# AngularJS Html编译（HTML Compiler）



**注意**：这篇文章是面向已经有一定 Angular 基础的开发者。 如果你仅仅只是刚上路，那么我们建议你看看 tutorial (tutorial/index) 先。如果你只是想创建几个自定义的指令，那可以去瞅瞅 directives guide (guide/directive)。而如果你想要更深入地了解 Angular 的编译过程，那么你来对了，这就是你该看的。

Angular 的 `HTML compiler` (api/ng.$compile) 让开发者可以教浏览器一些新的语法技能。编译器允许你往现有的HTML元素或属性添加更多的操作逻辑，甚至可以让你自己创建新的带有自定义行为操作的HTML元素或属性。Angular 把这些操作扩展称之为 `指令` (api/ng.$compileProvider#methods_directive)。

HTML 有一大堆概念用来将其格式化为声明性的静态文档。例如，某个元素需要居中放置，我们并没有必要提供指令让浏览器去将窗口大小分成两半来居中，要做的仅仅是给需要居中的元素加上 `align="center"` 属性，这样就可以达到想要的效果了。这就是声明式语言的力量。

但是话说回来，声明式语言也有其自己的限制，首当其冲的是它不能让开发者扩展浏览器适应新的语法。例如，相比居中文本，要让浏览器在窗口 1/3 的位置排列文本就没有那么简单了。所以这就引出了我们需要一种新的路子来教浏览器一些新技能：识别新的HTML语法。

Angular 预先绑定了一些常见的对构建应用极其有用的指令。同时你也可以创建一些与你应用直接相关的指令。这些扩展构成了与特定领域相关的语言来构建你的应用（译注：指令成了扩展性的特定语言）。

所有的编译在web浏览器中进行，没有任何服务器端的预编译的介入。



## 编译器

编译器是 Angular 提供的一项服务，用来遍历DOM节点，查找特定的属性。编译过程分为两个阶段：

1. **编译**：遍历DOM节点，收集所有的指令，返回一个连接函数（link func）
2. **连接**：将上一步收集到的每个指令与其所在的作用域（scope）连接生成一个实时视图。任何作用域中的模型改变都会实时在视图中反映出来，同时任何用户与视图的交互则会映射到作用域的模型中。这样，作用域中的数据模型就成了唯一的数据源。

一些如 `ng-repeat` (api/ng.directive:ngRepeat) 这样的指令，会为集合中的每个项目克隆一次DOM元素。由于克隆的模板只需要被编译一次，然后为每个克隆实例做一次连接，这样将编译分成编译和连接两个阶段就有效地提升了性能（译注：不用为每个克隆的实例都编译一次，只需对模板进行统一的一次编译，然后在连接阶段单独为每个实例进行到 scope 的连接即可）。

## 指令

在编译过程中，遇到特定的HTML结构（也就是指令）时，指令所声明的行为操作会被触发。指令可以被放在元素名，属性，类名，甚至是注释中。下面是一些等价的调用 `ng-bind` (api/ng.directive:ngBind) 指令的例子：

```
<span ng-bind="exp"></span>
<span class="ng-bind: exp;"></span>
<ng-bind></ng-bind>
<!-- directive: ng-bind exp -->
```

指令其实就是在编译器遍历DOM时碰到就需要执行的函数。另见 `指令API` (api/ng.$compileProvider#methods_directive) 查看更深入的如何写指令的文档。

下面是一个让元素可以被拖拽的指令。注意在 `<span>` 元素中的 `draggable` 属性：

## 源码

index.html script.js

```
<!doctype html>
<html ng-app="drag">
  <head>
    <script src="http://code.angularjs.org/1.2.25/angular.min.js"></script>
    <script src="script.js"></script>
  </head>
  <body>
    <span draggable>Drag ME</span>
  </body>
</html>
```

