http://www.angularjs.net.cn/tutorial/

AngularJS有着诸多特性，最为核心的是：MVVM、模块化、自动化双向数据绑定、语义化标签、依赖注入等等。

Angular是为了扩展HTML在构建应用时本应具备的能力而设计的。

Angular通过指令（directive）扩展HTML的语法。

例如：

- 通过`{{}}`进行数据绑定。
- 使用DOM控制结构来进行迭代或隐藏DOM片段。
- 支持表单和表单验证。
- 将逻辑代码关联到DOM元素上。
- 将一组HTML做成可重用的组件。

Angular的一些出众之处：

- 构建一个CRUD应用时可能用到的所有技术：数据绑定、基本模板指令、表单验证、路由、深度链接、组件重用、依赖注入。
- 可测试性：单元测试、端到端测试、模拟对象（mocks）、测试工具。
- 拥有一定目录结构和测试脚本的种子应用。

Angular通过给开发者呈现更高层次的抽象来简化应用的开发。和其他的抽象一样，它也以损失灵活性为代价。换句话说，Angular并不是适合任何应用的开发，Angular考虑的是构建CRUD应用。幸运的是，绝大多数WEB应用都是CRUD应用。为了理解Angular适用哪些场合，知道它不适合哪些场合是很有帮助的。

对于像游戏和有图形界面的编辑器之类的应用，会进行频繁且复杂的DOM操作，和CRUD应用不同。因此，可能不适合用Angular来构建。在这种场景下，使用更低抽象层次的类库可能会更好，例如：`jQuery`。

Angular 初始化过程

当初始的 **HTML** 文档被完全加载和解析完成之后，**DOMContentLoaded** 事件被触发，而无需等待样式表、图像和子框架的完成加载。

DOMContentLoaded DOM 已准备就绪

另一个不同的事件 `load` 应该仅用于检测一个完全加载的页面。

The `load` event is fired when a resource and its dependent resources have finished loading.

所以说一般情况下，`DOMContentLoaded`事件要在`window.onload`之前执行，当DOM树构建完成的时候就会执行DOMContentLoaded事件。当window.onload事件触发时，页面上所有的DOM，样式表，脚本，图片，flash都已经加载完成了。

如果你是个jQuery使用者，你可能会经常使用$(document).ready();或者(function(){})，这都是使用了DOMContentLoaded事件。

下面三个写法是等价的：

```
$(document).ready(handler)
$().ready(handler)(this is not recommended)
$(handler)
```

需要注意：

window.onload不能同时编写多个。

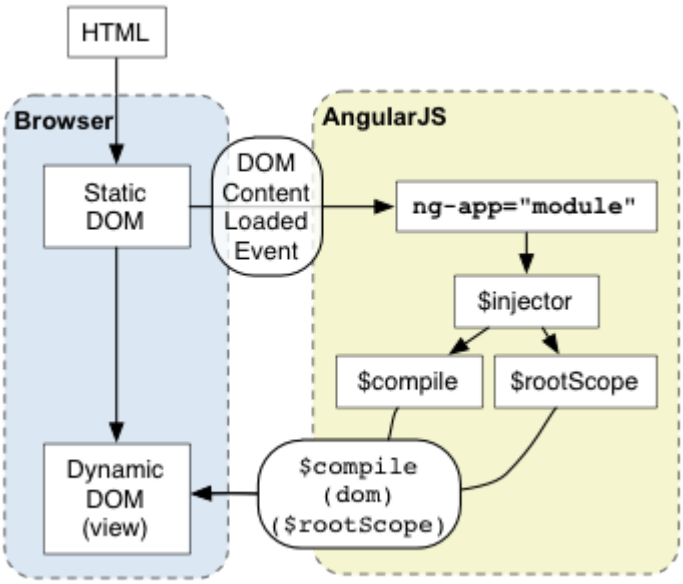
$(document).ready()可以同时编写多个。

```
//window.onload不能同时编写多个。
//以下代码无法正确执行，结果只输出第二个。
window.onload = function(){
  alert("test1");
};

window.onload = function(){
  alert("test2");
};

//$(document).ready()能同时编写多个
//结果两次都输出
$(document).ready(function(){
   alert("Hello World");
});
$(document).ready(function(){
   alert("Hello again");
});
```

自动初始化

Angular 在以下两种情况下自动初始化，一个是在 `DOMContentLoaded` 事件触发时，或者在 `angular.js` 脚本被执行的同时如果 `document.readyState` 被置为 `'complete'` 的话。初始化时，Angular 会去找 `ng-app` 这个指明应用开始所在的指令。如果 `ng-app` 指令被找到的话，Angular 会做以下几件事：

- 加载 `ng-app` 指令所指定的 模块
- 创建应用所需的 `injector`
- 以 `ng-app` 所在的节点为根节点，开始遍历并编译DOM树（ `ng-app` 指出了应用的哪一部份开始时 Angular 去编译的）



AngularJS 概念概述

概念	说明
模板(Template)	带有Angular扩展标记的**HTML**
指令(Directive)	用于通过**自定义属性和元素**扩展HTML的行为
模型(Model)	用于显示给用户并且与用户互动的**数据**
作用域(Scope)	用来存储模型(Model)的语境(context)。模型放在这个语境中才能被控制器、指令和表达式等访问到
表达式(Expression)	模板中可以通过它来访问作用域（Scope）中的变量和函数
编译器(Compiler)	用来**编译**模板(Template)，并且对其中包含的指令(Directive)和表达式(Expression)进行**实例化**
过滤器(Filter)	负责**格式化**表达式(Expression)的值，以便呈现给用户
视图(View)	用户看到的内容（即**DOM**）
数据绑定(Data Binding)	**自动同步**模型(Model)中的数据和视图(View)表现
控制器(Controller)	视图(View)背后的**业务逻辑**
依赖注入(Dependency Injection)	负责创建和自动装载对象或函数
注入器(Injector)	用来实现依赖注入(Injection)的容器
模块(Module)	用来配置注入器
服务(Service)	独立于视图(View)的、**可复用**的业务逻辑

```
<!-- index.html -->

<!doctype html>
<html ng-app>
  <head>
    <script src="http://code.angularjs.org/1.2.25/angular.min.js"></script>
  </head>
  <body>
    <div ng-init="qty=1;cost=2">
      <b>订单:</b>
      <div>
        数量: <input type="number" ng-model="qty" required >
      </div>
      <div>
        单价: <input type="number" ng-model="cost" required >
      </div>
      <div>
        <b>总价:</b>
      </div>
    </div>
  </body>
</html>
```

接下来我们将通读这个例子，并且详细讲解这里所发生的事情。这看起来很像标准的HTML，只是带了一些新的标记。在Angular中，像这样的文件叫做“模板(template)”。 当Angular启动你的应用时，它通过“编译器(compiler)”来解析并处理模板中的这些新标记。 这些经过加载、转换、渲染而成的DOM就叫做“视图(view)”

第一类新标记叫做“指令(directive)”。 它们通过HTML中的属性或元素来为页面添加特定的行为。上面的例子中，我们使用了`ng-app`属性，与此相关的指令(directive)则负责自动初始化我们的应用程序。 Angular还为`input`元素定义了一个指令，它负责添加额外的行为到这个元素上。例如，当它发现了`input`元素的`required`属性，它就会自动进行验证，确保所输入的文本不会是空白。 `ng-model`指令则负责从变量(比如这里的qty、cost等)加载`input`元素的`value`值，并且把`input`元素的`value`值写回变量中。并且，还会根据对`input`元素进行校验的结果自动添加相应的css类。 比如在上面这个例子中，我们就通过这些css类把空白`input`元素的边框设置为红色，来表示无效的输入。

第二类新标记是双大括号`{{ expression | filter }}`，其中expression是“表达式”语句，filter是“过滤器”语句。当编译器遇到这种标记时，它会把这些标记替换为这个表达式的计算结果。 模板中的"表达式"是一种**类似于**JavaScript的代码片段，它允许你读写变量。注意，表达式中所用的变量**并不是**全局变量。 就像JavaScript函数定义中的变量都属于某个作用域一样，Angular也为这些能从表达式中访问的变量提供了一个“作用域(scope)”。 这些存储于Angular作用域(Scope)中的变量叫做Scope变量，这些变量所代表的数据叫做“模型(model)”。 在上面的例子中，这些标记告诉Angular：“从这两个`input`元素中获取数据，并把它们乘在一起。”

上面这个例子中还包含一个"过滤器(filter)"。 过滤器格式化表达式的值，以便呈现给用户。 上面的例子中`currency`过滤器把一个数字格式化为金额的形式进行输出。

这个例子中最重要的一点是：Angular提供了**动态(live)**的绑定： 当`input`元素的值变化的时候，表达式的值也会自动重新计算，并且DOM所呈现的内容也会随着这些值的变化而自动更新。 这种模型(model)与视图(view)的联动就叫做“双向数据绑定”。

添加UI逻辑：控制器

index.html

```
<!doctype html>
<html ng-app="invoice1">
  <head>
    <script src="http://code.angularjs.org/1.2.25/angular.min.js"></script>
    <script src="invoice1.js"></script>
  </head>
  <body>
    <div ng-controller="InvoiceController as invoice">
      <b>订单:</b>
      <div>
        数量: <input type="number" ng-model="invoice.qty" required >
      </div>
      <div>
        单价: <input type="number" ng-model="invoice.cost" required >
        <select ng-model="invoice.inCurr">
          <option ng-repeat="c in invoice.currencies"></option>
        </select>
      </div>
      <div>
        <b>总价:</b>
        <span ng-repeat="c in invoice.currencies">

        </span>
        <button class="btn" ng-click="invoice.pay()">支付</button>
      </div>
    </div>
  </body>
</html>
```

invoice1.js

```

angular.module('invoice1', [])
  .controller('InvoiceController', function() {
    this.qty = 1;
    this.cost = 2;
    this.inCurr = 'EUR';
    this.currencies = ['USD', 'EUR', 'CNY'];
    this.usdToForeignRates = {
      USD: 1,
      EUR: 0.74,
      CNY: 6.09
    };

    this.total = function total(outCurr) {
      return this.convertCurrency(this.qty * this.cost, this.inCurr, outCurr);
    };
```

```
    this.convertCurrency = function convertCurrency(amount, inCurr, outCurr) {
      return amount * this.usdToForeignRates[outCurr] * 1 / this.usdToForeignRates[inCurr];
    };
    this.pay = function pay() {
      window.alert("谢谢！");
    };
  });
```

