JavaScript Promises

There and back again

By Jake Archibald

翻译： 晋晓炜(Amio)

已发布: 十二月 16th, 2013

已更新: 一月 29th, 2014

Comments: 164

女士们先生们，请准备好迎接 Web 开发历史上一个重大时刻……

[鼓声响起]

JavaScript 有了原生的 Promise！

[漫天的烟花绽放，人群沸腾了]

这时候你大概是这三种人之一：

- 你的身边拥挤着欢呼的人群，但是你却不在其中，甚至你还不大清楚“Promise”是什么。你耸耸肩，烟花的碎屑在你的身边落下。这样的话，不要担心，我也是花了多年的时间才明白 Promise 的意义，你可以从这里开始看起。
- 你一挥拳！太赞了对么！你已经用过一些 Promise 的库，但是所有这些第三方实现在API上都略有差异，JavaScript 官方的 API 会是什么样子？看这里！
- 你早就知道了，看着那些欢呼雀跃的新人你的嘴角泛起一丝不屑的微笑。你可以安静享受一会儿优越感，然后直接去看 API 参考吧。

他们都在激动什么？

JavaScript 是单线程的，这意味着任何两句代码都不能同时运行，它们得一个接一个来。在浏览器中，JavaScript 和其他任务共享一个线程，不同的浏览器略有差异，但大体上这些和 JavaScript 共享线程的任务包括重绘、更新样式、用户交互等，所有这些任务操作都会阻塞其他任务。

作为人类，你是多线程的。你可以用多个手指同时敲键盘，也可以一边开车一遍电话。唯一的全局阻塞函数是打喷嚏，打喷嚏期间所有其他事务都会暂停。很烦人对么？尤其是当你开着车打着电话的时候。我们都不喜欢这样打喷嚏的代码。

你应该会用事件加回调的办法来处理这类情况：

```
var img1 = document.querySelector('.img-1');

img1.addEventListener('load', function() {
  // 啊哈图片加载完成
});

img1.addEventListener('error', function() {
  // 哎哟出问题了
});
```

这样就不打喷嚏了。我们添加几个监听函数，请求图片，然后 JavaScript 就停止运行了，直到触发某个监听函数。

上面的例子中唯一的问题是，事件有可能在我们绑定监听器之前就已经发生，所以我们先要检查图片的“complete”属性：

```
var img1 = document.querySelector('.img-1');

function loaded() {
  // 啊哈图片加载完成
}

if (img1.complete) {
  loaded();
}
else {
  img1.addEventListener('load', loaded);
}

img1.addEventListener('error', function() {
  // 哎哟出问题了
});
```

这样还不够，如果在添加监听函数之前图片加载发生错误，我们的监听函数还是白费，不幸的是 DOM 也没有为这个需求提供解决办法。而且，这还只是处理一张图片的情况，如果有一堆图片要处理那就更麻烦了。

事件不是万金油

事件机制最适合处理同一个对象上反复发生的事情——keyup、touchstart 等等。你不需要考虑当绑定监听器之前所发生的事情，当碰到异步请求成功/失败的时候，你想要的通常是这样：

```
img1.callThisIfLoadedOrWhenLoaded(function() {
  // 加载完成
}).orIfFailedCallThis(function() {
  // 加载失败
});

// 以及……
whenAllTheseHaveLoaded([img1, img2]).callThis(function() {
  // 全部加载完成
}).orIfSomeFailedCallThis(function() {
  // 一个或多个加载失败
});
```

这就是 Promise。如果 HTML 图片元素有一个“ready()”方法的话，我们就可以这样：

```
img1.ready().then(function() {
  // 加载完成
}, function() {
  // 加载失败
});

// 以及……
Promise.all([img1.ready(), img2.ready()]).then(function() {
  // 全部加载完成
}, function() {
```

```
  // 一个或多个加载失败
});
```

基本上 Promise 还是有点像事件回调的，除了：

- 一个 Promise 只能成功或失败一次，并且状态无法改变（不能从成功变为失败，反之亦然）
- 如果一个 Promise 成功或者失败之后，你为其添加针对成功/失败的回调，则相应的回调函数会立即执行

这些特性非常适合处理异步操作的成功/失败情景，你无需再担心事件发生的时间点，而只需对其做出响应。

Promise 相关术语

Domenic Denicola 审阅了本文初稿，给我在术语方面打了个“F”，关了禁闭并且责令我打印 States and Fates 一百遍，还写了一封家长信给我父母。即便如此，我还是对术语有些迷糊，不过基本上应该是这样：

一个 Promise 的状态可以是：

肯定（fulfilled） - 该 Promise 对应的操作成功了
否定（rejected） - 该 Promise 对应的操作失败了
等待（pending） - 还没有得到肯定或者否定结果，进行中
结束（settled） - 已经肯定或者否定了

规范里还使用 **thenable** 来描述一个对象是否是“类 Promise”（拥有名为“then”的方法）的。这个术语使我想起来前英格兰足球经理 Terry Venables 所以我尽量少用它。

JavaScript 有了 Promise！

其实已经有一些第三方库实现了 Promise 的功能：

- Q
- when
- WinJS
- RSVP.js

上面这些库和 JavaScript 原生 Promise 都遵守一个通用的、标准化的规范：Promises/A+，jQuery 有个类似的方法叫 Deferred，但不兼容 Promises/A+ 规范，于是会有点小问题，使用需谨慎。jQuery 还有一个 Promise 类型，它其实是 Deferred 的缩减版，所以也有同样问题。

