Promsie 到底解决了哪些问题？

Promise 很多同学都听说过，但是同学们真的了解 Promise 么？为什么要用它，为什么要有它？以及它的出现为我们解决了怎么样的问题，这些都是我们需要知道的，接下来我们一步步进行分析，由浅入深。

为什么要使用 Promise？

在我们 javaScript 大环境下，我们的编程方式更多是基于异步编程，究竟什么是异步编程，为什么要异步编程，我们之后的文章会说。在异步编程中使用的最多的就是回调函数，先了解一下什么是回调函数。
回调函数指的是：**被调用者回头调用调用者的函数，这种由调用方自己提供的函数叫回调函数。**
应用场景举例：

对于数组的 filter 方法，里面实现的过滤的逻辑，实现了如何过滤，但是过滤的条件是什么，filter 方法中提供回调函数，让我们自己写。为提高程序的通用性，掉用者提供一个过滤条件函数，这样 filter 函数借此调用调用者的函数来进行过滤。

应用实例举例：

```
[1,2,3,4,5].filter(item => item % 2 ==0)
```

返回数组当中偶数构成的数组。
在前端当中涉及使用回调函数的地方非常的多。最常使用的地方在于我们发送 Ajax 请求。一个请求发出去我们在等待结果，就会有相应的成功处理函数，以及失败处理函数。这里处理函数指的就是我们的回调函数。同步回调没有什么问题，真的回调问题在于异步，记下来我们一起来看。

让我们异步回调的例子，但让我稍微修改它一下来画出重点：

```
// A
ajax( "..", function(..){
    // C
} );
// B
```

// A 和// B 代表程序的前半部分（也就是 现在），// C 标识了程序的后半部分（也就是 稍后）。前半部分立即执行，然后会出现一个不知多久的

“暂停”。在未来某个时刻，如果 Ajax 调用完成了，那么程序会回到它刚才离开的地方，并 继续 执行后半部分。

换句话说，回调函数包装或封装了程序的 延续。

让我们把代码弄得更简单一些：

```
// A
setTimeout( function(){
    // C
}, 1000 );
// B
```

稍停片刻然后问你自己，你将如何描述（给一个不那么懂 JS 工作方式的人）这个程序的行为。

现在大多数同学可能在想或说着这样的话:“做 A,然后设置一个等待 1000 毫秒的定时器，一旦它触发，就做 C”。与你的版本有多接近？

你可能已经发觉了不对劲儿的地方，给了自己一个修正版：“做 A，设置一个 1000 毫秒的定时器，然后做 B，然后在超时事件触发后，做 C”。这比第一个版本更准确。你能发现不同之处吗？

虽然第二个版本更准确，但是对于以一种将我们的大脑匹配代码，代码匹配 JS 引擎的方式讲解这段代码来说，这两个版本都是不足的。这里的鸿沟既是微小的也是巨大的，而且是理解回调作为异步表达和管理的缺点的关键。

只要我们以回调函数的方式引入一个延迟时间（或者像许多程序员那样引入几十个！），我们就允许了一个分歧在我们的大脑如何工作和代码将运行的方式之间形成。**当这两者背离时，我们的代码就不可避免地陷入这样的境地：更难理解，更难推理，更难调试，和更难维护。**主要的原因在于我们的大脑。我们大脑的逻辑，也就是人正常的思维逻辑与这种回调的方式不符。人更擅长做完一件事，在做另一件事。

接着来在一个实际场景下看看如何编程，以及在书写代码过程中会出现怎么样的问题。

问题： 实现网上购票流程。

解决上面的基本流程是：

1. 查询车票
2. 查询到车票，进行购买
3. 购买查询车票，进行占票
4. 占票成功后进行付款

5. 付款成功后打印车票

网上购票分为以上五个步骤，但是这里面每一步都是需要异步执行的。

```
$.ajax({   // 发送查询车票请求
        type: 'GET',
        url: '/api/serachTickey?begin="beijing"&end="Harbin"',
        success: function (req) { // 查询成功
            renderList(req.data.list) // 渲染列表
            //用户选择车次后继续买票
            $.ajax({
                type: 'GET',
                url:
'/api/buyTickey?begin=beijing&end=Harbin&trainNuber=8888',
                success: function (req) { // 抢票成功，接下来进行付款
                    $.ajax({
                        type: 'GET',
                        url: `/api/payMent?moeny=${req.money}`,
                        success: function () {
                            console.log('付款失败成功')
                        },
                        error: function () {
                            console.log('付款失败')
                        }
                    })
                },
                error: function (err) {   // 买票失败
                    console.log(err)
                }
            })
        },
        error: function (err) { // 查询失败
            console.log(err)
        }
    })
```

