Chapter 18: Controllers
事实上在Angular中，controllers并不是看上去的那种top-level role，而是只是directive system的一部分。"Directives get top billing in Angular"。本章包括了controller的三个部分，$controller provider，controller和directive compiler的integration，ngController directive。

$controller provider：添加$controller service为ng module的一个部分，自己有个provider。构造方法和其他provider一样，一个provider construct，里面有一个$get方法，返回一个具体的$controller service。

Controller instantiation：AngularJS中的Controllers通常是通过构造函数来创建的，开发者提供constructor function，$controller service在需要的时候实例化。最简单的方法是将构造函数作为参数提供给$controller，返回一个instance，并且注入了对应的dependencies。在$injector service中已经有了instantiate函数可以做这个任务，所以直接实现为：

src/controller.js

```
this.$get = ['$injector', function($injector) {

  return function(ctrl, locals) {
    return $injector.instantiate(ctrl, locals);
  };

}];
```

Controller registration：可以在configuration time注册controllers（注册构造函数到一个内部的object中），然后在runtime查找它（通过查找名字返回一个controller instance）。
另外一种更普遍的注册的方法是通过在module上注册controller constructors，modules有一个controller方法可以调用，module.controller，对于module objects来说是一个对于之前设置的$controllerProvider.register方法的一个queued-up invocation。

src/loader.js

```
var moduleInstance = {
  name: name,
  requires: requires,
  constant: invokeLater('$provide', 'constant', 'unshift'),
  provider: invokeLater('$provide', 'provider'),
  factory: invokeLater('$provide', 'factory'),
  value: invokeLater('$provide', 'value'),
  service: invokeLater('$provide', 'service'),
  decorator: invokeLater('$provide', 'decorator'),
  filter: invokeLater('$filterProvider', 'register'),
  directive: invokeLater('$compileProvider', 'directive'),
  controller: invokeLater('$controllerProvider', 'register'),
  config: invokeLater('$injector', 'invoke', 'push', configBlocks),
  run: function(fn) {
    moduleInstance._runBlocks.push(fn);
    return moduleInstance;
  },
  _invokeQueue: invokeQueue,
  _configBlocks: configBlocks,
  _runBlocks: []
};
```

还有一种注册controllers的方法是从global window object中查找constructor，不过默认不开启，会报错，这种方式也不推荐，因为依赖全局变量不利于模块化。

Directive controllers：controller实际被使用是在directive中。在directive的definition object中，指明一个controller key，value是构造函数的名称，这样当directive被link的时候constructor会被初始化。Controllers是被每个directive分别初始化的，对于同一个元素上的多个directive没有限制，也可以多次使用同一个controller constructor。实现：在applyDirectivesToNode中收集所有包括了controller的directive，然后在nodeLinkFn中调用$controller service进行初始化。
一个有趣的feature是当指明controller name为"@"时，controller是通过DOM元素中directive attribute的值进行查找的，这样可以使得在使用这个directive的时候再指明对应的directive controller，或者可以对同一个directive插入不同的controllers。

Locals in Directive Controllers：目前为止directive和controller之间还没有联系，因为controller还没有任何关于directive的信息。我们可以给controller提供需要的内容，添加以下参数：$scope - The directive's scope object，$element - The element the directive is being applied to，$attrs - The Attributes object of the element the directive is being applied to。将这些参数放在locals里面传给$controller即可。这样directive controller就具有link function可以做的所有事情。

Attaching Directive Controllers on the scope：现在可以把scope object传递给controller，也可以反过来把controller贴在scope上。这样可以使得将controller data and functions放在this上，而不是$scope，但是同样对于DOM中插入的表达式可用，以及child directives and controllers。方法是在definition object中定义controllAs键，从而controller会成为scope的这个属性。

