但是，第一个参数就有一些模糊了，Promise 文献通常将其称为 resolve(..)。这个词显然和决议（resolution）有关，而决议在各种文献（包括本书）中是用来描述“为 Promise 设定最终值 / 状态”。前面我们已经多次使用“Promise 决议”来表示完成或拒绝 Promise。

但是，如果这个参数是用来特指完成这个 Promise，那为什么不用使用 fulfill(..) 来代替 resolve(..) 以求表达更精确呢？要回答这个问题，我们先来看看两个 Promise API 方法：

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

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

Promise.resolve(..) 创建了一个决议为输入值的 Promise。在这个例子中，42 是一个非 Promise、非 thenable 的普通值，所以完成后的 promise fullfilledPr 是为值 42 创建的。Promise.reject("Oops") 创建了一个被拒绝的 promise rejectedPr，拒绝理由为 "Oops"。

现在我们来解释为什么单词 resolve（比如在 Promise.resolve(..) 中）如果用于表达结果可能是完成也可能是拒绝的话，既没有歧义，而且也确实更精确：

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

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

本章前面已经介绍过，Promise.resolve(..) 会将传入的真正 Promise 直接返回，对传入的 thenable 则会展开。如果这个 thenable 展开得到一个拒绝状态，那么从 Promise.resolve(..) 返回的 Promise 实际上就是这同一个拒绝状态。

所以对这个 API 方法来说，Promise.resolve(..) 是一个精确的好名字，因为它实际上的结果可能是完成或拒绝。

Promise(..) 构造器的第一个参数回调会展开 thenable（和 Promise.resolve(..) 一样）或真正的 Promise：

```
var rejectedPr = new Promise( function(resolve,reject){
    // 用一个被拒绝的promise完成这个promise
    resolve( Promise.reject( "Oops" ) );
} );

rejectedPr.then(
    function fulfilled(){
        // 永远不会到达这里
    },
    function rejected(err){
        console.log( err ); // "Oops"
```

```
    }
);
```

现在应该很清楚了，Promise(..) 构造器的第一个回调参数的恰当称谓是 resolve(..)。

前面提到的 reject(..) 不会像 resolve(..) 一样进行展开。如果向 reject(..) 传入一个 Promise/thenable 值，它会把这个值原封不动地设置为拒绝理由。后续的拒绝处理函数接收到的是你实际传给 reject(..) 的那个 Promise/thenable，而不是其底层的立即值。

不过，现在我们再来关注一下提供给 then(..) 的回调。它们（在文献和代码中）应该怎么命名呢？我的建议是 fulfilled(..) 和 rejected(..)：

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

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

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

对 then(..) 的第一个参数来说，毫无疑义，总是处理完成的情况，所以不需要使用标识两种状态的术语“resolve”。这里提一下，ES6 规范将这两个回调命名为 onFulfilled(..) 和 onRjected(..)，所以这两个术语很准确。

## 3.5 错误处理

前面已经展示了一些例子，用于说明在异步编程中 Promise 拒绝（调用 reject(..) 有意拒绝或 JavaScript 异常导致的无意拒绝）如何使得错误处理更完善。我们来回顾一下，并明确解释一下前面没有明说的几个细节。

