目录

[1 前言 4](#_Toc378153082)

[2 AngularJS概述 4](#_Toc378153083)

[2.1 AngularJS是什么？ 4](#_Toc378153084)

[2.2 AngularJS简单介绍 5](#_Toc378153085)

[2.3 什么时候该用AngularJS 5](#_Toc378153086)

[3 AugularJS特性 5](#_Toc378153087)

[3.1 特性一：双向的数据绑定 5](#_Toc378153088)

[3.2 特性二：模板 6](#_Toc378153089)

[3.3 特性三：MVC 7](#_Toc378153090)

[3.4 特性四：服务和依赖注入 7](#_Toc378153091)

[3.5 特性五：指令（Directives） 8](#_Toc378153092)

[4 功能介绍 9](#_Toc378153093)

[4.1数据绑定 9](#_Toc378153094)

[4.2 scopes、module、controller 10](#_Toc378153095)

[4.2.1 scopes 10](#_Toc378153096)

[4.2.2 module 10](#_Toc378153097)

[4.2.3 ng-controller 10](#_Toc378153098)

[4.3 ajax 11](#_Toc378153099)

[4.4表达式 12](#_Toc378153100)

[4.5过滤器 12](#_Toc378153101)

[4.5.1过滤器使用方式 13](#_Toc378153102)

[4.5.2 ng的内置过滤器 13](#_Toc378153103)

[4.5.3自定义过滤器及示例 15](#_Toc378153104)

[4.6指令 16](#_Toc378153105)

[4.6.1样式相关的指令 17](#_Toc378153106)

[4.6.2表单控件功能相关指令 18](#_Toc378153107)

[4.6.3事件绑定相关指令 18](#_Toc378153108)

[4.6.4特殊的ng-src和ng-href 19](#_Toc378153109)

[4.6.5 示例 20](#_Toc378153110)

[4.7服务 21](#_Toc378153111)

[4.7.1服务（service）介绍 21](#_Toc378153112)

[4.7.2自定义服务 21](#_Toc378153113)

[4.7.3管理服务的依赖关系 22](#_Toc378153114)

[4.7.4 示例 23](#_Toc378153115)

[4.8依赖注入DI 24](#_Toc378153116)

[4.9路由 26](#_Toc378153117)

[4.9.1 Route内容 26](#_Toc378153118)

[4.9.2 路由机制 26](#_Toc378153119)

[4.9.3 示例 27](#_Toc378153120)

[4.10 NG动画效果 29](#_Toc378153121)

[4.10.1 NG动画效果简介 29](#_Toc378153122)

[4.10.2 示例 30](#_Toc378153123)

[5 功能演示 30](#_Toc378153124)

[6 AngularJS进阶 31](#_Toc378153125)

[6.1数据绑定原理研究 31](#_Toc378153126)

[6.1.1 AngularJS扩展事件循环 31](#_Toc378153127)

[6.1.2 $watch 队列（$watch list） 31](#_Toc378153128)

[6.1.3 $digest循环 32](#_Toc378153129)

[6.1.4如何进入angular context 33](#_Toc378153130)

[6.1.5使用$watch来监视 34](#_Toc378153131)

[6.1.6 总结 36](#_Toc378153132)

[6.2自定义指令详解 36](#_Toc378153133)

[6.2.1指令的编译过程 37](#_Toc378153134)

[6.2.2指令的使用方式及命名方法 37](#_Toc378153135)

[6.2.3自定义指令的配置参数 37](#_Toc378153136)

[6.2.3指令的表现参数restrict等 38](#_Toc378153137)

[6.2.4指令的行为参数：compile和link 40](#_Toc378153138)

[6.2.5指令的划分作用域参数：scope 42](#_Toc378153139)

[6.2.6指令间通信参数：controller和require 45](#_Toc378153140)

[6.3 性能及调优 47](#_Toc378153141)

[6.3.1性能测试 47](#_Toc378153142)

[6.3.2七大调优法则 48](#_Toc378153143)

# 1 前言

AngularJS是google在维护，其在国外已经十分火热，现在最新版本已经到了angular4，但是考虑到实际项目中的使用情况，而且angular的插件之丰富，所以这里还是只是对angular1进行学习和研究，这里做一个学习记录，便于以后查阅。

# 2 AngularJS概述

## 2.1 AngularJS是什么？

AngularJs（后面就简称ng了）是一个用于设计动态web应用的结构框架。首先，它是一个框架，不是类库，是像EXT一样提供一整套方案用于设计web应用。它不仅仅是一个javascript框架，因为它的核心其实是对HTML标签的增强。

何为HTML标签增强？其实就是使你能够用标签完成一部分页面逻辑，具体方式就是通过自定义标签、自定义属性等，这些HTML原生没有的标签/属性在ng中有一个名字：指令（directive）。后面会详细介绍。那么，什么又是动态web应用呢？与传统web系统相区别，web应用能为用户提供丰富的操作，能够随用户操作不断更新视图而不进行url跳转。ng官方也声明它更适用于开发CRUD应用，即数据操作比较多的应用，而非是游戏或图像处理类应用。

为了实现这些，ng引入了一些非常棒的特性，包括模板机制、数据绑定、模块、指令、依赖注入、路由。通过数据与模板的绑定，能够让我们摆脱繁琐的DOM操作，而将注意力集中在业务逻辑上。

　 另外一个疑问，ng是MVC框架吗？还是MVVM框架？官网有提到ng的设计采用了MVC的基本思想，而又不完全是MVC，因为在书写代码时我们确实是在用ng-controller这个指令(起码从名字上看，是MVC吧)，但这个controller处理的业务基本上都是与view进行交互，这么看来又很接近MVVM。让我们把目光移到官网那个非醒目的title上：“AngularJS — Superheroic JavaScript MVW Framework”。

## 2.2 AngularJS简单介绍

AngularJS 重新定义了前端应用的开发方式。面对HTML和JavaScript之间的界线，它非但不畏缩不前，反而正面出击，提出了有效的解决方案。

很多前端应用的开发框架，比如Backbone、EmberJS等，都要求开发者继承此框架特有的一些JavaScript对象。这种方式有其长处，但它不必要地污染了开发者自己代码的对象空间，还要求开发者去了解内存里那些抽象对象。尽管如此我们还是接受了这种方式，因为网络最初的设计无法提供 我们今天所需的交互性，于是我们需要框架，来帮我们填补JavaScript和HTML之间的鸿沟。而且有了它，你不用再“直接”操控DOM，只要给你的DOM注上metadata（即AngularJS里的directive们），然后让AngularJS来帮你操纵DOM。同时，AngularJS不依赖（也不妨碍）任何其他的框架。你甚至可以基于其它的框架来开发AngularJS应用。

API地址：<http://docs.angularjs.org/api/>；

AngularJS最新中文版地址：https://angular.cn/docs/ts/latest/。

## 2.3 什么时候该用AngularJS

AngularJS是一个 MV\* 框架，最适于开发客户端的单页面应用。它不是个功能库，而是用来开发动态网页的框架。它专注于扩展HTML的功能，提供动态数据绑定（data binding），而且它能跟其它框架（如jQuery）合作融洽。

如果你要开发的是单页应用，AngularJS就是你的上上之选。Gmail、Google Docs、Twitter和Facebook这样的应用，都很能发挥AngularJS的长处。但是像游戏开发之类对DOM进行大量操纵、又或者单纯需要 极高运行速度的应用，就不是AngularJS的用武之地了。

# 3 AugularJS特性

AngularJS是一个新出现的强大客户端技术，提供给大家的一种开发强大应用的方式。这种方式利用并且扩展HTML，CSS和javascript，并且弥补了它们的一些非常明显的不足。本应该使用HTML来实现而现在由它开发的动态一些内容。

AngularJS有五个最重要的功能和特性：

## 3.1 特性一：双向的数据绑定

数据绑定可能是AngularJS最酷最实用的特性。它能够帮助你避免书写大量的初始代码从而节约开发时间。一个典型的web应用可能包含了80%的代码用来处理，查询和监听DOM。数据绑定是的代码更少，你可以专注于你的应用。

我们想象一下Model是你的应用中的简单事实。你的Model是你用来读取或者更新的部分。数据绑定指令提供了你的Model投射到view的方法。这些投射可以无缝的，毫不影响的应用到web应用中。

传统来说，当model变化了。 开发人员需要手动处理DOM元素并且将属性反映到这些变化中。这个一个双向的过程。一方面，model变化驱动了DOM中元素变化，另一方面，DOM元素的变化也会影响到Model。这个在用户互动中更加复杂，因为开发人员需要处理和解析这些互动，然后融合到一个model中，并且更新View。这是一个手动的复杂过程，当一个应用非常庞大的时候，将会是一件非常费劲的事情。

这里肯定有更好的解决方案！那就是AngularJS的双向数据绑定，能够同步DOM和Model等等。

这里有一个非常简单的例子，用来演示一个input输入框和<h1>元素的双向绑定(例01)：

<!doctype html>

<html ng-app="demoApp">

<head>

<script src="./js/angular.min.js"></script>

</head>

<body>

<div>

<label>Name:</label>

<input type="text" ng-model="user.name" placeholder="请输入名字">

<hr>

<h1>Hello, {{user.name}}!</h1>

</div>

</body>

</html>

说明：实际效果请大家看AngularJS/demo/index.html

## 3.2 特性二：模板

在AngularJS中，一个模板就是一个HTML文件。但是HTML的内容扩展了，包含了很多帮助你映射model到view的内容。

HTML模板将会被浏览器解析到DOM中。DOM然后成为AngularJS编译器的输入。AngularJS将会遍历DOM模板来生成一些指导，即，directive（指令）。所有的指令都负责针对view来设置数据绑定。

我们要理解AuguarJS并不把模板当做String来操作。输入AngularJS的是DOM而非string。数据绑定是DOM变化，不是字符串的连接或者innerHTML变化。使用DOM作为输入，而不是字符串，是AngularJS区别于其它的框架的最大原因。使用DOM允许你扩展指令词汇并且可以创建你自己的指令，甚至开发可重用的组件。

最大的好处是为设计师和开发者创建了一个紧密的工作流。设计师可以像往常一样开发标签，然后开发者拿过来添加上功能，通过数据绑定将会使得这个过程非常简单。

这里有一个例子，我们使用ng-repeat指令来循环图片数组并且加入img模板，如下：

function AlbumCtrl($scope) {

scope.images = [

{"image":"img/image\_01.png", "description":"Image 01 description"},

{"image":"img/image\_02.png", "description":"Image 02 description"},

{"image":"img/image\_03.png", "description":"Image 03 description"},

{"image":"img/image\_04.png", "description":"Image 04 description"},

{"image":"img/image\_05.png", "description":"Image 05 description"}

];

}

<div ng-controller="AlbumCtrl">

<ul>

<li ng-repeat="image in images">

<img ng-src="{{image.thumbnail}}" alt="{{image.description}}">

</li>

</ul>

<div ng-include="’/root/a.html’"></div>

</div>

这里还有一件事值得提一句，AngularJS并不强制你学习一个新的语法或者从你的应用中提出你的模板,注意ng-include写法双引号里面有单引号。

## 3.3 特性三：MVC

针对客户端应用开发AngularJS吸收了传统的MVC基本原则。MVC或者Model-View-Controll设计模式针对不同的人可能意味不同的东西。AngularJS并不执行传统意义上的MVC，更接近于MVVM（Moodel-View-ViewModel)。

Model

model是应用中的简单数据。一般是简单的javascript对象。这里没有必要继承框架的classes，使用proxy对象封装或者使用特别的setter/getter方法来访问。事实上我们处理vanilla javascript的方法就是一个非常好的特性，这种方法使得我们更少使用应用的原型。

ViewModel

viewmodel是一个用来提供特别数据和方法从而维护指定view的对象。

viewmodel是$scope的对象，只存在于AnguarJS的应用中。$scope只是一个简单的js对象，这个对象使用简单的API来侦测和广播状态变化。

Controller

controller负责设置初始状态和参数化$scope方法用以控制行为。需要指出的controller并不保存状态也不和远程服务互动。

View

view是AngularJS解析后渲染和绑定后生成的HTML 。这个部分帮助你创建web应用的架构。$scope拥有一个针对数据的参考，controller定义行为，view处理布局和互动。

## 3.4 特性四：服务和依赖注入

AngularJS服务其作用就是对外提供某个特定的功能。

AngularJS拥有内建的依赖注入（DI）子系统，可以帮助开发人员更容易的开发，理解和测试应用。

DI允许你请求你的依赖，而不是自己找寻它们。比如，我们需要一个东西，DI负责找创建并且提供给我们。

为了而得到核心的AngularJS服务，只需要添加一个简单服务作为参数，AngularJS会侦测并且提供给你：

function EditCtrl($scope, $location, $routeParams) {

// Something clever here...

