```
        }
        else {
            console.log( data );
        }
    }

    ajax( "http://some.url.1", timeoutify( foo, 500 ) );
```

还有一个信任问题是调用过早。在特定应用的术语中，这可能实际上是指在某个关键任务完成之前调用回调。但是更通用地来说，对于既可能在*现在*（同步）也可能在*将来*（异步）调用你的回调的工具来说，这个问题是明显的。

这种由同步或异步行为引起的不确定性几乎总会带来极大的 bug 追踪难度。在某些圈子里，人们用虚构的十分疯狂的恶魔 Zalgo 来描述这种同步 / 异步噩梦。常常会有“不要放出 Zalgo”这样的呼喊，而这也引出了一条非常有效的建议：永远异步调用回调，即使就在事件循环的下一轮，这样，所有回调就都是可预测的异步调用了。

关于 Zalgo 的更多信息，可以参考 Oren Golan 的“Don’t Release Zalgo!”（https://github.com/oren/oren.github.io/blob/master/posts/zalgo.md）以及 Issac Z. Schlueter 的“Designing APIs for Asynchrony”（http://blog.izs.me/post/59142742143/designing-apis-for-asynchrony）。

考虑：

```
function result(data) {
    console.log( a );
}

var a = 0;

ajax( "..pre-cached-url..", result );
a++;
```

这段代码会打印出 `0`（同步回调调用）还是 `1`（异步回调调用）呢？这要视情况而定。

你可以看出 Zalgo 的不确定性给 JavaScript 程序带来的威胁。所以听上去有点傻的“不要放出 Zalgo”实际上十分常用，并且也是有用的建议。永远要异步。

如果你不确定关注的 API 会不会永远异步执行怎么办呢？可以创建一个类似于这个“验证概念”版本的 `asyncify(..)` 工具：

```
function asyncify(fn) {
    var orig_fn = fn,
        intv = setTimeout( function(){
            intv = null;
            if (fn) fn();
        }, 0 )
```

```
    ;

    fn = null;

    return function() {
        // 触发太快,在定时器intv触发指示异步转换发生之前?
        if (intv) {
            fn = orig_fn.bind.apply(
                orig_fn,
                 // 把封装器的this添加到bind(..)调用的参数中,
                 // 以及克里化(currying)所有传入参数
                 [this].concat( [].slice.call( arguments ) )
             );
         }
         // 已经是异步
         else {
             // 调用原来的函数
             orig_fn.apply( this, arguments );
         }
     };
}
```

可以像这样使用 `asyncify(..)`：

```
function result(data) {
    console.log( a );
}

var a = 0;

ajax( "..pre-cached-url..", asyncify( result ) );
a++;
```

不管这个 Ajax 请求已经在缓存中并试图对回调立即调用，还是要从网络上取得，进而在将来异步完成，这段代码总是会输出 `1`，而不是 `0`——`result(..)` 只能异步调用，这意味着 `a++` 有机会在 `result(..)` 之前运行。

好啊，又“解决”了一个信任问题！但这是低效的，而且也会带来膨胀的重复代码，使你的项目变得笨重。

这就是回调的故事，讲了一遍又一遍。它们可以实现所有你想要的功能，但是你需要努力才行。这些努力通常比你追踪这样的代码能够或者应该付出的要多得多。

可能现在你希望有内建的 API 或其他语言机制来解决这些问题。最终，ES6 带着一些极好的答案登场了，所以，继续读下去吧！

2.5 小结

回调函数是 JavaScript 异步的基本单元。但是随着 JavaScript 越来越成熟，对于异步编程领

域的发展，回调已经不够用了。

第一，大脑对于事情的计划方式是线性的、阻塞的、单线程的语义，但是回调表达异步流程的方式是非线性的、非顺序的，这使得正确推导这样的代码难度很大。难于理解的代码是坏代码，会导致坏 bug。

我们需要一种更同步、更顺序、更阻塞的的方式来表达异步，就像我们的大脑一样。

第二，也是更重要的一点，回调会受到控制反转的影响，因为回调暗中把控制权交给第三方（通常是不受你控制的第三方工具！）来调用你代码中的 continuation。这种控制转移导致一系列麻烦的信任问题，比如回调被调用的次数是否会超出预期。

