Chapter 1. Java EE 应用和开发环境

1、对一个企业而言，选择 Java EE 构建信息化平台，更体现了一种长远的规划：企业的信息化是不断整合的过程，在未来的日子里，经常会有不同平台，不同的异构系统需要整合，Java EE 应用提供的跨平台性、开放性、及各种远程访问的技术，为异构系统的良好整合提供了保证

1.1. Java EE 应用概述

1、经典的 Java EE 应用往往以 EJB(企业级 Java Bean)为核心，以应用服务器为运行环境，所以通常开发、运行成本较高

2、本书介绍的轻量级 Java EE 应用具备了 Java EE 规范的种种特征，例如面向对象建模的思维方式、优秀的应用分层级良好的可扩展性、可维护性

3、轻量级 Java EE 应用保留了经典 Java 应用的构架，但开发、运行成本更低

1.1.1. Java EE 应用的分层模型

1、无论是经典的 Java EE 架构，还是本书介绍的轻量级 Java EE 架构，大致上可以分为如下几层

- **Domian Object(领域对象)层**：此层由一系列 OPJO(Plain Old Java Object，普通的、传统的 Java 对象)组成，这些对象是该系统的 Domian Object，往往包含了各自所需实现的业务逻辑方法
- **DAO(Data Access Object，数据访问对象)层**：此层由一系列的 DAO 组件组成，这些 DAO 实现了对数据库的创建、查询、更新和删除(CRUD)等原子操作
  - 在经典的 JAVA EE 应用中，DAO 层也被称为 EAO 层，EAO 层组件的作用域 DAO 层组件的作用基本相似，只是 EAO 层主要完成对实体(Entity)的 CRUD 操作，因此简称为 EAO 层
- **业务逻辑层**：此层由一系列的业务逻辑对象组成，这些业务逻辑对象实现了系统所需的业务逻辑方法。这些业务逻辑方法可能仅仅暴露 Domian Object 对象所实现的业务逻辑方法，也可能是依赖 DAO 组件实现的业务逻辑方法
- **控制器层**：此层由一系列控制器组成，这些控制器用于拦截用户请求，并调用业务逻辑组件的业务逻辑方法，处理用户请求，并根据处理结果转发到不同的表现层组件
- 各层的 Java EE 组件以松耦合的方式耦合在一起，各组件并不以硬编码方式耦合，这种方式是为了应用以后的扩展性

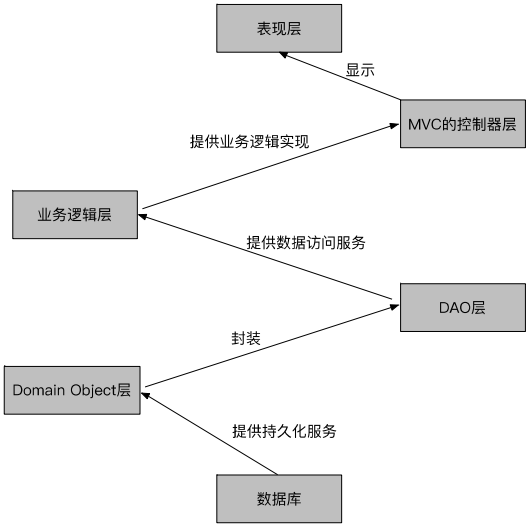

图 11 Java EE 应用架构

1.1.2. Java EE 应用的组件

1、Java EE 应用提供了系统架构上的飞跃，Java EE 架构提供了良好的分离，隔离了各组件之间的代码依赖

2、Java EE 应用大致包括如下几类组件

- **表现层组件**：主要负责收集用户输入数据，或者向客户显示系统状态
  - 最常用的表现层技术是 JSP，但 JSP 并不是唯一的表现层技术
  - 表现层还可由 Velocity、FreeMarker 和 Tapestry 等技术完成
- **控制器组件**：对于 Java EE 的 MVC 框架而言，框架提供一个前端核心控制器，而核心控制器负责拦截用户请求，并将其请求转发给用户实现的控制器组件，而这些用户实现的控制器则负责处理调用业务逻辑方法，处理用户请求
- **业务逻辑组件**：是系统的核心组件，实现系统的业务逻辑，通常一个业务逻辑方法对应一次用户操作，一个业务逻辑方法应该是一个整体，因此要求对业务逻辑方法增加事务性
  - 业务逻辑方法仅仅负责实现业务逻辑，不应该进行数据库访问，因此业务逻辑组件中不应该出现原始的 Hibernate、JDBC 等 API
- **DAO 组件：Data Access Object**，也被称为数据访问对象，这个类型的对象比较缺乏变化，每个 DAO 组件都提供 Domain Object 对象基本的创建、查询、更新和删除等操作，这些操作对应于数据表的 CRUD(创建，查询，更新，删除)等原子操作
  - 采用不同的持久层访问技术，DAO 组件的实现会完全不同
  - 为了业务逻辑组件的实现与 DAO 组件的实现分离，程序应该为每个 DAO 组件都提供接口，业务逻辑组件面向 DAO 接口编程，这样才能提供更好的解耦
- **领域对象组件**：领域对象(Domain Object)抽象了系统的对象模型
  - 通常而言，这些领域对象的状态都必须保存在数据库里
  - 因此每个领域对象通常对应一个或多个数据表，领域对象通常需要提供对数据记录访问方式

1.1.3. Java EE 应用的结构和优势

1、对于 Java EE 的初学者而言，常常有一个问题：明明可以使用 JSP 完成这个系统，为什么还要使用 Hibernate 等技术?难道仅仅是为了听起来高深一点?明明可以使用纯粹的 JSP 完成整个系统，为什么还要将系统分层

- 要回答这些问题，就不能仅仅考虑系统开发过程，还需要考虑系统后期的维护、扩展
- 而且不能仅仅考虑哪些小型系统，还要考虑大型系统的协同开发
- 对于个人学习、娱乐的个人站点，的确没有必要使用复杂的 Java EE 应用框架，采用纯粹的 JSP 就可以实现整个系统

2、对于大型的信息化系统而言，采用 Java EE 应用架构则有很大的优势

- 软件不是一次性系统，不仅与传统行业的产品有较大的差异，甚至与硬件产品也有较大的差异，硬件产品可以随时间的流逝而宣布过时，更换新一代硬件产品；软件不能彻底替换，只能在其原来的基础上延伸，因为软件往往是信息的延续，是企业命脉的延伸；如果支撑企业系统的软件不具备可扩展性，当企业平台发生改变时，如何面对这种改变?
- 对于信息化系统，前期开发工作对整个系统工作量而言，仅仅是小部分，而后期的维护、升级往往占更大的比重；更极端的情况是，可能在前期开发期间，企业需求已经发生改变，这种改变时客观的，而软件系统必须适应这种改变，这要求软件系统具有很好的伸缩性
- 最理想的软件系统应如同计算机的硬件系统，各种设备可以支持热插拔，各设备之间的影响非常小，设备与设备之间的实现完全透明，只要有通用的接口，设备之间就可以良好协作

3、致力于让应用的各组件以松耦合的方式组织在一起，让应用之间的耦合停留在接口层次，而不是代码层次

1.1.4. 常用的 Java EE 服务器

1、本书介绍一种优秀的轻量级 Java EE 架构：Struts2+Spring+Hibernate。采用这种架构的软件系统，无需专业的 Java EE 服务器支持，只需要简单的 Web 服务器就可以运行，Java 领域常见的 Web 服务器都是开源的，而且具有很好的稳定性

2、常见的 Web 服务器

- Tomcat：Tomcat 和 Java 结合的最好，是 Oracle 官方推荐的 JSP 服务器。Tomcat 是开源的 Web 服务器，经过长时间的发展，性能、稳定性等方面都非常优秀
- Jetty：另一个优秀的 Web 服务器。Jetty 有个更大的优点就是，Jetty 可作为一个嵌入式服务器，即：如果在应用中加入 Jetty 的 JAR，应用可在代码中对外提供 Web 服务
- Resin：目前最快的 JSP、Servlet 运行平台，支持 EJB。个人学习该服务器是免费的，但如果想将该服务器用作商业用途，则需要交纳相应的费用

3、常用的 Java EE 服务器

- JBoss：开源的 Java EE 服务器，全面支持各种最新的 Java EE 规范
- GlassFish：Oracle 官方提供的 Java EE 服务器，通常能最早支持各种 Java EE 规范，比如最新的 GlassFish 可以支持目前最先的 Java EE7
- WebLogic 和 WebSphere：这是两个专业的商用 Java EE 服务器，价格不菲，

性能相当出色

4、对于轻量级 Java EE 而言，没有必要使用 Java EE 服务器，使用简单的 Web 容器已经完全能胜任

1.2. 轻量级 Java EE 应用相关技术

1、轻量级 Java EE 应用以传统的 JSP 作为表现层技术，以一系列开源框架作为 MVC 层、中间层、持久层解决方案，并将这些开源框架有机地组合在一起，使得 Java EE 应用具有高度的可扩展性、可维护性

1.2.1. JSP、Servlet 3.x 和 JavaBean 及替代技术

1、JSP 是最早的 Java EE 规范之一，也是最经典的 Java EE 技术之一，直到今天，JSP 依然广泛地应用于各种 Java EE 应用中，充当 Java EE 应用的表现层角色

2、JSP 具有简单、易用的特点，JSP 的学习路线平坦，而且国内有大量 JSP 学习资料，所以大部分 Java 学习者学习 Java EE 开发都会选择从 JSP 开始

3、Servlet 和 JSP 其实是完全统一的，二者在底层的运行原理是完全一样的

- 实际上，JSP 必须被 Web 服务器编译成 Servlet，真正在 Web 服务器内运行的是 Servlet
- 从这个意义上来说，JSP 相当于是一个"草稿"文件，Web 服务器根据该"草稿"文件来生成 Servlet，真正提供 HTTP 服务的是 Servlet，因此广义的 Servlet 包含了 JSP 和 Servlet

4、就目前的 Java EE 应用来看，纯粹的 Servlet 已经很少使用，毕竟 Servlet 开发成本太高，而且使用 Servlet 充当表现层将充当表现层页面难以维护，不利于美工人员参与 Servlet 开发，所以实际开发中大都使用 JSP 充当表现层技术

5、Servlet 3.x 规范的出现，再次为 Java Web 开发带来了巨大的便捷，Servlet 3.x 提供了异步请求、注解、增强的 Servlet API、非阻塞 IO，这些功能都很好地简化了 Java Web 开发

6、JSP 只负责简单的显示逻辑，所以 JSP 无法直接访问应用的底层状态，Java EE 应用会选择使用 JavaBean 来传输数据，在严格的 Java EE 应用中，中间层的组件会将应用底层的状态信息封装成 JavaBean 集，这些 JavaBean 被称为 DTO(Data Transfer Object，数据传输对象)，并将这些 DTO 集传到 JSP 页面，从而让 JSP 可以显示应用的底层状态。

7、目前阶段，Java EE 应用除了可以使用 JSP 作为表现层技术之外，还可以使用 FreeMarker 或 Velocity 充当表现层技术，这些表现层技术更加纯粹，使用更加简捷，完全可以作为 JSP 的替代

1.2.2. Struts2.3 及替代技术

1、Struts 是全世界最早的 MVC(Model View Controller)框架，其作者是 JSP 规范的制定者，并参与了 Tomcat 开发，所以 Struts 从诞生的第一天起，就备受 Java EE 应用开发者的青睐

2、Struts 框架学习简单，而且是全世界应用最方便的 MVC 框架

3、Struts 框架毕竟太老了，无数设计上的硬伤使得该框架难以胜任更复杂的需求，于是古老的 Struts 结合了另一个优秀的 MVC 框架：WebWork，分娩出了全新的 Struts2，Struts2 拥有众多优秀的设计，而且吸收了传统的 Struts 和

WebWork 两者的精华，迅速成为了 MVC 框架中新的王者

4、在 MVC 框架领域还有两个替代者：Spring MVC 和 JSF

- Spring MVC 是 Spring 框架所提供的开源 MVC 框架，由于 Spring 框架拥有极高的市场占有率，因此导致 Spring MVC 也拥有不错的市场表现
- JSF 是 Oracle 所推荐的 Java EE 规范，拥有最纯正的血统，而且 Apache 也为 JSF 提供了 MyFaces 实现，这使得 JSF 具有很大的吸引力
- 从设计上来看，JSF 比 Struts 理念更加优秀，采用的是传统的 RAD(快速应用开发)理念，只是 Struts 早就深入人心，导致 JSF 在市场占有率上略逊一筹

1.2.3. Hibernate 4.3 及替代技术

1、传统的 Java 应用都是采用 JDBC 来访问数据库的，但传统的 JDBC 采用的是一种基于 SQL 的操作方式，这种操作方式与 Java 语言的面向对象特征不太一样，所以 Java EE 应用需要一种技术，通过这种技术能让以面向对象的方式操作关系数据库

2、这种特殊的技术就是 ORM(Object Relation Mapping)，最早的 ORM 是 Entity EJB(Enterprise JavaBean)，EJB 就是经典 Java EE 应用的核心

3、但是从 EJB 1.0 到 EJB 2.0，许多人觉得 EJB 非常繁琐，所以导致 EJB 被收诟病。在这种背景下，Hibernate 框架应运而生，Hibernate 框架是一种开源的、轻量级的 ORM 框架，**它允许将普通的、传统的 Java 对象(POJO)映射成持久化类，允许应用程序以面向对象的方式来操作 POJO，而 Hibernate 框架则负责将这种操作转换为底层的 SQL 操作**

4、后来，Sun 公司果断地抛弃了 EJB 2.X 规范，引入了 JPA(Java Persistence API)规范。JPA 规范是一种 ORM 规范，因此它底层可以使用 Hibernate、TopLink 等任意一种 ORM 框架作为实现

- 如果应用程序面向 JPA 编程，将可以让应用程序既可利用 Hibernate 的持久化技术---因为可以用 Hibernate 作为实现；也可让应用程序保持较好的可扩展性---因为可以再各种 ORM 技术之间自由切换

5、除了可以用 Hibernate 这种 ORM 框架之外，轻量级 Java EE 应用通常还可以选择 MyBatis 框架作为持久层框架，**MyBatis 是 Apache 组织提供的另一个轻量级持久层框架，MyBatis 允许将 SQL 语句查询结果映射成对象，因此常常也将 MyBatis 称为 SQL Mapping 工具**

6、除此之外，Oracle 的 TopLink、Apache 的 OJB 都可以作为 Hibernate 的替代方案

1.2.4. Spring4.0 及替代技术

1、Spring 只是抽象了大量 Java EE 应用中的常用代码，将它们抽象成一个框架，通过使用 Spring 可以大幅度地提高开发效率，并可以保证整个应用具有良好的设计

2、Spring 充满各种设计模式的应用，例如**单例模式、工厂模式、抽象工厂模式、命令模式、职责连模式、代理模式**等

3、Spring 号称是 Java EE 应用的一站式解决方案，**Spring 本身提供了一个设计优良的 MVC 框架：Spring MVC，使用 Spring 框架可以直接使用该 MVC 框架。但**

实际上，Spring 并未提供完整的持久层框架---这可以理解成一种"空"，但这种"空"正式 Spring 框架的魅力所在---Spring 能与大部分持久层框架无缝整合

4、Spring 向上可以与 MVC 框架无缝整合，向下可以与各种持久层框架无缝整合，具有强大的生命力

5、轻量级 Java EE 这个概念也是由 Spring 框架衍生出来的，Spring 框架暂时没有较好的替代框架

6、为什么需要框架?用 JSP 和 Servlet 已经足够了?

- 提出这些疑问的人通常还未真正进入企业开发，或从未开发一个真正的项目
- 真实的企业引用开发有**两个重要的关注点：可维护性**和**复用**
- 对于信息化系统而言，总有一些开发过程是重复的，为什么不将这些重复开发工作抽象成基础类库，这种抽象提高了开发效率，而且因为重复使用，也降低了引入错误的风险
- 因此只要是一个由实际开发经验的软件公司，就一定会有自己的一套基础类库，这就是需要使用框架的原因
- 从某个角度来看，框架也是一套基础类库，它抽象了软件开发的通用步骤，让实际开发人员可以直接利用这部分实现
- 一个从事实际开发的软件公司，不管他是否意识到，他已经在使用框架，区别只有：使用的框架是别人提供的还是自己抽象出来的
- 通常第三方框架更稳定，更有保证，因为第三方框架往往经过了更多人的测试，而使用自己抽象的框架则更加熟悉底层运行原理，出了问题更好把握

1.3. Tomcat 的下载和安装

1、Tomcat 是 Java 领域最著名的开源 Web 容器，简单、易用，稳定性极好，既可以作为个人学习之用，也可以作为商业产品发布

1.3.1. 安装 Tomcat 服务器

1、因为 Tomcat 完全是纯 Java 实现，因此它是平台无关的，在任何平台上运行完全相同

2、Mac 平台下安装 Tomcat：

- http://tomcat.apache.org
- 下载 binary distributions 的 tar 包
- tar -xv -f <tar 包>
- 进入解压后的目录，bash ./startup.sh 即可
- 打开浏览器，在地址栏中输入 localhost:8080 即可进入 Tomcat 控制台

3、Tomcat 安装成功后，必须进行简单的配置，这些配置包括 Tomcat 的端口，控制台等

1.3.2. 配置 Tomcat 的服务端口

1、Tomcat 的默认服务端口是 8080，可以通过管理 Tomcat 配置文件来改变该服务端口，甚至可以通过修改配置文件让 Tomcat 在多个端口提供服务

2、Tomcat 的配置文件都放在 conf 目录下，控制端口的配置文件也放在该路径下，打开 conf 下的 server.xml 文件，务必用记事本或 vi 等无格式的编译器

3、如果需要让 Tomcat 运行多个服务，只需要复制 server.xml 文件中的<Service>元素，并修改相应的参数，便可以实现一个 Tomcat 运行多个服务，当然必须在不同的端口提供服务

1.3.3. 进入控制台

1、Tomcat 主页右上角显示三个控制台：

- Server Status 控制台：监控服务器的状态
- Manager App 控制台：部署、监控 Web 应用
- Host Manager 控制台：

1.3.4. 部署 Web 应用

1、在 Tomcat 中部署 Web 应用的方式主要有如下几种

- 利用 Tomcat 的自动部署
- 利用控制台部署
- 增加自定义的 Web 部署文件
- 修改 server.xml 文件部署 Web 应用，不建议采用

2、利用 Tomact 部署方式是最简单、最常用的方式，只要将一个 Web 应用复制到 Tomcat 的 webapps 下，系统就会把该应用部署到 Tomcat 中

3、利用控制台部署 Web 应用也很简单，其实质依然是利用 Tomcat 的自动部署

1.3.5. 配置 Tomcat 的数据源

1、从 Tomcat5.5 开始，Tomcat 内置了 DBCP 的数据源实现，所以可以非常方便地配置 DBCP 数据源

2、Tomcat 提供了两种配置数据源的方式，两种方式所配置的数据源的访问范围不同

- 一种数据源可以让所有 Web 应用都访问，被称为全局数据源
- 另一种只能在单个 Web 应用中访问，称为局部数据源
- 无论配置哪种数据源，都需要提供特定数据的 JDBC 驱动

3、局部数据源无须修改系统的配置文件，只需修改用户自己的 Web 部署文件，不会造成系统的混乱，而且数据源被封装在一个 Web 应用之内，防止被其他 Web 应用访问，提供更好的封装性

- 局部数据源值与特定的 Web 应用相关，因此在该 Web 应用对应的部署文件中配置

4、JDNI

- Java Naming Directory Interface，即 Java 命名和目录接口
- 其实就是为某个 Java 对象起一个名字，从而让其他程序可以通过该名字来访问该数据源对象

5、为 Web 应用增加局部数据源

- 修改在 Tomcat 下 conf/Catalina/localhost 下的 dd.xml 文件即可
- 详见 P13

1.4. Eclipse 的安装和使用

1、Eclipse 平台是 IBM 向开放源码社区捐赠的开发框架，IBM 宣称为开发 Eclipse

投入了 4 千万美元
2、Eclipse 允许增加新工具来扩充 Eclipse 的功能，这些新工具就是 Eclipse 插件
3、Eclipse 本身所提供的功能比较有限，但它的插件则大大提高了它的功能
- 插件有很多，例如 Synchronizer、Lomboz、MyEclipse 等

1.4.1. Eclipse 的下载和安装
1、Eclipse 插件的安装方式主要分为以下三种
- 在线安装
- 手动安装
- 使用本地压缩包安装

1.4.2. 在线安装 Eclipse 插件
1、如果网络环境允许，在线安装是种比较好的安装方式
2、步骤
- 单击 Eclipse 的"Help"菜单，选择"Install New Software..."
- "Work with"下拉列表框，通过该列表框可以选择 Eclipse 已安装过的插件选择指定插件项目后，该对话框的下面将会列出该插件所有可更新的项目
- 若需要升级已有插件，则通过"Work with"下拉列表框选择指定插件，然后在下面勾选需要更新的插件，单击 next
- 若需要安装新插件，单击"Add..."按钮
  - 在"Name"文本框中输入插件名(该名称任意，只用于标志该安装项)
  - 在"Location"文本框中输入插件的安装地址(从插件的官方站点上查询)
- 单击 Finish 进入安装界面

1.4.3. 从本地压缩包安装插件
1、上一小节，单击"Add..."按钮后，若是在线安装，则在 Location 中输入安装地址，若是本地安装，则选择"Archive..."按钮
2、然后 next 即可
3、目前绝大部分 Eclipse 插件都提供了这种本地压缩包

1.4.4. 手动安装 Eclipse 插件
1、只需要已经下载的插件文件，无需网络支持
- 手动安装适合于没有网络支持的环境
- 手动安装的适应性广，但是需要开发者自己保证插件版本与 Eclipse 版本兼容性

2、手动安装也分两种
- 直接安装
- 兼容安装

3、直接安装
- 将插件中包含的 plugins 和 features 文件夹的内容直接复制到 Eclipse 的 plugins 和 features 文件夹内，重启 Eclipse 即可
- 直接安装简单易用，但是效果非常不好，因为容易导致混乱：如果安装

插件非常多，可能导致用户无法精确判断哪些是 Eclipse 默认的插件，哪些是后来扩展的插件

- 如果要停用某些插件，需要从 Eclipse 的 plugins 和 features 文件夹内删除这些插件的内容，安装和卸载过程较为复杂

4、扩展安装

- 步骤
  - 在 Eclipse 安装路径下新建 links 路径
  - 在 links 文件夹内建立 xxx.link 文件，该文件的文件名是任意的，但是为了有较好的可读性，通常推荐该文件的主文件名与插件名相同，后缀为.link
  - 编辑 xxx.link 的内容
    - 添加如下内容：path=<pluginPath>
    - path=是固定的
    - <pluginPath>是插件扩展安装路径
  - 在 xxx.link 文件中的<pluginPath>路径下新建 eclipse 文件夹，再在 eclipse 文件夹内建立 plugins 和 features 文件夹
  - 将插件中包含的 plugins 和 features 文件夹的内容，复制到上面建立的 plugins 和 features 文件夹中，重启 Eclipse 即完成安装
- 扩展安装方式使得每个插件放在单独的文件内，因而结构非常清晰，如果需要卸载某个插件，只需要将该插件对应的 link 文件删除即可
- 相当于在直接安装的基础上，通过 link 文件来指定自定义的放置 plugins 和 features 的位置

1.4.5. 使用 Eclipse 开发 Java EE 应用

1、<未完成>：P20

1.4.6. 导入 Eclipse 项目

1、步骤

- 单击"File"--->"Import..."
- 选择"General"--->"Existing Projects into Workspace"---->Next
- 在"Select root directory"文本框内输入 Eclipse 项目的保存位置

1.4.7. 导入非 Eclipse 项目

1、步骤

- 新建普通的 Eclipse 项目
- 单击"File"--->"Import..."菜单
- 选择"General"--->"File System"--->Next
- 输入需要导入文件的路径，选中需要导入的文件，并输入需要导入到 Eclipse 项目的哪个目录下，然后单击 Finish

2、因为不同 IDE 工具对项目文件的组织方式完全不同，因此只能采用文件的方式依次导入

1.5. Ant 的安装和使用

1、Ant 是一种基于 Java 的生成工具，从作用上看，它有些类似于 C 编程(UNIX 平台上使用较多)的 Make 工具，C/C++项目经常使用 Make 工具来管理整个项目的编译、生成

2、Make 工具主要的缺陷

- Make 工具的本质还是依赖 UNIX 平台的 Shell 语言，所以 Make 无法跨平台
- Make 工具的生成文件的格式比较严格，容易导致错误

3、Ant 工具是基于 Java 语言的生成工具，所以具有跨平台能力，而且 Ant 工具使用 XML 文件来编写生成文件，因而具有更好的适应性

4、Ant 是 Java 的 Make 工具，而且是跨平台的，具有简单、易用的特性

- 由于 Ant 具有跨平台的特性，所以编写 Ant 生成文件时可能会失去一些灵活性
- 为此，Ant 提供了一个 exec 核心任务，这个任务允许执行特定操作系统上的命令

1.5.1. Ant 的下载和安装

1、步骤

- 登陆 ant.apache.org 进行下载 Ant 的最新版
  - Windows 下载*.zip
  - Linux 下载*.gz 或*.bz2
- 解压后即可
- 配置环境变量，否则就得用绝对路径来运行
- Linux 下运行 ant 即可

1.5.2. 使用 Ant 工具

1、正确安装 Ant 后，并配置环境变量之后，只需要输入 ant 或 ant.bat 即可

- 如果运行 ant 命令没有指定任何参数，Ant 会在当前麃搜索 build.xml 文件，如果找到了就以该文件作为生成文件，并执行默认的 target

2、<ant>

- ant [-s] [-f 文件名] [-qv] [-l 日志文件名]
- -s：或者使用-find，Ant 回到上级目录中搜索生成文件，直至叨叨文件系统的根路径
- -f：或-file，后接指定的生成文件
- -q：运行时只输出少量信息
- -v：运行时输出详细信息
- -l：后接文件名，将提示信息输出到指定文件

1.5.3. 定义生成文件

1、使用 Ant 的关键就是编写生成文件

- 生成文件定义了该项目的各个生成任务(以 target 表示，每一个 target 表示一个生成任务)，并定义生成任务之间的依赖关系
- 生成文件的默认名为 build.xml，也可以取其他名字，但必须作为参数传

递给 Ant 工具

- **生成文件可以放在项目的任何位置，但通常做法是放在顶层目录中，这样有利于保持项目的简洁和清晰**

2、Ant 生成文件格式

- Ant 生成文件的根元素是<project.../>，每个项目下面可以定义多个生成目标，每个生成目标以一个<target.../>元素来定义，它是<project.../>元素的子元素
- project 元素可以有多个属性
  - default：**指定默认 target，必须的属性**，如果运行 ant.bat 命令时没有显式指定想执行的 target，Ant 将执行该 target
  - basedir：**指定项目的基准路径**，生成文件中的其他相对路径都是基于该路径的
  - name：指定项目名，该属性仅指定一个名字，对编译、生成项目没有太大实际作用
  - description：指定项目的表述信息，对编译、生成项目没有太大实际作用
- <target.../>元素
  - **name：指定该 target 的名称，是必须的属性**，当使用 Ant 运行指定的生成目标时，就是根据该 name 来确定生成目标的，因此同一个生成文件里面不能有两个同名的 name
  - depends：该属性可以指定一个或多个 target
  - ，表示运行该 target 之前先运行该 depends 属性所指定的一个或多个 target
  - if：该属性指定一个属性名，用属性表示仅当设置了该属性时才执行此 target
  - unless：该属性指定一个属性名，用属性表示仅当没有设置该属性时才执行此 target
  - description：指定该 target 的描述信息
- 每个生成目标又可能由一个或多个任务序列组成，当执行某个生成目标时，实际上就是一次完成该目标所包含的全部任务
  - <name attribute1="value1" attributes="value2".../>
- 简而言之，Ant 生成文件的基本结构是 project 元素里包含多个 target 元素，而每个 target 元素里包含多个任务

3、Ant 的任务可以分为如下三类

- 核心任务：核心任务是 Ant 自带的任务
- 可选任务：可选任务是来自第三方的任务，因此需要一个附加的 JAR 文件
- 用户自定义任务：用户自定义任务是用户自己开发的任务

4、<project.../>元素可以拥有如下两个重要的子元素

- <property.../>：用于定义一个或多个属性
- <path.../>：用于定义一个或多个文件和路径

5、property 元素

- <property.../>元素用于定义一个或多个属性，Ant 生成文件中的属性类似

于编程语言中的宏变量，它们都具有名称和值，与变成语言不同的是 Ant 生成文件中的属性值不可变

- <property name="builddir" value="dd"/>
- ${builddir}可以获取属性值
- $符在 Ant 生成文件中具有特殊意义，如果希望 Ant 将生成文件中的$当成普通字符，就该使用$$
- <echo>$${builddir}=${builddir}</echo> ==> [echo] ${builddir}=dd
- echo 是 Ant 的核心任务之一，该任务将直接输出某个字符串，通常用于输出某些提示信息
- <property.../>的常用属性
  - name：指定需要设置的属性名
  - value：指定需要设置的属性值
  - resource：指定属性文件的资源名称，Ant 将负责从属性文件中读取属性名和属性值
  - file：指定属性文件的文件名，Ant 将负责从属性文件中读取属性名和属性值
  - url：指定属性文件 url 地址，Ant 将负责从属性文件中读取属性名和属性值
  - environment：用于指定系统环境变量的前缀，通过这种方式允许 Ant 访问系统环境变量
  - classpathref：指定搜索属性文件的 classpath
  - classpathref：指定搜索属性文件的 classpath 引用，该属性并不是直接给出 classpath 值，而是引用<path.../>元素定义的文件或路径集

6、path 元素和 classpath 元素

- 使用 Ant 编译，运行 Java 文件时常常需要引用第三方 JAR 包，这就需要使用<classpath.../>元素
- <path.../>和<classpath.../>都用于定义文件和路径集
  - classpath 通常作为其他任务的子元素，即可引用已有的文件和路径集，也可临时定义一个文件和路径集
  - path 元素作为<project.../>元素的子元素，用于定义一个独立的，有名称的文件和路径集，用于被引用
- <path.../>和<classpath.../>这两个元素都接受一下子元素
  - <dirset.../>：采用模式字符串的方式制定系列目录
  - <fileset.../>：采用模式字符串的方式制定系列文件
    - dir：指定文件集里多个文件所在的基准路径，必须属性
    - casesensitive：指定是否区分大小写，默认区分大小写
    - 另外还可以使用<include.../>和<exclude.../>两个子元素来指定包含和不包含这些文件
  - <filelist.../>：采用直接列出系列文件名的方式制定系列文件
    - dir：指定文件集里多个文件所在的基准路径，这是必须属性
    - files：多个文件名列表，多个文件名之间以英文逗号或空白隔开
  - <pathelement.../>：用于指定一个或多个目录，可以包含以下两个属性中的一个

- <path.../>：指定一个或多个目录(或 JAR 文件)，多个目录或 JAR 文件之间以英文冒号(:)或英文分号(;)分开
- <location.../>：指定一个目录或 JAR 文件

1.5.4. Ant 的任务(task)

1、<target.../>元素的核心就是 task，即每个<target.../>由一个或多个 task 组成

2、常用的核心 task

- javac：用于编译一个或多个 Java 源文件，通常需要 srcdir 和 destdir 两个属性，用于指定 Java 源文件的位置和编译后 class 文件的保存位置
- java：用于运行某个 Java 类，通常需要 classname 属性，用于指定需要运行哪个类
- jar：用于生成 JAR 包，通常需要指定 destfile 属性，用于指定所创建 JAR 包的文件名，除此之外，通常还应指定一个文件集，表明需要将哪些文件打包到 JAR 包里
- sql：用于执行一条或多条 SQL 语句
  - 通常需要 driver、url、userid 和 passwd 等属性，用于指定连接数据库的驱动类、数据库 URL、用户名和密码等
  - 还可以通过 src 指定所需要的 SQL 脚本文件，或者直接使用文本内容的方式指定 SQL 脚本字符串
- echo：用于输出某个字符串
- exec：执行操作系统的特定命令，通常需要 executable 属性
- copy：用于复制文件或路径
- delete：用于删除文件或路径
- mkdir：用于创建文件夹
- move：用户移动文件或路径

1.5.5. 一个 build.xml 的简单示例

```
<?xml version="1.0" encoding="UTF-8"?>

<!--定义生成文件的 project 根元素，默认的 target 为空-->
<project name="antQs" basedir="." default="">
    <!--定义三个简单属性-->
    <property name="src" value="src"/>
    <property name="classes" value="classes"/>
    <property name="dest" value="dest"/>
    <!--定义一组文件和路径集-->
    <path id="classpath">
        <pathelement path="${classes}"/>
    </path>

    <!--定义 help target，用于输出该生成文件的帮助信息-->
    <target name="help" description="打印帮助信息">
        <echo>help - 打印帮助信息</echo>
        <echo>coompile - 编译 Java 源文件</echo>
```

```
        <echo>run - 运行程序</echo>
        <echo>build - 打包  JAR 包</echo>
        <echo>clean - 清除所有编译生成的文件</echo>
    </target>

    <!--定义 comipile target，用于编译 Java 源文件-->
    <target name="compile" description="编译 Java 源文件">
        <!--先删除 classes 属性所代表的文件夹-->
        <delete dir="${classes}"/>
        <!--创建 classes 属性所代表的文件夹-->
        <mkdir dir="${classes}"/>
        <!--编译 Java 文件，编译后的 class 文件放到 classes 属性所代表的文件
        内-->
        <javac destdir="${classes}" debug="true" includeantruntime="yes"
        deprecation="false" optimize="false" failonerror="true">
            <!--指定需要编译 Java 文件所在的位置-->
            <src path="${src}"/>
            <!--指定编译 Java 文件所需要第三方类库所在的位置-->
            <classpath refid="classpath"/>
        </javac>
    </target>
</project>
```

1.6. Maven 的安装和使用

1、Maven 是一个比 Ant 更先进的项目管理工具

- 它采用一种"约定优于配置(CoC)"的策略来管理项目
- 它不仅用于把源代码构建成可发布的项目(包括编译、打包、测试和分发)，还可以生成 Web 站点等

1.6.1. 下载和安装 Maven

1、步骤

- http://maven.apache.org，下载 bin 包(Binary 程序)(Mac 平台)
- 将下载到的压缩文件解压缩到任意路径，具有如下结构
  - bin：保存 Maven 的可执行命令
  - boot：该目录只包含一个 plexus-classworlds-2.5.2.jar，这是一个类加载器框架，与默认的 Java 类加载器相比，提供了更丰富的语法以方便配置，Maven 使用该框架加载自己的类库，通常无需理会该文件
  - conf：保存 Maven 配置文件的目录，该目录包含 settings.xml 文件，该文件用于设置 Maven 的全局行为，通常建议将该文件复制到~/.m2/目录下，这样可以指设置当前用户的 Maven 行为
  - lib：该目录包含了所有 Maven 运行时需要的类库，Maven 本身是分模块开发的，因此用户能看到诸如 maven-core-3.3.9.jar 等文件，此外还包含 Maven 所依赖的第三方类库

- LICENSE、README.txt 等说明性文档

- 配置环境变量
  - JAVA_HOME：这个应该不用配了
  - M2_HOME：该环境变量应该指向 Maven 安装路径

1.6.2. 设置 Maven

1、设置 Maven 行为有两种方式

- 全局方式：通过 Maven 安装目录下的 conf/settings.xml 文件进行设置
- 当前用户方式：通过用户 Home 目录(若以 Windows 为例，则是 C:\Users\用户名\)的 m2\目录下的 settings.xml 文件进行设置
- 以上两种方式只是起作用的范围不同，但他们都使用 settings.xml 作为配置文件，而且这两种方式中 settings.xml 文件允许定义的元素也是不同的

2、Maven 允许设置如下参数

- localRepository：通过<localRepository.../>元素设置，该元素的内容是一个路径字符串，该路径用于设置 Maven 的本地资源库的路径
  - 资源库：资源库是 Maven 的一个重要概念，Maven 构建项目所使用的插件、第三方依赖库都集中存放在本地资源库中
- interactiveMode：该参数通过<interactiveMode.../>元素设置，该参数设置 Maven 是否处于交互模式
  - 如果处于交互模式，每当 Maven 需要用户输入时，Maven 都会提示用户输入
  - 如果将该参数设置为 false，那么 Maven 将不提示用户输入，而使用默认值
- offline：该参数设置 Maven 是否处于离线状态
  - 若该参数为 false，每当 Maven 找不到插件、依赖库时，Maven 总会尝试从网络下载
- proxies：该参数用于为 Maven 设置代理服务器
  - 该参数可包含多个<proxy.../>，每个<proxy.../>设置一个代理服务器，包括代理服务器的 ID、协议、代理服务器地址、代理服务器端口、用户名、密码等信息

3、<mvn>

- mvn <plugin-prefix>:<goal> -D<属性名>=<属性值> ...
- plugin-prefix：有效的插件前缀
- goal：该插件所包含的指定目标
- -D：用于为该目标指定属性
- 每次运行 mvn 命令可以通过多个-D 选项来指定属性名、属性值
- 例如 mvn help:system
  - help 就是一个 Maven 插件
  - system 就是 help 插件中的 goal

4、Maven 插件

- Maven 插件是一个非常重要的概念
- Maven 核心是一个空的容器，Maven 核心其实并不做任何事情，它只是解析一些 XML 文档，管理生命周期和插件

- Maven 的强大来自插件，这些插件可以编译源代码，打包二进制代码，发布站点等
- 换句话说 Maven 的"空"才是它的强大，因为 Maven 是"空"的，所以它可以装各种插件，因此它的功能可以无限扩展

1.6.3. 创建、构建简单的项目

1、创建项目使用 Maven 的 archetype 插件，maven.apache.org/plugins/index.html

2、archetype 包含以下目标(goal)

- archetype:create：已经过时的 goal
- archetype:generate：指定圆形创建一个 Maven 项目
- archetype:create-from-project：使用已有的项目创建 Maven 项目
- archetype:crawl：从仓库中搜索原型

3、例子

- mvn archetype:generate -DinteractiveMode=false -DgroupId=org.fkjava \ > -DartifactId=mavenQs -Dpackage=org.fkjava.mavenqs
- 根路径下包含一个 pom.xml 文件，该文件的作用类似于 Ant 的 build.xml 文件
  - pom.xml 文件被称为项目对象模型(Project Object Model)描述文件
  - Maven 采用一种被称为项目对象模型的方式来管理项目
  - POM 用于描述如下问题：
    - 该项目是什么类型
    - 该项目的名称是什么
    - 该项目的构建能自定义吗?
    - Maven 使用 pom.xml 文件来描述项目对象模型
    - 因此 pom.xml 并不是简单的生成文件，而是一种项目对象模型的描述文件
  - 一般来说，pom.xml 包含以下属性
    - <groupId.../>
    - <artifactId.../>
    - <packaging.../>
    - <version.../>
    - 以上几个属性定义了该项目唯一标识，被称为 Maven 坐标
    - <name.../>和<url.../>只是 pom.xml 提供的描述性元素用于提供可阅读的名字
- 在包含 pom.xml 文件的所在的路径中输入如下命令
  - mvn compile
- 接下来使用 Maven 的 exec 插件来执行 Java 类
  - mvn exec:java -Dexec.mainClass="org.fkjava.mavenqs.App"
  - org.fkjava.mavenqs.App 就是该 Maven 项目所生成的主类

4、Maven 怎么知道如何编译项目，pom.xml 为何如此神奇

- Maven 运行时根据设置组合来运行的
- 每个 Maven 项目的 pom.xml 都有一个上级 pom.xml，当前项目的 pom.xml 的设置信息将会被合并到上级的 pom.xml 中

- mvn help:effective-pom <==可以看到完整的设置
- 如果开发者希望改变其中默认的设置，也可以在当前项目的 pom.xml 中定义对应的元素来覆盖上级 pom.xml 中的默认设置

1.6.4. Maven 的核心概念

1、只要将项目的源文件按 Maven 要求的规范组织，并提供 pom.xml 文件，即使 pom.xml 文件中只包含极少的信息，开发者也依然可以使用 Maven 来编译项目，运行程序，甚至可以运行测试用例，打包项目，这是由于 Maven 采用了"约定优于配置(Convention over Configuration, Coc)"，Maven 的主要约定有如下几条：

- 源代码应该位于${basedir}/src/main/java 路径下
- 源文件应该位于${basedir}/src/main/resources 路径下
- 测试代码应该位于${basedir}/src/test 路径下
- 编译生成的 class 文件应该位于${basedir}/target/classes 路径下
- 项目会产生一个 JAR 文件，并将生成的 JAR 包放在${basedir}/target 路径下
- 通过这些约定，就可以避免像 Ant 构建项目那样必须为每个子项目定义这些目录
- 除此之外，Maven 对核心插件也使用了一组通用的约定，用来编译源代码、打包可分发的 JAR、生成 Web 站点，以及许多其他过程

2、Maven 的强大很大程度来自于它的"约定"，Maven 预定义了一个固定的声明周期，以及一组用于构建和装配软件的通用插件，如果开发者完全遵循这些规定，Maven 只需要将源代码放到正确的目录下，Maven 即可处理剩下的事情

1.6.4.1. Maven 的生命周期(lifecycle)

1、进入 pom.xml 文件所在的路径执行如下命令(上一节生成的 mavenQs 项目)

- mvn install
- 上面的命令只是告诉 Maven 运行 intall，但从实际的运行结果看，Maven 不仅运行了 install，而且还在该插件之前运行了大量的插件，这就是 Maven 生命周期所导致的

2、生命周期：

Maven 构建项目包含多个有序的阶段(phase)，Maven 可以支持许多不同的声明周期，最常用的声明周期是 Maven 默认的生命周期

- Maven 生命周期中的元素称为(phase)(阶段)，每个生命周期由多个阶段组成，Maven 生命周期中的各个阶段总是按顺序、依次执行的
- Maven 默认的生命周期的开始阶段是验证项目的基本完整性
- Maven 默认的生命周期的结束阶段是将该项目发布到远程仓库
- mvn 命令除了使用 mvn <plugin-prefix>:<goal> 运行指定插件的目标外，还可以使用如下命令格式
  - mvn <phase1> <phase2>...
  - 当使用上述命令时，mvn 命令告诉 Maven 执行生命周期的某个阶段，Maven 会从生命周期的第一个阶段开始执行，直至 mvn 命令指定的阶段
- Maven 包含三个基本的生命周期

- clean 生命周期
  - 用于在构建项目之前进行一些清理工作，该生命周期包含如下三个核心阶段
  - pre-clean：在构建之前执行预清理
  - clean：执行清理
  - post-clean：最后清理
- default 生命周期
  - 默认的生命周期则包含了该项目构建的核心部分，默认的生命周期包含如下核心阶段
  - compile：编译项目
  - test：单元测试
  - package：项目打包
  - install：安装到本地仓库
  - deploy：部署到远程仓库
- site 生命周期
  - site 生命周期用于生成项目报告站点、发布站点，该生命周期包含如下四个核心阶段
  - pre-site：生成站点之前做验证
  - site：生成站点
  - post-site：生成站点之后做验证
  - site-deploy：发布到远程服务器

1.6.4.2. 插件和目标(plugins and goal)

1、Maven 插件甚至可以把 Ant 整合进来，使用 Maven 来运行 Ant 的生成文件

2、当使用 mvn 运行 Maven 生命周期的指定阶段时，各阶段所完成的工作其实也是由插件实现的

- 插件目标可以绑定到生命周期的各个阶段上，每个阶段可能绑定了零个或多个目标

3、Maven 生命周期的各阶段也是一个抽象的概念，对于软件构建过程来说，默认的生命周期被划分为 compile、test、package、install、deploy 这 5 个阶段

1.6.4.3. Maven 的坐标(coordinate)

1、POM 需要为项目提供一个唯一标识符，这个标识符就被称为 Maven 坐标，Maven 坐标由如下元素组成

- groudId：该项目的开发者的域名
- artifactId：指定项目名
- packaging：指定项目打包的类型
- version：指定项目的版本

1.6.4.4. Maven 的资源库(repository)

1、Maven 资源库用于保存 Maven 插件，以及各种第三方框架，简单来说，Maven 用到的插件、项目依赖的各种 JAR 包，都会保存在资源库中

2、Maven 资源库可以分为如下三种

- 本地资源库：
  - Maven 用到的所有插件、第三方框架都会下载到本地库
  - 只有当本地库中找不到时才采取从远程下载
  - 开发者可以通过 Maven 安装目录下的 conf/settings.xml 文件，或者用户 Home 目录下的 m2/settings.xml 文件中的<localResppository.../>元素进行设置
- 远程资源库：
  - 远程资源库通常由公司或团队进行集中维护
  - 通过远程资源库，可以让全公司的项目使用相同的 JAR 包系统
- 中央资源库(默认)：
  - 中央资源库由 Maven 官方维护，包括各种公开的 Maven 插件、各种第三方项目
  - http://repo1.maven.org/maven2

3、当 Maven 需要某个插件或 JAR 包时，搜索顺序为：本地资源库---远程资源库---中央资源库

1.6.5. 依赖管理

1、本节主要介绍如何使用 Maven 构建 Web 项目，并为 Web 项目添加第三方框架

2 、 mvn archetype:generate -DgroupId=org.crazyit -DartifactId=struts2qs -Dpackage=org.crazyit.struts2qs -DarchetypeArtifactId=maven-archetype-webapp -DinteractiveMode=false

3、在 pom.xml 中添加如下依赖

```
<dependency>
<groupId>org.apache.struts</groupId>
<artifactId>struts2-core</artifactId>
<version>2.3.16.3</version>
</dependency>
```

- 然后运行 mvn package

1.6.6. POM 文件的元素

1、元素如下

- <properties.../>：该元素用于定义全局属性
- <dependencies.../>：该元素用于定义依赖关系
- <dependencyManagent.../>：该元素用于定义依赖管理
- <build.../>：该元素用于定义构建信息
- <reporting.../>：该元素用于定义站点报告的相关信息
- <licenses.../>：该元素用于定义该项目的 License 信息
- <organization.../>：该元素指定该项目所属的组织信息
- <developers.../>：该元素用于配置该项目的开发者信息
- <contributors.../>：该元素用于配置该项目的贡献者信息
- <issueManagement.../>：定义该项目的 bug 跟踪系统
- <mailingLists.../>：定义该项目的邮件列表
- <scm.../>：指定该项目源代码管理工具，如 CVS、SVN 等

- <repositories.../>：该元素用于定义远程资源库的位置
- <pluginRepositorie.../>：该元素用于定义插件资源库的位置
- <distributionManagement.../>：部署管理
- <profiles.../>：该元素指定根据不同环境调整构建配置

1.7. 使用 SVN 进行协作开发

1、通常会选择合适的版本控制工具来进行版本控制和版本管理

- CVS(Concurrent Versions System)：目前开源项目、Java 项目中应用最广泛的版本控制工具
- SVN(Subversion)：SVN 是 CVS 的替代产物，SVN 尽力维持 CVS 的用法习惯，并对原来 CVS 进行增强
- VSS(Vistal Source Safe)：Windows 项目的版本控制工具，具有简单易用、方便高效的特点，但与 Windows 操作系统及微软开发工具高度集成

2、相对于 CVS 版本控制工具，SVN 具有以下优势

- **统一的版本号**：
  - CVS 是对每个文件单独顺序编排版本号，因此同一项目内个文件的版本号可能各
  - 相同
  - SVN 对任何一次提交都会对所有文件增加一个版本号，即使该次提交不涉及的文件也会增加一个版本号，因此 SVN 同一个项目所有文件在任意时刻的版本号是相同的，版本号相同的文件构成软件的一个版本
- **原子提交**：
  - **提交要么全部进入版本库，要么一点改变都不发生**
  - 这可以保证一次提交不管是单个文件还是多个文件，都将作为一个整体提交
  - 在这当中发生的任何意外(网络中断)都不会引起版本的不完整和数据损坏
  - 换言之，一旦提交过程中发生意外，将会撤销本次提交
- **目录版本控制**：
  - CVS 只能记录文件的版本变更历史
  - SVN 可以跟踪整个目录树的修改，它会记录所有文件和目录的版本变更历史
  - 因此 SNV 可以记录对项目中所有文件、目录的重命名、复制删除等操作
- **高效的分支和标签**：
  - SVN 创建分支，标签的开销非常小
- **优化过的数据库访问**：
  - 使得一些操作不必访问数据库就可以得到
  - 这样减少了很多不必要的和数据库主机之间的网络流量
- **支持元数据(Metadata)管理**：
  - 每个目录或文件可以定义一组附件的"属性(Property)"，这些属性是允许用于任意定义的 key-value(键/值)对
- **优化的版本库存储**：

- SVN 采用更加节省空间的存储方式来保存版本库

1.7.1. 下载和安装 SVN 服务器

1、Mac 自带的 svn

1.7.2. 配置 SVN 资源库

1、在任何位置创建一个空文件夹用于保存 SVN 资源库

2、svnadmin create /Users/HCF/svnData/webDemo

- create 是创建资源库的选项，webDemo 就是所创建资源库的名称

3、webDemo/conf/目录下保存了 SVN 资源库的相关配置信息

4、打开 conf/svnserve.conf 文件，取消如下两行的注释

- anon-access = read <==允许匿名用户读取该资源库
- auth-access = write <==允许授权用户对该资源库执行读取、写入操作

5、打开 conf/svnserve.conf 文件，取消下面一行的注释

- password-db = passwd <==指定该 SVN 资源库使用 passwd 文件来保存用户名、密码，当然可以改变存储用户名、密码的文件名

6、打开 conf/passwd 文件，添加如下内容

- liuye.org=hcflh19930101

7、经过以上几个步骤，SVN 服务器配置完成

- 启动 SVN 服务器：svnserve -d -r /Users/HCF/svnData
- svnserve 是 SVN 服务器安装路径下 bin 路径下的一个可执行程序
- 运行 SVN 服务器需要 3690 端口

<未完成>：不知道 Mac 上怎么安装 SVN 客户端，下载了一个 snvx 不会用，先跳过把

Chapter 2. JSP/Servlet 及相关技术详解

1、JSP(Java Server Page)和 Servlet 是 Java EE 规范的两个基本成员

- JSP 和 Servlet 的本质是一样的
- JSP 最终必须编译成 Servlet 才能运行
- 或者说 JSP 只是 Servlet 的"草稿"文件

2、JSP 的特点是在 HTML 页面中嵌入 Java 代码片段，或使用各种 JSP 标签，包括使用用户自定义的标签，从而可以动态地提供页面内容

- 早期 JSP 页面的使用非常广泛，一个 Web 应用可以全部由 JSP 页面组成，只辅以少量的 JavaBean 即可
- 从 Java EE 标准出现后，人们逐渐认识到使用 JSP 充当过多的角色是不合适的，因此 JSP 慢慢发展成单一的表现层技术，不再承担业务逻辑组件及持久层组件的责任

3、随着 Java EE 技术的发展，又出现了 FreeMarker、Velocity、Tapestry 等表现层技术，虽然这些技术基本可以取代 JSP 技术，但实际上 JSP 依然是应用最广泛的表现层技术

2.1. Web 应用和 web.xml 文件

1、JSP、Servlet、Listener 和 Filter 等都必须运行在 Web 应用中

2.1.1. 构建 Web 应用

1、要想成为一个优秀的程序员，应该从基本功练起，所有的代码都应该用简单的文本编辑器完成

- 坚持用最原始的工具来学习技术，会让你对整个技术的每个细节由更准确的把握
- 可以使用 IDE 工具，但绝不能依赖 IDE 工具
- 学习阶段，不要使用 IDE 工具
- 开发阶段，使用 IDE 工具
- IDE 工具会加快高手的开发效率，但会使初学者更白痴

2、"徒手"建立 Web 应用

1) 在任意目录下新建一个文件夹，此处将以 webDemo 文件夹建立一个 Web 应用
2) 在第 1 步所建的文件夹内建一个 WEB-INF 文件夹(注意大小写)
3) 进入 Tomcat 或其他任何 Web 容器内，找到一个 Web 应用，将 Web 应用的 WEB-INF 下的 web.xml 文件复制到第 2 步所建的 WEB-INF 文件夹下(对于 Tomcat 而言，其 webapps 路径下由大量的示例 Web 应用)
4) 修改复制后的 web.xml 文件，将该文件修改成只有一个根元素的 XML 文件(具体见 P60)
5) 在第 2 步所建的 WEB-INF 路径下，新建两个文件夹：classes 和 lib
   - 这两个文件夹的作用完全相同：都是用于保存 Web 应用所需要的 Java 类文件
   - 区别是：classes 保存单个*.class 文件；lib 保存打包后的 JAR 文件

- 经过以上步骤，已经建立了一个空 Web 应用，将 Web 应用复制到 Tomcat 的 webapps 路径下，该 Web 应用将可以自动部署在 Tomcat 中

- 已成功！

3、**必须将工程直接放在 webapps 路径下，而不能嵌几层，否则 Tomcat 无法读取<工程目录>/WEB-INFZ/classes/路径下的.class 文件**

2.1.2. 配置描述符 web.xml

1、在 Servlet 2.5 规范之前，每个 Java Web 应用都必须包含一个 web.xml 文件，且必须放在 WEB-INF 路径下

- 从 Servlet 3.0 开始，WEB-INF 路径下的 web.xml 文件不再是必需的，但通常还是建议保留该配置文件

2、对 Java Web 而言，WEB-INF 是一个特殊的文件夹，Web 容器会包含该文件夹下的内容，客户端浏览器无法访问 WEB-INF 路径下的任何内容

3、<未完成>

2.2. JSP 的基本原理

1、JSP 的本质是 Servlet，当用户指向指定 Servlet 发送请求时，Servlet 利用输出流动态生成 HTML 页面，包括每一个静态 HTML 标签和所有在 HTML 页面中出现的内容

- 由于包括大量的 HTML 标签、大量的静态文本及格式等，导致 Servlet 开发效率极为低下
- 所有表现逻辑，包括布局、色彩及图像等，都必须耦合在 Java 代码中

2、JSP 的出现弥补了 servlet 的不足

- JSP 通过在标准的 HTML 页面中嵌入 Java 代码，其静态的部分无须 Java 控制，只有那些需要从数据库读取或需要动态生成的页面内容，才使用 Java 脚本控制
- JSP 页面的内容可以分为如下两部分
  - 静态部分：标准的 HTML 标签、静态的页面内容，这些内容与 HTML 页面相同
  - 动态部分：受 Java 程序控制的内容，这些内容由 Java 脚本动态生成

3、JSP 页面

- <%Java 代码%> <==通过这种方式就可以把 Java 代码嵌入 HTML 页面中，这就变成了动态的 JSP 页面
- 表面上看，JSP 页面已经不需要 Java 类，似乎完全脱离了 Java 面向对象的特征
- 事实上，JSP 页面的本质依然是 Servlet(一个特殊的 Java 类)，每个 JSP 页面就是一个 Servlet 的实例---JSP 页面由系统编译成 Servlet，Servlet 再负责响应用户请求
- 也就是说，JSP 其实是 Servlet 的一种简化，使用 JSP 时，其实还是使用 Servlet，因为 Web 应用中的每个 JSP 页面都会由 Servlet 容器生成对应的 Servlet

4、启动 Tomcat 之后，可以在 Tomcat 的 work/Catalina/localhost/<Web 应用名称>/org/apache/jsp/<Web 应用名称>目录下找到.java 以及.class 文件

- 这两类文件都是由 Tomcat 生成的，Tomcat 根据 JSP 页面生成对应 Servlet 的 Java 文件和 class 文件

- .java 文件就是一个 Servlet 类的源代码，该 Java 类主要包含以下三个方法
  - init()：初始化 JSP/Servlet 的方法
  - destroy()：销毁 JSP/Servlet 之前的方法
  - service()：对用户请求生成响应的方法

5、总结

- JSP 文件必须在 JSP 服务器内运行
- JSP 文件必须生成 Servlet 才能执行
- 每个 JSP 页面的第一个访问者速度很慢，因为必须等待 JSP 编译成 Servlet
- JSP 页面的访问者无需安装任何客户端，甚至不需要可以运行 Java 的运行环境，因为 JSP 页面输送到客户端的是标准 HTML 页面
- JSP 技术的出现，大大提高了 Java 动态网站的开发效率，所以得到了 Java 动态网站开发者的广泛关注

2.3. JSP 的四种基本语法

1、编写 JSP 页面非常简单：在静态 HTML 页面中"镶嵌"动态 Java 脚本即可

2、现在开始学习的内容是：JSP 页面的四种基本语法---也就是 JSP 允许在静态 HTML 中"镶嵌"的成分

3、可以按照以下步骤开发 JSP 页面

- 编写一个静态 HTML 页面
- 用合适的语法向静态 HTML 页面中"镶嵌"4 种基本语法的一种或多种，这样即可为静态 HTML 页面增加动态内容

2.3.1. JSP 注释

1、JSP 注释用于标注程序开发过程中的开发提示，它不会输出到客户端

2、JSP 注释格式如下

- <%--注释内容--%>

3、与 JSP 注释形成对比的是 HTML 注释，HTML 注释的格式是：

- <!--注释内容-->

2.3.2. JSP 声明

1、JSP 声明用于声明变量和方法

- 在 JSP 声明中声明方法看起来很特别，似乎不需要定义类就可以直接定义方法，方法似乎可以脱离独立存在
- 实际上，JSP 声明将会转换成对应 Servlet 的成员变量或成员方法

2、声明的格式如下

- <%!声明部分%>

3、由于 JSP 声明语法定义的变量和方法对应于 Servlet 类的成员变量和方法

- 所以 JSP 声明部分定义的变量和方法可以使用 private、public 等访问控制修饰符，也可以使用 static 修饰，将其变成类属性和类方法
- 但不能使用 abstract 修饰声明部分的方法，因为抽象方法将导致 JSP 对应 Servlet 变成抽象类，从而导致无法实例化

4、JSP 页面会编译成一个 Servlet 类，每个 Servlet 在容器中只有一个实例，在 JSP 中声明的变量是成员变量，成员变量只在创建实例时初始化，该变量的值将

一直保存，直到实例销毁
5、JSP 声明中独立存在的方法，只是一种假象

2.3.3. 输出 JSP 表达式

1、JSP 提供了一种输出表达式值的简单方法，其语法格式如下
- <%=表达式%>

2、输出表达式将转换成 Servlet 里的输出语句

2.3.4. JSP 脚本

1、以前 JSP 脚本的应用非常广泛，因此 JSP 脚本里可以包含任何可执行的 Java 代码，通常来说，所有可执行性 Java 代码都可通过 JSP 脚本嵌入 HTML 页面
2、不仅 JSP 小脚本部分会转换成_jspService 方法里的可执行代码
- JSP 页面里的所有静态内容都将由_jspService 方法里输出语句来输出，这就是 JSP 脚本可以控制 JSP 页面中静态内容的原因
- 由于 JSP 脚本将转换成_jspService 方法里的可执行代码，而 Java 语法不允许在方法里面定义方法，因此 JSP 脚本能定义方法

3、<未完成>：如何导入 mysql 的 jar 包

2.4. JSP 的 3 个编译指令

1、常见的编译指令有如下三个
- page：该指令是针对当前页面的指令
- include：用于指定包含另一个页面
- taglib：用于定义和访问自定义标签

2、编译指令的语法格式如下
- <%@ 编译指令名 属性名="属性值"……%>

2.4.1. page 指令

1、page 指令通常位于 JSP 页面的顶端，一个 JSP 页面可以使用多条 page 指令
2、page 指令常用属性介绍
- language：声明当前 JSP 页面使用的脚本语言的种类，因为是 JSP 页面，该属性的值通常都是 java，该属性的默认值也是 java，通常无需设置
- extends：指定 JSP 页面编译所产生的 Java 类继承的父类，或所实现的接口
- import：用来导入包，下面几个包是默认导入的
  - java.lang.*
  - javax.servlet.*
  - javax.servlet.jsp.*
  - javax.servlet.http.*
- sesstion：设定这个 JSP 页面是否需要 HTTP Session
- buffer：指定输出缓冲区的大小
  - 输出缓冲器的 JSP 内部对象：out 用于缓存 JSP 页面对客户浏览器的输出，默认值为 8KB，可以设置为 none，也可以设置为其他值，单位 KB
- autoFlush：当输出缓冲区即将溢出时，是否需要强制输出缓冲区的内容

- 设置为 true 时为正常输出
- 设置为 flase 则会在 buffer 溢出时产生一个异常

➢ info：设置该 JSP 程序的信息，也可以看做其说明
- 可以通过 Servlet.getServletInfo()方法获取该值
- 如果在 JSP 页面中，直接调用 getServletInfo()方法获取该值，因为 JSP 页面的实质就是 Servlet

➢ errorPage：指定错误处理页面
- 如果本页面产生了异常或者错误，而该 JSP 页面没有对应的处理代码，则会自动调用该属性所指定的 JSP 页面

➢ isErrorPage：设置本 JSP 页面是否为错误处理程序
- 如果该页面本身已是错误处理页面，则通常无需指定 errorPage

➢ contentType：用于设定生成网页的文件格式和编码字符集
- 默认的 MIME 类型是 text/html
- 默认的字符集类型为 ISO-8859-1

➢ pageEncoding：指定生成网页的编码字符集

3、errorPage 属性的实质是 JSP 的异常处理机制
➢ JSP 脚本不要求强制处理异常，即使该异常时 checked 异常
➢ 如果 JSP 页面在运行中抛出未处理的异常，系统将自动跳转到 errorPage 属性指定的页面
➢ 如果 errorPage 没有指定错误页面，系统则直接把异常信息呈献给客户端浏览器---这是所有开发者都不愿意看到的场景

2.4.2. include 指令

1、使用 include 指令，可以将一个外部文件嵌入到当前 JSP 文件中，同时解析这个页面中的 JSP 语句
➢ 这是个静态的 include 语句，它会把目标页面的其他编译指令也包含进来，但动态 include 则不会

2、include 既可以包含静态的文本，也可以包含动态的 JSP 页面
➢ 静态的 include 编译指令会将被包含的页面加入本页面，融合成一个页面，因此被包含页面甚至不需要是一个完整的页面

3、语法如下
➢ <%@ include file="relativeRULSpec" %>
➢ 如果被嵌入的文件经常需要改变，建议使用<jsp:include>操作指令，因为它是动态的 include 语句

4、静态包含的意义：包含页面在编译时将完全包含了被包含页面的代码
➢ 静态包含还会将被包含页面的编译指令也包含进来，如果两个页面的编译指令冲突，那么页面就会出错

2.5. JSP 的 7 个动作指令

1、动作指令与编译指令不同
➢ 编译指令是通知 Servlet 引擎的处理消息，而动作指令只是运行时的动作
➢ 编译指令将在 JSP 编译成 Servlet 时起作用，而动作指令通常可替换成 JSP 脚本，它只是 JSP 脚本的标准化写法

2、JSP 动作指令主要有如下 7 个

- jsp:forward：执行页面转向，将请求的处理转发到下一个页面
- jsp:param：用于传递参数，必须与其他支持参数的标签一起使用
- jsp:include：用于动态引入一个 JSP 页面
- jsp:plugin：用于下载 JavaBean 或 Applet 到客户端执行
- jsp:useBean：创建一个 JavaBean 的实例
- jsp:setProperty：设置 JavaBean 实例的属性值
- jsp:getProperty：输出 JavaBean 实例的属性值

2.5.1. forward 指令

1、forward 指令用于将页面响应转发到另外的页面，既可以转发到静态的 HTML 页面，也可以转发到动态的 JSP 页面，或者转发到容器中的 Servlet

2、JSP 的 forward 指令的格式如下

- 对于 JSP 1.0

```
<jsp:forward page="{relativeURL|<%=expression%>"/>
```

- 对于 JSP 1.1 以上规范

```
<jsp:forward page="{relativeURL|<%=expression%>"/>
    {<jsp:param.../>}
</jsp:forward>
```

  - 第二种语法用于在转发时增加额外的请求参数
  - 增加的请求参数值可以通过 HttpServletRequest 类的 getParameter()方法获取

2.5.2. include 指令

1、include 指令是一个动态 include 指令，也用于包含某个页面，它不会导入被 include 页面编译指令，仅仅将被导入页面的 body 内容插入本页面

2、语法格式

- <jsp:include page="{relativeURL | <%=expression%>}" flush="true"/>
- <jsp:include page="{relativeURL | <%=expression%>}" flush="true">
  <jsp:param name="parameterName" value="patameterValue"/>
  </jsp:include>
- flush 属性用于指定输出缓存是否转移到被导入文件中
  - 若为 true：包含在被导入文件中
  - 若为 false：包含在原文件中

3、静态导入和动态导入有以下三点区别

- 静态导入是将被导入页面的代码完全融入，两个页面融合成为一个整体 Servlet；动态导入则在 **Servlet 中使用 include 方法**来引入被导入页面的内容
- 静态导入时被导入页面的编译指令会起作用；动态导入时被导入页面的编译指令则失去作用，只是插入被导入页面的 body 内容
- 动态包含还可以增加额外的参数

4、forward 和 include

- forward 动作指令和 include 动作指令非常相似(语法就很相似)，它们都采用方法来引入目标页面

- forward 指令使用_jspx_page_context 的 forward()方法来引入目标页面；而 include 指令则通过 JspRuntimeLibrary 的 include()方法来引入目标页面
- 执行 forward 时，被 forward 的页面将完全代替原有页面：而执行 include 时，被 include 的页面只是插入原有页面

2.5.3. useBean、setProperty、getProperty 指令

1、这三个指令都与 JavaBean 相关的指令
- useBean 指令用于在 JSP 页面中初始化一个 Java 实例
- setProperty 指令用于为 JavaBean 实例的属性设置值
- getProperty 指令用于输出 JavaBean 实例的属性

2、如果多个 JSP 页面中需要重复使用某段代码，则可以把这段代码定义成 Java 类的方法，然后让多个 JSP 页面调用该方法即可

3、useBean 的语法格式

```
<jsp:useBean id="name" class="classname" scope="page | request | session| application" />
```

- id：JavaBean 的实例名
- class：确定 JavaBean 的实现类
- scope：用于指定 JavaBean 实例的作用范围，该范围有如下 4 个值
  - page：该 JavaBean 实例仅在该页面有效
  - request：该 JavaBean 实例在本次请求有效
  - session：该 JavaBean 实例在本次 session 内有效
  - application：该 JavaBean 实例在本次应用内一直有效

4、setProperty 的语法格式

```
<jsp:setProperty name="BeanName" property="propertyName" value="value"/>
```

- name：用于确定需要设定 JavaBean 的实例名
- property 属性确定需要设置的属性名
- value 属性则确定需要设置的属性值

5、getProperty 的语法格式

```
<jsp:getProperty name="BeanName" property="propertyName" />
```

- name：确定属性需要输出的 JavaBean 的实例名
- property：确定需要输出的属性名

2.5.4. plugin 指令

1、plugin 指令主要用于下载服务器端的 JavaBean 或 Applet 到客户端执行，由于程序在客户端执行

2、由于现在很少使用 Applet，而且就算要用 Applet，也完全可以使用支持 Applet 的 HTML 标签，所以 jsp:plugin 标签的使用场景并不多

2.5.5. param 指令

1、param 指令用于设置参数值，这个指令本身不能单独使用，因为单独的 param 指令没有实际意义

2、param 指令可以与以下三个指令一起使用

- jsp:include：param 指令用于将参数值传入被导入的页面
- jsp:forward：param 指令用于将参数值传入被转向的页面
- jsp:plugin：param 指令用于将参数传入页面中的 JavaBean 实例或 Applet 实例

2. 6. JSP 脚本中的 9 个内置对象

1、JSP 脚本中包含 9 个内置对象，这个 9 个内置对象都是 Servlet API 接口的实例，只是 JSP 规范对它们进行了默认初始化(由 JSP 页面对应 Servlet 的_jspService()方法来创建这些实例)。也就是说，它们已经是对象，可以直接使用

2、9 个内置对象依次如下

- application：
  - javax.servlet.ServletContext 的实例
  - 该实例代表 JSP 所属的 Web 应用本身，可用于 JSP 页面，或者在 Servlet 之间交换信息
  - 常用方法
    - getAttribute(String attName)
    - setAttribute(String attName, String attValue)
    - getInitParameter(String paramName)
- config：
  - javax.servlet.ServletConfig 的实例
  - 该实例代表 JSP 的配置信息(事实上，JSP 页面通常无需配置，也就不存在配置信息，因此，该对象更多的在 Servlet 中无效)
  - 常用方法
    - getInitParameter(String paramName)
    - getInitParameternames()
- exception：
  - java.lang.Throwable 的实例
  - 该实例代表其他页面中的异常和错误
  - 只有当页面是错误处理页面，即编译指令 page 的 isErrorPage 属性为 ture 时，该对象才能用
  - 常用方法
    - getMessage()
    - printStackTrace()
- out：
  - javax.servlet.jsp.JspWriter 的实例
  - 该实例代表 JSP 页面的输出流，用于输出内容，形成 HTML 页面
- page：
  - 代表该页面本身，通常没有太大用处
  - 也就是 Servlet 中的 this，其类型就是生成的 Servlet 类
  - 能用 page 的地方就能用 this
- pageContext：
  - javax.servlet.jsp.PageContext 的实例
  - 该对象代表该 JSP 页面上下文，使用该对象可以访问页面中的共享数

据
- 常用方法
  - getServletContext()
  - getServletConfig()

- request：
  - javax.servlet.http.HttpServletRequest 的实例
  - 该对象封装了一次请求，客户端的请求参数都被封装在该对象里
  - 这是一个常用对象，获取客户端请求参数必须使用该对象
  - 常用方法
    - getParameter(String paramName)
    - getParameterValues(String paramName)
    - setAttribute(String attrName,Object attrValue)
    - getAttribute(String attrName)
    - setCharacterEncoding(String env)
- response：
  - javax.servlet.http.httpServletResponse 的实例
  - 代表服务器对客户端的响应
  - 通常很少使用该对象响应，而是使用 out 对象，除非需要生成非字符响应
  - response 对象通常用于重定向，常用方法有
    - getOutputStream()
    - sendRedirect(java.lang.String location)
- session：
  - javax.servlet.http.HttpSession 的实例
  - 该对象代表一次会话
    - 当客户端浏览器与站点建立连接时，会话开始
    - 当客户端关闭浏览器，会话结束
  - 常用方法
    - getAttribute(String attrName)
    - setAttribute(String attrName,Object attrValue)

3、JSP 内置对象的实质
- 找到任意一个 Servlet 类的实现即可
- 它们要么是_jspService()方法的形参，要么是_jspService()方法的局部变量，所以可以直接在 JSP 脚本(脚本将对应于 Servlet 的_jspService()方法部分)中调用这些对象，无须创建它们

2.6.1. application 对象

1、虽然把基于 Web 应用称为 B/S(Browser/Server)架构的应用，但其实 Web 应用一样是 C/S(Client/Server)结构的应用，只是这种应用的服务器是 Web 服务器，而客户端是浏览器

2、对于大部分浏览器而言，它通常负责完成三件事情
- 向远程服务器发送请求
- 去读远程服务器返回的字符串数据
- 负责根据字符串数据渲染出一个丰富多彩的页面

3、事实上，浏览器是一个非常复杂的网络通信程序，它除了可以向服务器发送请求，读取网络数据外，最大的技术难点在于将 HTML 文本渲染成页面，建立 HTML 页面的 DOM 模型，支持 JavaScript 脚本程序等

- 通常浏览器有 Internet Explorer、FireFox、Opera、Safari 等

4、对于每次客户端请求而言，服务器大致需要完成以下几个步骤

1) 启动单独的线程(最新版的 Tomcat 已经不需要对每个用户请求都启用单独的线程，使用普通 I/O 读取用户请求的数据，而是采用异步 IO，具有更高的性能)
2) 使用 I/O 读取用户请求的二进制数据流
3) 从请求数据中解析参数
4) 处理用户请求
5) 生成响应数据
6) 使用 IO 流向客户端发送请求数据

- 1、2、6 是通用的，可由 Web 服务器完成
- 3、4、5 存在差异：因为不同请求里包含的请求参数不同，处理用户请求的方式也不同，所生成的响应自然也不同
- Web 服务器会调用 Servlet 的_jspService()方法来完成 3、4、5 步，编写 JSP 页面时，页面里的静态内容，JSP 脚本都会转换成_jspService()方法的执行代码，这些执行代码负责完成解析参数、处理请求、生成响应等业务功能，**而 Web 服务器则负责完成多线程，网络通信等底层功能**
- Web 服务器在执行了第三步解析到用户的请求参数后，需要通过这些请求参数来创建 HttpServletRequest、HttpServletResponse 等对象，作为调用_jspService()方法的参数，一个 Web 服务器必须为 Servlet API 中绝大部分接口提供实现类

5、Web 应用里的 JSP 页面、Servlet 等程序都将由 Web 服务器调用，JSP、Servlet 之间通常不会相互调用，那么 JSP、Servlet 如何交换数据

- 为了解决这个问题，几乎所有 Web 服务器(包括 Java、ASP、PHP、Ruby 等)都会提供 4 个类似 Map 的结构，分别是 application、session、request、page，并允许 JSP、Servlet 将数据放入这 4 个类似 Map 的结构中，并允许从这 4 个 Map 结构中取出数据
- application：对于整个 Web 应用有效，一旦 JSP、Servlet 将数据放入 application 中，该数据将可以被应用下其他所有的 JSP、Servlet 访问
- session：仅对一次会话有效，一旦 JSP、Servlet 将数据放入 session 中，该数据将可以被本次会话的其他所有 JSP、Servlet 访问
- request：仅对本次请求有效，一旦 JSP、Servlet 将数据放入 request 中，该数据将可以被该次请求的其他 JSP、Servlet 访问
- page：仅对当前页面有效，一旦 JSP、Servlet 将数据放入 page 中，该数据只能被当前页面的 JSP 脚本、声明部分访问
- JSP 中的 application、session、request 和 pageContext4 个内置对象分别用于操作 application、session、request、page 范围中的数据

6、application 对象代表 Web 应用本身，因此使用 application 来操作 Web 应用相关数据，application 对象通常有如下两个作用

- 在整个 Web 应用的多个 JSP、Servlet 之间共享数据

- 访问 Web 应用的配置参数

7、application 还有一个重要用处：可用于获取 Web 应用的配置参数

2.6.2. config 对象

1、config 对象代表当前 JSP 配置信息，但 JSP 页面通常无须配置，因此也就不存在配置信息，所以 JSP 页面比较少用该对象

2、但是在 Servlet 中则用处相对较大，因为 Servlet 需要在 web.xml 文件中进行配置，可以指定配置参数

- JSP 被当做 Servlet 配置
- 为 Servlet 配置参数使用 init-param 元素，该元素可以接受 param-name 和 param-value 两个子元素

3、如果希望 JSP 页面可以获取 web.xml 配置文件中的配置信息，则必须通过为该 JSP 配置的路径来访问该页面，因为只有这样访问 JSP 页面才会让配置参数起作用

2.6.3. exception 对象

1、execption 对象是 Throwable 的实例，代表 JSP 脚本中产生的错误和异常，是 JSP 页面异常机制的一部分

2、在 JSP 脚本中无需处理异常，即使该异常是 checked 异常，事实上，JSP 脚本包含的所有可能出现的异常都可交给错误处理页面处理

3、exception 对象仅在异常处理页面中才有效

4、JSP 脚本和 HTML 部分都将转换成_jspService()方法里的执行性代码---这就是 JSP 脚本无须处理异常的原因：因为这些脚本已经处于 try 块中，一旦 try 块捕捉到 JSP 脚本的异常，并且_jspx_page_context 不为 null，就会由该对象来处理该异常

5、_jspx_page_context 对异常的处理也非常简单：如果该页面的 page 指令指定了 errorPage 属性，则将请求 forward 到 errorPage 属性指定的页面，否则使用系统页面来输出异常信息

2.6.4. out 对象

1、out 对象代表一个页面输出流，通常用于在页面上输出变量值及常量，一般在使用输出表达式的地方，都可以使用 out 对象来达到同样的效果

2.6.5. pageContext 对象

1、这个对象代表上下文，该对象主要用于访问 JSP 之间的共享数据，使用 pageContext 可以访问 page、request、session、application 范围的变量

2、pageContext 是 PageContext 类的实例，它提供了如下两个方法来访问 page、request、session、application 范围的变量

- getAttribute(String name)：取得 page 范围内的 name 属性
- getAttribute(String name,int scope)：取得指定范围内的 name 属性，其中 scope 可以是如下 4 个值
  - PageContext.PAGE_SCOPE：对应于 page 范围=1
  - PageContext.REQUEST_SCOPE：对应于 request 范围=2

- PageContext.SESSION_SCOPE：对应于 session 范围=3
- PageContext.APPLICATION_SCOPE：对应于 application 范围=4

3、与 getAttribute()方法对应，PageContext 也提供了两个对应的 setAttribute()方法，用于将指定变量放入 page、request、session、application 范围内

