单向绑定（ng-bind）和双向绑定（ng-model）的区别

`ng-bind` 单向数据绑定（ `$scope -> view` ），用于数据显示，简写形式是 `{{}}`。

`ng-bind`和 `{{}}` 两者的区别在于页面没有加载完毕， `{{val}}` 会直接显示到页面，直到 `Angular` 渲染该绑定数据（这种行为有可能将 `{{val}}` 让用户看到）；而 `ng-bind` 则是在`Angular` 渲染完毕后将数据显示。

`ng-model` 是双向数据绑定（ `$scope->viewandview->$scope->viewandview->$scope` ），用于绑定值会变化的表单元素等。

双向数据绑定是`AngularJS` 的核心机制之一。当 `view` 中有任何数据变化时，会更新到 `model` ，当 `model` 中数据有变化时， `view` 也会同步更新，显然，这需要一个监控。

双向数据绑定原理

`Angular`在`Scope` 模型上设置了一个 监听队列，用来监听数据变化并更新 `view` 。

每次绑定一个东西到 `view` 上时 `AngularJS` 就会往 `watch` 队列里插入一条 `watch` 队列里插入一条 `watcher` ，用来检测它监视的 `Model` 里是否有变化的东西。

当你写下表达式如 `{{ val }}` 时， `AngularJS` 在幕后会为你在 `Scope` 模型上设置一个 `watcher` （表达式将被 `Angular` 编译成一个监视函数），它用来在数据发生变化的时候更新 `view` 。这里的 `watcher` 和你会在 `AngularJS` 中设置的 `watcher` 是一样的：

```
$scope.$watch('val', function(newValue, oldValue) {
  //update the DOM with newValue
});
```

- 将数据附加到 `Scope` 上，数据自身不会对性能产生影响，如果没有监视器来监视这个属性，那个这个属性在不在 `Scope` 上是无关重要的； `Angular` 并不会遍历 `Scope` 上的属性，它将遍历所有的观察器。
- 每个监视函数是在每次 `$digest` 过程中被调用的。因此，我们要注意观察器的数量以及每个监视函数或者监视表达式的性能。

$digest循环是在什么时候以各种方式开始的？

`$digest` 的作用是检测当前 `Scope` 以及子 `Scope` 中所有的 `watcher` ，因为监听函数会在执行过程中修改 `Model`（ `Scope` 中的变量）， `$digest()` 会一直被调用直到 `Model` 没有再变。当调用超过 `10` 次时， `$digest()` 会抛出一个异常 `"Maximum iteration limit exceeded"` ，以此来防止程序进入一个死循环。

当浏览器接收到可以被 `angular context` 处理的事件时， `$digest` 循环就会触发，遍历所有的 `$digest` 循环就会触发，遍历 `watch` 队列里所有的 `watcher` ，最后更新 `dom` 。

```
<button ng-click="val=val+1">increase 1</button>
```

`click` 时会产生一次更新的操作（至少触发两次 `$digest` 循环）：

- 按下按钮
- 浏览器接收到一个事件，进入到 `angular context`
- `$digest` 循环开始执行，查询每个 `digest` 循环开始执行，查询每个 `watcher` 是否变化

- 由于监视 `$scope.val` 的 `$scope.val` 的 `watcher` 报告了变化，因此强制再执行一次 `$digest` 循环
- 新的 `$digest` 循环未检测到变化
- 浏览器拿回控制器，更新 `$scope.val` 新值对应的 `dom`

在调用了 `Scope.digest()` 后， `$digest` 循环就开始了。

假设你在一个 `ng-click` 指令对应的 `handler` 函数中更改了 `Scope` 中的一条数据，此时 `AngularJs` 会自动地通过调用 `$digest` 循环。

当 `$digest` 循环开始后，它会触发每个 `watcher` 。这些 `watchers` 会检查 `Scope` 中的当前 `Model` 值是否和上一次计算得到的 `Model` 值不同。如果不同，那么对应的回调函数会被执行。调用该函数的结果，就是 `view` 中的表达式内容(如 `{{ val }}`)会被更新。

除了 `ng-click` 指令，还有一些其它的 `built-in` 指令以及服务来让你更改 `Models` (比如 `ng-model` ， `timeout` 等)会自动触发一次 `$digest` 循环。

