Promise 的决议结果只有两种可能：完成或拒绝，附带一个可选的单个值。如果 Promise 完成，那么最终的值称为完成值；如果拒绝，那么最终的值称为原因（也就是“拒绝的原因”）。Promise 只能被决议（完成或者拒绝）**一次**。之后再次试图完成或拒绝的动作都会被忽略。因此，一旦 Promise 被决议，它就是不变量，不会发生改变。

显然，看待 Promise 有各种不同的角度。没有哪一个角度是完整的，但每一种看法都提供了整体的一面。最重要的一点是，它对只用回调的异步方法给予了重大改进，即提供了有序性、可预测性和可靠性。

4.1.1 构造和使用 Promise

可以通过构造器 `Promise(..)` 构造 promise 实例：

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

提供给构造器 `Promise(..)` 的两个参数都是函数，一般称为 `resolve(..)` 和 `reject(..)`。它们是这样使用的。

- 如果调用 `reject(..)`，这个 promise 被拒绝，如果有任何值传给 `reject(..)`，这个值就被设置为拒绝的原因值。
- 如果调用 `resolve(..)` 且没有值传入，或者传入任何非 promise 值，这个 promise 就完成。
- 如果调用 `resove(..)` 并传入另外一个 promise，这个 promise 就会采用传入的 promise 的状态（要么实现要么拒绝）——不管是立即还是最终。

下面是通过 promise 重构回调函数调用的常用方法。假定你最初是使用需要能够调用 error-first 风格回调的 `ajax(..)` 工具：

```
function ajax(url,cb) {
    // 建立请求，最终会调用cb(..)
}

// ..

ajax( "http://some.url.1", function handler(err,contents){
    if (err) {
        // 处理ajax错误
    }
    else {
        // 处理contents成功情况
    }
} );
```

可以将其转化为：

```
function ajax(url) {
    return new Promise( function pr(resolve,reject){
        // 建立请求，最终会调用resolve(..)或者reject(..)
    } );
}

// ..

ajax( "http://some.url.1" )
.then(
    function fulfilled(contents){
        // 处理contents成功情况
    },
    function rejected(reason){
        // 处理ajax出错原因
    }
);
```

Promise 有一个 `then(..)` 方法，接受一个或两个回调函数作为参数。前面的函数（如果存在的话）会作为 promise 成功完成后的处理函数。第二个函数（如果存在的话）会作为 promise 被显式拒绝后的处理函数，或者在决议过程中出现错误 / 异常的情况下的处理函数。

如果某个参数被省略，或者不是一个有效的函数——通常是 `null`，那么一个默认替代函数就会被采用。默认的成功回调把完成值传出，默认的出错回调会传递拒绝原因值。

`then(null, handleRejection)` 调用的简写形式是 `catch(handleRejection)`。

`then(..)` 和 `catch(..)` 都会自动构造并返回另外一个 promise 实例，这个实例连接到接受原来的 promise 的不管是完成或拒绝处理函数（实际调用的那个）的返回值。考虑：

```
ajax( "http://some.url.1" )
.then(
    function fulfilled(contents){
        return contents.toUpperCase();
    },
    function rejected(reason){
        return "DEFAULT VALUE";
    }
)
.then( function fulfilled(data){
    // 处理来自于原来promise的处理函数的数据
} );
```

这段代码中，从 `fulfilled(..)` 或者 `rejected(..)` 返回一个立即值，然后这个值在下次事件中被第二个 `then(..)` 的 `fulfilled(..)` 接受。如果传入的是新的 promise，那么这个新的 promise 就会作为决议结果被导入和采用：

```
ajax( "http://some.url.1" )
.then(
```

```
        function fulfilled(contents){
            return ajax(
                "http://some.url.2?v=" + contents
            );
        },
        function rejected(reason){
            return ajax(
                "http://backup.url.3?err=" + reason
            );
        }
    )
    .then( function fulfilled(contents){
        // contents来自于后续的ajax(..)调用，不管是哪个调用
    } );
```

需要注意的是：第一个 `fulfilled(..)` 内部的异常（即被拒绝的 promise）**不会**导致第一个 `rejected(..)` 被调用，因为这个处理函数只响应第一个原始 promise 的决议。而第二个 promise 会接受这个拒绝，这个 promise 是在第二个 `then(..)` 上调用的。