4、pageContext 还可以用于获取其他内置对象，pageContext 对象包括如下方法

- ServletRequest getRequest()：获取 request 对象
- ServletResponse getResponse()：获取 response 对象
- ServletConfig getServletConfig()：获取 config 对象
- ServletContext getServletContext()：获取 application 对象
- HttpSession getSession()：获取 session 对象
- 因此一旦在 JSP、Servlet 编程中获取了 pageContext 对象，就可以通过它提供的上面方法来获取其他内置对象

2.6.6. request 对象

1、request 对象是 JSP 中重要的对象，每个 request 对象封装这一次用户请求，并且所有的请求参数都被封装在 request 对象中，因此 request 对象是获取请求参数的重要途径

2、获取请求头/请求参数

- Web 应用是请求/响应架构的应用，浏览器发送请求时通常总会附带一些请求头，还可能包含一些请求参数发送给服务器，服务器端负责解析请求头/请求参数的就是 JSP 或 Servlet，而 JSP 和 Servlet 取得请求参数的途径就是 request
- request 是 HttpServletRequest 接口的实例，它提供了如下方法来获取请求参数
  - String getParameter(String paramName)：获取 paramName 请求参数的值
  - Map getParameterMap()：获取所有请求参数名和参数值所组成的 Map 对象
  - Enumeration getParameterNames()：获取所有请求参数名组成的 Enumeration 对象
  - String[] getParameterValue(String name)：paramName 请求参数的值，当该请求参数有多个值时，该方法将返回多个值所组成的数组
- HttpServletRequest 提供了如下方法来访问请求头
  - String getHeader(String name)：根据指定请求头的值
  - java.util.Enumeration<String>getHeaderNames()：获取所有请求头的名称
  - java.util.Enumeration<String>getHeaders(String name)：获取指定请求头的多个值
  - int getIntHeader(String name)：获取指定请求头的值，并将该值转为整型数值
- 对开发人员来说，请求头和请求参数都是用户发送到服务器的数据
  - 区别在于请求头通常由浏览器自动添加，因此一次请求总是包含若干个请求头

- 请求参数则通常需要开发人员控制添加，让客户端发送请求参数通常分为两种方法
  - GET 方式的请求：直接在浏览器地址输入访问地址所发送的请求或提交表单发送请求时，该表单对应的 form 元素没有设置 method 属性，或设置 method 属性为 get。**GET 方式的请求会将请求参数的名和值转换成字符串，并附加在原 URL 之后，因此可以在地址栏中看到请求参数名和值**，且 GET 请求传送的数据量较小，一般不能大于 2KB
  - POST 方式的请求：这种方式通常使用提交表单(由 form HTML 元素表示)的方式来传送，**且需要设置 form 元素的 method 属性为 post**。POST 方式传送的数据量较大，通常认为 POST 请求参数的大小不受限制，但往往取决于服务器的限制，POST 请求传输的数据量总比 GET 传输的数据量大，而且 POST 方式发送的请求参数以及对应的值放在 HTML HEADER 中传输，用户不能再地址栏里看到请求参数值，安全性相对较高
  - 通常应该采用 POST 方式发送请求

3、几乎每个网站都会大量使用表单，表单用于收集用户信息
- 一旦用户提交请求，表单的信息将会提交给对应的处理程序
- 如果 form 元素设置 method 属性为 post，则表示发送 POST 请求
- 并不是每个表单域都会生成请求参数，而是由 name 属性的表单域才生成请求参数
- 表单域和请求参数的关系如下
  - 每个有 name 属性的表单域对应一个请求参数
  - 如果有多个表单域有相同的 name 属性，则多个表单域只生成一个请求参数，只是该参数有多个值
  - 表单域的 name 属性指定请求参数名，value 指定请求参数值
  - 如果某个表单域设置了 disabled="disabled"属性，则该表单域不再生成请求参数

4、操作 request 范围的属性
- HttpServletRequest 还包含如下两个方法，用于设置和获取 request 范围的属性
  - setAttribute(String attName,Object attValue)：将 attValue 设置成 request 范围的属性
  - Object getAttribute(String attName)：获取 request 范围的属性

5、执行 forward 或 include
- request 还有一个功能就是执行 forward 和 include，也就是代替 JSP 所提供的 forward 和 include 动作指令
- HttpServletRequest 类提供了一个个台 RequestDispatcher(String path)方法，其中 path 就是希望 forward 或者 include 的目标路径，该方法返回 **RequestDispatcher**，该对象提供了如下两个方法
  - forward(ServletRequest request,ServletResponse response)：执行 forward
  - include(ServletRequest request,ServletResponse response)：执行 include

2.6.7. response 对象

1、response 代表服务器对客户端的响应

- 大部分时候，程序无须使用 response 来响应客户端请求，因为有个更简单的响应对象---out，它代表页面输出流，直接使用 out 生成响应更简单
- out 是 JspWriter 的实例，JspWriter 是 Writer 的子类，**Writer 是字符流，无法输出非字符内容**
- 如果需要在 JSP 页面中动态生成一副位图，或者输出一个 PDF 文档，使用 out 作为响应对象将无法完成，此时必须使用 response 作为响应输出
- 此外，还可以使用 response 来重定向请求，以及用于向客户端增加 Cookie

2、response 响应生成非字符响应

- 对于需要生成非字符响应的情况，就应该使用 response 来响应客户端请求
- response 是 HttpServletResponse 接口的实例，该接口提供了一个 getOutputStream 方法，该方法返回响应输出字节流

3、重定向

- 重定向是 response 的另一个用处，与 forward 不同的是，重定向会丢失所有请求参数和 request 范围的属性，因为重定向将生成第二次请求，与前一次请求不再一个 request 范围内，所以发送一此请求的请求参数和 request 范围的属性全部丢失
- HttpServletResponse 提供了一个 sendRedirect(String path)方法，该方法用于重定向到 path 资源，即重定向 path 资源发送请求
- forward 动作和 redirect 动作有些相似：它们都将请求传递到另一个页面；它们的差异如下表

转发(forward)	重定向(redirect)
执行 forward 后依然是上一次请求	执行 redirect 后生成第二次请求
forward 的目标页面可以访问原请求的请求参数，因为依然是同一次请求，所有请求的请求参数、request 范围的属性全部存在	redirect 的目标页面不能访问原请求的请求参数，因为是第二次请求了，所有原请求的请求参数、request 范围的属性全部丢失
地址栏请求的 URL 不会改变	**地址栏改为重定向的目标 URL，相当于在浏览器地址栏里输入新的 URL 按回车键**

4、增加 Cookie

- Cookie 通常用于网站记录客户的某些信息，比如客户的用户名及客户的喜好等
- 一旦用户下次登录，网站可以获取到客户的相关信息，根据这些客户信息，网站可以对客户提供更友好的服务
- Cookie 与 session 的不同之处在于：session 会随浏览器的关闭而失效，但 Cookie 会一直存放在客户端机器上，除非超出 Cookie 的生命期限
- 由于安全原因，使用 Cookie 客户端浏览器必须支持 Cookie 才行，客户端浏览器完全可以禁用 Cookie
- 增加 Cookie 也是使用 response 内置对象完成的

- response 对象提供了如下方法
  - void addCookie(Cookie cookie)：增加 Cookie
- 使用步骤
  - 创建 Cookie 实例，Cookie 的构造器为 Cookie(String name,String value)
  - 设置 Cookie 的生命期限，即该 Cookie 在多长时间内有效
  - 向客户端写 Cookie
- 客户端访问 Cookie 使用 request 对象，**request 对象提供了 getCookies()方法，该方法将返回客户端机器上所有 Cookie 组成的数组**，遍历该数组的每个元素，找到希望访问的 Cookie 即可
- 默认情况下，Cookie 值不允许出现中文字符，如果需要值为中文内容的 Cookie 怎么办，可以借助 java.net.URLEncoder 先对中文字符串进行编码，然后将编码后的结果设为 Cookie 值，当程序要读取 Cookie 时，则应该先读取，然后使用 java.net.URLDecoder 对其进行解码

2.6.8. session 对象

1、session 对象也是一个非常常用的对象，这个对象代表依次用户会话

- 一次用户会话的含义是：客户端浏览器连接服务器开始，到客户端浏览器与服务器断开为止，这个过程就是一次会话
- session 通常用于跟踪用户的会话信息，如判断用户是否登陆系统，或者在购物车应用中，用于跟踪用户购买的商品
- session 范围内的属性可以在多个页面的跳转之间共享，**一旦关闭浏览器，即 session 结束，session 范围内的属性将全部丢失**
- session 对象是 HttpSession 的实例，HttpSession 有如下两个常用方法
  - getAttribute(String attName,Object attValue)：设置 session 范围内 attName 属性的值为 attValue
  - getAttribute(String attName)：返回 session 范围内 attName 属性的值

2、通常应该只把与用户会话状态相关的信息放入 session 范围内，不要仅仅为了两个页面的交互信息，就将信息放入 session 内，如果仅仅为了两个页面的交换信息，可以将信息放入 request，然后 forward 即可

3、session 机制通常用于保存客户端状态信息，这些状态信息需要保存到 Web 服务器硬盘上，**所以要求 session 里的属性值必须是可序列化的**，否则将会引发不可序列化的异常

- session 属性值可以使任何可序列化的 Java 对象

2.7. Servlet 介绍

1、JSP 的本质就是 Servlet

- 开发者把编写好的 JSP 页面部署在 Web 容器中，Web 容器会将 JSP 编译成对应的 Servlet
- 直接使用 Servlet 的坏处：Servlet 的开发效率非常低，特别是当使用 Servlet 生成表现层页面时，页面中的所有 HTML 标签，都需要采用 Servlet 的输出流来输出，因此极其繁琐
- 而且 Servlet 是标准的 Java 类，必须由程序员开发、修改，美工人员难以参与页面的开发

- 这一系列问题，都阻碍了 Servlet 作为表现层的使用

2、自 MVC 规范出现后，Servlet 的责任开始明确，仅仅作为控制器使用，不再需要生成页面标签，也不再作为视图层角色使用

2.7.1. Servlet 的开发

1、Servlet 通常被称为服务器端小程序，是运行在服务器端的程序，用于处理及响应客户端的请求

2、Servlet 是个特殊的 Java 类，必须继承 HttpServlet，每个 Servlet 可以响应客户端的请求，Servlet 提供不同的方法用于响应客户端请求

- doGet：用于响应客户端的 GET 请求
- doPost：用于响应客户端的 POST 请求
- doPut：用于响应客户端的 PUT 请求
- deDelete：用于响应客户端的 Delete 请求
- 通常，客户端的请求通常只有 GET 和 POST 两种，Servlet 为了响应这两种请求，必须重写 doGet()和 doPost()两个方法，总之需要哪种响应，就要提供哪种方法的实现
- 大部分时候，Servlet 对于所有请求的响应都是完全一样的，此时，可以采用重写一个方法来代替上面几个方法：只需重写 service()方法即可响应客户端的所有请求

3、另外，HttpServlet 还包含两个方法

- init(ServletConfig config)：创建 Servlet 实例时，调用该方法的初始化 Servlet 资源
- destroy()：销毁 Servlet 实例时，自动调用该方法的回收资源
- 通常无需重写 init()和 destroy()两个方法，除非需要在初始化 Servlet，完成某些资源初始化的方法，才考虑重写 init 方法；如果需要在销毁 Servlet 之前，完成某些资源的回收，比如关闭数据库连接等，才需要重写 destroy 方法

4、注意点

```
@WebServlet(name="firstServlet",urlPatterns={"/firstServlet"})
```

- 那么访问的 url 就是.../firstServlet
- 生成.class 文件后，必须要重启 Tomcat，暂时不知道为什么

5、Servlet 和 JSP 的区别

- Servlet 中没有内置对象，原来 JSP 中的内置对象都必须由程序显式创建
- 对于静态的 HTML 标签，Servlet 都必须使用页面输出流逐行输出

6、JSP 是 Servlet 的一种简化

- 使用 JSP 只需要完成程序员需要输出到客户端的内容，至于 JSP 脚本如何嵌入一个类中，由 JSP 容器完成
- 而 Servlet 则是个完整的 Java 类，这个类的 service()方法用于生成对客户端的响应
- 普通 Servlet 类里的 service()方法的作用，完全等同于 JSP 生成 Servlet 类的_jspService()方法

2.7.2. Servlet 的配置

1、编辑好的 Servlet 源文件并不能响应用户请求，还必须将其编译成 class 文件，将编译后的.class 文件放在 WEB-INF/classes 路径下，如果 Servlet 有包，还应该将 class 文件放在对应的包路径下

2、如果要直接使用 javac 命令编译 Servlet 类，则必须将 Servlet API 接口和类添加到系统的 CLASSPATH 环境变量里，即 Tomcat 安装目录下 lib 目录中的 servlet-api.java 和 jsp-api.jar 添加到 CLASSPATH 环境变量中

3、为了让 Servlet 能响应用户请求，还必须将 Servlet 配置在 Web 应用中，配置 Servlet 时，需要修改 web.xml 文件，从 Servlet3.0 开始，配置 Servlet 有两种方式

- 在 Servlet 类中使用@WebServlet 注解进行配置
- 通过在 web.xml 文件中进行配置

4、使用@WebServlet 时可指定下表的属性

属性	是否必须	说明
asyncSupported	否	指定该 Servlet 是否支持异步操作模式
displayName	否	指定该 Servlet 的显示名
initParams	否	用于为该 Servlet 配置参数
loadOnStartup	否	用于将 Servlet 配置成 load-on-startup 的 Servlet
name	否	指定该 Servlet
urlPatterns/value	否	这两个属性的作用完全相同，指定该 Servlet 处理的 url

5、打算使用注解来配置 Servlet，需要注意

- 不要在 web.xml 文件根元素 (<web-app.../>) 中指定 metadata-complete="true"
- 不要在 web.xml 文件中配置该 Servlet

6、如果打算使用 web.xml 来配置该 Servlet，则需要配置如下两个部分

- 配置 Servlet 的名字：对应 web.xml 文件中的<servlet/>元素
- 配置 Servlet 的 URL：对应 web.xml 文件中的<servlet-mapping/>元素，可选，但是如果没有为 Servlet 配置 URL，则该 Servlet 不能响应用户请求

7、注意

- Servlet、Filter、Listener 等相关配置，都可以采用 web.xml 配置或者注解配置
- 两种方式只能采用一种

2.7.3. JSP/Servlet 的生命周期

1、JSP 的本质就是 Servlet，开发者编写 JSP 页面将由 Web 容器编译成对应的 Servlet，当 Servlet 在容器中运行中，其实例的创建及销毁等都不是由程序员决定的，而是由 Web 容器进行控制的

2、创建 Servlet 实例有两个时机

- 客户端第一次请求某个 Servlet 时，系统创建该 Servlet 的实例：大部分 Servlet 都是这种 Servlet
- Web 应用启动时立即创建 Servlet 实例，即 load-on-startup Servlet

3、Servlet 的运行都遵循如下生命周期

- 创建 Servlet 实例
- Web 容器调用 Servlet 的 init 方法，对 Servlet 进行初始化
- Servlet 初始化后，将一直存在于容器中，用于响应客户端请求
  - 若客户端发送 GET 请求，则容器调用 Servlet 的 doGet 方法处理并响应请求
  - 若客户端发送 POST 请求，容器调用 Servlet 的 doPost 方法处理并响应请求
  - 或者统一使用 service()方法处理来响应用户请求
- Web 容器决定销毁 Servlet 时，先调用 Servlet 的 destroy 方法，通常关闭 Web 应用之时销毁 Servlet

2.7.4. load-on-startup Servlet

1、创建 Servlet 实例有两个时机：用户请求之时或应用启动之时

- 应用启动时就创建 Servlet，通常是用于某些后台服务的 Servlet，或者需要拦截很多请求的 Servlet
- 这种 Servlet 通常作为应用的基础 Servlet 使用，提供重要的后台服务

2、load-on-startup 的 Servlet 有两种方式

- 在 web.xml 文件中通过<servlet.../>元素的<load-on-startup.../>子元素进行配置
- 通过@WebServlet 注解 loadOnStratup 属性决定
  @WebServlet(loadOnStartup=1)

2.7.5. 访问 Servlet 的配置参数

1、配置 Servlet 时，还可以增加额外的配置参数，通过使用配置参数，可以实现提供更好的可移植性，避免将参数以硬编码的方式写在程序代码中

2、为 Servlet 配置参数有两种方式

- 通过@WebServlet 的 initParams 属性来指定
- 通过在 web.xml 文件的<servlet.../>元素中添加<init-param.../>子元素来指定

3、访问 Servlet 配置参数通过 ServletConfig 对象完成(JSP 的内置对象 config 就是此处的 ServletConfig)，ServletConfig 提供如下方法

- java.lang.String getInitParameter(java.lang.String name)：用于获取初始参数

4、@WebServlet 方式的例子

```
@WebServlet(name="testServlet",
   urlPatterns={"/testServlet"},
   initParams={
      @WebInitParam(name="driver",value="com.mysql.jdbc.Driver"),
      @WebInitParam(name="url",value="jdbc:mysql://localhost:3306/javaee"),
      @WebInitParam(name="user",value="root"),
      @WebInitParam(name="pass",value="hcflh19930101")
   })
```

- initParam 属性用于为 Servlet 配置参数
- initParam 属性值的每个@WebInitParam 配置一个初始化参数

- 每个@WebInitParam 可以指定两个属性
  - name：指定参数名
  - value：指定参数值

5、web.xml 方式配置的例子

```
<servlet>
  <!--配置 Servlet
  -->
  <servlet-name>testServletHCF</servlet-name>
  <!--指定 Servlet 的实现类-->
  <servlet-class>lee.TestServlet</servlet-class>
  <!--配置 Servlet 的初始化参数:driver-->
  <init-param>
    <param-name>driver</param-name>
    <param-value>com.mysql.jdbc.Driver</param-value>
  </init-param>
  <!--配置 Servlet 的初始化参数:url-->
  <init-param>
    <param-name>url</param-name>
    <param-value>jdbc:mysql://localhost:3306/javaee</param-value>
  </init-param>
  <!--配置 Servlet 的初始化参数:user-->
  <init-param>
    <param-name>user</param-name>
    <param-value>root</param-value>
  </init-param>
  <!--配置 Servlet 的初始化参数:pass-->
  <init-param>
    <param-name>pass</param-name>
    <param-value>hcflh19930101</param-value>
  </init-param>
</servlet>
<servlet-mapping>
  <!--确定 Servlet
  -->
  <servlet-name>testServletHCF</servlet-name>
  <!--配置 Servlet 映射的 URL-->
  <url-pattern>/testServlet</url-pattern>
</servlet-mapping>
```

2.7.6. 使用 Servlet 作为控制器

1、Servlet 作为表现层有如下三个劣势

- 开发效率低，所有 HTML 标签都需要使用页面输出流完成
- 不利于团队协作开发，美工人员无法参与 Servlet 界面的开发
- 程序可维护性差，即使修改一个按钮的标题，都必须重新编辑 Java 代码，并重新编译

2、在标准的 MVC 模式中，Servlet 仅作为控制器使用。Java EE 应用架构正是遵循 MVC 模式的，对于遵循 MVC 模式的 Java EE 应用而言，JSP 仅作为表现层(View)技术，其作用如下

- 负责收集用户请求参数
- 将应用的处理结果、状态数据呈献给用户

3、Servlet 则仅充当控制器(Controller)角色，它的作用类似于调度员：

- 所有用户请求都发给 Servlet
- Servlet 调用 Model 来处理用户请求，并调用 JSP 来呈现处理结果
- 或者 Servlet 直接调用 JSP 将应用的状态数据呈献给用户

4、Model 通常由 JavaBean 充当，所有业务逻辑、数据访问逻辑都在 Model 中实现

2.8. JSP2 的自定义标签

1、在 JSP 规范的 1.1 版中增加了自定义标签库规范，自定义标签库是一种非常优秀的表现层组件技术，通过使用自定义标签库，可以在简单的标签中封装复杂的功能

2、为什么要自定义标签：为了取代丑陋的 JSP 脚本，在 HTML 页面中插入 JSP 脚本有几个坏处

- JSP 脚本非常丑陋，难以阅读
- JSP 脚本和 HTML 代码混杂，维护成本高
- HTML 页面中嵌入 JSP 脚本，导致美工人员无法参与开发

3、出于以上三个原因，Web 开发需要一种可在页面中使用的标签，这种标签具有 HTML 标签类似的语法，但又可以完成 JSP 脚本的功能---这种标签就是 JSP 自定义标签

4、在 JSP 2 中开发标签库只需要几个步骤

- 开发自定义标签处理类
- 建立一个*.tld 文件，每个*.tld 文件对应一个标签库，每个标签库可包含多个标签
- 在 JSP 文件中使用自定义标签

5、所有的 MVC 框架，如 Struts2、SpringMVC、JSF 等都提供了丰富的自定义标签

2.8.1. 开发自定义标签类

1、自定义标签应该继承自一个父类：java.servlet.jsp.tagext.SimpleTagSupport，除此之外，还应该有如下的要求

- 如果标签类包含属性，每个属性都有对应的 getter 和 setter 方法
- **重写 doTag()方法，这个方法负责生成页面内容**

2.8.2. 建立 TLD 文件

1、TLD 是 Tag Library Definition 的缩写，即标签库定义，文件的后缀是 tld，每个 TDL 文件对应一个标签库，一个标签库中可包含多个标签，TLD 文件也称为标签库定义文件

2、标签库定义文件的根元素是 taglib，它可以包含多个 tag 子元素，每个 tag 子

元素都定义一个标签

- 通常可以到 Web 容器下复制一个标签库定义文件，并在此基础上进行修改即可
- 可以复制到 WEB-INF 的任意子路径下

3、taglib 有如下三个子元素

- tlib-version：指定该标签库实现的版本，这是一个作为标志的内部版本号，对程序没有太大的作用
- short-name：该标签库的默认短名，该名称通常也没有太大用处
- **uri：这个属性非常重要，它指定该标签库的 URI，相当于指定该标签库的唯一标志，JSP 页面中使用标签库时就是根据该 URI 属性来定位标签库**
- tag：taglib 元素下可以包含多个 tag 元素，每个 tag 元素定义一个标签，tag 元素允许出现以下元素
  - **name：该标签的名称，这个子元素很重要，JSP 页面中就是根据该名称来使用此标签的**
  - tag-class：指定标签的处理类，毋庸置疑，这个元素非常重要，它指定了标签由哪个标签处理类来处理
  - body-content：这个元素也很重要，指定标签体内容，可以包含如下子元素
    - tagdependent：指定标签处理类自己负责处理标签体
    - empty：指定该标签只能作为空标签使用
    - scriptless：指定该标签的标签体可以是静态 HTML 元素，表达式语言，但不允许出现 JSP 脚本
    - JSP：指定改变前的标签体可以使用 JSP 脚本(JSP 2 规范不再推荐使用 JSP 脚本，所以 JSP 2 自定义标签的标签体中不能包含 JSP 脚本，所以 body-content 元素的值不能是 JSP)
- 将标签库文件放在 Web 应用的 WEB-INF 路径或任意自路径下，Java Web 规范会自动加载该文件，则该文件定义的标签库也将生效

2.8.3. 使用标签库

1、在 JSP 页面中确定指定的标签需要两点

- 标签库 URI：确定使用哪个标签库
- 标签名：确定使用哪个标签

2、使用标签库分成以下两个步骤

- 导入标签库：使用 taglib 编译指令导入标签库，就是将标签库和指定前缀关联起来
  - 语法格式如下
    <%@ taglib uri="tagliburi" prefix="tagPrefix" %>
  - uri 属性指定标签库的 URI，这个 URI 可以确定一个标签库
  - prefix 属性指定标签库前缀：所有使用该前缀的标签由此标签库处理
- 使用标签：在 JSP 页面中使用自定义标签
  - 语法格式如下
    <tagPrefix:tagName tagAttribute="tagValue" ...>
    <tabBody/>

</tagPrefix:tagName>
or
<tagPrefix:tagName tagAttribute="tagValue" ...>

2.8.4. 带属性的标签

1、带属性的标签

- 带属性的标签必须为每个属性提供对应的 setter 和 getter 方法

2.8.5. 带标签体的标签

1、带标签体的标签，可以在标签体内嵌入其他内容(包括静态 HTML 内容和动态 JSP 内容)，通常用于完成一些逻辑运算，例如判断和循环等

2.8.6. 以页面片段作为属性的标签

1、以"页面片段"为属性的标签与普通标签区别并不大，只有两个简单的改变

- 标签处理类中定义类型 JspFragment 的属性，该属性代表了"页面片段"
- 使用标签库时，通过<jsp:attribute.../>动作指令为标签的属性值定值

2.8.7. 动态属性的标签

1、动态属性的标签比普通标签多了两个额外要求

- 标签处理类还需要实现 DynamicAttributes 接口
- 配置标签时通过<dynamic-attributes.../>子元素指定该标签支持动态属性

2.9. Filter 介绍

1、Filter 可认为是 Servlet 的一种"加强版"，它主要用于对用户请求进行预处理，也可以对 HttpServletResponse 进行后处理

2、Filter 也可对用户请求生成响应，这一点与 Servlet 相同，但实际上很少会使用 Filter 向用户请求生成响应

3、使用 Filter 的完整流程：

- Filter 对用户请求进行预处理
- 接着将请求交给 Servlet 进行处理并生成响应
- Filter 再对服务器响应进行后处理

4、Filter 有如下几个作用

- 在 HttpServletRequest 到达 Servlet 之前，拦截客户的 HttpServletRequest
- 根据需要检查 HttpServletRequest，也可以修改 HttpServletRequest 头和数据
- 在 HttpServletResponse 到达客户端之前，拦截 HttpServletResponse
- 根据需要检查 HttpServletResponse，也可以修改 HttpServletResponse 头和数据

5、Filter 有如下几个类

- 用户授权的 Filter：Filter 负责检查用户请求，根据请求过滤用户非法请求
- 日志 Filter：详细记录某些特殊的用户请求
- 负责解码的 Filter：包括对非标准编码的请求解码
- 能改变 XML 内容的 XSLT Filter

- Filter 可负责拦截多个响应或请求，一个请求或响应也可以被多个 Filter 拦截

6、**创建一个 Filter 只需要两个步骤**

- 创建 Filter 处理类
- web.xml 文件中配置 Filter

2.9.1. 创建 Filter 类

1、创建 Filter 必须实现 javax.servlet.Filter 接口，在该接口中定义了如下三个方法

- void init(FilterConfig config)：用于完成 Filter 的初始化
- void destroy()：用于 Filter 销毁前，完成某些资源的回收
- void doFilter(ServletRequest request,ServletResponse response,FilterChain chain)：实现过滤功能，该方法就是对每个请求和相应增加额外处理

2.9.2. 配置 Filter

1、配置 Filter 与配置 Servlet 非常类似

- 需要两个部分
  - 配置 Filter
  -
  - 配置 Filter 拦截的 URL 模式
- 区别在于：Servlet 通常只配置一个 URL，而 Filter 可以同时拦截多个请求的 URL。因此在配置 Filter 的 URL 模式时通常会使用模式字符串，使得 Filter 可以拦截多个请求
- 与配置 Servlet 相似的是，配置 Filter 同样有两种方式
  - 在 Filter 类通过注解进行配置
  - 在 web.xml 文件中通过配置文件进行配置

2、@WebFilter 修饰一个 Filter 类，用于对 Filter 进行配置，它支持如下表的属性

属性	是否必须	说明
asyncSupported	否	指定该 Filter 是否支持异步操作模式
dispatcherTypes	否	指定该 Filter 仅对那种 dispatcher 模式的请求进行过滤，该属性支持 ASYNC、ERROR、FORWARD、INCLUDE、REQUEST 这 5 个值的任意组合
diaplsyName	否	指定该 Filter 的显示名
filterName		指定该 Filter 的名称
initParams	否	用于为该 Filter 配置参数
servletNames	否	该属性值可以指定多个 Servlet 的名称，用于指定该 Filter 仅对这几个 Servlet 执行过滤
urlPattenrs/value	否	这两个属性的作用完全相同

3、实际上，Filter 与 Servlet 极其相似，区别只是 Filter 的 doFilter()方法里多了个一个 FilterChain 的参数，通过该参数可以控制是否放行用户请求

- 在实际项目中，Filter 里 doFilter()方法里的代码就是从多个 Servlet 的 service()方法里抽取的通用代码

4、Filter 和 Servlet 具有完全相同的生命周期行为，且 Filter 也可以通过<init-param.../>元素或@WebFilter 的 initParams 属性来配置初始化参数，获取 Filter 的初始化参数则使用 FilterConfig 的 getInitParameter()方法

2.9.3. 使用 URL Rewrite 实现网站伪静态

1、对于以 JSP 为表现层开发的动态网站来说，用户访问的 URL 通常有如下形式

- xxx.jsp?param=value...

2、大部分搜索引擎都会优先考虑收录静态 HTML 页面，而不是动态的*.jsp、*.php 页面，但事实上，大部分网站都是动态的，不可能全是静态的 HTML 页面，因此互联网上大部分网站都会考虑使用伪静态---就是将*.jsp、*.php 这种动态 URL 伪装成静态的 HTML 页面

3、对于 Java Web 应用来说，要实现伪静态非常简单：可以通过 Filter 拦截所有发向*.html 请求，然后按某种规则将请求 forward 到实际的*.jsp 页面即可

- 现有的 URL Rewrite 开源项目为这种思路提供了实现，使用 URL Rewrite 实现网站伪静态也很简单

2.10. Listener 介绍

1、当 Web 应用在 Web 容器中运行时，Web 应用内部会不断地发生各种事情：如 Web 应用被启动，Web 应用被停止，用户 session 开始、用户 session 结束、用户请求到达等，通常来说，这些 Web 事件对开发者是透明的

2、Servlet API 提供了大量监听器来监听 Web 应用的内部事件，从而允许当 Web 内部事件发生时回调监听器内的方法

3、使用 Listener 的步骤

- 定义 Listener 实现类
- 通过注解或在 web.xml 中配置 Listener

2.10.1. 实现 Listener 类

1、与 AWT 事件编程完全相似，监听不同 Web 事件的监听器也不同，常用的 Web 事件监听器有如下几个

- ServletContextListener：用于监听 Web 应用的启动和关闭
- ServletContextAttributeListener：用于监听 ServletContext 范围(application)内属性的改变
- ServletRequestListener：用于监听用户请求
- ServletRequestAttributeListener：用于监听 ServletRequest 范围(request)内属性的改变
- HttpSessionListener：用于监听用户 session 的开始和结束
- HttpSessionAttributeListener：用于监听 HttpSession 范围(session)内属性的改变

2、ServletContextListener 用于监听 Web 应用的启动和关闭，该 Listener 类必须实现 ServletContextListener 接口，该接口包含如下方法

- contextInitialized(ServletContextEvent sce)：启动 Web 应用时，系统调用 listener 的该方法
- contextDestroyed(ServletContextEvent sce)：关闭 Web 应用时，系统调用

Listener 的该方法

2.10.2. 配置 Listener

1、配置 Listener 只要向 Web 应用注册 Listener 实现类即可，无需配置参数之类的东西，十分简单

- 使用@WebListener 修饰 Listener 实现类即可
- 在 web.xml 文档中使用<listener.../>元素进行配置

2、使用 @WebListener 时通常无需指定任何属性，只需要使用该注解修饰 Listener 实现类即可向 Web 应用注册该监听器

3、在 web.xml 中使用<listener.../>元素进行配置时只要配置如下子元素即可

- listener-class：指定 Listener 实现类

2.10.3. 使用 ServletContextAttributeListener

1、ServletContextAttributeListener 用于监听 ServletContext(application)范围内属性的变化，实现该接口的监听器需要实现如下三个方法

- attributeAdded(ServletContextAttributeEvent event)：当程序把一个属性存入 application 范围时触发该方法
- attributeRemoved(ServletContextAttributeEvent event)：当程序把一个属性从 application 范围删除时触发该方法
- attributeReplaced(ServletContextAttributeEvent event)：当程序替换 applicaiton 范围内的属性时触发该方法

2.10.4. ServletReuqestListener 和 ServletRequestAttributeListener

1、ServletRequestListener 用于监听用户请求的到达，实现该接口的监听器需要实现如下两个方法

- requestInitialized(ServletRequestEvent sre)：用户请求到达、被初始化时触发该方法
- requestDestroyed(ServletRequestEvent sre)：用户请求结束、被销毁时触发该方法

2、ServletRequestAttributeListener 用于监听 ServletRequest(request)范围内属性的变化，实现该接口的监听器需要实现

- attributeAdded()
- attributeRemoved()
- attributeReplaced()

3、ServletContextAttributeListener 和 ServletRequestAttributeListener 作用相似，都是监听属性的变化

- ServletRequestAttributeListener 监听 request 范围内属性的变化
- ServletContextAttributeListener 监听 application 范围内属性的变化

4、应用程序完全可以采用一个监听器来监听多个事件，只要让该监听器实现类同时实现多个监听器接口即可

2.10.5. HttpSessionListener 和 HttpSessionAttributeListener

1、HttpSessionListener 用于监听用户 session 的创建和销毁，实现该接口的监听器需要实现如下方法

- sessionCreated(HttpSessionEvent se)：用户与服务器的会话开始、创建时触发该方法
- sessionDestroyed(HttpSessionEvent se)：用户与服务器的会话断开、销毁时触发该方法

2、HttpSessionAttributeListener 用于监听 HttpSession(session)范围内属性的变化，实现该接口的监听器需要实现如下三个方法

- attributeAdded()
- attributeRemoved()
- attributeReplaced()

3、HttpSessionAttributeListener 和 ServletContextAttributeListener 的作用相似，都用于监听属性的改变

- HttpSessionAttributeListener 监听 session 范围内属性的改变
- ServletContextAttributeListener 监听的是 application 范围内属性的改变

4、实现 HttpSessionListener 接口的监听器可以监听每个用户会话的开始和结束，因此应用可以通过监听器监听系统的在线用户

2.11. JSP2 特性

1、JSP 2.0 升级了 JSP 1.2 规范，新增了一些额外的特性，JSP 2.0 使得动态网页的设计更加容易，甚至可以无须学习 Java，也可以做出 JSP 页面。目前 Servlet 3.1 对应于 JSP 2.3 规范，JSP 2.3 也被统称为 JSP 2

2、JSP 2 主要增加了如下特性

- 直接配置 JSP 属性
- 表达式语言
- 简化的自定义标签 API
- Tag 文件语法

2.11.1. 配置 JSP 属性

2.11.2. 表达式语言

1、表达式语言是一种简化的数据访问方式，避免使用 JSP 脚本

${expression}

2、表达式语言的内置对象

- pageContext
- pageScope
- requestScope
- sessionScope
- applicationScpoe
- param
- paramValues
- header
- headerValues
- initParam
- cookie

3、表达式语言的自定义函数

- 表达式语言除了可以使用基本的运算符外，还可以使用自定义函数
- 开发步骤
  - 开发函数处理类：函数处理类就是普通类，包含若干个静态方法
  - 使用标签库定义函数

2.11.3. Tag File支持

1、TagFile是自定义标签的简化用法，使用Tag File可以无须定义标签处理类和标签库文件

2、建立Tag File的步骤

- 建立Tag文件，在JSP所支持的Tag File规范下，Tag File代理了标签处理类，它的格式类似于JSP文件。可以这样理解：如同JSP可以代替Servlet作为表现层一样，Tag File则可以代替标签处理类

2.12. Servlet 3.0新特性

1、Servlet 3.0规范是Servlet规范历史上最重要的变革之一，它的许多特性都极大地简化了Java Web应用的开发

2.12.1. Servlet 3.0的注解

1、Servlet 3.0的一个显著改变是"顺应"潮流，抛弃了采用web.xml配置Servlet、Filter、Listener的繁琐步骤，允许开发人员使用注解修饰它们，从而进行部署

2、Servlet 3.0规范在javax.servlet.annotation包下提供了如下注解

- @WebServlet：用于修饰一个Servlet类，用于部署Servlet类
- @WebInitParam：用于与@WebServlet或@WebFilter一起使用，为Servlet、Filter配置参数
- @WebFilter：用于修饰Filter类，用于部署Filter类
- @MultipartConfig：用于修饰Servlet，指定该Servlet将会负责处理multipart/form-data类型的请求
- @ServletSecurity：这是一个与JAAS有关的注解，修饰Servlet指定该Servlet的安全与授权控制
- @HttpConstraint：用于与@ServletSecurity一起使用，用于指定该Servlet的安全与授权控制
- @HttpMethodConstrant：用于与@ServletSecurity一起使用，用于指定该Servlet的安全与授权控制

2.12.2. Servlet 3.0的Web模块支持

1、Servlet 3.0规范不再要求所有Web组件(如Servlet、Listener、Filter)都部署在web.xml文件中，而是允许采用"Web模块"来部署，管理他们

2、一个Web模块通常对应于一个JAR包

3、Web模块与普通JAR的最大区别在于需要在META-INF目录下添加一个web-fragment.xml文件，这个文件也被称为Web模块部署描述符

- web-fragment.xml文件被称为Web模块部署描述符
- web-fragment.xml文件与web.xml文件的作用、文档结构基本相似，因为

它们都用于部署、各种 Web 组件

- web-fragment.xml 可以指定如下两个元素
  - <name.../>：用于指定该 Web 模块的名称
  - <ordering.../>：用于指定加载该 Web 模块的相对顺序

2.12.3. Servlet 3.0 提供的异步处理

1、Servlet 3.0 规范引入了异步处理来解决这个问题，异步处理允许 Servlet 重新发起一条新线程去调用耗时的业务方法，这样就可以避免等待

2、Servlet 3.0 的异步处理是通过 AsyncContext 类来处理的，Servlet 可以通过 ServletRequest 的如下两个方法开启异步调用、创建 AsyncContext()对象

- AsyncContext startAsync()
- AsyncContext startAsync(ServletRequest,ServletResponse)

3、异步监听器需要实现 AsyncListener 接口，需要实现如下四个方法

- onStartAsync(AsyncEvent event)：当异步调用开始时触发该方法
- onComplete(AsyncEvent event)：当异步调用完成时触发该方法
- onError(AsyncEvent event)：当异步调用出错时触发该方法
- onTimeout(AsyncEvent event)：当异步调用超过时触发该方法

2.12.4. 改进的 Servlet API

1、Servlet 3.0 还有一个改变是改进了部分 API，这种改进很好地简化了 Java Web 开发，其中两大改进是

- HttpServletRequest 增加了对文件上传的支持
- ServletContext 允许通过编程的方式动态注册 Servlet、Filter

2、HttpServletRequest 提供了如下两个方法来处理文件上传

- Part getPart(String name)：根据名称来获取文件上传
- Collection<Part> getParts()：获取所有的文件上传域
- 上述两个方法返回的值都涉及一个 API：Part，每个 Part 对象对应于一个文件上传域，该对象提供了大量方法来访问上传文件的文件类型、大小、输入流等

2.13. Servlet 3.1 新增的非阻塞式 IO

1、Servlet 3.1 新特性包括强制更改 session Id(由 HttpServletRequest 的 changeSessionId()方法提供、非阻塞 IO 等。尤其是 Servlet 3.1 提供的非阻塞 IO 进行输入、输出，可以更好地提升性能

2、Servlet 底层的 IO 是通过如下两个 IO 流来支持

- ServletInputStream：Servlet 用于读取数据的输入流
- ServletOutputStream：Servlet 用于输出数据的输出流

3、传统的方式采用阻塞式 IO--当 Servlet 读取浏览器提交的数据时，如果数据暂时不用，或者数据没有读取完成，Servlet 当前所在线程将会被阻塞，无法继续向下执行

4、从 Servlet 3.1 开始，ServletInputStream 新增了一个 setReadListener(ReadListener readListener)方法，该方法允许以非阻塞 IO 读取数据，实现 ReadListener 监听器需要实现如下三个方法

- onAllDateRead()：当所有数据读取完成时激发该方法
- onDataAvailable()：当所有数据可用时激发该方法
- onError(Throwable t)：读取数据出现错误时激发该方法

5、Servlet 中使用非阻塞 IO 的步骤
- 调用 ServletRequest 的 startAsync()方法开启异步模式
- 通过 ServletRequest 获取 ServletInputStream，并为 ServletInputStream 设置监听器
- 实现 ReadListener 接口来实现监听器，在该监听器的方法中以非阻塞的方式读取数据

6、说白了就是用一个新线程来作为 IO 的处理???

2.14. Tomcat 8 的 WebSocket 支持

1、严格来说，WebSocket 并不属于 Java Web 相关规范，WebSocket 属于 HTML 5 规范的一部分，WebSocket 语序通过 JavaScript 建立与远程服务器的连接，从而允许远程服务器将数据推送给浏览器

2、通过使用 WebSocket，可以构建出实时性要求比较高的应用，比如在线游戏、在线证券、设备监控、新闻在线播报等，只要服务器端有了新数据，服务器端就可以直接将数据推送给浏览器，让浏览器显示最新的状态

3、WebSocket 规范已经相当成熟，而且各种主流浏览器都已经支持 WebSocket 技术，Java EE 规范则提供了 WebSocket 服务端规范，而 Tomcat 8 则对该规范提供了优秀的实现

4、使用 Tomcat 8 开发 WebSocket 服务端非常简单，大致有如下两种方式
- 使用注解方式开发，被 @ServerEndpoint 修饰的 Java 类即可作为 WebSocket 服务端
- 继承 Endpoint 基类实现 WebSocket 服务端

5、开发被@ServerEndpoint 修饰的 Java 类之后，该类中还可以定义如下方法
- 被@OnOpen 修饰的方法：当客户端与该 WebSocket 服务端建立连接时激发该方法
- 被@OnClose 修饰的方法：当客户端与该 WebSocket 服务端断开连接时激发该方法
- 被@OnMessage 修饰的方法：当 WebSocket 服务端收到客户端消息时激发该方法
- 被@OnError 修饰的方法：当客户端与该 WebSocket 服务端连接出现错误时激发该方法

6、**开发 WebSocket 服务端程序步骤**
- 定义@OnOpen 修饰的方法，每当客户端连接进来时激发该方法，程序使用集合保存所有连接进来的客户端
- 定义@OnMessage 修饰的方法，每当该服务端收到客户端消息时激发该方法，服务端收到消息之后遍历保存客户端的集合，并将消息逐个发送给客户端
- 定义@OnClose 修饰的方法，每当客户端断开与该服务端连接时激发该方法，程序将该客户端从集合中删除

Chapter 3. Struts2 的基本用法

1、与传统的 Struts1 相比

- Struts2 允许使用普通的、传统的 Java 对象作为 Action
- Action 的 execute()方法不在于 Servlet API 耦合，因而更易测试
- 支持更多的视图技术
- 基于 AOP 思想的拦截器机制，提供了极好的可扩展性
- 更强大、更易用的输入校验功能
- 整合的 Ajax 支持等

3.1. MVC 思想概述

1、MVC(Model View Controller)思想将应用中各组件按功能进行分类，不同的组件使用不同技术充当，甚至推荐了严格分层，不同组件被严格限制在其所在层内，各层之间以松耦合的方式组织在一起，从而提供良好的封装

3.1.1. 传统 Model 1 和 Model 2

1、在 Model 1 模式下，整个 Web 应用几乎全部由 JSP 页组成，JSP 页面接受处理客户端请求，对请求处理后直接作出响应，用少量的 JavaBean 来处理数据库链接、数据库访问等操作

2、Model 1 模式的实现比较简单，适用于快速开发小规模项目，从工程化角度来看，它的局限性非常明显：JSP 页面身兼 View 和 Controller 两种角色，将控制逻辑和表现逻辑混杂在一起，从而导致代码重用性非常低，增加了应用扩展性和维护难度

3、Model 2 架构中

- Servlet 作为前段控制器，负责接收客户端发送的请求
- 在 Servlet 中只包含控制逻辑和简单的前端处理
- 然后调用后端 JavaBean 来完成实际的处理逻辑
- 最后转发到相应的 JSP 页面处理显示逻辑

4、在 Model 2 模式下

- 模型(Model)由 JavaBean 充当
- 视图(View)由 JSP 页面充当
- 控制器(Controller)由 Servlet 充当

5、Model 2 提供了更好的可扩展性以及可维护性，但增加了前期开发成本，从某种程度上讲，Model 2 为了降低系统后期维护的复杂度，却导致前期开发的更高复杂度

3.1.2. MVC 思想及其优势

1、MVC 并不是 Java 语言特有的设计思想，也不是 Web 应用特有的思想，它是所有面向对象程序设计语言应该遵守的规范

2、MVC 思想将一个应用分成三个基本部分：

- Model(模型)
- View(视图)
- Controller(控制器)

3、在 MVC 模式中，事件由控制器处理，控制器根据事件类型改变模型或视图

- 每个模型对应一系列的视图列表，这种对应关系通常采用注册来完成
- 当模型发生改变时，模型向所有注册过的视图发送通知，接下来，视图从对应的模型中获得信息，然后完成视图显示的更新

4、从设计模式的角度来看，MVC 思想非常类似于观察者模式，但与观察者模式存在少许差别

- 观察者模式下观察者和被观察者可以是两个互相对等的对象
- 对于 MVC 思想而言，被观察者往往只是单纯的数据体，而观察者则是单纯的的视图页面

5、MVC 有如下特点

- 多个视图可以对应一个模型，按 MVC 设计模式，一个模型对应多个视图，可以减少代码的复制以及代码的维护量，一旦模型发生改变，也易于维护
- 模型返回的数据与显示逻辑分离，模型技术可以应用于任何的显示技术，例如使用 JSP 页面，Velocity 模板或者直接产生 Excel 文档
- 应用被分隔三层，降低了各层之间的耦合，提供了应用的可扩展性
- 控制层的概念也很有效，由于它把不同的模型和不同的视图组合在一起，完成不同的请求，因此，控制层可以说是包含了用户请求权限的概念
- MVC 更符合软件工程化管理的精神，不同的层各司其职，每一层的组件具有相同的特征，有利于通过工程化和工具化产生管理程序代码

3.2. Struts2 的下载和安装

1、如何添加 User Library

- Eclipse->Preferences(Mac 上是偏好设置)
- Java->Build Path->User Libraries
- new->新建自己的 Library

2、如何使用 User Library

- 右键工程->Build Path->Configure Build Path
- Add Library...
- 选择你自己的 Library 即可

3、如何通过 Eclipse 自动获取依赖的包(.jar)

- 将依赖项(*.jar)放入%工程根目录%/WebContent/WEB-INF/lib/目录下
- 在 Eclipse 左侧目录中空白处右击->refresh，然后包便可自动添加

4、**我用 User Library 添加依赖项无法运行，用 Eclipse 自动获取路径下的依赖项，应用可以运行，目前不知道为什么，以后解决吧**

3.2.1. 手动配置(非 IDE)

1、路径结构总览

```
<Web 应用
>
|----WEB-INF
        |----src
        |----lib
        |---web.sml
        |----classes
```

|----struts.xml

- WEB-INF 文件夹包含三个子文件夹，分别为 src，class，lib，以及 web.xml
- classes 文件夹内包含了配置文件 struts.xml

```
<action name="*">
    <result>/{1}.jsp</result>
</action>
```

  - 以上配置内容非常重要，暂时不是特别明白

3.3. Struts2 的流程

3.3.1. Struts2 应用的开发步骤

1、**在 web.xml 文件中定义核心 Filter 来拦截用户请求**

- 由于 Web 应用是基于请求/响应架构的应用，所以不管哪个 MVC Web 框架，都需要在 web.xml 中配置该框架的核心 Servlet 或 Filter，这样才可以让该框架介入 Web 应用中

```
<filter>
    <filter-name>struts2</filter-name>
    <filter-class>org.apache.struts2.dispatcher.ng.
        filter.StrutsPrepareAndExecuteFilter</filter-class>
</filter>
<!-- 让 Struts2 的核心 Filter 拦截所有请求  -->
<filter-mapping>
    <filter-name>struts2</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

2、**如果需要以 POST 方式提交请求，则定义包含表单数据的 JSP 页面，如果仅仅只是以 GET 方式发送请求，则无需经过这一步**

3、**定义处理用户请求的 Action 类**

- 这一步是所有 MVC 框架中必不可少的，因为这个 action 就是 MVC 中的 C，也就是控制器，该控制器负责调用 Model 里的方法来处理请求
- MVC 框架的底层机制：
  - 核心 Servlet 或 Filter 接收到用户请求后，通常会对用户请求进行简单预处理，例如解析、封装参数等，然后通过反射来创建 Action 实例，并调用 Action 的指定方法(Struts1 通常是 execute，Struts2 可以是任意方法)来处理用户请求
  - 如何知道创建哪个 Action 实例
    - 利用配置文件：例如配置 login.action 对应使用 LoginAction 类，这就可以让 MVC 框架知道创建哪个 Action 的实例了
    - 利用约定，这种用法可能是受 Rails 框架的启发，例如约定 xxx.action 总是对应 XxxAction 类，如果核心控制器接收到 regist.action 请求后，会调用 RegistAction 类来处理用户请求
- 在 MVC 框架中，控制器实际上由两个部分共同组成，即拦截所有用户请求，处理请求的通用代码都由核心控制器完成，而实际的业务控制(诸如调用 Model，返回处理结果等)则由 Action 处理

4、**配置 Action**

- 对于 Java 领域的绝大部分 MVC 框架而言，都非常喜欢使用 XML 文件来配置管理，这在以前是一种思维定势
- 配置 Action 就是指定哪个请求对应哪个 Action 进行处理，从而让核心控制器根据该配置来创建合适的 Action 实例，并调用该 Action 的业务控制方法

```
<action name="login" class="org.crazyit.app.action.LoginAction">
    <!-- 定义三个逻辑视图和物理资源之间的映射 -->
    <result name="error">/WEB-INF/content/error.jsp</result>
    <result name="success">/WEB-INF/content/welcome.jsp</result>
</action>
```

5、**配置处理结果和物理视图之间的对应关系**

- 当 Action 处理用户请求结束后，通常会返回一个处理结果(通常使用简单的字符串就可以了)，可以认为该名称就是逻辑视图名
- 这个逻辑视图名需要和指定物理视图资源关联才有价值

3.3.2. Struts2 的流程

1、流程图详见 P184

3.4. Struts2 的常规配置

1、虽然 Struts2 提供了 Convention 插件来管理 Action、结果映射，但对于大部分实际开发来说，通常还是会考虑使用 XML 文件来管理 Struts2 的配置信息

2、Struts2 的默认配置文件名为 struts.xml，该文件应该放在 Web 应用的类加载路径下，通常就是放在 WEB-INF/classes 路径下

3、struts.xml 配置文件最大的作用就是配置 Action 和请求之间的对应关系，并配置逻辑视图名和物理视图资源之间的对应关系，除此之外，struts.xml 文件还有一些额外的功能，例如 Bean 配置，配置常量、导入其他配置文件等

3.4.1. 常量配置

1、Struts2 除了使用 struts.xml 文件来管理配置之外，还可以使用 struts.properties 文件来管理常量，该文件定义了 Struts2 框架的大量常量，开发者可以通过改变这些常量来满足应用的需求

2、struts.properties 文件是一个标准的 Properties 文件，该文件包含了系列的 key-value 对，每个 key 就是一个 Struts2 常量，该 key 对应的 value 就是一个 Struts2 常量值

3、Struts2 常量相当于对 Struts2 应用整体其作用的属性，因此 Struts2 常量常常也被称为 Struts2 属性

4、只要将 struts.properties 文件放在 Web 应用的类加载路径下，Struts2 框架就可以加载该文件，通常将该文件放在 Web 应用的 WEB-INF/classes 路径下

5、以下是 struts.properties 中定义的 Struts2 常量

- **struts.configuration**：该常量指定加载 Struts2 配置文件的配置管理器
  - 该常量默认值为 org.apache.struts2.config.DefaultConfiguration
  - 如果需要实现自己的配置管理器，开发者则可以实现一个

Configuration 接口的类，该类可以自己加载 Struts2 配置文件

- struts.locale：指定 Web 应用的默认 Locale，默认 Locale 是 en_US
- struts.i18n.encoding：指定 Web 应用的默认编码集，默认值为 UTF-8
- ...详见 P186-187

6、Struts2 默认会加载类加载路径下的 struts.xml、struts-default.xml、struts-plugin.xml 三类文件

- struts.xml 是开发者定义的默认配置文件
- struts-default.xml 是 Struts2 框架自带的配置文件
- struts-plugin.xml 则是 Struts2 插件的默认配置文件

7、Struts2 配置方式总共有三种

- 通过 struts.properties 文件
- 通过 struts.xml 配置文件
- 通过 Web 应用的 web.xml 文件

8、Struts2 的所有配置文件，包括 struts-default.xml、struts-plugin.xml，甚至用户自定义的、只要能被 Struts2 加载的配置文件中都可以使用常量配置的方式来配置 Struts2 常量

9、**通常推荐在 struts.xml 文件中定义 Struts2 属性**

10、**通常，Struts2 框架按如下搜索顺序加载 Struts2 常量**

- strtus-defaut.xml：该文件保存在 struts2-core-x.x.x.jar 中
- struts-plugin.xml：该文件保存在 struts2-Xxx-x.x.x.jar 等 Struts2 插件 jar 文件中
- struts.xml：该文件是 Web 应用默认的 Struts2 配置文件
- struts.properties：该文件是 Struts2 默认的配置文件
- web.xml：该文件是 Web 应用的配置文件

11、**如果在多个配置文件中配置了同一个 Struts2 常量，则后一个文件中的配置的常量值会覆盖前面文件中配置的常量值**

12、**不同文件中配置常量的方式是不同的，但不管在哪个文件中，配置 Struts2 常量都需要指定两个属性，常量 name 和常量 value，分别指定 Struts2 属性名和属性值**

13、struts.xml 模板：P189

3.4.2. 包含其他配置文件

1、为了避免 struts.xml 文件过于庞大、臃肿，提高 struts.xml 文件的可读性，可以将 struts.xml 配置文件分解成多个配置文件，然后在 struts.xml 文件中包含其他配置文件

- 语法如下

```
<struts>
    ...
    <include file="struts-part1.xml"/>
    ...
</struts
```

- 将 Struts2 的所有配置文件都放在 Web 应用的 WEB-INF/classes 路径下，struts.xml 文件包含了其他配置文件，struts.xml 文件由 Struts2 框架负责

加载，从而可以将所有配置信息都加载进来

3.5. 实现 Action

1、对于 Struts2 应用来说，Action 才是核心

- 开发者需要提供大量的 Action，并在 struts.xml 文件中配置 Action
- Action 类里包含了对用户请求的处理逻辑，Action 类也被称为业务控制器

2、相对于 Struts1 而言，**Struts2 采用了低浸入式的设计，Struts2 不要求 Action 类继承任何 Struts2 基类，或者实现任何 Struts2 接口**

- 在这种设计方式下，Struts2 的 Action 类是一个普通的 POJO(通常应该包含一个无参数的 execute 方法)，从而有很好的代码复用性

3、Struts2 通常直接使用 Action 来封装 HTTP 请求参数，因此，**Action 类里还应该包含与请求参数对应的实例变量**，并且为这些实例变量提供对应的 setter 和 getter 方法

4、关于 getter 和 setter

- 假设 HTTP 有一个包含 user 和 pass 的请求参数
- 那么系统会调用 setUser()/getUser()以及 setPass()/getPass()来作用 Action 类，因此 Action 里面的域的名字是否是 user 或 pass 完全不重要，甚至没有也可以

3.5.1. Action 接口和 ActionSupport 基类

1、如下是 Action 接口的代码

```
public interface Action{
    public static final String ERROR="error";
    public static final String INPUT="input";
    public static final String LOGIN="login";
    public static final String NONE="none";
    public static final String SUCCESS="success";
    public String execute() throws Exception;
}
```

- 这五个字符串的作用是统一 execute 方法的返回值

2、ActionSupport 是一个默认的 Action 实现类(P194)，该类里已经提供了许多默认方法，包括获取国际化信息的方法、数据校验的方法、默认的处理用户请求的方法等

- 实际上，ActionSupport 类是 Struts2 默认的 Action 处理类，如果让开发者的 Action 类继承该 ActionSupport 类，则会大大简化 Action 的开发
- ActionSupport 完全符合一个 Action 的要求，所以也可以直接使用 ActionSupport 作为业务控制器，如果配置 Action 没有指定 class 属性(即没有用户提供 Action 类)，系统自动使用 ActionSupport 类作为 Action 处理类

3.5.2. Action 访问 Servlet API

1、Strtus2 的 Action 没有与任何 Servlet API 耦合，这是 Struts2 的一个改良之处，由于 Action

再与 Servlet API 耦合，从而能更轻松地测试该 Action

2、对于 Web 应用的控制器而言，不访问 Servlet API 几乎是不可能的，例如跟踪 HTTP Session 状态等

- Web 应用中通常需要访问的 Servlet API 就是 HttpServletRequest、HttpSession 和 ServletContext，这三个接口分别代表 JSP 内置对象中的 request、session 和 application
- **Struts2 提供了一个 ActionContext 类，Struts2 的 Action 可以通过该类来访问 Servlet API**，以下是 ActionContext 类中包含的几个常用方法
  - Object get(Object key)：该方法类似于调用 HttpServletRequest 的 getAttribute(String name)方法
  - Map getApplication()：返回一个 Map 对象，该对象模拟了该应用的 ServletContext 实例
  - static ActionContext getContext()：静态方法，获取系统的 ActionContext 实例
  - Map getParameters()：获取所有的请求参数，类似于调用 HttpServletRequest 对象的 getParameterMap()方法
  - Map getSession()：返回一个 Map 对象，该 Map 对象模拟了 HttpSession 实例
  - void setApplication(Map application)：直接传入一个 Map 实例，将该 Map 实例里的 key-value 对转换成 application 的属性名、属性值
  - void setSession(Map session)：直接传入一个 Map 实例，将该 Map 里的 key-value 对转化成 session 的属性名属性值

3.5.3. Action 直接访问 Servlet API

1、虽然 Struts2 提供了 ActionContext 来访问 Servlet API，但这种访问毕竟不是直接获得 Servlet API 的实例，为了在 Action 中直接访问 Servlet API，Struts2 还提供了如下几个接口

- ServletContextAware：实现该接口的 Action 可以直接访问 Web 应用的 ServletContext 实例
- ServletRequestAware：实现该接口的 Action 可以直接访问用户请求的 HttpServeltRequest 实例
- ServletResponseAware：实现该接口的 Action 可以直接访问服务器响应的 HttpServletResponse 实例

2、ServletResponseAware 接口仅要求实现如下方法

public void setServletResponse(HttpServletResponse response)

- 方法参数 response 就代表了 Web 应用对客户端的响应
- 在 setServletResponse(HttpServletResponse response)方法内访问到 Web 应用的响应对象，**并将该对象设置成 Action 的实例变量(通过 this.response=response 来获取传入的参数)，从而允许在 execute 方法中访问该 HttpServletResponse 对象**

3、ServletRequestAware 接口仅要求实现如下方法

public void setServletRequest(HttpServletRequest request)

- 通过该方法即可访问代表用户请求的 HttpServletRequest 对象

4、ServletContextAware 接口仅要求实现如下方法

public void setServletContext(ServletContext context)

- 通过该方法即可访问到代表 Web 应用的 ServletContext 对象

3.5.4. ServletActionContext 访问 Servlet API

1、为了能直接访问 Servlet API，Struts2 还提供了一个 ServletActionContext 工具类，这个类包含了如下几个静态方法

- static PageContext getPageContext()：取得 Web 应用的 PageContext 对象
- static HttpServletRequest getRequest()：取得 Web 应用的 HttpServletRequest 对象
- static HttpServletResponse getResponse()：取得 Web 应用的 HttpServletResponse 对象
- static ServletContext geServletContext()：取得 Web 应用的 ServletContext 对象

2、借助于 ServletActionContext 类的帮助，开发者也可以在 Action 中访问 Servlet API，并可避免 Action 类需要实现 XxxAware 接口---虽然如此，但该 Action 依然与 Servlet API 直接耦合，一样不利于高层次的解耦

3、借助于 ServletActionContext 工具类的帮助，Action 能以更简单的方式来访问 Servlet API

3.5.5. ActionContext

1、ActionContext 是 Action 的上下文，Struts2 自动在其中保存了一些在 Action 执行过程中所需的对象，比如 session, parameters, locale 等。Struts2 会根据每个执行 HTTP 请求的线程来创建对应的 ActionContext，即一个线程有一个唯一的 ActionContext。因此，使用者可以使用静态方法 ActionContext.getContext()来获取当前线程的 ActionContext，也正是由于这个原因，使用者不用去操心让 Action 是线程安全的

2、无论如何，ActionContext 都是用来存放数据的。Struts2 本身会在其中放入不少数据，而使用者也可以放入自己想要的数据。**ActionContext 本身的数据结构是映射结构，即一个 Map**，用 key 来映射 value。所以使用者完全可以像使用 Map 一样来使用它，或者直接使用 Action.getContextMap()方法来对 Map 进行操作

3、Struts2 本身在其中放入的数据有 ActionInvocation、application(即 ServletContext)、conversionErrors、Locale、action 的 name、request 的参数、HTTP 的 Session 以及值栈(ValueStack)等

4、**由于 ActionContext 的线程唯一和静态方法就能获得的特性**，使得在非 Action 类中可以直接获得它，而不需要等待 Action 传入或注入。需要注意的是，**它仅在由于 request 而创建的线程中有效(因为 request 时才创建对应的 ActionContext)**，而在服务器启动的线程中(比如 fliter 的 init 方法)无效。由于在非 Action 类中访问其的方便性，**ActionContext 也可以用来在非 Action 类中向 JSP 传递数据(因为 JSP 也能很方便的访问它)**

5、ValueStack 与 ActionContext 的联系和区别

- **相同点**：它们都是在一次 HTTP 请求的范围内使用的，即它们的生命周期

都是一次请求

- **不同点**：值栈是栈的结构，ActionContext 是映射(Map)的结构
- **联系**：
  - ValueStack.getContext()方法得到的 Map 其实就是 ActionContext 的 Map
  - 查看 Struts2 的源代码可知(Struts2.3.1.2 的 org.apache.struts2.dispatcher.ng.PrepareOperations 的第 79 行 createActionContext 方法)，在创建 ActionContext 时，就是把 ValueStack.getContext()作为 ActionContext 的构造函数的参数
  - 所以，ValueStack 和 ActionContext 本质上可以互相获得
  - 在一些文档中，**会出现把对象存入"Stack Context"的字样，其实就是把值存入了 ActionContext。所以在阅读这些文档时，要看清楚，到底是放入了栈结构(即值栈)，还是映射结构(值栈的 context，即 ActionContext)**

6、如何获得 ActionContext

- **在自定义的拦截器中**：使用 ActionInvocation.getInvocationContext()或者使用 ActionContext.getContext()
- **在 Action 类中**：让拦截器注入或者使用 ActionContext.getContext()
- **在非 Action 类中**：让 Action 类传递参数、使用注入机制注入或者使用 ActionContext.getContext()
- 注意：只有运行在 request 线程中的代码才能调用 ActionContext.getContext()，否则返回的是 null
- 在 JSP 中：一般不需要获得 ActionContext 本身

7、如何向 ActionContext 中存入值

- 在拦截器、Action 类、非 Action 类等 Java 类中：使用 ActionContext.put(Object key, Object value)方法
- 在 JSP 中：标签<s:set value="..."/>默认将值存入 ActionContext 中(当然，<s:set>标签还可以把值存到其他地方)。另外，许多标签都有 var 属性(以前用的是 id 属性，现在 id 属性已被弃用)，这个属性能向 ActionContext 存入值，key 为 var 属性的值，value 为标签的 value 属性的值。(有些文档写的是向 ValueStack 的 context 存入值，其实是一样的)

8、如何从 ActionContext 中读取值

- 在拦截器、Action 类、非 Action 类等 Java 类中：使用 ActionContext.get(Object key)方法。
- 在 JSP 中：使用#开头的 Ognl 表达式，比如<s:property value="#name"/>会调用 ActionContext.get("name")方法。注意：如果某标签的属性默认不作为 Ognl 表达式解析，则需要使用%{}把表达式括起来，于是就会出现类似"%{#name}的表达式"
  - 如果是 <s:property value="#name.age"/> 会调用 ActionContext.get("name").getAge()方法
- 总之，在 JSP 中使用 ActionContext 一方面是由于它是映射结构，另一方面是能读取 Action 的一些配置。当你需要为许多 Action 提供通用的值的话，可以让每个 Action 都提供 getXXX()方法，但更好的方法是在拦截器或 JSP 模板中把这些通用的值存放到 ActionContext 中(因为拦截器或 JSP

模板往往通用于多个 Action)

3.6. 配置 Action

1、实现了 Action 处理类之后，就可以在 struts.xml 文件中配置该 Action 了

- 配置 Action 就是让 Struts2 知道哪个 Action 处理哪个请求，也就是完成用户请求和 Action 之间的对应关系
- 可以把 Action 当成 Struts2 的基本"程序单位"

3.6.1. 包和命名空间

1、Struts2 使用包来组织 Action，因此 Action 定义放在包定义下完成

- 定义 Action 通过使用<package.../>下的<action.../>子元素来完成
- 每个 package 元素配置一个包
- ???啥意思

2、Struts2 框架中核心组件就是 Action、拦截器等，Struts2 框架使用包来管理 Action 和拦截器等，每个包就是多个 Action、多个拦截器、多个拦截器引用的集合

3、配置<package.../>元素

- 必须指定 name 属性，这个属性是引用该包的唯一标识
- 除此之外，还可以指定一个可选的 extends 属性，extends 属性值必须是另一个包的 name 属性，指定 extends 属性表示让该包继承另一个包，子包可以从一个或多个父包中继承到拦截器、拦截器栈、action 等配置
- 另外，Struts2 还提供一种所谓的抽象包，抽象包意味着该包不能包含 Action 的定义，为了显式指定一个包是抽象包，可以为该<package.../>元素增加 abstract="true"属性

4、在 struts.xml 文件中，<package.../>用于定义包配置，每个<package.../>元素定义了一个包配置，可以指定如下几个属性

- name：必须属性，该属性指定该包的名字，该名字是该包给其他包引用的 key
- extends：可选属性，该属性指定该包继承其他包，继承其他包可以继承其包中的 Action 定义、拦截器定义等
- namespace 属性：可选属性，该属性定义该包的命名空间
- abstract：可选属性，它指定该包是否为一个抽象包，抽象包中不能含有 Action 定义

5、Struts2 的配置文件是从上到下处理的，因此父包应该在子包前面定义

6、**开发者定义的 package 通常应该继承 struts-default 包**

- struts2-core-x.x.x.x.jar 中包含一个 struts-default.xml 文件
- 该包是一个抽象包，包含了大量结果类型定义、拦截器定义、拦截器引用定义等，这些定义是配置普通 Action 的基础

7、Struts2 之所以提供命名空间，主要是为了处理同一个 Web 应用中包含同名 Action 的情形

- Strtus2 以命名空间的方式来管理 Action，同一个命名空间里不能有同名的 Action，不同的命名空间里可以有同名的 Action
- Struts2 不支持为单独的 Action 设置命名空间，而是通过为包指定

namespace 属性来为包下面的所有 Action 指定共同的命名空间

- 如果配置<package.../>时没有指定 namespace 属性，则该包下所有 Actioin 处于默认的包空间下
- 当某个包指定了命名空间后，该包下所有的 Action 处理的 URL 应该是命名空间/Action
- http://localhost:8888/<Web 应用名>/<命名空间名>/<Action>
- Struts2 还可以显式指定根命名空间，通过设置某个包的 namespace="/"来指定根命名空间
- 默认命名空间和根命名空间的区别
  - 默认命名空间里的 Action 可以处理任何命名空间下的 Action 请求，例如，存在/barspace/bar.action 的请求，并且/barspace 的命名空间下没有名为 bar 的 Action，那么默认命名空间下名为 bar 的 Action 也会处理用户请求
  - 根命名空间下的 Action 只能处理根命名空间下的 Action 请求
- 命名空间只有一个级别，例如请求的 URL 是/bookservice/search/get.action，系统将先在/bookservice/search 的命名空间下查找名为 get 的 Action，如果找不到，则直接进入默认的命名空间中查找名为 get 的 Action，而不会在/bookservice 中继续查找

3.6.2. Action 的基本配置

1、定义 Action 时至少需要指定该 Action 的 name 属性，**该 name 属性既是该 Action 的名字，也指定了该 Action 所处理的请求的 URL**

- 通常 name 属性由数字和下划线组成
- 如果需要在 name 属性中使用斜线"/"，则需要指定 struts.enable.SlashesinActionNames 常量为 true
- 不推荐在 name 中使用点"."和中划线"-"，它们可能会引发未知异常

2、通常还需要为 action 元素指定一个 class 属性，该属性指定了该 Action 的实现类。class 属性并不是必须的，如果不为<action.../>元素指定 class 属性，系统则默认使用系统的 ActionSupport 类

3、配置逻辑视图和物理试图之间的映射关系是通过<result.../>元素来定义的，每个<result.../>元素定义逻辑视图和物理视图的一次映射

3.6.3. 使用 Action 的动态方法调用

1、使用动态方法调用必须设置 Struts2 允许动态方法调用

- 设置 struts.enable.DynamicMethodInvocation 常量为 true
- Struts2 的动态方法调用存在一定安全方面的缺陷，尽量少用

3.6.4. 指定 method 属性及使用通配符

1、例如一个表单包含多个提交按钮(登陆/注册/找回等等)，需要分别提交给不同的控制逻辑，除了动态方法调用之外，Struts2 还提供了一种处理方法，即将一个 Action 处理类定义成多个逻辑 Action

2、在配置<action.../>元素时，可以将它指定 method 属性，则可以让 Action 调用**指定方法**，而不是**默认的 execute()**方法来处理用户请求

- 通过这种方式可以将一个 Action 类定义成多个逻辑 Action，即 Action 类的每个处理方法都映射成一个逻辑 Action
- **前提是这些方法具有相似的方法签名：方法形参列表为空，返回值为 String**

3、于是，配置 Action 时，实际上可认为需要配置三个属性

- name 属性指定该 Action 处理怎样的请求，代表了请求的 URL，不可省略
- class 属性指定该 Action 的处理类，如果省略该 class 属性，则默认使用 **ActionSupport** 作为处理类
- method 属性指定使用哪个方法处理请求，如果省略该 method 属性，则默认使用 **execute()**方法处理请求

4、Struts2 还有另一种简化形式：使用通配符的方式

```
<action name="*Action" class="<packagename>.LoginRegistAction"
    method="{1}">
    ...
</action>
```

- *：代表一个或多个任意字符
- 对于上述配置，只要用户请求是*Action.action 的模式，都可以使用该 Action 来处理，**其中{1}代表第一个*所匹配的值**

5、重新理解以下配置片段

```
<action name="*" >
    <result>/WEB-INF/content/{1}.jsp</result>
</action>
```

- Action 的名字是*，即匹配任意的 Action，所有用户请求都可以通过该 Action 处理
- 由于没有指定 class 属性，即该 Action 使用 ActionSupport 来作为处理类，而且因为该 ActionSupport 类的 execute 方法返回 success 字符串，即该 Action 总是直接返回 result 中指定的 JSP 资源
- 通过这种方法可以避免让浏览者直接访问系统的 JSP 页面，而是让 Struts2 框架来管理所有用户请求

6、对于只是简单的超级链接请求，可以通过定义 name="*"的 Action 实现，除此之外，Struts2 还允许在容器中定义一个默认的 Action，当用户请求的 URL 在容器中找不到对应的 Action 时，系统将使用默认 Action 来处理用户请求

7、除非请求的 URL 与 Action 的 name 属性绝对相同，否则将按先后顺序来决定由哪个 Action 来处理用户请求，因此应该将 name="*"的 Action 配置在最后

- 例如存在这样两个 Action

```
<action name="*abc" ....> ... </action>
<action name="*" ...>...</action>
```

- 对于请求 babc 来说，name="*abc"并不会比 name="*"具有更高的优先级，而是哪个 action 在前就由哪个 action 处理
- 如果存在 name="babc"的 action，那么该 babc 请求一定由该 action 处理，而忽略另外两个包含通配符的 action，即使该 action 处于配置文件最后

3.6.5. 配置默认 Action

1、为了让 Struts2 的 Action 可以接管用户请求

- 可以配置 name="*"的 Action
- 除此之外，Struts2 还支持配置默认 Action，当用户请求找不到对应的 Action 时，系统默认的 Action 即将处理用户请求

2、配置默认 Action 通过<default-action-ref.../>标签完成

- 配置该元素时需要指定一个 name 属性，该属性指向容器中另一个有效 Action，该 Action 将成为该容器中的默认 Action

3、将默认 Action 配置在默认命名空间里就可以让该 Action 处理所有用户请求，因为默认命名空间的 Action 可以处理任何命名空间的请求

3.6.6. 配置 Action 的默认处理类

1、配置<action.../>元素可以不指定 class 属性，如果没有指定 class 属性，则系统默认使用 ActionSupport 作为 Action 处理类

2、Struts2 允许定义开发者自己配置 Action 的默认处理类，配置 Action 的默认处理类使用<default-class-ref.../>元素，配置该元素时只需指定一个 class 属性，该 class 属性指定的类就是 Action 的默认处理类

3.7. 配置处理结果

1、Action 只是 Struts2 控制器的一部分，所以它不能直接生成对浏览者的响应

- Action 只负责处理请求，负责生成响应的视图组件，通常就是 JSP 页面
- 必须使用<result.../>元素进行配置，该元素定义逻辑视图名和物理视图资源之间的映射关系

3.7.1. 理解处理结果

1、Action 处理完用户请求之后，将返回一个普通字符串，整个普通字符串就是一个逻辑视图名，Struts2 通过配置逻辑视图名和物理视图之间的映射关系，一旦系统收到 Action 返回的某个逻辑视图名，系统就会把对应的物理视图呈现给浏览者

2、关系图见 P212

3、相对于 Struts1 框架而言，Struts2 的逻辑视图不再是 ActionForward 对象，而是一个普通字符串，这样的设计更有利于将 Action 类与 Struts2 框架分离，提供了更好的代码复用性

4、Struts2 还支持多种结果映射

- 转向实际资源时，实际资源可以是 JSP 视图资源，也可以是 FreeMarker 视图资源
- 还可以将请求转给下一个 Action 处理，形成 Action 的链式处理

3.7.2. 配置结果

1、Struts2 的 Action 处理用户请求结束后，返回一个普通字符串---逻辑视图名，必须在 struts.xml 文件中完成逻辑视图和物理视图资源的映射，才可以让系统转到实际的视图资源

2、配置结果是告诉 Struts2 框架：当 Action 处理结束后，系统下一步做什么，

系统下一步应该调用哪个物理视图资源来显示处理结果

3、Struts2 在 struts.xml 文件中使用<result.../>元素来配置结果，根据<result.../>元素所在位置不同，Struts2 提供了两种结果

- 局部结果：将<result.../>作为<action.../>元素的子元素配置
- 全局结果：将<result.../>作为<global-results.../>元素的子元素配置

4、一个<action.../>元素可以有多个<result.../>子元素，这表示一个 Action 可以对应多个结果

5、形式

- 最繁琐的形式

```
<result name="success" type="dispatcher">
    <param name="location">/WEB-INF/content/thank_you.jsp</param>
</result>
```

  - <param.../>子元素用于配置一个参数，参数名由 name 属性指定，name 属性可以有如下两个值
    - location：该参数指定了该逻辑视图对应的实际视图资源
    - parse：该参数指定了是否允许在实际视图名字中使用 OGNL 表达式，该参数默认为 true，通常不需要修改
- 通常无需指定 param 参数

```
<result name="success" type="dispatcher">/WEB-INF/content/thank_you.jsp
                                                    </result>
```

- Struts2 还允许省略指定结果类型，默认使用的类型就是 dispatcher(用于与 JSP 整合的结果类型)

```
<result name="success">/WEB-INF/content/thank_you.jsp</result>
```

- Struts2 还允许省略逻辑视图名，默认的逻辑视图名是 success