这次有什么变动？

首先，出现了一个新的JavaScript文件，它包含一个被称为"控制器(controller)"的函数。 更确切点说：这个文件中定义了一个构造函数，它用来在将来真正需要的时候创建这个控制器函数的实例。 控制器的用途是导出一些变量和函数，供模板中的表达式(expression)和指令(directive)使用。

在创建一个控制器的同时，我们还往HTML中添加了一个`ng-controller`指令。 这个指令告诉Angular，我们创建的这个`InvoiceController`控制器将会负责管理这个带有ng-controller指令的div节点，及其各级子节点。

`InvoiceController as invoice`这个语法告诉Angular：创建这个`InvoiceController`的实例，并且把这个实例赋值给当前作用域(Scope)中的`invoice`变量。

同时，我们修改了页面中所有用于读写Scope变量的表达式，给它们加上了一个`invoice.`前缀。 我们还把可选的币种作为一个数组定义在控制器中，并且通过`ng-repeat`指令把它们添加到模板。 由于控制器中还包含了一个`total`函数，我们也能在DOM中使用`{{invoice.total(...) }}`表达式来绑定总价的计算结果。

同样，这个绑定也是动态(live)的，也就是说：当invoice.total函数的返回值变化的时候，DOM也会跟着自动更新。表单中的"支付"按钮附加上了指令`ngClick`。 这意味着当它被点击时，会自动执行`invoice.pay()`这个表达式，即：调用当前作用域中的pay函数。

在这个新的JavaScript文件中，我们还创建了一个模块(module)，并且在这个模块中注册了控制器(controller)。 接下来我们就讲一下模块(module)这个概念。

与视图(View)无关的业务逻辑：服务(Service)

现在，`InvoiceController`包含了我们这个例子中的所有逻辑。如果这个应用程序的规模继续成长，最好的做法是：把控制器中与视图无关的逻辑都移到"服务(service)"中。 以便这个应用程序的其他部分也能复用这些逻辑。

接下来，就让我们重构我们的例子，并且把币种兑换的逻辑移入到一个独立的服务(service)中。

index.html finance2.js invoice2.js

```
<!doctype html>
<html ng-app="invoice2">
  <head>
    <script src="http://code.angularjs.org/1.2.25/angular.min.js"></script>
    <script src="finance2.js"></script>
    <script src="invoice2.js"></script>
  </head>
  <body>
    <div ng-controller="InvoiceController as invoice">
      <b>订单:</b>
      <div>
        数量: <input type="number" ng-model="invoice.qty" required >
      </div>
      <div>
        单价: <input type="number" ng-model="invoice.cost" required >
        <select ng-model="invoice.inCurr">
          <option ng-repeat="c in invoice.currencies"></option>
        </select>
      </div>
      <div>
        <b>总价:</b>
        <span ng-repeat="c in invoice.currencies">

        </span>
        <button class="btn" ng-click="invoice.pay()">支付</button>
      </div>
    </div>
  </body>
</html>
```

index.html finance2.js invoice2.js

```
angular.module('finance2', [])
  .factory('currencyConverter', function() {
    var currencies = ['USD', 'EUR', 'CNY'],
        usdToForeignRates = {
      USD: 1,
      EUR: 0.74,
      CNY: 6.09
    };
    return {
      currencies: currencies,
      convert: convert
    };

    function convert(amount, inCurr, outCurr) {
      return amount * usdToForeignRates[outCurr] * 1 / usdToForeignRates[inCurr];
    }
  });
```

index.html finance2.js invoice2.js

```
angular.module('invoice2', ['finance2'])
  .controller('InvoiceController', ['currencyConverter', function(currencyConverter) {
    this.qty = 1;
    this.cost = 2;
    this.inCurr = 'EUR';
    this.currencies = currencyConverter.currencies;

    this.total = function total(outCurr) {
      return currencyConverter.convert(this.qty * this.cost, this.inCurr, outCurr);
    };
    this.pay = function pay() {
      window.alert("谢谢！");
    };
  }]);
```

这次有什么改动？

我们把 `convertCurrency` 函数和所支持的币种的定义独立到一个新的文件： `finance.js` 。但是控制器怎样才能找到这个独立的函数呢？

这下该"依赖注入(Dependency Injection)"出场了。依赖注入(DI)是一种设计模式(Design Pattern)，它用于解决下列问题：我们创建了对象和函数，但是它们怎么得到自己所依赖的对象呢？ Angular中的每一样东西都是用依赖注入(DI)的方式来创建和使用的，比如指令(Directive)、过滤器(Filter)、控制器(Controller)、服务(Service)。 在Angular中，依赖注入(DI)的容器(container)叫做"注入器(injector)"。

要想进行依赖注入，你必须先把这些需要协同工作的对象和函数注册(Register)到某个地方。在Angular中，这个地方叫做"模块(module)"。

在上面这个例子中： 模板包含了一个 `ng-app="invoice2"` 指令。这告诉Angular使用 `invoice2` 模块作为该应用程序的主模块。 像 `angular.module('invoice', ['finance'])` 这样的代码告诉Angular： `invoice` 模块依赖于 `finance` 模块。 这样一来，Angular就能同时使用 `InvoiceController` 这个控制器和 `currencyConverter` 这个服务了。

刚才我们已经定义了应用程序的所有部分，现在，需要Angular来创建它们了。 在上一节，我们了解到控制器(controller)是通过一个工厂函数创建的。 而对于服务(service)，则有多种方式来定义它们的工厂函数（参见服务指南）。 上面这个例子中，我们通过一个返回 `currencyConverter` 函数的函数作为创建 `currencyConverter` 服务的工厂。 （译注：js中的"工厂(factory)"是指一个以函数作为"返回值"的函数）

回到刚才的问题： `InvoiceController` 该怎样获得这个 `currencyConverter` 函数的引用呢？ 在Angular中，这非常简单，只要在构造函数中定义一些具有特定名字的参数就可以了。 这时，注入器(injector)就可以按照正确的依赖关系创建这些对象，并且根据名字把它们传入那些依赖它们的对象工厂中。 在我们的例子中， `InvoiceController` 有一个叫做 `currencyConverter` 的参数。 根据这个参数，Angular就知道 `InvoiceController` 依赖于 `currencyConverter` ，取得 `currencyConverter` 服务的实例，并且把它作为参数传给 `InvoiceController` 的构造函数。

这次改动中的最后一点是我们现在把一个数组作为参数传入到 `module.controller` 函数中，而不再是一个普通函数。 这个数组前面部分的元素包含这个控制器所依赖的一系列服务的名字，最后一个元素则是这个控制器的构造函数。 Angular可以通过这种数组语法来定义依赖，以免js代码压缩器(Minifier)破坏这个"依赖注入"的过程。 因为这些js代码压缩器通常都会把构造函数的参数重命名为很短的名字，比如 `a` ，而常规的依赖注入是需要根据参数名来查找"被注入对象"的。 （译注：因为字符串不会被js代码压缩器重命名，所以数组语法可以解决这个问题。）

访问后端

现在开始最后一个改动：通过Yahoo Finance API来获得货币之间的当前汇率。 下面的例子将告诉你在Angular中应该怎么做。

index.html invoice3.js finance3.js

```
angular.module('finance3', [])
  .factory('currencyConverter', ['$http', function($http) {
    var YAHOO_FINANCE_URL_PATTERN =
          'http://query.yahooapis.com/v1/public/yql?q=select * from '+
          'yahoo.finance.xchange where pair in ("PAIRS")&format=json&'+
          'env=store://datatables.org/alltableswithkeys&callback=JSON_CALLBACK',
        currencies = ['USD', 'EUR', 'CNY'],
        usdToForeignRates = {};
    refresh();
    return {
      currencies: currencies,
      convert: convert,
      refresh: refresh
    };

    function convert(amount, inCurr, outCurr) {
      return amount * usdToForeignRates[outCurr] * 1 / usdToForeignRates[inCurr];
    }

    function refresh() {
      var url = YAHOO_FINANCE_URL_PATTERN.
                replace('PAIRS', 'USD' + currencies.join('","USD"));
      return $http.jsonp(url).success(function(data) {
        var newUsdToForeignRates = {};
        angular.forEach(data.query.results.rate, function(rate) {
          var currency = rate.id.substring(3,6);
          newUsdToForeignRates[currency] = window.parseFloat(rate.Rate);
        });
        usdToForeignRates = newUsdToForeignRates;
      });
    }
  }]);
```

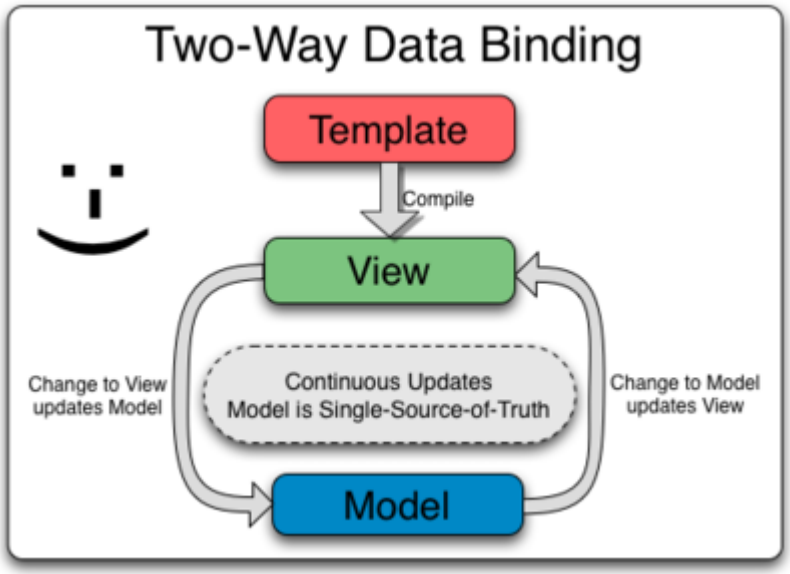
这次有什么变动？

这次我们的`finance`模块中的`currencyConverter`服务使用了`$http`服务 —— 它是由Angular内建的用于访问后端API的服务。 是对`XMLHttpRequest`以及JSONP的封装。详情参阅$http的API文档 — [‘http`] (http://www.angularjs.net.cn/api/105.html)。