可以发明一些特定逻辑来解决这些信任问题，但是其难度高于应有的水平，可能会产生更笨重、更难维护的代码，并且缺少足够的保护，其中的损害要直到你受到 bug 的影响才会被发现。

我们需要一个通用的方案来解决这些信任问题。不管我们创建多少回调，这一方案都应可以复用，且没有重复代码的开销。

我们需要比回调更好的机制。到目前为止，回调提供了很好的服务，但是未来的 JavaScript 需要更高级、功能更强大的异步模式。本书接下来的几章会深入探讨这些新型技术。

第 3 章

Promise

在第 2 章里，我们确定了通过回调表达程序异步和管理并发的两个主要缺陷：缺乏顺序性和可信任性。既然已经对问题有了充分的理解，那么现在是时候把注意力转向可以解决这些问题的模式了。

我们首先想要解决的是控制反转问题，其中，信任很脆弱，也很容易失去。

回忆一下，我们用回调函数来封装程序中的 continuation，然后把回调交给第三方（甚至可能是外部代码），接着期待其能够调用回调，实现正确的功能。

通过这种形式，我们要表达的意思是："这是将来要做的事情，要在当前的步骤完成之后发生。"

但是，如果我们能够把控制反转再反转回来，会怎样呢？如果我们不把自己程序的 continuation 传给第三方，而是希望第三方给我们提供了解其任务何时结束的能力，然后由我们自己的代码来决定下一步做什么，那将会怎样呢？

这种范式就称为 Promise。

随着开发者和规范撰写者绝望地清理他们的代码和设计中由回调地狱引发的疯狂行为，Promise 风暴已经开始席卷 JavaScript 世界。

实际上，绝大多数 JavaScript/DOM 平台新增的异步 API 都是基于 Promise 构建的。所以学习研究 Promise 应该是个好主意，你以为如何呢？！

本章经常会使用“立即”一词，通常用来描述某个 Promise 决议（resolution）动作。但是，基本上在所有情况下，这个“立即”指任务队列行为（参见第 1 章）方面的意义，而不是指严格同步的*现在*。

3.1 什么是 Promise

开发人员在学习新技术或新模式时，通常第一步就是“给我看看代码”。对我们来说，先跳进去学习细节是很自然的。

但是，事实证明，只了解 API 会丢失很多抽象的细节。Promise 属于这样一类工具：通过某人使用它的方式，很容易分辨他是真正理解了这门技术，还是仅仅学习和使用 API 而已。

所以，在展示 Promise 代码之前，我想先从概念上完整地解释 Promise 到底是什么。希望这能够更好地指导你今后将 Promise 理论集成到自己的异步流中。

明确这一点之后，我们先来查看一下关于 Promise 定义的两个不同类比。

3.1.1 未来值

设想一下这样一个场景：我走到快餐店的柜台，点了一个芝士汉堡。我交给收银员 1.47 美元。通过下订单并付款，我已经发出了一个对某个值（就是那个汉堡）的请求。我已经启动了一次交易。

但是，通常我不能马上就得到这个汉堡。收银员会交给我某个东西来代替汉堡：一张带有订单号的收据。订单号就是一个 IOU（I owe you，我欠你的）*承诺*（promise），保证了最终我会得到我的汉堡。

所以我得好好保留我的收据和订单号。我知道这代表了我未来的汉堡，所以不需要担心，只是现在我还是很饿！

在等待的过程中，我可以做点其他的事情，比如给朋友发个短信：“嗨，要来和我一起吃午饭吗？我正要吃芝士汉堡。”

我已经在想着未来的芝士汉堡了，尽管现在我还没有拿到手。我的大脑之所以可以这么做，是因为它已经把订单号当作芝士汉堡的占位符了。从本质上讲，这个占位符使得这个值不再依赖时间。这是一个*未来值*。

