## http状态码

常见的HTTP状态码：

200 ----请求成功

301 ----资源（网页等）被永久转移到其他url

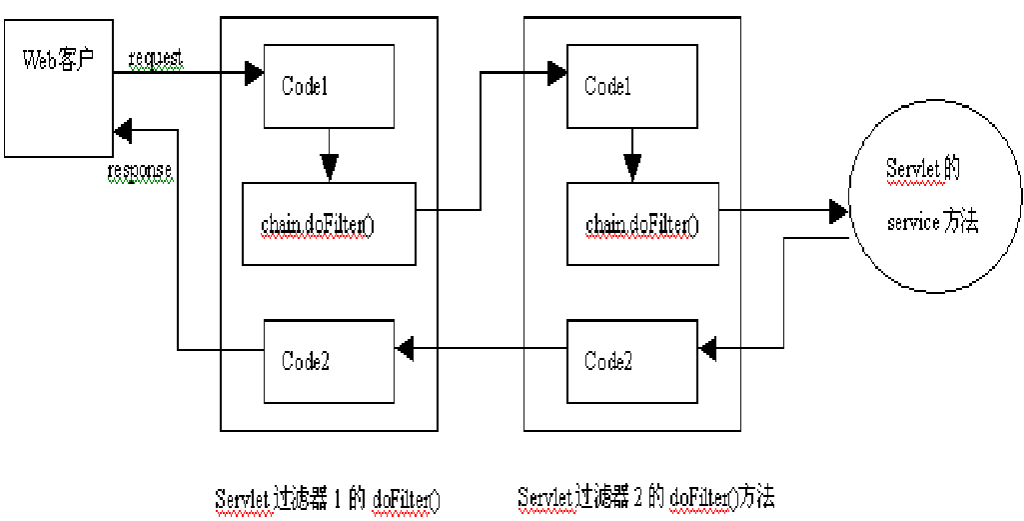
404 ----请求的页面不存在

500 ----内部服务器错误

具体请看：<http://www.runoob.com/http/http-status-codes.html>教程

## Filter学习

首先来看一下执行流程：



过滤器的生命周期，一般经历三个阶段：

初始化：

当容器第一次加载过滤器时，init()方法将被调用，该类在这个方法中包含了一个指向 Filter Config 对象的引用。

过滤：

过滤器的大多数时间都消耗在这里。doFilter方法被容器调用，同时传入分别指向这个请求/响应链中的 Servlet Request、Servlet Response 和 Filter Chain 对象的引用。然后过滤器就有机会处理请求，将处理任务传递给链中的下一个资源(通过调用 Filter Chain 对象引用上的 doFilter方法)，之后在处理控制权返回该过滤器时处理响应。

析构：

容器紧跟在垃圾收集之前调用 destroy()方法，以便能够执行任何必需的清理代码。

其中chain.doFilter(req, resp)方法是把请求转发给过滤器的下一个对象，可以是web.xml里面配置的另外的filter，也可以是servlet。

在这里我们用过滤器配置一个抛出异常到指定错误页面来实践一下。

首先是web.xml的配置：

<filter>

<filter-name>Page Request Timer</filter-name>

<filter-class>cn.filter.UserFilter</filter-class><!-- 指定Filter路径 -->

</filter>

<filter-mapping>

<filter-name>Page Request Timer</filter-name>

<servlet-name>Main Servlet</servlet-name><!-- 指定在哪个servlet下去使用Filter -->

</filter-mapping>

<servlet>

<servlet-name>Main Servlet</servlet-name>

<servlet-class>cn.servlet.FilterAction</servlet-class><!-- 指定Servlet路径 -->

</servlet>

<servlet-mapping>

<servlet-name>Main Servlet</servlet-name>

<url-pattern>/filter</url-pattern><!-- 通过什么请求来访问servlet -->

</servlet-mapping>

然后是Filter类：

package cn.filter;

import java.io.IOException;

import javax.servlet.Filter;

import javax.servlet.FilterChain;

import javax.servlet.FilterConfig;

import javax.servlet.ServletException;

import javax.servlet.ServletRequest;

import javax.servlet.ServletResponse;

import javax.servlet.annotation.WebFilter;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

import cn.exception.BsException;

@WebFilter("/UserFilter")

public class UserFilter implements Filter {

private String errorPage;//跳转的错误信息页

public void destroy() {

}

public void doFilter(ServletRequest req, ServletResponse resp,

FilterChain chain) throws IOException, ServletException {

HttpServletRequest request = (HttpServletRequest) req;

HttpServletResponse response = (HttpServletResponse) resp;

try {

chain.doFilter(req, resp);

} catch (RuntimeException e) {

if(e instanceof BsException){//如果是你定义的业务异常

request.setAttribute("BsException", e);//存储业务异常信息类

errorPage = "error.jsp";

request.getRequestDispatcher(errorPage).forward(request, response);//跳转到信息提示页面！！

}

e.printStackTrace();

}

}

public void init(FilterConfig config) throws ServletException {

}

}

其他的只要写一个错误页面error.jsp，一个异常实体类BsException，一个FileDaoImpl方法即可。在浏览器中输入：<http://localhost:8080/Day0918/filter>。

## 一些注意事项

From表单中action中地址=/项目名/servlet名，用submit按钮提交，前提是servlet中有注解@WebServlet("/UserAction")