对多数开发者来说，错误处理最自然的形式就是同步的 try..catch 结构。遗憾的是，它只能是同步的，无法用于异步代码模式：

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

try {
```

```
        foo();
        // 后面从 `baz.bar()` 抛出全局错误
    }
    catch (err) {
        // 永远不会到达这里
    }
```

try..catch 当然很好，但是无法跨异步操作工作。也就是说，还需要一些额外的环境支持，我们会在第 4 章关于生成器的部分介绍这些环境支持。

在回调中，一些模式化的错误处理方式已经出现，最值得一提的是 error-first 回调风格：

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

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

只有在 baz.bar() 调用会同步地立即成功或失败的情况下，这里的 try..catch 才能工作。如果 baz.bar() 本身有自己的异步完成函数，其中的任何异步错误都将无法捕捉到。

传给 foo(..) 的回调函数保留第一个参数 err，用于在出错时接收到信号。如果其存在的话，就认为出错；否则就认为是成功。

严格说来，这一类错误处理是支持异步的，但完全无法很好地组合。多级 error-first 回调交织在一起，再加上这些无所不在的 if 检查语句，都不可避免地导致了回调地狱的风险（参见第 2 章）。

我们回到 Promise 中的错误处理，其中拒绝处理函数被传递给 then(..)。Promise 没有采用流行的 error-first 回调设计风格，而是使用了分离回调（split-callback）风格。一个回调用于完成情况，一个回调用于拒绝情况：

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

p.then(
    function fulfilled(){
        // 永远不会到达这里
    },
    function rejected(err){
        console.log( err ); // "Oops"
    }
);
```

尽管表面看来，这种出错处理模式很合理，但彻底掌握 Promise 错误处理的各种细微差别常常还是有些难度的。

考虑：

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

p.then(
    function fulfilled(msg){
        // 数字没有string函数,所以会抛出错误
        console.log( msg.toLowerCase() );
    },
    function rejected(err){
        // 永远不会到达这里
    }
);
```

如果 `msg.toLowerCase()` 合法地抛出一个错误（事实确实如此！），为什么我们的错误处理函数没有得到通知呢？正如前面解释过的，这是因为那个错误处理函数是为 promise `p` 准备的，而这个 promise 已经用值 `42` 填充了。promise `p` 是不可变的，所以唯一可以被通知这个错误的 promise 是从 `p.then(..)` 返回的那一个，但我们在此例中没有捕捉。

这应该清晰地解释了为什么 Promise 的错误处理易于出错。这非常容易造成错误被吞掉，而这极少是出于你的本意。

如果通过无效的方式使用 Promise API，并且出现一个错误阻碍了正常的 Promise 构造，那么结果会得到一个立即抛出的异常，而不是一个被拒绝的 Promise。这里是一些错误使用导致 Promise 构造失败的例子：`new Promise(null)`、`Promise.all()`、`Promise.race(42)`，等等。如果一开始你就没能有效使用 Promise API 真正构造出一个 Promise，那就无法得到一个被拒绝的 Promise！

## 3.5.1 绝望的陷阱

Jeff Atwood 多年前曾提出：通常编程语言构建的方式是，默认情况下，开发者陷入“绝

望的陷阱”（pit of despair）（http://blog.codinghorror.com/falling-into-the-pit-of-success），要为错误付出代价，只有更努力才能做对。他呼吁我们转而构建一个“成功的坑”（pit of success），其中默认情况下你能够得到想要的结果（成功），想出错很难。

毫无疑问，Promise 错误处理就是一个“绝望的陷阱”设计。默认情况下，它假定你想要 Promise 状态吞掉所有的错误。如果你忘了查看这个状态，这个错误就会默默地（通常是绝望地）在暗处凋零死掉。

为了避免丢失被忽略和抛弃的 Promise 错误，一些开发者表示，Promise 链的一个最佳实践就是最后总以一个 `catch(..)` 结束，比如：

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

p.then(
    function fulfilled(msg){
        // 数字没有string函数,所以会抛出错误
        console.log( msg.toLowerCase() );
    }
)
.catch( handleErrors );
```

因为我们没有为 `then(..)` 传入拒绝处理函数，所以默认的处理函数被替换掉了，而这仅仅是把错误传递给了链中的下一个 promise。因此，进入 `p` 的错误以及 `p` 之后进入其决议（就像 `msg.toLowerCase()`）的错误都会传递到最后的 `handleErrors(..)`。

问题解决了，对吧？没那么快！

如果 `handleErrors(..)` 本身内部也有错误怎么办呢？谁来捕捉它？还有一个没人处理的 promise：`catch(..)` 返回的那一个。我们没有捕获这个 promise 的结果，也没有为其注册拒绝处理函数。

你并不能简单地在这个链尾端添加一个新的 `catch(..)`，因为它很可能会失败。任何 Promise 链的最后一步，不管是什么，总是存在着在未被查看的 Promise 中出现未捕获错误的可能性，尽管这种可能性越来越低。

