JavaScript 开发者们，现在是时候承认一个事实了：我们在 promises 的使用上还存在问题。但并不是 promises 本身有问题，被 A+标准 定义的 promises 是极好的。

在过去一年的课程中揭示给我的一个比较大的问题是，正如我所看到的，很多程序员在使用 PouchDB API 以及与其他重 promise 的 API 的过程中存在的一个问题是：

**我们一部分人在使用 promises 的过程中并没有真正的理解 promises。**

如果你觉得这不可思议，那么考虑下我最近在 Twitter 上的写的一个比较难的题目：

**问题：下面 4 个 promises 有什么区别呢？**

```
doSomething().then(function () {
  return doSomethingElse();
});

doSomething().then(function () {
  doSomethingElse();
});

doSomething().then(doSomethingElse());

doSomething().then(doSomethingElse);
```

答案在这篇文章的末尾，但是首先，我想在第一时间搞清楚为什么 promises 是如此的棘手，以及为什么我们这么多人，不管是新手还是像专家的人，都会被它们搞晕。我也将提供我认为的我对其独特的洞察力，用一个独特的技巧，使得对 promises 的理解成为有把握的事情。是的，在这之后我认为它们真的不是那么难。

但是在开始之前，让我们挑战一下对于 promises 常见的一些假设。

## 为什么要用 promises？

如果你读过关于 promises 的一些文章，你经常会发现对《世界末日的金字塔》这篇文章的引用，会有一些可怕的逐渐延伸到屏幕右侧的回调代码。

promises 确实解决了这个问题，但它并不只是关乎于缩进。正如在优秀的演讲《回调地狱的救赎》中所解释的那样，回调真正的问题在于它们剥夺了我们对一些像 return 和 throw 的关键词的使用能力。相反，我们的程序的整个流程都会基于一些副作用。一个函数单纯的调用另一个函数。

事实上，回调做了很多更加险恶的事情：它们剥夺了我们的堆栈，这些是我们在编程语言中经常要考虑的。没有堆栈来书写代码在某种程度上就好比驾车没有刹车踏板：你不会知道你是多么需要它，直到你到达了却发现它并不在这。

promises 的全部意义在于它给回了在函数式语言里面我们遇到异步时所丢失的 return，throw 和堆栈。为了更好的从中获益你必须知道如何正确的使用 promises。

## 新手常犯的错误

一些人尝试解释 promises 是卡通，或者是一种名词导向的方式：“你可以传递的东西就是代表着异步值”。

我并没有发现这些这些解释多么有帮助。对于我来说，promises 就是关乎于代码结构和流程。因此我认为，过一下一些常见的错误以及展示出如何修复它们是更好的方式。我把这称为“新手常犯的错误”的意义就在于，“你现在是个新手，初出茅庐，但你很快会成为专业人士”。

说一点题外话：“promises”对于不同的人有不同的理解，但是这篇文章的目的在于，我只是谈论官方标准，正如在现代浏览器中所暴露的 window.Promise API。尽管不是所有的浏览器都支持 window.Promise，对于一个很好的补充，可以查看名为 Lie 的项目，它是一个实现 promises 的遵循规范的最小集。

## 新手常见错误＃1:世界末日的 promise 金字塔

看下开发人员怎么使用拥有大量基于 promise 的 API 的 PouchDB，我发现了很多不好的 promise 模式。最常见的不好实践是这个：

```
remotedb.allDocs({
  include_docs: true,
  attachments: true
}).then(function (result) {
  var docs = result.rows;
  docs.forEach(function(element) {
    localdb.put(element.doc).then(function(response) {
      alert("Pulled doc with id " + element.doc._id + " and added to local db.");
    }).catch(function (err) {
      if (err.status == 409) {
        localdb.get(element.doc._id).then(function (resp) {
          localdb.remove(resp._id, resp._rev).then(function (resp) {
// et cetera...
```