Servlet中转发重定向地址从WebContent目录下开始

# Day0919

## 总结：今天主要是通过编写一个注册模块和用户列表的增删改查，来熟悉关于jsp+servlet+jdbc的基本操作，通过画图了解有关servlet的生命周期，http的请求与响应，http状态码等相关流程

## Ajax复习

Ajax的定义主要看四个步骤，分别是：

1. New一个新的xmlHttpRequest对象
2. 设置回调函数
3. 设置路径，请求方式，请求方法
4. 调用send传送数据

代码如下：

var xmlHttp = new XMLHttpRequest();

xmlHttp.onreadystatechange = callback;//不要用callback();

xmlHttp.open("post","/Day0919/RegisterAction?method=2&name="+name,true);

xmlHttp.send(null);

function callback(){

if(xmlHttp.status==200&&xmlHttp.readyState==4){

var flag = xmlHttp.responseText;

if(flag=="true"){

message="当前用户名已被使用";

}else{

message="";

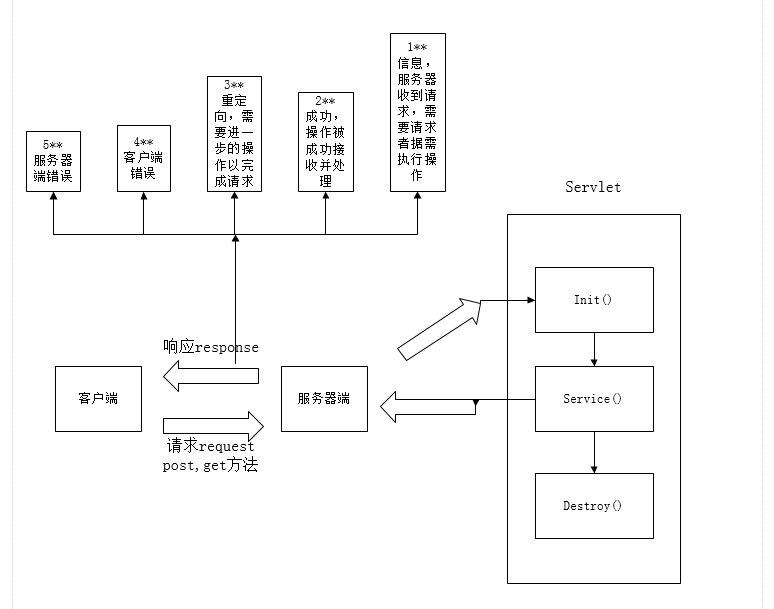
}

document.getElementById("showMsg").innerHTML=message;

}

}

## Servlet生命周期，http请求与响应，http状态码



# Day0920

## maven的功能和用法

1. **为什么要使用maven：**

它可以对项目依赖的jar包进行管理，可以让你的项目保持基本的依赖，排除冗余jar包，并且可以让你非常轻松的对依赖的jar包进行版本升级。而这些仅仅是Maven最基本的功能，它可以在这基础上对项目进行清理、编译、测试、打包、发布等等构建项目的工作。

1. **Maven的用法**

创建一个maven项目：

mvn archetype:generate

-DgroupId　　项目包名com.springmvc

-DartifactId　　项目名称springMVCStudy

-DarchetypeArtifactId　　使用的项目模板

（1）Java工程模版-maven-archetype-quickstart

（2）Web项目模版 maven-archetype-webapp

-DinteractiveMode=false 是否使用交互模式

例如创建一个Javaweb项目：

mvn archetype:generate -DgroupId=com.springmvc -DartifactId=demo-springmvc -DarchetypeArtifactId=maven-archetype-webapp

-DinteractiveMode=false

注意：

1. 控制台窗口要在想要创建项目的目录下去执行以上的命令
2. 进入到项目工程后要手动添加src/main/java、src/test/java、src/test/resources目录 ，少什么补什么
3. **Maven命令**

1．编译源代码： mvn compile

2. 编译测试代码：mvn test-compile

3. 运行测试：mvn test

4．产生site：mvn site

5. 打包：mvn package

6. 在本地Repository中安装jar：mvn install

7. 清除产生的项目：mvn clean

8. 生成eclipse项目：mvn eclipse:eclipse

9. 生成idea项目：mvn idea:idea

10. 组合使用goal命令，如只打包不测试：mvn -Dtest package

11. 编译测试的内容：mvn test-compile

12. 只打jar包: mvn jar:jar

13. 只测试而不编译，也不测试编译：mvn test -skipping compile -skipping test-compile

( -skipping 的灵活运用，当然也可以用于其他组合命令)

14. 清除eclipse的一些系统设置:mvn eclipse:clean

1. **pom.xml文件节点元素说明**：

　<project> pom文件的顶级节点  
<modelVersion>　object model版本，对Maven2和Maven3来说，只能是4.0.0　  
<groupId>　　　　项目创建组织的标识符，一般是域名的倒写  
<artifactId>　　　定义了项目在所属组织的标识符下的唯一标识，一个组织下可以有多个项目  
<version>　　　　当前项目的版本，SNAPSHOT，表示是快照版本，在开发中

　<packaging>　　 打包的方式，有jar、war、ear等  
　<name>　　　　　项目的名称  
　<url>　　　　　　项目的地址

<properties>　　属性配置，比如：

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  
<dependencies>　　   构建项目依赖的jar