看起来好像是个无解的问题吧？

## 3.5.2 处理未捕获的情况

这不是一个容易彻底解决的问题。还有其他（很多人认为是更好的）一些处理方法。

有些 Promise 库增加了一些方法，用于注册一个类似于“全局未处理拒绝”处理函数的东西，这样就不会抛出全局错误，而是调用这个函数。但它们辨识未捕获错误的方法是定义一个某个时长的定时器，比如 3 秒钟，在拒绝的时刻启动。如果 Promise 被拒绝，而在定

时器触发之前都没有错误处理函数被注册，那它就会假定你不会注册处理函数，进而就是未被捕获错误。

在实际使用中，对很多库来说，这种方法运行良好，因为通常多数使用模式在 Promise 拒绝和检查拒绝结果之间不会有很长的延迟。但是这种模式可能会有些麻烦，因为 3 秒这个时间太随意了（即使是经验值），也因为确实有一些情况下会需要 Promise 在一段不确定的时间内保持其拒绝状态。而且你绝对不希望因为这些误报（还没被处理的未捕获错误）而调用未捕获错误处理函数。

更常见的一种看法是：Promsie 应该添加一个 `done(..)` 函数，从本质上标识 Promsie 链的结束。`done(..)` 不会创建和返回 Promise，所以传递给 `done(..)` 的回调显然不会报告一个并不存在的链接 Promise 的问题。

那么会发生什么呢？它的处理方式类似于你可能对未捕获错误通常期望的处理方式：`done(..)` 拒绝处理函数内部的任何异常都会被作为一个全局未处理错误抛出（基本上是在开发者终端上）。代码如下：

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

p.then(
    function fulfilled(msg){
        // 数字没有string函数,所以会抛出错误
        console.log( msg.toLowerCase() );
    }
)
.done( null, handleErrors );

// 如果handleErrors(..)引发了自身的异常,会被全局抛出到这里
```

相比没有结束的链接或者任意时长的定时器，这种方案看起来似乎更有吸引力。但最大的问题是，它并不是 ES6 标准的一部分，所以不管听起来怎么好，要成为可靠的普遍解决方案，它还有很长一段路要走。

那我们就这么被卡住了？不完全是。

浏览器有一个特有的功能是我们的代码所没有的：它们可以跟踪并了解所有对象被丢弃以及被垃圾回收的时机。所以，浏览器可以追踪 Promise 对象。如果在它被垃圾回收的时候其中有拒绝，浏览器就能够确保这是一个真正的未捕获错误，进而可以确定应该将其报告到开发者终端。

在编写本书时候，Chrome 和 Firefox 对于这种（追踪）未捕获拒绝功能都已经有了早期的实验性支持，尽管还不完善。

但是，如果一个 Promise 未被垃圾回收——各种不同的代码模式中很容易不小心出现这种情况——浏览器的垃圾回收嗅探就无法帮助你知晓和诊断一个被你默默拒绝的 Promise。

还有其他办法吗？有。

### 3.5.3 成功的坑

接下来的内容只是理论上的，关于未来的 Promise 可以变成什么样。我相信它会变得比现在我们所拥有的高级得多。我认为这种改变甚至可能是后 ES6 的，因为我觉得它不会打破与 ES6 Promise 的 web 兼容性。还有，如果你认真对待的话，它可能是可以 polyfill/prollyfill 的。我们来看一下。

- 默认情况下，Promsie 在下一个任务或时间循环 tick 上（向开发者终端）报告所有拒绝，如果在这个时间点上该 Promise 上还没有注册错误处理函数。
- 如果想要一个被拒绝的 Promise 在查看之前的某个时间段内保持被拒绝状态，可以调用 `defer()`，这个函数优先级高于该 Promise 的自动错误报告。

如果一个 Promise 被拒绝的话，默认情况下会向开发者终端报告这个事实（而不是默认为沉默）。可以选择隐式（在拒绝之前注册一个错误处理函数）或者显式（通过 `defer()`）禁止这种报告。在这两种情况下，都是由你来控制误报的情况。

考虑：

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

// foo(..)是支持Promise的
foo( 42 )
.then(
    function fulfilled(){
        return p;
    },
    function rejected(err){
        // 处理foo(..)错误
    }
);
...
```

