问题引入

使用 Angular 进行过一段时间的开发后，基本上都会遇到一个这样的坑：

```
<div ng-controller="TestCtrl">
  <p>{{name}}</p>
  <div ng-if="show">
    <input type="text" ng-model="name">
  </div>
</div>
```

```
 function TestCtrl($scope){
     $scope.show = true;
     $scope.name = 'htf';
 }
```

把 `p` 元素和 `input` 元素绑定同一个变量，你本以为，在输入框输入内容，`p` 中显示的肯定也是随之变化的。然而并不是这样，不管 `input` 中的元素怎么变， `p` 元素中的都没变化，WTF。要说这是什么原因，那就要从 Angular 的作用域说起了。

作用域

每个 Angular 应用默认有一个根作用域 $rootScope， 根作用域位于最顶层，从它往下挂着各级作用域。

通常情况下，页面中 `ng-model` 绑定的变量都是在对应的 Controller 中定义的。如果一个变量未在当前作用域中定义，JavaScript 会通过当前 Controller 的 prototype 向上查找，也就是作用域的继承。

这又分两种情况。

基本类型变量

```
<div ng-controller="OuterCtrl">
  <p>{{x}}</p>
  <div ng-controller="InnerCtrl">
    <input type="text" ng-model="x">
  </div>
</div>
```

```
function OuterCtrl($scope){
    $scope.x = 'hello';
}
function InnerCtrl($scope){

}
```

运行后会发现跟文章开头一样的问题，里面输入框变了，外面的没跟着变。

原因在于，`InnerCtrl` 中并未定义 `x` 这个变量，取值的时候，会沿着原型链向上找，找到了 `OuterCtrl` 中定义的 `x` ，然后赋值给自己，在 `InnerCtrl` 的输入框输入值时，改变的是 `InnerCtrl` 中的 `x` ，而对 `OuterCtrl` 中的 `x` 无影响。此时，两个 `x` 是独立的。

不过，如果你不嫌麻烦的话，用 `$scope.$parent` 可以绑定并影响上一层作用域中的基本变量：

```
<input type="text" ng-model="$parent.x">
```

引用类型变量

那么，如果上下级作用域想共用变量怎么办呢？

答案是使用引用类型变量。

```
<div ng-controller="OuterCtrl">
  <p>{{x}}</p>
  <div ng-controller="InnerCtrl">
    <input type="text" ng-model="x">
  </div>
</div>
```

```
function OuterCtrl($scope){
    $scope.data = {};
    $scope.data.x = 'hello';
}
function InnerCtrl($scope){
}
```

在这种情况下，两者的 `data` 是同一个引用，对这个对象上面的属性修改，是可以反映到两级对象上的。

ng-if中的作用域

前面讲的是两级控制器之间的作用域，那跟前面提到的问题有什么关系呢？那个看着不是只有一个 Controller 吗？

其实，并不是只有 Controller 可以创建作用域，`ng-if` 等指令也会（隐式地）产生新作用域。

总结下来就是，`ng-if`、`ng-switch`、`ng-include` 等会动态创建一块界面的东西，都是自带一级作用域。

因此，在开发过程中，为了避免模板中的变量歧义，应当尽可能使用命名限定，比如 `data.x`，出现歧义的可能性就比单独的 `x` 要少得多。

总结

始终将页面中的元素绑定到对象的属性（`data.x`）而不是 直接绑定到基本变量（`x`）上。

作用域与事件

学习Angular，首先要理解其作用域机制。

Angular应用是分层的，主要有三个层面：视图，模型，视图模型。其中，视图很好理解，就是直接可见的界面，模型就是数据，那么视图模型是什么呢？是一种把数据包装给视图调用的东西。

所谓作用域，也就是视图模型中的一个概念。

根作用域

在第一章中，有这么一个很简单的数据绑定例子：

```
<input ng-model="rootA"/>
<div>{{rootA}}</div>
```

当时我们解释过，这个例子能够运行的的原因是，它的rootA变量被创建在根作用域上。每个Angular应用默认有一个根作用域，也就是说，如果用户未指定自己的控制器，变量就是直接挂在这个层级上的。

