходимые ей параметры и возвращает сам преобразователь (`fooThunk(..)`). Тем не менее такой подход к использованию преобразователей в JS нельзя назвать типичным.

Вместо того чтобы создавать сам преобразователь, функция `thunkify(..)` чаще (хотя это выглядит довольно загадочно) создает функцию, которая производит преобразователи.

Ммм... да, вот так.

Пример:

```
function thunkify(fn) {
    return function() {
        var args = [].slice.call( arguments );
        return function(cb) {
            args.push( cb );
            return fn.apply( null, args );
        };
    };
}
```

Главное отличие — лишний уровень `return function() { .. }`. А вот как выглядит ее использование:

```
var whatIsThis = thunkify( foo );

var fooThunk = whatIsThis( 3, 4 );

// позднее
fooThunk( function(sum) {
    console.log( sum );     // 7
} );
```

При виде этого фрагмента возникает резонный вопрос: что собой представляет `whatIsThis`? Это не преобразователь, а нечто, что производит преобразователи на базе вызовов `foo(..)`. Это своего рода «фабрика» для создания преобразователей. Специального термина для таких объектов вроде бы не существует, поэтому я буду использовать термин «фабрика преобразователей».

Таким образом, `thunkify(..)` производит фабрику преобразователей, а фабрика преобразователей производит преобразователи.

```
var fooThunkory = thunkify( foo );

var fooThunk1 = fooThunkory( 3, 4 );
var fooThunk2 = fooThunkory( 5, 6 );

// позднее
```

```
fooThunk1( function(sum) {
    console.log( sum );     // 7
} );

fooThunk2( function(sum) {
    console.log( sum );     // 11
} );
```

В примере с `foo(..)` предполагается стиль обратных вызовов, отличный от стиля «ошибка на первом месте». Конечно, стиль «ошибка на первом месте» встречается намного чаще. Если бы у функции `foo(..)` были какие-то законные ожидания по генерированию ошибок, можно было бы доработать ее так, чтобы она ожидала и использовала обратный вызов «ошибка на первом месте». Механизмы `thunkify(..)` не обращают внимания на то, какой стиль обратного вызова подразумевается в коде. Изменился бы только синтаксис использования: `fooThunk1(function(err,sum){...`

Раскрытие метода фабрики преобразователей (тогда как в более ранней реализации `thunkify(..)` этот промежуточный шаг скрывался) может показаться ненужным усложнением. Но в общем случае бывает полезно создавать фабрики преобразователей в начале программы для инкапсуляции существующих методов API, а затем иметь возможность передавать эти фабрики и обращаться к ним с вызовами, когда вам понадобятся преобразователи. Два раздельных шага обеспечивают более четкое разделение обязанностей.

Пример:

```
// более четко:
var fooThunkory = thunkify( foo );

var fooThunk1 = fooThunkory( 3, 4 );
var fooThunk2 = fooThunkory( 5, 6 );

// вместо:
var fooThunk1 = thunkify( foo, 3, 4 );
var fooThunk2 = thunkify( foo, 5, 6 );
```

Независимо от того, захочется вам явно работать с фабриками преобразователей или нет, использование преобразователей `fooThunk1(..)` и `fooThunk2(..)` остается неизменным.

s/promise/thunk/

Как все эти разговоры о преобразователях относятся к генераторам?

Общее сравнение преобразователей с обещаниями: они не взаимозаменяемы, так как не обладают эквивалентным поведением. Обещания обладают гораздо более широкими возможностями и пользуются большим доверием, чем простые преобразователи.

Но в каком-то смысле обе конструкции могут рассматриваться как запрос значения, ответ на который может предоставляться асинхронно.

Вспомните, что в главе 3 мы определили метод для «обещанизации» функции, который назывался `Promise.wrap(..)`. Обертка для обещаний не производит обещаний; она производит фабрики обещаний, которые, в свою очередь, производят обещания. Ситуация совершенно такая же, как с фабриками преобразователей и преобразователями.

Чтобы продемонстрировать симметрию, сначала изменим пример `foo(..)` и переведем его на обратный вызов в стиле «ошибка на первом месте»:

```
function foo(x,y,cb) {
    setTimeout( function(){
        // предполагается, что `cb(..)` использует
        // стиль "ошибка на первом месте"
        cb( null, x + y );
    }, 1000 );
}
```

А теперь сравним использование `thunkify(..)` и `promisify(..)` (то есть `Promise.wrap(..)` из главы 3):

```
// симметрия: конструирование того, кто выдает запросы
var fooThunkory = thunkify( foo );
var fooPromisory = promisify( foo );

// симметрия: выдача запроса
var fooThunk = fooThunkory( 3, 4 );
var fooPromise = fooPromisory( 3, 4 );

// получение ответа-преобразователя
fooThunk( function(err,sum){
    if (err) {
        console.error( err );
    }
    else {
        console.log( sum );     // 7
    }
} );

// получение ответа-обещания
fooPromise
.then(
    function(sum){
        console.log( sum );     // 7
    },
    function(err){
        console.error( err );
    }
);
```

И фабрика преобразователей, и фабрика обещаний фактически выдают запрос (на получение значения), соответственно, преобразователь `fooThunk` и обещание `fooPromise` представляют будущие ответы на этот вопрос. В таком свете симметрия очевидна.

Вооружившись этой точкой зрения, мы видим, что генераторы, которые выдают обещания для асинхронности, могут вместо этого выдавать преобразователи для асинхронности. Все, что для этого понадобится, — более умная реализация `run(..)` (см. выше), которая способна не только найти и связаться с выданным через `yield` обещанием, но и предоставить обратный вызов выданному преобразователю.

Пример:

```
function *foo() {
    var val = yield request( "http://some.url.1" );
    console.log( val );
}

run( foo );
```

В этом примере `request(..)` может быть как фабрикой обещаний, которая возвращает обещание, так и фабрикой преобразователей, которая возвращает преобразователь. С точки зрения того, что происходит в логике генератора, эта подробность реализации неважна, а это весьма удобно!

Таким образом, `request(..)` может задаваться любым из двух способов:

```
// `request(..)` с обещаниями (см. главу 3)
var request = Promise.wrap( ajax );

// или

// `request(..)` с фабрикой преобразователей
var request = thunkify( ajax );
```

Наконец, чтобы внести поддержку преобразователей в более раннюю функцию `run(..)`, нам понадобится логика следующего вида:

```
// ..
// получили преобразователь?
else if (typeof next.value == "function") {
    return new Promise( function(resolve,reject){
        // вызвать преобразователь с обратным вызовом
        // в стиле "ошибка на первом месте"
        next.value( function(err,msg) {
            if (err) {
                reject( err );
            }
            else {
                resolve( msg );
            }
```

```
        } );
    } )
    .then(
        handleNext,
        function handleErr(err) {
            return Promise.resolve(
                it.throw( err )
            )
            .then( handleResult );
        }
    );
}
```

Теперь наши генераторы могут вызывать либо фабрики обещаний для выдачи обещаний, либо фабрики преобразователей для выдачи преобразователей; в любом случае `run(..)` обработает это значение и использует его для ожидания завершения, чтобы продолжить работу генератора.

С точки зрения симметрии два подхода кажутся идентичными. Тем не менее следует заметить, что это справедливо только с точки зрения обещаний или преобразователей, представляющих собой продолжение генератора по будущему значению.

В более общей перспективе преобразователи вряд ли смогут предоставить гарантии доверия или средства композиции, с которыми проектировались обещания. Использование преобразователя как замены для обещания в этом конкретном паттерне асинхронности генератора — рабочее решение, но следует понимать, что оно далеко не идеально в отношении всех преимуществ, предоставляемых обещаниями (см. главу 3).

Если у вас есть выбор, используйте `yield` с обещаниями, а не с преобразователями. Тем не менее нет ничего плохого в реализации `run(..)`, которая может работать со значениями обоих видов.

Функция `runner(..)` из моей библиотеки asynquence (см. приложение А) обрабатывает выдачу обещаний, преобразователей и последовательностей (sequences) asynquence.

Генераторы до ES6

Надеюсь, вы уже убедились в том, что генераторы могут стать очень важным дополнением к вашему инструментарию асинхронного программирования. Но этот новый синтаксис появился только в ES6, а это означает, что вы не сможете просто определить полифилы для генераторов, как это делается для обещаний (которые всего лишь представляют новый API). Что можно сделать, чтобы добавить генераторы в JS-код в браузере, если вы не можете себе позволить игнорировать браузеры, не поддерживающие ES6?

Для всех новых синтаксических расширений ES6 существуют инструменты (чаще всего они называются *транспиляторами* — от «транс-компилятор»), которые могут взять синтаксис ES6 и преобразовать его в эквивалентный (но очевидно более уродливый!) код без поддержки ES6. Таким образом, генераторы могут транспилироваться в код, который обладает тем же поведением, но работает в ES5 и ниже.

Но как? На первый взгляд вообще непонятно, как транспилировать в другую форму «волшебное» поведение `yield`. Впрочем, я уже намекал на решение при обсуждении *итераторов* на базе замыканий.

Ручное преобразование

Прежде чем обсуждать транспиляторы, давайте разберемся, как ручная транспиляция могла бы работать в случае генераторов. Не стоит думать, что это чисто теоретическое упражнение, потому что оно поможет вам лучше понять, как работает такая замена.

Пример:

```
// `request(..)` - функция Ajax с поддержкой обещаний
function *foo(url) {
```

```
    try {
        console.log( "requesting:", url );
        var val = yield request( url );
        console.log( val );
    }
    catch (err) {
        console.log( "Oops:", err );
        return false;
    }
}
var it = foo( "http://some.url.1" );
```

Первое, на что следует обратить внимание, — это то, что нам все равно нужна обычная функция `foo()`, которая может вызываться, и она должна возвращать *итератор*. Итак, кратко обрисуем необходимое преобразование:

```
function foo(url) {

    // ..

    // создать и вернуть итератор
    return {
        next: function(v) {
            // ..
        },
        throw: function(e) {
            // ..
        }
    };
}

var it = foo( "http://some.url.1" );
```

Следующее, на что следует обратить внимание, — что «волшебное» поведение генератора заключается в приостановке его области видимости/состояния, но его можно эмулировать при помощи функциональных замыканий. Чтобы понять, как написать такой код, мы сначала разметим разные части генератора значениями состояния:

```
// `request(..)` - функция Ajax с поддержкой обещаний

function *foo(url) {

    // СОСТОЯНИЕ 1
    try {
        console.log( "requesting:", url );
        var TMP1 = request( url );

        // СОСТОЯНИЕ 2
        var val = yield TMP1;
        console.log( val );
    }
    catch (err) {
        // СОСТОЯНИЕ 3
        console.log( "Oops:", err );
        return false;
    }
}
```

Чтобы иллюстрация была более точной, мы разбили команду `val = yield request..` на две части с использованием временной переменной `TMP1`. `request(..)` происходит в состоянии 1, а присваивание значения завершения `val` происходит в состоянии 2. Мы избавимся от промежуточной переменной `TMP1`, когда займемся преобразованием кода в эквивалентный код без генераторов.

Другими словами, 1 — исходное состояние, 2 — состояние при успешном выполнении `request(..)`, 3 — состояние при неудачном выполнении `request(..)`. Вероятно, вы уже поняли, как дополнительные шаги `yield` могут кодироваться дополнительными состояниями.

Возвращаясь к транспилированному генератору, определим в замыкании переменную `state`, которая может использоваться для отслеживания состояния:

```
function foo(url) {
    // управление состоянием генератора
    var state;

    // ..
}
```

А теперь определим внутри замыкания внутреннюю функцию с именем `process(..)`, которая обрабатывает все состояния в конструкции `switch`:

```
// `request(..)` - функция Ajax с поддержкой обещаний
function foo(url) {
    // управление состоянием генератора
    var state;

    // объявления переменных уровня генератора
    var val;

    function process(v) {
        switch (state) {
            case 1:
                console.log( "requesting:", url );
                return request( url );
            case 2:
                val = v;
                console.log( val );
                return;
            case 3:
                var err = v;
                console.log( "Oops:", err );
                return false;
        }
    }

    // ..
}
```

Каждое состояние в генераторе представлено отдельной секцией `case` в команде `switch`. Функция `process(..)` будет вызываться каждый раз, когда потребуется обработать новое состояние. Вскоре мы вернемся к тому, как работает эта схема.

Любые объявления переменных уровня генератора (`val`) перемещаются за пределы `process(..)`, чтобы они могли пережить многократные вызовы `process(..)`. Переменная `err` с блочной областью видимости необходима только в состоянии 3, поэтому она остается на месте.

В состоянии 1 вместо `yield resolve(..)` используется `return resolve(..)`. В завершающем состоянии 2 явного возвращаемого значения нет, поэтому мы просто выполняем `return` — это то же самое, что `return undefined`. В завершающем состоянии 3 использовалась команда `return false`, поэтому мы сохраняем ее.

Теперь необходимо определить код в функциях *итератора*, чтобы они вызывали `process(..)` соответствующим образом:

```
function foo(url) {
    // управление состоянием генератора
    var state;
    // объявления переменных уровня генератора
    var val;

    function process(v) {
        switch (state) {
            case 1:
                console.log( "requesting:", url );
                return request( url );
            case 2:
                val = v;
                console.log( val );
                return;
            case 3:
                var err = v;
                console.log( "Oops:", err );
                return false;
        }
    }

    // создать и вернуть итератор
    return {
        next: function(v) {
```

```
            // исходное состояние
            if (!state) {
                state = 1;
                return {
                    done: false,
                    value: process()
                };
            }
            // yield возобновляется успешно
            else if (state == 1) {
                state = 2;
                return {
                    done: true,
                    value: process( v )
                };
            }
            // генератор уже завершен
            else {
                return {
                    done: true,
                    value: undefined
                };
            }
        },
        "throw": function(e) {
            // явная обработка ошибок выполняется только
            // в состоянии 1
            if (state == 1) {
                state = 3;
                return {
                    done: true,
                    value: process( e )
                };
            }
            // в противном случае ошибка не будет обработана,
            // поэтому она просто выдается обратно
            else {
                throw e;
            }
        }
    };
}
```

Как работает этот код?

1. Первый вызов функции `next()` итератора переведет генератор из неинициализированного состояния в состояние 1, после чего вызывается `process()` для обработки этого состояния. Возвращаемое значение `request(..)`, которое является обещанием для ответа Ajax, возвращается как свойство `value` из вызова `next()`.

2. Если запрос Ajax завершается успешно, то второй вызов `next(..)` должен отправить значение ответа Ajax, в результате чего происходит переход в состояние 2. `process(..)` вызывается снова (на этот раз с переданным значением ответа Ajax), а свойство `value`, возвращаемое `next(..)`, будет содержать `undefined`.

3. Но если запрос Ajax завершается неудачей, вызывается `throw(..)` с ошибкой, в результате чего происходит переход из состояния 1 в состояние 3 (вместо 2). И снова вызывается `process(..)`, на этот раз со значением ошибки. Эта секция `case` возвращает значение `false`, которое задается свойству `value`, возвращаемому из вызова `throw(..)`.

Снаружи, то есть при взаимодействии только с итератором, обычная функция `foo(..)` работает практически так же, как работал бы генератор `*foo(..)`. Итак, мы фактически транспилировали свой генератор ES6 в совместимый код до ES6.

После этого можно вручную создать экземпляр генератора и управлять его итератором, вызывать `var it = foo("..")` и `it.next(..)` и т. д. или, еще лучше, передать ранее определенной функции `run(..)` в виде `run(foo,"..")`.

Автоматическая транспиляция

Предыдущий пример с ручным определением преобразования генератора ES6 в эквивалентный код до ES6 демонстрирует работу генераторов на концептуальном уровне. Но это преобразование

было крайне запутанным и совершенно не портировалось для других генераторов в нашем коде. Было бы крайне непрактично выполнять такую работу вручную, и это сведет к нулю все преимущества генераторов.

Но к счастью, существуют определенные инструменты, которые могут автоматически преобразовывать генераторы ES6 в код, сходный с тем, который был создан в предыдущем разделе. Они не только берут на себя всю рутинную работу, но и разбираются с затруднениями, о которых я не упомянул.

Один из таких инструментов — *regenerator* (https://facebook.github.io/regenerator/) был разработан умными людьми из Facebook.

Если воспользоваться regenerator для траспиляции предыдущего генератора, будет получен следующий код (на момент написания книги):

```
// `request(..)` - функция Ajax с поддержкой обещаний

var foo = regeneratorRuntime.mark(function foo(url) {
    var val;

    return regeneratorRuntime.wrap(function foo$(context$1$0) {
        while (1) switch (context$1$0.prev = context$1$0.next) {
        case 0:
            context$1$0.prev = 0;
            console.log( "requesting:", url );
            context$1$0.next = 4;
            return request( url );
        case 4:
            val = context$1$0.sent;
            console.log( val );
            context$1$0.next = 12;
            break;
        case 8:
            context$1$0.prev = 8;
            context$1$0.t0 = context$1$0.catch(0);
            console.log("Oops:", context$1$0.t0);
            return context$1$0.abrupt("return", false);
        case 12:
        case "end":
```

```
            return context$1$0.stop();
        }
    }, foo, this, [[0, 8]]);
});
```

Заметно явное сходство с ручной реализацией (например, команды `switch`/`case` и даже извлечение `val` из замыкания, как это делали мы).

Конечно, для транспиляции regenerator необходима вспомогательная библиотека `regeneratorRuntime`, которая содержит всю универсальную логику управления обобщенным генератором/*итератором*. Большая часть шаблонного кода отличается от нашей версии, но даже при этом некоторые концепции все равно видны — например, отслеживание следующего состояния генератора в `context$1$0.next = 4`.

Главный вывод состоит в том, что полезность генераторов не ограничивается средами ES6. Как только вы поймете общие принципы, вы сможете применять их в своем коде и пользоваться инструментами для преобразования кода в форму, совместимую с более старыми средами.

Это потребует большего объема работы, чем при простом использовании полифилов Promise API для сред до ES6, но результат того, безусловно, стоит. Генераторы намного лучше справляются с выражением асинхронной программной логики в более понятной, последовательной, логичной форме, которая выглядит синхронно.

Стоит привыкнуть к генераторам, и вам уже никогда не захочется возвращаться к мешанине из асинхронных обратных вызовов!

Итоги

Генераторы — новая разновидность функций ES6, которые не выполняются до завершения, как обычные функции. Вместо этого

генератор может быть приостановлен на середине завершения (с полным сохранением состояния), а позднее продолжить работу с точки приостановки.

В основе переходов от приостановки к продолжению работы лежит принцип кооперативной работы, а не вытеснения; это означает, что только сам генератор может приостановить себя, используя ключевое слово `yield`, и только *итератор*, управляющий генератором, может (при помощи `next(..)`) возобновить выполнение генератора.

Дуализм `yield`/`next(..)` — не просто механизм управления; в действительности это механизм двусторонней передачи сообщений. Выражение `yield ..` фактически приостанавливается в ожидании значения, а следующий вызов `next(..)` передает значение (или неявное `undefined`) приостановленному выражению `yield`.

Главное преимущество генераторов, связанное с управлением асинхронной программной логикой, заключается в том, что код внутри генератора выражает последовательность шагов задачи естественным синхронным/последовательным образом. Фокус в том, что потенциальная асинхронность прячется за ключевым словом `yield`, то есть асинхронность перемещается в код, управляющий итератором генератора.

Другими словами, генераторы поддерживают паттерн последовательного синхронного блокирующего кода по отношению к асинхронному коду. Это позволяет нашему мозгу более естественно анализировать код и устраняет один из двух ключевых недостатков асинхронности на базе обратных вызовов.

5 Быстродействие программ

До настоящего момента в книге речь шла исключительно о более эффективном применении асинхронных паттернов. Однако мы до сих пор напрямую не рассматривали вопрос о том, почему асинхронность вообще важна для JS. Наиболее очевидная причина — *быстродействие*.

Например, если вам понадобилось выдать два запроса Ajax и эти запросы не зависят друг от друга, но вам необходимо дождаться завершения обоих до перехода к следующей операции, это взаимодействие можно построить по двум моделям: последовательной и параллельной.

Можно выдать первый запрос и отложить момент запуска второго запроса до того, как будет завершен первый. А можно, как было показано ранее с обещаниями и генераторами, выдать оба запроса параллельно и приказать шлюзу ожидать завершения обоих перед тем, как следовать дальше.

Очевидно, второй вариант эффективнее первого. А хорошее быстродействие обычно повышает качество взаимодействия с пользователем.

Не исключено, что асинхронность (параллельное выполнение с чередованием) улучшит только субъективное восприятие быстродействия, а общее время выполнения программы останется прежним. Восприятие быстродействия пользователем ничуть не менее — а то и более! — важно, чем результаты объективных измерений быстродействия.