创建 `p` 的时候，我们知道需要等待一段时间才能使用或查看它的拒绝结果，所以我们就调用 `defer()`，这样就不会有全局报告出现。为了便于链接，`defer()` 只是返回这同一个 promise。

从 `foo(..)` 返回的 promise 立刻就被关联了一个错误处理函数，所以它也隐式消除了出错全局报告。

但是，从 `then(..)` 调用返回的 promise 没有调用 `defer()`，也没有关联错误处理函数，所

以如果它（从内部或决议处理函数）拒绝的话，就会作为一个未捕获错误被报告到开发者终端。

这种设计就是成功的坑。默认情况下，所有的错误要么被处理要么被报告，这几乎是绝大多数情况下几乎所有开发者会期望的结果。你要么必须注册一个处理函数要么特意选择退出，并表明你想把错误处理延迟到将来。你这时候是在为特殊情况主动承担特殊的责任。

这种方案唯一真正的危险是，如果你 `defer()` 了一个 Promise，但之后却没有成功查看或处理它的拒绝结果。

但是，你得特意调用 `defer()` 才能选择进入这个绝望的陷阱（默认情况下总是成功的坑）。所以这是你自己的问题，别人也无能为力。

我认为 Promise 错误处理还是有希望的（后 ES6）。我希望权威组织能够重新思考现状，考虑一下这种修改。同时，你也可以自己实现这一点（这是一道留给大家的挑战性习题！），或者选择更智能的 Promise 库为实现！

这个错误处理 / 报告的精确模板是在我的 asynquesce Promise 抽象库中实现的。本部分的附录 A 中详细讨论了这个库。

## 3.6 Promise 模式

前文我们无疑已经看到了使用 Promise 链的顺序模式（this-then-this-then-that 流程控制），但是可以基于 Promise 构建的异步模式抽象还有很多变体。这些模式是为了简化异步流程控制，这使得我们的代码更容易追踪和维护，即使在程序中最复杂的部分也是如此。

原生 ES6 `Promise` 实现中直接支持了两个这样的模式，所以我们可以免费得到它们，用作构建其他模式的基本块。

### 3.6.1 `Promise.all([ .. ])`

在异步序列中（Promise 链），任意时刻都只能有一个异步任务正在执行——步骤 2 只能在步骤 1 之后，步骤 3 只能在步骤 2 之后。但是，如果想要同时执行两个或更多步骤（也就是“并行执行”），要怎么实现呢？

在经典的编程术语中，门（gate）是这样一种机制要等待两个或更多并行 / 并发的任务都完成才能继续。它们的完成顺序并不重要，但是必须都要完成，门才能打开并让流程控制继续。

在 Promise API 中，这种模式被称为 all([ .. ])。

假定你想要同时发送两个 Ajax 请求，等它们不管以什么顺序全部完成之后，再发送第三个 Ajax 请求。考虑：

```
// request(..)是一个Promise-aware Ajax工具
// 就像我们在本章前面定义的一样

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

Promise.all( [p1,p2] )
.then( function(msgs){
    // 这里,p1和p2完成并把它们的消息传入
    return request(
        "http://some.url.3/?v=" + msgs.join(",")
    );
} )
.then( function(msg){
    console.log( msg );
} );
```

Promise.all([ .. ]) 需要一个参数，是一个数组，通常由 Promise 实例组成。从 Promise.all([ .. ]) 调用返回的 promise 会收到一个完成消息（代码片段中的 msg）。这是一个由所有传入 promise 的完成消息组成的数组，与指定的顺序一致（与完成顺序无关）。

严格说来，传给 Promise.all([ .. ]) 的数组中的值可以是 Promise、thenable，甚至是立即值。就本质而言，列表中的每个值都会通过 Promise.resolve(..) 过滤，以确保要等待的是一个真正的 Promise，所以立即值会被规范化为为这个值构建的 Promise。如果数组是空的，主 Promise 就会立即完成。

