



目錄

简介	0
前言	1
简介	2
使用	3
Promise.resolve	3.1
Promise.reject	3.2
专栏: Promise只能进行异步操作?	3.3
Promise#then	3.4
Promise#catch	3.5
新promise?	3.6
Promise和数组	3.7
Promise.all	3.8
Promise.race	3.9
then or catch?	3.10
测试	4
高级	5
实现类库	5.1
resolve和Thenable	5.2
reject而非throw	5.3
Deferred和Promise	5.4
race和delay	5.5
Promise.prototype.done	5.6
方法链	5.7
顺序处理	5.8
接口	6
语意	7
参考	8
作者	9
译者	10

JavaScript Promise迷你书（中文版）

原著：azu / 翻译：liubin、kaku、honnkyou Version 1.4.1

Hash tag [#Promise迷你书](#)



可以从这里 [RSS](#) 得到更新信息

[Tweet](#)

本书以Creative Commons Attribution-NonCommercial [许可证](#)发布。另外你也可以从下面的链接下载PDF版本。

[下载本书PDF版本](#) [下载本书源代码](#)

关于本电子书的创作过程有兴趣的读者，可以下载这个文档供参考（日文版）[制作过程](#)。

-   [（日文版）附录下载](#)
- 本书开始编写的原因，如何编写的，以及编写过程是如何运作的。
- 可以在Gumroad上以包括0元在内的任何价格购买本附录内容。
- 如果你想捐赠，也可以通过本附录链接以任意的价格购买本附录的方式进行。

前言

写作初衷

本书的目的是以目前还在制定中的[ECMAScript 6 Promises](#)规范为中心，着重向各位读者介绍JavaScript中对Promise相关技术的支持情况。

通过阅读本书，我们希望各位读者能在下面三个目标上有所收获。

- 学习Promise相关内容，能熟练使用Promise模式并进行测试
- 学习Promise适合什么、不适合什么，知道Promise不是万能的，不能什么都想用Promise来解决
- 以ES6 Promises为基础进行学习，逐渐发展形成自己的风格

像上面所提到的那样，本书主要是以[ES6 Promises](#)，即JavaScript的标准规范为基础的、Promise的相关知识为主要讲解内容。

在Firefox和Chrome这样技术比较超前的浏览器上，不需要安装额外的插件就能使用Promise功能，此外ES6 Promises的规范来源于[Promises/A+](#)社区，它有很多版本的实现。

我们将会从基础API开始介绍可以在浏览器的原生支持或者通过插件支持的Promise功能。也希望各位读者能了解这其中Promise适合干什么，不适合干什么，能根据实际需求选择合适的技术实现方案。

开始阅读之前

本书的阅读对象需要对JavaScript有基本的了解和知识。

- [JavaScript: The Good Parts](#)
- [JavaScript Patterns](#)
- [JavaScript: The Definitive Guide, 6th Edition](#)
- [Perfect JavaScript（日文版）](#)
- [Effective JavaScript（日文版）](#)

如果你读过上面的其中一本的话，就应该非常容易理解本书的内容了。

另外如果你有使用JavaScript编写Web应用程序的经验，或者使用Node.js编写过命令行、服务器端程序的话，那么你可能会对本文中的一些内容感到非常熟悉。

本书的一本分章节将会以Node.js环境为背景进行说明，如果你有Node.js基础的话，那么一定会非常容易理解这部分内容了。

格式约定

本书为了节约篇幅，用了下面一些格式上的约定。

- 关于Promise的术语请参考[术语集](#)。
 - 一般一个名词第一次出现时都会附带相关链接。
- 实例方法都用 `instance#method` 的形式。
 - 比如 `Promise#then` 这种写法表示的是 Promise的实例对象的 `then` 这一方法。
- 对象方法都采用 `object.method` 的形式。
 - 这沿用了JavaScript中的使用方式，`Promise.all` 表示的是一个静态方法。

这部分内容主要讲述的是对正文部分的补充说明。

推荐浏览器

我们推荐使用内置对Promise支持的浏览器来阅读本书。

Firefox和Chrome的话都支持[ES6 Promises](#)标准。

此外，虽然不是推荐的阅读环境，但是读者还是能在iOS等移动终端上阅读本书。

运行示例代码

本网站使用了Promise的[Polyfill](#)类库，因此即使在不支持Promise的浏览器上也能执行示例代码。

此外像下面这样，各位读者可以通过运行按钮来运行可执行的示例代码。

```
var promise = new Promise(function(resolve){
  resolve(42);
});
promise.then(function(value){
  console.log(value);
}).catch(function(error){
  console.error(error);
});
```

[运行]

按下 [运行] 按钮之后，代码区会变成编辑区，代码也会被执行。当然你也可以通过这个按钮再次运行代码。
[清除log] 按钮用于清除由 `console.log` 打印出来的log。
[退出]按钮用来退出编辑模式。

如果你对哪里有疑问的话，都可以现场修改代码并执行，以加深对该部分代码的理解。

本书源代码/License

本书中示例代码都可以在GitHub上找到。

本书采用 [AsciiDoc](#) 格式编写。

- [azu/promises-book](#) 

此外代码仓库中还包含本书示例代码的测试代码。

源代码的许可证为MIT许可证，文章内容可以基于CC-BY-NC使用。

意见和疑问

如果有意见或者问题的话，可以直接在GitHub上提Issue即可。

- [Issues · azu/promises-book](#) 日文版
- [Issues · liubin/promises-book](#) 中文版

此外，你也可以在 [在线聊天](#) 上留言。

- 

各位读者除了能免费阅读本书，也有编辑本书的权利。你可以在GitHub上通过 [Pull Requests](#) 来贡献自己的工作。

什么是Promise

本章将主要对JavaScript中的Promise进行入门级的介绍。

什么是Promise

首先让我们了解一下到底什么是Promise。

Promise是抽象异步处理对象以及对其进行各种操作的组件。其详细内容在接下来我们还会进行介绍，Promise并不是从JavaScript中发祥的概念。

Promise最初被提出是在 E 语言中，它是基于并列/并行处理设计的一种编程语言。

现在JavaScript也拥有了这种特性，这就是本书所介绍的JavaScript Promise。

另外，如果说到基于JavaScript的异步处理，我想大多数都会想到利用回调函数。

使用了回调函数的异步处理

```
getAsync("fileA.txt", function(error, result){
    if(error){// 取得失败时的处理
        throw error;
    }
    // 取得成功时的处理
});
```

<1> 传给回调函数的参数为(error对象， 执行结果)组合 Node.js等则规定在JavaScript的回调函数的第一个参数为 Error 对象，这也是它的一个惯例。

像上面这样基于回调函数的异步处理如果统一参数使用规则的话，写法也会很明了。但是，这也仅是编码规约而已，即使采用不同的写法也不会出错。

而Promise则是把类似的异步处理对象和处理规则进行规范化，并按照采用统一的接口来编写，而采取规定方法之外的写法都会出错。

下面是使用了Promise进行异步处理的一个例子

```
var promise = getAsyncPromise("fileA.txt");
promise.then(function(result){
    // 获取文件内容成功时的处理
}).catch(function(error){
    // 获取文件内容失败时的处理
});
```

<1> 返回promise对象 我们可以向这个预设了抽象化异步处理的promise对象，注册这个promise对象执行成功时和失败时相应的回调函数。

这和回调函数方式相比有哪些不同之处呢？在使用promise进行一步处理的时候，我们必须按照接口规定的方法编写处理代码。

也就是说，除promise对象规定的方法(这里的 then 或 catch)以外的方法都是不可以使用的，而不会像回调函数方式那样可以自己自由的定义回调函数的参数，而必须严格遵守固定、统一的编程方式来编写代码。

这样，基于Promise的统一接口的做法，就可以形成基于接口的各种各样的异步处理模式。

所以，promise的功能是可以将复杂的异步处理轻松地进行模式化，这也可以说得上是使用promise的理由之一。

接下来，让我们在实践中来学习JavaScript的Promise吧。

Promise 简介

在 ES6 Promises 标准中定义的API还不是很多。

目前大致有下面三种类型。

Constructor

Promise类似于 XMLHttpRequest，从构造函数 Promise 来创建一个新建新promise对象作为接口。

要想创建一个promise对象、可以使用new来调用Promise的构造器来进行实例化。

```
var promise = new Promise(function(resolve, reject) { // 异步处理 // 处理结束后、调用resolve  
或 reject }); Instance Method
```

对通过new生成的promise对象为了设置其值在 resolve(成功) / reject(失败)时调用的回调函数可以使用promise.then() 实例方法。


```
promise.then(onFulfilled, onRejected)
resolve(成功)时
onFulfilled 会被调用
```

```
reject(失败)时
onRejected 会被调用
onFulfilled、onRejected 两个都为可选参数。
```

promise.then 成功和失败时都可以使用。 另外在只想对异常进行处理时可以采用 promise.then(undefined, or

```
promise.catch(onRejected)
Static Method
```

像 Promise 这样的全局对象还拥有一些静态方法。

包括 Promise.all() 还有 Promise.resolve() 等在内，主要都是一些对Promise进行操作的辅助方法。

Promise workflow

我们先来看一看下面的示例代码。

```
promise-workflow.js
function asyncFunction() {

    return new Promise(function (resolve, reject) {
        setTimeout(function () {
            resolve('Async Hello world');
        }, 16);
    });
}

asyncFunction().then(function (value) {
    console.log(value);    // => 'Async Hello world'
}).catch(function (error) {
    console.log(error);
});
```

运行 new Promise构造器之后，会返回一个promise对象

<1>为promise对象用设置 .then 调用返回值时的回调函数。 asyncFunction 这个函数会返回 promise对象， 对于这个promise对象，我们调用它的 then 方法来设置resolve后的回调函数， catch 方法来设置发生错误时的回调函数。

该promise对象会在setTimeout之后的16ms时被resolve, 这时 then 的回调函数会被调用，并输出 'Async Hello world' 。

在这种情况下 `catch` 的回调函数并不会被执行（因为`promise`返回了`resolve`），不过如果运行环境没有提供 `setTimeout` 函数的话，那么上面代码在执行中就会产生异常，在 `catch` 中设置的回调函数就会被执行。

当然，像`promise.then(onFulfilled, onRejected)` 的方法声明一样，如果不使用`catch` 方法只使用 `then`方法的话，如下所示的代码也能完成相同的工作。

```
asyncFunction().then(function (value) {  
  console.log(value);  
}, function (error) {  
  console.log(error);  
});
```

Promise的状态

我们已经大概了解了Promise的处理流程，接下来让我们来稍微整理一下Promise的状态。

用`new Promise` 实例化的`promise`对象有以下三个状态。

"has-resolution" - Fulfilled `resolve`(成功)时。此时会调用 `onFulfilled`

"has-rejection" - Rejected `reject`(失败)时。此时会调用 `onRejected`

"unresolved" - Pending 既不是`resolve`也不是`reject`的状态。也就是`promise`对象刚被创建后的初始化状态等 关于上面这三种状态的读法，其中 左侧为在 ES6 Promises 规范中定义的术语，而右侧则是在 Promises/A+ 中描述状态的术语。

基本上状态在代码中是不会涉及到的，所以名称也无需太在意。在这本书中，我们会基于 Promises/A+ 中 Pending、Fulfilled、Rejected 的状态名称进行讲述。

promise-states Figure 1. promise states 在 ECMAScript Language Specification ECMA-262 6th Edition – DRAFT 中 `[[PromiseStatus]]` 都是在内部定义的状态。由于没有公开的访问 `[[PromiseStatus]]` 的用户API，所以暂时还没有查询其内部状态的方法。到此在本文中我们已经介绍了`promise`所有的三种状态。

`promise`对象的状态，从Pending转换为Fulfilled或Rejected之后，这个`promise`对象的状态就不会再发生任何变化。

也就是说，Promise与Event等不同，在`.then` 后执行的函数可以肯定地说只会被调用一次。

另外，Fulfilled和Rejected这两个中的任一状态都可以表示为Settled(不变的)。

Settled `resolve`(成功) 或 `reject`(失败)。从Pending和Settled的对称关系来看，Promise状态的种类/迁移是非常简单易懂的。

当`promise`的对象状态发生变化时，用`.then` 来定义只会被调用一次的函数。

JavaScript Promises - Thinking Sync in an Async World // Speaker Deck 这个ppt中有关于Promise状态迁移的非常容易理解的说明。

编写Promise代码

这里我们来介绍一下如何编写Promise代码。

创建promise对象

创建promise对象的流程如下所示。

`new Promise(fn)` 返回一个promise对象

在fn 中指定异步等处理

处理结果正常的话，调用`resolve(处理结果值)`

处理结果错误的话，调用`reject(Error对象)`

按这个流程我们来实际编写下promise代码吧。

我们的任务是用Promise来通过异步处理方式来获取XMLHttpRequest(XHR)的数据。

创建XHR的promise对象

首先，创建一个用Promise把XHR处理包装起来的名为 `getURL` 的函数。

xhr-promise.js

```
function getURL(URL) {
    return new Promise(function (resolve, reject) {
        var req = new XMLHttpRequest();
        req.open('GET', URL, true);
        req.onload = function () {
            if (req.status === 200) {
                resolve(req.responseText);
            } else {
                reject(new Error(req.statusText));
            }
        };
        req.onerror = function () {
            reject(new Error(req.statusText));
        };
        req.send();
    });
}

// 运行示例
var URL = "http://httpbin.org/get";
getURL(URL).then(function onFulfilled(value){
    console.log(value);
}).catch(function onRejected(error){
    console.error(error);
});
```

运行 `getURL` 只有在通过XHR取得结果状态为200时才会调用 `resolve` - 也就是只有数据取得成功时，而其他情况（取得失败）时则会调用 `reject` 方法。

`resolve(req.responseText)` 在response的内容中加入了参数。 `resolve` 方法的参数并没有特别的规则，基本上把要传给回调函数参数放进去就可以了。（`then` 方法可以接收到这个参数值）

熟悉Node.js的人，经常会在写回调函数时将 `callback(error, response)` 的第一个参数设为 `error` 对象，而在Promise中`resolve/reject`则担当了这个职责（处理正常和异常的情况），所以在`resolve`方法中只传一个`response`参数是没有问题的。

接下来我们来看一下`reject`函数。

XHR中 `onerror` 事件被触发的时候就是发生错误时，所以理所当然调用`reject`。 这里我们重点来看一下传给`reject`的值。

发生错误时要像这样 `reject(new Error(req.statusText));`， 创建一个`Error`对象后再将具体的值传进去。 传给`reject` 的参数也没有什么特殊的限制，一般只要是`Error`对象（或者继承自`Error`对象）就可以。

传给`reject` 的参数，其中一般是包含了`reject`原因的`Error`对象。 本次因为状态值不等于200而被`reject`，所以`reject` 中放入的是`statusText`。（这个参数的值可以被 `then` 方法的第二个参数或者 `catch` 方法中使用）

编写promise对象处理方法

让我们在实际中使用一下刚才创建的返回promise对象的函数

```
getURL("http://example.com/"); // => 返回promise对象
```

如Promises Overview 中做的简单介绍一样，promise对象拥有几个实例方法，我们使用这些实例方法来为promise对象创建依赖于promise的具体状态、并且只会被执行一次的回调函数。

为promise对象添加处理方法主要有以下两种

promise对象被 resolve 时的处理(onFulfilled)

promise对象被 reject 时的处理(onRejected)

promise-resolve-flow Figure 2. promise value flow 首先，我们来尝试一下为 getURL 通信成功并取到值时添加的处理函数。

此时所谓的 通信成功， 指的就是在被resolve后， promise对象变为FulFilled状态。

被resolve后的处理，可以在.then 方法中传入想要调用的函数。

```
var URL = "http://httpbin.org/get";
getURL(URL).then(function onFulfilled(value){
    console.log(value);
});
```

为了方便理解我们把函数命名为 onFulfilled getURL函数 中的 resolve(req.responseText); 会将promise对象变为resolve（Fulfilled）状态，同时使用其值调用 onFulfilled 函数。

不过目前我们还没有对其中可能发生的错误做任何处理，接下来，我们就来为 getURL 函数添加发生错误时的异常处理。

此时 发生错误， 指的也就是reject后 promise对象变为Rejected状态。

被reject后的处理，可以在.then 的第二个参数 或者是在 .catch 方法中设置想要调用的函数。

把下面reject时的处理加入到刚才的代码，如下所示。

```
var URL = "http://httpbin.org/status/500";
getURL(URL).then(function onFulfilled(value){
    console.log(value);
}).catch(function onRejected(error){
    console.error(error);
});
```

服务端返回的状态码为500 为了方便理解函数被命名为 `onRejected` 在`getURL` 的处理中发生任何异常，或者被明确`reject`的情况下，该异常原因（`Error`对象）会作为 `.catch` 方法的参数被调用。

其实 `.catch`只是 `promise.then(undefined, onRejected)` 的别名而已，如下代码也可以完成同样的功能。

`getURL(URL).then(onFulfilled, onRejected);` `onFulfilled`, `onRejected` 是和刚才相同的函数 一般说来，使用`.catch`来将`resolve`和`reject`处理分开来写是比较推荐的做法，这两者的区别会在`then`和`catch`的区别中再做详细介绍。

总结

在本章我们简单介绍了以下内容：

用 `new Promise` 方法创建`promise`对象

用`.then` 或 `.catch` 添加`promise`对象的处理函数

到此为止我们已经学习了`Promise`的基本写法。其他很多处理都是由此基本语法延伸的，也使用了`Promise`提供的一些静态方法来实现。

实际上即使使用回调方式的写法也能完成上面同样的工作，而使用`Promise`方式的话有什么优点么？在本小节中我们没有讲到两者的对比及`Promise`的优点。在接下来的章节中，我们将会对`Promise`优点之一，即错误处理机制进行介绍，以及和传统的回调方式的对比。