是的，事实证明你是能使用像回调的 promises 的，而且是的，那有点像使用一个很有威力的磨砂机来打磨你的指甲，但是你是可以做到的。

如果你认为这一类型的错误会仅限于绝对的新手，你会惊讶的发现上面的代码就来自于黑莓开发者的官方博客。老的回调习惯很难改（致开发者：很抱歉拿你来举例，但是你的代码很有教育意义）。

一个更好的方式是：

```
remotedb.allDocs(...).then(function (resultOfAllDocs) {
  return localdb.put(...);
}).then(function (resultOfPut) {
  return localdb.get(...);
}).then(function (resultOfGet) {
```

```
  return localdb.put(...);
}).catch(function (err) {
  console.log(err);
});
```

这被称为组成式 promises（composing promises），它是有超能力的 promises 之一。每一个函数都在前面的 promise 被 resolve 之后被调用，而且将前面的 promise 的输出作为参数被调用。稍后将详细介绍。

## 新手常见错误 # 2:尼玛，我该怎么对 Promises 调用 forEach()呢？

这是大多数人开始理解 Promises 要突破的地方。尽管他们能熟悉 forEach()循环（或者 for 循环，或者 while 循环），他们并不知道如何对 Promises 使用这些循环。此时，他们写的代码会像是这样：

```
// I want to remove() all docs
db.allDocs({include_docs: true}).then(function (result) {
  result.rows.forEach(function (row) {
    db.remove(row.doc);
  });
}).then(function () {
  // I naively believe all docs have been removed() now!
});
```

这些代码有什么问题呢？问题在于第一个函数返回 undefined，意味着第二个函数并不是在等待 db.remove()在所有文件上被调用。实际上，它没有在等任何东西，并且在任何数量的文件被删除的时候都可能会执行。

这是一个极其阴险的 bug，因为你可能没有注意到任何有错误的地方，认为 PouchDB 会在你的 UI 更新前会删除掉所有的文件。这个 bug 可能只出现在奇怪的竞态条件，或者特定的浏览器中，此时要去做 debug 是不可能的。

这所有的症结其实在于 forEach()/for/while 并不是你要寻找的构想。你需要的是 Promise.all():

```
db.allDocs({include_docs: true}).then(function (result) {
  return Promise.all(result.rows.map(function (row) {
    return db.remove(row.doc);
  }));
}).then(function (arrayOfResults) {
  // All docs have really been removed() now!
});
```

发生了什么？通常 Promise.all()以 promises 的数组作为参数，然后返回另一个 promise，它只有在其他所有的 promise 都 resolve 之后执行 resolve。它是 for 循环的一个异步等价物。

Promises.all()将一个数组作为结果传给下一个函数，这是很有用的，比如你正在试图从 PouchDB 获取一些东西的时候。如果任意一个 all()的子 promise 被执行了 reject，all()也会被执行 reject，这甚至是更有用的。

**新手常见错误＃3:忘记添加.catch()**

这是另一个常见的错误。许多开发者会很自豪的认为他们的 promises 代码永远都不会出错，于是他们忘记在代码中添加.catch()方法。不幸的是，这会导致任何被抛出的错误都会被吞噬掉，甚至在你的控制台你也不会发现有错误输出。这在 debug 代码的时候真的会非常痛苦。

为了避免这种讨厌的场景，我已经习惯了在我的 promise 链中简单的添加如下代码：

```
somePromise().then(function () {
  return anotherPromise();
}).then(function () {
  return yetAnotherPromise();
}).catch(console.log.bind(console)); // <-- this is badass
```

甚至是你从未预料到会出错，添加.catch()方法都是很精明的做法。如果你的假设曾经被证明是错误的，它会让你的生活变的更简单。

## 新手常见错误＃4:使用 deferred

这是一个我总是会看到的错误，我甚至都不愿意在这里重复，为了以防万一，像阴间大法师那样，仅仅是提到它的名字就能得到更多的实例。

长话短说，promise 有个很长的传奇的历史，JavaScript 社区花了很长的时间来使得它的实现是正确的。在早期，jQuery 和 Angular 到处都在使用这个