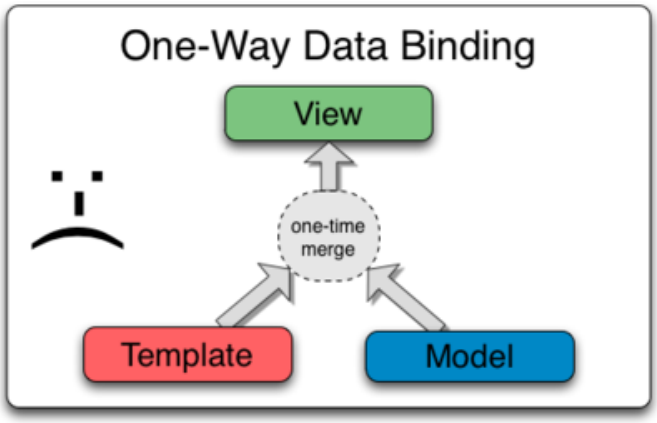
## 效果

Drag ME

`draggable` 属性出现在任何的元素中都会给其带来新的行为操作。我们以一种熟悉HTML语法规则的方式扩展了浏览器的HTML词汇及语法。

## 理解视图

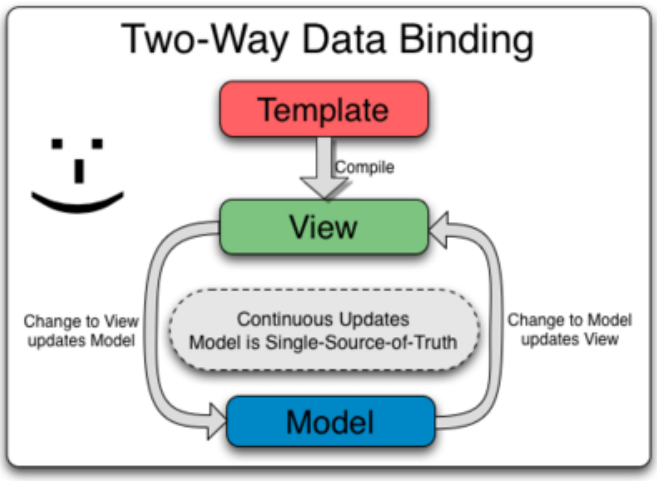
绝大多数模板引擎系统采用的是把字符串模板和数据拼接，然后输出一个新的字符串，在前端这个新的字符串作为元素的 `innerHTML` 属性的值。




这就意味着数据中的任何改变需要重新和模板合并，然后再赋给DOM元素的 `innerHTML` 属性。这里我们可以看到这种策略的一些问题：

1. 读取用户输入及将其与数据合并
2. 重写用户输入
3. 管理整个更新流程
4. 缺少行为表现

Angular 则不同。它的编译器直接使用DOM作为模板而不是用字符串模板。编译阶段的返回结果是一个连接函数（link func），在连接阶段会和特定的作用域中的数据模型连接生成一个实时的视图。视图和作用域数据模型的绑定是透明的。开发者不需要做任何特别的调用去更新视图。同时，我们不使用 `innerHTML` 属性，这样也就不会影响用户输入了。而且，Angular 指令不仅可以包含文本绑定，同时也支持行为操作的绑定（译注：此处可能翻译不甚到位）。




Angular 的这种策略生成的是稳定的DOM模板。DOM元素实例和数据模型实例的绑定在绑定期间是不会发生变化的（也就是说不是每次数据改变，最后产生的模板都要变化一次）。这就意味着在你的代码中可以去获取这些DOM模板元素并且注册相应的事件处理函数，而不用担心这个对DOM元素的引用会因为数据合并而产生变化。

---

## 编译指令

知道 Angular 的编译是在DOM节点上发生而非字符串上是很重要的。通常，你不会注意到这个约束，因为当一个页面加载时，浏览器自动将HTML解析为DOM树了。

然而，如果你自己手动调用 `$compile` 时，则需要注意上面说的注意点了。因为如果你传给它一个字符串，显然是要报错的。所以，在你传值给 `$compile` 之前，用 `angular.element` 将字符串转化为DOM。

HTML 编译可以细分为三个阶段：

1. `$compile` (api/ng.$compile) 遍历DOM节点，匹配指令。

   如果编译器发现某个元素匹配一个指令，那么这个指令就被添加到指令列表中（该列表与DOM元素对应）。一个元素可能匹配到多个指令（译注：也就是一个元素里面可能有多个指令）。

2. 当所有指令都匹配到相应的元素时，编译器按照指令的 `priority` 属性来排列指令的编译顺序。然后依次执行每个指令的 `compile` 函数。每个 `compile` 函数有一次更改该指令所对应的DOM模板的机会。然后，每个 `compile` 函数返回一个 `link` 函数。这些函数构成一个“合并的”连接函数，它会调用每个指令返回的 `link` 函数。
3. 之后， `$compile` 调用第二步返回的连接函数，将模板和对应的作用域连接。而这又会依次调用连接函数中包含的每个指令对应的 `link` 函数，进而在各个DOM元素上注册监听器以及在相应的 `scope` (api/ng.$rootScope.Scope) 中设置对应的 `$watch` s (api/ng.$rootScope.Scope#methods_$watch)。

