## **Spring**

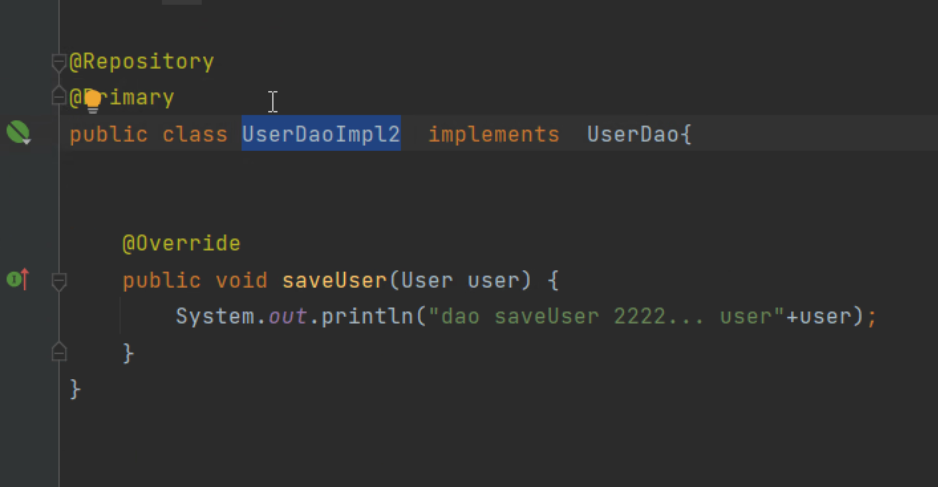
##### **1、IOC(inversion of controller)控制反转**

指的是将对象的创建权交给 Spring 去创建。使用 Spring 之前，对象的创建都是由我们自己在代码中new创建。而使用 Spring 之后，对象的创建都是给了 Spring 框架。spring在初始化的时候，容器里的对象就已经被创建出来了，默认是单例的。实现原理是反射。

ioc容器：ioc容器负责创建、配置和管理对象及其之间的依赖关系，存放控制反转生成的对象。

## **实现ioc的多种方式：**

set注入、注解注入、构造注入、类注入、xml配置







###### **单例和多例**

单例：就是所有的请求都用一个对象来处理，比如我们常用的service和dao层的对象通常都是单例的。

多例：则指每个请求用一个新的对象来处理。

<!--scope="singleton"  默认单例，只创建一个对象   懒/饿汉模式-->  
<!--scope="prototype"  多例，创建多个对象    饿汉模式-->  
<!--lazy-init="false"  
    懒加载：对象什么时候创建。  
       1.1 默认创建容器的时候创建(非懒加载)  
       1.2 获取对象的时候创建  
     只对单例（scope为懒汉模式）时起作用,多例时就是在获取对象时创建-->

###### **单例模式和多例模式说明：**

1.单例模式和多例模式属于对象模式。

2.单例模式的对象在整个系统中只有一份，多例模式可以有多个实例。

3.它们都不对外提供构造方法，即构造方法都为私有。

##### **2、AOP(Aspect Oriented Programming)面向切面编程**

简单的说就是可配置的动态代理，用来实现数据库事务、插入日志，等各种增强的功能。

###### **切点：**

所有连接点的集合，在代码中execution表达式就是切点 pointcut

###### **通知：**

五种通知：前置，后置，环绕，正常返回，异常返回

执行顺序：

前置通知 >>>>正常返回>>>>后置通知

前置通知 >>>>异常返回>>>>后置通知

###### **切面：**

切点+通知就是切面。在代码中，加上了aspect标签的类就是切面

###### **xml方式实现aop**

<!--首先配置一个通知类，里面包含五种通知-->  
<bean id="Adivce" class="com.demo02.Advice"/>  
    <aop:config>  
        <aop:aspect ref="Adivce">  
            <!--配置切点-->  
            <aop:pointcut id="a1" expression="execution(void com.demo02.EmpServiceImpl.getAllEmp())"/>  
            <!--配置五种通知-->  
            <aop:before method="before" pointcut-ref="a1"/>  
            <aop:after method="after" pointcut-ref="a1"/>  
            <aop:around method="around" pointcut-ref="a1"/>  
            <aop:after-returning method="returning" pointcut-ref="a1"/>  
            <aop:after-throwing method="throwing" pointcut-ref="a1"/>  
        </aop:aspect>  
    </aop:config>  
​  
​  
/\*  
这段代码是一个使用Spring AOP（面向切面编程）配置的示例。  
​  
首先，通过 <bean> 标签将一个名为 "Adivce" 的 bean 定义为 com.demo02.Advice 类的实例。  
​  
然后，在 <aop:config> 标签中配置了一个切面（aspect），使用 ref 属性引用了上述的 "Adivce" bean。  
​  
接下来，通过 <aop:pointcut> 标签配置了一个切点（pointcut），id 设置为 "a1"，expression 属性指定了切点的表达式，即指定了要切入的目标方法。这里的表达式是 execution(void com.demo02.EmpServiceImpl.getAllEmp())，表示切入 com.demo02.EmpServiceImpl 类的 getAllEmp() 方法。  
​  
然后，使用 <aop:before>、<aop:after>、<aop:around>、<aop:after-returning> 和 <aop:after-throwing> 标签配置了五种通知（advice）：  
​  
<aop:before> 标签配置了一个前置通知（before advice），method 属性指定了 "before" 方法，pointcut-ref 属性引用了上述的切点 "a1"。  
<aop:after> 标签配置了一个后置通知（after advice），method 属性指定了 "after" 方法，pointcut-ref 属性引用了上述的切点 "a1"。  
<aop:around> 标签配置了一个环绕通知（around advice），method 属性指定了 "around" 方法，pointcut-ref 属性引用了上述的切点 "a1"。  
<aop:after-returning> 标签配置了一个返回通知（after-returning advice），method 属性指定了 "returning" 方法，pointcut-ref 属性引用了上述的切点 "a1"。  
<aop:after-throwing> 标签配置了一个异常通知（after-throwing advice），method 属性指定了 "throwing" 方法，pointcut-ref 属性引用了上述的切点 "a1"。  
这样，通过配置切面和通知，就可以在目标方法执行前后或发生异常时插入相应的逻辑处理。  
\*/

###### **注解方式实现aop**

在spring容器里面

   <!--开启注解驱动-->  
    <context:annotation-config/>  
    <!--指定要扫描的包-->  
    <context:component-scan base-package="com.demo03"/>  
    <!--开启aop注解支持-->  
    <aop:aspectj-autoproxy/>  
    //在通知类上打注解 @Aspect  
    //定义切点execution（返回值类型 包名.类名.方法名（..）） \*号代表任意  
    @Pointcut("execution(\* com.demo03.\*.\*(..))")  
    public void aa(){  
   }

###### **aop的设计模式**

切面（Aspect）：

切面是一个模块化的单元，用于封装横切关注点的行为。

切面通常包含了一组切点（Pointcut）和相应的通知（Advice）。

切面可以在程序的不同位置插入横切关注点的逻辑。

切点（Pointcut）：

切点用于定义在程序中哪些位置插入横切关注点的逻辑。

切点可以通过正则表达式、注解、方法签名等方式进行定义。

切点可以选择在程序的类、方法、字段等级别进行匹配。

通知（Advice）：

通知定义了在切点处执行的横切关注点的行为。

通知可以在切点前、切点后、切点前后等不同的时间点执行。

常见的通知类型有前置通知（Before）、后置通知（After）、返回通知（AfterReturning）、异常通知（AfterThrowing）和环绕通知（Around）。

织入（Weaving）：

织入是将切面的行为应用到目标对象的过程。

织入可以在编译时、加载时或运行时进行。

织入可以通过源代码修改、字节码修改或代理对象等方式实现。

##### **3、DI(Dependency Injection)依赖注入**

类 A 依赖于类 B，所以意味着类 B 将通过 IoC 被注入到类 A 中。例如，service层依赖于dao层。通过使用 setter 方法和构造方法 constructor进行注入。

/\*  
在Spring的依赖注入（Dependency Injection，DI）中，有两种主要的方式可以进行注入，即使用setter方法和构造方法。  
​  
Setter方法注入：  
Setter方法注入是通过在被注入的类中定义对应的setter方法来实现的。Spring容器在创建Bean实例时，会调用相应的setter方法来设置依赖的属性。  
例如，假设有一个Person类，具有name和age属性，可以通过以下方式进行setter方法注入：  
​  
​  
public class Person {  
    private String name;  
    private int age;  
​  
    // setter方法  
    public void setName(String name) {  
        this.name = name;  
    }  
​  
    public void setAge(int age) {  
        this.age = age;  
    }  
}  
在Spring配置文件中，通过<bean>标签定义Person的Bean，并使用<property>标签设置属性值：  
​  
xml  
Copy code  
<bean id="personBean" class="com.example.Person">  
    <property name="name" value="John" />  
    <property name="age" value="25" />  
</bean>  
在上述配置中，Spring容器会创建Person的实例，并通过调用相应的setter方法将属性值设置为"name"和"age"。  
​  
构造方法注入：  
构造方法注入是通过在被注入的类中定义一个带有参数的构造方法来实现的。Spring容器在创建Bean实例时，会调用相应的构造方法，并提供参数值。  
继续以Person类为例，可以通过以下方式进行构造方法注入：  
​  
​  
public class Person {  
    private String name;  
    private int age;  
​  
    // 构造方法  
    public Person(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
}  
在Spring配置文件中，通过<bean>标签定义Person的Bean，并使用<constructor-arg>标签设置构造方法的参数值：  
​  
​  
<bean id="personBean" class="com.example.Person">  
    <constructor-arg name="name" value="John" />  
    <constructor-arg name="age" value="25" />  
</bean>  
在上述配置中，Spring容器会创建Person的实例，并通过调用带有参数的构造方法将"name"和"age"的值传递给构造方法。  
​  
无论是setter方法注入还是构造方法注入，Spring容器会负责解析配置文件并自动进行依赖注入。这样，您可以将依赖关系的管理交给Spring容器，使代码更加灵活、可维护和可测试。  
\*/

###### **@autowired(required =true)**

###### **@Qualifier(value = "") //名称注入**

先按照类型再按照名称。required属性：是否必须依赖注入。默认为true，必须注入bean对象，否直接报错。

通过Qualifier去指定名称。

###### **@resource(name="")**

先按照名称再按照类型。

#### **4.如何保证数据一致性，怎么进行回滚.**

声明式事务，首先在xml中配置事务的配置信息，tx：annocation-deriven

然后在需要支持事务的类上或是方法上写@Transactional标签。标签中默认事务的传播行为是required。即如果该方法有事务，就使用该方法的事务，如果该方法没有事务，就自己创建一个事务。传播行为有7中，我们使用的的最多的就是默认的传播事务。

spring是如何管理事务的？在执行支持事务的方法时，如果方法没有抛出异常，正常执行就会执行commit，如果方法抛出了异常，就会执行回滚。我们可以在标签中，设置robackfor的值指定回滚时机。rollbackfor：用于指定触发事务回滚的异常类型，用的最多是RunTimeException，还有theowable。

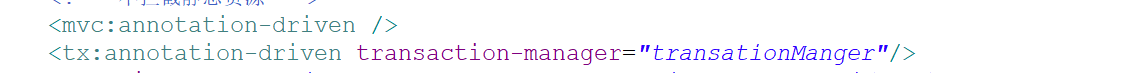
#### **5、Spring 事务管理**

事务管理分为声明式事务管理（仅仅使用注释或 XML 配置来管理事务）和编程式事务管理

###### **如何使用声明式事务：**

spring.xml里面配置

@Transactional注解有一个可选的属性"propagation"，用于指定事务的传播行为 rollback



首先在xml中配置事务的配置信息，tx：annocation-deriven 然后在需要支持事务的类上或是方法上写上@Transactional标签。

标签中默认事务的传播行为是required。即如果该方法有事务，就使用该方法的事务，如果该方法没有事务，就自己创建一个事务。传播行为有7中，我们使用的的最多的就是默认的传播事务。

/\*  
​  
Spring 框架中的事务管理器支持七种传播行为（Propagation Behavior），用于控制事务在方法调用链中的传播方式和行为。这些传播行为可以通过 @Transactional 注解或 XML 配置来设置。以下是七种传播行为：  
​  
REQUIRED（默认）：  
如果当前存在事务，则加入到当前事务中；如果当前没有事务，则创建一个新的事务。该传播行为保证了方法执行时必须处于一个事务之中。  
​  
SUPPORTS：  
如果当前存在事务，则加入到当前事务中；如果当前没有事务，则以非事务方式执行。该传播行为表示方法可以在事务环境中执行，也可以在非事务环境中执行。  
​  
MANDATORY：  
如果当前存在事务，则加入到当前事务中；如果当前没有事务，则抛出异常。该传播行为要求方法在一个已存在的事务中执行，如果当前没有事务则会抛出异常。  
​  
REQUIRES\_NEW：  
总是创建一个新的事务，如果当前存在事务，则挂起当前事务。该传播行为总是创建一个新的独立事务，并且如果当前存在事务，会将当前事务挂起。  
​  
NOT\_SUPPORTED：  
以非事务方式执行操作，如果当前存在事务，则将其挂起。该传播行为表示方法不应该运行在事务中，如果当前存在事务则会将其挂起。  
​  
NEVER：  
以非事务方式执行操作，如果当前存在事务，则抛出异常。该传播行为表示方法不应该运行在事务中，如果当前存在事务则会抛出异常。  
​  
NESTED：  
如果当前存在事务，则在嵌套事务中执行；如果当前没有事务，则创建一个新的事务。嵌套事务是外部事务的一部分，具有独立的保存点，并可以进行回滚。如果外部事务回滚，则嵌套事务也会回滚；如果嵌套事务回滚，对外部事务没有影响。  
​  
通过设置适当的传播行为，可以控制事务的范围和行为，以满足应用程序的需求。这样可以确保数据的一致性和完整性，并提供良好的事务管理机制。  
\*/

###### **spring是如何管理事务的？**

在执行支持事务的方法时，如果方法没有抛出异常，正常执行就会执行commit，如果方法抛出了异常，就会执行回滚。我们可以在标签中，设置robackfor的值指定回滚时机。rollbackfor：用于指定触发事务回滚的异常类型，默认是runtimeexception，还有throwable。传.class类型

###### **ThreadLock：线程**

ThreadLock（线程锁）是一种同步机制，用于控制多个线程对共享资源的访问，以保证线程安全性。线程锁可以防止多个线程同时访问临界区（共享资源），从而避免竞态条件和数据不一致的问题

###### **Spring事务实现原理：**

spring事务使用aop代理实现的，在代码运行时生成一个代理对象。然后根据transactional中的配置，由拦截器transactionalinterceptor拦截。当目标对象被拦截后，会在目标方法之前创建并加入事务，接着执行目标方法的逻辑，最后根据执行情况执行回滚或提交。

#### **数据库事务特性 (ACID)**

原子性：事务应该当作一个单独单元的操作，这意味着整个序列操作要么是成功，要么是失败的。

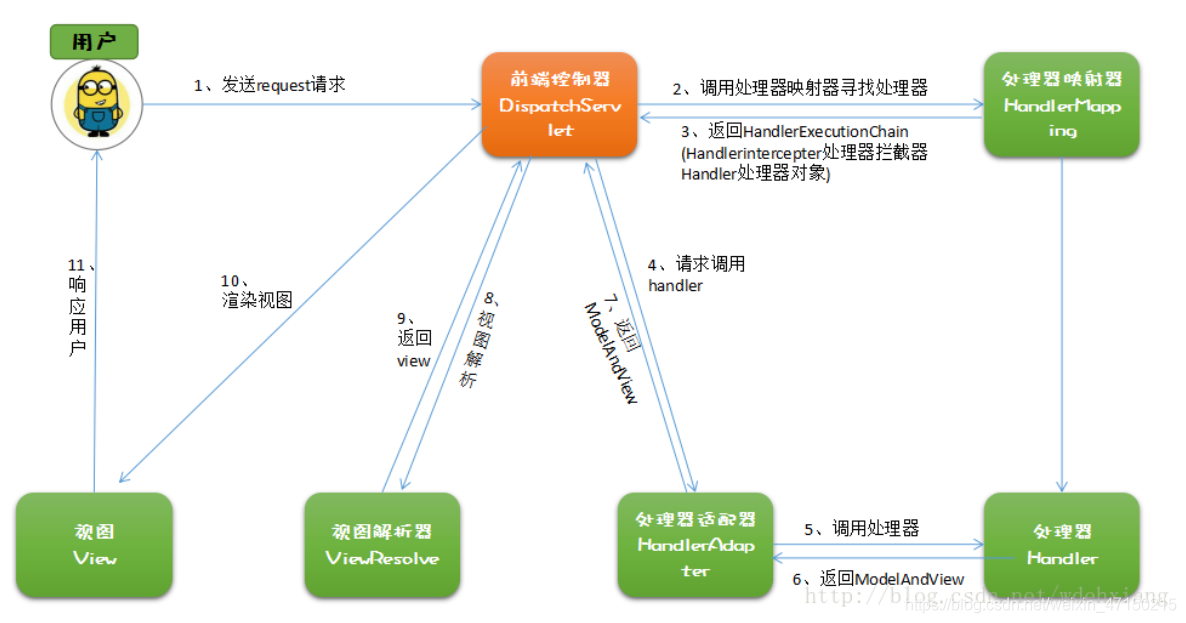
一致性：这表示数据库的引用完整性的一致性，表中唯一的主键等。 一致性是指事务必须使数据库从一个一致性状态变换到另一个一致性状态，也就是说一个事务执行之前和执行之后都必须处于一致性状态。

隔离性：可能同时处理很多有相同的数据集的事务，每个事务应该与其他事务隔离，以防止数据损坏。 隔离性是当多个用户并发访问数据库时，比如操作同一张表时，数据库为每一个用户开启的事务，不能被其他事务的操作所干扰，多个并发事务之间要相互隔离。

持久性：一个事务一旦完成全部操作后，这个事务的结果必须是永久性的，不能因系统故障而从数据库中删除。

## **Springmvc**

##### **1、springmvc执行顺序**



用户发送请求到前端控制器（DispatcherServlet）。

前端控制器请求处理器映射器（HandlerMapping）去查找处理器（Handler）。

找到以后处理器映射器（HandlerMappering）向前端控制器返回执行链（HandlerExecutionChain）。

前端控制器（DispatcherServlet）调用处理器适配器（HandlerAdapter）去执行处理器（Handler）。

处理器适配器去执行Handler。

处理器执行完给处理器适配器返回ModelAndView。

处理器适配器向前端控制器返回ModelAndView。

前端控制器请求视图解析器（ViewResolver）去进行视图解析。

视图解析器向前端控制器返回View。

前端控制器对视图进行渲染。

前端控制器向用户响应结果。

##### **2、Springmvc常用注解：**

1.@Controller 控制器

2.@RestController（@RestController = @Controller + @ResponseBody）将返回的数据结构转换为 JSON 格式

3.@RequestMapping 配置处理请求的路径

4.@RequestBody 将前端传过来的json格式的参数映射到对象里

5.@ResponseBody 返回json格式的数据

6.@PathVariable 要用来获取 Restful风格请求的URL 参数,跟@RequestMapping（"show/{id}/{name}"）结合起来用

7.@RequestParam 前端传过来的参数名字为name，后端想用nm接收这个参数，需要在nm前加上这个注解

8.@CookieValue 获取cookie值

9.@RequestHeader 获取请求头

10.@PostMapping 只接收post请求

11.@GetMapping 只接收get请求

@CrossOrigin 跨域的请求

@controllerAdvies

补：

@Component：标注一个普通的Spring Bean类。

@Controller：标注一个控制器组件类。

@Service：标注一个业务层组件类。

@Repository：标注一个dao持久层组件类。

## **拦截器：基于AOP实现**

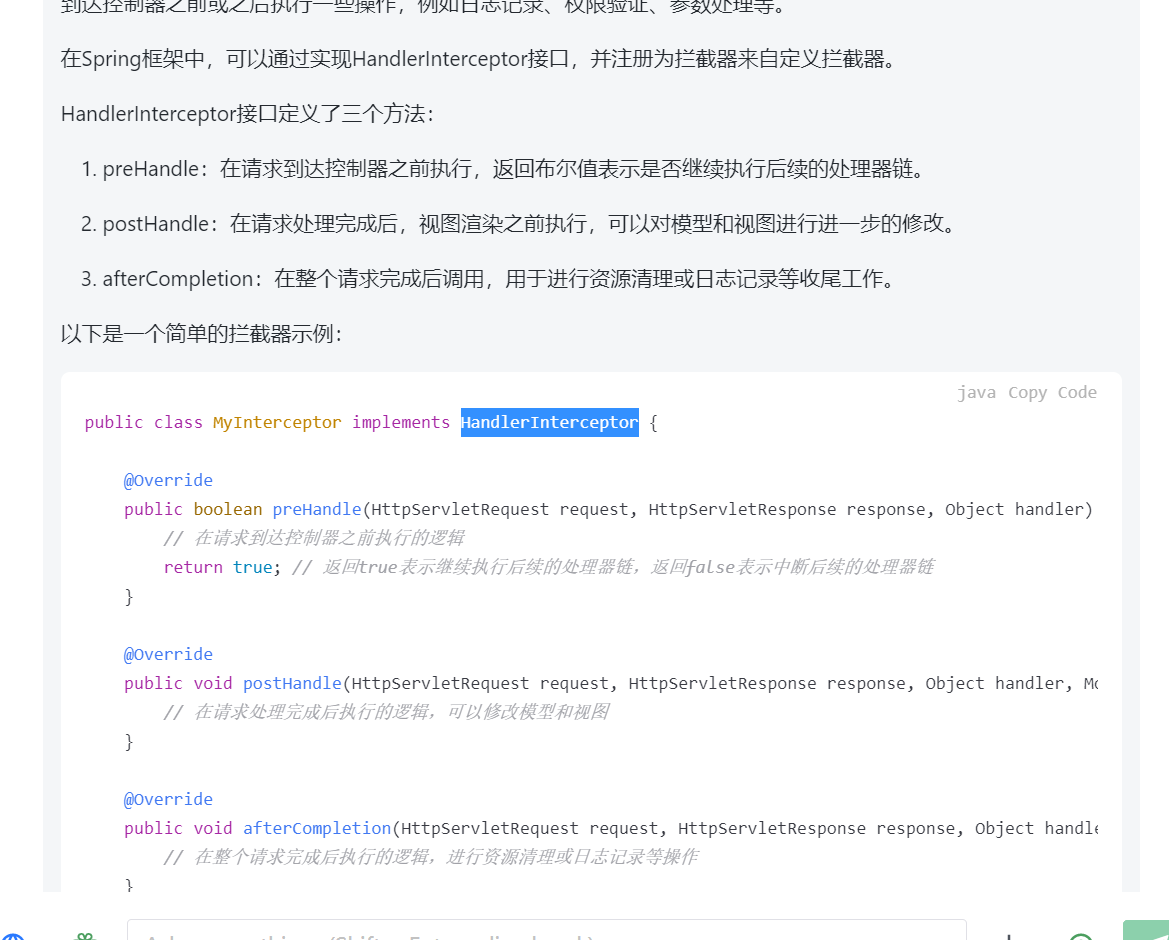
我们可以看到 HandlerInterceptor接口有三个方法，分别是preHandle、postHandle、afterCompletion,关于这三个方法

preHandle 方法：该方法在执行器方法之前执行。返回值为Boolean类型，如果返回false,表示拦截器不再向下执行；如果返回true,表示放行，程序向下执行(如果后边没有其他Interceptor,就会执行Controller方法)。所以，此方法可对方法进行判断，决定程序是否继续执行，或者进行一些初始化操作及对请求进行预处理。

postHandle方法：该方法在执行控制器方法调用之后，且在返回ModelAndView之前执行。由于该方法会在DispatcherServlet进行返回视图渲染之前被调用，所以此方法多被用于处理返回的视图，可通过此方法多被用于处理返回的视图，可通过此方法对请求域中的模型和视图做进一步的修改。

afterCompletion方法：该方法在执行完控制器之后执行。由于是在Controller方法执行完毕之后执行该方法，所以该方法适合进行一些资源清理、记录日志信息等处理操作。

使用单个拦截器实现登录验证



## **Mybatis**

##### **常用标签**

resultMap：处理查询的结果集和实体类属性之间的映射关系。

Collection：是处理一对多的映射

association：是处理多对一的映射

<!--数据库字段名和实体类属性的映射   n:1-->  
    <resultMap id="empMap" type="com.bean.Emp">  <!--type是返回值的类型-->  
        <id column="empno" property="yuangongbianhao"/>   
        <!-- id 主键 result 普通字段 column字段  property属性-->  
        <result column="ename" property="xingming"/>  
        <result column="job" property="zhiwei"/>  
        <result column="mgr" property="mgr"/>  
        <result column="hiredate" property="hiredate"/>  
        <result column="sal" property="sal"/>  
        <result column="comm" property="comm"/>  
        <result column="deptno" property="deptno"/>  
​  
        <!--n:1 的关系用association  emp对象中的dept属性设置,在emp实体类中加属性dept对象   private Dept dept;-->  
        <association property="dept" javaType="com.bean.Dept">  
            <id column="deptno" property="deptno"/>  
            <result column="dname" property="dname"/>  
            <result column="loc" property="difang"/>  
        </association>  
    </resultMap>      
​

当使用MyBatis进行多对一查询时，我们可以结合使用<resultMap>和SQL查询来实现。以下是一个示例，展示了如何在MyBatis中执行包含映射的多对一查询：  
​  
首先，假设我们有两个表，一个是orders表，另一个是customers表。每个订单对应一个客户，但一个客户可以有多个订单。我们想要检索出所有订单，并包含每个订单所对应的客户信息。  
​  
首先，我们需要定义一个包含映射的<resultMap>，如下所示：  
​  
<resultMap id="orderMap" type="com.bean.Order">  
  <id column="order\_id" property="orderId"/>  
  <result column="order\_date" property="orderDate"/>  
  <result column="order\_amount" property="orderAmount"/>  
  <association property="customer" javaType="com.bean.Customer">  
    <id column="customer\_id" property="customerId"/>  
    <result column="customer\_name" property="customerName"/>  
    <!-- 其他客户属性的映射 -->  
  </association>  
</resultMap>  
在上面的示例中，我们定义了一个名为orderMap的<resultMap>，类型为com.bean.Order。其中，我们通过<id>和<result>标签将列和属性进行映射。  
​  
接下来，在SQL查询中使用这个<resultMap>，如下所示：  
​  
<select id="getOrdersWithCustomers" resultMap="orderMap">  
  SELECT o.order\_id, o.order\_date, o.order\_amount, c.customer\_id, c.customer\_name  
  FROM orders o  
  JOIN customers c ON o.customer\_id = c.customer\_id  
</select>  
在上面的示例中，我们使用<select>标签定义了一个名为getOrdersWithCustomers的查询语句，并通过resultMap属性指定了使用前面定义的orderMap映射。  
​  
现在，我们可以在Java代码中调用这个查询，并获取结果，如下所示：  
​  
java  
Copy code  
SqlSession sqlSession = sqlSessionFactory.openSession();  
try {  
  List<Order> orders = sqlSession.selectList("getOrdersWithCustomers");  
  for (Order order : orders) {  
    System.out.println("Order ID: " + order.getOrderId());  
    System.out.println("Order Date: " + order.getOrderDate());  
    System.out.println("Order Amount: " + order.getOrderAmount());  
    Customer customer = order.getCustomer();  
    System.out.println("Customer ID: " + customer.getCustomerId());  
    System.out.println("Customer Name: " + customer.getCustomerName());  
    // 其他客户属性的访问  
    System.out.println("---------------------------");  
 }  
} finally {  
  sqlSession.close();  
}  
在上面的示例中，我们使用SqlSession对象执行查询语句，并将结果映射为包含多对一关系的对象。然后，我们可以通过对象的方法访问订单和客户的属性。

<!--数据库字段名和实体类属性的映射   1:n-->   
    <resultMap id="deptMap" type="com.bean.Dept">  
        <id column="deptno" property="deptno"/>  
        <result column="dname" property="dname"/>  
        <result column="loc" property="difang"/>  
​  
        <!--1:n的关系用collection   dept对象中的emps属性设置,在dept实体类中加emp对象的集合   private List<Emp> emps;-->  
        <collection property="emps" ofType="emp">  
            <id column="empno" property="yuangongbianhao"/>  
            <result column="ename" property="xingming"/>  
            <result column="job" property="zhiwei"/>  
            <result column="mgr" property="mgr"/>  
            <result column="hiredate" property="hiredate"/>  
            <result column="sal" property="sal"/>  
            <result column="comm" property="comm"/>  
            <result column="deptno" property="deptno"/>  
        </collection>  
    </resultMap>

where、set、if

select \* from emp  <where>  
                        <if test="empno !=0" >  
                            and empno =#{empno}  
                        </if>  
                    </where>  
update emp <set>  
               <if test="ename != null" >  
                    ename =#{ename},  
               </if>  
           </set>  
<!--if标签判断条件是否满足，满足就拼接  
    where 标签作用：把第一个and替换成where  
    set标签作用：把最后一个",",替换成set-->

trim

<trim prefix="where" prefixOverrides="and"></trim>   <!--相当于where标签，prefix前缀，prefixOverrides前缀要替换的-->  
<trim prefix="set" suffixOverrides="," ></trim>       <!--相当于set标签，prefix前缀，suffixOverrides后缀要替换的-->

foreach标签，代替sql中的in。

<!--根据员工编号，查询多个emp对象-->  
    <select id="getSomeEmp" resultMap="empMap">  
        select  \* from emp where empno in  
        <foreach item="item" index="index" collection="array"  open="(" separator="," close=")">  
            #{item}  
        </foreach>  
<!-- item：代表遍历集合的每个元素，生成的变量名  
     collection：属性的值有三个分别是list、array、map三种，分别对应的参数类型为：List、数组、map集合，我在上面传的参数为数组，所以值为array  
     collection：代表要遍历的集合  
     open：代表语句的开始部分  
     sperator：代表分隔符  
     close：代表结束部分-->  
    </select>

##### **mybatis的优缺点**

MyBatis的优点包括：

灵活性：它允许开发人员使用自定义SQL语句，而非强制要求开发人员按照ORM框架的命名规则编写代码。

易于学习：与其他ORM框架相比，MyBatis API的学习曲线较平滑，因为它使用简单直观的映射文件和注解来配置对象关系。

易于维护：MyBatis代码基于XML、注解或混合方式编写，易于阅读和理解，减少代码冗余和错误。

高度可定制性：MyBatis提供了很多插件，比如分页插件、缓存插件等，可以满足不同应用场景定制化需求。

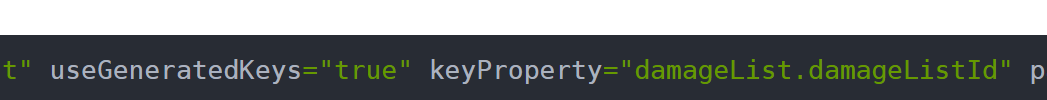
MyBatis的缺点包括：

大量手动SQL编写：虽然这也是灵活性之一，但是当处理简单场景时，需要开发人员不停编写SQL，可能会增加开发成本和Debug难度。

配置规模大：随着业务的扩展，MyBatis 的映射文件或注解数量会逐渐增加，这些配置过多会使代码维护变得复杂。

对象映射难以跟上Java语言的快速发展，某些功能上可能比不上JPA等ORM框架。

#### **自增主键问题**



## **.数据库三范式\*\*\*\***

第一范式（First Normal Form，1NF）：

数据表中的每个字段必须是原子的，即不能再细分成更小的字段。

每个字段的值都应该是一个单一的数据项，不允许包含多个值。

第二范式（Second Normal Form，2NF）：

数据表必须满足第一范式

非主键列必须完全依赖于主键，而不能部分依赖于主键。

如果数据表中存在部分依赖的列，应该将其分离成独立的表，并与原表建立关联关系。

第三范式（Third Normal Form，3NF）：

数据表必须满足第二范式

非主键列之间不能存在传递依赖关系。即，如果A列依赖于B列，B列又依赖于C列，那么A列应该直接依赖于C列，而不是通过B列间接依赖于C列。

## **SSM整合**

##### **两种容器的整合（父子容器）**

###### **spring.xml配置**

<!--1.扫描service，dao层 component-scan-->  
<!--2.引入数据库配置文件-->  
<!--3.配置数据库连接池和参数 dataSource-->  
<!--4.配置事务管理器transactionManager-->  
<!--5.识别@Transactional标签  annotation-driven-->  
<!--6.spring整合mybatis-->  
    <!--配置SqlSessionFactoryBean-->  
    <!--配置MapperScannerConfigure-->

###### **springmvc.xml配置**

<!--1.包扫描业务代码controller  component-scan-->  
<!--2.找到视图解析器的后拼接成：/WEB-INF/aa.jsp -->  
<!--3.配置拦截器  interceptors-->  
<!--4.文件上传  multipartResolver-->

###### **web.xml配置**

<!--整合spring-->  
    <context-param>  
        <param-name>contextConfigLocation</param-name>  
        <param-value>classpath:spring.xml</param-value>  
    </context-param>  
    <listener>  
        <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>  
    </listener>  
​  
<!--整合spingmvc-->  
    <servlet>  
        <servlet-name>DispatcherServlet</servlet-name>  
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>  
        <init-param>  
            <param-name>contextConfigLocation</param-name>  
            <param-value>classpath:springmvc.xml</param-value>  
        </init-param>  
        <load-on-startup>1</load-on-startup>  
            <!--默认容器启动时不加载且不初始化，那么当浏览器第一次访问servlet时，才会初始化  
                当设置值大于等于1时，容器创建时初始化-->  
    </servlet>  
    <servlet-mapping>  
        <servlet-name>DispatcherServlet</servlet-name>  
        <url-pattern>/</url-pattern>  
    </servlet-mapping>  
​

##### **一种容器的整合**

###### **springmvc.xml配置**

把两种容器整合时的spring.xml配置文件和springmvc.xml配置文件，放在一块

###### **web.xml配置**

只需要配置DispatcherServlet

# 数据库分页，分页是使用什么插件写的

数据库使用 limit PageHelper插件

首先在pom.xml中引入依赖。

<!-- pagehelper --> <dependency> <groupId>com.github.pagehelper</groupId> <artifactId>pagehelper</artifactId> <version>5.1.8</version> </dependency> <dependency> <groupId>com.github.pagehelper</groupId> <artifactId>pagehelper-spring-boot-starter</artifactId> <version>1.2.13</version> </dependency>

2.使用方法：

PageHelper.startPage(pageNum,pageSize); List<Map<String, Object>> data = groupTypeService.findList(map); PageInfo<Map<String, Object>> info = new PageInfo<>(data);

3.application.properties文件中

#pagehelper分页插件配置 以下属性不加也可以实现分页功能 pagehelper.helper-dialect=mysql pagehelper.reasonable=true pagehelper.support-methods-arguments=true pagehelper.params=count=countSql

## **Mysql**

##### **执行顺序：**

from >>> on >>> join >>> where >>> group by >>> avg,sum... >>> having >>> select >>> distinct >>> order by >>> limit

##### **当查询功能比较慢时，可以从以下几个方面进行优化：**

数据库优化

数据库是查询性能的关键因素。可以通过以下方法来优化数据库性能：

创建索引：索引可以提高查询速度。根据查询需求创建适当的索引，可以很大程度上提高查询速度。

查询优化：如果使用多条 SQL 语句来完成一个任务，则可以考虑合并查询语句，减少数据库访问次数，从而减轻数据库负担。

数据表分割：在数据表中拆分数据，提高查询效率。例如，将一个大型用户表分成多个小表，可以减少每次查询时加载的数据量。

代码优化

在编写查询代码时，一定要注意代码的效率。可以采取以下措施来提高查询效率：

减少重复查询：如果一些重复的查询可以被复用，就可以将这些结果缓存起来，减少重复查询。常见的缓存方式包括使用内存缓存或者分布式缓存。

使用适当的框架和工具：选择优

秀的 ORM 框架和工具，可以帮助我们更快速地开发查询功能，并提供高效的数据访问方式。

架构优化

如果查询能力仍然无法满足需求，可以考虑从架构层面对查询功能进行优化：

数据分片：采用分布式数据库，将数据水平分割成多个独立的数据块，并存储在不同的物理节点上，可以实现数据高可用和负载均衡。

缓存机制：将经常访问的数据暂存到缓存中，以此来提高查询速度。例如使用 Redis 等内存数据库做数据缓存。

当查询功能比较慢时，需要对查询/SQL 语句进行优化，并从数据库、代码和架构等多方面入手，找到问题并解决它们以提高查询的性能并优化用户体验。

# SQL优化：

建立索引。

SQL 语句尽量用大写的。

使用表的别名。

SELECT 子句中避免使用 \* 。

根据需要用 UNION ALL 替换 UNION。

用>=替代>。

用 EXISTS 替换 DISTINCT。

用 Where 子句替换 HAVING 子句。

用 EXISTS 替代 IN、用 NOT EXISTS 替代 NOT IN。

用 WHERE 替代 ORDER BY。

多使用commit。

## **查看sql语句的执行计划**

EXPLAIN SELECT \* FROM table\_name;

## 什么是视图？

视图是基于SQL语句的结果集的可视化的表。

视图包含行和列，就像一个真实的表。视图中的字段就是来自一个或多个数据库中的真实的表中的字段。视图并不在数据库中以存储的数据值集形式存在，而是存在于实际引用的数据库表中，视图的构成可以是单表查询，多表联合查询，分组查询以及计算(表达式)查询等。行和列数据来自由定义视图的查询所引用的表，并且在引用视图时动态生成。

## **索引**

定义

在数据库中，索引是用于提高数据检索速度的一种数据结构。然而，不是所有的列都适合建立索引，以下是一些可能需要创建索引的列的情况： 经常用于搜索条件的列：如果一个列经常用于查询的WHERE子句中，那么这个列可能就是一个很好的候选索引。 用于连接其他表的列：如果你经常需要通过某一列来连接其他表（在JOIN操作中），那么对这一列建立索引可能会提高连接的速度。 具有高度唯一性的列：如果一个列的值具有高度的唯一性，那么这个列可能就是一个很好的候选索引。因为索引的目标就是加快查找速度，所以当一个列的值非常唯一时，使用索引查找将会非常快。 经常用于排序的列：如果一个列经常需要进行排序操作，那么这个列可能就是一个很好的候选索引。因为索引本身就是一种排序的数据结构，所以在已经建立索引的列上进行排序操作会更快。 经常用于数据分组的列：如果一个列经常用于GROUP BY操作，那么这个列可能就是一个很好的候选索引。 然而，虽然索引可以提高查询的速度，但是过多的索引可能会导致写入操作（如INSERT, UPDATE, DELETE）的速度变慢，因为在写入数据的同时，数据库还需要更新索引。因此，应根据实际的业务需求和数据特性，合理选择需要建立索引的列。

Java Development Kit (JDK) 1.8，也被称为Java 8，引入了许多新特性以改进编程效率和系统性能。下面是一些主要的新特性： Lambda 表达式 (Lambda Expressions)：Lambda表达式是Java 8的最大变革。它允许我们将函数作为一个方法参数（函数作为参数），或者将代码看作数据。Lambda表达式有助于减少冗余代码，因此程序看起来更加清晰。 函数式接口 (Functional Interfaces)：Java 8在java.util.function包中引入了许多新的函数式接口。这些接口提供了一种在Lambda表达式和方法引用中使用的目标类型。 默认方法 (Default Methods)：Java 8允许我们向接口添加非抽象的方法实现，这就是所谓的默认方法或者扩展方法。 方法引用 (Method References)：方法引用是一种更简洁、更易读的Lambda表达式，它用来直接引用已存在的方法或构造器。 Stream API：Stream API是Java 8中的新特性，可以用于在Java中的集合、数组等数据源进行操作。Stream API可以并行执行操作，从而获得更高的性能。 Optional 类：Java 8 引入了一个新的 java.util.Optional<T> 类，可以防止 NullPointerException 异常。 日期/时间 API (Date/Time API)：Java 8引入了新的日期和时间API，解决了旧版日期/时间类库的一些问题。 接口的静态方法：Java 8允许我们在接口中定义静态方法。可以直接从接口中调用这些静态方法。 并行数组：Java 8增加了大量的新方法，使我们可以并行地处理数组，从而获得更高的性能。 Base64：Java 8提供了一个新的Base64类，用于编码和解码Base64数据。 Nashorn JavaScript引擎：Java 8引入了一个新的Nashorn JavaScript引擎，它允许在JVM上执行JavaScript代码。 以上就是Java 8（JDK 1.8）的一些主要新特性。这些新特性为开发者提供了更强大的编程工具，提高了编程效率，并且使得代码更加简洁易读。

工具

Visualgo (<https://visualgo.net/en/bst>)：Visualgo是一个用于学习数据结构和算法的可视化网站，其中包括B+树的可视化演示。 USFCA.edu（<https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html>）：这个网站提供了一些有关计算机科学数据结构和算法的可视化，其中包括B+树。

创建索引

CREATE [UNIQUE] INDEX index\_name  
ON table\_name (column1, column2, ...);

其中：

index\_name：索引的名称，可以自定义，用于在后续的查询中引用索引。

table\_name：要在其上创建索引的表名。

column1, column2, ...：要在其上创建索引的列名。可以指定一个或多个列。多个列之间用逗号分隔。

需要创建索引的列的情况

在数据库中，索引是用于提高数据检索速度的一种数据结构。然而，不是所有的列都适合建立索引，以下是一些可能需要创建索引的列的情况：

经常用于搜索条件的列：如果一个列经常用于查询的WHERE子句中，那么这个列可能就是一个很好的候选索引。

用于连接其他表的列：如果你经常需要通过某一列来连接其他表（在JOIN操作中），那么对这一列建立索引可能会提高连接的速度。

具有高度唯一性的列：如果一个列的值具有高度的唯一性，那么这个列可能就是一个很好的候选索引。因为索引的目标就是加快查找速度，所以当一个列的值非常唯一时，使用索引查找将会非常快。

经常用于排序的列：如果一个列经常需要进行排序操作，那么这个列可能就是一个很好的候选索引。因为索引本身就是一种排序的数据结构，所以在已经建立索引的列上进行排序操作会更快。

经常用于数据分组的列：如果一个列经常用于GROUP BY操作，那么这个列可能就是一个很好的候选索引。

然而，虽然索引可以提高查询的速度，但是过多的索引可能会导致写入操作（如INSERT, UPDATE, DELETE）的速度变慢，因为在写入数据的同时，数据库还需要更新索引。因此，应根据实际的业务需求和数据特性，合理选择需要建立索引的列。

不适合建立索引：

高度重复的列

如果某个列（如性别、国籍等）只有很少的几个值，那么对该列建立索引不会起到多大作用，反而会增加索引维护的开销。因此，在这种情况下不建议建立索引。

查询数据量很小的表

当表中的记录数非常少时，如只有几百行数据，即使不使用索引，查询的速度也很快。此时建立索引所带来的性能提升非常有限，反而会浪费存储空间和增加索引的维护成本。

频繁增删改的表

在频繁进行数据插入、更新或删除操作的表上建立索引，会增加索引修改的开销，降低写入性能并增加系统负担。因此，在这种场景下，需谨慎建立索引。

关联性很强的表

如果两个表之间已经通过外键关联，假如其中一个表已经为关联字段上建立了索引，那么就不必再在另一个表对应字段上再建一个索引，因为已经可以通过已有的索引完成查询操作。

总之，虽然建立索引可以提高查询效率，但如果不适当或者过度使用索引，反而会增加查询、写入成本和维护开销，甚至可能导致性能下降。因此，在需建立索引时应结合具体业务场景及实验数据进行分析，找到合适的方案来优化数据库性能。

索引不被使用的情况：

查询语句中使用 \*；

索引列上有函数运算；

语句中有隐式转换；

表中的数据较小，或者需要查询大部分的数据。数据库自动选择最优，可能不走索引；

语句中有！= 可能不走索引。

如何建立索引

create index index\_name on table\_name (column\_name）

CREATE [UNIQUE] INDEX index\_name  
ON table\_name (column1, column2, ...);

其中：

index\_name：索引的名称，可以自定义，用于在后续的查询中引用索引。

table\_name：要在其上创建索引的表名。

column1, column2, ...：要在其上创建索引的列名。可以指定一个或多个列。多个列之间用逗号分隔。

如何查看语句是否走了索引？

在语句上加入关键字explain

## **动态sql**

### where关键字

where 元素只会在子元素返回任何内容的情况下才插入 “WHERE” 子句。而且，若子句的开头为 “AND”或 “OR”，where 元素也会将它们去除

<select id="queryBlogIf" parameterType="map" resultType="blog">  
select \* from blog  
<where>  
<if test="title != null and title != ''">  
title = #{title}  
</if>  
<if test="author != null and author != ''">  
and author = #{author}  
</if>  
</where>  
</select>

### set关键字

上面的对于查询 SQL 语句包含 where 关键字，如果在进行更新操作的时候，含有 set 关键词，

<!--注意set是用的逗号隔开-->  
<update id="updateBlog" parameterType="map">  
update blog  
<set>  
<if test="title != null and title != '' ">  
title = #{title},  
</if>  
<if test="author != null and author != '' ">  
author = #{author}  
</if>  
</set>  
where id = #{id}  
</update>

### choose关键字

有时候，我们不想用到所有的查询条件，只想选择其中的一个，查询条件有一个满足即可，使用 [choose标签](https://so.csdn.net/so/search?q=choose%E6%A0%87%E7%AD%BE&spm=1001.2101.3001.7020)可以解决此类问题，类似于 Java 的 switch 语句

<select id="queryBlogChoose" parameterType="map" resultType="blog">  
  select \* from blog  
 <where>  
   <choose>  
     <when test="title != null and title != '' ">  
         title = #{title}  
     </when>  
     <when test="author != null and author != '' ">  
          and author = #{author}  
     </when>  
     <otherwise>  
          and views = #{views}  
     </otherwise>  
   </choose>  
 </where>  
</select>

### foreach关键字

注意：如果需要用SQL进行批量操作的话，数据库连接参数需要设置下面参数：

allowMultiQueries=true&rewriteBatchedStatements=true

jdbc:mysql://127.0.0.1:3306/test?  
useUnicode=true&characterEncoding=utf8&allowMultiQueries=true&rewriteBatchedStat  
ements=true&zeroDateTimeBehavior=convertToNull&useSSL=true&serverTimezone=GMT%2B8

#### **1.新增**

<insert id="batchInsertBlog" useGeneratedKeys="true" keyProperty="id">  
insert into blog (title, author, views, status) values  
<foreach collection="blogList" item="blog" separator=",">  
(#{blog.title}, #{blog.author}, #{blog.views}, #{blog.status})  
</foreach>  
</insert>

mapper接口写

int batchInsertBlog(List<Blog> blogList);

#### **2.删除**

<delete id="batchDeleteByIds">  
DELETE FROM blog WHERE id in  
<foreach collection="ids" item="item" index="index" open="(" separator=","close=")">  
#{item}  
</foreach>  
</delete>

mapper接口写

int batchDeleteByIds(@Param("ids") List<String> ids);

#### **3.修改**

对所有的表多适用 ， 共用写法

<update id="updateTable">  
UPDATE ${tableName} SET  
<foreach collection="dataMap" index="key" item="value" separator="," >  
${key} = #{value}  
</foreach>  
WHERE  
id = #{id}  
</update>

mapper接口写

void updateTable(@Param("tableName") String tableName, Param("id") String id,  
@Param("dataMap") HashMap dataMap);

#### **4.查询**

批量查询

<select id="getAllBlog" resultType="blog">

SELECT title, author, views, status

FROM blog

<if test="blogList != null and blogList.size() > 0">

WHERE id in

<foreach collection="blogList" item="item" index="index" open="("

separator="," close=")">

#{item}

</foreach>

</if>

</select>

mapper接口写

void getAllBlog(@Param("blogList") List<Long> blogList);

### SQL片段

有时候可能某个 sql 语句我们用的特别多，为了增加代码的重用性，简化代码，我们需要将这些代码抽取出来，然后使用时直接调用。 提取SQL片段：

<sql id="if-title-author">

<if test="title != null and title != '' ">

title = #{title}

</if>

<if test="author != null and author != '' ">

and author = #{author}

</if>

</sql>

引用SQL片段：

<select id="queryBlogIf" parameterType="map" resultType="blog">

select \* from blog

<where>

<!-- 引用 sql 片段，如果refid 指定的不在本文件中，那么需要在前面加上 namespace-->

<include refid="if-title-author"></include>

<!-- 在这里还可以引用其他的 sql 片段 -->

</where>

</select>

注意事项：

最好基于 单表来定义 sql 片段，提高片段的可重用性在 sql 片段中不要包括 where

### bind元素

bind 元素允许你在 OGNL 表达式以外创建一个变量，并将其绑定到当前的上下文。比如：

<select id="selectBlogsLike" resultType="Blog">

<bind name="pattern" value="'%' + \_parameter.getTitle() + '%'" />

SELECT \* FROM BLOG

WHERE title LIKE #{pattern}

</select>

## **HTTP协议:**

超文本传输协议（Hyper Text Transfer Protocol）:是一种位于应用层超文本传输协议。可以通过http协议在客户端和服务端之间通信。

它是一种短连接，无状态的协议。http1.0是无连接无状态的，http1.1是短连接，无状态的。

短连接是指一次请求和响应后，连接一会就关闭，这种方式可以节省资源。

无状态是指服务端在收到客户端发来的消息后，只知道收到了一条消息，并不知道是谁发送的。解决这个问题可以在客户端加上cookie。

http协议分为请求协议和响应协议:

##### **请求协议：**

请求首行 （GET /servletLearn/s1 HTTP/1.1） 请求方式 资源路径 协议版本号

请求头  key：value形式

请求空行

请求体：传输数据 （get请求没有请求体）



Connection:keep-alive 短连接

options：预提交方法

        有8中请求方式：Get，Post，Put，Head，Delete，Options，Trace， Conntection

##### **响应协议：**

响应首行 （HTTP/1.1200）协议版本号 响应状态码

响应头  key：value形式

响应空行

响应体：



响应状态码：1xx—5xx

   1xx  100：未返回数据，2xx  200：成功返回数据，3xx  302：重定向

   4xx：前台错误 400 Bad Request：请求参数有误，404请求路径错误，405 请求方法错误 ,415 请求格式错误(json,from)

   5xx：后台错误 500：服务器内部错误，无法完成请求

##### **HTTP 和 HTTPS（HTTP Secure）是两种不同的传输协议，它们之间的主要区别如下：**

安全性

HTTP 传输协议具有明文的特点，数据在传输过程中不加密，存在被窃取、篡改等风险；而 HTTPS 则采用 SSL/TLS 协议进行数据加密和身份验证，保证数据传输的安全可靠。