作用域在一个Angular应用中是以树的形状体现的，根作用域位于最顶层，从它往下挂着各级作用域。每一级作用域上面挂着变量和方法，供所属的视图调用。

如果想要在代码中显式使用根作用域，可以注入$rootScope。

怎么证实刚才的例子中，$rootScope确实存在，而且变量真的在它上面呢？我们来写个代码：

```
function RootService($rootScope) {
    $rootScope.$watch("rootA", function(newVal) {
        alert(newVal);
    });
}
```

这时候我们可以看到，这段代码并未跟界面产生任何关系，但里面的监控表达式确实生效了，也就是说，观测到了根作用域上rootA的变更，说明有人给它赋值了。

作用域的继承关系

在开发过程中，我们可能会出现控制器的嵌套，看下面这段代码：

```
<div ng-controller="OuterCtrl">
    <span>{{a}}</span>
    <div ng-controller="InnerCtrl">
        <span>{{a}}</span>
    </div>
</div>
```

```
function OuterCtrl($scope) {
    $scope.a = 1;
}

function InnerCtrl($scope) {
}
```

注意结果，我们可以看到界面显示了两个1，而我们只在OuterCtrl的作用域里定义了a变量，但界面给我们的结果是，两个a都有值。这里内层的a值显然来自外层，因为当我们对界面作出这样的调整之后，就只有一个了：

```
<div ng-controller="OuterCtrl">
    <span>{{a}}</span>
</div>

<div ng-controller="InnerCtrl">
    <span>{{a}}</span>
</div>
```

这是为什么呢？在Angular中，如果两个控制器所对应的视图存在上下级关系，它们的作用域就自动产生继承关系。什么意思呢？

先考虑在纯JavaScript代码中，两个构造函数各自有一个实例：

```
function Outer() {
    this.a = 1;
}

function Inner() {
}

var outer = new Outer();
var inner = new Inner();
```

在这里面添加什么代码，能够让inner.a == 1呢？

熟悉JavaScript原型的我们，当然毫不犹豫就加了一句：Inner.prototype = outer;

```
function Outer() {
    this.a = 1;
}

function Inner() {
}

var outer = new Outer();
Inner.prototype = outer;
var inner = new Inner();
```

于是就得到想要的结果了。

再回到我们的例子里，Angular的实现机制其实也就是把这两个控制器中的$scope作了关联，外层的作用域实例成为了内层作用域的原型。

以此类推，整个Angular应用的作用域，都存在自顶向下的继承关系，最顶层的是$rootScope，然后一级一级，沿着不同的控制器往下，形成了一棵作用域的树，这也就像封建社会：天子高高在上，分茅裂土，公侯伯子男，一级一级往下，层层从属。

简单变量的取值与赋值

既然作用域是通过原型来继承的，自然也就可以推论出一些特征来。比如说这段代码，点击按钮的结果是什么？

```
<div ng-controller="OuterCtrl">
    <span>{{a}}</span>
    <div ng-controller="InnerCtrl">
        <span>{{a}}</span>
        <button ng-click="a=a+1">a++</button>
    </div>
</div>
```

```
function OuterCtrl($scope) {
    $scope.a = 1;
}

function InnerCtrl($scope) {
}
```

点了按钮之后，两个a不一致了，里面的变了，外面的没变，这是为什么？原先两层不是共用一个a吗，怎么会出现两个不同的值？看这句就能明白了，相当于我们之前那个例子里，这样赋值了：

```
function Outer() {
    this.a = 1;
}

function Inner() {
}

var outer = new Outer();
Inner.prototype = outer;
var inner = new Inner();

inner.a = inner.a + 1;
```

最后这句，很有意思，它有两个过程，取值的时候，因为inner自身上面没有，所以沿着原型往上取到了1，然后自增了之后，赋值给自己，这个赋值的时候就不同了，敬爱的林副主席教导我们：有a就赋值，没有a，创造一个a也要赋值。

所以这么一来，inner上面就被赋值了一个新的a，outer里面的仍然保持原样，这也就导致了刚才看到的结果。

初学者在这个问题上很容易犯错，如果不能随时很明确地认识到这些变量的差异，很容易写出有问题的程序。既然这样，我们可以用一些别的方式来减少变量的歧义。

对象在上下级作用域之间的共享

比如说，我们就是想上下级共享变量，不创建新的，该怎么办呢？

考虑下面这个例子：

```
function Outer() {
    this.data = {
        a: 1
    };
}

function Inner() {
}

var outer = new Outer();
Inner.prototype = outer;

var inner = new Inner();

console.log(outer.data.a);
console.log(inner.data.a);

// 注意，这个时候会怎样？
inner.data.a += 1;

console.log(outer.data.a);
console.log(inner.data.a);
```