终于，我听到服务员在喊“订单 113”，然后愉快地拿着收据走到柜台，把收据交给收银员，换来了我的芝士汉堡。

换句话说，一旦我需要的值准备好了，我就用我的承诺值（value-promise）换取这个值本身。

但是，还可能有另一种结果。他们叫到了我的订单号，但当我过去拿芝士汉堡的时候，收银员满是歉意地告诉我：“不好意思，芝士汉堡卖完了。”除了作为顾客对这种情况感到愤怒之外，我们还可以看到*未来值*的一个重要特性：它可能成功，也可能失败。

每次点芝士汉堡，我都知道最终要么得到一个芝士汉堡，要么得到一个汉堡包售罄的坏消息，那我就得找点别的当午饭了。

在代码中，事情并非这么简单。这是因为，用类比的方式来说就是，订单号可能永远不会被叫到。在这种情况下，我们就永远处于一种未决议状态。后面会讨论如何处理这种情况。

1. 现在值与将来值

要把以上内容应用到代码里的话，前面的描述有点过于抽象，所以这里再具体说明一下。

但在具体解释 Promise 的工作方式之前，先来推导通过我们已经理解的方式——回调——如何处理*未来值*。

当编写代码要得到某个值的时候，比如通过数学计算，不管你有没有意识到，你都已经对这个值做出了一些非常基本的假设，那就是，它已经是一个具体的*现在值*：

```
var x, y = 2;

console.log( x + y ); // NaN  <-- 因为x还没有设定
```

运算 `x + y` 假定了 `x` 和 `y` 都已经设定。用术语简单地解释就是，这里我们假定 `x` 和 `y` 的值都是已决议的。

期望运算符 `+` 本身能够神奇地检测并等待 `x` 和 `y` 都决议好（也就是准备好）再进行运算是没有意义的。如果有的语句*现在*完成，而有的语句*将来*完成，那就会在程序里引起混乱，对不对？

如果两条语句的任何一个（或全部）可能还没有完成，你怎么可能追踪这两条语句的关系呢？如果语句 2 依赖于语句 1 的完成，那么就只有两个输出：要么语句 1 马上完成，一切顺利执行；要么语句 1 还未完成，语句 2 因此也将会失败。

学完第 1 章之后，如果这种情况你听起来很熟悉的话，非常好！

让我们回到 `x + y` 这个算术运算。设想如果可以通过一种方式表达：“把 `x` 和 `y` 加起来，但如果它们中的任何一个还没有准备好，就等待两者都准备好。一旦可以就马上执行加

运算。”

可能你已经想到了回调。好吧，那么……

```
function add(getX,getY,cb) {
    var x, y;
    getX( function(xVal){
        x = xVal;
        // 两个都准备好了?
        if (y != undefined) {
            cb( x + y );        // 发送和
        }
    } );
    getY( function(yVal){
        y = yVal;
        // 两个都准备好了?
        if (x != undefined) {
            cb( x + y );        // 发送和
        }
    } );
}

// fetchX() 和fetchY()是同步或者异步函数
add( fetchX, fetchY, function(sum){
    console.log( sum ); // 是不是很容易?
} );
```

先暂停片刻，认真思考一下这段代码的优美度（或缺少优美度，别急着喝彩）。

尽管其中的丑陋不可否认，但这种异步模式体现出了一些非常重要的东西。

在这段代码中，我们把 x 和 y 当作未来值，并且表达了一个运算 add(..)。这个运算（从外部看）不在意 x 和 y 现在是否都已经可用。换句话说，它把*现在*和*将来*归一化了，因此我们可以确保这个 add(..) 运算的输出是可预测的。

