Spring：

Spring是一个轻量级的控制反转(IoC)和面向切面(AOP)的容器框架。

控制反转(IoC)：Spring通过一种称作控制反转（IoC）的技术促进了松耦合。当应用了IoC，一个对象依赖的其它对象会通过被动的方式传递进来，而不是这个对象自己创建或者查找依赖对象。就如同打针，在需要注入的地方通过针管将液体注入进去。

面向切面(AOP)：Spring提供了[面向切面编程](https://link.zhihu.com/?target=https://baike.baidu.com/item/%E9%9D%A2%E5%90%91%E5%88%87%E9%9D%A2%E7%BC%96%E7%A8%8B" \t "https://zhuanlan.zhihu.com/p/_blank)的丰富支持，允许通过分离应用的业务逻辑与系统级服务（例如审计（auditing）和事务（transaction）管理）进行内聚性的开发。应用对象只实现它们应该做的——完成业务逻辑——仅此而已。它们并不负责（甚至是意识）其它的系统级关注点，例如日志或事务支持。AOP类似于将代码变成一个汉堡，不改变上下的面包片，中间根据需求加入食品。

IoC容器：

每个IOC容器都至少定义一个bean，在XML配置元数据，在<beans>元素中配置一个或多个<bean>元素。bean的定义包括:服务层对象,DAO层对象等。

bean的生命周期:先实例化javaBean,再初始化javaBean的实例，通过IOC注入，使用javaBean，最后销毁javaBean。

bean的创建:<bean>标签下设置id跟class属性，class为bean对象类型，bean由IOC容器管理，一般在xml文件中定义，配置其依赖关系。

实例化bean：1、用构造器来实例化

2、用静态工厂方法实例化

3、使用实例工厂方法实例化

bean的依赖：一个bean对另一个bean的依赖的做法就是设为另一个bean的属性，通过ref元素，被依赖的bean将在依赖的bean之前被适当的初始化。depends-on属性可以用于当前bean初始化之前强制一个或多个bean被初始化，若要依赖多个bean，可以在depends-on属性中指定多个bean名字，通过逗号、空格、分号等隔开。

bean的延迟初始化：通过bean元素中的lazy-init来进行控制。默认在IOC启动时完成所有bean的实例化，设置lazy-init的bean将在第一次使用时才会实例化。

创建IoC：IOC-BeanFactory,直到第一次使用geiBean()方法才会创建bean

IOC-ApplicationContext,自身被实例化时就会创建所有bean,一般用这个。

IOC依赖注入：1、构造函数注入（构造器参数类型匹配、参数索引）

2、属性注入（创建getter跟setter方法。XML中配置<bean>属性）

3、接口注入（不推荐）

4、注解方式（在java代码中使用@Autowired或@Resource注解进行装配，需要在XML配置文件中配置xmlns:context、xsi链接，以及在<beans>标签中加入<context:annotation-config />）。

@Autowired默认按类型装配可用于字段上或者set方法上，默认情况下要求依赖对象必须存在，若要允许空值，则可以设置required属性为false。若想按名称装配，可以配合@Qualifier注解一起使用： @AutoWired @Qualifier("bean名字")。

@Resource可以标注在字段者set方法上，默认按名称装配，通过name属性指定，如果没有指定name属性，默认取字段名称或者set方法的属性名作为bean名称寻找依赖对象，@Resource(name="名字")。

自动扫描机制：可以在类路径下扫描标注了下面注解的类，并且纳入spring容器管理，需要在配置文件中加入<context:component-scan base-package="包路径">

@Service用于标注业务层组件，@Controller用于标注控制层组件，@Repository用于标注数据访问组件，即DAO组件，@Component泛指组件，当组件不好归纳时可以用这个。