实战Promise

本章我们将会学习Promise提供的各种方法以及如何如何进行错误处理。

Promise.resolve

一般情况下我们都会使用 `new Promise()` 来创建promise对象，但是除此之外我们也可以使用其他方法。

在这里，我们将会学习如何使用 `Promise.resolve` 和 `Promise.reject`这两个方法。

new Promise的快捷方式

静态方法`Promise.resolve(value)` 可以认为是 `new Promise()` 方法的快捷方式。

比如 `Promise.resolve(42)`; 可以认为是以下代码的语法糖。

```
new Promise(function(resolve){
  resolve(42);
});
```

在这段代码中的 `resolve(42)`; 会让这个promise对象立即进入确定（即resolved）状态，并将42 传递给后面then里所指定的 `onFulfilled` 函数。

方法 `Promise.resolve(value)`; 的返回值也是一个promise对象，所以我们可以像下面那样接着对其返回值进行 `.then` 调用。

```
Promise.resolve(42).then(function(value){
  console.log(value);
});
```

运行 `Promise.resolve`作为 `new Promise()` 的快捷方式，在进行promise对象的初始化或者编写测试代码的时候都非常方便。

Thenable

`Promise.resolve` 方法另一个作用就是将 `thenable` 对象转换为promise对象。

ES6 Promises里提到了Thenable这个概念，简单来说它就是一个非常类似promise的东西。

就像我们有时称具有 `.length` 方法的非数组对象为Array like一样，`thenable`指的是一个具有 `.then` 方法的对象。

这种将`thenable`对象转换为promise对象的机制要求`thenable`对象所拥有的 `then` 方法应该和 `Promise`所拥有的 `then` 方法具有同样的功能和处理过程，在将`thenable`对象转换为promise对象的时候，还会巧妙的利用`thenable`对象原来具有的 `then` 方法。

到底什么样的对象能算是thenable的呢，最简单的例子就是 jQuery.ajax()，它的返回值就是 thenable 的。

因为 jQuery.ajax() 的返回值是 jqXHR Object 对象，这个对象具有 .then 方法。

`$.ajax('/json/comment.json');` // => 拥有 `.then` 方法的对象 这个 thenable 的对象可以使用 `Promise.resolve` 来转换为一个 promise 对象。

变成了 promise 对象的话，就能直接使用 `then` 或者 `catch` 等这些在 ES6 Promises 里定义的方法了。

将 thenable 对象转换 promise 对象

```
var promise = Promise.resolve($.ajax('/json/comment.json')); // => promise 对象
promise.then(function(value){
  console.log(value);
});
```

jQuery 和 thenable jQuery.ajax() 的返回值是一个具有 .then 方法的 jqXHR Object 对象，这个对象继承了来自 Deferred Object 的方法和属性。

但是 Deferred Object 并没有遵循 Promises/A+ 或 ES6 Promises 标准，所以即使看上去这个对象转换成了一个 promise 对象，但是会出现缺失部分信息的问题。

这个问题的根源在于 jQuery 的 Deferred Object 的 then 方法机制与 promise 不同。

所以我们应该注意，即使一个对象具有 .then 方法，也不一定就能作为 ES6 Promises 对象使用。

JavaScript Promises: There and back again - HTML5 Rocks

You're Missing the Point of Promises

https://twitter.com/hirano_y_aa/status/398851806383452160

`Promise.resolve` 只使用了共通的方法 `then`，提供了在不同的类库之间进行 promise 对象互相转换的功能。

这种转换为 thenable 的功能在之前是通过使用 `Promise.cast` 来完成的，从它的名字我们也不难想象它的功能是什么。

除了在编写使用 Promise 的类库等软件时需要了解 Thenable 之外，通常作为 end-user 使用的时候，我们可能不会用到此功能。

我们会在后面第4章的 `Promise.resolve` 和 Thenable 中进行详细的说明，介绍一下结合使用了 Thenable 和 `Promise.resolve` 的具体例子。简单总结一下 `Promise.resolve` 方法的话，可以认为它的作用就是将传递给它的参数填充（Fulfilled）到 promise 对象后并返回这个 promise 对象。

此外，Promise的很多处理内部也是使用了 `Promise.resolve` 算法将值转换为promise对象后再进行处理的。

Promise.reject

`Promise.reject(error)`是和 `Promise.resolve(value)` 类似的静态方法，是 `new Promise()` 方法的快捷方式。

比如 `Promise.reject(new Error("出错了"))` 就是下面代码的语法糖形式。

```
new Promise(function(resolve, reject){
  reject(new Error("出错了"));
});
```

这段代码的功能是调用该promise对象通过then指定的 `onRejected` 函数，并将错误（Error）对象传递给这个 `onRejected` 函数。

```
Promise.reject(new Error("BOOM!")).catch(function(error){
  console.error(error);
});
```

运行

它和`Promise.resolve(value)`的不同之处在于promise内调用的函数是`reject`而不是`resolve`，这在编写测试代码或者进行debug时，说不定会用得上。

专栏: Promise只能进行异步操作？

在使用`Promise.resolve(value)`等方法的时候，如果`promise`对象立刻就能进入`resolve`状态的话，那么你是不是觉得`.then`里面指定的方法就是同步调用的呢？

实际上，`.then`中指定的方法调用是异步进行的。

```
var promise = new Promise(function (resolve){
  console.log("inner promise"); // 1
  resolve(42);
});
promise.then(function(value){
  console.log(value); // 3
});
console.log("outer promise"); // 2
```

运行 执行上面的代码会输出下面的log，从这些log我们清楚地知道了上面代码的执行顺序。

```
inner promise // 1
outer promise // 2
42           // 3
```

由于JavaScript代码会按照文件的从上到下的顺序执行，所以最开始 <1> 会执行，然后是`resolve(42);`被执行。这时候 `promise` 对象的已经变为确定状态，`Fulfilled`被设置为了 42。

下面的代码 `promise.then` 注册了 <3> 这个回调函数，这是本专栏的焦点问题。

由于 `promise.then` 执行的时候`promise`对象已经是确定状态，从程序上说对回调函数进行同步调用也是行得通的。

但是即使在调用 `promise.then` 注册回调函数的时候`promise`对象已经是确定的状态，`Promise`也会以异步的方式调用该回调函数，这是在`Promise`设计上的规定方针。

因此 <2> 会最先被调用，最后才会调用回调函数 <3>。

为什么要对明明可以以同步方式进行调用的函数，非要使用异步的调用方式呢？

同步调用和异步调用同时存在导致的混乱

其实在`Promise`之外也存在这个问题，这里我们以一般的使用情况来考虑此问题。

这个问题的本质是接收回调函数的函数，会根据具体的执行情况，可以选择是以同步还是异步的方式对回调函数进行调用。

下面我们以 `onReady(fn)` 为例进行说明，这个函数会接收一个回调函数进行处理。

```
mixed-onready.js
function onReady(fn) {
  var readyState = document.readyState;
  if (readyState === 'interactive' || readyState === 'complete') {
    fn();
  } else {
    window.addEventListener('DOMContentLoaded', fn);
  }
}
onReady(function () {
  console.log('DOM fully loaded and parsed');
});
console.log('==Starting==');
```

运行 `mixed-onready.js` 会根据执行时 DOM 是否已经装载完毕来决定是对回调函数进行同步调用还是异步调用。

如果在调用 `onReady` 之前 DOM 已经载入的话 对回调函数进行同步调用

如果在调用 `onReady` 之前 DOM 还没有载入的话 通过注册 `DOMContentLoaded` 事件监听器来对回调函数进行异步调用 因此，如果这段代码在源文件中出现的位置不同，在控制台上打印的 log 消息顺序也会不同。

为了解决这个问题，我们可以选择统一使用异步调用的方式。

```
async-onready.js
function onReady(fn) {
  var readyState = document.readyState;
  if (readyState === 'interactive' || readyState === 'complete') {
    setTimeout(fn, 0);
  } else {
    window.addEventListener('DOMContentLoaded', fn);
  }
}
onReady(function () {
  console.log('DOM fully loaded and parsed');
});
console.log('==Starting==');
```

运行 关于这个问题，在 *Effective JavaScript* 的第67项 不要对异步回调函数进行同步调用 中也有详细介绍。

绝对不能对异步回调函数（即使在数据已经就绪）进行同步调用。

如果对异步回调函数进行同步调用的话，处理顺序可能会与预期不符，可能带来意料之外的后果。

对异步回调函数进行同步调用，还可能导致栈溢出或异常处理错乱等问题。

如果想在将来某时刻调用异步回调函数的话，可以使用 `setTimeout` 等异步API。

Effective JavaScript — David Herman 前面我们看到的 `promise.then` 也属于此类，为了避免上述中同时使用同步、异步调用可能引起的混乱问题，Promise在规范上规定 Promise只能使用异步调用方式。

最后，如果将上面的 `onReady` 函数用Promise重写的话，代码如下面所示。

```
onready-as-promise.js
function onReadyPromise() {
  return new Promise(function (resolve, reject) {
    var readyState = document.readyState;
    if (readyState === 'interactive' || readyState === 'complete') {
      resolve();
    } else {
      window.addEventListener('DOMContentLoaded', resolve);
    }
  });
}
onReadyPromise().then(function () {
  console.log('DOM fully loaded and parsed');
});
console.log('==Starting==');
```

运行 由于Promise保证了每次调用都是以异步方式进行的，所以我们在实际编码中不需要调用 `setTimeout` 来自己实现异步调用。

Promise#then

在前面的章节里我们对Promise基本的实例方法 `then` 和 `catch` 的使用方法进行了说明。

这其中，我想大家已经认识了 `.then().catch()` 这种链式方法的写法了，其实在Promise里可以将任意个方法连在一起作为一个方法链（method chain）。

promise可以写成方法链的形式

```
aPromise.then(function taskA(value){
  // task A
}).then(function taskB(vaue){
  // task B
}).catch(function onRejected(error){
  console.log(error);
});
```

如果把在 `then` 中注册的每个回调函数称为task的话，那么我们就可以通过Promise方法链方式来编写能以taskA → task B 这种流程进行处理的逻辑了。

Promise方法链这种叫法有点长（其实是在日语里有点长，中文还可以--译者注），因此后面我们会简化为 `promise chain` 这种叫法。

Promise之所以适合编写异步处理较多的应用，`promise chain`可以算得上是其中的一个原因吧。

在本小节，我们将主要针对使用 `then` 的`promise chain`的行为和流程进行学习。

promise chain

在第一章 `promise chain` 里我们看到了一个很简单的 `then` → `catch` 的例子，如果我们将方法链的长度变得更长的话，那在每个promise对象中注册的`onFulfilled`和`onRejected`将会怎样执行呢？

`promise chain` - 即方法链越短越好。在这个例子里我们是为了方便说明才选择了较长的方法链。我们先来看看下面这样的`promise chain`。

```
promise-then-catch-flow.js
function taskA() {
  console.log("Task A");
}
function taskB() {
  console.log("Task B");
}
function onRejected(error) {
  console.log("Catch Error: A or B", error);
}
function finalTask() {
  console.log("Final Task");
}

var promise = Promise.resolve();
promise
  .then(taskA)
  .then(taskB)
  .catch(onRejected)
  .then(finalTask);
```

运行上面代码中的promise chain的执行流程，如果用一张图来描述一下的话，像下面的图那样。

promise-then-catch-flow Figure 3. promise-then-catch-flow.js附图 在上述代码中，我们没有为 then 方法指定第二个参数(onRejected)，也可以像下面这样来理解。

then 注册onFulfilled时的回调函数

catch 注册onRejected时的回调函数 再看一下上面的流程图的话，我们会发现 Task A 和 Task B 都有指向 onRejected 的线出来。

这些线的意思是在 Task A 或 Task B 的处理中，在下面的情况下就会调用 onRejected 方法。

发生异常的时候

返回了一个Rejected状态的promise对象

在第一章中我们已经看到，Promise中的处理习惯上都会采用 try-catch 的风格，当发生异常的时候，会被 catch 捕获并被由在此函数注册的回调函数进行错误处理。

另一种异常处理策略是通过 返回一个Rejected状态的promise对象 来实现的，这种方法不通过 throw 就能在promise chain中对 onRejected 进行调用。

关于这种方法由于和本小节关系不大就不在这里详述了，大家可以参考一下第4章 使用reject 而不是throw 中的内容。

此外在promise chain中，由于在 onRejected 和 Final Task 后面没有 catch 处理了，因此在这两个Task中如果出现异常的话将不会被捕获，这点需要注意一下。

下面我们再来看一个具体的关于 Task A → onRejected 的例子。

Task A产生异常的例子

Task A 处理中发生异常的话，会按照TaskA → onRejected → FinalTask 这个流程来进行处理。

promise taska rejected flow Figure 4. Task A产生异常时的示意图 将上面流程写成代码的话如下所示。

```
promise-then-taska-throw.js
function taskA() {
  console.log("Task A");
  throw new Error("throw Error @ Task A")
}
function taskB() {
  console.log("Task B");// 不会被调用
}
function onRejected(error) {
  console.log(error);// => "throw Error @ Task A"
}
function finalTask() {
  console.log("Final Task");
}

var promise = Promise.resolve();
promise
  .then(taskA)
  .then(taskB)
  .catch(onRejected)
  .then(finalTask);
```

运行 执行这段代码我们会发现 Task B 是不会被调用的。

在本例中我们在taskA中使用了 throw 方法故意制造了一个异常。但在实际中想主动进行 onRejected调用的时候，应该返回一个Rejected状态的promise对象。关于这种两种方法的异同，请参考 使用reject而不是throw 中的讲解。

promise chain 中如何传递参数

前面例子中的Task都是相互独立的，只是被简单调用而已。

这时候如果 Task A 想给 Task B 传递一个参数该怎么办呢？

答案非常简单，那就是在 Task A 中 return 的返回值，会在 Task B 执行时传给它。

我们还是先来看一个具体例子吧。

```
promise-then-passing-value.js
function doubleUp(value) {
  return value * 2;
}
function increment(value) {
  return value + 1;
}
function output(value) {
  console.log(value); // => (1 + 1) * 2
}

var promise = Promise.resolve(1);
promise
  .then(increment)
  .then(doubleUp)
  .then(output)
  .catch(function(error){
    // promise chain中出现异常的时候会被调用
    console.error(error);
  });
```

运行 这段代码的入口函数是 `Promise.resolve(1);`，整体的promise chain执行流程如下所示。

`Promise.resolve(1);` 传递 1 给 `increment` 函数

函数 `increment` 对接收的参数进行 +1 操作并返回（通过`return`）

这时参数变为2，并再次传给 `doubleUp` 函数

最后在函数 `output` 中打印结果

promise-then-passing-value Figure 5. promise-then-passing-value.js示意图 每个方法中 `return` 的值不仅只局限于字符串或者数值类型，也可以是对象或者promise对象等复杂类型。

`return`的值会由 `Promise.resolve(return的返回值);` 进行相应的包装处理，因此不管回调函数中会返回一个什么样的值，最终 `then` 的结果都是返回一个新创建的promise对象。

关于这部分内容可以参考 专栏: 每次调用`then`都会返回一个新创建的promise对象，那里也对一些常见错误进行了介绍。也就是说，`Promise#then` 不仅仅是注册一个回调函数那么简单，它还会将回调函数的返回值进行变换，创建并返回一个promise对象。

Promise#catch

在前面的Promise#then 的章节里，我们已经简单地使用了 Promise#catch 方法。

这里我们再说一遍，实际上 Promise#catch 只是 promise.then(undefined, onRejected); 方法的一个别名而已。也就是说，这个方法用来注册当promise对象状态变为Rejected时的回调函数。

关于如何根据场景使用 Promise#then 和 Promise#catch 可以参考 then or catch? 中介绍的内容。

IE8的问题

Build Status

上面的这张图，是下面这段代码在使用 polyfill 的情况下在个浏览器上执行的结果。

polyfill是一个支持在不具备某一功能的浏览器上使用该功能的Library。 这里我们使用的例子则来源于 [jakearchibald/es6-promise](#) 。 Promise#catch的运行结果

```
var promise = Promise.reject(new Error("message"));
promise.catch(function (error) {
  console.error(error);
});
```

运行 如果我们在各种浏览器中执行这段代码，那么在IE8及以下版本则会出现 identifier not found 的语法错误。

这是怎么回事呢？实际上这和 catch 是ECMAScript的 保留字 (Reserved Word)有关。

在ECMAScript 3中保留字是不能作为对象的属性名使用的。而IE8及以下版本都是基于ECMAScript 3实现的，因此不能将 catch 作为属性来使用，也就不能编写类似 promise.catch() 的代码，因此就出现了 identifier not found 这种语法错误了。

而现在的浏览器都是基于ECMAScript 5的，而在ECMAScript 5中保留字都属于 IdentifierName ，也可以作为属性名使用了。

在ECMAScript5中保留字也不能作为 Identifier 即变量名或方法名使用。如果我们定义了一个名为 for 的变量的话，那么就不能和循环语句的 for 区分了。而作为属性名的话，我们还是很容易区分 object.for 和 for 的，仔细想想我们就应该能接受将保留字作为属性名来使用了。当然，我们也可以想办法回避这个ECMAScript 3保留字带来的问题。

点标记法（dot notation）要求对象的属性必须是有效的标识符（在ECMAScript 3中则不能使用保留字），

但是使用 中括号标记法（bracket notation）的话，则可以将非合法标识符作为对象的属性名使用。

也就是说，上面的代码如果像下面这样重写的话，就能在IE8及以下版本的浏览器中运行了（当然还需要polyfill）。

解决Promise#catch标识符冲突问题

```
var promise = Promise.reject(new Error("message"));
promise["catch"](function (error) {
    console.error(error);
});
```