前面的代码片段中，我们没有监听拒绝，这意味着它会默默保持这个状态等待未来的观测。如果永远不通过 `then(..)` 或 `catch(..)` 调用来观察的话，它就会一直保持未处理状态。有些浏览器开发者终端可能会监测到这些未处理拒绝并报告出来，但是这并不是可靠的保证；我们应该一直观测 promise 拒绝。

这里只对 Promise 的理论和行为给出了一个简要概述。关于更深入的探讨，参见本系列《你不知道的 JavaScript（中卷）》第二部分的第 3 章。

4.1.2 Thenable

`Promise(..)` 构造器的真正实例是 Promise。但还有一些类 promise 对象，称为 thenable，一般来说，它们也可以用 Promise 机制解释。

任何提供了 `then(..)` 函数的对象（或函数）都被认为是 thenable。Promise 机制中所有可以接受真正 promise 状态的地方，也都可以处理 thenable。

从根本上说，thenable 就是所有类 promise 值的一个通用标签，这些类 promise 不是被真正的 `Promise(..)` 构造器而是被其他系统创造出来。从这个角度来说，通常 thenable 的可靠性要低于真正的 Promise。举个例子，考虑下面这个胡作非为的 thenable：

```
var th = {
    then: function thener( fulfilled ) {
        // 每100ms调用一次fulfilled(..)，直到永远
        setInterval( fulfilled, 100 );
    }
};
```

如果接收到了这个 thenable，并通过 th.then(..) 把它链接起来，很可能你会吃惊地发现自己的完成处理函数会被重复调用，而正常的 Promise 应该只会决议一次。

一般来说，如果从某个其他系统接收到一个自称 promise 或者 thenable 的东西，不应该盲目信任它。在下一小节中，我们会介绍一个 ES6 Promise 包含的工具，用来帮助解决这个信任问题。

但为了更进一步理解这个问题的危害性，考虑一下：**任何**代码中的**任何**对象，当然如果是与 Promise 一起使用的话，只要定义了名为 then() 的方法，就可能被当作是一个 thenable，不管这个东西的目的是否与 Promise 风格的异步编码相关。

在 ES6 之前，并没有对 then(..) 方法名称有任何特殊保留，可以想象应该有一些实现选用了这个方法名称，而 Promise 那时候还没有出现呢。最可能出现的误用 thenable 的情况是那些使用了 then(..) 方法，但是并没有严格遵循 Promise 风格的异步库——确实有几个这样的“野生”库。

你的责任是，避免把可能被误认为 thenable 的值直接用于 Promise 机制。

4.1.3 Promise API

Promise API 还提供了一些静态方法与 Promise 一起工作。

Promise.resolve(..) 创建了一个决议到传入值的 promise。我们把它的工作机制与手动方法对比一下：

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

var p2 = new Promise( function pr(resolve){
    resolve( 42 );
} );
```

p1 和 p2 的最终行为方式是完全相同的。通过 promise 来决议也是一样：

```
var theP = ajax( .. );

var p1 = Promise.resolve( theP );

var p2 = new Promise( function pr(resolve){
    resolve( theP );
} );
```

Promise.resolve(..) 是一个针对上一小节中介绍的 thenable 信任问题的解决方案。对于任何还没有完全确定是可信 promise 的值，甚至它可能是立即值，都可以通过把它传给 Promise.resolve(..) 来规范化。如果这个值已经是可以确定的 promise 或者 thenable，它的状态 / 决议就会被直接采用，这样会避免出错。而如果它是一个立即值，那么它会被“封装”为一个真正的 promise，这样就把它的行为方式规范为异步的。

`Promise.reject(..)` 创建一个立即被拒绝的 promise，和与它对应的 `Promise(..)` 构造器一样：