端口号

HTTP 默认端口为 80，HTTPS 默认端口为 443。当然，这些端口可以配置修改。

证书

自己申请一个免费的证书

HTTPS 协议需要使用证书来建立安全连接，通过数字证书机构（CA）对网站的身份进行认证，防止恶意攻击和中间人攻击。

性能

由于 HTTPS 协议涉及到加密和解密等过程，相较于 HTTP 协议，性能会稍微有所下降，同时也需要占用更多的资源。

综上所述，HTTP 协议适合在内网环境或者不涉及安全性要求的场景下使用；而 HTTPS 协议则适合关注用户隐私和数据安全的 Web 应用场景，如银行、电商、社交网络等网站。

## **spring事务**

声明式事务，首先在xml中配置事务的配置信息，tx：annocation-deriven

然后在需要支持事务的类上或是方法上写@Transactional标签。标签中默认事务的传播行为是required。即如果该方法有事务，就使用该方法的事务，如果该方法没有事务，就自己创建一个事务。传播行为有7中，我们使用的的最多的就是默认的传播事务。

spring是如何管理事务的？在执行支持事务的方法时，如果方法没有抛出异常，正常执行就会执行commit，如果方法抛出了异常，就会执行回滚。我们可以在标签中，设置robackfor的值指定回滚时机。rollbackfor：用于指定触发事务回滚的异常类型，用的最多是RunTimeException，还有theowable。

## **数据库事务**

##### **数据库事务的四大特性（ACID）**

###### **原子性（Atomicity）**

原子性是指事务包含的所有操作要么全部成功，要么全部失败回滚。因此事务的操作如果成功就必须要完全应用到数据库，如果操作失败则不能对数据库有任何影响。

###### **一致性（Consistency）**

一致性是指事务必须使数据库从一个一致性状态变换到另一个一致性状态，也就是说一个事务执行之前和执行之后都必须处于一致性状态。

###### **隔离性（Isolation）**

隔离性是当多个用户并发访问数据库时，比如操作同一张表时，数据库为每一个用户开启的事务，不能被其他事务的操作所干扰，多个并发事务之间要相互隔离。

###### **持久性（Durability）**

持久性是指一个事务一旦被提交了，那么对数据库中的数据的改变就是永久性的，即便是在数据库系统遇到故障的情况下也不会丢失提交事务的操作。

##### **事务的四大隔离级别及对应存在的问题**

###### **read uncommitted（读未提交）**

第一隔离级别怕回滚，因为它能读取到别的事务未提交的数据。如果当前事务将这个数据当成真正的数据，那么如果别的事务回滚，那么就会数据错误。导致脏读。

脏读：是指在一个事务处理过程里读取了另一个未提交的事务中的数据。

###### **read committed（读已提交）**

第二隔离级别怕提交。提交会导致不可重复读。

不可重复读：不可重复读是指在对于数据库中的某个数据，一个事务范围内多次查询却返回了不同的数据值，这是由于在查询间隔，被另一个事务修改并提交了。

###### **repeatable read (可重复读)**

第三隔离级别。在MySQL数据库中默认的隔离级别为Repeatable read (可重复读)。两个事务可以同时执行查询操作，但是只有一方可以执行增删改，一方一旦执行增删改就会上锁，另一方需等待commit或rollback。

幻读： 一方执行insert语句之后提交，另一方执行update语句，就会把insert语句给刷新出来。

第二个事务进行增删改的时候另一个事务进行查询，查询显示不到第一个事务新增的内容但是实际存在，幻读问题

###### **Serializable (串行化)**

第四隔离级别，使用排它锁（锁增删改差）：只要有一方增删改，最后执行查询的就会上锁。隔离级别最高的是Serializable级别，可避免脏读、不可重复读、幻读的发生。



不可重复读和脏读的区别是：脏读是某一事务读取了另一个事务未提交的脏数据，而不可重复读则是读取了前一事务提交的数据。

##### **行级锁**

数据库事务的行级锁（Row-Level Locking）是一种并发控制机制，用于保护数据库中的行级数据，确保多个事务之间对同一行数据的并发操作不会出现数据不一致或冲突的情况。

行级锁是在数据库引擎层面实现的，它可以在事务级别或语句级别对数据行进行加锁。行级锁的主要特点如下：

细粒度控制：相比于表级锁或页级锁，行级锁提供了更细粒度的数据控制，可以在并发访问中实现更高的并发性和灵活性。

并发性能：行级锁的使用可以提高并发性能，允许多个事务同时访问同一张表的不同行，避免了不必要的阻塞或冲突。

数据一致性：行级锁可以保护数据的一致性，避免了多个事务对同一行数据进行不一致的修改或读取。

锁冲突：行级锁可能导致锁冲突，当多个事务同时请求对同一行数据进行修改或读取时，可能会发生锁等待和死锁的情况，需要进行适当的处理和调优。

数据库系统通常会自动管理行级锁，并根据事务的隔离级别和操作类型来自动获取和释放锁。在具体使用行级锁时，需要根据业务需求和性能要求选择合适的锁粒度，并了解数据库系统对行级锁的支持和行为。

需要注意的是，行级锁的实现和行为可能因不同的数据库管理系统而有所差异，因此在具体的数据库产品中使用行级锁时，建议查阅相应的文档和文档，了解其具体的实现方式和行为规则。

数据级别的行级锁： 数据级别的行级锁是在SQL语句级别上设置的，可以更加精细地控制锁的范围和持有时间。常见的数据级别锁包括：

FOR UPDATE：

在SELECT语句中使用FOR UPDATE子句可以获取指定数据行的排他锁。其他事务无法获取该数据行的共享锁或排他锁，直到当前事务释放锁。

LOCK IN SHARE MODE：

在SELECT语句中使用LOCK IN SHARE MODE子句可以获取指定数据行的共享锁。其他事务可以获取相同数据行的共享锁，但无法获取排他锁。

数据级别的行级锁需要开发人员显式地设置，并在事务中手动管理锁的获取和释放。这种方式提供了更细粒度的控制，但也需要开发人员更加谨慎地处理锁的使用，以避免死锁或性能问题。

## **Java类**

如何将字符串反转

使用StringBuilder或StringBuffer：利用StringBuilder或StringBuffer的reverse()方法直接反转字符串。

javaCopy codepublic static String reverseString(String str) {

StringBuilder stringBuilder = new StringBuilder(str);

return stringBuilder.reverse().toString();

}

使用递归：使用递归函数将字符串从最后一个字符开始反向连接。

javaCopy codepublic static String reverseString(String str) {

if (str.isEmpty()) {

return str;

} else {

return reverseString(str.substring(1)) + str.charAt(0);

}

}

这些方法都可以将给定的字符串反转。选择合适的方法取决于具体的需求和性能要求。请注意，上述示例中的方法都是基于Java语言的，可以根据需要进行调整和适应不同的编程语言。

## **反射**

程序在运行时动态地获取类信息、创建对象、调用方法或修改属性等。

在编译时根本无法知道该对象或类可能属于哪些类，程序只依靠运行时信息来发现该对象和类的真实信息。

通过反射可以使程序代码访问装载到JVM 中的类的内部信息

public static void main(String[] args) throws Exception {

// 1.加载字节码对象

Class<?> c = Class.forName("com.invoke.Person");

// 2.获取全部的构造方法对象

Constructor<?>[] constructors = c.getConstructors();

Constructor<?> constructor = c.getConstructor(int.class,String.class,int.class);

Object person = constructor.newInstance(10,"zs",20);//通过构造方法创建对象

for(Constructor con:constructors) {

System.out.println(con);

}

// 3.获取所有的成员变量对象，并用for循环打印出所有变量的修饰符对应的数字

Field[] declaredFields = c.getDeclaredFields();//获取所有成员变量对象

//循环打印出

for(Field f:declaredFields) {

System.out.println(f);

}

for(Field f:declaredFields) {

System.out.print(f.getType()+" ");//变量类型

System.out.print(f.getModifiers()+" ");//变量的权限修饰符

System.out.print(f.getName());//变量名字

System.out.println();

}

//for循环打印出所有变量的修饰符对应的数字

for(Field f:declaredFields) {

System.out.print(f.getModifiers()+" ");

}

// 4.获取account成员变量对象

Field field = c.getDeclaredField("account");

// Person person2 = new Person();

// person2.setAccount(new BigDecimal(100));//对象.属性(值)

field.set(person,new BigDecimal(100));//属性.set(对象,值)

field.get(person);

System.out.println(field);

// 5.获取所有的成员方法对象

Method[] methods = c.getDeclaredMethods();

Method[] methods2 = c.getMethods();

for(Method m:methods) {

System.out.println(m);

}

System.out.println();

// 6.获取howold方法对象，并获取howold方法的返回值

Method howold = c.getMethod("howold", int.class);

Object invoke = howold.invoke(person, 12);

System.out.println(howold.getReturnType());//获取返回值类型

// 7.调用introduction方法。

Method introduction = c.getMethod("introduction", Person.class);

Person per = new Person();

per.setName("张三");

per.setAge(23);

per.setId(1000);

//调用introduction方法对象，传入的person对象的自我介绍

introduction.invoke(person, per);

// introduction.invoke(per, per);

}

## **代理**

##### **静态代理：**

静态代理是一种在编译时就已经确定代理类的技术，它需要手动编写代理类来对目标对象进行包装。静态代理通常会实现与原始类相同的接口，并且在代理类中调用目标对象的方法之前或之后执行一些额外的逻辑，比如记录日志、验证参数等。静态代理可以帮助我们更方便地控制对原始类的访问，并且不需要对原始类的代码做出任何改变。

委托人将需求定义到接口中。

委托人和代理人都去实现这个接口。

代理人中添加委托人的成员变量和构造方法。

代理人在同名的方法中，调用委托人的方法。

代理人在同名的方法中，实现增强的功能。

缺点：

代理人的接口是指定的，一个代理人只可以实现一个委托人的指定需求。

代理人和委托人也是指定的。

##### **动态代理：**

动态代理是一种在运行时动态地生成代理类的技术，它可以在不修改原始类的情况下对其方法进行增强或拦截。使用动态代理可以实现很多功能，比如跟踪方法调用、实现 AOP 编程、实现远程方法调用等。在 Java 中，动态代理主要是利用反射机制来实现的。

委托人将需求定义到接口中。

代理类实现InvocationHandler 接口，委托人实现原有的接口。

代理人中添加object的成员变量和构造方法。

代理人在invoke的方法中，调用委托人的方法。

代理人在invoke的方法中，实现增强的功能。

动态代理是 Java 中常用的一种设计模式，通过在运行时创建一个实现特定接口或继承指定类的代理对象，在代理对象中可以进行额外的操作，如日志记录、性能统计、缓存等。下面介绍一种基于 Java 的动态代理实现方式：

定义接口

首先需要定义一个接口，该接口包含需要被代理的方法。

public interface MyService {

void method1();

String method2(int arg);

}

实现 InvocationHandler 接口

定义一个实现 InvocationHandler 接口的类，该类中包含一个 target 对象，代表实际要被代理的对象，并且可以在 invoke() 方法中对方法进行增强处理。

import java.lang.reflect.InvocationHandler;

import java.lang.reflect.Method;

public class MyInvocationHandler implements InvocationHandler {

private Object target;

public MyInvocationHandler(Object target) {

this.target = target;

}

@Override

public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {

System.out.println("Before method: " + method.getName());

// 调用实际对象的方法

Object result = method.invoke(target, args);

System.out.println("After method: " + method.getName());

return result;

}

}

创建代理对象

使用 Proxy 类的 newProxyInstance() 方法创建一个代理对象，该方法需要传入 ClassLoader 对象、代理的接口类型以及 InvocationHandler 对象。

import java.lang.reflect.Proxy;

public class MyProxy {

public static Object createProxy(Object target) {

return Proxy.newProxyInstance(target.getClass().getClassLoader(),

target.getClass().getInterfaces(),

new MyInvocationHandler(target));

}

}

测试代理对象

public static void main(String[] args) {

MyService myService = new MyServiceImpl();

// 创建代理对象

MyService myServiceProxy = (MyService) MyProxy.createProxy(myService);

// 调用代理对象的方法

myServiceProxy.method1();

String result = myServiceProxy.method2(123);

System.out.println("Result: " + result);

}

在代理对象中，调用目标对象的方法前后可添加额外操作，但不需要改变原有的代码逻辑，提高代码复用性和灵活性。

## **分布式事务**

在多个服务中同时成功同时失败

就是一次大的操作由不同的小操作组成，这些小的操作分布在不同的服务器上，且属于不同的应用，分布式事务需要保证这些小操作要么全部成功，要么全部失败。本质上来说，分布式事务就是为了保证不同数据库的数据一致性。

分布式事务是指事务的参与者、支持事务的服务器、资源服务器以及事务管理器 「分别位于不同的分布式系统的不同节点之上」 。

分布式事务是指跨越多个计算机系统的事务。在分布式系统中，由于数据和计算资源的分散和并行处理，可能会出现一些数据不一致的问题。分布式事务的目的是确保在跨越多个计算机系统的事务中，所有相关操作的一致性和原子性。

###### **分布式理论**

CAP定理：

一致性（C）：在分布式系统中的所有数据备份，「在同一时刻是否拥有同样的值」 。（等同于所有节点访问同一份最新的数据副本）

可用性（A）：在集群中一部分节点 「故障」 后，集群整体 「是否还能响应」 客户端的读写请求。（对数据更新具备高可用性）

分区容错性（P）：即使出现 「单个组件无法可用,操作依然可以完成」 。

BASE理论：

「Basically Available（基本可用）」

「Soft state（软状态）」

「Eventually consistent（最终一致性）」

###### **分布式事务解决方法：**

数据库层面

两阶段提交协议（Two-Phase Commit Protocol）：该协议由一个协调者和多个参与者组成，协调者负责协调参与者的事务提交。 该协议的第一阶段是协调者向所有参与者发送准备提交消息，等待所有参与者响应。 第二阶段是协调者向所有参与者发送提交或回滚消息，根据参与者的响应进行决策。 但是如果协调者此时宕机了，参与者一直收不到协调者的请求，就会一直占用着锁，占用资源。导致其他线程阻塞。这样并发性很低。（编程式的分布式事务解决方案）

三阶段提交（3pc）：在2pc的基础上引入了超时机制，当协调者宕机后，参与者如果一直收不到协调者的消息，会自动提交。但是这也会造成数据不一致的问题，如有在其中有的参与者操作失败需要进行回滚操作，但是另一部分完成了操作，进行了提交，这样就导致了数据不一致问题。

编程层面

补偿事务（Compensating Transaction）：「 针对每个操作，都要注册一个与其对应的确认和补偿（撤销）操作」 。补偿事务通常用于在分布式事务失败时回滚已经提交的操作。它通过执行相反的操作来恢复系统状态。例如，如果一个订单已经被支付，但是发生了错误导致订单无法完成，则补偿事务可以撤销付款并将订单状态设置为未支付。「Try,Confirm,Cancel」

Try阶段主要是对 「业务系统做检测及资源预留」

Confirm 阶段主要是对 「业务系统做确认提交」 ，Try阶段执行成功并开始执行 Confirm阶段时，默认 Confirm阶段是不会出错的。即：只要Try成功，Confirm一定成功。

Cancel 阶段主要是在业务执行错误，需要回滚的状态下执行的业务取消，「预留资源释放」 。

操作俩个数据库,比如说一个成功一个失败，将成功那个执行相反的操作。

4。本地消息表，定时任务或者

定时任务，用catch里面记录到日志中，宕机就没法记录。将发送的消息记录到本地消息表中，然后没有成功

无论哪种方法，分布式事务都是一个复杂的问题，需要仔细考虑并在系统设计和实现中加以处理。

## **分布式锁**

分布式锁是一种常见的处理分布式系统并发访问问题的技术，它可以保证在分布式系统中的多个节点上对某个共享资源进行互斥访问。

实现分布式锁的方式有很多种，下面介绍几种常用的实现方式：

基于数据库的实现

可以利用数据库的事务特性来实现分布式锁。比较简单的方法是，在数据库中创建一个表，每行代表一个被锁定的资源，使用该行的主键作为锁名称（或将锁名称保存在另一张表中）。在获取锁时，使用 SELECT ... FOR UPDATE 语句将该行记录加锁，并在同一事务中进行后续操作；释放锁时，提交事务即可。该方法的优点是易于理解和实现，缺点是需要访问数据库，性能较低，如果数据库出现问题可能影响系统正常运行。

基于缓存的实现

常见的缓存如 Redis 或 Memcached 等可以通过设置缓存过期时间来实现分布式锁。具体方法是，在缓存中定义一个以锁名称为 key 的缓存项，并指定过期时间；尝试获取锁时，使用 SETNX 命令进行原子操作，如果返回值为 1 则表示加锁成功，并在后续操作中指定适当的超时时间；释放锁时，使用 DEL 命令将缓存项删除即可。该方法的优点是性能较高、易于实现和扩展，缺点是存在死锁和锁竞争等问题。

基于 ZooKeeper 的实现

ZooKeeper 是一个分布式协调服务，可以通过创建临时节点来实现分布式锁。具体方法是，在 ZooKeeper 中创建一个唯一名称的临时节点，并以该节点为锁。尝试获取锁时，创建该节点并检查是否为最小序号节点，如果是则表示加锁成功，否则对其前一个节点注册监听器；释放锁时，删除该节点。该方法的优点是可靠性高，通用性好，缺点是性能较低、实现复杂、依赖于 ZooKeeper 服务的稳定性。

以上是三种常见的分布式锁实现方式，还有其他的分布式锁实现方式，如基于文件、消息队列等实现。根据具体应用场景和要求，选择适合的实现方式来保证系统的正确性和效率。

## **Redis**

5种数据类型：String，Hash，List，Set（无序集合），Sorted set（有序集合）

是对于value类型来说 key都是字符串的

##### **分布式锁**

Redis 的分布式锁是一种基于 Redis 数据库实现的锁机制，用于在分布式环境下控制对共享资源的访问，保证同一时刻只有一个线程或进程可以获得锁并执行关键操作，从而确保数据的一致性和并发控制。

常见的分布式锁实现方式有两种：

基于 SETNX 命令：通过使用 Redis 的 SETNX（SET if Not eXists）命令来实现。该命令可以在指定的键不存在时设置键的值，并返回成功与否的状态。利用这个特性，可以将某个键作为锁的标识，在获取锁时设置键的值，如果设置成功，则表示获取到了锁；否则表示锁已经被其他线程或进程持有。

基于 Lua 脚本：通过执行 Lua 脚本来实现分布式锁。Lua 脚本可以在 Redis 服务器端原子地执行多个命令，确保获取锁和释放锁的操作是原子的，避免竞态条件。脚本可以利用 Redis 的 SETNX 命令和 EXPIRE 命令来设置锁的值和过期时间，并在释放锁时根据锁的值验证并删除锁。

使用 Redis 分布式锁时需要考虑以下注意事项：

设置合理的锁超时时间：确保获取到锁的线程在执行完关键操作后及时释放锁，避免锁长时间占用，导致其他线程无法获取锁。

避免死锁：在释放锁时需要保证线程持有的锁与要释放的锁是一致的，避免误释放其他线程持有的锁。

锁的粒度：根据实际情况，合理选择锁的粒度，避免过大或过小的锁粒度影响系统性能。

容错处理：在获取锁和释放锁的过程中，需要处理可能出现的异常情况，例如 Redis 连接异常、锁被意外释放等，保证系统的可靠性和稳定性。

分布式锁是在分布式系统中常用的并发控制机制，通过 Redis 实现的分布式锁具有高性能、高可用性和易用性的特点，广泛应用于分布式场景中的并发控制和资源竞争问题。

##### **设置锁超时时间**

在使用 Redis 实现分布式锁时，设置锁超时时间是一种重要的机制，可以防止锁长时间被占用而导致死锁或资源浪费的情况发生。下面是设置锁超时时间的一般步骤：

在获取锁时，设置锁的键和值，并同时设置锁的过期时间。可以使用 Redis 的 SETNX 命令或执行 Lua 脚本来实现。

示例代码（使用 SETNX 命令）：

SETNX lock\_key "locked"

EXPIRE lock\_key lock\_timeout

示例代码（使用 Lua 脚本）：

local lock\_key = KEYS[1]

local lock\_timeout = ARGV[1]

if redis.call("SETNX", lock\_key, "locked") == 1 then

redis.call("EXPIRE", lock\_key, lock\_timeout)

return "OK"

else

return "FAILED"

end

执行关键操作。在获取到锁后，执行需要加锁的关键操作。

在关键操作执行完毕后，释放锁。可以通过删除锁的键来释放锁。

示例代码：

cssCopy code

DEL lock\_key

设置锁超时时间的关键是在获取锁时设置锁的过期时间，确保锁在一定时间后自动释放，避免长时间占用。一般来说，锁的超时时间应根据业务需求和关键操作的预估执行时间来合理设置。如果关键操作执行时间较长，建议将锁的超时时间设置得更长一些，以避免在关键操作未完成时锁被自动释放。但同时也要注意，设置过长的超时时间可能会导致其他线程或进程在等待获取锁时的延迟。

需要注意的是，在使用分布式锁时，锁的超时时间应该合理设置，避免因为锁的过早释放或过长占用导致并发控制的不准确性或性能问题。同时，还需要考虑异常处理，例如锁的获取失败或释放异常时的处理机制，以保证系统的稳定性和可靠性。

##### **Redis持久化机制**

Redis 为了保证效率，数据缓存在了内存中，但是会周期性的把更新的数据写入磁盘或者把修改操作写入追加的记录文件中，以保证数据的持久化。

有两种：

RDB：快照形式是直接把内存中的数据保存到一个 dump.rdb 的文件中，定时保存，保存策略。

AOF：把所有的对 Redis 的服务器进行修改的命令都存到一个文件里，命令的集合。Redis 默认是快照 RDB 的持久化方式。

当 Redis 重启的时候，它会优先使用 AOF 文件来还原数据集，因为 AOF 文件保存的数据集通常比 RDB 文件所保存的数据集更完整。你甚至可以关闭持久化功能，让数据只在服务器运行时存。

##### **redis缓存**

缓存击穿 ：是指在缓存中没有数据（数据过期了），但是在数据库中有数据。这时由于并发访问的用户数较多，在缓存中查询不到数据后同时去数据库中查询，造成数据库压力较大。

解决：

设置热点数据永不过期

加互斥锁。击穿的原因之一是并发查询，此时加个互斥锁，让用户排 队查询。

缓存穿透 ：是指缓存和数据库中都没有的数据。用户此时发起请求，但是因为缓存和数据库中都没有数据，导致用户一直不断请求这个数据，使数据库压力过大。

解决：用户查询没有结果的数据后返回一个代表空值的值。

缓存雪崩 ：是指缓存数据大批量同时到期，而查询的数据量过大，导致数据都从数据库中查询，使数据库压力过大。和缓存击穿不同的是，雪崩是大批量不同的数据同时过期了，击穿是并发查询同一条数据。

解决：

将缓存过期时间设置随机，避免同时过期的情况。

如果缓存数据库是分布式部署，将热点数据均匀分布在不同的缓存数据库中。

设置热点数据永远不过期。

##### **什么样的数据放在redis里面**

会话缓存：如果你的应用程序使用会话来跟踪用户，你可以将会话数据存储在Redis中，以减少数据库的负载。

全页缓存：许多网站会在Redis中缓存整个HTML页面，以便在下一次请求时直接从Redis中提取，而不是从数据库中重新生成。

用户信息/身份验证：你可以在Redis中存储用户信息，如用户名，邮箱，电话号码等。当然，你也需要确保这些信息的安全性。

队列和消息系统：Redis具有原生的队列和发布订阅功能，可以用于实现异步任务或消息队列。

排行榜或计数器：例如一个网站的页面访问次数，或者一个游戏的玩家得分排行榜。

实时分析：如果你正在对数据进行实时分析，你可以使用Redis作为一个缓冲区，将数据先存储在Redis中，然后周期性地将其写入数据库。

缓存数据库查询：如果你的应用程序经常对数据库进行一些重复的查询，那么将查询结果存储在Redis中会大大提高效率。

访问频率高：如果某些数据被频繁访问，将它们存储在内存中可以大大提高应用的性能。例如，用户会话、热门内容的详情等。

读取操作远多于写入操作：Redis适合读密集型的场景。例如，用户的基本信息，产品的详细信息，价格等，这些信息的读取次数通常远多于修改次数。

数据量不是太大：由于Redis是内存数据库，因此并不适合存储大量的数据。一般来说，数据量应该能够适应你的服务器内存大小。

对数据一致性要求不高：因为Redis通常作为缓存使用，数据可能在某些情况下（例如，缓存失效或者服务器重启）会丢失，因此，对于需要高度一致性的数据，如交易系统中的订单信息，就不适合存储在Redis中。

数据可以被序列化：Redis存储的是键值对，其中的值需要能够被序列化为字符串，所以你可以存储的数据类型受此限制。

#### **redis分片**

select 数据库 默认是16个

Redis分片是一种将数据分散存储在多个Redis节点上的技术，它可以扩展Redis数据库的处理能力以满足高并发和大规模应用程序的需求。通常情况下，实现Redis分片需要以下步骤：

划分数据：根据某些规则（如hash值）将数据分散到多个Redis节点上。

配置Redis节点：配置多个Redis实例以支持数据的保存和读取，每个Redis实例必须有不同的端口号和数据存放目录，并且它们之间要保持相互独立的状态。

客户端路由：客户端向具有该数据的特定Redis节点发出请求，或者使用代理进行客户端路由，使得请求被正确地路由到相应的Redis节点上。

水平扩展：随着数据量的增长，可以添加更多的Redis节点来水平扩展整个集群的处理能力。

需要注意的是，Redis分片需要考虑一些额外的问题，例如节点失效和数据迁移等问题。因此，在实践中，应该使用可靠的Redis分片解决方案，并遵守最佳实践以最大程度地减少数据丢失和死锁的风险。

##### **作用**

扩展存储容量：通过分片，可以将一个大的 Redis 数据库分布在多个物理节点上，每个节点存储部分数据，从而扩展整个系统的存储容量。这样可以避免单个节点存储的数据量过大，减轻了单个节点的存储压力。

提高系统吞吐量：通过将数据分散到多个节点上，每个节点处理自己负责的数据，可以提高整个系统的并发读写能力，从而提高系统的吞吐量。每个节点独立处理请求，避免了单个节点成为瓶颈。

并行处理和负载均衡：分片使得多个节点可以并行处理请求，从而提高系统的并发处理能力。同时，分片还可以实现负载均衡，将请求均匀地分配到不同的节点上，避免某个节点负载过重。

容灾和高可用性：通过复制和分片结合使用，可以实现数据的容灾和高可用性。每个分片可以配置多个副本，当某个节点出现故障时，可以从其他副本中恢复数据，保证系统的可用性。

## **Maven**

Maven 是一款基于项目对象模型（POM）的构建工具，它可以管理 Java 项目的构建过程、依赖关系以及相关文档的生成。Maven 的核心理念是「约定优于配置」，即定义好一些规范和默认值，减少不必要的配置和插件使用。

以下是 Maven 的一些常用概念和命令：

POM 文件：项目对象模型文件，描述了项目的基本信息、依赖项、构建版本等信息。

Artifact：一个可以被 Maven 管理的 jar、war 或者其他类型的文件。

Repository：存放所有依赖和构件的仓库。

生命周期：Maven 对项目构建过程进行划分并且抽象成了不同的阶段，如 clean、compile、test、package、install、deploy。每个生命周期包含多个阶段，不同的插件在各自阶段完成不同的任务。 clean：清空target文件夹         resource：将src中的代码复制到target中   compile: 将复制进target的代码编译   test-resource：重新复制一份，跟之前复制的做一个对比。   test-compile：重新编译一遍   package：根据pom.xml配置的进行打包，war、jar、pom，打的包在target文件夹中   install：安装，将package命令打好的包放到本地仓库（根据配置文件指定）中去；   deploy：发布，在install的基础上，将jar包上传到公司私服；

常用命令： （1）mvn compile：编译源代码； （2）mvn test：运行测试用例； （3）mvn package：将项目打成 jar 包、war 包等； （4）mvn clean：清除编译产生的目录和文件； （5）mvn install：将项目安装到本地仓库中； （6）mvn deploy：将项目部署到远程仓库中。

<project>：定义项目的基本信息和依赖关系。

<modelVersion>：指定POM（Project Object Model）的版本。

<groupId>：定义项目所属的组织或团队的唯一标识符。

<artifactId>：定义项目的唯一标识符。

<version>：定义项目的版本号。

<dependencies>：定义项目的依赖关系。

<dependency>：定义一个依赖项。

<groupId>：依赖项的组织或团队的唯一标识符。

<artifactId>：依赖项的唯一标识符。

<version>：依赖项的版本号。

<scope>：依赖项的作用范围，如编译时依赖、运行时依赖等。

<repositories>：定义项目使用的远程仓库。

<repository>：定义一个远程仓库。

<id>：远程仓库的唯一标识符。

<url>：远程仓库的URL。

<build>：定义项目的构建配置。

<plugins>：定义构建过程中使用的插件。

<plugin>：定义一个插件。

<groupId>：插件的组织或团队的唯一标识符。

<artifactId>：插件的唯一标识符。

<version>：插件的版本号。

<configuration>：插件的配置参数

Maven 提供了丰富的插件和组件，开发人员可以使用这些插件和组件实现各种自定义任务。同时还有很多 Maven 的工具和集成环境，如 Eclipse、IntelliJ IDEA 等，帮助 Java 开发人员更方便地使用 Maven 进行项目开发。

## **Nginx**

Nginx就是一个反向代理服务器。

##### **nginx作用：**

###### **1、部署前端资源**

将资源直接存放到html文件夹下，或者在html文件夹下新建picture文件，将资源存放在该文件下，配置文件如下：

静态资源放到root 指定的目录下

location / {

root html/picture;

index index.html index.htm;

}

<--访问picture文件下的资源路径为localhost/bizhi.jpeg-->

location / {

root html;

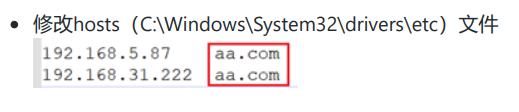
index index.html index.htm;

}

<--如果不修改配置文件root，访问地址为localhost/picture/bizhi.jpeg-->

###### **2、反向代理**

将ip地址映射成为域名，通过域名进行访问，因为域名好记。

修改hosts（C:\Windows\System32\drivers\etc）文件 

配置Nginx.conf 文件

location / {

proxy\_pass http://localhost:8080;

}

<-- 访问地址为aa.com/show.html -->

###### **3、负载均衡（提高访问量）**

由Nginx转发给多台服务器。在反向代理的基础上实现负载均衡

常用策略： 轮询、iphash

用轮询： 用到session的话，会造成session不共享问题

登录：用jwt一个用redis

redis宕机，jwt宕机也没事

ip\_hash:tomcat 访问量过大

轮询+redis:解决并发和session共享问题

<-- 配置负载均衡 -->

upstream emps {

#ip\_hash; 每个访客固定访问一个后端服务器，可以解决session的问题。

server localhost:8081 weight=5;

server localhost:8082 weight=1;

server localhost:8083 weight=10;

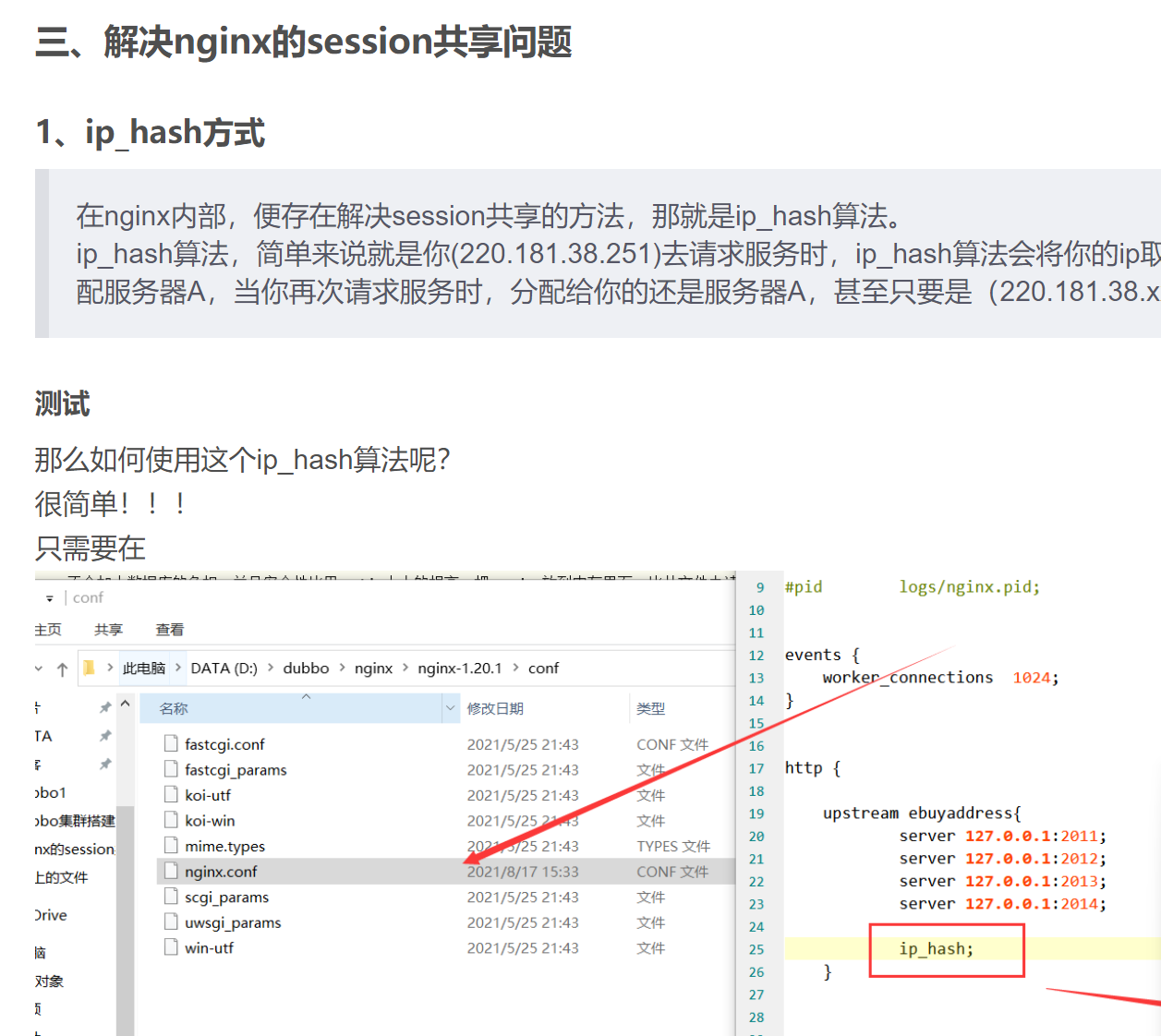
}

<-- 配置反向代理 -->

location / {

proxy\_pass http://emps;

}



###### **4、动静分离（减少tomcat的压力）**

把前端资源部署到Nginx上，后端资源部署到tomcat上。前端页面放到一个location中，后端资源放到另一个location中

#访问前端资源

location /empIsud {

root html;

index index.html index.html;

}

#配置反向代理,访问后端资源

location / {

proxy\_pass http://emps;

}

<-- 配置负载均衡 -->

upstream emps {

#ip\_hash; 每个访客固定访问一个后端服务器，可以解决session的问题。

server localhost:8081 weight=5;

server localhost:8082 weight=1;

server localhost:8083 weight=10;

}

##### **nginx常用命令：**

启动：start nginx

检查修改配置文件后是否成功：nginx -t -c ./conf/nginx.conf （successful即可）

重新加载配置文件：nginx -s reload

关闭：nginx -s quit（有序退出）    nginx -s stop（快速退出）

## **Linux**

###### **常用命令：**

ifconfig 查看ip

ps -ef | grep [进程关键字] 查看进程 Windows查看端口号【netstat -ano|findstr 端口号】

kill -9 [PID] 杀死进程 windows 杀死进程【taskkill -f -pid 进程号】

cd：切换当前工作目录。语法为 cd [目录路径]     cd ~ 打开当前用户的根目录         cd / 打开根目录         cd .. 返回上一级目录         cd - 返回进入的目录

pwd 查看当前目录命令

ls：显示目录内容。语法为 ls [option] [目录路径]

                ls 查看当前目录下所有的文件

                ll 查看详细目录信息

mkdir 创建文件夹

touch 创建文件

rmkdir 删除文件

强制删除文件或目录：rm -rf 目录或者文件

在删除前询问是否确认删除：rm -ri 目录或文件

将文件夹1名称更改为文件夹2：mv 文件1 文件2 (给文件更名)

将目录1的文件移动到目录2：mv 目录2 目录2  (将目录1的文件移动到目录2)

递归复制目录1下的所有的文件和文件夹到目录2：cp -r [目录1](" \l ")

查看文件命令：cat 文件名

查看、编辑文件命令：vi

               进入命令模式：Esc

                         yy 复制一行

                        p 粘贴

                        u 回退

                         dd 删除一 行 （3dd删除三行）

                         ：w 保存

                        ：x 保存退出

                         ：wq 保存退出

                         ：q! 不保存退出

                         ：set nu 显示行数

               进入输入模式：a、i、o

tail -100f 文件名 查看文件最后100行的信息 （tail 尾部） ​head -100 文件名 查看文件前面100行信息

echo “文字” > 文件名 将文字写入文件

chown 修改文件的创建和组别

添加新用户命令：useradd -m 用户名

删除用户：userdel 用户名

修改用户密码命令：passwd 密码

切换用户命令：su 用户名

配置环境变量：

source刷新

解压 gzip 包：tar -zxvf [包名]



## **spring和springmvc和springboot和springcloud区别**

Spring、Spring MVC、Spring Boot 和 Spring Cloud 都是由 Spring Framework 提供的不同模块和扩展，用于开发企业级Java应用程序。它们之间的主要区别如下：

Spring：Spring 是一个轻量级的开源框架，提供了依赖注入（DI）和控制反转（IoC）等核心特性，用于构建企业级的Java应用程序。Spring 框架提供了许多模块，包括核心容器、数据访问、AOP、Web、消息传递和测试等。

Spring MVC：Spring MVC 是基于 Spring 框架的一个模块，用于开发 Web 应用程序。它提供了一个MVC（Model-View-Controller）模式的实现，用于处理请求、响应和视图渲染。Spring MVC 提供了丰富的功能，包括路由、数据绑定、验证、拦截器等，使开发人员能够构建灵活、可扩展的Web应用程序。

Spring Boot：Spring Boot 是 Spring Framework 的增强版，旨在简化和加速 Spring 应用程序的开发过程。它提供了自动配置、快速启动、约定优于配置等特性，使得开发者可以更快地搭建独立运行的、生产级别的 Spring 应用程序。Spring Boot 还集成了许多常用的第三方库和框架，提供了开箱即用的特性。

Spring Cloud：Spring Cloud 是一组用于构建分布式系统和微服务架构的工具和框架。它建立在 Spring Boot 的基础上，并提供了诸如服务注册与发现、负载均衡、断路器、配置管理等功能，以帮助开发人员构建可弹性扩展和高可用性的分布式应用程序。Spring Cloud 通过整合多个开源项目来实现这些功能，例如 Netflix Eureka、Ribbon、Hystrix 等。

总结起来，Spring 是一个全面的企业级应用程序开发框架，Spring MVC 是用于构建 Web 应用程序的模块，Spring Boot 是简化 Spring 应用程序开发的增强版，而 Spring Cloud 则是用于构建分布式系统和微服务架构的工具和框架。它们相互关联，可以根据需求选择使用。

## **Springboot**

##### **Spring Boot 特有的常用注解包括：**

@SpringBootApplication：标记启动类，用于表示应用的入口。

@Import(RestTemplate.class) ：用于配置类，RestTemplate等同于ssm中的httpClient。

@Value：用于注入配置参数的值。

@ComponentScan：用于指定项目扫描的包，默认加载启动类所在的包以及子包中的类。

@Mapper：标注在dao层接口。

@EnableDubbo：在启动类打，表示启动dubbo服务。

@DubboService：用于服务端标注该类为dubbo service层。

@DubboReference：用于消费端依赖注入接口实现类。

@ConfigurationProperties：将配置文件中的属性值与 Java 类型绑定起来。

@EnableAutoConfiguration：自动配置 Spring 应用上下文。

@ConditionalOnProperty：条件化配置注解，根据配置属性是否存在来决定是否加载一个 Bean 等组件。

##### **Spring Boot 的优点包括：**

快速启动：可以快速构建应用，减少了繁琐的配置过程。

自动配置：提供了许多自动配置，可以避免手工编写大量样板化的配置代码。

微服务开发：对单个微服务提供了良好的支持，而且与 Spring Cloud 框架无缝集成，方便进行微服务架构开发。

组件丰富：集成了大量优秀的组件和框架，如Spring Framework、Hibernate、MyBatis、JUnit等，方便开发者使用。

易于测试：提供了一些方便的测试工具和注解，使得单元测试、集成测试等变得容易实现。

易于部署：应用可以打包成可执行 JAR 或 WAR 文件，并采用内嵌 Servlet 容器进行运行，简化了部署流程。

社区活跃：的社区极其活跃，有大量的文档、示例和支持，方便开发者及时解决问题。

##### **Spring Boot 和 SSM框架的区别及优缺点如下：**

区别：

Spring Boot是目标在快速构建基于Spring框架的应用，提供一系列默认配置的框架；

SSM框架是三个独立项目的组合，并不提供像SpringBoot那样的快速启动和默认配置。

Spring Boot 优缺点：

优点：快速启动、自动配置、内置Tomcat等 Servlet 容器，便于开发、测试、部署等，

缺点：由于自动配置机制可能会导致理解框架原理不够充分，另外初学者可能需要花费些时间来学习 Spring Boot 的详细配置以及优化方式。

SSM框架的优缺点：

优点：常用的主流框架，在国内社区有很多资料和教程；

缺点：需要手动进行大量的 XML 配置，且相对于 Spring Boot 稍微有些落后；不具备如 Spring Boot 进行快速启动、自动配置的优势，部署时较为繁琐。

总体而言，如果项目简单、开发速度要求高等特点，可以选择使用 Spring Boot 来提高开发效率；如果项目规模较大，复杂度比较高，可以选择使用 SSM 等传统 Spring 生态系框架来满足需求。

##### **springboot 自动配置是什么？原理是**

Spring Boot的自动配置是一种方便的功能，它可以帮助开发者自动设置和配置项目。这种功能主要是依据classpath上的jar依赖，有条件地配置Spring应用程序。你添加什么库，Spring Boot就会增加相应的自动配置。例如，如果你的classpath上有H2和Spring Data JPA库，那么Spring Boot会自动配置内存数据库和设置JPA。

Spring Boot自动配置的基础是@SpringBootConfiguration，这是一种特殊类型的@Configuration。Spring Boot在启动时会去寻找并加载所有的@SpringBootConfiguration。

自动配置的核心机制在于 @EnableAutoConfiguration 注解，这个注解会启用 Spring Boot 的自动配置功能。@EnableAutoConfiguration 的作用是收集并加载所有在类路径下 META-INF/spring.factories 配置文件中的类。Spring Boot 会读取这些文件，并且查找 key 为 org.springframework.boot.autoconfigure.EnableAutoConfiguration 的项，然后把他们注册为bean。

自动配置尝试根据添加的jar依赖自动配置你的Spring应用。例如，如果你的classpath下有HSQLDB，而你并没有配置任何数据库连接的Beans，那么Spring Boot将自动配置一个内存（in-memory）数据库。

注意，自动配置是非侵入式的。自动配置不会在你定义了自己的配置的情况下尝试去改变。例如，如果你自定义了DataSource bean，那么默认的数据库配置就会失效。

也就是说，Spring Boot的自动配置是一种“开箱即用”的设计，能够帮助开发者省去大量的配置时间，提升开发效率。你只需要根据项目需要添加对应的依赖，然后Spring Boot就会为你自动完成大部分的基础配置。

##### **springboot自动配置和starters什么区别**

Spring Boot的自动配置和Starters是两个互补的特性，共同工作以提高应用程序的开发效率。这两个特性的主要区别在于它们的目标和方式。

自动配置（Autoconfiguration）：Spring Boot的自动配置是一种依赖于你的classpath和你定义的bean的功能。当Spring Boot在classpath上发现特定的库时，它会自动配置需要使用这些库的beans。例如，如果你的classpath下有Spring MVC的库，那么Spring Boot会自动配置一个DispatcherServlet和其他相关的MVC beans。如果你已经定义了自己的bean，那么Spring Boot不会覆盖你的设置。自动配置的目标是减少你必须编写的配置代码，使你能够更快速地开始编写应用程序的业务逻辑。

Starters：Starters是Spring Boot的一种依赖描述符，其设计目标是包含并管理Spring和相关技术的一系列依赖库，使得它们能够很好地一起工作，并且避免了版本冲突。例如，spring-boot-starter-web是一个用于构建web应用程序的starter，它包含了Spring MVC，Tomcat，Jackson（用于JSON处理）等相关的库。Starters的目标是简化你的构建配置，使你能够只关注需要的依赖，而不必关注它们的版本和互相之间的兼容性。

总的来说，自动配置和Starters都是Spring Boot的核心特性，它们共同工作，以帮助你更快、更容易地创建Spring应用程序。

## **SpringCloud**

Spring Cloud是基于Spring框架的微服务开发工具集，它提供了多个模块用于构建和部署云原生应用程序，包括服务发现、断路器、配置服务等。Spring Cloud与其他Spring技术紧密集成，使开发人员可以轻松地构建可扩展、容错的基于微服务的应用程序。

Spring Cloud是一个基于Spring平台的框架，它为开发人员提供了构建和部署云原生应用程序的工具。它包括各种模块，如服务发现、断路器、配置服务器等，以促进分布式系统的开发。Spring Cloud与其他Spring技术很好地集成，使开发人员能够轻松构建可伸缩、容错的基于微服务的应用程序。

###### **springcloud Alibaba有那些组件**

Spring Cloud Alibaba 是一个基于 Spring Cloud 体系的一站式企业级微服务解决方案，它是由阿里巴巴集团开源的。Spring Cloud Alibaba 项目包含以下组件：

Nacos（Naming and Configuration Service）：一个易于使用的服务发现、配置管理和服务管理平台。

Sentinel：一款高可用性的流量控制、熔断降级框架，可实现精细化的流量控制。

RocketMQ：一个分布式消息中间件，具有高可用、高性能、高扩展性等特点。

Seata：一个分布式事务解决方案，支持本地微服务和云原生场景的分布式事务处理。

Dubbo：阿里巴巴开源的高性能、轻量级 RPC 框架，支持多种序列化和通讯协议。

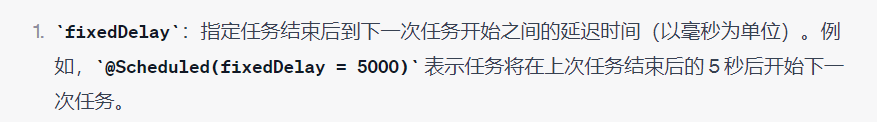
Alibaba Cloud ACM：一款针对云原生应用设计的配置管理平台，可实时同步配置、实现动态配置、故障恢复等。

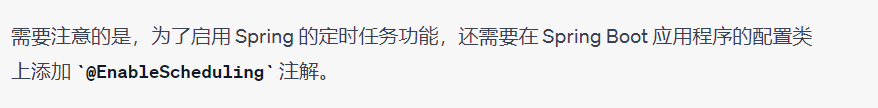
Alibaba Cloud OSS：阿里云对象存储服务，具有高扩展性、高可用性等特点，方便用户存储和管理海量数据。

Alibaba Cloud SMS：阿里云短信服务，提供全球短信发送和接收服务，可用于注册验证、营销推广、通知提醒等场景。

此外，Spring Cloud Alibaba 还提供对 Spring Cloud Gateway、Spring Cloud Alibaba Sentinel RSocket、Spring Cloud Alibaba Seata 的支持，并为用户提供与 Spring Boot、Spring Cloud 无缝集成的开发体验。

## **定时任务**





## **项目**

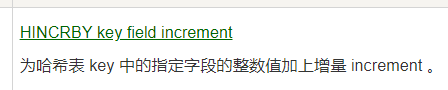
##### **1.商城购物车**

商城项目注意问题：

库存余额不足:(多用户同时购买时造成减库存的并发问题):sql语句加减库存用字段去加减添加where条件 库存>=购买数量和id。

表单重复提交： 立即购买返回随机字符串，并将随机字符串存入redis，随机字符串为redis string类型的key值，value值为1，提交订单时，获取随机字符串，删除redis中存储的随机字符串key，返回值为1，继续执行，若返回值为0，抛出异常终止程序。前端返回重复提交。

购物车的实现：redis hash类型 key值为用户id+固定的字符串，filed 为商品的编号，value为商品的数量，添加商品到购物车，先循环filed 查看商品是否已经存在购物车，如果存在改变商品数量（原来的+需要加入购物车的），如果不存在，添加商品id-key 数量-vlaue。incrBy



string 类型的 ： key 存储用户id+字符串，将商品对象放入ArrayList，转成json字符串，存入redis string（value），将json发送到前端。 使用先将josn转成ArrayList 遍历list。同上

网关回调失败： （补偿机制 最终一致性）

1、try catch起来， 失败之后 在catch里面重新发送，一直到成功。 （失败后会抛出异常）

2、本地消息事务：将失败记录写入日志中，记录里面包含参数和url路径，定时任务，从数据库读取失败的任务记录，将失败的任务记录重新发送，发送成功后删除任务，未成功一直发送。 (url json data 次数 调通后删除不通次数加一)

3、rabbit mq: 将失败的记录转成消息放到rabbit mq服务器中，通过rabbit mq发送消息，等到获取成功回应，回应即可，如果没有回应，mq会一直发送。

接口注意：

参数： 注意参数的格式，与前端约定好发送参数的类型，

返回值：与前端约定好发送参数的类型，并告诉前端返回值代表的意思。

接口幂等性（多次调用接口的结果要一致）：

提交物流表单案例：

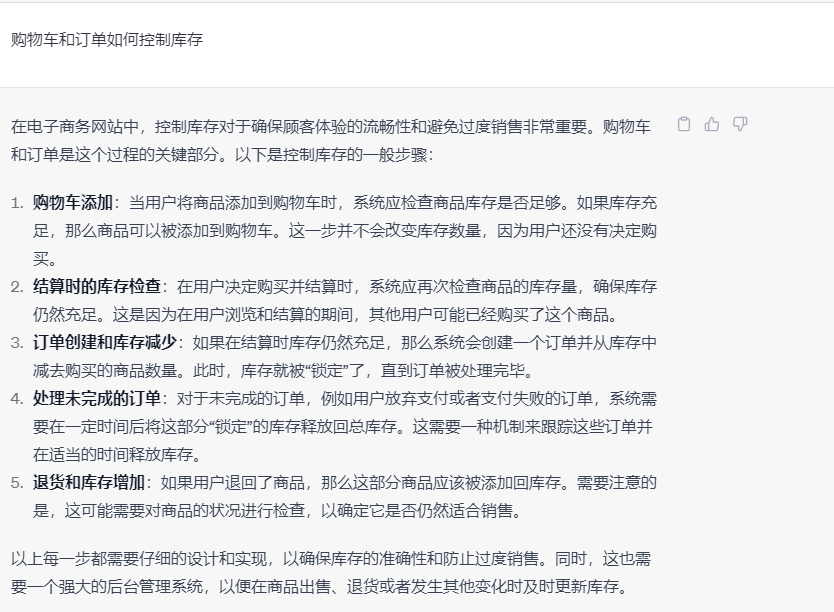
先进行订单状态的修改，在调用物流接口，修改状态的时候需要查看返回值，返回值为1，提交物流表单，继续执行程序，返回 data code msg ； 返回值为0 抛出异常，不执行物流。修改状态语句update 表名 set 状态=新状态 where 订单编号=订单编号 and 状态 = 旧状态

统一异常：

后端报错后同样需要返回msg，不能将tomcat的错误页面返回。

充值：同时对一个账户充值

rabbit mq，本地消息事务



##### **2.网关**

1.支付宝接口 两个 接收的参数写在对象里面

私钥签名-->表明是谁发的 公钥解密

用签名不用加密是为了整明是谁发送的不是为了保护数据 ，数据可以允许别人看到

放在treemap里面 -->自动排序 因为同一份数据按照不同顺序排列的签名和验签时的结果是不一样的

2.微信

1.支付流程

登录获取openid

支付后台传

##### **3.充值**

密码一定是加密的

1.拦截是否登录

返回code 跳转页面

2.充值余额

3..回调

回调显示的余额是充值后余额 = 上次充值后余额+当前充值金额

充值后余额同步到账户表的余额

订单状态 根据订单状态的判断可解决支付宝重复调用的问题

4.一个账号在2-3的客户端登录

5.充值后明细 显示余额

加锁 修改订单状态 -->修改账户余额 -->修改订单余额 三个update在一个事务里 完成后进行提交

##### **4.单点登录**

1.sso 拦截 统一异常 验签 登录

2.逻辑

SsoA -->拦截器先判断有没有cookie 如果有就可以进入A自己的系统 如果没有去判断有没有token 如果有就带token去sso验签，验签通过后给一个A自己的cookie，成功登录系统，如果没有就去sso登录

Sso服务器-->拦截器先判断有没有cookie如果有就验证cookie后生成一个token回传，如果没有cookie就去登录，登录验证后生成自己的cookie证明登录过 ， 后生成一个token给系统回传

##### **5.权限管理**

菜单表 二级菜单有动作url



对角色进行增删改查

## **多线程**

##### **实现线程的三种方式**

继承Thread类：创建一个新的类并继承Thread类，重写run()方法来定义需要执行的任务，然后通过创建该类的实例对象来启动线程。

实现Runnable接口：创建一个实现Runnable接口的类，并实现其run()方法，通过创建该类的实例对象，并将其作为Thread类的构造方法参数来启动线程。

实现Callable接口：创建一个实现Callable接口的类，并实现其call()方法，可以返回执行结果。然后通过创建该类的实例对象，并将其作为FutureTask类的构造方法参数，再创建Thread类来启动线程。

##### **线程池的3种创建方式**

/\*

<dependency>

<groupId>org.apache.commons</groupId>

<artifactId>commons-lang3</artifactId>

<version>3.12.0</version>

</dependency>

\*/

//创建固定大小的线程池

ExecutorService threadPool = Executors.newFixedThreadPool(3);

//创建缓存线程池(根据任务数量去创建线程)

ExecutorService threadPool = Executors.newCachedThreadPool();

//创建单一线程池（如果线程结束，可以创建一个新的）

ExecutorService threadPool = Executors.newSingleThreadExecutor();

threadPool.execute(new Runnable());

threadPool.shutdown();//没有任务时，关闭线程池



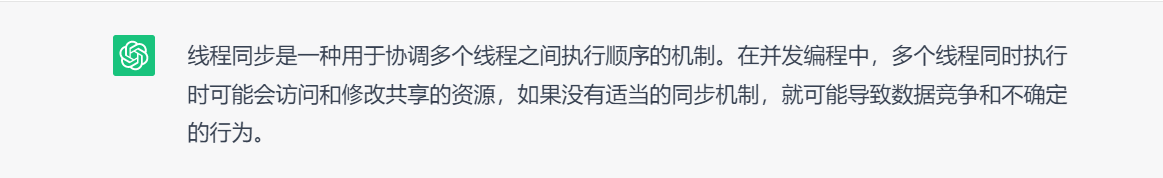
<https://segmentfault.com/a/1190000041517823> 线程的7个参1.