“deferred”模式，现在已被更换为 ES6 Promise 标准，正如一些很好的库如 Q, When, RSVP, Bluebird, Lie 以及其他库所实现的那样。

如果你正在你的代码里写 deferred 这种模式（我不会再重复第三次），那么你做的都是错的。下面是如何来避免这种错误。

首先，大多数的 promise 库提供了一种方式从第三方库中导入 promises。例如，Angular 的$q 模块允许你使用$q.when()来封装非$q 的模块。因此 Angular 的用户可以以这种方式来封装 PouchDB 的 promises：

```
$q.when(db.put(doc)).then(/* ... */); // <-- this is all the code you need
```

另一个策略是使用揭示构造函数，这种策略对于封装非 promise 的 API 非常有用。例如，封装基于回调的 API 比如 Node 的 fs.readFile()，你可以简单的这样做：

```
new Promise(function (resolve, reject) {
  fs.readFile('myfile.txt', function (err, file) {
    if (err) {
      return reject(err);
    }
    resolve(file);
  });
}).then(/* ... */)
```

完成！我们已经击败了可怕的 def...我住嘴：）

为什么这是一种反模式更多的信息可以查看：the Bluebird wiki page on promise anti-patterns。

## 新手常见错误＃5:使用其副作用而不是 return

下面的代码有什么问题？

```
somePromise().then(function () {
  someOtherPromise();
}).then(function () {
  // Gee, I hope someOtherPromise() has resolved!
  // Spoiler alert: it hasn't.
});
```

这是一个很好的点来谈论你所需要知道的所有关于 promise 的东西。

认真一点，这是一个有点奇怪的技巧，一旦你理解了它，就会避免我所谈论的所有的错误。你准备好了吗？

正如我之前所说，promises 的神奇之处在于它给回了我们之前的 return 和 throw。但是在实际的实践中它看起来会是什么样子呢？

每一个 promise 都会给你一个 then()方法（或者 catch，它们只是 then(null,...)的语法糖）。这里我们是在 then()方法的内部来看：

```
somePromise().then(function () {
```

```
  // I'm inside a then() function!
});
```

我们在这里能做什么呢？有三种事可以做：

1、返回另一个 promise；

2、返回一个同步值（或者 undefined）；

3、抛出一个同步错误。

就是这样。一旦你理解了这个技巧，你就明白了什么是 promises。让我们一条条来说。

## 1、返回另一个 promise

在 promise 的文档中这是一种常见的模式，正如上面的“组成式 promise”例子中所看到的：

```
getUserByName('nolan').then(function (user) {
  return getUserAccountById(user.id);
}).then(function (userAccount) {
  // I got a user account!
});
```

注意，我正在返回第二个 promise - return 是很关键的。如果我没有说返回，getUserAccountById()方法将会产生一个副作用，下一个函数将会接收 undefined 而不是 userAccount。

## 2、返回一个同步值（或 undefined）

返回 undefined 通常是一个错误，但是返回一个同步值则是将同步代码转化为 promise 代码的绝好方式。比如说有一个在内存里的用户的数据。我们可以这样做：

```
getUserByName('nolan').then(function (user) {
  if (inMemoryCache[user.id]) {
    return inMemoryCache[user.id];    // returning a synchronous value!
  }
  return getUserAccountById(user.id); // returning a promise!
}).then(function (userAccount) {
  // I got a user account!
});
```

难道这不棒吗？第二个函数并不关心 userAccount 是同步还是异步获取的，第一个函数对于返回同步还是异步数据是自由的。

不幸的是，这存在一个很不方便的事实，在 JavaScript 技术里没有返回的函数默认会自动返回 undefined，这也就意味着当你想返回一些东西的时候很容易不小心引入一些副作用。