}

你也可以定义自己的服务并且让它们注入：

angular.module('MyServiceModule', []).

factory('notify', ['$window', function (win) {

return function (msg) {

win.alert(msg);

};

}]);

function myController(scope, notifyService) {

scope.callNotify = function (msg) {

notifyService(msg);

};

}

myController.$inject = ['$scope', 'notify'];

## 3.5 特性五：指令（Directives）

指令是我个人最喜欢的特性。你是不是也希望浏览器可以做点儿有意思的事情？那么AngularJS可以做到。

指令可以用来创建自定义的标签。它们可以用来装饰元素或者操作DOM属性。可以作为标签、属性、注释和类名使用。

这里是一个例子，它监听一个事件并且针对的更新它的$scope ，如下：

myModule.directive('myComponent', function(mySharedService) {

return {

restrict: 'E',

controller: function($scope, $attrs, mySharedService) {

$scope.$on('handleBroadcast', function() {

$scope.message = 'Directive: ' + mySharedService.message;

});

},

replace: true,

template: '<input>'

};

});

然后，你可以使用这个自定义的directive来使用：

<my-component ng-model="message"></my-component>

使用一系列的组件来创建你自己的应用将会让你更方便的添加，删除和更新功能。

# 4 功能介绍

## 4.1数据绑定

AngularJS的双向数据绑定，意味着你可以在Mode(JS)中改变数据，而这些变动立刻就会自动出现在View上，反之亦然。即：一方面可以做到model变化驱动了DOM中元素变化，另一方面也可以做到DOM元素的变化也会影响到Model。

在我们使用jQuery的时候，代码中会大量充斥类似这样的语句：var val = $(‘#id’).val(); $(‘#id’).html(str);等等，即频繁的DOM操作（读取和写入），其实我们的最终目的并不是要操作DOM，而是要实现业务逻辑。ng的绑定将让你摆脱DOM操作，只要模板与数据通过声明进行了绑定，两者将随时保持同步，最新的数据会实时显示在页面中，页面中用户修改的数据也会实时被记录在数据模型中。

从View到Controller再到View的数据交互（例01）：

<html ng-app="demoApp">

……

<input type="text" ng-model="user.name" placeholder="请输入名称"/>

Hello, {{ user.name }}！

……

关键： ng-app 、 ng-model 和 { {user.name } }

首先： <html>元素的ng-app属性。标识这个DOM里面的内容将启用AngularJS应用。

其次：告诉AngularJS，对页面上的“user.name” 这个Model进行双向数据绑定。

第三：告诉AngularJS，在“{{ user.name}}”这个指令模版上显示“user.name”这个Model的数据。

从Server到Controller再到View的数据交互（例02）：

<html ng-app="demoApp">

……

<div ng-controller="demoController">

<input type="text" ng-model="user.name" disabled="disabled"/>

<a href="javascript:void(0);" ng-click="getAjaxUser()">AJAX获取名字</a>

……

demoApp.controller("demoController", function($http, $scope){

$scope. getAjaxUser = function(){

// $http.get({url:"../xxx.action"}).success(function(data){

// $scope.user= data;

// });

$scope.user = {"name":"从JOSN中获取的名称","age":22};

};

});

改变$scope中的user，View也会自动更新。

## 4.2 scopes、module、controller

### 4.2.1 scopes

$scope是一个把view（一个DOM元素）连结到controller上的对象。在我们的MVC结构里，这个 $scope 将成为model，它提供一个绑定到DOM元素（以及其子元素）上的excecution context。

尽管听起来有点复杂，但 $scope 实际上就是一个JavaScript对象，controller和view都可以访问它，所以我们可以利用它在两者间传递信息。在这个 $scope 对象里，我们既存储数据，又存储将要运行在view上的函数。

每一个Angular应用都会有一个 $rootScope。这个 $rootScope 是最顶级的scope，它对应着含有 ng-app 指令属性的那个DOM元素。

app.run(function($rootScope) { $rootScope.name = "张三"; });

如果页面上没有明确设定 $scope ，Angular 就会把数据和函数都绑定到这里， 第一部分中的例子就是靠这一点成功运行的。

这样，我们就可以在view的任何地方访问这个name属性，使用模版表达式{{}}，像这样：

{{ name }}

### 4.2.2 module

首先需要明确一下模板的概念。在我还不知道有模板这个东西的时候，曾经用js拼接出很长的HTML字符串，然后append到页面中，这种方式想想真是又土又笨。后来又看到可以把HTML代码包裹在一个<script>标签中当作模板，然后按需要取来使用。

在ng中，模板十分简单，它就是我们页面上的HTML代码，不需要附加任何额外的东西。在模板中可以使用各种指令来增强它的功能，这些指令可以让你把模板和数据巧妙的绑定起来。

在<html>标签上多了一个属性ng-app=”MyApp”，它的作用就是用来指定ng的作用域是在<html>标签以内部分。在js中，我们调用angular对象的module方法来声明一个模块，模块的名字和ng-app的值对应。这样声明一下就可以让ng运行起来了。

示例：

<html ng-app="demoApp">

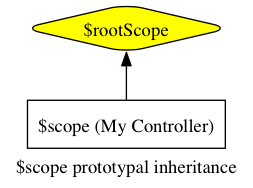
var demoApp = angular.module('demoApp', []);

### 4.2.3 ng-controller

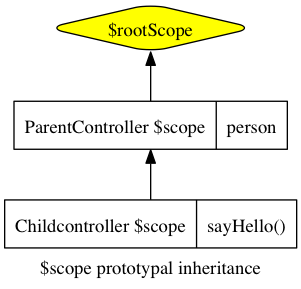
要明确创建一个$scope 对象，我们就要给DOM元素安上一个controller对象，使用的是ng-controller 指令属性：

<div ng-controller="MyController"> {{ person.name }} </div>

ng-controller指令给所在的DOM元素创建了一个新的$scope 对象，并将这个$scope 对象包含进外层DOM元素的$scope 对象里。在上面的例子里，这个外层DOM元素的$scope 对象，就是$rootScope 对象。这个scope链是这样的：



所有scope都遵循原型继承（prototypal inheritance），这意味着它们都能访问父scope们。对任何属性和方法，如果AngularJS在当前scope上找不到，就会到父 scope上去找，如果在父scope上也没找到，就会继续向上回溯，一直到$rootScope 上。即如果controller是多层嵌套的，就会从最里面一直往外找，这个scope链是这样的：



唯一的例外：有些指令属性可以选择性地创建一个独立的scope，让这个scope不继承它的父scope们，这个会在指令详解中说明。

## 4.3 ajax

$http 服务是AngularJS的核心服务之一，它帮助我们通过XMLHttpRequest对象或JSONP与远程HTTP服务进行交流。

$http 服务是这样一个函数：它接受一个设置对象，其中指定了如何创建HTTP请求；它将返回一个承诺（\*参考JavaScript异步编程的promise模式），其中提供两个方法： success方法和error方法。

demoApp.controller("demoController", function($http, $scope){

$scope. getAjaxUser = function(){

$http.get({url:"../xxx.action"}).success(function(data){

alert(data);

}).error(function(){

Alert(“出错了！”);

});

};

});

AngularJS的AJAX与jquery等框架的AJAX基本一致，这里就不多说了。还有一个angualr-resource的插件可以用来定制请求，注意先要引入angular-resource.js,这个要和angular的版本相互对应否则报错，记得用之前要注入该模块.它对常用的五种方法进行了封装'get','save','query','remove','delete','update'。

var User = $resource('/api/users/:path/:userId', {path:’@path’,userId:'@id'});

User.get({path:’info’,id:'123'}, successFn, errorFn);

该方法向url发送一个get请求，并期望一个json类型的响应。这里会向/api/users/info/?id=123发送一个请求，successFn处理请求成功响应，errorFn处理错误,如果上面那个@id改成@userId和前面一样那就是向/api/users/info/123发请求，不知道为什么反正angular-resource的1.3.2是这样的

## 4.4表达式

ng中的表达式与javascript表达式类似但是不可以划等号，它是ng自己定义的一套模式。表达式可以作为指令的值，如ng-modle=”people.name”、ng-click=”showMe()”，看起来是如此像字符串，故而也叫字符串表达式。也可以在标记中使用表达式，如{{1+2}}，或者与过滤器一起使用{{1+2 | currency}}。在框架内部，字符串不会简单的使用eval()来执行，而是有一个专门的$parse服务来处理。在ng表达式中不可以使用循环语句、判断语句，事实上在模板中使用复杂的表达式也是一个不推荐的做法，这样视图与逻辑就混杂在一起了

我们在使用其他模板库时，一般都会有模板的循环输出、分支输出、逻辑判断等类似的控制。

要想理解指令属性的运作，我们必须先理解表达式。在之前的例子里我们已经见过表达式，例如 {{ user.name }}。

请查看例03、例04、例05。

{{ 8 + 1 }} 9

{{ person }} {"name":"Ari Lerner"}

{{ 10 \* 3.3 | currency }} $33.00

表达式粗略来看有点像 eval(javascript) 的结果。它们会经过Angular.js的处理，从而拥有以下重要而独特的性质：

* 所有表达式都在scope这个context里被执行，因此可以使用所有本地 $scope 中的变量。
* 如果一个表达式的执行导致类型错误或引用错误，这些错误将不会被抛出。
* 表达式里不允许任何控制函数流程的功能（如if/else等条件语句）
* 表达式可接受一个或多个串联起来的过滤器。

## 4.5过滤器

过滤器（filter）正如其名，作用就是接收一个输入，通过某个规则进行处理，然后返回处理后的结果。主要用在数据的格式化上，例如获取一个数组中的子集，对数组中的元素进行排序等。过滤器通常是伴随标记来使用的，将你model中的数据格式化为需要的格式。表单的控制功能主要涉及到数据验证以及表单控件的增强。ng内置了一些过滤器，它们是：

currency(货币)、date(日期)、filter(子串匹配)、json(格式化json对象)、limitTo(限制个数)、lowercase(小写)、uppercase(大写)、number(数字)、orderBy(排序)。

### 4.5.1过滤器使用方式

总共九种。除此之外还可以自定义过滤器，这个就强大了，可以满足任何要求的数据处理。Filter还是很简单的，需要明白的是内置的filter如何使用，以及自己如何定义一个filter。

filter的两种使用方法：

　　1. 在模板中使用filter

　　我们可以直接在{{}}中使用filter，跟在表达式后面用 | 分割，语法如下：

{{ expression | filter }}

也可以多个filter连用，上一个filter的输出将作为下一个filter的输入：

{{ expression | filter1 | filter2 | ... }}

filter可以接收参数，参数用 : 进行分割，如下：

{{ expression | filter:argument1:argument2:... }}

除了对{{}}中的数据进行格式化，我们还可以在指令中使用filter，例如先对数组array进行过滤处理，然后再循环输出：

<span ng-repeat="a in array | filter ">

2. 在controller和service中使用filter

　　我们的js代码中也可以使用过滤器，方式就是我们熟悉的依赖注入，例如我要在controller中使用currency过滤器，只需将它注入到该controller中即可，代码如下：

app.controller('testC',function($scope,currencyFilter){

$scope.num = currencyFilter(123534);

}

在模板中使用{{num}}就可以直接输出$123,534.00了！在服务中使用filter也是同样的道理。

　　如果你要在controller中使用多个filter，并不需要一个一个注入吗，ng提供了一个$filter服务可以来调用所需的filter，你只需注入一个$filter就够了，使用方法如下：

app.controller('testC',function($scope,$filter){

$scope.num = $filter('currency')(123534);

$scope.date = $filter('date')(new Date());

}

可以达到同样的效果。好处是你可以方便使用不同的filter了。

### 4.5.2 ng的内置过滤器

ng内置了九种过滤器，使用方法都非常简单，看文档即懂。不过为了以后不去翻它的文档，我在这里还是做一个详细的记录。

currency(货币)、date(日期)、filter(子串匹配)、json(格式化json对象)、limitTo(限制个数)、lowercase(小写)、uppercase(大写)、number(数字)、orderBy(排序)

#### 1. currency (货币处理)

　　使用currency可以将数字格式化为货币，默认是美元符号，你可以自己传入所需的符号，例如我传入人民币：

{{num | currency : '￥'}}

#### 2. date (日期格式化)

　　原生的js对日期的格式化能力有限，ng提供的date过滤器基本可以满足一般的格式化要求。用法如下：

{{date | date : 'yyyy-MM-dd hh:mm:ss EEEE'}}

参数用来指定所要的格式，y M d h m s E 分别表示 年 月 日 时 分 秒 星期，你可以自由组合它们。也可以使用不同的个数来限制格式化的位数。另外参数也可以使用特定的描述性字符串，例如“shortTime”将会把时间格式为12:05 pm这样的。ng提供了八种描述性的字符串，个人觉得这些有点多余，我完全可以根据自己的意愿组合出想要的格式，不愿意去记这么多单词~

#### 3. filter(匹配子串)

　　这个名叫filter的filter。用来处理一个数组，然后可以过滤出含有某个子串的元素，作为一个子数组来返回。可以是字符串数组，也可以是对象数组。如果是对象数组，可以匹配属性的值。它接收一个参数，用来定义子串的匹配规则。下面举个例子说明一下参数的用法，我用现在特别火的几个孩子定义了一个数组：

$scope.childrenArray = [

{name:'kimi',age:3},

{name:'cindy',age:4},

{name:'anglar',age:4},

{name:'shitou',age:6},

{name:'tiantian',age:5}

];