运行 或者我们不单纯的使用 catch，而是使用 then 也是可以避免这个问题的。

使用Promise#then代替Promise#catch

```
var promise = Promise.reject(new Error("message"));
promise.then(undefined, function (error) {
    console.error(error);
});
```

运行 由于 catch 标识符可能会导致问题出现，因此一些类库（Library）也采用了 caught 作为函数名，而函数要完成的工作是一样的。

而且很多压缩工具自带了将 promise.catch 转换为 promise["catch"] 的功能，所以可能不经意间也能帮我们解决这个问题。

如果各位读者需要支持IE8及以下版本的浏览器的话，那么一定要将这个 catch 问题牢记在心中。

专栏: 每次调用then都会返回一个新创建的promise对象

从代码上乍一看，`aPromise.then(...).catch(...)` 像是针对最初的 `aPromise` 对象进行了一连串的方法链调用。

然而实际上不管是 `then` 还是 `catch` 方法调用，都返回了一个新的promise对象。

下面我们就来看看如何确认这两个方法返回的到底是不是新的promise对象。

```
var aPromise = new Promise(function (resolve) {
  resolve(100);
});
var thenPromise = aPromise.then(function (value) {
  console.log(value);
});
var catchPromise = thenPromise.catch(function (error) {
  console.error(error);
});
console.log(aPromise !== thenPromise); // => true
console.log(thenPromise !== catchPromise); // => true
```

运行 `===` 是严格相等比较运算符，我们可以看出这三个对象都是互不相同的，这也就证明了 `then` 和 `catch` 都返回了和调用者不同的promise对象。

Then Catch flow 我们在对Promise进行扩展的时候需要牢牢记住这一点，否则稍不留神就有可能对错误的promise对象进行了处理。

如果我们知道了 `then` 方法每次都会创建并返回一个新的promise对象的话，那么我们就应该不难理解下面代码中对 `then` 的使用方式上的差别了。

// 1: 对同一个promise对象同时调用 `then` 方法

```
var aPromise = new Promise(function (resolve) {
  resolve(100);
});
aPromise.then(function (value) {
  return value * 2;
});
aPromise.then(function (value) {
  return value * 2;
});
aPromise.then(function (value) {
  console.log("1: " + value); // => 100
})

// vs

// 2: 对 `then` 进行 promise chain 方式进行调用
var bPromise = new Promise(function (resolve) {
  resolve(100);
});
bPromise.then(function (value) {
  return value * 2;
}).then(function (value) {
  return value * 2;
}).then(function (value) {
  console.log("2: " + value); // => 100 * 2 * 2
});
```

运行 第1种写法中并没有使用promise的方法链方式，这在Promise中是应该极力避免的写法。这种写法中的 then 调用几乎是在同时开始执行的，而且传给每个 then 方法的 value 值都是 100。

第2中写法则采用了方法链的方式将多个 then 方法调用串连在了一起，各函数也会严格按照 resolve → then → then → then 的顺序执行，并且传给每个 then 方法的 value 的值都是前一个promise对象通过 return 返回的值。

下面是一个由方法1中的 then 用法导致的比较容易出现很有代表性的反模式的例子。

✗ then 的错误使用方法

```
function badAsyncCall() {
  var promise = Promise.resolve();
  promise.then(function() {
    // 任意处理
    return newVar;
  });
  return promise;
}
```

这种写法有很多问题，首先在 `promise.then` 中产生的异常不会被外部捕获，此外，也不能得到 `then` 的返回值，即使其有返回值。

由于每次 `promise.then` 调用都会返回一个新创建的`promise`对象，因此需要像上述方式2那样，采用`promise chain`的方式将调用进行链式化，修改后的代码如下所示。

`then` 返回返回新创建的`promise`对象

```
function anAsyncCall() {
  var promise = Promise.resolve();
  return promise.then(function() {
    // 任意处理
    return newVar;
  });
}
```

关于这些反模式，详细内容可以参考 [Promise Anti-patterns](#)。

这种函数的行为贯穿在`Promise`整体之中，包括我们后面要进行说明的 `Promise.all` 和 `Promise.race`，他们都会接收一个`promise`对象为参数，并返回一个和接收参数不同的、新的`promise`对象。

Promise和数组

到目前为止我们已经学习了如何通过 `.then` 和 `.catch` 来注册回调函数，这些回调函数会在 `promise` 对象变为 `Fulfilled` 或 `Rejected` 状态之后被调用。

如果只有一个 `promise` 对象的话我们可以像前面介绍的那样编写代码就可以了，如果要在多个 `promise` 对象都变为 `Fulfilled` 状态的时候才要进行某种处理该如何操作呢？

我们以当所有 XHR（异步处理）全部结束后要进行某操作为例来进行说明。

各位读者现在也许有点难以在大脑中描绘出这么一种场景，我们可以先看一下下面使用了普通的回调函数风格的 XHR 处理代码。

通过回调方式来进行多个异步调用

```
multiple-xhr-callback.js
function getURLCallback(URL, callback) {
    var req = new XMLHttpRequest();
    req.open('GET', URL, true);
    req.onload = function () {
        if (req.status === 200) {
            callback(null, req.responseText);
        } else {
            callback(new Error(req.statusText), req.response);
        }
    };
    req.onerror = function () {
        callback(new Error(req.statusText));
    };
    req.send();
}
// <1> 对JSON数据进行安全的解析
function jsonParse(callback, error, value) {
    if (error) {
        callback(error, value);
    } else {
        try {
            var result = JSON.parse(value);
            callback(null, result);
        } catch (e) {
            callback(e, value);
        }
    }
}
// <2> 发送XHR请求
var request = {
    comment: function getComment(callback) {
```



```
        return getURLCallback('http://azu.github.io/promises-book/json/comment.json',
    },
    people: function getPeople(callback) {
        return getURLCallback('http://azu.github.io/promises-book/json/people.json',
    }
    };
// <3> 启动多个XHR请求，当所有请求返回时调用callback
function allRequest(requests, callback, results) {
    if (requests.length === 0) {
        return callback(null, results);
    }
    var req = requests.shift();
    req(function (error, value) {
        if (error) {
            callback(error, value);
        } else {
            results.push(value);
            allRequest(requests, callback, results);
        }
    });
}
function main(callback) {
    allRequest([request.comment, request.people], callback, []);
}
// 运行的例子
main(function(error, results){
    if(error){
        return console.error(error);
    }
    console.log(results);
});
```

运行 这段回调函数风格的代码有以下几个要点。

直接使用 `JSON.parse` 函数的话可能会抛出异常，所以这里使用了一个包装函数 `jsonParse`

如果将多个XHR处理进行嵌套调用的话层次会比较深，所以使用了 `allRequest` 函数并在其中对`request`进行调用。

回调函数采用了 `callback(error,value)` 这种写法，第一个参数表示错误信息，第二个参数为返回值

在使用 `jsonParse` 函数的时候我们使用了 `bind` 进行绑定，通过使用这种偏函数（Partial Function）的方式就可以减少匿名函数的使用。（如果在函数回调风格的代码能很好的做到函数分离的话，也能减少匿名函数的数量）

```
jsonParse.bind(null, callback);  
// 可以认为这种写法能转换为以下的写法  
function bindJSONParse(error, value){  
    jsonParse(callback, error, value);  
}
```

在这段回调风格的代码中，我们也能发现如下一些问题。

需要显示进行异常处理

为了不让嵌套层次太深，需要一个对request进行处理的函数

到处都是回调函数

下面我们再来看看如何使用 Promise#then 来完成同样的工作。

使用Promise#then同时处理多个异步请求

需要事先说明的是 Promise.all 比较适合这种应用场景的需求，因此我们故意采用了大量 .then 的晦涩的写法。

使用了.then 的话，也并不是说能和回调风格完全一致，大概重写后代码如下所示。

```
multiple-xhr.js
function getURL(URL) {
    return new Promise(function (resolve, reject) {
        var req = new XMLHttpRequest();
        req.open('GET', URL, true);
        req.onload = function () {
            if (req.status === 200) {
                resolve(req.responseText);
            } else {
                reject(new Error(req.statusText));
            }
        };
        req.onerror = function () {
            reject(new Error(req.statusText));
        };
        req.send();
    });
}

var request = {
    comment: function getComment() {
        return getURL('http://azu.github.io/promises-book/json/comment.json').then(JS
    },
    people: function getPeople() {
        return getURL('http://azu.github.io/promises-book/json/people.json').then(JS
    }
};

function main() {
    function recordValue(results, value) {
        results.push(value);
        return results;
    }
    // [] 用来保存初始化的值
    var pushValue = recordValue.bind(null, []);
    return request.comment().then(pushValue).then(request.people).then(pushValue);
}

// 运行的例子
main().then(function (value) {
    console.log(value);
}).catch(function (error) {
    console.error(error);
});
```

运行 将上述代码和回调函数风格相比，我们可以得到如下结论。

可以直接使用 `JSON.parse` 函数

函数 `main()` 返回promise对象

错误处理的地方直接对返回的promise对象进行处理

向前面我们说的那样，main的 then 部分有点晦涩难懂。

为了应对这种需要对多个异步调用进行统一处理的场景，Promise准备了 Promise.all 和 Promise.race 这两个静态方法。

在下面的小节中我们将对这两个函数进行说明。

Promise.all

Promise.all 接收一个 promise 对象的数组作为参数，当这个数组里的所有 promise 对象全部变为 resolve 或 reject 状态的时候，它才会去调用 .then 方法。

前面我们看到的批量获得若干XHR的请求结果的例子，使用 Promise.all 的话代码会非常简单。

之前例子中的 getURL 返回了一个 promise 对象，它封装了XHR通信的实现。向 Promise.all 传递一个由封装了XHR通信的 promise 对象数组的话，则只有在全部的XHR通信完成之后（变为 Fulfilled 或 Rejected 状态）之后，才会调用 .then 方法。

```
promise-all-xhr.js
function getURL(URL) {
  return new Promise(function (resolve, reject) {
    var req = new XMLHttpRequest();
    req.open('GET', URL, true);
    req.onload = function () {
      if (req.status === 200) {
        resolve(req.responseText);
      } else {
        reject(new Error(req.statusText));
      }
    };
    req.onerror = function () {
      reject(new Error(req.statusText));
    };
    req.send();
  });
}
var request = {
  comment: function getComment() {
    return getURL('http://azu.github.io/promises-book/json/comment.json').then(JS
  },
  people: function getPeople() {
    return getURL('http://azu.github.io/promises-book/json/people.json').then(JS
  }
};
function main() {
  return Promise.all([request.comment(), request.people()]);
}
// 运行示例
main().then(function (value) {
  console.log(value);
}).catch(function (error) {
  console.log(error);
});
```

运行这个例子的执行方法和前面的例子一样。不过Promise.all在以下几点和之前的例子有所不同。

main中的处理流程显得非常清晰

Promise.all接收promise对象组成的数组作为参数

Promise.all([request.comment(), request.people()]); 在上面的代码中，request.comment()和request.people()会同时开始执行，而且每个promise的结果（resolve或reject时传递的参数值），和传递给Promise.all的promise数组的顺序是一致的。

也就是说，这时候.then得到的promise数组的执行结果的顺序是固定的，即[comment, people]。

main().then(function (results) { console.log(results); // 按照[comment, people]的顺序 }); 如果像下面那样使用一个计时器来计算一下程序执行时间的话，那么就可以非常清楚的知道传递给Promise.all的promise数组是同时开始执行的。

```
promise-all-timer.js
// `delay`毫秒后执行resolve
function timerPromisify(delay) {
  return new Promise(function (resolve) {
    setTimeout(function () {
      resolve(delay);
    }, delay);
  });
}
var startDate = Date.now();
// 所有promise变为resolve后程序退出
Promise.all([
  timerPromisify(1),
  timerPromisify(32),
  timerPromisify(64),
  timerPromisify(128)
]).then(function (values) {
  console.log(Date.now() - startDate + 'ms');
  // 約128ms
  console.log(values); // [1, 32, 64, 128]
});
```

运行timerPromisify会每隔一定时间（通过参数指定）之后，返回一个promise对象，状态为Fulfilled，其状态值为传给timerPromisify的参数。

而传给Promise.all的则是由上述promise组成的数组。

```
var promises = [  
  timerPromisify(1),  
  timerPromisify(32),  
  timerPromisify(64),  
  timerPromisify(128)  
];
```

这时候，每隔1, 32, 64, 128 ms都会有一个promise发生 resolve 行为。

也就是说，这个promise对象数组中所有promise都变为resolve状态的话，至少需要128ms。实际我们计算一下Promise.all 的执行时间的话，它确实是消耗了128ms的时间。

从上述结果可以看出，传递给 Promise.all 的promise并不是一个个的顺序执行的，而是同时开始、并行执行的。

如果这些promise全部串行处理的话，那么需要 等待1ms → 等待32ms → 等待64ms → 等待128ms，全部执行完毕需要225ms的时间。

要想了解更多关于如何使用Promise进行串行处理的内容，可以参考第4章的Promise中的串行处理中的介绍。

Promise.race

接着我们来看看和 Promise.all 类似的对多个promise对象进行处理的 Promise.race 方法。

它的使用方法和Promise.all一样，接收一个promise对象数组为参数。

Promise.all 在接收到的所有的对象promise都变为 FulFilled 或者 Rejected 状态之后才会继续进行后面的处理，与之相对的是 Promise.race 只要有一个promise对象进入 FulFilled 或者 Rejected 状态的话，就会继续进行后面的处理。

像Promise.all时的例子一样，我们来看一个带计时器的 Promise.race 的使用例子。

```
promise-race-timer.js
// `delay` 毫秒后执行resolve
function timerPromisify(delay) {
  return new Promise(function (resolve) {
    setTimeout(function () {
      resolve(delay);
    }, delay);
  });
}
// 任何一个promise变为resolve或reject 的话程序就停止运行
Promise.race([
  timerPromisify(1),
  timerPromisify(32),
  timerPromisify(64),
  timerPromisify(128)
]).then(function (value) {
  console.log(value);    // => 1
});
```

运行 上面的代码创建了4个promise对象，这些promise对象会分别在1ms，32ms，64ms和128ms后变为确定状态，即FulFilled，并且在第一个变为确定状态的1ms后，.then 注册的回调函数就会被调用，这时候确定状态的promise对象会调用 resolve(1) 因此传递给 value 的值也是1，控制台上会打印出1来。

下面我们再来看看在第一个promise对象变为确定（FulFilled）状态后，它之后的promise对象是否还在继续运行。


```
promise-race-other.js
var winnerPromise = new Promise(function (resolve) {
  setTimeout(function () {
    console.log('this is winner');
    resolve('this is winner');
  }, 4);
});
var loserPromise = new Promise(function (resolve) {
  setTimeout(function () {
    console.log('this is loser');
    resolve('this is loser');
  }, 1000);
});
// 第一个promise变为resolve后程序停止
Promise.race([winnerPromise, loserPromise]).then(function (value) {
  console.log(value);    // => 'this is winner'
});
```

运行 我们在前面代码的基础上增加了 `console.log` 用来输出调试信息。

执行上面代码的话，我们会看到 `winner`和`loser` `promise`对象的 `setTimeout` 方法都会执行完毕， `console.log` 也会分别输出它们的信息。

也就是说， `Promise.race` 在第一个`promise`对象变为`Fulfilled`之后，并不会取消其他`promise`对象的执行。

在 `ES6 Promises` 规范中，也没有取消（中断）`promise`对象执行的概念，我们必须确保 `promise`最终进入`resolve` or `reject`状态之一。也就是说`Promise`并不适用于 状态 可能会固定不变的处理。也有一些类库提供了对`promise`进行取消的操作。

then or catch?

