目录

[目录 1](#_Toc427655640)

[环境安装 2](#_Toc427655641)

[Maven 2](#_Toc427655642)

[1特点 3](#_Toc427655643)

[2常用命令 3](#_Toc427655644)

[3推荐书籍 4](#_Toc427655645)

[4配置 4](#_Toc427655646)

[5 生命周期 5](#_Toc427655647)

[6 jetty 插件使用 6](#_Toc427655648)

[bower 11](#_Toc427655649)

[Windows环境下的NodeJS+NPM+Bower安装配置 11](#_Toc427655650)

[用到的技术 13](#_Toc427655651)

[SpringMVC 13](#_Toc427655652)

[1 Spring Web MVC是什么 13](#_Toc427655653)

[2 Spring Web MVC能帮我们做什么 14](#_Toc427655654)

[3 Spring Web MVC架构 14](#_Toc427655655)

[4 Spring Web MVC优势 19](#_Toc427655656)

[Hibernate 20](#_Toc427655657)

[1核心接口和类 20](#_Toc427655658)

[2主键介绍 21](#_Toc427655659)

[3包的作用 23](#_Toc427655660)

[4缓存管理 24](#_Toc427655661)

[5延迟加载 27](#_Toc427655662)

[6性能优化 28](#_Toc427655663)

[7 Hibernate和Mybatis的区别 31](#_Toc427655664)

[Shiro 32](#_Toc427655665)

[Shiro架构介绍 32](#_Toc427655666)

[Shiro 认证 34](#_Toc427655667)

[Shiro 授权 38](#_Toc427655668)

[Realm 实现 45](#_Toc427655669)

[Shiro 配置说明 47](#_Toc427655670)

[AngularJS 49](#_Toc427655671)

[1. 关于AngularJS 49](#_Toc427655672)

[2. 关于本文档 49](#_Toc427655673)

[3. 开始的例子 49](#_Toc427655674)

[4. 依赖注入 52](#_Toc427655675)

[5. 作用域 53](#_Toc427655676)

[6. 数据绑定与模板 53](#_Toc427655677)

[7. 模板 56](#_Toc427655678)

[8. 模板中的过滤器 64](#_Toc427655679)

[9. 锚点路由 67](#_Toc427655680)

[10. 定义模板变量标识标签 71](#_Toc427655681)

[11. AJAX 71](#_Toc427655682)

[12. 工具函数 75](#_Toc427655683)

[13. 其它服务 77](#_Toc427655684)

[14. 自定义模块和服务 79](#_Toc427655685)

[15. 附加模块 ngResource 81](#_Toc427655686)

[16. AngularJS与其它框架的混用(jQuery, Dojo) 86](#_Toc427655687)

[17. 自定义过滤器 90](#_Toc427655688)

[18. 自定义指令directive 91](#_Toc427655689)

[Bootstrap 118](#_Toc427655690)

[什么是 Bootstrap？ 118](#_Toc427655691)

[历史 119](#_Toc427655692)

[为什么使用 Bootstrap？ 119](#_Toc427655693)

[Bootstrap 包的内容 119](#_Toc427655694)

[学习网站 120](#_Toc427655695)

[实例网站项目结构 120](#_Toc427655696)

[技术结构 120](#_Toc427655697)

[代码结构 121](#_Toc427655698)

# 环境安装

开发工具使用IntelliJ IDEA

开发网站经常需要使用其他人的JAR包和JS文件

有两个工具分别对这两类文件进行管理，即Maven(管理项目依赖的Jar文件)、bower(管理项目依赖的JS文件)

## Maven

Maven是基于项目对象模型(POM)，可以通过一小段描述信息来管理项目的构建，报告和文档的软件项目管理工具。

Maven 除了以程序构建能力为特色之外，还提供高级项目管理工具。由于 Maven 的缺省构建规则有较高的可重用性，所以常常用两三行 Maven 构建脚本就可以构建简单的项目。由于 Maven 的面向项目的方法，许多 Apache Jakarta 项目发文时使用 Maven，而且公司项目采用 Maven 的比例在持续增长。

Maven这个单词来自于意第绪语，意为知识的积累，最早在Jakata Turbine项目中它开始被用来试图简化构建过程。当时有很多项目，它们的Ant build文件仅有细微的差别，而JAR文件都由CVS来维护。于是Maven创始者开始了Maven这个项目，该项目的清晰定义包括，一种很方便的发布项目信息的方式，以及一种在多个项目中共享JAR的方式。

### 1特点

那么，Maven 和 Ant 有什么不同呢？在回答这个问题以前，首先要强调一点：Maven 和 Ant 针对构建问题的两个不同方面。Ant 为 Java 技术开发项目提供跨平台构建任务。Maven 本身描述项目的高级方面，它从 Ant 借用了绝大多数构建任务。因此，由于 Maven 和 [Ant](http://baike.baidu.com/subview/42824/15924875.htm)代表两个差异很大的工具，所以接下来只说明这两个工具的等同组件之间的区别，如表 1 所示。[1]

|  |  |  |
| --- | --- | --- |
|  | **Maven** | **Ant** |
| 标准构建文件 | project.xml 和 maven.xml | build.xml |
| 特性处理顺序 | 1. ${maven.home}/bin/driver.properties 2. ${project.home}/project.properties 3. ${project.home}/build.properties 4. ${user.home}/build.properties 5. 通过 -D 命令行选项定义的系统特性   **最后一个**定义起决定作用。 | 1. 通过 -D 命令行选项定义的系统特性 2. 由 任务装入的特性   **第一个**定义最先被处理。 |
| 构建规则 | 构建规则更为动态（类似于编程语言）；它们是基于 Jelly 的可执行 XML。 | 构建规则或多或少是静态的，除非使用<script>任务 |
| 扩展语言 | 插件是用 Jelly（XML）编写的。 | 插件是用 Java 语言编写的。 |
| 构建规则可扩展性 | 通过定义 <preGoal> 和 <postGoal> 使构建 goal 可扩展。 | 构建规则不易扩展；可通过使用 <script> 任务模拟 <preGoal> 和 <postGoal> 所起的作用。 |

Maven是一个项目管理工具，它包含了一个项目对象模型 (Project Object Model)，一组标准集合，一个项目生命周期(Project Lifecycle)，一个依赖管理系统(Dependency Management System)，和用来运行定义在生命周期阶段(phase)中插件(plugin)目标(goal)的逻辑。当你使用Maven的时候，你用一个明确定义的项目对象模型来描述你的项目，然后Maven可以应用横切的逻辑，这些逻辑来自一组共享的（或者自定义的）插件。

Maven 有一个生命周期，当你运行 mvn install 的时候被调用。这条命令告诉 Maven 执行一系列的有序的步骤，直到到达你指定的生命周期。遍历生命周期旅途中的一个影响就是，Maven 运行了许多默认的插件目标，这些目标完成了像编译和创建一个 JAR 文件这样的工作。

此外，Maven能够很方便的帮你管理项目报告，生成站点，管理JAR文件，等等。

### 2常用命令

mvn archetype：create 创建Maven项目

mvn compile 编译源代码

mvn deploy 发布项目

mvn test-compile 编译测试源代码

mvn test 运行应用程序中的单元测试

mvn site 生成项目相关信息的网站

mvn clean 清除项目目录中的生成结果

mvn package 根据项目生成的jar

mvn install 在本地Repository中安装jar

mvn eclipse:eclipse 生成eclipse项目文件

mvn jetty:run 启动jetty服务

mvn tomcat:run 启动tomcat服务

mvn clean package -Dmaven.test.skip=true:清除以前的包后重新打包，跳过测试类

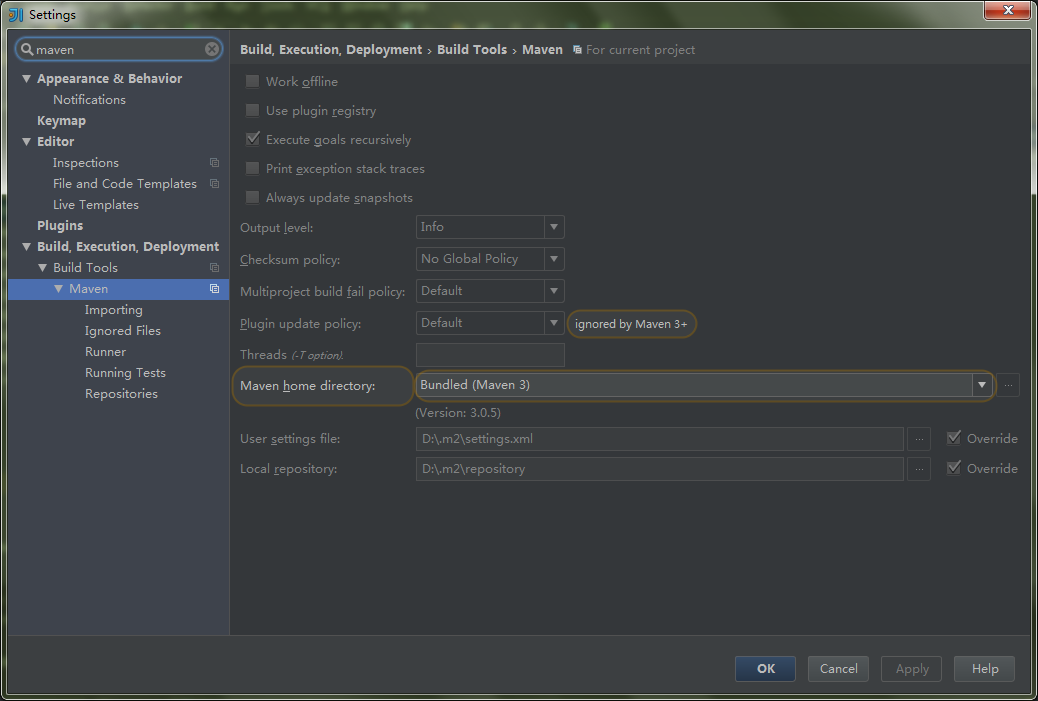
### 3推荐书籍

发文时中文版对maven介绍比较全的书籍是《Maven实战》

### 4配置

IntelliJ IDEA自带Maven,不需要单独下载

点击File→Settings,在搜索栏中搜索maven



在此处配置maven相关路径，默认maven的配置文件和jar仓库会放在C盘的用户文件夹中，建议修改到其他位置。

可以在配置文件中修改Jar包的下载地址，可建立公司内部Jar包服务器。

### 5 生命周期

Maven的生命周期是为了对所有的构建过程进行了抽象了，便于统一。

**clean**(清理)

此生命周期旨在给工程做清理工作，它主要包含以下阶段：

**pre-clean** - 执行项目清理前所需要的工作。

**clean** - 清理上一次build项目生成的文件。

**post-clean** - 执行完成项目清理所需的工作.

default(默认)

**validate** - 验证项目是否正确且所有必要的信息都可用。

**initialize** - 初始化构建工作，如：设置参数，创建目录等。

**generate-sources** - 为包含在编译范围内的代码生成源代码.

**process-sources** - 处理源代码, 如过滤值.

**generate-resources** -

**process-resources** - 复制并处理资源文件，至目标目录，准备打包。

**compile** - 编译项目中的源代码.

**process-classes** - 为编译生成的文件做后期工作, 例如做Java类的字节码增强.

**generate-test-sources** - 为编译内容生成测试源代码.

**process-test-sources** - 处理测试源代码。

**generate-test-resources** -

**process-test-resources** - 复制并处理资源文件，至目标测试目录。

**test-compile** - 将需测试源代码编译到路径。一般来说，是编译/src/test/java目录下的java文件至目标输出的测试classpath目录中。

**process-test-classes** -

**test** - 使用合适的单元测试框架运行测试。这些测试代码不会被打包或部署。

**prepare-package** -

**package** - 接受编译好的代码，打包成可发布的格式，如 JAR 。

**pre-integration-test** -

**integration-test** - 按需求将发布包部署到运行环境。

**post-integration-test** -

**verify** -

**install** -将包安装到本地仓库，给其他本地引用提供依赖。

**deploy** -完成集成和发布工作，将最终包复制到远程仓库以便分享给其他开发人员。

site(站点)

**pre-site** - 执行一些生成项目站点前的准备工作。

**site** - 生成项目站点的文档。

**post-site** - 执行需完成站点生成的工作，如站点部署的准备工作。

**site-deploy** - 向制定的web服务器部署站点生成文件。[2]

Maven生命周期参考：Apache Maven Project[2] ，书籍《Maven In Action》

### 6 jetty 插件使用

#### pom.xml 配置

在你的 pom.xml 文件中添加 jetty 插件的描述信息（[查看Jetty更多的版本信息](http://search.maven.org/#search%7Cgav%7C1%7Cg%3A%22org.eclipse.jetty%22%20AND%20a%3A%22jetty-maven-plugin%22)）：

[...]  
<build>  
  <plugins>  
    <plugin>  
      <groupId>org.eclipse.jetty</groupId>  
      <artifactId>jetty-maven-plugin</artifactId>  
      <version>9.2.8.v20150217</version>  
    </plugin>  
  </plugins>  
</build>  
[...]

#### 启动与停止

如果你希望通过命令 mvn jetty:stop 执行关闭 jetty 服务，你需要像下面一样在你的 pom.xml 配置文件中添加一个特殊的端口和控制键：

<configuration>  
  [...]  
  <stopKey>shutdown</stopKey>  
  <stopPort>9966</stopPort>  
  [...]  
</configuration>

你仍可以通过 mvn jetty:run 启动 jetty 服务，可以通过 mvn jetty:stop 来停止 jetty 服务。

#### 端口配置

jetty 默认使用的端口是 8080，命令行的方式修改端口的命令是：mvn -Djetty.port=8081 jetty:run 。pom.xml 配置方式如下：

<configuration>  
  [...]  
  <httpConnector>  
    <port>8081</port>  
  </httpConnector>  
  [...]  
</configuration>

#### 自动热部署

在你的 pom.xml 中添加如下配置：

<configuration>  
  [...]  
  <scanIntervalSeconds>2</scanIntervalSeconds>  
  [...]  
</configuration>

默认值是 0。大于 0 的数值表示开启，0 表示关闭，单位为秒。以配置数值为一个周期，自动的扫描文件检查其内容是否有变化，如果发现文件的  
内容被改变，则自动重新部署运用。命令行的方式：mvn -Djetty.scanIntervalSeconds=2 jetty:run 。

#### 手动重加载

在你的 pom.xml 文件中添加如下配置，reload 的可选值 ：[automatic|manual]

<configuration>  
  [...]  
  <reload>manual</reload>  
  [...]  
</configuration>

默认值为 automatic，它与大于 0 的 scanIntervalSeconds 节点一起作用，实现自动热部署的工作。设为 manual 的好处是，当你改变文件  
内容并保存时，不会马上触发自动扫描和重部署的动作，你还可以继续的修改，直至你在 Console 或命令行中敲回车键（Enter）的时候才触发重新加载的动作。这样可以更加的方便调试修改。命令行的方式是：mvn -Djetty.reload=manual jetty:run 。

#### 访问日志

在你的 pom.xml 文件添加如下配置：

<configuration>  
  [...]  
  <requestLog implementation="org.eclipse.jetty.server.NCSARequestLog">  
    <filename>target/access-yyyy\_mm\_dd.log</filename>  
    <filenameDateFormat>yyyy\_MM\_dd</filenameDateFormat>  
    <logDateFormat>yyyy-MM-dd HH:mm:ss</logDateFormat>  
    <logTimeZone>GMT+8:00</logTimeZone>  
    <append>true</append>  
    <logServer>true</logServer>  
    <retainDays>120</retainDays>  
    <logCookies>true</logCookies>  
  </requestLog>  
  [...]  
</configuration>

[org.eclipse.jetty.server.NCSARequestLog](http://download.eclipse.org/jetty/stable-9/apidocs/org/eclipse/jetty/server/NCSARequestLog.html) 是 [org.eclipse.jetty.server.RequestLog](http://download.eclipse.org/jetty/stable-9/apidocs/org/eclipse/jetty/server/RequestLog.html) 的一个实现类。  
org.eclipse.jetty.server.NCSARequestLog 是一种伪标准的 NCSA 日志格式。下面是一些节点参数的解释：  
filename：日志文件的名称  
filenameDateFormat：日志文件的名称的日期格式，它要求日志文件名必须含有 yyyy\_mm\_dd 串  
logDateFormat：日志内容的时间格式  
logTimeZone：时区  
append：追加到日志  
logServer：记录访问的主机名  
retainDays：日志文件保存的天数, 超过删除  
logCookies：记录 cookies  
启动 jetty 服务，在项目的 target 目录下会生成一个 access-2015\_06\_23.log 文件，该文件中的其中一条记录如下：

localhost 0:0:0:0:0:0:0:1 - - [2015-06-23 01:17:05] "GET /css/main.css HTTP/1.1" 304 -   
"http://localhost:8081/"  "Mozilla/5.0 (Windows NT 6.3; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)   
Chrome/35.0.1916.153 Safari/537.36 SE 2.X MetaSr 1.0" "JSESSIONID=2gyikovul2iz168116l2afo4f"

#### 转储快照

在你的 pom.xml 文件添加如下配置：

<configuration>  
  [...]  
  <dumpOnStart>true</dumpOnStart>  
  [...]  
</configuration>

dumpOnStart 默认值为 false，如果设为 true，jetty 在启动时会把当前服务进程的内存信息输出到控制台中，但这并不会保存到文件中。

#### WEB上下文

最常用的是 contextPath，它的配置如下：

<configuration>  
  [...]  
  <webApp>  
    <contextPath>/${project.artifactId}</contextPath>  
  </webApp>  
  [...]  
</configuration>

contextPath 的默认值的 /，${project.artifactId} 引用了 <artifactId> 节点的值，即项目的名称。  
项目的静态资源文件目录默认是 src/main/webapp，如果静态资源目录有多个，或者不在默认的 src/main/webapp 目录下，可做如下配置：

<configuration>  
  [...]  
  <webApp>  
    <contextPath>/${project.artifactId}</contextPath>  
    <resourceBases>  
      <resourceBase>${project.basedir}/src/main/webapp</resourceBase>  
      <resourceBase>${project.basedir}/commons</resourceBase>  
    </resourceBases>  
  </webApp>  
  [...]  
</configuration>

引用静态资源文件时，路径不包含资源目录的名称，如 commons/main.css，引用方式为：<link href="main.css" rel="stylesheet" />   
更多参数信息可参考 [jetty-maven-plugin.html#configuring-your-webapp](http://www.eclipse.org/jetty/documentation/current/jetty-maven-plugin.html#configuring-your-webapp)

#### 完整的配置

附 pom.xml 文件中 jetty 插件的完整配置片段：

<build>  
  [...]  
  <plugins>  
    <plugin>  
      <groupId>org.eclipse.jetty</groupId>  
      <artifactId>jetty-maven-plugin</artifactId>  
      <version>9.2.8.v20150217</version>  
      <configuration>  
        <httpConnector>  
          <port>8081</port>  
        </httpConnector>  
        <stopKey>shutdown</stopKey>  
        <stopPort>9966</stopPort>  
        <!--  
        <scanIntervalSeconds>2</scanIntervalSeconds>  
        -->  
        <reload>manual</reload>  
        <dumpOnStart>true</dumpOnStart>  
        <webApp>  
          <contextPath>/${project.artifactId}</contextPath>  
          <!--  
          <resourceBases>  
            <resourceBase>${project.basedir}/src/main/webapp</resourceBase>  
            <resourceBase>${project.basedir}/commons</resourceBase>  
          </resourceBases>  
          -->  
        </webApp>  
        <requestLog implementation="org.eclipse.jetty.server.NCSARequestLog">  
          <filename>target/access-yyyy\_mm\_dd.log</filename>  
          <filenameDateFormat>yyyy\_MM\_dd</filenameDateFormat>  
          <logDateFormat>yyyy-MM-dd HH:mm:ss</logDateFormat>  
          <logTimeZone>GMT+8:00</logTimeZone>  
          <append>true</append>  
          <logServer>true</logServer>  
          <retainDays>120</retainDays>  
          <logCookies>true</logCookies>  
        </requestLog>  
      </configuration>  
    </plugin>  
  </plugins>  
  [...]  
</build>

更多有关 jetty 的配置信息可参考 <http://www.eclipse.org/jetty/documentation/current/jetty-maven-plugin.html>

#### 使用IntelliJ IDEA, Maven and the jetty plugin调试web应用程序

**配置Maven：**  
File-->setting-->Maven-->  
修改Maven Home Director, User setting file, Locale repostiory;  
选择相应的配置就可以；  
  
**配置运行环境：**  
打开IntelliJ IDEA ，创建Run/Debug的配置，下拉菜单中选择"Edit configurations"，点击左上角的 '+' (plus) 按钮 ，创建一个新的 "Maven" configuration.   
Working directory：项目根目录  
Command Line：clean jetty:run -DskipTests  
Profile(separaed with space): pom.xml  
  
**运行/调试：**  
点击运行或者调试按钮

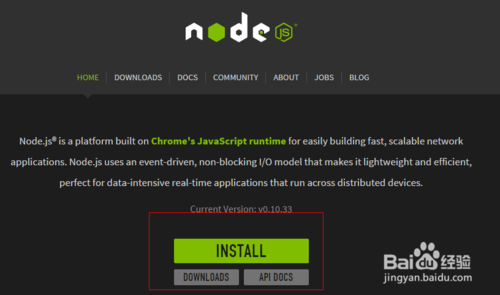
## bower

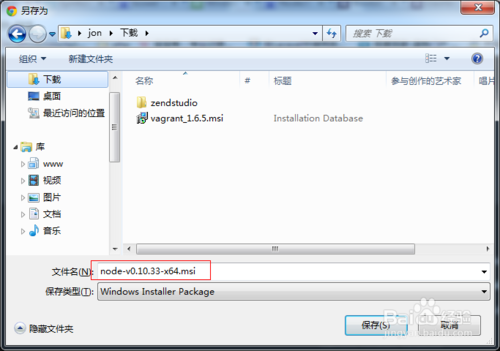
### Windows环境下的NodeJS+NPM+Bower安装配置

Node.js 是一个基于Chrome JavaScript 运行时建立的一个平台， 用来方便地搭建快速的 易于扩展的网络应用· Node.js 借助事件驱动， 非阻塞I/O 模型变得轻量和高效， 非常适合 运行在分布式设备 的 数据密集型 的实时应用。

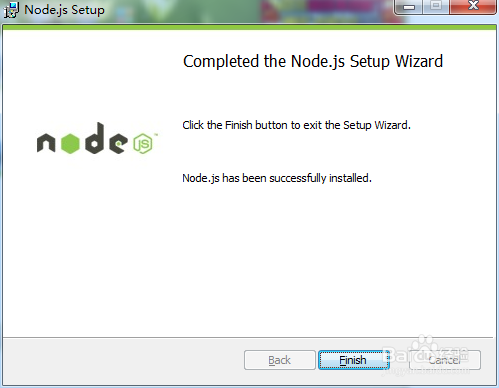
方法/步骤

1. 进入nodejs官方网站下载软件(nodejs.org),

[](http://jingyan.baidu.com/album/2d5afd69e243cc85a2e28efa.html?picindex=1)

[](http://jingyan.baidu.com/album/2d5afd69e243cc85a2e28efa.html?picindex=2)

1. 下载完成后，双击默认安装。安装程序会自动添加环境变量

[](http://jingyan.baidu.com/album/2d5afd69e243cc85a2e28efa.html?picindex=3)

1. 检测nodejs是否安装成功。打开cmd命令行 输入 node - v 显示当前版本号

[](http://jingyan.baidu.com/album/2d5afd69e243cc85a2e28efa.html?picindex=4)

1. 检查npm是否安装。由于新版的NodeJS已经集成了npm，所以之前npm也一并安装好了。同样可以使用cmd命令行中键入 npm -v

[Windows环境下的NodeJS+NPM+Bower安装配置](http://jingyan.baidu.com/album/2d5afd69e243cc85a2e28efa.html?picindex=5)

1. npm作为一个NodeJS的模块管理，之前我由于没有系统地看资料所以导致安装配置模块的时候走了一大段弯路，所以现在很有必要列出来记录下。我们要先配置npm的全局模块的存放路径以及cache的路径，例如我希望将以上两个文件夹放在NodeJS的主目录下，便在NodeJs下建立“node\_global”及“node\_cache”两个文件夹。我们就在cmd中键入两行命令：

npm config set prefix "D:\Program Files\nodejs\node\_global"

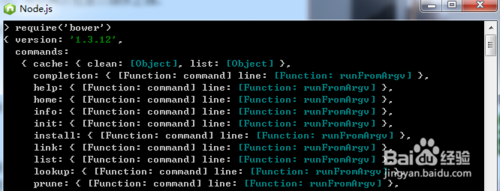
　　和

npm config set cache "D:\Program Files\nodejs\node\_cache"

1. 下面这一步非常关键，我们需要设置系统变量。进入我的电脑→属性→高级→环境变量。在系统变量下新建“NODE\_PATH”，输入“D:\Program Files\nodejs\node\_global\node\_modules”。
2. 安装bower, npm install bower -g; -g表示全局

输入 node，

require('bower') 显示如下表示 安装成功！

[](http://jingyan.baidu.com/album/2d5afd69e243cc85a2e28efa.html?picindex=6)

# 用到的技术

## SpringMVC

### 1 Spring Web MVC是什么

Spring Web MVC是一种基于Java的实现了Web MVC设计模式的请求驱动类型的轻量级Web框架，即使用了MVC架构模式的思想，将web层进行职责解耦，基于请求驱动指的就是使用请求-响应模型，框架的目的就是帮助我们简化开发，Spring Web MVC也是要简化我们日常Web开发的。

另外还有一种基于组件的、事件驱动的Web框架在此就不介绍了，如Tapestry、JSF等。

Spring Web MVC也是服务到工作者模式的实现，但进行可优化。前端控制器是DispatcherServlet；应用控制器其实拆为处理器映射器(Handler Mapping)进行处理器管理和视图解析器(View Resolver)进行视图管理；页面控制器/动作/处理器为Controller接口（仅包含ModelAndView handleRequest(request, response) 方法）的实现（也可以是任何的POJO类）；支持本地化（Locale）解析、主题（Theme）解析及文件上传等；提供了非常灵活的数据验证、格式化和数据绑定机制；提供了强大的约定大于配置（惯例优先原则）的契约式编程支持。

### 2 Spring Web MVC能帮我们做什么

√让我们能非常简单的设计出干净的Web层和薄薄的Web层；

√进行更简洁的Web层的开发；

√天生与Spring框架集成（如IoC容器、AOP等）；

√提供强大的约定大于配置的契约式编程支持；

√能简单的进行Web层的单元测试；

√支持灵活的URL到页面控制器的映射；

√非常容易与其他视图技术集成，如Velocity、FreeMarker等等，因为模型数据不放在特定的API里，而是放在一个Model里（Map数据结构实现，因此很容易被其他框架使用）；