```
<result>/WEB-INF/content/thank_you.jsp</result>
```

3.7.3. Struts2 支持的结果类型

1、Struts2 支持多种视图技术，例如 JSP、Velocity 和 FreeMarker 等

- 当一个 Action 处理用户请求结束后，仅仅返回一个字符串，这个字符串是逻辑视图名，但该逻辑视图并未与任何的视图技术及任何的视图资源关联---直到在 struts.xml 文件中配置物理逻辑视图资源

2、Struts2 的结果类型要求实现 com.opensymphony.xwork2.Result

- 这个结果是所有结果类型的通用接口
- 如果开发者想实现自己的结果类型，也需要提供一个实现该接口的类，并且在 struts2.xml 文件中配置该结果类型
- 每个<result-type.../>元素定义一个结果类型，<result.../>元素中的 name 属性指定了该结果类型的名字，class 属性指定了该结果类型的实现类

3、Struts2 内建的支持结果类型如下

- chain 结果类型：Action 链式处理的结果类型
- dispatcher 结果类型：用于指定使用 JSP 作为视图的结果类型
- freemarker 结果类型：用于指定使用 Freearker 模板作为视图的结果类型
- httpheader 结果类型：用于控制特殊的 HTTP 行为的结果类型
- redirect 结果类型：用于直接跳转到其他 URL 的结果类型
- redirectAction 结果类型：用于直接跳转到其他 Action 的结果类型

- stream 结果类型：用于向浏览器返回一个 InputStream(一般用于文件下载)
- velocity 结果类型：用于指定使用 Velocity 模板作为视图的结果类型
- xslt 结果类型：用于与 XML/XSLT 整合的结果类型
- plainText 结果类型：用于显示某个页面的原始代码的结果类型

3.7.4. plainText 结果类型

1、这个结果类型并不常用，它的作用太过局限---主要用于显示实际视图资源的源代码

2、使用 plainText 结果类型时可指定如下两个参数(<param name="..."/>)

- location：指定实际的视图资源

  <param name="location">...</param>
- charSet：指定输出页面时所用的字符集

  <param name="charSet">...</param>

3.7.5. redirect 结果类型

1、这种结果类型与 dispatcher 结果类型相对

- dispatcher 结果类型，是将请求 **forward(转发)**到指定的 JSP 资源
- redirect 结果类型，则意味着将请求 **redirect(重定向)**到指定的视图资源
- 重定向会时区所有的请求参数、请求属性---当然也丢失了 Action 的处理结果

2、使用 redirect 结果类型的效果是，系统将调用 HttpServletResponse 的 sendRedirect(String)方法来重定向指定视图资源，这种重定向的效果就是重新产生一个请求

3、配置 redirect 结果类型时，可以指定如下两个参数(<param name="..."/>)

- location：该参数指定 Action 处理完用户请求后跳转的地址
- parse：该参数指定是否允许在 location 参数值中使用表达式，默认为 true，通常无需指定 parse 属性值，因此可以简化成如下形式

  <result type="redirect">/welcome.jsp</result>

4、注意

- **使用 redirect 类型的结果时，不能重定向到/WEB-INF/路径下的任何资源，因为重定向相当于重新发送请求，而 Web 应用下的/WEB-INF/路径下的资源是受保护资源**

3.7.6. redirectAction 结果类型

1、该结果类型与 redirect 类型非常相似，一样是重新生成一个全新的请求，但与 redirect 结果类型的区别在于

- redirectAction 使用 ActionMapperFactory 提供的 ActionMapper 来重定向请求
- redirect 常用于生成一个对具体资源的请求
- redirectAction 常用于生成对另一个 Action 请求

2、当需要让一个 Action 处理结束后，直接将请求重定向(重定向，而不是转发)到另一个 Action 就该使用这种类型

3、配置 redirectAction 结果类型时，可以指定如下两个参数(<param name="..."/>)

- actionName：该参数指定重定向的 Action
-
- namespace：该参数指定需要重定向的 Action 所在命名空间

3.7.7. 动态结果

1、动态结果的含义是指在指定实际视图资源时使用了表达式语法，通过这种语法可以允许 Action 处理完用户请求后，动态转入实际的视图资源

2、除了 Action 的 name 属性与 class 或 method 属性可以使用通配符以及表达式语法"{N}"外，也可以在配置<result.../>元素时使用表达式语法，从允许根据请求动态决定实际资源

3、与配置 class 属性和 method 属性相比，配置<result.../>元素时，还允许使用 OGNL 表达式，这种用法允许根据 Action 属性值来定位物理视图资源

3.7.8. Action 属性值决定物理视图资源

1、配置<result.../>时，不仅可以使用${0}表达式形式来指定视图资源，还可以使用${属性名}的方式来指定视图资源

- **${属性名}里的属性名就是对应 Action 实例里的属性，就是对应的 Action 类中包含一个该名字的域**
- 还允许使用 OGNL 表达式，即${属性名.属性名.属性名...}

3.7.9. 全局结果

1、在<global-results.../>元素中配置<result.../>时，该<result.../>元素配置了一个全局结果，全局结果将对所有 Action 都有效

2、如果一个 Action 里包含了与全局结果同名的结果，那么局部 Result 将会覆盖全局 Result，只有在 Action 里的局部结果里找不到逻辑视图对应的结果，才会到全局结果里搜索

3、<global-results.../>元素也配置在 struts.xml 文件中

3.7.10. 使用 PreResultListener

1、PreResultListener 是一个监听器接口，它可以在 Action 完成控制处理之后，系统转入实际的物理视图之间被回调

2、Struts2 应用可由 Action、拦截器添加 PreResultListener 监听器

- 添加 PreResultListener 监听器通过 ActionInvocation 的 addPreResultListener()方法完成
- 一旦为 Action 添加了 PreResultListener 监听器，该监听器就可以在应用转入实际物理视图之前回调该监听器的 beforeResult()方法
- 一旦为拦截器添加了 PreResultListener 监听器，该监听器会对该拦截器的所有 Action 都起作用

3、通过使用 PreResultListener 监听指定 Action 转入不同 Result 的细节，因此也可以作为日志的实现方式

3.8. 配置 Struts2 的异常处理

1、任何成熟的 MVC 框架都应该提供成熟的异常处理机制

- 我们可以在 execute 方法中手动捕捉异常，当捕捉到特定异常时，返回特定逻辑视图名---但这种方式非常繁琐，需要书写大量 catch 块，其最大的缺点还在于异常处理与代码耦合
- 最好的方式是通过声明式的方式管理异常处理

3.8.1. Struts2 的异常处理机制

1、如果手动处理异常，然后 return 一个字符串作为逻辑视图名，其实质就是完成异常类型和逻辑视图名之间的对应关系，既然如此，我们完全可以把这种对应关系推迟到 struts.xml 文件中进行管理

2、Action 接口里的 execute()方法签名

- public String execute() throws Exception
- 该方法可以抛出所有异常，这意味着重写该方法时，完全无须进行任何异常处理，而是把异常直接抛给 Struts2 框架处理
- Struts2 框架接收到 Action 抛出的异常之后，将根据 struts.xml 文件配置的异常映射，转入指定的视图资源

3、为了使用 Struts2 的异常处理机制，必须打开 Struts2 的异常映射功能，开启异常映射需要一个拦截器

- struts-default.xml 已经开启了 Struts2 的异常映射功能

3.8.2. 声明式异常捕捉

1、Struts2 的异常处理机制是通过在 struts.xml 文件中配置<exception-mapping.../>元素完成的，配置该元素时，需要指定如下两个属性

- exception：此属性指定该异常映射所设置的异常类型
- result：此属性指定 Action 出现该异常时，系统返回 result 属性对应的逻辑视图名

2、根据<exception-mapping.../>元素出现位置的不同，异常映射又分为如下两种

- 局部异常映射：将<exception-mapping.../>作为<action.../>元素的子元素配置
- 全局异常映射：将<exception-mapping.../>作为<global-exception-mappings>元素的子元素配置
- 局部异常映射优先级大于全局异常映射(如果配置了同一个异常类型，那么局部异常映射会覆盖全局异常映射)

3.8.3. 输出异常信息

1、当 Struts2 框架控制系统进入异常处理页面后，还需要在对应页面中输入指定异常信息

2、为了在异常处理页面中显示异常信息，可以使用 Struts2 的如下标签来输出异常信息

- <s:property value="exception"/>：输出异常对象本身
- <s:property value="exceptionStack"/>：输出异常堆栈信息
- <s:property value="exception.message"/>：输出异常的 message 信息

3、相对于 Struts1 只能输出异常对象的 message 属性值，而无法输出异常的跟踪栈信息，Struts2 能输出异常对象完整的跟踪栈信息，因此更加有利于项目调试

3.9. Convention 插件与"约定"支持

1、Struts2.1 开始，Struts2 引入了 Convention 插件来支持零配置，插件完全可以抛弃配置信息，不仅不需要使用 struts.xml 文件进行配置，甚至不需要使用 Annotation 进行配置，而是由 Struts2 根据约定来自动配置
2、约定优于配置

3.9.1. Action 的搜索和映射约定

1、为了使用 Convention 插件，必须在 Struts2 应用中安装 Convention 插件，安装 Convention 插件只需要将 Struts2 项目下的 struts2-convention-plugin-x.x.x.x.jar 文件复制到 Struts2 应用的 WEB-INF/lib 路径下即可
2、对于 Convention 插件而言，它会自动搜索位于 action、actions、struts、struts2 包下的所有 Java 类，Convention 插件会把如下两种 Java 类当成 Action 处理

- 所有实现了 com.opensymphony.xwork2.Action 的 java 类
- 所有类名以 Action 结尾的 Action 类

3、Struts2 的 Convention 插件还允许设置如下三个常量

- strtus.convention.exclude.packages：指定不扫描哪些包下的 Java 类，位于这些包结构下的 Java 类将不会被自动映射成 Java 类
- struts.convention.package.locators：Convention 插件使用该常量指定的包作为搜寻 Action 的根包
  - 对于 action.lee.LoginAction 类，按照原本约定应该映射到/lee/login
  - 如果将该常量设为 lee，则该 Action 将会映射到/login
- strtus.convention.action.packages：Convention 插件以该常量指定包作为根包来搜索 Action 类
  - Convention 插件除了扫描 action、actions、struts、struts2 四个包的类之外，还会扫描该常量指定的一个或多个包，Convention 会试图从中发现 Action 类

4、找到合适的 Action 之后，Convention 插件会按照约定部署这些 Action，部署 Action 时，actions、action、struts、struts2 包会映射成跟命名空间，而这些子包则被映射成对应的命名空间
5、Struts2 的 Action 都是以 package 的形式组织的，而 package 还有父 package，每个 Action 所处的 package 与其 Action 类所在的包名相似(除去 actions、action、struts、struts2 这些包及父包部分)

- 例如 org.crazyit.actions.books.GetBooks
- 这个 Java 类的包为 org.crazyit.actions.books
- 该 Java 类的包除去 actions、action、struts、struts2 这些包及父包部分剩下的就是 books
- 该 Action 对应的 package 为 books

6、Action 的 name 属性(也就是 Action 所要处理的 RUL)根据该 Action 的类名映射，

映射规则如下

- 如果该 Action 类名包含 Action 后缀，将该 Action 类名的 Action 后缀去掉，否则不做任何处理
- 将 Action 类名的驼峰写法(每个单词首字母大写、其他字母小写的写法)转成**中划线写法(所有字母小写，单词与单词之间以中划线隔开)**
- 例如
  - LoginAction===>login
  - GetBooks===>get-books
  - AddEmployeeAction===>add-employee

7、于是完整的 URL 就是<包名>/<Action 的 name>.action

- 该包名除去 actions、action、struts、struts2 这些包及父包部分
- 例子
  - org.crazyit.actions.books.GetBooks==>/books/get-books.action
  - org.crazyit.struts2.wage.hr.AddEmployeeAction==>
    /wage/hr/add-employee.action

3.9.2. 按约定映射 Result

1、Action 处理用户请求之后会返回一个字符串作为逻辑视图，该逻辑视图必须映射到实际的物理视图才有意义，Convention 默认也作为逻辑视图和物理视图之间的映射提供了约定

2、默认情况下，Convention 总会到 Web 应用的 WEB-INF/content 路径下定位物理资源，定位资源的约定是：actionName+resultcode+suffix

- 当某个逻辑视图找不到对应的视图资源时，Convention 会自动视图使用 actionName+suffix 作为物理视图资源
- 例如
  - org.crazyit.app.action.user.LoginAction 返回 success 字符串时，Convention 会优先考虑使用 WEB-INF/content/user 目录下的 login-success.jsp 作为视图资源，如果找不到该文件，login.jsp 也可作为对应的视图资源

3、为了看到 Struts2 应用里 Action 等各种资源的映射情况，Struts2 提供了 Config Browser 插件，这个插件并不是用来增强 Struts2 功能的，这个插件主要是更有利于开发者调试的，使用该插件可以清楚地看出 Struts2 应用下部署了哪些 Action，以及每个 Action 详细的映射情况

- 安装 Config Browser 插件：将 Struts2 项目的 lib 目录下的 struts2-config-browser-plugin-x.x.x.x.jar 文件复制到 Struts2 应用的 WEB-INF/lib 目录下，重启应用即可
- **千万不要放在 Tomcat 的 lib 下面!!!!!!!**
- 插件的地址为：http://localhost:8888/<Web 应用>/config-browser/actionNames.acion
- **注意 Config Brower 插件不是为 Convention 插件设计的，无论开发者使用 struts.xml 文件进行配置管理，还是使用 Convention 插件的约定法则管理 Action 和 Result，Config Browser 插件一样可用**

3.9.3. Action 链的约定

1、如果希望一个 Action 处理结束后不是进入视图页面，而是进入另一个 Action 形成 Action 链，则通过 Convention 插件只需遵守如下三个约定即可

- 第一个 Action 返回的逻辑视图字符串没有对应的视图资源
- 第二个 Action 与第一个 Action 处于同一个包下
- 第二个 Action 映射的 URL 为 firstactionName+resultcode

3.9.4. 自动重加载映射

1、由于 Convention 插件是根据 Action、JSP 页面来动态生成映射的，因此不管是 Action 改变还是 JSP 页面的改变，都需要 Convention 插件重新加载映射

2、Convention 插件完全支持自动冲加载映射，只要为 Struts2 应用配置如下两个常量即可

- <constant name="struts2.devMode" value="true"/>
- <constant name="struts.convention.classes.reload" value="true"/>

3.9.5. Convention 插件的相关常量

1、实际上，想真正让 Struts2 变成"零配置"还是有些难度的，至少要在 web.xml 文件中配置 Struts2 的核心 Filter，Struts2 应用的各种全局配置，如 Bean 配置、拦截配置等，依然还需要借助 Struts2 的配置文件

2、Convention 插件主要致力于解决 Action 管理、Result 管理等最常见的、最琐碎的配置，将开发者从庞大而繁琐的 struts.xml 文件中释放出来，而不是完全舍弃 struts.xml 文件

3.9.6. Convention 插件相关 Annotation

1、Struts2 的 Convention 插件主要集中在管理 Action 和 Result 映射之上，而 Struts2 的配置文件除了管理 Action、Result 之外，还需要管理拦截器、异常处理等相关信息，Convention 使用"约定"来管理这些配置，除此之外，Convention 还允许使用注解管理器 Action 和 Result 的配置，从而覆盖 Convention 的约定

3.10. 使用 Struts2 的国际化

1、现在的软件系统不再是简单的单机程序，往往都是一个开放系统，需要面对来自全世界各个地方的浏览者，因此国际化是商业系统中不可或缺的部分

2、Struts2 的国际化建立在 Java 国际化基础之上，通过提供不同国家/语言环境的消息资源，然后通过 ResourceBundle 加载指定 Locale 对应的资源文件，在取得该资源文件中指定 key 对应的消息---整个过程与 Java 程序国际化完全相同

3、Struts2 国际化的步骤如下

- 让系统加载国际化资源文件
  - 自动加载：Action 范围的国际化资源文件、包范围的国际化资源文件由系统加载
  - 手动加载：JSP 范围的国际化资源文件、全局范围的国际化资源文件，分别使用标签、配置常量的方式来手动加载
- 输出国际化
  - 在视图页面上输出国际化消息，需要使用 Struts2 的标签库

- 在 Action 类中输出国际化消息，需要使用 ActionSupport 的 getText()方法来完成

3.10.1. 视图页面的国际化

1、在 JSP 页面中指定国际化资源需要借助 Struts2 的另外一个标签：<s:i18n/>

- 如果把<s:i18n.../>标签作为<s:text.../>标签的父标签，则<s:text.../>标签将会直接加载<s:i18n.../>标签里指定的国际化资源文件
- 如果把<s:i18n.../>标签作为表单标签的父标签，则表单标签的 key 属性将会从国际化资源文件中加载该消息

2、国际化资源文件的位置在 WEB-INF/classes/路径下

3、<native2ascii>

- native2ascii -[options] [inputfile [outputfile]]
- -[options]：表示命令开关，有两个选项可供选择
  - -reverse：将 Unicode 编码转为本地或者指定编码，不指定编码情况下，将转为本地编码
  - -encoding encoding_name：指定编码，encoding_name 为编码名称，默认为本地编码
    - **在 Linux 平台或者 Mac 平台上，将 Windows 平台的以 GBK 编码的文件进行转换成 Unicode 编码时，必须明确指定编码，否则会以 utf-8 来转换，但实际是 GBK，因此是不正确的**
- [inputfile [outputfile]]
  - inputfile：表示输入文件全名
  - outputfile：输出文件名。如果缺少此参数，将输出到控制台

3.10.2. Action 的国际化

1、如果需要对 Action 以及 Action 的输入校验提示信息进行国际化，则可以为 Action 单独指定一份国际化资源文件

2、为 Action 单独指定国际化资源文件的方法：

- 在 Action 类文件所在的路径建立多个文件名为 ActionName_language_country.properties 的文件
- 一旦建立了这个系列的国际化资源文件，那么 Action 就可以访问该 Action 范围的资源文件了

3、Struts2 的国际化资源文件由系统自动加载，该 Action 类以及该 Action 对应的校验规则文件都可以使用这份国际化资源文件

4、国际化资源特征

- 国际化资源文件的 baseName 与 Action 类名相同
- 国际化资源文件与 Action 类的*.class 文件保存在同一个路径下

5、**Action 范围内的国际化资源消息可以通过如下三种方式来使用**

- 在 JSP 页面中输出国际化信息，可以使用 Struts2 的<s:text.../>标签，该标签可以指定一个 name 属性，该属性指定了国际化资源文件中的 key
- 如果想在表单元素的 label 中输出国际化信息，可以为该表单标签指定一个 key 属性，该属性的值为国际化资源的 key
- 为了在 Action 类中访问国际化消息，可以使用 ActionSupport 类的

getText()方法，该方法可以接受一个 name 参数，该参数指定了国际化资源文件中的 key

- 如果需要在 Action 的校验规则文件中使用 Action 范围的国际化消息，则可以通过<message.../>元素指定 key 属性来实现

3.10.3. 使用包范围的国际化资源文件

1、包范围的国际化资源文件的功能基本与 Action 范围的国际化资源文件的功能相似，Struts2 也可以自动加载包范围的国际化资源文件

2、与 Action 范围的国际化资源文件的区别是：**包范围的国际化资源文件可以被包下所有 Action 使用**

3、包范围的国际化资源文件的文件名：package_<language>_<country>.properties，一旦建立了多份这样的国际化资源文件，Struts2 会自动加载这些国际化资源文件，该包下的所有 Action 都可以访问这些资源文件

- **包范围的资源文件的 baseName 就是 package，不是 Action 所在的包名**
- **该文件只需要放在该包的根路径下即可**

4、当 Action 范围的资源文件和包范围的资源文件同时存在时，系统将优先使用 Action 范围的资源文件

5、推荐使用 Action 范围的国际化消息资源，可以提供更好的可维护性

3.10.4. 使用全局国际化资源

1、不论在 struts.xml 文件中配置常量，还是在 struts.properties 文件中配置常量，只需要配置 struts.custom.i18n.resources 常量即可加载全局国际化资源文件

- 配置 struts.custom.i18n.resources 常量时，该常量的值为国际化资源文件的 baseName

2、全局国际化消息资源可以被整个应用的所有组件(包括 JSP 页面、Action、Action 校验规则文件等使用，因此使用时比较方便)

3、一般来说，全局国际化消息资源文件中只应该保存那些对整个应用都有效的全局消息，比如类型转换失败的通用提示消息，文件上传失败的提示消息等等

4、使用

- 如果 JSP 页面中需要使用全句话资源文件里的国际化消息，直接通过<s:text.../>标签或表单标签的 key 属性来使用即可
- 为了在 Action 中访问国际化消息，则可以利用 ActionSupport 类的 getText()获取国际化消息

3.10.5. 输出代占位符的国际化消息

1、国际化消息可能包含占位符，这些占位符必须使用参数来填充

2、在 Struts2 中，提供了如下两种方式来填充消息字符串中的占位符

- 如果需要在 JSP 页面中填充国际化消息里的占位符，则可以通过在<s:text.../>标签中使用多个<s:param.../>标签来填充消息中的占位符
- 如果需要在 Action 中填充国际化消息中的占位符，则可以通过调用 getText(String aTextName,List args)或 getText(String key,String[] args)方法来填充占位符，即字符串数组或者 list 的第 n 个元素填充第 n 个占位符

3、Struts2 还提供了对占位符的一种替代方式，这种方式允许在国际化消息中使用表达式，对于这种方式，则可避免在使用国际化消息时还需要为占位符传入参数值

3.10.6. 加载资源文件的顺序

1、优先加载系统中保存在 ChidAction 的类文件相同位置，且 baseName 为 ChildAction 的系列资源文件
2、如果在 1 中找不到指定 key 对应的消息，且 ChildAction 有父类 ParentAction，则加载系统中保存在 ParentAction 的类文件相同位置，且 baseName 为 ParentAction 的系列资源文件
3、如果在 2 中找不到指定 key 对应的消息，且 ChildAction 有实现接口 IChildAction，则加载系统中保存在 IChildAction 的类文件相同位置，且 baseName 为 IChildAction 的系列资源文件
4、如果在 3 中找不到指定 key 对应的消息，且 ChildAction 有实现接口 ModelDriven(即使用模型驱动模式)，则对于 getModel()方法返回的 model 对象，重新执行 1 步操作
5、如果在 4 中找不到 key 对应的消息，则查找当前包下的 baseName 为 package 的系列资源文件
6、如果在 5 中找不到指定 key 对应的消息，则沿着当前包上溯，知道最顶层包来查找 baseName 为 package 的系列资源文件
7、如果在 6 中找不到指定 key 对应的消息，则查找 struts.custom.i18n.resources 常量指定的 baseName 的系列资源文件
8、如果经过上面步骤一直找不到该 key 对应的消息，将直接输出该 key 属性的值；如果在上面的步骤 1-7 的任何一步中找到 key 对应的消息，系统将停止搜索，直接输出该 key 对应的消息

9、对于 JSP 中访问国际化消息，则简单得多，它们又可分为两种形式

- 对于使用<s:i18n.../>标签作为父标签的<s:text.../>标签、表单标签形式
  1) 将从<s:i18n.../>标签作为父标签的<s:text.../>标签、表单标签形式
  2) 如果在 1 中找不到指定 key 对应的消息，则查找 struts.custom.i18n.resources 常量指定 baseName 的系列资源文件
  3) 如果经过上面步骤一直找不到该 key 对应的消息，直接输出个 key 的字符串值
- 如果<s:text.../>标签、表单标签没有使用<s:i18n.../>标签作为父标签
  1) 直接加载 struts.custom.i18n.resources 常量指定 baseName 的系列资源文件，如果找不到 key 对应的消息，则直接输出 key 属性的值

3.11. 使用 Struts2 的标签库

3.11.1. Struts2 标签库概述

1、与 Struts1 相比，Struts2 的标签库有一个巨大的改进之处：**Struts2 标签库的标签不依赖于任何表现层技术**，也就是说，Struts2 提供的大部分标签，可以在各种表现层技术中使用，包括最常用的 JSP 页面，也可在 Velocity 和 FreeMarker 等模板技术中使用

2、Struts2 不像 Struts1 那样，对整个标签库提供了严格的分类，Struts2 把所有标签都定义在一个 s 标签库里

3、虽然 Struts2 把所有的标签都定义在 URI 为"/struts-tags"的空间下，但依然可以对 Struts2 标签进行简单的分类，从最大范围来分，可以分为如下三类

- UI(User Interface，用户界面)标签：主要用于生成 HTML 元素的标签
  - 表单标签：主要用于生成 HTML 页面的 form 元素，以及普通表单元素的标签
  - 非标单标签：主要用于生成页面上的树、Tab 页等标签
- 非 UI 标签：主要用于数据访问、逻辑控制等的标签
  - 流程控制标签：主要包含用于实现分支、循环控制流程的标签
  - 数据访问标签：主要包含用于输出 ValueStack 中的值、完成国际化等功能的标签
- Ajax 标签：用于 Ajax(Asynchronous JavaScript And XML)支持的标签

3.11.2. 使用 Struts2 标签

1、标签库开发包括两个步骤：开发标签处理类和定义标签库定义文件，Struts2 框架已经完成了这两个步骤，即 Struts2 既提供了标签的处理类，也提供了 Struts2 的标签库定义文件

2、为了使 JSP 页面具有更好的兼容性，因此推荐定义 Struts2 标签库的 URI 时，使自定义的 Struts2 标签库 URI 与默认的 URI 相同

3、使用 Struts2 标签必须先导入标签库，在 JSP 页面中使用如下代码来导入 Struts2 标签库

```
<%@taglib prefix="s" uri="/struts-tags"%>
```

3.11.3. Struts2 的 OGNL 表达式语言

1、Struts2 利用内建的 OGNL(Object Graph Navigation Language)表达式语言支持，大大加强了 Struts2 的数据访问功能

2、OGNL 理解

- Struts2 应用中，视图页面可通过标签直接访问 Action 属性值
  - 实际上这只是一种假象
  - 类似于 Web 应用保持 application、session、request 和 page 四个范围的"银行"一样，Struts2 自行维护一个特定范围的"银行"，Action 将数据放入其中，而 JSP 页面可从其中取出数据，表面上似乎 JSP 可直接访问 Action 数据
- 当 Action 属性不是简单值(基本类型值或 String 类型值)时，而是某个对象，甚至是数组、集合等，就需要使用表达式语言来访问这些对象、数组、集合的内部数据
- Struts2 利用 OGNL 表达式语言来实现这个功能
- 事实上，OGNL 不是真正的编程语言，只是一种数据访问语言

3、Struts2 可以从对象中获取属性，Struts2 提供了一个特殊的 OGNL PropertyAccessor(属性访问器)，它可以自动搜寻 Stack Context 的所有实体(从上到下)，直到找到与求值表达式匹配的属性

4、Struts2 使用标准的 Context 来进行 OGNL 表达式语言求值，**OGNL 的顶级对象**

是 Stack Context(有时也称为 OGNL Context)，Stack Context 对象就是一个 Map 类型的实例，其根对象就是 ValueStack

- 除此之外，Struts2 还提供了一些命名对象，这些命名对象都不是 Stack Context 的"根"对象，它们只是存在于 Stack Context 中，访问这些对象需要使用#前缀来指明
- OGNL 的 Stack Context 是整个 OGNL 计算、求值的 Context，而 ValueStack 只是 StackContext 内的"根"对象而已
- OGNL 的 Stack Context 里除了包括 ValueStack 这个根之外，还包括 parameters、request、session、application、attr 等命名对象，但这些命名对象都不是根
- 根对象和普通命名对象的区别在于
  - 访问 Stack Context 里的命名对象需要在对象名之前加#前缀
  - 访问 OGNL 的 Stack Context 里的"根"对象时，可省略对象名
  - 访问非根对象属性，例如#session.msg 表达式
    - 由于 Struts2 中值栈被视为根对象，所以访问其他非根对象时，需要加#前缀
    - 实际上，#相当于 ActionContext. getContext()
    - 那访问值栈中的属性时相当于什么???
    - #session.msg 表达式相当于 ActionContext.getContext().getSession().getAttribute("msg")

5、OGNL 的基本概念

- OGNL 表达式的计算是围绕 OGNL 上下文进行的
  - OGNL 上下文实际上就是一个 Map 对象(暂时理解为[key：对象，value：属性组成的的 List]这样的映射关系)，由 ognl.OgnlContext 类表示。它里面可以存放很多个 JavaBean 对象。它有一个上下文根对象。
  - 上下文中的根对象可以直接使用名来访问或直接使用它的属性名访问它的属性值。否则要加前缀"#key"
- Struts2 的标签库都是使用 OGNL 表达式来访问 ActionContext 中的对象数据的

  <s:property value="xxx"/>
- Struts2 将 ActionContext 设置为 OGNL 上下文，并将值栈作为 OGNL 的根对象放置到 ActionContext 中
- 值栈(ValueStack)
  - 可以在值栈中放入、删除、查询对象。访问值栈中的对象不用"#"
  - Struts2 总是把当前 Action 实例放置在栈顶。所以在 OGNL 中引用 Action 中的属性也可以省略"#"
- 调用 ActionContext 的 put(key,value)放入的数据，需要使用#访问

3.11.4. OGNL 中的集合操作

1、使用 OGNL 表达式可以直接创建集合对象，语法如下

{e1,e2,e3...}

2、直接生成 Map 类型集合的语法如下

#{key1:value1,key2:value2,...}

3、对于集合，OGNL 提供两个运算符：in 和 not in，其中 in 判断某个元素是否在指定集合中，not in 用于判断某个元素是否不在指定集合中

4、OGNL 还允许通过某个规则取得集合的子集

- ?：取出所有符合选择逻辑的元素
- ^：取出符合选择逻辑的第一个元素
- $：取出符合选择逻辑的最后一个元素

person.relatives.{? #this.gender == 'male' }

- .{}运算符表明取出该集合的子集
- 在{}使用?表明取出所有符合选择逻辑的元素
- #this 代表集合里的元素

3.11.5. 访问静态成员

1、OGNL 表达式还提供了一种访问静态成员(包括调用静态方法、访问静态成员变量)的方式，但 Struts2 默认关闭了访问静态方法，只允许通过 OGNL 表达式访问静态 Field

- 为了让 OGNL 表达式可以访问静态方法，应该在 Struts2 应用中将 struts.ognl.allowStaticMethodAccess 设置为 true

<constant name="struts.ognl.allowStaticMethodAccess" value="true"/>

- 按上述设置好常量后，可以用如下语法访问静态成员
  - @className@staticField
  - @className@staticMethod(val...)

3.11.6. Lambda 表达式

1、<s:property value="#fib =:[#this==0? 0: #this==1? 1:
#fib(#this-2)+#fib(#this-1)],#fib(11)"/>

3.11.7. 控制标签

1、Struts2 的非 UI 标签包括控制标签和数据标签，主要用于完成流程控制，以及操作 Struts2 的 ValueStack

- 数据标签主要结合 OGNL 表达式进行数据访问
- 控制标签可以完成流程控制

2、控制标签有如下 9 个

- if：用于控制选择输出的标签
- elseIf/elseif：与 if 标签结合使用，用于控制选择输出的标签
- else：与 if 标签结合使用，用于控制选择输出的标签
- append：用于将多个集合拼接成一个新的集合
- generator：它是一个字符串解析器，用于将一个字符串解析成一个集合
- iterator：这是一个迭代器，用于将集合迭代输出
- merge：用于将多个集合拼接成一个新的集合，但与 append 的拼接方式有所不同
- sort：这个标签用于对集合进行排序
- subset：这个标签用于截取集合的部分元素，形成新的子集合

3、if/elseif/else 标签

- 只有<s:if.../>标签可以单独使用，<s:elseif.../>与<s:else.../>都不可单独使用
- <s:if.../>标签与<s:elseif.../> 标签可以接受 test 属性，该属性确定执行判断的 boolean 表达式

```
<s:if test="表达式">
    标签体
</s:if>
<s:elseif test="表达式">
    标签体
</s:elseif>
    ...可以多次出现 elseif
<s:else>
    标签体
</s:else>
```

4、iterator 标签

- iterator 标签主要用于对集合进行迭代，这里的集合包括 List、Set 和数组，也可对 Map 集合进行迭代输出
- <s:iterator.../>标签对集合进行迭代输出时，可以指定如下三个属性
  - value：这是一个可选的属性，value 属性用于指定被迭代的集合，***被迭代的集合通常都使用 OGNL 表达式指定，如果没有指定 value 属性，则使用 ValueStack 栈顶的集合***
  - id：这是一个可选的属性，该属性指定了集合里元素的 ID
    - 若迭代一个 List，那么指定 id="字符串"，然后用<s:property value="字符串" />即可访问每个元素
    - 若迭代一个 Map，那么无需指定 id，直接用<s:property value="key" />以及<s:property value="value" />来访问键和值即可
    - **是不是可以这样理解：在 iterator 标签体内，整个集合将位于 ValueStack，一旦标签体结束，该集合将被移出 ValueStack，那么定义 id 相当于给予当前迭代对象一个属性名**
  - status：这是一个可选的属性，该属性指定迭代时的 IteratorStatus 实例，通过该实例即可判断当前迭代元素的属性(**该 IteratorStatus 实例被临时放入了 Stack Context???**)
- 如果为<s:iterator.../>标签指定 status 属性，即每次迭代时都会有一个 IteratorStatus 实例，该实例包含了如下几个方法
  - int getCount()：返回当前迭代了几个元素
  - int getIndex()：返回当前迭代元素的索引
  - boolean isEven()：返回当前被迭代元素的索引是否为偶数
  - boolean isFirst()：返回当前被迭代元素是否第一个元素
  - boolean isLast()：返回当前被迭代元素是否是最后一个元素
  - boolean isOdd()：返回当前被迭代元素的索引是否是奇数

5、append 标签

- append 标签用于将多个集合对象拼接起来，组成一个新的集合，通过这种拼接方式，从而允许通过一个<s:iterator.../>标签就完成多个集合的迭代

- 使用<s:append.../>标签时需要指定一个 var 属性(也可以使用 id 属性，但是推荐 var)，该属性确定拼接生成的新集合的名字，**该新集合被放入 Stack Context 中**

6、generator 标签

- 使用 generator 标签可以将指定字符串按指定分隔符分隔成多个子串，临时生成多个子串可以使用 iterator 标签来迭代输出
- 可以这样理解，generator 将一个字符串转为一个 Iterator 集合，**在该标签的标签体内，整个临时生成的集合位于 ValueStack 顶端，一旦该标签结束，该集合将被移出 ValueStack**
- generator 标签的作用类似于 String 对象的 split()方法，但这个 generator 标签比 split()方法功能更强大
- 使用 generator 可以指定如下几个属性
  - count：该属性是一个可选的属性，该属性指定生成集合中的元素的总数
  - separator：这是一个必填的属性，该属性指定用于解析字符串的分隔符
  - val：这是一个必填的属性，该属性指定被解析的字符串
  - converter：这是一个可选属性，该属性指定一个转换器，该转换器负责将集合中每个字符串转换成对象，通过该转换器可以将一个字符串解析成对象集合，该属性值必须是一个 org.apache.Struts2.util.IteratorGenerator.Converter 对象
  - var：这是一个可选的属性，**如果指定了该属性，则将生成的 Iterator 对象放入 Stack Context 中**，该属性也可替换 id，但推荐使用 var 属性
- Struts2 的很多标签都与 generator 类似，都可以指定 var(以前是 id)属性
  - **一旦指定了 var 属性，则会将新生成、新设置的值放入 Stack Context 中(必须通过#name 形式访问)(相当于延长了生命周期，因为放入 ValueStack 中的值会在标签结束后弹出)**
  - **如果不指定 var 属性，则新生成、新设置的值不会放入 Stack Context 中，因此只能在该标签内部访问新生成、新设置的值---此时新生成、新设置的值位于 ValueStack 中，因此可以直接访问**

7、merge 标签

- merge 标签的用法看起来非常像 append 标签，也是用于将多个集合拼接成一个集合，但它采用的拼接方式与 append 的拼接方式有所区别
- 假设有三个集合，每个集合包含三个集合元素，分别使用 append 和 merge 方式进行拼接
- append 方式
  - 第 1 个集合中的第 1 个元素
  - 第 1 个集合中的第 2 个元素
  - 第 1 个集合中的第 3 个元素
  - 第 2 个集合中的第 1 个元素
  - 第 2 个集合中的第 2 个元素
  - 第 2 个集合中的第 3 个元素
  - 第 3 个集合中的第 1 个元素

- 第 3 个集合中的第 2 个元素
- 第 3 个集合中的第 3 个元素

➢ merge 方式

- 第 1 个集合中的第 1 个元素
- 第 2 个集合中的第 1 个元素
- 第 3 个集合中的第 1 个元素
- 第 1 个集合中的第 2 个元素
- 第 2 个集合中的第 2 个元素
- 第 3 个集合中的第 2 个元素
- 第 1 个集合中的第 3 个元素
- 第 2 个集合中的第 3 个元素
- 第 3 个集合中的第 3 个元素

8、subset 标签

➢ subset 标签用于取得集合的子集，该标签的底层通过 org.apache.struts2.util.SubsetIteratorFilter 类提供实现

➢ 使用 subset 标签还可以指定如下几个属性

- count：可选属性，该属性指定了子集中元素的个数，若不指定，则默认取得全部元素
- source：可选属性，该属性指定源集合，若不指定，则默认取得 ValueStack 栈顶的集合
- start：可选属性，该属性指定子集从源集合的第几个元素开始截取，默认从第一个元素(即 start 的默认值为 0)开始截取
- decider：这是一个可选属性，该属性指定由开发者自己决定是否选中该元素，该属性必须指定一个 org.apache.struts2.util.SubsetIteratorFilter.Decider 对象
- var：可选属性，如果指定了该属性，则将生成的 Iterator 对象设置成 page 范围的属性(放入 Stack Context???)，该属性也可替换成 id，但推荐 var

➢ **在 subset 标签内，subset 标签生成的子集和放在 ValueStack 的栈顶，所以可以在标签内直接迭代该标签生成的子集和，该标签结束后，该标签生成的子集合将被移除 ValueStack 栈**

➢ Struts2 还允许开发者决定截取标准，如果开发者需要实现自己的截取标准，则需要实现一个 Decider 类，Decider 类需要实现 SubsetIteratorFilter.Decider 接口，实现该类时，需要实现一个 boolean decide(Object element)方法，如果返回真，则表明该元素将被选入子集中

- <s:bean var="mydecider" name="org.crazyit.app.util.MyDecider"/>
- <s:subset source="{'1','2','...}" decider="#mydecider" var="newList"/>

➢ subset 标签和 generator 标签中的 var 属性作用并不相同，subset 的 var 属性是将心机和放入 pageScope 内，并不放入 Stack Context 中，而 generator 标签的 var 属性将新集合放入 Stack Context 以及 requestScope 内，但 Struts2 官方文档中 subset 标签和 generator 标签这两个标签中的 var 属性完全相同，这应该是 Struts2 或者文档的小 Bug，那么问题来了，到底一样不一样

9、sort 标签

- sort 标签用于对指定的集合元素进行排序，进行排序时，必须提供自己的排序规则，即实现自己的 Comparator，自己的 Comparator 需要实现 java.util.Comparator 接口
- sort 标签可以指定如下几个属性
  - comparator：这是一个必填的属性，该属性指定进行排序的 Comparator 实例
  - source：可选属性，该属性指定被排序的集合，如果不指定该属性，则对 ValueStack 栈顶的集合进行排序
  - var：可选属性，如果指定了该属性，将生成的 Iterator 对象设置成 page 范围的属性，不放入 Stack Context，**该属性的作用与 subset 标签中的 var 属性作用相同**
- 在 sort 标签内时，sort 标签生成的子集合放在 ValueStack 的栈顶，所以可以在标签内直接迭代该标签生成的子集合，该标签结束后，该标签生成的子集合将被移出 ValueStack 栈
- **Struts2 标签的 var 属性非常烦人，有的标签 var 属性会将新创建、新生成的值放入 Stack Context 中，有的不放入 Stack Context 中**

3. 11. 8. 数据标签

1、数据标签主要用于提供各种数据访问相关的功能，包含显示一个 Action 里的属性，以及生成国际化输出等功能

- action：该标签用于在 JSP 页面直接调用一个 Action，通过指定 executeResult 参数，还可将 Action 的处理结果包含在本页面中
- bean：该标签用于创建一个 JavaBean 实例，如果指定了 var 属性，可以将创建的 JavaBean 实例放入 Stack Context 中
- date：用于格式化输出的一个日期
- debug：用于在页面上生成一个调试链接，单击该链接时，可以看到当前 ValueStack 和 StackContext 中的内容
- i18n：用于指定国际化资源文件中的 baseName
- include：用于在 JSP 页面中包含其他的 JSP 或 Servlet 资源
- param：用于设置一个参数，通常是用作 bean 标签、url 标签的子标签
- push：用于将某个值放入 ValueStack 的栈顶
- set：用于设置一个新变量，并将新变量放入指定范围中
- text：用于输出国际化消息
- url：用于生成一个 URL 地址
- property：用于输出某个值，包括 ValueStack、Stack Context 和 Action Context 中的值

2、action 标签

- 使用 action 标签可以允许在 JSP 页面中直接调用 Action，因为需要调用 Action，所以可以指定需要被调用 Action 的 name 及 namespace，如果指定了 executeResult 参数的属性值为 true，该标签还会把 Action 的处理结果包含到本页面中来
- action 标签可以指定如下几个属性

- var：可选属性，一旦定义了该属性，Action 将被放入 Stack Context 中，该属性可以用 id 替代，推荐用 var
- name：必填属性，通过该属性指定该标签调用哪个 Action
- namespace：可选属性，指定该标签调用 Action 所在的 namespace
- executeResult：可选属性，该属性指定是否要将 Action 的处理结果页面包含到本页面
- ignoreContextParam：可选参数，指定该页面中的请求参数是否需要传入调用的 Action，默认 false，即将本页面的请求参数传入被调用的 Action

3、bean 标签

- bean 标签用于创建一个 JavaBean 实例，创建 JavaBean 实例时，可以在该标签体内使用<param.../>标签为该 JavaBean 实例传入属性，如果需要使用<param.../>标签为该 JavaBean 实例传入属性值，则应该为 JavaBean 类提供对应的 setter 方法，如果还希望访问该 JavaBean 的某个属性值，则应该为该属性提供对应的 getter 方法
- 使用 bean 标签时可以指定如下两个属性
  - name：必填属性，该属性指定要实例化 JavaBean 的实现类
  - var：可选属性，**如果指定了该属性，则该 JavaBean 实例会被放入 Stack Context 中**，并放入 requestScope 中，可用 id 替代，但推荐 var
- 在 bean 标签的标签体内时，bean 标签创建的 JavaBean 实例位于 ValueStack 的顶端，一旦该 bean 标签结束了，则 bean 标签创建的 JavaBean 实例被移出 ValueStack，将无法再次访问该 JavaBean 实例
- 关于 getter/setter 方法
  - 比方说有一个 name 属性，那么需要 getName 与 setName 方法
  - 置于在该 JavaBean 的实现类中，该 name 的字段名称是否一定要是 name，其实是无关紧要的
  - 引用 JavaBean 属性依据的是 getName 与 setName 方法中的属性名字，而非真实存储该属性的变量名字

4、date 标签

- date 标签用于格式化输出一个日期，除了可以直接格式化输出一个日期外，date 标签还可以计算指定日期和当前时刻之间的时间差
- 使用 date 标签时可以指定如下几个属性
  - format：可选属性，如果指定了该属性，将根据该属性指定的格式来格式化日期
  - nice：可选属性，该属性只能为 true 或 false，它用于指定是否输出指定日期和当前日期之间的时间差，该属性默认 false，即不输出时间差
  - name：必填属性，指定了要格式化的日期值
  - var：可选属性，**如果指定了该属性，格式化后的字符串将被放入 Stack Context 中，并放入 requestScope，但不会在页面上输出**
  - 如果既指定了 nice="true"，也指定了 format 属性，则会输出指定日期和当前时刻之间的时间差，即 format 属性失效
  - 如果既没有指定 format 属性，也没有指定 nice="true"，则系统会到国际化资源文件中寻找 key 为 struts.date.format 的消息，将该消息当成格

式化文本来格式化日期，如果无法找到 key 为 struts.date.format 的消息，则默认采用 DataFormat.MEDIUM 格式输出

5、debug 标签

- debug 主要用于辅助调试，它在页面上生成一个超级链接，通过该链接可以查看到 ValueStack 和 Stack Context 中所有的值信息
- 使用 debug 标签只有一个 id 属性，这个属性并没有太大的意义，仅仅是该元素的一个引用 id

6、include 标签

- include 标签用于将一个 JSP 页面，或者一个 Servlet 包含到本页面中
- 该标签有如下属性
  - value：必填属性，该属性指定需要被包含的 JSP 页面，或者 Servlet
- 除此之外，还可以为<s:include.../>标签指定多个<param.../>子标签，用于将多个参数值传入被包含的 JSP 页面或者 Servlet

7、param 标签

- param 标签主要用于为其他标签提供参数，例如为 include 标签和 bean 标签提供参数
- param 标签可以接受如下参数
  - name：可选属性，指定需要设置参数的参数名
  - value：可选属性，指定需要设置参数的参数值
- name 属性是可选的，如果提供了 name 属性，则要求 Component 提供该属性的 setter 方法，系统正是根据 setter 方法来传入参数的。如果不提供，则外层标签必须实现 UnnamedParametric 接口
- value 属性是可选的，因为有如下两种使用方法
  - <param name="color">blue</param>
  - <param name="color" value="blue"/> <==该参数的值是 blue 对象的值
  - <param name="color" value="'blue'"/> <==该参数的值是字符串

8、push 标签

- push **标签用于将某个值放到 ValueStack 的栈顶**，从而可以更简单地访问该值
- push 标签可以提供如下属性
  - value：必填属性，该属性指定需要放入 ValueStack 栈顶的值
- 只有在 push 标签内时，被 push 标签放入 ValueStack 中的对象才存在，一旦离开了 push 标签，则刚刚放入的对象将立即被移出 ValueStack

9、set 标签

- set 标签用于将某个值放入指定范围内，例如 application 范围、session 范围
- 当某个值所在对象图深度非常深时，例如有如下的值：person.worker.wife.parent.age，每次访问该值不仅性能低下，而且代码可读性也差
- 使用 set 标签可以理解为定义一个新变量，且将一个已有的值复制给新变量，并且可以将新变量放到指定的范围内
- set 标签有如下属性
  - scope：可选属性，指定新变量的范围，可以接受

application、session、request、page、action5 个值，默认 action

- value：可选属性，指定将赋给变量的值，如果没有指定该属性，则将 ValueStack 栈顶的值赋给新变量
- var：可选属性，如果指定了该属性，则会将该值放入 ValueStack 中

➢ 如果 scope 选择了 action 范围，**则该值将被放入 request 范围中**，并被放入 OGNL 的 Stack Context 中

10、url 标签

➢ url 标签用于生成一个 URL 地址，可以通过为 url 标签指定 param 子元素，从而向指定 URL 发送请求参数

➢ url 标签可以指定如下几个属性

- action：可选属性，指定生成 URL 的地址为哪个 Action，如果不提供，就使用 value 作为 URL 的地址值
- anchor：可选属性，指定 URL 的锚点
- encode：可选属性，指定是否需要对参数进行编码，默认 true
- escapeAmp：可选参数，指定是否需要对&符号进行编码，默认 true
- forceAddSchemeHostAndPort：可选参数，指定是否需要在 URL 对应的地址里强制添加 scheme、主机和端口
- includeContext：可选属性，指定是否要将当前上下文包含在 URL 地址中
- includeParams：可选属性，该属性指定是否包含请求参数，该属性的属性值只能为 none、get 或者 all，默认 get
- method：可选属性，该属性指定 Action 方法，当使用 Action 来生成 URL 时，如果指定了该属性，则 URL 将链接到指定 Action 的特定方法
- namespace：可选属性，该属性指定命名空间，当使用 Action 来生成 URL 时，如果能指定该属性，则 URL 将链接到此 namespace 指定的 Action 处
- portletMode：可选属性，指定结果页面的 portlet 模式
- scheme：可选属性，用于设置 scheme 属性
- value：可选属性，指定生成 URL 的地址值，如果 value 不提供就用 action 属性指定的 Action 作为 URL 地址
- var：可选属性，如果指定了该属性，将会把该链接值放入 Struts2 的 ValueStack 中，该属性可用 id 替代，推荐 var
- windowState：可选属性，指定结果页面的 portlet 窗口状态

11、property 标签

➢ property 标签的作用就是输出指定值，property 标签输出 value 属性指定的值，如果没有指定 value 属性，则默认输出 ValueStack 栈顶的值

➢ property 标签由如下几个属性

- default：可选属性，如果需要输出的属性值为 null，则显示 default 属性指定的值
- escape：可选属性，指定是否 escape HTML 代码，默认 true
- **value：可选属性，指定需要输出的属性值，若没有指定该属性，则默认输出 ValueStack 栈顶的值**

3.11.9. 主题和模板

1、Struts2 所有的 UI 标签都是基于主题和模板的，主题和模板是 Struts2 所有 UI 标签的核心

- 模板是一个 UI 标签的外在表示形式，例如当使用<s:select.../>标签时，Struts2 就会根据对应的 select 模板来生成一个由模板特色的下拉列表框
- 如果为所有的 UI 标签都提供了对应的模板，那么这个系列的模板就会形成一个主题

2、对于一个 JSP 页面里包含的 UI 标签而言，既可以直接设置该 UI 标签需要使用的模板，也可以设置该 UI 标签使用的主题

- 对于界面开发者而言，并不推荐直接设置模板属性，因为模板是以主题的形式组织在一起的，界面开发者应该选择特定主题，而不是强制使用特定模板来表现一个 UI 标签

3、主题是模板的组织形式，模板被包装在主题里面，对于开发者应该是透明的，当需要使用特定模板来表现某个 UI 标签时，应该让主题来负责模板的加载

4、设置主题的方法

- 通过设定 UI 标签上的 theme 属性来指定主题
- 通过设定 UI 标签外围的<s:form.../>标签的 theme 属性来指定主体
- 通过取得 page 会话范围内以 theme 为名称的属性来确定主题
- 通过取得 request 会话范围内的命名为 theme 的属性来确定主题
- 通过取得 session 会话范围内的命名为 theme 的属性来确定主题
- 通过取得 application 会话范围内的命名为 theme 的属性来确定的主题
- 通过设置名为 struts.ui.theme 的常量(默认值是 xhtml)来确定默认主题，该常量可以在 struts.properties 文件或者 struts.xml 文件中确定
- 以上的方式优先级从高到低排列，如果重复设置，以上面的为准

5、Struts2 完全允许在一个视图页面中使用几种不同的主题

- 如果需要改变整个表单，则可以直接设置该表单标签的 theme 属性
- 如果需要让某次用户会话使用特定的主题，则可以通过在 session 中设置一个 theme 的变量
- 如果想改变整个应用的主题，则应该通过修改 struts.ui.theme 常量值来实现

6、Struts2 的模板目录是通过 struts.ui.templateDir 常量来指定的，该常量的默认值是 template，意味着 Struts2 会从 Web 应用的 template 目录、CLASSPATH(包括 Web 应用的 WEB-INF/classes 路径和 WEB-INF/lib 路径)的 template 目录一次加载特定模板文件。加载模板文件的顺序如下

- 搜索 Web 应用里/template/xhtml/select.ftl
- 搜索 CLASSPATH 路径下的/template/xhtml/select.ftl

7、Struts2 默认的模板文件是*.ftl 文件，*.ftl 文件是 FreeMarker 模板文件，Struts2 使用 FreeMarker 技术来定义所有模板文件

8、Struts2 默认提供了三个主题：simple、xhtml 和 css_xhtml

- simple 主题是最简单的主题，它是最底层的基础，主要用于构建最基本的 HTML UI 组件
- xhtml 主题和 css_xhtml 主题都是对 simple 主题的包装和扩展
- xhtml 主题是 Struts2 默认的主题，它对 simple 主题进行扩展，在该主题

的基础上增加了如下特性

- 针对 HTML 标签(textfield 和 select 标签)使用标准的两列表格布局
- 每个 HTML 标签增加了配套的 label，label 既可以出现在 HTML 元素的左边，也可以出现在上边，这取决于 labelposition 属性的设置
- 自动输出校验错误提示
- 输出 JavaScript 的客户端校验

➢ css_xhtml 主题则对原有的 xhtml 主题进行了扩展，在 xhtml 主题基础上加入了 CSS 样式控制

3.11.10. 自定义主题

1、创建自定义主题有如下三种方式

➢ 开发者完全实现一个全新的主题
- 允许开发者选择自己的模板技术，例如使用 JSP 或者 Velocity
- 需要开发者为每个 UI 标签都提供自定义的模板文件，工作量非常大

➢ 包装一个现有的主题
- 包装就是在现有主题基础上，增加一些自定义代码部分，从而完成改写
- 使用纯粹包装方法来创建主题时，开发者必须为每个 UI 组件都提供自定义主题的模板文件，即使自定义主题里某个 UI 组件与原来主题里 UI 组件的行为完全一样

➢ 扩展一个现有的主题
- 开发者只需要提供自定义的模板文件

3.11.11. 表单标签

1、Struts 的表单标签，可分为两种：form 标签本身和单个表单元素的标签

➢ form 标签的行为不同于表单匀速标签

➢ Struts2 的表单元素标签都包含了非常多的属性，但有很多属性完全是通用的

2、所有表单标签处理类都继承了 UIBean 类，UIBean 包含了一些通用的属性，这些通用属性分成三种

➢ 模板相关属性

➢ JavaScript 相关属性

➢ 通用属性

3、与模板相关的通用属性，详见 P271-273

4、表单标签的 name 和 value 属性

➢ 对于表单标签而言，name 和 value 属性之间存在一个特殊关系，因为每个表单元素会被映射成 Action 属性，所以如果某个表单对应的 Action 已经被实例化(该表单被提交过)、且其属性有值时，则该 Action 对应表单里的表单元素会显示出该属性的值，这个值将作为 value 值

➢ name 属性设置表单元素的名字，表单元素的名字实际上封装着一个请求参数，而请求参数是被封装到 Action 属性的，因此可以将该 name 属性指定为你希望绑定值的表达式

➢ ???不懂

5、checkboxlist 标签

- checkboxlist 标签可以一次创建多个复选框，用于同时生成多个<input type="checkbox".../>的 HTML 标签，它根据 list 属性指定的集合来生成多个复选框
- checkboxlis 还有两个常用属性
  - listKey：该属性指定结合元素中的某个属性，例如集合集合元素为 Persion 实例，指定 Persion 实例的 name 属性作为复选框的 value，如果集合元素为 Map，使用 key 或 value 指定 Map 对象的 key 或 value 作为复选框的 value
  - listValue：该属性指定集合元素中的某个属性，同上

6、radio 标签

- radio 标签的用法与 checkboxlist 的用法几乎完全相同，一样可以指定 label、list、listKey 和 listValue 等属性
- 与 checkboxlist 唯一不同的是，checkboxlist 生成多个复选框，而 radio 生成多个单选钮

7、select 标签

- select 标签用于生成一个下拉列表框，使用该标签必须指定 list 属性，系统会使用 list 属性指定的集合来生成下拉列表框的选项
- 这个 list 属性指定的集合，既可以是普通的集合，也可以是 Map 对象，也可以是元素对象的集合
- select 有如下几个常用属性
  - listKey：该属性指定集合元素中的某个属性，例如集合集合元素为 Persion 实例，指定 Persion 实例的 name 属性作为复选框的 value，如果集合元素为 Map，使用 key 或 value 指定 Map 对象的 key 或 value 作为复选框的 valu
  - listValue：该属性指定集合元素中的某个属性，同上
  - multiple：设置该列表框是否允许多选

8、optgroup 标签

- optgroup 标签用于生成一个下拉列表框的选项组
- 该标签必须放在<s:select.../>标签中使用
- 一个下来列表框中可以包含多个选项组
- 与 select 标签类似，一样需要指定 list、listKey 和 listValue 等属性，且含义相同
- 另外，optgroup 标签也可以指定 label 属性，但这个 label 属性不是下拉列表框的 label，而是该选项组的组名

9、head 标签

- 该标签朱勇用于生成 HTML 页面的<head.../>部分，因为有些主题需要包含特定的 CSS 和 JavaScript 代码，而该标签则用于生成对这些 CSS 和 JavaScript 代码的引用
- 例如，如果需要在页面中使用 Ajax 组件，则使用一个带 theme="ajax"属性的 head 标签，就可以将标准 Ajax 的头信息包含在页面中
- 使用 Ajax 主题时，可以通过设置 head 标签的 debug 参数为 true，从而打开调试标志

- 一般使用 Struts2 的 UI 标签，JavaScript 客户端校验等需要 JavaScript 库和 CSS 支持功能时，都应该先使用 head 标签

10、updownselect 标签

- 类似 select 标签的用法，一样可以指定 list、listKey、listValue 等属性，这些属性的作用与使用 select 标签时指定的 list、listKey 和 listValue 等属性完全相同
- 区别于 select 标签，updownselect 标签生成的表框可以上下移动选项
- 该标签支持以下几个属性
  - allowMoveUp：是否显示"上移"按钮，默认 true
  - allowMoveDown：是否显示"下移"按钮，默认 true
  - allowSelectAll：是否显示"全选"按钮，默认 true
  - moveUpLabel：设置"上移"按钮上的文本，默认是^符号
  - moveDownLabel：设置"下移"按钮上的文本，默认是 v 符号
  - selectAllLabel：设置"全选"按钮上的文本，默认是*符号
  - **注意，上移下移指的是你选中的选项**

11、doubleselect 标签

- doubleselect 标签会生成一个级联列表框(会生成两个下拉列表框)，当选择第一个下拉列表框时，第二个下拉列表框的内容会随之改变
- 该标签有如下常用属性
  - list：指定用于输出第一个下拉列表框中选项的集合
  - listKey：该属性指定集合元素中的某个属性(get???/set???中的???)，例如对于 Person，有一个 name 属性，有 getName/setName 方法，例如 Map 有 key 属性和 value 属性，有 getKey 和 getValue 方法
    - 注意，这个属性并非是
  - listValue：该元素指定集合元素中的某个属性
  - doubleList：指定用于输出第二个下拉列表框中选项的集合
  - doubleListKey：该属性指定集合元素中的某个属性
  - doubleList：该属性指定集合元素中的某个属性
  - doubleName：指定第二个下拉列表框的 name 属性

12、optiontransferselect 标签

- optiontransferselect 会生成两个列表选择框，并生成系列的按钮用于控制各选项在两个下拉列表框之间移动、升降等
- 当提交该表单时，两个列表选择框对应的请求参数都会被提交
- 常用属性见 P281

13、token 标签

- 这是一个用于放置重复提交表单的标签，token 标签能阻止重复提交表单的问题(避免刷新页面导致的重复提交)
- 如果需要改标签起作用，则应该在 Struts2 的配置文件中启用 TokenInterceptor 拦截器或 TokenSessionStoreInterceptor 拦截器
- token 标签的实现原理是在表单中增加一个隐藏域，每次加载该页面时，该隐藏域的值都不相同，而 TokenInterceptor 拦截器则拦截所有用户请求，如果两次请求时该 token 对应隐藏域的值相同(前一次提交时 token 隐藏域的值保存在 session 里)，则组织表单提交

- 只需注意两个步骤
  - 页面中添加 toke 标签
    <s:token/>
  - 配置 Action 时启动 token 拦截器
    <interceptor-ref name="defaultStack"/>
    <interceptor-ref name="token"/>
    - 注意，这里需要显式使用 defaultStack 拦截器，之前都是默认使用，这里需要显式使用的原因是，显式使用了 token 拦截器
- 如果表单页没有<s:token/>标签，则千万不要使用 token 拦截器，否则它将导致无法提交表单

3.11.12. 非表单标签

1、非表单标签主要用于在页面中生成一些非表单的可视化元素，例如 Tab 页面、输出 HTML 页面的树形结构等

2、非表单标签主要有以下几个

- actionerror：如果 Action 实例的 getActionErrors()方法返回不为 null，则该标签负责输出该方法返回的系列错误
- actionmessage：如果 Action 实例的 getActionMessages()方法返回不为 null，则该标签负责输出该方法返回的系列消息
- component：使用此标签可以生成一个自定义组件
- fielderror：如果 Action 实例存在表单域的类型转换错误、校验错误、该标签负责输出这些错误提示

3、actioinerror 和 actionmessage 标签

- 这两个标签用法完全一样，作用也几乎完全一样，都是负责输出 Action 实例里封装的信息
- actionerror 标签负责输出 Action 实例的 getActionErrors()方法返回的值
- actionmessage 标签负责输出 Action 实例的 getActionMessages()方法的返回值

4、component 标签

- component 标签用于创建自定义视图组件，这是一个非常灵活的用法，如果开发者经常需要使用某个效果片段，就可以考虑将这个效果片段定义成一个视图组件，然后在页面中使用 component 标签来使用该自定义组件
- 由于使用自定义组件还是基于主题、模板管理的，因此在使用 component 标签时，可以指定如下三个属性
  - theme：自定义组件所使用的主题，如果不指定该属性，则默认使用 xhtml 主题
  - templateDir：指定自定义组件的主题目录，如果不能指定，则默认使用系统的主题目录，即 template 目录
  - template：指定自定义组件所使用的模板
- 除此之外，还可以在 component 标签内使用 param 子标签，子标签表示向该标签模板中传入额外的参数

Chapter 4. 深入使用 Struts2

1、本章将介绍 Struts2 的拦截器机制，拦截器是 Struts2 框架的灵魂，拦截器完成了 Struts2 框架的绝大部分功能

4.1. 详解 Struts2 的类型转换

1、所有的 MVC 框架都需要负责解析 HTTP 请求参数，并将请求参数传给控制器组件，此时问题来了

- HTTP 请求参数都是字符串类型，但 Java 是强类型语言
- **因此 MVC 框架必须将这些字符串参数转换成相应的数据类型---这个工作是所有 MVC 框架都应该提供的功能**

2、Struts2 提供了非常强大的类型转换机制

- Struts2 的类型转换可以基于 OGNL 表达式，只要把 HTTP 参数(表单元素和其他 GET/POST 的参数)命名为合法的 OGNL 表达式，就可以充分利用 Struts2 的类型转换机制
- 此外，Struts2 提供了很好的扩展性，开发者可以非常简单地开发出自己的类型转换器，完成字符串和自定义复杂类型之间的转换，如果类型转换中出现了未知异常，类型转换器开发者无需关心异常处理逻辑，Struts2 的 conversionError 拦截器会自动处理该异常，并在页面上生成提示信息
- Struts2 的类型转换器提供了非常强大的表现层数据处理机制，开发者可以利用 Struts2 的类型转换机制来完成任意的类型转换

3、表现层另一个数据处理是数据校验，数据校验可分为客户端校验和服务器校验两种，两者都是比不可少的，它们完成不同的任务

- 客户端校验：
  - 客户端校验进行基本校验，如检验非空字段是否为空，数字格式是否正确等，客户端校验主要用来过滤用户的错误操作
  - 客户端校验的作用：拒绝误操作输入提交到服务器处理，降低服务器端负担
- 服务器端校验：
  - 服务器端校验防止非法数据进入程序，导致程序异常、底层数据库异常
  - 服务器端校验是保证程序有效运行及数据完整的手段

4.1.1. Struts2 内建的类型转换器

1、对于大部分的常用类型，开发者无需理会类型转换，Struts2 可以完成大多数常用的类型转换，这些常用的类型转换是通过 Struts2 内建的类型转换器完成的，Struts2 已经内建了字符串类型和如下类型之间相互转换的转换器

- boolean 和 Boolean：完成字符串和布尔值之间的转换
- char 和 Character：完成字符串和字符之间的转换
- int 和 Integer：完成字符串和整型值之间的转换
- long 和 Long：完成字符串和长整型值之间的转换
- float 和 Float：完成字符串和单精度浮点值之间的转换
- double 和 Double：完成字符串和双精度浮点值之间的转换

- Date：完成字符串和日期类型之间的转换
- 数组：默认情况下，数组元素是字符串，如果用户提供了自定义类型转换器，也可以是其他符合类型的数组
- 集合：在默认情况下，假定集合元素类型是 String，创建一个新的 ArrayList 封装所有的字符串

2、由于 Struts2 提供了上述类型转换器，如果需要把 HTTP 请求参数转换成上面的这些类型，则无需开发者进行任何特殊处理，因此大部分实际开发中，开发人员无须自己进行类型转换

4.1.2. 基于 OGNL 的类型转换

1、借助于内置的类型转换器，Struts2 可以完成字符串和基本类型之间的类型转换。除此之外，借助于 OGNL 表达式的支持，Struts2 允许以另一种简单方式将请求参数转换成符合类型

2、几个例子

```
<s:textfield name="user.name" label="用户名" />
```

- Struts2 会把 user.name 参数的值赋给 Action 实例的 user 属性的 name 属性
- 通过这种方式 Struts2 可以将普通请求参数转换成复合类型对象(并不是将 String 解析为一个对象，而是通过这种表达式来转换)，但在使用这种方式时有如下几点需要注意
  - Struts2 将通过反射来创建一个符合类(User 类)的实例，因此系统必须为该符合类提供无参构造器
  - 如果希望使用 user.name 请求参数的形式为 Action 实例的 user 属性的 name 属性赋值，则必须为 user 属性对应的复合类(User 类)提供 setName 方法，因为 Struts2 是通过调用该方法来为该属性赋值的

```
<s:textfield name="users['one'].name" label="第 one 个用户名"/>
```

- 将表单域的 name 属性设置为"Action 属性名['key 值'].属性名"的形式，其中"Action 属性名是 Action 类里包含的 Map 类型属性，后一个属性名则是 Map 对象里复合类型对象(Map 的 Value)的属性名

4.1.3. 指定集合元素的类型

1、泛型可以让 Struts2 了解集合元素的类型，Struts2 就可以通过反射来创建对应类的对象，并将这些对象添加到 List 中，**如果使用泛型，Struts2 当然不知道如何处理 users 属性**

2、Struts2 允许开发者通过局部类型转换文件来指定集合元素的类型，类型转换文件就是一个普通的 Properties(*.properties)文件，类型转换文件里提供了类型转换的相关配置信息

- 局部类型转换文件的文件名应该是 ActionName-conversion.properties 形式，其中 ActionName 是需要 Action 的类名，后面的 conversion.properties 字符串则是固定部分
- 类型转换文件应该放在和 Action 类文件相同的位置
- 为了指定 List 集合里元素的数据类型，需要指定两个部分
  - List 集合**属性的名称(仍然指的是 setter/getter 方法中的那个名字而不是实际字段的变量名称)**

- List 集合里元素的类型
- 通过在局部类型转换文件中指定如下 key-value 对即可

Element_<ListPropName>=<ElementType>

Element_users=org.crazyit.app.domain.User

➢ 为了指定 Map 集合里元素的数据类型，则需要指定两个项，每项两个部分

- Map 的 key 类型

Key_<MapPropName>=<KeyType>

  - Key 是固定的
  - <MapPropName>是 Map 类型属性的**属性名(仍然指的是 setter/getter 方法中的那个名字而不是实际字段的变量名称)**，复合类型指定的是 Map 的 key 值的全限定类名

- Map 的 value 类型

Element_<MapPropName>=<ValueType>

  - Element 是固定的
  - <MapPropName>是 Map 类型属性的**属性名(仍然指的是 setter/getter 方法中的那个名字而不是实际字段的变量名称)**，复合类型指定的是 Map 属性的 value 类型的全限定名

3、总结，为了让 Struts2 能了解集合属性中元素的类型，可以使用如下方式

➢ 通过为集合属性指定泛型

➢ 通过在 Action 的局部类型转换文件中指定集合元素类型

4.1.4. 自定义类型转换器

1、**大部分时候，使用 Struts2 提供的类型转换器，以及基于 OGNL 的类型转换机制，就能满足大部分类型转换需求，但有时候如果需要把一个字符串转换成一个复合对象(例如 User 对象)，这就需要使用自定义类型转换器**

➢ **可以利用<s:textfield name="user.name" label="..."/>的方式来利用 OGNL 表达式进行类型转换，但是 name 必须是基本类型或者 String**

➢ **因此不能用<s:textfield name="use" label="..."/>这样的方式将输入的字符串直接转为 User 对象，必须自定义类型转换器，若不想定义就将该复合类型分解为基本类型的 OGNL 表达式**

2、Struts2 的类型转换器实际上依然是基于 OGNL 框架，在 OGNL 项目中有一个 TypeConverter 接口，这个接口就是自定义类型转换器必须实现的接口

```
public interface TypeConverter{
    public Object convertValue(Map context, Object target, Member member,
                String propertyName, Object value, Class toType);
}
```

➢ 实现类型转换器必须实现上面的 TypeConverter，不过该接口方法太过复杂，所以 OGNL 项目还为该接口提供了一个实现类：DefaultTypeConverter，通常都采用扩展该类来实现自定义类型转换器

3、convertValue 方法的作用

➢ 该方法负责完成类型的转换，不过这种转换是双向的

- 当需要把字符串转成 User 实例时，是通过该方法实现的
- 当需要把 User 实例转换成字符串时，也是通过该方法实现的

- 为了让该方法实现双向转换，程序通过判断 toType 的类型即可判断转换的方向
  - 当 toType 类型是 User 类型时，表明需要将字符串转换成 User 实例
  - 当 toType 类型是 String 类型时，表明需要把 User 实例转换成字符串类型
- 一旦通过 toType 类型判断了类型转换的方向后，接下来即可分别实现两个方向的转换逻辑了

4、convertValue 方法参数和返回值的意义
- 第一个参数：context 是类型转换的上下文
- 第二个参数：value 是需要转换的参数
  - 当把字符串类型向 User 类型转换时，value 是原始字符串数组
  - 当需要把 User 类型向字符串类型转换时，value 是 User 实例
- 第三个参数：toType 是转换后的目标类型

5、当把字符串向 User 类型转换时，为什么 value 是一个字符串数组，而不是一个字符串
- 对于有些 UI 控件传入的确实是字符串
- 但是有些控件传入的是字符串数组
- 为了统一，都采用字符串数组，无非是字符串的时候转为长度为 1 的字符串数组

4.1.5. 注册类型转换器

1、仅仅为该应用提供类型转换器还不够，因为 Struts2 依然不知道何时使用这些类型转换器，所以还必须将类型转换器注册在 Web 应用中，Struts2 框架才可以正常使用该类型转换器

2、Struts2 支持如下三种注册类型转换器的方式
- 注册局部类型转换器：局部类型转换器仅仅对某个 Action 属性起作用
- 注册全局类型转换器：全局类型转换器对所有 Action 的特定类型的属性都会生效
- 使用 JDK 1.5 的注解来注册类型转换器：通过注解方式来注册类型转换器

3、局部类型转换器
- 注册局部类型转换器使用局部类型转换文件制定，只要在**局部类型转换文件(就是上面为 List 以及 Map 指定泛型参数的文件)**中增加如下一行即可

  <propName>=<ConverterClass>
  - <propName>替换成需要进行类型转换的属性
  - <ConverterClass>替换成类型转换器的实现类即可
  - 例子：user=org.crazyit.app.converter.UserConverter
- 局部类型转换器只对指定 Action 的特定属性起作用，这具有很大的局限性，即花费了大量的时间完成了一个类型转换器，却只能用一次
- 通常会将类型转换器注册成全局类型转换器，让该类型转换器对该类型的所有属性都起作用

4、全局类型转换器
- 局部类型转换器的局限：它只能对指定 Action 的指定属性起作用，但如

果应用中有多个 Action 都包含了 User 类型的属性，或者一个 Action 中包含了多个 User 类型的属性，使用全局类型转换器将更合适

- 全局类型转换器不是对指定的 Action 的指定属性起作用，而是对指定类型起作用
- 注册全局类型转换器应该提供一个 xwork-conversion.properties 文件，该文件也是 Properties 文件，该文件就是全局类型转换文件，该文件直接放在 WEB-INF/classes 路径下即可
- 全局类型转换文件内容由多项"<propType>=<ConvertClass>"项组成，将<propType>替换成需要进行类型转换的类型，将<ConvertClass>替换成类型转换器的实现类即可

org.crazyit.app.domain.User=org.crazyit.app.converter.UserConverter

5、局部类型转换器对指定 Action 的指定属性起作用，一个属性只调用 convertValue()方法一次，全局类型转换器对所有 Action 的特定类型起作用，因此可能对一个属性多次调用 convertValue()方法进行转换

4.1.6. 基于 Struts2 的自定义类型转换器

1、为了简化类型转换器的实现，Struts2 提供了一个 StrutsTypeConvert 抽象类，这个抽象类是 DefaultTypeConverter 类的子类

- StrutsTypeConvert 类简化了类型转换器的实现，该类已经实现了 DefaultTypeConverter 的 convertValue()方法
- 实现该方法时，它将两个不同转换方向替换成不同方法
  - 当需要把字符串转换成复合类型时，调用 convertFromString()抽象方法
  - 当需要把复合类型转换成字符串时，调用 convertToString()抽象方法
- 通过继承 StrutsTypeConverter 类来实现类型转换器，分别实现 convertFromString()和 convertToString()，这两个方法分别代表不同的转换逻辑---程序逻辑更加清晰
- 注册方法完全一致，不再赘述

4.1.7. 处理 Set 集合

1、通常不建议在 Action 中使用 Set 集合属性

- 因为 Set 集合里元素处于无序状态，所以 Struts2 不能准确地将请求参数转换成 Set 集合的元素
- 不仅如此，由于 Set 集合里元素的无序性，所以 Struts2 也不能准确读取 Set 集合里的元素
- 除非 Set 集合里的元素有一个标识属性，这个标识属性可以唯一地标识集合元素，这样 Struts2 就可以根据该标识属性来存取集合元素了

2、为了让 Struts2 准确地存取 Set 集合元素，还必须让 Struts2 明白 Set 集合元素的标识属性，指定 Struts2 根据该标识属性来存取 Set 集合元素

3、Struts2 允许通过局部类型转换文件类指定 Set 集合元素的标识属性，在局部类型转换文件中增加如下一行指定 Set 集合元素的标识属性

KeyProperty_<SetPropName>=<keyPropName>

- <SetPropName>替换成集合属性名
- <keyPropName>替换成集合元素的标识属性即可

- 例如：keyProperty_users=name

4、取用的例子

- <s:property value="users('crazyit.org').name"/>
- 问题如下
- 首先 users 会读取值栈中的 users 属性，这是一个 Set<User>
- 那么('crazyti.org')这部分如何转换呢?上面提到了为了让 Set 可以存取元素，提供了一个标识符，是否会通过调用 User.equals 方法来判断是否存取
- 取到了 User 对象后，那么.name 就相当于调用了 User.getName()方法

4.1.8. 类型转换中的错误处理

1、表现层数据是由用户输入的，用户输入则是非常复杂的，正常用户的偶然错误，还有 Cracker(破坏者)的恶意输入，都可能导致系统出现非正常情况

2、表现层数据涉及的两个处理：数据校验和类型转换是紧密相连的，只有当输入数据是有效数据时，系统才可以进行有效的类型转换

3、Struts2 提供了一个名为 conversionError 的拦截器，这个拦截器被注册在默认的拦截器栈中

4、为了让 Struts2 框架处理类型转换的错误，以及使用后面的数据校验机制，系统的 Action 类都应该通过继承 ActionSupport 类来实现，ActionSupport 类为完成类型转换错误处理，数据校验实现了许多基础工作

- ActionSupport 负责收集类型转换错误、输入校验错误，并将它们封装成 FieldError 对象，添加到 ActionContext 中
- 在页面中使用**<s:fielderror/>**标签即可输出类型转换错误信息

5、对于中文环境而言，用户通常希望看到中文的提示消息，因此应该改变默认的提示信息，只需在应用的国际化资源文件中增加如下代码，即可改变默认的类型转换错误的提示信息

xwork.default.invalid.fieldvalue={0}字段类型转换失败

- 由于包含西欧字符，因此必须使用 native2ascii 命令来处理该文件
- 某些时候，可能还需要对特定字段指定特别的提示信息，此时可以通过 Action 的局部资源文件来实现

invalid.fieldvalue.<propName>=<tipMsg>

6、处理集合属性的转换错误

- 如果 Action 里包含一个集合属性，只要 Struts2 能检测到集合里元素的类型(可以通过局部类型转换文件指定，也可以通过泛型方式指定)
- 例如 Action 中的 users 是一个 List 集合，此处由两种方式为 users 传入请求参数
  - 只传入一个 users 请求参数，该请求参数的值是字符串数组的形式
  - 分别传入多个 users[0]、users[1]...形式的请求参数，这种形式将会充分利用 OGNL 表达式类型转换机制

4.2. 使用 Struts2 的输入校验

1、输入校验也是所有 Web 应用必须处理的问题，应用通过输入页面收集的数据是非常复杂的，不仅会包含正常用户的误输入，还可能包含恶意用户的恶意输入，一个健壮的应用系统必须将这些非法输入阻止在应用之外，防止这些非

法输入进入系统，这样才可以保证系统不受影响
2、通常做法是遇到异常输入时应用程序直接返回，提示浏览者必须重新输入，也就是将那些异常输入过滤掉，对异常输入的过滤，就是输入校验，也称为数据校验
3、输入校验分为客户端校验和服务器校验

- 客户端校验主要是过滤正常用户的误操作，主要通过 JavaScript 代码完成
- 服务器端校验是整个应用阻止非法数据的最后防线，主要通过在应用中编程实现

4.2.1. 编写校验规则文件

1、Struts2 提供了基于验证框架的输入校验，在这种校验方式下，所有的输入校验只需要编写简单的配置文件，Struts2 的验证框架将会负责进行服务器校验和客户端校验
2、校验文件的根元素<validators.../>元素，该元素可以包含多个<field.../>或<validator.../>元素，它们都用于配置校验规则

- <field-validator.../>是字段校验器的配置风格
- <validator.../>是费字段校验器的配置风格

3、**Struts2 的校验文件规则与 Struts1 的校验文件设计方式不同，Struts2 中每个 Action 都有一个校验文件，因此该文件名应该遵守如下规则**

<Action 名字>-validation.xml

- 该文件应该被保存在与 Action class 文件相同的路径下
- 与类型转换失败相似的是，当输入校验失败后，Struts2 也是自动返回名为"input"的 Result，因此需要在 struts.xml 配置文件中配置名为"input"的 Result

4.2.2. 国际化提示信息

1、上一小节的数据校验中，所有的提示信息都是通过硬编码的方式卸载配置文件中，这种方式不利于国际化
2、<field-validator.../>元素包含了一个必填的<message.../>子元素，这个子元素中的内容就是校验失败后的提示信息，为了国际化该提示消息，为 message 元素指定 key 属性，该 key 指定的是国际化提示信息对应的 key

4.2.3. 使用客户端校验

1、Struts2 应用中使用客户端校验非常简单，只需要改变如下两个地方

- 将输入页面的表单元素改为使用 Struts2 标签来生成表单
- 为该<s:form.../>元素增加 validate="true"属性

2、客户端校验是基于 JavaScript 完成的，因为 JavaScript 脚本本身的限制，有些服务器端校验不能转换成客户端校验，也就是说并不是所有服务器端校验都可以转成客户端校验
3、客户端校验仅支持以下几种校验器

- required validator：必填校验器
- requiredstring validator：必填字符串校验器
- stringlength validator：字符串长度校验器

- regex validator：表达式校验器
- email validator：邮件校验器
- url validator：网址校验器
- int validator：整数校验器
- double validator：双精度校验器

4、客户端校验器 4 个值得注意的地方

- Struts2 的<s:form.../>元素有一个 theme 属性，不要将该属性指定为 simple
- 浏览者不能直接访问启用客户端校验的表单页，这样会引发异常，可以把启用客户端校验的表单页放到 WEB-INF 路径下，让浏览者访问所有资源之前都经过核心 Filter
- 如果客户端校验希望输出国际化提示消息，那就需要使用全局国际化资源文件，不能使用 Action 范围的国际化资源文件
- 启用客户端校验的表单页面的 action 和 namespace 要分开写，例如 <s:form action="regist" namespace="/lee"> 而不能写成 <s:form action="lee/regist>

4.2.4. 字段校验器配置风格

1、Struts2 提供了两种方式来配置校验规则：字段校验风格和非字段校验风格，这两种风格没有本质的不同，只是组织校验规则的方式不同。一种是字段优先，称为字段校验器风格；另一种是校验器优先，称为非字段校验器风格

2、<validators.../>是校验规则文件的根元素，该根元素下可以出现两个元素：<field.../>元素和<validator.../>元素，出现第一种元素时就是字段优先，就是字段校验器配置风格，出现第二种元素时，就是校验器优先，就是非字段校验器配置风格