其中**由groupId、artifactId和version唯一的确定了一个项目坐标**

**注：maven项目几种部署：**

<http://www.micmiu.com/software/build/maven-web-eclipse-deploy/>

## ****SVN的功能与用法****

1. **svn是什么**

svn是版本管理工具，譬如团队进行项目开发，项目代码都储存在服务器上，成员可用svn在本地获得并更新代码

1. **为什么要使用它**
2. 备份工作是十分重要的，您永远不知道计算机上的硬盘何时会坏掉。
3. 版本控制非常重要。您无法保证手头上最新版本永远都是对的。很多时候，在经过数天努力工作之后，您才发现走错方向，需要将所有的修改恢复到数天前的版本。没有几个人能够完全记住自己修改过什么东西。如果没有做好版本控制，那么，最差的状况就是要全部重来。
4. 伙伴间的数据同步也是非常重要的，很多时候，除了您个人外，还有其他的同伴也需要您的档案，怎么把最新的档案提供给伙伴呢，就是通过svn。
5. 如果没有一个更好的办法，备份不同版本是很耗费硬盘空间的。
6. **用法**

详情请看如下教程：

<http://blog.csdn.net/u014486880/article/details/45822405>

## MYSQL语法大全

1. 命令行进入mysql数据库：mysql –u root –p
2. 退出MySQL数据库：exit;
3. 修改密码：首先在DOS 下进入mysql安装路径的bin目录下，然后键入以下命令:

mysqladmin -uroot –p原密码 password 新密码;

1. 增加用户：

#格式:grant 权限 on 数据库.\* to 用户名@登录主机 identified by '密码'

/\*

如，增加一个用户user1密码为password1，让其可以在本机上登录， 并对所有数据库有查询、插入、修改、删除的权限。首先用以root用户连入mysql，然后键入以下命令：

grant select,insert,update,delete on \*.\* to user1@localhost Identified by "password1";

如果希望该用户能够在任何机器上登陆mysql，则将localhost改为"%"。

如果你不想user1有密码，可以再打一个命令将密码去掉。

grant select,insert,update,delete on mydb.\* to user1@localhost identified by "";

\*/

grant all privileges on wpj1105.\* to sunxiao@localhost identified by '123'; #all privileges 所有权限

1. Mysql数据库操作基础
   1. 显示数据库

Show databases;

* 1. 判断是否存在数据库wpj，有的话先删除

Drop database if exists wpj;

* 1. 创建数据库

Create database wpj;

* 1. 删除数据库：

Drop database wpj;

* 1. 使用数据库：

Use wpj;

* 1. 显示数据库的表：

Show tables

* 1. 先判断表是否存在,存在先删除:

drop table if exists student;

* 1. 创建表：

Create table student(

Id int auto\_increment primary key,

Name varchar(50),

Sex varchar(50),

Date varchar(50)

)default charset=utf8;

* 1. 删除表：

drop table student;

* 1. 查看表结构：

Desc student;

* 1. 插入数据：

insert into student values(null,'aa','男','1988-10-2');  
insert into student values(null,'bb','女','1889-03-6');  
insert into student values(null,'cc','男','1889-08-8');  
insert into student values(null,'dd','女','1889-12-8');  
insert into student values(null,'ee','女','1889-09-6');  
insert into student values(null,'ff','null','1889-09-6');

* 1. 查询表中数据

select \* from student;  
select id,name from student;

* 1. 修改某一条数据

update student set sex='男' where id=4;

* 1. 删除数据

delete from student where id=5;

* 1. And 且

select \* from student where date>'1988-1-2' and date<'1988-12-1';

* 1. Or 或

select \* from student where date<'1988-11-2' or date>'1988-12-1';

* 1. Between

select \* from student where date between '1988-1-2' and '1988-12-1';

* 1. 查询指定集合内的数据

select \* from student where id in (1,3,5);

* 1. 排序 asc升序，desc降序

select \* from student order by id asc;

* 1. 分组查询聚合函数

select max(id),name,sex from student group by sex;

select min(date) from student;

select avg(id) as '求平均' from student;

select count(\*) from student;   #统计表中总数

select count(sex) from student;   #统计表中性别总数  若有一条数据中sex为空的话,就不予以统计~

select sum(id) from student;

* 1. 查询从坐标x开始，往后y条数据

select \* from student limit x,y(下标从0开始，包括x);

* 1. 修改表名：

alter table c rename to a;

* 1. 向表中添加一个字段：  
     alter table test add  columnname varchar(20);
  2. 修改表中某个字段的名字

alter table test change name uname varchar(50);

* 1. 导入.sql文件命令（例如D:/mysql.sql）

source d:/mysql.sql;

## JDBC事务和原理

1. 事务的特性：
2. 原子性

事务是数据库的逻辑工作单位，而且是必须是原子工作单位，对于其数据修改，要么全部执行，要么全部不执行。

1. 一致性

事务在完成时，必须是所有的数据都保持一致状态。在相关数据库中，所有规则都必须应用于事务的修改，以保持所有数据的完整性。

1. 隔离性

一个事务的执行不能被其他事务所影响。

1. 持久性

一个事务一旦提交，事物的操作便永久性的保存在DB中。即使此时再执行回滚操作也不能撤消所做的更改。

事务(Transaction):是并发控制的单元，是用户定义的一个操作序列。这些操作要么都做，要么都不做，是一个不可分割的工作单位。通过事务，sql server 能将逻辑相关的一组操作绑定在一起，以便服务器 保持数据的完整性。事务通常是以begin transaction开始，以commit或rollback结束。Commint表示提交，即提交事务的所有操作。具体地说就是将事务中所有对数据的更新写回到磁盘上的物理数据库中去，事务正常结束。Rollback表示回滚，即在事务运行的过程中发生了某种故障，事务不能继续进行，系统将事务中对数据库的所有已完成的操作全部撤消，滚回到事务开始的状态。

