类似。但是，Promise.resolve(..) 也会展开 thenable 值（前面已多次介绍）。在这种情况下，返回的 Promise 采用传入的这个 thenable 的最终决议值，可能是完成，也可能是拒绝：

```
var fulfilledTh = {
    then: function(cb) { cb( 42 ); }
};
var rejectedTh = {
    then: function(cb,errCb) {
        errCb( "Oops" );
    }
};

var p1 = Promise.resolve( fulfilledTh );
var p2 = Promise.resolve( rejectedTh );

// p1是完成的promise
// p2是拒绝的promise
```

还要记住，如果传入的是真正的 Promise，Promise.resolve(..) 什么都不会做，只会直接把这个值返回。所以，对你不了解属性的值调用 Promise.resolve(..)，如果它恰好是一个真正的 Promise，是不会有额外的开销的。

3.7.3 then(..) 和 catch(..)

每个 Promise 实例（不是 Promise API 命名空间）都有 then(..) 和 catch(..) 方法，通过这两个方法可以为这个 Promise 注册完成和拒绝处理函数。Promise 决议之后，立即会调用这两个处理函数之一，但不会两个都调用，而且总是异步调用（参见 1.5 节）。

then(..) 接受一个或两个参数：第一个用于完成回调，第二个用于拒绝回调。如果两者中的任何一个被省略或者作为非函数值传入的话，就会替换为相应的默认回调。默认完成回调只是把消息传递下去，而默认拒绝回调则只是重新抛出（传播）其接收到的出错原因。

就像刚刚讨论过的一样，catch(..) 只接受一个拒绝回调作为参数，并自动替换默认完成回调。换句话说，它等价于 then(null,..)：

```
p.then( fulfilled );

p.then( fulfilled, rejected );

p.catch( rejected ); // 或者p.then( null, rejected )
```

then(..) 和 catch(..) 也会创建并返回一个新的 promise，这个 promise 可以用于实现 Promise 链式流程控制。如果完成或拒绝回调中抛出异常，返回的 promise 是被拒绝的。如果任意一个回调返回非 Promise、非 thenable 的立即值，这个值会被用作返回 promise 的完成值。如果完成处理函数返回一个 promise 或 thenable，那么这个值会被展开，并作为返回 promise 的决议值。

3.7.4 Promise.all([..]) 和 Promise.race([..])

ES6 Promise API 静态辅助函数 Promise.all([..]) 和 Promise.race([..]) 都会创建一个 Promise 作为它们的返回值。这个 promise 的决议完全由传入的 promise 数组控制。

对 Promise.all([..]) 来说，只有传入的所有 promise 都完成，返回 promise 才能完成。如果有任何 promise 被拒绝，返回的主 promise 就立即会被拒绝（抛弃任何其他 promise 的结果）。如果完成的话，你会得到一个数组，其中包含传入的所有 promise 的完成值。对于拒绝的情况，你只会得到第一个拒绝 promise 的拒绝理由值。这种模式传统上被称为门：所有人都到齐了才开门。

对 Promise.race([..]) 来说，只有第一个决议的 promise（完成或拒绝）取胜，并且其决议结果成为返回 promise 的决议。这种模式传统上称为门闩：第一个到达者打开门闩通过。考虑：

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

Promise.race( [p1,p2,p3] )
.then( function(msg){
    console.log( msg );     // 42
} );

Promise.all( [p1,p2,p3] )
.catch( function(err){
    console.error( err );   // "Oops"

} );