```
<field name="被校验的字段">
  <field validator type="校验器名">
    <!--为不同校验器指定数量不等的校验参数-->
    <param name="参数名">参数值</param>
    ...
    <!--校验失败后的提示信息，其中 key 指定国际化信息的 key-->
    <message key="I18Nkey">校验失败后的提示信息</message>
  </field-validator>
  <!--如果该字段需要满足多个规则，可以设置多个校验器-->
  ...
</field>
```

4.2.5. 非字段校验器配置风格

1、对于非字段校验器配置风格，这是一种以校验器优先的配置方式，在这种配置方式下，校验规则文件的根元素下包含了多个<validator.../>元素，每个<validator.../>元素定义了一个校验规则

```
<validator type="校验器名">
  <param name="fieldName">需要被校验的字段</param>
  <!--此处需要 Wie 不同校验器指定数量不等的校验参数-->
```

```
        <param name="参数名">参数值</param>
        ...
        <!--校验失败后的提示信息，其中 key 指定国际化信息的 key-->
        <message key="I18Nkey">校验失败后的提示信息</message>
    </validator>
```

4.2.6. 短路校验器

1、校验规则文件的<validator.../>元素和<field-validator.../>元素可以指定一个可选的 short-circuit 属性，这个属性指定该校验器是否是短路校验器，该属性默认是 false，即默认是非短路校验器

2、对于同一个字段内的多个校验器，如果一个短路校验器校验失败，其他校验器都根本不会继续校验，这就是短路的意思

3、通常，在一个<field.../>元素内定义字段校验器比使用一个带有 fieldName 参数的<validator.../>元素好得多，而且 XML 代码本身也清晰很多，因此通常推荐使用字段校验器风格

4.2.7. 校验文件的搜索规则

1、Struts2 的一个 Action 中可能包含了多个处理逻辑

- 当一个 Action 类中包含多个类似于 execute()的方法时，每个方法都是一个处理逻辑
- 不同的处理逻辑可能需要不同的校验规则，Struts2 允许为不同控制逻辑指定不同校验规则的支持

2、为了能精确控制每个校验逻辑，Struts2 允许通过位校验规则文件名增加 Action 别名来指定具体需要校验的处理逻辑

<ActionClassName>-<ActionAliasName>-validation.xml

3、假设系统有两个 Action：BaseAction 和 RegistAction，其中 RegistAction 继承 BaseAction，那么 loginPro 的 Action 的校验规则搜索如下

- BaseAction-validation.xml
- BaseAction-loginPro-validation.xml
- RegistAction-validation.xml
- RegistAction-loginPro-validation.xml
- 这种搜索与其他搜索不同的是，即使找到第一个校验规则，系统还会继续搜索，即系统总是按固定顺序搜索，因此
- 如果对于 regist 的 Action 而言，那么不包含 BaseAction-loginPro-validation.xml 与 RegistAction-loginPro-validation.xml

4、**Struts2 搜索规则文件是从上而下的，实际用的校验规则是所有校验规则的总和，如果两个校验规则中指定的校验规则冲突，则后面的起作用**

4.2.8. 校验顺序和短路

1、校验器增加了短路的特性后，校验器执行的顺序就变得非常重要了，因为前面执行的校验器可能阻止后面校验器的执行

- 校验器的执行顺序有如下原则
  - 所有非字段风格的校验器优先于字段风格的校验器
  - 所有非字段风格的校验器中，排在前面的先执行

- 所有字段风格的校验器中，排在前面的先执行

➢ 校验器短路的原则如下

- 所有非字段校验器是最优先执行，如果某个非字段校验器校验失败了则该字段上所有字段校验器都不会获得校验的机会
- 非字段校验器的校验失败，不会阻止其他非字段校验的执行
- 如果一个字段校验器失败后，则该字段下且排在该校验失败的校验器之后的其他字段校验器不会获得校验机会
- 字段校验器永远不会阻止非字段校验器的执行

2、如果应用中所需要的校验规则非常复杂，用户可以有两个选择：开发自己的校验器，或者重写 Action 的 validate 方法

4.2.9. 内建校验器

1、Struts2 提供了大量的内建校验器，这些内建的校验器可以满足大部分应用的校验需求，开发者只需要使用这些校验器即可。如果应用有一个特别复杂的校验需求，而且该校验有很好的复用性，开发者可以开发自己的校验器

2、校验器注册

➢ 通过一个<validator.../>元素即可注册一个校验器

➢ 每个<validator.../>元素的 name 属性指定该校验器的名字，class 属性指定该校验器的实现类

3、如果开发者开发了一个自己的校验器，则可以通过添加一个 validators.xml 文件(该文件应该放在 WEB-INF/classes 路径下)来注册校验器，validators.xml 文件的内容也是由多个<validator.../>元素组成的，每个<validator.../>元素注册一个校验器

4、注意：**如果 Struts2 系统在 WEB-INF/classes 路径下找到了一个 validators.xml 文件，则不会再加载系统默认的 default.xml 文件，因此，如果开发者提供了自己的校验器注册文件(validators.xml)，一定要把 default.xml 文件里面的全部内容复制到 validators.xml 文件中**

4.2.9.1. 必填校验器

1、名字：required

2、功能：该校验器要求指定的字段必须有值(非空)

3、可接受参数

➢ fieldName：该参数指定校验的 Action 属性名，如果采用字段校验器风格，则无需指定该参数

4、非字段校验器配置风格

```
<validators>
    <validator type="required">
        <param name="fieldName">username</param>
        <message>username must not be null</message>
    </validator>
    ...
<validators>
```

5、字段校验器配置风格

```
<validators>
```

```
        <field name="username">
            <field-validator type="required">
                <message>username must not be null</message>
            </field-validator>
            ...
        </field>
        ...
    <validators>
```

4.2.9.2. 必填字符串校验器

1、名字：requiredstring
2、功能：该校验器要求字段值必须非空且长度大于 0，即该字符串不能是""
3、可接受参数

- fieldName：该参数指定校验的 Action 属性名，如果采用字段校验器风格，则无需指定该参数
- trim：是否在校验前截断被校验属性值前后的空白，该属性可选，默认 true

4、非字段校验器配置风格

```
    <validators>
        <validator type="requiredstring">
            <param name="fieldName">username</param>
            <param name="trim">true</param>
            <message>username must not be null</message>
        </validator>
        ...
    <validators>
```

5、字段校验器配置风格

```
    <validators>
        <field name="username">
            <field-validator type="requiredstring">
                <param name="trim">true</param>
                <message>username must not be null</message>
            </field-validator>
            ...
        </field>
        ...
    <validators>
```

4.2.9.3. 整数校验器

1、名字：int、long、short
2、功能：该校验器要求字段的整数值必须在指定范围内
3、可接受参数

- fieldName：该参数指定校验的 Action 属性名，如果采用字段校验器风格，则无需指定该参数
- min：指定该属性的最小值，该参数可选，如果没有指定，则不检查最小值

- max：指定该属性的最大值，该参数可选，如果没有指定，则不检查最大值

4、非字段校验器配置风格

```
<validators>
  <validator type="int">
    <param name="fieldName">age</param>
    <param name="min">20</param>
    <param name="max">50</param>
    <message>Age needs to be between ${min} and ${max}</message>
  </validator>
  ...
<validators>
```

5、字段校验器配置风格

```
<validators>
  <field name="age">
    <field-validator type="int">
      <param name="min">20</param>
      <param name="max">50</param>
      <message> Age needs to be between ${min} and ${max}</message>
    </field-validator>
    ...
  </field>
  ...
<validators>
```

4. 2. 9. 4. 日期校验器

1、名字：date

2、功能：该校验器要求字段的日期值必须在指定范围内，该校验器可以接受如下参数

3、可接受参数

- fieldName：该参数指定校验的 Action 属性名，如果采用字段校验器风格，则无需指定该参数
- min：指定该属性的最小值，该参数可选，如果没有指定，则不检查最小值
- max：指定该属性的最大值，该参数可选，如果没有指定，则不检查最大值

4、非字段校验器配置风格

```
<validators>
  <validator type="date">
    <param name="fieldName">birth</param>
    <param name="min">1990-01-01</param>
    <param name="max">2010-01-01</param>
    <message>Birthday must be between ${min} and ${max}</message>
  </validator>
  ...
<validators>
```

5、字段校验器配置风格

```
<validators>
    <field name="birth">
        <field-validator type="date">
            <param name="min">1990-01-01</param>
            <param name="max">2010-01-01</param>
            <message> Birthday must be between ${min} and ${max}</message>
        </field-validator>
        ...
    </field>
    ...
<validators>
```

4.2.9.5. 表达式校验器

1、名字：expression

2、功能：**这是一个非字段校验器，不可在字段校验器的配置风格中使用**，该表达式要求 OGNL 表达式返回 true，当返回 true，该校验通过，否则不通过

3、可接受参数

- express：该参数指定一个逻辑表达式，该逻辑表达式基于 ValueStack 进行求值，最后返回一个 Boolean 值；当返回 true 时，校验通过，否则校验失败

4、非字段校验器配置风格，<未完成>：不懂

```
<validators>
    <validator type="expression">
        <param name="expression">.....</param>
        <message>Failed to meet Ognl Expression .....</message>
    </validator>
    ...
<validators>
```

4.2.9.6. 字段表达式校验器

1、名字：fieldexpression

2、功能：他要求指定字段满足一个逻辑表达式

3、可接受参数

- fieldName：该参数指定校验的 Action 属性名，如果采用字段校验器风格，则无需指定该参数
- expression：该参数指定一个逻辑表达式，该逻辑表达式基于 ValueStack 进行求值，最后返回一个 Boolean 值；当返回 true 时，校验通过，否则失败

4、非字段校验器配置风格

```
<validators>
    <validator type="fieldexpression">
        <param name="fieldName">pass</param>
        <param name="expression"><![CDATA[(pass==rpass)]]</param>
        <message>密码必须和确认密码相等</message>
```

```
        </validator>
        ...
    <validators>
```

5、字段校验器配置风格

```
    <validators>
        <field name="pass">
            <field-validator type="fieldexpression">
                <param name="expression"><![CDATA[(pass==rpass)]]</param>
                <message>密码必须和确认密码相等</message>
            </field-validator>
            ...
        </field>
        ...
    <validators>
```

4.2.9.7. 邮件地址校验器

1、名字：email

2、功能：要求被检查字段的字符如果是非空，则必须是合法的邮件地址，随着技术的发展，这个校验器可能不能完全覆盖实际的电子邮件地址，此时，建议开发者使用正则表达式校验器来完成邮件校验

3、可接受参数

- fieldName：该参数指定校验的 Action 属性名，如果采用字段校验器风格，则无需指定该参数

4、非字段校验器配置风格

```
    <validators>
        <validator type="email">
            <param name="fieldName">email</param>
            <message>你的电子邮件地址必须是一个有效的电邮地址</message>
        </validator>
        ...
    <validators>
```

5、字段校验器配置风格

```
    <validators>
        <field name="email">
            <field-validator type="email">
                <message>你的电子邮件地址必须是一个有效的电邮地址</message>
            </field-validator>
            ...
        </field>
        ...
    <validators>
```

4.2.9.8. 网址校验器

1、名字：url

2、功能：它要求被检查字段的字符如果非空，则必须是合法的 URL 地址，随着

技术的发展，这个校验器不能完全覆盖所有网址，此时建议开发者使用正则表达式校验器进行网址校验

3、可接受参数

- fieldName：该参数指定校验的 Action 属性名，如果采用字段校验器风格，则无需指定该参数

4、非字段校验器配置风格

```
<validators>
  <validator type="required">
    <param name="fieldName">username</param>
    <message>你的主页地址必须是一个有效的网址</message>
  </validator>
  ...
<validators>
```

5、字段校验器配置风格

```
<validators>
  <field name="username">
    <field-validator type="required">
      <message>你的主页地址必须是一个有效的网址</message>
    </field-validator>
    ...
  </field>
  ...
<validators>
```

4.2.9.9. Visitor 校验器

1、名字：visitor

2、功能：Visitor 校验器主要用于检测 Action 里的复合属性，例如一个 Action 里包含了 User 类型的属性

3、可接受参数

- fieldName：该参数指定校验的 Action 属性名，如果采用字段校验器风格，则无需指定该参数
- context：指定属性对应 Action 的校验规则文件的???
- appendPrefix：若为 true，在提示信息中增加前缀

4、字段校验器配置风格

```
<validators>
  <field name="user">
    <field-validator type="visitor">
      <param name="context">userContext</param>
      <param name="appendPrefix">true</param>
      <message>用户的：</message>
    </field-validator>
    ...
  </field>
  ...
```

<validators>

- 以上的校验规则并未指定 User 类里各字段应该遵守怎样的校验规则，因此还必须为 User 类指定对应的校验规则文件
- 默认情况下，该校验文件的规则文件名为 User-validation.xml，因为配置 Visitor 校验器时指定了 context 为 userContext，于是校验规则文件的文件名为 User-userContext-validation.xml(该文件应该放在与 User.class 相同的路径下)

4.2.9.10. 转换校验器

1、名字：conversion
2、功能：它检查被校验字段在类型转换过程中是否出现错误
3、可接受参数

- fieldName：该参数指定校验的 Action 属性名，如果采用字段校验器风格，则无需指定该参数
- repopulateField：该参数指定当类型转换失败后，返回 input 页面时，类型转换失败的表单域是否保留原来的错误输入

4、非字段校验器配置风格

```
<validators>
  <validator type="conversion">
    <param name="fieldName">age</param>
    <param name="repopulateField">false</param>
    <message>你的年龄必须是一个整数</message>
  </validator>
  ...
<validators>
```

5、字段校验器配置风格

```
<validators>
  <field name="age">
    <field-validator type="conversion">
      <param name="repopulateField">false</param>
      <message>你的年龄必须是一个整数</message>
    </field-validator>
    ...
  </field>
  ...
<validators>
```

4.2.9.11. 字符串长度校验器

1、名字：stringlength
2、功能：它要求被校验字段长度必须在指定的范围之内，否则就校验失败
3、可接受参数

- fieldName：该参数指定校验的 Action 属性名，如果采用字段校验器风格，则无需指定该参数
- maxLength：该参数指定字段值的最大长度，该参数可选，如果不指定该参数，则最大长度不受限制

- minLength：该参数指定字段值的最小长度，该参数可选，如果不指定该参数，则最小长度不受限制
- trim：指定校验该字段之前是否截断该字段值前后的空白，该参数可选，默认 true

4、非字段校验器配置风格

```
<validators>
  <validator type="stringlength">
    <param name="fieldName">user</param>
    <param name="minLength">4</param>
    <param name="maxLength">20</param>
    <message>你的用户名长度必须在 4 到 20 之间</message>
  </validator>
  ...
<validators>
```

5、字段校验器配置风格

```
<validators>
  <field name="user">
    <field-validator type="stringlength">
      <param name="minLength">4</param>
      <param name="maxLength">20</param>
      <message>你的用户名长度必须在 4 到 20 之间</message>
    </field-validator>
    ...
  </field>
  ...
<validators>
```

4.2.9.12. 正则表达式校验器

1、名字：regex

2、功能：它检查被校验字段是否匹配一个正则表达式

3、可接受参数

- fieldName：该参数指定校验的 Action 属性名，如果采用字段校验器风格，则无需指定该参数
- regex：该参数是必须的，指定匹配用的正则表达式
- caseSensitive：该参数指明进行正则表达式匹配时，是否区分大小写，该参数可选，默认 true

4、非字段校验器配置风格

```
<validators>
  <validator type="regex">
    <param name="fieldName">user</param>
    <param name="regex">![CDATA[(\w{4,20})]]</param>
    <message>用户名长须在 4 到 20 之间，且必须是字母和数字
    </message>
  </validator>
  ...
```

```
    <validators>
```

5、字段校验器配置风格

```
    <validators>
      <field name="user">
        <field-validator type="regex">
          <param name="regex">![CDATA[(\w{4,20})]]</param>
          <message>用户名长须在 4 到 20 之间，且必须是字母和数字
          </message>
        </field-validator>
        ...
      </field>
      ...
    <validators>
```

4. 2. 10. 基于注解的输入校验

1、基于注解的输入校验实质上也属于 Struts2"零配置"特性的部分，它允许使用注解来定义每个字段应该满足的规则，Struts2 在 com.opensymphony.xwork2.validator.annotations 包下提供了大量校验器相关的 Annotation，这些 Annotation 和前面介绍的校验器大致上一一对应

2、这些注解实质上属于 Struts2 的"零配置"特性，但由于这种特性并不是 Convention 插件提供的，而是由 XWork 框架提供的，因此不需要 Convention 插件

3、在 setter 方法前使用注解

4、使用注解来代替 XML 配置文件，这是 JDK1.5 新增注解后的一个趋势，使用这种方式无需编写 XML 文件，从而可以简化应用开发，但带来的副作用是所有内容都被写入 Java 代码中，会给后期维护带来一定困难

4. 2. 11. 手动完成输入校验

1、基于 Struts2 校验器的校验可以完成绝大部分输入校验，但这些校验器都具有固定的校验逻辑，无法满足一些特殊的校验规则；**对于一些特殊的校验要求，可能需要在 Struts2 中进行手动校验，Struts2 提供了良好的可扩展性，从而允许通过手动方式完成自定义校验**

2、重写 validate()方法

- 通过重写 ActionSupport 类的 validate 方法
- 一旦校验失败，就把校验失败提供通过 addFieldError()方法添加进系统的 FieldError 中，这与类型转换失败后的处理是完全一样的，除此之外，程序无须做额外的处理
- 如果 Struts2 发现系统的 FieldError 不为空，将会自动跳转到 input 逻辑视图(这个 input 逻辑视图是对应特定例子来说明的)
- 可以通过 ActionSupport 类里的 getText()方法来取得国际化信息

3、重写 validateXxx()方法

- 如果 Struts2 的 Action 类里可以包含多个处理逻辑，不同的处理逻辑对应不同的方法
- 如果输入校验指向校验某个处理逻辑，也就是仅校验某个处理方法，那

么如果重写了 Action 的 validate()方法，则该方法将会校验所有的处理逻辑

- 为了实现校验指定处理逻辑的功能，Struts2 的 Action 允许提供一个 validateXxx()方法，其中 xxx 即是 Action 对应的处理逻辑方法
- 例如

```
public String regist(){...}
pubilc void validateRegist(){...}
```

- **无论向 Action 哪个方法发送请求，Action 内的 validate()方法都会被调用，如果该 Action 内还有方法对应的 validateXxx()方法，则该方法会在 validate()方法之前被调用**
- **通过重写 validate()或 validateXxx()方法来进行输入校验时，如果开发者在这两个方法中添加 FieldError，但 Struts2 的表单标签不会自动显示这些 FieldError 提示，必须使用<s:filederror/>标签来显式输出错误提示**

4、Struts2 的输入校验需要经过如下几个步骤

1) 类型转换器负责对字符串的请求参数执行类型转换，并将这些值设置成 Action 的属性值
2) 在执行类型转换过程中可能出现异常，如果出现异常，将异常信息保存到 ActionContext 中，conversionError 拦截器负责将其封装到 FieldError 里，然后执行第 3 步，如果转换过程没有异常信息，则直接进入第 3 步
3) 使用 Struts2 应用中所配置的校验器进行输入校验
4) 通过反射调用 validateXxx 方法()，其中 Xxx 是即将处理用户请求的处理逻辑所对应的方法
5) 调用 Action 里的 validate()方法
6) 如果经过上面 5 步都没有出现 FieldError，将调用 Action 里处理用户请求的处理方法；如果出现了 FieldError，系统将转入 input 逻辑视图所指定的视图资源

4.3. 使用 Struts2 控制文件上传

1、**为了能上传文件，必须将表单的 method 设置为 POST，将 enctype 设置为 multipart/form-data，只有在这种情况下，浏览器才会把用户选择文件的二进制数据发送给服务器**

2、一旦设置了 enctype 为 multipart/form-data，此时浏览器将采用二进制流的方式来处理表单数据

3、Struts2 的文件上传还没有来得及使用 Servlet 3.0API，因此 Struts2 的文件上传还需要依赖于 Common-FieldUpload、COS 等文件上传组件

4.3.1. Struts2 的文件上传

1、Struts2 并未提供自己的请求解析器，也就是说，Struts2 不会自己去处理 multipart/form-data 的请求，它需要调用其他上传框架来解析二进制请求数据，但 Struts2 在原有的上传解析器基础上做了进一步封装，更进一步简化了文件上传

2、在 Struts2 的 struts.properties 配置文件中对上传的上传解析器进行配置，即

修改 struts.multipart.parser 常量即可

- struts.multipart.parser=cos
- struts.multipart.parser=pell
- struts.multipart.parser=jakarta

3、**Struts2 的封装隔离了底层文件上传组件的区别**，开发者只需要在此处配置文件上传所使用的解析器，就可以轻松地在不同文件上传框架之间切换

4、Struts2 默认使用的是 Jakarta 的 Common-FieldUpload 的文件上传框架，因此，如果需要使用 Struts2 的文件上传功能，则需要在 Web 应用中增加两个 JAR 文件，即 commons-io-2.2.jar 和 commons-fileupload-1.3.1.jar，将 Struts2 项目的 lib 下的这两个文件复制到 Web 应用的 WEB-INF/lib 路径下即可

4.3.2. 实现文件上传的 Action

1、Struts2 的 Action 无需负责处理 HttpServerRequest 请求，正如前面介绍的，Struts2 的 Action 已经与 ServletAPI 彻底分离了，Struts2 框架负责解析 HttpServletRequest 请求中的参数，包括文件域，Struts2 使用 File 类型来封装文件域

2、**如果表单中包含一个 name 属性为 xxx 的文件域，则对应 Action 需要使用三个成员变量来封装该文件域的信息**

- 类型为 File 的 xxx 成员变量封装了该文件域对应的文件内容(File)
- 类型为 String 的 xxxFileName 成员变量封装了该文件域对应的文件的文件名(String)
- 类型为 String 的 xxxContentType 成员变量封装了该文件域对应的文件的文件类型(String)

3、Struts2 的 Action 中的属性，功能非常丰富，除了可用于封装 HTTP 请求参数外，也可以封装 Action 的处理结果，不仅如此，Action 的属性还可通过在 Struts2 配置文件中进行配置，接受 Struts2 框架注入，允许在配置文件中为该属性动态指定值

4.3.3. 配置文件上传的 Action

1、配置文件上传的 Action

- 与配置普通 Action 相同，同样需要指定该 Action 的 name 以及该 Action 的实现类，以及为该 Action 配置<result.../>元素
- 区别是，该 Action 还配置了一个<param.../>元素，该元素用于为该 Action 的属性动态分配属性值

2、通过 Struts2，只要将文件域与 Action 中一个类型为 File 的成员变量关联，就可以轻松访问到上传文件的文件内容---置于 Struts2 使用何种 Multipart 解析器，对开发者完全透明

3、**Struts2 实现文件上传的编程关键**：使用三个成员变量来封装文件域，其中一个用于封装该文件的文件名，一个用于封装该文件的文件类型，一个用于封装该文件内容

4.3.4. 手动实现文件过滤

1、大部分时候，Web 应用不允许浏览者自由上传，尤其不能上传可执行文件--因为可能是病毒程序。通常只可以允许浏览者上传图片、上传压缩文件等。除

此之外，还必须对浏览者上传的文件大小进行限制，因此必须在文件上传中进行文件过滤

2、如果要实现手动过滤，可以按照如下步骤进行

- 在 Action 中定义一个专用于进行文件过滤的方法，该方法的方法名字是任意的，该方法的逻辑就是判断上传文件类型是否为允许类型

```
/**
* 过滤文件类型
* @param types 系统所有允许上传的文件类型
* @return 如果上传文件的文件类型允许上传，返回 null，否则返回 error 字
*                                    符串
*/
public String filterTypes(String[] types){
        String fileType=getUploadContentType();
        for(String type:types){
                if(type.equals(fileType)){
                        return null;
                }
        }
        return ERROR;
}
```

- 上述方法判断了上传文件的文件类型是否在允许上传文件类型列表中，为了让应用程序可以动态配置允许上传的文件列表，为该 Action 增加一个 allowTypes 属性，该属性的值列出了所有允许上传的文件类型，为了可以在 struts.xml 文件中配置 allowTypes 属性的值，必须在 Action 类中提供如下代码

```
private String allowTypes;
public Strig getAllowTypes(){
        return allowTypes;
}
public void setAllowTpes(String allowTypes){
        this.allowTypes=allowTypes;
}
```

- 利用 Struts2 的输入校验来判断用户输入的文件是否符合要求，如果不符合要求，接下来就将错误提示添加到 FieldError 中

```
public void validate(){
        String filterResult=filterType(getAllowTypes().split(","));
        if(filterResult!=null){
                addFieldError("upload","您要上传的文件类型不正确!");
        }
}
```

3、实现文件大小过滤，与实现文件类型过滤的方法基本相似，虽然在 Action 类中并没有方法直接获取上传文件的大小，但 Action 中包含了一个类型为 File 的属性，该属性封装了文件域对应的文件内容，而 File 类有一个 length()方法，该方法可以返回文件的大小，通过比较该文件的大小和允许上传的文件大小，从而决定是否允许上传该文件

4.3.5. 拦截器实现文件过滤

1、上面手动实现文件过滤的方式虽然简单，但毕竟要书写大量的过滤代码，不利于程序的高层次解构，而且开发复杂

2、**Struts2 提供了一个文件上传的拦截器，通过配置该拦截器可以更轻松地实现文件过滤，Struts2 中文件上传的拦截器是 fileUpload，为了让该拦截器起作用，只需要在该 Action 中配置该拦截器即可**

3、配置 fileUpload 拦截器时，可以指定连个参数

- allowedTypes：该参数指定允许上传的文件类型，多个文件类型之间以英文逗号隔开
- maximumSize：该参数指定允许上传的文件大小，单位是字节

4、通过配置 fileUpload 的拦截器，可以更轻松地实现文件过滤，当文件过滤失败后，系统自动转入 input 逻辑视图，因此必须为该 Action 配置名为 input 的逻辑视图，除此之外，还必须显式地为该 Action 配置 defaultStack 的拦截器引用

- 配置例子，详见 P341

4.3.6. 输出错误提示

1、如果上传失败，系统返回 input 逻辑视图，Struts2 的标签可以自动显示上传失败的提示信息，并让浏览者重新上传，当然也可以使用<s:fielderror/>来显式输出上传失败的校验提示

2、key

- 文件太大的提示信息的 key：struts.messages.error.file.too.large
- 文件类型错误的 key：struts.messages.error.content.type.not.allowed
- 非以上两个错误的未知错误的 key：struts.messages.error.uploading

4.3.7. 文件上传的常量配置

1、struts.multipart.saveDir：指定临时文件夹，Struts2 默认使用 javax.servlet.context.tempdir，即在 Tomcat 安装路径下的 work/Catalina/localhost/路径

2、struts.multipart.maxSize：该属性设置真个表单请求内容的最大字节数

4.4. 使用 Struts2 控制文件下载

1、Struts2 提供了 stream 结果类型，该结果类型就是专门用于支持文件下载功能的，指定 stream 结果类型时，需要指定一个 inputName 参数，该参数指定了一个输入流，这个输入流是被下载文件的入口

2、通过 Struts2 的文件下载支持，允许系统控制浏览者下载文件的权限，包括实现文件名为非修字符的文件下载

4.4.1. 实现文件下载的 Action

1、文件下载，可以直接在页面上给出一个链接，该链接的 href 属性等于要下载文件的文件名，就可以实现下载了。但是如果文件名为中文名，在某些早期的浏览器上就会导致下载失败，如果应用程序要在用户下载之前进行进一步检查，包括判断用户是否有足够权限来下载该文件等，就需要让 Struts2 来控制下载了

2、Struts2 的文件下载 Action 与普通 Action 并没有太大的不同，仅仅是该 Action

需要提供一个返回 InputStream 流的方法，该输入流代表了被下载文件的入口

4.4.2. 配置 Action

1、配置该文件下载的 Action 与配置普通 Action 并没有太大不同，**关键是需要配置一个类型为 stream 的结果，该 stream 类型的结果将使用文件下载作为响应**

2、配置 stream 结果需要指定如下 4 个属性

- contentType：指定被下载文件的文件类型
- inputName：指定被下载文件的入口输入流
- contentDisposition：指定下载的文件名(下载下来的文件的文件名，并非是在服务器上的那个文件名)
- bufferSize：指定下载文件时的缓冲大小

3、stream 结果类型的逻辑视图是返回给客户端的一个输入流，因此无需指定 location 属性

4.4.3. 下载前的授权控制

1、通过 Struts2 的下载支持，应用程序可以在用户下载文件之前，先通过 Action 来检查用户是否有权下载该文件，就可以实现下载前的授权控制

4.5. 详解 Struts2 的拦截器机制

1、拦截器体系是 Struts2 框架的重要组成部分，可以把 Struts2 理解成一个空容器，而大量的内建拦截器完成了该框架的大部分操作

- params 拦截器负责解析 HTTP 请求的参数，并设置 Action 的属性
- servlet-config 拦截器直接将 HTTP 请求中 HttpServletRequest 实例和 HttpServletResponse 实例传给 Action
- fileUpload 拦截器则负责解析请求参数中的文件域，并将一个文件域设置成 Action 的三个属性
- 等等...

2、Struts2 拦截器是可插拔式设计：

- 如果需要使用某个拦截器，只需要在配置文件中应用该拦截器即可
- 如果不需要使用某个拦截器，只需要在配置文件中取消应用该拦截器
- 无论是否应用某个拦截器，对于 Struts2 框架不会有任何影响

3、Struts 拦截器由 struts-default.xml、struts.xml 等配置文件进行管理，所以开发者很容易扩展自己的拦截器，从而可以最大限度地扩展 Struts2 框架

4.5.1. 拦截器在 Struts 中的作用

1、对于任何 MVC 框架来说，它们都会完成一些通用的控制逻辑，例如解析请求参数、类型转换，将请求参数封装成 DTO(Data Transfer Object)，执行输入校验，解析文件上传表单中的文件域，放置表单多次提交...

2、早期的 Struts1 框架把这些动作都写死在系统的核心控制器里，这样做的缺点有如下两个

- 灵活性非常差：这种框架强制所有项目都必须使用该框架提供的全部功能，不管用户是否需要，核心控制器总是会完成这些操作
- 可扩展性很差：如果用户需要让核心控制器完成更多自定义的处理，这

就比较困难了，在 Struts1 时代需要通过扩展 Struts1 的核心控制器来实现

3、Struts2 改变了这种做法，它把大部分核心控制器需要完成的工作按功能分开定义，每个拦截器完成一个功能，这些拦截器可以自由选择，灵活组合(甚至不用 Struts2 的任何拦截器)，开发者需要使用哪些拦截器，只需要在 struts.xml 文件中指定使用该拦截器即可

4、Struts2 框架的绝大部分功能都是通过拦截器来完成的，当 StrutsPrepareAndExecuteFilter 拦截到用户请求之后，大量拦截器将会对用户请求进行处理，然后才会调用用户开发的 Action 实例的方法来处理请求

5、Struts2 默认启用了大量通用功能的拦截器，只要配置 Action 时所在的 package 继承了 struts-default 包，这些拦截器就会起作用

4.5.2. Struts2 内建的拦截器

1、从 Struts 框架来看，拦截器几乎完成了 Struts2 框架的 70%的工作，包括解析请求参数，将请求参数赋值给 Action 属性，执行数据校验，文件上传

- 当需要扩展 Struts2 功能时，只需要提供对应拦截器，并将它配置在 Struts2 容器中即可
- 当不需要某功能时，也只需要取消该拦截器的配置即可
- 这种插拔式的设计，正是软件领域孜孜以求的目标

2、Struts2 内建了大量的拦截器，这些拦截器以 name-class 对的形式配置在 struts-default.xml 文件中

- name 是拦截器的名字，就是以后使用该拦截器的唯一标识
- class 则指定了该拦截器的实现类，如果程序定义的 package 继承了 Struts2 的默认 struts-defautl 包，则可以使用下面定义的拦截器，否则必须自己定义这些拦截器

3、Struts2 内建拦截器的介绍

- 详见 P347

4.5.3. 配置拦截器

1、在 struts.xml 文件中定义拦截器只需为拦截器类指定一个拦截器名，就完成了拦截器定义，拦截器使用<interceptor.../>元素来定义，格式如下

- 大部分时候，只需要通过下面的格式就可以完成拦截器的配置

```
<interceptor name="拦截器名" class="拦截器实现类"/>
```

- 如果还需要在配置拦截器时传入拦截器参数，则需要在<interceptro.../>元素中使用<param.../>子元素

```
<interceptor name="拦截器名" class="拦截器实现类">
    <param name="参数名">参数值</param>
</interceptor>
```

- 还可以把多个拦截器连在一起组成拦截器栈
  - 例如如果需要在 Action 执行前做登陆检查、安全检查和日志记录，则可以把这三个动作对应的拦截器组成一个拦截器栈
  - 定义拦截栈使用<interceptor-stack.../>，拦截器栈时由多个拦截器组成的，因此需要在<interceptor-stack.../>元素中使用<interceptor-ref.../>元

素来定义多个拦截器引用，即该拦截器栈由多个<interceptor-ref.../>元素指定的拦截器组成

- 从程序结构上来看，拦截器栈是由多个拦截器组成的，即一个拦截器栈包含多个拦截器，**但是从程序功能上来看，拦截器栈和拦截器是统一的，它们包含的方法都会在 Action 的 execute 方法(或者指定的其他方法???)执行前自动执行，因此完全可以把拦截器栈当成一个更大的拦截器**

```
<interceptor-stack name="拦截器栈名">
  <interceptor-ref name="拦截器 1"/>
  <interceptor-ref name="拦截器 2"/>
  ...
</interceptor-stack>
```

➢

2、系统为拦截器指定参数有如下两个时机

- ➢ 定义拦截器时指定参数值：这种参数值将作为拦截器参数的默认参数值
- ➢ 使用拦截器时指定参数值：在配置 Action 时为拦截器参数指定值
- ➢ 通过为<interceptor-ref.../>元素增加<param.../>子元素，就可在使用拦截器时为参数指定值

4.5.4. 使用拦截器的配置语法

1、一旦定义了拦截器和拦截器栈后，就可以使用这个拦截器或拦截器栈来拦截 Action 了，拦截器(包含拦截器栈)的拦截行为将会在 Action 的 execute 方法执行之前被执行

2、通过<interceptor-ref.../>元素可以在 Action 内使用拦截器，在 Action 中使用拦截器的配置语法，与配置拦截器栈时引用拦截器的语法完全一样

4.5.5. 配置默认拦截器

1、当配置一个包时，可以为其指定默认拦截器

- ➢ 一旦为某个包指定了默认拦截器，**如果包中的 Action 没有显式指定拦截器，则默认的拦截器将会起作用(注意，只有在 Action 没有显式指定拦截器时，包的拦截器才起作用)**
- ➢ 如果一旦为该包中的 Action 显式应用了某个拦截器，则默认的拦截器不会起作用，如果该 Action 还需要默认的拦截器，则必须手动配置该拦截器的引用

2、配置默认拦截器使用<default-interceptor-ref.../>元素，该元素作为<package.../>元素的子元素使用，为该包下所有 Action 配置默认的拦截器

- ➢ 配置<default-interceptor-ref.../>元素时，需要指定一个 name 属性，该 name 属性值是一个已经存在拦截器(栈)的名字，表明该拦截器(栈)配置成该包的默认拦截器
- ➢ 每个<package.../>只能有一个<default-interceptor-ref.../>子元素，即每个包只能有一个默认拦截器
- ➢ 如果确实需要指定多个拦截器共同作为默认拦截器，应该讲这个拦截器定义成拦截器栈，然后把这个拦截器配置成默认拦截器即可

3、配置默认拦截器是一种使用拦截器的方式---避免在每个 Action 中单独配置拦截器，通过在该包下配置拦截器，可以实现为该包下所有 Action 同时配置相同的拦截器

4、通常，自定义的包会继承自 struts-default(通过 extends 属性)，而 struts-default 包下已经配置了默认的拦截器，因此之前的例子中都没有配置过拦截器

5、总结

- <interceptors.../>元素：定义拦截器，所有拦截器和拦截器栈都在该元素下定义
- <interceptor.../>元素：该元素用于定义单个拦截器，定义拦截器只需要指定两个属性，即 name 和 class，分别指定拦截器的名称和实现类
- <interceptor-stack.../>元素：该元素用于定义拦截器栈，该元素中包含多个<interceptor-ref.../>元素，用于将多个拦截器或拦截器栈组合成一个新的拦截器
- <interceptor-ref.../>元素：该元素引用一个丽娜姐漆或拦截器栈，表明应用指定拦截器，该元素只需指定 name 属性，该属性值为一个已经定义的拦截器或拦截器栈，该元素可以作为<interceptor-stack.../>和<action.../>元素的子元素使用
- <param.../>：该元素用于为拦截器指定参数，可以作为<interceptor.../>和<interceptor-ref.../>元素的子元素使用
- <default-interceptor-ref.../>：该元素为指定包配置默认拦截器，该元素作为<package.../>元素的子元素使用

4.5.6. 实现拦截器类

1、Struts2 拦截器系统非常简单、易用

2、如果用户要开发自己的拦截器类，应该实现 com.opensymphony.xwork2.interceptor.Interceptor 接口，该接口代码如下

```
public interface Interceptor extends Serializable{
    //销毁该拦截器之前的回调方法
    void destroy();
    //初始化该拦截器的回调方法
    void init();
    //拦截器实现拦截的逻辑方法
    String intercept(ActionInvocation invocation) throws Exception;
}
```

- init()：在该拦截器被实例化之后，在该拦截器执行拦截器之前，系统将回调该方法，对于每个拦截器而言，其 init()方法只执行一次，因此该方法体主要用于初始化资源，例如数据库连接等
- destroy()：该方法与 init()方法对应，在拦截器实例被销毁之前，系统将回调该拦截器的 destroy 方法，该方法用于销毁在 init()方法里打开的资源
- intercept(ActionInvocation invocation)：该方法是用户需要实现的拦截动作，就像 Action 的 execute 方法一样，intercept 方法会返回一个字符串作为逻辑视图
  - 如果该方法直接返回了一个字符串，系统将会跳转到该逻辑视图对应

的实际视图资源，不会调用被拦截的 Action

- 该方法的 ActionInvocation 参数包含了被拦截的 Action 的引用，可以通过调用该参数的 invoke 方法，将控制权转给下一个拦截器，或者转给 Action 的 execute 方法

3、除此之外，Struts2 还提供了一个 AbstractInterceptor 类，该类提供了一个 init()和 destroy()方法的空实现，如果实现的拦截器不需要打开资源，则可以无须实现这两个方法，继承 Abastractinterceptor 类来实现自定义拦截器会更加简单

4.5.7. 使用拦截器

1、使用拦截器需要两个步骤

- 通过<interceptor.../>元素来定义拦截器
- 通过<interceptor-ref.../>元素来使用拦截器

2、Struts2 拦截器的功能非常强大，它既可以在 Action 的 execute 方法之前插入执行代码，也可以在 execute 方法之后插入执行代码，这种方式是指就是 AOP(面向切面)的思想

4.5.8. 拦截方法的拦截器

1、在默认情况下，如果为某个 Action 定义了拦截器，则这个拦截器会拦截该 Action 内的所有方法，若不想拦截所有方法，只需要拦截指定方法，此时就需要使用 Struts2 拦截器的方法过滤特性

2、为了实现方法过滤的特性，Struts2 提供了一个 MethodFilterInterceptor 类，该类是 Abstractinterceptor 类的子类，如果用户需要自己实现的拦截器支持方法过滤特性，则应该继承 MethodFilterInterceptor

- MethodFilterInterceptor 类重写了 AbstractInterceptor 类的 intercept(ActionInvocation invocation)方法，但提供了一个 doIntercept(ActionInvocation invocation)抽象方法
- 可以看出，MethodFilterInterceptor 类的 intercept 已经实现了对 Action 的拦截行为(只是实现了方法过滤的逻辑)，但真正的拦截逻辑还需要开发者提供，也就是通过回调 doIntercept 方法实现
- 如果用户需要实现自己的拦截逻辑，则应该重写 doIntercept(ActionInvocation invocation)方法
- 在 MethodFilterInterceptor 类中，额外增加了如下两个方法(可以在配置文件中进行配置，因为是 setter 方法)
  - public void setExcludeMethods(String excludeMethods)：排除需要过滤的方法，设置方法"黑名单"，所有在 excludeMethods 字符串中列出的方法都不会被拦截
  - public void setIncludeMethods(String includeMethods)：设置需要过滤的方法，设置方法"白名单"，所有在 includeMethods 字符串中列出的方法都会被拦截
  - 如果一个方法同时为位于两个方法所对应的字段中，那么该方法会被拦截
  - 如果需要指定多个方法，那么方法之间以英文逗号隔开

```
...
<interceptor-ref name="mySimple">
   <param name="excludeMethods">execute</param>
</interceptor-ref>
...
```

3、Struts2 中提供了这种方法过滤的拦截器由如下几个

- TokenInterceptor
- TokenSessionStoreInterceptor
- DefaultWorkflowInterceptor
- ValidationInterceptor

4.5.9. 拦截器的执行顺序

1、随着系统中配置拦截器的顺序的不同，系统中执行拦截器的顺序也不一样，通常认为，先配置的拦截器，会先获得执行的机会，但有时候在一些特殊的情况下可能有少许出入

4.5.10. 拦截器结果的监听器

1、简单拦截器在 execute()方法执行之前、执行之后的动作都定义在拦截器的 intercept(ActionInvocation invocation)方法中，这种方式看上去结构不够清晰

2、为了精确定义在 execute()方法执行结束后，在处理物理资源转向之前的动作，Struts2 提供了用于拦截结果的监听器，这个监听器是通过手动注册在拦截器内部的

3、拦截结果的监听器接口：PreResultListener

- 可以在 Action 中注册监听器，那么该监听器将只对指定 Action 有效
- 可以在拦截器中注册监听器，那么该拦截器起作用的地方，拦截接口的监听器都会触发
- 虽然 beforeResult()方法也获得 ActionInvocation 类型的参数，但通过这个参数来控制 Action 已经没有太大作用了，千万不要通过该实例再次调用 invoke 方法，如果再次调用 invoke 方法，将会再次执行 Actin 处理，Action 处理之后紧跟的是 beforeResult，将会进入死循环

4.5.11. 覆盖拦截器栈里特定拦截器的参数

1、有时候，Action 需要使用一个拦截器栈，当使用这个拦截器栈时，又需要覆盖该拦截器栈中某个拦截器的指定参数值

- 此时就需要在配置使用拦截器栈的<interceptor-ref.../>元素中使用<param.../>元素来传入参数
- 在<param.../>元素中指定参数名时应使用：<拦截器名>.<参数名>这种形式，这样才可以让 Struts2 明白程序想覆盖哪个拦截器的哪个参数

4.5.12. 使用拦截器完成权限控制

1、需要实现的逻辑如下：

- 当浏览者需要请求执行某个操作时，应用需要先检查浏览者是否登陆，以及是否有足够的权限来执行该操作

2、对于上述需求，可以在 Action 的执行实际处理逻辑之前，先执行权限检查逻

辑，但这种做法不利于代码复用，因为大部分 Action 里的权限检查代码都大同小异，故将这些权限检查的逻辑放在拦截器中进行将会更加优雅
3、检查用户是否登陆，通常都是通过跟踪用户的 Session 来完成的，通过 ActionContext 即可访问到 session 中的属性，拦截器的 intercept(ActionInvocation invocation)方法的 invocation 参数可以轻松地访问到请求相关的 ActionContext 实例

4.6. 使用 Struts2 的 Ajax 支持

1、Ajax(Asynchronous JavaScript And XML)，即异步 JavaScript 和 XML 技术，也是 Web 2.0 核心技术之一
2、Ajax 技术改进了传统的 Web 技术；通过 Ajax 技术，浏览者与服务器之间采用异步通信机制，从而避免了浏览者的等待，带给浏览者连续的体验

- 它让用户可以连续发送多次异步请求，而无需等待服务器响应
- 当服务器成功响应成功返回浏览器时，浏览器使用 DOM(Dcument Object Model)将服务器响应装载到当前页面的指定容器内

3、传统的 Web 应用大都采用一种独占式的请求方法，每个请求对应一个页面，因此每当服务器响应到达客户端时，浏览器都会重新转载该响应，从而导致频繁的页面刷新
4、由于传统 B/S 结构应用里每个页面的使用时间都很短暂(只用于一次发送请求，或一次装载服务器响应)，因此不可能将页面制作成表现功能丰富的页面(这样用户端下载成本太高)，所以传统 B/S 结构应用的表现层页面都很简陋
5、Ajax 技术的出现，完善了传统的 Web 应用的不足，Ajax 技术使用异步方式发送用户请求：

- 用户在浏览页面的同时可以发送异步请求，在第一个请求的服务器响应还没有完成时，浏览器可以再次发送请求，页面状态不会停止，即使服务器响应还没有到达，浏览者还可以浏览原来的页面
- 当服务器响应到达客户端时，浏览器也无须重新加载整个页面，它只更新页面的部分数据，从而提高了页面的利用时间(可以使用一个页面发送无数个请求，装载无数次响应)，因此可以将表现层页面制作成表现功能非常丰富的页面

4.6.1. 使用 stream 类型的 Result 实现 Ajax

1、Struts2 支持一种 stream 类型的 Result，这种类型的 Result 可以直接向客户端浏览器生成二进制响应、文本响应等。那么就可以让 Struts2 的 Action 来直接生成文本响应，接下来在客户端页面动态加载该响应即可

4.6.2. JSON 的基本知识

1、JSON 的全称是 JavaScript Object Notation，即 JavaScript 对象符号，它是一种轻量级的数据交换格式

- JSON 的数据格式既适合人来读/写，也适合计算机本身解析和生成
- 最早的时候，JSON 是 JavaScript 语言的数据交换格式，后来慢慢发展成一种语言无关的数据交换格式，这一点非常类似于 XML

2、JSON 主要类似于 C 的编程语言中广泛使用，包括 C、C+

+、C#、Java、JavaScript、Perl、Python 等

- JSON 提供了多种语言之间完成数据交换的能力，因此 JSON 也是一种非常理想的数据交换格式
- JSON 主要有如下两种数据结构
  - 有 key-value 组成的数据结构，这种数据结构在不同语言中有不同的实现。例如在 JavaScript 中是一个对象，在 Java 中是一种 Map 结构，在 C 语言中则是一个 struts，在其他语言中，可能有 record、dictionary、hash table 等
  - 有序集合，这种数据结构在不同语言中，可能有 list、vector 数组和序列等实现

3、在 JavaScript 中主要有两种 JSON 的语法，一种用于创建对象，一种用于创建数组

4、使用 JSON 语法创建对象

- JSON 创建对象以"{"开始，以"}"结束，对象的每个属性名和属性值之间以英文冒号:隔开，多个属性定义之间以英文逗号","隔开

```
object=
{
    propertyName1 : propertyValue1 ,
    propertyName2 : propertyValue2 ,
    ...
    propertyName2 : propertyValue2   <==最后一个不能有逗号哦
}
```

- 使用 JSON 语法创建 JavaScript 对象时，属性值不仅可以是普通字符串，也可以是任何基本数据类型，还可以是函数、数组，甚至是另外一个 JSON 语法创建的对象

5、使用 JSON 语法创建数组

- JSON 创建数组以英文方括号"["开始，然后依次放入数组元素，元素与元素之间以英文逗号","隔开，最后一个数组元素后面不需要英文逗号，但以英文反方括号"]"结束

```
arr = [value 1, value 2,..., value n]
```

4.6.3. 实现 Action 逻辑

1、使用@JSON 注解，该注解支持如下几个属性

- serialize：设置是否序列化该属性
- deserialize：设置是否反序列化该属性
- format：设置用于格式化输出、解析日期表单域的格式

2、<未完成>：啥意思，没看懂

4.6.4. JSON 插件与 json 类型的 Result

1、JSON 插件提供了一种 json 类型的 Result，一旦为某个 Action 指定了一个类型为 json 的 Result，则该 Result 无须映射到任何视图资源，因为 JSON 插件会负责将 Action 里的状态信息序列化成 JSON 格式的字符串，并将该字符串返回给客户端浏览器

2、简单地说，JSON 插件允许在客户端页面的 JavaScript 中异步调用 Action，而

且 Action 不再需要使用视图资源来显示该 Action 里的状态信息，而是由 JSON 插件负责将 Action 里的状态信息返回给调用页面--通过这种方式，就可以完成 Ajax 交互

3、配置提供返回 JSON 字符串的 Action

- 配置 struts.i18n.encoding 常量，不再使用 GBK 编码，而是使用 UTF-8 编码，这是因为 Ajax 的 POST 请求都是以 UTF-8 的方式进行编码的
- 配置包时，继承自 json-default，只有在该包下此案有 json 类型的 Result

4、一旦将某个逻辑视图名配置成 json 类型，这意味着该逻辑视图无需指定物理视图资源，因为 JSON 插件会将该 Action 序列化后发给客户端

4.6.5. 实现 JSP 页面

1、<未完成>：不懂，前端的代码，呵呵

Chapter 5. Hibernate的基本用法

1、Hibernate是轻量级Java EE应用的持久层解决方案，Hibernate不仅管理Java类到数据库表的映射(包括Java数据类型到SQL数据类型的映射)，还提供数据查询和获取数据的方法，可以大幅度缩短处理数据持久化的时间

2、目前主流数据库依然是关系数据库，而Java语言则是面向对象的编程语言，当把二者结合在一起使用时相当麻烦，而Hibernate则减少了这个问题的困扰，它完成对象模型和基于SQL的关系模型的映射关系，使得应用开发者可以完全采用面向对象的方式来开发应用程序

3、Hibernate较之另一个持久层框架MyBatis，Hibernate更具有面向对象的特征

- 收到Hibernate的影响，Java EE 5规范抛弃了传统的Entity EJB，改为使用JPA作为持久层解决方案，而JPA实体完全可以当成Hibernate PO(Persistent Object，持久化对象)使用
- Hibernate倡导低侵入式的设计，完全采用普通的Java对象(POJO，Plain Old Java Object)，不要求PO继承Hibernate的某个超类或实现Hibernate的某个接口

4、Hibernate充当了面向对象的程序设计语言和关系数据库之间的桥梁，Hibernate允许程序开发者采用面向对象的方式来操作关系数据库

5、因为有了Hibernate的支持，使得Java EE应用的OOA(面向对象分析)、OOD(面向对象设计)和OOP(面向对象编程)三个过程一脉相承，成为一个整体

5.1. ORM和Hibernate

1、Java，C#等流行的编程语言，都是面向对象的编程语言，而且主流的数据库产品，例如Oracle、DB2等，依然是关系数据库。编程语言和底层数据库的发展不协调，催生出了ORM框架，ORM框架可作为面向对象编程语言和数据库之间的桥梁

5.1.1. 对象/关系数据库映射(ORM)

1、ORM的全称是Object/Relation Mapping，即对象/关系数据库映射

2、ORM可理解为一种规范，**它概述了这类框架的基本特征：完成面向对象的编程语言到关系数据库的映射**

3、ORM框架是面向对象程序设计语言与关系数据库发展不同步时的中间解决方案，随着面向对象数据库的发展，其理论逐步完善，最终会取代关系数据库

4、面向对象的程序设计语言代表了目前程序设计语言的主流趋势，具有如下优势

- 面向对象的建模、操作
- 多态、继承
- 摒弃难以理解的过程
- 简单易用、易理解

5、关系数据库具有以下优势

- 大量数据查找、排序
- 集合数据连接操作、映射
- 数据库访问的并发、事务
- 数据库的约束、隔离

6、采用 ORM 框架之后，应用程序不再直接访问底层数据库，而是以面向对象的方式来操作持久化对象(例如创建、修改、删除等)，而 ORM 框架则将这些面向对象的操作转换成底层的 SQL 操作

7、ORM 的唯一作用：把持久化对象的保存、删除、修改等操作，转换成对数据库的操作

5.1.2. 基本映射方式

1、ORM 工具提供了持久化类和数据表之间的映射关系，通过这种映射关系的过渡，程序员可以很方便地通过持久化类实现对数据表的操作

2、所有的 ORM 工具大致上都遵循相同的映射思路，ORM 基本映射有如下几条映射关系

- 数据表映射**类**：持久化类被映射到一个数据表，程序使用这个持久化类来创建实例、修改属性、删除实例时，系统会自动转换为对这个表进行 CRUD 操作
- 数据表的行映射**对象(即实例)**：持久化类会生成很多实例，每个实例就对应数据表中的一行记录。当程序在应用中修改持久化类的某个实例时，ORM 工具将会转换成对对应数据表中特定行的操作
- 数据表的列映射**对象的属性(字段)**：当程序修改某个持久化对象的指定属性时(持久化实例映射到数据行)，ORM 将会转换成对对应数据表中指定数据行、指定列的操作

5.1.3. 流行的 ORM 框架简介

1、JPA

- JPA 本身只是一种 ORM 规范，并不是 ORM 产品，它是 Java EE 规范制定者向开源世界学习的结果
- JPA 实体与 Hibernate PO 十分相似，甚至 JPA 实体完全可作为 Hibernate PO 类使用，因此很多地方把 Hibernate PO 称为实体
- 相对于其他开源 ORM 框架，JPA 最大的优势在于它是官方标准，因此具有通用性
- 如果应用程序**面向 JPA 编程**，那么应用程序就可以在各种 ORM 框架之间自由切换

2、Hibernate

- 目前最流行的开源 ORM 框架，已经被选作 JBoss 的持久层解决方案
- JBoss 又加入了 Red Hat 组织，因此 Hibernate 属于 Red Hat 组织的一部分

3、MyBatis(早期名称是 iBATIS)

- Apache 软件基金组织的子项目
- 曾经扮演非常重要的角色，因为不支持纯面向对象的操作，因此现在逐渐开始被取代
- 对于一些数据访问特别灵活的地方，MyBatis 更加灵活，它允许开发人员直接编写 SQL 语句

4、TopLink

- Oracle 公司的铲平
- 现在主要作为 JPA 实现，GlassFish 服务器的 JPA 实现就是 TopLink

5.1.4. Hibernate 概述

1、Hibernate 是一个面向 Java 环境的对象/关系数据库映射工具，用于把面向对象模型表示的对象映射到基于 SQL 的关系模型数据库中

2、Hibernate 的目标是释放开发者通常的数据持久化相关的编程任务的 95%

- 对于以数据为中心的程序而言，往往在数据库中使用存储过程实现商业逻辑，Hibernate 可能不是最好的解决方案
- 对于基于 Java 的中间件应用，设计采用面向对象的业务模型和商业逻辑，Hibernate 是最有用的
- Hibernate 能消除那些针对特定数据库厂商的 SQL 代码，并且把结果从表格式的形式转换成值对象的形式

3、Hibernate 不仅管理 Java 类到数据库表的映射(包括 Java 数据类型到 SQL 数据类型的映射)，还提供数据查询和获取数据的方法，可以大幅度减少开发时人工使用 SQL 和 JDBC 处理数据的时间

4、Hibernate 与其他 ORM 框架对比有如下优势

- 开源和免费的 License，方便需要时研究源代码，改写源代码，进行功能指定
- 轻量级封装，避免引入过多复杂的问题，调试容易，减轻程序负担
- 有可扩展性，API 开放
- 开发者活跃，产品有稳定的发展保障

5.2. Hibernate 入门

1、Hibernate 的用法非常简单，只要在 Java 项目中引入 Hibernate 框架，就能以面向对象的方式操作关系数据库了

5.2.1. Hibernate 下载和安装

1、hibernate.org/orm/

2、由于 Hibernate 底层依然是基于 JDBC 的，因此在应用程序中使用 Hibernate 执行持久化同样少不了 JDBC 驱动

5.2.2. Hibernate 的数据库操作

1、在所有 ORM 框架中有一个非常重要的媒介：PO(持久化对象)

- 持久化对象的作用是完成持久化操作
- 通过该对象可以对数据执行曾、删、改的操作---以面向对象的方式操作数据库

2、应用程序无须直接访问数据库，甚至无须理会底层数据库采用何种数据库---这一切对应用程序完全透明

- 应用程序只需要创建、修改、删除持久化对象即可
- Hibernate 则负责把这种操作转换为指定数据表的操作

3、Hibernate 里的 PO 是非常简单的，Hibernate 是低侵入式设计，完全采用普通 Java 对象作为持久化对象使用

- Hibernate 不要求持久化类继承任何父类，或者实现任何接口，这样可以保证代码不被污染

4、普通的 JavaBean 不具备持久化操作的能力，为了使其具备持久化操作的能力，还需要为 POJO 添加一些注解

- @Entity 注解：声明该类是一个 Hibernate 的持久化类
- @Table：指定该类映射的表
- @Id：指定该类的标志属性，即关系数据库表中的主键
- @GeneratedValue：用于指定生成策略

5、PO=POJO+持久化注解

6、通过持久化注解

- Hibernate 可以知道持久化类和数据表之间的映射关系，也可以知道持久化类的属性与数据表的各列之间的对应关系
- 但是无法知道连接哪个数据库，以及连接数据库时所用的连接池、用户名和密码等详细信息，Hibernate 把这些通用信息称为配置信息，配置信息使用配置文件指定

7、Hibernate 配置文件既可以使用*.properties 属性文件，也可以使用 xml 文件配置

- Hibernate 配置文件的默认文件名为 hibernate.cfg.xml
  - 当程序调用 Configuration 对象的 configure()方法时，Hibernate 将自动加载该文件
  - 当调用带参数的 configure()方法时，加载指定的文件
- Hibernate 配置文件的根元素是<hibernate-configuration.../>，根元素里有<session-factory.../>子元素，该元素依次有很多<property.../>子元素，这些<property.../>子元素配置 Hibernate 连接数据的必要信息，如连接数据库的驱动、URL、用户名、密码等信息
- Hibernate 不推荐采用 DriverManager 来连接数据库，而是推荐使用数据源来管理数据库连接，这样能保证最好的性能。Hibernate 推荐使用 C3P0 数据源

8、数据源是一种提高数据库连接性能的常规手段，数据源会负责维持一个数据连接池

- 当程序创建数据源实例，系统会一次性地创建多个数据库连接，并把这些数据库连接保存在连接池中
- 当程序需要进行数据库访问时，无须重新获得数据库连接，而是从连接池中取出一个空闲的数据库连接
- 当传给你续使用数据库连接访问数据库结束后，无须关闭数据库连接，而是将数据库连接归还给连接池即可
- 通过这种方式，可以避免频繁获取数据库连接、关闭数据库连接所导致的性能下降

9、**PO 只有在 Session 的管理下才可完成数据库访问，为了使用 Hibernate 进行持久化操作，通常有如下操作步骤**

1) 开发持久化类，由 POJO+持久化注解组成
2) 获取 Configuration
3) 获取 SessionFactory
4) 获取 Session，打开事务
5) 用面向对象的方式操作数据库

6) 关闭事务，关闭 Session

10、随 PO 与 Session 的关联关系，PO 可以有如下三种状态

- 瞬态：如果 PO 实例从未与 Session 关联过，该 PO 实例处于瞬态状态
- 持久化：如果 PO 实例与 Session 关联起来，且该实例对应到数据库记录，则该实例处于持久化状态
- 脱管：如果 PO 实例曾经与 Session 关联过，但因为 Session 的关闭等原因，PO 实例脱离了 Session 的管理，这种状态被称为脱管状态

11、对 PO 的操作必须在 Session 管理下才能同步到数据库

- Session 由 SessionFactory 工厂产生，SessionFactory 是数据库编译后的内存镜像，通常一个应用对应一个 SessionFactory
- SessionFactory 由 Configuration 对象生成，Configuration 对象负责加载 Hibernate 配置文件

12、Hibernate 的两个显著优点

- 不再需要编写 SQL 语句，而是允许采用 OO 方式来访问数据库
- 在 JDBC 访问过程中大量的 checked 异常被包装成 Hibernate 的 Runtime 异常，从而不再要求程序必须处理所有异常

5.2.3. 在 Eclipse 中使用 Hibernate

1、下载和安装 HibernateTools

- tools.jboss.org/downloads

2、使用方法详见 P385

3、遇到的问题以及解决方法

- 按照书上的步骤做无法成功，原因是 Hibernate 的配置文件问题
- 将上一小结的配置文件复制过来，即可

5.3. Hibernate 的体系结构

1、重要概念：Hibernate Session

- 只有处于 Session 管理下的 POJO 才有持久化操作的能力
- 当应用程序对处于 Session 管理下的 POJO 实例执行操作时，Hibernate 将这种面向对象的操作转换为持久化操作

2、Hibernate 需要 hibernate.properties 文件以及持久化注解

- hibernate.properties 文件用于配置 Hibernate 和数据库的连接信息(通常采用另一种形式的配置文件 hibernate.cfg.xml 文件，这两种文件的实质一样，只是文件形式不同而已)
- POJO 中的持久化注解管理持久化类和数据表、数据列之间的对应关系

3、Hibernate 的持久化解决方案将用户从原始的 JDBC 访问中释放出来，用户无需关注底层的 JDBC 操作，而是以面向对象的方式进行持久层操作。底层数据库连接的获取、数据访问的实现、事务控制都无需用户关心

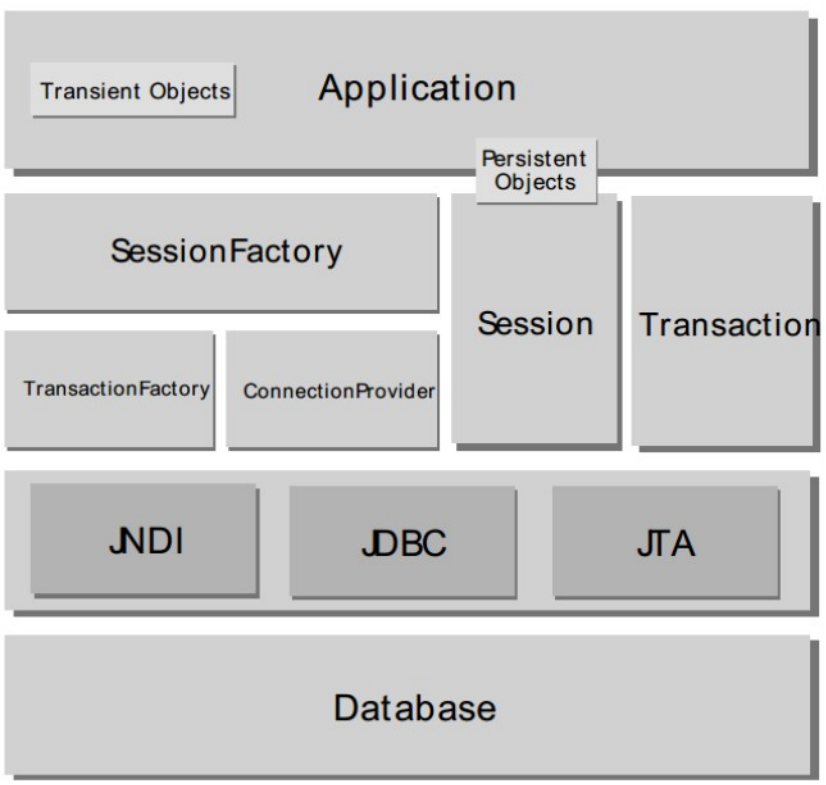

图 52 Hibernate 全面解决方案的体系架构

4、Hibernate 体系架构

- **SessionFactory**：
  - 这是 Hibernate 的关键对象，它是单个数据库映射关系经过编译后的内存镜像，也是线程安全的
  - 它是生成 Session 的工厂，本身需要依赖于 ConnectionProvider
  - 该对象可以在进程或集群的级别上，为那些事务之间可以重用的数据提供可同选择的二级缓存
- **Session**：
  - **它是应用程序与持久存储层之间交互操作的一个单线程对象**
  - **也是 Hibernate 持久化操作的关键对象，所有的持久化对象必须在 Session 管理下才可以进行持久化操作**
  - 此对象生存期很短
  - 它底层封装了 JDBC 连接，它也是 Transaction 的工厂
  - Session 对象持有必选的一级缓存，在显式执行 flush 之前，所有持久化操作的数据都在缓存中的 Session 对象处
- **持久化对象(Persistent Object)**：
  - 系统创建的 POJO 实例，一旦与特定 Session 关联，并对应数据表的指定记录，该对象就处于持久化状态，这一系列对象都被称为持久化对象
  - 在程序中对持久化对象执行的修改，都将自动转换为对持久层的修改
  - 持久化对象完全可以是普通的 JavaBeans/POJO，唯一的区别是它们正与一个 Session 关联
- **瞬态对象和脱管对象**：
  - 系统通过 new 关键字创建的 Java 实例，没有与 Session 相关联，此时

处于瞬态

- 瞬态实例可能是在被应用程序实例化后，尚未进行持久化的对象
- 如果一个曾经持久化过的实例，如果 Session 被关闭则转换为脱管状态

➢ **事务**：

- 代表依次原子操作，它具有数据库事务的概念
- Hibernate 事务是对底层具体的 JDBC、JTA 以及 CORBA 事务的抽象
- 在某些情况下，一个 Session 之内可能包含多个 Transaction 对象
- 虽然事务操作是可选的，但所有持久化操作都应该在事务管理下进行即使是只读操作

➢ **连接提供者**：

- 连接提供者(ConnectionProvider)：它是生成 JDBC 连接的工厂，它通过抽象将应用程序与底层的 DataSource 或 DriveManager 隔离开
- 这个对象无须应用程序直接访问，仅在应用程序需要扩展时使用

➢ **事务工厂(TransactionFactory)**：

- 它是生成 Transaction 对象实例的工厂
- 该对象无须应用程序直接访问
- 它负责对底层具体的事务实现进行封装，将底层具体的事务抽象成 Hibernate 事务

5.4. 深入 Hibernate 配置文件

1、Hibernate 的持久化操作离不开 SessionFactory 对象，这个对象是整个数据库映射关系经过编译后的内存镜像，该对象的 openSession()方法可打开 Session 对象

2、每个 Hibernate 配置文件对应一个 Configuration 对象

5.4.1. 创建 Configuration 对象

1、org.hibernate.cfg.Configuration 实例代表了应用程序到 SQL 数据库的配置信息，Configuration 对象提供了一个 buildSessionFactory()方法，该方法可以产生一个不可变的 SessionFactory 对象

2、Hibernate 所使用配置文件的不同，创建 Configuration 对象的方式也不同

- ➢ 使用 hibernate.properties 文件作为配置文件
- ➢ 使用 hibernate.cfg.xml 文件作为配置文件
- ➢ 不使用任何配置文件，以编码方式创建 Configuration 对象

3、**Configuration 实例的唯一作用就是创建 SessionFactory 实例，所以它被设计成启动期间对象，一旦 SessionFactory 创建完成，它就被丢弃了**

4、使用 hibernate.properties 作为配置文件

- ➢ 该文件没有提供添加 Hibernate 持久化类的方式
- ➢ 使用 hibernate.properties 作为配置文件时，必须调用 Configuration 对象的 addAnnotatedClass()或 addPackage()方法，使用这些方法添加持久化类
- ➢ 人们宁愿选择在配置文件中添加持久化类，也不愿意在 Java 代码中手动添加，这就是在实际中不适用 hibernate.properties 文件作为配置文件的原因

```
//实例化 Configuration
```

Configuration cfg=new Configuration();
cfg.addAnnotatedClass(Item.class);
cfg.addAnnotatedClass(Bid.class);

5、使用 hibernate.cfg.xml 文件作为配置文件

- 可以在该配置文件中添加 Hibernate 持久化类，因此无须通过编程方式添加持久化类

Configuration cfg=new Configuration();
cfg.configure();

6、不使用配置文件创建 Configuration 实例

- 这是一种极端的情况，通常不会用这种方式创建 Configuration 实例
- Configuration 对象提供了如下常用方法
  - Configuration addAnnotatedClass(Class annotatedClass)：用于为 Configuration 对象添加一个持久化类
  - Configuration addPackage(String packageName)：用于为 Configuration 对象添加指定包下的所有持久化类
  - Configuration setProperties(Properties properties)：用于为 Configuration 对象设置一系列属性，这一系列属性通过 Properties 实例传入
  - Configuration setProperty(String propertyName,String value)：用于为 Configuration 对象设置一个单独的属性
- P389
- 实际上，在代码中设置 Hibernate 配置属性也并非完全没有用处，这种策略可与前面两种策略结合，将部分关键的配置属性放在代码中添加

5.4.2. hibernate.properties 文件与 hibernate.cfg.xml 文件

1、在实际项目的开发，通常都会使用 hibernate.cfg.xml 文件作为配置文件

2、指定数据库，在两个文件中的例子

- hibernate.properties 文件

hibernate.dialect org.hibernate.dialect.MySQL5InnoDBDialect

- hibernate.cfg.xml 文件

<property
name="dialect">org.hibernate.dialect.MySQL5InnoDBDialect</property>

5.4.3. JDBC 连接属性

1、JDBC 连接配置中最重要的设置

- hibernate.connection.driver_class：设置连接数据库的驱动
- hibernate.connection.url：设置所需连接数据库服务的 URL
- hibernate.connection.username：设置连接数据库的用户名
- hibernate.connection.password：设置连接数据库的密码
- hibernate.connection.pool_size：设置 Hibernate 数据库连接池的最大并发连接数
  - 该配置属性并不推荐在实际项目中使用
  - 在实际项目中可以使用 C3P0 或 Proxool 连接池，只需要用这些连接池配置替代 hibernate.connection.pool_size 配置属性即可
- hibernate.dialect：设置连接数据库所使用的方言

2、C3P0 配置片段

