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 (../tutorial/3.html) 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 (../tutorial/3.html) 的变化并传播事件

作用域有什么

- 作用域提供了 (`$watch` (api/ng.$rootScope.Scope#methods_$watch)) 方法监听数据模型的变化
- 作用域提供了 (`$apply` (api/ng.$rootScope.Scope#methods_$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 (../tutorial/3.html) 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) (guide/compiler) 阶段时， `指令(directive)` (api/ng.$compileProvider#methods_directive)会设置一个 `$watch` (api/ng.$rootScope.Scope#methods_$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）

让我们来看下面一个例子，可以说明作用域作为视图与控制器的黏合剂：

源码

index.html script.js


```
<!doctype html>
<html ng-app>
  <head>
    <script src="http://code.angularjs.org/1.2.25/angular.min.js"></script>
    <script src="script.js"></script>
  </head>
  <body>
    <div ng-controller="MyController">
      Your name:
        <input type="text" ng-model="username">
        <button ng-click='sayHello()'>greet</button>
      <hr>

    </div>
  </body>
</html>
```

效果

Your name: greet

在上面这个例子中，我们有：

- 控制器： `MyController` ，它引用了 `$scope` 并在其上注册了两个属性和一个方法
- $scope 对象：持有上面例子所需的数据模型，包括 `username` 属性、 `greeting` 属性（注：这是在 `sayHello()` 方法被调用时注册的）和 `sayHello()` 方法
- 视图：拥有一个输入框、一个按钮以及一个利用双向绑定来显示数据的内容块

那么具体整个示例有这样两个流程，**从控制器发起的角度**来看就是：

1. 控制器往作用域中写属性：

- 给作用域中的 `username` 赋值，然后作用域通知视图中的 `input` 数据变化了， `input` 因为通过 `ng-model` 实现了双向绑定可以知道 `username` 的变化，进而在视图中渲染出改变的值，这里是 `World`

2. 控制器往作用域中写方法
   - 给作用域中的 `sayHello()` 方法赋值，该方法被视图中的 `button` 调用，因为 `button` 通过 `ng-click` 绑定了该方法，当用户点击按钮时， `sayHello()` 被调用，这个方法读取作用域中的 `username` 属性，加上前缀字符串 `Hello` ，然后赋值给在作用域中新创建的 `greeting` 属性

整个示例的过程如果**从视图的角度看**，那主要是以下三个部分：

1. `input` 中的渲染逻辑：展示了通过 `ng-model` 进行的作用域和 视图中某表单元素的双向绑定
   - 根据 `ng-model` 中的 `username` 去作用域中取，如果已经有值，那么用这个默认值填充当前的输入框
   - 接受用户输入，并且将用户输入的字符串传给 `username` ，这时候作用域中的该属性值实时更新为用户输入的值
2. `button` 中的逻辑
   - 接受用户单击，调用作用域中的 `sayHello()` 方法
3. `{{greeting}}` 的渲染逻辑
   - 在用户未单击按钮时，不显示内容
   - *取值阶段*：在用户单击后，这个表达式会去scope中取 `greeting` 属性，而这个作用域和控制器是同一个的（这个例子中），这时候，该作用域下 `greeting` 属性已经有了，这时候这个属性就被取回来了
   - *计算阶段*：在当前作用域下去计算 `greeting` 表达式 (../tutorial/3.html) ，然后渲染视图，显示 `Hello World`

经过以上的两种角度分析示例过程，我们可以知道：**作用域(scope)**对象以及其属性是视图渲染的唯一数据来源。

从测试的角度来看，视图与控制器分离的需求在于它允许测试人员可以单独对应用的操作逻辑进行测试，而不必考虑页面的渲染细节。


```
1.     it('should say hello', function() {
2.       var scopeMock = {};
3.       var cntl = new MyController(scopeMock);
4.
5.       // 确保username被预先填充为World
6.       expect(scopeMock.username).toEqual('World');
7.
8.       // 确保我们输入了新的username后得到了正确的greeting值
9.       scopeMock.username = 'angular';
10.      scopeMock.sayHello();
11.      expect(scopeMock.greeting).toEqual('Hello angular!');
12.    });
```

