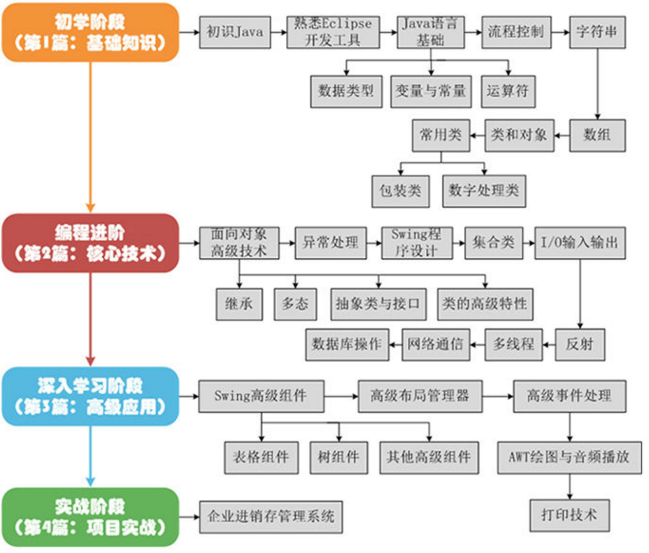
# Java EE笔记



## Java 精通

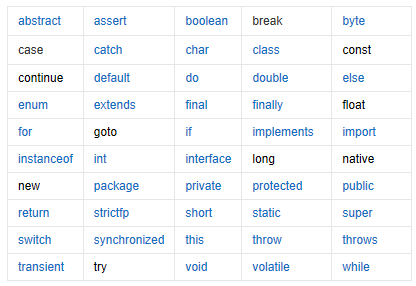
### 第一章java基础

#### 1.1 java基础语法

##### 1.1.1 java 标识符

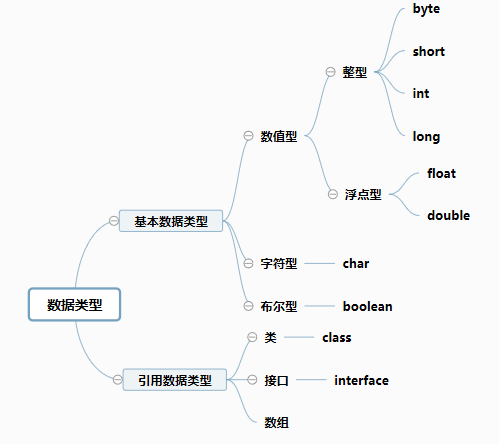
|  |  |  |
| --- | --- | --- |
| Java命名规范 | 示例 | 说明 |
| 包的命名 | scnu.cn.edu.controllor | 全部小写，由域名组成 |
| 类的命名 | GraphicsTest | 单词首字母大写 |
| 方法的命名 | drawImage | 首字母小写，其后单词首字母大写 |
| 常量的命名 | MAX\_VALUE | 全部大写，下划线分词 |
| 参数命名 | fValue | 同方法的命名 |
| 变量命名 | s\_sentence或者strSentence（常用） | 匈牙利命名，类型\_变量名|类型变量名 |

##### 1.1.2 java 关键字



#### 1.2 java数据类型

##### 1.2.1 java 数据类型分类



##### 1.2.2 java数据类型空间占用

|  |  |
| --- | --- |
| 类型 | 占内存大小 |
| byte | 1B |
| short | 2B |
| int | 4B |
| long | 8B |
| float | 4B |
| double | 8B |

#### 1.3 java 控制流程

##### 1.3.1 java运算符

算术运算符，关系运算符，逻辑运算符

|  |  |  |
| --- | --- | --- |
| 算术运算符 | 关系运算符 | 逻辑运算符 |
| + 加法 | == 等于 | && 与（短路） |
| * 减法 | != 不等于 | || 或（短路） |
| * 乘法 | > 大于 | ! 非 |
| / 除法 | < 小于 | & 与（非短路） |
| % 取模 | >= 大于等于 | | 或（非短路） |
| -- 自减 | <= 小于等于 |  |
| ++ 自加 | expression ? true:false(三元) |  |

所谓的“短路”，就是判断一半如果就能确定真假，后面的表达式就不会判断了。比如“&&”操作，如果前面的是假，整个表达式就是假，“&&”后面的表达式就不会判断。

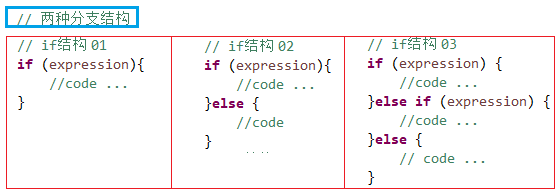
移位运算符：

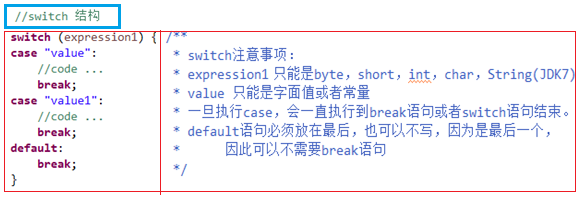
|  |  |
| --- | --- |
| & 按位与 | 1<<3 1\*23 左移 |
| | 按位或 | 8>>3 8/23 右移 |
| ^ 按位异或 | >>> 无符号右移(补0) |
| ~ 按位补(反转) |  |

##### 1.3.2 java 控制语句(6种)

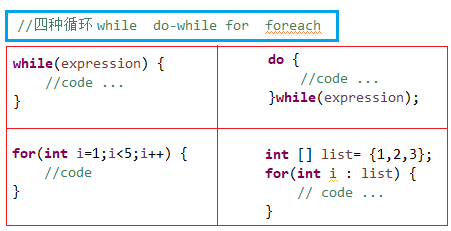
两种分支

两种分支：if语句或者switch语句。而if语句中又有三种结构。如下代码：





四种循环 while do-while for foreach



**注意**： break关键字：结束循环

continue关键字：跳出本次循环

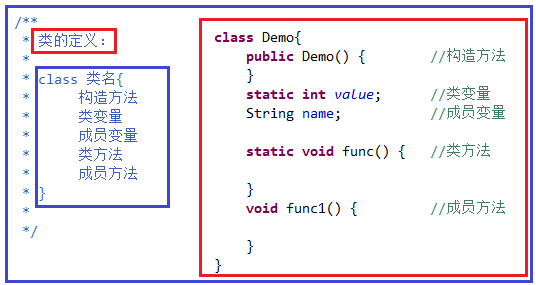
### 第二章 java进阶

#### 2.1 面向对象思想

面向对象，即是通过**消息**传递实现各个对象之间的合作，从而实现程序的功能。因此，一个对象能接收什么样的**消息**是进行面向对象设计的重心，也即是需求分析。

http://www.cnblogs.com/carsonzhu/p/5472159.html

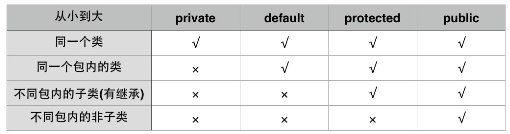
##### 2.1.1 创建类



#### 2.2 封装

##### 2.2.1 权限修饰符

***总之：private同一类内访问，default 同一包内访问，public 同一工程内访问。***

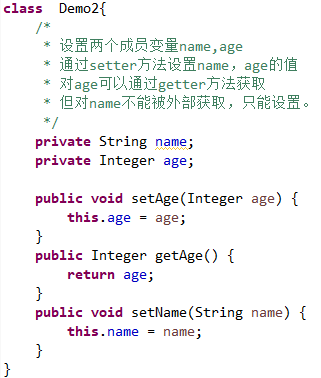


##### 2.2.2 封装性的使用

1）封装的目的：给予属性和方法权限。

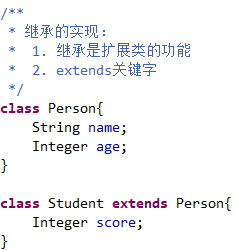
2）封装的实现（示例）：

为属性和方法通过private 进行封装，通过setter和getter方法访问。

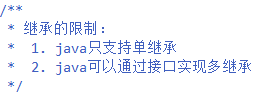


#### 2.3 继承

##### 2.3.1 继承的实现



##### 2.3.2 继承的限制

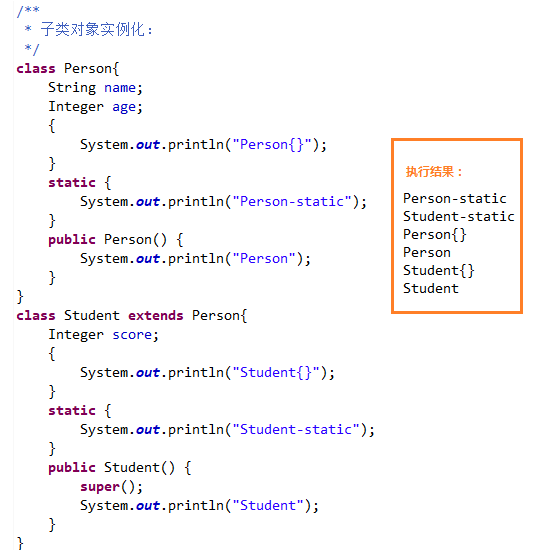


##### 2.3.3 子类对象的实例化

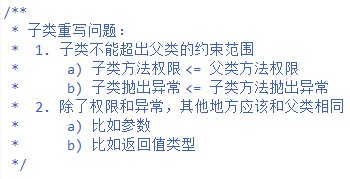
执行顺序：

1. 静态代码块 > 构造代码块 > 构造方法
2. 父类 > 子类
3. 即：

***父类静态代码块 –> 子类静态代码块 –> 父类构造代码块 –> 父类构造方法 –> 子类构造代码块 –> 子类构造方法***

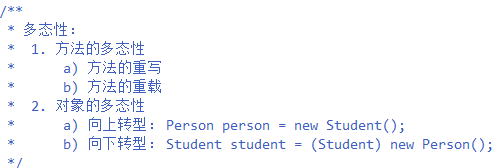


##### 2.3.4 重写



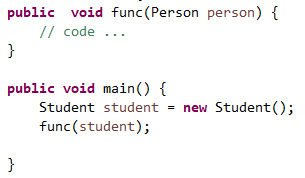
#### 2.4 多态

##### 2.4.1 多态的概念



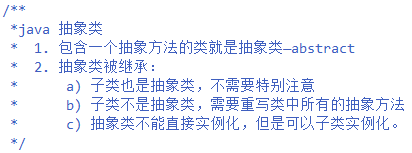
##### 2.4.2 多态的使用

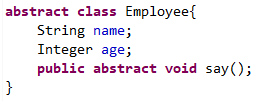
方法传递参数时，可以将参数设置为父类，传入时根据需求传入指定的子类。

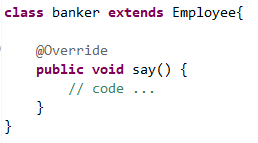


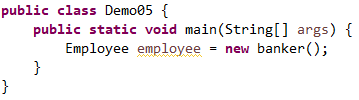
#### 2.5 接口

##### 2.5.1 java抽象类



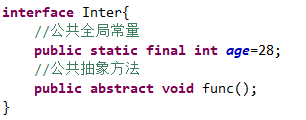




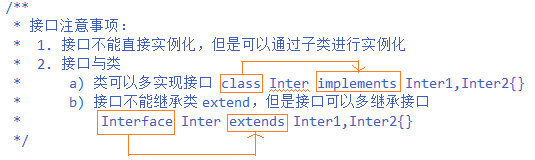


##### 2.5.2 java接口

接口的定义：



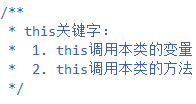
接口的特点：



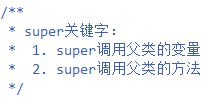
#### 2.6 类中各种关键字

|  |  |  |
| --- | --- | --- |
| 关键字 | 示例 | 说明 |
| this | this . name = name; | 本类 |
| super | super(); | 父类 |
| final | final double PI;  final Person person; | final 在指向基本类型时表示：**该基本类型是常量，比如π**  final 在指向引用类型时表示：  **该引用类型指向不变，但是内容可变。** |
| static | static int age; | **类变量，即只在类第一次实例化时创建赋值，之后的实例化不会重新创建。** |
| instanceof | 对象 instanceof 类 | 判断对象是不是该类的对象 |

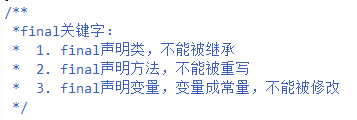
##### 2.6.1 this关键字



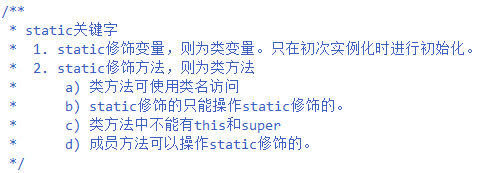
##### 2.6.2 super 关键字



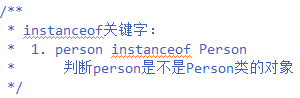
##### 2.6.3 final 关键字



##### 2.6.4 static 关键字



##### 2.6.4 instanceof 关键字



#### 2.7 其他特性与包

##### 2.7.1 匿名对象

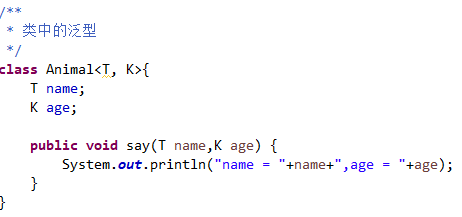


##### 2.7.2 泛型

###### 2.7.2.1 泛型的定义



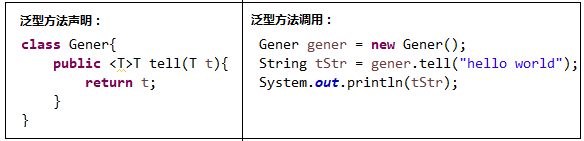
###### 2.7.2.2类/接口中的泛型



**泛型的声明与使用：**



###### 2.7.2.3 泛型方法



###### 2.7.2.4 通配符



##### 2.7.3 包



#### 2.8 常用类

##### 2.8.1 数组

###### 2.8.1.1 数组的声明



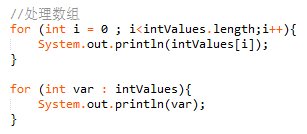
###### 2.8.1.2 数组的创建



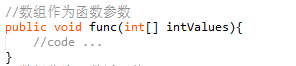
###### 2.8.1.3 数组的初始化



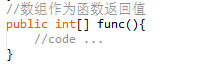
###### 2.8.1.4 数组的处理



###### 2.8.1.5 数组作为函数参数



###### 2.8.1.6 数组作为函数返回值



###### 2.8.1.7 java.util.Arrays类

Array类，其提供的方法都是类方法static

**1. 给数组赋值，fill方法**

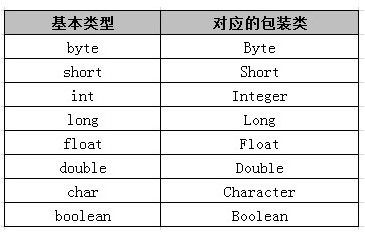
**2. 对数组排序 sort方法**

**3. 比较数组 通过equals比较是否相等**

**4. 查找数组元素：通过binarySearch执行二分法查找**

##### 2.8.2 包装类

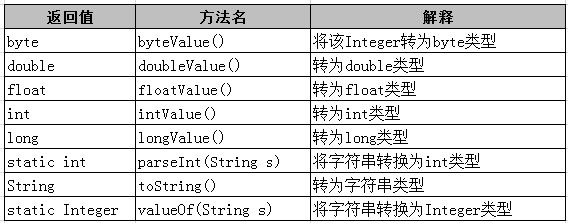
###### 2.8.2.1 基本类型与包装类的对应关系：



其中包装类主要提供了两大类型的方法：

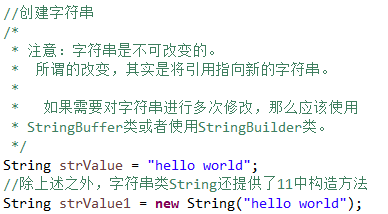
1. **将本类型和其他基本类型进行转换的方法**
2. **将字符串和本类型以及其包装类互相转换的方法**

###### 2.8.2.2 以Integer类型为例，其常用方法如下：



##### 2.8.3 String类

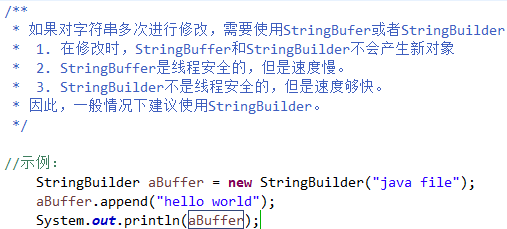
###### 2.8.3.1 String类







###### 2.8.3.2 StringBuffer和StringBuilder类



##### 2.8.4 集合类

##### 2.8.5 日期类

###### 2.8.5.1 日期处理的关键类

**Instant**——它代表的是时间戳

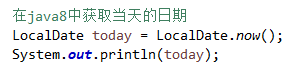
**LocalDate**——不包含具体时间的日期，比如2014-01-14。它可以用来存储生日，周年纪念日，入职日期等。

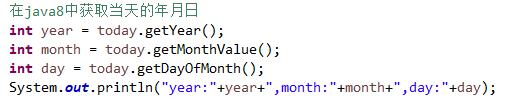
**LocalTime**——它代表的是不含日期的时间

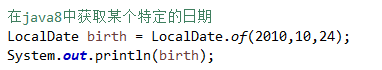
**LocalDateTime**——它包含了日期及时间，不过还是没有偏移信息或者说时区。

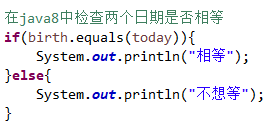
**ZonedDateTime**——这是一个包含时区的完整的日期时间，偏移量是以UTC/格林威治时间为基准的。

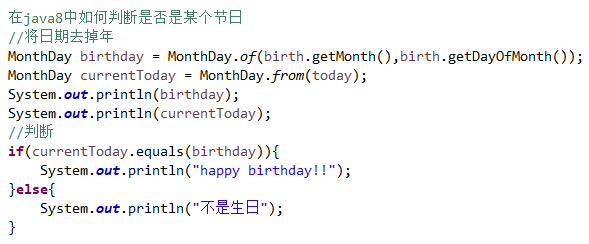
###### 2.8.5.2 日期类的测试：

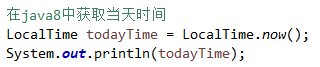


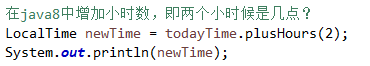


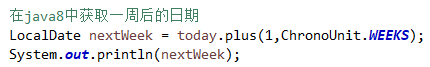


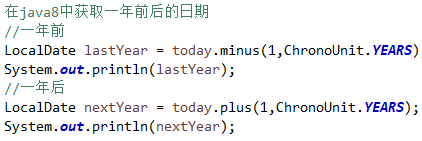












### 第三章 java高级

#### 3.1 多线程

#### 3.2 网络编程

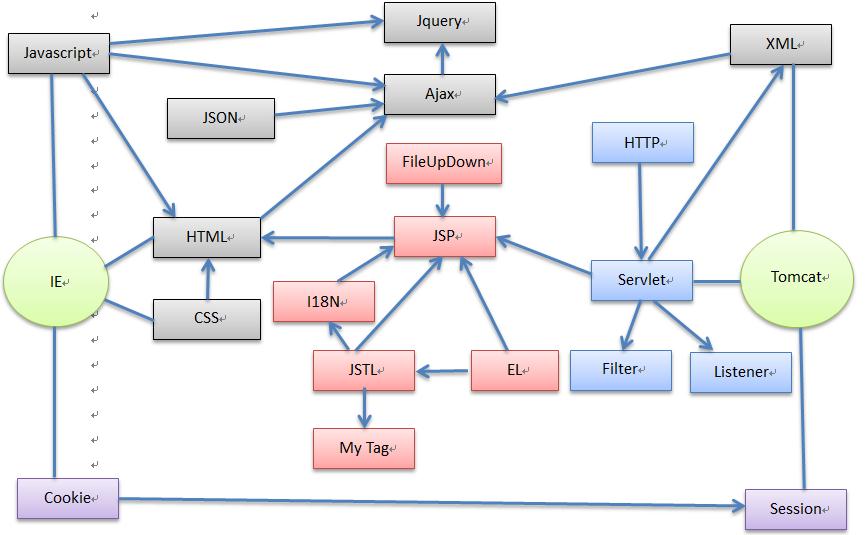
#### 3.3 反射

#### 3.4 设计模式

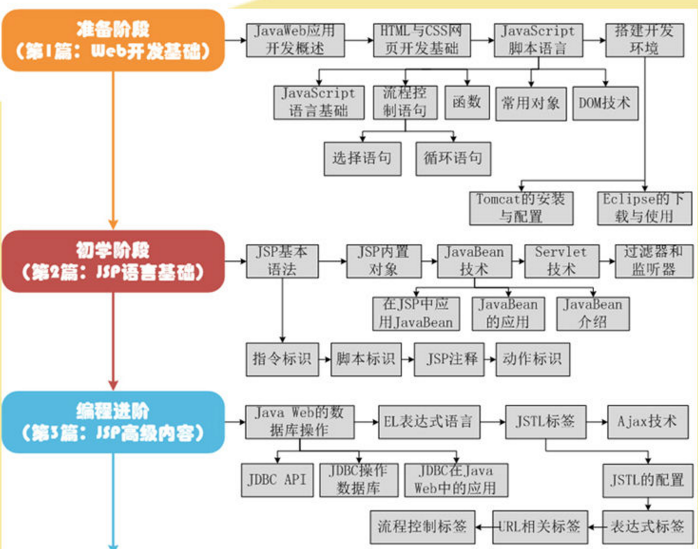
#### 3.5 GUI编程

## JAVA Web精通

JavaWEB 知识结构图



JavaWeb 学习流程图



### HTTP

#### HTTP的主要特点

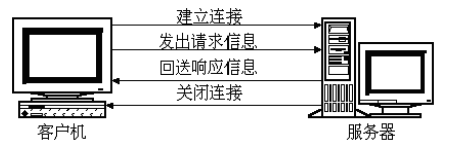
简单，快速，灵活，无连接，无状态

所谓的无连接，就是在交互后关闭连接，下一次交互会发送新的请求

所谓的无状态，就是下一次交互的页面与上一次的无关，每次请求都是独立的，这时如果下一个页面需要上一个页面的信息，就需要重新传送，这样就会导致数据量比较大，但是如果不需要上一个页面的信息，那么会反应比较快。

