Chapter 21: Interpolation
interpolation是Angular常见的markup，类似于{{data}}。interpolation可以很简单的实现expression和DOM的绑定。实现interpolation需要用到之前的Watches, expressions and directives。

$interpolate service：一般开发者很少直接用到$interpolate，而是$compile用到。
bootstrap一个service的过程和之前一样。

$interpolating function：$interpolate function处理一个string，返回一个函数，函数可以用来evaluate all the expressions in the string，并且得到interpolated result。该函数有一个参数context object，通常是一个scope，用来evaluate the expressions。
这个过程和$parse很相似，实际上也是基于$parse函数得到的。除去{{和}}之后用$parse解析表达式，对于一个以上的表达式，用一个循环收集到parts中。收集了之后使用_.reduce函数，将得到的字符串连接在一起。
src/interpolate.js

```
function $interpolate(text) {
  var index = 0;
  var parts = [];
  var startIndex, endIndex, exp, expFn;
  while (index < text.length) {
    startIndex = text.indexOf('{{', index);
    endIndex = text.indexOf('}}', index);
    if (startIndex !== -1 && endIndex !== -1) {
      if (startIndex !== index) {
        parts.push(text.substring(index, startIndex));
      }
      exp = text.substring(startIndex + 2, endIndex);
      expFn = $parse(exp);
      parts.push(expFn);
      index = endIndex + 2;
    } else {
      parts.push(text.substring(index));
      break;
    }
  }

  return function interpolationFn(context) {
    return _.reduce(parts, function(result, part) {
      if (_.isFunction(part)) {
        return result + part(context);
      } else {
        return result + part;
      }
    }, '');
  };

}
```

用一个stringify函数处理每一个value，对于null和undefined返回''，对于object使用JSON.stringify方法。

escaped interpolation symbols：如果在UI中要用到{{或者}}需要escape，加一个\符号，每次将静态内容添加到parts中的时候需要经过unescapeText函数。
src/interpolate.js

```
function unescapeText(text) {
  return text.replace(/\\{\\{/g, '{{')
             .replace(/\\}\\}/g, '}}');
}
```

skipping interpolation when no expressions：性能优化，如果没有需要interpolate的内容可以不必返回interpolation function。$interpolate函数增加第二个参数mustHaveexpressions，在循环中判断是否存在expression，是的话仍然返回函数，否则不返回。

text node interpolation：在compile过程中，可能有text node和attribute两种interpolation。当在text node中存在interpolated expression的时候，会被替换成通过上下文得到的expression value，并且该expression会被watched，当value变化的时候text node也变化。
test/compile_spec.js

```
describe('interpolation', function() {

  it('is done for text nodes', function() {
    var injector = makeInjectorWithDirectives({});
    injector.invoke(function($compile, $rootScope) {
      var el = $('<div>My expression: {{myExpr}}</div>');
      $compile(el)($rootScope);

      $rootScope.$apply();
      expect(el.html()).toEqual('My expression: ');

      $rootScope.myExpr = 'Hello';
      $rootScope.$apply();
      expect(el.html()).toEqual('My expression: Hello');
    });
  });

});
```

实现方法是当在compilation中遇到text node的时候，凭空生成一个directive并添加到该text node上，在这个directive里执行interpolation logic。collectDirectives目前只关心ELEMENT_NODE和COMMENT_NODE，增加新的else if。inject $interpolate service并在分支中调用。
src/compile.js

```
function addTextInterpolateDirective(directives, text) {
  var interpolateFn = $interpolate(text, true);
  if (interpolateFn) {
    directives.push({
      priority: 0,
      compile: function() {
        return function link(scope, element) {
          scope.$watch(interpolateFn, function(newValue) {
            element[0].nodeValue = newValue;
          });
        };
      }
    });
  }
}
```

priority不是很重要，因为text node上没有其他的directives，link function中进行$watch，interpolateFn可以直接作为watch function，listener function中更新text node value。
此外，Angular还在该directive中，对当前元素的父元素增加'ng-binding' css，并对父元素增加$binding，包括了所有被interpolated的actual expressions.