Promise.all( [p1,p2] )
.then( function(msgs){
    console.log( msgs );    // [42,"Hello World"]
} );
```

当心！若向 Promise.all([..]) 传入空数组，它会立即完成，但 Promise.race([..]) 会挂住，且永远不会决议。

ES6 Promise API 非常简单直观。它至少足以处理最基本的异步情况，并且如果要重新整理，把代码从回调地狱解救出来的话，它也是一个很好的起点。

但是，应用常常会有很多更复杂的异步情况需要实现，而 Promise 本身对此在处理上具有局限性。下一节会深入探讨这些局限，理解 Promise 库出现的动机。

3.8 Promise 局限性

这一节讨论的许多细节本章之前都已经有所提及，不过我们还是一定要专门总结这些局限性才行。

3.8.1 顺序错误处理

本章前面已经详细介绍了适合 Promise 的错误处理。Promise 的设计局限性（具体来说，就是它们链接的方式）造成了一个让人很容易中招的陷阱，即 Promise 链中的错误很容易被无意中默默忽略掉。

关于 Promise 错误，还有其他需要考虑的地方。由于一个 Promise 链仅仅是连接到一起的成员 Promise，没有把整个链标识为一个个体的实体，这意味着没有外部方法可以用于观察可能发生的错误。

如果构建了一个没有错误处理函数的 Promise 链，链中任何地方的任何错误都会在链中一直传播下去，直到被查看（通过在某个步骤注册拒绝处理函数）。在这个特定的例子中，只要有一个指向链中最后一个 promise 的引用就足够了（下面代码中的 `p`），因为你可以在那里注册拒绝处理函数，而且这个处理函数能够得到所有传播过来的错误的通知：

```
// foo(..), STEP2(..)以及STEP3(..)都是支持promise的工具

var p = foo( 42 )
.then( STEP2 )
.then( STEP3 );
```

虽然这里可能有点鬼祟、令人迷惑，但是这里的 `p` 并不指向链中的第一个 promise（调用 `foo(42)` 产生的那一个），而是指向最后一个 promise，即来自调用 `then(STEP3)` 的那一个。

还有，这个 Promise 链中的任何一个步骤都没有显式地处理自身错误。这意味着你可以在 `p` 上注册一个拒绝错误处理函数，对于链中任何位置出现的任何错误，这个处理函数都会得到通知：

```
p.catch( handleErrors );
```

但是，如果链中的任何一个步骤事实上进行了自身的错误处理（可能以隐藏或抽象的不可见的方式），那你的 `handleErrors(..)` 就不会得到通知。这可能是你想要的——毕竟这是一个“已处理的拒绝”——但也可能并不是。完全不能得到（对任何“已经处理”的拒绝错误的）错误通知也是一个缺陷，它限制了某些用例的功能。

基本上，这等同于 `try..catch` 存在的局限：`try..catch` 可能捕获一个异常并简单地吞掉它。所以这并不是 Promise 独有的局限性，但可能是我们希望绕过的陷阱。

遗憾的是，很多时候并没有为 Promise 链序列的中间步骤保留的引用。因此，没有这样的引用，你就无法关联错误处理函数来可靠地检查错误。

3.8.2 单一值

根据定义，Promise 只能有一个完成值或一个拒绝理由。在简单的例子中，这不是什么问题，但是在更复杂的场景中，你可能就会发现这是一种局限了。

一般的建议是构造一个值封装（比如一个对象或数组）来保持这样的多个信息。这个解决方案可以起作用，但要在 Promise 链中的每一步都进行封装和解封，就十分丑陋和笨重了。

1. 分裂值

有时候你可以把这一点当作提示你可以 / 应该把问题分解为两个或更多 Promise 的信号。

设想你有一个工具 `foo(..)`，它可以异步产生两个值（`x` 和 `y`）：

```
function getY(x) {
    return new Promise( function(resolve,reject){
        setTimeout( function(){
            resolve( (3 * x) - 1 );
        }, 100 );
    } );
}

function foo(bar,baz) {
    var x = bar * baz;

    return getY( x )
    .then( function(y){
        // 把两个值封装到容器中
        return [x,y];
    } );
}

foo( 10, 20 )
.then( function(msgs){
    var x = msgs[0];
    var y = msgs[1];

    console.log( x, y );    // 200 599
} );
```

首先，我们重新组织一下 `foo(..)` 返回的内容，这样就不再需要把 `x` 和 `y` 封装到一个数组值中以通过 promise 传输。取而代之的是，我们可以把每个值封装到它自己的 promise：

```
function foo(bar,baz) {
    var x = bar * baz;

    // 返回两个promise
```

```
        return [
            Promise.resolve( x ),
            getY( x )
        ];
    }