$scope.func = function(e){return e.age>4;}{{ childrenArray | filter : 'a' }} //匹配属性值中含有a的

{{ childrenArray | filter : 4 }} //匹配属性值中含有4的

{{ childrenArray | filter : {name : 'i'} }} //参数是对象，匹配name属性中含有i的

{{childrenArray | filter : func }} //参数是函数，指定返回age>4的

#### 4. json(格式化json对象)

　　json过滤器可以把一个js对象格式化为json字符串，没有参数。这东西有什么用呢，我一般也不会在页面上输出一个json串啊，官网说它可以用来进行调试，嗯，是个不错的选择。或者，也可以用在js中使用，作用就和我们熟悉的JSON.stringify()一样。用法超级简单：

{{ jsonTest | json}}

#### 5. limitTo(限制数组长度或字符串长度)

　　limitTo过滤器用来截取数组或字符串，接收一个参数用来指定截取的长度，如果参数是负值，则从数组尾部开始截取。个人觉得这个filter有点鸡肋，首先只能从数组或字符串的开头/尾部进行截取，其次，js原生的函数就可以代替它了，看看怎么用吧：

{{ childrenArray | limitTo : 2 }} //将会显示数组中的前两项

#### 6. lowercase(小写)

　　把数据转化为全部小写。太简单了，不多解释。同样是很鸡肋的一个filter，没有参数，只能把整个字符串变为小写，不能指定字母。怎么用我都懒得写了。

#### 7. uppercase(大写)

　　同上。

#### 8. number(格式化数字)

　　number过滤器可以为一个数字加上千位分割，像这样，123,456,789。同时接收一个参数，可以指定float类型保留几位小数：

{{ num | number : 2 }}

#### 9. orderBy(排序)

　　orderBy过滤器可以将一个数组中的元素进行排序，接收一个参数来指定排序规则，参数可以是一个字符串，表示以该属性名称进行排序。可以是一个函数，定义排序属性。还可以是一个数组，表示依次按数组中的属性值进行排序（若按第一项比较的值相等，再按第二项比较），还是拿上面的孩子数组举例：

<div>{{ childrenArray | orderBy : 'age' }}</div> //按age属性值进行排序，若是-age，则倒序

<div>{{ childrenArray | orderBy : orderFunc }}</div> //按照函数的返回值进行排序

<div>{{ childrenArray | orderBy : ['age','name'] }}</div> //如果age相同，按照name进行排序　　内置的过滤器介绍完了，写的我都快睡着了。。。正如你所看到的，ng内置的过滤器也并不是万能的，事实上好多都比较鸡肋。更个性化的需求就需要我们来定义自己的过滤器了，下面来看看如何自定义过滤器。

### 4.5.3自定义过滤器及示例

　　filter的自定义方式也很简单，使用module的filter方法，返回一个函数，该函数接收输入值，并返回处理后的结果。话不多说，我们来写一个看看。比如我需要一个过滤器，它可以返回一个数组中下标为奇数的元素，代码如下：

app.filter('odditems',function(){

return function(inputArray){

var array = [];

for(var i=0;i<inputArray.length;i++){

if(i%2!==0){

array.push(inputArray[i]);

}

}

return array;

}

});

格式就是这样，你的处理逻辑就写在内部的那个闭包函数中。你也可以让自己的过滤器接收参数，参数就定义在return的那个函数中，作为第二个参数，或者更多个参数也可以。

自定义过滤器实例（例04）：

/\* View html \*/

First name：<input ng-model="user.firstName"/><br/>

Last name：<input ng-model="user.lastName"/> <br/>

First name：{{user.firstName}} &nbsp;&nbsp; &nbsp;&nbsp;Last name：{{user.lastName}} <br/>

Fullname：{{user | flFullname}}<br/>

Fullname：{{user | flFullname:"－"}}<br/>

Fullname：{{user | flFullname:"•" | uppercase }}

/\* Controller js \*/

demoApp.filter("flFullname", function() {

return function(user, sep) {

sep = sep || " ";

user = user || {};

fullName = "";

if(user.firstName){fullName += user.firstName;}

if(user.lastName){fullName = fullName + sep + user.lastName;}

if(fullName && fullName.length>0){return fullName;

}else{return "";}

};

});

## 4.6指令(directive)

　　通过使用模板，我们可以把model和controller中的数据组装起来呈现给浏览器，还可以通过数据绑定，实时更新视图，让我们的页面变成动态的。

　 模板中可以使用的东西包括以下四种：

1.指令(directive)：ng提供的或者自定义的标签和属性，用来增强HTML表现力；

2.标记(markup)：即双大括号{{}}，可将数据单向绑定到HTML中；

3.过滤器(filter)：用来格式化输出数据；

4.表单控制：用来增强表单的验证功能。

其中，指令无疑是使用量最大的，ng内置了很多指令用来控制模板，如ng-repeat，ng-class，也有很多指令来帮你完成业务逻辑，如ng-controller,ng-model。

指令的几种使用方式如下：

* 作为标签：<my-dir></my-dir>
* 作为属性：<span my-dir="exp"></span>
* 作为注释：<!-- directive: my-dir exp -->
* 作为类名：<span class="my-dir: exp;"></span>

其实常用的就是作为标签和属性。

### 4.6.1样式相关的指令

　　既然模板就是普通的HTML，那我首要关心的就是样式的控制，元素的定位、字体、背景色等等如何可以灵活控制。下面来看看常用的样式控制指令。

1. ng-class

　　 ng-class用来给元素绑定类名，其表达式的返回值可以是以下三种：

* 类名字符串，可以用空格分割多个类名，如’redtext boldtext’；
* 类名数组，数组中的每一项都会层叠起来生效；
* 一个名值对应的map，其键值为类名，值为boolean类型，当值为true时，该类会被加在元素上。

　　下面来看一个使用map的例子：

ng-class测试

红色 加粗 删除线

map:{redtext:{{red}}, boldtext:{{bold}}, striketext:{{strike}}}

　　如果你想拼接一个类名出来，可以使用插值表达式，如：

　　<div class=”{{style}}text”>字体样式测试</div>

　　然后在controller中指定style的值：

　　$scope.style = ‘red’;

　　注意我用了class而不是ng-class，这是不可以对换的，官方的文档也未做说明，姑且认为这是ng的语法规则吧。

　　与ng-class相近的，ng还提供了ng-class-odd、ng-class-even两个指令，用来配合ng-repeat分别在奇数列和偶数列使用对应的类。这个用来在表格中实现隔行换色再方便不过了。

2. ng-style

　　ng-style用来绑定元素的css样式，其表达式的返回值为一个js对象，键为css样式名，值为该样式对应的合法取值。用法比较简单：

<div ng-style="{color:'red'}">ng-style测试</div>

<div ng-style="style">ng-style测试</div>

$scope.style = {color:'red'};

3. ng-show，ng-hide

　　　对于比较常用的元素显隐控制，ng也做了封装，ng-show和ng-hide的值为boolean类型的表达式，当值为true时，对应的show或hide生效。框架会用display:block和display:none来控制元素的显隐。

### 4.6.2表单控件功能相关指令

　　对于常用的表单控件功能，ng也做了封装，方便灵活控制。

　　ng-checked控制radio和checkbox的选中状态

　　ng-selected控制下拉框的选中状态

　　ng-disabled控制失效状态

　　ng-multiple控制多选

　　ng-readonly控制只读状态

　　以上指令的取值均为boolean类型，当值为true时相关状态生效，道理比较简单就不多做解释。注意： 上面的这些只是单向绑定，即只是从数据到模板，不能反作用于数据。要双向绑定，还是要使用 ng-model 。

### 4.6.3事件绑定相关指令

事件绑定是javascrpt中比较重要的一部分内容，ng对此也做了详细的封装，正如我们之前使用过的ng-click一样，事件的指令如下：

ng-click

　　ng-change

　　ng-dblclick

　　ng-mousedown

　　ng-mouseenter

　　ng-mouseleave

　　ng-mousemove

　　ng-mouseover

　　ng-mouseup

　　ng-submit

　　事件绑定指令的取值为函数，并且需要加上括号，例如：

<select ng-change=”change($event)”></select>

然后在controller中定义如下：

$scope.change = function($event){

alert($event.target);

//……………………

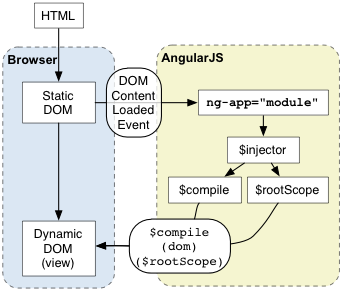
}

在模板中可以用变量$event将事件对象传递到controller中。

对于ng的这种设计，一些人有所质疑，视图与事件绑定混在一起到底好不好？我们不是要讲究视图与逻辑分离吗？如此一来，把事件的绑定又变回了内联的，岂不是历史的倒退。我也一样对此表示不解，因为不写onclick已经很多年。。。但既然已经存在了，我们不妨往合理的方向上想一想，或许ng的设计者压根就不想让模板成为单纯的视图层，本来就是想增强HTML，让它有一点业务能力。这么想的话似乎也能想通，好吧，先欺骗一下自己吧~

### 4.6.4特殊的ng-src和ng-href

在说明这两个指令的特殊之前，需要先了解一下ng的启动及执行过程，如下图：



1) 浏览器加载静态HTML文件并解析为DOM；

　　2) 浏览器加载angular.js文件；

　　3) angular监听DOMContentLoaded 事件，监听到时开始启动；

　　4) angular寻找ng-app指令，确定作用范围；

　　5) 找到app中定义的Module使用$injector服务进行依赖注入；

　　6) 根据$injector服务创建$compile服务用于编译；

　　7) $compile服务编译DOM中的指令、过滤器等；

　　8) 使用ng-init指令，将作用域中的变量进行替换；

　　9) 最后生成了我们在最终视图。

　　可以看到，ng框架是在DOMcontent加载完毕后才开始发挥作用。假如我们模板中有一张图片如下：

　　<img src=”{{imgUrl}}” />

　　那么在页面开始加载到ng编译完成之前，页面上会一直显示一张错误的图片，因为路径{{imgUrl}}还未被替换。

　　为了避免这种情况，我们使用ng-src指令，这样在路径被正确得到之前就不会显示找不到图片。同理，<a>标签的href属性也需要换成ng-href，这样页面上就不会先出现一个地址错误的链接。

顺着这个思路再多想一点，我们在模板中使用{{}}显示数据时，在ng编译完成之前页面上岂不是会显示出大括号及里面的表达式？确实是这样。为了避免这个，ng中有一个与{{}}等同的指令:ng-bind，同样用于单向绑定，在页面刚加载的时候就不会显示出对用户无用的数据了。尽管这样你可能不但没舒心反而更纠结了，{{}}那么好用易理解，还不能用了不成？好消息是我们依然可以使用。因为我编写的是单页面应用，页面只会在加载index.html的时候出这个问题，只需在index.html中的模板中换成ng-bind就行。其他的模板是我们动态加载的，就可以放心使用{{}}了。

### 4.6.5 自定义指令示例

下面我们来解析下指令的例子（例07）。

1.首先，我们定义一个名为userInfo的指令：

demoApp.directive('userInfo',function(){

return {

restrict : 'E',

templateUrl : 'userInfoTemplate.html',

replace : true,

transclude : true,

scope : {

mytitle : '=etitle'

},

link : function(scope,element,attris){

scope.showText = false;

scope.toggleText = function(){

scope.showText = ! scope.showText;

}

}

};

})

Restrict为'E'：用作标签；replace为true：用模板替换当前标签；transclude为true：将当前元素的内容转移到模板中；scope 为 {mytitle : '=etitle'}：定义一个名为mytitle的MODEL，其值指向当前元素的etitle属性；templateUrl为'userInfoTemplate.html'：模板内容为ng-template定义ID为userInfoTemplate.html的内容；link：指定所包含的行为。其具体的说明及其他参数，请参考：6.2指令详解。

2. userInfoTemplate.html模板为：

<script type="text/ng-template" id="userInfoTemplate.html">

<div class="mybox">

<div class="mytitle" style="cursor: pointer;" ng-click="toggleText()">

{ {mytitle} }

</div>

<div ng-transclude ng-show="showText">

</div>

</div>

</script>

将当前元素的内容添加到有ng-transclude属性的这个DIV下，默认是隐藏的。

3.Controller信息：

demoApp.controller("test7Controller", function($scope){

$scope.title = '个人简介';

$scope.text = '大家好，我正在研究AngularJs，欢迎大家与我交流。';

$scope.updateInfo = function (){

$scope.title = '个人信息';

$scope.text = '大家好，今天天气真好！';

}

});

4.指令使用方式（View信息）为：

<user-info etitle="title">{ {text} }</user-info>

Etitle指向Controller中的$scope.title。注意命名方式：指令名为userInfo，对应的标签为user-info。

## 4.7服务（service）

### 4.7.1服务介绍

　　服务这个概念其实并不陌生，在其他语言中如java便有这样的概念，其作用就是对外提供某个特定的功能，如消息服务，文件压缩服务等，是一个独立的模块。ng的服务是这样定义的：

Angular services are singletons objects or functions that carry out specific tasks common to web apps.