从 Promise.all([ .. ]) 返回的主 promise 在且仅在所有的成员 promise 都完成后才会完成。如果这些 promise 中有任何一个被拒绝的话，主 Promise.all([ .. ])promise 就会立即被拒绝，并丢弃来自其他所有 promise 的全部结果。

永远要记住为每个 promise 关联一个拒绝 / 错误处理函数，特别是从 Promise.all([ .. ]) 返回的那一个。

### 3.6.2 Promise.race([ .. ])

尽管 Promise.all([ .. ]) 协调多个并发 Promise 的运行，并假定所有 Promise 都需要完成，但有时候你会想只响应“第一个跨过终点线的 Promise”，而抛弃其他 Promise。

这种模式传统上称为门闩，但在 Promise 中称为竞态。

虽然“只有第一个到达终点的才算胜利”这个比喻很好地描述了其行为特性，但遗憾的是，由于竞态条件通常被认为是程序中的 bug（参见第 1 章），所以从某种程度上说，“竞争”这个词已经是一个具有固定意义的术语了。不要混淆了 `Promise.race([..])` 和竞态条件。

`Promise.race([ .. ])` 也接受单个数组参数。这个数组由一个或多个 Promise、thenable 或立即值组成。立即值之间的竞争在实践中没有太大意义，因为显然列表中的第一个会获胜，就像赛跑中有一个选手是从终点开始比赛一样！

与 `Promise.all([ .. ])` 类似，一旦有任何一个 Promise 决议为完成，`Promise.race([ .. ])` 就会完成；一旦有任何一个 Promise 决议为拒绝，它就会拒绝。

一项竞赛需要至少一个“参赛者”。所以，如果你传入了一个空数组，主 `race([..])` Promise 永远不会决议，而不是立即决议。这很容易搬起石头砸自己的脚！ ES6 应该指定它完成或拒绝，抑或只是抛出某种同步错误。遗憾的是，因为 Promise 库在时间上早于 ES6 `Promise`，它们不得已遗留了这个问题，所以，要注意，永远不要递送空数组。

再回顾一下前面的并发 Ajax 例子，不过这次的 `p1` 和 `p2` 是竞争关系：

```
// request(..)是一个支持Promise的Ajax工具
// 就像我们在本章前面定义的一样

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

Promise.race( [p1,p2] )
.then( function(msg){
    // p1或者p2将赢得这场竞赛
    return request(
        "http://some.url.3/?v=" + msg
    );
} )
.then( function(msg){
    console.log( msg );
} );
```

因为只有一个 promise 能够取胜，所以完成值是单个消息，而不是像对 `Promise.all([ .. ])` 那样的是一个数组。

#### 1. 超时竞赛

我们之前看到过这个例子，其展示了如何使用 `Promise.race([ .. ])` 表达 Promise 超时模式：

```
// foo()是一个支持Promise的函数
```

```
// 前面定义的timeoutPromise(..)返回一个promise,
// 这个promise会在指定延时之后拒绝

// 为foo()设定超时
Promise.race( [
    foo(),                  // 启动foo()
    timeoutPromise( 3000 )  // 给它3秒钟
] )
.then(
    function(){
        // foo(..)按时完成!
    },
    function(err){
        // 要么foo()被拒绝,要么只是没能够按时完成,
        // 因此要查看err了解具体原因
    }
);
```

在多数情况下，这个超时模式能够很好地工作。但是，还有一些微妙的情况需要考虑，并且坦白地说，对于 `Promise.race([ .. ])` 和 `Promise.all([ .. ])` 也都是如此。

### 2. finally

一个关键问题是："那些被丢弃或忽略的 promise 会发生什么呢？"我们并不是从性能的角度提出这个问题的——通常最终它们会被垃圾回收——而是从行为的角度（副作用等）。Promise 不能被取消，也不应该被取消，因为那会摧毁 3.8.5 节讨论的外部不变性原则，所以它们只能被默默忽略。