这样的代码常被称为“回调地狱（callback hell）”，有时也被称为“末

日金字塔（pyramid of doom）”（由于嵌套的缩进使它看起来像一个放倒的三角形）。

但是“回调地狱”实际上与嵌套/缩进几乎无关。我们可以把上面的代码改写成

```
$ajax({
    type: 'GET',
    url: '/api/serachTickey?begin="beijing"&end="Harbin"',
    success: buyTicket,
    error: error
})

function buyTicket(req) {
    renderList(req.data.list) //  渲染列表
    //用户选择车次后继续买票
    $.ajax({
        type: 'GET',
        url:
'/api/buyTickey?begin=beijing&end=Harbin&trainNuber=8888',
        success: payMent,
        error: error
    })
}
function payMent(req) {
        // 抢票成功，接下来进行付款
        $.ajax({
            type: 'GET',
            url: `/api/payMent?moeny=${req.money}`,
            success: printMsg,
            error: error
        })
}
function printMsg(msg) {
    console.log(msg)
}

function error(err) {
```

```
        console.log(err)
    }
```

样的代码组织形式几乎看不出来有前一种形式的嵌套/缩进困境，但它的每一处依然容易受到“回调地狱”的影响。为什么呢？

当我们线性地（顺序地）推理这段代码，我们不得不从一个函数跳到下一个函数，再跳到下一个函数，并在代码中弹来弹去以“看到”顺序流。并且要记住，这个简化的代码风格是某种最佳情况。我们都知道真实的 JS 程序代码经常更加神奇地错综复杂，使这样量级的顺序推理更加困难。

另一件需要注意的事是：为了将第 2，3，4 步链接在一起使他们相继发生，回调独自给我们的启示是将第 2 步硬编码在第 1 步中，将第 3 步硬编码在第 2 步中，将第 4 步硬编码在第 3 步中，如此继续。硬编码不一定是一件坏事，如果第 2 步应当总是在第 3 步之前真的是一个固定条件。

不过硬编码绝对会使代码变得更脆弱，因为它不考虑任何可能使在步骤前行的过程中出现偏差的异常情况。举个例子，如果第 2 步失败了，第 3 步永远不会到达，第 2 步也不会重试，或者移动到一个错误处理流程上，等等。

所有这些问题你都可以手动硬编码在每一步中，但那样的代码总是重复性的，而且不能在其他步骤或你程序的其他异步流程中复用。

即便我们的大脑可能以顺序的方式规划一系列任务（这个，然后这个，然后这个），但我们大脑运行的事件的性质，使恢复/重试/分流这样的流程控制几乎毫不费力。如果你出去购物，而且你发现你把购物单忘在家里了，这并不会因为你没有提前计划这种情况而结束这一天。你的大脑会很容易地绕过这个小问题：你回家，取购物单，然后回头去商店。

但是手动硬编码的回调（甚至带有硬编码的错误处理）的脆弱本性通常不那么优雅。一旦你最终指明了（也就是提前规划好了）所有各种可能性/路径，代码就会变得如此复杂以至于几乎不能维护或更新。

这 才是“回调地狱”想表达的！嵌套/缩进基本上一个余兴表演，尽管看起来还是有些不方便的。

上面是多个回调配合着嵌套产生的回调地狱问题，回调还会产生信任问题。在顺序的大脑规划和 JS 代码中回调驱动的异步处理间的不匹配只是关于回调的问题的一部分。还有一些更深刻的问题值得担忧。

让我们再一次重温这个概念——回调函数是我们程序的延续（也就是程序的第二部分）：

```
// A
ajax( "..", function(..){
    // C
} );
// B
```

// A 和// B 现在 发生，在 JS 主程序的直接控制之下。但是// C 被推迟到稍后 再发生，并且在另一部分的控制之下——这里是 ajax(..)函数。在基本的感觉上，这样的控制交接一般不会让程序产生很多问题。

但是不要被这种控制切换不是什么大事的罕见情况欺骗了。事实上，它是回调驱动的设计的最可怕的（也是最微妙的）问题。这个问题围绕着一个想法展开：有时 ajax(..)（或者说你向之提交回调的部分）不是你写的函数，或者不是你可以直接控制的函数。很多时候它是一个由第三方提供的工具。当你把你程序的一部分拿出来并把它执行的控制权移交给另一个第三方时，我们称这种情况为“控制反转”。在你的代码和第三方工具之间有一个没有明言的“契约”——一组你期望被维护的东西。