通过使用这个时间上一致的 add(..)——从*现在*到*将来*的时间，它的行为都是一致的——大大简化了对这段异步代码的追踪。

说得更直白一些就是，为了统一处理*现在*和*将来*，我们把它们都变成了*将来*，即所有的操作都成了异步的。

当然，这个粗糙的基于回调的方法还有很多不足。要体会追踪未来值的益处而不需要考虑其在时间方面是否可用，这只是很小的第一步。

2. Promise 值

本章后面一定会深入介绍很多 Promise 的细节，因此这里如果读起来有些困惑的话，不必担心。我们先来大致看一下如何通过 Promise 函数表达这个 x + y 的例子：

```
function add(xPromise,yPromise) {
    // Promise.all([ .. ])接受一个promise数组并返回一个新的promise,
    // 这个新promise等待数组中的所有promise完成
    return Promise.all( [xPromise, yPromise] )

    // 这个promise决议之后,我们取得收到的X和Y值并加在一起
    .then( function(values){
        // values是来自于之前决议的promisei的消息数组
        return values[0] + values[1];
    } );
}

// fetchX()和fetchY()返回相应值的promise,可能已经就绪,
// 也可能以后就绪
add( fetchX(), fetchY() )

// 我们得到一个这两个数组的和的promise
// 现在链式调用 then(..)来等待返回promise的决议
.then( function(sum){
    console.log( sum ); // 这更简单!
} );
```

这段代码中有两层 Promise。

`fetchX()` 和 `fetchY()` 是直接调用的，它们的返回值（promise！）被传给 `add(..)`。这些 promise 代表的底层值的可用时间可能是*现在*或*将来*，但不管怎样，promise 归一保证了行为的一致性。我们可以按照不依赖于时间的方式追踪值 `X` 和 `Y`。它们是未来值。

第二层是 `add(..)`（通过 `Promise.all([ .. ])`）创建并返回的 promise。我们通过调用 `then(..)` 等待这个 promise。`add(..)` 运算完成后，未来值 `sum` 就准备好了，可以打印出来。我们把等待未来值 `X` 和 `Y` 的逻辑隐藏在了 `add(..)` 内部。

在 `add(..)` 内部，`Promise.all([ .. ])` 调用创建了一个 promise（这个 promise 等待 `promiseX` 和 `promiseY` 的决议）。链式调用 `.then(..)` 创建了另外一个 promise。这个 promise 由 `return values[0] + values[1]` 这一行立即决议（得到加运算的结果）。因此，链 `add(..)` 调用终止处的调用 `then(..)`——在代码结尾处——实际上操作的是返回的第二个 promise，而不是由 `Promise.all([ .. ])` 创建的第一个 promise。还有，尽管第二个 `then(..)` 后面没有链接任何东西，但它实际上也创建了一个新的 promise，如果想要观察或者使用它的话就可以看到。本章后面会详细介绍这种 Promise 链。

就像芝士汉堡订单一样，Promise 的决议结果可能是拒绝而不是完成。拒绝值和完成的 Promise 不一样：完成值总是编程给出的，而拒绝值，通常称为*拒绝原因*（rejection reason），可能是程序逻辑直接设置的，也可能是从运行异常隐式得出的值。

通过 Promise，调用 then(..) 实际上可以接受两个函数，第一个用于完成情况（如前所示），第二个用于拒绝情况：

```
add( fetchX(), fetchY() )
.then(
    // 完成处理函数
    function(sum) {
        console.log( sum );
    },
    // 拒绝处理函数
    function(err) {
        console.error( err ); // 烦!
    }
);
```

如果在获取 X 或 Y 的过程中出错，或者在加法过程中出错，add(..) 返回的就是一个被拒绝的 promise，传给 then(..) 的第二个错误处理回调就会从这个 promise 中得到拒绝值。

从外部看，由于 Promise 封装了依赖于时间的状态——等待底层值的完成或拒绝，所以 Promise 本身是与时间无关的。因此，Promise 可以按照可预测的方式组成（组合），而不用关心时序或底层的结果。

另外，一旦 Promise 决议，它就永远保持在这个状态。此时它就成为了*不变值*（immutable value），可以根据需求多次查看。

Promise 决议后就是外部不可变的值，我们可以安全地把这个值传递给第三方，并确信它不会被有意无意地修改。特别是对于多方查看同一个 Promise 决议的情况，尤其如此。一方不可能影响另一方对 Promise 决议的观察结果。不可变性听起来似乎一个学术话题，但实际上这是 Promise 设计中最基础和最重要的因素，我们不应该随意忽略这一点。

这是关于 Promise 需要理解的最强大也最重要的一个概念。经过大量的工作，你本可以通过丑陋的回调组合专门创建出类似的效果，但这真的不是一个有效的策略，特别是你不得不一次又一次重复操作。