Чтобы обсудить более общие аспекты быстродействия на уровне программы, нужно выйти за пределы локальных асинхронных паттернов.

Возможно, вас больше интересуют вопросы быстродействия микроуровня, скажем, какая из операций выполняется быстрее, a++ или ++a? Такие аспекты быстродействия будут рассматриваться в главе 6.

Веб-работники

Если вы используете операции, требующие интенсивных вычислений, но не хотите выполнять их в главном потоке (что может привести к замедлению браузера/пользовательского интерфейса), возможно, вы бы предпочли, чтобы JavaScript мог работать в условиях многопоточной модели.

В главе 1 мы подробно обсудили, почему JavaScript является однопоточным языком. И это остается правдой. Однако однопоточность — это не единственный способ организации выполнения вашей программы.

Представьте, что ваша программа разбита на две части. Одна из этих частей выполняется в главном UI-потоке, а другая в совершенно отдельном потоке.

Какие проблемы может создать такая архитектура? Прежде всего, нужно знать, означает ли выполнение в отдельном потоке параллельное выполнение (в многопроцессорных/многоядерных си-

стемах), чтобы продолжительные процессы во втором потоке не блокировали главный поток программы. В противном случае «виртуальные потоки» не принесут особой пользы по сравнению с тем, чего уже можно добиться в JS при помощи асинхронного параллельного выполнения.

Также необходимо знать, имеют ли две части программы доступ к общей области видимости/ресурсам. Если имеют, тогда вам придется решать все проблемы многопоточных языков (Java, C++ и т. д.), такие как необходимость кооперативной или вытесняющей блокировки (мьютексы и т. д.). Все это требует значительной дополнительной работы, не стоит ее недооценивать. А может быть, нужно понять, как эти две части будут взаимодействовать, если они не могут использовать общую область видимости/ресурсы.

Все эти вопросы непременно должны учитываться во время знакомства с новой возможностью, добавленной на платформу веб-технологий приблизительно с выходом HTML5 — *веб-работниками* (Web Workers). Это возможность браузера (то есть управляющей среды), которая не имеет почти никакого отношения к самому языку JS. Таким образом, *в настоящее время* в JavaScript нет никаких средств поддержки многопоточного выполнения.

Однако такая среда, как браузер, легко может предоставить несколько экземпляров движка JavaScript, каждое из которых выполняется в отдельном потоке, и дать вам возможность запустить разные программы в разных потоках. Каждая из этих отдельных потоковых частей программы называется *веб-работником*. Данная разновидность параллелизма называется *параллелизмом на уровне задач*, так как основное внимание в ней уделяется разбиению программы на части для параллельного выполнения.

Экземпляр веб-работника создается в главной программе JS (или в другом веб-работнике) следующим образом:

```
var w1 = new Worker( "http://some.url.1/mycoolworker.js" );
```

URL-адрес должен указывать на местонахождение файла JS (не страницы HTML!), который должен быть загружен в веб-работнике. Браузер создает отдельный поток и дает возможность файлу выполняться как независимой программе в этом потоке.

Разновидность веб-работника, создаваемого для таких URL-адресов, называется выделенным (dedicated) веб-работником. Но вместо URL-адреса внешнего файла также можно создать встроенного (inline) веб-работника, предоставив ему двоичный (Blob) URL-адрес (еще одна возможность HTML5); по сути, это встроенный файл, хранящийся в одном (двоичном) значении. Впрочем, двоичные URL-адреса выходят за рамки нашего обсуждения.

Работники не используют ресурсы или области видимости совместно друг с другом или главной программой, это вывело бы на передний план все ужасы многопоточного программирования. Вместо этого они связываются при помощи базового механизма передачи событий.

Объект работника `w1` является триггером и слушателем событий, что позволяет подписаться на события, отправляемые работником, а также отправлять события работнику.

Прослушивание событий (вернее, фиксированного события `"message"`) происходит так:

```
w1.addEventListener( "message", function(evt){
    // evt.data
} );
```

А вот как событие `"message"` отправляется работнику:

```
w1.postMessage( "something cool to say" );
```

Внутри работника передача сообщений выглядит абсолютно симметрично:

```
// "mycoolworker.js"
addEventListener( "message", function(evt){
    // evt.data
} );
postMessage( "a really cool reply" );
```

Обратите внимание: у выделенного работника существует связь типа «один к одному» с создавшей его программой. А именно — событие `"message"` не создает никакой неоднозначности, потому что оно гарантированно может поступить только из этой связи «один к одному» — либо от работника, либо от главной страницы.

Обычно приложение главной страницы создает работников, но работник может создавать экземпляры своих подчиненных работников (subworkers). Иногда бывает полезно делегировать такие подробности «главному» работнику, который порождает других работников для обработки частей задачи. К сожалению, на момент написания книги в Chrome подчиненные работники не поддерживались (но поддерживались в Firefox).

Чтобы немедленно уничтожить работника из программы, которая его создала, вызовите `terminate()` для объекта работника (как `w1` в приведенном фрагменте). Моментальное прерывание потока работника не дает ему возможности завершить свою работу или освободить занятые ресурсы (по аналогии с закрытием вкладки браузера для уничтожения страницы).

Если в браузере открыты две и более страниц (или несколько вкладок с одной страницей), которые пытаются создать работника на базе одинаковых URL-адресов файлов, в итоге для них будут созданы совершенно разные работники. Вскоре мы обсудим возможность совместного использования работников.

Казалось бы, вредоносная или просто тупая программа JS может легко организовать DoS-атаку на систему, порождая сотни работников, каждый из которых имеет собственный поток. И хотя размещение каждого работника в отдельном потоке отчасти гарантировано, эта гарантия не является неогра-

ниченной. Система сама может решать, сколько реальных потоков/процессоров/ядер она будет использовать. Невозможно предсказать или гарантировать, сколько из них будут доступны, хотя многие считают, что это число по крайней мере не меньше количества процессов/ядер. Я думаю, можно безопасно считать, что существует по крайней мере один поток кроме главного UI-потока, но на этом все и заканчивается.

Рабочая среда

Внутри работника никакие ресурсы основной программы не доступны. Это означает, что вы не можете обращаться ни к каким глобальным переменным, к модели DOM или другим ресурсам страницы. Помните: это совершенно отдельный поток.

Впрочем, в нем можно выполнять сетевые операции (Ajax, WebSockets) и устанавливать таймеры. Кроме того, работнику доступна собственная копия нескольких важных глобальных переменных/объектов, включая `navigator`, `location`, `JSON` и `applicationCache`.

Также можно загружать в работнике дополнительные сценарии JS командой `importScripts(..)`:

```
// внутри работника
importScripts( "foo.js", "bar.js" );
```

Эти сценарии загружаются синхронно; это означает, что `importScripts(..)` блокирует выполнение работника до момента завершения загрузки и выполнения файлов.

Также обсуждалась возможность предоставления работникам API `<canvas>`, что в сочетании с поддержкой передачи данных для canvas (см. раздел «Передача данных» этой главы) позволит работнику выполнять более сложную внепоточную обработку графики, что может быть полезно для высокопроизводительных игровых (WebGL) и других аналогичных приложений. И хотя эта поддержка еще не существует в браузерах, она может появиться в ближайшем будущем.

Для каких целей чаще всего используются веб-работники?

- Интенсивные математические вычисления.
- Сортировка больших наборов данных.
- Операции с данными (сжатие, анализ аудиоданных, обработка изображений на уровне пикселов и т. д.).
- Передача данных по сети с высоким уровнем трафика.

Передача данных

Возможно, вы заметили общую особенность большинства перечисленных применений: они требуют передачи большого объема информации через барьер между потоками (возможно, в обоих направлениях).

На первых порах существования работников единственным вариантом была сериализация всех данных в строковое значение. Кроме потери скорости из-за двусторонней сериализации, другим серьезным недостатком было копирование данных, что означало удвоение затрат памяти (и последующие затраты ресурсов на уборку мусора).

К счастью, теперь есть несколько более эффективных вариантов.

При передаче алгоритма *структурированный алгоритм клонирования* (https://developer.mozilla.org/en-US/docs/Web/Guide/API/DOM/The_structured_clone_algorithm) используется для копирования/дублирования объекта на другой стороне. Это довольно сложный алгоритм, который может даже дублировать объекты с циклическими ссылками. Потери на преобразование в строку/из строки отсутствуют, но дублирование памяти присутствует и при таком подходе. Поддержка реализована в IE10, а также во всех основных браузерах.

Еще лучший вариант, особенно для больших наборов данных, — передаваемые объекты (Transferable Objects). В этом случае пере-

дается принадлежность объекта, но сами данные не перемещаются. После того как объект будет передан работнику, в исходном месте он становится пустым или недоступным — это избавляет от рисков многопоточного программирования в общей области видимости. Конечно, передача принадлежности может осуществляться в обоих направлениях. Чтобы использовать передаваемые объекты, от вас потребуется совсем немного; любая структура данных, реализующая интерфейс `Transferable`, автоматически будет передаваться таким образом (поддерживается в Firefox и Chrome).

Например, типизованные массивы (такие, как `Uint8Array`) являются передаваемыми объектами. Вот как осуществляется отправка передаваемого объекта с использованием `postMessage(..)`:

```
// `foo` является `Uint8Array`

postMessage( foo.buffer, [ foo.buffer ] );
```

В первом параметре передается низкоуровневый буфер, а во втором список того, что передается.

Браузеры, не поддерживающие передаваемые объекты, просто возвращаются к использованию структурированного клонирования, что означает снижение быстродействия вместо полного нарушения функциональности.

Общие работники

Если ваш сайт или приложение позволяют создать несколько вкладок для одной страницы (довольно распространенная возможность), для снижения затрат ресурсов в системе можно запретить дублирование выделенных работников; самым распространенным ограниченным ресурсом в этом отношении является подключение к сети через сокет, так как браузеры ограничивают количество одновременных подключений к одному хосту. Конечно, ограничение подключений от клиента также снижает нагруз-

ку на сервер. В этом случае создание одного централизованного работника, общего для всех экземпляров страницы вашего сайта или приложения, может быть весьма полезным.

Общие работники (`SharedWorker`) создаются следующим образом (пока их поддержка ограничивается Firefox и Chrome):

```
var w1 = new SharedWorker( "http://some.url.1/mycoolworker.js" );
```

Поскольку общий работник может быть соединен с одним или несколькими экземплярами программы или страницы вашего сайта, он должен каким-то способом узнать, от какой программы пришло сообщение. Этот механизм однозначной идентификации называется *портом* (по аналогии с портами сетевых сокетов). Таким образом, вызывающая программа должна использовать для обмена данными объект `port` работника:

```
w1.port.addEventListener( "message", handleMessages );
// ..
w1.port.postMessage( "something cool" );
```

Кроме того, подключения к портам должны инициализироваться:

```
w1.port.start();
```

Внутри общего работника необходимо обрабатывать дополнительное событие: `"connect"`. Это событие предоставляет объект `port` для этого конкретного подключения. Самый удобный способ раздельного поддержания нескольких одновременных подключений основан на использовании замыканий на основании `port`, с прослушиванием и передачей событий для этого подключения внутри обработчика для события `"connect"`:

```
// внутри общего работника
addEventListener( "connect", function(evt){
    // порт для этого подключения
    var port = evt.ports[0];
    port.addEventListener( "message", function(evt){
        // ..
```

```
        port.postMessage( .. );
        // ..
    } );
    // инициализировать подключение через порт
    port.start();
} );
```

Если не считать этого различия, общие и выделенные работники обладают одинаковой функциональностью и семантикой.

Общие работники переживают разрыв подключения через порт, если другие подключения еще работоспособны, а выделенные работники завершаются при завершении подключения к запустившей их программе.

Полифилы для веб-работников

Веб-работники чрезвычайно эффективны с точки зрения параллельного выполнения программ JS. Однако может оказаться, что ваш код должен работать в старых браузерах, в которых они не поддерживаются. Поскольку работники представляют собой API, а не синтаксис, для них можно до определенной степени определить полифил.

Если браузер не поддерживает работников, имитировать многопоточность с точки зрения быстродействия просто невозможно. Принято считать, что `iframe` предоставляют параллельную среду выполнения, но во всех современных браузерах они на самом деле выполняются в одном потоке с главной страницей, так что для имитации параллелизма их недостаточно.

Как объяснялось в главе 1, асинхронность JS (не параллелизм) происходит от очереди цикла событий, так что вы можете имитировать асинхронность фиктивных работников при помощи таймеров (`setTimeout(..)` и т. д.). Затем остается предоставить полифилы для API работников. Некоторые варианты перечислены на GitHub-странице Modernizr GitHub (https://github.com/Modernizr/

Modernizr/wiki/HTML5-Cross-Browser-Polyfills), но откровенно говоря, ни один из них особо не впечатляет.

Я сделал набросок полизаполнения для работника (https://gist.github.com/getify/1b26accb1a09aa53ad25). При всей простоте он справится с простой поддержкой работников при условии, что двусторонняя передача сообщений работает правильно (как и обработка `"onerror"`). Вероятно, вы сможете расширить ее дополнительной функциональностью (например, `terminate()` или имитацией общих работников) по своему усмотрению.

Имитировать синхронную блокировку не удастся, так что это заполнение просто запрещает использование `importScripts(..)`. Другим возможным вариантом мог бы стать разбор и преобразование кода работника (после загрузки Ajax) для обработки перезаписи в некоторой асинхронной форме полифила `importScripts(..)` — возможно, через интерфейс с поддержкой обещаний.

SIMD

SIMD (Single instruction, multiple data, то есть «один поток команд, несколько потоков данных») — разновидность *параллелизма уровня данных* (в отличие от *параллелизма уровня задач* с веб-работниками), потому что основное внимание уделяется не параллелизму блоков программной логики, а параллельной обработке нескольких битов данных.

С SIMD параллелизм обеспечивается не потоками. Вместо этого современные процессоры предоставляют функциональность SIMD в виде «векторов» чисел (считайте, что это специализированные массивы) и команд, которые могут работать параллельно с этими числами; по сути, это низкоуровневые операции, использующие параллелизм уровня команд.

Попытки предоставить функциональность SIMD в языке JavaScript в первую очередь развивались под руководством ком-

пании Intel, а именно Мохаммеда Хагигата (Mohammad Haghighat) (на момент написания книги) в сотрудничестве с командами Firefox и Chrome. SIMD сейчас находится в начале пути принятия стандарта и имеет хорошие шансы на попадание в будущую версию JavaScript, вероятнее всего, в период ES7.

SIMD JavaScript предлагает открыть доступ коду JS к коротким векторным типам и API, которые в системах с поддержкой SIMD отображают операции прямо на эквивалентные средства процессора; в системах без поддержки SIMD используются непараллельные «прокладки совместимости» (shims).

Выигрыш по быстродействию в приложениях, интенсивно работающих с данными (анализ сигналов, матричные операции с графикой и т. д.), при параллельной обработке математических операций вполне очевиден!

В ранних предложениях SIMD API на момент написания книги выглядели примерно так:

```
var v1 = SIMD.float32x4( 3.14159, 21.0, 32.3, 55.55 );
var v2 = SIMD.float32x4( 2.1, 3.2, 4.3, 5.4 );

var v3 = SIMD.int32x4( 10, 101, 1001, 10001 );
var v4 = SIMD.int32x4( 10, 20, 30, 40 );

SIMD.float32x4.mul( v1, v2 );
    // [ 6.597339, 67.2, 138.89, 299.97 ]
SIMD.int32x4.add( v3, v4 );
    // [ 20, 121, 1031, 10041 ]
```

Здесь показаны два разных типа данных векторов: 32-разрядные числа с плавающей точкой и 32-разрядные целые числа. Как видно из листинга, размер этих векторов определяется равным четырем 32-разрядным элементам, так как это соответствует размерам векторов SIMD (128 бит) в большинстве современных процессоров. Возможно, в будущем мы увидим версии этих API с размером x8 (или больше!).

Кроме `mul()` и `add()`, с большой вероятностью будут включены другие операции, такие как `sub()`, `div()`, `abs()`, `neg()`, `sqrt()`, `reciprocal()`, `reciprocalSqrt()` (арифметическая), `shuffle()` (перестановка элементов векторов), `and()`, `or()`, `xor()`, `not()` (логическая), `equal()`, `greaterThan()`, `lessThan()` (сравнение), `shiftLeft()`, `shiftRightLogical()`, `shiftRightArithmetic()` (сдвиги), `fromFloat32x4()` и `fromInt32x4()` (преобразования).

Существует официальный «полифил» (будем надеяться, что это полифил, направленный в будущее) для доступной функциональности SIMD, который демонстрирует гораздо больше запланированных возможностей SIMD, чем представлено в этом разделе.

asm.js

asm.js — название подмножества языка JavaScript, которое поддерживает очень высокую степень оптимизации. Код в стиле asm.js, тщательно избегающий некоторых плохо оптимизируемых механизмов и паттернов (уборка мусора, преобразование типов и т. д.), распознается движком JS, и к нему применяются агрессивные низкоуровневые оптимизации.

В отличие от других механизмов повышения быстродействия, описанных в этой главе, asm.js не является чем-то таким, что обязательно должно быть принято в спецификацию языка JS. Спецификация asm.js существует, но она предназначена в основном для отслеживания согласованного набора кандидатов на оптимизацию, а не набора требований к движку JS.

В настоящее время никаких предложений в отношении нового синтаксиса не существует. Вместо этого asm.js предлагает способы выявления существующего стандартного синтаксиса JS, который соответствует правилам asm.js и позволяет движкам реализовать их собственные оптимизации соответствующим образом.

Как оптимизировать с asm.js

Первое, что следует понять об оптимизациях asm.js, относится к типам и преобразованиям типов. Если движку JS приходится отслеживать несколько разных типов значений в переменной при разных операциях, чтобы выполнять преобразования между типами по мере необходимости, это создает большое количество дополнительной работы, которая снижает эффективность оптимизации.

Мы будем использовать код в стиле asm.js для демонстрационных целей. Тем не менее обычно не ожидается, что вы будете писать такой код вручную. asm.js скорее рассматривается как цель компиляции, выполняемой другими инструментами, такими как Emscripten (https://github.com/kripken/emscripten/wiki). Конечно, возможно написать код asm.js самостоятельно, но обычно это очень плохая мысль, потому что код получается низкоуровневым, а управление им может оказаться делом слишком долгим и ненадежным. Тем не менее в отдельных случаях вам, возможно, придется вручную подстраивать свой код для целей оптимизации.

Некоторые приемы позволяют передать движку JS с поддержкой asm.js информацию о предполагаемом типе для переменных/операций, чтобы движок мог пропустить отслеживание информации для преобразования типов.

Пример:

```
var a = 42;
// ..
var b = a;
```

В этой программе присваивание `b = a` оставляет возможность расхождения типов переменных. Тем не менее эту команду также можно записать в виде:

```
var a = 42;
// ..
var b = a | 0;
```

Здесь используется операция | (бинарное ИЛИ) со значением 0. Она не оказывает влияния на значение кроме того, что она гарантирует, что это значение является 32-разрядным целым числом. При выполнении в обычном движке JS код работает нормально, но при выполнении в движке JS с поддержкой asm.js он может сообщать, что переменная `b` всегда должна рассматриваться как 32-разрядное целое число, так что отслеживание информации преобразования типов можно пропустить.

Аналогичным образом операция сложения двух переменных может быть ограничена более быстрым целочисленным сложением (вместо сложения с плавающей точкой):

```
(a + b) | 0
```

В этом случае движок JS с поддержкой asm.js также может воспринять подсказку и сделать вывод, что операция `+` должна быть 32-разрядным целочисленным сложением, потому что конечный результат всего выражения все равно будет автоматически преобразован в 32-разрядное целое число.

Модули asm.js

Среди основных факторов, снижающих быстродействие JS, следует отметить выделение памяти, уборку мусора и обращения к областям видимости. Одним из вариантов решения этих проблем asm.js предлагает объявить более формализованный «модуль» asm.js — не путайте с модулями ES6.

Для модуля asm.js следует явно передать строго соответствующее пространство имен (в спецификации оно обозначается `stdlib`, так как оно должно представлять необходимые стандартные библиотеки) для импортирования необходимых символических имен (вместо простого использования глобальных имен в лексической области видимости). В простейшем случае объект `window` может быть допустимым объектом `stdlib` для целей модуля asm.js, но вы можете (а возможно, и должны) сконструировать и более ограниченный вариант.

Также необходимо объявить *кучу* (heap) — зарезервированную область памяти, в которой переменные уже могут использоваться без требований дополнительной памяти или освобождения ранее использованной памяти, — и передать ее, чтобы модулю asm.js не приходилось делать ничего, что вызвало бы перераспределение памяти; он может просто использовать заранее зарезервированное пространство.

