我并不喜欢最后还得用 thenable 鸭子类型检测作为 Promise 的识别方案。还有其他选择，比如 branding，甚至 anti-branding。可我们所用的似乎是针对最差情况的妥协。但情况也并不完全是一片黯淡。后面我们就会看到，thenable 鸭子类型检测还是有用的。只是要清楚，如果 thenable 鸭子类型误把不是 Promise 的东西识别为了 Promise，可能就是有害的。

3.3 Promise 信任问题

前面已经给出了两个很强的类比，用于解释 Promise 在不同方面能为我们的异步代码做些什么。但如果止步于此的话，我们就错过了 Promise 模式构建的可能最重要的特性：信任。

未来值和完成事件这两个类比在我们之前探讨的代码模式中很明显。但是，我们还不能一眼就看出 Promise 为什么以及如何用于解决 2.3 节列出的所有控制反转信任问题。稍微深入探究一下的话，我们就不难发现它提供了一些重要的保护，重新建立了第 2 章中已经毁掉的异步编码可信任性。

先回顾一下只用回调编码的信任问题。把一个回调传入工具 `foo(..)` 时可能出现如下问题：

- 调用回调过早；
- 调用回调过晚（或不被调用）；
- 调用回调次数过少或过多；
- 未能传递所需的环境和参数；
- 吞掉可能出现的错误和异常。

Promise 的特性就是专门用来为这些问题提供一个有效的可复用的答案。

3.3.1 调用过早

这个问题主要就是担心代码是否会引入类似 Zalgo 这样的副作用（参见第 2 章）。在这类问题中，一个任务有时同步完成，有时异步完成，这可能会导致竞态条件。

根据定义，Promise 就不必担心这种问题，因为即使是立即完成的 Promise（类似于 `new Promise(function(resolve){ resolve(42); })`）也无法被同步观察到。

也就是说，对一个 Promise 调用 `then(..)` 的时候，即使这个 Promise 已经决议，提供给 `then(..)` 的回调也总会被异步调用（对此的更多讨论，请参见 1.5 节）。

不再需要插入你自己的 `setTimeout(..,0)` hack，Promise 会自动防止 Zalgo 出现。

3.3.2 调用过晚

和前面一点类似，Promise 创建对象调用 `resolve(..)` 或 `reject(..)` 时，这个 Promise 的 `then(..)` 注册的观察回调就会被自动调度。可以确信，这些被调度的回调在下一个异步事件点上一定会被触发（参见 1.5 节）。

同步查看是不可能的，所以一个同步任务链无法以这种方式运行来实现按照预期有效延迟另一个回调的发生。也就是说，一个 Promise 决议后，这个 Promise 上所有的通过 `then(..)` 注册的回调都会在下一个异步时机点上依次被立即调用（再次提醒，请参见 1.5 节）。这些回调中的任意一个都无法影响或延误对其他回调的调用。

举例来说：

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

这里，`"C"` 无法打断或抢占 `"B"`，这是因为 Promise 的运作方式。

Promise 调度技巧

但是，还有很重要的一点需要指出，有很多调度的细微差别。在这种情况下，两个独立 Promise 上链接的回调的相对顺序无法可靠预测。

如果两个 promise `p1` 和 `p2` 都已经决议，那么 `p1.then(..)`; `p2.then(..)` 应该最终会先调用 `p1` 的回调，然后是 `p2` 的那些。但还有一些微妙的场景可能不是这样的，比如以下代码：

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

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

p2 = new Promise( function(resolve,reject){
    resolve( "A" );
} );

p1.then( function(v){
    console.log( v );
} );
```

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

// A B    <-- 而不是像你可能认为的B A
```

后面我们还会深入介绍，但目前你可以看到，p1 不是用立即值而是用另一个 promise p3 决议，后者本身决议为值 "B"。规定的行为是把 p3 展开到 p1，但是是异步地展开。所以，在异步任务队列中，p1 的回调排在 p2 的回调之后（参见 1.5 节）。

要避免这样的细微区别带来的噩梦，你永远都不应该依赖于不同 Promise 间回调的顺序和调度。实际上，好的编码实践方案根本不会让多个回调的顺序有丝毫影响，可能的话就要避免。

3.3.3 回调未调用

这个问题很常见，Promise 可以通过几种途径解决。

首先，没有任何东西（甚至 JavaScript 错误）能阻止 Promise 向你通知它的决议（如果它决议了的话）。如果你对一个 Promise 注册了一个完成回调和一个拒绝回调，那么 Promise 在决议时总是会调用其中的一个。