1. 事务并发处理可能引起的问题

脏读(dirty read) 一个事务读取了另一个事务尚未提交的数据，

在事务未commit前进行查询

不可重复读(non-repeatable read) 一个事务的操作导致另一个事务前后两次读取到不同的数据

事务A有两次查询，事务B有一次修改操作，顺序是A第一次查询-->B修改，comiit🡪B第二次查询发现数据不一样

幻读(phantom read) 一个事务的操作导致另一个事务前后两次查询的结果数据量不同。

事务A有两次查询数据量（聚合函数），事务B有一次添加或删除操作，顺序是A第一次查询-->B修改，comiit🡪B第二次查询发现数据不一样

举例：

事务A、B并发执行时，

当A事务update后，B事务select读取到A尚未提交的数据，此时A事务rollback，则B读到的数据是无效的"脏"数据。

当B事务select读取数据后，A事务update操作更改B事务select到的数据，此时B事务再次读去该数据，发现前后两次的数据不一样。

当B事务select读取数据后，A事务insert或delete了一条满足A事务的select条件的记录，此时B事务再次select，发现查询到前次不存在的记录("幻影")，或者前次的某个记录不见了。

1. JDBC的事务支持

主要体现在三个方面：

1.自动提交模式(Auto-commit mode)

Connection提供了一个auto-commit的属性来指定事务何时结束。

a.当auto-commit为true时，当每个独立SQL操作的执行完毕，事务立即自动提交，也就是说每个SQL操作都是一个事务。

一个独立SQL操作什么时候算执行完毕，JDBC规范是这样规定的：

对数据操作语言(DML，如insert,update,delete)和数据定义语言(如create,drop)，语句一执行完就视为执行完毕。

对select语句，当与它关联的ResultSet对象关闭时，视为执行完毕。

对存储过程或其他返回多个结果的语句，当与它关联的所有ResultSet对象全部关闭，所有update count(update,delete等语句操作影响的行数)和output parameter(存储过程的输出参数)都已经获取之后，视为执行完毕。

b. 当auto-commit为false时，每个事务都必须显示调用commit方法进行提交，或者显示调用rollback方法进行回滚。auto-commit默认为true。

JDBC提供了5种不同的事务隔离级别，在Connection中进行了定义。

2.事务隔离级别(Transaction Isolation Levels)

JDBC定义了五种事务隔离级别：

TRANSACTION\_NONE JDBC驱动不支持事务

TRANSACTION\_READ\_UNCOMMITTED 允许脏读、不可重复读和幻读。

TRANSACTION\_READ\_COMMITTED 禁止脏读，但允许不可重复读和幻读。

TRANSACTION\_REPEATABLE\_READ 禁止脏读和不可重复读，单运行幻读。Mysql默认事务隔离级别

TRANSACTION\_SERIALIZABLE 禁止脏读、不可重复读和幻读。

3.保存点(SavePoint)

JDBC定义了SavePoint接口，提供在一个更细粒度的事务控制机制。当设置了一个保存点后，可以rollback到该保存点处的状态，而不是rollback整个事务。

Connection接口的setSavepoint和releaseSavepoint方法可以设置和释放保存点。

JDBC规范虽然定义了事务的以上支持行为，但是各个JDBC驱动，数据库厂商对事务的支持程度可能各不相同。如果在程序中任意设置，可能得不到想要的效果。为此，JDBC提供了DatabaseMetaData接口，提供了一系列JDBC特性支持情况的获取方法。比如，通过DatabaseMetaData.supportsTransactionIsolationLevel方法可以判断对事务隔离级别的支持情况，通过DatabaseMetaData.supportsSavepoints方法可以判断对保存点的支持情况。

四．PreparedStatement和statement区别：

PreparedStatement支持动态参数化查询

preparedStatement使用预处理语句查询比statement要快

PreparedStatement可以阻止常见的sql注入攻击

五．InnoDB锁描述

有两种锁分别是：共享锁和排它锁

共享锁：不同事务之间可以查询语句没有影响，增删改会等待锁释放，添加方法：在一个事务的查询语句后面加上lock in share mode

排它锁：增删改查都要等待锁释放，添加方法：在一个事务的查询语句后面加上 for update

注意要在不同事务里面，可以这么操作：命令行mysql下set autocommit = 0;取消自动commit，start transaction开启事务，在eclipse下connection对象调用setAutoCommit(false);禁止自动提交，相当于开启事务，后面要有commit操作和rollback操作。

# Day0921

## 总结：昨天先是学习maven和svn的功能与用法。然后过了一遍mysql的语法，接着对JDBC事务原理梳理了一遍。今天主要是搭建好spring的开发环境，了解spring一些基本功能

**什么是AOP**

AOP:面向切面编程，通过预编译方式和运行期动态代理实现程序功能的统一维护的一种技术  
功能例如：日志记录，性能统计，安全控制，事务处理，异常处理等等

**AOP实现方式**

* 预编译  
  -AspectJ
* 运行期动态代理  
  -SpringAOP,JbossAOP

**几个相关的概念**

| **名称** | **说明** |
| --- | --- |
| 切面（Aspect） | 一个关注点的模块话，这个关注点可能会横切多个对象 |
| 连接点（Joinpoint） | 程序执行过程中某个特定的点 |
| 通知（Advice） | 在切面的某个特定的连接点上执行的动作 |
| 切入点（Pointcut） | 匹配连接点的断言，在AOP中通知和一个切入点表达式关联 |
| 引入（Introduction） | 在不修改类代码的前提下，为类添加新的方法和属性 |
| 目标对象（Target Object） | 被一个或者多个切面所通知的对象 |
| AOP代理（AOP Proxy） | Aop框架创建的对象，用来实现切面契约 |
| 织入（Weaving） | 把切面连接到其他的应用程序类型或者对象上，并且创建一个被通知的对象，分为 编译时织入，类加载时织入，执行时织入 |