    Promise.all(
        foo( 10, 20 )
    )
    .then( function(msgs){
        var x = msgs[0];
        var y = msgs[1];

        console.log( x, y );
    } );
```

一个 promise 数组真的要优于传递给单个 promise 的一个值数组吗？从语法的角度来说，这算不上是一个改进。

但是，这种方法更符合 Promise 的设计理念。如果以后需要重构代码把对 x 和 y 的计算分开，这种方法就简单得多。由调用代码来决定如何安排这两个 promise，而不是把这种细节放在 `foo(..)` 内部抽象，这样更整洁也更灵活。这里使用了 `Promise.all([ .. ])`，当然，这并不是唯一的选择。

2. 展开 / 传递参数

`var x = ..` 和 `var y = ..` 赋值操作仍然是麻烦的开销。我们可以在辅助工具中采用某种函数技巧（感谢 Reginald Braithwaite，推特：@raganwald）：

```
function spread(fn) {
    return Function.apply.bind( fn, null );
}

Promise.all(
    foo( 10, 20 )
)
.then(
    spread( function(x,y){
        console.log( x, y );    // 200 599
    } )
)
```

这样会好一点！当然，你可以把这个函数戏法在线化，以避免额外的辅助工具：

```
Promise.all(
    foo( 10, 20 )
)
.then( Function.apply.bind(
    function(x,y){
        console.log( x, y );    // 200 599
    },
```

```
        null
    ) );
```

这些技巧可能很灵巧，但 ES6 给出了一个更好的答案：解构。数组解构赋值形式看起来是这样的：

```
Promise.all(
    foo( 10, 20 )
)
.then( function(msgs){
    var [x,y] = msgs;

    console.log( x, y );     // 200 599
} );
```

不过最好的是，ES6 提供了数组参数解构形式：

```
Promise.all(
    foo( 10, 20 )
)
.then( function([x,y]){
    console.log( x, y );     // 200 599
} );
```

现在，我们符合了“每个 Promise 一个值”的理念，并且又将重复样板代码量保持在了最小！

关于 ES6 解构形式的更多信息，请参考本系列的《你不知道的 JavaScript（下卷）》的“ES & Beyond”部分。

3.8.3 单决议

Promise 最本质的一个特征是：Promise 只能被决议一次（完成或拒绝）。在许多异步情况中，你只会获取一个值一次，所以这可以工作良好。

但是，还有很多异步的情况适合另一种模式——一种类似于事件和 / 或数据流的模式。在表面上，目前还不清楚 Promise 能不能很好用于这样的用例，如果不是完全不可用的话。如果不在 Promise 之上构建显著的抽象，Promise 肯定完全无法支持多值决议处理。

设想这样一个场景：你可能要启动一系列异步步骤以响应某种可能多次发生的激励（就像是事件），比如按钮点击。

这样可能不会按照你的期望工作：

```
// click(..)把"click"事件绑定到一个DOM元素
// request(..)是前面定义的支持Promise的Ajax

var p = new Promise( function(resolve,reject){
    click( "#mybtn", resolve );
} );

p.then( function(evt){
    var btnID = evt.currentTarget.id;
    return request( "http://some.url.1/?id=" + btnID );
} )
.then( function(text){
    console.log( text );
} );
```

只有在你的应用只需要响应按钮点击一次的情况下，这种方式才能工作。如果这个按钮被点击了第二次的话，promise `p` 已经决议，因此第二个 `resolve(..)` 调用就会被忽略。

因此，你可能需要转化这个范例，为每个事件的发生创建一整个新的 Promise 链：

```
click( "#mybtn", function(evt){
    var btnID = evt.currentTarget.id;

    request( "http://some.url.1/?id=" + btnID )
    .then( function(text){
        console.log( text );
    } );
} );
```

这种方法可以工作，因为针对这个按钮上的每个 `"click"` 事件都会启动一整个新的 Promise 序列。

由于需要在事件处理函数中定义整个 Promise 链，这很丑陋。除此之外，这个设计在某种程度上破坏了关注点与功能分离（SoC）的思想。你很可能想要把事件处理函数的定义和对事件的响应（那个 Promise 链）的定义放在代码中的不同位置。如果没有辅助机制的话，在这种模式下很难这样实现。

另外一种清晰展示这种局限性的方法是：如果能够构建某种“可观测量”（observable），可以将一个 Promise 链对应到这个“可观测量”就好了。有一些库已经创建了这样的抽象（比如 RxJS，http://rxjs.codeplex.com），但是这种抽象看起来非常笨重，以至于你甚至已经看不到任何 Promise 本身的特性。这样厚重的抽象带来了一些需要考虑的重要问题，比如这些机制（无 Promise）是否像 Promise 本身设计的那样可以信任。附录 B 会再次讨论这种“可观测量”模式。

3.8.4 惯性

要在你自己的代码中开始使用 Promise 的话，一个具体的障碍是，现存的所有代码都还不理解 Promise。如果你已经有大量的基于回调的代码，那么保持编码风格不变要简单得多。

“运动状态（使用回调的）的代码库会一直保持运动状态（使用回调的），直到受到一位聪明的、理解 Promise 的开发者的作用。”

Promise 提供了一种不同的范式，因此，编码方式的改变程度从某处的个别差异到某种情况下的截然不同都有可能。你需要刻意的改变，因为 Promise 不会从目前的编码方式中自然而然地衍生出来。

考虑如下的类似基于回调的场景：

