Chapter 15：DOM Compilation and Basic Directives
directive compiler：将自定义的行为通过特殊元素、属性、类、注释来附着在DOM元素上。
compilation：将directives应用到DOM上的过程。输入DOM结构，输出转变之后的DOM结构。几乎任何转变都可能

注册$compile Provider：
```
ngModule.provider('$compile', require('./compile'));
```
注册Directives：
directive的注册和filters、service、factory类似，通过模块的directive方法。在loader.js的moduleInstance中增加directive方法
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'),
  config: invokeLater('$injector', 'invoke', 'push', configBlocks),
  run: function(fn) {
    moduleInstance._runBlocks.push(fn);
    return moduleInstance;
  },
  _invokeQueue: invokeQueue,
  _configBlocks: configBlocks,
  _runBlocks: []
};
```

注册一个directive的时候实际上相当于给injector（provider）一个factory，但是一个directive name可以注册多个directive（维护为每个name对应一个数组）。注册时会自动在名字后面加上Directive后缀。

directive definition object：键值对对应这个directive的行为。一个key是compile，定义compilation function。应用到DOM上是通过给DOM加一个和directive name相同名字的element。
单元测试的内容：
test/compile_spec.js
```
it('compiles element directives from a single element', function() {
  var injector = makeInjectorWithDirectives('myDirective', function() {
    return {
      compile: function(element) {
        element.data('hasCompiled', true);
      }
    };
  });
  injector.invoke(function($compile) {
    var el = $('<my-directive></my-directive>');
    $compile(el);
    expect(el.data('hasCompiled')).toBe(true);
  });
});
```

1，注册一个myDirective到一个injector里面
2，使用jQuery来解析一个DOM段，element name是my-element
3，调用injector的$compile方法来编译

编译过程：compileNodes遍历整个jQuery object，分别处理每个node。collectDirectives输入一个DOM node，通过addDirective找到应该应用的directives，然后作为一个数组返回。回到compileNodes，通过applyDirectivesToNode方法将directives应用到node上，通过调用每个directive的compile方法。
Angular将directive和DOM搭配的方法包括四种，element name, attribute name, class name, special comments

recursing to child elements：通过node.childNodes访问到子节点，递归调用
前缀：Angular允许的DOM名字的前缀有x和data，也可以使用_-,:符号

restricting directive application：给directive definition object增加一个restrict属性，来限制这个directive可用于搭配哪几种matching modes。
• E means "match with element names"
• A means "match with attribute names"
• C means "match with class names"
• M means "match with comments"
• EA means "match with element names and attribute names"
• MCA means "match with comments, class names, and attribute names" • etc.

Prioritizing directives：directive definition object增加priority属性，priority越大优先级越大。priority相同比较名字，名字相同比较注册的先后顺序。

terminating compilation：terminal directive用于中止编译，例如ng-if的情况。增加terminal:true属性

Applying directives across multiple nodes:
```
it('allows applying a directive to multiple elements', function() { var compileEl = false;
var injector = makeInjectorWithDirectives('myDir', function() {
return {
multiElement: true, compile: function(element) {
compileEl = element;
}
};
});
injector.invoke(function($compile) {
var el = $('<div my-dir-start></div><span></span><div my-dir-end></div>'); $compile(el);
expect(compileEl.length).toBe(3);
});
});
```
directive需要设置multiElement:true的属性，并且只能应用于attribute matching。compile函数应用的对象是start element和end element中间的所有object
1，发现-start后缀时，处理名字去掉后缀，2，寻找有没有对应的multi-element directive
实现时在处理了name之后，对对应的directive object增加$$start和$$end。
src/compile.js
```
function addDirective(directives, name, mode, attrStartName, attrEndName) {
  if (hasDirectives.hasOwnProperty(name)) {
    var foundDirectives = $injector.get(name + 'Directive');
    var applicableDirectives = _.filter(foundDirectives, function(dir) {
      return dir.restrict.indexOf(mode) !== -1;
    });
    _.forEach(applicableDirectives, function(directive) {
      if (attrStartName) {
        directive = _.create(directive, {
          $$start: attrStartName,
          $$end: attrEndName
        });
      }
      directives.push(directive);
    });
  }
}
```

applyDirectivesToNode时，当发现了$$start属性，则调用groupScan方法寻找作用的nodes。
src/compile.js
```
function groupScan(node, startAttr, endAttr) {
  var nodes = [];
  if (startAttr && node && node.hasAttribute(startAttr)) {
    var depth = 0;
    do {
      if (node.nodeType === Node.ELEMENT_NODE) {
        if (node.hasAttribute(startAttr)) {
          depth++;
        } else if (node.hasAttribute(endAttr)) {
          depth--;
        }
      }
      nodes.push(node);
      node = node.nextSibling;
    } while (depth > 0);
  } else {
    nodes.push(node);
  }
  return $(nodes);
}
```

In this chapter you have learned:
• How the $compile provider is constructed
• That directive registration happens outside of the core dependency injection mechanisms but integrates with it.