**AOP术语通俗解释**

* 通知、增强处理（Advice） 就是切面里面所有定义的方法
* 连接点（JoinPoint） 这个就更好解释了，就是spring允许你是通知（Advice）的地方，那可就真多了，基本每个方法的前、后（两者都有也行），或抛出异常是时都可以是连接点，spring只支持方法连接点。
* 切入点（Pointcut） 上面说的连接点的基础上，来定义切入点，你的一个类里，有15个方法，那就有十几个连接点了对吧，但是你并不想在所有方法附件都使用通知（使用叫织入，下面再说），你只是想让其中几个，在调用这几个方法之前、之后或者抛出异常时干点什么，那么就用切入点来定义这几个方法，让切点来筛选连接点，选中那几个你想要的方法。
* 切面（Aspect） 切面是通知和切入点的结合。现在发现了吧，没连接点什么事，链接点就是为了让你好理解切点搞出来的，明白这个概念就行了。通知说明了干什么和什么时候干（什么时候通过方法名中的befor，after，around等就能知道），二切入点说明了在哪干（指定到底是哪个方法），这就是一个完整的切面定义。
* 引入（introduction） 允许我们向现有的类添加新方法属性。这不就是把切面（也就是新方法属性：通知定义的）用到目标类中吗
* 目标（target） 引入中所提到的目标类，也就是要被通知的对象，也就是真正的业务逻辑，他可以在毫不知情的情况下，被咱们织入切面。二自己专注于业务本身的逻辑。
* 代理（proxy） 怎么实现整套AOP机制的，都是通过代理。
* 织入（weaving） 把切面应用到目标对象来创建新的代理对象的过程。有三种方式，spring采用的是运行时。
* 目标对象 – 项目原始的Java组件。
* AOP代理 – 由AOP框架生成java对象。
* AOP代理方法 = advice +　目标对象的方法。

**代码演示**

首先配置beans.xml

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

<beans

xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:p="http://www.springframework.org/schema/p"

xmlns:context="http://www.springframework.org/schema/context"

xmlns:aop="http://www.springframework.org/schema/aop"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context-3.0.xsd

http://www.springframework.org/schema/aop

http://www.springframework.org/schema/aop/spring-aop-3.0.xsd

"

default-autowire="no">

<bean id="BaseDao" class="cn.dao.impl.BaseDaoImpl" scope="prototype">

</bean>

<bean id="ServiceDao" class="cn.service.impl.ServiceDaoImpl">

</bean>

<bean id="aspectDemo" class="cn.aspect.AspectDemo">

</bean>

<aop:config>

<aop:aspect id="aspectDemoAop" ref="aspectDemo">

<aop:pointcut expression="execution(\* cn.service.impl.ServiceDaoImpl.\*(..))" id="aspectPointCut"/>

<aop:before method="before" pointcut-ref="aspectPointCut"/>

<aop:after-returning method="afterReturning" pointcut-ref="aspectPointCut"/>

<aop:after-throwing method="afterThrowing" pointcut-ref="aspectPointCut"/>

<aop:after method="after" pointcut-ref="aspectPointCut"/>

<aop:around method="around" pointcut-ref="aspectPointCut"/>

</aop:aspect>

</aop:config>

</beans>

其中<aop:aspect id="aspectDemoAop" ref="aspectDemo"> </aop:aspect>是对切面的定义，要写在aop:config属性中，pointcut是切入点的意思 在expression写触发时会发生的方法事件，\* cn.service.impl.ServiceDaoImpl.\*(..)这一行代表的意思是在ServiceDaoImpl类下所有的方法执行都会触发。   
<aop:before method="before" pointcut-ref="aspectPointCut"/>  
<aop:after-returning method="afterReturning" pointcut-ref="aspectPointCut"/>  
<aop:after-throwing method="afterThrowing" pointcut-ref="aspectPointCut"/>  
<aop:after method="after" pointcut-ref="aspectPointCut"/>  
<aop:around method="around" pointcut-ref="aspectPointCut"/>  
这五行都是通知的不同类型分别是方法事件触发前执行，方法事件出发后返回执行，方法事件抛出异常后执行，方法结束后执行，方法环绕执行

package cn.aspect;

import org.aspectj.lang.ProceedingJoinPoint;

public class AspectDemo {

public void before(){

System.out.println("通知前已响应");

}

public void afterReturning(){

System.out.println("返回后响应");

}

public void afterThrowing(){

System.out.println("抛出异常后响应");

}

public void after(){

System.out.println("方法结束后响应");

}

public Object around(ProceedingJoinPoint pjp){

Object obj = null;

try {

System.out.println("1");

obj = pjp.proceed();

System.out.println("2");

} catch (Throwable e) {

e.printStackTrace();

}

return obj;

}

}

运行结果：

log4j:WARN No appenders could be found for logger (org.springframework.context.support.ClassPathXmlApplicationContext).

log4j:WARN Please initialize the log4j system properly.