那么如果前面例子中的 `foo()` 保留了一些要用的资源，但是出现了超时，导致这个 promise 被忽略，这又会怎样呢？在这种模式中，会有什么为超时后主动释放这些保留资源提供任何支持，或者取消任何可能产生的副作用吗？如果你想要的只是记录下 `foo()` 超时这个事实，又会如何呢？

有些开发者提出，Promise 需要一个 `finally(..)` 回调注册，这个回调在 Promise 决议后总是会被调用，并且允许你执行任何必要的清理工作。目前，规范还没有支持这一点，不过在 ES7+ 中也许可以。只好等等看了。

它看起来可能类似于：

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

p.then( something )
.finally( cleanup )
.then( another )
.finally( cleanup );
```

在各种各样的 Promise 库中，`finally(..)` 还是会创建并返回一个新的 Promise（以支持链接继续）。如果 `cleanup(..)` 函数要返回一个 Promise 的话，这个 promise 就会被连接到链中，这意味着这里还是会有前面讨论过的未处理拒绝问题。

同时，我们可以构建一个静态辅助工具来支持查看（而不影响）Promise 的决议：

```
// polyfill安全的guard检查
if (!Promise.observe) {
    Promise.observe = function(pr,cb) {
        // 观察pr的决议
        pr.then(
            function fulfilled (msg){
                // 安排异步回调(作为Job)
                Promise.resolve( msg ).then( cb );
            },
            function rejected(err){
                // 安排异步回调(作为Job)
                Promise.resolve( err ).then( cb );
            }
        );

        // 返回最初的promise
        return pr;
    };
}
```

下面是如何在前面的超时例子中使用这个工具：

```
Promise.race( [
    Promise.observe(
        foo(),                          // 试着运行foo()
        function cleanup(msg){
            // 在foo()之后清理,即使它没有在超时之前完成
        }
    ),
    timeoutPromise( 3000 )  // 给它3秒钟
] )
```

这个辅助工具 `Promise.observe(..)` 只是用来展示可以如何查看 Promise 的完成而不对其产生影响。其他的 Promise 库有自己的解决方案。不管如何实现，你都很可能遇到需要确保 Promise 不会被意外默默忽略的情况。

### 3.6.3 `all([ .. ])` 和 `race([ .. ])` 的变体

虽然原生 ES6 `Promise` 中提供了内建的 `Promise.all([ .. ])` 和 `Promise.race([ .. ])`，但这些语义还有其他几个常用的变体模式。

- `none([ .. ])`
  这个模式类似于 `all([ .. ])`，不过完成和拒绝的情况互换了。所有的 Promise 都要被拒绝，即拒绝转化为完成值，反之亦然。

- `any([ .. ])`
  这个模式与 `all([ .. ])` 类似，但是会忽略拒绝，所以只需要完成一个而不是全部。

- `first([ .. ])`
  这个模式类似于与 `any([ .. ])` 的竞争，即只要第一个 Promise 完成，它就会忽略后续的任何拒绝和完成。

- `last([ .. ])`
  这个模式类似于 `first([ .. ])`，但却是只有最后一个完成胜出。

有些 Promise 抽象库提供了这些支持，但也可以使用 Promise、`race([ .. ])` 和 `all([ .. ])` 这些机制，你自己来实现它们。

比如，可以像这样定义 `first([ .. ])`：