#### HTTP 的会话方式

##### 1.2.1 浏览器与客户端至少需要四个步骤



##### 1.2.2 HTTP请求方式1.0 与 1.1

在1.0 版本中，如果网页中带有图片之类，会开启一个新的链接下载图片。但是1.1之后都是使用同一个连接

#### HTTP请求

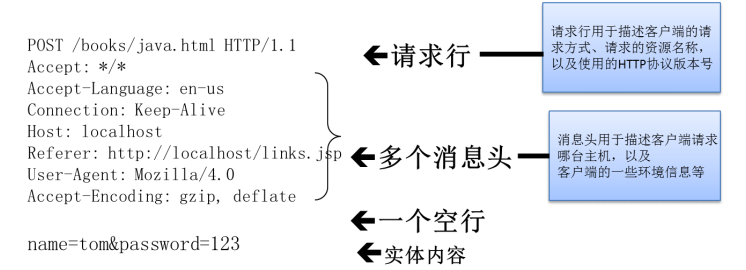
##### 一个完整的HTTP请求

一个完整的HTTP请求包括：

①请求行

②请求头

③请求体



##### 请求方法

①HTTP中定义了7中请求方式：POST，GET，HEAD，OPTIONS，DELETE，TRACE，PUT

②GET请求

GET请求是比较常用的，无论是在浏览器中输入url地址还是超链接等都是以GET的方式进行的

GET对于参数的携带都是在URL地址之后的<http://xxx.htm?name> = abc & pwd=123

从上可知，GET在携带数据上大小是有限制的，因为数据放在请求头中。

GET请求由于将数据放在请求头中，因此本身没有请求体

③POST请求

POST一般用在表单提交或者在Ajax中指定使用POST，由于POST方式是将数据放在请求体中，因此可以携带大量数据，并且大小是没有限制的。

POST请求，是将form表单提交的数据放在请求体中，发送给服务器的，因此会有请求体。

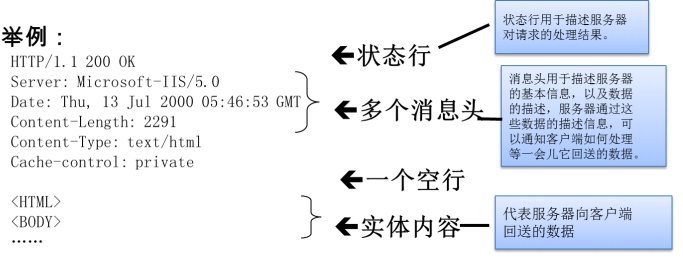
##### HTTP的响应

一个HTTP的响应也是由三部分组成的：

①响应状态行，包括协议版本，状态码等

②响应消息头，响应头

③实体内容：响应体



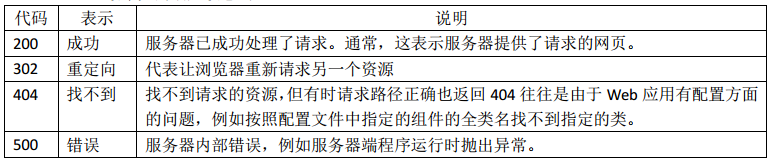
常见的响应状态码：

响应状态码以2开头通常表示成功

响应状态码以3开头通常表示转移，重定向

响应状态码以4开头通常表示无法访问，找不到或者没权限，如404

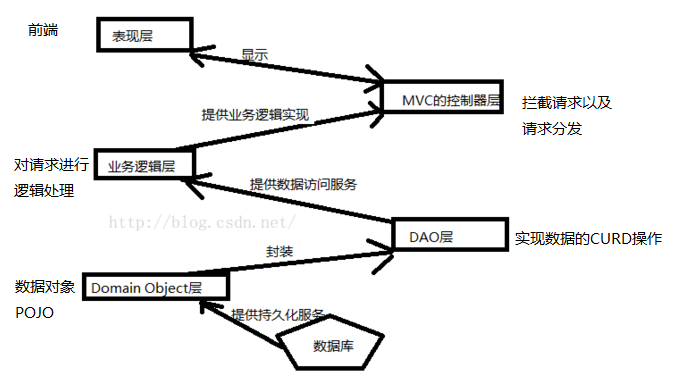
响应状态码以5开头通常表示服务器错误



### 第二章 Tomcat

### 第三章 Servlet

#### 3.1 JavaEE的分层模型



其中Servlet主要实现的是业务逻辑层

#### 3.2 Servlet的概念

①servlet是javaEE 的web组件，组件需要在容器中运行，配置web.xml

②Servlet是一个特殊的java类，是继承了HttpServlet的类

③重写HttpServlet中的方法

#### 3.3 Servlet 的示例：

①创建myServlet类并且继承HttpServlet类，并且重写doPost以及doGet方法。

②部署Servlet，即配置web.xml文件

MyServlet.java

public class MyServlet extends HttpServlet {

private static final long *serialVersionUID* = 1L;

@Override

protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {

//设置编码格式

resp.setContentType("text/html;charset=utf-8");

//获取输出流

PrintWriter out = resp.getWriter();

out.println("HTML");

}

@Override

protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {

this.doPost(req, resp);

}

}

Web.xml

<!-- 设置Servlet类 -->

<servlet>

<servlet-name>myServlet</servlet-name>

<servlet-class>sc.edu.scnu.servlet.MyServlet</servlet-class>

</servlet>

<!-- 部署Servlet，即设置访问URL -->

<servlet-mapping>

<servlet-name>myServlet</servlet-name>

<url-pattern>/first</url-pattern>

</servlet-mapping>

#### 3.4 Servlet 获取表单数据

①String getParameter(String name)，通过name 获取数据

②String[] getParameter(String name)，通过name获取多个数据，例如checkbox等。一个name对应多个数据时。

示例：

Index.jsp

<form action="/myServlet/first" method="post">

username :<input type="text" name="username"><br>

password :<input type="password" name = "pwd"><br>

<input type="checkbox" name = "favor" value="favor1">favor1<br>

<input type="checkbox" name = "favor" value="favor2">favor2<br>

<input type="checkbox" name = "favor" value="favor3">favor3<br>

<input type="submit" value="提交">

</form>

MyServlet.java —— doPost

protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {

// TODO Auto-generated method stub

String username = req.getParameter("username");

String pwd = req.getParameter("pwd");

System.out.println(username);

System.out.println(pwd);

System.err.println("---------------");

String[] strings = req.getParameterValues("favor");

for(String str:strings) {

System.out.println(str);

}

}

3.5 Servlet 控制跳转页面

① 响应重定向方式，两次请求——客户端行为

例如，我需要设计一个软件，就找到A公司，A公司说不会，然后我去找B公司，完成后，我可以对外说是B公司帮我完成的。

String username = req.getParameter("username");

String pwd = req.getParameter("pwd");

if(username.equalsIgnoreCase("admin")&&pwd.equalsIgnoreCase("admin")){

resp.sendRedirect("/myServlet/view/success.jsp");

}else {

resp.sendRedirect("/myServlet/view/fail.html");

}

②请求转发，一次请求——服务器端行为。

例如，我需要设计一个软件，就找到A公司，A公司做不了，就外包给B公司，但是对我来说，我会对外称是A公司设计的，因为B公司对我是透明的。也就是说，请求转发时，URL地址并不会发生改变。

if(username.equalsIgnoreCase("admin")&&pwd.equalsIgnoreCase("admin")){

//请求转发

//req.getRequestDispatcher("/view/fail.html").forward(req, resp);

//请求重定向

resp.sendRedirect("/myServlet/view/success.jsp");

}else {

//请求转发

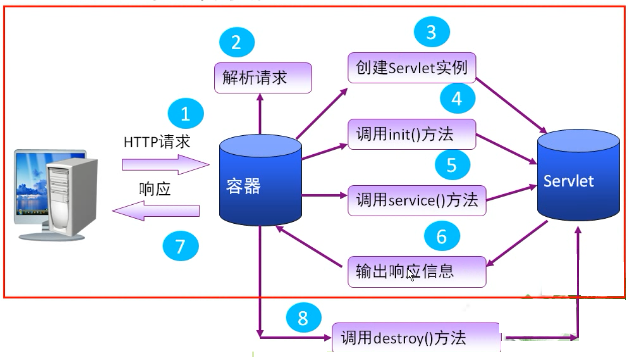
req.getRequestDispatcher("/view/fail.html").forward(req, resp);

//请求重定向

//resp.sendRedirect("/myServlet/view/fail.html");

}

#### 3.5 Servlet 的生命周期



### 第四章 JSP

#### 4.1 JSP中的概念

从本质上说，JSP也是一个Servlet，只是JSP是实现view部分，Servlet是实现逻辑控制部分。

JSP，java Servlet Page。 == HTML+java代码

在JSP中主要有两个元素：

① 脚本元素： <% java 代码%>

②表达式：<%= 需要输出的内容 %>

示例：

<!-- 在jsp中显示10个checkbox，使用jsp中的脚本以及表达式 -->

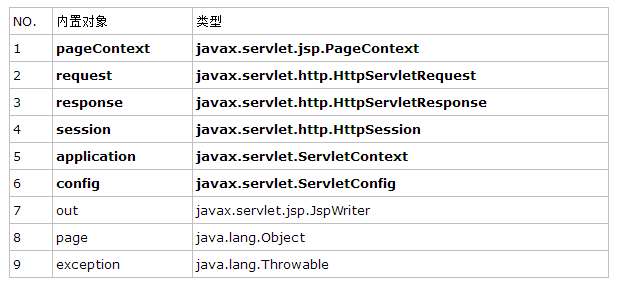
<%for(int i = 0;i<10;i++){ %>

<input type=*"checkbox"*> <%=i %>

<%} %>

#### 4.2 JSP中的内置对象

##### 4.2.1 认识JSP中的九大内置对象



##### 4.2.2 JSP中的常用内置对象详解

①request，从上表中可以看出，request对象是来自javax.http.HttpServletRequest类的。

②response

③session

④application

⑤pageContext

#### 4.3 JSP与Servlet之间传递数据（请求转发forward）

##### 4.3.1 请求参数

JSP 🡪 Servlet（getParameter或者getParameterValues）

请求参数是自动封装到请求中的。

###### 1. 表单提交时，post

###### 2. 使用URL提交，get

##### 4.3.2 请求属性

Servlet 🡪 JSP或者其他Servlet

请求属性是需要编写代码保存，还需要编写代码取出。

HttpServletREquest接口中的两个方法：

###### 1. Void setAttribute（String,Object） //Servlet通过这个方法存放数据

###### 2. Object getAttribute(String); //JSP可以通过这个方法获取到Servlet存放的数据

#### 4.4 JSP的自定义标签库

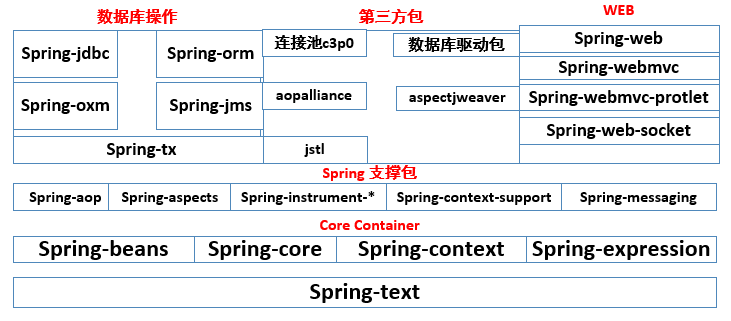
#### 4.5 JSP 的EL表达式

#### 4.6 JSP 的JSTL

## 第三节 Spring 精通

### 第一章 Spring框架

#### 1.1 Spring框架图



#### 1.2 Spring各种包的作用

Spring AOP：Spring的面向切面编程，提供AOP（面向切面编程）的实现  
Spring Aspects：Spring提供的对AspectJ框架的整合  
Spring Beans：Spring IOC的基础实现，包含访问配置文件、创建和管理bean等。  
Spring Context：在基础IOC功能上提供扩展服务，此外还提供许多企业级服务的支持，有邮件服务、任务调度、JNDI定位，EJB集成、远程访问、缓存以及多种视图层框架的支持。  
Spring Context Support：Spring context的扩展支持，用于MVC方面。  
Spring Core：Spring的核心工具包  
Spring expression：Spring表达式语言  
Spring Framework Bom：  
Spring Instrument：Spring对服务器的代理接口  
Spring Instrument Tomcat：Spring对tomcat连接池的集成  
Spring JDBC：对JDBC 的简单封装  
Spring JMS：为简化jms api的使用而做的简单封装  
Spring Messaging：  
Spring orm：整合第三方的orm实现，如hibernate，ibatis，jdo以及spring 的jpa实现  
Spring oxm：Spring对于object/xml映射的支持，可以让JAVA与XML之间来回切换  
Spring test：对JUNIT等测试框架的简单封装  
Spring tx：为JDBC、Hibernate、JDO、JPA等提供的一致的声明式和编程式事务管理。  
Spring web：包含Web应用开发时，用到Spring框架时所需的核心类，包括自动载入WebApplicationContext特性的类、Struts与JSF集成类、文件上传的支持类、Filter类和大量工具辅助类。  
Spring webmvc：包含SpringMVC框架相关的所有类。包含国际化、标签、Theme、视图展现的FreeMarker、JasperReports、Tiles、Velocity、XSLT相关类。当然，如果你的应用使用了独立的MVC框架，则无需这个JAR文件里的任何类。  
Spring webmvc portlet：Spring MVC的增强

#### Spring的环境配置

***将配置文件指定输出到classes目录下***

<build>  
 <resources>  
 <resource>  
 <directory>src/main/java</directory>  
 <includes>  
 <include>\*\*/\*.xml</include>  
 </includes>  
 </resource>  
 </resources>  
</build>

***Junit包***

<dependency>  
 <groupId>junit</groupId>  
 <artifactId>junit</artifactId>  
 <version>4.12</version>  
 <scope>test</scope>  
</dependency>

***依赖文件***

<dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-context</artifactId>  
 <version>4.3.8.RELEASE</version>  
</dependency>  
<dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-core</artifactId>  
 <version>4.3.8.RELEASE</version>  
</dependency>  
<dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-beans</artifactId>  
 <version>4.3.8.RELEASE</version>  
</dependency>  
<dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-aop</artifactId>  
 <version>4.3.8.RELEASE</version>  
</dependency>  
<dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-tx</artifactId>  
 <version>4.3.8.RELEASE</version>  
</dependency>  
<dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-expression</artifactId>  
 <version>4.3.8.RELEASE</version>  
</dependency>  
<dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-aspects</artifactId>  
 <version>4.3.8.RELEASE</version>  
</dependency>  
<dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-orm</artifactId>  
 <version>4.3.8.RELEASE</version>  
</dependency>  
<dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-oxm</artifactId>  
 <version>4.3.8.RELEASE</version>  
</dependency>  
<!-- https://mvnrepository.com/artifact/aopalliance/aopalliance -->  
<dependency>  
 <groupId>aopalliance</groupId>  
 <artifactId>aopalliance</artifactId>  
 <version>1.0</version>  
</dependency>  
  
<!-- https://mvnrepository.com/artifact/org.aspectj/aspectjweaver -->  
<dependency>  
 <groupId>org.aspectj</groupId>  
 <artifactId>aspectjweaver</artifactId>  
 <version>1.8.10</version>  
</dependency>

***测试文件：***

public class App   
{  
 public static void main( String[] args )  
 {  
  
 ApplicationContext context =   
 new ClassPathXmlApplicationContext("classpath:conf/applicationContext.xml");  
  
 Demo demo = (Demo) context.getBean(Demo.class);  
  
 demo.print();  
 }  
}

#### Spring的面向接口编程

##### 1.4.1 接口

接口，从表面上看是一些方法的集合体，有唯一的名称，可以被类和其他接口实现。

##### 1.4.2 面向接口编程

在系统分析或者架构涉及中，每个层级的程序并不直接提供程序服务，而是定义一组接口，通过实现接口来提供功能。

从分工看，接口一般由架构师设定，编程人员实现。架构师会根据架构规则、涉及规则来指定接口。

从实现上看，接口的定义时间很短，但接口的实现时间较长，调用其它人员的方法，可以采用多态的方法获取接口对象，通过接口对象调用其他人员实现的方法。可以实现共同开发。

从架构设计上看，接口实现分开，程序清晰，易读。

接口可以有多个实现：

可扩展，实现类要扩展，可以采用重新实现接口的方式

Interface dao{ class OracleDao{ class MySqlDao{

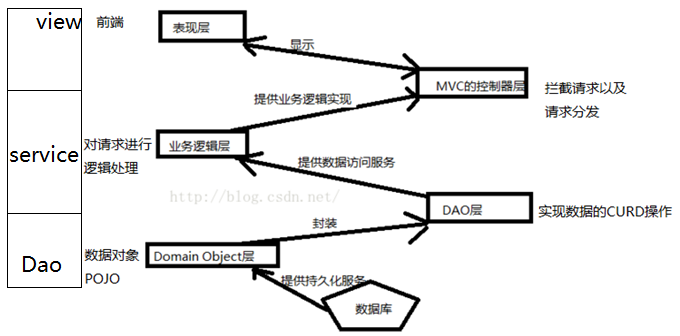
Void insert(); void insert(){} void insert(){}

} } }

易于配置，多实现可以通过配置文件配置接口的实现类。

<bean id="userDao" class="cn.edu.scnu.pojo.UserDaoImpl"/>

#### Spring的三层架构模式



### 第二章 spring 核心容器IOC

#### 2.1 bean的实例化

##### 2.1.1 构造器方式

***User.java***

public class User {  
 private Integer id;  
 private String name;  
  
 public User(Integer id, String name) {  
 this.id = id;  
 this.name = name;  
 }

}

***ApplicationContext.xml***

<bean id="user" class="cn.edu.scnu.pojo.User">  
 <constructor-arg index="0" value="1"/>  
 <constructor-arg index="1" value="mike"/>  
</bean>  
<!--  
 其中 index表示构造器的参数位置，从0开始  
 value表示 参数的值  
-->

或者

<bean id="user" class="cn.edu.scnu.pojo.User">  
 <constructor-arg name="id" value="1"/>  
 <constructor-arg name="name" value="Jhone"/>  
</bean>

##### 2.1.2 静态工厂方式

***Product.java***

public class Product {  
 public void print(){  
 System.*out*.println("hello world");  
 }  
}

***StaticProduct.java***

public class StaticFactory {  
 public ***static Product getProduct***(){  
 return new Product();  
 }  
}

***applicationContext.xml***

<bean id="staticFactory"   
 class="cn.edu.scnu.factory.***StaticFactory***"   
  ***factory-method="getProduct"***/>

##### 2.1.3 普通工厂方式

***CustomProduct.java***

public class CustomProduct {  
 public Product getProduct(){  
 return new Product();  
 }  
***}***