√非常灵活的数据验证、格式化和数据绑定机制，能使用任何对象进行数据绑定，不必实现特定框架的API；

√提供一套强大的JSP标签库，简化JSP开发；

√支持灵活的本地化、主题等解析；

√更加简单的异常处理；

√对静态资源的支持；

√支持Restful风格。

### 3 Spring Web MVC架构

Spring Web MVC框架也是一个基于请求驱动的Web框架，并且也使用了前端控制器模式来进行设计，再根据请求映射规则分发给相应的页面控制器（动作/处理器）进行处理。首先让我们整体看一下Spring Web MVC处理请求的流程：

#### 3.1 Spring Web MVC处理请求的流程



具体执行步骤如下：

1、  首先用户发送请求————>前端控制器，前端控制器根据请求信息（如URL）来决定选择哪一个页面控制器进行处理并把请求委托给它，即以前的控制器的控制逻辑部分；图中的1、2步骤；

2、  页面控制器接收到请求后，进行功能处理，首先需要收集和绑定请求参数到一个对象，这个对象在Spring Web MVC中叫命令对象，并进行验证，然后将命令对象委托给业务对象进行处理；处理完毕后返回一个ModelAndView（模型数据和逻辑视图名）；图中的3、4、5步骤；

3、  前端控制器收回控制权，然后根据返回的逻辑视图名，选择相应的视图进行渲染，并把模型数据传入以便视图渲染；图中的步骤6、7；

4、  前端控制器再次收回控制权，将响应返回给用户，图中的步骤8；至此整个结束。

问题：

1、  请求如何给前端控制器？

2、  前端控制器如何根据请求信息选择页面控制器进行功能处理？

3、  如何支持多种页面控制器呢？

4、  如何页面控制器如何使用业务对象？

5、  页面控制器如何返回模型数据？

6、  前端控制器如何根据页面控制器返回的逻辑视图名选择具体的视图进行渲染？

7、  不同的视图技术如何使用相应的模型数据？

首先我们知道有如上问题，那这些问题如何解决呢？请让我们先继续，在后边依次回答。

#### 3.2、Spring Web MVC架构

1、Spring Web MVC核心架构图，如图2-2



图2-2

架构图对应的DispatcherServlet核心代码如下：

**Java代码**

1. //前端控制器分派方法
2. **protected** **void** doDispatch(HttpServletRequest request, HttpServletResponse response) **throws** Exception {
3. HttpServletRequest processedRequest = request;
4. HandlerExecutionChain mappedHandler = **null**;
5. **int** interceptorIndex = -1;
7. **try** {
8. ModelAndView mv;
9. **boolean** errorView = **false**;
11. **try** {
12. //检查是否是请求是否是multipart（如文件上传），如果是将通过MultipartResolver解析
13. processedRequest = checkMultipart(request);
14. //步骤2、请求到处理器（页面控制器）的映射，通过HandlerMapping进行映射
15. mappedHandler = getHandler(processedRequest, **false**);
16. **if** (mappedHandler == **null** || mappedHandler.getHandler() == **null**) {
17. noHandlerFound(processedRequest, response);
18. **return**;
19. }
20. //步骤3、处理器适配，即将我们的处理器包装成相应的适配器（从而支持多种类型的处理器）
21. HandlerAdapter ha = getHandlerAdapter(mappedHandler.getHandler());
23. // 304 Not Modified缓存支持
24. //此处省略具体代码
26. // 执行处理器相关的拦截器的预处理（HandlerInterceptor.preHandle）
27. //此处省略具体代码
29. // 步骤4、由适配器执行处理器（调用处理器相应功能处理方法）
30. mv = ha.handle(processedRequest, response, mappedHandler.getHandler());
32. // Do we need view name translation?
33. **if** (mv != **null** && !mv.hasView()) {
34. mv.setViewName(getDefaultViewName(request));
35. }
37. // 执行处理器相关的拦截器的后处理（HandlerInterceptor.postHandle）
38. //此处省略具体代码
39. }
40. **catch** (ModelAndViewDefiningException ex) {
41. logger.debug("ModelAndViewDefiningException encountered", ex);
42. mv = ex.getModelAndView();
43. }
44. **catch** (Exception ex) {
45. Object handler = (mappedHandler != **null** ? mappedHandler.getHandler() : **null**);
46. mv = processHandlerException(processedRequest, response, handler, ex);
47. errorView = (mv != **null**);
48. }
50. //步骤5 步骤6、解析视图并进行视图的渲染
51. //步骤5 由ViewResolver解析View（viewResolver.resolveViewName(viewName, locale)）
52. //步骤6 视图在渲染时会把Model传入（view.render(mv.getModelInternal(), request, response);）
53. **if** (mv != **null** && !mv.wasCleared()) {
54. render(mv, processedRequest, response);
55. **if** (errorView) {
56. WebUtils.clearErrorRequestAttributes(request);
57. }
58. }
59. **else** {
60. **if** (logger.isDebugEnabled()) {
61. logger.debug("Null ModelAndView returned to DispatcherServlet with name '" + getServletName() +
62. "': assuming HandlerAdapter completed request handling");
63. }
64. }
66. // 执行处理器相关的拦截器的完成后处理（HandlerInterceptor.afterCompletion）
67. //此处省略具体代码

70. **catch** (Exception ex) {
71. // Trigger after-completion for thrown exception.
72. triggerAfterCompletion(mappedHandler, interceptorIndex, processedRequest, response, ex);
73. **throw** ex;
74. }
75. **catch** (Error err) {
76. ServletException ex = **new** NestedServletException("Handler processing failed", err);
77. // Trigger after-completion for thrown exception.
78. triggerAfterCompletion(mappedHandler, interceptorIndex, processedRequest, response, ex);
79. **throw** ex;
80. }
82. **finally** {
83. // Clean up any resources used by a multipart request.
84. **if** (processedRequest != request) {
85. cleanupMultipart(processedRequest);
86. }
87. }
88. }

核心架构的具体流程步骤如下：

1、  首先用户发送请求——>DispatcherServlet，前端控制器收到请求后自己不进行处理，而是委托给其他的解析器进行处理，作为统一访问点，进行全局的流程控制；

2、  DispatcherServlet——>HandlerMapping， HandlerMapping将会把请求映射为HandlerExecutionChain对象（包含一个Handler处理器（页面控制器）对象、多个HandlerInterceptor拦截器）对象，通过这种策略模式，很容易添加新的映射策略；

3、  DispatcherServlet——>HandlerAdapter，HandlerAdapter将会把处理器包装为适配器，从而支持多种类型的处理器，即适配器设计模式的应用，从而很容易支持很多类型的处理器；

4、  HandlerAdapter——>处理器功能处理方法的调用，HandlerAdapter将会根据适配的结果调用真正的处理器的功能处理方法，完成功能处理；并返回一个ModelAndView对象（包含模型数据、逻辑视图名）；

5、  ModelAndView的逻辑视图名——> ViewResolver， ViewResolver将把逻辑视图名解析为具体的View，通过这种策略模式，很容易更换其他视图技术；

6、  View——>渲染，View会根据传进来的Model模型数据进行渲染，此处的Model实际是一个Map数据结构，因此很容易支持其他视图技术；

7、返回控制权给DispatcherServlet，由DispatcherServlet返回响应给用户，到此一个流程结束。

此处我们只是讲了核心流程，没有考虑拦截器、本地解析、文件上传解析等，后边再细述。

到此，再来看我们前边提出的问题：

1、  请求如何给前端控制器？这个应该在web.xml中进行部署描述，在HelloWorld中详细讲解。

2、  前端控制器如何根据请求信息选择页面控制器进行功能处理？ 我们需要配置HandlerMapping进行映射

3、  如何支持多种页面控制器呢？配置HandlerAdapter从而支持多种类型的页面控制器

4、  如何页面控制器如何使用业务对象？可以预料到，肯定利用Spring IoC容器的依赖注入功能

5、  页面控制器如何返回模型数据？使用ModelAndView返回

6、  前端控制器如何根据页面控制器返回的逻辑视图名选择具体的视图进行渲染？ 使用ViewResolver进行解析

7、  不同的视图技术如何使用相应的模型数据？ 因为Model是一个Map数据结构，很容易支持其他视图技术

在此我们可以看出具体的核心开发步骤：

1、  DispatcherServlet在web.xml中的部署描述，从而拦截请求到Spring Web MVC

2、  HandlerMapping的配置，从而将请求映射到处理器

3、  HandlerAdapter的配置，从而支持多种类型的处理器

4、  ViewResolver的配置，从而将逻辑视图名解析为具体视图技术

5、处理器（页面控制器）的配置，从而进行功能处理

### 4 Spring Web MVC优势

1、清晰的角色划分：前端控制器（DispatcherServlet）、请求到处理器映射（HandlerMapping）、处理器适配器（HandlerAdapter）、视图解析器（ViewResolver）、处理器或页面控制器（Controller）、验证器（   Validator）、命令对象（Command  请求参数绑定到的对象就叫命令对象）、表单对象（Form Object 提供给表单展示和提交到的对象就叫表单对象）。

2、分工明确，而且扩展点相当灵活，可以很容易扩展，虽然几乎不需要；

3、由于命令对象就是一个POJO，无需继承框架特定API，可以使用命令对象直接作为业务对象；

4、和Spring 其他框架无缝集成，是其它Web框架所不具备的；

5、可适配，通过HandlerAdapter可以支持任意的类作为处理器；

6、可定制性，HandlerMapping、ViewResolver等能够非常简单的定制；

7、功能强大的数据验证、格式化、绑定机制；

8、利用Spring提供的Mock对象能够非常简单的进行Web层单元测试；

9、本地化、主题的解析的支持，使我们更容易进行国际化和主题的切换。

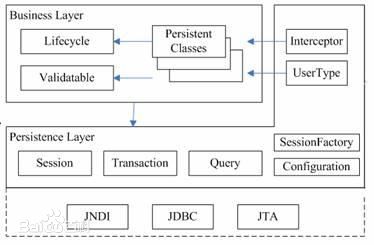
10、强大的JSP标签库，使JSP编写更容易。

………………还有比如RESTful风格的支持、简单的文件上传、约定大于配置的契约式编程支持、基于注解的零配置支持等等。

## Hibernate

Hibernate是一个开放源代码的对象关系映射框架，它对JDBC进行了非常轻量级的对象封装，使得Java程序员可以随心所欲的使用对象编程思维来操纵数据库。 Hibernate可以应用在任何使用JDBC的场合，既可以在Java的客户端程序使用，也可以在Servlet/JSP的Web应用中使用，最具革命意义的是，Hibernate可以在应用EJB的J2EE架构中取代CMP，完成数据持久化的重任。

### 1核心接口和类



Hibernate的核心类和接口一共有6个，分别为:Session、SessionFactory、Transaction、Query、Criteria和Configuration。这6个核心类和接口在任何开发中都会用到。通过这些接口，不仅可以对持久化对象进行存取，还能够进行事务控制。下面对这6个核心接口和类分别加以介绍。

#### Session

Session接口负责执行被持久化对象的CRUD操作(CRUD的任务是完成与数据库的交流，包含了很多常见的SQL语句)。但需要注意的是Session对象是非线程安全的。同时，Hibernate的session不同于JSP应用中的HttpSession。这里当使用session这个术语时，其实指的是Hibernate中的session，而以后会将HttpSession对象称为用户session。

#### SessionFactory

SessionFactory接口负责初始化Hibernate。它充当数据存储源的代理，并负责创建Session对象。这里用到了工厂模式。需要注意的是SessionFactory并不是轻量级的，因为一般情况下，一个项目通常只需要一个SessionFactory就够，当需要操作多个数据库时，可以为每个数据库指定一个SessionFactory。

#### Transaction

Transaction 接口是一个可选的API，可以选择不使用这个接口，取而代之的是Hibernate 的设计者自己写的底层事务处理代码。 Transaction 接口是对实际事务实现的一个抽象，这些实现包括JDBC的事务、JTA 中的UserTransaction、甚至可以是CORBA 事务。之所以这样设计是能让开发者能够使用一个统一事务的操作界面，使得自己的项目可以在不同的环境和容器之间方便地移植。

#### Query

Query接口让你方便地对数据库及持久对象进行查询，它可以有两种表达方式：HQL语言或本地数据库的SQL语句。Query经常被用来绑定查询参数、限制查询记录数量，并最终执行查询操作。

#### Criteria

Criteria接口与Query接口非常类似，允许创建并执行面向对象的标准化查询。值得注意的是Criteria接口也是轻量级的，它不能在Session之外使用。

#### Configuration

Configuration 类的作用是对Hibernate 进行配置，以及对它进行启动。在Hibernate 的启动过程中，Configuration 类的实例首先定位映射文档的位置，读取这些配置，然后创建一个SessionFactory对象。虽然Configuration 类在整个Hibernate 项目中只扮演着一个很小的角色，但它是启动hibernate 时所遇到的第一个对象。

### 2主键介绍

#### Assigned

Assigned方式由用户生成主键值，并且要在save()之前指定否则会抛出异常

特点：主键的生成值完全由用户决定，与底层数据库无关。用户需要维护主键值，在调用session.save()之前要指定主键值。

#### Hilo

Hilo使用高低位算法生成主键，高低位算法使用一个高位值和一个低位值，然后把算法得到的两个值拼接起来作为数据库中的唯一主键。Hilo方式需要额外的数据库表和字段提供高位值来源。默认情况下使用的表是hibernate\_unique\_key，默认字段叫作next\_hi。next\_hi必须有一条记录否则会出现错误。

特点：需要额外的数据库表的支持，能保证同一个数据库中主键的唯一性，但不能保证多个数据库之间主键的唯一性。Hilo主键生成方式由Hibernate 维护，所以Hilo方式与底层数据库无关，但不应该手动修改hi/lo算法使用的表的值，否则会引起主键重复的异常。

#### Increment

Increment方式对主键值采取自动增长的方式生成新的主键值，但要求底层数据库的主键类型为long,int等数值型。主键按数值顺序递增，增量为1。

/\*特点：由Hibernate本身维护，适用于所有的数据库，不适合多进程并发更新数据库，适合单一进程访问数据库。不能用于群集环境。\*/

#### Identity

Identity方式根据底层数据库，来支持自动增长，不同的数据库用不同的主键增长方式。

特点：与底层数据库有关，要求数据库支持Identity，如MySQl中是auto\_increment, SQL Server 中是Identity，支持的数据库有MySql、SQL Server、DB2、Sybase和HypersonicSQL。 Identity无需Hibernate和用户的干涉，使用较为方便，但不便于在不同的数据库之间移植程序。

#### Sequence

Sequence需要底层数据库支持Sequence方式，例如Oracle数据库等

特点：需要底层数据库的支持序列，支持序列的数据库有DB2、PostgreSql、Oracle、SAPDb等在不同数据库之间移植程序，特别从支持序列的数据库移植到不支持序列的数据库需要修改配置文件。

#### Native

Native主键生成方式会根据不同的底层数据库自动选择Identity、Sequence、Hilo主键生成方式

特点：根据不同的底层数据库采用不同的主键生成方式。由于Hibernate会根据底层数据库采用不同的映射方式，因此便于程序移植，项目中如果用到多个数据库时，可以使用这种方式。

#### UUID

UUID使用128位UUID算法生成主键，能够保证网络环境下的主键唯一性，也就能够保证在不同数据库及不同服务器下主键的唯一性。

特点：能够保证数据库中的主键唯一性，生成的主键占用比较多的存贮空间

#### Foreign GUID

Foreign用于一对一关系中。GUID主键生成方式使用了一种特殊算法，保证生成主键的唯一性，支持SQL Server和MySQL

### 3包的作用

**net.sf.hibernate.\***

该包的类基本上都是接口类和异常类

**net.sf.hibernate.cache.\***

JCS的实现类

**net.sf.hibernate.cfg.\***

配置文件读取类

**net.sf.hibernate.collection.\***

Hibernate集合接口实现类，例如List，Set，Bag等等，Hibernate之所以要自行编写集合接口实现类是为了支持lazy loading

**net.sf.hibernate.connection.\***

几个数据库连接池的Provider

**net.sf.hibernate.dialect.\***

支持多种数据库特性，每个Dialect实现类代表一种数据库，描述了该数据库支持的数据类型和其它特点，例如是否有AutoIncrement，是否有Sequence，是否有分页sql等等

**net.sf.hibernate. eg.\***

Hibernate文档中用到的例子

**net.sf.hibernate.engine.\***

这个包的类作用比较散

**net.sf.hibernate.expression.\***

HQL支持的表达式

**net.sf.hibernate.hq.\***

HQL实现

**net.sf.hibernate. id.\***

ID生成器

**net.sf.hibernate.impl.\***

最核心的包，一些重要接口的实现类，如Session，SessionFactory，Query等

**net.sf.hibernate.jca.\***

JCA支持，把Session包装为支持JCA的接口实现类

**net.sf.hibernate.jmx.\***

JMX是用来编写App Server的管理程序的，大概是JMX部分接口的实现，使得App Server可以通过JMX接口管理Hibernate

**net.sf.hibernate.loader.\***

也是很核心的包，主要是生成sql语句的

**net.sf.hibernate.lob.\***

Blob和Clob支持

**net.sf.hibernate.mapping.\***

hbm文件的属性实现

**net.sf.hibernate.metadata.\***

PO的Meta实现

**net.sf.hibernate.odmg.\***

ODMG是一个ORM标准，这个包是ODMG标准的实现类

**net.sf.hibernate.persister.\***

核心包，实现持久对象和表之间的映射

**net.sf.hibernate.proxy.\***

Proxy和Lazy Loading支持

**net.sf.hibernate. ps.\***

该包是PreparedStatment Cache

**net.sf.hibernate.sql.\***

生成JDBC sql语句的包

**net.sf.hibernate.test.\***

测试类，你可以用junit来测试Hibernate

**net.sf.hibernate.tool.hbm2ddl.\***

用hbm配置文件生成DDL

**net.sf.hibernate.transaction.\***

Hibernate Transaction实现类

**net.sf.hibernate.type.\***

Hibernate中定义的持久对象的属性的数据类型

**net.sf.hibernate.util.\***

一些工具类，作用比较散

**net.sf.hibernate.xml.\***

XML数据绑定

### 4缓存管理

Hibernate 中提供了两级Cache（高速缓冲存储器），第一级别的缓存是Session级别的缓存，它是属于事务范围的缓存。这一级别的缓存由hibernate管理的，一般情况下无需进行干预；第二级别的缓存是SessionFactory级别的缓存，它是属于进程范围或群集范围的缓存。这一级别的缓存可以进行配置和更改，并且可以动态加载和卸载。 Hibernate还为查询结果提供了一个查询缓存，它依赖于第二级缓存。

第一级缓存 第二级缓存 存放数据的形式 相互关联的持久化对象的散装数据 缓存的范围 事务范围，每个事务都有单独的第一级缓存进程范围或集群范围，缓存被同一个进程或集群范围内的所有事务共享 并发访问策略由于每个事务都拥有单独的第一级缓存，不会出现并发问题，无需提供并发访问策略由于多个事务会同时访问第二级缓存中相同数据，因此必须提供适当的并发访问策略，来保证特定的事务隔离级别数据过期策略没有提供数据过期策略。处于一级缓存中的对象永远不会过期，除非应用程序显式清空缓存或者清除特定的对象必须提供数据过期策略，如基于内存的缓存中的对象的最大数目，允许对象处于缓存中的最长时间，以及允许对象处于缓存中的最长空闲时间 物理存储介质内存和硬盘。对象的散装数据首先存放在基于内存的缓存中，当内存中对象的数目达到数据过期策略中指定上限时，就会把其余的对象写入基于硬盘的缓存中。缓存的软件实现 在Hibernate的Session的实现中包含了缓存的实现由第三方提供，Hibernate仅提供了缓存适配器(CacheProvider)。用于把特定的缓存插件集成到Hibernate中。启用缓存的方式只要应用程序通过Session接口来执行保存、更新、删除、加载和查询数据库数据的操作，Hibernate就会启用第一级缓存，把数据库中的数据以对象的形式拷贝到缓存中，对于批量更新和批量删除操作，如果不希望启用第一级缓存，可以绕过Hibernate API，直接通过JDBC　API来执行指操作。用户可以在单个类或类的单个集合的粒度上配置第二级缓存。如果类的实例被经常读但很少被修改，就可以考虑使用第二级缓存。只有为某个类或集合配置了第二级缓存，Hibernate在运行时才会把它的实例加入到第二级缓存中。 用户管理缓存的方式第一级缓存的物理介质为内存，由于内存容量有限，必须通过恰当的检索策略和检索方式来限制加载对象的数目。Session的evit()方法可以显式清空缓存中特定对象，但这种方法不值得推荐。 第二级缓存的物理介质可以是内存和硬盘，因此第二级缓存可以存放大量的数据，数据过期策略的maxElementsInMemory属性值可以控制内存中的对象数目。管理第二级缓存主要包括两个方面：选择需要使用第二级缓存的持久类，设置合适的并发访问策略：选择缓存适配器，设置合适的数据过期策略。

一级缓存

当应用程序调用Session的save()、update()、saveOrUpdate()、get()或load()，以及调用查询接口的 list()、iterate()或filter()方法时，如果在Session缓存中还不存在相应的对象，Hibernate就会把该对象加入到第一级缓存中。当清理缓存时，Hibernate会根据缓存中对象的状态变化来同步更新数据库。

Session为应用程序提供了两个管理缓存的方法：

evict(Object obj)：从缓存中清除参数指定的持久化对象。

clear()：清空缓存中所有持久化对象。

二级缓存

1. Hibernate的二级缓存策略的一般过程如下：

1) 条件查询的时候，总是发出一条select \* from table\_name where …. （选择所有字段）这样的SQL语句查询数据库，一次获得所有的数据对象。

2) 把获得的所有数据对象根据ID放入到第二级缓存中。

3) 当Hibernate根据ID访问数据对象的时候，首先从Session一级缓存中查；查不到，如果配置了二级缓存，那么从二级缓存中查；查不到，再查询数据库，把结果按照ID放入到缓存。

4) 删除、更新、增加数据的时候，同时更新缓存。

Hibernate的二级缓存策略，是针对于ID查询的缓存策略，对于条件查询则毫无作用。为此，Hibernate提供了针对条件查询的Query Cache。

2. 什么样的数据适合存放到第二级缓存中？

* 很少被修改的数据
* 不是很重要的数据，允许出现偶尔并发的数据
* 不会被并发访问的数据
* 参考数据,指的是供应用参考的常量数据，它的实例数目有限，它的实例会被许多其他类的实例引用，实例极少或者从来不会被修改。

3. 不适合存放到第二级缓存的数据？

* 经常被修改的数据
* 财务数据，绝对不允许出现并发
* 与其他应用共享的数据。

4. 常用的缓存插件 Hibernater 的二级缓存是一个插件，下面是几种常用的缓存插件：

EhCache：可作为进程范围的缓存，存放数据的物理介质可以是内存或硬盘，对Hibernate的查询缓存提供了支持。

OSCache：可作为进程范围的缓存，存放数据的物理介质可以是内存或硬盘，提供了丰富的缓存数据过期策略，对Hibernate的查询缓存提供了支持。

SwarmCache：可作为群集范围内的缓存，但不支持Hibernate的查询缓存。

JBossCache：可作为群集范围内的缓存，支持事务型并发访问策略，对Hibernate的查询缓存提供了支持。

上述4种缓存插件的对比情况列于表9-3中。

表9-3 4种缓存插件的对比情况

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 缓存插件 | 支持只读 | 支持非严格读写 | 支持读写 | 支持事务 |
| EhCache | 是 | 是 | 是 |  |
| OSCache | 是 | 是 | 是 |  |
| SwarmCache | 是 | 是 |  |  |
| JBossCache | 是 |  |  | 是 |

它们的提供器列于表9-4中。

表9-4 缓存策略的提供器

|  |  |
| --- | --- |
| 缓 存 插 件 | 提供器（Cache Providers） |
| Hashtable（只能测试时使用） | org.hibernate.cache.HashtableCacheProvider |
| EhCache | org.hibernate.cache.EhCacheProvider |
| OSCache | org.hibernate.cache.OSCacheProvider |

在默认情况下，Hibernate使用EhCache进行JVM级别的缓存。用户可以通过设置Hibernate配置文件中的hibernate.cache.provider\_class的属性，指定其他的缓存策略，该缓存策略必须实现org.hibernate.cache.CacheProvider接口。

5. 配置二级缓存的主要步骤：

1) 选择需要使用二级缓存的持久化类，设置它的命名缓存的并发访问策略。这是最值得认真考虑的步骤。

2) 选择合适的缓存插件，然后编辑该插件的配置文件。

### 5延迟加载

#### 延迟加载

Hibernate对象关系映射提供延迟的与非延迟的对象初始化。非延迟加载在读取一个对象的时候会将与这个对象所有相关的其他对象一起读取出来。这有时会导致成百的（如果不是成千的话）select语句在读取对象的时候执行。这个问题有时出现在使用双向关系的时候，经常会导致整个数据库都在初始化的阶段被读出来了。当然，你可以不厌其烦地检查每一个对象与其他对象的关系，并把那些最昂贵的删除，但是到最后，我们可能会因此失去了本想在ORM工具中获得的便利。

一个明显的解决方法是使用Hibernate提供的延迟加载机制。这种初始化策略只在一个对象调用它的一对多或多对多关系时才将关系对象读取出来。这个过程对开发者来说是透明的，而且只进行了很少的数据库操作请求，因此会得到比较明显的性能提升。这项技术的一个缺陷是延迟加载技术要求一个Hibernate会话要在对象使用的时候一直开着。这会成为通过使用DAO模式将持久层抽象出来时的一个主要问题。为了将持久化机制完全地抽象出来，所有的数据库逻辑，包括打开或关闭会话，都不能在应用层出现。最常见的是，一些实现了简单接口的DAO实现类将数据库逻辑完全封装起来了。一种快速但是笨拙的解决方法是放弃DAO模式，将数据库连接逻辑加到应用层中来。这可能对一些小的应用程序有效，但是在大的系统中，这是一个严重的设计缺陷，妨碍了系统的可扩展性。

#### Web层延迟加载

幸运的是，Spring框架为Hibernate延迟加载与DAO模式的整合提供了一种方便的解决方法。以一个Web应用为例，Spring提供了OpenSessionInViewFilter和OpenSessionInViewInterceptor。我们可以随意选择一个类来实现相同的功能。两种方法唯一的不同就在于interceptor在Spring容器中运行并被配置在web应用的上下文中，而Filter在Spring之前运行并被配置在web.xml中。不管用哪个，他们都在请求将当前会话与当前（数据库）线程绑定时打开Hibernate会话。一旦已绑定到线程，这个打开了的Hibernate会话可以在DAO实现类中透明地使用。这个会话会为延迟加载数据库中值对象的视图保持打开状态。一旦这个逻辑视图完成了，Hibernate会话会在Filter的doFilter方法或者Interceptor的postHandle方法中被关闭。