当然，如果你的回调函数本身包含 JavaScript 错误，那可能就会看不到你期望的结果，但实际上回调还是被调用了。后面我们会介绍如何在回调出错时得到通知，因为就连这些错误也不会被吞掉。

但是，如果 Promise 本身永远不被决议呢？即使这样，Promise 也提供了解决方案，其使用了一种称为竞态的高级抽象机制：

```
// 用于超时一个Promise的工具
function timeoutPromise(delay) {
    return new Promise( function(resolve,reject){
        setTimeout( function(){
            reject( "Timeout!" );
        }, delay );
    } );
}

// 设置foo()超时
Promise.race( [
    foo(),                      // 试着开始foo()
    timeoutPromise( 3000 )      // 给它3秒钟
] )
.then(
    function(){
        // foo(..)及时完成!
    },
```

```
        function(err){
            // 或者foo()被拒绝,或者只是没能按时完成
            // 查看err来了解是哪种情况
        }
    );
```

关于这个 Promise 超时模式还有更多细节需要考量，后面我们会深入讨论。

很重要的一点是，我们可以保证一个 `foo()` 有一个输出信号，防止其永久挂住程序。

3.3.4 调用次数过少或过多

根据定义，回调被调用的正确次数应该是 1。“过少”的情况就是调用 0 次，和前面解释过的“未被”调用是同一种情况。

“过多”的情况很容易解释。Promise 的定义方式使得它只能被决议一次。如果出于某种原因，Promise 创建代码试图调用 `resolve(..)` 或 `reject(..)` 多次，或者试图两者都调用，那么这个 Promise 将只会接受第一次决议，并默默地忽略任何后续调用。

由于 Promise 只能被决议一次，所以任何通过 `then(..)` 注册的（每个）回调就只会被调用一次。

当然，如果你把同一个回调注册了不止一次（比如 `p.then(f); p.then(f);`），那它被调用的次数就会和注册次数相同。响应函数只会被调用一次，但这个保证并不能预防你搬起石头砸自己的脚。

3.3.5 未能传递参数 / 环境值

Promise 至多只能有一个决议值（完成或拒绝）。

如果你没有用任何值显式决议，那么这个值就是 `undefined`，这是 JavaScript 常见的处理方式。但不管这个值是什么，无论当前或未来，它都会被传给所有注册的（且适当的完成或拒绝）回调。

还有一点需要清楚：如果使用多个参数调用 `resovle(..)` 或者 `reject(..)`，第一个参数之后的所有参数都会被默默忽略。这看起来似乎违背了我们前面介绍的保证，但实际上并没有，因为这是对 Promise 机制的无效使用。对于这组 API 的其他无效使用（比如多次重复调用 `resolve(..)`），也是类似的保护处理，所以这里的 Promise 行为是一致的（如果不是有点令人沮丧的话）。

如果要传递多个值，你就必须要把它们封装在单个值中传递，比如通过一个数组或对象。

对环境来说，JavaScript 中的函数总是保持其定义所在的作用域的闭包（参见《你不知道的 JavaScript（上卷）》的“作用域和闭包”部分），所以它们当然可以继续访问你提供的

环境状态。当然，对于只用回调的设计也是这样，因此这并不是 Promise 特有的优点——但不管怎样，这仍是我们可以依靠的一个保证。

3.3.6 吞掉错误或异常

基本上，这部分是上个要点的再次说明。如果拒绝一个 Promise 并给出一个理由（也就是一个出错消息），这个值就会被传给拒绝回调。

不过在这里还有更多的细节需要研究。如果在 Promise 的创建过程中或在查看其决议结果过程中的任何时间点上出现了一个 JavaScript 异常错误，比如一个 `TypeError` 或 `ReferenceError`，那这个异常就会被捕捉，并且会使这个 Promise 被拒绝。

举例来说：

```
var p = new Promise( function(resolve,reject){
    foo.bar();  // foo未定义,所以会出错!
    resolve( 42 );  // 永远不会到达这里 :(
} );

p.then(
    function fulfilled(){
        // 永远不会到达这里 :(
    },
    function rejected(err){
        // err将会是一个TypeError异常对象来自foo.bar()这一行
    }
);
```

