--

[小结 1](#_Toc19251)

[Javascript异步编程的4种方法 1](#_Toc24975)

[回调函数 1](#_Toc9906)

[事件监听 1](#_Toc4497)

[发布/订阅 1](#_Toc26956)

[Promises对象 1](#_Toc901)

[Generator 函数，async函数 2](#_Toc20817)

[nodejs异步趣谈 2](#_Toc20600)

[异步编程 观察者模式的简单实现 3](#_Toc8079)

[从异步ajax体验观察者模式 3](#_Toc31252)

[jQuery观察者模式的扩展方法 3](#_Toc19160)

[jQuery的deferred对象详解 3](#_Toc20616)

[api总结 3](#_Toc19637)

[ES6 Promise 和 jQuery Deferred 的异同 4](#_Toc4325)

[deferred其他使用方式 5](#_Toc14556)

[使用deferred对象写JS动画非常方便 5](#_Toc18256)

[模拟$.Deferred 的实现原理 5](#_Toc7452)

[Promise 5](#_Toc7082)

[API小结 6](#_Toc21981)

[.then() 9](#_Toc9379)

[promise注意点 10](#_Toc9058)

[catch() then(null, ...)并非完全等价 11](#_Toc31836)

[promises vs promises factories 11](#_Toc28280)

[异步编程 promise模式 的简单实现 13](#_Toc27383)

[说明 13](#_Toc24371)

[代码实现 13](#_Toc24263)

[代码分析 13](#_Toc16095)

[异步控制工具 14](#_Toc26379)

[比较 14](#_Toc9823)

[q.js 14](#_Toc19785)

[promise的传递 14](#_Toc17543)

[方法传递 15](#_Toc2765)

[.fail 15](#_Toc9264)

[.progress 15](#_Toc727)

[promise链 15](#_Toc22560)

[promise组合 16](#_Toc28538)

[结束promise链 16](#_Toc25338)

[其他 16](#_Toc7744)

[bluebird 17](#_Toc9162)

[promise-异步编程 17](#_Toc17159)

[升级 17](#_Toc5107)

[Generator 17](#_Toc6838)

[Generator 函数的异步应用 17](#_Toc18822)

[Generator 18](#_Toc2373)

[1 18](#_Toc12682)

[2 18](#_Toc19946)

[3 18](#_Toc13615)

[yield\* 19](#_Toc5730)

[CO 19](#_Toc2359)

[co thunkify 20](#_Toc6938)

[async await 22](#_Toc23871)

[async 函数的含义和用法 22](#_Toc19096)

[Async/Await使用案例 23](#_Toc21922)

[demo-以读取文件为例说明 24](#_Toc12275)

向WindJS致敬\_Node异步流程控制4

<http://www.tuicool.com/articles/yIbEj2Z>

wind.js助力异步编程

<http://blog.fens.me/nodejs-async-windjs/>

小结

Javascript异步编程的4种方法

|  |  |  |
| --- | --- | --- |
|  | <http://www.ruanyifeng.com/blog/2012/12/asynchronous%EF%BC%BFjavascript.html>  参考链接\* [Asynchronous JS: Callbacks, Listeners, Control Flow Libs and Promises](http://sporto.github.com/blog/2012/12/09/callbacks-listeners-promises/" \t "http://www.ruanyifeng.com/blog/2012/12/_blank)  <https://zhuanlan.zhihu.com/p/24444262>  最后谈一次 JavaScript 异步编程 | |
|  | 你可能知道，Javascript语言的执行环境是"单线程"(single thread)。  所谓"单线程"，就是指一次只能完成一件任务。如果有多个任务，就必须排队，前面一个任务完成，再执行后面一个任务，以此类推。  这种模式的好处是实现起来比较简单，执行环境相对单纯；坏处是只要有一个任务耗时很长，后面的任务都必须排队等着，会拖延整个程序的执行。常见的浏览器无响应(假死)，往往就是因为某一段Javascript代码长时间运行(比如死循环)，导致整个页面卡在这个地方，其他任务无法执行。  为了解决这个问题，Javascript语言将任务的执行模式分成两种：同步(Synchronous)和异步(Asynchronous)。  "同步模式"就是上一段的模式，后一个任务等待前一个任务结束，然后再执行，程序的执行顺序与任务的排列顺序是一致的、同步的；"异步模式"则完全不同，每一个任务有一个或多个回调函数(callback)，前一个任务结束后，不是执行后一个任务，而是执行回调函数，后一个任务则是不等前一个任务结束就执行，所以程序的执行顺序与任务的排列顺序是不一致的、异步的。  "异步模式"非常重要。在浏览器端，耗时很长的操作都应该异步执行，避免浏览器失去响应，最好的例子就是Ajax操作。在服务器端，"异步模式"甚至是唯一的模式，因为执行环境是单线程的，如果允许同步执行所有http请求，服务器性能会急剧下降，很快就会失去响应。  本文总结了"异步模式"编程的4种方法，理解它们可以让你写出结构更合理、性能更出色、维护更方便的Javascript程序。 | |
| 回调函数 | 这是异步编程最基本的方法。  假定有两个函数f1和f2，后者等待前者的执行结果。  　　f1();  　　f2();  如果f1是一个很耗时的任务，可以考虑改写f1，把f2写成f1的回调函数。  　　function f1(callback){  　　　　setTimeout(function () {  　　　　　　// f1的任务代码  　　　　　　callback();  　　　　}, 1000);  　　}  执行代码就变成下面这样：  　　f1(f2);  采用这种方式，我们把同步操作变成了异步操作，f1不会堵塞程序运行，相当于先执行程序的主要逻辑，将耗时的操作推迟执行。  回调函数的优点是简单、容易理解和部署，缺点是不利于代码的阅读和维护，各个部分之间高度[耦合](http://en.wikipedia.org/wiki/Coupling_(computer_programming)" \t "http://www.ruanyifeng.com/blog/2012/12/_blank)(Coupling)，流程会很混乱，而且每个任务只能指定一个回调函数。 | |
| 事件监听 | 另一种思路是采用事件驱动模式。任务的执行不取决于代码的顺序，而取决于某个事件是否发生。  还是以f1和f2为例。首先，为f1绑定一个事件(这里采用的jQuery的[写法](http://api.jquery.com/on/" \t "http://www.ruanyifeng.com/blog/2012/12/_blank))。  　　f1.on('done', f2);  上面这行代码的意思是，当f1发生done事件，就执行f2。然后，对f1进行改写：  　　function f1(){  　　　　setTimeout(function () {  　　　　　　// f1的任务代码  　　　　　　f1.trigger('done');  　　　　}, 1000);  　　}  f1.trigger('done')表示，执行完成后，立即触发done事件，从而开始执行f2。  这种方法的优点是比较容易理解，可以绑定多个事件，每个事件可以指定多个回调函数，而且可以["去耦合"](http://en.wikipedia.org/wiki/Decoupling" \t "http://www.ruanyifeng.com/blog/2012/12/_blank)(Decoupling)，有利于实现[模块化](http://www.ruanyifeng.com/blog/2012/10/javascript_module.html" \t "http://www.ruanyifeng.com/blog/2012/12/_blank)。缺点是整个程序都要变成事件驱动型，运行流程会变得很不清晰。 | |
| 发布/订阅 | 上一节的"事件"，完全可以理解成"信号"。  我们假定，存在一个"信号中心"，某个任务执行完成，就向信号中心"发布"(publish)一个信号，其他任务可以向信号中心"订阅"(subscribe)这个信号，从而知道什么时候自己可以开始执行。这就叫做["发布/订阅模式"](http://en.wikipedia.org/wiki/Publish-subscribe_pattern" \t "http://www.ruanyifeng.com/blog/2012/12/_blank)(publish-subscribe pattern)，又称["观察者模式"](http://en.wikipedia.org/wiki/Observer_pattern" \t "http://www.ruanyifeng.com/blog/2012/12/_blank)(observer pattern)。  这个模式有多种[实现](http://msdn.microsoft.com/en-us/magazine/hh201955.aspx" \t "http://www.ruanyifeng.com/blog/2012/12/_blank)，下面采用的是Ben Alman的[Tiny Pub/Sub](https://gist.github.com/661855" \t "http://www.ruanyifeng.com/blog/2012/12/_blank)，这是jQuery的一个插件。  首先，f2向"信号中心"jQuery订阅"done"信号。  　　jQuery.subscribe("done", f2);  然后，f1进行如下改写：  　　function f1(){  　　　　setTimeout(function () {  　　　　　　// f1的任务代码  　　　　　　jQuery.publish("done");  　　　　}, 1000);  　　}  jQuery.publish("done")的意思是，f1执行完成后，向"信号中心"jQuery发布"done"信号，从而引发f2的执行。  此外，f2完成执行后，也可以取消订阅(unsubscribe)。  　　jQuery.unsubscribe("done", f2);  这种方法的性质与"事件监听"类似，但是明显优于后者。因为我们可以通过查看"消息中心"，了解存在多少信号、每个信号有多少订阅者，从而监控程序的运行。 | |
| Promises对象 | Promises对象是CommonJS工作组提出的一种规范，目的是为异步编程提供[统一接口](http://wiki.commonjs.org/wiki/Promises/A" \t "http://www.ruanyifeng.com/blog/2012/12/_blank)。  简单说，它的思想是，每一个异步任务返回一个Promise对象，该对象有一个then方法，允许指定回调函数。比如，f1的回调函数f2,可以写成：  f1().then(f2);  f1要进行如下改写(这里使用的是jQuery的[实现](http://www.ruanyifeng.com/blog/2011/08/a_detailed_explanation_of_jquery_deferred_object.html" \t "http://www.ruanyifeng.com/blog/2012/12/_blank))：  　　function f1(){  　　　　var dfd = $.Deferred();  　　　　setTimeout(function () {  　　　　　　// f1的任务代码  　　　　　　dfd.resolve();  　　　　}, 500);  　　　　return dfd.promise;  　　}  这样写的优点在于，回调函数变成了链式写法，程序的流程可以看得很清楚，而且有一整套的[配套方法](http://api.jquery.com/category/deferred-object/" \t "http://www.ruanyifeng.com/blog/2012/12/_blank)，可以实现许多强大的功能。  比如，指定多个回调函数：  f1().then(f2).then(f3);  再比如，指定发生错误时的回调函数：  f1().then(f2).fail(f3);  而且，它还有一个前面三种方法都没有的好处：如果一个任务已经完成，再添加回调函数，该回调函数会立即执行。所以，你不用担心是否错过了某个事件或信号。这种方法的缺点就是编写和理解，都相对比较难。 | |
|  | // 没有 promise  a(getResultFromA, (aResult, err) => {  if (!err) {  b(getResultFromB, (bResult, err) => {  if (!err) {  c(getResultFromC, (cResult, err) => {  if (!err) {  // do something  } else {  throw err  }  })  } else {  throw err  }  })  } else {  throw err  }  }) | // 用了 promise 后  new promise((resolve, reject) => {  a(getResultFromA, (aResult, err) => {  if (!err) resolve(aResult) else reject(err)  })  })  .then(data => {  return new Promise((resolve, reject) => {  b(getResultFromB, (bResult, err) => {  if (!err) resolve(bResult) else reject(err)  })  }  })  .then(data => {  return new Promise((resolve, reject) => {  c(getResultFromC, (cResult, err) => {  if (!err) resolve(cResult) else reject(err)  })  }  })  .then(data => {  // do something  })  .catch(err => {  throw err  }) |
|  | Promise 的核心作用就是拉平 “洋葱结构”。不过错误处理以及 Promise 链这种用法当然也非常有意义的。但是，聪明的你一定发现了，好像用了 Promise 也没有让代码好看多少，好写多少啊？说得没错，所以才有接下来的 generator 和 async/await 啊。说到这里， Promise 最大的贡献其实是统一了 JavaScript 异步编程的标准规范。  Promise 并没有真正消除 callback ，Promise 只不过是用 then 方法来延迟了 callback 的绑定。  真正消灭 callback 的 generator 和 async/await；  .then()参数就是向前面的异步函数注册成功和失败的回调 | |
| Generator 函数，async函数 | 用了 Promise 其实并没有真正消灭 callback，并且还新增了很多 .then(data => { .... }) 这些很没有意义的 “模板代码”。所以爱折腾的人们终于搞出了generator 和 async/await，generator 有一些很神奇的特性这个就不多说了，自己看 MDN 上的文档就好，但是 generator 也能挺方便地处理异步（当然，要带上 tj/co 库才行），真正消灭了 callback 的存在。  // 使用 tj/co 库和 generator  co(function \*() {  try {  const aResult = yield new Promise(/\* getResultFromAPromise \*/)  const bResult = yield new Promise(/\* getResultFromBProimse \*/)  const cResult = yield new Promise(/\* getResultFromCPromise \*/)  // do something  } catch (err) {  throw err  }  })  看上去 generator 已经很好用了，但是用 generator 处理异步还是离不了 tj/co ，并且还有一些坑爹 bug 和兼容性问题。接下来再看 async/await  async function () {  try {  const aResult = await new Promise(/\* getResultFromAPromise \*/)  const bResult = await new Promise(/\* getResultFromBProimse \*/)  const cResult = await new Promise(/\* getResultFromCPromise \*/)  // do something  } catch (err) {  throw err  }  }  总体来说 async/await 看上去和使用 co 库后的 generator 看上去很相似，不过反正 callback 是没有了。代码结构又回到了我们所熟悉并且好看的 “序列结构” 了。 | |
|  |  | |
|  | 总结  对一门语言特性的深入理解，可以更好地帮助我们写出高效优雅的代码。平常工作中我们可以使用一些异步流程控制库如:async,then.js,bluebird,Q,co等，可以通过npm安装或直接下载，上手方便，简单易用，可以快速解决你遇到的异步流程问题。  回调函数Callbacks  异步JavaScript  Promise/a+规范  生成器Generators/ yield(es6)  Async/ await(es7)  async | |
| nodejs异步趣谈 | 作者：黄珏珅 链接：<https://www.zhihu.com/question/22098326/answer/22645677>  JavaScript设计上对异步没有很好的支持，对于Node这种完全异步的平台而言，编写复杂逻辑的应用程序简直就是灾难，举个例子： 通常，同步的逻辑是这样的： **function getForgeMetadata**(cid, itemId) {  **var** character = getCharacter(cid);  **var** item = getItem(itemId);  **var** itemDesc = getItemDesc(item.type);  **var** forgeLevel = getForgeLevel(item);  **return getForgeMetadata**(character, itemDesc, forgeLevel); } 然而，如果我们采用node.js中标准的回调格式来编写代码的话，您的代码会变成这样： **function getForgeMetadata**(cid, itemId, cb) {  getCharacter(cid, **function**(err, character) {  **if** (err) **return** cb(err);  getItem(itemId, **function**(err, item) {  **if** (err) **return** cb(err);  getItemDesc(item.type, **function**(err, itemDesc) {  **if** (err) **return** cb(err);  **var** forgeLevel = getForgeLevel(item);  **getForgeMetadata**(character, itemDesc, forgeLevel, cb);  });  });  }); } 看到那一长串的if (**err**) **return** cb(err);你会不会有种想要自杀的感觉，后来终于社区也看不下去了就弄了个async库出来，但基本上差别也不太大，再后来有了promise(通常使用库q来实现)，情况略有好转，代码可以变成这样： **function getForgeMetadata**(cid, itemId) {  **return** getCharacter(cid).then(**function** (character) {  **return** getItem(itemId).then(**function** (item) {  **return** {'character': character, 'item': item};  });  }).then(**function** (meta) {  **return** getItemDesc(meta.item.type).then(**function** (itemDesc) {  meta.itemDesc = itemDesc;  **return** meta;  });  }).then(**function** (meta) {  **var** forgeLevel = getForgeLevel(meta.item);  **return getForgeMetadata**(meta.character, meta.itemDesc, forgeLevel);  }); }  然而这坑爹货的代码还是一样的难读和难以维护啊，终于有一个人看不下去了，他就是 @赵颉，他把C#中的await语义移植到了js中，弄了个Wind.js出来，这样大家终于能写人话了： **function getForgeMetadata**(cid, itemId) {  **var** character = $await(getCharacter(cid));  **var** item = $await(getItem(itemId));  **var** itemDesc = $await(getItemDesc(item.type));  **var** forgeLevel = getForgeLevel(item);  **return** $await(**getForgeMetadata**(character, itemDesc, forgeLevel)); } 然而顽固不化的Node.js社区断然拒绝了这绝妙的设计，老赵也停止了Wind.js的开发，最终版本的Wind.js也没加入只差一步的调试功能，我们刚进到蜜罐就又被迫迁移到promise上来，继续痛苦的旅程。 | |