```
<!-- 指定连接数据库所用的驱动 -->
<property name="connection.driver_class">com.mysql.jdbc.Driver</property>
<!-- 指定连接数据库的 url，其中 hibernate 是本应用连接的数据库名 -->
<property name="connection.url">jdbc:mysql://localhost/hibernate</property>
<!-- 指定连接数据库的用户名 -->
<property name="connection.username">root</property>
<!-- 指定连接数据库的密码 -->
<property name="connection.password">hcflh19930101</property>
<!-- 指定连接池里最大连接数 -->
<property name="hibernate.c3p0.max_size">20</property>
<!-- 指定连接池里最小连接数 -->
<property name="hibernate.c3p0.min_size">1</property>
<!-- 指定连接池里连接的超时时长 -->
<property name="hibernate.c3p0.timeout">5000</property>
<!-- 指定连接池里最大缓存多少个 Statement 对象 -->
<property name="hibernate.c3p0.max_statements">100</property>
<property name="hibernate.c3p0.idle_test_period">3000</property>
<property name="hibernate.c3p0.acquire_increment">2</property>
<property name="hibernate.c3p0.validate">true</property>
<!-- 指定数据库方言 -->
<property
name="dialect">org.hibernate.dialect.MySQL5InnoDBDialect</property>
<!-- 根据需要自动创建数据表 -->
<property name="hbm2ddl.auto">update</property><!--①-->
<!-- 显示 Hibernate 持久化操作所生成的 SQL -->
<property name="show_sql">true</property>
<!-- 将 SQL 脚本进行格式化后再输出 -->
<property name="hibernate.format_sql">true</property>
<!-- 罗列所有持久化类的类名 -->
<mapping class="org.crazyit.app.domain.News"/>
```

5.4.4. 数据库方言

1、Hibernate 底层依然使用 SQL 语句来执行数据库操作，虽然所有关系数据库都支持使用标准 SQL 语句，但所有数据库都对标准 SQL 进行了一些扩展，所以在语法细节上存在一些细节。因此 Hibernate 需要根据数据库来识别这些差异

- 开发者只需要告诉 Hibernate 应用程序底层即将使用哪种数据库---**这就是数据库方言**

2、不同数据库及其对应方言，详细见 P392

5.4.5. JNDI 数据源的连接属性

1、如果无须 Hibernate 自己管理数据源，而是直接访问容器管理数据源，Hibernate 可使用 JNDI(Java Naming Directory Interface，Java 命名目录接口)数据源相关配置

- hibernate.connection.datasource： 指定 JNDI 数据源的名字
- hibernate.jndi.url：指定 JNDI 提供者的 URL，该属性是可选的，如果 JNDI 与 Hibernate 持久化访问代码处于同一个应用中，则无须指定该属性
- hibernate.jndi.class：指定 JNDI InitialContextFactory 的实现类，该属性也是可选的，如果 JNDI 与 Hibernate 持久化访问的代码处于同一个应用中，则无须指定该属性
- hibernate.connection.username：指定连接数据库的用户名，该属性可选
- hibernate.connection.password：指定连接数据库的密码，该属性可选

2、即使使用 JNDI 数据源，也一样需要指定连接数据库的方言。虽然设置方言不是必须的，但对于优化持久层访问很有必要

5.4.6. Hibernate 事务属性

1、事务也是 Hibernate 持久层访问的重要方面，Hibernate 不仅提供了局部事务支持，也允许使用容器管理的全局事务

2、Hibernate 关于事务管理的属性有如下几个

- hibernate.transaction.factory_class：指定 Hibernate 所用的事务工厂类型，该属性必须是 TransactionFactory 的直接或间接子类
- jta.UserTransaction：该属性值是一个 JNDI
- ， Hibernate 将使用 JTATransactionFactory 从应用服务器获取 JTA UserTransaction
- hibernate.transaction.manager_lookup_class ： 该属性值应为一个 TransactionManagerLookup 类名，当使用 JVM 级别的缓存时，或者在 JTA 环境中使用 hilo 生成器策略时，需要该类
- hibernate.transaction.flush_before_completion：指定 Session 是否在事务完成后自动将数据刷新(flush)到底层数据库。该属性只能为 true 或 false。现在更好的办法是使用 Context 相关的 Session 管理
- hibernate.transaction.auto_colse_session：指定是否在事务结束后自动关闭 Session。该属性只能是 true 或 false。现在更好的方法是使用 Context 相关的 Session 管理

5.4.7. 二级缓存相关属性

1、Hibernate 的 SessionFactory 可持有一个可选的二级缓存，通过使用这种二级缓存可以提高 Hibernate 的持久化访问的能力

2、Hibernate 的二级缓存有如下几个属性

- hibernate.cache.use_second_level_cache：用于设置是否启用二级缓存，该属性可完全禁止使用二级缓存
- hibernate.cache.region.factory_class ： 该属性用于设置二级缓存 RegionFactory 实现类的类名
- hibernate.cache.region_prefix：设置二级缓存区名称的前缀
- hibernate.cache.use_minimal_puts：以频繁读操作为代价，优化二级缓存以实现最小化写操作，这个设置对集群缓存非常有用，对集群缓存的实现而言，默认是开启的
- hibernate.cache.query_cache_factory：设置查询缓存工厂的类名，查询缓

存工厂必须实现 QueryCache 接口，该属性默认为内建的 StandardQueryCache

- hibernate.cache.use_structured_entries：用于设置是否强制 Hibernate 以可读性更好的格式将数据存入二级缓存

5.4.8. 外连接抓取属性

1、外连接抓取能限制执行 SQL 语句的次数来提高效率，这种外连接抓取通过在单个 select 语句中使用 outer join 来一次抓取多个数据表的数据

2、外链接抓取允许在单个 select 语句中通过@ManyToOne、@OneToMany、@ManyToMany 和@OneToOne 等关联获取连接对象的整个对象图

3、将 hibernate.max_fetch_depth 设为 0，将在全局方位内禁止外链接抓取，设为 1 或更高值能启用 N-1 或 1-1 的外连接抓取。此外，还应该在持久化注解中通过 fetch=FetchType.EAGER 来指定这种外连接抓取

4、<未完成>：啥几把意思

5.4.9. 其常用的配置属性

1、Hibernate 的其他常用配置属性

- hibernate.show_sql：是否在控制台输出 Hibernate 持久化操作底层所使用的 SQL 语句，只能为 true 和 false
- hibernate.format_sql：是否将 SQL 语句转成格式良好的 SQL，只接受 true 和 false
- hibernate.use_sql_comments：是否在 Hibernate 生成的 SQL 语句中添加有助于调试的注释，只接受 true 和 false
- hibernate.jdbc.fetch_size：指定 JDBC 抓取数量的大小，可接受整数值，实质是调用 Statement.setFetchSize()方法
- hibernate.jdbc.batch_size：指定 Hibernate 使用 JDBC2 的批量更新大小，可接受整数值，建议 5-30
- hibernate.connetcion.autocommit：设置是否自动提交，通常不建议打开自动提交
- hibernate.hbm2ddl.auto：设置当创建 SessionFactory 时，是否根据持久化类的映射关系自动建立数据表

5.5. 深入理解持久化对象

1、Hibernate 是一个纯粹的 O/R Mapping 框架，通过 Hibernate 的支持，程序开发者只需要管理对象的状态，无需理会底层数据库系统的细节

- 相对于常见的 JDBC 持久层方案中需要手动管理 SQL 依据，Hibernate 采用完全面向对象的方式来操作数据库
- 对于程序开发者而言，眼里只有对象、属性，无需理会底层数据表、数据列等概念

3、开发者需要深入了解 Hibernate 底层运行，对 Hibernate 的数据访问进行优化时，就需要了解 Hibernate 的底层 SQL 操作了，对于绝大部分应用来说，数据访

问就是一个巨大的、耗时的操作，因此深入掌握 Hibernate 底层对应的 SQL 操作非常必要

5.5.1. 持久化类的要求

1、Hibernate 采用低侵入式设计，这种设计对持久化类几乎不作任何要求。即 Hibernate 操作的持久化类基本上都是普通的、传统的 Java 对象(JOPO)。对于这种 Java 类，在程序开发中可以采用更灵活的领域建模方式

2、Hibernate 对持久化类米有太多要求，但还是应该遵守如下规则

- **提供一个无参的构造器**：
  - 所有的持久化类都应该提供一个无参的构造器，这个构造器可以不采用 public 访问控制符
  - 只要提供了无参数构造器，Hibernate 就可以使用 Constructor.newInstance()来创建持久化类的实例
  - 通常为了方便 Hibernate 在运行时生成代理，构造器的访问控制修饰符至少是包可见的
- **提供一个标识属性**：
  - 标识属性通常映射数据库表的主键字段，这个属性可以叫任何名字
  - 其类型可以是任何基本类型、基本类型的包装类型、java.lang.String 或者 java.util.Date
  - 如果使用数据库表的联合主键，甚至可以用一个用户自定义的类，该类拥有这些类型的属性
  - 也可以不指定任何标识属性，而是在持久化注解中直接将多个普通属性映射成一个联合主键，通常不推荐这么做
  - **通常建议使用基本类型的包装类型作为标识属性的类型**
- 为持久化类每个成员变量提供 setter 和 getter 方法
  - Hibernate 默认采用属性方式来访问持久化类的成员变量
  - Hibernate 持久化 JavaBeans 风格的属性，认可如下形式(以 foo 属性为例)的方法名：getFoo()、isFoo()、setFoo()
- 使用非 final 的类
  - 在运行时生成代理是 Hibernate 的一个重要功能
  - 如果持久化类没有实现任何接口，Hibernate 使用 javassist 生成代理，该代理对象是持久化类的子类的实例
  - 如果使用了 final 类，则无法生成 javassist 代理，将无法进行性能优化
  - 还有一个策略，让 Hibernate 持久化类实现一个所有方法都声明为 public 的接口，此时将使用 JDK 的动态代理
  - 同时应该避免在非 final 类中声明 public final 方法，如果非要使用一个有 public fianl 方法的类，则必须通过设置 lazy="false"来明确禁用代理
- 重写 equals()和 hashCode()方法
  - 如果需要把持久化类的实例放入 Set 中(当需要进行关联映射时，推荐这么做)，则应该为该持久化类重写 equals()和 hashCode()方法
  - 实现 equals()和 hashCode()方法最显而易见的方法是比较两个对象标识属性的值，如果值相同，则两个对象对应于数据库的同一行，因此它们是相等的

- 采用自动生成标识值的对象不能使用这种方法
- Hibernate 仅为那些持久化对象指定标识值，一个新创建的实例将不会有任何标识值

5.5.2. 持久化对象的状态

1、Hibernate 持久化对象支持如下几种对象状态

- 瞬态：
  - **对象由 new 操作符创建，且尚未与 Hibernate Session 关联的对象被认为处于瞬态**
  - 瞬态对象不会被持久化到数据库中，也不会被赋予持久化标志
  - 如果程序中失去了瞬态对象的引用，瞬态对象将被垃圾回收机制销毁
  - 使用 Hibernate 可以将其变为持久化状态
- 持久化：
  - 持久化实例在数据库中有对应的记录，并拥有一个持久化标志(identifier)
  - 持久化的实例可以是刚刚保存的，也可以是刚刚被加载的，无论哪种持久化对象都必须与指定 Hibernate Session 关联
  - Hibernate 会检测到处于持久化状态对象的改动，在当前操作执行完成时将对象数据写回数据库
  - 开发者不需要手动执行 update
- 脱管：
  - 某个实例曾经处于持久化状态，但随着与之关联的 Session 被关闭，该对象就变成脱管状态
  - 脱管对象的引用依然有效，对象可继续被修改
  - 如果重新让脱管对象与某个 Session 关联，这个脱管对象会重新转换为持久化对象，而脱管期间的改动不会丢失，也可被写入数据库
  - 正因为这个功能，逻辑上的长事务成为可能，它被称为应用程序事务即事务可以跨越用户的思考，因为当对象处于脱管状态时，对该对象的操作无须锁定数据库，不会造成性能下降

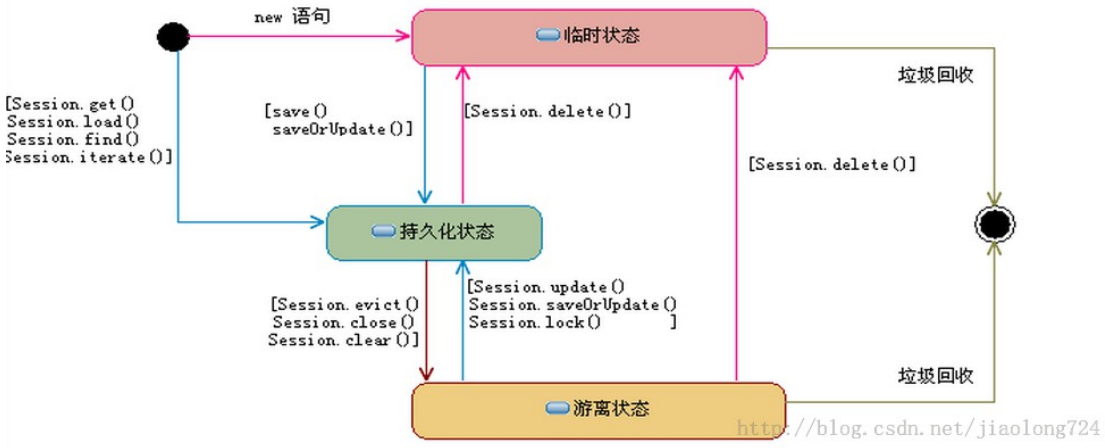

图 53 Hibernate 持久化对象的状态演化图

5.5.3. 改变持久化对象状态的方法

1、通过 new 新建一个持久化实例时，该实例处于瞬态

5.5.3.1. 持久化实体

1、为了让瞬态对象转换为持久化状态，Hibernate Session 提供了如下几个方法

- Serializable save(Object obj)：将 obj 对象转为持久化状态，该对象的属性将被保存到数据库
- void persist(Object obj)：将 obj 对象转化为持久化状态，该对象的属性将被保存到数据库
- Serializable save(Object,Object pk)：将 obj 对象保存到数据库，保存到数据库时，指定主键值
- void persist(Object obj,Object pk)：将 obj 对象转化为持久化状态，保存到数据库时，指定主键值

2、把一个瞬态实体变成持久化状态时，Hibernate 会在底层对应地生成一条 return 语句，这条语句负责把该实体对应的数据记录插入数据表

3、Hibernate 提供与 save()功能几乎完全类似的 persist()方法

- 一方面为了照顾 JPA 的用法习惯
- 另一方面
  - 使用 save()方法保存持久化对象时，该方法返回该持久化对象的标识属性值(即对应记录的主键值)
  - 使用 persist()方法保存持久化对象时，该方法没有任何返回值
  - 因为 save()方法需要立即返回持久化对象的标识属性值，所以程序执行 save()方法会立即将持久化对象对应的数据插入数据库
  - 而 persist()则保证当它在一个事务外部被调用时，并不立即转换成 insert 语句，这个功能很有用，尤其是需要封装一个长会话流程的时候，persist()方法就显得尤为重要

5.5.3.2. 根据主键加载持久化实体

1、程序可以通过 load()来加载一个持久化实例，这种加载就是根据持久化类的标识属性值加载持久化实例---其实质就是根据主键从数据表中加载一条新纪录

```
News n=sess.load(News.class,pk);
```

- pk 就是需要加载的持久化实例的标识属性
- 如果没有匹配的数据库记录，load()方法可能抛出 HibernateException 异常
- **如果在持久化注解中指定了延迟加载，则 load()方法会返回一个未初始化的代理对象，这个代理对象并没有加载数据记录，直到程序调用该代理对象的某方法时，Hibernate 才会去访问数据库(为什么用代理：如果只是返回一个空的引用，那么在需要加载的时候无法通过该引用来进行加载，因为是空，如果是代理对象的引用，那么这个代理对象是存在的，只是其包含的被代理对象的引用是空，那么当我需要使用该对象时，可以利用代理对象的某个方法去读取数据库来获得该对象的实体)**

2、与 load()方法类似的是 get()方法，get()方法也用于根据主键加载持久化实体，但 get()方法会立刻访问数据库，如果没有对应记录的话，get()方法返回 null，而不是返回一个代理对象

3、当程序通过 load()或 get()方法加载实体时，Hibernate 会在底层对应地生成一

条 select 语句，这条 select 语句带有"where<主键列>=<标识属性值>"子句，表明将会根据主键加载

4、load()方法和 get()方法的主要区别在于是否延迟加载

- 使用 load()方法将具有延迟加载功能，load()方法不会立即访问数据库，当试图加载的记录不存在时
- load()方法可能返回一个未初始化的代理对象，而 get()方法总是立即访问数据库，当试图加载的记录不存在时，get()方法直接返回 null

5.5.4. 更新持久化实体

1、一旦加载了该持久化实例后，该实体就处于持久化状态，在代码中对持久化实例所作的修改被保存到数据库

2、程序对持久化实例所作的修改会在 Session flush 之前被自动保存到数据库，无需程序调用其他方法(不需要调用 update()方法)来将修改持久化，也就是说，修改对象最简单的方法就是在 Session 处于打开状态时 load 它，然后直接修改即可

3、如果调用持久化实体的 setter 方法改变属性值，Hibernate 会在 Session flush 之前生成一条 update 语句，这条 update 语句带有"where<主键列>=<标识属性值>"子句，表明将会根据主键来修改特定记录

```
News n=sess.load(News.class,pk);
n.setTitle("新标题");
sess.flush();
```

4、以上代码片段，表面上看有极大的性能缺陷：程序需要修改某条记录时，这种做法将产生两条 SQL 语句，一条用于查询指定记录的 select 语句，另一条用于修改该记录的 update 语句。**对于一个 Java EE 应用而言，Hibernate 通常的处理流程是，从数据库里加载记录->将信息发送到表现层供用户修改->将所作修改重新保存到数据库，在这种处理流程下，应用本身就需要两条 SQL 语句**

5.5.5. 更新脱管实体

1、对于一个曾经持久化过、但现在已脱离了 Session 管理的持久化对象，它被认为处于脱管状态，当程序修改脱管对象的状态后，程序应该显式使用新的 Session 来保存这些修改，Hibernate 提供了 update()、merge()、updateOrSave()等方法来保存这些修改

```
News n=firstSess.load(News.class,pk);
firstSess.close();
n.setTitle("新标题");
Session secondSess=...
secondSess.update(n);
```

2、使用另一个 Session 保存这种修改后，该脱管对象再次回到 Session 的管理之下，也就再次回到持久化状态

3、当需要使用 update()来保存程序对持久化对象所作的修改时，如果不清楚该对象是否曾经持久化过(数据库中有无记录)，那么程序可以选择使用 updateOrSave()方法，该方法自动判断该对象是否曾经持久化过，如果曾经持久化过，就执行 update()操作，否则执行 save()操作

4、merge()方法可以将程序对脱管对象所作的修改保存到数据库

- 但 merge()与 update()方法的最大区别是
  - merge()方法不会持久化给定的对象
  - sess.update(a)后，a 变为持久化状态
  - sess.merge(a)后，a 依然不是持久化状态，merge()方法会返回 a 的副本---该副本处于持久化状态
- 当程序使用 merge()方法来保存程序对脱管对象所作的修改时，如果 Session 中存在相同持久化标识的持久化对象，merge()方法里提供的对象状态将会覆盖原有持久化实例的状态，如果 Session 中没有响应的持久化实例，则尝试从数据库中加载，或者创建新的持久化实例，最后返回该持久化实例

5.5.5.1. 删除持久化实体

1、通过 Session 的 delete()方法来删除该持久化实例，一旦删除了该持久化实例，该持久化实例对应的数据记录也将被删除

```
News n=sess.load(News.class,pk);
sess.delete(n);
```

2、上述做法也会产生两条 SQL 语句，但在实际 Java EE 应用中也没有任何问题，因为 Java EE 应用总需要先选出一条记录，将其输出到表现层，等用户确认删除这条记录时，系统才会真正删除这条记录

3、**Hibernate 本身不提供直接执行 update 或 delete 语句的 API，Hibernate 提供的是一种面向对象的状态管理**

4、如果需要执行 DML 风格的 update 或 delete 类似语句，建议使用 Hibernate 的批处理功能

5.6. 深入 Hibernate 映射

1、正是@Entity、@Table、@Id、@GeneratedValue 等注解将 POJO 变成 PO 类

2、Hibernate 提供了如下三种方式将 POJO 变成 PO 类

- 使用持久化注解(以 JPA 标准注解为主，如果有一些特殊要求，则依然需要使用 Hibernate 本身提供的注解)
- 使用 JPA2 提供的 XML 配置描述文件(XML deployment descriptor)，这种方式可以让 Hibernate 的 PO 类与 JPA 实体类兼容，但在实际开发中，很少有公司使用这种方式
- 使用 Hibernate 传统的 XML 映射文件(*.hbm.xml 文件的形式)，由于这种方式是传统的 Hibernate 的推荐方式，因此依然有少数企业会采用这种方式

3、对于 Hibernate PO 类而言，通常可以采用如下两个注解来修饰它

- @Entity：被该注解修饰的 POJO 就是一个实体，使用该注解时可指定一个 name 属性，name 属性指定该实体类的名称，但大部分时候无须指定该属性，因为系统默认以该类的类名作为实体类的名称
- @Table：该注解指定持久化类锁映射的表

属性	是否必须	说明
catalog	否	用于设置将持久化类映射的表放入指定的 catalog 中，如果没有指定该属性，数据表将

		放入默认的 catalog 中
indexes	否	为持久化类所映射的表设置索引。该属性的值是一个@Index 注解数组
name	否	设置持久化类所映射的表的表明，如果没有指定该属性，那么该表的表明将于持久化类的类名相同
schema	否	设置将持久化类所映射的表放入指定的 schema 中，如果没有指定该属性，数据表将放入默认的 schema 中
uniqueConstraints	否	为持久化类所映射的表设置唯一约束。该属性的值是一个@UniqueConstraint 注解数组

- @UniqueConstraint：用于为数据表定义唯一约束，它的用法非常简单，使用该注解时可以指定如下唯一的属性
  - columnNames：该属性的值是一个字符串数组，每个字符串元素代表一个数据列
- Index：用于为数据表定义索引，可以指定如下属性

属性	是否必须	说明
columnList	是	设置对哪些列建立索引，该属性的值可指定多个数据列的列名
name	否	设置该索引的名字
unique	否	设置该索引是否具有唯一性。该属性只能是 boolean 值

  - 如果希望改变 Hibernate 的属性访问策略，则可使用标准的@Access 注解修改该持久化类，该注解的 value 属性支持 AccessType.PROPERTY 和 AccessType.FIELD 两个值
    - AccessType.PROPERTY：默认属性，即使用 getter/setter 方法访问属性(比如需要访问 abc 属性，则应该提供 setAbc()和 getAbc()两个方法)
    - AccessType.FIELD：Hibernate 将直接通过成员变量来访问属性，一般不建议这么做
    -
- @Proxy：该注解的 proxyClass 属性指定一个接口，在延迟加载时作为代理使用，也可以在这里指定该类自己的名字
- @DynamicInsert：指定用于插入记录的 insert 语句是否在运行时动态生成，并且只插入那些非空字段。该属性的值默认是 false。开启该属性将导致 Hibernate 需要更多来生成 SQL 语句
- @DynamicUpdate：指定用于更新记录的 update 语句是否在运行时动态生成，并且只更新那些改变过的字段，该属性的值默认是 false。开启该属性将导致 Hibernate 需要更多时间来生成 SQL 语句
  - 当程序打开了@DynamicUpdate 之后，持久化注解可以指定如下几种乐观锁定的策略
    - OptimisticLockType.VERSION：检查 version/timestamp 字段
    - OptimisticLockType.ALL：检查全部字段

        - OptimisticLockType.DIRTY：只检查修改过的字段
        - OptimisticLockType.NONE：不使用乐观锁定
    - 强烈建议在 Hibernate 中使用 version/timestamp 字段来进行乐观锁定，对性能来说，这是最好的选择，并且也是唯一能够处理 Session 外进行脱管操作的策略
- @SelectBeforeUpdate：指定 Hibernate 在更新(update)某个持久化对象之前是否需要先进行一次查询(select)
    - 如果该值设为 true，则 Hibernate 可以保证只有当持久化对象的状态被修改过时，才会使用 update 语句来保存其状态(即使程序显式使用 saveOrUpdate()来保存该对象，但如果 Hibernate 查询到对应记录与该持久化对象的状态相同，也不会使用 update 语句来保存其状态
    - 该注解的 value 默认为 false
    - 通常来说，将@SelectBeforeUpdate 设为 true 会降低性能，如果应用程序中某个持久化对象的状态经常会发生改变，那么该属性应该设置为 false。如果该持久化对象的状态很少发生改变，而程序又经常要保存该对象时，则可将该属性设置为 true
- @PolymorphismType：当采用了 TABLE_PER_CLASS 继承映射策略时，该注解用于指定是否需要采用隐式多态查询。该注解的 value 的默认值为 PolymorphismType.IMPLICIT，即支持隐式多态查询
    - @PolymorphismType 的 value 设为 PolymorphismType.IMPLICIT 时，如果查询时给出的是任何超类、该类实现的接口或该类的名字，那么都会返回该类(及其子类)的实例；如果查询中给出的是子类的名字，则只返回子类的实例；否则只有在查询时明确给出某个类名时，才会返回这个类的实例
    - 大部分时候需要使用隐式多态查询
- @Where：该注解的 clause 属性可指定一个附加的 SQL 语句过滤条件(类似于添加 where 子句)，如果一旦指定了该注解，则不管采用 load()、get()还是其他查询方法，只要视图加载该持久化类的对象，该 where 条件就会生效
- @BatchSize：当 Hibernate 抓取集合属性或延迟加载的实体时，该注解的 size 属性指定每批抓取的实例数
- @OptimisticLocking：该注解的 type 属性指定乐观锁定策略。Hibernate 支持 OptimisticLocking.ALL、OptimisticLocking.DIRTY、OptimisticLocking.NONE、OptimisticLocking.VERSION 这 4 个枚举值，默认值为 OptimisticLocking.VERSION
- @Check：该注解可通过 constraints 指定一个 SQL 表达式，用于为该持久化类所对应的表指定一个 Check 约束
- @Subselect：该注解用于映射不可变的、只读实体
    - 通俗地说，就是将数据库的子查询英社称 Hibernate 持久化对象
    - 当需要使用视图(实质就是查询)来代替数据表时，该注解比较有用

5.6.1. 映射属性

1、在默认情况下，被@Entity 修饰的持久化类的所有属性都会被映射到底层数据表，为了指定某个属性所映射的数据列的详细信息，如列名、列字段长度等，可以在实体类中使用@Column 修饰该属性

2、使用@Column 时指定的常用属性如下表所示

属性	是否必须	说明
columnDefinition	否	该属性的值是一个代表列定义的 SQL 字符串，指定创建该数据列的 SQL 语句
insertable	否	指定该列是否包含在 Hibernate 生成的 insert 语句的列表中，默认 true
length	否	指定该列所能保存的数据的最大长度，默认 255
name	否	指定该列的列名，该列的列名默认与@Column 修饰的成员变量名相同
nullable	否	指定该列是否允许为 null，默认 true
percision	否	当该列的 decimal 类型时，该属性指定该列支持的最大有效数字位
scale	否	当该列是 decimal 类型时，该属性指定该列最大支持的小数位数
table	否	指定该列所属的表明，当需要用多个表来保存一个实体时，通常要指定该属性
unique	否	指定该列是否具有唯一约束，默认 false
updateble	否	指定该列是否包含在 Hibernate 生成的 update 语句的列表中，默认 true

3、Hibernate 同样允许使用@Access 修饰该属性，用于单独改变 Hibernate 对该属性的访问策略，该@Access 用于覆盖在持久化类上指定的@Access 注解

4、除此之外还允许使用如下特殊注解

- @Formula：该注解的 value 属性可指定一个 SQL 表达式，指定该属性的值将根据表达式来计算。持久化类对应的表中没有和计算属性对应的数据列---因为该属性值是动态计算出来的，无需保存到数据库
  - value="(sql)"的英文括号
  - 能少
  - value="()"的括号里面是 SQL 表达式，SQL 表达式中的列名与表名应该和数据库对应，而不是和持久化对象的属性对应
  - 如果需要在@Formula 的 value 属性中使用参数，则直接使用 where cur.id=currencyID 形式，其中 currencyID 就是参数，当前持久化对象的 currencyID 属性将作为参数传入
- @Generated：设置该属性映射的数据列的值是否由数据库生成，该注解的 value 属性可以接受如下三个值之一
  - GenerationTime.NEVER：不由数据库生成
  - GenerationTime.INSERT：该属性值在执行 insert 语句时生成，但不会在执行 update 语句时重新生成
  - GenerationTime.ALWAYS：该属性值在执行 insert 和 update 语句时都会

被重新生成

5.6.1.1. 使用@Transient 修饰不想持久保存的属性

1、在默认情况下，持久化类的所有属性都会自动映射到数据表的数据列，如果在实际应用中不想持久保存某些属性，则可以考虑使用@Transient 来修饰它们

5.6.1.2. 使用@Enumerated 修饰枚举类型的属性

1、在有些极端的情况下，持久化类的属性不是普通的 Java 类型，而是一个枚举类型，这意味着该属性只能接受有限的几个固定值

2、对于枚举值而言，既可在程序中通过枚举值的名字来代表，也可使用枚举值的需要来代表。同样的，底层数据库既可保存枚举值名称来代表枚举值，也可保存枚举值序号来代表枚举值，这一点可以通过 @Enumerated 的 value 属性来指定

- 当@Enumerated 的 value 属性为 EnumType.STRING 时，底层数据库保存枚举值的名称
- 当@Enumerated 的 value 属性为 EnumType.ORDINAL 时，底层数据库保存枚举值的序号

5.6.1.3. 使用@Lob、@Basic 修饰大数据类型的属性

1、对于使用数据库保存图片、保存大段文章，数据库通常需要采用 Blob、Clob 类型的数据列来保存它们，而 JDBC 则会采用 java.sql.Blob、java.sql.Clob 来表示这些大数据类型的值

2、Hibernate 也为这种大数据类型的值提供了支持，Hibernate 使用@Lob 来修饰这种大数据类型，当持久化类的属性为 byte[]、Byte[]或 java.io.Serializable 类型时，@Lob 修饰的属性将映射为底层的 Blob 列；当持久化类的属性为 char[]、Character[]或 java.lang.String 类型时，@Lob 修饰的属性将映射为底层的 Clob 列

3、对于使用@Lob 修饰的大数据类型，底层数据库往往采用 Blob 或 Clob 类型的列来保存这种大数据类型的值，数据库加载这种大数据类型的值也是需要较大开销的

4、当需要访问非大数据属性的其他属性时，如果加载整个实体，那么会产生非常大不必要的开销，为了改变这种情况，**Hibernate 加载对象时，并不立即加载它的大数据类型的属性，而是只加载一个"虚拟"的代理，等待程序真正需要 pic 属性的时候，才从底层数据表中加载数据，这就是典型的代理模式。Hibernate 为这种机制提供了支持，并将这种机制称为延时加载，只要在开发实体时使用 @Basic 修饰该属性即可**

5、@Basic 可以指定如下属性

- fetch：指定是否延迟加载该属性，该属性可接受 FetchType.EAGER、FetchType.LAZY 两个值之一，其中前者指定立即加载；后者指定使用延迟加载
- optional：指定该属性映射的数据列是否允许使用 null 值

5.6.1.4. 使用@Temporal 修饰日期类型的属性

1、对于 Java 程序而言，表示日期、时间的类型只有两种：java.util.Date 和 java.util.Calendar；对于数据库而言，表示日期、时间的类型有很多，例如 date、time、datetime、timestamp 等

2、在这样的背景下，如果持久化类定义了一个 java.util.Date 类型的属性时，Hibernate 到底将这种类型的属性映射车 date 类型的列、time 类型的列还是 timestamp 类型的列呢

3、于是，可以使用@Temporal 来修饰这种类型的属性，该注解可以指定一个 value 属性

- TemporalType.DATA：映射到 date
- TemporalType.TIME：映射到 time
- TemporalType.TIMESTAMP：映射到 timestamp

5.6.2. 映射主键

1、通常情况下，Hibernate 建议为持久化类定义一个标识属性，用于唯一地标识某个持久化实例，而标识属性则需要映射到底层数据表的主键

2、**所有现代的数据库建模理论都推荐不要使用具有实际意义的物理主键，而是推荐使用没有任何实际意义的逻辑主键，尽量避免使用复杂的物理主键，应考虑为数据库增加一列，作为逻辑主键**

- 从表面上看，增加逻辑主键增加了数据冗余，但如果从外键关联的角度看，使用逻辑主键的主从表关联中，从表只需要增加一个外键列
- 如果使用多列作为联合主键，则需要从表中增加多个外键列，如果有多个从表需要增加外键列，则数据冗余更大
- 使用武力主键还会增加数据库维护的复杂度，主从表之间的约束关系隐晦难懂，难于维护

3、逻辑主键没有实际意义，仅仅用来标识一行记录。Hibernate 为这种逻辑主键提供了主键生成器，他负责为每个持久化实例生成唯一的逻辑主键值

- 如果实体类的标识属性(映射成主键列)是基本数据类型、基本类型的包装类、String、Date 等类型，可以简单地使用@Id 修饰该实体属性即可，使用@Id 注解时无需指定任何属性
- 如果希望 Hibernate 为逻辑主键自动生成主键值，则还应该使用@GeneratedValue 来修饰实体的标识属性，使用@GeneratedValue 时可指定如下属性

属性	是否必须	说明
strategy	否	指定 Hibernate 对该主键列使用怎样的主键生成策略，该属性支持以下 4 个属性值 ➢ GenerationType.AUTO：Hibernate 自动选择最合适底层数据库的主键生成策略，这是默认值 ➢ GenerationType.IDENTITY：对于 MySQL、SQL Server 这样的数据库，选择自增长的主键生成策略 ➢ GenerationType.SEQUENCE：对于 Oracle 这样

		的数据库，选择使用基于 Sequence 的主键生成策略。应与@SequenceGenerator 一起使用 ➢ GenerationType.TABLE：使用辅助表来生成主键，应与@TableGenerator 一起使用
generator	否	当使用 GenerationType.SEQUENCE、GenerationType.TABLE 主键生成策略时，该属性引用@SequenceGenerator、@TableGenerator 所定义的生成器的名称，即 name 属性

4、@SequenceGenerator 定义主键生成器

- 使用@SequenceGenerator 定义主键生成器还会在底层数据库中额外生成一个 Sequence，因此必须底层数据库本身能支持 Sequence 机制(如 Oracle 数据库)
- @SequenceGenerator 支持以下属性

属性	是否必须	说明
name	是	该属性指定该主键生成器的名称
allocationSize	否	该属性指定底层 Sequence 每次生成主键值的个数。对于 Oracle 而言，该属性指定的整数值，将作为定义 Sequence 时的 increment by
catalog	否	该属性指定将底层 Sequence 放入指定 catalog 中，如果不指定该属性，该 Sequence 将放入默认的 catalog 中
schema	否	该属性指定将底层 Sequence 放入指定 schema 中，如果不指定该属性，该 Sequence 将放入默认的 schema 中
initialValue	否	该属性指定底层 Sequence 的初始值，对于 Oracle 而言，该属性指定的整数值将作为定义 Sequence 时 start with 的值
sequenceName	否	该属性指定底层 Sequence 的名称

5、@TableGenerator 定义主键生成器

- 使用@TableGenerator 定义主键生成器会在底层数据库中额外生成一个辅助表
- @TableGenerator 可以指定如下属性

属性	是否必须	说明
name	是	该属性指定该主键生成器的名称
allocationSize	否	该属性指定底层辅助表每次生成主键值的个数
catalog	否	该属性指定将辅助表放入指定 catalog 中，如果不指定该属性，该辅助表将放入默认的 catalog 中
schema	否	该属性将底层辅助表放入指定 schema 中，如果不指定该属性，该辅助表将放入默认的 schema 中

table	否	指定辅助表的表明
initialValue	否	该属性指定的整数值将作为辅助表的初始值，默认 0
pkColumnName	否	该属性指定存放主键名的列
pkColumnValue	否	该属性指定属性名
valueColumnNam e	否	该属性指定存放主键值的列名
indexes	否	该属性是一个@Index 数组，用于为辅助表定义索引
uniqueConstraints	否	该属性值是一个@UniqueConstraint 数组，用于为辅助表创建唯一约束

5.6.3. 使用 Hibernate 的主键生成策略

1、JPA 标准注解只支持 AUTO、IDENTITY、SEQUENCE 和 TABLE 这 4 种主键生成策略，但实际上 hibernate 支持更多的主键生成策略，如果希望使用 Hibernate 提供的主键生成策略，就需要使用 Hibernate 本身的@GenericGenerator 注解，该注解用于定义主键生成器

2、@GenericGenerator 注解支持以下两个属性

- name：必须属性，设置主键生成器的名称，该名称可以被@GeneratedValue 的 generator 属性引用
- strategy：必须属性，设置该主键生成器的主键生成策略
  - increment：为 long、short、int 类型主键生成唯一标识符，只有在没有其他进程往同一个表中插入数据时才能使用
  - identity：在 DB2、MySQL、Microsoft SQL Server、Sybase 和 HypersonicSQL 提供 identity(自增长)主键支持的数据表中适用，返回的标识属性值是 long、short 或 int 类型的
  - sequence：在 DB2、PostgreSQL、Oracle、SAP DB、McKoi 等提供 Sequence 支持的数据表中适用，返回的标识属性值是 long、short 或 int 类型的
  - hilo：使用一个高/低位算法高效的生成 long、short、int 类型的标识符。给定一个表和字段(默认分别是 hibernate_unique_key 和 next_hi)作为高位值的来源。高/低位算法生成的标志属性值只在一个特定的数据库中唯一
  - seqhilo：使用一个高/低位算法来高效地生成 long、short、int 类型的标识符，需要给定一个数据库 Sequence
  - 。该算法与 hilo 稍有不同，它将逐渐历史状态保存在 Sequence 中，适用于支持 Sequence 的数据库，如 Oracle
  - uuid：用一个 128 位的 UUID 算法生成字符串类型的标识符，这在一个网络中是唯一的(IP 地址也作为算法的数据源)。UUID 被编码为一个 32 位十六进制数的字符串(**UUID 算法会根据 IP 地址，JVM 的启动时间(精确到 1/4 秒)、系统时间和一个计数器值(在 JVM 中唯一)来生成一个 32 位的字符串，因此通常 UUID 生成的字符串在一个网络中是唯一的**)
  - guid：在 Microsoft SQL Server 和 MySQL 中使用数据库生成的 GUID 字符

串

- native：根据底层数据库的能力选择 identity、sequence 或 hilo 中的一个
- assigned：让应用程序在 save()之前为对象分配一个标识符，这相当于不指定主键生成策略所采用的默认策略
- select：通常数据库触发器选择某一个主键的行，并返回其主键值作为标识属性值
- foreign：表明直接使用另一个关联的对象的标识属性值(即本持久化对象不能生成主键)

5.6.4. 映射集合属性

1、集合属性大致有两种：一种是单纯的集合属性，例如 List、Set 或数组等集合属性；另一种是 Map 结构的集合属性，每个属性值都有对应的 key 映射

2、Hibernate 要求持久化集合值字段必须声明为接口，实际的接口可以是 java.util.Set、java.util.Collection、java.util.List、java.util.Map、java.util.SortedSet、java.utilSortedMap 等，甚至可以使自定义类型，只要实现 org.hibernate.usertype.UserCollectionType 接口即可

3、Hibernate 之所以要求用集合接口来声明集合属性，是因为当程序持久化某个实例时，Hibernate 会自动把程序中的集合实现类替换成 Hibernate 自己的集合实现类，因此不要把 Hibernate 集合属性强制类型转换为集合实现类，如 HashSet、HashMap 等，但是转换为 Set、Map 等集合，因为 Hibernate 自己的集合类也实现了 Map、Set 等接口

4、集合类实例具有值类型的行为

- 当持久化对象被保存时，这些集合属性会被自动持久化
- 当持久化对象被删除时，这些集合属性对应的记录将被自动删除
- 假设集合元素被从一个持久化对象传递到另一个持久化对象，该集合元素对应的记录会从一个表转移到另一个表

5、**两个持久化对象不能共享同一个集合元素的引用**

6、无论哪种类型的集合属性，都统一使用@ElementCollection 注解进行映射，该注解可以指定如下属性

属性	是否必须	说明
fetch	否	指定该实体对集合属性的抓取策略(当程序初始化该实体时，是否立即从数据库抓取该实体的集合属性中的所有元素) 该属性支持 FetchType.EAGER(立即抓取)和 FetType.LAZY(延迟抓取)两个属性值，默认为 FetType.LAZY
targetClass	否	该属性指定集合属性中集合元素的类型

7、由于集合属性总需要保存到另一个数据表中，所以保存集合属性的数据表必须包含一个外键列，用于参照到主键列，该外键列使用@JoinColumn 进行映射

8、Hibernate 使用标准的@CollectionTable 注解映射保存集合属性的表，使用该注解可指定如下属性

属性	是否必须	说明

name	否	指定保存集合属性的数据表名
catalog	否	指定将保存集合属性的数据表放入指定 catelog 中，如果没有指定该属性，则将保存集合属性的数据表放入默认的 catelog 中
schema	否	指定将保存集合属性的数据表放入指定 schema 中，如果没有指定该属性，则将保存集合属性的数据表放入默认的 schema 中
indexes	否	为持久化类所映射的表设置索引，该属性值是一个@Index 注解数组
joinColumns	否	该属性值为 @JoinColumn 数组，每个 @JoinColumn 映射一个外键列(通常只需要一个外键列即可，但如果主实体采用了复合主键保存集合属性的表就需要定义多个外键列)
uniqueConstraints	否	为持久化类所映射的表设置唯一约束。该属性值是一个@UniqueConstraint 注解数组

9、@JoinColumn 注解专门用于定义外键列，可指定以下属性

属性

是否必须

说明

columnDefinition

否

指定 Hibernate 使用该属性值指定的 SQL 片段来创建外键列

name

否

指定该外键列的列名

insertable

否

指定该列是否包含在 Hibernate 生成的 insert 语句的列表中，默认 true

updatable

否

指定该列是否包含在 Hibernate 生成的 update 语句的列表中，默认 true

nullable

否

指定该列是否允许为 null，默认 true

table

否

指定该列所在的数据表表

unique

否

指定是否为该列增加唯一约束

referencedColumnName

否

指定该列所参照的主键列的列名

10、当集合元素是基本数据类型、字符串类型、日期类型或其他符合类型时，因为这些集合元素都是从属于持久化对象的，而且这些数据类型在数据表只需要一列就可以保存，因此使用@Column 注解定义集合元素列即可

11、在 Java 的所有集合类型(包括数组、Map)中，只有 Set 集合是无序的，即没有显式的索引值

- List、数组使用整数作为集合元素的索引值，而 Map 则使用 key 作为集合元素的索引
- 如果要映射带索引的集合(List、数组、Map)，就需要为集合元素所在的数据表指定一个索引列---用于保存数组索引、List 的索引、或者 Map 集合的 key 索引

12、用于映射索引列的注解有如下两个

- @OrderColumn：用于定义 List 集合、数组的索引列
- @MapKeyColumn：用于映射 Map 集合的索引列
- 这两个注解支持的属性与@Column 大致相似，详见映射属性

13、如果程序需要显式指定 Map key 的类型，则可以使用@MapKeyClass 注解，该注解只有一个 value 属性，该属性用于指定 Map key 的类型

14、Hibernate 集合元素的数据类型几乎可以是任意数据类型，包括基本类型，字符串类型，日期类型，自定义类型，复合类型以及对其他持久化对象的引用

- 如果集合元素是基本类型、字符串类型、日期类型、自定义类型、复合类型等，则位于集合中的对象可能根据"值"语义来操作(其生命周期完全依赖于集合持有者，必须通过集合持有者来访问这些元素)
- 如果集合元素是其他持久化对象的引用，此时就变成了关系映射，这些集合元素都具有自己的生命周期

15、综上，集合元素的类型大致可分为如下几种情况

- 集合元素是基本类型及其包装类、字符串类型和日期类型：此时使用@ElementCollection 映射集合属性，并使用普通的@Column 映射集合元素对应的列
- 集合元素是组件(非持久化实体的复合类型)：此时使用@ElementCollection 映射集合属性，然后使用@Embeddable 修饰非持久化实体的复合类
- 集合元素是关联的持久化实体：此时已经不再是集合属性了，应该使用@OneToMany 或@ManyToMany 进行关联映射

5.6.4.1. List 集合属性

1、List 是有序集合，因此持久化到数据库时也必须增加一列来表示集合元素的次序

2、示例详见 P415，该例子用了如下四个注解

1) @ElementCollection：用于映射集合属性
2) @CollectionTable：用于映射集合属性表，其中 name 属性指定集合属性的表名，joinColumns 属性用于映射外键列
3) @Column：用于映射保存集合元素的数据列(保存集合属性的表)
4) @OrderColumn：用于映射 List 集合的索引列

5.6.4.2. 数组属性

1、Hibernate 对数组和 List 的处理方式非常相似，实际上，List 和数组也非常像，尤其是 JDK 1.5 增加自动装箱、自动拆箱特性之后，它们用法的区别只是 List 的长度可变化，而数组的长度不可变化而已

2、示例详见 P416

5.6.4.3. Set 集合属性

1、Set 集合属性的映射与 List 有点不同，但因为 Set 是无序、不可重复的集合，因此 Set 集合属性无需使用@OrderColumn 注解映射集合元素的索引列

2、与映射 List 集合相同的是，映射 Set 集合同样需要使用@ElementCollection 映射集合属性，使用@CollectionTable 映射保存集合属性的表，如果集合元素是基本类型及其包装类、String、Date 等类型，也可使用@Column 映射保存集合元素的数据列

3、声明 Set 集合属性时，只能使用 Set 接口，不能使用实现类

4、示例详见 P417

5.6.4.4. Map 集合属性

1、Map 集合属性同样需要使用@ElementCollection 映射集合属性，使用@CollectionTable 映射保存集合属性的数据表，如果 Map 的 value 是基本类型及其包装类型、String 或 Date 类型，同样也可使用@Column 映射保存 Map value 的数据列

2、除此之外，需要使用@MapKeyColumn 映射保存 Map key 的数据列

3、Hibernate 将以外键列和 key 列作为联合主键

4、与所有集合属性类似的是，集合属性的声明只能使用接口，但程序依然需要显式初始化该集合属性

5、虽然可以通过使用泛型指定 Map 集合的 key、value 类型，然后让 Hibernate 通过反射去获取类型，但是通过显式注明(@ElementCollection 的 targetClass 属性和@MapKeyClass 注解)类型，可以避免 Hibernate 通过反射获取，从而提升程序性能

6、示例详见 P418

5.6.5. 集合属性的性能分析

1、当系统从数据库中初始化某个持久化类时，集合属性是否随持久化类一起初始化呢

- 如果集合属性里包含十万甚至百万的记录，在初始化持久化实体时，立即抓取所有的集合属性，将导致性能急剧下降
- 完全有可能系统只需要使用持久化类集合属性中的部分记录，而不是集合属性的全部，这样就没有不要一次加载所有的集合属性

2、**对于集合属性，通常推荐使用延迟加载策略：所谓延迟加载就是等系统需要使用集合属性时才从数据库加载关联数据**

3、Hibernate 对集合属性默认采用延迟加载，在某些特殊情况下，为@ElementCollection 注解设置 fetch=FetchType.EAGER 来取消延迟加载

4、集合可分为如下两类

- 有序集合：集合里的元素可以根据 key 和 index 访问
  - 有序集合都拥有一个由外键列和集合元素索引列组成的联合主键，在这种情况下，集合属性的更新时非常搞笑的---主键已经被有效索引，因此当 Hibernate 试图更新或删除一行时，可以迅速找到该行数据
  - 显然，有序集合的属性在增加、删除、修改中拥有较好的性能表现
  - 需要指出的是，虽然数组也是有序集合，但数组无法使用延迟加载(因为数组长度不可变)，所以实际上用数组作为集合的性能并不高
- 无需集合：集合里的元素只能遍历
  - 无需集合，例如 Set 的主键由外键列和其他元素字段构成，或者根本没有主键
  - 如果集合中元素是组合元素或者大文本、大二进制字段，数据库可能无法有效地对复杂的主键进行索引，即使可以建立索引，性能也非常差
  - 由于 Set 集合内部结构的原因，如果改变 Set 集合的某个元素，Hibernate 并不会立即更新(update)该元素对应的数据行，因此对于 Set 集合而言，只有在执行插入(insert)和删除(delete)操作时改变才有效

5、在 Hibernate 中，Set 应该是最通用的集合类型，这是因为 Set 的语义最贴近关系模型的关联关系，因此 Hibernate 的关联映射都是采用 Set 集合进行管理的，而且在设计良好的 Hibernate 领域模型中，1-N 关联中的 1 的一端通常不再控制关联关系，所有的更新操作将会在 N 的一端进行处理，对于这种情况，无需考虑其集合的更新性能(这里的意思是 Set 更好咯???那与下面矛盾了啊)

6、一旦指定了 1 的一端不再控制关联关系，则表明该集合映射作为反向集合使用，在这情况下

- 使用 List 集合属性将有较好的性能，因为可以在未初始化集合元素的情况下直接向 List 集合中添加新元素，因为 Collection.add() 和 Collection.addAll()以及 List.add()和 List.addAll()总是返回 true(这尼玛与上一条不是矛盾了???)
- Set 集合不同，Set 集合需要保证集合元素不能重复，因此当程序视图向 Set 集合中添加元素时，Set 集合需要先加载所有的集合元素，再依次比较添加的新元素是否重复，最后才可以决定是否能成功添加新元素，所以向 Set 集合中添加元素时性能较低

7、当程序视图删除集合的全部元素时，Hibernate 是比较智能的

- 例如调用 List 集合的 clear()方法删除全部集合元素，Hibernate 不会逐个删除集合元素，而是使用一条 delete 语句
- 但如果不是删除所有元素，那么 Hibernate 不会那么智能，会逐个删除指定的所有元素

8、**集合属性表里的记录完全"从属"于主表的实体**

- 当主表的记录被删除时，集合属性表里"从属"于该记录的数据将会被删除
- Hibernate 无法直接加载、查询集合属性表中的记录，只能先加载主表实体，再通过主表实体去获取集合属性对应的记录

5.6.6. 有序集合映射

1、Hibernate 还支持使用 SortedSet 和 SortedMap 两个有序集合

- 当映射这种有序集合时，只有使用 Hibernate 本身提供的@SortNatural 或 @SortComparator 注解，其中前者表明对集合元素采用自然排序，后者表明对集合元素采用定制排序
- 因此使用@SortComparator 时必须指定 value 属性，该属性为 Comparator 实现类

2、Hibernate 的有序集合的行为与 java.util.TreeSet 或 java.util.TreeMap 的行为非常相似

3、如果希望数据库查询自己对集合元素排序，则可以利用 Hibernate 自己提供的@OrderBy 注解，该注解只能在 JDK 1.4 或更高版本(因为底层需要利用 LinkedHashSet 或 LinkedHashMap 来实现)中有效，它会在 SQL 查询中完成排序，而不是在内存中完成排序

5.6.7. 映射数据库对象

1、如果希望在映射文件中创建和删除触发器、存储过程等数据库对象，Hibernate 提供了<database-object.../>元素来满足这种需求

2、借助于 Hibernate 的 Schema 交互工具，就可以让 Hibernate 映射文件拥有完全定义用户 Schema 的能力

- 映射数据库对象这种功能目前无法用注解来实现，只能通过 Hibernate 传统的*.hbm.xml 映射文件的方式来实现

3、使用<database-object.../>元素只有如下两种形式

- 第一种形式是在映射文件中显式声明 create 和 drop 命令

```
<hibernate-mapping>
    ...
    <database-object>
        <create> create trigger t_full_content_gen...</create>
        <drop> create trigger t_full_content_gen</drop>
    </database-object>
<hibernate-mapping>
```

  - <create.../>元素里的内容就是一个完整的 DDL 语句，用于创建一个触发器
  - <drop.../>元素里定义了删除指定数据库对象的 DDL
  - 每个<database-object>元素中只有一组<create.../>、<drop.../>对
- 第二种形式是提供一个类，这个类知道如何组织 create 和 drop 命令，这个特别类必须实现 org.hibernate.mapping.AuxiliaryDatebaseObject 接口

```
<hibernate-mapping>
...
    <database-object>
        <definition class="MyTriggerDefinition"/>
    </database-object>
<hibernate-mapping>
```

4、如果向指定某些数据库对象仅在特定的方言中才可使用，还可在<database-object.../>元素里使用<dialect-scope.../>子元素来进行配置

5、<未完成>：不懂啥意思

5.7. 映射组件属性

1、**组件属性**：持久化类的属性并不是基本数据类型，也不是字符串、日期等标量类型的变量，而是一个符合类型的对象，在持久化过程中，它仅仅被当做值类型，而并非引用另一个持久化实体

2、组件属性的类型可以是任何自定义类

- 对于一个复合类型，Hibernate 肯定无法使用单独的数据列来保存该类型的对象，因此不能直接使用@Column 注解来映射 name 属性
- 为了让 Hibernate 知道 Name 将会作为组件类型使用，可使用@Embeddable 注解，该注解与@Entity 类似，都不需要指定任何属性，只是@Entity 修饰的类将作为持久化类使用，而@Embeddable 修饰的类将作为持久化类的组件使用
- 对于@Embeddable 修饰的类，同样可使用@Column 来指定这些属性对应的数据列
- 对于只包含基本类型、字符串、日期类型等属性的组件，Hibernate 将会把每个属性映射成一个数据列---即@Embeddable 修饰的类中每个属性映射成一个数据列

3、另外，Hibernate 为组件映射提供了另外一种映射策略，这种映射策略无须在组件类上使用@Embeddable 注解，而是直接在持久化类中使用@Embedded 注解修饰组件属性

- 如果需要为组件属性所包含的紫属性指定列名，则可使用@AttributeOverrides 和 @AttributeOverride 注解，其中每个@AttributeOverride 指定一个属性的映射配置
- 示例详见 P426

5.7.1. 组件属性为集合

1、如果组件类又包括了 List、Set、Map 等集合属性，则可以直接在组件类中使用@ElementCollection 修饰集合属性，并使用@CollectionTable 指定保存集合属性的数据表---**与普通实体类中映射集合属性的方式基本相同**

5.7.2. 集合属性的元素为组件

1、集合除了可以存放基本类型、字符串、日期类型之外，还可以存放组件对象(也就是复合类型)，实际上，在更多情况下，集合里存放的都是组件对象

2、对于集合元素是组件的集合属性，程序依然使用@ElementCollection 修饰集合属性，使用@CollectionTable 映射保存集合属性的表

- 对于带索引的集合，如果是 List 集合，则使用@OrderColumn 映射索引列；若是 Map 则使用@MapKeyColumn 映射索引列
- 不同的是，程序不再使用@Column 映射保存集合元素(组件类型)的数据列---Hibernate 无法使用单独的数据列保存集合元素(组件类型)，程序只要使用@Embeddable 修饰组件类即可

3、示例详见 P428

5.7.3. 组件作为 Map 的索引

1、由于 Map 集合的特殊性，它允许使用复合类型的对象作为 map 的 key，所以 Hibernate 也对这种组件作为 Map key 的情形提供支持

- Hibernate 依然使用@ElementCollection 修饰集合属性，使用@CollectionTable 映射保存集合属性的表
- 对带索引的集合，如果是 List 则使用@OrderColumn 映射索引列，如果是 Map 集合，则使用@MapKeyColumn 映射索引列
- 不同的是，由于 Map key 是组件类型，建议使用@MapKeyClass 注解指定 Map key 的类型

5.7.4. 组件作为复合主键

1、如果数据库采用简单的逻辑主键，则不会出现组件类型的主键，但在一些特殊情况下，总会出现组件类型的主键，Hibernate 也为这种组件类型的主键提供了支持

2、**使用组件作为复合主键，也就是使用组件作为持久化类的标识符，该组件必须满足以下要求**

- 有无参数的构造器
- 必须实现 java.io.Serializable 接口
- 建议正确地重写 equals()和 hashCode()方法，也就是根据组件类的关键属性来区分组件对象

3、当使用组件作为复合主键时，Hibernate 无法为这种复合主键自动生成主键值，所以程序必须为持久化实例分配这种组件标识符

4、当持久化类使用组件作为复合主键时，程序需要使用@EmbeddedId 来修饰该主键

- @EmbeddedId 和@Embedded 的用法基本类似，只是@Embedded 用于修饰普通属性的组件属性，而@EmbeddedId 用于修饰组件类型的主键
- 使用@EmbeddedId 同样可以结合@AttributeOverrides 和 @AttributeOverride 两个注解

5.7.5. 多列作为联合主键

1、Hibernate 还提供了另一种联合主键支持，Hibernate 允许直接将持久化类的多个属性映射成联合主键。如果需要直接将持久化类的多列映射成联合主键，则该持久化类必须满足如下条件

- 有无参构造器
- 实现 java.io.Serializable 接口
- 建议根据联合主键列锁映射的属性来重写 equals()和 hashCode()方法

2、将持久化类的多个属性映射为联合主键非常简单，直接使用多个@Id 修饰这些属性即可

5.8. 使用传统的映射文件

1、在 JDK 1.5 出现以前，Hibernate 已经出现，并开始了广泛的流传，因此传统的 Hibernate 并不使用注解管理映射关系，而是使用 XML 映射文件来管理映射关系

2、由来由于 Java 注解的盛行，很多原来采用 XML 配置文件进行管理的信息，现在都开始改为使用注解进行管理，比如 Struts2 以及 Sprint 等
3、无论是 XML 配置文件还是注解，本质都是一样的，只是信息的载体不同而已。Sun 公司后来推出了 JPA 规范(本质是 ORM 规范)，JAP 规范推荐使用注解来管理实体的映射关系，因此注解在 Hibernate 中使用得越来越广泛

5.8.1. 增加 XML 映射文件

1、当采用 XML 映射文件来管理实体类的映射关系后，可以得到如下公式

PO=POJO+映射文件

2、如果使用传统方式来开发 Hibernate 持久化类，每个持久化类需要由两部分组成：Java 类与 XML 映射文件
3、示例详见 P435

5.8.2. 注解还是 XML 映射文件

1、当开发一个以 Hibernate 作为持久层的 Java EE 应用时，开发者既可以选择注解来管理映射关系，也可以选择 XML 映射文件来管理映射关系
2、对于一个新项目，推荐使用注解，理由如下

- 使用注解更加简洁，开发者可以将实体类的 Java 代码、注解集中在一个文件中管理，因此更加简单
- 基于注解的实体具有更好的可保值性，因为这些注解并不属于 Hibernate，而是属于 JPA 规范，因此这些实体类不仅对于 Hibernate 可用，对于 JPA 也是可用的，因此如果有一天打算把应用迁移到其他 ORM 框架上，底层实体类无须做任何改变

Chapter 6. 深入使用 Hibernate

1、通过 Hibernate，应用程序可以从底层的 JDBC 中释放出来，以面向对象的方式进行数据库访问

2、除此之外，Hibernate 可以理解面向对象的继承、多态等概念，一旦建立了正确的继承映射，程序就能以面向对象的方式进行数据库访问

6.1. Hibernate 的关联映射

1、**实例之间的互相访问就是关联关系**

2、关联关系式面向对象分析、面向对象设计最重要的只是，Hibernate 完全可以理解这种关联关系，如果映射得当，Hibernate 的关联映射将可以大大简化持久层数据的访问

3、关联关系大致有如下两个分类

- 单向关系：只需单向访问关联端
  - 1->1
  - 1->N
  - N->1
  - N->N
- 双向关系：关联的两端可以相互访问
  - 1--1
  - 1--N 双向关系里没有 N--1，因为双向关系 1--N 与 N--1 是完全相同的
  - N--N

4、映射策略

- 基于外键的映射策略
- 基于连接表的映射策略

6.1.1. 单项 N-1 关联

1、N--1 是非常常见的关联关系，最常见的父子关系也是 N--1 关联，单向的 N--1 关联只需从 N 的一段端以访问 1 的一端

2、为了让两个持久化类支持这种关联映射，程序应该在 N 的一端的持久化类中增加一个属性，该属性引用 1 的一端的关联实体

3、对于 N--1 关联，无论是单向关联还是双向关联，都需要在 N 的一端使用 @ManyToOne 修饰代表关联实体的属性，该注解可指定的属性如下

属性	是否必须	说明
cascade	否	指定 Hibernate 对关联实体采用怎样的级联策略，该级联策略支持如下 5 个属性值 ➢ CascadeType.ALL：指定 Hibernate 将所有的持久化操作都级联到关联实体 ➢ CascadeType.MERGE：指定 Hibernate 将 merge 操作级联到关联实体 ➢ CascadeType.PERSIST：指定 Hibernate 将 persist 操作级联到关联实体 ➢ CascadeType.REFRESH：指定 Hibernate 将 refresh 操作级联到关联实体

		➢ CascadeType.REMOVE：指定 Hibernate 将 remove 操作级联到关联实体
fetch	否	指定抓取关联实体时的抓取策略 ➢ FetchType.EAGER：抓取实体时，立即抓取关联实体，这是默认值 ➢ FetchType.LAZY：抓取实体时，延迟抓取关联实体，等到真正用到关联实体时才去抓取
optional	否	该属性指定关联关系是否可选
targetEntity	否	该属性指定关联实体类名，默认情况下 Hibernate 将通过反射来判断关联实体的类名

6.1.1.1. 无连接表的 N—1 关联

1、对于无连接表的 N--1 关联而言，程序只要在 N 的一端增加一列外键，让外键值记录对象所属的实体即可。Hibernate 可使用@JoinColumn 来修饰代表关联实体的属性，@JoinColumn 用于映射底层的外键列

2、在所有既有的基于外键约束的关联关系中，都必须牢记：要么总是先持久化主表记录对应的实体，要么设置级联操作；否则当 Hibernate 试图插入从表记录时，如果发现该从表记录参照的主表记录不存在，那一定会抛出异常

6.1.1.2. 有连接表的 N—1 关联

1、对于绝大部分单向 N--1 关联而言，使用基于外键的关联映射已经足够了。但由于底层数据库建模时也可以使用连接表来建立这种关联关系，因此 Hibernate 也未这种关联关系提供了支持

2、如果需要使用连接表来映射单向 N--1 关联，程序需要显式使用@JoinTable 注解来映射连接表，该注解可以指定如下属性

属性
是否必须
说明
name
否
指定该连接表的表

catalog
否
设置该连接表放入指定的 catalog 中。如果没有指定该属性，连接表将放入默认的 catalog 中
schema
否
设置该连接表放入指定的 schema 中。如果没有指定该属性，连接表将放入默认的 schema 中
targetEntity
否
该属性指定关联实体的类名。在默认的情况下，Hibernate 将通过反射来判断关联实体的类名

indexes
否
该属性值为@Index 注解数组，用于为该连接表定义多个索引
joinColumns
否
该属性值可接受多个@JoinColumn，用于配置连接表中外键列的列信息，这些外键列**参照(<未完成>：什么意思)**当前实体对应表的主键列
inverseJoinColumns
否
该属性值可接受多个@JoinColumn，用于配置连接表中外键列的列信息，这些外键列**参照(<未完成>：什么意思)**当前实体的关联实体对应表的主键列
uniqueConstraints
否
该属性用于连接表增加唯一约束

3、**对于使用连接表的 N--1 关联而言，由于两个实体对应的数据表都无须增加外键列，因此两个实体对应的数据表不存在主从关系。因此无论先持久化哪个实体，程序都不会引发性能问题(<未完成>：什么意思)**

- <连接表>：例如 Person 与 Address 要产生关联，但是在这两个表中都不存放对应的列(即 Person 没有存 Address 的 id 的列，Address 也没有存 Personid 的列)，它们之间的关系是通过另外一张表建立的。即一个连接表 A，A 中有两列，一列是 person_id，一列是 address_id，通过这种形式建立的连接

6.1.2. 单向 1—1 关联

1、对于单向的 1--1 关联关系，需要在持久化类里增加代表关联实体的成员变量，并为该成员变量增加 setter 和 getter 方法

2、对于 1--1 关联(不管是单向关联还是双向关联)，都需要使用@OneToOne 修饰代表关联实体的属性，该注解可以指定如下属性

属性	是否必须	说明
cascade	否	指定 Hibernate 对关联实体采用怎样的级联策略，该级联策略支持如下 5 个属性值 ➢ CascadeType.ALL：指定 Hibernate 将所有的持久化操作都级联到关联实体 ➢ CascadeType.MERGE：指定 Hibernate 将 merge 操作级联到关联实体 ➢ CascadeType.PERSIST：指定 Hibernate 将 persist 操作级联到关联实体 ➢ CascadeType.REFRESH：指定 Hibernate 将 refresh 操作级联到关联实体 ➢ CascadeType.REMOVE：指定 Hibernate 将 remove 操作级联到关联实体
fetch	否	指定抓取关联实体时的抓取策略 ➢ FetchType.EAGER：抓取实体时，立即抓取

		关联实体，这是默认值 FetchType.LAZY：抓取实体时，延迟抓取关联实体，等到真正用到关联实体时才去抓取
mappedBy	否	该属性合法的属性值为关联实体的属性名，该属性指定关联实体中哪个属性可引用到当前实体
orphanRemoval	否	该属性设置是否删除"孤儿"实体，如果某个实体所关联的父实体不存在(即该实体对应记录的外键为 null)，该实体就是所谓的"孤儿"实体
optional	否	该属性指定关联关系是否可选
targetEntity	否	该属性指定关联实体的类名。在默认情况下，Hibernate 将通过反射来判断关联实体的类名

6.1.2.1. 基于外键的单向 1—1 关联

1、对于基于外键的 1--1 关联而言，只要先用@OneToOne 注解修饰代表关联实体的属性，在使用@JoinColumn 映射外键列即可---由于是 1--1 关联，因此该为@JoinColumn 增加 unique=true

6.1.2.2. 有连接表的单向 1—1 关联

1、虽然这种情况很少见，但是 Hibernate 同样允许这样采用连接表映射单向 1--1 关联。有连接表的 1--1 关联同样需要显式使用@JoinTable 映射连接表---由于是 1--1 关联，因此该为@JoinColumn 增加 unique=true

6.1.3. 单向 1—N 关联

1、单向 1---N 关联的持久类发生了改变，持久化类里需要使用集合属性，因为 1 的一端要访问 N 的一端，而 N 的一端将以集合(Set)形式表现

2、对于单项的 1--N 关联关系，只需要在 1 的一端增加 Set 类型的成员变量，该成员变量记录当前实体所有的关联实体，当然还要为这个 Set 类型的属性增加 setter 和 getter 方法

3、为了映射 1--N 关联，hibernate 需要使用@OneToMany 注解，使用@OneToMany 则可指定如下属性

属性	是否必须	说明
cascade	否	指定 Hibernate 对关联实体采用怎样的级联策略，该级联策略支持如下 5 个属性值 ➢ CascadeType.ALL：指定 Hibernate 将所有的持久化操作都级联到关联实体 ➢ CascadeType.MERGE：指定 Hibernate 将 merge 操作级联到关联实体 ➢ CascadeType.PERSIST：指定 Hibernate 将 persist 操作级联到关联实体 ➢ CascadeType.REFRESH：指定 Hibernate 将 refresh 操作级联到关联实体 ➢ CascadeType.REMOVE：指定 Hibernate 将 remove 操作级联到关联实体

fetch	否	指定抓取关联实体时的抓取策略 ➢ FetchType.EAGER：抓取实体时，立即抓取关联实体，这是默认值 FetchType.LAZY：抓取实体时，延迟抓取关联实体，等到真正用到关联实体时才去抓取
orphanRemoval	否	该属性设置是否删除"孤儿"实体，如果某个实体所关联的父实体不存在(即该实体对应记录的外键为 null)，该实体就是所谓的"孤儿"实体
mappedBy	否	该属性合法的属性值为关联实体的属性名，该属性指定关联实体中哪个属性可引用到当前实体
targetEntity	否	该属性指定关联实体的类名。在默认情况下，Hibernate 将通过反射来判断关联实体的类名

6.1.3.1. 无连接表的单向 1—N 关联

1、对于无连接表的 1--N 单向关联而言，同样需要在 N 的一端增加外键列来维护关联关系，但程序此时只让 1 的一端来控制关联关系，因此直接在 1 的一端使用@JoinColumn 修饰 Set 集合属性、映射外键列即可

2、对于双向的 1--N 父子关联，使用 1 的一端控制关系的性能，比使用 N 的一端控制关系的性能低

- 性能低的原因是当使用 1 的一端控制关联关系时，由于插入数据时无法同时插入外键列(N 的一端在插入时并不知道它所关联的实体，因为是 1-->N)，因此会额外多出一条 update 语句
- 而且还无法增加非空约束，因为添加关联实体时，总会先插入一条外键列为 null 的记录

3、尽量避免使用 1--N 的关联模型，即使一定要用，也要使用有连接表的 1--N 关联

6.1.3.2. 有连接表的单向 1—N 关联

1、对于有连接表的 1--N 关联，同样需要使用@OneToMany 修饰代表关联实体的集合属性，还应该使用@JoinTable 显式指定连接表

2、对于采用连接表的单向 1--N 关联而言，由于采用了连接表来维护 1--N 关联关系，两个实体对应的数据表都无须增加外键列，因此不存在主从表关系，程序完全可以想先持久化哪个实体，就先持久化哪个实体，无论先持久化哪个实体，程序都不会引发性能问题

6.1.4. 单向 N—N 关联

1、单向的 N--N 关联和 1--N 关联的持久化类代码完全相同，控制关系的一端需要增加一个 Set 类型的属性，被关联的持久化实例以集合形式存在

2、N--N 关联需要使用@ManyToMany 注解来修饰代表关联实体的集合属性，使用@ManyToMany 时可指定如下属性

属性	是否必须	说明
cascade	否	指定 Hibernate 对关联实体采用怎样的级联策略，该级联策略支持如下 5 个属性值

		➢ CascadeType.ALL：指定 Hibernate 将所有的持久化操作都级联到关联实体 ➢ CascadeType.MERGE：指定 Hibernate 将 merge 操作级联到关联实体 ➢ CascadeType.PERSIST：指定 Hibernate 将 persist 操作级联到关联实体 ➢ CascadeType.REFRESH：指定 Hibernate 将 refresh 操作级联到关联实体 CascadeType.REMOVE：指定 Hibernate 将 remove 操作级联到关联实体
fetch	否	指定抓取关联实体时的抓取策略 ➢ FetchType.EAGER：抓取实体时，立即抓取关联实体，这是默认值 FetchType.LAZY：抓取实体时，延迟抓取关联实体，等到真正用到关联实体时才去抓取
mappedBy	否	该属性合法的属性值为关联实体的属性名，该属性指定关联实体中哪个属性可引用到当前实体
targetEntity	否	该属性指定关联实体的类名。在默认情况下，Hibernate 将通过反射来判断关联实体的类名

3、N--N 关联必须使用连接表，N--N 关联与有连接表的 1--N 关联非常相似，因此都需要使用@JoinTable 来映射连接表，区别是 N--N 关联要去掉@JoinTable 注解的 inverseJoinColumns 属性所指定的@JoinColumn 中的 unique=true

6.1.5. 双向 1—N 关联

1、对于 1--N 关联，Hibernate 推荐使用双向关联，而且不要让 1 的一端控制关联关系，而使用 N 的一端控制关联关系

2、双向的 1--N 关联与 N--1 关联是完全相同的两种情形，两端都需要增加对关联属性的访问，N 的一端增加引用到关联实体的属性，1 的一端增加集合属性，集合元素为关联实体

3、Hibernate 同样对这种双向关联映射提供了两种支持：有连接表和无连接表，大部分时候，对于 1--N 的双向关联映射，使用无连接表的映射策略即可

6.1.5.1. 无连接表的双向 1—N 关联

1、无连接表的双向 1--N 关联，N 的一端需要增加@ManyToOne 注解来修饰代表关联实体的属性，而 1 的一端需要使用@OneToMany 注解来修饰代表关联实体的属性

2、底层数据库为了记录这种 1--N 关联关系，实际上只需要在 N 的一端的数据表里增加一个外键列即可，因此应该在使用@ManyToOne 注解的同时，使用@JoinColumn 来映射外键列

3、对于双向的 1--N 关联映射，通常不应该允许 1 的一端控制关联关系，而应该由 N 的一端来控制关联关系，因此应该在使用@OneToMany 注解时指定 mappedBy 属性---一旦为@OneToMany、@ManyToMany 指定了该属性，则表明当前实体不能控制关联关系，当@OnetToMany、@ManyToMany、@OneToOne

所在的当前实体放弃控制关联关系之后，Hibernate 就不允许使用@JoinColumn 或@JoinTable 修饰代表关联实体的属性了

6.1.5.2. 有连接表的双向 1—N 关联

1、对于有连接表的双向 1--N 关联而言，1 的一端无需任何改变，只要在 N 的一端显式使用@JoinTable 来指定连接表即可

2、如果希望 1 的一端也可以控制关联关系，那么只需要删除@OneToMany 注解的 mappedBy 属性，并同时配合使用@JoinTable 注解

6.1.6. 双向 N—N 关联

1、双向 N--N 关联需要两端都使用 Set 集合属性，两端都增加对集合属性的访问

2、**双向 N--N 只能用连接表来建立两个实体之间的关联关系**

3、双向 N--N 关联需要在两端分别使用@ManyToMany 修饰 Set 属性，并在两端都使用@JoinTable 显式指定映射连接表。**在两端映射连接表时，两端指定的连接表的表**
应该相同，而且两端使用@JoinTable 时指定的外键列的列名也是相互对应的

4、如果程序希望某一端放弃控制关联关系，则可在这一端的@ManyToMany 注解中指定 mappedBy 属性，这一端就无须、也不能使用@JoinTable 映射连接表

6.1.7. 双向 1—1 关联

1、双向 1--1 关联需要让两个持久化类都增加引用关联实体的属性，并为该属性提供 setter 和 getter 方法

6.1.7.1. 基于外键的双向 1—1 关联

1、对于双向 1--1 关联而言，两端都需使用@OneToOne 注解进行映射

2、对于基于外键的双向 1--1 关联，外键可以存放在任意一端，存放外键的一端，需要增加@JoinColumn 注解来映射外键列，还应该为@JoinColumn 注解增加 unique=true 属性来表示该实体实际上是 1 的一端

3、对于 1--1 的关联关系，两个实体原本处于平等状态，当选择任意一端来增加外键后，该表即变为从表，而另一个表则成为主表

4、对于双向 1--1 关联的主表对应的实体，也不应该用于控制关联关系(否则会导致生成额外的 update 语句，从而引起性能下降)，因此主表对应的实体中使用@OneToOne 注解时，应增加 mappedBy 属性---该属性用于表明该实体不管理关联关系，这一端对应的数据表将作为主表使用，不能使用@JoinColumn 映射外键列

6.1.7.2. 有连接表的双向 1—1 关联

1、采用连接表的双向 1--1 关联是相当罕见的，映射相当复杂，数据模型繁琐，通常不推荐使用这种

2、有连接表的双向 1--1 关联需要在两端分别使用@OneToOne 修饰代表关联实体的属性，并在两端都使用@JoinTable 显式映射连接表。在映射连接表时，**两端指定的连接表的表**
应该相同，而且两端使用@JoinTable 时指定的外键列的列名也是相互对应的

6.1.8. 组件属性包含的关联实体

1、组件里的属性不仅可以是基本类型、字符串、日期类型等，也可以是值类型行为的组件，甚至可以是关联实体

2、对于组件的属性是关联实体的情形，可以使用@OneToOne、@OneToMany、@ManyToOne、@ManyToMany 修饰代表关联实体的属性

- 如果采用基于外键的映射策略，还需要配合@JoinColumn 注解---该注解用于映射外键列
- 如果采用基于连接表的映射策略，还需要配合 @JoinTable 注解---该注解用于映射连接表

3、由于组件不映射为持久化类(1--N，其中 1 端是组件，N 端是持久化类)

- 在组件中使用@OneToMay，在 N 端的持久化类中所映射的数据表增加外键列
- 有点混乱，详见 P457 的例子，以及 P458 的解释

6.1.9. 基于复合主键的关联关系

1、实际项目中并不推荐使用复合主键，总是建议采用没有物理意义的逻辑主键，复合主键的做法不仅会增加数据库建模的难度，而且会增加关联关系的维护成本

2、在某些特殊情形下，或者由于某些人的特殊习惯，总有可能需要面对基于复合主键的关联，Hibernate 也为这种特殊的关联提供了支持

6.1.10. 复合主键的成员属性为关联实体

1、复合主键并没有带来什么特别的好处，却给编程、数据库维护带来额外的麻烦

2、本节使用的复合主键更特殊，符合主键的成员是关联实体

- 复合主键的成员是关联实体看上去比较特殊，但在实际项目中却很受欢迎
- 例如开发一个进销存管理系统，该系统涉及订单、商品、订单项三个实体，其中一个订单可以包含多个订单项，一个订单用于订购某个商品，以及订购数量，一个商品可以多次出现在不同的订单中
- 订单和订单项之间存在双向的 1--N 关联关系，订单项和商品之间存在单向的 N--1 关联关系，这就是普通的双向关联，单向关联
- 问题是：实际项目中，有些成员不为订单项定义额外的逻辑主键，而是使用订单主键、商品主键、订货数量作为复合主键，这就比较特殊，需要做一些特殊的映射
- 实例详见 P461-462

6.1.11. 持久化的传播性

1、当程序中有两个关联实体时，程序需要主动保存、删除或重关联每个持久化实体；如果需要处理多处彼此关联的实体，则需要依次保存每个实体，这很繁琐

2、从数据库建模的角度来看，两个表之间 1--N 关联关系总是用外键约束来表

示，其中保留外键的数据表称为从表，被从表参照的数据表称为主表。对于这种主从表约束关系，Hibernate 则有两种映射策略

- 将从表记录映射成持久化类的组件，这就是上一章介绍的集合属性的集合元素是组件
- 将从表记录也映射成持久化实体，这就是此处介绍的关联关系

3、如果将从表记录映射成持久化类的组件，这些组件的生命周期总是依赖于父对象，Hibernate 会默认启用级联操作，不需要额外的动作。当父对象被保存时，这些组件子对象也被保存，父对象被删除，子对象也被删除

4、如果将从表记录映射成持久化实体，则从表实体也有了自己的生命周期，从而应该允许其他实体共享对它的引用。例如从集合中移除一个实体，不意味着它可以被删除，所以 Hibernate 默认不启用实体到其他关联实体之间的级联操作

5、对于关联实体而言，Hibernate 默认不会启用级联操作，当父对象被保存时，它关联的子实体不会被保存；父对象被删除时，它关联的子实体不会被删除。

6、为了启用不同持久化操作的级联行为，Hibernate 定义了如下级联风格

- CascadeType.ALL：指定 Hibernate 将所有的持久化操作都级联到关联实
- CascadeType.MERGE：指定 Hibernate 将 merge 操作级联到关联实体
- CascadeType.PERSIST：指定 Hibernate 将 persist 操作级联到关联实体
- CascadeType.REFRESH：指定 Hibernate 将 refresh 操作级联到关联实体
- CascadeType.REMOVE：指定 Hibernate 将 remove 操作级联到关联实体
- 如果程序希望某个操作能被级联传播到关联实体，则可在配置 @OneToMany、@OneToOne、@ManyToMany 时通过 cascade 属性来指定

7、Hibernate 还支持一个特殊的级联策略：删除"孤儿"记录(可通过 @OneToMany、@OneToOne 的 orphanRemoval 属性来启动该级联策略)，**该级联策略只对当前实体时 1 的一端，且底层数据表为主表时有效**

- 对于启用了 orphanRemoval 策略的级联操作而言，当程序通过主表实体切断与从表实体的关联关系时---虽然此时主表实体对应的记录并没有删除，但由于从表实体失去了对主表实体的引用，因此这些从表实体就变成了"孤儿"记录，Hibernate 会自动删除这些记录

8、对于级联策略的设定，Hibernate 有如下建议

- 在 @ManyToOne 中指定级联没有意义，级联通常在 @OneToOne 和 @OneToMany 关系中比较有用---**因为级联操作应该是由主表记录传播到从表记录，通常从从表记录则不应该传播到主表记录，因此 @ManyToOne 不支持 cascade 属性**
- 如果从表记录被完全限制在主表记录之内(当主表记录被删除后，从表记录没有存在的意义)，则可以指定 cascade=Cascade.ALL，再配合 orphanRemoval=true 级联策略，将从表实体的生命周期完全交给主表实体管理
- 如果经常在某个事务中同时使用主表实体和从表实体，则可以考虑指定 cascade={CascadeType.PERSIST,CascadeType.MERGE}级联策略

9、对于 cascade=CascadeType.ALL 级联策略详细解释如下

- 如果主表实体被 persist()，那么关联的从表实体也会被 persist()
- 如果主表实体被 merge()，那么关联的从表实体也会被 merge()
- 如果主表实体被 save()、update()或 saveOrUpdate()，那么所有的从表实

体则会被 saveOrUpdate()

- 如果把持久化状态下的主表实体和瞬态或脱管的从表实体建立关联，则从表实体将被自动持久化
- 如果主表实体被删除，那么关联的从表实体也会被删除
- 如果没有把主表实体删除，只是切断主表实体和从表实体之间的关联关系，则关联的从表实体不会被删除，只是将关联的从表实体的外键列设置为 null

10、如果指定了 orphanRemoval=true 策略，则只要一个从表实体失去了关联的主表实体，不管该主表实体是被删除，还是切断了主表实体和它的关联，那么该从表实体就变成了 orphan(孤儿)，Hibernate 将自动删除该从表实体

11、所有操作都是在调用期(call time)将持久化操作级联到关联实体上，save-update 和 orphanRemoval 操作是在 Session flush 时(写入期)才级联到关联对象上的

6. 2. 继承映射

1、对于面向对象的程序设计语言，继承、多态是两个最基本的概念。Hibernate 的继承映射可以理解为两个持久化类之间的继承关系。例如老师和人之间的关系，老师继承了人，可以认为老师是一个特殊的人，如果对人进行查询，老师实例也将被得到---**而无需关注人的实例、老师的实例底层数据库的存储**

2、Hibernate 支持多种继承映射策略，不管哪种继承映射策略，Hibernate 的多态查询都可以运行良好

3、对于类与类之间的继承关系，Hibernate 提供了三种映射策略

- 整个类层次对应一个表
- 连接子类的映射策略
- 每个具体类对应一个表

6. 2. 1. 整个类层次对应一个表的映射策略

1、整个类层次对应一个表的映射策略是 Hibernate 继承映射默认的映射策略

2、系统如何分辨一条记录到底属于哪个实体，为该表额外增加一列，使用该列来区分每行记录到底是哪个类的实例，这个列被称为辨别者列

3、在这种映射策略下，需要使用@DiscriminatorColumn 来配置辨别者列，包括制定辨别者列的名称、类型等信息，使用@DiscriminatorColumn 时可以指定如下属性

属性	是否必须	说明
columnDefinition	否	指定 Hibernate 使用该属性值指定的 SQL 片段来创建该辨别者列
name	否	指定辨别者列的名称，该属性的默认值是"DTYPE"
discriminatorType	否	指定该辨别者列的数据类型，支持如下几种类型 ➢ DiscriminatorType.CHAR：字符类型 ➢ DiscriminatorType.INTENGER：整数类型 ➢ DiscriminatorType.STRING：字符串类型

length	否	该属性指定辨别者列的字符长度

4、在 Hibernate 中使用整个类层次对应一个表的映射策略时，使用@DiscriminatorColumn 修饰整棵继承树的根父类。除此之外，使用这种映射策略时还需要使用@DiscriminatorValue 来修饰每个子类，使用该注解时只需指定一个 value 属性，该 value 属性指定不同实体在辨别者列上的值，而 Hibernate 就是根据该辨别者列上的值来区分记录属于哪个实体的(这种想法很自然)

5、映射结果会导致很多列会出现 NULL 值，这正是这种映射策略的劣势---所有子类定义的字段，不能有非空约束，因为如果为这些字段增加非空约束，那么父类的实例在这些列上根本没有值，这肯定引起数据完整性冲突，导致父类的实例无法保存到数据库

6、这种映射策略有一个非常大的好处---在这种映射策略下，整棵继承树的所有数据都保存在一个表内，因此不管进行怎样的查询，不管查询继承树的哪一层实体，底层数据库都只需要在一个表中查询即可，无需进行多表连接查询，也无需进行 union 查询，因此这种映射策略是性能最好的

6.2.2. 连接子类的映射策略

1、这种策略不是 Hibernate 继承映射的默认策略，因此如果需要在继承映射中采用这种映射策略，必须在继承树的根类中使用@Inheritance 指定映射策略

2、使用@Inheritance 时必须指定 strategy 属性，该属性支持如下三个值

- InheritanceType.SINGLE_TABLE：整个类层次对应一个表的映射策略，这是默认值
- InheritanceType.JOINED：连接子类的映射策略
- InheritanceType.TABLE_PER_CLASS：每个具体类对应一个表的映射策略

3、采用这种映射策略时，父类实体保存在父类里，而子类实体则由父类表和子类表共同存储，因为子类实体也是一个特殊的父类实体，因此必然也包含了父类实体的属性，于是将子类与父类共有的属性保存在父类表中，而子类增加的属性则保存在子类表中

4、使用连接子类的继承映射策略，当程序查询子类实例时，需要跨越多个表查询，到底需要跨越多少个表，取决于该子类有多少层父类

5、采用连接子类的映射策略时，无须使用辨别者列，子类增加的属性也可以有非空约束，是一种比较理想的映射策略。在查询子类实体的数据时，可能需要跨越多个表来查询，对于类继承层次较深的继承树来说，查询子类实体时需要在多个子类表之间进行连接操作，可能导致性能低下

6.2.3. 每个具体类对应一个表的映射策略

1、Hibernate 规范还支持每个具体类对应一个表的映射策略，在这种映射策略下，子类增加的属性也可以有非空约束---即父类实例的数据保存在父表中，而子类实例的数据则保存在子表中

2、与连接子类映射策略不同的是，子类实例的数据仅保存在子类表中，没有在父类表中有任何记录，在这种映射策略下，子类表的字段比父类表的字段要多，因为子类表的字段等于父类属性加子类属性的总和

3、在这种映射策略下，如果单从数据库来看，几乎难以看出它们之间存在继承关系，只是多个实体之间的主键值具有某种连续性---因此不能让数据库为各数

据表自动生成主键值，因此采用这种继承策略时，不能使用 GenerationType.IDENTITY、GenerationType.AUTO 两种主键生成策略，而是采用 hilo 策略，该策略保证生成值在某个特定数据库中唯一

4、与连接子类映射策略相似的是，采用这种映射策略时，开发者必须在继承树的根类中使用 @Inheritance 修饰，使用该注解时指定 strategy=InheritanceType.TABLE_PER_CLASS 属性

5、在这种策略下

- 不同的实体对象保存在不同的表中，不会出现加载一个实体需要跨越多个表取数据的情况
- 执行多态查询时，也需要跨越多个数据表进行查询，例如查询满足某个条件的 Person 实例，Hibernate 将会从 person_inf 中查询，也会从 Person 所有子类对应的表中查询数据，然后对这些查询结果进行 union 晕眩

6、采用这种映射策略，底层数据库的数据看起来更符合正常的数据库设计，不同实体的数据保存在不同的数据表中，因此更易理解

6.3. Hibernate 的批量处理

1、Hibernate 完全以面向对象的方式来操作数据库，当程序里以面向对象的方式操作持久化对象时，将被自动转换为对数据库的操作

2、问题是：如果程序需要同时更新 100000 条记录，是不是要逐一加载 100000 条记录，然后依次调用 setter 方法---这样不仅繁琐，数据访问的性能也十分糟糕，面对这种批量处理场景，Hibernate 提供了批量处理的解决方案

6.3.1. 批量插入

1、例如下面的代码