CAS简介 自旋 比较并交换(compare and swap, CAS)，是原子操作的一种。在多线程没有锁的状态下，可以保证多个线程对同一个值的更新。

CAS可用于在多线程编程中实现不被打断的数据交换操作，从而避免多线程同时改写某一数据时由于执行顺序不确定性以及中断的不可预知性，产生的数据不一致问题。该操作通过将内存中的值与指定数据进行比较，当数值一样时将内存中的数据替换为新的值。

## **多线程同步工具**

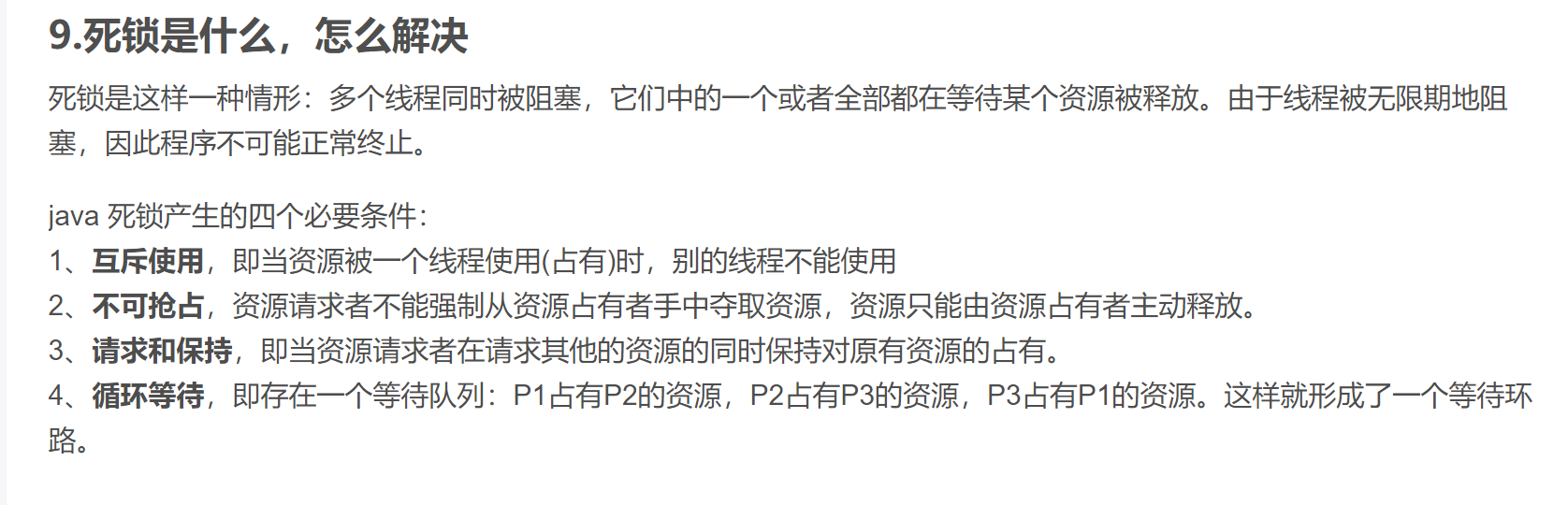
#### **什么是线程同步？**



同步代码块和notify和await区别

#### **redis的死锁：消费者和生产者**

死锁：一组互相竞争的资源的线程之间相互等待，导致永久阻塞的现象



#### **Semaphore信号灯**

允许线程最大访问量，如果超出最大访问量，其它线程要等待

Semaphore semap = new Semaphore(3);//最大访问量为3

semap.acquire();//获取信号灯

semap.release();//释放信号灯

#### **Countdownlatch**

简单的说就是线程等待，用来协调线程的同步，当所有线程执行到就count-1，全部集合之后再继续执行。一个典型应用场景就是启动一个服务时，主线程需要等待多个组件加载完毕，之后再继续执行。

//所有运动员准备好，等待裁判发出起跑命令

CountDownLatch latch1 = new CountDownLatch(1);

CountDownLatch latch2 = new CountDownLatch(3);

latch1.await();

latch2.countDown();//计数器减1

在 Java 中，主线程可以通过不同的方式将值传递给子线程。以下是几种常见的方法：

构造函数传参： 子线程的类可以定义带有参数的构造函数，在创建子线程时，通过构造函数将值传递给子线程对象。

示例代码：

class MyThread extends Thread {

private int value;

public MyThread(int value) {

this.value = value;

}

public void run() {

// 在子线程中使用传递的值

System.out.println("Received value in thread: " + value);

}

}

public class Main {

public static void main(String[] args) {

int valueToSend = 42;

MyThread thread = new MyThread(valueToSend);

thread.start();

}

}

setter 方法： 子线程的类可以定义一个公共的 setter 方法，主线程可以通过该方法将值传递给子线程。

示例代码：

class MyThread extends Thread {

private int value;

public void setValue(int value) {

this.value = value;

}

public void run() {

// 在子线程中使用传递的值

System.out.println("Received value in thread: " + value);

}

}

public class Main {

public static void main(String[] args) {

int valueToSend = 42;

MyThread thread = new MyThread();

thread.setValue(valueToSend);

thread.start();

}

}

共享变量： 主线程和子线程可以共享一个变量，主线程将值设置给该变量，子线程可以读取该变量。

示例代码：

class MyThread extends Thread {

private int value;

public void run() {

// 在子线程中使用共享的值

System.out.println("Received value in thread: " + value);

}

}

public class Main {

public static void main(String[] args) {

MyThread thread = new MyThread();

thread.value = 42; // 在主线程中设置共享变量的值

thread.start();

}

}

无论哪种方式，需要注意的是在线程之间共享值时，要确保线程安全性，以避免潜在的并发问题。可以使用同步机制（如锁）或使用线程安全的数据结构来保护共享资源的访问。

#### **Cyclicbarrie**

是所有线程都进行等待，直到所有线程都准备好进入await()方法之后，执行条件变量的signalAll方法唤醒等待的线程。所有线程同时开始执行！

CyclicBarrier barrier = new CyclicBarrier(5);

barrier.await();

#### **读写锁 ReentrantReadWriteLock**

读锁和读锁不互斥，写锁和写锁互斥，写锁和读锁也互斥

ReadWriteLock rwl = new ReentrantReadWriteLock();

rwl.readLock().lock();//上读锁

rwl.readLock().unlock();//释放读锁，在finally块里释放锁

rwl.writeLock().lock();//上写锁

rwl.writeLock().unlock();//释放写锁

#### **线程之间的通信Condition**

就是线程1执行10次，换另一个线程执行5次，实现线程之间的通信

ReentrantLock lock = new ReentrantLock();

Condition condition1 = lock.newCondition();//在锁的基础上创建条件

lock.lock();//先上锁

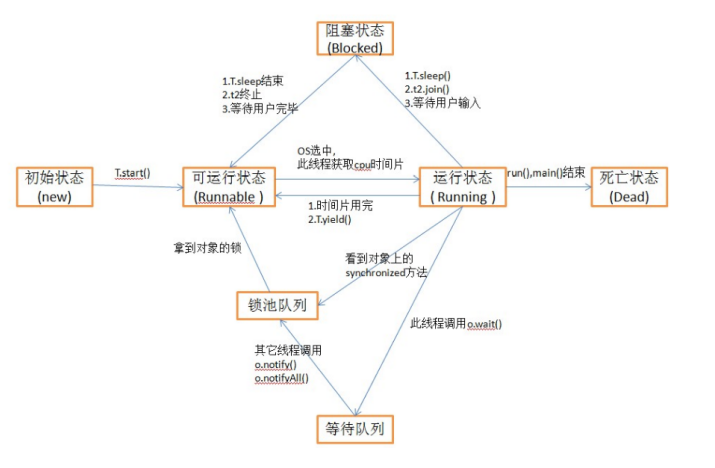
condition1.await();//线程1等待执行

condition2.signal();//通知线程2执行完毕

### 线程的生命周期

5个阶段：1、新建，是刚使用new方法，new出来的线程；2、就绪，是调用的线程的start()方法后，这时候线程处于等待CPU分配资源阶段；3、运行，当就绪的线程被调度并获得CPU资源时，便进入运行状态；4、阻塞，在运行状态的时候，可能因为某些原因导致运行状态的线程变成了阻塞状态；5、销毁，线程正常执行完毕后或线程被提前强制性的终止或出现异常导致结束，那么线程就要被销毁。

共5个（新建、就绪、运行、阻塞、死亡）



### 线程安全问题

#### **多线程访问同一份成员代码，如果执行逻辑是对就是线程安全问题。**

当多个线程访问某个方法时，不管你通过怎样的调用方式、或者说这些线程如何交替地执行，我们在主程序中不需要去做任何的同步，这个类的结果行为都是我们设想的正确行为，那么我们就可以说这个类是线程安全的。

##### **线程同步**

线程同步是多线程编程中的一个概念，用于确保两个或多个并发线程不同时访问某一共享资源（如共享变量、数组或者文件等）。线程同步可以解决由于线程的并发执行而引发的问题，例如数据不一致、数据丢失等。

线程同步的主要目的是防止同时访问共享资源，导致数据混乱的情况。为了实现这个目标，需要在访问共享资源的代码前后加上同步控制，如锁等机制，以确保任一时刻只有一个线程在访问资源。

在Java中，线程同步可以通过以下几种方式实现：

Synchronized 关键字：可以修饰方法或者代码块。当它修饰一个方法或一个代码块时，能够保证在同一时刻最多只有一个线程执行该段代码。

ReentrantLock 类：Java提供的另一种锁机制，相比于synchronized，ReentrantLock类提供了一些高级功能，比如具备可重入性、中断响应的加锁、公平和非公平锁模式、读写分离锁等。

Volatile 关键字： 它能确保将修改的值立即同步到主内存，以及每次使用前立即从主内存刷新。

Atomic 类：在Java中，原子操作是不能被线程调度机制打断的操作；也就是说，当一个线程开始执行原子操作后，如果线程在进行原子操作期间被调度程序打断，那么该操作依然能够继续完成。

CountDownLatch、CyclicBarrier、Semaphore：这些都是高级的线程同步工具类，可以帮助开发者更好的控制线程的同步。

这些都是实现线程同步的方法，不同的方法有其适用的场景，需要根据具体的需求来选取合适的同步方法。

##### **线程和进程的区别**

进程（Process）和线程（Thread）是操作系统中的两个重要概念，它们具有以下区别：

定义：进程是执行中的程序实例，具有独立的内存空间和系统资源；线程是进程的执行单位，是进程内的一个独立执行流程。

资源占用：每个进程都有独立的内存空间和系统资源（如文件句柄、打开的网络连接等），而线程共享所属进程的资源。

调度和切换：进程是系统进行调度和分配资源的基本单位，进程间的切换代价较高；线程是在进程内部切换执行的，线程间的切换代价较低。

并发性：在多核处理器上，多个进程可以并发执行，每个进程可以利用多个线程实现并发执行。

通信与同步：进程间通信需要使用特定的机制，如管道、消息队列、共享内存等；线程之间可以通过共享内存直接进行通信和数据共享，但需要考虑同步和互斥问题。

稳定性：一个进程的异常或错误通常不会影响其他进程的正常执行；但一个线程的异常或错误可能会导致整个进程崩溃。

创建和销毁：创建和销毁进程的开销较大；创建和销毁线程的开销相对较小。

总的来说，进程是资源分配和调度的基本单位，线程是进程内部并发执行的基本单位。进程之间相互独立，而线程共享进程的资源。在设计和开发中，需要根据具体的需求和场景合理选择使用进程或线程，以达到最佳的性能和资源利用。

##### **sleep和wait区别**

sleep是来自于Thread类，wiat是来自于Object类

线程sleep后，不会释放锁，进入阻塞队列。线程wait后释放了锁，进入等待队列。

sleep可以在任何地方使用，wait只能在同步代码里使用

##### **synchronized和Lock的异同**

相同点：Lock能完成synchronized所实现的所有功能；

不同点：

Syschronized是一个关键字，而Lock是一个类；

Lock有比synchronized更精确的线程语义和更好的性能，而且不强制性的要求一定要获得锁；

Syschronized同步数据少量的话，性能比Lock好，而数据大量同步，Lock性能要好

synchronized会自动释放锁，而Lock一定要求程序员手工释放，并且最好在finally块中释放（这是释放外部资源的最好的地方）。

##### **多线程的状态**

多线程在执行过程中会经历不同的状态，常见的多线程状态包括：

新建状态（New）：线程被创建后尚未启动的状态。

运行状态（Runnable）：线程正在执行任务的状态。处于运行状态的线程可能正在执行、等待系统资源（如 CPU 资源），或者等待其他线程的通知。

阻塞状态（Blocked）：线程因为某些原因暂时停止执行的状态。当线程等待某个条件满足时（如等待锁的释放），它将进入阻塞状态。

等待状态（Waiting）：线程因为某些原因无限期等待的状态。当线程调用了 wait() 方法、join() 方法或 LockSupport.park() 方法时，它将进入等待状态。

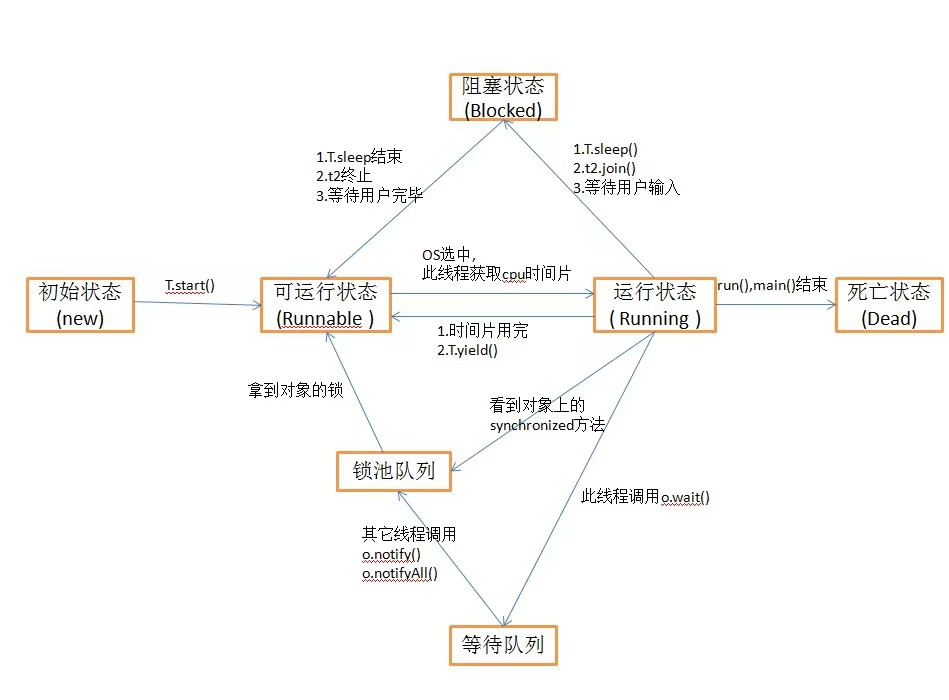
计时等待状态（Timed Waiting）：线程因为某些原因有一定时间限制的等待状态。当线程调用了带有超时参数的 sleep() 方法、wait(timeout) 方法、join(timeout) 方法或 LockSupport.parkNanos() 方法时，它将进入计时等待状态。

终止状态（Terminated）：线程执行完任务或因异常退出的状态。

这些状态在多线程编程中是相互转换的，线程的状态会随着不同的操作而发生改变。例如，新建的线程通过调用 start() 方法进入运行状态，运行中的线程可以被阻塞或进入等待状态，最终线程执行完任务或异常退出进入终止状态。

#### **sycho进入锁池队列，碰到wait进入等待队列，notify是进入锁池对象**

##### **锁时队列和等待队列**



## **nacos**

如果你想配置服务名、端口和分组等信息，你可以在Spring Cloud项目的配置文件中添加相应的配置项。下面是一个示例：

spring:

application:

name: your-service-name

server:

port: 8080

spring:

cloud:

nacos:

discovery:

server-addr: localhost:8848

service: ${spring.application.name}

port: ${server.port}

group: your-group

在上面的示例中，我们做了以下配置：

spring.application.name：指定了服务名为 your-service-name。该值将被用于服务注册和发现。

server.port：指定了服务的端口为 8080。

spring.cloud.nacos.discovery.server-addr：指定了Nacos服务器地址为 localhost:8848。

spring.cloud.nacos.discovery.service：使用${spring.application.name}将服务名动态地设置为与spring.application.name相同的值。

spring.cloud.nacos.discovery.port：使用${server.port}将端口动态地设置为与server.port相同的值。

spring.cloud.nacos.discovery.group：指定了服务所属的分组为 your-group。这是可选的配置项。

## **Zookeeper**

Zookeeper是一个分布式协调服务，用于在分布式系统中管理和协调配置信息、命名服务、分布式锁和分布式应用程序的数据共享。

Zookeeper提供了一个分层的命名空间，类似于文件系统，其中每个节点都可以包含数据，每个节点都可以有多个子节点。而且节点的数据是一个字节数组，可以存储任意类型的数据。

当Zookeeper节点数据发生变化时，观察者可以得到通知。这种观察者模式在分布式应用程序中，当一个节点上的数据发生变化时，所有依赖该节点数据的应用程序都会收到通知。

Zookeeper还提供了分布式锁服务，可以用于在分布式环境中同步访问共享资源。它使用基于序列的节点名称和事件监听来实现同步和竞争控制，解决在分布式系统中的竞争条件问题。

Zookeeper是一个高可用性的系统，它使用集群来提供高可用性和可扩展性。当Zookeeper集群中的一个节点发生故障时，它会自动重新选举新的领导者，并继续提供服务，这使得Zookeeper集群可以容忍节点故障并保持可用性。

#### **zookeeper常用命令**

启动 ZK 服务: sh bin/zkServer.sh start

查看 ZK 服务状态: sh bin/zkServer.sh status

停止 ZK 服务: sh bin/zkServer.sh stop

重启 ZK 服务: sh bin/zkServer.sh restart

连接 ZK 服务：/bin/zkCli.sh -server ip:端口

ls命令：查看目录下的节点信息

create命令：创建节点，默认持久节点

set命令：修改节点内容

get命令：获取节点/文件内容

delete命令：只能删除没有子节点的节点。如果其有子节点时，无法删除

deleteall命令：级联删除该节点和子节点。

#### **面试题**

你对Zookeeper有什么了解？它的主要功能是什么？

答：Zookeeper是一个分布式的开源协调服务，用于构建可靠的分布式系统。它提供了分布式数据管理和协调的功能，包括统一命名空间、数据发布/订阅、分布式锁、分布式队列等。主要功能包括配置管理、命名服务、分布式锁、分布式队列等。

在Java应用程序中，你如何使用Zookeeper来实现分布式锁？

答：使用Zookeeper实现分布式锁的一种常见方法是使用临时顺序节点。每个客户端在Zookeeper上创建一个临时顺序节点，表示它想要获取锁。当某个客户端的节点成为最小的节点时，它获得锁。其他客户端可以通过监视前一个节点的方式等待释放锁的通知。

你在项目中使用过Zookeeper来解决哪些具体的问题？请分享一下你的经验。

分布式配置管理：我们的应用程序有多个实例运行在不同的节点上，需要统一管理配置信息。我们使用Zookeeper来存储和管理应用程序的配置数据。每个应用程序实例都可以监听Zookeeper上配置节点的变化，并及时获取最新的配置信息，从而实现动态配置的更新和同步。

分布式锁：我们的系统中存在一些需要保证互斥访问的临界资源。通过使用Zookeeper的临时顺序节点，我们实现了一个分布式锁机制。每个应用程序实例在需要访问临界资源时创建一个临时顺序节点，然后检查自己是否是最小的节点，如果是，则获得锁，否则等待。这样确保了只有一个实例可以同时访问临界资源。

在Zookeeper中，你如何处理会话过期的情况？可以提供一些处理过期会话的方法吗？

答：当会话过期时，Zookeeper会发送会话过期事件给客户端。在Java应用程序中，我们可以通过实现Watcher接口并在会话过期事件中执行相应的操作来处理会话过期。一种常见的处理方法是重新连接到Zookeeper集群，并根据需要重新注册Watcher和恢复会话状态。

## **Seata**

如何配置Seata？ Seata的配置主要包括注册中心配置、事务组配置、数据源代理配置等。你需要在Seata的registry.conf、file.conf和registry.conf文件中进行相应的配置。其中，registry.conf用于配置注册中心，file.conf用于配置事务组，nacos.conf用于配置数据源代理。你可以根据具体的需求和使用的注册中心、事务组和数据源代理进行相应的配置。

如何在Spring Boot项目中使用Seata？ 在Spring Boot项目中使用Seata需要进行以下步骤：

添加Seata的依赖。

registry.conf：注册中心配置文件，用于配置Seata的注册中心信息。示例内容如下：

配置Seata的相关配置文件（registry.conf、file.conf和registry.conf）。

在需要进行分布式事务管理的方法上添加@GlobalTransactional注解。

启动Seata服务端（TC）和相应的资源管理器（RM）。

启动Spring Boot应用程序。

什么是Seata？

Seata（简称为分布式事务解决方案）是一个开源的分布式事务解决方案，旨在解决分布式环境下的数据一致性问题。它提供了分布式事务管理、数据源代理、事务协调和分布式锁等功能，使得分布式系统中的多个服务可以在保持数据一致性的前提下进行事务操作。

registry.conf：注册中心配置文件，用于配置Seata的注册中心信息。示例内容如下

registry {

type = "nacos"

nacos {

serverAddr = "localhost:8848"

namespace = "seata\_namespace"

group = "seata\_group"

}

}

file.conf：文件配置文件，用于配置Seata的存储方式和事务日志存储位置。示例内容如下：

service {

vgroupMapping.my\_group = "default"

default.grouplist = "127.0.0.1:8091"

enableDegrade = false

disableGlobalTransaction = false

}

store {

mode = "db"

db {

driverClassName = "com.mysql.jdbc.Driver"

url = "jdbc:mysql://localhost:3306/seata?useUnicode=true"

user = "seata"

password = "seata"

}

}

## **sentinel**

什么是Sentinel？ Sentinel是阿里巴巴开源的一款分布式系统的流量防卫兵，用于保护微服务架构中的稳定性和可靠性。它提供实时监控、流量控制、熔断降级、系统负载保护等功能，帮助开发者构建健壮的分布式系统。

Sentinel的主要特性是什么？ Sentinel具有以下主要特性：

流量控制：通过设置QPS阈值和并发线程数限制，保护系统免受过载的影响。

熔断降级：当系统中的某个资源出现异常或超出阈值时，自动触发熔断策略，避免级联故障。

系统负载保护：通过设置系统的全局QPS阈值，保护整个系统的稳定性。

实时监控：提供实时的运行指标和统计信息，可通过仪表盘进行可视化监控。

规则配置：支持动态配置流控规则和熔断规则，可以根据业务需求进行灵活调整。

Sentinel支持哪些流控规则？ Sentinel支持以下常见的流控规则：

直接关联：基于具体的资源进行流控，例如接口、方法等。

关联关系：基于资源之间的关联关系进行流控，例如关联资源A的流量对资源B进行限制。

参数限流：基于请求参数进行流控，例如对某个接口的某个参数进行限制。

热点参数限流：基于热点参数进行流控，例如对某个接口的某个参数的热点值进行限制。

如何在Spring Cloud项目中使用Sentinel？ 在Spring Cloud项目中使用Sentinel需要进行以下步骤：

添加Sentinel的依赖。

配置Sentinel的相关配置文件（例如sentinel.properties）。

# 配置 Sentinel 控制台地址

spring.cloud.sentinel.transport.dashboard=localhost:8080

# 配置应用程序名称

spring.application.name=your-application-name

# 配置 Sentinel 日志级别

logging.level.com.alibaba.csp.sentinel=DEBUG

# 配置 Sentinel 限流异常处理页面

# 当限流发生时，返回自定义的错误信息页面

spring.mvc.throw-exception-if-no-handler-found=true

spring.resources.add-mappings=false

error.whitelabel.enabled=false

# 配置其他 Spring Boot 属性

# 这里可以添加其他的 Spring Boot 配置项，例如数据库连接、服务器端口等

在需要进行流控或熔断降级的方法上添加@SentinelResource注解，配置相应的规则和处理逻辑。

启动Spring Cloud应用程序，并启动Sentinel控制台进行监控和管理。

Sentinel的热点参数限流是如何工作的？ Sentinel的热点参数限流基于统计信息和滑动时间窗口来实现。它通过统计请求参数的QPS、线程数等指标，对参数的热点值进行动态的限流。可以设置阈值、统计时长和滑动窗口的大小来调整热点参数限流的粒度和准确性。

## **gateway**

Spring Cloud Gateway的核心概念是什么？ Spring Cloud Gateway的核心概念包括：

Spring Cloud Gateway 是一个基于 Spring Boot 的开源网关解决方案，用于构建微服务架构中的 API 网关。它提供了一种简单而强大的方式来路由、过滤和转换传入的请求，并与后端的多个微服务进行交互。

路由（Route）：定义了请求的匹配规则、目标URI和过滤器链等配置。

断言（Predicate）：用于匹配请求的条件，例如路径匹配、Header匹配等。

过滤器（Filter）：用于对请求和响应进行处理，例如认证、鉴权、限流等。

过滤器工厂（Filter Factory）：用于创建过滤器实例的工厂。

如何配置Spring Cloud Gateway？ 在Spring Cloud Gateway中，你可以使用配置文件或编程方式进行配置。常见的配置方式包括：

使用application.yml或application.properties文件配置路由、断言和过滤器等。

# 配置应用程序名称

spring:

application:

name: your-application-name

# 配置应用程序的端口号

server:

port: 8080

# 配置示例的路由规则

spring:

cloud:

gateway:

routes:

- id: example-route

uri: http://example.com

predicates:

- Path=/example/\*\*

filters:

- RewritePath=/example/(?<segment>.\*), /${segment}

Spring Cloud Gateway如何处理限流和熔断？ Spring Cloud Gateway可以与限流和熔断框架（例如Sentinel）集成，通过配置相应的过滤器和断言来实现限流和熔断的功能。你可以使用全局过滤器或自定义过滤器来拦截请求，进行限流和熔断处理。

## **Dubbo**

###### **dubbo原理：**

dubbo是一个高性能轻量级rpc（远程方法调用）框架，核心能力：面向接口的远程方法调用，智能容错，负载均衡，服务的自动注册和发现

Dubbo 是阿里巴巴开源的一个高性能、轻量级的分布式服务治理框架，其核心原理可以简单概括为：

服务提供者在启动时会向注册中心注册自己所提供的服务，包括服务接口名称、版本号等元数据信息。

服务消费者在启动时会从注册中心获取所需服务的元数据信息，并通过客户端代理类将服务请求发送至服务提供者。

Dubbo 的核心通信模型是基于长连接的 NIO 通信，通信协议支持多种形式，如Dubbo协议、HTTP 协议、Hessian 协议等。

服务提供者和服务消费者之间通过 URL 进行匹配，确定调用的服务实例。

Dubbo 支持多种集群容错策略，包括 failover（失败重试）、failfast（快速失败）、failsafe（失败安全）等。

Dubbo 支持多种负载均衡策略，包括 Random（随机调用）、RoundRobin（轮询调用）、LeastActive（并发数最小调用）等。

Dubbo 支持多种服务路由策略，包括基于权重、基于标签、基于 IP 地址等方式。

Dubbo 支持对服务调用过程进行统一的监控和管理，包括服务调用次数、调用延迟、错误次数等指标的收集和报警。

Dubbo 的核心概念是：服务提供者、服务消费者、注册中心，以及代理对象、URL、扩展点等，这些概念和构成 Dubbo 框架的各种组件相互协作实现了 Dubbo 的分布式服务治理能力。

###### **流程：**

Dubbo 的执行流程如下：

服务提供者启动，向注册中心注册自己提供的服务。

服务消费者启动，向注册中心订阅需要消费的服务。

消费者从注册中心获取到可用的服务地址列表，并根据负载均衡策略选择一个提供者。

消费者通过网络请求选择的提供者，并将参数传递给提供者。

提供者接收到请求后处理业务逻辑并返回结果。

消费者接收到提供者的响应，进行相应的处理。

1、服务提供者启动，创建Zookeeper客户端，向注册中心注册服务；

2、服务消费者启动，通过Zookeeper向注册中心获取服务提供者列表；

3、服务消费者通过接口开始远程调用服务，ProxyFactory通过初始化Proxy对象，Proxy通过创建动态代理对象；

4、动态代理对象通过invoke方法，层层包装生成一个Invoker对象，该对象包含了代理对象；

5、Invoker通过路由，负载均衡选择了一个最合适的服务提供者，在通过加入各种过滤器，协议层包装生成一个新的DubboInvoker对象；

6、再通过交换成将DubboInvoker对象包装成一个Reuqest对象，该对象通过序列化通过NettyClient传输到服务提供者的NettyServer端；

7、到了服务提供者这边，再通过反序列化、协议解密等操作生成一个DubboExporter对象,再层层传递处理,会生成一个服务提供端的Invoker对象；

8、这个Invoker对象会调用本地服务，获得结果再通过层层回调返回到服务消费者，服务消费者拿到结果后，再解析获得最终结果。ssm配置（服务端，客户端）

## **Rabbitmq**

##### **消息队列概述：**

消息队列中间件是分布式系统中重要的组件，主要解决应用解耦，异步消息，流量削锋等问题，实现高性能，高可用，可伸缩和最终一致性架构。目前使用较多的消息队列有ActiveMQ，RabbitMQ，ZeroMQ，Kafka，MetaMQ，RocketMQ

##### **消息队列应用场景：**

异步处理

场景说明：用户注册后，需要发注册邮件和注册短信。传统的做法有两种 1.串行的方式；2.并行方式

串行方式：将注册信息写入数据库成功后，发送注册邮件，再发送注册短信。以上三个任务全部完成后，返回给客户端。

并行方式：将注册信息写入数据库成功后，发送注册邮件的同时，发送注册短信。以上三个任务完成后，返回给客户端。与串行的差别是，并行的方式可以提高处理的时间。

应用解耦

场景说明：用户下单后，订单系统需要通知库存系统。传统的做法是，订单系统调用库存系统的接口。

传统模式的缺点：假如库存系统无法访问，则订单减库存将失败，从而导致订单失败，订单系统与库存系统耦合。

订单系统：用户下单后，订单系统完成持久化处理，将消息写入消息队列，返回用户订单下单成功

库存系统：订阅下单的消息，采用拉/推的方式，获取下单信息，库存系统根据下单信息，进行库存操作

假如：在下单时库存系统不能正常使用。也不影响正常下单，因为下单后，订单系统写入消息队列就不再关心其他的后续操作了。实现订单系统与库存系统的应用解耦。

流量削峰

应用场景：秒杀活动或团抢活动中，一般会因为流量过大，导致流量暴增，应用挂掉。为解决这个问题，一般需要在应用前端加入消息队列。

a、可以控制活动的人数

b、可以缓解短时间内高流量压垮应用

用户的请求，服务器接收后，首先写入消息队列。假如消息队列长度超过最大数量，则直接抛弃用户请求或跳转到错误页面。秒杀业务根据消息队列中的请求信息，再做后续处理。

日志处理

日志处理是指将消息队列用在日志处理中，比如Kafka的应用，解决大量日志传输的问题。

日志采集客户端，负责日志数据采集，定时写受写入Kafka队列；

Kafka消息队列，负责日志数据的接收，存储和转发；

日志处理应用：订阅并消费kafka队列中的日志数据

消息通讯

消息通讯是指，消息队列一般都内置了高效的通信机制，因此也可以用在纯的消息通讯。比如实现点对点消息队列，或者聊天室等。

##### **rabbitmq五种消息模型**

Simple-简单模型

生产者producer将消息发送到队列，消费者consumer从队列中获取消息，队列是存储消息的缓冲区。

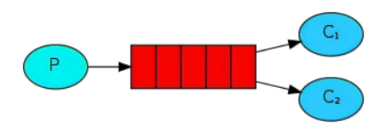


Work-工作模型

将任务封装为消息并将其发送到队列。 在后台运行的工作进程将获取任务并最终执行作业。

当你运行许多消费者时，任务将在他们之间共享，但是一个消息只能被一个消费者获取。

（发送者将消息发送到队列中，接收者从队列中接收消息，每个消息只能被一个接收者消费。）



Fanout-广播模型

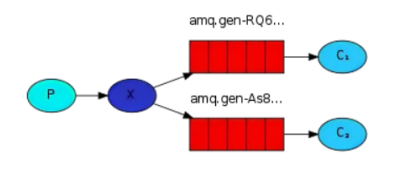
（发送者将消息发送到交换机上，交换机将消息广播给绑定在它上面的所有队列，每个队列都会收到一份拷贝。）

可以有多个消费者，每个消费者有自己的queue（队列）

每个队列都要绑定到Exchange（交换机）

生产者发送的消息，只能发送到交换机，交换机来决定要发给哪个队列，生产者无法决定。

队列的消费者都能拿到消息。实现一条消息被多个消费者消费

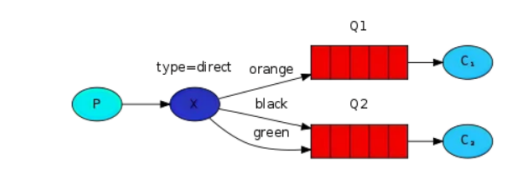


Direct-定向模型

（发送者指定消息的路由键，交换机根据路由键将消息分发到对应的队列，每个队列只会收到符合其路由键的消息。）

在Direct模型下，队列与交换机的绑定，不能是任意绑定了，而是要指定一个RoutingKey（路由key）；

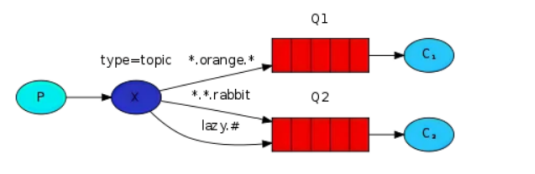
消息的发送方在向Exchange发送消息时，也必须指定消息的routing key



Topic-主题模型

Topic类型的Exchange与Direct相比，都是可以根据RoutingKey把消息路由到不同的队列。只不过Topic类型Exchange可以让队列在绑定Routing key 的时候使用通配符！（类似于路由模型，但是路由键可以使用通配符进行匹配，支持更灵活的消息路由。）

\*（星号）可以代替一个词。#（哈希）可以替换零个或多个单词。



在使用消息队列（MQ）进行消息传递时，确保消息的幂等性是防止消息重复消费的一种常见方法。幂等性是指无论执行多少次相同的操作，最终的结果都是一致的。

##### **下面是一些可以帮助您防止MQ消息重复消费的方法：**

消息去重：在消费者端，可以维护一个已经处理过的消息的记录，可以是一个数据库表、缓存或者其他存储方式。在消费消息之前，先查询这个记录来检查消息是否已经处理过，如果已经处理，则忽略该消息。

唯一标识符：为每条消息分配一个唯一标识符，在消费者端对接收到的消息进行标识，以便检测和过滤重复消息。可以使用消息中的某个字段，比如消息ID或者自定义的唯一标识符。

幂等处理：设计消息消费逻辑时，要确保消费操作是幂等的，即无论执行多少次，最终结果都是相同的。这样即使重复消费了同一条消息，也不会对系统产生副作用。例如，如果消费操作是写入数据库，可以使用数据库的唯一键或者乐观锁来确保重复写入不会产生错误。

消息确认机制：在某些MQ系统中，消费者在处理完消息后需要发送确认（acknowledgement）给MQ，告知消息已经成功处理。如果消费者在处理过程中发生异常或者失败，可以选择不发送确认，让MQ将消息重新投递。这样可以确保消息不会在处理失败的情况下被标记为已处理，避免重复消费。

消息过期时间：在消息发送时可以设置一个过期时间，确保消息在一定时间内必须被处理，否则过期的消息可以被丢弃或者进行特殊处理。这样可以避免长时间未被处理的消息重复消费。

保证消息可靠性：使用可靠性投递机制，例如使用消息队列的事务或者确认模式，确保消息在传递过程中不会丢失。如果消息队列支持重试机制，可以设置重试次数和间隔，以便在处理失败时进行重试，而不是重新消费同一条消息。

综合使用以上方法，可以有效地防止消息队列中的消息重复消费。根据具体的消息队列系统和业务场景，您可以选择适合的方法或者结合多种方法来保证消息的幂等性和可靠性。

##### **在面试中，关于消息队列（MQ）的问题可能涉及以下方面：**

当涉及RabbitMQ的面试问题时，以下是一些可能会被问到的问题以及简要的回答：

什么是消息队列？消息队列的作用是什么？

消息队列是一种在应用程序之间传递消息的通信方式。它通过将消息发送到中间代理（消息队列）来实现应用程序的解耦和异步通信。消息队列的作用包括解决异步通信、削峰填谷、解耦应用组件等。

请介绍一下RabbitMQ。

RabbitMQ是一个开源的消息队列中间件，实现了AMQP（高级消息队列协议）的标准。它以可靠性、灵活的路由和丰富的功能而闻名，支持多种消息传递模式，如点对点、发布/订阅、工作队列等。

RabbitMQ的优点是什么？有哪些典型的使用场景？

RabbitMQ的优点包括灵活的路由、可靠性、持久化、高可用性和可扩展性。典型的使用场景包括异步任务处理、应用解耦、流量削峰、日志收集、事件驱动架构等。

RabbitMQ的实现原理是什么？

RabbitMQ使用Erlang语言开发，基于AMQP协议实现。它通过交换器（Exchange）、队列（Queue）和绑定（Binding）的组合来实现消息的路由和传递。

如何确保RabbitMQ的高可用性和可靠性？

RabbitMQ通过使用集群、镜像队列和持久化消息等机制来实现高可用性和可靠性。集群可以提供容错和负载均衡，镜像队列可以在多个节点之间复制消息，持久化消息可以在节点重启后恢复消息。

如何处理RabbitMQ中的消息丢失或者重复消费的问题？

要处理消息丢失，可以使用持久化消息、确认机制和备份机制。为了避免重复消费，可以使用消息的唯一ID、幂等处理和消费者端的去重机制。

请描述RabbitMQ的消费模型和消费者的处理方式。

RabbitMQ的消费模型是基于订阅/发布模式的。消费者通过订阅队列来接收消息，并可以使用手动确认机制来告知RabbitMQ消息已被处理。消费者可以以轮询或者推送的方式来处理消息。

RabbitMQ的负载均衡是如何实现的？有哪些负载均衡策略？

RabbitMQ通过交换器的路由策略和队列的消费者分配策略来实现负载均衡。常见的负载均衡策略包括：

Round-robin（轮询）：消息依次分发给每个消费者，实现简单的平均分配。

Least Connections（最少连接）：将消息发送给当前连接数最少的消费者，以实现动态负载均衡。

Random（随机）：随机选择一个消费者来处理消息。

Consistent Hashing（一致性哈希）：根据消息的某个属性值计算哈希值，然后将消息发送给哈希环上的对应消费者，实现一致性负载均衡。

请介绍一下RabbitMQ的持久化机制和消息的存储方式。

RabbitMQ的持久化机制通过将交换器、队列和消息标记为持久化来实现。持久化的消息和元数据会被写入磁盘，以在RabbitMQ节点重启后恢复。RabbitMQ使用内部数据库（Mnesia）来存储元数据和消息的索引，而消息的实际内容则存储在磁盘上。

如何保证RabbitMQ中消息的顺序性？

RabbitMQ本身并不保证消息的严格顺序性。但可以通过以下方式实现近似的消息顺序性：

使用单个队列：将具有相同顺序要求的消息发送到同一个队列，由同一个消费者按序处理。

设置队列级别的单一消费者模式：通过限制队列只能被一个消费者订阅，确保消息按照发送顺序被处理。

在消息中包含序列号：在消息的属性或内容中添加一个序列号字段，消费者在处理消息时根据序列号进行顺序性处理。

## **日志**

### 级别：

info() error() debug() warn()

### SLF4J在Spring MVC中

下面是一个完整的使用SLF4J在Spring MVC中测试四种日志级别的示例代码，包括配置：

添加SLF4J和Spring MVC的依赖：在项目的pom.xml文件中添加SLF4J和Spring MVC的依赖项。可以使用以下Maven配置：

<dependencies>

<!-- SLF4J -->

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-api</artifactId>

<version>1.7.32</version>

</dependency>

<dependency>

<groupId>ch.qos.logback</groupId>

<artifactId>logback-classic</artifactId>

<version>1.2.6</version>

</dependency>

<!-- Spring MVC -->

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<!-- 其他依赖项 -->

</dependencies>

请注意，上述示例使用了Logback作为SLF4J的实现和日志框架。您可以根据需要选择适合的日志实现。

创建Spring MVC配置类：创建一个配置类，用于配置Spring MVC。在该配置类中，添加SLF4J的配置。

@Configuration

@EnableWebMvc

public class MvcConfig implements WebMvcConfigurer {

@Bean

public Logger logger() {

return LoggerFactory.getLogger("LogController");

}

@Bean

public ViewResolver viewResolver() {

InternalResourceViewResolver resolver = new InternalResourceViewResolver();

resolver.setPrefix("/WEB-INF/views/");

resolver.setSuffix(".jsp");

return resolver;

}

@Override

public void configureDefaultServletHandling(DefaultServletHandlerConfigurer configurer) {

configurer.enable();

}

}

在上面的示例中，我们配置了一个名为logger的SLF4J记录器，并使用LoggerFactory创建它。还配置了一个视图解析器，用于解析JSP视图。

创建控制器类：创建一个控制器类，用于处理不同日志级别的请求。

@Controller  
public class LogController {  
​  
    @Autowired  
    private Logger logger;  
​  
    @RequestMapping("/testErrorLevel")  
    public ModelAndView testErrorLevel() {  
        logger.error("This is an error message.");  
​  
        return generateLogModelAndView("error", "This is an error message.");  
   }  
​  
    @RequestMapping("/testInfoLevel")  
    public ModelAndView testInfoLevel() {  
        logger.info("This is an info message.");  
​  
        return generateLogModelAndView("info", "This is an info message.");  
   }  
​  
    @RequestMapping("/testDebugLevel")  
    public ModelAndView testDebugLevel() {  
        logger.debug("This is a debug message.");  
​  
        return generateLogModelAndView("debug", "This is a debug message.");  
   }  
​  
    @RequestMapping("/testWarnLevel")  
    public ModelAndView testWarnLevel() {  
        logger.warn("This is a warning message.");  
​  
        return generateLogModelAndView("warn", "This is a warning message.");  
   }  
​  
    private ModelAndView generateLogModelAndView(String level, String message) {  
        ModelAndView modelAndView = new ModelAndView("logView");  
        modelAndView.addObject("level", level);  
        modelAndView.addObject("message", message);  
        return modelAndView;  
   }  
}

在上面的示例中，我们使用`Logger

Factory创建一个名为logger的SLF4J记录器，并使用@Autowired`注解将其注入到控制器中。然后，我们定义了四个处理不同日志级别请求的方法，并在每个方法中记录相应的日志消息。

generateLogModelAndView()方法创建一个ModelAndView对象，并将日志级别和消息添加到视图模型中。

创建日志视图模板：在WEB-INF/views目录下创建一个名为logView.jsp的JSP文件，并在其中编写日志视图的模板。

<!DOCTYPE html>  
<html>  
<head>  
    <title>Log View</title>  
</head>  
<body>  
    <h2>Log Details</h2>  
    <p><strong>Level:</strong> ${level}</p>  
    <p><strong>Message:</strong> ${message}</p>  
</body>  
</html>

在上面的示例中，我们使用EL表达式${level}和${message}从视图模型中获取日志级别和消息，并将其显示在JSP页面中。

创建Spring Boot启动类：创建一个Spring Boot的启动类，用于启动应用程序。

@SpringBootApplication

@Import(MvcConfig.class)

public class Application {

public static void main(String[] args) {

SpringApplication.run(Application.class, args);

}

}

在上面的示例中，我们使用@SpringBootApplication注解标记启动类，并使用@Import注解引入Spring MVC的配置类。

现在，您可以运行应用程序，并访问以下URL来测试不同的日志级别：

http://localhost:8080/testErrorLevel：测试error级别的日志。

http://localhost:8080/testInfoLevel：测试info级别的日志。

http://localhost:8080/testDebugLevel：测试debug级别的日志。

http://localhost:8080/testWarnLevel：测试warn级别的日志。

在访问以上URL时，将记录相应级别的日志，并显示日志的级别和消息。

请确保已经正确导入依赖项，并根据需要调整配置和URL路径。

### SLF4J和Log4j

SLF4J和Log4j是Java中常用的日志框架，它们通常一起使用，但是它们是两个不同的库，具有不同的功能和目的。

SLF4J（Simple Logging Facade for Java）：SLF4J是一个日志门面框架，旨在提供统一的日志接口。它不是具体的日志实现，而是为日志记录器（Logger）提供了一种通用的API，以便在运行时选择不同的日志实现。使用SLF4J，您可以在应用程序中使用一套统一的日志API，并在需要时更轻松地切换到不同的日志实现。

Log4j：Log4j是一个强大的日志框架，提供了灵活的日志记录和日志管理功能。它是一个完整的日志实现，实现了SLF4J所定义的日志接口。Log4j具有丰富的配置选项，可以根据需要灵活地配置日志输出格式、目标（如控制台、文件、数据库等）和日志级别等。

通常情况下，您会将SLF4J作为日志门面框架引入项目中，并选择一个具体的日志实现（如Log4j）作为后端实现。这样做的好处是，您可以将代码与具体的日志实现解耦，使得在需要更换或升级日志实现时更加方便。