Скорее всего, куча будет представлять собой типизованный массив `ArrayBuffer`:

```
var heap = new ArrayBuffer( 0x10000 );  // куча 64k
```

При использовании этого предварительно зарезервированного 64-килобайтного пространства модуль asm.js может хранить и читать значения из этого буфера без потерь, связанных с выделением памяти или уборкой мусора. Например, буфер кучи может использоваться внутри модуля для хранения массива 64-разрядных вещественных значений:

```
var arr = new Float64Array( heap );
```

Рассмотрим простой, глупый пример модуля в стиле asm.js для демонстрации того, как эти компоненты работают в сочетании друг с другом. Определим функцию `foo(..)`, которая получает начальную (`x`) и конечную (`y`) целочисленную границу диапазона, вычисляет все внутренние произведения соседних чисел в диапазоне, а затем усредняет полученные значения:

```
function fooASM(stdlib,foreign,heap) {
    "use asm";

    var arr = new stdlib.Int32Array( heap );

    function foo(x,y) {
        x = x | 0;
        y = y | 0;

        var i = 0;
        var p = 0;
```

```
        var sum = 0;
        var count = ((y|0) - (x|0)) | 0;

        // вычислить все внутренние соседние произведения
        for (i = x | 0;
            (i | 0) < (y | 0);
            p = (p + 8) | 0, i = (i + 1) | 0
        ) {
            // сохранить результат
            arr[ p >> 3 ] = (i * (i + 1)) | 0;
        }

        // вычислить среднее по всем промежуточным значениям
        for (i = 0, p = 0;
            (i | 0) < (count | 0);
            p = (p + 8) | 0, i = (i + 1) | 0
        ) {
            sum = (sum + arr[ p >> 3 ]) | 0;
        }

        return +(sum / count);
    }

    return {
        foo: foo
    };
}

var heap = new ArrayBuffer( 0x1000 );
var foo = fooASM( window, null, heap ).foo;

foo( 10, 20 );      // 233
```

Этот пример для asm.js написан вручную для демонстрационных целей, так что он не содержит кода, который будет получен при компиляции для asm.js. Тем не менее он показывает типичный код asm.js, особенно подсказки типов и использование буфера кучи для хранения временных переменных.

Первый вызов `fooASM(..)` настраивает модуль asm.js с выделением памяти кучи. Результатом является функция `foo(..)`, которая может вызываться столько раз, сколько потребуется. Эти вызовы

`foo(..)` должны специально оптимизироваться движком JS с поддержкой asm.js. Что еще важнее, приведенный код полностью стандартен и должен нормально работать (без специальной оптимизации) в движке без поддержки asm.js.

Очевидно, природа ограничений, которые обеспечивают оптимизируемость кода asm.js, значительно сокращает возможные применения такого кода. asm.js не обязательно будет составлять обобщенный набор оптимизаций для любой заданной программы JS. Скорее, asm.js предполагает оптимизированный способ решения специализированных задач, таких как интенсивные математические вычисления (например, обработка графики для компьютерных игр).

Итоги

Первые четыре главы этой книги основаны на предположении о том, что паттерны асинхронного программирования позволяют писать более эффективный код, что обычно является очень важным улучшением. Однако возможности асинхронного поведения не бесконечны, потому что оно, по сути, связано с однопоточным циклом событий.

Поэтому в этой главе были представлены некоторые механизмы программного уровня, позволяющие дополнительно улучшить быстродействие.

Веб-работники позволяют запустить файл JS (то есть программу) в отдельном потоке, используя асинхронные события для передачи информации между потоками. Они прекрасно подходят для перемещения очень долгих или интенсивно потребляющих ресурсы задач в другой поток, чтобы снять нагрузку с основного UI-потока.

SIMD отображает параллельные математические операции уровня процессора на JavaScript API для высокопроизводительного

выполнения операций с параллелизмом уровня данных (например, числовой обработки больших наборов данных).

Наконец, asm.js описывает небольшое подмножество JavaScript, которое избегает частей JS, плохо поддающихся оптимизации (например, уборки мусора и преобразования типов), и позволяет движку JS распознавать и выполнять такой код посредством агрессивных оптимизаций. Код asm.js может быть написан вручную, но это крайне монотонная работа с высоким риском ошибок, сродни ручному написанию кода на языке ассемблера (отсюда и название). В первую очередь предполагается, что asm.js станет хорошей целью для кросс-компиляции программ из других высокооптимизируемых языков, например транспиляции C/C++ на JavaScript с использованием Emscripten.

Хотя в этой главе другие возможности не рассматриваются, сейчас на очень ранней стадии обсуждаются еще более радикальные идеи для JavaScript, включая аппроксимации прямой многопоточной функциональности (не просто скрытой за API структурой данных). Произойдет ли это явно или мы просто увидим, как параллелизм все в большей степени проникает в JS, будущее оптимизации программного уровня в JS выглядит многообещающе.

6 Хронометраж и настройка

Первые четыре главы этой книги были посвящены быстродействию как паттерну программирования (асинхронность и параллельное выполнение), а глава 5 была посвящена быстродействию на уровне макропрограммной архитектуры. В этой главе речь пойдет о быстродействии на микроуровне; мы сосредоточимся на отдельных выражениях/командах.

Одна из самых частых областей для экспериментов (некоторые разработчики буквально одержимы ею!) — анализ и тестирование различных вариантов написания строки или блока кода и определение того, какой вариант работает быстрее.

Мы рассмотрим некоторые аспекты микроуровня, но при этом важно с самого начала понимать, что эта глава написана не для того, чтобы подпитывать навязчивое стремление к оптимизации микроуровня (например, выполняет ли конкретный движок JS операцию `++a` быстрее, чем `a++`). Самое важное в этой главе — разобраться в том, какие виды оптимизаций в JS важны, а какие нет *и как отличить одни от других*.

Но еще перед тем, как браться за изучение материала, необходимо выяснить, как наиболее точно и надежно проводить тестирование

быстродействия JS, потому что бесчисленные неверные представления и мифы на эту тему забивают нашу коллективную базу знаний. Сначала нужно отсеять весь мусор, чтобы картина приобрела ясность.

Хронометраж

Постараюсь рассеять некоторые неверные представления. Бьюсь об заклад, что подавляющее большинство разработчиков JS, если им предложат измерить скорость (то есть время выполнения) некоторой операции, изначально предложат что-нибудь такое:

```
var start = (new Date()).getTime(); // или `Date.now()`
// выполнить некоторую операцию
var end = (new Date()).getTime();
console.log( "Duration:", (end - start) );
```

Поднимите руку, если нечто похожее пришло вам в голову. Да, я так и думал. Такой подход неверен во многих отношениях. Не стыдитесь, мы все там были.

Что именно сообщит вам полученный результат? Понимание того, что он говорит и не говорит о времени выполнения операции, ключ к правильному хронометражу быстродействия в JavaScript.

Если вы получите результат 0, может возникнуть впечатление, что выполнение заняло менее миллисекунды. Тем не менее это неточный результат. Некоторые платформы не поддерживают точность до одной миллисекунды, а обновляют таймер с большими приращениями. Например, старые версии Windows (а следовательно, IE) обеспечивали точность только до 15 мс, а это означает, что для получения ненулевого результата операция должна занимать не менее 15 мс!

Более того, для полученной продолжительности вы в действительности знаете лишь то, что операция заняла приблизительно столько времени в этом конкретном выполнении. Уверенность в том,

что она всегда будет выполняться с такой скоростью, близка к нулю. Вы понятия не имеете, не вмешивается ли движок или система в выполнение операции, а в любое другое время операция может выполняться быстрее.

А если вы получили продолжительность 4? Можно ли быть уверенными в том, что она заняла 4 мс? Нет. Операция могла занять меньше времени, а какая-то задержка могла привести к получению начального или конечного времени.

Но самое неприятное, что вы не можете быть уверены в том, что обстоятельства проведения теста не были излишне оптимистичными. Могло оказаться, что движок JS обнаружил способ оптимизации вашего изолированного тестового примера, а в реальной программе такая оптимизация окажется менее эффективной или невозможной, так что операция будет выполняться медленнее, чем в вашем тесте.

Итак... Что же мы узнали? К сожалению, как следует из этих рассуждений, узнали очень немного. Данные с такой низкой степенью доверия даже отдаленно не подходят для построения каких-то выводов. Ваши хронометражные данные практически бесполезны. И что еще хуже, они опасны в том отношении, что внушают ложную уверенность не только вам, но и другим разработчикам, которые не будут критически думать об условиях, приводящих к такому результату.

Повторение

Теперь вы говорите: «Ладно, давайте заключим операцию в цикл, чтобы весь тест занимал больше времени». Если повторить операцию 100 раз и выполнение всего цикла заняло 137 мс, можно разделить это значение на 100 и получить среднюю продолжительность 1,37 мс для каждой операции, верно? Не совсем так.

Тривиального математического усреднения определенно недостаточно для того, чтобы делать выводы о быстродействии, если

вы собираетесь экстраполировать результаты в масштабах всего приложения. С сотней итераций даже пара выбросов (аномально высоких или низких значений) может исказить среднее, а затем при повторном применении этого вывода искажение может быть раздуто до неимоверных пределов.

Вместо того чтобы просто отрабатывать фиксированное количество итераций, можно в цикле выполнять тесты до истечения некоторого интервала времени. Возможно, этот вариант более надежен, но как принять решение с продолжительностью выполнения? Казалось бы, логично предположить, что он должен быть кратен времени, необходимому для однократного выполнения операции. Неправильно.

На самом деле промежуток времени для повторений должен базироваться на точности используемого таймера, а именно на минимизации вероятности погрешности. Чем меньшей точностью обладает таймер, тем дольше нужно выполнять код, чтобы свести к минимуму вероятность ошибки. Пятнадцатимиллисекундный таймер очень плохо подходит для точного хронометража; чтобы неопределенность (то есть погрешность) была ниже 1 %, каждый цикл тестовых итераций должен выполняться не менее 750 мс. Для 1-миллисекундного таймера цикл должен выполняться всего 50 мс.

Но это всего лишь одна выборка. Чтобы быть уверенными в том, что вы исключили смещение, нужно иметь много выборок для усреднения. Также необходимо кое-что понимать относительно того, насколько медленно работает худшая выборка, насколько быстро работает самая лучшая выборка, насколько разнесены лучший и худший случаи и т. д. Нужно знать не только число, которое сообщает, насколько быстро что-то выполнялось, но и некоторую количественную метрику доверия к этому числу.

Возможно, вам также стоит объединить эти разные методы (и добавить к ним другие), чтобы добиться наилучшего баланса по всем возможным методам.

И это абсолютный минимум, просто для начала. Если вы подходили к хронометражу быстродействия менее серьезно... что ж, вам еще только предстоит узнать о правильном хронометраже.

Benchmark.js

Любые актуальные и надежные данные хронометража должны базироваться на статистически достоверных практических методах. Я не буду писать отдельную главу о статистике, поэтому просто буду пользоваться некоторыми терминами: «стандартное отклонение», «дисперсия», «погрешность». Если вы не знаете, что означают эти термины (я проходил курс статистики в колледже, так что у меня тоже нет полной ясности) — вашей квалификации недостаточно для написания собственной логики хронометража.

К счастью, умные люди, такие как Джон-Дэвид Дальтон (John-David Dalton) и Матиас Байненс (Mathias Bynens), понимают эти концепции, поэтому написали статистически достоверное средство хронометража Benchmark.js. Поэтому я не стану нагнетать напряжение и скажу: «Просто используйте этот инструмент».

Я не буду повторять всю документацию по работе Benchmark.js; для API доступна превосходная документация, которую стоит прочитать. Кроме того, в разных местах можно найти отличные статьи с описанием подробностей и методологии.

Но пока просто для демонстрации я покажу, как использовать Benchmark.js для проведения несложного теста быстродействия:

```
function foo() {
    // тестируемые операции
}

var bench = new Benchmark(
    "foo test",             // имя теста
    foo,                    // тестируемая функция
```

```
                                // (только содержимое)
    {
        // ..                   // дополнительные параметры
                                // (см. документацию)
    }
);

bench.hz;                       // количество операций в секунду
bench.stats.moe;                // погрешность
bench.stats.variance;           // дисперсия между выборками
// ..
```

О Benchmark.js необходимо знать гораздо *больше* поверхностного обзора, который я здесь даю. Суть в том, что Benchmark.js берет на себя все сложности настройки надежного, объективного и состоятельного хронометража для заданного блока кода JavaScript. Если вы собираетесь тестировать свой код и проводить хронометраж, начинать следует отсюда.

Здесь продемонстрировано применение Benchmark.js для тестирования одиночной операции (скажем, X), но на практике бывает нужно сравнить X с Y. Эта задача легко решается простым созданием двух разных тестов в *семействе* (организационная возможность Benchmark.js). Затем тесты выполняются «на равных», а на основании их статистик делается вывод о том, что работало быстрее — X или Y.

Конечно, Benchmark.js может использоваться для тестирования JavaScript в браузере (см. раздел «jsPerf.com» этой главы), но также возможна работа и в небраузерных средах (Node.js и т. д.).

Один из сценариев использования Benchmark.js, о котором часто незаслуженно забывают, — использование в средах разработки или контроля качества для проведения автоматизированных регрессионных тестов быстродействия для критических ветвей кода JavaScript вашего приложения. По аналогии с проведением наборов модульных тестов перед развертыванием вы также можете сравнить быстродействие с предыдущими данными и понять, повысилось или понизилось быстродействие приложения.

Setup и teardown

В предыдущем фрагменте мы обошли вниманием объект «дополнительных параметров» `{ .. }`. Однако о двух параметрах, `setup` и `teardown`, все же следует поговорить.

Эти два параметра определяют функции, которые должны вызываться до и после выполнения тестового примера.

Невероятно важно понимать, что код `setup` и `teardown` *не выполняется после каждой итерации теста*. Правильнее всего считать, что существуют два цикла: внешний (повторение наборов тестов) и внутренний (повторение итераций тестов). `setup` и `teardown` выполняются в начале и в конце каждой итерации *внешнего* цикла, но не во внутреннем цикле.

Почему это важно? Представьте, что у вас имеется тестовый пример, который выглядит так:

```
a = a + "w";
b = a.charAt( 1 );
```

Затем вы определяете код `setup` следующим образом:

```
var a = "x";
```

Возможно, вам кажется, что `a` начинает со значения `"x"` при каждой итерации теста.

Но это не так! Значение `"x"` будет присваиваться `a` для каждого набора тестов, после чего из-за повторяющихся конкатенаций `+ "w"` значение `a` будет становиться все длиннее, хотя вы по-прежнему обращаетесь только к символу `"w"` в позиции 1.

Чаще всего эта проблема проявляется при внесении изменений с побочными эффектами в DOM (например, присоединения дочернего элемента). Возможно, вы думаете, что родительский элемент каждый раз становится пустым, но на самом деле к нему

добавляется множество элементов, что может привести к значительному искажению результатов хронометража.

Все зависит от контекста

Не забывайте проверять контекст конкретного хронометража быстродействия, особенного сравнение между задачами X и Y. Даже если ваш тест показывает, что X быстрее Y, это не означает, что заключение «X быстрее Y» действительно обоснованно.

Допустим, тест быстродействия показывает, что X выполняет 10 000 000 операций в секунду, а Y выполняет 8 000 000 операций в секунду. Можно утверждать, что Y работает на 20 % медленнее X, и с математической точки зрения это будет правильно, но ваш вывод не настолько убедителен, как вам кажется.

Взгляните на результаты с более критических позиций: 10 000 000 операций в секунду — это 10 000 операций в миллисекунду, или 10 операций в миллисекунду. Другими словами, одна операция занимает 0,1 мс, или 100 нс. Трудно представить, насколько мал этот интервал. Для сравнения: часто утверждается, что человеческий глаз не способен различить что-либо, происходящее менее чем за 100 мс, а это в миллион раз медленнее 100-наносекундной скорости операции X.

Даже ранние научные исследования, показывавшие, что, возможно, мозг может обрабатывать события со скоростью до 13 мс (примерно в 8 раз быстрее предыдущего утверждения), означают лишь то, что X работает в 125 000 раз быстрее предельной скорости, воспринимаемой человеческим мозгом. *X выполняется очень, очень быстро*.

Но что еще важнее, стоит взглянуть на различия между X и Y — 2 000 000 операций в секунду. Если X выполняется за 100 нс, а Y — за 80 нс, разность составляет 20 нс, что в лучшем случае

составляет всего 1/650 000 интервала, воспринимаемого человеческим мозгом.

К чему я клоню? Что эти различия в быстродействия вообще не важны!

Но постойте, а если эта операция будет повторена много раз подряд? Разность будет суммироваться, верно?

Значит, мы задаемся другим вопросом: насколько вероятно, что операция X будет повторяться снова и снова, раз за разом, и это должно произойти 650 000 раз подряд, чтобы появилась хотя бы малая надежда, что разность будет заметна человеческому мозгу? А скорее всего, она должна быть выполнена от 5 000 000 до 10 000 000 раз в сплошном цикле, чтобы разность стала актуальной хотя бы в первом приближении.

Хотя ваш внутренний теоретик может возразить, что это возможно, более громкий голос реализма должен задуматься над тем, насколько это вероятно или маловероятно. Даже если это актуально в редких случаях, чаще всего это неактуально.

Подавляющее большинство результатов хронометража с малыми операциями — как в случае с мифом про `++x` и `x++` — совершенно недостоверно как основа вывода о том, что операции X должно отдаваться предпочтение перед Y на основании быстродействия.

Оптимизации движка

Вы просто не можете уверенно экстраполировать, что если в вашем изолированном тесте операция X выполнялась на 10 мс быстрее, чем Y, то это означает, что X всегда выполняется быстрее Y и должна всегда использоваться вместо нее. Быстродействие работает не так — все намного, намного сложнее.

Например, представьте (чисто гипотетически), что вам нужно протестировать некое поведение из области микробыстродействия, например, сравнение:

```
var twelve = "12";
var foo = "foo";

// тест 1
var X1 = parseInt( twelve );
var X2 = parseInt( foo );

// тест 2
var Y1 = Number( twelve );
var Y2 = Number( foo );
```

Если вы понимаете, что делает `parseInt(..)` по сравнению с `Number(..)`, можно предположить, что `parseInt(..)` теоретически приходится выполнять больше работы, особенно в случае `foo`. Или же что в случае `foo` в обоих случаях выполняется одинаковый объем работы, потому что обе функции могут остановиться на первом символе `f`.

Какое предположение верно? Честно говоря, не знаю. Но я возьмусь утверждать, что в данном случае это и не важно. Какие результаты могут быть получены при тестировании? Еще раз: я рассуждаю чисто гипотетически, я не пытался тестировать этот пример (и меня не интересует результат).

Допустим, тесты показывают, что X и Y статистически идентичны. Подтвердит ли это вашу гипотезу о символе `f`? Нет.

В нашем гипотетическом случае движок может понять, что переменные `twelve` и `foo` используются в каждом тесте только в одном месте, и подставить их значения в код. Тогда он может понять, что `Number( "12" )` можно заменить простым `12`. И возможно, он придет к тому же выводу с `parseInt(..)` — а может, и нет.

Или в процесс может вмешаться эвристика удаления неиспользуемого кода, которая определит, что переменные `X` и `Y` не используются, так что их объявления нерелевантны. В итоге движок вообще не будет ничего делать в обоих случаях.

И все это происходит только из предположений относительно одного запуска теста. Современные движки неизмеримо сложнее

того, о чем мы рассуждаем здесь. Они проделывают всевозможные трюки — например, отслеживание и трассировку поведения фрагмента кода за короткий период времени или с ограниченным набором входных данных.

А что, если движок оптимизирует определенную ветвь из-за фиксированных входных данных, но в реальной программе ввод будет более разнообразным и в ходе оптимизации будут приняты иные решения (или вообще никаких)? Или если движок применяет оптимизации, потому что видит, что функция хронометража выполняет код десятки тысяч раз, но в реальной программе он будет выполнен только сотню раз, и при таких условиях движок решит, что оптимизация не оправдает затраченных усилий?

И все эти предполагаемые оптимизации могут случиться в нашем ограниченном тесте, а потом движок не станет применять их в более сложной программе (по разным причинам). Или ситуация может быть иной: движок не будет оптимизировать тривиальный код, но будет склонен к применению агрессивных оптимизаций, когда более сложная программа повышает нагрузку на систему.

Я все это говорю к тому, что на самом деле вы точно не знаете, что происходит «под капотом». Никакие догадки и гипотезы не могут считаться веским основанием для принятия таких решений.

Означает ли это, что полезный хронометраж в принципе невозможен? Определенно нет!