• That directives are available from the injector though they're usually applied via DOM compilation.
• That an Angular application may have several directives with the same name.
• How several directives can be registered with one module.directive() invocation using the shorthand object notation.
• How the DOM compilation happens: For each node in the compiled DOM, matching directives are collected and then applied to the node.
• How the compiler recurses into child nodes after their parents are compiled.
• How the directive name can be prefixed in di erent ways when specified in the DOM.
• How the di erent matching modes (element, attribute, class, comment) work.
• How directives can be restricted to only match certain modes, where the default is 'A' for attribute matching.
• How directives can be applied over a group of sibling nodes by having -start and -end attributes in separate elements.

Chapter 16: Directive Attributes
element directive和class directive通常需要能够接触到属性。Attributes可以被用来配置Directives，同directives传递信息，并且Attributes也可以提供directive to directive 的通讯，通过observing的机制。
首先给compile函数增加第二个参数 attrs。对于每个编译的node需要构造一个attributes对象，并且在collectDirectives函数里收集这些attributes放入对象中。单元测试的一般过程：1，注册一个directive，2，编译一个DOM段，3，拿到attributes object，4，对这个attributes object运行一些测试。

Boolean attributes: 例如disabled，Boolean attributes的特点在于他们一般没有一个显式的值，而是他们的存在意味着"true"，并且只应用与standard HTML里面定义的属性。
ng-attr-前缀的属性会覆盖没有这个前缀的同名属性。正则表达式：var isNgAttr = /^ngAttr[A-Z]/.test(normalizedAttrName);

$set方法：attribute object现在包括了normalized attributes，我们给它除了读还加上写属性的方法$set。给Attributes构造函数放加上Attributes(element)参数，Attributes.prototype.$set里面将key,value设置在自己上，this[key]=value，同时如果要将参数写到DOM元素上，还要调用element的DOM上attr方法，this.$$element.attr(key,value)。$set方法增加一个参数决定是否将set的内容写到DOM元素上。

Observing attributes：Attributes对象提供了一种directives中间针对单个元素的交流的方法。如果一个directive对一个attribute做了改变，要通知另一个directive这个改变事件，这个机制可以通过$watch完成，但是更好的是$observe机制。原因包括节约CPU cycle，
test/compile_spec.js
```
it('calls observer immediately when attribute is $set', function() {
  registerAndCompile(
    'myDirective',
    '<my-directive some-attribute="42"></my-directive>',
    function(element, attrs) {

      var gotValue;
      attrs.$observe('someAttribute', function(value) {
        gotValue = value;
      });

      attrs.$set('someAttribute', '43');

      expect(gotValue).toEqual('43');
    }
  );
});
```

实现方法是维护一个observers的表，key是attribute name，value是针对这个attribute的observer functions的数组。
src/compile.js
```
Attributes.prototype.$observe = function(key, fn) {
  var self = this;
  this.$$observers = this.$$observers || Object.create(null);
  this.$$observers[key] = this.$$observers[key] || [];
  this.$$observers[key].push(fn);
  $rootScope.$evalAsync(function() {
    fn(self[key]);
  });
};
```
这里用Object.create(null)而不是直接用null的原因是为了防止一些Object prototype潜在的属性名如toString, constructor等。虽然observers通常和digest没有关系，在初次调用里面仍然调用了Scope.$evalAsync，为了方便结异步调用。
```
  if (this.$$observers) {
    _.forEach(this.$$observers[key], function(observer) {
      try {
        observer(value);
      } catch (e) {
        console.log(e);
      }
    });
  }
```
在每个循环体里面，注意用try...catch...，当一个function出错的时候不影响其他这个observers里面的函数执行。

providing class directives as attributes：class name同样会被放在attributes里面，设为undefined。因为class name可能会有value：
test/compile_spec.js
```
it('terminates class directive attribute value at semicolon', function() {
  registerAndCompile(
    'myDirective',
    '<div class="my-directive: my attribute value; some-other-class"></div>',
    function(element, attrs) {