在上一章里，我们说过 `.catch` 也可以理解为 `promise.then(undefined, onRejected)`。

在本书里我们还是会将 `.catch` 和 `.then` 分开使用来进行错误处理的。

此外我们也会学习一下，在 `.then` 里同时指定处理对错误进行处理的函数相比，和使用 `catch` 又有什么异同。

不能进行错误处理的onRejected

我们看看下面的这段代码。

then-throw-error.js

```
function throwError(value) {
  // 抛出异常
  throw new Error(value);
}
// <1> onRejected不会被调用
function badMain(onRejected) {
  return Promise.resolve(42).then(throwError, onRejected);
}
// <2> 有异常发生时onRejected会被调用
function goodMain(onRejected) {
  return Promise.resolve(42).then(throwError).catch(onRejected);
}
// 运行示例
badMain(function(){
  console.log("BAD");
});
goodMain(function(){
  console.log("GOOD");
});
```

运行 在上面的代码中，`badMain` 是一个不太好的实现方式（但也不是说它有多坏），`goodMain` 则是一个能非常好的进行错误处理的版本。

为什么说 `badMain` 不好呢？，因为虽然我们在 `.then` 的第二个参数中指定了用来错误处理的函数，但实际上它却不能捕获第一个参数 `onFulfilled` 指定的函数（本例为 `throwError`）里面出现的错误。

也就是说，这时候即使 `throwError` 抛出了异常，`onRejected` 指定的函数也不会被调用（即不会输出“BAD”字样）。

与此相对的是，goodMain的代码则遵循了throwError→onRejected的调用流程。这时候throwError中出现异常的话，在会被方法链中的下一个方法，即.catch所捕获，进行相应的错误处理。

.then方法中的onRejected参数所指定的回调函数，实际上针对的是其promise对象或者之前的promise对象，而不是针对.then方法里面指定的第一个参数，即onFulfilled所指向的对象，这也是then和catch表现不同的原因。

.then和.catch都会创建并返回一个新的promise对象。Promise实际上每次在方法链中增加一次处理的时候所操作的都不是完全相同的promise对象。Then Catch flow Figure 6. Then Catch flow 这种情况下then是针对Promise.resolve(42)的处理，在onFulfilled中发生异常，在同一个then方法中指定的onRejected也不能捕获该异常。

在这个then中发生的异常，只有在该方法链后面出现的catch方法才能捕获。

当然，由于.catch方法是.then的别名，我们使用.then也能完成同样的工作。只不过使用.catch的话意图更明确，更容易理解。

```
Promise.resolve(42).then(throwError).then(null, onRejected);
```

总结

这里我们又学习到了如下一些内容。

1. 使用promise.then(onFulfilled, onRejected)的话
 - 在onFulfilled中发生异常的话，在onRejected中是捕获不到这个异常的。
2. 在promise.then(onFulfilled).catch(onRejected)的情况下
 - then中产生的异常能在.catch中捕获
3. .then和.catch在本质上是没区别的
 - 需要分场合使用。

我们需要注意如果代码类似badMain那样的话，就可能出现程序不会按预期运行的情况，从而不能正确的进行错误处理。

Promise测试

这章我们学习如果编写Promise 的测试代码

基本测试

关于ES6 Promises的语法我们已经学了一些，我想大家应该也能够在实际项目中编写Promise 的Demo代码了吧。

这时，接下来你可能要苦恼该如何编写Promise 的测试代码了。

那么让我们先来学习下如何使用 Mocha来对Promise 进行基本的测试吧。

先声明一下，这章中涉及的测试代码都是运行在Node.js环境下的。

本书中出现的示例代码也都有相应的测试代码。测试代码可以参考 [azu/promises-book](#)。

Mocha

Mocha是Node.js下的测试框架工具,在这里，我们并不打算对 Mocha本身进行详细讲解。对Mocha感兴趣的读者可以自行学习。

Mocha可以自由选择BDD、TDD、exports中的任意风格，测试中用到的Assert 方法也同样可以跟任何其他类库组合使用。也就是说，Mocha本身只提供执行测试时的框架，而其他部分则由使用者自己选择。

这里我们选择使用Mocha，主要基于下面3点理由。

它是非常著名的测试框架

支持基于Node.js 和浏览器的测试

支持"Promise测试"

最后至于为什么说 支持"Promise测试"，这个我们在后面再讲。

要想在本章中使用Mocha，我们需要先通过npm来安装Mocha。

`$ npm install -g mocha` 另外，Assert库我们使用的是Node.js自带的assert模块，所以不需要额外安装。

首先，让我们试着编写一个对传统回调风格的异步函数进行测试的代码。

回调函数风格的测试

如果想使用回调函数风格来对一个异步处理进行测试，使用Mocha的话代码如下所示。

basic-test.js

```
var assert = require('power-assert');
describe('Basic Test', function () {
  context('When Callback(high-order function)', function () {
    it('should use `done` for test', function (done) {
      setTimeout(function () {
        assert(true);
        done();
      }, 0);
    });
  });
  context('When promise object', function () {
    it('should use `done` for test?', function (done) {
      var promise = Promise.resolve(1);
      // このテストコードはある欠陥があります
      promise.then(function (value) {
        assert(value === 1);
        done();
      });
    });
  });
});
```

将这段代码保存为 basic-test.js，之后就可以使用刚才安装的Mocha的命令行工具进行测试了。

```
$ mocha basic-test.js
```

Mocha的 it 方法指定了 done 参数，在 done() 函数被执行之前，该测试一直处于等待状态，这样就可以对异步处理进行测试。

Mocha中的异步测试，将会按照下面的步骤执行。

```
it("should use `done` for test", function (done) {

  setTimeout(function () {
    assert(true);
    done();
  }, 0);
});
```

回调式的异步处理 调用done 后测试结束 这也是一种非常常见的实现方式。

使用done 的Promise测试

接下来，让我们看看如何使用 `done` 来进行Promise测试。

```
it("should use `done` for test?", function (done) {
  var promise = Promise.resolve(42);
  promise.then(function (value) {
    assert(value === 42);
    done();
  });
});
```

创建名为Fulfilled的promise对象 调用done 后测试结束 Promise.resolve 用来返回promise对象， 返回的promise对象状态为FulFilled。最后，通过 `.then` 设置的回调函数也会被调用。

像专栏: Promise只能进行异步操作？中已经提到的那样， promise对象的调用总是异步进行的，所以测试也同样需要以异步调用的方式来编写。

但是，在前面的测试代码中，在assert失败的情况下就会出现問題。

对异常promise测试

```
it("should use `done` for test?", function (done) {
  var promise = Promise.resolve();
  promise.then(function (value) {
    assert(false); // => throw AssertionError
    done();
  });
});
```

在此次测试中 `assert` 失败了，所以你可能认为应该抛出“测试失败”的错误，而实际情况却是测试并不会结束，直到超时。

promise test timeout Figure 7. 由于测试不会结束，所以直到发生超时时间未知，一直会处于挂起状态。通常情况下，`assert` 失败的时候，会throw一个error，测试框架会捕获该error，来判断测试失败。

但是，Promise的情况下 `.then` 绑定的函数执行时发生的error会被Promise捕获，而测试框架则对此error将会一无所知。

我们来改善一下`assert`失败的promise测试，让它能正确处理 `assert` 失败时的测试结果。

测试正常失败的示例

```
it("should use `done` for test?", function (done) {  
  var promise = Promise.resolve();  
  promise.then(function (value) {  
    assert(false);  
  }).then(done, done);  
});
```

在上面测试正常失败的示例中，为了确保 `done` 一定会被调用，我们在最后添加了 `.then(done, done);` 语句。

`assert` 测试通过（成功）时会调用 `done()`，而 `assert` 失败时则调用 `done(error)`。

这样，我们就编写出了和 回调函数风格的测试 相同的Promise测试。

但是，为了处理 `assert` 失败的情况，我们需要额外添加 `.then(done, done);` 的代码。这就要求我们在编写Promise测试时要格外小心，忘了加上上面语句的话，很可能就会写出一个永远不会返回直到超时的测试代码。

在下一节，让我们接着学习一下最初提到的使用Mocha理由中的支持"Promises测试"究竟是一种什么机制。

Mocha对Promise的支持

在这里，我们将会学习什么是Mocha支持的“对Promise测试”。

官方网站 [Asynchronous code](#) 也记载了关于Promise测试的概要。

Alternately, instead of using the `done()` callback, you can return a promise. This is useful if the APIs you are testing return promises instead of taking callbacks: 这段话的意思是，在对Promise进行测试的时候，不使用 `done()` 这样的回调风格的代码编写方式，而是返回一个 `promise` 对象。

那么实际上代码将会是什么样的呢？这里我们来看个具体的例子应该容易理解了。

```
mocha-promise-test.js  
var assert = require('power-assert');  
describe('Promise Test', function () {  
  it('should return a promise object', function () {  
    var promise = Promise.resolve(1);  
    return promise.then(function (value) {  
      assert(value === 1);  
    });  
  });  
});
```

这段代码将前面 前面使用 done 的例子 按照Mocha的Promise测试方式进行了重写。

修改的地方主要在以下两点：

删除了 done

返回结果为promise对象

采用这种写法的话，当 assert 失败的时候，测试本身自然也会失败。

```
it("should be fail", function () {
  return Promise.resolve().then(function () {
    assert(false); // => 测试失败
  });
});
```

采用这种方法，就能从根本上省略诸如 .then(done, done); 这样本质上跟测试逻辑并无直接关系的代码。

Mocha已经支持对Promises的测试 | Web scratch 这篇（日语）文章里也提到了关于Mocha对Promise测试的支持。 3.2.1. 意料之外（失败的）的测试结果 因为Mocha提供了对Promise的测试，所以我们会认为按照Mocha的规则来写会比较好。 但是这种代码可能会带来意想不到的异常情况的发生。

比如对下面的maybeRejected() 函数的测试代码，该函数返回一个当满足某一条件就变为Rejected的promise对象。

想对Error Object进行测试

```
function maybeRejected(){
  return Promise.reject(new Error("woo"));
}
it("is bad pattern", function () {
  return maybeRejected().catch(function (error) {
    assert(error.message === "woo");
  });
});
```

这个函数用来对返回的promise对象进行测试 这个测试的目的包括以下两点：

maybeRejected() 返回的promise对象如果变为Fulfilled状态的话 测试将会失败

maybeRejected() 返回的promise对象如果变为Rejected状态的话 在 assert 中对Error对象进行检查 上面的测试代码，当promise对象变为Rejected的时候，会调用在 onRejected 中注册的函数，从而没有走正promise的处理常流程，测试会成功。

这段测试代码的问题在于当`maybeRejected()` 返回的是一个 为`Fulfilled`状态的`promise`对象时，测试会一直成功。

```
function maybeRejected(){
    return Promise.resolve();
}
it("is bad pattern", function () {
    return maybeRejected().catch(function (error) {
        assert(error.message === "woo");
    });
});
```

返回的`promise`对象会变为`Fulfilled` 在这种情况下，由于在 `catch` 中注册的 `onRejected` 函数并不会被调用，因此 `assert` 也不会被执行，测试会一直通过（passed，成功）。

为了解决这个问题，我们可以在 `.catch` 的前面加入一个 `.then` 调用，可以理解为如果调用了 `.then` 的话，那么测试就需要失败。

```
function failTest() {
    throw new Error("Expected promise to be rejected but it was fulfilled");
}
function maybeRejected(){
    return Promise.resolve();
}
it("should bad pattern", function () {
    return maybeRejected().then(failTest).catch(function (error) {
        assert.deepEqual(error.message === "woo");
    });
});
```

通过`throw`来使测试失败 但是，这种写法会像在前面 `then or catch?` 中已经介绍的一样，`failTest` 抛出的异常会被 `catch` 捕获。

Then Catch flow Figure 8. Then Catch flow 程序的执行流程为 `then` → `catch`，传递给 `catch` 的`Error`对象为`AssertionError`类型，这并不是我们想要的东西。

也就是说，我们希望测试只能通过状态会变为`onRejected`的`promise`对象，如果`promise`对象状态为`onFulfilled`状态的话，那么该测试就会一直通过。

明确两种状态，改善测试中的意外（异常）状况

在编写 上面对`Error`对象进行测试的例子 时，怎么才能剔除那些会意外通过测试的情况呢？

最简单的方式就是像下面这样，在测试代码中判断在各种`promise`对象的状态下，应进行如何的操作。

变为Fulfilled状态的时候 测试会预期失败

变为Rejected状态的时候 使用 `assert` 进行测试 也就是说，我们需要在测试代码中明确指定在Fulfilled和Rejected这两种状态下，都需进行什么样的处理。

```
function maybeRejected() {
  return Promise.resolve();
}
it("catch -> then", function () {
  // 变为Fulfilled的时候测试失败
  return maybeRejected().then(failTest, function (error) {
    assert(error.message === "woo");
  });
});
```

像这样的话，就能在promise变为Fulfilled的时候编写出失败用的测试代码了。

Promise onRejected test Figure 9. Promise onRejected test 在 then or catch? 中我们已经讲过，为了避免遗漏对错误的处理，与使用 `.then(onFulfilled, onRejected)` 这样带有二个参数的调用形式相比，我们更推荐使用 `then → catch` 这样的处理方式。

但是在编写测试代码的时候，Promise强大的错误处理机制反而成了限制我们的障碍。因此我们不得已采取了 `.then(failTest, onRejected)` 这种写法，明确指定promise在各种状态下进行何种的处理。

总结

在本小节中我们对在使用Mocha进行Promise测试时可能出现的一些意外情况进行了介绍。

普通的代码采用 `then → catch` 的流程的话比较容易理解

这是为了错误处理的方便。请参考 then or catch?

将测试代码集中到 `then` 中处理

为了能将AssertionError对象传递到测试框架中。

通过使用 `.then(onFulfilled, onRejected)` 这种形式的写法，我们可以明确指定promise对象在变为 Fulfilled或Rejected时如何处理。

但是，由于需要显示的指定 Rejected时的测试处理，像下面这样的代码看起来总是有一些让人感到不太直观的感觉。

`promise.then(failTest, function(error){ // 使用assert对error进行测试 });` 在下一小节，我们会介绍如何编写helper函数以方便编写Promise的测试代码，以及怎样去编写更容易理解的测试代码。

编写可控测试（controllable tests）

在继续进行说明之前，我们先来定义一下什么是可控测试。在这里我们对可控测试的定义如下。

待测试的promise对象

如果编写预期为Fulfilled状态的测试的话

Rejected的时候要 Fail

assertion 的结果不一致的时候要 Fail

如果预期为Rejected状态的话

结果为Fulfilled 测试为 Fail

assertion 的结果不一致的时候要 Fail

如果一个测试能网罗上面的用例（Fail）项，那么我们就称其为可控测试。

也就是说，一个测试用例应该包括下面的测试内容。

结果满足 Fulfilled or Rejected 之一

对传递给assertion的值进行检查

在前面使用了 .then 的代码就是一个期望结果为 Rejected 的测试。

```
promise.then(failTest, function(error){
  // 通过assert验证error对象
  assert(error instanceof Error);
});
```

必须明确指定转换后的状态

为了编写有效的测试代码，我们需要明确指定 promise的状态 为 Fulfilled or Rejected 的两者之一。

但是由于 .then 的话在调用的时候可以省略参数，有时候可能会忘记加入使测试失败的条件。

因此，我们可以定义一个helper函数，用来明确定义promise期望的状态。

笔者（原著者）创建了一个类库 `azu/promise-test-helper` 以方便对Promise进行测试，本文中用的是这个类库的简略版。首先我们创建一个名为 `shouldRejected` 的helper函数，用来在刚才的 .then 的例子中，期待测试返回状态为 `onRejected` 的结果的例子。

```

shouldRejected-test.js
var assert = require('power-assert');
function shouldRejected(promise) {
  return {
    'catch': function (fn) {
      return promise.then(function () {
        throw new Error('Expected promise to be rejected but it was fulfilled');
      }, function (reason) {
        fn.call(promise, reason);
      });
    }
  };
}
it('should be rejected', function () {
  var promise = Promise.reject(new Error('human error'));
  return shouldRejected(promise).catch(function (error) {
    assert(error.message === 'human error');
  });
});

```

shouldRejected 函数接收一个promise对象作为参数，并且返回一个带有 catch 方法的对象。

在这个 catch 中可以使用和 onRejected 里一样的代码，因此我们可以在 catch 使用基于 assertion 方法的测试代码。

在 shouldRejected 外部，都是类似如下、和普通的promise处理大同小异的代码。

将需要测试的promise对象传递给 shouldRejected 方法

在返回的对象的 catch 方法中编写进行onRejected处理的代码

在onRejected里使用assertion进行判断

在使用 shouldRejected 函数的时候，如果是 Fulfilled 被调用了的话，则会throw一个异常，测试也会失败。

```

promise.then(failTest, function(error){
  assert(error.message === 'human error');
});
// == 几乎这两段代码是同样的意思
shouldRejected(promise).catch(function (error) {
  assert(error.message === 'human error');
});

```

使用 shouldRejected 这样的helper函数，测试代码也会变得很直观。

Promise onRejected test Figure 10. Promise onRejected test 像上面一样，我们也可以编写一个测试promise对象期待结果为Fulfilled的 shouldFulfilled helper函数。

```
shouldFulfilled-test.js
var assert = require('power-assert');
function shouldFulfilled(promise) {
  return {
    'then': function (fn) {
      return promise.then(function (value) {
        fn.call(promise, value);
      }, function (reason) {
        throw reason;
      });
    }
  };
}
it('should be fulfilled', function () {
  var promise = Promise.resolve('value');
  return shouldFulfilled(promise).then(function (value) {
    assert(value === 'value');
  });
});
```

这和上面的 `shouldRejected-test.js` 结构基本相同，只不过返回对象的 `catch` 方法变为了 `then`，`promise.then` 的两个参数也调换了。

小结

在本小节我们学习了如何编写针对Promise特定状态的测试代码，以及如何使用便于测试的helper函数。

这里我们使用到的 `shouldFulfilled` 和 `shouldRejected` 也可以在下面的类库中找到。

`azu/promise-test-helper`。此外，本小节中的helper方法都是以 Mocha对Promise的支持 为前提的，在基于done 的测试 中使用的话可能会比较麻烦。

是使用基于测试框架对Promis的支持，还是使用基于类似done 这样回调风格的测试方式，每个人都可以自由的选择，只是风格问题，我觉得倒没必要去争一个孰优孰劣。

比如在 CoffeeScript下进行测试的话，由于CoffeeScript 会隐式的使用return返回，所以使用done 的话可能更容易理解一些。

对Promise进行测试比对通常的异步函数进行测试坑更多，虽说采取什么样的测试方法是个人的自由，但是在同一项目中采取前后风格一致的测试则是非常重要的。

高级用法

在这一章里，我们会基于前面学到的内容，再深入了解一下Promise里的一些高级内容，加深对Promise的理解。

Promise的实现类库（Library）

在本小节里，我们将不打算对浏览器实现的Promise进行说明，而是要介绍一些第三方实现的和Promise兼容的类库。

为什么需要这些类库？

为什么需要这些类库呢？我想有些读者不免会有此疑问。首先能想到的原因是有些运行环境并不支持 ES6 Promises。

当我们在网上查找Promise的实现类库的时候，有一个因素是首先要考虑的，那就是是否具有 Promises/A+兼容性。

Promises/A+ 是 ES6 Promises 的前身，Promise的 then 也是来自于此的基于社区的规范。

如果说一个类库兼容 Promises/A+ 的话，那么就是说它除了具有标准的 then 方法之外，很多情况下也说明此类库还支持 Promise.all 和 catch 等功能。

但是 Promises/A+ 实际上只是定义了关于 Promise#then 的规范，所以有些类库可能实现了其它诸如 all 或 catch 等功能，但是可能名字却不一样。

如果我们说一个类库具有 then 兼容性的话，实际上指的是 Thenable，它通过使用 Promise.resolve 基于ES6 Promise的规定，进行promise对象的变换。

