--

[jquery deferred 1](#_Toc21252)

[1 deferred 1](#_Toc8458)

[2 progress 1](#_Toc3017)

[3 then 1](#_Toc16073)

[4 pipe 2](#_Toc14079)

[5 与Promise 2](#_Toc6634)

[6 .promise() 2](#_Toc20721)

[7 $.when() 2](#_Toc1337)

[8 使用实例 3](#_Toc12858)

[使用 jQuery Deferred 和 Promise 创建响应式应用程序 5](#_Toc3033)

[异步缓存 5](#_Toc25220)

[通用异步缓存 5](#_Toc25499)

[图片加载 5](#_Toc513)

[定时 6](#_Toc28417)

[同步多个动画 6](#_Toc2006)

[一次性事件 6](#_Toc13445)

[组合助手 7](#_Toc17630)

[结论 7](#_Toc7380)

jquery deferred

|  |  |
| --- | --- |
|  |  |
| 1 deferred | 一、什么是deferred对象？  开发网站的过程中，我们经常遇到某些耗时很长的javascript操作。其中，既有异步的操作(比如ajax读取服务器数据)，也有同步的操作(比如遍历一个大型数组)，它们都不是立即能得到结果的。  通常的做法是，为它们指定回调函数(callback)。即事先规定，一旦它们运行结束，应该调用哪些函数。  但是，在回调函数方面，jQuery的功能非常弱。为了改变这一点，jQuery开发团队就设计了deferred对象。  简单说，deferred对象就是jQuery的回调函数解决方案。在英语中，defer的意思是"延迟"，所以deferred对象的含义就是"延迟"到未来某个点再执行。  它解决了如何处理耗时操作的问题，对那些操作提供了更好的控制，以及统一的编程接口。它的主要功能，可以归结为四点。下面我们通过示例代码，一步步来学习。 |
|  | deferred对象代表了将要完成的某种操作，并提供了一些方法，帮助用户使用。它是jQuery对Promises接口的实现。jQuery的所有Ajax操作函数，默认返回的就是一个deferred对象。  简单说，Promises是异步操作的通用接口，扮演代理人（proxy）的角色，将异步操作包装成具有同步操作特性的特殊对象。异步操作的典型例子就是Ajax操作、网页动画、web worker等等。  由于JavaScript单线程的特点，如果某个操作耗时很长，其他操作就必需排队等待。为了避免整个程序失去响应，通常的解决方法是将那些排在后面的操作，写成“回调函数”（callback）的形式。这样做虽然可以解决问题，但是有一些显著缺点：  > 回调函数往往写成函数参数的形式，形成所谓的“持续传递风格”（即参数就是下一步操作，Continuation-passing style），导致函数的输入和输出非常混乱，整个程序的可阅读性差；  > 回调函数往往只能指定一个，如果有多个操作，就需要改写回调函数。  > 除了正常的报错机制，错误还可能通过回调函数的形式返回，增加了除错和调试的难度。  > 正常的函数输入和输出可以区分得很清楚，回调函数使得函数的输出不再重要。  Promises就是为了解决这些问题而提出的，它的主要目的就是取代回调函数，成为非同步操作的解决方案。它的核心思想就是让非同步操作返回一个对象，其他操作都针对这个对象来完成。比如，假定ajax操作返回一个Promise对象。  var promise = get('http://www.example.com');  然后，Promise对象有一个then方法，可以用来指定回调函数。一旦非同步操作完成，就调用指定的回调函数。  promise.then(function (content) {  console.log(content)  }) |
|  | deferred对象的基本用法  （1）生成deferred对象  第一步是通过$.Deferred()方法，生成一个deferred对象。  var deferred = $.Deferred();  （2）deferred对象的状态  deferred对象有三种状态:  > pending：表示操作还没有完成。  > resolved：表示操作成功。  > rejected：表示操作失败。  state方法用来返回deferred对象当前状态：  $.Deferred().state() // 'pending'  $.Deferred().resolve().state() // 'resolved'  $.Deferred().reject().state() // 'rejected'  （3）改变状态的方法  resolve方法将deferred对象的状态从pending改为resolved，reject方法则将状态从pending改为rejected。  var deferred = $.Deferred();  deferred.resolve("hello world");  resolve方法的参数，用来传递给回调函数。  （4）绑定回调函数  deferred对象在状态改变时，会触发回调函数。  done方法指定状态变为resolved（操作成功）时的回调函数；  fail方法指定状态变为rejected（操作失败）时的回调函数；  always方法指定，不管状态变为resolved或rejected，都会触发的方法。  var deferred = $.Deferred();  deferred.done(function(value) {  console.log(value);  }).resolve('hello world');  // hello world  上述三种方法都返回的原有的deferred对象，因此可以采用链式写法，在后面再链接别的方法（包括done和fail在内）。  $.Deferred().done(f1).fail(f2).always(f3); |
| 2 progress | notify() 和 progress()  progress()用来指定一个回调函数，当调用notify()方法时，该回调函数将执行。它的用意是提供一个接口，使得在非同步操作执行过程中，可以执行某些操作，比如定期返回进度条的进度。  var userProgress = $.Deferred();  var $profileFields = $("input");  var totalFields = $profileFields.length  userProgress.progress(function (filledFields) {  var pctComplete = (filledFields / totalFields) \* 100;  $("#progress").html(pctComplete.toFixed(0));  });  userProgress.done(function () {  $("#thanks").html("Thanks for completing your profile!").show();  });  $("input").on("change", function () {  var filledFields = $profileFields.filter("[value!='']").length;  userProgress.notify(filledFields);  if (filledFields == totalFields) {  userProgress.resolve();  }  }); |