实现方法在web.xml中加入

<filter>

<filter-name>hibernateFilter</filter-name>

<filter-class>

org.springframework.orm.hibernate3.support.OpenSessionInViewFilter

</filter-class>

</filter>

<filter-mapping>

<filter-name>hibernateFilter</filter-name>

<url-pattern>\*.do</url-pattern>

</filter-mapping>

### 6性能优化

初用HIBERNATE的人也许都遇到过性能问题，实现同一功能，用HIBERNATE与用JDBC性能相差十几倍很正常，如果不及早调整，很可能影响整个项目的进度。 大体上，对于HIBERNATE性能调优的主要考虑点如下：

.数据库设计调整

.HQL优化

.API的正确使用(如根据不同的业务类型选用不同的集合及查询API)

.主配置参数(日志，查询缓存，fetch\_size, batch\_size等)

.映射文件优化(ID生成策略，二级缓存，延迟加载，关联优化)

.一级缓存的管理

.针对二级缓存，还有许多特有的策略

.事务控制策略。

#### 数据库设计

a) 降低关联的复杂性

b) 尽量不使用联合主键

c) ID的生成机制，不同的数据库所提供的机制并不完全一样

d) 适当的冗余数据，不过分追求高范式

#### HQL优化

HQL如果抛开它同HIBERNATE本身一些缓存机制的关联，HQL的优化技巧同普通的SQL优化技巧一样，可以很容易在网上找到一些经验之谈。

#### 主配置

a) 查询缓存，同下面讲的缓存不太一样，它是针对HQL语句的缓存，即完全一样的语句再次执行时可以利用缓存数据。但是，查询缓存在一个交易系统(数据变更频繁，查询条件相同的机率并不大)中可能会起反作用:它会白白耗费大量的系统资源但却难以派上用场。

b) fetch\_size，同JDBC的相关参数作用类似，参数并不是越大越好，而应根据业务特征去设置

c) batch\_size同上。

d) 生产系统中，切记要关掉SQL语句打印。

#### 缓存

a) 数据库级缓存:这级缓存是最高效和安全的，但不同的数据库可管理的层次并不一样，比如，在Oracle中，可以在建表时指定将整个表置于缓存当中。

b) SESSION缓存:在一个HibernateSESSION有效，这级缓存的可干预性不强，大多于HIBERNATE自动管理，但它提供清除缓存的方法，这在大批量增加/更新操作是有效的。比如，同时增加十万条记录，按常规方式进行，很可能会发现OutofMemeroy的异常，这时可能需要手动清除这一级缓存:Session.evict以及 Session.clear

c) 应用缓存:在一个SESSIONFACTORY中有效，因此也是优化的重中之重，因此，各类策略也考虑的较多，在将数据放入这一级缓存之前，需要考虑一些前提条件：

i. 数据不会被第三方修改(比如，是否有另一个应用也在修改这些数据?)

ii. 数据不会太大

iii. 数据不会频繁更新(否则使用CACHE可能适得其反)

iv. 数据会被频繁查询

v. 数据不是关键数据(如涉及钱，安全等方面的问题)。

缓存有几种形式，可以在映射文件中配置:read-only(只读，适用于很少变更的静态数据/历史数据)，nonstrict-read- write，read-write(比较普遍的形式，效率一般)，transactional(JTA中，且支持的缓存产品较少)

d) 分布式缓存:同c)的配置一样，只是缓存产品的选用不同，oscache, jboss cache，的大多数项目，对它们的用于集群的使用(特别是关键交易系统)都持保守态度。在集群环境中，只利用数据库级的缓存是最安全的。

#### 延迟加载

a) 实体延迟加载:通过使用动态代理实现

b) 集合延迟加载:通过实现自有的SET/LIST，HIBERNATE提供了这方面的支持

c) 属性延迟加载:

#### 方法选用

a) 完成同样一件事，Hibernate提供了可供选择的一些方式，但具体使用什么方式，可能用性能/代码都会有影响。显示，一次返回十万条记录 (List/Set/Bag/Map等)进行处理，很可能导致内存不够的问题，而如果用基于[游标](http://baike.baidu.com/view/176618.htm)(ScrollableResults)或 Iterator的[结果集](http://baike.baidu.com/view/691831.htm)，则不存在这样的问题。

b) Session的load/get方法，前者会使用[二级缓存](http://baike.baidu.com/view/27650.htm)，而后者则不使用。

c) Query和list/iterator，如果去仔细研究一下它们，你可能会发现很多有意思的情况，二者主要区别(如果使用了Spring，在HibernateTemplate中对应find,iterator方法):

i. list只能利用查询缓存(但在交易系统中查询缓存作用不大)，无法利用二级缓存中的单个实体，但list查出的对象会写入二级缓存，但它一般只生成较少的执行SQL语句，很多情况就是一条(无关联)。

ii. iterator则可以利用二级缓存，对于一条查询语句，它会先从数据库中找出所有符合条件的记录的ID，再通过ID去缓存找，对于缓存中没有的记录，再构造语句从数据库中查出，因此很容易知道，如果缓存中没有任何符合条件的记录，使用iterator会产生N+1条SQL语句(N为符合条件的记录数)

iii. 通过iterator，配合缓存管理API，在海量数据查询中可以很好的解决内存问题，如:

while(it.hasNext()){

YouObject object = (YouObject)it.next();

session.evict(youObject);

sessionFactory.evice(YouObject.class, youObject.getId());

}

如果用list方法，很可能就出OutofMemory错误了。

#### 集合的选用

在Hibernate3.1文档的“19.5. Understanding Collection performance”中有详细的说明。

#### 事务控制

事务方面对性能有影响的主要包括:事务方式的选用，事务隔离级别以及锁的选用

a) 事务方式选用:如果不涉及多个事务管理器事务的话，不需要使用JTA，只有

JDBC的事务控制就可以。

b) 事务隔离级别:参见标准的SQL事务隔离级别

c) 锁的选用:悲观锁(一般由具体的事务管理器实现)，对于长事务效率低，但安全。乐观锁(一般在应用级别实现)，如在HIBERNATE中可以定义 VERSION字段，显然，如果有多个应用操作数据，且这些应用不是用同一种乐观锁机制，则乐观锁会失效。因此，针对不同的数据应有不同的策略，同前面许多情况一样，很多时候我们是在效率与安全/准确性上找一个平衡点，无论如何，优化都不是一个纯技术的问题，你应该对你的应用和业务特征有足够的了解。

#### 批量操作

即使是使用JDBC，在进行大批数据更新时，BATCH与不使用BATCH有效率上也有很大的差别。可以通过设置batch\_size来让其支持批量操作。

举个例子，要批量删除某表中的对象，如“delete Account”，打出来的语句，HIBERNATE找出了所有ACCOUNT的ID，再进行删除，这主要是为了维护二级缓存，这样效率肯定高不了，在后续的版本中增加了bulk delete/update，但这也无法解决缓存的维护问题。也就是说，由于有了二级缓存的维护问题，HIBERNATE的批量操作效率并不尽如人意。

**hibernate工作原理：**

1、通过Configuration().configure();读取并解析hibernate.cfg.xml配置文件。

2、由hibernate.cfg.xml中的<mappingresource="com/xx/User.hbm.xml"/>读取解析映射信息。

3、通过config.buildSessionFactory();//得到sessionFactory。

4、sessionFactory.openSession();//得到session。

5、session.beginTransaction();//开启事务。

6、persistent operate;

7、session.getTransaction().commit();//提交事务

8、关闭session;

9、关闭sessionFactory;

**hibernate优点：**

1、封装了jdbc，简化了很多重复性代码。

2、简化了DAO层编码工作，使开发更对象化了。

3、移植性好，支持各种数据库，如果换个数据库只要在配置文件中变换配置就可以了，不用改变hibernate代码。

4、支持透明持久化，因为hibernate操作的是纯粹的（pojo）java类，没有实现任何接口，没有侵入性。所以说它是一个轻量级框架。

**hibernate**延迟加载：

get不支持延迟加载，load支持延迟加载。

1、hibernate2对 实体对象和集合 实现了延迟加载

2、hibernate3对 提供了属性的延迟加载功能

hibernate延迟加载就是当使用session.load(User.class,1)或者session.createQuery()查询对象或者属性的时候

这个对象或者属性并没有在内存中，只有当程序操作数据的时候，才会存在内存中，这样就实现延迟加载，节省了内存的开销，从而提高了服务器的性能。

Hibernate的缓存机制

一级缓存：session级的缓存也叫事务级的缓存，只缓存实体，生命周期和session一致。不能对其进行管理。

不用显示的调用。

二级缓存：sessionFactory缓存，也叫进程级的缓存，使用第3方插件实现的，也值缓存实体，生命周期和sessionFactory一致，可以进行管理。

首先配置第3方插件，我们用的是EHCache，在hibernate.cfg.xml文件中加入

<propertyname="hibernate.cache.user\_second\_level\_cache">true</property>

在映射中也要显示的调用，<cacheusage="read-only"/>

二级缓存之查询缓存：对普通属性进行缓存。如果关联的表发生了修改，那么查询缓存的生命周期也结束了。

在程序中必须手动启用查询缓存：query.setCacheable(true);

**优化Hibernate**

1、使用一对多的双向关联，尽量从多的一端维护。

2、不要使用一对一，尽量使用多对一。

3、配置对象缓存，不要使用集合缓存。

4、表字段要少，表关联不要怕多，有二级缓存撑腰。

**hibernate 类与类之间关系**

关联关系

聚集关系

继承关系

Hibernate继承关系映射策略分为三种：一张表对应一整棵类继承树、一个类对应一张表、每一个具体类对应一张表。

### 7 Hibernate和Mybatis的区别

Hibernate和Mybatis都是orm对象关系映射框架，都是用于将数据持久化的框架技术。

Hiberante较深度的封装了jdbc，对开发者写sql的能力要求的不是那么的高，我们只要通过hql语句操作对象即可完成对数据持久化的操作了。

另外hibernate可移植性好，如一个项目开始使用的是mysql数据库，但是随着业务的发展，现mysql数据库已经无法满足当前的绣球了，现在决定使用Oracle数据库，虽然sql标准定义的数据库间的sql语句差距不大，但是不同的数据库sql标准还是有差距的，那么我们手动修改起来会存在很大的困难，使用hibernate只需改变一下数据库方言即可搞定。用hibernate框架，数据库的移植变的非常方便。

但是hibernate也存在着诸多的不足，比如在实际开发过程中会生成很多不必要的sql语句耗费程序资源，优化起来也不是很方便，且对存储过程支持的也不够太强大。但是针对于hibernate它也提供了一些优化策略，比如说懒加载、缓存、策略模式等都是针对于它的优化方案。

Mybatis 也是对jdbc的封装，但是封装的没有hibernate那么深，我们可以再配置文件中写sql语句，可以根据需求定制sql语句，数据优化起来较hibernate容易很多。

Mybatis要求程序员写sql的能力要相对使用hibernate的开发人员要高的多，且可移植性也不是很好。

涉及到大数据的系统使用Mybatis比较好，因为优化较方便。涉及的数据量不是很大且对优化没有那么高，可以使用hibernate

### 8 Druid

https://github.com/alibaba/druid/wiki/%E5%B8%B8%E8%A7%81%E9%97%AE%E9%A2%98

**Druid是什么？**

Druid首先是一个数据库连接池。Druid是目前最好的数据库连接池，在功能、性能、扩展性方面，都超过其他数据库连接池，包括DBCP、C3P0、BoneCP、Proxool、JBoss DataSource。

Druid已经在阿里巴巴部署了超过600个应用，经过一年多生产环境大规模部署的严苛考验。

同时Druid不仅仅是一个数据库连接池，它包括四个部分：  
Druid是一个JDBC组件，它包括三个部分：

* 基于Filter－Chain模式的插件体系。
* [DruidDataSource](http://code.alibabatech.com/wiki/display/Druid/DruidDataSource) 高效可管理的数据库连接池。
* [SQLParser](http://code.alibabatech.com/wiki/display/Druid/SQL+Parser)

**Druid可以做什么？**

* 替换DBCP和C3P0。Druid提供了一个高效、功能强大、可扩展性好的数据库连接池。
* 可以监控数据库访问性能，Druid内置提供了一个功能强大的[StatFilter](http://code.alibabatech.com/wiki/display/Druid/StatFilter)插件，能够详细统计SQL的执行性能，这对于线上分析数据库访问性能有帮助。
* 数据库密码加密。直接把数据库密码写在配置文件中，这是不好的行为，容易导致安全问题。DruidDruiver和DruidDataSource都支持PasswordCallback。
* SQL执行日志，Druid提供了不同的LogFilter，能够支持Common-Logging、Log4j和JdkLog，你可以按需要选择相应的LogFilter，监控你应用的数据库访问情况。
* 扩展JDBC，如果你要对JDBC层有编程的需求，可以通过Druid提供的Filter机制，很方便编写JDBC层的扩展插件。

在项目中使用Druid非常简单，只要修改下配置文件就可以了

下载 druid-0.2.20.jar <http://download.csdn.net/detail/wind520/5670085>

Maven配置

<dependency>  
 <groupId>com.alibaba</groupId>  
 <artifactId>druid</artifactId>  
 <version>1.0.11</version>  
</dependency>

applicationContext.xml 数据源配置

**[html]** [view plaincopy](http://blog.csdn.net/wind520/article/details/9202555)

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<beans** xmlns="http://www.springframework.org/schema/beans"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xmlns:aop="http://www.springframework.org/schema/aop"
5. xmlns:tx="http://www.springframework.org/schema/tx"
6. xsi:schemaLocation="
7. http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-3.2.xsd
8. http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-3.2.xsd
9. http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-3.2.xsd"**>**
11. **<description>**Spring公共配置文件 **</description>**
13. <!-- 定义受环境影响易变的变量 -->
14. **<bean** class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer"**>**
15. **<property** name="systemPropertiesModeName" value="SYSTEM\_PROPERTIES\_MODE\_OVERRIDE" **/>**
16. **<property** name="ignoreResourceNotFound" value="true" **/>**
17. **<property** name="locations"**>**
18. **<list>**
19. <!-- 标准配置 -->
20. **<value>**classpath\*:/application.properties**</value>**
21. **</list>**
22. **</property>**
23. **</bean>**
25. <!-- 数据源配置,使用应用内的c3p0数据库连接池 -->
26. <!--
27. **<bean** id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource" destroy-method="close"**>**
28. **<property** name="driverClass" value="${jdbc.driverClassName}"**/>**
29. **<property** name="jdbcUrl" value="${jdbc.url}"**/>**
30. **<property** name="user" value="${jdbc.username}"**/>**
31. **<property** name="password" value="${jdbc.password}"**/>**
32. **<property** name="initialPoolSize" value="5"**/>**
33. **<property** name="minPoolSize" value="5"**/>**
34. **<property** name="maxPoolSize" value="100"**/>**
35. **<property** name="checkoutTimeout" value="5000"**/>**
36. **<property** name="maxIdleTime" value="1800"**/>**
37. **<property** name="idleConnectionTestPeriod" value="3000"**/>**
38. **<property** name="acquireIncrement" value="3"**/>**
39. **</bean>**
40. --**>**
41. **<bean** id="dataSource" class="com.alibaba.druid.pool.DruidDataSource" init-method="init" destroy-method="close"**>**
42. **<property** name="url" value="${jdbc.url}" **/>**
43. **<property** name="username" value="${jdbc.username}" **/>**
44. **<property** name="password" value="${jdbc.password}" **/>**
45. **<property** name="filters" value="stat" **/>**
46. **<property** name="maxActive" value="20" **/>**
47. **<property** name="initialSize" value="1" **/>**
48. **<property** name="maxWait" value="60000" **/>**
49. **<property** name="minIdle" value="1" **/>**
50. **<property** name="timeBetweenEvictionRunsMillis" value="3000" **/>**
51. **<property** name="minEvictableIdleTimeMillis" value="300000" **/>**
52. **<property** name="validationQuery" value="SELECT 'x'" **/>**
53. **<property** name="testWhileIdle" value="true" **/>**
54. **<property** name="testOnBorrow" value="false" **/>**
55. **<property** name="testOnReturn" value="false" **/>**
56. **<property** name="poolPreparedStatements" value="true" **/>**
57. **<property** name="maxPoolPreparedStatementPerConnectionSize" value="20" **/>**
58. **</bean>**
59. <!-- Hibernate配置 -->
60. **<bean** id="sessionFactory" class="org.springframework.orm.hibernate4.LocalSessionFactoryBean"**>**
61. **<property** name="dataSource" ref="dataSource" **/>**
63. **<property** name="hibernateProperties"**>**
64. **<props>**
65. **<prop** key="hibernate.dialect"**>**${jdbc.dialect}**</prop>**
66. **<prop** key="hibernate.show\_sql"**>**true**</prop>**
67. **<prop** key="hibernate.format\_sql"**>**true**</prop>**
68. **<prop** key="hibernate.cache.region.factory\_class"**>**org.hibernate.cache.ehcache.SingletonEhCacheRegionFactory**</prop>**
69. **<prop** key="hibernate.cache.provider\_configuration\_file\_resource\_path"**>**${hibernate.ehcache\_config\_file}**</prop>**
70. **<prop** key="hibernate.cache.use\_query\_cache"**>**true**</prop>**
71. **<prop** key="hibernate.cache.use\_second\_level\_cache"**>**true**</prop>**
72. **</props>**
73. **</property>**
74. **<property** name="packagesToScan"**>**
75. **<list>**
76. **<value>**com.kingmed.jusmartcare.health.entity**</value>**
77. **</list>**
78. **</property>**
80. **</bean>**
82. <!-- 事务管理器配置,单数据源事务 -->
83. **<bean** id="transactionManager" class="org.springframework.orm.hibernate4.HibernateTransactionManager"**>**
84. **<property** name="sessionFactory" ref="sessionFactory" **/>**
85. **</bean>**
87. <!-- 使用annotation定义事务 -->
88. **<tx:annotation-driven** transaction-manager="transactionManager" proxy-target-class="true" **/>**
90. **</beans>**

web.xml 加上

**[html]** [view plaincopy](http://blog.csdn.net/wind520/article/details/9202555)

1. **<servlet>**
2. **<servlet-name>**DruidStatView**</servlet-name>**
3. **<servlet-class>**com.alibaba.druid.support.http.StatViewServlet**</servlet-class>**
4. **</servlet>**
5. **<servlet-mapping>**
6. **<servlet-name>**DruidStatView**</servlet-name>**
7. **<url-pattern>**/druid/\***</url-pattern>**
8. **</servlet-mapping>**

就可以了.  
监控页面

Url:项目地址根目录+"/druid"

## Shiro

### Shiro架构介绍

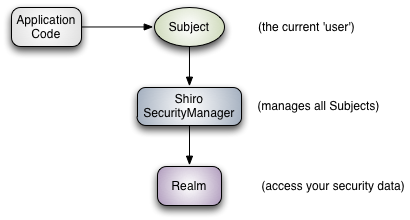
#### 一、什么是Shiro

Apache Shiro是一个强大易用的Java安全框架，提供了认证、授权、加密和会话管理等功能：

* 认证 - 用户身份识别，常被称为用户“登录”；
* 授权 - 访问控制；
* 密码加密 - 保护或隐藏数据防止被偷窥；
* 会话管理 - 每用户相关的时间敏感的状态。

对于任何一个应用程序，Shiro都可以提供全面的安全管理服务。并且相对于其他安全框架，Shiro要简单的多。

#### 二、Shiro的架构介绍

首先，来了解一下Shiro的三个核心组件：Subject, SecurityManager 和 Realms. 如下图：   
   
Subject：即“当前操作用户”。但是，在Shiro中，Subject这一概念并不仅仅指人，也可以是第三方进程、后台帐户（Daemon Account）或其他类似事物。它仅仅意味着“当前跟软件交互的东西”。但考虑到大多数目的和用途，你可以把它认为是Shiro的“用户”概念。   
Subject代表了当前用户的安全操作，SecurityManager则管理所有用户的安全操作。   
  
SecurityManager：它是Shiro框架的核心，典型的Facade模式，Shiro通过SecurityManager来管理内部组件实例，并通过它来提供安全管理的各种服务。   
  
Realm： Realm充当了Shiro与应用安全数据间的“桥梁”或者“连接器”。也就是说，当对用户执行认证（登录）和授权（访问控制）验证时，Shiro会从应用配置的Realm中查找用户及其权限信息。   
从这个意义上讲，Realm实质上是一个安全相关的DAO：它封装了数据源的连接细节，并在需要时将相关数据提供给Shiro。当配置Shiro时，你必须至少指定一个Realm，用于认证和（或）授权。配置多个Realm是可以的，但是至少需要一个。   
Shiro内置了可以连接大量安全数据源（又名目录）的Realm，如LDAP、关系数据库（JDBC）、类似INI的文本配置资源以及属性文件等。如果缺省的Realm不能满足需求，你还可以插入代表自定义数据源的自己的Realm实现。   
  
  
**Shiro完整架构图：**   
  
   
除前文所讲Subject、SecurityManager 、Realm三个核心组件外，Shiro主要组件还包括：   
Authenticator ：认证就是核实用户身份的过程。这个过程的常见例子是大家都熟悉的“用户/密码”组合。多数用户在登录软件系统时，通常提供自己的用户名（当事人）和支持他们的密码（证书）。如果存储在系统中的密码（或密码表示）与用户提供的匹配，他们就被认为通过认证。   
Authorizer ：授权实质上就是访问控制 - 控制用户能够访问应用中的哪些内容，比如资源、Web页面等等。   
SessionManager ：在安全框架领域，Apache Shiro提供了一些独特的东西：可在任何应用或架构层一致地使用Session API。即，Shiro为任何应用提供了一个会话编程范式 - 从小型后台独立应用到大型集群Web应用。这意味着，那些希望使用会话的应用开发者，不必被迫使用Servlet或EJB容器了。或者，如果正在使用这些容器，开发者现在也可以选择使用在任何层统一一致的会话API，取代Servlet或EJB机制。   
CacheManager :对Shiro的其他组件提供缓存支持。

### Shiro 认证

认证就是验证用户身份的过程。在认证过程中，用户需要提交实体信息(Principals)和凭据信息(Credentials)以检验用户是否合法。最常见的“实体/凭证”组合便是“用户名/密码”组合。

#### 一、Shiro认证过程

**1、收集实体/凭据信息**

**Java代码**

1. //Example using most common scenario of username/password pair:
2. UsernamePasswordToken token = **new** UsernamePasswordToken(username, password);
3. //”Remember Me” built-in:
4. token.setRememberMe(**true**);

UsernamePasswordToken支持最常见的用户名/密码的认证机制。同时，由于它实现了RememberMeAuthenticationToken接口，我们可以通过令牌设置“记住我”的功能。   
但是，“已记住”和“已认证”是有区别的：   
已记住的用户仅仅是非匿名用户，你可以通过subject.getPrincipals()获取用户信息。但是它并非是完全认证通过的用户，当你访问需要认证用户的功能时，你仍然需要重新提交认证信息。   
这一区别可以参考亚马逊网站，网站会默认记住登录的用户，再次访问网站时，对于非敏感的页面功能，页面上会显示记住的用户信息，但是当你访问网站账户信息时仍然需要再次进行登录认证。   
  
**2、提交实体/凭据信息**

**Java代码**

1. Subject currentUser = SecurityUtils.getSubject();
2. currentUser.login(token);

收集了实体/凭据信息之后，我们可以通过SecurityUtils工具类，获取当前的用户，然后通过调用login方法提交认证。  
  
**3、认证处理**

**Java代码**

1. **try** {
2. currentUser.login(token);
3. } **catch** ( UnknownAccountException uae ) { ...
4. } **catch** ( IncorrectCredentialsException ice ) { ...
5. } **catch** ( LockedAccountException lae ) { ...
6. } **catch** ( ExcessiveAttemptsException eae ) { ...
7. } ... **catch** your own ...
8. } **catch** ( AuthenticationException ae ) {
9. //unexpected error?
10. }

如果login方法执行完毕且没有抛出任何异常信息，那么便认为用户认证通过。之后在应用程序任意地方调用SecurityUtils.getSubject() 都可以获取到当前认证通过的用户实例，使用subject.isAuthenticated()判断用户是否已验证都将返回true.   
相反，如果login方法执行过程中抛出异常，那么将认为认证失败。Shiro有着丰富的层次鲜明的异常类来描述认证失败的原因，如代码示例。

#### 二、登出操作

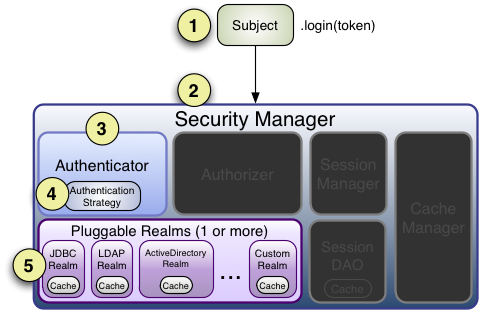
登出操作可以通过调用subject.logout()来删除你的登录信息，如：

**Java代码**

1. currentUser.logout(); //removes all identifying information and invalidates their session too.