AngularJS 数据绑定（ Data Binding ）

在Angular网页应用中，数据绑定是数据模型(model)与视图(view)组件的自动同步。Angular的实现方式允许你把应用中的模型看成单一数据源。而视图始终是数据模型的一种展现形式。当模型改变时，视图就能反映这种改变，反之亦然。

Angular模板中的数据绑定



Angular模板的工作方式如图表中所示。差别体现在：

其一，模板（指未经编译的附加了标记和指令的HTML）是在浏览器中编译的。

其二，编译阶段生成了动态(live)视图。

保持视图动态的原因是，任何视图中的改变都会立刻反映到数据模型中去，任何数据模型的改变都会传播到视图中去。这使得数据模型始终是应用的单一数据源。大幅度简化了开发者的编程核心，你可以将视图简单的理解为数据模型的实时映射。

因此，将视图作为数据模型的一种映射，使得控制器完全与视图分离，而不必关心视图的展现。这使测试变得小菜一碟，你可以在没有视图和有关DOM/浏览器的依赖情况下轻松测试你的应用。

1. 关于angular的双向绑定需要理解angular的dirty check.
2. 双向绑定则引入angular的watch,一个angular页面理想状况为200左右的watch，一般大家默认2000watch为上限(IE),这是为了页面更好的体验效果，而并不意味着一定是angular dirty check上限。
3. 过多的watch，我们需要考虑页面的性能，对于表格推荐采用服务端分页,按屏加载ngInfiniteScroll,以及移除watch:angularjs移除不必要的watch.

AngularJS 控制器(Controllers)

在Angular中，控制器就像 JavaScript 中的**构造函数**一般，是用来增强 Angular作用域(scope) 的。

当一个控制器通过 `ng-controller` 指令被添加到DOM中时，ng 会调用该控制器的**构造函数**来生成一个控制器对象，这样，就创建了一个新的**子级 作用域(scope)**。在这个构造函数中，作用域(scope)会作为 `$scope` 参数注入其中，并允许用户代码访问它。

一般情况下，我们使用控制器做两件事：

- 初始化 `$scope` 对象
- 为 `$scope` 对象添加行为（方法）

初始化 `$scope` 对象

当我们创建应用程序时，我们通常要为Angular的 `$scope` 对象设置初始状态，这是通过在 `$scope` 对象上添加属性实现的。这些属性就是供在视图中展示用的**视图模型**（**view model**），它们在与此控制器相关的模板中均可以访问到。

下面的例子中定义了一个非常简单的控制器构造函数：`GreetingCtrl`，我们在该控制器所创建的 scope 中添加一个 `greeting` 属性：

```
function GreetingCtrl($scope) {
    $scope.greeting = 'Hola!';
}
```

如上所示，我们有了一个控制器，它初始化了一个 `$scope` 对象，并且有一个 `greeting` 属性。当我们把该控制器关联到DOM节点上，模板就可以通过数据绑定来读取它：

```
<div ng-controller="GreetingCtrl">

</div>
```

注意：虽然Angular允许我们在全局作用域下(window)定义控制器函数，但**建议不要**用这种方式。在一个实际的应用程序中，推荐在 Angular模块 下通过 `.controller` 为你的应用创建控制器，如下所示：

```
var myApp = angular.module('myApp',[]);
myApp.controller('GreetingCtrl', ['$scope', function($scope) {
    $scope.greeting = 'Hola!';
}]);
```

在上面例子中，我们使用**内联注入**的方式声明 `GreetingCtrl` 依赖于Angular提供的 `$scope` 服务。更多详情，参阅 依赖注入 。

为 `$scope` 对象添加行为

为了对事件作出响应，或是在视图中执行计算，我们需要为 scope 提供相关的行为操作的逻辑。上面一节中，我们为 scope 添加属性来让模板可以访问数据模型，现在，我们为 `$scope` 添加方法来让它提供相关的交互逻辑。添加完之后，这些方法就可以在**模板/视图**中被调用了。

下面的例子将演示为控制器的 scope 添加方法，它用来使一个数字翻倍：

```
var myApp = angular.module('myApp',[]);
myApp.controller('DoubleCtrl', ['$scope', function($scope) {
    $scope.double = function(value) { return value * 2; };
}]);
```

当上述控制器被添加到DOM之后，`double` 方法即可被调用，如在模板中的一个Angular表达式中：

```
<div ng-controller="DoubleCtrl">
    <input ng-model="num"> 翻倍后等于
</div>
```

如 概述 部分所指出的一样，任何对象（或者原生类型的变量）被添加到 scope 后都将成为 scope 的属性，作为数据模型供模板/视图调用。任何方法被添加到 scope 后，也能在模板/视图中通过Angular表达式或是Angular的事件处理器（如：`ngClick`）调用。

正确使用控制器

通常情况下，控制器不应被赋予太多的责任和义务，它只需要负责一个单一视图所需的业务逻辑。

最常见的保持控制器"纯度"的方法是将那些不属于控制器的逻辑都封装到服务（services）中，然后在控制器中通过依赖注入调用相关服务。详见指南中的 依赖注入 服务 这两部分。

注意，下面的场合**千万不要用控制器**：

- 任何形式的DOM操作：控制器只应该包含业务逻辑。DOM操作则属于应用程序的表现层逻辑操作，向来以测试难度之高闻名于业界。把任何表现层的逻辑放到控制器中将会大大增加业务逻辑的测试难度。ng 提供数据绑定（数据绑定）来实现自动化的DOM操作。如果需要手动进行DOM操作，那么最好将表现层的逻辑封装在 指令 中
- 格式化输入：使用 angular表单控件 代替
- 过滤输出：使用 angular过滤器 代替
- 在控制器间复用有状态或无状态的代码：使用angular服务 代替
- 管理其它部件的生命周期（如手动创建 service 实例）

将控制器与 scope 对象关联

通过两种方法可以实现控制器和 scope 对象的关联：

- `ngController` 指令 这个指令就会创建一个新的 scope
- $route路由服务

Scope 继承

我们常常会在不同层级的DOM结构中添加控制器。由于 `ng-controller` 指令会创建新的子级 scope ，这样我们就会获得一个与DOM层级结构相对应的的基于继承关系的 scope 层级结构。（译者注：由于 Js 是基于原型的继承，所以）底层（内层）控制器的 `$scope` 能够访问在高层控制器的 scope 中定义的属性和方法。详情参见 理解"作用域"。

译者注：下面是一个拥有三层div结构，也就对应有三层 scope 继承关系的层级结构（不包括 rootScope 的话），demo中的蓝色边框很清晰的展现了 scope 的层级和DOM层级的对应关系。它还展示了"scope 是由 `ng-controller` 指令创建并由其对应的控制器所管理"这个概念。

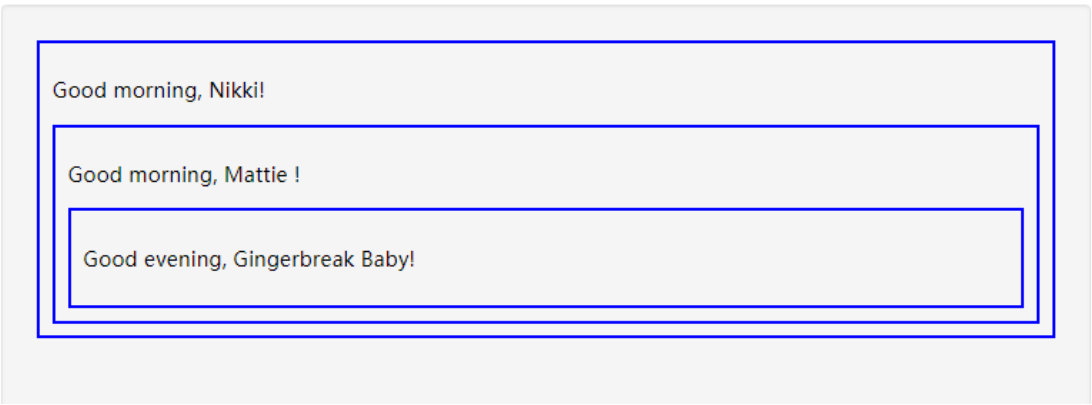
index.html style.css script.js

```html
<!doctype html>
<html ng-app="scopeInheritance">
  <head>
    <script src="http://code.angularjs.org/1.2.25/angular.min.js"></script>
    <script src="script.js"></script>
  </head>
  <body>
    <div ng-app="scopeInheritance" class="spicy">
      <div ng-controller="MainCtrl">
        <p>Good {{timeOfDay}}, {{name}}!</p>

        <div ng-controller="ChildCtrl">
          <p>Good {{timeOfDay}}, {{name}} !</p>

          <div ng-controller="GrandChildCtrl">
            <p>Good {{timeOfDay}}, {{name}}!</p>
          </div>
        </div>
      </div>
    </div>
  </body>
</html>
```

index.html style.css script.js

```css
div.spicy div {
  padding: 10px;
  border: solid 2px blue;
}
```

效果



index.html style.css script.js

```javascript
var myApp = angular.module('scopeInheritance', []);
myApp.controller('MainCtrl', ['$scope', function($scope){
  $scope.timeOfDay = 'morning';
  $scope.name = 'Nikki';
}]);
myApp.controller('ChildCtrl', ['$scope', function($scope){
  $scope.name = 'Mattie';
}]);
myApp.controller('GrandChildCtrl', ['$scope', function($scope){
  $scope.timeOfDay = 'evening';
  $scope.name = 'Gingerbreak Baby';
}]);
```