异步编程 观察者模式的简单实现

|  |  |
| --- | --- |
|  | <http://www.jb51.net/article/58832.htm> |
|  | 在jQuery中，on方法可以为元素绑定事件，trigger方法可以手动触发事件，围绕这2个方法，我们来体验jQuery中的观察者模式(Observer Pattern)。  on方法绑定内置事件，自然触发  比如，我们给页面的body元素绑定一个click事件，这样写。       $(function() {             $('body').on('click', function () {                 console.log('被点击了~~');             });         });  以上，我们只有点击body，才能触发click事件。也就是说，当给页面元素绑定内置事件后，事件的触发是在内置事件发生的那刻。  on方法绑定内置事件，手动触发  使用trigger方法，也可以手动触发元素绑定的内置事件。          $(function() {             $('body').on('click', function () {                 console.log('被点击了~~');             });             $('body').trigger('click');         });  以上，无需点击body，在页面加载完毕，body自动触发了click事件。  于是，我们发现：我们可以为元素绑定自定义事件，并且用trigger方法触发该事件。  当然，自定义事件的名称可以按照"命名空间.自定义事件名称"的形式来写，比如app.someclick，这在大型项目中尤其有用，这样可以有效避免自定义事件名称冲突。  如果从"发布订阅"这个角度来看，on方法相当于订阅者、观察者，trigger方法相当于发布者。 |
| 从异步ajax体验观察者模式 | **/\*  \* 从"异步获取json数据"来体验jQuery观察者模式  \* 在根目录下，有一个data.json的文件，  \* \*/** //通过异步的方式来获取json数据，可以获取数据 $.getJSON('data.json', **function** (data) {  console.log(data); });  //如果用一个全局变量来接收异步获取的json数据，得到的结果却是undefined，因为，当$.getJSON方法还在获取数据的时候，就已经执行console.log(data)，而此时data还没有数据； **var data**; $.getJSON('data.json', **function** (results) {  **data** = results; }); console.log(**data**); //如果在$.getJSON方法之外先定义好需要执行的方法，然后在$.getJSON方法的回调函数里真正触发这个方法，方可获得data； $.getJSON('data.json', **function** (results) {  $(document).trigger('app.myevent', results); //相当于发布 }); $(document).on('app.myevent', **function** (e, results) { //相当于订阅  console.log(results); }); //以上，on方法就像一个订阅者，它订阅了自定义事件app.myevent；而trigger方法就像一个发布者，它发布事件和参数后，才真正让订阅者方法得以执行。 |
| jQuery观察者模式的扩展方法 | //为此，我们还可以为jQuery观察者模式专门写一个扩展方法。 (**function**($) {  **var** o = $({});//自定义事件对象  $.each({  trigger: 'publish',  on: 'subscribe',  off: 'unsubscribe'  }, **function**(key, val) {  jQuery[val] = **function**() {  o[key].apply(o, arguments);  };  }); })(jQuery); //以上，定义了全局的publish和subscribe方法，我们在任何时候都可以调用。 $(**function** () {  **/\*定义\*/** $.subscribe('app.myEvent', **function**(e, results) {  console.log(results);  });  **/\*触发\*/** $.getJSON('data.json', **function** (results) {  $.publish('app.myEvent', results);  }); }); //总结：jQuery的观察者模式，实际上是让on方法绑定的自定义事件先不执行，直到使用trigger方法来触发事件。使用jQuery的观察者模式的好处是：一次发布，多次订阅。 |
|  |  |
|  |  |

jQuery的deferred对象详解

|  |  |  |
| --- | --- | --- |
|  | jQuery的deferred对象详解  <http://www.ruanyifeng.com/blog/2011/08/a_detailed_explanation_of_jquery_deferred_object.html>  分类：延迟对象  <http://www.jquery123.com/category/deferred-object/>  jQuery.Deferred对象  <http://javascript.ruanyifeng.com/jquery/deferred.html#toc3> | |
| api总结 | jQuery.Deferred() | 创建一个新的Deferred对象的构造函数，可以带一个可选的函数参数，它会在构造完成后被调用。 |
| jQuery.when() | 通过该方式来执行基于一个或多个表示异步任务的对象上的回调函数  为多个操作指定回调函数。除了这些方法以外，deferred对象还有二个重要方法，上面的教程中没有涉及到。 |
| jQuery.ajax() | 执行异步Ajax请求，返回实现了promise接口的jqXHR对象 |
| deferred.then(resolveCallback，rejectCallback) | 添加处理程序被调用时，递延对象得到解决或者拒绝的回调。  有时为了省事，可以把done()和fail()合在一起写，这就是then()方法。 $.when($.ajax( "/main.php" )).then(successFunc, failureFunc ); 如果then()有两个参数，那么第一个参数是done()方法的回调函数，第二个参数是fail()方法的回调方法。如果then()只有一个参数，那么等同于done()。 |
| deferred.done() | 当延迟成功时调用一个函数或者数组函数. |
| deferred.fail() | 当延迟失败时调用一个函数或者数组函数.。 |
| deferred.resolve(ARG1,ARG2,...) | 调用Deferred对象注册的'done’回调函数并传递参数  手动改变deferred对象的运行状态为"已完成",从而立即触发done()方法。 |
| deferred.resolveWith(context,args) | 调用Deferred对象注册的'done’回调函数并传递参数和设置回调上下文 |
| deferred.isResolved | 确定一个Deferred对象是否已经解决。 |
| deferred.reject(arg1,arg2,...) | 调用Deferred对象注册的'fail’回调函数并传递参数  deferred.reject() 这个方法与deferred.resolve()正好相反，调用后将deferred对象的运行状态变为"已失败"，从而立即触发fail()方法。 |
| deferred.rejectWith(context，args) | 调用Deferred对象注册的'fail’回调函数并传递参数和设置回调上下文 |
| deferred.promise() | 返回promise对象，这是一个伪造的deferred对象：它基于deferred并且不能改变状态所以可以被安全的传递  没有参数时，返回一个新的deferred对象，该对象的运行状态无法被改变；接受参数时，作用为在参数对象上部署deferred接口。 |
| deferred.always() | 这个方法也是用来指定回调函数的，它的作用是，不管调用的是deferred.resolve()还是deferred.reject()，最后总是执行。 $.ajax( "test.html" ).always( function() { log("已执行！");} ); |

ES6 Promise 和 jQuery Deferred 的异同

|  |  |
| --- | --- |
|  | 通过 ES6 Promise 和 jQuery Deferred 的异同学习 Promise  <https://segmentfault.com/a/1190000003691961>  参考  [ECMAScript 2015 Language Specification - ECMA-262 6th Edition](http://www.ecma-international.org/ecma-262/6.0/index.html" \t "https://segmentfault.com/a/_blank)  [Deferred - Mozilla | MDN](https://developer.mozilla.org/en-US/docs/Mozilla/JavaScript_code_modules/Promise.jsm/Deferred" \t "https://segmentfault.com/a/_blank)  [Promise - Mozilla | MDN](https://developer.mozilla.org/en-US/docs/Mozilla/JavaScript_code_modules/Promise.jsm/Promise" \t "https://segmentfault.com/a/_blank)  [Deferred Object | jQuery Documentation](http://api.jquery.com/category/deferred-object/" \t "https://segmentfault.com/a/_blank) |
|  | ES6 和 jQuery 都有 Deffered 和 Promise，但是略有不同。不过它们的作用可以简单的用两句话来描述  1.Deffered 触发 resolve 或 reject  2.Promise 中申明 resolve 或 reject 后应该做什么（回调）  在 jQuery 中  var deferred = $.Deferred();  var promise = deferred.promise();  在 ES6 中  var deferred = Promise.defer();  var promise= defered.promise;  MDN 宣布 Deferred 在 Gecko 30 中被申明为过期，不应该再使用，而应该用 new Promise() 来代替。关于 new Promise() 将在后面说明。 |
|  | jQuery 中最常用的 Promise 对象是 $.ajax() 返回的，最常用的方法不是 then，而是 done、fail 和 always。除了$.ajax() 外，jQuery 也提供了 $.get()、$.post() 和 $.getJSON() 等简化 Ajax 调用，它们返回的和 $.ajax() 的返回值一样，是个 Promise 对象。  实际上 $.ajax() 返回的是一个 [jqXHR 对象](http://api.jquery.com/jQuery.ajax/" \l "jqXHR" \t "https://segmentfault.com/a/_blank)。但 jqXHR 实现了 jQuery 的 Promise 接口，所以也是一个 Promise 对象。  done()、fail() 和 always()  done() 添加 deferred.resolve() 的回调，fail() 添加 deferred.reject() 的回调。所以在 Ajax 调用成功的情况下执行done() 添加的回调，调用失败时执行 fail() 添加的回调。但不管成功与否，都会执行 always() 添加的回调。  这里 done()、fail() 和 always() 都是以类似事件的方式添加回调，也就意味着，不管执行多次次 done()、fail() 或always()，它们添加的若干回调都会在符合的条件下依次执行。 |
| demo | //一般情况下会这样执行 Ajax: 调用 Ajax 提交数据，假设返回的是 JSON 数据 **var jqxhr** = $.ajax("do/example", {  type: "post",  dataType: "json",  data: "{}" }); **jqxhr**.done(**function** (data) {  // Ajax 调用成功  console.log("success with data", data); }).fail(**function** () {  // Ajax 调用失败  console.log("failed") }).always(**function** () {  // 不管成功与否，都会执行，取消按钮的禁用状态 });  **/\*  假设我们定义返回的 JSON 是这样的格式：  {  "code": "int, 0 表示成功，其它值表示出错",  "message": "string, 附加的消息，可选",  "data": "object，附加的数据，可选  }  则实现方式如下：  \*/** app.ajax = **function** (url, data) {  **return** $.ajax(url, {  type: "post",  dataType: "json",  data: data  }).done(**function** (json) {  **if** (json.code !== 0) {  showError(json.message || "操作发生错误");  }  }).fail(**function** () {  showError("服务器错误，请稍后再试");  }).always(**function** () {  }); }; app.ajax("do/example", "{}").done(**function** (json) {  **if** (json.code === 0) {  // 只需要处理正确的情况啦  } }); **/\*不过还是有点不爽，如果不需要判断 json.code === 0 就更好了。这个……可以自己用一个 Deferred 来处理：\*/** app.ajax = **function** (url, data) {  **var** deferred = $.Deferred();  $.ajax(url, {  type: "post",  dataType: "json",  data: data  }).done(**function** (json) {  **if** (json.code !== 0) {  showError(json.message || "操作发生错误");  deferred.reject();  } **else** {  deferred.resolve(json);  }  }).fail(**function** () {  showError("服务器错误，请稍后再试");  deferred.reject();  }).always(**function** () {  });  **return** deferred.promise();**/\*deferred.promise() 保证deferred状态不被外部改变\*/** }; app.ajax("do/example", "{}").done(**function** (json) {  // json.code === 0 总是成立  // 正常处理 json.data 就好 }); |
|  | jQuery deferred.then()  在 jQuery 1.8 以前（不含 1.8，比如 jQuery 1.7.2），deferred.then() 就是一个把 done() 和 fail() 放在一起的语法糖。jQuery 在 1.8 版本的时候修改了 deferred.then() 的行为，使 then() 的行为与 Promise 的 then() 相似。从 jQuery 的文档可以看到 1.8 版本的变化——干掉了 callback，换成了 filter：  // version added: 1.5, removed: 1.8  deferred.then( doneCallbacks, failCallbacks )  // version added: 1.7, removed: 1.8  deferred.then( doneCallbacks, failCallbacks [, progressCallbacks ] )  // version added: 1.8  deferred.then( doneFilter [, failFilter ] [, progressFilter ] )  可以简单的把 callback 当作一个事件处理，值用于 callback 之后一般不会改变；而 filter 不同，一个值传入 filter 再从 filter 返回出来，可能已经变了。还是举个例子来说明  var deferred = $.Deferred();  var promise = deferred.promise();  promise.then(function(v) {  console.log(`then with ${v}`);  }).done(function(v) {  console.log(`done with ${v}`);  });  deferred.resolve("resolveData");  在 jQuery 1.7.2 中的结果  then with resolveData  done with resolveData  在 jQuery 1.8.0 中的结果  then with resolveData  done with undefined  从上面来看，jQuery 的 deferred.then() 语义和 ES6 Promise.then() 语义基本一致。如果把上面的 app.ajax 换成 then() 实现会有助于对 ES6 Promise 的理解。  app.ajax = function(button, url, data) {  if (button) {  button.prop("disabled", true);  }  return $.ajax(url, {  type: "post",  dataType: "json",  data: data  }).then(function(json) {  if (json.code !== 0) {  showError(json.message || "操作发生错误");  return $.Deferred().reject().promise();  } else {  return $.Deferred().resolve(json).promise();  }  }, function() {  showError("服务器错误，请稍后再试");  deferred.reject();  }).always(function() {  if (button) {  button.prop("disabled", false);  }  });  };  // 调用方式没变，用 done，也可以用 then  app.ajax("do/example", getFormData()).done(function(json) {  // json.code === 0 总是成立  // 正常处理 json.data 就好  }); |
|  | **/\* 从 jQuery Promise 到 ES6 Promise \*/ var promise** = $.ajax(); **promise**.then(**function** (data) {  // resolve  **return** data.code == 0 ? **new** Promise().resolve(data) : **new** Promise().reject();  // 如果没有错，就返回一个新的 promise，并使用 data 来 resolve，  // 也可以直接返回 data，  // 这样后面 then 的 resolve 部分才能收到数据 }, **function** () {  // rejected }); // 调用阶段 **promise**.then(**function** (data) {  // 处理 data });  **/\*用 ES6 Promise 改写上面的示意代码\*/ var promise** = **new** Promise(**function** (resolve, reject) {  $.ajax().then(resolve, reject);  // 上面这句没看懂？那换成这样你一定会懂  // $.ajax().then(function(data) {  // resolve(data);  // }, function() {  // reject();  // }); }).then(**function** (data) {  **return** data.code == 0 ? Promise.resolve(data) : Promise.reject();  **/\* 这里 Promise.resolve(data) 同样可以直接替换为 data\*/** });  // 调用没变 **promise**.then(**function** (data) {  // 处理 data }); |
|  | 怎么样，差别不大吧。不知不觉就会 ES6 Promise 了,常用方法:  注意，小写的 promise 表示 Promise 对象  new Promise(executor)，产生一个新的 Promise 对象  > `executor(resolve, reject)`  > `executor`、`resolve` 和 `reject` 均为函数，在 `executor` 中，正确处理调用 `resolve()` 返回数据，异常处理直接 `throw new Error(...)` 或调 `reject()` 返回数据。  Promise.resolve(data)，产生 Promise 对象并 resolve  Promise.reject()，产生 Promise 对象并 reject  promise.then(onResolve, onReject)，然后……继续处理  promise.catch(onReject)，project.then(null, onReject) 的语法糖，和 jQuery 的 promise.fail() 差不多（但不同）。 |

deferred其他使用方式

|  |  |
| --- | --- |
| 使用deferred对象写JS动画非常方便 | 使用deferred对象写JS动画非常方便，例如：  // Animation flows. $.when( preloadImage() ) .then( animation01 ) .then( animation02 ) .then( animation03 ) .then( transition ) .then( merge ) .then( zoom ) .then( showContent ) .then( flicker );  优点： 1、可以任意调整动画的先后顺序。 2、添加 SKIP(跳过动画)功能也很方便。 3、调试动画也可以节省大量时间。 把不需要调试的动画项注释掉。 例如:  $.when( preloadImage() ) // .then( animation01 ) // .then( animation02 ) // .then( animation03 ) // .then( transition ) // .then( merge ) // .then( zoom ) .then( showContent ) .then( flicker ); |
| 模拟$.Deferred 的实现原理 | // 模拟$.Deferred 的实现原理 **function Deferred**() {  **this**.doneCallbacks = [];  **this**.failCallbacks = []; }  **Deferred**.prototype.done = **function** (cb) {  **this**.doneCallbacks.push(cb);  **return this**; };  **Deferred**.prototype.fail = **function** (cb) {  **this**.failCallbacks.push(cb);  **return this**; };  **Deferred**.prototype.resolve = **function** () {  **var** dcbs = **this**.doneCallbacks;  **for** (**var** i = 0, len = dcbs.length; i < len; i++) {  dcbs[i]();  }  **return this**; }  // 模拟$.when的实现 **function when**() {  **var** aIdx = 0, aLen = arguments.length,  dfd = **new Deferred**();   **function callback**() {  **alert**("已执行了[ " + ++aIdx + " ]个方法");  **if** (aIdx == aLen) {  dfd.resolve(); // 改变Deferred对象的执行状态  }  }   **for** (**var** i = 0; i < aLen; i++) {  arguments[i].done(**callback**);  }  **return** dfd; }  // 自定义方法 **var wait** = **function** (intval) {  **var** dfd = **new Deferred**(); //在函数内部，新建一个Deferred对象  **var tasks** = **function** () {  **alert**("耗时[ " + intval / 1000 + " ]秒的操作执行完毕！");  dfd.resolve(); // 改变Deferred对象的执行状态  };  setTimeout(**tasks**, intval);  **return** dfd; // 返回Deferred对象 };  // 调用测试 **when**(**wait**(5000), **wait**(10000)).done(**function** () {  **alert**('调用第一个done'); }).done(**function** () {  **alert**('调用第二个done'); }); |

Promise

|  |  |
| --- | --- |
| www | MDN Web technology for developers JavaScript JavaScript reference Standard built-in objects Promise  <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise>  JavaScript Promise迷你书（中文版）  <http://liubin.org/promises-book/>  Promise A+ 规范  <http://malcolmyu.github.io/malnote/2015/06/12/Promises-A-Plus/>  Bluebird-NodeJs的Promise  <http://www.2cto.com/kf/201507/425777.html>  Polyfill with Promise ECMAScript 2015(ES6) Object Library  <https://github.com/linkFly6/Promise>  [Promise/A+规范](https://promisesaplus.com/)  [Promise/A+规范（翻译）](http://segmentfault.com/a/1190000002452115)  <https://segmentfault.com/a/1190000002452115>  [JavaScript Promise启示录](http://segmentfault.com/a/1190000000492290)  <https://segmentfault.com/a/1190000000492290>  [[译]深入理解Promise五部曲：1. 异步问题](http://segmentfault.com/a/1190000000586666)  <https://segmentfault.com/a/1190000000586666>  Promise 对象  <http://es6.ruanyifeng.com/#docs/promise>  剖析Promise内部结构，一步一步实现一个完整的、能通过所有Test case的Promise类  <https://github.com/xieranmaya/blog/issues/3> |
|  | 1. ES6 Promise语言标准   promise是javascript针对异步操作场景所用。  使用同步的方式来写代码，执行的操作是异步的，保证程序的执行顺序是同步的  ---promise对象的三种状态:未完成(pending)、已完成(fulfilled)、失败(rejected)   1. Promise A+ 2. A+规范通过术语thenable来区分promise对象 3. A+定义inFulfilled/onRejected必须是作为函数来调用，而且调用过程必须是异步的 4. A+严格定义了then方法链式调用时onFulfilled/onRejected的调用顺序 5. Promise then   promiseObj.then(onFulfilled,onRejected)  onFulfilled = function(data){return promiseObj2} //返回值必须是一个promise对象  onRejected = function(err){}   1. promise 库   早在ES6的Promise之前，Q，when.js，bluebird等等库早就根据Promise标准（参考[Promise/A+](https://promisesaplus.com/?source=Blog_Email_[Promises for better ]" \t "http://web.jobbole.com/87662/_blank)）造出了自己的promise轮子。  bluebird Q then.js es6-promise ypromise async native-promise-only ...  Promise的作用是解决回调金字塔的问题，对于控制异步流程实际上没有起到很大的作用。真正使用Promise对异步流程进行控制，我们还要借助ES6 generator函数。（例如[Tj大神](https://github.com/tj" \t "http://web.jobbole.com/87662/_blank)的co库的实现)。  然而ES7将有一个更加牛逼的解决方案：async/await，这个方案类似于co,但是加了原生支持。拭目以待吧。 |
|  | Promise 的实现  [q](https://github.com/kriskowal/q" \t "http://www.jianshu.com/p/_blank)  <https://github.com/kriskowal/q>  [bluebird](https://github.com/petkaantonov/bluebird" \t "http://www.jianshu.com/p/_blank)  <https://github.com/petkaantonov/bluebird>  <http://bluebirdjs.com/docs/api-reference.html>  [co](https://github.com/tj/co" \t "http://www.jianshu.com/p/_blank)  <https://github.com/tj/co>  [when](https://github.com/cujojs/when" \t "http://www.jianshu.com/p/_blank)  <https://github.com/cujojs/when>  promises 拥有一个漫长并且戏剧化的历史，Javascript 社区花费了大量的时间让其走上正轨。在早期，deferred 在 Q，When，RSVP，Bluebird，Lie等等的 “优秀” 类库中被引入， jQuery 与 Angular 在使用 ES6 Promise 规范之前，都是使用这种模式编写代码。 |