ES6 Promise 里关于promise对象的规定包括在使用 catch 方法，或使用 Promise.all 进行处理的时候不能出现错误。4.1.2. Polyfill和扩展类库 在这些Promise的实现类库中，我们这里主要对两种类型的类库进行介绍。

一种是被称为 Polyfill（这是一款英国产品，就是装修刮墙用的腻子，其意义可想而知——译者注）的类库，另一种是即具有 Promises/A+兼容性，又增加了自己独特功能的类库。

Promise的实现类库数量非常之多，这里我们只是介绍了其中有限的几个。Polyfill

只需要在浏览器中加载Polyfill类库，就能使用IE10等或者还没有提供对Promise支持的浏览器中使用Promise里规定的方法。

也就是说如果加载了Polyfill类库，就能在还不支持Promise的环境中，运行本文中的各种示例代码。

jakearchibald/es6-promise 一个兼容 ES6 Promises 的Polyfill类库。它基于 RSVP.js 这个兼容 Promises/A+ 的类库，它只是 RSVP.js 的一个子集，只实现了Promises 规定的 API。

yahoo/ypromise 这是一个独立版本的 YUI 的 Promise Polyfill，具有和 ES6 Promises 的兼容性。本书的示例代码也都是基于这个 ypromise 的 Polyfill 来在线运行的。

getify/native-promise-only 以作为ES6 Promises的polyfill为目的的类库 它严格按照ES6 Promises的规范设计，没有添加在规范中没有定义的功能。如果运行环境有原生的Promise支持的话，则优先使用原生的Promise支持。Promise扩展类库

Promise扩展类库除了实现了Promise中定义的规范之外，还增加了自己独自定义的功能。

Promise扩展类库数量非常的多，我们只介绍其中两个比较有名的类库。

kriskowal/q 类库 Q 实现了 Promises 和 Deferreds 等规范。它自2009年开始开发，还提供了面向Node.js的文件IO API Q-IO 等，是一个在很多场景下都能用得着的类库。

petkaantonov/bluebird 这个类库除了兼容 Promise 规范之外，还扩展了取消promise对象的运行，取得promise的运行进度，以及错误处理的扩展检测等非常丰富的功能，此外它在实现上还在性能问题下了很大的功夫。Q 和 Bluebird 这两个类库除了都能在浏览器里运行之外，充实的API reference也是其特征。

API Reference · kriskowal/q Wiki

Q等文档里详细介绍了Q的Deferred和jQuery里的Deferred有哪些异同，以及要怎么进行迁移 Coming from jQuery 等都进行了详细的说明。

bluebird/API.md at master · petkaantonov/bluebird

Bluebird的文档除了提供了使用Promise丰富的实现方式之外，还涉及到了在出现错误时的对应方法以及 Promise中的反模式 等内容。

这两个类库的文档写得都很友好，即使我们不使用这两个类库，阅读一下它们的文档也具有一定的参考价值。

4.1.3. 总结 本小节介绍了Promise的实现类库中的 Polyfill 和扩展类库这两种。

Promise的实现类库种类繁多，到底选择哪个来使用完全看自己的喜好了。

但是由于这些类库实现的 Promise 同时具有 Promises/A+ 或 ES6 Promises 共通的接口，所以在使用某一类库的时候，有时候也可以参考一下其他类库的代码或者扩展功能。

熟练掌握Promise中的共通概念，进而能在实际中能对这些技术运用自如，这也是本书的写作目的之一。

Promise.resolve和Thenable

在第二章的Promise.resolve 中我们已经说过， Promise.resolve 的最大特征之一就是可以将thenable的对象转换为promise对象。

在本小节里，我们将学习一下利用将thenable对象转换为promise对象这个功能都能具体做些什么事情。

4.2.1. 将Web Notifications转换为thenable对象 这里我们以桌面通知 API Web Notifications 为例进行说明。

关于Web Notifications API的详细信息可以参考下面的网址。

使用 Web Notifications - WebAPI | MDN

Can I use Web Notifications

简单来说，Web Notifications API就是能像以下代码那样通过 new Notification 来显示通知消息。

new Notification("Hi!"); 当然，为了显示通知消息，我们需要在运行 new Notification 之前，先获得用户的许可。

确认是否允许Notification的对话框 Figure 11. 确认是否允许Notification的对话框 用户在这个是否允许Notification的对话框选择后的结果，会通过 Notification.permission 传给我们的程序，它的值可能是允许("granted")或拒绝("denied")这二者之一。

是否允许Notification对话框中的可选项，在Firefox中除了允许、拒绝之外，还增加了 永久有效 和 会话范围内有效 两种额外选项，当然 Notification.permission 的值都是一样的。在程序中可以通过 Notification.requestPermission() 来弹出是否允许Notification对话框，用户选择的结果会通过 status 参数传给回调函数。

从这个回调函数我们也可以看出来，用户选择允许还是拒绝通知是异步进行的。

Notification.requestPermission(function (status) { // status的值为 "granted" 或 "denied"
console.log(status); }); 运行 到用户收到并显示通知为止，整体的处理流程如下所示。

显示是否允许通知的对话框，并异步处理用户选择结果

如果用户允许的话，则通过 new Notification 显示通知消息。这又分两种情况

用户之前已经允许过

当场弹出是否允许桌面通知对话框

当用户不允许的时候，不执行任何操作

虽然上面说到了几种情景，但是最终结果就是用户允许或者拒绝，可以总结为如下两种模式。

允许时("granted") 使用 `new Notification` 创建通知消息

拒绝时("denied") 没有任何操作 这两种模式是不是觉得有在哪里看过的感觉？呵呵，用户的选择结果，正和在Promise中promise对象变为 `Fulfilled` 或 `Rejected` 状态非常类似。

`resolve(成功)`时 == 用户允许("granted") 调用 `onFulfilled` 方法

`reject(失败)`时 == 用户拒绝("denied") 调用 `onRejected` 函数 是不是我们可以用Promise的方式去编写桌面通知的代码呢？我们先从回调函数风格的代码入手看看到底怎么去做。

4.2.2. Web Notification 包装函数 (wrapper) 首先，我们以回到函数风格的代码对上面的Web Notification API包装函数进行重写，新代码如下所示。

```
notification-callback.js function notifyMessage(message, options, callback) { if (Notification
&& Notification.permission === 'granted') { var notification = new Notification(message,
options); callback(null, notification); } else if (Notification.requestPermission) {
Notification.requestPermission(function (status) { if (Notification.permission !== status) {
Notification.permission = status; } if (status === 'granted') { var notification = new
Notification(message, options); callback(null, notification); } else { callback(new Error('user
denied')); } }); } else { callback(new Error('doesn\'t support Notification API')); } } // 运行实例 //
第二个参数是传给 Notification 的option对象 notifyMessage("Hi!", {}, function (error,
notification) { if(error){ return console.error(error); } console.log(notification);// 通知对象 }); 运行
在回调风格的代码里，当用户拒绝接收通知的时候， error 会被设置值，而如果用户同意
接收通知的时候，则会显示通知消息并且 notification 会被设置值。
```

回调函数接收error和notification两个参数 `function callback(error, notification){`

`}` 下面，我想再将这个回调函数风格的代码使用Promise进行改写。

4.2.3. Web Notification as Promise 基于上述回调风格的 `notifyMessage` 函数，我们再来创建一个返回promise对象的 `notifyMessageAsPromise` 方法。

```
notification-as-promise.js function notifyMessage(message, options, callback) { if
(Notification && Notification.permission === 'granted') { var notification = new
Notification(message, options); callback(null, notification); } else if
(Notification.requestPermission) { Notification.requestPermission(function (status) { if
(Notification.permission !== status) { Notification.permission = status; } if (status ===
'granted') { var notification = new Notification(message, options); callback(null, notification); }
else { callback(new Error('user denied')); } }); } else { callback(new Error('doesn\'t support
Notification API')); } } function notifyMessageAsPromise(message, options) { return new
Promise(function (resolve, reject) { notifyMessage(message, options, function (error,
notification) { if (error) { reject(error); } else { resolve(notification); } }); }); } // 运行示例
```

`notifyMessageAsPromise("Hi!").then(function (notification) { console.log(notification); // 通知对象 }).catch(function(error){ console.error(error); });` 运行 在用户允许接收通知的时候，运行上面的代码，会显示 "Hi!" 消息。

当用户接收通知消息的时候，`.then` 函数会被调用，当用户拒绝接收消息的时候，`.catch` 方法会被调用。

由于浏览器是以网站为单位保存Web Notifications API的许可状态的，所以实际上有下面四种模式存在。

已经获得用户许可 `.then` 方法被调用

弹出询问对话框并获得许可 `.then` 方法被调用

已经是被用户拒绝的状态 `.catch` 方法被调用

弹出询问对话框并被用户拒绝 `.catch` 方法被调用 也就是说，如果使用原生的Web Notifications API的话，那么需要在程序中对上述四种情况都进行处理，我们可以像下面的包装函数那样，将上述四种情况简化为两种以方便处理。上面的 `notification-as-promise.js` 虽然看上去很方便，但是实际上使用的时候，很可能出现 在不支持Promise的环境下不能使用 的问题。

如果你想编写像`notification-as-promise.js`这样具有Promise风格和的类库的话，我觉得你有如下的一些选择。

支持Promise的环境是前提 需要最终用户保证支持Promise

在不支持Promise的环境下不能正常工作（即应该出错）。

在类库中实现Promise 在类库中实现Promise功能

例如）`localForage`

在回调函数中也应该能够使用 Promise 用户可以选择合适的使用方式

返回Thenable类型

`notification-as-promise.js`就是以Promise存在为前提的写法。

回归正文，在这里Thenable是为了帮助实现在回调函数中也能使用Promise的一个概念。

Web Notifications As Thenable

我们已经说过，`thenable`就是一个具有 `.then`方法的一个对象。下面我们就在`notification-callback.js`中增加一个返回值为 `thenable` 类型的方法。

```
notification-thenable.js function notifyMessage(message, options, callback) { if (Notification
&& Notification.permission === 'granted') { var notification = new Notification(message,
options); callback(null, notification); } else if (Notification.requestPermission) {
Notification.requestPermission(function (status) { if (Notification.permission !== status) {
Notification.permission = status; } if (status === 'granted') { var notification = new
Notification(message, options); callback(null, notification); } else { callback(new Error('user
denied')); } }); } else { callback(new Error('doesn\'t support Notification API')); } } // 返回
thenable function notifyMessageAsThenable(message, options) { return { 'then': function
(resolve, reject) { notifyMessage(message, options, function (error, notification) { if (error) {
reject(error); } else { resolve(notification); } }); } }; } // 运行示例
Promise.resolve(notifyMessageAsThenable("message")).then(function (notification) {
console.log(notification); // 通知对象 }).catch(function (error) { console.error(error); }); 运行
notification-thenable.js里增加了一个 notifyMessageAsThenable方法。这个方法返回的对象
具备一个then方法。
```

then方法的参数和 `new Promise(function (resolve, reject){})` 一样，在确定时执行 `resolve` 方法，拒绝时调用 `reject` 方法。

then 方法和 `notification-as-promise.js` 中的 `notifyMessageAsPromise` 方法完成了同样的工作。

我们可以看出，`Promise.resolve(thenable)` 通过使用 `thenable` 这个promise对象，就能利用Promise功能了。

`Promise.resolve(notifyMessageAsThenable("message")).then(function (notification) { console.log(notification); // 通知对象 }).catch(function (error) { console.error(error); });` 使用了Thenable的`notification-thenable.js` 和依赖于Promise的 `notification-as-promise.js`，实际上都是非常相似的使用方法。

`notification-thenable.js` 和 `notification-as-promise.js`比起来，有以下的不同点。

类库侧没有提供 Promise 的实现

用户通过 `Promise.resolve(thenable)` 来自己实现了 Promise

作为Promise使用的时候，需要和 `Promise.resolve(thenable)` 一起配合使用

通过使用Thenable对象，我们可以实现类似已有的回调式风格和Promise风格中间的一种实现风格。

总结

在本小节我们主要学习了什么是Thenable，以及如何通过`Promise.resolve(thenable)`使用Thenable，将其作为promise对象来使用。

Callback — Thenable — Promise

Thenable风格表现为位于回调和Promise风格中间的一种状态，作为类库的公开API有点不太成熟，所以并不常见。

Thenable本身并不依赖于Promise功能，但是Promise之外也没有使用Thenable的方式，所以可以认为Thenable间接依赖于Promise。

另外，用户需要对 `Promise.resolve(thenable)` 有所理解才能使用好Thenable，因此作为类库的公开API有一部分会比较难。和公开API相比，更多情况下是在内部使用Thenable。

在编写异步处理的类库的时候，推荐采用先编写回调风格的函数，然后再转换为公开API这种方式。

貌似Node.js的Core module就采用了这种方式，除了类库提供的基本回调风格的函数之外，用户也可以通过Promise或者Generator等自己擅长的方式进行实现。

最初就是以能被Promise使用为目的的类库，或者其本身依赖于Promise等情况下，我想将返回promise对象的函数作为公开API应该也没什么问题。什么时候该使用Thenable？

那么，又是在什么情况下应该使用Thenable呢？

恐怕最可能被使用的是在 Promise类库 之间进行相互转换了。

比如，类库Q的Promise实例为Q promise对象，提供了 ES6 Promises 的promise对象不具备的方法。Q promise对象提供了 `promise.finally(callback)` 和 `promise.nodeify(callback)` 等方法。

如果你想将ES6 Promises的promise对象转换为Q promise的对象，轮到Thenable大显身手的时候就到了。

使用thenable将promise对象转换为Q promise对象 `var Q = require("Q");` // 这是一个ES6的promise对象 `var promise = new Promise(function(resolve){ resolve(1); });` // 变换为Q promise对象 `Q(promise).then(function(value){ console.log(value); }).finally(function(){ console.log("finally"); });` 因为是Q promise对象所以可以使用 `finally` 方法 上面代码中最开始被创建的promise对象具备`then`方法，因此是一个Thenable对象。我们可以通过`Q(thenable)`方法，将这个Thenable对象转换为Q promise对象。

可以说它的机制和 `Promise.resolve(thenable)` 一样，当然反过来也一样。

像这样，Promise类库虽然都有自己类型的promise对象，但是它们之间可以通过Thenable这个共通概念，在类库之间（当然也包括native Promise）进行promise对象的相互转换。

我们看到，就像上面那样，Thenable多在类库内部实现中使用，所以从外部来说不会经常看到Thenable的使用。但是我们必须牢记Thenable是Promise中一个非常重要的概念。

使用reject而不是throw

Promise的构造函数，以及被 then 调用执行的函数基本上都可以认为是在 try...catch 代码块中执行的，所以在这些代码中即使使用 throw ，程序本身也不会因为异常而终止。

如果在Promise中使用 throw 语句的话，会被 try...catch 住，最终promise对象也变为 Rejected状态。

```
var promise = new Promise(function(resolve, reject){ throw new Error("message"); });
promise.catch(function(error){ console.error(error);// => "message" });
```

运行 代码像这样其实运行时倒也不会有什么问題，但是如果想把 promise对象状态 设置为Rejected状态的话，使用 reject 方法则更显得合理。

所以上面的代码可以改写为下面这样。

```
var promise = new Promise(function(resolve, reject){ reject(new Error("message")); });
promise.catch(function(error){ console.error(error);// => "message" })
```

运行 其实我们也可以这么来考虑，在出错的时候我们并没有调用 throw 方法，而是使用了 reject ，那么给 reject 方法传递一个Error类型的对象也就很好理解了。

使用reject有什么优点？

话说回来，为什么在想将promise对象的状态设置为Rejected的时候应该使用 reject 而不是 throw 呢？

首先是因为我们很难区分 throw 是我们主动抛出来的，还是因为真正的其它 异常 导致的。

比如在使用Chrome浏览器的时候，Chrome的开发者工具提供了在程序发生异常的时候自动在调试器中break的功能。

Pause On Caught Exceptions Figure 12. Pause On Caught Exceptions 当我们开启这个功能的时候，在执行到下面代码中的 throw 时就会触发调试器的break行为。

```
var promise = new Promise(function(resolve, reject){ throw new Error("message"); });
```

本来这是和调试没有关系的地方，也因为在Promise中的 throw 语句被break了，这也严重的影响了浏览器提供的此功能的正常使用。

在then中进行reject

在Promise构造函数中，有一个用来指定 reject 方法的参数，使用这个参数而不是依靠 throw 将promise对象的状态设置为Rejected状态非常简单。

那么如果像下面那样想在 then 中进行reject的话该怎么办呢？

```
var promise = Promise.resolve(); promise.then(function (value) { setTimeout(function () { // 经过一段时间后还没处理完的话就进行reject - 2 }, 1000); // 比较耗时的处理 - 1 somethingHardWork(); }).catch(function (error) { // 超时错误 - 3 });
```

上面的超时处理，需要在 then 中进行 reject 方法调用，但是传递给当前的回调函数的参数只有前面的一promise对象，这该怎么办呢？

关于使用Promise进行超时处理的具体实现方法可以参考 使用Promise.race和delay取消XHR请求 中的详细说明。在这里我们再次回忆下 then 的工作原理。

在 then 中注册的回调函数可以通过 return 返回一个值，这个返回值会传给后面的 then 或 catch 中的回调函数。

而且return的返回值类型不光是简单的字面值，还可以是复杂的对象类型，比如promise对象等。

这时候，如果返回的是promise对象的话，那么根据这个promise对象的状态，在下一个 then 中注册的回调函数中的onFulfilled和onRejected的哪一个会被调用也是能确定的。

```
var promise = Promise.resolve(); promise.then(function () { var retPromise = new Promise(function (resolve, reject) { // resolve or reject 的状态决定 onFulfilled or onRejected 的哪个方法会被调用 }); return retPromise; }).then(onFulfilled, onRejected);
```