为此，我把在 then()函数里总是返回数据或者抛出异常作为我的个人编码习惯。我也推荐你这么做。

### 3、抛出一个同步错误

说到 throw，promises 可以做到更棒。比如为了避免用户被登出我们想抛出一个同步错误。这很简单：

```
getUserByName('nolan').then(function (user) {
  if (user.isLoggedOut()) {
    throw new Error('user logged out!'); // throwing a synchronous error!
  }
  if (inMemoryCache[user.id]) {
    return inMemoryCache[user.id];       // returning a synchronous value!
  }
  return getUserAccountById(user.id);    // returning a promise!
}).then(function (userAccount) {
  // I got a user account!
}).catch(function (err) {
  // Boo, I got an error!
});
```

如果我们的用户被登出了我们的 catch()方法将接收到一个同步错误，而且任意的 promises 被拒绝它都将接收到一个同步错误。再一次强调，函数并不关心错误是同步的还是异步的。

这是非常有用的，因为它能够帮助我们在开发中识别代码错误。比如，在一个 then()方法内部的任意地方，我们做了一个 JSON.parse()操作，如果 JSON 参数不合法那么它就会抛出一个同步错误。用回调的话该错误就会被吞噬掉，但是用 promises 我们可以轻松的在 catch()方法里处理掉该错误。

## 高级错误

好的，现在你已经学习了一个单一的技巧来使得 promises 变动极其简单，现在让我们来谈论一些边界情况。因为在编码过程中总存在一些边界情况。

这些错误我把它们归类为高级错误，因为我只在一些对于 promise 非常熟悉的程序员的代码中发现。但是，如果我们想解决我在文章开头提出的疑惑的话，我们需要讨论这些高级错误。

## 高级错误＃1:不了解 Promise.resolve()

正如我上面提到的，promises 在封装同步代码为异步代码上是非常有用的。然而，如果你发现自己打了这样一些代码：

```
new Promise(function (resolve, reject) {
  resolve(someSynchronousValue);
}).then(/* ... */);
```

你可以使用 Promise.resolve()来更简洁的表达：

```
Promise.resolve(someSynchronousValue).then(/* ... */);
```

而且这在捕捉任意的同步错误上会难以置信的有用。它是如此有用，以致于我习惯于几乎将我所有的基于 promise 返回的 API 方法以下面这样开始：

```
function somePromiseAPI() {
  return Promise.resolve().then(function () {
    doSomethingThatMayThrow();
    return 'foo';
  }).then(/* ... */);
}
```

记住：对于被彻底吞噬的错误以致于不能 debug 的任意代码，做同步的错误抛出都是一个很好的选择。但是你把每个地方都封装为 Promise.resolve()，你要确保后面你都会执行 caotch()。

类似的，有一个 Promise.reject()方法可以返回一个立即被拒绝的 promise：

```
Promise.reject(new Error('some awful error'));
```

## 高级错误＃2:catch()并不和 then(null,...)一摸一样

我在上面说过 catch()只是一个语法糖。下面两个代码片段是等价的：

```
somePromise().catch(function (err) {
  // handle error
});
```

```
somePromise().then(null, function (err) {
  // handle error
});
```

然而，这并不意味着下面两个片段也是等价的：

```
somePromise().then(function () {
  return someOtherPromise();
}).catch(function (err) {
  // handle error
});
```

```
somePromise().then(function () {
  return someOtherPromise();
}, function (err) {<a target=_blank href="http://mochajs.org/" target="_blank">点击打开链接</a>
  // handle error
});
```

如果你疑惑为什么它们不是等价的，思考第一个函数抛出一个错误会发生什么：

```
somePromise().then(function () {
  throw new Error('oh noes');
}).catch(function (err) {
  // I caught your error! :)
});
```

```
somePromise().then(function () {
  throw new Error('oh noes');
}, function (err) {
  // I didn't catch your error! :(
});
```