下面是一个使用SLF4J和Log4j的示例配置：

添加依赖项：在项目的pom.xml文件中添加SLF4J和Log4j的依赖项。例如：

<dependencies>

<!-- SLF4J -->

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-api</artifactId>

<version>1.7.32</version>

</dependency>

<!-- Log4j -->

<dependency>

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

<artifactId>log4j-slf4j-impl</artifactId>

<version>2.17.0</version>

</dependency>

<dependency>

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

<artifactId>log4j-core</artifactId>

<version>2.17.0</version>

</dependency>

<!-- 其他依赖项 -->

</dependencies>

请注意，上述示例使用Log4j 2作为日志实现。您可以根据需要选择适合的Log4j版本。

配置Log4j：在项目的classpath下创建一个名为log4j2.xml的Log4j配置文件，并根据需要配置日志输出格式、目标和日志级别等。以下是一个简单的Log4j配置示例：

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

<Configuration>

<Appenders>

<Console name="Console" target="

SYSTEM\_OUT">

<PatternLayout pattern="%d{yyyy-MM-dd HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n"/>

</Console>

</Appenders>

<Loggers>

<Root level="info">

<AppenderRef ref="Console"/>

</Root>

</Loggers>

</Configuration>

上述示例配置了一个控制台（Console）输出器，并使用PatternLayout定义了日志输出的格式。根级别被设置为info，意味着只有info级别及以上的日志会被输出到控制台。

使用SLF4J：在代码中使用SLF4J接口来记录日志。例如：

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

public class MyClass {

private static final Logger logger = LoggerFactory.getLogger(MyClass.class);

public void doSomething() {

logger.info("This is an info message.");

logger.error("This is an error message.");

logger.debug("This is a debug message.");

logger.warn("This is a warning message.");

}

}

在上述示例中，我们使用LoggerFactory创建一个Logger实例，并使用info()、error()、debug()和warn()方法记录不同级别的日志消息。

通过上述配置和示例代码，您可以将SLF4J与Log4j集成到您的项目中，并使用SLF4J的统一API来记录日志。然后，根据Log4j的配置文件，日志消息将根据其级别输出到适当的目标（例如控制台）。

### Tomcat内置

在Tomcat中，内置的打印日志的类是org.apache.juli.logging.Log接口以及其实现类。Tomcat使用自己的日志实现，称为"JULI"（Tomcat的私有Java Util Logging实现）。JULI实现了Log接口，并提供了对Tomcat内部日志记录的支持。

以下是一些常用的Tomcat内置日志类：

org.apache.juli.logging.Log：Tomcat内置日志的接口，定义了一些常见的日志记录方法，如info()、error()、debug()、warn()等。

org.apache.juli.logging.LogFactory：用于获取Log实例的工厂类。通过调用LogFactory.getLog(Class)或LogFactory.getLog(String)方法，可以获取与指定类或名称相关联的Log实例。

org.apache.juli.logging.DirectJDKLog：JULI的默认日志实现之一，基于JDK的标准日志库java.util.logging。当Tomcat没有配置其他的日志实现时，默认使用该实现。

org.apache.juli.logging.DirectJULILog：JULI的默认日志实现之一，基于JUL（Java Util Logging）库。它是JDK 1.4之前的标准日志库，但在JDK 1.4及更高版本中仍然可用。

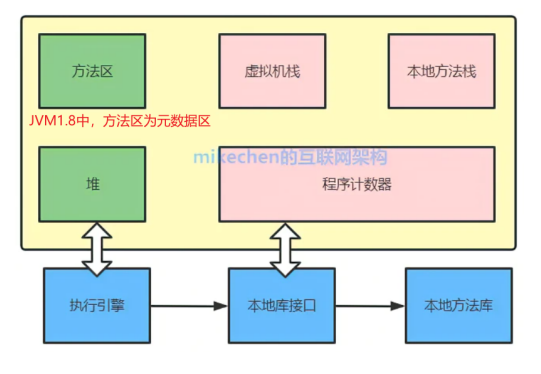
org.apache.juli.logging.DirectCommonsLog：JULI的默认日志实现之一，基于Apache Commons Logging（JCL）库。Apache Commons Logging是一种通用的日志抽象框架，允许在运行时选择不同的日志库实现。

请注意，Tomcat的内置日志类主要用于Tomcat内部的日志记录。如果您正在开发自己的Web应用程序，并希望使用Tomcat之外的日志库，您可以在应用程序中引入自己喜欢的日志框架，如SLF4J和Log4j，来记录日志。

## **JVM**

##### **JVM内存模型：**

可以分为两个部分，如下图所示，堆和方法区是所有线程共有的，而虚拟机栈，本地方法栈和程序计数器则是线程私有的。

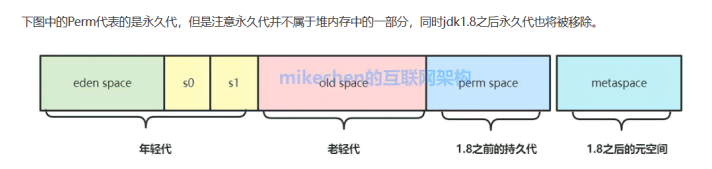


###### **堆（Heap）**

堆被划分成两个不同的区域：新生代 ( Young )、老年代 ( Old )，新生代 ( Young ) 又被划分为三个区域：Eden、From Survivor、To Survivor。

该内存区域存放了对象实例及数组（但不是所有的对象实例都在堆中）。

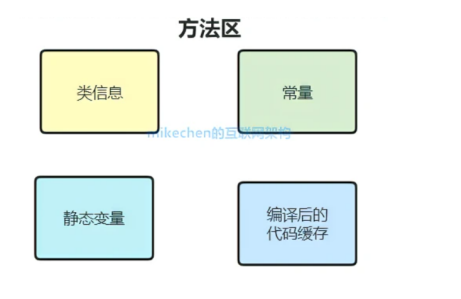
-Xms(最小值)和-Xmx(最大值)参数设置（最大最小值都要小于1G）



垃圾回收： 垃圾回收机制是分代回收策略。堆内存分为年轻代和年老代。年轻代分为eden区和幸存区，幸存区有幸存1区和幸存2区。他们之间大小比例是1:1：8。 当一个对象被new出来后，存入到eden区，当伊甸园区满了后，进行yong gc，使用复制算法，把幸存下来的对象放进幸存1区中。当对象被回收15次后仍然存活，会将对象放进年老区中。如果年老区内存满了，会执行full gc，对整个堆进行一次垃圾回收，使用的是标记清除算法。如果经过多次垃圾回收后，年轻区和年老区都满了，jvm会报出堆内存溢出异常。

###### **方法区（Method Area）**

是各个线程共享的内存区域，它用于存储已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码缓存等数据。



###### **虚拟机栈(JVM Stack)**

虚拟机栈（Java Virtual Machine Stack），早期也叫Java栈，每个线程在创建时都会创建一个虚拟机栈，其内部保存一个个的栈帧（Stack Frame），对应着一次次的Java方法调用。

虚拟机栈的作用：主管Java程序的运行，它保存方法的局部变量、部分结果，并参与方法的调用和返回。

###### **本地方法栈(Native Stack)**

本地方法栈（Native Method Stacks）与虚拟机栈所发挥的作用是非常相似的，其区别不过是虚拟机栈为虚拟机执行Java 方法（也就是字节码）服务，而本地方法栈则是为虚拟机使用到的Native 方法服务。

###### **程序计数器（PC Register）**

在JVM的概念模型里，字节码解释器工作时就是通过改变这个计数器的值来选取下一条需要执行的字节码指令。

分支、循环、跳转、异常处理、线程恢复等基础功能都需要依赖这个计数器来完成。

JVM的多线程是通过线程轮流切换并分配处理器执行时间的方式来实现的，为了各条线程之间的切换后计数器能恢复到正确的执行位置，所以每条线程都会有一个独立的程序计数器。

当线程正在执行一个Java方法，程序计数器记录的是正在执行的JVM字节码指令的地址；如果正在执行的是一个Natvie（本地方法），那么这个计数器的值则为空（Underfined）。

##### **内存泄漏和内存溢出**

内存泄漏（Memory leak）是指程序中已经不再使用的内存没有被正确释放，导致系统原本可以使用的内存空间被占据而无法使用。

是指应用程序在申请内存后，无法释放已经申请的内存空间。一次内存泄露危害可以忽略，但如果任其发展最终会导致内存溢出。如读取文件后流要进行及时的关闭以及对数据库连接的释放。

内存溢出（Memory overflow）则是指当程序运行时，申请的内存超过了系统实际可用的内存大小，导致程序崩溃或者出现其他异常情况。一般是由于程序设计不合理或算法错误等问题导致的。

是指应用程序在申请内存时，没有足够的内存空间供其使用。如我们在项目中对于大批量数据的导入，采用分段批量提交的方式。

## **类加载器**

类加载器是jre的一部分，负责动态将类添加到Java虚拟机。

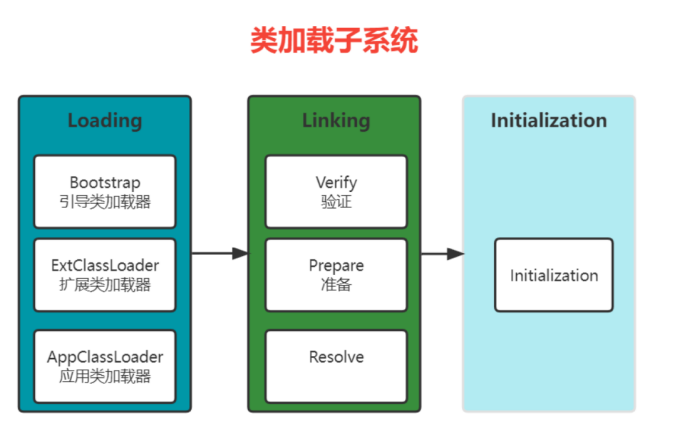
##### **类加载分类**

    1、启动类加载器 bootstrap classloader ：加载<JAVA\_HOME\>\jre\lib路径下的核心类库，由于安全考虑只加载 包名 java、javax、sun开头的类

    2、扩展类加载器 extension classloader ：加载<JAVA\_HOME>\jre\lib\ext目录下的类库

    3、应用程序类加载器 application classloader：加载 classpath环境变量所指定的类库，是用户自定义类的默认类加载器。

###### **类加载过程**



加载：将字节码文件通过IO流读取到JVM的方法区，并同时在堆中生成Class对象。

验证：校验字节码文件的正确性。

准备：为类的静态变量分配内存，并初始化为默认值；对于final static修饰的变量，在编译时就已经分配好内存了。

解析：将类中的符号引用转换为直接引用。

初始化：对类的静态变量初始化为指定的值，执行静态代码。

###### **双亲委派机制**

是指当一个类加载器收到一个类加载请求时，该类加载器首先会把请求委派给父类加载器。每个类加载器都是如此，只有在父类加载器在自己的搜索范围内找不到指定类时，子类加载器才会尝试自己去加载。

## **AtomicInteger**

AtomicInteger 是 Java 中提供的原子操作类之一，用于对整型数据进行原子性操作。它可以确保在多线程环境下对整型数据的操作是线程安全的，避免了竞态条件和数据不一致的问题。

AtomicInteger 类提供了一系列的方法来进行原子性操作，包括自增、自减、加法、减法、获取当前值等。下面是一些常用的方法：

int get()：获取当前的整型值。

void set(int newValue)：设置新的整型值。

int getAndSet(int newValue)：设置新的整型值并返回旧的值。

int getAndIncrement()：自增并返回自增前的值。

int getAndDecrement()：自减并返回自减前的值。

int incrementAndGet()：自增并返回自增后的值。

int decrementAndGet()：自减并返回自减后的值。

int addAndGet(int delta)：将指定值增加给当前值，并返回增加后的值。

int getAndAdd(int delta)：将指定值增加给当前值，并返回增加前的值。

以下是一个简单的示例，展示如何使用 AtomicInteger 进行原子操作：

javaCopy codeimport java.util.concurrent.atomic.AtomicInteger;

public class AtomicIntegerExample {

private static AtomicInteger counter = new AtomicInteger(0);

public static void main(String[] args) {

// 自增并获取当前值

int value = counter.incrementAndGet();

System.out.println("Value after increment: " + value);

// 自减并获取当前值

value = counter.decrementAndGet();

System.out.println("Value after decrement: " + value);

// 将指定值增加给当前值，并获取增加后的值

value = counter.addAndGet(5);

System.out.println("Value after adding 5: " + value);

}

}

需要注意的是，AtomicInteger 类提供的操作是原子性的，但不保证多个操作之间的原子性。如果需要对多个操作进行原子性组合，可以使用 synchronized 关键字或其他锁机制来保证操作的原子性。

通过使用 AtomicInteger 类，可以方便地进行线程安全的整型数据操作，避免了显式的锁机制带来的复杂性和性能开销。

## **AtomicLong**

生成订单编号 一毫秒内可以接收一万个以内不重复的订单编号

package com.southwind.controller;

import java.text.SimpleDateFormat;

import java.util.Date;

import java.util.concurrent.atomic.AtomicLong;

public class Demo {

private static AtomicLong seq = new AtomicLong(0);

private static final int max\_seq=9999;

public static String generateOrderNumber(){

SimpleDateFormat sdf = new SimpleDateFormat("yyyyMMddmmssSSS");

String timestamp = sdf.format(new Date());

long sequence = seq.getAndIncrement() % max\_seq;

String orderNumber = timestamp + String.format("%04d",sequence);

return orderNumber;

}

}

## **ABA问题**

使用原子类时有增有减时会发生aba问题

线程A读取变量的值为A。

在线程A执行CAS操作之前，线程B将变量的值从A修改为B，然后又将其修改回A。

线程A执行CAS操作，发现变量的值仍然为A，认为没有发生变化，执行交换操作。

在这种情况下，尽管变量的值经历了 A -> B -> A 的变化，但线程A无法察觉到中间的变化，导致出现ABA问题。

然而，在原子自增类中，比如 AtomicInteger，不存在以上情况。因为原子自增操作是一个连续的递增过程，不会发生类似于 ABA 的问题。原子自增操作本身是原子性的，确保在多线程环境下递增操作的正确性。

为了解决 ABA 问题，可以使用一些机制来增加对变量状态的判断和控制：

版本号或时间戳：引入一个额外的版本号或时间戳，记录每次变量的修改操作。在读取变量时，不仅要比较值是否相等，还要比较版本号或时间戳是否相等，以检测到变量是否发生了变化。

引入中间状态：在修改变量时，可以引入中间状态，即在 A 和 B 之间添加一个不同的状态，例如 C。这样，当其他线程读取到 C 时，就可以意识到变量发生了修改。

使用带有 CAS（Compare and Swap）操作的原子类：CAS 操作可以在更新变量时比较当前值与期望值，只有在当前值与期望值相等的情况下才进行更新。CAS 操作可以避免了传统的锁机制带来的性能开销，并且在进行变量修改时，会比较当前值是否发生了变化，从而避免了 ABA 问题。

在解决 ABA 问题时，通常会使用带有版本号的时间戳来确保数据的一致性。Java 中可以使用 AtomicStampedReference 类来实现这个目的。

AtomicStampedReference 是 Java 提供的原子引用类，它在引用的基础上添加了一个整型的时间戳（或版本号）。时间戳用于标识引用的更新状态，从而解决 ABA 问题。以下是使用 AtomicStampedReference 生成时间戳的示例：

javaCopy codeimport java.util.concurrent.atomic.AtomicStampedReference;

public class AtomicStampedReferenceExample {

private static AtomicStampedReference<String> data = new AtomicStampedReference<>("Initial Value", 0);

public static void main(String[] args) {

// 获取当前时间戳

long timestamp = System.currentTimeMillis();

// 获取当前版本号

int version = data.getStamp();

// 更新引用并增加版本号

data.set("New Value", version + 1);

// 其他操作...

// 检查引用和版本号是否一致

boolean success = data.compareAndSet("New Value", "Updated Value", version + 1, version + 2);

if (success) {

System.out.println("Update successful");

} else {

System.out.println("Update failed");

}

}

}

在上述示例中，AtomicStampedReference 对象 data 存储了一个字符串引用和一个整型的时间戳（版本号）。在更新引用时，除了更新引用值之外，还需要更新时间戳（版本号）。然后，使用 compareAndSet 方法比较引用和版本号，确保在操作过程中没有发生 ABA 问题。

## **锁**

(数据)修改丢失问题 ： 两个用户同时修改 后者会覆盖前者修改的数据 造成前者的修改覆盖丢失问题

乐观锁：在数据库表中增加一个版本号（或时间戳）字段，每次更新时比较当前版本号与更新前的版本号是否一致，如果一致则更新成功，否则表示数据已被其他操作修改，需要处理冲突。

时间戳是查询的提交时间 修改完之后查询一下是否一致

悲观锁：在更新数据之前使用锁机制，确保同一时间只有一个线程能够修改数据。可以使用数据库的行级锁或者应用层的锁来实现。

使用事务：将修改操作放在一个事务中，确保在事务提交之前其他线程无法修改同一数据，从而避免修改丢失的问题。

添加字段校验：在进行数据修改操作之前，先获取最新的数据并与修改前的数据进行比较，确保修改的数据与最新的数据一致，避免基于旧数据的修改导致丢失。

记录修改日志：在数据表中添加修改日志字段，记录每次的修改操作，包括修改前的数据和修改后的数据，以便追溯和恢复。

行级锁

表锁

悲观锁 sql语句后加 for update

乐观锁

## **Vue**

### 设置css只在当前组件中生效

<style scoped> //添加scoped进行设置

.my-style {

color: red;

}

</style>

### 常用的事件：

@click 点击事件

@input 输入事件

@submit 提交事件

@change 改变事件

@mouseover 鼠标移上去事件

@mouseout 鼠标移开事件

@keydown 按键按下事件

@keyup 按键抬起事件

@blur 失焦事件

@focus 聚焦事件

在Vue中，指令是带有前缀“v-”的特殊属性，用于对DOM元素进行动态绑定和操作。

### 以下是一些常见的Vue指令：

v-bind：用于动态绑定数据到元素的属性中。例如：<p v-bind:class="{'active': isActive}">。

v-if/v-show：用于根据表达式的值控制元素的显示或隐藏。其中，v-if会完全根据条件创建或销毁元素，而v-show只是简单地切换元素的CSS样式来显示或隐藏元素。

v-for：基于数组或对象的迭代器，用于渲染列表或表格等结构。例如：<li v-for="(item, index) in items">{{ item }}</li>。

v-on：用于监听DOM事件，并在触发时执行相关的方法。例如：Click me。

v-model：用于将表单元素与组件实例的property双向绑定在一起，以便用户可以以交互方式更新状态。例如：<input v-model="message">。

v-html：解析模板字符串并输出HTML内容。不过需要注意安全问题。

v-pre：跳过该元素和它的子元素的编译过程，直接输出未编译的Mustache标签。

v-cloak：防止闪现，指令保持在元素上直到相应实例结束编译。

这些Vue指令可以通过不同的方式来操作DOM元素，使得页面更加动态和交互。

### Vue 组件的生命周期：

beforeCreate：在实例刚被创建之初，数据观测和初始化事件还未开始。

created：实例已经完成了数据观测和初始化事件，但是尚未挂载到 DOM 上。

beforeMount：在挂载之前被调用，相关的 render 函数首次被调用。

mounted：组件挂载到 DOM 上后调用，可以访问到 DOM 节点。

beforeUpdate：响应式数据更新时调用，但是虚拟 DOM 并未重新渲染，也就是页面上的变化还未显示。

updated：虚拟 DOM 重新渲染并应用到页面后调用。

activated（keep-alive 组件特有）：被 keep-alive 缓存的组件激活时调用。 keep-alive 是 Vue 内置的一个组件，可以使被包含的组件保留状态，或避免重新渲染。

deactivated（keep-alive 组件特有）：被 keep-alive 缓存的组件停用时调用。

beforeDestroy：实例销毁之前调用，在这里可以进行一些清理工作。

destroyed：实例销毁后调用，这里没有访问到实例和 DOM 。

其中，beforeCreate 和 created 阶段中一般用于全局配置和初始化业务逻辑，mounted 中适合操作 DOM 节点以及与服务端交互。beforeDestroy 和 destroyed 阶段一般用于解除事件绑定、清除计时器等操作。

### vue其它

this.$refs.nextTick： 等待dom更新之后执行方法中的函数体，vue异步更新DOM

this.$refs.nextTick(callback) 的功能是：在下次 DOM 更新循环结束之后执行传入的回调函数。这个方法主要的作用是为了确保我们在更新数据之后可以获取到更新后的 DOM。

可以确保应用程序的 DOM 已经被更新后再执行回调函数，从而使我们能够更好地控制 DOM 的更新过程，提高应用程序的性能和可维护性。

mounted() {

// DOM 还没有更新

console.log(this.$refs.myRef.innerText); // ""

this.$nextTick(() => {

// DOM 更新完成

console.log(this.$refs.myRef.innerText); // "当前值是：20"

});

this.currentValue = 20; // 修改数据

}

### 怎么让css只在当前组件使用：

在 Vue 中，可以通过以下 3 种方法实现让 CSS 只在当前组件使用：

scoped 样式

在组件的 <style> 标签中使用 scoped 属性即可将样式限制在组件内部使用。示例代码如下：

<template>

<div class="example-class">

这是一个示例

</div>

</template>

<style scoped>

.example-class {

color: red;

}

</style>

这里的 .example-class 样式只会在当前组件中起作用。

#### **CSS module**

在 Vue 中，可以使用 CSS 模块化功能来实现将样式模块化并只对当前组件生效，具体实现方法如下：

在组件的 <style> 标签中使用 module 属性声明样式模块，在模块中定义的所有类都是局部作用域，不会影响到其他组件或全局样式。示例代码如下：

<template>

<div class="example-class">

这是一个示例

</div>

</template>

<style module>

.example-class {

color: red;

}

</style>

通过上述代码，我们可以实现将特定样式绑定到组件上，并使该样式只在当前组件内部起作用。

#### **CSS in JS**

CSS-in-JS 是一种方法，它允许你在 JavaScript 代码中编写 CSS 并直接应用于组件中。这种方法也可以做到将 CSS 样式限制在当前组件中使用，例如使用 styled-components 库，在 Vue 中实现的示例代码如下：

<template>

<example-class>这是一个示例</example-class>

</template>

<script>

import styled from 'vue-styled-components';

const StyledExampleClass = styled.div`

color: red;

`;

export default {

components: {

ExampleClass: StyledExampleClass

}

}

</script>

通过使用上述3种方法中的任何一种，我们都可以轻松地将样式应用到当前组件上，并让其只在当前组件内部使用。同时，由于它们具有不同的适用场景和特点，开发者可以根据实际需求进行选择和使用。

### 前端web项目如何优化

减少HTTP请求: 合并和压缩CSS和JavaScript文件。使用CSS Sprites技术可以将多个小图像合并到一张大图像中，然后使用CSS进行定位。

使用CDN（内容分发网络）: CDN可以将你的内容分发到全球各地的服务器上，使用户可以从地理位置最近的服务器获取内容，从而加速页面加载速度。

启用压缩: 启用Gzip压缩可以显著减少发送到浏览器的文件大小，从而加快页面加载速度。

缓存利用：利用浏览器缓存可以减少对同一资源的多次请求，尤其是对于那些不经常更改的资源。

代码优化：对JavaScript和CSS进行压缩和混淆，删除不必要的字符和空格，可以减少文件大小。同时，尽量避免使用高开销的JavaScript操作，例如全局变量、复杂的DOM操作等。

图片优化：只使用必要的图片，并确保它们被适当地压缩和优化。例如，使用WebP格式可以提供更高的压缩率，同时保持良好的图像质量。

使用异步加载：对于JavaScript和CSS，尽可能使用异步加载，以避免阻塞页面的渲染。

使用服务端渲染（SSR）或预渲染：这可以提高首次内容绘制（FCP）和首次有效绘制（FMP）的速度，从而提高SEO和用户体验。

使用性能分析工具：例如Lighthouse，PageSpeed Insights，Chrome DevTools等，它们可以帮助你检查和改善网站性能。

使用现代前端框架：React、Vue、Angular等现代前端框架都有各自的性能优化技术，如虚拟DOM、异步组件等。 以上只是一些基本的优化方法，你可以根据具体项目的需要来选择最适合的优化策略。

### 父子组件传值

父传子：props：{属性名，类型} props：【属性名】

子传父：this.$emit(中间事件，值)

使用props属性：父组件通过props属性向子组件传递数据，而子组件接收该数据。

// 在父组件中，使用v-bind指令将数据传递给子组件

<child-component :prop-data="parentData"></child-component>

// 在子组件中，接收该数据并使用它进行渲染

props: {

propData: String

}

emit方法：子组件通过emit方法触发一个自定义事件，并将数据作为参数传递给父组件。

// 在子组件中，使用$emit方法触发一个名为"update"的事件，并将数据作为参数传递给父组件

this.$emit('update', childData)

// 在父组件中，使用v-on指令监听该事件，并在事件处理函数中接收数据

<child-component v-on:update="handleUpdate"></child-component>

methods: {

handleUpdate(childData) {

// 处理子组件传递过来的数据

}

}

### $refs用法

在Vue中，refs是一个特殊的属性，用于在组件中访问子元素或组件实例。refs在父组件中通过ref属性设置，并在需要的时候在父组件中访问它。以下是使用$refs的一些常见场景：

访问DOM元素: 通过$refs访问子组件中的DOM元素。

<template>

<div ref="myDiv"></div>

</template>

mounted() {

const div = this.$refs.myDiv;

console.log(div); // 输出DOM元素

}

访问子组件实例：在父组件中使用$refs访问子组件的实例，以便直接调用子组件的方法和属性。

<template>

<child-component ref="myChild"></child-component>

</template>

mounted() {

const child = this.$refs.myChild;

child.myMethod(); // 调用子组件的方法

}

需要注意的是，在使用$refs时要注意不要过度使用，因为它违背了Vue中数据流向单项性的原则，可能会导致代码可读性和可维护性的下降。

### 全局事件总线

//main.js

new Vue({

router,

render: h => h(App),

beforeCreate(){

Vue.prototype.$bus =this

}

}).$mount('#app')

//接收，发起事件，跨组件传值

this.$bus.$emit('total',total)

//

this.$bus.$on('total', this.handletotal)

### 路由导航守卫

路由导航守卫（Navigation Guards）是Vue Router提供的一些机制，用于在路由发生变化时控制导航行为。通过跳转前或跳转后的勾子函数，可以实现一些页面需要登录权限、查询用户信息、取消路由等等功能。

使用路由导航守卫可以控制页面的跳转行为，从而实现诸如用户登录状态检测、权限验证、日志记录等功能。常用的导航守卫有以下三种：

全局导航守卫（全局前置守卫、全局后置钩子）：通过在router对象上设置beforeEach和afterEach方法来实现，分别在路由跳转前和跳转后调用。例如：

router.beforeEach((to, from, next) => {

// ...

next();

})

router.afterEach((to, from) => {

// ...

})

路由独享守卫：也称为组件级别的导航守卫，在定义路由时通过设置beforeEnter来实现。例如：

const router = new VueRouter({

routes: [

{

path: '/home',

component: Home,

beforeEnter: (to, from, next) => {

// ...

next();

}

}

]

})

组件内的导航守卫：组件内部定义beforeRouteEnter、beforeRouteUpdate、beforeRouteLeave等方法来实现路由切换时的逻辑控制。例如：

export default {

beforeRouteEnter(to, from, next) {

// ...

next();

},

beforeRouteUpdate(to, from, next) {

// ...

next();

},

beforeRouteLeave(to, from, next) {

// ...

next();

}

}

### Vuex

Vuex是Vue.js的官方状态管理库，用于实现全局状态管理和数据共享。以下是使用Vuex的一般步骤：

安装Vuex

可以使用npm或yarn来安装vuex：npm install vuex@3, 或者yarn add vuex

创建store

创建一个store.js文件，使用Vuex提供的createStore函数创建一个新的store对象。

import Vuex from 'vuex';

import Vue from 'vue';

Vue.use(Vuex);

const store = new Vuex.Store({

// 状态(state)

state: {

count: 0

},

// 更改状态(mutations)

mutations: {

increment(state) {

state.count++;

}

}

});

export default store;

注入store

在根Vue实例中，使用$store属性注入创建的store对象：

import Vue from 'vue';

import store from './store'

new Vue({

el: '#app',

store, // 将store对象注入到根Vue实例中

data: {

message: 'Hello Vue!'

}

});

使用state

在组件中，可以通过this.$store.state访问存储在state中的状态。例如，this.$store.state.count获取count状态值：

<template>

<div>

<p>{{ $store.state.count }}</p>

<button @click="$store.commit('increment')">增加</button>

</div>

</template>

<script>

export default {

name: 'Example',

methods: {

onAdd() {

this.$store.commit('increment');

}

}

};

</script>

在上述代码中，this.$store.commit('increment')会调用到store定义的mutations中的increment来更改count状态值。通过这种方式，我们就可以实现全局状态的共享和管理。

需要注意的是，Vuex提供了getter、action、module等辅助函数和特性，用于处理更复杂的业务规则和数据流程。在实际使用时，需要结合具体的应用场景，灵活使用Vuex的不同功能特性，实现高效的状态管理和数据共享。

### vue2和vue3的区别

更小的包大小：Vue 3采用了更小的代码库，从而可以更快地加载和渲染页面。

组合式API：Vue 3引入了组合式API，这使得逻辑复用和代码组织变得更加容易和灵活。

性能：Vue 3比Vue 2快得多，主要是由于在内部实现上进行了许多优化。

## **其它**

##### **日志级别**

info 一般处理业务逻辑的时候使用，就跟 system.err打印一样，用于说明此处是干什么的。slf4j使用的时候是可以动态的传参的，使用占位符 {} 。后边一次加参数，会挨个对应进去。

DEBUG（调试）：DEBUG日志通常用于在开发阶段为了检查是否按照预期执行了某些代码，以及输出重要的变量和方法执行过程中的状态信息。 一般放于程序的某个关键点的地方，用于打印一个变量值或者一个方法返回的信息之类的信息。

INFO（信息）：INFO级别用于常规信息输出，表示程序运行正常。

WARN（警告）：WARN级别表示出现一个警告情况，通常是由于一些异常或意料之外的情形而导致的。警告，不会影响程序的运行，但是值得注意

ERROR（错误）：ERROR级别表示出现了错误情况，例如应用程序无法提供某个服务或引发了异常。用户程序报错，必须解决的时候使用此级别打印日志。

# 设置日志级别

logging.level.com.example=DEBUG

# 将日志输出到控制台

logging.pattern.console=%d{yyyy-MM-dd HH:mm:ss} [%thread] %-5level %logger{36} - %msg%n

##### **java1.7和1.8区别是什么，主要在哪方面**

Lambda表达式：Java 1.8引入了Lambda表达式，这是一种更简洁、更灵活的函数式编程方式，使得代码更易于编写和理解。

Stream API：Java 1.8引入了Stream API，这是一个用于处理集合数据的API，可以大大简化集合的遍历和操作。

时间API：Java 1.8引入了一个新的时间API，提供了更好的日期和时间处理能力。

PermGen取消：Java 1.8取消了PermGen空间，取而代之的是Metaspace，这样可以更好地管理内存空间。

接口默认方法：Java 1.8允许在接口中定义默认方法，这样可以在不破坏现有代码的情况下向接口添加新功能。

并发集合类：Java 1.8增加了一些新的并发集合类，如ConcurrentHashMap、ConcurrentSkipListMap和ConcurrentSkipListSet等，这些集合类使得并发编程更加容易和高效。

合并重复的String：Java 1.8对字符串常量池做了优化，在编译期间就将相同的字符串合并为一个，并放到常量池中，减少了内存开销。

总的来说，Java 1.8在Lambda表达式、函数式编程、集合处理、时间处理、内存管理和并发编程等方面都进行了重大改进，使得Java编程更加先进和高效。

Java Development Kit (JDK) 1.8，也被称为Java 8，引入了许多新特性以改进编程效率和系统性能。下面是一些主要的新特性：

Lambda 表达式 (Lambda Expressions)：Lambda表达式是Java 8的最大变革。它允许我们将函数作为一个方法参数（函数作为参数），或者将代码看作数据。Lambda表达式有助于减少冗余代码，因此程序看起来更加清晰。

函数式接口 (Functional Interfaces)：Java 8在java.util.function包中引入了许多新的函数式接口。这些接口提供了一种在Lambda表达式和方法引用中使用的目标类型。

默认方法 (Default Methods)：Java 8允许我们向接口添加非抽象的方法实现，这就是所谓的默认方法或者扩展方法。

方法引用 (Method References)：方法引用是一种更简洁、更易读的Lambda表达式，它用来直接引用已存在的方法或构造器。

Stream API：Stream API是Java 8中的新特性，可以用于在Java中的集合、数组等数据源进行操作。Stream API可以并行执行操作，从而获得更高的性能。

Optional 类：Java 8 引入了一个新的 java.util.Optional<T> 类，可以防止 NullPointerException 异常。

日期/时间 API (Date/Time API)：Java 8引入了新的日期和时间API，解决了旧版日期/时间类库的一些问题。

接口的静态方法：Java 8允许我们在接口中定义静态方法。可以直接从接口中调用这些静态方法。

并行数组：Java 8增加了大量的新方法，使我们可以并行地处理数组，从而获得更高的性能。

Base64：Java 8提供了一个新的Base64类，用于编码和解码Base64数据。

Nashorn JavaScript引擎：Java 8引入了一个新的Nashorn JavaScript引擎，它允许在JVM上执行JavaScript代码。 以上就是Java 8（JDK 1.8）的一些主要新特性。这些新特性为开发者提供了更强大的编程工具，提高了编程效率，并且使得代码更加简洁易读。

##### **常用的包：**

java.lang java.io java.sql java.util java.net java.math java.text java.time

##### **常用类：**

Object 、Integer、String 、StringBuffer（线程安全）、StringBuilder、Bigdecimal、Date、File、Random、Math 、 System

List、set、map、ArrayList、hashMap、treeMap、Thread、ThreadLocal、

##### **常见的异常类型：**

RuntimeException: 运行时期异常；

NullPointerException：空指针异常;

IndexOutOfBoundsException：数组下角标越界异常;

ArrayIndexOutOfBoundsException（数组越界异常）：当尝试访问数组中不存在的索引时抛出。

ClassCastException（类转换异常）：当试图将对象强制转换为不是实例的子类时，抛出此异常;

ArithmeticException：当出现异常的运算条件时，抛出此异常。例如，一个整数“除以零”时，抛出此类的一个实例。

NumberFormatException：当试图将字符串转换成数字时，失败了，抛出;

IllegalArgumentException：抛出的异常表明向方法传递了一个不合法或不正确的参数;

SQLException：数据库相关的异常;（编译时异常）

IOException：当发生某种IO异常时抛出;（编译时异常）

FileNotFoundException：打开文件失败时抛出;（编译时异常）

ClassNotFoundException：没有找到指定的类。（编译时异常）

##### **HashMap 和 TreeMap、ConcurrentHashMap、HashTable的区别：**

HashMap、Hashtable 和 TreeMap 均是 Java 中常用的数据结构，用于存储键值对。它们有以下不同点：

线程安全性

HashMap 非线程安全，而 Hashtable 是线程安全的。因此，在多线程环境下需要保证对 HashMap 的互斥访问，可以使用 Collections.synchronizedMap() 或者使用 ConcurrentHashMap 来实现；而 Hashtable 可以直接在多线程环境下使用。

null 键和值

HashMap 允许键或值为 null，而 Hashtable 不允许键或值为 null。在 HashMap 中，将 null 作为键时，其 hash 值始终为 0；在 Hashtable 中，不能将 null 作为键或值，否则会抛出 NullPointerException 异常。

效率

HashMap 的效率一般高于 Hashtable，因为它采用了非同步处理方式，而 Hashtable 采用了 synchronized 关键字对方法或代码块进行加锁，相对更耗费性能。适用于单线程环境或少量数据操作时，建议使用 HashMap；在多线程环境中并发量很大时，建议使用 ConcurrentHashMap。

排序

TreeMap 是基于红黑树实现的，可以按照键的顺序（默认升序）进行排序；而 HashMap 和 Hashtable 则没有内置排序功能。

遍历顺序

HashMap 遍历顺序是无序的，Hash 表中的 key 的顺序是按照 key 的 hash 值来添加进去的，不保证顺序；而 TreeMap 则按照键的自然顺序（默认升序）来遍历；Hashtable 遍历顺序也是无序的，同样是按照哈希表中的顺序来遍历。

综上所述，HashMap 适合存储数据量大且无需考虑线程安全和排序的场景；Hashtable 适合多线程环境，但较为过时；TreeMap 适合需要排序的场景，数据量不宜过大。当然，具体的选择还需要根据实际业务场景进行综合考虑。

###### **ConcurrentHashMap实现原理**

ConcurrentHashMap 是 Java 中的线程安全哈希表，其实现原理可以简单概括如下：

ConcurrentHashMap 内部维护了一个数组（table），数组中每个元素又是一个链表或者红黑树，存储键值对。

ConcurrentHashMap 通过将整个 table 分成多个段/桶（segments/buckets），每个段/桶都是一个独立的哈希表，实现了对不同部分的读写操作进行并发控制，也就是分段锁（segment-level lock）机制。

ConcurrentHashMap 的 put 操作先获取相应的段/桶，然后对该段/桶加锁，再进行插入操作，put 操作结束之后释放锁。

ConcurrentHashMap 的 get 操作也先获取相应的段/桶，然后对该段/桶加锁，获取相应的键值对，get 操作结束后释放锁。

通过以上分段锁机制，ConcurrentHashMap 可以支持较高的并发度，并且具有较好的可伸缩性。同时为了保证在高并发情况下能够保证正确性，ConcurrentHashMap 还处理一些片段级别的条件判断和 CAS 操作等。

##### **面向对象和面向过程的区别**

面向对象和面向过程都是编程范式，两者的主要区别在于：

数据和行为的封装方式：面向对象编程将数据和行为封装在对象中，通过对象之间的交互来完成任务；而面向过程编程将数据和函数分开处理，主要强调的是程序执行的过程。

程序的设计思想：面向对象编程注重的是程序设计的结构和模块化，更加注重程序的可扩展性、复用性和维护性；面向过程编程则注重将任务分解成一组小的步骤，并通过函数来实现这些步骤的顺序执行。

程序开发的方式：面向对象编程采用自上而下的开发方式，首先定义对象、方法等概念，然后将这些概念转化成代码实现；而面向过程编程则采用自下而上的开发方式，先定义独立的函数，并将其组合成完成任务的程序。

代码的可读性：面向对象编程的代码可读性更强，方法名称和对象名称可以直接表达程序的含义和逻辑，更易于理解；而面向过程编程的代码可读性相对较弱，只能通过函数调用的顺序来推断程序的含义和逻辑。

总之，面向对象编程更加注重程序的架构和复用性，适合开发大型复杂应用；面向过程编程注重执行过程和具体实现，适合处理流程化、简单的问题。

##### **面向对象三大特征：**

###### **1、封装**

是指一种将抽象性函式接口的实现细节部分包装、隐藏起来的方法。封装可以被认为是一个保护屏障，防止该类的代码和数据被外部类定义的代码随机访问。要访问该类的代码和数据，必须通过严格的接口控制。 适当的封装可以让程式码更容易理解与维护，也加强了程式码的安全性。

封装步骤： 私有化成员变量 ，提供getter和setter方法 ，（重写toString方法）

###### **2、继承**

就是子类继承父类的特征和行为，使得子类对象（实例）具有父类的实例域和方法，或子类从父类继承方法，使得子类具有父类相同的行为。

父类的非private成员变量和抽象方法，子类可以直接使用。

父类私有的属性和方法，子类不能继承。

父类final修饰的变量，子类不能修改。

父类final修饰的方法，子类不能重写。

###### **3、多态**

多态是同一个行为具有多个不同表现形式或形态的能力。

继承的多态： 父类的引用类型变量指向子类的实例化对象。父类的引用类型会调用到子类的方法。

接口的多态： 接口变量指向接口的实现类对象。接口的引用类型变量能够访问到它的实现类的方法。

多态存在的三个必要条件：1. 继承 2.重写 3.父类引用指向子类对象。

当使用多态方式调用方法时，首先检查父类中是否有该方法，如果没有，则编译错误；如果有，再去调用子类的同名方法。

多态的好处： 可以使程序有良好的扩展，并可以对所有类的对象进行通用处理。

##### **重写和重载：**

重写： 重写是子类对父类的允许访问的方法的实现过程进行重新编写。子类在继承了父类的方法之后，如果父类的方法不满足子类的需求，那么子类可以

重写父类的方法。当子类重写之后，子类只能调用到重写之后的业务逻辑。子类只能继承一个父类。

重载： 重载是在一个类里面，方法名字相同，而形参列表不同。返回类型可以相同也可以不同

重载条件： 一个相同（方法名），一个不同（形参列表不同）（跟类型有关、跟个数有关、跟顺序有关、跟参数名无关， 两个无关（访问权限和返回类型）

##### **抽象类和接口：**

抽象类： 一个类中没有包含足够的信息来描绘一个具体的对象。用abstract修饰，抽象类不能被new。

抽象方法只包含一个方法名，而没有方法体，方法名后面直接跟一个分号。

如果一个类包含抽象方法，那么该类必须是抽象类。抽象类中不一定有抽象方法。

任何子类必须重写父类的抽象方法，或者声明自身为抽象类。最终，必须有子类实现该抽象方法，否则，从最初的父类到最终的子类都不能用来实例化对象。

接口： 是一个特殊的类，是抽象方法的集合（都没有方法体），接口通常以interface来声明。一个类通过继承接口的方式，从而来继承接口的抽象方法。

一个实现接口的类，必须实现接口内所描述的所有方法，否则就必须声明为抽象类。

接口不能包含成员变量，除了 static 和 final 类型的变量，必须指定初始值。

接口不是被类继承了，而是要被类实现。接口支持多继承。 抽象类和接口都可以用于实现多态性和抽象化设计，但是有以下几点区别：

抽象类可以包含非抽象方法的实现，而接口中只能包含方法的签名，没有实现。

一个类只能继承一个抽象类，但可以实现多个接口。

抽象类的变量可以是公共、受保护、私有的，而接口中定义的变量全都是 public static final 的常量。

接口中不能有构造函数，而抽象类有构造函数。

抽象类可以拥有实例变量，而接口中只有静态常量。

总之，当需要定义一些默认行为或者共享的状态时，应该使用抽象类。当需要定义一些规则和契约时，应该使用接口。

##### **final 在 java 中有什么作用**

final 修饰的类叫最终类，该类不能被继承。

final 修饰的方法不能被重写。

final 修饰的变量叫常量，常量必须初始化，初始化之后值就不能被修改。

##### **static 静态关键字有什么作用**

修饰类：整个类都是静态的

修饰方法：没有成员变量，不用new直接用

修饰成员变量：一个类的成员变量对这个类的实例对象来说是同一份

静态代码块：类加载时候加载，只执行一次

##### **常用的设计模式**

###### **单例模式：**

Java 开发中的单例模式（Singleton Pattern）是一种创建型设计模式，保证一个类只有一个实例，并且提供全局访问点。在 Java 中，单例模式常用于需要唯一实例的场景，例如线程池、配置文件管理等。使用单例模式时需要注意线程安全问题，尤其是在懒汉式和双重检查单例中需要使用 synchronized 关键字或者 volatile 修饰符保证线程安全。同时也需要注意单例模式可能带来的风险，例如程序的可测试性不佳、内存泄漏等问题。

实现单例模式的方法：

饿汉式单例：在类加载时就创建对象，线程安全，但无法进行延迟加载。

懒汉式单例：在第一次使用时才创建对象，可以进行延迟加载，但线程不安全。

双重检查单例：通过加锁和双重检查来保证线程安全，在第一次使用时才创建对象，是一种常用的实现方法。

枚举单例： JDK 5 引入了枚举类型，因为枚举类型的实例化是线程安全的且只会执行一次，因此可以使用枚举实现单例。

###### **代理模式：**

Java 开发中的代理模式（Proxy Pattern）是一种结构型设计模式，能够控制对对象的访问。在代理模式中，代理对象扮演了被代理对象的替身，客户端通过访问代理对象来实现对被代理对象的间接访问。代理模式的实现通常使用接口或者抽象类定义代理和被代理对象的公共接口，代理对象与被代理对象实现相同的接口，从而可以在客户端中无感知地替换。常见的代理模式实现包括 JDK 动态代理 和 CGLIB 代理等。

场景：远程对象的操作，日志记录、缓存数据，安全代理，虚拟代理

###### **工厂模式：**

工厂模式（Factory Pattern）是一种创建型设计模式，它通过定义一个用于创建对象的接口，让子类决定实例化哪一个类。工厂模式可以隐藏对象创建的复杂性，降低模块之间的依赖关系，提高代码的可扩展性和可维护性。

在 Java 中实现工厂模式的方法有多种，常见的包括：

简单工厂模式：由一个工厂类根据传入的参数，决定创建哪一种产品类的实例。

工厂方法模式：将对象的创建延迟到子类中去实现，即由具体的工厂子类来创建出具体的产品对象。

抽象工厂模式：提供一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类。

在使用工厂模式时，根据需要选择不同的实现方式。简单工厂模式适合创建的对象较少的场景，但拓展性比较差；工厂方法模式和抽象工厂模式可以更灵活地进行对象的创建。为了保持代码的简洁性和易读性，应该尽可能地避免在代码中直接使用 new 关键字创建对象，而是使用工厂模式来管理对象的创建过程。

###### **设计模式的原则**

设计模式有一些原则，这些原则帮助开发人员更好地理解和应用设计模式。

开闭原则（Open-Closed Principle）：对扩展开放，对修改关闭。这意味着在改变一个系统的功能时，应该通过添加新代码来扩展它，而不是修改已有的代码。

单一职责原则（Single Responsibility Principle）：一个类应该只有一个引起它各种行为的原因。这个原则表示应该尽可能地将每个类限制在单一的责任范围内，这有助于提高类的可重用性和可维护性。

里氏替换原则（Liskov Substitution Principle）：所有引用父类对象的地方，都能够透明地使用其子类的对象。也就是说，子类不能改变父类原有的行为和属性，否则会导致代码出现问题。

依赖倒置原则（Dependency Inversion Principle）：高层模块不应该依赖低层模块，而是应该依赖于抽象。这个原则强调模块之间的依赖应该基于抽象而不是具体实现。

接口隔离原则（Interface Segregation Principle）：多个专门的接口优于一个通用的接口。这个原则指出，应该尽可能地将接口分离为更小的、更具体的接口，而不是使用通用的、多用途的接口。

迪米特法则（Law of Demeter，又称最少知道原则）：一个对象应该对其他对象尽可能少的了解。换句话说，一个对象不应该直接调用另一个对象的方法，而是应该通过它们之间的中介对象进行通信。

组合/聚合复用原则（Composite/Aggregate Reuse Principle）：通过组合或聚合关系来实现代码复用。相比于继承，组合/聚合更加灵活，且不会带来额外的耦合关系。

##### **定时任务**

在 Spring Boot 中，我们可以使用注解和定时任务调度器来实现定时任务。以下是使用 Spring Boot 实现定时任务的示例：

首先，在 pom.xml 文件中添加 spring-boot-starter-quartz 依赖：

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-quartz</artifactId>

</dependency>

然后，在应用程序的主类文件中加上 @EnableScheduling 注解来启用 Spring 的自动定时任务：

@SpringBootApplication

@EnableScheduling

public class Application {

public static void main(String[] args) {

SpringApplication.run(Application.class, args);

}

}

最后，在需要定时执行的方法上加上 @Scheduled 注解，并设置好执行的时间间隔等参数。例如：

@Component

public class MyTask {

// 每10秒钟执行一次任务

@Scheduled(cron = "0/10 \* \* \* \* ?")

public void execute() {

// 这里填写你要执行的任务

System.out.println("Job is done!");

}

}

上述代码中，我们定义了一个名为 MyTask 的类，并使用 @Component 注解将它注册为 Spring 容器中的一个 Bean。然后在 MyTask 类中定义了一个 execute() 方法，并使用 @Scheduled 注解来声明每隔 10 秒就执行一次该方法。

注意：在使用 @Scheduled 注解时需要同时指定执行的间隔时间或者 Cron 表达式；如果不需要使用 Cron 表达式，也可以使用 fixedDelay 或 fixedRate 来指定执行的时间间隔。

总之，通过使用 Spring Boot 的定时任务功能，我们可以方便地实现任务调度、周期性更新等操作，提高开发效率和应用程序的稳定性。

##### **Cookie、localStorage和sessionStorage都是用于客户端存储数据的工具，但它们有以下区别：**

存储大小：Cookie存储数据的大小为4KB，而localStorage和sessionStorage的大小限制通常为5MB左右。

存储时效性与有效性：Cookie可以设置指定的过期日期；sessionStorage的有效时间为窗口关闭，localStorage可以一直保留数据直到用户手动删除。

存储位置：Cookie和localStorage是存储在本地客户端，而sessionStorage是存储在浏览器的会话存储中。对于同一应用程序的所有页面，它们共享相同的sessionStorage。

安全性：Cookie 和localStorage 可被浏览器存储历史和第三方插件所窃取；SessionStorage对当前会话保留有效，关闭浏览器后丢失。考虑到安全问题，尤其是用户登录和使用敏感数据时，应优先使用服务器端存储。

与服务器端的交互方式：cookie在页面请求时会自动发送到服务器端，而localStorage和sessionStorage 不会自动发送。

综上所述，Cookie和localStorage、sessionStorage 都有各自的用途和优缺点。在选择使用时，需根据具体的应用场景来选择合适的存储工具。

##### **如何实现登录只能在一个设备登录**

要实现登录只能在一个设备登录，可以考虑以下实现方案：

使用Token或者Session等标识用户身份的机制，在用户成功登录后，将当前用户登录标识存储在数据库中或存储在服务器端的缓存中。

每次用户访问需要登录的资源时，都需要验证用户登录标识是否有效，以此来判断用户是否已经在其他设备上登录了。

对于用户在其他设备上的登录，可以在验证用户登录标识的时候发现，此时可以直接踢掉其他设备上的登录，强制其下线。

如果某个用户发起登录请求，而该用户的登录标识在数据库或者缓存中已经存在，说明该用户已经在其他设备上登录，此时可以拒绝该设备的登录请求，提示用户必须先完成其他设备上的注销或者下线操作之后才能在该设备上登录。

如果用户在某个设备上主动注销或者下线操作，需要及时将其登录标识从数据库或者缓存中删除，以便其可以在其他设备上完成登录操作。

需要注意的是，这种方案可能会对用户体验产生一定的影响，而且需要考虑到跨设备同步等问题，需要根据具体应用场景谨慎使用。

# 知识点

## **序列化和反序列化**

json对象转为json字符串是序列化、

Java对象转为xml 和 json

在Java中，序列化（Serialization）是指将对象转换为字节序列的过程，以便可以将其存储到文件、内存或通过网络进行传输。反序列化（Deserialization）则是指将字节序列转换回对象的过程。