对于isolate scope directives上使用controllers，和non-isolated contexts的情况区别不大，不过有一些和isolate scopes相关的feature有区别。isolate scope应该在instantiating controllers之前被创建，剩下的setup code在instantiation之后进行。isolate scope应当只传递给索取它的directive。
这么做的原因是因为bindToController flag，它是directive definition object的一个属性，用来控制所有的isolate scope bindings会被附着的位置，是controller object还是Isolate scope。isolate scope bindings总是在constructor被调用之前就存在，因为可能在constructor中会用到这个bindings，然而如果bindToController被使用了，isolate scope bindings必须被绑定在controller object上，也就是说我们必须在设置isolate bindings（调用controller constructor）之前就拿到这个controller object。
解决这个问题的方法是给$controller一个later参数，用来使得这个函数返回一个"semi constructed controller"而不是完整的。这个"semi-constructed"的意思是这个controller object已经存在，但是controller constructer却还没有被调用，这样$controller函数的返回值就具有这两个特征：1，这个函数会调用controller constructer，2，他有一个instance属性指向这个controller object。
这样，$controller的调用者就可以在创建controller object和调用constructer之间做一些事情，这里则是设置isolate scope bindings。

src/controller.js

```
return function(ctrl, locals, later, identifier) {
  if (_.isString(ctrl)) {
    if (controllers.hasOwnProperty(ctrl)) {
      ctrl = controllers[ctrl];
    } else if (globals) {
      ctrl = window[ctrl];
    }
  }
  var instance;
  if (later) {
    instance = Object.create(ctrl);
    return _.extend(function() {
      $injector.invoke(ctrl, instance, locals);
      return instance;
```



```
    }, {
      instance: instance
    });
  } else {
    instance = $injector.instantiate(ctrl, locals);
    if (identifier) {
      addToScope(locals, identifier, instance);
    }
    return instance;
  }
};
```

在$controller中修改了之后，在$compile里面做一些修改。首先需要存储semi-constructed controller functions到一个object中，directive.name作为key，然后在"设置isolate scope bindings之前，调用prelink functions之后"，调用这些semi-constructed controller functions。

requiring controllers：Controllers还可以用来作为另外一个不同的directives中间交流的途径，一个directive可以通过设置另外一个directive的名字作为"require"key来索取另外一个directive，如果成功则required directive的controller会成为这个requiring directive的link function的第四个参数。

Requiring Controllers from parent elements：require flag可以设置，修改为不仅从当前元素查找controllers，同样也在父元素查找。设置方法是在名字前面加上"^"。实现需要不仅得到当前元素的controllers，也要得到当前DOM的结构。"^^"前缀和"^"的区别在于只搜索父元素。"?"前缀表示optionally requiring controllers，这样如果找不到controller不会报错，只会返回null。

ngController Directive：先注册ng-controller为ngModule的一个new directive，

```
ngModule.directive('ngController',
  require('./directives/ng_controller'));
```

而ngcontrollerdirective内部的实现很简单，

src/directives/ng_controller.js

```
'use strict';

var ngControllerDirective = function() {
  return {
    restrict: 'A',
    scope: true,
    controller: '@'
  };
};

module.exports = ngControllerDirective;
```

因为已经有了$controller service和$compile 的支持。

Summary:
Controllers are an important part of Angular applications, and in this chapter we have learned pretty much everything there is to know about them.
The implementation of controllers is perhaps surprisingly tightly coupled with directives. Even when "standalone" controllers are used, as with ngController, it's really just an application of the directive controller system.
In this chapter you've learned:

• How controllers can be instantiated by the $controller service.
• How controller instantiation supports dependency injection.
• How controllers can be pre-registered in the config phase using $controllerProvider or modules.
• That controllers can be optionally looked up from window, even though doing it is not recommended.
• How controllers can be attached to directives.
• That every directive gets its own controller instance, and that there can be many directives with controllers on the same DOM node.
• How the special @ value can be used to defer the directive controller resolution to the directive user.
• How $scope, $element, and $attrs are made available for dependency injection in directive controllers.
• How a directive controller can be attached to the directive's scope using controllerAs.
• How isolate scope bindings can be attached to controller objects instead of the isolate scope by using bindToController.
• How bindings can be attached using bindToController even without using an isolate scope.
• How controllers are instantiated in a deferred fashion inside $compile to support bindToController
• How a sibling directive's controller can be required by specifying a require attribute whose value is the sibling directive's name.
• How multiple requires can be specified using arrays.
• That if a directive has a controller and does not require any other directives, it is as if the directive requires itself.
• How parent directives can be required with ^ and ^^.
• How requires can be made optional with ?.
• How the ngController directive works.
• HowcontrollerscanalsobeattachedonthescopebygivingaControllerConstructor as scopeName style expression to $controller.
• How $controller also tries to find controller constructors from the scope, in addition to its own register and window.