这会证明，当你使用 then(resolveHandler,rejectHandler)格式，如果 resolveHandler 自己抛出一个错误 rejectHandler 并不能捕获。

基于这个原因，我已经形成了自己的一个习惯，永远不要对 then()使用第二个参数，并总是优先使用 catch()。一个例外是当我写异步的 Mocha 测试的时候，我可能写一个测试来保证错误被抛出：

```
it('should throw an error', function () {
  return doSomethingThatThrows().then(function () {
    throw new Error('I expected an error!');
  }, function (err) {
    should.exist(err);
  });
});
```

说到这，Mocha 和 Chai 是测试 promise API 的友好的组合。pouchdb-plugin-seed 项目有很多你可以入手的简单的测试。

**高级错误＃3：promises vs promise 工厂**

我们假定你想要一个接一个的，在一个序列中执行一系列的 promise。就是说，你想要 Promise.all()这样的东西，不会并行的执行 promises。

你可能会单纯的这样写一些东西：

```
function executeSequentially(promises) {
  var result = Promise.resolve();
  promises.forEach(function (promise) {
```

```
  result = result.then(promise);
 });
 return result;
}
```

不幸的是，它并不会按你所期望的那样工作。你传递给 executeSequentially() 的 promises 会并行执行。

之所以会这样是因为其实你并不想操作一个 promise 的数组。每一个 promise 规范都指定，一旦一个 promise 被创建，它就开始执行。那么，其实你真正想要的是一个 promise 工厂数组：

```
function executeSequentially(promiseFactories) {
 var result = Promise.resolve();
 promiseFactories.forEach(function (promiseFactory) {
  result = result.then(promiseFactory);
 });
 return result;
}
```

我知道你在想什么：“这个 Java 程序员到底是谁，为什么他在谈论工厂？“不过一个 promise 工厂是很简单的，它只是一个返回一个 promise 的函数：

```
function myPromiseFactory() {
 return somethingThatCreatesAPromise();
}
```

这为什么能工作呢？它能工作是因为一个 promise 工厂并不会创建 promise 直到它被要求这么做。它的工作方式和 then 函数相同 - 实际上它们是同一个东西。

如果你在看上面的 executeSequentially()函数，并且假定 myPromiseFactory 在 result.then()内部被取代，那么希望你能灵光一闪。那时，你将实现 promise 启蒙（译者注：其实此时就是相当于执行：onePromise.then().then()...then()）。

**高级错误＃4:好吧，假设我想要获取两个 promises 的结果将会怎样？**

通常，一个 promise 是依赖于另一个 promise 的，但是这里我们想要两个 promise 的输出。例如：

```
getUserByName('nolan').then(function (user) {
  return getUserAccountById(user.id);
}).then(function (userAccount) {
  // dangit, I need the "user" object too!
});
```

如果想成为优秀的 JavaScript 开发者并避免世界末日的金字塔，我们可能在一个更高的的作用域中存储一个 user 对象变量：

```
var user;
getUserByName('nolan').then(function (result) {
```

```
  user = result;
  return getUserAccountById(user.id);
}).then(function (userAccount) {
  // okay, I have both the "user" and the "userAccount"
});
```

这也能达到目的，但是我个人觉得这有点拼凑的感觉。我推荐的做法：放手你的偏见，并拥抱金字塔：

```
getUserByName('nolan').then(function (user) {
  return getUserAccountById(user.id).then(function (userAccount) {
    // okay, I have both the "user" and the "userAccount"
  });
});
```

至少，临时先这么干。如果缩进成为一个问题，你可以做 JavaScript 开发者一直以来都在做的事情，提取函数为一个命名函数：

```
function onGetUserAndUserAccount(user, userAccount) {
  return doSomething(user, userAccount);
}

function onGetUser(user) {
  return getUserAccountById(user.id).then(function (userAccount) {
    return onGetUserAndUserAccount(user, userAccount);
  });
}

getUserByName('nolan')
  .then(onGetUser)
  .then(function () {
```

```
  // at this point, doSomething() is done, and we are back to indentation 0
});
```