Attribute interpolation：另外一种DOM中的interpolation是在attributes中，和text node interpolation类似，但是attributes可能会被该元素上的其他directives影响，因此略复杂一些。
实现思路和text node interpolation类似，生成一个directive并且设置watch function。
src/compile.js

```
function addAttrInterpolateDirective(directives, value, name) {
  var interpolateFn = $interpolate(value, true);
  if (interpolateFn) {
    directives.push({
      priority: 100,
      compile: function() {
        return function link(scope, element) {
          scope.$watch(interpolateFn, function(newValue) {
            element.attr(name, newValue);
          });
        };
      }
    });
  }
}
```

注意priority设置到了100，目的是为了在directives中优先被编译。
和text node interpolation的不同点有以下这些：
1，其他directives可能通过Attributes.$observe 关联了属性，因此被interpolate改变了值的时候这些observer应该被触发。jQuery的attr函数是不会触发的，应该用Attributes object的$set方法。这样还有一个问题，这些observer会被interpolation之前的值所触发，如：
test/compile_spec.js

```
it('fires observers just once upon registration', function() {
  var observerSpy = jasmine.createSpy();
  var injector = makeInjectorWithDirectives({
    myDirective: function() {
      return {
        link: function(scope, element, attrs) {
          attrs.$observe('alt', observerSpy);
```



```
        }
      };
    }
  });
  injector.invoke(function($compile, $rootScope) {
    var el = $('<img alt="{{myAltText}}" my-directive>');
    $compile(el)($rootScope);
    $rootScope.$apply();

    expect(observerSpy.calls.count()).toBe(1);
  });
});
```

在Attributes object中我们设置了，observers会在被注册之后的下一个digest被触发，现在我们想skip这个feature，而是在interpolation watcher设置了attribute value之后的下一个digest触发。方式是修改$observe方法，如果一个attribute包含了$$inter属性就跳过。
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.$$observers[key].push(fn);
  $rootScope.$evalAsync(function() {
    if (!self.$$observers[key].$$inter) {
      fn(self[key]);
    }
  });
  return function() {
    var index = self.$$observers[key].indexOf(fn);
    if (index >= 0) {
      self.$$observers[key].splice(index, 1);
    }
  };
};
```

在凭空生成的directive的link函数中设置这个flag。

```
return function link(scope, element, attrs) {
  attrs.$$observers = attrs.$$observers || {};
  attrs.$$observers[name] = attrs.$$observers[name] || [];
  attrs.$$observers[name].$$inter = true;
  scope.$watch(interpolateFn, function(newValue) {
    attrs.$set(name, newValue);
  });
};
```

2，interpolation应该在link之前发生，然而当前还不是，原因在于interpolation是在linking之后的first digest进行的，我们应该在任何digests之前就完成interpolate，所以要手动调用一次，attrs[name]=interpolateFn(scope)。这还不够，因为优先级设置为100，然而interpolation现在是在这个directive的post-link function中，在post-link functions里是reverse priority order发生的，因此反而变成最后进行的了。因此我们要手动修改为pre-link function，return function前面增加pre: key。
3，isolate scope bindings for attributes，@ attribute bindings是通过observers实现的，当前attribute bindings指向了interpolation之前的值，原因在于initializeDirectiveBindings是在directives之前发生的，所以也要手动调用$interpolate函数。
4，directives可能会在编译的时候去修改element attribute，而attribute interpolation function现在还没有发现这个修改，用的是修改之前的值。
test/compile_spec.js

```
it('is done for attributes so that compile-time changes apply', function() {
  var injector = makeInjectorWithDirectives({
    myDirective: function() {
      return {
        compile: function(element, attrs) {
          attrs.$set('myAttr', '{{myDifferentExpr}}');
        }
      };
    }
  });
  injector.invoke(function($compile, $rootScope) {
    var el = $('<div my-directive my-attr="{{myExpr}}"></div>');
    $rootScope.myExpr = 'Hello';
    $rootScope.myDifferentExpr = 'Other Hello';
    $compile(el)($rootScope);
    $rootScope.$apply();

    expect(el.attr('my-attr')).toEqual('Other Hello');
  });
});
```

attribute interpolation function是在编译的时候产生的，而这个替换因为优先级比较低，发生在编译过程的之后的时间。因此在attribute interpolation directive的link function中需要增加一个检查，如果compilation中更改了这个attribute value，就重新调用$interpolate函数产生interpolation function。如果remove了，就完全删掉。
5，event handlers中间不应该有interpolation，因为event handler是在Angular digest之外被触发的，所以不会接触到任何scope，所以没有作用。用正则表达式判断是否是on...这样的属性名，是的话抛出错误。

Optimizing Interpolation Watches With A Watch Delegate：优化interpolation result被构造的次数，方法是通过watch delegates，在part2里面介绍的feature。原理是expressions只会在至少一个value变化的时候才会变化，然而目前不管有没有变化，每次watch都会重新构造一个新的string。
可以在watch function上面添加$$watchDelegate属性，从而watcher会根据delegate判断是否有没有变化，而不是watch function。我们当前可以构造的watch delegate目标是在不重新生成字符串的前提下判断interpolated expressions是否有变化。
通过_.reduce方法在interpolateFn上增加$$watchDelegate属性。增加expressionFns数组用来存放所有生成的解析之后的expression functions。使用$watchGroup方法来监控这个数组。
src/interpolate.js

