Chapter 13: Promises
Javascript的asynchronous computation：callback functions
缺点：1，business logic和实现细节被混在一起，2，control flow太复杂，pyramid of doom的问题，3，关于错误的处理没有已有的方式，都是依赖于一个特殊的error arguments in callbacks。

Promises：解决上述问题的尝试。一个将未来的asynchronous call的结果绑定到一个object上的机制，promise object控制了当一个函数的有了结果之后给你对这个value的access。使用Promise时函数不会将callback作为参数，而是返回一个Promise来接受这个callback。
promise支持then的chaining，catch方法用来处理错误，errors也可以通过promise chain进行传播。

```
computeBalance(a, b)
  .then(storeBalance)
  .then(displayResults)
  .catch(handleError);
```

AngularJS内置的service是$q，和其他的promise实现的区别是$q和digest loop关联的，$q可以使用$evalAsync而不是setTimeout。

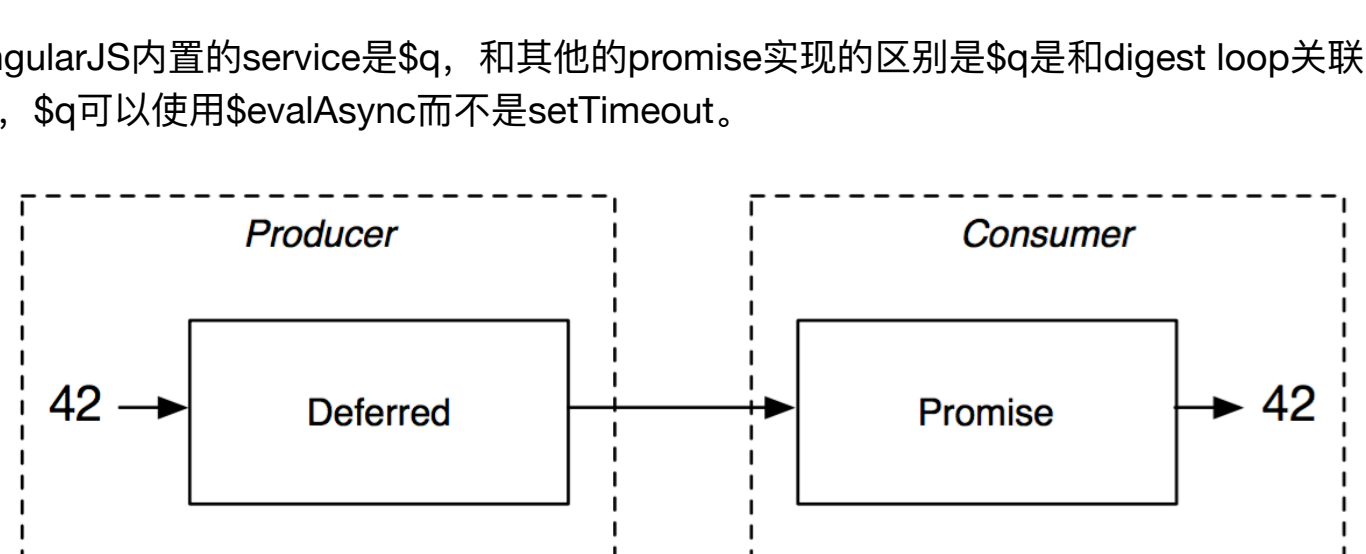

Figure 13.1: The relationship between a Deferred and a Promise

deferred: computation that makes that value available，创建一个deferred对象时也就创建了一个promise

resolve a deferred：promise内部的$$state变量记录callback函数相关的pending, value等变量
When we have a Deferred and a Promise, we can attach a callback on the Promise. Then, after the Deferred is resolved to a value, the callback on the Promise will at some point in the future get invoked with that value.
resolve和给promise注册callback的顺序不需要严格先后，promise callbacks在resolving the Deferred之后的下一个digest被调用。$rootScope.$evalAsync，需要在$QProvider inject一个$rootScope。

Deferred的一个重要feature是只会被resolve一次，如果试图resolve第二次则对应的调用会被忽略，实现方式是resolved时内部的$$state记录$$state.status=1，如果resolve的时候status已经是1则立刻return