经过这三个阶段之后，结果是我们得到了一个作用域和DOM绑定的实时视图。所以在这之后，任一发生在已经经过编译的作用域上的数据模型的变化都会反映在DOM之中。

下面是使用 `$compile` 服务的相关代码。它应该能帮你理解 Angular 内部在做些什么（译注：下面代码中的注释就不翻译了，因为有一些如 compile 和 link 翻译效果反而不好）：



```
var $compile = ...; // injected into your code
var scope = ...;
var parent = ...; // DOM element where the compiled template can be appended

var html = '<div ng-bind="exp"></div>';

// Step 1: parse HTML into DOM element
var template = angular.element(html);

// Step 2: compile the template
var linkFn = $compile(template);

// Step 3: link the compiled template with the scope.
var element = linkFn(scope);

// Step 4: Append to DOM (optional)
parent.appendChild(element);
```

## compile 和 link 的区别

这会儿，我想你该很疑惑为什么编译过程被分成了编译和连接两个阶段（译注：这里其实用英文会更好的，compile 和 link，就可以免去歧义了）。（译注：按现在知乎流行的说法，每个问题都有个短答案和长答案两种版本）简短地回答呢，那就是任何时候任一数据模型的改变引起的DOM**结构**的改变都需要这种两阶段编译的支持。

指令有 **compile function** 是不多见的，因为大部分指令通常只关心如何操作特定的DOM元素实例，而不是去改变它的整体结构。

指令通常有 **link function**。连接函数让指令能够往特定的DOM元素实例的克隆对象上注册监听器，同时可以将作用域中的内容复制到DOM中去。

> **最佳实践**：任何能够在指令实例中共享的操作，为了性能考虑，最好是都移到指令的 compile function 中去。

"Compile" vs "Link" 的一个例子

为了理解编译和连接这两个阶段，让我们以 ngRepeat 为例来看一下：

```
1.  Hello , you have these actions:
2.  <ul>
3.    <li ng-repeat="action in user.actions">
4.
5.    </li>
6.  </ul>
```

当我们编译上述例子时，编译器遍历每个节点查找指令。

{{user}} 匹配了 插值指令 (api/ng.$interpolate)，而 ng-repeat 匹配了 ngRepeat 指令 (api/ng.directive:ngRepeat)。

但是编译 ngRepeat (api/ng.directive:ngRepeat) 陷入了困境。

它需要为每个在 user.actions 中的 action 条目克隆一个 <li> 元素。这个刚看起来很平常，但是当你考虑到如果 user.actions 可能会有动态的新的条目添加的话，情况就变得复杂起来。因为这意味着我们需要保存一份干净的 <li> 元素以备后期克隆用。

（译注：在数据模型中）当新的 action 被插入时， <li> 元素作为模板需要被克隆然后插入到 ul 中。但是仅仅克隆 <li> 元素是不够的。同时还要编译这个 <li> 元素，这样才能使得其内部的别的指令，如 {{action.description}} ，能够在正确的 作用域 (api/ng.$rootScope.Scope) 中被计算渲染出结果。



一种比较天真的解决这个问题的策略无非是每次有一个新的 action 我就简单地插入一个 <li> 的拷贝然后将其编译。这种方法的问题在于对于每个我们克隆的 <li> 元素我们都要去编译，这无疑让我们的工作加倍了。具体来说，在每次克隆 <li> 之前，我们都得遍历一下它看看是否有别的指令（译注：此处可能翻译不是很到位，不理解的建议看原文）。这样无疑就让编译过程变的更慢，进而导致应用在插入新节点时变得反应迟钝。

解决方法就是我们上面一直强调的把编译过程分成两个阶段：

**编译阶段**收集所有的指令并按照优先级排序，之后在**连接阶段**将特定的 作用域 (api/ng.$rootScope.Scope)实例与特定的 <li> 实例连接。

> **注意**： *连接* 意味着在DOM上设置监听器以及在相关的作用域中设置 $watch 以保证二者（译注：DOM和作用域）的同步。

ngRepeat (api/ng.directive:ngRepeat) 通过阻止编译阶段进入到 <li> 元素来解决上面的问题，这样就可以克隆原始的 <li> 来处理插入和一处DOM节点的问题了。

（译注：待大神翻译）Instead the ngRepeat (api/ng.directive:ngRepeat) directive compiles <li> separately. The result of the <li> element compilation is a linking function which contains all of the directives contained in the <li> element, ready to be attached to a specific clone of the <li> element.