Promise 是一种封装和组合未来值的易于复用的机制。

3.1.2 完成事件

如前所述，单独的 Promise 展示了未来值的特性。但是，也可以从另外一个角度看待 Promise 的决议：一种在异步任务中作为两个或更多步骤的流程控制机制，时序上的 this-then-that。

假定要调用一个函数 foo(..) 执行某个任务。我们不知道也不关心它的任何细节。这个函

数可能立即完成任务，也可能需要一段时间才能完成。

我们只需要知道 foo(..) 什么时候结束，这样就可以进行下一个任务。换句话说，我们想要通过某种方式在 foo(..) 完成的时候得到通知，以便可以继续下一步。

在典型的 JavaScript 风格中，如果需要侦听某个通知，你可能就会想到事件。因此，可以把对通知的需求重新组织为对 foo(..) 发出的一个完成事件（completion event，或 continuation 事件）的侦听。

是叫完成事件还是叫 continuation 事件，取决于你的视角。你是更关注 foo(..) 发生了什么，还是更关注 foo(..) 之后发生了什么？两种视角都是合理有用的。事件通知告诉我们 foo(..) 已经完成，也告诉我们现在可以继续进行下一步。确实，传递过去的回调将在事件通知发生时被调用，这个回调本身之前就是我们之前所说的 continuation。完成事件关注 foo(..) 更多一些，这也是目前主要的关注点，所以在后面的内容中，我们将其称为完成事件。

使用回调的话，通知就是任务（foo(..)）调用的回调。而使用 Promise 的话，我们把这个关系反转了过来，侦听来自 foo(..) 的事件，然后在得到通知的时候，根据情况继续。

首先，考虑以下伪代码：

```
foo(x) {
    // 开始做点可能耗时的工作
}

foo( 42 )

on (foo "completion") {
    // 可以进行下一步了!
}

on (foo "error") {
    // 啊,foo(..)中出错了
}
```

我们调用 foo(..)，然后建立了两个事件侦听器，一个用于 "completion"，一个用于 "error"——foo(..) 调用的两种可能结果。从本质上讲，foo(..) 并不需要了解调用代码订阅了这些事件，这样就很好地实现了关注点分离。

遗憾的是，这样的代码需要 JavaScript 环境提供某种魔法，而这种环境并不存在（实际上也有点不实际）。以下是在 JavaScript 中更自然的表达方法：

```
function foo(x) {
    // 开始做点可能耗时的工作
```

```
    // 构造一个listener事件通知处理对象来返回

    return listener;
}

var evt = foo( 42 );

evt.on( "completion", function(){
    // 可以进行下一步了!
} );

evt.on( "failure", function(err){
    // 啊,foo(..)中出错了
} );
```

`foo(..)` 显式创建并返回了一个事件订阅对象，调用代码得到这个对象，并在其上注册了两个事件处理函数。

相对于面向回调的代码，这里的反转是显而易见的，而且这也是有意为之。这里没有把回调传给 `foo(..)`，而是返回一个名为 `evt` 的事件注册对象，由它来接受回调。

如果你回想一下第 2 章的话，应该还记得回调本身就表达了一种控制反转。所以对回调模式的反转实际上是对反转的反转，或者称为*反控制反转*——把控制返还给调用代码，这也是我们最开始想要的效果。

一个很重要的好处是，可以把这个事件侦听对象提供给代码中多个独立的部分；在 `foo(..)` 完成的时候，它们都可以独立地得到通知，以执行下一步：

```
var evt = foo( 42 );

// 让bar(..)侦听foo(..)的完成
bar( evt );

// 并且让baz(..)侦听foo(..)的完成
baz( evt );
```

对控制反转的恢复实现了更好的关注点分离，其中 `bar(..)` 和 `baz(..)` 不需要牵扯到 `foo(..)` 的调用细节。类似地，`foo(..)` 不需要知道或关注 `bar(..)` 和 `baz(..)` 是否存在，或者是否在等待 `foo(..)` 的完成通知。

从本质上说，`evt` 对象就是分离的关注点之间一个中立的第三方协商机制。

Promise “事件”