通知前已响应

1

逻辑层判断

返回后响应

方法结束后响应

2

**after-returing和after-throwing和after**

after-returing和after-throwing两者只能其中一个执行，若抛出异常则after-throwing，否则就after-returing，after则在两者之后执行，且与前两者是否执行无关。  
round的话必须在方法里写ProceedingJoinPoint参数才能执行

**execution表达式例子如下：**

任意公共方法的执行：  
　　　　execution(public \* *(..))  
　　任何一个以“set”开始的方法的执行：  
　　　　execution(* set\*(..))  
　　AccountService 接口的任意方法的执行：  
　　　　execution(\* com.xyz.service.AccountService.*(..))  
　　定义在service包里的任意方法的执行：  
　　　　execution(* com.xyz.service.*.*(..))  
　　定义在service包和所有子包里的任意类的任意方法的执行：  
　　　　execution(\* com.xyz.service..*.*(..))  
　　定义在pointcutexp包和所有子包里的JoinPointObjP2类的任意方法的执行：  
　　　　execution(\* com.test.spring.aop.pointcutexp..JoinPointObjP2.\*(..))")

**Spring声明式事务管理**

声明式事务管理是建立在AOP的基础之上的，其本质是对方法前后进行拦截，然后在目标方法开始之前创建或者加入一个事务，在执行完目标方法之后根据执行情况提交或者回滚事务。声明式事务最大的优点就是不需要通过编程的方式管理事务，这样就不需要在业务逻辑代码中掺杂事务管理的代码，只需在配置文件中做相关的事务规则声明(或通过基于@Transactional注解的方式)，便可以将事务规则应用到业务逻辑中。  
Spring配置文件加入如下配置

<!-- Spring事务管理 -->

<bean id="txDao" class="cn.spring.dao.impl.TxDaoImpl">

<property name="jdbcTemplate" ref="jdbcTemplate"></property>

</bean>

<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">

<property name="driverClassName" value="com.mysql.jdbc.Driver" />

<property name="url" value="jdbc:mysql://localhost:3306/test?characterEncoding=utf-8" />

<property name="username" value="root" />

<property name="password" value="123456" />

</bean>

<bean name="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">

<property name="dataSource" ref="dataSource" />

</bean>

<tx:advice id="advice" transaction-manager="transactionManager">

<tx:attributes>

<tx:method name="\*" rollback-for="Exception"/>

</tx:attributes>

</tx:advice>

<aop:config>

<aop:pointcut id="testService" expression="execution (\* cn.spring.dao.impl.TxDaoImpl.\*(..))"/>

<aop:advisor advice-ref="advice" pointcut-ref="testService"/>

</aop:config>

TxDaoImpl：

package cn.spring.dao.impl;

import org.springframework.jdbc.core.JdbcTemplate;

public class TxDaoImpl {

private JdbcTemplate jdbcTemplate;

public JdbcTemplate getJdbcTemplate() {

return jdbcTemplate;

}

public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {

this.jdbcTemplate = jdbcTemplate;

}

public void execute() throws Exception{

String sql = "INSERT INTO aaa(number) VALUES(?)";

for(int i=1;i<=20;i++){

jdbcTemplate.update(sql,i);

if(i==8) throw new Exception();

}

}

}

当执行到第八条插入语句时显然会抛出异常，此时事务发生回滚，数据库中将不会有任何新添加的数据

**Spring-IOC小结**

**Bean容器初始化的三种方式：**

* 本地文件
* ClassPath
* Web应用中依赖servlet或Listener

**Spring注入的两种方式：**

* 设值注入：

<bean id="injectionService" class="com.imooc.ioc.injection.service.InjectionServiceImpl">

<property name="injectionDAO" ref="injectionDAO"></property> -->

</bean>

<bean id="injectionDAO" class="com.imooc.ioc.injection.dao.InjectionDAOImpl"></bean>

</beans>

* 构造注入：

<bean id="injectionService" class="com.imooc.ioc.injection.service.InjectionServiceImpl">

<constructor-arg name="injectionDAO" ref="injectionDAO"></constructor-arg>

</bean>

<bean id="injectionDAO" class="com.imooc.ioc.injection.dao.InjectionDAOImpl"></bean>

</beans>

**Bean的作用域：**

1、singleton:当一个bean的作用域为singleton, 那么Spring IoC容器中只会存在一个共享的bean实例，并且所有对bean的请求，只要id与该bean定义相匹配，则只会返回bean的同一实例。 注意：Singleton作用域是Spring中的缺省作用域。要在XML中将bean定义成singleton，可以这样配置：   
2、prototype：一个bean定义对应多个对象实例。Prototype作用域的bean会导致在每次对该bean请求（将其注入到另一个bean中，或者以程序的方式调用容器的getBean()方法）时都会创建一个新的bean实例。根据经验，对有状态的bean应该使用prototype作用域，而对无状态的bean则应该使用singleton作用域。  
3、request：在一次HTTP请求中，一个bean定义对应一个实例；即每次HTTP请求将会有各自的bean实例， 它们依据某个bean定义创建而成。该作用域仅在基于web的Spring ApplicationContext情形下有效。 考虑下面bean定义： <bean id="loginAction" class=cn.csdn.LoginAction" scope="request"/> 针对每次HTTP请求，Spring容器会根据loginAction bean定义创建一个全新的LoginAction bean实例， 且该loginAction bean实例仅在当前HTTP request内有效，因此可以根据需要放心的更改所建实例的内部状态， 而其他请求中根据loginAction bean定义创建的实例，将不会看到这些特定于某个请求的状态变化。 当处理请求结束，request作用域的bean实例将被销毁。  
4、session：在一个HTTP Session中，一个bean定义对应一个实例。该作用域仅在基于web的Spring ApplicationContext情形下有效。 考虑下面bean定义： 针对某个HTTP Session，Spring容器会根据userPreferences bean定义创建一个全新的userPreferences bean实例， 且该userPreferences bean仅在当前HTTP Session内有效。 与request作用域一样，你可以根据需要放心的更改所创建实例的内部状态，而别的HTTP Session中根据userPreferences创建的实例， 将不会看到这些特定于某个HTTP Session的状态变化。 当HTTP Session最终被废弃的时候，在该HTTP Session作用域内的bean也会被废弃掉。  
5、global session：在一个全局的HTTP Session中，一个bean定义对应一个实例。典型情况下，仅在使用portlet context的时候有效。该作用域仅在基于web的Spring ApplicationContext情形下有效。