尽管 Promise 的各路实现遵循同一规范，它们的 API 还是各不相同。JavaScript Promise 的 API 比较接近 RSVP.js，如下创建 Promise：

```
var promise = new Promise(function(resolve, reject) {
  // 做一些异步操作的事情，然后……

  if (/* 一切正常 */) {
    resolve("Stuff worked!");
  }
  else {
    reject(Error("It broke"));
  }
});
```

Promise 的构造器接受一个函数作为参数，它会传递给这个回调函数两个变量 resolve 和 reject。在回调函数中做一些异步操作，成功之后调用 resolve，否则调用 reject。

调用 reject 的时候传递给它一个 Error 对象只是个惯例并非必须，这和经典 JavaScript 中的 throw 一样。传递 Error 对象的好处是它包含了调用堆栈，在调试的时候会有点用处。

现在来看看如何使用 Promise：

```
promise.then(function(result) {
  console.log(result); // “完美！”
}, function(err) {
  console.log(err); // Error: "出问题了"
});
```

“then”接受两个参数，成功的时候调用一个，失败的时候调用另一个，两个都是可选的，所以你可以只处理成功的情况或者失败的情况。

JavaScript Promise 最初以“Futures”为名归为 DOM 规范，后来改名为“Promises”，最终纳入 JavaScript 规范。将其加入 JavaScript 而非 DOM 的好处是方便在非浏览器环境中使用，如Node.js（他们会不会在核心API中使用就是另一回事了）。

尽管它被归位 JavaScript 特性，DOM 也很乐意拿过来用。事实上，所有包含异步成功/失败方法的新 DOM API 都会使用 Promise 机制，已经实现的包括 Quota Management, Font Load Events, ServiceWorker, Web MIDI, Streams 等等。

浏览器支持和 Polyfill

目前的浏览器已经（部分）实现了 Promise。

Chrome 32、Opera 19 和 Firefox 29 以上的版本已经默认支持 Promise。由于是在 WebKit 内核中所以我们有理由期待下个版本的 Safari 也会支持，并且 IE 也在[不断的开发](http://status.modern.ie/promiseses6?term=promise)中。

要在这两个浏览器上达到兼容标准 Promise，或者在其他浏览器以及 Node.js 中使用 Promise，可以看看这个 polyfill（gzip 之后 2K）。

与其他库的兼容性

JavaScript Promise 的 API 会把任何包含有 then 方法的对象当作“类 Promise”（或者用术语来说就是 *thenable*。*叹气*）的对象，所以如果你使用的库返回一个 Q Promise，那没问题，无缝融入新的 JavaScript Promise。

尽管，如前所述，jQuery 的 Deferred 对象有点……没什么用，不过幸好还可以转换成标准 Promise，你最好一拿到对象就马上加以转换：

```
var jsPromise = Promise.resolve($.ajax('/whatever.json'));
```

这里 jQuery 的 `$.ajax` 返回一个 Deferred 对象，含有“then”方法，因此 `Promise.resolve` 可以将其转换为 JavaScript Promise。不过有时候 Deferred 对象会给它的回调函数传递多个参数，例如：

```
var jqDeferred = $.ajax('/whatever.json');
```

```
jqDeferred.then(function(response, statusText, xhrObj) {
  // ...
}, function(xhrObj, textStatus, err) {
  // ...
});
```

除了第一个参数，其他都会被 JavaScript Promise 忽略掉：

```
jsPromise.then(function(response) {
  // ...
}, function(xhrObj) {
  // ...
});
```

……还好这通常就是你想要的了，至少你能够用这个办法实现想要的。另外还要注意，jQuery 也没有遵循给否定回调函数传递 Error 对象的惯例。

复杂的异步代码变得简单了

OK，现在我们来写点实际的代码。假设我们想要：

1. 显示一个加载指示图标
2. 加载一篇小说的 JSON，包含小说名和每一章内容的 URL。
3. 在页面中填上小说名
4. 加载所有章节正文
5. 在页面中添加章节正文
6. 停止加载指示

……这个过程中如果发生什么错误了要通知用户，并且把加载指示停掉，不然它就会不停转下去，令人眼晕，或者搞坏界面什么的。

当然了，你不会用 JavaScript 去这么繁琐地显示一篇文章，直接输出 HTML 要快得多，不过这个流程是非常典型的 API 请求模式：获取多个数据，当它们全部完成之后再做一些事情。

首先，搞定从网络加载数据的步骤：

将 Promise 用于 XMLHttpRequest

只要能保持向后兼容，现有 API 都会更新以支持 Promise，`XMLHttpRequest` 是重点考虑对象之一。不过现在我们先来写个 GET 请求：

```
function get(url) {
  // 返回一个新的 Promise
  return new Promise(function(resolve, reject) {
    // 经典 XHR 操作
    var req = new XMLHttpRequest();
    req.open('GET', url);

    req.onload = function() {
      // 当发生 404 等状况的时候调用此函数
      // 所以先检查状态码
```

```
      if (req.status == 200) {
        // 以响应文本为结果，完成此 Promise
        resolve(req.response);
      }
      else {
        // 否则就以状态码为结果否定掉此 Promise
        // （提供一个有意义的 Error 对象）
        reject(Error(req.statusText));
      }
    };

    // 网络异常的处理方法
    req.onerror = function() {
      reject(Error("Network Error"));
    };

    // 发出请求
    req.send();
  });
}
```

现在可以调用它了：

```
get('story.json').then(function(response) {
  console.log("Success!", response);
}, function(error) {
  console.error("Failed!", error);
});
```

点击这里查看代码。现在我们不需要手敲 `XMLHttpRequest` 就可以直接发起 HTTP 请求，这样感觉好多了，能少看一次这个狂驼峰命名的 `XMLHttpRequest` 我就多快乐一点。