随着你的 promise 代码变得更加复杂，你可能发现你自己在抽取越来越多的函数为命名函数。我发现这样会形成非常美观的代码，看起来会像是这样：

```
putYourRightFootIn()
  .then(putYourRightFootOut)
  .then(putYourRightFootIn)
  .then(shakeItAllAbout);
```

这就是 promises。

## 高级错误＃5:promises 丢失

最后，这个错误是我在上面引入 promise 疑惑的时候提到的。这是一个非常深奥的用例，可能永远不会在你的代码中出现，但它却让我感到疑惑。

你认为下面的代码会打印出什么？

```
Promise.resolve('foo').then(Promise.resolve('bar')).then(function (result) {
  console.log(result);
});
```

如果你认为打印出 bar，那你就大错特错了。它实际上会打印出 foo。

原因是当你给 then()传递一个非函数（比如一个 promise）值的时候，它实际上会解释为 then(null)，这会导致之前的 promise 的结果丢失。你可以自己测试：

```
Promise.resolve('foo').then(null).then(function (result) {
  console.log(result);
});
```

你想加多少的 then(null)都可以，它始终会打印出 foo。

这其实是一个循环，回到了上面我提到的 promises vs promises 工厂的问题上。简言之，你可以直接给 then()方法传递一个 promise，但是它并不会像你想的那样工作。then()默认接收一个函数，其实你更多的是想这样做：

```
Promise.resolve('foo').then(function () {
  return Promise.resolve('bar');
}).then(function (result) {
  console.log(result);
});
```

这次会如我们预期的那样返回 bar。

所以要提醒你自己：永远给 then()传递一个函数参数。

## 解决疑惑

现在我们已经学习了关于 promises 要知道的所有的东西（或者接近于此），我们应该能够解决我在这篇文章开始时提出的疑惑了。

这里是每一个疑惑的答案，以图形的格式展示因此你可以更好的来理解。

疑惑＃1:

```
doSomething().then(function () {
  return doSomethingElse();
}).then(finalHandler);
```

答案：

疑惑＃2:

```
doSomething().then(function () {
  doSomethingElse();
}).then(finalHandler);
```

答案：

疑惑＃3:

```
doSomething().then(doSomethingElse())
  .then(finalHandler);
```

答案：

doSomething

|----------------|

doSomethingElse(undefined)

|--------------------------------|

finalHandler(resultOfDoSomething)

|------------------|

疑惑＃4:

```
doSomething().then(doSomethingElse)
  .then(finalHandler);
```

答案：

如果这些答案仍然没有讲通，那么我鼓励重新阅读文章，或者去定义 doSomething()以及 doSomethingElse()然后在你的浏览器中自己尝试。

说明：对于这些例子，我假定 doSomething()和 doSomethingElse()都返回 promises，并且这些 promises 代表在 JavaScript 事件轮训（内嵌数据库，网络，setTimeout）之外的处理的一些东西,这就是为什么在某些时候是以并发的形式展现。这里是在 JSBin 的一个证明。

