## **SSM 框架相关知识点**

Mybatis:

构建者模式：把对象创建的细节隐藏，使用者只需要调用方法即可，sqlsessionfactorybuilder（）；

工厂模式：实现了程序间的解耦操作。

代理模式：proxy.Newproxyinstance(classloader,interface,invocationHandler)在不改变源码的基础上，对已有方法进行增强。

Spring:

IoC（Inverse Of Control： 反转控制）:控制反转，（解耦合）将对象间的依赖关系交给 Spring 容器，使用配置文件来创建所依赖的对象，由主动创建对象改为了被动方式；

DI(依赖注入)：依赖注入有set注入，构造器注入，接口注入，**是**指程序运行过程中，如果需要调用另一个对象协助时，无须在代码中创建被调用者，而是**依赖**于外部的**注入**。通过标签bean的方式导入。

AOP:面向切面编程，将功能代码从业务逻辑代码中分离出来。

1. AOP 的实现方式有哪几种？如何选择？

JDK 动态代理实现和 cglib 实现。

如果目标对象实现了接口，默认情况下会采用 JDK 的动态代理实现 AOP，也可以强制使用 cglib 实现 AOP；

如果目标对象没有实现接口，必须采用 cglib 库，Spring 会自动在 JDK 动态代理和 cglib 之间转换。

2.JDK 动态代理如何实现？

答：JDK 动态代理，只能对实现了接口的类生成代理，而不是针对类，该目标类型实现的接口都将被代理。原理是通过在运行期间创建一个接口的实现类来完成对目标对象的代理。

1. 定义一个实现接口 InvocationHandler 的类；
2. 通过构造函数，注入被代理类；
3. 实现 invoke（ Object proxy, Method method, Object[] args）方法；
4. 在主函数中获得被代理类的类加载器；
5. 使用 Proxy.newProxyInstance( ) 产生一个代理对象；
6. 通过代理对象调用各种方法。

**BeanFactory 和 ApplicationContext 的区别**

BeanFactory 才是 Spring 容器中的顶层接口。

ApplicationContext 是它的子接口。

BeanFactory 和 ApplicationContext 的区别：

创建对象的时间点不一样。

ApplicationContext：只要一读取配置文件，默认情况下就会创建对象。

BeanFactory：什么使用什么时候创建对象。

**@Component**

**作用：**

把资源让 spring 来管理。相当于在 xml 中配置一个 bean。

**@Controller：**一般用于表现层的注解。**@Service：**一般用于业务层的注解。**@Repository：**一般用于持久层的注解。**细节：如果注解中有且只有一个属性要赋值时，且名称是 value，value 在赋值是可以不写。**

**@Autowired**

**作用：**

自动按照类型注入。当使用注解注入属性时，set 方法可以省略。它只能注入其他 bean 类型。当有多个 类型匹配时，使用要注入的对象变量名称作为 bean 的 id，在 spring 容器查找，找到了也可以注入成功。找不到 就报错。

@Override

**public** Object invoke(Object proxy, Method method

Object[] args) **throws** Throwable {

Object rtValue = **null**;

**try** {

//开启事务

TransactionManager.*beginTransaction*();

//执行业务层方法

rtValue = method.invoke(accountService, args);

//提交事务

TransactionManager.*commit*();

}**catch**(Exception e) {

//回滚事务

TransactionManager.*rollback*();

e.printStackTrace();

}**finally** {

//释放资源

TransactionManager.*release*();

}

**return** rtValue;

}

});

**return** proxyAccountService;

}

}

@Component("txManager")

**2.3.2 配置步骤**

**2.3.2.1 第一步：配置事务管理器**

<!-- 配置一个事务管理器 -->

<bean id=*"transactionManager"*

class=*"org.springframework.jdbc.datasource.DataSourceTransactionManager"*>

<!-- 注入 DataSource -->

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

</bean>

**2.3.2.2 第二步：配置事务的通知引用事务管理器**

<!-- 事务的配置 -->

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

</tx:advice>

**采用注解的方式配置事务**：

**配置事务管理器并注入数据源**

<!-- 配置事务管理器 --> <bean id=*"transactionManager"* ass=*"org.springframework.jdbc.datasource.DataSourceTransactionManager"*> <property name=*"dataSource"* ref=*"dataSource"*></property></bean>

**第二步：在业务层使用@Transactional 注解**

**在配置文件中开启 spring 对注解事务的支持**

<!-- 开启 spring 对注解事务的支持 -->

<tx:annotation-driven transaction-manager=*"transactionManager"*/>

**Joinpoint(连接点):**

所谓连接点是指那些被拦截到的点。在 spring 中,这些点指的是方法,因为 spring 只支持方法类型的

连接点。

AOP日志：

利用springAop技术，把数据插入进入数据库。

@Aspect基于注解配置

@before(“execution(controller.\*.\*(..))”)拦截controller所有方法

创建前置通知doBefore(JoinPoint jp)