注意，上面例子中我们在HTML模板中嵌套了三个 `ng-controller` 指令，这导致我们的视图中有4个 scope：

- root scope，所有作用域的“根”
- `MainCtrl` 控制器管理的 scope（简称 `MainCtrl` scope），拥有 `timeOfDay` 和 `name` 两个属性
- `ChildCtrl` 控制器管理的 scope（简称 `ChildCtrl` scope），继承了 `MainCtrl` scope 中的 `timeOfDay` 属性，但重写了它的 `name` 属性
- `GrandChildCtrl` 控制器管理的 scope（简称 `GrandChildCtrl` scope），重写了 `MainCtrl` scope 中的 `timeOfDay` 属性和 `ChildCtrl` scope 中的 `name` 属性

控制器中，方法继承和属性继承的工作方式是一样的，所以，上面例子中的所有属性，我们也可以改写成能够返回字符串值的方法，同样有效。

AngularJS 作用域(Scope)

`作用域(Scope)`

- 是一个存储应用数据模型的对象
- 为 表达式 提供了一个执行上下文
- 作用域的层级结构对应于 DOM 树结构
- 作用域可以监听 表达式 的变化并传播事件

作用域有什么

- 作用域提供了（`$watch`）方法监听数据模型的变化
- 作用域提供了（`$apply`）方法把不是由Angular触发的数据模型的改变引入Angular的控制范围内（如控制器，服务，及Angular事件处理器等）
- 作用域提供了基于原型链继承其父作用域属性的机制，就算是嵌套于独立的应用组件中的作用域也可以访问共享的数据模型（这个涉及到指令间嵌套时作用域的几种模式）
- 作用域提供了 表达式 的执行环境，比如像 `{{username}}` 这个表达式，必须得是在一个拥有属性这个属性的作用域中执行才会有意义，也就是说，作用域中可能会像这样 `scope.username` 或是 `$scope.username`，至于有没有 $ 符号，看你是在哪里访问作用域了

作用域作为数据模型使用

作用域是Web应用的控制器和视图之间的粘结剂。在Angular中，最直观的表现是：在自定义指令中，处在模版的 链接(linking) 阶段时， `指令(directive)` 会设置一个 `$watch` 函数监听着作用域中各表达式（注：这个过程是隐式的）。这个 `$watch` 允许指令在作用域中的属性变化时收到通知， 进而让指令能够根据这个改变来对DOM进行重新渲染，以便更新已改变的属性值（注：属性值就是scope对象中的属性，也就是数据模型）。

其实，不止上面所说的指令拥有指向作用域的引用，控制器中也有（注：可以理解为控制器与指令均能引用到与它们相对应的DOM结构所处的作用域）。 但是控制器与指令是相互分离的，而且它们与视图之间也是分离的，这样的分离，或者说耦合度低，可以大大提高对应用进行测试的工作效率。

注：其实可以很简单地理解为有以下两个链条关系：

- 控制器 --> 作用域 --> 视图（DOM）
- 指令 --> 作用域 --> 视图（DOM）

作用域(scope)对象以及其属性是视图渲染的唯一数据来源。

作用域分层结构

如上所说，作用域的结构对应于DOM结构，那么最顶层，和DOM树有根节点一样，每个Angular应用有且仅有一个 `root scope` ，当然啦，子级作用域就和DOM树的子节点一样，可以有多个的。

应用可以拥有多个作用域，比如 指令 会创建子级作用域（至于指令创建的作用域是有多种类型的，详情参加指令相关文档）。一般情况下，当新的作用域被创建时，它是以嵌入在父级作用域的子级的形式被创建的，这样就形成了与其所关联的DOM树相对应的一个作用域的树结构。（译注：作用域的层级继承是基于原型链的继承，所以在下面的例子中会看到，读属性时会一直往上溯源，直到有未知）

作用域的分层的一个简单例子是，假设现在HTML视图中有一个表达式 `{{name}}` ，正如上面解释过，Angular需要经历取值和计算两个阶段才能最终在视图渲染结果。那么这个取值的阶段，其实就是根据作用域的这个层级结构（或树状结构）来进行的。

- 首先，Angular在该表达式当前所在的DOM节点所对应的作用域中去找有没有 `name` 这个属性
- 如果有，Angular返回取值，计算渲染；如果在当前作用域中没有找到，那么Angular继续往上一层的父级作用域中去找 `name` 属性，直到找到为止，最后实在没有，那就到达 `$rootScope` 了

从DOM中抓取作用域

作用域对象是与指令或控制器等Angular元素所在的DOM节点相关联的，也就是说，其实DOM节点上是可以抓取到作用域这个对象的（当然，为了调试偶尔会用，一般不用）。 而对于 `$rootScope` 在哪里抓呢？它藏在 `ng-app` 指令所在的那个DOM节点之中，请看更多关于 `ng-app` 指令。通常， `ng-app` 放在 `<html>` 标签中, 当然，如果你的应用中只是视图的某一部分想要用Angular控制，那你可以把它放在想要控制的元素的最外层。

那来看看如何在调试的时候抓取作用域吧：

1. 右键选去你想审查的元素，调出debugger，通常F12即可，这样你选中的元素会高亮显示（*译注：文档都看到这的人了，会需要这句提示么？原文档这是在卖萌么*）
2. 此时，调试器（debugger）允许你用变量 `$0` 来获取当前选取的元素
3. 在console中执行 `angular.element($0).scope()` 或直接输入 $scope 即可看到你想要查询的当前DOM元素节点绑定的作用域了

基于作用域的事件传播

作用域可以像DOM节点一样，进行事件的传播。主要是有两个方法：

- `broadcasted` ：从父级作用域广播至子级 scope
- `emitted` ：从子级作用域往上发射到父级作用域

让我们来看个例子：

作用域的生命周期

作用域的执行上下文

译注：这个小节应该是在看完下个小节的基础上再回过来看这个，所以建议先看下个小节：scope生命周期拆解。由于要遵从原文档的大体顺序，所以顺序没做改动。

浏览器接收一个事件的标准的工作流程应该是：

```
1 接收事件 --> 触发回调 --> 回调执行结束返回 --> 浏览器重绘DOM --> 浏览器返回等待下一个事件
```

上面的过程中，如果一切都发生在Angular的执行上下文的话，那相安无事，Angular能够知道数据模型发生的改变；但是如果当浏览器的控制权跑到原生的 JavaScript中去时（译注：比如通过jQuery监听事件之类的非Angular的回调等），那么应用执行的上下文就发生在Angular的上下文之外了，这样就导致Angular无法知晓数据模型的任何改变。想要让Angular*重新掌权并知晓正在发生的数据模型的变化*的话，那就需要通过使用 `$apply` 方法让上下文执行环境重新进入到Angular的上下文中（*注：用法 scope.apply()*）。只有执行上下文重新回到Angular中，那样数据模型的改变才能被Angular所识别并作出相应操作（注：当然，如果执行上下文没有发生改变，也就没有必要显式地去进行 apply 操作）。举个例子，像 `ng-click` 这个指令，监听DOM事件时，表达式的计算就必须放在 `$apply()` 中（注：例子不够完备，待补充）。

在计算完表达式之后，`$apply()` 方法执行Angular的 `$digest` 阶段。 在 `$digest` 阶段，scope 检查所有通过 `$watch()` 监测的表达式（或别的数据）并将其与它们自己之前的值进行比较。这就是所谓的 `脏值检查(dirty checking)`。 另外，需要注意的是，`$watch()` 的监测是异步执行的。这就意味着当给一个作用域中的属性被赋值时，如：`$scope.username="angular"`，`$watch()` 方法不会马上被调用，它会被延迟直到 `digest()` 阶段跑完（*注：至于$digest 阶段到底是干嘛的，你可以认为就是个缓冲阶段，而且是必要的阶段*）。 通过 `$digest()` 给我们提供的这个延迟是很有必要的，也正是应用程序常常想要的（注：出于性能的考虑），因为有这个延迟，我们可以等待几个或多个数据模型的改变/更新攒到一块， 合并起来放到一个 `$watch()` 中去监测，而且这样也能从一定程度上保证在一个 `$wathc()` 在监测期间没有别的 `$watch()` 在执行。 这样，当前的 `$watch()` 可以返回给应用最准确的更新通知，进而刷新试图或是进入一个新的 `$digest()` 阶段。（译注：这一段有点晦涩，可以看下面的一张图结合着学习；还有就是可以把整个过程想象为为了提升效率， 把多个同性质的数据放在同一个 $digest 轮循中处理能够大大提高效率，就像zf办事经常这样，当然，它们的效率不高，ng则不同，效率相对高）

scope生命周期拆解

相信看了上面一段话，没理解的还是很多人，因为标题虽说是讲作用域的生命周期，但是一上来就跟我讲的是关于Angular的执行上下文，怎么也没联系到一块。说实话，翻译这段，真心有点要命的感觉。当然，把它拆分成多个步骤来看，相信会更清晰，因为下面我们是真要讲作用域的生命周期，让我们来过一遍。

1. **创建期**

   `root scope` 是在应用程序启动时由 `$injector` 创建的。另外，在指令的模版链接阶段（template linking），指令会创建一些新的子级 scope。

2. **注册$watch**

   在模版链接阶段（template linking），指令会往作用域中注册 `监听器(watch)`，而且不止一个。这些 `$watch` 用来监测数据模型的更新并将更新值传给DOM。

3. **数据模型变化**

   正如上面一节所提到的，要想让数据模型的变化能够很好的被Angular监测，需要让它们在 `scope.$apply()` 里发生。 当然，对于Angular本身的API来讲，无论是在控制器中做同步操作，还是通过 `$http` 或者 `$timeout` 做的非同步操作， 抑或是在Angular的服务中，是没有必要手动去将数据模型变化的操作放到 `$apply()` 中去的，因为Angular已经隐式的为我们做了这一点。

4. **数据模型变化监测**

   在把数据变化 `$apply` 进来之后，Angular开始进入 `$digest` 轮循（就是调用 `$digest()` 方法），首先是 rootscope 进入 `$digest` ，然后由其把各个监听表达式或是函数的任务传播分配给所有的子级作用域，那样各个作用域就各司其职了，如果监听到自己负责的数据模型有变化，马上就调用 `$watch` 。（*译注：这里所说的从根scope往下分发是译者自己的想法，如有错误，请纠正*）