在上面的例子中，**AngularJs并不直接调用 `digest()` ，而是调用 `Scope.apply()` ，后者会调用 `$rootScope.$digest()` 。**因此，一轮 `$digest` 循环在 `$digest` 循环在 `$rootScope` 开始，随后会访问到所有的 `children Scope` 中的 `watchers` 。

通常写代码时我们无需主动调用 `apply` 或 `digest` ，是因为 `angular` 在外部对我们的回调函数做了包装。例如常用的 `ng-click` ，这是一个指令（ `directive` ），内部实现则类似于：

```
DOM.addEventListener('click', function ($scope) {
  $scope.$apply(() => userCode());
});
```

可以看到： `ng-click` 帮我们做了 `apply` 这个操作。类似的不只是这些事件回调函数，还有 `apply` 这个操作。类似的不只是这些事件回调函数，还有 `http` 、 `timeout` 等。很多人抱怨说 `angular` 这个库太大了什么都管，其实你可以不用它自带的这些服务（ `service` ），只要你记得手工调用 `Scope.$apply` 。

现在，假设你将 `ng-click` 指令关联到了一个 `button` 上，并传入了一个 `function` 名到 `ng-click` 上。当该 `button` 被点击时， `AngularJs` 会将此 `function` 包装到一个 `wrapping function` 中，然后传入到 `Scope.apply()` 。因此，你的 `function` 会正常被执行，修改 `Models` (如果需要的话)，此时一轮 `$digest` 循环也会被触发，用来确保 `view` 也会被更新。

`Scope.apply()` 会自动地调用 `$rootScope.$digest()` 。 `$apply()` 方法有两种形式。第一种会接受一个 `function` 作为参数，执行该 `function` 并且触发一轮 `$digest` 循环。第二种会不接受任何参数，只是触发一轮 `$digest` 循环。我们马上会看到为什么第一种形式更好。

$digest 循环会运行多少次？

`$digest` 循环的上限是 10 次（超过 10次后抛出一个异常，防止无限循环）。

`$digest` 循环不会只运行一次。在当前的一次循环结束后，它会再执行一次循环用来检查是否有 `Models` 发生了变化。

这就是**脏检查（Dirty Checking）**，它用来处理在 **listener** 函数被执行时可能引起的 `Model` 变化。因此 `$digest` 循环会持续运行直到 `Model` 不再发生变化，或者 `$digest` 循环会持续运行直到 `Model` 不再发生变化，或者 `$digest` 循环的次数达到了 10 次（超过 10 次后抛出一个异常，防止无限循环）。

当 `$digest` 循环结束时，DOM 相应地变化。

脏检查是`$digest`执行的，另一个更常用的用于触发脏检查的函数`$apply`——其实就是`$digest`的一个简单封装（还做了一些抓异常的工作）。

`$apply()`和`$digest()`的区别？

`$apply`是`Scope`（或者是`direcvie`里的`link`函数中的`Scope`）的一个函数，调用它会强制一次`$digest`循环（除非当前正在执行循环，这种情况下会抛出一个异常，这是我们不需要在那里执行`$digest`循环）。

`$apply()`和`$digest()`有两个区别。

- 最直接的差异是，`$apply`可以带参数，它可以接受一个函数，然后在应用数据之后，调用这个函数。所以，一般在集成非`Angular`框架（比如`jQuery`）的代码时，可以把代码写在这个里面调用。
- 当调用`$digest`的时候，只触发当前作用域和它的子作用域上的监控，但是当调用`$apply`的时候，会触发作用域树上的所有监控。

什么时候手动调用 $apply() 方法？

取决于是否在`Angular`上下文环境（`angular context`）。

`AngularJS`对此有着非常明确的要求，就是它只负责对发生于**AngularJS上下文环境**中的变更会做出自动地响应(即，在`apply()`方法中发生的对于`Models`的更改)。

`AngularJs`的`built-in`指令就是这样做的，所以任何的`model`变更都会被反映到`view`中。但是，如果你在**AngularJS上下文之外**的任何地方修改了`Model`，那么你就需要通过手动调用`apply()`方法中发生的对于`Models`的更改。这就像告诉`AngularJs`，你修改了一些`Models`，希望`AngularJs`帮你触发`watchers`来做出正确的响应。