At runtime the `ngRepeat` (api/ng.directive:ngRepeat) watches the expression and as items are added to the array it clones the `<li>` element, creates a new `scope` (api/ng.$rootScope.Scope) for the cloned `<li>` element and calls the link function on the cloned `<li>`.

---

## 作用域与Transcluded指令是如何工作的

指令最常见的一个应用是创建可重用构件。

下面的为代码展示了一个简单版本的对话框指令的使用。

```
<div>
  <button ng-click="show=true">show</button>

  <dialog title="Hello ."
          visible="show"
          on-cancel="show = false"
          on-ok="show = false; doSomething()">
     Body goes here:  is .
  </dialog>
</div>
```

点击 "show" 按钮会打开对话框。对话框会有一个和 `username` 绑定的标题，同时会有一个主体，这个主体我们是通过在对话框指令定义中的模板通过 transclude 插入的。

下面是 `dialog` 指令中的模板属性：

```
<div ng-show="visible">
  <h3></h3>
  <div class="body" ng-transclude></div>
  <div class="footer">
    <button ng-click="onOk()">Save changes</button>
    <button ng-click="onCancel()">Close</button>
  </div>
</div>
```

上面这个指令的模板还不能适当地渲染，除非我们施上一些魔法。

第一个问题是解决对话框模板中需要的 `title` 数据。而我们希望模板中的 `title` 和指令 `<dialog>` 被使用时的 `title` 属性一致（也就是 `"Hello {{username}}"`）。而且，模板中的按钮会去调用作用域中的 `onOk` 和 `onCancel` 两个函数，而这两个函数的来源也在指令的属性中有定义，要解决的就是映射的问题了。为了解决这个映射问题，我们使用本地 scope 创建本地变量（模板中需要的数据和函数）和外部变量（指令中已有的属性）映射：

```
  scope: {
    title: '@',             // the title uses the data-binding from the parent scope
    onOk: '&',              // create a delegate onOk function
    onCancel: '&',          // create a delegate onCancel function
    visible: '='            // set up visible to accept data-binding
  }
```

在指令的作用域创建本地变量会产生两个问题：

1. 独立的作用域 - 如果用户（译注：使用对话框指令的开发者）在使用 `dialog` 指令时忘记设置 `title` 属性，那么对话框指令的模板中的 `title` 解析时则会去绑定父级作用域中的同名属性。这是完全不可预测的，也不是我们希望看到的。
2. transclusion - 通过引用包含的DOM节点可以看到指令的本地变量，而这本地变量有可能会重写掉一些 transclusion （引用包含）中数据绑定的同名属性。在上述例子中，比如像指令所在的 scope 中的 `title` 属性就重写了在 transclusion （引用包含）的作用域的 `title` 属性（译注：这里 `Body goes here: {{username}} is {{title}}.` 是通过 transclusion 插入到 `dialog` 指令模版中拥有 `ng-transclude` 属性的div中，这样它里面的 title 插值就会被 `dialog` 本地的 title 值改写）

为了解决缺少隔离的问题，指令会声明一个 `isolated` 作用域。一个隔离的作用域不会通过基于原型的方式继承它的父级作用域，所以我嗯就不用担心会有属性被意外改写的情况了。

但是，独立的作用域引来了另外一个问题：如果一个 transcluded（引用包含的） DOM节点是一个指令独立作用域的孩子节点的话，那么它不会绑定到任何数据（译注：像我们上面例子中的情况，就属于绑定不到数据）。出于此，在指令为本地变量创建出独立的作用域之前，我们需要声明 transcluded（引用包含的）作用域是原始作用域（也是独立作用域的父级）的孩子。这样，引用包含创建的作用域就和独立作用域拥有同样的父级，也就是说它们是兄弟作用域。

这会让一切看起来意想不到的复杂，但至少它让指令（控件）的用户和开发者不会那么难以接受。

因此，我们上面例子中指令作用域的声明，最后开起来是这样子的：

```
1.   transclude: true,
2.   scope: {
3.       title: '@',             // the title uses the data-binding from the parent scope
4.       onOk: '&',              // create a delegate onOk function
5.       onCancel: '&',          // create a delegate onCancel function
6.       visible: '='            // set up visible to accept data-binding
7.   },
8.   restrict: 'E',
9.   replace: true
```

译注：最后关于引用包含的作用域这边是一个例子 (http://jsfiddle.net/grahamle/4ag3e/)，更好帮你理解 transclue