`foo.bar()` 中发生的 JavaScript 异常导致了 Promise 拒绝，你可以捕捉并对其作出响应。

这是一个重要的细节，因为其有效解决了另外一个潜在的 Zalgo 风险，即出错可能会引起同步响应，而不出错则会是异步的。Promise 甚至把 JavaScript 异常也变成了异步行为，进而极大降低了竞态条件出现的可能。

但是，如果 Promise 完成后在查看结果时（`then(..)` 注册的回调中）出现了 JavaScript 异常错误会怎样呢？即使这些异常不会被丢弃，但你会发现，对它们的处理方式还是有点出乎意料，需要进行一些深入研究才能理解：

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

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

```
        // 永远也不会到达这里 :(
    }
);
```

等一下，这看起来像是 `foo.bar()` 产生的异常真的被吞掉了。别担心，实际上并不是这样。但是这里有一个深藏的问题，就是我们没有侦听到它。`p.then(..)` 调用本身返回了另外一个 promise，正是这个 promise 将会因 `TypeError` 异常而被拒绝。

为什么它不是简单地调用我们定义的错误处理函数呢？表面上的逻辑应该是这样啊。如果这样的话就违背了 Promise 的一条基本原则，即 Promise 一旦决议就不可再变。`p` 已经完成为值 `42`，所以之后查看 `p` 的决议时，并不能因为出错就把 `p` 再变为一个拒绝。

除了违背原则之外，这样的行为也会造成严重的损害。因为假如这个 promise `p` 有多个 `then(..)` 注册的回调的话，有些回调会被调用，而有些则不会，情况会非常不透明，难以解释。

3.3.7 是可信任的 Promise 吗

基于 Promise 模式建立信任还有最后一个细节需要讨论。

你肯定已经注意到 Promise 并没有完全摆脱回调。它们只是改变了传递回调的位置。我们并不是把回调传递给 `foo(..)`，而是从 `foo(..)` 得到某个东西（外观上看是一个真正的 Promise），然后把回调传给这个东西。

但是，为什么这就比单纯使用回调更值得信任呢？如何能够确定返回的这个东西实际上就是一个可信任的 Promise 呢？这难道不是一个（脆弱的）纸牌屋，在里面只能信任我们已经信任的？

关于 Promise 的很重要但是常常被忽略的一个细节是，Promise 对这个问题已经有一个解决方案。包含在原生 ES6 `Promise` 实现中的解决方案就是 `Promise.resolve(..)`。

如果向 `Promise.resolve(..)` 传递一个非 Promise、非 thenable 的立即值，就会得到一个用这个值填充的 promise。下面这种情况下，promise `p1` 和 promise `p2` 的行为是完全一样的：

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

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

而如果向 `Promise.resolve(..)` 传递一个真正的 Promise，就只会返回同一个 promise：

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

var p2 = Promise.resolve( p1 );
```

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

更重要的是，如果向 `Promise.resolve(..)` 传递了一个非 Promise 的 thenable 值，前者就会试图展开这个值，而且展开过程会持续到提取出一个具体的非类 Promise 的最终值。

考虑：

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

// 这可以工作,但只是因为幸运而已
p
.then(
    function fulfilled(val){
        console.log( val ); // 42
    },
    function rejected(err){
        // 永远不会到达这里
    }
);
```

这个 `p` 是一个 thenable，但并不是一个真正的 Promise。幸运的是，和绝大多数值一样，它是可追踪的。但是，如果得到的是如下这样的值又会怎样呢：

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

p
.then(
    function fulfilled(val){
        console.log( val ); // 42
    },

    function rejected(err){
        // 啊,不应该运行!
        console.log( err ); // 邪恶的笑
    }
);
```

这个 `p` 是一个 thenable，但是其行为和 promise 并不完全一致。这是恶意的吗？还只是因为它不知道 Promise 应该如何运作？说实话，这并不重要。不管是哪种情况，它都是不可信任的。

尽管如此，我们还是都可以把这些版本的 `p` 传给 `Promise.resolve(..)`，然后就会得到期望

中的规范化后的安全结果：

```
Promise.resolve( p )
.then(
    function fulfilled(val){
        console.log( val ); // 42
    },
    function rejected(err){
        // 永远不会到达这里
    }
);
```

`Promise.resolve(..)` 可以接受任何 thenable，将其解封为它的非 thenable 值。从 `Promise.resolve(..)` 得到的是一个真正的 Promise，是一个可以信任的值。如果你传入的已经是真正的 Promise，那么你得到的就是它本身，所以通过 `Promise.resolve(..)` 过滤来获得可信任性完全没有坏处。

假设我们要调用一个工具 `foo(..)`，且并不确定得到的返回值是否是一个可信任的行为良好的 Promise，但我们可以知道它至少是一个 thenable。`Promise.resolve(..)` 提供了可信任的 Promise 封装工具，可以链接使用：