你是一个开发者，正在建造一个贩卖昂贵电视的网站的结算系统。你已经将结算系统的各种页面顺利地制造完成。在最后一个页面，当用户点解“确定”购买电视时，你需要调用一个第三方函数（假如由一个跟踪分析公司提供），以便使这笔交易能够被追踪。

你注意到它们提供的是某种异步追踪工具，也许是为了最佳的性能，这意味着你需要传递一个回调函数。在你传入的这个程序的延续中，有你最后的代码——划客人的信用卡并显示一个感谢页面。

这段代码可能看起来像这样：

```
analytics.trackPurchase( purchaseData, function(){
    chargeCreditCard();
    displayThankyouPage();
} );
```

足够简单，对吧？你写好代码，测试它，一切正常，然后你把它部署到生产环境。大家都很开心！

若干个月过去了，没有任何问题。你几乎已经忘了你曾写过的代码。一天早上，工作之前你先在咖啡店坐坐，悠闲地享用着你的拿铁，直到你接到老板慌张的电话要求你立即扔掉咖啡并冲进办公室。

当你到达时，你发现一位高端客户为了买同一台电视信用卡被划了 5 次，而且可以理解，他不高兴。客服已经道了歉并开始办理退款。但你的老板要求知道这是怎么发生的。“我们没有测试过这样的情况吗！？”

你甚至不记得你写过的代码了。但你还是往回挖掘试着找出是什么出错了。

在分析过一些日志之后，你得出的结论是，唯一的解释是分析工具不知怎么的，也就是第三方的函数出了问题，由于某些原因，将你的回调函数调用了 5 次而非一次。他们的文档中没有任何东西提到此事。
十分令人沮丧，你联系了客户支持，当然他们和你一样惊讶。他们同意将此事向上提交至开发者，并许诺给你回复。第二天，你收到一封很长的邮件解释他们发现了什么，然后你将它转发给了你的老板。
看起来，分析公司的开发者曾经制作了一些实验性的代码，在一定条件下，将会每秒重试一次收到的回调，在超时之前共计 5 秒。他们从没想要把这部分推到生产环境，但不知怎地他们这样做了，而且他们感到十分难堪而且抱歉。然后是许多他们如何定位错误的细节，和他们将要如何做以保证此事不再发生。等等，等等。
你找你的老板谈了此事，但是他对事情的状态不是感觉特别舒服。他坚持，而且你也勉强地同意，你不能再相信他们了，而你将需要指出如何保护放出的代码，使它们不再受这样的漏洞威胁。

修修补补之后，你实现了一些如下的特殊逻辑代码，团队中的每个人看起来都挺喜欢：

```
var tracked = false;
analytics.trackPurchase( purchaseData, function(){
    if (!tracked) {
        tracked = true;
        chargeCreditCard();
        displayThankyouPage();
    }
} );
```

在这里我们实质上创建了一个门阀来处理我们的回调被并发调用多次的情况。
但一个 QA 的工程师问，“如果他们没调你的回调怎么办？” 噢。谁也没想过。
你开始布下天罗地网，考虑在他们调用你的回调时所有出错的可能性。这里是你得到的分析工具可能不正常运行的方式的大致列表：

调用回调过早（在它开始追踪之前）
调用回调过晚 (或不调)
调用回调太少或太多次（就像你遇到的问题！）
没能向你的回调传递必要的环境/参数
吞掉了可能发生的错误/异常

...
这感觉像是一个麻烦清单，因为它就是。你可能慢慢开始理解，你将要不得不为 每一个传递到你不能信任的工具中的回调 都创造一大堆的特殊逻辑。

在上面的过程中我们发现几个大问题

1. **回调配合着嵌套会产生回调地狱问题，思路很不清晰。**
2. **由于回调存在着依赖反转，在使用第三方提供的方法时，存在信任问题。**
3. **当我们不写错误的回调函数时，会存在异常无法捕获**
4. **导致我们的性能更差，本来可以一起做的但是使用回调，导致多件事情顺序执行。用的时间更多**

针对这样的问题我们该怎么解决呢？

我们首先想要解决的是信任问题，信任是如此脆弱而且是如此的容易丢失。

回想一下，我们将我们的程序的延续包装进一个回调函数中，将这个回调交给另一个团体（甚至是潜在的外部代码），并双手合十祈祷它会做正确的事情并调用这个回调。

我们这么做是因为我们想说，“这是 稍后 将要发生的事，在当前的步骤完成之后。”

但是如果我们能够反向倒转这种控制反转呢？如果不是将我们程序的延续交给另一个团体，而是希望它返回给我们一个可以知道它何时完成的能力，然后我们的代码可以决定下一步做什么呢？