当执行完登出操作后，Session信息将被清空，subject将被视作为匿名用户。

#### 三、认证内部处理机制

以上，是Shiro认证在应用程序中的处理过程，下面将详细解说Shiro认证的内部处理机制。   
   
  
如上图，我们通过Shiro架构图的认证部分，来说明Shiro认证内部的处理顺序：   
1、应用程序构建了一个终端用户认证信息的AuthenticationToken 实例后，调用Subject.login方法。   
2、Sbuject的实例通常是DelegatingSubject类（或子类）的实例对象，在认证开始时，会委托应用程序设置的securityManager实例调用securityManager.login(token)方法。   
3、SecurityManager接受到token(令牌)信息后会委托内置的Authenticator的实例（通常都是ModularRealmAuthenticator类的实例）调用authenticator.authenticate(token). ModularRealmAuthenticator在认证过程中会对设置的一个或多个Realm实例进行适配，它实际上为Shiro提供了一个可拔插的认证机制。   
4、如果在应用程序中配置了多个Realm，ModularRealmAuthenticator会根据配置的AuthenticationStrategy(认证策略)来进行多Realm的认证过程。在Realm被调用后，AuthenticationStrategy将对每一个Realm的结果作出响应。   
注：如果应用程序中仅配置了一个Realm，Realm将被直接调用而无需再配置认证策略。   
5、判断每一个Realm是否支持提交的token，如果支持，Realm将调用getAuthenticationInfo(token); getAuthenticationInfo 方法就是实际认证处理，我们通过覆盖Realm的doGetAuthenticationInfo方法来编写我们自定义的认证处理。

#### 四、使用多个Realm的处理机制：

**1、Authenticator**   
默认实现是ModularRealmAuthenticator,它既支持单一Realm也支持多个Realm。如果仅配置了一个Realm，ModularRealmAuthenticator 会直接调用该Realm处理认证信息，如果配置了多个Realm，它会根据认证策略来适配Realm，找到合适的Realm执行认证信息。   
自定义Authenticator的配置：

**Java代码  [收藏代码](javascript:void())**

1. [main]
2. ...
3. authenticator = com.foo.bar.CustomAuthenticator
4. securityManager.authenticator = $authenticator

**2、AuthenticationStrategy（认证策略）**   
当应用程序配置了多个Realm时，ModularRealmAuthenticator将根据认证策略来判断认证成功或是失败。   
例如，如果只有一个Realm验证成功，而其他Realm验证失败，那么这次认证是否成功呢？如果大多数的Realm验证成功了，认证是否就认为成功呢？或者，一个Realm验证成功后，是否还需要判断其他Realm的结果？认证策略就是根据应用程序的需要对这些问题作出决断。   
认证策略是一个无状态的组件，在认证过程中会经过4次的调用：

* 在所有Realm被调用之前
* 在调用Realm的getAuthenticationInfo 方法之前
* 在调用Realm的getAuthenticationInfo 方法之后
* 在所有Realm被调用之后

认证策略的另外一项工作就是聚合所有Realm的结果信息封装至一个AuthenticationInfo实例中，并将此信息返回，以此作为Subject的身份信息。   
Shiro有3中认证策略的具体实现：

|  |  |
| --- | --- |
| AtLeastOneSuccessfulStrategy | 只要有一个（或更多）的Realm验证成功，那么认证将被视为成功 |
| FirstSuccessfulStrategy | 第一个Realm验证成功，整体认证将被视为成功，且后续Realm将被忽略 |
| AllSuccessfulStrategy | 所有Realm成功，认证才视为成功 |

ModularRealmAuthenticator 内置的认证策略默认实现是AtLeastOneSuccessfulStrategy 方式，因为这种方式也是被广泛使用的一种认证策略。当然，你也可以通过配置文件定义你需要的策略，如：

**Java代码**

1. [main]
2. ...
3. authcStrategy = org.apache.shiro.authc.pam.FirstSuccessfulStrategy
4. securityManager.authenticator.authenticationStrategy = $authcStrategy
5. ...

**3、Realm的顺序**   
由刚才提到的认证策略，可以看到Realm在ModularRealmAuthenticator 里面的顺序对认证是有影响的。   
ModularRealmAuthenticator 会读取配置在SecurityManager里的Realm。当执行认证是，它会遍历Realm集合，对所有支持提交的token的Realm调用getAuthenticationInfo 。   
因此，如果Realm的顺序对你使用的认证策略结果有影响，那么你应该在配置文件中明确定义Realm的顺序，如：

**Java代码**

1. blahRealm = com.company.blah.Realm
2. ...
3. fooRealm = com.company.foo.Realm
4. ...
5. barRealm = com.company.another.Realm
7. securityManager.realms = $fooRealm, $barRealm, $blahRealm

### Shiro 授权

授权即访问控制，它将判断用户在应用程序中对资源是否拥有相应的访问权限。   
如，判断一个用户有查看页面的权限，编辑数据的权限，拥有某一按钮的权限，以及是否拥有打印的权限等等。

#### 一、授权的三要素

授权有着三个核心元素：权限、角色和用户。   
  
**权限**   
权限是Apache Shiro安全机制最核心的元素。它在应用程序中明确声明了被允许的行为和表现。一个格式良好好的权限声明可以清晰表达出用户对该资源拥有的权限。   
大多数的资源会支持典型的CRUD操作（create,read,update,delete）,但是任何操作建立在特定的资源上才是有意义的。因此，权限声明的根本思想就是建立在资源以及操作上。   
而我们通过权限声明仅仅能了解这个权限可以在应用程序中做些什么，而不能确定谁拥有此权限。   
于是，我们就需要在应用程序中对用户和权限建立关联。   
通常的做法就是将权限分配给某个角色，然后将这个角色关联一个或多个用户。   
  
**权限声明及粒度**   
Shiro权限声明通常是使用以冒号分隔的表达式。就像前文所讲，一个权限表达式可以清晰的指定资源类型，允许的操作，可访问的数据。同时，Shiro权限表达式支持简单的通配符，可以更加灵活的进行权限设置。   
下面以实例来说明权限表达式。   
可查询用户数据   
User:view   
可查询或编辑用户数据   
User:view,edit   
可对用户数据进行所有操作   
User:\* 或 user   
可编辑id为123的用户数据   
User:edit:123   
  
**角色**   
Shiro支持两种角色模式：   
1、传统角色：一个角色代表着一系列的操作，当需要对某一操作进行授权验证时，只需判断是否是该角色即可。这种角色权限相对简单、模糊，不利于扩展。   
2、权限角色：一个角色拥有一个权限的集合。授权验证时，需要判断当前角色是否拥有该权限。这种角色权限可以对该角色进行详细的权限描述，适合更复杂的权限设计。   
下面将详细描述对两种角色模式的授权实现。

#### 二、授权实现

Shiro支持三种方式实现授权过程：

* 编码实现
* 注解实现
* JSP Taglig实现

**1、基于编码的授权实现**   
  
**1.1基于传统角色授权实现**   
当需要验证用户是否拥有某个角色时，可以调用Subject 实例的hasRole\*方法验证。

**Java代码**

1. Subject currentUser = SecurityUtils.getSubject();
2. **if** (currentUser.hasRole("administrator")) {
3. //show the admin button
4. } **else** {
5. //don't show the button?  Grey it out?
6. }

相关验证方法如下：

|  |  |
| --- | --- |
| Subject方法 | 描述 |
| hasRole(String roleName) | 当用户拥有指定角色时，返回true |
| hasRoles(List<String> roleNames) | 按照列表顺序返回相应的一个boolean值数组 |
| hasAllRoles(Collection<String> roleNames) | 如果用户拥有所有指定角色时，返回true |

**断言支持**   
Shiro还支持以断言的方式进行授权验证。断言成功，不返回任何值，程序继续执行；断言失败时，将抛出异常信息。使用断言，可以使我们的代码更加简洁。

**Java代码**

1. Subject currentUser = SecurityUtils.getSubject();
2. //guarantee that the current user is a bank teller and
3. //therefore allowed to open the account:
4. currentUser.checkRole("bankTeller");
5. openBankAccount();

断言的相关方法：

|  |  |
| --- | --- |
| Subject方法 | 描述 |
| checkRole(String roleName) | 断言用户是否拥有指定角色 |
| checkRoles(Collection<String> roleNames) | 断言用户是否拥有所有指定角色 |
| checkRoles(String... roleNames) | 对上一方法的方法重载 |

**1.2 基于权限角色授权实现**   
相比传统角色模式，基于权限的角色模式耦合性要更低些，它不会因角色的改变而对源代码进行修改，因此，基于权限的角色模式是更好的访问控制方式。   
它的代码实现有以下几种实现方式：   
**1、基于权限对象的实现**   
创建org.apache.shiro.authz.Permission的实例，将该实例对象作为参数传递给Subject.isPermitted（）进行验证。

**Java代码**

1. Permission printPermission = **new** PrinterPermission("laserjet4400n", "print");
2. Subject currentUser = SecurityUtils.getSubject();
3. **if** (currentUser.isPermitted(printPermission)) {
4. //show the Print button
5. } **else** {
6. //don't show the button?  Grey it out?
7. }
8. Permission printPermission = **new** PrinterPermission("laserjet4400n", "print");
9. Subject currentUser = SecurityUtils.getSubject();
10. **if** (currentUser.isPermitted(printPermission)) {
11. //show the Print button
12. } **else** {
13. //don't show the button?  Grey it out?
14. }

相关方法如下：

|  |  |
| --- | --- |
| Subject方法 | 描述 |
| isPermitted(Permission p) | Subject拥有制定权限时，返回treu |
| isPermitted(List<Permission> perms) | 返回对应权限的boolean数组 |
| isPermittedAll(Collection<Permission> perms) | Subject拥有所有制定权限时，返回true |

**2、 基于字符串的实现**   
相比笨重的基于对象的实现方式，基于字符串的实现便显得更加简洁。

**Java代码  [收藏代码](javascript:void())**

1. Subject currentUser = SecurityUtils.getSubject();
2. **if** (currentUser.isPermitted("printer:print:laserjet4400n")) {
3. //show the Print button
4. } **else** {
5. //don't show the button?  Grey it out?
6. }

使用冒号分隔的权限表达式是org.apache.shiro.authz.permission.WildcardPermission 默认支持的实现方式。   
这里分别代表了 资源类型：操作：资源ID   
  
类似基于对象的实现相关方法，基于字符串的实现相关方法：   
isPermitted(String perm)、isPermitted(String... perms)、isPermittedAll(String... perms)   
  
**基于权限对象的断言实现**

**Java代码**

1. Subject currentUser = SecurityUtils.getSubject();
2. //guarantee that the current user is permitted
3. //to open a bank account:
4. Permission p = **new** AccountPermission("open");
5. currentUser.checkPermission(p);
6. openBankAccount();

**基于字符串的断言实现**

**Java代码**

1. Subject currentUser = SecurityUtils.getSubject();
2. //guarantee that the current user is permitted
3. //to open a bank account:
4. currentUser.checkPermission("account:open");
5. openBankAccount();

断言实现的相关方法

|  |  |
| --- | --- |
| Subject方法 | 说明 |
| checkPermission(Permission p) | 断言用户是否拥有制定权限 |
| checkPermission(String perm) | 断言用户是否拥有制定权限 |
| checkPermissions(Collection<Permission> perms) | 断言用户是否拥有所有指定权限 |
| checkPermissions(String... perms) | 断言用户是否拥有所有指定权限 |

**2、基于注解的授权实现**   
Shiro注解支持AspectJ、Spring、Google-Guice等，可根据应用进行不同的配置。   
  
相关的注解：   
**@ RequiresAuthentication**   
可以用户类/属性/方法，用于表明当前用户需是经过认证的用户。

**Java代码  [收藏代码](javascript:void())**

1. @RequiresAuthentication
2. **public** **void** updateAccount(Account userAccount) {
3. //this method will only be invoked by a
4. //Subject that is guaranteed authenticated
5. ...
6. }

**@ RequiresGuest**   
表明该用户需为”guest”用户   
  
**@ RequiresPermissions**   
当前用户需拥有制定权限

**Java代码  [收藏代码](javascript:void())**

1. @RequiresPermissions("account:create")
2. **public** **void** createAccount(Account account) {
3. //this method will only be invoked by a Subject
4. //that is permitted to create an account
5. ...
6. }

**@RequiresRoles**   
当前用户需拥有制定角色   
  
**@ RequiresUser**   
当前用户需为已认证用户或已记住用户   
  
**3、基于JSP  TAG的授权实现**   
Shiro提供了一套JSP标签库来实现页面级的授权控制。   
在使用Shiro标签库前，首先需要在JSP引入shiro标签：

**Java代码**

1. <%@ taglib prefix="shiro" uri="http://shiro.apache.org/tags" %>

下面一一介绍Shiro的标签：   
guest标签   
验证当前用户是否为“访客”，即未认证（包含未记住）的用户

**Xml代码**

1. **<shiro:guest>**
2. Hi there!  Please **<a** href="login.jsp"**>**Login**</a>** or **<a** href="signup.jsp"**>**Signup**</a>** today!
3. **</shiro:guest>**

user标签   
认证通过或已记住的用户

**Xml代码**

1. **<shiro:user>**
2. Welcome back John!  Not John? Click **<a** href="login.jsp"**>**here**<a>** to login.
3. **</shiro:user>**

authenticated标签   
已认证通过的用户。不包含已记住的用户，这是与user标签的区别所在。

**Xml代码**

1. **<shiro:authenticated>**
2. **<a** href="updateAccount.jsp"**>**Update your contact information**</a>**.
3. **</shiro:authenticated>**

notAuthenticated标签   
未认证通过用户，与authenticated标签相对应。与guest标签的区别是，该标签包含已记住用户。

**Xml代码**

1. **<shiro:notAuthenticated>**
2. Please **<a** href="login.jsp"**>**login**</a>** in order to update your credit card information.
3. **</shiro:notAuthenticated>**

principal 标签   
输出当前用户信息，通常为登录帐号信息

**Xml代码**

1. Hello, **<shiro:principal/>**, how are you today?

hasRole标签   
验证当前用户是否属于该角色

**Xml代码**

1. **<shiro:hasRole** name="administrator"**>**
2. **<a** href="admin.jsp"**>**Administer the system**</a>**
3. **</shiro:hasRole>**

lacksRole标签   
与hasRole标签逻辑相反，当用户不属于该角色时验证通过

**Xml代码**

1. **<shiro:lacksRole** name="administrator"**>**
2. Sorry, you are not allowed to administer the system.
3. **</shiro:lacksRole>**

hasAnyRole标签   
验证当前用户是否属于以下任意一个角色。

**Xml代码**

1. **<shiro:hasAnyRoles** name="developer, project manager, administrator"**>**
2. You are either a developer, project manager, or administrator.
3. **</shiro:lacksRole>**

hasPermission标签   
验证当前用户是否拥有制定权限

**Xml代码**

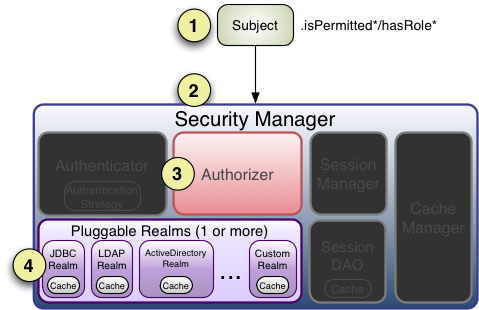
1. **<shiro:hasPermission** name="user:create"**>**
2. **<a** href="createUser.jsp"**>**Create a new User**</a>**
3. **</shiro:hasPermission>**

lacksPermission标签   
与hasPermission标签逻辑相反，当前用户没有制定权限时，验证通过

**Xml代码**

1. **<shiro:hasPermission** name="user:create"**>**
2. **<a** href="createUser.jsp"**>**Create a new User**</a>**
3. **</shiro:hasPermission>**

#### 三、Shiro授权的内部处理机制

   
1、在应用程序中调用授权验证方法(Subject的isPermitted\*或hasRole\*等)   
2、Sbuject的实例通常是DelegatingSubject类（或子类）的实例对象，在认证开始时，会委托应用程序设置的securityManager实例调用相应的isPermitted\*或hasRole\*方法。   
3、接下来SecurityManager会委托内置的Authorizer的实例（默认是ModularRealmAuthorizer 类的实例，类似认证实例，它同样支持一个或多个Realm实例认证）调用相应的授权方法。   
4、每一个Realm将检查是否实现了相同的 Authorizer 接口。然后，将调用Reaml自己的相应的授权验证方法。   
  
当使用多个Realm时，不同于认证策略处理方式，授权处理过程中：   
1、当调用Realm出现异常时，将立即抛出异常，结束授权验证。   
2、只要有一个Realm验证成功，那么将认为授权成功，立即返回，结束认证。

### Realm 实现

在认证、授权内部实现机制中都有提到，最终处理都将交给Real进行处理。因为在Shiro中，最终是通过Realm来获取应用程序中的用户、角色及权限信息的。通常情况下，在Realm中会直接从我们的数据源中获取Shiro需要的验证信息。可以说，Realm是专用于安全框架的DAO.

#### 一、认证实现

正如前文所提到的，Shiro的认证过程最终会交由Realm执行，这时会调用Realm的getAuthenticationInfo(token)方法。   
该方法主要执行以下操作:   
1、检查提交的进行认证的令牌信息   
2、根据令牌信息从数据源(通常为数据库)中获取用户信息   
3、对用户信息进行匹配验证。   
4、验证通过将返回一个封装了用户信息的AuthenticationInfo实例。   
5、验证失败则抛出AuthenticationException异常信息。   
  
而在我们的应用程序中要做的就是自定义一个Realm类，继承AuthorizingRealm抽象类，重载doGetAuthenticationInfo ()，重写获取用户信息的方法。

**Java代码**

1. **protected** AuthenticationInfo doGetAuthenticationInfo(AuthenticationToken authcToken) **throws** AuthenticationException {
2. UsernamePasswordToken token = (UsernamePasswordToken) authcToken;
3. User user = accountManager.findUserByUserName(token.getUsername());
4. **if** (user != **null**) {
5. **return** **new** SimpleAuthenticationInfo(user.getUserName(), user.getPassword(), getName());
6. } **else** {
7. **return** **null**;
8. }
9. }

#### 二、授权实现

而授权实现则与认证实现非常相似，在我们自定义的Realm中，重载doGetAuthorizationInfo()方法，重写获取用户权限的方法即可。

**Java代码**

1. **protected** AuthorizationInfo doGetAuthorizationInfo(PrincipalCollection principals) {
2. String userName = (String) principals.fromRealm(getName()).iterator().next();
3. User user = accountManager.findUserByUserName(userName);
4. **if** (user != **null**) {
5. SimpleAuthorizationInfo info = **new** SimpleAuthorizationInfo();
6. **for** (Group group : user.getGroupList()) {
7. info.addStringPermissions(group.getPermissionList());
8. }
9. **return** info;
10. } **else** {
11. **return** **null**;
12. }
13. }

### Shiro 配置说明

Apache Shiro的配置主要分为四部分：

* 对象和属性的定义与配置
* URL的过滤器配置
* 静态用户配置
* 静态角色配置

其中，由于用户、角色一般由后台进行操作的动态数据，因此Shiro配置一般仅包含前两项的配置。   
  
Apache Shiro的大多数组件是基于POJO的，因此我们可以使用POJO兼容的任何配置机制进行配置，例如：Java代码、Sping XML、YAML、JSON、ini文件等等。下面，以Spring XML的配置方式为例，并且对其中的一些配置参数进行一些简单说明。   
  
**Shiro对象的配置：**   
主要是对Shiro各个组件的实现进行定义配置，主要组件在前文已做过简单介绍，这里不再一一说明。

**Xml代码**

1. **<bean** id="securityManager" class="org.apache.shiro.mgt.DefaultSecurityManager"**>**
2. **<property** name="cacheManager" ref="cacheManager"**/>**
3. **<property** name="sessionMode" value="native"**/>**
4. <!-- Single realm app.  If you have multiple realms, use the 'realms' property instead. -->
5. **<property** name="realm" ref="myRealm"**/>**
6. **<property** name="sessionManager" ref="sessionManager"**/>**
7. **</bean>**

**Shiro过滤器的配置**   
Shiro主要是通过URL过滤来进行安全管理，这里的配置便是指定具体授权规则定义。

**Xml代码**

1. **<bean** id="shiroFilter" class="org.apache.shiro.spring.web.ShiroFilterFactoryBean"**>**
2. **<property** name="securityManager" ref="securityManager"**/>**
3. **<property** name="loginUrl" value="/login.jsp"**/>**
4. **<property** name="successUrl" value="/home.jsp"**/>**
5. **<property** name="unauthorizedUrl" value="/unauthorized.jsp"**/>** --**>**
6. **<property** name="filterChainDefinitions"**>**
7. **<value>**
8. # some example chain definitions:
9. /admin/\*\* = authc, roles[admin]
10. /docs/\*\* = authc, perms[document:read]
11. /\*\* = authc
12. # more URL-to-FilterChain definitions here
13. **</value>**
14. **</property>**
15. **</bean>**

**URL过滤器配置说明：**   
Shiro可以通过配置文件实现基于URL的授权验证。FilterChain定义格式：   
URL\_Ant\_Path\_Expression = Path\_Specific\_Filter\_Chain   
每个URL配置，表示匹配该URL的应用程序请求将由对应的过滤器进行验证。   
例如：   
[urls]   
/index.html = anon   
/user/create = anon   
/user/\*\* = authc   
/admin/\*\* = authc, roles[administrator]   
/rest/\*\* = authc, rest   
/remoting/rpc/\*\* = authc, perms["remote:invoke"]   
  
**URL表达式说明**   
1、URL目录是基于HttpServletRequest.getContextPath()此目录设置   
2、URL可使用通配符，\*\*代表任意子目录   
3、Shiro验证URL时，URL匹配成功便不再继续匹配查找。所以要注意配置文件中的URL顺序，尤其在使用通配符时。   
  
**Filter Chain定义说明**   
1、一个URL可以配置多个Filter，使用逗号分隔   
2、当设置多个过滤器时，全部验证通过，才视为通过   
3、部分过滤器可指定参数，如perms，roles   
  
**Shiro内置的FilterChain**

|  |  |
| --- | --- |
| Filter Name | Class |
| anon | org.apache.shiro.web.filter.authc.AnonymousFilter |
| authc | org.apache.shiro.web.filter.authc.FormAuthenticationFilter |
| authcBasic | org.apache.shiro.web.filter.authc.BasicHttpAuthenticationFilter |
| perms | org.apache.shiro.web.filter.authz.PermissionsAuthorizationFilter |
| port | org.apache.shiro.web.filter.authz.PortFilter |
| rest | org.apache.shiro.web.filter.authz.HttpMethodPermissionFilter |
| roles | org.apache.shiro.web.filter.authz.RolesAuthorizationFilter |
| ssl | org.apache.shiro.web.filter.authz.SslFilter |
| user | org.apache.shiro.web.filter.authc.UserFilter |

## AngularJS

### 1. 关于AngularJS

AngularJS 是 Google 开源出来的一套 js 工具。下面简称其为 ng 。这里只说它是“工具”，没说它是完整的“框架”，是因为它并不是定位于去完成一套框架要做的事。更重要的，是它给我们揭示了一种新的应用组织与开发方式。

ng 最让我称奇的，是它的数据双向绑定。其实想想，我们一直在提数据与表现的分离，但是这里的“双向绑定”从某方面来说，是把数据与表现完全绑定在一起——数据变化，表现也变化。反之，表现变化了，内在的数据也变化。有过开发经验的人能体会到这种机制对于前端应用来说，是很有必要的，能带来维护上的巨大优势。当然，这里的绑定与提倡的分离并不是矛盾的。

ng 可以和 jQuery 集成工作，事实上，如果没有 jQuery ， ng 自己也做了一个轻量级的 jQuery ，主要实现了元素操作部分的 API 。

关于 ng 的几点：

* 对 IE 方面，它兼容 IE8 及以上的版本。
* 与 jQuery 集成工作，它的一些对象与 jQuery 相关对象表现是一致的。
* 使用 ng 时不要冒然去改变相关 DOM 的结构。

### 2. 关于本文档

这份文档如其名，是我自己学习 ng 的过程记录。只是过程记录，没有刻意像教程那样去做。所以呢，从前至后，中间不免有一些概念不清不明的地方。因为事实上，在某个阶段对于一些概念本来就不可能明白。所以，整个过程只求在形式上的能用即可——直到最后的“自定义”那几章，特别是“自定义指令”，那几章过完，你才能看清 ng 本来的面貌。前面就不要太纠结概念，本质，知道怎么用就好。

### 3. 开始的例子

我们从一个完整的例子开始认识 ng ：

1 <!DOCTYPE html>

2 <html>

3 <head>

4 <meta charset="utf-8" />

5

6 <title>试验</title>

7

8 <script type="text/javascript" src="jquery-1.8.3.js"></script>

9 <script type="text/javascript" src="angular.js"></script>

10

11 </head>

12 <body>

13 <div ng-controller="BoxCtrl">

14 <div style="width: 100px; height: 100px; background-color: red;"

15 ng-click="click()"></div>

16 <p>{{ w }} x {{ h }}</p>

17 <p>W: <input type="text" ng-model="w" /></p>

18 <p>H: <input type="text" ng-model="h" /></p>

19 </div>

20

21

22 <script type="text/javascript" charset="utf-8">

23

24

25 var BoxCtrl = function($scope, $element){

26

27 //$element 就是一个 jQuery 对象

28 var e = $element.children().eq(0);

29 $scope.w = e.width();

30 $scope.h = e.height();

31

32 $scope.click = function(){

33 $scope.w = parseInt($scope.w) + 10;

34 $scope.h = parseInt($scope.h) + 10;

35 }

36

37 $scope.$watch('w',

38 function(to, from){

39 e.width(to);

40 }

41 );

42

43 $scope.$watch('h',

44 function(to, from){

45 e.height(to);

46 }

47 );

48 }

49

50 angular.bootstrap(document.documentElement);

51 </script>

52 </body>

53 </html>

从上面的代码中，我们看到在通常的 HTML 代码当中，引入了一些标记，这些就是 ng 的模板机制，它不光完成数据渲染的工作，还实现了数据绑定的功能。

同时，在 HTML 中的本身的 DOM 层级结构，被 ng 利用起来，直接作为它的内部机制中，上下文结构的判断依据。比如例子中 p 是 div 的子节点，那么 p 中的那些模板标记就是在 div 的Ctrl 的作用范围之内。

其它的，也同样写一些 js 代码，里面重要的是作一些数据的操作，事件的绑定定义等。这样，数据的变化就会和页面中的 DOM 表现联系起来。一旦这种联系建立起来，也即完成了我们所说的“双向绑定”。然后，这里说的“事件”，除了那些“点击”等通常的 DOM 事件之外，我们还更关注“数据变化”这个事件。

最后，可以使用：

angular.bootstrap(document.documentElement);

来把整个页面驱动起来了。（你可以看到一个可被控制大小的红色方块）

更完整的方法是定义一个 APP ：

1 <!DOCTYPE html>

2 <html ng-app="MyApp">

3 <head>

4 <meta charset="utf-8" />

5

6 <title>数据正向绑定</title>

7

8 <script type="text/javascript" src="jquery-1.8.3.js"></script>

9 <script type="text/javascript" src="angular.js"></script>

10

11 </head>

12 <body>

13

14 <div ng-controller="TestCtrl">

15 <input type="text" value="" id="a" />

16 </div>

17

18

19 <script type="text/javascript">

20 var TestCtrl = function(){

21 console.log('ok');

22 }

23

24 //angular.bootstrap(document.documentElement);

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

26 </script>

27

28 </body>

29 </html>

这里说的一个 App 就是 ng 概念中的一个 Module 。对于 Controller 来说， 如果不想使用全局函数，也可以在 app 中定义：

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

app.controller('TestCtrl',

function($scope){

console.log('ok');

}

);

上面我们使用 ng-app 来指明要使用的 App ，这样的话可以把显式的初始化工作省了。一般完整的过程是：

var app = angular.module('Demo', [], angular.noop);

angular.bootstrap(document, ['Demo']);

使用 angular.bootstrap 来显示地做初始化工具，参数指明了根节点，装载的模块（可以是多个模块）。

### 4. 依赖注入

injector ， 我从 ng 的文档中得知这个概念，之后去翻看源码时了解了一下这个机制的工作原理。感觉就是虽然与自己的所想仅差那么一点点，但就是这么一点点，让我感慨想象力之神奇。

先看我们之前代码中的一处函数定义：

var BoxCtrl = function($scope, $element){}

在这个函数定义中，注意那两个参数： $scope ， $element ，这是两个很有意思的东西。总的来说，它们是参数，这没什么可说的。但又不仅仅是参数——你换个名字代码就不能正常运行了。