当一个对象需要在不同的系统之间进行传输或持久化存储时，需要进行序列化和反序列化操作。序列化可以将对象转换为字节流，使其能够在网络中进行传输或在存储介质中进行存储。而反序列化则可以将字节流重新转换为对象，以便恢复对象的状态。

## **将java对象序列化**

在Java中，可以使用不同的库来将一个 JSON 对象序列化为字符串或字节数组。以下是两种常用的库及其使用方法：

Jackson 库：Jackson 是一个流行的 JSON 处理库，可以轻松地在 Java 对象和 JSON 之间进行序列化和反序列化。下面是使用 Jackson 库将 JSON 对象序列化为字符串的示例：

javaCopy codeimport com.fasterxml.jackson.core.JsonProcessingException;

import com.fasterxml.jackson.databind.ObjectMapper;

public class JsonSerializationExample {

public static void main(String[] args) throws JsonProcessingException {

// 创建一个 JSON 对象

MyObject myObject = new MyObject("John", 25);

// 创建 ObjectMapper 对象

ObjectMapper objectMapper = new ObjectMapper();

// 将 JSON 对象序列化为字符串

String jsonString = objectMapper.writeValueAsString(myObject);

// 打印序列化结果

System.out.println(jsonString);

}

}

Gson 库：Gson 是另一个流行的 JSON 处理库，它提供了一组简单易用的 API 来进行序列化和反序列化。下面是使用 Gson 库将 JSON 对象序列化为字符串的示例：

javaCopy codeimport com.google.gson.Gson;

public class JsonSerializationExample {

public static void main(String[] args) {

// 创建一个 JSON 对象

MyObject myObject = new MyObject("John", 25);

// 创建 Gson 对象

Gson gson = new Gson();

// 将 JSON 对象序列化为字符串

String jsonString = gson.toJson(myObject);

// 打印序列化结果

System.out.println(jsonString);

}

}

在上述示例中，MyObject 是一个自定义的 Java 类，代表要序列化的对象。使用相应的库，将其转换为 JSON 字符串即可。

## **Java设计原则**

在Java编程中，有一些重要的设计原则可以帮助开发人员编写可维护、可扩展和高效的代码。以下是几个常见的Java设计原则：

单一职责原则（Single Responsibility Principle，SRP）：一个类应该有且只有一个引起它变化的原因。换句话说，一个类应该只负责一项任务或职责。这样可以使类的设计更加清晰、模块化，并且易于维护和扩展。

开放封闭原则（Open-Closed Principle，OCP）：软件实体（类、模块、函数等）应该对扩展开放，对修改封闭。通过使用抽象和接口，可以使系统在不修改现有代码的情况下进行扩展。

里氏替换原则（Liskov Substitution Principle，LSP）：子类应该能够替换掉父类并且不产生任何错误或异常。这意味着子类在行为上应该与其基类保持一致，而不是改变基类的预期行为。

依赖倒置原则（Dependency Inversion Principle，DIP）：高层模块不应该依赖于低层模块，而是应该依赖于抽象。抽象不应该依赖于具体实现细节，而是应该依赖于接口或抽象类。这样可以降低模块之间的耦合度，提高代码的可扩展性。

接口隔离原则（Interface Segregation Principle，ISP）：客户端不应该依赖它不需要的接口。一个类对其他类的依赖应该建立在最小的接口集上。通过定义精确的接口，可以避免类与不相关的接口产生依赖关系，从而减少类之间的耦合。

迪米特法则（Law of Demeter，LoD）：一个对象应该对其他对象有尽可能少的了解。一个类应该只与其直接合作的类发生交互，而不应该了解其他类的内部细节。这样可以降低类之间的依赖关系，提高代码的灵活性和可维护性。

这些设计原则是软件开发中的基本指导原则，有助于提高代码的质量、可读性和可维护性。当应用这些原则时，开发人员应根据具体情况进行权衡和适当的应用，以实现良好的软件设计。

## **Java中的八大基本数据类型**

在Java中，有八种基本数据类型，分为两类：原始数据类型和引用数据类型。以下是Java的八大基本数据类型：

byte：表示8位有符号整数，取值范围为-128到127。

short：表示16位有符号整数，取值范围为-32,768到32,767。

int：表示32位有符号整数，取值范围为-2,147,483,648到2,147,483,647。

long：表示64位有符号整数，取值范围为-9,223,372,036,854,775,808到9,223,372,036,854,775,807。

float：表示32位单精度浮点数，用于表示带小数点的数字。它可以表示约7位有效数字。

double：表示64位双精度浮点数，用于表示带小数点的数字。它可以表示约15位有效数字。

char：表示16位Unicode字符，用于表示单个字符。

boolean：表示布尔值，只能取两个值之一：true或false。

这些基本数据类型是Java中的构建块，用于存储不同种类的数据。它们具有不同的取值范围和用途，开发人员可以根据需要选择适当的数据类型来存储和操作数据

## **单例的设计模式**

单例模式是一种创建型设计模式，其目的是确保一个类只有一个实例，并提供全局访问点以获取该实例。

实现单例模式的常见方法是：

饿汉式单例（Eager Initialization）： 在类加载时就创建实例，并在类中直接提供一个静态常量来引用该实例。这种方式在多线程环境下是线程安全的，但可能会导致资源浪费，因为即使不使用该实例，它也会被创建。

javaCopy codepublic class Singleton {

private static final Singleton instance = new Singleton();

private Singleton() {}

public static Singleton getInstance() {

return instance;

}

}

懒汉式单例（Lazy Initialization）： 延迟创建实例，只有在需要时才会创建。在多线程环境下，需要考虑线程安全性。

javaCopy codepublic class Singleton {

private static Singleton instance;

private Singleton() {}

public static synchronized Singleton getInstance() {

if (instance == null) {

instance = new Singleton();

}

return instance;

}

}

双重检查锁定（Double-Checked Locking）： 在懒汉式单例的基础上进行改进，减少锁的使用，提高性能。

javaCopy codepublic class Singleton {

private static volatile Singleton instance;

private Singleton() {}

public static Singleton getInstance() {

if (instance == null) {

synchronized (Singleton.class) {

if (instance == null) {

instance = new Singleton();

}

}

}

return instance;

}

}

除了上述方法外，还可以使用静态内部类、枚举等方式来实现单例模式。

单例模式在需要保证只有一个实例存在且可全局访问时非常有用，例如数据库连接池、日志记录器等。然而，过度使用单例模式可能导致代码耦合性增加，使得代码难以测试和维护。因此，使用时需要权衡其优缺点，并根据具体场景进行选择

## **jdk,jre,jvm是什么及其关系**

在Java开发中，JDK（Java Development Kit）、JRE（Java Runtime Environment）和JVM（Java Virtual Machine）是三个重要的概念，它们之间存在紧密的关系。

JDK（Java Development Kit）： JDK是Java开发工具包，它是开发Java应用程序所必需的主要工具集合。JDK包括了Java编译器（javac）、Java虚拟机（JVM）、Java类库（Java API）等工具和组件。开发人员可以使用JDK来编写、编译和调试Java程序。

JRE（Java Runtime Environment）： JRE是Java运行时环境，它是在计算机上运行Java程序所必需的环境。JRE包括Java虚拟机（JVM）和Java类库（Java API）。当用户运行Java程序时，JRE会提供必要的运行时支持。

JVM（Java Virtual Machine）： JVM是Java虚拟机，它是Java程序的执行环境。JVM负责将Java字节码（由Java源代码编译而来）解释或编译为本地机器代码，以便计算机可以理解和执行。JVM提供了内存管理、垃圾回收、安全性和其他运行时特性。

关系： JDK包含了JRE和一些开发工具（如编译器和调试器）。JDK提供了完整的Java开发环境，可以用于编写、编译和调试Java程序。

JRE是Java程序运行的基本环境，包括Java虚拟机（JVM）和Java类库。JRE没有提供开发工具，只用于执行Java程序。

JVM是Java虚拟机，它是JRE的一部分。JVM负责在运行时执行Java字节码，并提供了内存管理、垃圾回收等功能。

简单来说，JDK是用于开发Java程序的工具包，JRE是用于运行Java程序的运行时环境，而JVM是在运行时执行Java程序的虚拟机

## **怎么测试servlet是单例还是多例**

在Java Servlet中，Servlet容器负责创建和管理Servlet的实例。默认情况下，Servlet容器会使用多线程来处理客户端请求，并且每个请求都会在独立的线程中执行。这可能导致对于同一个Servlet类的多个实例的并发访问。

如果你想确定一个Servlet是单例还是多例，可以进行以下测试：

实例变量：在Servlet类中添加一个实例变量，并观察它在不同请求之间的行为。如果每个请求都访问相同的实例变量值，那么该Servlet是单例。如果实例变量的值在不同请求之间发生变化，那么该Servlet是多例。

javaCopy code@WebServlet("/test")

public class TestServlet extends HttpServlet {

private int counter = 0;

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

counter++;

response.getWriter().println("Counter: " + counter);

}

}

打印实例标识：在Servlet的初始化方法（init()）中打印出当前Servlet实例的标识（如内存地址或唯一标识符），并观察是否每次请求都打印相同的标识。如果标识在不同请求之间相同，则说明该Servlet是单例。

@WebServlet("/test")

public class TestServlet extends HttpServlet {

public void init() throws ServletException {

System.out.println("Servlet instance: " + this);

}

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

// ...

}

}

需要注意的是，Servlet规范并没有要求Servlet一定是单例或多例。Servlet的实例化和管理是由Servlet容器负责的，容器可以根据自身的实现和配置来确定Servlet的实例化方式。如果你需要确保Servlet是单例的，可以考虑使用线程安全的设计和编码实践，或者使用依赖注入框架来管理Servlet实例的生命周期

## **单例和多例的优势**

单例和多例是在软件设计中使用的两种不同的对象创建和管理方式，它们各自具有不同的优势和适用场景。

单例的优势：

全局访问：单例模式确保只有一个实例存在，并提供全局访问点，可以在系统的任何地方方便地获取该实例。

资源共享：单例模式适用于需要共享资源的情况。由于只有一个实例存在，可以避免多个实例之间的资源冲突和浪费。

节约资源：单例模式可以减少系统中对象的创建和销毁次数，从而节约了系统的资源消耗。

数据一致性：单例模式可以确保系统中的某个状态或数据保持一致性，避免因为多个实例引入的数据不一致问题。

多例的优势：

灵活性：多例模式允许创建多个实例，每个实例都可以拥有自己的状态和行为，提供更灵活的对象创建和管理方式。

并发处理：多例模式适用于多线程环境，不同的实例可以同时处理多个请求，避免了线程安全的问题。

隔离性：多例模式可以在不同的实例之间进行隔离，每个实例可以具有不同的配置、属性或行为，满足不同的需求。

扩展性：多例模式可以方便地扩展，根据需要创建更多的实例，而无需修改原有代码。

选择单例或多例取决于具体的应用场景和需求。如果需要确保全局唯一性或共享资源，单例是一个合适的选择。如果需要灵活性、隔离性或并发处理，多例可能更适合。在实际设计中，需要根据系统的特点、性能需求和并发情况等因素进行权衡和选择

### 各自的缺点

单例和多例模式各自存在一些缺点，需要在设计时加以考虑。

单例模式的缺点：

难以扩展：由于单例模式只有一个实例，因此在需要扩展和修改时可能会面临困难。对于某些情况下需要多个实例的变化需求，修改现有的单例实现可能会引入风险。

耦合性增加：单例模式将对象的创建和使用耦合在一起，可能会导致代码难以测试和维护。在依赖单例实例的代码中，很难模拟和替换该实例，导致测试困难。

线程安全问题：在多线程环境下，需要特别关注单例实例的线程安全性。如果没有正确处理并发访问，可能会导致竞态条件和数据不一致的问题。

多例模式的缺点：

对象创建成本高：相比单例模式，多例模式需要创建多个实例，可能会增加对象的创建和销毁成本。如果对象较为复杂或资源消耗较大，可能会对系统性能产生一定影响。

难以全局管理：由于多例模式允许创建多个实例，需要在全局范围内管理和跟踪这些实例。如果管理不当，可能会导致资源泄漏或混乱。

代码复杂性增加：多例模式引入了更多的实例管理逻辑，可能会增加代码的复杂性和维护难度。

在选择单例或多例模式时，需要综合考虑具体的需求和系统特点。如果单例模式能够满足需求，且不会引入过多的耦合和线程安全问题，可以优先考虑使用单例。如果需要更大的灵活性、隔离性和并发处理能力，并且能够承受对象创建成本的增加，可以考虑使用多例模式

## **什么对象设置成单例**

单例模式适用于以下情况：

资源共享：当多个部分需要共享同一个资源时，可以使用单例模式确保只有一个实例存在，避免资源冲突和重复创建。

全局访问：某些对象需要在整个应用程序中全局访问，例如日志记录器、配置管理器等。使用单例模式可以方便地获取对象的实例。

控制对象数量：有些对象的创建和销毁成本较高，为了节约资源和提高性能，可以将其设计为单例，避免多次创建和销毁。

以下是一些常见的对象可以设置为单例：

数据库连接池：确保数据库连接池在应用程序中只有一个实例，供多个线程共享数据库连接。

日志记录器：保证只有一个日志记录器实例，可以在整个应用程序中记录日志。

缓存管理器：确保只有一个缓存管理器实例，用于管理应用程序中的缓存数据。

配置管理器：用于加载和管理应用程序的配置信息，保证全局唯一性。

线程池：保证线程池在应用程序中只有一个实例，用于管理并发执行的任务。

需要注意的是，并非所有对象都适合设置为单例。在决定将对象设计为单例时，需要考虑对象的生命周期、资源共享需求以及线程安全等因素，并权衡单例带来的优势和缺点。

## **ioc容器 默认是单例**

在常见的IoC（Inversion of Control）容器中，默认情况下，管理的对象通常是以单例（Singleton）的方式创建和管理的。

单例模式可以确保IoC容器中的对象在整个应用程序生命周期内只有一个实例，并且可以被多个组件或服务共享。这样可以提高性能、节省资源，并确保对象的状态和数据的一致性。

当使用IoC容器（如Spring）进行依赖注入时，如果没有显式地指定作用域（Scope），通常会使用默认的单例作用域。这意味着在容器中只会创建一个实例，并且每次请求该实例时都会返回同一个实例。

但是，IoC容器也支持其他作用域，如原型（Prototype）作用域，这意味着每次请求都会创建一个新的实例。这对于某些需要独立状态的对象或有状态的组件是有用的。

需要注意的是，每个IoC容器可能有不同的默认行为和配置选项，因此确切的默认作用域可能会有所不同。在使用具体的IoC容器时，最好查阅相关文档或配置来确定默认的对象作用域。

## **什么是线程安全的**

线程安全是指在多线程环境下，对共享资源的访问和操作能够保证正确和一致的行为。

在并发编程中，多个线程可以同时访问和修改共享资源，如果没有适当的同步机制和线程安全的设计，可能会导致以下问题：

竞态条件（Race Condition）：多个线程同时访问和修改共享资源，导致结果依赖于线程执行的相对时序，产生不确定的结果。

数据不一致：多个线程同时修改共享资源，导致数据状态不一致，可能会引发逻辑错误或数据损坏。

死锁（Deadlock）：多个线程在等待彼此持有的资源，导致所有线程都无法继续执行，造成系统停滞。

为了实现线程安全，需要采取合适的并发控制手段和设计策略，例如：

互斥访问：使用互斥锁（Mutex）或其他同步机制，保证同一时间只有一个线程可以访问共享资源，避免竞态条件。

原子操作：使用原子操作或原子类（Atomic Class）来保证对共享资源的操作是不可分割的，从而避免数据不一致性。

同步机制：使用同步关键字（synchronized）或其他并发工具（如信号量、条件变量等）来协调多个线程的执行，保证线程之间的正确协作。

不可变性（Immutability）：设计不可变对象，避免共享资源被修改，从而避免竞态条件和数据不一致性。

线程安全的设计和实现需要考虑多线程访问共享资源的各种可能情况，并采取适当的措施来保证线程安全性。

互斥锁

在Java中，可以使用内置的互斥锁机制来实现对共享资源的互斥访问。Java提供了 java.util.concurrent 包中的 Lock 接口和 ReentrantLock 类，可以用于创建和管理互斥锁。

下面是一个示例代码，展示了如何在Java中使用互斥锁：

import java.util.concurrent.locks.Lock;

import java.util.concurrent.locks.ReentrantLock;

// 创建互斥锁

Lock mutex = new ReentrantLock();

// 线程类

class MyThread extends Thread {

public void run() {

// 获取锁

mutex.lock();

try {

// 访问共享资源

// ...

} finally {

// 释放锁

mutex.unlock();

}

}

}

// 创建并启动多个线程

for (int i = 0; i < numThreads; i++) {

Thread thread = new MyThread();

thread.start();

}

// 等待所有线程完成

for (Thread thread : threads) {

try {

thread.join();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

在示例中，使用 ReentrantLock 类创建了一个互斥锁对象 mutex。每个线程在执行需要互斥访问的操作之前，通过调用 lock() 方法获取锁，并在完成后通过 unlock() 方法释放锁。使用 try-finally 块确保在任何情况下都能正确释放锁。

需要注意的是，使用互斥锁时，应该遵循相同的获取和释放锁的规则，以确保线程能够正确同步并避免死锁等问题。此外，Lock 接口和 ReentrantLock 类还提供了其他有用的方法，如 tryLock() 可以尝试获取锁而不阻塞，newCondition() 可以创建条件变量等，可以根据具体需求进行使用。

## **进程和线程的区别**

进程（Process）和线程（Thread）是操作系统中的两个重要概念，它们具有以下区别：

定义：进程是执行中的程序实例，具有独立的内存空间和系统资源；线程是进程的执行单位，是进程内的一个独立执行流程。

资源占用：每个进程都有独立的内存空间和系统资源（如文件句柄、打开的网络连接等），而线程共享所属进程的资源。

调度和切换：进程是系统进行调度和分配资源的基本单位，进程间的切换代价较高；线程是在进程内部切换执行的，线程间的切换代价较低。

并发性：在多核处理器上，多个进程可以并发执行，每个进程可以利用多个线程实现并发执行。

通信与同步：进程间通信需要使用特定的机制，如管道、消息队列、共享内存等；线程之间可以通过共享内存直接进行通信和数据共享，但需要考虑同步和互斥问题。

稳定性：一个进程的异常或错误通常不会影响其他进程的正常执行；但一个线程的异常或错误可能会导致整个进程崩溃。

创建和销毁：创建和销毁进程的开销较大；创建和销毁线程的开销相对较小。

总的来说，进程是资源分配和调度的基本单位，线程是进程内部并发执行的基本单位。进程之间相互独立，而线程共享进程的资源。在设计和开发中，需要根据具体的需求和场景合理选择使用进程或线程，以达到最佳的性能和资源利用。

## **单例怎么写**

要实现单例模式，可以按照以下步骤进行编写：

私有化构造方法：将类的构造方法设置为私有，这样外部无法直接实例化该类。

创建静态私有实例：在类内部创建一个私有静态变量，用于保存类的唯一实例。

提供公共静态方法：提供一个公共的静态方法，用于获取类的实例。

在公共静态方法中判断实例是否存在：在公共静态方法内部判断实例是否已经存在。如果存在，直接返回该实例；如果不存在，创建新的实例并返回。

下面是一个基本的单例模式的示例（线程安全的懒汉式单例模式）：

javaCopy codepublic class Singleton {

private static Singleton instance;

private Singleton() {

// 私有化构造方法

}

public static synchronized Singleton getInstance() {

// 提供公共的静态方法，获取类的实例

if (instance == null) {

instance = new Singleton();

}

return instance;

}

}

在上述示例中，通过将构造方法私有化，防止外部直接实例化该类。在getInstance()方法中，使用懒加载的方式判断实例是否已经存在，如果不存在则创建新的实例并返回。由于加了synchronized关键字，保证了线程安全，但可能会影响性能。

需要注意的是，这只是一种基本的单例模式实现方式。根据具体的需求和场景，还可以使用其他的单例模式实现方式，如饿汉式单例、双重检查锁单例、静态内部类单例等，以满足不同的线程安全、性能和延迟加载等需求。

## **IOC,AOP**

IoC（Inversion of Control，控制反转）和AOP（Aspect-Oriented Programming，面向切面编程）是两种常见的软件开发设计原则和编程范式。

IoC（控制反转）：

IoC 是一种设计原则，它将对象的创建和依赖关系的管理从应用程序代码中解耦，交给容器来负责。它反转了传统的控制流程，由容器来控制对象的创建和生命周期。

在 IoC 中，应用程序通过配置或注解来描述对象之间的依赖关系，容器根据配置信息实例化对象，并将所需的依赖注入到对象中，从而实现解耦和灵活的组件管理。

AOP（面向切面编程）：

AOP 是一种编程范式，它通过将系统功能的不同关注点进行分离和模块化，以横切关注点（Cross-cutting Concerns）的方式来提高代码的可维护性和重用性。

在 AOP 中，系统的关注点（如日志记录、安全检查、事务管理等）被抽象为切面（Aspect），然后通过横切技术将切面与主要业务逻辑进行织入，从而实现功能的复用和解耦。

示例代码如下：

package com.example;

public class MyService {

public void method1() {

System.out.println("Executing method1");

}

public void method2(String param) {

System.out.println("Executing method2 with parameter: " + param);

}

public int method3(int num) {

System.out.println("Executing method3 with number: " + num);

return num \* 2;

}

}

我们希望在 MyService 类的所有方法执行之前和之后记录日志。

根据上述示例代码，我们可以使用以下切入点表达式来匹配 MyService 类中的所有方法：

execution(\* com.example.MyService.\*(..))

其中：

execution()：切入点表达式的起始关键字。

\*：通配符，表示任意返回类型。

com.example.MyService：目标类的完全限定名。

\*：通配符，表示匹配任意方法名。

(..)：括号中的表达式表示方法的参数，.. 表示匹配任意数量、任意类型的参数。

现在，我们可以创建一个切面类，使用 @Before 和 @After 注解来定义前置通知和后置通知：

javaCopy codepackage com.example;

import org.aspectj.lang.annotation.Aspect;

import org.aspectj.lang.annotation.Before;

import org.aspectj.lang.annotation.After;

import org.springframework.stereotype.Component;

@Aspect

@Component

public class LoggingAspect {

@Before("execution(\* com.example.MyService.\*(..))")

public void beforeMethodExecution() {

System.out.println("Before method execution: Logging started");

}

@After("execution(\* com.example.MyService.\*(..))")

public void afterMethodExecution() {

System.out.println("After method execution: Logging completed");

}

}

在上述示例中，LoggingAspect 类被定义为一个切面类，使用 @Aspect 和 @Component 注解标记。

@Before("execution(\* com.example.MyService.\*(..))") 表示在切入点匹配的方法执行之前执行前置通知的逻辑。

@After("execution(\* com.example.MyService.\*(..))") 表示在切入点匹配的方法执行之后执行后置通知的逻辑。

当我们在应用程序中使用上述配置时，每次调用 MyService 类中的方法时，切面类中定义的前置通知和后置通知将会执行，输出相应的日志信息。

请注意，上述示例是一个简化的示例，实际的AOP应用可能涉及更复杂的切面逻辑和切入点表达式的定义。同时，需要将切面类配置为Spring应用程序的一部分，以便Spring容器能够自动检测和应用切面。具体的配置方式可以根据使用的Spring版本和项目结构进行调整。

IoC 和 AOP 在实际应用中经常一起使用：

IoC 容器提供了 AOP 的基础设施，可以在容器中定义和管理切面（Aspect）以及与主要业务逻辑进行织入。

AOP 可以通过 IoC 容器来实现切面的自动织入，无需显式地在业务代码中编写繁琐的切面逻辑。

常见的 IoC 容器框架（如Spring）提供了集成的 AOP 功能，使开发者能够方便地应用 IoC 和 AOP 的概念和技术，提高代码的可维护性、可扩展性和可测试性。

### aop在项目中的应用

Spring AOP（Aspect-Oriented Programming）是Spring框架提供的一种面向切面编程的支持。它通过在运行时动态地将代码织入到应用程序的特定位置，实现对横切关注点的处理，例如日志记录、性能监控、事务管理等。

Spring AOP的应用可以在项目中实现以下功能：

日志记录：通过使用AOP，可以在方法执行前后插入日志记录的逻辑，用于记录方法的入参、返回值、执行时间等信息，方便系统的运行监控和故障排查。

事务管理：通过AOP可以实现声明式事务管理，将事务的开启、提交和回滚等操作与业务逻辑解耦。可以通过在方法上添加事务相关的注解，如@Transactional，实现对方法的事务管理。

安全性控制：通过AOP可以实现对系统的安全性控制，例如对访问某些敏感接口或资源的权限验证。可以在方法执行前判断用户是否具有相应的权限，从而决定是否允许访问。

性能监控：通过AOP可以实现性能监控功能，例如统计方法的执行时间、调用次数等指标。可以在方法执行前后记录时间戳，计算方法的执行时间，并进行统计和分析。

异常处理：通过AOP可以实现全局的异常处理，将异常的捕获和处理逻辑与业务逻辑解耦。可以在AOP切面中定义异常处理的逻辑，对方法抛出的异常进行捕获和处理。

在项目中使用Spring AOP需要进行以下步骤：

定义切面：创建一个切面类，使用@Aspect注解标识，该类包含一组切点和通知。切点定义了需要拦截的方法，通知定义了在切点处执行的逻辑。

配置切面：在Spring的配置文件（如XML配置文件或Java配置类）中，配置切面的实例和相关的切点、通知等信息。

启用AOP：在Spring的配置文件或启动类中，启用AOP的支持。可以使用<aop:aspectj-autoproxy>标签或@EnableAspectJAutoProxy注解来启用AOP。

应用切面：在需要应用AOP的地方，通过注解或XML配置等方式将切面应用到目标方法或类上。

以上是Spring AOP在项目中的应用方式。通过使用AOP，可以将一些通用的横切关注点与业务逻辑进行解耦，提高代码的复用性、可维护性和可扩展性。

## **作用**

IoC（控制反转）和AOP（面向切面编程）的作用如下：

IoC 的作用：

解耦和灵活性：通过将对象的创建和依赖关系的管理交给容器，实现了对象之间的解耦，使得应用程序的组件更加灵活和可配置。

可测试性：由于依赖关系被注入到对象中，可以方便地进行单元测试，通过模拟和替换依赖对象来进行测试。

可扩展性：通过配置或注解来描述对象之间的依赖关系，可以方便地添加、修改或替换组件，实现应用程序的可扩展性。

生命周期管理：IoC 容器可以管理对象的生命周期，确保对象的创建、销毁和依赖关系的管理在合适的时机进行，减少资源的浪费和管理的复杂性。

AOP 的作用：

横切关注点的模块化：通过将系统的关注点（如日志、事务、安全等）进行分离和模块化，使得关注点的代码可以集中处理，提高了代码的可维护性和重用性。

解耦和复用：将系统功能中与业务逻辑无关的横切关注点（如日志记录）从业务代码中剥离出来，使得业务代码更加专注于核心功能，提高了代码的可读性和维护性。

避免代码重复：通过将公共的横切关注点抽象为切面，可以在不同的模块或组件中重用，避免了重复编写相同代码的问题。

动态代理和增强：AOP 可以通过动态代理技术对目标对象进行包装，并在特定的切点（Join Point）上应用切面逻辑，实现对目标对象的增强和功能扩展。

综上所述，IoC 和 AOP 提供了一种设计和编程的思想和技术，能够提高代码的可维护性、可测试性和可扩展性，降低了代码的耦合度，促进了代码的重用和模块化。这些特性使得软件开发更加灵活、高效和可靠。

## **DI**

DI（Dependency Injection，依赖注入）是一种实现 IoC（控制反转）的具体技术和模式，用于解决对象之间的依赖关系管理问题。

在传统的开发方式中，对象通常负责创建和管理它们所依赖的其他对象。这种紧耦合的关系会导致代码难以测试、难以扩展和维护困难。DI 通过将对象的依赖关系交由外部的容器来管理，实现了对象之间的解耦。

依赖注入的基本思想是：通过构造函数、属性注入或方法注入等方式，将一个对象所依赖的其他对象注入到它的内部，使得对象无需自己创建或获取所需的依赖对象。这样做的好处是，对象只需要关注自己的核心业务逻辑，而不需要关心依赖对象的创建和生命周期管理。

依赖注入有以下几个关键概念：

依赖关系：表示一个对象依赖于另一个对象，通常体现为一个对象使用另一个对象的实例或接口。

被注入对象：需要依赖注入的对象，即被注入依赖的对象。

依赖注入器（容器）：负责管理对象的创建和依赖关系的注入的容器或框架，例如 Spring 容器。

注入方式：有多种方式可以进行依赖注入，包括构造函数注入、属性注入和方法注入等。

DI 的优势包括：

解耦：通过将依赖关系外置，实现了对象之间的解耦，使得对象更加独立和可复用。

可测试性：由于依赖关系被注入，可以方便地进行单元测试，通过模拟和替换依赖对象来进行测试。

可扩展性：通过配置或注解来描述依赖关系，可以方便地添加、修改或替换依赖对象，实现应用程序的可扩展性。

总之，依赖注入是一种实现控制反转的重要技术，通过解耦和外置依赖关系的管理，提高了代码的灵活性、可测试性和可扩展性。

## **作用**

依赖注入（Dependency Injection，DI）的主要作用如下：

解耦：通过将对象之间的依赖关系外置，实现了对象之间的解耦。被依赖的对象无需自己创建或管理依赖对象，而是由外部容器负责创建和注入依赖对象。这样可以降低代码之间的耦合度，使得代码更加灵活、可维护和可测试。

可测试性：依赖注入使得对象的依赖关系可配置和替换，从而方便进行单元测试。在测试过程中，可以通过注入模拟或替代的依赖对象来隔离被测试对象，从而更容易编写和执行单元测试。

可扩展性：通过依赖注入，可以方便地添加、修改或替换依赖对象，从而实现应用程序的可扩展性。通过配置或注解描述依赖关系，可以在不修改代码的情况下进行依赖的调整，从而满足不同的需求和变化。

可重用性：通过将依赖关系外置，可以将对象设计为更加独立和可复用的组件。依赖注入使得对象的创建和管理与对象本身的功能解耦，从而可以更方便地在不同的上下文中重用对象。

生命周期管理：依赖注入容器通常负责对象的生命周期管理。它可以在适当的时机创建、初始化和销毁对象，确保对象的生命周期管理在控制之下，避免资源泄露和管理的复杂性。

总的来说，依赖注入通过解耦、可测试性、可扩展性和可重用性等特性，提供了一种灵活、可维护和可测试的方式来管理对象之间的依赖关系。它是实现控制反转（IoC）的重要手段，为软件开发提供了更好的代码组织和管理方式。

## **ioc的作用域**

在 IoC（Inversion of Control，控制反转）容器中，对象的作用域定义了对象在容器中的生命周期和可见范围。不同的作用域决定了对象在容器中的创建和销毁时机，以及对象在不同环境下的可见性。

常见的 IoC 容器作用域包括以下几种：

单例（Singleton）作用域：

在单例作用域下，IoC 容器中只存在一个对象的实例。

对象在容器启动时创建，并在整个应用程序的生命周期中保持单一实例。

所有请求该对象的组件都共享同一个实例。

单例作用域适用于那些无状态的、线程安全的对象，或者需要共享状态的对象。

原型（Prototype）作用域：

在原型作用域下，每次从容器中获取对象时都会创建一个新的实例。

每个请求该对象的组件都获得一个独立的实例。

对象的创建和销毁由容器动态管理。

原型作用域适用于那些需要独立状态或频繁创建和销毁的对象。

会话（Session）作用域：

会话作用域适用于 Web 应用程序中，用于表示用户会话的作用域范围。

在同一个会话中，请求该对象的组件都获得相同的实例。

对象的创建和销毁由容器管理，并在会话开始和结束时进行。

请求（Request）作用域：

请求作用域适用于 Web 应用程序中，用于表示每个 HTTP 请求的作用域范围。

在同一个请求中，请求该对象的组件都获得相同的实例。

对象的创建和销毁由容器管理，并在每个请求处理期间进行。

其他自定义作用域：

除了上述常见的作用域之外，一些 IoC 容器还支持自定义作用域。

自定义作用域可以根据特定的业务需求定义对象的生命周期和可见性。

不同的作用域适用于不同的场景和需求。选择适当的作用域可以确保对象的生命周期和可见性符合业务需求，提高代码的灵活性、可维护性和性能。

## **ioc内的懒汉模式和饿汉模式**

懒汉模式和饿汉模式是常见的单例模式实现方式，它们与 IoC（Inversion of Control，控制反转）并没有直接的关系。但是，可以在 IoC 容器中使用这些单例模式来管理对象的创建和生命周期。

下面是将懒汉模式和饱汉模式改造为适合在 IoC 容器中使用的示例：

懒汉模式（Lazy Initialization）：

javaCopy codepublic class LazySingleton {

private static LazySingleton instance;

private LazySingleton() {

// 私有构造函数

}

public static synchronized LazySingleton getInstance() {

if (instance == null) {

instance = new LazySingleton();

}

return instance;

}

}

在 IoC 容器中使用懒汉模式的单例对象时，可以将其注册为单例的 Bean，并在需要时延迟创建实例。

饿汉模式（Eager Initialization）：

javaCopy codepublic class EagerSingleton {

private static final EagerSingleton instance = new EagerSingleton();

private EagerSingleton() {

// 私有构造函数

}

public static EagerSingleton getInstance() {

return instance;

}

}

在 IoC 容器中使用饱汉模式的单例对象时，可以直接将其注册为单例的 Bean，由容器在启动时创建并管理实例。

需要注意的是，懒汉模式的改造中添加了线程同步，以保证线程安全性。在多线程环境中，如果不进行同步处理，可能会导致多个线程同时创建实例，破坏了单例的特性。

当然，使用 IoC 容器时，可以采用更加简洁和优雅的方式来管理对象的单例性，比如通过配置或注解来声明对象的作用域为单例。这样，容器会负责创建、管理和提供单例对象，无需手动实现单例模式。

## **mybatis中的dao能重载吗**

在 MyBatis 中的 DAO（Data Access Object）接口是基于映射文件（Mapper XML）的，而接口中的方法是通过 XML 配置文件中的 SQL 语句进行映射的。由于 XML 配置文件中的 SQL 是通过方法名进行匹配的，因此 DAO 接口中的方法是不能重载的。

如果在 DAO 接口中定义了多个具有相同方法名但参数列表不同的方法，MyBatis 在解析 SQL 映射时将会出现冲突，无法确定具体要执行哪个 SQL 语句。这会导致运行时错误或不可预测的结果。

如果需要定义多个具有类似功能但参数列表不同的方法，可以考虑使用不同的方法名或者使用参数的注解来区分方法。例如，可以在方法中使用 @Param 注解来指定参数的名称，从而实现不同的 SQL 映射。

## **ioc的设计模式**

IoC（Inversion of Control，控制反转）是一种设计模式，它通过将对象的依赖关系外部化，实现了对象之间的解耦和灵活性。IoC 的设计模式包括以下几种：

依赖注入（Dependency Injection，DI）：依赖注入是 IoC 的一种具体实现方式。它通过将对象所依赖的其他对象注入到它的内部，而不是由对象自己创建或管理依赖对象。依赖注入可以通过构造函数、属性注入或方法注入等方式实现。

控制反转容器（Inversion of Control Container）：控制反转容器是 IoC 模式的核心，负责管理对象的创建、依赖注入和生命周期。它通过配置或编程方式描述对象的依赖关系，并在运行时负责解析和注入依赖对象。常见的 IoC 容器包括 Spring Framework 的 ApplicationContext、Google Guice 等。

依赖查找（Dependency Lookup）：依赖查找是另一种实现 IoC 的方式。它与依赖注入相反，对象通过查询容器来获取所需的依赖对象。典型的例子是使用 JNDI（Java Naming and Directory Interface）查找资源。

服务定位器（Service Locator）：服务定位器是一种用于解决对象依赖关系的中心化注册和查找机制。它通过提供统一的服务注册和查找接口，将对象与其所需的依赖解耦。典型的例子是使用 JNDI 或自定义的服务定位器来查找和获取服务对象。

这些设计模式和技术在实现 IoC 的过程中起着重要的作用，帮助实现对象之间的解耦和灵活性。它们可以提高代码的可维护性、可测试性和可扩展性，并促进模块化和组件化的开发。

## **单例模式和工厂模式**

单例模式（Singleton Pattern）和工厂模式（Factory Pattern）是两种常见的设计模式，它们用于解决不同的问题。

单例模式：

单例模式旨在确保一个类只有一个实例，并提供全局访问点以获取该实例。

它适用于需要在整个应用程序中共享状态或提供统一访问点的类。

单例模式通常包含一个私有的构造函数，限制了实例的创建，并提供一个静态方法或静态变量来获取该实例。

单例模式的实现方式包括懒汉模式（Lazy Initialization）、饿汉模式（Eager Initialization）等。

工厂模式：

工厂模式旨在封装对象的创建逻辑，并通过工厂方法来创建对象，而不是直接使用 new 关键字。

它适用于需要创建不同类型对象的场景，并提供一致的创建接口，隐藏具体对象的创建细节。

工厂模式通常包含一个工厂类，其中定义了工厂方法，根据不同的参数或条件来创建具体的对象实例。

工厂模式的实现方式包括简单工厂模式、工厂方法模式、抽象工厂模式等。

单例模式和工厂模式在目的和应用场景上有所不同。单例模式关注于确保一个类只有一个实例，提供全局访问点，适用于需要共享状态或统一访问点的情况。而工厂模式关注于封装对象的创建逻辑，提供一致的创建接口，适用于需要创建不同类型对象的场景。在实际应用中，它们经常一起使用，工厂模式可以用来创建单例对象的实例。例如，可以使用工厂模式创建单例对象的实例，并提供全局访问点来获取该实例。

## **对象配置到ioc容器内的几种方式**

在将对象配置到 IoC（Inversion of Control，控制反转）容器中时，有几种常见的方式：

XML 配置方式：

使用 XML 文件来配置对象的创建和依赖关系。

在 XML 配置文件中定义对象的 Bean 定义，包括对象的类名、属性值、依赖关系等。

IoC 容器在启动时解析 XML 配置文件，并根据配置创建和管理对象的实例。

注解配置方式：

使用注解来标记对象、依赖关系和作用域等信息。

在对象的类上使用注解（如 @Component、@Service、@Autowired 等）来表示其在容器中的角色和属性。

IoC 容器扫描类路径下的注解，并根据注解信息创建和管理对象的实例。

Java 配置方式：

使用 Java 代码来配置对象的创建和依赖关系。

创建一个 Java 配置类，使用特定的注解（如 @Configuration、@Bean）来定义对象的创建和依赖关系。

IoC 容器在启动时读取 Java 配置类，并根据配置创建和管理对象的实例。

基于接口的配置方式：

实现特定的接口来定义对象的创建和依赖关系。

创建一个实现特定接口的类，该接口定义了对象的创建方法和依赖关系。

IoC 容器通过接口来创建和管理对象的实例。

这些方式可以单独使用，也可以组合使用，根据具体的需求和偏好选择适合的方式。无论使用哪种方式，目标都是将对象配置到 IoC 容器中，由容器负责创建、管理和提供对象的实例。

### 结合代码

首先，创建一个名为 UserService 的类：

javaCopy codepublic class UserService {

private UserRepository userRepository;

public UserService(UserRepository userRepository) {

this.userRepository = userRepository;

}

public void addUser(String username) {

userRepository.saveUser(username);

System.out.println("User added: " + username);

}

}

然后，创建一个名为 UserRepository 的接口：

javaCopy codepublic interface UserRepository {

void saveUser(String username);

}

接下来，在 XML 配置文件中定义对象的配置。假设配置文件名为 applicationContext.xml：

xmlCopy code<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">

<bean id="userRepository" class="com.example.UserRepositoryImpl" />

<bean id="userService" class="com.example.UserService">

<constructor-arg ref="userRepository" />

</bean>

</beans>

接着，创建一个 UserRepositoryImpl 类，实现 UserRepository 接口：

javaCopy codepublic class UserRepositoryImpl implements UserRepository {

public void saveUser(String username) {

System.out.println("Saving user: " + username);

// 实际的保存逻辑

}

}

最后，编写一个启动类，加载配置文件并获取 UserService 实例：

javaCopy codeimport org.springframework.context.support.ClassPathXmlApplicationContext;

public class Main {

public static void main(String[] args) {

// 加载配置文件

ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");

// 获取 UserService 实例

UserService userService = context.getBean("userService", UserService.class);

// 使用 UserService

userService.addUser("John");

// 关闭容器

context.close();

}

}

以上示例展示了如何使用 XML 配置方式将 UserService 和 UserRepository 对象配置到 IoC 容器中，并通过容器来获取 UserService 实例。在实际运行时，IoC 容器会负责创建 UserService 实例并注入相应的依赖（即 UserRepository 实例），从而实现了对象的创建和依赖管理。

### 实现原理是反射

在示例中，IoC 容器实现对象的创建和依赖注入的机制通常使用了反射技术。当容器加载配置文件并初始化时，它会使用反射来实例化配置文件中定义的对象，并将对象的依赖关系注入到相应的属性或构造函数中。

在使用反射创建对象时，可以通过以下步骤来实现：

获取对象的类信息：容器读取配置文件中定义的类名，并使用反射机制获取该类的 Class 对象。

实例化对象：通过调用 Class 对象的 newInstance() 方法或通过 Constructor 对象的 newInstance() 方法，创建对象的实例。

注入依赖关系：通过反射获取对象的属性或构造函数，并将依赖的对象注入到相应的属性或构造函数参数中。这可以通过反射调用对象的 setter 方法或使用构造函数的参数来实现。

返回对象实例：将创建好并注入依赖的对象实例返回给调用方。

反射机制使得容器可以在运行时动态地创建对象，并在不了解具体对象类型的情况下进行依赖注入。这种方式使得容器能够灵活地适应不同的对象和配置，并实现对象的解耦和可扩展性。

需要注意的是，尽管反射是实现 IoC 容器的一种常见方式，但并不是唯一的方式。在某些情况下，容器可能使用其他技术或实现方式来创建和管理对象，例如使用字节码操作库或生成动态代理等。不同的容器实现可能使用不同的技术和策略来实现对象的创建和依赖注入。

## **什么是反射**

在Java中，反射（Reflection）是指在运行时动态地获取类的信息并操作类或对象的能力。它允许程序在运行时检查类的属性、方法和构造函数，以及在运行时实例化对象、调用方法和访问/修改字段，而无需在编译时知道这些类的确切信息。

Java的反射API位于java.lang.reflect包中，主要由以下几个核心类组成：

Class类：代表一个类或接口，在运行时可以获取类的各种信息，如类的名称、父类、接口、字段、方法、构造函数等。

Constructor类：表示类的构造函数，可以用于创建类的实例。

Method类：表示类的方法，可以用于调用方法。

Field类：表示类的字段（成员变量），可以用于获取和设置字段的值。

通过使用反射，可以在运行时动态地加载和操作类，以适应不同的场景和需求。一些常见的应用包括：

动态创建对象：使用反射可以在运行时根据类名实例化对象，而不需要提前知道类的具体信息。

调用方法：通过反射可以调用类的方法，包括公共方法、私有方法和静态方法。

访问和修改字段：可以使用反射获取和设置类的字段的值，包括公共字段和私有字段。

动态代理：通过反射可以在运行时创建代理对象，从而实现对其他对象的拦截和增强。

## **动态代理和静态代理**

动态代理和静态代理是两种常见的代理模式，用于在对象之间提供额外的间接访问层。它们的主要区别在于代理类的生成时机和方式。

静态代理：

静态代理是在编译时就已经确定代理类的代码，代理类是在编写代码时手动创建的。

代理类和目标类需要实现相同的接口或继承相同的父类，代理类中的方法会调用目标对象的方法，并在必要的时候添加额外的逻辑。

静态代理的优点是简单易懂，可以在编译时对代理类进行优化，但缺点是需要手动创建代理类，当目标类接口发生变化时，代理类也需要相应修改。

动态代理：

动态代理是在运行时动态生成代理类的代码，不需要手动创建代理类。

Java 提供了两种动态代理的实现方式：基于接口的 JDK 动态代理和基于类的 CGLIB 动态代理。

JDK 动态代理通过 java.lang.reflect.Proxy 类和 java.lang.reflect.InvocationHandler 接口来实现，在运行时动态生成代理类。

CGLIB 动态代理通过继承目标类并生成子类的方式来实现代理。

动态代理的优点是可以减少手动创建代理类的工作，对目标类的代理更加灵活，但缺点是运行时生成代理类的过程可能会带来一些性能损耗。

无论是静态代理还是动态代理，它们的目的都是在目标对象的方法调用前后添加额外的逻辑，例如日志记录、权限验证、事务管理等。选择使用静态代理还是动态代理取决于具体的需求和情况。静态代理适合于代理类数量有限且固定的场景，而动态代理适用于代理类数量较多且需要动态生成的场景。

## **静态代理的几种实现方式**

静态代理是在编译时就已经确定代理类的代码，下面介绍几种常见的静态代理实现方式：

接口方式：

创建一个接口，定义目标对象和代理对象共同实现的方法。

创建目标类实现该接口，实现具体的业务逻辑。

创建代理类实现该接口，代理类持有一个目标对象的引用，并在代理方法中调用目标对象的方法，并在必要时添加额外的逻辑。

在使用时，通过创建代理对象来间接访问目标对象的方法。

继承方式：

创建一个父类，定义目标对象和代理对象共同继承的方法。

创建目标类继承父类，实现具体的业务逻辑。

创建代理类继承父类，代理类持有一个目标对象的引用，并在代理方法中调用目标对象的方法，并在必要时添加额外的逻辑。

在使用时，通过创建代理对象来间接访问目标对象的方法。

包装方式：

创建一个接口或父类，定义目标对象和代理对象共同实现或继承的方法。

创建目标类实现或继承该接口或父类，实现具体的业务逻辑。

创建代理类，代理类持有一个目标对象的引用，并在代理方法中调用目标对象的方法，并在必要时添加额外的逻辑。

在使用时，通过创建代理对象来间接访问目标对象的方法。

这些实现方式都可以通过在代理类中调用目标对象的方法，并在必要时添加额外的逻辑，来实现对目标对象的代理访问。具体选择哪种实现方式取决于具体的需求和设计。无论使用哪种方式，静态代理的核心思想是在目标对象的方法调用前后添加额外的逻辑，以实现对目标对象的增强或控制。

### 结合代码

下面是一个使用接口方式实现静态代理的示例代码：

首先，定义一个接口 Subject，该接口包含了目标对象和代理对象共同实现的方法：

javaCopy codepublic interface Subject {

void doSomething();

}

接下来，创建目标类 RealSubject 实现 Subject 接口：

javaCopy codepublic class RealSubject implements Subject {

public void doSomething() {

System.out.println("RealSubject is doing something.");

}

}

然后，创建代理类 ProxySubject 实现 Subject 接口，代理类持有一个目标对象的引用，并在代理方法中调用目标对象的方法，并在必要时添加额外的逻辑：

class ProxySubject implements Subject {

private Subject realSubject;

public ProxySubject(Subject realSubject) {

this.realSubject = realSubject;

}

public void doSomething() {

System.out.println("ProxySubject is doing something before calling realSubject.");

realSubject.doSomething();

System.out.println("ProxySubject is doing something after calling realSubject.");

}

}

最后，在使用时，通过创建代理对象来间接访问目标对象的方法：

javaCopy codepublic class Main {

public static void main(String[] args) {

Subject realSubject = new RealSubject(); // 创建目标对象

Subject proxySubject = new ProxySubject(realSubject); // 创建代理对象

proxySubject.doSomething(); // 通过代理对象调用目标对象的方法

}

}

在上述代码中，ProxySubject 类是代理类，它持有一个 RealSubject 类的引用，并在 doSomething() 方法中添加了额外的逻辑，即在调用目标对象的方法之前和之后分别输出一些信息。通过创建代理对象 ProxySubject 来间接调用目标对象 RealSubject 的方法，实现了对目标对象的静态代理访问。

### 动态代理

动态代理是在运行时动态生成代理类的代码，不需要手动创建代理类。在 Java 中，有两种常见的动态代理实现方式：基于接口的 JDK 动态代理和基于类的 CGLIB 动态代理。

JDK 动态代理：

JDK 动态代理是通过 java.lang.reflect.Proxy 类和 java.lang.reflect.InvocationHandler 接口来实现的。

首先定义一个接口，作为目标对象和代理对象共同实现的接口。

创建一个实现 InvocationHandler 接口的代理处理器类，重写 invoke() 方法，在其中添加额外的逻辑。

使用 Proxy 类的 newProxyInstance() 方法动态生成代理类的实例。

在使用时，通过代理对象调用目标对象的方法，代理处理器会拦截方法调用并执行相应的额外逻辑。

CGLIB 动态代理：

CGLIB 动态代理通过继承目标类并生成子类的方式来实现代理。

首先定义一个类作为目标类，它不需要实现任何接口。

创建一个 MethodInterceptor 接口的实现类，重写 intercept() 方法，在其中添加额外的逻辑。

使用 Enhancer 类生成目标类的子类，并设置拦截器。

在使用时，通过代理对象调用目标对象的方法，拦截器会拦截方法调用并执行相应的额外逻辑。

以下是一个使用 JDK 动态代理的示例代码：

import java.lang.reflect.InvocationHandler;

import java.lang.reflect.Method;

import java.lang.reflect.Proxy;

public interface Subject {

void doSomething();

}

public class RealSubject implements Subject {

public void doSomething() {

System.out.println("RealSubject is doing something.");

}

}

public class ProxyHandler implements InvocationHandler {

private Object target;

public ProxyHandler(Object target) {

this.target = target;

}

public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {

System.out.println("Proxy is doing something before calling realSubject.");

Object result = method.invoke(target, args);

System.out.println("Proxy is doing something after calling realSubject.");

return result;

}

}

public class Main {

public static void main(String[] args) {

Subject realSubject = new RealSubject(); // 创建目标对象

Subject proxySubject = (Subject) Proxy.newProxyInstance(

realSubject.getClass().getClassLoader(),

realSubject.getClass().getInterfaces(),

new ProxyHandler(realSubject)); // 创建代理对象

proxySubject.doSomething(); // 通过代理对象调用目标对象的方法

}

}

在上述示例中，ProxyHandler 类是实现了 InvocationHandler 接口的代理处理器类，它在 invoke() 方法中添加了额外的逻辑。通过 Proxy 类的 newProxyInstance() 方法动态生成代理类的实例，并在使用时通过代理对象调用目标对象的方法。代理处理器会拦截方法调用并执行相应的额外逻辑。