promises 更多的使用说明，请参考我的 promise 主要用法背忘单。

```
// Promise.all is good for executing many promises at once
Promise.all([
  promise1,
  promise2
]);

// Promise.resolve is good for wrapping synchronous code
Promise.resolve().then(function () {
```

```
  if (somethingIsNotRight()) {
    throw new Error("I will be rejected asynchronously!");
  } else {
    return "This string will be resolved asynchronously!";
  }
});

// execute some promises one after the other.
// this takes an array of promise factories, i.e.
// an array of functions that RETURN a promise
// (not an array of promises themselves; those would execute immediately)
function sequentialize(promiseFactories) {
  var chain = Promise.resolve();
  promiseFactories.forEach(function (promiseFactory) {
    chain = chain.then(promiseFactory);
  });
  return chain;
}

// Promise.race is good for setting a timeout:
Promise.race([
  new Promise(function (resolve, reject) {
    setTimeout(reject, 10000); // timeout after 10 secs
  }),
  doSomethingThatMayTakeAwhile()
]);

// Promise finally util similar to Q.finally
// e.g. promise.then(...).catch().then(...).finally(...)
function finally (promise, cb) {
  return promise.then(function (res) {
    var promise2 = cb();
    if (typeof promise2.then === 'function') {
      return promise2.then(function () {
        return res;
      });
```

```
    }
    return res;
  }, function (reason) {
    var promise2 = cb();
    if (typeof promise2.then === 'function') {
      return promise2.then(function () {
        throw reason;
      });
    }
    throw reason;
  });
};
```

## 关于 promise 最后的话

promises 非常棒。如果你仍然在使用回调，我强烈鼓励你切换到 promises。你的代码将变得更少，更优雅，更容易维护。

如果你不相信我，这里有证明：PouchDB’ 的 map/reduce 模块的一次重构来将回调替换为 promises。结果是：290 个插入，555 个删除。

顺便说一下，写令人讨厌的回调代码的其实是我。promises 的原始力量成为了我的第一课，也感谢 PouchDB 的其它贡献者一路上对我的指导。

那也就是说，prmoises 并不是完美的。它们确实比回调要更好，但是那有点像说肠子上的一个穿孔比拔掉牙齿要更好。当然了，一个比另一个更可取，但是如果你有更好的选择，你最好都规避它们。

虽然比回调要优越，但是 promises 是理解比较困难而且容易出错，我感觉有必要写这篇博客就是明证。新手和专家都会把这个东西搞的一塌糊涂，事实上，这并不是他们的错。问题是 promises 本身，和我们在同步代码中使用的模式类似，是一个不错的替代又不完全一样。

实际上，你不应该不得不去学习一堆晦涩难懂的规则和新的 API 来做这些事，在同步的世界里，你可以完美的使用像是 return，catch，throw 以及 for 循环这些熟悉的模式。在你的脑海中不应该总是保持着两套并行的体系。

**异步等待/等待**

这就是我在《驯服 ES7 的异步野兽》这篇文章中所阐明的观点，在这篇文章中我探索了 ES7 的 async/await 关键词，以及他们是如何更深入的将 promises 集成进语言中的。替代不得不去写一些伪同步代码（用一个像 catch 的 catch() 方法，但并不是真正的 catch），ES7 将允许我们使用真正的 try/catch/return 关键字，就像我们在 CS 101 中学到的。

对于 JavaScript 作为一门语言来说，这是一个巨大的福利。因为在最后，这些 promise 的反模式将仍然会此起彼伏，只要当我们在犯错误的时候我们的的工具没有告诉我们。

取 JavaScript 历史中的一个例子，说 JSLint 和 JSHint 比《JavaScript：语言精粹》对社区做了更大的贡献我认为是公平的，尽管它们包含了相同的信息。

这就是明确告诉你你代码中的错误，而不是去读一本书来试图理解其他人的错误之间的区别。

ES7 中 await/async 的优美之处在于，大多数情况下，你的错误是语法或编译器错误而不是微妙的运行时 bug。尽管是这样，到这之前，知道 promises 能做什么，以及在 ES5 和 ES6 中如何合理的使用它们总是好的。

所以在我意识到这些之后，这篇博客影响有限，就像《JavaScript：语言精粹》这本书所做的，希望当你看到有人在犯相同的错误的时候你可以指出。因为还有太多的人需要承认：“对于 promises 我还有问题”。

更新：已经有人跟我指出 Bluebird 3.0 能打印出警告来避免我在这篇文章中所鉴定的一些错误。所以在我们还在等待 ES7 的时候使用 Bluebird 是另一个很棒的选择。