      expect(attrs.myDirective).toEqual('my attribute value');
    }
  );
});
```
所以需要用到正则表达式来拿出class name，正则表达式是：/([\d\w\-_]+)(?:\:([^;]+))?;?/
（[\d\w\-_]+)对应class name，(?...)?代表可选，其他两个括号是capturing group（通过比对正则表达式的结果拿到的）。

manipulating classes：Attributes对象同样提供了一些操纵元素的CSS classes的方法，比如增加、删除、更新。实现方法直接使用this.$$element.addClass, removeClass方法，update则使用是_.difference比较两个数组，然后删掉旧的classes加上新的classes。

In this chapter you have learned:
• How an element's attribute values are made available to directives using the Attributes object
• That boolean attribute values are always true when present, but only for the standard HTML boolean attributes.
• How ng-attr- prefixes are handled.
• That the same Attributes object is shared by all the directives of an element.
• How attributes can be set using $set, and how the caller can decide whether to also change the corresponding DOM attribute.
• What the attribute name denormalization rules are when $setting an attribute on the DOM.
• How attribute changes can be observed, and that only applies to the attributes set with $set and not to attribute changes by other means.
• That class directives also become attributes and may optionally have values attached to them.
• That comment directives also become attributes and may optionally have values attached to them.
• How the Attributes object provides some convenience methods for adding, removing, and updating the CSS classes of the element.

Chapter 17: Directive Linking and Scopes
之前两章建立了DOM compilation，遍历整个DOM然后寻找directives，这一章做的是Angular directive system的下一步，Linking。Linking是将compiled DOM tree和scope object结合，包括application data and functions，以及绑定之后的结果。

Public link function：将directives应用到DOM上然后返回一个public link function。Link compile DOM tree to Scopes。$compile函数返回一个函数叫做public link function。
该函数做的事情包括：1，将一些debug information以jQuery/jqLite data的形式放到DOM上。2，主要工作是初始化actual linking of directives to the DOM，也就是directive link functions。

Directive link functions：每个directive可以有自己的link function，和compile function类似，区别在于：1，调用的时间不同，2，link function除了能够访问DOM element and Attributes object之外，还可以访问到链接的Scope object，通常application data and functionality所在的地方。
定义Directive linking functions的方法有几种：
1，directive的compile function返回一个link function。比如：
test/compile_spec.js
```
it('calls the directive link function with scope', function() {
  var givenScope, givenElement, givenAttrs;
  var injector = makeInjectorWithDirectives('myDirective', function() {
    return {
      compile: function() {
        return function link(scope, element, attrs) {
          givenScope = scope;
          givenElement = element;
          givenAttrs = attrs;
        };
      }
    };
  });
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-directive></div>');
    $compile(el)($rootScope);
    expect(givenScope).toBe($rootScope);
    expect(givenElement[0]).toBe(el[0]);
    expect(givenAttrs).toBeDefined();
    expect(givenAttrs.myDirective).toBeDefined();
  });
});
```

这个link function有三个参数，我们期望scope=给public link function的scope，element=这个directive应用的elements，attrs=这个elements的参数。这种compile function返回link function的pattern应用到了整个compilation和linking process的各个level。
实现时需要能够这么几步：对于一个nodes的compile函数返回一个composite link function用来链接所有的nodes，对于一个node有自己的link functions。目前compile nodes和link nodes是一对一的关系，但是之后不是这样（因为nodes可能会有变动），步骤如下图：

• The public link function is used to link the whole DOM tree that we're compiling
  – The composite link function links a collection of nodes
    * The node link function links all the directives of a single node
      · The directive link function links a single directive.