## **两者的优缺点**

JDK 动态代理和 CGLIB 动态代理各有优缺点，下面是它们的比较：

JDK 动态代理的优点：

基于接口：JDK 动态代理要求目标对象实现一个接口，这使得它更适合于基于接口的代理。

简单易用：使用 JDK 动态代理相对简单，无需引入第三方库，直接使用 Java 标准库中的 Proxy 类和 InvocationHandler 接口即可。

官方支持：JDK 动态代理是 Java 标准库的一部分，由官方提供和维护，稳定可靠。

JDK 动态代理的缺点：

只能代理接口：JDK 动态代理只能代理实现了接口的类，无法对没有实现接口的类进行代理。

性能较低：JDK 动态代理在运行时使用反射机制生成代理类，相比直接调用目标对象的方法，会有一定的性能损耗。

CGLIB 动态代理的优点：

可代理类：CGLIB 动态代理可以代理没有实现接口的类，可以扩展更广泛的代理对象范围。

高性能：CGLIB 动态代理在运行时通过生成目标类的子类来实现代理，无需使用反射，因此性能相对较高。

CGLIB 动态代理的缺点：

需要第三方库：CGLIB 动态代理需要引入第三方库，如 cglib、asm 等。

不支持 final 类和 final 方法：由于 CGLIB 动态代理是通过继承目标类生成子类来实现代理，所以无法代理 final 类和 final 方法。

根据具体的场景和需求，选择适合的动态代理方式。如果目标对象实现了接口且性能要求较高，可以使用 JDK 动态代理；如果目标对象没有实现接口或对性能要求较高，可以考虑使用 CGLIB 动态代理。

## **静态代理和动态代理的优缺点**

静态代理和动态代理各有优缺点，下面是它们的比较：

静态代理的优点：

编译时确定：静态代理在编译时就已经确定代理类的代码，因此运行时效率较高。

控制力强：可以在代理类中灵活地添加额外的逻辑，对目标对象的访问进行控制或增强。

简单易理解：静态代理相对简单，代码结构清晰，易于理解和维护。

静态代理的缺点：

代码冗余：每个目标对象都需要对应一个代理类，当目标对象较多时，会导致代理类的数量增多，代码冗余。

扩展性差：如果需要对目标对象的方法进行扩展或修改，需要修改代理类的代码，不利于代码的灵活性和可扩展性。

静态绑定：静态代理在编译时就已经确定了代理类和目标类的关系，无法动态改变代理的行为。

动态代理的优点：

灵活性强：动态代理是在运行时动态生成代理类的代码，可以根据需要对目标对象的方法进行动态扩展、修改或过滤。

无需手动创建代理类：使用动态代理可以省去手动编写代理类的步骤，简化了开发过程。

可代理多个接口或类：动态代理可以代理实现了接口的类，也可以代理没有实现接口的类，扩展范围更广。

动态代理的缺点：

运行时效率较低：动态代理使用反射机制，在运行时动态生成代理类的代码，相对于静态代理来说会有一定的性能损耗。

复杂性增加：动态代理的实现较为复杂，需要理解反射机制和动态生成代理类的原理。

需要第三方库支持：在 Java 中，使用动态代理需要借助第三方库，如 JDK 的 java.lang.reflect.Proxy 类或 CGLIB 库。

根据具体的需求和场景，选择适合的代理方式。静态代理适用于代理类数量有限且代理逻辑相对简单的情况；动态代理适用于代理类数量较多或需要动态修改代理行为的情况。

## **线程内共享变量的工具类**

在 Java 中，可以使用以下几种工具类来实现线程内共享变量：

ThreadLocal： ThreadLocal 是 Java 提供的一个线程级别的变量容器，它可以在每个线程中存储变量的副本，实现线程内共享。每个线程访问 ThreadLocal 变量时，实际上是访问该线程自己的副本，互不干扰。 示例代码如下：

codepublic class SharedVariableUtil {

private static ThreadLocal<Integer> sharedVariable = new ThreadLocal<>();

public static void setSharedVariable(int value) {

sharedVariable.set(value);

}

public static int getSharedVariable() {

return sharedVariable.get();

}

}

InheritableThreadLocal： InheritableThreadLocal 是 ThreadLocal 的一个子类，它可以实现父线程和子线程之间共享变量的功能。子线程会继承父线程的 InheritableThreadLocal 变量副本。 示例代码如下：

javaCopy codepublic class SharedVariableUtil {

private static InheritableThreadLocal<Integer> sharedVariable = new InheritableThreadLocal<>();

public static void setSharedVariable(int value) {

sharedVariable.set(value);

}

public static int getSharedVariable() {

return sharedVariable.get();

}

}

ThreadLocalRandom： ThreadLocalRandom 是 Java 提供的一个线程级别的随机数生成器，它可以在每个线程中生成随机数，实现线程内独立的随机数生成。 示例代码如下：

javaCopy codepublic class RandomUtil {

public static int getRandomNumber() {

return ThreadLocalRandom.current().nextInt();

}

}

这些工具类可以帮助实现线程内共享变量或线程内独立的变量访问，根据具体的需求选择适合的工具类。注意，这些工具类都是针对线程级别的共享或隔离，并不适用于线程间的共享。如果需要在多个线程之间共享变量，需要考虑使用其他的线程同步机制，如锁或线程安全的集合类等。

## **拦截器和过滤器**

拦截器（Interceptor）和过滤器（Filter）是在 Java Web 应用中常用的两种组件，用于对请求进行拦截和处理。它们在功能上有一些相似之处，但在使用方式和应用场景上有所区别。

拦截器（Interceptor）：

拦截器是针对具体的请求处理器（Handler）而言的，它是基于面向切面编程（AOP）的思想，可以在请求进入处理器前、后以及返回响应时进行拦截和处理。

拦截器是在请求处理器层面上的，可以拦截指定的方法或路径，并对请求进行预处理、后处理以及响应处理等操作。

拦截器可以对请求进行更精细的控制，可以通过获取请求参数、修改响应内容等方式对请求进行干预和增强。

在 Java Web 应用中，常见的拦截器是通过框架提供的拦截器接口（如 Spring MVC 的 HandlerInterceptor 接口）来实现的。

### 以下是一个简单的示例代码，演示如何使用拦截器（Interceptor）实现请求拦截和处理：

首先，创建一个实现了 HandlerInterceptor 接口的拦截器类：

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

import org.springframework.web.servlet.HandlerInterceptor;

import org.springframework.web.servlet.ModelAndView;

public class MyInterceptor implements HandlerInterceptor {

@Override

public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler)

throws Exception {

// 在请求处理之前执行的逻辑

System.out.println("Pre-handle logic");

// 返回 true 表示继续处理请求，返回 false 表示拦截请求

return true;

}

@Override

public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler,

ModelAndView modelAndView) throws Exception {

// 在请求处理之后、视图渲染之前执行的逻辑

System.out.println("Post-handle logic");

}

@Override

public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler,

Exception ex) throws Exception {

// 在请求完成后执行的逻辑，包括视图渲染完成后

System.out.println("After-completion logic");

}

}

然后，在配置文件中注册拦截器并指定拦截的路径：

xmlCopy code<bean id="myInterceptor" class="com.example.MyInterceptor" />

<mvc:interceptors>

<mvc:interceptor>

<mvc:mapping path="/\*\*" /> <!-- 拦截所有请求 -->

<bean class="com.example.MyInterceptor" />

</mvc:interceptor>

</mvc:interceptors>

在上述示例中，MyInterceptor 实现了 HandlerInterceptor 接口，并重写了其中的三个方法：preHandle、postHandle 和 afterCompletion。在 preHandle 方法中，我们可以添加拦截请求的逻辑，比如进行身份验证、权限检查等。在 postHandle 方法中，我们可以对请求处理的结果进行后续处理，比如添加一些公共的数据到视图中。在 afterCompletion 方法中，我们可以进行一些清理操作，比如释放资源等。

在配置文件中，我们将拦截器 MyInterceptor 注册为一个 Bean，并使用 <mvc:interceptor> 配置进行拦截器的设置，指定要拦截的路径。在这个示例中，我们使用 <mvc:mapping path="/\*\*" /> 拦截所有请求。

这样，在应用程序运行时，当有请求进入时，拦截器会根据配置对请求进行拦截，并执行相应的逻辑。

### 过滤器（Filter）：

过滤器是在请求到达目标资源（如 Servlet）之前或之后进行拦截和处理的组件。

过滤器是在请求到达 Servlet 容器前进行拦截的，可以对请求进行预处理、后处理以及响应处理等操作。

过滤器对请求的拦截和处理是基于请求路径的，可以对指定的 URL 或 URL 模式进行拦截和过滤。

过滤器通常用于实现一些通用的功能，如请求编码转换、请求参数过滤、请求日志记录等。

在 Java Web 应用中，过滤器是通过实现 Servlet 规范中的 Filter 接口来实现的。

总结： 拦截器和过滤器都用于对请求进行拦截和处理，但拦截器更关注于请求处理器层面上的拦截和增强，而过滤器更关注于请求到达目标资源前的拦截和处理。拦截器可以实现更细粒度的控制和处理，而过滤器更适用于实现通用的请求处理逻辑。在具体应用中，可以根据需求选择合适的拦截器或过滤器来实现相应的功能。

## **spring事务**

Spring 提供了对事务管理的支持，它简化了在应用程序中实现事务的过程。Spring 事务管理具有以下特点：

声明式事务管理：Spring 提供了声明式事务管理的机制，通过在配置文件中或使用注解来声明事务的边界，将事务管理从业务逻辑中分离出来，使得代码更加清晰简洁。

支持多种事务管理器：Spring 支持多种事务管理器，包括 JDBC、Hibernate、JPA、JTA 等，可以根据应用的需求选择合适的事务管理器。

编程式事务管理：除了声明式事务管理，Spring 也支持编程式事务管理，允许在代码中以编程的方式管理事务，对于一些特殊场景或需要更细粒度控制事务的情况下，可以使用编程式事务管理。

事务传播行为：Spring 定义了多种事务传播行为，用于控制事务的边界和事务之间的关系，例如 REQUIRED、REQUIRES\_NEW、NESTED 等。事务传播行为指定了一个事务方法调用另一个事务方法时事务如何进行传播和合并。

异常处理和回滚策略：Spring 允许配置事务的异常处理和回滚策略，可以指定特定异常触发回滚，或者根据异常类型决定是回滚还是提交事务。

AOP 和事务：Spring 的事务管理是通过 AOP 实现的，它利用了 Spring AOP 的机制，在运行时为事务方法创建代理，从而实现事务的控制和管理。

使用 Spring 的事务管理，可以保证数据库操作的一致性和完整性，减少手动处理事务的工作量，提供了更高级别的事务管理机制，使开发者可以专注于业务逻辑的实现而无需过多关注事务的细节。

## **static什么时候修饰一个变量**

在Java中，static 关键字用于修饰变量时，表示该变量为静态变量（类变量），即在类级别上共享的变量，而不是每个对象实例独立拥有的变量。static 修饰的变量有以下几种常见的使用场景：

静态常量：使用 static final 修饰的变量被称为静态常量，一般使用大写字母命名，并在声明时进行初始化，一旦初始化后不可再修改。静态常量在类加载时被初始化，并且在整个程序运行期间保持不变。例如：

javaCopy codepublic class MathUtil {

public static final double PI = 3.14159;

}

可以通过 MathUtil.PI 来访问静态常量。

共享变量：静态变量在类的所有对象实例之间共享，可以在不创建对象实例的情况下直接通过类名访问。静态变量在类加载时被初始化，且只有一个副本存在于内存中。例如：

javaCopy codepublic class Counter {

private static int count = 0;

public static void increment() {

count++;

}

public static int getCount() {

return count;

}

}

可以通过 Counter.increment() 来增加计数器的值，通过 Counter.getCount() 来获取计数器的值。

工具类中的方法参数：有时，静态方法需要操作的数据不属于任何对象，这时可以将数据作为静态变量传递给静态方法。例如：

javaCopy codepublic class StringUtil {

public static boolean isNullOrEmpty(String str) {

return str == null || str.isEmpty();

}

}

在 StringUtil.isNullOrEmpty() 方法中，str 参数是一个静态变量，不依赖于任何对象实例。

需要注意的是，静态变量与对象实例无关，因此它们不可以直接访问非静态的成员变量或方法。另外，静态变量在内存中存在于静态区，生命周期比对象实例长，直到程序退出才会被销毁。

## **static什么时候修饰一个方法**

在Java中，static 关键字用于修饰方法时，表示该方法为静态方法（类方法），即不依赖于对象实例，可以直接通过类名调用。静态方法具有以下几种常见的使用场景：

工具方法：静态方法通常用于实现一些通用的功能，不需要访问对象的状态或实例变量。它们在类加载时就被加载到内存中，可以通过类名直接调用，无需创建对象实例。例如：

javaCopy codepublic class MathUtil {

public static int add(int a, int b) {

return a + b;

}

}

可以通过 MathUtil.add(2, 3) 来调用静态方法。

工厂方法：静态方法可以作为工厂方法，用于创建和返回类的对象实例。在工厂方法中，不需要通过 new 关键字显式创建对象，可以在方法内部创建并返回对象实例。例如：

javaCopy codepublic class ShapeFactory {

public static Shape createShape(String type) {

if (type.equalsIgnoreCase("circle")) {

return new Circle();

} else if (type.equalsIgnoreCase("rectangle")) {

return new Rectangle();

} else {

return null;

}

}

}

可以通过 ShapeFactory.createShape("circle") 来创建 Circle 对象实例。

辅助方法：静态方法可以作为辅助方法，用于在类的内部提供额外的功能支持，例如计算、转换、验证等操作。这些方法可以在类的其他方法中被直接调用，无需创建对象实例。例如：

javaCopy codepublic class StringUtil {

public static boolean isNullOrEmpty(String str) {

return str == null || str.isEmpty();

}

}

在 StringUtil 类中，isNullOrEmpty() 方法是一个静态方法，用于判断字符串是否为 null 或为空。

需要注意的是，静态方法无法直接访问非静态的成员变量或方法，因为它们不依赖于对象实例。此外，静态方法在内存中存在于静态区，生命周期比对象实例长，直到程序退出才会被销毁。静态方法可以直接通过类名调用，也可以通过对象实例调用，但建议使用类名调用以表明方法的静态性质。

## **final什么时候修饰一个类，变量，方法**

final 关键字可以用于修饰类、变量和方法，具体的使用场景如下：

修饰类：

当一个类被声明为 final 时，表示该类是最终的，不能被继承。这意味着其他类无法扩展该类，防止对该类的修改和继承破坏原有设计。

final 类通常用于不希望被继承的情况，例如工具类、不可变类等。典型的例子是 Java 核心库中的 java.lang.String 类，它被声明为 final，不能被子类继承。

修饰变量：

当一个变量被声明为 final 时，表示该变量是常量，一旦赋值后就不能再改变其值。

final 变量必须在声明时或构造方法中进行初始化，并且不能再被赋予新的值。在方法内部，也可以使用 final 关键字来声明局部变量，保证变量值的不可变性。

常量通常使用全大写字母命名，多个单词之间用下划线分隔。例如：final int MAX\_SIZE = 100;

修饰方法：

当一个方法被声明为 final 时，表示该方法是最终的，不能被子类重写（覆盖）。

final 方法在父类中已经实现了最终的逻辑，子类无法对其进行修改。这样可以确保方法的行为不会被子类改变，提供了设计的安全性和稳定性。

final 方法经常与 final 类一起使用，以防止子类修改父类的行为。

需要注意的是，final 类、变量和方法具有不可修改或不可重写的特性，因此在使用时需要慎重考虑，确保在设计和实现时符合需求和预期。

## **Java中的几种修饰符**

在Java中，有以下几种修饰符（访问修饰符和非访问修饰符）：

访问修饰符（Access Modifiers）：

public：可以被任何类访问，没有访问限制。

protected：可以被同一包内的类访问，以及不同包下的子类访问。

private：只能被同一类内部访问，其他类无法访问。

默认（不加修饰符）：可以被同一包内的类访问，但对于不同包下的类没有访问权限。

非访问修饰符（Non-access Modifiers）：

final：表示不可继承（对于类）或不可覆盖（对于方法和变量）。

abstract：用于抽象类和抽象方法，表示类不能被实例化，或方法没有具体实现，需要在子类中实现。

static：表示静态变量或静态方法，可以通过类名直接访问，而不需要创建类的实例。

synchronized：用于多线程编程，表示方法或代码块在同一时间只能被一个线程访问。

volatile：用于多线程编程，确保变量在多个线程之间的可见性。

transient：用于对象序列化，表示某个字段不会被序列化保存。

native：表示方法的实现由外部的本地代码（如C、C++）提供。

strictfp：表示浮点数计算使用严格的浮点规范，保证不同平台上的一致性。

这些修饰符可以应用于类、接口、变量、方法等。它们提供了不同的访问控制级别和行为约束，用于控制类和成员的可见性、继承性、多态性、线程安全性等特性。在设计和编写Java程序时，正确使用这些修饰符可以增强代码的可维护性和可靠性。

## **Java中的vailaste**

在 Java 中，"validate" 是一个常见的词汇，用于表示验证、检查或确认数据的有效性、正确性或合法性。在软件开发中，数据验证是一项重要的任务，可以确保输入的数据满足特定的规则、条件或约束。

在 Java 中，我们可以使用各种方式进行数据验证，例如：

条件判断语句：使用 if、else、switch 等语句来检查数据是否满足特定的条件。

正则表达式：使用正则表达式来验证字符串是否匹配特定的模式。Java 提供了 java.util.regex 包来支持正则表达式的使用。

异常处理：使用异常机制来捕获并处理数据验证过程中可能出现的异常情况。例如，当数据不满足验证条件时，可以抛出自定义的异常或使用预定义的异常类（如 IllegalArgumentException、IllegalStateException 等）。

校验框架：使用第三方校验框架，如 Hibernate Validator、Apache Commons Validator 等，可以简化数据验证的编写和管理过程，并提供丰富的验证规则和错误处理机制。

例如，以下示例展示了一个简单的数据验证方法，用于验证一个年龄是否大于等于18岁：

javaCopy codepublic class Validator {

public static boolean validateAge(int age) {

if (age >= 18) {

return true;

} else {

return false;

}

}

public static void main(String[] args) {

int age = 20;

if (validateAge(age)) {

System.out.println("年龄验证通过");

} else {

System.out.println("年龄不符合要求");

}

}

}

在上述示例中，validateAge() 方法接收一个整数参数 age，通过条件判断验证该年龄是否大于等于18岁。main() 方法中使用该验证方法来验证一个年龄，并根据验证结果输出相应的消息。

请注意，数据验证的具体方式和实现取决于具体的需求和场景。在实际开发中，我们应根据数据的类型、验证规则和业务逻辑等因素选择适合的验证方法和策略。

## **java中怎么定义泛型类**

在 Java 中，可以通过使用泛型来定义泛型类。泛型类是具有类型参数的类，可以在类定义中使用这些参数来表示通用的类型。

泛型类的定义方式如下：

javaCopy codepublic class GenericClass<T> {

private T data;

public GenericClass(T data) {

this.data = data;

}

public T getData() {

return data;

}

public void setData(T data) {

this.data = data;

}

}

在上述示例中，GenericClass<T> 是一个泛型类，其中 <T> 表示类型参数。在类定义中，可以使用类型参数 T 来声明成员变量、方法参数和返回值的类型。

通过在创建泛型类的对象时指定类型参数，可以实现对不同类型的对象进行操作。例如：

javaCopy codeGenericClass<Integer> intObj = new GenericClass<>(10);

intObj.setData(20);

System.out.println(intObj.getData()); // 输出: 20

GenericClass<String> stringObj = new GenericClass<>("Hello");

stringObj.setData("World");

System.out.println(stringObj.getData()); // 输出: World

在上述示例中，首先创建了一个 GenericClass<Integer> 类型的对象 intObj，并通过 setData() 方法设置了整数值。然后创建了一个 GenericClass<String> 类型的对象 stringObj，并通过 setData() 方法设置了字符串值。通过调用 getData() 方法，可以获取设置的值并进行输出。

泛型类的优势在于可以提供类型安全性和代码重用性，使得代码更加灵活和可扩展。可以通过泛型类来处理各种不同类型的数据，而无需针对每种类型编写单独的类或方法。

## **泛型方法**

在Java中，我们可以使用泛型方法为特定的方法引入类型参数。泛型方法是指在方法定义中使用泛型类型参数的方法。通过泛型方法，我们可以在方法级别上实现类型的灵活性和安全性。

泛型方法的定义方式如下：

javaCopy codepublic <T> void genericMethod(T data) {

// 方法逻辑

}

在上述示例中，<T> 表示类型参数，可以在方法参数、返回值或方法体中使用。通过泛型方法，我们可以根据传入的参数类型来确定方法的行为。

以下是一个示例，展示如何使用泛型方法来打印不同类型的数组元素：

javaCopy codepublic class GenericMethodExample {

public <T> void printArray(T[] array) {

for (T element : array) {

System.out.println(element);

}

}

public static void main(String[] args) {

Integer[] intArray = {1, 2, 3, 4, 5};

String[] stringArray = {"Hello", "World"};

GenericMethodExample example = new GenericMethodExample();

example.printArray(intArray); // 打印整数数组

example.printArray(stringArray); // 打印字符串数组

}

}

在上述示例中，printArray() 方法是一个泛型方法，使用了类型参数 <T>。通过传入不同类型的数组参数，我们可以在方法内部打印数组的元素。

泛型方法可以带有多个类型参数，并且可以与泛型类一起使用，以提供更灵活的类型支持。

需要注意的是，泛型方法的类型参数在方法调用时可以根据实际情况进行类型推断，无需显式指定类型参数。例如，可以使用 example.<Integer>printArray(intArray); 显式指定类型参数，或者直接使用 example.printArray(intArray); 让编译器根据参数类型进行类型推断。

## **spring中的常用注解**

Spring 框架提供了许多注解来简化开发和配置。以下是一些常用的 Spring 注解：

@Component: 用于标识一个类为组件类，让 Spring 自动扫描并创建对象实例。

@Repository: 用于标识一个类为数据访问层（DAO）的组件类。

@Service: 用于标识一个类为业务逻辑层（Service）的组件类。

@Controller: 用于标识一个类为控制器层（Controller）的组件类。

@Autowired: 自动装配依赖注入，通过类型匹配自动将依赖对象注入到标记的字段、构造函数或方法参数中。

@Qualifier: 指定一个唯一的 bean 名称，用于与 @Autowired 配合使用，解决多个相同类型的依赖对象的自动装配问题。

@Value: 用于注入配置属性的值，可以通过 ${} 占位符来引用配置文件中的属性。

@Configuration: 声明一个类为配置类，用于定义和组织 Bean 的创建和配置。

@Bean: 在配置类中使用，用于声明一个方法返回的对象为一个 Bean，并由 Spring 管理和维护。

@Scope: 定义 Bean 的作用域，包括单例（singleton）、原型（prototype）等。

@RequestMapping: 用于映射 HTTP 请求到控制器的处理方法，指定请求的 URL 和 HTTP 方法。

@PathVariable: 用于将请求的 URL 中的参数值绑定到方法参数中。

@RequestParam: 用于将请求的查询参数绑定到方法参数中。

@ResponseBody: 标识返回的结果直接作为响应体，而不是跳转到视图。

@PathVariable: 用于将请求的 URL 中的参数值绑定到方法参数中。

@ExceptionHandler: 定义异常处理方法，用于处理特定类型的异常。

这些是 Spring 中的一些常用注解，用于简化配置、依赖注入和请求处理等操作。具体使用哪些注解取决于具体的需求和场景。可以根据实际情况选择合适的注解来简化开发工作。

## **懒汉模式的注解**

在Java中，"懒汉模式"是一种延迟加载的单例设计模式。它的特点是在首次使用时才会创建对象实例，而不是在类加载时就创建实例。

在Spring框架中，并没有专门的注解来标记懒汉模式。然而，你可以使用其他注解和配置方式来实现懒汉模式的单例对象。

一种常见的方式是结合@Lazy注解和@Component注解来实现懒加载的单例对象。@Lazy注解用于延迟加载组件，而@Component注解用于将类标识为Spring组件。通过将这两个注解结合使用，可以在首次使用时创建单例对象。

以下是一个示例代码：

@Lazy

@Component

public class LazySingleton {

private static LazySingleton instance;

private LazySingleton() {

// 私有构造函数

}

public static LazySingleton getInstance() {

if (instance == null) {

instance = new LazySingleton();

}

return instance;

}

}

在上述示例中，@Lazy注解标记了LazySingleton类，表示它是一个延迟加载的组件。LazySingleton类的构造函数为私有，确保只能通过getInstance()方法获取实例。

当你在其他类中注入LazySingleton对象时，它会在首次使用时才会被实例化：

@Component

public class MyComponent {

@Autowired

private LazySingleton lazySingleton;

// ...

}

通过这种方式，你可以在需要使用懒汉模式的单例对象时，通过注入的方式来获取实例，并且保证实例的延迟加载。

需要注意的是，Spring还提供了其他的注解和配置方式来实现对象的延迟加载，如使用@Scope("singleton")和LazyInit属性等。具体选择哪种方式取决于你的具体需求和项目的架构。

## **springmvc的常用注解**

Spring MVC 是基于 Spring 框架的一种 Web 应用开发框架，用于开发和管理 Web 应用程序。下面是一些常用的 Spring MVC 注解：

@Controller: 用于标识一个类为控制器类，处理用户请求并返回相应的结果。

@RequestMapping: 用于映射 HTTP 请求到控制器的处理方法，指定请求的 URL 和 HTTP 方法。

@GetMapping: 简化版的 @RequestMapping，用于映射 GET 请求。

@PostMapping: 简化版的 @RequestMapping，用于映射 POST 请求。

@PutMapping: 简化版的 @RequestMapping，用于映射 PUT 请求。

@DeleteMapping: 简化版的 @RequestMapping，用于映射 DELETE 请求。

@PathVariable: 用于将请求的 URL 中的参数值绑定到方法参数中。

@RequestParam: 用于将请求的查询参数绑定到方法参数中。

@RequestBody: 用于将请求的请求体内容绑定到方法参数中。

@ResponseBody: 标识方法的返回值直接作为响应体返回，而不是跳转到视图。

@ModelAttribute: 用于将请求参数绑定到方法参数或方法返回值中的模型属性上。

@Valid: 用于启用方法参数的验证。

@Validated: 用于启用方法参数的分组验证。

@SessionAttribute: 用于将模型属性存储到会话中。

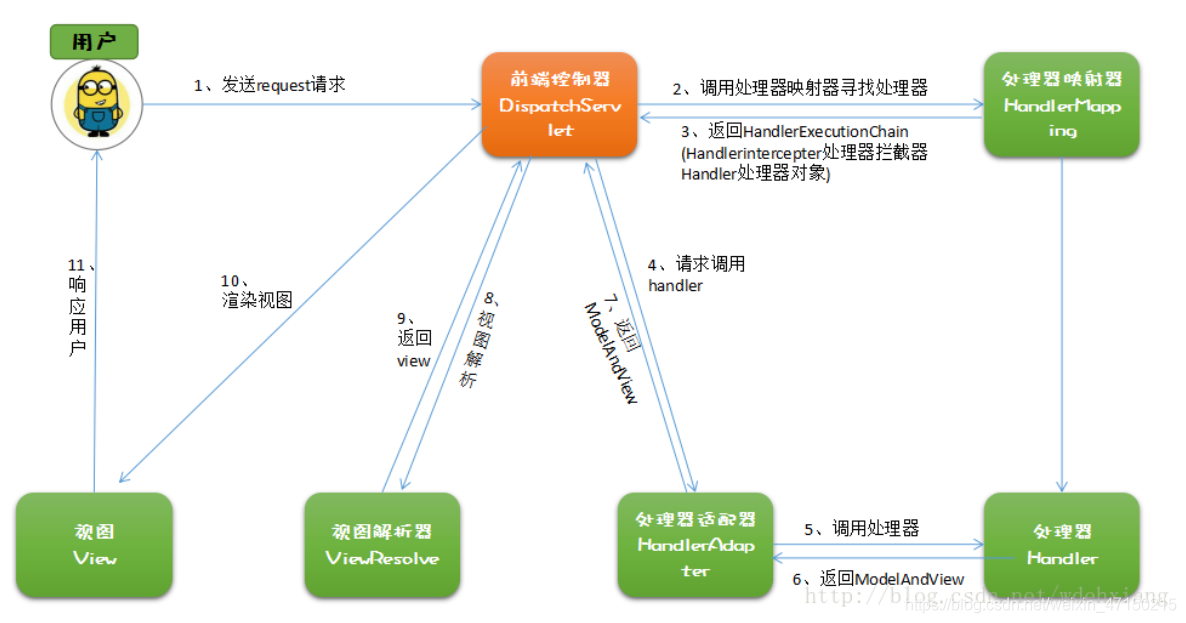
@SessionAttributes: 用于指定模型属性存储到会话中的属性名列表。

@InitBinder: 用于配置表单数据绑定的初始化。

这些注解可以在控制器类或控制器方法上使用，用于定义请求的映射关系、参数绑定、返回结果处理等操作。使用这些注解可以简化开发过程，提高代码的可读性和可维护性。

需要注意的是，Spring MVC 还有其他的注解和配置方式，用于处理拦截器、异常处理、文件上传等功能。具体使用哪些注解取决于你的具体需求和项目的架构。

## **mvc执行流程**



用户发送请求到前端控制器（DispatcherServlet）。

前端控制器请求处理器映射器（HandlerMapping）去查找处理器（Handler）。

找到以后处理器映射器（HandlerMappering）向前端控制器返回执行链（HandlerExecutionChain）。

前端控制器（DispatcherServlet）调用处理器适配器（HandlerAdapter）去执行处理器（Handler）。

处理器适配器去执行Handler。

处理器执行完给处理器适配器返回ModelAndView。

处理器适配器向前端控制器返回ModelAndView。

前端控制器请求视图解析器（ViewResolver）去进行视图解析。

视图解析器向前端控制器返回View。

前端控制器对视图进行渲染。

前端控制器向用户响应结果。

## **mybatis配置类里的常用标签**

在 MyBatis 的配置文件中，常用的标签包括：

<configuration>：MyBatis 配置文件的根标签，用于包含所有的配置信息。

<properties>：用于定义属性配置，可以在配置文件中使用占位符 ${} 引用属性值。

<typeAliases>：用于定义类型别名，将 Java 类型映射为短名称，简化映射配置。

<environments>：用于定义数据库环境，包括数据源配置和事务管理器配置。

<dataSource>：用于配置数据源，包括连接池的配置和数据库连接信息。

<mappers>：用于引入 Mapper 接口或 Mapper XML 文件的配置。

<mapper>：用于配置 Mapper 接口或 Mapper XML 文件的路径。

<plugins>：用于配置插件，可以在 SQL 执行过程中拦截、修改或扩展功能。

<settings>：用于配置 MyBatis 的全局属性和设置，如缓存配置、日志配置等。

<typeHandlers>：用于配置类型处理器，用于将 Java 类型和数据库类型进行转换。

这些是 MyBatis 配置文件中常用的标签，通过配置这些标签，可以实现对数据源、事务管理、类型别名、Mapper 文件等的配置和管理。根据具体的需求，可以灵活地配置这些标签来满足项目的要求。

## **一对多怎么映射**

在 MyBatis 中实现一对多关系的映射，可以使用两种方式：嵌套查询（Nested Queries）和嵌套结果（Nested Results）。

嵌套查询（Nested Queries）：

在父实体的映射配置中，通过嵌套查询的方式查询关联的子实体。

使用 <select> 标签定义子实体的查询语句，并使用 parentId 或其他关联属性作为参数。

在父实体的映射配置中，使用 <collection> 标签关联子实体查询结果，并指定 column 和 select 属性。

示例代码：

xmlCopy code<resultMap id="parentResultMap" type="ParentEntity">

<id property="id" column="parent\_id" />

<result property="name" column="parent\_name" />

<collection property="children" ofType="ChildEntity">

<id property="id" column="child\_id" />

<result property="name" column="child\_name" />

<select property="children" column="parent\_id" resultMap="childResultMap"/>

</collection>

</resultMap>

<resultMap id="childResultMap" type="ChildEntity">

<id property="id" column="child\_id" />

<result property="name" column="child\_name" />

<result property="parentId" column="parent\_id" />

</resultMap>

<select id="getParentsWithChildren" resultMap="parentResultMap">

SELECT p.id as parent\_id, p.name as parent\_name,

c.id as child\_id, c.name as child\_name, c.parent\_id

FROM parents p

LEFT JOIN children c ON p.id = c.parent\_id

</select>

嵌套结果（Nested Results）：

在父实体的映射配置中，通过嵌套结果的方式关联子实体的结果。

使用 <association> 标签关联子实体的结果，并指定 property 和 javaType 属性。

在子实体的映射配置中，使用 <id> 和 <result> 标签定义属性的映射关系。

示例代码：

xmlCopy code<resultMap id="parentResultMap" type="ParentEntity">

<id property="id" column="parent\_id" />

<result property="name" column="parent\_name" />

<association property="children" javaType="java.util.List" resultMap="childResultMap"/>

</resultMap>

<resultMap id="childResultMap" type="ChildEntity">

<id property="id" column="child\_id" />

<result property="name" column="child\_name" />

<result property="parentId" column="parent\_id" />

</resultMap>

<select id="getParentsWithChildren" resultMap="parentResultMap">

SELECT p.id as parent\_id, p.name as parent\_name,

c.id as child\_id, c.name as child\_name, c.parent\_id

FROM parents p

LEFT JOIN children c ON p.id = c.parent\_id

</select>

以上示例展示了一对多关系的映射配置，通过使用嵌套查询或嵌套结果的方式，可以将父实体与多个子实体关联起来，并实现数据的查询和映射。具体选择哪种方式取决于你的需求和数据结构的设计。

## **Java中多对一怎么映射**

java 中实现多对一关系的映射，可以使用以下方式：

使用外键关联：

在多的一方实体中，添加一个外键字段，用于关联到一的一方实体的主键。

在多的一方实体中，添加一个对一的一方实体的引用字段，并使用 @ManyToOne 注解进行关联。

在多的一方实体中，使用 @JoinColumn 注解指定外键字段的映射关系。

示例代码：

javaCopy codepublic class ManyEntity {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

@ManyToOne

@JoinColumn(name = "one\_entity\_id")

private OneEntity oneEntity;

// getter and setter

}

使用关联表：

在关系表中，添加一个外键字段分别关联到多的一方实体和一的一方实体的主键。

在多的一方实体中，添加一个对一的一方实体的集合字段，并使用 @OneToMany 注解进行关联。

在一的一方实体中，添加一个对多的一方实体的集合字段，并使用 @OneToMany 注解进行关联。

示例代码：

javaCopy codepublic class OneEntity {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

@OneToMany(mappedBy = "oneEntity")

private List<ManyEntity> manyEntities;

// getter and setter

}

public class ManyEntity {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

@ManyToOne

@JoinColumn(name = "one\_entity\_id")

private OneEntity oneEntity;

// getter and setter

}

以上示例展示了多对一关系的映射方式。通过使用外键关联或关联表的方式，可以将多个实体关联到一个实体，并实现数据的查询和映射。具体选择哪种方式取决于你的需求和数据结构的设计。

## **http的请求协议**

HTTP（Hypertext Transfer Protocol）是一种用于传输超文本的应用层协议。它是在 Web 开发中最为常见的协议之一，用于客户端和服务器之间的通信。

HTTP 请求协议定义了客户端向服务器发送请求的格式和规则，包括以下几个主要部分：

请求行（Request Line）：包含请求方法、请求的资源路径和协议版本，格式为 Method URL HTTPVersion，例如：

bashCopy code

GET /example HTTP/1.1

请求头部（Request Headers）：包含了关于请求的一些附加信息，如用户代理、内容类型、请求的来源等，每个头部以键值对的形式表示，例如：

makefileCopy codeHost: www.example.com

User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/88.0.4324.150 Safari/537.36

Content-Type: application/json

空行：请求头部和请求体之间必须有一个空行，用于分隔请求头部和请求体。

请求体（Request Body）：可选的，一般用于 POST、PUT 等请求方法，用于传递请求的参数或数据。

HTTP 请求的常见方法（Method）包括：

GET：请求获取指定资源。

POST：提交数据，用于创建新资源或提交数据到服务器处理。

PUT：更新指定资源，用于替换指定资源或创建新资源。

DELETE：删除指定资源。

PATCH：对资源进行局部更新。

以上是 HTTP 请求协议的基本组成部分。通过遵循这些规则和格式，客户端可以向服务器发送请求，服务器则根据请求进行处理并返回相应的响应。

## **http协议**

HTTP（Hypertext Transfer Protocol）是一种用于传输超文本的应用层协议。它是在 Web 开发中最为常见的协议之一，用于客户端和服务器之间的通信。

HTTP 协议的主要特点包括：

简单性：HTTP 协议采用简单的请求-响应模型，客户端发送请求，服务器返回响应。

无状态性：HTTP 协议是无状态的，每个请求与响应之间是独立的，服务器不会保留客户端的状态信息。

可扩展性：HTTP 协议允许通过头部字段进行扩展，可以定义和传递自定义的信息。

基于文本：HTTP 协议使用文本格式进行数据传输，请求和响应的内容都是以可读的文本形式表示。

HTTP 协议的基本流程如下：

客户端向服务器发送请求，请求包括请求方法（GET、POST、PUT 等）、请求的资源路径（URL）以及协议版本等信息。

服务器接收到请求后，解析请求并根据请求进行相应的处理，可以是获取资源、处理业务逻辑等。

服务器处理完请求后，生成响应，响应包括状态码、响应头部和响应体。

服务器将响应发送给客户端，客户端接收到响应后进行解析处理，根据响应进行相应的操作，如显示网页内容、处理返回的数据等。

HTTP 协议支持多种请求方法，常见的包括：

GET：请求获取指定资源。

POST：提交数据，用于创建新资源或提交数据到服务器处理。

PUT：更新指定资源，用于替换指定资源或创建新资源。

DELETE：删除指定资源。

PATCH：对资源进行局部更新。

HTTP 协议还支持状态码表示请求和响应的状态，常见的状态码包括 200 OK（请求成功）、404 Not Found（资源未找到）、500 Internal Server Error（服务器内部错误）等。

通过遵循 HTTP 协议，客户端和服务器可以进行有效的通信，实现数据的请求、传输和响应。

## **状态码**

HTTP 协议中的状态码用于表示客户端向服务器发送请求后，服务器对请求的处理结果进行反馈。每个状态码都具有特定的含义，用于指示请求的成功、失败或其他特定情况。

下面是一些常见的 HTTP 状态码及其含义：

1xx（信息类状态码）：表示请求已被接收，需要进一步处理。

100 Continue：请求已被服务器接收，客户端可以继续发送请求的剩余部分。

101 Switching Protocols：服务器正在根据客户端的请求切换协议。

2xx（成功类状态码）：表示请求已成功被服务器接收、理解和处理。

200 OK：请求成功，服务器返回所请求的资源。

201 Created：请求成功，服务器创建了新的资源。

204 No Content：请求成功，但响应中不包含实体内容。

3xx（重定向类状态码）：表示需要进一步操作以完成请求。

301 Moved Permanently：资源的 URL 已永久变更，需要更新书签或链接。

302 Found：资源的 URL 临时变更，客户端应继续使用原始 URL。

304 Not Modified：客户端缓存的资源未变更，可以直接使用缓存。

4xx（客户端错误类状态码）：表示请求包含错误或无法被服务器理解。

400 Bad Request：请求无效或不合法。

401 Unauthorized：请求要求身份验证。

404 Not Found：请求的资源未找到。

5xx（服务器错误类状态码）：表示服务器在处理请求时发生了错误。

500 Internal Server Error：服务器内部错误。

503 Service Unavailable：服务器当前无法处理请求，一般为临时性状态。

这些状态码只是一小部分常见的状态码，HTTP 协议还定义了更多的状态码以满足不同的场景和需求。了解这些状态码的含义可以帮助开发者诊断和调试与服务器通信的问题，并根据不同的状态码采取相应的处理措施。

415 是 HTTP 协议中的状态码之一，表示"Unsupported Media Type"（不支持的媒体类型）。

当客户端向服务器发送请求时，如果请求中指定的媒体类型（Media Type）不被服务器支持或接受，服务器将返回 415 状态码作为响应。

这通常发生在以下情况下：

客户端发送的请求中包含的 Content-Type（请求头部字段）指定的媒体类型不被服务器支持。

客户端发送的请求中的实体内容（请求体）的媒体类型不被服务器支持。

服务器返回 415 状态码时，通常还会在响应中包含一条错误信息或说明，帮助客户端了解导致此错误的原因。

对于客户端来说，当收到 415 状态码时，可能需要检查请求中指定的媒体类型是否正确，并根据服务器的要求进行相应的调整，以确保请求能够被正确处理。

总之，415 状态码表示客户端发送的请求包含的媒体类型不被服务器支持，需要客户端根据服务器的要求进行相应的调整。

## **请求头和响应头**

请求头（Request Headers）和响应头（Response Headers）是 HTTP 协议中的一部分，用于在请求和响应中传递附加的信息和元数据。

请求头（Request Headers）是客户端发送给服务器的请求中包含的头部字段集合，用于描述请求的附加信息，例如：

User-Agent：指定客户端的用户代理（浏览器、操作系统等）。

Content-Type：指定请求体的媒体类型。

Accept：指定客户端可接受的响应内容类型。

Authorization：指定请求的身份验证凭证。

Cookie：指定客户端发送的 Cookie 信息。

响应头（Response Headers）是服务器发送给客户端的响应中包含的头部字段集合，用于描述响应的附加信息，例如：

Content-Type：指定响应体的媒体类型。

Content-Length：指定响应体的长度。

Cache-Control：指定响应的缓存策略。

Set-Cookie：指定服务器发送的 Cookie 信息。

Location：指定重定向的目标 URL。

请求头和响应头都由多个键值对组成，每个键值对表示一个头部字段和其对应的值。这些头部字段提供了丰富的功能和扩展性，用于控制请求和响应的行为、格式和内容。

客户端在发送请求时可以自定义请求头，服务器在发送响应时也可以自定义响应头。通过合理设置请求头和响应头，可以实现各种功能，例如控制缓存、实现身份验证、指定内容类型等。

## **设置编码格式**

在 HTTP 协议中，可以通过设置请求头和响应头来指定编码格式。

请求编码格式：

请求头中的 "Content-Type" 字段用于指定请求体的媒体类型和字符编码。常见的设置为 "Content-Type: text/plain; charset=utf-8"，其中 "charset=utf-8" 指定了使用 UTF-8 编码。

响应编码格式：

响应头中的 "Content-Type" 字段用于指定响应体的媒体类型和字符编码。常见的设置为 "Content-Type: text/html; charset=utf-8"，其中 "charset=utf-8" 指定了使用 UTF-8 编码。

具体地，使用 Java Servlet 编程时，可以通过设置 HttpServletResponse 对象的响应头来指定编码格式。以下是示例代码：

javaCopy code// 设置请求编码格式

request.setCharacterEncoding("UTF-8");

// 设置响应编码格式

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

上述代码中，request 和 response 分别是 HttpServletRequest 和 HttpServletResponse 对象。

请注意，设置编码格式的位置要根据实际情况而定。通常，在处理请求之前设置请求编码格式，而在生成响应之前设置响应编码格式。这样可以确保请求和响应的数据正确地进行编码和解码。

## **Content-Type的参数**

Content-Type 是 HTTP 协议中的一个请求头和响应头字段，用于指定传输的数据的媒体类型和字符编码。它由两个部分组成：媒体类型和参数。

媒体类型（Media Type）是 Content-Type 字段的主要部分，它指定了数据的类型。常见的媒体类型包括：

text/plain：纯文本类型。

text/html：HTML 文档类型。

application/json：JSON 数据类型。

application/xml：XML 数据类型。

multipart/form-data：用于文件上传的表单数据类型。

参数（Parameters）是 Content-Type 字段的附加部分，用于进一步细化数据类型的特性。参数通常以键值对的形式出现，用分号进行分隔。常见的参数包括：

charset：指定字符编码，用于指定文本的编码格式。例如：charset=utf-8。

boundary：用于分隔不同部分的边界标识符，通常在 multipart/form-data 类型中使用。

下面是一些常见的 Content-Type 示例：

text/plain：表示纯文本类型，没有指定字符编码。

text/html; charset=utf-8：表示 HTML 文档类型，使用 UTF-8 字符编码。

application/json; charset=utf-8：表示 JSON 数据类型，使用 UTF-8 字符编码。

application/xml; charset=utf-8：表示 XML 数据类型，使用 UTF-8 字符编码。

multipart/form-data; boundary=----WebKitFormBoundary123ABC：表示用于文件上传的表单数据类型，指定了边界标识符。

根据具体的需求，可以根据数据类型和特性来设置合适的 Content-Type，并可选地指定字符编码等参数。确保正确设置 Content-Type 可以确保数据在传输过程中得到正确的解析和处理。

## **什么是跨域**

跨域（Cross-Origin）指的是在浏览器环境下，当一个网页的 JavaScript 代码试图访问不同源（Origin）的资源时，就会产生跨域问题。

在 Web 安全策略中，同源策略（Same-Origin Policy）是浏览器的一项重要安全机制。同源策略要求网页只能访问与其来源相同的资源，包括协议、域名和端口号都必须完全一致。如果请求的资源与当前网页的来源不同，浏览器将阻止该请求，这就是跨域问题。

跨域问题的产生是为了保护用户数据安全和防止恶意攻击。通过同源策略，浏览器可以防止不同域下的网页对用户数据进行非法访问。例如，一个恶意网站无法通过 JavaScript 代码获取到用户在其他网站的登录凭证。

然而，有些情况下需要进行跨域访问，例如前端网页需要与后端 API 进行通信、跨域资源共享（CORS）等。为了实现跨域访问，需要在服务器端进行相应的配置，允许跨域请求。

常见的跨域解决方案包括：

JSONP：通过动态创建 <script> 标签实现跨域请求。

CORS：服务器端设置响应头允许跨域请求，浏览器根据响应头判断是否允许访问。

代理服务器：在同源策略下，前端请求自己的后端，再由后端请求其他域的资源，将结果返回给前端。

需要注意的是，跨域问题只存在于浏览器环境下，而在服务器端不存在跨域限制。因此，通过后端代理转发请求或直接在后端进行数据访问是不受跨域限制的。

## **数据库事务**

数据库事务（Database Transaction）是一组数据库操作（例如读取、写入、更新、删除等）的逻辑单元，这些操作被当作一个不可分割的工作单元进行管理。事务的目的是保证数据的一致性、完整性和可靠性。

事务具有以下特性（ACID 特性）：

原子性（Atomicity）：事务作为一个整体被执行，要么全部成功，要么全部失败。如果事务中的任何一个操作失败，则会回滚（Rollback）到事务开始前的状态，保证数据的一致性。

一致性（Consistency）：事务执行前后数据库的状态应保持一致。事务开始前的状态和结束后的状态必须满足预设的规则和约束。

隔离性（Isolation）：事务的执行是相互隔离的，一个事务的操作不会被其他事务所干扰。隔离级别定义了不同事务之间的隔离程度，包括读未提交、读已提交、可重复读和串行化。

持久性（Durability）：一旦事务被提交，对数据的修改将会永久保存在数据库中，即使在发生故障的情况下也能够恢复。

数据库事务可以使用以下语句来控制：

BEGIN TRANSACTION：开始一个事务。

COMMIT：提交事务，使事务中的所有操作生效。

ROLLBACK：回滚事务，撤销事务中的所有操作。

事务的应用场景包括银行转账、订单处理、库存管理等需要保证数据完整性和一致性的业务操作。通过使用事务，可以确保数据库操作的正确执行和数据的正确性。

## **数据库的五种数据类型**

常见的数据库系统支持的五种基本数据类型包括：

整型（Integer）：用于表示整数值，包括正整数、负整数和零。常见的整型数据类型有：INT、INTEGER、TINYINT、SMALLINT、BIGINT 等。

浮点型（Floating-Point）：用于表示带有小数部分的数值。浮点型数据类型分为单精度和双精度，通常使用的是 DOUBLE 和 FLOAT。

字符串型（String）：用于表示文本数据。字符串型数据类型用于存储字符序列，可以是字母、数字、符号或其他字符的组合。常见的字符串型数据类型有：VARCHAR、CHAR、TEXT 等。

日期与时间型（Date and Time）：用于表示日期和时间信息。日期与时间型数据类型用于存储年、月、日、时、分、秒等时间相关信息。常见的日期与时间型数据类型有：DATE、TIME、DATETIME、TIMESTAMP 等。

布尔型（Boolean）：用于表示真值，只能取两个值之一：True 或 False。在某些数据库系统中，布尔型用数字 1 表示 True，用数字 0 表示 False。

除了以上五种基本数据类型，不同的数据库系统可能还支持其他特定的数据类型，如二进制型、大文本型、枚举型、数组型等。具体支持的数据类型取决于数据库系统的设计和实现。

## **redis的五种数据类型**

Redis 是一种高性能的内存数据存储系统，它支持多种数据类型。以下是 Redis 中常用的五种数据类型：

字符串（String）：字符串是 Redis 最基本的数据类型，可以存储任意类型的数据，包括文本、数字等。字符串类型的值最大可以达到 512MB。

列表（List）：列表是一个有序的字符串集合，可以在列表的两端进行插入和删除操作。可以用来实现队列（Queue）或栈（Stack）等数据结构。

哈希（Hash）：哈希是一种键值对存储结构，类似于关联数组或字典。在 Redis 中，哈希可以存储多个字段和对应的值，常用于存储对象或实体的属性。

集合（Set）：集合是一个无序的字符串集合，不允许重复的元素。可以进行集合间的交集、并集、差集等操作，还可以快速判断一个元素是否存在于集合中。

有序集合（Sorted Set）：有序集合是集合的扩展，每个元素都关联一个分数（Score），并按照分数进行排序。可以根据分数范围获取元素，还可以为元素设置权重和排序优先级。

通过使用不同的数据类型，Redis 提供了丰富的功能和灵活的数据操作方式。开发者可以根据具体的业务需求选择合适的数据类型来存储和操作数据。

## **多线程的状态**

多线程在执行过程中会经历不同的状态，常见的多线程状态包括：

新建状态（New）：线程被创建后尚未启动的状态。

运行状态（Runnable）：线程正在执行任务的状态。处于运行状态的线程可能正在执行、等待系统资源（如 CPU 资源），或者等待其他线程的通知。

阻塞状态（Blocked）：线程因为某些原因暂时停止执行的状态。当线程等待某个条件满足时（如等待锁的释放），它将进入阻塞状态。

等待状态（Waiting）：线程因为某些原因无限期等待的状态。当线程调用了 wait() 方法、join() 方法或 LockSupport.park() 方法时，它将进入等待状态。