***applicationContext.xml***

<!--普通工厂方式-->  
<bean id="product" *factory-bean*="customProduct" ***factory-method***="getProduct"/>  
  
<bean id="customProduct" class="cn.edu.scnu.factory.CustomProduct"/>

##### 2.1.4 定义多个配置文件

<import resource="applicationContext1.xml"/>

注意:resource的值只能是配置文件的相对路径（相对于主配置文件）

#### 2.2 bean的管理

##### 2.2.1 bean的作用域

***示例：***

<bean id="demo" class="cn.edu.scnu.pojo.Demo" ***scope="singleton"***>  
 <property name="msg" value="hello world"/>  
</bean>

Spring在创建一个Bean实例时，不仅可以完成Bean的实例化，还可以为Bean指定特定的作用于。

Spring支持5中作用域：

1. ***singleton***：单例模式，在整个spring IOC容器中，只有一个Bean实例
2. ***prototype***：原型模式，每次通过getBean方法获取Bean时，都会产生一个新的Bean实例。
3. ***request***：对于每次http请求，使用request定义的Bean都将产生一个新实例。只有web应用时，该作用域才有效。
4. ***session***：对于每次Http Session，使用session定义的Bean都将产生一个新实例。
5. ***globalsession***：每个全局的http session，使用session定义的Bean都将产生一个新实例。典型情况下，仅在使用portlet context时候有效。

其中，比较常用的是singleton和prototype两种作用域。对于单例模式，一旦产生Bean，spring会追踪Bean的状态，负责维护生命周期。而***prototype每次都产生一个Bean，因此，spring仅仅new实例，并不会追踪。***

如果不指定作用域，默认singleton。***Prototype作用域Bean的撞见销毁代缴比较大，应予以避免***

##### 2.2.2 bean的初始化时间

1. 当scope=prototype：每次获取对象实例（执行getBean方法），spring都会新创建一个对象实例。

2. 当scope=singleton：根据bean标签的***lazy-init属性值选择创建时间***：

--lazy-init取值范围：默认default=false。

a) true:当每一次获取对象实例时创建对象（**getBean**）

b) false：容器加载时创建对象（**new ClassXMLPathApplicationContext**）

***注意：lazy-init只在scope=singleton时，才会生效***

***测试：通过构造方法，因为构造方法只会在对象创建的时候执行。***

***Demo.java***

public class Demo {  
  
 public Demo() {  
 System.*out*.println("hello world");  
 }

}

***applicationContext.xml***

<bean id="demo" class="cn.edu.scnu.pojo.Demo" scope="singleton">  
 <property name="msg" value="hello world"/>  
</bean>

***Test.java***

public class App   
{  
 public static void main( String[] args )  
 {  
 ApplicationContext context =  
 new ClassPathXmlApplicationContext("classpath:conf/applicationContext.xml");  
 Demo demo = (Demo) context.getBean(Demo.class);  
 Demo demo1 = (Demo) context.getBean(Demo.class);  
 Demo demo2 = (Demo) context.getBean(Demo.class);  
 }  
}

结果：只会打印一次helloworld

***如果applicationContext.xml如下，scope=”prototype“，则会打印三次hello world***

<bean id="demo" class="cn.edu.scnu.pojo.Demo" scope="prototype">  
 <property name="msg" value="hello world"/>  
</bean>

***测试scope=”singleton” lazy-init=”false”***

***applicatioContext.xml***

<bean id="demo" class="cn.edu.scnu.pojo.Demo" scope="singleton" lazy-init="false">  
 <property name="msg" value="hello world"/>  
</bean>

Test.java

public class App   
{  
 public static void main( String[] args )  
 {  
 ApplicationContext context =  
 new ClassPathXmlApplicationContext("classpath:conf/applicationContext.xml");  
 }  
}

结果：只加载容器，也会打印hello world

而如果lazy-init=true，则不会打印hello world，只有在getBean才会打印。

##### 2.2.3 bean的初始化和销毁方法

***首先创建Bean，如下：***

<bean id="demo" class="cn.edu.scnu.pojo.Demo" ***init-method="init" destroy-method="destroy"***>  
 <property name="msg" value="hello world"/>  
</bean>

Spring在***创建Bean实例后，会调用bean的初始化方法***。Spring在容器关闭后，会进入垃圾回收阶段，因此，在***容器关闭前会调用bean的销毁方法***(***只有singleton的bean才会执行销毁方法***)，因为prototype每次都会创建新bean，并且sprng并不追踪。

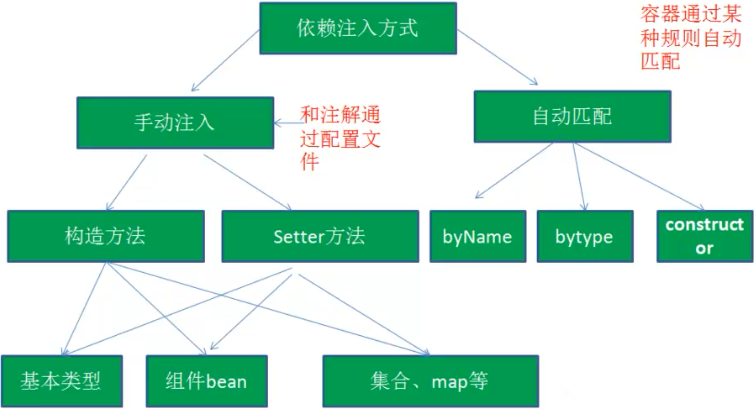
***Test.java***

public class App   
{  
 public static void main( String[] args )  
 {  
 /\*  
 因为ApplicationContext是一个接口，并且没有close方法  
 ClassPathXmlApplicationContext是有close方法的，  
 因此，直接用ClassPathXmlApplicationContext替换ApplicationContext  
 \*/  
 ClassPathXmlApplicationContext context =  
 new ClassPathXmlApplicationContext("classpath:conf/applicationContext.xml");  
// ApplicationContext context =  
// new ClassPathXmlApplicationContext("classpath:conf/applicationContext.xml");  
 */\*\*  
 \* 创建Bean后，调用初始化方法  
 \* hello world————先调用构造器  
 \* init()... ————调用初始化方法  
 \*/* Demo demo = (Demo) context.getBean(Demo.class);  
 context.close();  
 }  
}

#### 2.3 bean属性的设置

##### 2.3.1 依赖注入

Spring容器，不仅可以初始化对象，同时也可以对对象的成员变量进行赋值。



注意：建议使用手动注入，不会产生歧义。

***传统的方式：***

//男孩有一只狗

Boy boy = new Boy();

Dog dog = new Dog();

boy.setDog(dog);

***依赖注入的方式***：

<bean id=”dog”/>

<bean id=”boy”>

<property name=”dog” ref=”dog”/>

</bean>

##### 2.3.2 依赖注入之手动注入

***手动注入方式包括两种：setter方法注入和构造器方法注入***

1. ***Setter注入方式***
   1. ***基本类型注入***

***其中，基本类型包括8种基本数据类型，8种基本类型的包装类型，String以及StringBuffer类型。***

<bean id="user01" class="cn.edu.scnu.pojo.User">  
 <!--基本数据类型注入-->  
 <property name="id" value="2"/>  
 <property name="name" value="mike"/>  
  
 <!--spring组件类型注入-->  
 <property name="dog" ref="dog"/>

</bean>

* 1. ***Spring组件类型注入***

<bean id="user01" class="cn.edu.scnu.pojo.User">  
 <!--spring组件类型注入-->  
 <property name="dog" ref="dog"/>

</bean>

1. ***集合类型注入***
   1. ***List类型***

<bean id="customer" class="cn.edu.scnu.pojo.Customer">  
 <property name="lists">  
 <list>  
 <value type="java.lang.String">aa</value>  
 <ref bean="dog"/>  
 <bean class="cn.edu.scnu.pojo.Dog">  
 <property name="name" value="jingba"/>  
 </bean>  
 </list>  
 </property>

</bean>

Map类型

<!--Map类型注入，key-value-->  
<property name="maps">  
 <map>  
 <entry key="123" value="abc"/>  
 <entry key="1234" value-ref="dog"/>  
 <entry key-ref="dog" value-ref="dog"/>  
 </map>  
</property>

* 1. ***Properties类型***

<!--Properties类型注入-->

<bean>  
<property name="pros">  
 <props>  
 <prop key="username">root</prop>  
 <prop key="password">12345</prop>  
 </props>  
 </property>

</bean>

* 1. ***Set类型***

<property name="sets">  
 <set>  
 <value>1</value>  
 <ref bean="dog" />  
 <bean class="cn.edu.scnu.pojo.Dog">  
 <property name="name" value="藏獒" />  
 </bean>  
 </set>  
</property>

1. ***构造器注入方式***

构造器方法注入：spring在初始化bean组件时，调用含有参数的构造器对全局变量进行赋值。参数类型可以为基本类型、bean组件类型、集合类型。赋值方式***和setter方式的赋值方式一致***。

<bean id="user" class="cn.edu.scnu.pojo.User">  
 <!--为避免多个构造器出现歧义，建议写上type标签-->  
 <constructor-arg index="0" value="123" type="java.lang.Integer"/>  
 <constructor-arg index="1" value="mike" type="java.lang.String"/>  
 <constructor-arg type="java.util.List" index="2">  
 <list>  
 <value type="java.lang.String">aa</value>  
 <value type="java.lang.String">bb</value>  
 </list>  
 </constructor-arg>  
</bean>

##### 2.3.1 依赖注入之自动注入

通过标签autowire标签实现自动装配，比如byType，byName。

不建议使用。

#### 2.4 使用注解完成IOC配置

##### 2.4.1 注解配置文件

Spring3.0 之后，引入了组件扫描机制，可以扫描如下几个注解:

* + 1. ***@Component*** 泛化的概念，表示一个bean组件
    2. ***@Service*** 一般用于业务层，但是和①功能相同
    3. ***@Controller*** 一般用于控制层，但是和①功能相同
    4. ***@Repository*** 用于Dao层，除①功能，还封装了数据库访问异常。

可以将上面注解的类放入spring容器中管理，它的作用和xml中bean节点的配置相同。

需要引入命名空间

http://www.springframework.org/schema/context   
http://www.springframework.org/schema/context/spring-context.xsd">

添加要自动扫描的包，base-package是需要扫描的包及其子包

<context:component-scan base-package="cn.edu.scnu.pojo"/>

##### 2.4.2 使用注解配置spring组件

###### 1. 注解配置

@Component(value="employee") //如果不指定value，默认为类名首字母小写  
public class Employee {  
 private Integer id;  
 private String name;  
 private String addr;

}

###### 初始化和销毁方法配置



###### 基本类型的注入

@Component(value="employee") //如果不指定value，默认为类名首字母小写  
public class Employee {  
 @Value(value = "12")  
 private Integer id;  
 @Value(value = "mike")  
 private String name;  
 @Value(value = "beijing")  
 private String addr;  
}

注意：在属性上使用注解是可以省略setter方法的，但是不建议使用

###### spring组件的注入

@Component(value="employee") //如果不指定value，默认为类名首字母小写  
public class Employee {  
 @Resource(name = "dog")  
 private Dog dog;

}

###### 注解集合注入

### 第三章 spring 的AOP机制

#### 3.1 spring AOP的原理

##### 3.1.1 spring AOP的原理

切面：本质上是*一个方法，封装了一些通用逻辑*，可以作用到其他组件上

切入点：指定将切面用在哪些组件上。***指的是应用的地点***，spring通过切入点表达式来指定这个地点。

通知：***指定切面执行的位置***。方法前执行还是方法后，或者抛出异常后

切入点程序：可以是spring组件（Bean）中的任何方法。***我们的业务逻辑***

切面程序：是spring组件（bean）中的某个方法，要求：

public void doBefore（JoinPoint jp）{//jp为前置通知参数

//无返回类型

//参数类型和通知类型有关，可以无参数。

}

##### 3.1.2 spring AOP的环境搭建

<dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-aop</artifactId>  
 <version>4.3.8.RELEASE</version>  
</dependency>  
<!-- https://mvnrepository.com/artifact/aopalliance/aopalliance -->  
<dependency>  
 <groupId>aopalliance</groupId>  
 <artifactId>aopalliance</artifactId>  
 <version>1.0</version>  
</dependency>

<!—与AspectJ框架整合 -->

<dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-aspects</artifactId>  
 <version>4.3.8.RELEASE</version>  
</dependency>

<!-- https://mvnrepository.com/artifact/org.aspectj/aspectjweaver -->  
<dependency>  
 <groupId>org.aspectj</groupId>  
 <artifactId>aspectjweaver</artifactId>  
 <version>1.8.10</version>  
</dependency>  
<!— cglib动态代理 -->  
<dependency>  
 <groupId>cglib</groupId>  
 <artifactId>cglib</artifactId>  
 <version>3.2.5</version>  
</dependency>

#### 3.2 spring AOP的几种通知

##### 3.2.1 前置通知

目标组件执行程序前执行的通知

###### 1. 普通程序，即切入点程序

*/\*\*  
 \* 切入点程序，也就是主要的业务逻辑  
 \*/*public class Hello {  
  
 public void print(){  
 System.*out*.println("hello world");  
 }  
}

###### 2. 切面程序，比如日志处理程序等

*/\*\*  
 \* 切面，预处理方法封装，比如日志处理等  
 \*/*public class Aspect {  
  
 public void doBefore(JoinPoint jp){  
 System.*out*.println("log.....");  
 }  
}

###### 3. applicationContext.xml配置

<!--配置切入点的Bean-->  
<bean id="hello" class="cn.edu.scnu.aspectJ.Hello"/>  
<!--配置切面的Bean-->  
<bean id="aspect" class="cn.edu.scnu.aspectJ.Aspect"/>  
  
<!--配置AOP-->  
<aop:config>  
 <!--配置切入点-->  
 <aop:pointcut id="pointcut" expression="execution(\* cn.edu.scnu.aspectJ.Hello.print())"/>  
 <!--配置切面-->  
 <aop:aspect ref="aspect">  
 <!--配置通知  
 在执行pointcut切入点前执行doBefore前置通知  
 -->  
 <aop:before method="doBefore" pointcut-ref="pointcut"/>  
 </aop:aspect>  
</aop:config>

###### 4. 测试程序

@Test  
public void beforeInfo(){  
 Hello hello = (Hello) context.getBean("hello");  
 hello.print();  
}

结果:

Log…..

Hello world

##### 3.2.2 返回通知

目标组件***正常执行并返回参数***后执行的程序，因为有返回值，因此切面程序除了参数JointPoint外，还有一个存储返回值的参数。

###### 1. 普通程序，即切入点程序

*/\*\*  
 \* 切入点程序，也就是主要的业务逻辑  
 \*/*public class Hello {  
 public String getName(){  
 return "mike";  
 }  
}

###### 2. 切面程序，比如日志处理程序

*/\*\*  
 \* 切面，预处理方法封装，比如日志处理等  
 \*/*public class Aspect {  
 public void doAfterReturning(JoinPoint jp,Object result){  
 //注意返回通知，因为有返回值所以，除jp参数外，还有result参数  
 System.*out*.println("关闭数据库");  
 System.*out*.println("返回的参数为："+result);  
 }  
}

###### 3. applicationContext.xml

<!--配置切入点的Bean-->  
<bean id="hello" class="cn.edu.scnu.aspectJ.Hello"/>  
<!--配置切面的Bean-->  
<bean id="aspect" class="cn.edu.scnu.aspectJ.Aspect"/>  
<!--配置AOP-->  
<aop:config>  
 <aop:pointcut id="afterReturning" expression="execution(\* cn.edu.scnu.aspectJ.Hello.getName())"/>  
 <!--配置切面-->  
 <aop:aspect ref="aspect">  
 <!—返回通知-->  
 <aop:after-returning method="doAfterReturning" pointcut-ref="afterReturning" returning="result"/>  
 </aop:aspect>  
</aop:config>

###### 4. 测试程序

@Test  
 public void beforeInfo(){  
 Hello hello = (Hello) context.getBean("hello");  
 hello.getName();  
 }

结果：

关闭数据库

返回的参数为：mike

##### 3.2.3 异常通知

目标函数抛出异常的时候执行的通知，切面程序除了JointPoint参数外，还有抛出的异常参数Throwable。

###### 1. 普通程序，即切入点程序

public class Hello {  
 public void getNum(){  
 System.*out*.println("你好");  
 System.*out*.println(8/0);  
 }  
}

###### 2. 切面程序，比如日志处理程序

public class Aspect {  
 public void doException(JoinPoint jp,Throwable e){  
 System.*out*.println("异常信息："+e.getMessage());  
 System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  
 }  
}

###### 3. applicationContext.xml

<!--配置切入点的Bean-->  
<bean id="hello" class="cn.edu.scnu.aspectJ.Hello"/>  
<!--配置切面的Bean-->  
<bean id="aspect" class="cn.edu.scnu.aspectJ.Aspect"/>  
<!--配置AOP-->  
<aop:config>  
 <aop:pointcut id="exception" expression="execution(\* cn.edu.scnu.aspectJ.Hello.getNum())"/>  
 <!--配置切面-->  
 <aop:aspect ref="aspect">  
 <aop:after-throwing method="doException" pointcut-ref="exception" throwing="e"/>  
 </aop:aspect>  
</aop:config>

###### 4. 测试程序

@Test  
 public void beforeInfo(){  
 Hello hello = (Hello) context.getBean("hello");  
 hello.getNum();  
 }

结果:

你好

异常信息：/ by zero

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

##### 3.2.4 后置通知

目标方法执行结束之后，无论有无异常，都会执行后置通知

###### 1. 普通程序，即切入点程序

public class Hello {  
 public String getName(){  
 System.*out*.println("hello world");  
 //System.*out*.println(10/0);  
 return "mike";  
 }  
}

###### 2. 切面程序，比如日志处理程序

public class Aspect {  
  
 public void doBefore(JoinPoint jp){  
 System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*前置通知\*\*\*\*\*\*\*\*\*\*\*\*\*");  
 System.*out*.println("前置通知，在目标方法执行前执行log.....");  
  
 }  
 public void doAfterReturning(JoinPoint jp,Object result){  
 //注意返回通知，因为有返回值所以，除jp参数外，还有result参数  
 System.*out*.println("\*\*\*\*\*\*\*\*返回通知\*\*\*\*\*\*\*\*\*\*\*\*\*");  
 System.*out*.println("返回通知，正常执行，返回后执行，关闭数据库");  
 System.*out*.println("返回的参数为："+result);  
  
 }  
 public void doException(JoinPoint jp,Throwable e){  
 System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*异常通知\*\*\*\*\*\*\*\*\*\*\*\*\*");  
 System.*out*.println("异常通知，只有方法出现异常才会执行。");  
 System.*out*.println("异常信息："+e.getMessage());  
 }  
 public void doAfter(JoinPoint jp){  
 System.*out*.println("\*\*\*\*\*\*\*\*\*\*后置通知\*\*\*\*\*\*\*\*\*\*\*\*");  
 System.*out*.println("后置通知，无论与否都会执行");  
 System.*out*.println("执行位置在返回通知之后，或者异常通知之前");  
 }  
}

###### 3. applicationContext.xml

<!--配置切入点的Bean-->  
<bean id="hello" class="cn.edu.scnu.aspectJ.Hello"/>  
<!--配置切面的Bean-->  
<bean id="aspect" class="cn.edu.scnu.aspectJ.Aspect"/>  
  
<!--配置AOP-->  
<aop:config>  
 <!--配置切入点-->  
 <aop:pointcut id="pointcut" expression="execution(\* cn.edu.scnu.aspectJ.Hello.getName())"/>  
 <!--配置切面-->  
 <aop:aspect ref="aspect">  
 <!—前置通知-->  
 <aop:before method="doBefore" pointcut-ref="pointcut"/>  
 <!—返回通知-->  
 <aop:after-returning method="doAfterReturning" pointcut-ref="pointcut" returning="result"/>

<!—异常通知-->  
 <aop:after-throwing method="doException" pointcut-ref="pointcut" throwing="e"/>

<!—后置通知-->  
 <aop:after method="doAfter" pointcut-ref="pointcut"/>  
 </aop:aspect>  
</aop:config>

###### 4. 测试程序

@Test  
 public void beforeInfo(){  
 Hello hello = (Hello) context.getBean("hello");  
 hello.getName();  
 }

##### 3.2.5 环绕通知

###### 1. 普通程序，即切入点程序

public class Hello {  
 public String getName(){  
 System.*out*.println("hello world");  
 //System.*out*.println(10/0);  
 return "mike";  
 }  
}