**Bean的生命周期**

* 定义
* 初始化
* 使用
* 销毁  
  Bean初始化/销毁的三种方法：  
  （1）通过实现 InitializingBean/DisposableBean 接口来定制初始化之后/销毁之前的操作方法；要覆盖相应的方法

（2）通过 元素的 init-method/destroy-method属性指定初始化之后 /销毁之前调用的操作方法；

（3）还有一种是

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

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans

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

default-init-method="defautInit" default-destroy-method="defaultDestroy">

这三种方法的执行顺序是（1）>（2）但（1）或者（2）实现的时候（3）不会实现，而且（3）可以有定义而没方法，但（2）只要定义了方法名就一定要在相应的类中写方法

**Spring容器为什么要设置单例：**

理由很简单，一方面spring容器在初始化时就实例化了bean类，提高了效率，另一方面也能大大的减少内存开销，但仍然存在一些问题：  
比如在Controller类中如果设置了非final类型修饰全局变量的话，可能会出现线程安全问题，这个时候通过注解@Scope("prototype")来把Controller类设置为原型模式。

**Spring自动装配**

No:不做任何操作  
ByName：根据属性名自动装配。此选项将检查容器并根据名字查找与属性完全一致的bean，并将其与属性自动装配。  
ByType：如果容器中存在一个预指定属性类型相同的bean，那么将于该属性自动装配，并不考虑ID，如果存在多个该类型的bean，抛出异常，找不到相匹配的则什么事都不会发生  
constructor：与ByType方式类似，不同之处在于她应用于构造器参数，如果容器中没有找到与构造器参数类型一致的bean，那么抛出异常。  
即不必在bean属性下写property属性或者constructor-arg属性，只需在beans xmlns定义default-autowire即可。  
注意byName和byType都是通过set方法设值注入的，constructor通过构造器注入，记住要写出原本类的默认构造方法

# Day0922

## 站会：昨天主要是针对SpringAOP和IOC做了系统的复习。

任务：全面学习spring事务（例如：什么是事务，事务的隔离性，事务并发引发的问题以及解决，spring事务的优缺点，spring事务的传播性，隔离级别等等）

## 什么是事务？

首先我们要了解一下并发控制，并发控制是指多个用户同时更新运行时，用于保护数据库完整性的各种技术。简单的讲是为了保证一个用户的操作不会对另外一个用户的操作造成不利的影响。而事务，就是并发控制的基本单位。是用户定义的一个操作序列。这些操作要么全做，要么全不做，是一个不可分割的工作单位。

事务通常以begin transaction开始，以commit和rollback结束。

## 事务的特性（ACID）

1. 原子性（Atonicity）

要么全做，要么不做

1. 一致性（Consistency）

事务的执行结果要使数据库从一个一致性状态变成另一个一致性状态

1. 隔离性（Isolation）

一个事务的执行不能被其他事务干扰

1. 持续性/永久性（Durability）

一个事务一旦提交，他对数据库中数据的改变应该是永久性的

## 事务并发引起的问题

1. 脏读

在事务未commit前进行查询

1. 不可重复读

有两次查询，事务B有一次修改操作，顺序是A第一次查询-->B修改，comiitB第二次查询发现数据不一样

1. 幻读

事务A有两次查询数据量（聚合函数），事务B有一次添加或删除操作，顺序是A第一次查询-->B修改，comiitB第二次查询发现数据不一样

1. 第一类丢失更新(lost update)： 在完全未隔离事务的情况下，两个事务更新同一条数据资源，某一事物异常终止，回滚造成第一个完成的更新也同时丢失。
2. 第二类丢失更新(second lost updates)：是不可重复读的特殊情况，如果两个事务都读取同一行，然后两个都进行写操作，并提交，第一个事务所做的改变就会丢失。

## 事务的隔离级别

Mysql数据库给我们提供的四种隔离级别：

* 1. Serializable（串行化）：可避免脏读，不可重复读，幻读的发生。但是性能极差，一般不会这么用。
  2. Repeatable read（可重复读）：可以避免脏读，不可重复读的发生。（mysql默认）
  3. Read commited（读已经提交）：可避免脏读的发生
  4. Read uncommitted（读未提交）：最低级别，任何情况都无法保证。

## 事务的传播行为

我的感觉是：一个被定义为事务的方法A在执行的过程中，去调用了另一个被定义为事务的方法B，此时方法B的事务是该新建，还是嵌入方法A事务，这就是事务传播行为去定义的。