Все сказанное сводится к тому, что тестирование *нереального* кода не даст *реальных* результатов. Если это возможно и оправданно, тестируйте реальные нетривиальные фрагменты своего кода и постарайтесь приблизить условия к реальным настолько, насколько это возможно. Только тогда полученные результаты имеют хоть какой-то шанс на отражение реального положения дел.

Микрохронометражные тесты вроде сравнения `++x` с `x++` с такой высокой вероятностью дадут бесполезные результаты, что можно сразу же считать их таковыми.

jsPerf.com

Хотя Benchmark.js пригодится при тестировании быстродействия вашего кода в любой среде JS, невозможно выразить, насколько важно собрать результаты тестирования из множества разных сред (настольные браузеры, мобильные устройства и т. д.), если вы надеетесь получить достоверные выводы из тестов.

Например, Chrome на высокопроизводительной настольной машине вряд ли будет работать хотя бы приблизительно с таким же быстродействием, как мобильная версия Chrome на смартфоне. А смартфон с полным зарядом аккумулятора вряд ли будет работать с таким же быстродействием, как смартфон с 2 % заряда, когда устройство понижает энергопотребление процессора.

Если вы хотите делать сколько-нибудь осмысленные выводы типа «X быстрее, чем Y» более чем для одной среды, вам придется протестировать как можно больше таких реальных сред. Если Chrome выполняет некую операцию X быстрее операции Y, это не означает, что та же картина будет наблюдаться во всех браузерах. И конечно, результаты тестовых запусков для разных браузеров должны быть сопоставлены с демографией ваших пользователей.

Для этой цели существует замечательный веб-сайт jsPerf. Он использует библиотеку Benchmark.js, упоминавшуюся выше, для проведения статистически точных и надежных тестов, причем проводит тест на общедоступном URL-адресе, который вы можете передавать другим.

Каждый раз при проведении теста результаты собираются и сохраняются с тестом. Накопленные результаты теста отображаются на диаграмме, с которой могут ознакомиться все желающие.

Создание теста на сайте начинается с двух тестовых примеров, но вы можете добавить столько, сколько сочтете нужным. Также предусмотрена возможность создания `setup`-кода, выполняемого

в начале каждого тестового цикла, и `teardown`-кода, выполняемого в конце каждого цикла.

Если вы хотите создать только один тестовый пример (если вы ведете хронометраж одного решения вместо сравнительного анализа разных решений), заполните входные поля второго теста произвольным текстом при создании, а потом отредактируйте тест и оставьте данные второго теста пустыми; это приведет к его удалению. Вы всегда сможете добавить новые тестовые примеры позднее.

Вы можете определить начальную подготовку страницы (импортирование библиотек, определение вспомогательных функций, объявление переменных и т. д.). Также предусмотрены параметры для определения `setup`/`teardown`-поведения при необходимости.

Проверка на здравый смысл

jsPerf — великолепный ресурс, но на нем опубликовано множество тестов, которые при анализе оказываются несостоятельными или неполноценными по различным причинам, описанным в этой главе.

Пример:

```
// Пример 1
var x = [];
for (var i=0; i<10; i++) {
    x[i] = "x";
}

// Пример 2
var x = [];
for (var i=0; i<10; i++) {
    x[x.length] = "x";
}
```

```
// Пример 3
var x = [];
for (var i=0; i<10; i++) {
    x.push( "x" );
}
```

Несколько моментов, на которые следует обратить внимание в этом тестовом примере:

- Разработчики очень часто включают циклы в свои тестовые примеры, забывая о том, что Benchmark.js уже обеспечивает все необходимое повторение. Циклы `for` в таких примерах почти наверняка являются абсолютно лишним шумом.
- Объявление и инициализация `x` включается в каждый тестовый пример — возможно, без надобности. Вспомните, о чем говорилось ранее: если команда `x = []` включается в код `setup`, она будет выполняться не при каждой тестовой итерации, а в начале каждой серии. Это означает, что `x` вырастет довольно основательно, а не до размера 10, подразумеваемого циклами `for`.
- Означает ли это, что мы хотим ограничить тестирование поведением движка JS с очень малыми массивами (размер 10)? Возможно, именно так и предполагается, но в таком случае необходимо задуматься над тем, не слишком ли сильно вы зацикливаетесь на второстепенных подробностях внутренней реализации.
- С другой стороны, не подразумевает ли создатель теста, что массивы будут расти до большого размера? Насколько поведение движка JS с большими массивами актуально и точно в сравнении с его предполагаемым применением в реальных программах?
- Хочет ли автор теста определить, как `x.length` или `x.push(..)` влияют на быстродействие операции присоединения элементов к массиву `x`? Да, это может быть допустимой целью теста. Но `push(..)` является вызовом функции, и разумеется, он будет медленнее обращения `[..]`. Пожалуй, примеры 1 и 2 дают более объективную картину, чем пример 3.

Другой пример, который также демонстрирует типичную ошибку «сравнения яблок с апельсинами»:

```
// Пример 1
var x = ["John","Albert","Sue","Frank","Bob"];
x.sort();

// Пример 2
var x = ["John","Albert","Sue","Frank","Bob"];
x.sort( function mySort(a,b){
    if (a < b) return -1;
    if (a > b) return 1;
    return 0;
} );
```

Очевидно, эти тесты предназначены для определения того, насколько пользовательский компаратор `mySort(..)` медленнее встроенного компаратора по умолчанию. Однако из-за использования функции `mySort(..)` как встроенного функционального выражения вы создаете несостоятельный/неполноценный тест. Во втором примере тестируется не только пользовательская функция JS, но и создание нового функционального выражения для каждой итерации.

Если провести аналогичный тест, но обновить его для изоляции только с целью создания встроенного функционального выражения (в отличие от заранее объявленной функции), вариант с созданием встроенного функционального выражения может работать от 2 до 20 % медленнее. Удивительно, не правда ли?

Если только в этом тесте вы не пытаетесь оценить затраты на создание встроенного функционального выражения, более правильный/честный тест поместил бы объявление `mySort(..)` в код настройки страницы (не в код `setup` теста, так как это приведет к избыточному переобъявлению для каждой серии) и просто обращался бы к нему по имени в тестовом примере: `x.sort(mySort)`.

В развитие темы предыдущего примера, здесь появляется еще одна потенциальная ловушка: непрозрачное исключение или добавление

лишней работы в один тестовый пример, который воссоздает уже упоминавшуюся ситуацию «сравнения яблок с апельсинами»:

```
// Пример 1
var x = [12,-14,0,3,18,0,2.9];
x.sort();

// Пример 2
var x = [12,-14,0,3,18,0,2.9];
x.sort( function mySort(a,b){
    return a - b;
} );
```

Если забыть об упоминавшейся ловушке со встроенным функциональным выражением, вызов `mySort(..)` из второго примера работает, потому что ему передаются числа; но конечно, со строками он бы завершился неудачей. В первом примере ошибка не выдается, но на самом деле он работает иначе и имеет другой результат! Это должно быть очевидно, но из-за различий в результатах между двумя тестовыми примерами тест почти наверняка станет недействительным!

Но помимо разных результатов, в данном случае встроенный компаратор `sort(..)` в действительности выполняет лишнюю работу, которую `mySort()` не выполняет — встроенный вариант преобразует сравниваемые значения в строки и выполняет лексикографическое сравнение. Первый фрагмент дает результат `[-14, 0, 0, 12, 18, 2.9, 3]`, тогда как второй фрагмент дает (что, скорее всего, лучше соответствует намерениям разработчика) `[-14, 0, 0, 2.9, 3, 12, 18]`.

Таким образом, этот тест некорректен, потому что в разных случаях выполняется разная работа. Любые результаты, которые вы получите, будут несостоятельными.

Такие ловушки бывают намного более коварными:

```
// Пример 1
var x = false;
var y = x ? 1 : 2;
```

```
// Пример 2
var x;
var y = x ? 1 : 2;
```

Возможно, разработчик здесь хочет протестировать быстродействие преобразования к логическому типу, которое будет выполнено оператором `? :`, если выражение `x` не является логическим. Следовательно, вас устраивает тот факт, что во втором случае выполняется лишняя работа по преобразованию типа.

Неочевидная проблема? В первом случае значение `x` задается, а во втором нет; в первом случае выполняется работа, которая не выполняется во втором. Чтобы устранить любое потенциальное (пусть и незначительное) смещение, можно поступить так:

```
// Пример 1
var x = false;
var y = x ? 1 : 2;

// Пример 2
var x = undefined;
var y = x ? 1 : 2;
```

В обоих случаях выполняется присваивание, поэтому тестируемый фактор — преобразование типа `x` или его отсутствие — с большей вероятностью будет более точно изолировано и протестировано.

Написание хороших тестов

Попробую сформулировать более важный факт.

Хорошее написание тестов требует точного анализа различий между двумя тестовыми примерами, а также характера этих различий (*намеренный* или *непредвиденный*).

Конечно, намеренные различия — абсолютно нормальное явление, но разработчик слишком легко создает непредвиденные различия,

способные исказить результаты. Вы должны действовать очень, очень осторожно, чтобы избежать этого смещения. Более того, вы можете заложить намеренные различия, но другим читателям вашего теста ваши намерения могут быть неясны, и они могут ошибочно усомниться в вашем тесте (или, наоборот, довериться ему). Что с этим делать?

Напишите более качественные, понятные тесты. Но также не жалейте времени на документирование (с использованием поля «Description» на сайте jsPerf.com и/или комментариев в коде) точных намерений вашего теста вплоть до мелких подробностей. Четко обозначьте намеренные различия; это поможет другим (и вам самому в будущем) быстрее выявить непредвиденные различия, которые могут влиять на результаты тестов.

Отделите аспекты, не относящиеся к вашему тесту, объявите их заранее в коде настройки страницы или теста, чтобы они были за пределами хронометрируемых частей теста.

Вместо того, чтобы пытаться сконцентрироваться на крошечном фрагменте реального кода и провести хронометраж именно этого фрагмента вне контекста, лучше включить в тесты и хронометражные данные более широкий контекст (без потери релевантности). Такие тесты выполняются медленнее, это означает, что любые выявленные различия будут более релевантны в контексте.

Микробыстродействие

До сих пор мы ходили вокруг да около различных аспектов микробыстродействия и в целом неодобрительно относились к излишней зацикленности на них. А сейчас я хочу заняться ими напрямую.

Первое, что необходимо понять для того, чтобы более уверенно чувствовать себя при хронометраже быстродействия вашего кода, — это то, что написанный вами код не всегда будет тем

кодом, который фактически выполняется движком. Эта тема была кратко представлена в главе 1, когда мы рассматривали переупорядочение команд компилятором, но здесь речь идет о другом: иногда компилятор может решить выполнить не тот код, который вы написали: не только выполнять команды в другом порядке, но запустить совсем другой код.

Рассмотрим следующий фрагмент кода:

```
var foo = 41;

(function(){
    (function(){
        (function(baz){
            var bar = foo + baz;
            // ..
        })(1);
    })();
})();
```

Обращение к `foo` на максимальном уровне вложенности требует трехуровневого поиска по области видимости. В книге «Область видимости и замыкания» этой серии рассказано о том, как работает лексическая область видимости. Компилятор обычно кэширует такие обращения, чтобы обращения к `foo` из разных областей видимости не создавали лишних затрат.

Однако здесь необходимо учитывать более глубокие факторы. Что, если компилятор осознает, что к `foo` нет других обращений, кроме этого места, а сама переменная не принимает других значений, кроме 41?

Вполне возможно, что компилятор JS решит просто полностью исключить значение переменной `foo` и подставить значение в код, как в следующем примере:

```
(function(){
    (function(){
        (function(baz){
            var bar = 41 + baz;
```

```
            // ..
        })(1);
    })();
})();
```

Конечно, компилятор также может провести аналогичный анализ и замену с переменной `baz`.

Но если вы начнете думать о своем коде JS не как о буквальном требовании, а как о рекомендации или подсказке для движка, то вы поймете, что зацикленность на отдельных синтаксических мелочах с большой вероятностью необоснованна.

Другой пример:

```
function factorial(n) {
    if (n < 2) return 1;
    return n * factorial( n - 1 );
}

factorial( 5 );     // 120
```

Старый добрый алгоритм вычисления факториала! Возможно, вы предполагаете, что движок JS в основном выполнит код «как есть». И откровенно говоря, это вполне возможно, просто я не уверен.

А теперь сюрприз: если написать тот же код на языке C и откомпилировать его с расширенными оптимизациями, компилятор поймет, что вызов `factorial(5)` можно заменить постоянным значением 120, полностью исключив вызов функции!

Более того, в некоторых движках предусмотрена так называемая *раскрутка рекурсии* — движок понимает, что выраженная вами рекурсия может быть выражена проще (то есть с оптимизацией) в виде цикла. Может оказаться, что движок JS перепишет предыдущий код для выполнения в следующем виде:

```
function factorial(n) {
    if (n < 2) return 1;
```

```
    var res = 1;
    for (var i=n; i>1; i--) {
        res *= i;
    }
    return res;
}

factorial( 5 );     // 120
```

А теперь представьте, что в предыдущем фрагменте вы беспокоитесь о том, что работает быстрее — `n * factorial(n-1)` или `n *= factorial(--n)`. Возможно, вы даже проведете хронометраж и попробуете определить, какой из вариантов лучше. Но вы упускаете из виду тот факт, что в более широком контексте движок может не выполнять ни ту ни другую строку кода, потому что он выполнит раскрутку рекурсии!

Раз уж речь зашла о `--`, ситуация «`--n` против `n--`» часто приводится как пример того, где выбор `--n` может стать микрооптимизацией, потому что теоретически он требует меньше усилий на уровне ассемблера.

В современном JavaScript подобные попытки бессмысленны. Заботу о таких вещах лучше доверить движку. Пишите тот код, который выглядит наиболее разумно. Сравните следующие три цикла `for`:

```
// Вариант 1
for (var i=0; i<10; i++) {
    console.log( i );
}

// Вариант 2
for (var i=0; i<10; ++i) {
    console.log( i );
}

// Вариант 3
for (var i=-1; ++i<10; ) {
    console.log( i );
}
```

Даже если у вас есть теория о том, что второй или третий вариант работает чуть-чуть быстрее первого (что в лучшем случае сомнительно), третий цикл наименее понятен, потому что `i` приходится начинать со значения `-1` из-за использования префиксной формы `++i`. А различия между первым и вторым вариантом на самом деле пренебрежимо малы.

Вполне возможно, что движок JS обнаружит место, в котором используется `i++`, и заменит его эквивалентом `++i`; это означает, что время, потраченное вами на выбор, было потрачено зря.

Другой пример глупой зацикленности на микрооптимизациях:

```
var x = [ .. ];

// Вариант 1
for (var i=0; i < x.length; i++) {
    // ..
}

// Вариант 2
for (var i=0, len = x.length; i < len; i++) {
    // ..
}
```

Теоретически длину массива `x` (которая остается неизменной) следует кэшировать в переменной `len`, чтобы избежать затрат на разрешение `x.length` при каждой итерации цикла.

Но если провести сравнительный хронометраж кода с `x.length` и кода с кэшированием в переменной `len`, вы убедитесь, что на практике измеренные различия статистически абсолютно нерелевантны.

Собственно, в некоторых движках (например, в v8) можно показать (http://mrale.ph/blog/2014/12/24/array-length-caching.html), что предварительное кэширование длины (вместо того, чтобы доверить ее определение движку) может слегка ухудшить быстродействие. Не пытайтесь превзойти движок JavaScript; в том, что касается оптимизаций быстродействия, вы почти наверняка проиграете.

Различия между движками

Разные движки JS в разных браузерах могут «соответствовать спецификации», но при этом совершенно по-разному работать с кодом. Спецификация JS не предъявляет никаких требований, связанных с быстродействием, — кроме разве что «оптимизации хвостового вызова» ES6, рассмотренной в разделе «Оптимизация хвостового вызова (TCO)» этой главы.

Движок может свободно решить, что должен уделить одной операции особое внимание для оптимизации, возможно, с ущербом для быстродействия другой операции. Будет очень трудно найти способ выполнения операции, который всегда работает быстрее во всех браузерах.

В сообществе разработчиков JS (особенно работающих с Node.js) существует движение, которое анализирует конкретные внутренние подробности реализации движка JavaScript v8 для принятия решений о написании кода JS, адаптированного для наиболее эффективного использования v8. Такие попытки позволяют достигнуть на удивление высокой степени оптимизации быстродействия, так что потраченные усилия окупятся.

Пара часто встречающихся примеров для v8:

- Не передавать переменную `arguments` между функциями, так как это замедляет реализацию функций.
- Изолировать `try..catch` в отдельной функции. У браузеров возникают проблемы с оптимизацией любых функций, содержащих `try..catch`, так что перемещение конструкции в отдельную функцию означает изоляцию вреда от деоптимизации с сохранением возможности оптимизации окружающего кода.

Но вместо того чтобы сосредоточиться на этих конкретных советах, давайте проанализируем подход к оптимизации только для v8 в более общем смысле.

Вы действительно пишете код, предназначенный для выполнения только в одном конкретном движке JS? Даже если ваш код *в данный момент* полностью предназначен для Node.js, насколько надежно предположение о том, что *всегда* будет использоваться именно движок JS v8? Возможно ли, что когда-нибудь, например, через несколько лет, появится другая платформа JS на стороне сервера, на которой вы решите выполнять свой код? Что, если ваши сегодняшние оптимизации будут работать намного медленнее на новом движке?

Или что, если ваш код всегда будет работать на v8, но v8 в какой-то момент решит изменить работу некоторого набора операций, и то, что работает быстро, будет работать медленнее, и наоборот?

Такие ситуации вовсе не относятся к чистой теории. Когда-то для выполнения конкатенации поместить несколько строковых значений в массив, а затем вызвать `join("")` для массива было быстрее, чем напрямую использовать `+` со значениями. Этот факт объяснялся подробностями внутренней реализации механизма хранения и управления строковыми значениями в памяти.

В результате в отрасли распространилось мнение, которое рекомендовало разработчикам всегда использовать метод с `join(..)`. И многие к этому мнению прислушались.

Вот только где-то по пути в движках JS сменился подход к внутреннему управлению строками, и усилия были направлены на оптимизацию конкатенации `+`. Они не замедляли операцию `join(..)` как таковую, но направили больше усилий на операцию `+`, которая все еще была достаточно распространенной.

Практика стандартизации или оптимизации конкретного метода на основании его широкого применения часто называется (метафорически) «асфальтированием коровьей тропы».

К сожалению, после того как новый подход к работе со строками и конкатенации прижился, весь существующий код, в котором

для конкатенации строк использовалась операция `join(..)` с массивами, стал субоптимальным.

Другой пример: одно время браузер Opera отличался от других браузеров способом упаковки/распаковки оберток для примитивов. Из-за этого разработчикам рекомендовалось использовать объект `String` вместо примитивного строкового значения, если программа должна была обращаться к свойствам (например, `length`) или методам (например, `charAt(..)`). На тот момент этот совет был правильным для Opera, но в других основных современных браузерах ситуация была обратной — в них оптимизации предназначались для строковых примитивов, а не для объектных оберток.

Я думаю, что подобные проблемы не исключены даже сегодня. По этой причине я очень осторожно отношусь к обширным оптимизациям в моем коде JS, основанным исключительно на подробностях реализации движка, *особенно если эти подробности присущи только одному движку*.

Также не стоит забывать об обратном: вы не обязаны изменять код для того, чтобы решить проблемы одного движка с выполнением кода на уровне с приемлемым быстродействием.

Исторически IE был источником многих огорчений такого рода, так как в старых версиях IE часто возникали проблемы с какими-то аспектами производительности, с которыми не было проблем в других основных браузерах. Недавно упоминавшаяся ситуация с конкатенацией строк была реальной проблемой во времена IE6 и IE7, когда от `join(..)` можно было добиться лучшего быстродействия, чем от `+`.

Тем не менее было бы неправильно считать, что проблемы с быстродействием в одном браузере оправдывают использование кода, который может оказаться субоптимальным во всех остальных браузерах. Даже если этот браузер занимает значительную долю аудитории вашего сайта, будет более практично написать правильный код и рассчитывать на то, что браузер со временем будет обновлен новыми улучшенными оптимизациями.

«Нет ничего более постоянного, чем временные решения». Скорее всего, код, который вы напишете сейчас для обхода некоторого недостатка быстродействия, переживет ошибку быстродействия в самом браузере.

В те времена, когда браузеры обновлялись только раз в пять лет, такие решения принимались сложнее. Но в наше время браузеры сплошь и рядом обновляются намного чаще (хотя очевидно, мобильный мир в этом отношении все еще отстает), и все они конкурируют за то, чтобы все лучше и лучше оптимизировать веб-технологии.