```
function foo(x,y,cb) {
    ajax(
        "http://some.url.1/?x=" + x + "&y=" + y,
        cb
    );
}

foo( 11, 31, function(err,text) {
    if (err) {
        console.error( err );
    }
    else {
        console.log( text );
    }
} );
```

能够很快明显看出要把这段基于回调的代码转化为基于 Promise 的代码应该从哪些步骤开始吗？这要视你的经验而定。实践越多，越会觉得得心应手。但可以确定的是，Promise 并没有明确表示要如何实现转化。没有放之四海皆准的答案，责任还是在你的身上。

如前所述，我们绝对需要一个支持 Promise 而不是基于回调的 Ajax 工具，可以称之为 `request(..)`。你可以实现自己的版本，就像我们所做的一样。但是，如果不得不为每个基于回调的工具手工定义支持 Promise 的封装，这样的开销会让你不太可能选择支持 Promise 的重构。

Promise 没有为这个局限性直接提供答案。多数 Promise 库确实提供辅助工具，但即使没有库，也可以考虑如下的辅助工具：

```
// polyfill安全的guard检查
if (!Promise.wrap) {
    Promise.wrap = function(fn) {
        return function() {
            var args = [].slice.call( arguments );
```

```
            return new Promise( function(resolve,reject){
                fn.apply(
                    null,
                    args.concat( function(err,v){
                        if (err) {
                            reject( err );
                        }
                        else {
                            resolve( v );
                        }
                    } )
                );
            } );
        };
    };
}
```

好吧，这不只是一个简单的小工具。然而，尽管它看起来有点令人生畏，但是实际上并不像你想的那么糟糕。它接受一个函数，这个函数需要一个 error-first 风格的回调作为第一个参数，并返回一个新的函数。返回的函数自动创建一个 Promise 并返回，并替换回调，连接到 Promise 完成或拒绝。

与其花费太多时间解释这个 `Promise.wrap(..)` 辅助工具的工作原理，还不如直接看看其使用方式：

```
var request = Promise.wrap( ajax );

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

哇，非常简单！

`Promise.wrap(..)` 并不产出 Promise。它产出的是一个将产生 Promise 的函数。在某种意义上，产生 Promise 的函数可以看作是一个 Promise 工厂。我提议将其命名为“promisory”（“Promise”+“factory”）。

把需要回调的函数封装为支持 Promise 的函数，这个动作有时被称为“提升”或“Promise 工厂化”。但是，对于得到的结果函数来说，除了“被提升函数”似乎就没有什么标准术语可称呼了。所以我更喜欢“promisory”这个词，我认为它的描述更准确。

promisory 并不是编造的。它是一个真实的单词，意思是包含或传输一个 promise。这正是这些函数所做的，所以这个术语与其意义匹配得很完美。

于是，Promise.wrap(ajax) 产生了一个 ajax(..) promisory，我们称之为 request(..)。这个 promisory 为 Ajax 响应生成 Promise。

如果所有函数都已经是 promisory，我们就不需要自己构造了，所以这个额外的步骤有点可惜。但至少这个封装模式（通常）是重复的，所以我们可以像前面展示的那样把它放入 Promise.wrap(..) 辅助工具，以帮助我们的 promise 编码。