5. **销毁作用域**

当子级作用域不再需要的时候，这时候创建它们的就会负责把它们回收或是销毁（注：比如在指令中，创建是隐式的，销毁可以不但可以是隐式的，也可以是显式的，如 `scope.$destroy()` ）。销毁是通过 `scope.$destroy()` 这个方法。销毁之后， `$digest()` 方法就不会继续往子级作用域传播了，这样也就可以让垃圾回收系统把这一个作用域上用来存放数据模型的内存给回收利用了。

作用域和指令

在编译（或说解析）阶段, 编译器在HTML解析器解析页面遇到非传统的或是自己不能识别的标签或别的表达式时，Angular编译器就将这些HTML解析器不懂的东西（其实就是 指令 ）在当前的DOM环境下解析出来。通常，指令分为两种，一种就是我们常说的指令，另外一种就是我们通常叫它Angular表达式的双大括号形式，具体如下：

- 监测型 指令 ，像双大括号表达式 `{{expression}}` 。这种类型的指令需要在 `$watch()` 方法中注册一个监听处理器（译注：隐式还是显式的需要看执行上下文），来监听控制器或是别的操作引起的表达式值改变，进而来更新视图。
- 监听型 指令 ，像 `ng-click` ， 这种是在HTML标签属性中直接写好当 `ng-click` 发生时调用什么处理器，当DOM监听到 `ng-click` 被触发时，这个指令就会通过 `$apply()` 方法执行相关的表达式操作或是别的操作进而更新视图。

综上，无论是哪种类型的指令，当外部事件（可能是用户输入，定时器，ajax等）发生时，相关的 表达式 必须要通过 `$apply()` 作用于相应的作用域，这样所有的监听器才能被正确更新，然后进行后续的相关操作。

可以创建作用域的指令

大多数情况下， 指令 和作用域相互作用，但并不创建作用域的新实例。但是，有一些特殊的指令，如 `ng-controller` 和 `ng-repeat` 等，则会创建新的下级作用域，并且把这个新创建的作用域和相应的DOM元素相关联。如前面说过的从DOM元素抓取作用域的方式（如果你还记得的话），就是调用 `angular.element(aDomElement).scope()` 方法。

作用域与控制器

作用域和控制器的交互大概有以下几种情况：

- 控制器通过作用域对模版暴露一些方法供其调用，详情见 `ng-controller`
- 控制器中定义的一些方法（译注：行为或操作逻辑）可以改变注册在作用域下的数据模型（也就是作用域的属性）
- 控制器在某些场合可能需要设置 监听器 来监听作用域中的数据模型(model)。这些监听器在控制器的相关方法被调用时立即执行。

更多内容，请看 `ng-controller` 一节。

作用域 `$watch` 性能

因为在Angular中对作用域进行脏值检查（ `$watch` ）实时跟踪数据模型的变化是一个非常频繁的操作，所以，进行脏值检查的这个函数必须是高效的。一定要注意的是，用 `$watch` 进行脏值检查时，一定不要做任何的DOM操作，因为DOM操作拖慢甚至是拖垮整体性能的能力比在 JavaScript对象上做属性操作高好几个数量级。

与浏览器事件轮循整合

下图与示例描述了Angular如何与浏览器事件轮循进行交互。

1. 浏览器的事件轮循等待事件到来，事件可以是用户交互，定时器事件，或是网络事件（如 ajax 返回）
2. 事件发生，其回调被执行，回调的执行就使得应用程序的执行上下文进入到了 JavaScript 的上下文。然后在JavaScript的上下文中执行，并修改相关的DOM结构
3. 一旦回调执行完毕，浏览器就离开 JavaScript的上下文回到浏览器上下文并基于DOM结构的改变重新渲染视图

讲了那么多些，那么Angular是怎么在这里横插一杠呢？看图，Angular是插进了 JavaScript的上下文中，通过提供Angular自己的事件处理轮循来改变正常的JavaScript工作流。它其实是把JavaScript上下文很成了两块：一个是传统的JavaScript执行上下文（图中浅蓝色区域），一个是Angular的执行上下文（图中淡黄色区域）。 只有在Angular上下文执行的操作才会受益于Angular的数据绑定，异常处理，属性检测，等等。当然，如果不在Angular的上下文中，你也可以使用 `$apply()` 来进入Angular的执行上下文。 需要注意的是， `$apply()` 在Angular本身的很多地方（如控制器，服务等）都已经被隐式地调用了来处理事件轮循。 显示地使用 `$apply()` 只有在你从JavaScript上下文或是从第三方类库的回调中想要进入Angular时才需要。让我们来看看具体的流程：

1. 进入Angular执行上下文的方法，调用 `scope` . `$apply` `(stimulusFn)` 。上面 `$apply()` 中的参数 `stimulusFn` 是你想要让它进入Angular上下文的代码
2. 进入 `$apply()` 之后，Angular执行 `stimulusFn()` ，而这个函数通常会改变应用程序的状态（可能是数据，或是方法调用等）
3. 之后，Angular进入 `$digest` 轮循。这个轮循是由两个较小的轮循构成，一个是处理 `$evalAsync` 队列（异步计算的队列），另一个是处理 `$watch` 列表。 `$digest` 轮循不断迭代变更（在 `$eval` 和 `$watch` 之间变更）直到数据模型稳定，这个状态其实就是 `evalAsync` 队列为空且 `$watch` 列表不再监测到变化为止。（译注：其实这里就是所有外来的异步操作堆起来成为一个队列，由 `$eval` 一个个计算，然后 `$watch` 看一下这个异步操作对应的数据模型是否还有改变，有改变，就继续 `$eval` 这个异步操作，如果没改变，那就拿异步操作队列里的下个异步操作重复上述步骤，直到异步操作队列为空以及 `$watch` 不再监测到任何数据模型变化为止）
4. `$evalAsync` 队列是用来安排那些待进入Angular `$digest` 的异步操作，这些操作往往是在浏览器的视图渲染之前，且常常是通过 `setTimeout(0)` 触发。但是用 `setTimeout(0)` 这个方法就不得不承受缓慢迟钝的响应以及可能引起的闪屏（因为浏览器在每次事件发生后都会渲染一次）（译注：这里个人觉得不要理解的太复杂，按照上面第三点理解就够用了，这边个人翻译的也不是太好，后期配以例子完善）
5. `$watch` 列表则是存放了一组经过 `$eval` 迭代之后可能会改变的Angular的表达式集合。如果数据模型变化被监测到，那么 `$watch` 函数被调用进而用新值更新DOM。
6. 一旦Angular的 `$digest` 轮循完成，那么应用程序的执行就会离开Angular及 JavaScript的上下文。然后浏览器重新渲染DOM来反映发生的变化

接下来是传统的 `Hello world` 示例（就是本节的第一个例子）的流程剖析，这样你应该就能明白整个例子是如何在用户输入时产生双向绑定的。

1. 编译阶段:
    1. `ng-model` 和 `input` `指令` 在 `<input>` 标签中设置了一个 `keydown` 监听器
    2. 在 `{{greeting}}` 插值（也就是表达式）这里设置了一个 `$watch` 来监测 `username` 的变化
2. 执行阶段:
    1. 在 `<input>` 输入框中按下 '`X`' 键引起浏览器发出一个 `keydown` 事件
    2. `input` 指令捕捉到输入值的改变调用 `$apply` `("username = 'X';")` 进入Angular的执行环境来更新应用的数据模型
    3. Angular将 `username = 'X';` 作用在数据模型之上，这样 `scope.username` 就被赋值为 `'X'` 了
    4. `$digest` 轮循开始
    5. `$watch` 列表中监测到 `username` 有一个变化，然后通知 `{{greeting}}` 插值表达式，进而更新DOM

6. 执行离开Angular的上下文，进而 `keydown` 事件结束，然后执行也就退出了 JavaScript的上下文；这样 `$digest` 完成
7. 浏览器用更新了的值重新渲染视图

AngularJS 依赖注入(Dependency Injection)

DI简介

对象或函数可以通过三种方式获得所依赖的对象（简称依赖）：

1. 创建依赖，通常是通过 `new` 操作符
2. 查找依赖，在一个全局的注册表中查阅它
3. 传入依赖，需要此依赖的地方等待被依赖对象注入进来

前两种方式：创建或是查找依赖都不是那么理想，因为它们都将依赖写死在对象或函数里了。问题在于，想要修改这两种方式获得依赖对象的逻辑是很困难的。尤其是在测试的时候，会遇到很多问题，因为测试时常常需要我们提供所依赖对象的替身(MOCK)。

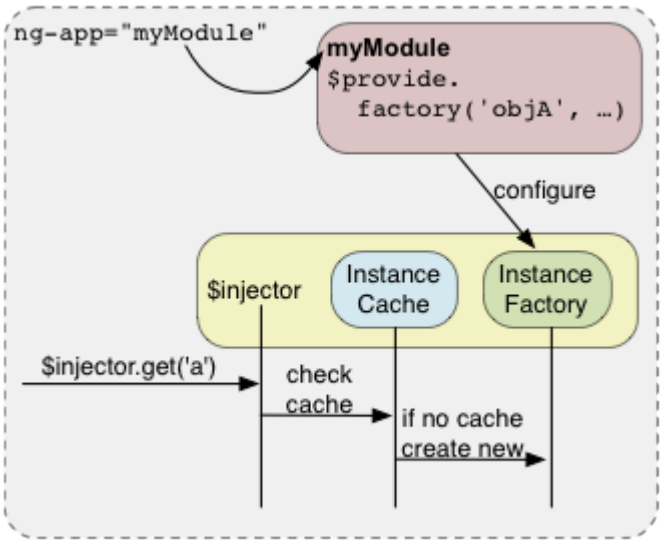
第三种方式是最理想的，因为它免除了客户代码里定位相应的依赖这个负担，反过来，依赖总是能够很简单地被注入到需要它的组件中。

```
function SomeClass(greeter) {
   this.greeter = greeter;
 }

 SomeClass.prototype.doSomething = function(name) {
   this.greeter.greet(name);
 }
```