确保callback会被调用：此时如果一个callback在一个deferred已经被resolved之后，且一个digest已经运行了之后才注册，则这个callback应该也要被调用。实现是在then方法（注册callback）中判断status，如果大于0则进行一次scheduleProcessQueue。

一个Deferred有且只有一个Promise，但是一个Promise应当可以有多个callback。实现是将pending改成数组。注意为了确保每个callback只被调用一次，当进入processQueue时候需要将pending数组保存在临时变量，然后清空state.pending，这样当之后有新的callback的时候再reinitialize。

rejecting deferred：关于error handling的解决，1，需要有办法能够signal一个Deferred，告诉它有事情不对，这是rejecting the Deferred，2，需要有办法让Deferred在一个rejection发生的时候被notified。实现的时候给then增加第二个参数，作为被reject的时候的callback。
一个Promise最多被reject一次，一个promise一旦被reject了之后就不能再次resolve，一个promise只会有一个outcome，要么是resolution要么是rejection。
实现和resolve很相似，主要是status不同，rejected是2.然后pending数组里面放的不是单个函数了，而是数组，[null, onFulfilled, onRejected]，根据status选择callback。

finally：和try...catch...finally相似，finally方法参数是一个callback，主要用来在asynchronous work之后做cleanup work，在这个promise被resolve之后被调用，不接受任何参数。实现方法是当做一个有同样的success和failure callback的then方法，不论如何都被调用。

promise chaining：

```
it( does not modify original resolution in chains , function() { var d = $q.defer();
var fulfilledSpy = jasmine.createSpy();
d.promise.then(function(result) { return result + 1;
}).then(function(result) { return result * 2;
  });
  d.promise.then(fulfilledSpy);
  d.resolve(20);
  $rootScope.$apply();
  expect(fulfilledSpy).toHaveBeenCalledWith(20);
});
```

实际发生的事情是每个then方法都返回了另外一个new Promise object，用来做further callbacks。因为是new Promise，所以原先Promise上其他的callback都不受影响。
实现：每次then返回一个新的Deferred对象的promise, 然后需要把这个Deferred传递给onFulfilled callback实际上被调用的地方，这样结果可以被传递下去。pending数组改成[result, onFulfilled, onRejected]，然后每次processQueue的时候将callback的结果返回给这个Deferred对象。

```
function processQueue(state) {
var pending = state.pending;
delete state.pending; _.forEach(pending, function(handlers) {
var deferred = handlers[0];
var fn = handlers[state.status]; if (_.isFunction(fn)) {
      deferred.resolve(fn(state.value));
    }
}); }
```

每个新的Deferred对象都是和原来的Deferred对象独立的，但是他们都是在原来的对象被resolved的时候才被resolved。
chains的另外一个重要性质是一个value会被传递直到一个callback handler。比如

```
it( catches rejection on chained handler , function() { var d = $q.defer();
var rejectedSpy = jasmine.createSpy(); d.promise.then(_.noop).catch(rejectedSpy);
  d.reject( fail );
  $rootScope.$apply();
  expect(rejectedSpy).toHaveBeenCalledWith( fail );
});
```

实现是在processQueue方法里，根据当前的status，resolve or reject the chained Deferred with the current Promise's value。

```
function processQueue(state) {
var pending = state.pending;
delete state.pending; _.forEach(pending, function(handlers) {
var deferred = handlers[0];
var fn = handlers[state.status]; if (_.isFunction(fn)) {
deferred.resolve(fn(state.value)); } else if (state.status === 1) { deferred.resolve(state.value);
} else {
      deferred.reject(state.value);
    }
}); }
```

当一个rejection发生的时候，下一个catch会处理，而catch handler的返回值会被当成resolution而不是rejection，继续正常执行

exception handling：explicitly rejecting是一回事，发生异常是另一回事，当一个promise callback抛出异常时应当在下一个rejection handler被处理。同样在processQueue中增加try...catch，有e存在则deferred.reject(e)，作为next promise的rejection。

一个promise callback返回另外一个promise：应该将这个返回的promise连接到chain里面的下一个callback。一个Deferred可能会被另外一个promise进行resolve，这时这个deferred的resolution取决于另外一个promise的resolution。
当一个finally返回promise时，要等待这个promise被resolve才会继续。

notifying progress：Deferred有一个notify方法，用来发送一些关于what kind of progress的信息。notify callback作为then的第三个参数，pending的数组里面也加上它。Deferred的notify方法遍历所有的pending handlers，然后调用progress callbacks。resolution之后notif则不应该再调用progress callback，通过status判断。notifications是会被通过chains传播的。当一个progress callback抛出异常不影响其他callbacks。