事实上，这两个参数，除了完成“参数”的本身任务之外，还作为一种语法糖完成了“依赖声明”的任务。本来这个函数定义，完整的写法应该像 AMD 声明一样，写成：

var BoxCtrl = ['$scope', '$element', function(s, e){}];

这样就很明显，表示有一个函数，它依赖于两个东西，然后这两个东西会依次作为参数传入。

简单起见，就写成了一个函数定义原本的样子，然后在定义参数的名字上作文章，来起到依赖声明的作用。

在处理时，通过函数对象的 toString() 方法可以知道这个函数定义代码的字符串表现形式，然后就知道它的参数是 $scope 和 $element 。通过名字判断出这是两个外部依赖，然后就去获取资源，最后把资源作为参数，调用定义的函数。

所以，参数的名字是不能随便写的，这里也充分利用了 js 的特点来尽量做到“反省”了。

在 Python 中受限于函数名的命名规则，写出来不太好看。不过也得利于反省机制，做到这点也很容易：

# -\*- coding: utf-8 -\*-

def f(Ia, Ib):

print Ia, Ib

args = f.func\_code.co\_varnames

SRV\_MAP = {

'Ia': '123',

'Ib': '456',

}

srv = {}

for a in args:

if a in SRV\_MAP:

srv[a] = SRV\_MAP[a]

f(\*\*srv)

### 5. 作用域

这里提到的“作用域”的概念，是一个在范围上与 DOM 结构一致，数据上相对于某个 $scope 对象的属性的概念。我们还是从 HTML 代码上来入手：

<div ng-controller="BoxCtrl">

<div style="width: 100px; height: 100px; background-color: red;"

ng-click="click()">

</div>

<p>{{ w }} x {{ h }}</p>

<p>W: <input type="text" ng-model="w" /></p>

<p>H: <input type="text" ng-model="h" /></p>

</div>

上面的代码中，我们给一个 div 元素指定了一个 BoxCtrl ，那么， div 元素之内，就是 BoxCtrl这个函数运行时， $scope 这个注入资源的控制范围。在代码中我们看到的 click() ， w ， h 这些东西，它们本来的位置对应于 $scope.click ， $scope.w ， $scope.h 。

我们在后面的 js 代码中，也可以看到我们就是在操作这些变量。依赖于 ng 的数据绑定机制，操作变量的结果直接在页面上表现出来了。

### 6. 数据绑定与模板

我纠结了半天，“数据绑定”与“模板”这两个东西还真没办法分开来说。因为数据绑定需要以模板为载体，离开了模板，数据还绑个毛啊。

ng 的一大特点，就是数据双向绑定。双向绑定是一体，为了描述方便，下面分别介绍。

#### 6.1. 数据->模板

数据到表现的绑定，主要是使用模板标记直接完成的：

<p>{{ w }} x {{ h }}</p>

使用 {{ }} 这个标记，就可以直接引用，并绑定一个作用域内的变量。在实现上， ng 自动创建了一个 watcher 。效果就是，不管因为什么，如果作用域的变量发生了改变，我们随时可以让相应的页面表现也随之改变。我们可以看一个更纯粹的例子：

<p id="test" ng-controller="TestCtrl">{{ a }}</p>

<script type="text/javascript">

var TestCtrl = function($scope){

$scope.a = '123';

}

angular.bootstrap(document.documentElement);

上面的例子在页面载入之后，我们可以在页面上看到 123 。这时，我们可以打开一个终端控制器，输入：

$('#test').scope().a = '12345';

$('#test').scope().$digest();

上面的代码执行之后，就可以看到页面变化了。

对于使用 ng 进行的事件绑定，在处理函数中就不需要去关心 $digest() 的调用了。因为 ng 会自己处理。源码中，对于 ng 的事件绑定，真正的处理函数不是指定名字的函数，而是经过$apply() 包装过的一个函数。这个 $apply() 做的一件事，就是调用根作用域 $rootScope 的$digest() ，这样整个世界就清净了：

<p id="test" ng-controller="TestCtrl" ng-click="click()">{{ a }}</p>

<script type="text/javascript" charset="utf-8">

var TestCtrl = function($scope){

$scope.a = '123';

$scope.click = function(){

$scope.a = '456';

}

}

angular.bootstrap(document.documentElement);

那个 click 函数的定义，绑定时变成了类似于：

function(){

$scope.$apply(

function(){

$scope.click();

}

)

}

这里的 $scope.$apply() 中做的一件事：

$rootScope.$digest();

#### 6.2. 模板->数据

模板到数据的绑定，主要是通过 ng-model 来完成的：

<input type="text" id="test" ng-controller="TestCtrl" ng-model="a" />

<script type="text/javascript" charset="utf-8">

var TestCtrl = function($scope){

$scope.a = '123';

}

这时修改 input 中的值，然后再在控制终端中使用：

$('#test').scope().a

查看，发现变量 a 的值已经更改了。

实际上， ng-model 是把两个方向的绑定都做了。它不光显示出变量的值，也把显示上的数值变化反映给了变量。这个在实现上就简单多了，只是绑定 change 事件，然后做一些赋值操作即可。不过 ng 里，还要区分对待不同的控件。

#### 6.3. 数据->模板->数据->模板

现在要考虑的是一种在现实中很普遍的一个需求。比如就是我们可以输入数值，来控制一个矩形的长度。在这里，数据与表现的关系是：

* 长度数值保存在变量中
* 变量显示于某个 input 中
* 变量的值即是矩形的长度
* input 中的值变化时，变量也要变化
* input 中的值变化时，矩形的长度也要变化

当然，要实现目的在这里可能就不止一种方案了。按照以前的做法，很自然地会想法，绑定input 的 change 事件，然后去做一些事就好了。但是，我们前面提到过 ng-model 这个东西，利用它就可以在不手工处理 change 的条件下完成数据的展现需求，在此基础之上，我们还需要做的一点，就是把变化后的数据应用到矩形的长度之上。

最开始，我们面对的应该是这样一个东西：

<div ng-controller="TestCtrl">

<div style="width: 100px; height: 10px; background-color: red"></div>

<input type="text" name="width" ng-model="width" />

</div>

<script type="text/javascript" charset="utf-8">

var TestCtrl = function($scope){

$scope.width = 100;

}

angular.bootstrap(document.documentElement);

</script>

我们从响应数据变化，但又不使用 change 事件的角度来看，可以这样处理宽度变化：

var TestCtrl = function($scope, $element){

$scope.width = 100;

$scope.$watch('width',

function(to, from){

$element.children(':first').width(to);

}

);

}

使用 $watch() 来绑定数据变化。

当然，这种样式的问题，有更直接有效的手段， ng 的数据绑定总是让人惊异：

<div ng-controller="TestCtrl">

<div style="width: 10px; height: 10px; background-color: red" ng-style="style">

</div>

<input type="text" name="width" ng-model="style.width" />

</div>

<script type="text/javascript" charset="utf-8">

var TestCtrl = function($scope){

$scope.style = {width: 100 + 'px'};

}

angular.bootstrap(document.documentElement);

</script>

### 7. 模板

前面讲了数据绑定之后，现在可以单独讲讲模板了。

作为一套能称之谓“模板”的系统，除了能干一些模板的常规的事之外（好吧，即使是常规的逻辑判断现在它也做不了的），配合作用域 $scope 和 ng 的数据双向绑定机制， ng 的模板系统就变得比较神奇了。

#### 7.1. 定义模板内容

定义模板的内容现在有三种方式：

1. 在需要的地方直接写字符串
2. 外部文件
3. 使用 script 标签定义的“内部文件”

第一种不需要多说。第二种和第三种都可以和 ng-include 一起工作，来引入一段模板。

直接引入同域的外部文件作为模板的一部分：

<div ng-include src="'tpl.html'">

</div>

<div ng-include="'tpl.html'">

</div>

注意， src 中的字符串会作为表达式处理（可以是 $scope 中的变量），所以，直接写名字的话需要使用引号。

引入 script 定义的“内部文件”：

<script type="text/ng-template" id="tpl">

here, {{ 1 + 1 }}

</script>

<div ng-include src="'tpl'"></div>

配合变量使用：

<script type="text/ng-template" id="tpl">

here, {{ 1 + 1 }}

</script>

<a ng-click="v='tpl'">Load</a>

<div ng-include src="v"></div>

#### 7.2. 内容渲染控制

##### 7.2.1. 重复 ng-repeat

这算是唯一的一个控制标签么……，它的使用方法类型于：

<div ng-controller="TestCtrl">

<ul ng-repeat="member in obj\_list">

<li>{{ member }}</li>

</ul>

</div>

var TestCtrl = function($scope){

$scope.obj\_list = [1,2,3,4];

}

除此之外，它还提供了几个变量可供使用：

* $index 当前索引
* $first 是否为头元素
* $middle 是否为非头非尾元素
* $last 是否为尾元素

<div ng-controller="TestCtrl">

<ul ng-repeat="member in obj\_list">

<li>{{ $index }}, {{ member.name }}</li>

</ul>

</div>

var TestCtrl = function($scope){

$scope.obj\_list = [{name: 'A'}, {name: 'B'}, {name: 'C'}];

}

##### 7.2.2. 赋值 ng-init

这个指令可以在模板中直接赋值，它作用于 angular.bootstrap 之前，并且，定义的变量与$scope 作用域无关。

<div ng-controller="TestCtrl" ng-init="a=[1,2,3,4];">

<ul ng-repeat="member in a">

<li>{{ member }}</li>

</ul>

</div>

#### 7.3. 节点控制

##### 7.3.1. 样式 ng-style

可以使用一个结构直接表示当前节点的样式：

<div ng-style="{width: 100 + 'px', height: 100 + 'px', backgroundColor: 'red'}">

</div>

同样地，绑定一个变量的话，威力大了。

##### 7.3.2. 类 ng-class

就是直接地设置当前节点的类，同样，配合数据绑定作用就大了：

<div ng-controller="TestCtrl" ng-class="cls">

</div>

ng-class-even 和 ng-class-odd 是和 ng-repeat 配合使用的：

<ul ng-init="l=[1,2,3,4]">

<li ng-class-odd="'odd'" ng-class-even="'even'" ng-repeat="m in l">{{ m }}</li>

</ul>

注意里面给的还是表示式，别少了引号。

##### 7.3.3. 显示和隐藏 ng-show ng-hide ng-switch

前两个是控制 display 的指令：

<div ng-show="true">1</div>

<div ng-show="false">2</div>

<div ng-hide="true">3</div>

<div ng-hide="false">4</div>

后一个 ng-switch 是根据一个值来决定哪个节点显示，其它节点移除：

<div ng-init="a=2">

<ul ng-switch on="a">

<li ng-switch-when="1">1</li>

<li ng-switch-when="2">2</li>

<li ng-switch-default>other</li>

</ul>

</div>

##### 7.3.4. 其它属性控制

ng-src 控制 src 属性：

<img ng-src="{{ 'h' + 'ead.png' }}" />

ng-href 控制 href 属性：

<a ng-href="{{ '#' + '123' }}">here</a>

总的来说：

* ng-src src属性
* ng-href href属性
* ng-checked 选中状态
* ng-selected 被选择状态
* ng-disabled 禁用状态
* ng-multiple 多选状态
* ng-readonly 只读状态

注意： 上面的这些只是单向绑定，即只是从数据到展示，不能反作用于数据。要双向绑定，还是要使用 ng-model 。

#### 7.4. 事件绑定

事件绑定是模板指令中很好用的一部分。我们可以把相关事件的处理函数直接写在 DOM 中，这样做的最大好处就是可以从 DOM 结构上看出业务处理的形式，你知道当你点击这个节点时哪个函数被执行了。

* ng-change
* ng-click
* ng-dblclick
* ng-mousedown
* ng-mouseenter
* ng-mouseleave
* ng-mousemove
* ng-mouseover
* ng-mouseup
* ng-submit

对于事件对象本身，在函数调用时可以直接使用 $event 进行传递：

<p ng-click="click($event)">点击</p>

<p ng-click="click($event.target)">点击</p>

#### 7.5. 表单控件

表单控件类的模板指令，最大的作用是它预定义了需要绑定的数据的格式。这样，就可以对于既定的数据进行既定的处理。

##### 7.5.1. form

form 是核心的一个控件。 ng 对 form 这个标签作了包装。事实上， ng 自己的指令是叫 ng-form 的，区别在于， form 标签不能嵌套，而使用 ng-form 指令就可以做嵌套的表单了。

form 的行为中依赖它里面的各个输入控制的状态的，在这里，我们主要关心的是 form 自己的一些方法和属性。从 ng 的角度来说， form 标签，是一个模板指令，也创建了一个FormController 的实例。这个实例就提供了相应的属性和方法。同时，它里面的控件也是一个NgModelController 实例。

很重要的一点， form 的相关方法要生效，必须为 form 标签指定 name 和 ng-controller ，并且每个控件都要绑定一个变量。 form 和控件的名字，即是 $scope 中的相关实例的引用变量名。

<form name="test\_form" ng-controller="TestCtrl">

<input type="text" name="a" required ng-model="a" />

<span ng-click="see()">{{ test\_form.$valid }}</span>

</form>

var TestCtrl = function($scope){

$scope.see = function(){

console.log($scope.test\_form);

console.log($scope.test\_form.a);

}

}

除去对象的方法与属性， form 这个标签本身有一些动态类可以使用：

* ng-valid 当表单验证通过时的设置
* ng-invalid 当表单验证失败时的设置
* ng-pristine 表单的未被动之前拥有
* ng-dirty 表单被动过之后拥有

form 对象的属性有：

* $pristine 表单是否未被动过
* $dirty 表单是否被动过
* $valid 表单是否验证通过
* $invalid 表单是否验证失败
* $error 表单的验证错误

其中的 $error 对象包含有所有字段的验证信息，及对相关字段的 NgModelController 实例的引用。它的结构是一个对象， key 是失败信息， required ， minlength 之类的， value 是对应的字段实例列表。

注意，这里的失败信息是按序列取的一个。比如，如果一个字段既要求 required ，也要求minlength ，那么当它为空时， $error 中只有 required 的失败信息。只输入一个字符之后，required 条件满足了，才可能有 minlength 这个失败信息。

<form name="test\_form" ng-controller="TestCtrl">

<input type="text" name="a" required ng-model="a" />

<input type="text" name="b" required ng-model="b" ng-minlength="2" />

<span ng-click="see()">{{ test\_form.$error }}</span>

</form>

var TestCtrl = function($scope){

$scope.see = function(){

console.log($scope.test\_form.$error);

}

}

##### 7.5.2. input

input 是数据的最主要入口。 ng 支持 HTML5 中的相关属性，同时对旧浏览器也做了兼容性处理。最重要的， input 的规则定义，是所属表单的相关行为的参照（比如表单是否验证成功）。

input 控件的相关可用属性为：

* name 名字
* ng-model 绑定的数据
* required 是否必填
* ng-required 是否必填
* ng-minlength 最小长度
* ng-maxlength 最大长度
* ng-pattern 匹配模式
* ng-change 值变化时的回调

<form name="test\_form" ng-controller="TestCtrl">

<input type="text" name="a" ng-model="a" required ng-pattern="/abc/" />

<span ng-click="see()">{{ test\_form.$error }}</span>

</form>

input 控件，它还有一些扩展，这些扩展有些有自己的属性：

* input type="number" 多了 number 错误类型，多了 max ， min 属性。
* input type="url" 多了 url 错误类型。
* input type="email" 多了 email 错误类型。

##### 7.5.3. checkbox

它也算是 input 的扩展，不过，它没有验证相关的东西，只有选中与不选中两个值：

<form name="test\_form" ng-controller="TestCtrl">

<input type="checkbox" name="a" ng-model="a" ng-true-value="AA" ng-false-value="BB" />

<span>{{ a }}</span>

</form>

var TestCtrl = function($scope){

$scope.a = 'AA';

}

两点：

1. controller 要初始化变量值。
2. controller 中的初始化值会关系到控件状态（双向绑定）。

##### 7.5.4. radio

也是 input 的扩展。和 checkbox 一样，但它只有一个值了：

<form name="test\_form" ng-controller="TestCtrl">

<input type="radio" name="a" ng-model="a" value="AA" />

<input type="radio" name="a" ng-model="a" value="BB" />

<span>{{ a }}</span>

</form>

##### 7.5.5. textarea

同 input 。

##### 7.5.6. select

这是一个比较牛B的控件。它里面的一个叫做 ng-options 的属性用于数据呈现。

对于给定列表时的使用。

最简单的使用方法， x for x in list ：

<form name="test\_form" ng-controller="TestCtrl" ng-init="o=[0,1,2,3]; a=o[1];">

<select ng-model="a" ng-options="x for x in o" ng-change="show()">

<option value="">可以加这个空值</option>

</select>

</form>

<script type="text/javascript">

var TestCtrl = function($scope){

$scope.show = function(){

console.log($scope.a);

}

}

angular.bootstrap(document.documentElement);

</script>

在 $scope 中， select 绑定的变量，其值和普通的 value 无关，可以是一个对象：

<form name="test\_form" ng-controller="TestCtrl"

ng-init="o=[{name: 'AA'}, {name: 'BB'}]; a=o[1];">

<select ng-model="a" ng-options="x.name for x in o" ng-change="show()">

</select>

</form>

显示与值分别指定， x.v as x.name for x in o ：

<form name="test\_form" ng-controller="TestCtrl"

ng-init="o=[{name: 'AA', v: '00'}, {name: 'BB', v: '11'}]; a=o[1].v;">

<select ng-model="a" ng-options="x.v as x.name for x in o" ng-change="show()">

</select>

</form>

加入分组的， x.name group by x.g for x in o ：

<form name="test\_form" ng-controller="TestCtrl"

ng-init="o=[{name: 'AA', g: '00'}, {name: 'BB', g: '11'}, {name: 'CC', g: '00'}]; a=o[1];">

<select ng-model="a" ng-options="x.name group by x.g for x in o" ng-change="show()">

</select>

</form>

分组了还分别指定显示与值的， x.v as x.name group by x.g for x in o ：

<form name="test\_form" ng-controller="TestCtrl" ng-init="o=[{name: 'AA', g: '00', v: '='}, {name: 'BB', g: '11', v: '+'}, {name: 'CC', g: '00', v: '!'}]; a=o[1].v;">

<select ng-model="a" ng-options="x.v as x.name group by x.g for x in o" ng-change="show()">

</select>

</form>

如果参数是对象的话，基本也是一样的，只是把遍历的对象改成 (key, value) ：

<form name="test\_form" ng-controller="TestCtrl" ng-init="o={a: 0, b: 1}; a=o.a;">

<select ng-model="a" ng-options="k for (k, v) in o" ng-change="show()">

</select>

</form>

<form name="test\_form" ng-controller="TestCtrl"

ng-init="o={a: {name: 'AA', v: '00'}, b: {name: 'BB', v: '11'}}; a=o.a.v;">

<select ng-model="a" ng-options="v.v as v.name for (k, v) in o" ng-change="show()">

</select>

</form>

<form name="test\_form" ng-controller="TestCtrl"

ng-init="o={a: {name: 'AA', v: '00', g: '=='}, b: {name: 'BB', v: '11', g: '=='}}; a=o.a;">

<select ng-model="a" ng-options="v.name group by v.g for (k, v) in o" ng-change="show()">

</select>

</form>

<form name="test\_form" ng-controller="TestCtrl"

ng-init="o={a: {name: 'AA', v: '00', g: '=='}, b: {name: 'BB', v: '11', g: '=='}}; a=o.a.v;">

<select ng-model="a" ng-options="v.v as v.name group by v.g for (k, v) in o" ng-change="show()">

</select>

</form>

### 8. 模板中的过滤器

这里说的过滤器，是用于对数据的格式化，或者筛选的函数。它们可以直接在模板中通过一种语法使用。对于常用功能来说，是很方便的一种机制。

多个过滤器之间可以直接连续使用。

#### 8.1. 排序 orderBy

orderBy 是一个排序用的过滤器标签。它可以像 sort 函数那样支持一个排序函数，也可以简单地指定一个属性名进行操作：

<div ng-controller="TestCtrl">

{{ data | orderBy: 'age' }} <br />

{{ data | orderBy: '-age' }} <br />

{{ data | orderBy: '-age' | limitTo: 2 }} <br />

{{ data | orderBy: ['-age', 'name'] }} <br />

</div>

<script type="text/javascript">

var TestCtrl = function($scope){

$scope.data = [

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

{name: 'A', age: 1},

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

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

];

}

angular.bootstrap(document.documentElement);

</script>

#### 8.2. 过滤列表 filter

filter 是一个过滤内容的标签。

如果参数是一个字符串，则列表成员中的任意属性值中有这个字符串，即为满足条件（忽略大小写）：

<div ng-controller="TestCtrl">

{{ data | filter: 'b' }} <br />

{{ data | filter: '!B' }} <br />

</div>

<script type="text/javascript">

var TestCtrl = function($scope){

$scope.data = [

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

{name: 'A', age: 1},

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

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

];

}

angular.bootstrap(document.documentElement);

</script>

可以使用对象，来指定属性名， $ 表示任意属性：

{{ data | filter: {name: 'A'} }} <br />

{{ data | filter: {$: '3'} }} <br />

{{ data | filter: {$: '!3'} }} <br />

自定义的过滤函数也支持：

<div ng-controller="TestCtrl">

{{ data | filter: f }} <br />

</div>

<script type="text/javascript">

var TestCtrl = function($scope){

$scope.data = [

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

{name: 'A', age: 1},

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

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

];

$scope.f = function(e){

return e.age > 2;

}

}

angular.bootstrap(document.documentElement);

</script>

#### 8.3. 其它

时间戳格式化 date ：

<div ng-controller="TestCtrl">

{{ a | date: 'yyyy-MM-dd HH:mm:ss' }}

</div>

<script type="text/javascript">

var TestCtrl = function($scope){

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

}

angular.bootstrap(document.documentElement);

</script>

列表截取 limitTo ，支持正负数：

{{ [1,2,3,4,5] | limitTo: 2 }}

{{ [1,2,3,4,5] | limitTo: -3 }}

大小写 lowercase ， uppercase ：

{{ 'abc' | uppercase }}

{{ 'Abc' | lowercase }}

#### 8.4. 例子：表头排序

1 <div ng-controller="TestCtrl">

2 <table>

3 <tr>

4 <th ng-click="f='name'; rev=!rev">名字</th>

5 <th ng-click="f='age'; rev=!rev">年龄</th>

6 </tr>

7

8 <tr ng-repeat="o in data | orderBy: f : rev">

9 <td>{{ o.name }}</td>

10 <td>{{ o.age }}</td>

11 </tr>

12 </table>

13 </div>

14

15 <script type="text/javascript">

16 var TestCtrl = function($scope){

17 $scope.data = [

18 {name: 'B', age: 4},

19 {name: 'A', age: 1},

20 {name: 'D', age: 3},

21 {name: 'C', age: 3},

22 ];

23 }

24

25 angular.bootstrap(document.documentElement);

26 </script>

#### 8.5. 例子：搜索

<div ng-controller="TestCtrl" ng-init="s=data[0].name; q=''">

<div>

<span>查找：</span> <input type="text" ng-model="q" />

</div>

<select ng-multiple="true" ng-model="s"

ng-options="o.name as o.name + '(' + o.age + ')' for o in data | filter: {name: q} | orderBy: ['age', 'name'] ">

</select>

</div>

<script type="text/javascript">

var TestCtrl = function($scope){

$scope.data = [

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

{name: 'A', age: 1},

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

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

];

}

angular.bootstrap(document.documentElement);

</script>

### 9. 锚点路由

准确地说，这应该叫对 hashchange 事件的处理吧。

就是指 URL 中的锚点部分发生变化时，触发预先定义的业务逻辑。比如现在是 /test#/x ，锚点部分的值为 # 后的 /x ，它就对应了一组处理逻辑。当这部分变化时，比如变成了/test#/t ，这时页面是不会刷新的，但是它可以触发另外一组处理逻辑，来做一些事，也可以让页面发生变化。

这种机制对于复杂的单页面来说，无疑是一种强大的业务切分手段。就算不是复杂的单页面应用，在普通页面上善用这种机制，也可以让业务逻辑更容易控制。

ng 提供了完善的锚点路由功能，虽然目前我觉得相当重要的一个功能还有待完善（后面会说），但目前这功能的几部分内容，已经让我思考了很多种可能性了。

ng 中的锚点路由功能是由几部分 API 共同完成的一整套方案。这其中包括了路由定义，参数定义，业务处理等。

#### 9.1. 路由定义

要使用锚点路由功能，需要在先定义它。目前，对于定义的方法，我个人只发现在“初始化”阶段可以通过 $routeProvider 这个服务来定义。

在定义一个 app 时可以定义锚点路由：

<html ng-app="ngView">

... ...