链式调用

“then”的故事还没完，你可以把这些“then”串联起来修改结果或者添加进行更多异步操作。

值的处理

你可以对结果做些修改然后返回一个新值：

```
var promise = new Promise(function(resolve, reject) {
  resolve(1);
});

promise.then(function(val) {
  console.log(val); // 1
  return val + 2;
}).then(function(val) {
  console.log(val); // 3
});
```

回到前面的代码：

```
get('story.json').then(function(response) {
  console.log("Success!", response);
```

```
});
```

收到的响应是一个纯文本的 JSON，我们可以修改 get 函数，设置 responseType 为 JSON 来指定服务器响应格式，也可以在 Promise 的世界里搞定这个问题：

```
get('story.json').then(function(response) {
  return JSON.parse(response);
}).then(function(response) {
  console.log("Yey JSON!", response);
});
```

既然 JSON.parse 只接收一个参数，并返回转换后的结果，我们还可以再精简一下：

```
get('story.json').then(JSON.parse).then(function(response) {
  console.log("Yey JSON!", response);
});
```

点击这里查看代码运行页面，打开控制台查看输出结果。 事实上，我们可以把 getJSON 函数写得超级简单：

```
function getJSON(url) {
  return get(url).then(JSON.parse);
}
```

getJSON 会返回一个获取 JSON 并加以解析的 Promise。

队列的异步操作

你也可以把“then”串联起来依次执行异步操作。

当你从“then”的回调函数返回的时候，这里有点小魔法。如果你返回一个值，它就会被传给下一个“then”的回调；而如果你返回一个“类 Promise”的对象，则下一个“then”就会等待这个 Promise 明确结束（成功/失败）才会执行。例如：

```
getJSON('story.json').then(function(story) {
  return getJSON(story.chapterUrls[0]);
}).then(function(chapter1) {
  console.log("Got chapter 1!", chapter1);
});
```

这里我们发起一个对“story.json”的异步请求，返回给我们更多 URL，然后我们会请求其中的第一个。Promise 开始首次显现出相较事件回调的优越性了。你甚至可以写一个抓取章节内容的独立函数：

```
var storyPromise;

function getChapter(i) {
  storyPromise = storyPromise || getJSON('story.json');

  return storyPromise.then(function(story) {
    return getJSON(story.chapterUrls[i]);
  })
}
```

```
// 用起来非常简单:
getChapter(0).then(function(chapter) {
  console.log(chapter);
  return getChapter(1);
}).then(function(chapter) {
  console.log(chapter);
});
```

我们一开始并不加载 story.json，直到第一次 `getChapter`，而以后每次 `getChapter` 的时候都可以重用已经加载完成的 story Promise，所以 story.json 只需要请求一次。Promise 好棒！

错误处理

前面已经看到，“then”接受两个参数，一个处理成功，一个处理失败（或者说肯定和否定，按 Promise 术语）：

```
get('story.json').then(function(response) {
  console.log("Success!", response);
}, function(error) {
  console.log("Failed!", error);
});
```

你还可以使用“catch”：

```
get('story.json').then(function(response) {
  console.log("Success!", response);
}).catch(function(error) {
  console.log("Failed!", error);
});
```

这里的 catch 并无任何特殊之处，只是 `then(undefined, func)` 的语法糖衣，更直观一点而已。注意上面两段代码的行为不仅相同，后者相当于：

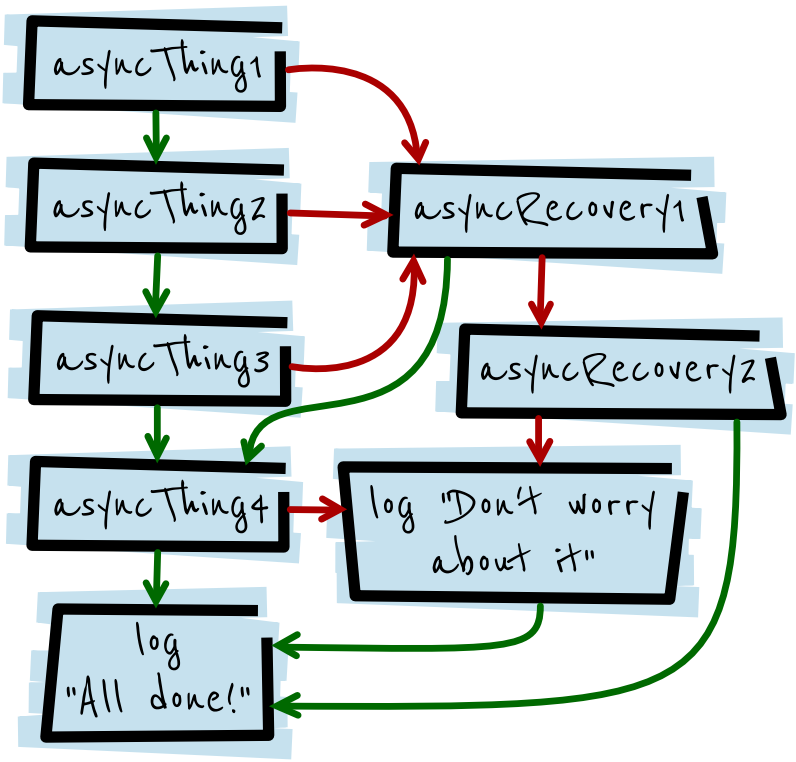
```
get('story.json').then(function(response) {
  console.log("Success!", response);
}).then(undefined, function(error) {
  console.log("Failed!", error);
});
```