```
// polyfill安全的guard检查
if (!Promise.first) {
    Promise.first = function(prs) {
        return new Promise( function(resolve,reject){
            // 在所有promise上循环
            prs.forEach( function(pr){
                // 把值规整化
                Promise.resolve( pr )
                // 不管哪个最先完成,就决议主promise
                .then( resolve );
            } );
        } );
    };
}
```

在这个 `first(..)` 实现中，如它的所有 promise 都拒绝的话，它不会拒绝。它只会挂住，非常类似于 `Promise.race([])`。如果需要的话，可以添加额外的逻辑跟踪每个 promise 拒绝。如果所有的 promise 都被拒绝，就在主 promise 上调用 `reject()`。这个实现留给你当练习。

## 3.6.4 并发迭代

有些时候会需要在一列 Promise 中迭代，并对所有 Promise 都执行某个任务，非常类似于对同步数组可以做的那样（比如 `forEach(..)`、`map(..)`、`some(..)` 和 `every(..)`）。如果要对每个 Promise 执行的任务本身是同步的，那这些工具就可以工作，就像前面代码中的

forEach(..)。

但如果这些任务从根本上是异步的，或者可以 / 应该并发执行，那你可以使用这些工具的异步版本，许多库中提供了这样的工具。

举例来说，让我们考虑一下一个异步的 map(..) 工具。它接收一个数组的值（可以是 Promise 或其他任何值），外加要在每个值上运行一个函数（任务）作为参数。map(..) 本身返回一个 promise，其完成值是一个数组，该数组（保持映射顺序）保存任务执行之后的异步完成值：

```
if (!Promise.map) {
    Promise.map = function(vals,cb) {
        // 一个等待所有map的promise的新promise
        return Promise.all(
            // 注:一般数组map(..)把值数组转换为 promise数组
            vals.map( function(val){
                // 用val异步map之后决议的新promise替换val
                return new Promise( function(resolve){
                    cb( val, resolve );

                } );
            } )
        );
    };
}
```

在这个 map(..) 实现中，不能发送异步拒绝信号，但如果在映射的回调（cb(..)）内出现同步的异常或错误，主 Promise.map(..) 返回的 promise 就会拒绝。

下面展示如何在一组 Promise（而非简单的值）上使用 map(..)：

```
var p1 = Promise.resolve( 21 );
var p2 = Promise.resolve( 42 );
var p3 = Promise.reject( "Oops" );

// 把列表中的值加倍,即使是在Promise中
Promise.map( [p1,p2,p3], function(pr,done){
    // 保证这一条本身是一个Promise
    Promise.resolve( pr )
    .then(
        // 提取值作为v
        function(v){
            // map完成的v到新值
            done( v * 2 );
        },
        // 或者map到promise拒绝消息
        done
    );
```

```
} )
.then( function(vals){
    console.log( vals );         // [42,84,"Oops"]
} );
```

## 3.7 Promise API 概述

本章已经在多处零零碎碎地展示了 ES6 `Promise` API，现在让我们来总结一下。

下面的 API 只对于 ES6 是原生的，但是有符合规范的适配版（不只是对 Promise 库的扩展），其定义了 Promise 及它的所有相关特性，这样你在前 ES6 浏览器中也可以使用原生 Promise。这样的适配版之一是 Native Promise Only（http://github.com/getify/native-promise-only），是我写的。

### 3.7.1 new Promise(..) 构造器

有启示性的构造器 `Promise(..)` 必须和 `new` 一起使用，并且必须提供一个函数回调。这个回调是同步的或立即调用的。这个函数接受两个函数回调，用以支持 promise 的决议。通常我们把这两个函数称为 `resolve(..)` 和 `reject(..)`：

```
var p = new Promise( function(resolve,reject){
    // resolve(..)用于决议/完成这个promise
    // reject(..)用于拒绝这个promise
} );
```

`reject(..)` 就是拒绝这个 promise；但 `resolve(..)` 既可能完成 promise，也可能拒绝，要根据传入参数而定。如果传给 `resolve(..)` 的是一个非 Promise、非 thenable 的立即值，这个 promise 就会用这个值完成。

但是，如果传给 `resolve(..)` 的是一个真正的 Promise 或 thenable 值，这个值就会被递归展开，并且（要构造的）promise 将取用其最终决议值或状态。

### 3.7.2 Promise.resolve(..) 和 Promise.reject(..)

创建一个已被拒绝的 Promise 的快捷方式是使用 `Promise.reject(..)`，所以以下两个 promise 是等价的：

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

var p2 = Promise.reject( "Oops" );
```

`Promise.resolve(..)` 常用于创建一个已完成的 Promise，使用方式与 `Promise.reject(..)`