Propagation\_required: 如果当前存在事务，则加入该事务；如果当前没有事务，则创建一个新的事务。这是默认值。

Propagation\_requires\_new:创建一个新的事务，如果当前存在事务，则将其高高挂起

Propagation\_supports:如果当前存在事务，则加入该事务，如果当前没有事务，则以非事务的方式继续运行

Propagation\_not\_supported:以非事务方式运行，如果当前存在事务，则把事务高高挂起。

Propagation\_never:以非事务方式运行，如果当前存在事务，抛异常

Propagation\_mandatory:如果当前存在事务，则加入该事务，如果当前没有事务，抛异常

Propagation\_nested:如果当前存在事务，则创建一个事务作为当前事务的嵌套事务来运行；如果当前没有事务，则该取值等价于TransactionDefinition.PROPAGATION\_REQUIRED。

疑难杂点：

Required事务A和事务B只要有一处发生异常都会回滚

Requires\_new：假设事务A调用事务B，事务B抛出异常，那么事务B肯定回滚，但是如果此时事务A去捕获这个异常，那么事务A还是可以正常commit的

Nested：Nested的事务和他的父事务是相依的，他的提交是要等和他的父事务一块提交的。也就是说，如果父事务最后回滚，他也要回滚的。

## Spring事务

Spring事务解决了全局事务和本地事务的缺陷，支持编程式事务管理和声明式事务管理。

声明式事务管理又分为两大部分：

1. 基于xml配置的事务管理
2. 注解

第一点昨天已经做了一个demo，今天做第二点有关的demo

先讲一些有关注解的基本配置：

@Component，@Service，@Controller功能相同，字面含义不同，方便开发者了解这是哪一个类，都是在类名上面添加，表明这个类的实例化交给Spring容器去托管

@Autowired和@Resource

这两个意思都是在某一个类中注入其他类的实例，一般都写在类属性的前面，其中@Autowired是用byType自动注入，@Resource默认是用byName，也可以用byType

@PostConstruct和@PreDestroy ，这两个只能应用于方法上，标注了 @PostConstruct 注释的方法将在类实例化后调用，而标注了 @PreDestroy 的方法将在类销毁之前调用。

@Scope指定bean作用域范围

@Transactional()事务注解

**一些比较常用的属性：**

1. rollbackFor：设置哪些异常会执行回滚
2. noRollbackFor:设置抛出哪些异常不会执行回滚
3. timeout：事务超时间设置
4. isolation：可选的事务隔离级别设置：默认为数据源的默认隔离级别
5. propagation：可选的事务传播行为设置：默认为required
6. readOnly：读写或者只读事务，默认读写

如果全部都是读取操作的话，可以设置为只读事务，这是一个优化的提示，在一些情况下，一些事务策略能够起到显著的最优化效果，例如在使用Object/Relational映射工具（如：Hibernate或TopLink）时避免dirty checking（试图“刷新”）。

## 锁概念

**有了事务为什么还要乐观锁悲观锁**

事务是粗粒度的概念，乐观锁悲观锁可以更细粒度的控制

事务可以用来保证一致性和隔离性，但锁用来保证并发性，隔离性和并发性有点类似，但是隔离性只是保证不会出现相互读取中间数据，却无法解决并发的问题

**乐观锁：**

乐观锁不是数据库自带的，需要我们自己去实现。乐观锁是指操作数据库时(更新操作)，想法很乐观，认为这次的操作不会导致冲突，在操作数据时，并不进行任何其他的特殊处理（也就是不加锁），而在进行更新后，再去判断是否有冲突了。

**实现原理**：在表的数据进行操作时，先给数据表加一个版本（version）字段，每操作一次，

将那条记录的版本号加1. 也就是先查询出那条记录，获取出version字段,如果要对那条记录进行操作(更新),则先判断此刻version的值是否与刚刚查询出来时的version的值相等，如果相等，则说明这段期间，没有其他程序对其进行操作，则可以执行更新，将version字段的值加1；如果更新时发现此刻的version值与刚刚获取出来的version的值不相等，则说明这段期间已经有其他程序对其进行操作了，则不进行更新操作。Hibernate已经封装好了乐观锁的实现

**悲观锁：**

在操作数据时，认为此操作会出现数据冲突，所以在进行每次操作时都要通过获取锁才能进行对相同数据的操作，这点跟java中的synchronized很相似，所以悲观锁需要耗费较多的时间。另外与乐观锁相对应的，悲观锁是由数据库自己实现了的，要用的时候，我们直接调用数据库的相关语句就可以了。共享锁和排它锁都属于悲观锁

**实现方式**：

共享锁：在执行语句后面加上lock in share mode就代表对某些资源加上共享锁了。可以查，不能增删改。

排它锁：在需要执行的语句后面加上for update就可以了，增删改查都不可以。

**各自的优点和缺点：**

乐观锁并发控制相信事务之间的数据竞争(data race)的概率是比较小的，因此尽可能直接做下去，直到提交的时候才去锁定，所以不会产生任何锁和死锁。但如果直接简单这么做，还是有可能会遇到不可预期的结果，例如两个事务都读取了数据库的某一行，经过修改以后写回数据库，这时就遇到了问题。

悲观锁并发控制实际上是”先取锁再访问”的保守策略，为数据处理的安全提供了保证。但是在效率方面，处理加锁的机制会让数据库产生额外的开销，还有增加产生死锁的机会；另外，在只读型事务处理中由于不会产生冲突，也没必要使用锁，这样做只能增加系统负载；还有会降低了并行性，一个事务如果锁定了某行数据，其他事务就必须等待该事务处理完才可以处理那行数

**总结**

乐观锁适用于多读的应用类型，这样可以提高吞吐量，像数据库如果提供类似于write\_condition机制的其实都是提供的乐观锁。 相反，如果经常发生冲突，上层应用会不断进行 retry，这样反而降低了性能，所以这种情况下用悲观锁比较合适