差异不大，但意义非凡。Promise 被否定之后会跳转到之后第一个配置了否定回调的 then（或 catch，一样的）。对于 `then(func1, func2)` 来说，必会调用 `func1` 或 `func2` 之一，但绝不会两个都调用。而 `then(func1).catch(func2)` 这样，如果 `func1` 返回否定的话 `func2` 也会被调用，因为他们是链式调用中独立的两个步骤。看下面这段代码：

```
asyncThing1().then(function() {
  return asyncThing2();
}).then(function() {
  return asyncThing3();
}).catch(function(err) {
  return asyncRecovery1();
}).then(function() {
  return asyncThing4();
```

```
}, function(err) {
  return asyncRecovery2();
}).catch(function(err) {
  console.log("Don't worry about it");
}).then(function() {
  console.log("All done!");
});
```

这段流程非常像 JavaScript 的 try/catch 组合，“try”代码块中发生的错误会立即跳转到“catch”代码块。这是上面那段代码的流程图（我最爱流程图了）：



绿线是肯定的 Promise 流程，红线是否定的 Promise 流程。

JavaScript 异常和 Promise

Promise 的否定回调可以由 Promise.reject() 触发，也可以由构造器回调中抛出的错误触发：

```
var jsonPromise = new Promise(function(resolve, reject) {
  // 如果数据格式不对的话 JSON.parse 会抛出错误
  // 可以作为隐性的否定结果：
  resolve(JSON.parse("This ain't JSON"));
});

jsonPromise.then(function(data) {
  // 永远不会发生：
  console.log("It worked!", data);
}).catch(function(err) {
  // 这才是真相：
  console.log("It failed!", err);
});
```

这意味着你可以把所有 Promise 相关操作都放在它的构造函数回调中进行，这样发生任何错误都能捕捉到并且触发 Promise 否定。

“then”回调中抛出的错误也一样：

```
get('/').then(JSON.parse).then(function() {
  // This never happens, '/' is an HTML page, not JSON
  // so JSON.parse throws
  console.log("It worked!", data);
}).catch(function(err) {
  // Instead, this happens:
  console.log("It failed!", err);
});
```

实践错误处理

回到我们的故事和章节，我们用 `catch` 来捕捉错误并显示给用户：

```
getJSON('story.json').then(function(story) {
  return getJSON(story.chapterUrls[0]);
}).then(function(chapter1) {
  addHtmlToPage(chapter1.html);
}).catch(function() {
  addTextToPage("Failed to show chapter");
}).then(function() {
  document.querySelector('.spinner').style.display = 'none';
});
```

如果请求 `story.chapterUrls[0]` 失败（http 500 或者用户掉线什么的）了，它会跳过之后所有针对成功的回调，包括 `getJSON` 中将响应解析为 JSON 的回调，和这里把第一张内容添加到页面里的回调。JavaScript 的执行会进入 catch 回调。结果就是前面任何章节请求出错，页面上都会显示“Failed to show chapter”。

和 JavaScript 的 try/catch 一样，捕捉到错误之后，接下来的代码会继续执行，按计划把加载指示器给停掉。上面的代码就是下面这段的非阻塞异步版：

```
try {
  var story = getJSONSync('story.json');
  var chapter1 = getJSONSync(story.chapterUrls[0]);
  addHtmlToPage(chapter1.html);
}
catch (e) {
  addTextToPage("Failed to show chapter");
}

document.querySelector('.spinner').style.display = 'none';
```

如果只是要捕捉异常做记录输出而不打算在用户界面上对错误进行反馈的话，只要抛出 Error 就行了，这一步可以放在 `getJSON` 中：

```
function getJSON(url) {
  return get(url).then(JSON.parse).catch(function(err) {
    console.log("getJSON failed for", url, err);
    throw err;
  });
}
```

现在我们已经搞定第一章了，接下来搞定全部章节。

并行和串行 —— 鱼与熊掌兼得

异步的思维方式并不符合直觉，如果你觉得起步困难，那就试试先写个同步的方法，就像这个：

```
try {
  var story = getJSONSync('story.json');
  addHtmlToPage(story.heading);

  story.chapterUrls.forEach(function(chapterUrl) {
    var chapter = getJSONSync(chapterUrl);
    addHtmlToPage(chapter.html);
  });

  addTextToPage("All done");
}
catch (err) {
  addTextToPage("Argh, broken: " + err.message);
}

document.querySelector('.spinner').style.display = 'none';
```

它执行起来完全正常（查看示例）！不过它是同步的，在加载内容时会卡住整个浏览器。要让它异步工作的话，我们用 then 把它们一个接一个串起来：

```
getJSON('story.json').then(function(story) {
  addHtmlToPage(story.heading);

  // TODO: 获取并显示 story.chapterUrls 中的每个 url
}).then(function() {
  // 全部完成啦!
  addTextToPage("All done");
}).catch(function(err) {
  // 如果过程中有任何不对劲的地方
  addTextToPage("Argh, broken: " + err.message);
}).then(function() {
  // 无论如何要把 spinner 隐藏掉
  document.querySelector('.spinner').style.display = 'none';
});
```

那么我们如何遍历章节的 URL 并且依次请求？这样**是不行的**：

```
story.chapterUrls.forEach(function(chapterUrl) {
  // Fetch chapter
  getJSON(chapterUrl).then(function(chapter) {
    // and add it to the page
    addHtmlToPage(chapter.html);
  });
});
```

“forEach” 没有对异步操作的支持，所以我们的故事章节会按照它们加载完成的顺序显示，基本上《低俗小说》就是这么写出来的。我们不写低俗小说，所以得修正它：

创建序列

我们要把章节 URL 数组转换成 Promise 的序列，还是用 `then`：

```
// 从一个完成状态的 Promise 开始
var sequence = Promise.resolve();

// 遍历所有章节的 url
story.chapterUrls.forEach(function(chapterUrl) {
  // 从 sequence 开始把操作接龙起来
  sequence = sequence.then(function() {
    return getJSON(chapterUrl);
  }).then(function(chapter) {
    addHtmlToPage(chapter.html);
  });
});
```