典型的需要调用`$apply()`方法的场景是：

- 使用了`JavaScript`中的`setTimeout()`来更新一个`scope model`
- 用指令设置一个`DOM`事件`listener`并且在该`listener`中修改了一些`models`

```
$scope.setMsg = function() {
    setTimeout(function() {
        $scope.message = 'hello world';
        console.log('message:' + $scope.message);
    }, 2000);
}
$scope.setMsg();
```

运行这个例子，会看到过了两秒钟之后，控制台确实会显示出已经更新的`model`，然而，`view`并没有更新。

```
$scope.getMessage = function() {
    setTimeout(function() {
        $scope.$apply(function() {
            $scope.message = 'hello world';
            console.log('message:' + $scope.message);
        });
    }, 2000);
}
```

再运行就OK了。 不过，在 `AngularJS` 中应该尽量使用 `timeoutService` 来代替 `setTimeout()` ，因为前者会帮你调用 `timeoutService` 来代替 `setTimeout()` ，因为前者会帮你调用 `apply()` ，让你不需要手动地调用它（ `$timeout()` , `$interval()` ）。

```
app.directive("inc", function() {
    return function (scope, element, attr) {
        element.on("click", function() {
            scope.val++;
        });
    };
});
```

跟场景一的结果一样，这个时候，点击按钮，界面上的数字并不会增加。但查看调试器，发现数据确实已经增加了。在 `scope.val++` 一行后面添加 `$scope.$apply()` 或者 `$scope.$digest()` 就 OK 了。

$apply()方法的两种形式

```
//无参
$scope.$apply()
//有参
$scope.$apply(function(){
    ...
})
```

应该总使用接受一个 `function` 作为参数的 `$apply()` 方法。这是因为当传入一个 `function` 到 `$apply()` 方法的时候，这个 `function` 会被包装到一个 `try…catch` 块中，所以一旦有异常发生，该异常会被 `$exceptionHandler service` 处理。

想象一下如果有个 `alert` 框显示错误给用户，然后有个第三方的库进行一个网络调用然后失败了，如果不把它封装进 `$apply` 里面， `Angular` 永远不会知道失败了， `alert` 框就永远不会弹出来了。

在 AngularJS 中使用 $watch注意事项？

如果要监听的是一个对象，那还需要第三个参数

```
$scope.data.name = 'htf';
$scope.$watch('data', function(newValue, oldValue) {
    if (newValue === oldValue) { return; }
    $scope.updated++;
}, true);
```

表示比较的是对象的值而不是引用，如果不加第三个参数 true ，在 data.name 变化时，不会触发相应操作，因为引用的是同一引用。

脏检查的范围

`angular` 会对所有绑定到 `UI` 上的表达式做脏检查。其实，在 `angular` 实现内部，所有绑定表达式都被转换为 `$scope.$watch` 。每个 `watcher` 记录了上一次表达式的值。有 `ng-bind='a'` 即有 `$scope.watch('a',callback)` ，而 `$scope.$watch` 可不会管 `watch` 的表达式是否跟除非脏检查的事件有关。

```
<div ng-show="false">
  <span id="span1" ng-bind="content"></span>
</div>
<span id="span2" ng-bind="content"></span>
<button ng-click="">test</button>
```

问：点击 `test` 这个按钮时会触发脏检查吗？触发几次？

首先：`ng-click=""` 什么都没有做。`angular` 会因为这个事件回调函数什么都没做就不进行脏检查吗？不会。

然后：`#span1` 被隐藏掉了，会检查绑定在它上面的表达式吗？尽管用户看不到，但是 `$scope.watch('content', callback)` 还在。就算你直接把这个 `span` 元素干掉，只要 `watch` 表达式还在，要检查的还会检查。

再次：重复的表达式会重复检查吗？会。

最后：别忘了 `ng-show="false"`。可能是因为 `angular` 的开发人员认为这种绑定常量的情况并不多见，所以 `$watch` 并没有识别所监视的表达式是否是常量。常量依旧会重复检查。