这次的结果就跟上次不同了，原因是什么呢？因为两者的data是同一个引用，对这个对象上面的属性修改，是可以反映到两级对象上的。我们通过引入一个data对象的方式，继续使用了原先的变量。把这个代码移植到AngularJS里，就变成了下面这样：

```
<div ng-controller="OuterCtrl">
    <span>{{data.a}}</span>
    <div ng-controller="InnerCtrl">
        <span>{{data.a}}</span>
        <button ng-click="data.a=data.a+1">increase a</button>
    </div>
</div>
```

```
function OuterCtrl($scope) {
    $scope.data = {
        a: 1
    };
}

function InnerCtrl($scope) {
}
```

从这个例子我们就发现了，如果想要避免变量歧义，显式指定所要使用的变量会是比较好的方式，那么如果我们确实就是要在上下级分别存在相同的变量该怎么办呢，比如说下级的点击，想要给上级的a增加1，我们可以使用$parent来指定上级作用域。

```
<div ng-controller="OuterCtrl">
    <span>{{a}}</span>
    <div ng-controller="InnerCtrl">
        <span>{{a}}</span>
        <button ng-click="$parent.a=a+1">increase a</button>
    </div>
</div>
```

```
function OuterCtrl($scope) {
    $scope.a = 1;
}

function InnerCtrl($scope) {
}
```

控制器实例别名

从Angular 1.2开始，引入了控制器实例的别名机制。在之前，可能都需要向控制器注入$scope，然后，控制器里面定义可绑定属性和方法都是这样：

```
function CtrlA($scope) {
    $scope.a = 1;
    $scope.foo = function() {
    };
}
```

```
<div ng-controller="CtrlA">
    <div>{{a}}</div>
    <button ng-click="foo()">click me</button>
</div>
```

其实
$scope$的注入是一个比较冗余的概念，没有必要把这种概念过分暴露给用户。在应用中出现的作用域，有的是充当视图模型，而有些则是处于隔离数据的需要，前者如$ng-control$
scope了，语法是这样：

```
function CtrlB() {
    this.a = 1;
    this.foo = function() {
    };
}
```

这里面，就完全没有$scope的身影了，那这个控制器怎么使用呢？

```
<div ng-controller="CtrlB as instanceB">
    <div>{{instanceB.a}}</div>
    <button ng-click="instanceB.foo()">click me</button>
</div>
```

注意我们在引入控制器的时候，加了一个as语法，给CtrlB的实例取了一个别名叫做instanceB，这样，它下属的各级视图都可以显式使用这个名称来调用其属性和方法，不易引起歧义。

在开发过程中，为了避免模板中的变量歧义，应当尽可能使用命名限定，比如a.b，出现歧义的可能性就比单独的b要少得多。

不请自来的新作用域

在一个应用中，最常见的会创建作用域的指令是ng-controller，这个很好理解，因为它会实例化一个新的控制器，往里面注入一个$scope，也就是一个新的作用域，所以一般人都会很自然地理解这里面的作用域隔离关系。但是对于另外一些情况，就有些困惑了，比如说，ng-repeat，怎么理解这个东西也会创建新作用域呢？

还是看之前的例子：

```
$scope.arr = [1, 2, 3];
```

```
<ul>
    <li ng-repeat="item in arr track by $index">{{item}}</li>
</ul>
```

在ng-repeat的表达式里，有一个item，我们来思考一下，这个item是个什么情况。在这里，数组中有三个元素，在循环的时候，这三个元素都叫做item，这时候就有个问题，如何区分每个不同的item，可能我们这个例子还不够直接，那改一下：

```
<div>outer: {{sum1}}</div>
<ul>
    <li ng-repeat="item in arr track by $index">
        {{item}}
        <button ng-click="sum1=sum1+item">increase</button>
        <div>inner: {{sum1}}</div>
    </li>
</ul>
```

这个例子运行一下，我们会发现每个item都会独立改变，说明它们确实是区分开了的。事实上，Angular在这里为ng-repeat的每个子项都创建了单独的作用域，所以，每个item都存在于自己的作用域里，互不影响。有时候，我们是需要在循环内部访问外层变量的，回忆一下，在本章的前面部分中，我们举例说，如果两个控制器，它们的视图有包含关系，内层控制器的作用域可以通过$parent来访问外层控制器作用域上的变量，那么，在这种循环里，是不是也可以如此呢？