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` (api/ng.$rootScope)，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 (guide/directive) 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

上面一个简单的例子展示了在作用域分层结构中找属性，是基于原型继承的模式。接下来这个demo用一个图具体展示了作用域的层级结构，让你可以有更直观的了解。

源码

index.html style.css script.js

```
<!doctype html>
<html ng-app>
  <head>
    <script src="http://code.angularjs.org/1.2.25/angular.min.js"></script>
    <script src="script.js"></script>
  </head>
  <body>
    <div class="show-scope-demo">
      <div ng-controller="GreetCtrl">
        Hello !
      </div>
      <div ng-controller="ListCtrl">
        <ol>
          <li ng-repeat="name in names"> from </li>
        </ol>
      </div>
    </div>
  </body>
</html>
```

效果

看到上面的框中，注意，Angular会自动为每个拥有作用域的DOM节点加上 `ng-scope` 类。上图中，拥有红色边框样式的节点，就意味着该节点拥有了自己的作用域，无论它是通过什么方式创建的（*译注：上面可以看到有通过控制器创建的新的作用域，也有通过指令如 `ng-repeat` 创建的*）。上例中， `ng-repeat` 创建的子级作用域是极其必要的，因为每个 `<li>` 中想要渲染输出的 `{{name}}` 显然是不同的值，那就需要为它们提供不同的作用域。同样的，Angular在渲染 `{{department}}` 表达式时，先在当前和 `<li>` 相对应的作用域去找有没有这个属性，如果没有，接着往上找，在这个例子中，直到找到 `$rootScope` 下时，才找到 `department` 属性，然后将其取回，计算，渲染输出。

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` (api/ng.directive:ngApp) 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` (api/ng.$rootScope.Scope#methods_$broadcast)： 从父级作用域广播至子级 scope
- `emitted` (api/ng.$rootScope.Scope#methods_$emit)： 从子级作用域往上发射到父级作用域

让我们来看个例子：

源码

index.html script.js

```
<!doctype html>
<html ng-app>
  <head>
    <script src="http://code.angularjs.org/1.2.25/angular.min.js"></script>
    <script src="script.js"></script>
  </head>
  <body>
    <div ng-controller="EventController">
      Root作用域<tt>MyEvent</tt> count:
      <ul>
        <li ng-repeat="i in [1]" ng-controller="EventController">
          <button ng-click="$emit('MyEvent')">$emit('MyEvent')</button>
          <button ng-click="$broadcast('MyEvent')">$broadcast('MyEvent')</button>
          <br>
          Middle作用域<tt>MyEvent</tt> count:
          <ul>
            <li ng-repeat="item in [1, 2]" ng-controller="EventController">
              Leaf作用域<tt>MyEvent</tt> count:
            </li>
          </ul>
        </li>
      </ul>
    </div>
  </body>
</html>
```

效果

译注：上面例子很简单，有几个需要注意的是：

- `$emit` 和 `$broadcast` 是直接被写在 html 模版中的，而不是写在控制器的 JavaScript代码中，因为这两个方法是直接在 $scope 中就有的，
- 同一个控制器 `EventController` 被用在了三个不同的DOM节点中（*这是为了省事，通常不这样写的*）
- 上面的事件无非就是点击两个按钮，分别出发广播/冒泡（发射）事件，然后在各节点设置监听，这里只要用 `$scope.$on()` 方法（注：如果在指令中，可能就是 `scope.$on()`），就可以进行监听了