这种规范被称为 Promise。

Promise 正在像风暴一样席卷 JS 世界，因为开发者和语言规范作者之流拼命地想要在他们的代码/设计中结束回调地狱的疯狂。事实上，大多数新被加入 JS/DOM 平台的异步 API 都是建立在 Promise 之上的。

什么是 Promise？

Promise 是异步编程的一种解决方案：从语法上讲，promise 是一个对象，从它可以获取异步操作的消息；从本意上讲，它是承诺，承诺它过一段时间会给你一个结果。promise 有三种状态：pending(等待态)，fulfiled(成功

态)，rejected(失败态)；状态一旦改变，就不会再变。创造 promise 实例后，它会立即执行。一般来说我们会碰到的回调嵌套都不会很多，一般就一到两级，但是某些情况下，回调嵌套很多时，代码就会非常繁琐，会给我们的编程带来很多的麻烦，这种情况俗称——回调地狱。
这时候我们的 promise 就应运而生、粉墨登场了。

Promise 的基本使用

Promise 是一个构造函数，自己身上有 all、reject、resolve 这几个眼熟的方法，原型上有 then、catch 等同样很眼熟的方法。

那就 new 一个

```
let p = new Promise((resolve, reject) => {
    //做一些异步操作
    setTimeout(() => {
        console.log('执行完成');
        resolve('我是成功！！');
    }, 2000);
});
```

Promise 的构造函数接收一个参数：函数，并且这个函数需要传入两个参数：
resolve ：异步操作执行成功后的回调函数
reject：异步操作执行失败后的回调函数
then 链式操作的用法
所以，从表面上看，Promise 只是能够简化层层回调的写法，而实质上，Promise 的精髓是“状态”，用维护状态、传递状态的方式来使得回调函数能够及时调用，它比传递 callback 函数要简单、灵活的多。所以使用 Promise 的正确场景是这样的，把我们之前的问题修改一下：

```
function myAjax(url,type="GET") {
        return new Promise((resolve, reject) => {
            $.ajax({
                type,
                url,
                success: resolve,
```

```
                error: reject
            })
        })
    }

    myAjax('/api/serachTickey?begin="beijing"&end="Harbin"')
    .then(req => {
        renderList(req.data.list)
        return
myAjax('/api/buyTickey?begin=beijing&end=Harbin&trainNuber=8888')
    },err => {    console.log(err)})
    .then(req => {
        return myAjax(`/api/payMent?moeny=${req.money}`)
    },err => console.log(err))
    .then(req => {
        console.log(req.msg)
    }, err => console.log(err))
```

通过 Promise 这种方式很好的解决了回调地狱问题，使得异步过程同步化，让代码的整体逻辑与大脑的思维逻辑一致，减少出错率。

reject 的用法 ：

把 Promise 的状态置为 rejected，这样我们在 then 中就能捕捉到，然后执行“失败”情况的回调。看下面的代码。

```
    let p = new Promise((resolve, reject) => {
        //做一些异步操作
      setTimeout(function(){
            var num = Math.ceil(Math.random()*10); //生成 1-10 的随机数
            if(num<=5){
                resolve(num);
            }
            else{
                reject('数字太大了');
            }
      }, 2000);
    });
    p.then((data) => {
            console.log('resolved',data);
```

```
        },(err) => {
            console.log('rejected',err);
        }
    );
```

then 中传了两个参数，then 方法可以接受两个参数，第一个对应 resolve 的回调，第二个对应 reject 的回调。所以我们能够分别拿到他们传过来的数据。多次运行这段代码，你会随机得到两种结果

catch 的用法

我们知道 Promise 对象除了 then 方法，还有一个 catch 方法，它是做什么用的呢？其实它和 then 的第二个参数一样，用来指定 reject 的回调。用法是这样：

```
p.then((data) => {
    console.log('resolved',data);
}).catch((err) => {
    console.log('rejected',err);
});
```

效果和写在 then 的第二个参数里面一样。不过它还有另外一个作用：在执行 resolve 的回调（也就是上面 then 中的第一个参数）时，如果抛出异常了（代码出错了），那么并不会报错卡死 js，而是会进到这个 catch 方法中。请看下面的代码：

```
p.then((data) => {
    console.log('resolved',data);
    console.log(somedata); //此处的 somedata 未定义
})
.catch((err) => {
    console.log('rejected',err);
});
```

在 resolve 的回调中，我们 console.log(somedata);而 somedata 这个变量是没有被定义的。如果我们不用 Promise，代码运行到这里就直接在控制台报错了，不往下运行了