看这个例子：

```
<div>outer: {{sum2}}</div>
<ul>
    <li ng-repeat="item in arr track by $index">
        {{item}}
        <button ng-click="$parent.sum2=sum2+item">increase</button>
        <div>inner: {{sum2}}</div>
    </li>
</ul>
```

果然是可以的。很多时候，人们会把$parent误认为是上下两级控制器之间的访问通道，但从这个例子我们可以看到，并非如此，只是两级作用域而已，作用域跟控制器还是不同的，刚才的循环可以说是有两级作用域，但都处于同一个控制器之中。

刚才我们已经提到了ng-controller和ng-repeat这两个常用的内置指令，两者都会创建新的作用域，除此之外，还有一些其他指令也会创建新的作用域，很多初学者在使用过程中很容易产生困扰。

第一章我们提到用ng-show和ng-hide来控制某个界面块的整体展示和隐藏，但同样的功能其实也可以用ng-if来实现。那么这两者的差异是什么呢，所谓show和hide，大家很好理解，就是某个东西原先有，只是控制是否显式，而if的含义是，如果满足条件，就创建这块DOM，否则不创建。所以，ng-if所控制的界面块，只有条件为真的时候才会存在于DOM树中。

除此之外，两者还有个差异，ng-show和ng-hide是不自带作用域的，而ng-if则自己创建了一级作用域。在用的时候，两者就是有差别的，比如说内部元素访问外层定义的变量，就需要使用类似ng-repeat那样的$parent语法了。

相似的类型还有ng-switch，ng-include等等，规律可以总结，也就是那些会动态创建一块界面的东西，都是自带一级作用域。

“悬空”的作用域

一般而言，在Angular工程中，基本是不需要手动创建作用域的，但真想创建的话，也是可以做到的。在任意一个已有的作用域上调用$new()，就能创建一个新的作用域：

```
var newScope = scope.$new();
```

刚创建出来的作用域是一个“悬空”的作用域，也就是说，它跟任何界面模板都不存在绑定关系，创建它的作用域会成为它的$parent$。这种作用域可以经过compile阶段，与某视图模板进行融合。

为了帮助理解，我们可以用DocumentFragment作类比，当作用域被创建的时候，就好比是创建了一个DocumentFragment，它是不在DOM树上的，只有当它被append到DOM树上，才能够被当做普通的DOM来使用。

那么，悬空的作用域是不是什么用处都没有呢？也不是，尽管它未与视图关联，但是它的一些方法仍然可以用。

我们在第一章里提到了$watch，这就是定义在作用域原型上的。如果我们想要监控一个数据的变化，但这个数据并非绑定到界面上的，比如下面这样，怎么办？

```
function IsolateCtrl($scope) {
    var child = {
        a: 1
    };

    child.a++;
}
```

注意这个child，它并未绑定到$scope上，如果我们想要在a变化的时候做某些事情，是没有办法做的，因为直到最近的某些浏览器中，才实现了Object.observe这样的对象变更观测方法，之前某些浏览器中要做这些，会比较麻烦。

但是我们的$watch$和eval之类的方法，其实都是实现在作用域对象上的，也就是说，任何一个作用域，即使没有与界面产生关联，也是能够使用这些方法的。

```
function IsolateCtrl($scope) {
    var child = $scope.$new();
    child.a = 1;

    child.$watch("a", function(newValue) {
        alert(newValue);
    });

    $scope.change = function() {
        child.a++;
    };
}
```

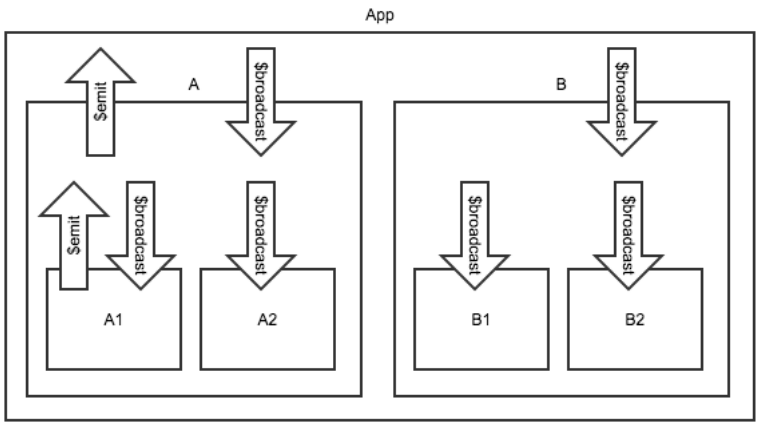
这时候child里面a的变更就可以被观测到，并且，这个child只有本作用域可以访问到，相当于是一个增强版的数据模型。如果我们要做一个小型流程引擎之类的东西，作用域对象上提供的这些方法会很有用。