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：

```
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 `$apply` (api/ng.$rootScope.Scope#methods_$apply) 方法让上下文执行环境重新进入到Angular的上下文中（*注：用法 `$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 $apply 操作）。举个例子，像 `ng-click` (api/ng.directive:ngClick) 这个指令，监听DOM事件时，表达式的计算就必须放在 `$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` (api/ng.$rootScope.Scope#methods_$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` (api/ng.$rootScope) 是在应用程序启动时由 `$injector` (api/AUTO.$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)` (api/ng.$rootScope.Scope#methods_$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.$apply()` (api/ng.$rootScope.Scope#methods_$apply) 里发生。 当然，对于Angular本身的API来讲，无论是在控制器中做同步操作，还是通过 `$http` (api/ng.$http) 或者 `$timeout` (api/ng.$timeout) 做的非同步操作， 抑或是在Angular的服务中，是没有必要手动去将数据模型变化的操作放到 `$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` (api/ng.$rootScope.Scope#methods_$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()` (api/ng.$rootScope.Scope#methods_$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 (guide/compiler)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` (api/ng.$compileProvider#methods_directive)）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` (api/ng.$compileProvider#methods_directive) ，像双大括号表达式 `{{expression}}` 。这种类型的指令需要在 `$watch()` (api/ng.$rootScope.Scope#methods_$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` (api/ng.$compileProvider#methods_directive) ，像 `ng-click` (api/ng.directive:ngClick) ， 这种是在HTML标签属性中直接写好当 `ng-click` 发生时调用什么处理器，当DOM监听到 `ng-click` 被触发时，这个指令就会通过 `$apply()` (api/ng.$rootScope.Scope#methods_$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 (../tutorial/3.html) 必须要通过 `$apply()` (api/ng.$rootScope.Scope#methods_$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 (api/ng.$compileProvider#methods_directive)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` (api/ng.directive:ngController) 和 `ng-repeat` (api/ng.directive:ngRepeat) 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` (api/ng.directive:ngController)
- 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 (api/ng.$rootScope.Scope#methods_$watch) 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` (api/ng.directive:ngController) 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 (guide/scope) . `$apply` (api/ng.$rootScope.Scope#methods_$apply) `(stimulusFn)` 。上面 `$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` (api/ng.$rootScope.Scope#methods_$digest) 轮循。这个轮循是由两个较小的轮循构成，一个是处理 `$evalAsync` (api/ng.$rootScope.Scope#methods_$evalasync) 队列（异步计算的队列），另一个是处理 `$watch` (api/ng.$rootScope.Scope#methods_$watch) 列表。 `$digest` (api/ng.$rootScope.Scope#methods_$digest) 轮循不断迭代变更（在 `$eval` 和 `$watch` 之间变更）直到数据模型稳定，这个状态其实就是 `evalAsync` (api/ng.$rootScope.Scope#methods_$evalasync) 队列为空且 `$watch` (api/ng.$rootScope.Scope#methods_$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. `$evalAsync` (api/ng.$rootScope.Scope#methods_$evalasync) 队列是用来安排那些待进入Angular `$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. `$watch` (api/ng.$rootScope.Scope#methods_$watch) 列表则是存放了一组经过 `$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` (api/ng.$rootScope.Scope#methods_$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` (api/ng.directive:ngModel) 和 `input` (api/ng.directive:input) 指令 (guide/directive) 在 `<input>` 标签中设置了一个 `keydown` 监听器
    2. 在 `{{greeting}}` (api/ng.$interpolate) 插值（也就是表达式）这里设置了一个 `$watch` (api/ng.$rootScope.Scope#methods_$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` (api/ng.directive:input) 指令捕捉到输入值的改变调用 `$apply` (api/ng.$rootScope.Scope#methods_$apply) `("username = 'X';")` 进入Angular的执行环境来更新应用的数据模型
    3. Angular将 `username = 'X';` 作用在数据模型之上，这样 `scope.username` 就被赋值为 `'X'` 了
    4. `$digest` (api/ng.$rootScope.Scope#methods_$digest) 轮循开始
    5. `$watch` (api/ng.$rootScope.Scope#methods_$watch) 列表中监测到 `username` 有一个变化，然后通知 `{{greeting}}` (api/ng.$interpolate) 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