Если вы столкнулись с ситуацией, в которой у браузера возникают проблемы с быстродействием, отсутствующие у других, обязательно сообщите о проблеме разработчикам любым доступным способом. У большинства браузеров для этой цели имеются общедоступные системы отслеживания ошибок.

Я рекомендую искать пути решения проблем быстродействия в браузере только в том случае, если они не просто раздражают, а действительно делают работу невозможной. И еще стоит тщательно следить за тем, чтобы трюки с быстродействием не создали заметных негативных побочных эффектов в другом браузере.

Общая картина

Вместо того чтобы беспокоиться обо всех проблемах микробыстродействия, следует направить внимание на более общие и универсальные оптимизации. Как узнать, насколько они универсальны? Необходимо сначала понять, выполняется ли код по критическому пути программы. Если он не относится к критическому пути, скорее всего, ваши оптимизации особой пользы не принесут. Когда-нибудь слышали выражение «преждевременная оптимизация»? Оно происходит от знаменитого высказывания

Дональда Кнута: «Преждевременная оптимизация — корень всех зол». Многие разработчики приводят эту цитату в предположении, что большинство оптимизаций относится к «преждевременным», а следовательно, являются напрасной тратой времени. Как обычно, правда оказывается более сложной.

Вот как выглядит цитата Кнута в контексте (*курсив мой*)[1]:

> «Программисты тратят очень много времени, беспокоясь о скорости *некритических* частей своих программ; такие попытки на самом деле имеют серьезные отрицательные последствия для отладки и сопровождения кода. Стоит забыть о мелких оптимизациях где-то в 97 % случаев: преждевременная оптимизация — корень всех зол. Тем не менее мы не должны упускать возможности оптимизации в этих *критических* 3 %».
>
> Computing Surveys 6 (декабрь 1974)

Я думаю, это изречение Кнута можно перефразировать: «Оптимизация некритических путей — корень всех зол». Таким образом, нужно прежде всего определить, находится ли ваш код на критическом пути, и если находится, его стоит оптимизировать!

Можно пойти еще дальше: время, проведенное за оптимизацией критических путей, не пропадет даром, какой бы ничтожной ни была экономия; но никакая трата времени, проведенного за оптимизацией некритических путей, оправдана не будет.

Если ваш код находится на критическом пути (например, это «активный» фрагмент кода, который будет выполняться снова и снова или в местах, критических для взаимодействия с пользователем (например, цикл анимации или обновление стилей CSS), не жалейте сил на попытки поиска актуальных, объективно значительных оптимизаций.

---

[1] http://web.archive.org/web/20130731202547/http://pplab.snu.ac.kr/courses/adv_pl05/papers/p261-knuth.pdf

Для примера рассмотрим цикл анимации на критическом пути, который преобразует строковое значение в число. Конечно, это можно сделать несколькими разными способами, но какой из них работает быстрее (и есть ли он вообще)?

```
var x = "42";   // need number `42`

// Вариант 1: автоматическое выполнение неявного преобразования
var y = x / 2;

// Вариант 2: использование `parseInt(..)`
var y = parseInt( x, 0 ) / 2;

// Вариант 3: использование `Number(..)`
var y = Number( x ) / 2;

// Вариант 4: использование унарного оператора `+`
var y = +x / 2;

// Вариант 5: использование унарного оператора `|`
var y = (x | 0) / 2;
```

Если вам захочется проанализировать незначительные различия в быстродействии этих вариантов, можете написать соответствующий тест самостоятельно.

Если рассматривать представленные варианты, функция `parseInt(..)` справляется с задачей, но при этом делает намного больше — она разбирает строку, а не только выполняет преобразование типа. Есть основания предположить, что `parseInt(..)` — это более медленный вариант, которого, вероятно, следует избегать.

Конечно, если `x` окажется значением, которое *нуждается* в разборе (например, `"42px"` для стилей CSS), `parseInt(..)` останется единственным возможным вариантом!

`Number(..)` также является вызовом функции. С точки зрения поведения этот вариант идентичен варианту с унарным оператором `+`, но может работать немного медленнее, поскольку выполнение функции требует дополнительных действий. Также возможно, что

движок JS распознает симметрию поведения и просто подставит поведение `Number(..)` в код (то есть `+x`) за вас!

Но запомните: зацикленность на сравнениях `+x` с `x | 0` в большинстве случаев является напрасной тратой времени. Это проблема микробыстродействия, и вы не должны позволить ей повлиять на удобочитаемость программы.

Быстродействие чрезвычайно важно в критических путях вашей программы, но это не единственный фактор. Если доступно несколько вариантов, приблизительно сходных по быстродействию, следующим важным фактором должна стать удобочитаемость.

Оптимизация хвостовых вызовов (TCO)

Как кратко упоминалось выше, в ES6 включено конкретное требование, которое заходит в область оптимизаций. Оно связано с конкретной формой оптимизаций, которые могут применяться к вызовам функций: *оптимизацией хвостовых вызовов*.

В двух словах: «*хвостовым*» (tail) называется вызов функции, который располагается в «хвосте» другой функции, так что после завершения функции ничего делать не нужно (кроме разве что возврата возвращаемого значения).

Например, вот как выглядит нерекурсивная конфигурация с хвостовыми вызовами:

```
function foo(x) {
    return x;
}

function bar(y) {
    return foo( y + 1 );    // хвостовой вызов
}

function baz() {
```

```
    return 1 + bar( 40 );   // не хвостовой вызов
}

baz();                      // 42
```

`foo(y+1)` в `bar(..)` является хвостовым вызовом, потому что после завершения `foo(..)` функция `bar(..)` тоже завершается и ей остается только вернуть результат вызова `foo(..)`. С другой стороны, `bar(40)` хвостовым вызовом не является, потому что после завершения его возвращаемое значение должно быть увеличено на 1, прежде чем `baz()` сможет его вернуть.

Не углубляясь в подробности, вызов новой функции требует резервирования дополнительной памяти для управления стеком вызовов (*кадр стека*). Таким образом, предыдущий фрагмент обычно требует создания кадра стека для каждого из вызовов `baz()`, `bar(..)` и `foo(..)`.

Но если движок с поддержкой TCO поймет, что вызов `foo(y+1)` находится в хвостовой позиции (то есть функция `bar(..)` фактически завершена), то при вызове `foo(..)` он может обойтись без создания нового кадра стека, а вместо этого повторно использовать существующий кадр стека из `bar(..)`. Такой код не только быстрее работает, но и снижает затраты памяти.

В простом фрагменте подобные оптимизации не имеют особого эффекта, но они начинают играть более важную роль при рекурсии, особенно когда рекурсия приводит к созданию сотен или тысяч кадров стека. С TCO движок может выполнить все эти вызовы с одним кадром стека!

Рекурсия — неоднозначная тема в JS, потому что без TCO движкам приходится устанавливать произвольные (и разные!) предельные значения глубины рекурсии, на которых она должна останавливаться, чтобы предотвратить исчерпание памяти. С TCO рекурсивные функции с хвостовыми вызовами могут выполняться практически неограниченно, потому что никаких лишних затрат памяти нет!

Рассмотрим рекурсивную функцию `factorial(..)`, приведенную выше, но переписанную с расчетом на TCO:

```
function factorial(n) {
    function fact(n,res) {
        if (n < 2) return res;

        return fact( n - 1, n * res );
    }

    return fact( n, 1 );
}

factorial( 5 );     // 120
```

Эта версия `factorial(..)` остается рекурсивной, но она также может оптимизироваться средствами TCO, потому что оба внутренних вызова `fact(..)` находятся в хвостовой позиции.

Важно заметить, что TCO может применяться только при фактическом наличии хвостового вызова. Если вы пишете рекурсивные функции без хвостовых вызовов, быстродействие вернется к обычному выделению кадров стека, а лимиты движка на стек рекурсивных вызовов будут действовать. Многие рекурсивные функции могут быть переписаны в том виде, который был представлен для `factorial(..)`, но разработчик должен быть очень внимателен к мелочам.

Почему же ES6 требует, чтобы движки реализовали TCO, а не оставляет это на их усмотрение? Одна из причин заключается в том, что отсутствие TCO сокращает вероятность того, что некоторые алгоритмы будут реализованы на JS с использованием рекурсии из-за опасения ограничений стека вызовов. Если бы отсутствие TCO в движке просто приводило к отказоустойчивости при пониженном быстродействии во всех случаях, *требовать* это в ES6 было бы не обязательно. Но поскольку при отсутствии TCO некоторые программы могут стать неприемлемыми с практической точки зрения, это скорее важная возможность языка, чем скрытая подробность реализации.

ES6 гарантирует, что в дальнейшем разработчики JS смогут рассчитывать на эту оптимизацию во всех ES6+-совместимых браузерах. И это важный шаг для быстродействия JS!

Итоги

Для эффективного хронометража быстродействия фрагмента кода (особенно при сравнении его с другим вариантом того же кода для определения того, какой вариант быстрее) требуется тщательное внимание к подробностям.

Вместо того чтобы писать собственную статистически достоверную логику хронометража, воспользуйтесь библиотекой Benchmark.js, она сделает это за вас. Но будьте осторожны с написанием тестов: слишком легко написать тест, который выглядит разумно, но на самом деле некорректен — даже крошечные различия могут исказить результаты до полной недостоверности.

Важно получить как можно больше результатов тестов из максимального количества разных сред, чтобы устранить смещение оборудования/устройств. jsPerf.com — замечательный сайт для проведения хронометражных тестов.

К сожалению, многие распространенные тесты быстродействия зацикливаются на несущественных подробностях микробыстродействия (например, `x++` против `++x`). Чтобы писать хорошие тесты, вы должны уметь сосредоточиться на более общих аспектах, включая оптимизацию на критическом пути и умение обходить различные ловушки (например, подробности реализации разных движков JS).

Оптимизация хвостовых вызовов (TCO) в ES6 определяется как обязательная, потому что она позволяет применять на практике некоторые рекурсивные паттерны JS, которые были бы невозможны в остальных случаях. TCO позволяет выполнить вызов функции в конечной позиции другой функции без выделения дополнительных ресурсов. А это означает, что движку не придется устанавливать произвольные ограничения на глубину стека вызовов для рекурсивных алгоритмов.

Приложение А
Библиотека asynquence

В главах 1 и 2 довольно подробно рассматривались типичные паттерны асинхронного программирования и стандартные способы их реализации на базе обратных вызовов. Но также я показал, почему возможности обратных вызовов фатально ограниченны. Это привело нас к главам 3 и 4, посвященным обещаниям и генераторам, которые предоставляют гораздо более надежную, доверенную и удобную для анализа основу для построения асинхронности в программах.

Я уже несколько раз упоминал свою асинхронную библиотеку *asynquence* (от «async» + «sequence», то есть «асинхронность» + «последовательность»). Теперь я хочу кратко объяснить, как она работает и почему ее уникальная архитектура так важна и удобна.

В приложении Б будут рассмотрены некоторые расширенные асинхронные паттерны, но скорее всего, для практического применения этих паттернов лучше воспользоваться библиотекой. Мы используем *asynquence* для выражения этих паттернов, так что вам стоит заранее познакомиться с этой библиотекой.

Разумеется, *asynquence* не единственная библиотека для качественного асинхронного программирования, в этой области есть много отличных библиотек. Однако *asynquence* объединяет лучшие стороны всех этих паттернов в одной библиотеке, более того, она строится на одной базовой абстракции — (асинхронной) последовательности.

При разработке библиотеки я исходил из того, что в сложных программах JS фрагменты разных асинхронных паттернов часто требуется связать воедино, и в том, как это лучше сделать, обычно приходится разбираться самому разработчику. Вместо того чтобы подключать две и более разные асинхронные библиотеки, направленные на разные аспекты асинхронности, *asynquence* объединяет их в разнообразные фазы последовательностей, а вам достаточно изучить и установить всего одну базовую библиотеку.

На мой взгляд, библиотека *asynquence* достаточно удобна, а программирование асинхронной программной логики с семантикой в стиле обещаний с ней реализуется очень просто. По этой причине здесь я сосредоточусь исключительно на этой библиотеке.

Для начала я объясню принципы проектирования, заложенные в основу *asynquence*, а затем работа API будет продемонстрирована на примерах кода.

Последовательности и архитектура, основанная на абстракциях

Понимание *asynquence* начинается с понимания фундаментальной абстракции: любая серия шагов задачи (независимо от того, являются они по отдельности синхронными или асинхронными) в совокупности рассматривается как *последовательность*. Иначе говоря, последовательность является контейнером, представляющим задачу и состоящим из отдельных (возможно, асинхронных) шагов для завершения этой задачи.

Каждый шаг последовательности во внутренней реализации управляется обещанием (см. главу 3). Другими словами, каждый шаг, включаемый в последовательность, неявно создает обещание, которое связывается с предыдущим концом последовательности. Из-за семантики обещаний каждый отдельный шаг последователь-

ности является асинхронным, даже если этот шаг выполняется синхронно.

Кроме того, последовательность всегда следует линейно от шага к шагу; это означает, что шаг 2 всегда наступает после завершения шага 1, и т. д.

Конечно, от существующей последовательности может отходить новая последовательность; это означает, что ветвление произойдет только при достижении главной последовательностью соответствующей точки. Последовательности также могут объединяться различными способами, в том числе с включением одной последовательности в другую в конкретной точке потока выполнения.

Последовательность отчасти напоминает цепочку обещаний. Тем не менее с цепочками обещаний нет возможности создать «манипулятор», который бы обозначал цепочку в целом. Любое обещание, на которое у вас имеется ссылка, представляет только текущий шаг цепочки, а также другие шаги, отходящие от него. По сути, невозможно создать ссылку на цепочку обещаний, кроме создания ссылки на первое обещание в цепочке.

Тем не менее во многих ситуациях было бы весьма полезно иметь манипулятор, обозначающий всю цепочку в совокупности. Самая важная из таких ситуаций связана с отменой последовательностей. Как объяснялось в главе 3, обещания сами по себе отменяться не могут ни при каких условиях, потому что это нарушало бы фундаментальный принцип проектирования: внешнюю неизменяемость.

Но у последовательностей принципа неизменяемости не существует прежде всего потому, что последовательности не передаются как контейнеры будущих значений, которым необходима семантика неизменяемости значения. А значит, последовательности находятся на подходящем уровне абстракции для реализации поведения отмены. Последовательности *asynquence* могут быть отменены вызовом `abort()` в любое время: последовательность остановится в указанной точке и уже не продолжится.

Есть немало других причин, по которым абстракция последовательности поверх цепочек обещаний хорошо подходит для управления программной логикой.

Во-первых, объединение обещаний в цепочки должно происходить вручную, и этот процесс вам быстро надоест, когда вы начнете создавать и сцеплять обещания во множестве ваших программ. Раздражающее однообразие пойдет только во вред и заставит разработчика отказаться от использования обещаний в тех местах, где они были бы вполне уместны.

Абстракции должны избавлять от шаблонности и однообразия, поэтому абстракция последовательности становится хорошим решением проблемы. С обещаниями ваше внимание направлено на конкретный шаг, и ничего не предполагает продолжение цепочки. С последовательностями используется обратный подход: предполагается, что новые шаги будут добавляться до бесконечности.

Снижение сложности абстракции проявляется особенно сильно, если вспомнить о паттернах обещаний более высокого порядка (помимо `race([..])` и `all([..])`).

Например, в середине последовательности вам может понадобиться выразить шаг, концептуально сходный с `try..catch`: он всегда выполняется успешно, либо с предполагаемым основным результатом, либо с положительным сигналом для перехваченной ошибки. Или вы захотите выразить шаг, напоминающий цикл повторных попыток: один шаг повторяется снова и снова, пока не будет достигнут успех.

Выразить такие абстракции, используя только примитивы обещаний, достаточно сложно, а если вы начнете это делать в середине существующей цепочки обещаний, результат будет не из приятных. Но если вы абстрагируете свое мышление, выйдете на уровень последовательностей и будете рассматривать шаг как обертку для обещания, такая обертка может скрыть технические подробности и позволит свободно думать об управлении про-

граммной логикой на самом содержательном уровне, не отвлекаясь на мелочи.

Во-вторых (и это, возможно, более важно), рассмотрение асинхронной программной логики как шагов последовательности позволяет абстрагироваться от подробностей типов асинхронности, задействованных на каждом отдельном шаге. Где-то внутри обещание всегда управляет шагом, но снаружи этот шаг может выглядеть как обратный вызов продолжения (простой вариант по умолчанию), или как реальное обещание, или как генератор выполнения до завершения, или... В общем, вы поняли.

В-третьих, последовательности проще адаптируются для разных образов мышления — например, программирования событийного, потокового или основанного на активизации. *asynquence* предоставляет паттерн, который я называю *реактивными последовательностями* (reactive sequences) (см. далее). Это разновидность концепции активизируемых наблюдаемых объектов RxJS (Reactive Extensions), позволяющей многократным событиям активизироваться для каждого нового экземпляра последовательности. Обещания активизируются только один раз, так что выражать асинхронность с повторениями на базе одних обещаний довольно неудобно.

Другой альтернативный образ мышления инвертирует функциональность разрешения/управления в паттерне, который я называю *итерируемыми последовательностями*. Вместо того чтобы каждый отдельный шаг управлял своим собственным завершением (а следовательно, продвижением последовательности), последовательность инвертируется таким образом, что управление продвижением осуществляется через внешний итератор, а каждый шаг итерируемой последовательности просто реагирует на вызов управляющего метода `next(..)` итератора.

Все эти вариации будут рассмотрены в оставшейся части приложения, так что не беспокойтесь, если сейчас мы слишком быстро пройдем мимо этих возможностей.

Из всего сказанного следует, что последовательности являются более мощной и разумной абстракцией для сложной асинхронности, чем одни обещания (цепочки обещаний) или генераторы, а библиотека *asynquence* спроектирована для выражения этой абстракции с уровнем синтаксических удобств, который делает асинхронное программирование более понятным и приятным.

asynquence API

Для создания последовательности (экземпляра `asynquence`) используется функция `ASQ(..)`. Вызов `ASQ()` без параметров создает пустую исходную последовательность, тогда как в случае передачи `ASQ(..)` одного или нескольких значений или функций при создании последовательности каждый аргумент представляет исходные шаги последовательности.

В приведенных примерах кода я буду использовать идентификатор верхнего уровня asynquence в глобальном пространстве браузера: ASQ. Если вы включаете и используете asynquence через систему модулей (в браузере или на сервере), конечно, вы можете определить любое символическое имя на свое усмотрение — для asynquence это несущественно!

Многие методы API, описанные в этом приложении, встроены в базовую часть *asynquence*, другие предоставляются при включении необязательного пакета плагинов *contrib*. За информацией о том, является метод встроенным или определяется через плагин, обращайтесь к документации *asynquence*.

Шаги

Если функция представляет собой обычный шаг последовательности, в первом параметре передается обратный вызов продолжения, а все последующие параметры — сообщения, передаваемые

с предыдущего шага. Шаг не завершится до тех пор, пока не будет активизирован обратный вызов продолжения. После вызова любые переданные аргументы будут отправлены как сообщения следующему шагу последовательности.

Чтобы добавить новый обычный шаг последовательности, вызовите метод `then(..)` (который имеет практически такую же семантику, как вызов `ASQ(..)`):

```
ASQ(
    // шаг 1
    function(done){
        setTimeout( function(){
            done( "Hello" );
        }, 100 );
    },
    // шаг 2
    function(done,greeting) {
        setTimeout( function(){
            done( greeting + " World" );
        }, 100 );
    }
)
// шаг 3
.then( function(done,msg){
    setTimeout( function(){
        done( msg.toUpperCase() );
    }, 100 );
} )
// шаг 4
.then( function(done,msg){
    console.log( msg );         // HELLO WORLD
} );
```

Хотя имя `then(..)` совпадает с именем функции из встроенного Promise API, эта функция `then(..)` отличается от нее. Вы можете передать `then(..)` столько функций или значений, сколько посчитаете нужным; они интерпретируются как разные шаги. Семантика с двумя обратными вызовами «выполнение/отказ» не используется.

В отличие от обещаний, у которых для сцепления одного обещания со следующим необходимо создать и вернуть это обещание из обработчика выполнения `then(..)`, в *asynquence* для этого достаточно активизировать обратный вызов продолжения. Я всегда присваиваю ему имя `done()`, но вы можете назвать его так, как считаете нужным, а также передать сообщения завершения в аргументах.