包装的API：$q.reject：immediate rejection，$q.when或者$q.resolve：immediate resolution
$q.all：关于promise的一个collection，用$q.all来合并和处理所有的异步操作，返回单个Promise对象来resolve一个results的数组。

ES6-style Promises：没有明显的Deferred的概念，

```
return new Promise(function(resolve, reject) { doAsyncStuff(function(err) {
if (err) { reject(err);
} else {
resolve(); }
}); });
```

$$q：Promises without $digest integration，使用browser timeout，和digest没有关系。

In this chapter you have learned:
- What kinds of problems Promises are designed to solve.
- About some of the existing Promise implementations for JavaScript.
- How AngularJS Promises compare to some of the other existing implementations.
- That Promises are made available in Angular by the $q and $$q services.
- That Promises are always paired with Deferreds, and whereas a Promise is accessed by the consumer of an asynchronously produced value, a Deferred is accessed by its producer.
- That when a Deferred is resolved, the associated Promise callbacks get invoked with `$evalAsync`.
- That each Promise is resolved at most once.
- That each Promise callback is invoked at most once.
- How a Promise callback is invoked even when registered after the Promise was already resolved.
- How Promises can be either resolved or rejected.
- That rejections can be caught by rejection errbacks.
- That you can execute resource cleanup code in finally callbacks, which are callbacks that are invoked for both rejected and resolved Promises.
- How you can chain several `then`, `catch`, and `finally` calls together, and each call creates a new Promise chained to the previous one.
- How even the last (or only) `then` in your Promise chain creates another Promise, but that it is simply ignored.
- How an exception thrown in a Promise handler rejects the next Promise in the chain.
- How an exception caught by a rejection errback resolves the next Promise in the chain.
- That a Promise callback may return another Promise, and how that Promise's eventual resolution or rejection is linked to the Promise chain.
- That finally handlers can also return Promises and they are resolved before continuing with the chain, but that the values they resolve to are ignored.
- How Deferreds can notify about progress, and how progress callbacks are registered.
- That, unlike all other Promise callbacks, notification callbacks may get called several times.
- How notification messages can be transformed in a Promise chain.
- How an immediately rejected Promise can be created with `$q.reject`.
- How an immediately resolved Promise can be created with `$q.when`.
- How `$q.when` can also be used to adopt a foreign Promise.
- How an array or object of Promises can be resolved to an array or object of values with `$q.all`.
- That `$q.all` is rejected if even one of its argument Promises is rejected.
- That not all of the items in the collection given to `$q.all` have to be Promises.
- How `$q` also supports an alternative ES6-style Promise API.

Chapter 14: $http
$http service，处理Angular applications的HTTP通信。$http对象的基础是通过浏览器的XMLHttpRequest support进行HTTP请求，包装了error management，interceptor等等功能。
（这一章除了主要功能之外，除了主要收发请求的实现之外小功能很多比较乱，就和summary放一起总结了，可能跳过一些琐碎的内容）

- That HTTP requests in Angular are handled by two collaborating services:`$http` and `$httpBackend`. The backend may be overriden to provide an alternative transport or to mock it out during testing.

测试的时候使用SinonJS库来检测发出的请求，和假装收response。
HTTP请求是异步的，因此$http不能直接返回response，而是返回一个Promise。$httpBackend来做实际的发送标准XMLHttpRequest，不处理任何的Deferred或者Promises，而是接受一个callback function，作为xhr.onload。xhr.onload会试图从hr.response或xhr.responseText接收response body。$http中构造这个callback函数传给$httpBackend。

- That `$http` kicks o a digest on the `$rootScope` when the response arrives - unless `useApplyAsync` is enabled, in which case it uses `$rootScope.$applyAsync` to do it later.