<div ng-view></div>

<script type="text/javascript">

angular.module('ngView', [],

function($routeProvider){

$routeProvider.when('/test',

{

template: 'test',

}

);

}

);

</script>

首先看 ng-view 这个 directive ，它是一个标记“锚点作用区”的指令。目前页面上只能有一个“锚点作用区”。有人已经提了，“多个可命名”的锚点作用区的代码到官方，但是目前官方还没有接受合并，我觉得多个作用区这个功能是很重要的，希望下个发布版中能有。

锚点作用区的功能，就是让锚点路由定义时的那些模板， controller 等，它们产生的 HTML 代码放在作用区内。

比如上面的代码，当你刚打开页面时，页面是空白的。你手动访问 /#/test 就可以看到页面上出现了 'test' 的字样。

在 angular.bootstrap() 时也可以定义：

angular.bootstrap(document.documentElement, [

function($routeProvider){

$routeProvider.when('/test',

{

template: 'test'

}

);

}

]);

#### 9.2. 参数定义

在作路由定义时，可以匹配一个规则，规则中可以定义路径中的某些部分作为参数之用，然后使用 $routeParams 服务获取到指定参数。比如 /#/book/test 中， test 作为参数传入到 controller 中：

<div ng-view></div>

<script type="text/javascript">

angular.module('ngView', [],

function($routeProvider){

$routeProvider.when('/book/:title',

{

template: '{{ title }}',

controller: function($scope, $routeParams){

$scope.title = $routeParams.title;

}

}

);

}

);

</script>

访问： /#/book/test

不需要预定义模式，也可以像普通 GET 请求那样获取到相关参数：

angular.module('ngView', [],

function($routeProvider){

$routeProvider.when('/book',

{

template: '{{ title }}',

controller: function($scope, $routeParams){

$scope.title = $routeParams.title;

}

}

);

}

);

访问： /#/book?title=test

#### 9.3. 业务处理

简单来说，当一个锚点路由定义被匹配时，会根据模板生成一个 $scope ，同时相应的一个 controller 就会被触发。最后模板的结果会被填充到 ng-view 中去。

从上面的例子中可以看到，最直接的方式，我们可以在模板中双向绑定数据，而数据的来源，在 controller 中控制。在 controller 中，又可以使用到像 $scope ， $routeParams 这些服务。

这里先提一下另外一种与锚点路由相关的服务， $route 。这个服务里锚点路由在定义时，及匹配过程中的信息。比如我们搞怪一下：

angular.module('ngView', [],

function($routeProvider){

$routeProvider.when('/a',

{

template: '{{ title }}',

controller: function($scope){

$scope.title = 'a';

}

}

);

$routeProvider.when('/b',

{

template: '{{ title }}',

controller: function($scope, $route){

console.log($route);

$route.routes['/a'].controller($scope);

}

}

);

}

);

回到锚点定义的业务处理中来。我们可以以字符串形式写模板，也可以直接引用外部文件作为模板：

angular.module('ngView', [],

function($routeProvider){

$routeProvider.when('/test',

{

templateUrl: 'tpl.html',

controller: function($scope){

$scope.title = 'a';

}

}

);

}

);

tpl.html 中的内容是：

{{ title }}

这样的话，模板可以预定义，也可以很复杂了。

现在暂时忘了模板吧，因为前面提到的，当前 ng-view 不能有多个的限制，模板的渲染机制局限性还是很大的。不过，反正会触发一个 controller ，那么在函数当中我们可以尽量地干自己喜欢的事：

angular.module('ngView', [],

function($routeProvider){

$routeProvider.when('/test',

{

template: '{{}}',

controller: function(){

$('div').first().html('<b>OK</b>');

}

}

);

}

);

那个空的 template 不能省，否则 controller 不会被触发。

### 10. 定义模板变量标识标签

由于下面涉及动态内容，所以我打算起一个后端服务来做。但是我发现我使用的 Tornado 框架的模板系统，与 ng 的模板系统，都是使用 {{ }} 这对符号来定义模板表达式的，这太悲剧了，不过幸好 ng 已经提供了修改方法：

angular.bootstrap(document.documentElement,

[function($interpolateProvider){

$interpolateProvider.startSymbol('[[');

$interpolateProvider.endSymbol(']]');

}]);

使用 $interpolateProvider 服务即可。

### 11. AJAX

ng 提供了基本的 AJAX 封装，你直接面对 promise 对象，使用起来还是很方便的。

#### 11.1. HTTP请求

基本的操作由 $http 服务提供。它的使用很简单，提供一些描述请求的参数，请求就出去了，然后返回一个扩充了 success 方法和 error 方法的 promise 对象（下节介绍），你可以在这个对象中添加需要的回调函数。

var TestCtrl = function($scope, $http){

var p = $http({

method: 'GET',

url: '/json'

});

p.success(function(response, status, headers, config){

$scope.name = response.name;

});

}

$http 接受的配置项有：

* method 方法
* url 路径
* params GET请求的参数
* data post请求的参数
* headers 头
* transformRequest 请求预处理函数
* transformResponse 响应预处理函数
* cache 缓存
* timeout 超时毫秒，超时的请求会被取消
* withCredentials 跨域安全策略的一个东西

其中的 transformRequest 和 transformResponse 及 headers 已经有定义的，如果自定义则会覆盖默认定义：

1 var $config = this.defaults = {

2 // transform incoming response data

3 transformResponse: [function(data) {

4 if (isString(data)) {

5 // strip json vulnerability protection prefix

6 data = data.replace(PROTECTION\_PREFIX, '');

7 if (JSON\_START.test(data) && JSON\_END.test(data))

8 data = fromJson(data, true);

9 }

10 return data;

11 }],

12

13 // transform outgoing request data

14 transformRequest: [function(d) {

15 return isObject(d) && !isFile(d) ? toJson(d) : d;

16 }],

17

18 // default headers

19 headers: {

20 common: {

21 'Accept': 'application/json, text/plain, \*/\*',

22 'X-Requested-With': 'XMLHttpRequest'

23 },

24 post: {'Content-Type': 'application/json;charset=utf-8'},

25 put: {'Content-Type': 'application/json;charset=utf-8'}

26 }

27 };

注意它默认的 POST 方法出去的 Content-Type

对于几个标准的 HTTP 方法，有对应的 shortcut ：

* $http.delete(url, config)
* $http.get(url, config)
* $http.head(url, config)
* $http.jsonp(url, config)
* $http.post(url, data, config)
* $http.put(url, data, config)

注意其中的 JSONP 方法，在实现上会在页面中添加一个 script 标签，然后放出一个 GET 请求。你自己定义的，匿名回调函数，会被 ng 自已给一个全局变量。在定义请求，作为 GET 参数，你可以使用 JSON\_CALLBACK 这个字符串来暂时代替回调函数名，之后 ng 会为你替换成真正的函数名：

var p = $http({

method: 'JSONP',

url: '/json',

params: {callback: 'JSON\_CALLBACK'}

});

p.success(function(response, status, headers, config){

console.log(response);

$scope.name = response.name;

});

$http 有两个属性：

* defaults 请求的全局配置
* pendingRequests 当前的请求队列状态

$http.defaults.transformRequest = function(data){console.log('here'); return data;}

console.log($http.pendingRequests);

#### 11.2. 广义回调管理

和其它框架一样， ng 提供了广义的异步回调管理的机制。 $http 服务是在其之上封装出来的。这个机制就是 ng 的 $q 服务。

不过 ng 的这套机制总的来说实现得比较简单，按官方的说法，够用了。

使用的方法，基本上是：

* 通过 $q 服务得到一个 deferred 实例
* 通过 deferred 实例的 promise 属性得到一个 promise 对象
* promise 对象负责定义回调函数
* deferred 实例负责触发回调

var TestCtrl = function($q){

var defer = $q.defer();

var promise = defer.promise;

promise.then(function(data){console.log('ok, ' + data)},

function(data){console.log('error, ' + data)});

//defer.reject('xx');

defer.resolve('xx');

}

了解了上面的东西，再分别看 $q ， deferred ， promise 这三个东西。

##### 11.2.1. $q

$q 有四个方法：

* $q.all() 合并多个 promise ，得到一个新的 promise
* $q.defer() 返回一个 deferred 对象
* $q.reject() 包装一个错误，以使回调链能正确处理下去
* $q.when() 返回一个 promise 对象

$q.all() 方法适用于并发场景很合适：

var TestCtrl = function($q, $http){

var p = $http.get('/json', {params: {a: 1}});

var p2 = $http.get('/json', {params: {a: 2}});

var all = $q.all([p, p2]);

p.success(function(res){console.log('here')});

all.then(function(res){console.log(res[0])});

}

$q.reject() 方法是在你捕捉异常之后，又要把这个异常在回调链中传下去时使用：

要理解这东西，先看看 promise 的链式回调是如何运作的，看下面两段代码的区别：

var defer = $q.defer();

var p = defer.promise;

p.then(

function(data){return 'xxx'}

);

p.then(

function(data){console.log(data)}

);

defer.resolve('123');

var defer = $q.defer();

var p = defer.promise;

var p2 = p.then(

function(data){return 'xxx'}

);

p2.then(

function(data){console.log(data)}

);

defer.resolve('123');

从模型上看，前者是“并发”，后者才是“链式”。

而 $q.reject() 的作用就是触发后链的 error 回调：

var defer = $q.defer();

var p = defer.promise;

p.then(

function(data){return data},

function(data){return $q.reject(data)}

).

then(

function(data){console.log('ok, ' + data)},

function(data){console.log('error, ' + data)}

)

defer.reject('123');

最后的 $q.when() 是把数据封装成 promise 对象：

var p = $q.when(0, function(data){return data},

function(data){return data});

p.then(

function(data){console.log('ok, ' + data)},

function(data){console.log('error, ' + data)}

);

##### 11.2.2. deferred

deferred 对象有两个方法一个属性。

* promise 属性就是返回一个 promise 对象的。
* resolve() 成功回调
* reject() 失败回调

var defer = $q.defer();

var promise = defer.promise;

promise.then(function(data){console.log('ok, ' + data)},

function(data){console.log('error, ' + data)});

//defer.reject('xx');

defer.resolve('xx');

##### 11.2.3. promise

promise 对象只有 then() 一个方法，注册成功回调函数和失败回调函数，再返回一个 promise对象，以用于链式调用。

### 12. 工具函数

#### 12.1. 上下文绑定

angular.bind 是用来进行上下文绑定，参数动态绑定的工具函数。

var f = angular.bind({a: 'xx'},

function(){

console.log(this.a);

}

);

f();

参数动态绑定：

var f = function(x){console.log(x)}

angular.bind({}, f, 'x')();

#### 12.2. 对象处理

对象复制： angular.copy()

var a = {'x': '123'};

var b = angular.copy(a);

a.x = '456';

console.log(b);

对象聚合： angular.extend()

var a = {'x': '123'};

var b = {'xx': '456'};

angular.extend(b, a);

console.log(b);

空函数： angular.noop()

大小写转换： angular.lowercase() 和 angular.uppercase()

JSON转换： angular.fromJson() 和 angular.toJson()

遍历： angular.forEach() ，支持列表和对象：

var l = {a: '1', b: '2'};

angular.forEach(l, function(v, k){console.log(k + ': ' + v)});

var l = ['a', 'b', 'c'];

angular.forEach(l, function(v, i, o){console.log(v)});

var context = {'t': 'xx'};

angular.forEach(l, function(v, i, o){console.log(this.t)}, context);

#### 12.3. 类型判定

* angular.isArray
* angular.isDate
* angular.isDefined
* angular.isElement
* angular.isFunction
* angular.isNumber
* angular.isObject
* angular.isString
* angular.isUndefined

### 13. 其它服务

#### 13.1. 日志

ng 提供 $log 这个服务用于向终端输出相关信息：

* error()
* info()
* log()
* warn()

var TestCtrl = function($log){

$log.error('error');

$log.info('info');

$log.log('log');

$log.warn('warn');

}

#### 13.2. 缓存

ng 提供了一个简单封装了缓存机制 $cacheFactory ，可以用来作为数据容器：

var TestCtrl = function($scope, $cacheFactory){

$scope.cache = $cacheFactory('s\_' + $scope.$id, {capacity: 3});

$scope.show = function(){

console.log($scope.cache.get('a'));

console.log($scope.cache.info());

}

$scope.set = function(){

$scope.cache.put((new Date()).valueOf(), 'ok');

}

}

调用时，第一个参数是 id ，第二个参数是配置项，目前支持 capacity 参数，用以设置缓存能容留的最大条目数。超过这个个数，则自动清除较旧的条目。

缓存实例的方法：

* info() 获取 id , size 信息
* put(k, v) 设置新条目
* get(k) 获取条目
* remove(k) 删除条目
* removeAll() 删除所有条目
* destroy() 删除对本实例的引用

$http 的调用当中，有一个 cache 参数，值为 true 时为自动维护的缓存。值也可以设置为一个 cache 实例。

#### 13.3. 计时器

$timeout 服务是 ng 对 window.setTimeout() 的封装，它使用 promise 统一了计时器的回调行为：

var TestCtrl = function($timeout){

var p = $timeout(function(){console.log('haha')}, 5000);

p.then(function(){console.log('x')});

//$timeout.cancel(p);

}

使用 $timeout.cancel() 可以取消计时器。

#### 13.4. 表达式函数化

$parse 这个服务，为 js 提供了类似于 Python 中 @property 的能力：

var TestCtrl = function($scope, $parse){

$scope.get\_name = $parse('name');

$scope.show = function(){console.log($scope.get\_name($scope))}

$scope.set = function(){$scope.name = '123'}

}

$parse 返回一个函数，调用这个函数时，可以传两个参数，第一个作用域，第二个是变量集，后者常用于覆盖前者的变量：

var get\_name = $parse('name');

var r = get\_name({name: 'xx'}, {name: 'abc'});

console.log(r);

$parse 返回的函数，也提供了相应的 assign 功能，可以为表达式赋值（如果可以的话）：

var get\_name = $parse('name');

var set\_name = get\_name.assign;

var r = get\_name({name: 'xx'}, {name: 'abc'});

console.log(r);

var s = {}

set\_name(s, '123');

var r = get\_name(s);

console.log(r);

#### 13.5. 模板单独使用

ng 中的模板是很重要，也很强大的一个机制，自然少不了单独运用它的方法。不过，即使是单独使用，也是和 DOM 紧密相关的程度：

* 定义时必须是有 HTML 标签包裹的，这样才能创建 DOM 节点
* 渲染时必须传入 $scope

之后使用 $compile 就可以得到一个渲染好的节点对象了。当然， $compile 还要做其它一些工作，指令处理什么的。

var TestCtrl = function($scope, $element,$compile){

$scope.a = '123';

$scope.set = function(){

var tpl = $compile('<p>hello {{ a }}</p>');

var e = tpl($scope);

$element.append(e);

}

}

### 14. 自定义模块和服务

#### 14.1. 模块和服务的概念与关系

总的来说，模块是组织业务的一个框框，在一个模块当中定义多个服务。当你引入了一个模块的时候，就可以使用这个模块提供的一种或多种服务了。

比如 AngularJS 本身的一个默认模块叫做 ng ，它提供了 $http ， $q 等等服务。

服务只是模块提供的多种机制中的一种，其它的还有命令（ directive ），过滤器（ filter ），及其它配置信息。

然后在额外的 js 文件中有一个附加的模块叫做 ngResource ， 它提供了一个 $resource 服务。

定义时，我们可以在已有的模块中新定义一个服务，也可以先新定义一个模块，然后在新模块中定义新服务。

使用时，模块是需要显式地的声明依赖（引入）关系的，而服务则可以让 ng 自动地做注入，然后直接使用。

#### 14.2. 定义模块

定义模块的方法是使用 angular.module 。调用时声明了对其它模块的依赖，并定义了“初始化”函数。

var my\_module = angular.module('MyModule', [], function(){

console.log('here');

});

这段代码定义了一个叫做 MyModule 的模块， my\_module 这个引用可以在接下来做其它的一些事，比如定义服务。

#### 14.3. 定义服务

服务本身是一个任意的对象。但是 ng 提供服务的过程涉及它的依赖注入机制。在这里呢，就要先介绍一下叫 provider 的东西。

简单来说， provider 是被“注入控制器”使用的一个对象，注入机制通过调用一个 provider 的$get() 方法，把得到的东西作为参数进行相关调用（比如把得到的服务作为一个 Controller 的参数）。

在这里“服务”的概念就比较不明确，对使用而言，服务仅指 $get() 方法返回的东西，但是在整体机制上，服务又要指提供了 $get() 方法的整个对象。

//这是一个provider

var pp = function(){

this.$get = function(){

return {'haha': '123'};

}

}

//我在模块的初始化过程当中, 定义了一个叫 PP 的服务

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

$provide.provider('PP', pp);

});

//PP服务实际上就是 pp 这个 provider 的 $get() 方法返回的东西

app.controller('TestCtrl',

function($scope, PP){

console.log(PP);

}

);

上面的代码是一种定义服务的方法，当然， ng 还有相关的 shortcut， ng 总有很多 shortcut 。

第一个是 factory 方法，由 $provide 提供， module 的 factory 是一个引用，作用一样。这个方法直接把一个函数当成是一个对象的 $get() 方法，这样你就不用显式地定义一个 provider 了：

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

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

return {'hello': '123'};

});

});

app.controller('TestCtrl', function($scope, PP){ console.log(PP) });

在 module 中使用：

var app = angular.module('Demo', [], function(){ });

app.factory('PP', function(){return {'abc': '123'}});

app.controller('TestCtrl', function($scope, PP){ console.log(PP) });

第二个是 service 方法，也是由 $provide 提供， module 中有对它的同名引用。 service 和factory 的区别在于，前者是要求提供一个“构造方法”，后者是要求提供 $get() 方法。意思就是，前者一定是得到一个 object ，后者可以是一个数字或字符串。它们的关系大概是：

var app = angular.module('Demo', [], function(){ });

app.service = function(name, constructor){

app.factory(name, function(){

return (new constructor());

});

}

这里插一句，js 中 new 的作用，以 new a() 为例，过程相当于：

1. 创建一个空对象 obj
2. 把 obj 绑定到 a 函数的上下文当中（即 a 中的 this 现在指向 obj ）
3. 执行 a 函数
4. 返回 obj

service 方法的使用就很简单了：

var app = angular.module('Demo', [], function(){ });

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

this.abc = '123';

});

app.controller('TestCtrl', function($scope, PP){ console.log(PP) });

#### 14.4. 引入模块并使用服务

结合上面的“定义模块”和“定义服务”，我们可以方便地组织自己的额外代码：

angular.module('MyModule', [], function($provide){

$provide.factory('S1', function(){

return 'I am S1';

});

$provide.factory('S2', function(){

return {see: function(){return 'I am S2'}}

});

});

var app = angular.module('Demo', ['MyModule'], angular.noop);

app.controller('TestCtrl', function($scope, S1, S2){

console.log(S1)

console.log(S2.see())

});

### 15. 附加模块 ngResource

#### 15.1. 使用引入与整体概念

ngResource 这个是 ng 官方提供的一个附加模块。附加的意思就是，如果你打算用它，那么你需要引入一人单独的 js 文件，然后在声明“根模块”时注明依赖的 ngResource 模块，接着就可以使用它提供的 $resource 服务了。完整的过程形如：

<!DOCTYPE html>

<html ng-app="Demo">

<head>

<meta charset="utf-8" />

<title>AngularJS</title>

<script type="text/javascript" src="http://ajax.googleapis.com/ajax/libs/angularjs/1.0.3/angular.min.js"></script>

<script type="text/javascript" src="http://ajax.googleapis.com/ajax/libs/angularjs/1.0.3/angular-resource.js"></script>

</head>

<body>

<div ng-controller="TestCtrl"></div>

<script type="text/javascript" charset="utf-8">

var app = angular.module('Demo', ['ngResource'], angular.noop);

app.controller('TestCtrl', function($scope, $resource){

console.log($resource);

});

</script>

</body>

</html>

$resource 服务，整体上来说，比较像是使用类似 ORM 的方式来包装了 AJAX 调用。区别就是 ORM 是操作数据库，即拼出 SQL 语句之后，作 execute 方法调用。而 $resource 的方式是构造出 AJAX 请求，然后发出请求。同时，AJAX 请求是需要回调处理的，这方面，$resource 的机制可以使你在一些时候省掉回调处理，当然，是否作回调处理在于业务情形及容错需求了。

使用上 $resource 分成了“类”与“实例”这两个层面。一般地，类的方法调用就是直观的调用形式，通常会返回一个对象，这个对象即为“实例”。

“实例”贯穿整个服务的使用过程。“实例”的数据是填充方式，即因为异步关系，回调函数没有执行时，实例已经存在，只是可能它还没有相关数据，回调执行之后，相关数据被填充到实例对象当中。实例的方法一般就是在类方法名前加一个 $ ，调用上，根据定义，实例数据可能会做一些自动的参数填充，这点是区别实例与类的调用上的不同。

好吧，上面这些话可能需要在看了接下来的内容之后再回过来理解。

#### 15.2. 基本定义

就像使用 ORM 一般要先定义 Model 一样，使用 $resource 需要先定义“资源”，也就是先定义一些 HTTP 请求。

在业务场景上，我们假设为，我们需要操作“书”这个实体，包括创建create，获取详情read，修改update，删除delete，批量获取multi，共五个操作方法。实体属性有：唯一标识id，标题title，作者author。

我们把这些操作定义成 $resource 的资源：

var app = angular.module('Demo', ['ngResource'], angular.noop);

app.controller('BookCtrl', function($scope, $resource){

var actions = {

create: {method: 'POST', params: {\_method: 'create'}},

read: {method: 'POST', params: {\_method: 'read'}},

update: {method: 'POST', params: {\_method: 'update'}},

delete: {method: 'POST', params: {\_method: 'delete'}},

multi: {method: 'POST', params: {\_method: 'multi'}}

}

var Book = $resource('/book', {}, actions);

});

定义是使用使用 $resource 这个函数就可以了，它接受三个参数：

* url
* 默认的params（这里的 params 即是 GET 请求的参数，POST 的参数单独叫做“postData”）
* 方法映射

方法映射是以方法名为 key ，以一个对象为 value ，这个 value 可以有三个成员：

* method, 请求方法，'GET', 'POST', 'PUT', 'DELETE' 这些
* params, 默认的 GET 参数
* isArray, 返回的数据是不是一个列表

#### 15.3. 基本使用

在定义了资源之后，我们看如果使用这些资源，发出请求：

var book = Book.read({id: '123'}, function(response){

console.log(response);

});

这里我们进行 Book 的“类”方法调用。在方法的使用上，根据官方文档：

HTTP GET "class" actions: Resource.action([parameters], [success], [error])

non-GET "class" actions: Resource.action([parameters], postData, [success], [error])

non-GET instance actions: instance.$action([parameters], [success], [error])

我们这里是第二种形式，即类方法的非 GET 请求。我们给的参数会作为 postData 传递。如果我们需要 GET 参数，并且还需要一个错误回调，那么：

var book = Book.read({get: 'haha'}, {id: '123'},

function(response){

console.log(response);

},

function(error){

console.log(error);

}

);

调用之后，我们会立即得到的 book ，它是 Book 类的一个实例。这里所谓的实例，实际上就是先把所有的 action 加一个 $ 前缀放到一个空对象里，然后把发出的参数填充进去。等请求返回了，把除 action 以外的成员删除掉，再把请求返回的数据填充到这个对象当中。所以，如果我们这样：

var book = Book.read({id: '123'}, function(response){

console.log(book);

});

console.log(book)

就能看到 book 实例的变化过程了。

现在我们得到一个真实的实例，看一下实例的调用过程：

//响应的数据是 {result: 0, msg: '', obj: {id: 'xxx'}}

var book = Book.create({title: '测试标题', author: '测试作者'}, function(response){

console.log(book);

});

可以看到，在请求回调之后， book 这个实例的成员已经被响应内容填充了。但是这里有一个问题，我们返回的数据，并不适合一个 book 实例。格式先不说，它把 title 和 author 这些信息都丢了（因为响应只返回了 id ）。

如果仅仅是格式问题，我们可以通过配置 $http 服务来解决（ AJAX 请求都要使用 $http 服务的）：

$http.defaults.transformResponse = function(data){return angular.fromJson(data).obj};

当然，我们也可以自己来解决一下丢信息的问题：

var p = {title: '测试标题', author: '测试作者'};

var book = Book.create(p, function(response){

angular.extend(book, p);

console.log(book);

});

不过，始终会有一些不方便了。比较正统的方式应该是调节服务器端的响应，让服务器端也具有和前端一样的实例概念，返回的是完整的实例信息。即使这样，你也还要考虑格式的事。

现在我们得到了一个真实的 book 实例了，带有 id 信息。我们尝试一下实例的方法调用，先回过去头看一下那三种调用形式，对于实例只有第三种形式：

non-GET instance actions: instance.$action([parameters], [success], [error])

首先解决一个疑问，如果一个实例是进行一个 GET 的调用会怎么样？没有任何问题，这当然没有任何问题的，形式和上面一样。

如何实例是做 POST 请求的话，从形式上看，我们无法控制请求的 postData ？是的，所有的 POST 请求，其 postData 都会被实例数据自动填充，形式上我们只能控制 params 。

所以，如果是在做修改调用的话：

book.$update({title: '新标题', author: '测试作者'}, function(response){

console.log(book);

});

这样是没有意义的并且错误的。因为要修改的数据只是作为 GET 参数传递了，而 postData传递的数据就是当前实例的数据，并没有任何修改。

正确的做法：

book.title = '新标题'

book.$update(function(response){

console.log(book);

});

显然，这种情况下，回调都可以省了：

book.title = '新标题'

book.$update();

#### 15.4. 定义和使用时的占位量

两方面。一是在定义时，在其 URL 中可以使用变量引用的形式（类型于定义锚点路由时那样）。第二时定义默认 params ，即 GET 参数时，可以定义为引用 postData 中的某变量。比如我们这样改一下：

var Book = $resource('/book/:id', {}, actions);

var book = Book.read({id: '123'}, {}, function(response){

console.log(response);

});

在 URL 中有一个 :id ，表示对 params 中 id 这个变量的引用。因为 read 是一个 POST 请求，根据调用形式，第一个参数是 params ，第二个参数是 postData 。这样的调用结果就是，我们会发一个 POST 请求到如下地址， postData 为空：

/book/123?\_method=read

再看默认的 params 中引用 postData 变量的形式：

var Book = $resource('/book', {id: '@id'}, actions);

var book = Book.read({title: 'xx'}, {id: '123'}, function(response){

console.log(response);

});

这样会出一个 POST 请求， postData 内容中有一个 id 数据，访问的 URL 是：

/book?\_method=read&id=123&title=xx

这两个机制也可以联合使用：

var Book = $resource('/book/:id', {id: '@id'}, actions);

var book = Book.read({title: 'xx'}, {id: '123'}, function(response){

console.log(response);

});

结果就是出一个 POST 请求， postData 内容中有一个 id 数据，访问的 URL 是：

/book/123?\_method=read&title=xx

#### 15.5. 实例

ngResource 要举一个实例是比较麻烦的事。因为它必须要一个后端来支持，这里如果我用 Python 写一个简单的后端，估计要让这个后端跑起来对很多人来说都是问题。所以，我在几套公共服务的 API 中纠结考察了一番，最后使用 [www.rememberthemilk.com](http://www.rememberthemilk.com/) 的 API 来做了一个简单的，可用的例子。

例子见： <http://zouyesheng.com/demo/ng-resource-demo.html> (可以直接下载看源码)

先说一下 API 的情况。这里的请求调用全是跨域的，所以交互上全部是使用了 JSONP 的形式。 API 的使用有使用签名认证机制，嗯， js 中直接算 md5 是可行的，我用了一个现成的库（但是好像不能处理中文吧）。

这个例子中的 LoginCtrl 大家就不用太关心了，参见官方的文档，走完流程拿到 token 完事。与 ngResource 相关的是 MainCtrl 中的东西。

其实从这个例子中就可以看出，目前 ngResource 的机制对于服务端返回的数据的格式是严重依赖的，同时也可以反映出 $http 对一些场景根本无法应对的局限。所以，我现在的想法是理解 ngResource 的思想，真正需要的人自己使用 jQuery 重新实现一遍也许更好。这应该也花不了多少时间， ngResource 的代码本来不多。

我为什么说 $http 在一些场景中有局限呢。在这个例子当中，所有的请求都需要带一个签名，签名值是由请求中带的参数根据规则使用 md5 方法计算出的值。我找不到一个 hook 可以让我在请求出去之前修改这个请求（添加上签名）。所以在这个例子当中，我的做法是根据ngResource 的请求最后会使用 $httpBackend 这个底层服务，在 module 定义时我自己复制官方的相关代码，重新定义 $httpBackend 服务，在需要的地方做我自己的修改：

script.src = sign\_url(url);