也就是说进到 catch 方法里面去了，而且把错误原因传到了 reason 参数中。即便是有错误的代码也不会报错了，这与我们的 try/catch 语句有相同的功能

all 的用法：谁跑的慢，以谁为准执行回调。all 接收一个数组参数，里面的值最终都算返回 Promise 对象

Promise 的 all 方法提供了并行执行异步操作的能力，并且在所有异步操作

执行完后才执行回调。看下面的例子：

```
let Promise1 = new Promise(function(resolve, reject){})
let Promise2 = new Promise(function(resolve, reject){})
let Promise3 = new Promise(function(resolve, reject){})

let p = Promise.all([Promise1, Promise2, Promise3])

p.then(funciton(){
  // 三个都成功则成功
}, function(){
  // 只要有失败，则失败
})
```

有了 all，你就可以并行执行多个异步操作，并且在一个回调中处理所有的返回数据，是不是很酷？有一个场景是很适合用这个的，一些游戏类的素材比较多的应用，打开网页时，预先加载需要用到的各种资源如图片、flash 以及各种静态文件。所有的都加载完后，我们再进行页面的初始化。在这里可以解决时间性能的问题，我们不需要在把每个异步过程同步出来。

race 的用法：谁跑的快，以谁为准执行回调

race 的使用场景：比如我们可以用 race 给某个异步请求设置超时时间，并且在超时后执行相应的操作，代码如下：

```
//请求某个图片资源
    function requestImg(){
        var p = new Promise((resolve, reject) => {
            var img = new Image();
            img.onload = function(){
                resolve(img);
            }
            img.src = '图片的路径';
        });
        return p;
    }
    //延时函数，用于给请求计时
    function timeout(){
        var p = new Promise((resolve, reject) => {
            setTimeout(() => {
                reject('图片请求超时');
```

```
            }, 5000);
        });
        return p;
    }
    Promise.race([requestImg(), timeout()]).then((data) =>{
        console.log(data);
    }).catch((err) => {
        console.log(err);
    });。
```

接下来再说一说 Promise 解决回调信任问题

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。

调用过早

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。

调用过晚

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。

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。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。

```
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

有很多调度的细微差别。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()应该最终会制调用 p1 的回调，然后是 p2。但还有一些微妙的场景可能不是这样。

```
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，而不是像你认为的 B A
```

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。

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。

回调未调用

首先，没有任何东西（甚至 JS 错误）能阻止 Prmise 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一个。

当然，如果你的回调函数本身包含 JS 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。后面讨论，这些错误并不会被吞掉。

但是，如果 Promise 永远不决议呢？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(),
    timeoutPromise( 3000 );
])
.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
    }
)
```

我们可保证一个 foo()有一个信号，防止其永久挂住程序。

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”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)），那头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。

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 JS 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 resovel()或者 reject()第一个参数之后的所有参数都会被默默忽略。

如果要传递多个值，你就必须要把它们封装在一个数组或对象中。

对环境来说，JS 中的函数总是保持其定义所在的作用域的闭包，所以它们当然可继续你提供的环境状态。

吞掉错误或异常

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 JS 异常错误，比如一个 TypeError 或 RefernceError，那这个异常就会被捕捉，并且会使这个 Promise 被拒绝。

```
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 JS 异常导致了 Promise 拒绝，你可捕捉并对其做出响应。Promise 甚至把 JS 异常也变成了异步行为，进而极大降低了竟态条件出现

的可能。

但是，如果 Promise 完成后在查看结果时（then()注册回调中）出现了 JS 异常错误会怎样呢？

```
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 qvnn 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。但是这里有一个深的问题。就是我们没有侦听到它。p.then()调用本身返回了另一个 promise，正是这个 promise 将会因 TypeError 异常而被拒绝。

是可信任的 Promise 吗

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，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？

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
```

如果向 Promise.resolve()传递了一个非 Promise 的 thenable 值，前者会试图展开这个值，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。

```
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){
        // 永远不会到这里
    }
)
```

但是，下面又会怎样呢？

```
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
    }
)
```

尽管如此，我们还是都可把这些版本的 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){
        // 永远不会到这里
    }
)
```

Promise.resolve()可接受任何 thenable，将其解封完它的非 thenable 值。从 Promise.resolve()得到的是一个真正的 Promise,是一个可信任的值。如果你传入的已经是真正的 Promise，那们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，但我们可知道它至少是一个 thenable。Promise.resolve()提供了可信任的 Promise 封装工具，可链接使用：

```
// 不要这么做
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))就能保证总返回一个 Promise 结果。