上述例子中，`SomeClass` 不必在意它所依赖的`greeter`对象是从哪里来的，只要知道一点：在运行的时候，`greeter` 依赖已经被传进来了，直接用就是了。

这个例子中的代码虽然理想，但是它却把获得所依赖对象的大部分责任都放在了我们创建 `SomeClass` 的客户代码中。

为了分离"创建依赖"的职责，每个 Angular 应用都有一个 `injector` 对象。这个 `injector` 是一个服务定位器，负责创建和查找依赖。（译注：当你的app的某处声明需要用到某个依赖时，Angular 会调用这个依赖注入器去查找或是创建你所需要的依赖，然后返回来给你用）



下面是一个利用 `injector` 服务例子：

```
// Provide the wiring information in a module
  angular.module('myModule', []).

    // 下面是教 injector 如何构建一个 'greeter' 依赖
    // 注意 greeter 本身依赖于 '$window'
    factory('greeter', function($window) {
      // 这是一个 factory 函数，负责创建 'greeter' 服务
      return {
        greet: function(text) {
          $window.alert(text);
        }
      };
    });

  // 从 module 创建的 injector
  // 这个常常是 Angular 启动时自动完成的
  var injector = angular.injector(['myModule', 'ng']);

  // 通过 injector 请求任意的依赖
  var greeter = injector.get('greeter');
```

函数或对象通过请求依赖解决了硬编码的问题，但同时也就意味着 injector 需要通过应用传递，而传递 injector 破坏了 Law of Demeter。为了弥补这个，我们通过像下面例子那样声明依赖，将依赖查找的任务丢给了 injector 去做：

```
  <!-- Given this HTML -->
  <div ng-controller="MyController">
    <button ng-click="sayHello()">Hello</button>
  </div>
```

```
// And this controller definition
function MyController($scope, greeter) {
  $scope.sayHello = function() {
    greeter.greet('Hello World');
  };
}

// The 'ng-controller' directive does this behind the scenes
injector.instantiate(MyController);
```

注意，通过让 `ng-controller` 在背后调用 injector 初始化控制器类满足了 `MyController` 需要依赖的需求，而且可以让控制器根本不知道 injector 的存在（译注：好一招瞒天过海）。这是最好的结果了。应用中的代码简单地提交需要它需要某依赖的请求，不需要去管 injector ，而且这样也不违反迪米特法则 。

依赖注释

（译注：此处注释非代码注释，应理解为依赖声明的方法）

那么，injector 是如何知道哪些服务需要被注入呢？

应用开发者需要提供 injector 需要使用的注释信息来解析依赖。Angular 之中，按照API文档说明，某些 API 方法需要通过 injector 调用。这样，injector 要知道得往这个方法中注入什么服务。下面是用服务名信息来进行注释的三种等价的方式，它们可以互用，按照你觉得适合的情况选用相应的方式。

推断依赖

最简单的获取依赖的方法是让你的函数的参数名直接使用依赖名。

```
function MyController($scope, greeter) {    ...  }
```

给 injector 一个函数，它可以通过检查函数声明并抽取参数名可以推断需要注入的服务名。在上面的例子中，`$scope` 和 `greeter` 是两个需要被注入到函数中的服务。

虽然这种方式很直观明了，但是它对于压缩的 JavaScript 代码来说是不起作用的，因为压缩过后的 JavaScript 代码重命名了函数的参数名。这就让这种注释方式只对 pretotyping 和 demo级应用有用。

`$inject` 注释

为了让重命名了参数名的压缩版的 JavaScript 代码能够正确地注入相关的依赖服务。函数需要通过 `$inject` 属性进行标注，这个属性是一个存放需要注入的服务的数组。

```
var MyController = function(renamed$scope, renamedGreeter) {    ...  }
MyController['$inject'] = ['$scope', 'greeter'];
```

在这种场合下，`$inject` 数组中的服务名顺序必须和函数参数名顺序一致。以上述代码段为例，`$scope` 将会被注入到 'renamed$scope'，而 'greeter' 则是注入到 'renamedGreeter'。需要注意 'inject` 注释是和真实的函数声明中的参数保持同步的。

这种注释方法对于控制器声明很有用处，因为它是把注释信息赋给了函数。

行内注释

有时候用 `$inject` 注释的方式不方便，比如标注指令的时候（译注：这里标注指令可以理解为告诉指令需要加载哪些服务依赖的说明）。

看下面的例子：

```
1   someModule.factory('greeter', function($window) {    ...  });
```

由于需要一个临时的变量导致代码膨胀：

```
1   var greeterFactory = function(renamed$window) {    ...  };    greeterFactory.$inject =
['$window'];    someModule.factory('greeter', greeterFactory);
```

所以，第三种注释风格被引入，如下：

```
1   someModule.factory('greeter', ['$window', function(renamed$window) {    ...  }]);
```

记住，所有上面的三种依赖注释风格（译注：声明依赖的风格）是等价的，在 Angular 中任何支持依赖注入的地方都可以使用。

哪里使用DI

在 Angular 中，DI 无处不在。通常在控制器和工厂方法（译注：所谓的工厂方法个人理解为 Angular 中的API）中使用较多。

控制器中使用DI

控制器是负责应用操作逻辑的 JavaScript 类，推荐的在控制器中使用DI的方式是用数组标记风格，如下：

```
1   someModule.controller('MyController', ['$scope', 'dep1', 'dep2', function($scope, dep1,
dep2) {    ...    $scope.aMethod = function() {      ...    }    ...  }]);
```

如上那样，避免了在全局作用域内创建控制器，同时也避免了代码压缩时引起的注入问题。

工厂方法中使用DI

工厂方法负责创建 Angular 中的绝大多数对象。例如指令，服务，和过滤器等。工厂方法是注册在模块之下的，推荐的声明方式如下：

```
1   angular.module('myModule', []).    config(['depProvider', function(depProvider){      ...
}]).    factory('serviceId', ['depService', function(depService) {      ...    }]).
directive('directiveName', ['depService', function(depService) {      ...    }]).
filter('filterName', ['depService', function(depService) {      ...    }]).
run(['depService', function(depService) {      ...    }]);
```

AngularJS 模板（Templates）

Angular的模板是一个声明式的视图，它指定信息从模型、控制器变成用户在浏览器上可以看见的视图。 它把一个静态的DOM —— 只包含HTML，CSS以及Angular添加的标记和属性，然后引导Angular为其加上一些行为和格式转换器，最终变成一个动态的DOM。

在Angular中有以下元素属性可以直接在模板中使用:

- 指令(Directive) — 一个可扩展已有DOM元素或者代表可重复使用的DOM组件，用扩展属性(或者元素)标记。
- 表达式(Expressions) — 用双括号 `{{ }}` 给元素绑定表达式。
- 过滤器(Filter) — 格式化数据显示在界面上。
- 表单控件(Form Control) — 验证用户输入。

注: 除了在模板中声明元素外, 你也可以在JavaScript代码中访问这些元素。

下面的代码片段展示了一个简单的Angular模板，主要由带有(指令)的HTML标准标签和`{{ }}`表达式(Expressions)组成:

```
<html ng-app> <!-- Body标记带有ngController参数  -->
    <body ng-controller="MyController">
        <input ng-model="foo" value="bar">
        <!-- Button标记带有ng-click指令，字符串表达式'buttonText'被包裹在""中 -->
        <button ng-click="changeFoo()"></button>
        <script src="angular.js">
     </body>
</html>
```

在一个简单的单页程序中，模板包含HTML，CSS和Angular指令，通常只是一个HTML文件(如`index.html`)，在一个更复杂的程序中，你可以在一个主要的页面用"零件(partials)"展示多个视图，这些 "零件(partials)"都是独立的HTML文件，在主页面可以包含(include) 这些"零件(partials)"页面，通过路由 $route 和 ngView指令结合， 相关的示例代码参考 phonecat教程 中的第七，八步骤。

AngularJS 使用css(Working With CSS)

Angular提供了这些CSS类，它们为你的应用提供便于使用的风格。

Angular 使用的CSS类

- `ng-scope`
  - **用法:** angular把这个类附加到所有创建了新`作用域(Scope)`的HTML元素上。(参见 作用域)
- `ng-binding`
  - **用法:** angular把这个类附加到所有通过 `ng-bind` 或 绑定了任何数据的元素上。(参见 数据绑定)
- `ng-invalid`, `ng-valid`
  - **用法:** angular把这个类附加到进行了验证操作的所有input组件元素上。 (参见 `input` 指令)
- `ng-pristine`, `ng-dirty`
  - **用法:** angular的`input`指令给所有新的、还没有与用户交互的input元素附加上`ng-pristine`类，当用户有任何输入时，则附加上 `ng-dirty`.