| 3 then | then方法  （1）概述  then方法的作用也是指定回调函数，它可以接受三个参数，也就是三个回调函数。第一个参数是resolve时调用的回调函数（相当于done方法），第二个参数是reject时调用的回调函数（相当于fail方法），第三个参数是progress()方法调用的回调函数。  deferred.then( doneFilter [, failFilter ] [, progressFilter ] )  （2）返回值  在jQuery 1.8之前，then()只是.done().fail()写法的语法糖，两种写法是等价的。在jQuery 1.8之后，then()返回一个新的promise对象，而done()返回的是原有的deferred对象。如果then()指定的回调函数有返回值，该返回值会作为参数，传入后面的回调函数。  var defer = jQuery.Deferred();  defer.done(function(a,b){  return a \* b;  }).done(function( result ) {  console.log("result = " + result);  }).then(function( a, b ) {  return a \* b;  }).done(function( result ) {  console.log("result = " + result);  }).then(function( a, b ) {  return a \* b;  }).done(function( result ) {  console.log("result = " + result);  });  defer.resolve( 2, 3 );  在jQuery 1.8版本之前，上面代码的结果是：  result = 2  result = 2  result = 2  在jQuery 1.8版本之后，返回结果是  result = 2  result = 6  result = NaN  这一点需要特别引起注意。  $.ajax( url1, { dataType: "json" } )  .then(function( data ) {  return $.ajax( url2, { data: { user: data.userId } } );  }).done(function( data ) {  // 从url2获取的数据  });  上面代码最后那个done方法，处理的是从url2获取的数据，而不是从url1获取的数据。  （3）对返回值的修改  利用then()会修改返回值这个特性，我们可以在调用其他回调函数之前，对前一步操作返回的值进行处理。  var post = $.post("/echo/json/").then(function (p) {  return p.firstName;  });  post.done(function (r) { console.log(r); });  上面代码先使用then()方法，从返回的数据中取出所需要的字段（firstName），所以后面的操作就可以只处理这个字段了。  有时，Ajax操作返回json字符串里面有一个error属性，表示发生错误。这个时候，传统的方法只能是通过done()来判断是否发生错误。通过then()方法，可以让deferred对象调用fail()方法。  var myDeferred = $.get('/echo/json/').then(function (response) {  if (response.error) {  return $.Deferred().reject(response);  }  return response;  }, function () {  return $.Deferred().reject({ error: true });  });  myDeferred.done(function (response) {  $("#status").html("Success!");  }).fail(function (response) {  $("#status").html("An error occurred");  });  上面代码中，不管是通信出错，或者服务器返回一个错误，都会调用reject方法，返回一个**新的deferred对象，状态为rejected，因此就会触发fail方法指定的回调函数**。  关于error的处理，jQuery的deferred对象与其他实现Promises规范的函数库有一个重大不同。就是说，如果deferred对象执行过程中，抛出一个非Promises对象的错误，那么将不会被后继的then方法指定的rejected回调函数捕获，而会一直传播到应用程序层面。  **为了代码行为与Promises规范保持一致，建议出错时，总是使用reject方法返回错误。**  d = $.Deferred()  d.then(function () {  throw new Error('err')  }).fail(function () {  console.log('fail')  });  d.resolve();  // Error: err  上面代码中，then的回调函数抛出一个错误，按照Promises规范，应该被fail方法的回调函数捕获，但是jQuery的部署是上升到应用程序的层面。  （4）回调函数的返回值  如果回调函数返回deferred对象，则then方法的返回值将是对应这个返回值的promise对象。  var d1 = $.Deferred();  var promise = $.when('Hello').then(function(h){  return $.when(h,d1);  })  promise.done(function (s1,s2) {  console.log(s1);  console.log(s2);  })  d1.resolve('World')  // Hello  // World  上面代码中，done方法的回调函数，正常情况下只能接受一个参数。但是由于then方法的回调函数，返回一个when方法生成的deferred对象，导致它可以接受两个参数。 |
| 4 pipe | pipe方法  pipe方法接受一个函数作为参数，表示在调用then方法、done方法、fail方法、always方法指定的回调函数之前，先运行pipe方法指定的回调函数。它通常用来对服务器返回的数据做初步处理。  $(function () {  var defer = $.Deferred(),  filtered = defer.pipe(function( value ) {  return value \* 2;  });    filtered.done(function( value ) {  alert( "值是 ( 2\*5 = ) 10: " + value );  });  defer.resolve( 5 );  })  deferred.pipe() 函数用于过滤 and/or 链式延迟对象的工具方法。  注意：从jQuery 1.8开始, deferred.pipe() 方法过时。使用 deferred.then() 代替它。 |