使用Angular的人不用考虑何时调用$apply，框架会处理。如果有错误发生，可以直接reject the Promise而不是resolve。包括400之类的response code，或者网络故障，CORS等。
response code 在200以上300以下是success。

- That `$http.defaults` (and `$httpProvider.defaults`) allows setting some default configurations that'll be used for all requests.
- How you can supply HTTP headers by providing a `headers` object in the request configuration.
- How Angular has some default headers, such as `Accept` and `Content-Type`, but that you can override them using `$http.defaults`.
- That default and request-specific headers are merged case-insensitively, since that's how headers work according to the HTTP standard.
- That request header values may also be functions that return the concrete header values when called.
request headers：headers可以默认设置，也有request specific，使用_.extend方法将这些headers拼在一起。不是所有HTTP方法的default headers长一样，例如POST应该有一个默认的Content-Type header设置为JSON，GET没有content type。HTTP headers 对key的大小写不敏感。
header也可能是一个返回string的函数。

- That response headers are available through the `headers` function attached to the response object, and how that function treats header names case-insensitively.
- That response headers are parsed lazily only when requested.

response headers首先在$httpBackend获得，xhr.getAllResponseHeaders()，headersGetter函数输入headers string，返回解析了的header names to header values。这个解析是lazy的，到第一个header被request才开始解析。实际的解析工作在parseHeaders函数内完成，根据每一行的:符号进行解析。

- How CORS authorization can be enabled by supplying the `withCredentials` flag in request objects.

CORS默认下跨站请求不包括任何的cookies或者authentication headers，如果需要则需要在XMLHttpRequest设置withCredentials flag。

- That both request and response bodies can be transformed using transforms that are registered with the `transformRequest` and `transformResponse` attributes, which may be either supplied in `$http.defaults` or individual request objects

- That Angular uses request and response transforms to do JSON serialization and parsing by default.
- How request data is serialized to JSON when it is an array or an object, unless it is a special object like a Blob, File, or FormData.
- How response data is parsed as JSON when it's either specified as such with a `Content-Type` header, or when it merely looks like JSON.

JSON like：正则表达式判断，

```
function isJsonLike(data) {
if (data.match(/^\{(?!\{)/)) {
return data.match(/\}$/);
} else if (data.match(/^\[/)) {
return data.match(/\]$/); }
}
```

- How single- and multi-value URL parameters can be supplied, and how they are escaped and attached to the request URL.
- That objects are serialized into JSON when used as URL parameters.
buildUrl方法构造url，例如 http://teropa.info?a=42&b=42

- That Dates are serialized into their ISO 8601 representation when used as URL parameters.

编码方法：Javascript自带的encodeURIComponent

- That three short-hand methods are provided for HTTP methods without bodies: `$http.get`, `$http.head`, and `$http.delete`.
- That three short-hand methods are provided for HTTP methods that do have bodies:`$http.post`, `$http.put`, and `$http.patch`.
- How interceptor factories can be registered directly to the `$httpProvider` or as references to existing factories.
- That interceptors are objects with one or more of the following methods: `request`, `requestError`, `response`, and `responseErrors`.
- How interceptors form a Promise-based pipeline for processing the HTTP request and response.

interceptors：另外一个可以处理HTTP requests and responses的功能，比transform功能更多。interceptors可以随意修改和替换requests and responses。interceptors基于promise实现，因此可以做一些异步的工作。
interceptors是包括了request, requestError, response, responseError 的对象，这些key对应的value是HTTP request不同阶段的函数。例如

```
it( allows intercepting requests , function() {
var injector = createInjector([ ng , function($httpProvider) {
$httpProvider.interceptors.push(function() { return {
request: function(config) { config.params.intercepted = true;
return config; }
}; });
  }]);
  $http = injector.get( $http );
  $rootScope = injector.get( $rootScope );
  $http.get( http://teropa.info , {params: {}});
  $rootScope.$apply();
  expect(requests[0].url).toBe( http://teropa.info?intercepted=true );
});
```