API小结

|  |  |
| --- | --- |
| 前言 | <http://www.jianshu.com/p/063f7e490e9a>  ECMAScript 是 JavaScript 语言的国际标准，JavaScript 是 ECMAScript 的实现。ES6 的目标，是使得 JavaScript 语言可以用来编写大型的复杂的应用程序，成为企业级开发语言。  回调函数真正的问题在于他剥夺了我们使用 return 和 throw 这些关键字的能力。而 Promise 很好地解决了这一切。  ES6 原生提供了 Promise 对象。  所谓 Promise，就是一个对象，用来传递异步操作的消息。它代表了某个未来才会知道结果的事件（通常是一个异步操作），并且这个事件提供统一的 API，可供进一步处理。  Promise 对象有以下两个特点。  （1）对象的状态不受外界影响。Promise 对象代表一个异步操作，有三种状态：**Pending（进行中）、Resolved（已完成，又称 Fulfilled）和 Rejected（已失败）**。只有异步操作的结果，可以决定当前是哪一种状态，任何其他操作都无法改变这个状态。这也是 Promise 这个名字的由来，它的英语意思就是「承诺」，表示其他手段无法改变。  （2）一旦状态改变，就不会再变，任何时候都可以得到这个结果。Promise 对象的状态改变，只有两种可能：从 Pending 变为 Resolved 和从 Pending 变为 Rejected。只要这两种情况发生，状态就凝固了，不会再变了，会一直保持这个结果。就算改变已经发生了，你再对 Promise 对象添加回调函数，也会立即得到这个结果。这与事件（Event）完全不同，事件的特点是，如果你错过了它，再去监听，是得不到结果的。  有了 Promise 对象，就可以将异步操作以同步操作的流程表达出来，避免了层层嵌套的回调函数。此外，Promise 对象提供统一的接口，使得控制异步操作更加容易。  Promise 也有一些缺点。首先，无法取消 Promise，一旦新建它就会立即执行，无法中途取消。其次，如果不设置回调函数，Promise 内部抛出的错误，不会反应到外部。第三，当处于 Pending 状态时，无法得知目前进展到哪一个阶段（刚刚开始还是即将完成）。 |
|  | promise的基本概念。  promise只有三种状态，未完成，完成(fulfilled)和失败(rejected)。  promise的状态可以由未完成转换成完成，或者未完成转换成失败。  promise的状态转换只发生一次 |
|  | var promise = new Promise(function(resolve, reject) {  if ("success"){/\* 异步操作成功 \*/  resolve(value);  } else {  reject(error);  } }); promise.then(function(value) {  // success }, function(value) {  // failure });  Promise 构造函数接受一个函数作为参数，该函数的两个参数分别是 resolve 方法和 reject 方法。  如果异步操作成功，则用 resolve 方法将 Promise 对象的状态，从「未完成」变为「成功」（即从 pending 变为 resolved）；  如果异步操作失败，则用 reject 方法将 Promise 对象的状态，从「未完成」变为「失败」（即从 pending 变为 rejected）。  基本的 api：  Promise.resolve()  Promise.reject()  Promise.prototype.then()  Promise.prototype.catch()  Promise.all() // 所有的完成 var p = Promise.all([p1,p2,p3]);  Promise.race() // 竞速，完成一个即可  Promise 构造函数拥有的方法：resolve reject all race  promise 实例对象拥有的方法：then catch |
|  |  |
| new Promise() | 使用new来调用Promise的构造器进行实例化  var p = new Promise(function (resolve,reject) {  //异步处理  //处理结束后，调用resolve或reject  console.log(arguments);  //resolve表示成功状态下执行  //reject表示失败状态下执行 });  ===  Promise是一个对象，它的**构造函数接收一个回调函数，这个回调函数参数有两个函数：分别在成功状态下执行和失败状态下执行，Promise有三个状态，分别为：等待态（Pending）、执行态（Fulfilled）和拒绝态（Rejected）**。  传递的这个回调函数，等同被Promise重新封装，并传递了两个参数回调，这两个参数用于驱动Promise数据的传递。resolve和reject本身承载着触发器的使命：  默认的Promise对象是等待态（Pending）。  调用resolve()表示这个Promise进入执行态（Fulfilled）  调用reject()表示这个promise()进入拒绝态（Rejected）  Promise对象可以从等待状态下进入到执行态和拒绝态，并且无法回退。  而执行态和拒绝态不允许互相转换（例如执行态转换到拒绝态）。  ===  对通过new生成的promise对象,为了设置其在resolve(成功)/reject(失败)时调用的回调函数可以使用promise.then()实例方法:  promise.then(onFulfilled, onRejected);//以防误解，上述的即为以下这种形式  promise.then(function(){...}, function(){...});  当resolve(成功)时，会调用onFulfilled函数；reject(失败)时，会调用onRejected函数。  ===  若只想处理异常情况的函数，可promise.then(undefined, onRejected)，当然更好的选择是用promise.catch()来处理。二者效果相同。  promise.then(function (value) {  console.log(value);  }).catch(function (error) {  console.log(error);  });  //等同于以下形式  promise.then(function (value) {  console.log(value);  }, function (error) {  console.log(error);  }); |
| 示例： | **function getURL**(URL) {  **return new** Promise(**function** (resolve, reject) {  **var** req = **new** XMLHttpRequest();  req.open('GET', URL, **true**);  req.onload = **function** () {  **if** (req.status === 200) {  resolve(req.responseText);  } **else** {  reject(**new** Error(req.statusText));  }  };  req.onerror = **function** () {  reject(**new** Error(req.statusText));  };  req.send();  }); } // 运行示例 **var URL** = "http://httpbin.org/get"; **getURL**(**URL**).then(**function** (value) {  console.log(value); }).catch(**function** (error) {  console.error(error); });  getURL 只有在通过XHR取得结果状态为200时才会调用 resolve - 也就是只有数据取得成功时，而其他情况（取得失败）时则会调用 reject 方法。  resolve(req.responseText) 在response的内容中加入了参数。 resolve方法的参数并没有特别的规则，基本上把要传给回调函数参数放进去就可以了。 ( then 方法可以接收到这个参数值)  XHR中 onerror 事件被触发的时候就是发生错误时，所以理所当然调用reject。 这里我们重点来看一下传给reject的值。  发生错误时要像这样 reject(new Error(req.statusText)); ，创建一个Error对象后再将具体的值传进去。 传给reject 的参数也没有什么特殊的限制，一般只要是Error对象（或者继承自Error对象）就可以。  传给reject 的参数，其中一般是包含了reject原因的Error对象。 本次因为状态值不等于200而被reject，所以reject 中放入的是statusText。 （这个参数的值可以被 then 方法的第二个参数或者 catch 方法中使用）  **var URL** = "http://httpbin.org/status/500"; getURL(**URL**).then(**function** (value) {  console.log(value); }).catch(**function** (error) {  console.error(error); }); //其实 .catch只是 promise.then(undefined, onRejected) 的别名而已， 如下代码也可以完成同样的功能。 getURL(**URL**).then(onFulfilled, onRejected); //一般说来，使用.catch来将resolve和reject处理分开来写是比较推荐的做法， 这两者的区别会在then和catch的区别中再做详细介绍。 |
| demo | console.log(0); **new** Promise(**function** (resolve, reject) {  console.log(1.1);  reject(**true**);  console.log(1.2);  //xx.xx;//报错会导致下面不执行  setTimeout(**function** () {  console.log(1.3);  resolve(**false**);  console.log(1.4);  }, 0);  console.log(1.5); }).then(  **function** () {  console.log(3.1);  },  **function** () {  console.log(3.2);  } ); console.log(2);  **/\*  0  1.1  1.2  1.5  2  3.2  1.3  1.4  \*/** |
| demo | // 演示加载一张图片  var imgLoad = function (src, success, error) {  var img = new Image();  img.onload = function () {  success('done');  }  img.onabort = img.onerror = function () {  error('fail');  };  img.src = src;  };  //调用  imgLoad('foo.jpg', function (str) {  console.log(str, '成功');//=> 'done 成功'  }, function (str) {  console.log(str, '失败');//=> 'fail 失败'  });  // Promise实现  var imgLoad = function (src) {  //返回Promise  return new Promise(function (resolve, reject) {  var img = new Image();  img.onload = function () {  resolve('done');  }  img.onabort = img.onerror = function () {  reject('fail');  };  img.src = src;  });  };  //调用  imgLoad('foo.jpg')  //异步逻辑抹平  .then(function (str) {  console.log(str, '成功');//=> 'done 成功'  }, function (str) {  console.log(str, '失败');//=> 'fail 失败'  });  或者利用它可以把异步的代码封装的更加扁平化，我们应该清楚Promise的使命，抹平了异步代码的回调金字塔，我们会有很多依赖上一层异步的代码：  **var url** = 'http://www.cnblogs.com/silin6/'; ajax(**url**, **function** (data) {  ajax(**url** + data, **function** (data2) {  ajax(**url** + data2, **function** (data3) {  ajax(**url** + data3, **function** () {  //回调金字塔  });  });  }); }); //使用Promise则抹平了代码： promise.then(**function** (data) {  **return** ajax(**url** + data); }).then(**function** (data2) {  **return** ajax(**url** + data2); }).then(**function** (data3) {  **return** ajax(**url** + data3); }).then(**function** (data) {  //扁平化代码 });  传统的异步代码和Promise代码对比：  [IMG_256](https://camo.githubusercontent.com/cb6627f16c3fe1f7414739f6cdf650ddba0b2194/687474703a2f2f696d616765732e636e626c6f67732e636f6d2f636e626c6f67735f636f6d2f73696c696e362f3539363832302f6f5f50726f6d697365416e6443616c6c6261636b2e706e67" \t "https://github.com/linkFly6/_blank) |
| demo-与回调的对比 | Promise的解决  Promise优雅的修正了异步代码，我们使用Promise重写我们setTimeout的示例：  var name,  p = new Promise(function (resolve) {  setTimeout(function () {//异步回调  resolve();  }, 1000);//1s后执行  });  p.then(function () {  name = 'linkFly';  console.log(name);//linkFly  }).then(function () {  name = 'cnBlog';  console.log(name);  });  //这段代码1s后会输出linkFly,cbBlog  我们先不要太过在意Promise对象的API，后续会讲解，我们只需要知道这段代码完成了和之前同样的工作。我们的console.log(name)正确的输出了linkFly，并且我们还神奇的输出了cnBlog。  或许你觉得这段代码实在繁琐，还不如setTimeout来的痛快，那么我们再来改写上面的ajax：  var ajax = function (url) {  //我们改写ajax，让它以Promise的方式工作  return new Promise(function (resolve) {  $.ajax({  url: url,  success: function (data) {  resolve(data);  }  });  });  };  ajax('http://www.cnblogs.com/silin6/map')  .then(function (key) {  //我们得到key，发起第二条请求  return ajax('http://www.cnblogs.com/silin6/source/' + key);  })  .then(function (data) {  console.log(data);//这时候我们会接收到第二次ajax返回的数据  });  或许它晦涩难懂，那么我们尝试用setTimeout来模拟这次的ajax，这个例子演示了Promise数据的传递，一如ajax：  var name,  ajax = function (data) {  return new Promise(function (resolve) {  setTimeout(function () {//我们使用setTimeout模拟ajax  resolve(data);  }, 1000);//1s后执行  });  };  ajax('linkFly').then(function (name) {  return ajax("i'm " + name);//模拟第二次ajax  }).then(function (value) {  //2s后，输出i'm linkFly  console.log(value);  });  上面的代码，从代码语义上达到了下面的流程：  IMG_256  我们仅观察代码就知道现在的它变得非常优雅，两次异步的代码被完美的抹平。但我们应该时刻谨记，Promise改变的是你异步的代码和编程思想，而并没有改变异步代码的执行——它是一种由卓越的编程思想所衍生的对象。 |
| demo-简单使用 | new Promise(function (resolve, reject) {  //异步操作  setTimeout(function () {  resolve('foo');//1s后标识成功  }, 1000);  }).then(function (data) {  //Resolved  console.log(data);//1s后输出foo  return 'bar';//传递结果到下一个链  }, function () {  //Rejected  }).then(function (data) {  console.log(data);//1s输出foo后，输出bar  }); |
|  |  |
|  | 一般情况下我们都会使用 new Promise() 来创建promise对象，但是除此之外我们也可以使用其他方法。  在这里，我们将会学习如何使用 [Promise.resolve](http://liubin.org/promises-book/" \l "Promise.resolve) 和 [Promise.reject](http://liubin.org/promises-book/" \l "Promise.reject)这两个方法。 |
| Promise.resolve | Promise.resolve(value)可认为是new Promise()方法的快捷方式  Promise.resolve(value);  //等同于以下代码  new Promise(function(resolve){  resolve(value);  });  此时这个promise对象会进入fulfilled状态。而resolve(value)中的value会传递给后面then中指定的onFulfilled函数。  Promise.resolve(value)返回值也是一个promise对象，所以可以进行链式调用. |
| Promise.reject | Promise.reject(error)与上述静态方法类似，也是new Promise()方法的快捷方式  Promise.reject(new Error("出错了"));  //等同于以下代码  new Promise(function(resolve, reject){  reject(new Error("出错了"));  });  这段代码则是调用该promise对象通过then指定的onRejected函数，并将错误对象(Error)传递给onRejected函数。 |
| Thenable对象转化 | Promise.resolve 方法另一个作用就是将 [thenable](http://liubin.org/promises-book/" \l "Thenable) 对象转换为promise对象。  [ES6 Promises](http://liubin.org/promises-book/" \l "es6-promises)里提到了[Thenable](http://liubin.org/promises-book/" \l "Thenable)这个概念，简单来说它就是一个非常类似promise的东西。  就像我们有时称具有 .length 方法的非数组对象为Array like一样，thenable指的是一个具有 .then 方法的对象。  这种将thenable对象转换为promise对象的机制要求thenable对象所拥有的 then 方法应该和Promise所拥有的 then 方法具有同样的功能和处理过程，在将thenable对象转换为promise对象的时候，还会巧妙的利用thenable对象原来具有的 then 方法。  到底什么样的对象能算是thenable的呢，最简单的例子就是 [jQuery.ajax()](https://api.jquery.com/jQuery.ajax/)，它的返回值就是thenable的。  因为jQuery.ajax() 的返回值是 [jqXHR Object](http://api.jquery.com/jQuery.ajax/" \l "jqXHR) 对象，这个对象具有 .then 方法。  $.ajax('/json/comment.json');// => 拥有 `.then` 方法的对象  这个thenable的对象可以使用 Promise.resolve 来转换为一个promise对象。  变成了promise对象的话，就能直接使用 then 或者 catch 等这些在 [ES6 Promises](http://liubin.org/promises-book/" \l "es6-promises)里定义的方法了。  将thenable对象转换promise对象  var promise = Promise.resolve($.ajax('/json/comment.json'));// => promise对象  promise.then(function(value){  console.log(value);  });   |  | | --- | | jQuery和thenable  [jQuery.ajax()](https://api.jquery.com/jQuery.ajax/)的返回值是一个具有 .then 方法的 [jqXHR Object](http://api.jquery.com/jQuery.ajax/" \l "jqXHR)对象，这个对象继承了来自 [Deferred Object](http://api.jquery.com/category/deferred-object/) 的方法和属性。  但是Deferred Object并没有遵循[Promises/A+](http://liubin.org/promises-book/" \l "promises-aplus)或[ES6 Promises](http://liubin.org/promises-book/" \l "es6-promises)标准，所以即使看上去这个对象转换成了一个promise对象，但是会出现缺失部分信息的问题。  这个问题的根源在于jQuery的 [Deferred Object](http://api.jquery.com/category/deferred-object/) 的 then 方法机制与promise不同。  所以我们应该注意，即使一个对象具有 .then 方法，也不一定就能作为ES6 Promises对象使用。  [JavaScript Promises: There and back again - HTML5 Rocks](http://www.html5rocks.com/en/tutorials/es6/promises/" \l "toc-lib-compatibility)  [You're Missing the Point of Promises](http://domenic.me/2012/10/14/youre-missing-the-point-of-promises/)  <https://twitter.com/hirano_y_aa/status/398851806383452160> |   Promise.resolve 只使用了共通的方法 then ，提供了在不同的类库之间进行promise对象互相转换的功能。  这种转换为thenable的功能在之前是通过使用 Promise.cast 来完成的，从它的名字我们也不难想象它的功能是什么。  除了在编写使用Promise的类库等软件时需要对Thenable有所了解之外，通常作为end-user使用的时候，我们可能不会用到此 |
| .then() | promises 的奇妙在于给予我们以前的 return 与 throw，每个 Promise 都会提供一个 then() 函数，和一个 catch()，实际上是 then(null, ...) 的语法糖)。 |
| then可以接受3个函数参数 | 生成的promise实例（如上面的变量p）拥有方法then()，then()方法是Promise对象的核心，它返回一个新的Promise对象，因此可以像jQuery一样链式操作，非常优雅。  then方法可以接受3个函数作为参数。前两个函数对应promise的两种状态fulfilled, rejected的回调函数。第三个函数用于处理进度信息。  promiseSomething().then(function (fulfilled) {  //当promise状态变成fulfilled时，调用此函数  }, function (rejected) {  //当promise状态变成rejected时，调用此函数  }, function (progress) {  //当返回进度信息时，调用此函数  }); |
|  |  |
| 函数参数内部中可以完成 | somePromise().then(**function** () {  });  可以在then内部完成：  1.return 另一个 promise  2.return 一个同步的值 (或者 undefined)  3.throw 一个同步异常  就是这样。一旦你理解了这个技巧，你就理解了 promises。因此让我们逐个了解下。  返回另一个 promise  这是一个在 promise 文档中常见的使用模式，也就是我们在上文中提到的 "composing promises"：  getUserInfoById("userId").then(**function** (user) {  **return** getUserFamilyByFamilyId(user.familyId) }).then(**function** (userFamily) {  //get family info  });  注意到我是 return 第二个 promise，这个 return 非常重要。如果我没有写return，getUserAccountById() 就会成为一个副作用，并且下一个函数将会接收到undefined 而非 userFamily。  p.then(**function** () {  //我们返回一个promise  **return new** Promise(**function** (resolve) {  setTimeout(**function** () {  resolve('resolve');  }, 1000);//异步1s  }); }, **function** () {  console.log('rejected'); }).then(**function** (state) {  console.log(state);//如果为执行态，输出resolve }, **function** (data) {  console.log(data);//如果为拒绝态，输出undefined });  返回一个同步值 (或者 undefined)  返回 undefined 通常是错误的，但是返回一个同步值实际上是将同步代码包裹为 promise 风格代码的一种非常赞的手段。举例来说，我们对 users 信息有一个内存缓存。我们可以这样做：  getUserByName("nameId").then(function (user) {  if (inMemoryCache[user.id]) {  return inMemoryCache[user.id];//返回一个同步值  }  return getUserAccountById(user.id);//返回一个promise  }).then(function (userAccount) {  // I got a user account !  });  是不是很赞？第二个函数不需要关心 userAccount 是从同步方法还是异步方法中获取的，并且第一个函数可以非常自由的返回一个同步或者异步值。  不幸的是，有一个不便的现实是在 JavaScript 中无返回值函数在技术上是返回 undefined，这就意味着当你本意是返回某些值时，你很容易会不经意间引入副作用。  出于这个原因，我个人养成了在 then() 函数内部 永远返回或抛出 的习惯。我建议你也这样做。  var p1 = Promise.resolve("a");  p1.then(function(msg){  console.log(msg);//a  }).then(function(msg){  console.log(msg);//undefined  })  抛出同步异常  谈到 throw，这是让 promises 更加赞的一点。比如我们希望在用户已经登出时，抛出一个同步异常。这会非常简单：  getUserByName("userName").then(**function** (user) {  **if** (user.isLoggedOut) {  **throw new** Error("user logged out!");//throwing a synchronous error!  }  **if** (getMemoryCache(user.id)) {  **return** getMemoryCache(user.id);//returning a synchronous value!  }  **return** getUserFamilyByUserFamilyId(user.familyId);//returning a promise }).then(**function** (userFamily) {  // got a user's family info }).catch(**function** (err) {  // got an error });  如果用户已经登出，我们的 catch() 会接收到一个同步异常，并且如果 后续的 promise 中出现异步异常，他也会接收到。再强调一次，这个函数并不需要关心这个异常是同步还是异步返回的。  这种特性非常有用，因此它能够在开发过程中帮助定位代码问题。举例来说，如果在then() 函数内部中的任何地方，我们执行JSON.parse()，如果 JSON 格式是错误的，那么它就会抛出一个异常。如果是使用回调风格，这个错误很可能就会被吃掉，但是使用 promises，我们可以轻易的在 catch() 函数中处理它了。 |
| 状态的继承关系 | // then()方法的返回值由它相应状态下执行的函数决定：  // then()方法传递的进入的回调函数，如果返回promise对象，则then()方法返回这个promise对象； // 否则将默认构建一个**新的promise对象**，并继承调用then()方法的promise的状态。  // 1 如果对应状态所执行的函数返回一个全新的Promise对象，则会覆盖掉当前Promise; **var p** = **new** Promise(**function** (resolve) {  resolve();//直接标志执行态 }); **var temp** = **p**.then(**function** () {  //返回新的promise对象，和p的状态无关  **return new** Promise(**function** (resolve, reject) {  reject();//标志拒绝态  }); }); **temp**.then(**function** () {  console.log('fulfilled'); }, **function** () {  console.log('rejected');//输出 });  // 2 这个函数返回undefined，则then()方法构建一个默认的Promise对象，并且这个对象拥有then()方法所属的Promise对象的状态。 **var p** = **new** Promise(**function** (resolve) {  resolve();//直接标志执行态 }); **var temp** = **p**.then(**function** () {  //传入执行态函数，不返回值 }); **temp**.then(**function** () {  console.log('fulfilled');//拥有p的状态 }); console.log(**temp** === **p**);//默认构建的promise，但已经和p不是同一个对象，输出false |
| all | var p1 = Promise.resolve("a");  var p2 = Promise.resolve("b");  var p3 = Promise.resolve("c");  Promise  .all([p1, p2])  .then(function (msgs) {  console.log(msgs);//["a","b"]  return p3;  })  .then(function (msg) {  console.log(msg);//"c"  });  Promise.all([ .. ])期待一个单独的参数，一个array，一般由Promise的实例组成。从Promise.all([ .. ])返回的promise将会收到完成的消息（在这段代码中是msgs），它是一个由所有被传入的promise的完成消息按照被传入的顺序构成的array（与完成的顺序无关）。 |
| API | Promise.prototype.catch()：用于指定发生错误时的回调函数（捕获异常），并具有冒泡性质。  Promise.all()，Promise.race()：Promise.all方法用于将多个Promise实例，包装成一个新的Promise实例。  Promise.resolve()，Promise.reject():将现有对象转为Promise对象。 |
|  | **/\*1. 封装同步与异步代码\*/**  **new Promise**(**function** (resolve, reject) {  resolve(someValue); }); //---写成 **Promise**.resolve(someValue);  **/\*2. 捕获同步异常\*/ new Promise**(**function** (resolve, reject) {  **throw new** Error('悲剧了，又出 bug 了'); }).catch(**function** (err) {  console.log(err);  }); //如果是同步代码，可以写成 **Promise**.reject(**new** Error("什么鬼"));  **/\*3. 多个异常捕获，更加精准的捕获\*/** somePromise.then(**function** () {  **return a**.b.c.d(); }).catch(TypeError, **function** (e) {  //If a is defined, will end up here because  //it is a type error to reference property of undefined }).catch(ReferenceError, **function** (e) {  //Will end up here if a wasn't defined at all }).catch(**function** (e) {  //Generic catch-the rest, error wasn't TypeError nor  //ReferenceError });  **/\*4. 获取两个 Promise 的返回值\*/** //1..then 方式顺序调用 2. 设定更高层的作用域 3. spread  **/\*5. finally\*/** //任何情况下都会执行的 ， 一般写在 catch 之后  **/\*6. bind : how can the second then get promise value\*/** somethingAsync().bind({})  .spread(**function** (aValue, bValue) {  **this**.aValue = aValue;  **this**.bValue = bValue;  **return** somethingElseAsync(aValue, bValue);  })  .then(**function** (cValue) {  **return this**.aValue + **this**.bValue + cValue;  }); //或者 你也可以这样 **var scope** = {}; somethingAsync()  .spread(**function** (aValue, bValue) {  **scope**.aValue = aValue;  **scope**.bValue = bValue;  **return** somethingElseAsync(aValue, bValue);  })  .then(**function** (cValue) {  **return scope**.aValue + **scope**.bValue + cValue;  }); //然而，这有非常多的区别，你必须先声明，有浪费资源和内存泄露的风险；  //不能用于放在一个表达式的上下文中，效率更低   **// 7. all 。 非常用于于处理一个动态大小均匀的 Promise 列表**  **// 8. join 。 非常适用于处理多个分离的 Promise** **var join** = **Promise**.join; **join**(getPictures(), getComments(), getTweets(), **function** (pictures, comments, tweets) {  console.log("in total: " + pictures.length + comments.length + tweets.length); });  **// 9. props 。 处理一个 promise 的 map 集合 。 只有有一个失败 ， 所有的执行都结束** **Promise**.props({  pictures: getPictures(),  comments: getComments(),  tweets: getTweets() }).then(**function** (result) {  console.log(result.tweets, result.pictures, result.comments); });  **// 10. any 、 some 、 race Promise**.some([  ping("ns1.example.com"),  ping("ns2.example.com"),  ping("ns3.example.com"),  ping("ns4.example.com") ], 2).spread(**function** (first, second) {  console.log(first, second); }).catch(AggregateError, **function** (err) {  err.forEach(**function** (e) {  console.error(e.stack);  }); }); //有可能 ， 失败的 promise 比较多 ， 导致 ， Promsie 永远不会 fulfilled  **//11. .map(Function mapper [, Object options ]) 用于处理一个数组 ， 或者 promise 数组 ，** // Option: concurrency 并发现 // map(..., {concurrency: 1} ) ; // 以下为不限制并发数量 ， 读书文件信息  **var Promise** = require("bluebird"); **var join** = **Promise**.join; **var fs** = **Promise**.promisifyAll(require("fs")); **var concurrency** = parseFloat(process.argv[2] || "Infinity");  **var fileNames** = ["file1.json", "file2.json"]; **Promise**.map(**fileNames**, **function** (fileName) {  **return fs**.readFileAsync(fileName)  .then(JSON.parse)  .catch(SyntaxError, **function** (e) {  e.fileName = fileName;  **throw** e;  }) }, {concurrency: **concurrency**}).then(**function** (parsedJSONs) {  console.log(parsedJSONs); }).catch(SyntaxError, **function** (e) {  console.log("Invalid JSON in file " + e.fileName + ": " + e.message); }); **/\*  结果  $ sync && echo 3 > /proc/sys/vm/drop\_caches  $ node test.js 1  reading files 35ms  $ sync && echo 3 > /proc/sys/vm/drop\_caches  $ node test.js Infinity  reading files: 9ms  \*/  //11. .reduce(Function reducer [, dynamic initialValue]) -> Promise** **Promise**.reduce(["file1.txt", "file2.txt", "file3.txt"], **function** (total, fileName) {  **return fs**.readFileAsync(fileName, "utf8").then(**function** (contents) {  **return** total + parseInt(contents, 10);  }); }, 0).then(**function** (total) {  //Total is 30 });  //12. Time // .delay(int ms) -> Promise // .timeout(int ms [, String message]) -> Promise |
|  |  |
|  | JavaScript Promise 探微  <http://malcolmyu.github.io/malnote/2014/08/30/JavaScript-Promise-In-Wicked-Detail/>  Promise A+ 规范  <http://malcolmyu.github.io/malnote/2015/06/12/Promises-A-Plus/> |
|  |  |
|  |  |