计时等待状态（Timed Waiting）：线程因为某些原因有一定时间限制的等待状态。当线程调用了带有超时参数的 sleep() 方法、wait(timeout) 方法、join(timeout) 方法或 LockSupport.parkNanos() 方法时，它将进入计时等待状态。

终止状态（Terminated）：线程执行完任务或因异常退出的状态。

这些状态在多线程编程中是相互转换的，线程的状态会随着不同的操作而发生改变。例如，新建的线程通过调用 start() 方法进入运行状态，运行中的线程可以被阻塞或进入等待状态，最终线程执行完任务或异常退出进入终止状态。

了解线程的不同状态对于多线程编程和调试非常重要，可以更好地控制和管理线程的行为。

## **redis的 分布式锁**

Redis 的分布式锁是一种基于 Redis 数据库实现的锁机制，用于在分布式环境下控制对共享资源的访问，保证同一时刻只有一个线程或进程可以获得锁并执行关键操作，从而确保数据的一致性和并发控制。

常见的分布式锁实现方式有两种：

基于 SETNX 命令：通过使用 Redis 的 SETNX（SET if Not eXists）命令来实现。该命令可以在指定的键不存在时设置键的值，并返回成功与否的状态。利用这个特性，可以将某个键作为锁的标识，在获取锁时设置键的值，如果设置成功，则表示获取到了锁；否则表示锁已经被其他线程或进程持有。

基于 Lua 脚本：通过执行 Lua 脚本来实现分布式锁。Lua 脚本可以在 Redis 服务器端原子地执行多个命令，确保获取锁和释放锁的操作是原子的，避免竞态条件。脚本可以利用 Redis 的 SETNX 命令和 EXPIRE 命令来设置锁的值和过期时间，并在释放锁时根据锁的值验证并删除锁。

使用 Redis 分布式锁时需要考虑以下注意事项：

设置合理的锁超时时间：确保获取到锁的线程在执行完关键操作后及时释放锁，避免锁长时间占用，导致其他线程无法获取锁。

避免死锁：在释放锁时需要保证线程持有的锁与要释放的锁是一致的，避免误释放其他线程持有的锁。

锁的粒度：根据实际情况，合理选择锁的粒度，避免过大或过小的锁粒度影响系统性能。

容错处理：在获取锁和释放锁的过程中，需要处理可能出现的异常情况，例如 Redis 连接异常、锁被意外释放等，保证系统的可靠性和稳定性。

分布式锁是在分布式系统中常用的并发控制机制，通过 Redis 实现的分布式锁具有高性能、高可用性和易用性的特点，广泛应用于分布式场景中的并发控制和资源竞争问题。

## **设置锁超时时间**

在使用 Redis 实现分布式锁时，设置锁超时时间是一种重要的机制，可以防止锁长时间被占用而导致死锁或资源浪费的情况发生。下面是设置锁超时时间的一般步骤：

在获取锁时，设置锁的键和值，并同时设置锁的过期时间。可以使用 Redis 的 SETNX 命令或执行 Lua 脚本来实现。

示例代码（使用 SETNX 命令）：

//设置key

SETNX lock\_key "locked"

//给key添加超时时间

EXPIRE lock\_key lock\_timeout

示例代码（使用 Lua 脚本）：

local lock\_key = KEYS[1]

local lock\_timeout = ARGV[1]

if redis.call("SETNX", lock\_key, "locked") == 1 then

redis.call("EXPIRE", lock\_key, lock\_timeout)

return "OK"

else

return "FAILED"

end

执行关键操作。在获取到锁后，执行需要加锁的关键操作。

在关键操作执行完毕后，释放锁。可以通过删除锁的键来释放锁。

示例代码：

DEL lock\_key

设置锁超时时间的关键是在获取锁时设置锁的过期时间，确保锁在一定时间后自动释放，避免长时间占用。一般来说，锁的超时时间应根据业务需求和关键操作的预估执行时间来合理设置。如果关键操作执行时间较长，建议将锁的超时时间设置得更长一些，以避免在关键操作未完成时锁被自动释放。但同时也要注意，设置过长的超时时间可能会导致其他线程或进程在等待获取锁时的延迟。

需要注意的是，在使用分布式锁时，锁的超时时间应该合理设置，避免因为锁的过早释放或过长占用导致并发控制的不准确性或性能问题。同时，还需要考虑异常处理，例如锁的获取失败或释放异常时的处理机制，以保证系统的稳定性和可靠性。

## **锁时队列和等待队列**

锁时队列（Lock Queue）和等待队列（Wait Queue）是在并发编程中用于管理线程或进程竞争共享资源的机制。它们在实现分布式锁或线程同步时起到重要的作用。

锁时队列（Lock Queue）：

定义：锁时队列是一个用于存储等待获取锁的线程或进程的队列。当多个线程或进程同时竞争同一个锁时，只有一个线程或进程可以成功获取锁，其他线程或进程需要在锁时队列中等待。

特点：

先进先出（FIFO）：锁时队列通常采用先进先出的策略，即等待时间最长的线程或进程先获取锁。

公平性：锁时队列可以提供一定程度的公平性，保证所有等待获取锁的线程或进程都有机会获取到锁，避免饥饿现象。

等待队列（Wait Queue）：

定义：等待队列是一个用于存储等待某个条件满足的线程或进程的队列。当某个条件不满足时，线程或进程需要进入等待队列，直到条件满足后才能继续执行。

特点：

条件等待：等待队列通常与条件变量（Condition Variable）相关联，线程或进程通过等待条件变量的信号（Signal）或广播（Broadcast）来等待条件满足。

唤醒操作：当条件满足时，等待队列中的线程或进程可以被唤醒，从等待状态转换为可执行状态。

在分布式锁的实现中，锁时队列常用于管理多个进程或节点之间的锁竞争，确保只有一个进程或节点可以获取到锁。等待队列常用于线程同步机制中，当某个条件不满足时，线程进入等待状态，直到条件满足后再继续执行。

这两种队列的实现方式可以根据具体的应用场景和需求来选择，常见的实现方式包括使用阻塞队列、链表等数据结构来管理等待线程或进程。这些队列的设计和使用都需要考虑并发安全性、性能和公平性等因素，以保证线程或进程的正确等待和唤醒操作。

## **数据库事务的行级锁**

数据库事务的行级锁（Row-Level Locking）是一种并发控制机制，用于保护数据库中的行级数据，确保多个事务之间对同一行数据的并发操作不会出现数据不一致或冲突的情况。

行级锁是在数据库引擎层面实现的，它可以在事务级别或语句级别对数据行进行加锁。行级锁的主要特点如下：

细粒度控制：相比于表级锁或页级锁，行级锁提供了更细粒度的数据控制，可以在并发访问中实现更高的并发性和灵活性。

并发性能：行级锁的使用可以提高并发性能，允许多个事务同时访问同一张表的不同行，避免了不必要的阻塞或冲突。

数据一致性：行级锁可以保护数据的一致性，避免了多个事务对同一行数据进行不一致的修改或读取。

锁冲突：行级锁可能导致锁冲突，当多个事务同时请求对同一行数据进行修改或读取时，可能会发生锁等待和死锁的情况，需要进行适当的处理和调优。

数据库系统通常会自动管理行级锁，并根据事务的隔离级别和操作类型来自动获取和释放锁。在具体使用行级锁时，需要根据业务需求和性能要求选择合适的锁粒度，并了解数据库系统对行级锁的支持和行为。

需要注意的是，行级锁的实现和行为可能因不同的数据库管理系统而有所差异，因此在具体的数据库产品中使用行级锁时，建议查阅相应的文档和文档，了解其具体的实现方式和行为规则。

### 怎么设置行级锁

设置锁级别：在执行查询或更新操作之前，你可以使用SET语句设置MySQL的锁级别。通过执行java.sql.Statement对象的executeUpdate()方法，可以执行任意SQL语句。下面是一个设置行级锁的示例：

codeStatement statement = connection.createStatement();

String lockLevelQuery = "SET innodb\_lock\_wait\_timeout = 50";

statement.executeUpdate(lockLevelQuery);

在SQL语句中，你可以使用MySQL的锁机制来设置行级锁。MySQL提供了不同类型的锁，包括共享锁（Shared Lock）和排他锁（Exclusive Lock）。下面是使用SQL语句设置行级锁的示例：

设置共享锁（Shared Lock）：

SELECT \* FROM your\_table\_name WHERE your\_condition LOCK IN SHARE MODE;

在上述示例中，使用SELECT语句查询数据，并在LOCK IN SHARE MODE子句后设置共享锁。这将获取一个共享锁来保护所选行，其他事务可以读取但不能修改这些行。

设置排他锁（Exclusive Lock）：

SELECT \* FROM table\_name WHERE your\_condition FOR UPDATE;

在上述示例中，使用SELECT语句查询数据，并在FOR UPDATE子句后设置排他锁。这将获取一个排他锁来保护所选行，其他事务既不能读取也不能修改这些行。

请注意，在设置行级锁时，your\_table\_name是你要锁定的表名，your\_condition是用于选择特定行的条件。你需要根据你的实际情况替换它们。

另外，行级锁的行为和效果受数据库引擎的支持和配置影响。请确保你了解你使用的MySQL版本和所选的存储引擎对行级锁的支持情况。

## **定时任务**

Spring TaskScheduler是Spring框架提供的定时任务调度器，用于在Spring应用中执行定时任务。它基于TaskScheduler接口和相关实现类，提供了灵活且易于使用的方式来定义和调度定时任务。

首先，需要在Spring配置文件中配置TaskScheduler的实现类，例如使用ThreadPoolTaskScheduler：

xmlCopy code<bean id="taskScheduler" class="org.springframework.scheduling.concurrent.ThreadPoolTaskScheduler">

<property name="poolSize" value="10" />

</bean>

上述配置将创建一个ThreadPoolTaskScheduler对象，并设置线程池大小为10。

接下来，在需要执行定时任务的方法上添加@Scheduled注解，并设置定时任务的执行规则：

javaCopy code@Component

public class MyScheduledTasks {

@Scheduled(fixedRate = 5000) // 每隔5秒执行一次

public void myTask() {

// 定时任务的执行逻辑

// ...

}

}

上述代码中，myTask方法使用@Scheduled注解，并指定了fixedRate属性为5000，表示每隔5秒执行一次。

除了fixedRate属性外，@Scheduled注解还支持其他属性，例如fixedDelay、cron等，可以根据需求灵活配置定时任务的执行规则。

最后，需要确保Spring容器中包含了TaskScheduler对象，并且MyScheduledTasks组件被扫描到。

Spring TaskScheduler会在每次触发时间点执行被注解标记的方法，从而实现定时任务的调度和执行。

需要注意的是，定时任务的方法需要遵循一定的命名规则和参数要求，具体可以参考Spring框架的官方文档。此外，定时任务的执行可以配置在单个应用实例中，也可以配置在分布式环境中的多个应用实例中，具体要根据项目的需求和架构来确定。

# 真题

## **rabbitmq**

RabbitMQ是一个开源的消息队列（message broker）软件，它实现了高级消息队列协议（AMQP）标准。它提供了可靠的消息传递机制，用于在分布式系统之间进行异步通信。以下是关于RabbitMQ的一些重要概念和特性的介绍：

消息代理：

RabbitMQ作为消息代理，负责接收、存储和转发消息。它充当了生产者和消费者之间的中间件，协调消息的传输和处理。

消息队列：

RabbitMQ使用消息队列来存储消息。消息队列是一种先进先出（FIFO）的数据结构，生产者将消息发送到队列，而消费者从队列中接收消息。

生产者（Producer）：

生产者是消息的发送者。它将消息发布到RabbitMQ中的交换器（Exchange），并指定消息的路由键。

交换器（Exchange）：

交换器是消息的接收和分发中心。它接收来自生产者的消息，并根据规则将消息路由到一个或多个消息队列中。

队列（Queue）：

队列是RabbitMQ中的消息存储区域。它接收交换器发送的消息，并等待消费者来接收和处理这些消息。

消费者（Consumer）：

消费者是消息的接收者。它订阅一个或多个队列，从队列中获取消息并进行处理。

在项目中使用RabbitMQ可以实现异步消息传递和解耦，提高系统的可伸缩性和可靠性。下面是一些在项目中使用RabbitMQ的常见场景和示例：

异步任务处理：

在项目中，可以将耗时的任务发送到RabbitMQ队列，由消费者异步处理。这样可以避免阻塞主线程，提高系统的响应性能。

示例场景：发送邮件、生成报表、文件处理等耗时操作。

消息通知和广播：

使用RabbitMQ可以实现消息的广播和通知功能。生产者将消息发布到交换器，交换器将消息路由到多个队列，消费者可以订阅相应的队列来接收消息。

示例场景：系统广播通知、实时消息推送等。

解耦和微服务通信：

在分布式系统中，不同的微服务之间可以通过RabbitMQ进行消息通信，实现解耦和松耦合的架构。

示例场景：微服务之间的事件触发、状态同步、数据同步等。

日志收集和分析：

可以使用RabbitMQ作为日志的中间件，生产者将日志消息发送到队列，消费者将日志存储到数据库或发送到其他系统进行分析和监控。

示例场景：日志收集、分析和告警等。

异步消息传递： RabbitMQ允许应用程序通过发送和接收消息进行异步通信。它解耦了应用程序之间的直接依赖关系，使得消息的生产者和消费者可以独立地进行开发和扩展。消息可以包含任意类型的数据，并且可以在不同的系统、进程或语言之间进行传递。

解耦和削峰填谷： RabbitMQ的消息队列机制实现了解耦和削峰填谷的功能。生产者将消息发送到消息队列中，消费者从队列中接收消息进行处理。这样可以使生产者和消费者之间的处理速度解耦，即使某一方处理速度较慢或暂时不可用，消息也会得到保存并在后续合适的时间被消费。这种机制可以帮助平衡系统的负载，提高系统的稳定性和可伸缩性。

可靠性和持久化： RabbitMQ提供了可靠性的消息传递机制。它支持消息的持久化存储，即将消息存储到磁盘上，以防止在发生故障或重启后丢失消息。同时，RabbitMQ还支持消息的确认机制，生产者可以在消息发送后等待消费者的确认，确保消息被正确处理。

路由和消息策略： RabbitMQ提供了丰富的路由和消息策略机制。通过交换器（Exchange）和队列的绑定关系，可以实现灵活的消息路由。RabbitMQ支持多种路由模式，如直接路由、主题路由和广播路由，以满足不同场景下的消息分发需求。

可扩展性和多语言支持： RabbitMQ是一个开源的消息中间件，它提供了丰富的客户端库和插件，支持多种编程语言和开发框架。这使得开发人员可以方便地集成RabbitMQ到各种应用中，并根据需要进行定制和扩展。

在项目中使用RabbitMQ的一般步骤如下：

安装和配置RabbitMQ服务器。

引入RabbitMQ的Java客户端依赖。

创建生产者代码，连接到RabbitMQ服务器，声明交换器和队列，发送消息到队列。

创建消费者代码，连接到RabbitMQ服务器，订阅队列，接收和处理消息。

配置异常处理和错误处理机制，确保消息传递的可靠性。

部署和运行生产者和消费者，观察消息的发送和接收情况。

需要注意的是，在使用RabbitMQ时要考虑消息的可靠性、幂等性和消息重复处理等问题，以确保系统的稳定性和数据的一致性。

什么是RabbitMQ？它的作用是什么？ 答：RabbitMQ是一个开源的消息队列中间件，它提供了可靠的异步消息传递机制，用于在分布式系统中进行应用程序之间的通信和解耦。

RabbitMQ与其他消息队列中间件的区别是什么？ 答：RabbitMQ与其他消息队列中间件的区别在于其灵活的消息路由机制、丰富的可靠性保证机制、插件系统的可扩展性，以及广泛的开发社区支持等方面。

RabbitMQ的主要组件有哪些？ 答：RabbitMQ的主要组件包括生产者（Producer）、消费者（Consumer）、消息队列（Queue）、交换机（Exchange）和绑定（Binding）等。

RabbitMQ的消息传递模型是什么？ 答：RabbitMQ采用的消息传递模型是发布/订阅模型。生产者将消息发布到交换机，交换机根据绑定规则将消息路由到一个或多个队列中，然后消费者从队列中订阅并接收消息。

RabbitMQ中的交换机和队列有什么区别？ 答：交换机用于接收生产者发送的消息并将其路由到一个或多个队列，而队列用于存储消息直到消费者准备好接收和处理它们。

RabbitMQ如何保证消息的可靠传递？ 答：RabbitMQ通过消息确认机制来保证消息的可靠传递。消费者在处理完消息后发送确认消息给RabbitMQ，如果消费者在处理消息时发生错误，消息将被重新投递给其他消费者或等待重新处理。

RabbitMQ中的消息确认机制是什么？如何使用它？ 答：RabbitMQ的消息确认机制是通过ACK机制实现的。消费者在处理消息后发送ACK消息给RabbitMQ，告知它已经成功处理了消息。可以通过设置手动ACK模式来使用消息确认机制。

RabbitMQ如何实现消息的持久化？ 答：可以将消息和队列进行持久化来实现消息的持久化。将消息标记为持久化后，即使RabbitMQ服务器重启，消息也不会丢失。同时，还可以将队列和交换机进行持久化配置。

RabbitMQ的高可用性和可伸缩性如何实现？ 答：RabbitMQ通过集群部署来实现高可用性和可伸缩性。多个RabbitMQ节点组成一个集群，消息可以在节点之间进行复制和分布，确保即使某个节点故障，消息仍然可以正常传递。

RabbitMQ的延迟消息处理如何实现？ 答：RabbitMQ本身不提供延迟消息处理的机制，但可以通过插件或自定义实现来实现延迟消息处理，例如使用TTL（Time-to-Live）和死信队列机制来实现延迟消息。

## **dubbo**

Dubbo是一个高性能、轻量级的开源分布式服务框架，由阿里巴巴集团开发和维护。它提供了面向分布式架构的服务治理、远程调用和负载均衡等功能，旨在简化分布式系统的开发和管理。下面是对Dubbo的一些重要概念和特性的介绍：

服务提供者（Provider）：

服务提供者是提供具体服务实现的节点。它将自己的服务注册到注册中心，并响应消费者的远程调用请求。

服务消费者（Consumer）：

服务消费者是调用远程服务的节点。它从注册中心获取服务提供者的地址，并通过代理方式发起远程调用。

注册中心（Registry）：

注册中心是Dubbo框架中的服务注册与发现的核心组件。服务提供者在启动时将自己的地址信息注册到注册中心，而服务消费者从注册中心获取可用的服务地址。

远程调用（RPC）：

Dubbo通过远程调用实现服务之间的通信。它提供了高效的序列化和网络传输协议，支持多种传输方式（如TCP、HTTP）和序列化协议（如Hessian、JSON）。

服务接口（Service Interface）：

服务接口定义了服务的契约，包括服务的方法和参数。服务提供者和服务消费者通过共享服务接口来实现解耦和互操作。

服务注册与发现：

Dubbo通过注册中心实现服务的注册与发现。服务提供者在启动时将自己的服务信息注册到注册中心，服务消费者从注册中心获取可用的服务地址。

负载均衡（Load Balancing）：

Dubbo支持多种负载均衡策略，用于在多个服务提供者之间分发调用请求，实现负载均衡和性能优化。

## **#和$的区别**

#是预编译的可以防止sql注入 $与等于占位符，有可能被sql注入

## **进行数学计算用的是哪个类**

在Java中，进行数学计算可以使用java.lang.Math类。Math类提供了一系列静态方法，用于执行常见的数学运算，例如求绝对值、取整、取余、幂运算、三角函数、对数函数等。下面是Math类的一些常用方法：

abs(x): 返回参数x的绝对值。

ceil(x): 返回大于或等于参数x的最小整数，且类型为double。

floor(x): 返回小于或等于参数x的最大整数，且类型为double。

round(x): 返回参数x的四舍五入整数，且类型为long。

max(x, y): 返回x和y中的较大值。

min(x, y): 返回x和y中的较小值。

pow(x, y): 返回x的y次幂。

sqrt(x): 返回参数x的平方根。

sin(x), cos(x), tan(x): 分别返回参数x的正弦、余弦、正切值。

log(x): 返回参数x的自然对数（以e为底）。

random(): 返回0.0到1.0之间的随机浮点数。

除了Math类之外，还可以使用java.util.BigDecimal类来处理更高精度的数学计算，以避免浮点数精度问题。BigDecimal类提供了精确的十进制运算功能。

需要注意的是，Math类和BigDecimal类的方法都是静态方法，可以直接通过类名调用。例如：Math.abs(-5)会返回5。

## **linux中/和/\*的区别**

/是根目录 ， /\*是根目录下的所有资源

## **Zookeeper**

ZooKeeper是一个开源的分布式协调服务，由Apache软件基金会开发和维护。它提供了一个可靠的分布式环境，用于协调和管理分布式应用程序的配置、命名服务、分布式锁、分布式队列等。以下是一些关键概念和特性的介绍：

分布式协调服务：

ZooKeeper提供了一个分布式环境，使得多个应用程序可以共享和协调资源。它提供了一致性、可靠性和高性能的服务。

命名服务（Name Service）：

ZooKeeper允许应用程序将自己的服务注册到ZooKeeper中，使得其他应用程序可以通过名称查找和访问这些服务。

配置管理（Configuration Management）：

ZooKeeper可以用于集中管理和分发分布式应用程序的配置信息。应用程序可以监听配置节点的变化，实时更新配置。

分布式锁（Distributed Locking）：

ZooKeeper提供了分布式锁的实现，可以协调多个应用程序之间的互斥访问，保证数据的一致性和可靠性。

分布式队列（Distributed Queue）：

ZooKeeper提供了分布式队列的支持，可以实现任务的顺序执行和负载均衡，适用于任务调度和消息队列等场景。

高性能和高可靠性：

ZooKeeper使用ZAB（ZooKeeper Atomic Broadcast）协议，保证了数据的原子性和一致性。它的设计目标是提供低延迟、高吞吐量的性能。

数据模型：

ZooKeeper使用类似文件系统的数据模型，由层级的节点（znode）组成。每个节点可以存储数据和子节点，并且可以触发事件通知。

顺序一致性：

ZooKeeper提供了顺序一致性的保证。所有的更新操作都按照严格的顺序进行，每个操作都有一个全局唯一的递增编号。

客户端连接：

应用程序通过ZooKeeper提供的客户端库连接到ZooKeeper服务器，进行数据的读取、写入、监听等操作。

ZooKeeper在分布式系统中扮演着重要的角色，为应用程序提供了分布式协调和管理的基础设施。它的特性和可靠性使得它成为构建分布式系统和应用程序的重要工具之一。

## **seata**

Seata 是一个开源的分布式事务解决方案，主要用于在分布式系统中实现事务的一致性和可靠性。它提供了一套完整的分布式事务管理框架，包括事务协调器（Transaction Coordinator，TC）、事务管理器（Transaction Manager，TM）和资源管理器（Resource Manager，RM）等组件。

Seata 的主要作用包括：

分布式事务管理：Seata 提供了分布式事务的管理和协调机制，可以确保多个服务之间的事务操作的一致性。它协调参与者的事务操作，并最终决定事务的提交或回滚，保证分布式系统中的数据一致性。

数据库事务支持：Seata 对各种主流的关系型数据库（如MySQL、Oracle、PostgreSQL等）提供了事务支持，可以将多个数据库操作纳入到一个分布式事务中进行管理，保证数据的一致性。

弹性和容错：Seata 提供了事务的弹性和容错机制。在分布式系统中，当某个服务或资源出现故障时，Seata 可以自动回滚或重试事务，确保系统的稳定性和可靠性。

分布式锁和并发控制：Seata 提供了分布式锁和并发控制的功能，可以防止多个服务对同一资源进行并发修改，保证数据的完整性和准确性。

全局事务追踪和监控：Seata 提供了全局事务追踪和监控功能，可以对分布式事务进行跟踪和监控，实时查看事务的状态和性能指标，便于故障排查和系统优化。

通过使用 Seata，开发人员可以简化分布式事务的管理和控制，提高系统的可靠性和一致性。Seata 提供了与各种框架和中间件的集成支持，如Spring Boot、Dubbo、RocketMQ等，使得在不同的分布式应用场景中使用 Seata 变得更加便捷和灵活。

## **sentinel**

Sentinel是一个开源的流量控制和熔断降级框架，由阿里巴巴集团开发和维护。它旨在提供高可用性、稳定性和保护性的分布式系统的流量控制和熔断降级能力。以下是Sentinel的一些关键特点和功能：

流量控制：Sentinel可以根据系统的负载情况和资源的可用性，对流入系统的请求进行限流控制。它支持基于QPS（每秒查询率）、线程数等多种维度的流量控制策略。

熔断降级：Sentinel可以根据系统的异常情况和错误率，对服务进行熔断和降级处理。它能够实时监控服务的调用情况，并根据预设的规则自动触发熔断和降级操作，以避免故障的扩散和影响。

系统保护：Sentinel提供了一系列系统级别的保护机制，包括系统负载保护、慢调用比例保护、异常比例保护等。这些保护机制可以有效地防止系统在高负载或异常情况下的崩溃和雪崩效应。

实时监控和统计：Sentinel提供了实时监控和统计功能，可以对服务的流量、熔断状态、异常情况等进行实时监控和统计，帮助开发人员快速定位和解决问题。

动态规则配置：Sentinel支持动态的规则配置，可以通过控制台、API等方式动态地修改流量控制和熔断降级的规则，而无需重启服务或重新部署。

高可扩展性：Sentinel可以与各种框架和中间件进行集成，如Spring Cloud、Dubbo、gRPC等，使其能够适应不同的分布式应用场景和架构。

Sentinel的目标是保护分布式系统免受流量突发和故障的影响，提供流量控制、熔断降级等机制来保证系统的可靠性和稳定性。它可以与微服务架构和云原生应用的开发相结合，为分布式系统提供强大的容错和保护能力。

## **OpenFeign**

OpenFeign是一个基于注解的声明式Web服务客户端框架，它简化了编写HTTP客户端的过程。OpenFeign是Spring Cloud生态系统的一部分，旨在简化微服务架构中服务间的通信。

使用OpenFeign，开发人员可以通过编写接口的方式来定义对远程服务的调用，而无需手动编写HTTP请求和处理响应。OpenFeign将这些接口的实现自动生成，包括HTTP请求的发送、编码和解码等过程。它提供了一套注解和配置选项，用于定义请求的细节，如URL路径、请求方法、请求参数、请求头等。

以下是OpenFeign的一些关键特点和功能：

声明式编程模型：使用OpenFeign，开发人员可以通过接口的方式定义对远程服务的调用，将远程服务的API定义直接暴露在接口中。这种声明式的编程模型更加直观和易于理解。

注解驱动：OpenFeign提供了一系列注解，用于标记接口方法的请求细节，如URL路径、请求方法、请求参数、请求头等。这些注解让开发人员可以更加方便地配置和定制请求。

自动化实现：OpenFeign会自动根据接口的定义生成对远程服务的调用代码，包括HTTP请求的发送和响应的处理。开发人员只需要定义接口并添加相应的注解，无需手动编写与HTTP相关的代码。

负载均衡和服务发现：OpenFeign集成了Ribbon和Eureka等负载均衡和服务发现的组件，可以实现对远程服务的负载均衡和自动服务发现。

可插拔的编码和解码：OpenFeign支持不同的编码和解码器，可以与不同的数据格式（如JSON、XML等）进行交互。开发人员可以根据需求选择合适的编码和解码方式。

集成Spring Cloud：OpenFeign与Spring Cloud生态系统紧密集成，可以与其他组件（如Zuul、Hystrix、Config等）一起使用，构建完整的微服务架构。

OpenFeign简化了微服务架构中的服务调用，减少了编写HTTP客户端的工作量，提高了开发效率。它与Spring Cloud的其他组件配合使用，可以实现负载均衡、服务发现、熔断降级等功能，帮助构建可靠和弹性的分布式系统。

启用OpenFeign：在Spring Boot应用程序的启动类上添加@EnableFeignClients注解，以启用OpenFeign。

@SpringBootApplication

@EnableFeignClients

public class YourApplication {

public static void main(String[] args) {

SpringApplication.run(YourApplication.class, args);

}

}

创建OpenFeign客户端接口：定义一个接口，用于声明对远程服务的调用。接口上使用@FeignClient注解指定要调用的服务名称，并使用@RequestMapping等注解定义请求的URL路径、请求方法等。

@FeignClient(name = "your-service")

public interface YourServiceClient {

@RequestMapping(method = RequestMethod.GET, value = "/api/resource/{id}")

YourResource getResourceById(@PathVariable("id") Long id);

@RequestMapping(method = RequestMethod.POST, value = "/api/resource")

void createResource(@RequestBody YourResource resource);

// 其他方法声明

}

使用OpenFeign客户端：在需要调用远程服务的地方，通过依赖注入的方式使用OpenFeign客户端接口。

javaCopy code@RestController

public class YourController {

private final YourServiceClient yourServiceClient;

public YourController(YourServiceClient yourServiceClient) {

this.yourServiceClient = yourServiceClient;

}

@GetMapping("/resource/{id}")

public YourResource getResourceById(@PathVariable("id") Long id) {

return yourServiceClient.getResourceById(id);

}

@PostMapping("/resource")

public void createResource(@RequestBody YourResource resource) {

yourServiceClient.createResource(resource);

}

// 其他方法

}

在上述代码中，YourServiceClient是一个OpenFeign的客户端接口，定义了对远程服务的调用方法。YourController是一个Spring MVC的控制器，通过依赖注入的方式使用YourServiceClient来调用远程服务。

需要注意的是，以上示例中的代码是简化的示例，实际使用中可能需要配置更多的参数和选项，如请求头、请求参数、错误处理等。你可以根据具体的需求和场景进行适当的配置和定制。

总结来说，使用OpenFeign时，需要定义一个接口来声明对远程服务的调用，通过注解配置请求的细节，然后通过依赖注入的方式使用该接口来实现服务间的通信。

## **session设置过期时间**

// 获取当前Session对象

HttpSession session = request.getSession();

// 设置过期时间为30分钟（以秒为单位）

int sessionTimeout = 30 \* 60; // 30分钟

session.setMaxInactiveInterval(sessionTimeout);

此外，还可以通过在web.xml配置文件中设置全局的Session过期时间，例如：

<session-config>

<session-timeout>30</session-timeout> <!-- 设置全局过期时间为30分钟 -->

</session-config>

## **tokice设置过期时间**

.setExpiration(new Date(System.currentTimeMillis() + expirationTimeMillis))

获取当前时间的毫秒数 设置的过期时间

## **怎么设置行级锁**

设置锁级别：在执行查询或更新操作之前，你可以使用SET语句设置MySQL的锁级别。通过执行java.sql.Statement对象的executeUpdate()方法，可以执行任意SQL语句。下面是一个设置行级锁的示例：

codeStatement statement = connection.createStatement();

String lockLevelQuery = "SET innodb\_lock\_wait\_timeout = 50";

statement.executeUpdate(lockLevelQuery);

在SQL语句中，你可以使用MySQL的锁机制来设置行级锁。MySQL提供了不同类型的锁，包括共享锁（Shared Lock）和排他锁（Exclusive Lock）。下面是使用SQL语句设置行级锁的示例：

设置共享锁（Shared Lock）：

sqlCopy code

SELECT \* FROM your\_table\_name WHERE your\_condition LOCK IN SHARE MODE;

在上述示例中，使用SELECT语句查询数据，并在LOCK IN SHARE MODE子句后设置共享锁。这将获取一个共享锁来保护所选行，其他事务可以读取但不能修改这些行。

设置排他锁（Exclusive Lock）：

sqlCopy code

SELECT \* FROM your\_table\_name WHERE your\_condition FOR UPDATE;

在上述示例中，使用SELECT语句查询数据，并在FOR UPDATE子句后设置排他锁。这将获取一个排他锁来保护所选行，其他事务既不能读取也不能修改这些行。

请注意，在设置行级锁时，your\_table\_name是你要锁定的表名，your\_condition是用于选择特定行的条件。你需要根据你的实际情况替换它们。

另外，行级锁的行为和效果受数据库引擎的支持和配置影响。请确保你了解你使用的MySQL版本和所选的存储引擎对行级锁的支持情况。

## **spring mvc boot cloud区别**

Spring MVC（Model-View-Controller）：

Spring MVC是Spring框架的一部分，用于构建基于Java的Web应用程序。

它遵循MVC设计模式，将应用程序分为模型（Model）、视图（View）和控制器（Controller）三个部分。

Spring MVC提供了处理请求、响应和页面渲染的机制，以及处理表单提交、数据校验等功能。

开发者需要手动配置和管理各个组件，包括路由、视图解析器、控制器等。

使用Spring MVC，你可以更细粒度地控制应用程序的配置和行为。

Spring Boot：

Spring Boot是基于Spring框架的快速开发框架，旨在简化Spring应用程序的配置和部署。

它通过自动配置和约定优于配置的原则，提供了开箱即用的特性。

Spring Boot通过简化配置和提供默认设置，减少了繁琐的XML配置和手动依赖管理的工作量。

它还集成了内嵌的Servlet容器（如Tomcat、Jetty等），简化了应用程序的部署和运行。

Spring Boot提供了各种开发和监控工具，如自动刷新、健康检查、Actuator等。

Spring Cloud：

Spring Cloud是用于构建分布式系统和微服务架构的框架。

它提供了一系列工具和组件，用于服务注册与发现、负载均衡、配置管理、断路器等。

Spring Cloud可以与Spring Boot集成，通过注解和配置来实现微服务的开发和管理。

它支持常用的云原生技术和模式，如服务注册中心（如Eureka、Consul）、服务网关（如Zuul、Gateway）、配置中心（如Config Server）、断路器（如Hystrix）、分布式追踪（如Zipkin）等。

总结：

Spring MVC是用于构建基于Java的Web应用程序的模块。

Spring Boot是用于快速开发和部署Spring应用程序的框架。

Spring Cloud是用于构建分布式系统和微服务架构的框架。

Spring MVC和Spring Boot可以结合使用，Spring Boot简化了Spring MVC应用程序的配置和部署。

Spring Cloud是构建在Spring Boot之上，提供了构建分布式系统和微服务的工具和组件。

## **三大内置对象**

一.[Servlet](https://so.csdn.net/so/search?q=Servlet&spm=1001.2101.3001.7020)(server applet)

二.Filter：过滤器

三.Listener：监听器

## **九大内置对象**

1、request对象 request 对象是 javax.servlet.httpServletRequest类型的对象。 该对象代表了客户端的请求信息，主要用于接受通过HTTP协议传送到服务器的数据。（包括头信息、系统信息、请求方式以及请求参数等）。request对象的作用域为一次请求。

2、response对象 response 代表的是对客户端的响应，主要是将JSP容器处理过的对象传回到客户端。response对象也具有作用域，它只在JSP页面内有效。

3、session对象 session 对象是由服务器自动创建的与用户请求相关的对象。服务器为每个用户都生成一个session对象，用于保存该用户的信息，跟踪用户的操作状态。session对象内部使用Map类来保存数据，因此保存数据的格式为 “Key/value”。 session对象的value可以使复杂的对象类型，而不仅仅局限于字符串类型。

4、application对象 application 对象可将信息保存在服务器中，直到服务器关闭，否则application对象中保存的信息会在整个应用中都有效。与session对象相比，application对象生命周期更长，类似于系统的“全局变量”。

5、out 对象 out 对象用于在Web浏览器内输出信息，并且管理应用服务器上的输出缓冲区。在使用 out 对象输出数据时，可以对数据缓冲区进行操作，及时清除缓冲区中的残余数据，为其他的输出让出缓冲空间。待数据输出完毕后，要及时关闭输出流。

6、pageContext 对象 pageContext 对象的作用是取得任何范围的参数，通过它可以获取 JSP页面的out、request、reponse、session、application 等对象。pageContext对象的创建和初始化都是由容器来完成的，在JSP页面中可以直接使用 pageContext对象。

7、config 对象 config 对象的主要作用是取得服务器的配置信息。通过 pageConext对象的 getServletConfig() 方法可以获取一个config对象。当一个Servlet 初始化时，容器把某些信息通过 config对象传递给这个 Servlet。 开发者可以在web.xml 文件中为应用程序环境中的Servlet程序和JSP页面提供初始化参数。

8、page 对象 page 对象代表JSP本身，只有在JSP页面内才是合法的。 page隐含对象本质上包含当前 Servlet接口引用的变量，类似于Java编程中的 this 指针。

9、exception 对象 exception 对象的作用是显示异常信息，只有在包含 isErrorPage=“true” 的页面中才可以被使用，在一般的JSP页面中使用该对象将无法编译JSP文件。excepation对象和Java的所有对象一样，都具有系统提供的继承结构。exception 对象几乎定义了所有异常情况。在Java程序中，可以使用try/catch关键字来处理异常情况； 如果在JSP页面中出现没有捕获到的异常，就会生成 exception 对象，并把 exception 对象传送到在page指令中设定的错误页面中，然后在错误页面中处理相应的 exception 对象。

## **数据库中的事务是如何实现的**

数据库中的事务是通过以下方式实现的：

ACID特性： 事务遵循ACID特性，即原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation）和持久性（Durability）。这些特性确保了事务的可靠性和可恢复性。

事务日志： 数据库通过使用事务日志（Transaction Log）来记录事务的操作。事务日志将事务的操作序列化地记录下来，包括数据修改、事务开始和提交等操作。在发生故障或崩溃时，数据库可以利用事务日志进行恢复，保证事务的原子性和持久性。

锁机制： 数据库使用锁机制来实现事务的隔离性。锁可以用于控制并发事务对数据的访问，确保同时只有一个事务能够修改数据。数据库使用不同级别的锁（如行级锁、表级锁）来管理并发访问，以保证事务的隔离性和数据的一致性。

回滚和提交： 在事务的执行过程中，如果发生错误或需要取消操作，数据库提供了回滚（Rollback）操作，将事务中的操作恢复到之前的状态。而当事务成功执行完成时，可以通过提交（Commit）操作将事务中的操作永久保存到数据库中。

并发控制： 数据库通过并发控制机制来管理多个并发执行的事务。并发控制确保了事务的隔离性，避免了数据不一致的问题。常见的并发控制技术包括锁机制、并发事务控制协议（如两阶段提交协议）、多版本并发控制（MVCC）等。

通过以上的实现方式，数据库可以保证事务的原子性、一致性、隔离性和持久性，提供可靠的数据操作和数据完整性。这使得数据库成为处理复杂事务的重要工具，并广泛应用于许多应用领域，如银行、电子商务等。

## **spring的两大核心组件**

Spring框架的两个核心组件是：

Spring IOC（Inversion of Control，控制反转）： Spring IOC是Spring框架的核心概念之一。它通过依赖注入（Dependency Injection，DI）的方式，将对象的创建、组装和管理的责任从应用程序代码转移到容器中。控制反转使得对象之间的依赖关系更松散，提高了代码的可维护性和可测试性。Spring IOC容器负责实例化、配置和管理这些对象，通过配置文件（如XML配置）或注解来描述对象的依赖关系和属性。

Spring AOP（Aspect-Oriented Programming，面向切面编程）： Spring AOP是Spring框架的另一个核心组件，它提供了一种通过横向切割应用程序的方式来实现横切关注点的模块化。面向切面编程允许开发人员将与业务逻辑无关的功能（如日志记录、事务管理、安全性等）从主要的业务逻辑中分离出来，并将其定义为可重用的模块。Spring AOP通过在运行时动态地将这些横切关注点织入到目标对象的方法中，实现了横切关注点的透明化。

这两个核心组件共同构成了Spring框架的基础，使得应用程序的开发和维护更加简单和灵活。通过Spring IOC和AOP的支持，开发人员可以更好地实现松耦合、模块化和可测试的代码。除了这两个核心组件，Spring框架还提供了许多其他功能和扩展，如Spring MVC、Spring Security、Spring Data等，进一步增强了开发的便利性和功能性。

Spring AOP的工作原理是什么？ Spring AOP使用动态代理技术来实现切面的织入。对于基于接口的代理，Spring使用JDK动态代理；对于基于类的代理，Spring使用CGLIB代理。在运行时，Spring AOP会在目标对象的方法调用前、后或抛出异常时，将切面的增强逻辑织入到目标对象的方法中。

Spring AOP中的切点是什么？ 切点（Pointcut）定义了哪些方法或类应该被切入增强逻辑。它通过表达式或注解来指定连接点（Join Point），可以选择匹配某个包、类、方法等。切点是AOP中非常重要的概念，它决定了在何处应用切面的增强。

Spring AOP支持哪些类型的增强（Advice）？ Spring AOP支持以下类型的增强：

前置增强（Before advice）：在目标方法执行之前执行。

后置增强（After advice）：在目标方法执行之后执行，不考虑其结果。

返回增强（After-returning advice）：在目标方法正常返回结果之后执行。

异常增强（After-throwing advice）：在目标方法抛出异常之后执行。

环绕增强（Around advice）：在目标方法之前和之后执行，可以控制方法的执行。

如何在Spring配置文件中声明和配置切面和增强？ 在Spring配置文件中，可以使用<aop:config>元素声明AOP配置。通过<aop:aspect>元素定义切面，使用<aop:pointcut>元素定义切点，使用<aop:advisor>元素配置增强，可以使用不同的子元素（如<aop:before>、<aop:after>等）来指定增强的类型和具体逻辑。

什么是连接点（Join Point）和切入点（Pointcut）？ 连接点是应用程序中可以插入切面的特定点，如方法调用、方法执行、异常处理等。切入点是连接点的一个子集，它定义了切面在何处应用的规则，通过表达式或注解来匹配连接点。换句话说，切入点是一组连接点的集合。

Spring AOP支持哪些切点表达式？ Spring AOP支持基于AspectJ切点表达式语言的切点定义，常用的切点表达式包括：

execution：基于方法执行的切点表达式，如execution(\* com.example.service.\*.\*(..))。

within：基于类或包的切点表达式，如within(com.example.service.\*)。

this/target：基于目标对象类型的切点表达式，如this(com.example.service.MyService)。

args：基于方法参数的切点表达式，如args(java.lang.String)。

## **mysql除了表还有什么**

除了表（Table）之外，MySQL数据库中还有其他重要的数据库对象和组件，包括：

视图（View）：视图是基于一个或多个表的查询结果集，以虚拟表的形式呈现给用户。它提供了一种简化数据访问和查询的方式，并可以用作数据的过滤、聚合和封装。

索引（Index）：索引是对表中一个或多个列的值进行排序的数据结构，用于快速查找和检索数据。索引可以提高查询性能，减少数据扫描的开销。

存储过程（Stored Procedure）：存储过程是一组预定义的SQL语句集合，经过编译和存储在数据库中。它们可以接受输入参数、执行逻辑操作，并返回结果。存储过程可以在应用程序中被调用，提供了一种封装和重用数据库操作的方式。

函数（Function）：函数是一段可重用的SQL代码，接受输入参数并返回一个值。MySQL提供了多种内置函数，如数学函数、字符串函数、日期函数等。此外，还可以创建自定义函数来满足特定的业务需求。

约束（Constraint）：约束是定义在表上的规则，用于限制和保护数据的完整性。常见的约束类型包括主键约束、外键约束、唯一约束和检查约束。它们确保数据的有效性和一致性。

触发器（Trigger）：触发器是一段与表关联的自动执行的代码，通过定义在特定事件（如插入、更新、删除）发生时自动触发。触发器可以用于实现复杂的业务逻辑、数据验证和审计跟踪等功能。

## **分布式事务**

在分布式系统中，"CAP" 和 "BASE" 是两个不同的概念，它们描述了分布式事务的特性和性质。 CAP（一致性、可用性和分区容错性）：CAP 理论是由计算机科学家 Eric Brewer 提出的，用于描述分布式系统的三个重要特性。 一致性（Consistency）：指系统中的所有节点在同一时间具有相同的数据副本。即，如果一个节点修改了数据，那么所有其他节点在一定时间内都能看到这个修改。 可用性（Availability）：指系统在任何时间都能提供服务，即使出现节点故障或网络分区。 分区容错性（Partition tolerance）：指系统能够在网络分区的情况下继续正常运行，即使网络中的消息传递出现延迟或丢失。 CAP 理论指出，在分布式系统中，由于网络延迟和分区故障的存在，无法同时满足一致性、可用性和分区容错性这三个特性。在发生网络分区时，系统必须在一致性和可用性之间进行权衡。 BASE（基本可用、软状态和最终一致性）：BASE 是对传统 ACID（原子性、一致性、隔离性和持久性）事务模型的一种补充。BASE 强调系统在面对分布式环境下的柔性和可扩展性。

BASE理论是对CAP中的一致性和可用性进行一个权衡的结果，理论的核心思想就是：我们无法做到强一致，但每个应用都可以根据自身的业务特点，采用适当的方式来使系统达到最终一致性（Eventual consistency）。

基本可用性（Basically Available）：系统保持可用状态，尽管在出现故障或异常情况时，系统的可用性可能会降低。 软状态（Soft state）：系统中的状态可以在一段时间内是不确定的，即系统可能在特定时间点看到不同的数据副本或状态。 最终一致性（Eventual Consistency）：系统最终会达到一致的状态，但在分布式环境下，由于异步复制和延迟，数据的一致性可能需要一段时间来实现。 总结来说，CAP 理论描述了分布式系统的三个关键特性，而 BASE 提供了一种柔性的事务模型，强调可用性和最终一致性。CAP 和 BASE 可以看作是描述分布式事务中不同特性的两个概念，它们并不直接关联。

## **除了MySQL还用过什么数据库**

oracle

开源 vs. 商业产品：

MySQL是一个开源数据库，具有开放的社区支持和广泛的用户群体。它提供了免费的社区版和商业版，适用于中小规模应用和项目。

Oracle是一个商业数据库，由Oracle Corporation开发和支持。它提供了企业级功能和支持，适用于大规模和复杂的企业应用。

数据库规模：

MySQL主要面向中小型应用和项目，尤其适合于Web应用和简单的数据管理需求。

Oracle主要面向大型企业级应用和复杂的数据管理需求，具有更高的扩展性和处理能力。

数据库功能：

Oracle提供了丰富的功能和高级选项，如高级安全性、高可用性、数据分区、复制、集群和在线数据重组等。它支持复杂的事务处理和数据管理需求。

MySQL在功能上相对较简单，提供了基本的事务支持、索引、视图、存储过程等常见功能，但缺少一些高级特性。

性能：

MySQL在处理简单查询和大量读操作时通常具有较好的性能，尤其在高并发读取场景下表现出色。

Oracle在处理复杂查询、大量写操作和并发性能方面表现较好，特别适合于大型企业级应用和高负载环境。

可扩展性：

MySQL在水平扩展方面相对较好，可以通过分片、复制和主从配置等方式实现横向扩展。

Oracle在垂直和水平扩展方面都具备较强的能力，可以通过分区、集群、分布式数据库等方式实现大规模的扩展。

成本：

MySQL的社区版是免费的，商业版的价格相对较低，适用于预算有限的项目。

Oracle是商业产品，其许可和支持费用较高，适用于有更高预算和复杂需求的企业项目。

代码的不同

连接数据库：

MySQL使用com.mysql.jdbc.Driver作为JDBC驱动类，连接字符串的格式为jdbc:mysql://hostname:port/database。

Oracle使用oracle.jdbc.driver.OracleDriver作为JDBC驱动类，连接字符串的格式为jdbc:oracle:thin:@hostname:port:database。

数据类型：

MySQL和Oracle支持不同的数据类型，例如MySQL中的VARCHAR，在Oracle中对应的是VARCHAR2。

在处理日期和时间时，MySQL使用DATE、DATETIME和TIMESTAMP，而Oracle使用DATE和TIMESTAMP。

分页查询： Sequence）和触发器（Trigger）来实现自增主键的功能。

## **http接口 servlet mvc/boot dubbo openfeign的接口**

在Java开发中，有多种框架和技术可以用于构建HTTP接口。以下是其中几种常见的接口实现方式：

Servlet接口：

Servlet是JavaEE规范中定义的用于处理HTTP请求和响应的接口。通过实现Servlet接口并重写相关方法（如doGet()、doPost()等），可以创建基于Servlet的HTTP接口。Servlet接口提供了底层的HTTP请求处理和状态管理功能，但需要手动处理请求和响应的细节。

MVC框架（如Spring MVC和Spring Boot）：

MVC（Model-View-Controller）框架提供了更高级的抽象和功能，用于构建基于模型、视图和控制器的Web应用程序。通过使用注解、配置和约定，可以将HTTP请求映射到特定的处理方法，并使用模板引擎生成响应视图。MVC框架提供了更高级的路由、参数绑定、验证和异常处理等功能。

Dubbo：

Dubbo是一个分布式服务框架，用于构建高性能和可扩展的微服务架构。它提供了RPC（远程过程调用）的功能，使得不同服务之间可以通过网络进行通信。Dubbo的接口定义通常使用接口定义语言（IDL）（如Dubbo的自定义IDL或者使用Apache Thrift、Protocol Buffers等）进行描述，并通过配置文件或注解进行服务暴露和引用。

OpenFeign：

OpenFeign是一个声明式的HTTP客户端框架，用于简化服务间的HTTP通信。它通过定义接口并使用注解来描述HTTP请求，底层会自动生成实际的HTTP请求代码。OpenFeign可以与Spring Cloud等微服务框架集成，提供负载均衡、容错和服务发现等功能。