后面的then调用哪个回调函数是由promise对象的状态来决定的 也就是说，这个 retPromise 对象状态为 Rejected的时候，会调用后面then中的 onRejected 方法，这样就实现了即使在 then 中不使用 throw 也能进行reject处理了。

```
var onRejected = console.error.bind(console); var promise = Promise.resolve(); promise.then(function () { var retPromise = new Promise(function (resolve, reject) { reject(new Error("this promise is rejected")); }); return retPromise; }).catch(onRejected);
```

运行使用 Promise.reject 的话还能再将代码进行简化。

```
var onRejected = console.error.bind(console); var promise = Promise.resolve(); promise.then(function () { return Promise.reject(new Error("this promise is rejected")); }).catch(onRejected);
```

运行 4.3.3. 总结 在本小节我们主要学习了

使用 reject 会比使用 throw 安全

在 then 中使用reject的方法

也许实际中我们可能不常使用 reject，但是比起来不假思索的使用 throw 来说，使用 reject 的好处还是很多的。

关于上面讲的内容的比较详细的例子，大家可以参考在 使用Promise.race和delay取消XHR请求 小节的介绍。

Deferred和Promise

这一节我们来简单介绍下Deferred和Promise之间的关系

什么是Deferred？

说起Promise，我想大家一定同时也听说过Deferred这个术语。比如 jQuery.Deferred 和 JSDeferred 等，一定都是大家非常熟悉的内容了。

Deferred和Promise不同，它没有共通的规范，每个Library都是根据自己的喜好来实现的。

在这里，我们打算以 jQuery.Deferred 类似的实现为中心进行介绍。

Deferred和Promise的关系

简单来说，Deferred和Promise具有如下的关系。

Deferred 拥有 Promise

Deferred 具备对 Promise的状态进行操作的特权方法（图中的"特権メソッド"）

Deferred和Promise Figure 13. Deferred和Promise 我想各位看到此图应该就很容易理解了，Deferred和Promise并不是处于竞争的关系，而是Deferred内涵了Promise。

这是jQuery.Deferred结构的简化版。当然也有的Deferred实现并没有内涵Promise。光看图的话也许还难以理解，下面我们就看看怎么通过Promise来实现Deferred。

Deferred top on Promise

基于Promise实现Deferred的例子。

```
deferred.js function Deferred() { this.promise = new Promise(function (resolve, reject) {
this._resolve = resolve; this._reject = reject; }.bind(this)); } Deferred.prototype.resolve =
function (value) { this._resolve.call(this.promise, value); }; Deferred.prototype.reject =
function (reason) { this._reject.call(this.promise, reason); }; 我们再将之前使用Promise实现的
getURL 用Deferred改写一下。
```

```
xhr-deferred.js function Deferred() { this.promise = new Promise(function (resolve, reject) {
this._resolve = resolve; this._reject = reject; }.bind(this)); } Deferred.prototype.resolve =
function (value) { this._resolve.call(this.promise, value); }; Deferred.prototype.reject =
function (reason) { this._reject.call(this.promise, reason); }; function getURL(URL) { var
deferred = new Deferred(); var req = new XMLHttpRequest(); req.open('GET', URL, true);
req.onload = function () { if (req.status === 200) { deferred.resolve(req.responseText); } else
```



```
{ deferred.reject(new Error(req.statusText)); } }; req.onerror = function () {  
deferred.reject(new Error(req.statusText)); }; req.send(); return deferred.promise; } // 运行示  
例 var URL = "http://httpbin.org/get"; getURL(URL).then(function onFulfilled(value){  
console.log(value); }).catch(console.error.bind(console)); 运行 所谓的能对Promise状态进行  
操作的特权方法，指的就是能对promise对象的状态进行resolve、reject等调用的方法，而通  
常的Promise的话只能在通过构造函数传递的方法之内对promise对象的状态进行操作。
```

我们来看看Deferred和Promise相比在实现上有什么异同。

```
xhr-promise.js function getURL(URL) { return new Promise(function (resolve, reject) { var req  
= new XMLHttpRequest(); req.open('GET', URL, true); req.onload = function () { if (req.status  
=== 200) { resolve(req.responseText); } else { reject(new Error(req.statusText)); } };  
req.onerror = function () { reject(new Error(req.statusText)); }; req.send(); }); } // 运行示例 var  
URL = "http://httpbin.org/get"; getURL(URL).then(function onFulfilled(value){  
console.log(value); }).catch(console.error.bind(console)); 运行 对比上述两个版本的 getURL  
，我们发现它们有如下不同。
```

Deferred 的话不需要将代码用Promise括起来

由于没有被嵌套在函数中，可以减少一层缩进

反过来没有Promise里的错误处理逻辑

在以下方面，它们则完成了同样的工作。

整体处理流程

调用 resolve、reject 的时机

函数都返回了promise对象

由于Deferred包含了Promise，所以大体的流程还是差不多的，不过Deferred有用对Promise进行操作的特权方法，以及高度自由的对流程控制进行自由定制。

比如在Promise一般都会在构造函数中编写主要处理逻辑，对 resolve、reject 方法的调用时机也基本是很确定的。

new Promise(function (resolve, reject){ // 在这里进行promise对象的状态确定 }); 而使用Deferred的话，并不需要将处理逻辑写成一大块代码，只需要先创建deferred对象，可以在任何时机对 resolve、reject 方法进行调用。

```
var deferred = new Deferred();
```

```
// 可以在随意的时机对 resolve 、 reject 方法进行调用 上面我们只是简单的实现了一个  
Deferred ， 我想你已经看到了它和 Promise 之间的差异了吧。
```

如果说Promise是用来对值进行抽象的话，Deferred则是对处理还没有结束的状态或操作进行抽象化的对象，我们也可以从这一层的区别来理解一下这两者之间的差异。

换句话说，Promise代表了一个对象，这个对象的状态现在还不确定，但是未来一个时间点它的状态要么变为正常值（Fulfilled），要么变为异常值（Rejected）；而Deferred对象表示了一个处理还没有结束的这种事实，在它的处理结束的时候，可以通过Promise来取得处理结果。

如果各位读者还想深入了解一下Deferred的话，可以参考下面的这些资料。

Promise & Deferred objects in JavaScript Pt.1: Theory and Semantics.

Twisted 入门 — Twisted Intro

Promise anti patterns · petkaantonov/bluebird Wiki

Coming from jQuery · kriskowal/q Wiki

Deferred最初是在Python的 Twisted 框架中被提出来的概念。在JavaScript领域可以认为它是由 MochiKit.Async、dojo/Deferred 等Library引入的。

使用Promise.race和delay取消XHR请求

在本小节中，作为在第2章所学的 Promise.race 的具体例子，我们来看一下如何使用 Promise.race来实现超时机制。

当然XHR有一个 timeout 属性，使用该属性也可以简单实现超时功能，但是为了能支持多个XHR同时超时或者其他功能，我们采用了容易理解的异步方式在XHR中通过超时来实现取消正在进行中的操作。

让Promise等待指定时间

首先我们来看一下如何在Promise中实现超时。

所谓超时就是要在经过一定时间后进行某些操作，使用 setTimeout 的话很好理解。

首先我们来串讲一个单纯的在Promise中调用 setTimeout 的函数。

delayPromise.js function delayPromise(ms) { return new Promise(function (resolve) { setTimeout(resolve, ms); }); } delayPromise(ms) 返回一个在经过了参数指定的毫秒数后进行 onFulfilled操作的promise对象，这和直接使用 setTimeout 函数比较起来只是编码上略有不同，如下所示。

setTimeout(function () { alert("已经过了100ms !"); }, 100); // == 几乎同样的操作
delayPromise(100).then(function () { alert("已经过了100ms !"); }); 在这里 promise对象 这个概念非常重要，请切记。

Promise.race中的超时

让我们回顾一下静态方法 Promise.race，它的作用是在任何一个promise对象进入到确定（解决）状态后就继续进行后续处理，如下面的例子所示。

```
var winnerPromise = new Promise(function (resolve) { setTimeout(function () {  
  console.log('this is winner'); resolve('this is winner'); }, 4); });  
var loserPromise = new Promise(function (resolve) { setTimeout(function () { console.log('this is loser'); resolve('this is loser'); }, 1000); });  
// 第一个promise变为resolve后程序停止  
Promise.race([winnerPromise, loserPromise]).then(function (value) { console.log(value); // => 'this is winner' });  
我们可以将刚才的 delayPromise 和其它promise对象一起放到 Promise.race 中来是实现简单的超时机制。
```

```
simple-timeout-promise.js function delayPromise(ms) { return new Promise(function  
(resolve) { setTimeout(resolve, ms); }); } function timeoutPromise(promise, ms) { var timeout  
= delayPromise(ms).then(function () { throw new Error('Operation timed out after ' + ms + '
```

ms'); }); return Promise.race([promise, timeout]); } 函数 timeoutPromise(比较对象promise, ms) 接收两个参数，第一个是需要使用超时机制的promise对象，第二个参数是超时时间，它返回一个由 Promise.race 创建的相互竞争的promise对象。

之后我们就可以使用 timeoutPromise 编写下面这样的具有超时机制的代码了。

```
function delayPromise(ms) { return new Promise(function (resolve) { setTimeout(resolve, ms); }); } function timeoutPromise(promise, ms) { var timeout = delayPromise(ms).then(function () { throw new Error('Operation timed out after ' + ms + 'ms'); }); return Promise.race([promise, timeout]); } // 运行示例 var taskPromise = new Promise(function(resolve){ // 随便一些什么处理 var delay = Math.random() * 2000; setTimeout(function(){ resolve(delay + "ms"); }, delay); }); timeoutPromise(taskPromise, 1000).then(function(value){ console.log("taskPromise在规定时间内结束：" + value); }).catch(function(error){ console.log("发生超时", error); }); 运行 虽然在发生超时的时候抛出了异常，但是这样的话我们就不能区分这个异常到底是普通的错误还是超时错误了。
```

为了能区分这个 Error 对象的类型，我们再来定义一个Error 对象的子类 TimeoutError。

定制Error对象

Error 对象是ECMAScript的内建（build in）对象。

但是由于stack trace等原因我们不能完美的创建一个继承自 Error 的类，不过在这里我们的目的只是为了和Error有所区别，我们将创建一个 TimeoutError 类来实现我们的目的。

在ECMAScript6中可以使用 class 语法来定义类之间的继承关系。

class MyError extends Error{ // 继承了Error类的对象 } 为了让我们的 TimeoutError 能支持类似 error instanceof TimeoutError 的使用方法，我们还需要进行如下工作。

```
TimeoutError.js function copyOwnFrom(target, source) { Object.getOwnPropertyNames(source).forEach(function (propName) { Object.defineProperty(target, propName, Object.getOwnPropertyDescriptor(source, propName)); }); return target; } function TimeoutError() { var superInstance = Error.apply(null, arguments); copyOwnFrom(this, superInstance); } TimeoutError.prototype = Object.create(Error.prototype); TimeoutError.prototype.constructor = TimeoutError; 我们定义了 TimeoutError 类和构造函数，这个类继承了Error的prototype。
```

它的使用方法和普通的 Error 对象一样，使用 throw 语句即可，如下所示。

```
var promise = new Promise(function(){ throw TimeoutError("timeout"); });
```

promise.catch(function(error){ console.log(error instanceof TimeoutError);// true }); 有了这个 TimeoutError 对象，我们就能很容易区分捕获的到底是因为超时而导致的错误，还是其他原因导致的Error对象了。

本章里介绍的继承JavaScript内建对象的方法可以参考 Chapter 28. Subclassing Built-ins，那里有详细的说明。此外 Error - JavaScript | MDN 也针对Error对象进行了详细说明。

通过超时取消XHR操作

到这里，我想各位读者都已经对如何使用Promise来取消一个XHR请求都有一些思路了吧。

取消XHR操作本身的话并不难，只需要调用 XMLHttpRequest 对象的 abort() 方法就可以了。

为了能在外部调用 abort() 方法，我们先对之前本节出现的 getURL 进行简单的扩展，cancelableXHR 方法除了返回一个包装了XHR的promise对象之外，还返回了一个用于取消该XHR请求的abort方法。

```
delay-race-cancel.js function cancelableXHR(URL) { var req = new XMLHttpRequest(); var
promise = new Promise(function (resolve, reject) { req.open('GET', URL, true); req.onload =
function () { if (req.status === 200) { resolve(req.responseText); } else { reject(new
Error(req.statusText)); } }; req.onerror = function () { reject(new Error(req.statusText)); };
req.onabort = function () { reject(new Error('abort this request')); }; req.send(); }); var abort =
function () { // 如果request还没有结束的话就执行abort //
```

https://developer.mozilla.org/en/docs/Web/API/XMLHttpRequest/Using_XMLHttpRequest if
(req.readyState !== XMLHttpRequest.UNSENT) { req.abort(); } }; return { promise: promise,
abort: abort }; } 在这些问题都明了之后，剩下只需要进行Promise处理的流程进行编码即可。
大体的流程就像下面这样。

通过 cancelableXHR 方法取得包装了XHR的promise对象和取消该XHR请求的方法

在 timeoutPromise 方法中通过 Promise.race 让XHR的包装promise和超时用promise进行竞争。

XHR在超时前返回结果的话

和正常的promise一样，通过 then 返回请求结果

发生超时的时候

抛出 throw TimeoutError 异常并被 catch

catch的错误对象如果是 TimeoutError 类型的话，则调用 abort 方法取消XHR请求

将上面的步骤总结一下的话，代码如下所示。

```
delay-race-cancel-play.js function copyOwnFrom(target, source) {
Object.getOwnPropertyNames(source).forEach(function (propName) {
Object.defineProperty(target, propName, Object.getOwnPropertyDescriptor(source,
propName)); }); return target; } function TimeoutError() { var superInstance = Error.apply(null,
arguments); copyOwnFrom(this, superInstance); } TimeoutError.prototype =
```

```
Object.create(Error.prototype); TimeoutError.prototype.constructor = TimeoutError; function
delayPromise(ms) { return new Promise(function (resolve) { setTimeout(resolve, ms); }); }
function timeoutPromise(promise, ms) { var timeout = delayPromise(ms).then(function () {
return Promise.reject(new TimeoutError('Operation timed out after ' + ms + ' ms')); }); return
Promise.race([promise, timeout]); } function cancelableXHR(URL) { var req = new
XMLHttpRequest(); var promise = new Promise(function (resolve, reject) { req.open('GET',
URL, true); req.onload = function () { if (req.status === 200) { resolve(req.responseText); }
else { reject(new Error(req.statusText)); } }; req.onerror = function () { reject(new
Error(req.statusText)); }; req.onabort = function () { reject(new Error('abort this request')); };
req.send(); }); var abort = function () { // 如果request还没有结束的话就执行abort //
https://developer.mozilla.org/en/docs/Web/API/XMLHttpRequest/Using\_XMLHttpRequest if
(req.readyState !== XMLHttpRequest.UNSENT) { req.abort(); } }; return { promise: promise,
abort: abort }; } var object = cancelableXHR('http://httpbin.org/get'); // main
timeoutPromise(object.promise, 1000).then(function (contents) { console.log('Contents',
contents); }).catch(function (error) { if (error instanceof TimeoutError) { object.abort(); return
console.log(error); } console.log('XHR Error:', error); }); 运行 上面的代码就通过在一一定的时
间内变为解决状态的promise对象实现了超时处理。
```

通常进行开发的情况下，由于这些逻辑会频繁使用，因此将这些代码分割保存在不同的文件应该是一个不错的选择。

promise和操作方法

在前面的 cancelableXHR 中，promise对象及其操作方法都是在一个对象中返回的，看起来稍微有些不太好理解。

从代码组织的角度来说一个函数只返回一个值（promise对象）是一个非常好的习惯，但是由于在外面不能访问 cancelableXHR 方法中创建的 req 变量，所以我们需要编写一个专门的函数（上面的例子中的abort）来对这些内部对象进行处理。

当然也可以考虑到对返回的promise对象进行扩展，使其支持abort方法，但是由于promise对象是对值进行抽象化的对象，如果不加限制的增加操作作用的方法的话，会使整体变得非常复杂。

大家都知道一个函数做太多的工作都不认为是一个好的习惯，因此我们不会让一个函数完成所有功能，也许像下面这样对函数进行分割是一个不错的选择。

返回包含XHR的promise对象

接收promise对象作为参数并取消该对象中的XHR请求

将这些处理整理为一个模块的话，以后扩展起来也方便，一个函数所做的工作也会比较精炼，代码也会更容易阅读和维护。

我们有很多方法来创建一个模块（AMD,CommonJS,ES6 module etc.），在这里，我们将会把前面的 cancelableXHR 整理为一个Node.js的模块使用。

```
cancelableXHR.js "use strict"; var requestMap = {}; function createXHRPromise(URL) { var
req = new XMLHttpRequest(); var promise = new Promise(function (resolve, reject) {
req.open('GET', URL, true); req.onreadystatechange = function () { if (req.readyState ===
XMLHttpRequest.DONE) { delete requestMap[URL]; } }; req.onload = function () { if
(req.status === 200) { resolve(req.responseText); } else { reject(new Error(req.statusText)); }
}; req.onerror = function () { reject(new Error(req.statusText)); }; req.onabort = function () {
reject(new Error('abort this req')); }; req.send(); }); requestMap[URL] = { promise: promise,
request: req }; return promise; }
```

```
function abortPromise(promise) { if (typeof promise === "undefined") { return; } var request;
Object.keys(requestMap).some(function (URL) { if (requestMap[URL].promise === promise)
{ request = requestMap[URL].request; return true; } }); if (request != null &&
request.readyState !== XMLHttpRequest.UNSENT) { request.abort(); } }
module.exports.createXHRPromise = createXHRPromise; module.exports.abortPromise =
abortPromise; 使用方法也非常简单，我们通过 createXHRPromise 方法得到XHR的promise
对象，当想对这个XHR进行abort操作的时候，将这个promise对象传递给
abortPromise(promise) 方法就可以了。
```

```
var cancelableXHR = require("./cancelableXHR");
```

```
var xhrPromise = cancelableXHR.createXHRPromise('http://httpbin.org/get');
xhrPromise.catch(function (error) { // 调用 abort 抛出的错误 });
cancelableXHR.abortPromise(xhrPromise); 创建包装了XHR的promise对象 取消在1中创建的
promise对象的请求操作
```