```
// 不要只是这么做：
foo( 42 )
.then( function(v){
    console.log( v );
} );

// 而要这么做：
Promise.resolve( foo( 42 ) )
.then( function(v){
    console.log( v );
} );
```

对于用 `Promise.resolve(..)` 为所有函数的返回值（不管是不是 thenable）都封装一层。另一个好处是，这样做很容易把函数调用规范为定义良好的异步任务。如果 `foo(42)` 有时会返回一个立即值，有时会返回 Promise，那么 `Promise.resolve( foo(42) )` 就能够保证总会返回一个 Promise 结果。而且避免 Zalgo 就能得到更好的代码。

3.3.8 建立信任

很可能前面的讨论现在已经完全“解决”（resolve，英语中也表示“决议”的意思）了你的疑惑：Promise 为什么是可信任的，以及更重要的，为什么对构建健壮可维护的软件来说，这种信任非常重要。

可以用 JavaScript 编写异步代码而无需信任吗？当然可以。JavaScript 开发者近二十年来一

直都只用回调编写异步代码。

可一旦开始思考你在其上构建代码的机制具有何种程度的可预见性和可靠性时，你就会开始意识到回调的可信任基础是相当不牢靠。

Promise 这种模式通过可信任的语义把回调作为参数传递，使得这种行为更可靠更合理。通过把回调的控制反转反转回来，我们把控制权放在了一个可信任的系统（Promise）中，这种系统的设计目的就是为了使异步编码更清晰。

3.4 链式流

尽管我们之前对此有过几次暗示，但 Promise 并不只是一个单步执行 this-then-that 操作的机制。当然，那是构成部件，但是我们可以把多个 Promise 连接到一起以表示一系列异步步骤。

这种方式可以实现的关键在于以下两个 Promise 固有行为特性：

- 每次你对 Promise 调用 `then(..)`，它都会创建并返回一个新的 Promise，我们可以将其链接起来；
- 不管从 `then(..)` 调用的完成回调（第一个参数）返回的值是什么，它都会被自动设置为被链接 Promise（第一点中的）的完成。

先来解释一下这是什么意思，然后推导一下其如何帮助我们创建流程控制异步序列。考虑如下代码：

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

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

    // 用值42填充p2
    return v * 2;
} );

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

我们通过返回 `v * 2`（即 42），完成了第一个调用 `then(..)` 创建并返回的 promise `p2`。`p2` 的 `then(..)` 调用在运行时会从 `return v * 2` 语句接受完成值。当然，`p2.then(..)` 又创建了另一个新的 promise，可以用变量 `p3` 存储。

但是，如果必须创建一个临时变量 `p2`（或 `p3` 等），还是有一点麻烦的。谢天谢地，我们很容易把这些链接到一起：

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

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

    // 用值42完成连接的promise
    return v * 2;
} )
// 这里是链接的promise
.then( function(v){
    console.log( v );    // 42
} );
```

现在第一个 then(..) 就是异步序列中的第一步，第二个 then(..) 就是第二步。这可以一直任意扩展下去。只要保持把先前的 then(..) 连到自动创建的每一个 Promise 即可。

但这里还漏掉了一些东西。如果需要步骤 2 等待步骤 1 异步来完成一些事情怎么办？我们使用了立即返回 return 语句，这会立即完成链接的 promise。

使 Promise 序列真正能够在每一步有异步能力的关键是，回忆一下当传递给 Promise.resolve(..) 的是一个 Promise 或 thenable 而不是最终值时的运作方式。Promise.resolve(..) 会直接返回接收到的真正 Promise，或展开接收到的 thenable 值，并在持续展开 thenable 的同时递归地前进。

从完成（或拒绝）处理函数返回 thenable 或者 Promise 的时候也会发生同样的展开。考虑：

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

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

    // 创建一个promise并将其返回
    return new Promise( function(resolve,reject){
        // 用值42填充
        resolve( v * 2 );
    } );
} )
.then( function(v){
    console.log( v );       // 42
} );
```

虽然我们把 42 封装到了返回的 promise 中，但它仍然会被展开并最终成为链接的 promise 的决议，因此第二个 then(..) 得到的仍然是 42。如果我们向封装的 promise 引入异步，一切都仍然会同样工作：

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

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