AOP：AOP由切面、连接点、切入点、通知组成，使用AOP，需要在配置文件中引入AOP命名空间。

注解方式的AOP编程：@Aspect定义一个切面

@Pointcut("execution()")定义一个切入点,括号内表达式描述哪些对象哪些方法执行。execution()括号内表达式：

（public \* \*(..)）任意公开方法执行

（\* set\*(..)）任意set方法执行

（\* com.xx.\*(..)）xx接口的任意方法执行

（\* com.\*.\*(..)）service包里任意方法执行

（\* com..\*.\*(..)）service包或者子包里任意方法执行

@Before("")定义前置通知

@AfterReturning(pointcut="",returning="")定义后置通知

@AfterThrowing(pointcut="",throwing="")定义例外通知

@After("")定义最终通知

@Around("")定义环绕通知

在配置文件中加入<aop:aspectj-autoproxy>

配置方式的AOP编程：采用静态配置文件实现

<aop:config>

定义一个切面并制定通知类<aop:aspect id="" ref="">

定义一个切入点，定义切入点名和匹配表达式<aop:pointcut id="" expressiong="execution()" />

定义切入点执行方法<aop:after method="" pointcut-ref="" />

</aop:aspect>

</aop:config>

多种切入点的定义：<aop:pointcut id="id" expression="execution()" />

<aop:before pointcut-ref="id" method="方法名" />定义前置通知

<aop:after-returning pointcut-ref="id" method="方法名" />定义后置通知

<aop:after-throwing pointcut-ref="id" method="方法名" />定义例外通知

<aop: after pointcut-ref="id" method="方法名" />定义最终通知

<aop:around pointcut-ref="id" method="方法名" />环绕通知。

**Mybatis：**

MyBatis 是一款优秀的持久层框架，它支持定制化 SQL、存储过程以及高级映射。MyBatis 避免了几乎所有的 JDBC 代码和手动设置参数以及获取结果集。MyBatis 可以使用简单的 XML 或注解来配置和映射原生信息，将接口和 Java 的 POJOs(Plain Ordinary Java Object,普通的 Java对象)映射成数据库中的记录。

**功能架构：**

API接口层：通过本地API来操纵数据库，接口层一接收到调用请求就会调用数据处理层来完成具体的数据处理。

数据处理层：根据调用请求完成一次数据库操作。

基础支撑层：为上层数据处理层提供最基础的支撑。包括连接管理、事务管理、配置加载和缓存处理。

**工作流程：**

1、加载配置并初始化（将SQL配置信息加载成为一个个MappedStatement对象）。

2、接收调用请求(为SQL的ID传入参数对象)。

3、处理操作请求(根据SQL的ID查找对应的MappedStatement对象，根据传入参数对象解析MappedStatement对象，得到最终要执行的SQL和执行传入参数，获取数据库连接，根据MappedStatement对象中的结果映射配置得到执行结果进行处理，得到处理结果，释放连接资源)。

4、返回结果。

**创建流程：**

SqlMapConfig.xml文件

<configuration>

<environments default="development">

<environment id="development">

<transactionManager type="JDBC" />

<dataSource type="POOLED">

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

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

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

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

</dataSource>

</environment>

</environments>

</configuration>

mapper.xml文件配置

<mapper name="">

<select id="" parameterType="接收参数的类型" resultType="返回值类型">

SQL语句

</select>

<insert>

insert语句默认返回插入记录的条数，如果需要得到记录的主键，可以通过配置方式完成

<selectKey keyProperty="id" order="" resultType="">

SQL语句

</selectKey>

</insert>

<delete>

<update>

</mapper>

动态SQL：

if语句

where

（可以在自己包含的内容前后加上某些后缀，属性是prefix和suffix）

set（主要是用在更新操作，可以在包含的语句前输出一个set，如果包含的语句以逗号结束的话会将该逗号省略）

foreach（主要用在构建in条件中，可以迭代一个集合，属性主要有item迭代时的别名,index指定一个名字，在迭代过程中每次迭代到的位置,collection,open该语句以书面开始,separator每次迭代之间以什么符号作为分隔符,close以什么结束）

choose(when,otherwise)判断内部when标签中test条件是否成立，若有一个成立则choose结束，所有条件都不成立时，执行otherwise的SQL

**核心配置文件：**

SqlMapConfig.xml全局配置文件

文档结构：

configuration配置

properties属性

settings设置

typeAliases类型命名

typeHandler类型处理器

objectFactory对象工厂

plugins插件

Environments环境

mappers映射器

MyBatis按照下面顺序加载属性：

在properties元素体内定义的属性首先被读取

然后会读取properties元素中resource或者url加载的属性，会覆盖已经读取的同名属性

最后读取parameterType传递的属性，会覆盖已经读取的同名属性

mapper.xml映射文件

需要配置的基本元素

1、cache 配置给定模式的缓存

2、cache-ref 从别的模式中引用一个缓存

3、resultMap 最复杂却强大的一个元素，描述如何从结果集中加载对象

4、sql 一个可以被其他语句复用的SQL块

5、insert 映射INSERT语句

6、update 映射UPDATE语句

7、delete 映射DELETE语句

8、select 映射SELECT语句

**Mybatis核心接口和类：**

SqlSessionFactoryBuilder：首先获取SqlSessionFactoryBuilder对象，可以根据XML配置文件或

Configuration类的实例对象构建该对象。

SqlSessionFactory：通过SqlSessionFactoryBuilder对象来获取SqlSessionFactory对象，再进

而获取SqlSession实例，该实例对象完全包含以数据库为背景的所有执行

SQL操作的方法，可以直接执行已映射的SQL语句。

**mapper代理开发：**

mapper代理方法：只需要有mapper接口和mapper.xml映射件,Mybatis 可以自动生成mapper接口实现类代理对象。程序员编写mapper接口需要遵循一些开发规范。