总结

在这里我们学到了如下内容。

经过一定时间后变为解决状态的delayPromise

基于delayPromise和Promise.race的超时实现方式

取消XHR promise请求

通过模块化实现promise对象和操作的分离

Promise能非常灵活的进行处理流程的控制，为了充分发挥它的能力，我们需要注意不要将一个函数写的过于庞大冗长，而是应该将其分割成更小更简单的处理，并对之前JavaScript中提到的机制进行更深入的了解。

什么是 `Promise.prototype.done` ？

如果你使用过其他的Promise实现类库的话，可能见过用done代替then的例子。

这些类库都提供了 `Promise.prototype.done` 方法，使用起来也和 `then` 一样，但是这个方法并不会返回promise对象。

虽然 ES6 Promises和Promises/A+等在设计上并没有对`Promise.prototype.done` 做出任何规定，但是很多实现类库都提供了该方法的实现。

在本小节中，我们将会学习什么是 `Promise.prototype.done` ，以及为什么很多类库都提供了对该方法的支持。

使用done的代码示例

看一下实际使用done的代码的例子，应该就非常容易理解 `done` 方法的行为了。

```
promise-done-example.js if (typeof Promise.prototype.done === 'undefined') {
  Promise.prototype.done = function (onFulfilled, onRejected) { this.then(onFulfilled,
    onRejected).catch(function (error) { setTimeout(function () { throw error; }, 0); }); }; } var
  promise = Promise.resolve(); promise.done(function () { JSON.parse('this is not json'); // =>
  SyntaxError: JSON.parse }); // => 请打开浏览器的开发者工具中的控制台窗口看一下 运行 在
  前面我们已经说过，promise设计规格并没有对 Promise.prototype.done做出任何规定，因此
  在使用的时候，你可以使用已有类库提供的实现，也可以自己去实现。
```

我们会在后面讲述如何去自己实现，首先我们这里先对使用 `then` 和使用 `done`这两种方式进行一下比较。

使用then的场景 `var promise = Promise.resolve(); promise.then(function () {
 JSON.parse("this is not json"); }).catch(function (error) { console.error(error); // =>
 "SyntaxError: JSON.parse" });` 运行 从上面我们可以看出，两者之间有以下不同点。

`done` 并不返回promise对象

也就是说，在done之后不能使用 `catch` 等方法组成方法链

`done` 中发生的异常会被直接抛给外面

也就是说，不会进行Promise的错误处理（Error Handling）

由于done 不会返回promise对象，所以我们不难理解它只能出现在一个方法链的最后。

此外，我们已经介绍过了Promise具有强大的错误处理机制，而done则会在函数中跳过错误处理，直接抛出异常。

为什么很多类库都提供了这个和Promise功能相矛盾的函数呢？看一下下面Promise处理失败的例子，也许我们多少就能理解其中原因了吧。

消失的错误

Promise虽然具备了强大的错误处理机制，但是（调试工具不能顺利运行的时候）这个功能会导致人为错误（human error）更加复杂，这也是它的一个缺点。

也许你还记得，我们在 `then or catch?` 中也看到了类似的内容。

像下面那样，我们看一个能返回promise对象的函数。

```
json-promise.js function JSONPromise(value) { return new Promise(function (resolve) {
  resolve(JSON.parse(value)); }); } 这个函数将接收到的参数传递给 JSON.parse，并返回一个
基于JSON.parse的promise对象。
```

我们可以像下面那样使用这个Promise函数，由于 `JSON.parse` 会解析失败并抛出一个异常，该异常会被 `catch` 捕获。

```
function JSONPromise(value) { return new Promise(function (resolve) {
  resolve(JSON.parse(value)); }); } // 运行示例 var string = "非合法json编码字符串";
JSONPromise(string).then(function (object) { console.log(object); }).catch(function(error){ //
=> JSON.parse抛出异常时 console.error(error); }); 运行 如果这个解析失败的异常被正常捕获
的话则没什么问题，但是如果编码时忘记了处理该异常，一旦出现异常，那么查找异常发生
的源头将会变得非常棘手，这就是使用promise需要注意的一面。
```

忘记了使用`catch`进行异常处理的例子 `var string = "非合法json编码字符串";`
`JSONPromise(string).then(function (object) { console.log(object); });` 虽然抛出了异常，但是没有对该异常进行处理 如果是`JSON.parse` 这样比较好找的例子还算好说，如果是拼写错误的话，那么发生了`Syntax Error`错误的话将会非常麻烦。

typo错误 `var string = "{}"; JSONPromise(string).then(function (object) { conosle.log(object); });` 存在`conosle`这个拼写错误 这个例子里，我们错把 `console` 拼成了 `conosle`，因此会发生如下错误。

`ReferenceError: conosle is not defined` 但是，由于Promise的try-catch机制，这个问题可能会被内部消化掉。如果在调用的时候每次都无遗漏的进行 `catch` 处理的话当然最好了，但是在实现的过程中出现了这个例子中的错误的话，那么进行错误排除的工作也会变得困难。

这种错误被内部消化的问题也被称为 `unhandled rejection`，从字面上看就是在`Rejected`时没有找到相应处理的意思。

这种`unhandled rejection`错误到底有多难检查，也依赖于Promise的实现。比如 `ypromise` 在检测到 `unhandled rejection` 错误的时候，会在控制台上提示相应的信息。

Promise rejected but no error handlers were registered to it 另外，Bluebird 在比较明显的人为错误，即ReferenceError等错误的时候，会直接显示到控制台上。

"Possibly unhandled ReferenceError. console is not defined 原生（Native）的 Promise实现为了应对同样问题，提供了GC-based unhandled rejection tracking功能。

该功能是在promise对象被垃圾回收器回收的时候，如果是unhandled rejection的话，则进行错误显示的一种机制。

Firefox 或 Chrome 的原生Promise都进行了部分实现。

done的实现

作为方法论，在Promise中 done 是怎么解决上面提到的错误被忽略呢？其实它的方法很简单直接，那就是必须要进行错误处理。

由于可以在 Promise上实现 done 方法，因此我们看看如何对 Promise.prototype.done 这个 Promise的prototype进行扩展。

promise-prototype-done.js "use strict"; if (typeof Promise.prototype.done === "undefined") { Promise.prototype.done = function (onFulfilled, onRejected) { this.then(onFulfilled, onRejected).catch(function (error) { setTimeout(function () { throw error; }, 0); }); }; } 那么它是如何将异常抛到Promise的外面的呢？其实这里我们利用的是在setTimeout中使用throw方法，直接将异常抛给了外部。

setTimeout的回调函数中抛出异常 try{ setTimeout(function callback() { throw new Error("error"); }, 0); }catch(error){ console.error(error); } 这个例外不会被捕获 关于为什么异步的callback中抛出的异常不会被捕获的原因，可以参考下面内容。

JavaScript和异步错误处理 - Yahoo! JAPAN Tech Blog（日语博客）

仔细看一下 Promise.prototype.done的代码，我们会发现这个函数什么也没 return 。也就是说， done按照「Promise chain在这里将会中断，如果出现了异常，直接抛到promise外面即可」的原则进行了处理。

如果实现和运行环境实现的比较完美的话，就可以进行 unhandled rejection 检测，done也不一定是必须的了。另外像本小节中的 Promise.prototype.done一样，done也可以在既有的 Promise之上进行实现，也可以说它没有进入到 ES6 Promises的设计规范之中。

本文中的 Promise.prototype.done 的实现方法参考了 promisejs.org 。

总结

在本小节中，我们学习了 Q、Bluebird 和 pfun 等Promise类库提供的 done 的基础和实现细节，以及done方法和 then 方法有什么区别等内容。

我们也学到了 `done` 有以下两个特点。

`done` 中出现的错误会被作为异常抛出

终结 Promise chain

和 `then` or `catch`? 中说到的一样，由Promise内部消化掉的错误，随着调试工具或者类库的改进，大多数情况下也许已经不是特别大的问题了。

此外，由于 `done` 不会有返回值，因此不能在它之后进行方法链的创建，为了实现Promise方法风格上的统一，我们也可以使用`done`方法。

ES6 Promises 本身提供的功能并不是特别多。因此，我想很多时候可能需要我们自己进行扩展或者使用第三方类库。

我们好不容易将异步处理统一采用Promise进行统一处理，但是如果做过头了，也会将系统变得特别复杂，因此，保持风格的统一是Promise作为抽象对象非常重要的部分。

在 Promises: The Extension Problem (part 4) | getiblog 中，介绍了一些如何编写Promise扩展程序的方法。

扩展 `Promise.prototype` 的方法

利用 `Wrapper/Delegate` 创建抽象层

此外，关于 `Delegate` 的详细使用方法，也可以参考 Chapter 28. Subclassing Built-ins ， 那里有详细的说明。

Promise和方法链（method chain）

在Promise中你可以将 `then` 和 `catch` 等方法连在一起写。这非常像DOM或者jQuery中的方法链。

一般的方法链都通过返回 `this` 将多个方法串联起来。

关于如何创建方法链，可以从参考 方法链的创建方法 - 余味（日语博客）等资料。另一方面，由于Promise 每次都会返回一个新的promise对象，所以从表面上看和一般的方法链几乎一模一样。

在本小节里，我们会在不改变已有采用方法链编写的代码的外部接口的前提下，学习如何在内部使用Promise进行重写。

fs中的方法链

我们下面将会以 Node.js中的fs 为例进行说明。

此外，这里的例子我们更重视代码的易理解性，因此从实际上来说这个例子可能并不算太实用。

```
fs-method-chain.js "use strict"; var fs = require("fs"); function File() { this.lastValue = null; } // Static method for File.prototype.read File.read = function FileRead(filePath) { var file = new File(); return file.read(filePath); }; File.prototype.read = function (filePath) { this.lastValue = fs.readFileSync(filePath, "utf-8"); return this; }; File.prototype.transform = function (fn) { this.lastValue = fn.call(this, this.lastValue); return this; }; File.prototype.write = function (filePath) { this.lastValue = fs.writeFileSync(filePath, this.lastValue); return this; }; module.exports = File; 这个模块可以将类似下面的 read → transform → write 这一系列处理，通过组成一个方法链来实现。
```

```
var File = require("./fs-method-chain"); var inputFilePath = "input.txt", outputFilePath = "output.txt"; File.read(inputFilePath) .transform(function (content) { return ">>" + content; }) .write(outputFilePath); transform 接收一个方法作为参数，该方法对其输入参数进行处理。在这个例子里，我们对通过read读取的数据在前面加上了 >> 字符串。
```

基于Promise的fs方法链

下面我们就在不改变刚才的方法链对外接口的前提下，采用Promise对内部实现进行重写。

```
fs-promise-chain.js "use strict"; var fs = require("fs"); function File() { this.promise = Promise.resolve(); } // Static method for File.prototype.read File.read = function (filePath) { var file = new File(); return file.read(filePath); };
```

```
File.prototype.then = function (onFulfilled, onRejected) { this.promise =
this.promise.then(onFulfilled, onRejected); return this; }; File.prototype["catch"] = function
(onRejected) { this.promise = this.promise.catch(onRejected); return this; };
File.prototype.read = function (filePath) { return this.then(function () { return
fs.readFileSync(filePath, "utf-8"); }); }; File.prototype.transform = function (fn) { return
this.then(fn); }; File.prototype.write = function (filePath) { return this.then(function (data) {
return fs.writeFileSync(filePath, data); }); }; module.exports = File; 新增加的then 和catch都可以看做是指向内部保存的promise对象的别名，而其它部分从对外接口的角度来说都没有改变，使用方法也和原来一样。
```

因此，在使用这个模块的时候我们只需要修改 require 的模块名即可。

```
var File = require("./fs-promise-chain"); var inputFilePath = "input.txt", outputFilePath =
"output.txt"; File.read(inputFilePath) .transform(function (content) { return ">>" + content; })
.write(outputFilePath); File.prototype.then 方法会调用 this.promise.then 方法，并将返回的
promise对象赋值给了 this.promise 变量这个内部promise对象。
```

这究竟有什么奥妙么？通过以下的伪代码，我们可以更容易理解这背后发生的事情。

```
var File = require("./fs-promise-chain"); File.read(inputFilePath) .transform(function (content)
{ return ">>" + content; }) .write(outputFilePath); // => 处理流程类似以下的伪代码
promise.then(function read(){ return fs.readFileSync(filePath, "utf-8"); }).then(function
transform(content) { return ">>" + content; }).then(function write(){ return
fs.writeFileSync(filePath, data); }); 看到 promise = promise.then(...) 这种写法，会让人以为
promise的值会被覆盖，也许你会想是不是promise的chain被截断了。
```

你可以想象为类似 `promise = addPromiseChain(promise, fn);` 这样的感觉，我们为promise对象增加了新的处理，并返回了这个对象，因此即使自己不实现顺序处理的话也不会带来什么问题。

两者的区别

同步和异步

要说fs-method-chain.js和Promise版两者之间的差别，最大的不同那就要算是同步和异步了。

如果在类似 fs-method-chain.js 的方法链中加入队列等处理的话，就可以实现几乎和异步方法链同样的功能，但是实现将会变得非常复杂，所以我们选择了简单的同步方法链。

Promise版的话如同在 专栏: Promise只能进行异步处理？ 里介绍过的一样，只会进行异步操作，因此使用了promise的方法链也是异步的。

错误处理

虽然fs-method-chain.js里面并不包含错误处理的逻辑，但是由于是同步操作，因此可以将整段代码用 try-catch 包起来。

在 Promise版 提供了指向内部promise对象的then 和 catch 别名，所以我们可以像其它 promise对象一样使用catch来进行错误处理。

fs-promise-chain中的错误处理 `var File = require("./fs-promise-chain");`
`File.read(inputFilePath).transform(function (content) { return ">>" + content; })`
`.write(outputFilePath).catch(function(error){ console.error(error); });` 如果你想在fs-method-chain.js中自己实现异步处理的话，错误处理可能会成为比较大的问题；可以说在进行异步处理的时候，还是使用Promise实现起来比较简单。

Promise之外的异步处理

如果你很熟悉Node.js的话，那么看到方法链的话，你是不是会想起来 Stream 呢。

如果使用 Stream 的话，就可以免去了保存 this.lastValue 的麻烦，还能改善处理大文件时候的性能。另外，使用Stream的话可能会比使用Promise在处理速度上会快些。

使用Stream进行read→transform→write

`readableStream.pipe(transformStream).pipe(writableStream);` 因此，在异步处理的时候并不是说Promise永远都是最好的选择，要根据自己的目的和实际情况选择合适的实现方式。

Node.js的Stream是一种基于Event的技术 关于Node.js中Stream的详细信息可以参考以下网页。

利用Node.js Stream API对数据进行流式处理 - Block Rockin' Codes

Stream2基础

关于Node-v0.12新功能

Promise wrapper

再回到 fs-method-chain.js 和 Promise版，这两种方法相比较内部实现也非常相近，让人觉得是不是同步版本的代码可以直接就当做异步方式来使用呢？

由于JavaScript可以向对象动态添加方法，所以从理论上来说应该可以从非Promise版自动生成Promise版的代码。（当然静态定义的实现方式容易处理）

尽管 ES6 Promises 并没有提供此功能，但是著名的第三方Promise实现类库 bluebird 等提供了被称为 Promisification 的功能。

如果使用类似这样的类库，那么就可以动态给对象增加promise版的方法。

```
var fs = Promise.promisifyAll(require("fs"));
```

```
fs.readFileAsync("myfile.js", "utf8").then(function(contents){ console.log(contents);
}).catch(function(e){ console.error(e.stack); });
```

 Array的Promise wrapper

前面的 Promisification 具体都干了些什么光凭想象恐怕不太容易理解，我们可以通过给原生的 Array 增加Promise版的方法为例来进行说明。

在JavaScript中原生DOM或String等也提供了很多创建方法链的功能。Array 中就有诸如 map 和 filter 等方法，这些方法会返回一个数组类型，可以用这些方法方便的组建方法链。

```
array-promise-chain.js "use strict"; function ArrayAsPromise(array) { this.array = array;
this.promise = Promise.resolve(); } ArrayAsPromise.prototype.then = function (onFulfilled,
onRejected) { this.promise = this.promise.then(onFulfilled, onRejected); return this; };
ArrayAsPromise.prototype["catch"] = function (onRejected) { this.promise =
this.promise.catch(onRejected); return this; };
Object.getOwnPropertyNames(Array.prototype).forEach(function (methodName) { // Don't
overwrite if (typeof ArrayAsPromise[methodName] !== "undefined") { return; } var
arrayMethod = Array.prototype[methodName]; if (typeof arrayMethod !== "function") { return;
} ArrayAsPromise.prototype[methodName] = function () { var that = this; var args =
arguments; this.promise = this.promise.then(function () { that.array =
Array.prototype[methodName].apply(that.array, args); return that.array; }); return this; });
```

```
module.exports = ArrayAsPromise; module.exports.array = function
newArrayAsPromise(array) { return new ArrayAsPromise(array); };
```