###### 2. 切面程序，比如日志处理程序

public class Aspect {  
//  
 public void doBefore(JoinPoint jp){  
 System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*前置通知\*\*\*\*\*\*\*\*\*\*\*\*\*");  
 System.*out*.println("前置通知，在目标方法执行前执行log.....");  
  
 }  
 public void doAfterReturning(JoinPoint jp,Object result){  
 //注意返回通知，因为有返回值所以，除jp参数外，还有result参数  
 System.*out*.println("\*\*\*\*\*\*\*\*返回通知\*\*\*\*\*\*\*\*\*\*\*\*\*");  
 System.*out*.println("返回通知，正常执行，返回后执行，关闭数据库");  
 System.*out*.println("返回的参数为："+result);  
  
 }  
 public void doException(JoinPoint jp,Throwable e){  
 System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*异常通知\*\*\*\*\*\*\*\*\*\*\*\*\*");  
 System.*out*.println("异常通知，只有方法出现异常才会执行。");  
 System.*out*.println("异常信息："+e.getMessage());  
 }  
 public void doAfter(JoinPoint jp){  
 System.*out*.println("\*\*\*\*\*\*\*\*\*\*后置通知\*\*\*\*\*\*\*\*\*\*\*\*");  
 System.*out*.println("后置通知，无论与否都会执行");  
 System.*out*.println("执行位置在返回通知之后，或者异常通知之前");  
 }  
 ***//环绕通知，可以替代其他通知***  
 public void doRound(ProceedingJoinPoint pjp){  
 System.*out*.println("前置处理程序.....");  
 doBefore(pjp);  
 try{  
 //执行目标程序  
 Object result = pjp.proceed();  
 System.*out*.println("返回处理程序");  
 doAfterReturning(pjp,result);  
 }catch (Throwable e){  
 System.*out*.println("异常处理程序");  
 doException(pjp,e);  
 }  
 System.*out*.println("后置处理程序");  
 doAfter(pjp);  
 }  
}

###### 3. applicationContext.xml

<!--配置切入点的Bean-->  
<bean id="hello" class="cn.edu.scnu.aspectJ.Hello"/>  
<!--配置切面的Bean-->  
<bean id="aspect" class="cn.edu.scnu.aspectJ.Aspect"/>  
<!--配置AOP-->  
<aop:config>  
 <aop:pointcut id="pointcut" expression="execution(\* cn.edu.scnu.aspectJ.Hello.getName())"/>  
 <!--配置切面-->  
 <aop:aspect ref="aspect">  
 <aop:around method="doRound" pointcut-ref="pointcut"/>  
 </aop:aspect>  
</aop:config>

###### 4. 测试程序

@Test  
 public void beforeInfo(){  
 Hello hello = (Hello) context.getBean("hello");  
 hello.getName();  
 }

##### 3.2.6 切入点表达式

切入点表达式：用于声明spring中那些组件的函数是目标函数。

切入点表达式：

按类匹配：匹配类中所有函数作为目标函数，within关键字

<aop:pointcut id="pointcut" expression="within(cn.edu.scnu.aspectJ.\*)"/>

按函数匹配：匹配函数作为目标函数，execution关键字

<aop:pointcut id="pointcut" expression="execution(\* cn.edu.scnu.aspectJ.Hello.getName())"/>

可以使用通配符：

execution(\* cn.edu.scnu.aspectJ.\*.getName(..))

execution(\* cn..Hello.getName())

execution(\* cn..\*.getName())

按Bean的id匹配：匹配Bean中的全部函数作为目标函数。bean关键字

<aop:pointcut id="pointcut" expression="bean(hello)"/>

可以使用通配符匹配

<aop:pointcut id="pointcut" expression="bean(hell\*)"/>

#### 3.3 AOP 的注解配置

##### 3.3.1 注解配置文件

<!--开启组件自动扫描-->  
<context:component-scan base-package="cn.edu.scnu.aspectJ"/>  
<!--开启注解aop自动代理-->  
<aop:aspectj-autoproxy/>

##### 3.3.2 切入点程序，普通程序

//将切入点程序，目标程序纳入spring容器  
@Component  
public class Hello {  
 public String getName(){  
 System.*out*.println("hello world");  
 System.*out*.println(10/0);  
 return "mike";  
 }  
}

##### 3.3.3 切面程序

//首先声明这是一个Bean组件  
 //其次声明这是一个切面组件  
@Component  
@org.aspectj.lang.annotation.Aspect  
public class Aspect {  
  
 // @Before(value = "execution(\* cn.edu.scnu.aspectJ.Hello.getName())")  
 public void doBefore(JoinPoint jp){  
 System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*前置通知\*\*\*\*\*\*\*\*\*\*\*\*\*");  
 System.*out*.println("前置通知，在目标方法执行前执行log.....");  
  
 }  
// @AfterReturning(  
// value = "execution(\* cn.edu.scnu.aspectJ.Hello.getName())",  
// returning = "result"  
// )  
 public void doAfterReturning(JoinPoint jp,Object result){  
 //注意返回通知，因为有返回值所以，除jp参数外，还有result参数  
 System.*out*.println("\*\*\*\*\*\*\*\*返回通知\*\*\*\*\*\*\*\*\*\*\*\*\*");  
 System.*out*.println("返回通知，正常执行，返回后执行，关闭数据库");  
 System.*out*.println("返回的参数为："+result);  
  
 }  
// @AfterThrowing(  
// value = "execution(\* cn.edu.scnu.aspectJ.Hello.getName())",  
// throwing = "e"  
// )  
 public void doException(JoinPoint jp,Throwable e){  
 System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*异常通知\*\*\*\*\*\*\*\*\*\*\*\*\*");  
 System.*out*.println("异常通知，只有方法出现异常才会执行。");  
 System.*out*.println("异常信息："+e.getMessage());  
 }  
// @After(value = "execution(\* cn.edu.scnu.aspectJ.Hello.getName())")  
 public void doAfter(JoinPoint jp){  
 System.*out*.println("\*\*\*\*\*\*\*\*\*\*后置通知\*\*\*\*\*\*\*\*\*\*\*\*");  
 System.*out*.println("后置通知，无论与否都会执行");  
 System.*out*.println("执行位置在返回通知之后，或者异常通知之前");  
 }  
  
 @Around(value = "execution(\* cn.edu.scnu.aspectJ.Hello.getName())")  
 public void doRound(ProceedingJoinPoint pjp){  
 System.*out*.println("前置处理程序.....");  
 doBefore(pjp);  
 try{  
 //执行目标程序  
 Object result = pjp.proceed();  
 System.*out*.println("返回处理程序");  
 doAfterReturning(pjp,result);  
 }catch (Throwable e){  
 System.*out*.println("异常处理程序");  
 doException(pjp,e);  
 }  
 System.*out*.println("后置处理程序");  
 doAfter(pjp);  
 }  
}

### 第四章 spring的数据库操作

#### 4.1 spring集成jdbc—jdbcTemplate

##### 4.1.1 jdbcTemplate简介

Spring提供了一个工具类jdbcTemplate，对jdbc进行了轻量级的封装，有点如下：

1. 直接使用sql语句操作数据库，效率很高
2. 支持数据库的分区，比如银行可以每年新建一张表。不用总是使用同一张表，这是Hibernate无法实现的，因为映射文件是提前写好的。
3. 使用java语言拼sql语句，效率很高，这是ibatis无法达到的，ibaits使用配置文件虽然可以拼接sql语句，但是没有java方便，效率也不如java拼sql。

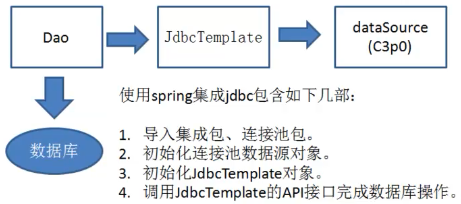
缺点如下：

1. sql语句直接写在java程序中，很难管理，但是可以进行重新封装，实现sql语句的简单管理。
2. 编码量大。

##### 4.1.2 jbdcTemplate实现原理

JdbcTemplate封装了操作数据库的各种方法，该类包含一个dataSource属性（数据源），只有在初始化数据源的情况下，才能使用jdbcTemplate。

##### 4.1.3 spring集成jdbcTemplate



###### 1. 导入jdbc集成包，连接池包以及数据库驱动包

<!--数据库驱动包-->  
<dependency>  
 <groupId>mysql</groupId>  
 <artifactId>mysql-connector-java</artifactId>  
 <version>5.1.44</version>  
</dependency>

<!--jdbc集成包-->  
<!-- https://mvnrepository.com/artifact/org.springframework/spring-jdbc -->  
<dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-jdbc</artifactId>  
 <version>4.3.8.RELEASE</version>  
</dependency>  
  
<!--事务控制包-->  
<dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-tx</artifactId>  
 <version>4.3.8.RELEASE</version>  
</dependency>  
<!--连接池包-->  
<!-- https://mvnrepository.com/artifact/com.mchange/c3p0 -->  
<dependency>  
 <groupId>com.mchange</groupId>  
 <artifactId>c3p0</artifactId>  
 <version>0.9.5.2</version>  
</dependency>

###### 2. 初始化连接池数据源

针对C3P0数据源：初始化数据源对象（ComboPooledDataSource），该对象包含连接数据库的基本属性以及连接池的配置属性。

<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">  
 <property name="driverClass" value="com.mysql.jdbc.Driver"/>  
 <property name="jdbcUrl" value="jdbc:mysql://localhost:3306/market\_lay"/>  
 <property name="user" value="root"/>  
 <property name="password" value="12345"/>  
 <!--连接池最小连接数-->  
 <property name="minPoolSize" value="1"/>  
 <!--连接池最大连接数-->  
 <property name="maxPoolSize" value="10"/>  
 <!--最大连接时间，如果超过时间没操作就会踢出连接池-->  
 <property name="maxIdleTime" value="100"/>  
 <property name="initialPoolSize" value="3"/>  
 <!--每次申请数据库的连接数量-->  
 <property name="acquireIncrement" value="3"/>  
</bean>

###### 3. 初始化jdbcTemplate对象

***方法一：由于java中继承为单继承，这样本类就不能继承其他类了。***

//通过继承JdbcTemplate类。这样本类就是jdbcTemplate的子类  
//可以直接通过this关键字调用本类方法  
public class UserDaoImpl extends JdbcTemplate implements UserDao {  
 public void insert(User user) {  
 this.execute();  
 }

}

***方法二：***

public class UserDaoImpl implements UserDao {  
 //通过创建jdbcTemplate对象，并提供getter和setter方法  
 //通过spring容器将jdbcTemplate对象注入到本类中。  
 //可以通过 this.getJdbcTemplate().batchUpdate();使用  
 private JdbcTemplate jdbcTemplate;  
 public JdbcTemplate getJdbcTemplate() {  
 return jdbcTemplate;  
 }  
 public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {  
   
 this.jdbcTemplate = jdbcTemplate;  
 }  
 public void update(User user) {  
 this.getJdbcTemplate().batchUpdate();  
 }  
}

###### 4. 配置文件注入

<bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">  
 <property name="dataSource" ref="dataSource"/>  
</bean>  
<bean id="userDao" class="cn.edu.scnu.jdbcTemplate.dao.UserDaoImpl">  
 <property name="jdbcTemplate" ref="jdbcTemplate"/>  
</bean>

###### 5. 更新添加删除操作，update（sql，Array）

public void insert(User user) {  
 String sql = "INSERT INTO users(id,name,age,addr,phone) VALUES (?,?,?,?,?)";  
 Integer num = this.getJdbcTemplate().update(sql,new Object[]{  
 user.getId(),  
 user.getName(),  
 user.getAge(),  
 user.getAddr(),  
 user.getPhone()  
 });  
 //返回值num为修改的记录条数  
 System.*out*.println(num);  
}

###### 6. 单条记录的查询

public User findById(Integer id) {  
 return this.jdbcTemplate.query("select \* from users where id = ?", new Object[]{  
 id  
 }, new ResultSetExtractor<User>() {  
 public User extractData(ResultSet resultSet) throws SQLException, DataAccessException {  
 if(resultSet.next()){  
 User user = new User();  
 user.setId(resultSet.getInt("id"));  
 user.setName(resultSet.getString("name"));  
 user.setAge(resultSet.getInt("age"));  
 user.setAddr(resultSet.getString("addr"));  
 user.setPhone(resultSet.getString("phone"));  
 return user;  
 }  
 return null;  
 }  
 });  
  
}

###### 7. 多条记录的查询

public List<User> findAll() {  
 return this.getJdbcTemplate().query("select \* from users", new RowMapper<User>() {  
 public User mapRow(ResultSet resultSet, int i) throws SQLException {  
 //会自动生成一个集合并把下面的每一个user存到List users中  
 //并且会自动迭代，不需要判断resultSet.next();  
 User user = new User();  
 user.setId(resultSet.getInt("id"));  
 user.setName(resultSet.getString("name"));  
 user.setAge(resultSet.getInt("age"));  
 user.setAddr(resultSet.getString("addr"));  
 user.setPhone(resultSet.getString("phone"));  
 return user;  
 }  
 });  
}

###### 8. 增删改操作

public void delete(Integer id) {  
 String sql = "DELETE FROM users WHERE id = ?";  
 Object[] args = {id};  
 int changeNum = jdbcTemplate.update(sql,args);  
 if(changeNum>0){  
 System.*out*.println("删除成功");  
 }else{  
 System.*out*.println("删除失败");  
 }  
}  
  
public void update(User user) {  
 String sql = "UPDATE users SET name=?,age=? where id= ?";  
 Object[] args = {user.getName(),user.getAge(),user.getId()};  
 int changeNum = jdbcTemplate.update(sql,args);  
 if(changeNum>0){  
 System.*out*.println("更新成功");  
 }else{  
 System.*out*.println("更新失败");  
 }  
  
}  
  
public void add(User user) {  
 System.*out*.println("----------");  
 String sql = "INSERT INTO users(id,name,age,addr,phone) " +  
 "VALUES (?,?,?,?,?)";  
 Object[] args={user.getId(),user.getName(),user.getAge(),user.getAddr(),user.getPhone()};  
 int changeNum = jdbcTemplate.update(sql,args);  
 if(changeNum>0){  
 System.*out*.println("插入成功");  
 }else{  
 System.*out*.println("插入失败");  
 }  
}

###### 9. 查询方法第二种实现

class UserRowMapper implements RowMapper {    
     public Object mapRow(ResultSet rs, int index) throws SQLException {    
         User user = new User();    
     
         user.setId(rs.getString("user\_id"));    
         user.setName(rs.getString("name"));    
         user.setSex(rs.getString("sex").charAt(0));    
         user.setAge(rs.getInt("age"));    
     
         return user;    
     }    
 }    
  //多条查询  
 public List findAllByRowMapperResultReader() {    
     String sql = "SELECT \* FROM USER";    
     return jdbcTemplate.query(sql, new RowMapperResultReader(new UserRowMapper()));    
 }    
   
 //单条查询  
 public User getUser(final String id) throws DataAccessException {    
     String sql = "SELECT \* FROM USER WHERE user\_id=?";    
     final Object[] params = new Object[] { id };    
     List list = jdbcTemplate.query(sql, params, new RowMapperResultReader(new UserRowMapper()));    
     
     return (User) list.get(0);    
 }

#### 4.2 spring的事务控制

##### 4.2.1 事务概述

事务管理的是一组SQL，这一组只要有一条SQL语句出现错误，那么这一组操作就要全部回滚。

事务的四大特性：

1. 原子性，要么都发生，要么都不发生
2. 一致性，数据应该不被破坏
3. 隔离性，用户间操作不相混淆
4. 持久性，永久保存。

###### 1. 编程式事务

需要手动编写代码

session.BeginTransaction();//开始事务  
//sql指令，save，update，delete  
session.getTransaction().commit();//事务提交

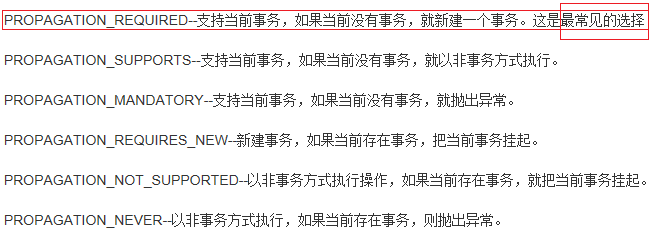
###### 2. 声明式事务

不需要手动编写代码

***注意事项：***

① spring控制事务是按照bean中***的函数为单位***，即*一个函数会默认作为一个事务，函数中有错，会将整个函数的操作回滚*。

② 事务的传播策略：如果bean的*函数存在相互调用*的情况。Spring提供了一组配置方式供开发者选择，这些配置方式就叫事务的传播策略。



③ 数据库的隔离级别：脏读，不可重复读，幻读。



④ 只读事务与读写事务，这个与具体的数据库和数据库驱动有关，有些更新操作也默认为只读事务。如果声明只读，数据库会对该事务进行一定的优化。

使用场景：

只读事务：单纯的数据库查询

读写事务：对数据进行修改的操作

##### 4.2.2 spring的声明式事务控制

###### 1. 配置事务控制管理器

org.springframework.jdbc.datasource.DataSourceTransactionManager

###### 2. 配置事务通知

配置哪些函数作为事务，以及配置其隔离级别，传播行为，是否只读。其中隔离级别isolation采用默认级别，传播行为建议required，read-only在查询函数上为true，其他为false。

###### 配置事务的切入点

切入点程序即普通业务程序，指明spring的那些组件的函数配置事务通知。在spring三层架构中，事务控制，建议放在service层。

###### 4. applicationContext.xml

<!--配置事务控制管理器-->  
<bean id = "dataSourceTransactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">  
 <property name="dataSource" ref="dataSource"/>  
</bean>  
  
<!--配置事务通知-->  
<tx:advice id="txAdvisor" transaction-manager="dataSourceTransactionManager">  
 <tx:attributes>  
 <tx:method name="add\*" isolation="DEFAULT" read-only="false"/>  
 <tx:method name="delete\*" isolation="DEFAULT" read-only="false"/>  
 <tx:method name="update\*" isolation="DEFAULT" read-only="false"/>  
 <tx:method name="find\*" isolation="DEFAULT" read-only="true"/>  
 </tx:attributes>  
</tx:advice>  
  
<!--配置切入点-->  
<aop:config>  
 <!--cn.edu.scnu.service包中所有的类-->  
 <aop:pointcut expression="within(cn.edu.scnu.service.\*)" id="txPointcut"/>  
 <aop:advisor advice-ref="txAdvisor" pointcut-ref="txPointcut"/>  
</aop:config>  
  
  
<!--  
 ***查询切入点表达式中所有的方法，如果某个符合事务通知中配置的name名称，那么就这个方法就作为一个事务纳入spring进行管理，并且安装事务通知中的配置，进行隔离级别等的处理。***  
-->

##### 4.2.3 spring使用注解完成事务控制

###### 1. 启用spring的自动扫描功能

<context:component-scan base-package="cn.edu.scnu.service"/>

###### 2. 配置事务控制管理器

<!--配置事务控制管理器-->  
<bean id = "dataSourceTransactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">  
 <property name="dataSource" ref="dataSource"/>  
</bean>  
<tx:annotation-driven transaction-manager="dataSourceTransactionManager"/>

###### 3. 在业务层上增加@Transactional

在业务层方法上添加注解，并配置要处理的方式，如隔离级别等。

public interface UserService {  
 @Transactional(propagation = Propagation.*REQUIRED*,isolation = Isolation.*DEFAULT*)  
 //readOnly默认为false  
 void addBatch(List<User> users);  
}