compile and link functions的关系：

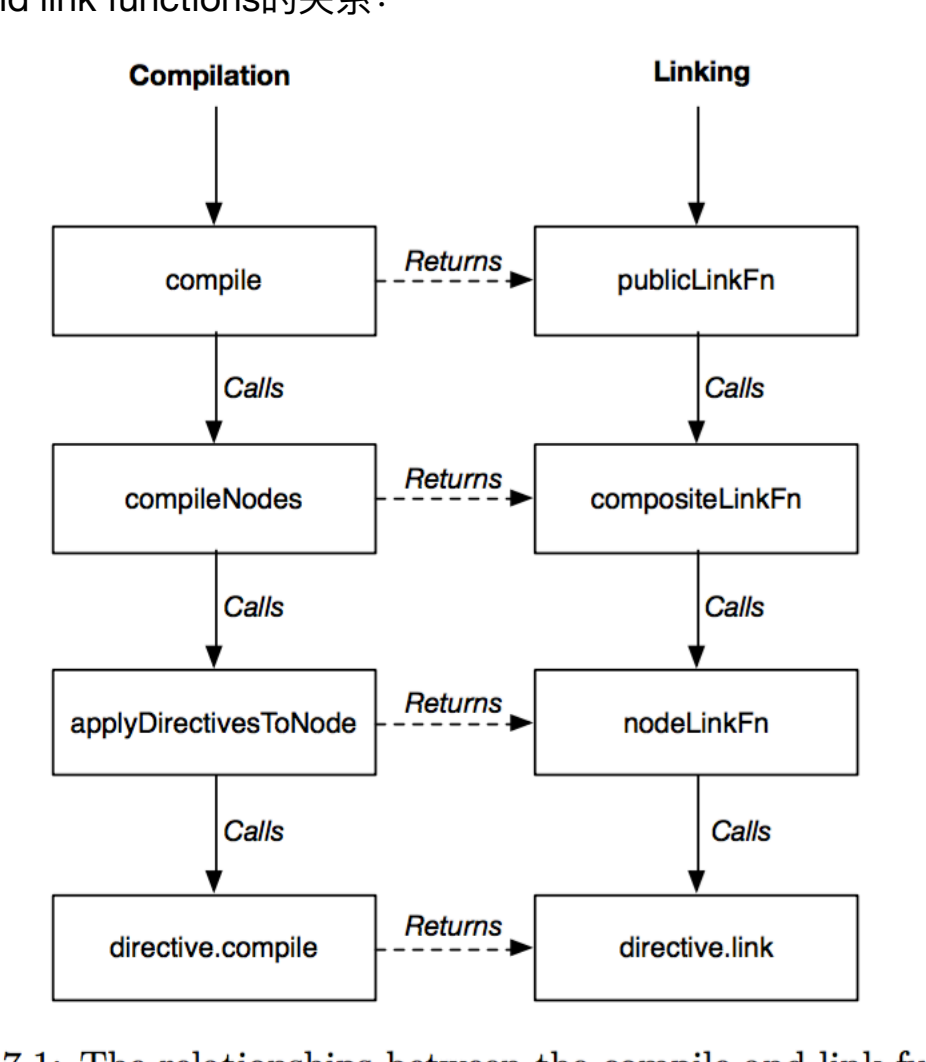

Figure 17.1: The relationships between the compile and link functions

Linking Child Nodes：当编译了整个DOM tree之后，directives on lower levels应该先link，也就是说child elements should be linked before parent element。另外加入父节点没有directives applied to it，子节点仍然应该get linked。

Pre- and Post- Linking：存在两种link functions，Prelink functions在子节点被linked之前调用，postlink则在之后。
```
it('supports prelinking and postlinking', function() {
  var linkings = [];
  var injector = makeInjectorWithDirectives('myDirective', function() {
    return {
      link: {
        pre: function(scope, element) {
          linkings.push(['pre', element[0]]);
        },
        post: function(scope, element) {
          linkings.push(['post', element[0]]);
        }
      }
    };
  });
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-directive><div my-directive></div></div>');
    $compile(el)($rootScope);
    expect(linkings.length).toBe(4);
    expect(linkings[0]).toEqual(['pre',  el[0]]);
    expect(linkings[1]).toEqual(['pre',  el[0].firstChild]);
    expect(linkings[2]).toEqual(['post', el[0].firstChild]);
    expect(linkings[3]).toEqual(['post', el[0]]);
  });
});
```
现在确定link functions invocation的顺序是：1. Parent prelink 2. Child prelink 3. Child postlink 4. Parent postlink

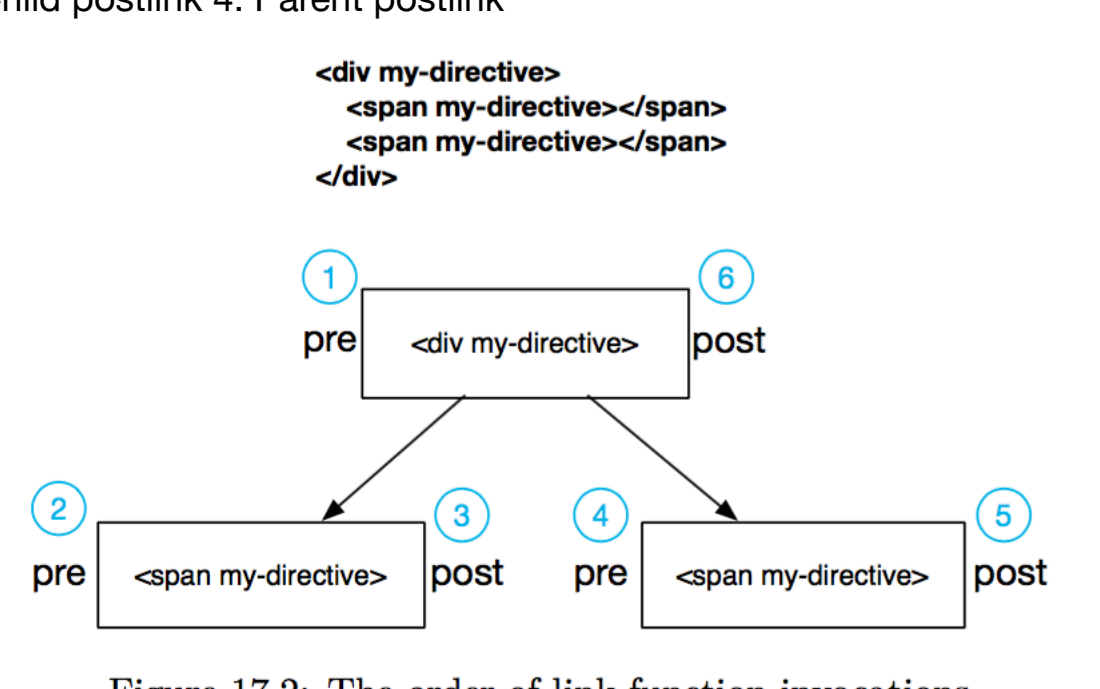

Figure 17.2: The order of link function invocations