```
var p1 = Promise.reject( "Oops" );

var p2 = new Promise( function pr(resolve,reject){
    reject( "Oops" );
} );
```

`resolve(..)` 和 `Promise.resolve(..)` 可以接受 promise 并接受它的状态 / 决议，而 `reject (..)` 和 `Promise.reject(..)` 并不区分接收的值是什么。所以，如果传入 promise 或 thenable 来拒绝，这个 promise / thenable 本身会被设置为拒绝原因，而不是其底层值。

`Promise.all([ .. ])` 接受一个或多个值的数组（比如，立即值、promise、thenable）。它返回一个 promise，如果所有的值都完成，这个 promise 的结果是完成；一旦它们中的某一个被拒绝，那么这个 promise 就立即被拒绝。

从这些值 /promise 开始：

```
var p1 = Promise.resolve( 42 );
var p2 = new Promise( function pr(resolve){
    setTimeout( function(){
        resolve( 43 );
    }, 100 );
} );
var v3 = 44;
var p4 = new Promise( function pr(resolve,reject){
    setTimeout( function(){
        reject( "Oops" );
    }, 10);
} );
```

让我们考虑一下 `Promise.all([ .. ])` 如何与这些值的组合合作：

```
Promise.all( [p1,p2,v3] )
.then( function fulfilled(vals){
    console.log( vals );             // [42,43,44]
} );

Promise.all( [p1,p2,v3,p4] )
.then(
    function fulfilled(vals){
        // 不会到达这里
    },
    function rejected(reason){
        console.log( reason );         // Oops
    }
);
```

`Promise.all([ .. ])` 等待所有都完成（或者第一个拒绝），而 `Promise.race([ .. ])` 等待第一个完成或者拒绝。考虑：

```
// 注意：重新设置检测值，以避免被时序问题所误导！

Promise.race( [p2,p1,v3] )
.then( function fulfilled(val){
    console.log( val );                 // 42
} );

Promise.race( [p2,p4] )
.then(
    function fulfilled(val){
        // 不会到达这里
    },
    function rejected(reason){
        console.log( reason );          // Oops
    }
);
```

`Promise.all([])` 将会立即完成（没有完成值），`Promise.race([])` 将会永远挂起。这是一个很奇怪的不一致，因此我建议，永远不要用空数组使用这些方法。

4.2 生成器 + Promise

可以把一系列 promise 以链式表达，用以代表程序的异步流控制。考虑：

```
step1()
.then(
    step2,
    step2Failed
)
.then(
    function(msg) {
        return Promise.all( [
            step3a( msg ),
            step3b( msg ),
            step3c( msg )
        ] )
    }
)
.then(step4);
```

但是，还有一种更好的方案可以用来表达异步流控制，而且从编码规范的角度来说也要比很长的 promise 链可取得多。我们可以使用在第 3 章学到的生成器来表达我们的异步流控制。

这个重要的模式需要理解一下：生成器可以 `yield` 一个 promise，然后这个 promise 可以被绑定，用其完成值来恢复这个生成器的运行。

考虑一下用生成器来表达前面代码片段的异步流控制：

```
function *main() {
    var ret = yield step1();

    try {
        ret = yield step2( ret );
    }
    catch (err) {
        ret = yield step2Failed( err );
    }

    ret = yield Promise.all( [
        step3a( ret ),
        step3b( ret ),
        step3c( ret )
    ] );

    yield step4( ret );
}
```

表面看来，这段代码似乎比前面代码中的等价 promise 链实现更冗长。但是，它提供了一种更有吸引力，同时也是更重要、更易懂、更合理且看似同步的编码风格（通过给“返回”值的 = 赋值等）。特别是由于 `try..catch` 出错处理可以跨过这些隐藏的异步边界。

为什么与生成器一起使用 Promise？不用 Promise 肯定也可以实现异步生成器编码。

Promise 是一种把普通回调或者 thunk 控制反转（参见本系列《你不知道的 JavaScript（中卷）》第二部分）反转回来的可靠系统。因此，把 Promise 的可信任性与生成器的同步代码组合在一起有效解决了回调所有的重要缺陷。另外，像 `Promise.all([ .. ])` 这样的工具也是在生成器的单个 `yield` 步骤表达并发性的一个优秀又简洁的方法。

这个魔法是如何实现的呢？我们需要一个可以运行生成器的**运行器**（runner），接受一个 `yield` 出来的 promise，然后将其连接起来用以恢复生成器，方法是或者用完成成功值，或者用拒绝原因值抛出一个错误到生成器。

很多支持异步的工具 / 库都有这样的“运行器”，比如 `Q.spawn(..)`，以及我的 asyncquence 库的 `runner(..)` 插件。而这里是用一个单独的运行器来说明这个过程的工作原理：