| 5 与Promise | 与Promise A+规格的差异  Promise事实上的标准是社区提出的Promise A+规格，jQuery的实现并不完全符合Promise A+，主要是对错误的处理。  var promise2 = promise1.then(function () {  throw new Error("boom!");  });  上面代码在回调函数中抛出一个错误，Promise A+规定此时Promise实例的状态变为reject，该错误被下一个catch方法指定的回调函数捕获。但是，jQuery的Deferred对象此时不会改变状态，亦不会触发回调函数，该错误一般情况下会被window.onerror捕获。换句话说，在Deferred对象中，总是必须使用reject方法来改变状态。 |
| 6 .promise() | promise对象  （1）概念  一般情况下，从外部改变第三方完成的异步操作（比如Ajax）的状态是毫无意义的。为了防止用户这样做，可以在deferred对象的基础上，返回一个针对它的promise对象。  简单说，promise对象就是不能改变状态的deferred对象，也就是deferred的只读版。或者更通俗地理解成，promise是一个对将要完成的任务的承诺，排除了其他人破坏这个承诺的可能性，只能等待承诺方给出结果。  你可以通过promise对象，为原始的deferred对象添加回调函数，查询它的状态，但是无法改变它的状态，也就是说promise对象不允许你调用resolve和reject方法。  （2）生成promise对象  deferred对象的promise方法，用来生成对应的promise对象。  function getPromise(){  return $.Deferred().promise();  }  try{  getPromise().resolve("a");  } catch(err) {  console.log(err);  }  // TypeError  上面代码对promise对象，调用resolve方法，结果报错。  jQuery的ajax()方法返回的就是一个Promise对象。此外，Animation类操作也可以使用promise方法。  $('body').toggle('blinds').promise().then(  function(){  $('body').toggle('blinds')  }  ) |
|  |  |
| 7 $.when() | $.when()方法  $.when()接受多个deferred对象作为参数，当它们全部运行成功后，才调用resolved状态的回调函数，但只要其中有一个失败，就调用rejected状态的回调函数。它相当于将多个非同步操作，合并成一个。实质上，when方法为多个deferred对象，返回一个单一的promise对象。  deferred对象的另一大好处，就是它允许你为多个事件指定一个回调函数，这是传统写法做不到的。用到了一个新的方法$.when()。  $.when(  $.ajax("/main.php"),  $.ajax("/modules.php"),  $.ajax("/lists.php")  ).then(function (resp1, resp2, resp3) {  console.log(resp1);  console.log(resp2);  console.log(resp3);  }, failureFunc);  这段代码的意思是，上面代码表示，要等到三个ajax操作都结束以后，才执行then方法指定的回调函数。 三个都成功了就运行successFunc指定的回调函数； 如果有一个失败或都失败了，就执行failureFunc指定的回调函数；  when方法里面要执行多少个操作，回调函数就有多少个参数，对应前面每一个操作的返回结果。上面代码的回调函数有三个参数，resp1、resp2和resp3，依次对应前面三个ajax操作的返回结果。  如果when方法的参数不是deferred或promise对象，则直接作为回调函数的参数。 |
|  | d = $.Deferred()  $.when(d, 'World').done(function (s1, s2){  console.log(s1);  console.log(s2);  })  d.resolve('Hello')  // Hello  // World  上面代码中，when的第二个参数是一个字符串，则直接作为回调函数的第二个参数。  此外，如果when方法的参数都不是deferred或promise对象，那么when方法的回调函数将立即运行。 |
|  |  |
| 8 使用实例 | 1. wait方法   我们可以用deferred对象写一个wait方法，表示等待多少毫秒后再执行：  $.wait = function (time) {  return $.Deferred(function (dfd) {  setTimeout(dfd.resolve, time);  });  };  $.wait(5000).then(function () {  console.log("Hello from the future!");  });  在上面的wait方法的基础上，还可以改写setTimeout方法，让其返回一个deferred对象。  function doSomethingLater(fn, time) {  var dfd = $.Deferred();  setTimeout(function() {  dfd.resolve(fn());  }, time || 0);  return dfd.promise();  }  var promise = doSomethingLater(function (){  console.log( '已经延迟执行' );  }, 100); |
| 1. 自定义操作使用deferred接口   我们可以利用deferred接口，使得任意操作都可以用done()和fail()指定回调函数：  var Twitter = {  search: function (query) {  var dfd = $.Deferred();  $.ajax({  url: "http://search.twitter.com/search.json",  data: { q: query },  dataType: 'jsonp',  success: dfd.resolve  });  return dfd.promise();  }  }  Twitter.search('javaScript').then(function (data) {  alert(data.results[0].text);  });  deferred对象的另一个优势是可以附加多个回调函数。  下面的例子使用了上面所改写的setTimeout函数。  function doSomething(arg) {  var dfd = $.Deferred();  setTimeout(function () {  dfd.reject("Sorry, something went wrong.");  });  return dfd;  }  doSomething("uh oh").done(function () {  console.log("Won't happen, we're erroring here!");  }).fail(function (message) {  console.log(message);  }); |