```
Session session=sessionFactory.openSession();
Transaction tx=session.beginTransaction();
for(int i=0;i<100000;i++){
    User u=new User(...);
    session.save(u);
}
tx.commit();
session.close();
```

- 可能随着程序的运行，总会在某个时候失败，并且抛出 OutOfMemoryException 异常(内存溢出异常)
- 这是因为 Hibernate 的 Session 持有一个必选的一级缓存，所有的 User 实例都将在 Session 级别的缓存区进行缓存的缘故

2、为了解决这个问题，有个很简单的思路：定时将 Session 缓存的数据刷入数据库，而不是一直在 Session 级别缓存，可以考虑设计一个累加器，每保存一个 User 实例，累加器增加 1，根据累加器的值决定是否要将 Session 缓存中的数据刷入数据库

3、除了要手动清空 Session 级别的缓存外，最好关闭 SessionFactory 级别的二级缓存，否则即使手动 flush Session 级别的缓存，但因为在 SessionFactory 还有二级缓存，也可能引发异常

6.3.2. 批量更新

1、如果需要返回多行数据，应该使用 scroll 方法，从而可以充分利用服务器端游标所带来的性能优势(<未完成>：啥意思)

6.3.3. DML 风格的批量更新/删除

1、Hibernate 提供的 HQL 语句也支持批量 update 和 delete 语法

2、批量 update 和 delete 语句的语法格式如下

update | delete from? <ClassName> [where where_conditions]

- 在 from 子句中，from 关键字是可选的，即完全可以不写 from 关键字
- 在 from 子句中只能有一个类名，可以在该类名后指定别名
- 不能在批量 HQL 语句中使用连接，显式或隐式的都不行。但可以在 WHERE 子句中使用子查询
- 整个 where 子句是可选的，where 子句的语法和 HQL 语句的 where 子句的语法完全相同

6.4. 使用 HQL 查询

1、Hibernate 提供了异常强大的查询体系，使用 Hibernate 有多种查询方式可以选择---既可以使用 Hibernate 的 HQL 查询，也可以使用条件查询，甚至可以使用原生的 SQL 查询语句。不仅如此，Hibernate 还提供了一种数据过滤功能，这些都用于筛选目标数据

2、HQL 查询时 Hibernate 配备的功能强大的查询语言，这种 HQL 语句被设计为完全面向对象的查询，它可以理解如继承、多态和关联之类的概念。并且，HQL 可以使用绝大部分 SQL 函数、EJB 3.0 操作和函数，并提供了一些 HQL 函数，用以提高 HQL 查询的功能

6.4.1. HQL 查询

1、HQL 是 Hibernate Query Language 的缩写，HQL 的语法很像 SQL 的语法，但 HQL 是一种面向对象的查询语言。SQL 的操作对象是数据表、列等数据库对象，而 HQL 操作的对象是类、实例、属性等

2、HQL 是完全面向对象的查询语言，因此可以支持继承、多态等特性

3、HQL 查询依赖 Query 类，每个 Query 实例对应一个查询对象。使用 HQL 查询按如下步骤进行

- 获取 Hibernate Session 对象
- 编写 HQL 语句
- 以 HQL 语句为参数，调用 Session 的 createQuery()方法创建查询对象
- 如果 HQL 语句包含参数，则调用 Query 的 setXxx()方法为参数赋值
- 调用 Query 对象的 list()或 uniqueResult()方法返回查询结果列表(持久化实体集)

4、Query 对象可以连续多次为 HQL 参数赋值，这得益于 Hibernate Query 的设计，通常 setXxx()方法的返回值都是 void，但 Hibernate Query 的 setXxx()方法的返回值是 Query 本身，因此程序通过 Session 创建 Query 后，直接多次调用 setXxx()方法为 HQL 语句的参数赋值

5、Query 最后调用 list()方法返回查询到的全部结果

6、Query 还包含如下两个方法

- setFirstResult(int firstResult)：设置返回的结果集从第几条记录开始
- setMaxResult(int maxResult)：设置本次查询返回的结果数目
- 上述两个方法用于对 HQL 查询时间分页控制

7、HQL 语句本身是不区分大小写的。也就是说，HQL 语句的关键字、函数都是不区分大小写的(推荐大写)，但 HQL 语句中所使用的包名、类名、实例名、属性名都区分大小写

6.4.2. HQL 查询的 from 子句

1、from 是最简单的 HQL 语句，也是最基本的 HQL 语句。from 关键字后紧跟持久化类的类名，例如

```
from Person
```

- 表明从 Person 持久化类中选出全部的实例
- 大部分的时候，推荐为该 Person 的每个实例起别名，例如

```
from Person as p
```

- 起别名时，as 关键字是可选的，但为了增加可读性，建议保留
- from 后面还可同时出现多个持久化类，此时将产生一个笛卡尔积或跨表的连接，但实际上这种用法很少使用，因为通常需要使用跨表连接的时候，可以考虑使用隐式连接或者显式连接，而不是直接在 from 后跟多个表
-

6.4.3. 关联和连接

1、当程序需要从多个数据表中取得数据时，SQL 语句将会考虑使用多表连接查询。Hibernate 使用关联映射来处理底层数据表之间的连接，一旦提供了正确的关联映射后，当程序通过 Hibernate 进行持久化访问时，将可利用 Hibernate 的关联来进行连接

2、HQL 支持两种关联连接(join)形式：隐式(implicit)与显式(explicit)

- 隐式连接不适用 join 关键字，使用英文点号(.)来隐式连接关联实体，而 Hibernate 底层将自动进行关联查询，例如

```
from Person p where p.myEvent.title > :title
```

  - 上面的 p.myEvent 属性的实质是一个持久化实体，因此 Hibernate 底层隐式地自动进行连接查询
- 显式则需要使用 xxx join 关键字，例如

```
from Person p
inner join p.myEvent event
where event.happenDate < :endDate
```

  - 使用显式连接时可以为相关联的实体，甚至是关联集合中的全部元素指定一个别名

3、Hibernate 可以有如下几种连接方式

- inner join(内连接)，可简写成 join
- left outer join(左外连接)，可简写成 left join
- right outer join(右外链接)，可简写成 right join
- full join(全连接)，并不常用

- 这里的 inner join、left outer join、right outer join、full join 的实质依然是基于底层 SQL 的内、左、右、外连接，所以如果底层 SQL 不支持这些外连接，那么执行对应的 HQL 时就会引发异常

4、使用显式连接时，还可以通过 HQL 的 with 关键字来提供额外的连接条件

```
from Person p
inner join p.myEvent event
with p.id > event.id
where event.happenDate < :endDate
```

- Hibernate 会将这种显式连接转换成 SQL99 多表连接的语法，所有 HQL 语句中的 with 关键字的作用基本等同于 SQL99 中 on 关键字的作用：都用于指定连接条件

5、隐式连接和显式连接还有如下两点区别

- 隐式连接底层将转换成 SQL99 的交叉连接，显式连接底层将转换成 SQL99 的 innter join、left join、right join 等连接
- 隐式连接和显式连接查询后返回的结果不同
  - 当 HQL 语句中省略 select 关键字时，使用隐式连接查询返回的结果是多个被查询实体组成的集合
  - 使用显式连接查询的 HQL 语句中省略 select 关键字时，返回的结果也是集合，但集合元素是被查询的持久化对象、所有被关联的持久化对象所组成的数组

6、在 Hibernate 3.2.3 以后的版本，如果关联实体时单个实体或单个的组件属性，HQL 依然可以使用英文点号(.)来隐式连接关联实体或组件；但如果关联实体是集合(包括 1--N 关联、N--N 关联和集合元素是组件等)，则必须使用 xxx join 来显式连接关联实体或组件

- 对于有集合属性的，Hibernate 默认采用延迟加载策略，即在加载某一个持久化实体时，其关联实体可能并未真正加载，可能只加载了一个代理，那么当 Session 关闭，该实例就无法访问其关联的属性了
- 为了解决这个问题，可以在 Hibernate 持久化注解中指定 fetch=FetchType.EAGER 来关闭延迟加载
- 还有一种方法，使用 join fetch

```
from Person as p
join fetch p.scores
```

  - 上面的 fetch 关键字将导致 Hibernate 在初始化 Person 对象时，同时抓取该 Person 关联的 scores 集合属性(或关联实体)
  - 使用 join fetch 时通常无需指定别名，因为相关联的对象不应当在 where 子句(或其他任何子句)中使用。而且被关联的对象也不会在被查询的结果中直接返回，而是应该通过其父对象来访问

7、使用 fetch 关键字有如下几个注意点

- fetch 不应该与 setMaxResults()或 setFirstResult()共用。因为这些操作是基于结果集的，而在预先抓取集合类时可能包含重复的数据，即无法预先知道精确的行数
- fetch 不能与独立的 with 条件一起使用

- 如果在一次查询中 fetch 多个集合，可以查询返回笛卡尔积，因此请多加注意
- full join fetch 与 right join fetch 是没有任何意义的

6.4.4. HQL 查询的 select 子句

1、select 子句用于选择指定的属性或直接选择某个实体，当然 select 选择的属性必须是 from 后持久化类包含的属性

```
select p.name from Person as p
```

- select 可以选择任意属性，即不仅可以选择持久化类的直接属性，还可以选择组件属性包含的属性

```
select p.name.firstName from Person as p
```

- 在通常情况下，使用 select 子句查询的结果是集合，而集合元素就是 select 后的实例、属性等组成的数组
- 在特殊情况下，如果 select 后只有一项(包括持久化实例或属性)，则查询得到的集合元素就是该持久化实例或属性
- 如果 select 后有多项，则每个集合元素就是选择出的多项组成的数组，例如

```
select p.name, p from Person as p
```

  - 执行该 HQL 语句得到的集合元素是类似于[String,Person]结构的数组，其中第一个元素是 Person 实例的 name 属性，第二个元素是 Person 实例

2、select 支持将选择出的属性存入一个 List 对象中

```
select new list(p.name,p.address) from Person as p
```

3、甚至可以将选择出的属性直接封装成对象

```
select new ClassTest(p.name, p.address) from Person as p
```

- 前提是必须有对应的构造器才行

4、select 还支持给选中的表达式命名别名

```
select p.name as personName form Person as p
select new map(p.name as personName) from Person as p
```

6.4.5. HQL 查询的聚集函数

1、HQL 也支持在选出的属性上使用聚集函数，HQL 支持的聚集函数与 SQL 的完全不同，有如下 5 个

- avg：计算属性平均值
- count：统计选择对象的数量
- max：统计属性值的最大值
- min：统计属性值的最小值
- sum：计算属性值的总和

```
select count(*) from Person
select max(p.age) from Person as p
```

2、select 子句还支持字符串连接符、算数运算符以及 SQL 函数

3、select 子句也支持使用 distinct 和 all 关键字，此时的效果与 SQL 中的效果完全相同

6.4.6. 多态查询

1、HQL 语句被设计成能理解多态查询，from 后跟持久化类名，不仅会查询出该持久化类的全部实例，还会查询出该类的子类的全部实例

```
from Person as p
```

- 该查询不仅会查询出 Person 的全部实例，还会查询出其子类的全部实例

2、HQL 支持在 from 子句中指定任何 Java 类或接口，查询会返回继承了该类的持久化子类的实例或返回实现该接口的持久化类的实例

6.4.7. HQL 查询的 where 子句

1、where 用于筛选选中的结果，缩小选择的范围，如果没有为持久化实例命名别名，则可以直接使用属性名来引用属性

```
from Person where name like 'tom%'
from Person as p where p.name like 'tom%'
```

- 上述两句等价

2、只要没有出现集合属性，HQL 语句可使用点号来隐式连接多个数据表

3、where 子句中的运算符只支持基本类型或字符串，因此 where 子句中的属性表达式必须以基本类型或字符串结尾，不要使用组件类型属性结尾

4、在进行多态持久化的情况下，class 关键字用来存取一个实例的辨别值(discriminator value)，嵌入 where 子句中的 Java 类名，将被作为该类的辨别值

6.4.8. 表达式

1、HQL 的功能非常丰富，where 子句后支持的运算符也非常丰富，不仅包括 SQL 的运算符，也包括 EJB-QL 的运算符等

2、where 子句中允许使用大部分 SQL 支持的表达式

- 数学运算符：+、-、*、/等
- 二进制比较运算符：=、>=、<=、<>、!=、like 等
- 逻辑运算符：and、or、not 等
- in 、 not in 、 between 、 is null 、 is not null 、 is empty 、 is not empty、member of and not member of 等
- 简单的 case ， case...when...then...else...end 和 case ， case when...then...else..end 等
- 字符串连接符：如 value1||value2，或者使用字符串连接函数 concat(value1,value2)
- 时间操作函数：current_date()、current_time()、current_timestamp()、second()、minute()、hour()、day()、month()、year()等
- HQL 还支持 EJB-QL 3.0 所支持的函数或操作：substring()、trim()、lower()、upper()、length()、locate()、abs()、sqrt()、bit_length()、coalesce()和 nullif()等
- 支持数据库的类型转换函数，如 cast(...as...)，第二个参数是 Hibernate 的类型名，或者 extract(...from...)，前提是底层数据库支持 ANSI cast()和 extract()
- 如果底层数据库支持单行函数：sign()、trunc()、rtrim()、sin()

- HQL 语句支持使用命名参数作为占位符，方法是在参数名前加英文冒号(:)，例如(:start_date)等，也支持使用英文问号+数字的形式(?N)的参数作为占位符

3、where 子句还支持如下特殊关键字用法

- HQL index()函数，作用于 join 的有序集合的别名
- HQL 函数，把集合作为参数：
  size()、minelement()、maxelement()、minindex()、maxindex()，还有特别的 element()和 indices 函数
- in 与 between...and

```
from DomesticCat cat where cat.name between 'A' and 'B';
from DomesticCat cat where cat.name in ('Foo','Bar','Baz');
```

- is null 与 is not null 可以被用来测试空值

```
from DomesticCat cat where cat.name is null;
from Person as p where p.address is not null;
```

- size 关键字用于返回一个集合的大小
- 对于有序集合，还可以使用 minindex()与 maxindex()函数代表最小与最大的索引序数。同理 minelement()与 maxelement()函数代表集合中最小于最大的元素
- elements()和 indices()函数用于返回指定集合的所有元素和所有索引
- 在 where 子句中，有序集合(数组、List 集合、Map 对象)的元素可以通过[]运算符访问

6.4.9. order by 子句

1、查询返回的集合可以根据类或组件属性的任何属性进行排序

- 还可以指定 asc 或 desc 关键字指定升序或降序的排序规则
- 如果没有指定排序规则，则默认采用升序规则

```
from Person as p
order by p.name, p.age;

from Person as p
order by p.name asc, p.age desc;
```

6.4.10. group by 子句

1、返回聚集值的查询可以对持久化类或组件属性的属性进行分组，分组使用 group by 子句

- having 子句用于对分组进行过滤
- having 子句用于对分组进行过滤，因此 having 子句只能在 group by 子句时才可以使用

```
select cat
from Cat cat
join cat.kittens kitten
group by cat
having avg(kitten.weight) > 100
order by count(kitten) asc, sum(kitten.weight) desc;
```

6.4.11. 子查询

1、如果底层数据库支持子查询，则可以 HQL 语句中使用子查询，与 SQL 中查询相似的是，HQL 中的子查询也需要使用英文括号(())括起来

2、HQL 子查询只可以在 select 子句或 where 子句中出现

6.4.12. 命名查询

1、HQL 查询还支持将查询所用的 HQL 语句放入注解中，而不是代码中，通过这种方式，可以大大提高程序的解耦

2、Hibernate 支持使用标准的@NamedQuery 注解来配置命名查询，使用@NamedQuery 指定如下属性

- name：必须，该属性指定命名查询的名称
- query：必须，该属性指定命名查询所使用的 HQL 查询语句

3、如果程序需要使用多个@NamedQuery 定义命名查询，则可使用@NamedQueries 来组合多个@NamedQuery

4、实际上，命名查询的作用仅仅是将 HQL 语句从 Java 代码中提取出来

- 命名 HQL 查询的本质就是把查询语句从 Java 代码中取出来，放到注解中进行管理，并未这条查询语句起个名字
- 当使用 Hibernate 传统的 XML 映射文件时，命名查询的本质就是将查语句从 Java 代码中取出来，放到配合文件中管理，这样当应用需要修改查询语句时，开发者无需打开 Java 代码进行修改，直接修改配置文件中的查询语句即可
- 但对于这种在注解中配置命名查询的方式，似乎并没有太大的必要，因为注解本身依然在 Java 程序里
- 如果希望在 XML 文件中使用 XML 元素来定义命名查询，可以在<hibernate-mapping.../>元素中添加如下子元素

```
<query name="myNamedQuery">
    from Person as pwhere p.age > ?
</query>
```

6.5. 条件查询

1、条件查询是更具面向对象特色的数据查询方式，条件查询通过如下三个类完成

- Criteria：代表一次查询
- Criterion：代表一个查询条件
- Restrictions：产生查询条件的工具类

2、执行条件查询的步骤如下

- 获得 Hibernate 的 Session 对象
- 以 Session 对象创建 Criteria 对象
- 使用 Restrictions 的静态方法创建 Criterion 查询条件
- 向 Criteria 查询中添加 Criterion 查询条件
- 执行 Criteria 的 list()或 uniqueResult()方法返回结果集

3、Criteria 包含如下方法

- Criteria setFirstResult(int firstResult)：设置查询返回的第一行记录
- Criteria setMaxResult(int maxResults)：设置查询返回的记录数
- Criteria add(Criterion criterion)：增加查询条件
- Criteria addOrder(Order order)：增加排序规则
- List list()：返回结果集

4、Criterion 接口代表一个查询条件，该查询条件由 Restrictions 负责产生，Restrictions 是专门用于产生查询条件的工具类，它的方法大部分都是静态方法，常用方法详见 P490-491

6.5.1. 关联和动态关联

1、如果需要使用关联实体的属性来增加查询条件，则应该对属性再次使用 createCriteria()方法

2、使用关联类的条件查询，依然是查询原有持久化类的实例，而不是查询被关联类的实例

3、可以使用 createAlias()方法代替 createCriteria()方法

- createAlias()方法并不创建一个新的 Criteria 实例，它只是给关联实体(包括集合里包含的关联实体)起一个别名，让后面的过滤条件可以根据该关联实体进行筛选

4、在默认情况下，条件查询将根据持久化注解指定的延迟加载策略来加载关联实体，如果希望在条件查询中改变延迟加载策略，可以通过 Criteria 的 setFetchMode()方法来实现，该方法也接受一个 FetchMode 枚举类型的值，FetchMode 支持如下枚举类型

- DEFAULT：使用配置文件指定延迟加载策略处理
- JOIN：使用外连接、预初始化所有的关联实体
- SELECT：启用延迟加载，系统将使用单独的 select 语句来初始化关联实体，只有当真正访问关联实体的时候，才会执行第二条 select 语句

6.5.2. 投影、聚合和分组

1、投影运算实际上就是一种基于列的运算，通常用于投影到指定列(也就是过滤其他列，类似于 select 子句的作用)，还可以完成 SQL 语句中常用的分组、组筛选等功能

2、Hibernate 的条件过滤中使用 Projection 代表投影运算，Projection 是一个接口，而 Projections 作为 Projection 的工厂，负责生成 Projection 对象

3、一旦生产了 Projection 对象之后，就可通过 Criteria 提供的 setProjection(Projection projection)方法来进行投影运算

- 从该方法上看，每个 Criteria 只能接受一个投影运算，似乎无法进行多个投影运算，但实际上 Hibernate 又提供了一个 ProjectionList 类，该类是 Projection 的子类，并可以包含多个投影运算，通过这种方式即可完成多个投影运算

4、对于增加了投影运算后的条件查询，返回结果是数组，数组的前 N 个元素依次是投影运算的结果，最后一个数组元素才是条件查询得到的实体

5、如果希望对分组(投影)后属性进行排序，那就需要为投影运算指定一个别名，为投影运算指定别名有如下三个方法

- 使用 Projections 的 alias()方法为指定投影指定别名
  - Projections 的 alias()方法为指定 Projection 指定别名，并返回原 Projection 对象
  - 一旦指定了别名，就可以根据该 Projection 别名来进行其他操作了，比如排序

```
List list=session.createCriteria(Enrolment.class)
        .setProjection(Projections.projectionList()
            .add(Projections.groupProperty("course"))
            .add(Projections.alias(Projections.rowCount(),"c")))
            .addOrder(Order.asc("c"))
            .list();
```

- 使用 SimpleProjection 的 as()方法为自身指定别名
- 使用 ProjectionList 的 add()方法，有如下两种重载形式
  - 一种是直接添加一个投影
  - 另一种是在添加投影时指定别名
- 除此之外，Hibernate 还提供了 Property 执行投影运算，Property 投影的作用类似于 SQL 语句中的 select，条件查询的结果只有被 Property 投影的列才会被选出

6.5.3. 离线查询和子查询

1、条件查询的离线查询由 DetachedCriteria 来代表，DetachedCriteria 类允许在一个 Session 范围之外创建一个查询，并且可以使用任意 Session 来执行它

2、使用 DetachedCriteria 来执行离线查询，通常使用如下方法获得一个离线查询

```
DetachedCriteria.forClass(Class entity)
```

- DetachedCriteria 还可代表子查询，当把 DetachedCriteria 传入 Criteria 中作为查询条件时，DetachedCriteria 就变成了子查询
- 条件实例包含子查询通过 Subqueries 或 Property 来获得
- 当创建一个 DetachedCriteria()方法来执行 DetachedCriteria 对象，则它被当成离线查询使用，如果程序使用 Property 的系列方法来操作 DetachedCriteria 对象，则它被当成子查询使用

6.6. SQL 查询

1、Hibernate 还支持使用原生 SQL 查询，使用原生 SQL 查询可以利用某些数据库的特性，或者需要将原有的 JDBC 应用迁移到 Hibernate 应用上，也可能需要使用原生 SQL 查询

2、类似于 HQL 查询，原生 SQL 查询也支持将 SQL 语句放在配置文件中配置，从而提高程序的解耦

3、如果是一个新的应用，通常不要使用 SQL 查询

4、SQL 查询是通过 SQLQuery 接口来表示的。SQLQuery 接口是 Query 接口的子接口，因此完全可以调用 Query 接口的方法

- setFirstResult()：设置返回的结果集的起始点
- setMaxResults()：设置查询获取的最大记录数

- list()：返回查询到的结果集

5、SQLQuery 比 Query 多了两个重载方法

- addEntity()：将查询到的记录与特定的实体关联
- addScalar()：将查询的记录关联成标量值

6、执行 SQL 查询的步骤

- 获取 Hibernate Session 对象
- 编写 SQL 语句
- 以 SQL 语句作为参数，调用 Session 的 createSQLQuery()方法创建查询对象
- 调用 SQLQuery 对象的 addScalar()或 addEntity()方法将选出的结果与标量值或实体进行关联，分别用于进行标量查询或实体查询
- 如果 SQL 语句包含参数，则调用 Query 的 setXxx()方法为参数赋值
- 调用 Query 的 list()方法或 uniqueResult()方法返回查询的结果集

6.6.1. 标量查询

1、最基本的 SQL 查询就是获得一个标量(数值)列表，例如

```
session.createSQLQuery("select * from student_inf").list();
```

- 在默认情况下，上面的查询语句将返回由 Object 数组(Object[])组成的 list
- Hibernate 会通过 ResultSetMetadata 来判断所返回数据列的实际顺序和类型
- 如果 select 后面只有一个字段，那么返回的 List 集合元素就不是数组，而是单个的变量值

2、在 JDBC 中使用过多的 ResultSetMetadata 会降低程序性能，因此建议为这些数据列指定更明确的返回值类型。明确指定返回值类型通过 addScalar()方法来实现

```
session.createSQLQuery("select * from student_inf")
    .addScalar("name",StandardBasicTypes.STRING)
    .list();
```

- SQL 字符串
- 查询返回的字段列表
- 查询返回的个字段类型

3、标量查询中的 addScalar()方法有两个作用

- 指定查询结果包含哪些数据列---没有被 addScalar()选出的列将不会包含在查询结果中
- 指定查询结果中数据列的数据类型

6.6.2. 实体查询

1、标量值查询只返回一些标量的结果，这种查询方式与 JDBC 查询的效果基本类似。查询返回多个记录行，每行记录对应一个列表元素，这个列表元素是一个数组，每个数组元素对应当前行，当前列的值

2、如果查询返回了某个数据表的全部数据列(记住：是选出全部数据列)，且该数据表有对应的持久化类映射，接下来就可把查询结果转换成实体，将查询结果转换成实体，可以使用 SQLQuery 提供的多个重载的 addEntity()方法

3、使用原生 SQL 查询必须注意的是

- 程序必须选出所有数据列才可被转换成持久化实体，假设实体在映射时有一个@ManyToOne 关联指向另外一个实体，则 SQL 查询中必须返回该@ManyToOne 关联实体对应的外键列，否则将导致抛出"column not found"异常
- 最简单的做法是，在 SQL 字符串中使用星号(*)来表示返回所有列

4、Hibernate 还可将查询结果转换成非持久化实体(即普通 JavaBean)，只要改 JavaBean 为这些数据列提供了对应的 setter 和 getter 方法即可

5、Query 接口提供了一个 setResultTransformer()方法，该方法可接受一个 Transformers 对象，通过使用该对象即可把查询到的结果集转换成 JavaBean 集

6.6.3. 处理关联和继承

1、只要原生 SQL 查询选出了足够的数据列，则程序除了可以将指定数据列转换成持久化实体之外，还可以将实体的关联实体(通常以属性的形式存在)转换成查询结果

- 将关联实体转换成查询结果的方法是 SQLQuery addJoin(String alias,String path)，该方法的第一个参数是转换后的实体名，第二个参数是待转换的实体属性

6.6.4. 命名 SQL 查询

1、可以将 SQL 语句不放在程序中，而是放在注解中管理，这种方式以松耦合的方式来配置 SQL 语句，从而可以更好地提高程序解耦

2、Hibernate 允许使用@NamedNativeQuery 注解来定义命名的原生 SQL 查询，如果程序有多个命名的原生 SQL 查询需要定义，则可以使用@NamedNativeQueries 注解，它可用于组合多个命名的原生 SQL 查询

3、使用@NamedNativeQuery 时可指定如下属性

属性	是否必须	说明
name	是	该属性指定命名的原生 SQL 查询的名称
query	是	该属性指定原生 SQL 查询的查询字符串
resultClass	否	该属性指定一个实体类的类名，用于指定将查询结果集映射成该实体类的实例
resultSetMapping	否	该属性指定一个 SQL 结果映射集，用于指定使用该 SQL 结果映射来转换查询结果集

4、如果需要为@NamedNativeQuery 注解指定 resultSetMapping 属性，则还需要使用@SqlResultSetMapping 定义 SQL 结果映射，@SqlResultSetMapping 的作用是将查询得到的结果集转换为标量查询或实体查询---基本等同于 SQLQuery 对象的 addScalar()或 addEntity()方法的功能

- 使用@SqlResultSetMapping 可指定如下属性

属性	是否必须	说明
name	是	该属性指定 SQL 结果映射的名称
column	否	该属性的值为@ColumnResult 注解数组，每个@ColumnResult 注解定义一个标量查询
entities	否	该属性的值为@EntityResult 注解数组，每个@EntityResult 注解定义一个实体查询

classes	否	该属性的值为@ConstructorResult 注解数组，每个@ConstructorResult 负责将指定的多列转换为普通类的对应属性

- @ColumnResul 注解的作用类似于 SQLQuery 的 addScalar()方法的作用，程序为@SqlResultSetMapping 注解的 columns 属性指定了几个@ColumnResult，将相当于调用了 SQLQuery 的 addScalar()方法几次
- @EntityResult 注解的作用类似于 SQLQuery 的 addEntity()方法的作用，程序为@SqlResultSetMapping 注解的 entities 属性指定了几个@EntityResult，就相当于调用了 SQLQuery 的 addEntity 方法几次
- @ConstructorResult 注解的作用有点类似于 Query 的 setResultTransformers()方法的作用，该注解可把查询结果转换为普通的 JavaBean

5、示例详见 P503、504

6.6.5. 调用存储过程

1、从 Hibernate3 开始，Hibernate 可以通过命名 SQL 查询来调用存储过程或函数

- 对于函数，该函数必须返回一个结果集
- 对于存储过程，该存储过程的第一个参数必须是传出参数，且其数据类型是结果集

2、示例详见 P504-505

- 先定义一个存储过程
- 然后将调用存储过程"Call ???(...)"封装成@NamedNativeQuery 注解的形式
- 然后剩余过程就与命名 SQL 查询相同了

3、Hibernate 当前仅支持存储过程返回标量和实体，调用存储过程还有如下注意点

- 建议采用的调用方式是标准 SQL92 语法，如{?=call functionName(<parameters>)}或{call procedureName(<parameters>)}，不支持原生的调用语法
- 因为存储过程本身完成了查询的全部操作，因此，调用存储过程进行的查询无法使用 setFirstResult()/setMaxResult()进行分页

6.6.6. 使用定制 SQL

1、Hibernate 本身提供了极好的可扩展性，通过使用定制 SQL 可以完全控制 Hibernate 底层持久化所用的 SQL 语句

2、当 Hibernate 需要保存、更新、删除持久化实体时，默认通过一套固定的 SQL 语句来完成这些功能，如果程序需要改变这套默认的 SQL 语句，就可以使用 Hibernate 所提供的定制 SQL 功能

3、Hibernate 本身为定制 SQL 提供如下注解

- @SQLInsert：定制插入记录的 SQL 语句
- @SQLUpdate：定制更新记录的 SQL 语句
- @SQLDelete：定制删除记录的 SQL 语句
- @SQLDeleteAll：定制删除所有记录的 SQL 语句

- 上述 4 各注解都要指定一个 sql 属性，该属性值为插入、更新、删除、删除所有记录的 SQL 语句
- 一旦使用上面注解修饰了某个实体类，Hibernate 将不再使用默认的 SQL 语句来执行插入、更新、删除和删除所有记录的操作，而是使用此处定制的插入、更新、删除和删除所有记录的 SQL 语句进行操作
- 如果希望使用存储过程来执行插入、更新、删除、删除所有等操作，只需为上述 4 个注解指定 callable=true 即可

6. 7. 数据过滤

1、数据过滤并不是一种常规的数据查询方法，而是一种整体的筛选方法。数据过滤也可以对数据进行筛选，因此也把数据过滤当成 Hibernate 查询框架的一部分

2、如果一旦启用了数据过滤器，则不管数据查询还是数据加载，该过滤器将自动作用于所有数据，只有满足过滤条件的记录才会被选出来

3、过滤器与修饰持久化类的@Where 注解非常相似，它们的区别是过滤器可以带参数，应用程序可以在运行时决定是否启用指定的过滤器，使用怎样的参数值；而修饰持久化类的@Where 注解将一直生效，且无法动态传入参数

4、过滤器的用法很像数据库视图，区别是视图在数据库中已经定义完成，而过滤器则还需在应用程序中确定参数值

5、过滤器使用分三步

1) 定义过滤器。使用 Hibernate 提供的@FilterDef 注解定义过滤器。如果需要定义多个过滤器，还需要使用@FilterDefs 注解来组合多个@FilterDef
2) 使用过滤器。使用@Filter 元素应用过滤器
3) 在代码中通过 Session 启用过滤器

- @FilterDef 通常用于修饰持久化类，用于定义注解；
- @Filter 则通常用于修饰持久化类或集合属性(包括关联实体)，表示对指定持久化类或集合属性(包括关联实体)应用过滤器
- 一个持久化类或集合可以使用多个过滤器，而一个过滤器也可以作用域多个持久化类或集合属性(包括关联实体)

6、@FilterDef 可指定如下属性

属性	是否必须	说明
name	是	该属性用于指定过滤器的名称
defaultCondition	否	该属性的值为带参数的 SQL 条件表达式，用于指定该过滤器默认的过滤条件
parameters	否	该属性指定过滤器中 SQL 条件表达式支持的参数

- 定义过滤器只是指定该过滤器的名称，并指定该过滤条件所支持的参数
- 至于过滤条件，定义过滤器时可以无须指定，完全可以等到应用过滤时，使用@Filter 注解的 condition 属性指定

7、示例详见 P509

8、在一个持久化注解中定义的过滤器，完全可以在其他不同的持久化类中使用，前提是这些持久化类由一个 SessionFactory 负责加载并管理

6.8. 事务控制

1、每个业务逻辑方法都是由一系列数据库访问完成的，这一系列数据库访问可能会修改多条数据记录，这一系列修改应该是一个整体，决不能仅修改其中的几条记录。也就是说，多个数据库原子访问应该绑定成一个整体---这就是事务

2、事务是一个最小的逻辑执行单元，整个事务的执行不能分开执行，要么同时执行，要么同时放弃

6.8.1. 事务的概念

1、事务是一步或几步基本操作组成的逻辑执行单元，这些基本操作作为一个整体执行单元，它们要么全部执行，要么全部取消执行，决不能仅仅执行部分

2、一般而言，每次用户请求对应一个业务逻辑方法，一个业务逻辑方法旺旺具有逻辑上的原子性，应该使用事务

- 例如一个转账操作，对应修改两个账户余额，这两个账户的修改要么同时生效，要么同时取消---同时生效是转账成功，同时取消是转账失败。但不可只修改其中一个账户，那将破坏数据库的完整性

3、**通常来讲，事务具备 4 个特性：原子性(Atomicity)、一致性(Consistency)、隔离性(Isolation)和持续性(Durability)，简称为 ACID 性**

- 原子性(Automicity)：事务是应用中最小执行单位，就如原子是自然界最小颗粒，具有不可再分的特征一样。事务是应用中不可再分的最小逻辑执行体
- 一致性(Consistency)：事务执行的结果，必须是数据库从一种一致性状态，变到另一种一致性状态
  - 当数据库只包含事务成功提交的结果时，数据库处于一致性状态
  - 如果系统运行发生中断，某个事务尚未完成而被迫中断，而该未完成的事务对数据库所作的修改已被写入数据库，此时数据库就处于一种不正确的状态
  - 一致性是通过原子性来保证的
- 隔离性(Isolation)：各个事务的执行互不干扰，任意一个事务的内部操作对其他并发事务，都是隔离的。即并发执行的事务之间不能互相影响
- 持续性(Durability)：持续性也称为持久性(Persistence)，
- 事务一旦提交，对数据所作的任何改变都要记录到永久存储器中，通常就是保存进物理数据库

6.8.2. Session 与事务

1、Hibernate 的事务(Transaction 对象)通过 Session 的 beginTransaction()方法显式打开，Hibernate 自身并不提供事务控制行为(没有添加任何附加锁定行为)，Hibernate 底层直接使用 JDBC 连接，JTA 资源或其他资源的事务

2、Hibernate 只是对底层事务进行了抽象，让应用程序可以直接面向 Hibernate 事务编程，从而将应用程序和 JDBC 连接，JTA 资源或其他事务资源隔离开

- 从编程角度来看，Hibernate 的事务由 Session 对象开启
- 从底层实现来看，Hibernate 事务由 TransactionFactory 的实例来产生

3、TransactionFactory 是一个事务工厂的接口，Hibernate 为不同的事务环境提供了不同的实现类，如

- CMTTransactionFactory(针对容器管理事务环境的实现类)
- JDBCTransactionFactory(针对 JDBC 局部事务环境的实现类)
- JTATransactionFactory(针对 JTA 全局事务环境的实现类)

4、应用程序编程后无须手动操作 TransactionFactory 产生事务，这是因为 SessionFactory 底层已经封装了 TransactionFactory

- SessionFactory 对象的创建代价很高，它是线程安全的对象，被设计成可以被所有线程共享
- 通常 SessionFactory 会在应用程序启动时创建，一旦创建了 SessionFactory 就不会轻易关闭，只有当应用程序退出才关闭 SessionFactory

5、Session 对象是轻量级的，它也是线程不安全的。对于单个业务进程、单个工作单元而言，Session 只被使用一次

- 创建 Session 时，并不会立即打开与数据库之间的连接，只有需要进行数据库操作时，Session 才会获取 JDBC 连接
- 打开和关闭 Session，并不会对性能造成很大的影响。甚至无法确定一个请求是否需要数据库访问，也可以打开 Session 对象，因为如果不进行数据库访问，Session 不会获取 JDBC 连接
- 因此，长 Session 对应用性能的影响并不大，只要它没有长时间打开数据库连接

6、数据库事务应该尽可能短，从而降低数据库锁定造成的资源争用。数据库长事务会导致应用程序无法承载高并发的负荷

7、Hibernate 的所有持久化访问都必须在 Session 管理下进行，但并不推荐因为一次简单的数据库原子调用，就打开和关闭一次 Session，数据库事务也是如此。因为对于一次原子操作打开的事务没有任何意义---事务应该是将多个操作组合成一个逻辑整体

8、Hibernate 建议采用每个请求对应一次 Session 的模式---因此一次请求通常表示需要执行一个完整的业务功能，这个功能由一系列数据库原子操作组成，而且它们应该是一个逻辑上的整体

- 每个请求对应一次 Session 的模式不仅可以用于设计操作单元，甚至很多业务处理流程都需要组合一系列用户操作，即用户对数据库的交叉访问

9、几乎在所有情况下，都不要使用每个应用对应一次 Hibernate Session 模式，也尽量不要使用每次 HTTP Session 对应一次 Hibernate Session 的模式。但在实际应用中，常常需要面对这种应用程序长事务(例如在第一个页面，用户打开对话框，用户打开一个特定 Session 装入的数据，用户可以随意修改对话框中的数据，修改完后，用户将修改结果存入数据库)，对于这种情况，Hibernate 主要有如下三种方式来解决

- 自动化版本：Hibernate 能够自动进行乐观并发控制，如果在用户思考的过程中持久化实体发生并发修改，Hibernate 能够自动检测到
- 脱管对象：如果采用每次用户请求对应一次 Session 的模式，那么前面载入的实例在用户思考的过程中，始终与 Session 脱离，处于脱管状态。Hibernate 允许把脱管对象重新关联到 Session 上，并且对修改进行持久化。在这种模式下，自动版本化被用来隔离并发修改。这种模式也被称为使用脱管对象的每次请求对应一个 Hibernate Session

- 长生命周期 Session：Session 可以在数据库事务提交之后，断开和底层的 JDBC 连接。当新的客户端请求到来时，它又重新连接上底层的 JDBC 连接。这种模式被称为每个应用程序事务对应一个 Session。因为应用程序事务是相当长(跨越多个用户请求)的，所以也被称为长生命周期 Session

10、Session 缓存了处于持久化状态的每个对象(Hibernate 会监视和检查脏数据)

- 也就是说，如果程序让 Session 打开很长一段时间，或者载入了过多的数据，Session 占用的内存会一直增长，直到抛出 OutOfMemoryException 异常
- 为了解决这个问题，程序定期调用 Session 的 clear()和 evict()方法来管理 Session 缓存。对于大批量的数据处理，推荐使用 DML 风格和 HQL 语句完成

11、如果在 Session 范围之外访问未初始化的集合或代理(由 Hibernate 的延迟加载特性引起)，Hibernate 将会抛出 LazyInitializationException 异常。也就是说，在脱管状态下，访问一个实体所拥有的集合，或者访问其指向代理的属性时，都将引发异常

12、为保证在 Session 关闭之前初始化代理属性或集合属性，程序可以强行调用 teacher.getName()之类的方法来实现(通过代理来实际加载底层数据???)，但这样代码具有较差的可读性。除此之外，也可使用 Hibernate 的 initialize(Object proxy)静态方法来强制初始化某个集合或代理，只要 Session 处于打开状态，Hibernate.initialize(teacher)将会强制初始化 teacher 代理，Hibernate.initialize(teacher.getStudents())对 students 集合具有同样的功能

13、还有另外一种选择，就是程序让 Session 一直处于打开状态，直到装入所有需要的集合或代理。在某些应用架构中，特别是对于那些需要使用 Hibernate 进行数据访问的代码，以及那些需要在不同应用层和不同进程中使用 Hibernate 的应用，如何保证 Session 处于打开状态也是一个问题，通常有两种方法解决这个问题

- 在一个 Web 应用中，可以使用过滤器(Filter)，在用户请求结束、页面生成结束时关闭 Session。也就是保证在视图显示层一直打开 Session，这就是所谓的 Open Session in View 模式。采用这种模式时，必须保证所有的异常得到正确处理，在呈现视图界面之前，或者在生成视图界面的过程中发生异常，必须保证可以正确关闭 Session，并结束事务
- 让业务逻辑层来负责准备数据，在业务逻辑层返回数据之前，业务逻辑层对每个所需集合调用 Hibernate.initialize()方法，或者使用带有 fetch 子句或 FetchMode.JOIN 的查询，事先取得所有数据，并将这些数据封装成 VO(值对象)集合，然后程序可以关闭 Session 了。业务逻辑层将 VO 集合传入视图层，让视图层只负责简单的显示逻辑。在这种模式下，可以让视图层和 Hibernate API 彻底分离，保证视图层不会出现持久层 API，从而提供更好的解耦

6.8.3. 上下文相关的 Session

1、从 Hibernate 3 开始，Hibernate 增加了 SessionFactory.getCurrentSession()方法，该方法可以直接获取"上下文相关"的 Session

- 上下文相关的 Session 的早期实现必须依赖于 JTA 事务，因此比较适合于

在容器中使用 Hibernate 的情形

2、从 Hibernate 3.1 开始，SessionFactory.getCurrentSession()的底层实现是可插拔的，Hibernate 引入了 CurrentSessionContext 接口，并通过 hibernate.current_session_context_class 参数来管理上下文相关的 Session 的底层实现

3、CurrentSessionContext 接口有如下三个实现类

1) org.hibernate.context.JTASessionContext：根据 JTA 来跟踪和界定上下文相关的 Session。这和最早的支持 JTA 的方法是完全一样的
2) org.hibernate.context.ThreadLocalSessionContext：通过当前正在执行的线程来跟踪和界定上下文相关的 Session
3) org.hibernate.context.ManagedSessionContext：通过当前执行的线程来跟踪和界定上下文相关的 Session。但是程序需要使用这个类的静态方法将 Session 实例绑定、取消绑定，它并不会自动打开、flush 或者关闭任何 Session

- 如果使用 ThreadLocalSessionContext 策略，Hibernate 的 Session 会随着 getCurrentSession()方法自动打开，并随着事务提交自动关闭，非常方便
  - 对于在容器中使用 Hibernate 的场景，通常会采用第一种方式
  - 对于独立的 Hibernate 应用而言，通常会采用第二种方式

6.9. 二级缓存和查询缓存

1、Hibernate 包括两个级别的缓存

1) 默认总是启用 Session 级别的一级缓存
2) 可选的 SessionFactory 级别的二级缓存

- 其中 Session 级别的一级缓存并不需要开发者关心，默认总是有效的，当应用保存持久化实体、修改持久化实体时，Session 并不会立即把这种改变 flush 到数据库，而是缓存在当前 Session 的一级缓存中，除非程序显式调用 Session 的 flush()方法，或程序关闭 Session 时才会把这些改变一次性地 flush 到底层数据库。通过这种缓存，可以减少与数据库的交互，从而提高数据库访问性能
- SessionFactory 级别的二级缓存时全局性的，应用的所有 Session 都共享这个二级缓存。该二级缓存默认是关闭的，必须由程序显式开启。一旦在应用中开启了二级缓存，当 Session 需要抓取数据时，Session 会先查找一级缓存，再查找二级缓存，当只有一级缓存和二级缓存中没有需要抓取的数据时，才回去查找底层数据库

6.9.1. 开启二级缓存

1、为开启 Hibernate 的二级缓存，需要在 hibernate.cfg.xml 文件中设置如下属性

```
<property name="hibernate.cache.use_second_level_cache">true</property>
```

- 一旦开启了二级缓存，并且设置了对某个持久化实体类启用缓存，SessionFactory 就会缓存应用访问过的该实体类的每个对象，除非缓存的数据超出缓存空间

2、实际应用一般不需要开发者自己实现缓存，直接使用第三方提供的开源缓存实现即可

- 因此，在 hibernate.cfg.xml 文件中设置开启缓存之后，还需要设置使用哪种二级缓存实现类

```
<property name="hibernate.cache.region.factory_class">
    org.hibernate.cache.ehcache.EhCacheRegionFactory</property>
```

- Hibernate 4.3 支持如下缓存实现

缓存	缓存实现类	类型	集群安全	查询缓存支持
ConcurrentHashMap	org.hibernate.testing.cache.CachingRegionFactory	内存		
EhCache	org.hibernate.cache.ehcache.EhCacheRegionFactory	内存、磁盘、事务性、支持集群	是	是
Infinispan	org.hibernate.cache.infinispan.InfinispanRegionFactory	事务性、支持集群	是	是

  - 其中 ConcurrentHashMap 只是一种内存级别的缓存实现，因此这种缓存实现类只是 Hibernate 作为测试使用的缓存实现，因此不推荐在实际项目中使用这种策略
- 可以把 Hibernate 的缓存理解为一个 Map

3、以常见的 EhCache 为例介绍 Hibernate 二级缓存用法

1. 在 hibernate.cfg.xml 文件中开启二级缓存，需要做两件事情
   1) 设置启用二级缓存
   2) 设置二级缓存的实现类
2. 复制二级缓存的 JAR 包，将 Hibernate 项目路径下的 lib\optional\下的对应缓存的 JAR 包赋值到应用的类加载路径中
   -
3. 将缓存实现所需要的配置文件添加到系统的类加载路径中，对于 EhCache 缓存，它需要一个 ehcache.xml 配置文件
   - 配置文件中各属性说明如下
   - maxElementsInMemory：设置缓存中最多可放多少个对象
   - eternal：设置缓存是否永久有效
   - timeToIdleSeconds：设置缓存的对象多少秒没有被使用就会清理掉
   - timeToLiveSeconds：设置缓存的对象在过期之前可以缓存多少秒
   - diskPersistent：设置缓存是否被持久化到硬盘中，保存路径由<diskStore.../>元素指定
4. 设置对哪些实体类、实体的那些集合属性启用二级缓存，这一步有两种方式
   1) 修改要使用缓存的持久化类文件，使用 Hibernate 提供的@Cache 注解修饰该持久化类，或使用该注解修饰集合属性
   2) 在 hibernate.cfg.xml 文件中使用<class-cache.../>或<collection-cache.../>元素对指定的持久化类、集合属性启用二级缓存
   - 这两种方式只是形式不同，本质一样，推荐使用第一种

4、Hibernate 支持的缓存策略

- 只读(READ_ONLY)：如果应用程序只需读取持久化实体的对象，无需对其进行修改，那么就可以对其设置"只读"缓存策略
- 读/写缓存策略(READ_WRITE)：如果应用程序需要更新数据，那么就需要使用"读/写"缓存策略，如果应用程序要求使用"序列化事务"的隔离级别，那么就不能使用这种缓存策略

- 非严格读/写(NONSTRICT_READ_WRITE)：如果应用程序只需要偶尔更新数据(也就是说，两个事务同时更新同一记录的情况很少见)，也不需要十分严格的事务隔离，那么比较适合使用非严格读/写缓存策略
- 事务缓存(TRANSACTIONAL)：Hibernate 的事务缓存策略提供了全事务的缓存支持，这样的缓存只能在 JTA 环境中使用，必须指定 hibernate.transaction.manager_lookup_class 属性的值

6.9.2. 管理缓存和统计缓存

1、Session 级别的一级缓存是局部缓存，它只对当前 Session 有效；SessionFactory 级别的二级缓存时全局缓存，它对所有的 Session 都有效

- 对于 Session 级别的一级缓存而言，所有经它操作的实体，不管使用 save()、update()或 saveOrUpdate()方法保存一个对象，还是使用 load()、get()、list()、iterate()或 scroll()方法获得一个对象，该对象都将被放入 Session 级别的一级缓存中---在 Session 调用 flush()方法或 close()方法之前，这些对象一直在一级缓存中
- 在某些特殊情况下，可能需要从一级缓存中去掉某个大对象或集合属性，可以调用 Session 的 evict(Object object)方法，将该对象或集合从一级缓存中剔除
- 为了判断某个对象是否处于 Session 缓存中，可以借助于 Session 提供的 contains(Object object)方法，该方法返回一个 boolean 值，用于标识某个实例是否处于当前 Session 的缓存中

2、类似地，Hibernate 同样提供了方法来操作 SessionFactory 的二级缓存所缓存的实体

- SessionFactory 提供了一个 getCache()方法，该方法的返回值是 Cache 对象，通过该对象即可操作二级缓存中的实体、集合等

6.9.3. 使用查询缓存

1、一级、二级缓存都是对整个实体进行缓存，他不会缓存普通属性，如果相对普通属性进行缓存，则可以考虑使用查询缓存

2、需要指出的是：在大部分情况下，查询缓存并不能提供应用性能，甚至反而会降低应用性能，因此在实际项目中请慎重使用查询缓存

3、对于查询缓存来说，它缓存的 key 就是查询所用的 HQL 或 SQL 语句，需要指出的是，查询缓存不仅要求所使用的 HQL 语句、SQL 语句相同，甚至要求所传入的参数也相同，Hibernate 才能直接从查询缓存中取得数据

4、查询缓存也是默认关闭的，为了开启查询缓存，必须在 hibernate.cfg.xml 文件中增加如下配置

```
<property name="hibernate.cache.use_query_cache">true</property>
```

5、示例详见 P519-520

6.10. 事件机制

1、在 Hibernate 执行持久化的过程中，应用程序通常无法参与其中，所有的数据持久化操作，对用户都是透明的，用户无法插入自己的动作

2、通过事件框架，Hibernate 允许应用程序能响应特定的内部事件，从而允许

实现某些通用的同能，或者对 Hibernate 功能进行扩展

3、Hibernate 的事件框架由两个部分组成

- 拦截器机制：对于特定动作拦截，回调应用中的特定动作
- 事件系统：重写 Hibernate 的事件监听器

6.10.1. 拦截器

1、通过 Interceptor 接口，可以从 Session 中回调应用程序的特定方法，这种回调机制可让应用程序在持久化对象被保存、更新、删除或加载之前，检查并修改器属性

2、通过 Interceptor 接口，可以在数据进入数据库之前，对数据进行最后的检查，如果数据不符合要求，则可以修改数据，从而避免非法数据进入数据库

3、使用拦截器按如下步骤进行

1) 定义实现 Interceptor 接口的拦截器类：
   - 程序可以通过实现 Interceptor 接口来创建拦截器，但最好通过继承 EmptyInterceptor 来实现拦截器
   - EmptyInterceptor 和 Interceptor 的关系就像事件监听器和事件适配器的关系
2) 通过 Session 启用拦截器，或者通过 Configuration 启用全局拦截器

4、拦截器的使用有两种方法

1) 通过 SessionFactory 的 openSession(Interceptor in)方法打开一个带局部拦截器的 Session
2) 通过 Configuration 的 setInterceptor(Interceptor in)方法设置全局拦截器

6.10.2. 事件系统

1、Hibernate 的事件系统是功能更强大的事件框架，事件系统完全可以替代拦截器，也可以作为拦截器的补充来使用

2、Session 接口的每个方法都有对应的事件，比如 LoadEvent、FlushEvent 等，当 Session 调用某个方法时，Hibernate Session 会生成对应的事件，并激活对应的事件监听器

3、系统默认监听器实现的处理过程，完成了所有的数据持久化操作，包括插入、修改等操作。如果用户定义了自己的监听器，则意味着用户必须完成对象的持久化操作

4、监听器是单例模式对象，即所有同类型的事件处理共享同一个监听器实例，因此监听器不应该保存任何状态，即不应该使用成员变量

5、使用事件系统按如下步骤进行

1) 实现自己的事件监听器类
2) 注册自定义事件监听器，代替系统默认的事件监听器

6、实现用户自定义监听器有如下三种方法

1) 实现对应的监听器接口：实现接口必须实现接口所有方法，关键是必须实现 Hibernate 对应的持久化操作，及数据库访问，这意味着程序员完全取代了 Hibernate 的底层操作
2) 继承事件适配器：可以有选择地实现需要关注的方法，但依然视图取代 Hibernate 完成数据库的访问

3) 继承系统默认的事件监听器：扩展特定方法

- 通常推荐使用第三种方法实现自己的事件监听器，Hibernate 默认的事件监听器都被声明成 non-final，以便用户继承他们
- 扩展用户自定义监听器时，别忘了在方法中调用父类的对应方法，否则 Hibernate 默认的持久化行为都会失效，因为这些持久化行为本身就是通过拦截器完成的

7、为了让开发者注册自定义的事件监听器，Hibernate 提供了一个 EventListenerRegistry 接口，该接口提供如下三类方法来注册事件监听器

- appendListeners()：该方法有两个重载的版本，都用于将自定义的事件监听器追击到系统默认的事件监听器序列的后面
- prependListeners()：该方法有两个重载版本，都用于将自定义的事件监听器添加到系统默认的事件监听器序列的前面
- setListeners()：该方法有两个重载版本，都用于使用自定义的事件监听器代替系统默认的事件监听器序列

8、通过 Hibernate 的事件系统，开发者可以在 Hibernate 执行持久化操作的同时插入额外的行为，比如记录数据库访问日志等，事件系统是一种强大的扩展机制，只要开发者向对 Hibernate 的持久化操作添加额外的行为，都可通过扩展 Hibernate 的事件系统来完成

Chapter 7. Spring的基本用法

1、Spring是一个很普通但很实用的框架，它提取了大量实际开发中需要重复解决的步骤，将这些步骤抽象成一个框架

7.1. Spring简介和Spring4.0的变化

7.1.1. Spring简介

1、Spring是一个从实际开发中抽取出来的框架，因此它完成了大量开发中的通用步骤，留给开发者的仅仅是与特定应用相关的部分，从而大大提高了企业应用的开发效率

2、Spring为企业应用的开发提供了一个轻量级的解决方案，包括基于依赖注入的核心机制、基于AOP的声明式事务管理、与多种持久层技术的整合，以及优秀的Web MVC框架等

3、总结起来Spring有如下优点

- 低侵入式设计，代码污染极低
- 独立于各种应用服务器，基于Sprint框架的应用，可以真正实现Write One，Run Anywhere的承诺
- Spring的IoC容器降低了业务对象替换的复杂性，提高了组件之间的解耦
- Spring的AOP支持允许将一些通用任务如安全、事务、日志等进行集中式处理，从而提供了更好的复用
- Spring的ORM和DAO提供了与第三方持久层框架的良好整合，并简化了底层的数据库访问
- Spring高度开放性，并不强制应用完全依赖于Spring，开发者可自由选用Spring框架的部分或全部

4、当使用Spring框架时，必须使用Spring Core Container(Spring容器)，它代表了Spring框架的核心机制

7.1.2. Spring 4.0的变化

1、变化包括

- Spring 4.0全面支持Java 8，如新增的java.time包下的日期、时间类
- 删除了一些已过时的包和类
- 核心IoC容器新增了泛型限定式依赖注入、Map依赖注入、List(数组)注入、延迟注入的功能，这些功能主要体现在基于注解的配置上
- Spring 4.0的Web支持已经升级为支持Servlet 3.0以及更高的规范
- 从Spring 4.0开始，Spring支持使用Groovy DSL进行Bean配置
- Spring 4.0新增了spring-websocke模块，该模块支持WebSocket、SocketJS、STOMP通信

7.2. Spring入门

7.2.1. Spring下载和安装

1、详见P528

7.2.2. 使用Spring管理Bean

1、Spring核心容器的理论很简单：Spring容器就是一个超级大工厂，所有的对

象(包括数据源、Hibernate SessionFactory 等基础性资源)都会被当成 Spring 核心容器管理的对象---Spring 把容器中的一切对象统称为 Bean

2、Spring 容器中的 Bean，与以前听过的 Java Bean 是不同的，不像 Java Bean，必须遵守一些特定的规范，而 Spring 对 Bean 没有任何要求，只要是一个 Java 类，Spring 就可以管理该 Java 类，并将它当成 Bean 处理

3、对 Spring 框架而言，一切 Java 对象都是 Bean

4、使用 Spring 框架之后，Spring 核心容器是整个应用中的超级大工厂，所有的 Java 对象都交给 Spring 容器管理---这些 Java 对象被统称为 Spring 容器中的 Bean

- Spring 使用 XML 配置文件来管理容器中的 Bean
- Spring 对 XML 配置文件的文件名没有任何要求，可以随意指定
- 配置文件的根元素是<beans.../>，根元素主要就是包括多个<bean.../>元素，每个<bean.../>元素定义一个 Bean

5、Spring 可以把一切"Java 对象"当成容器中的 Bean，因此不管该 Java 类是 JDK 提供的还是第三方框架提供的，抑或是开发者自己实现的，只要是个 Java 类，并将它配置在 XML 配置文件中，Spring 容器就可以管理它

- 配置文件中的<bean.../>元素默认以反射方式来调用该类的无参构造器
- Spring 框架通过反射根据<bean.../>元素的 class 属性指定的类名创建一个 Java 对象，并以<bean.../>元素的 id 属性的值为 key，将该对象放入 Spring 容器中---这个 Java 对象就成为了 Spring 容器中的 Bean
  - 其中，class 属性的值必须是 Bean 实现类的完整类名(必须带包名)，不能是接口，不能是抽象类，否则 Spring 无法使用反射创建该类的实例
- <property.../>子元素通常作为<bean.../>元素的子元素，它驱动 Spring 在底层以反射执行一次 setter 方法，其中<property.../>的 name 属性值决定执行哪个 setter 方法，而 value 或 ref 决定执行 setter 方法的传入参数
  - 如果传入参数是基本类型及其包装类型、String 等类型，则使用 value 属性指定传入参数
  - 如果以容器中其他 Bean 作为传入参数，则使用 ref 属性指定传入参数
- Spring 框架只要看到<property.../>子元素，Spring 框架就会在底层以反射方式执行一次 setter 方法。也就是说，<bean.../>元素驱动 Spring 调用构造器创建对象，<property.../>子元素驱动 Spring 执行 setter 方法

6、程序可以通过 Spring 容器来访问容器中的 Bean，ApplicationContext 是 Spring 容器最常用的接口，该接口有如下两个实现类

- ClassPathXmlApplicationContext：从类加载路径下搜索配置文件，并根据配置文件来创建 Spring 容器
- FileSystemXmlApplicationContext：从文件系统的相对路径或绝对路径下去搜索配置文件，并根据配置文件来创建 Spring 容器
- 对于 Java 项目而言，类加载路径总是稳定的，因此通常总是使用 ClassPathXmlApplicationContext 创建 Spring 容器

7、Spring 容器获取 Bean 对象主要有如下两种方法

- Object getBean(String id)：根据 Bean 的 id 来获取指定 Bean，获取 Bean 之后需要进行强制类型转换

- T getBean(String name,Class<T>requiredType)：根据容器中的 Bean 的 id 来获取 Bean，但该方法带一个泛型参数，因此获取 Bean 之后无须进行强制类型转换

8、使用 Spring 框架最大的改变之一：程序不再使用 new 调用构造器创建 Java 对象，所有的 Java 对象都由 Spring 容器负责创建

7.2.3. 在 Eclipse 中使用 Spring

1、详见 P532-535，没什么特别的

7.3. Spring 的核心机制：依赖注入

1、纵观所有 Java 应用(从基于 Applet 的小应用到多层结构的企业级应用)，这些应用中大量存在对象需要调用 B 对象方法的情形，这种情形被 Spring 称为依赖，即 A 对象依赖 B 对象

2、对于 Java 应用而言，它们总是由一些互相调用的对象构成，Spring 把这种互相调用的关系称为依赖关系

3、**Spring 框架的核心功能有两个**

1) Spring 容器作为超级大工厂，负责创建、管理所有的 Java 对象，这些 Java 对象被称为 Bean
2) Spring 容器管理容器中的 Bean 之间的依赖关系，Spring 使用一种被称为"依赖注入"的方式来管理 Bean 之间的依赖关系

- 通过这种依赖注入，Java EE 应用中的各种组件不需要以硬编码方式耦合在一起，甚至无须使用工厂模式
- 依赖注入达到的效果，非常类似于传说中的共产主义，当某个 Java 实例需要其他 Java 实例时，系统自动提供所需要的实例，无须程序显式获取

7.3.1. 理解依赖注入

1、Spring 并不是依赖注入的首创者，但 Rod Johnson 是第一个高度重视以配置文件来管理 Java 实例的协作关系的人，他给这种方式起了一个名字：控制反转(Inversion of Control，IoC)。在后来的日子里，Martine Fowler 为这种方式起了另一个名字：依赖注入(Dependency Injection)

2、因此不管依赖注入还是控制反转，其含义完全相同，当某个 Java 对象(调用者)需要调用另一个 Java 对象(被依赖对象)的方法时，在传统模式下通常有如下两种做法

1) 原始做法：调用者主动创建被依赖对象，然后再调用被依赖对象的方法
2) 简单工厂模式：调用者先找到被依赖对象的工厂，然后主动通过工厂去获取被依赖对象，最后在调用被依赖对象的方法

- 对于第一种方式，由于调用者需要通过形如"new 被依赖者对象构造器"的代码创建对象，因此必然导致调用者与被依赖对象实现类的硬编码耦合，非常不利于项目升级的维护
- 对于简单工厂的方式，大致需要把握三点
  - 调用者面向被依赖者对象的接口编程
  - 将被依赖对象的创建交给工厂完成
  - 调用者通过工厂来获得被依赖组件

- 这种方式的唯一缺点是，调用组件需要主动通过工厂去获取被依赖对象，这就会带来调用组件与被依赖对象工厂的耦合

3、使用 Spring 框架之后，调用者无须主动获取被依赖对象，调用者只要被动接受 Spring 容器为调用者的成员变量赋值即可(只要配置一个<property.../>子元素，Spring 就会执行相应的 setter 方法为调用者的成员变量赋值)

- 由此可见，使用 Spring 框架之后，调用者获取被依赖对象的方式由原来的主动获取变成了被动接受---于是 Rod Johnson 将这种方式称为控制反转
- 从 Spring 容器的角度看，Spring 容器负责将被依赖对象赋值给调用者的成员变量---相当于为调用者注入它依赖的实例，因此 Martine Fowler 将这种方式称为控制反转
- 正是因为 Spring 将被依赖对象注入给调用者，所以调用者无须主动获取被依赖对象，只要被动等待 Spring 容器注入即可，由此可见，控制反转和依赖注入其实是同一个行为的两种表达，只是描述的角度不同而已

4、以人和斧子为例理解依赖注入

1) 在原始社会，几乎没有社会分工。需要斧头的人(调用者)只能自己去磨一把斧头(被依赖对象)。对应的情形为：Java 程序里的调用者自己创建被依赖对象，通常采用 new 关键字调用构造器创建一个被依赖对象，这是 Java 初学者经常干的事情
2) 进入工业社会，工厂出现了，斧头不再由普通人完成，而在工厂里被生产出来，此时需要斧头的人(调用者)找到工厂，购买斧头，无须关心斧头的制作过程。对应简单工厂设计模式，调用者只需要定位工厂，无须理会被依赖对象的具体实现过程
3) 进入"共产主义"社会，需要斧头的人甚至无需定位工厂，"坐标"社会提供即可。调用者无须关心被依赖对象的实现，无须理会工厂，等待 Spring 依赖注入

- 第二种情况下，Java 实例的调用者创建被调用的 Java 实例，调用者直接使用 new 关键字创建被依赖对象，程序高度耦合，效率低下，真正应用极少采用这种方式，其坏处如下
  - 可扩展性差：由于"人"组件与"斧头"组件的实现类高度耦合，当试图扩展斧头类时，人组件的代码也要随之改变
  - 各组件职责不清：对于人而言，只需要关心调用斧头组件的方法即可并不关心斧头组件的创建过程，但在这种模式下，"人"组件却要主动创建"斧头"组件，职责混乱
- 第二种情况下，调用者无须关心被依赖对象的具体实现过程，只需要找到符合某种标准(接口)的实例，即可使用。此时调用的代码面向接口编程，可以让调用者和被依赖对象的实现解耦，这也是工厂模式大量使用的原因，但调用者依然需要主动定位工厂，调用者与工厂耦合在一起
- 第三种情况下，程序完全无须理会被依赖对象的实现，也无须主动定位工厂，这是一种优秀的解耦方式。实例之间的一来关系由 IoC 容器负责管理

5、使用 Spring 框架之后的两个主要改变：

- 程序无须使用 new 调用构造器创建对象，所有的 Java 对象都可交给 Spring 容器创建

- 当调用者需要调用被依赖对象的方法时，调用者无须主动获取被依赖对象，只要等待 Spring 容器注入即可

6、依赖注入通常有如下两种方式

- 设值注入：IoC 容器使用成员变量的 setter 方法来注入被依赖对象
- 构造注入：IoC 容器使用构造器来注入被依赖对象

7.3.2. 设值注入

1、设值注入是指 IoC 容器通过成员变量的 setter 方法来注入被依赖对象，这种注入方式简单、直观，因而在 Spring 的依赖注入里大量使用

2、Spring 推荐面向接口编程，不管是调用者还是被依赖对象，都应该为之定义接口，程序应该面向它们的接口，而不是面向实现类编程，这样以便程序后期的升级、维护

3、Srping 推荐面向接口编程，这样可以更好地让规范和实现分离，从而提供更好的解耦。对于一个 Java EE 应用，不管是 DAO 组件，还是业务逻辑组件，都应该先定义一个接口，该接口定义了该组件应该实现的功能，但功能的实现则由其实现类提供

4、Spring 采用 XML 配置文件，从 Spring 2.0 开始，Spring 推荐采用 XML Schema 来定义配置文件的语义约束，当采用 XML Schema 来定义配置文件的语义约束时，还可以利用 Spring 配置文件的扩展性，进一步简化 Spring 配置

5、在配置文件中，Spring 配置 Bean 实例通常会指定两个属性

1) id：指定该 Bean 的唯一标识，Spring 根据 id 属性值来管理 Bean，程序通过 id 属性值来访问该 Bean 实例
2) class：指定该 Bean 的实现类，此处不再用接口，必须使用实现类，Spring 容器会使用 XML 解析器读取该属性值，并利用反射来创建该实现类的实例

- Bean 与 Bean 之间的依赖关系放在配置文件里组织，而不是写在代码里，通过配置文件的指定，Spring 能精确地为每个 Bean 成员变量注入值
- Spring 会自动检测每个<bean.../>定义里的<property.../>元素定义，Spring 会在调用默认的构造器创建 Bean 实例后，立即调用对应的 setter 方法为 Bean 的成员变量注入值
- 每个 Bean 的 id 属性是该 Bean 的唯一标识，程序通过 id 属性值访问 Bean，Spring 容器也通过 Bean 的 id 属性值管理 Bean 与 Bean 之间的依赖

6、Bean 与 Bean 之间的依赖关系由 Spring 管理，Spring 采用 setter 方法为目标 Bean 注入所依赖的 Bean，这种方式被称为设值注入

7、依赖注入以配置文件管理 Bean 实例之间的耦合，让 Bean 实例之间的耦合从代码层次分离出来，依赖注入是一种优秀的解耦方式

8、Spring IoC 的三个基本要点

1) 应用程序的各个组件面向接口编程，面向接口编程可以将组件之间的耦合关系提升到接口层次，从而有利于项目后期的扩展
2) 应用程序的各组件不再由程序主动创建，而是由 Spring 容器来负责产生并初始化
3) Spring 采用配置文件或注解来管理 Bean 的实现类，依赖关系，Spring 容器则根据配置文件或注解，利用反射来创建实例，并为之注入依赖关系

7.3.3. 构造注入

1、通过 setter 方法为目标 Bean 注入依赖关系的方式被称为设值注入；另外还有一种注入方式，这种方式在构造实例时，已经为其完成了依赖关系的初始化。这种依赖构造器来设置依赖关系的方式，被称为构造注入

2、Spring 在底层以反射的方式执行带指定参数的构造器，当执行带参数的构造器时，就可利用构造器参数对成员变量执行初始化---这就是构造注入的本质

3、可以利用<constructor-arg.../>子元素来让 Spring 调用有参数的构造器去创建对象(默认是用无参构造器创建对象的)

- 每个<constructor-arg.../>子元素代表一个构造器参数
- <constructor-arg.../>可指定 index 属性，用于指定该构造参数值作为第几个构造参数值
- 为了明确指定数据类型，Spring 允许为<constructor-arg.../>元素指定一个 type 属性，例如<<constructor-arg value="23" type="int"/>

4、两种注入方式的差异

- 设值注入是先通过无参构造器创建一个 Bean 实例，然后调用对应的 setter 方法注入依赖关系
- 构造注入直接调用有参数的构造器，当 Bean 实例创建完毕之后，已经完成了依赖关系的注入

7.3.4. 两种注入方式的比对

1、设置注入具有如下优点

- 与传统的 JavaBean 的写法更相似，程序开发人员更容易理解、接受。通过 setter 方法设定依赖关系显得更加直观、自然
- 对于复杂的依赖关系，如果采用构造注入，会导致构造器过于臃肿，难以阅读。Spring 在创建 Bean 实例时，需要同时实例化其依赖的全部实例，因而导致性能下降。而是用设置注入，则能避免这些问题
- 尤其是在某些成员变量可选的情况下，多参数的构造器更加笨重

2、构造注入有如下优点

- 构造注入可以在构造器中决定依赖关系的注入顺序，优先依赖的优先注入。采用构造注入可以在代码中清晰地决定注入顺序
- 对于依赖关系无须变化的 Bean，构造注入更有用处。因为没有 setter 方法，所有的依赖关系全部在构造器内设定。因此无须担心后续的代码对依赖关系产生破坏
- 依赖关系只能在构造器中设定，则只有组件的创建者才能改变组件的依赖关系。对组件的调用者而言，组件内部的依赖关系完全透明，更符合高内聚的原则

3、建议采用以设置注入为主，构造注入为辅的注入策略。对于依赖关系无须变化的注入，尽量采用构造注入；而其他依赖关系的注入，则考虑采用设置注入

7.4. 使用 Spring 容器

1、Spring 有两个核心接口：BeanFactory 和 ApplicationContext，其中 ApplicationContext 是 BeanFactory 的子接口。它们都可以代表 Spring 容器，

Spring 容器是生成 Bean 实例的工厂，并管理容器中的 Bean。在基于 Spring 的 Java EE 应用中，所有的组件都被当成 Bean 处理，包括数据源、Hibernate 的 SessionFactory、事务管理器等

2、应用中所有的组件都处于 Spring 的管理下，都被 Spring 以 Bean 的方式管理，Spring 负责创建 Bean 实例，并管理其生命周期。Spring 里的 Bean 是非常广义的概念，任何的 Java 对象，Java 组件都被当成 Bean 处理。对于 Spring 而言，一切 Java 对象都是 Bean

3、Bean 在 Spring 容器中运行，无须感受 Spring 容器的存在，一样可以接受 Spring 的依赖注入，包括 Bean 成员变量的注入、协作者的注入、依赖关系的注入等

4、Java 程序面向接口编程，无须关心 Bean 实例的实现类；但 Spring 容器负责创建 Bean 实例，因此必须精确知道每个 Bean 实例的实现类，故 Spring 配置文件必须指定 Bean 实例的实现类

7.4.1. Spring 容器

1、Spring 容器最基本的接口就是 BeanFactory。BeanFactory 负责配置、创建、管理 Bean，它有一个子接口：ApplicationContext，因此也被称为 Spring 上下文。Spring 容器还负责管理 Bean 与 Bean 之间的依赖关系

2、BeanFactory 接口包含如下几个基本方法

- boolean containsBean(String name)：判断 Spring 容器是否包含 id 为 name 的 Bean 实例
- <T> T getBean(Class<T> requiredType)：获取 Spring 容器中属于 requiredType 类型的、唯一 Bean 实例
- Object getBean(String name)：返回容器 id 为 name 的 Bean 实例
- <T> T getBean(String name,Class requiredType)：返回容器中 id 为 name，并且类型为 requiredType 的 Bean
- Class<?> getType(String name)：返回容器中 id 为 name 的 Bean 实例的类型

3、BeanFactory 最常用的实现类是 DefaultListableBeanFactory

4、ApplicationContext 是 BeanFactory 的子接口，使用它作为容器更方便。常用实现类有如下几种

- FileSystemXmlApplicationContext
- ClassPathXmlApplicationContext
- AnnotationConfigWebApplication
- 如果在 Web 应用中使用 Spring 容器，通常使用如下两个实现类
- XmlWebApplicationContext
- AnnotationConfigWebApplicationContext

7.4.2. 使用 ApplicationContext

1、大部分使用，都不会使用 BeanFactory 实例作为 Spring 容器，而是使用 ApplicationContext 实例作为容器，因此也把 Spring 容器称为 Spring 上下文

2、ApplicationContext 允许以声明的方式操作容器，无须手动创建，可利用如 ContextLoader 的支持类，在 Web 应用启动时自动创建 ApplicationContext

3、ApplicationContext 除了 BeanFactory 的基本功能外，还有如下额外功能

- ApplicationContext 默认会预初始化所有的 singleton Bean，也可通过配置取消预初始化
- ApplicationContext 继承 MessageSource 接口，因此提供国际化支持
- 资源访问，比如访问 URL 和文件
- 事件机制
- 同时加载多个配置文件
- 以声明方式启动并创建 Spring 容器

4、当系统创建 ApplicationContext 容器时，默认会预初始化所有的 singleton Bean。当 ApplicationContext 容器初始化完成后，容器会自动初始化所有的 singleton Bean，包括调用构造器创建该 Bean 的实例，并根据<property.../>元素执行 setter 方法

- 这意味着：系统前期创建 ApplicationContext 时将有较大的系统开销，但一旦 ApplicationContext 初始化完成，程序后面获取 singleton Bean 实例时将拥有较好的性能
- 为了阻止 Spring 容器预初始化容器中的 singleton Bean，可以为<bean.../>元素指定 lazy-init="true"，该属性用于阻止容器预初始化该 Bean

7.4.3. ApplicationContext 的国际化支持

1、ApplicationContext 接口继承了 MessageSource 接口，因此具有国际化功能，含有如下两个用于国际化的方法

- String getMessage(String code,Object[] args,Locale loc)
- String getMessage(String code,Object[] args,String default,Locale loc)

2、当程序创建 ApplicationContext 容器时，Spring 自动查找配置文件中名为 messageSource 的 Bean 实例

- 一旦找到这个 Bean 实例，上述两个方法的调用就被委托给该 messageSource Bean
- 如果没有该 Bean，ApplicationContext 会查找其父容器中的 messageSource Bean，如果找到它将作为 messageSource Bean 使用
- 如果找不到，则系统会创建一个空的 StaticMessageSource Bean，该 Bean 能接受上述两个方法的调用

7.4.4. ApplicationContext 的事件机制

1、ApplicationContext 的事件机制是观察者设计模式的实现，通过 ApplicationEvent 类和 ApplicationListener 接口，可以实现 ApplicationContext 的事件处理

2、Spring 的事件框架由如下两个重要成员

- ApplicationEvent：容器事件，必须由 ApplicationContext 发布
- ApplicationListener：监听器，可由容器中的任何监听器 Bean 担任

3、Spring 的事件机制与所有的事件机制都基本相似，它们都需要由事件源、事件和事件监听器组成

4、容器事件类

- 需要继承 ApplicationEvent 类，除此之外，它就是一个普通的 Java 类，只

要一个 Java 类继承了 ApplicationEvent 基类，那该对象就可作为 Spring 容器的容器事件

5、容器事件的监听器类

- 必须实现 ApplicationListener 接口，实现该接口必须实现如下方法
  - onApplicationEvent(ApplicationEvent event)：每当容器内发生任何事件时，此方法都被触发

6、当系统创建 Spring 容器、加载 Spring 容器时会自动触发容器事件，容器事件监听器可以监听到这些事件。此外，也可以调用 ApplicationContext 的 pulishEvent()方法来主动触发容器事件

7、Spring 提供如下几个内置事件

- ContextRefreshedEvent：ApplicationContext 容器初始化或刷新触发该事件。此处的初始化是指，所有的 Bean 被成功加载，后处理的 Bean 被检测并激活，所有的 singleon Bean 被预实例化，ApplicationContext 容器已就绪可用
- ContextStartedEvent：当使用 ConfigurableApplicationContext(ApplicationContext 的子接口)接口的 start()方法启动 ApplicationContext 容器时触发该事件。容器管理生命周期的 Bean 实例将获得一个指定的启动信号，这在经常需要停止后重新启动的场合比较常见
- ContextClosedEvent：当使用 ConfigurableApplicationContext(ApplicationContext 的子接口)接口的 close()方法关闭 ApplicationContext 容器时触发该事件
- ContextStoppedEvent：ConfigurableApplicationContext(ApplicationContext 的子接口)接口的 stop()方法使 ApplicationContext 停止触发该事件。此处的"停止"意味着容器管理生命周期的 Bean 实例将获得一个指定的停止信号，被停止的 Spring 容器可再次调用 start()方法重新启动
- RequestHandledEvent：Web 相关的事件，只能应用于使用 DispatcherServlet 的 Web 应用中。在使用 Spring 作为前端的 MVC 控制器时，当 Spring 处理用户请求结束后，系统会自动触发该事件

7.4.5. 让 Bean 获取 Spring 容器

1、前几个示例中，都是程序先创建 Spring 容器，再调用 Spring 容器的 getBean()方法来获取 Spring 容器中的 Bean。在这种访问模式下，程序中总是持有 Spring 容器的引用

2、在 Web 应用汇总，Spring 容器通常采用声明式方式配置产生：开发者只要在 web.xml 文件中配置一个 Listener，该 Listener 将会负责初始化 Spring 容器，前端 MVC 框架可以直接调用 Spring 容器中的 Bean，无需访问 Spring 容器本身。在这种情况下，容器中的 Bean 处于容器管理下，无须主动访问容器，只需接受容器的依赖注入即可

3、在某些特殊情况下，Bean 需要实现某个功能(比如该 Bean 需要输出国际化消息，或者改 Bean 要向 Spring 容器发布时间.....)，但该功能必须借助 Spring 容器才能实现，此时就必须让 Bean 先获取 Spring 容器，然后借助于 Spring 容器来实现该功能

4、为了让 Bean 获取它所在的 Spring 容器，可以让该 Bean 实现 BeanFactoryAware 接口，该接口只有如下方法

setBeanFactory(BeanFactory beanFactory)：该方法有一个参数 beanFactory，该参数指向创建它的 BeanFactory

- 该方法由 Spring 负责调用，将 Spring 容器作为参数传入该方法

5、Spring 容器会检测容器中所有 Bean，如果发现某个 Bean 实现了 ApplicationContextAware 接口，Spring 容器会在创建该 Bean 之后，自动调用该 Bean 的 setApplicationContextAware()方法，调用该方法时，会将容器本身作为参数传给该方法

7.5. Spring 容器中的 Bean

1、从本质上来看，Spring 容器就是一个超级大工厂，Spring 容器中的 Bean 就是该工厂的产品。Spring 容器能产生哪些产品，则完全取决于开发者在配置文件中的配置

2、对于开发者来说，开发者使用 Spring 框架主要做两件事

1) 开发 Bean
2) 配置 Bean

- 对于 Spring 框架来说，它要做的就是根据配置文件来创建 Bean 实例，并调用 Bean 实例的方法完成"依赖注入"，这就是 IoC 的本质

3、**Spring 框架的本质就是**：通过 XML 配置文件来驱动 Java 代码，这样就可以把原本由 Java 代码管理的耦合关系，提取到 XML 配置文件中管理，这就实现了系统中各个组件的解耦，有利于后期的升级和维护

7.5.1. Bean 的基本定义和 Bean 别名

1、<beans.../>元素是 Spring 配置文件的根元素，该元素可以指定如下属性

- default-lazy-init：指定该<beans.../>元素下配置的所有 Bean 默认的延迟初始化行为
- default-merge：指定该<beans.../>元素下配置的所有 Bean 默认的 merge 行为
- default-autowire：指定该<beans.../>元素下配置的所有 Bean 默认的自动装配行为
- default-autowire-candidates：指定该<beans.../>元素下配置的所有 Bean 默认是否作为自动装配的候选 Bean
- default-init-method：指定该<beans.../>元素下配置的所有 Bean 默认的初始化方法
- default-destroy-method：指定该<beans.../>元素下配置的所有 Bean 默认的回收方法

2、<beans.../>元素下所能指定的属性都可以在每个<bean.../子元素中指定，只需要将属性名去掉 default 即可

- 为<bean.../>元素指定这些属性，只对特定 Bean 起作用
- 为<beans.../>元素指定这些属性，将对所有 Bean 起作用
- 当两者冲突时，以<bean.../>为准

3、<bean.../>元素是<beans.../>元素的子元素，<beans.../>元素可以包含多个<bean.../>子元素，每个<bean.../>子元素定义一个 Bean，每个 Bean 对应 Spring 容器里的一个 Java 实例

4、定义 Bean 时，通常需要指定两个属性

1) id：确定该 Bean 的唯一标识，容器对 Bean 的管理、访问以及该 Bean 的依赖关系，都通过该属性完成，Bean 的 id 属性在 Spring 容器中应该唯一
2) class：指定该 Bean 的具体实现类，这里不能是接口。Spring 容器必须知道创建 Bean 的实现类，而不能是接口。在通常情况下，Spring 会直接使用 new 关键字创建该 Bean 实例，因此，这里必须提供 Bean 实现类的类名

➢ id 属性不能包含例如"/"等特殊符号作为属性值，但在某些情况下必须包含，那么可以使用 name 属性，用于指定 Bean 的别名，通过访问 Bean 别名也可以访问到 Bean 实例

5、指定别名有两种方式

1) 定义<bean.../>元素时通过 name 属性指定别名：如果需要为 Bean 实例指定多个别名，则可以在 name 属性中使用逗号、冒号或者空格来分隔多个别名，后面通过任一别名即可访问该 Bean 实例