这是我们第一次用到 `Promise.resolve`，它会依据你传的任何值返回一个 Promise。如果你传给它一个类 Promise 对象（带有 `then` 方法），它会生成一个带有同样肯定/否定回调的 Promise，基本上就是克隆。如果传给它任何别的值，如 `Promise.resolve('Hello')`，它会创建一个以这个值为完成结果的 Promise，如果不传任何值，则以 undefined 为完成结果。

还有一个对应的 `Promise.reject(val)`，会创建以你传入的参数（或 undefined）为否定结果的 Promise。

我们可以用 array.reduce 精简一下上面的代码：

```
// 遍历所有章节的 url
story.chapterUrls.reduce(function(sequence, chapterUrl) {
  // 从 sequence 开始把操作接龙起来
  return sequence.then(function() {
    return getJSON(chapterUrl);
  }).then(function(chapter) {
    addHtmlToPage(chapter.html);
  });
}, Promise.resolve());
```

它和前面的例子功能一样，但是不需要显式声明 `sequence` 变量。`reduce` 回调会依次应用在每个数组元素上，第一轮里的“sequence”是 `Promise.resolve()`，之后的调用里“sequence”就是上次函数执行的的结果。`array.reduce` 非常适合用于把一组值归并处理为一个值，正是我们现在对 Promise 的用法。

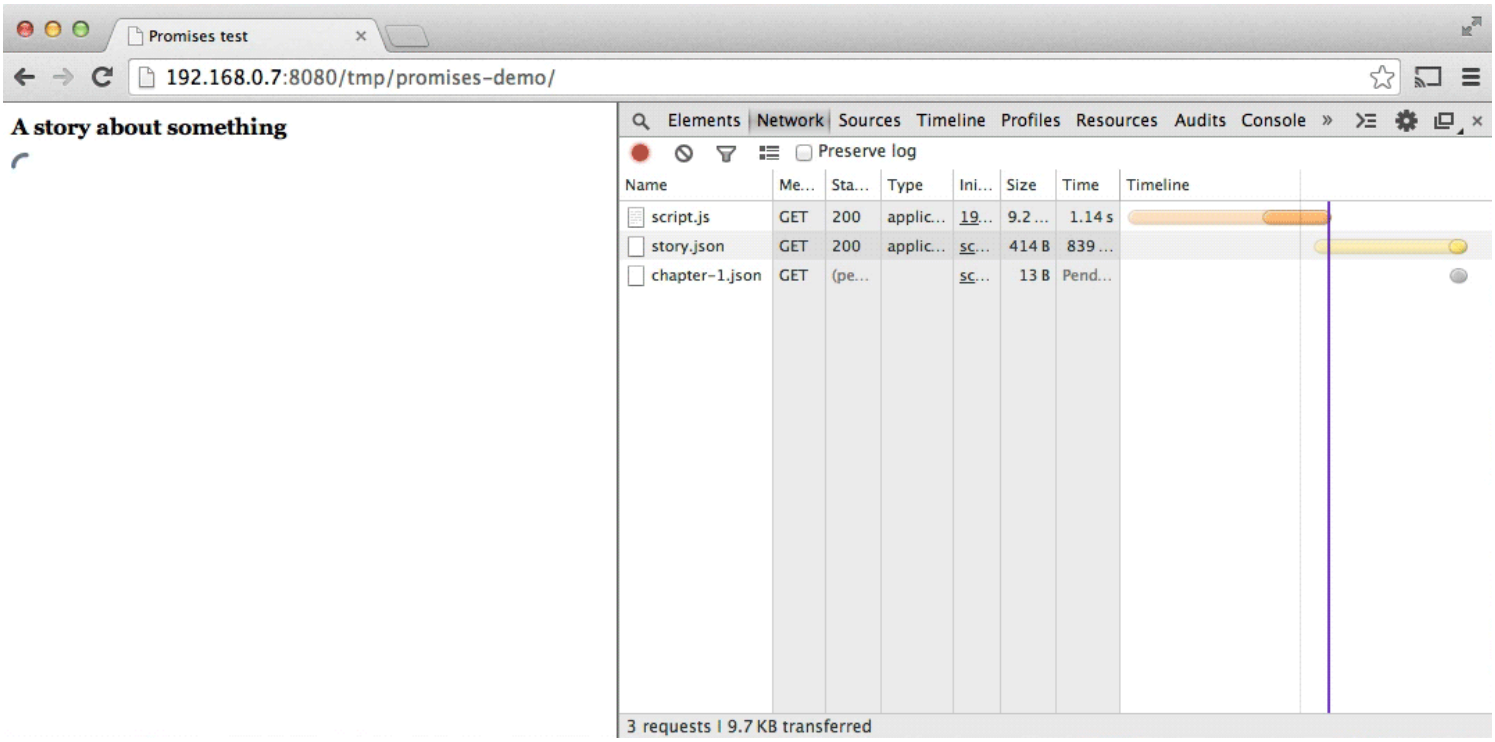
汇总上面的代码：

```
getJSON('story.json').then(function(story) {
  addHtmlToPage(story.heading);

  return story.chapterUrls.reduce(function(sequence, chapterUrl) {
    // 当前一个章节的 Promise 完成之后……
    return sequence.then(function() {
      // ……获取下一章
      return getJSON(chapterUrl);
    }).then(function(chapter) {
      // 并添加到页面
      addHtmlToPage(chapter.html);
    });
  }, Promise.resolve());
}).then(function() {
  // 现在全部完成了!
  addTextToPage("All done");
}).catch(function(err) {
  // 如果过程中发生任何错误
  addTextToPage("Argh, broken: " + err.message);
}).then(function() {
```

```
  // 保证 spinner 最终会隐藏
  document.querySelector('.spinner').style.display = 'none';
});
```