|  | 二、ajax操作的链式写法  首先，回顾一下jQuery的ajax操作的传统写法：  $.ajax({  url: "test.html",  success: function () {  log("哈哈，成功了！");  },  error: function () {  log("出错啦！");  } });  在上面的代码中，$.ajax()接受一个对象参数，这个对象包含两个方法：success方法指定操作成功后的回调函数，error方法指定操作失败后的回调函数。  $.ajax()操作完成后，如果使用的是低于1.5.0版本的jQuery，返回的是XHR对象，你没法进行链式操作；如果高于1.5.0版本，返回的是deferred对象，可以进行链式操作。  现在，新的写法是这样的：  $.ajax("test.html").done(function () {  log("哈哈，成功了！"); }).fail(function () {  log("出错啦！"); });  可以看到，[done()](http://api.jquery.com/deferred.done/" \t "http://www.ruanyifeng.com/blog/2011/08/_blank)相当于success方法，[fail()](http://api.jquery.com/deferred.fail/" \t "http://www.ruanyifeng.com/blog/2011/08/_blank)相当于error方法。采用链式写法以后，代码的可读性大大提高。 |
|  | 三、指定同一操作的多个回调函数  deferred对象的一大好处，就是它允许你自由添加多个回调函数。  还是以上面的代码为例，如果ajax操作成功后，除了原来的回调函数，我还想再运行一个回调函数，怎么办？  很简单，直接把它加在后面就行了。  $.ajax("test.html").done(function () {  log("哈哈，成功了！"); }).fail(function () {  log("出错啦！"); }).done(function () {  log("第二个回调函数！"); });  回调函数可以添加任意多个，它们按照添加顺序执行。 |
|  | 四、为多个操作指定回调函数 |
|  | 五、普通操作的回调函数接口(上)  deferred对象的最大优点，就是它把这一套回调函数接口，从ajax操作扩展到了所有操作。也就是说，任何一个操作----不管是ajax操作还是本地操作，也不管是异步操作还是同步操作----都可以使用deferred对象的各种方法，指定回调函数。 |
| **/\*  \* 假定有一个很耗时的操作wait，我们为它指定回调函数，应该怎么做呢？  \* 很自然的，你会想到，可以使用$.when()：  \* \*/ var wait** = **function**(){  **var tasks** = **function**(){  log("执行完毕！");  };  setTimeout(**tasks**,3000); };  $.when(**wait**()).done(**function** () {  log("哈哈，成功了！"); }).fail(**function** () {  log("出错啦！"); });  **/\*  \* 但是，这样写的话，done()方法会立即执行，起不到回调函数的作用:  \* 原因在于$.when()的参数只能是deferred对象，所以必须对wait()进行改写；  \* 使wait()函数返回的是deferred对象，这就可以加上链式操作了。  \* \*/ var wait** = **function** (waitDef) {  **var tasks** = **function** () {  **log**("执行完毕！");  waitDef.resolve();// 改变deferred对象的执行状态  };  setTimeout(**tasks**, 5000);  **return** waitDef; }; **var waitDef** = $.Deferred(); // 新建一个deferred对象 $.when(**wait**(**waitDef**)).done(**function** () {  **log**("哈哈，成功了！"); }).fail(**function** () {  **log**("出错啦！"); }); //wait()函数运行完，就会自动运行done()方法指定的回调函数。 |
|  |
|  | 六、deferred.resolve()方法和deferred.reject()方法  如果仔细看，你会发现在上面的wait()函数中，还有一个地方我没讲解。那就是[dtd.resolve()](http://api.jquery.com/deferred.resolve" \t "http://www.ruanyifeng.com/blog/2011/08/_blank)的作用是什么？  要说清楚这个问题，就要引入一个新概念"执行状态"。jQuery规定，deferred对象有三种执行状态:[未完成,已完成,已失败]  如果执行状态是"已完成"(resolve),deferred对象立刻调用done()方法指定的回调函数；  如果执行状态是"已失败"(reject)，调用fail()方法指定的回调函数；  如果执行状态是"未完成"，则继续等待，或者调用[progress()](http://api.jquery.com/deferred.progress/" \t "http://www.ruanyifeng.com/blog/2011/08/_blank)方法指定的回调函数(jQuery1.7版本添加)。  前面部分的ajax操作时，deferred对象会根据返回结果，自动改变自身的执行状态；但是，在wait()函数中，这个执行状态必须由程序员手动指定。dtd.resolve()的意思是，将dtd对象的执行状态从"未完成"改为"已完成"，从而触发done()方法。  类似的，还存在一个[deferred.reject()](http://api.jquery.com/deferred.reject" \t "http://www.ruanyifeng.com/blog/2011/08/_blank)方法，作用是将dtd对象的执行状态从"未完成"改为"已失败"，从而触发fail()方法。  var dtd = $.Deferred(); // 新建一个Deferred对象 var wait = function (dtd) {  var tasks = function () {  log("执行完毕！");  dtd.reject(); // 改变Deferred对象的执行状态  };  setTimeout(tasks, 5000);  return dtd; }; $.when(wait(dtd)).done(function () {  log("哈哈，成功了！"); }).fail(function () {  log("出错啦！"); });  结果：  执行完毕！  出错啦！ |