Предполагается, что каждый шаг, определяемый `then(..)`, является асинхронным. Если у вас имеется синхронный шаг, вы можете либо просто немедленно вызвать `done(..)`, либо использовать более простую вспомогательную функцию `val(..)`:

```
// шаг 1 (синхронный)
ASQ( function(done){
    done( "Hello" );    // ручная синхронность
} )
// шаг 2 (синхронный)
.val( function(greeting){
    return greeting + " World";
} )
// шаг 3 (асинхронный)
.then( function(done,msg){
    setTimeout( function(){
        done( msg.toUpperCase() );
    }, 100 );
} )
// шаг 4 (синхронный)
.val( function(msg){
    console.log( msg );
} );
```

Как видите, шаги с `val(..)`-активизацией не получают обратного вызова продолжения, так как эта часть предполагается по умолчанию, и в результате список параметров становится более компактным! Чтобы отправить сообщение со следующим шагом, просто используйте `return`.

Можно считать, что `val(..)` представляет синхронный шаг «только значение», который может пригодиться для синхронных операций со значениями, ведения журнала и т. д.

Ошибки

Одно важное отличие *asynquence* от обещаний — обработка ошибок.

Со значениями каждое отдельное значение (шаг) в цепочке может иметь собственную независимую ошибку, и каждый последующий шаг имеет возможность обработать (или не обработать) ошибку. Главная причина для такой семантики (снова) происходит от ориентации на отдельные обещания вместо цепочки (последовательности) как единого целого.

Я считаю, что в большинстве случаев ошибка в одной части последовательности обычно не допускает восстановления, так что последующие шаги последовательности теряют актуальность и могут быть пропущены. Таким образом, по умолчанию ошибка на любом шаге последовательности переводит всю последовательность в режим ошибки, а остальные нормальные шаги игнорируются.

Если вам понадобится создать шаг, ошибки на котором допускают возможность восстановления, для этой цели можно воспользоваться некоторыми методами API, например `try(..)` (ранее он упоминался как разновидность шага `try..catch`) или `until(..)` (цикл повторных попыток: один шаг повторяется снова и снова, пока не будет достигнут успех или цикл не будет прерван ручным вызовом `break()`). У *asynquence* даже есть методы `pThen(..)` и `pCatch(..)`, которые работают аналогично методам обещаний `then(..)` и `catch(..)` (см. главу 3), так что при желании вы можете выполнить локальную обработку ошибок в середине последовательности.

Доступны оба варианта, но по моему опыту, чаще встречается вариант по умолчанию. С обещаниями для получения цепочки шагов, игнорирующей все шаги с момента возникновения ошибки, необходимо действовать внимательно и не зарегистрировать обработчик отказа на каком-либо шаге; в противном случае ошибка будет поглощена как обработанная, а последовательность может

продолжиться (возможно, неожиданно для вас). Правильно и надежно реализовывать такое поведение будет неудобно.

Для регистрации обработчика уведомления об ошибке для последовательности `asynquence` предоставляет метод последовательности `or(..)`, у которого также имеется синоним `onerror(..)`. Этот метод можно вызвать в любой точке последовательности, при этом вы можете зарегистрировать столько обработчиков, сколько сочтете нужным. Это позволяет нескольким разным потребителям прослушивать последовательность и определять, произошла ошибка или нет; в этом отношении метод действует как своего рода обработчик события ошибки.

Как и в случае с обещаниями, все исключения JS становятся ошибками последовательности; также можно выдать сигнал об ошибке последовательности на программном уровне:

```
var sq = ASQ( function(done){
    setTimeout( function(){
        // выдать сигнал ошибки для последовательности
        done.fail( "Oops" );
    }, 100 );
} )
.then( function(done){
    // управление сюда не передается
} )
.or( function(err){
    console.log( err );         // Упс!
} )
.then( function(done){
    // сюда тоже не передается
} );
// позднее
sq.or( function(err){
    console.log( err );         // Упс!
} );
```

Другое очень важное отличие обработки ошибок в *asynquence* по сравнению с обещаниями — поведение по умолчанию для необработанных исключений. Как подробно обсуждалось в главе 3,

отклоненное обещание без зарегистрированного обработчика отказа просто незаметно поглощает ошибку; вы должны помнить о том, что цепочка должна завершаться финальным вызовом `catch(..)`.

В *asynquence* все наоборот.

Если в последовательности происходит ошибка и если *на этот момент* обработчики не зарегистрированы, сообщение об ошибке направляется на `console`. Другими словами, последовательность по умолчанию всегда сообщает о необработанных отказах, чтобы они не были поглощены и потеряны.

Как только вы регистрируете обработчик ошибок для последовательности, он выводит последовательность из этих оповещений, чтобы предотвратить дублирование шума. Возможно, в каких-то случаях вам потребуется создать последовательность, которая может перейти в состояние ошибки до того, как у вас будет возможность зарегистрировать обработчик. Такая ситуация нетипична, но она встречается время от времени.

В таких случаях также можно вывести экземпляр последовательности из режима уведомления об ошибках, вызвав `defer()` для последовательности. Это следует делать только в том случае, если вы уверены, что в конечном итоге ошибки будут обработаны:

```
var sq1 = ASQ( function(done){
    doesnt.Exist();         // выдаст исключение в консоль
} );

var sq2 = ASQ( function(done){
    doesnt.Exist();         // выдаст только ошибку
последовательности
} )
// вывести из режима уведомления об ошибках
.defer();

setTimeout( function(){
    sq1.or( function(err){
        console.log( err ); // ReferenceError
    } );
```

```
    sq2.or( function(err){
        console.log( err ); // ReferenceError
    } );
}, 100 );

// ReferenceError (из sq1)
```

Такое поведение обработки ошибок лучше того, что предоставляют сами обещания, потому что оно строится по принципу «бездны успеха», а не «бездны отчаяния» (см. главу 3).

Если последовательность подключается к другой последовательности (см. раздел «Объединение последовательностей» этого приложения), то исходная последовательность выводится из режима уведомлений об ошибках, и теперь следует рассматривать уведомления об ошибках целевой последовательности (или их отсутствие).

Параллельные шаги

Не все шаги ваших последовательностей будут ограничиваться выполнением только одной (асинхронной) задачи; некоторым понадобится выполнять несколько шагов параллельно. Шаг последовательности, в котором несколько подшагов обрабатываются параллельно, называется *шлюзом* `gate(..)` — также имеется псевдоним `all(..)`, если вы предпочитаете; он симметричен встроенному методу `Promise.all([..])`.

Если все шаги `gate(..)` завершаются успешно, то все сообщения об успехе будут переданы следующему шагу последовательности. Если какие-то из них порождают ошибки, то вся последовательность немедленно переходит в состояние ошибки.

Пример:

```
ASQ( function(done){
    setTimeout( done, 100 );
} )
```

```
.gate(
    function(done){
        setTimeout( function(){
            done( "Hello" );
        }, 100 );
    },
    function(done){
        setTimeout( function(){
            done( "World", "!" );
        }, 100 );
    }
)
.val( function(msg1,msg2){
    console.log( msg1 );    // Hello
    console.log( msg2 );    // [ "World", "!" ]
} );
```

Для наглядности сравним этот пример с реализацией на базе встроенных обещаний:

```
new Promise( function(resolve,reject){
    setTimeout( resolve, 100 );
} )
.then( function(){
    return Promise.all( [
        new Promise( function(resolve,reject){
            setTimeout( function(){
                resolve( "Hello" );
            }, 100 );
        } ),
        new Promise( function(resolve,reject){
            setTimeout( function(){
                // примечание: здесь необходим массив [ ]
                resolve( [ "World", "!" ] );
            }, 100 );
        } )
    ] );
} )
.then( function(msgs){
    console.log( msgs[0] ); // Hello
    console.log( msgs[1] ); // [ "World", "!" ]
} );
```

Ужасно. С обещаниями требуется намного больше шаблонного кода для выражения той же асинхронной программной логики. Этот пример наглядно показывает, почему API и абстракции *asynquence* существенно упрощают реализацию шагов с обещаниями. И чем сложнее асинхронность, тем больше выигрыш.

Разновидности шагов

В плагинах *contrib* включен ряд разновидностей шагов `gate(..)`, которые могут быть весьма полезными:

- `any(..)` — аналог `gate(..)`, в котором для продолжения основной последовательности один сегмент должен завершиться успешно.
- `first(..)` — аналог `any(..)`, за исключением того, что основная последовательность продолжается, как только будет успешно завершен любой сегмент (последующие результаты от других сегментов игнорируются).
- `race(..)` (симметрично `Promise.race([..])`) — аналог `first(..)`, за исключением того, что основная последовательность продолжается, как только будет завершен любой сегмент (успешно или неудачно).
- `last(..)` — аналог `any(..)`, за исключением того, что последний успешно завершенный сегмент отправляет свое сообщение(-я) по основной последовательности.
- `none(..)` — противоположность `gate(..)`: основная последовательность продолжается только в том случае, если все сегменты завершаются неудачей (все сообщения об ошибках меняются с сообщениями об успехе, и наоборот).

Начнем с определения нескольких вспомогательных функций, которые сделают объяснение более наглядным:

```
function success1(done) {
    setTimeout( function(){
        done( 1 );
```

```
    }, 100 );
}

function success2(done) {
    setTimeout( function(){
        done( 2 );
    }, 100 );
}

function failure3(done) {
    setTimeout( function(){
        done.fail( 3 );
    }, 100 );
}

function output(msg) {
    console.log( msg );
}
```

А теперь продемонстрируем эти разновидности шагов `gate(..)`:

```
ASQ().race(
    failure3,
    success1
)
.or( output );      // 3

ASQ().any(
    success1,
    failure3,
    success2
)
.val( function(){
    var args = [].slice.call( arguments );
    console.log(
        args        // [ 1, undefined, 2 ]
    );
} );

ASQ().first(
    failure3,
    success1,
    success2
)
.val( output );     // 1
```

```
ASQ().last(
    failure3,
    success1,
    success2
)
.val( output );     // 2

ASQ().none(
    failure3
)
.val( output )      // 3
.none(
    failure3
    success1
)
.or( output );      // 1
```

Другая разновидность шагов `map(..)`позволяет асинхронно отобразить элементы массива на разные значения. Шаг не продолжается, пока не будут завершены все отображения. Шаг `map(..)` очень похож на `gate(..)`, за исключением того, что он берет исходные значения из массива вместо отдельно заданных функций, а вы определяете функцию обратного вызова, которая должна быть применена к каждому значению:

```
function double(x,done) {
    setTimeout( function(){
        done( x * 2 );
    }, 100 );
}

ASQ().map( [1,2,3], double )
.val( output );                 // [2,4,6]
```

Кроме того, `map(..)` может получить любой из своих параметров (массив или обратный вызов) из сообщений, отправленных с предыдущего шага:

```
function plusOne(x,done) {
    setTimeout( function(){
        done( x + 1 );
```

```
    }, 100 );
}

ASQ( [1,2,3] )
.map( double )          // поступает сообщение `[1,2,3]`
.map( plusOne )         // поступает сообщение `[2,4,6]`
.val( output );         // [3,5,7]
```

Еще одна разновидность — `waterfall(..)`, отчасти напоминает гибрид поведения сбора сообщений `gate(..)` с последовательной обработкой `then(..)`.

Сначала выполняется шаг 1, затем сообщение об успехе с шага 1 передается шагу 2, далее оба сообщения об успехе передаются шагу 3, после чего все три сообщения об успехе поступают на шаг 4 и т. д.; таким образом, сообщения в каком-то смысле собираются и опускаются по каскадному принципу.

Пример:

```
function double(done) {
    var args = [].slice.call( arguments, 1 );
    console.log( args );

    setTimeout( function(){
        done( args[args.length - 1] * 2 );
    }, 100 );
}

ASQ( 3 )
.waterfall(
    double,                 // [ 3 ]
    double,                 // [ 6 ]
    double,                 // [ 6, 12 ]
    double                  // [ 6, 12, 24 ]
)
.val( function(){
    var args = [].slice.call( arguments );
    console.log( args );    // [ 6, 12, 24, 48 ]
} );
```

Если в какой-то момент в «каскаде» происходит ошибка, вся последовательность немедленно переходит в состояние ошибки.

Устойчивость к ошибкам

Иногда бывает нужно управлять ошибками на уровне шагов, не позволяя им отправлять всю последовательность в состояние ошибки. *asynquence* предлагает две разновидности шагов для этой цели.

`try(..)` пытается выполнить шаг; если попытка завершается успехом, то последовательность продолжается как обычно, а если происходит ошибка, то она преобразуется в сообщение об успехе, отформатированное в виде `{ catch: .. }` с подстановкой сообщения(-й) об ошибке:

```
ASQ()
.try( success1 )
.val( output )          // 1
.try( failure3 )
.val( output )          // { catch: 3 }
.or( function(err){
    // управление сюда не передается
} );
```

Вместо этого можно запустить цикл повторных попыток с использованием `until(..)`. Он пытается выполнить шаг, и если попытка завершается неудачей, повторяет шаг на следующем тике цикла событий, и т. д.

Цикл повторных попыток может продолжаться бесконечно, но если вы хотите выйти из него, вызовите метод `break()` для триггера завершения; при этом основная последовательность переводится в состояние ошибки:

```
var count = 0;

ASQ( 3 )
.until( double )
.val( output )                  // 6
.until( function(done){
    count++;
    setTimeout( function(){
        if (count < 5) {
```

```
            done.fail();
        }
        else {
            // выйти из цикла повторных попыток `until(..)`
            done.break( "Oops" );
        }
    }, 100 );
} )
.or( output );                  // Упс!
```

Шаги в стиле обещаний

Если вы предпочитаете иметь встроенную в вашу последовательность семантику в стиле обещаний (например, `then(..)` и `catch(..)` у обещаний) (см. главу 3), используйте плагины `pThen` и `pCatch`:

```
ASQ( 21 )
.pThen( function(msg){
    return msg * 2;
} )
.pThen( output )                // 42
.pThen( function(){
    // выдать исключение
    doesnt.Exist();
} )
.pCatch( function(err){
    // перехват исключения (отказ)
    console.log( err );         // ReferenceError
} )
.val( function(){
    // основная последовательность снова находится
    // в состоянии успеха, потому что
    // предыдущее исключение было перехвачено
    // `pCatch(..)`
} );
```

`pThen(..)` и `pCatch(..)` спроектированы для выполнения в последовательности, но ведут себя так, словно они являются обычной цепочкой обещаний. Соответственно, вы можете разрешать подлинные обещания или последовательности *asynquence* из обработчика выполнения, переданного `pThen(..)` (см. главу 3).

Ветвление последовательностей

Одна из полезных возможностей обещаний — это присоединение нескольких регистраций обработчиков `then(..)` к одному обещанию, что фактически приводит к ветвлению программной логики на этом обещании:

```
var p = Promise.resolve( 21 );

// ветвление 1 (из `p`)
p.then( function(msg){
    return msg * 2;
} )
.then( function(msg){
    console.log( msg );     // 42
} )

// ветвление 2 (из `p`)
p.then( function(msg){
    console.log( msg );     // 21
} );
```

В *asynquence* аналогичное ветвление легко реализуется вызовом `fork()`:

```
var sq = ASQ(..).then(..).then(..);

var sq2 = sq.fork();

// ветвление 1
sq.then(..)..;

// ветвление 2
sq2.then(..)..;
```

Объединение последовательностей

Если вы хотите выполнить операцию, обратную ветвлению `fork()`, объедините две последовательности методом экземпляра `seq(..)`:

```
var sq = ASQ( function(done){
    setTimeout( function(){
        done( "Hello World" );
```

```
    }, 200 );
} );

ASQ( function(done){
    setTimeout( done, 100 );
} )
// подключение последовательности `sq` к этой последовательности
.seq( sq )
.val( function(msg){
    console.log( msg );     // Hello World
} )
```

`seq(..)` может передавать как саму последовательность, как показано здесь, так и функцию. Если `seq(..)` получает функцию, ожидается, что функция будет возвращать последовательность при вызове, поэтому предыдущий код мог бы быть реализован так:

```
// ..
.seq( function(){
    return sq;
} )
// ..
```

Этот же шаг может быть реализован с использованием `pipe(..)`:

```
// ..
.then( function(done){
    // присоединение `sq` к обратному вызову продолжения `done`
    sq.pipe( done );
} )
// ..
```

При подключении последовательности также подключаются потоки сообщений об успехе и ошибках.

Как упоминалось в предыдущем примечании, подключение (ручное с `pipe(..)` или автоматическое с `seq(..)`) выводит исходную последовательность из системы уведомления об ошибках, но не влияет на статус уведомления об ошибках целевой последовательности.

Значение и последовательности ошибки

Если хотя бы один шаг последовательности представляет собой нормальное значение, это значение отображается на сообщение завершения этого шага:

```
var sq = ASQ( 42 );
sq.val( function(msg){
    console.log( msg );     // 42
} );
```

Если вы хотите создать последовательность, которая автоматически переходит в ошибочное состояние:

```
var sq = ASQ.failed( "Oops" );

ASQ()
.seq( sq )
.val( function(msg){
    // сюда управление не передается
} )
.or( function(err){
    console.log( err );     // Oops
} );
```

Возможно, вам также потребуется автоматически создать последовательность с отложенным значением или отложенной ошибкой. При использовании плагинов `after` или `failAfter` это делается легко:

```
var sq1 = ASQ.after( 100, "Hello", "World" );
var sq2 = ASQ.failAfter( 100, "Oops" );

sq1.val( function(msg1,msg2){
    console.log( msg1, msg2 );      // Hello World
} );

sq2.or( function(err){
    console.log( err );             // Oops
} );
```

Вы также можете вставить задержку в середину последовательности вызовом `after(..)`:

```
ASQ( 42 )
// вставить задержку в последовательность
.after( 100 )
.val( function(msg){
    console.log( msg );     // 42
} );
```

Обещания и обратные вызовы

Я считаю, что последовательности *asynquence* существенно расширяют возможности встроенных обещаний. В большинстве случаев работать на таком уровне абстракции приятнее, и он предоставляет больше возможностей. Тем не менее вы почти наверняка столкнетесь с практической необходимостью интеграции *asynquence* с другим кодом, не использующим *asynquence*.

Обещание (например, «thenable» — см. главу 3) можно легко подключить к последовательности методом экземпляра `promise(..)`:

```
var p = Promise.resolve( 42 );

ASQ()
.promise( p )           // также возможно: function(){ return p;
}
.val( function(msg){
    console.log( msg ); // 42
} );
```

А если потребуется пойти в противоположном направлении и ответвить обещание от последовательности в какой-то точке, используйте плагин `toPromise` из пакета *contrib*:

```
var sq = ASQ.after( 100, "Hello World" );

sq.toPromise()
// теперь это стандартная цепочка обещаний
```

```
.then( function(msg){
    return msg.toUpperCase();
} )
.then( function(msg){
    console.log( msg );     // HELLO WORLD
} );
```

Для адаптации *asynquence* к системам, использующим обратные вызовы, существует несколько встроенных средств. Чтобы автоматически сгенерировать из последовательности обратный вызов в стиле «ошибка на первом месте» для подключения к средствам, ориентированным на обратные вызовы, используйте значение `errfcb`:

```
var sq = ASQ( function(done){
// примечание: ожидает получить обратный вызов
// в стиле "ошибка на первом месте"
    someAsyncFuncWithCB( 1, 2, done.errfcb )
} )
.val( function(msg){
    // ..
} )
.or( function(err){
    // ..
} );

// примечание: ожидает получить обратный вызов
// в стиле "ошибка на первом месте"
anotherAsyncFuncWithCB( 1, 2, sq.errfcb() );
```

Также может возникнуть необходимость в создании версии вспомогательной функции, «обернутой» в последовательность (вспомните «фабрики обещаний» из главы 3 и «фабрики преобразователей» из главы 4). Для этой цели *asynquence* предоставляет метод `ASQ.wrap(..)`:

```
var coolUtility = ASQ.wrap( someAsyncFuncWithCB );

coolUtility( 1, 2 )
.val( function(msg){
    // ..
```

```
} )
.or( function(err){
    // ..
} );
```

Для полноты картины введем термин «фабрика последовательностей» для функции, которая производит последовательность на базе `ASQ.wrap(..)`, — такой как `coolUtility` в приведенном примере.

Итерируемые последовательности

В нормальной парадигме последовательностей каждый шаг сам по себе несет ответственность за свое завершение, которое обеспечивает продвижение последовательности. Обещания работают аналогичным образом.

К сожалению, иногда требуется иметь внешний контроль над обещанием/шагом, что приводит к громоздким конструкциям извлечения функциональности.

Рассмотрим следующий пример обещаний:

```
var domready = new Promise( function(resolve,reject){
    // не хочется здесь размещать эту команду,
    // потому что она логически относится
    // к другой части кода
    document.addEventListener( "DOMContentLoaded", resolve );
} );

// ..

domready.then( function(){
    // модель DOM готова!
} );
```

Антипаттерн извлечения функциональности с обещаниями выглядит так:

```
var ready;

var domready = new Promise( function(resolve,reject){
    // извлечение функциональности `resolve()`
    ready = resolve;
} );

// ..

domready.then( function(){
    // DOM is ready!
} );

// ..

document.addEventListener( "DOMContentLoaded", ready );
```

На мой взгляд, это типичный пример кода с «душком», но по какой-то непостижимой причине он нравится некоторым разработчикам.