它是一个单例对象或函数，对外提供特定的功能。

首先是一个单例，即无论这个服务被注入到任何地方，对象始终只有一个实例。

其次这与我们自己定义一个function然后在其他地方调用不同，因为服务被定义在一个模块中，所以其使用范围是可以被我们管理的。ng的避免全局变量污染意识非常强。

　　ng提供了很多内置的服务，可以到API中查看http://docs.angularjs.org/api/。知道了概念，我们来拉一个service出来溜溜，看看到底是个什么用法。

　　我们在controller中直接声明$location服务，这依靠ng的依赖注入机制。$location提供地址栏相关的服务，我们在此只是简单的获取当前的地址。

　　服务的使用是如此简单，我们可以把服务注入到controller、指令或者是其他服务中。

### 4.7.2自定义服务

　　如同指令一样，系统内置的服务以$开头，我们也可以自己定义一个服务。定义服务的方式有如下几种：

* 使用系统内置的$provide服务；
* 使用Module的factory方法；
* 使用Module的service方法。

　　下面通过一个小例子来分别试验一下。我们定义一个名为remoteData服务，它可以从远程获取数据，这也是我们在程序中经常使用的功能。不过我这里没有远程服务器，就写死一点数据模拟一下。

//使用$provide来定义

var app = angular.module('MyApp', [], function($provide) {

$provide.factory('remoteData', function() {

var data = {name:'n',value:'v'};

return data;

});

});

//使用factory方法

app.factory('remoteData',function(){

var data = {name:'n',value:'v'};

return data;

});

//使用service方法

app.service('remoteData',function(){

this.name = 'n';

this.value = 'v';

});

Module的factory和$provide的factory方法是一模一样的，从官网文档看它们其实就是一回事。至于Module内部是如何调用的，我此处并不打算深究，我只要知道怎么用就好了。

再看Module的service方法，它没有return任何东西，是因为service方法本身返回一个构造器，系统会自动使用new关键字来创建出一个对象。所以我们看到在构造器函数内可以使用this，这样调用该服务的地方便可以直接通过remoteData.name来访问数据了。

### 4.7.3管理服务的依赖关系

　　服务与服务中间可以有依赖关系，例如我们这里定义一个名为validate的服务，它的作用是验证数据是否合法，它需要依赖我们从远程获取数据的服务remoteData。代码如下：

　 在factory的参数中，我们可以直接传入服务remoteData，ng的依赖注入机制便帮我们做好了其他工作。不过一定要保证这个参数的名称与服务名称一致，ng是根据名称来识别的。若参数的名次与服务名称不一致，你就必须显示的声明一下，方式如下：

app.factory('validate',['remoteData',function(remoteDataService){

return function(){

if(remoteDataService.name=='n'){

alert('验证通过');

}

};

}]);

我们在controller中注入服务也是同样的道理，使用的名称需要与服务名称一致才可以正确注入。否则，你必须使用$inject来手动指定注入的服务。比如：

function testC(scope,rd){

scope.getData = function(){

alert('name：'+rd.name+' value：'+rd.value);

}

}

testC.$inject = ['$scope','remoteData'];

　　在controller中注入服务，也可以在定义controller时使用数组作为第二个参数，在此处把服务注入进去，这样在函数体中使用不一致的服务名称也是可以的，不过要确保注入的顺序是一致的，如：

app.controller('testC',['$scope','remoteData',function($scope,rd){

$scope.getData = function(){

alert('name：'+rd.name+' value：'+rd.value);

}

}]);

### 4.7.4 自定义服务示例

接下来让我们看下例子（例08 自定义服务）代码，自定义userService服务：

demoApp.factory('userService', ['$http', function($http) {

var doGetUser = function(userId, path) {

//return $http({

//method: 'JSONP',

//url: path

//});

/\*手动指定数据\*/

var data = {userId:"woshishui",userName:"我是谁",userInfo:"我是谁！我是谁！"};;

if(userId=='zhangsan'){

data = {userId:"zhangsan",userName:"张三",userInfo:"我是张三，我为自己"};

}else if(userId=='lisi'){

data = {userId:"lisi",userName:"李四",userInfo:"我是李四，我为卿狂！"};

}

return data;

}

return {

/\*userService对外暴露的函数，可有多个\*/

getUser: function(userId) {

return doGetUser(userId, '../xxx/xxx.action');

}

};

}]);

我们创建了一个只有一个方法的userService，getUser为这个服务从后台获取用户信息的函数，并且对外暴露。当然，由于这是一个静态的例子，无法访问后台，那么我们便制定其返回的数据。

然后我们把这个服务添加到我们的controller中。我们建立一个controller并加载（或者注入）userService作为运行时依赖，我们把service的名字作为参数传递给controller 函数：

demoApp.controller("test8Controller", function($scope,userService){

/\*文章信息\*/

$scope.articles = [{

title : "爱飞像风",

userId : "zhangsan",

userName : "张三"

},{

title : "无法停止的雨",

userId : "lisi",

userName : "李四"

}];

$scope.showUserInfo = false;//显示作者详细信息开关

$scope.currentUser = {}; //当前选中的作者

$scope.getUserInfo = function(userId){

$scope.currentUser = userService.getUser(userId);

//调用 userService的getUser函数

$scope.showUserInfo = true;

setTimeout(function(){//定时器：隐藏作者详细信息

$scope.showUserInfo = false;

},3000);

}

});

我们的userService注入到我们的test8Controller后，我们就可以像使用其他服务（我们前面提到的$http服务）一样的使用userService了。

相关的HTML代码如下：

/\* View HTML\*/

<tr ng-repeat="article\_ in articles">

<td>

{{article\_.title}}

</td>

<td>

<a href="javascript:void(0);" ng-click="getUserInfo(article\_.userId)"> {{article\_.userName}} </a>

</td>

</tr>

......

<div ng-show="showUserInfo">

用户ID：{{currentUser.userId}}<br/>

用户名：{{currentUser.userName}}<br/>

用户简介：{{currentUser.userInfo}}<br/>

</div>

## 4.8依赖注入DI

通过依赖注入，ng想要推崇一种声明式的开发方式，即当我们需要使用某一模块或服务时，不需要关心此模块内部如何实现，只需声明一下就可以使用了。在多处使用只需进行多次声明，大大提高可复用性。

　　比如我们的controller，在定义的时候用到一个$scope参数。

app.controller('testC',function($scope){});

如果我们在此处还需操作其他的东西，比如与浏览器地址栏进行交互。我们只需再多添一个参数$location进去：

app.controller('testC',function($scope,$location){});

这样便可以通过$location来与地址栏进行交互了，我们仅仅是声明了一下，所需的其他代码，框架已经帮我们注入了。我们很明显的感觉到了这个函数已经不是常规意义上的javascript函数了，在常规的函数中，把形参换一个名字照样可以运行，但在此处若是把$scope换成别的名字，程序便不能运行了。因为这是已经定义好的服务名称。

这便是依赖注入机制。顺理成章的推断，我们可以自己定义模块和服务，然后在需要的地方进行声明，由框架来替我们注入。

来看下我们如何定义一个服务：

app.factory('tpls',function(){

return ['tpl1','tpl2','tpl3','tpl4'];

});

看上去相当简单，是因为我在这里仅仅是直接返回一个数组。在实际应用中，这里应该是需要向服务器发起一个请求，来获取到这些模板们。服务的定义方式有好几种，包括使用provider方法、使用factory方法，使用service方法。它们之间的区别暂且不关心。我们现在只要能创建一个服务出来就可以了。我使用了factory方法。一个需要注意的地方是，框架提供的服务名字都是由$开头的，所以我们自己定义的最好不要用$开头，防止发生命名冲突。

定义好一个服务后，我们就可以在控制器中声明使用了，如下：

app.controller('testC',function($scope,tpls){

$scope.question = questionModel;

$scope.nowTime = new Date().valueOf();

$scope.templates = tpls; //赋值到$scope中

$scope.addOption = function(){

var o = {content:''};

$scope.question.options.push(o);

};

$scope.delOption = function(index){

$scope.question.options.splice(index,1);

};

});

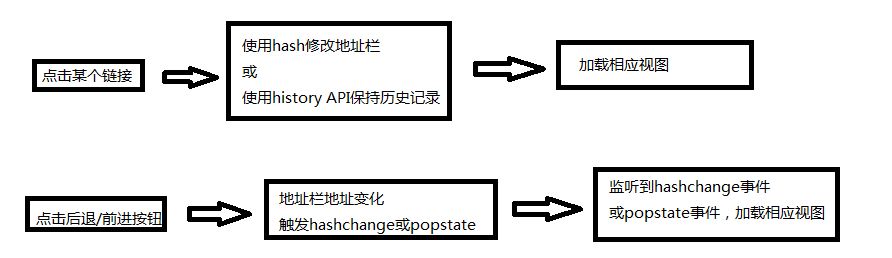
此时，若在模板中书写如下代码，我们便可以获取到服务tpls所提供的数据了：

模板:

<a href="javascript:void(0);" ng-repeat="t in templates">{{t}}&nbsp;&nbsp;</a><br />

## 4.9路由（route）

在谈路由机制前有必要先提一下现在比较流行的单页面应用，就是所谓的single page APP。为了实现无刷新的视图切换，我们通常会用ajax请求从后台取数据，然后套上HTML模板渲染在页面上，然而ajax的一个致命缺点就是导致浏览器后退按钮失效，尽管我们可以在页面上放一个大大的返回按钮，让用户点击返回来导航，但总是无法避免用户习惯性的点后退。解决此问题的一个方法是使用hash，监听hashchange事件来进行视图切换，另一个方法是用HTML5的history API，通过pushState()记录操作历史，监听popstate事件来进行视图切换，也有人把这叫pjax技术。基本流程如下：



如此一来，便形成了通过地址栏进行导航的深度链接（deeplinking ），也就是我们所需要的路由机制。通过路由机制，一个单页应用的各个视图就可以很好的组织起来了。

### 4.9.1 Route内容

ng的路由机制是靠ngRoute提供的，通过hash和history两种方式实现了路由，可以检测浏览器是否支持history来灵活调用相应的方式。ng的路由(ngRoute)是一个单独的模块，你可以引入angular-route.min.js这个文件来路由，但是现在使用基于状态的angular-ui-route.min.js的路由会更加好用这里只介绍ui-router

**路由所会触发的事件的介绍：**

***$stateChangeError***

路由状态变化错误时事件。参数有：event，toState，toParams，fromState，fromParams，error

***$stateChangeStart***

路由状态变化发生前触发的事件。参数有：event，toState，toParams，fromState，fromParams

***$stateChangeSuccess***

路由状态变化正确时触发的事件。参数有：event，toState，toParams，fromState，fromParams

***$stateNotFound***

路由状态没找到的时候触发的事件。参数有：event，unfoundState，fromState，fromParams

和配置块不同，运行块在注入器创建**之后**被执行，它是所有AngularJS应用中第一个被执行的方法,运行块通常用来注册全局的事件监听器。例如，我们会在.run()块中设置路由事件的监听器以及过滤未经授权的请求

Angular.module(‘myapp’,[‘ui.router’])

.run(function($rootScope,AuthService){

$rootScope.$on(‘$stateChangeStart’,function(event，toState，toParams){

If(AuthService.userloggin()){

Console.log(“has loggin”)

}

})

})

### 4.9.2 ui-router的路由参数介绍

**$urlRouterProvider**

**deterIntercept(defer) 这个方法来保持url的同步，默认defer是true；**

**otherwise(rule) 定义当一个请求的路径是无效路径是跳转的路径，rule表示**你想重定向的url路径或一个返回的网址路径的规则函数。函数传入两个参数：$injector和$location服务，而且必须返回一个string的url；

when(what,handler) 其中第一个参数what表示匹配到的url，第二个参数为相应的处理函数；

**$urlRouter**

href(urlMacther,params,options)一个生成URL的方法，为给定的UrlMatcher返回编译后的URL，并且用提供的参数填充；

sync() 该方法可以触发更新，发生在地址栏URL变化是执行相同更新；

**$state**

get(stateOrName,context) 返回任何指定的状态或所有状态的配置对象其中stateOrName：如果提供此参数，返回的是指定状态的配置对象；如果不提供此参数，则返回全部状态的配置对象；context：当context是一个相对的参考状态，状态会在相关上下文中检索；

go(to,params,options) 其中 to：string，即将跳转的状态；params:object，跳转所带的参数；  
options：object，可选配置对象。有 location(是否更新地址栏的url，或以什么字符串替换url)，inherit(是否继承当前url的参数)，relative(当变化相对路径：如"^,定义的状态是相对的)，notify(是否广播$stateChangeStart和$stateChangeSuccess事件)，reload(是否重新载入)；

href(stateOeName,params,options) 一个URL生成方法，返回为给定的状态填充指定的参数编译后的链接；stateOeName：string，你想要生成的url的状态或者状态对象；params：object，一个用于填充状态需要的参数的对象；options：可选配置对象。有lossy(当第一个参数url未被提供时是否继承导航的url进行构建href),inherit(是否继承当前url的参数),relative(当变化相对路径：如"^,定义的状态是相对的),absolute(是否生成绝对url)；

include(stateOeName,params,options)一个确定当前有效的状态是不是与stateOrName平级的还是其子状态；stateOeName：string，部分名称，相对名称，或者通过当前状态进行全局模式查找；params：object，一个参数对象；options：可选配置对象。有relative；

is(stateOeName,params,options)与$state.include相似，只是这个针对的是全名。参数性质同上

reload(state) 重新载入当前状态的方法;；state：一个状态名称或者状态对象

transitionTo(to,toParams,options) 过度到一个新状态的方法；to：状态名称；toParams：将会发送到下一个状态的参数；options：可选参数。有location，inherit，relative，notify，reload；

**$stateProvider**

decorator(name,func)通过内部的$stateProvider以扩展或者重写状态生成器。可用于添加ui-router的自定义功能，例如，基于状态名称推断templateUrl。警告：因为生成器的函数执行顺序的不确定，decorator不应该相互依赖

state(name,stateConfig)所有配置如下：

name：状态的名称。  
stateConfig：状态配置对象。配置具有以下各项属性：  
template： string/function，html模板字符串，或者一个返回html模板字符串的函数。  
templateUrl：string/function，模板路径的字符串，或者返回模板路径字符串的函数。  
templateProvider：function，返回html模板字符串或模板路径的服务。  
controller：string/function，新注册一个控制器函数或者一个已注册的控制器的名称字符串，注意得用数组形式写法[‘$state’,function($state){}]，还有就是如果模板没有实例化的话controller是不运行的，也就是template必须放到页面相应的ui-view里面才行  
controllerProvider：function，返回控制器或者控制器名称的服务  
controllerAs：string，控制器别名。  
parent：string/object，手动指定该状态的父级。  
resolve：object，将会被注入controller去执行的函数，<string,function>形式。  
url：string，当前状态的对应url。  
views：object，视图展示的配置。<string,object>形式。  
abstract：boolean，一个永远不会被激活的抽象的状态，但可以给其子级提供特性的继承

onEnter：function，当进入一个状态后的回调函数。  
onExit：function，当退出一个状态后的回调函数。  
reloadOnSearch：boolean，如果为false，那么当一个search/query参数改变时不会触发相同的状态，用于当你修改$location.search()的时候不想重新加载页面。默认为true。  
data：object，任意对象数据，用于自定义配置。继承父级状态的data属性。换句话说，通过原型继承可以达到添加一个data数据从而整个树结构都能获取到。  
params：url里的参数值，通过它可以实现页面间的参数传递

**$stateParams获取路由传参数的对象**

**ui-sref**  
一种将链接（<a>标签）绑定一个状态的指令。点击该链接将触发一个可以带有可选参数的状态转换。  
<a ui-sref="app.index">首页</a>

<!-- 这里是正常的跳转 -->

<a ui-sref="app.index({id:yourId})">你的主页</a>

<!-- 这里是带参数对象的跳转，名称是id，值是yourId -->

### 4.9.3 路由示例

$stateProvider

//必须先载入这个总的contacts路由否则下面子路由无用，而且一级目录时候状态名称必须和内部url相同

//任何其子目录的载入都会触发以及目录加载一次，之后的就不会触发了

.state('contacts',{

abstract:true,

//上面参数的效果是直接走contacts时候就会无法匹配而是走例如如下

//$urlRouterProvider.otherwise('contacts/list')也就重新再匹配到下面的路由了

url: '/contacts',

templateUrl: 'contacts.list.html'

})

.state('contacts.list',{

url: '/list',

templateUrl: 'contacts.list.html'

})

.state('contacts.detail', {

url: '/{contactId:[0-9]{1,4}}',

// view 用在该状态下有多个 ui-view 的情况，可以对不同的 ui-view 使用特定的 template,controller, resolve data

// 绝对 view 使用 '@' 符号来区别，比如 'foo@bar' 表明名为 'foo' 的 ui-view 使用了 'bar' 状态的模板(template)，相对 view 则无

views: {

// 无名 view

'': {

templateUrl: 'contacts.detail.html',

controller: ['$scope', '$stateParams', 'utils', function ($scope, $stateParams, utils) {

$scope.contact = utils.findById($scope.contacts, $stateParams.contactId);

}]

},

// for "ui-view='hint'"

'hint@': {

template: 'This is contacts.defail populating the "hint" ui-view'

},

// for "ui-view='menuTip'"

'menuTip': {

templateProvider: ['$stateParams', function($stateParams){

return '<hr><small class="muted">Contact ID: ' + $stateParams.contactId + '</small>'; } ]

}

},

//resolve里面的键所对应的函数或者数组会先运行进行预载入，这里面可以进行数据拉取文件加载什么的，会保证在controll里面可以访问到

//在resolve里面是无法完成跳转的，因为此时模板没有实例化

resolve:{

user: function() {

return {

name: "peter",

email: "audiogroup@qq.com"

}

},

},

//注意每个controller必须有一个对应的模板url使其实例化否则不运行controller

controller:function(user,detail,myId$scope){

alert(user.name)

}

})

<h2>All Contacts</h2>

<ul>

<li ng-repeat="contact in contacts">

<a ui-sref="contacts.detail({contactId:contact.id})">{{contact.name}}</a>

</li>

</ul>

## 4.10 NG动画效果

### 4.10.1 NG动画效果简介

NG动画效果，现在可以通过CSS3或者是JS来实现，如果是通过JS来实现的话，需要其他JS库（比如JQuery）来支持，实际上底层实现还是靠其他JS库，只是NG将其封装了，使其更易使用。

NG动画效果包含以下几种：

* enter：元素添加到DOM中时执行动画；
* leave：元素从DOM删除时执行动画；
* move：移动元素时执行动画；
* beforeAddClass：在给元素添加CLASS之前执行动画；
* addClass：在给元素添加CLASS时执行动画；
* beforeRemoveClass：在给元素删除CLASS之前执行动画；
* removeClass：在给元素删除CLASS时执行动画。

其相关参数为：

var ngModule = angular.module('YourApp', ['ngAnimate']);

demoApp.animation('.my-crazy-animation', function() {

return {

enter: function(element, done) {

//run the animation here and call done when the animation is complete

return function(cancelled) {

//this (optional) function will be called when the animation

//completes or when the animation is cancelled (the cancelled

//flag will be set to true if cancelled).

};

},

leave: function(element, done) { },

move: function(element, done) { },

//animation that can be triggered before the class is added

beforeAddClass: function(element, className, done) { },

//animation that can be triggered after the class is added

addClass: function(element, className, done) { },

//animation that can be triggered before the class is removed

beforeRemoveClass: function(element, className, done) { },

//animation that can be triggered after the class is removed

removeClass: function(element, className, done) { }

};

});

### 4.10.2 动画效果示例

下面我们来看下DEMO中的例子（例10）。

1.首先，我们在demoApp下定义一个动画效果，匹配CLASS：” .border-animation”

/\*定义动画\*/

demoApp.animation('.border-animation', function(){

return{

beforeAddClass : function (element, className, done) {

$(element).stop().animate({

'border-width':1

},2000, function() {

done();

});

},

removeClass : function (element ,className ,done ) {

$(element).stop().animate({

'border-width':50

},3000, function() {

done();

});

}

};

});

动画效果的含义就是：在匹配CLASS为border-animation的元素添加一个CLASS之前使其边框的宽度在2秒内变为1PX；并在其移除一个CLASS时使其边框的宽度在3秒内变为50PX。

2. 视图中的代码如下（主要，其他相关样式请查看例子代码）：

<div class="border-animation" ng-show="testShow"></div>

<a href="javascript:void(0);" ng-click="testShow=!testShow" >Change</a>

ng-show为false时会为其加上“ng-hide“的CLASS; ng-show为true时会为其移除“ng-hide“的CLASS，从而触发动画效果。

3.其他代码：

demoApp.controller("test10Controller", function($scope, $animate){

$scope.testShow = true;

});

# 5 功能演示

略（详情请看AngularJS/demo WEB演示）

# 6 AngularJS进阶

## 6.1数据绑定原理研究

Angular用户都想知道数据绑定是怎么实现的。你可能会看到各种各样的词汇：$watch、$apply、$digest、dirty-checking...它们是什么？它们是如何工作的呢？这里我想回答这些问题，其实它们在官方的文档里都已经回答了，但是我还是想把它们结合在一起来讲，但是我只是用一种简单的方法来讲解，如果要想了解技术细节，查看源代码。

### 6.1.1 AngularJS扩展事件循环

我们的浏览器一直在等待事件，比如用户交互。假如你点击一个按钮或者在输入框里输入东西，事件的回调函数就会在javascript解释器里执行，然后你就可以做任何DOM操作，等回调函数执行完毕时，浏览器就会相应地对DOM做出变化。（记住，这是个重要的概念），为了解释什么是context以及它如何工作，我们还需要解释更多的概念。

### 6.1.2 $watch 队列

每次你绑定一些东西到你的DOM上时你就会往$watch队列里插入一条$watch。想象一下$watch就是那个可以检测它监视的model里时候有变化的东西。例如你有如下的代码：

/\*View index.html \*/

User: <input type="text" ng-model="user" />

Password: <input type="password" ng-model="pass" />

在这里我们有个$scope.user，他被绑定在了第一个输入框上，还有个$scope.pass，它被绑定在了第二个输入框上，然后我们在$watch list里面加入两个$watch。

再看下面的例子：

/\*Controller controllers.js \*/

app.controller('MainCtrl', function($scope) {

$scope.foo = "Foo";

$scope.world = "World";

});

/\*View index.html \*/

Hello, {{ World }}

这里，即便我们在$scope上添加了两个东西，但是只有一个绑定在了DOM上，因此在这里只生成了一个$watch。

再看下面的例子：

/\*Controller controllers.js \*/

app.controller('MainCtrl', function($scope) {

$scope.people = [...];

});

/\*View index.html \*/

<ul>

<li ng-repeat="person in people">

{{person.name}} - {{person.age}}

</li>

</ul>

这里又生成了多少个$watch呢？每个person有两个（一个name，一个age），然后ng-repeat又有一个，因此10个person一共是(2 \* 10) +1,也就是说有21个$watch。

因此，每一个绑定到了DOM上的数据都会生成一个$watch。

那这写$watch是什么时候生成的呢？

当我们的模版加载完毕时，也就是在linking阶段（Angular分为compile阶段和linking阶段），Angular解释器会寻找每个directive，然后生成每个需要的$watch。

### 6.1.3 $digest循环

还记得我前面提到的扩展的事件循环吗？当浏览器接收到可以被angular context处理的事件时，$digest循环就会触发。这个循环是由两个更小的循环组合起来的。一个处理evalAsync队列，另一个处理$watch队列。 这个是处理什么的呢？$digest将会遍历我们的$watch，然后询问：

•嘿，$watch，你的值是什么？

◦是9。

•好的，它改变过吗？

◦没有，先生。

•（这个变量没变过，那下一个）

•你呢，你的值是多少？

◦报告，是Foo。

•刚才改变过没？

◦改变过，刚才是Bar。

•（很好，我们有DOM需要更新了）

•继续询问直到$watch队列都检查过。

这就是所谓的dirty-checking。既然所有的$watch都检查完了，那就要问了：有没有$watch更新过？如果有至少一个更新过，这个循环就会再次触发，直到所有的$watch都没有变化。这样就能够保证每个model都已经不会再变化。记住如果循环超过10次的话，它将会抛出一个异常，防止无限循环。当$digest循环结束时，DOM相应地变化。

例如：

/\*Controller controllers.js \*/

app.controller('MainCtrl', function() {

$scope.name = "Foo";

$scope.changeFoo = function() {

$scope.name = "Bar";

}

});

/\*View index.html \*/

{{ name }}

<button ng-click="changeFoo()">Change the name</button>

这里我们有一个$watch因为ng-click不生成$watch（函数是不会变的）。

我们可以看出ng的处理流程：

•我们按下按钮；

•浏览器接收到一个事件，进入angular context；

•$digest循环开始执行，查询每个$watch是否变化；

•由于监视$scope.name的$watch报告了变化，它会强制再执行一次$digest循环；

•新的$digest循环没有检测到变化；

•浏览器拿回控制权，更新与$scope.name新值相应部分的DOM。

这里很重要的是每一个进入angular context的事件都会执行一个$digest循环，也就是说每次我们输入一个字母循环都会检查整个页面的所有$watch。

### 6.1.4如何进入angular context

谁决定什么事件进入angular context，而哪些又不进入呢？通过$apply！

如果当事件触发时，你调用$apply，它会进入angular context，如果没有调用就不会进入。现在你可能会问：刚才的例子里我也没有调用$apply啊，为什么？Angular已经做了！因此你点击带有ng-click的元素时，时间就会被封装到一个$apply调用。如果你有一个ng-model="foo"的输入框，然后你敲一个f，事件就会这样调用$apply("foo = 'f';")。

Angular什么时候不会自动为我们$apply呢？

这是Angular新手共同的痛处。为什么我的jQuery不会更新我绑定的东西呢？因为jQuery没有调用$apply，事件没有进入angular context，$digest循环永远没有执行。

我们来看一个有趣的例子：

假设我们有下面这个directive和controller。

/\*Controller app.js \*/

app.directive('clickable', function() {

return {

restrict: "E",

scope: {

foo: '=',

bar: '='

},

template: '<ul style="background-color: lightblue"><li>{{foo}}</li><li>{{bar}}</li></ul>',

link: function(scope, element, attrs) {

element.bind('click', function() {

scope.foo++;

scope.bar++;

});

}

}

});

app.controller('MainCtrl', function($scope) {

$scope.foo = 0;

$scope.bar = 0;

});

它将foo和bar从controller里绑定到一个list里面，每次点击这个元素的时候，foo和bar都会自增1。那我们点击元素的时候会发生什么呢？我们能看到更新吗？答案是否定的。因为点击事件是一个没有封装到$apply里面的常见的事件，这意味着我们会失去我们的计数吗？不会。

真正的结果是：$scope确实改变了，但是没有强制$digest循环，监视foo 和bar的$watch没有执行。也就是说如果我们自己执行一次$apply那么这些$watch就会看见这些变化，然后根据需要更新DOM。

执行$apply：

element.bind('click', function() {

scope.foo++;

scope.bar++;

scope.$apply();

});

$apply是我们的$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$digest循环（除非当前正在执行循环，这种情况下会抛出一个异常，这是我们不需要在那里执行$apply的标志）。

更好的使用$apply的方法：

element.bind('click', function() {

scope.$apply(function() {

scope.foo++;

scope.bar++;

});

})

有什么不一样的？差别就是在第一个版本中，我们是在angular context的外面更新的数据，如果有发生错误，Angular永远不知道。很明显在这个像个小玩具的例子里面不会出什么大错，但是想象一下我们如果有个alert框显示错误给用户，然后我们有个第三方的库进行一个网络调用然后失败了，如果我们不把它封装进$apply里面，Angular永远不会知道失败了，alert框就永远不会弹出来了。

因此，如果你想使用一个jQuery插件，并且要执行$digest循环来更新你的DOM的话，要确保你调用了$apply。

有时候我想多说一句的是有些人在不得不调用$apply时会“感觉不妙”，因为他们会觉得他们做错了什么。其实不是这样的，Angular不是什么魔术师，他也不知道第三方库想要更新绑定的数据。

### 6.1.5使用$watch来监视

你已经知道了我们设置的任何绑定都有一个它自己的$watch，当需要时更新DOM，但是我们如果要自定义自己的watches呢？简单，来看个例子：

/\*Controller app.js \*/

app.controller('MainCtrl', function($scope) {

$scope.name = "Angular";

$scope.updated = -1;

$scope.$watch('name', function() {

$scope.updated++;

});

});

/\*View index.html\*/

<body ng-controller="MainCtrl">

<input ng-model="name" />

Name updated: {{updated}} times.

</body>

这就是我们创造一个新的$watch的方法。第一个参数是一个字符串或者函数，在这里是只是一个字符串，就是我们要监视的变量的名字，在这里，$scope.name(注意我们只需要用name)。第二个参数是当$watch说我监视的表达式发生变化后要执行的。我们要知道的第一件事就是当controller执行到这个$watch时，它会立即执行一次，因此我们设置updated为-1。

例子2：

/\*Controller app.js \*/

app.controller('MainCtrl', function($scope) {

$scope.name = "Angular";

$scope.updated = 0;

$scope.$watch('name', function(newValue, oldValue) {

if (newValue === oldValue) { return; } // AKA first run

$scope.updated++;

});

});

/\*View index.html\*/

<body ng-controller="MainCtrl">

<input ng-model="name" />

Name updated: {{updated}} times.

</body>

watch的第二个参数接受两个参数，新值和旧值。我们可以用他们来略过第一次的执行。通常你不需要略过第一次执行，但在这个例子里面你是需要的。

例子3：

/\*Controller app.js \*/

app.controller('MainCtrl', function($scope) {

$scope.user = { name: "Fox" };

$scope.updated = 0;

$scope.$watch('user', function(newValue, oldValue) {

if (newValue === oldValue) { return; }

$scope.updated++;

});

});

/\*View index.html\*/

<body ng-controller="MainCtrl">

<input ng-model="user.name" />

Name updated: {{updated}} times.

</body>

我们想要监视$scope.user对象里的任何变化，和以前一样这里只是用一个对象来代替前面的字符串。

呃？没用，为啥？因为$watch默认是比较两个对象所引用的是否相同，在例子1和2里面，每次更改$scope.name都会创建一个新的基本变量，因此$watch会执行，因为对这个变量的引用已经改变了。在上面的例子里，我们在监视$scope.user，当我们改变$scope.user.name时，对$scope.user的引用是不会改变的，我们只是每次创建了一个新的$scope.user.name，但是$scope.user永远是一样的。

例子4：

/\*Controller app.js \*/

app.controller('MainCtrl', function($scope) {

$scope.user = { name: "Fox" };

$scope.updated = 0;

$scope.$watch('user', function(newValue, oldValue) {

if (newValue === oldValue) { return; }

$scope.updated++;

}, true );

});

/\*View index.html\*/

<body ng-controller="MainCtrl">

<input ng-model="user.name" />

Name updated: {{updated}} times.

</body>

现在有用了吧！因为我们对$watch加入了第三个参数，它是一个bool类型的参数，表示的是我们比较的是对象的值而不是引用。由于当我们更新$scope.user.name时$scope.user也会改变，所以能够正确触发。

### 6.1.6 总结

我希望你们已经学会了在Angular中数据绑定是如何工作的。我猜想你的第一印象是dirty-checking很慢，好吧，其实是不对的。它像闪电般快。但是，如果你在一个模版里有2000-3000个watch，它会开始变慢。但是我觉得如果你达到这个数量级，就可以找个用户体验专家咨询一下了。

无论如何，随着ECMAScript6的到来，在Angular未来的版本里我们将会有Object.observe那样会极大改善$digest循环的速度。

## 6.2自定义指令详解

angular的指令机制。angular通过指令的方式实现了HTML的扩展，增强后的HTML不仅长相焕然一新，同时也获得了很多强大的技能。更厉害的是，你还可以自定义指令，这就意味着HTML标签的范围可以扩展到无穷大。angular赋予了你造物主的能力。既然是作为angular的精华之一，相应的指令相关的知识也很多的。

### 6.2.1指令的编译过程

　　在开始自定义指令之前，我们有必要了解一下指令在框架中的执行流程：

1.浏览器得到 HTML 字符串内容，解析得到 DOM 结构。

2.ng 引入，把 DOM 结构扔给 $compile 函数处理：

① 找出 DOM 结构中有变量占位符；

② 匹配找出 DOM 中包含的所有指令引用；

③ 把指令关联到 DOM；

④ 关联到 DOM 的多个指令按权重排列；

⑤ 执行指令中的 compile 函数（改变 DOM 结构，返回 link 函数）；

⑥ 得到的所有 link 函数组成一个列表作为 $compile 函数的返回。

3. 执行 link 函数（连接模板的 scope）。

这里注意区别一下$compile和compile，前者是ng内部的编译服务，后者是指令中的编译函数，两者发挥作用的范围不同。compile和link函数息息相关又有所区别，这个在后面会讲。了解执行流程对后面的理解会有帮助。

在这里有些人可能会问，angular不就是一个js框架吗，怎么还能跟编译扯上呢，又不是像C++那样的高级语言。其实此编译非彼编译，ng编译的工作是解析指令、绑定监听器、替换模板中的变量等。因为工作方式很像高级语言编辑中的递归、堆栈过程，所以起名为编译，不要疑惑。

### 6.2.2指令的使用方式及命名方法

　　指令的几种使用方式如下：

* 作为标签：<my-dir></my-dir>
* 作为属性：<span my-dir="exp"></span>
* 作为注释：<!-- directive: my-dir exp -->
* 作为类名：<span class="my-dir: exp;"></span>

　　其实常用的就是作为标签和属性，下面两种用法目前还没见过，感觉就是用来卖萌的，姑且留个印象。我们自定义的指令就是要支持这样的用法。

关于自定义指令的命名，你可以随便怎么起名字都行，官方是推荐用[命名空间-指令名称]这样的方式，像ng-controller。不过你可千万不要用ng-前缀了，防止与系统自带的指令重名。另外一个需知道的地方，指令命名时用驼峰规则，使用时用-分割各单词。如：定义myDirective，使用时像这样：<my-directive>。

### 6.2.3自定义指令的配置参数

下面是定义一个标准指令的示例，可配置的参数包括以下部分：

myModule.directive('namespaceDirectiveName', function factory(injectables) {

var directiveDefinitionObject = {

restrict: string,//指令的使用方式，包括标签，属性，类，注释

priority: number,//指令执行的优先级

template: string,//指令使用的模板，用HTML字符串的形式表示

templateUrl: string,//从指定的url地址加载模板

replace: bool,//是否用模板替换当前元素，若为false，则append在当前元素上

transclude: bool,//是否将当前元素的内容转移到模板中

scope: bool or object,//指定指令的作用域

controller: function controllerConstructor($scope, $element, $attrs, $transclude){...},//定义与其他指令进行交互的接口函数

require: string,//指定需要依赖的其他指令

link: function postLink(scope, iElement, iAttrs) {...},//以编程的方式操作DOM，包括添加监听器等

compile: function compile(tElement, tAttrs, transclude){

return: {

pre: function preLink(scope, iElement, iAttrs, controller){...},

post: function postLink(scope, iElement, iAttrs, controller){...}

}

}//编程的方式修改DOM模板的副本，可以返回链接函数

};

return directiveDefinitionObject;

});

看上去好复杂的样子，定义一个指令需要这么多步骤嘛？当然不是，你可以根据自己的需要来选择使用哪些参数。事实上priority和compile用的比较少，template和templateUrl又是互斥的，两者选其一即可。所以不必紧张，接下来分别学习一下这些参数：

* 指令的表现配置参数：restrict、template、templateUrl、replace、transclude；
* 指令的行为配置参数：compile和link；
* 指令划分作用域配置参数：scope；
* 指令间通信配置参数：controller和require。

### 6.2.3指令的表现参数restrict等

指令的表现配置参数：restrict、template、templateUrl、replace、transclude。

我将先从一个简单的例子开始。

例子的代码如下：

var app = angular.module('MyApp', [], function(){console.log('here')});

app.directive('sayHello',function(){

return {

restrict : 'E',

template : '<div>hello</div>'

};

})

然后在页面中，我们就可以使用这个名为sayHello的指令了，它的作用就是输出一个hello单词。像这样使用：

<say-hello></say-hello>

这样页面就会显示出hello了，看一下生成的代码：

<say-hello>

<div>hello</div>

</say-hello>

　　稍稍解释一下我们用到的两个参数，restirct用来指定指令的使用类型，其取值及含义如下：

|  |  |  |
| --- | --- | --- |
| **取值** | **含义** | **使用示例** |
| E | 标签 | <my-menu title=Products></my-menu> |
| A | 属性 | <div my-menu=Products></div> |
| C | 类 | <div class="my-menu":Products></div> |
| M | 注释 | <!--directive:my-menu Products--> |

默认值是A。也可以使用这些值的组合，如EA，EC等等。我们这里指定为E，那么它就可以像标签一样使用了。如果指定为A，我们使用起来应该像这样：

<div say-hello></div>

从生成的代码中，你也看到了template的作用，它就是描述你的指令长什么样子，这部分内容将出现在页面中，即该指令所在的模板中，既然是模板中，template的内容中也可以使用ng-modle等其他指令，就像在模板中使用一样。

在上面生成的代码中，我们看到了<div>hello</div>外面还包着一层<say-hello>标签，如果我们不想要这一层多余的东西了，replace就派上用场了，在配置中将replace赋值为true，将得到如下结构：

<div>hello</div>

　　replace的作用正如其名，将指令标签替换为了temple中定义的内容。不写的话默认为false。

上面的template未免也太简单了，如果你的模板HTML较复杂，如自定义一个ui组件指令，难道要拼接老长的字符串？当然不需要，此时只需用templateUrl便可解决问题。你可以将指令的模板单独命名为一个html文件，然后在指令定义中使用templateUrl指定好文件的路径即可，如：

templateUrl : ‘helloTemplate.html’

系统会自动发一个http请求来获取到对应的模板内容。是不是很方便呢，你不用纠结于拼接字符串的烦恼了。如果你是一个追求完美的有考虑性能的工程师，可能会发问：那这样的话岂不是要牺牲一个http请求？这也不用担心，因为ng的模板还可以用另外一种方式定义，那就是使用<script>标签。使用起来如下：

<script type="text/ng-template" id="helloTemplate.html">

<div>hello</div>

</script>

你可以把这段代码写在页面头部，这样就不必去请求它了。在实际项目中，你也可以将所有的模板内容集中在一个文件中，只加载一次，然后根据id来取用。

接下来我们来看另一个比较有用的配置：transclude，定义是否将当前元素的内容转移到模板中。看解释有点抽象，不过亲手试试就很清楚了，看下面的代码（例06）：

app.directive('sayHello',function(){

return {

restrict : 'E',

template : '<div>hello，<b ng-transclude></b>！</div>',

replace : true,

transclude : true

};

})

指定了transclude为true，并且template修改了一下，加了一个<b>标签，并在上面使用了ng-transclude指令，用来告诉指令把内容转移到的位置。那我们要转移的内容是什么呢？请看使用指令时的变化：

<say-hello>美女</say-hello>

内容是什么你也看到了哈~在运行的时候，美女将会被转移到<b>标签中，原来此配置的作用就是——乾坤大挪移！看效果：

hello, 美女！

这个还是很有用的，因为你定义的指令不可能老是那么简单，只有一个空标签。当你需要对指令中的内容进行处理时，此参数便大有可用。

### 6.2.4指令的行为参数：compile和link

6.2.3中简单介绍了自定义一个指令的几个简单参数，restrict、template、templateUrl、replace、transclude，这几个理解起来相对容易很多，因为它们只涉及到了表现，而没有涉及行为。我们继续学习ng自定义指令的几个重量级参数：compile和link

* 理解compile和link

　　不知大家有没有这样的感觉，自己定义指令的时候跟写jQuery插件有几分相似之处，都是先预先定义好页面结构及监听函数，然后在某个元素上调用一下，该元素便拥有了特殊的功能。区别在于，jQuery的侧重点是DOM操作，而ng的指令中除了可以进行DOM操作外，更注重的是数据和模板的绑定。jQuery插件在调用的时候才开始初始化，而ng指令在页面加载进来的时候就被编译服务($compile)初始化好了。

在指令定义对象中，有compile和link两个参数，它们是做什么的呢？从字面意义上看，编译、链接，貌似太抽象了点。其实可大有内涵，为了在自定义指令的时候能正确使用它们，现在有必要了解一下ng是如何编译指令的。

* 指令的解析流程详解

　　我们知道ng框架会在页面载入完毕的时候，根据ng-app划定的作用域来调用$compile服务进行编译，这个$compile就像一个大总管一样，清点作用域内的DOM元素，看看哪些元素上使用了指令(如<div ng-modle=”m”></div>)，或者哪些元素本身就是个指令(如<mydierc></mydirec>)，或者使用了插值指令( {{}}也是一种指令，叫interpolation directive)，$compile大总管会把清点好的财产做一个清单，然后根据这些指令的优先级(priority)排列一下，真是个细心的大总管哈~大总管还会根据指令中的配置参数(template，place，transclude等)转换DOM，让指令“初具人形”。

然后就开始按顺序执行各指令的compile函数，注意此处的compile可不是大总管$compile，人家带着$是土豪，此处执行的compile函数是我们指令中配置的，compile函数中可以访问到DOM节点并进行操作，其主要职责就是进行DOM转换，每个compile函数执行完后都会返回一个link函数，这些link函数会被大总管汇合一下组合成一个合体后的link函数，为了好理解，我们可以把它想象成葫芦小金刚，就像是进行了这样的处理。

//合体后的link函数

function AB(){

A(); //子link函数

B(); //子link函数

}

接下来进入link阶段，合体后的link函数被执行。所谓的链接，就是把view和scope链接起来。链接成啥样呢？就是我们熟悉的数据绑定，通过在DOM上注册监听器来动态修改scope中的数据，或者是使用$watchs监听 scope中的变量来修改DOM，从而建立双向绑定。由此也可以断定，葫芦小金刚可以访问到scope和DOM节点。

不要忘了我们在定义指令中还配置着一个link参数呢，这么多link千万别搞混了。那这个link函数是干嘛的呢，我们不是有葫芦小金刚了嘛？那我告诉你，其实它是一个小三。此话怎讲？compile函数执行后返回link函数，但若没有配置compile函数呢？葫芦小金刚自然就不存在了。

正房不在了，当然就轮到小三出马了，大总管$compile就把这里的link函数拿来执行。这就意味着，配置的link函数也可以访问到scope以及DOM节点。值得注意的是，compile函数通常是不会被配置的，因为我们定义一个指令的时候，大部分情况不会通过编程的方式进行DOM操作，而更多的是进行监听器的注册、数据的绑定。所以，小三名正言顺的被大总管宠爱。

听完了大总管、葫芦小金刚和小三的故事，你是不是对指令的解析过程比较清晰了呢？不过细细推敲，你可能还是会觉得情节生硬，有些细节似乎还是没有透彻的明白，所以还需要再理解下面的知识点：

* compile和link的区别

　　其实在我看完官方文档后就一直有疑问，为什么监听器、数据绑定不能放在compile函数中，而偏偏要放在link函数中？为什么有了compile还需要link？就跟你质疑我编的故事一样，为什么最后小三被宠爱了？所以我们有必要探究一下，compile和link之间到底有什么区别。好，正房与小三的PK现在开始。

首先是性能。举个例子：

<ul>

<li ng-repeat="a in array">

<input ng-modle=”a.m” />

</li>

</ul>

我们的观察目标是ng-repeat指令。假设一个前提是不存在link。大总管$compile在编译这段代码时，会查找到ng-repeat，然后执行它的compile函数，compile函数根据array的长度复制出n个<li>标签。而复制出的<li>节点中还有<input>节点并且使用了ng-modle指令，所以compile还要扫描它并匹配指令，然后绑定监听器。每次循环都做如此多的工作。而更加糟糕的一点是，我们会在程序中向array中添加元素，此时页面上会实时更新DOM，每次有新元素进来，compile函数都把上面的步骤再走一遍，岂不是要累死了，这样性能必然不行。

现在扔掉那个假设，在编译的时候compile就只管生成DOM的事，碰到需要绑定监听器的地方先存着，有几个存几个，最后把它们汇总成一个link函数，然后一并执行。这样就轻松多了，compile只需要执行一次，性能自然提升。

另外一个区别是能力。

尽管compile和link所做的事情差不多，但它们的能力范围还是不一样的。比如正房能管你的存款，小三就不能。小三能给你初恋的感觉，正房却不能。

我们需要看一下compile函数和link函数的定义：

function compile(tElement, tAttrs, transclude) { ... }

function link(scope, iElement, iAttrs, controller) { ... }

这些参数都是通过依赖注入而得到的，可以按需声明使用。从名字也容易看出，两个函数各自的职责是什么，compile可以拿到transclude，允许你自己编程管理乾坤大挪移的行为。而link中可以拿到scope和controller，可以与scope进行数据绑定，与其他指令进行通信。两者虽然都可以拿到element，但是还是有区别的，看到各自的前缀了吧？compile拿到的是编译前的，是从template里拿过来的，而link拿到的是编译后的，已经与作用域建立了关联，这也正是link中可以进行数据绑定的原因。

　　我暂时只能理解到这个程度了。实在不想理解这些知识的话，只要简单记住一个原则就行了：如果指令只进行DOM的修改，不进行数据绑定，那么配置在compile函数中，如果指令要进行数据绑定，那么配置在link函数中。

### 6.2.5指令的划分作用域参数：scope

我们在上面写了一个简单的<say-hello></say-hello>，能够跟美女打招呼。但是看看人家ng内置的指令，都是这么用的：ng-model=”m”，ng-repeat=”a in array”，不单单是作为属性，还可以赋值给它，与作用域中的一个变量绑定好，内容就可以动态变化了。假如我们的sayHello可以这样用：<say-hello speak=”content”>美女</say-hello>，把要对美女说的话写在一个变量content中，然后只要在controller中修改content的值，页面就可以显示对美女说的不同的话。这样就灵活多了，不至于见了美女只会说一句hello，然后就没有然后。

为了实现这样的功能，我们需要使用scope参数，下面来介绍一下。

使用scope为指令划分作用域

　　顾名思义，scope肯定是跟作用域有关的一个参数，它的作用是描述指令与父作用域的关系，这个父作用域是指什么呢？想象一下我们使用指令的场景，页面结构应该是这个样子：

<div ng-controller="testC">

<say-hello speak="content">美女</say-hello>

</div>

外层肯定会有一个controller，而在controller的定义中大体是这个样子：

var app = angular.module('MyApp', [], function(){console.log('here')});

app.controller('testC',function($scope){

$scope.content = '今天天气真好！';

});

所谓sayHello的父作用域就是这个名叫testC的控制器所管辖的范围，指令与父作用域的关系可以有如下取值：

|  |  |
| --- | --- |
| **取值** | **说明** |
| false | 默认值。使用父作用域作为自己的作用域 |
| true | 新建一个作用域，该作用域继承父作用域 |
| javascript对象 | 与父作用域隔离，并指定可以从父作用域访问的变量 |

乍一看取值为false和true好像没什么区别，因为取值为true时会继承父作用域，即父作用域中的任何变量都可以访问到，效果跟直接使用父作用域差不多。但细细一想还是有区别的，有了自己的作用域后就可以在里面定义自己的东西，与跟父作用域混在一起是有本质上的区别。好比是父亲的钱你想花多少花多少，可你自己挣的钱父亲能花多少就不好说了。你若想看这两个作用域的区别，可以在link函数中打印出来看看，还记得link函数中可以访问到scope吧。

最有用的还是取值为第三种，一个对象，可以用键值来显式的指明要从父作用域中使用属性的方式。当scope值为一个对象时，我们便建立了一个与父层隔离的作用域，不过也不是完全隔离，我们可以手工搭一座桥梁，并放行某些参数。我们要实现对美女说各种话就得靠这个。使用起来像这样：

scope: {

attributeName1: 'BINDING\_STRATEGY',

attributeName2: 'BINDING\_STRATEGY',...

}

键为属性名称，值为绑定策略。等等！啥叫绑定策略？最讨厌冒新名词却不解释的行为！别急，听我慢慢道来。

　　先说属性名称吧，你是不是认为这个attributeName1就是父作用域中的某个变量名称？错！其实这个属性名称是指令自己的模板中要使用的一个名称，并不对应父作用域中的变量，稍后的例子中我们来说明。再来看绑定策略，它的取值按照如下的规则：

|  |  |  |
| --- | --- | --- |
| **符号** | **说明** | **举例** |
| @ | 传递一个字符串作为属性的值 | str : ‘@string’ |
| = | 使用父作用域中的一个属性，绑定数据到指令的属性中 | name : ‘=username’ |
| & | 使用父作用域中的一个函数,可以在指令中调用 | getName : ‘&getUserName’ |

　　总之就是用符号前缀来说明如何为指令传值。你肯定迫不及待要看例子了，我们结合例子看一下，小二，上栗子~

举例说明

我想要实现上面想像的跟美女多说点话的功能，即我们给sayHello指令加一个属性，通过给属性赋值来动态改变说话的内容 主要代码如下：

app.controller('testC',function($scope){

$scope.content = '今天天气真好！';

});

app.directive('sayHello',function(){

return {

restrict : 'E',

template: '<div>hello,<b ng-transclude></b>,{{ cont }}</div>',

replace : true,

transclude : true,

scope : {

cont : '=speak'

}

};

});

然后在模板中，我们如下使用指令：

<div ng-controller="testC">

<say-hello speak=" content ">美女</say-hello>

</div>

看看运行效果：

美女今天天气真好！

　　执行的流程是这样的：

　　① 指令被编译的时候会扫描到template中的{ {cont} }，发现是一个表达式；

　　② 查找scope中的规则：通过speak与父作用域绑定，方式是传递父作用域中的属性；

　　③ speak与父作用域中的content属性绑定，找到它的值“今天天气真好！”；

　　④ 将content的值显示在模板中。

　　这样我们说话的内容content就跟父作用域绑定到了一其，如果动态修改父作用域的content的值，页面上的内容就会跟着改变，正如你点击“换句话”所看到的一样。

　　这个例子也太小儿科了吧！简单虽简单，但可以让我们理解清楚，为了检验你是不是真的明白了，可以思考一下如何修改指令定义，能让sayHello以如下两种方式使用：

<span say-hello speak="content">美女</span>

<span say-hello="content" >美女</span>

　　答案我就不说了，简单的很。下面有更重要的事情要做，我们说好了要写一个真正能用的东西来着。接下来就结合所学到的东西来写一个折叠菜单，即点击可展开，再点击一次就收缩回去的菜单。

控制器及指令的代码如下（例07）：

app.controller('testC',function($scope){

$scope.title = '个人简介';

$scope.text = '大家好，我是一名前端工程师，我正在研究AngularJs，欢迎大家与我交流';

});

app.directive('expander',function(){

return {

restrict : 'E',

templateUrl : 'expanderTemp.html',

replace : true,

transclude : true,

scope : {

mytitle : '=etitle'

},

link : function(scope,element,attris){

scope.showText = false;

scope.toggleText = function(){

scope.showText = ! scope.showText;

}

}

};

});

HTML中的代码如下：

<script type="text/ng-template" id="expanderTemp.html">

<div class="mybox">

<div class="mytitle" ng-click="toggleText()">

{{mytitle}}

</div>

<div ng-transclude ng-show="showText">

</div>

</div>

</script>

<div ng-controller="testC">

<expander etitle="title">{{text}}</expander>

</div>

　　还是比较容易看懂的，我只做一点必要的解释。首先我们定义模板的时候使用了ng的一种定义方式<script type=”text/ng-template”id="expanderTemp.html">，在指令中就可以用templateUrl根据这个id来找到模板。指令中的{{mytitle}}表达式由scope参数指定从etitle传递，etitle指向了父作用域中的title。为了实现点击标题能够展开收缩内容，我们把这部分逻辑放在了link函数中，link函数可以访问到指令的作用域，我们定义showText属性来表示内容部分的显隐，定义toggleText函数来进行控制，然后在模板中绑定好。 如果把showText和toggleText定义在controller中，作为$scope的属性呢？显然是不行的，这就是隔离作用域的意义所在，父作用域中的东西除了title之外通通被屏蔽。

上面的例子中，scope参数使用了=号来指定获取属性的类型为父作用域的属性，如果我们想在指令中使用父作用域中的函数，使用&符号即可，是同样的原理。

### 6.2.6指令间通信参数：controller和require

　　使用指令来定义一个ui组件是个不错的想法，首先使用起来方便，只需要一个标签或者属性就可以了，其次是可复用性高，通过controller可以动态控制ui组件的内容，而且拥有双向绑定的能力。当我们想做的组件稍微复杂一点，就不是一个指令可以搞定的了，就需要指令与指令的协作才可以完成，这就需要进行指令间通信。

想一下我们进行模块化开发的时候的原理，一个模块暴露（exports）对外的接口，另外一个模块引用（require）它，便可以使用它所提供的服务了。ng的指令间协作也是这个原理，这也正是自定义指令时controller参数和require参数的作用。

controller参数用于定义指令对外提供的接口，它的写法如下：

controller: function controllerConstructor($scope, $element, $attrs, $transclude)

它是一个构造器函数，将来可以构造出一个实例传给引用它的指令。为什么叫controller（控制器）呢？其实就是告诉引用它的指令，你可以控制我。至于可以控制那些东西呢，就需要在函数体中进行定义了。先看controller可以使用的参数，作用域、节点、节点的属性、节点内容的迁移，这些都可以通过依赖注入被传进来，所以你可以根据需要只写要用的参数。关于如何对外暴露接口，我们在下面的例子来说明。

require参数便是用来指明需要依赖的其他指令，它的值是一个字符串，就是所依赖的指令的名字，这样框架就能按照你指定的名字来从对应的指令上面寻找定义好的controller了。不过还稍稍有点特别的地方，为了让框架寻找的时候更轻松些，我们可以在名字前面加个小小的前缀：^，表示从父节点上寻找，使用起来像这样：require : ‘^directiveName’，如果不加，$compile服务只会从节点本身寻找。另外还可以使用前缀：？，此前缀将告诉$compile服务，如果所需的controller没找到，不要抛出异常。

所需要了解的知识点就这些，接下来是例子时间，依旧是从书上抄来的一个例子，我们要做的是一个手风琴菜单，就是多个折叠菜单并列在一起，此例子用来展示指令间的通信再合适不过。

首先我们需要定义外层的一个结构，起名为accordion，代码如下：

app.directive('accordion',function(){

return {

restrict : 'E',

template : '<div ng-transclude></div>',

replace : true,

transclude : true,

controller :function(){

var expanders = [];

this.gotOpended = function(selectedExpander){

angular.forEach(expanders,function(e){

if(selectedExpander != e){

e.showText = false;

}

});

}

this.addExpander = function(e){

expanders.push(e);

}

}

}

});

需要解释的只有controller中的代码，我们定义了一个折叠菜单数组expanders，并且通过this关键字来对外暴露接口，提供两个方法。gotOpended接受一个selectExpander参数用来修改数组中对应expander的showText属性值，从而实现对各个子菜单的显隐控制。addExpander方法对外提供向expanders数组增加元素的接口，这样在子菜单的指令中，便可以调用它把自身加入到accordion中。

看一下我们的expander需要做怎样的修改呢：

app.directive('expander',function(){

return {

restrict : 'E',

templateUrl : 'expanderTemp.html',

replace : true,

transclude : true,

require : '^?accordion',

scope : {

title : '=etitle'

},

link : function(scope,element,attris,accordionController){

scope.showText = false;

accordionController.addExpander(scope);

scope.toggleText = function(){

scope.showText = ! scope.showText;

accordionController.gotOpended(scope);

}

}

};

});

首先使用require参数引入所需的accordion指令，添加?^前缀表示从父节点查找并且失败后不抛出异常。然后便可以在link函数中使用已经注入好的accordionController了，调用addExpander方法将自己的作用域作为参数传入，以供accordionController访问其属性。然后在toggleText方法中，除了要把自己的showText修改以外，还要调用accordionController的gotOpended方法通知父层指令把其他菜单给收缩起来。

指令定义好后，我们就可以使用了，使用起来如下：

<accordion>

<expander ng-repeat="expander in expanders" etitle="expander.title">

{{expander.text}}

</expander>

</accordion>

外层使用了accordion指令，内层使用expander指令，并且在expander上用ng-repeat循环输出子菜单。请注意这里遍历的数组expanders可不是accordion中定义的那个expanders，如果你这么认为了，说明还是对作用域不够了解。此expanders是ng-repeat的值，它是在外层controller中的，所以，在testC中，我们需要添加如下数据：

$scope.expanders = [

{title: '个人简介',

text: '大家好，我是一名前端工程师，我正在研究AngularJs，欢迎大家与我交流'},

{title: '我的爱好',

text: 'LOL '},

{title: '性格',

text: ' 我的性格就是无性格'}

];

## 6.3 性能及调优

### 6.3.1性能测试

AnglarJS作为一款优秀的Web框架，可大大简化前端开发的负担。

AnglarJS很棒，但当处理包含复杂数据结构的大型列表时，其运行速度就会非常慢。

这是我们将核心管理页面迁移到AngularJS过程中遇到的问题。这些页面在显示500行数据时本应该工作顺畅，但首个方法的渲染时间竟花费了7秒，太可怕了。后来，我们发现了在实现过程中存在两个主要性能问题。一个与“ng-repeat ”指令有关，另一个与过滤器有关。

AngularJS 中的ng-repeat在处理大型列表时，速度为什么会变慢？

AngularJS中的ng-repeat在处理2500个以上的双向数据绑定时速度会变慢。这是由于AngularJS通过“dirty checking”函数来检测变化。每次检测都会花费时间，所以包含复杂数据结构的大型列表将降低你应用的运行速度。

提高性能的先决条件

时间记录指令

为了测量一个列表渲染所花费的时间，我们写了一个简单的程序，通过使用“ng-repeat”的属性“$last”来记录时间。时间存放在TimeTracker服务中，这样时间记录就与服务器端的数据加载分开了。

// Post repeat directive for logging the rendering time

angular.module('siApp.services').directive('postRepeatDirective',

['$timeout', '$log', 'TimeTracker',

function($timeout, $log, TimeTracker) {

return function(scope, element, attrs) {

if (scope.$last){

$timeout(function(){

var timeFinishedLoadingList = TimeTracker.reviewListLoaded();

var ref = new Date(timeFinishedLoadingList);

var end = new Date();

$log.debug("## DOM rendering list took: " + (end - ref) + " ms");

});

}

};

}

]);

// Use in HTML:

<tr ng-repeat="item in items" post-repeat-directive>…</tr>

Chrome开发者工具的时间轴（Timeline）属性

在Chrome开发者工具的时间轴标签中，你可以看见事件、每秒内浏览器帧数和内存分配。“memory”工具用来检测内存泄漏，及页面所需的内存。当帧速率每秒低于30帧时就会出现页面闪烁问题。“frames”工具可帮助了解渲染性能，还可显示出一个JavaScript任务所花费的CPU时间。

通过限制列表的大小进行基本的调优

缓解该问题，最好的办法是限制所显示列表的大小。可通过分页、添加无限滚动条来实现。

分页，我们可以使用AngularJS的“limitTo”过滤器（AngularJS1.1.4版本以后）和“startFrom”过滤器。可以通过限制显示列表的大小来减少渲染时间。这是减少渲染时间最高效的方法。

### 6.3.2七大调优法则

1.渲染没有数据绑定的列表

这是最明显的解决方案，因为数据绑定是性能问题最可能的根源。如果你只想显示一次列表，并不需要更新、改变数据，放弃数据绑定是绝佳的办法。不过可惜的是，你会失去对数据的控制权，但除了该法，我们别无选择。

2.不要使用内联方法计算数据

为了在控制器中直接过滤列表，不要使用可获得过滤链接的方法。“ng-repeat”会评估每个表达式。在我们的案例中，“filteredItems()”返回过滤链接。如果评估过程很慢，它将迅速降低整个应用的速度。

* <li ng-repeat="item in filteredItems()"> //这并不是一个好方法，因为要频繁地评估。
* <li ng-repeat="item in items"> //这是要采用的方法

3.使用两个列表（一个用来进行视图显示，一个作为数据源）

将要显示的列表与总的数据列表分开，是非常有用的模型。你可以对一些过滤进行预处理，并将存于缓存中的链接应用到视图上。下面案例展示了基本实现过程。filteredLists变量保存着缓存中的链接，applyFilter方法来处理映射。

/\* Controller \*/

// Basic list

var items = [{name:"John", active:true }, {name:"Adam"}, {name:"Chris"}, {name:"Heather"}];

// Init displayedList

$scope.displayedItems = items;

// Filter Cache

var filteredLists['active'] = $filter('filter)(items, {"active" : true});

// Apply the filter

$scope.applyFilter = function(type) {

if (filteredLists.hasOwnProperty(type){ // Check if filter is cached

$scope.displayedItems = filteredLists[type];

} else {

/\* Non cached filtering \*/

}

}

// Reset filter

$scope.resetFilter = function() {

$scope.displayedItems = items;

}

/\* View \*/

<button ng-click="applyFilter('active')">Select active</button>

<ul><li ng-repeat="item in displayedItems">{{item.name}}<li></ul>

4.在其他模板中使用ng-if来代替ng-show

如果你用指令、模板来渲染额外的信息，例如通过点击来显示列表项的详细信息，一定要使用 ng-if（AngularJSv. 1.1.5以后）。ng-if可阻止渲染（与ng-show相比）。所以其它DOM和数据绑定可根据需要进行评估。

<li ng-repeat="item in items">

<p> {{ item.title }} </p>

<button ng-click="item.showDetails = !item.showDetails">Show details</buttons>

<div ng-if="item.showDetails">

{{item.details}}

</div>

</li>

5.不要使用ng-mouseenter、ng-mouseleave等指令

使用内部指令，像ng-mouseenter，AngularJS会使你的页面闪烁。浏览器的帧速率通常低于每秒30帧。使用jQuery创建动画、鼠标悬浮效果可以解决该问题。确保将鼠标事件放入jQuery的.live()函数中。

6.关于过滤的小提示：通过ng-show隐藏多余的元素

对于长列表，使用过滤同样会减低工作效率，因为每个过滤都会创建一个原始列表的子链接。在很多情况下，数据没有变化，过滤结果也会保持不变。所以对数据列表进行预过滤，并根据情况将它应用到视图中，会大大节约处理时间。

在ng-repeat指令中使用过滤器，每个过滤器会返回一个原始链接的子集。AngularJS 从DOM中移除多余元素（通过调用 $destroy），同时也会从$scope中移除他们。当过滤器的输入发生改变时，子集也会随着变化，元素必须进行重新链接，或着再调用$destroy。

大部分情况下，这样做很好，但一旦用户经常过滤，或者列表非常巨大，不断的链接与销毁将影响性能。为了加快过滤的速度，你可以使用ng-show和ng-hide指令。在控制器中，进行过滤，并为每项添加一个属性。依靠该属性来触发ng-show。结果是，只为这些元素增加ng-hide类，来代替将它们移除子列表、$scope和DOM。

触发ng-show的方法之一是使用表达式语法。ng-show的值由表达式语法来确定。可以看下面的例子：

<input ng-model="query"></input>

<li ng-repeat="item in items" ng-show="([item.name] | filter:query).length"> {{item.name}} </li>

<span style="font-size: 14px; line-height: 24px; font-family:; white-space: normal;"></span>

7.关于过滤的小提示：防抖动输入

解决第6点提出的持续过滤问题的另一个方法是防抖动用户输入。例如，如果用户输入一个搜索关键词，只当用户停止输入后，过滤器才会被激活。使用该防抖动服务的一个很好的解决方案请见： http://jsfiddle.net/Warspawn/6K7Kd/。将它应用到你的视图及控制器中，如下所示：

/\* Controller \*/

// Watch the queryInput and debounce the filtering by 350 ms.

$scope.$watch('queryInput', function(newValue, oldValue) {

if (newValue === oldValue) { return; }

$debounce(applyQuery, 350);

});

var applyQuery = function() {

$scope.filter.query = $scope.query;

};

/\* View \*/

<input ng-model="queryInput"/>

<li ng-repeat= item in items | filter:filter.query>{{ item.title }} </li>

</ul>

Ng会根据array的长度复制出n个<li>标签。而复制出的<li>节点中还有<input>节点并且使用了ng-modle指令，所以ng会对所有的<input>绑定监听器（事件）。如果array很大，就会绑定太多的事件，性能出现问题。