创建切面类处理日志，访问开始时间startTime，访问的类execution，访问的方法executionMethod;clazz = jp.getTarget().getClass(); //具体要访问的类  
String methodName = jp.getSignature().getName(); //获取访问的方法的名称  
Object[] args = jp.getArgs();//获取访问的方法的参数  
  
//获取具体执行的方法的Method对象  
if (args == null || args.length == 0) {  
 method = clazz.getMethod(methodName); //只能获取无参数的方法  
} else {  
 Class[] classArgs = new Class[args.length];  
 for (int i = 0; i < args.length; i++) {  
 classArgs[i] = args[i].getClass();  
 }  
 clazz.getMethod(methodName, classArgs);

在前置通知获取上述信息，

//后置通知  
@After("execution(\* com.itheima.ssm.controller.\*.\*(..))")  
public void doAfter(JoinPoint jp) throws Exception {

在后置通知获取访问时长，ip地址，url。

获取url:clazz.getAnnotation(RequestMapping.class);

类.getAnnotation(RequetMapping.class)

获取后是为classAnnocation,再获取其value，classAnnocation.value;

获取Ip

在web.xml中配置监听器，RequestContextListener，获取requset

String ip = request.getRemoteAddr();

//获取当前操作的用户  
SecurityContext context = SecurityContextHolder.getContext();//从上下文中获了当前登录的用户  
User user = (User) context.getAuthentication().getPrincipal();  
String username = user.getUsername();

**附：**

Linux:

1. 查看端口占用 netstat -ano netstat -nao|findstr 8090(端口号) 查询具体的端口号的信息，可以查看pid.
2. lsof -i :8888根据端口号查找进程
3. 结束进程命令： **kill 进程号**，如kill 2522 若不能停止，则执行**kill -9 2522**强制通知命令.
4. **ps -ef | grep tomcat查看进程**

**JAVAWEB:**

**JDBC:**

步骤：

1. 导入驱动jar包 mysql-connector-java-5.1.37-bin.jar

2. 注册驱动Class.forName("com.mysql.jdbc.Driver");

3. 获取数据库连接对象 conn = DriverManager.getConnection("jdbc:mysql:///db3", "root", "root");

4. 定义sql

\* 注意：sql的参数使用？作为占位符。 如：select \* from user where username = ? and password = ?;

5. 获取执行sql语句的对象 PreparedStatement Connection.prepareStatement(String sql)

6. 给？赋值：

\* 方法： setXxx(参数1,参数2)