```
function run(gen) {
    var args = [].slice.call( arguments, 1), it;

    it = gen.apply( this, args );

    return Promise.resolve()
        .then( function handleNext(value){
            var next = it.next( value );

            return (function handleResult(next){
                if (next.done) {
                    return next.value;
```

```
                }
                else {
                    return Promise.resolve( next.value )
                        .then(
                            handleNext,
                            function handleErr(err) {
                                return Promise.resolve(
                                    it.throw( err )
                                )
                                .then( handleResult );
                            }
                        );
                }
            })( next );
        } );
}
```

参见本系列《你不知道的 JavaScript（中卷）》第二部分，可以获取这个工具更详尽注释的版本。另外，由各种异步库提供的这种 run 工具通常比我们这里展示的更强大，功能更完善。举例来说，asynquence 的 `runner(..)` 能够处理 `yield` 出来的 promise、序列、thunk 和立即（非 promise）值，为我们提供了无限的灵活性。

所以现在运行前面代码中的 `*main()` 就这么简单：

```
run( main )
.then(
    function fulfilled(){
        // *main()成功完成
    },
    function rejected(reason){
        // 哎呀，出错了
    }
);
```

本质上说，只要代码中出现超过两个异步步骤的流控制逻辑，都可以也应该使用由 run 工具驱动的 promise-yield 生成器以异步风格表达控制流。这样可以使代码理解和维护起来更简单。

这种“yield 一个 promise 来恢复生成器”的模式将会成为一个常用模式，这个模式非常强大，下一个版本的 JavaScript 几乎肯定会引入一个新的函数类型来自动执行这种模式而无需 run 工具。我们将在第 8 章介绍 `async function`（应该是叫这个名字）。

4.3 小结

随着 JavaScript 越来越成熟以及应用越来越广泛，异步编程越发地成为核心问题。随着需求变得越来越复杂，回调也变得越来越难以胜任，直到完全崩溃。

值得高兴的是，ES6 新增了 Promise 来弥补回调的主要缺陷之一：缺少对可预测行为方式的保证。Promise 代表了来自于可能异步的任务的未来完成值，跨越同步和异步边界对行为进行规范化。

但是，Promise 与生成器的结合完全实现了重新安排异步流控制代码来消除丑陋的回调乱炖（或称“地狱”）。

现在，我们可以在各种异步库运行器的帮助下管理这些交互，而 JavaScript 最终会提供专门的语法来支持这种交互。

第 5 章

集合

对于任何 JavaScript 程序来说，对数据的结构化组合和访问都是一个关键部分。这个语言从一开始到现在，创建数据结构的主要机制一直都是数组和对象。当然，基于它们已经建立了很多作为用户库的高级数据结构。

在 ES6 中，已经把一部分最有用的（也是性能最优的！）数据结构抽象作为原生组件新增到语言之中。

这一章里，我们会从 TypedArray 开始探讨，严格说它是几年之前 ES5 时期的技术，但那时是作为 WebGL 而不是 JavaScript 组件。在 ES6 中，它已经被语言规范直接采纳，进入一级（first class）状态。

Map 就像是一个对象（键 / 值对），但是键值并非只能为字符串，而是可以使用任何值——甚至是另一个对象或 map！ Set 与数组（值的序列）类似，但是其中的值是唯一的；如果新增的值是重复的，就会被忽略。还有相应的弱（与内存 / 垃圾回收相关）版本：WeakMap 和 WeakSet。

5.1 TypedArray

正如在本系列《你不知道的 JavaScript（中卷）》第一部分中介绍的，JavaScript 拥有一组内置类型，比如 `number` 和 `string`。很容易把称为“类型数组”这样的特性想象为一个特定类型的值构成的数组，比如一个只有字符串构成的数组。

但实际上带类型的数组更多是为了使用类数组语义（索引访问等）结构化访问二进制数