不错，我就改了这一句，但我不得不复制了 50 行官方源码到我的例子中。

另外一个需要说的是对返回数据的处理。因为 ngResource 会使用返回的数据直接填充实例，所以这个数据格式就很重要。

首先，我们可以使用 $http.defaults.transformResponse 来统一处理一下返回的数据，但是这并不能解决所有问题，可目前 ngResource 并不提供对每一个 action 的单独的后处理回调函数项。除非你的服务端是经过专门的适应性设计的，否则你用 ngResource 不可能爽。例子中，我为了获取当前列表的结果，我不得不自己去封装结果：

var list\_list = List.getList(function(){

var res = list\_list[1];

while(list\_list.length > 0){list\_list.pop()};

angular.forEach(res.list, function(v){

list\_list.push(new List({list: v}));

});

$scope.list\_list = list\_list;

$scope.show\_add = true;

return;

});

### 16. AngularJS与其它框架的混用(jQuery, Dojo)

这个问题似乎很多人都关心，但是事实是，如果了解了 ng 的工作方式，这本来就不是一个问题了。

在我自己使用 ng 的过程当中，一直是混用 jQuery 的，以前还要加上一个 Dojo 。只要了解每种框架的工作方式，在具体的代码中每个框架都做了什么事，那么整体上控制起来就不会有问题。

回到 ng 上来看，首先对于 jQuery 来说，最开始说提到过，在 DOM 操作部分， ng 与 jQuery 是兼容的，如果没有 jQuery ， ng 自己也实现了兼容的部分 API 。

同时，最开始也提到过， ng 的使用最忌讳的一点就是修改 DOM 结构——你应该使用 ng 的模板机制进行数据绑定，以此来控制 DOM 结构，而不是直接操作。换句话来说，在不动 DOM 结构的这个前提之下，你的数据随便怎么改，随便使用哪个框架来控制都是没问题的，到时如有必要使用 $scope.$digest() 来通知 ng 一下即可。

下面这个例子，我们使用了 jQuery 中的 Deferred ( $.ajax 就是返回一个 Deferred )，还使用了 ng 的 $timeout ，当然是在 ng 的结构之下：

1 <!DOCTYPE html>

2 <html ng-app="Demo">

3 <head>

4 <meta charset="utf-8" />

5 <title>AngularJS</title>

6 </head>

7 <body>

8

9 <div ng-controller="TestCtrl">

10 <span ng-click="go()">{{ a }}</span>

11 </div>

12

13 <script type="text/javascript"

14 src="http://ajax.googleapis.com/ajax/libs/jquery/1.9.1/jquery.min.js">

15 </script>

16 <script type="text/javascript"

17 src="http://ajax.googleapis.com/ajax/libs/angularjs/1.0.3/angular.min.js">

18 </script>

19

20 <script type="text/javascript">

21 var app = angular.module('Demo', [], angular.noop);

22 app.controller('TestCtrl', function($scope, $timeout){

23 $scope.a = '点击我开始';

24

25 var defer = $.Deferred();

26 var f = function(){

27 if($scope.a == ''){$scope.a = '已停止'; return}

28 defer.done(function(){

29 $scope.a.length < 10 ? $scope.a += '>' : $scope.a = '>';

30 $timeout(f, 100);

31 });

32 }

33 defer.done(function(){$scope.a = '>'; f()});

34

35 $scope.go = function(){

36 defer.resolve();

37 $timeout(function(){$scope.a = ''}, 5000);

38 }

39 });

40 </script>

41 </body>

42 </html>

再把 Dojo 加进来看与 DOM 结构相关的例子。之前说过，使用 ng 就最好不要手动修改 DOM 结构，但这里说两点：

1. 对于整个页面，你可以只在局部使用 ng ，不使用 ng 的地方你可以随意控制 DOM 。
2. 如果 DOM 结构有变动，你可以在 DOM 结构定下来之后再初始化 ng 。

下面这个例子使用了 AngularJS ， jQuery ， Dojo ：

1 <!DOCTYPE html>

2 <html>

3 <head>

4 <meta charset="utf-8" />

5 <title>AngularJS</title>

6 <link rel="stylesheet"

7 href="http://ajax.googleapis.com/ajax/libs/dojo/1.9.1/dijit/themes/claro/claro.css" media="screen" />

8 </head>

9 <body class="claro">

10

11 <div ng-controller="TestCtrl" id="test\_ctrl">

12

13 <p ng-show="!btn\_disable">

14 <button ng-click="change()">调用dojo修改按钮</button>

15 </p>

16

17 <p id="btn\_wrapper">

18 <button data-dojo-type="dijit/form/Button" type="button">{{ a }}</button>

19 </p>

20

21 <p>

22 <input ng-model="dialog\_text" ng-init="dialog\_text='对话框内容'" />

23 <button ng-click="dialog(dialog\_text)">显示对话框</button>

24 </p>

25

26 <p ng-show="show\_edit\_text" style="display: none;">

27 <span>需要编辑的内容:</span>

28 <input ng-model="text" />

29 </p>

30

31 <div id="editor\_wrapper">

32 <div data-dojo-type="dijit/Editor" id="editor"></div>

33 </div>

34

35 </div>

36

37

38 <script type="text/javascript"

39 src="http://ajax.googleapis.com/ajax/libs/dojo/1.9.1/dojo/dojo.js">

40 </script>

41 <script type="text/javascript"

42 src="http://ajax.googleapis.com/ajax/libs/jquery/1.9.1/jquery.min.js">

43 </script>

44 <script type="text/javascript"

45 src="http://ajax.googleapis.com/ajax/libs/angularjs/1.0.3/angular.min.js">

46 </script>

47

48 <script type="text/javascript">

49

50 require(['dojo/parser', 'dijit/Editor'], function(parser){

51 parser.parse($('#editor\_wrapper')[0]).then(function(){

52 var app = angular.module('Demo', [], angular.noop);

53

54 app.controller('TestCtrl', function($scope, $timeout){

55 $scope.a = '我是ng, 也是dojo';

56 $scope.show\_edit\_text = true;

57

58 $scope.change = function(){

59 $scope.a = 'DOM结构已经改变(不建议这样做)';

60 require(['dojo/parser', 'dijit/form/Button', 'dojo/domReady!'],

61 function(parser){

62 parser.parse($('#btn\_wrapper')[0]);

63 $scope.btn\_disable = true;

64 }

65 );

66 }

67

68 $scope.dialog = function(text){

69 require(["dijit/Dialog", "dojo/domReady!"], function(Dialog){

70 var dialog = new Dialog({

71 title: "对话框哦",

72 content: text,

73 style: "width: 300px"

74 });

75 dialog.show();

76 });

77 }

78

79 require(['dijit/registry'], function(registry){

80 var editor = registry.byId('editor');

81 $scope.$watch('text', function(new\_v){

82 editor.setValue(new\_v);

83 });

84 });

85

86 });

87

88 angular.bootstrap(document, ['Demo']);

89 });

90

91 });

92

93 </script>

94 </body>

95 </html>

### 17. 自定义过滤器

先来回顾一下 ng 中的一些概念：

* module ，代码的组织单元，其它东西都是在定义在具体的模块中的。
* app ，业务概念，可能会用到多个模块。
* service ，仅在数据层面实现特定业务功能的代码封装。
* controller ，与 DOM 结构相关联的东西，即是一种业务封装概念，又体现了项目组织的层级结构。
* filter ，改变输入数据的一种机制。
* directive ，与 DOM 结构相关联的，特定功能的封装形式。

上面的这几个概念基本上就是 ng 的全部。每一部分都可以自由定义，使用时通过各要素的相互配合来实现我们的业务需求。

我们从最开始一致打交道的东西基本上都是 controller 层面的东西。在前面，也介绍了 module和 service 的自定义。剩下的会介绍 filter 和 directive 的定义。基本上这几部分的定义形式都是一样的，原理上是通过 provider 来做注入形式的声明，在实际操作过程中，又有很多 shortcut 式的声明方式。

过滤器的自定义是最简单的，就是一个函数，接受输入，然后返回结果。在考虑过滤器时，我觉得很重要的一点： 无状态 。

具体来说，过滤器就是一个函数，函数的本质含义就是确定的输入一定得到确定的输出。虽然filter 是定义在 module 当中的，而且 filter 又是在 controller 的 DOM 范围内使用的，但是，它和具体的 module ， controller ， scope 这些概念都没有关系（虽然在这里你可以使用 js 的闭包机制玩些花样），它仅仅是一个函数，而已。换句话说，它没有任何上下文关联的能力。

过滤器基本的定义方式：

var app = angular.module('Demo', [], angular.noop);

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

var filter = function(input){

return input + '...';

};

return filter;

});

上面的代码定义了一个叫做 map 的过滤器。使用时：

<p>示例数据: {{ a|map }}</p>

过滤器也可以带参数，多个参数之间使用 : 分割，看一个完整的例子：

1 <div ng-controller="TestCtrl">

2 <p>示例数据: {{ a|map:map\_value:'>>':'(no)' }}</p>

3 <p>示例数据: {{ b|map:map\_value:'>>':'(no)' }}</p>

4 </div>

5

6

7 <script type="text/javascript">

8

9 var app = angular.module('Demo', [], angular.noop);

10 app.controller('TestCtrl', function($scope){

11 $scope.map\_value = {

12 a: '一',

13 b: '二',

14 c: '三'

15 }

16 $scope.a = 'a';

17 });

18

19 app.filter('map', function(){

20 var filter = function(input, map\_value, append, default\_value){

21 var r = map\_value[input];

22 if(r === undefined){ return default\_value + append }

23 else { return r + append }

24 };

25 return filter;

26 });

27

28 angular.bootstrap(document, ['Demo']);

29 </script>

### 18. 自定义指令directive

这是 ng 最强大的一部分，也是最复杂最让人头疼的部分。

目前我们看到的所谓“模板”系统，只不过是官方实现的几个指令而已。这意味着，通过自定义各种指令，我们不但可以完全定义一套“模板”系统，更可以把 HTML 页面直接打造成为一种 DSL （领域特定语言）。

#### 18.1. 指令的使用

使用指令时，它的名字可以有多种形式，把指令放在什么地方也有多种选择。

通常，指令的定义名是形如 ngBind 这样的 “camel cased” 形式。在使用时，它的引用名可以是：

* ng:bind
* ng\_bind
* ng-bind
* x-ng-bind
* data-ng-bind

你可以根据你自己是否有 “HTML validator” 洁癖来选择。

指令可以放在多个地方，它们的作用相同：

* <span my-dir="exp"></span> 作为标签的属性
* <span class="my-dir: exp;"></span> 作为标签类属性的值
* <my-dir></my-dir> 作为标签
* <!-- directive: my-dir exp --> 作为注释

这些方式可以使用指令定义中的 restrict 属性来控制。

可以看出，指令即可以作为标签使用，也可以作为属性使用。仔细考虑一下，这在类 XML 的结构当中真算得上是一种神奇的机制。

#### 18.2. 指令的执行过程

ng 中对指令的解析与执行过程是这样的：

* 浏览器得到 HTML 字符串内容，解析得到 DOM 结构。
* ng 引入，把 DOM 结构扔给 $compile 函数处理：
  + 找出 DOM 结构中有变量占位符
  + 匹配找出 DOM 中包含的所有指令引用
  + 把指令关联到 DOM
  + 关联到 DOM 的多个指令按权重排列
  + 执行指令中的 compile 函数（改变 DOM 结构，返回 link 函数）
  + 得到的所有 link 函数组成一个列表作为 $compile 函数的返回
* 执行 link 函数（连接模板的 scope）。

#### 18.3. 基本的自定义方法

自定义一个指令可以非常非常的复杂，但是其基本的调用形式，同自定义服务大概是相同的：

<p show style="font-size: 12px;"></p>

<script type="text/javascript">

var app = angular.module('Demo', [], angular.noop);

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

var func = function($scope, $element, $attrs){

console.log($scope);

console.log($element);

console.log($attrs);

}

return func;

//return {compile: function(){return func}}

});

angular.bootstrap(document, ['Demo']);

</script>

如果在 directive 中直接返回一个函数，则这个函数会作为 compile 的返回值，也即是作为link 函数使用。这里说的 compile 和 link 都是一个指令的组成部分，一个完整的定义应该返回一个对象，这个对象包括了多个属性：

* name
* priority
* terminal
* scope
* controller
* require
* restrict
* template
* templateUrl
* replace
* transclude
* compile
* link

上面的每一个属性，都可以单独探讨的。

下面是一个完整的基本的指令定义例子：

<code lines>

//失去焦点使用 jQuery 的扩展支持冒泡

app.directive('ngBlur', function($parse){

return function($scope, $element, $attr){

var fn = $parse($attr['ngBlur']);

$element.on('focusout', function(event){

fn($scope, {$event: event});

});

}

});

</code>

<div code lines>

//失去焦点使用 jQuery 的扩展支持冒泡

app.directive('ngBlur', function($parse){

return function($scope, $element, $attr){

var fn = $parse($attr['ngBlur']);

$element.on('focusout', function(event){

fn($scope, {$event: event});

});

}

});

</div>

1 var app = angular.module('Demo', [], angular.noop);

2

3 app.directive('code', function(){

4 var func = function($scope, $element, $attrs){

5

6 var html = $element.text();

7 var lines = html.split('\n');

8

9 //处理首尾空白

10 if(lines[0] == ''){lines = lines.slice(1, lines.length - 1)}

11 if(lines[lines.length-1] == ''){lines = lines.slice(0, lines.length - 1)}

12

13 $element.empty();

14

15 //处理外框

16 (function(){

17 $element.css('clear', 'both');

18 $element.css('display', 'block');

19 $element.css('line-height', '20px');

20 $element.css('height', '200px');

21 })();

22

23 //是否显示行号的选项

24 if('lines' in $attrs){

25 //处理行号

26 (function(){

27 var div = $('<div style="width: %spx; background-color: gray; float: left; text-align: right; padding-right: 5px; margin-right: 10px;"></div>'

28 .replace('%s', String(lines.length).length \* 10));

29 var s = '';

30 angular.forEach(lines, function(\_, i){

31 s += '<pre style="margin: 0;">%s</pre>\n'.replace('%s', i + 1);

32 });

33 div.html(s);

34 $element.append(div);

35 })();

36 }

37

38 //处理内容

39 (function(){

40 var div = $('<div style="float: left;"></div>');

41 var s = '';

42 angular.forEach(lines, function(l){

43 s += '<span style="margin: 0;">%s</span><br />\n'.replace('%s', l.replace(/\s/g, '<span>&nbsp;</span>'));

44 });

45 div.html(s);

46 $element.append(div);

47 })();

48 }

49

50 return {link: func,

51 restrict: 'AE'}; //以元素或属性的形式使用命令

52 });

53

54 angular.bootstrap(document, ['Demo']);

上面这个自定义的指令，做的事情就是解析节点中的文本内容，然后修改它，再把生成的新内容填充到节点当中去。其间还涉及了节点属性值 lines 的处理。这算是指令中最简单的一种形式。因为它是“一次性使用”，中间没有变量的处理。比如如果节点原来的文本内容是一个变量引用，类似于 {{ code }} ，那上面的代码就不行了。这种情况麻烦得多。后面会讨论。

#### 18.4. 属性值类型的自定义

官方代码中的 ng-show 等算是我说的这种类型。使用时主要是在节点加添加一个属性值以附加额外的功能。看一个简单的例子：

<p color="red">有颜色的文本</p>

<color color="red">有颜色的文本</color>

<script type="text/javascript">

var app = angular.module('Demo', [], angular.noop);

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

var link = function($scope, $element, $attrs){

$element.css('color', $attrs.color);

}

return {link: link,

restrict: 'AE'};

});

angular.bootstrap(document, ['Demo']);

</script>

我们定义了一个叫 color 的指令，可以指定节点文本的颜色。但是这个例子还无法像 ng-show 那样工作的，这个例子只能渲染一次，然后就无法根据变量来重新改变显示了。要响应变化，我们需要手工使用 scope 的 $watch 来处理：

1

2 <div ng-controller="TestCtrl">

3 <p color="color">有颜色的文本</p>

4 <p color="'blue'">有颜色的文本</p>

5 </div>

6

7 <script type="text/javascript">

8

9 var app = angular.module('Demo', [], angular.noop);

10

11 app.directive('color', function(){

12 var link = function($scope, $element, $attrs){

13 $scope.$watch($attrs.color, function(new\_v){

14 $element.css('color', new\_v);

15 });

16 }

17 return link;

18 });

19

20 app.controller('TestCtrl', function($scope){

21 $scope.color = 'red';

22 });

23

24 angular.bootstrap(document, ['Demo']);

25 </script>

#### 18.5. Compile的细节

指令的处理过程，是 ng 的 Compile 过程的一部分，它们也是紧密联系的。继续深入指令的定义方法，首先就要对 Compile 的过程做更细致的了解。

前面说过， ng 对页面的处理过程：

* 浏览器把 HTML 字符串解析成 DOM 结构。
* ng 把 DOM 结构给 $compile ，返回一个 link 函数。
* 传入具体的 scope 调用这个 link 函数。
* 得到处理后的 DOM ，这个 DOM 处理了指令，连接了数据。

$compile 最基本的使用方式：

var link = $compile('<p>{{ text }}</p>');

var node = link($scope);

console.log(node);

上面的 $compile 和 link 调用时都有额外参数来实现其它功能。先看 link 函数，它形如：

function(scope[, cloneAttachFn]

第二个参数 cloneAttachFn 的作用是，表明是否复制原始节点，及对复制节点需要做的处理，下面这个例子说明了它的作用：

<div ng-controller="TestCtrl"></div>

<div id="a">A {{ text }}</div>

<div id="b">B </div>

app.controller('TestCtrl', function($scope, $compile){

var link = $compile($('#a'));

//true参数表示新建一个完全隔离的scope,而不是继承的child scope

var scope = $scope.$new(true);

scope.text = '12345';

//var node = link(scope, function(){});

var node = link(scope);

$('#b').append(node);

});

cloneAttachFn 对节点的处理是有限制的，你可以添加 class ，但是不能做与数据绑定有关的其它修改（修改了也无效）：

app.controller('TestCtrl', function($scope, $compile){

var link = $compile($('#a'));

var scope = $scope.$new(true);

scope.text = '12345';

var node = link(scope, function(clone\_element, scope){

clone\_element.text(clone\_element.text() + ' ...'); //无效

clone\_element.text('{{ text2 }}'); //无效

clone\_element.addClass('new\_class');

});

$('#b').append(node);

});

修改无效的原因是，像 {{ text }} 这种所谓的 Interpolate 在 $compile 中已经被处理过了，生成了相关函数（这里起作用的是 directive 中的一个 postLink 函数），后面执行 link 就是执行了 $compile 生成的这些函数。当然，如果你的文本没有数据变量的引用，那修改是会有效果的。

前面在说自定义指令时说过， link 函数是由 compile 函数返回的，也就像前面说的，应该把改变 DOM 结构的逻辑放在 compile 函数中做。

$compile 还有两个额外的参数：

$compile(element, transclude, maxPriority);

maxPriority 是指令的权重限制，这个容易理解，后面再说。

transclude 是一个函数，这个函数会传递给 compile 期间找到的 directive 的 compile 函数（编译节点的过程中找到了指令，指令的 compile 函数会接受编译时传递的 transclude 函数作为其参数）。

但是在实际使用中，除我们手工在调用 $compile 之外，初始化时的根节点 compile 是不会传递这个参数的。

在我们定义指令时，它的 compile 函数是这个样子的：

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

事实上， transclude 的值，就是 directive 所在的 原始 节点，把原始节点重新做了编译之后得到的 link 函数（需要 directive 定义时使用 transclude 选项），后面会专门演示这个过程。所以，官方文档上也把 transclude 函数描述成 link 函数的样子（如果自定义的指令只用在自己手动 $compile 的环境中，那这个函数的形式是可以随意的）：

{function(angular.Scope[, cloneAttachFn]}

所以记住，定义指令时， compile 函数的第三个参数 transclude ，就是一个 link ，装入scope 执行它你就得到了一个节点。

#### 18.6. transclude的细节

transclude 有两方面的东西，一个是使用 $compile 时传入的函数，另一个是定义指令的compile 函数时接受的一个参数。虽然这里的一出一进本来是相互对应的，但是实际使用中，因为大部分时候不会手动调用 $compile ，所以，在“默认”情况下，指令接受的 transclude又会是一个比较特殊的函数。

看一个基本的例子：

var app = angular.module('Demo', [], angular.noop);

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

var func = function(element, attrs, transclude){

var sum = transclude(1, 2);

console.log(sum);

console.log(element);

}

return {compile: func,

restrict: 'E'};

});

app.controller('TestCtrl', function($scope, $compile, $element){

var s = '<more>123</more>';

var link = $compile(s, function(a, b){return a + b});

var node = link($scope);

$element.append(node);

});

angular.bootstrap(document, ['Demo']);

我们定义了一个 more 指令，它的 compile 函数的第三个参数，就是我们手工 $compile 时传入的。

如果不是手工 $compile ，而是 ng 初始化时找出的指令，则 transclude 是一个 link 函数（指令定义需要设置 transclude 选项）：

<div more>123</div>

app.directive('more', function($rootScope, $document){

var func = function(element, attrs, link){

var node = link($rootScope);

node.removeAttr('more'); //不去掉就变死循环了

$('body', $document).append(node);

}

return {compile: func,

transclude: 'element', // element是节点没,其它值是节点的内容没

restrict: 'A'};

});

#### 18.7. 把节点内容作为变量处理的类型

回顾最开始的那个代码显示的例子，那个例子只能处理一次节点内容。如果节点的内容是一个变量的话，需要用另外的思路来考虑。这里我们假设的例子是，定义一个指令 showLenght ，它的作用是在一段文本的开头显示出这段节点文本的长度，节点文本是一个变量。指令使用的形式是：

<div ng-controller="TestCtrl">

<div show-length>{{ text }}</div>

<button ng-click="text='xx'">改变</button>

</div>

从上面的 HTML 代码中，大概清楚 ng 解析它的过程（只看 show-length 那一行）：

* 解析 div 时发现了一个 show-length 的指令。
* 如果 show-length 指令设置了 transclude 属性，则 div 的节点内容被重新编译，得到的 link 函数作为指令 compile 函数的参数传入。
* 如果 show-length 指令没有设置 transclude 属性，则继续处理它的子节点（TextNode ）。
* 不管是上面的哪种情况，都会继续处理到 {{ text }} 这段文本。
* 发现 {{ text }} 是一个 Interpolate ，于是自动在此节点中添加了一个指令，这个指令的 link 函数就是为 scope 添加了一个 $watch ，实现的功能是是当 scope作 $digest 的时候，就更新节点文本。

与处理 {{ text }} 时添加的指令相同，我们实现 showLength 的思路，也就是：

* 修改原来的 DOM 结构
* 为 scope 添加 $watch ，当 $digest 时修改指定节点的文本，其值为指定节点文本的长度。

代码如下：

app.directive('showLength', function($rootScope, $document){

var func = function(element, attrs, link){

return function(scope, ielement, iattrs, controller){

var node = link(scope);

ielement.append(node);

var lnode = $('<span></span>');

ielement.prepend(lnode);

scope.$watch(function(scope){

lnode.text(node.text().length);

});

};

}

return {compile: func,

transclude: true, // element是节点没,其它值是节点的内容没

restrict: 'A'};

});

上面代码中，因为设置了 transclude 属性，我们在 showLength 的 link 函数（就是 return的那个函数）中，使用 func 的第三个函数来重塑了原来的文本节点，并放在我们需要的位置上。然后，我们添加自己的节点来显示长度值。最后给当前的 scope 添加 $watch ，以更新这个长度值。

#### 18.8. 指令定义时的参数

指令定义时的参数如下：

* name
* priority
* terminal
* scope
* controller
* require
* restrict
* template
* templateUrl
* replace
* transclude
* compile
* link

现在我们开始一个一个地吃掉它们……，但是并不是按顺序讲的。

priority

这个值设置指令的权重，默认是 0 。当一个节点中有多个指令存在时，就按着权限从大到小的顺序依次执行它们的 compile 函数。相同权重顺序不定。

terminal

是否以当前指令的权重为结束界限。如果这值设置为 true ，则节点中权重小于当前指令的其它指令不会被执行。相同权重的会执行。

restrict

指令可以以哪些方式被使用，可以同时定义多种方式。

* E 元素方式 <my-directive></my-directive>
* A 属性方式 <div my-directive="exp"> </div>
* C 类方式 <div class="my-directive: exp;"></div>
* M 注释方式 <!-- directive: my-directive exp -->

transclude

前面已经讲过基本的用法了。可以是 'element' 或 true 两种值。

compile

基本的定义函数。 function compile(tElement, tAttrs, transclude) { ... }

link

前面介绍过了。大多数时候我们不需要单独定义它。只有 compile 未定义时 link 才会被尝试。 function link(scope, iElement, iAttrs, controller) { ... }

scope

scope 的形式。 false 节点的 scope ， true 继承创建一个新的 scope ， {} 不继承创建一个新的隔离 scope 。 {@attr: '引用节点属性', =attr: '把节点属性值引用成scope属性值', &attr: '把节点属性值包装成函数'}

controller

为指令定义一个 controller ， function controller($scope, $element, $attrs, $transclude) { ... }

name

指令的 controller 的名字，方便其它指令引用。

require

要引用的其它指令 conroller 的名字， ?name 忽略不存在的错误， ^name 在父级查找。

template

模板内容。

templateUrl

从指定地址获取模板内容。

replace

是否使用模板内容替换掉整个节点， true 替换整个节点， false 替换节点内容。

<a b></a>

var app = angular.module('Demo', [], angular.noop);

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

var func = function(element, attrs, link){

console.log('a');

}

return {compile: func,

priority: 1,

restrict: 'EA'};

});

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

var func = function(element, attrs, link){

console.log('b');

}

return {compile: func,

priority: 2,

//terminal: true,

restrict: 'A'};

});

上面几个参数值都是比较简单且容易理想的。

再看 scope 这个参数：

<div ng-controller="TestCtrl">

<div a b></div>

</div>

1 var app = angular.module('Demo', [], angular.noop);

2

3 app.directive('a', function(){

4 var func = function(element, attrs, link){

5 return function(scope){

6 console.log(scope);

7 }

8 }

9

10 return {compile: func,

11 scope: true,

12 restrict: 'A'};

13 });

14

15 app.directive('b', function(){

16 var func = function(element, attrs, link){

17 return function(scope){

18 console.log(scope);

19 }

20 }

21

22 return {compile: func,

23 restrict: 'A'};

24 });

25

26 app.controller('TestCtrl', function($scope){

27 $scope.a = '123';

28 console.log($scope);

29 });

对于 scope ：

* 默认为 false ， link 函数接受的 scope 为节点所在的 scope 。
* 为 true 时，则 link 函数中第一个参数（还有 controller 参数中的 $scope）， scope 是节点所在的 scope 的 child scope ，并且如果节点中有多个指令，则只要其中一个指令是 true 的设置，其它所有指令都会受影响。

这个参数还有其它取值。当其为 {} 时，则 link 接受一个完全隔离（isolate）的 scope ，于true 的区别就是不会继承其它 scope 的属性。但是这时，这个 scope 的属性却可以有很灵活的定义方式：