\* 参数1：？的位置编号 从1 开始

\* 参数2：？的值

7. 执行sql，接受返回结果，不需要传递sql语句

8. 处理结果

9. 释放资源

**Servlet:**

\* 执行原理：

1. 当服务器接受到客户端浏览器的请求后，会解析请求URL路径，获取访问的Servlet的资源路径

2. 查找web.xml文件，是否有对应的<url-pattern>标签体内容。

3. 如果有，则在找到对应的<servlet-class>全类名

4. tomcat会将字节码文件加载进内存，并且创建其对象

5. 调用其方法

\* Servlet中的生命周期方法：

1. 被创建：执行init方法，只执行一次

\* Servlet什么时候被创建？

\* 默认情况下，第一次被访问时，Servlet被创建

\* 可以配置执行Servlet的创建时机。

\* 在<servlet>标签下配置

1. 第一次被访问时，创建

\* <load-on-startup>的值为负数

2. 在服务器启动时，创建

\* <load-on-startup>的值为0或正整数

\* Servlet的init方法，只执行一次，说明一个Servlet在内存中只存在一个对象，Servlet是单例的

\* 多个用户同时访问时，可能存在线程安全问题。

\* 解决：尽量不要在Servlet中定义成员变量。即使定义了成员变量，也不要对修改值

2. 提供服务：执行service方法，执行多次

\* 每次访问Servlet时，Service方法都会被调用一次。

3. 被销毁：执行destroy方法，只执行一次

\* Servlet被销毁时执行。服务器关闭时，Servlet被销毁

\* 只有服务器正常关闭时，才会执行destroy方法。

\* destroy方法在Servlet被销毁之前执行，一般用于释放资源

**HTTP:**

Hyper Text Transfer Protocol 超文本传输协议

\* 传输协议：定义了，客户端和服务器端通信时，发送数据的格式

\* 特点：

1. 基于TCP/IP的高级协议

2. 默认端口号:80

3. 基于请求/响应模型的:一次请求对应一次响应

4. 无状态的：每次请求之间相互独立，不能交互数据

\* 历史版本：

\* 1.0：每一次请求响应都会建立新的连接

\* 1.1：复用连接

\* 请求消息数据格式

1. 请求行

请求方式 请求url 请求协议/版本

GET /login.html HTTP/1.1

\* 请求方式：

\* HTTP协议有7中请求方式，常用的有2种

\* GET：

1. 请求参数在请求行中，在url后。

2. 请求的url长度有限制的

3. 不太安全

\* POST：

1. 请求参数在请求体中

2. 请求的url长度没有限制的

3. 相对安全

2. 请求头：客户端浏览器告诉服务器一些信息

请求头名称: 请求头值

\* 常见的请求头：

1. User-Agent：浏览器告诉服务器，我访问你使用的浏览器版本信息

\* 可以在服务器端获取该头的信息，解决浏览器的兼容性问题

2. Referer：http://localhost/login.html

\* 告诉服务器，我(当前请求)从哪里来？

\* 作用：

1. 防盗链：

2. 统计工作：

3. 请求空行

空行，就是用于分割POST请求的请求头，和请求体的。

4. 请求体(正文)：

\* 封装POST请求消息的请求参数的

**在获取参数前，设置request的编码request.setCharacterEncoding("utf-8");**

**@WebServlet("/loginServlet")**

**public class LoginServlet extends HttpServlet {**

**@Override**

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

**//1.设置编码**

**req.setCharacterEncoding("utf-8");**

**//2.获取请求参数**

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

**String password = req.getParameter("password");**

**//3.封装user对象**

**User loginUser = new User();**

**loginUser.setUsername(username);**

**loginUser.setPassword(password);**

**//4.调用UserDao的login方法**

**UserDao dao = new UserDao();**

**User user = dao.login(loginUser);**

**//5.判断user**

**if(user == null){**

**//登录失败**

**req.getRequestDispatcher("/failServlet").forward(req,resp);**

**}else{**

**//登录成功**

**//存储数据**

**req.setAttribute("user",user);**

**//转发**

**req.getRequestDispatcher("/successServlet").forward(req,resp);**

**}**

**}**

**@Override**

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

**this.doGet(req,resp);**

**}**

**}**

**7. 编写FailServlet和SuccessServlet类**

**@WebServlet("/successServlet")**

**public class SuccessServlet extends HttpServlet {**

**protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {**

**//获取request域中共享的user对象**

**User user = (User) request.getAttribute("user");**

**if(user != null){**

**//给页面写一句话**

**//设置编码**

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

**//输出**

**response.getWriter().write("登录成功！"+user.getUsername()+",欢迎您");**

**}**

**}**

**Response:**

**\* 重定向的特点:redirect**

**1. 地址栏发生变化**

**2. 重定向可以访问其他站点(服务器)的资源**

**3. 重定向是两次请求。不能使用request对象来共享数据**

**\* 转发的特点：forward**

**1. 转发地址栏路径不变**

**2. 转发只能访问当前服务器下的资源**

**3. 转发是一次请求，可以使用request对象来共享数据**

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

**Cookie和session：**

**\* session与Cookie的区别：**

**1. session存储数据在服务器端，Cookie在客户端**

**2. session没有数据大小限制，Cookie有**

**3. session数据安全，Cookie相对于不安全**

**JSP:**

**MVC：**

**1. M：Model，模型。JavaBean**

**\* 完成具体的业务操作，如：查询数据库，封装对象**

**2. V：View，视图。JSP**

**\* 展示数据**

**3. C：Controller，控制器。Servlet**

**\* 获取用户的输入**

**\* 调用模型**

**\* 将数据交给视图进行展示**

**Ajax&json:**

JQeury实现方式

1. $.ajax()

\* 语法：$.ajax({键值对});

//使用$.ajax()发送异步请求

$.ajax({

url:"ajaxServlet1111" , // 请求路径

type:"POST" , //请求方式

//data: "username=jack&age=23",//请求参数

data:{"username":"jack","age":23},

success:function (data) {

alert(data);

},//响应成功后的回调函数

error:function () {

alert("出错啦...")