AngularJS 过滤器（Filters）

过滤器用来格式化表达式中的值。它可以用在视图模板(templates)、控制器(controllers)或者服务(services)中。我们很容易就能自定义过滤器。

过滤器的API可以在`filterProvider`中找到。

在模板中使用过滤器

过滤器可以应用在视图模板中的表达式中，按如下的格式：

```
{{ 表达式 | 过滤器名 }}
```

例如，在"{{ 12 | currency }}"标记中格式化了数字12作为一种货币的形式来显示，它使用了`currency`过滤器。格式化之后的结果是"$12.00"。

过滤器可以应用在另外一个过滤器的结果之上。这叫做“链式”使用，按如下格式：

```
{{ 表达式 | 过滤器1 | 过滤器2 |...}}
```

过滤器可以拥有（多个）参数，按如下格式：

```
{{ 表达式 | 过滤器:参数1:参数2:...}}
```

例如，在“{{ 1234 | number:2 }}”的标记中格式化显示了数字1234为小数点后两位，使用了`number`过滤器。显示的结果为"1,234.00"。

在控制器和服务中使用过滤器

你同样可以在控制器和服务中使用过滤器。在这种情况下，在你的控制器或者服务中添加以“<过滤器名>Filter”为名的依赖。例如，使用"numberFilter"为依赖时，会相应的注入number过滤器。

下面的例子展示了一个名为filter的过滤器。 这个过滤器根据不同的参数将一个数组拆分成多个子数组。这个过滤器可以在模板中以{{ctrl.array | filter:'a'}}`的方式来使用，这会以'a'作为查询字符串来进行过滤。但是，在视图模板中使用过滤器会在每次的更新中重新调用过滤器，当数组很大的时候，开销会很大。

紧接着下面的例子在控制器中直接调用了这个过滤器。 使用这种方式，控制器可以在需要的时候手动调用（例如在从后台获取数据或者过滤器中的表达式有改变的时候）。

index.html script.js

```
<!doctype html>
<html ng-app="FilterInControllerModule">
  <head>
    <script src="http://code.angularjs.org/1.2.25/angular.min.js"></script>
    <script src="script.js"></script>
  </head>
  <body>
    <div ng-controller="FilterController as ctrl">
      <div>
        All entries:
        <span ng-repeat="entry in ctrl.array">{{entry.name}} </span>
      </div>
      <div>
        Entries that contain an "a":
        <span ng-repeat="entry in ctrl.filteredArray">{{entry.name}} </span>
      </div>
    </div>
  </body>
</html>
```

index.html script.js

```
angular.module('FilterInControllerModule', []).
  controller('FilterController', ['filterFilter', function(filterFilter) {
    this.array = [
      {name: 'asnowwolf'},
      {name: 'why520crazy'},
      {name: 'joe'},
      {name: 'ckken'},
      {name: 'lightma'},
      {name: 'FrankyYang'}
    ];
    this.filteredArray = filterFilter(this.array, 'a');
  }]);
```

效果

All entries: Tobias Jeff Brian Igor James Brad
Entries that contain an "a": Tobias Brian James Brad

创建自定义过滤器

创建自定义过滤器的过程很简单:仅仅需要在模块中注册一个新的过滤器工厂方法。其中使用了`filterProvider`。这个工厂方法应该返回一个以输入值为第一个参数的新过滤方法，过滤器中的参数都会作为附加参数传递给它。

下面的示例过滤器反转显示一个字符串。另外，它可以根据参数把字母全部大写。

```
<!doctype html>
<html ng-app="MyReverseModule">
  <head>
    <script src="http://code.angularjs.org/1.2.25/angular.min.js"></script>
    <script src="script.js"></script>
  </head>
```

```
  <body>
    <div ng-controller="Ctrl">
      <input ng-model="greeting" type="text"><br>
      未添加过滤器: {{greeting}}<br>
      逆置: {{greeting|reverse}}<br>
      逆置 + 大写: {{greeting|reverse:true}}<br>
    </div>
  </body>
</html>
```

```
angular.module('myReverseFilterApp', [])
.filter('reverse', function() {
  return function(input, uppercase) {
    input = input || '';
    var out = '';
    for (var i = 0; i < input.length; i++) {
      out = input.charAt(i) + out;
    }
    // conditional based on optional argument
    if (uppercase) {
      out = out.toUpperCase();
    }
    return out;
  };
})
.controller('MyController', ['$scope', 'reverseFilter', function($scope, reverseFilter) {
  $scope.greeting = 'hello';
  $scope.filteredGreeting = reverseFilter($scope.greeting);
}]);
```

AngularJS 表单(Forms)

控件 (`input` , `select` , `textarea`) 是用户输入数据的一种方式。一个表单就是多个控件的集合，用来组织相关的控件。

表单和控件提供验证服务，在用户输入信息有误的时候进行提示。 这提升了用户体验，因为我们在第一时间告知用户什么地方出错了、如何修正错误。 记住，尽管客户端（浏览器）验证在用户体验方面起了重要作用，但是，它很容易被绕过，因此是不能信任的。 为了应用的安全，服务端的验证仍然是必须的。

简单的表单

理解双向绑定的关键指令是 `ngModel` . 指令 `ngModel` 通过维护"数据到视图"的同步以及"视图到数据"的同步实现了双向绑定。 另外，它还提供了 `API` 来让其他指令扩展其行为。

使用 CSS 类

为了允许对表单和控件自定义样式， `ngModel` 增加了如下的CSS类： - `ng-valid` - `ng-invalid` - `ng-pristine` - `ng-dirty`

接下来的例子中使用这些CSS类来表明每一个表单控件是否有效。 在例子中，`user.name` 和 `user.email` 都是必需的（required），它们仅在成为脏数据(dirty)时才会显示红色背景，但不会在初始状态下（也为空）显示。 这样就防止了用户在尚未与该控件进行交互，而该控件的当前状态恰好不符合验证条件的时候，出现不恰当的错误提示。

与表单绑定和控制状态

一个表单是一个 `FormController` 的实例。 这个表单实例可以通过 'name' 属性装载到scope中去。

类似的，一个拥有 `api/ng.directive:ngModel` 指令的输入控件，包含一个 `NgModelController` 实例。 这样一个控件的实例可以使用 'name' 属性装载到表单实例中去，作为表单实例的一个字段。name属性指定了其在表单实例中的字段名。

这意味着，在视图中使用基本的数据绑定形式就可以访问到表单和控件中的内部状态。

这让我们可以为上面的例子扩展以下功能：

- 重置按钮 只在表单有改变的时候才可用
- 保存按钮 只在表单有改变且数据有效的时候才可用
- 为 `user.email` 和 `user.agree` 自定义错误提示信息

自定义验证

Angular提供了一些常用的html5输入控件的验证实现：(`text` , `number` , `url` , `email` , `radio` , `checkbox`), 以及一些用于验证的指令 (`required` , `pattern` , `minlength` , `maxlength` , `min` , `max`).

定义你自己的验证器可以首先定义一个指令，这个指令为 'ngModel' `控制器` 添加自定义验证方法。 我们通过指定一个依赖来获得对这个控制器的引用，从下面的例子中可以看出。

我们的验证在两个时机触发：

- **数据到视图的更新** - 任何时候，受约束的模型改变时，所有在 `NgModelController#$formatters` 数组中的方法会被管道式调用（即：一个接一个的调用方法），这样一来，所有的方法都有机会对值来进行格式化并改变表单和控件的有效性状态，这将通过调用 `NgModelController#$setValidity` 来实现。
- **视图到数据的更新** - 类似的，当用户与一个控件交互时，调用 `NgModelController#$setViewValue` . 这又反过来管式调用了所有在 `NgModelController#$parsers` 数组中的方法，这样一来，所有的方法都有机会对值来进行转换并改变表单和控件的有效性状态，通过 `NgModelController#$setValidity` 来实现。

下面的示例中我们创建了两个指令。

- 第一个指令是 'integer' 整形数字，它验证了输入是否是一个合法的整形数字。 例如，'1.23'是一个非法值，因为它包含小数部分。 注意，我们并没有对数组进行压栈操作，而是插入数组的开头。 这是因为我们希望指令能成为第一个解析和处理被控制的值，而不希望在我们执行验证前，值已经在其他环节被转换成了数字。
- 第二个指令是 'smart-float' 智能浮点数。 它能解析 '1.2' 和 '1,2' 并将其转换成合法的浮点数 '1.2'. 注意，我们此时不能在HTML5的浏览器中使用 'number' 类型，因为在这种情况下，浏览器不会允许用户输入像 '1,2' 这种被认为是不合法的数字。

```
<!doctype html>
<html ng-app="form-example1">
  <head>
    <script src="http://code.angularjs.org/1.2.25/angular.min.js"></script>
    <script src="script.js"></script>
  </head>
  <body>
    <div ng-controller="Controller">
      <form name="form" class="css-form" novalidate>
        <div>
          大小 (整数 0 - 10):
          <input type="number" ng-model="size" name="size"
                 min="0" max="10" integer /><br />
          <span ng-show="form.size.$error.integer">这是无效的数字!</span>
          <span ng-show="form.size.$error.min || form.size.$error.max">
            值必需在0到10之间!</span>
        </div>

        <div>
          长度 (浮点):
          <input type="text" ng-model="length" name="length" smart-float />
          <br />
          <span ng-show="form.length.$error.float">
            这是无效的浮点数!</span>
        </div>
      </form>
    </div>
  </body>
</html>
```

```
var app = angular.module('form-example1', []);

var INTEGER_REGEXP = /^\-?\d+$/;
app.directive('integer', function() {
  return {
    require: 'ngModel',
    link: function(scope, elm, attrs, ctrl) {
      ctrl.$parsers.unshift(function(viewValue) {
        if (INTEGER_REGEXP.test(viewValue)) {
          // 验证通过
          ctrl.$setValidity('integer', true);
          return viewValue;
        } else {
          // 验证不通过 返回 undefined (不会有模型更新)
          ctrl.$setValidity('integer', false);
          return undefined;
        }
      });
    }
  };
});

var FLOAT_REGEXP = /^\-?\d+((\.|\,)\d+)?$/;
app.directive('smartFloat', function() {
  return {
    require: 'ngModel',
    link: function(scope, elm, attrs, ctrl) {
      ctrl.$parsers.unshift(function(viewValue) {
        if (FLOAT_REGEXP.test(viewValue)) {
          ctrl.$setValidity('float', true);
          return parseFloat(viewValue.replace(',', '.'));
        } else {
          ctrl.$setValidity('float', false);
          return undefined;
        }
      });
    }
  };
});
```

实现自定义form控件(使用 'ngModel')

Angular实现了所有基本的HTML表单控件((`input` , `select` , `textarea`)，它们在大多数情况下都很有效。 然而，如果你需要更多的灵活性，你可以使用指令来实现你的自定义表单控件。

为了能让自定义控件能够与'ngModel'正常工作，达到双向绑定的效果，它需要：

- 实现 '$render'方法，它负责在数据传递给方法NgModelController#formatters之后渲染数据。
- 调用 '$setViewValue' 方法，在任何用户与控件交互后，模型需要更新的时候调用。这通常在一个DOM事件监听器里完成。

查看 $compileProvider.directive 获得更多的信息。

接下来的例子展示了如何为一个可编辑元素添加双向绑定。

AngularJS 模块(Modules)

大多数应用程序都有个 `main` 函数来初始化、连接以及启动整个应用。ng 中虽然没有 `main` 函数，但它用模块来描述应用将如何启动。这种策略有如下几种优势：