```
$$watchDelegate: function(scope, listener) {
  return scope.$watchGroup(expressionFns, function() {
    listener(_.reduce(parts, function(result, part) {
      if (_.isFunction(part)) {
        return result + stringify(part(scope));
      } else {
        return result + part;
      }
    }, ''));
  });
}
```

watch delegate目前还有一点不足，因为Listener应该被new values，old values，scope调用，目前只有new values。用一个变量存放old values。

```
$$watchDelegate: function(scope, listener) {
  var lastValue;
  return scope.$watchGroup(expressionFns, function(newValues, oldValues) {
    var newValue = compute(scope);
    listener(
      newValue,
      (newValues === oldValues ? newValue : lastValue),
      scope
    );
    lastValue = newValue;
  });
}
```

还可以做的一点优化是，newValues目前并没有用上，每个newValue还是compute(scope)得来的，也就是说和之前$interpolate的循环里面一样，总共重复计算了两遍。优化是重写compute函数，原来是evaluating expression functions，现在则是处理precomputed values，将parts数组里面原来的expression替换成stringily(value)。

making interpolation symbols configurable：开始和结束的符号是可以更改的，增加startSymbol和endSymbol接口来更改这两个符号。同样关于unescaping需要更改，但是现在是动态的形成正则表达式了，使用RegEx constructor。
src/interpolate.js

```
this.$get = ['$parse', function($parse) {
  var escapedStartMatcher =
    new RegExp(startSymbol.replace(/./g, escapeChar), 'g');
  var escapedEndMatcher   =
    new RegExp(endSymbol.replace(/./g, escapeChar), 'g');

  // ...

}];
```

escapeChar是在字符前面加三个\，每个\又需要escape，所以是'\\\\\\'+char。
因为代码可能需要和第三方的代码一起使用，所以在$compile中，需要denormalize所有的directive templates，将所有默认的'{{'和'}}'都替换成自定义的符号。

Summary
Expression interpolation is a hugely important feature in Angular. It's pretty much the first thing taught to new Angular developers, and no Angular application can get very far with it. The implementation of the feature itself isn't hugely complicated, since it builds on the expression parsing in `$parse`, the watchers on Scopes, as well as the directive implementation provided by `$compile`. For us, it serves as a nice demonstration to how these low-level features can be combined to produce something very useful.
We also learned some tricks on how watch delegates and watch groups can be used to eliminate work that would otherwise have to be done during change detection. Similar tricks could be applied in application code as well.
In this chapter you have learned:
• That there is an `$interpolate` service with which you can turn any string into an interpolation function.
• How the service parses the expressions contained in the string, and evaluates them when the interpolation function is called.
• How expression results are stringified before they are concatenated into the resulting string.
• That you can escape the start and end symbols if you want to include them literally in your UI: \{\{ and \}\}.
• That Angular documentation actually recommends doing this escaping for all server-provided dynamic content, as an additional cross-site scripting forgery prevention measure.
• How you can instruct `$interpolate` to skip creating an interpolation function if there are no dynamic expressions in the string, using the `mustHaveExpressions` flag.
• How interpolation integrates into text nodes: Directives are generated for text nodes on the fly. The directives watch the interpolation expressions and update the node contents.
• That the `hg-binding` classes and `$binding` data is added to the parents of text nodes
to aid in tooling and introspection.
• How interpolation integrates into element attributes: Directives are generated on the fly. The directives watch the interpolation expressions and update the attribute values.
• How attribute interpolation integrates with attribute observers.
• That Angular does extra work to make sure interpolation is done before other directives for the element are linked.
• How Angular prevents you from doing interpolation in standard DOM event listeners.
• How interpolation watching is optimized with watch delegates to try to minimise the work needed to check for changes.
• How you can configure your own custom interpolation start and end symbols to replace {{ and }}.
• That Angular denormalizes directive templates to enable code reuse even when you reconfigure your interpolation symbols.

最后一章：Bootstrapping Angular
ngClick Directive：现在理论上具有实现任何一个directive的能力了，这里以ngClick为例。
ngClick需要做以下的事情：
1，点击的时候进行一次Angular digest。在link function里面定义。
2，点击的时候执行定义的函数。从Attribute object得到ng-click的value，使用scope.$apply(attires.ngClick)，在当前scope执行这个expression。
3，使得click event object对于该expression可用，作为$event对象。实现时将这个DOM event作为一个local变量$event，因为scope.$apply不能传local，所以用scope.$eval执行。
src/directives/ng_click.js