promise注意点

|  |  |
| --- | --- |
|  | <http://bubufx.com/detail-1778122.html> |
| Promise.resolve() | 不知道Promise.resolve()  promises 在封装同步与异步代码时非常的有用。然而，如果你发现你经常写出下面的代码：  new Promise(function (resolve, reject) {  resolve(someSynchronousValue) }).then()  使用Promise.resolve将会更简洁：  Promise.resolve(someSynchrounousVlaue).then()  它在用来捕获同步异常时也极其的好用。由于它实在是好用，因此我已经养成了在我所有 promise 形式的 API 接口中这样使用它：  **function somePromiseAPI** () {  **return** Promise.resolve().then(**function** () {  doSomethingThatMyThrow();  **return** "foo";  }).then(**/\*---\*/**) }  切记：任何有可能 throw 同步异常的代码都是一个后续会导致几乎无法调试异常的潜在因素。但是如果你将所有代码都使用Promise.resolve() 封装，那么你总是可以在之后使用catch() 来捕获它。  类似的，还有 Promise.reject() 你可以用来返回一个立刻返回失败的 promise。  Promise.reject(new Error("some awful error")); |
| then or catch | 使用promise.then(onFulfilled, onRejected) 的话，在 onFulfilled 中发生异常的话，在 onRejected 中是捕获不到这个异常的。  在 promise.then(onFulfilled).catch(onRejected) 的情况下，then 中产生的异常能在 .catch 中捕获[.then](http://liubin.org/promises-book/" \l "promise.then) 和 [.catch](http://liubin.org/promises-book/" \l "promise.catch) 在本质上是没有区别的，需要分场合使用。 |
| catch() then(null, ...)并非完全等价 | //resolve中异常在then2 catch中只执行一次 错误不会穿透  var ll=console.log;  var promise = new Promise(function (resolve, reject) {  resolve(value); //error  reject(error);  });    promise.then(function (value) {  ll("then 1");  ll(value);  }, function (value) {  ll("then 2");  }).catch (function () {  ll("catch 3");  });  promise.then(function (value) {  ll("then 4");  ll(value);  }).catch (function () {  ll("catch 5");  });  promise.then(function (value) {  ll("then 6");  ll(value);  }, function (value) {  ll("then 7");  }).then(function () {  ll("then 8")  }, function () {  ll("then 9")  })  // then 2  // then 7  // catch 5  // then 8  // 先执行7 因为先执行第一阶梯线 |
|  | var p = new Promise(function (resolve, reject) {  xx; // 一个错误  resolve(42); // 永远不会跑到这里 :(  });  p.then(function fulfilled() {  // 永远不会跑到这里 :(  }, function rejected(err) {  ll(1)  xx  }).catch (function () {  ll(22)  }); |
| catch 与 then | 之前我说过 catch() 仅仅是一个语法糖。因此下面两段代码是等价的：  somePromise().catch(function (err) {  // handle error  });  somePromise().then(null, function (err) {  // handle error  });  然而，这并不意味着下面两段代码是等价的：  somePromise().then(function () {  return someOtherPromise();  }).catch(function (err) {  // handle error  });  somePromise().then(function () {  return someOtherPromise();  }, function (err) {  // handle error  });  如果你好奇为何这两段代码并不等价，可以考虑一下如果第一个函数抛出异常会发生什么：  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! :(  });  因此，当你使用 then(resolveHandler, rejectHandler) 这种形式时，**rejectHandler 并不会捕获由 resolveHandler 引发的异常。**  鉴于此，我个人的习惯是不适用 then() 的第二个参数，而是总是使用 catch()。唯一的例外是当我写一些异步的 [Mocha](http://mochajs.org/" \t "http://bubufx.com/_blank) 测试用例时，我可能会希望用例的异常可以正确的被抛出：  it('should throw an error', function () {  return doSomethingThatThrows().then(function () {  throw new Error('I expected an error!');  }, function (err) {  should.exist(err);  });});  说到这里，[Mocha](http://mochajs.org/" \t "http://bubufx.com/_blank) 和 [Chai](http://chaijs.com/" \t "http://bubufx.com/_blank) 用来测试 promise 接口时，是一对非常好的组合。[pouchdb-plugin-seed](https://github.com/pouchdb/plugin-seed" \t "http://bubufx.com/_blank) 项目中有一些 [示例](https://github.com/pouchdb/plugin-seed/blob/master/test/test.js" \t "http://bubufx.com/_blank) 可以帮助你入门。 |
| promises vs promises factories | 当我们希望执行一个个的执行一个 promises 序列，即类似 Promise.all() 但是并非并行的执行所有 promises。  你可能天真的写下这样的代码：  function executeSequentially(promises) {  var result = Promise.resolve();  promises.forEach(function (promise) {  result = result.then(promise);  });  return result;  }  不幸的是，这份代码不会按照你的期望去执行，你传入 executeSequentially() 的 promises 依然会并行执行。  其根源在于你所希望的，实际上根本不是去执行一个 promises 序列。依照 promises 规范，一旦一个 promise 被创建，它就被执行了。因此你实际上需要的是一个 promise factories 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：“这是哪个见鬼的 Java 程序猿，他为啥在说 factories？” 。实际上，一个 promises factory 是十分简单的，它仅仅是一个可以返回 promise 的函数：  function myPromiseFactory() {  return somethingThatCreatesAPromise();  }  为何这样就可以了？这是因为一个 promise factory 在被执行之前并不会创建 promise。它就像一个 then 函数一样，而实际上，它们就是完全一样的东西。  如果你查看上面的 executeSequentially() 函数，然后想象 myPromiseFactory 被包裹在 result.then(...) 之中，也许你脑中的小灯泡就会亮起。在此时此刻，对于 promise 你就算是悟道了。 |
|  | 如果我希望获得两个 promises 的结果怎么办  有时候，一个 promise 会依赖于另一个，但是如果我们希望同时获得这两个 promises 的输出。举例来说：  getUserByName('nolan').then(function (user) {  return getUserAccountById(user.id);  }).then(function (userAccount) {  // dangit, I need the "user" object too!  });  为了成为一个优秀的 Javascript 开发者，并且避免金字塔问题，我们可能会将 user 对象存在一个更高的作用域中的变量里：  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"  });  这样是没问题的，但是我个人认为这样做有些杂牌。我推荐的策略是抛弃成见，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"  });  });  …至少暂时这样是没问题的。一旦缩进开始成为问题，你可以通过 Javascript 开发者从远古时期就开始使用的技巧，将函数抽离到一个命名函数中：  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  });  由于你的 promise 代码开始变得更加复杂，你可能发现自己开始将越来越多的函数抽离到命名函数中，我发现这样做，你的代码会越来越漂亮，就像这样：  putYourRightFootIn()  .then(putYourRightFootOut)  .then(putYourRightFootIn)  .then(shakeItAllAbout);  这就是 promises 的重点。 |
| promises 穿透 | 最后，这个错误就是我开头说的 promises 谜题所影射的错误。这是一个非常稀有的用例，并且可能完全不会出现在你的代码中，但是的的确确震惊了我。  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);//foo  });  如果你认为它会打印出 bar，那么你就错了。它实际上打印出来的是 foo！  发生这个的原因是如果你像 then() 传递的并非是一个函数（比如 promise），它实际上会将其解释为 then(null)，这就会导致前一个 promise 的结果会穿透下面。你可以自己测试一下：  Promise.resolve('foo').then(null).then(function (result) {  console.log(result);// foo  });  添加任意数量的 then(null)，它依然会打印 foo。  这实际上又回到了我之前说的 promises vs promise factories。简单的说，你可以直接传递一个 promise 到 then() 函数中，但是它并不会按照你期望的去执行。  then() 是期望获取一个函数，因此你希望做的最可能第三种使用方式：  var p1 = Promise.resolve("a");  var p2 = Promise.resolve("b");  p1.then(p2).then(function (msg) {  console.log(msg); //a  });  p1.then(function () {  p2;  }).then(function (msg) {  console.log(msg); //undefined  });  p1.then(function () {  return p2;  }).then(function (msg) {  console.log(msg); //b  });  这样他就会如我们所想的打印出 bar。  因此记住：永远都是往 then() 中传递函数！ |
| forEach | 用了 promises 后怎么用 forEach?  这里是大多数人对于 promises 的理解开始出现偏差。一旦当他们要使用他们熟悉的 forEach() 循环 (无论是 for 循环还是 while 循环)，他们完全不知道如何将 promises 与其一起使。因此他们就会写下类似这样的代码。  // 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!  });  这份代码有什么问题？问题在于第一个函数实际上返回的是 undefined，这意味着第二个方法不会等待所有 documents 都执行 db.remove()。实际上他不会等待任何事情，并且可能会在任意数量的文档被删除后执行！  这是一个非常隐蔽的 bug，因为如果 PouchDB 删除这些文档足够快，你的 UI 界面上显示的会完成正常，你可能会完全注意不到有什么东西有错误。这个 bug 可能会在一些古怪的竞态问题或一些特定的浏览器中暴露出来，并且到时可能几乎没有可能去定位问题。  简而言之，forEach()/for/while 并非你寻找的解决方案。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!  });  上面的代码是什么意思呢？大体来说，Promise.all()会以一个 promises 数组为输入，并且返回一个新的 promise。这个新的 promise 会在数组中所有的 promises 都成功返回后才返回。他是异步版的 for 循环。  并且 Promise.all() 会将执行结果组成的数组返回到下一个函数，比如当你希望从 PouchDB 中获取多个对象时，会非常有用。此外一个更加有用的特效是，一旦数组中的 promise 任意一个返回错误，Promise.all() 也会返回错误。 |
| 忘记使用.catch | 忘记使用 .catch()  这是另一个常见的错误。单纯的坚信自己的 promises 会永远不出现异常，很多开发者会忘记在他们的代码中添加一个 .catch()。然而不幸的是这也意味着，任何被抛出的异常都会被吃掉，并且你无法在 console 中观察到他们。这类问题 debug 起来会非常痛苦。  类似 Bluebird 之类的 Promise 库会在这种场景抛出 UnhandledRejectionError 警示有未处理的异常，这类情况一旦发现，就会造成脚本异常，在 Node 中更会造成进程 Crash 的问题，因此正确的添加 .catch() 非常重要。 译者注  为了避免这类讨厌的场景，我习惯于像下面的代码一样使用 promise:  somePromise().then(function () {  return anotherPromise();  }).then(function () {  return yetAnotherPromise();  }).catch(console.log.bind(console)); // <-- this is badass  即使你坚信不会出现异常，添加一个 catch() 总归是更加谨慎的。如果你的假设最终被发现是错误的，它会让你的生活更加美好。 |
| 与deferred的关系 | 使用deferred  这是一个我[经常可以看到的错误](http://gonehybrid.com/how-to-use-pouchdb-sqlite-for-local-storage-in-your-ionic-app/" \t "http://bubufx.com/_blank)，以至于我甚至不愿意在这里重复它，就像惧怕 Beetlejuice 一样，仅仅是提到它的名字，就会召唤出来更多。  简单的说，promises 拥有一个漫长并且戏剧化的历史，Javascript 社区花费了大量的时间让其走上正轨。在早期，deferred 在 Q，When，RSVP，Bluebird，Lie等等的 “优秀” 类库中被引入， jQuery 与 Angular 在使用 ES6 Promise 规范之前，都是使用这种模式编写代码。  因此如果你在你的代码中使用了这个词 (我不会把这个词重复第三遍！)，你就做错了。下面是说明一下如何避免它。  首先，大部分 promises 类库都会提供一个方式去包装一个第三方的 promises 对象。举例来说，Angular的 $q 模块允许你使用 $q.when包裹非 $q 的 promises。因此 Angular 用户可以这样使用 PouchDB promises.  $q.when(db.put(doc)).then(/\* ... \*/); // <-- this is all the code you need  另一种策略是使用[构造函数声明模式](https://blog.domenic.me/the-revealing-constructor-pattern/" \t "http://bubufx.com/_blank)，它在用来包裹非 promise API 时非常有用。举例来说，为了包裹一个回调风格的 API 如 Node 的 fs.readFile ，你可以简单的这么做:  new Promise(function (resolve, reject) {  fs.readFile(‘myfile.txt‘, function (err, file) {  if (err) {  return reject(err);  }  resolve(file);  });}).then(/\* ... \*/)  完工！我们打败了可怕的 def….啊哈，抓到自己了。:)  关于为何这是一种反模式更多的内容，请查看 Bluebird 的 [promise anti-patterns](https://github.com/petkaantonov/bluebird/wiki/Promise-anti-patterns" \l "the-deferred-anti-pattern" \t "http://bubufx.com/_blank) wiki 页 |
|  | 使用副作用调用而非返回  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.  }); |
| 几个对比 | Q: 下面的四种 promises 的区别是什么  doSomething().then(function () {  return doSomethingElse();  });  doSomething().then(function () {  doSomethingElse();  });  doSomething().then(doSomethingElse());  doSomething().then(doSomethingElse); |
|  | 这里是谜题的所有答案，我以图形的格式展示出来方便你查看：  Puzzle #1  doSomething().then(function () {  return doSomethingElse();  }).then(finalHandler);  Answer:  doSomething  |-----------------|  doSomethingElse(undefined)  |------------------|  finalHandler(resultOfDoSomethingElse)  |------------------|  Puzzle #2  doSomething().then(function () {  doSomethingElse();  }).then(finalHandler);  Answer:  doSomething  |-----------------|  doSomethingElse(undefined)  |------------------|  finalHandler(undefined)  |------------------|  Puzzle #3  doSomething().then(doSomethingElse()).then(finalHandler);  Answer:  doSomething  |-----------------|  doSomethingElse(undefined)  |---------------------------------|  finalHandler(resultOfDoSomething)  |------------------|  Puzzle #4  doSomething().then(doSomethingElse).then(finalHandler);  Answer:  doSomething  |-----------------|  doSomethingElse(resultOfDoSomething)  |------------------|  finalHandler(resultOfDoSomethingElse)  |------------------|  如果这些答案你依然无法理解，那么我强烈建议你重新读一下这篇文章，或者实现一下 doSomething() 和 doSomethingElse() 函数并且在浏览器中自己试试看。  声明：在这些例子中，我假定 doSomething() 和 doSomethingElse() 均返回 promises，并且这些 promises 代表某些在 JavaScript event loop (如 IndexedDB, network, setTimeout) 之外的某些工作结束，这也是为何它们在某些时候表现起来像是并行执行的意义。这里是一个模拟用的 [JSBin](http://jsbin.com/tuqukakawo/1/edit?js,console,output" \t "http://bubufx.com/_blank)。  关于更多 promises 的进阶用法，可以参考我的 [promise protips cheat sheet](https://gist.github.com/nolanlawson/6ce81186421d2fa109a4" \t "http://bubufx.com/_blank)  <https://gist.github.com/nolanlawson/6ce81186421d2fa109a4> |