- 整个过程是声明式的，更容易理解

- 在单元测试中，没有必要加载所有模块，这样有利于单元测试的代码书写
- 在场景测试中，额外的模块可以被加载进来，进而重写一些配置，这样有助于实现应用的端到端的测试
- 第三方代码可以很容易被打包成可重用的模块
- 模块可以用任意顺序或并行顺序加载（得益于模块执行的延迟性）

推荐配置

上面的例子太过简单了，对于大型应用显然不适用。对于大型应用，我们建议把它像这样分成多个模块：

- 服务模块
- 指令模块
- 过滤器模块
- 一个应用的模块，依赖于上述的三个模块，而且包含应用的初始化及启动代码

这样划分模块的原因主要是在你的测试中，经常需要忽略难以测试的初始化的代码，而且这样测试时可以单独加载模块进行相关的测试。

上述模块划分仅仅是一种建议性的方案，你可以根据自己的应用去调整。下面的代码显示了上面所述的模块划分：

AngularJS 表达式(Expressions)

"表达式"是一种类似JavaScript的代码片段，通常在视图中以''的形式使用。表达式由`$parse`服务解析，而解析之后经常会使用过滤器来格式化成一种更加用户友好的形式。

例如，下面这些都是Angular中合法的表达式：

- `1+2`
- `user.name`

Angular表达式与JS表达式

可能会有人认为Angular视图表达式就是JavaScript表达式，但这不完全正确，因为Angular并没有使用JavaScript中的'eval()'来解析表达式。你可以认为Angular表达式与JavaScript表达式有如下的区别：

- **属性解析：** 所有的属性的解析都是相对于作用域(scope)的，而不像JavaScript中的表达式解析那样是相对于全局'window'对象的。
- **容错性：** 表达式的解析对'undefined'和'null'具有容错性，这不像在JavaScript中，试图解析未定义的属性时会抛出`ReferenceError`或`TypeError`错误.
- **禁止控制流语句：** 表达式中不允许包括下列语句：条件判断(if)，循环(for/while)，抛出异常(throw)。

另一方面，如果你想执行特定的JavaScript代码，你应该在一个控制器里导出一个方法，然后在模板中调用这个方法。如果你想在JavaScript中解析一个Angular表达式，使用`$eval()`方法。

属性解析

属性解析发生在scope上。不像在JavaScript中，将默认的属性解析放在全局的window对象中，Angular表达式必须使用`$window`来指向全局的'window'对象。例如，如果你想调用在'window'上定义的'alert()'方法，在表达式中，你必须使用'$window.alert()'。作者故意这样设定，就是为了防止对全局状态非正常的访问（一些奇怪bug的常见来源）。

容错性

表达式的解析对undefined和null具有容错性。在JavaScript中，解析'a,b,c'时，如果'a'不是一个对象会抛出异常。在一些语言中这可能会有用，但表达式的解析在Angular中主要用在数据绑定上，我们会像下面这样使用表达式：

```
{{a.b.c}}
```

此时如果'a'是未定义的（也许我们正等着服务器响应数据，在不久的将来其会被定义），解析抛出异常将不再合理。如果表达式不具有容错性的话，我们的代码会变的繁杂且影响阅读，例如：`{{((a||{}).b||{}).c}}`。

类似的，在undefined和null的对象上调用方法'a.b.c()'时会返回undefined。

禁止控制流语句

在表达式中禁止写控制流语句。这背后的原因在于，Angular设计哲学的核心认为，应用逻辑应该在控制器中，而不是在视图中控制。如果你需要条件表达式，循环或者抛出异常出现在你的视图表达式中，请将其委托到JavaScript方法中执行。

AngularJS 供应者（Providers）

你所构建的每个web应用都是由互相协作以达成特定目标的对象构成。为了让应用得以运行，这些对象还需要被实例化并绑定在一起。在基于Angular框架的应用里，这些对象大都是通过`注入服务`自动地实例化并绑定在一起。

注入器创建两类对象，**服务**和**专用对象**。

服务是对象，而这些对象的API是由编写服务的开发人员所决定的。

专用对象遵循Angular框架特定的API。这些对象包括**控制器**，**指令**，**过滤器**或**动画**。

注入器需要知道如何去创建这些对象。你应该通过注册一种“图纸”来告诉Angular如何创建你的对象。这里共有5种图纸。

最冗长同时又最复杂的图纸是Provider图纸，其余4种分别是 —— Value，Factory，Service和Constant，这4种都只是基于Provider之上的语法糖。

现在让我们看看通过不同图纸来创建和使用服务的场景。首先我们从最简单的例子开始 —— 你代码在很多地方都要使用同一个字符串，这个场景下，我们通过Value图纸来完成服务的创建。

注意：关于模块。

为了使注入器知道如何创建这些对象，并让它们能绑定在一起协同工作，我们需要一张关于“图纸”的注册表。每个图纸都有对象的识别码以及如何创建该对象的说明。

每个图纸都属于一个`Angular模块`。一个Angular模块就像是装着一张或多张图纸的袋子。通过手工记录模块依赖关系是很无趣的工作，所以一个模块里也应该包含该模块依赖于哪些其他模块的信息。

当基于Angular的应用从一个指定的应用模块启动时，Angular会创建一个注入器的实例，紧接着该注入器实例就会创建一张包含“图纸”的注册表，这张注册表就是由Angular核心模块、应用模块以及应用模块的依赖里面定义的所有图纸的集合。当注入器需要为你的应用创建一个对象时，注入器就会查询这张注册表。

Value 图纸

假设我们要实现一个非常简单的服务叫做"clientId"，该服务提供一个表示调用某些远程API时会用到的鉴权id的字符串。

```
var myApp = angular.module('myApp', []);
myApp.value('clientId', 'a12345654321x');
```

请注意我们如何创建一个叫做"myApp"的Angular模块，以及如何指出该模块中包含构建用于`clientId`服务的图纸。在这个例子中`clientId`服务只是一个简单的字符串。

下面我们通过Angular的数据绑定来显示clientId：

```
myApp.controller('DemoController', ['clientId', function DemoController(clientId) {
  this.clientId = clientId;
}]);
```

```
<html ng-app="myApp">
  <body ng-controller="DemoController as demo">
    Client ID: {{demo.clientId}}
  </body>
</html>
```

在上面的例子中，当`DemoController`需要id为"clientId"的服务时，我们通过Value图纸定义了需要注入过去的值。

好了，接下来，我们来学习更复杂的例子！

Factory 图纸

虽然Value图纸很容易编码，但缺少很多我们在创建服务时需要的重要特性。让我们看看比Value图纸更强大的兄弟—— Factory图纸。

Factory图纸增加了以下能力：

- 使用其他服务的能力(即可以有依赖)
- 服务初始化
- 延迟/惰性初始化

Factory图纸通过一个拥有0～n个参数(参数表示该服务对其他服务的依赖)的函数来创建服务，而函数返回值就是Factory图纸创建的服务实例。

注意：Angular框架里所有的服务都是单例对象。这意味着注入器只会使用一次图纸来创建服务实例，然后注入器就会缓存这些服务实例的引用，以备将来使用。

既然我们说Factory是功能相比Value图纸更强大图纸类型，那么你当然可以通过Factory图纸来创建相同的服务。针对我们前面举的`clientId`的Value图纸例子，我们可以用Factory图纸做如下重新实现：

```
myApp.factory('clientId', function clientIdFactory() {
  return 'a12345654321x';
});
```

但是既然token只是一个字符串常量，我们还是坚持使用Value图纸吧，这样使得代码更简洁明了。

但是，假如我们想创建另一个服务，在调用远程API时，该服务可以用来计算鉴权token。这个token叫做'apiToken'，并且是基于`clientId`的值和一个存储在浏览器本地存储的密码计算出来的：

```
myApp.factory('apiToken', ['clientId', function apiTokenFactory(clientId) {
  var encrypt = function(data1, data2) {
    // NSA-proof加密算法：
    return (data1 + ':' + data2).toUpperCase();
  };

  var secret = window.localStorage.getItem('myApp.secret');
  var apiToken = encrypt(clientId, secret);

  return apiToken;
}]);
```

在上面的代码里，我们能看到依赖于`clientId`服务的`apiToken`服务是如何通过Factory图纸定义的。这个工厂服务使用NSA-proof加密来生成鉴权token。

注意：将工厂方法命名为"Factory"是最佳实践（比如，apiTokenFactory）。虽然这种命名方式不是强制性的，但是它有助于浏览代码仓库或者在调试器里跟踪调用堆栈。

就像Value图纸一样，Factory图纸能创建任何类型的服务，不管是原生类型，对象常量，函数，甚至自定义类型的实例。

Service 图纸

Javascript开发人员经常使用自定义类型来编写面向对象的代码。现在，让我们一起探讨如何通过自定义类型实例——`unicornLauncher`服务，将一头独角兽发射到太空中去：

```
function UnicornLauncher(apiToken) {

  this.launchedCount = 0;
  this.launch() {
    // 带上apiToken来发起远程调用
    ...
    this.launchedCount++;
  }
}
```

现在我们准备发射独角兽，但我们注意到`UnicornLauncher`依赖于我们的`apiToken`.我们可以通过使用Factory图纸来满足对`apiToken`的依赖：

```
myApp.factory('unicornLauncher', ["apiToken", function(apiToken) {
  return new UnicornLauncher(apiToken);
}]);
```

然而，这个用例使用Service图纸最合适。

Service图纸实例化服务时，和Value和Factory图纸类似，只是它通过*使用 new 操作符调用构造函数*来实现。构造函数可以接受0～n个参数，这些参数代表着该服务实例的依赖。

注意：Service图纸遵循 "构造函数注入" 的设计模式。

既然我们已经拥有了`UnicornLauncher`类型的构造函数，我们可以像下面代码那样使用Service图纸来替代以上的Factory图纸：

```
1 myApp.service('unicornLauncher', ["apiToken", UnicornLauncher]);
```

是不是更简单了！

注意：是的，我们将一种Service图纸命名为'Service'型，我们对此感到后悔，并且知道我们将来会以某种形式受到惩罚。就像为我们的儿女之一取名叫“孩子”一样，“孩子”，这会惹恼老师的。（译者代言：愿语文老师宽恕我）。