作用域上的事件

我们刚才提到使用$parent来处理上下级的通讯，但其实这不是一种好的方式，尤其是在不同控制器之间，这会增加它们的耦合，对组件复用很不利。那怎样才能更好地解耦呢？我们可以使用事件。

提到事件，可能很多人想到的都是DOM事件，其实DOM事件只存在于上层，而且没有业务含义，如果我们想要传递一个明确的业务消息，就需要使用业务事件。这种所谓的业务事件，其实就是一种消息的传递。

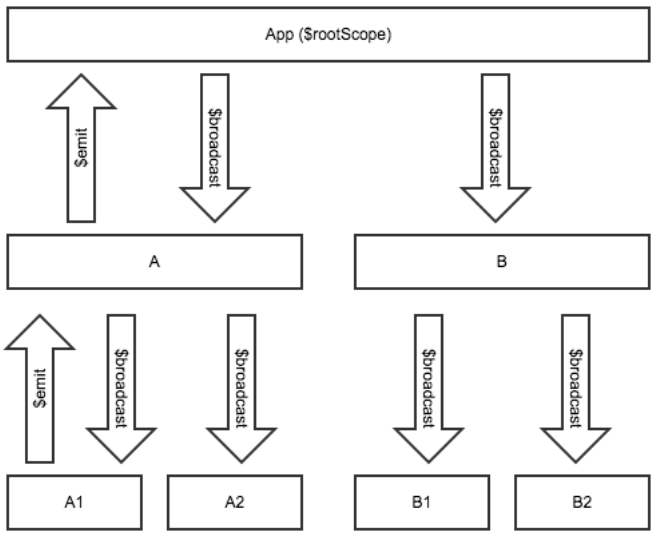
假设有如图所示的应用：



这张图中有一个应用，下面存在两个视图块A和B，它们分别又有两个子视图。这时候，如果子视图A1想要发出一个业务事件，使得B1和B2能够得到通知，过程就会是：

- 沿着父作用域一路往上到达双方共同的祖先作用域
- 从祖先作用域一级一级往下进行广播，直到到达需要的地方

刚才的图形体现了界面的包含关系，如果把这个图再立体化，就会是下面这样：



对于这种事件的传播方式，可以有个类似的比喻：

比如说，某军队中，1营1连1排长想要给1营2连下属的三个排发个警戒通知，他的通知方向是一级一级向上汇报，直到双方共同的上级，也就是1营指挥人员这里，然后再沿着二连这个路线向下去通知。

- 从作用域往上发送事件，使用scope.$emit

```
$scope.$emit("someEvent", {});
```

- 从作用域往下发送事件，使用scope.$broadcast

```
$scope.$broadcast("someEvent", {});
```

这两个方法的第二个参数是要随事件带出的数据。

注意，这两种方式传播事件，事件的发送方自己也会收到一份。

使用事件的主要作用是消除模块间的耦合，发送方是不需要知道接收方的状况的，接收方也不需要知道发送方的状况，双方只需要传送必要的业务数据即可。

事件的接收与阻止

无论是$emit$还是broadcast发送的事件，都可以被接收，接收这两种事件的方式是一样的：

```
$scope.$on("someEvent", function(e) {
    // 这里从e上可以取到发送过来的数据
});
```

注意，事件被接收了，并不代表它就中止了，它仍然会沿着原来的方向继续传播，也就是：

- $emit的事件将继续向上传播
- $broadcast的事件将继续向下传播

有时候，我们希望某一级收到事件之后，就让它停下来，不再传播，可以把事件中止。这时候，两种事件的区别就体现出来了，只有$emit$发出的事件是可以被中止的，broadcast发出的不可以。

如果想要阻止$emit事件的继续传播，可以调用事件对象的stopPropagation()方法。

```
$scope.$on("someEvent", function(e) {
    e.stopPropagation();
});
```