```
        // 创建一个promise并返回
        return new Promise( function(resolve,reject){
            // 引入异步!
            setTimeout( function(){
                // 用值42填充
                resolve( v * 2 );
            }, 100 );
        } );

    } )
    .then( function(v){
        // 在前一步中的100ms延迟之后运行
        console.log( v );       // 42
    } );
```

这种强大实在不可思议！现在我们可以构建这样一个序列：不管我们想要多少个异步步骤，每一步都能够根据需要等待下一步（或者不等！）。

当然，在这些例子中，一步步传递的值是可选的。如果不显式返回一个值，就会隐式返回 `undefined`，并且这些 promise 仍然会以同样的方式链接在一起。这样，每个 Promise 的决议就成了继续下一个步骤的信号。

为了进一步阐释链接，让我们把延迟 Promise 创建（没有决议消息）过程一般化到一个工具中，以便在多个步骤中复用：

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

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

调用 `delay(200)` 创建了一个将在 200ms 后完成的 promise，然后我们从第一个 `then(..)` 完成回调中返回这个 promise，这会导致第二个 `then(..)` 的 promise 等待这个 200ms 的 promise。

如前所述，严格地说，这个交互过程中有两个 promise：200ms 延迟 promise，和第二个 `then(..)` 链接到的那个链接 promise。但是你可能已经发现了，在脑海中把这两个 promise 合二为一之后更好理解，因为 Promise 机制已经自动为你把它们的状态合并在了一起。这样一来，可以把 `return delay(200)` 看作是创建了一个 promise，并用其替换了前面返回的链接 promise。

但说实话，没有消息传递的延迟序列对于 Promise 流程控制来说并不是一个很有用的示例。我们来考虑如下这样一个更实际的场景。

这里不用定时器，而是构造 Ajax 请求：

```
// 假定工具ajax( {url}, {callback} )存在

// Promise-aware ajax
function request(url) {
    return new Promise( function(resolve,reject){
        // ajax(..)回调应该是我们这个promise的resolve(..)函数
        ajax( url, resolve );
    } );
}
```

我们首先定义一个工具 `request(..)`，用来构造一个表示 `ajax(..)` 调用完成的 promise：

```
request( "http://some.url.1/" )
.then( function(response1){
    return request( "http://some.url.2/?v=" + response1 );
} )
.then( function(response2){
    console.log( response2 );
} );
```

开发者常会遇到这样的情况：他们想要通过本身并不支持 Promise 的工具（就像这里的 `ajax(..)`，它接收的是一个回调）实现支持 Promise 的异步流程控制。虽然原生 ES6 `Promise` 机制并不会自动为我们提供这个模式，但所有实际的 Promise 库都会提供。通常它们把这个过程称为“提升”“promise 化”或者其他类似的名称。我们稍后会再介绍这种技术。

利用返回 Promise 的 `request(..)`，我们通过使用第一个 URL 调用它来创建链接中的第一步，并且把返回的 promise 与第一个 `then(..)` 链接起来。

`response1` 一返回，我们就使用这个值构造第二个 URL，并发出第二个 `request(..)` 调用。第二个 `request(..)` 的 promise 返回，以便异步流控制中的第三步等待这个 Ajax 调用完成。最后，`response2` 一返回，我们就立即打出结果。

我们构建的这个 Promise 链不仅是一个表达多步异步序列的流程控制，还是一个从一个步

骤到下一个步骤传递消息的消息通道。

如果这个 Promise 链中的某个步骤出错了怎么办？错误和异常是基于每个 Promise 的，这意味着可能在链的任意位置捕捉到这样的错误，而这个捕捉动作在某种程度上就相当于在这一位置将整条链“重置”回了正常运作：

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

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

    // 永远不会到达这里
    return request( "http://some.url.2/?v=" + response1 );
} )

// 步骤3:
.then(
    function fulfilled(response2){
        // 永远不会到达这里
    },

    // 捕捉错误的拒绝处理函数
    function rejected(err){
        console.log( err );
        // 来自foo.bar()的错误TypeError
        return 42;
    }
)

// 步骤4:
.then( function(msg){
    console.log( msg );          // 42
} );
```

第 2 步出错后，第 3 步的拒绝处理函数会捕捉到这个错误。拒绝处理函数的返回值（这段代码中是 42），如果有的话，会用来完成交给下一个步骤（第 4 步）的 promise，这样，这个链现在就回到了完成状态。