[异步编程 promise模式 的简单实现](https://segmentfault.com/a/1190000003028634)

|  |  |
| --- | --- |
| 说明 | <https://segmentfault.com/a/1190000003028634> |
| promise的核心是有一个promise对象，这个对象有一个重要的then()方法, 它用于指定回调函数，如：  f1().then(f2);  promise模式在任何时刻都有三种状态：已完成(resolved)，未完成(unfulfilled)，那么then()方法就是为状态变化指定不同的回调函数，并总是返回一个promise对象，方便链式调用。  那promise模式下，返回的数据如何在各个回调函数之间传播呢，通过resolve方法，你可以将一个函数的返回值作为参数传递给另一个函数，并且将另一个函数的返回值作为参数再传递给下一个函数……像一条“链”一样无限的这么做下去。 |
| 代码实现 | **/\*constructor\*/ var Promise** = **function** () {  **this**.callbacks = []; };  **Promise**.prototype = {  construct: **Promise**,  resolve: **function** (result) {  **this**.complete("resolve", result);  },  reject: **function** (result) {  **this**.complete("reject", result);  },  complete: **function** (type, result) {  **while** (**this**.callbacks[0]) {  **this**.callbacks.shift()[type](result);  }  },  then: **function** (successHandler, failedHandler) {  **this**.callbacks.push({  resolve: successHandler,  reject: failedHandler  });  **return this**;  } };  **/\*test:\*/ var promise** = **new Promise**(); **var delay1** = **function** () {  setTimeout(**function** () {  **promise**.resolve('数据1');  }, 1000);  **return promise**; }; **var callback1** = **function** (re) {  re = re + '数据2';  console.log(re);  **promise**.resolve(re); }; **var callback2** = **function** (re) {  console.log(re + '数据3'); }; **delay1**().then(**callback1**).then(**callback2**);  **/\*result:{ 1秒后显示 }\*/** //数据1数据2 //数据1数据2数据3 |
| 代码分析 | 通过创建一个Promise构造函数来实现promise模式，其构造函数的结构：  callbacks: 用于管理回调函数  resolve: 请求成功时执行的方法  reject:请求失败时执行的方法  complete: 执行回调  then：绑定回调函数 |
| 分析： //第一步 **var delay1** = **function**() {  setTimeout(**function**() {  promise.resolve('数据1');  }, 1000);  **return** promise; }; 这个函数通过setTimeout方法，异步传递一个数据1，并返回一个promise对象(必须)。 //第二步 **var callback1** = **function**(re) {  re = re + '数据2';  console.log(re);  promise.resolve(re); }; callback1和callback2都是要通过then方法注册的回调函数，其中callback1通过resolve方法把数据往下传递。 //第三步 **delay1**().then(**callback1**).then(callback2); **delay1**()方法执行完，因为返回了一个promise对象，所以可以再调用then()方法为delay1()的setTimeout异步执行操作指定回调函数, 又因为then()方法也返回的是promise对象，所以还可以再调用then方法 //第四步 setTimeout(**function**() {  promise.resolve('数据1'); }, 1000); 一秒之后，当其他代码执行完成，开始执行异步代码promise.resolve('数据1');，这里调用promise的resolve()方法，指定了一个成功状态，并把数据1作为参数。 //第五步 resolve: **function**(result) {  **this**.complete("resolve", result); }, //第六步：循环执行回调，将上一个回调的结果传递给下一个回调 complete: **function**(type, result) {  **while** (**this**.callbacks[0]) {  **this**.callbacks.shift()[type](result);  } }, 这其中比较难理解的就是第五，六步。 |
|  | 难理解请参考：  [Promise/A的误区以及实践](http://qingbob.com/promise-a-misunderstanding-and-practical/" \t "https://segmentfault.com/a/_blank) [JavaScript异步编程的Promise模式](http://www.infoq.com/cn/news/2011/09/js-promise" \t "https://segmentfault.com/a/_blank) [Javascript异步编程的4种方法](http://www.ruanyifeng.com/blog/2012/12/asynchronous%EF%BC%BFjavascript.html" \t "https://segmentfault.com/a/_blank) |

异步控制工具

|  |  |
| --- | --- |
| 比较 | nodejs异步控制「co、async、Q 、『es6原生promise』、then.js、bluebird」有何优缺点？最爱哪个？哪个简单？  <https://www.zhihu.com/question/25413141> |
|  | 作者：尤雨溪 链接：https://www.zhihu.com/question/25413141/answer/30767780 来源：知乎 著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。  要说简单，async 是最简单的，只是在 callback 上加了一些语法糖而已。在不是很复杂的用例下够用了，前提是你已经习惯了 callback 风格的写法。  then.js 上手也是比较简单的，因为也是基于 callback 和 continuation passing，并不引入额外的概念，比起 async，链式 API 更流畅，个人挺喜欢的。我挺久以前写过一个在 Node 里面跑 shell 命令的小工具，思路差不多：[https://www.npmjs.org/package/shell-task](https://link.zhihu.com/?target=https://www.npmjs.org/package/shell-task" \t "_blank)  Callback-based 方案的最大问题在于异常处理，每个 callback 都得额外接受一个异常参数，发生异常就得一个一个往后传，异常发生后的定位很麻烦。  ES6 Promise, Q, Bluebird 核心都是 Promise，缺点嘛就是必须引入这个新概念并且要用就得所有的地方都用 Promise。对于 Node 的原生 API，需要进行二次封装。Q 和 Bluebird 都是在实现 Promise A+ 标准的基础上提供了一些封装和帮助方法，比如 Promise.map 来进行并行操作等等。Promise 的一个问题就是性能，而 Bluebird 号称速度是所有 Promise 库里最快的。ES6 Promise 则是把 Promise 的包括进 js 标准库里，这样你就不需要依赖第三方实现了。  关于 Promise 能够如何改进异步流程，建议阅读：[http://www.html5rocks.com/en/tutorials/es6/promises/#toc-parallelism-sequencing](https://link.zhihu.com/?target=http://www.html5rocks.com/en/tutorials/es6/promises/%23toc-parallelism-sequencing" \t "_blank)  co 是 TJ 大神基于 ES6 generator 的异步解决方案。要理解 co 你得先理解 ES6 generator，这里就不赘述了。co 最大的好处就是能让你把异步的代码流程用同步的方式写出来，并且可以用 try/catch |
|  |  |