mapper代理开发规范：

1、在mapper.xml中namespace等于mapper接口地址。

2、mapper.java接口中的方法名和mapper.xml中statement的id一致。

3、mapper.java接口中的方法输入参数类型和mapper.xml中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statement的resultType指定的类型一致

mapper代理开发方法流程：

1、创建接口，方法名、参数、返回值与mapper.xml中映射配置保持一致

2、修改mapper.xml配置文件，namespace和接口地址一致

3、在SqlMapConfig.xml中加载mapper.xml

4、测试运行

逆向工程：

1、新建一个java工程。

2、引入使用的jar包，mybatis核心包、数据库驱动包。

3、添加配置文件generatorConfig.xml，设置数据库驱动、配置、包名、文件保存位置、表名等。

4、定义GeneratorSqlmap类，调用Mybatis自动创建接口，在main方法中执行自动创建。

可以自动根据数据库表名自动创建mapper接口类跟配置文件还有pojo，基本满足单表操作，但要涉及

多表联查等操作还是需要自己手动写mapper。

**Mybatis查询：**在进行select映射的时候，返回值类型可以用resultType直接表示返回类型，也可以用resultMap对外部ResultMap的引用，这两者不能同时存在。

MyBatis进行查询映射时，查询出来的每一个属性都是放在一个对应的Map里面，其中键是属性名，值是对应的值。

1、当提供的返回类型属性是resultType时，MyBatis会将Map里面的键值对赋值给resultType所指定的对象对应的属性。其实MyBatis的每一个查询映射的返回值类型都是ResultMap，只是当提供的返回值类型属性是resultType的时候，会自动的把对应的值赋值给resultType所指定对象的属性。

2、当提供的返回类型是resultMap时，因为Map不能很好表示领域模型，就需要自己再进一步转化为对应的对象，这在复杂查询中很有作用。

**延迟加载：**只有在真正使用到该对象时，才进行mapping操作，以减少数据库查询开销，从而提升系统性能。

但是也有缺点，按需加载会多次连接数据库，同事会增加数据库的压力，所以实际使用时，要衡量是否使用延迟加载。

resultMap可以实现高级映射（使用association、collection实现一对一及一对多映射），association、collection具备延迟加载功能

延迟加载配置：

Mybatis默认没有开启延迟加载，需要在SqlMapConfig.xml中setting配置。

lazyLoadingEnabled：true使用延迟加载，false禁用延迟加载，默认为false。

aggressiveLazyLoading：true启用时，当延迟加载开启时访问对象中一个懒对象属性时，

将会完全加载这个对象的所有懒对象属性。false，当延迟加载时，按需加载对象属性（即

访问对象中一个懒对象属性，不会加载对象中其他的懒对象属性）。默认为true。

**Mybatis查询缓存：**缓存技术是一种“以空间换时间”的设置理念，是利用内存空间资源来提高数据检索速度的有效手段之一。

一级缓存：一级缓存基于PerpetuanCache的HashMap本地缓存，其存储作用域为Session，当Session flush或 close之后，该Session中所有的Cache就将清空。

Mybaatis默认支持一级缓存，不需要去配置文件中配置。

二级缓存：二级缓存与一级缓存其机制相同，默认也是采用PerpetualCache，HashMap存储，不同在于其存储作用域为Mapper(Namespace)，并且可以自定义存储源，如Ehcache、Hazelcast等。