Chapter 19: Directive Templates
一个directive不仅可以修改已有的element，也可以往element中增加自己的内容，例如<login-form>会自动渲染一个表单。目前directive system可以通过compile和link functions做一些DOM manipulation，然而还不够简便。Directive templates的作用就是提供一个html字符串，当应用这个directive的时候将这个字符串填入元素中。（Directive templates+isolate scopes可以通过"components"的概念来构造自己的application UI，Angular 2中有所体现）

Basic templating：一个directive可以在definition object中定义template属性，包括一个html字符串，编译的时候简单替换元素的html code。替换的html也会被编译，所以在templates中用到的directive也会被应用到。实现时在遍历directive时，如果有template属性，直接$compileNode.html(directive.template)就可以了。
如果一个元素上有多个template试图编译会报错，在applyDirectivesToNode用一个临时变量记录并检测。
Template Functions：template属性的内容也可以是函数，输入两个参数：DOM node和node的Attributes对象，返回一个string。

isolate scope directives+templates：当一个directive同时定义了isolate scope和template，在template中使用到的directives会收到这个isolate scope或者其后代的一个作为参数。所以如果newIsolateScopeDirective有template，则childLinkFn传入的是这个isolate scope。

templateUrl：更加方便的做法是将模板放在一个单独的html文件里，用URL来进行加载。这个加载过程是异步的，所以在加载的时候，编译过程需要暂停，在模板加载完成之后再恢复。这个"暂停-恢复"的过程是一个实现的难点。
1，加载模板的时候这个元素剩下的所有directives，包括当前directive都不应该被编译，在_.forEach循环中添加短路，return false。
2，当前元素的内容在加载的时候会被清除，避免不必要的编译。增加新函数compileTemplateUrl($compileNode)，函数内清除node内容。
3，使用$http service加载该模板，$http.get.success，回调函数中替换html。
4，此时可以恢复之前的编译了。compileTemplateUrl需要拿到之前还没有编译的directive数组，用_.drop获取这个子数组，还需要拿到这个DOM元素的Attributes object。然后在compileTemplateUrl的success回到函数中，继续使用applyDirectiveToNode函数处理接下来的子数组。
一个小问题是需要替换掉子数组第一个元素的templateUrl为null，否则编译又会中断。
同样，当前元素的子元素也要考虑，在回调函数中增加compileNodes($compileNode[0].childNodes)。

前后两次调用applyDirectivesToNode的上下文是不一样的，第二次调用的时候所有局部变量都不见了，因此需要传进第二次的调用中来。因此我们增加一个对象，成为previous compile context，放入需要的上下文。

src/compile.js

```
function compileTemplateUrl(
    directives, $compileNode, attrs, previousCompileContext) {
  var origAsyncDirective = directives.shift();
  var derivedSyncDirective = _.extend(
    {},
    origAsyncDirective,
    {templateUrl: null}
  );
  var templateUrl = _.isFunction(origAsyncDirective.templateUrl) ?
                      origAsyncDirective.templateUrl($compileNode, attrs) :
                      origAsyncDirective.templateUrl;
  $compileNode.empty();
  $http.get(templateUrl).success(function(template) {
    directives.unshift(derivedSyncDirective);
    $compileNode.html(template);
    applyDirectivesToNode(
      directives, $compileNode, attrs, previousCompileContext);
    compileNodes($compileNode[0].childNodes);
  });
}
```

Linking Asynchronous Directives：现在可以在拿到模板之后恢复compilation了，但是还不能link所有的directives，因为第二次调用applyDirectivesToNode返回值都没用上。解决方案分为几个步骤，
1，当node上的一个directive需要异步获取template的时候，不应该返回正常的node link function，而应该返回一种特殊的"delayed node link function"，从compileTemplateUrl返回。这样就可以覆盖循环外本地的nodeLinkFn，成为applyDirectivesToNode的返回值。
"delayed node link function"负责处理关于node的整个linking process，因为我们要让他覆盖本地的nodeLinkFn。在里面该做什么呢?
a. link all the directives that we've compiled asynchronously - both from the current node and from the child nodes，这些link functions就是之前没有管的返回值，我们将他们保存为afterTemplateNodeLinkFn and afterTemplateChildLinkFn，因为是在装载template之后进行link的函数。
b. 在delayedNodeLinkFn中，应该调用这些after link function了，他们会被当做一般的node link function处理，也要接收三个参数，child link function, scope to link, node being linked。其中，child link function是我们在加载template之前的child link function，而不是之后，然而因为我们都把node children抹掉了，所以也就可以忽略掉了。

```
var afterTemplateNodeLinkFn, afterTemplateChildLinkFn;
$compileNode.empty();
$http.get(templateUrl).success(function(template) {
  directives.unshift(derivedSyncDirective);
  $compileNode.html(template);
  afterTemplateNodeLinkFn = applyDirectivesToNode(
    directives, $compileNode, attrs, previousCompileContext);
  afterTemplateChildLinkFn = compileNodes($compileNode[0].childNodes);
});

return function delayedNodeLinkFn(_ignoreChildLinkFn, scope, linkNode) {
  afterTemplateNodeLinkFn(afterTemplateChildLinkFn, scope, linkNode);
};
}
```