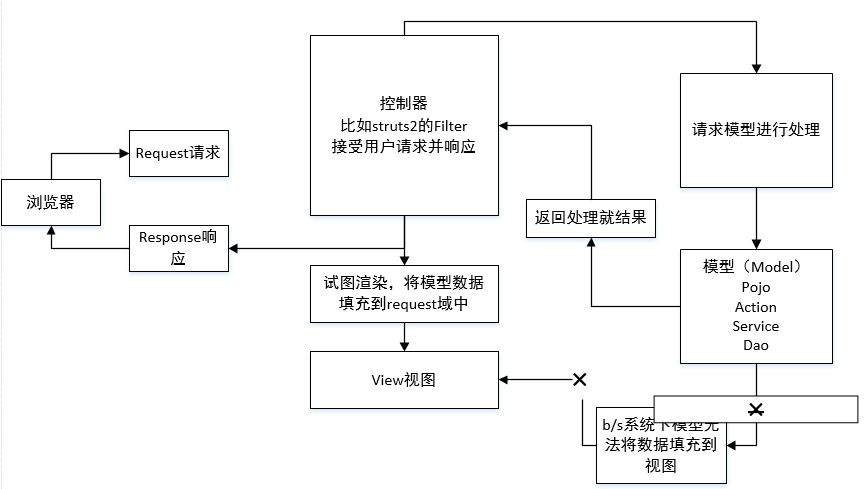
//注解放在接口上和放在实现类上效果相同

## 第四节 Spring MVC精通

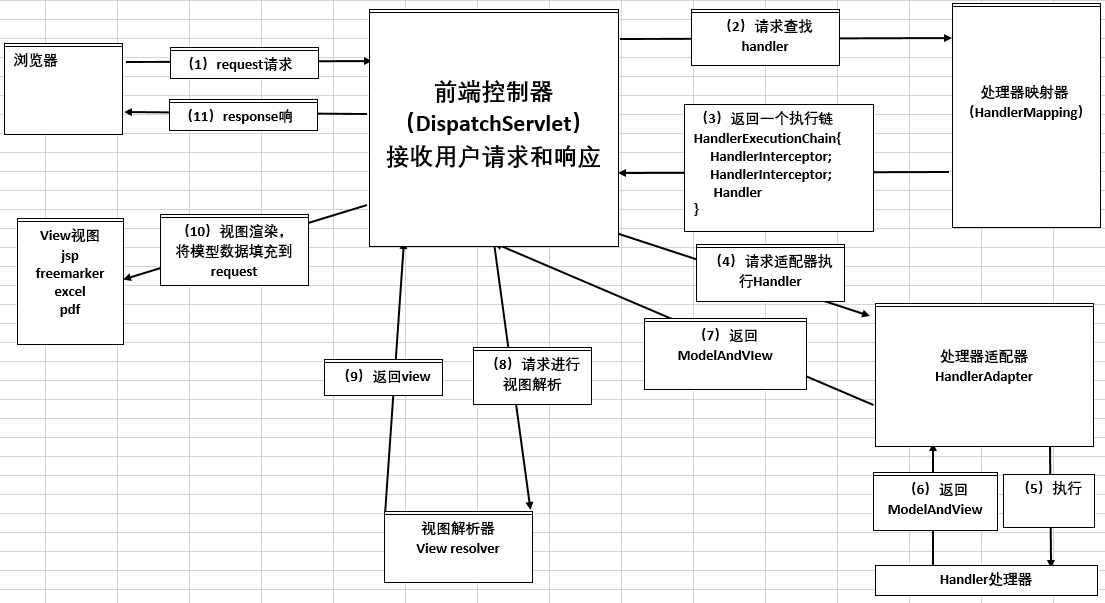
### spring mvc原理

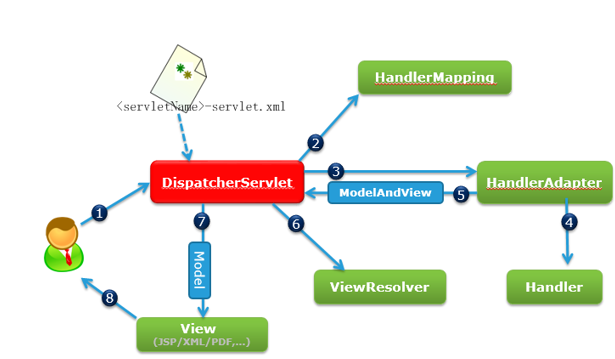
#### spring mvc 框架原理

##### mvc设计模式原理



##### 1.1.2 spring mvc模式

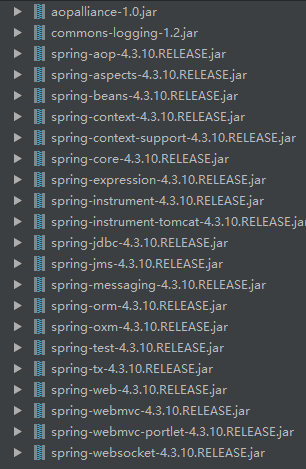




#### 1.2 spring mvc的入门程序

##### 1.2.1 环境配置

###### 1. Maven环境依赖

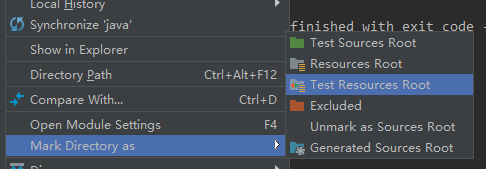


<dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-web</artifactId>  
 <version>4.3.8.RELEASE</version>  
 </dependency>  
  
 <dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-webmvc</artifactId>  
 <version>4.3.8.RELEASE</version>  
 </dependency>  
  
 <dependency>  
 <groupId>mysql</groupId>  
 <artifactId>mysql-connector-java</artifactId>  
 <version>5.1.44</version>  
 </dependency>  
  
 <dependency>  
 <groupId>javax.servlet</groupId>  
 <artifactId>jstl</artifactId>  
 <version>1.2</version>  
 </dependency>  
  
 <dependency>  
 <groupId>commons-logging</groupId>  
 <artifactId>commons-logging</artifactId>  
 <version>1.2</version>  
 </dependency>  
  
 <dependency>  
 <groupId>javax.servlet</groupId>  
 <artifactId>javax.servlet-api</artifactId>  
 <version>4.0.0</version>  
 <scope>provided</scope>  
 </dependency>  
  
 <dependency>  
 <groupId>org.apache.commons</groupId>  
 <artifactId>commons-lang3</artifactId>  
 <version>3.6</version>  
 </dependency>  
  
 <dependency>  
 <groupId>commons-fileupload</groupId>  
 <artifactId>commons-fileupload</artifactId>  
 <version>1.3.3</version>  
 </dependency>  
</dependencies>

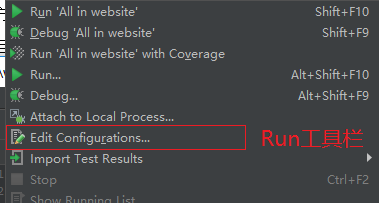
***Web.xml***

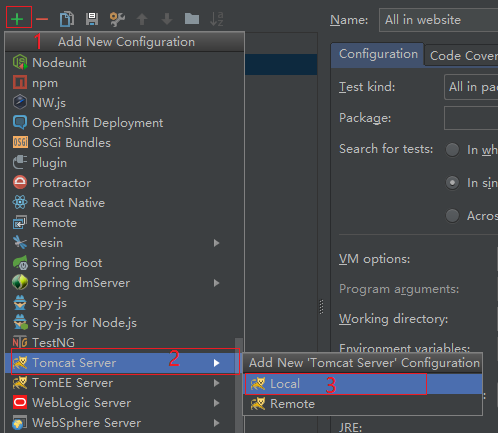
<?xml version="1.0" encoding="UTF-8"?>  
<web-app xmlns="http://java.sun.com/xml/ns/javaee"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://java.sun.com/xml/ns/javaee  
 http://java.sun.com/xml/ns/javaee/web-app\_3\_0.xsd"  
 version="3.0">  
 <display-name>springmvc</display-name>

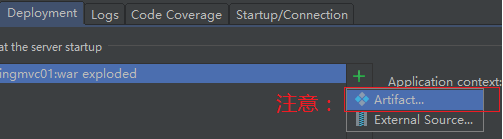
###### 2. 设置文件夹类型

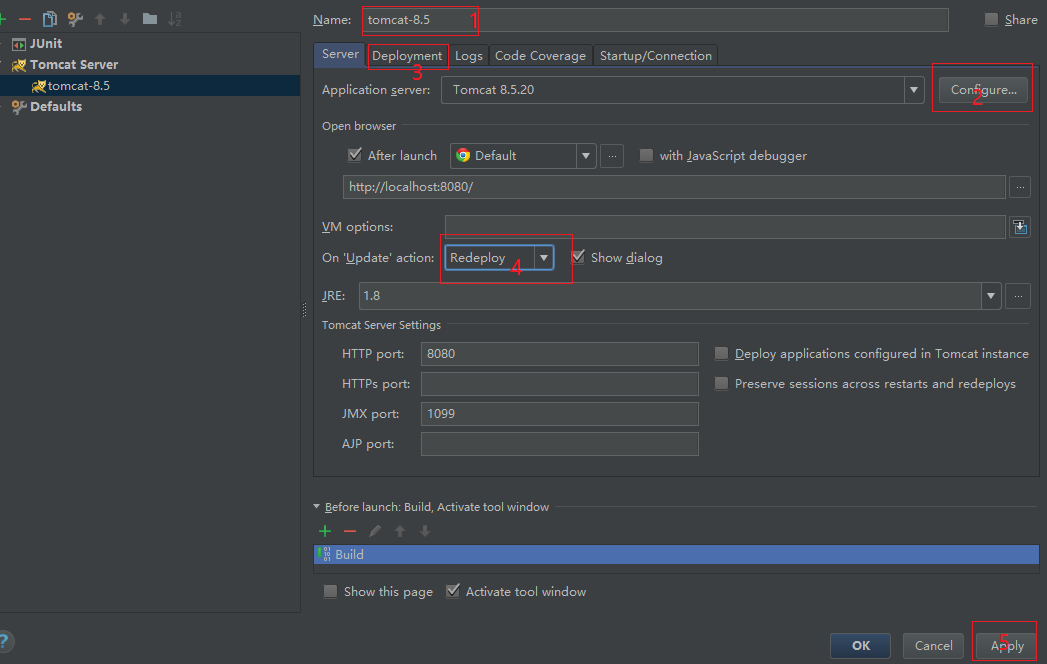


###### 3.配置tomcat









##### 1.2.2 基于非注解的入门程序

###### 1. 配置前端控制器

***Web.xml***

<?xml version="1.0" encoding="UTF-8"?>  
<web-app xmlns="http://java.sun.com/xml/ns/javaee"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://java.sun.com/xml/ns/javaee  
 http://java.sun.com/xml/ns/javaee/web-app\_3\_0.xsd"  
 version="3.0">  
 <!-- 配置web.xml，使其具有springmvc特性，主要配置两处，一个是ContextLoaderListener，一个是DispatcherServlet -->

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>classpath\*:applicationContext.xml</param-value>

</context-param>

<!-- 配置ContextLoaderListener表示，该工程要以spring的方式启动。启动时会默认在/WEB-INF目录下查找applicationContext.xml  
作为spring容器的配置文件，该文件里可以初始化一些bean -->

<listener>

<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

</listener>

<!-- 字符集过滤器 -->

<filter>

<filter-name>encodingFilter</filter-name>

<filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>

<init-param>

<param-name>encoding</param-name>

<param-value>UTF-8</param-value>

</init-param>

</filter>

<filter-mapping>

<filter-name>encodingFilter</filter-name>

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

</filter-mapping>

<!-- spring mvc 配置 -->

<!-- 配置DispatcherServlet表示，该工程将采用springmvc的方式。启动时也会默认在/WEB-INF目录下查找XXX-servlet.xml作为配置文件，

XXX就是DispatcherServlet的名字，该文件中将配置两项重要的mvc特性：HandlerMapping,负责为DispatcherServlet这个前端控制器的请求查找Controller；

ViewResolver,负责为DispatcherServlet查找ModelAndView的视图解析器。

此处使用指定的配置文件spring-mvc.xml -->

<servlet>

<servlet-name>contacts</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

<init-param>

<param-name>contextConfigLocation</param-name>

<!--<param-value>/WEB-INF/classes/spring-mvc-servlet.xml</param-value>-->

<param-value>classpath\*:/spring-mvc.xml</param-value>

</init-param>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>contacts</servlet-name>

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

</servlet-mapping>

<welcome-file-list>

<welcome-file>index.jsp</welcome-file>

</welcome-file-list>

</web-app>

###### 2. 配置处理器映射器，springmvc-servlet.xml

<!--配置处理器映射器  
 使用BeanNameUrlHandlerMapping处理器映射器  
 即，将bean的Name作为url进行查找，即在配置handler时需要配置bean的name属性  
-->  
<bean class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"/>

<!--简单url映射器-->  
<bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">  
 <property name="mappings">  
 <props>  
 <!--userHandler为handler的id-->  
 <prop key="/query.action">userHandler</prop>  
 </props>  
 </property>  
</bean>

###### 3. 配置处理器适配器

<!--处理器适配器  
 指定使用SimpleControllerHandlerAdapter适配器

要求编写的handler类实现Controllor接口  
-->  
<bean class="org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter"/>

<!--另一个适配器

要求编写的handler类实现HttpRequestHandler接口

-->  
<bean class="org.springframework.web.servlet.mvc.HttpRequestHandlerAdapter"/>

###### 4. 配置视图解析器

<!--视图解析器  
 配置解析jsp的视图解析器，默认使用jstl  
-->  
<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver"/>

###### 5. 编写Handler

**//适配器为：.SimpleControllerHandlerAdapter**

public class UserHandler implements Controller {  
 public ModelAndView handleRequest(HttpServletRequest httpServletRequest,  
 HttpServletResponse httpServletResponse) throws Exception {  
 //调用Service查找数据库，并返回数据  
  
 //假设service返回的数据如下：  
 User user = new User();  
 user.setId(13);  
 user.setName("mike");  
 user.setAge(34);  
 user.setAddr("shanghai");  
 user.setPhone("13890294399");  
  
 //创建ModelAndView  
 ModelAndView modelAndView = new ModelAndView();  
 //将数据存放到ModelAndView中  
 modelAndView.addObject("user",user);  
 //确定view页面  
 modelAndView.setViewName("/WEB-INF/views/user.jsp");  
   
 return modelAndView;  
 }  
}

***如果处理器映射器用的是BeanNameUrlHandlerMapping处理器映射器，那么需要配置handler的bean的name属性。如下***：

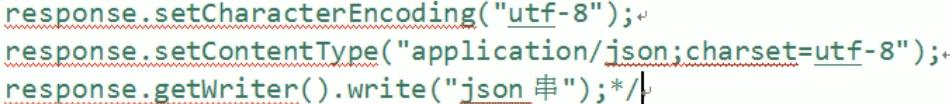
<bean name="/query.action" class="cn.edu.scnu.handler.UserHandler"></bean>

**适配器为：HttpRequestHandlerAdapter**

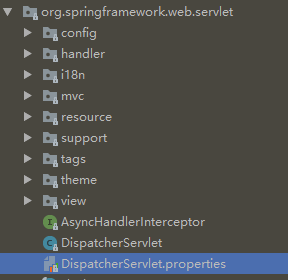
public class UserHandler1 implements HttpRequestHandler {  
 public void handleRequest(HttpServletRequest req,   
 HttpServletResponse res) throws ServletException, IOException {  
 User user = new User();  
 req.setAttribute("user",user);

//下面的路径可以在视图解析器中进行配置，以简化  
 req.getRequestDispatcher("/WEB-INF/views/user.jsp");  
 }  
}

//***使用这种方式可以修改response设置，可以返回json数据***



注意：即使不配置适配器和映射器，也不会出错，因为，不配置的情况下，会默认加载一个配置文件，如下图，这个文件里已经定义了各种适配器和映射器。



##### 1.2.3 基于注解的入门程序

###### 1. 注解的处理器映射器和适配器

<context:component-scan base-package="cn.edu.scnu.handler"/>

<!--注解的映射器-->  
<bean class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping" />  
<!--注解的适配器-->  
<bean class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter"/>

<!—可以代替上面两个注解的配置-->

<mvc:annotation-driven/>

###### 2. handler的编写

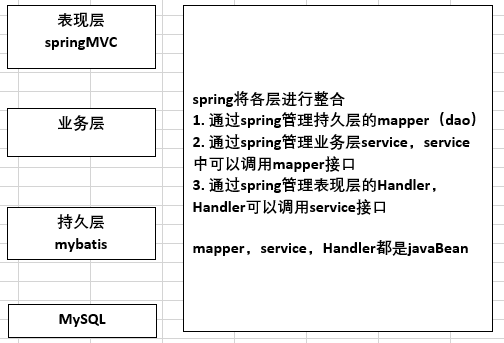
@Controller  
public class UserHandler02 {  
/\*  
使用注解方式，可以实现多个方法。value值一般和方法名相同，便于维护  
 \*/  
 @RequestMapping(value = "queryUser.action")  
 public ModelAndView queryUser(){  
 User user = new User();  
 ModelAndView modelAndView = new ModelAndView();  
 modelAndView.addObject("user",user);  
 return modelAndView;  
 }  
 @RequestMapping(value = "updateUser.action")  
 public ModelAndView updateUser(){  
 User user = new User();  
 ModelAndView modelAndView = new ModelAndView();  
 modelAndView.addObject("user",user);  
 return modelAndView;  
 }  
}

###### 3. 视图解析器的前缀和后缀，主要简化"/WEB-INF/views/user.jsp"

<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">  
 <property name="prefix" value="/WEB-INF/views/"/>  
 <property name="suffix" value=".jsp"/>  
</bean>

#### 1.3 sping + spring mvc + mybatis的整合

##### 1.3.1 springMVC和mybatis整合思路



##### 1.3.2 整合spring和mybatis

见mybatis第七章

##### 1.3.3 整合spring和springmvc

其中spring和spring是无缝结合的，不需要整合。

### 第二章 springMVC的各个组件

#### 2.1 handlerMapping

##### 2.1.1 BeanNameUrlHandlerMapping

官方推荐使用的默认映射器

#### 2.2 控制器

#### 2.3 基于注解的控制器

#### 2.4 ModelAndViewResolver

2.4.1 ModelAndView

ModelAndView代表了MVC Web程序中Model和View的对象，ModelAndVIew的作用封装这两个物件，以方便一次性的返回Model和View两个物件

###### 1. ModelAndView(String viewName)

带着view的名称返回，之后被视图解析器解析

###### 2. ModelAndView(String viewName,Map model)

返回view和model，并且model可以用map作为一个集合。一次性返回多组数据。并且之后可以在视图中被取出。

###### 3.转发和重定向

Forward：return ModelAndView(“forward:/somePage”);

Redirect：return ModelAndView(“redirect:/somePage);

### 第三章 springMVC 实现功能

#### 3.1 异常处理

#### 3.2 拦截器

#### 3.3 数据传递

#### 3.4 国际化

#### 3.5 类型转换

#### 3.6 验证

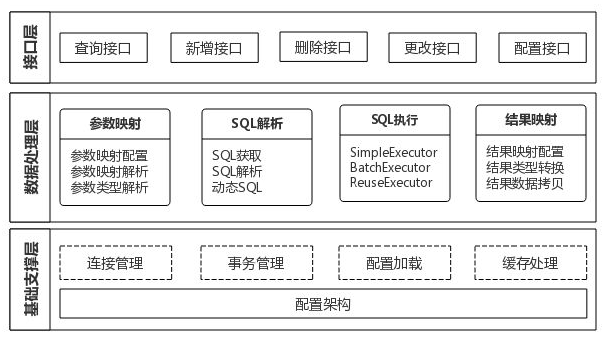
参数绑定，数据回显，上传图片，json数据交互，RESTful支持，拦截器

## 第五节 MyBatis 精通

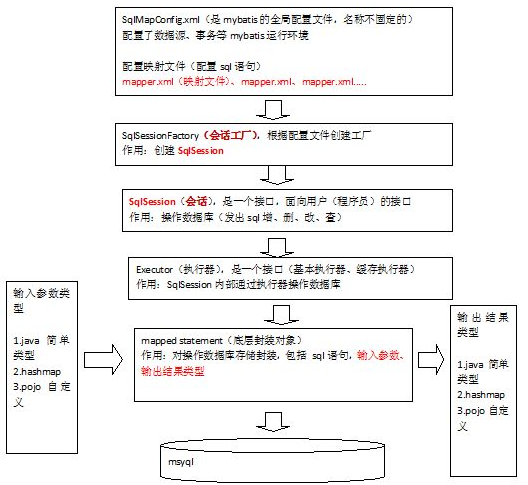
### MyBatis原理

<http://www.mybatis.org/mybatis-3/getting-started.html>

#### 1.1 MyBatis 架构图



#### 1.2 MyBatis运行流程



#### 1.3 MyBatis的增删改查

<http://www.cnblogs.com/woshimrf/p/5651966.html> 示例

##### 1.3.1 需求

* + - 按id查询用户
    - 按name查询用户
    - 增加操作
    - 更新操作
    - 删除操作

##### 1.3.2 环境

* + - Java环境，jdk8 ， IDEA，Maven
    - Mybatis环境，mybatis.jar以及其依赖包
    - Mysql环境，mysql-connect.jar以及其依赖包

##### 1.3.3 Log4j.properties

#开发环境设为DEBUG，生产环境INFO或这ERROR  
# Global logging configuration  
log4j.rootLogger=DEBUG, stdout  
# Console output...  
log4j.appender.stdout=org.apache.log4j.ConsoleAppender  
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout  
log4j.appender.stdout.layout.ConversionPattern=%5p [%t] - %m%n