Asynquence предлагает инвертированную разновидность последовательности, которую я называю *итерируемой последовательностью*. Она выводит функциональность управления на внешний уровень (что может быть весьма полезно в таких примерах, как `domready`):

```
// примечание: `domready` - итератор, который
// управляет последовательностью
var domready = ASQ.iterable();

// ..

domready.val( function(){
    // модель DOM готова
} );
// ..
document.addEventListener( "DOMContentLoaded", domready.next );
```

Возможности итерируемых последовательностей не ограничиваются приведенным примером. Мы вернемся к ним в приложении Б.

Выполнение генераторов

В главе 4 была написана функция `run(..)`, которая может выполнять генераторы до завершения, прослушивая выдаваемые через `yield` обещания и используя их для асинхронного возобновления работы генератора. В *asynquence* существует такая встроенная функция с именем `runner(..)`.

Начнем с создания вспомогательных функций для наглядности:

```
function doublePr(x) {
    return new Promise( function(resolve,reject){
        setTimeout( function(){
            resolve( x * 2 );
        }, 100 );
    } );
}

function doubleSeq(x) {
    return ASQ( function(done){
        setTimeout( function(){
            done( x * 2)
        }, 100 );
    } );
}
```

Теперь `runner(..)` можно использовать как шаг в середине последовательности:

```
ASQ( 10, 11 )
.runner( function*(token){
    var x = token.messages[0] + token.messages[1];

    // выдать реальное обещание
    x = yield doublePr( x );

    // выдать последовательность
    x = yield doubleSeq( x );

    return x;
} )
```

```
.val( function(msg){
    console.log( msg );         // 84
} );
```

Обертки для генераторов

Вы также можете создать автономный генератор, то есть нормальную функцию, которая выполняет заданный вами генератор и возвращает последовательность для его завершения, с вызовом `ASQ.wrap(..)`:

```
var foo = ASQ.wrap( function*(token){
    var x = token.messages[0] + token.messages[1];

    // выдать реальное обещание
    x = yield doublePr( x );

    // выдать последовательность
    x = yield doubleSeq( x );

    return x;
}, { gen: true } );

// ..

foo( 8, 9 )
.val( function(msg){
    console.log( msg );         // 68
} );
```

Функция `runner(..)` открывает еще много интересных возможностей, но мы вернемся к этой теме в приложении Б.

Итоги

asynquence предоставляет простую абстракцию — последовательность как серия (асинхронных) шагов — поверх обещаний. Последовательности значительно упрощают работу с различ-

ными асинхронными паттернами без ущерба для функциональности.

В базовом API *asynquence* и плагинах *contrib* присутствуют и другие полезные возможности, не продемонстрированные в приложении, но их изучение остается читателю для самостоятельной работы.

Мы рассмотрели основную суть *asynquence*. Главный вывод заключается в том, что последовательность состоит из шагов, каждый из которых может относиться к любому из десятков разновидностей обещаний, или может запускать генераторы, или... Выбор за вами; вы можете свободно объединять любые конструкции управления асинхронной программной логикой, подходящие для ваших задач. Вам больше не нужно переключаться между разными библиотеками для использования различных асинхронных паттернов.

Если эти примеры *asynquence* показались вам разумными, значит, вы понемногу осваиваете библиотеку. А впрочем, изучить ее не так уж сложно!

Если вы все еще путаетесь в том, как она работает (или почему!), вам стоит выделить немного времени на изучение предыдущих примеров и эксперименты с *asynquence*, прежде чем переходить к приложению Б, в котором рассматриваются более сложные и мощные асинхронные паттерны.

Приложение Б
Расширенные асинхронные паттерны

В приложении А представлена библиотека *asynquence* для управления асинхронной программной логикой, ориентированная на работу с последовательностями и построенная в основном на базе обещаний и генераторов.

А сейчас мы исследуем другие расширенные асинхронные паттерны, построенные на основе существующего понимания асинхронности и функциональности, и посмотрим, как *asynquence* позволяет легко комбинировать эти сложные асинхронные методы в программах без использования многочисленных специальных библиотек.

Итерируемые последовательности

Итерируемые последовательности *asynquence* уже были представлены в приложении А, но сейчас их стоит рассмотреть более подробно.

Небольшое напоминание:

```
var domready = ASQ.iterable();

// ..
```

```
domready.val( function(){
    // модель DOM готова
} );

// ..

document.addEventListener( "DOMContentLoaded", domready.next );
```

А теперь определим последовательность из нескольких шагов в виде итерируемой последовательности:

```
var steps = ASQ.iterable();

steps
.then( function STEP1(x){
    return x * 2;
} )
.steps( function STEP2(x){
    return x + 3;
} )
.steps( function STEP3(x){
    return x * 4;
} );

steps.next( 8 ).value;  // 16
steps.next( 16 ).value; // 19
steps.next( 19 ).value; // 76
steps.next().done;      // true
```

Как видите, итерируемая последовательность представляет собой итератор, соответствующий стандартам (см. главу 4). Таким образом, для ее перебора может использоваться цикл ES6 `for..of`, а также генератор (или любой другой итерируемый объект):

```
var steps = ASQ.iterable();

steps
.then( function STEP1(){ return 2; } )
.then( function STEP2(){ return 4; } )
.then( function STEP3(){ return 6; } )
.then( function STEP4(){ return 8; } )
```

```
.then( function STEP5(){ return 10; } );

for (var v of steps) {
    console.log( v );
}
// 2 4 6 8 10
```

Кроме примера со срабатыванием событий, приведенного в примере A, у итерируемых последовательностей есть и другие интересные особенности: по сути, они могут рассматриваться как замена для генераторов или цепочек обещаний, но при этом обладают большей гибкостью.

Рассмотрим пример с несколькими запросами Ajax (этот сценарий уже был продемонстрирован в главах 3 и 4 как цепочка обещаний и как генератор соответственно), выраженный в виде итерируемой последовательности:

```
// ajax с последовательностями
var request = ASQ.wrap( ajax );

ASQ( "http://some.url.1" )
.runner(
    ASQ.iterable()

    .then( function STEP1(token){
        var url = token.messages[0];
        return request( url );
    } )

    .then( function STEP2(resp){
        return ASQ().gate(
            request( "http://some.url.2/?v=" + resp ),
            request( "http://some.url.3/?v=" + resp )
        );
    } )

    .then( function STEP3(r1,r2){ return r1 + r2; } )
)
.val( function(msg){
    console.log( msg );
} );
```

Итерируемая последовательность выражает последовательную серию (синхронных или асинхронных) шагов, которые очень похожи на цепочку обещаний, другими словами, они выглядят намного приятнее простых вложенных обратных вызовов, но далеко не так приятно, как последовательный синтаксис генераторов на базе `yield`. Но итерируемая последовательность передается методу `ASQ#runner(..)`, который выполняет ее до завершения, как если бы это был генератор. Тот факт, что итерируемая последовательность ведет себя практически так же, как генератор, примечателен по нескольким причинам.

Прежде всего, итерируемые последовательности являются своего рода эквивалентом некоторого подмножества генераторов ES6; это означает, что вы либо пишете их напрямую (чтобы они могли выполняться где угодно), либо пишете генераторы ES6 и транспилируете/преобразуете их в итерируемые последовательности (или цепочки обещаний, если уж на то пошло).

Восприятие асинхронно выполняемого до завершения генератора как простого синтаксического удобства для цепочек обещаний — это важный аспект понимания изоморфных отношений между ними.

Прежде чем двигаться дальше, следует заметить, что предыдущий фрагмент можно было бы выразить в *asynquence* в следующем виде:

```
ASQ( "http://some.url.1" )
.seq( /*STEP 1*/ request )
.seq( function STEP2(resp){
    return ASQ().gate(
        request( "http://some.url.2/?v=" + resp ),
        request( "http://some.url.3/?v=" + resp )
    );
} )
.val( function STEP3(r1,r2){ return r1 + r2; } )
.val( function(msg){
    console.log( msg );
} );
```

Более того, шаг 2 можно было бы выразить в следующем виде:

```
.gate(
    function STEP2a(done,resp) {
        request( "http://some.url.2/?v=" + resp )
        .pipe( done );
    },
    function STEP2b(done,resp) {
        request( "http://some.url.3/?v=" + resp )
        .pipe( done );
    }
)
```

Итак, зачем нужно брать на себя труд и выражать программную логику в виде итерируемой последовательности в шаге `ASQ#runner(..)`, когда вроде бы существует более простая/деструктурированная цепочка *asynquence*, которая хорошо справляется с работой?

Потому что у формы итерируемой последовательности имеется один важный трюк, который предоставляет вам новые возможности. Читайте дальше.

Расширение итерируемых последовательностей

Генераторы, нормальные асинхронные последовательности и цепочки обещаний *обрабатываются немедленно* — любая программная логика, выраженная в исходной формулировке, фиксируется и четко выполняется.

Однако для итерируемых последовательностей используется *отложенная обработка*; это означает, что во время выполнения итерируемой последовательности можно расширить последовательность новыми шагами, если потребуется.

Возможно только присоединение к концу итерируемой последовательности, но не вставка в середину.

Начнем с более простого (синхронного) примера использования этой возможности:

```
function double(x) {
    x *= 2;

    // продолжать расширение?
    if (x < 500) {
        isq.then( double );
    }

    return x;
}

// создание итерируемой последовательности из одного шага
var isq = ASQ.iterable().then( double );

for (var v = 10, ret;
    (ret = isq.next( v )) && !ret.done;
) {
    v = ret.value;
    console.log( v );
}
```

Итерируемая последовательность начинается только с одного определенного шага (`isq.then(double)`), но последовательность продолжает расширяться при некоторых условиях (`x < 500`). С технической точки зрения последовательности *asynquence* и цепочки обещаний *могут* делать нечто похожее, но вскоре вы увидите, почему их возможностей недостаточно.

Хотя этот пример достаточно тривиален и его с таким же успехом можно было выразить циклом `while` в генераторе, мы рассмотрим более сложные случаи.

Например, можно проанализировать ответ от запроса Ajax, и если он показывает, что нужны дополнительные данные, в итерируемую последовательность условно вставляются дополнительные шаги для создания дополнительных запросов. Также возможно условно добавить шаг форматирования значения в конец обработки Ajax.

Пример:

```
var steps = ASQ.iterable()

.then( function STEP1(token){
    var url = token.messages[0].url;

    // дополнительный шаг форматирования был определен?
    if (token.messages[0].format) {
        steps.then( token.messages[0].format );
    }

    return request( url );
} )

.then( function STEP2(resp){
    // добавить еще один запрос Ajax в последовательность?
    if (/x1/.test( resp )) {
        steps.then( function STEP5(text){
            return request(
                "http://some.url.4/?v=" + text
            );
        } );
    }

    return ASQ().gate(
        request( "http://some.url.2/?v=" + resp ),
        request( "http://some.url.3/?v=" + resp )
    );
} )

.then( function STEP3(r1,r2){ return r1 + r2; } );
```

В листинге видны два разных места, в которых мы условно расширяем `steps` вызовом `steps.then(..)`. И чтобы выполнить итерируемую последовательность `steps`, мы просто подключаем ее к программной логике основной программы с асинхронной последовательностью (здесь она называется `main`) при помощи `ASQ#runner(..)`:

```
var main = ASQ( {
    url: "http://some.url.1",
    format: function STEP4(text){
        return text.toUpperCase();
```

```
    }
} )
.runner( steps )
.val( function(msg){
    console.log( msg );
} );
```

Можно ли выразить гибкость (условное поведение) итерируемой последовательности `steps` при помощи генератора? Отчасти да, но логику придется переупорядочить довольно неуклюжим способом:

```
function *steps(token) {
    // ШАГ 1
    var resp = yield request( token.messages[0].url );

    // ШАГ 2
    var rvals = yield ASQ().gate(
        request( "http://some.url.2/?v=" + resp ),
        request( "http://some.url.3/?v=" + resp )
    );
    // ШАГ 3
    var text = rvals[0] + rvals[1];

    // ШАГ 4
    // дополнительный шаг форматирования был определен?
    if (token.messages[0].format) {
        text = yield token.messages[0].format( text );
    }

    // ШАГ 5
    // добавить еще один запрос Ajax в последовательность?
    if (/foobar/.test( resp )) {
        text = yield request(
            "http://some.url.4/?v=" + text
        );
    }

    return text;
}

// примечание: `*steps()` может выполняться той же
последовательностью
// `ASQ`, как и `steps` в приведенном примере
```

Оставляя в стороне уже описанные преимущества последовательного, синхронно выглядящего синтаксиса генераторов (см. главу 4), логику `steps` необходимо переупорядочить в форме генератора `*steps()`, чтобы имитировать динамическую природу расширяемой итерируемой последовательности `steps`.

А как насчет выражения функциональности с обещаниями или последовательностями? *Можно* поступить примерно так:

```
var steps = something( .. )
.then( .. )
.then( function(..){
    // ..

    // расширяем цепочку, верно?
    steps = steps.then( .. );

    // ..
})
.then( .. );
```

Проблема не очевидна, но вы должны понять ее суть. Подумайте, что происходит при подключении цепочки обещаний `steps` к основной программной логике — на этот раз выраженной с использованием обещаний вместо *asynquence*:

```
var main = Promise.resolve( {
    url: "http://some.url.1",
    format: function STEP4(text){
        return text.toUpperCase();
    }
} )
.then( function(..){
    return steps;           // подсказка!
} )
.val( function(msg){
    console.log( msg );
} );
```

Ну что, заметили? Смотрите внимательнее!

Существует состояние гонки в порядке последовательности `steps`. Когда вы возвращаете `steps`, в этот момент `steps` *может* быть исходно определенной цепочкой обещаний, а может указывать на цепочку обещаний, расширенную вызовом `steps = steps.then(..)`, — это зависит от того, в каком порядке происходят операции.

Возможны два результата:

- Если `steps` все еще остается исходной цепочкой обещаний, после «расширения» вызовом `steps = steps.then(..)` расширенное обещание в конце цепочки *не учитывается* логикой `main`, так как она уже «захватила» цепочку `steps`. Это нежелательное следствие *немедленной обработки*.
- Если `steps` уже является расширенной цепочкой обещаний, все работает так, как и ожидалось, — `main` «захватывает» расширенное обещание.

Если не считать того очевидного факта, что состояние гонки в принципе недопустимо, проблема возникает в первом случае, в котором проявляется *немедленная обработка* цепочки обещаний. С другой стороны, итерируемая последовательность легко расширяется без подобных проблем, потому что итерируемые последовательности используют *отложенную обработку*.

Чем более динамичным должно быть управление программной логикой, тем нагляднее проявляются достоинства итерируемых последовательностей.

За дополнительной информацией и примерами итерируемых последовательностей обращайтесь на сайт asynquence (https://github.com/getify/asynquence/blob/master/README.md).

Реакция на события

Из главы 3 (и не только!) должно быть ясно, что обещания являются исключительно мощным инструментом в вашем инструмен-

тарии асинхронного программирования. Однако в их функциональности есть один очевидный пробел: они не позволяют обрабатывать потоки событий, так как каждое обещание может быть разрешено только один раз. И откровенно говоря, такая же слабость присуща простым последовательностям *asynquence*.

Рассмотрим сценарий, в котором при срабатывании некоего события каждый раз должна отрабатывать серия шагов. Одиночное обещание (или последовательность) не может представлять все возникновения этого события. Следовательно, вам придется создавать новую цепочку обещаний (или последовательность) для каждого возникновения события, например:

```
listener.on( "foobar", function(data){

    // создать новую цепочку обещаний для обработки событий
    new Promise( function(resolve,reject){
        // ..
    } )
    .then( .. )
    .then( .. );

} );
```

В этом решении присутствует необходимая базовая функциональность, но такой способ выражения предполагаемой логики оставляет желать лучшего. В этой парадигме объединяются два разных аспекта функциональности: прослушивание событий и реагирование на событие. Принцип разделения обязанностей подталкивает нас к тому, чтобы разделить эти аспекты.

Наблюдательный читатель заметит, что эта проблема отчасти симметрична проблемам, описанным в главе 2 для обратных вызовов; она тоже является разновидностью инверсии управления.

Представьте, что в этой парадигме происходит отмена инверсии:

```
var observable = listener.on( "foobar" );

// потом
observable
```

```
.then( .. )
.then( .. );

// в другом месте
observable
.then( .. )
.then( .. );
```

Значение `observable` не является обещанием в полном смысле, но за ним можно *наблюдать* практически так же, как вы наблюдаете за обещанием, так что они тесно связаны. Наблюдать можно много раз, и оно будет отправлять уведомления каждый раз при возникновении события (`"foobar"`).

Продемонстрированный паттерн значительно упрощает концепции и причины, лежащие в основе реактивного программирования (RP, Reactive Programming), реализованного/интерпретированного в нескольких замечательных проектах и языках. Одной из разновидностей RP является функциональное реактивное программирование (FRP) — применение концепций функционального программирования (неизменяемость, целостность ссылок и т. д.) к потокам данных. Под «реактивностью» имеется в виду распространение этой функциональности во времени в ответ на события. Заинтересованному читателю стоит изучить раздел «Reactive Observables» потрясающей библиотеки Reactive Extensions (RxJS для JavaScript, http://reactive-extensions.github.io/RxJS/) компании Microsoft; она намного сложнее и мощнее того, что я продемонстрировал. Кроме того, у Андре Штальца (Andre Staltz) имеется отличная статья (https://gist.github.com/staltz/868e7e9bc2a7b8c1f754), которая представляет RP в практическом аспекте на конкретных примерах.

Наблюдаемые объекты в ES7

На момент написания книги существовало раннее предложение для ES7 нового типа данных наблюдаемых объектов (https://github.com/jhusain/asyncgenerator), который по духу близок к описанной теме, но определенно обладает гораздо большими возможностями.

Основная идея наблюдаемых объектов такого рода заключается в том, что для «подписки» на события от потока вы передаете генератор, хотя на самом деле заинтересованной стороной здесь является итератор, метод `next(..)` которого будет вызываться для каждого события.

Это можно представить себе примерно так:

```
// `someEventStream` - поток событий (например,is a stream of
events, like от щелчков кнопкой мыши и т. д.

var observer = new Observer( someEventStream, function*(){
    while (var evt = yield) {
        console.log( evt );
    }
} );
```

Передаваемый вами генератор приостановит цикл `while` в ожидании следующего события. У итератора, связанного с экземпляром генератора, метод `next(..)` будет вызываться каждый раз, когда у `someEventStream` публикуется новое событие, а данные события возобновят работу генератора/итератора с данными `evt`.

В функции подписки на события важен итератор, а не генератор. Таким образом, концептуально вы можете передать практически любой итерируемый объект, включая итерируемые последовательности `ASQ.iterable()`.

Интересно, что также были предложены адаптеры, которые должны упростить конструирование наблюдаемых объектов от некоторых типов потоков, таких как `fromEvent(..)` для событий DOM. Если взглянуть на предполагаемую реализацию `fromEvent(..)` из предыдущего предложения ES7, она очень похожа на `ASQ.react(..)`, как будет показано в следующем разделе.

Конечно, все это только предложения, так что конечный результат вполне может отличаться по внешнему виду и поведению от того, что показано здесь. Тем не менее очень интересно проследить за ранним согласованием концепций между предложениями для разных библиотек и языков!

Реактивные последовательности

Руководствуясь этим кратким описанием наблюдаемых объектов (и F/RP), я продемонстрирую адаптацию небольшого подмножества «реактивных наблюдаемых объектов», которую я назову «реактивной последовательностью».

Начнем с создания наблюдаемого объекта, используя функцию плагина *asynquence* с именем `react(..)`:

```
var observable = ASQ.react( function setup(next){
    listener.on( "foobar", next );
} );
```

Теперь посмотрим, как определить последовательность, которая «реагирует» — в F/RP это обычно называется «подпиской» (subscribing) — на объект `observable`:

```
observable
.seq( .. )
.then( .. )
.val( .. );
```

Таким образом, вы просто определяете последовательность, создавая цепочку на базе наблюдаемого объекта. Несложно, верно?