查看代码运行示例，前面的同步代码改造成了完全异步的版本。我们还可以更进一步。现在页面加载的效果是这样：



浏览器很擅长同时加载多个文件，我们这种一个接一个下载章节的方法非常低效率。我们希望同时下载所有章节，全部完成后一次搞定，正好就有这么个 API：

```
Promise.all(arrayOfPromises).then(function(arrayOfResults) {
  //...
});
```

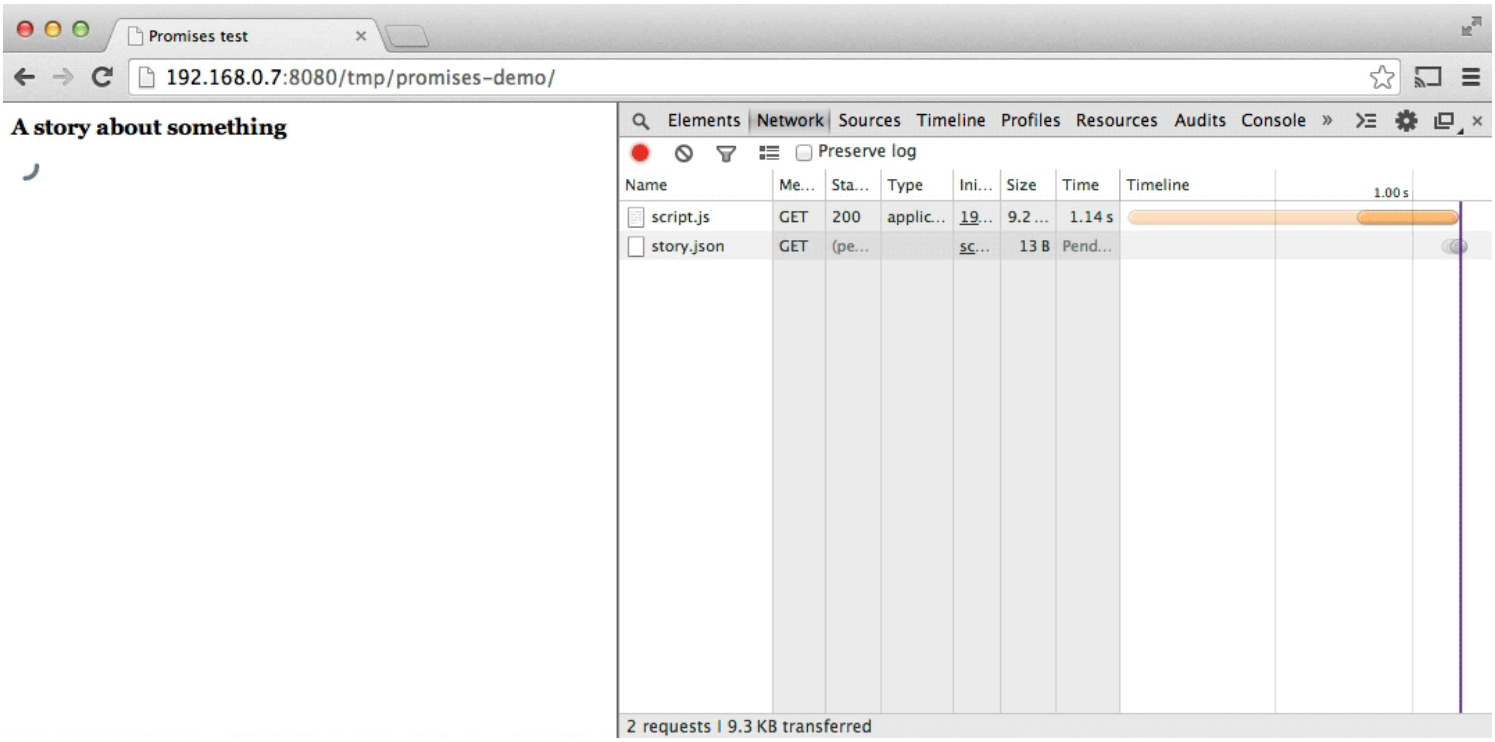
`Promise.all` 接受一个 Promise 数组为参数，创建一个当所有 Promise 都完成之后就完成的 Promise，它的完成结果是一个数组，包含了所有先前传入的那些 Promise 的完成结果，顺序和将它们传入的数组顺序一致。

```
getJSON('story.json').then(function(story) {
  addHtmlToPage(story.heading);

  // 接受一个 Promise 数组并等待他们全部完成
  return Promise.all(
    // 把章节 URL 数组转换成对应的 Promise 数组
    story.chapterUrls.map(getJSON)
  );
}).then(function(chapters) {
  // 现在我们有了顺序的章节 JSON，遍历它们……
  chapters.forEach(function(chapter) {
    // ……并添加到页面中
    addHtmlToPage(chapter.html);
  });
  addTextToPage("All done");
}).catch(function(err) {
  // 捕获过程中的任何错误
  addTextToPage("Argh, broken: " + err.message);
```

```
}).then(function() {
  document.querySelector('.spinner').style.display = 'none';
});
```