开启：mybatis的二级缓存是mapper范围级别，除了在SqlMapConfig.xml设置二级缓存总开关，还要在具体的mapper.xml中开启二级缓存。

在核心配置文件SqlMapConfig.xml中加入<setting name="cacheEnabled" value="true"/> ，

在XXXMapper.xml中加入<cache type="org.mybatis,caches.ehcache.EhcacheCache"/>

注意pojo类必须实现序列化接口。

禁用：在statement中设置useCache=false可以禁用当前select语句的二级缓存，即每次查询都会发出sql去查询，默认情况是true。

刷新缓存：在mapper的同一个namespace中，如果有其他insert、update、delete操作数据后需要刷新缓存，如果不执行刷新缓存会出现脏读。

设置statement配置中的flushCache=“true”属性，默认情况下为true即刷新缓存，如果改成false则不会刷新。

## **Spring MVC**

**Spring MVC组件：**

前端控制器DispatcherServlet（不需要程序员开发）用于接受请求，响应结果，相当于转发器，中央处理器。

处理器映射器HandlerMapping（不需要程序员开发）根据请求的url查找Handler。

处理器适配器HandlerAdapter 按照特定的规则去执行Handler。

处理器Handler（需要程序员开发）编写Handler时按照HandlerAdapter的要求去做，这样适配器才能正确执行Handler。

视图解析器View resolver（不需要程序员开发）进行视图解析，根据逻辑视图名解析成真正的视图。

视图View（需要程序员开发jsp）View是一个接口，实现类支持不同的View类型。

**开发步骤：**

1、导入jar包

2、在resource下创建springmvc.xml配置文件

3、在web,xml里配置前端控制器DispatcherServlet

<web-app>

<display-name>Archetype Created Web Application</display-name>

<servlet>

<servlet-name>springmvc</serlvet>

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

<init-param>

<param-name>ContextConfigLocation</param-name>

<param-value>classpath:springmvc.xml</param-value>

</init-param>

</servlet>

<servlet-mapping>

<servlet-name>springmvc</serlvet-name>

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

</servlet-mapping>

</web-app>

4、新建一个类继承Controller

5、springmvc.xml配置访问映射<bean name="/\*\*.action" class="控制器的类路径"/>

6、视图编写（jsp）

**配置文件开发：**

HandlerMapping处理器映射器，负责根据request请求找到对应的Handler处理器及Interceptor拦截器，将它们封装在HandlerExecutionChain对象中给前端控制器返回 **。**

HandlerAdapter处理器适配器，根据适配器接口对后端控制器进行包装后可对处理器进行执行，通过拓展处理器适配器可以执行多种类型的处理器。

**注解开发：**

RequestMappingHandlerMapping注解式处理器映射器，对标记了@RequestMapping的方法进行映射，根据RequestMapping定义的url匹配RequestMapping标记的方法，匹配成功返回HandlerMethod对象给前端控制器，HandlerMethod对象中封装url对应的方法Method。配置信息:<bean class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping" />

在spring容器中，根据请求的url与定义的bean的name进行匹配

<bean name="请求url" class="响应请求的方法路径" />

配置注解映射器和适配器

<mvc:annotation-driven></mvc:annotation-driven>

注解扫描

<context:component-scan base-package="控制器路径"></context:component-scan>

@Controller修饰处理

@RequestMapping("")修饰处理器的方法，参数使用请求url

@RequestParam(value="",required="",defaultValue="")绑定请求参数,value表示参数名字，required表示是否必须，默认为true，defaultValue表示默认值，如果请求中没有同名参数时的默认值。

*Spring MVC使用@RequestMapping注解为控制器指定可以处理哪些url请求，DispatcherServlet截获请求后，就通过控制器上的@RequestMapping提供的映射信息确定请求所有对应的处理方法。@RequestMapping除了可以使用请求url映射请求外，还可以使用请求方法、请求参数、请求头映射请求。@RequestMapping的value、method、params和headers分别表示请求url、请求方法、请求参数及请求头的映射条件，他们之间是与的关系。*

**映射请求@RequestMapping：**

标注在类定义处时，提供初步的映射信息，相对于web应用的根目录。

标注在方法处时，提供进一步的细分映射信息。

映射： url：@RequestMapping（“url”）

method：@RequestMapping（value=“/xxx”，method=｛RequestMethod.GET，RequestMethod.POST｝）

params：支持简单的表达式（param1表示请求必须包含名为param1的请求参数，!param1表示请求不能包含名为param1的请求参数，param1!=value表示请求包含名为param1的请求参数，但值不能为value1，{"param1=value1"，"param2"｝表示请求必须包含名为param1和param2两个请求参数，而且param1的值必须为value1）。

**controller方法的返回值：**

返回ModelAndView：

//controller方法中定义ModelAndView对象并返回，对象中可以添加model数据，指定view。

ModelAndView modelAndView = new ModelAndView();

//使用addObject方法，相当于request.setAttribute方法。

modelAndView.addObject("args",args);

//指定视图返回

modelAndView.setViewName("xx/xxx");

return modelAndView;

返回字符串

有三种含义：

1、表示返回逻辑视图名

真正的视图（jsp路径）=前缀+逻辑视图名+后缀

2、redirect重定向

浏览器地址栏中的url会变化，修改提交的request数据无法传到重定向后的地址，因为重定向后重新进行request

return "redirect:xx.action";

3、forward页面转发

进行页面转发，浏览器地址栏的url不变，request可以共享

return "forward:xx.action";

返回void

在controller方法形参上可以定义request和response，指定响应的结果，使用request转向页面

request.getRequestDispatcher("页面路径").forward(request,response)

使用response页面重定向

response.sendRedirect("url");

**绑定参数：**

简单类型绑定：如果不使用@RequestPatam，要求request传入参数名称和controller方法的形参名称一致即可绑定。如果使用@RequestParam，不用限制request传入参数名称和controller方法的形参名称一致。

POJO类型绑定：将pojo对象中的属性名与传递进来的参数名对应。

自定义参数绑定：自定义Converter，实现Converter接口向处理器适配器中注入自定义的参数绑定组件。

集合类型绑定：支持字符串数组、List、Map。