正如之前讨论过的，当从完成处理函数返回一个 promise 时，它会被展开并有可能延迟下一个步骤。从拒绝处理函数返回 promise 也是如此，因此如果在第 3 步返回的不是 42 而是一个 promise 的话，这个 promise 可能会延迟第 4 步。调用 `then(..)` 时的完成处理函数或拒绝处理函数如果抛出异常，都会导致（链中的）下一个 promise 因这个异常而立即被拒绝。

如果你调用 promise 的 `then(..)`，并且只传入一个完成处理函数，一个默认拒绝处理函数就会顶替上来：

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

var p2 = p.then(
    function fulfilled(){
        // 永远不会达到这里
    }
    // 假定的拒绝处理函数,如果省略或者传入任何非函数值
    // function(err) {
    //     throw err;
    // }
);
```

如你所见，默认拒绝处理函数只是把错误重新抛出，这最终会使得 p2（链接的 promise）用同样的错误理由拒绝。从本质上说，这使得错误可以继续沿着 Promise 链传播下去，直到遇到显式定义的拒绝处理函数。

稍后我们会介绍关于 Promise 错误处理的更多细节，因为还有其他一些微妙的细节需要考虑。

如果没有给 `then(..)` 传递一个适当有效的函数作为完成处理函数参数，还是会有作为替代的一个默认处理函数：

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

p.then(
    // 假设的完成处理函数,如果省略或者传入任何非函数值
    // function(v) {
    //       return v;
    // }
    null,
    function rejected(err){
        // 永远不会到达这里
    }
);
```

你可以看到，默认的完成处理函数只是把接收到的任何传入值传递给下一个步骤（Promise）而已。

`then(null,function(err){ .. })` 这个模式——只处理拒绝（如果有的话），但又把完成值传递下去——有一个缩写形式的 API：`catch(function(err){ .. })`。下一小节会详细介绍 `catch(..)`。

让我们来简单总结一下使链式流程控制可行的 Promise 固有特性。

- 调用 Promise 的 `then(..)` 会自动创建一个新的 Promise 从调用返回。
- 在完成或拒绝处理函数内部，如果返回一个值或抛出一个异常，新返回的（可链接的）Promise 就相应地决议。
- 如果完成或拒绝处理函数返回一个 Promise，它将会被展开，这样一来，不管它的决议值是什么，都会成为当前 `then(..)` 返回的链接 Promise 的决议值。

尽管链式流程控制是有用的，但是对其最精确的看法是把它看作 Promise 组合到一起的一个附加益处，而不是主要目的。正如前面已经多次深入讨论的，Promise 规范化了异步，并封装了时间相关值的状态，使得我们能够把它们以这种有用的方式链接到一起。

当然，相对于第 2 章讨论的回调的一团乱麻，链接的顺序表达（this-then-this-then-this...）已经是一个巨大的进步。但是，仍然有大量的重复样板代码（`then(..)` 以及 `function() { ... }`）。在第 4 章，我们将会看到在顺序流程控制表达方面提升巨大的优美模式，通过生成器实现。

术语：决议、完成以及拒绝

对于术语*决议*（resolve）、*完成*（fulfill）和*拒绝*（reject），在更深入学习 Promise 之前，我们还有一些模糊之处需要澄清。先来研究一下构造器 `Promise(..)`：

```
var p = new Promise( function(X,Y){
    // X()用于完成
    // Y()用于拒绝
} );
```

你可以看到，这里提供了两个回调（称为 `X` 和 `Y`）。第一个通常用于标识 Promise 已经完成，第二个总是用于标识 Promise 被拒绝。这个“通常”是什么意思呢？对于这些参数的精确命名，这又意味着什么呢？

追根究底，这只是你的用户代码和标识符名称，对引擎而言没有意义。所以从技术上说，这无关紧要，`foo(..)` 或者 `bar(..)` 还是同样的函数。但是，你使用的文字不只会影响你对这些代码的看法，也会影响团队其他开发者对代码的认识。错误理解精心组织起来的异步代码还不如使用一团乱麻的回调函数。

所以事实上，命名还是有一定的重要性的。

第二个参数名称很容易决定。几乎所有的文献都将其命名为 `reject(..)`，因为这就是它真实的（也是唯一的！）工作，所以这样的名字是很好的选择。我强烈建议大家要一直使用 `reject(..)` 这一名称。