Keeping The Node List Stable for Linking：之前说过，composite link function目前需要compile nodes和link nodes一一对应，然而因为directives通常会进行操作DOM的活动，所以这并不一定成立。实现方式是在运行node link functions之前先copy一份node collection叫做stableNodeList，注意copy的时候要保留index：
```
function compositeLinkFn(scope, linkNodes) {
  var stableNodeList = [];
  _.forEach(linkFns, function(linkFn) {
    var nodeIdx = linkFn.idx;
    stableNodeList[nodeIdx] = linkNodes[nodeIdx];
  });
```

Linking directives across multiple nodes：multiElement directives的linking需要考虑从-start到-end element的所有elements的集合。在public link function和directive link function之间加了一层wrapper，用来根据start element和start, end attribute names来得到对应的element group。

Linking and scope inheritance：在处理了linking process之后，目前所有的directive link functions 用的是同一个scope，现在需要让directive能够有自己的scope。当一个element至少有一个directive索取inherited scope的时候，这个元素上的所有directives都会使用这个inherited scope。当一个元素涉及了scope inheritance的时候，元素上会增加一个ng-scope的CSS class，和一个新的Scope object作为jQuery/jqLite data。
Directives may ask new scopes to be created, in which case the elements where the directives are applied - as well as all of their children - get the inherited scope.
实现：当看到一个directive在要求一个新的scope的时候，在composite link function使用scope = scope.$new()替换scope为子scope，并设置相应的attrs.$element。

Isolated Scopes：isolated scopes的特点是在scope hierarchy中但是不继承父scope的属性。当在directive中使用isolate scope的时候更容易让directive变得更加modular。Directive system允许使用isolate scope bindings将元素从surrounding environment绑定到isolate scope上，和前面的normal scope inheritance的区别在于这里的属性传递必须是显式的。如果一个directive使用了isolate scope不影响这个元素上的其他scope，也不影响子元素。
isolate scope不和该元素和其子元素的其他directives共享，并且一个元素只能有一个isolate scope，否则会throw。