所以：触发三次。一次 `false`，一次 `content`，一次 `content`

所以说一个绑定表达式只要放在当前 DOM 树里就会被监视，不管它是否可见，不管它是否被放在另一个 Tab 里，更不管它是否与用户操作相关，都会进行脏检查。

另外，就算在不同 `Controller` 里构造的 scope也会互相影响，别忘了 `angular` 还有全局的 `rootScope`，你还可以 `$scope.emit`。`angular` 无法保证你绝对不会在一个 `controller` 里更改另一个 `controller` 生成的 `scope`，包括 自定义指令（`Directive`）生成的 `scope` 和 `Angular 1.5` 里新引入的组件（`Component`）。

所以说不要怀疑用户在输入表单时 `angular` 会不会监听页面左边导航栏的变化。

如何优化脏检查与运行效率

脏检查慢吗？

说实话脏检查效率是不高，但是也谈不上有多慢。简单的数字或字符串比较能有多慢呢？十几个表达式的脏检查可以直接忽略不计；上百个也可以接受；成百上千个就有很大问题了。绑定大量表达式时请注意所绑定的表达式效率。建议注意一下几点：

- 表达式（以及表达式所调用的函数）中少写太过复杂的逻辑。
- 不要连接太长的 `filter`（往往 `filter` 里都会遍历并且生成新数组）。
- 不要访问 `DOM` 元素。

1. 使用单次绑定减少绑定表达式数量单次绑定（`One-time binding` 是 `Angular 1.3` 就引入的一种特殊的表达式，它以 `::` 开头，当脏检查发现这种表达式的值不为 `undefined` 时就认为此表达式已经稳定，并取消对此表达式的监视。这是一种行之有效的减少绑定表达式数量的方法，与 `ng-repeat` 连用效果更佳（下文会提到），但过度使用也容易引发 `bug`。
2. 善用 `ng-if` 减少绑定表达式的数量

`ng-if` 不仅可以减少 `DOM` 树中元素的数量（而非像 `ng-hide` 那样仅仅只是加个 `display: none`），每一个 `ng-if` 拥有自己的 `Scope`，`ng-if` 下面的 `$watch` 表达式都是注册在 `ng-if` 自己 `Scope` 中。当 `ng-if` 变为 `false`，`ng-if` 下的 `Scope` 被销毁，注册在这个 `Scope` 里的绑定表达式也就随之销毁了。

```
<ul>
  <li ng-class="{ selected: selectedTab === 1 }">Tab 1 title</li>
  <li ng-class="{ selected: selectedTab === 2 }">Tab 2 title</li>
  <li ng-class="{ selected: selectedTab === 3 }">Tab 3 title</li>
  <li ng-class="{ selected: selectedTab === 4 }">Tab 4 title</li>
</ul>
<div ng-show="selectedTab === 1">[[Tab 1 body...]]</div>
<div ng-show="selectedTab === 2">[[Tab 2 body...]]</div>
<div ng-show="selectedTab === 3">[[Tab 3 body...]]</div>
<div ng-show="selectedTab === 4">[[Tab 4 body...]]</div>
```

对于这种会反复隐藏、显示的元素，通常人们第一反应都是使用 `ng-show` 或 `ng-hide` 简单的用 `display: none` 把元素设置为不可见。然而入上文所说，肉眼不可见不代表不会跑脏检查。如果将 `ng-show` 替换为 `ng-if` 或 `ng-switch-when`。

```
<div ng-if="selectedTab === 1">[[Tab 1 body...]]</div>
<div ng-if="selectedTab === 2">[[Tab 2 body...]]</div>
<div ng-if="selectedTab === 3">[[Tab 3 body...]]</div>
<div ng-if="selectedTab === 4">[[Tab 4 body...]]</div>
```

有如下优点：

- 首先 DOM 树中的元素个数显著减少至四分之一，降低内存占用。
- 其次 `$watch` 表达式也减少至四分之一，提升脏检查循环的速度。
- 如果这个 `tab` 下面有 `controller`（例如每个 `tab` 都被封装为一个组件），那么仅当这个 `tab` 被选中时该 `controller` 才会执行，可以减少各页面的互相干扰。
- 如果 `controller` 中调用接口获取数据，那么仅当对应 `tab` 被选中时才会加载，避免网络拥挤。