q.js

|  |  |
| --- | --- |
|  | 利用q.js实现node 常用api的promise化  <http://www.tuicool.com/articles/AjaUjyJ>  Q.js中的几种创建promise对象的方法对比  <https://segmentfault.com/a/1190000004159962>  q.js与jquery promise的区别  <http://stackoverflow.com/questions/13610741/use-jquery-or-q-js-for-promises>  学习 q 的 API，利用 q 来替代回调函数(<https://github.com/kriskowal/q>)  <https://github.com/alsotang/node-lessons/tree/master/lesson17> |
|  | **/\*  关于啥是promise以及promise解决的是啥问题，  敬请体验node的回调异步编码大法，顺带移步http://wiki.commonjs.org/wiki/Promises/A 看看是咋定义的，在此不再赘述。   这里我们看看怎么用q.js 实现node api的promise。  1、万事开始皆为install  npm install q  2、标准node style api 的promise化方法  \*/** |
|  | // 学习一个简单的例子： **var Q** = require('q'); **var defer** = **Q**.defer();  **var initDefer** = **defer**.promise; **initDefer**.then(**function** (success) {  console.log(success); }, **function** (error) {  console.log(error); }, **function** (progress) {  console.log(progress); }); **defer**.notify('in progress');//控制台打印in progress **defer**.resolve('resolve'); //控制台打印resolve **defer**.reject('reject'); //没有输出。promise的状态只能改变一次 |
| begin | **var Q** = require('q'); **var defer** = **Q**.defer(); /\*\*  \* 通过defer获得promise  \* **@private** \*/ **function getInputPromise**() {  **return defer**.promise; } |
| promise的传递 | // var outputPromise = getInputPromise().then(function (fulfilled) { }, function (rejected) { }); // then方法会返回一个promise，在下面这个例子中，我们用outputPromise指向then返回的promise。 // outputPromise就变成了受 function(fulfilled) 或者 function(rejected)控制状态的promise了。  // 当function(fulfilled)或者function(rejected)返回一个值，比如一个字符串，数组，对象等等，那么outputPromise的状态就会变成fulfilled。 // 在下面这个例子中，我们可以看到，当我们把inputPromise的状态通过defer.resovle()变成fulfilled时，控制台输出fulfilled. // 当我们把inputPromise的状态通过defer.reject()变成rejected，控制台输出rejected   /\*\*  \* 当inputPromise状态由未完成变成fulfil时，调用function(fulfilled)  \* 当inputPromise状态由未完成变成rejected时，调用function(rejected)  \* 将then返回的promise赋给outputPromise  \* function(fulfilled) 和 function(rejected) 通过返回字符串将outputPromise的状态由  \* 未完成改变为fulfilled  \* **@private** \*/ **var outputPromise** = **getInputPromise**().then(**function** (fulfilled) {  **return** 'fulfilled'; }, **function** (rejected) {  **return** 'rejected'; });  /\*\*  \* 当outputPromise状态由未完成变成fulfil时，调用function(fulfilled)，控制台打印'fulfilled: fulfilled'。  \* 当outputPromise状态由未完成变成rejected, 调用function(rejected), 控制台打印'rejected: rejected'。  \*/ **outputPromise**.then(**function** (fulfilled) {  console.log('fulfilled: ' + fulfilled); }, **function** (rejected) {  console.log('rejected: ' + rejected); });  // 处理方式1： 将inputPromise的状态由未完成变成rejected **defer**.reject(); //输出 fulfilled: rejected  // 处理方式2： 将inputPromise的状态由未完成变成fulfilled //defer.resolve(); //输出 fulfilled: fulfilled |
|  | **/\* 当function(fulfilled)或者function(rejected)抛出异常时，那么outputPromise的状态就会变成rejected \*/**  /\*\*  \* 当inputPromise状态由未完成变成fulfil时，调用function(fulfilled)  \* 当inputPromise状态由未完成变成rejected时，调用function(rejected)  \* 将then返回的promise赋给outputPromise  \* function(fulfilled) 和 function(rejected) 通过抛出异常将outputPromise的状态由  \* 未完成改变为reject  \* **@private** \*/ **var outputPromise** = **getInputPromise**().then(**function** (fulfilled) {  **throw new** Error('fulfilled'); }, **function** (rejected) {  **throw new** Error('rejected'); });  /\*\*  \* 当outputPromise状态由未完成变成fulfil时，调用function(fulfilled)。  \* 当outputPromise状态由未完成变成rejected, 调用function(rejected)。  \*/ **outputPromise**.then(**function** (fulfilled) {  console.log('fulfilled: ' + fulfilled); }, **function** (rejected) {  console.log('rejected: ' + rejected); });  /\*\* 将inputPromise的状态由未完成变成rejected \*/ **defer**.reject(); //控制台打印 rejected [Error:rejected]  /\*\* 将inputPromise的状态由未完成变成fulfilled \*/ //defer.resolve(); //控制台打印 rejected [Error:fulfilled] |
|  | // 当function(fulfilled)或者function(rejected)返回一个promise时，outputPromise就会成为这个新的promise. // 这样做有什么意义呢 ? 主要在于聚合结果(Q.all) ，管理延时，异常恢复等等 // 比如说我们想要读取一个文件的内容，然后把这些内容打印出来。可能会写出这样的代码：  **var outputPromise** = **getInputPromise**().then(**function** (fulfilled) {//错误的写法，因为function(fulfilled)并没有返回任何值。  **fs**.readFile('test.txt', 'utf8', **function** (err, data) {  **return** data;  }); });  **var outputPromise** = **getInputPromise**().then(**function** (fulfilled) {//正确的写法，  **var** myDefer = **Q**.defer();  **fs**.readFile('test.txt', 'utf8', **function** (err, data) {  **if** (!err && data) {  myDefer.resolve(data);  }  });  **return** myDefer.promise; }, **function** (rejected) {  **throw new** Error('rejected'); });  //当outputPromise状态由未完成变成fulfil时，调用function(fulfilled)，控制台打印test.txt文件内容。 **outputPromise**.then(**function** (fulfilled) {  console.log(fulfilled); }, **function** (rejected) {  console.log(rejected); });  // 将inputPromise的状态由未完成变成fulfilled **defer**.resolve(); //控制台打印出 test.txt 的内容  // 将inputPromise的状态由未完成变成rejected //defer.reject(); |
| 方法传递 | **/\* 方法传递有些类似于Java中的try和catch。当一个异常没有响应的捕获时，这个异常会接着往下传递。 方法传递的含义是当一个状态没有响应的回调函数，就会沿着then往下找。  例如：[ var outputPromise = getInputPromise().then(function(fulfilled){}) ]中outputPromise没有提供function(rejected)； 如果inputPromise的状态由未完成变成rejected, 此时对rejected的处理会由outputPromise来完成。 \*/**  // 当inputPromise状态由未完成变成fulfil时，调用function(fulfilled) // 当inputPromise状态由未完成变成rejected时，这个rejected会传向outputPromise **var outputPromise** = **getInputPromise**().then(**function**(fulfilled){  **return** 'fulfilled' }); **outputPromise**.then(**function**(fulfilled){  console.log('fulfilled: ' + fulfilled); },**function**(rejected){  console.log('rejected: ' + rejected); });  **defer**.reject('input promise rejected'); // 将inputPromise的状态由未完成变成rejected ; 控制台打印rejected: input promise rejected //defer.resolve();// 将inputPromise的状态由未完成变成fulfilled |
|  | **/\*  如果没有提供function(fulfilled)：  var outputPromise = getInputPromise().then(null,function(rejected){})  如果 inputPromise的状态由未完成变成fulfilled, 此时对fulfil的处理会由outputPromise来完成。 \*/** **var outputPromise** = **getInputPromise**().then(**null**,**function**(rejected){  **return** 'rejected'; }); **outputPromise**.then(**function**(fulfilled){  console.log('fulfilled: ' + fulfilled); },**function**(rejected){  console.log('rejected: ' + rejected); }); //defer.reject('inputpromise rejected'); **defer**.resolve('input promise fulfilled'); //控制台打印fulfilled: input promise fulfilled |
| .fail | //可以使用fail(function(error))来专门针对错误处理，而不是使用then(null,function(error)) //var outputPromise = getInputPromise().fail(function(error){})  /\*\*  \* 当inputPromise状态由未完成变成fulfil时，调用then(function(fulfilled))  \* 当inputPromise状态由未完成变成rejected时，调用fail(function(error))  \* function(fulfilled)将新的promise赋给outputPromise  \* 未完成改变为reject  \* **@private** \*/ **var outputPromise** = **getInputPromise**().then(**function**(fulfilled){  **return** fulfilled; }).fail(**function**(error){  console.log('fail: ' + error); }); **defer**.reject('input promise rejected');//控制台打印fail: input promise rejected //defer.resolve('input promise fulfilled'); |
| .progress | // 可以使用progress(function(progress))来专门针对进度信息进行处理，而不是使用 then(function(success){},function(error){},function(progress){}) // var outputPromise = getInputPromise().progress(function(progress){ console.log(progress); })  /\*\*  \* 为promise设置progress信息处理函数  \*/ **var outputPromise** = **getInitialPromise**().then(**function**(success){  }).progress(**function**(progress){  console.log(progress); }); **defer**.notify(1); **defer**.notify(2); //控制台打印1，2 |
| promise链 | **/\*  promise链提供了一种让函数顺序执行的方法。  函数顺序执行是很重要的一个功能。比如知道用户名，需要根据用户名从数据库中找到相应的用户，然后将用户信息传给下一个函数进行处理。 \*/  var Q** = require('q'); **var defer** = **Q**.defer(); **var users** = [{'name': 'andrew', 'passwd': 'password'}];//一个模拟数据库 **function getUsername**() {  **return defer**.promise; } **function getUser**(username) {  **var** user;  **users**.forEach(**function** (element) {  **if** (element.name === username) {  user = element;  }  });  **return** user; } //promise链 **getUsername**().then(**function** (username) {  **return getUser**(username); }).then(**function** (user) {  console.log(user); }); **defer**.resolve('andrew');   **/\*  我们通过两个then达到让函数顺序执行的目的。then的数量其实是没有限制的。当然，then的数量过多，要手动把他们链接起来是很麻烦的。比如:  foo(initialVal).then(bar).then(baz).then(qux)   这时我们需要用代码来动态制造promise链:   var funcs = [foo,bar,baz,qux]  var result = Q(initialVal)  funcs.forEach(function(func){  result = result.then(func)  })  return result   当然，我们可以再简洁一点:   var funcs = [foo,bar,baz,qux]  funcs.reduce(function(pre,current),Q(initialVal){  return pre.then(current)  }) \*/**  **function foo**(result) {  console.log(result);  **return** result + result; } //手动链接 Q('hello').then(**foo**).then(**foo**).then(**foo**); // 控制台输出： // hello // hellohello // hellohellohello  //动态链接 **var funcs** = [**foo**, **foo**, **foo**]; **var result** = Q('hello'); **funcs**.forEach(**function** (func) {  **result** = **result**.then(func); });  //精简后的动态链接 **funcs**.reduce(**function** (prev, current) {  **return** prev.then(current); }, Q('hello')); |
| promise组合 | **/\* 修改回调链 \*/** //邪恶金字塔(Pyramid of Doom) **var fs** = require('fs'); **fs**.readFile('sample01.txt', 'utf8', **function** (err, data) {  **fs**.readFile('sample02.txt', 'utf8', **function** (err, data) {  **fs**.readFile('sample03.txt', 'utf8', **function** (err, data) {  **fs**.readFile('sample04.txt', 'utf8', **function** (err, data) {   });  });  }); });  //它改写成promise链 **var Q** = require('q'),  **fs** = require('fs'); **function printFileContent**(fileName) {  **var** defer = **Q**.defer();  **fs**.readFile(fileName, 'utf8', **function** (err, data) {  **if** (!err && data) {  console.log(data);  defer.resolve();  }  });  **return** defer.promise; } //手动链接 **printFileContent**('sample01.txt')  .then(**printFileContent**('sample02.txt'))  .then(**printFileContent**('sample03.txt'))  .then(**printFileContent**('sample04.txt')); //控制台顺序打印sample01到sample04的内容 |
|  | **/\*  很有成就感是不是。然而如果仔细分析，我们会发现为什么要他们顺序执行呢，如果他们能够并行执行不是更好吗? 我们只需要在他们都执行完成之后，得到他们的执行结果就可以了。  我们可以通过Q.all([promise1,promise2...])将多个promise组合成一个promise返回。   注意：  当all里面所有的promise都fulfil时，Q.all返回的promise状态变成fulfil  当任意一个promise被reject时，Q.all返回的promise状态立即变成reject   我们来把上面读取文件内容的例子改成并行执行吧  \* \*/ var Q** = require('q'); **var fs** = require('fs'); **function printFileContent**(fileName) {  //***Todo: 这段代码不够简洁。可以使用Q.denodeify来简化* var** defer = **Q**.defer();  **fs**.readFile(fileName, 'utf8', **function** (err, data) {  **if** (!err && data) {  console.log(data);  defer.resolve(fileName + ' success ');  } **else** {  defer.reject(fileName + ' fail ');  }  });  **return** defer.promise; } **var printFileContentArr** = [  **printFileContent**('sample01.txt'),  **printFileContent**('sample02.txt'),  **printFileContent**('sample03.txt'),  **printFileContent**('sample04.txt') ]; **Q**.all(**printFileContentArr**).then(**function** (success) {  console.log(success); }); //控制台打印各个文件内容 顺序不一定 |
|  | **/\*  \* 现在知道Q.all会在任意一个promise进入reject状态后立即进入reject状态。  \* 如果我们需要等到所有的promise都发生状态后(有的fulfil, 有的reject)，再转换Q.all的状态, 这时我们可以使用Q.allSettled  \* \*/ Q**.allSettled(**printFileContentArr**).then(**function** (results) {  results.forEach(  **function** (result) {  console.log(result.state);  }  ); }); |
| 结束promise链 | **/\*  通常，对于一个promise链，有两种结束的方式。第一种方式是返回最后一个promise  如 return foo().then(bar);   第二种方式就是通过done来结束promise链  如 foo().then(bar).done()   为什么需要通过done来结束一个promise链呢?  如果在我们的链中有错误没有被处理，那么在一个正确结束的promise链中，这个没被处理的错误会通过异常抛出。  \*/  var Q** = require('q'); **function getPromise**(msg, timeout, opt) {  **var** defer = **Q**.defer();  setTimeout(**function** () {  console.log(msg);  **if** (opt) {  defer.reject(msg);  } **else** {  defer.resolve(msg);  }  }, timeout);  **return** defer.promise; }  **/\*  \*没有用done()结束的promise链  \*由于getPromise('2',2000,'opt')返回rejected, getPromise('3',1000)就没有执行  \*然后这个异常并没有任何提醒，是一个潜在的bug  \*/ getPromise**('1', 3000).then(**function** () {  **return getPromise**('2', 2000, 'opt') }).then(**function** () {  **return getPromise**('3', 1000) });  **/\*  \*用done()结束的promise链  \*有异常抛出  \*/ getPromise**('1', 3000).then(**function** () {  **return getPromise**('2', 2000, 'opt') }).then(**function** () {  **return getPromise**('3', 1000) }).done(); |
| 其他 | //1、使用Q.nfcall 相对于Q.fcall ,Q.nfcall 就是node 的Q.fcall。 **var fs** = require('fs'),  **Q** = require('q'),  **colors** = require('colors'),  **file** = 'example.txt'; **var fsReadFile** = **Q**.nfcall(**fs**.readFile, **file**, encoding); **var fsReadFile** = **Q**.nfapply(**fs**.readFile, ['foo.txt', 'utf-8']); **fsReadFile**.then(**function** (result) {  console.log(("invoke in nfcall " + **file**).red);  console.log(result.green);  }, **function** (error) {  console.log("invoke in nfcall".red);  console.log(error.toString().red);  } ); //2、使用Q.deferd **var fsReadFile\_deferd** = **function** (file, encoding) {  **var** deferred = **Q**.defer();  **fs**.readFile(file, encoding, **function** (error, result) {  **if** (error) {  deferred.reject(error.toString().red);  }  deferred.resolve(result);  });  **return** deferred.promise; };  **fsReadFile\_deferd**(**file**).then(**function** (result) {  console.log("invoke in deferd".red);  console.log(result.toString().green);  }, **function** (error) {  console.log("invoke in deferd".red);  console.log(error.toString().red);  } );  **/\*  以上三种方法，都存在一个麻烦的问题：如果我的需求场景变成了首先读入foo.txt文件，然后根据文件中的内容来决定是否调用另一个文件，并且打印出另一个文件的内容。这种场景我们常常会在数据库查找中用到——先查询到某个id，然后在另一个库中根据这个id查到其他某些字段的数据。这就涉及到链式调用。  上面的三种方法，如果在链式调用中我不想让代码变得冗余，写一堆readB、readC……的方法（有些时候甚至究竟链式调用会经历多少步，即要使用多少个then我们事先都不清楚，需要根据获得的数据来动态决定）。  在这种情况下，使用Q.denodeify最合适：  \* \*/** //3使用Q.denodeify **var fsReadFile\_denodeify** = **Q**.denodeify(**fs**.readFile); **fsReadFile\_denodeify**(**file**, encoding).then(**function** (result) {  console.log("invoke in denodeify".red);  console.log(result.green)  }, **function** (error) {  console.log("invoke in denodeify".red);  console.log(error.toString().red);  } );  **fsReadFile\_denodeify**('foo.txt', 'utf-8').then(**function** (data) {  console.log(data);  **return fsReadFile\_denodeify**('foo2.txt', 'utf-8');//只要在这里返回新的promise对象，就能把异步执行结果返回给下一个then()方法 }).then(**function** (data) {  console.log(data);  **return fsReadFile\_denodeify**("f.txt", "utf-8"); });   //4、使用makeNodeResolver() **var fsReadFile\_makeNodeResolver** = **function** (file, encoding) {  **var** deferred = **Q**.defer();  **fs**.readFile(file, encoding, deferred.makeNodeResolver());  **return** deferred.promise; };  **fsReadFile\_makeNodeResolver**(**file**, encoding).then(**function** (result) {  console.log("invoke in makeNodeResolver".red);  console.log(result.green); }, **function** (error) {  console.log(error.toString().red); }); |

bluebird

|  |  |
| --- | --- |
| promise-异步编程 | 关于动画的一个案例  npm install bluebird |
|  | 获取全局的变量 Promise  **var Promise**= window.Promise; **function promiseAnimate**(ball,time){  **return new Promise**(**function**(resolve,reject){  **function \_animate**( ){//定义  setTimeout(**function**(){  **if**("满足执行回调的条件"){  resolve  }**else**{  **\_animate**()  }  },13)  }  **\_animate**();//执行  }) }  **promiseAnimate**("ball1",100).then(**function**(){  **return promiseAnimate**("ball2",200); }).then(**function**(){  **return promiseAnimate**("ball3",300); }).then(**function**(){  **return promiseAnimate**("ball2",200); }) |
|  | <https://jdc.jd.com/archives/1704>  <http://www.zcfy.cc/article/what-is-promise-try-and-why-does-it-matter-joepie91-s-ramblings-385.html>  <http://cryto.net/~joepie91/blog/2016/05/11/what-is-promise-try-and-why-does-it-matter/>  [[译] 什么是Promise.try，它为何重要？](https://jdc.jd.com/archives/1704" \o "[译] 什么是Promise.try，它为何重要？)  以上的代码看起来似乎是多余的。但实际上它有以下几个优点：  1.更好的错误处理 同步代码中的异常不论出现在何处都会以rejection的形式向Promise链后端传递。  2.更好的兼容性 你可以始终使用你自己喜欢的Promise实现，而不用担心第三方代码在使用哪个。  3.更好的代码阅读体验 所有的代码在水平方向上将处于同一个缩进层级，这将使你阅读代码变得更容易。  <http://ricostacruz.com/cheatsheets/bluebird.html>  bluebird.js cheatsheet |
|  |  |

升级

|  |  |
| --- | --- |
|  | [Generator函数的含义与用法](http://www.ruanyifeng.com/blog/2015/04/generator.html" \t "http://www.ruanyifeng.com/blog/2015/05/_blank)  [Thunk函数的含义与用法](http://www.ruanyifeng.com/blog/2015/05/thunk.html" \t "http://www.ruanyifeng.com/blog/2015/04/_blank)  [co函数库的含义与用法](http://www.ruanyifeng.com/blog/2015/05/co.html" \t "http://www.ruanyifeng.com/blog/2015/04/_blank)  [async函数的含义与用法](http://www.ruanyifeng.com/blog/2015/05/async.html" \t "http://www.ruanyifeng.com/blog/2015/04/_blank) |

Generator

Generator 函数的异步应用

|  |  |
| --- | --- |
|  | <http://es6.ruanyifeng.com/#docs/generator-async>  Generator 函数的异步应用 |
|  | 以前，异步编程的方法，大概有[下面四种](http://www.ruanyifeng.com/blog/2012/12/asynchronous%EF%BC%BFjavascript.html" \t "http://www.ruanyifeng.com/blog/2015/04/_blank)。  > 回调函数  > 事件监听  > 发布/订阅  > Promise 对象  ECMAScript 6 (简称 ES6 )作为下一代 JavaScript 语言，将 JavaScript 异步编程带入了一个全新的阶段。这组系列文章的主题，就是介绍更强大、更完善的 ES6 异步编程方法。  新方法比较抽象，初学时，我常常感到费解，直到很久以后才想通，异步编程的语法目标，就是怎样让它更像同步编程。这组系列文章，将帮助你深入理解 JavaScript 异步编程的本质。所有将要讲到的内容，都已经实现了。也就是说，马上就能用，套用一句广告语，就是"未来已来"。 |
| 回调函数 | 回调函数的概念  JavaScript 语言对异步编程的实现，就是回调函数。所谓回调函数，就是把任务的第二段单独写在一个函数里面，等到重新执行这个任务的时候，就直接调用这个函数。它的英语名字 callback，直译过来就是"重新调用"。  读取文件进行处理，是这样写的。  fs.readFile('/etc/passwd', **function** (err, data) {  **if** (err) **throw** err;  console.log(data); });  上面代码中，readFile 函数的第二个参数，就是回调函数，也就是任务的第二段。等到操作系统返回了 /etc/passwd 这个文件以后，回调函数才会执行。  一个有趣的问题是，为什么 Node.js 约定，回调函数的第一个参数，必须是错误对象err(如果没有错误，该参数就是 null)？原因是执行分成两段，在这两段之间抛出的错误，程序无法捕捉，只能当作参数，传入第二段。 |
| Promise | 回调函数本身并没有问题，它的问题出现在多个回调函数嵌套。假定读取A文件之后，再读取B文件，代码如下。  **fs**.readFile(fileA, **function** (err, data) {  **fs**.readFile(fileB, **function** (err, data) {  // ...   }); });  不难想象，如果依次读取多个文件，就会出现多重嵌套。代码不是纵向发展，而是横向发展，很快就会乱成一团，无法管理。这种情况就称为["回调函数噩梦"](http://callbackhell.com/" \t "http://www.ruanyifeng.com/blog/2015/04/_blank)(callback hell)。  Promise就是为了解决这个问题而提出的。它不是新的语法功能，而是一种新的写法，允许将回调函数的横向加载，改成纵向加载。采用Promise，连续读取多个文件，写法如下。  **var readFile** = require('fs-readfile-promise'); **readFile**(fileA).then(**function** (data) {  console.log(data.toString()); }).then(**function** () {  **return readFile**(fileB); }).then(**function** (data) {  console.log(data.toString()); }).catch(**function** (err) {  console.log(err); });  上面代码中，我使用了 [fs-readfile-promise](https://www.npmjs.com/package/fs-readfile-promise" \t "http://www.ruanyifeng.com/blog/2015/04/_blank) 模块，它的作用就是返回一个 Promise 版本的 readFile 函数。Promise 提供 then 方法加载回调函数，catch方法捕捉执行过程中抛出的错误。  可以看到，Promise 的写法只是回调函数的改进，使用then方法以后，异步任务的两段执行看得更清楚了，除此以外，并无新意。  Promise 的最大问题是代码冗余，原来的任务被Promise 包装了一下，不管什么操作，一眼看去都是一堆 then，原来的语义变得很不清楚。 |