你可能已经猜到，事件侦听对象 `evt` 就是 Promise 的一个模拟。

在基于 Promise 的方法中，前面的代码片段会让 `foo(..)` 创建并返回一个 Promise 实例，而且这个 Promise 会被传递到 `bar(..)` 和 `baz(..)`。

我们侦听的 Promise 决议“事件”严格说来并不算是事件（尽管它们实现目标的行为方式确实很像事件），通常也不叫作 "completion" 或 "error"。事实上，我们通过 then(..) 注册一个 "then" 事件。或者可能更精确地说，then(..) 注册 "fullfillment" 和 / 或 "rejection" 事件，尽管我们并不会在代码中直接使用这些术语。

考虑：

```
function foo(x) {
    // 可是做一些可能耗时的工作

    // 构造并返回一个promise
    return new Promise( function(resolve,reject){
        // 最终调用resolve(..)或者reject(..)
        // 这是这个promise的决议回调
    } );

}

var p = foo( 42 );

bar( p );

baz( p );
```

new Promise(function(..){ .. }) 模式通常称为 revealing constructor（http://domenic.me/2014/02/13/the-revealing-constructor-pattern/）。传入的函数会立即执行（不会像 then(..) 中的回调一样异步延迟），它有两个参数，在本例中我们将其分别称为 resolve 和 reject。这些是 promise 的决议函数。resolve(..) 通常标识完成，而 reject(..) 则标识拒绝。

你可能会猜测 bar(..) 和 baz(..) 的内部实现或许如下：

```
function bar(fooPromise) {
    // 侦听foo(..)完成
    fooPromise.then(
        function(){
            // foo(..)已经完毕,所以执行bar(..)的任务
        },
        function(){
            // 啊,foo(..)中出错了!
        }
    );
}

// 对于baz(..)也是一样
```

Promise 决议并不一定要像前面将 Promise 作为未来值查看时一样会涉及发送消息。它也可

以只作为一种流程控制信号，就像前面这段代码中的用法一样。

另外一种实现方式是：

```
function bar() {
    // foo(..)肯定已经完成,所以执行bar(..)的任务
}

function oopsBar() {
    // 啊,foo(..)中出错了,所以bar(..)没有运行
}

// 对于baz()和oopsBaz()也是一样

var p = foo( 42 );

p.then( bar, oopsBar );

p.then( baz, oopsBaz );
```

如果以前有过基于 Promise 的编码经验的话，那你可能就会不禁认为前面代码的最后两行可以用链接的方式写作 `p.then( .. ).then( .. )`，而不是 `p.then(..); then(..)`。但是，请注意，那样写的话意义就完全不同了！目前二者的区别可能还不是很清晰，但与目前为止我们看到的相比，这确实是一种不同的异步模式——分割与复制。别担心，对于这一点，本章后面还会深入介绍。

这里没有把 promise `p` 传给 `bar(..)` 和 `baz(..)`，而是使用 promise 控制 `bar(..)` 和 `baz(..)` 何时执行，如果执行的话。最主要的区别在于错误处理部分。

在第一段代码的方法里，不论 `foo(..)` 成功与否，`bar(..)` 都会被调用。并且如果收到了 `foo(..)` 失败的通知，它会亲自处理自己的回退逻辑。显然，`baz(..)` 也是如此。

在第二段代码中，`bar(..)` 只有在 `foo(..)` 成功时才会被调用，否则就会调用 `oppsBar(..)`。`baz(..)` 也是如此。

这两种方法本身并谈不上对错，只是各自适用于不同的情况。

不管哪种情况，都是从 `foo(..)` 返回的 promise `p` 来控制接下来的步骤。

另外，两段代码都以使用 promise `p` 调用 `then(..)` 两次结束。这个事实说明了前面的观点，就是 Promise（一旦决议）一直保持其决议结果（完成或拒绝）不变，可以按照需要多次查看。

一旦 `p` 决议，不论是*现在*还是*将来*，下一个步骤总是相同的。

3.2 具有 then 方法的鸭子类型

在 Promise 领域，一个重要的细节是如何确定某个值是不是真正的 Promise。或者更直接地说，它是不是一个行为方式类似于 Promise 的值？

既然 Promise 是通过 `new Promise(..)` 语法创建的，那你可能就认为可以通过 `p instanceof Promise` 来检查。但遗憾的是，这并不足以作为检查方法，原因有许多。