|  | 七、deferred.promise()方法  上面这种写法，还是有问题。那就是dtd是一个全局对象，所以它的执行状态可以从外部改变。  请看下面的代码：  var dtd = $.Deferred(); // 新建一个Deferred对象 var wait = function(dtd){  var tasks = function(){  log("执行完毕！");  dtd.resolve(); // 改变Deferred对象的执行状态  };  setTimeout(tasks,5000);  return dtd; }; $.when(wait(dtd)).done(function () {  log("哈哈，成功了！"); }).fail(function () {  log("出错啦！"); }); dtd.resolve();  我在代码的尾部加了一行dtd.resolve()，这就改变了dtd对象的执行状态，因此导致done()方法立刻执行，输出"哈哈，成功了！"，等5秒之后再输出"执行完毕！"。  **为了避免这种情况，jQuery提供了[deferred.promise()](http://api.jquery.com/deferred.promise/" \t "http://www.ruanyifeng.com/blog/2011/08/_blank)方法。它的作用是，在原来的deferred对象上返回另一个deferred对象，后者只开放与改变执行状态无关的方法(比如done()方法和fail()方法)，屏蔽与改变执行状态有关的方法(比如resolve()方法和reject()方法)，从而使得执行状态不能被改变。**  请看下面的代码：  var wait = function(dtd){  var tasks = function(){  log("执行完毕！");  dtd.resolve(); // 改变Deferred对象的执行状态  };   setTimeout(tasks,5000);  return dtd.promise(); // 返回promise对象 };  var dtd = $.Deferred(); // 新建一个Deferred对象 var d = wait(dtd); // 新建一个d对象，改为对这个对象进行操作 $.when(d).done(function () {  log("哈哈，成功了！"); }).fail(function () {  log("出错啦！"); }); d.resolve(); // 此时，这个语句是无效的：Uncaught TypeError: d.resolve is not a function  在上面的这段代码中，wait()函数返回的是promise对象。然后，我们把回调函数绑定在这个对象上面，而不是原来的deferred对象上面。这样的好处是，无法改变这个对象的执行状态，要想改变执行状态，只能操作原来的deferred对象。  不过，更好的写法是[allenm](http://blog.allenm.me/2012/01/jquery_deferred_promise_method/" \t "http://www.ruanyifeng.com/blog/2011/08/_blank)所指出的，将dtd对象变成wait()函数的内部对象。  var wait = function () {  var dtd = $.Deferred(); //在函数内部，新建一个Deferred对象  var tasks = function () {  log("执行完毕！");  dtd.resolve(); // 改变Deferred对象的执行状态  };   setTimeout(tasks, 5000);  return dtd.promise(); // 返回promise对象 };  var d=wait(); $.when(d).done(function () {  log("哈哈，成功了！"); }).fail(function () {  log("出错啦！"); }); |
|  | 八、普通操作的回调函数接口(中)  另一种防止执行状态被外部改变的方法，是使用deferred对象的建构函数$.Deferred()。  jQuery规定，$.Deferred()可以接受一个函数作为参数，$.Deferred()所生成的deferred对象将作为这个函数的默认参数。  一个构造函数，返回一个链式实用对象方法来注册多个回调，回调队列， 调用回调队列，并转达任何同步或异步函数的成功或失败状态。  添加的版本: 1.5  jQuery.Deferred( [beforeStart ] )  beforeStart: Function( Deferred deferred )  一个构造函数返回之前调用的函数。  jQuery.Deferred()构造函数创建一个新的Deferred(延迟)对象。new 运算是可选的。  jQuery.Deferred方法可以传递一个可选的函数， 这个函数在构造函数返回之前调用，并且传递构建 deferred(延迟)对象作为 this对象作为第一个参数传递给函数。被调用的函数可以使用 deferred.then()附加回调。  这时，wait函数还是保持不变，我们直接把它传入$.Deferred()：  var wait = function (dtd) {  var tasks = function () {  log("执行完毕！");  dtd.resolve(); // 改变Deferred对象的执行状态  };  setTimeout(tasks, 5000); }; $.Deferred(wait).done(function () {  log("哈哈，成功了！"); }).fail(function () {  log("出错啦！"); });  $.Deferred(function (dtd) {  var tasks = function () {  log("执行完毕！");  dtd.resolve(); // 改变Deferred对象的执行状态  };  setTimeout(tasks, 5000); }).done(function () {  log("哈哈，成功了！"); }).fail(function () {  log("出错啦！"); }); |
|  | 九、普通操作的回调函数接口(下)  除了上面两种方法以外，我们还可以直接在wait对象上部署deferred接口。  var doSth = function (dtd) {  log("开始执行...");  setTimeout(function () {  log("执行完毕！");  dtd.resolve(); // 改变Deferred对象的执行状态  }, 5000); };  var dtd = $.Deferred(); // 生成Deferred对象 dtd.promise(doSth); doSth.done(function () {  log("哈哈，成功了！"); }).fail(function () {  log("出错啦！"); });  console.log("doSth(dtd);"); doSth(dtd);  这里的关键是dtd.promise(wait)这一行，它的作用就是在wait对象上部署Deferred接口。正是因为有了这一行，后面才能直接在wait上面调用done()和fail()。 |
|  | 十、小结：deferred对象的方法  (致谢：本文第一稿发表后，[allenm](http://blog.allenm.me/" \t "http://www.ruanyifeng.com/blog/2011/08/_blank)来信指出原文对promise()的理解是错的。现在的第二稿是根据[他的文章](http://blog.allenm.me/2012/01/jquery_deferred_promise_method/" \t "http://www.ruanyifeng.com/blog/2011/08/_blank)修改的，在此我表示衷心感谢。)  (完) |