isolate attribute bindings：方法包括：1，让scope attributes 和element的属性绑定，element 的 attribute observer使用observe机制来更新scope attribute。优势在于因为所有directives共享同一个Attributes object，方便共享信息。
test/compile_spec.js
```
it('allows observing attribute to the isolate scope', function() {
  var givenScope, givenAttrs;
  var injector = makeInjectorWithDirectives('myDirective', function() {
    return {
      scope: {
        anAttr: '@'
      },
      link: function(scope, element, attrs) {
        givenScope = scope;
        givenAttrs = attrs;
      }
    };
  });
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-directive></div>');
    $compile(el)($rootScope);

    givenAttrs.$set('anAttr', '42');
    expect(givenScope.anAttr).toEqual('42');
  });
});
```

scope中的anattr属性设置了character @，表示"Attribute bindings"。实现是在nodeLinkFn中，对于每个$$isolateBindings属性，设置attrs.$observe。attribute name和isolate scope中的name也可以修改，通过在directive的scope definition中设置别名在@后面，用正则表达式匹配。
2，bi-directional data binding：

Bi-directional data binding allows for some of the same kind of data sharing between parent and child scopes as non-isolated inherited scopes do, but there are a few crucial differences:

Scope Inheritance	Bi-Directional Data Binding
Everything is shared from the parent to the child.	Only attributes explicitly mentioned in expressions are shared.
One-to-one correspondence between parent and child attributes.	Child attributes may not have matching parent attributes, but but can be any expressions instead.