所以，回到前面的例子，我们需要为 ajax(..) 和 foo(..) 都构造一个 promisory：

```
// 为ajax(..)构造一个promisory
var request = Promise.wrap( ajax );

// 重构foo(..),但使其外部成为基于外部回调的,
// 与目前代码的其他部分保持通用
// ——只在内部使用 request(..)的promise
function foo(x,y,cb) {
    request(
        "http://some.url.1/?x=" + x + "&y=" + y
    )
    .then(
        function fulfilled(text){
            cb( null, text );
        },
        cb
    );
}

// 现在,为了这段代码的目的,为foo(..)构造一个 promisory
var betterFoo = Promise.wrap( foo );

// 并使用这个promisory
betterFoo( 11, 31 )
.then(
    function fulfilled(text){
        console.log( text );
    },
    function rejected(err){
        console.error( err );
    }
);
```

当然，尽管我们在重构 foo(..) 以使用新的 request(..) promisory，但是也可以使 foo(..) 本身成为一个 promisory，而不是保持基于回调的形式并需要构建和使用后续的 betterFoo(..) promisory。这个决策就取决于 foo(..) 是否需要保持与代码库中其他部分兼容的基于回调的形式。

考虑：

```
现在foo(..)也是一个promisory,因为它委托了request(..) promisory
function foo(x,y) {
```

```
    return request(
        "http://some.url.1/?x=" + x + "&y=" + y
    );
}

foo( 11, 31 )
.then( .. )
..
```

尽管原生 ES6 Promise 并没有提供辅助函数用于这样的 promisory 封装，但多数库都提供了这样的支持，或者你也可以构建自己的辅助函数。不管采用何种方式，解决 Promise 这个特定的限制都不需要太多代价（可对比回调地狱给我们带来的痛苦！）。

3.8.5 无法取消的 Promise

一旦创建了一个 Promise 并为其注册了完成和 / 或拒绝处理函数，如果出现某种情况使得这个任务悬而未决的话，你也没有办法从外部停止它的进程。

很多 Promise 抽象库提供了工具来取消 Promise，但这个思路很可怕！很多开发者希望 Promise 的原生设计就具有外部取消功能，但问题是，这可能会使 Promise 的一个消费者或观察者影响其他消费者查看这个 Promise。这违背了未来值的可信任性（外部不变性），但更坏的是，这是“远隔作用”（action at a distance）反模式的体现（http://en.wikipedia.org/wiki/Action_at_a_distance_%28computer_programming%29）。不管看起来如何有用，这实际上会导致你重陷与使用回调同样的噩梦。

考虑前面的 Promise 超时场景：

```
var p = foo( 42 );

Promise.race( [
    p,
    timeoutPromise( 3000 )
] )
.then(
    doSomething,
    handleError
);

p.then( function(){
    // 即使在超时的情况下也会发生 :(
} );
```

这个“超时”相对于 promise `p` 是外部的，所以 `p` 本身还会继续运行，这一点可能并不是我们所期望的。

一种选择是侵入式地定义你自己的决议回调：

```
var OK = true;

var p = foo( 42 );

Promise.race( [
    p,
    timeoutPromise( 3000 )
    .catch( function(err){
        OK = false;
        throw err;
    } )
] )
.then(
    doSomething,
    handleError
);

p.then( function(){
    if (OK) {
        // 只在没有超时情况下才会发生 :)
    }
} );
```

这很丑陋。它可以工作，但是离理想实现还差很远。一般来说，应避免这样的情况。

但如果没法避免的话，这个解决方案的丑陋应该是一个线索，它提示取消这个功能属于 Promise 之上更高级的抽象。我建议你应查看 Promise 抽象库以获得帮助，而不是 hack 自己的版本。

我的 Promise 抽象库 asynquence 提供了这样一个抽象，还有一个为序列提供的 `abort()` 功能，这些内容都会在本部分的附录 A 中讨论。

单独的一个 Promise 并不是一个真正的流程控制机制（至少不是很有意义），这正是取消所涉及的层次（流程控制）。这就是为什么 Promise 取消总是让人感觉很别扭。

相比之下，集合在一起的 Promise 构成的链，我喜欢称之为一个“序列”，就是一个流程控制的表达，因此将取消定义在这个抽象层次上是合适的。

单独的 Promise 不应该可取消，但是取消一个可序列是合理的，因为你不会像对待 Promise 那样把序列作为一个单独的不变值来传送。

3.8.6 Promise 性能

这个特定的局限性既简单又复杂。

把基本的基于回调的异步任务链与 Promise 链中需要移动的部分数量进行比较。很显然，Promise 进行的动作要多一些，这自然意味着它也会稍慢一些。请回想 Promise 提供的信任保障列表，再与你要在回调之上建立同样的保护自建的解决方案来比较一下。