##### 1.3.4 maven配置

Maven将xml文件编译到classes文件夹下

<build>  
 <resources>  
 <resource>  
 <directory>src/main/java</directory>  
 <includes>  
 <include>\*\*/\*.xml</include>  
 </includes>  
 </resource>  
 </resources>  
</build>

Maven的依赖包

<dependencies>

<!-- https://mvnrepository.com/artifact/junit/junit -->

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.12</version>

<scope>test</scope>

</dependency>

<!-- mysql5与mysql6的区别，com.mysql.jdbc.Driver

Com.mysql.cj.jdbc.Driver

-->

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>5.1.44</version>

</dependency>

<dependency>

<groupId>org.mybatis</groupId>

<artifactId>mybatis</artifactId>

<version>3.4.5</version>

</dependency>

<!-- https://mvnrepository.com/artifact/asm/asm -->

<dependency>

<groupId>asm</groupId>

<artifactId>asm</artifactId>

<version>3.3.1</version>

</dependency>

<!-- https://mvnrepository.com/artifact/cglib/cglib -->

<dependency>

<groupId>cglib</groupId>

<artifactId>cglib</artifactId>

<version>3.2.5</version>

</dependency>

<dependency>

<groupId>commons-logging</groupId>

<artifactId>commons-logging</artifactId>

<version>1.2</version>

</dependency>

<dependency>

<groupId>org.javassist</groupId>

<artifactId>javassist</artifactId>

<version>3.21.0-GA</version>

</dependency>

<dependency>

<groupId>org.apache.logging.log4j</groupId>

<artifactId>log4j-core</artifactId>

<version>2.8.2</version>

</dependency>

<dependency>

<groupId>log4j</groupId>

<artifactId>log4j</artifactId>

<version>1.2.17</version>

</dependency>

<dependency>

<groupId>org.apache.logging.log4j</groupId>

<artifactId>log4j-api</artifactId>

<version>2.8.2</version>

</dependency>

<!-- https://mvnrepository.com/artifact/org.slf4j/slf4j-api -->

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-api</artifactId>

<version>1.7.25</version>

</dependency>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-log4j12</artifactId>

<version>1.7.25</version>

<scope>test</scope>

</dependency>

</dependencies>

##### 1.3.5 全局配置文件sqlMaperConfig.xml

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

<!DOCTYPE configuration

PUBLIC "-//mybatis.org//DTD Config 3.0//EN"

"http://mybatis.org/dtd/mybatis-3-config.dtd">

<configuration>

<!-- 加载java的配置文件或者声明属性信息 -->

<properties resource=*"db.properties"*>

<property name=*"db.username"* value=*"123"* />

</properties>

<!-- <settings></settings> -->

<!-- 自定义别名 -->

<typeAliases>

<!-- 单个别名定义 -->

<!-- <typeAlias type="com.itheima.mybatis.po.User" alias="user"/> -->

<!-- 批量别名定义（推荐） -->

<!-- package：指定包名称来为该包下的po类声明别名，默认的别名就是类名（首字母大小写都可） -->

<package name=*"com.itheima.mybatis.po"* />

</typeAliases>

<!-- 配置mybatis的环境信息，与spring整合，该信息由spring来管理 -->

<environments default=*"development"*>

<environment id=*"development"*>

<!-- 配置JDBC事务控制，由mybatis进行管理 -->

<transactionManager type=*"JDBC"*></transactionManager>

<!-- 配置数据源，采用mybatis连接池 -->

<dataSource type=*"POOLED"*>

<property name=*"driver"* value=*"${db.driver}"* />

<property name=*"url"* value=*"${db.url}"* />

<property name=*"username"* value=*"${db.username}"* />

<property name=*"password"* value=*"${db.password}"* />

</dataSource>

</environment>

</environments>

<!-- 加载映射文件 -->

<mappers>

<mapper resource=*"User.xml"* />

<!-- <mapper resource="mapper/UserMapper.xml" /> -->

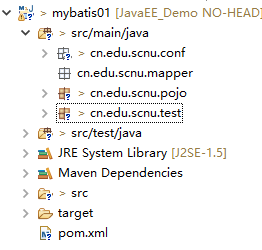
<!-- 批量加载映射文件 -->

<package name="com.itheima.mybatis.mapper" />

</mappers>

</configuration>

##### 1.3.6 工程目录结构



##### 1.3.7 映射文件 UserMapper.xml

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

<!DOCTYPE mapper

PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"

"http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<mapper namespace=*"test"*>

<!--

其中id是statement的id。

paramterTyper是参数#{id}的类型

resultType:是返回值的类型

-->

<select id="findById" parameterType="int" resultType="cn.edu.scnu.pojo.City">  
 SELECT \* FROM areas WHERE id = #{id}  
</select>

</mapper>

##### 1.3.8 测试文件 \_ 按ID查询用户信息

@Test  
public void findById() throws IOException {  
 //创建会话工厂  
  
 //cn/edu/scnu/conf/mybatis-config.xml  
 String resource = "cn/edu/scnu/conf/mybatis-config.xml";  
 InputStream inputStream = Resources.*getResourceAsStream*(resource);  
  
 SqlSessionFactory sessionFactory = new SqlSessionFactoryBuilder().build(inputStream);  
  
 //创建会话  
 SqlSession session = sessionFactory.openSession();  
  
 City city = session.selectOne("test.findById",2);  
  
 System.*out*.println(city.toString());  
  
 session.close();  
}

##### 1.3.9 按name查询用户信息

Mapper.xml

<!--  
 注意：  
 resultType:返回无论单条还是多条，都指的是数据类型  
 ${} : 表示拼接sql，如果是基本数据类型，{}必须是{value}，否则会出错。  
 由于是拼接sql，因此会出现sql注入，比如value= 'or 1=1 or'，拼接之后：  
 '%' or 1=1 or '%'  
-->  
<select id="findByName" parameterType="java.lang.String" resultType="cn.edu.scnu.pojo.City">  
  
 SELECT \* FROM areas WHERE area LIKE '%${value}%'  
  
</select>

findByName.java

@Test  
public void findByName() throws IOException {  
 //创建会话工厂  
  
 //cn/edu/scnu/conf/mybatis-config.xml  
 String resource = "cn/edu/scnu/conf/mybatis-config.xml";  
 InputStream inputStream = Resources.*getResourceAsStream*(resource);  
  
 SqlSessionFactory sessionFactory = new SqlSessionFactoryBuilder().build(inputStream);  
  
 //创建会话  
 SqlSession session = sessionFactory.openSession();  
  
 List<City> cities = session.selectList("test.findByName","区");  
  
 for(City city:cities){  
 System.*out*.println(city.toString());  
 }  
  
 session.close();  
}

##### 1.3.10 插入用户信息

Mapper.xml

<!--  
 parameterType : 插入的数据对象  
 #{}，可以获取对象中的指定属性，比如id属性。  
-->  
<insert id="insertUser" parameterType="cn.edu.scnu.pojo.User">  
 INSERT INTO users(id,name,age,addr) VALUE(#{id},#{name},#{age},#{addr})  
</insert>

insertObject,java  
@Test  
public void insertObject() throws IOException {  
 User user = new User();  
 user.setId(6);  
 user.setName("ff");  
 user.setAge(24);  
 user.setAddr("shandong");  
  
 System.*out*.println(user.toString());  
 System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  
 //创建会话工厂  
  
 //cn/edu/scnu/conf/mybatis-config.xml  
 String resource = "cn/edu/scnu/conf/mybatis-config.xml";  
 InputStream inputStream = Resources.*getResourceAsStream*(resource);  
  
 SqlSessionFactory sessionFactory = new SqlSessionFactoryBuilder().build(inputStream);  
  
 //创建会话  
 SqlSession session = sessionFactory.openSession();  
  
  
 int i = session.insert("test.insertUser",user);  
 session.commit();  
 System.*out*.println(i);  
 session.close();  
  
}

##### 1.3.11 自增主键在插入后的获取

Mapper.xml

<!--  
 其中：  
 SELECT LAST\_INSERT\_ID()：得到刚插入的主键，仅仅适用于自增主键  
 keyProperty:表示的是将返回值，赋值给parameterType指定对象的某个属性  
 order：表示执行顺序在insert语句之前还是之后  
 resultType:指定结果类型  
-->  
<insert id="insertAndGetId" parameterType="cn.edu.scnu.pojo.User">  
  
 <selectKey keyProperty="id" order="AFTER" resultType="java.lang.Integer">  
 SELECT LAST\_INSERT\_ID()  
 </selectKey>  
 INSERT INTO users(name,age,addr) VALUE(#{name},#{age},#{addr})  
</insert>

Insert\_getId.java

@Test  
 public void insert\_getId() throws IOException {   
 User user = new User();  
 user.setName("hh");  
 user.setAge(24);  
 user.setAddr("shandong");  
  
 System.*out*.println(user.toString());  
 System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  
 //创建会话工厂  
  
 //cn/edu/scnu/conf/mybatis-config.xml  
 String resource = "cn/edu/scnu/conf/mybatis-config.xml";  
 InputStream inputStream = Resources.*getResourceAsStream*(resource);  
  
 SqlSessionFactory sessionFactory = new SqlSessionFactoryBuilder().build(inputStream);  
  
 //创建会话  
 SqlSession session = sessionFactory.openSession();  
  
  
 session.insert("test.insertAndGetId",user);  
 session.commit();  
 session.close();  
  
 System.*out*.println(user.getId());  
 }

##### 1.3.12 非自增主键，获取uuid值

##### 1.3.13 更新

Mapper.xml

<update id="updateUser" parameterType="cn.edu.scnu.pojo.User">  
 update users set name = #{name},age = #{age},addr = #{addr} WHERE id = #{id}  
</update>

UpdateUser.java

@Test  
public void updateUser() throws IOException {  
 User user = new User();  
 user.setId(1);  
 user.setName("hh");  
 user.setAge(24);  
 user.setAddr("shandong");  
  
 System.*out*.println(user.toString());  
 System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  
  
 String resource = "cn/edu/scnu/conf/mybatis-config.xml";  
 InputStream inputStream = Resources.*getResourceAsStream*(resource);  
  
 SqlSessionFactory sessionFactory = new SqlSessionFactoryBuilder().build(inputStream);  
  
 //创建会话  
 SqlSession session = sessionFactory.openSession();  
  
 session.update("test.updateUser",user);  
 session.commit();  
 session.close();  
  
}

##### 1.3.14 删除

Mapper.xml

<delete id="deleteUser" parameterType="java.lang.Integer">  
 delete FROM users WHERE id = #{id}  
</delete>

DeleteUser.java

@Test  
public void delteUser() throws IOException {  
 String resource = "cn/edu/scnu/conf/mybatis-config.xml";  
 InputStream inputStream = Resources.*getResourceAsStream*(resource);  
  
 SqlSessionFactory sessionFactory = new SqlSessionFactoryBuilder().build(inputStream);  
  
 //创建会话  
 SqlSession session = sessionFactory.openSession();  
  
 session.delete("test.deleteUser",1);  
 session.commit();  
  
 session.close();  
}

### MyBatis开发Dao方法

#### 2.1 原始dao开发方法

##### 2.1.1 定义dao接口

public interface UserDao {  
 /\*  
 public static final 常量  
 public abstract 方法  
  
 可以省略  
 \*/  
  
 City findById(Integer id);  
 List findByName(String name);  
 void deleteUser(Integer id);  
  
}

##### 2.1.2 实现dao接口，并将sqlSessionFactory 注入

public class UserDaoImpl implements UserDao {  
 private SqlSessionFactory sqlSessionFactory;  
  
 public UserDaoImpl(SqlSessionFactory sqlSessionFactory) {  
 this.sqlSessionFactory = sqlSessionFactory;  
  
 }  
  
 public City findById(Integer id) {  
 SqlSession sqlSession = sqlSessionFactory.openSession();  
 return sqlSession.selectOne("test.findById",1);  
  
 }  
  
 public List findByName(String name) {  
 SqlSession sqlSession = sqlSessionFactory.openSession();  
 return sqlSession.selectList("test.findByName","区");  
 }  
  
 public void deleteUser(Integer id) {  
 SqlSession sqlSession = sqlSessionFactory.openSession();  
 sqlSession.delete("test.deleteUser",3);  
 }  
}

##### 2.1.3 测试文件

public class DaoTest {  
 private SqlSessionFactory sqlSessionFactory;  
  
 @Before  
 public void beforeInfo() throws IOException {  
 String resource = "cn/edu/scnu/conf/mybatis-config.xml";  
 InputStream inputStream = Resources.*getResourceAsStream*(resource);  
 sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);  
 }  
  
  
 @Test  
 public void findById(){  
 UserDaoImpl userDao = new UserDaoImpl(sqlSessionFactory);  
 City city = userDao.findById(1);  
 System.*out*.println(city.toString());  
 }  
  
}

##### 2.1.4 总结

1. dao接口方法中存在大量模板方法（重复代码）；

2. statement的id硬编码 return sqlSession.selectOne("test.findById",1);  
 3. Sqlsession使用泛型，传入类型错误不会报错。

#### 2.2 Mapper代理方法

##### 2.2.1 编写Mapper.xml文件

<mapper namespace="cn.edu.scnu.mapper.CityMapper">  
 <select id="findById" parameterType="int" resultType="cn.edu.scnu.pojo.City">  
 SELECT \* FROM areas WHERE id = #{id}  
 </select>

</mapper>

##### 2.2.2 编写Mapper接口（相当于dao接口）

程序员在白那些mapper接口需要遵循一些开发规范，mybatis会自动生成mapper接口的实现类对象。

开发规范:

1. namespace = mapper接口

<mapper namespace="cn.edu.scnu.mapperInter.UserMapper">

2. mapper.java中的方法名 = mapper.xml中的statement的id

3. mapper.java接口中的方法输入参数类型和mapper.xml中的statement的

parameterType指定的类型一致。

4. mapper.java接口中的方法返回值类型和mapper.xml中的statement的

resultType指定的类型一致。

Mapper.java

public interface CityMapper {  
  
 City findById(int id) throws Exception;  
}

##### 2.2.3 测试

public class MapperTest {  
 private SqlSessionFactory sqlSessionFactory;  
 @Before  
 public void beforeInfo() throws IOException {  
  
 String resource = "cn/edu/scnu/conf/mybatis-config.xml";  
  
 InputStream inputStream = Resources.*getResourceAsStream*(resource);  
  
 sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);  
 }  
  
 @Test  
 public void findById() throws Exception {  
 SqlSession session = sqlSessionFactory.openSession();  
  
 CityMapper cityMapper = session.getMapper(CityMapper.class);  
  
 City city = cityMapper.findById(3);  
 System.*out*.println(city.toString());  
 }  
}

##### 2.2.4 总结

1. 如果返回单独一个对象，mybatis会默认调用selectOne，如果返回一个List类型，mybatis默认调用selectList

2. mapper接口方法参数只有一个，但是也可以使用包装类型的pojo以便满足不同的需求。

3. 注意：持久层dao可以使用包装类型宽展，但是业务层不建议使用包装类型。

### 第三章 全局配置文件—mybatis-config.xml

#### 3.1 properties定义

##### 3.1.1 db.properties

jdbc.driver = com.mysql.jdbc.Driver  
jdbc.url = jdbc:mysql://localhost:3306/city\_test  
jdbc.username=root  
jdbc.password=12345

##### 3.1.2 mybatis-config.xml

<configuration>  
<!--加载属性文件-->  
<properties resource="cn/edu/scnu/conf/db.properties"></properties>  
  
<!--加载属性，建议不要使用-->  
<properties>  
 <property name="" value=""></property>  
</properties>

<!-- 配置mybatis的环境信息，与spring整合，该信息由spring来管理 -->  
 <environments default="development">  
 <environment id="development">  
 <!-- 配置JDBC事务控制，由mybatis进行管理 -->  
 <transactionManager type="JDBC"></transactionManager>  
 <!-- 配置数据源，采用mybatis连接池 -->  
 <dataSource type="POOLED">  
 <property name="driver" value="${jdbc.driver}" />  
 <property name="url" value="${jdbc.url}" />  
 <property name="username" value="${jdbc.username}" />  
 <property name="password" value="${jdbc.password}" />  
 </dataSource>  
 </environment>  
 </environments>  
 <mappers>  
 <mapper resource="cn/edu/scnu/sqlMap/CityMapper.xml"></mapper>  
 <mapper resource="cn/edu/scnu/sqlMap/UserMapper.xml"></mapper>  
 </mappers>  
</configuration>

属性读取顺序：

先读取<properties>,再读取resource,最后读取parameterType传递的属性，会覆盖前者。

因此：

parameterType传递的属性最高级

resouce属性次之

<properties>属性最低

#### 3.2 settings全局参数配置

全局参数配置主要配置缓存，懒加载等内容。

<settings>  
 <setting name="" value=""/>  
</settings>

#### 3.3 别名定义

##### 3.3.1 默认别名



##### 3.3.2 自定义别名

在mapper.xml中，定义很多的statement，statement需要parameterType指定输入参数的类型、需要resultType指定输出结果的映射类型。

如果在指定类型时输入类型的全路径，不方便开发，可以针对parameterType和resultType指定的类型定义一些别名。

resultType="cn.edu.scnu.pojo.City"

mybatis-config.xml

<typeAliases>  
 <!--单个类型的别名定义-->  
 <typeAlias type="cn.edu.scnu.pojo.City" alias="City"></typeAlias>  
 <!--批量定义别名，别名就是类型，首字母大小写都行-->  
 <package name="cn.edu.scnu.pojo"></package>  
</typeAliases>

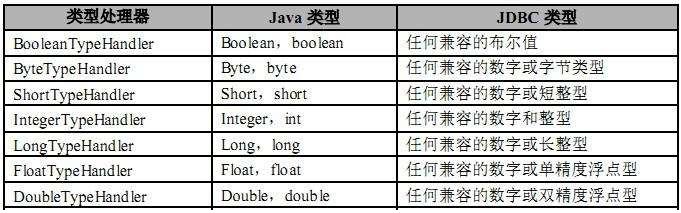
#### 3.4 类型处理器

Mybatis中通过typeHandlers完成jdbc类型和java类型的转换。

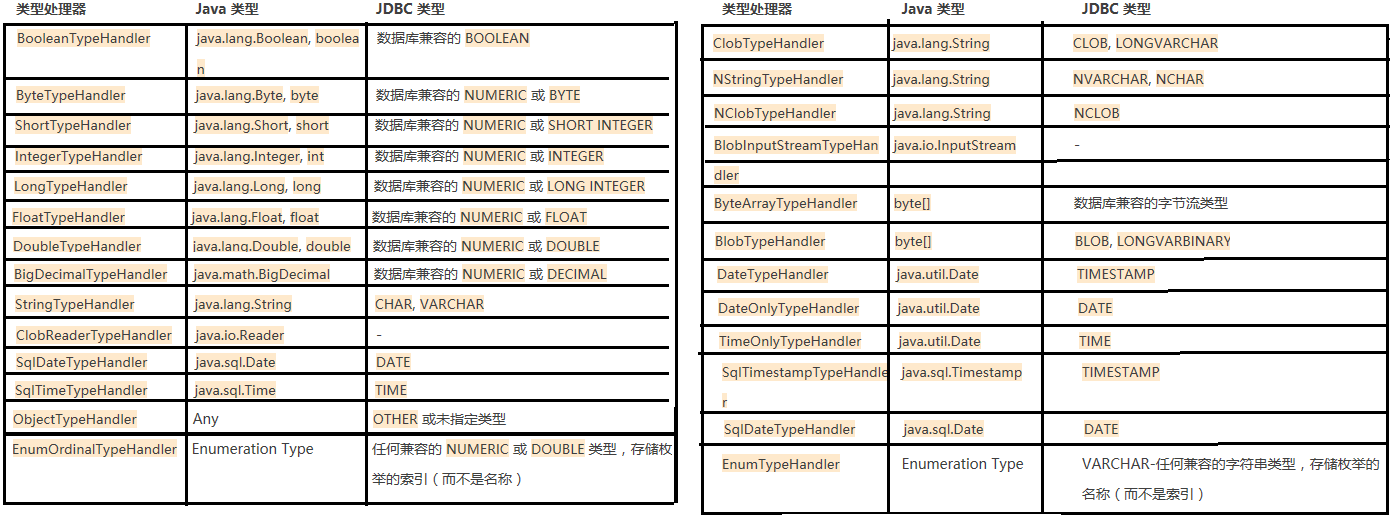
通常情况下，mybatis提供的类型处理器满足日常需要，不需要自定义。

<typeHandlers>  
 <!--单个类型处理器-->  
 <typeHandler handler=""></typeHandler>  
 <!--多个类型处理器-->  
 <package name=""></package>  
</typeHandlers>