根据连接状况，改进的代码会比顺序加载方式提速数秒（查看示例），甚至代码行数也更少。章节加载完成的顺序不确定，但它们显示在页面上的顺序准确无误。



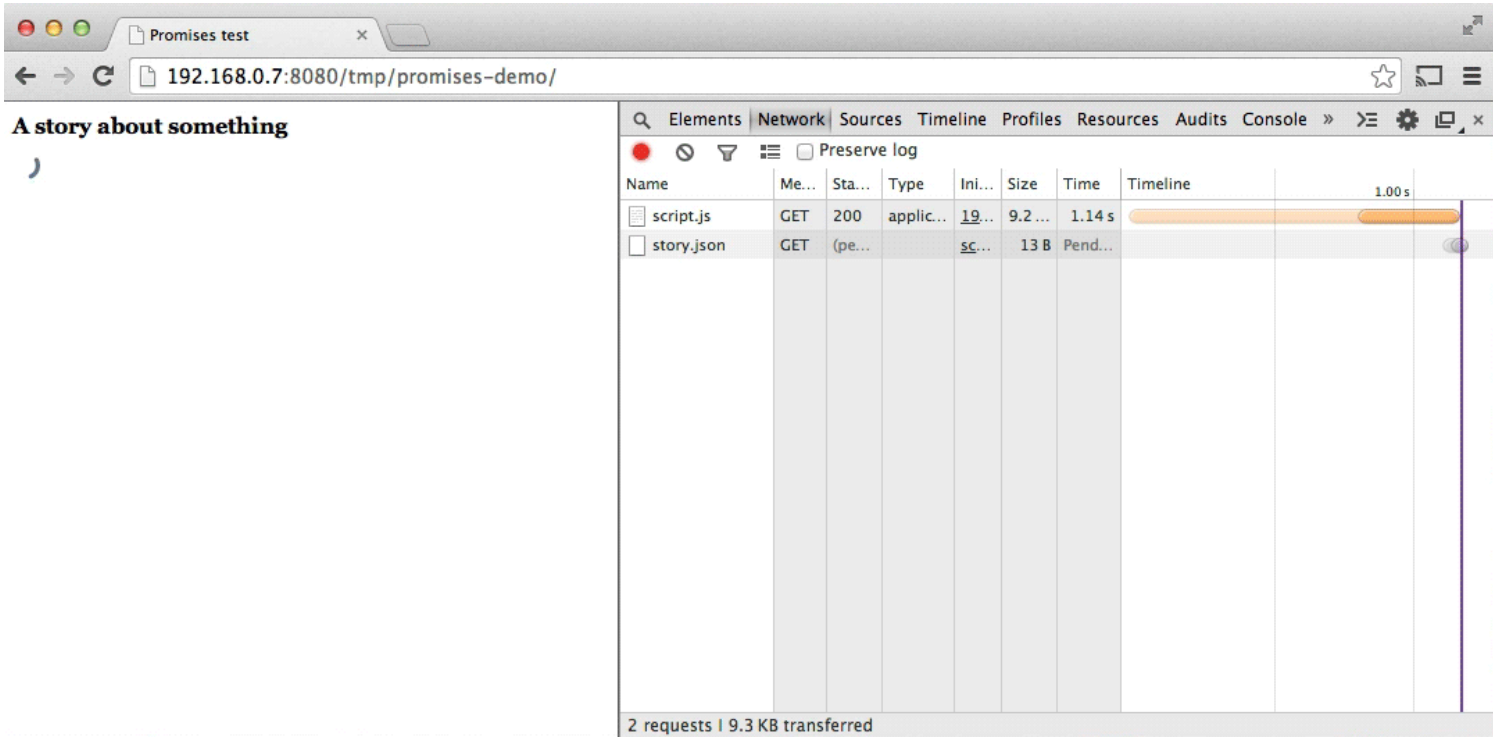
然而这样还是有提高空间。当第一章内容加载完毕我们可以立即填进页面，这样用户可以在其他加载任务尚未完成之前就开始阅读；当第三章到达的时候我们不动声色，第二章也到达之后我们再把第二章和第三章内容填入页面，以此类推。

为了达到这样的效果，我们同时请求所有的章节内容，然后创建一个序列依次将其填入页面：

```
getJSON('story.json').then(function(story) {
  addHtmlToPage(story.heading);

  // 把章节 URL 数组转换成对应的 Promise 数组
  // 这样就可以并行加载它们
  return story.chapterUrls.map(getJSON)
    .reduce(function(sequence, chapterPromise) {
      // 使用 reduce 把这些 Promise 接龙
      // 以及将章节内容添加到页面
      return sequence.then(function() {
        // 等待当前 sequence 中所有章节和本章节的数据到达
        return chapterPromise;
      }).then(function(chapter) {
        addHtmlToPage(chapter.html);
      });
    }, Promise.resolve());
}).then(function() {
  addTextToPage("All done");
}).catch(function(err) {
  // 捕获过程中的任何错误
  addTextToPage("Argh, broken: " + err.message);
}).then(function() {
  document.querySelector('.spinner').style.display = 'none';
});
```

哈哈（查看示例），鱼与熊掌兼得！加载所有内容的时间未变，但用户可以更早看到第一章。



这个小例子中各部分章节加载差不多同时完成，逐章显示的策略在章节内容很多的时候优势将会更加显著。

上面的代码如果用 Node.js 风格的回调或者事件机制实现的话代码量大约要翻一倍，更重要的是可读性也不如此例。然而，Promise 的厉害不止于此，和其他 ES6 的新功能结合起来还能更加高效……

附赠章节：Promise 和 Generator

接下来的内容涉及到一大堆 ES6 的新特性，不过对于现在应用 Promise 来说并非必须，把它当作接下来的第二部豪华续集的预告片来看就好了。

ES6 还给我们带来了 Generator，允许函数在特定地方像 return 一样退出，但是稍后又能恢复到这个位置和状态上继续执行：

```
function *addGenerator() {
  var i = 0;
  while (true) {
    i += yield i;
  }
}
```