更多的工作，更多的保护。这些意味着 Promise 与不可信任的裸回调相比会更慢一些。这是显而易见的，也很容易理解。

但会慢多少呢？呃，实际上，要精确回答这个问题极其困难。

坦白地说，这有点像是拿苹果和桔子相比，所以这可能就是一个错误的问题。实际上，应该比较的是提供了同样保护的手工自建回调系统是否能够快于 Promise 实现。

如果说 Promise 确实有一个真正的性能局限的话，那就是它们没有真正提供可信任性保护支持的列表以供选择（你总是得到全部）。

虽然如此，如果我们承认 Promise 通常要比其非 Promise、非可信任回调的等价系统稍微慢一点（假定有些情况下你认为可以接受可信任性的缺乏），这是否意味着应该完全避免 Promise，就好像你整个应用的唯一驱动力就是必须采用尽可能快的代码呢？

合理性检查：如果你的代码有合理的理由这样要求，那么 JavaScript 是否真的是实现这样任务的正确语言呢？我们可以优化 JavaScript，使其高性能运行应用（参见第 5 章和第 6 章）。但是，耿耿于 Promise 微小的性能损失而无视它提供的所有优点，真的合适吗？

另外一个微妙的问题是：Promise 使所有一切都成为异步的了，即有一些立即（同步）完成的步骤仍然会延迟到任务的下一步（参见第 1 章）。这意味着一个 Promise 任务序列可能比完全通过回调连接的同样的任务序列运行得稍慢一点。

当然，这里的问题是：本章介绍的 Promise 的这些优点是否值得付出这些微小的性能损失。

我的观点是：几乎所有那些你可能认为 Promise 性能会慢到需要担心的情况，实际上都是通过绕开 Promise 可信任性和可组合性优化掉了它们带来的好处的反模式。

取而代之的是，在默认情况下，你应该在代码中使用它们，然后对你应用的热路径进行性能分析。Promise 真的是性能瓶颈呢，还是只有理论上的性能下降呢？只有这样，具备了真实有效的性能测评（参见第 6 章），在这些识别出来的关键区域分离出 Promise 才是审慎负责的。

Promise 稍慢一些，但是作为交换，你得到的是大量内建的可信任性、对 Zalgo 的避免以及可组合性。可能局限性实际上并不是它们的真实表现，而是你缺少发现其好处的眼光呢？

3.9 小结

Promise 非常好，请使用。它们解决了我们因只用回调的代码而备受困扰的*控制反转*问题。

它们并没有摈弃回调，只是把回调的安排转交给了一个位于我们和其他工具之间的可信任的中介机制。

Promise 链也开始提供（尽管并不完美）以顺序的方式表达异步流的一个更好的方法，这有助于我们的大脑更好地计划和维护异步 JavaScript 代码。我们将在第 4 章看到针对这个问题的一种更好的解决方案！

第 4 章

生成器

在第 2 章里，我们确定了用回调表达异步控制流程的两个关键缺陷：

- 基于回调的异步不符合大脑对任务步骤的规划方式；
- 由于控制反转，回调并不是可信任或可组合的。

在第 3 章里，我们详细介绍了 Promise 如何把回调的控制反转反转回来，恢复了可信任性 / 可组合性。

现在我们把注意力转移到一种顺序、看似同步的异步流程控制表达风格。使这种风格成为可能的“魔法”就是 ES6 生成器（generator）。

4.1 打破完整运行

在第 1 章中，我们解释了 JavaScript 开发者在代码中几乎普遍依赖的一个假定：一个函数一旦开始执行，就会运行到结束，期间不会有其他代码能够打断它并插入其间。

可能看起来似乎有点奇怪，不过 ES6 引入了一个新的函数类型，它并不符合这种运行到结束的特性。这类新的函数被称为生成器。

考虑如下这个例子来了解其含义：

```
var x = 1;

function foo() {
    x++;
    bar();              // <-- 这一行是什么作用?
```