当然也有缺点：

- DOM 重建本身费时间。
- 如果 `tab` 下有 `controller`，那么每次该 `tab` 被选中时 `controller` 都会被执行。
- 如果在 `controller` 里面调接口获取数据，那么每次该 `tab` 被选中时都会重新加载。
- 各位读者自己取舍。

3. 给 `ng-repeat` 手工添加 `track by`

不恰当的 `ng-repeat` 会造成 `DOM` 树反复重新构造，拖慢浏览器响应速度，造成页面闪烁。除了上面这种比较极端的情况，如果一个列表频繁拉取 `Server` 端数据自刷新的话也一定要手工添加 `track by`，因为接口给前端的数据是不可能包含 `$$hashKey` 这种东西的，于是结果就造成列表频繁的重建。

其实不必考虑那么多，总之加上没坏处，至少可以避免 `angular` 生成 `$$hashKey` 这种奇奇怪怪的东西。

脏检测的利弊？

很多人对 `Angular` 的脏检测机制感到不屑，推崇基于 `setter`，`getter` 的观测机制，在我看来，这只是同一个事情的不同实现方式，并没有谁完全胜过谁，两者是各有优劣的。 **最好的办法是了解各自使用方式的差异，考虑出它们性能的差异所在，在不同的业务场景中，避开最容易造成性能瓶颈的用法。**

ng-if跟ng-show/hide的区别有哪些？

第一点区别是，`ng-if` 在后面表达式为 `true` 的时候才创建这个 `dom` 节点，`ng-show` 是初始时就创建了，用 `display:block` 和 `display:none` 来控制显示和不显示。第二点区别是，`ng-if` 会（隐式地）产生新作用域，`ng-switch`、`ng-include` 等会动态创建一块界面的也是如此。

ng-repeat迭代数组的时候，如果数组中有相同值，会有什么问题，如何解决？

会提示 `Duplicates in a repeater are not allowed` ，加`track by $index`可解决。当然，也可以 `trace by` 任何一个普通的值，只要能唯一性标识数组中的每一项即可（建立 `dom` 和数据之间的关联）。

ng-click中写的表达式，能使用JS原生对象上的方法，比如Math.max之类的吗？为什么？

不可以。只要是在页面中，就不能直接调用原生的 `JS` 方法，因为这些并不存在于与页面对应的 `Controller` 的 `Scope` 中。除非在 `Scope` 中添加了这个函数：

```
$scope.parseInt = function(x){
    return parseInt(x);
}
```

factory和service，provider是什么关系？

`factory` 把 `service` 的方法和数据放在一个对象里，并返回这个对象；`service` 通过构造函数方式创建 `service`，返回一个实例化对象；`provider` 创建一个可通过 `config` 配置的 `service`。从底层实现上来看，`service` 调用了 `factory`，返回其实例；`factory` 调用了 `provider`，将其定义的内容放在 `$get` 中返回。`factory` 和 `service` 功能类似，只不过 `factory` 是普通 `function`，可以返回任何东西（`return` 的都可以被访问，所以那些私有变量怎么写你懂的）；`service` 是构造器，可以不返回（绑定到 `this` 的都可以被访问）；`provider` 是加强版 `factory`，返回一个可配置的 `factory`。

详述angular的“依赖注入”。

`AngularJS` 是通过**构造函数的参数名字**来推断**依赖服务名称**的，通过 `toString()` 来找到这个定义的 `function` 对应的字符串，然后用正则解析出其中的参数（依赖项），再去依赖映射中取到对应的依赖，实例化之后传入。因为 `AngularJS` 的 `injector` 是假设函数的参数名就是依赖的名字，然后去查找依赖项。

所以，通常会使用下面两种方式注入依赖（对依赖添加的顺序有要求）。

数组注释法：

```
myApp.controller('myCtrl', ['$scope', '$http', function($scope, $http){
    ...
}])
```

显式 `$inject` ：

```
myApp.controller('myCtrl', myCtrl);
function myCtrl = ($scope, $http){
    ...
}
myCtrl.$inject = ['$scope', '$http'];
```

对于一个 `DI` 容器，必须具备三个要素：依赖项的注册，依赖关系的声明和对象的获取。在 `AngularJS` 中，`module` 和 `$provide` 都可以提供依赖项的注册；内置的 `injector` 可以获取对象（自动完成依赖注入）；依赖关系的声明，就是上面的那两种方式。