|  |  |
| --- | --- |
| Generator | 协程  传统的编程语言，早有异步编程的解决方案(其实是多任务的解决方案)。其中有一种叫做["协程"](https://en.wikipedia.org/wiki/Coroutine" \t "http://www.ruanyifeng.com/blog/2015/04/_blank)(coroutine)，意思是多个线程互相协作，完成异步任务。  协程有点像函数，又有点像线程。它的运行流程大致如下。  第一步，协程A开始执行。  第二步，协程A执行到一半，进入暂停，执行权转移到协程B。  第三步，(一段时间后)协程B交还执行权。  第四步，协程A恢复执行。  上面流程的协程A，就是异步任务，因为它分成两段(或多段)执行。  举例来说，读取文件的协程写法如下。  function asnycJob() {  // ...其他代码  var f = yield readFile(fileA);  // ...其他代码  }  上面代码的函数 asyncJob 是一个协程，它的奥妙就在其中的 yield 命令。它表示执行到此处，执行权将交给其他协程。也就是说，yield命令是异步两个阶段的分界线。  协程遇到 yield 命令就暂停，等到执行权返回，再从暂停的地方继续往后执行。它的最大优点，就是代码的写法非常像同步操作，如果去除yield命令，简直一模一样。 |
|  | Generator函数的概念  ES6带来了很多新的特性，其中生成器、yield等能对之前金字塔式的异步回调做到很好地解决，而基于此封装的co框架能让我们完全已同步的方式来编写异步代码。这篇文章就对生成器函数（GeneratorFunction）及框架thunkify、co的核心代码做比较彻底的分析。co的使用还是比较广泛的，除了我们日常的编码要用到外，一些知名框架也是基于co实现的，比如被称为下一代的Nodejs web框架的koa等。  Generator 函数是协程在 ES6 的实现，最大特点就是可以交出函数的执行权(即暂停执行)。  生成器函数是写成：  function\* func(){}  其本质也是一个函数，所以它具备普通函数所具有的所有特性。除此之外，它还具有以下有用特性：  > 执行生成器函数后返回一个生成器（Generator），且生成器具有throw()方法，可手动抛出一个异常，也常被用于判断是否是生成器；  > 在生成器函数内部可以使用yield（或者yield\*），函数执行到yield的时候都会暂停执行，并返回yield的右值（函数上下文，如变量的绑定等信息会保留），通过生成器的next()方法会返回一个对象，含当前yield右边表达式的值（value属性），以及generator函数是否已经执行完（done属性）等的信息。每次执行next()方法，都会从上次执行的yield的地方往下，直到遇到下一个yield并返回包含相关执行信息的对象后暂停，然后等待下一个next()的执行；  > 生成器的next()方法返回的是包含yield右边表达式值及是否执行完毕信息的对象；而next()方法的参数是上一个暂停处yield的返回值。 |
| 1 | function\* gen(x){  var y = yield x + 2;  return y;  }  上面代码就是一个 Generator 函数。它不同于普通函数，是可以暂停执行的，所以函数名之前要加星号，以示区别。  整个 Generator 函数就是一个封装的异步任务，或者说是异步任务的容器。异步操作需要暂停的地方，都用 yield 语句注明。Generator 函数的执行方法如下。  var g = gen(1);  g.next() // { value: 3, done: false }  g.next() // { value: undefined, done: true }  上面代码中，调用 Generator 函数，会返回一个内部指针(即[遍历器](http://es6.ruanyifeng.com/" \l "docs/iterator) )g 。这是 Generator 函数不同于普通函数的另一个地方，即执行它不会返回结果，返回的是指针对象。调用指针 g 的 next 方法，会移动内部指针(即执行异步任务的第一段)，指向第一个遇到的 yield 语句，上例是执行到 x + 2 为止。  换言之，next 方法的作用是分阶段执行 Generator 函数。每次调用 next 方法，会返回一个对象，表示当前阶段的信息( value 属性和 done 属性)。value 属性是 yield 语句后面表达式的值，表示当前阶段的值；done 属性是一个布尔值，表示 Generator 函数是否执行完毕，即是否还有下一个阶段。 |
| 2 | function test() {  return 'b';  }  function \*func() {  var a = yield 'a';  console.log('gen:', a); // gen: undefined  var b = yield test();  console.log('gen:', b); // gen: undefined  }  var func1 = func();  var a = func1.next();  console.log('next:', a);// next: { value: 'a', done: false }  var b = func1.next();  console.log('next:', b);// next: { value: 'b', done: false }  var c = func1.next();  console.log('next:', c);// next: { value: undefined, done: true }  根据上面说过的第3条执行准则：“生成器的next()方法返回的是包含yield右边表达式值及是否执行完毕信息的对象；而next()方法的参数是上一个暂停处yield的返回值”，因为我们没有往生成器的next()中传入任何值，所以：var a = yield 'a';中a的值为undefined。  那我们可以将例子稍微修改下：  function test(){  return 'b';  }    function\* func(){  var a = yield 'a';  console.log('gen:',a);// gen:1  var b = yield test();  console.log('gen:',b);// gen:2  }  var func2 = func();  var a = func2.next();  console.log('next:', a);// next: { value: 'a', done: false }  var b = func2.next(1);  console.log('next:', b);// next: { value: 'b', done: false }  var c = func2.next(2);  console.log('next:', c);// next: { value: undefined, done: true } |
| 3 | function\* test(p){  console.log(p); // 1  var a = yield p + 1;  console.log(a); // 3  }  var g = test(1);  var ret;  ret = g.next();  console.log(ret); // { value: 2, done: false }  ret = g.next(ret.value + 1);  console.log(ret); // { value: undefined, done: true }  > yield关键字可以让当前函数暂停执行并保存现场，并跳出到调用此函数的代码处继续执行。  > 可以利用函数执行时的返回句柄的next方法回到之前暂停处继续执行  > next执行的返回值的value即是yield关键字后面部分的表达式结果  > 下一个next的唯一参数值可以作为yield的整体返回值，并赋值给a变量  看下执行顺序就能比较清楚Generator是怎么工作的了  IMG_256 |
|  | var ll=console.log;  function\*gen(x) {  ll(2);  var y = yield x + 2;  ll(4);  ll(y)  return y;  }  var g = gen(1);  ll(1)  g.next(88) // { value: 3, done: false }  ll(3)  g.next(99) // { value: 99, done: true }  ll(5)  1  2  3  4  99  5 |
|  | Generator 函数的数据交换和错误处理  Generator 函数可以暂停执行和恢复执行，这是它能封装异步任务的根本原因。除此之外，它还有两个特性，使它可以作为异步编程的完整解决方案：函数体内外的数据交换和错误处理机制。  next 方法返回值的 value 属性，是 Generator 函数向外输出数据；next 方法还可以接受参数，这是向 Generator 函数体内输入数据。  function\* gen(x){  var y = yield x + 2;  return y;  }  var g = gen(1);  g.next() // { value: 3, done: false }  g.next(2) // { value: 2, done: true }  上面代码中，第一个 next 方法的 value 属性，返回表达式 x + 2 的值(3)。第二个 next 方法带有参数2，这个参数可以传入 Generator 函数，作为上个阶段异步任务的返回结果，被函数体内的变量 y 接收。因此，这一步的 value 属性，返回的就是2(变量 y 的值)。  Generator 函数内部还可以部署错误处理代码，捕获函数体外抛出的错误。  function\* gen(x){  try {  var y = yield x + 2;  } catch (e){  console.log(e);  }  return y;  }  var g = gen(1);  g.next();  g.throw('出错了');  // 出错了  上面代码的最后一行，Generator 函数体外，使用指针对象的 throw 方法抛出的错误，可以被函数体内的 try ... catch 代码块捕获。这意味着，出错的代码与处理错误的代码，实现了时间和空间上的分离，这对于异步编程无疑是很重要的。 |
|  | 异步任务的封装  下面看看如何使用 Generator 函数，执行一个真实的异步任务。  var fetch = require('node-fetch');  function\* gen(){  var url = '<https://api.github.com/users/github>';  var result = yield fetch(url);  console.log(result.bio);  }  上面代码中，Generator 函数封装了一个异步操作，该操作先读取一个远程接口，然后从 JSON 格式的数据解析信息。就像前面说过的，这段代码非常像同步操作，除了加上了 yield 命令。  执行这段代码的方法如下。  var g = gen();  var result = g.next();  result.value.then(function(data){  return data.json();  }).then(function(data){  g.next(data);  });  上面代码中，首先执行 Generator 函数，获取遍历器对象，然后使用 next 方法(第二行)，执行异步任务的第一阶段。由于 [Fetch 模块](https://github.com/bitinn/node-fetch" \t "http://www.ruanyifeng.com/blog/2015/04/_blank)返回的是一个 Promise 对象，因此要用 then 方法调用下一个next 方法。  可以看到，虽然 Generator 函数将异步操作表示得很简洁，但是流程管理却不方便(即何时执行第一阶段、何时执行第二阶段)。本系列的[后面部分](http://www.ruanyifeng.com/blog/2015/05/thunk.html" \t "http://www.ruanyifeng.com/blog/2015/04/_blank)，就将介绍如何自动化异步任务的流程管理。另外，本文对 Generator 函数的介绍很简单，详尽的教程请阅读我写的[《ECMAScript 6入门》](http://es6.ruanyifeng.com/" \l "docs/generator)。 |

yield\*

|  |  |
| --- | --- |
|  | 关于yield\*  yield暂停执行并只返回右值，而yield\*则将函数委托到另一个生成器或可迭代的对象（如：字符串、数组、类数组以及ES6的Map、Set等）。举例如下：  function\* genFunc(){      yield arguments;      yield\* arguments;  }    var gen = genFunc(1,2);  console.log(gen.next().value); // { '0': 1, '1': 2 }  console.log(gen.next().value); // 1  console.log(gen.next().value); // 2  function\* gen1(){      yield 2;      yield 3;  }    function\* gen2(){      yield 1;      yield\* gen1();      yield 4;  }    var g2 = gen2();  console.log(g2.next().value); // 1  console.log(g2.next().value); // 2  console.log(g2.next().value); // 3  console.log(g2.next().value); // 4 |

CO

|  |  |
| --- | --- |
|  | [co 模块](https://github.com/tj/co)是著名程序员 TJ Holowaychuk 于2013年6月发布的一个小工具，用于 Generator 函数的自动执行。是异步解决方案的库，用于让异步的代码 "同步化"。  Generator实际上是一种特殊的迭代器，不过nodejs下主流的场景是将异步回调变成同步模式。  [co模块](https://www.npmjs.org/package/co" \t "https://cnodejs.org/topic/_blank)来解释Generator的用途；  co是coordination的缩写，即协同程序。  它构建在generator、Promise基础上； |
| 首先我们简单了解下 generator：  // 定义一个generators  function\* foo(){  yield console.log("bar");  yield console.log("baz");  }  var g = foo();  g.next(); // prints "bar"  g.next(); // prints "baz"  简单来说，generator 实现了状态暂停/函数暂停 —— 通过 yield 关键字暂停函数，并返回当前函数的状态。 |
| co 实现了 generator 的 自动执行，我们使用 co 和 Promise 修改上面的代码：  var co = require('co');  function\* foo() {  yield Promise.resolve(console.log("bar"));  yield Promise.resolve(console.log("baz"));  }  co(foo);  co 有个使用条件：generator 函数的 yield 命令后面，只能是 [Thunk](https://tasaid.com/Home/Cl?url=http://www.ruanyifeng.com/blog/2015/05/thunk.html&sgs=blog-link" \t "https://tasaid.com/blog/_blank) 函数或 Promise 对象。  正是这个条件，让 co 强悍无比。 |
|  | 为什么 co 可以自动执行 Generator 函数？  前面说过，Generator 就是一个异步操作的容器。它的自动执行需要一种机制，当异步操作有了结果，能够自动交回执行权。  两种方法可以做到这一点。  （1）回调函数。将异步操作包装成 Thunk 函数，在回调函数里面交回执行权。  （2）Promise 对象。将异步操作包装成 Promise 对象，用then方法交回执行权。  co 模块其实就是将两种自动执行器（Thunk 函数和 Promise 对象），包装成一个模块。使用 co 的前提条件是，Generator 函数的yield命令后面，只能是 Thunk 函数或 Promise 对象。如果数组或对象的成员，全部都是 Promise 对象，也可以使用 co，详见后文的例子。（co v4.0版以后，yield命令后面只能是 Promise 对象，不再支持 Thunk 函数。）  上一节已经介绍了基于 Thunk 函数的自动执行器。下面来看，基于 Promise 对象的自动执行器。这是理解 co 模块必须的。 |
| demo | 下面是一个 Generator 函数，用于依次读取两个文件。  var gen = function \* () {  var f1 = yield readFile('/etc/fstab');  var f2 = yield readFile('/etc/shells');  console.log(f1.toString());  console.log(f2.toString());  };  co 模块可以让你不用编写 Generator 函数的执行器。  var co = require('co');  co(gen);  上面代码中，Generator 函数只要传入co函数，就会自动执行。  co函数返回一个Promise对象，因此可以用then方法添加回调函数。  co(gen).then(function (){  console.log('Generator 函数执行完成');  });  上面代码中，等到 Generator 函数执行结束，就会输出一行提示。 |
| demo | var co = require('co');  co(function\* (){  var now = Date.now();  yield sleep200ms;  console.log(Date.now() - now);  })();  function sleep200ms(cb) {  setTimeout(cb, 200);  }  很神奇吧？在ES6之前js中不可能实现同步的sleep功能。 |
| 我们再改写下上面的代码，让sleep函数支持自定义延迟的时间。  var co = require('co');  co(function \*(){  var now = Date.now();  yield sleep(500);  console.log(Date.now() - now);  })();  function sleep(ms){  return function(cb){  setTimeout(cb, ms);  };  } |
| 基于co，我们就可以写出类似这样的业务代码：  co(function \*(){  var rs = yield db.query('select url from xxx');  rs.forEach(rs){  var content = yield getUrl(rs.url);  ...  }  })();  在这之前，我们只能用洋葱式回调写法：  db.query('select url from xxx', function(rs){  rs.forEach(r){  getUrl(r.url, function(content){  ...  });  }  }); |
| co 支持并发的异步操作，即允许某些操作同时进行，等到它们全部完成，才进行下一步。  这时，要把并发的操作都放在数组或对象里面，跟在yield语句后面。  // 数组的写法  co(function\* () {  var res = yield [  Promise.resolve(1),  Promise.resolve(2)  ];  console.log(res);  }).catch(onerror);  // 对象的写法  co(function\* () {  var res = yield {  1: Promise.resolve(1),  2: Promise.resolve(2),  };  console.log(res);  }).catch(onerror);  下面是另一个例子。  co(function\* () {  var values = [n1, n2, n3];  yield values.map(somethingAsync);  });  function\* somethingAsync(x) {  // do something async return y  }  上面的代码允许并发三个somethingAsync异步操作，等到它们全部完成，才会进行下一步。 |
| 对比 | 使用 回调函数/Callback 的方式封装一个常见的 ajax 异步任务：  function ajax(q, callback) {  var xhr = new XMLHttpRequest();  xhr.onreadystatechange = function () {  if (xhr.readyState == 4 && xhr.status == 200) {  callback(xhr.responseText);  }  }  xhr.open("GET", "query?q=" + q);  }  我们使用 回调函数 的方式连续发 2 条请求：  ajax('foo', function (foo) {  console.log(foo);  ajax('bar', function (bar) {  console.log(bar);  });  });  这是 js 中最典型的异步处理方案。 |
| 使用 Promise 封装异步 ajax，让回调函数扁平化：  function ajax(q, callback) {  // 使用 Promise 封装  return new Promise(function (resolve) {  var xhr = new XMLHttpRequest();  xhr.onreadystatechange = function () {  if (xhr.readyState == 4 && xhr.status == 200) {  resolve(xhr.responseText);  }  }  xhr.open("GET", "query?q=" + q);  });  }  然后修改请求代码，扁平化异步代码：  ajax('foo')  .then(function (foo) {  console.log(foo);  return ajax('bar')  })  .then(function (bar) {  console.log(bar);  }); |
| 使用 co.js 来修改请求代码：  var co = require('co');  co(function\* () {  var foo = yield ajax('foo');  console.log(foo);  var bar = yield ajax('bar');  console.log(bar);  });  最终我们的异步任务，在代码中同步化了。  对于异步代码来说，回调函数是最基础的方案，带来的弊端也显而易见。  Promise 让代码扁平化，而 co 让代码同步化。 |
| co的实现原理 | 我们先来看个简化版的co代码，也算是co的骨架，官方co仅仅是功能更多更强大。  co(function \*( input ) {  var now = Date.now();  yield sleep200;  console.log(Date.now() - now);  });  function co(fn){  var gen = fn();  next();  function next(res){  var ret;  ret = gen.next(res);  // 全部结束  if(ret.done){  return;  }  // 执行回调  if (typeof ret.value == 'function') {  ret.value(function(){  next.apply(this, arguments);  });  return;  }  throw 'yield target no supported!';  }  }  function sleep200(cb){  setTimeout(cb, 200)  }  上面的代码中核心是Generator的流程控制，以及回调的执行。  通过co来执行代码的性能如何呢？以下是co官方的测试数据：  用co前： 570ms  用co后： 610ms  从数据上看，性能有一定的降低，但对大部分场景完全可以忽略这种性能损耗。 |

co thunkify

|  |  |
| --- | --- |
|  | [彻底理解thunk函数与co框架](http://blog.csdn.net/fendouzhe123/article/details/50426156)  <http://blog.csdn.net/fendouzhe123/article/details/50426156> |
| 之前的异步模块如何使用 | 几乎所有的node原生模块，以及大量的npm模块，都可以利用TJ的[thunkify模块](https://www.npmjs.org/package/thunkify" \t "https://cnodejs.org/topic/_blank)进行封装。  var co = require('co');  var thunkify = require('thunkify');  var request = require('request');  var get = thunkify(request.get);  co(function \*(){  var a = yield get('http://google.com');  var b = yield get('http://yahoo.com');  var c = yield get('http://cloudup.com');  console.log(a[0].statusCode);  console.log(b[0].statusCode);  console.log(c[0].statusCode);  })()  co(function \*(){  var a = get('http://google.com');  var b = get('http://yahoo.com');  var c = get('http://cloudup.com');  var res = yield [a, b, c];  console.log(res);  })()  // Error handling  co(function \*(){  try {  var res = yield get('http://badhost.invalid');  console.log(res);  } catch(e) {  console.log(e.code) // ENOTFOUND  }  })() |
| thunkify的实现原理 | 异步封装成同步还是非常简单的，我们看下官方的代码：  function thunkify(fn) {  assert('function' == typeof fn, 'function required');  return function() {  var args = new Array(arguments.length);  var ctx = this;  for (var i = 0; i < args.length; ++i) {  args[i] = arguments[i];  }  return function(done) {  var called;  args.push(function() {  if (called) return;  called = true;  done.apply(null, arguments);  });  try {  fn.apply(ctx, args);  } catch (err) {  done(err);  }  }  }  };  在外层封装了函数，将参数传递给异步函数，并在回调中调用done，以触发next。  可能有心的人已经留意到，无论是co还是thunkify，都增加了防重复执行判断。  由于异步回调可能会触发多次，假如多次触发next，就会造成流程混乱。 |
| 和fibers的差异 | co和Nodejs之前的[Fibers模块](https://www.npmjs.org/package/fibers" \t "https://cnodejs.org/topic/_blank)功能基本上一致。  同样我们先看下fibers是如何将异步变同步的：  var Fiber = require('fibers');  Fiber(function() {  var now = Date.now();  sleep(500);  console.log(Date.now() - now);  }).run();  function sleep(ms) {  var fiber = Fiber.current;  setTimeout(function() {  fiber.run();  }, ms);  Fiber.yield();  }  是不是感觉和co的代码风格非常神似？  fibers主流程的代码上并没有出现特殊关键字，直观上来看和PHP等各类同步语言比较接近。  我们再来看看fibers是如何将异步结果回传的。  var Fiber = require('fibers');  Fiber(function() {  var now = Date.now();  var ret = delayTask(500);  console.log(ret);  console.log(Date.now() - now);  }).run();  function delayTask(ms) {  var fiber = Fiber.current;  var ret;  setTimeout(function() {  ret = 'ok';  fiber.run();  }, ms);  Fiber.yield();  return ret;  }  上面的代码在流程上完全没变，仅仅在Fiber.yield();执行添加了return ret;。  并且在fiber.run();前进行了赋值ret = 'ok';;  本质思想上两种方案没本质区别：  Generator是利用yield特殊关键字来暂停执行，而fibers是利用Fiber.yield()暂停  Generator是利用函数返回的Generator句柄来控制函数的继续执行，而fibers是在异步回调中利用Fiber.current.run()继续执行。  简单来讲，Generator把句柄暴露给外部，而fibers把句柄暴露给内部。  上面的第2点特性决定了Generator必需把异步代码委托给外部，根据异步返回结果决定流程控制。  而fibers由于是对内暴露，因此必需要在异步回调中恢复父函数的执行。  假如异步回调多的话，就有可能在流程及内部变量使用上出现混乱。  我们再写个DEMO测下co的主流程中的异步特性：  co(function \*( input ) {  var a;  setTimeout(function(){  console.log(a); // 'abc'  }, 300)  a = yield delayTask(200);})();  function delayTask(ms){  return function(cb){  setTimeout(function(){  cb(null, 'abc')  }, ms);  }}  从结果上来看，此处特性和fibers是一致的，yield同样是不影响异步函数的执行的。  Generator在经过co封装之后，就基本上和fibers没差异了。  因此，基于co，我们也可以写出漂亮的同步风格代码。 |
|  | 主要功能：配合co使用，将function(param,callback),风格的函数，转化为thunky风格的函数，以供co yield 后面 使用    Thunks  Thunks are functions that only have a single argument, a callback. Thunk support only remains for backwards compatibility and may be removed in future versions of co.  什么是thunks  Thunks 是一个只有一个参数（即回调函数）的函数，thunk 主要是为了向后兼容，未来co可能就不会再支持这种风格的函数了  --------------------------如下代码使用thunkfy-----------------  var co = require('co');var thunkify = require("thunkify");  // 模拟一个普通的异步请求类似readFile，不是thunk风格，有两个参数第一个是函数参数，第二个是回调function myThunk(userName,callback)  {  setTimeout(function(){  callback(null,"hello "+userName );  },200)  }  // myR 是封装之后符合thunk风格的函数  var myR =thunkify(myThunk);co(function \*(){  var a = yield myR("a");  console.log(a);  return "over";  }).then(function(r){  console.log(r);  })  -------------------------------------  如上代码等价于如下不使用thunkify的代码（但是有些现成的函数库是无法这样改造的，最典型的是readFile）  var co = require('co');function myR(userName)  {  return function(callback)  {  setTimeout(function () {  callback(null,"hello "+userName );  },200)  }  }co(function \*(){  var a = yield myR("a");  console.log(a);  return "over";  }).then(function(r){  console.log(r);  }) |