```
<bean id="person" class="..." name="#abc,@123,abc*"/>
```

2) 通过<alias.../>元素为已有的 Bean 指定别名

```
<alias name="person" alias="jack"/>
```

7.5.2. 容器中 Bean 的作用域

1、当通过 Spring 容器创建一个 Bean 实例时，不仅可以完成 Bean 实例的实例化，还可以为 Bean 指定特定的作用域，Spring 支持如下 5 中作用域

1) singleton：单例模式，在整个 Spring IoC 容器中，singleton 作用域的 Bean 将只生成一个实例
2) prototype：每次通过容器的 getBean()方法获取 prototype 作用域的 Bean 时，都将产生一个新的 Bean 实例
3) request：对于一次 HTTP 请求，request 作用域 Bean 将只生成一个实例，这意味着，在同一次 HTTP 请求内，程序每次请求该 Bean，得到的总是同一个实例。只有在 Web 应用中使用 Spring 时，该作用域才真正有效
4) session：对于一次 HTTP 会话，session 作用域的 Bean 将只生成一个实例，这意味着，在同一次 HTTP 会话内，程序每次请求该 Bean，得到的总是同一个实例。只有在 Web 应用中使用 Spring 时，该作用域才会真正有效
5) global session：每个全局的 HTTP Session 对应一个 Bean 实例，在典型情况下，仅在使用 protletcontext 的时候有效。只有在 Web 应用中使用 Spring 时，该作用域才真正有效

➢ 比较常用的是 singleton 和 prototype 两种作用域

- 对于 singleton 作用域的 Bean，每次请求该 Bean 都将获得相同的实例，容器负责跟踪 Bean 实例的状态，负责维护 Bean 实例的声明周期行为
- 对于 prototype 作用域的 Bean，程序每次请求该 id 的 Bean，Spring 都会新建一个 Bean 实例，然后返回给程序，一旦创建成功，容器就不再跟踪实例，也不会维护 Bean 实例的状态

- 如果不指定 Bean 的作用域，Spring 默认使用 singleton 作用域

2、Spring 配置文件通过 scope 属性指定 Bean 的作用域，该属性可以接受 singleton、prototype、request、session 和 globalSession 五个值

- request 和 session 作用域只在 Web 应用中才有效，并且必须在 Web 应用中增加额外配置才会生效。
- **为了让 request 和 session 两个作用域生效，必须将 HTTP 请求对象绑定到为该请求提供服务的线程上，这使得具有 request 和 session 作用域的 Bean 实例能够在后面的调用链中被访问到**

3、对于支持 Servlet 2.4 以及更新规范的 Web 容器，可以在 Web 应用的 web.xml 文件中增加如下 Listener 配置，该 Listener 负责使 request 作用域生效

```
<listener>
    <listener-class>
      org.springframework.web.context.request.RequestContextListener
    </listener-class>
</listener>
```

7.5.3. 配置依赖

1、Java 应用中各组件相互调用的实质可以归纳为依赖关系，根据注入方式的不同，Bean 的依赖注入通常有如下两种形式

1) 设值注入：通过<property.../>元素驱动 Spring 执行 setter 方法
2) 构造注入：通过<constructor-arg.../>元素驱动 Spring 执行带参数的构造器

2、依赖关系的值要么是一个确定的值，要么是 Spring 容器中其他 Bean 的引用

- 通常不建议使用配置文件管理 Bean 的基本类型的属性值，通常只使用配置文件管理容器中 Bean 与 Bean 之间的依赖关系
- 对于 singleton 作用域的 Bean，如果没有强行取消其预初始化行为，系统会在创建 Spring 容器时预初始化所有的 singleton Bean，与此同时，该 Bean 所依赖的 Bean 也被一起实例化

3、BeanFactory 与 ApplicationContext 实例化容器中的 Bean 的时机不同

- BeanFactory 需要等到程序需要 Bean 实力时才创建 Bean
- ApplicationContext 在容器创建 ApplicationContext 实例时，会预初始化容器中所有的 singleton Bean

4、创建 BeanFactory 时不会立即创建 Bean 实例，所以有可能程序可以正确地创建 BeanFactory 实例，但当请求 Bean 实例时依然抛出一个异常：创建 Bean 实例或注入它的依赖关系时出现错误

- 这样会导致配置错误延迟出现，也会给系统引入不安全的因素
- ApplicationContext 默认预初始化所有 singleton 作用域的 Bean，所以 ApplicationContext 实例化的过程比 BeanFactory 实例化过程时间和内存开销大，但可以在容器初始化阶段就检验出配置错误

5、Spring 把所有的 Java 对象都称为 Bean，因此完全可以把任何 Java 类都部署在 Spring 容器中---只要该 Java 类具有相应的构造器即可。除此之外，Spring 可以为任何 Java 对象注入任何类型的属性---只要该 Java 对象为该属性提供了对应的 setter 方法即可

6、Spring 允许通过如下元素为 setter 方法、构造器参数指定参数值

- value
- ref
- bean
- list、set、map 以及 props

7.5.4. 设置普通属性值

1、<value.../>元素用于指定基本类型及其包装类型、字符串类型的参数值，Spring 使用 XML 解析器来解析出这些数据，然后利用 java.beans.PropertyEditor 完成类型转换：从 java.lang.String 类型转换为所需的参数值类型

```
<property name="integerField" value="1"/>
```

- 早期的 Spring 支持一个更为臃肿的写法，例如需要配置一个驱动类

```
<property name=driverClass" value="com.mysql.jdbc.Driver"/>

<property name="driverClass">
    <value>com.mysql.jdbc.Driver</value>
</property>
```

- 采用<value.../>子元素导致配置文件非常臃肿，而采用 value 属性则更加简洁，于是后来 Spring 推荐采用 value 属性的方式来配置

7.5.5. 配置合作者 Bean

1、如果需要为 Bean 设置的属性值是容器中的另一个 Bean 实例，则应该使用<ref.../>元素，使用<ref.../>元素时可以指定一个 bean 属性，该属性用于引用容器中其他 Bean 实例的 id 属性值

```
<bean id="Chinese" class="...">
    <property name="axe">
        <ref bean="steelAxe"/>
    </property>
</bean>
```

或者更简洁的写法

```
<bean id="Chinese" class="...">
    <property name="axe" ref="steelAxe"/>
</bean>
```

2、<constructor-arg.../>元素也可以增加 ref 属性，从而指定将容器中另一个 Bean 作为构造器参数

7.5.6. 使用自动装配注入合作者 Bean

1、Spring 能自动装配 Bean 与 Bean 之间的依赖关系，即无须使用 ref 显式指定依赖 Bean，而是由 Spring 容器检查 XML 配置文件内容，根据某种规则，为调用者 Bean 注入被依赖的 Bean

2、Spring 的自动装配可通过<beans.../>元素的 default-autowire 属性指定，该属性对配置文件中所有的 Bean 起作用；也可以通过<bean.../>元素的 autowire 属性指定，该属性只对该 Bean 起作用

3、自动装配可以减少配置文件的工作量，但降低了依赖关系的透明性和清晰性

4、autowire、default-autowire 属性可以接受如下值

- no：不使用自动装配。Bean 依赖必须通过 ref 元素定义，这是默认的配

置，在较大的部署环境中不鼓励改变这个配置，显式配合作者能够得到更清晰的依赖关系

- byName：根据 setter 方法名进行自动装配。Spring 容器查找容器中的全部 Bean，找出其 id 与 setter 方法名去掉 set 前缀，并小写首字母后同名的 Bean 来完成注入。如果没有找到匹配的 Bean 实例，则 Spring 不会进行任何注入
- byType：根据 setter 方法的形参类型来自动装配。Spring 容器查找容器中的全部 Bean，如果正好有一个 Bean 类型与 setter 方法的形参类型匹配，就自动注入这个 Bean；如果找到多个这样的 Bean，就抛出异常；如果没有找到，则什么不会发生，setter 方法不会被调用
- constructor：与 byType 类似，区别是用于自动匹配构造器的参数，如果容器不能恰好找到一个与构造器参数类型匹配的 Bean，则会抛出一个异常
- autodetect：Spring 容器根据 Bean 内部结构，自行决定使用 constructor 或 byType 策略。如果找到一个默认的构造函数，那么就会应用 byType 策略

7.5.6.1. byName 规则

1、byName 规则是指 setter 方法的方法名与 Bean 的 id 进行匹配，假设 Bean A 的实现类包含 setB()方法，而 Spring 配置文件恰好包含 id 为 b 的 Bean，则 Spring 容器会将 b 实例注入 Bean A 中，如果容器中没有名字匹配的 Bean，Spring 则不会做任何事情

7.5.6.2. byType 规则

1、byType 规则是指根据 setter 方法的参数类型与 Bean 的类型进行匹配，假如 A 实例有 setB(B b)方法，而 Spring 配置文件中恰好有一个类型为 B 的 Bean 实例，容器为 A 注入类型匹配的实例，如果容器中没有类型为 B 的实例，Spring 不会调用 setB()方法；但如果容器中包含多于一个的 B 实例，程序将会抛出异常

7.5.7. 注入嵌套 Bean

1、如果某个 Bean 所依赖的 Bean 不想被 Spring 容器直接访问，则可以使用嵌套 Bean

2、把<bean.../>配置成<property.../>或<constructor-args.../>的子元素，那么该<bean.../>元素配置的 Bean 仅仅作为 setter 注入、构造注入的参数，这种 Bean 就是嵌套 Bean。由于容器不能获取嵌套 Bean，因此它不需要指定 id 属性

3、嵌套 Bean 提高了程序的内聚性，但降低了程序的灵活性。只有在完全确定无须通过 Spring 容器访问某个 Bean 实例时，才考虑使用嵌套 Bean 来配置该 Bean

4、使用嵌套 Bean 与使用 ref 引用容器中另一个 Bean 在本质上是一样的

5、Spring 框架的本质就是通过 XML 配置文件来驱动 Java 代码，当程序要调用 setter 方法或有参数的构造器时，程序总需要传入参数值，随参数类型的不同，Spring 配置文件当然也要随之改变

- 形参类型是基本类型、String、日期等，直接使用 value 指定字面值即可

- 形参类型是复合类型，需要传入一个 Java 对象作为实参
  - 使用 ref 引用一个容器中已经配置的 Bean(Java 对象)
  - 使用<bean.../>元素配置一个嵌套 Bean(Java 对象)

7.5.8. 注入集合值

1、如果需要调用形参类型为集合的 setter 方法，或调用形参类型为集合的构造器，则可以使用集合元素<list.../>、<set.../>、<map.../>和<props.../>分别来设置类型为 List、Set、Map 和 Properties 的集合参数值

2、Spring 对 List 集合和数组的处理是一样的，都用<list.../>元素来配置

3、当使用<list.../>、<set.../>、<map.../>等元素配置集合类型的参数值时，还需要配置集合元素。由于集合元素又可以是基本类型值、引用容器中的其他 Bean、嵌套 Bean 或集合属性等，因此<list.../>、<set.../>、<map.../>元素可以接受如下子元素

- value：指定集合元素是基本数据类型或字符串类型
- ref：指定集合元素是容器中另一个 Bean 实例
- bean：指定集合元素是一个嵌套的 Bean
- list、set、map、props：指定集合元素又是集合

4、<props.../>元素用于配置 Properties 类型的参数值，Properties 类型是一种特殊的类型，其 key 和 value 都只能是字符串，故 Spring 配置 Properties 类型的参数值比较简单：每个 key-value 对只要分别给出 key 和 value 就足够了---而且 key 和 value 都是字符串类型

```
<prop key="血压">正常</prop>
```

5、<map.../>元素配置 Map 参数值比较复杂，因为 Map 集合的每个元素由 key、value 两个部分组成，所以配置文件中的每个<entry.../>配置一组 key-value 对，其中<entry.../>元素支持如下 4 个属性

- key：如果 Map key 是基本类型或字符串，则可使用该属性来指定 Map key
- key-ref：如果 Map key 是容器中的另一个 Bean 实例，则可使用该属性指定容器中其他 Bean 的 id
- value：如果 Map value 是基本类型值或字符串，则可使用该属性来指定 Map value
- value-ref：如果 Map value 是容器中另一个 Bean 实例，则可使用该属性指定容器中其他 Bean 的 id

6、从 JDK 1.5 以后，Java 可以使用泛型指定集合元素的类型，则 Spring 可通过反射来获取集合元素的类型，这样 Spring 的类型转换器也会起作用了

7.5.9. 组合属性

1、Spring 还支持组合属性的方式，例如使用配置文件为形如 foo.bar.name 的属性设置参数值。为 Bean 的组合属性设置参数值时，除最后一个属性之外，其他属性值都不允许为 null

```
<bean id="exampleBean" class="...">
    <property name="person.name" value="孙悟空"/>
</bean>
```

- 先调用 getter 方法然后再调用 setter 方法
- 组合属性只有最后一个属性才调用 setter 方法，前面各属性实际上对应于调用 getter 方法---这也是前面属性不能为 null 的缘故

7.5.10. Spring 的 Bean 和 JavaBean

1、Spring 容器对 Bean 没有特殊要求，甚至不要求该 Bean 像标准的 **JavaBean---必须为每个属性提供对应的 setter 和 getter 方法**。Spring 中的 Bean 是 Java 实例、Java 组件；而传统 Java 应用中的 JavaBean 通常作为 DTO(数据传输对象)，用来封装值对象，在各层之间传递数据

2、Spring 中的 Bean 比 JavaBean 的功能要复杂，用法也更丰富。传统的 JavaBean 可作为 Spring 的 Bean，从而接受 Spring 管理

3、虽然 Spring 对 Bean 没有特殊要求，但依然建议 Spring 中的 Bean 应该满足如下几个原则

1) 尽量为每个 Bean 实现类提供无参数的构造器
2) 接受构造注入的 Bean，则应提供对应的、带参数的构造函数
3) 接受设值注入的 Bean，则应该提供对应的 setter 方法，并不要求提供对应的 getter 方法

4、传统的 JavaBean 和 Spring 中的 Bean 存在如下区别

- 用处不同：传统的 JavaBean 更多是作为值对象传递参数；Spring 的 Bean 用户几乎无所不包，任何应用组件都被称为 Bean
- 写法不同：传统的 JavaBean 作为值对象，要求每个属性都提供 getter 和 setter 方法；但 Spring 只需为接受设值注入的属性提供 setter 方法即可
- 生命周期不同：传统的 JavaBean 作为值对象传递，不接受任何容器管理其生命周期；Spring 中的 Bean 由 Spring 管理其生命周期

7.6. Spring3.0 提供的 Java 配置管理

1、Spring3.0 为不喜欢 XML 的人提供了一种选择：如果不喜欢使用 XML 来管理 Bean，以及 Bean 之间的依赖关系，Spring 允许开发者使用 Java 进行配置管理

2、常用 Annotation

- @Configuration：用于修饰一个 Java 配置类
- @Bean：用于修饰一个方法，将该方法的返回值定义成容器中的一个 Bean
- @Value：用于修饰一个 Field，用于为该 Field 配置一个值，相当于配置一个变量

3、一旦使用了 Java 配置类来管理 Spring 容器中的 Bean 以及其依赖关系，此时就需要使用如下方法来创建 Spring 容器

```
ApplicationContext ctx=
        new AnnotationConfigApplicationContext(AppConfig.class)
```

- 获得 Spring 容器之后，接下来利用 Spring 容器获取 Bean 实例、调用 Bean 方法就没有任何特别之处

4、使用 Java 配置类时，还有如下常用 Annotation

- @Import：修饰一个 Java 配置类，用于向当前 Java 配置类中导入其他 Java 配置类

- @Scope：用于修饰一个方法，指定该方法对应的 Bean 的生命域
- @Lazy：用于修饰一个方法，指定该方法对应的 Bean 是否需要延迟初始化
- @DependsOn：用于修饰一个方法，指定在初始化该方法对应的 Bean 之前初始化指定的 Bean

5、就普通用户习惯来看，还是使用 XML 配置文件管理 Bean 及其依赖关系更为方便---毕竟使用 XML 文件来管理 Bean 及其依赖关系是为了解耦。但这种 Java 配置类的方法又退回到代码耦合层次，只是将这种耦合集中到一个或多个 Java 配置类当中

6、XML 配置文件与 Java 配置类共存的原因

1) 目前绝大多数采用 Spring 进行开发的项目，几乎都是基于 XML 配置方式的，Spring 在引入注解的同时，必须保证注解能够与 XML 和谐共存，这是前提
2) 由于注解引入较晚，因此功能也没有发展多年的 XML 强大，对于复杂的配置，注解还很难独当一面，在一段时间内仍需要 XML 的配合才能解决问题
3) Spring 的 Bean 的配置方式与 Spring 核心模块之间是解耦的，因此改变配置方式对 Spring 的框架自身是透明的

7、如果以 XML 配置为主，就需要让 XML 配置能加载 Java 类配置，设置方式详见 P573

8、如果以 Java 类配置为主，就需要让 Java 配置类能加载 XML 配置，需要借助 @ImportResource 注解，详见 P573

7.7. 创建 Bean 的 3 种方式

1、在大多数情况下，Spring 容器直接通过 new 关键字调用构造器来创建 Bean 实例，而 class 属性指定了 Bean 实例的实现类。因此<bean.../>元素必须指定 Bean 实例的 class 属性，但这并不是实例化 Bean 的唯一方法

2、Spring 支持使用如下方式来创建 Bean

- 调用构造器创建 Bean
- 调用静态工厂方法创建 Bean
- 调用实例工厂方法创建 Bean

7.7.1. 使用构造器创建 Bean 实例

1、使用构造器来创建 Bean 实例是最常见的情况，如果不采用构造注入，Spring 底层会调用 Bean 类的无参数构造器来创建实例，因此要求该 Bean 类提供无参数的构造器。在这种情况下，class 元素是必须的

2、如果不采用构造注入，Spring 容器使用默认的构造器来创建 Bean 实例

- Spring 对 Bean 实例的所有属性执行默认初始化，即所有基本类型的值为 0 或 false，所有引用类型初始化为 null
- 接下来，BeanFactory 会根据配置文件决定依赖关系，先实例化被依赖的 Bean 实例，然后为 Bean 注入依赖关系，最后将一个完整的 Bean 实例返回给程序

3、如果采用构造注入，则要求配置文件为<bean.../>元素添加<constructor-

arg.../>元素

- 每个<constructor-arg.../>子元素配置一个构造器参数
- Spring 容器将使用带对应参数的构造器来创建 Bean 实例，Spring 调用构造器传入的参数即可用于初始化 Bean 实例变量，最后也将一个完整的 Bean 实例返回给程序

7.7.2. 使用静态工程方法创建 Bean

1、使用静态工厂方法创建 Bean 实例时，class 属性也必须指定，但此时 class 属性并不是指定 Bean 实例，而是静态工厂类，Spring 通过哪个属性知道由哪个工厂类来创建 Bean 实例

2、还需要使用 factory-method 属性来指定静态工厂方法，Spring 将调用静态工厂方法(可能包含一组参数)返回一个 Bean 实例，一旦获得了指定 Bean 实例，Spring 后面的处理步骤与采用普通方法创建 Bean 实例完全一样

3、采用静态工厂方法创建 Bean 实例时，<bean.../>元素需要指定如下两个属性

1) class：该属性的值为静态工厂类的类名
2) factory-method：该属性指定静态工厂方法来生产 Bean 实例

4、使用静态工厂方法创建实例时必须提供工厂类，工厂类包含生产实例的静态工厂方法。通过静态工厂方法创建实例时需要对配置文件进行如下改变

1) class 属性的值不再是 Bean 实例的实现类，而是生成 Bean 实例的静态工厂类
2) 使用 factory-method 属性指定创建 Bean 实例的静态工厂方法
3) 如果静态工厂方法需要参数，则使用<constructor-arg.../>元素指定静态工厂方法的参数

7.7.3. 调用实例工厂方法创建 Bean

1、实例工厂方法与静态工厂方法只有一点不同：调用静态工厂方法只需要工厂类即可，而调用实例工厂方法则需要工厂实例。因此配置实例工厂方法与配置静态工厂方法基本类似，只有一点区别：配置静态工厂方法使用 class 指定静态工厂类，而配置实例工厂方法则使用 factory-bean 指定工厂实例

2、使用实例工厂方法时，配置 Bean 实例的<bean.../>元素无须 class 属性，因为 Spring 容器不再直接实例化该 Bean，Spring 容器仅仅调用实例工厂的工厂方法，工厂方法负责创建 Bean 实例

3、采用实例工厂方法创建 Bean 的<bean.../>元素时需要指定如下两个属性

1) factory-bean：该属性的值为工厂的 Bean 的 id
2) factory-method：该属性的值指定实例工厂的工厂方法

- 与静态工厂方法相似，如果需要在调用实例工厂方法时传入参数，则使用<constructor-arg.../>元素确定参数值

4、调用实例工厂方法创建 Bean，与调用静态工厂方法创建 Bean 的用法基本相似

- 区别如下

1) 配置实例工厂方法创建 Bean，必须将实例工厂配置成 Bean 实例；而配置静态工厂方法创建 Bean，则无须配置工厂 Bean
2) 配置实例工厂方法创建 Bean，必须使用 factory-bean 属性确定工厂

Bean；而配置静态工厂方法创建 Bean，则使用 class 元素确定静态工厂类

- 相同之处如下
  1) 都需要使用 factory-method 属性指定产生 Bean 实例的工厂方法
  2) 工厂方法如果需要参数，都使用<constructor-arg.../>元素指定参数值
  3) 普通的设值注入，都使用<property.../>元素确定参数值

7.8. 深入理解容器中的 Bean

1、Spring 框架绝大部分的工作都集中在对容器的 Bean 的管理上，包括管理容器中的 Bean 的生命周期，使用 Bean 继承等特殊功能

7.8.1. 抽象 Bean 与子 Bean

1、在实际开发中，随着项目越来越大，Spring 配置文件出现多个<bean.../>配置具有大致相同的配置信息，只有少量信息不同，这样导致配置文件出现很多重复的内容。如果保留这种配置，则可能导致的问题是

- 配置文件臃肿
- 后期难以修改、维护

2、为了解决该问题，可以考虑把多个<bean.../>配置中相同的信息提取出来，集中成配置模板---这个配置模板并不是真正的 Bean，因此 Spring 不应该创建该配置模板，于是需要为该<bean.../>配置增加 abstract="true"---这就是抽象 Bean

- 抽象 Bean 不能被实例化，Spring 容器不会创建抽象 Bean 实例，抽象 Bean 的价值在于被继承，抽象 Bean 通常作为父 Bean 被继承
- 抽象 Bean 只是配置信息的模板，指定 abstract="true"即可阻止 Spring 实例化该 Bean，因此抽象 Bean 可以不指定 class 属性

3、将大部分相同信息配置成抽象 Bean 之后，将实际的 Bean 实例配置成该抽象 Bean 的子 Bean 即可。子 Bean 定义可以从父 Bean 继承实现类、构造器参数、属性值等配置信息，除此之外，子 Bean 配置可以增加新的配置信息，并可以指定新的配置信息覆盖父 Bean 的定义

4、通过为一个<bean.../>元素指定 parent 属性即可指定该 Bean 是一个子 Bean，parent 属性指定该 Bean 所继承的父 Bean 的 id

5、子 Bean 无法继承如下属性

- depends-on
- autowire
- singleton
- scope
- lazy-init

6、子 Bean 可以覆盖父 Bean 的配置信息：当子 Bean 拥有和父 Bean 相同的配置信息时，以子 Bean 为准

7.8.2. Bean 继承与 Java 继承的区别

1、Spring 中的 Bean 继承与 Java 中的继承截然不同。前者是实例与实例之间的参数值的延续，后者则是一般到特殊的细化；前者是对象与对象之间的关系，后者则是类与类之间的关系

2、区别如下

- Spring 中的 Bean 和父 Bean 可以是不同类型，但 Java 中的继承则可保证子类是一种特殊的父类
- Spring 中的 Bean 的继承是实例之间的关系，因此主要表现为参数值的延续；而 Java 中的继承是类之间的关系，主要表现为方法、属性的延续
- Spring 中的子 Bean 可不作为父 Bean 使用，不具备多态性；Java 中的子类实例完全可以当成父类实例使用

7.8.3. 容器中的工厂 Bean

1、此处的工厂 Bean，与前面介绍的实例工厂方法创建 Bean，或者静态工厂方法创建 Bean 的工厂有所区别：

- 前者的工厂是标准的工厂模式，Spring 只是负责调用工厂方法来创建 Bean 实例
- 此处的工厂 Bean 是 Spring 中的特殊 Bean，这种工厂 Bean 必须实现 FactoryBean 接口

2、FactoryBean 接口是工厂 Bean 的标准接口，把工厂 Bean(实现 FactoryBean 接口的 Bean)部署在容器中之后，如果程序通过 getBean()方法来获取它时，容器返回的不是 FactoryBean 实现类的实例，而是返回 FactoryBean 的产品(即该工厂 Bean 的 getObject()方法的返回值)

3、FactoryBean 接口提供如下三个方法

- T getObject()：实现该方法负责返回工厂 Bean 生成的 Java 实例
- Class<?> getObjectType()：实现该方法返回该工厂 Bean 生成的 Java 实例的实现类
- boolean isSingleton()：实现该方法表示该工厂 Bean 生成的 Java 实例是否为单例模式

4、配置 FactoryBean 与配置普通 Bean 的定义没有区别，但当程序向 Spring 容器请求获取该 Bean 时，容器返回该 FactoryBean 的产品，而不是返回该 FactoryBean 本身

5、实现 FactoryBean 接口的最大作用在于：Spring 容器并不是简单地返回该 Bean 的实例，而是返回该 Bean 实例的 getObject()方法的返回值，而 getObject()方法则由开发者负责实现，这样开发者希望 Spring 返回什么，只要按规律重写 getObject()方法即可

6、部署工厂 Bean 与部署普通 Bean 其实没有任何区别，同样只需要为该 Bean 配置 id、class 两个属性即可，但 Spring 对 FactoryBean 接口的实现类的处理有所不同

7、当程序需要获得 FactoryBean 本身时，并不直接请求 Bean id，而是在 Bean id 前面加&符号，容器则返回 FactoryBean 本身，而不是其产品 Bean

7.8.4. 获得 Bean 本身的 id

1、对于实际的 Java 应用而言，Bean 与 Bean 之间的关系是通过依赖注入管理的，通常不会通过调用容器的 getBean()方法来获取 Bean 实例

2、如果想要通过 Bean 获取其 id，可以借助 Spring 提供的 BeanNameAware 接口，通过该接口，该接口提供了一个方法：setBeanName(String name)，该方法的 name 参数就是 Bean 的 id，实现该方法的 Bean 类就可通过该方法来获得部署该

Bean 的 id 了

- 该方法不是由程序员来调用的，而是由 Spring 容器负责调用，当 Spring 容器调用这个 setter 方法时，会把部署该 Bean 的 id 属性值作为参数传入

7.8.5. 强制初始化 Bean

1、在大多数情况下，Bean 之间的依赖非常直接，Spring 容器在返回 Bean 实例之前，先要完成 Bean 依赖关系注入。例如 Bean A 依赖于 Bean B，程序请求 Bean A 时，Spring 容器会自动先初始化 Bean B，再将 Bean B 注入 Bean A，最后将具备完整依赖的 Bean A 返回给程序

2、在极端情况下，Bean 之间的依赖不够直接，例如某个类的初始化块中使用其他 Bean，Spring 总是先初始化主调 Bean，当执行块初始化时，被依赖的 Bean 可能还没实例化，此时将引发异常

3、为了显式指定被依赖的 Bean 在目标 Bean 之前初始化，可以使用 depends-on 属性，该属性可以在初始化主调 Bean 之前，强制初始化一个或多个 Bean

7.9. 容器中 Bean 的生命周期

1、Spring 可以管理 singleton 作用域的 Bean 的生命周期，Spring 可以精确地知道该 Bean 何时被创建、何时被初始化完成、容器何时准备销毁该 Bean 实例

2、对于 prototype 作用域的 Bean，Spring 容器仅仅负责创建，当容器创建了 Bean 实例之后，Bean 实例完全交给客户端代码管理，容器不再跟踪其声明周期。每次客户端请求 prototype 作用域的 Bean 时，Spring 都会产生一个新的实例，Spring 容器无从知道它曾经创建了多少个 prototype 作用域的 Bean，也无从知道这些 prototype 作用域的 Bean 什么时候才会被销毁。因此 Spring 无法管理 prototype 作用域的 Bean

3、对于 singleton 作用域的 Bean，客户端代码请求时返回同一个共享实例，客户端代码不能控制 Bean 的销毁，Spring 容器负责跟踪 Bean 实例的产生、销毁。Spring 容器可以在创建 Bean 之后，进行某些通用资源申请，还可以在销毁 Bean 实例之前，先回收某些资源，比如数据库连接

4、管理 Bean 的声明周期行为主要有两个时机

- 注入依赖关系之后
- 即将销毁 Bean 之前

7.9.1. 依赖关系注入之后的行为

1、Spring 提供两种方式在 Bean 全部属性设置成功后执行特定行为

- 使用 init-method 属性：使用该属性指定某个方法在 Bean 全部依赖关系设置结束后自动执行，使用这种方法不需要将代码与 Spring 的接口耦合在一起，代码污染小
- 实现 InitializingBean 接口：让 Bean 实现 InitializingBean 接口，该接口提供一个方法：void afterPropertiesSet() throws Exception;Spring 容器会在为该 Bean 注入依赖关系之后，调用该 Bean 所实现的 afterPropertiesSet()方法

2、如果既采用 init-method 属性指定初始化方法，又实现 InitializingBean 接口的方法来指定初始化方法，Spring 容器会执行两个初始化方法：先执行 InitializingBean 接口中定义的方法，然后执行 init-method 属性指定的方法

7.9.2. Bean 销毁之前的行为

1、与定制初始化行为相似，Spring 也提供两种方式定制 Bean 实例销毁之前的特定行为，这两种方式如下

- 使用 destroy-method 属性：使用 destroy-method 属性指定某个方法在 Bean 销毁之前被自动执行。使用这种方式，不需要将代码与 Spring 接口耦合在一起，代码污染小
- 实现 DisposableBean 接口：让 Bean 实现 DisposableBean 接口，该接口提供一个方法：void destroy() throws Excepiton;该方法就是 Bean 实例销毁之前执行的方法

3、如果既采用 destory-method 属性指定销毁之前的方法，又采用实现 DisposalbeBean 接口来指定销毁之前的方法，Spring 容器会执行两个方法：先执行 DisposableBean 接口中定义的方法，然后执行 destroy-method 属性指定的方法

4、如果容器中很多 Bean 都需要指定特定的生命周期行为，则可以利用 <beans.../>元素的 default-init-method 属性和 default-destroy-method 属性，这两个属性的作用域类似于<bean.../>元素的 init-method 和 destory-method 属性的作用，区别是 default-init-method 属性和 default-destroy-method 属性是属于 <beans.../>元素的，将对容器中所有 Bean 生效

7.9.3. 协调作用域不同步的 Bean

1、当两个 singleton 作用域的 Bean 存在依赖关系时，或者当 prototype 作用域的 Bean 依赖 singleton 作用域的 Bean 时，使用 Spring 提供的依赖注入进行管理即可

2、问题引入：singleton 作用域的 Bean 只有一次初始化的机会，它的依赖关系也只在初始化阶段被设置，当 singleton 作用域的 Bean 依赖 prototype 作用域的 Bean 时，Spring 容器会在初始化 singleton 作用域的 Bean 之前，先创建被依赖的 prototype Bean，然后才初始化 singleton Bean，并将 prototype Bean 注入 singleton Bean，这会导致以后无论何时通过 singleton Bean 去访问 prototype Bean，得到的永远是最初哪个 prototype Bean---这样就相当于 singleton Bean 把它所依赖的 prototype Bean 变成了 singleton 行为

3、当 singleton 作用域的 Bean 依赖于 prototype 作用域的 Bean 时，会产生不同步的现象，解决该问题有两种思路

1) 放弃依赖注入：singleton 作用域的 Bean 每次需要 prototype 作用域的 Bean 时，主动向容器请求新的 Bean 实例，即可保证每次注入的 prototype Bean 实例都是最新的实例
2) 利用方法注入

- 第一种方式显然不是一个好的做法，代码主动请求新的 Bean 实例，必然导致程序代码与 Spring API 耦合，造成代码污染
- 通常采用第二种做法，使用方法注入

4、方法注入通常使用 lookup 方法注入，使用 lookup 方法注入可以让 Spring 容器重写容器中 Bean 的抽象或具体方法，返回查找容器中其他 Bean 的结果，被查找的 Bean 通常是一个 non-singleton Bean(尽管也可以是一个 singleton 的)。Spring 通过 JDK 动态代理或者 cglib 库修改客户端的二进制代码，从而实现上述要求

5、为了使用 lookup 方法注入，大致需要如下两步

1) 将调用者 Bean 的实现类定义为抽象类，并定义一个抽象方法来获取被依赖的 Bean
2) 在<bean.../>元素中添加<lookup-method.../>子元素让 Spring 为调用者 Bean 的实现类实现指定的抽象方法

6、在配置文件中为<bean.../>元素添加<lookup-method.../>子元素，<lookup-method.../>子元素告诉 Spring 需要实现哪个抽象方法。Spring 为抽象方法提供实现提之后，这个方法就会变成具体方法，这个类也就变成了具体类，接下来 Spring 就可以创建该 Bean 的实例了

7、使用<lookup-method.../>元素需要指定如下两个属性

- name：指定需要让 Spring 实现的方法
- bean：指定 Spring 实现该方法的返回值

8、通常情况下，Java 类里所有方法都应该由程序员来负责实现，但在某些情况下，系统可以实现一些极其简单的方法，例如<lookup-method.../>元素的 name 属性所指定的方法

9、Spring 会采用运行时动态增强的方式来实现<lookup-method.../>元素所指定的抽象方法，如果目标抽象类实现过接口，Spring 会采用 JDK 动态代理来实现该抽象类，并为之实现抽象方法；如果目标抽象类没有实现过接口，Spring 会采用 cglib 实现抽象类，并为之实现抽象方法

10、要保证 lookup 方法注入每次产生新的 Bean 实例，必须将目标 Bean 部署成 prototpye 作用域，否则，如果容器中只有一个被依赖的 Bean 实例，即使采用 lookup 方法注入，每次也依然返回同一个 Bean 实例

7.10. 高级依赖关系配置

1、Spring 允许将 Bean 实例的所有成员变量，甚至基本类型的成员变量都通过配置文件来指定值，这种方式提供了很好的解耦，但是大大降低了程序的可读性(必须同时参照配置文件才可知道程序中成员变量的值)，因此，滥用依赖注入也会引起一些问题

2、通常的建议是，组件与组件之间的耦合，采用依赖注入管理；但基本类型的成员变量值，应直接在代码中设置。对于组件之间的耦合关系，通过使用控制反转，代码变得非常清晰。因此 Bean 无须管理依赖关系，而是由容器提供注入，Bean 无须知道这些实例在哪里，以及它们具体的实现

3、在实际的应用中，某个 Bean 实例的属性值可能是某个方法的返回值，或者是类的 Field 值，或者另一个对象的 getter 方法返回值，Spring 同样可以支持这种非常规的注入方式。Spring 甚至支持将任意方法的返回值、类或对象的 Field 值、其他 Bean 的 getter 方法返回值，直接定义成容器中的一个 Bean

4、Spring 框架的本质是：开发者在 Spring 配置文件中使用 XML 元素进行配置，实际驱动 Spring 执行相应的代码

1) 使用<bean.../>元素，实际启动 Spring 执行无参或有参构造器，或者调用工厂方法创建 Bean
2) 使用<property.../>元素，实际驱动 Spring 执行一次 setter 方法

- Java 程序还可能有其他类型的语句，如调用 getter 方法、调用普通方法、访问类或对象的 Field，而 Spring 也为这种语句提供了对应的配置语法

3) 调用 getter 方法：使用 PropertyPathFactoryBean
4) 访问类或对象的 Field 值：使用 FieldRetrievingFactoryBean
5) 调用普通方法：使用 MethodInvokingFactoryBean

- 从另一个角度来看 Spring 框架：它可以让开发者无须书写 Java 代码就可进行 Java 编程，当开发者 XML 采用合适语法进行配置之后，Spring 就可以通过反射在底层执行任意的 Java 代码

7.10.1. 获取其他 Bean 的属性值

1、PropertyPathFactoryBean 用来获取目标 Bean 的属性值(实际上就是它的 getter 方法的返回值)，获得的值可注入给其他 Bean，也可直接定义成新的 Bean
2、使用 PropertyPathFactoryBean 来调用其他 Bean 的 getter 方法需要指定如下信息

- 调用哪个对象：setTargetObject(Object targetObject)
- 调用哪个 setter 方法：setPropertyPath(String propertyPath)

3、示例详见 P594
4、<util:property-path.../>元素可作为 PropertyPathFactoryBean 的简化配置，使用该元素时可指定如下两个属性

- id：该属性指定将 getter 方法的返回值定义成名为 id 的 Bean 实例
- path：该属性指定将哪个 Bean 实例、哪个属性(支持复合属性)暴露出来

5、配置 PropertyPathFactoryBean 工厂 Bean 时指定的 id 属性，并不是该 Bean 的唯一标识，而是用于指定属性表达式的值，例如

```
<bean id="son2" class="org.crazyit.app.service.Son">
    <property name="age">
        <!-- 使用嵌套 Bean 为调用 setAge()方法指定参数值 -->
        <!-- 以下是访问指定 Bean 的 getter 方法的简单方式,
        person.son.age 代表获取 person.getSon().getAge()-->
        <bean id="person.son.age" class=
        "org.springframework.beans.factory.config.PropertyPathFactoryBean"/>
    </property>
</bean>
```

7.10.2. 获取 Field 值

1、通过 FieldRetrievingFactoryBean 类，可访问类的静态 Field 或对象的实例 Field 值。获得指定的 Field 的值之后，即可将获得的值注入其他 Bean，也可直接定义成新的 Bean
2、使用 FieldRetrievingFactoryBean 访问 Field 的值可分为两种情形

1) 如果要访问的 Field 是静态 Field，则需要指定

- 调用哪个类：setTargetClass(String targetClass)

- 访问哪个 Field：setTargetField(String targetField)

2) 如果要访问的 Field 是实例 Field，则需要指定
   - setTargetObject(Object targetObject)
   - setTargetField(String targetField)

➢ 第二种用法，在实际中几乎没有多大用处，原因是根据良好的封装原则，Java 类的实例 Field 应该用 private 修饰，并使用 getter 和 setter 来访问和修改，但 FieldRetrievingFactoryBean 则要求实例 Field 以 public 修饰

3、<util:constant.../>元素可作为 FieldRetrievingFactoryBean 访问静态 Field 的简化配置，使用该元素时可指定如下两个属性

1) 该属性指定将静态 Field 的值定义成名为 id 的 Bean 实例
2) static-field：该属性指定访问哪个类的哪个静态 Field

7.10.3. 获取方法返回值

1、通过 MethodInvokingFactoryBean 工厂 Bean，可调用任意类的类方法，也可调用任意对象的实例方法，如果调用的方法有返回值，则既可将该指指定方法的返回值定义成容器中的 Bean，也可将指定方法的返回值注入给其他 Bean

2、使用 MethodInvokingFactoryBean 来调用任意方法时，可以分为两种情形

1) 如果希望调用的方法是静态方法
   - 调用哪个类：setTargetClass(String targetClass)
   - 调用哪个方法：setTargetMethod(String targetMethod)
   - 调用方法的参数：setArguments(Object[] arguments)
2) 如果希望调用的方法是实例方法
   - 调用哪个对象：setTargetObject(Object targetObject)
   - 调用哪个方法：setTargetMethod(String targetMethod)
   - 调用方法的参数：setArguments(Object[] arguments)

3、示例详见 P598

4、总结

➢ 调用构造器创建对象(包括使用工厂方法创建对象)，用<bean.../>元素
➢ 调用 setter 方法，用<property.../>元素
➢ 调用 getter 方法，用 PropertyPathFactoryBean 或<util:property-path.../>元素
➢ 调用普通方法，用 MethodInvokingFactoryBean 工厂 Bean
➢ 获取 Field，用 FieldRetrievingFactoryBean 或<util:constant.../>元素

5、一般来说，应该讲如下两类信息放到 XML 配置文件中管理

1) 项目升级、维护时经常需要改动的信息
2) 控制项目内各组件耦合关系的代码

➢ 这样就体现了 Spring IoC 容器的作用：将原来使用 Java 代码管理的耦合关系，提取到 XML 中进行管理，从而降低了各组件之间的耦合，提高了软件系统的可维护性

7.11. 基于 XML Schema 的简化配置方式

1、从 Spring 2.0 开始，Spring 允许使用基于 XML Schema 的配置方式来简化 Spring 配置文件

2、早期 Spring 用一种<bean.../>元素即可配置所有 Bean 实例，而每个设值注入再用一个<property.../>元素即可。这种配置方式简单、直观，而且能以相同风格处理所有 Bean 的配置---唯一的缺点是配置繁琐，当 Bean 实例的属性足够多，且属性类型复杂时，基于 DTD 的配置文件将变得更加繁琐

7.11.1. 使用 p:命名空间简化配置

1、p:命名空间甚至不需要特定的 Schema 定义，它直接存在于 Spring 内核中，u 采用<property.../>元素定义 Bean 的属性不同的是，当导入 p:命名空间之后，就可直接在<bean.../>元素中使用属性来驱动执行 setter 方法

```
<bean id="chinese" class="..." p:age="29" p:axe-ref="stoneAxe"/>
```

- 因为 axe 设值注入的参数需要引用容器中另一个已存在的 Bean 实例，故在 axe 后增加了"-ref"后缀，这个后缀指定该值不是一个具体的值，而是对另外一个 Bean 的引用
- 使用 p:命名空间没有标准的 XML 格式灵活，如果某个 Bean 的属性名以"-ref"结尾，那么采用 p:命名空间定义时就会出现冲突，而采用标准的 XML 格式定义则不会出现这种问题

7.11.2. 使用 c:命名空间简化配置

1、p:命名空间主要用于简化设值注入，而 c:命名空间则用于简化构造注入

```
<bean id="chinese" class="..."
c:axe-ref="steelAxe" c:age="29"/>
```

- 使用 c:指定构造器参数的格式为：c:构造器参数名="值"或 c:构造器参数名-ref="其他 Bean 的 id"

```
<bean id="chinese" class="..."
c:_0-ref="steelAxe" c:_1="29"/>
```

- 或者使用索引来配置构造器参数的方式，c:_N 中的 N 代表第几个构造器参数

7.11.3. 使用 util:命名空间简化配置

1、在 Spring 框架解压缩包的 schema\util\路径下包含有 util:命名空间的 XML Schema 文件，为了使用 util:命名空间的元素，必须先在 Spring 配置文件中导入最新的 spring-util-4.0.xsd，需要增加一端配置片段，详见 P602

2、util Schema 提供了如下几个元素

- constant：该元素用于获取指定类的静态 Field 值，它是 FieldRetrievingFactoryBean 的简化配置
- property-path：该元素用于获取指定对象的 getter 方法的返回值，它是 PropertyPathFactoryBean 的简化配置
- list：该元素用于定义一个 List Bean，支持使用<value.../>、<ref.../>、<bean.../>等子元素来定义 List 集合元素，支持以下三个属性
  - id：该属性指定定义一个名为 id 的 list Bean 实例
  - list-class：该属性指定 Spring 使用哪个 List 实现类来创建 Bean 实例，默认使用 ArrayList 作为实现类

- scope：指定该 List Bean 实例的作用域

- set：该元素用于定义一个 Set Bean，支持使用<value.../>、<ref.../>、<bean.../>等子元素来定义 Set 集合元素，支持以下三个属性
  - id：该属性指定定义一个名为 id 的 Set Bean 实例
  - set-class：该属性指定 Spring 使用哪个 Set 实现类来创建 Bean 实例，默认使用 HashSet 作为实现类
  - scope：指定该 Set Bean 实例的作用域
- map
  - id：该属性指定定义一个名为 id 的 Map Bean 实例
  - map-class：该属性指定 Spring 使用哪个 Map 实现类来创建 Bean 实例，默认使用 HashMap 作为实现类
  - scope：指定该 Map Bean 实例的作用域
- properties：该元素用于加载一份资源文件，并根据加载的资源文件创建一个 Properties Bean 实例，支持以下三个属性
  - id：该属性指定定义一个名为 id 的 Properties Bean 实例
  - location：指定资源文件的位置
  - scope：指定该 Properties Bean 实例的作用域

7.12. Spring3.0 提供的表达式语言(SqEL)

1、Spring 表达式语言(简称 SpEL)是一种与 JSP 2 的 EL 功能类似的表达式语言，它可以在运行时查询和操作对象图

2、与 JSP2 的 EL 相比，SpEL 的功能更强大，它甚至支持方法调用和基本字符串模板函数

3、SpEL 可以独立于 Spring 容器使用---只是当成简单的表达式语言来用；也可在 Annotation 或 XML 配置中使用 SpEL，这样可以充分利用 SpEL 简化 Spring 的 Bean 配置

7.12.1. 使用 Expression 接口进行表达式求值

1、Spring 的 SpEL 可以单独使用，可以使用 SpEL 对表达式计算、求值。SpEL 主要提供了如下三个接口

- ExpressionParser：该接口的实例负责解析一个 SpEL 表达式，返回一个 Expression 对象
- Expression：该接口的实例代表一个表达式
- EvaluationContext：代表计算表达式值的上下文

7.12.2. Bean 定义中的表达式语言支持

1、SpEL 的一个重要作用就是扩展 Spring 容器的功能，允许在 Bean 定义中使用 SpEL。在 XML 配置文件和 Annotation 中都可以使用 SpEL，在 XML 配置文件和 Annotation 中使用 SpEL 时，在表达式外面增加#{}包围即可

7.12.3. SpEL 语法详述

1、直接量表达式

- 直接量表达式是 SpEL 中最简单的表达式，直接量表达式就是在表达式中使用 Java 语言支持的直接量，包括字符串、日期、数值、boolean 值和 null

2、在表达式中创建数组
- SpEL 表达式直接支持使用静态初始化、动态初始化两种语法来创建数组

3、在表达式中创建数组
- SpEL 表达式直接支持使用静态初始化、动态初始化两种语法来创建数组

4、在表达式中访问 List、map 等集合元素

5、调用方法
- 在 SpEL 中调用方法与在 Java 代码中调用方法没有任何区别

6、算数、比较、逻辑、赋值、三目等运算符
- 与 JSP 2 EL 类似的是 SpEL 同样支持算数、比较、逻辑、赋值、三目运算符等各种运算符

7、类型运算符
- SpEL 提供了一个特殊的运算符：T()，这个运算符用于告诉 SpEL 将该运算符内的字符串当成"类"处理，避免 Spring 对其进行其他解析。尤其是调用某个类的静态方法时，T()运算符尤其有用

8、调用构造器
- SpEL 允许在表达式中直接使用 new 来调用构造器，这种调用可以创建一个 Java 对象

9、变量
- SpEL 允许通过 EvaluationContext 来使用变量，该对象包含了一个 setVariable(String name,Object value)方法，该方法用于设置一个变量
- 一旦在 EvaluationContext 中设置了变量，就可以在 SpEL 中通过#name 来访问该变量
  - 两个特殊变量

  1) #this：引用 SpEL 当前正在计算的对象
  2) #root：引用 SpEL 的 EvaluationContext 的 root 对象

10、自定义函数
- SpEL 允许开发者开发自定义函数，类似于 JSP 2 EL 中的自定义函数，所谓自定义函数，也就是为 Java 方法重新起个名字而已
- 通过 StandardEvaluationContext 的如下方法即可在 SpEL 中注册自定义函数

11、Elvis 运算符
- Elvis 运算符只是三目运算符的特殊写法
  name!=null?name:"newValue"
  name?:"newVal"

12、安全导航操作
- 若引用是 null，对引用的操作则直接返回 null，而不会抛出异常

13、集合选择
- SpEL 允许直接对集合进行选择操作，这种选择操作可以根据指定表达式对集合元素进行筛选，只有符合条件的集合元素才会被选择出来

14、集合投影

- SpEL 允许对集合进行投影运算，这种投影运算将依次迭代每个集合元素，迭代时根据指定表达式对集合元素进行计算得到一个新的结果，依次将每个结果收集成新的集合，这个新的集合将作为投影运算的结果

15、表达式模板

- 表达式模板的本质是对"直接量表达式"的扩展，它允许在"直接量表达式"中插入一个或多个#{expr}，#{expr}将会被动态计算出来

Chapter 8. 深入使用 Spring

8.1. 两种后处理器

1、Spring 框架提供了很好的扩展性，除了可以与各种第三方框架良好整合外，其 IoC 容器也允许开发者进行扩展，这种扩展甚至无须实现 BeanFactory 或 ApplicationContext 接口，而是允许通过两个后处理器对 IoC 容器进行扩展

2、Spring 提供两种常用的后处理器

- Bean 后处理器：这种处理器会对容器中的 Bean 进行后处理，对 Bean 进行额外加强
- 容器后处理器：这种后处理器对 IoC 容器进行后处理，用于增强容器功能

8.1.1. Bean 后处理器

1、Bean 后处理器是一种特殊的 Bean，这种 Bean 并不对外提供服务，它甚至可以无须 id 属性，它主要负责对容器中的其他 Bean 执行后处理，例如为容器中的目标 Bean 生成代理等，这种 Bean 被称为 Bean 后处理器

2、Bean 后处理器会在 Bean 实例创建成功之后，对 Bean 实例进行进一步的增强处理

3、Bean 后处理器必须实现 BeanPostProcessor 接口，BeanPostProcessor 接口包含如下两个方法

1) Object postProcessBeforeInitialization(Object bean,String name) throws BeansException：该方法的第一个参数是系统即将进行后处理的 Bean 实例，第二个参数是该 Bean 的配置 id
2) Object postProcessAfterInitialization(Object bean,String name) throws BeansException：该方法的第一个参数是系统即将进行后处理的 Bean 实例，第二个参数是该 Bean 的配置 id

- 实现该接口的 Bean 后处理器必须实现这两个方法，这两个方法会对容器的 Bean 进行后处理，会在目标 Bean 初始化之前、初始化之后分别被回调
- Bean 后处理器是对 IoC 容器一种极好的扩展，Bean 后处理器可以对容器中的 Bean 进行后处理，而到底对 Bean 进行怎样的后处理则完全取决于开发者。Spring 容器负责把各 Bean 创建出来，Bean 后处理器(由开发者提供)可以依次对每个 Bean 进行某种修改、增强、从而可以对容器中的 Bean 集中增加某种功能

4、如果使用 ApplicationContext 作为 Spring 容器，Spring 容器会自动检测容器中的所有 Bean，如果发现某个 Bean 类实现了 BeanPostProcessor 接口，ApplicationContext 会自动将其注册为 Bean 后处理器

5、如果使用 BeanFactory 作为 Spring 容器，则必须手动注册 Bean 后处理器，程序必须获取 Bean 后处理器实例，然后手动注册，在这种需求下，程序需要在配置文件中为 Bean 后处理器指定 id 属性，实例详见 P617

8.1.2. Bean 后处理器的用处

1、两个常用的后处理器

1) BeanNameAutoProxyCreator：根据 Bean 实例的 name 属性，创建 Bean 实

例的代理

2) DefaultAdvisorAutoProxyCreator：根据提供的 Advisor，对容器中的所有 Bean 实例创建代理

- 这两个 Bean 后处理器都用于根据容器中配置的拦截器，创建代理 Bean，代理 Bean 就是对目标 Bean 进行增强，在目标 Bean 的基础上进行修改得到的新 Bean
- 如果需要对容器中某一批 Bean 进行通用的增强处理，则可以考虑使用 Bean 后处理器

8.1.3. 容器后处理器

1、Spring 还提供了一种容器后处理器，Bean 后处理器负责容器中所有 Bean 实例，而容器后处理器负责处理容器本身

2、容器后处理器必须实现 BeanFactoryPostProcessor 接口，实现该接口必须实现如下方法

- postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory)

3、类似于 BeanPostProcessor，ApplicationContext 可以自动检测到容器中的容器后处理器，并且自动注册容器后处理器，但若使用 BeanFactory 作为 Spring 容器，则必须手动调用该容器后处理器来处理 BeanFactory 容器

4、Spring 提供如下几个常用的容器后处理器

- PropertyPlaceholderConfigurer：属性占位符配置器
- PropertyOverrideConfigurer：重写占位符配置器
- CustomAutowireConfigurer：自定义自动装配的配置器
- CustomScopeConfigurer：自定义作用域的配置器

5、如果有需要，程序可以配置多个容器后处理器，多个容器后处理器可设置 order 属性来控制容器后处理器的执行次序

- 为了给容器后处理器指定 order 属性，则要求容器后处理器必须实现 Ordered 接口，因此在实现 BeanFactoryPostProcessor 时，就应当考虑实现 Ordered 接口

8.1.4. 属性占位符配置器

1、Spring 提供了 PropertyPlaceholderConfigurer，负责读取 Properties 属性文件里的属性值，并将这些属性值设置成 Spring 配置文件的数据

2、通过使用 PropertyPlaceholderConfigurer 后处理器，可以将 Spring 配置文件中的部分数据放在属性文件中设置，这样的优势：可以将部分相似的配置(比如数据库的 URL、用户名和密码)放在特定的属性文件中

3、通过这种方式，可以从主 XML 配置文件中分离出部分配置信息，降低修改配置文件产生错误的风险

4、示例详见 P620

8.1.5. 重写占位符配置器

1、PropertyOverrideConfigurer 是 Spring 提供的另一个容器后处理器，这个后处理器比 PropertyPlaceholderConfigurer 后处理器功能更加强大，PropertyOverrideConfigurer 的属性文件指定的信息可以直接覆盖 Spring 配置文件

中的元数据---当 XML 配置文件和属性文件指定的元数据不一致时，属性文件的信息取胜

8.2. Spring 的"零配置"支持

8.2.1. 搜索 Bean 类

1、既然不再使用 Spring 配置文件来配置任何 Bean 实例，那么只能希望 Spring 会自动搜索某些路径下的 Java 类，并将这些 Java 类注册成 Bean 实例

2、Spring 没有采用"约定优于配置"的策略，Spring 依然要求程序员显式指定搜索哪些路径下的 Java 类，Spring 将会把合适的 Java 类全部注册成 Spring Bean

3、Spring 通过使用一些特殊的 Annotation 来标注 Bean 类

1) @Compoent：标注一个普通的 Spring Bean 类
2) @Controller：标注一个控制器组件类
3) @Service：标注一个业务逻辑组件类
4) @Repository：标注一个 DAO 组件类

4、Bean 实例名称

- 在基于 XML 配置方式下，每个 Bean 实例的名称都由其 id 属性指定的
- 在基于 Annotation 的方式下，Spring 采用约定的方式来为这些 Bean 实例指定名称---默认是 Bean 类的首字母小写，其他不变
- Spring 也允许在使用@Component 标注时指定 Bean 实例的名称

5、Spring 会自动搜索所有以@Compoent、@Controller、@Service、@Repository 标注的 Java 类，并将它们当成 Spring Bean 来处理；除此之外，还可通过为 <component-scan.../>元素添加<include-filter.../>或<exclude-filter.../>子元素来指定 Spring Bean 类，只要位于指定路径下的 Java 类满足这种规则，即使这些 Java 类没有使用任何 Annotation 标注，Spring 一样会将它们当成 Bean 类来处理

- <include-filter.../>元素用于指定满足规则的 Java 类会被当成 Bean 类处理
- <exclude-filter.../>元素用于指定满足规则的 Java 类
- 被当成 Bean 类处理
- 使用这两个元素都需要指定如下两个属性
  - type：指定过滤器类型
  - expression：指定过滤器所需要的表达式

6、Spring 内建支持如下 4 种过滤器

- annotation：Annotation 过滤器，该过滤器需要指定一个 Annotation
-
- assignable：类名过滤器，该过滤器直接指定一个 Java 类
- regex：正则表达式过滤器，该过滤器指定一个正则表达式，匹配该正则表达式的 Java 类满足该过滤规则，例如"org\.example\.Default.*"
- aspectj：Aspectj 过滤器，例如"org.example..*Service+"

8.2.2. 指定 Bean 的作用域

1、当使用 XML 配置方式来配置 Bean 实例时，可以通过 scope 来指定 Bean 实例的作用域，没有指定 scope 属性的 Bean 实例的作用域默认是 singleton

2、当采用零配置方式来管理 Bean 实例时，可以使用@Scope Annotation，只要在该 Annotation 中提供做作用域的名称即可

8.2.3. 使用@Resource 配置依赖

1、@Resource 位于 javax.annotation 包下，是来自 Java EE 规范的一个 Annotation，Spring 直接借鉴了该 Annotation，通过使用该 Annotation 为目标 Bean 指定协作者 Bean

2、@Resource 有一个 name 属性，在默认情况下，Spring 将这个值解释为需要被注入的 Bean 实例的 id，换句话说，使用@Resource 与<property.../>元素的 ref 属性有相同的效果

3、@Resource 不仅可以修饰 setter 方法，也可以直接修饰实例变量，如果使用@Resource 修饰实例变量将会更加简单，此时 Spring 将会直接使用 Java EE 规范的 Field 注入，此时连 setter 方法都可以不要

8.2.4. 使用@PostConstruct 和@PreDestroy 定制生命周期行为

1、@PostConstruct 和@PreDestroy 同样位于 javax.annotation 包下，也是来自 Java EE 规范的两个 Annotation，Spring 直接借鉴了它们，用于定制 Spring 容器中 Bean 的生命周期行为

2、<bean.../>元素可以指定 init-method、destory-method 两个属性

- init-method 指定 Bean 的初始化方法---Spring 容器将会在 Bean 的依赖关系注入完成后回调该方法
- destroy-method 指定 Bean 销毁之前的方法---Spring 容器将会在销毁该 Bean 之前回调该方法
- @PostConstruct 和@PreDestory 两个注解的作用大致与此相似，都用于修饰方法，无需任何属性

8.2.5. Spring3.0 新增的注解

1、Spring3.0 再次增加两个 Annotation：@DependsOn 和 @Lazy，其中@DependsOn 用于强制初始化其他 Bean；而@Lazy 用于指定该 Bean 是否取消预初始化

2、@DependsOn 可以修饰 Bean 类或方法，使用该 Annotation 可以指定一个字符串数组作为参数，每个数组元素对应于一个强制初始化的 Bean

3、@Lazy 修饰 Spring Bean 类用于指定该 Bean 的预初始化行为，使用该注解时可指定一个 boolean 类型的 value 属性，该属性决定是否要预初始化该 Bean

8.2.6. Spring4.0 增强的自动装配和精确装配

1、Spring 提供了@Autowired 注解来指定自动装配，@Autowired 可以修饰 setter 方法、普通方法、实例变量和构造器等。当使用@Autowired 标注 setter 方法时，默认采用 byType 自动装配策略

2、@Autowired 总是采用 byType 的自动装配策略，在这种策略下，符合自动装配类型的候选 Bean 常常有多个，这个时候就可能引起异常(对于数组类型参数、集合类型参数则不会)

3、为了实现精确装配，Spring 提供了@Qualifier 注解，通过使用@Qualifier，允许根据 Bean 的 id 来执行自动装配

- 使用 @Qualifier 注解的意义并不大，如果程序使用 @Autowired 和

@Qualifier 实现精确的自动装配，还不如直接使用@Resource 注解执行依赖注入

- @Qualifier 通常用于修饰实例变量
- @Qualifier 还可以标注方法的形参

8.3. 资源访问

1、创建 Spring 容器时通常需要访问 XML 配置文件。除此之外，程序还可能大量地访问各种类型的文件、二进制流等---Spring 把这些文件、二进制流统称为资源文件

2、在 Sun 所提供的标准 API 里，资源访问通常由 java.net.URL 和文件 IO 来完成，如果需要访问来自网络的资源时，则通常会选择 URL 类

3、Spring 改进了 Java 资源访问的策略，Spring 为资源访问提供了一个 Resource 接口，该接口提供了更强的资源访问能力，Spring 框架本身大量使用了 Resource 来访问底层资源

4、Resource 本身是一个接口，是具体资源访问策略的抽象，也是所有资源访问类所实现的接口，该接口提供了如下几个方法

- getInputStream()：定位并打开资源，返回资源对应的输入流。每次调用都返回新的输入流。调用者必须负责关闭输入流
- exists()：返回 Resource 所指向的资源是否存在
- isOpen()：返回资源文件是否打开，如果资源文件不能多次读取，每次读取结束时应该显式关闭，以防止资源泄露
- getDescription()：返回资源的描述信息，用于资源处理出错时输出该信息，通常是全限定文件名或实际 URL
- getFile()：返回资源对应的 File 对象
- getURL()：返回资源对应的 URL 对象

5、Resource 接口本身并没有提供访问任何底层资源的实现逻辑，针对不同的底层资源，Spring 将会提供不同的 Resource 实现类，不同的实现类负责不同的资源访问逻辑

8.3.1. Resource 实现类

1、Resource 接口是 Spring 资源访问的接口，具体的资源访问由该接口的实现类完成。Spring 提供 Resource 接口的大量实现类

- UrlResource：访问网络资源的实现类
- ClassPathResource：访问类加载路径里资源的实现类
- FileSystemResouce：访问文件系统里资源的实现类
- ServletContextResource：访问相对于 ServletContext 路径下单资源的实现类
- InputStreamResource：访问输入流资源的实现类
- ByteArrayResource：访问字节数组资源的实现类

8.3.1.1. 访问网络资源

1、访问网络资源通过 UrlResource 类实现，UrlResource 是 java.net.URL 类的包装，主要用于访问之前通过 URL 类访问的资源对象。URL 资源通常应该提供标准的

协议前缀，例如：file:用于访问文件系统，http:用于访问 HTTP 协议访问资源；ftp:用于通过 FTP 协议访问资源等

8.3.1.2. 访问类加载路径下的资源

1、ClassPathResource 用来访问类加载路径下的资源，相对于其他的 Resource 实现类，其主要优势是方便访问类加载路径下的资源，尤其对于 Web 应用，ClassPathResource 可自动搜索位于 WEB-INF/classes 下的资源文件，无须使用绝对路径访问

2、ClassPathResource 实例可使用 ClassPathResource 构造器显式创建，但更多时候它都是隐式创建。当执行 Spring 的某个方法时，该方法接受一个代表资源路径的字符串参数，当 Spring 识别该字符串参数中包含 classpath:前缀后，系统将会自动创建 ClassPathResource 对象

8.3.1.3. 访问文件系统资源

1、Spring 提供的 FileSystemResource 类用于访问文件系统资源。使用 FileSystemResource 来访问文件系统资源并没有太大优势，因为 Java 提供的 File 类也可用于访问文件系统资源

2、FileSystemResource 实例可使用 FileSystemResource 构造器显式创建，但更多时候它都是隐式创建。当执行 Spring 的某个方法时，该方法接受一个代表资源路径的字符串参数，当 Spring 识别该字符串参数中包含 file:前缀后，系统将会自动创建 FileSystemResource 对象

8.3.1.4. 访问应用相关资源

1、Spring 提供了 ServletContextResource 类来访问 Web Context 下相对路径下的资源，ServletContextResource 构造器接受一个代表资源位置的字符串参数，该资源位置是相对于 Web 应用根路径的位置

2、使用 ServletContextResource 访问的资源，也可通过文件 IO 访问或 URL 访问。通过 java.io.File 访问要求资源被解压缩，而且在本地文件系统中；但使用 ServletContextResource 进行访问时则无须关心资源是否被解压缩出来，或者直接存放在 JAR 文件中，总可通过 Servlet 容器访问

- 当直接通过 File 来访问 Web Context 下相对路径下的资源时，应该先使用 ServletContext 的 getRealPath()获取资源绝对路径，然后再以该绝对路径创建 File 对象

8.3.1.5. 访问字节数组资源

1、Spring 提供了 InputStreamResource 来访问二进制输入流资源，InputStreamResource 是针对输入流的 Resource 实现，只有当没有合适的 Resource 实现时，才考虑使用该 InputStreamResource。通常情况下，优先考虑使用 ByteArrayResource，或者基于文件的 Resource 实现

2、Spring 提供的 ByteArrayResource 用于直接访问字节数组资源，字节数组是一种常见的信息传输方式：网络 Socket 之间的信息交换，或者线程之间的信息交换，字节数组都被作为信息载体。ByteArrayResource 可以将字节数组包装成 Resource 使用

3、在实际应用中，字节数组可能通过网络传输获得，也可能通过管道流获得，还可能通过其他方式获得。只要得到了代表资源的字节数组，程序就可以通过 ByteArrayResource 将字节数组包装成 Resource 实例，并利用 Resource 来访问该资源

8.3.2. ResourceLoader 接口和 ResourceLoaderAware 接口

1、Spring 提供两个标志性接口

1) ResourceLoader：该接口实现类的实例可获得一个 Resource 实例
   - Resource getResource(String location)：该接口仅包含这个方法，该方法用于返回一个 Resource 实例
   - ApplicationContext 的实现类都实现 ResourceLoader 接口，因此 ApplicationContext 可用于直接获取 Resource 实例
   - ApplicationContext 实例获取 Resource 实例时，默认采用与 ApplicationContext 相同的资源访问策略
2) ResourceLoaderAware：该接口实现类的实例将获得一个 ResourceLoader 的引用

2、Spring 需要进行资源访问时，实际上并不需要直接使用 Resource 实现类，而是调用 ResourceLoader 实例的 getResource()方法来获得资源，ResourceLoader 将会负责选择 Resource 的实现类，也就是确定具体的资源访问策略，从而将应用程序和具体的资源访问策略分离开来

3、使用 ApplicationContext 访问资源时，可不理会 ApplicationContext 实现类，强制使用指定的 ClassPathResource、FileSystemResource 等实现类，这可通过不同前缀来指定