注意函数名前的星号，这表示该函数是一个 Generator。关键字 yield 标记了暂停/继续的位置，使用方法像这样：

```
var adder = addGenerator();
adder.next().value; // 0
adder.next(5).value; // 5
adder.next(5).value; // 10
adder.next(5).value; // 15
adder.next(50).value; // 65
```

这对 Promise 有什么用呢？你可以用这种暂停/继续的机制写出来和同步代码看上去差不多（理解起来也一样简

单）的代码。下面是一个辅助函数（helper function），我们在 `yield` 位置等待 Promise 完成：

```
function spawn(generatorFunc) {
  function continuer(verb, arg) {
    var result;
    try {
      result = generator[verb](arg);
    } catch (err) {
      return Promise.reject(err);
    }
    if (result.done) {
      return result.value;
    } else {
      return Promise.resolve(result.value).then(onFulfilled, onRejected);
    }
  }
  var generator = generatorFunc();
  var onFulfilled = continuer.bind(continuer, "next");
  var onRejected = continuer.bind(continuer, "throw");
  return onFulfilled();
}
```

这段代码[原样拷贝自 Q](#)，只是改成 JavaScript Promise 的 API。把我们前面的最终方案和 ES6 最新特性结合在一起之后：

```
spawn(function *() {
  try {
    // 'yield' 会执行一个异步的等待，返回这个 Promise 的结果
    let story = yield getJSON('story.json');
    addHtmlToPage(story.heading);

    // 把章节 URL 数组转换成对应的 Promise 数组
    // 以便并行加载
    let chapterPromises = story.chapterUrls.map(getJSON);

    for (let chapterPromise of chapterPromises) {
      // 等待每个章节加载完成，将其添加至页面
      let chapter = yield chapterPromise;
      addHtmlToPage(chapter.html);
    }

    addTextToPage("All done");
  }
  catch (err) {
    // try/catch 即可。否定的 Promise 会在这里抛出。
    addTextToPage("Argh, broken: " + err.message);
  }
  document.querySelector('.spinner').style.display = 'none';
});
```

功能完全一样，读起来要简单得多。这个例子目前可以在 Chrome Canary 中运行（[查看示例](#)），不过你得先到 **about:flags** 中开启 **Enable experimental JavaScript** 选项。

这里用到了一堆 ES6 的新语法：Promise、Generator、let、for-of。当我们把 `yield` 应用在一个 Promise 上，spawn 辅助函数会等待 Promise 完成，然后才返回最终的值。如果 Promise 给出否定结果，spawn 中的 yield 则会抛出一个异常，我们可以用 try/catch 捕捉到。这样写异步代码真是超级简单！

Promise API 参考

除非额外注明，Chrome、Opera 和 Firefox（nightly）均支持下列所有方法。这个 polyfill 则在所有浏览器内实现了同样的接口。

静态方法

`Promise.resolve(promise);`
返回一个 Promise（当且仅当 `promise.constructor == Promise`）

`Promise.resolve(thenable);`
从 thenable 对象创建一个新的 Promise。一个 thenable（类 Promise）对象是一个带有“then”方法的对象。

`Promise.resolve(obj);`
创建一个以 obj 为肯定结果的 Promise。

`Promise.reject(obj);`
创建一个以 obj 为否定结果的 Promise。为了一致性和调试便利（如堆栈追踪），obj 应该是一个 Error 实例对象。

`Promise.all(array);`
创建一个 Promise，当且仅当传入数组中的所有 Promise 都肯定之后才肯定，如果遇到数组中的任何一个 Promise 以否定结束，则抛出否定结果。每个数组元素都会首先经过 Promise.resolve，所以数组可以包含类 Promise 对象或者其他对象。肯定结果是一个数组，包含传入数组中每个 Promise 的肯定结果（且保持顺序）；否定结果是传入数组中第一个遇到的否定结果。

`Promise.race(array);`
创建一个 Promise，当数组中的任意对象肯定时将其结果作为肯定结束，或者当数组中任意对象否定时将其结果作为否定结束。

备注：我不大确定这个接口是否有用，我更倾向于一个 `Promise.all` 的对立方法，仅当所有数组元素全部给出否定的时候才抛出否定结果

构造器

```
new Promise(function(resolve, reject) {});
```

`resolve(thenable)`
你的 Promise 将会根据这个 “thenable” 对象的结果而返回肯定/否定结果

`resolve(obj)`
你的 Promise 将会以 `obj` 作为肯定结果完成

`reject(obj)`
你的 Promise 将会以 `obj` 作为否定结果完成。出于一致性和调试（如栈追踪）方便，`obj` 应该是一个 Error 对象的实例。构造器的回调函数中抛出的错误会被立即传递给 `reject()`。

实例方法

promise.then(onFulfilled, onRejected)
当 promise 以肯定结束时会调用 onFulfilled。 当 promise 以否定结束时会调用 onRejected。 这两个参数都是可选的，当任意一个未定义时，对它的调用会跳转到 then 链的下一个 onFulfilled/onRejected 上。 这两个回调函数均只接受一个参数，肯定结果或者否定原因。 当 Promise.resolve 肯定结束之后，then 会返回一个新的 Promise，这个 Promise 相当于你从 onFulfilled/onRejected 中返回的值。如果回调中抛出任何错误，返回的 Promise 也会以此错误作为否定结果结束。

promise.catch(onRejected)
promise.then(undefined, onRejected) 的语法糖。

非常感谢 Anne van Kesteren，Domenic Denicola，Tom Ashworth，Remy Sharp，Addy Osmani，Arthur Evans 以及 and Yutaka Hirano 对本文做出的贡献（审阅/纠错或建议）。