},//表示如果请求响应出现错误，会执行的回调函数

dataType:"text"//设置接受到的响应数据的格式

});

2. $.get()：发送get请求

\* 语法：$.get(url, [data], [callback], [type])

\* 参数：

\* url：请求路径

\* data：请求参数

\* callback：回调函数

\* type：响应结果的类型

3. $.post()：发送post请求

\* 语法：$.post(url, [data], [callback], [type])

\* 参数：

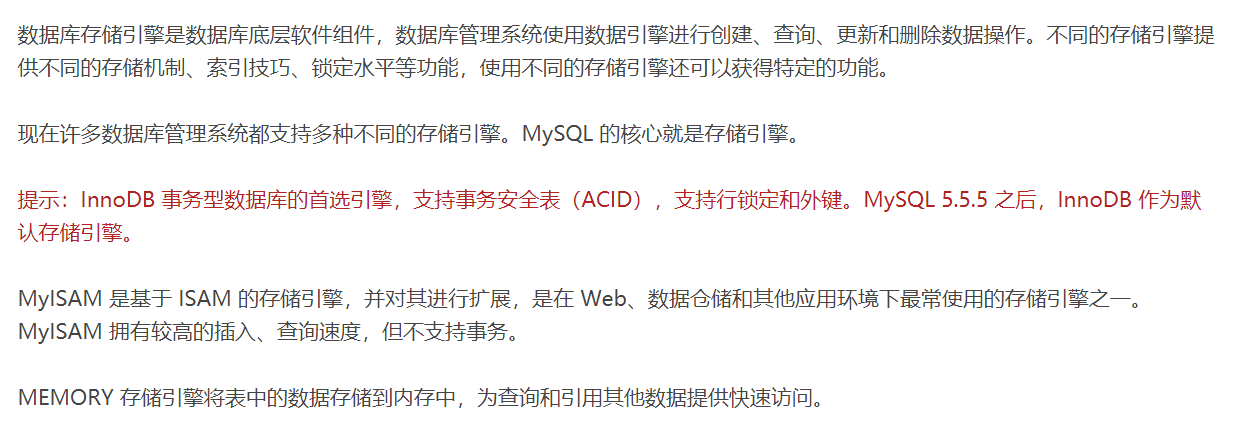
\* url：请求路径

\* data：请求参数

\* callback：回调函数

\* type：响应结果的类型

**数据库：**

****

### **一、索引是什么?**

索引是帮助MySQL高效获取数据的数据结构。

### **二、索引能干什么?**

索引非常关键，尤其是当表中的数据量越来越大时，索引对于性能的影响愈发重要。 索引能够轻易将查询性能提高好几个数量级，总的来说就是可以明显的提高查询效率。

### **三、索引的分类?**

1、从存储结构上来划分：BTree索引（B-Tree或B+Tree索引），Hash索引，full-index全文索引，R-Tree索引。这里所描述的是索引存储时保存的形式，

2、从应用层次来分：普通索引，唯一索引，复合索引

3、根据中数据的物理顺序与键值的逻辑（索引）顺序关系：聚集索引，非聚集索引。

## **删除索引**

1. alter table table\_name drop index index\_name
2. drop index index\_name on table\_name

## **在已经存在的表上创建索引**

1. create index index\_name  on table\_name (column\_name[length], ...) [asc|desc]
2. alter table table\_name add [unique|fulltext|spatial] [index|key] index\_name(column\_name[length, ...]) [asc|desc]

## **创建索引的原则**

* 更新频繁的列不应设置索引
* 数据量小的表不要使用索引（毕竟总共2页的文档，还要目录吗？）
* 重复数据多的字段不应设为索引（比如性别，只有男和女，一般来说：重复的数据超过百分之15就不该建索引）
* 首先应该考虑对where 和 order by 涉及的列上建立索引

Hash索引：

存储引擎将所有的索引列计算出哈希吗，将哈希吗存储在索引中，在哈希表中保存指向每个数据的指针，有个哈希函数，创建一个空间，获取hash函数的到的值，如果有相同的值将挂载在对应的空间下，类似数组+链表结构，然后当查找数据时，先计算hashcode,然后通过hash表找到对应的指针指向位置。

二叉树：任何一个节点的子节点的数量不超过2；

完全二叉树：所有叶子节点都在最后一层或倒数第二层，且最后一层的叶子节点在左边连续，倒数第二层的叶子节点在右边连续。

满二叉树：所有的叶子节点都在最后一层，而且节点的总数为：2^n-1 其中为树的高度；

堆排序原理：1.先变成大顶推（升序排序）对于任何一颗子树，所有的父节点权大于子节点的权。

1. 最后一个数字和第一个数字交换位子；
2. 把最后一个数字拿出来，放在数组最后，剩下的再转换成大顶推。。。。。递归。

如何调整为大顶推：

拿着最后一个非叶子节点与其子节点对比，如果小于则交换位置，再处理前面非叶子节点