```
Resource r=ctx.getResource("classpath:beans.xml")
```

4、常见的前缀及对应的访问策略

- classpath:---以 ClassPathResource 实例访问类加载路径下的资源
- file:---以 UrlResource 实例访问本地文件系统的资源
- http:---以 UrlResource 实例访问基于 HTTP 协议的网络资源
- 无前缀---由 ApplicationContext 的实现类来决定访问策略

5、ResourceLoaderAware 完全类似于 Spring 提供的 BeanFactoryAware、BeanNameAware 接口，ResourceLoaderAware 接口也提供了一个 setResourceLoader()方法，该方法由 Spring 容器负责调用，Spring 容器会将一个 ResourceLoader 对象作为该方法的参数传入

- 如果把实现 ResourceLoaderAware 接口的 Bean 部署在 Spring 容器中，Spring 容器会将自身当成 ResourceLoader 作为 setResourceLoader()方法的参数传入

8.3.3. 使用 Resource 作为属性

1、当应用程序中的 Bean 实例需要访问资源时，Spring 有更好的解决办法：直接利用依赖注入

2、总结起来，如果 Bean 实例需要访问资源，则有两种解决方案

1) 在代码中获取 Resource 实例
2) 使用依赖注入

- 对于第一种方式，当程序获取 Resource 实例时，总需要提供 Resource 所在的位置，这意味着资源所在的物理位置将被耦合到代码中，如果资源位置发生改变，则必须改写程序
- 通常建议采用第二种方式，让 Spring 为 Bean 实例依赖注入资源，采用依赖注入，允许动态配置资源文件位置，无须将资源文件位置写在代码中，当资源文件位置发生变化时，无须改写程序，直接修改配置文件即可

8.3.4. 在 ApplicationContext 中使用资源

1、不管以怎样的方式创建 ApplicationContext 实例，都需要为 ApplicationContext 指定配置文件，Spring 允许使用一份或多份 XML 配置文件

2、ApplicationContext 确定资源访问策略通常有两种方法

- 使用 ApplicationContext 实现类指定访问策略
- 使用前缀指定访问策略

8.3.4.1. 使用 ApplicationContext 实现指定访问策略

1、创建 ApplicationContext 对象时，通常可以使用如下三个实现类

1) ClassPathXmlApplicationContext：对应使用 ClassPathResource 进行资源访问
2) FileSystemXmlApplicationContext：对应使用 FileSystemResource 进行资源访问
3) XmlWebApplicationContext：对应使用 ServletContextResource 进行资源访问

- 当使用 ApplicationContext 的不同实现类时，就意味着 Spring 使用相应的资源访问策略

```
ApplicationContext ctx=new FileSystemXmlApplicationContext("beans.xml");
```

8.3.4.2. 使用前缀指定访问策略

1、Spring 也允许使用前缀来指定资源访问策略

```
ApplicationContext ctx=
new FileSystemXmlApplicationContext("classpath:beans.xml");
```

- 即使采用了 FileSystemXmlApplicationContext 实现类，但程序依然从类加载路径下搜索 beans.xml 配置文件
- 前缀仅仅对当前访问有效，后面进行访问时，如果没有加前缀，仍然以 ApplicationContext 的实现类来选择对应的资源访问策略

8.3.4.3. classpath*:前缀的用法

1、classpath*:前缀提供了加载多个 XML 配置文件的能力，当使用 classpath*:前缀来指定 XML 配置文件时，系统将搜索类加载路径，找出所有与文件名匹配的文件，分别加载文件中的配置定义，最后合并成一个 ApplicationContext

2、如果是 classpath:前缀，Spring 则只加载第一个符合条件的 XML 文件

3、该前缀 classpath*:仅对 ApplicationContext 有效，因此该前缀不可用于 Resource，使用 classpath*:前缀一次性访问多个资源是行不通的

4、另外还有一种可以一次性加载多个配置文件的方式，即：指定配置文件时使用通配符

ApplicationContext ctx=new ClassPathXmlApplicationContext("beans*.xml");

8.3.4.4. file:前缀的用法

1、当 FileSystemXmlApplicationContext 作为 ResourceLoader 使用时，它会简单地让所有绑定的 FileSystemResource 实例把绝对路径都当成相对路径处理，不管是否以斜杠"/"开头

2、当程序中需要访问绝对路径，建议强制使用 file:前缀来加以区分

8.4. Spring 的 AOP

1、AOP(Aspect Orient Programming)，面向切面的编程，作为面向对象编程的一种补充，已经成为一种比较成熟的编程方式

2、AOP 和 OOP 互为补充，面向对象编程将程序分解成各个层次的对象，而面向切面的编程将程序运行过程分解成各个切面

3、可以这样理解：面向对象编程是从静态角度考虑程序结构，面向切面编程则是从动态角度考虑程序运行过程

8.4.1. 为什么需要 AOP

1、传统的 OOP 编程思想里以对象为核心，整个软件系统由一系列相互依赖的对象组成，而这些对象将被抽象成一个类，并允许使用类继承来管理类与类之间一般到特殊的关系，随着软件规模的增大，应用程序的逐渐升级，慢慢出现了一些 OOP 很难解决的问题

8.4.2. 使用 Aspectj 实现 AOP

1、Aspectj 是一个基于 Java 语言的 AOP 框架，提供了强大的 AOP 功能，其他很多 AOP 框架都借鉴或采纳其中的一些思想

2、Aspectj 主要包括两个部分：

1) 一个部分定义如何表达、定义 AOP 编程中的语法规范，通过这套语法规范，可以方便地用 AOP 来解决 Java 语言中存在的交叉关注点的问题
2) 另一个部分是工具部分，包括编译器、调试工具等

8.4.2.1. 下载和安装 AspectJ

1) http://www.eclipse.org/aspectj/downloads.php
2) java -jar aspectj-x.x.x.jar
3) 按图形化界面操作即可
4) 最后将 PATH 以及 classpath 配置好即可

8.4.2.2. AspectJ 使用入门

1、在 AspectJ 的安装路径下有如下文件结构

- bin：该路径下存放了 aj、aj5、ajc、ajdoc、ajbrowser 等命令，其中 ajc 最常用，它的作用类似于 javac，用于对普通的 java 类进行编译时增强
- docs：该路径下存放了 AspectJ 的使用说明、参考手册、API 文档等文档
- lib：该路径下的 4 个 JAR 文件是 AspectJ 的核心类库
- 相关授权文件

2、客户源代码不需要进行修改，在编译时会加入 AspectJ 的很多内容，因此被称为编译时增强的 AOP 框架

3、AOP 要达到的效果是，保证在程序员不修改源代码的前提下，为系统中业务组件的多个业务方法添加某种通用功能。但 AOP 的本质是，依然要去修改业务组件的多个业务方法的源代码---只是这个修改由 AOP 框架完成，程序员不需要修改

4、AOP 实现可分为两类(按 AOP 框架修改源代码的时机)

1) 静态 AOP 实现：AOP 框架在编译阶段对程序进行修改，即实现对目标类的增强，生成静态的 AOP 代理类(生成的*.class 文件已经被改掉了，需要使用特定的编译器)，以 AspectJ 为代表
2) 动态 AOP 实现：AOP 框架在运行阶段动态生成 AOP 代理(在内存中以 JDK 动态代理或 cglib 动态生成 AOP 代理类)，以实现对目标对象的增强，以 Spring AOP 为代表

- 一般来说，静态 AOP 实现具有较好的性能，但需要使用特殊的编译器，动态 AOP 实现是纯 Java 实现，因此无须特殊的编译器，但是通常性能略差

8.4.3. AOP 的基本概念

1、AOP 从程序运行角度考虑程序的流程，提取业务处理过程的切面，AOP 面向的是程序运行中各个步骤，希望以更好的方式来组合业务处理的各个步骤

2、AOP 框架并不与特定的代码耦合，AOP 框架能处理程序执行中特定的切入点(Pointcut)，而不与某个具体类耦合，AOP 框架由如下两个特征

1) 各步骤之间的良好隔离性
2) 源代码无关性

3、AOP 编程的术语介绍

1) 切面(Aspect)：切面用于组织多个 Advice，Advice 放在切面中定义
2) 连接点(Joinpoint)：程序执行过程中明确的店，如方法的调用，或者异常的抛出，在 Spring AOP 中，连接点总是方法的调用
3) 增强处理(Advice)：AOP 框架在特定的切入点执行的增强处理，处理有"around"、"before"、"after"等类型
4) 切入点(Pointcut)：可以插入增强处理的连接点。简而言之，当某个连接点满足指定要求时，该连接点将被添加增强处理，该连接点也就变成了切入点

4、如何使用表达式来定义切入点是 AOP 的核心，Spring 默认使用 AspectJ 切入点语法

1) 引入：将方法或字段添加到被处理的类中。Spring 允许将新的接口引入到任何被处理的对象中
2) 目标对象：被 AOP 框架进行增强处理的对象，也被称为被增强的对象，如果 AOP 框架采用的是动态 AOP 实现，那么该对象就是一个被代理的对象
3) AOP 代理：AOP 框架创建的对象，简单地说，代理就是目标对象的加强。Spring 中的 AOP 代理可以是 JDK 动态代理，也可以是 cglib 代理

4) 织入(Weaving)：将增强处理添加到目标对象中，并创建一个被增强的对象(AOP 代理)的过程就是织入。织入有两种实现方式---编译时增强(AspectJ)和运行时增强(Spring AOP)

8.4.4. Spring 的 AOP 支持

1、Spring 中的 AOP 代理由 Spring 的 IoC 容器负责生成、管理，其依赖关系也由 IoC 容器负责管理。因此 AOP 代理可以直接使用容器中的其他 Bean 实例作为目标，这种关系可由 IoC 容器的依赖注入提供。Spring 默认使用 Java 动态代理来创建 AOP 代理，这样就可以为任何接口实例创建代理了

2、Spring 也可以使用 cglib 代理，在需要代理类而不是代理接口的时候，Spring 会自动切换为使用 cglib 代理。但 Spring 推荐使用面向接口编程，因此业务对象通常都会出现一个或多个接口，此时默认将使用 JDK 动态代理，但也可强制使用 cglib 代理

3、Spring AOP 使用纯 Java 实现，不需要特定的编译工具，Spring AOP 也不需要控制类装载器层次，因此它可以在所有的 Java Web 容器或应用服务器中运行良好

4、Spring 目前仅支持将方法调用作为连接点(Joinpoint)，如果需要把对成员变量访问和更新也作为增强处理的连接点，则考虑使用 AspectJ

5、Spring 实现 AOP 的方法与其他框架不同，Spring 并不是要提供完整的 AOP 实现，Spring 侧重于 AOP 实现也 Spring IoC 容器之间的整合，用于帮助解决企业级开发中的常见问题

6、纵观 AOP 编程，其中需要程序员参与的有如下三个部分

- 定义普通业务组件
- 定义切入点，一个切入点可能横切多个业务组件
- 定义增强处理，增强处理就是 AOP 框架为普通业务组件织入的处理动作

AOP 代理方法=增强处理+目标对象的方法

7、Spring 有如下两种选择来定义切入点和增强处理

- 基于注解的"零配置"方式：使用@Aspect、@Pointcut 等注解来标注切入点和增强处理
- 基于 XML 配置文件的管理方式：使用 Spring 配置文件来定义切入点和增强处理

8.4.5. 基于注解的"零配置"方式

1、AspectJ 允许使用注解定义切面、切入点和增强处理，而 Spring 框架则可识别并根据这些注解来生成 AOP 代理，Spring 只是使用了 AspectJ 5 一样的注解，但并没有使用 AspectJ 的编译器或者织入器，底层依然使用的是 Spring AOP，依然是在运行时动态生成 AOP 代理，并不依赖于 AspectJ 的编译器或者织入器

2、自动增强，指的是 Spring 会判断一个或多个切面是否需要对指定 Bean 进行增强，并据此自动生成响应的代理，从而使得增强处理在合适的时候被调用

3、如果不打算使用 Spring 的 XML Schema 配置方式，则应该在 Spring 配置文件中增加如下片段来启用@AspectJ 支持

```
<bean class="org.springframework.aop.aspectj.annotation.
AnnotationAwareAspectJAutoProxyCreator"/>
```

- AnnotationAwareAspectJAutoProxyCreator 是一个 Bean 后处理器，该 Bean 后处理器将会为容器中的所有 Bean 生成 AOP 代理
- 为了在 Spring 应用中启动@AspectJ 支持，还需要在应用的类加载路径下增加两个 AspectJ 库：aspectjweaver.jar 和 aspectjrt.jar，直接使用 AspectJ 安装路径下 lib 目录中的两个 jar 文件即可
- 除此之外，还需要一个 aopalliance.jar

8.4.5.1. 定义切面 Bean

1、当启动了@AspectJ 支持后，只要在 Spring 容器中配置一个带@Aspect 注解的 Bean，Spring 将会自动识别该 Bean，并将该 Bean 作为切面处理
2、切面类(用@Aspect 修饰的类)和其他类一样可以有方法、成员变量定义，还可能包括切入点、增强处理定义
3、当使用@Aspect 来修饰一个 Java 类之后，Spring 将不会把该 Bean 当成组件 Bean 处理，因此负责自动增强的后处理 Bean 将会略过该 Bean，不会对该 Bean 进行任何增强处理

8.4.5.2. 定义 Before 增强处理

1、在一个切面类里使用@Before 来修饰一个方法时，该方法将作为 Before 增强处理，使用@Before 修饰时，通常需要指定一个 value 属性值，该属性值指定一个切入点表达式(既可以是一个已有的切入点，也可以直接定义切入点表达式)，用于指定该增强处理将被织入哪些切入点

8.4.5.3. 定义 AfterReturning 增强处理

1、类似于使用@Before 注解可修饰 Before 增强处理，使用@AfterReturning 可修饰 AfterReturning 增强处理，AfterReturning 增强处理将在目标方法正常完成后被织入
2、使用@AfterReturning 注解可指定如下两个常用属性

1) pointcut/value：这两个属性的作用是一样的，它们都用于指定该切入点对应的切入表达式。一样既可以是一个已有的切入点，也可以直接定义切入点的表达式。当指定 pointcut 属性值后，value 属性值将会被覆盖
2) returning：该属性值指定一个形参名，用于表示 Advice 方法中可定义与此同名的形参，该形参可用于访问目标方法的返回值。除此之外，在 Adivce 方法中定义该形参(代表目标方法的返回值)时指定的类型，会限制目标方法必须返回指定类型的值会没有返回值

- returning 属性还有一个额外的作用：它可用于限定切入点只匹配具有对应返回值类型的方法---若为 Object 则可以匹配任意返回类型的方法

3、虽然 AfterReturning 增强处理可以访问到目标方法的返回值，但它不可以改变目标方法的返回值

8.4.5.4. 定义 AfterThrowing 增强处理

1、使用@AfterThrowing 注解可修饰 AfterThrowing 增强处理，AfterThrowing 增强处理主要用于处理程序中未处理的异常
2、使用@AfterThrowing 注解可以指定两个常用属性

1) pointcut/value：这两个属性的作用是一样的，都用于指定切入点对应的切入表达式。一样既可以是一个已有的切入点，也可以直接定义切入点的表达式。当指定 pointcut 属性值后，value 属性值将会被覆盖
2) throwing：该属性指定一个形参名，用于表示 Advice 方法中可定义与此同名的形参，该形参可用于访问目标方法抛出的异常。除此之外，在 Advice 方法中定义该形参(代表目标方法抛出的异常)时指定的类型，会限制目标方法必须抛出指定类型的异常

3、AOP 的 AfterThrowing 处理虽然可以对目标方法的异常进行处理，但这种处理与直接使用 catch 捕捉不同---catch 捕捉意味着完全处理该异常，如果 catch 块中没有重新抛出新异常，则该方法可以正常结束；而 AfterThrowing 处理了该异常，但它不能完全处理该异常，该异常依然会传播到上一级调用者

8.4.5.5. After 增强处理

1、Spring 提供了 After 增强处理，它与 AfterReturning 增强处理有点相似，但也有区别

- AfterReturning 增强处理只在目标方法成功完成后才会被织入
- After 增强处理不管目标方法如何结束(包括成功完成和遇到异常终止两种情况)，它都会被织入，因此常用于释放资源

2、使用@After 注解修饰一个方法，即可将该方法转成 After 增强处理，使用@After 注解时需要指定一个 value 属性，该属性值用于指定该增强处理被织入的切入点，即可是一个已有的切入点，也可直接指定切入点表达式

8.4.5.6. Around 增强处理

1、@Around 注解用于修饰 Around 增强处理，Around 增强处理是功能比较强大的增强处理，它近似等于 Before 增强处理和 AfterReturning 增强处理的总和，Around 增强处理即可在执行目标方法之前织入增强动作，也可在执行目标方法之后织入增强动作

2、与 Before 增强处理、AfterReturning 增强处理不同的是，Around 增强处理可以决定目标方法在什么时候执行，如何执行，甚至可以完全阻止目标方法的执行

3、Around 增强处理可以改变目标方法的参数值，也可以改变执行目标方法后的返回值

4、Around 增强处理方法使用@Around 来标注，使用@Around 注解时需要指定一个 value 属性，该属性指定该增强处理被织入的切入点

5、当定义一个 Around 增强处理方法时，该方法的第一个形参必须是 ProceedingJoinPoint 类型(至少包含一个形参)，在增强处理方法体内，调用 ProceedingJoinPoint 参数的 proceed()方法才会执行目标方法---这就是 Around 增强处理可以完全控制目标方法的执行时机、如何执行的关键

6、调用 ProceedingJoinPoint 参数的 proceed()方法时，还可以传入一个 Object[]对象作为参数，该数组中的值被传入目标方法作为执行方法的实参

7、为了能获取目标方法的参数的个数和类型，需要增强处理方法能访问目标方法的参数

8.4.5.7. 访问目标方法的参数

1、访问目标方法最简单的做法就是定义增强处理方法时将第一个参数定义为 JoinPoint 类型，当该增强处理方法被调用时，该 JoinPoint 参数就代表了织入增强处理的连接点。JoinPoint 里包含了如下几个常用方法

1) Object[] getArgs()：返回执行目标方法时的参数
2) Signature getSignature()：返回被增强的方法的相关信息
3) Object getTarget()：返回被织入增强处理的目标对象
4) Object getThis()：返回 AOP 框架为目标对象生成的代理对象

2、Spring AOP 采用和 AspectJ 一样的优先顺序来织入增强处理：在进入连接点时，具有最高优先级的增强处理将先被织入；在退出连接点时，具有最高优先级的增强处理会最后被织入

3、当不同切面里的两个增强处理需要在同一个连接点被织入时，Spring AOP 将以随机顺序来织入这两个增强处理。如果应用需要指定不同切面类里增强处理的优先级，有如下两种方案

- 可以让切面类实现 org.springframework.core.Ordered 接口，实现该接口只需要实现一个 int getOrder()方法，该方法的返回值越小，优先级越高
- 直接使用@Order 注解来修饰一个切面类，使用@Order 注解时可指定一个 int 类型的 value 属性，该属性值越小，优先级越高

8.4.5.8. 定义切入点

1、定义切入点，其实质就是为一个切入点表达式起一个名称，从而允许在多个增强处理中重用该名称

2、Spring AOP 只支持将 Spring Bean 的方法执行作为连接点，所以可以把切入点看成所有能和切入点表达式匹配的 Bean 方法

3、切入点定义包含两个部分

1) 一个切入点表达式
2) 一个包含名字和任意参数的方法签名

- 在@AspectJ 风格的 AOP 中，切入点签名采用一个普通的方法定义(方法体通常为空)来提供，且该方法的返回值必须为 void，切入点表达式需要使用@Pointcut 注解来标注

8.4.5.9. 切入点指示符

1、定义切入点表达式时大量使用了 executiong 表达式，其中 execution 就是一个切入点指示符

2、Spring AOP 仅支持部分 AspectJ 的切入点指示符，但 Spring AOP 还额外支持一个 bean 切入点指示符

3、Spring AOP 只支持使用方法调用作为连接点，所以 Spring AOP 的切入点指示符仅匹配方法执行的连接点

4、Spring AOP 支持的切入点指示符，详见 P666

5、bean 切入点表达式是 Spring AOP 额外支持的，并不是 AspectJ 所支持的切入点指示符，这个指示符对 Spring 框架来说非常实用，可以明确指定为 Spring 的哪个 Bean 织入增强处理

8.4.5.10. 组合切入点表达式

1、Spring 支持使用如下三个逻辑运算符来组合切入点表达式

1) &&
2) ||
3) !

8.4.6. 基于 XML 配置文件的管理方式

1、相比之下，XML 配置方式的优点

- 采用 XML 配置方式对早期的 Spring 用户来说更加习惯，而且这种方式允许使用纯粹的 POJO 来支持 AOP

2、XML 配置方式的缺点

- 使用 XML 配置方式不能将切面、切入点、增强处理等封装到一个地方。如果需要查看切面、切入点、增强处理，必须同时结合 Java 文件和 XML 配置文件来查看；但使用@AspectJ 时，则只需要一个单独的类文件即可看到切面、切入点和增强处理的全部信息
- XML 配置方式比@AspectJ 方式有更多的限制

3、@Aspect 还有一个优点，就是能被 Spring AOP 和 AspectJ 同时支持，容易迁移，因此@AspectJ 风格会有更大的吸引力

4、在 Spring 配置文件中，所有的切面、切入点和增强处理都必须定义在<aop:config.../>元素内部。<bean.../>元素下可以包含多个<aop:config.../>元素，一个<aop:config>可以包含 pointcut、advisor 和 aspect 元素，且这三个元素必须按照此顺序来定义

8.4.6.1. 配置切面

1、定义切面使用<aop:aspect.../>元素，使用该元素来定义切面时，其实质是将一个已有的 Spring Bean 转换成切面 Bean，所以需要先定义一个普通的 Spring Bean

2、切面 Bean 可以当成一个普通的 Spring Bean 来配置，所以完全可以为该切面 Bean 配置依赖注入。当切面 Bean 定义完成后，通过在<aop:aspect.../>元素中使用 ref 属性来引用该 Bean，就可以将 Bean 转换成一个切面 Bean 了

3、配置<aop:aspect.../>元素可以指定如下三个属性

1) id：定义该切面的标识
2)
3) ref：用于将 ref 属性所引用的普通 Bean 转换为切面 Bean
4) order：指定该切面 Bean 的优先级，该属性的作用与前面@AspectJ 中的@Order 注解、Ordered 接口的作用完全一样，order 属性值越小，该切面对应的优先级更高

4、Spring 支持将切面 Bean 当成普通 Bean 来管理，所以完全可以利用依赖注入来管理切面 Bean，管理切面 Bean 的属性值、依赖关系等

8.4.6.2. 配置增强处理

1、与使用@AspectJ 完全一样，使用 XML 一样可以配置 Before、After、AfterReturning、AfterThrowing 和 Around 五种增强处理，而且完

全支持和@AspectJ 完全一样的语义
2、使用 XML 配置增强处理分别依赖于如下几个元素
1) <aop:before.../>：配置 Before 增强处理
2) <aop:after.../>：配置 After 增强处理
3) <aop:after-returning.../>：配置 AfterReturning 增强处理
4) <aop:after-throwing.../>：配置 AfterThrowing 增强处理
5) <aop:around.../>：配置 Around 增强处理
- 这些元素都不支持使用子元素，但可以指定如下属性

1) pointcut：该元素指定一个切入点表达式，Spring 将在匹配该表达式的连接点时织入增强处理
2) pointcut-ref：该属性指定一个已存在的切入点名称，通常 pointcut 和 pointcut-ref 两个属性选其一
3) method：该属性指定一个方法名，指定将切面 Bean 的该方法转换为增强处理
4) throwing：该属性只对<after-throwing.../>元素有效，用于指定一个形参名，AfterThwoing 增强处理方法可通过该形参访问目标方法的返回值
5) returning：该属性只对<after-returning.../>元素有效，用于指定一个形参名，AfterReturning 增强处理方法可通过该形参访问目标方法的返回值

3、XML 配置方式不再使用简单的&&、||、!作为组合运算符，而是使用 and、or、not 作为组合运算符
4、示例详见 P671-672

8.4.6.3. 配置切入点

1、类似于@AspectJ 方式，允许定义切入点来重用切入点表达式，XML 配置方式也可通过定义切入点来重用切入点表达式，Spring 提供<aop:pointcut.../>元素来定义切入点
- 当把<aop:pointcut.../>元素作为<aop:config.../>的子元素定义时，表明该切入点可被多个切面共享
- 当把<aop:pointcut.../>元素作为<aop:aspect.../>的子元素定义时，表明该切入点只能在该切面中有效

2、配置<aop:pointcut.../>元素时通常需要指定如下两个属性
- id：指定该切入点的标识
-
- expression：指定该切入点关联的切入点表达式

8.5. Spring3.1 新增的缓存机制

1、Spring3.1 新增了全新的缓存机制，这种缓存机制与 Spring 容器无缝地整合在一起，可以对容器中任意 Bean 或 Bean 的方法增加缓存
2、Spring 的缓存机制非常灵活，它可以对容器中任意 Bean 或 Bean 的任意方法进行缓存，因此这种缓存机制可以在 Java EE 应用的任何层次上进行缓存
3、Spring 缓存同样不是一种具体的缓存实现方案，它底层同样需要依赖 EhCache、Guava 等具体的缓存工具。应用程序只需要面向 Spring 缓存 API 编程，

应用底层的缓存实现可以在不同的缓存实现之间自由切换，应用程序无须任何改变，只要对配置文件略作修改即可

8.5.1. 启用 Spring 缓存

1、Spring 配置文件专门为缓存提供了一个 cache:命名空间，为了启用 Spring 缓存，需要在配置文件中导入 cache:命名空间

2、导入 cache:命名空间后，启用 Spring 缓存还需要两步

1) 在 Spring 配置文件中添加<cache:annotation-driven cache-manager="缓存管理器 ID"/>，该元素指定 Spring 根据注解来启用 Bean 级别或方法级别的缓存
2) 针对不同的缓存实现配置对应的缓存管理器

8.5.1.1. Spring 内置缓存实现的配置

1、Spring 内置的缓存实现只是一种内存中的缓存，并非真正的缓存实现，因此通常只能用于简单的测试环境，不建议在实际项目中使用 Spring 内置的缓存实现

2、Spring 内置的缓存实现使用 SimpleCacheManager 作为缓存管理器，使用 SimpleCacheManager 配置缓存非常简单，直接 Spring 容器中配置该 Bean，然后通过<property.../>驱动该缓存管理器执行 setCaches()方法来设置缓存区即可

3、SimpleCacheManager 是一种内存中的缓存区，底层直接使用了 JDK 的 ConcurrentMap 来实现缓存，SimpleCacheManager 使用了 ConcurrentMapCacheFactoryBean 作为缓存区，每个 ConcurrentMapCacheFactoryBean 配置一个缓存区

8.5.1.2. EhCache 缓存实现的配置

1、在配置 EhCache 缓存实现之前，首先需要将 EhCache 缓存的 JAR 包添加到项目的类加载路径中(只要将 Hibernate 解压路径下 lib\optional\ehcache\路径下的 ehcache-core-2.4.3.jar 和 slf4j-api-1.6.1.jar 复制到项目类加载路径下即可)

- ehcache-core-2.4.3.jar 是 EhCache 的核心 JAR 包
- slf4j-api-1.6.1.jar 则是该缓存工具所使用的日志工具

2、为了使用 EhCache，需要在应用的类加载路径下添加一个 ehcache.xml 配置文件

3、Spring 使用 EhCacheCacheManager 作为 EhCache 缓存实现的缓存管理，因此只要该对象配置在 Spring 容器中，它就可作为缓存管理器使用，但 EhCacheCacheManager 底层需要依赖一个 net.sf.ehcache.CacheManager 作为实际的缓存管理器

- 为了将 net.sf.ehcache.CacheManager 纳入 Spring 容器的管理之下，Spring 提供了 EhCacheManagerFactoryBean 工厂 Bean，该工厂 Bean 实现了 FactoryBean<CacheManager>接口
- 当程序把 EhCacheManagerFactoryBean 部署在 Spring 容器中，并通过 Spring 容器请求获取该工厂 Bean 时，实际返回的是它的产品---也就是 CacheManager 对象

8.5.2. 使用@Cacheable执行缓存

1、@Cacheable 可用于修饰类或修饰方法

- 当使用@Cacheable 修饰类时，用于告诉 Spring 在类级别上进行缓存---程序调用该类的实例的任何方法都需要缓存，而且共享同一个缓存区
- 当使用@Cacheable 修饰方法时，用于告诉 Spring 在方法级别上进行缓存---只有当程序调用该方法时才需要缓存

8.5.2.1. 类级别的缓存

1、使用@Cacheable 修饰类时，就可控制 Spring 在类级别进行缓存，这样当程序调用该类的任意方法，只要传入的参数相同，Spring 就会缓存

2、缓存的意思是：当程序第一次调用该类的实例的某个方法时，Spring 缓存机制会将该方法返回的数据放入指定缓存区---就是@Cacheable 注解的 value 属性值所指定的缓存区。以后程序调用该类的实例的任何方法时，只要传入的参数下相同，Spring 将不会真正执行该方法，而是直接利用缓存区中的数据

3、类级别的缓存默认以所有方法参数作为 key 来缓存方法返回的数据---同一个类不管调用哪个方法，只要调用方法时传入的参数相同，Spring 都会直接利用缓存区中的数据(那方法的逻辑含义不同，即使参数一样返回也可能不一样啊，这种情况怎么办???)

4、@Cacheable 可指定的属性

- value：必须属性
- key：通过 SpEL 表达式显式指定缓存 key
- condition：该属性指定一个返回 boolean 值的 SpEL 表达式，只有当该表达式返回 true，Spring 才会缓存方法返回值
- unless：该属性指定一个返回 boolean 值的 SpEL 表达式，当该表达式返回 true 时，Spring 就不缓存方法返回值

8.5.2.2. 方法级别的缓存

1、使用@Cacheable 修饰方法时，就可控制 Spring 在方法级别进行缓存，这样当程序调用该方法时，只要传入的参数相同，Spring 就会使用缓存

8.5.3. 使用@CacheEvict清除缓存

1、被@CacheEvict 注解修饰的方法可用于清除缓存，使用@CacheEvict 注解时可以指定如下属性

- value：必须属性，用于指定该方法用于清除哪个缓存区的数据
- allEntries：该属性指定是否清空整个缓存区
- beforeInvocation：该属性指定是否在执行方法之前清除缓存，默认是在方法成功完成后才清除缓存
- condition：该属性指定一个 SqEL 表达式，只有当表达式为 true 时才清除缓存
- key：通过 SqEL 表达式显式指定缓存的 key

8.6. Spring 的事务

1、Spring 的事务管理不需要与任何特定的事务 API 耦合，对不同的持久层访问

技术，编程式事务提供了一致的事务编程风格

8.6.1. Spring 支持的事务策略

1、Java EE 应用的传统事务由两种策略：全局事务和局部事务

- 全局事务由应用服务器管理，需要底层服务器的 JTA(Java Transaction API)支持
- 局部事务和底层所采用的持久化技术有关，当采用 JDBC 持久化技术时，需要使用 Connection 对象来操作事务；采用 Hibernate 持久化技术时，需要使用 Session 对象来操作事务

2、全局事务可以跨多个事务性资源(典型例子是关系数据库和消息队列)；使用局部事务，应用服务器不需要参与事务管理，因此不能保证跨多个事务性资源的事务的正确性

3、当采用传统的事务编程策略时，程序代码必然和具体的事务操作代码耦合，这样造成的后果是：当应用程序需要在不同的事务策略之间切换，开发者必须手动修改程序代码。如果使用 Spring 事务管理策略，就可以改变这种现状

4、Spring 事务策略是通过 PlatformTransactionManager 接口体现的，该接口是 Spring 事务策略的核心，包含以下三个方法

1) TransactionStatus getTransaction(TransactionDefinition definition) throws TransactionException;
   - TransactionStatus 对象表示一个事务，TransactionStatus 被关联在当前执行的线程上
   - 返回的 TransactionStatus 对象，可能是一个新事务，也可能是一个已经存在的事务对象。如果当前执行的线程已经处于事务管理下，则返回当前线程的事务对象；否则，系统将新建一个事务对象后返回
2) void commit(TransactionStatus status) throws TransactionException;
3) void rollback(TransactionStatus status) throws TransactionException;

- PlatformTransactionManager 接口有许多不同的实现类，应用程序面向平台无关的接口编程，当底层采用不同的持久层技术时，系统只需使用不同的 PlatformTransactionManager 实现类即可---这种切换由 Spring 容器负责管理，应用程序无须与具体的事务 API 耦合，也无需与特定实现类耦合，从而将应用和持久化技术、事务 API 彻底分离开来
- TransactionDefinition 接口定义了一个事务规则，该接口必须指定如下几个属性值
  - 事务隔离：当前事务和其他事务的隔离程度，例如这个事务能否看到其他事务未提交的数据等
  - 事务传播：通常，在事务中执行的代码都会在当前事务中运行。但如果一个事务上下文已经存在，有几个选项可指定该事务性方法的执行行为。例如，在大多数情况下，简单地在现有的事务上下文中运行；或者挂起现有事务，创建一个新的事务。Spring 提供 EJB CMT(Contain Manager Transaction，事务管理容器)中所有的事务传播选项
  - 事务超时：事务在超时前能运行多久，也就是事务的最长持续时间。如果事务一直没有被提交或回滚，将在超出该时间后，系统自动回滚事务
  - 只读状态：只读事务不修改任何数据，在某些情况下，只读事务是非

常有用的优化

- TransactionStatus 代表事务本身，它提供了简单的事务执行和查询事务状态的方法，这些方法在所有的事务 API 中都是相同的

5、Spring 提供了如下两种事务管理方式

- 编程式事务管理：即使使用 Spring 的编程式事务，程序也可直接获取容器中的 transactionManagerBean，该 Bean 总是 PlatformTransactionManager 的实例，所以可以通过该接口提供的三个方法来开始事务、提交事务和回滚事务
- 声明式事务管理：无须在 Java 程序中书写任何事务操作代码，而是通过在 XML 文件中为业务组件配置事务代理(AOP 代理的一种)，AOP 为事务代理所织入的增强处理也由 Spring 提供---在目标方法执行之前，织入开始事务；在目标方法执行之后，织入结束事务

8.6.2. 使用 XML Schema 配置事务策略

1、Spring 同时支持编程式事务策略和声明式事务策略，通常推荐采用声明式事务策略，其优势如下

1) 声明式事务能大大降低开发者的代码书写量，而且声明式事务几乎不影响应用的代码，因此无论底层事务策略如何变化，应用程序都无须任何改变
2) 应用程序代码无须任何事务处理代码，可以更专注于业务逻辑的实现
3) Spring 则可以对任何 POJO 的方法提供事务管理，而 Spring 的声明式事务管理无须容器支持，可在任何环境下使用
4) EJB 的 CMT 无法提供声明式回滚规则：而通过配置文件，Spring 可指定事务在遇到特定异常时自动回滚。Spring 不仅可以在代码中使用 setRollbackOnly 回滚事务，也可以在配置文件中配置回滚规则
5) 由于 Spring 采用 AOP 方式管理事务，因此可以在事务回滚动作中插入用户自己的动作而不仅仅是执行系统默认的回滚

2、Spring 2.x 的 XML Schema 方式提供了简洁的事务配置策略，Spring 2.x 提供了 tx:命名空间来配置事务管理，tx:命名空间下提供了<tx:advice.../>元素来配置事务增强处理，一旦使用该元素配置了事务增强处理，就可以直接使用 <aop:advisor.../>元素来启用自动代理了

3、配置<tx:advice.../>元素除了需要 transaction-manager 属性指定事务管理器之外，还需要配置一个<attributes.../>子元素，该子元素里又可以包含多个 <method.../>子元素，具体关系图见 P687

4、配置<tx:advice.../>元素的重点就是配置<method.../>子元素，实际上每个 <method.../>子元素都未一批方法指定了所需的事务定义，包括事务传播属性、事务超时属性、只读事务、对指定异常回滚、对指定异常不会滚等

1) name：必选属性，与该事物语义相关的方法名，支持通配符
2) propagation：指定事务传播行为，该属性值可为 Propagation 枚举类的任一枚举值，默认为 Propagation.REQUIRED
3) isolation：指定事务隔离级别，该属性值可为 Isolation 枚举类的任一枚举值，默认为 Isolation.DEFAULT
4) timeout：指定事务超时时间(以秒为单位)，指定-1 意味着

5) 超时，默认值为-1
6) read-only：指定事务是否只读，默认 false
7) rollback-for：指定触发事务回滚的异常类(应使用全限定类名)，该属性可指定多个异常类，多个异常类之间以英文逗号隔开
8) no-rollback-for：指定不触发事务回滚的异常类(应使用全限定类名)，该属性可指定多个异常类，多个异常类之间以英文逗号隔开

8.6.3. 使用@Transactional

1、Spring 还允许将事务配置放在 Java 类中定义，这需要借助@Transactional 注解，该注解既可用于修饰 Spring Bean 类，也可用于修饰 Bean 类中的某个方法
2、如果用@Transactional 修饰 Bean 类，则表明这些事务设置对整个 Bean 类起作用，如果@Transactional 修饰 Bean 类的某个方法，表明这些事务设置只对该方法有效
3、使用@Transactional 可指定如下属性

- isolation：用于指定事务的隔离级别，默认为底层事务的隔离级别
- noRollbackFor：指定遇到特定异常时强制不回滚事务
- noRollbackForClassName：指定遇到特定的多个异常时强制不回滚事务。该属性值可以指定多个异常类名
- propagation：指定事务传播行为
- readOnly：指定事务是否只读
- rollbackFor：指定遇到特定异常时强制回滚事务
- rollbackForClassName：指定遇到特定的多个异常时强制回滚事务。该属性值可以指定多个异常类名
- timeout：指定事务的超时时长

8.7. Spring 整合 Struts2

1、虽然 Spring 也提供了自己的 MVC 组件，但一来 Spring 的 MVC 组件略嫌烦琐；而来 Struts2 的拥护者太多。因此很多项目都会选择使用 Spring 整合 Struts2 框架，而且 Spring 完全可以无缝整合 Struts2 框架，二者结合成一个更实际的 Java EE 开发平台

8.7.1. 启动 Spring 容器

1、对于使用 Spring 的 Web 应用，无须手动创建 Spring 容器，而是通过配置文件声明式地创建 Spring 容器，在 Web 应用中创建 Spring 容器有如下两种方式

1) 直接在 web.xml 文件中配置创建 Spring 容器
2) 利用第三方 MVC 框架的扩展点，创建 Spring 容器

- 第一种创建 Spring 容器的方式更常见，为了让 Spring 容器随 Web 应用的启动而自动启动，借助于 ServletContextListener 监听器即可完成，该监听器可以在 Web 应用启动时回调自定义方法---该方法就可以启动 Spring 容器

2、Spring 提供了一个 ContextLoaderListener，该监听器类实现了 ServeltContextListener 接口。该类可以作为 Listener 使用，它会在创建时自动查找 WEB-INF/下的 applicationContext.xml 文件

- 在 web.xml 文件中增加如下配置片段即可

```
<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener
    </listener-class>
</listener>
```

3、如果有多个配置文件要载入，则考虑使用<context-param.../>元素来确定配置文件的文件名，ContextLoaderListener 加载时，会查找名为 contextConfigLocation 的初始化参数。因此配置<context-param.../>时应该指定参数名为 contextConfigLocation

- 如果没有使用 contextConfigLocation 指定配置文件，则 Spring 自动查找/WEB-INF/路径下的 applicationContext.xml 配置文件
- 如果有 contextConfigLocation，则使用该参数确定的配置文件

8.7.2. MVC 框架与 Spring 整合的思考

1、对于一个基于 B/S 架构的 Java EE 应用而言，用户请求总是向 MVC 框架的控制器请求，而当控制器拦截到用户请求后，必须调用业务逻辑组件来处理用户请求。此时有一个问题：控制器应该如何获得业务逻辑组件

2、最容易想到的策略是，直接通过 new 关键字创建业务逻辑组件，然后调用业务逻辑组件的方法，根据业务逻辑方法的返回值确定结果，这是一种非常差的策略，其原因有三

1) 控制器直接创建业务逻辑组件，导致控制器和业务逻辑组件的耦合降低到代码层次，不利于高层解耦
2) 控制器不应该负责业务逻辑组件的创建，控制器只是业务逻辑组件的使用者，无须关心业务逻辑组件的实现
3) 每次创建新的业务逻辑组件将导致性能下降

3、较好的策略是工厂模式或者服务定位器模式

- 对于采用服务定位器模式，是远程访问的场景。在这种场景下，业务逻辑组件已经在某个容器中运行，并对外提供某种服务。控制器无须理会该业务逻辑组件的创建，直接调用该服务即可，但在调用之前，必须先找到该服务---这就是服务定位器的概念
- 对于轻量级的 Java EE 应用，工厂模式则是更实际的策略。在轻量级 Java EE 应用中，业务逻辑组件不是 EJB，通常就是一个 POJO，业务逻辑组件的生成通常应该由工厂负责，而且工厂可以保证该组件的实例只需一个就够了，可以避免重复实例化造成的系统开销

4、采用工厂模式，将控制器与业务逻辑组件的实现分离，从而提供更好的解耦，在采用工厂模式的访问策略中，所有的业务逻辑组件的创建由工厂负责，业务逻辑组件的运行也由工厂负责，而控制器只需要负责定位工厂实例即可

5、如果系统采用 Spring 框架，则 Spring 成为最大的工厂。Spring 负责业务逻辑组件的创建和生成，并可管理业务逻辑组件的生命周期。Spring 是一个性能非常优秀的工厂，可以生产出所有的实例，从业务逻辑组件，到持久层组件，甚至控制器组件

6、控制器如何访问到 Spring 容器中的业务逻辑组件呢?为了让 Action 访问到 Spring 中的业务逻辑组件，有以下两种策略

- Spring 容器负责管理控制器 Action，并利用依赖注入为控制器注入业务逻辑组件
- 利用 Spring 的自动装配，Action 将会自动从 Spring 容器中获取所需的业务逻辑组件

8.7.3. 让 Spring 管理控制器

1、让 Spring 容器来管理应用中的控制器，可以充分利用 Spring 的 IoC 特性，但需要将配置 Struts2 的控制器部署在 Spring 容器中，因此导致配置文件冗余

2、Struts2 的核心控制器首先拦截到用户请求，然后将请求转发给对应的 Action 处理，在此过程中，Struts2 将负责创建 Action 实例，并调用其 execute()方法，这个过程是固定的。如果把 Action 实例交由 Spring 容器来管理，而不是由 Struts2 产生，那么核心控制器如何知道调用 Spring 容器中的 Action 而不是自行创建 Action 实例呢，这个工作由 Struts2 提供的 Spring 插件来完成

3、进入 Struts2 项目的 lib 目录下，找到一个 struts2-spring-plugin-2.3.16.3.jar，这个 JAR 包就是 Struts2 整合 Spring 插件，简称 Spring 插件，将其复制到 Web 应用的 WEB-INF/lib 目录下即可

4、Spring 插件提供了一种伪 Action，在 struts.xml 文件中配置 Action 时，通常需要指定 class 属性，该属性就是用于创建 Action 实例的实现类。但 Struts2 提供的 Spring 插件允许指定 class 属性时，不再指定 Action 的实际实现类，而是指定为 Spring 容器中的 Bean ID，这样 Struts2 不再自己负责创建 Action 实例，而是直接通过 Spring 容器获取 Action 对象

- 这种整合策略的关键：当 Struts2 将请求转发给指定的 Action 时，Struts2 中的该 Action 只是一个傀儡，它只是一个代号，并没有指定实际的实现类，当然也不可能创建 Action，而隐藏在该 Action 下的是 Spring 容器中的 Action 实例---它才是真正处理用户请求的控制器
- 协作示意图见 P695
- 在这种整合策略下，处理用户请求的 Action 由 Spring 插件负责创建，但 Spring 插件创建 Action 实例时，并不是利用配置 Action 时指定的 class 属性来创建该 Action 实例，而是从 Spring 容器中取出对应的 Bean 实例完成创建的

5、当使用 Spring 容器管理 Struts2 的 Action 时，由于每个 Action 对应一次用户请求，且封装了该次请求的状态信息，所以不应将 Action 配置成单例模式，因此必须指定 scope 属性，该属性可以指定为 prototype 或 request

6、Struts2 与 Spring 的整合策略充分利用了 Spring 的 Ioc 特性，是一种较为优秀的解耦策略，这种策略也有一些不足之处，归纳起来，有如下不足之处

1) Spring 管理 Action，必须将所有的 Action 配置在 Spring 容器中，而 struts.xml 文件中还需要配置一个"伪 Action"，从而导致配置文件臃肿、冗余
2) Action 业务逻辑组件接受容器注入，将导致代码的可读性降低

8.7.4. 使用自动装配

1、在自动装配策略下，Action 还是由 Spring 插件创建，Spring 插件在创建 Action 实例时，利用 Spring 的自动装配策略，将对应的业务逻辑组件注入 Action

实例中，这种整合策略的配置文件简单，但控制器和业务逻辑组件耦合有提升到了代码层次，耦合较高
2、如果不指定自动装配，则系统默认使用按 byName 自动装配
3、所谓自动装配，即让 Spring 自动管理 Bean 与 Bean 之间的依赖关系，无需使用 ref 显式指定依赖 Bean。Spring 容器会自动检查 XML 配置文件的内容，为主调 Bean 注入依赖 Bean。自动装配可以减少配置文件的工作量，但会降低依赖关系的透明性和清晰性
4、这种方式存在如下两个缺点

1) Action 与业务逻辑组件的耦合降低到代码层次，必须在配置文件中配置与 Action 所需控制器同名的业务逻辑组件，不利于高层解耦
2) Action 接受 Spring 容器的自动装配，代码的可读性较差

8.8. Spring 整合 Hibernate

8.8.1. Spring 提供的 DAO 支持

1、DAO 模式是一种标准的 Java EE 设计模式，DAO 模式的核心思想是，所有的数据库访问都通过 DAO 组件完成，DAO 组件封装了数据库的增、删、改等原子操作。业务逻辑组件依赖于 DAO 组件提供的数据库原子操作，完成系统业务逻辑的实现
2、对于 Java EE 应用的架构，有非常多的选择，但不管细节如何改变，Java EE 应用都大致可分为如下三层

1) 表现层
2) 业务逻辑层
3) 数据持久层

3、轻量级 Java EE 架构以 Spring IoC 容器为核心，承上启下：向上管理来自表现层的 Action，向下管理业务逻辑层组件，同时负责管理业务逻辑层所需的 DAO 对象
4、DAO 组件是整个 Java EE 应用的持久层访问的重要组件，每个 Java EE 应用的底层实现都难以离开 DAO 组件的支持。Spring 对实现 DAO 组件提供了许多工具类，系统的 DAO 组件可以通过继承这些工具类完成
5、Spring 提供了一些列抽象类，这些抽象类将被作为应用中 DAO 实现类的父类。通过继承这些抽象类，Spring 简化了 DAO 的开发步骤，能以一致的方式使用数据库访问技术。不管底层采用 JDBC、JDO 还是 Hibernate，应用中都可以采用一致的编程模式
6、DAO 组件继承这些抽象基类会大大简化应用的开发，而且继承这些基类 DAO 能以一致的方式访问数据库，这意味着应用程序可以在不同的持久层访问技术中切换

8.8.2. 管理 Hibernate 的 SessionFactory

1、当通过 Hibernate 进行持久层访问时，必须先获得 SessionFactory 对象，它是单个数据库映射关系编译后的内存镜像。在大部分情况下，一个 Java EE 应用对应一个数据库，即对应一个 SessionFactory 对象
2、在纯粹的 Hibernate 访问中，应用程序需要手动创建 SessionFactory 实例，在实际开发中，希望以一种声明式的方式管理 SessionFactory 实例，直接以配置文

件来管理 SessionFactory 实例
3、Spring 的 IoC 容器正好提供了这种管理方式，它不仅能以声明式的方式配置 SessionFactory 实例，也能充分利用 IoC 容器的作用，为 SessionFactory 注入数据源引用

8.8.3. 实现 DAO 组件的基类

1、使用 Spring 容器管理 SessionFactory 之后，Spring 就可以将 SessionFactory 注入应用的 DAO 组件中，对于 Spring 与 Hibernate4 的整合，Spring 推荐调用 SessionFactory 的 getCurrentSession()方法来获取 Hibernate session
2、通常来说，所有的 DAO 组件都应该提供如下方法

1) 根据 ID 加载持久化实体
2) 保存持久化实体
3) 更新持久化实体
4) 删除持久化实体，以及根据 ID 删除持久化实体
5) 获取所有的持久化实体

3、示例详见 P704-705

8.8.4. 传统的 HibernateTemplate 和 HibernateDaoSupport

1、略过了

8.8.5. 实现 DAO 组件

1、有了之前开发的 BaseDAO 之后，只要让普通的 DAO 组件继承 BaseDao，并制定对应的泛型类型即可---每个 DAO 组件除了包含通用的 CURD 方法之外，还可能需要定义一些业务相关的查询方法，这些方法就只能定义在各自的 DAO 组件内

8.8.6. 使用 IoC 容器组装各种组件

1、至此，Java EE 应用所需的各种组件都已经出现，从 MVC 层的控制器组件，到业务逻辑组件以及持久层的 DAO 组件，这种组件并未直接耦合，组件与组件之间面向接口编程，所以需要利用 Spring 的 IoC 容器将它们组合在一起
2、从用户角度来看，用户发出 HTTP 请求，当 MVC 框架的控制器组件拦截到用户请求时，将调用系统的业务逻辑组件，业务逻辑组件则调用系统的 DAO 组件，而 DAO 组件则依赖于 SessionFactory 和 DataSource 等底层组件实现数据库访问
3、从系统实现角度来看，IoC 容器先创建 SessionFactory 和 DataSource 等底层组件，然后将这些底层组件注入给 DAO 组件，提供一个完整的 DAO 组件，并将此 DAO 组件注入给业务逻辑组件，从而提供完整的业务逻辑组件，而业务逻辑组件又被注入给控制器组件，控制器组件负责拦截用户请求，并将处理结果呈献给用户---这一系列的衔接，都由 Spring 的 IoC 容器提供实现

8.8.7. 使用声明式事务

1、Spring 的事务机制非常优秀，它允许程序员在开发过程中无需理会任何事务逻辑，等到应用开发完成后使用声明式事务进行统一的事务管理。只需要在配置文件中增加事务控制片段，业务逻辑组件的方法将会具有事务性，而且

Spring 的声明式事务支持在不同的事务策略之间自由切换
2、为业务逻辑组件添加事务只需要如下几个步骤

- 针对不同的事务策略配置对应的事务管理器
- 使用<tx:advice.../>元素配置事务增强处理 Bean，配置事务增强处理 Bean 时使用多个<method.../>子元素为不同方法指定相应的事务语义
- 在<aop:config.../>元素中使用<aop:advisor.../>元素配置自动事务代理

8.9. Spring 整合 JPA

1、虽然 Hibernate 十分优秀，而且在实际开发中拥有广泛的占有率，但 JPA 却是更高层次的规范，它的本质是一种 ORM 规范，底层可以采用任何遵循这种规范的 ORM 框架作为实现，比如 Hibernate、TopLink 等，因此 JPA 发展势头良好
2、就实际开发而言，同样推荐采用面向 JPA 的 API 编程，那么该应用底层就可以在各种 ORM 框架(包括 Hibernate)之间自由切换，因此具有更好的可扩展性

Chapter 9. 企业级应用开发的思考和策略

1、企业级应用开发的平台有很多，例如 Java EE、Dot Net、PHP 和 Ruby On Rails 等。这些平台为企业级应用开发提供了丰富的支持，都实现了企业级应用底层所需的功能：缓冲池、多线程以及持久访问等

9.1. 企业级应用开发面临的挑战

1、企业级应用的开发往往需要面对更多的问题：大量的并发访问，复杂的环境，网络的不稳定，还有外部的 Crack 行为等

9.1.1. 可扩展性、可伸缩性

1、优秀的企业级应用必须具备良好的可扩展性和可伸缩性，良好的可扩展性可允许系统动态增加新功能，而不会影响原有的功能

2、在 Java EE 应用里，大多采用 XML 文件作为配置文件，使用 XML 配置文件可以避免修改代码从而能极好地提高程序的解耦

3、使用 XML 配置文件提高解耦的方式，是目前企业级应用最常见的解耦方式，而依赖注入的方式则提供了更高层次的解耦。使用依赖注入可以将各个模块之间的调用从代码中分离出来，并通过配置文件来装配组件。依赖注入的容器有很多，例如 Spring、HiveMind 等

9.1.2. 快捷、可控的开发

1、采用更优秀、更新颖的技术，通常可以保证软件系统的性能更加稳定，例如从早期的 C/S 架构到 B/S 架构的过渡，以及从 Model1 到 Model2 的过渡等

9.1.3. 稳定性、高效性

1、企业级应用还有个显著特点：并发访问量大，访问频繁。因此稳定性，高效性是企业级信息化系统必须达到的要求

2、企业级应用必须有优秀的性能，如采用缓冲池技术---缓冲池用于保存那些创建开销大的对象，如果对象的创建开销大，花费时间长，该技术可将这些对象缓存，避免了重复创建，从而提高系统性能。典型应用是数据连接池

3、提高企业级应用性能的另一个方法是---数据缓存。但是数据缓存有缺点：数据缓存虽然在内存中，可极好地提高系统的访问速度，但缓存的数据占用了大量的内存空间，这将会导致系统的性能下降。因此，数据缓存必须根据实际硬件设施制定，最好使用配置文件来动态管理缓存的大小

9.1.4. 花费最小化、利益最大化

1、在良好的 Java EE 架构设计中，复用是一个永恒的追求目标。架构设计师希望系统中大部分的组件可以复用，甚至能让系统的整个层可以复用

9.2. 如何面对挑战

9.2.1. 使用建模工具

1、任何语言的描述都很空洞，而且具有很大的歧义性。使用图形则更加直观，而且意义更加明确。推荐使用建模工具主要出于如下两个方面的考虑

1) 用于软件开发者与行业专家之间的沟通

2) 用于软件开发者之间的沟通

2、关于建模工具，推荐采用统一建模语言：UML

9.2.2. 利用优秀的框架

1、优秀的框架本身就是从实际开发中抽取的通用部分，使用框架就可以避免重复开发通用部分。使用框架不仅可以直接使用框架中的基本组件和类库，还可以提高软件开发人员对系统架构设计的把握

2、提高生产效率

3、具有更稳定、更优秀的性能

4、更好的保值性

9.2.3. 选择性地扩展

1、开发自己的框架可以获得全部的控制权，但这也意味着需要很多资源来实现它，实现自己的框架需要开发者保证框架的稳定性及性能

9.2.4. 使用代码生成器

1、使用代码生成器可以自动生成部分程序，不但可以省去许多重复性的劳动，而且在系统开发过程中可以大大节省时间

2、代码生成器还有个最大的作用：在原型开发期间可以大量重复利用代码生成器

9.3. 常见设计模式精讲

1、设计模式的概念最早起源于建筑设计大师 Alexander 的《建筑的永恒方法》一书，在这本书中，Alexander 是这样描述模式的：模式是一条由三个部分组成的通用规则：它表示了一个特定环境、一类问题和一个解决方案之间的关系。每一个模式描述了一个不断重复发生的问题，以及该问题解决方案的核心设计

2、软件领域的设计模式也有类似的定义：设计模式是对处于特定环境下，经常出现的某类软件开发问题的，一种相对成熟的设计方案

3、**设计模式并不是一个非常高深的概念，实际上对设计模式的理解必须以足够的代码基类量作为基础，这样才能理解设计模式带来的好处**

4、根据 Erich Gamma、Richard Helm、Ralph Johnson、John Vlissides(软件设计模式的奠基人)的说法，设计模式常常被分为如下三类

1) 创建型：创建对象时，不再直接实例化对象；而是根据特定场景，由程序来确定创建对象的方式，从而保证更高的性能，更好的架构优势。创建模式主要有简单工厂模式、工厂方法、抽象工厂模式、单例模式、生成器模式和原型模式
2) 结构型：用于帮助将多个对象组织成更大的结构。结构性模式主要有适配器模式、桥接模式、组合器模式、装饰器模式、门面模式、享元模式和代理模式
3) 行为型：用于帮助系统间各对象的通信，以及如何控制复杂系统中的流程。行为型模式主要有命令模式、解释器模式、迭代器模式、中介者模式、
4) 忘录模式、观察者模式、状态模式、策略模式、模板模式和访问者模式

9.3.1. 单例模式

1、有些时候，允许自由创建某个类的实例是没有意义，还可能造成系统性能下降(因为创建对象带来的系统性能开销问题)

2、如果一个类始终只能创建一个实例，则这个类被称为单例类，这种模式就被称为单例模式

3、对于 Spring 框架而言，可以在配置 Bean 实例时指定 scope="singleton"来配置单例模式，如果配置<bean.../>元素没有指定 scope 属性，则该 Bean 实例默认是单例模式

4、Spring 推荐奖所有业务逻辑组件、DAO 组件、数据源组件等配置成单例的行为方式，因为这些组件无须保存任何用户状态，故所有客户端都可以共享这些业务逻辑组件、DAO 组件、因此推荐将这些组件配置成单例模式的行为方式

5、即使不借助 Spring 框架，也可以手动实现单例模式。

- 为了保证该类只能产生一个实例，程序不能允许自由创建该类的对象，而是只允许为该类创建一个对象。为了避免程序自由创建该类的实例，使用 private 修饰该类的构造器，从而将该类的构造器隐藏起来
- 将该类的构造器隐藏起来，则需要提供一个 public 方法作为该类的访问点，用于创建该类的对象且该方法必须使用 static 修饰
- 除此之外，该类还必须缓存已经创建的对象，否则该类无法知道是否曾经创建过实例，也就无法保证之创建一个实例。为此该类需要使用一个静态属性来保存曾经创建的实例，且该属性需要被静态方法访问，所以该属性也应该使用 static 修饰

6、单例模式主要优势有如下两个

1) 减少创建 Java 实例带来的系统开销
2) 便于系统跟踪单个 Java 实例的生命周期、实例状态等

9.3.2. 简单工厂

1、对于一个典型的 Java 应用而言，应用之中各实例之间存在复杂的调用关系(Spring 把这种调用关系称为依赖关系，例如实例 A 调用 B 实例的方法，称为 A 依赖于 B)

2、换一个角度看待这个问题

- 对于 A 对象而言，它只需要调用 B 对象的方法，并不是关心 B 对象的实现、创建过程
- 考虑让 B 类实现一个 IB 接口，而 A 类只需要与 IB 接口耦合
- A 类并不直接使用 new 关键字来创建 B 实例，而是重新定义一个工厂类：IBFactory，由该工厂类来负责创建 IB 实例
- 而 A 类通过调用 IBFactory 工厂的方法来得到 IB 的实例

3、关键

1) 将依赖的类型设计成一个接口，将主调类(A)与依赖类(B)的实现分离开，主调类(A)只与依赖接口(IB)耦合，该接口的实现对主调类透明
2) 通过工厂来创建 IB 的实例

4、简单工厂模式的优势与劣势

- 优势：让对象的调用者和对象创建过程分离，当对象调用者需要对象时，直接向工厂请求即可，避免了对象的调用者与对象的实现类以硬编码方

式耦合，以提高系统的可维护性，可扩展性

- 缺陷：当产品修改时，工厂类也要做相应的修改

9.3.3. 工厂方法和抽象工厂

1、在简单工厂模式里，系统使用工厂类生产所有产品实例(这些实例是同一个接口的不同实现)，且该工厂类决定生产哪个类的实例，即该工厂负责所有的逻辑判断、实例创建等工作

2、如果不想在工厂类中进行逻辑判断，程序可以为不同的产品类提供不同的工厂，不同的工厂类生产不同的产品

3、当使用工厂方法设计模式时，对象调用者需要与具体的工厂类耦合：当需要不同对象时，程序需要调用相应工厂对象的方法来得到所需的对象

4、对于采用工厂方法的设计架构，客户端代码成功与被调用对象的实现类分离，但带来了另一种耦合：客户端代码与不同的工厂类耦合，这依然是一个问题

5、为了解决客户端代码与不同工厂类耦合的问题，接着考虑再增加一个工厂类，该工厂类不是生产被调用对象的实例，而是生产工厂的实例，即这个工厂类不制造具体的被调用对象，而是制造不同工厂对象。这个特殊的工厂被称为抽象工厂类，这种设计方式也被称为抽象工厂模式

- 通过这种设计模式，就可以让客户端程序只需与抽象工厂耦合
- 对于 Spring 的 IoC 容器，它到底是简单工厂还是抽象工厂?---倾向于认为是抽象工厂，因为 Spring IoC 容器可以包括万象，它不仅可以管理普通 Bean 实例，也可以管理工厂实例

6、**不要过分纠结于简单工厂模式、抽象工厂模式这些概念，可以把它们统称为工厂模式。如果工厂直接生产被调用对象，那就是简单工厂模式；如果工厂生成了工厂对象，那就会升级为抽象工厂模式**

9.3.4. 代理模式

1、代理模式是一种应用非常管饭的设计模式，当客户端代码需要调用某个对象时，客户端实际上也不关心是否准确得到该对象，它只要一个能提供该功能的对象即可，此时就可返回该对象的代理(Proxy)

2、在这种设计方式下，系统会为某个对象提供一个代理对象，并由代理对象控制源对象的引用。代理就是一个 Java 对象代表另一个 Java 对象采取行动。在某些情况下，客户端代码不想活不能直接调用被调用者，代理对象可以在客户和目标对象之间起到中介作用

3、对客户端而言，它不能分辨出代理对象和真实对象的区别，它也无须分辨代理对象和真实对象的区别。客户端代码并不知道真正的被代理对象，客户端代码面向接口编程，它仅仅支持一个被代理对象的接口

4、只要客户端代码不能或者不想直接访问被调用对象---例如需要创建一个系统开销很大的对象，或者被调用对象在远程主机上，或者目标对象的功能还不足以满足需求，而是额外创建一个代理对象返回给客户端使用，那么这种设计方式就是代理模式

5、大图片的例子

- 通过代理，只有真正要访问该图片对象时，才会通过代理对象去创建该图片对象的实例，而创建代理对象本身的开销很小
- 把创建 BigImage 推迟到真正需要它时才创建，这样能保证前面程序运行

的流畅性，而且能减小 BigImage 在内存中的存活时间，从宏观上节省了系统的内存开销

- 在有些情况下，也许程序永远不会真正创建 BigImage 对象，在这种情况下，使用代理模式可以显著提高系统运行性能---这正是 Hibernate 延迟加载所采用的设计模式(当 A 实体和 B 实体之间存在关联关系时，Hibernate 默认采用延迟加载，当系统加载 A 实体时，A 实体关联的 B 实体并未被加载出来，A 实体所关联的 B 实体全部是代理对象---只有等到 A 实体真正需要访问 B 实体时，系统才会去数据库里抓取 B 实体所对应的记录)

6、代理模式还有另一种常用场景：当目标对象的功能不足以满足客户端需求时，系统可以为该对象创建一个代理对象，而代理对象可以增强原目标对象的功能
7、借助于 Java 提供的 Proxy 和 IvocationHandler，可以实现在运行时生成动态代理的功能，而动态代理对象就可以作为目标对象使用，而且增强了目标对象的功能---JDK 动态代理只能创建指定接口的动态代理
8、动态代理可以非常灵活地实现解耦，通过这种动态代理，程序就为被代理对象增加了额外的功能，但是每一次增加额外的功能，都必须修改代理的代码
9、这种动态代理在 AOP(Aspect Orient Program，面向切面编程)里被称为 AOP 代理，AOP 代理可以代替目标对象，AOP 代理包含了目标对象的全部方法
10、Spring AOP 更加灵活，当 Spring 定义 InvocationHandler 类的 invoke 时，它并没有以硬编码方式解决调用哪些拦截器，而是通过配置文件来决定 invoke()方法中要调用哪些拦截器，这就实现了更彻底的解耦---当程序需要为目标对象扩展新功能时，根本无须改变 Java 代理，只需要在配置文件中增加更多的拦截器配置即可

9.3.5. 命令模式

1、考虑这样一种场景：某个方法需要完成一个功能，完成这个功能的大部分步骤已经确定，但可能有少量具体步骤无法确定，必须等到执行方法时才可以确定
2、上述要求看起来有点奇怪：这个方法不仅要求参数可以变化，甚至要求方法执行体的代码也可以变化，需要能把"处理行为"作为一个参数传入该方法
3、要求把"处理行为"作为参数传入该方法，而"处理行为"用编程来实现就是一段代码。在 Java 语言中，方法不能独立存在，所以实际传入该方法的是一个对象，该对象通常是某个接口的匿名实现类的实例，该接口通常被称为命令接口，这种设计方式也被称为命令模式
4、Java 8 新增了 Lambda 表达式功能，Java 8 允许使用 Lambda 表达式创建函数式接口的实例。所谓函数式接口，
的是只包含一个抽象方法的接口

9.3.6. 策略模式

1、策略模式用于封装系列的算法，这些算法通常被封装在一个被称为 Context 的类中，客户端程序可以自由选择一种算法，或让 Context 为客户端选择一个最佳的算法---使用策略模式的优势是为了支持算法的自由切换
2、使用策略模式可以让客户端代码在不同的策略之间切换，但也有一个小小的遗憾：客户端代码需要和不同的策略类耦合，为了弥补这个不足，可以考虑使

用配置文件来指定使用哪种策略，这就彻底分离了客户端代码和具体的策略
3、说白了就是定义一个接口，然后不同的具体策略就是不同的实现类，不同的实现类向上转型为接口类型，因此，实现类的具体类型就对面向接口编程的代码透明

9.3.7. 门面模式

1、随着系统的不断改进和开发，它们会变得越来越复杂，系统会生成大量的类，这使得程序流程更难被理解。门面模式可以为这些类提供一个简化的接口，从而简化访问这些类的复杂性，有时这种简化可能降低访问这些底层类的灵活性，但除了要求特别苛刻的客户端之外，它通常可以提供所需的全部功能，当然，那些苛刻的用户仍然可以直接访问底层的类和方法
2、门面模式(Facade)也被称为正面模式，外观模式，这种模式用于将一组复杂的类包装到一个简单的外部接口中
3、简而言之，可以如下理解

1) A 对象要完成一个复杂的功能，需要依次按顺序调用 B 对象、C 对象、D 对象的某方法，即 B、C、D 对象共同提供了完整的一个功能
2) 将调用 B 对象、C 对象、D 对象的方法封装成一个方法，供 A 对象来调用
3) 如果不采用门面模式，客户端，也就是 A 对象，需要自行决定需要哪些类的哪些方法，这会增加客户端编程的复杂度

4、使用与不使用门面模式的程序结构图：P750
5、Java EE 应用里使用的业务逻辑组件来封装 DAO 组件也是典型的门面模式---每个业务逻辑组件都是众多 DAO 组件的门面，系统的控制器类无须直接访问 DAO 组件，而是由业务逻辑方法来组合多个 DAO 方法完成所需功能，而 Action 只需与业务逻辑组件交互即可

9.3.8. 桥接模式

1、桥接模式是一种结构模式，它主要对应的是：由于实际的需要，某个类具有两个或两个以上的维度的变化，如果只是使用继承将无法实现这种需要，或者使得设计变得相当臃肿
2、举例来说，假设需要为某个餐厅制造菜单，餐厅提供牛肉面、猪肉面...而且顾客可根据自己的口味选择是否添加辣椒。此时就产生了一个问题，如何应对这种变化：是否需要定义辣椒牛肉面、无辣牛肉面、辣椒猪肉面、无辣猪肉面四个子类?如果餐厅还提供羊肉面、韭菜面....这样会导致一直忙于定义子类
3、为了解决这个问题可以使用桥接模式，桥接模式的做法是把变化部分抽象出来，使变化部分与主类分离开，从而将多个维度的变化彻底分离，最后提供一个管理类来组合不同维度上的变化
4、桥接模式在 Java EE 框架中有非常广泛的用途，由于 Java EE 应用需要实现跨数据库的功能，程序为了在不同数据库之间迁移，因此系统需要在持久化技术上这个维度存在改变；除此之外，系统也需要在不同业务逻辑实现之间迁移，因此也需要在逻辑实现这个维度上存在的改变，这正好符合桥接模式的使用场景。因此 Java EE 应用都会推荐使用业务逻辑组件和 DAO 组件分离的结构，让 DAO 组件负责持久化技术这个维度上的改变，让业务逻辑组件负责业务逻辑实现这个维度上的改变
5、同一段代码可能蕴含了多种不同的设计模式，每种设计模式的理解角度不同

9.3.9. 观察者模式

1、观察者模式定义了对象之间的一对多依赖关系，让一个或多个观察者对象观察一个主题对象。当主题对象的状态发生变化时，系统能通知所有的依赖于此对象的观察者对象，从而使得观察者对象能够自动更新

2、在观察者模式中，被观察的对象常常被称为目标或主题(Subject)，依赖的对象被称为观察者(Observer)

3、观察者模式通常包含如下 4 个角色

1) 被观察者的抽象基类：它通常会持有多个观察者对象的引用。Java 提供了 java.util.Observable 基类来代表被观察者的抽象基类，所以实际开发中无须自己开发这个角色
2) 观察者接口：该接口是所有观察对象应该实现的接口，通常它只包含一个抽象方法 update()。Java 同样提供了 java.util.Observer 接口来代表观察者接口，实际开发中也无须开发该角色
3) 被观察者实现类：该类继承 Observable 基类
4) 观察者实现类：实现 Observer 接口，实现 update()抽象方法

4、事实上完全可以把观察者接口理解成监听接口，而被观察者对象也可当成事件源来处理---Java 事件机制底层的实现，本身就是通过观察者模式来实现的

9.4. 常见的架构设计策略

9.4.1. 贫血模型

1、贫血模型是最常用的应用架构，也是最容易理解的架构

2、所谓贫血，只 Domain Object 只是单纯的数据路，

包含业务逻辑方法，即每个 Domian Object 类只包含相关属性，并为每个属性提供基本的 setter 和 getter 方法，所有的业务逻辑都由业务逻辑组件实现，这种 Domian Object 就是所谓的贫血的 Domian Object，采用这种 Domain Object 的架构即所谓的贫血模型

2、贫血 Domain Object 只是单纯的数据体，类似于 C 语言的数据结构，贫血模型相当于抛弃了 Java 面向对象的性质

3、贫血模型的分层非常清晰。Domian Object 并不具备领域对象的业务逻辑功能，仅仅是 ORM 框架持久化所需的持久化实体类，仅是数据载体。贫血模型容易理解，便于开发，但背离了面向对象的设计思想，所有的 Domain Object 并不是完整的 Java 对象

4、总结起来，贫血模型存在如下缺点

- 项目需要书写大量的贫血类，当然也可以借助某些工具自动生成
- Domian Object 的业务逻辑得不到体现，由于业务逻辑对象的复杂度大大增加，许多不应该由业务逻辑对象实现的业务逻辑方法，完全由业务逻辑对象实现，从而使得业务逻辑对象的实现变得相当庞大

5、贫血模型的优点

- 开发简单、分层清晰，架构明晰且不易混淆
- 所有的依赖都是单向依赖，解耦优秀
- 适合于初学者以及对架构把握不十分清晰的开发团队

9.4.2. 领域对象模型

1、根据完整的面向对象规则，每个 Java 类都应该提供其相关的业务方法，如果在系统中设计更完备的 Domain Object 对象，则 Domain Object 不再是单纯的数据载体，Domain Object 包含了相关的业务逻辑方法

2、业务逻辑方法很多，哪些业务逻辑方法应该放在 Domain Object 对象中实现，哪些业务逻辑方法完全由业务逻辑对象实现?

1) 可重用度高，与 Domain Object 密切相关的业务方法应该放在 Domain Object 对象中实现

3、Rich Domain Object 模型主要的问题是业务逻辑组件比较复杂---业务逻辑组件需要作为 DAO 组件的门面，而且还需要包装 Domain Object 的业务逻辑方法，暴露这些业务逻辑方法，让 Action 组件可以调用这些方法

4、为了简化业务逻辑对象的开发，Rich Domain Object 模型可以有如下两个方向的改变

1) 合并业务逻辑对象与 DAO 对象
2) 合并业务逻辑对象和 Domain Object

5、模型示意图：P762

9.4.3. 合并业务逻辑对象与 DAO 对象

1、在这种模型下 DAO 对象不仅包含了各种 CRUD 方法，而且还包含各种业务逻辑方法。此时的 DAO 对象，已经完成了业务逻辑对象所有任务，变成了 DAO 对象和业务逻辑对象混合体。此时，业务逻辑对象依赖 Domain Ojbect，既提供基本的 CRUD 方法，也提供相应的业务逻辑方法

2、这种模型也导致了 DAO 方法和业务逻辑方法混合在一起，显得职责不够单一，软件分层结构不够清晰，而且使业务逻辑对象之间交叉依赖，容易产生混乱，未能做到彻底简化

3、模型示意图：P763

9.4.4. 合并业务逻辑对象和 Domain Object

1、在这种架构下，所有的业务逻辑都应该被放在 Domain Object 里面，而此时的业务逻辑层不再是传统的业务逻辑层，它仅仅封装了事务和少量逻辑，不再提供任何业务逻辑的实现。而 Domain Object 依赖于 DAO 对象执行持久化操作，此处 Domain Object 和 DAO 对象形成双向依赖。在这种设计思路下，业务逻辑层变得非常"薄"，它的功能也变得非常微弱。如果将事务控制、权限控制等逻辑以 AOP 形式织入到 Domain Object，那就可以取消业务逻辑层

2、在这种设计架构下，几乎不再需要业务逻辑层，而 Domain Object 则依赖 DAO 对象完成持久化操作，因此 Domain Object 必须接受 IoC 容器的注入，而 Domain Object 获取容器容器注入的 DAO 对象，通过 DAO 对象完成持久化操作

3、这种架构的优点

1) 整个应用几乎不需要业务逻辑层，即使需要业务逻辑组件，该业务逻辑组件也非常简单，只提供简单的事务控制、权限控制等通用逻辑、业务逻辑对象无须依赖 DAO 对象

4、这种架构的缺点

1) 业务逻辑组件和 Domain 组件功能混在一起，不容易管理，容易导致架构混乱
2) 如果使用业务逻辑对象提供事务封装性，业务逻辑层必须对所有的 Domain Object 的逻辑提供相应的事务封装，因此业务逻辑对象必须重新定义 Domain Object 实现的业务逻辑，其工作相当繁琐，因此，一般建议彻底摒弃业务逻辑层

5、模型示意图：P764

9.4.5. 抛弃业务逻辑层

1、在 Rich Domain Object 架构的各种变化中，虽然努力简化业务逻辑对象，但业务逻辑对象依然存在，使用业务逻辑对象始终是 DAO 组件访问的门面

2、抛弃业务逻辑层也有两种形式

- Domain Object 彻底取代业务逻辑对象
- 由控制器直接调用 DAO 对象

9.4.5.1. Domain Object 完全取代业务逻辑对象

1、如果业务逻辑对象的作用仅仅只是提供事务封装，因此业务逻辑对象存在的必要性不是很大，考虑对 Domain Object 的业务逻辑方法增加事务管理、权限控制等，而 Web 层的控制器则直接依赖于 Domain Object

2、这种架构更加简化，Domain Object 与 DAO 组件形成双向依赖，而 Web 层的控制器直接调用 Domain Object 的业务逻辑方法

3、这种架构的优点：分层少、代码实现简单

4、这种架构的缺点

1) 业务逻辑对象的所有业务逻辑都在 Domain Object 中实现，不易管理
2) Domain Object 必须直接传递到 Web 层，从而将持久化 API 直接传递到 Web 层，因此可能引发一些意想不到的问题

9.4.5.2. 控制器完成业务逻辑

1、在这种模型里，控制器直接调用 DAO 对象的 CRUD 方法，通过调用基本的 CRUD 方法，完成对应的业务逻辑方法。在这种模型下，业务逻辑对象的功能由控制器完成。事务则推迟到控制器中完成，因此对控制器的 execute()方法增加事务控制即可

2、对基本的 CRUD 操作，控制器可以直接调用 DAO 对象的方法，省略了业务逻辑对象的封装，这就是这种模型的最大优势。对于业务逻辑简单(当业务逻辑只是大量的 CRUD 操作时)的项目，使用这种模型也未尝不是一种很好的选择

3、这种模型导致控制器变得臃肿，因为每个控制器除了包含原有的 execute()方法外，还必须包含所需要的业务逻辑方法的实现。极大地省略了业务逻辑层的开发，避免了业务逻辑对象不得不大量封装基本的 CRUD 方法的弊端

4、这种架构的缺点极为明显，很少被采用

1) 因为没有业务逻辑层，对于那些需要多个 DAO 参与的复杂业务逻辑，在控制器中必须重复实现，其效率低，也不利于软件重用
2) Web 层的功能不再清晰，Web 层的控制器相当复杂。Web 层不仅负责实现控制器逻辑，还需要完成业务逻辑的实现，因此必须精确控制何时调

用 DAO 方法控制持久化

重要的关系示意图

1) Servlet 的生命周期：P112
2) Model2 的流程：P176
3) MVC 框架的异常处理流程：P223
4) toType 参数和转换方向之间的关系：P296
5) 转换方向和方法之间的对应关系：P298
6) Struts2 类型转换中错误处理流程：P302
7) Struts2 执行数据校验的流程图：P334
8) 拦截器与 Action 之间的关系：P346
9) Spring 容器中 Bean 实例的生命周期：P588
10) Bean 后处理器两个方法的回调时机：P617
11) AOP 代理的方法与目标对象的方法：P652
12) <aop:config.../>各子元素的关系：P668
13) <tx:advice.../>元素关系：P687
14) 工厂模式顺序图：P694
15) Spring 管理 Action 的协作图：P695
16) 轻量级 Java EE 应用架构：P701
17) Java EE 应用组件之间的调用关系：P710
18) 抽象工厂模式的 UML 类图：P736
19) 桥接模式类图：P752
20) Rich Domain Object 的组件关系图：P762

问题

1) 为什么面向对象编程和关系数据库之间有矛盾，什么叫做关系数据库