В F/RP поток событий обычно проходит через группу функциональных преобразований, таких как `scan(..)`, `map(..)`, `reduce(..)` и т. д. С реактивными последовательностями каждое событие проходит через новый экземпляр последовательности. Рассмотрим более конкретный пример:

```
ASQ.react( function setup(next){
    document.getElementById( "mybtn" )
    .addEventListener( "click", next, false );
} )
.seq( function(evt){
    var btnID = evt.target.id;
    return request(
```

```
        "http://some.url.1/?id=" + btnID
    );
} )
.val( function(text){
    console.log( text );
} );
```

«Реактивная» часть в названии реактивной последовательности происходит от назначения одного или нескольких обработчиков событий для активизации триггера события (вызова `next(..)`).

«Последовательность» в названии относится к последовательностям, которые мы уже рассмотрели: каждый шаг может быть любым асинхронным механизмом, от обратного вызова продолжения или обещания до генератора.

После того как вы создали реактивную последовательность, она будет инициировать экземпляры последовательности до тех пор, пока события будут срабатывать. Если вы захотите остановить реактивную последовательность, вызовите `stop()`.

Если реактивная последовательность останавливается вызовом `stop()`, скорее всего, вы также хотите отменить регистрацию обработчика(-ов) события; для этой цели можно зарегистрировать `teardown`-обработчик:

```
var sq = ASQ.react( function setup(next,registerTeardown){
    var btn = document.getElementById( "mybtn" );
    btn.addEventListener( "click", next, false );
    // будет вызываться при вызове `sq.stop()`
    registerTeardown( function(){
        btn.removeEventListener( "click", next, false );
    } );
} )
.seq( .. )
.then( .. )
.val( .. );

// позднее
sq.stop();
```

Ссылка this внутри обработчика setup(..) эквивалентна реактивной последовательности sq, так что ссылка this может использоваться для расширения определения реактивной последовательности, вызова таких методов, как stop(), и т. д.

Следующий пример из мира Node.js использует реактивные последовательности для обработки входящих запросов HTTP:

```
var server = http.createServer();
server.listen(8000);

// реактивный наблюдатель
var request = ASQ.react( function setup(next,registerTeardown){
    server.addListener( "request", next );
    server.addListener( "close", this.stop );

    registerTeardown( function(){
        server.removeListener( "request", next );
        server.removeListener( "close", request.stop );
    } );
});

// реакция на запросы
request
.seq( pullFromDatabase )
.val( function(data,res){
    res.end( data );
} );

// завершение node
process.on( "SIGINT", request.stop );
```

Триггер next(..) также может легко адаптироваться к потокам с использованием методов onStream(..) и unStream(..):

```
ASQ.react( function setup(next){
    var fstream = fs.createReadStream( "/some/file" );
```

```
    // передать событие "data" потока `next(..)`
    next.onStream( fstream );

    // прослушивать конец потока
    fstream.on( "end", function(){
        next.unStream( fstream );
    } );
} )
.seq( .. )
.then( .. )
.val( .. );
```

Также можно использовать комбинации последовательностей для объединения нескольких потоков реактивных последовательностей:

```
var sq1 = ASQ.react( .. ).seq( .. ).then( .. );
var sq2 = ASQ.react( .. ).seq( .. ).then( .. );

var sq3 = ASQ.react(..)
.gate(
    sq1,
    sq2
)
.then( .. );
```

Основной вывод заключается в том, что `ASQ.react(..)` представляет собой облегченную адаптацию концепций F/RP, которая позволяет связать поток событий с последовательностью, — отсюда термин «реактивная последовательность». Реактивные последовательности обычно обладают достаточной функциональностью для основных реактивных применений.

Пример использования ASQ.react(..) для управления состоянием пользовательского интерфейса доступен по адресу http://jsbin.com/rozipaki/6/edit?js,output, а другой пример обработки потоков запросов/ответов HTTP с `ASQ.react(..)` — по адресу https://gist.github.com/getify/bba5ec0de9d6047b720e.

Генераторные сопрограммы (Generator Coroutine)

Хочется надеяться, что глава 4 помогла вам освоить генераторы ES6. Сейчас мы вернемся к обсуждению параллельного выполнения генераторов и разовьем его.

Тогда мы рассматривали функцию `runAll(..)`, которая может получить два и более генератора и выполнить их параллельно, позволяя им кооперативно передавать управление от одного к другому, с необязательной передачей сообщений.

Кроме возможности выполнения одного генератора до завершения, метод `ASQ#runner(..)`, рассмотренный в приложении А, представляет собой сходную концепцию `runAll(..)`, которая может параллельно выполнять несколько генераторов до завершения.

А теперь посмотрим, как реализовать параллельный сценарий Ajax из главы 4:

```
ASQ(
    "http://some.url.2"
)
.runner(
    function*(token){
        // передача управления
        yield token;

        var url1 = token.messages[0]; // "http://some.url.1"

        // стереть сообщения и начать заново
        token.messages = [];

        var p1 = request( url1 );

        // передача управления
        yield token;

        token.messages.push( yield p1 );
    },
    function*(token){
```

```
        var url2 = token.messages[0]; // "http://some.url.2"

        // запись сообщения и передача управления
        token.messages[0] = "http://some.url.1";
        yield token;

        var p2 = request( url2 );

        // передача управления
        yield token;

        token.messages.push( yield p2 );

        // передать результаты следующему шагу последовательности
        return token.messages;
    }
)
.val( function(res){
    // `res[0]` поступает от "http://some.url.1"
    // `res[1]` поступает от "http://some.url.2"
} );
```

Основные различия между `ASQ#runner(..)` и `runAll(..)`:

- Каждый генератор (сопрограмма) получает аргумент, которому мы присвоили имя `token` — специальное значение, которое должно выдаваться `yield`, когда вы хотите явно передать управление следующей сопрограмме.
- `token.messages` — массив, содержащий все сообщения, переданные от предыдущего шага последовательности. Он также является структурой данных, которая может использоваться для обмена сообщениями между сопрограммами.
- Передача через `yield` значения обещания (или последовательности) не приводит к передаче управления, а приостанавливает обработку сопрограммы до момента готовности этого значения.
- Последнее возвращенное через `return` или выданное через `yield` значение от обработки сопрограммы будет передано следующему шагу последовательности.

Вы также можете легко наложить вспомогательные функции поверх базовой функциональности `ASQ#runner(..)` для различных целей.

Конечные автоматы

Конечные автоматы (state machines) хорошо знакомы многим программистам. С помощью простой косметической функции можно создать модель конечного автомата, которая легко выражается в программном коде.

Попробуем представить такую функцию, она будет называться `state(..)`. Функция получает два аргумента: значение состояния и генератор, который обрабатывает это состояние. `state(..)` выполняет всю черную работу по созданию и возвращению генератора-адаптера для передачи `ASQ#runner(..)`.

Пример:

```
function state(val,handler) {
    // создать обработчик сопрограммы для этого состояния
    return function*(token) {
        // обработчик перехода состояния
        function transition(to) {
            token.messages[0] = to;
        }

        // задать исходное состояние (если оно еще не задано)
        if (token.messages.length < 1) {
            token.messages[0] = val;
        }

        // продолжать до достижения финального состояния (false)
        while (token.messages[0] !== false) {
            // текущее состояние соответствует
            // этому обработчику?
            if (token.messages[0] === val) {
                // делегировать обработчику состояния
                yield *handler( transition );
            }
```

```
            // передать управление другому обработчику
            // состояния?
            if (token.messages[0] !== false) {
                yield token;
            }
        }
    };
}
```

Присмотревшись внимательнее, вы увидите, что `state(..)` возвращает генератор, который получает маркер `token`, а затем создает цикл `while`, выполняемый до достижения конечным автоматом финального состояния (для которого мы произвольно выбрали значение `false`); именно такой генератор должен передаваться `ASQ#runner(..)`!

Мы также произвольно зарезервировали элемент `token.messages[0]` как позицию для отслеживания текущего состояния конечного автомата; это означает, что мы даже можем инициализировать исходное состояние значением, переданным от текущего шага последовательности.

Как использовать вспомогательную функцию `state(..)` в сочетании с `ASQ#runner(..)`?

```
var prevState;

ASQ(
    /* не обязательно: исходное состояние */
    2
)
// запустить конечный автомат
// переходы: 2 -> 3 -> 1 -> 3 -> false
.runner(
    // обработчик состояния `1`
    state( 1, function *stateOne(transition){
        console.log( "in state 1" );
        prevState = 1;
        yield transition( 3 );  // перейти в состояние `3`
    } ),

    // обработчик состояния `2`
```

```
    state( 2, function *stateTwo(transition){
        console.log( "in state 2" );

        prevState = 2;
        yield transition( 3 );  // перейти в состояние `3`
    } ),

    // обработчик состояния `3`
    state( 3, function *stateThree(transition){
        console.log( "in state 3" );

        if (prevState === 2) {
            prevState = 3;
            yield transition( 1 ); // перейти в состояние `1`
        }
        // готово!
        else {
            yield "That's all folks!";

            prevState = 3;
            yield transition( false ); // terminal state
        }
    } )
)
// работа конечного автомата завершена, идти дальше
.val( function(msg){
    console.log( msg ); // That's all folks!
} );
```

Важно заметить, что сами генераторы `*stateOne(..)`, `*stateTwo(..)` и `*stateThree(..)` вызываются заново при каждом входе в это состояние и завершаются при вызове `transition(..)` с другим значением. Хотя это здесь не показано, конечно, эти обработчики состояний могут асинхронно приостанавливаться посредством `yield`-выдачи обещаний/последовательности/преобразователей.

Скрытые генераторы, производимые вспомогательной функцией `state(..)` и фактически переданные `ASQ#runner(..)`, продолжают параллельно работать на протяжении всего времени работы конечного автомата. Каждый из них в кооперативном режиме обеспечивает `yield`-передачу управления следующему, и т. д.

Пример ping pong (http://jsbin.com/qutabu/1/edit?js,output) более полно демонстрирует кооперативное параллельное выполнение с генераторами под управлением `ASQ#runner(..)`.

Взаимодействующие последовательные процессы

Модель *взаимодействующих последовательных процессов* (CSP, Communicating Sequential Processes) впервые была описана Чарльзом Э. Хоаром (C. A. R. Hoare) в академической статье 1978 года и позднее рассмотрена в одноименной книге 1985 года. CSP описывает формальный механизм взаимодействия параллельных «процессов» во время выполнения. Возможно, вы помните, что мы упоминали параллельно выполняемые «процессы» в главе 1, так что наше исследование CSP здесь будет базироваться на этом понимании.

Как и большинство выдающихся концепций в компьютерной теории, CSP в значительной мере происходит из академических исследований, выраженных в алгебре процессов. Тем не менее я подозреваю, что теоремы символической алгебры вряд ли принесут практическую пользу читателю, так что нам придется поискать другой подход к CSP.

За формальными описаниями и доказательствами я предлагаю читателю обращаться к работе Хоара и многим другим отличным работам, написанным с того времени. А здесь мы попробуем кратко объяснить идею CSP с неакадемических — и хочется верить, интуитивно понятных — позиций.

Передача сообщений

Основной принцип CSP заключается в том, что все взаимодействия/обмен данными между независимыми (в остальном) про-

цессами должны осуществляться посредством формальной передачи сообщений. Возможно, вопреки вашим ожиданиям передача сообщений в CSP описывается как синхронное действие, при котором процесс-отправитель и процесс-получатель должны быть готовы к передаче сообщения.

Как синхронная передача сообщений может быть связана с асинхронным программированием на JavaScript?

В основе конкретности отношений лежит использование генераторов ES6 для создания действий, которые выглядят синхронно, но во внутренней реализации могут быть синхронными или (более вероятно) асинхронными. Другими словами, два и более параллельно выполняемых генератора могут на первый взгляд синхронно обмениваться сообщениями без потери фундаментальной асинхронности системы, поскольку код каждого генератора приостанавливается (блокируется), ожидая возобновления асинхронного действия.

Как это все работает?

Представьте генератор («процесс») с именем «A», который хочет отправить сообщение генератору «B». Сначала «A» выдает через `yield` сообщение (что приводит к приостановке «A») для отправки «B». Когда генератор «B» будет готов и получит сообщение, «A» продолжает работу (разблокируется).

Одно из самых популярных выражений теории передачи сообщений CSP происходит из библиотеки ClojureScript *core.async*, а также из языка *go*. В этих подходах к CSP воплощается описанная семантика передачи данных через *канал*, открытый между процессами.

Использование термина «канал» отчасти объясняется тем, что в некоторых режимах в буфер канала может передаваться более одного значения; это чем-то напоминает наши представления о потоках. Здесь этот механизм подробно не рассматривается, но он может быть очень мощным средством управления потоками данных.

В простейшем представлении CSP канал, созданный между точками «A» и «B», должен содержать метод `take(..)` для блокировки при получении значения и метод `put(..)` для блокировки при отправке значения.

Это может выглядеть примерно так:

```
var ch = channel();

function *foo() {
    var msg = yield take( ch );

    console.log( msg );
}

function *bar() {
    yield put( ch, "Hello World" );

    console.log( "message sent" );
}

run( foo );
run( bar );
// Hello World
// "message sent"
```

Сравните эту структурированную, синхронную (вернее, синхронно выглядящую) передачу сообщений с неформальным и неструктурированным обменом сообщений, которую предоставляет `ASQ#runner(..)`, с массивом `token.messages` и кооперативным использованием `yield`. По сути, `yield put(..)` представляет собой одну операцию, которая одновременно отправляет значение и приостанавливает выполнение для передачи управления, тогда как в предыдущих примерах это делалось в разных шагах.

Кроме того, CSP наглядно показывает, что вы не передаете управление явно, но вместо этого проектируете свои параллельные функции для блокировки в ожидании либо значения, полученного из канала, либо попытки отправить сообщение по каналу. Именно блокировка по получению или отправке сообщений является механизмом координации поведения между сопрограммами.

Небольшое предупреждение: это очень мощный паттерн, но чтобы привыкнуть к нему, потребуется некоторое время. Немного потренируйтесь, чтобы привыкнуть к этому новому подходу к координации параллелизма.

Для этой разновидности CSP, реализованной в JavaScript, существует несколько хороших библиотек, и прежде всего библиотека *js-csp* (https://github.com/ubolonton/js-csp), которую запустил Джеймс Лонг (James Long) (https://github.com/jlongster/js-csp) и о которой он много писал (http://jlongster.com/Taming-the-Asynchronous-Beast-with-CSP-in-JavaScript). Кроме того, статьи Дэвида Нолена (David Nolen) (http://twitter.com/swannodette) содержат исключительно полезную информацию об адаптации CSP ClojureScript в *go*-стиле *core.async* для генераторов JS (http://swannodette.github.io/2013/08/24/es6-generators-and-csp/).

Эмуляция CSP в asynquence

Поскольку ранее мы здесь обсуждали асинхронные паттерны в контексте моей библиотеки *asynquence*, возможно, вам будет интересно узнать, что поверх обработки генераторов `ASQ#runner(..)` можно довольно легко добавить прослойку эмуляции, которая обеспечивает практически идеальное портирование CSP API и реализацию поведения. Прослойка эмуляции поставляется как дополнительная часть пакета *asynquence-contrib* вместе с *asynquence*.

По аналогии со вспомогательной функцией `state(..)`, `ASQ.csp.go(..)` получает генератор — в терминологии *go/core.async* он называется *go-процедурой* (goroutine) — и адаптирует его для использования с `ASQ#runner(..)`, возвращая новый генератор.

Вместо передачи `token` ваша go-процедура получает изначально созданный канал (`ch`), который совместно используется всеми go-процедурами для этого запуска программы. Вы можете создать дополнительные каналы (что часто бывает весьма полезно!) вызовом `ASQ.csp.chan(..)`.

В CSP вся асинхронность моделируется в контексте блокирования по сообщениям канала вместо блокирования в ожидании завершения обещания/последовательности/преобразователя.

Итак, вместо выдачи через `yield` обещания, возвращенного от `request(..)`, функция `request(…)` должна вернуть канал, из которого запрашивается значение вызовом `take(..)`. Иначе говоря, канал с одним значением в этом контексте/варианте использования приблизительно эквивалентен обещанию/последовательности.

Для начала создадим версию `request(..)` с поддержкой каналов:

```
function request(url) {
    var ch = ASQ.csp.channel();
    ajax( url ).then( function(content){
        // `putAsync(..)` - версия `put(..)`, которая может
        // использоваться за пределами генератора. Она
        // возвращает обещание для завершения операции. Здесь
        // это обещание не используется, но мы могли бы
        // использовать его, если бы потребовалось получать
        // уведомления о получении значения вызовом `take(..)`.
        ASQ.csp.putAsync( ch, content );
    } );
    return ch;
}
```

Вспомните: в главе 3 мы ввели термин «фабрика обещаний» для функции, производящей обещания; в главе 4 — термин «фабрика преобразователей» для функции, производящей преобразователи; и наконец, в приложении А — термин «фабрика последовательностей» для функции, производящей последовательности.

Естественно, аналогичный термин стоит ввести и для функции, производящей каналы. Неудивительно, что мы назовем ее «фабрикой каналов». В качестве упражнения для самостоятельной работы читатель может определить функцию `channelify(..)` по аналогии с `Promise.wrap(..)`/`promisify(..)` (глава 3), `thunkify(..)` (глава 4) и `ASQ.wrap(..)` (приложение А).

А теперь рассмотрим пример параллелизма Ajax с использованием CSP в стиле *asynquence*:

```
ASQ()
.runner(
    ASQ.csp.go( function*(ch){
        yield ASQ.csp.put( ch, "http://some.url.2" );

        var url1 = yield ASQ.csp.take( ch );
        // "http://some.url.1"

        var res1 = yield ASQ.csp.take( request( url1 ) );

        yield ASQ.csp.put( ch, res1 );
    } ),
    ASQ.csp.go( function*(ch){
        var url2 = yield ASQ.csp.take( ch );
        // "http://some.url.2"
        yield ASQ.csp.put( ch, "http://some.url.1" );

        var res2 = yield ASQ.csp.take( request( url2 ) );
        var res1 = yield ASQ.csp.take( ch );

        // передать результаты следующему шагу
        // последовательности
        ch.buffer_size = 2;
        ASQ.csp.put( ch, res1 );
        ASQ.csp.put( ch, res2 );
    } )
)
.val( function(res1,res2){
    // `res1` comes from "http://some.url.1"
    // `res2` comes from "http://some.url.2"
} );
```

Передача сообщений, которая обеспечивает обмен строками URL между двумя go-процедурами, реализована тривиально. Первая go-процедура выдает запрос Ajax к первому URL-адресу и помещает ответ в канал `ch`. Вторая go-процедура выдает запрос Ajax ко второму URL-адресу, после чего получает первый ответ `res1` из канала `ch`. В этой точке оба ответа, `res1` и `res2`, завершены и готовы.

Если в канале `ch` в конце go-процедуры остаются какие-либо значения, они будут переданы следующему шагу последовательности. Таким образом, чтобы передать сообщения из последней go-процедуры, поместите их в `ch` вызовом `put(..)`. Чтобы избежать блокирования этих завершающих вызовов `put(..)`, мы переводим `ch` в режим буферизации, присваивая свойству `buffer_size` значение `2` (по умолчанию равно `0`).

Другие примеры использования CSP в стиле asynquence доступны на GitHub: https://gist.github.com/getify/e0d04f1f5aa24b1947ae.

Итоги

Обещания и генераторы предоставляют фундаментальные структурные элементы, на базе которых строится более сложная и функциональная асинхронность.

В *asynquence* имеются средства для реализации итерируемых последовательностей, реактивных последовательностей, параллельных сопрограмм и даже go-процедур CSP.

Эти паттерны в сочетании с функциональностью обратных вызовов продолжения и обещаний предоставляют *asynquence* мощную комбинацию разных асинхронных функциональных средств, интегрированных в одну стройную абстракцию управления асинхронной программной логикой — последовательность.

Об авторе

Кайл Симпсон — евангелист Open Web из Остина (штат Техас), большой энтузиаст всего, что касается JavaScript. Автор нескольких книг, преподаватель, спикер и участник/лидер проектов с открытым кодом.

Кайл Симпсон

{Вы не знаете JS}
Асинхронная обработка и оптимизация

Перевел с английского Е. Матвеев

Заведующая редакцией	*Ю. Сергиенко*
Ведущий редактор	*К. Тульцева*
Литературный редактор	*М. Устимова*
Художественный редактор	*В. Мостипан*
Корректоры	*С. Беляева, Н. Викторова*
Верстка	*Л. Егорова*

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: 05.2019. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12.000 —
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 14.05.19. Формат 60x90/16. Бумага офсетная. Усл. п. л. 22,000.
Тираж 1000. Заказ 0000.

Отпечатано в ОАО «Первая Образцовая типография». Филиал «Чеховский Печатный Двор».
142300, Московская область, г. Чехов, ул. Полиграфистов, 1.
Сайт: www.chpk.ru. E-mail: marketing@chpk.ru
Факс: 8(496) 726-54-10, телефон: (495) 988-63-87