2，现在的compile过程是先等待模板加载完成，然后调用public link function，这并不是必要的条件，因为如果这样的话，public link function的调用者需要等待所有的template加载完，但是使用Angular的程序员并不应该考虑它，所以需要支持在加载完成之前就调用public link function。
实现方法是使用一个本地变量link queue，在delayedNodeLinkFn里面，如果有linkQueue，就将参数放在里面，如果没有，则像之前一样直接调用即可。

```
return function delayedNodeLinkFn(_ignoreChildLinkFn, scope, linkNode) {
  if (linkQueue) {
    linkQueue.push({scope: scope, linkNode: linkNode});
  } else {
    afterTemplateNodeLinkFn(afterTemplateChildLinkFn, scope, linkNode);
  }
};
```

在收到了模板之后再一次调用link queue里面的项目。

```
$http.get(templateUrl).success(function(template) {
  directives.unshift(derivedSyncDirective);
  $compileNode.html(template);
  afterTemplateNodeLinkFn = applyDirectivesToNode(
    directives, $compileNode, attrs, previousCompileContext);
  afterTemplateChildLinkFn = compileNodes($compileNode[0].childNodes);
  _.forEach(linkQueue, function(linkCall) {
    afterTemplateNodeLinkFn(
      afterTemplateChildLinkFn, linkCall.scope, linkCall.linkNode);
  });
  linkQueue = null;
});
```

这也就意味着，你不能总是确定函数返回的时候所有的linking都完成了，只能确定在加载完成之后是这样。

Linking directives that were compiled earlier：现在还漏了一件事情，在遇到异步加载template之前所有编译了的directives还没有linking。因此需要将前面的preLinkFns和postLinkFns放入第二次调用applyDirectivesToNode的上下文对象里。newIsolateScopeDirectives也是同样，还有controller directive mapping object，统统放入上下文对象中。然后在applyDirectivesToNode里的初始化中再拿出来。

src/compile.js

```
function applyDirectivesToNode(
    directives, compileNode, attrs, previousCompileContext) {
  previousCompileContext = previousCompileContext || {};
  var $compileNode = $(compileNode);
  var terminalPriority = -Number.MAX_VALUE;
  var terminal = false;
  var preLinkFns = previousCompileContext.preLinkFns || [];
  var postLinkFns = previousCompileContext.postLinkFns || [];
  var controllers = {};
  var newScopeDirective;
  var newIsolateScopeDirective = previousCompileContext.newIsolateScopeDirective;
  var templateDirective = previousCompileContext.templateDirective;
  var controllerDirectives = previousCompileContext.controllerDirectives;
```

Summary
Directive templates are not a complicated feature to understand or to implement. In fact, in the beginning of the chapter, we already had a functional implementation after just a couple of pages.
However, when we started to build support for the asynchronous loading of templates, things got a lot more complicated. We needed to build a full pause/resume mechanism for compilation and linking, and in the process ended up shifting a lot of state around between functions. It is sometimes surprising just how complicated it can be to implement seemingly simple features!
In this chapter you have learned:
• That when a directive has a template attribute, its contents are used to populate the inner HTML of the element.
• That when a template is used, it replaces any existing contents an element may have had.
• How a template's contents also get compiled and linked.
• That only one template directive may be used for each element.
• How you can also use a function as the value of template or templateUrl, to support dynamically forming the template or its URL.
• That when isolate scopes are used together with templates, the contents of the template are linked with the isolate scope.
• How the compilation process is paused and later resumed when a template is asynchronously loaded with templateUrl.
• How the linking process may be suspended and later resumed if the public link function is called while there are templates loading.