async await

async 函数的含义和用法

|  |  |
| --- | --- |
|  | <http://www.ruanyifeng.com/blog/2015/05/async.html> |
|  | 异步操作是 JavaScript 编程的麻烦事，麻烦到一直有人提出各种各样的方案，试图解决这个问题。  从最早的回调函数，到 Promise 对象，再到 Generator 函数，每次都有所改进，但又让人觉得不彻底。它们都有额外的复杂性，都需要理解抽象的底层运行机制。  async 函数就是隧道尽头的亮光，很多人认为它是异步操作的终极解决方案。  async 函数就是 Generator 函数的语法糖。 |
|  | //一个 Generator 函数，依次读取两个文件。 **var fs** = require('fs'); **var readFile** = **function** (fileName){  **return new** Promise(**function** (resolve, reject){  **fs**.readFile(fileName, **function**(error, data){  **if** (error) reject(error);  resolve(data);  });  });  }; **var** gen = **function**\* (){  **var** f1 = **yield** readFile('/etc/fstab');  **var** f2 = **yield** readFile('/etc/shells');  console.log(**f1**.toString());  console.log(**f2**.toString()); }; //写成 async 函数，就是下面这样。 **var asyncReadFile** = async **function** (){  **var** f1 = await **readFile**('/etc/fstab');  **var** f2 = await **readFile**('/etc/shells');  console.log(f1.toString());  console.log(f2.toString()); }; //一比较就会发现，async 函数就是将 Generator 函数的星号(\*)替换成 async，将 yield 替换成 await，仅此而已。 |
|  | async 函数的优点  async 函数对 Generator 函数的改进，体现在以下三点。  (1)内置执行器。 Generator 函数的执行必须靠执行器，所以才有了 co 函数库，而 async 函数自带执行器。也就是说，async 函数的执行，与普通函数一模一样，只要一行。  var result = asyncReadFile();  (2)更好的语义。 async 和 await，比起星号和 yield，语义更清楚了。async 表示函数里有异步操作，await 表示紧跟在后面的表达式需要等待结果。  (3)更广的适用性。 co 函数库约定，yield 命令后面只能是 Thunk 函数或 Promise 对象，而 async 函数的 await 命令后面，可以跟 Promise 对象和原始类型的值(数值、字符串和布尔值，但这时等同于同步操作)。 |
|  | async 函数的实现  async 函数的实现，就是将 Generator 函数和自动执行器，包装在一个函数里。  async function fn(args){  // ...  }  // 等同于  function fn(args){  return spawn(function\*() {  // ...  });  }  所有的 async 函数都可以写成上面的第二种形式，其中的 spawn 函数就是自动执行器。  下面给出 spawn 函数的实现，基本就是前文自动执行器的翻版。  function spawn(genF) {  return new Promise(function(resolve, reject) {  var gen = genF();  function step(nextF) {  try {  var next = nextF();  } catch(e) {  return reject(e);  }  if(next.done) {  return resolve(next.value);  }  Promise.resolve(next.value).then(function(v) {  step(function() { return gen.next(v); });  }, function(e) {  step(function() { return gen.throw(e); });  });  }  step(function() { return gen.next(undefined); });  });  }  async 函数是非常新的语法功能，新到都不属于 ES6，而是属于 ES7。目前，它仍处于提案阶段，但是转码器 Babel 和 regenerator 都已经支持，转码后就能使用。 |
|  | async 函数的用法  同 Generator 函数一样，async 函数返回一个 Promise 对象，可以使用 then 方法添加回调函数。当函数执行的时候，一旦遇到 await 就会先返回，等到触发的异步操作完成，再接着执行函数体内后面的语句。  下面是一个例子。  async function getStockPriceByName(name) {  var symbol = await getStockSymbol(name);  var stockPrice = await getStockPrice(symbol);  return stockPrice;  }  getStockPriceByName('goog').then(function (result){  console.log(result);  });  上面代码是一个获取股票报价的函数，函数前面的async关键字，表明该函数内部有异步操作。调用该函数时，会立即返回一个Promise对象。  下面的例子，指定多少毫秒后输出一个值。  function timeout(ms) {  return new Promise((resolve) => {  setTimeout(resolve, ms);  });  }  async function asyncPrint(value, ms) {  await timeout(ms);  console.log(value)  }  asyncPrint('hello world', 50);  上面代码指定50毫秒以后，输出"hello world"。 |
| 注意点 | await 命令后面的 Promise 对象，运行结果可能是 rejected，所以最好把 await 命令放在 try...catch 代码块中。  async function myFunction() {  try {  await somethingThatReturnsAPromise();  } catch (err) {  console.log(err);  }  }  // 另一种写法  async function myFunction() {  await somethingThatReturnsAPromise().catch(function (err){  console.log(err);  });  }  await 命令只能用在 async 函数之中，如果用在普通函数，就会报错。  async function dbFuc(db) {  let docs = [{}, {}, {}];  // 报错  docs.forEach(function (doc) {  await db.post(doc);  });  }  上面代码会报错，因为 await 用在普通函数之中了。但是，如果将 forEach 方法的参数改成 async 函数，也有问题。  async function dbFuc(db) {  let docs = [{}, {}, {}];  // 可能得到错误结果  docs.forEach(async function (doc) {  await db.post(doc);  });  }  上面代码可能不会正常工作，原因是这时三个 db.post 操作将是并发执行，也就是同时执行，而不是继发执行。正确的写法是采用 for 循环。  async function dbFuc(db) {  let docs = [{}, {}, {}];  for (let doc of docs) {  await db.post(doc);  }  }  如果确实希望多个请求并发执行，可以使用 Promise.all 方法。  async function dbFuc(db) {  let docs = [{}, {}, {}];  let promises = docs.map((doc) => db.post(doc));  let results = await Promise.all(promises);  console.log(results);  }  // 或者使用下面的写法  async function dbFuc(db) {  let docs = [{}, {}, {}];  let promises = docs.map((doc) => db.post(doc));  let results = [];  for (let promise of promises) {  results.push(await promise);  }  console.log(results);  } |

Async/Await使用案例

|  |  |
| --- | --- |
|  | <https://cnodejs.org/topic/5640b80d3a6aa72c5e0030b6>  体验异步的终极解决方案-ES7的Async/Await |
|  | 阅读本文前，期待您对promise和ES6(ECMA2015)有所了解，会更容易理解。 本文以体验为主，不会深入说明，结尾有详细的文章引用。  第一个例子  Async/Await应该是目前最简单的异步方案了，首先来看个例子。  这里我们要实现一个暂停功能，输入N毫秒，则停顿N毫秒后才继续往下执行。  var sleep = function (time) {  return new Promise(function (resolve, reject) {  setTimeout(function () {  resolve();  }, time);  })  };  var start = async function () {  // 在这里使用起来就像同步代码那样直观  console.log('start');  await sleep(3000);  console.log('end');  };  start();  控制台先输出start，稍等3秒后，输出了end。  基本规则  async 表示这是一个async函数，await只能用在这个函数里面。  await 表示在这里等待promise返回结果了，再继续执行。  await 后面跟着的应该是一个promise对象（当然，其他返回值也没关系，只是会立即执行，不过那样就没有意义了…）  获得返回值  await等待的虽然是promise对象，但不必写.then(..)，直接可以得到返回值。  var sleep = function (time) {  return new Promise(function (resolve, reject) {  setTimeout(function () {  // 返回 ‘ok’  resolve('ok');  }, time);  })  };  var start = async function () {  let result = await sleep(3000);  console.log(result); // 收到 ‘ok’  };  捕捉错误  既然.then(..)不用写了，那么.catch(..)也不用写，可以直接用标准的try catch语法捕捉错误。  var sleep = function (time) {  return new Promise(function (resolve, reject) {  setTimeout(function () {  // 模拟出错了，返回 ‘error’  reject('error');  }, time);  })  };  var start = async function () {  try {  console.log('start');  await sleep(3000); // 这里得到了一个返回错误    // 所以以下代码不会被执行了  console.log('end');  } catch (err) {  console.log(err); // 这里捕捉到错误 `error`  }  };  循环多个await  await看起来就像是同步代码，所以可以理所当然的写在for循环里，不必担心以往需要闭包才能解决的问题。  ..省略以上代码  var start = async function () {  for (var i = 1; i <= 10; i++) {  console.log(`当前是第${i}次等待..`);  await sleep(1000);  }  };  值得注意的是，await必须在async函数的上下文中的。  ..省略以上代码  let 一到十 = [1,2,3,4,5,6,7,8,9,10];  // 错误示范  一到十.forEach(function (v) {  console.log(`当前是第${v}次等待..`);  await sleep(1000); // 错误!! await只能在async函数中运行  });  // 正确示范  for(var v of 一到十) {  console.log(`当前是第${v}次等待..`);  await sleep(1000); // 正确, for循环的上下文还在async函数中  } |
|  | 第二个例子  这个例子是一个小应用，根据电影文件名，自动下载对应的海报。  直接贴出代码，就不说明了。  import fs from 'fs';import path from 'path';import request from 'request';  var movieDir = \_\_dirname + '/movies',  exts = ['.mkv', '.avi', '.mp4', '.rm', '.rmvb', '.wmv'];  // 读取文件列表  var readFiles = function () {  return new Promise(function (resolve, reject) {  fs.readdir(movieDir, function (err, files) {  resolve(files.filter((v) => exts.includes(path.parse(v).ext)));  });  });  };  // 获取海报  var getPoster = function (movieName) {  let url = `https://api.douban.com/v2/movie/search?q=${encodeURI(movieName)}`;  return new Promise(function (resolve, reject) {  request({url: url, json: true}, function (error, response, body) {  if (error) return reject(error);  resolve(body.subjects[0].images.large);  })  });  };  // 保存海报  var savePoster = function (movieName, url) {  request.get(url).pipe(fs.createWriteStream(path.join(movieDir, movieName + '.jpg')));  };  (async () => {  let files = await readFiles();  // await只能使用在原生语法  for (var file of files) {  let name = path.parse(file).name;  console.log(`正在获取【${name}】的海报`);  savePoster(name, await getPoster(name));  }  console.log('=== 获取海报完成 ===');  })();  其他信息  微软的Edge浏览器已经率先支持了async/await语法，相信不久之后chrome等浏览器、node.js也会跟进的，超期待！~(≧▽≦)/~  一些资料和工具  [文中讲到的例子的源代码 https://github.com/think2011/ES7-Async-Await-Demo](https://github.com/think2011/ES7-Async-Await-Demo" \t "https://cnodejs.org/topic/_blank)  [方便的在线babel运行环境 https://babeljs.io/repl/](https://babeljs.io/repl/" \t "https://cnodejs.org/topic/_blank)  [很详细的异步编程教程 http://es6.ruanyifeng.com/#docs/async](http://es6.ruanyifeng.com/" \l "docs/async" \t "https://cnodejs.org/topic/_blank)  [很详细的promise小书 http://liubin.github.io/promises-book/#introduction](http://liubin.github.io/promises-book/" \l "introduction" \t "https://cnodejs.org/topic/_blank) |

demo-以读取文件为例说明

|  |  |
| --- | --- |
|  | http://blog.percymong.com/2017/07/16/javascript-the-history-of-async-programming/ |
|  | 设定需求  为了能使用例子更直观地表述出 JavaScript 异步编程的变化，我们现在设定下面的一个需求。  需求有以下规则  只能使用 fs.readFile(path, [options,] callback) 来读文件  按顺序读取 5 个文件 (1.txt、2.txt、3.txt、4.txt、5.txt)  打印读取到的数据  不需要向 fs.readFile 函数传选项对象（options）  为了让代码看起来更直观，回调函数不会使用箭头函数写法  做好错误处理工作  为了方便查看读取到的数据，建议上面要被读取的 5 个文件只要有少量的唯一的内容即可。   |  | | --- | | // 我每次读完文件后打印的数据如下  1111111111  2222222222  333333333  444444444  55555555  读取完毕 ~ |   回调函数写法  不得不说，纯粹的回调函数写法拥有最正宗的回调地狱(callback hell)   |  | | --- | | // 纯粹的嵌套回调函数  fs.readFile('1.txt', function(err, data) {  if (err) throw err;  console.log(data.toString())  fs.readFile('2.txt', function(err, data) {  if (err) throw err;  console.log(data.toString())  fs.readFile('3.txt', function(err, data) {  if (err) throw err;  console.log(data.toString())  fs.readFile('4.txt', function(err, data) {  if (err) throw err;  console.log(data.toString())  fs.readFile('5.txt', function(err, data) {  if (err) throw err;  console.log(data.toString())  // 深陷回调地狱，不能自拔  console.log('读取完毕 ~');  })  })  })  })  }) |   Promise 写法  为了更方便的使用 promise 对象，我们需要把 fs.readFile 用函数封装为其对应的 Promise 版本。   |  | | --- | | // 封装 fs.readFile 为 Promise 版本  function readFilePromise(path) {  return new Promise(function(resolve, reject) {  fs.readFile(path, function(err, data) {  if (err) {  reject(err);  } else {  resolve(data);  }  })  })  } |  |  | | --- | | // 执行任务  readFilePromise('1.txt')  .then(function(data) {  console.log(data.toString())  return readFilePromise('2.txt');  })  .then(function(data) {  console.log(data.toString())  return readFilePromise('3.txt');  })  .then(function(data) {  console.log(data.toString())  return readFilePromise('4.txt');  })  .then(function(data) {  console.log(data.toString())  return readFilePromise('5.txt');  })  .then(function(data) {  console.log(data.toString())  console.log('读取完毕 ～')  })  .catch(function(err) {  console.log(err);  }) |   Promise 的链式写法的确比回调函数看起来好很多，但是放眼望去，一堆 then，难免有代码冗余之嫌，并且原来的语义也变得不是很清楚。  Generator 函数写法  利用 Generator 函数的 可暂停性 可以对一个或多个异步任务进行封装，封装后，Generator 函数就变为一个封装异步任务的容器。  Generator 函数中，每一个有异步任务的地方，都需要用 yield 语句注明，即在 yield 语句后面书写异步任务的代码。  一般地，yield 语句后可以直接跟异步代码，或者也可以跟使用某种方式封装好的异步代码（常见的封装就是 promise 封装 和 thunk 函数封装）。（使用封装写法的好处是可以使代码看起来更像同步代码）  thunk 函数封装请参考阮一峰老师的文章，我这里只专注 promise 封装   |  | | --- | | // 使用 generator 函数封装异步任务  function\* asyncTaskWrapper() {  let data = yield readFilePromise('1.txt')  console.log(data.toString())  data = yield readFilePromise('2.txt')  console.log(data.toString())  data = yield readFilePromise('3.txt')  console.log(data.toString())  data = yield readFilePromise('4.txt')  console.log(data.toString())  data = yield readFilePromise('5.txt')  console.log(data.toString())  console.log('读取完毕 ～')  } |   好了，上面的异步任务倒是封装完了，接下来就该执行了。我们先根据 generator 函数的特性手动地使用 next() 方法一步一步地执行。   |  | | --- | | // 手动执行上面的 generator 函数  var g = asyncTaskWrapper();  g.next()  .value.then(function(data) {  g.next(data)  .value.then(function(data) {  g.next(data)  .value.then(function(data) {  g.next(data)  .value.then(function(data) {  g.next(data)  .value.then(function(data) {  g.next(data)  });  });  });  });  }) |   通过 next(value) 方法向生成器函数传入的参数将作为上次 yield 语句的返回值  好像，我们又陷入回调地狱了。。。  是的，手动执行 generator 函数不仅累，并且效率低。观察上面手动执行的代码，其实我们可以写一个函数，利用递归来自动执行 generator 函数。   |  | | --- | | // 工具函数，利用递归使执行 generator 函数自动化  function run(gen) {  let g = gen();  function next(data) {  let result = g.next(data);  if (result.done) {  return result.value;  } else {  result.value.then(function(data) {  next(data)  })  }  }  next();  }  // 运行上面的工具函数，自动执行上面的 generator 函数  run(asyncTaskWrapper); |   针对上面的情况，nodejs 界的 tj 大神 写了一个工具模块 [co](https://github.com/tj/co" \t "http://blog.percymong.com/2017/07/16/javascript-the-history-of-async-programming/_blank)， 专门用来自动执行 Generator 函数。  <https://github.com/tj/co>  细节可参看文首阮一峰老师相关的文章   |  | | --- | | // 借助 co 模块，自动执行上面的 generator 函数  // 使用 npm install co 来安装模块  const co = require('co');  co(asyncTaskWrapper) |   异步函数写法  异步函数使用 async/await 语法，其解决上面的需求的代码如下。   |  | | --- | | async function asyncTaskWrapper() {  let data = await readFilePromise('1.txt')  console.log(data.toString())  data = await readFilePromise('2.txt')  console.log(data.toString())  data = await readFilePromise('3.txt')  console.log(data.toString())  data = await readFilePromise('4.txt')  console.log(data.toString())  data = await readFilePromise('5.txt')  console.log(data.toString())  console.log('读取完毕 ～')  }  asyncTaskWrapper() |   放眼望去，就如同步函数写法一样。拿上面的代码与上一节 generator 函数封装异步任务的代码相比，颇为相似，只不过 async 函数写法就是将 Generator 函数的星号（\*）替换成 async，将 yield 替换成 await。  可以说，async 函数就是 Generator 函数的语法糖。只不过，相对 generator 函数，async 函数还是有所改进的  Generator 函数的执行必须靠执行器，所以才有了 co 函数库，而 async 函数自带执行器。也就是说，async 函数的执行，与普通函数一模一样，只要一行。  更好的语义  更广的适用性。 co 函数库约定，yield 命令后面只能是 Thunk 函数或 Promise 对象，而 async 函数的 await 命令后面，可以跟 Promise 对象和原始类型的值（数值、字符串和布尔值，但这时等同于同步操作）。 |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |