

实 战

AngularJS 深度剖析与最佳实践

雪 狼 破 狼 彭洪伟 编著



机械工业出版社
China Machine Press

图书在版编目 (CIP) 数据

AngularJS 深度剖析与最佳实践 / 雪狼, 破狼, 彭洪伟编著. —北京: 机械工业出版社, 2015.11

(实战)

ISBN 978-7-111-52096-2

I. A… II. ①雪… ②破… ③彭… III. 超文本标记语言—程序设计 IV. TP312

中国版本图书馆 CIP 数据核字 (2015) 第 270565 号

本书深入讲解 AngularJS 的基本概念及其背后的原理, 包括完整的开发框架与最佳实践。不仅抽丝剥茧地展现了 AngularJS 的诸多特性与技巧, 还讲解了工程实践中容易陷入的“坑”, 是从小工走向专家的必备参考。

本书首先从实战开始, 通过实战演练逐步带领读者体验 Angular 的开发过程, 并随着进度的推进, 引入所需的技术和概念。其次对在实战中提到的一些概念进行深入讲解: 包括这些概念怎么用, 什么时候用, 什么时候不用等。然后讲解这些概念背后的原理, 看看这些概念之间是如何协作的, 包括 AngularJS 的工作模式等。最后介绍最佳实践, 将主要介绍实战经验, 包括如何发掘一些不常用但很有用的 API, 如何把看起来平淡无奇的框架特性运用得出神入化等。此外, 作者还从实际工作中总结了一些开发技巧和容易陷入的“坑”, 以及常见的优秀工具及其使用经验, 这对于实际开发工作非常有参考价值。



AngularJS 深度剖析与最佳实践

出版发行: 机械工业出版社 (北京市西城区百万庄大街 22 号 邮政编码: 100037)

责任编辑: 吴 怡

责任校对: 殷 虹

印 刷:

版 次: 2016 年 1 月第 1 版第 1 次印刷

开 本: 186mm×240mm 1/16

印 张: 21.5

书 号: ISBN 978-7-111-52096-2

定 价: 69.00 元

凡购本书, 如有缺页、倒页、脱页, 由本社发行部调换

客服热线: (010) 88379426 88361066

投稿热线: (010) 88379604

购书热线: (010) 68326294 88379649 68995259

读者信箱: hzit@hzbook.com

版权所有·侵权必究

封底无防伪标均为盗版

本书法律顾问: 北京大成律师事务所 韩光 / 邹晓东

Preface 序

这是一本具有强烈 ThoughtWorks 项目风格的书。书中打造的实战项目完全遵循了 ThoughtWorks 工程实践，一步一步从最初的框架通过快速迭代逐步丰富项目的骨肉，并在这个过程中抽丝剥茧地展现了 AngularJS 的诸多特性与技巧，如循循善诱的导师一步步指导着你从 AngularJS 的小工走向专家。

这里所谓的“专家”不仅仅是指你对 AngularJS 的诸多技巧尽皆了然于胸，能够挥洒自如地运用于项目开发中——若能如此，不过是“唯手熟尔”的工匠罢了。真正的专家需要从大处着手，挖掘这门技术背后隐含的设计思想与哲学，换言之，需要知其所以然，却又不偏废细节，锱铢必较每个变量函数的命名格式，使代码臻于完美，并从中提炼出能够推而广之的最佳实践。

从知其所以然入手，书中的第 3 章“背后的原理”加强了内容的深度，使得本书不至于沦落为一本 Example Step by Step。书中通过对 MVVM 模式的阐释，解释了 Angular JS 的设计原理与启动流程，并给出了 Angular JS 开发的注意事项。书中写道：

MVVM 模式的要点是：以领域 Model 为中，遵循“分离关注点”设计原则。这也是 Angular 的模型驱动思维与 jQuery 的 DOM 驱动思维的显著差异。所以我们在做 Angular 开发的时候应该谨记以下两点：

- ❑ 绝不要先设计你的页面，然后用 DOM 操作去改变它。
- ❑ 指令不是封装 jQuery 代码的“天堂”。

又例如细节之处，本书作者仿佛是踮着针尖在跳舞，刻绘的细节纤毫毕现；又佐以代码，论证有理有据；阅读时，真好像是你和雪狼、破狼在一起结对编程呢。例如书中在提及对服务访问对象（SAO）的封装时，给出了这样两段代码：

```
angular.module('com.ngnice.app').controller('ReaderCreateCtrl', function Reader-  
CreateCtrl($resource) {
```

```

var vm = this;
var Reader = $resource('/api/readers/:id', {id: '@id'});
vm.submit = function(form) {
    Reader.save(form);
};
});

```

封装后：

```

angular.module('com.ngnice.app').controller('ReaderCreateCtrl', function
    Reader-Ctrl(Reader) {
    var vm = this;
    vm.submit = function(form) {
        Reader.save(form);
    };
});

```

寥寥几行代码的区别，却体现了作者对于代码可读性的执着追求。如此内容在书中俯拾皆是。作者对整洁代码的敏感度，就好像水银温度计对气温的感知一般，哪怕是一丝一毫都能准确感知，进而在展开的文字叙述中潜移默化地影响读者。尤其针对初学者，作者从一开始就展示了什么是 AngularJS 之美，什么是代码之美，什么是设计之美，就好似建立了 AngularJS 世界的“潜规则”，入了这个圈，你需如此这般，否则就得荆棘一路，步履蹒跚。而那些优秀的工程实践，例如测试驱动开发、面向模型编程、迭代的演化、一次只做一件事情的行为准则，则完全融化成本书的血液，通过简单朴实的词语，天然地流淌在整本书中，和风细雨，润物细无声。

我与本书的作者之一破狼相交甚深，虽然一直未有机会共同战斗在一个项目，却也有许多机会彼此沟通各自对设计的理解。在面向对象设计、领域驱动设计、架构设计等诸多方面，我们抱有相同的设计态度，可谓志同道合。问道技术，犹如饮酒论文，酒酣耳热时，得聆佳音，当浮一大白，人生乐趣大抵如此。虽然我对前端技术所知了了，但阅读此书，许多论点刚好击中我的肺腑，那种如寻觅到知己一般的快乐，真可以说是阅读之余的额外收获了。我喜欢此书的朴实，它没有去构架飘渺高深的理论，没有浮夸地吹嘘 AngularJS 如何优秀，在前端开发中所向披靡。技术人写文章，常常没有卖弄，只是踏实地表达对技术的一己之得，读者得到的却是字字铮铮的金石之音。

显然，这是一本工程师写给工程师阅读的书，我只可惜这本书的出版来得有点晚了。是为序。

张逸

2015 年 8 月 7 日夜 旅行中，在斯坦福大学安静的校园

新时代

新挑战

时代已经不同了！

17 年前，当我的第一个作品推入市场的时候，互联网才刚刚传入中国。

那时候的软件不需要联网，每个用户也不需要知道其他用户的存在。

那时候只需要考虑 PC 运行环境，而需要考虑的屏幕分辨率也只有区区三种。

那时候的软件项目组多则十几人，少则一人，而发布周期常常会达到半年之久。

现在，一切都不同了。

现在，连一个手机手电筒软件都在偷偷联网，不能联网的游戏也已经是老古董的代名词。

现在，软件不但运行在 PC 上，还要运行在智能手机上，运行在各种 Pad 上，屏幕分辨率更是多到让研发和测试工程师发怵的地步。

现在，外界看到的产品其实只是冰山一角，它背后还有很多子系统紧密协作来提供支持，需求和架构的复杂度也暴增。

但最大的挑战恐怕还是来自发布周期——一期版本在一个月上线已是常态，而修复 bug 的时间限制则往往以小时计，甚至以分钟计。

没错，这些都是新的挑战！好在，我们也有了新技术！

新技术

这 17 年间，软件业最大的技术革命，当然首推互联网。

互联网不但拓展了软件业的业务范围，更改变了程序员获取知识和解决问题的方式。

如今，一个不会 Google（以及翻墙），没上过 GitHub，不知道 Stackoverflow 的程序员

很难想象会有什么发展空间。

排在第二位的技术革命，当推移动互联网以及智能终端。这场革命不但把曾经的王者诺基亚打落凡尘，而且让苹果重新登上王位。

这两场技术革命让开源运动遍地开花，更催生了无数的新技术。

且不提 HTTP/HTML/JavaScript/CSS 这些耳熟能详的互联网基石，就连在互联网革命爆发之前已经就已经相当成熟的 OO 领域也有了很大的进展。

以 MVC 为例，它不但衍生出 MVP、MVVM 等很多变种，而且从后端领域扩展到了前端领域。而现在日益火爆的 Angular，正是 MVC 在前端领域的代表作之一。

一个“极客”总是痴迷于各种“漂亮”的技术，而 Angular 当之无愧地是其中之一，它可供借鉴的地方很多：

- 如何弥补语言的先天不足。
- 如何干净漂亮地解耦。
- 如何设计“小而美”的类 / 代码块。

所以，即使你还没有下决心把 Angular 应用到项目中，也可以在学习 Angular 的过程中获得一些启迪，帮助你重构现有项目。

面对技术的快速进步，有人会感到恐慌，有人会盲目地追踪一切新技术，而真正的极客会看到“新”技术中那些“不变”的元素，会在“新挑战”中看到“新机遇”，并且把握。

新机遇

一方面出现了前所未有的挑战，另一方面出现了前所未有的技术，这样的机遇并不多，“极客”们欢呼雀跃。

对于公司，它将影响产品形态、开发速度和产品品质，也会对团队的组织架构带来改变。比如，伴随着设备的多样化，网络服务的访问入口变得多样化：不但需要有供电脑访问的网站，还需要供手机访问的网站、供 Pad 访问的网站，对于一些追求极致用户体验的公司来说，还会提供给安卓设备用的 App、给苹果设备用的 App。

作为开发人员，也许你会看到或正在经历一个工作量暴增的时代，但是，不要紧张，事情没那么坏。在新时代，有一项重要且迅速成长的技术革新，那就是“前后端分离架构”，它可以有效遏制工作量的暴增。“前后端分离架构”正是伴随着“前端 MVC”的成长而成长的。

它的原理很简单：虽然多出了很多访问入口，但是其背后的业务逻辑并没有本质性变化，那么，我们是否可以让这一套业务逻辑为多种不同形态的终端服务呢？答案是肯定的，

那就是让后端只提供跟业务逻辑紧密相关的那部分 API，而用户交互等非核心逻辑则交给前端程序来完成。

这样，我们的工作并不会成倍增长，而是可以先着重开发一个版本，让后端 API 和一种形态的前端应用变得成熟，然后再去开发其他形态的前端应用。而这些其他形态的前端程序的工作量和风险都比较容易得到控制。

但是，根据康威定律，在新的程序架构下，项目的组织架构甚至整个公司的组织架构都将发生相应的变化，而最显著的变化就是出现了专门的前端工程师。前端工程师往往不是零基础开始的，一小部分来自原来负责切图或写 JavaScript 特效的工程师，不过大部分是从以前开发 Web 应用的程序员转型而来的。

无论对于公司还是个人，“前端 MVC”以及相应的“前后端分离架构”都是一个新的机遇。不思进取的王者终会没落，勤奋好学的新星将会崛起，希望本书能有幸成为你的助力。

致读者

写给想转职或兼修前端 Web 工程师

本书面向的读者，第一大群体是 Web 工程师。“前后端分离架构”出现之前，在大多数 Web 应用中，无论是核心的业务逻辑，还是表现层的交互逻辑，都是完全运行在服务端的。写这类程序的程序员就是这里所说的 Web 工程师。

随着“前后端分离架构”的普及，原来的开发方式将主动或被迫转变。本书将通过实例引导你完成到“前后端分离架构”的思维转变，以及与此相关的技术。

如果你是个 Web 工程师，在读本书的时候请留意用户交互逻辑是如何完全移交给前端程序的，而后端程序又做了哪些精简，特别要注意体会模块职责的单一化、专业化趋势。

对于部分转职过来的 Web 工程师，除了转换思维以外，还有一大挑战是前端庞杂的知识体系：HTML/CSS/JavaScript/ 前端工具链 / 浏览器兼容性等，每一个领域都相当庞大。

在本书中，我们无法对此展开讲解，但这些知识对于做实际项目又是必需的。所以，我们只能在附录中提供一些重要的技术要点和“坑”，并且给出一些在线学习资源和书单。这些大部分都是从我们开展培训时所使用的课件改编而来的，具有很强的实战性、实用性。希望可以为你提供一些第三方资料，作为进一步学习的起点。

写给想进阶为专业前端的切图师

在很多开发组中，切图师往往由初级程序员或美工担任，有没有想过自己将来向哪里

发展？除了面向对象、项目管理等必学的基础技能之外，还可以学习数据库、后端框架、安全技术等，转职为后端工程师。也可以学习 HTML/CSS/JavaScript、用户体验、交互设计、前端框架等，转职为前端工程师。

当然，如果你足够聪明和有足够的进取心，你也可以两者兼修，成为一名全栈工程师。不过，相对来说，前端这条路径可能更加平缓。而且，这几年前端职位正逐渐火爆，从个人职业发展来说，这也是个不错的选择，过一段时间后未必再有这样好的机遇。从切图师到前端，这条路并非荆棘重重。事实上，没有传统 Web 工程师的思维定势，这反倒会是个优点。在笔者的编程、咨询和教学过程中，曾接触过一些对 Angular 感兴趣的人，总体上说，转变思维比导入新思维的难度更大。圈子里还常流传一些无稽之谈，比如，Angular 是 Google 开发的，面向的是 Google 中那些妖怪级程序员。那都是乱传的，没那么恐怖。

我写下这些，是希望你们可以轻装上阵，Angular 的很多设计都是遵循“最小意外”原则的，靠直觉就可以掌握，“高估难度”有害无益。

不过，难度仍然是有的。读本书之前，你至少应该已经熟悉了 JavaScript 语法，对 Angular 的各种概念有了大致的了解。如果你对很多新名词不知所云，那么建议先去翻阅一下附录中的书籍，浏览一下网上关于 Angular 入门的文章。

对于切图师来说，MVC 方面的基础往往会成为短板，而 JavaScript 中一些诡异的特性也常常带来困扰。所以，本书会穿插一些这方面的简短知识。但是，对于一个立志成为“极客”的初级程序员来说，这仍然是不够的，所以，在附录中我们还提供了一些网址和书单，希望本书能帮你开启职业生涯的新阶段。

2.0 要来了，本书会过时吗？

Angular 2.x 已经进入了 Alpha 测试阶段，那么，不免有人担心，等到 2.x 推出的时候，本书会过时吗？从实现细节上来讲，会的。从思想上来讲，不会。从实用性上来讲，不会。

1.x 和预计 2016 年推出的 2.x 在语法甚至一些底层实现上是截然不同的。

据目前得到的消息，2.x 将使用 TypeScript 和 ES6 作为主体语言，那时候，本书的很多代码将不再适用于 2.x。而由于 2.x 彻底抛弃了 IE11 之前的低版本浏览器，它可以借助最新的浏览器特性进行底层实现，不用为了向后兼容而使用“脏检查”等技术来弥补浏览器的不足。在这些细节上，1.x 和 2.x 几乎没有共通之处。这一点一直被人诟病，也是一些人对 Angular 的前途深表担忧的原因。不过，从另一个角度来看，2.x 的这种改进也是一种勇敢的改革，可以让它轻装前进，更有利于长远发展。

好消息是，2.x 不是 1.x 的替代品。官方已经宣布，即使 2.x 推出，也仍然会对 1.x 进行

长期维护。这就有点类似于 Query 2.x 不再兼容 IE8，而 Query 1.x 仍然兼容 IE8 并继续向前发展一样。这种版本策略可以防止 Angular 背上向老旧浏览器兼容的包袱。

1.x 和 2.x 在编程模型上并没有太大的差异，它们都基于 MVVM 模型，都具有双向绑定功能（即使底层实现方式已经变了），都具有相同的设计哲学——利用高内聚的小模块组合出最终程序。而这些在我们的书中都有所体现。在目录结构、指令的分类等方面，本书也从 2.x 中引入了很多更好的实践。

从实用性上来说，本书更不会过时。2.x 的浏览器兼容性起点就是 IE11，这不是因为细节层面的问题，而是从底层原理上就不可能——它依赖太多的新特性。而在国内市场上，彻底抛弃 IE11 以下的版本恐怕还会是一个长期的历程——即使最乐观的估计，至少也需要两年。当然，手机端的浏览器版本更新要快得多，所以，预计 2.x 最早会被用在手机版上。

固然，2.x 是个高大上的版本，但目前在国内还是个屠龙之术。如果要在现实中使用，还是先学好 1.x 吧。按照本书的指引，你可以提前领略 2.x 的优点，而不用付出兼容性的代价。当然，等本书的 2.x 版推出时，这种熟悉的味道也会让你有一个更高的起点。

阅读指南

Angular 的学习曲线大概是这样的：入门非常容易，中级的时候会发现需要深入理解很多概念，高级的时候需要掌握 Angular 的工作原理，而想成为专家则很难，需要经过很多工程实践的磨练。

本书的主体结构也是针对这样的学习曲线设计的。

首先，初级阶段，实战演练

我们会带你实战中逐步体验 Angular 的开发过程，并随着进度的推进，逐步引入所需的技术和概念。

然后，中级阶段，概念介绍

在实战中提到的一些概念不会就地展开，而是只做简介，到了这个阶段，会对概念进行深入讲解：是什么，为什么，怎么用，什么时候用，什么时候不用等。

接下来，高级阶段，工作原理

学习了这些概念，我们还要把它们串起来，向你揭示 Angular 的工作原理，看看这些概念之间是如何协作的。

最后，专家阶段：最佳实践，技巧

前面主要是入门和理论，而这部分将主要以实战经验为主。

只把 Angular 用熟了是不够的，我们还要把它整合进更宏观的开发过程中，不但要考虑开发，更要考虑维护。我们要如何开发容易维护的 Angular 程序？请看第 4 章。

专家还需要掌握一些技巧去把复杂问题简单化，发掘一些不常用但很有用的 API，把看起来平淡无奇的框架特性运用得出神入化，第 5 章将集中展现这一点。

坑

在前面的章节中零零散散提到了一些需要注意的地方，但是这样不方便查阅，所以我们把需要注意的地方作为独立的一大章，把我们帮别人解决过的一些典型问题收集在一起。当然，我们也会在读者社区继续维护并更新这些“坑”，而不是等再版时才发布。我们希望能把这本书做成“活的”，让这本书更加物超所值，不辜负读者对我们的信任。

工具

工欲善其事，必先利其器。充分发挥工具的力量是开发人员的重要素质，日常用到的工具你真的用熟练了吗？有没有更好的工具？我们会把实战中觉得对自己帮助最大的工具及其使用经验分享给你。

更多

在实战中，有很多需求是不显眼但很重要的，比如 SEO、访问统计等，在实际项目中，这些往往是不能忽视的。我们会专门通过一章来讲解如何结合 Angular 和第三方软件干净漂亮地解决这些问题。

Hybrid 应用和手机 Web 越来越普及，手机版开发的需求也越来越高，在 Angular 的基础上，开发手机版变得容易多了。而且，也已经有了比较成熟的工具和框架，我们会简要讲解一下手机版开发的方法和框架。

附录

软件开发需要很多综合技能，但本书容量有限，我们也不可能是每个领域的专家。因此，我们会“授人以渔”，给出一些在线资源和书单，供大家深入学习或作为备查资料。

关于随书代码

书中所摘录的只是全部代码的一小部分，大部分代码都放在了 GitHub 上。地址是 <https://github.com/ng-nice/code>。

如果你查看 GitHub 历史，会发现总的提交数并不多。这是因为要方便教学，所以在提交前进行了合并。所保留的这些提交大都和书中的主要进度有关，略去了细节提交。所以，本书中代码的提交粒度不能代表实际项目中的提交粒度，在实际项目中，其提交粒度通常比本书中所示范的更小。阅读代码时请记住这一点，以免养成“大粒度提交”的坏习惯。

另外，文中的 JavaScript 代码（包括摘引的 Angular 源码）全都使用了两格缩进模式，这主要是考虑到图书排版问题，希望少一些不必要的换行。你们在现实项目中愿意用两格或四格均可，只要项目组内保持一致即可。

关于内容的重复

仔细阅读，可能会发现有些内容会在多个不同的章节中重复讲解，这当然不是凑字数，而是尽可能符合人的记忆规律——把重要的内容在不同的场景下重复，对于深入掌握重点是很有帮助的。

关于写作风格

这是一本多人协作的书，虽然我们进行了后期统稿，但在语言风格等方面仍难免会有不一致的地方，我们期待你们的反馈，以便将来改进。

你的好，我永远记得！

双狼的感恩

双狼的本次合作起于机械工业出版社编辑吴怡的邀请。作为 ThoughtWorks 的 Tech Lead，双狼都有很多工作任务，原定 6 个月的写书计划，被拖到了 8 个月，感谢吴怡的耐心与推动。

还有很多 ThoughtWorker 为本书做出了贡献：

- ❑ 张逸，资深 ThoughtWorker，很多技术书籍的作者或译者。一直在鼓励我们，并给了我们很多帮助。
- ❑ 彭洪伟，本书的第三作者。在交稿压力最大的时候，承担了“工具”篇的撰写工作，保障了本书的尽早交稿。
- ❑ 陈嘉，幕后的贡献者，全栈式工程师。帮我们设计了“双狼说”微信公众号的 Logo，从技术的角度帮我们审稿，并提了一些非常有用的建议。

还有很多 ThoughtWorker 和社区朋友帮助我们从技术层面和语言层面进行修改。他们有的是 Angular 专家，有的是新手，给了我们比较全面的反馈。能将枯燥、乏味的技术平易近人地展现在这本书中，一定要感谢他们所作出的奉献。他们是（排名不分先后）：冯尔东、朱本威、李科伟、杨琛、彭琰、叶志敏、ng 群 as。

还要感谢 Angular 中文社区 QQ 群和关注“双狼说”微信号的网友们，是你们的鼓励给

了我们写作的信心和动力！

雪狼的感恩

开始写书的时候，刚刚认识我的女友娜娜，今天，我们即将走进婚姻的殿堂。我这样一个负情商的程序员，生活有多么枯燥乏味，不问可知。感谢你点亮了我的人生。你的好，我永远记得！

能专注开发 17 年，要感谢我父母和弟弟的支持。人到中年，本应是最纠结的时代，特别是我这样的前“单身狗”。我无法经常回家，是弟弟经常回去探望父母。父母的乐观与健康，让我可以心无旁骛地工作。你们的好，我永远记得！

能走入软件开发这一行，要感谢我的伯乐何战涛和王勇的帮助。还记得那个沉默而不自信的“小汪”吗？当初，他什么也不会，犯过很多错误；如今，他在尽力为别人的职业生涯提供帮助。你们的好，我永远记得！

在这 17 年间，我尝试过很多角色，从写文档、测试到小公司的 CTO，走过了丰富多彩的人生。特别需要感谢的是林先生和余姐等前同事，我们追随林先生走过 8 年创业之旅，这一过程让我具备了更开阔的视野和更坚韧的性格。虽然因为生活压力不得不离开，但，你们的好，我永远记得！

最后，感谢 ThoughtWorks！作为“敏捷”的倡导者，ThoughtWorks 处处体现着敏捷思想，这是一个把“敏捷”变成“文化”并渗透到骨子里的公司。这是一个很“奇葩”的公司。这一点从“全球 CEO 和中国区程序员撞衫”事件就可见一斑。这是一个很“简单”的公司。引用我们一位 HR 的说法：“进了 ThoughtWorks，我感觉自己的情商都下降了！”这是一个很“技术”的公司。每年两期的技术雷达都来自全球近 3000 名工程师的实践总结。这里几乎会涉足每一项前沿技术：大数据、React、Scala 等。仅以 Angular 为例，我们在工程实践中使用它是在 4 年前，那时候 Angular 还是 0.6 版呢。

还差一句话就变成招聘软文了，索性补上吧：ThoughtWorks，你的好，进来才知道！

破狼的感恩

在写作本书之际，我作为 ThoughtWorks 高级敏捷咨询师、架构师，因为项目曾辗转多地，没有太多时间投入到这次的写作之中。在此之前，已经有很多的出版社联系我写本国人的深度解析 Angular 的书籍，但都被我婉言拒绝了。直到雪狼告诉我他希望写一本关于 Angular 的书籍的时候，恰巧吴怡编辑也跟我提起了写书的事。这次我们应承下来了。在这里首先感谢吴怡的知遇之情和写书过程之中的包容与耐心。还有雪狼大叔的最终推动。

我是一个具有承诺“强迫症”的人，一旦应承下来的事情，我就会尽自己最大的努力把它做好。特别在快截稿的几个月中，每天写作到凌晨 1 ~ 2 点，次日还需要准备与客户 7:00 的站会。和老婆结婚已经一年多了，可是由于工作原因，我也出差在外一年多了。加上写作此书的时候，连陪伴父母和老婆的时间也被我挤出来写作本书了。所以在此，首先要感谢我的父母和老婆，感谢你们的支持和包容，在这本书的背后也包含着你们对我的一份宝贵的爱。爸妈、老婆：我也爱你们！

还要感谢我大学的导师刘继光老师和柳翠寅老师，是你们让我学到了软件开发技能，以及帮我寻找到一份实习的机会，给了我比别人多 3 年的实战机会，因此我才能独自闯荡成都这座大城市，并开始了软件开发之路。还有你们对知识的追求和坚强的毅力，深深感染了我。至此坚持了 6 年多的博客写作和回馈开源社区，因此才有了本书的顺利完成。你们是我一辈子的良师益友，谢谢你们的付出！

同时还要感谢我在 ThoughtWorks 的同事们。是我的 Sponsor 张逸一直鼓励我写书，是熊节每年的阅读数量激励我更加坚定地持续获取更多的知识来武装自己。还有很多的 twer，同样在这里感谢你们长久以来对我的帮助和鼓励。

我常常告诉自己：要么读书，要么旅行，身体和心灵总有一个在路上。与君共勉。

彭洪伟的感恩

首先，我要感谢我父母！即使是在家境最困难的时候，他们也时时刻刻鼓励我、支持我，让我能坚持走自己的路。

我第一次接触到计算机是在 2003 年。那时刚上初中，正是 CS 和传奇火爆的时候，想自己申请一个 QQ 号都感觉很困难。直到 2006 年初中毕业，莫名其妙地开始了信息学奥林匹克竞赛的培训之路，用 C 语言写出了人生的第一行 Hello World 代码。

感谢张宗弋老师的悉心教导和高中三年对 C 语言、算法、图论、数论的培训，还记得大家都叫您小弋（误读作：Ge）子。是您激发了我的“程序员梦”。以至于后来在报志愿的时候，第一志愿写的都是计算机信息科学与技术，虽然那时候并不知道这个专业是干什么的。

在大学里，特别要感谢的是邓芳老师，是您鼓励我们，除了学好课堂上的东西，更要积极去探索自己感兴趣的東西。您“授人以鱼，不如授人以渔”的教学思想，让我逐渐养成了自学的习惯，至今让我受益匪浅。后来，您帮我引荐了两位让我受益良多的导师：陈宇副教授和王海军副教授。是他们把我带到了物联网的领域，让我对计算机的领域产生了更浓厚的兴趣。三位老师的指引，让我接触到了更多课堂上没有的东西，终于有幸加入

ThoughtWorks 这个大家庭中。

在这里，我认识了有黑客范儿的马伟和段子手 Jojo（周哲武），是他们给了我建设性的意见，让我感觉到了 ThoughtWorks 的不同。不得不提的还有 Jojo 帮我推荐的 Sponsor——破狼，是他不断地在开源的世界里给我指出新机会，鼓励我大胆尝试。不然我也不能坚持翻译完 Scala 构建工具 SBT：<http://www.scala-sbt.org/0.13/tutorial/zh-cn/index.html> 的使用文档。如果没有破狼的引荐，我也不会认识前端牛人雪狼和参与本书的编写。

最后，如果没有未婚妻韩盼盼的爱、支持与鼓励，我也难以完成这么多富有挑战性的工作。谢谢，我爱你！



Contents 目 录

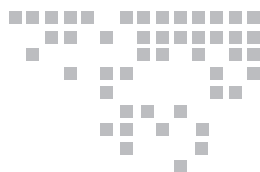
序	1.5.2 实现过滤功能	40
前 言	1.5.3 实现分页功能	42
	1.5.4 实现主题树	44
	1.5.5 实现递归主题树	56
	1.5.6 实现“查看详情”功能	58
第 1 章 从实战开始	1.6 实现 AOP 功能	59
1.1 环境准备	1.6.1 实现登录功能	60
1.2 需求分析与迭代计划	1.6.2 实现对话框	65
1.3 创建项目	1.6.3 实现错误处理功能	67
1.3.1 Yeoman	1.7 实战小结	68
1.3.2 FrontJet		
1.4 实现第一个页面：注册	第 2 章 概念介绍	70
1.4.1 约定优于配置	2.1 什么是 UI	70
1.4.2 定义路由	2.2 模块	71
1.4.3 把后端程序跑起来	2.3 作用域	72
1.4.4 连接后端程序	2.4 控制器	73
1.4.5 添加验证器	2.5 视图	74
1.4.6 “错误信息提示”指令	2.6 指令	75
1.4.7 用过滤器生成用户友好的提示	2.6.1 组件型指令	76
信息	2.6.2 装饰器型指令	79
1.4.8 实现自定义验证规则	2.7 过滤器	81
1.4.9 实现图形验证码	2.8 路由	82
1.5 实现更多功能：主题		
1.5.1 实现主题列表		

2.9 服务	83	3.6.3 使用场景	125
2.9.1 服务	85	3.7 REST	127
2.9.2 工厂	86	3.7.1 REST 的六大要点	128
2.10 承诺	88	3.7.2 REST 的四个级别	130
2.11 消息	92	3.8 跨域	131
2.12 单元测试	93	3.8.1 同源策略与跨域	131
2.12.1 MOCK 的使用方式	94	3.8.2 如何解决跨域问题	132
2.12.2 测试工具与断言库	95	3.9 前端安全技术	133
2.13 端到端测试	96	3.9.1 前端攻击的基本原理和类型	133
第 3 章 背后的原理	98	3.9.2 前端安全与前后端分工	136
3.1 Angular 中的 MVVM 模式	98	3.9.3 移动时代的特殊挑战	137
3.2 Angular 启动过程	102	3.9.4 安全无止境	138
3.3 依赖注入	106	第 4 章 最佳实践	140
3.3.1 什么是依赖注入	106	4.1 调整开发协作流程	140
3.3.2 如何在 JavaScript 中实现 DI	107	4.2 前后端分离部署	143
3.3.3 Angular 中的 DI	108	4.3 样式中心页	144
3.3.4 DI 与 minify	109	4.4 CSS 的扩展语言与架构	145
3.4 脏检查机制	110	4.5 HTML 的表意性	146
3.4.1 浏览器事件循环和 Angular 的 MVW	110	4.6 table, 天使还是魔鬼	148
3.4.2 Angular 中的 \$watch 函数	111	4.7 测试什么? 怎么测?	150
3.4.3 Angular 中的 \$digest 函数	113	4.7.1 准备工作	150
3.4.4 Angular 中的 \$apply	116	4.7.2 如何测试 Controller	151
3.5 指令的生命周期	117	4.7.3 如何测试 Service	151
3.5.1 Injecting	118	4.7.4 如何测试 Filter	152
3.5.2 compile 和 link 过程	120	4.7.5 如何测试组件型指令	152
3.6 Angular 中的 \$parse、\$eval 和 \$observe、\$watch	122	4.7.6 如何测试装饰器型指令	153
3.6.1 \$parse 和 \$eval	122	4.7.7 如何测试网络请求	153
3.6.2 \$observe 和 \$watch	124	4.7.8 如何测试 setTimeout 类功能	153
		4.7.9 如何 Mock Service	154
		4.8 如何设计友好的 REST API	155

4.8.1	URI	155	5.3	在非独立作用域指令中实现 scope 绑定	185
4.8.2	资源拆分	155	5.4	表单验证错误信息显示	186
4.8.3	资源命名	155	5.5	Angular 中的 AOP 机制	187
4.8.4	方法	156	5.5.1	拦截器案例	188
4.8.5	返回值	157	5.5.2	拦截器源码分析	192
4.8.6	综合案例: 分页 API	159	5.5.3	Angular 中的装饰器	195
4.9	使用 controller as vm 方式	160	5.5.4	Angular 装饰器源码分析	197
4.9.1	源码分析	161	5.6	Ajax 请求和响应数据的转换	198
4.9.2	推荐用法和优势	161	5.6.1	兼容老式 API	198
4.9.3	路由中的 controller as 语法	162	5.6.2	Ajax 请求配置的源码分析	201
4.9.4	指令中的 controller as 语法	163	5.7	在代码中注入 Filter	205
4.10	移除不必要的 \$watch	163	5.7.1	复用指定 Filter	205
4.10.1	双向绑定和 watchers 函数	164	5.7.2	重用多个 Filter 案例	206
4.10.2	其他指令中的 watchers 函数	166	5.7.3	Filter 源码分析	207
4.10.3	慎用 \$watch 和及时销毁	167	5.8	防止 Angular 表达式闪烁	208
4.10.4	one-time 绑定	168	5.8.1	表达式闪烁解决方案	208
4.10.5	滚屏加载	170	5.8.2	ngCloak 源码分析	208
4.10.6	其他	171	5.8.3	最佳实践	209
4.11	总是用 ng-model 作为输出	172	5.9	实现前端权限控制	209
4.12	用打包代替动态加载	173	5.9.1	事件方案	210
4.13	引入 Angular-hint	173	5.9.2	resolve 方案	211
4.13.1	通过 batarang 插件使用 angular-hint	174	5.10	依赖注入——\$injector	214
4.13.2	手动集成 angular-hint	174	5.10.1	\$injector 的创建	214
4.13.3	Module hints	175	5.10.2	\$injector 注入方式	215
4.13.4	Controller hints	176	5.10.3	\$injector 的妙用	217
4.13.5	Directive hints	176	5.11	在指令中让使用者自定义模板	219
第 5 章	Angular 开发技巧	178	5.12	跨多个节点的 ng-if 或 ng-repeat	223
5.1	\$timeout 的妙用	178	5.13	阻止事件冒泡和浏览器默认行为	224
5.2	ngTemplate 寄宿方式	182	5.14	动态绑定 HTML	226

第 6 章 Angular 常见的“坑”229	
6.1 module 函数的声明和获取重载.....229	
6.2 ngModel 绑定值不更改.....232	
6.2.1 验证引起的 model 值不显示233	
6.2.2 原型链继承问题235	
6.3 指令不生效239	
6.4 Angular 中锚点的使用240	
6.5 ngRepeat 验证失效.....241	
6.5.1 简单的验证显示242	
6.5.2 复杂的验证显示242	
6.6 有些指令需要唯一的根节点243	
6.7 指令优先级 -Priority243	
6.8 ngRepeat 报重复内容错误244	
6.9 单元测试中 promise 不触发245	
第 7 章 编码规范247	
7.1 目录结构.....248	
7.1.1 按照类型优先、业务功能其次 的组织方式248	
7.1.2 按照业务功能优先、类型其次 的组织方式249	
7.2 模块组织.....250	
7.2.1 命名250	
7.2.2 Module 声明.....250	
7.2.3 依赖声明251	
7.2.4 Module 组件声明.....251	
7.3 控制器252	
7.3.1 命名252	
7.3.2 ControllerAs vm 声明252	
7.3.3 初始化数据253	
7.3.4 DOM 操作253	
7.3.5 依赖的声明253	
7.3.6 精简控制器逻辑254	
7.3.7 禁止用 \$rootScope 传递数据255	
7.3.8 格式化显示逻辑255	
7.3.9 Resolve.....255	
7.4 服务.....256	
7.4.1 命名256	
7.4.2 代码复用256	
7.4.3 使用场景256	
7.4.4 Service 返回值257	
7.4.5 缓存不变数据257	
7.4.6 RESTful257	
7.5 过滤器258	
7.5.1 命名258	
7.5.2 重用已有 Filter.....258	
7.5.3 禁止复杂的 Filter.....258	
7.6 指令.....259	
7.6.1 命名259	
7.6.2 Template 声明.....259	
7.6.3 link 函数的 scope 参数命名259	
7.6.4 pre-link 和 post-link.....260	
7.6.5 DOM 操作260	
7.6.6 Directive 分类.....260	
7.6.7 Directive 不是封装 jQuery 代码 “天堂”260	
7.6.8 自动回收261	
7.7 模板.....261	
7.7.1 表达式绑定261	
7.7.2 Src、Href 问题.....261	
7.7.3 Class 优于 Style.....262	
7.8 工具.....262	

7.9 其他	264	第 9 章 杂项知识	282
7.9.1 内置 \$ 服务替代原生服务	264	9.1 Angular 2.0	282
7.9.2 Promise 解决回调地狱	264	9.2 SEO	284
7.9.3 减少 \$watch	265	9.3 IE 兼容性	287
7.9.4 TDD	265	9.3.1 问题概述	287
第 8 章 工具	267	9.3.2 问题分类	288
8.1 WebStorm 与 IntelliJ	267	9.4 访问统计	292
8.2 Chrome	269	9.5 响应式布局	293
8.3 Gulp	273	9.6 国际化	294
8.4 Swagger	274	9.7 动画	296
8.4.1 前后端分离	274	9.7.1 CSS 动画	296
8.4.2 Swagger	275	9.7.2 JavaScript 动画	297
8.4.3 契约测试	277	9.8 手机版开发	298
8.5 TSD	277	9.8.1 Hybrid 应用	298
8.6 Postman	280	9.8.2 Ionic	300
8.6.1 安装	280	附录 A 相关资源	301
8.6.2 功能介绍	280	后记 提问的智慧	318



从实战开始

Brooks 在《人月神话》中有一句著名的论断：“没有银弹”。Angular 也不例外，事实上，我在 17 年的软件开发过程中，还从没有遇到过可称为“银弹”的技术。任何一个成功的项目都需要多种技术的配合，经过需求、设计、编码、测试、项目组织等过程来进行，任何一方面的短板都可能限制整个项目的成就。本书不可能面面俱到，但这些过程如此重要，以至于可能成为在项目中成功应用 Angular 的绊脚石。因此我们先从实战出发，通过一个简单的例子讲解这些过程。

本章将由浅入深地讲解从环境准备、需求分析与迭代计划、创建项目、写页面，到对登录、错误处理等进行面向切面编程（AOP）。本书可能对初学者有一定难度，建议初学者先到 [ngnice \(http://www.ngnice.com/\)](http://www.ngnice.com/) 了解 Angular 基础知识。

1.1 环境准备

进行开发的第一步是准备开发工具。对于用惯了 IDE 的程序员来说，可能需要适应一下 IDE 配合命令行的模式，不过最终你会爱上命令行模式的快速和简洁。

我们将要使用的环境如下。

1. Node

Node 全称是 Node.js，它是一个让 JavaScript 访问各种本地 API 和网络 API 的运行环境，在本书中，将大量使用基于 Node 的模块和工具。

Node 的安装非常简单，如果你使用 Linux/Mac 操作系统，建议从 <https://github.com/creationix/nvm> 下载；如果你使用 Windows 操作系统，建议从 Nodejs 官网下载：<http://nodejs.org/>。

受互联网等因素的影响，Linux/Mac 版的安装可能会遇到问题，如果使用过程中遇到问题，也可以从 Nodejs 官网下载，但是要用这种安装方式将迫使你使用 root 权限，后面可能经常需要输入登录密码，从此以后，你需要 root 权限才能安装某些第三方 Node 包，相应的，`npm ...` 命令也要改为 `sudo npm ...` 命令。

2. cnpm

`npm` 是 Node 自带的包管理器，它的中心服务器架设在海外，受到互联网等因素的影响，有时下载速度会非常慢，甚至部分包完全无法下载。我们可以使用阿里在国内架设的 CDN 来解决此问题。阿里的 CDN 提供了一个独立的包管理命令：`cnpm install ...`，不过在使用它之前需要先安装：`npm install -g cnpm --registry=https://registry.npm.taobao.org`。今后所有需要出现 `npm` 命令的地方，都可以替换为 `cnpm` 命令。假设你已经安装过 `cnpm` 命令，所以本书后面出现的 `npm` 命令也会替换为 `cnpm` 命令。

3. Java

Angular 的前端开发并不需要使用 Java，但是 WebStorm 等 IDE 是基于 Java 开发的，本书范例中的后端服务器也是基于 Java 开发的。如果需要，可以到 <http://www.oracle.com/technetwork/java/javase/downloads/> 下载并安装 Java。

4. IntelliJ

本书中将使用 IntelliJ 作为演示用 IDE。这是 JetBrains 公司出品的软件，既能写前端，也能写后端，默认支持 Java 语言和前端技术栈，也可以通过插件支持更多种语言。它还有一个专门面向前端开发的精简版本，叫作 WebStorm。它的安装方式都有官方说明，在此不再详述。

JetBrains 的软件是收费的，不过它带来的便利确实值得你投资。此外，Eclipse 和 Visual Studio 也都是不错的选择。

5. IntelliJ 的 AngularJS 插件

IntelliJ 的 AngularJS 插件是个非常实用的插件。它可以帮你检查模板中使用的官方指令和自定义指令的语法，并且支持按组合键 `Ctrl+B`（Windows 系统）/`Cmd-B`（Mac 系统）进行跳转。

6. Git

本书的源码将全部通过 Git 发布在 GitHub 上，所以，你有必要安装一个 Git 工具。而且，Git 的作用远不止于管理本书的源码，在做其他开发的过程中也很有用，甚至本书的写稿也是通过 Git 来协调多位作者和编辑之间的合作的。

对于 Linux 系统，主流的版本都可以使用其内置的包管理器来安装，比如 Ubuntu 下的 `sudo apt-get install git`；对于 Mac 系统，Xcode 内置了 Git，也可以通过 brew 工具安装最新版，如：`brew install git`；而 Windows 下比较特殊，由于 Git 没有原生的 Windows 编译版本，所以需要下载一个第三方版本 <https://msysgit.github.io/>。

7. cygwin

Windows 不是前端开发的理想环境，如果有条件，最好使用 Linux 或 Mac。如果确实要在 Windows 下操作，那么请先安装 cygwin。cygwin 是个在 Windows 下面的 Linux 命令模拟器。本书中列出的绝大多数命令都需要在 cygwin 下执行。当然，如果你熟悉 Linux 命令和 Windows 命令的对应关系，也可以在 Windows 的 CMD 窗口中开发。

不过，最好的方式是用 VirtualBox 装一个 Linux 虚拟机，在虚拟机中进行这些尝试。

8. 开发指南与 API

由于受互联网等因素的影响，部分用户可能无法正常访问官方网站（angularjs.org），也就无法正常查阅开发指南和 API，这将成为学习 Angular 的障碍。为了解决此问题，我们在国内架设了一个网站：www.ngnice.com，这里有我们组织翻译的开发指南（guide）以及 API 的英文版。

1.2 需求分析与迭代计划

本案例是一个教学项目，它是一个小型的系统，其目标是作为本书的读者交流区，你将看到它如何从简陋走向完善。

它完全开源，有兴趣的读者也可以实际参与到它的功能扩展中。显然，如果我把代码都粘贴到本书中，编辑大人会跟我拼命的。所以我在本书中只摘录并讲解其中适合用来示范技术的功能。

完整的代码会作为一个开源项目托管在 GitHub 上：<https://github.com/ng-nice/code>。想要获取源码，请执行命令：`git clone git@github.com:ng-nice/code.git`，它会把项目源码下载到当前目录的 code 子目录下，里面会有 README.md 文件指引你如何运行本项目，以及如何参与开发。

1. 系统隐喻

作为项目的第一步，我们会建立一个“系统隐喻”，也就是描述一下我们的系统类似于哪些现有系统，从而让项目组成员更形象地记住目标，也让客户更容易理解目标。当然，世上没有完全相同的系统，描述系统隐喻时要重点描述其与现有系统的不同点。

系统隐喻：一个迷你的论坛系统，侧重交流而不是宣传。与传统论坛 / 留言板系统相比，本系统更加重视对读者反馈意见的收集和自动化分析。

2. 业务目标

系统隐喻过于粗略，接下来我们要定义一个业务目标，这是一个更明确的功能列表。但是，要注意，业务目标不仅仅包含“要做什么”，而且要包含“不做什么”。软件开发中有一个大敌，叫作“需求蔓延”。在这个阶段定义“不做什么”就是为了防范“需求蔓延”。

要做什么

(1) 非功能需求。

- ☐ 每个系统只支持一本书。
- ☐ 支持多个作者（包括志愿者），数量预估在 10 人以下，不会超过百人。
- ☐ 支持多个读者，数量预估在千人左右，不会超过一万人。
- ☐ 防范 XSS 等常见前端攻击。
- ☐ 防范 SQL 注入、DDoS 等后端攻击。
- ☐ 使用公认安全的后端权限框架，以免出现典型安全问题。
- ☐ 面向读者的功能需求。
- ☐ 允许读者注册。
- ☐ 允许读者登录。
- ☐ 不允许匿名发帖或 +1。
- ☐ 对主题进行树状显示。
- ☐ 读者可以发布勘误，作者可以接受“勘误”，或标为“重复”。
- ☐ 读者可以发布问题，作者可以答复问题。
- ☐ 读者可以发布建议，作者可以接受“建议”，或标为“重复”。
- ☐ 其他读者可以对问题或建议进行 +1 操作，提升作者的关注度。
- ☐ 读者可以通过第三方搜索引擎搜索。
- ☐ 系统可以自动整理 FAQ，供读者阅读。

(2) 面向作者的功能需求：

- ☐ 作者可以答复问题。

- ❑ 作者可以删除不当主题。这里的删除不是真正的删除，作者仍然可以看到并恢复。
- ❑ 作者可以发布“读者调查”。
- ❑ 作者可以对特定主题进行高亮显示。
- ❑ 系统会自动从内容中提取一批关键字，同时作者也可以手动为文章设置一批关键字。
- ❑ 作者可以进行数据库搜索，不做限制。
- ❑ 支持上传图片和附件，图片和附件最大不超过 10MB。
- ❑ 支持提交代码和显式：HTML/CSS/JavaScript/Java，不需要支持其他语言。
- ❑ 作者可以统计出所有提供过建议、勘误的读者，将名字列入下一版中的“贡献名单”。
- ❑ 出于安全原因，读者上传的非图片类附件需要作者审核后才能显示。

不做什么

- ❑ 不做多级权限，只区分作者和读者。
- ❑ 不支持手动配置权限，硬编码权限对应的操作。
- ❑ 不支持读者进行数据库搜索，但对读者的搜索关键字进行统计。
- ❑ 出于版权原因，不支持 QQ 等表情包，只支持普通的符号表情。
- ❑ 默认读者是善意的，不对内容进行预先过滤，只支持事后删除。
- ❑ 不做专门的回收站，而是让作者就地删除和恢复。
- ❑ 不做全面的视觉美化，但注重“操作友好性”。

3. 需求分析

在真正的项目组中，会有一个集体估算“功能点 / 复杂度”的过程，但是作为单人开发者，就只好“独裁”了。

不过，比估算“功能点 / 复杂度”的具体数量更重要的是拆分任务、估算风险，并最终决定优先级。在排定优先级之后，我们可以把整个项目划分成多个阶段，每个阶段都可以发布可用的产品版本，这将带来多方面的好处：

- ❑ 产品可以尽早推出，尽早获得回报。
- ❑ 可以尽早为客户方建立信心，而获得客户方的信任，这对软件项目的成功至关重要。
- ❑ 可以尽早发现并控制风险。
- ❑ 可以给项目组成员一段休养时间，防止疲劳作战。
- ❑ 可以留出一些重构和回顾的时间，抽取复用模块。

我们将这些“阶段”称为“基线”（Baseline）。它的英文原词非常形象：多个项目阶段就像是一个个台阶，每完成一个阶段，我们就迈上一个台阶，站在一个稳固的基础（Base）上。有了基线，我们就可以对各期的目标进行隔离，即使下一期目标达不到，我们也随时有一个可用的版本。

这种交付方式称为“迭代式开发”，更形象的说法叫作“小步快跑”。

基线的粒度取决于所用的开发方式，对于敏捷开发，通常基线会划分到一周左右的粒度，即使是传统开发方式，也可以向这个方向努力。基线的内在形式是一个表格，通常包括编号、名称、价值、难度、风险、工作量、优先级等几项，建议用五级评估。其中：

价值是指此项功能对业务的重要性。这往往来自产品经理或客户代表。对于敏捷团队，还要让全体成员理解它的业务价值。要注意，业务价值是有时效性的，当前阶段迫切需要的功能会有更高的价值。随着业务的发展，这些业务价值也会相应变化。

难度是指根据项目组现有技术积累评估出的难易程度。比如“能处理每秒十万人同时下订单”对当前团队来说可能难度较大。难度主要决定人员分工，特别是人员能力。

风险是指不确定性。比如，假设我从来没集成过富文本编辑器，那么虽然我明知它应该很容易，但仍然是一个较高的风险。我可能评估难度为 1，风险为 3。再比如，我们的某项功能要和一个不够成熟的第三方系统对接，它就会具有很高的风险，可能评估难度为 3，风险为 5。风险决定项目安排，高风险的任务往往需要先开展 SPIKE（技术预研），或者提前安排资源保障或制定备用方案。

工作量是指在一切顺利的情况下需要的时间。比如有些系统对视觉效果的要求非常严格，那么它虽然既没有难度，也没有风险，但是具有很高的工作量。工作量往往影响人员的分工，特别是人员数量。

优先级是前面几项的综合。一般来讲，排定优先级的依据是“性价比”，这里的“性”是指业务价值，“价”则根据难度、风险、工作量进行综合评定；如果时间紧迫，那么高风险的项目会被安排较低的优先级，也就是尽量推到后面的版本中实现；另一方面，如果一项工作不完成，后续的功能就无法实现，它也需要较高的优先级，比如读者注册。这通常由项目经理最终决定，敏捷团队中则往往由客户方和开发方共同商定。

项目管理中有一个误区，就是把本来不是很紧迫的任务故意排得很紧迫，希望以此来提高开发人员的“效率”，但是这往往适得其反——高收益高风险的需求会被缺乏安全感的项目组推到后续版本中。

当然，上面这几项也不是必须全都列出和准确估值的，可以根据不同的项目进行定制，比如团队沟通顺畅，或组里都是能力比较强的成员，那么难度、风险、工作量可以打包评

估成一个“复杂度”项或干脆叫“点”。

基线的外在形式则比较多样，既可能是一个 Word 文档，也可能是一个 HTML 或 Markdown 文件，在敏捷开发中，则常常表现为一组故事卡（Story card）。至于采用哪种形式，取决于项目的组织架构和外在条件，没有统一的规定。只要让每个人随时都清楚自己的目标就可以了。比如我在个人开发中就把这些任务加到 Evernote 中，并且加上复选框，完成一项勾选一项，任务和进展一目了然。

需求评估表见表 1-1。

表 1-1 需求评估表

编号	名称	价值	难度	风险	工作量	优先级	备注
10.	读者可以注册	5	2	1	2	5	
20.	认证与权限体系（Spring security）	5	2	2	2	5	
30.	对主题进行树状显示	5	2	1	2	5	
40.	读者可以发布问题，作者可以答复问题	5	2	1	2	5	
50.	读者可以发布勘误，作者可以接受“勘误”，或标为“重复”	4	2	1	2	4	早期可以作为普通问题发布
60.	读者可以发布建议，作者可以接受“建议”，或标为“重复”	4	2	1	2	4	早期可以作为普通问题发布
70.	其他读者可以对问题或建议进行 +1 操作，提升作者的关注度	5	2	1	2	5	
80.	读者可以通过第三方搜索引擎搜索	3	1	3	2	3	早期数据量小，没必要提供读者的高级搜索功能
90.	系统可以自动整理 FAQ，供读者阅读	2	4	2	3	2	算法较复杂，早期数据量小，没必要整理
100.	作者可以答复问题	5	2	2	2	5	
110.	作者可以删除不当主题，作者仍然可以看到并恢复	4	2	1	2	4	
120.	作者可以发布“读者调查”	3	4	3	5	3	UI 和模型较复杂，早期可以使用金数据等第三方工具进行调查
130.	作者可以对特定主题进行高亮显示	4	2	1	2	4	
140.	作者可以设定关键字，系统自动对内容设定关键字	3	2	1	1	2	

(续)

编号	名称	价值	难度	风险	工作量	优先级	备注
150.	作者可以进行数据库搜索, 不做限制	4	2	2	2	4	访问量可控, 不用考虑性能问题, 此功能可为其他操作提供数据
160.	支持上传图片和附件, 图片和附件最大不超过 10MB	3	2	4	2	2	自己实现容易被攻击, 使用第三方服务需要考虑成本和集成风险
170.	支持提交代码和显示代码: HTML/CSS/JavaScript/Java	4	2	2	2	4	使用第三方库, 曾经在其他项目实现过
180.	作者可以统计出所有提供过建议、勘误的读者	3	2	1	3	4	成熟技术, 有一些显示方面的工作量

在列出需求评估表之后, 可以做出一个简要的项目基线了。

原则上, 我们把优先级为 5 的需求划为第一条基线, 部分优先级为 4 的需求也可以划为第一条基线。其他需求则留待未来实现, 但不用提前决定它们到底属于二期还是三期。

于是我们得出了第一条基线, 见表 1-2。

表 1-2 第一条基线

编号	名称	价值	难度	风险	工作量	优先级	备注
10.	读者可以注册	5	2	1	2	5	
20.	认证与权限体系 (Spring security)	5	2	2	2	5	
30.	对主题进行树状显示	5	2	1	2	5	
40.	读者可以发布问题, 作者可以答复问题	5	2	1	2	5	
50.	读者可以发布勘误, 作者可以接受“勘误”, 或标为“重复”	4	2	1	2	4	早期可以作为普通问题发布
70.	其他读者可以对问题或建议进行 +1 操作, 提升作者的关注度	5	2	1	2	5	
100.	作者可以答复问题	5	2	2	2	5	
110.	作者可以删除不当主题, 作者仍然可以看到并恢复	4	2	1	2	4	
130.	作者可以对特定主题进行高亮显示	4	2	1	2	4	
150.	作者可以进行数据库搜索, 不做限制	4	2	2	2	4	访问量可控, 不用考虑性能问题, 此功能可为其他操作提供数据

(续)

编号	名称	价值	难度	风险	工作量	优先级	备注
170.	支持提交代码和显示代码: HTML/CSS/JavaScript/Java	4	2	2	2	4	使用第三方库, 曾经在其他项目实现过
180.	作者可以统计出所有提供过建议、勘误的读者	3	2	1	3	4	成熟技术, 有一些显示方面的工作量

4. 定义导航图

需求分析完毕, 我们需要定义一下导航图 (Site map)。导航图主要用来体现页面之间的跳转关系。定义完导航图, 就可以大致了解整个网站有多少个页面, 以及各个页面的大体复杂度。

导航图可以有多种形式。如果需要演示给客户或者老板, 那么可以安装一个 FreeMind 软件来制作脑图。这个软件是开源的, 并且支持各主流操作系统。如果要更高级的效果, 也可以使用 Sketch (Mac 版的收费软件) 等软件制作线框图。如果只是用作开发组内部交流, 那么可以直接画在或贴在白板上, 让每个人一抬头就能看到, 这是敏捷开发中常用的方式。如果涉及远程协作开发, 或者团队沟通环境不那么顺畅, 那么也可以直接写成 Markdown 文件, 并纳入版本管理。

我们先制作一个 Markdown 格式的导航图。

- 1. 首页
 - 1. 发布问题
 - 1. 发布勘误
 - 1. 帖子 (问题 / 勘误) 列表 / 树
 - 1. +1 按钮
 - 1. 帖子详情
 - 1. 删除按钮 (限作者)
 - 1. 高亮按钮 (限作者)
 - 1. 搜索框 (限作者)
 - 1. 统计 (限作者)
 - 1. 帖子详情 (含主贴、跟帖)
 - 1. 回复
 - 1. +1 按钮
 - 1. 删除本帖 (限作者)
 - 1. 高亮本帖 (限作者)

可以看到，从导航结构来讲，一期需求是非常简单的。这也是很多项目的常态，通过第一期的成功来树立开发方和客户方的信心，后面的开发就会顺利很多。这也是你凭借自己的专业性来对客户负责的表现。

如果第一期的需求很复杂，就要提高警惕了，这可能是项目管理不良或客户沟通不畅的信号。除此之外，一期需求过于复杂还容易导致过度设计，引入不必要的复杂性，给将来的重构工作带来困扰。

可以看出，导航图中有一些项其实指向同一功能，但是我们不用在这里消除重复，做导航图的目的是更直观地了解系统概貌，我们会在后面的阶段中消除这些重复。

5. 第一个迭代

定义完一期需求，我们就要开始制订项目计划了。通过几次迭代方式来进行：

我们不需要制订一个庞大的计划，只需要划分出第一个迭代的任务就可以了。通常，每个迭代都不应该超过当前项目组一周的工作量。注意，必须根据“当前”项目组的能力进行估算。对于相互之间还不太了解的项目组，宁可使用比较保守的估计。确保第一个迭代的成功无论在维护士气还是客户关系上都非常重要。

把哪些功能放进第一个迭代也很有讲究。划分方式有纵向划分和横向划分两种。

纵向划分方式是一个功能一个功能完成，每开发一个功能就把它完全实现，做完的功能从数据库到用户体验都基本达到发布级质量。这种方式下，通常会把“被依赖”的功能安排在前面，比如用户注册、登录等。

这种方式的优点是只做已经明确的需求，减少无用功。进展到一定阶段后随时可以对外发布版本。这是最简单明了的方式，可能也是很多人正在实际使用的方式，不再展开讲。

横向划分方式是先搭一个粗糙的骨架（会走的骷髅），它难看、缺少细节（如数据有效性校验），但是能在第一时间把大部分流程串起来，甚至可以在第一个迭代进行中就邀请用户参与。在这个阶段中，UX（用户体验设计师）、BA（业务分析师）等可以并行工作：做样板页、细化用户故事。一部分没有任务的研发也可以同步进行 **SPIKE**（技术预研）工作。

这种方式适用于对需求和业务知识还不太熟悉的开发团队，也适用于客户方的需求本身就不够明确的情况，基本上符合从原型演化到正式系统的节奏。对于配合默契的成熟团队，这种方式“高度并行化”的特点可以大幅度提高研发进度。

从表面上看，横向划分方式有点像瀑布模型，但其实并非如此。横向划分方式不是用文档去记载需求，而是用代码去演示需求，并且在尽可能早的时间点上把最终使用者引入开发过程中。

如果和客户沟通比较顺畅，那么很多表面看来应该放在第一期的需求可能会被推到第

二期甚至砍掉，这样有助于控制需求的范围。同时，由于其“粗糙”的特点，其开发代价非常小，就算写出来再砍掉也没什么可惜的。

当然，在现实项目中，这两种方式往往是穿插进行的。

- ❑ 第一个迭代先做一个“会走的骷髅”，然后和客户沟通，同时 UX 并行设计样板页，BA 并行写用户故事的细节。
- ❑ 从第二个迭代开始，把客户认为最紧迫的页面实现到发布级质量，提交测试。端到端的“场景测试”也可以开始根据上个迭代完成的“会走的骷髅”来写了，但只需要先保障“乐观流程”(Happy path)的正确性。
- ❑ 从第三个迭代开始，就可以针对已经完成的用户故事写场景测试和进行用户化测试了。

就本书而言，由于需求简单而明确，所以就直接采用纵向划分方式了，而且这种方式也有利于教学。但这并不表示纵向划分优于横向划分，而是它更适合于当前场景。

作为第一个迭代，我们划定的范围是：

- 10. 注册
- 20. 登录（认证与权限体系）
- 30. 主题树

当然，这排不满一周的工作量，但是由于还要穿插写书工作，因此也是一个比较合理的划分。在教学上，它也能示范出足够多的 Angular 特性。

1.3 创建项目

接下来，我们要新建一个项目。传统的方式是使用 Yeoman 工具，它是基于 Node 的一个项目生成器引擎，但本书使用的是 FrontJet 方式，所以这里讲两个方式。

1.3.1 Yeoman

这节我们先简单讲讲 Yeoman。

首先用 `cnpm install -g yo` 命令来安装它。

Yeoman 只是个项目生成引擎，我们还需要安装一个 Angular 的项目模板，可以使用 `cnpm install -g generator-gulp-angular@0.8.1` 命令。为了让后续步骤和本书的描述保持一致，请完全按照刚才的命令行来安装指定版本，先不要使用新版本。建议等熟悉了此版本后再尝试。

这个生成器还依赖两个全局命令，也需要先行安装：`cpnm install -g gulp bower`。

接下来我们就可以创建项目了。先建立一个项目目录，比如：`mkdir ~/dev && cd ~/dev`，`mkdir book-forum-yo`，然后进入这个目录 `cd book-forum-yo`，再运行命令 `yo gulp-angular`。

这是一个交互式过程，Yeoman 会提出一系列问题供你选择，请跟随我的选项：

- ☐ Which version of Angular do you want? 这里是选择 Angular 版本，由于我们需要兼容 IE8，所以这里要选 1.2.x 项。按方向键把光标移到 1.2.x 项上，回车即可。
- ☐ Which Angular's modules would you want to have? 这里是选择 Angular 的子模块，具体的用途我后面会讲，直接回车即可。
- ☐ Would you need jQuery or perhaps Zepto? 这里是选择 jQuery 的版本，由于我们需要兼容 IE8，所以这里要选择 jQuery 1.x 版本。
- ☐ Would you like to use a REST resource library? 这里选择 `ngResource`，这是 Angular 内置的 REST API 访问库。另一个选项 `Restangular` 有点过度设计，不够简洁，所以这里不选择它。
- ☐ Would you like to use a router? 这里选择 `UI Router`，这是一个第三方库。另一个选项 `ngRoute` 是内置路由库，`UI Router` 用起来比 `ngRoute` 稍好一些。
- ☐ Which UI framework do you want? 这里选择 `Bootstrap`，这是 Twitter 开源的一个 CSS 库，也有一些 JavaScript 组件，它简洁大方，成熟度高，而且可以自由使用在商业项目中，所以作为首选。
- ☐ How do you want to implements your Bootstrap components? 这里选择 `Angular UI Bootstrap`，这是 `Angular-ui` 组推出的一个第三方 Angular 组件库，本项目中会使用到其中的一些指令。
- ☐ Which CSS preprocessor do you want? 这里选择 `Sass (Node)`。`Sass` 是一种 CSS 的增强语言，支持嵌套以及变量、循环等，最终要编译成 CSS，这里选择的是用 C 语言编写的一个 `Sass` 编译程序，相对 `Ruby` 的版本，它功能较少，但是执行速度要快很多，而这种差别在大中型项目中是相当显著的，所以宁可牺牲 `Sass` 的高级特性，也要选择它。
- ☐ Which JavaScript preprocessor do you want? 这里选择 `None`，也就是普通 JavaScript。考虑到一些读者可能不熟悉其他几种语言，所以为了便于示范，我们选择普通 JavaScript。
- ☐ Which html template engine would you want to have? 这里选择 `Jade`，事实上这里选

择什么都可以，因为为了便于示范，我们在这里不会使用 HTML 模板引擎，而是直接使用普通 HTML。

至此，选项的选择已经全部完成，系统会开始安装 bower 和 npm 包，前者用来安装前端组件，而后者用来安装开发环境。

不过，由于 GFW 的阻挡，npm 的某些包可能安装失败，因此我们要按 Ctrl+C 组合键阻止这个自动安装过程，而是自己手动安装，请用如下命令：

```
bower install
rm -fr node_modules
cnpm install
```

如果以前用过 cnpm 和 npm，那么可能会出现一些版本不匹配的警告。这些一般不会导致问题，不过如果要去掉这些警告，那么可以用如下命令清除一下缓存，并重新安装：

```
npm cache clean
cnpm cache clean
rm -fr node_modules
cnpm install
```

1.3.2 FrontJet

用 Yeoman/gulp-angular@0.8.1 方式创建的项目有一些缺点：

- ❑ 它会在当前目录下放入一个巨大的 node_modules 目录，但是这些文件只是给 gulp 工具集使用的。除了浪费很大空间外，它还具有很深的目录结构，容易导致 Windows 系统下的复制、剪切、移动等操作出错。
- ❑ 对中文的支持不够好——它引入 script 时没有加上 utf-8 选项。
- ❑ 目录结构设计不理想，这主要表现在它将 bower_components 放入 app 目录中，这将干扰“在目录中搜索”等操作。
- ❑ 选项过多，容易让新手困惑，这也阻碍了通过社区提问来解决问题，因为过多的选项让你很难简短地说清问题。
- ❑ 自动化程度不够。每个程序文件都需要自己手动加入 index.html 中。
- ❑ 工具部分不容易升级。gulp-angular 本身也会升级，但由于老版本已经嵌入项目中，所以要对它进行升级会比较困难，特别是在项目比较多的时候。
- ❑ 有 Bug，创建了新文件或者删除了文件时不会触发 reload。

针对这些缺点，我写了一个开源工具——FrontJet（前端喷气式引擎），它是我根据自己的一些项目经历逐步改造而成的。相对于 gulp-angular，它主要有以下亮点：

- ❑ 可独立安装、升级。
- ❑ 去掉了很多选项，直接选用经过实践检验的固定技术栈，简化创建过程。
- ❑ 自带一个种子工程，里面包含根据实战经验总结出来的目录结构和开发指南，可用于创建新工程。
- ❑ 自动注入项目中的 JavaScript 文件和 Scss 文件，引入 JavaScript 文件时会加上 `charset="utf-8"` 选项。
- ❑ 对文件进行增删改时都能正常触发 reload。
- ❑ 增加编译 Web font 的功能，即把一个 svg 文件放入 icons 目录，就会自动编译成 font 文件 (ttf、woff 等)，以及相应的 Scss 文件。
- ❑ 增加 Forks 功能，可生成针对不同操作系统的文件，开发服务器会根据浏览器所在的操作系统返回相应分支下的文件，这特别适合于手机版调试。
- ❑ 增加 Mock 功能，基于 node-restify 库，生成一个内置的 Mock 服务器，可在与真实服务端对接之前提供一个模拟服务器。这些 Mock 数据也会被自动用于单元测试。
- ❑ 增加内置的启动为 https 服务的功能，可用于排查 https 的特有问题。
- ❑ 增加针对特定 URL 的反向代理、模拟延迟功能。反向代理虽然在 gulp-angular 中也有实现，不过比较粗糙，需要修改 gulp 源码才能工作，我将其移到 fj.conf.js 中。模拟延迟则用于模拟真实环境中的网络延迟，以便设计更好的用户体验。
- ❑ 在 Linux/Mac 下增加了系统级错误提示框：当 FrontJet 编译过程中发现语法错误时，会通过系统本身的通知功能显示一个错误提示框，以免被忽略。

FrontJet 的安装非常简单，使用 `cnpm install -g fj` 即可。fj 是 FrontJet 的缩写，而且正好是键盘上的两个定位基准键，非常便于输入。安装完之后，即可在任何目录下使用 fj 命令，常用的命令见表 1-3。

表 1-3 常用命令

命 令	参 数	含 义
fj init	无	初始化，用于为第三方生成的目录创建一个 fj.conf.js 文件，使其可运行 FrontJet 命令
fj create [name]	name: 项目 (目录) 名称	在当前目录下创建一个名为 name 的新工程
fj serve [-s] [-p \$PORT]	-s: 是否用 https 方式启动 -p \$PORT: 启动在 \$PORT 端口下，默认为 5000	启动一个开发服务器。它会同时自动启动 TDD 模式
fj build [--ios] [--android]	--ios: 针对 IOS 系统编译分支版本 --android: 针对 Android 系统编译分支版本	编译出供发布的文件

还有一些不常用的命令见表 1-4。

表 1-4 不常用命令

命 令	功 能
fj help 或 fj	显示帮助信息
fj tdd	启动 TDD 模式，这种模式下，所有 JavaScript 文件（包括项目文件和单元测试文件）的变更都会自动执行相应的单元测试
fj ut	执行所有单元测试，然后退出
fj sass	主动编译一次 Sass 文件
fj coffee	主动编译一次 coffee 文件
fj ts	主动编译一次 TypeScript 文件
fj wireApp	自动注入 app 中的文件
fj wireBower	自动注入 bower 中的文件
fj clean	清理 .tmp 目录和 dist 目录

1. 使用 FrontJet 创建项目

使用 FrontJet 创建项目非常简单：

```
mkdir ~/dev && cd ~/dev
fj create BookForum
```

2. 启动开发服务器

现在就可以直接在 IDE 中打开它了。对于 IntelliJ/WebStorm 的最新版本，只要在“File|Open”菜单中选择 BookForum 目录即可。

如果 IDE 支持内嵌命令行窗口功能，如 IntelliJ/WebStorm 的 Terminal 窗口，那么建议打开它，并且在此处执行 fj serve 命令来启动一个开发服务器，这样就可以在不切换窗口的情况下直接看到 JavaScript 或 Scss 的语法错误等问题。如果不支持，请打开一个独立的终端窗口来运行 fj serve，并且时常留意一下终端窗口和系统的错误提示框。

3. 项目结构

用 FrontJet 创建的项目，具有一个默认的目录结构。

这个结构中有很多 README.md 文件，用于解释当前目录的结构以及用途，它们不会出现在编译结果中，并且可以随意删除。也可以自行编辑它，用于在项目组中保持共识。

本项目的结构简介如下：

```
|-- app（源码的根目录）
|   |-- animations（自定义动画）
```

```

| | | |-- README.md
| | | `-- ease.js (动画样例)
| |-- app.js (app模块的定义文件)
| |-- components (组件型指令)
| | |-- README.md
| | `-- layout (外框架)
| | | |-- _layout.html (模板)
| | | |-- _layout.js (控制器)
| | | |-- _layout.test.js (与控制器对应的单元测试)
| | | |-- _layout.scss (样式)
| | | |-- footer.html
| | | |-- footer.js
| | | |-- footer.scss
| | | |-- header.html
| | | |-- header.js
| | | |-- header.scss
| | |-- menu.html
| | |-- menu.js
| | `-- menu.scss
|-- configs (配置)
| | |-- README.md
| | |-- config.js (config阶段的代码)
| | |-- router.js (路由定义)
| | `-- run.js (run阶段的代码)
|-- consts (常量)
| | |-- README.md
| | `-- api.js (API定义)
|-- controllers (控制器)
| | |-- README.md
| | `-- home (首页)
| | | |-- index.html (模板)
| | | |-- index.js (控制器)
| | | |-- index.scss (样式)
| | | |-- notFound.html (模板)
| | | |-- notFound.js (控制器)
| | | `-- notFound.scss (样式)
|-- decorators (装饰器型指令)
| | `-- README.md
|-- favicon.ico (网站图标)
|-- filters (过滤器)
| | `-- README.md
|-- forks (系统分支)
| | |-- README.md
| | |-- android (适用于安卓浏览器的文件)
| | | `-- README.md
| | |-- default (适用于其他系统的文件)
| | | `-- README.md
| | `-- ios (适用于iOS浏览器的文件)
| | `-- README.md
|-- icons (svg图标源文件)

```

```

| | `-- README.md
| |-- images (普通图片)
| | `-- README.md
| | `-- logo.png
| |-- index.html (首页)
| |-- libraries (第三方非Angular库、非bower文件, 会被最先引用)
| | `-- README.md
| |-- services (服务)
| | |-- interceptors (拦截器, 用于过滤通过Ajax上传或下载的数据)
| | | |-- AuthHandler.js (401的处理器)
| | | |-- ErrorHandler.js (其他4xx、5xx错误码的处理器)
| | | |-- LoadingHandler.js (加载中界面)
| | | `-- README.md
| | |-- sao (服务访问对象)
| | | `-- README.md
| | `-- utils (工具类服务)
| | `-- README.md
|-- styles (样式定义)
| |-- README.md
| |-- _app.scss (应用程序的定义, 自动引入所有具体页面的Scss)
| |-- _bootstrap.scss (对Bootstrap的样式重定义)
| |-- _common.scss (具有跨项目复用价值的样式)
| |-- _icons.scss (根据svg图标编译出的样式文件)
| |-- _variables.scss (变量定义, 包括对Bootstrap的覆盖式样式定义)
| `-- main.scss (总的CSS文件, 用于依次引入其他文件, 一般不在此处定义样式)
|-- bower.json (bower库的名称及版本列表)
|-- bower_components (bower库文件)
|-- dist (编译结果/供最终发布的文件)
|-- fj.conf.js (FrontJet的配置文件, 详情见注释)
|-- mock (Mock服务器)
| |-- README.md
| |-- package.json
| |-- resources (资源数据定义)
| | `-- users.js
| |-- routers (服务端路由实现)
| | `-- users.js
| |-- routers.js (路由列表)
| |-- server.js (服务器启动文件)
| `-- utils (工具类)
| `-- resourceMixin.js
|-- test (测试)
| |-- e2e (端到端测试)
| | |-- demo.js
| | `-- readme.md
| |-- karma.conf.js (Karma的配置文件, 用于单元测试)
| |-- protractor.conf.js (Protractor的配置文件, 用于端到端测试)
| `-- unit (单元测试)
| `-- readme.md
|-- .bowerrc (bower的配置文件, 用于指定bower路径等)
|-- .editorconfig (编辑器配置, 用于在不同的编辑器之间统一缩进等代码风格)

```



```
|-- .gitignore (Git的忽略列表, 匹配的不会被添加到Git库中)  
|-- .jshintrc (JavaScript代码风格检查工具jshint的配置文件, 用于定制代码检查规则)  
|-- tsd.json (第三方库名称及版本列表)  
`-- typings (第三方库定义)
```

其中 `app` 目录的内部结构都是可以任意调整的, 不会影响 `FrontJet` 的运行。当要对传统项目使用 `FrontJet` 时, 可以将其源文件拷贝到 `app` 目录下。其他目录和文件的用途这里只做简单介绍, 后面的文章中涉及时再展开讲解。

1.4 实现第一个页面：注册

接下来, 我们开始实现第一个迭代的第一个功能: 10. 注册。

我们把能够通过 URL 独立访问的一项功能简称为“一个路由”, 这里为注册功能分配一个名叫 `/reader/create` 的路由。

之所以不使用 `/register` 的形式, 是希望在各个 URL 之间保持统一, 这也是我们在整个项目中将贯穿的一个约定。

1.4.1 约定优于配置

如同后端开发一样, 我们将 `reader` 称为 `controller`, `create` 称作 `action`, 中间还可以有一个 `id`, 所以, 典型的 URL 是这样的: `/$controller/:id?/$action`, 其中的 `id` 字段是可以省略的, 取决于具体的 `action`。

这样, 我们在 URL 和文件所在的路径之间就可以建立一个简单的映射关系: 拿到一个 URL, 如 `/reader/1/edit`, 其中 `reader` 是 `$controller`, `edit` 是 `$action`, 于是我们知道它的代码位于 `app/controllers/reader` 目录下, 其模板为 `edit.html`, 控制器为 `edit.js`, 样式为 `edit.scss`。

这有什么好处呢? 比如测试人员报告说一个页面存在 Bug, 其 URL 是 `/book/1/preview`, 我们一看 Bug 报告, 判断出其错误位于控制器中, 于是, 我们直接开始修改 `app/controllers/book/preview.js`。

如果使用的是 `IntelliJ/WebStorm`, 那么我们只要按下 `Cmd-Shift-N` (`Navigate|File...`, `Mac` 下) 组合键, 输入 `preview.js`, 然后回车, 就可以直接开始编辑了。其他 IDE 也有类似的功能。

不要小看这一点点约定, 它可以节省很多不必要的时间浪费, 特别是在多人协作开发时, 你的代码可能会被很多人修改, 如果连这个都需要沟通或者阅读代码, 那么浪费的时间和精力也是很可观的。

1.4.2 定义路由

分配完 URL，我们还需要把这个 URL 和控制器、模板对应起来。虽然我们有了一个约定，但是程序并不知道，我们还需要找个地方声明一下，这个地方就是 `app/configs/router.js`。

范例工程会给它生成一个代码骨架，为了方便加注释，我对一些语句进行了额外换行，实际代码中是不需要这么多换行的：

```
// 声明此JavaScript为严格模式，以回避一些JavaScript的低级错误，后面的代码摘录时将省略这句，
// 但其实每个JavaScript文件中都有这句话
'use strict';

// 引用模块——它已经在app.js中创建过了，所以这里只需要引用
angular.module('com.ngnice.app')
// 声明config函数，它的参数是一个回调函数，这个回调函数将在模块加载时运行，以便对模块进行配置，
// 路由就是配置的一种
.config(
// 声明config回调函数，它需要两个参数，一个叫$stateProvider，一个叫$urlRouterProvider，这两个都是
// 第三方模块angular-ui-router提供的服务。注意，这些参数名都不可随便修改，具体原因我会在
// 第3章中讲解
function ($stateProvider, $urlRouterProvider) {
  // 声明一个state
  $stateProvider.state(
    // 路由名称
    'home',
    // 路由定义对象
    {
      // URL
      url: '/',
      // 模板所在路径，从app起算
      templateUrl: 'controllers/home/index.html',
      // 控制器名称，as vm是一项最佳实践，其原理将在第4章中详细讲解
      controller: 'HomeIndexCtrl as vm'
    }
  );

  // 定义“页面未找到”路由
  $stateProvider.state('notFound', {
    url: '/notFound',
    templateUrl: 'controllers/home/notFound.html',
    controller: 'HomeNotFoundCtrl as vm'
  });

  // 定义默认路由，即遇到未定义过的URL时跳转到那里
  $urlRouterProvider.otherwise('/notFound');

  // 自己的路由写在这里

});
```

我们自己的路由要如何声明呢？为了保持一致性，我们这里使用 `angular-ui-router` 的方

式声明为多级路由：

```
// 定义一个父路由，它只用于提供URL
$stateProvider.state('reader', {
  // 所有子路由都会继承这个URL
  url: '/reader',
  // 父路由中一般只要提供一个这样的template就够了，不必使用templateUrl，页面中公共的部分
  // 通过组件型指令去实现会更灵活、更漂亮
  template: '<div ui-view></div>',
  // 抽象路由不能通过URL直接访问，比如直接访问/reader路径会跳转到otherwise中去
  abstract: true
});

// 定义一个子路由
$stateProvider.state(
// 名称，注意，这个名称不是随便取的，angular-ui-router会使用“点”对其进行分割，并且从前往后
// 逐个执行，所以这个名称中的每一段都要存在
'reader.create',
{
  // 子路由的路径，angular-ui-router会把各级父路由与当前路由的URL组合起来，作为最终的访问
  // 路径，如：`/reader/create`
  url: '/create',
  // 子路由模板
  templateUrl: 'controllers/reader/create.html',
  // 子路由控制器
  controller: 'ReaderCreateCtrl as vm'
});
```

定义完路由，我们仍然不能直接通过 URL 访问它，如果访问则会在浏览器控制台中发现一个错误信息：angular.js:10126 Error: [ng:areq] Argument 'ReaderCreateCtrl' is not a function, got undefined(angular.js:10126 错误：[ng:areq] 参数 'ReaderCreateCtrl' 不是函数，而是未定义。)，原因是没有找到名为 ReaderCreateCtrl 的控制器。

接下来，我们就对它所引用的模板和控制器进行实现。

首先我们要创建一个 app/controllers/reader/create.js 文件，和一个 app/controllers/reader/create.html 文件，我们来看一个空白的 create.js 文件：

```
// 引用应用模块
angular.module('com.ngnice.app')
// 注册一个控制器
.controller(
// 控制器名称
'ReaderCreateCtrl',
// 控制器实现
function ReaderCreateCtrl() {
  // 在ReaderCreateCtrl as vm语法下，当前函数的this指针指向的其实是$scope.vm变量，作为
  // 一项约定和最佳实践，我们把它赋值给vm变量。我们在程序中不再直接使用this，因为JavaScript
  // 中的this很容易给一些不熟悉JavaScript的程序员造成混乱。
```

```
var vm = this;
});
```

接下来，我们就要开始实现它们了。

如果已经有 UX（用户体验设计师）或 BA（业务分析师）给出的原型图，那么建议从设计 Model 的数据结构开始，这样有助于更深入的理解 Angular 开发中最显著的特点：模型驱动。如果是从零开始，也可以先设计 HTML。我们这个项目的开发是单人项目，所以我们先直接设计 HTML。到本节的最后，我再来讲解根据原型图做模型驱动开发的过程。

我们要设计一个 HTML，但不用设计一个“漂亮的”HTML。注意，在一个项目组中，不同的角色是需要分工的，要把 UX 擅长的工作留给 UX；即使是单人项目，也需要把不同类型的工作分开完成。

在注册页，我们需要一个表单，它具有如下业务意义上的字段：邮箱、昵称、密码、确认密码；还需要一些技术和法律意义上的字段：图形验证码（captcha）、网站服务协议、“同意服务协议”复选框。

由于我们的业务并不需要手机号、年龄之类的字段，那么我们就不要收集它。这种“最小信息”原则，可以帮助你在受到安全攻击的时候把损失控制在最小。同时，把需要填写的内容控制在最小范围内，也有利于提升用户体验。

我们的第一个 HTML 页面如下：

```
<!-- 这是一个表单 -->
<form>
  <!-- 邮箱 -->
  <div>
    <!-- for标签用来在label和input之间建立关联：你点击label的时候就相当于点击了for所指向的input -->
    <label for="_email">邮箱</label>
    <!-- 注意type=email -->
    <input id="_email" type="email"/>
  </div>
  <div>
    <label for="_nickname">昵称</label>
    <input id="_nickname" type="text"/>
  </div>
  <!-- 密码型字段 -->
  <div>
    <label for="_password">密码</label>
    <input id="_password" type="password"/>
  </div>
  <div>
    <label for="_retypedPassword">确认密码</label>
    <input id="_retypedPassword" type="password"/>
  </div>
```

```

<div>
  <label for="_captcha">图形验证码</label>
  <input id="_captcha" type="text" />
  <img src="" alt="图形验证码"/>
</div>
<div>
  <input id="_accepted" type="checkbox"/>
  <label for="_accepted">
    我已阅读并同意
    <a href="">用户服务协议</a>
  </label>
</div>
<div>
  <button type="submit">提交</button>
</div>
</form>

```

注意，用来在 input 和 label 之间建立关联的 id 字段都是用下划线开头的，这并不是随意为之，而是要把 id 留给写“端到端测试”的人员。我们把这些不能不用的 id 全用下划线开头，有助于防止潜在的冲突。

现在，我们切到浏览器中，会看到一个很难看的表单，如图 1-1 所示。

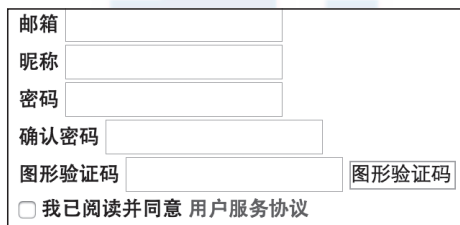


图 1-1 简单的表单

虽然简陋，但已经足够表现我们的 HTML 骨架了。

接下来，我们需要把它修到可正常交互的级别。

首先，我们需要给每个输入型字段（input/textarea/select 等）绑定一个 Model 变量。仅以邮箱字段为例，我们把它修改为：

```

<div>
  <label for="_email">邮箱</label>
  <input id="_email" type="email" ng-model="vm.form.email" />
</div>

```

这里的 ng-model 就是 Angular 中一系列“魔法”的关键。它是一个 Angular 指令，其作用是把所在的 input 元素和 ng-model="" 中的表达式建立双向绑定，这种双向绑定意味着，当表达式的值发生变化时，input 的 value 会跟着变化，反过来，当 input 中的 value 由

于用户操作而发生变化时，绑定表达式的值也会相应跟着变化。而且这两者的数据格式并不需要保持一致，`ng-model` 指令提供了一系列机制在两者之间进行转换。

`ng-model` 并不限于用在 `input/textarea/select` 元素中，事实上，它几乎可以用在任何元素中。但只有这几个元素可以直接使用 `ng-model`，用于其他元素时需要自己写自定义指令来实现双向绑定。这是因为 Angular 自带了对 `input/textarea/select` 的重写指令，重定义了它们的行为，使其可以支持 `ng-model`。在后面的章节，我们会看到如何在自定义指令中支持 `ng-model`。

这里还涉及另一项约定和最佳实践：把当前表单的所有字段绑定到一个叫作 `vm.form` 的对象中，这样我们就可以很方便的把表单数据作为一个整体进行处理，比如提交或重置。

除了“确认密码”和“同意协议”之外的其他的字段可以以此类推，不再摘引源码。

“确认密码”字段之所以特殊，是因为它并不需要最终提交给服务端，我们只是为了防止用户输入错误，靠前端来校验就已经足够了。我们设想一下攻击场景，发现对它只做前端校验并不会构成安全漏洞。所以，我们不需要把它绑定到 `vm.form` 对象的属性中去，用个独立的 `vm.retypedPassword` 变量就可以了。

而“同意协议”字段也同样可以依靠纯前端验证。在法律上，我们只要尽到了提醒和询问的义务即可。如果用户通过非正规手段绕过前端直接访问服务端，不能作为未曾同意协议的借口。

注意，前面我们只是修改了 HTML 就已经完成了双向绑定，我们并没有在控制器中定义 `vm.form.email` 变量，甚至连 `vm.form` 对象都没有定义。这是因为在 Angular 中做了容错处理，发现一个变量没有定义时，它会自动帮我们定义一下，而不会触发错误，这个特性在写模板时非常有用。

“图形验证码”的字段也比较特殊，我们把它留到下一节去处理。现在，我们的界面就已经到了最初的可交互级别。

接下来我们有两个分支可以走：套用 Bootstrap 类进行初步美化（UX 分支），或者开始实现表单提交代码（程序员分支），两者可以同时进行。

为了尽快看到“可行走骨架”，我们选择程序员的分支：实现表单提交代码。要在收集了表单数据的基础上进一步实现表单提交，我们就要借助另一个指令了，这个指令叫作 `ng-submit`。

直觉上，我们可能希望在提交按钮上绑定一个事件来完成表单提交，但更好的方式是在 `form` 上绑定一个 `ng-submit` 事件。这是因为触发表单提交并不是只有点击“提交”按钮

这一种方式，用户还可以在 input 中敲回车键来直接提交表单，在大多数场景下，这是更为友好的方式。但更重要的是，这样的 HTML，其表意性更强一些。

这个步骤对代码的影响很小，在 HTML 中，我们只要把 `<form>` 改为 `<form ng-submit="vm.submit(vm.form)">` 即可，在 JavaScript 中增加几句指令即可：

```
vm.submit = function(form) {
  console.log(form);
};
```

由于还没有写后端代码，因此我们这里只把它要提交的代码打印到控制台中，我们只要打开 Chrome 控制台就可以验证一下我们要提交的数据是否正确了。

在实现真正的后端提交之前，我们还需要实现一个服务器，由于本书只讲前端开发，所以我们就把后端程序当做一个黑盒子，不再讲解实现原理。下一节我们会简单讲一下如何让它在本机跑起来，以及如何通过设置反向代理来解决跨域问题。

1.4.3 把后端程序跑起来

我们把书中的后端代码也放在了 GitHub 上，作为起步，建议只下载 BookForumApi.zip 文件就可以了。解压之后，会发现一个 server 文件和一个 server.bat 文件，如果在 Windows 下面，直接执行 server.bat 即可，如果在 Linux 或 Mac 下，请执行 `./server` 命令。它会启动一个开发服务器，这个开发服务器会内置一个 H2 内存数据库，每次重新启动服务器就会将其内容清除。它会自动监听 5080 端口，如果你的机器上 5080 端口已经被占用，请使用 `server 5090` 等命令来重新指定一个端口。

接下来，我们需要修改前端项目中的 FrontJet 配置文件——项目根目录下的 `fj.conf.js`：

```
module.exports = function (config) {
  // 可以定义多条规则，后面的规则会覆盖前面的
  config.rules = [
    {
      url: /^\/api\/(.*)$/, // 要代理的URL，可以是正则表达式，也可以是字符串，如字符串'/api'，将被处理成/^\/api\/(.*)$/的形式，两者等价
      rewrite: '$1', // 可选，默认把原来的URL完全传过来，即：不重写
      proxy: 'http://localhost:5080/BookForumApi/', // 反向代理设置
      cookie: {
        path: '/api', // 覆盖原服务器的cookie path设置（如果有）
        domain: 'localhost' // 覆盖原服务器的cookie domain设置（如果有）
      },
      delay: 500 // 延迟毫秒数，可选
    },
    {
      url: '^/api/readers',
```



```
        delay: 0
    }
    ];
    // 用户自定义的Middleware将优先于默认的
    config.middlewares = [
        function(req, res, next) {
            next();
        }
    ];
};
```

现在可以通过 POST <http://localhost:5000/api/readers> 来模拟读者注册的后端 API 了。至此前端已经配置完毕。如果你只关心前端编程，那么可以跳过本节的剩余部分。

写给要深入后端程序的同学

事实上，后端程序可选用任何语言和框架，只要它可以提供 HTTP 形式的接口即可。这就给整体的技术架构提供了高度灵活性，甚至可以在不中断服务的情况下逐步把后端程序从一种框架替换为另一种框架，甚至从一种语言替换为另一种语言。

如果要了解后端程序的工作原理或对其进行修改，那么需要把全部源码下载下来。我在这里不展开讲解，只对后端程序的框架写个简介。

这个范例中的后端程序使用的是一个基于 Groovy 的框架，叫作 Grails，Java 程序员应该对 Groovy 不陌生，它就是现在正火爆的 Gradle 中所使用的编程语言。Groovy 的源码和 Java 的写法兼容程度很高，事实上，大部分 Java 代码直接拷贝到 Groovy 中就能编译，不需要做什么修改。

而 Grails 是一个全栈式的后端框架，它的设计思想是来自 Ruby 世界中的 Rails 框架，而它的基础技术栈则是 Hibernate、Spring MVC 等成熟的 Java 框架。它可以流畅的运行在 JVM 上，还可以无缝接入现有的 Java 框架或库，具有成熟的生态环境。对于讨厌 Java 传统框架的 xml 配置的人，Grails 是个福音。而且，虽然不推荐，但必要的时候，你仍然可以通过定义 xml 文件来使用传统风格的配置方式。

Grails 中囊括的技术很多，但是在前后端分离的架构下，我们只要使用它的两大特性就行了：GORM、REST，其他的，如 GSP、taglib、i18n 等特性则可以完全不用管它。

GORM 最大的好处就是只要定义领域类就行了，不用写 getter/setter，不用配置 XML，不用写 annotation。

除此之外，它还能直接定义业务规则，如用户名的长度限制等。这些限制会在保存实例的时候自动被检查。

不过，它最大的亮点是支持 DSL 式的查询，事实上，它的底层就是 Hibernate 的

Criteria 特性，这保障了它的成熟度和稳定性。而在 Groovy 的强力支持下，它被写成了内嵌的 DSL 语句。IntelliJ 甚至可以对这个 DSL 进行相当细致的智能提示。

REST 是另一大亮点，虽然它提供的默认控制器仍然不够成熟，但好在它只是很薄的一层，我们可以通过自定义模板的方式来按我们的方式自动生成 REST 服务。而 JSON 解析、回应、数据格式转换等很多 REST 中必备的特性，它都实现的相当不错。

1.4.4 连接后端程序

现在，我们已经在 5000 端口上跑着前端程序，在 5080 端口上跑着后端程序，并且通过 FrontJet 的反向代理功能把后端也代理到了 5000 端口下，从而避免了跨域问题。

万事俱备，我们可以正式向服务端提交请求了。

首先，我们需要写一个服务访问对象（Service Access Object, SAO），这是一个和后端的 DAO 类似的概念，不过好在我们不用从头实现它。在 Angular 中，有两个用于支持 Ajax 请求的服务，一个叫 \$http，一个叫 \$resource。\$http 是一个广义的 Ajax 服务，可用于支持任何形式的 Ajax 请求；\$resource 则是专门用于访问 REST 服务的对象，它封装了一系列 REST 最佳实践和规约。

显然，对于我们这样一个新系统，直接使用 \$resource 是最好的选择。对于一些老系统，则可以穿插使用 \$http 和 \$resource。

不过只要有条件，最好把后端 API 逐步改造为 REST 风格，本书中有两处对 REST 进行详解：一处是在第 3 章“背后的原理”中，对 REST 原理、概念进行解释；另一处是在第 4 章“最佳实践”中，介绍了 REST 方面的最佳实践和开发规范。

通过 \$resource 定义一个 SAO 非常简单，最简单的 API 只要一句就够了：

```
angular.module('com.ngnice.app').factory('Reader', function ReaderFactory ($resource) {
    return $resource('/api/readers/:id', {id: '@id'});
});
```

在这里，我只封装了一个 API 路径，其他的 CRUD（创建、读取、更新、删除）操作则直接使用 \$resource 已经提供好的封装。虽然封装很浅，但却很实用，我们来看一下封装前后调用方式的区别。

封装前：

```
angular.module('com.ngnice.app').controller('ReaderCreateCtrl', function
    ReaderCreateCtrl($resource) {
    var vm = this;
    var Reader = $resource('/api/readers/:id', {id: '@id'});
```

```

    vm.submit = function(form) {
        Reader.save(form);
    };
});

```

封装后：

```

angular.module('com.ngnice.app').controller('ReaderCreateCtrl', function
    ReaderCreateCtrl(Reader) {
        var vm = this;
        vm.submit = function(form) {
            Reader.save(form);
        };
    });

```

表面看，封装之后省不了多少代码，但是，我们封装的首要目的可不是节省代码，而是提高可读性，对比一下两者，体会它们在表意性上的差别！

除此之外，我们的封装还获得了额外的灵活性。这得益于由 \$http 服务提供并且被 \$resource 服务继承的拦截器（interceptor）机制。在后面的章节中，我们会根据需求的深化，逐步展示拦截器的用法，这里先不展开。

上面的代码只是提交了请求，并未关心是否创建成功，但这对用户来说显然很重要。接下来，我们增加处理返回结果的功能。

```

Reader.save(form,
    function (reader) {
        console.log(reader);
    },
    function (resp) {
        console.log(resp);
    }
);

```

\$resource 的 save、update、get、query、delete 等都可以带两个回调函数，第一个回调函数是成功回调，第二个是失败回调。成功回调函数会传一个参数过来，就是由服务器返回来的 JSON 对象，而失败回调函数则传回一个 response 对象，其中有状态码等信息。

在这里，我只是把它们输出到控制台来演示执行效果。正式项目下，对于 save、update 或 remove 的成功回调，它通常是个路由跳转语句；在 query 或 get 请求中，则把返回值赋值到 vm 的成员上，供模板中绑定。失败回调通常不用单独处理，而是通过拦截器（interceptors）进行统一处理，这属于一个略有难度的话题，因此我们把它放在稍后的章节中实现。

我们和后端的对接就先暂时告一段落，过几节之后再加入高级功能。

1.4.5 添加验证器

至此，我们在注册页面中还缺少一个重要功能：数据验证。

数据验证包括两个主题：一是定义验证规则，用于验证数据的有效性；二是显示验证结果，要把验证的结果用友好的方式显示给用户。

HTML5 内置了一些验证功能，并且会显示内置的提示，但是这种提示容易破坏 UX 设计的整体效果，因此我们要先把它禁用掉。我们只要在 form 上增加一个属性即可：

```
<form ng-submit="vm.submit(vm.form)" novalidate="novalidate">
```

我们先来看邮箱字段。它一共有两个验证规则：首先，它必须是格式合法的邮箱；其次，它是必填项。

Angular 有一个内置指令改写了 `input[type=email]` 元素，它会在接收到输入的时候检查是否符合标准的邮箱格式，并且设置相应的有效性标识，所以我们只要指定了 `type=email` 就不用再管了。

验证必填项规则有两种选择：可以使用 HTML 中标准的 `required` 属性，如 `<input type="email" required="required" />`，Angular 改写的 `input` 指令会检测它，并且设置相应的有效性标识；还可以使用 Angular 的 `ng-required` 指令，如 `<input type="email" ng-required="true"/>`。两者的区别在于后者是可以编程控制的。也就是说后者的值可以是一个 `scope` 变量，如 `<input type="email" ng-required="vm.needEmail"/>`，而前者是不能通过 `<input type="email" required="{{vm.needEmail?'required':''}}"/>` 的形式来控制的，因为在 HTML 中，像 `required/disabled` 之类的属性，只要“出现”就视为有效，而不管它的值是不是空。注意，`ng-required` 的值并不需要使用 `{{}}` 包裹起来，详细的原因我们会在第 2 章“概念介绍”中讲解。现在只需记住：除下列指令外的内置指令都不需要把值用 `{{}}` 包裹起来：`ng-src`、`ng-href` 以及不太常用的 `ng-bind-template` 和 `ng-srcset`。

每次用户输入之后，这些验证器就会按照优先级和出现顺序依次被验证。当验证全部成功时，Angular 就会把用户的输入转换成 Model 中的值；如果任何一个验证不成功，那么 Model 中的值会保持原样，并且在 Model 上增加一个 `$error` 对象，每一个失败的验证器，都会在其中出现一个和验证器同名的属性，其值为 `true`。如，邮箱格式的验证器出错时，`$error` 的内容就是 `{email: true}`。

禁用了内置的验证提示之后，我们还要使用自定义的界面来显示错误信息。

这时，Model 变量虽然是存在的，我们却没法在模板中直接引用它。要想引用它，我们还需要给它所属的 form 指定一个名字，再给它所在的 `input` 指定一个名字。例如：

```

<form name="form" ng-submit="vm.submit(vm.form)" novalidate="novalidate">
  <div>
    <label for="_email">邮箱</label>
    <input id="_email" name="email" type="email" ng-required="true" ng-
      model="vm.form.email"/>
  </div>
  ...
</form>

```

这时候，我们的 `$scope` 上就多出来一个名为 `form` 的变量，它的内容为：

```

{
  $dirty: false
  $error: Object
  $invalid: true
  $name: "form"
  $pristine: true
  $removeControl: function (control) {...
  $setDirty: function () {...
  $setPristine: function () {...
  $setValidity: function (validationToken, isValid, control) {...
  $valid: false
  accepted: Constructor
  captcha: Constructor
  email: Constructor
  nickname: Constructor
  password: Constructor
  retypedPassword: Constructor
}

```

其中的几个函数我们到第 3 章“背后的原理”中再详细讲解，我们先看看这几个成员变量：

- ❑ `$dirty` 表示用户是否在这个表单内任何一个输入框中输入过。
- ❑ `$pristine` 和 `$dirty` 正相反，表示尚未输入过。
- ❑ `$error` 在前面已经介绍过，不再赘述。
- ❑ `$invalid` 表示这个表单中的数据是否有无效的。只要任何一个输入框是无效的则将整个表单视为无效的。
- ❑ `$valid` 和 `$invalid` 正相反，表示数据有效。
- ❑ `$name` 表示这个表单的名字，也就是前面我们指定的 `name` 属性。
- ❑ `accepted/captcha/email/nickname/password/retypedPassword` 是表单中各个输入框的 `Model`。

我们再来看看 `email` 字段的 `Model`：

```

{
  $$validityState: ValidityState
  $dirty: false
  $error: Object
  $formatters: Array[2]
  $invalid: true
  $isEmpty: function (value) {...
  $modelValue: undefined
  $name: "email"
  $parsers: Array[2]
  $pristine: true
  $render: function () {...
  $setPristine: function () {...
  $setValidity: function (validationErrorKey, isValid) {...
  $setViewValue: function (value) {...
  $valid: false
  $viewChangeListeners: Array[0]
  $viewValue: undefined
}

```

它也同样包含了 form 中的几个属性，其含义也类似，但适用范围是这个 Model 所在的输入框。这些值全都由 Angular 的内置指令进行维护。

其他的属性和方法我们也将第 3 章“背后的原理”中详细讲解。本节中我们只要记住这几个常用属性：\$dirty/\$pristine、\$valid/\$invalid、\$error。因为接下来我们就要使用它们了。

看完这些数据结构，我们就可以在模板中通过数据绑定来显示了：

```

<div>
  <label for="_email">邮箱</label>
  <input id="_email" name="email" type="email" ng-required="true" ng-model="vm.
    form.email"/>
  <!-- 用户输入过，并且无效，则显示 -->
  <ul ng-if="form.email.$dirty && form.email.$invalid">
    <!-- 如果“邮箱格式”验证器报错 -->
    <li ng-if="form.email.$error.email">无效的邮箱格式</li>
    <!-- 如果“必填项”验证器报错 -->
    <li ng-if="form.email.$error.required">此项为必填项</li>
  </ul>
</div>

```

除了需要绑定到 Model 变量之外，这就是一段很普通的模板。

显然，这不是一种好办法，除了必须给每个字段取一个名字之外，当字段多的时候，模板中还会出现大量的重复代码。如何解决这个问题呢？我们来写本书的第一个自定义指令吧。

1.4.6 “错误信息提示”指令

我们之所以添加 name 属性，是为了能在模板中引用到 \$error 对象，但我们其实还有另一种方式来引用它——那就是“指令”。

我们先来看代码，把一些基础知识插入注释中，然后再解释原理：

```
angular.module('com.ngnice.app').directive(
// 指令名称，它会按照约定转换成减号分隔的标识符后才能在模板中使用：`bf-field-error`，这里的bf
// 是book-forum的缩写，用这个前缀来防止和其他指令冲突。
'bfFieldError', function bfFieldError($compile) {
  return {
    // 限制为只能通过属性 (Attribute) 的形式使用，如`<input bf-field-error />`，另一种
    // 常用的限制是E，表示通过元素 (Element) 的形式使用，如`<bf-field-error> </bf-
    // field-error>`，两者也可以组合，如`restrict: 'EA'`
    restrict: 'A',
    // 这个元素上必须有一个ng-model属性，如果没有，就会报错
    require: 'ngModel',
    // link函数会在当前指令初始化的时候被自动执行
    link: function (scope, element, attrs, ngModel/* 将传入require的值 */) {
      // 创建一个`子scope`，`true`参数表示这个子scope是个独立作用域，它不会从父级作
      // 用域自动继承属性
      var subScope = scope.$new(true);
      // 在子scope上增加两个函数，供模板中使用
      // 是否有需要显示的错误
      subScope.hasError = function() {
        // 除了判断数据是否无效 ($invalid) 外，还要判断用户是否已经输入过 ($dirty)，
        // 否则刚显示表单就会出现一堆错误信息，这显然是不对的
        return ngModel.$invalid && ngModel.$dirty;
      };
      // 错误信息的内容
      subScope.errors = function() {
        // 我们先直接把$error显示到界面上，回头再改进为用户友好的格式
        return ngModel.$error;
      };
      // 把一段HTML编译成“活DOM (Live DOM)”，然后把subScope传给它，这个Live DOM
      // 将会跟随subScope的变化自动更新自己。
      var hint = $compile('<ul ng-if="hasError()">{{errors()}}</ul>')
        (subScope);
      // 把这段Live DOM追加到当前元素后面，好让它显示出来
      element.after(hint);
    }
  };
});
```

这段指令的关键在于 require 属性。

require 的值是另一个指令的名称，但实际上它引用的是那个指令的控制器实例。require 机制非常有用，因为有时候多个指令是需要互相配合的，它们必须相互“看到”才

能进行协作，`require` 就是让它们相互“看到”的标准化机制。比如，这个指令中声明了 `require: 'ngModel'`，那么当 Angular 初始化它的时候，就会在它所在的元素上寻找一个叫作 `ng-model` 的指令，然后取得它的控制器实例。找到之后，就会把这个控制器的实例作为 `link` 函数的第四个参数传进来。传进来之后，我们就可以直接访问 `ngModel` 上的属性和方法了，这就是所谓“指令之间的协作”。

接下来，我们创建一个模板和一个 `scope`，然后把它们绑定在一起，生成一个所谓的“Live DOM”。Live DOM 在使用上和普通的 DOM 元素没什么区别，只是它们可以感知到 `scope` 中数据的变化，并且据此来更新自己。你可能也注意到了，我绑定的数据不是属性，而是一个函数，这是 Angular 中的一个常见用法。这种神奇的更新机制，其工作原理是什么呢？我们将在第 3 章“背后的原理”中详细讲解。`$compile` 函数是把它们绑定在一起的关键，事实上，路由中也是用同样的机制来把 `scope` 和模板绑定在一起的。把静态 DOM 和 `scope` 变量绑定在一起，使其变成 Live DOM 的过程，在 Angular 中被称为编译（`compile`）。

我们拿到了编译结果之后，还要把它展示给用户并且和用户进行交互，于是我们通过 `element.after` 函数把它插入到当前元素的后面。

现在，这个指令已经可以工作了，但仍然是很粗糙的，它只是把 `$error` 对象的源码显示出来。接下来，我们要把它简单美化一下，变成一个普通的无符号列表。

我们先把它从一个对象显示成一个名字列表：

```
var hint = $compile('<ul class="bf-field-error" ng-if="hasError()"><li ng-repeat="(name, wrong) in errors()" ng-if="wrong">{{name}}</li></ul>')(subScope);
```

这里面涉及一个在 Angular 中非常重要的内置指令：`ng-repeat`。它可以对数据项进行枚举，这个数据项既可以是数组，也可以是对象，比如我们的 `$error` 就是一个对象。正如上面这段代码所示范的，枚举对象的语法非常简单：（属性名，属性值）in 对象，这些属性名和属性值可以被用于当前元素和子元素的绑定表达式中。枚举数组的语法更简单：迭代变量 in 数组。

不过，`ng-repeat` 虽然语法简单，但仍然有很多高级用法和坑，我们在后面的章节中会陆续提到，不过现在只要先了解这些就够用了。

但是 `email` 这样的提示对用户来说仍然太不友好了，我们需要把它转换成更友好的字符串。但是我们又想保证其内聚性，最好不要把生成用户友好的提示信息的逻辑放进来，这就用到了 Angular 中的另一个重要概念：过滤器。

我们先把过滤器的用法写下来，下一节我们再来实现它。

```
var hint = $compile('<ul class="bf-field-error" ng-if="hasError()"><li ng-repeat="(name, wrong) in errors()" ng-if="wrong">{{name | error}}</li></ul>')(subScope);
```

{{name | error}}，其中竖线后面的就是过滤器，用于把 error 的名称，如 email，转换成用户友好的提示信息。

1.4.7 用过滤器生成用户友好的提示信息

在本节中，我们将实现 error 过滤器。

我们先来看一个空白的过滤器：

```
angular.module('com.ngnice.app').filter('error', function () {
  return function (input) {
    // TODO: 实现转换/格式化逻辑
  };
});
```

大致来说，过滤器就是一个函数，它接收一个输入，然后返回一个输出，函数体负责把输入的变量转换成输出结果。

接下来，我们实现 TODO 部分的逻辑：

```
angular.module('com.ngnice.app').filter('error', function () {
  var messages = {
    email: '不是有效格式的邮件地址',
    required: '此项不能为空'
  };
  return function (name) {
    return messages[name] || name;
  };
});
```

我们首先定义了一个（名称，信息）的对照表，然后直接根据名称查阅，如果没有值则返回名称。注意，我们还重构了变量名，从 input 改成 name，其表意性更强。

如果将来错误类型进一步膨胀，那么我们就需要再次重构，这次要把 messages 提取出去。提取的方式是使用 constant，于是将其拆成了两个文件：

1) app/constants/Errors.js:

```
angular.module('com.ngnice.app').constant('Errors', {
  email: '不是有效格式的邮件地址',
  required: '此项不能为空'
});
```

2) app/filters/error.js:

```
angular.module('com.ngnice.app').filter('error', function (Errors) {
```

```

    return function (name) {
        return Errors[name] || name;
    };
});

```

我们先把错误信息拆成了一个独立的常量对象 (constant)，然后把它注入到原来的 filters 中，代替以前的 messages 变量。这样拆分出去之后，我们不但可以在过滤器中使用 Errors 常量，也可以在控制器、指令、服务等任何地方使用它了。

过滤器的实质是数据转换，常见的转换包括格式化和筛选。比如，我们本节实现的过滤器就是个格式化函数，用来把 Model 层的数据类型转换成 View 层的用户友好化信息。在本章稍后 1.5 节中，我们会通过“主题列表”来示范过滤器的筛选功能，最后再把主题列表及其回复扩展成主题树。不过，我们先要回到主线，把注册页面中剩下的功能写完。

1.4.8 实现自定义验证规则

邮箱字段的验证非常简单，Angular 直接内置了验证规则，但是“确认密码”的验证规则和“同意协议”的验证规则就没这么简单了。我们先来实现“确认密码”的验证规则，实现方式同样是写自定义指令 (bf-assert-same-as)，但是写法又有些不同了：

```

angular.module('com.ngnice.app').directive('bfAssertSameAs', function
    bfAssertSameAs() {
    return {
        restrict: 'A',
        require: 'ngModel',
        link: function (scope, element, attrs, ngModel) {
            var isSame = function (value) {
                // 取对照值，通过scope.$eval把attrs.bfAssertSameAs作为一个表达式在当前
                // 作用域中求值，否则它只是一个固定的字符串
                var anotherValue = scope.$eval(attrs.bfAssertSameAs);
                return value === anotherValue;
            };
            // 1.2.x中只能使用$parsers实现验证，1.3.x中增加了专门的$validators数组，可用
            // 更好的方式实现验证。
            ngModel.$parsers.push(function (value) {
                // 调用$setValidity设置验证结果，第一个参数就是名字，和$error中的属性名一
                // 致，但是取值相反，因为这里表示的是“有效”，而$error中表示的是“无效”
                ngModel.$setValidity('same', isSame(value));
                // 验证通过则返回转换后的值，否则返回undefined
                return isSame(value) ? value : undefined;
            });
            // 这是assertSameAs验证器所特有的，因为当对照值发生变化时，也要更新有效性状态
            scope.$watch(
                // 对这个函数值进行监控，变化时就触发下一个函数：变化通知
                function () {
                    return scope.$eval(attrs.bfAssertSameAs);
                }
            );
        }
    };
});

```

```

    },
    function () {
        // 变化时重新判断并设置验证结果
        ngModel.$setValidity('same', isSame(ngModel.$modelValue));
    }
    );
}
};
});

```

使用时的形式如下：

```

<input id="_retypedPassword" bf-field-error type="password" ng-required="true"
      ng-model="vm.retypedPassword" bf-assert-same-as="vm.form.password"/>

```

当确认密码和密码不同的时候，bf-field-error 指令在检查结果中就会显示 “same”。

接下来，我们需要让这个 same 显示成用户友好的提示信息。这一步就简单了，我们前面已经把错误信息提取到了一个 constant 里，现在只要把 same 加进去就行了：

```

angular.module('com.ngnice.app').constant('Errors', {
    email: '不是有效格式的邮件地址',
    required: '此项不能为空',
    same: '此项必须与上一项相同'
});

```

因为追求通用性，我们的提示信息还不够友好，还需要一种机制来进一步定制提示信息。定制的方式是给 bs-field-error 传入一个参数。如：

```

<input id="_retypedPassword" bf-field-error="{same: '确认密码必须与密码相同'}"
      type="password" ng-required="true" ng-model="vm.retypedPassword" bf-assert-
      same-as="vm.form.password"/>

```

这就需要我们修改 bfFieldError 指令了，修改后的代码如下：

```

...
    subScope.customMessages = scope.$eval(attrs.bfFieldError);
    var hint = $compile('<ul class="bf-field-error" ng-if="hasError()"><li ng-
      repeat="(name, wrong) in errors()" ng-if="wrong">{{name|error:customMessa
      ges}}</li></ul>')(subScope);
    ...

```

在这里，我们取了 bf-field-error 的表达式，并且交给 scope.\$eval 去求值，这样我们就取到了一个自定义的消息对象。然后我们通过 error:customMessages 语法把这个自定义消息作为参数传给 error 过滤器。当然，这时候的 error 过滤器还不能使用这个参数，接下来我们就改一下 error 过滤器的代码：

```

angular.module('com.ngnice.app').filter('error', function (Errors) {
    return function (name, customMessages) {

```

```

    // 把Errors和customMessages合并在一起，作为查阅源，customMessages中的同名属性会
    // 覆盖Errors中的
    var errors = angular.extend({}, Errors, customMessages);
    return errors[name] || name;
  };
});

```

这样，我们只修改了几句代码就实现了自定义消息功能。

“同意协议”的验证可以直接借助 `bf-assert-same-as` 指令来完成，只使用一个固定值“true”就可以了：

```

<input id="_accepted" type="checkbox" ng-model="vm.accepted" bf-assert-same-
as="true"/>

```

至此，我们把所有的验证都添加完了，接下来我们还需要对表单的提交进行控制，即：只有全部字段都通过了验证的表单才允许提交。它的原理很简单：只要在表单无效时把“提交”按钮禁用掉就行了。换成 Angular 的语言就是：“视图”中“提交”按钮的禁用状态来自于“模型”中表单的有效性。注意，这种思维的转换对于真正掌握 Angular 是很重要的——从以过程为中心的思维，转换成以模型为中心的思维。

Angular 提供了一个现成的指令：`ng-disabled`，可以让我们完成这个功能：

```

<form name="form" ng-submit="vm.submit(vm.form)" novalidate="novalidate">
...
  <button type="submit" class="btn btn-primary" ng-disabled="form.$invalid">立
    即注册</button>
...
</form>

```

我们首先给 form 加上了一个 name，以便我们可以在视图中引用它，然后我们通过 `ng-disabled` 指令把提交按钮的禁用状态关联到 `form.$invalid` 属性。当表单变为无效时，按钮被禁用；反之则被启用，这个过程完全是自动的，Angular 会负责视图和模型之间的同步。为了能看到更明显的禁用效果，我们给这个按钮加上了来自知名 CSS 框架 Bootstrap 的 `btn` 和 `btn-primary` 类。

至此，前端的验证功能已经全都完成了。

另外，Angular 1.3 以上的版本提供了更友好的自定义验证器：`ngModel.$validators`。如果你的系统没有兼容 IE8 的负担，那么不妨参考官方 API 文档，来尝试采用新的方式。

1.4.9 实现图形验证码

我们在注册表单上的欠账还剩一个图形验证码，不过这个实现起来就简单多了。

```

```

我们只要给 `img` 元素加上一个 `ng-src` 指令，然后把服务端的 `captcha` 地址传给它就行了。

可能有人会问，为什么这里我不直接用 HTML 内置的 `src` 属性呢？事实上 `ng-src` 确实是通过设置 `img` 的 `src` 属性来工作的。它们大多数情况下都相同，除了一个场景：假设我们的服务端地址是可以配置的，那么我们绑定的时候就得绑定到一个变量了：``。这时候，如果我们换成 `src`：`` 会怎样呢？当页面刚开始加载的时候，由于 Angular 还没有机会执行，所以花括号中的绑定表达式还没有机会被求值，于是浏览器会尝试读取 `{{api.root}}/captcha.jpg`，显然，这个地址是不存在的，于是浏览器得到一个 404 错误，这不是我们所预期的。而 `ng-src` 是个 Angular 指令，它在 Angular 启动完成之后才会被执行，于是得到了正确的值：`/api/captcha.jpg`，并且赋给 `src` 属性，这样就不会出现一个意外的 404 了。

我们还需要加上一个功能：点击图形验证码的时候自动换一张。为了让注册页的代码内聚性更高、更清洁，我们把 `captcha` 功能封装成一个独立的指令：

```
angular.module('com.ngnice.app').directive('bfCaptcha', function bfCaptcha() {
  return {
    restrict: 'A',
    link: function (scope, element) {
      var changeSrc = function() {
        // 通过附加随机码来强制更换图形验证码
        element.attr('src', '/api/captcha.jpg?random=' + new Date().
          getTime());
      };
      // 启动的时候先更换一次，以便显示出来
      changeSrc();
      // 点击的时候也更换一次
      element.on('click', function() {
        changeSrc();
      });
    }
  };
});
```

使用这个指令的代码就很简单了：``。

至此，注册页的功能部分就全部完成了，美化部分如果有额外的人力就已经可以开始了，不过我们人力不够，所以还是等到最后再统一处理吧。我们先继续完成后面的功能：主题树和登录功能。

1.5 实现更多功能：主题

1.5.1 实现主题列表

我们还没有实现发帖和取帖子列表的后端功能，甚至连这个 API 该设计成什么样都还不清楚。不过没关系，我们可以先在前端做一些模拟数据，等到前端所需的数据结构确定下来，API 也就基本成型了，这也是敏捷方法的一种应用。

我们先做些准备工作。在 `router.js` 中增加两个路由定义：

```
$stateProvider.state('thread', {
  url: '/thread',
  template: '<div ui-view></div>',
  abstract: true
});
$stateProvider.state('thread.list', {
  url: '/list',
  templateUrl: 'controllers/thread/list.html',
  controller: 'ThreadListCtrl as vm'
});
```

然后创建两个空文件：`app/controllers/thread/list.js` 和 `app/controllers/thread/list.html`。

接下来我们就正式开始实现。

最明显，我们需要几个字段：标题（`title`）、发帖人（`poster`）、发帖时间（`dateCreated`）。于是我们得出了第一个控制器：

```
angular.module('com.ngnice.app').controller('ThreadListCtrl', function ThreadListCtrl() {
  var vm = this;
  vm.items = [
    {
      title: '这是第一个主贴',
      poster: '雪狼',
      dateCreated: '2015-02-19T00:00:00'
    },
    {
      title: '这是第二个主贴，含有字母abcd和数字1234',
      poster: '破狼',
      dateCreated: '2015-02-19T15:00:00'
    }
  ];
  for (var i = 0; i < 10; ++i) {
    vm.items.push({
      title: '主题' + i,
      poster: 'user' + i,
      dateCreated: '2015-02-18T15:00:00'
    });
  }
});
```



```
    }
  });
```

和第一个模板：

```
<table class="table table-hover table-striped">
  <thead>
    <tr>
      <th>主题</th>
      <th>作者</th>
      <th>发帖时间</th>
    </tr>
  </thead>
  <tbody>
    <tr ng-repeat="item in vm.items">
      <td>{{item.title}}</td>
      <td>{{item.poster}}</td>
      <td>{{item.dateCreated}}</td>
    </tr>
  </tbody>
</table>
```

这里起主要作用的是我们的老朋友：ng-repeat 指令。

现在我们已经可以通过在地址栏输入 <http://localhost:5000/#/thread/list> 来预览效果了。

但是这里的日期显示的仍然是 ISO-8601 格式的日期，对用户来说不够友好，接下来我们把它改成可读性更好些的格式。我们前面提到过，在 Angular 中，负责对数据进行格式化的是“过滤器”（filter），于是我们查阅 Angular 的 API 文档，发现在 filter 分类下有一个 date 过滤器，顾名思义，它应该是用来格式化日期的，于是我们把它加到 dateCreated 上：{{item.dateCreated|date}}。

不过，预览的时候我们发现它被格式化成了英文格式，对中文用户来说太不友好了。针对不同国家、语种的用户进行特殊处理的工作，叫作 i18n，它是 internationalization（国际化）的缩写，因为它中间正好有 18 个字符。好在，Angular 已经内置了一组 i18n 文件，中文简体的文件名叫作 angular-locale_zh-cn.js。我们把它从 github 的 Angular 项目中下载下来，然后复制到 app/libraries 目录里就可以了。当然，放到 app 的其他子目录中也可以，不过我们的约定是把除 bower 之外的第三方库都放到 app/libraries 目录下，遵循这个约定有助于提高程序的可维护性。

date 过滤器还可以带参数，如果我们要把它显示成“年-月-日 时:分”的自定义格式，那么就要这么写：{{item.dateCreated:date:'yyyy-MM-dd HH:mm'}}。

如果要进一步提高用户友好性，最好把时间显示成“刚才”、“一分钟前”、“昨天”等形式。这种情况下，我们可以自定义一个新的过滤器，来完成转换工作。由于这个过滤器

不涉及新的知识点，就不再赘述了，你可以自己尝试实现，还可以借助一个第三方时间库 momentjs 来降低复杂度。

1.5.2 实现过滤功能

接下来，我们给读者提供一个搜索框，可以用它对主贴标题、作者、发帖时间进行过滤。为了展示 Angular 的威力，我们先不借助后端，直接在前端实现过滤功能。即使在现实项目中，前端过滤也非常有用：在数据条数较少时，前端过滤可以大幅简化代码。在做原型的时候，它也具有出色的编程速度和表现力。

前几节我们说过，过滤器实际上应该叫转换器，常见的用法为：格式化和过滤。我们已经展示过它的格式化功能，现在我们就要用到它的过滤功能了。

我们先来看代码：

```
<div>
  <label for="_keyword">
    过滤条件:
  </label>
  <!-- 把用户输入的结果存到vm.filter.$变量中，供后面使用 -->
  <input id="_keyword" type="text" placeholder="请输入主题、作者或发帖时间的任意部分
    进行筛选" ng-model="vm.filter.$"/>
</div>
<table class="table table-hover table-striped">
  <thead>
    <tr>
      <th>主题</th>
      <th>作者</th>
      <th>发帖时间</th>
    </tr>
  </thead>
  <tbody>
    <!-- 使用Angular内置的名为filter的过滤器来实现过滤 -->
    <tr ng-repeat="item in vm.items | filter:vm.filter">
      <td>{{item.title}}</td>
      <td>{{item.poster}}</td>
      <td>{{item.dateCreated|date:'yyyy-MM-dd HH:mm'}}</td>
    </tr>
  </tbody>
</table>
```

可以看到，为了实现前端过滤，我并没有写任何 JavaScript 代码，只在 HTML 中通过很简单的几步就实现了。

加入一个输入框，用于接收关键字，并且存入 vm.filter 对象上名为 \$ 的属性中。

把这个 vm.filter 对象作为 filter 过滤器的参数，filter 过滤器会根据这个参数对 vm.items

数组进行筛选。把过滤的结果传给 `ng-repeat` 指令，Angular 会据此更新显示内容。

现在，是否已经体验到了“出色的编程速度”？我们再来看“表现力”：当你输入任何字符或汉字时，它都会对每个对象中的数据进行部分匹配，并且立即更新列表。如果不考虑大数据量下的性能问题，这显然是一种相当不错的体验。

在实际开发中，我一般以 2000 条普通查询结果作为区分用前端过滤还是用后端过滤的界限，即：当预期的最大数据条数小于 2000 时，用前端过滤，否则用后端过滤。如果后端逻辑复杂，查询代价高，或者单条数据量大，传输代价高，可以适当减小这个限额。当然，如果要想实现基于语义分析的模糊匹配，显然就该用后端过滤来实现了。

如果要想实现后端过滤功能该怎么写呢？

```
<div>
  <label for="_keyword">
    过滤条件:
  </label>
  <!-- 把用户输入的结果存到vm.filter.$变量中，通过vm.search函数发起请求，把结果存入
        vm.items变量中 -->
  <input id="_keyword" type="text" placeholder="请输入主题、作者或发帖时间的任意部分
        进行筛选" ng-model="vm.filter.$" ng-change="vm.search()" />
</div>
<table class="table table-hover table-striped">
...
  <!-- vm.items是经过后端过滤的数据 -->
  <tr ng-repeat="item in vm.items">
...
    </tr>
  </tbody>
</table>
```

其工作原理是这样的：

- ❑ 当用户进行输入时，`ng-change` 事件会被触发，并且把过滤条件作为参数传给后端。
- ❑ 后端解释这些过滤条件，然后返回符合条件的数据。
- ❑ 前端收到这些数据，就把它赋值给 `vm.items`。
- ❑ Angular 检测到 `vm.items` 变化了，就会据此更新界面。

如果要进一步改进，还可以修改 `ng-change` 的处理函数 `vm.search`，让它在用户连续输入的时候不发起网络请求，直到用户的输入告一段落时再发起。这个看起来酷炫的功能其实不必自己写，直接用第三方库 `lodash` 的 `debounce` 函数来完成就可以了，代码如下：

```
var search = function() {
  // 发起网络请求，把vm.filter作为过滤条件传给后端
};
```

```
// 包装后的结果供模板调用
vm.search = _.debounce(search, 500);
```

就这么简单！

lodash 是个很强大的 JavaScript 算法库，值得深入学习。

1.5.3 实现分页功能

分页和排序也是列表中常用的功能，它们同样有前端实现和后端实现两种方式。过滤、分页、排序的实现原理大同小异，在本节中，我们将一起实现前端分页功能，同时作为对上一节的复习。而后端分页功能和排序功能，我们不再讲解，读者朋友可以尝试自己实现。

我们先按照“模型驱动开发”的方式来思考分页：

- 我们有一大堆数据。
- 我们需要从中截取出一部分，显示给用户。

假设当前页码为 `page`（我们从 0 开始编号），每页条目数为 `pageSize`，则我们要截取的范围是：`page * pageSize` 到 `(page + 1) * pageSize`。

抛开形形色色的表现形式，分页的核心逻辑就是这么简单。

那么，我们如何做数据截取呢？我们首先要明白数据截取的本质就是一种过滤，其过滤条件是“起始索引号”和“截止索引号”。想通了这一点，答案就呼之欲出了：过滤器，我们的老朋友。

于是，我们设计了一个叫作 `page` 的过滤器：

```
angular.module('com.ngnice.app').filter('page', function () {
  return function (input, page, pageSize) {
    if (!input) {
      return input;
    }
    if (page < 0 || pageSize <= 0) {
      return [];
    }
    var start = page * pageSize;
    var end = (page + 1) * pageSize;
    return input.slice(start, end);
  };
});
```

由于这里涉及边界问题，容易出错，我们还要写一个单元测试来保障它的行为不会超出预期。

```
describe('filter > page >', function () {
  beforeEach(module('com.ngnice.app'));
```

```

var pageFilter;
// 注入器会忽略变量名前后的下划线
// 每个filter其实都是一个nameFilter的服务，我们可以用这种方式快速注入，不再引用$filter服务，这样表意性也更强
beforeEach(inject(function (_pageFilter_) {
    pageFilter = _pageFilter_;
})));

var items;
beforeEach(function () {
    items = [1, 2, 3, 4, 5, 6];
});
it('第一页为满页时', function() {
    expect(pageFilter(items, 0, 3)).toEqual([1,2,3]);
});
it('第一页为不满页时', function() {
    expect(pageFilter(items, 0, 7)).toEqual([1,2,3,4,5,6]);
});
it('最后一页为满页时', function() {
    expect(pageFilter(items, 1, 3)).toEqual([4,5,6]);
});
it('最后一页为不满页时', function() {
    expect(pageFilter(items, 1, 4)).toEqual([5,6]);
});
it('大于最大页码时', function() {
    expect(pageFilter(items, 2, 4)).toEqual([]);
});
it('小于最小页码时', function() {
    expect(pageFilter(items, -2, 4)).toEqual([]);
});
});

```

这里唯一需要注意的是引用 filter 的一个小技巧，它有助于我们理解 Angular 的工作原理。

官方推荐的方式是注入 `$filter`，然后调用 `$filter('page')` 来获得 page 过滤器。事实上，在 Angular 中，每个 filter 都是一个 service，只是它的名字加上了 Filter 后缀。比如 page 过滤器对应的服务名就是 `pageFilter`。这两者是等价的，但由于我讨厌在代码中硬编码字符串，所以一般使用直接注入 service 的方式。

使用这个过滤器就很简单了，我们先在界面中通过一个输入框来输入页码，并且规定每页大小为 5 条，然后我们再想办法美化它。这也是在遵循敏捷方法：每一步只做一件事。

```

<table class="table table-hover table-striped">
...
  <tr ng-repeat="item in vm.items | filter:vm.filter | page:vm.page.index:5">
...

```

```

    </tr>
  </table>
  <label>
    页号:
    <input type="number" ng-model="vm.page.index"/>
  </label>

```

剩下的工作就是分页标签的美化，最简单的方式是使用第三方库：angular-bootstrap 中的 pagination 指令，它已经实现得相当不错了。如果想要自己写，这个指令也可以作为一个复杂度适中的练习题，但是要记住以下几点：

- ❑ 面向模型编程。
- ❑ 测试驱动开发。
- ❑ 先保障交互逻辑，再调整细节。

做完练习之后可以与 angular-bootstrap 的 pagination 指令（注意，它的起始页码是 1 而不是 0），以及我随书源码中所实现的分页标签作对比。如果对我的实现方式感到惊讶，不用困惑，也不用着急，在下一节中，我会详细讲解这种实现思路。

1.5.4 实现主题树

把主题和回复按照回复关系组织起来，就构成了一棵“主题树”。相比传统论坛列表形式的组织方式，主题树可以更加直观地展现出回复关系，并且更方便按照分组进行操作，比如把一个回复及其后续回复收藏起来或删除。这些特性不太适合人气火爆的论坛，但是比较适合我们这个小型的读者交流社区。

树在各种前端框架中都不是一个小话题，为了避免一次引入大量新概念，我们拆成两步来实现它：首先，做一个固定层数，带有折叠和级联选择功能的树；然后，把它扩展成不限层数，可以任意递归的树。

我们先来构造模拟数据：

```

angular.module('com.ngnice.app').controller('ThreadTreeCtrl', function ThreadTreeCtrl() {
  var vm = this;
  vm.items = [
    {
      id: 1,
      title: '这是第一个主题',
      poster: '雪狼',
      dateCreated: '2015-02-19T00:00:00',
      items: [
        {
          id: 11,
          title: '这是第一个回复',

```

```

        poster: '雪狼',
        dateCreated: '2015-02-19T00:00:01',
        items: [
            {
                id: 111,
                title: '回复1.1',
                poster: '破狼',
                dateCreated: '2015-02-19T00:01:00'
            },
            {
                id: 112,
                title: '回复1.2',
                poster: '破狼',
                dateCreated: '2015-02-19T00:01:30'
            }
        ]
    },
    {
        id: 12,
        title: '这是第二个回复',
        poster: '雪狼',
        dateCreated: '2015-02-19T00:00:03'
    }
]
},
{
    id: 2,
    title: '这是第二个主题, 含有字母abcd和数字1234',
    poster: '破狼',
    dateCreated: '2015-02-19T15:00:00'
}
];
});

```

对于我们的目标来说，三层数据已经有了足够的代表性。接下来，我们就把它展示到界面上：

```

<ul ng-if="vm.items">
  <li ng-repeat="item1 in vm.items">
    {{item1.title}}
    <ul ng-if="item1.items">
      <li ng-repeat="item2 in item1.items">
        {{item2.title}}
        <ul ng-if="item2.items">
          <li ng-repeat="item3 in item2.items">
            {{item3.title}}
          </li>
        </ul>
      </li>
    </ul>
  </li>
</ul>

```



```

    </li>
  </ul>

```

显然，它是个递归结构，但是由于 Angular 的一些限制，我们没法直接用指令递归的方式实现它，而要借助一些特殊的技巧，这些技巧涉及一些 Angular 高级知识，我们把它留到后面再讲。

接下来，我们要实现折叠功能。所谓折叠，就是把子节点隐藏起来。但这是一个面向 DOM 的思维方式，我们把它换成面向模型的思维方式：下级子节点的“显示与否”取决于当前节点的一个 Model，即“是否折叠”(\$folded)。

现在我们用代码来表达这个思路：

```

<ul ng-if="vm.items">
  <li ng-repeat="item1 in vm.items">
    <!-- $folded 表示当前节点是否折叠的 Model，点击是切换它 -->
    <div ng-click="item1.$folded = !item1.$folded">
      <!-- 有子节点时才显示加减号 -->
      <span ng-if="item1.items">
        <!-- 未折叠的显示减号 -->
        <span ng-if="!item1.$folded">-</span>
        <!-- 已折叠的显示加号 -->
        <span ng-if="item1.$folded">+</span>
      </span>
      {{item1.title}}
    </div>
    <!-- 子节点的显示与否，取决于当前节点的 Model: $folded -->
    <ul ng-if="item1.items && !item1.$folded">
      <li ng-repeat="item2 in item1.items">
        <div ng-click="item2.$folded = !item2.$folded">
          <span ng-if="item2.items">
            <span ng-if="!item2.$folded">-</span>
            <span ng-if="item2.$folded">+</span>
          </span>
          {{item2.title}}
        </div>
        <ul ng-if="item2.items && !item2.$folded">
          <li ng-repeat="item3 in item2.items">
            {{item3.title}}
          </li>
        </ul>
      </li>
    </ul>
  </li>
</ul>

```

我们没有写一句 JavaScript 代码，只修改了一下 HTML 模板就实现了折叠功能。这里之所以用 \$ 开头，主要有两个目的，一是减小和原有数据冲突的可能，二是如果这些数据

通过 \$http 或 \$resource 提交给服务器，它们所调用的 angular.toJson() 函数会忽略所有以 \$ 开头的属性，这样我们扩展出来的属性就不会被提交了。这带来一个额外的小便利：阅读的时候一看到是 \$ 开头的，也就知道是扩展出来的属性了。

但是把 item1.\$folded = !item1.\$folded 这类代码混合在 HTML 中是不够干净的，我们要把它重构出去。重构的方式有多种。最明显的一种是把整棵树作成指令，但这也是最坏的一种，事实上，这是在用写 jQuery 组件的思路写 Angular，它会将视图和模型紧密地混合在一起，难以复用和测试。这是一种坏习惯，我就不再演示了。

好一些的是把切换的代码封装成函数，放在 controller 中：vm.toggle = function(item) {item.\$folded = !item.\$folded; }，在视图中直接调用 vm.toggle(item1)。这种情况下，HTML 中倒是“干净”了，但控制器又“脏”了。而且一旦我们加入比折叠更复杂的代码，那么这些代码将不得不写很多份。

更好的方式是作成服务。我们真正要实现的逻辑，其实是对模型中的 \$folded 属性进行操作，然后通过一些指令来告诉 Angular 如何让视图和模型保持同步。所以，我们只要给模型加上 \$foldToggle() 之类的操作方法和 \$isFolded() 之类的判断函数就可以了。我们理想的使用方式是这样的：

```
<ul ng-if="vm.$hasChildren()">
  <li ng-repeat="item1 in vm.items">
    <!-- 点击时调用切换折叠状态的函数 -->
    <div ng-click="item1.$foldToggle()">
      <!-- 判断是否有子节点 -->
      <span ng-if="item1.$hasChildren()">
        <!-- 隐藏私有变量$folded, 改用$isFolded函数 -->
        <span ng-if="!item1.$isFolded()"></span>
        <span ng-if="item1.$isFolded()"></span>
      </span>
      {{item1.title}}
    </div>
    <ul ng-if="item1.$hasChildren() && !item1.$isFolded()">
      ...
    </ul>
  </li>
</ul>
```

显然，item1 本身是不存在这三个函数的，但是我们可以加上它：

```
angular.module('com.ngnice.app').controller('ThreadTreeCtrl', function
  ThreadTreeCtrl(tree) {
    var vm = this;
    vm.items = ...
    tree.enhance(vm.items);
  });
```

这里的 `tree` 是个我们所期望的一个工具类服务，它负责对数据进行强化，给它添加这些函数。

接下来，我们就来实现这个工具类服务：

```
angular.module('com.ngnice.app').service('tree', function Tree() {
  var self = this;
  // 强化条目
  var enhanceItem = function (item, childrenName) {
    item.$hasChildren = function() {
      var subItems = this[childrenName];
      return angular.isArray(subItems) && subItems.length;
    };
    item.$foldToggle = function () {
      this.$folded = !this.$folded;
    };
    item.$isFolded = function () {
      return this.$folded;
    };
  };
  // 对传进来的数据进行强化
  this.enhance = function (items, childrenName) {
    if (angular.isUndefined(childrenName)) {
      childrenName = 'items';
    }
    angular.forEach(items, function (item) {
      enhanceItem(item, childrenName);
      // 如果有子节点，则递归处理
      if (item.$hasChildren()) {
        self.enhance(item[childrenName], childrenName);
      }
    });
    return items;
  };
});
```

这样用起来还是不够方便，而且把强化数据的代码也都放进了控制器中，还是不够干净。

还有什么方式能让它更简单点呢？有请我们的老朋友——过滤器：

```
<ul ng-if="vm.$hasChildren()">
  <li ng-repeat="item1 in vm.items|tree">
    ...
  </li>
</ul>
```

我们记得过滤器的本质是数据转换，并且示范过它的两大主要用法：格式化和筛选。现在则是一个不常用但很有用的用法：强化，也就是为数据添加成员和方法。

我们期望有这样一个过滤器，接着就来实现它，但是我们不用从头写代码，直接使用已经写好的 `tree` 服务就可以了：

```
angular.module('com.ngnice.app').filter('tree', function (tree) {
  return function (items, childrenName) {
    tree.enhance(items, childrenName);
    return items;
  };
});
```

这是在编程中常用的一种方式：先想好准备怎么用，然后再逐步去实现它。这其实是“测试驱动开发”的一个退化变种。由于折叠的逻辑非常简单，所以我们直接用这种方式就够用了。

接下来，我们要实现一个更复杂的功能，这个功能我们就要用“测试驱动开发”的方式来实现了。

我们先来详细描述一下“级联选择”这个需求：

- ☐ 每个节点前有一个复选框。
- ☐ 选中父节点时，自动选中所有子节点。
- ☐ 选中所有子节点时，自动选中父节点。
- ☐ 取消选中所有子节点时，自动取消选中父节点。
- ☐ 部分选中子节点时，父节点显示为“待定”状态。

这次，我们不再像以前那样先写视图，而是直接来设计模型。“以模型为核心的思维方式”是你从新手成为 Angular 专家的一个重要特征。

像我们设计折叠功能时看到的一样，每个节点需要有两个方法：`$checkToggle()` 和 `$isChecked()`。接着，为了支持待定状态，我们还需要一个方法：`$isIndeterminate()`。由于 `$checkToggle()` 对当前状态的依赖比较大，测试的时候很不方便，和 `checkbox` 配合的时候也不理想。为了提供更高的可控性，我们再增加 `$check()`、`$uncheck()` 和 `$setCheckState` 函数。

供视图中使用的接口先定下这 6 个就足够了，其他的方法在我们编程的过程中自然会显现出来，现在不用急。

接下来，我们就要来实现它了。我们要在前面所写的 `tree` 服务的基础上改进一下，实现“级联选择”功能。

级联选择的逻辑远比折叠的逻辑复杂，那么，是时候请出我们的必杀器：“测试驱动开发”(TDD)了！

在 FrontJet 的支持下，TDD 是一种享受。我们的 `fj serve` 命令会同步开启 TDD 模式，只要留意下它的控制台窗口，就可以开始我们的 TDD 之旅了。

首先，我们写一个针对 tree service 的单元测试文件，按照约定，我们把它命名为 tree.test.js，.test.js 是我们给单元测试文件加的后缀，用于和程序文件相对应的同时区分开文件名。文件内容如下：

```
// 用describe函数对测试进行分组
describe('tree service >', function () {
  // 每次执行前加载app模块，它依赖的模块也会被级联加载
  beforeEach(module('com.ngnice.app'));
  var tree;
  // 这里可以注入服务或控制器，注入器会忽略名称两端的下划线
  beforeEach(inject(function (_tree_) {
    tree = _tree_;
  }));
  // 把我们前面写的需求规约逐个加进来
  it('选中父节点时，自动选中所有各级子节点', function () {
  });
});
```

当我们保存这个文件时，FrontJet 中集成的 Karma 会自动跑一遍单元测试，并且在控制台给出反馈：

```
INFO [watcher]: Changed file "/Users/.../services/utils/tree.test.js".
PhantomJS 1.9.8 (Mac OS X): Executed 1 of 1 SUCCESS (0.001 secs / 0.016 secs)
```

我们看到，测试执行成功了，这是测试驱动开发的“绿灯”状态。接下来，我们先构造数据，并把它们都变为“红灯”状态，以免将来遗漏了需求。

```
describe('tree service >', function () {
  beforeEach(module('com.ngnice.app'));
  var tree;
  // 这里可以注入服务或控制器，注入器会忽略名称两端的下划线
  beforeEach(inject(function (_tree_) {
    tree = _tree_;
  }));
  var treeData;
  // 定义两个供测试用的工具函数
  var getFlattenData = function (items) {
    var result = items || [];
    angular.forEach(items, function (item) {
      result = result.concat(getFlattenData(item.items));
    });
    return result;
  };
  var findById = function (id) {
    return _.findWhere(getFlattenData(treeData), {id: id});
  };
  // 构造尽量少但足够有代表性的模拟数据
  beforeEach(function () {
```

```

treeData = [
  {
    id: 1,
    items: [
      {
        id: 11,
        items: [
          {
            id: 111
          },
          {
            id: 112
          }
        ]
      },
      {
        id: 12
      }
    ]
  }
];
tree.enhance(treeData);
});
it('选中父节点时, 自动选中直接子节点', function () {
  // 动作: 选中中间节点
  findById(11).$check();
  expect(findById(111).$isChecked()).toBeTruthy();
  expect(findById(112).$isChecked()).toBeTruthy();
});
it('选中祖先节点时, 自动选中间接子节点', function() {
  // 动作: 选中根节点
  findById(1).$check();
  // 期待: 子节点被选中
  expect(findById(11).$isChecked()).toBeTruthy();
  expect(findById(12).$isChecked()).toBeTruthy();
  // 期待: 孙子节点被选中
  expect(findById(111).$isChecked()).toBeTruthy();
  expect(findById(112).$isChecked()).toBeTruthy();
});
});

```

这段代码很长, 不过做的事情并不是很多。我们先定义了测试数据, 然后用代码定义了一系列断言来落实需求。每个测试的组成大致是: 做一个动作, 检查所期待的状态。

注意, 我们把这条需求拆分成了两个: 选中父节点时自动选中“直接子节点”, 选中祖先节点时自动选中“间接子节点”。把它们合在一个测试中当然也是可以的, 不过, 作为单元测试的一条重要原则, 我们应该尽可能让每个单元测试只做一件事。

我们定义完了这一系列规约, 但是还没有实现 `tree` 服务, 所以我们看到如下错误:

```

INFO [watcher]: Changed file "/Users/.../services/utils/tree.test.js".
PhantomJS 1.9.8 (Mac OS X) tree service > 选中父节点时, 自动选中直接子节点 FAILED
  TypeError: 'undefined' is not a function (evaluating 'findById(11).$check()')
    at /Users/.../services/utils/tree.test.js:45
    ...
PhantomJS 1.9.8 (Mac OS X) tree service > 选中祖先节点时, 自动选中间接子节点 FAILED
  TypeError: 'undefined' is not a function (evaluating 'findById(1).$check()')
    at /Users/.../services/utils/tree.test.js:51
    ...
PhantomJS 1.9.8 (Mac OS X): Executed 2 of 2 (2 FAILED) ERROR (0.001 secs / 0.014 secs)

```

现在两个测试全部失败了，这是测试驱动开发中的下一个状态：红灯。测试驱动开发过程中会期待红绿灯交替出现。接下来，我们就把它变为绿灯状态。

我们先修改下 tree 服务：

```

var enhanceItem = function (item, childrenName) {
  ...
  // 递归设置
  var setCheckState = function(node, checked) {
    node.$checked = checked;
    if (node.$hasChildren()) {
      angular.forEach(node[childrenName], function(subNode) {
        setCheckState(subNode, checked);
      });
    }
  };
  item.$check = function() {
    setCheckState(item, true);
  };
  item.$isChecked = function() {
    return this.$checked;
  };
};

```

当程序文件发生变化时，单元测试也会被自动执行。控制台中会出现下列输出：

```

INFO [watcher]: Changed file "/Users/.../app/services/utils/tree.js".
PhantomJS 1.9.8 (Mac OS X): Executed 2 of 2 SUCCESS (0.001 secs / 0.011 secs)

```

于是，我们又回到了“绿灯”状态，这就是一个 TDD 工作循环。对于逻辑比较复杂的功能，我们可能还要经过多个红绿灯才能完成一个工作循环。

接下来，我们按照上面的工作方式，把另外三项需求也逐步加进来，我不再把代码摘录到书中，只把最终的单元测试列在这里，大家可以尝试自己实现。

```

it('选中所有子节点时, 自动选中父节点', function () {
  findById(111).$check();
  findById(112).$check();

```



```

    expect(findById(11).$isChecked()).toBeTruthy();
    findById(12).$check();
    expect(findById(1).$isChecked()).toBeTruthy();
  });
  it('取消选中所有子节点时, 自动取消选中父节点', function () {
    // 先全部选中
    findById(1).$check();
    expect(findById(112).$isChecked()).toBeTruthy();
    // 逐个反选
    findById(111).$uncheck();
    findById(112).$uncheck();
    expect(findById(11).$isChecked()).toBeFalsy();
    findById(12).$uncheck();
    expect(findById(1).$isChecked()).toBeFalsy();
  });
  it('部分选中子节点时, 父节点显示为“待定”状态', function () {
    findById(111).$check();
    expect(findById(11).$isIndeterminate()).toBeTruthy();
    expect(findById(1).$isIndeterminate()).toBeTruthy();
    // 子节点本身和没有子节点的节点不受影响
    expect(findById(111).$isIndeterminate()).toBeFalsy();
    expect(findById(12).$isIndeterminate()).toBeFalsy();
  });
  it('全部选中子节点时, 父节点为“非待定”状态', function () {
    findById(111).$check();
    expect(findById(11).$isIndeterminate()).toBeTruthy();
    findById(112).$check();
    expect(findById(11).$isIndeterminate()).toBeFalsy();
  });
  it('$checkToggle会切换当前节点的选中状态', function () {
    var node = findById(111);
    node.$checkToggle();
    expect(node.$isChecked()).toBeTruthy();
    node.$checkToggle();
    expect(node.$isChecked()).toBeFalsy();
  });
});

```

我们实现完服务之后, 再把它应用到页面中:

```

<ul ng-if="vm.items">
  <li ng-repeat="item1 in vm.items">
    <div>
      <span ng-if="item1.$hasChildren()" ng-click="item1.$foldToggle()">
        <span ng-if="!item1.$isFolded()">-</span>
        <span ng-if="item1.$isFolded()">+</span>
      </span>
      <label>
        <!-- 在数据变化时, 调用服务中的相应函数来更新父节点、子节点的状态 -->
        <!-- 通过自定义指令bf-check-indeterminate来把模型更新到界面上 -->
        <input type="checkbox" bf-check-indeterminate="item1.$isIndeterminate()" ng-model="item1.$checked" ng-change="item1.$setCheckState

```

```

        (item1.$checked) "/>
        {{item1.title}}
    </label>
</div>
<ul ng-if="item1.$hasChildren() && !item1.$isFolded()">
    <li ng-repeat="item2 in item1.items">
        <div>
            <span ng-if="item2.$hasChildren()" ng-click="item2.$foldToggle()">
                <span ng-if="!item2.$isFolded()">-</span>
                <span ng-if="item2.$isFolded()">+</span>
            </span>
            <label>
                <input type="checkbox" bf-check-indeterminate="item2.$isIndeterminate()" ng-model="item2.$checked" ng-change="item2.$setCheckState(item2.$checked)" />
                {{item2.title}}
            </label>
        </div>
        <ul ng-if="item2.$hasChildren() && !item2.$isFolded()">
            <li ng-repeat="item3 in item2.items">
                <label>
                    <input type="checkbox" bf-check-indeterminate="item3.$isIndeterminate()" ng-model="item3.$checked" ng-change="item3.$setCheckState(item3.$checked)" />
                    {{item3.title}}
                </label>
            </li>
        </ul>
    </li>
</ul>
</li>
</ul>
</ul>

```

现在我们还缺一个 `bf-check-indeterminate` 指令，它的实现代码如下：

```

angular.module('com.ngnice.app').directive('bfCheckIndeterminate', function bfCheckIndeterminate() {
    return {
        restrict: 'A',
        link: function (scope, element, attrs) {
            scope.$watch(
                function () {
                    return scope.$eval(attrs.bfCheckIndeterminate);
                },
                function (value) {
                    angular.forEach(element, function (DOM) {
                        DOM.indeterminate = value;
                    });
                }
            );
        }
    };
});

```

```
    };
  });
```

测试代码如下：

```
describe('checkIndeterminate指令 >', function () {
  beforeEach(module('com.ngnice.app'));
  var $compile;
  var $rootScope;
  beforeEach(inject(function (_$compile_, _$rootScope_) {
    $compile = _$compile_;
    $rootScope = _$rootScope_;
  }));
  it('使用常量作为默认值', function () {
    var scope = $rootScope.$new(true);
    var dom = $compile('<input type="checkbox" bf-check-indeterminate="true" />')(scope);
    scope.$digest();
    expect(dom[0].indeterminate).toBeTruthy();
  });
  it('通过代码修改变量', function() {
    var scope = $rootScope.$new(true);
    scope.checked = true;
    var dom = $compile('<input type="checkbox" bf-check-indeterminate="checked" />')(scope);
    scope.$digest();
    expect(dom[0].indeterminate).toBeTruthy();
    scope.checked = false;
    scope.$digest();
    expect(dom[0].indeterminate).toBeFalsy();
  });
});
```

可以看到，我们只响应了一个事件、写了一个很简单的自定义指令，就完成了模型和视图的绑定。

在我们实现整个功能的过程中，用 TDD 的方式覆盖了绝大部分交互逻辑，对这些复杂度较高、不太有把握的部分，建立了信心。然后，通过非常简单的几句代码，就把模型和视图绑定在一起了。

同时，可以看出，通过 TDD 方式写复杂逻辑的效率比传统方式高多了：如果是传统方式，我们得写一些代码，然后切换到浏览器中，操作几下，看结果是否符合预期，还要留意我们的新改动是否破坏了已经“完成”的部分工作逻辑；在 TDD 方式下，我们写好测试，然后写程序代码，随时保存、随时看测试结果，等所有测试都通过的时候，我们就知道核心逻辑已经完成了。

回忆我们曾经写过的这些指令，会发现一个共同的特点：它们都非常简单，只有少数

几行语句，这些语句的作用仅仅是把一个很简单的 Model 的变更情况表现到视图中去。曾经有很多人问我如何对指令进行单元测试，我给出的答案是：不要测试指令。当然，这只是一个“当头棒喝”式的答案，并不意味着无法或没必要对指令进行单元测试。我要表达的核心意思是：用服务代替指令，让指令尽可能保持简约。理解这一点，也是转换到 Angular “模型驱动”思维的重要一步。

1.5.5 实现递归主题树

接下来，我们就要进入 Angular 指令的“深水区”了。在这一节，我们会开始使用一些高级技巧来实现递归树。

指令递归嵌套自身会导致停止响应，这是我们要解决的核心问题。具体的分析过程我将在第 3 章“背后的原理”中讲解。这里我直接给出答案：指令的解析分成两个阶段：编译（compile）和链接（link），如果指令嵌套了，那么编译过程会被不断执行，刚执行完一次又触发一次，相当于出现了无限循环。

原因找到了，大体的解决思路也就有了：只要我们不直接嵌套自身，而是通过其他指令间接嵌套一次。当然，通过其他指令间接嵌套也同样是有点问题的，除非我们能错开执行时间，比如把间接嵌套的过程放到链接阶段：生成一个 live DOM，然后把生成的 live DOM 动态插入到期望的位置。这种方式，我们曾在“错误信息提示”指令中使用过，还记得吧？不过和上次不同的是，这次还有几个新问题：

- ❑ 准备如何使用？
- ❑ 如何取得模板？
- ❑ 如何取得数据？
- ❑ 如何生成子节点？

我们先来设想预期的用法。

首先，我们肯定需要两个指令来配合：一个上级指令，用来取得模板和数据；一个下级指令，用来渲染子节点。

```
<ul>
  <!-- 取得数据源，并且赋给$dataSource变量 -->
  <li bf-template="vm.items" ng-repeat="node in $dataSource">
    {{node.title}}
    <ul>
      <li bf-recurse="node.items"></li>
    </ul>
  </li>
</ul>
```

然后，我们来解决“如何取得模板”的问题：

对于递归指令，顾名思义，上下级的模板是完全相同的。所以，我们只要把上级节点的模板保存起来，将来用它来编译下级节点就行了。所以，我们写一个指令，其作用就是把所在的节点的 HTML 取下来，存到一个 scope 变量里。不过还有一个问题：在链接阶段，所在的节点已经变成了 live DOM，我们取不到原始的 HTML 模板。解决方案很简单，那就是在编译阶段取模板，这个阶段的节点还是静态的。代码如下：

```
angular.module('com.ngnice.app').directive('bfTemplate', function bfTemplate() {
  return {
    restrict: 'A',
    compile: function (element) {
      var template = element[0].outerHTML;
      return function (scope) {
        scope.$template = template;
      };
    }
  };
});
```

接下来，是解决“如何取得数据”的问题。

递归指令的数据也同样是递归的。这就要求指令所依赖的 scope 结构也是递归的，显然，如果直接把 controller 的 scope 用于递归指令，那么将对 controller 产生严重的限制。我们需要一个独立的 scope 结构，它的数据来源于 controller，但是最终我们要把它赋值给我们定义的内部数据结构。从 controller 中取数据的方式我们也同样在“错误信息提示”指令中使用过：scope.\$eval。

于是，我们的指令修改为：

```
angular.module('com.ngnice.app').directive('bfTemplate', function bfTemplate() {
  return {
    restrict: 'A',
    // 提高优先级，原因稍后讲解
    priority: 2000,
    compile: function (element) {
      var template = element[0].outerHTML;
      return function (scope, element, attrs) {
        scope.$template = template;
        if (!scope.$dataSource) {
          scope.$dataSource = scope.$eval(attrs.bfTemplate);
        }
      };
    }
  };
});
```

不过，还有一个问题，`ng-repeat` 等指令具有很高的优先级，所以它会被首先执行，而这个时候 `bf-template` 指令还没有机会执行，所以 `$dataSource` 变量是空值。我们必须让 `bf-template` 指令在 `ng-repeat` 之前执行，在 Angular 中，这个机制叫作优先级（priority）。即，优先级高的指令会先执行。而 `ng-repeat` 的优先级是 1000，于是，我们把 `bf-template` 的优先级指定为 2000，确保它先执行。

最后，我们再来解决“如何生成子节点”的问题。

```
angular.module('com.ngnice.app').directive('bfRecurse', function bfRecurse
($compile) {
  return {
    restrict: 'A',
    link: function (scope, element, attrs) {
      // 建立一个独立作用域
      var subScope = scope.$new(true);
      // 取得本节点的数据，这个数据将被传给模板
      subScope.$dataSource = scope.$eval(attrs.bfRecurse);
      // 生成live DOM
      var dom = $compile(scope.$template)(subScope);
      // 用live DOM替换当前节点
      element.replaceWith(dom);
    }
  };
});
```

实现类似功能的还有一个第三方指令 `angular-tree-repeat`，当初我第一次看到这个指令，就为它巧妙而漂亮的实现方式而惊艳，本书的实现方式也是受其启发改进来的。有兴趣的也可以对照两者的代码，体会一下我的改进思路。

1.5.6 实现“查看详情”功能

接下来，我们把递归主题树的功能扩展一下，给每个节点加一个“查看详情”链接。用户点击这个链接的时候，就跳转到“查看详情”页面。

本节很简短，主要着眼于示范如下特性：

- ❑ 在路由中使用参数。
- ❑ 通过 `ui-sref` 指令构建链接。

我们首先要实现“查看详情”页面，这包括三步：在 `router.js` 中添加路由定义，实现 controller，实现模板。

新增的路由定义如下：

```
$stateProvider.state('thread.show', {
  // 声明一个id参数和一个title参数
```

```
url:('/:id/show?title&poster',
templateUrl: 'controllers/thread/show.html',
controller: 'ThreadShowCtrl as vm'
});
```

可以看到，这个路由定义中声明了两个路由参数，一个是行内的 id 参数，一个是 query 参数 title。

接下来就是在 controller 中使用了，具体代码如下：

```
angular.module('com.ngnice.app').controller('ThreadShowCtrl',function Thread-
ShowCtrl($stateParams) {
var vm = this;
vm.id = $stateParams.id;
vm.title = $stateParams.title;
vm.poster = $stateParams.poster;
});
```

这里的关键是 \$stateParams 服务。凡是从路由中传过来的参数都可以通过这个服务访问，但是这里有一个常见的坑是：只有在路由定义中声明过的参数才能使用，比如这里只声明了两个 query 参数：title 和 poster。对于 URL：/thread/1/show?title=abc&poster=wzc&dateCreated=2015-03-01，接收到的参数中，title 和 poster 都正常，而 dateCreated 参数为空，这就是因为前面没有声明它。

详情页中我们只实现一个最简单的界面：

```
编号: {{vm.id}}<br/>
标题: {{vm.title}}<br/>
作者: {{vm.poster}}
```

接下来，我们要做的就是从主题树中链接到详情页：

```
<a ui-sref="thread.show({id: node.id, title: node.title, poster: node.poster})">查看</a>
```

ui-sref 是个 angular-ui-router 提供的指令，其参数是一个路由的名字，而括号中的参数是一个由各个路由参数组成的对象。如果使用系统内置的 ngRoute 作为路由库，那么就要自己拼接 ng-href 了，如：<a ng-href="#/thread/{{node.id}}/show?title={{node.title}}&poster={{node.poster}}"> 查看 。虽然也不错，但从可读性和灵活性方面，要比前者差一些。

1.6 实现 AOP 功能

至此，实现路由页面时用到的技术我们已经基本示范过了，接下来我们将开始实现一些高级功能。这些功能具有全局性的影响——基本上每个路由都会涉及它，如果我们把它嵌入到每个路由的实现里，那么代码中将出现大量的重复，编写和维护将会变成噩梦。这

类功能，我们称其为“ AOP 功能”，也就是“面向切面功能”（形象点说：路由是竖着并列在一起的，AOP 功能则像一个平台一样支撑着它们）。最典型的就是“登录”和“错误处理”。接下来，我们先来实现“登录”。

1.6.1 实现登录功能

传统的登录过程是这样的：

- ❑ 浏览器访问一个网址。
- ❑ 服务器判断这个网址是否需要登录才能访问。
- ❑ 如果需要，则给浏览器回复一个 Redirect 头。
- ❑ Redirect 的地址是登录页，地址中还带有一个登录成功后的回调地址。
- ❑ 浏览器把登录页显示给用户，用户输入有效的用户名密码之后提交到服务器。
- ❑ 服务器检查用户名密码是否有效，如果有效则给浏览器回复 Redirect 头来跳转到回调地址。

这样用户就完成了登录。

单页面应用（SPA）下固然也可以用传统方式进行登录，但是在 Ajax 的支持下，还有更为友好的登录方式，这也是一些大型网站中常用的方式。

实现 Ajax 登录的核心思想是：只需要保护后端 API 就够了，通常不用保护前端 URL。大致的登录过程如下：

- ❑ 前端发起一个请求。
- ❑ 服务端判断这个网址是否需要登录才能访问。
- ❑ 如果需要，则给前端发回一个状态码 401（未认证身份，即：未登录）。
- ❑ 前端收到 401 状态码，则弹出一个对话框，提示用户输入用户名和密码。
- ❑ 用户输入用户名和密码之后提交。
- ❑ 前端发起一个登录请求，并等待登录成功。
- ❑ 登录成功后，前端重新发送刚才被拒绝的请求。

这种方式的优点如下：

- ❑ 把控制逻辑完全交给前端，后端只要提供“纯业务 API”就够了，这样前后端的分工非常明确。
- ❑ 完全在当前页面中执行，不用多次加载页面，用户操作非常顺畅。
- ❑ 在 Promise 机制的支持下，登录过程对前端的应用逻辑可以是完全透明的（调用 API 的代码不需要区分中间是否发生过登录，也不需要对此做任何处理）。

原理清楚了，接下来我们就来考虑代码实现。

在 Angular 中有一种机制叫作拦截器（interceptor），它是 \$http 的一个扩展点，它类似于后端框架中的过滤器（filter）机制。它会对每个 \$http 请求的发送和接收过程进行过滤，由于 \$http 是 Angular 中的底层服务，所以基于它的 \$resource 等服务也同样会受到影响。

在 Angular 的官方文档中给出的拦截器范例代码如下所示：

```
// 把拦截器注册为一个factory型服务
angular.module('com.ngnice.app').factory('myHttpInterceptor', function($q,
    dependency1, dependency2) {
    return {
        // 发起请求之前的拦截函数。可选。
        'request': function(config) {
            // config就是执行$http时的参数，如 {url: '...', method: 'get', ...}
            // 可以返回原config、新config、新Promise或通过返回$q.reject(rejection)来阻止
            // 发起请求
            return config;
        },
        // 请求被其他拦截器阻止时的拦截函数。可选。
        'requestError': function(rejection) {
            // 如果这个错误是可以补救的，则返回一个响应对象或新的Promise，稍后会详细讲解这种机制
            if (canRecover(rejection)) {
                return responseOrNewPromise
            }
            return $q.reject(rejection);
        },
        // 收到服务器给出的成功回应时的拦截函数。可选。
        'response': function(response) {
            // 成功时可以直接返回响应对象，也可以返回一个新的Promise。
            return response;
        },
        // 收到服务器给出的失败回应时的拦截函数。可选。
        'responseError': function(rejection) {
            // 如果这个错误是可以补救的，则返回一个响应对象或新的Promise。
            if (canRecover(rejection)) {
                return responseOrNewPromise
            }
            return $q.reject(rejection);
        }
    };
});

// 把这个服务追加到$httpProvider.interceptors中，以便$http给它们进行拦截的机会。
$httpProvider.interceptors.push('myHttpInterceptor');

// 也可以不定义服务，直接把一个函数push进去。建议不要用这种写法，灵活性差，而且容易干扰前端工具
```

链的依赖注入处理。

```
$httpProvider.interceptors.push(function($q, dependency1, dependency2) {
  return {
    'request': function(config) {
      // same as above
    },
    'response': function(response) {
      // same as above
    }
  };
});
```

这段代码其实很简单：先声明一个对象，这个对象有四个拦截函数，它们分别在四个不同的情况下被调用，让你有机会进行处理，然后把这个对象注册进 \$http 中即可。

这段代码的难点在于这里有两个很重要的新概念：**Provider** 和 **Promise**。

我们先不用深究 **Provider** 的实现原理，第 2 章“概念介绍”中会讲到，这里我们只需要了解它的用途是对特定的服务进行配置，比如：\$httpProvider 就是对 \$http 服务进行配置的，\$locationProvider 就是对 \$location 服务进行配置的。这里的 interceptors 就是 \$http 的一种配置项。

Promise 则要难一些，简单地说，**Promise** 就是一个承诺，比如，我们调用 \$http 时的代码：

```
$http.get('/api/readers').then(function(data) {
  ...
});
```

这里的 \$http.get 函数返回的对象就是一个 **Promise**（承诺）表示“我现在先给你一个承诺，我现在不能给你答案，但是一旦条件具备了我就会履行这个承诺”。这个 **Promise** 有一个成员函数叫 **then**。我们可以把一个回调函数传给 **then** 函数，它会把这个回调函数保存在 **Promise** 对象中。当 \$http.get 获得了从后端返回的数据时，我们传进去的这个回调函数就会被执行，并且把来自服务器的响应对象传给我们的回调函数。

Promise 在 **Angular** 中是个非常重要而且非常有用的概念，在实际项目中会大量用到。如果仍然没有理解，可以翻到第 2 章“概念介绍”中通过生活中的一个例子了解其基本概念再往下读。

接下来，我们回到主题，看看拦截器的工作原理。

由于 **Angular** 中 \$http 和 interceptors 部分的代码涉及 **Promise** 的高级用法，读起来有点绕。这里用最直观的伪代码来表明其工作原理：

```

// 演示两个interceptor时的工作原理
var interceptors = [interceptor1, interceptor2];
var $http = function(config) {
  // request阶段, 先注册的先执行
  var requestPromise1 = interceptor1.request(config);
  // 第一个interceptor.request的结果作为第二个interceptor.request的参数, 以此类推
  requestPromise1.then(function(config1) {
    var requestPromise2 = interceptor2.request(config1);
    requestPromise2.then(function(config2) {
      // 最后一个interceptor.request的结果将被实际发送出去
      sendAjaxRequest(config2, function successHandler(response) {
        // response阶段, 后注册的先执行
        var responsePromise2 = interceptor2.response(response);
        responsePromise2.then(function(response2) {
          var responsePromise1 = interceptor1.response(response2);
          responsePromise1.then(function(response1) {
            // 此处触发$http(config).then的第一个回调函数, 并且把response1传过去
          });
        });
      }, function errorHandler(rejection) {
        // responseError阶段, 后注册的先执行
        var responseErrorPromise2 = interceptor2.responseError(rejection);
        responseErrorPromise2.then(function(rejection2) {
          var responseErrorPromise1 = interceptor1.responseError(rejection2);
          responseErrorPromise1.then(function(rejection1) {
            // 此处触发$http(config).then的第二个回调函数, 并且把rejection1传过去
          });
        });
      });
    });
  });
};

```

相比 Angular 中的代码, 这段代码臃肿不堪, 而且只能支持两个拦截器, 这是因为没有充分发挥 Promise 的威力。读者可以自己尝试用 Promise 来实现支持无限个拦截器, 然后在 Angular 源码中搜索 `var chain = [serverRequest, undefined]`; 并与其对照, 来学习 Promise 的高级用法。

我们要实现登录对话框, 那么该怎么写拦截器呢?

请看代码:

```

angular.module('com.ngnice.app').factory('AuthHandler', function AuthHandlerFactory($q)
{
  return {
    responseError: function(rejection) {
      // 如果服务器返回了401 unauthorized, 那么就表示需要登录
      if (rejection.status === 401) {
        // “未登录”是一个可补救的错误, 但我们先不补救, 直接弹出一个对话框
      }
    }
  };
});

```

```

        alert('需要登录');
        return $q.reject(rejection);
    } else {
        // 其他错误不用管，留给其他interceptor去处理
        return $q.reject(rejection);
    }
}
});
});

```

这是一个最简单的登录处理器，它不会弹出登录对话框，让用户登录，来试图“补救”这个错误，而是直接弹出一个对话框，然后继续原有逻辑。

接下来，就是重头戏了，我们要“补救”它！弹出一个输入框，提示用户输入密码，然后把固定的用户名（xuelang）和用户输入的密码发给服务器。

```

angular.module('com.ngnice.app').factory('AuthHandler', function AuthHandlerFactory
($q, $injector) {
    return {
        responseError: function (rejection) {
            // 如果服务器返回了401 unauthorized, 那么就表示需要登录
            if (rejection.status === 401) {
                var password = prompt('请输入密码: ');
                if (password) {
                    var Login = $injector.get('Login');
                    var $http = $injector.get('$http');
                    // 尝试登录
                    return Login.save({
                        username: 'xuelang',
                        password: password
                    }).$promise.then(function () {
                        // 登录成功了，就补救：重新发送刚才的请求，返回一个新的Promise，当
                        // 这个Promise完成时，原有的then回调函数会被执行
                        return $http(rejection.config);
                    });
                } else {
                    return $q.reject(rejection);
                }
            } else {
                // 其他的错误，留给其他的拦截器来处理
                return $q.reject(rejection);
            }
        }
    };
});

```

当实现了这段代码之后，整个执行过程是这样的：

- ❑ 前端调用需要登录的 api: `$http.get('/api/readers').then(ajaxCallback);`。
- ❑ 服务器返回 401（未认证）。

- ❑ 触发 AuthHandler 的 responseError 函数。
- ❑ 弹出 prompt 对话框，要求输入密码。
- ❑ 发送登录请求，传入用户名和密码。
- ❑ 登录完成后，重新发送刚才的 \$http.get('/api/readers') 请求，包含所有参数。
- ❑ 把这个请求作为新的 Promise，当它完成时，会触发刚才注册的 ajaxCallback 函数。

对于 /api/readers 的调用者来说，登录过程是完全“透明”的，它不用对登录过程做任何特殊处理——不用管 401，不用管重发。它的控制逻辑和不需要登录的 API 是完全一样的。

这种编程风格称为 AOP——面向切面编程（Aspect-Oriented Programming）。它的优点是“透明性”，专业的说法是“无侵入性”，也就是实现具体业务的代码不需要做任何改写。非常适合实现登录、错误处理、日志等与具体业务无关的通用类功能。

这样一来，开发具体业务逻辑的程序员就不需要关注这些了，而是由一两个程序员来专注处理这些通用功能。分工更加专业，沟通需求和耦合程度更低，开发组织的结构更加优化。

1.6.2 实现对话框

我们刚才通过系统的 Prompt API 实现了登录功能，但显然，它还不能用于产品级别：不能填写用户名、密码是明文、UI 无法定制。

我们需要一个可以完全由自己控制的登录对话框。不过，我们不用自己实现它——第三方组件库 angular-bootstrap 中就有一个。我们直接借用它就行了。

实现代码如下。

（1）JavaScript

```
// 使用独立controller，以便编写单元测试和复用
angular.module('com.ngnice.app').controller('UiPromptCtrl', function
  UiPromptCtrl($scope) {
    var vm = this;
    vm.submit = function () {
      // 有输入时才关闭
      if (vm.result) {
        $scope.$close(vm.result);
      }
    };
  });
angular.module('com.ngnice.app').service('ui', function Ui($modal, $rootScope) {
  this.prompt = function (message, defaultValue, title, secret) {
    // 想把参数传到界面中，就要通过一个scope传进去，我们这里通过给$new()传一个true参
```

```

    数, 来创建一个独立作用域
    var scope = $rootScope.$new(true);
    scope.title = title;
    scope.message = message;
    scope.secret = secret;
    // 打开对话框
    var modal = $modal.open({
        templateUrl: 'services/ui/prompt.html',
        controller: 'UiPromptCtrl as vm',
        // 指定对话框大小
        size: 'sm',
        scope: scope
    });
    // 返回一个监听对话框是否被关闭或取消的Promise
    return modal.result;
};
this.promptPassword = function (message, defaultValue, title) {
    return this.prompt(message, defaultValue, title, true);
};
});

```

(2) 模板

```

<div class="modal-header">
    <button type="button" class="close" ng-click="$dismiss()"><span aria-
        hidden="true">&times;</span><span
        class="sr-only">关闭</span></button>
    <h2 class="modal-title">{{title}}</h2>
</div>
<div class="modal-body">
    <form ng-submit="vm.submit()">
        <label>
            {{message}}
        </label>
        <input type="text" ng-if="!secret" class="form-control" ng-model="vm.result"/>
        <input type="password" ng-if="secret" class="form-control" ng-model="vm.result"/>
    </form>
</div>
<div class="modal-footer" autofocus>
    <button type="button" class="btn btn-primary" ng-click="vm.submit()">确定</button>
    <button type="button" class="btn btn-default" ng-click="$dismiss()">取消</button>
</div>

```

使用代码如下:

```

ui.promptPassword('这是一个测试' /*提示信息*/, 'abc' /*默认值*/, '提问' /*标题*/).
    then(function(value) {
        console.log(value);
    });

```


angular-bootstrap 的 `$modal` 函数已经封装了大部分逻辑，我们只是在它的基础上来定制一些特定用途的对话框，提高表意性，并去除冗余代码。

`$modal` 的接口文档可以参见其官方文档，这虽然是英文的，不过都很浅显，而且还有范例。

这里面需要注意的有几点。

控制器不要内联在 `$modal` 调用中，如 `$modal.open({...controller: function() {}})` 的形式是不提倡的，这将带来两大缺点：

- ❑ 这种控制器无法在外部引用到，因此难以被单元测试或被复用。
- ❑ 部分 minify 工具无法识别它，导致控制器中注入的变量被重命名而注入失败。

`modal.open` 的结果是一个对象，包括四个成员：

- ❑ `opened`：一个 Promise，当对话框成功打开时触发 `then` 函数。
- ❑ `result`：一个 Promise，当对话框打开后再被关闭时触发 `then` 函数，这通常表示用户点击了“确定”等按钮。
- ❑ `close`：一个函数，用于通过程序关闭对话框，触发 `then` 函数。
- ❑ `dismiss`：一个函数，用于通过程序关闭对话框，触发 `catch` 函数或 `then` 的第二个回调函数。

`vm` 的使用，这里是为了避免原型链继承问题，这个问题比较复杂，请参见第 4 章“最佳实践”中的 4.9 节“使用 `controller as vm` 方式”。

像 `prompt` 这类函数，通常只要关心最终结果就行了，所以没必要响应 `opened` 事件。所以这里做了一个简单的封装，使其直接返回 `result` 这个 Promise。

1.6.3 实现错误处理功能

用户的任何操作都可能产生错误，如果把错误处理代码分散的到处都是，代码就会显得很“脏”。怎么解决这个问题呢？本节中我们就来分析并给出解决方案。

错误分成两大类：一种是用户操作违反了业务规则导致的错误，我们称之为用户错误，例如我们要求密码至少有六位长，而用户输入了 3 位就提交了；另一种是我们自己程序本身的缺陷导致的错误，我们称之为系统错误，比如数据库连接中断。用户错误是意料之中的，我们知道用户可能出错，而系统错误是意料之外的，我们可能在设计之初就没有想到它会中断，或者认为它是小概率事件而暂时未加防范。

对用户错误的处理，我们已经在 1.4.5 节“添加验证器”等节中示范过，不再讲解，本节我们主要处理系统错误。

处理系统错误的原理和实现登录功能一样，都是基于 `Interceptor` 机制，区别在于登录功能处理的是 401 错误，而本节处理的是所有其他错误（4xx、5xx 等）。代码如下：

```
angular.module('com.ngnice.app').factory('myHttpInterceptor', function($q,
    dependency1, dependency2) {
    return {
        // 收到服务器给出的失败回应时的拦截函数。可选。
        'responseError': function(rejection) {
            // 0表示用户取消了Ajax请求，这通常出现在用户刷新当前页面的时候，不用管它
            if (rejection.status === 0) {
                return $q.reject(rejection);
            }
            // 401表示用户需要登录，我们在另一个interceptor中处理了它，这里忽略
            if (rejection.status === 401) {
                return $q.reject(rejection);
            }
            // 显示信息
            alert(rejection.data);
            return $q.reject(rejection);
        }
    };
});
```

这里不管三七二十一，把服务端返回的错误信息直接弹对话框显示出来。为了对用户更加友好，我们需要对 `status` 进行区分，翻译成更加友好的提示信息。另外，需要进一步改造对话框的外观，这里不再展开，读者可以参照前面的 `prompt` 函数来自己实现。

1.7 实战小结

通过本章的学习，你应该对敏捷方法有了一个直观的概念，如果想要深入学习，建议阅读《敏捷软件开发》等书籍。

“工欲善其事，必先利其器”，你还应该掌握了基本的前端工具链：Yeoman 或 FrontJet，它是进行前端开发的基础。

接下来，我们实现了第一个表单：新用户注册页面。在这里，你应该掌握了做表单、使用校验规则以及显示错误信息的技术。

后面的小节中，我们体会了内置指令的威力，你应该对这些内置指令有了一定的认识，但是要把它们用到极致，还需要很多的磨练，遇到需求时，请先想一想如何用内置指令实现。我们还写了一个初级的自定义指令，学会了它，就能对页面结构进行模块化分割，并且在必要时添加一些依靠内置指令无法实现的功能。

接下来，是一个高级功能，展示了一个足以让你进阶为 Angular 高手的特性：递归指令。充分理解它，将对你的学习很有帮助；但是如果还不怎么理解，也不用担心，只要学会用我写好的这个指令就可以了，不用理解其原理。

最后，我们讲解了 Angular 的一个重要特性：interceptor（拦截器），它可以在与服务器进行交互时，处理很多公共逻辑，如登录、错误显示等。这样可以防止公共逻辑分散到控制器等各种业务代码中。

同时，别忘了，我们还用单元测试贯穿了大部分小节，这些没有单独列为一节不是因为不重要，而是因为太重要。在后面的章节中，我们还将对测试进行深入讲解。



概念介绍

子曰：必也正名乎。在实际开发中，让每个人对概念都有一致的理解是很重要的，写书也是如此。

在上一章，我们使用了很多概念，并且假设你已经知道了这些概念的名字和大致用法。对于入门来说，这样已经足够了，但本书可不是一本入门书籍。

本章中，我们将深入讲解这些概念，有些提法甚至可能颠覆你以前的认知。不过没关系，这是成长中的重要一步。

有了稳固的基础，才能理解后面更深的专家级章节。

2.1 什么是 UI

提起 UI，你一定知道它是指用户界面（User Interface），但是如果细细剖析，你会发现它没那么简单。

对于一个用户界面，它实际上包括三个主要部分：

- ❑ **内容**：你想展现哪些信息？包括动态信息和静态信息。注意，这里的内容不包括它的格式，比如生日，跟它显示为红色还是绿色无关，跟它显示为年月日还是显示为生辰八字也无关。
- ❑ **外观**：这些信息要展示为什么样子？这包括格式和样式。样式还包括静态样式和动画效果等。

□ **交互**：用户点击了加入购物车按钮时会发生什么？还要更新哪些显示？

在前端技术栈中，这三个部分分别由三项技术来负责：HTML 负责描述内容，CSS 负责描述外观，JavaScript 负责实现交互。当然，这三者之间没有明确的界限，比如有些格式化需要 JavaScript 来实现，而 HTML 也往往会影响到一些样式。

如果进一步抽象，它们分别对应 MVC 的三个主要部分：内容—Model，外观—View，交互—Controller。

对应到 Angular 中的概念，“静态内容”对应模板，“动态内容”对应 Scope，交互对应 Controller，外观部分略微复杂点：CSS 决定样式，过滤器（filter）则决定格式。

有了这些概念为基础，我们再来深入讲解下 Angular 中的概念。

2.2 模块

与其他现代语言不同，当前版本的 JavaScript (ECMAScript 5) 并没有内置模块化语法。但是，随着程序规模越来越大，模块化的需求越来越重要，于是出现了 require.js 等第三方库，试图用库来弥补语言的不足。Angular 并不依赖 require.js 等第三方库，而是自己实现了模块化系统，这个系统的核心就是模块（module）。

我们先来回顾一下“模块”的概念，然后自然就明白 Angular 中的 module 是怎么回事了。

所谓模块是指把相关的一组编程元素（如类、函数、变量等）组织到同一个发布包中。这些编程元素之间紧密协作，隐藏实现细节，只通过公开的接口与其他模块合作。

模块是一个粒度适中的复用单位，也是最常见的复用形式。比如我们常用的第三方库往往对外公开好几个类，使用者只要关注其公开接口就可以了，不用了解其实现细节，这种第三方库就是一个“模块”。

Angular 的 module 也是如此。Angular 中的编程元素包括 Service、Directive、Filter、Controller 等，它们只有通过模块进行“导出”才能供别人使用。如：`angular.module('com.ngnice.app').service('ui', function() {...});` 语句的作用是：先取出一个名为 `com.ngnice.app` 的模块，然后把 `function() {...}` 作为一个回调函数以 `ui` 作为名字注册进去。这样，别人就可以随时通过 `ui` 这个名字把它从 `com.ngnice.app` 模块中取出来。

所以，我们可以简单地把模块看做一个注册表（registry），它保存着名字和编程元素的对照表，既可以存入，也可以取出。

一个程序往往不会只含有一个模块，这些模块需要互相协作，这就导致了模块之间

具有依赖关系，比如有一个可复用模块，名叫 `common`，而我们的应用想要使用其中名叫 `authHandler` 的 `service`。那么我们就要先声明这种依赖关系：`angular.module('com.ngnice.app', ['common'])`，这样，Angular 就知道该去 `common` 模块中查找这个名叫 `authHandler` 的 `Service`。如果没有声明这种依赖关系，那么就算引入了它所在的 JavaScript 文件，也照样是无法找到的，这是新手很容易踩坑的地方，请特别注意。同时，Angular 还可以自动检测出循环依赖，以免出现无限递归。

注意，我们刚才调用了两次 `module` 函数：`angular.module('com.ngnice.app')` 和 `angular.module('com.ngnice.app', ['common'])`，前者可不是后者的简写形式，而是具有完全不同的含义：前者的作用是引用一个模块，也就是说查找一个名叫 `app` 的模块，并返回其引用，如果模块不存在，则会触发一个异常 `[$injector:nomod] Module 'com.ngnice.app' is not available...`；而后者的作用是创建一个模块，并且声明这个模块需要依赖一个名为 `common` 的模块，第二个参数是个数组，所以还可以声明依赖更多个模块。

模块依赖关系是一棵树，这就意味着：凡是依赖了 `app` 模块的更高级模块，也会自动依赖 `app` 所依赖的 `common` 等模块。

2.3 作用域

如果你曾经用 jQuery 写过传统的非 MVC 前端程序，那么在第 1 章“从实战开始”中看到的这些代码可能会让你感到惊讶——不再有连篇累牍的 DOM 操作，不再有混合了业务逻辑和视图逻辑的臃肿代码，不再一听到写测试就闪过“不可能”三个字……

“光荣属于 MVC！”我们说 Angular 是个 MVW（Model-View-Whatever）风格的框架，而在 Angular 中扮演 Model 角色的概念，就是作用域（scope）。

在 Angular 中，scope 通过原型继承的方式被组织成了一棵树，它的根节点就是 `$rootScope`，这是 Angular 在启动时自动创建的，它通常对应于 `ng-app` 指令，并且关联到 `ng-app` 所在的节点。

接下来，Angular 会解析 `ng-app` 包含的 HTML，其中的一些指令，如 `ng-view`、`ng-if`、`ng-repeat` 等也会创建自己的 scope，这些 scope 都是从 `$rootScope` 及其子 scope 创建出来的，它们的嵌套关系也和这些指令的嵌套关系一致。

由于它们使用了原型继承的方式，所以，凡是上级 scope 拥有的属性，都可以从下级 scope 中读取，但是当需要对这些继承下来的属性进行写入的时候，问题就来了：写入会导致在下级 scope 上创建一个同名属性，而不会修改上级 scope 上的属性。这不是 Bug，而是

“原型继承”的固有语义，这是用惯了“类继承”的后端程序员需要特别注意的。更详细的分析和解决方式请参见 4.9 节“使用 controller as vm 方式”。

这种 scope 树很好的体现了 Model 之间的嵌套关系，对业务数据的结构是一个很恰当的抽象。

scope 之所以如此重要，是因为它事实上是 Angular 解耦业务逻辑层和视图层的关键：Controller 操作 scope，View 则展现 scope 的内容，传统前端程序中大量复杂的 DOM 操纵逻辑都被转变成对 scope 的操作。

这种树形结构不但体现在数据的继承关系上，而且体现在消息的冒泡机制上。有 DOM 基础的同学可以拿它和 DOM 的消息冒泡机制做类比，在后面的 2.11 节“消息”中我也会专门对此进行讲解。

2.4 控制器

和传统的 MVC 程序中一样，Angular 中的控制器（controller）用来对模块（scope）进行操作，包括初始化数据和定义事件响应函数等。

我们常见的定义控制器的方法是：`angular.module('com.ngnice.app').controller('UserListCtrl', function() {...})`；其中：`angular.module('com.ngnice.app')` 语句在前面已经讲过，是返回一个现有的 module 实例，而 Controller 就是这个 module 实例上的一个方法，它的作用是把后面的函数以 `UserListCtrl` 为名字，注册到模块中去，以便需要时可以根据名字找到它。

使用控制器的场景有几种。最常见的是用在路由中。

比如在 `angular-ui-router` 中，我们可以通过下面的语句使用它：

```
$stateProvider.state('user.list', {  
  url: '/list',  
  templateUrl: 'views/user/list.html',  
  controller: 'UserListCtrl'  
});
```

这样，当用户访问 `/user/list` 这个 URL 的时候，`angular-ui-router` 插件就会实例化一个名为 `UserListCtrl` 的控制器，同时，创建一个 scope 对象并传给它。这个控制器实例会在这个 scope 对象上创建属性、方法等，这个过程称为“初始化 scope 对象”。

初始化完之后，加载模板，并且把 scope 对象传入，模板中会通过 Angular 指令绑定这些属性、方法。Angular 会通过一个称为摘要循环（digest cycle）的过程，自动维护 scope 变量和视图中 DOM 节点的一致性，这时候的 DOM 我们称之为“活 DOM（Live DOM）”。

更具体的工作原理我们会在第3章“背后的原理”的3.2节“Angular启动过程”中详细讲解。

另一个常用的场景是在单元测试中。

通常，在单元测试阶段我们不必测试视图，而是将其留待端到端测试阶段。在单元测试阶段，我们要关注的是控制器的工作逻辑。前面我们提过，控制器的作用是在 scope 对象上创建属性、方法，所以我们的测试逻辑就是看它是否创建了正确的属性，以及由它创建的方法是否能正常工作。

于是我们得出了下列测试逻辑：

```
var scope = $rootScope.$new();
var ctrl = $controller('UserListCtrl');
ctrl(scope);
// TODO: 检测scope中是否如同预期的产生了初始数据
// TODO: 调用scope中的方法，然后检测scope变量是否发生了预期的变化
```

其中的 \$controller 是 Angular 提供的一个系统服务，用来查找以前通过 module.controller('UserListCtrl', function() {...}); 注册的控制器函数。

还有一个不常用但值得提倡的场景是在指令中，特别是用来封装一个界面片段的“组件型指令”，我们在2.6节“指令”中会进一步展开讲解。之所以在这种类型的指令中提倡使用控制器，主要是为了方便进行单元测试，而不用引入对视图层的测试。

有一些第三方服务或指令也会使用控制器，它们所做的工作实际上和 angular-ui-router 一样的：创建一个 scope，找到一个控制器，然后用控制器对 scope 进行初始化，最后把 scope 绑定到视图，把生成的 Live DOM 渲染出来。

掌握了控制器的工作原理，在自己的代码中也可以灵活运用，凡是需要 Live DOM 的地方都可以通过各种形式使用。

生成 Live DOM 时涉及另一个重要的知识点 \$compile，完整的实现方式我会在后面2.6节“指令”中讲解。

2.5 视图

我们有了模块和控制器之后，内容和交互逻辑就已经基本确定了，接下来我们就得把它们展示给用户了，这时就用到“视图”(view)。

CSS 并不在 Angular 的范围内，在实践中，常常结合一套成熟的 CSS 架构来做，比如 Bootstrap 就可以和 Angular 结合得非常好。

Angular 中实现视图的主体是模板。最常见的模板形式当然是 HTML，也有通过 Jade 等中间语言编译为 HTML 的。模板中包括静态信息和动态信息，静态信息是指直接写死 (hard code) 在模板中的，而动态信息则是对 scope 中内容的展示。

展示动态信息的方式有两种：

- ❑ **绑定表达式**：形式如 `{{username}}`，绑定表达式可以出现在 HTML 中的文本部分或节点的属性部分。
- ❑ **指令**：形式如 ``，事实上任何指令都可以用来展示动态信息，展示的方式取决于指令的内在实现逻辑。

视图中的绑定表达式或指令中用到的变量或函数都是 `$scope` 中的一个属性或方法，上面两个表达式绑定的都是 `$scope.username`，但是也可以绑定到方法，如 `$scope.getFirstName()`。绑定到方法是一种很常用的方式，但是如果使用不当，也可能导致一些很难追查的 Bug。特别要注意不要在方法中返回一个新对象或新数组，否则会出现 “10 \$digest iterations reached.” 错误。

另一个要点是，在表达式中无法直接使用 window 对象下的全局属性或方法。Angular 这样的设计可以确保视图的局部性，以免受到意料之外的干扰而出现 Bug。如果确实需要调用，请在 Controller 中简单包装一层。

除了展示信息之外，常常还需要对信息进行格式化，比如把一个 Date 对象格式化为 “年 - 月 - 日” 格式或 “年 - 月 - 日 时 : 分 : 秒” 格式。一个分层清晰的系统，在 scope 中通常是没有格式的概念的，想要显示成什么格式是视图层的事情。这样的设计可以提高 Model 层的内聚性（只做一件事），把与此相关的需求变更在视图层处理，可以提高 Model 层的稳定性。

Angular 中对信息进行格式化的机制是过滤器 (Filter)，如：`{{birthday|date}}`。对过滤器的更深入讲解，请参见稍后的 2.7 节 “过滤器”。

2.6 指令

指令 (directive) 是 Angular 中一个很重要的概念，相当于一个自定义的 HTML 元素，在 Angular 官方文档中称它为 HTML 语言的 DSL (特定领域语言) 扩展。

按照指令的使用场景和作用可以分为两种类型的指令：它们分别为组件型指令 (Component) 和装饰型器指令 (Decorator)，它们的分类命名，并不是笔者独创的新法，它是在 Angular 2.x 中提出的概念，笔者认为它们也同样可以使用于 Angular 1.x。

组件型指令主要是为了将复杂而庞大的 View 分离，使得页面的 View 具有更强的可读性和维护性，实现“高内聚低耦合”和“分离关注点”的有效手段；而装饰器型指令则是为 DOM 添加行为，使其具有某种能力，如自动聚焦（autoFocus）、双向绑定、可点击（ngClick）、条件显示 / 隐藏（ngShow/ngHide）等能力，同时它也是链接 Model 和 View 之间的桥梁，保持 View 和 Model 的同步。在 Angular 中内置的大多数指令，是属于装饰器型指令，它们负责收集和创建 \$watch，然后利用 Angular 的“脏检查机制”保持 View 的同步。

对于组件型指令和装饰器型指令的这两种区分是非常重要的，它们在写法、业务含义、适用范围等方面都有非常明显的区别，理解了它们，对于我们日常的指令开发也具有很好的指导作用。

2.6.1 组件型指令

组件型指令是一个小型的、自封装和内聚的一个整体，它包含业务所需要显示的视图以及交互逻辑，比如：我们需要在首页放置一个登录框和一个 FAQ 列表，如果我们把它们都直接写在首页的视图和控制器中，那么首页的视图和控制器将会变得非常庞大，这样不利于我们的分工协作和页面的长期维护。这时候更好的方案应该是，把它们拆分成两个独立的内聚的指令 login-panel 和 faq-list，然后分别将 <login-panel></login-panel> 和 <faq-list></faq-list> 两个指令嵌入到首页。

注意，我们在这里拆出这两个指令的直接目的不是为了复用，更重要的目的应该是分离 View，促进代码结构的优化，达到更好的语义化和组件化，当然对于这样独立内聚的指令，有时我们还能意外地获得更好的复用性。

组件型指令应该是满足封装的自治性、内聚性的，它不应该直接引用当前页面的 DOM 结构、数据等。如果存在需要的信息，则可以通过指令的属性传递或者利用后端服务接口来自我满足。如 login-panel 应该在其内部访问登录接口来实现自我的功能封装。它的 Scope 应该是独立的（isolated），不需要对父作用域的结构有任何依赖，否则一旦父作用域的结构发生改变，可能它也需要相应地变更，这种封装是很脆弱的。更好的封装应该是“高内聚低耦合”的，内聚是描述组件内部实现了它所应该包含的逻辑功能，耦合则描述它和外部组件之间应该是尽量少的相互依赖。

组件型指令的写法通常是这样的：

```
// 声明一个指令
angular.module('com.ngnice.app').directive('jobCategory', function () {
    return {
        // 可以用作HTML元素，也可以用作HTML属性
```

```

    restrict: 'EA',
    // 使用独立作用域
    scope: {
        configure: '='
    },
    // 指定模板
    templateUrl: 'components/configure/tree.html',
    // 声明指令的控制器
    controller: function JobCategoryCtrl($scope) {
        ...
    }
  };
});

```

指令中 `return` 的这个结果，我们称之为“指令定义对象”。

`restrict` 属性用来表示这个指令的应用方式，它的取值可以是 E（元素）、A（属性）、C（类名）、M（注释）这几个字母的任意组合，工程实践中常用的是 E、A、EA 这三个，对于 C、M 笔者并不建议使用它们。对于组件型指令来说，标准的用法是 E，但是为了兼容 IE8，通常也支持一个 A，因为 IE8 的自定义元素需要先用 `document.createElement` 注册，用 A 可以省去注册的麻烦。

`scope` 有三种取值：不指定（`undefined`）/ `false`、`true` 或一个哈希对象。

不指定或为 `false` 时，表示这个指令不需要新作用域。它直接访问现有作用域上的属性或方法，也可以不访问作用域。如果同一节点上有新作用域或独立作用域指令，则直接使用它，否则直接使用父级作用域。

为 `true` 时，表示它需要一个新作用域，可以跟本节点上的其他新作用域指令共享作用域，如果任何指令都没有新作用域，它就会创建一个。

为哈希对象时，表示它需要一个独立的（`isolated`）作用域。所谓独立作用域，是指独立于父作用域，它不会从父节点自动继承任何属性，这样的话，就不会无意间引用到父节点上的属性，导致意料之外的耦合。

要注意，一个节点上如果已经出现了一个独立作用域指令，那么就不能再出现另一个独立作用域指令或者新作用域指令，否则使用 `scope` 的代码将无法区分两者，如果自动将两个作用域合并，又会失去“独立性”。总之，记住一句话：独立作用域指令是“排它”的。

那么哈希对象的内容呢？它表示的是属性绑定规则，如：

```

{
  // 绑定字面量
  name: '@',
  // 绑定变量
  details: '=',

```

```

// 绑定事件
onUpdate: '&'
}

```

这里我们绑定了三个属性，以 `<user-details name='test' details='details' on-update='updateIt(times)'></user-details>` 为例，`name` 的值将被绑定为字符串 'test'，而 `details` 的值不是 'details'，而是绑定到父页面 `scope` 上一个名为 `details` 的变量，当父页面 `scope` 的 `details` 变量变化时，指令中的值也会随之变化——即使绑定到 `number` 等原生类型也一样。而 `onUpdate` 绑定的则是一个回调函数，它是父页面 `scope` 上一个名为 `updateIt` 的函数。当指令代码中调用 `scope.onUpdate()` 的时候，父页面 `scope` 的 `updateIt` 就会被调用。当然，`name` 也同样可以绑定到变量，但是要通过绑定表达式的方式，比如 `<user-details name="{{name}}"></user-details>` 中，`name` 将会绑定到父页面 `scope` 中的 `name` 变量，并且也会同步更新。

记住，对于组件型指令，更重要的是内容信息的展示，所以我们一般不涉及指令的 `link` 函数，而应该尽量地将业务逻辑放置在 `Controller` 中。

组件化的开发方式以及组件化的复用，是我们在前端开发中一直追求的一个理想目标。从最初的 `iframe`、`jQuery UI`、`Extjs`、`jQuery easyui`，我们一直在不懈地朝着组件化的方向前进。`Angular` 首次在其框架中提出指令这种以 `HTML DSL` 方式进行语义化、组件化扩展的方式，就我们在这里描述的组件型指令。笔者也更愿意将它称为“`Directive as component`”（指令即组件）。只要告诉大家下面的实例代码是一个在线支付页面，相信大家很快就能从页面中读懂它业务逻辑了：

```

<form novalidate name="orderForm" ng-submit="processPage();">
  <error-panel errors="order.errors"></error-panel>
  <fieldset class="field-group">
    <post-address class="post-address" view-model="order.poastAddress" post-address-
      change="order.postAddressChange();"></post-address>
  </fieldset>
  <fieldset class="field-group">
    <payment-way class="payment-way" viewmodel="order.paymentWay"></
      payment-way>
  </fieldset>
  <fieldset class="field-group">
    <item-list class="item-list" viewmodel="order.items"></item-list>
  </fieldset>
  .....
  <div class="submit">
    <button class="btn primary-btn">提交订单</button>
  </div>
</form>

```

从上面的代码中，我们能很快地识别出此页面包含：全局错误显示、邮寄地址、在线支付方式、购买商品信息这几个领域概念，然而对于它们的修改和维护也很容易，组件更加的内聚，并且遵守单一职责原则（SRP）。这就是“Directive as component”和组件型指令的迷人之处。

继 Angular 的指令之后，React 也推出了以 JSX 模板为核心的类 HTML 语法扩展，以此来实现组件化的开发，而且它也是 React 中的最重要的核心概念。Google 和 Mozilla 也在推进 Web Component 技术，它主要以 Custom Elements、HTML Templates、Shadow DOM、HTML Imports 四大技术为核心，让我们能够像浏览器开发者一样使用 HTML、CSS、JavaScript 来构建更酷、更炫、独立的 HTML 节点，使得我们能够快速的应对越来越复杂、多样化的用户体验要求，而不是继续等待浏览器厂商来实现它们。

有兴趣的读者，可以自行阅读更多关于 Web Component 的资料。可以参考：<http://webcomponents.org/>、Google 的 polymer 框架：<http://www.polymer-project.org/> 以及 Mozilla 的 X-Tags 框架：<http://x-tags.org/>。

2.6.2 装饰器型指令

对于装饰器型指令，其定义方式则如下：

```
angular.module('com.ngnice.app').directive('twTitle', function () {
  return {
    // 用作属性
    restrict: 'A',
    link: function (scope, element, attrs) {
      ...
    }
  };
});
```

装饰器型指令主要用于添加行为和保持 View 和 Model 的同步，所以它不同于组件型指令，我们经常需要进行 DOM 操作。其 restrict 属性通常为 A，也就是属性声明方式，这种方式更符合装饰器的语义：它并不是一个内容的主体，而是附加行为能力的连接器。

同时，由于多个装饰器很可能被用于同一个指令，包括独立作用域指令，所以装饰器型指令通常不使用新作用域或独立作用域。如果要访问绑定属性，该怎么做呢？仍然看前面的例子 `<user-details name="test" details="details" on-update="updateIt(times)"></user-details>`，假如不使用独立作用域，我们该如何获取这些属性的值呢？

- ❑ 对于 @ 型的绑定，我们可以直接通过 attrs 取到它：attrs.name 等价于 name: '@'。
- ❑ 对于 = 型的绑定，我们可以通过 scope.\$eval 取到它：scope.\$eval(attrs.details) 等价

于 details: '='。

& 型的绑定理解起来会稍有困难, 先看代码: `scope.$eval(attrs.onUpdate, {times: 3});`。

和 = 型绑定一样, `onUpdate` 属性在本质上是当前 `scope` 上的一个表达式。特殊的地方在于, 这个表达式是一个函数, `$eval` 发现它是函数时, 就可以传一个参数表 (在 Angular 中称之为 `locals`) 给它。`onUpdate` 表达式中可以使用的参数名和它的参数值, 都来自这个参数表。

使用的时候, 我们可以在视图中引用这个哈希对象的某个属性作为参数, 比如对于刚才的定义, 视图中的 `on-update="updateIt(times)"` 所引用的 `times` 变量就来自我们刚才在 `callback` 中传入的 `times` 属性, 而 `updateIt` 函数被调用时将会接收到它, 参数值是 3。

```
$scope.updateIt = function(times) {
    // 这里times的值应该是3, 但是这个times不需要跟视图和指令中的名称一致, 它叫什么都可以。但
    // 视图和指令中的名称必须一致
};
```

在装饰器指令中, 其实还有一种细分的分支, 它完全不操纵 DOM, 只是对当前 `scope` 进行处理, 如添加成员变量、函数等。代码如下:

```
angular.module('com.ngnice.app').directive('twToggle', function () {
    return {
        restrict: 'A',
        scope: true,
        link: function(scope) {
            scope.$node = {
                folded: false,
                toggle: function() {
                    this.folded = !this.folded;
                }
            };
        }
    };
});
```

使用的时候:

```
<ul>
  <li ng-repeat="item in items" tw-toggle="">
    <span ng-click="$node.toggle()">切换</span>
    <ul ng-if="$node.folded">
      ...
    </ul>
  </li>
</ul>
```

它的作用是在当前元素的作用域上创建一个名为 `$node` 的哈希对象, 这个哈希对象具

有一组自定义的属性和方法，可用来封装交互逻辑。

也许你已经想到了，这种类型的指令还可以进一步改进。如何改进呢？

```
angular.module('com.ngnice.app').directive('twToggle', function () {
  return {
    restrict: 'A',
    scope: true,
    controller: function($scope) {
      $scope.folded = false;
      $scope.toggle = function() {
        $scope.folded = !$scope.folded;
      };
    }
  };
});
```

它好在哪里？笔者不直接给出答案，请读者自行分析，并尝试理解它，这对于指令的认识是很重要的。

2.7 过滤器

我们在第1章中使用了多个系统内置的过滤器（filter），还写了一个自定义过滤器，这里我们再系统化的从理论层面讲一下。

过滤器标准的定义方式是：

```
angular.module('com.ngnice.app').filter('myFilter', function(/* 这里可以用参数进行依赖注入 */) {
  return function(input) {
    var result;
    // TODO: 把input变换成result
    return result;
  };
});
```

可以看出，过滤器是一个特殊的函数，它返回一个函数，这个函数接收的第一个参数就是被过滤的变量，如使用 `{{1|myFilter}}` 时，这个 `input` 参数的值就是 1，当这个值是个变量时，它的变化会导致 `myFilter` 再次被执行。

过滤器还可以接收第二个参数，乃至第 N 个参数，如：

```
return function(input, arg1, arg2, arg3) {
  ...
};
```

而使用者则通过 `{{1|myFilter:2:3:4}}` 的形式调用它。这种情况下，`arg1` 的值为 2，`arg2`

的值为 3，arg3 的值为 4。

从使用者的角度，我们可以把 filter 看做一个函数，它负责接收输入，然后转换成输出。每当输入参数发生变化时，它就被执行，其输出会被视图使用。

filter 除了可以用在绑定表达式之外，还可以用在指令中通过值绑定的属性，如 `<li ng-repeat="item in items | filter:'a'">...`。

由于其简单灵巧，filter 非常适合复用。官方提供的几个 filter 就有很多种用法，读者可以参考官方的 API 文档和实战篇的案例，自行尝试用 ng-repeat, filter, orderBy 的组合来实现具有前端过滤功能的表格，这有助于对过滤器的深入理解。

2.8 路由

前端“路由”（router）的概念和后端的路由是一样的，也就是根据 URL 找到 view-controller 组合的机制。最开始的时候，Angular 的路由库合并的核心库中，现在，路由库从 Angular 核心库中剥离出来。官方的路由库称为 ngRoute，由于其过于简陋，我在工程实践中比较常用的是一个第三方路由库：angular-ui-router。

ngRoute 的写法是：

```
$routeProvider.when('/url', {
  templateUrl: 'path/to/template.html',
  controller: 'SomeCtrl'
});
```

angular-ui-router 的写法是：

```
$stateProvider.state('name', {
  url: '/url',
  templateUrl: 'path/to/template.html',
  controller: 'SomeCtrl'
});
```

虽然写法不同，但是其工作原理都是类似的：

监听 \$locationChangeSuccess 事件，它将在每次 URL（包括 # 后面的 hash 部分）发生变化时触发。

在这个事件中，将根据 \$routeProvider/\$stateProvider 中注册的路由表中的 URL 部分查阅其路由对象，如：

```
{
  url: '/url',
  templateUrl: 'path/to/template.html',
```

```

    controller: 'SomeCtrl'
  }

```

从这个路由对象中，可以取到两个关键参数：`templateUrl/controller`。

- ❑ 创建一个 `scope` 对象。
- ❑ 加载模板，借助浏览器的能力把它解析为静态的 DOM。
- ❑ 使用 `Controller` 对 `scope` 进行初始化，添加属性和方法。
- ❑ 使用 `$compile` 服务把刚才生成的 DOM 和 `scope` 关联起来，变成一个 Live DOM。
- ❑ 用这个 Live DOM 替换 `ng-view/ui-view` 中的所有内容。

你可能还有印象，这个过程我们在前面的第 1 章也用过。

这个过程非常简单。作为练习，你可以使用类似的原理来实现一个简单的路由功能，这个过程中会接触到很多 Angular 核心服务，对理解 Angular 的核心工作原理非常有用。

2.9 服务

如果你是一个后端程序员，那么对服务（Service）的概念一定不会陌生。在 Angular 中，服务的概念是一样的，差别只在于技术细节。

服务是对公共代码的抽象，比如，如果在多个控制器中都出现了相似的代码，那么把它们提取出来，封装成一个服务，你将更加遵循 DRY 原则（即：不要重复你自己），在可维护性等方面获得提升。

如同我们在第 1 章的 `tree` 服务中所看到的，由于服务剥离了和具体表现相关的部分，而聚焦于业务逻辑或交互逻辑，它更加容易被测试和复用。

但是，在工程实践中，我们引入服务的首要目的是为了优化代码结构，而不是复用。复用只是一项结果，不是目标。所以，当你发现你的代码中混杂了表现层逻辑和业务层逻辑的时候，你就要认真考虑抽取服务了——哪怕它还看不到复用价值。

如果你遵循着测试驱动开发的方式，那么当你觉得测试很难写的时候，回头审视下，看是否这里可以抽取出一个服务，转而对服务进行测试。

服务的概念通常是和依赖注入紧密相关的，Angular 中也一样。如果你困惑于在 JavaScript 中是如何实现依赖注入的，请参见第 3 章“背后的原理”中的 3.3 节“依赖注入”。

由于依赖注入的要求，服务都是单例的，这样我们才能把它们到处注入，而不用手动管理它们的生命周期，并容许 Angular 实现“延迟初始化”等优化措施。

在 Angular 中，服务分成很多种类型：

- ❑ 常量（Constant）：用于声明不会被修改的值。

- ❑ **变量 (Value)**: 用于声明会被修改的值。
- ❑ **服务 (Service)**: 没错, 它跟服务这个大概念同名, 原作者在“开发者指南”中把这种行为比喻为“把自己的孩子取名叫‘孩子’——一个会气疯老师的名字”。事实上, 同名的原因是——它跟后端领域的“服务”实现方式最为相似: 声明一个类, 等待 Angular 把它 new 出来, 然后保存这个实例, 供它到处注入。
- ❑ **工厂 (Factory)**: 它跟上面这个“服务”不同, 它不会被 new 出来, Angular 会调用这个函数, 获得返回值, 然后保存这个返回值, 供它到处注入。它被取名为“工厂”是因为: 它本身不会被用于注入, 我们使用的是它的产品。但是与现实中的工厂不同, 它只产出一份产品, 我们只是到处使用这个产品而已。
- ❑ **供应商 (Provider)**: “工厂”只负责生产产品, 它的规格是不受我们控制的, 而“供应商”更加灵活, 我们可以对规格进行配置, 以便获得定制化的产品。

事实上, 除了 Constant 外, 所有这些类型的服务, 背后都是通过 Provider 实现的, 我们可以把它们看做让 Provider 更容易写的语法糖。一个明显的佐证是: 当你使用一个未定义的服务时, Angular 给你的错误提示是它对应的 Provider 未找到, 比如我们使用一个未定义的服务: test, 那么 Angular 给出的提示是: Unknown provider: testProvider <- test。

Constant 比较特殊, 我们稍后讲解, 我们先来看其他几个。

Provider 的声明方式如下:

```
angular.module('com.ngnice.app').provider('greeting', function() {
  var _name = 'world';
  this.setName = function(name) {
    _name = name;
  };
  this.$get = function(/*这里可以放依赖注入变量*/) {
    return 'Hello, ' + _name;
  };
});
```

使用时:

```
angular.module('com.ngnice.app').controller('SomeCtrl', function($scope, greeting) {
  // 这里greeting应该等于'Hello, world', 怎么样, 你猜对了吗?
  $scope.message = greeting;
});
```

对 Provider 进行配置时:

```
angular.module('com.ngnice.app').config(function(greetingProvider) {
  greetingProvider.setName('wolf');
});
```

容器的伪代码如下：

```
var instance = diContainer['greeting'];           // 先找是否已经有了一个实例
if (!angular.isUndefined(instance)) {
    return instance;                             // 如果已经有了一个实例，直接返回
}

var ProviderClass = angular.module('com.ngnice.app').lookup('greetingProvider');
// 在服务名后面自动加上Provider后缀是Angular遵循的一项约定
var provider = new ProviderClass(); // 把Provider实例化
provider.setName('wolf');
instance = provider.$get();          // 调用$get，并传入依赖注入参数
diContainer['greeting'] = instance;  // 把调用结果存下来
return instance;
```

事实上，如果不需要对 **name** 参数进行配置，声明代码可以简化为：

```
angular.module('com.ngnice.app').value('greeting', 'Hello, world');
```

这也就是需要这么多语法糖的原因。

下面给出其他语法糖的等价形式：

2.9.1 服务

```
angular.module('com.ngnice.app').service('greeting', function() {
    this.sayHello = function(name) {
        return 'Hello, ' + name;
    };
});
```

等价于：

```
angular.module('com.ngnice.app').provider('greeting', function() {
    this.$get = function() {
        var Greeting = function() {
            this.sayHello = function(name) {
                return 'Hello, ' + name;
            };
        };
        return new Greeting();
    };
});
```

使用时：

```
angular.module('com.ngnice.app').controller('SomeCtrl', function($scope, greeting) {
    $scope.message = greeting.sayHello('world');
});
```

2.9.2 工厂

```
angular.module('com.ngnice.app').factory('greeting', function() {
    return 'Hello, world';
});
```

等价于：

```
angular.module('com.ngnice.app').provider('greeting', function() {
    this.$get = function() {
        var greeting = function() {
            return 'Hello, world';
        };
        return greeting();
    }
});
```

使用时：

```
angular.module('com.ngnice.app').controller('SomeCtrl', function($scope, greeting) {
    $scope.message = greeting;
});
```

在 Angular 源码中，它们的实现是这样的：

```
function factory(name, factoryFn) { return provider(name, { $get: factoryFn }); }

function service(name, constructor) {
    return factory(name, ['$injector', function($injector) {
        return $injector.instantiate(constructor);
    }]);
}

function value(name, val) { return factory(name, valueFn(val)); }
```

Angular 提供了这么多种形式的服务，那么我们在工程实践中该如何选择？我们可以遵循下列决策流程：

- ☐ 需要全局的可配置参数？用 Provider。
- ☐ 是纯数据，没有行为？用 Value。
- ☐ 只 new 一次，不用参数？用 Service。
- ☐ 拿到类，我自己 new 出实例？用 Factory。
- ☐ 拿到函数，我自己调用？用 Factory。

但是，还有另一种更加敏捷的方式：

- ☐ 是纯数据时，先用 Value；当发现需要添加行为时，改写为 Service；或当发现需要通过计算给出结果时，改写为 Factory；当发现需要进行全局配置时，改写为

Provider。

- ❑ 最酷的是，这个过程对于使用者是透明的——它不需要因为实现代码的改动而更改原有代码。如上面 Value 和 Factory 的使用代码，仅仅从使用代码中我们区分不出它是 Value 还是 Factory。

接下来，我们来看 Constant。与其他 Service 不同，Constant 不是 Provider 函数的语法糖。更重要的差别是，它的初始化时机非常早，可以在 `angular.module('com.ngnice.app').config` 函数中使用，而其他的服务是不能被注入到 `config` 函数中的。这也意味着，如果你需要在 `config` 中使用一个全局配置项，那么它就只能声明为常量，而不能声明为变量。

在官方的开发指南中，给出了一个完整的对比表，见表 2-1。

表 2-1 服务指令对比表

类 型	Factory	Service	Value	Constant	Provider
可以依赖其他服务	是	是	否	否	是
使用类型友好的注入	否	是	是	是	否
在 config 阶段可用	否	否	否	是	是
可用于创建函数 / 原生对象	是	否	是	是	是

下面给出解释：

- ❑ 可以依赖其他服务：由于 Value 和 Constant 的特殊声明形式，显然没有进行依赖注入的时机。
- ❑ 使用类型友好的注入：这条没有官方的解释，我的理解是——由于 Factory 可以根据程序逻辑返回不同的数据类型，所以我们无法推断其结果是什么类型，也就是对类型不够友好。Provider 由于其灵活性比 Factory 更高，因此在类型友好性上和 Factory 是一样的。
- ❑ 在 config 阶段可用：只有 Constant 和 Provider 类型在 config 阶段可用，其他都是 Provider 实例化之后的结果，所以只有 config 阶段完成后才可用。

可用于创建函数 / 原生对象：由于 Service 是 new 出来的，所以其结果必然是类实例，也就无法直接返回一个可供调用的函数或数字等原生对象。

如果你确实需要对一个没有提供 Provider 的第三方服务进行配置，该怎么办呢？Angular 提供了另一种机制：decorator。这个 decorator 和前面提到过的装饰器型指令没有关系，它是用来改变服务的行为的。

比如我们有一个第三方服务，名叫 ui，它有一个 `prompt` 函数，我们不能改它源码，但需要让它每次弹出提问框时都在控制台输出一条记录，那么我们可以这样写：


```
angular.module('com.ngnice.app').config(function($provide) {
  // $delegate是ui的原始服务
  $provide.decorator('ui', function($delegate) {
    // 保存原始的prompt函数
    var originalPrompt = $delegate.prompt;
    // 用自己的prompt替换
    $delegate.prompt = function() {
      // 先执行原始的prompt函数
      originalPrompt.apply($delegate, arguments);
      // 再写一条控制台日志
      console.log('prompt');
    };
    // 返回原始服务的实例，但也可以返回一个全新的实例
    return $delegate;
  })
});
```

这种方式给你了超级灵活性，你可以改写包括 Angular 系统服务在内的任何服务——事实上，angular-mocks 模块就是使用 decorator 来 MOCK \$httpBackend、\$timeout 等服务的。

不过，如果你大幅修改了原始服务的逻辑，那么，这可能会给自己和维护者挖坑。俗话说，“不作死就不会死”。如果让我来总结 decorator 的使用原则，那就是——慎用、慎用、慎用，如果确实想用，请务必遵循“Liskov 代换”原则，并写好单元测试。特别是，如果你想修改系统服务的工作逻辑，建议先多看几遍文档，确保你正确理解了它的每一个细节！

2.10 承诺

承诺（Promise）不是 Angular 首创的。作为一种编程模式，它出现在……1976 年，比 JavaScript 还要古老得多。Promise 全称是 Futures and promises（未来与承诺）。要想深入了解，可以参见 http://en.wikipedia.org/wiki/Futures_and_promises。

而在 JavaScript 世界中，一个广泛流行的库叫作 Q (<https://github.com/kriskowal/q>)。而 Angular 中的 \$q 就是从它引入的。

1. 生活中的一个例子

Promise 解决的是异步编程的问题，对于生活在同步编程世界中的程序员来说，它可能比较难于理解，这也构成了 Angular 入门门槛之一，本节将用生活中的一个例子对此做一个形象的讲解。

假设有一个家具厂，而它有一个 VIP 客户张先生。

有一天张先生需要一个豪华衣柜，于是，他打电话给家具厂说：“我需要一个衣柜，回

头做好了给我送来”，这个操作就叫 `$q.defer()`，也就是延期。因为这个衣柜不是现在要的，所以张先生这是在发起一个可延期的请求。

家具厂接下了这个订单，给他留下了一个回执号，并对他说：“我们做好了会给您送过去，放心吧”。这叫作 `Promise`，也就是给了张先生一个“承诺”。

这样，这个 `defer` 算是正式创建了，于是他把这件事记录在自己的日记上，并且同时记录了回执号，这个变量叫作 `deferred`，也就是已延期事件。

现在，张先生就不用再去想着这件事了，该做什么做什么，这就是“异步”请求的含义。

假设家具厂在一周后做完了这个衣柜，并如约送到了张先生家（包邮哦，亲），这就叫作 `deferred.resolve`（衣柜），也就是“问题已解决，这是您的衣柜”。而这时候张先生只要取出一下这个“衣柜”参数就行了。而且，这个“邮包”中也不一定只有衣柜，还可以包含别的东西，比如厂家宣传资料、产品名录等。整个过程中轻松愉快，谁也没等谁，没有浪费任何时间。

假设家具厂在评估后发现这个规格的衣柜我们做不了，那么它就需要 `deferred.reject`（理由），也就是“我们不得不拒绝您的请求，因为……”。拒绝没有时间限制，可以发生在给出承诺之后的任何时候，甚至可能发生在快做完的时候。而且拒绝时候的参数也不仅仅限于理由，还可以包含一个道歉信，违约金之类的。总之，你想给他什么就给他什么，如果你觉得不会惹恼客户，那么不给也没关系。

假设家具厂发现，自己正好有一个符合张先生要求的存货，它就可以用 `$q.when`（现有衣柜）来兑现给张先生的承诺。于是，这件事立刻解决了，皆大欢喜。张先生可不在乎你是从头做的还是现有的成品，只要达到自己的品质要求就满意了。

假设这个家具厂对客户格外的细心，它还可以通过 `deferred.notify`（进展情况）给张先生发送进展情况的“通知”。

这样，整个异步流程圆满完成！无论成功还是失败，张先生都没有往里面投入任何额外的时间成本。

好，我们再扩展一下这个故事：

张先生又来订货了，这次他分多次订了一张桌子，三把椅子，一张席梦思。但他不希望今天收到个桌子，明天收到个椅子，后天又得签收一次席梦思，而是希望家具厂做好了之后一次性送过来，但是他当初又是分别下单的，那么他就可以重新跟家具厂要一个包含上述三个承诺的新承诺，这就是 `$q.all`（桌子承诺，椅子承诺，席梦思承诺），这样，他就不用再关注以前的三个承诺了，直接等待这个新的承诺完成，到时候只要一次性签收了前

面的这些承诺就行了。

2. 回调地狱和 Promise

通过上面这个生活中例子，相信作为读者的你已经了解到了异步和 Promise 的方式。为什么我们需要 Promise 呢？

JavaScript 是一门很灵活的语言，由于它寄宿在浏览器中以事件机制为核心，所以在我们的 JavaScript 编码中存在很多的回调函数。这是一个高性能的编程模式，所以它衍生出了基于异步 I/O 的高性能 Nodejs 平台。但是如果不注意我们的编码方法，那么我们会陷入“回调地狱”，也有人称为“回调金字塔”。嵌套式的回调地狱，代码将会变得像意大利面条一样。如下边的嵌套回调函数一样：

```
async1(function() {  
    async2(function() {  
        async3(function() {  
            async4(function() {  
                ....  
            });  
        });  
    });  
});
```

这样嵌套的回调函数，让我们的代码的可读性变得很差，而且很难于调试和维护。所以为了降低异步编程的复杂性，开发人员一直寻找简便的方法来处理异步操作。其中一种处理模式称为 Promise，它代表了一种可能会长时间运行而且不一定必须完成的操作的结果。这种模式不会阻塞和等待长时间的操作完成，而是返回一个代表了承诺的 (Promised) 结果的对象。它通常会实现一种名叫 then 的方法，用来注册状态变化时对应的回调函数。

Promise 在任何时刻都处于以下三种状态之一：未完成 (pending)、已完成 (resolved) 和拒绝 (rejected) 三个状态。以 CommonJS Promise/A 标准为例，Promise 对象上的 then 方法负责添加针对已完成和拒绝状态下的处理函数。then 方法会返回另一个 Promise 对象，以便于形成 Promise 管道，这种返回 Promise 对象的方式能够让开发人员把异步操作串联起来，如 then(resolvedHandler, rejectedHandler)。resolvedHandler 回调函数在 Promise 对象进入完成状态时会触发，并传递结果；rejectedHandler 函数会在拒绝状态下调用。

所以我们上边的嵌套回调函数可以修改为：

```
async1().then(async2).then(async3).catch(showError);
```

这下代码看着清爽多了，我们不再需要忍受嵌套的无底深渊。

在 ES6 的标准版中已经包含了 Promise 的标准，很快它就将会从浏览器本身得到更好的支持。与此同时在 ES6 的标准版中，还引入了 Python 这类语言中的 generator（迭代器的生成器）概念，它本意并不是为异步而生的，但是它拥有天然的 yield 暂停函数执行的能力，并保存上下文，再次调用时恢复当时的状态，所以它也被很好地运用于 JavaScript 的异步编程模型中，其中最出名的案例当属 Node Express 的下一代框架 KOA 了。

最后还有个好消息，在 ES7 的标准中将有可能引入 async 和 await 这两个关键词，来更大的简化我们的 JavaScript 异步编程模型。我们就可以如下的方式以同步的方式编写我们的异步代码：

```
async function sleep(timeout) {
  return new Promise((resolve, reject) => {
    setTimeout(function() {
      resolve();
    }, timeout);
  });
}

(async function() {
  console.log('做一些事情, ' + new Date());
  await sleep(3000);
  console.log('做另一些事情, ' + new Date());
})();
```

3. Angular 中的 Promise

在 Angular 中大量使用着 Promise，最简单的是 \$timeout 的实现，我拷贝过来并加上了注释：

```
function timeout(fn, delay, invokeApply) {
  // 创建一个延期请求
  var deferred = $q.defer(),
      promise = deferred.promise,
      skipApply = (isDefined(invokeApply) && !invokeApply),
      timeoutId;

  timeoutId = $browser.defer(function() {
    try {
      // 成功，将触发then的第一个回调函数
      deferred.resolve(fn());
    } catch(e) {
      // 失败，将触发then的第二个回调函数或catch的回调函数
      deferred.reject(e);
      $exceptionHandler(e);
    } finally {

```

```

        delete deferreds[promise.$$timeoutId];
    }

    if (!skipApply) $rootScope.$apply();
}, delay);

promise.$$timeoutId = timeoutId;
deferreds[timeoutId] = deferred;
// 返回承诺
return promise;
}

timeout.cancel = function(promise) {
    if (promise && promise.$$timeoutId in deferreds) {
        deferreds[promise.$$timeoutId].reject('canceled');
        delete deferreds[promise.$$timeoutId];
        return $browser.defer.cancel(promise.$$timeoutId);
    }
    return false;
};
};

```

2.11 消息

在传统的 DOM 编程中，消息（message）机制非常有用，特别是消息冒泡机制，让我们不用额外的代码就可以实现“职责链”模式。但是我们要尽量摆脱 DOM 操作，难道这是必须使用 DOM 操作的场景吗？不是的，Angular 中也有一种不依赖 DOM 的消息机制，本节中我们就对它进行详细讲解。

我们知道，Scope 也被组织成了一棵树，跟 DOM 树具有相似的结构。Angular 的消息机制就是通过 scope 上的几个函数实现的：

- ❑ `$broadcast(name, args)`：向当前 scope 及其所有下级 scope 递归广播名为 name 的消息，并带上 args 参数。
- ❑ `$emit(name, args)`：向当前 scope 及其所有直线上级 scope 发送名为 name 的消息，并带上 args 参数。
- ❑ `$on(name, listener)`：监听本 scope 收到的消息，listener 的形式为：`function(event, args) {}`，event 参数的结构和 DOM 中的 event 类似。

以图 2-1 所示的结构 scope 为例：

当我们在 rootScope 上调用 `$broadcast` 广播一个消息时，任何一个 scope（包括 rootScope）上通过 `$on` 注册的 listener 都将收到这个消息。当我们在 scope1 上调用 `$broadcast` 广播一个消息时，scope1/scope1.1/scope1.2 将依次收到这个消息。当我们在

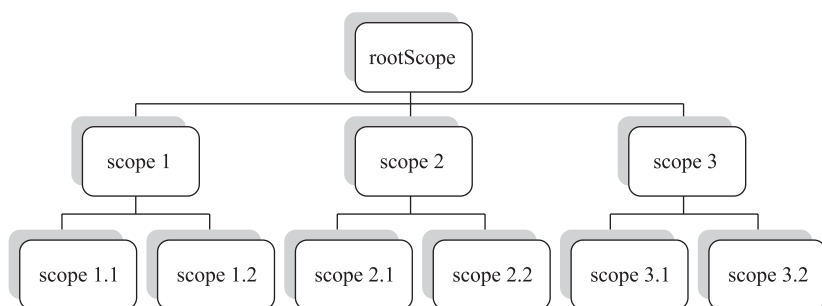


图 2-1 scope 树

rootScope 上调用 \$emit 上传一个消息时，rootScope 将收到这个消息。当我们在 scope1.1 上调用 \$emit 上传一个消息时，scope1.1/scope1/rootScope 将依次收到这个消息。

当通过 \$emit 上传一个消息时，将使用冒泡机制，比如，假设我们在 scope1.1 上调用 \$emit，我们在 scope1 上注册一个 listener：

```
scope1.$on('name', function(event) {
    event.stopPropagation();
});
```

这个 stopPropagation 函数将阻止冒泡，也就是说 scope1.1 和 scope1 都将正常接收到这个消息，但 rootScope 就接收不到这个消息了。

有时候，用消息机制和普通回调函数都能达到类似的效果，如何选择呢？

当一个嵌套结构具有树形的业务含义时，我们就优先使用消息机制来通讯。或者从另一个角度看，符合“职责链”模式的适用场景时，消息机制比较合适。反之，应该使用普通的回调函数。

如果难以决定使用消息还是回调函数，那么就优先使用回调函数（主要是 Angular 事件），因为这种情况下执行路径比较明确，容易跟踪。或者在对此有深入理解前，先使用表面的判断方式：一个事件是否需要被很多地方处理？调用 stopPropagation 是否有意义？如果是，那么用消息，否则用回调。

2.12 单元测试

我们在第 1 章中已经写过两个单元测试（unit test）了，这里我们简单讲一下理论知识。

在 Angular 中，单元测试的概念和传统的后端编程是一样的。也就是对某些小型功能块儿进行测试，保障其工作逻辑正常。单元测试要尽可能局部化，不要牵扯进很多个模块，必要时可进行 mock（模拟）。

2.12.1 MOCK 的使用方式

由于 JavaScript 语言的动态特性，Mock 一个普通对象不需要进行特别处理。比如，如果一个测试函数需要访问 `scope` 中的一个变量：`name`，但不用访问 `$watch` 等 `scope` 的特有函数，那么传入一个普通的哈希对象 `{name: 'someName'}` 即可，并不需要 `new` 出一个 `scope` 来。

除了局部化以外，对单元测试来说，一大挑战是网络操作，如果使用真实的网络操作，那么将带来几个问题：

- ❑ 网络的不稳定性，导致单元测试的不稳定性。想象一个有时成功，有时失败的单元测试，会让程序员多么头疼吧！
- ❑ 网络响应的速度会拖慢整体速度。单元测试执行得必须尽可能快速，如果被迫由于网络操作而变慢，那么一旦多了就会变得很慢，也就会有很多时间浪费在这里。
- ❑ 网络的异步性。虽然异步调用对于单元测试来说并不是不可接受的，但是由于其返回时机不受控制，所以写起来还是比同步调用复杂一些。

另一大挑战是与时间有关的测试。比如一段代码中设置了一小时后触发的定时器，难道我们的单元测试就要一个小时后才能完成？这显然是不合理的。

好在，Angular 对网络和定时器等进行了封装，变成了 `$http`、`$timeout`、`$interval` 等服务。这就意味着，我们只要使用这些内置服务而不是 `setTimeout` 等原生函数，那么我们就可以对它们进行 Mock，克服上述问题。

对于这些内置服务，Angular 提供了一个独立的测试库：`angular-mocks.js`。

它对 Angular 的一些内置服务进行了 Mock，比如 `$httpBackend`、`$timeout`、`$interval`、`$ExceptionHandler`、`$log` 等服务。还提供了一些工具函数，如用于加载模块的 `module` 函数、用于依赖注入的 `inject` 函数、用于调试的 `dump` 函数等，这些函数都是顶层函数，不需要加前缀就可以调用。

但 Angular 实际上没有 Mock `$http` 服务，而是 Mock 了 `XHR` (`XMLHttpRequest`) 对象，它把原来发送到服务端的 Ajax 调用，转变成本地调用。这个通过本地调用来模拟服务器的对象则是 `$httpBackend`（模拟 http 后端，也就是服务器）。

```
$httpBackend.whenGET('/someUrl').respond({name: 'wolf'}, {'X-Record-Count': 100});
```

上述语句声明了一个模拟服务端，当被测试代码请求 `GET /someUrl` 这个地址时，将被 `$httpBackend` 拦截，并返回一个 JSON 对象 `{"name": "wolf"}`，同时，返回一个额外的 `response header`：`X-Record-Count`，其值为 `"100"`。

注意，我们这里其实只是定义了返回规则，并没有规定啥时候返回这些数据，也就是说，虽然被测代码中的 `$http` 函数已经能正确返回我们期望的数据，但目前还不会被触发——直到我们调用了 `$httpBackend.flush` 函数。这样，我们就把测试中的异步调用变成了同步调用。

`respond` 中的参数不但可以是一个或两个哈希对象，还可以在前面增加一个返回码，如 `respond(401, {message: 'Unauthorized'}, {'X-Sign-It': '1887a6b'})` 等，Angular 会自动判断它的数据类型，来决定使用哪种重载形式。如果你需要更多的控制力，还可以转而传入一个函数，其原型是：`function(method, url, data, headers) {}`，这个函数中的四个参数都是由 `$http` 请求发来的数据，这个函数的返回值是一个数组，包含状态码、数据等信息，完整的范例如：

```
$httpBackend.whenPOST('/someUrl').respond(function(method, url, data, headers) {
  var result = 'Hello, ' + data.name;
  return [201, result, {'X-Greeting': 'Say Hello'}, 'OK'];
});
```

这样，当被测代码请求调用 `$http.post('/someUrl', {name: 'wolf'})`，然后调用 `$httpBackend.flush()` 时，获得的回应为：状态码 201，回应体：Hello, wolf，回应头：X-Greeting: 'Say Hello'，同时它的 status text 为 OK。

不过，虽然这种形式很灵活，但对于单元测试来说，还是不应该把 mock 逻辑写得过于复杂，否则，如果测试代码本身都可能出错，会让你的测试变得非常痛苦。写 mock 时，推荐的最佳实践是“给出固定数据，返回固定数据”。

如果把上述代码改写为：`$httpBackend.whenPOST('/someUrl', {name: 'wolf'}).respond('Hello, wolf', {'X-Greeting': 'Say Hello'}, 'OK');`，不但代码量少了很多，而且更加简洁明确，更能发挥“测试”作为“规约”的作用。

`$timeout` 和 `$interval` 的 Mock 就比较简单了，只是增加了一个 `flush` 函数，它的作用和 `$httpBackend.flush` 一样，也是立即触发这个异步操作。

2.12.2 测试工具与断言库

我们写好了测试，还要把它跑起来，用来跑测试的工具称为 Test Runner，Angular 的范例工程中集成的测试工具是 Karma，它的用法对写测试来说几乎可以不用管。

而代码中用来写断言的库称为断言库，在范例工程中集成的是 jasmine。我们测试代码中的 `expect` 和 `toBe` 等函数都是来自它的。具体的使用方式可以参见它们的官方文档，此处就不展开讲解了。

2.13 端到端测试

端到端测试 (e2e test)，也称为场景测试，它模拟的是用户真实的操作场景：

- ❑ 用户打开 `http://xxx` 地址。
- ❑ 在搜索框中输入了 `abc`。
- ❑ 然后点击其后的搜索按钮。

这时候，他期望看到一个列表，列出所有在标题的任意位置包含了字符串 `abc` 的条目，并且每条结果中的 `abc` 这三个字母被高亮。

所谓端到端，也就是一端是浏览器，另一端是服务器，这个测试贯通了前后端，具有近似于验收测试的价值。

端到端测试不是什么新技术，它在前后端分离架构盛行之前就已经被广泛采用了，比如 Selenium，而且 Selenium 也同样可应用于 Angular 中。

Angular 的端到端测试工具称为 Protractor，事实上，它就是基于 Selenium 的，在 Selenium 的基础上，它增加了一些 Angular 特有的元素选取方式，如根据 `ng-model` 选取元素等。

我的建议是，除非你所在的开发组织已经把 Angular 作为唯一的前端选项，否则请使用 Selenium 中传统的函数，而不要使用 Protractor 特有的根据 `ng-model` 选取元素等函数，这将让你们的测试独立于前端技术栈而被复用。

在我的工程实践中，只会使用 `id`、`class` 等少数查阅方式，而不会根据 `ng-model` 等进行查阅。并且，由于 Angular 的特点，被测试程序中可以不用任何 `id`，所以，我们可以完全把 `id` 留给测试人员使用。如果写测试的人员有权修改源码，那么他/她可以自由的添加、删除 `id`，而不用担心破坏了程序的逻辑和样式。遵循这个约定，可以让开发与测试的协同更加有效。

这里不展开讲解，只把我在种子工程中写的一些代码加上注释供大家自行研究：

1) pages/HomePage.js

```
// 这是一个页面对象，用来封装页面中的元素和操作，以简化规约代码，并提供一定的变更隔离
module.exports = function() {
  this.title = function() {
    // browser对象封装一组用来访问浏览器属性的函数
    return browser.getTitle();
  };
  // 根据id查找元素
  this.name = element(by.id('name'));
  this.nameEcho = element(by.id('name-echo'));
  this.get = function() {
    // 控制浏览器访问特定地址
```

```

        browser.get('http://localhost:5000/#/');
    };
};

```

2) demoSpec.js

```

// 取得页面对象
var HomePage = require('./pages/HomePage');

describe('e2e范例, 如果修改了首页, 请修改本测试 >', function () {
    var homePage;
    // 所有测试语句执行之前, 先在浏览器中打开它
    beforeEach(function () {
        homePage = new HomePage();
        homePage.get();
    });

    it('默认的标题是Showcase', function() {
        expect(homePage.title()).toBe('Showcase');
    });
    it('输入用户名后应该回显', function() {
        // 检查初始状态是否符合预期
        expect(homePage.nameEcho.getText()).toBe('Hello, ');
        // 模拟用户输入
        homePage.name.sendKeys('test');
        // 检查操作后状态是否符合预期
        expect(homePage.nameEcho.getText()).toBe('Hello, test');
    });
});

```