 原生的 Array 和 ArrayAsPromise 在使用时有什么差异呢？我们可以通过对 上面的代码 进行测试来了解它们之间的不同点。

```
array-promise-chain-test.js "use strict"; var assert = require("power-assert"); var
ArrayAsPromise = require("../src/promise-chain/array-promise-chain"); describe("array-
promise-chain", function () { function isEven(value) { return value % 2 === 0; }
```

```
function double(value) {
  return value * 2;
}

beforeEach(function () {
  this.array = [1, 2, 3, 4, 5];
});
describe("Native array", function () {
  it("can method chain", function () {
    var result = this.array.filter(isEven).map(double);
    assert.deepEqual(result, [4, 8]);
  });
});
describe("ArrayAsPromise", function () {
  it("can promise chain", function (done) {
    var array = new ArrayAsPromise(this.array);
    array.filter(isEven).map(double).then(function (value) {
      assert.deepEqual(value, [4, 8]);
    }).then(done, done);
  });
});
```

}); 我们看到，在 `ArrayAsPromise` 中也能使用 `Array` 的方法。而且也和前面的例子类似，原生的 `Array` 是同步处理，而 `ArrayAsPromise` 则是异步处理，这也是它们的不同之处。

仔细看一下 `ArrayAsPromise` 的实现，也许你已经注意到了，`Array.prototype` 的所有方法都被实现了。但是，`Array.prototype` 中也存在着类似 `array.indexOf` 等并不会返回数组类型数据的方法，这些方法如果也要支持方法链的话就有些不自然了。

在这里非常重要的一点是，我们可以通过这种方式，为具有接收相同类型数据接口的API动态的创建Promise版的API。如果我们能意识到这种API的规则性的话，那么就可能发现一些新的使用方法。

前面我们看到的 `Promisification` 方法，借鉴了 `Node.js` 的 `Core` 模块中在进行异步处理时将 `function(error,result){}` 方法的第一个参数设为 `error` 这一规则，自动的创建由 `Promise` 包装好的方法。

总结

在本小节我们主要学习了下面的这些内容。

Promise版的方法链实现

Promise并不是总是异步编程的最佳选择

Promisification

统一接口的重用

ES6 Promises只提供了一些Core级别的功能。因此，我们也许需要对现有的方法用Promise方式重新包装一下。

但是，类似Event等调用次数没有限制的回调函数等在并不适合使用Promise，Promise也不能说什么时候都是最好的选择。

至于什么情况下应该使用Promise，什么时候不该使用Promise，并不是本书要讨论的目的，我们需要牢记的是不要什么都用Promise去实现，我想最好根据自己的具体目的和情况，来考虑是应该使用Promise还是其它方法。

使用Promise进行顺序（sequence）处理

在第2章 Promise.all 中，我们已经学习了如何让多个promise对象同时开始执行的方法。

但是 Promise.all 方法会同时运行多个promise对象，如果想进行在A处理完成之后再开始B的处理，对于这种顺序执行的话 Promise.all就无能为力了。

此外，在同一章的Promise和数组 中，我们也介绍了一种效率不是特别高的，使用了 重复使用多个then的方法 来实现如何按顺序进行处理。

在本小节中，我们将对如何在Promise中进行顺序处理进行介绍。

循环和顺序处理

在 重复使用多个then的方法 中的实现方法如下。

```
function getURL(URL) { return new Promise(function (resolve, reject) { var req = new
XMLHttpRequest(); req.open('GET', URL, true); req.onload = function () { if (req.status ===
200) { resolve(req.responseText); } else { reject(new Error(req.statusText)); } }; req.onerror =
function () { reject(new Error(req.statusText)); }; req.send(); }); } var request = { comment:
function getComment() { return getURL('http://azu.github.io/promises-
book/json/comment.json').then(JSON.parse); }, people: function getPeople() { return
getURL('http://azu.github.io/promises-book/json/people.json').then(JSON.parse); } }; function
main() { function recordValue(results, value) { results.push(value); return results; } // [] 用来保
存初始化的值 var pushValue = recordValue.bind(null, []); return
request.comment().then(pushValue).then(request.people).then(pushValue); } // 运行示例
main().then(function (value) { console.log(value); }).catch(function(error){
console.error(error); }); 运行 使用这种写法的话那么随着 request 中元素数量的增加，我们也
需要不断增加对 then 方法的调用
```

因此，如果我们将处理内容统一放到数组里，再配合for循环进行处理的话，那么处理内容的增加将不会再带来什么问题。首先我们就使用for循环来完成和前面同样的处理。

```
promise-foreach-xhr.js function getURL(URL) { return new Promise(function (resolve, reject)
{ var req = new XMLHttpRequest(); req.open('GET', URL, true); req.onload = function () { if
(req.status === 200) { resolve(req.responseText); } else { reject(new Error(req.statusText)); }
}; req.onerror = function () { reject(new Error(req.statusText)); }; req.send(); }); } var request =
{ comment: function getComment() { return getURL('http://azu.github.io/promises-
book/json/comment.json').then(JSON.parse); }, people: function getPeople() { return
getURL('http://azu.github.io/promises-book/json/people.json').then(JSON.parse); } }; function
main() { function recordValue(results, value) { results.push(value); return results; } // [] 用来保
```

存初始化值 `var pushValue = recordValue.bind(null, []); // 返回promise对象的函数的数组` `var tasks = [request.comment, request.people]; var promise = Promise.resolve(); // 开始的地方` `for (var i = 0; i < tasks.length; i++) { var task = tasks[i]; promise = promise.then(task).then(pushValue); } return promise; } // 运行示例` `main().then(function (value) { console.log(value); }).catch(function(error){ console.error(error); });` 运行使用for循环的时候，如同我们在 专栏: 每次调用then都会返回一个新创建的promise对象 以及 Promise 和方法链 中学到的那样，每次调用 `Promise#then` 方法都会返回一个新的promise对象。

因此类似 `promise = promise.then(task).then(pushValue);` 的代码就是通过不断对promise进行处理，不断的覆盖 `promise` 变量的值，以达到对promise对象的累积处理效果。

但是这种方法需要 `promise` 这个临时变量，从代码质量上来说显得不那么简洁。

如果将这种循环写法改用 `Array.prototype.reduce` 的话，那么代码就会变得聪明多了。

Promise chain和reduce

如果将上面的代码用 `Array.prototype.reduce` 重写的话，会像下面一样。

```
promise-reduce-xhr.js function getURL(URL) { return new Promise(function (resolve, reject) {
var req = new XMLHttpRequest(); req.open('GET', URL, true); req.onload = function () { if
(req.status === 200) { resolve(req.responseText); } else { reject(new Error(req.statusText)); }
}; req.onerror = function () { reject(new Error(req.statusText)); }; req.send(); }); } var request =
{ comment: function getComment() { return getURL('http://azu.github.io/promises-
book/json/comment.json').then(JSON.parse); }, people: function getPeople() { return
getURL('http://azu.github.io/promises-book/json/people.json').then(JSON.parse); } }; function
main() { function recordValue(results, value) { results.push(value); return results; } var
pushValue = recordValue.bind(null, []); var tasks = [request.comment, request.people]; return
tasks.reduce(function (promise, task) { return promise.then(task).then(pushValue); },
Promise.resolve()); } // 运行示例 main().then(function (value) { console.log(value);
}).catch(function(error){ console.error(error); });
```

 运行 这段代码中除了 `main` 函数之外的其他处理都和使用for循环的时候相同。

`Array.prototype.reduce` 的第二个参数用来设置盛放计算结果的初始值。在这个例子中，`Promise.resolve()` 会赋值给 `promise`，此时的 `task` 为 `request.comment`。

在`reduce`中第一个参数中被 `return` 的值，则会被赋值为下次循环时的 `promise`。也就是说，通过返回由 `then` 创建的新的promise对象，就实现了和for循环类似的 `Promise chain` 了。

下面是关于 `Array.prototype.reduce` 的详细说明。

`Array.prototype.reduce()` - JavaScript | MDN

azu / `Array.prototype.reduce` Dance - Glide

使用reduce和for循环不同的地方是reduce不再需要临时变量 promise 了，因此也不用编写 `promise = promise.then(task).then(pushValue)`；这样冗长的代码了，这是非常大的进步。

虽然 `Array.prototype.reduce` 非常适合用来在Promise中进行顺序处理，但是上面的代码有可能让人难以理解它是如何工作的。

因此我们再来编写一个名为 `sequenceTasks` 的函数，它接收一个数组作为参数，数组里面存放的是要进行的处理Task。

从下面的调用代码中我们可以非常容易的从其函数名想到，该函数的功能是对 `tasks` 中的处理进行顺序执行了。

```
var tasks = [request.comment, request.people]; sequenceTasks(tasks);
```

定义进行顺序处理的函数

基本上我们只需要基于 使用reduce的方法 重构出一个函数。

```
promise-sequence.js function sequenceTasks(tasks) { function recordValue(results, value) {
results.push(value); return results; } var pushValue = recordValue.bind(null, []); return
tasks.reduce(function (promise, task) { return promise.then(task).then(pushValue); },
Promise.resolve()); } 需要注意的一点是，和 Promise.all 等不同，这个函数接收的参数是一个
函数的数组。
```

为什么传给这个函数的不是一个promise对象的数组呢？这是因为promise对象创建的时候，XHR已经开始执行了，因此再对这些promise对象进行顺序处理的话就不能正常工作了。

因此 `sequenceTasks` 将函数(该函数返回一个promise对象)的数组作为参数。

最后，使用 `sequenceTasks` 重写最开始的例子的话，如下所示。

```
promise-sequence-xhr.js function sequenceTasks(tasks) { function recordValue(results,
value) { results.push(value); return results; } var pushValue = recordValue.bind(null, []);
return tasks.reduce(function (promise, task) { return promise.then(task).then(pushValue); },
Promise.resolve()); } function getURL(URL) { return new Promise(function (resolve, reject) {
var req = new XMLHttpRequest(); req.open('GET', URL, true); req.onload = function () { if
(req.status === 200) { resolve(req.responseText); } else { reject(new Error(req.statusText)); }
}; req.onerror = function () { reject(new Error(req.statusText)); }; req.send(); }); } var request =
{ comment: function getComment() { return getURL('http://azu.github.io/promises-
book/json/comment.json').then(JSON.parse); }, people: function getPeople() { return
getURL('http://azu.github.io/promises-book/json/people.json').then(JSON.parse); } }; function
main() { return sequenceTasks([request.comment, request.people]); } // 运行示例
main().then(function (value) { console.log(value); }).catch(function(error){
console.error(error); }); 运行 怎样， main() 中的流程是不是更清晰易懂了。
```

如上所述，在Promise中，我们可以选择多种方法来实现处理的按顺序执行。

循环使用then调用的方法

使用for循环的方法

使用reduce的方法

分离出顺序处理函数的方法

但是，这些方法都是基于JavaScript中对数组及进行操作的for循环或forEach等，本质上并无大区别。因此从一定程度上来说，在处理Promise的时候，将大块的处理分成小函数来实现是一个非常好的实践。

总结

在本小节中，我们对如何在Promise中进行和Promise.all相反，按顺序让promise一个个进行处理的实现方式进行了介绍。

为了实现顺序处理，我们也对从过程风格的编码方式到自定义顺序处理函数的方式等实现方式进行了介绍，也再次强调了在Promise领域我们应遵循将处理按照函数进行划分的基本原则。

在Promise中如果还使用了Promise chain将多个处理连接起来的话，那么还可能使源代码中的一条语句变得很长。

这时候如果我们回想一下这些编程的基本原则进行函数拆分的话，代码整体结构会变得非常清晰。

此外,Promise的构造函数以及then都是高阶函数，如果将处理分割为函数的话，还能得到对函数进行灵活组合使用的副作用，意识到这一点对我们也会有一些帮助的。

高阶函数指的是一个函数可以接受其参数为函数对象的实例

Promises API Reference

Promise#then

```
promise.then(onFulfilled, onRejected);
```

then 代码示例

```
var promise = new Promise(function(resolve, reject){
    resolve("传递给then的值");
});
promise.then(function (value) {
    console.log(value);
}, function (error) {
    console.error(error);
});
```

运行

这段代码创建一个promise对象，定义了处理onFulfilled和onRejected的函数（handler），然后返回这个promise对象。

这个promise对象会在变为resolve或者reject的时候分别调用相应注册的回调函数。

当handler返回一个正常值的时候，这个值会传递给promise对象的onFulfilled方法。

定义的handler中产生异常的时候，这个值则会传递给promise对象的onRejected方法。

Promise#catch

```
promise.catch(onRejected);
```

catch 代码示例

```
var promise = new Promise(function(resolve, reject){
  resolve("传递给then的值");
});
promise.then(function (value) {
  console.log(value);
}).catch(function (error) {
  console.error(error);
});
```

运行 这是一个等价于`promise.then(undefined, onRejected)`的语法糖。

Promise.resolve

```
Promise.resolve(promise);
Promise.resolve(thenable);
Promise.resolve(object);
```

Promise.resolve代码示例

```
var taskName = "task 1"
asyncTask(taskName).then(function (value) {
  console.log(value);
}).catch(function (error) {
  console.error(error);
});
function asyncTask(name){
  return Promise.resolve(name).then(function(value){
    return "Done! " + value;
  });
}
```

运行 根据接收到的参数不同，返回不同的promise对象。

虽然每种情况都会返回promise对象，但是大体来说主要分为下面3类。

接收到promise对象参数的时候 返回的还是接收到的promise对象

接收到thenable类型的对象的时候 返回一个新的promise对象，这个对象具有一个 `then` 方法

接收的参数为其他类型的时候（包括JavaScript对或null等） 返回一个将该对象作为值的新promise对象

Promise.reject

```
Promise.reject(object)
```

Promise.reject代码示例

```
var failureStub = sinon.stub(xhr, "request").returns(Promise.reject(new Error("bad!")));
```

返回一个使用接收到的值进行了reject的新的promise对象。

而传给Promise.reject的值也应该是一个 Error 类型的对象。

另外，和 Promise.resolve不同的是，即使Promise.reject接收到的参数是一个promise对象，该函数也还是会返回一个全新的promise对象。

```
var r = Promise.reject(new Error("error"));
console.log(r === Promise.reject(r)); // false
```

运行

Promise.all

```
Promise.all(promiseArray);
```

Promise.all代码示例

```
var p1 = Promise.resolve(1),
    p2 = Promise.resolve(2),
    p3 = Promise.resolve(3);
Promise.all([p1, p2, p3]).then(function (results) {
  console.log(results); // [1, 2, 3]
});
```

运行 生成并返回一个新的promise对象。

参数传递promise数组中所有的promise对象都变为resolve的时候，该方法才会返回，新创建的promise则会使用这些promise的值。

如果参数中的任何一个promise为reject的话，则整个Promise.all调用会立即终止，并返回一个reject的新的promise对象。

由于参数数组中的每个元素都是由 Promise.resolve 包装（wrap）的，所以Promise.all可以处理不同类型的promise对象。

Promise.race

```
Promise.race(promiseArray);
```

Promise.race代码示例

```
var p1 = Promise.resolve(1),  
    p2 = Promise.resolve(2),  
    p3 = Promise.resolve(3);  
Promise.race([p1, p2, p3]).then(function (value) {  
    console.log(value); // 1  
});
```

运行 生成并返回一个新的promise对象。

参数 promise 数组中的任何一个promise对象如果变为resolve或者reject的话，该函数就会返回，并使用这个promise对象的值进行resolve或者reject。

6. 用語集

Promises

Promise规范自身

promise对象

promise对象指的是 Promise 实例对象

ES6 Promises

如果想明确表示使用 ECMAScript 6th Edition 的话，可以使用ES6作为前缀（prefix）

Promises/A+

Promises/A+。这是ES6 Promises的前身，是一个社区规范，它和 ES6 Promises 有很多共通的内容。

Thenable

类Promise对象。拥有名为.then方法的对象。

promise chain

指使用 then 或者 catch 方法将promise对象连接起来的行为。此用语只是在本书中的说法，而不是在 ES6 Promises 中定义的官方用语。

7. 参考网站

[w3ctag/promises-guide](#) Promises指南 - 这里有很多关于概念方面的说明

[domenic/promises-unwrapping](#) ES6 Promises规范的repo - 可以通过查看issue来了解各种关于规范的来龙去脉和信息

[ECMAScript Language Specification ECMA-262 6th Edition – DRAFT](#) ES6 Promises的规范 - 如果想参考关于ES6 Promises的规范，则应该先看这里

[JavaScript Promises: There and back again - HTML5 Rocks](#) 关于Promises的文章 - 这里的示例代码和参考（reference）的完成度都很高

[Node.js Promise再次降临！ - ぼちぼち日記](#) 关于Node.js和Promise的文章 - thenable部分参考了本文

8. 关于作者

[azu](#) (Twitter : [@azu_re](#))

关注浏览器、JavaScript相关的最新技术。

擅长将目的作为手段，本书也是因此而成。

管理着个人主页 [Web Scratch](#) 和 [JSer.info](#) 。

关于译者

- liubin <https://github.com/liubin>
 - 除去kakau和honkkyou的其余部分的翻译、整体校对，以及源代码，工具部分的翻译
- kaku <https://github.com/kaku87>
 - 1.1. Promise是什么、1.2. Promise 简介、1.3. 编写Promise代码
- honkkyou <https://github.com/honkkyou>
 - 3.1. 基本测试

给原著者留言、后记

后记.pdf 里面记录了笔者为什么要写这么一本书，编写的过程，以及如何进行测试。

- 下载后记（日文版） JavaScript Promise迷你书后记（日文版）

你可以在Gumroad以免费的价格或者自己设定一个任意的价格来下载本书的后记。

在下载的时候，会有一个给作者留言的地方， 希望各位读者能写下一点什么之后下载。

如果本书有任何问题的话，也可以通过 GitHub或者Gitter 来提交。

- Issues · azu/promises-book
- azu/promises-book - Gitter