Mybatis支持的类型处理器（以下只是其中一部分）：



所有的类型处理器



#### 3.5 mapper的加载

<mappers>  
 <!--使用resource加载mapper.xml文件-->  
 <mapper resource="cn/edu/scnu/sqlMap/CityMapper.xml"></mapper>  
 <!--通过mapper接口加载，规范如下：  
 1. 必须使用mapper代理方法开发  
 2. 接口名必须和xml文件名保持一致。  
 -->  
 <mapper class="cn.edu.scnu.mapper.CityMapper"></mapper>  
 <!--批量加载-->  
 <package name="cn.edu.scnu.mapper"></package>  
</mappers>

### 第四章 映射文件配置—Mapper.xml

#### 4.1 输入映射—parameterType

##### 4.1.1 简单类型

<select id="findById" parameterType="int" resultType="city">  
 SELECT \* FROM areas WHERE id = #{id}  
</select>

##### 4.1.2 插入时的包装类型

<insert id="insertUser" parameterType="cn.edu.scnu.pojo.User">  
 INSERT INTO users(id,name,age,addr) VALUE(#{id},#{name},#{age},#{addr})  
</insert>

##### 4.1.3 复杂查询时的包装类型

***UserMapper.xml***

<mapper namespace="cn.edu.scnu.mapper.UserMapper">  
 <select id="findByAge" parameterType="cn.edu.scnu.vo.UserQueryVo" resultType="cn.edu.scnu.pojo.User">  
 SELECT \* FROM users WHERE age = #{age} AND addr LIKE '%${addr}%'  
 </select>  
</mapper>

***UserMapper.java***

public interface UserMapper {  
  
 User findByAge(UserQueryVo userQueryVo) throws Exception;  
  
}

***MapperTest.java***

public class MapperTest {  
 private SqlSessionFactory sqlSessionFactory;  
 @Before  
 public void beforeInfo() throws IOException {  
  
 String resource = "cn/edu/scnu/conf/mybatis-config.xml";  
  
 InputStream inputStream = Resources.*getResourceAsStream*(resource);  
  
 sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);  
 }  
  
 @Test  
 public void findByAge() throws Exception {  
 SqlSession session = sqlSessionFactory.openSession();  
 UserMapper userMapper = session.getMapper(UserMapper.class);  
 UserQueryVo userQueryVo = new UserQueryVo();  
 userQueryVo.setAge(34);  
 userQueryVo.setAddr("shan");  
 User user = userMapper.findByAge(userQueryVo);  
 System.*out*.println(user.toString());  
 }  
}

#### 4.2 输出映射—resultType/resultMap

##### 4.2.1 resultType

输出pojo类型以及pojo类型的列表

使用resultType进行输出映射，只有查询出来的列名和pojo中的属性名一致，该列才能映射成功。

如果查询出来的列名和pojo中的属性名全部不一致，则没有创建pojo对象。

只要查询出来的列名有一个一致，就会创建pojo对象。只是列名不一致的属性值为null。

Eg: resultType=”cn.edu.scnu.pojo.User”

输出简单类型

查询出来的结果集只有一行一列，可以使用简单类型进行输出映射

例如mysql中的聚合操作，select count(\*) from users where sex=’1’

Eg: resultType = “int”

##### 4.2.2 resultMap

如果查询出来的列名和pojo的属性不一致，通过定义一个resultMap对列名和pojo属性名之间作一个映射关系

1. 定义resultMap

2. 使用resultMap作为statement的输出映射类型

Eg:

SELECT id id\_, NAME username, addr address FROM users WHERE id =3

定义resultMap

<!--  
 id:resultMap的唯一标识  
 type：要映射的pojo  
-->  
<resultMap id="UserResultMap" type="cn.edu.scnu.pojo.User">  
 <!--  
 id 表示结果集中的唯一标志，主键  
 column：表示查询出来的列名  
 property：表示要映射的pojo的属性名  
  
 result：表示结果集中的其他列  
 -->  
 <id column="id\_" property="id"></id>  
 <result column="username" property="name"></result>  
 <result column="address" property="addr"></result>  
</resultMap>

定义select语句

<!--resultMap:指定定义的resultMap的id，如果这个resultMap在其他的mapper文件，前边需要加namespace-->

<select id="findResultMap" parameterType="int" resultMap="UserResultMap">  
 SELECT id id\_, name username, addr address FROM users WHERE id =#{id}  
</select>

#### 4.3 动态SQL

##### 4.3.1 if判断

<select id="findByAge" parameterType="cn.edu.scnu.vo.UserQueryVo" resultType="cn.edu.scnu.pojo.User">  
 SELECT \* FROM users  
 <!--  
 where标签会自动忽略第一个and 然后满足条件后拼接sql，如下  
 select \* from users where age = #{age} AND addr LIKE '%${addr}%'  
 -->  
 <where>  
 <if test="age>0">  
 and age = #{age}  
 </if>  
 <if test=" addr!=null and addr != ''">  
 AND addr LIKE '%${addr}%'  
 </if>  
 </where>  
</select>

##### 4.3.2 SQL片段

<!--  
 SQL片段：  
 1. 针对单表操作更容易重用  
 2. 不带where 更容易重用  
-->  
<sql id="QuerySQL">  
 <if test="age>0">  
 and age = #{age}  
 </if>  
 <if test=" addr!=null and addr != ''">  
 AND addr LIKE '%${addr}%'  
 </if>  
</sql>  
  
<select id="findByAge" parameterType="cn.edu.scnu.vo.UserQueryVo" resultType="cn.edu.scnu.pojo.User">  
 SELECT \* FROM users  
 <!--  
 where标签会自动忽略第一个and 然后满足条件后拼接sql，如下  
 select \* from users where age = #{age} AND addr LIKE '%${addr}%'  
 -->  
 <where>  
 <include refid="QuerySQL"></include>

<!-- …… 可以多个include-->  
 </where>  
</select>

##### 4.3.3 foreach循环

UserForVo.java

public class UserForVo {  
 private List<Integer> ids;  
 public List<Integer> getIds() {  
 return ids;  
 }  
  
 public void setIds(List<Integer> ids) {  
 this.ids = ids;  
 }  
}

UserMapper.xml

<select id="findForEach" parameterType="cn.edu.scnu.vo.UserForVo" resultType="cn.edu.scnu.pojo.User">  
 SELECT \* FROM users  
 <where>  
 <if test="ids!=null">  
 <foreach collection="ids" item="value\_id" open="and (" close=")" separator="or">  
 id=#{value\_id}  
 </foreach>  
 </if>  
 </where>  
</select>

UserMapper.java

public interface UserMapper {  
 List<User> findForEach(UserForVo userForVo);  
}

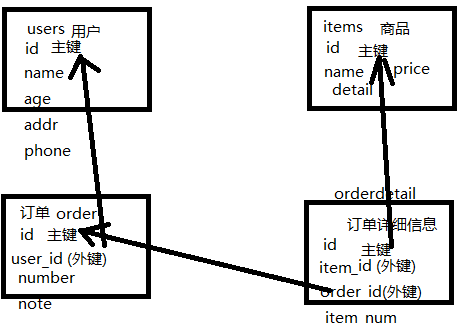
MapperTest.java

@Test  
public void findForEach(){  
 SqlSession session = sqlSessionFactory.openSession();  
 UserMapper userMapper = session.getMapper(UserMapper.class);  
  
 UserForVo userForVo = new UserForVo();  
 List<Integer> ids = new ArrayList<Integer>();  
 ids.add(3);  
 ids.add(4);  
 ids.add(5);  
 userForVo.setIds(ids);  
  
 List<User> users = userMapper.findForEach(userForVo);  
 for(User u:users){  
 System.*out*.println(u.toString());  
 }  
}

### 第五章 高级映射

#### 5.1 需求分析

商城的数据库建模



分析：

数据库级别的关系：

用户和订单：

用户🡪订单，一个用户可以创建多个订单，一对多

订单🡪用户，一个订单只对应一个用户，一对一

订单和订单明细

订单🡪订单明细，一个订单对应多个商品，即对应多个订单明细，一对多

订单明细🡪订单，一个订单明细只对应一个订单，一对一

订单明细和商品

订单明细🡪商品，一个订单明细只对应一个商品信息，一对一

商品🡪订单明细，一个商品可以包含在多个明细中，一对多

业务级别的关系：

订单和商品，多对多的关系

订单🡪商品，一个订单对应多个商品，一对多

商品🡪订单，一个商品可以对应多个订单，一对多

用户和商品，多对多关系

用户🡪商品，一个用户可以购买多个商品，一对多

商品🡪用户，一个商品可以让多个用户购买，一对多。

***Market.sql***

SET NAMES utf8mb4;

SET FOREIGN\_KEY\_CHECKS = 0;

-- ----------------------------

-- Table structure for items

-- ----------------------------

DROP TABLE IF EXISTS `items`;

CREATE TABLE `items` (

`id` int(11) NOT NULL AUTO\_INCREMENT,

`name` varchar(255) CHARACTER SET utf8 COLLATE utf8\_general\_ci NOT NULL,

`price` decimal(10, 2) NOT NULL,

`detail` varchar(255) CHARACTER SET utf8 COLLATE utf8\_general\_ci DEFAULT NULL,

PRIMARY KEY (`id`) USING BTREE

) ENGINE = InnoDB AUTO\_INCREMENT = 5 CHARACTER SET = utf8 COLLATE = utf8\_general\_ci ROW\_FORMAT = Compact;

-- ----------------------------

-- Records of items

-- ----------------------------

INSERT INTO `items` VALUES (1, 'book1', 23.00, 'book1');

INSERT INTO `items` VALUES (2, 'book2', 25.80, 'book2');

INSERT INTO `items` VALUES (3, 'book3', 345.30, 'book3');

INSERT INTO `items` VALUES (4, 'book4', 23.90, 'book4');

-- ----------------------------

-- Table structure for orderdetail

-- ----------------------------

DROP TABLE IF EXISTS `orderdetail`;

CREATE TABLE `orderdetail` (

`id` int(11) NOT NULL AUTO\_INCREMENT,

`item\_id` int(11) NOT NULL,

`order\_id` int(11) NOT NULL,

`item\_num` int(11) NOT NULL,

PRIMARY KEY (`id`) USING BTREE,

INDEX `item\_id`(`item\_id`) USING BTREE,

INDEX `order\_id`(`order\_id`) USING BTREE,

CONSTRAINT `item\_id` FOREIGN KEY (`item\_id`) REFERENCES `items` (`id`) ON DELETE RESTRICT ON UPDATE RESTRICT,

CONSTRAINT `order\_id` FOREIGN KEY (`order\_id`) REFERENCES `orders` (`id`) ON DELETE RESTRICT ON UPDATE RESTRICT

) ENGINE = InnoDB AUTO\_INCREMENT = 5 CHARACTER SET = utf8 COLLATE = utf8\_general\_ci ROW\_FORMAT = Compact;

-- ----------------------------

-- Records of orderdetail

-- ----------------------------

INSERT INTO `orderdetail` VALUES (1, 3, 2, 2);

INSERT INTO `orderdetail` VALUES (2, 2, 3, 1);

INSERT INTO `orderdetail` VALUES (3, 4, 1, 3);

INSERT INTO `orderdetail` VALUES (4, 1, 4, 3);

-- ----------------------------

-- Table structure for orders

-- ----------------------------

DROP TABLE IF EXISTS `orders`;

CREATE TABLE `orders` (

`id` int(11) NOT NULL AUTO\_INCREMENT,

`number` int(11) NOT NULL,

`note` varchar(255) CHARACTER SET utf8 COLLATE utf8\_general\_ci DEFAULT NULL,

`user\_id` int(11) NOT NULL,

PRIMARY KEY (`id`) USING BTREE,

INDEX `user\_id`(`user\_id`) USING BTREE,

CONSTRAINT `user\_id` FOREIGN KEY (`user\_id`) REFERENCES `users` (`id`) ON DELETE RESTRICT ON UPDATE RESTRICT

) ENGINE = InnoDB AUTO\_INCREMENT = 5 CHARACTER SET = utf8 COLLATE = utf8\_general\_ci ROW\_FORMAT = Compact;

-- ----------------------------

-- Records of orders

-- ----------------------------

INSERT INTO `orders` VALUES (1, 2, NULL, 1);

INSERT INTO `orders` VALUES (2, 1, NULL, 2);

INSERT INTO `orders` VALUES (3, 3, NULL, 3);

INSERT INTO `orders` VALUES (4, 5, NULL, 4);

-- ----------------------------

-- Table structure for users

-- ----------------------------

DROP TABLE IF EXISTS `users`;

CREATE TABLE `users` (

`id` int(11) NOT NULL AUTO\_INCREMENT,

`name` varchar(100) CHARACTER SET utf8 COLLATE utf8\_general\_ci NOT NULL,

`age` int(11) DEFAULT NULL,

`addr` varchar(255) CHARACTER SET utf8 COLLATE utf8\_general\_ci NOT NULL,

`phone` varchar(255) CHARACTER SET utf8 COLLATE utf8\_general\_ci NOT NULL,

PRIMARY KEY (`id`) USING BTREE

) ENGINE = InnoDB AUTO\_INCREMENT = 5 CHARACTER SET = utf8 COLLATE = utf8\_general\_ci ROW\_FORMAT = Compact;

-- ----------------------------

-- Records of users

-- ----------------------------

INSERT INTO `users` VALUES (1, 'aa', 23, 'beijing', '12345');

INSERT INTO `users` VALUES (2, 'bb', 24, 'shanghai', '34567');

INSERT INTO `users` VALUES (3, 'cc', 35, 'guangzhou', '565768');

INSERT INTO `users` VALUES (4, 'dd', 32, 'shenzhen', '958674');

SET FOREIGN\_KEY\_CHECKS = 1;

#### 5.2 一对一映射

##### 5.2.1 sql语句

需求分析：

通过orders表关联查询用户。

关联查询：

1. 确定主表，orders

2. 确定关联表，users

3. 确定关联时内连接还是外连接

a) 如果通过orders表的外键(user\_id)只能查到一个user，那么可以使用内链接

eg： SELECT \* FROM orders,users WHERE orders.user\_id = users.id

去除重复的id字段。

SELECT orders.\*, users.name, users.age, users.addr, users.phone FROM orders,users WHERE orders.user\_id = users.id

b)

##### 5.2.2 pojo

***User.java***

private Integer id;  
private String name;  
private Integer age;  
private String addr;  
private String phone;

***Order.java***

private Integer id;  
private Integer number;  
private String note;  
private Integer user\_id;

***OrderDetail.java***

private Integer id;  
private Integer item\_id;  
private Integer order\_id;  
private Integer item\_num;

***Item.java***

private Integer id;  
private String name;  
private double price;  
private String detail;

##### 5.2.3 vo

***OrderCustom.java***

public class OrderCustom extends Order{  
 //SELECT orders.\*, users.name, users.age, users.addr, users.phone FROM orders,users WHERE orders.user\_id = users.id  
   
 private String name;  
 private Integer age;  
 private String addr;  
 private String phone;  
  
 @Override  
 public String toString() {  
 return "OrderCustom{" +  
 "name='" + name + '\'' +  
 ", age=" + age +  
 ", addr='" + addr + '\'' +  
 ", phone='" + phone + '\'' +  
 "} " + super.toString();  
 }

}

##### 5.2.4 sqlMap

***OrderMapper.xml***

<mapper namespace="cn.edu.scnu.mapper.OrderCustomMapper">  
 <select id="findOrderUser" resultType="cn.edu.scnu.vo.OrderCustom">  
 SELECT  
 orders.\*,  
 users.name,  
 users.age,  
 users.addr,  
 users.phone FROM orders,users  
 WHERE orders.user\_id = users.id  
 </select>  
</mapper>

##### 5,2,5 ResultMap方式的实现

***OrderMapper.xml***

<!--resultMap方式返回，在order pojo中添加用户信息-->  
  
<resultMap id="OrderUserMap" type="cn.edu.scnu.pojo.OrderResultMap">  
 <!--这个id表示唯一标识Order-->  
 <id column="id" property="id"></id>  
 <result column="number" property="number"></result>  
 <result column="note" property="note"></result>  
 <result column="user\_id" property="user\_id"></result>  
  
 <!--association要指定属性，并且指定属性的类型-->  
 <association property="user" javaType="cn.edu.scnu.pojo.User">  
 <!-- 这个id表示唯一标识用户-->  
 <id column="user\_id" property="id"></id>  
 <result column="name" property="name"></result>  
 <result column="age" property="age"></result>  
 <result column="addr" property="addr"></result>  
 <result column="phone" property="phone"></result>  
 </association>  
</resultMap>  
<select id="findOrderUserResultMap" resultMap="OrderUserMap">  
 SELECT  
 orders.\*,  
 users.name,  
 users.age,  
 users.addr,  
 users.phone FROM orders,users  
 WHERE orders.user\_id = users.id  
</select>

***OrderResultMap.java***

public class OrderResultMap {  
 private Integer id;  
 private Integer number;  
 private String note;  
 private Integer user\_id;  
 private User user;

}

***OrderMapper.java***

List<OrderResultMap> findOrderUserResultMap();

***Test.java***

@org.junit.Test  
public void findOrderUserResultMap(){  
 SqlSession session = sqlSessionFactory.openSession();  
 OrderCustomMapper orderCustomMapper = session.getMapper(OrderCustomMapper.class);  
  
 List<OrderResultMap> orderResultMaps = orderCustomMapper.findOrderUserResultMap();  
  
 session.close();  
  
 for(OrderResultMap orderResultMap:orderResultMaps){  
 System.*out*.println(orderResultMap.toString());  
 }  
}

##### 5.2.6 总结

实现一对一查询:

1. resultType:实现较为简单，如果pojo中没有包括查询出来的列名，只需要增加对应属性即可。建议使用，但是resultType不能实现延迟加载。

2. resultMap，需要单独定义resultMap，实现比较麻烦。但是可以完成将关联查询映射到pojo的属性中。并且可以实现延迟加载

#### 5.3 一对多映射

##### 5.3.1 需求

查询订单以及订单明细的信息。下面是sql语句：

确定主表：订单表

确定关联表：订单明细表，用户表

SELECT users.name,orders.\*,orderdetail.item\_id,orderdetail.item\_num FROM users,orders,orderdetail WHERE orders.user\_id=users.id and orderdetail.order\_id = orders.id

##### 5.3.2 OrderResultMap.java

public class OrderResultMap {  
 private Integer id;  
 private Integer number;  
 private String note;  
 private Integer user\_id;  
 private User user;  
 private List<OrderDetail> orderDetails;

}

##### 5.3.3 OrderMapper.xml

<resultMap id="OrderAndOrderDetail" type="cn.edu.scnu.pojo.OrderResultMap" extends="OrderUserMap">  
 <!--配置order-->  
 <!--配置User-->  
 <!--上面两个OrderUserMap中已经实现，继承之后不需要再写-->  
  
 <!--配置OrderDetail,因为是一对多的关系，OrderDetail在pojo中是List类型，因此使用collection  
 ofType：指定集合的泛型  
 -->  
 <collection property="orderDetails" ofType="cn.edu.scnu.pojo.OrderDetail">  
 <id column="orderdetail\_id" property="id"></id>  
 <result column="item\_id" property="item\_id"></result>  
 <result column="item\_num" property="item\_num"></result>  
 </collection>  
  
</resultMap>  
  
<select id="findOrderUserAndOrderDetail" resultMap="OrderAndOrderDetail">  
 SELECT  
 orders.\*,  
 users.name,  
 users.age,  
 users.addr,  
 users.phone,  
 orderdetail.id orderdetail\_id,  
 orderdetail.item\_id,  
 orderdetail.item\_num  
 FROM users,orders,orderdetail  
 WHERE orders.user\_id=users.id and orderdetail.order\_id = orders.id  
</select>

##### 5.3.4 OrderMapper.java

List<OrderResultMap> findOrderUserAndOrderDetail();

##### 5.3.5 Test.java

@org.junit.Test  
public void findOrderUserAndOrderDetail(){  
 SqlSession session = sqlSessionFactory.openSession();  
 OrderCustomMapper orderCustomMapper = session.getMapper(OrderCustomMapper.class);  
 List<OrderResultMap> orderResultMaps = orderCustomMapper.findOrderUserAndOrderDetail();  
 session.close();  
  
 for (OrderResultMap orderResultMap:orderResultMaps){  
 System.*out*.println(orderResultMap.toString());  
 }  
}

#### 5.4 多对多映射

多对多映射和一对多映射类似，只是在xml处理时候，collection与association重叠使用

#### 5.5 延迟加载

##### 5.5.1 延迟加载的定义

resultMap可以实现高级映射（使用association、collection实现一对一以及一对多映射），*association、collection具备延迟加载的功能。*

