

Оглавление

Предисловие	10
Введение	12
Задача	13
О книге	15
Типографские соглашения	15
Использование программного кода примеров	16
От издательства	17
Глава 1. Асинхронность: сейчас и потом	18
Блочное строение программы	19
Асинхронный вывод в консоль	22
Цикл событий	23
Параллельные потоки	26
Выполнение до завершения	30
Параллельное выполнение	33
Отсутствие взаимодействий	36
Взаимодействия	36
Кооперация	42
Задания	45
Упорядочение команд	46
Итоги	50
Глава 2. Обратные вызовы	52
Продолжения	53
Последовательное мышление	55

Работа и планирование	56
Вложенные/сцепленные обратные вызовы	59
Проблемы доверия	65
История о пяти обратных вызовах.	66
Не только в чужом коде.	69
Попытки спасти обратные вызовы.	71
Итоги	76
Глава 3. Обещания	78
Что такое обещание?	79
Будущее значение	80
Событие завершения	86
События обещаний	90
Утиная типизация с методом <code>then()(thenable)</code>	93
Доверие <code>Promise</code>	96
Слишком ранний обратный вызов	97
Слишком поздний обратный вызов.	97
Обратный вызов вообще не вызывается.	100
Слишком малое или слишком большое количество вызовов	101
Отсутствие параметров/переменных среды	102
Поглощение ошибок/исключений.	102
Обещания, заслуживающие доверия?	104
Формирование доверия	108
Сцепление	109
Терминология: разрешение, выполнение и отказ	118
Обработка ошибок	121
Бездна отчаяния	125
Обработка неперехваченных ошибок	126
Бездна успеха	128
Паттерны обещаний	131
<code>Promise.all([..])</code>	131
<code>Promise.race([..])</code>	133
Вариации на тему <code>all([..])</code> и <code>race([..])</code>	137

Параллельно выполняемые итерации	139
Снова о Promise API	140
Конструктор new Promise(..)	141
Promise.resolve(..) и Promise.reject(..)	141
then(..) и catch(..)	142
Promise.all([..]) и Promise.race([..])	143
Ограничения обещаний.	145
Последовательность обработки ошибок	145
Единственное значение	147
Инерция	152
Неотменяемость обещаний	157
Эффективность обещаний	159
Итоги	161
Глава 4. Генераторы.	162
Нарушение принципа выполнения до завершения.	162
Ввод и вывод.	166
Передача сообщений при итерациях	167
Множественные итераторы	171
Генерирование значений	176
Производители и итераторы	176
Итерируемые объекты	180
Итераторы генераторов	182
Асинхронный перебор итераторов.	186
Синхронная обработка ошибок.	190
Генераторы + обещания	192
Выполнение генератора с поддержкой обещаний.	195
Параллелизм обещаний в генераторах.	199
Делегирование	204
Почему делегирование?	207
Делегирование сообщений.	208
Делегирование асинхронности.	213
Делегирование рекурсии	214
Параллельное выполнение генераторов	216

Преобразователи	222
s/promise/thunk/.	227
Генераторы до ES6	231
Ручное преобразование	231
Автоматическая транспиляция	237
Итоги	239
Глава 5. Быстродействие программ	241
Веб-работники	242
Рабочая среда	246
Передача данных.	247
Общие работники	248
Полифилы для веб-работников	250
SIMD.	251
asm.js	253
Как оптимизировать с asm.js	254
Модули asm.js	255
Итоги	258
Глава 6. Хронометраж и настройка	260
Хронометраж	261
Повторение	262
Benchmark.js	264
Все зависит от контекста.	267
Оптимизации движка	268
jsPerf.com	271
Проверка на здравый смысл	272
Написание хороших тестов	276
Микробыстродействие.	277
Различия между движками	282
Общая картина	285
Оптимизация хвостовых вызовов (TCO).	288
Итоги	291

Приложение А. Библиотека `asynquence` 292

Последовательности и архитектура, основанная на абстракциях	293
<code>asynquence</code> API	297
Шаги	297
Ошибки	300
Параллельные шаги	303
Ветвление последовательностей	311
Объединение последовательностей	311
Значение и последовательности ошибки	313
Обещания и обратные вызовы	314
Итерируемые последовательности	316
Выполнение генераторов	318
Обертки для генераторов	319
Итоги	319

Приложение Б. Расширенные асинхронные паттерны. 321

Итерируемые последовательности	321
Расширение итерируемых последовательностей	325
Реакция на события	330
Наблюдаемые объекты в ES7	332
Реактивные последовательности	334
Генераторные сопрограммы (Generator Coroutine)	338
Конечные автоматы	340
Взаимодействующие последовательные процессы	343
Передача сообщений	343
Эмуляция CSP в <code>asynquence</code>	346
Итоги	349

Об авторе 350

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 выигрывает — 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 Ажах, но в контексте гонки между p1 и p2:

```
// `request(..)` - функция Ажах 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.