其中最主要的是，Promise 值可能是从其他浏览器窗口（iframe 等）接收到的。这个浏览器窗口自己的 Promise 可能和当前窗口 /frame 的不同，因此这样的检查无法识别 Promise 实例。

还有，库或框架可能会选择实现自己的 Promise，而不是使用原生 ES6 `Promise` 实现。实际上，很有可能你是在早期根本没有 Promise 实现的浏览器中使用由库提供的 Promise。

在本章后面讨论 Promise 决议过程的时候，你就会了解为什么有能力识别和判断类似于 Promise 的值是否是真正的 Promise 仍然很重要。不过，你现在只要先记住我的话，知道这一点很重要就行了。

因此，识别 Promise（或者行为类似于 Promise 的东西）就是定义某种称为 thenable 的东西，将其定义为任何具有 `then(..)` 方法的对象和函数。我们认为，任何这样的值就是 Promise 一致的 thenable。

根据一个值的形态（具有哪些属性）对这个值的类型做出一些假定。这种*类型检查*（type check）一般用术语*鸭子类型*（duck typing）来表示——“如果它看起来像只鸭子，叫起来像只鸭子，那它一定就是只鸭子”（参见本书的“类型和语法”部分）。于是，对 thenable 值的鸭子类型检测就大致类似于：

```
if (
    p !== null &&
    (
        typeof p === "object" ||
        typeof p === "function"
    ) &&
    typeof p.then === "function"
) {
    // 假定这是一个thenable!
}
else {
    // 不是thenable
}
```

除了在多个地方实现这个逻辑有点丑陋之外，其实还有一些更深层次的麻烦。

如果你试图使用恰好有 `then(..)` 函数的一个对象或函数值完成一个 Promise，但并不希望

它被当作 Promise 或 thenable，那就有点麻烦了，因为它会自动被识别为 thenable，并被按照特定的规则处理（参见本章后面的内容）。

即使你并没有意识到这个值有 `then(..)` 函数也是这样。比如：

```
var o = { then: function(){} };

// 让v [[Prototype]]-link到o
var v = Object.create( o );

v.someStuff = "cool";
v.otherStuff = "not so cool";

v.hasOwnProperty( "then" );     // false
```

`v` 看起来根本不像 Promise 或 thenable。它只是一个具有一些属性的简单对象。你可能只是想要像对其他对象一样发送这个值。

但你不知道的是，`v` 还 `[[Prototype]]` 连接（参见《你不知道的 JavaScript（上卷）》的“this 和对象原型”部分）到了另外一个对象 `o`，而后者恰好具有一个 `then(..)` 属性。所以 thenable 鸭子类型检测会把 `v` 认作一个 thenable。

甚至不需要是直接有意支持的：

```
Object.prototype.then = function(){};
Array.prototype.then = function(){};

var v1 = { hello: "world" };
var v2 = [ "Hello", "World" ];
```

`v1` 和 `v2` 都会被认作 thenable。如果有任何其他代码无意或恶意地给 `Object.prototype`、`Array.prototype` 或任何其他原生原型添加 `then(..)`，你无法控制也无法预测。并且，如果指定的是不调用其参数作为回调的函数，那么如果有 Promise 决议到这样的值，就会永远挂住！真是疯狂。

难以置信？可能吧。

但是别忘了，在 ES6 之前，社区已经有一些著名的非 Promise 库恰好有名为 `then(..)` 的方法。这些库中有一部分选择了重命名自己的方法以避免冲突（这真糟糕！）。而其他的那些库只是因为无法通过改变摆脱这种冲突，就很不幸地被降级进入了“与基于 Promise 的编码不兼容”的状态。

标准决定劫持之前未保留的——听起来是完全通用的——属性名 `then`。这意味着所有值（或其委托），不管是过去的、现存的还是未来的，都不能拥有 `then(..)` 函数，不管是有意的还是无意的；否则这个值在 Promise 系统中就会被误认为是一个 thenable，这可能会导致非常难以追踪的 bug。