@attr 引用节点的属性。

<div ng-controller="TestCtrl">

<div a abc="here" xx="{{ a }}" c="ccc"></div>

</div>

var app = angular.module('Demo', [], angular.noop);

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

var func = function(element, attrs, link){

return function(scope){

console.log(scope);

}

}

return {compile: func,

scope: {a: '@abc', b: '@xx', c: '@'},

restrict: 'A'};

});

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

$scope.a = '123';

});

* @abc 引用 div 节点的 abc 属性。
* @xx 引用 div 节点的 xx 属性，而 xx 属性又是一个变量绑定，于是 scope 中 b 属性值就和 TestCtrl 的 a 变量绑定在一起了。
* @ 没有写 attr name ，则默认取自己的值，这里是取 div 的 c 属性。

=attr 相似，只是它把节点的属性值当成节点 scope 的属性名来使用，作用相当于上面例子中的 @xx ：

<div ng-controller="TestCtrl">

<div a abc="here"></div>

</div>

var app = angular.module('Demo', [], angular.noop);

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

var func = function(element, attrs, link){

return function(scope){

console.log(scope);

}

}

return {compile: func,

scope: {a: '=abc'},

restrict: 'A'};

});

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

$scope.here = '123';

});

&attr 是包装一个函数出来，这个函数以节点所在的 scope 为上下文。来看一个很爽的例子：

<div ng-controller="TestCtrl">

<div a abc="here = here + 1" ng-click="show(here)">这里</div>

<div>{{ here }}</div>

</div>

1 var app = angular.module('Demo', [], angular.noop);

2

3 app.directive('a', function(){

4 var func = function(element, attrs, link){

5 return function llink(scope){

6 console.log(scope);

7 scope.a();

8 scope.b();

9

10 scope.show = function(here){

11 console.log('Inner, ' + here);

12 scope.a({here: 5});

13 }

14 }

15 }

16

17 return {compile: func,

18 scope: {a: '&abc', b: '&ngClick'},

19 restrict: 'A'};

20 });

21

22 app.controller('TestCtrl', function($scope){

23 $scope.here = 123;

24 console.log($scope);

25

26 $scope.show = function(here){

27 console.log(here);

28 }

29 });

scope.a 是 &abc ，即：

scope.a = function(){here = here + 1}

只是其中的 here 是 TestCtrl 的。

scope.b 是 &ngClick ，即：

scope.b = function(){show(here)}

这里的 show() 和 here 都是 TestCtrl 的，于是上面的代码最开始会在终端输出一个 124 。

当点击“这里”时，这时执行的 show(here) 就是 llink 中定义的那个函数了，与 TestCtrl 无关。但是，其间的 scope.a({here:5}) ，因为 a 执行时是 TestCtrl 的上下文，于是向 a 传递的一个对象，里面的所有属性 TestCtrl 就全收下了，接着执行 here=here+1 ，于是我们会在屏幕上看到 6 。

这里是一个上下文交错的环境，通过 & 这种机制，让指令的 scope 与节点的 scope 发生了互动。真是鬼斧神工的设计。而实现它，只用了几行代码：

case '&': {

parentGet = $parse(attrs[attrName]);

scope[scopeName] = function(locals) {

return parentGet(parentScope, locals);

}

break;

}

再看 controller 这个参数。这个参数的作用是提供一个 controller 的构造函数，它会在 compile函数之后， link 函数之前被执行。

<a>haha</a>

1 var app = angular.module('Demo', [], angular.noop);

2

3 app.directive('a', function(){

4 var func = function(){

5 console.log('compile');

6 return function(){

7 console.log('link');

8 }

9 }

10

11 var controller = function($scope, $element, $attrs, $transclude){

12 console.log('controller');

13 console.log($scope);

14

15 var node = $transclude(function(clone\_element, scope){

16 console.log(clone\_element);

17 console.log('--');

18 console.log(scope);

19 });

20 console.log(node);

21 }

22

23 return {compile: func,

24 controller: controller,

25 transclude: true,

26 restrict: 'E'}

27 });

controller 的最后一个参数， $transclude ，是一个只接受 cloneAttachFn 作为参数的一个函数。

按官方的说法，这个机制的设计目的是为了让各个指令之间可以互相通信。参考普通节点的处理方式，这里也是处理指令 scope 的合适位置。

<a b>kk</a>

1 var app = angular.module('Demo', [], angular.noop);

2

3 app.directive('a', function(){

4 var func = function(){

5 }

6

7 var controller = function($scope, $element, $attrs, $transclude){

8 console.log('a');

9 this.a = 'xx';

10 }

11

12 return {compile: func,

13 name: 'not\_a',

14 controller: controller,

15 restrict: 'E'}

16 });

17

18 app.directive('b', function(){

19 var func = function(){

20 return function($scope, $element, $attrs, $controller){

21 console.log($controller);

22 }

23 }

24

25 var controller = function($scope, $element, $attrs, $transclude){

26 console.log('b');

27 }

28

29 return {compile: func,

30 controller: controller,

31 require: 'not\_a',

32 restrict: 'EA'}

33 });

name 参数在这里可以用以为 controller 重起一个名字，以方便在 require 参数中引用。

require 参数可以带两种前缀（可以同时使用）：

* ? ，如果指定的 controller 不存在，则忽略错误。即：
* require: '?not\_b'

如果名为 not\_b 的 controller 不存在时，不会直接抛出错误， link 函数中对应的$controller 为 undefined 。

* ^ ，同时在父级节点中寻找指定的 controller ，把上面的例子小改一下：
* <a><b>kk</b></a>

把 a 的 require 改成（否则就找不到 not\_a 这个 controller ）：

require: '?^not\_a'

还剩下几个模板参数：

template 模板内容，这个内容会根据 replace 参数的设置替换节点或只替换节点内容。

templateUrl 模板内容，获取方式是异步请求。

replace 设置如何处理模板内容。为 true 时为替换掉指令节点，否则只替换到节点内容。

<div ng-controller="TestCtrl">

<h1 a>原始内容</h1>

</div>

var app = angular.module('Demo', [], angular.noop);

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

var func = function(){

}

return {compile: func,

template: '<p>标题 {{ name }} <button ng-click="name=\'hahaha\'">修改</button></p>',

//replace: true,

//controller: function($scope){$scope.name = 'xxx'},

//scope: {},

scope: true ,

controller: function($scope){console.log($scope)},

restrict: 'A'}

});

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

$scope.name = '123';

console.log($scope);

});

template 中可以包括变量引用的表达式，其 scope 遵寻 scope 参数的作用（可能受继承关系影响）。

templateUrl 是异步请求模板内容，并且是获取到内容之后才开始执行指令的 compile 函数。

最后说一个 compile 这个参数。它除了可以返回一个函数用为 link 函数之外，还可以返回一个对象，这个对象能包括两个成员，一个 pre ，一个 post 。实际上， link 函数是由两部分组成，所谓的 preLink 和 postLink 。区别在于执行顺序，特别是在指令层级嵌套的结构之下，postLink 是在所有的子级指令 link 完成之后才最后执行的。 compile 如果只返回一个函数，则这个函数被作为 postLink 使用：

<a><b></b></a>

1 var app = angular.module('Demo', [], angular.noop);

2

3 app.directive('a', function(){

4 var func = function(){

5 console.log('a compile');

6 return {

7 pre: function(){console.log('a link pre')},

8 post: function(){console.log('a link post')},

9 }

10 }

11

12 return {compile: func,

13 restrict: 'E'}

14 });

15

16 app.directive('b', function(){

17 var func = function(){

18 console.log('b compile');

19 return {

20 pre: function(){console.log('b link pre')},

21 post: function(){console.log('b link post')},

22 }

23 }

24

25 return {compile: func,

26 restrict: 'E'}

27 });

#### 18.9. Attributes的细节

节点属性被包装之后会传给 compile 和 link 函数。从这个操作中，我们可以得到节点的引用，可以操作节点属性，也可以为节点属性注册侦听事件。

<test a="1" b c="xxx"></test>

var app = angular.module('Demo', [], angular.noop);

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

var func = function($element, $attrs){

console.log($attrs);

}

return {compile: func,

restrict: 'E'}

整个 Attributes 对象是比较简单的，它的成员包括了：

$$element 属性所在的节点。

$attr 所有的属性值（类型是对象）。

$normalize 一个名字标准化的工具函数，可以把 ng-click 变成 ngClick 。

$observe 为属性注册侦听器的函数。

$set 设置对象属性，及节点属性的工具。

除了上面这些成员，对象的成员还包括所有属性的名字。

先看 $observe 的使用，基本上相当于 $scope 中的 $watch ：

<div ng-controller="TestCtrl">

<test a="{{ a }}" b c="xxx"></test>

<button ng-click="a=a+1">修改</button>

</div>

var app = angular.module('Demo', [], angular.noop);

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

var func = function($element, $attrs){

console.log($attrs);

$attrs.$observe('a', function(new\_v){

console.log(new\_v);

});

}

return {compile: func,

restrict: 'E'}

});

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

$scope.a = 123;

});

$set 方法的定义是： function(key, value, writeAttr, attrName) { ... } 。

* key 对象的成员名。
* value 需要设置的值。
* writeAttr 是否同时修改 DOM 节点的属性（注意区别“节点”与“对象”），默认为true 。
* attrName 实际的属性名，与“标准化”之后的属性名有区别。

<div ng-controller="TestCtrl">

<test a="1" ys-a="123" ng-click="show(1)">这里</test>

</div>

var app = angular.module('Demo', [], angular.noop);

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

var func = function($element, $attrs){

$attrs.$set('b', 'ooo');

$attrs.$set('a-b', '11');

$attrs.$set('c-d', '11', true, 'c\_d');

console.log($attrs);

}

return {compile: func,

restrict: 'E'}

});

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

$scope.show = function(v){console.log(v);}

});

从例子中可以看到，原始的节点属性值对，放到对象中之后，名字一定是“标准化”之后的。但是手动 $set 的新属性，不会自动做标准化处理。

#### 18.10. 预定义的 NgModelController

在前面讲 conroller 参数的时候，提到过可以为指令定义一个 conroller 。官方的实现中，有很多已定义的指令，这些指令当中，有两个已定义的 conroller ，它们是 NgModelController 和FormController ，对应 ng-model 和 form 这两个指令（可以参照前面的“表单控件”一章）。

在使用中，除了可以通过 $scope 来取得它们的引用之外，也可以在自定义指令中通过 require参数直接引用，这样就可以在 link 函数中使用 controller 去实现一些功能。

先看 NgModelController 。这东西的作用有两个，一是控制 ViewValue 与 ModelValue 之间的转换关系（你可以实现看到的是一个值，但是存到变量里变成了另外一个值），二是与FormController 配合做数据校验的相关逻辑。

先看两个应该是最有用的属性：

$formatters 是一个由函数组成的列表，串行执行，作用是把变量值变成显示的值。

$parsers 与上面的方向相反，把显示的值变成变量值。

假设我们在变量中要保存一个列表的类型，但是显示的东西只能是字符串，所以这两者之间需要一个转换：

<div ng-controller="TestCtrl">

<input type="text" ng-model="a" test />

<button ng-click="show(a)">查看</button>

</div>

1 var app = angular.module('Demo', [], angular.noop);

2

3 app.directive('test', function(){

4 var link = function($scope, $element, $attrs, $ctrl){

5

6 $ctrl.$formatters.push(function(value){

7 return value.join(',');

8 });

9

10 $ctrl.$parsers.push(function(value){

11 return value.split(',');

12 });

13 }

14

15 return {compile: function(){return link},

16 require: 'ngModel',

17 restrict: 'A'}

18 });

19

20 app.controller('TestCtrl', function($scope){

21 $scope.a = [];

22 //$scope.a = [1,2,3];

23 $scope.show = function(v){

24 console.log(v);

25 }

26 });

上面在定义 test 这个指令， require 参数指定了 ngModel 。同时因为 DOM 结构， ng-model 是存在的。于是， link 函数中就可以获取到一个 NgModelController 的实例，即代码中的 $ctrl 。

我们添加了需要的过滤函数：

* 从变量( ModelValue )到显示值( ViewValue )的过程， $formatters 属性，把一个列表变成一个字符串。
* 从显示值到变量的过程， $parsers 属性，把一个字符串变成一个列表。

对于显示值和变量，还有其它的 API ，这里就不细说了。

另一部分，是关于数据校验的，放到下一章同 FormController 一起讨论。

#### 18.11. 预定义的 FormController

前面的“表单控制”那章，实际上讲的就是 FormController ，只是那里是从 scope 中获取到的引用。现在从指令定义的角度，来更清楚地了解 FormController 及 NgModelController 是如何配合工作的。

先说一下， form 和 ngForm 是官方定义的两个指令，但是它们其实是同一个东西。前者只允许以标签形式使用，而后者允许 EAC 的形式。DOM 结构中， form 标签不能嵌套，但是 ng 的指令没有这个限制。不管是 form 还是 ngForm ，它们的 controller 都被命名成了 form 。 所以 require 这个参数不要写错了。

FormController 的几个成员是很好理解的：

$pristine 表单是否被动过

$dirty 表单是否没被动过

$valid 表单是否检验通过

$invalid 表单是否检验未通过

$error 表单中的错误

$setDirty() 直接设置 $dirty 及 $pristine

<div ng-controller="TestCtrl">

<div ng-form test>

<input ng-model="a" type="email" />

<button ng-click="do()">查看</button>

</div>

</div>

var app = angular.module('Demo', [], angular.noop);

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

var link = function($scope, $element, $attrs, $ctrl){

$scope.do = function(){

//$ctrl.$setDirty();

console.log($ctrl.$pristine); //form是否没被动过

console.log($ctrl.$dirty); //form是否被动过

console.log($ctrl.$valid); //form是否被检验通过

console.log($ctrl.$invalid); //form是否有错误

console.log($ctrl.$error); //form中有错误的字段

}

}

return {compile: function(){return link},

require: 'form',

restrict: 'A'}

});

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

});

$error 这个属性，是一个对象， key 是错误名， value 部分是一个列表，其成员是对应的NgModelController 的实例。

FormController 可以自由增减它包含的那些，类似于 NgModelController 的实例。在 DOM 结构上，有 ng-model 的 input 节点的 NgMoelController 会被自动添加。

$addControl() 添加一个 conroller

$removeControl() 删除一个 controller

这两个手动使用机会应该不会很多。被添加的实例也可以手动实现所有的 NgModelController的方法

<div ng-controller="TestCtrl">

<bb />

<div ng-form test>

<input ng-model="a" type="email" />

<button ng-click="add()">添加</button>

</div>

</div>

1 var app = angular.module('Demo', [], angular.noop);

2

3 app.directive('test', function(){

4 var link = function($scope, $element, $attrs, $ctrl){

5 $scope.add = function(){

6 $ctrl.$addControl($scope.bb);

7 console.log($ctrl);

8 }

9 }

10

11 return {compile: function(){return link},

12 require: 'form',

13 restrict: 'A'}

14 });

15

16 app.directive('bb', function(){

17 var controller = function($scope, $element, $attrs, $transclude){

18 $scope.bb = this;

19 this.$name = 'bb';

20 }

21

22 return {compile: angular.noop,

23 restrict: 'E',

24 controller: controller}

25 });

26

27 app.controller('TestCtrl', function($scope){

28 });

整合 FormController 和 NgModelController 就很容易扩展各种类型的字段:

<div ng-controller="TestCtrl">

<form name="f">

<input type="my" ng-model="a" />

<button ng-click="show()">查看</button>

</form>

</div>

1 var app = angular.module('Demo', [], angular.noop);

2

3 app.directive('input', function(){

4 var link = function($scope, $element, $attrs, $ctrl){

5 console.log($attrs.type);

6 var validator = function(v){

7 if(v == '123'){

8 $ctrl.$setValidity('my', true);

9 return v;

10 } else {

11 $ctrl.$setValidity('my', false);

12 return undefined;

13 }

14 }

15

16 $ctrl.$formatters.push(validator);

17 $ctrl.$parsers.push(validator);

18 }

19

20 return {compile: function(){return link},

21 require: 'ngModel',

22 restrict: 'E'}

23 });

24

25 app.controller('TestCtrl', function($scope){

26 $scope.show = function(){

27 console.log($scope.f);

28 }

29 });

虽然官方原来定义了几种 type ，但这不妨碍我们继续扩展新的类型。如果新的 type 参数值不在官方的定义列表里，那会按 text 类型先做处理，这其实什么影响都没有。剩下的，就是写我们自己的验证逻辑就行了。

上面的代码是参见官方的做法，使用格式化的过程，同时在里面做有效性检查。

#### 18.12. 示例：文本框

这个例子与官网上的那个例子相似。最终是要显示一个文本框，这个文本框由标题和内容两部分组成。而且标题和内容则是引用 controller 中的变量值。

HTML 部分的代码：

<div ng-controller="TestCtrl">

<ys-block title="title" text="text"></ys-block>

<p>标题: <input ng-model="title" /></p>

<p>内容: <input ng-model="text" /></p>

<ys-block title="title" text="text"></ys-block>

</div>

从这个期望实现效果的 HTML 代码中，我们可以考虑设计指令的实现方式：

* 这个指令的使用方式是“标签”， 即 restrict 这个参数应该设置为 E 。
* 节点的属性值是对 controller 变量的引用，那么我们应该在指令的 scope 中使用 =的方式来指定成员值。
* 最终的效果显示需要进行 DOM 结构的重构，那直接使用 template 就好了。
* 自定义的标签在最终效果中是多余的，所有 replace 应该设置为 true 。

JS 部分的代码：

var app = angular.module('Demo', [], angular.noop);

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

return {compile: angular.noop,

template: '<div style="width: 200px; border: 1px solid black;"><h1 style="background-color: gray; color: white; font-size: 22px;">{{ title }}</h1><div>{{ text }}</div></div>',

replace: true,

scope: {title: '=title', text: '=text'},

restrict: 'E'};

});

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

$scope.title = '标题在这里';

$scope.text = '内容在这里';

});

angular.bootstrap(document, ['Demo']);

可以看到，这种简单的组件式指令，只需要作 DOM 结构的变换即可实现，连 compile 函数都不需要写。

#### 18.13. 示例：模板控制语句 for

这个示例尝试实现一个重复语句，功能同官方的 ngRepeat ，但是使用方式类似于我们通常编程语言中的 for 语句：

<div ng-controller="TestCtrl" ng-init="obj\_list=[1,2,3,4]; name='name'">

<ul>

<for o in obj\_list>

<li>{{ o }}, {{ name }}</li>

</for>

</ul>

<button ng-click="obj\_list=[1,2]; name='o?'">修改</button>

</div>

同样，我们从上面的使用方式去考虑这个指令的实现：

* 这是一个完全的控制指令，所以单个节点应该只有它一个指令起作用就好了，于是权重要比较高，并且“到此为止”—— priority 设置为 1000 ， terminal 设置为 true。
* 使用时的语法问题。事实上浏览器会把 for 节点补充成一个正确的 HTML 结构，即里面的属性都会变成类似 o="" 这样。我们通过节点的 outerHTML 属性取到字符串并解析取得需要的信息。
* 我们把 for 节点之间的内容作为一个模板，并且通过循环多次渲染该模板之后把结果填充到合适的位置。
* 在处理上面的那个模板时，需要不断地创建新 scope 的，并且 o 这个成员需要单独赋值。

注意：这里只是简单实现功能。官方的那个 ngRepeat 比较复杂，是做了专门的算法优化的。当然，这里的实现也可以是简单把 DOM 结构变成使用 ngRepeat 的形式 :)

JS 部分代码：

1 var app = angular.module('Demo', [], angular.noop);

2

3 app.directive('for', function($compile){

4 var compile = function($element, $attrs, $link){

5 var match = $element[0].outerHTML.match('<for (.\*?)=.\*? in=.\*? (.\*?)=.\*?>');

6 if(!match || match.length != 3){throw Error('syntax: <for o in obj\_list>')}

7 var iter = match[1];

8 var list = match[2];

9 var tpl = $compile($.trim($element.html()));

10 $element.empty();

11

12 var link = function($scope, $ielement, $iattrs, $controller){

13

14 var new\_node = [];

15

16 $scope.$watch(list, function(list){

17 angular.forEach(new\_node, function(n){n.remove()});

18 var scp, inode;

19 for(var i = 0, ii = list.length; i < ii; i++){

20 scp = $scope.$new();

21 scp[iter] = list[i];

22 inode = tpl(scp, angular.noop);

23 $ielement.before(inode);

24 new\_node.push(inode);

25 }

26

27 });

28 }

29

30 return link;

31 }

32 return {compile: compile,

33 priority: 1000,

34 terminal: true,

35 restrict: 'E'};

36 });

37

38 app.controller('TestCtrl', angular.noop);

39 angular.bootstrap(document, ['Demo']);

#### 18.14. 示例：模板控制语句 if/else

这个示例是尝试实现：

<div ng-controller="TestCtrl">

<if true="a == 1">

<p>判断为真, {{ name }}</p>

<else>

<p>判断为假, {{ name }}</p>

</else>

</if>

<div>

<p>a: <input ng-model="a" /></p>

<p>name: <input ng-model="name" /></p>

</div>

</div>

考虑实现的思路：

* else 与 if 是两个指令，它们是父子关系。通过 scope 可以联系起来。至于 scope是在 link 中处理还是 controller 中处理并不重要。
* true 属性的条件判断通过 $parse 服务很容易实现。
* 如果最终效果要去掉 if 节点，我们可以使用注释节点来“占位”。

JS 代码：

1 var app = angular.module('Demo', [], angular.noop);

2

3 app.directive('if', function($parse, $compile){

4 var compile = function($element, $attrs){

5 var cond = $parse($attrs.true);

6

7 var link = function($scope, $ielement, $iattrs, $controller){

8 $scope.if\_node = $compile($.trim($ielement.html()))($scope, angular.noop);

9 $ielement.empty();

10 var mark = $('<!-- IF/ELSE -->');

11 $element.before(mark);

12 $element.remove();

13

14 $scope.$watch(function(scope){

15 if(cond(scope)){

16 mark.after($scope.if\_node);

17 $scope.else\_node.detach();

18 } else {

19 if($scope.else\_node !== undefined){

20 mark.after($scope.else\_node);

21 $scope.if\_node.detach();

22 }

23 }

24 });

25 }

26 return link;

27 }

28

29 return {compile: compile,

30 scope: true,

31 restrict: 'E'}

32 });

33

34 app.directive('else', function($compile){

35 var compile = function($element, $attrs){

36

37 var link = function($scope, $ielement, $iattrs, $controller){

38 $scope.else\_node = $compile($.trim($ielement.html()))($scope, angular.noop);

39 $element.remove();

40 }

41 return link;

42 }

43

44 return {compile: compile,

45 restrict: 'E'}

46 });

47

48 app.controller('TestCtrl', function($scope){

49 $scope.a = 1;

50 });

51

52 angular.bootstrap(document, ['Demo']);

代码中注意一点，就是 if\_node 在得到之时，就已经是做了变量绑定的了。错误的思路是，在$watch 中再去不断地得到新的 if\_node 。

## Bootstrap

### 什么是 Bootstrap？

Bootstrap 是一个用于快速开发 Web 应用程序和网站的前端框架。Bootstrap 是基于 HTML、CSS、JAVASCRIPT 的。

### 历史

Bootstrap 是由 *Twitter* 的 *Mark Otto* 和 *Jacob Thornton* 开发的。Bootstrap 是 2011 年八月在 GitHub 上发布的开源产品。

### 为什么使用 Bootstrap？

* **移动设备优先**：自 Bootstrap 3 起，框架包含了贯穿于整个库的移动设备优先的样式。
* **浏览器支持**：所有的主流浏览器都支持 Bootstrap。

Internet Explorer Firefox Opera Google Chrome Safari

* **容易上手**：只要您具备 HTML 和 CSS 的基础知识，您就可以开始学习 Bootstrap。
* **响应式设计**：Bootstrap 的响应式 CSS 能够自适应于台式机、平板电脑和手机。更多有关响应式设计的内容详见 [Bootstrap 响应式设计](http://www.runoob.com/bootstrap/bootstrap-responsive-design.html)。



* 它为开发人员创建接口提供了一个简洁统一的解决方案。
* 它包含了功能强大的内置组件，易于定制。
* 它还提供了基于 Web 的定制。
* 它是开源的。

### Bootstrap 包的内容

* **基本结构**：Bootstrap 提供了一个带有网格系统、链接样式、背景的基本结构。这将在 **Bootstrap 基本结构** 部分详细讲解。
* **CSS**：Bootstrap 自带以下特性：全局的 CSS 设置、定义基本的 HTML 元素样式、可扩展的 class，以及一个先进的网格系统。这将在 **Bootstrap CSS** 部分详细讲解。
* **组件**：Bootstrap 包含了十几个可重用的组件，用于创建图像、下拉菜单、导航、警告框、弹出框等等。这将在 **布局组件** 部分详细讲解。
* **JavaScript 插件**：Bootstrap 包含了十几个自定义的 jQuery 插件。您可以直接包含所有的插件，也可以逐个包含这些插件。这将在 **Bootstrap 插件** 部分详细讲解。
* **定制**：您可以定制 Bootstrap 的组件、LESS 变量和 jQuery 插件来得到您自己的版本。

### 学习网站

<http://www.runoob.com/bootstrap/bootstrap-tutorial.html>

# 实例网站项目结构

## 技术结构

技术框架.emf

图中展现了项目中用到的主要技术及技术结构。

项目是Maven项目，使用Maven来管理所有依赖的包。

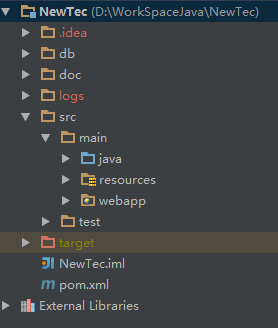
前端UI层：

* AngularJS，构建前端单页引用，路由控制页面加载
* Bootstrap,前端展现样式
* Angular-shiro,用于权限控制

项目整体使用SpringMVC进行控制，由于使用了AngularJS做单页应用,这里的SpringMVC弱化了对页面的控制，controller主要用来封装给页面调用的Restful服务。同时，使用Shiro对服务进行权限控制，使用Mongo访问非关系型数据库MongoDB,ORM使用Hibernate来访问关系型数据库MySQL。

同时，项目整体使用log4j做日志记录，使用ehcache做缓存。

## 代码结构



db:项目初始化需要的创建的表和初始数据的sql文件

doc:项目相关的参考文档

pom.xml:maven用来管理项目所用到的所有依赖包

src:项目主要代码

src/main:主要代码

src/test:测试相关代码，目前没有写测试用例

src/main/java:项目所有java代码，根据具体业务逻辑分成不同包，每个包中结构根据SpringMVC结构划分为controller,dao,dto,entity,service等包

src/main/resources:项目所有的配置文件，SpringMVC配置，hibernate配置，日志配置，缓存配置等等。

src/main/webapp：项目所有前端界面，目前内部结构还有点混乱