但是，想要阻止$broadcast事件的传播，就麻烦了，我们只能通过变通的方式：

首先，调用事件对象的preventDefault()方法，然后，在收取这个事件对象的时候，判断它的defaultPrevented属性，如果为true，就忽略此事件。这个过程比较麻烦，其实我们一般是不需要管的，只要不监听对应的事件就可以了。在实际使用过程中，也应当尽量少使用事件的广播，尤其是从较高的层级进行广播。

上级作用域

```
$scope.$on("someEvent", function(e) {
    e.preventDefault();
});
```

下级作用域

```
$scope.$on("someEvent", function(e) {
    if (e.defaultPrevented) {
        return;
    }
});
```

事件总线

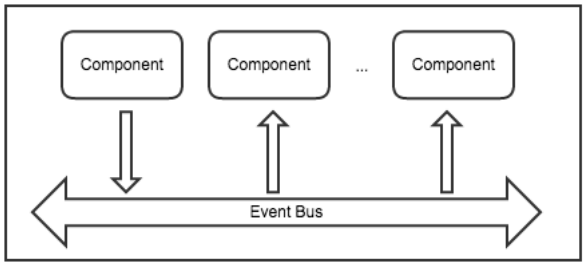
在Angular中，不同层级作用域之间的数据通信有多种方式，可以通过原型继承的一些特征，也可以收发事件，还可以使用服务来构造单例对象进行通信。

前面提到的这个军队的例子，有些时候沟通效率比较低，特别是层级多的时候。想象一下，刚才这个只有三层，如果更复杂，一个排长的消息都一定要报告到军长那边再下发到其他基层主官，必定贻误军情，更何况有很多下级根本不需要知道这个消息。 那怎么办呢，难道是直接打电话沟通吗？这个效率高是高，就是容易乱，这也就相当于界面块之间的直接通过id调用。

Angular的作用域树类似于传统的组织架构树，一个大型企业，一般都会有若干层级，近年来有很多管理的方法论，比如说组织架构的扁平化。

我们能不能这样：搞一个专门负责通讯的机构，大家的消息都发给它，然后由它发给相关人员，其他人员在理念上都是平级关系。

这就是一个很典型的订阅发布模式，接收方在这里订阅消息，发布方在这里发布消息。这个过程可以用这样的图形来表示：



代码写起来也很简单，把它做成一个公共模块，就可以被各种业务方调用了：

```
app.factory("EventBus", function() {
    var eventMap = {};

    var EventBus = {
        on : function(eventType, handler) {
            //multiple event listener
            if (!eventMap[eventType]) {
                eventMap[eventType] = [];
            }
            eventMap[eventType].push(handler);
        },

        off : function(eventType, handler) {
            for (var i = 0; i < eventMap[eventType].length; i++) {
                if (eventMap[eventType][i] === handler) {
                    eventMap[eventType].splice(i, 1);
                    break;
                }
            }
        },

        fire : function(event) {
            var eventType = event.type;
            if (eventMap && eventMap[eventType]) {
                for (var i = 0; i < eventMap[eventType].length; i++) {
                    eventMap[eventType][i](event);
                }
            }
        }
    };
    return EventBus;
});
```

事件订阅代码：

```
EventBus.on("someEvent", function(event) {
    // 这里处理事件
    var c = event.data.a + event.data.b;
});
```

事件发布代码：

```
EventBus.fire({
    type: "someEvent",
    data: {
        aaa: 1,
        bbb: 2
    }
});
```

注意，如果在复杂的应用中使用事件总线，需要慎重规划事件名，推荐使用业务路径，比如："portal.menu.selectedMenuChange"，以避免事件冲突。

小结

在本章，我们学习了作用域相关的知识，以及它们之间传递数据的方式。作用域在整个Angular应用中形成了一棵树，以$rootScope为根部，开枝散叶。这棵树独立于DOM而存在，又与DOM相关联。事件在整个树上传播，如蜂飞蝶舞。

总体来说，使用AngularJS对JavaScript的基本功是有一定要求的，因为这里面大部分实现都依赖于纯JavaScript语法，比如原型继承的使用。如果对这一块有充分的认识，理解Angular的作用域就会比较容易。

一个大型单页应用，需要对部件的整合方式和通信机制作良好的规划，为它们建立良好的秩序，这对于确保整个应用的稳定性是非常必要的。