```
function ngClickDirective() {
  return {
    restrict: 'A',
    link: function(scope, element, attrs) {
      element.on('click', function(evt) {
        scope.$eval(attrs.ngClick, {$event: evt});
        scope.$apply();
      });
    }
  };
}
```

Bootstrapping Angular applications manually：
Bootstrapping分为两种方式，手动或者自动，手动是调用Angular Javascript APIs，自动则是添加ng-app属性让Angular去找并且自动启动。
manual bootstrapping可以通过angular.bootstrap()函数进行。
src/bootstrap.js

```
'use strict';

var $ = require('jquery');
var publishExternalAPI = require('./angular_public');

publishExternalAPI();

window.angular.bootstrap = function() {
};
```

函数内该做些什么呢?
a. dependency injection setup
1，使用createInjector()创建一个injector，并且作为函数的返回值。2，一定有一个HTML element作为应用的root element，需要作为参数，bootstrap(element)，函数内需要将injector绑定到这个element上。（主要是为了tooling and debugging)。3，加载所有Angular built-in services，createInjector(['ng'])。4，用户会有自己的modules，这些modules是bootstrap函数的第二个参数，是一个module names的array。这些module之前应当已经用angular.module方法注册了，所以调用createInjector(modules)即可。5，bootstrap还制造了一个特殊的dependency，$rootElement，用来作为injection的目标。实现是注册一个function module，并unshift到modules数组里的第一项。

b. compile the DOM
1，使用$injector.invoke()调用$compile service，编译该$element。2，compile之后马上执行返回的函数进行link。

c. run the very first digest
1，在初始的DOM进行compile和link之后，interpolation应当已经完成。将之前的compiling and linking包装到$rootScope.$apply()，确保在linking之后会进行一次digest。

d. 支持strict dependency injector mode，作用是是的没有包装在arrays 或者没有使用$inject进行annotate的injections都会抛出错误。bootstrap增加第三个参数config object，包含strictDi flag。
src/bootstrap.js

```
window.angular.bootstrap = function(element, modules, config) {
  var $element = $(element);
  modules = modules || [];
  config = config || {};
  modules.unshift(['$provide', function($provide) {
    $provide.value('$rootElement', $element);
  }]);
  modules.unshift('ng');
  var injector = createInjector(modules, config.strictDi);
  $element.data('$injector', injector);
  injector.invoke(['$compile', '$rootScope', function($compile, $rootScope) {
    $rootScope.$apply(function() {
      $compile($element)($rootScope);
    });
  }]);
  return injector;
};
```

至此完成了整个Angular application bootstrap process。

Bootstrapping Angular Applications Automatically：
1，使用jQuery DOM ready callback，寻找ng-app属性（或者另外几个同等作用的），使用CSS attribute selecotr来定位这些属性。可能会有属性值作为module name，同样可以得到。不过和manual bootstrapping不同，这里只能定义一个module name，其他的module都需要被定义为module dependencies。
2，找到了这个element之后，调用bootstrap函数即可。
src/bootstrap.js

```
var ngAttrPrefixes = ['ng-', 'data-ng-', 'ng:', 'x-ng-'];
$(document).ready(function() {
  var foundAppElement, foundModule, config = {};
  _.forEach(ngAttrPrefixes, function(prefix) {
    var attrName = prefix + 'app';
    var selector = '[' + attrName.replace(':', '\\:') + ']';
    var element;
    if (!foundAppElement &&
        (element = document.querySelector(selector))) {
      foundAppElement = element;
      foundModule = element.getAttribute(attrName);
    }
  });
  if (foundAppElement) {
    config.strictDi = _.any(ngAttrPrefixes, function(prefix) {
      var attrName = prefix + 'strict-di';
      return foundAppElement.hasAttribute(attrName);
    });
    window.angular.bootstrap(
      foundAppElement,
      foundModule ? [foundModule] : [],
      config
    );
  }
});
```

这样完成了auto-bootstrap implementation。

使用browserify构造Production bundle，使用UglifyJS来minify，然后就可以使用了，哈哈哈哈哈哈完工了。

Summary
We have come a long way! Out of thin air, we have implemented our own version of AngularJS, which has all the core features the Angular framework itself has.
Even more importantly, you now have a deep and rich mental model of what the Angular framework does and how it works. You know everything about change detection, expressions, dependency injection, directives, controllers, transclusion, templates, interpolation, promises, and `$http`. You are a true Angular expert!
In this chapter you have learned:
• How the `ngClick` directive works.
• What the manual bootstrap process entails: Setting up an injector, compiling and linking the DOM, and running a digest.
• How automatic bootstrapping finds the `ng-app` DOM element automatically and then runs the bootstrap function automatically.
• How production bundles of the framework can be generated.
• How everything ties together and how all the code we've written supports actual "Angular light" applications.