Assigning an attribute on the child scope *shadows* the parent attribute.	Assigning an attribute in the child *updates* the parent attribute - hence *bi-directional data binding.*

最简单的双向绑定配置是在scope definition object里使用"="符号，理解为"将这个属性看作是parent scope中的一个表达式"。双向绑定有用的时候在于你可以从parent scope的一些表达式中使用一些isolate scope中的属性。
实现步骤是：1，得到DOM中关于这个binding的expression string，用正则表达式，2，将这个字符串解析为Angular expression，使用$parse，3，在parent scope的上下文中执行编译得到的expression，4，将结果设为isolate scope的一个$watch的属性。
src/compile.js
```
case '=':
  var parentGet = $parse(attrs[attrName]);
  isolateScope[scopeName] = parentGet(scope);
  scope.$watch(parentGet, function(newValue) {
    isolateScope[scopeName] = newValue;
  });
  break;
```

双向：当在isolate scope上assign an attribute时，可能影响到parent scope。双向的binding中parent优先。定义parentValueWatch函数监控parentValue，如果parentValue和lastValue不同说明parent有变化，赋值给child，否则child赋值给parent，并且更新lastValue。
src/compile.js
```
case '=':
  var parentGet = $parse(attrs[attrName]);
  var lastValue = isolateScope[scopeName] = parentGet(scope);
  var parentValueWatch = function() {
    var parentValue = parentGet(scope);
    if (isolateScope[scopeName] !== parentValue) {
      if (parentValue !== lastValue) {
        isolateScope[scopeName] = parentValue;
      } else {

        parentValue = isolateScope[scopeName];
        parentGet.assign(scope, parentValue);
      }
    }
    lastValue = parentValue;
    return lastValue;
  };
  scope.$watch(parentValueWatch);
  break;
```

这里采用的是 reference watches，这对于数组等collection是无效的，因此需要使用$watchCollection，对应的特殊符号是"=*"。双向绑定可以设置为可选，意思是如果引用的attribute在DOM element上不存在，则不会创建对应的watcher，符号是"=?"。

Expression binding：符号是"&"，和之前两种绑定的区别在于这里主要用来绑定行为而不是数据，当你应用这个directive的时候，你给这个directive提供一个在某事件发生的时候可以调用的表达式，如ngClick这种event-driven directives的场景下最有用。
src/compile.js
```
case '&':
  var parentExpr = $parse(attrs[attrName]);
  isolateScope[scopeName] = function(locals) {
    return parentExpr(scope, locals);
  };
  break;
```

Summary
By this point we have both of the core processes of the directive system implemented: Compilation and linking. We understand how directives and scopes interact and how the directive system creates new scopes.
In this chapter you have learned:
• How linking is built into the functions returned by the compile functions.
• How the public link function, the composite link functions, the node link functions, and the directive link functions are all return values of their respective compile functions, and how they are chained together during linking.
• That a directive's compile function should return its link function.
• That you can omit a directive's compile function and supply the link function directly.
• How child nodes get linked whether the parent nodes have directives or not.
• That prelink functions are invoked before child node linking, and post link functions after it.
• That a link function is always a postlink function unless explicitly defined otherwise.
• How the linking process protects itself from DOM mutations that occur during linking.
• How the nodes for multi-element directives are resolved during linking.
• How directives can request new, inherited scopes.
• That inherited scopes are shared by all the directives in the same element and its children.
• How CSS classes and jQuery data are added to elements that have directives with inherited scopes.
• How directives can request new isolate scopes.
• That isolate scopes are not shared between directives in the same element or its children.
• That there can only be one isolate scope directive per element.
• That there cannot be inherited scope directives on an element when there is an isolate scope directive.
• How element attributes can be bound as observed values on an isolate scope.
• How bi-directional data bindings can be attached on an isolate scope.
• That when both the parent and child change simultaneously in a bi-directional data binding, the parent takes precedence.
• How collections are supported in bi-directional data bindings.
• How invokable expressions can be attached on an isolate scope.
• How named arguments can be used with invokable expressions.