需求：

如果查询订单并且关联查询用户信息。如果先查询订单信息即可满足要求，当我们需要查询用户信息时再查询用户信息。把对用户信息的按需去查询就是延迟加载。

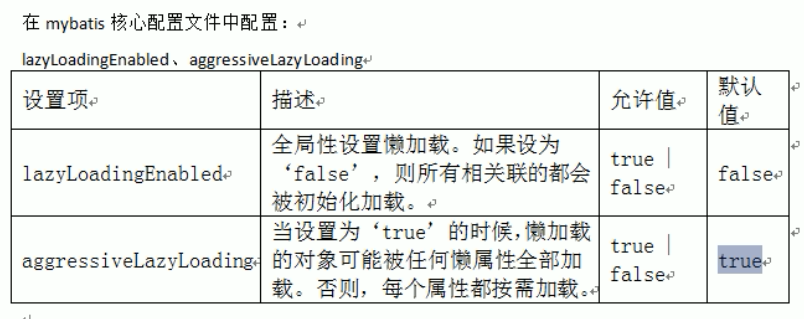
延迟加载：先从单表查询、需要时再从关联表去关联查询，大大提高数据库性能，因为查询单表要比关联查询多表速度要快 。

##### 5.5.2 延迟加载的实现

***1. 打开延迟加载开关 mybatis-conf.xml***

在mybatis核心配置文件中配置

lazyLoadingEnabled ; aggressiveLazyLoading



<settings>  
 <setting name="lazyLoadingEnabled" value="true"></setting>  
 <setting name="aggressiveLazyLoading" value="false"></setting>  
</settings>

***OrderMapper.xml***

<!--延迟加载-->  
<select id="findUserById" parameterType="int" resultType="cn.edu.scnu.pojo.User">  
 select \* from users where id=#{id}  
</select>  
<resultMap id="UserLazyLoading" type="cn.edu.scnu.pojo.OrderResultMap" extends="OrderUserMap">  
 <!--这个id表示唯一标识Order-->  
 <id column="id" property="id"></id>  
 <result column="number" property="number"></result>  
 <result column="note" property="note"></result>  
 <result column="user\_id" property="user\_id"></result>  
 <!--association要指定属性，并且指定属性的类型  
 select:表示查询user表的select  
 column：表示在订单表中的user外键  
 -->  
 <association property="user" javaType="cn.edu.scnu.pojo.User" select="findUserById" column="user\_id">  
 <!-- 这个id表示唯一标识用户-->  
 <id column="user\_id" property="id"></id>  
 <result column="name" property="name"></result>  
 <result column="age" property="age"></result>  
 <result column="addr" property="addr"></result>  
 <result column="phone" property="phone"></result>  
 </association>  
</resultMap>  
<select id="findOrderLazyLoading" resultMap="UserLazyLoading">  
 select \* from orders  
</select>

***Test.java***

首先查询orders的订单信息

当循环调用getUser()时，才会查询User信息

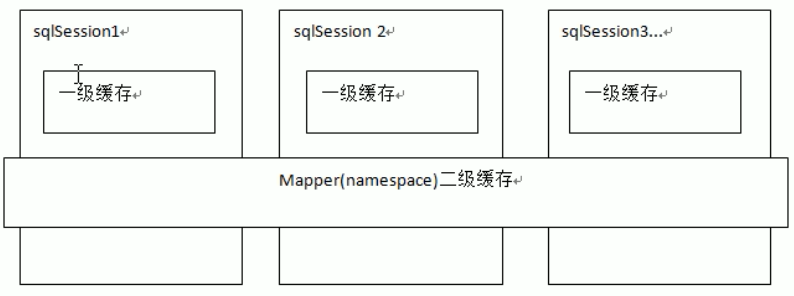
@org.junit.Test  
public void findOrderLazyLoading(){  
 SqlSession session = sqlSessionFactory.openSession();  
 OrderCustomMapper orderCustomMapper = session.getMapper(OrderCustomMapper.class);  
 List<OrderResultMap> orderResultMaps = orderCustomMapper.findOrderLazyLoading();  
 for(OrderResultMap orderResultMap:orderResultMaps){  
 System.*out*.println(orderResultMap.getUser().toString());  
 }  
 session.close();  
  
}

### 第六章 查询缓存

#### 6.1 mybais缓存介绍

Mybatis提供查询缓存，用于减轻数据压力，提高数据库性能。

如下图，是mybatis一级缓存和二级缓存的区别图解：



一级缓存，sqlSession级别缓存。在操作数据库时需要构造sqlSession对象，在对象中有一个数据金额狗（HashMap）用于存储缓存数据。不同的sqlSession之间的缓存数据区域（hashMap）是相互不影响的。

二级缓存，是mapper级别的缓存。多个sqlSession可能操作同一个Mapper中的sql语句，多个sqlSession可以共用二级缓存，二级缓存是跨sqlSession的。

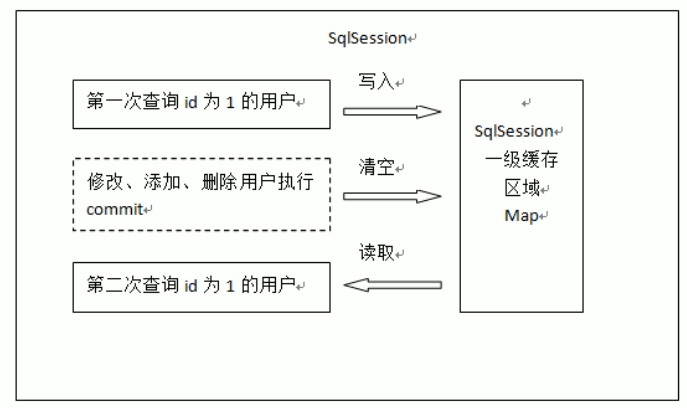
为什么要用缓存？

如果缓存中有数据就不用从数据库获取，大大提高系统性能。

#### 6.2 一级缓存

##### 6.2.1 一级缓存的原理

通过ID查询用户，一级缓存示意图：



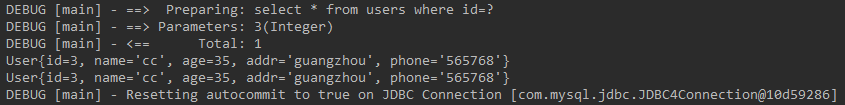
从图中可以看出，第一次查询会存入一级缓存中，第二次查询直接从缓存中查询。而**如果sqlSession去执行commit操作（执行插入，更新，删除），会清空sqlsession中的一级缓存。可以让缓存中总是存储最新信息，避免脏数据。**

##### 6.2.2 一级缓存的测试：

//其中一级缓存mybatis是默认支持的，因此不需要在配置文件中打开。

@org.junit.Test  
public void findUserById(){  
 SqlSession session = sqlSessionFactory.openSession();  
 OrderCustomMapper orderCustomMapper = session.getMapper(OrderCustomMapper.class);  
 User user = orderCustomMapper.findUserById(3);  
 System.*out*.println(user.toString());  
 User user1 = orderCustomMapper.findUserById(3);  
 System.*out*.println(user1.toString());  
 session.close();  
}

结果如下，只发出了一次sql语句



##### 6.2.3 一级缓存的应用

在实际开发中，mybaits是和spring整合开发的，事务控制在service中。

一个service方法中有很多mapper方法的调用。

Service{ //一旦调用service，会开启事务，创建sqlSession对象

//第一次调用findByID(1)

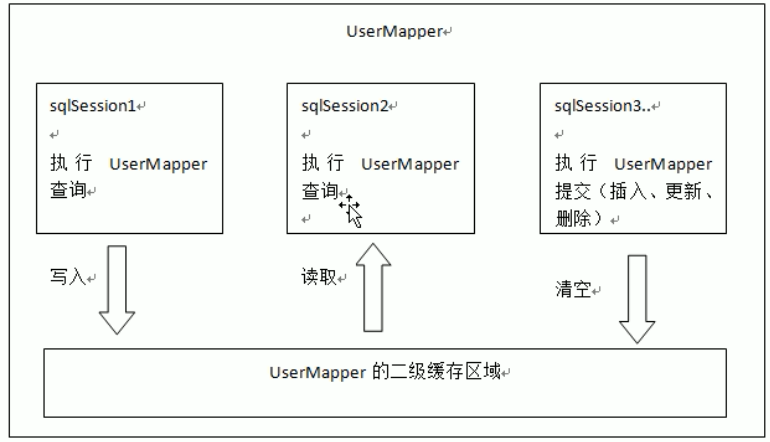
//第二次调用findByID(1)

}//方法结束，会调用sqlSession.close()。

因此，只要在service的方法体中，总是走一级缓存的，但是如果两次调用service是不走一级缓存的。

#### 6.3 二级缓存

##### 6.3.1 二级缓存的原理



Mybatis中，默认是不支持二级缓存的，因此需要开启二级缓存

sqlSession1去查询id=1的用户信息，查询到的用户信息存入到二级缓存中

sqlSession2去查询id=1的用户信息，会从二级缓存中读取。

sqlSession3执行提交（commit）操作时，会清空二级缓存。

从上面的流程可以看出，不同的sqlSession可以通过二级缓存来优化数据库的查询操作。多个sqlSession可以共享一个UserMapper的缓存区域。

其中，**二级缓存是mapper级别的**，因此二级缓存区域时按照namespace划分的。即，**每一个namespace的mapper有一个二级缓存区域**。

##### 6.3.2 二级缓存的测试

***开启二级缓存***

<settings>  
 <!--开启二级缓存-->  
 <setting name="cacheEnabled" value="true"></setting>  
</settings>

开启mapper的二级缓存区域

<cache/>

对pojo实现序列化

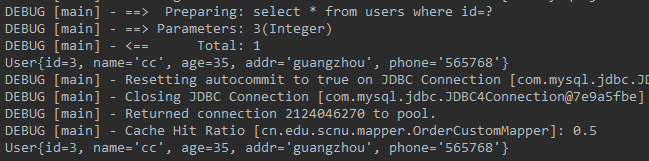
因为二级缓存的存储介质不一定是内存，因此，序列化和反序列化就很重要了。

Test.java

@org.junit.Test  
public void findUserById(){  
 SqlSession session = sqlSessionFactory.openSession();  
 OrderCustomMapper orderCustomMapper = session.getMapper(OrderCustomMapper.class);  
 User user = orderCustomMapper.findUserById(3);  
 System.*out*.println(user.toString());

//如果不关闭，是写不到二级缓存区域的。  
 session.close();  
  
  
 SqlSession session1 = sqlSessionFactory.openSession();  
 OrderCustomMapper orderCustomMapper1 = session1.getMapper(OrderCustomMapper.class);  
 User user1 = orderCustomMapper1.findUserById(3);  
 System.*out*.println(user1.toString());  
 session1.close();  
}

结果如下，只发送了一次sql语句



userCache配置：

在select语句中，可以设置userCache，禁用二级缓存，这样每次查询这样的sql时，就不会存储到二级缓存中。适应变化比较多的sql查询语句。

<select id="findUserById" parameterType="int" resultType="cn.edu.scnu.pojo.User" useCache="false">  
 select \* from users where id=#{id}  
</select>

刷新缓存

每次执行insert这种需要commit的语句，都会自动刷新缓存，如果不想刷新缓存可以设置flushCache=“false”

<insert id="insetUser" flushCache="false">

##### 6.3.3 二级缓存的应用

针对查询多，但是实时性要求不高。或者比较耗时的统计分析，电话账单等。

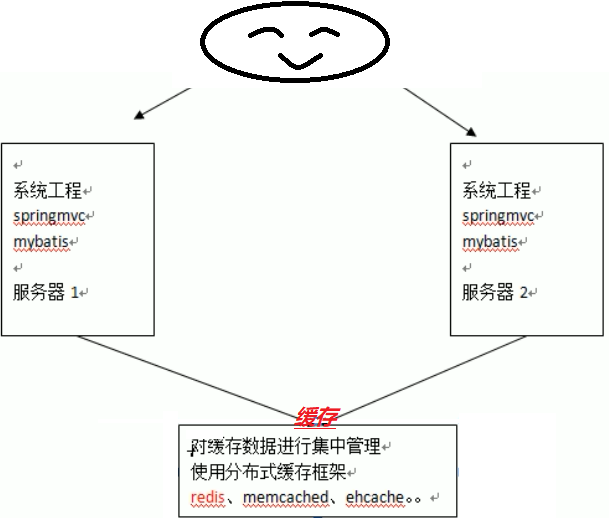
实现思路：设置刷新频率flushInterval，比如30分钟，24小时等。

局限性：

二级缓存，如果改动一个数据是全部刷新，而不是只刷新一个数据。

##### 6.3.4 mybatis整合ehcache

ehcache是一个分布式缓存。其他分布式缓存，如redis，memached等。



从上图中可以看出，如果不使用分布式缓存，那么二级缓存会在每台服务器上单独存储。

如果使用分布式缓存，相当于将缓存集中存储，都将缓存存储到“***缓存***”区域。

Mybatis中本身存在cache接口，可以实现该接口，从而实现自定义的缓存处理模式。

因此，与ehcache的整合，就是将ehcache实现cache的实现类配置到mybatis中。

***ehcache包的添加***

<!-- https://mvnrepository.com/artifact/net.sf.ehcache/ehcache-core -->  
<dependency>  
 <groupId>net.sf.ehcache</groupId>  
 <artifactId>ehcache-core</artifactId>  
 <version>2.6.11</version>  
</dependency>  
<!-- https://mvnrepository.com/artifact/org.mybatis.caches/mybatis-ehcache -->  
<dependency>  
 <groupId>org.mybatis.caches</groupId>  
 <artifactId>mybatis-ehcache</artifactId>  
 <version>1.1.0</version>  
</dependency>

***Mybatis与ehcache的整合***

<!--在mybatis-ehcache的包中找到实现cache接口的实现类，并将其指定为type参数的值-->  
<cache type="org.mybatis.caches.ehcache.EhcacheCache"/>

***ehcache.xml配置文件***

<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:noNamespaceSchemaLocation="../config/ehcache.xsd">       
  
 <diskStore path="D:\Lay\JavaEE\ehcache"/>  
 <defaultCache  
 maxElementsInMemory="10000"  
 eternal="false"  
 timeToIdleSeconds="120"  
 timeToLiveSeconds="120"  
 overflowToDisk="true"  
 maxElementsOnDisk="10000000"  
 diskPersistent="false"  
 diskExpiryThreadIntervalSeconds="120"  
 memoryStoreEvictionPolicy="LRU">   
         
 </defaultCache>  
</ehcache>

       maxElementsInMemory：缓存最大个数。

       eternal:对象是否永久有效，一但设置了，timeout将不起作用。

       timeToIdleSeconds：设置对象在失效前的允许闲置时间（单位：秒）。仅当eternal=false对象不是永久有效时使用，可选属性，默认值是0，也就是可闲置时间无穷大。

       timeToLiveSeconds：设置对象在失效前允许存活时间（单位：秒）。最大时间介于创建时间和失效时间之间。仅当eternal=false对象不是永久有效时使用，默认是0.，也就是对象存活时间无穷大。

       overflowToDisk：当内存中对象数量达到maxElementsInMemory时，Ehcache将会对象写到磁盘中。

       diskSpoolBufferSizeMB：这个参数设置DiskStore（磁盘缓存）的缓存区大小。默认是30MB。每个Cache都应该有自己的一个缓冲区。

       maxElementsOnDisk：硬盘最大缓存个数。

       diskPersistent：是否缓存虚拟机重启期数据 Whether the disk store persists between restarts of the Virtual Machine. The default value is false.

       diskExpiryThreadIntervalSeconds：磁盘失效线程运行时间间隔，默认是120秒。

       memoryStoreEvictionPolicy：当达到maxElementsInMemory限制时，Ehcache将会根据指定的策略去清理内存。默认策略是LRU（最近最少使用）。你可以设置为FIFO（先进先出）或是LFU（较少使用）。

       clearOnFlush：内存数量最大时是否清除。

### **第七章 mybatis和spring的整合**

#### 7.1 环境配置

添加spring的jar包，以及如下：

<!-- https://mvnrepository.com/artifact/org.mybatis/mybatis-spring -->  
<dependency>  
 <groupId>org.mybatis</groupId>  
 <artifactId>mybatis-spring</artifactId>  
 <version>1.3.1</version>  
</dependency>  
<!-- https://mvnrepository.com/artifact/org.mybatis/mybatis -->  
<dependency>  
 <groupId>org.mybatis</groupId>  
 <artifactId>mybatis</artifactId>  
 <version>3.4.5</version>  
</dependency>

#### 7.2 整合后的原始Dao开发

7.2.1 配置sqlSessionFactory

SqlSessionFactory在mybatis和spring的整合包下

***配置sqlSessionFactory***

<!--配置sqlSessionFactory-->  
<bean id="sessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">  
 <!--引入mybatis配置文件-->  
 <property name="configuration" value="cn/edu/scnu/conf/mybatis-conf.xml"></property>  
 <!--配置数据源-->  
 <property name="dataSource" ref="dataSource"/>  
</bean>

***配置数据源***

<!--引入db.properties-->  
<context:property-placeholder location="cn/edu/scnu/conf/db.properties"/>  
<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">  
 <property name="driverClass" value="${jdbc.driver}"/>  
 <property name="jdbcUrl" value="${jdbc.url}"/>  
 <property name="password" value="${jdbc.password}"/>  
 <property name="user" value="${jdbc.username}"/>  
 <property name="maxPoolSize" value="10"/>  
 <property name="minPoolSize" value="5"/>  
 <property name="maxIdleTime" value="5"/>  
</bean>

***Mybatis-config.xml修整后***

<configuration>  
 <!--加载属性文件-->  
 <properties resource="cn/edu/scnu/conf/db.properties"/>  
 <settings>  
 <!--开启延迟加载-->  
 <setting name="lazyLoadingEnabled" value="true"></setting>  
 <setting name="aggressiveLazyLoading" value="false"></setting>  
  
 <!--开启二级缓存-->  
 <setting name="cacheEnabled" value="true"></setting>  
 </settings>  
 <mappers>  
 <mapper resource="cn/edu/scnu/sqlMap/OrderMapper.xml"></mapper>  
 <!--<package name="cn.edu.scnu.sqlMap"></package>-->  
 </mappers>  
</configuration>

7.2.2 原始Dao的开发

***定义UserDao接口***

public interface UserDao {  
 User findUserById(Integer);  
}

***定义UserDaoImpl的实现类***

public class UserDaoImpl extends SqlSessionDaoSupport implements UserDao {

//如果没有继承SqlSessionDaoSupport，需要指定sqlSessionFactory，如下：  
 // private SqlSessionFactory sqlSessionFactory;  
  
 // public void setSqlSessionFactory(SqlSessionFactory sqlSessionFactory) {  
 // this.sqlSessionFactory = sqlSessionFactory;  
 // }

//如果继承了SqlSessionDaoSupport，由于在这个类中有setSqlSessionFactory()方法，因此可以如下编程：  
  
 public User findUserById() {

//下面是继承SqlSessionDaoSupport后

//SqlSessionFactory sqlSessionFactory = this.sqlSessionFactory;

//无论继承SqlSessionDaoSupport与否，都不需要session.close()，因为事务管理已经

//交给了Spring。  
 return null;  
 }  
}

***对UserDao注入sqlSession接口***

<bean id="user" class="cn.edu.scnu.dao.UserDaoImpl">  
 <property name="sqlSessionFactory" ref="sessionFactory"/>  
</bean>

#### 7.3 整合后的Mapper代理开发

在mapper代理开发中由于只有UserMapper接口。因此，在配置Bean的时候，需要mapper的接口实现。

配置如下

<bean id="orderCustomMapper" class="org.mybatis.spring.mapper.MapperFactoryBean">  
 <!--注入sqlSessionFactory-->  
 <property name="sqlSessionFactory" ref="sessionFactory"/>  
 <!--指定mapper的接口-->  
 <property name="mapperInterface" value="cn.edu.scnu.mapper.OrderCustomMapper"/>  
</bean>

// 此方法需要针对每一个mapper进行配置。比较麻烦，可以通过扫描器

//通过扫描器扫描，mybatis-config.xml中就不需要进行mapper的扫描了。

//自动扫描出mapper的beanid为mapper的类型首字母小写。

<bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">  
 <!--配置要扫描的包名

如果扫描多个包，用“，”隔开，并且必须在半角输入法下。

-->  
 <property name="basePackage" value="cn.edu.scnu.mapper"/>  
 <!--注入sqlSessionFactory，注意：sqlSessionFactoryBeanName-->  
 <property name="sqlSessionFactoryBeanName" value="sessionFactory"/>  
</bean>

## 第六节 SpingData