[使用 jQuery Deferred 和 Promise 创建响应式应用程序](http://www.cnblogs.com/lhb25/archive/2013/03/05/jquery-deferred-promise.html)

|  |  |
| --- | --- |
|  | [使用 jQuery Deferred 和 Promise 创建响应式应用程序](http://www.cnblogs.com/lhb25/archive/2013/03/05/jquery-deferred-promise.html)  <http://www.cnblogs.com/lhb25/archive/2013/03/05/jquery-deferred-promise.html> |
|  | jQuery延时实现的核心是jQuery.Deferred：一个可以链式调用的构造函数。  需要注意的是任何deferred对象的默认状态是unresolved， 回调会通过 .then() 或 .fail()方法添加到队列，并在稍后的过程中被执行。  下面这个$.when() 接受多个参数的例子  function successFunc(){ console.log("success!"); }  function failureFunc(){ console.log("failure!"); }  $.when(  $.ajax( "/main.php" ),  $.ajax( "/modules.php" ),  $.ajax("/lists.php")  ).then( successFunc, failureFunc );  在$.when() 的实现中有趣的是，它并非仅能解析deferred对象，还可以传递不是deferred对象的参数，在处理的时候会把它们当做deferred对象并立 即执行回调(doneCallbacks)。 这也是jQuery的Deferred实现中值得一提的地方，此外，deferred.then()还为deferred.done和 deferred.fail()方法在deferred的队列中增加回调提供支持。  利用前面介绍的表中提到的deferred功能，我们来看一个代码示例：  在这里，我们创建一个非常基本的应用程序：通过$.get方法(返回一个promise)获取一条外部新闻源(1)并且(2)获取最新的一条回复。  同时程序还通过函数(prepareInterface())实现新闻和回复内容显示容器的动画。  为了确保在执行其他相关行为前，上面的这三个步骤确保完成，我们使用$.when()。  根据您的需要 .then()和.fail() 处理函数可以被用来执行其他程序逻辑。   |  | | --- | | **function getLatestNews**() {  **return** $.get("latestNews.php", **function** (data) {  console.log("news data received");  $(".news").html(data);  }); } **function getLatestReactions**() {  **return** $.get("latestReactions.php", **function** (data) {  console.log("reactions data received");  $(".reactions").html(data);  }); }  **function prepareInterface**() {  **return** $.Deferred(**function** (dfd) {  **var** latest = $(".news, .reactions");  latest.slideDown(500, dfd.resolve);  latest.addClass("active");  }).promise(); }  $.when(**getLatestNews**(), **getLatestReactions**(), **prepareInterface**()).then(**function** () {  console.log("fire after requests succeed"); }).fail(**function** () {  console.log("something went wrong!"); }); | | var d1 = $.Deferred();  var d2 = $.Deferred();  var d3 = $.Deferred();  $.when(d1, d2, d3).done(function (v1, v2, v3) {  console.log(v1); // v1 is undefined  console.log(v2); // v2 is "abc"  console.log(v3); // v3 is an array [ 1, 2, 3, 4, 5 ]  });  setTimeout(function () {  console.log("d1 do");  d1.resolve();  }, 3000)  d2.resolve("abc");  d3.resolve(1, 2, 3, 4, 5); |   deferreds在ajax的幕后操作中使用并不意味着它们无法在别处使用。 在本节中，我们将看到在一些解决方案中，使用deferreds将有助于抽象掉异步的行为，并解耦我们的代码。 |
| 异步缓存 | 当涉及到异步任务，缓存可以是一个有点苛刻的，因为你必须确保对于同一个key任务仅执行一次。因此，代码需要以某种方式跟踪入站任务。 例如下面的代码片段：  $.cachedGetScript( url, callback1 );  $.cachedGetScript( url, callback2 );  缓存机制需要确保 脚本不管是否已经存在于缓存，只能被请求一次。 因此，为了缓存系统可以正确地处理请求,我们最终需要写出一些逻辑来跟踪绑定到给定url上的回调。  值得庆幸的是，这恰好是deferred所实现的那种逻辑，因此我们可以这样来做：  var cachedScriptPromises = {};  $.cachedGetScript = function( url, callback ) {  if ( !cachedScriptPromises[ url ] ) {  cachedScriptPromises[ url ] = $.Deferred(function( defer ) {  $.getScript( url ).then( defer.resolve, defer.reject );  }).promise();  }  return cachedScriptPromises[ url ].done( callback );  };  代码相当简单：我们为每一个url缓存一个promise对象。 如果给定的url没有promise，我们创建一个deferred，并发出请求。 如果它已经存在我们只需要为它绑定回调。 该解决方案的一大优势是，它会透明地处理新的和缓存过的请求。 另一个优点是一个基于deferred的缓存 会优雅地处理失败情况。 当promise以'rejected’状态结束的话，我们可以提供一个错误回调来测试：  $.cachedGetScript( url ).then( successCallback, errorCallback );  请记住：无论请求是否缓存过，上面的代码段都会正常运作！ |
| 通用异步缓存 | 为了使代码尽可能的通用，我们建立一个缓存工厂并抽象出实际需要执行的任务​​：  $.createCache = function( requestFunction ) {  var cache = {};  return function( key, callback ) {  if ( !cache[ key ] ) {  cache[ key ] = $.Deferred(function( defer ) {  requestFunction( defer, key );  }).promise();  }  return cache[ key ].done( callback );  };  }  现在具体的请求逻辑已经抽象出来，我们可以重新写cachedGetScript：  $.cachedGetScript = $.createCache(function( defer, url ) {  $.getScript( url ).then( defer.resolve, defer.reject );  });  每次调用createCache将创建一个新的缓存库，并返回一个新的高速缓存检索函数。现在，我们拥有了一个通用的缓存工厂，它很容易实现涉及从缓存中取值的逻辑场景。 |