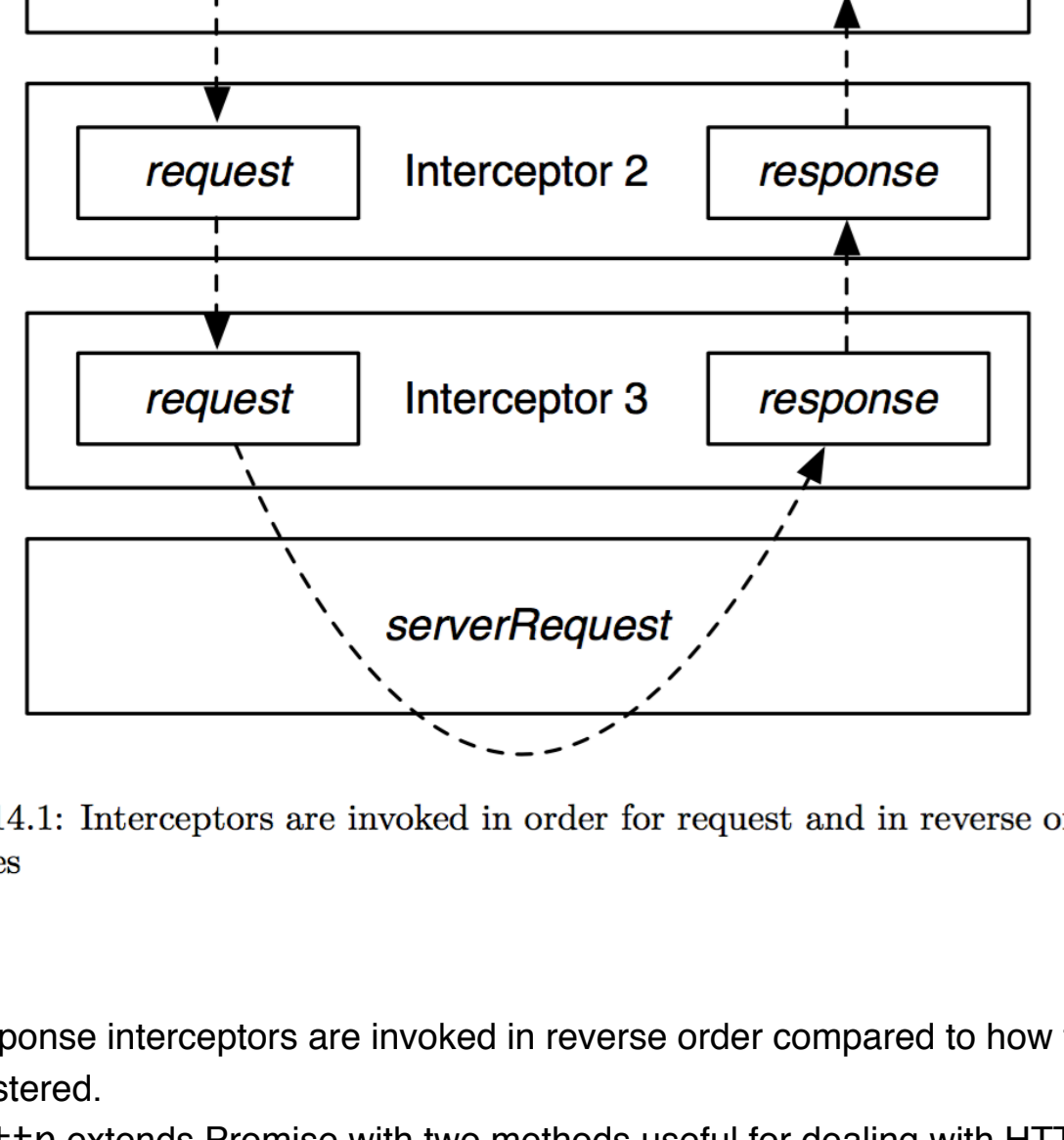

Figure 14.1: Interceptors are invoked in order for request and in reverse order for responses

- That response interceptors are invoked in reverse order compared to how they were registered.
- How `$http` extends Promise with two methods useful for dealing with HTTP responses: `success` and `error`.
- How a request can be aborted with a Promise-based or a numeric timeout attribute.

timeout：应对一些太长时间返回或者没有返回的情况，可以在request configuration object上设置timeout属性，然后设置一些这个值的promise，如果到了时间Angular会中止请求，resolve这个timeout。

- How `$http` can use the `$applyAsync` feature of Scope to skip unnecessary digests when several HTTP responses arrive in quick succession.
- That the `$applyAsync` optimization is not enabled by default, and you have to call `$httpProvider.useApplyAsync(true)` to enable it.

关于$applyAsync和$http，原本的motivation是，在应用启动时通常会发起多个请求，如果服务器很快response也会很快，这是Angular会对每个response开始一个新的digest。优化是如果几个请求很近，他们触发的变化在一个digest内完成，这样可以减少时间消耗。这不是默认的优化，需要添加参数开启。