| 图片加载 | //另一个候选场景是图像加载：确保我们不加载同一个图像两次，我们可能需要加载图像。  **//** 使用createCache很容易实现：  $.loadImage = $.createCache(**function**( defer, url ) {  **var** image = **new** Image();  **function cleanUp**() {  image.onload = image.onerror = **null**;  }  defer.then( **cleanUp**, **cleanUp** );  image.onload = **function**() {  defer.resolve( url );  };  image.onerror = defer.**reject**;  image.src = url; }); //接下来的代码片段如下：无论image.png是否已经被加载，或者正在加载过程中，缓存都会正常工作。 $.loadImage( "my-image.png" ).done( callback1 ); $.loadImage( "my-image.png" ).done( callback2 ); |
|  | **/\*缓存数据的API响应\*/** //哪些你的页面的生命周期过程中被认为是不可变的API请求，也是缓存完美的候选场景。 //比如，执行以下操作：  $.searchTwitter = $.createCache(**function**( defer, query ) {  $.ajax({  url: "http://search.twitter.com/search.json",  data: { q: query },  dataType: "jsonp",  success: defer.resolve,  error: defer.**reject** }); }); //程序允许你在Twitter上进行搜索，同时缓存它们： $.searchTwitter( "jQuery Deferred", callback1 ); $.searchTwitter( "jQuery Deferred", callback2 ); |
| 定时 | //基于deferred的缓存并不限定于网络请求;它也可以被用于定时目的。  //例如，您可能需要在网页上给定一段时间后执行一个动作，来吸引用户对某个不容易引起注意的特定功能的关注或处理一个延时问题。 虽然setTimeout适合大多数用例，但在计时器出发后甚至理论上过期后就无法提供解决办法。 我们可以使用以下的缓存系统来处理：  **var readyTime**; $(**function**() { **readyTime** = jQuery.now(); }); $.afterDOMReady = $.createCache(**function**( defer, delay ) {  delay = delay || 0;  $(**function**() {  **var** delta = $.now() - **readyTime**;  **if** ( delta >= delay ) { defer.resolve(); }  **else** {  setTimeout( defer.resolve, delay - delta );  }  }); }); //新的afterDOMReady辅助方法用最少的计数器提供了domReady后的适当时机。 如果延迟已经过期，回调会被马上执行。 |
| 同步多个动画 | //动画是另一个常见的异步任务范例。 然而在几个不相关的动画完成后执行代码仍然有点挑战性。尽管在jQuery1.6中才提供了在动画元素上取得promise对象的功能，但它是很容易的手动实现：  $.fn.animatePromise = **function**( prop, speed, easing, callback ) {  **var** elements = **this**;  **return** $.Deferred(**function**( defer ) {  elements.animate( prop, speed, easing, **function**() {  defer.resolve();  **if** ( callback ) {  callback.apply( **this**, arguments );  }  });  }).promise(); }; //然后，我们可以使用$.when()同步化不同的动画：  **var fadeDiv1Out** = $( "#div1" ).animatePromise({ opacity: 0 }),  **fadeDiv2In** = $( "#div1" ).animatePromise({ opacity: 1 }, "fast" );  $.when( **fadeDiv1Out**, **fadeDiv2In** ).done(**function**() {  **/\* both animations ended \*/** });  //我们也可以使用同样的技巧，建立了一些辅助方法： $.each(["slideDown", "slideUp", "slideToggle", "fadeIn", "fadeOut", "fadeToggle"],  **function** (\_, name) {  $.fn[name + "Promise"] = **function** (speed, easing, callback) {  **var** elements = **this**;  **return** $.Deferred(**function** (defer) {  elements[name](speed, easing, **function** () {  defer.resolve();  **if** (callback) {  callback.apply(**this**, arguments);  }  });  }).promise();  };  } );  //然后想下面这样使用新的助手代码来同步动画：  $.when($("#div1").fadeOutPromise(), $("#div2").fadeInPromise("fast")).done(**function** () { **/\* both animations are done \*/** }); |
| 一次性事件 | //虽然jQuery提供你可能需要的所有的时间绑定方法，但当事件仅需要处理一次时，情况可能会变得有点棘手。( 与$.one() 不同 )  //例如，您可能希望有一个按钮，当它第一次被点击时打开一个面板，面板打开之后，执行特定的初始化逻辑。 在处理这种情况时，人们通常会这样写代码： **var buttonClicked** = **false**; $( "#myButton" ).click(**function**() {  **if** ( !**buttonClicked** ) {  **buttonClicked** = **true**;  initializeData();  showPanel();  } }); //不久后，你可能会在面板打开之后点击按钮时添加一些操作，如下：  **if** (**buttonClicked**) { **/\* perform specific action \*/** } //这是一个非常耦合的解决办法。 如果你想添加一些其他的操作，你必须编辑绑定代码或拷贝一份。 如果你不这样做，你唯一的选择是测试buttonClicked。由于buttonClicked可能是false，新的代码可能永远不会被执行，因此你 可能会失去这个新的动作。  //使用deferreds我们可以做的更好 (为简化起见，下面的代码将只适用于一个单一的元素和一个单一的事件类型，但它可以很容易地扩展为多个事件类型的集合)：  $.fn.bindOnce = **function**( event, callback ) {  **var** element = $( **this**[ 0 ] ),  defer = element.data( "bind\_once\_defer\_" + event );  **if** ( !defer ) {  defer = $.Deferred();  **function deferCallback**() {  element.unbind( event, **deferCallback** );  defer.resolveWith( **this**, arguments );  }  element.bind( event, **deferCallback** )  element.data( "bind\_once\_defer\_" + event , defer );  }  **return** defer.done( callback ).promise(); }; **/\* 该代码的工作原理如下：   检查该元素是否已经绑定了一个给定事件的deferred对象  如果没有，创建它，使它在触发该事件的第一时间解决  然后在deferred上绑定给定的回调并返回promise  代码虽然很冗长，但它会简化相关问题的处理。 让我们先定义一个辅助方法： \*/** $.fn.firstClick = **function**( callback ) {  **return this**.bindOnce( "click", callback ); }; //然后，之前的逻辑可以重构如下： **var openPanel** = $( "#myButton" ).firstClick(); **openPanel**.done( initializeData ); **openPanel**.done( showPanel ); //如果我们需要执行一些动作，只有当面板打开以后，所有我们需要的是这样的： **openPanel**.done(**function**() { **/\* perform specific action \*/** }); //如果面板没有打开，行动将得到延迟到单击该按钮时。 |
| 组合助手 | //单独看以上每个例子，promise的作用是有限的 。 然而，promise真正的力量是把它们混合在一起。 **/\* \* 在第一次点击时加载面板内容并打开面板 \* 假如，我们有一个按钮，可以打开一个面板，请求其内容然后淡入内容。使用我们前面定义的助手方法，我们可以这样做： \* \*/ var panel** = $( "#myPanel" ); **panel**.firstClick(**function**() {  $.when(  $.get( "panel.html" ),  **panel**.slideDownPromise()  ).done(**function**( ajaxResponse ) {  **panel**.html( ajaxResponse[ 0 ] ).fadeIn();  }); });  **/\* 在第一次点击时载入图像并打开面板 \*/** //假如，我们已经的面板有内容，但我们只希望当第一次单击按钮时加载图像并且当所有图像加载成功后淡入图像。HTML代码如下： <**div** id="myPanel">  <img **data**-**src**="image1.png" />  <img **data**-**src**="image2.png" />  <img **data**-**src**="image3.png" />  <img **data**-**src**="image4.png" /> </**div**> //我们使用data-src属性描述图片的真实路径。 那么使用promise助手来解决该用例的代码如下： $( "#myButton" ).firstClick(**function**() {  **var** panel = $( "#myPanel" ),  promises = [];  $( "img", panel ).each(**function**() {  **var** image = $( **this** ), src = element.attr( "data-src" );  **if** ( src ) {  promises.push(  $.loadImage( src ).then( **function**() {  image.attr( "src", src );  }, **function**() {  image.attr( "src", "error.png" );  } )  );  }  });  promises.push( panel.slideDownPromise() );  $.when.apply( **null**, promises ).done(**function**() { panel.fadeIn(); }); }); //这里的窍门是跟踪所有的LoadImage 的promise，接下来加入面板slideDown动画。 因此首次点击按钮时，面板将slideDown并且图像将开始加载。 一旦完成向下滑动面板和已加载的所有图像，面板才会淡入。  **/\*在特定延时后加载页面上的图像\*/ /\*  假如，我们要在整个页面实现递延图像显示。 要做到这一点，我们需要的HTML的格式如下：  <img data-src="image1.png" data-after="1000" src="placeholder.png" />  <img data-src="image2.png" data-after="1000" src="placeholder.png" />  <img data-src="image1.png" src="placeholder.png" />  <img data-src="image2.png" data-after="2000" src="placeholder.png" />  意思非常简单：  image1.png，第三个图像立即显示，一秒后第一个图像显示  image2.png 一秒钟后显示第二个图像，两秒钟后显示第四个图像  我们将如何实现呢？ \*/** $("img").each(**function** () {  **var** element = $(**this**), src = element.attr("data-src"), after = element.attr("data-after");  **if** (src) {  $.when($.loadImage(src), $.afterDOMReady(after)).then(**function** () {  element.attr("src", src);  }, **function** () {  element.attr("src", "error.png");  }).done(**function** () {  element.fadeIn();  });  } });  //如果我们想延迟加载的图像本身，代码会有所不同： $( "img" ).each(**function**() {  **var** element = $( **this** ),  src = element.attr( "data-src" ),  after = element.attr( "data-after" );  **if** ( src ) {  $.afterDOMReady( after, **function**() {  $.loadImage( src ).then(**function**() {  element.attr( "src", src );  }, **function**() {  element.attr( "src", "error.png" );  } ).done(**function**() {  element.fadeIn();  });  } );  } }); //这里，我们首先在尝试加载图片之前等待延迟条件满足。当你想在页面加载时限制网络请求的数量会非常有意义。 |
| 结论 | //正如你看到的，即使在没有Ajax请求的情况下，promise也非常有用的。通过使用jQuery 1.5中的deferred实现 ，会非常容易的从你的代码中分离出异步任务。 这样的话，你可以很容易的从你的应用程序中分离逻辑。 |