## **开启事务的方法**

当涉及到数据库操作时，使用事务可以确保一组相关操作的原子性、一致性、隔离性和持久性（ACID属性）。以下是关于JDBC事务和Spring事务的详细介绍：

JDBC事务：

JDBC（Java Database Connectivity）是Java提供的用于与关系型数据库进行交互的API。在JDBC中，可以使用以下方法来管理事务：

setAutoCommit(false)：通过将自动提交关闭，将所有数据库操作包含在一个事务中。

commit()：将当前事务中的所有操作提交到数据库，使其永久生效。

rollback()：回滚当前事务中的所有操作，取消之前的修改。

JDBC事务的使用步骤如下：

获取数据库连接（Connection）。

关闭自动提交（connection.setAutoCommit(false)）。

执行一系列数据库操作。

根据操作结果决定是提交事务（connection.commit()）还是回滚事务（connection.rollback()）。

关闭连接。

注意：在使用JDBC事务时，需要手动处理事务的开启、提交和回滚，以及异常处理。如果在事务中发生异常，需要在异常处理代码块中执行回滚操作。

Spring事务：

Spring框架提供了一种声明式事务管理的方式，通过在方法或类上使用@Transactional注解来管理事务。Spring事务的特点如下：

自动管理事务的开启、提交和回滚，无需手动编写相关代码。

支持声明式的事务定义和事务传播行为，通过注解参数进行配置。

可以与各种数据访问技术（如JDBC、JPA、Hibernate等）无缝集成。

使用Spring事务的步骤如下：

在配置文件中启用事务管理器（如DataSourceTransactionManager）。

在需要事务管理的方法上添加@Transactional注解，配置事务的属性（如隔离级别、传播行为等）。

执行一系列数据库操作，包含在带有@Transactional注解的方法内。

根据操作结果，Spring会根据事务配置自动提交或回滚事务。

Spring事务还支持事务切面（Transaction Aspect），它可以在方法执行前后织入额外的逻辑，如日志记录、性能监控等。

总结：JDBC事务需要手动管理事务的开启、提交和回滚，适用于简单的事务场景。而Spring事务通过声明式的方式自动管理事务，提供了更高层次的抽象和便利，适用于复杂的事务管理需求，并与各种数据访问技术集成。

## **异常的处理方式 注解配置异常**

在Java开发中，异常的处理方式主要有两种：使用try-catch块进行捕获和处理异常，或者使用throws关键字声明方法可能抛出的异常，将异常的处理责任交给调用者。下面分别介绍这两种方式。

使用try-catch块进行异常处理：

在可能会引发异常的代码块中，使用try块包裹，并在catch块中捕获和处理异常。catch块可以根据异常类型来捕获特定的异常，并采取相应的处理逻辑。

通常，可以在catch块中打印异常信息、记录日志、进行错误恢复操作或抛出新的异常，以便更好地处理异常情况。

示例代码如下：

javaCopy codetry {

// 可能引发异常的代码

} catch (ExceptionType1 e1) {

// 处理 ExceptionType1 类型的异常

} catch (ExceptionType2 e2) {

// 处理 ExceptionType2 类型的异常

} finally {

// 可选的 finally 块，用于执行清理操作

}

使用throws关键字声明方法可能抛出的异常：

在方法声明中使用throws关键字，列出方法可能抛出的异常类型。这样做的目的是将异常的处理责任交给方法的调用者，在调用方法的地方进行异常处理。

当方法中发生异常时，异常将被传播到方法的调用者，调用者可以选择使用try-catch块捕获并处理异常，或者继续向上一级调用者传播异常。

示例代码如下：

javaCopy codepublic void someMethod() throws ExceptionType1, ExceptionType2 {

// 可能抛出 ExceptionType1 或 ExceptionType2 类型的异常

}

注解配置异常： 在Spring框架中，可以使用@ExceptionHandler注解来配置全局异常处理或针对特定异常的处理。通过在类或方法上添加@ExceptionHandler注解，并指定要处理的异常类型，可以定义异常处理逻辑。 示例代码如下：

javaCopy code@ControllerAdvice

public class GlobalExceptionHandler {

@ExceptionHandler(Exception.class)

public ResponseEntity<String> handleException(Exception ex) {

// 处理异常逻辑

return new ResponseEntity<>("An error occurred", HttpStatus.INTERNAL\_SERVER\_ERROR);

}

@ExceptionHandler(MyCustomException.class)

public ResponseEntity<String> handleCustomException(MyCustomException ex) {

// 处理自定义异常逻辑

return new ResponseEntity<>(ex.getMessage(), HttpStatus.BAD\_REQUEST);

}

}

上述代码中的@ControllerAdvice注解用于定义全局异常处理类，@ExceptionHandler注解用于定义具体的异常处理方法。根据不同的异常类型，选择相应的处理逻辑，并返回适当的响应给客户端。

通过使用try-catch块和throws关键字，以及注解配置异常处理，可以灵活地处理不同类型的异常，并提供适当的异常处理策略。具体使用哪种方式取决于开发者对异常的处理需求和场景的不同。

## **char和varchar的区别**

在数据库中，char和varchar是用于存储字符数据的数据类型，它们之间有一些区别。

存储方式：char类型使用固定长度的存储方式，而varchar类型使用可变长度的存储方式。对于char类型，无论实际存储的字符数是多少，它都会占用固定长度的存储空间。而varchar类型则根据实际存储的字符数动态调整存储空间的长度，只占用实际需要的空间。

空间占用：由于char类型使用固定长度的存储方式，即使实际存储的字符数少于定义的长度，它仍然会占用整个固定长度的存储空间。而varchar类型则根据实际存储的字符数来决定占用的存储空间，所以相对于char类型，varchar类型可以节省存储空间。

存储效率：由于char类型使用固定长度的存储方式，对于访问和查询操作来说，在某些情况下可能会比varchar类型更高效。因为在定长存储中，数据库引擎可以直接根据偏移量计算数据的位置，而无需解析存储的长度信息。但是对于变长存储的varchar类型，需要解析存储的长度信息才能定位数据。

字符数限制：char类型有固定的字符数限制，无论实际存储的字符数是多少，都会占用定义的长度。而varchar类型可以存储可变长度的字符数据，可以根据需要存储不同长度的字符。

根据具体的需求，选择使用char还是varchar类型。如果你的数据具有固定长度且长度相对较短，可以考虑使用char类型以节省存储空间。如果数据长度不确定或变化较大，可以选择使用varchar类型以提高存储效率和节省空间。

## **事务的传播特性**

事务的传播特性（Transaction Propagation）是指在多个事务操作嵌套或相互调用时，事务如何进行传播和交互的规则和方式。在Java中，通常通过事务管理器（如Spring的事务管理器）来定义和控制事务的传播特性。

以下是一些常见的事务传播特性：

REQUIRED（默认）：如果当前存在事务，则加入该事务，如果不存在事务，则新建一个事务。这是最常用的传播特性，适用于大多数情况。

SUPPORTS：如果当前存在事务，则加入该事务，如果不存在事务，则以非事务的方式执行。

MANDATORY：要求当前必须存在事务，否则抛出异常。

REQUIRES\_NEW：无论当前是否存在事务，都会挂起当前事务，创建一个新的事务来执行。

NOT\_SUPPORTED：以非事务的方式执行操作，如果当前存在事务，则挂起该事务。

NEVER：以非事务的方式执行操作，如果当前存在事务，则抛出异常。

NESTED：如果当前存在事务，则在嵌套事务内执行。嵌套事务是外部事务的一部分，可以独立地进行提交或回滚。如果外部事务提交，则嵌套事务也会提交；如果外部事务回滚，则嵌套事务会回滚到保存点。

## **什么是分布式事务**

## **分布式事务实现方式**

实现分布式事务的方式有多种，以下是一些常见的实现方式：

两阶段提交（Two-Phase Commit，2PC）：2PC是一种经典的分布式事务协议。它包括协调者和参与者两种角色。在事务提交过程中，协调者负责协调各参与者的状态，并最终决定是否提交或回滚事务。

补偿事务（Compensating Transaction）：补偿事务是一种通过执行补偿操作来实现分布式事务的一致性的方式。当事务执行过程中发生错误或失败时，通过执行预定义的补偿操作来撤销或逆转之前的操作，以维持系统的一致性。

基于消息的事务（Message-Based Transaction）：该方式使用消息队列作为事务的协调机制。事务参与者将操作请求发布到消息队列中，协调者订阅并接收到请求后，根据事务的一致性规则决定是否提交或回滚事务。

TCC（Try-Confirm-Cancel）事务：TCC是一种基于补偿机制的事务管理方式。它将事务拆分为三个阶段：尝试（Try）、确认（Confirm）和取消（Cancel）。在尝试阶段，参与者会执行业务逻辑，并预留资源。在确认阶段，确认操作将提交事务。如果某个参与者无法确认，则进入取消阶段，执行相应的补偿操作。（tcc是基于编程的方式）

Saga模式：Saga模式是一种用于处理长事务和保证分布式系统中事务一致性的架构模式。它通过将复杂的事务拆分为一系列小的、可逆的事务片段，每个片段都有对应的补偿操作，以实现分布式事务的可靠执行。

这些实现方式各有优缺点，选择合适的实现方式取决于具体的业务需求和系统设计。同时，还有一些开源的分布式事务管理框架，如Seata、Atomikos、Narayana等，可以简化分布式事务的实现过程。

6. 三阶段提交 3pc 在中间插入一个准备阶段

## **spring事务是怎么实现的**

Spring框架提供了一种简化和统一的方式来实现事务管理，主要通过以下几个关键组件来实现：

事务管理器（Transaction Manager）：Spring事务管理器负责管理事务的创建、提交和回滚等操作。它提供了统一的接口，可以与不同的数据访问技术（如JDBC、Hibernate、JPA等）进行集成。

事务定义（Transaction Definition）：事务定义用于定义事务的属性，包括隔离级别、传播行为、超时设置等。Spring使用@Transactional注解或XML配置来定义事务的属性。

事务切面（Transaction Aspect）：Spring利用AOP（面向切面编程）技术，将事务管理逻辑织入到应用程序中。通过在方法或类上使用@Transactional注解，Spring会自动创建代理对象，并在方法调用前后添加事务管理逻辑。

事务通知（Transaction Advice）：事务通知是指在方法调用前后执行的代码块，用于开始、提交或回滚事务。Spring提供了Before、After和Around等类型的事务通知。

事务切点（Transaction Pointcut）：事务切点定义了在哪些方法上应用事务通知。它可以基于方法的名称、返回类型、参数等条件来进行匹配。

事务上下文（Transaction Context）：事务上下文是指在事务管理过程中保存事务状态的对象。它存储了当前事务的属性、连接信息等，用于协调事务的管理和执行。

通过上述组件的配合，Spring能够在方法调用过程中自动处理事务的开始、提交和回滚等操作。开发者只需使用@Transactional注解或XML配置来标识事务的边界和属性，Spring会自动为其创建事务代理，并在方法调用时触发相应的事务管理逻辑。

值得注意的是，Spring的事务管理是基于底层数据访问技术的，如JDBC或Hibernate等。它不直接处理分布式事务，但可以与其他分布式事务管理器（如Seata）进行集成，实现分布式事务的管理。

## **数据库设计的三范式**

数据库设计的三范式是指关系数据库设计中的三个规范化层次，用于消除数据冗余并提高数据的一致性和完整性。这些范式被称为第一范式（1NF）、第二范式（2NF）和第三范式（3NF）：

第一范式（1NF）： 第一范式要求数据库中的每个字段都是原子性的，不可再分的。换句话说，每个字段只能包含一个值，而不能包含多个值或者重复的值。这样可以避免数据冗余和复杂性。

第二范式（2NF）： 第二范式在满足第一范式的基础上，要求数据库中的非主键字段必须完全依赖于整个主键，而不能只依赖于主键的一部分。如果有部分依赖存在，就需要将相关的字段分离出来形成新的表。

第三范式（3NF）： 第三范式在满足第二范式的基础上，进一步要求数据库中的非主键字段之间不能存在传递依赖关系。换句话说，非主键字段之间应该直接依赖于主键，而不是依赖于其他非主键字段。如果存在传递依赖，就需要进行分离，将相关的字段分别放置在不同的表中。

范式的设计原则旨在提高数据库的数据完整性和一致性，减少数据冗余，并简化数据操作和维护。但是，在某些情况下，过度的范式规范可能会导致性能问题，因此在实际设计中需要权衡和综合考虑范式化和反范式化的需求。

## **springmvc分哪几层**

Spring MVC（Model-View-Controller）是一种基于Java的Web开发框架，它采用了经典的三层架构模式，将应用程序的不同职责划分到不同的层次上。Spring MVC分为以下几层：

控制器层（Controller Layer）： 控制器层负责处理客户端请求并根据请求的类型、参数等决定如何处理请求。它接收用户请求并调用适当的服务层组件进行处理，并将处理结果返回给客户端。控制器层通常由Spring MVC框架的控制器组件（如注解@Controller或基于接口的控制器）实现。

服务层（Service Layer）： 服务层提供了业务逻辑的处理和服务。它接收控制器层传递的请求，协调多个领域对象（如实体对象、数据访问对象等）的操作，并进行必要的业务处理。服务层通常包含一些服务接口（Service Interface）和其实现类（Service Implementation），用于封装具体的业务逻辑。

数据访问层（Data Access Layer）： 数据访问层负责与数据库或其他数据存储系统进行交互，并执行数据的读取、写入和更新等操作。它封装了对数据的具体访问细节，提供了持久化数据的功能。数据访问层通常使用一些数据访问对象（Data Access Object，DAO）或者使用Spring的ORM框架（如Hibernate、MyBatis）来进行数据访问。

视图层（View Layer）： 视图层负责将处理结果呈现给用户，通常是通过生成HTML、JSON、XML等格式的响应来展示数据。视图层接收控制器层传递的数据，将其展示出来并呈现给用户。在Spring MVC中，视图层通常使用JSP、Thymeleaf、Freemarker等模板引擎来生成动态的视图。

这些层次之间通过Spring MVC框架提供的注解、配置和依赖注入等机制进行协作和交互，实现了松耦合的架构。

服务层和数据访问层也称为模型层

## **mvc分哪几层**

MVC（Model-View-Controller）是一种常见的软件架构模式，用于将应用程序的不同组成部分分离开来，以实现更好的可维护性和可扩展性。MVC架构通常被用于构建Web应用程序，其中包含三个主要的组件层：模型（Model）、视图（View）和控制器（Controller）。

下面是MVC架构的三个组件层的概述：

模型（Model）：

模型层是应用程序的数据和业务逻辑的表示。

模型负责处理应用程序的数据，包括数据的获取、存储、验证和处理等。

模型层通常包含数据模型、数据库操作、业务逻辑和服务等。

视图（View）：

视图层是用户界面的表示。

视图负责将模型的数据以适当的方式呈现给用户，如HTML页面、XML文档、图形界面等。

视图通常负责接收用户的输入，并将其发送给控制器进行处理。

控制器（Controller）：

控制器层负责处理用户的输入、协调模型和视图之间的交互。

控制器接收用户的请求，根据请求调用适当的模型处理业务逻辑，并选择合适的视图来呈现响应。

控制器还可以处理错误处理、身份验证、路由等任务。

## **异常的基类**

在Java中，所有异常类的基类是java.lang.Throwable。Throwable类是异常处理的根类，它是一个抽象类，定义了异常处理的基本行为和属性。

## **MySQL的几个链表查询方式**

在MySQL数据库中，有几种常见的链表查询方式，可以使用它们来连接多个表以获取所需的数据。以下是几个常用的链表查询方式：

内连接（INNER JOIN）： 内连接返回满足连接条件的匹配行。只有当两个表中的连接列具有匹配值时，才会返回结果。语法如下：

SELECT 列名

FROM 表1

INNER JOIN 表2 ON 连接条件;

左连接（LEFT JOIN）： 左连接返回左表中的所有行，以及满足连接条件的右表中的匹配行。如果右表中没有匹配的行，则会返回 NULL 值。语法如下：

SELECT 列名

FROM 表1

LEFT JOIN 表2 ON 连接条件;

右连接（RIGHT JOIN）： 右连接返回右表中的所有行，以及满足连接条件的左表中的匹配行。如果左表中没有匹配的行，则会返回 NULL 值。语法如下：

SELECT 列名

FROM 表1

RIGHT JOIN 表2 ON 连接条件;

全连接（FULL JOIN）： 全连接返回左表和右表中的所有行，无论是否有匹配的行。如果某个表中没有匹配的行，则会返回 NULL 值。语法如下：

SELECT 列名

FROM 表1

FULL JOIN 表2 ON 连接条件;

自连接（Self JOIN）： 自连接是将单个表视为两个独立的表来进行连接操作。它常用于处理具有层级结构的数据。语法如下：

SELECT 列名

FROM 表 t1, 表 t2

WHERE t1.列 = t2.列;

这些链表查询方式可以根据具体的需求选择使用，并根据表之间的关系设置合适的连接条件来获取所需的数据。

## **mysql语句优化**

使用索引： 索引是提高查询性能的关键。确保在查询条件中使用适当的列上的索引，以加快数据检索速度。可以使用EXPLAIN命令来分析查询语句并查看索引的使用情况。

优化查询语句： 编写高效的查询语句是关键。避免使用不必要的通配符，减少返回的数据量。合理使用WHERE子句、ORDER BY和LIMIT限制结果集的大小。考虑使用连接（JOIN）来代替子查询。

避免全表扫描： 全表扫描是性能瓶颈之一。通过合理创建索引、使用查询条件和优化查询语句来避免对整个表的扫描，以减少查询时间。

优化表结构： 合理设计数据库表结构可以提高查询性能。使用适当的数据类型和字段长度，避免使用过多的冗余字段。遵循数据库范式规则，确保数据的一致性和完整性。

避免使用过多的连接： 建立和关闭数据库连接是有成本的。尽量减少不必要的连接次数，考虑使用连接池来复用连接对象，以提高效率。

适当调整数据库缓冲区和缓存大小： 通过调整MySQL的缓冲区和缓存大小参数（如innodb\_buffer\_pool\_size、key\_buffer\_size）来优化内存使用和查询性能。

定期优化表： 使用OPTIMIZE TABLE命令可以进行表的优化，修复表碎片并提高性能。

监控和分析数据库性能： 使用MySQL提供的性能监控工具（如SHOW STATUS、SHOW PROCESSLIST、EXPLAIN等）来分析查询性能，并根据需要进行调整和优化。

## **Linux常用命令**

文件和目录操作：

ls：列出目录内容。

cd：切换当前工作目录。

pwd：显示当前工作目录的路径。

mkdir：创建新目录。

rm：删除文件或目录。

cp：复制文件或目录。

mv：移动文件或目录。

文件内容查看和编辑：

cat：显示文件内容。

less：按页查看文件内容。

head：显示文件开头部分。

tail：显示文件结尾部分。

grep：在文件中搜索指定模式。

vim（或nano）：编辑文件。

文件权限和用户管理：

chmod：修改文件权限。

chown：修改文件所属用户和组。

chgrp：修改文件所属组。

useradd -m：添加新用户。

passwd：修改用户密码。

userdel：删除用户。

系统信息和进程管理：

uname：显示系统信息。

whoami：显示当前用户。

ps：显示当前运行进程。

top：动态显示系统资源使用情况。

kill：终止指定进程。

网络相关：

ifconfig：显示和配置网络接口信息。

ping：检测与另一个主机的网络连接。

ssh：通过SSH协议远程登录到另一台计算机。

scp：通过SSH协议在本地和远程主机之间传输文件。

压缩和解压缩：

tar：创建或提取tar压缩文件。

gzip：压缩文件。

gunzip：解压缩文件。

其他实用工具：

wget：从网络下载文件。

curl：用于传输数据。

date：显示和设置系统日期和时间。

find：在文件系统中搜索文件。

history：显示最近使用过的命令历史。

## **rabbitmq使用场景**

异步任务处理： RabbitMQ可以作为任务队列，用于处理异步任务。应用程序可以将任务发布到RabbitMQ中，由后台的工作进程（消费者）异步地消费和处理这些任务。这种方式可以将任务的执行与请求的响应分离开来，提高系统的响应性能和可扩展性。

应用解耦和消息通信： RabbitMQ作为消息队列，可以实现应用程序之间的解耦和消息通信。不同的应用程序可以通过RabbitMQ进行异步的、松耦合的通信，提高系统的灵活性和可维护性。例如，可以使用RabbitMQ在微服务架构中进行服务间的通信，或者在分布式系统中进行事件驱动的消息传递。

日志收集和分发： RabbitMQ可以用于日志收集和分发，将日志消息发送到消息队列中，然后由消费者进行处理和存储。这种方式可以将日志的生成和处理分离开来，提高日志系统的可靠性和可扩展性。

流量控制和负载均衡： RabbitMQ可以用于流量控制和负载均衡，通过控制消息的生产和消费速率来平衡系统的负载。生产者可以根据消费者的处理能力和系统的负载情况来动态地调整消息的发送速度，避免因为大量请求导致系统的崩溃或性能下降。

数据同步和复制： RabbitMQ可以用于数据同步和复制。生产者将数据变更的消息发送到消息队列中，消费者接收并处理这些消息，实现数据的同步和复制。这种方式可以用于构建分布式系统的数据一致性机制，确保多个节点之间的数据一致性和可靠性。

任务调度和分发： RabbitMQ可以用于任务调度和分发。生产者将需要执行的任务发布到消息队列中，消费者根据任务的优先级和调度策略来选择并处理任务。这种方式可以实现任务的动态调度和分发，提高任务的执行效率和系统的吞吐量。

## **vue用的那些插件**

Vue Router：提供了路由功能，用于实现单页应用的前端路由管理。

Vuex：用于管理应用程序的状态（state）的集中式状态管理模式。它可以帮助组织和管理大型Vue.js应用中的组件之间的共享状态。

Axios：用于进行基于Promise的HTTP请求的插件，可以方便地与后端API进行数据交互。

Vue CLI：提供了一套用于快速搭建Vue.js项目的命令行工具。它可以帮助生成项目模板、自动化构建、运行开发服务器等。

Vuetify：一个基于Vue.js的Material Design风格的UI框架，提供了丰富的可重用组件，帮助开发者构建漂亮的用户界面。

Element UI：另一个流行的基于Vue.js的UI框架，提供了一套美观且易用的组件库，方便快速构建Web应用程序。

Vue-i18n：用于实现国际化（i18n）的插件，可以在Vue.js应用程序中轻松管理多语言文本。

VueXfire：用于与Firebase实时数据库集成的插件，可以实时同步Vue.js应用程序的状态和数据。

Vue Test Utils：用于编写单元测试和集成测试的工具库，帮助开发者测试Vue.js组件的行为和功能。

Vue Devtools：一个浏览器插件，提供了一个开发者工具面板，用于调试和分析Vue.js应用程序。

## **用户登录注册**

## **token是用什么加密的，后台需要解密吗**

Token通常是使用加密算法进行签名，而不是加密。常见的加密算法包括HMAC-SHA256、RSA、JWT等。

在身份验证和授权场景中，通常使用JSON Web Token (JWT)作为Token的表示形式。JWT由三部分组成：头部（Header）、载荷（Payload）和签名（Signature）。头部和载荷部分是Base64编码的JSON对象，包含了关于Token的元数据和有效载荷数据。签名部分使用加密算法对头部、载荷和密钥进行处理生成的摘要，以确保Token的完整性和验证身份。

后台在验证Token时需要进行解析和验证签名。后台会使用密钥（对称加密）或公钥（非对称加密）与接收到的Token进行解码，并进行签名验证。通过解析和验证Token，后台可以获取其中的信息，如用户身份、访问权限等。

需要注意的是，Token的签名验证可以防止Token被篡改，但不会加密Token的内容。所以在设计和使用Token时，应避免将敏感信息直接存储在Token的载荷中，以确保数据的安全性。

## **Vue怎么与后端进行交互**

使用Axios库： 首先，在Vue.js项目中使用Axios库，它是一个基于Promise的HTTP客户端，可以用于发送HTTP请求。

发送HTTP请求： 在Vue组件中，可以使用Axios发送HTTP请求到后端API。例如，可以在Vue的methods中定义一个方法来发送GET请求：

javascriptCopy codeimport axios from 'axios';

export default {

// ...

methods: {

fetchData() {

axios.get('/api/data')

.then(response => {

// 处理后端返回的数据

console.log(response.data);

})

.catch(error => {

// 处理请求错误

console.error(error);

});

}

}

};

处理后端响应： 在Axios的then回调中，可以处理后端返回的数据。根据后端API的设计，可以将数据展示在Vue组件中，更新组件的状态或执行其他操作。

发送其他类型的请求： 除了GET请求，还可以使用Axios发送POST、PUT、DELETE等其他类型的请求。例如，发送POST请求可以用以下方式：

javascriptCopy codeaxios.post('/api/data', { data: 'example' })

.then(response => {

// 处理后端返回的数据

})

.catch(error => {

// 处理请求错误

});

可以根据后端API的要求和业务逻辑，使用合适的HTTP方法和传递参数。

配置请求的基本URL： 在Vue.js项目中，可以在Axios中配置请求的基本URL，以避免在每个请求中都重复指定完整的URL。例如，在Vue项目的入口文件或配置文件中配置基本URL：

javascriptCopy codeimport axios from 'axios';

axios.defaults.baseURL = 'http://example.com/api';

然后，在发送请求时，只需提供相对路径即可：

javascriptCopy codeaxios.get('/data')

.then(response => {

// 处理后端返回的数据

})

.catch(error => {

// 处理请求错误

});

通过使用Axios库，Vue.js可以方便地与后端进行交互，发送HTTP请求并处理后端返回的数据。需要根据具体的后端API设计和业务需求来进行配置和处理。同时，还可以使用拦截器（interceptors）等高级功能来处理请求和响应的拦截、认证、错误处理等。

## **tomcat部署项目**

打包项目：首先，将项目打包为WAR文件（Web Application Archive）。WAR文件是一种特殊的压缩文件格式，用于打包和部署Java Web应用程序。

停止Tomcat服务器：在部署新的应用程序之前，建议先停止Tomcat服务器以避免冲突。

将WAR文件复制到Tomcat的webapps目录：将打包好的WAR文件复制到Tomcat服务器的webapps目录下。可以通过命令行或图形界面进行复制。

启动Tomcat服务器：启动Tomcat服务器，它将自动检测并部署位于webapps目录下的WAR文件。

访问部署的应用程序：一旦Tomcat启动完成，可以通过浏览器访问部署的应用程序。默认情况下，访问路径的格式为http://localhost:8080/应用程序名称，其中localhost是服务器地址，8080是Tomcat的默认端口号。

配置项目：根据项目的需要，可能需要进行一些配置。例如，数据库连接、日志配置、权限设置等。这通常涉及修改项目的配置文件，如web.xml、application.properties等。

注意事项：

确保Tomcat的版本与项目的要求兼容。

在部署之前，确保Tomcat服务器已正确安装和配置。

在部署新的应用程序之前，确保Tomcat服务器已停止。

如果项目依赖外部库或资源文件，确保将它们正确地包含在WAR文件中或将其放置在Tomcat服务器上的适当位置。

## **jar包怎么运行**

运行JAR包：在命令行中使用java -jar命令来运行JAR包。例如，假设JAR包的文件名为myapp.jar，可以使用以下命令来运行：

Copy code

java -jar myapp.jar

如果JAR包有一些参数或配置选项，可以在命令中添加相应的选项。例如：

cssCopy code

java -jar myapp.jar --option1=value1 --option2=value2

JAR包开始运行：一旦运行命令被执行，JAR包将开始运行，并在命令行中显示相应的输出和日志信息。

## **异常处理怎么做的**

捕获异常：使用try-catch语句块捕获可能抛出异常的代码块。将可能引发异常的代码放在try块中，并在catch块中捕获异常。

javaCopy codetry {

// 可能抛出异常的代码

} catch (Exception e) {

// 异常处理逻辑

}

处理异常：在catch块中编写异常处理逻辑来处理捕获到的异常。处理逻辑可以根据异常类型和业务需求进行不同的操作，例如打印错误日志、回滚事务、发送错误信息等。

javaCopy codetry {

// 可能抛出异常的代码

} catch (Exception e) {

// 异常处理逻辑

e.printStackTrace(); // 打印异常堆栈信息

// 其他处理操作

}

处理多个异常类型：可以在catch块中处理多个异常类型。可以根据不同的异常类型执行不同的处理逻辑。

javaCopy codetry {

// 可能抛出异常的代码

} catch (IOException e) {

// 处理IO异常

} catch (SQLException e) {

// 处理数据库异常

} catch (Exception e) {

// 处理其他异常

}

使用finally块（可选）：可以使用finally块来定义无论异常是否发生都需要执行的代码块。例如，释放资源、关闭连接等操作可以放在finally块中。

javaCopy codetry {  
 // 可能抛出异常的代码  
} catch (Exception e) {  
 // 异常处理逻辑  
} finally {  
 // 无论异常是否发生，都会执行的代码  
}

抛出异常（可选）：在某些情况下，可以通过throw关键字主动抛出异常。这通常在自定义异常或在特定条件下需要中断程序流程时使用。

javaCopy codeif (condition) {  
 throw new CustomException("Error message");  
}

通过合理地使用异常处理机制，可以改善程序的可靠性和健壮性。异常处理应该根据具体的业务需求和异常类型来进行适当的处理和恢复操作。在处理异常时，可以根据需要记录日志、提供友好的错误信息、回滚事务、关闭资源等，以确保程序的正确性和可用性。

## **最常抛出的异常，什么时候会导致这种异常**

NullPointerException（空指针异常）：当尝试在一个空对象上调用方法、访问属性或数组时，会抛出空指针异常。这通常发生在以下情况下：

对一个未初始化的对象引用进行操作。

对一个已经被回收的对象进行操作。

对一个返回空值的方法结果进行操作。

IllegalArgumentException（非法参数异常）：当方法接收到一个不合法的参数时，会抛出非法参数异常。这可能是因为参数不符合预期的范围、格式或约束条件，导致方法无法继续执行。

ArrayIndexOutOfBoundsException（数组下标越界异常）：当尝试访问数组中不存在的索引位置时，会抛出数组下标越界异常。例如，当使用一个大于数组长度或小于0的索引访问数组元素时，会导致该异常。

ClassCastException（类转换异常）：当试图将一个对象强制转换为不兼容的类型时，会抛出类转换异常。这通常发生在向下转型时，如果对象的实际类型与所需的类型不兼容，则会导致该异常。

IOException（输入输出异常）：当处理输入和输出操作时发生错误时，会抛出输入输出异常。例如，读取文件时文件不存在、网络连接断开等情况都可能导致该异常。

SQLException（数据库异常）：当在数据库操作中发生错误时，例如连接数据库失败、执行SQL语句错误等，会抛出数据库异常。

## **技术网站**

## **redis常用操作命令**

SET key value: 设置指定 key 的值为给定的 value。

GET key: 获取指定 key 的值。

DEL key: 删除指定的 key。

EXISTS key: 检查指定的 key 是否存在。

TTL key: 获取指定 key 的过期时间（以秒为单位）。

KEYS pattern: 查找所有符合给定模式的 key。

INCR key: 将指定 key 的值递增 1。

DECR key: 将指定 key 的值递减 1。

HSET key field value: 在指定的哈希表中设置字段(field)的值。

HGET key field: 获取指定哈希表中指定字段的值。

HDEL key field: 删除指定哈希表中的一个或多个字段。

LPUSH key value1 [value2 ...]: 将一个或多个值插入到列表的头部。

RPUSH key value1 [value2 ...]: 将一个或多个值插入到列表的尾部。

LPOP key: 移除并返回列表的头元素。

RPOP key: 移除并返回列表的尾元素。

SADD key member1 [member2 ...]: 向集合添加一个或多个成员。

SMEMBERS key: 返回集合中的所有成员。

SREM key member1 [member2 ...]: 从集合中移除一个或多个成员。

ZADD key score1 member1 [score2 member2 ...]: 向有序集合添加一个或多个成员，或者更新已存在成员的分数。

ZRANGE key start stop: 根据索引区间返回有序集合中的成员。

## **left join与join的区别**

在关系型数据库中，JOIN是一种用于将两个或多个表根据某些条件关联起来的操作。JOIN操作根据指定的连接条件将匹配的行组合在一起，并返回结果集。

JOIN操作有几种类型，其中包括INNER JOIN和LEFT JOIN。它们之间的主要区别在于如何处理连接条件不满足的情况。

INNER JOIN: INNER JOIN只返回两个表中连接条件匹配的行。它根据连接条件从两个表中选择匹配的行，并返回这些匹配的行作为结果集。如果连接条件不满足，则不返回任何行。

例如，假设有两个表A和B，执行以下查询：

cssCopy codeSELECT \*  
FROM A  
INNER JOIN B ON A.id = B.id;

结果集将只包含表A和表B中id列匹配的行。

LEFT JOIN: LEFT JOIN返回左表（左侧的表，通常是在FROM子句中写在前面的表）中的所有行，以及与右表（右侧的表，通常是在JOIN子句中写在后面的表）中连接条件匹配的行。如果连接条件不满足，则在结果集中返回右表的列为NULL的行。

例如，假设有两个表A和B，执行以下查询：

cssCopy codeSELECT \*  
FROM A  
LEFT JOIN B ON A.id = B.id;

结果集将包含表A中的所有行，以及与表B中id列匹配的行。如果没有匹配的行，则在结果集中以NULL填充表B的列。

总结：

INNER JOIN只返回满足连接条件的行，而LEFT JOIN返回左表的所有行以及满足连接条件的右表的行。

INNER JOIN只返回匹配的行，而LEFT JOIN在连接条件不满足时返回NULL值的行。

选择使用哪种类型的JOIN取决于数据关系和查询需求。如果想要包含左表的所有行并且有些行可能没有匹配的右表数据，则可以使用LEFT JOIN。如果只需要返回满足连接条件的匹配行，则可以使用INNER JOIN。

## **try-catch finally**

try：包含可能抛出异常的代码块。

catch：用于捕获并处理指定类型的异常。可以有多个 catch 块，分别处理不同类型的异常。

finally：无论是否发生异常，都会执行的代码块。通常用于执行必要的清理操作，如关闭数据库连接、释放资源等。

try-catch-finally 的执行流程如下：

首先，执行 try 块中的代码。

如果在 try 块中发生了异常，并且该异常的类型匹配到某个 catch 块中指定的异常类型，程序将跳转到匹配的 catch 块，并执行对应的异常处理代码。然后继续执行 catch 块之后的代码。

如果在 try 块中发生了异常，但是没有匹配到任何 catch 块中指定的异常类型，异常将不会被处理，程序会跳转到包含该 try-catch 结构的外层代码中，继续执行后续的代码。

无论是否发生异常，finally 块中的代码都会被执行。即使在 try 块中使用了 return 语句或抛出另一个异常，finally 块中的代码也会在跳出方法之前被执行。

## **前端Vue分页组件**

vue-paginate: 一个简单易用的Vue分页组件，支持基本的分页功能和自定义样式。GitHub链接：<https://github.com/TahaSh/vue-paginate>

vue-pagination: 一个轻量级的Vue分页组件，提供基本的分页功能和样式自定义选项。GitHub链接：<https://github.com/matfish2/vue-pagination>

vuejs-paginate: 一个用于Vue.js的简单分页组件，支持基本的分页和自定义样式。GitHub链接：<https://github.com/lokyoung/vuejs-paginate>

vue-simple-pagination: 一个简单易用的Vue分页组件，提供基本的分页功能和自定义样式选项。GitHub链接：<https://github.com/JonathanDn/vue-simple-pagination>

v-pagination: 一个功能强大的Vue分页组件，支持多种分页模式和自定义样式选项，还提供了一些高级功能，如页码跳转、每页显示数量选择等。GitHub链接：<https://github.com/lokyoung/vuejs-paginate>

## **vue路由模式**

哈希模式（Hash Mode）： 在哈希模式下，Vue路由器将使用URL的哈希部分（#）来模拟完整的URL路径。哈希模式的URL看起来像这样：http://example.com/#/path。

优点：哈希模式兼容性好，可以在不同的服务器环境中正常运行。

缺点：URL中带有哈希，不够美观；不支持服务端渲染（SSR）。

要使用哈希模式，可以在Vue路由器的配置中设置mode属性为hash：

javascriptCopy codeconst router = new VueRouter({

mode: 'hash',

// ...

});

历史模式（History Mode）： 在历史模式下，Vue路由器使用HTML5 History API来管理URL，不再使用哈希。历史模式的URL看起来像常规的URL：http://example.com/path。

优点：URL更加美观，没有哈希部分；支持服务端渲染（SSR）。

缺点：需要服务器配置支持，以处理直接访问路由的情况。

要使用历史模式，可以在Vue路由器的配置中设置mode属性为history：

javascriptCopy codeconst router = new VueRouter({

mode: 'history',

// ...

});

选择使用哈希模式还是历史模式取决于项目的需求和服务器环境。如果项目不需要支持服务端渲染，并且服务器配置允许，历史模式通常是更好的选择。如果项目需要更好的兼容性或无法进行服务器配置更改，则可以使用哈希模式。

## **keep-live**

keep-alive 是 Vue.js 的一个内置组件，用于缓存和管理动态组件的状态，以提高组件的性能和渲染效率。它可以将包裹的组件保持在内存中，而不是每次重新渲染。

使用 keep-alive 组件的主要步骤如下：

将需要缓存的组件包裹在 <keep-alive> 标签中，例如：

htmlCopy code<keep-alive>

<YourComponent />

</keep-alive>

在需要访问缓存组件的地方，使用 <router-view> 组件来渲染。这样，当路由切换到对应的组件时，如果该组件已经被缓存，将直接从缓存中取出并渲染，而不是重新创建组件实例。

keep-alive 组件还提供了一些属性，用于进一步控制缓存的行为，其中常用的属性包括：

include：指定需要缓存的组件名称，可以是一个字符串或一个正则表达式。

exclude：指定不需要缓存的组件名称，可以是一个字符串或一个正则表达式。

max：指定最大缓存组件的数量，当缓存的组件超过这个数量时，最久没有被访问的组件会被销毁。

例如，下面的示例中，将会缓存名称为 "Home" 和 "About" 的组件，其他组件不会被缓存：

htmlCopy code<keep-alive :include="['Home', 'About']">

<router-view />

</keep-alive>

使用 keep-alive 可以有效减少组件的销毁和创建，提高页面的响应速度和用户体验。然而，需要注意的是，对于一些需要实时更新的组件或具有副作用的组件（如表单组件），可能不适合使用 keep-alive 缓存，因为缓存的组件在重新渲染时不会触发生命周期钩子函数和一些事件。因此，在使用 keep-alive 时，需要根据实际需求和组件的特性进行权衡和选择。

## **分布式锁和分布式事务**

分布式锁和分布式事务是在分布式系统中常用的两个概念，它们解决了不同的问题。

分布式锁是一种用于协调分布式系统中并发访问共享资源的机制。在分布式系统中，多个节点同时访问共享资源可能导致数据不一致或冲突的问题。分布式锁通过在共享资源的访问过程中引入互斥机制，确保在同一时间只有一个节点可以访问该资源，从而解决并发访问的问题。常见的实现方式包括基于数据库的锁、基于缓存的锁（如Redis分布式锁）等。分布式锁的设计和实现需要考虑锁的粒度、可重入性、锁的超时机制等因素。

分布式事务是一种用于处理分布式系统中跨多个资源和服务的事务操作的机制。在分布式系统中，多个节点可能涉及到对多个数据源或服务的并发操作，需要保证数据的一致性和完整性。分布式事务通过协调参与者之间的操作，确保一组相关的操作要么全部成功提交，要么全部回滚，从而保证整个操作的一致性。常见的分布式事务解决方案包括两阶段提交（2PC）、补偿事务（TCC）、最大努力通知（Best Effort Delivery）等。分布式事务的设计和实现需要考虑事务的边界、协调者和参与者的角色、事务的隔离级别等因素。

需要注意的是，分布式锁和分布式事务是两个不同的概念，解决的问题也不同。分布式锁主要解决并发访问共享资源的问题，而分布式事务主要解决分布式系统中多个节点之间的事务一致性问题。在设计和实现分布式系统时，根据具体的需求和情况，可以选择使用分布式锁和分布式事务来解决相应的问题。

## **支付操作有几步操作**

在Java开发中进行支付操作通常涉及以下几个步骤：

创建订单：首先需要创建一个订单对象，包含订单号、商品信息、金额等必要的订单信息。

发起支付请求：根据支付方式（如支付宝、微信支付等），使用相应的支付接口或SDK发起支付请求。请求中通常包括订单号、支付金额、支付回调URL等参数。

获取支付结果：支付请求成功后，会返回一个支付结果，通常是一个支付凭证，如支付宝的交易号或微信支付的预支付交易会话标识（prepay\_id）。可以根据该支付凭证进行后续的支付结果查询和处理。

支付结果查询：为了确保支付的准确性和及时性，可以定期或根据需要查询支付结果。通过支付接口提供的查询接口或SDK，传入订单号或支付凭证等参数，获取支付结果的查询响应。

验证支付结果：在收到支付结果查询响应后，需要验证支付结果的有效性和一致性。通常需要校验支付金额、订单号、交易状态等信息，以确保支付结果的准确性。

处理支付回调：支付平台在支付成功后会异步通知支付结果，通常通过回调URL进行通知。开发者需要在服务器端接收并处理支付平台的回调请求，验证回调的合法性，并更新订单状态等相关操作。

返回支付结果：在支付结果处理完成后，将支付结果返回给前端或调用方。可以通过回调返回、接口响应等方式将支付结果返回给用户或系统。

这些步骤是通用的支付操作流程，具体的实现方式会根据所采用的支付平台和接口进行调整。支付操作中需要注意支付安全性、支付结果的可靠性以及支付回调的处理等方面的考虑。同时，根据业务需求和支付平台的特性，可能需要进行一些额外的操作，如退款、退款查询、订单关闭等。

## **mybatis中动态sql要用到什么标签**

MyBatis中使用动态SQL可以根据条件来生成不同的SQL语句，提供了以下几个标签用于实现动态SQL：

if：用于条件判断，根据条件动态生成SQL语句片段。示例：

xmlCopy code<select id="getUser" resultType="User">

SELECT \* FROM user

<where>

<if test="name != null">

AND name = #{name}

</if>

<if test="age != null">

AND age = #{age}

</if>

</where>

</select>

choose/when/otherwise：类似于Java中的switch语句，根据不同的条件选择不同的分支。示例：

xmlCopy code<select id="getUser" resultType="User">

SELECT \* FROM user

<where>

<choose>

<when test="name != null">

AND name = #{name}

</when>

<when test="age != null">

AND age = #{age}

</when>

<otherwise>

AND status = 1

</otherwise>

</choose>

</where>

</select>

foreach：用于遍历集合或数组，生成重复的SQL语句片段。示例：

xmlCopy code<select id="getUsersByIds" resultType="User">

SELECT \* FROM user

WHERE id IN

<foreach collection="ids" item="id" open="(" separator="," close=")">

#{id}

</foreach>

</select>

trim/where/set：用于对SQL语句进行修剪，去除不必要的关键字或条件。示例：

xmlCopy code<update id="updateUser" parameterType="User">

UPDATE user

<set>

<if test="name != null">

name = #{name},

</if>

<if test="age != null">

age = #{age},

</if>

</set>

WHERE id = #{id}

</update>

这些动态SQL标签可以根据业务需求和条件灵活地生成不同的SQL语句，避免了硬编码和重复代码的问题，提高了SQL语句的可读性和维护性。可以根据具体的需求选择合适的标签来构建动态SQL语句。

## **vue怎么调后台接口**

在Vue中调用后台接口通常涉及以下几个步骤：

安装依赖：首先，在Vue项目中需要使用到网络请求库，比如axios、vue-resource等。可以通过npm或yarn安装所需的依赖库。

创建服务：接下来，可以创建一个服务文件（例如api.js），用于封装与后台接口的交互逻辑。在服务文件中，可以使用上述网络请求库发送请求到后台接口，并处理响应结果。

配置后台接口地址：在服务文件中，需要配置后台接口的地址，包括接口的URL、请求方法、请求参数等。可以根据后台接口的要求，使用合适的请求方法，如GET、POST等。

调用接口：在Vue组件中，可以引入服务文件，并调用其中定义的接口方法来发起请求。可以通过调用服务方法传递请求参数，并处理响应结果。

下面是一个简单的示例，演示如何在Vue中调用后台接口：

javascriptCopy code// api.js

import axios from 'axios';

const api = axios.create({

baseURL: 'http://example.com/api', // 后台接口的基本URL

timeout: 5000 // 请求超时时间

});

export const getUser = (id) => {

return api.get(`/user/${id}`); // 发起GET请求，获取用户信息

};

export const saveUser = (user) => {

return api.post('/user', user); // 发起POST请求，保存用户信息

};

vueCopy code<!-- User.vue -->

<template>

<div>

<h2>User Details</h2>

<div>

<label>ID:</label>

<span>{{ user.id }}</span>

</div>

<div>

<label>Name:</label>

<span>{{ user.name }}</span>

</div>

<!-- Other user details -->

</div>

</template>

<script>

import { getUser } from './api.js';

export default {

data() {

return {

user: {} // 用户信息

};

},

mounted() {

this.fetchUser(); // 在组件加载时获取用户信息

},

methods: {

fetchUser() {

const userId = 1; // 要获取的用户ID

getUser(userId)

.then(response => {

this.user = response.data; // 更新用户信息

})

.catch(error => {

console.error(error);

});

}

}

};

</script>

在上面的示例中，api.js文件封装了两个接口方法：getUser用于获取用户信息，saveUser用于保存用户信息。在User.vue组件中，通过调用fetchUser方法来获取用户信息，并将结果显示在模板中。

当fetchUser方法被调用时，会使用getUser接口方法发起GET请求，并根据响应结果更新组件的user数据。如果请求失败，则会在控制台打印错误信息。

## **vue接受什么类型参数**

在Vue中，组件可以接受多种类型的参数，包括：

String（字符串）：可以将字符串作为组件的参数传递，例如 name="John"。

Number（数字）：可以将数字作为组件的参数传递，例如 age="25"。

Boolean（布尔值）：可以将布尔值作为组件的参数传递，例如 isVIP="true"。

Object（对象）：可以将对象作为组件的参数传递，例如 user="{ name: 'John', age: 25 }"。

Array（数组）：可以将数组作为组件的参数传递，例如 numbers="[1, 2, 3, 4, 5]"。

Function（函数）：可以将函数作为组件的参数传递，例如 onClick="handleClick"。

Component（组件）：可以将其他组件作为参数传递，例如 childComponent="ChildComponent"。

除了以上基本类型，Vue组件还可以接受其他自定义类型的参数，如枚举类型、日期类型等。在Vue组件中，可以通过在组件的props属性中声明接受的参数，以及指定参数的类型和默认值等。

例如，在Vue组件中声明接受一个字符串类型的参数：

vueCopy code<template>

<div>

<h2>Hello, {{ name }}</h2>

</div>

</template>

<script>

export default {

props: {

name: {

type: String,

required: true

}

}

};

</script>

在使用该组件时，可以通过传递一个名为name的字符串参数来指定name属性的值：

vueCopy code<template>

<div>

<my-component name="John" />

</div>

</template>

<script>

import MyComponent from './MyComponent.vue';

export default {

components: {

MyComponent

}

};

</script>

在上述示例中，通过传递name="John"的参数，将字符串"John"作为name属性的值传递给MyComponent组件，从而在组件内部可以通过name访问到该值。

需要注意的是，对于接受的参数，可以根据实际需求设置是否为必需参数（required: true）以及设置默认值。此外，还可以通过validator属性自定义参数的验证规则，以确保接受的参数符合预期的类型和值范围。

## **写过数据列表吗**

在前端开发中，数据列表是非常常见的组件，用于展示和管理大量的数据。通常，数据列表会包含以下功能：

数据展示：将后台返回的数据以列表的形式展示在页面上，每一行显示一条数据记录。

分页：对于大量数据，可以使用分页功能将数据分页显示，提供翻页操作和显示当前页码、总页数等信息。

排序：支持根据列头点击进行升序或降序排序，方便用户按照特定字段对数据进行排序。

筛选：提供筛选条件，根据用户选择的条件对数据进行过滤，只显示符合条件的数据。

搜索：提供搜索框或搜索表单，允许用户输入关键词对数据进行搜索，显示与关键词相关的数据。

编辑/删除：对每一行数据提供编辑和删除功能，允许用户修改数据或删除数据记录。

批量操作：支持对选中的多条数据进行批量操作，如批量删除、批量导出等。

导出/导入：支持将数据导出为Excel或其他格式，或者支持导入数据文件进行批量操作。

数据刷新：提供刷新按钮或自动刷新功能，允许用户手动或自动更新数据列表。

以上是常见的数据列表功能，具体功能和样式可以根据实际需求进行调整和扩展。在实现数据列表时，通常会使用Vue的组件化开发方式，将数据列表拆分为多个可复用的组件，提高代码的可维护性和复用性。同时，也会结合后端接口进行数据的获取、操作和更新等。

## **MD5 能解密吗 微信为什么要用MD5加密**

MD5是一种哈希算法，它是单向不可逆的，即无法通过已经生成的MD5摘要来还原原始数据。因此，MD5是无法被解密的。

微信在某些场景中使用MD5加密是为了数据的完整性和安全性。具体来说，微信在与第三方进行数据传输时，为了确保数据在传输过程中不被篡改，会使用MD5进行数据签名，然后将签名结果发送给对方。对方在接收到数据后，也会使用相同的方式进行签名，并与接收到的签名进行比对，以验证数据的完整性和确保数据没有被篡改。

虽然MD5不可逆，但它可以用于验证数据的完整性。通过对数据进行MD5加密，可以生成一个固定长度的摘要，即签名。如果接收方生成的签名与发送方的签名一致，就可以确定数据在传输过程中没有被篡改。

## **同一台机器Linux传文件到另一台机器需要用到什么命令**

rz上传 sz下载

## **不同一台机器Linux传文件到另一台机器需要用到什么命令（从机器A复制一个文件到机器B）**

scp命令：scp命令用于在不同的Linux机器之间进行文件传输，基于SSH协议进行安全的文件拷贝。

语法：

cssCopy code

scp [选项] [源文件] [目标地址]

示例：

rubyCopy code

scp username@machineA:/path/to/source/file username@machineB:/path/to/destination/directory

这将从机器A上的 /path/to/source/file 文件复制到机器B上的 /path/to/destination/directory 目录下。

需要提供机器A和机器B的用户名、IP地址或主机名，以及源文件的路径和目标目录的路径。

rsync命令：rsync命令用于在本地机器和远程机器之间同步文件和目录。

语法：

cssCopy code

rsync [选项] [源文件/目录] [目标地址]

示例：

rubyCopy code

rsync -avz username@machineA:/path/to/source/file username@machineB:/path/to/destination/directory

这将通过SSH连接将机器A上的 /path/to/source/file 文件同步到机器B上的 /path/to/destination/directory 目录下。

需要提供机器A和机器B的用户名、IP地址或主机名，以及源文件的路径和目标目录的路径。

## **Tomcat windows 的启动命令**

使用 catalina.bat run 命令启动Tomcat时，命令窗口将一直保持打开状态，显示Tomcat的日志输出。而使用 startup.bat 命令启动Tomcat时，命令窗口将在Tomcat成功启动后自动关闭。

## **axios有哪些配置**

URL：请求的 URL 地址。

Method：请求的 HTTP 方法，如 GET、POST、PUT、DELETE 等。

Params：请求的 URL 参数，可作为查询字符串附加在 URL 上。

Data：请求的数据体，适用于 POST、PUT 等方法发送数据。

Headers：请求的 HTTP 头部信息，可以设置自定义的头部字段。

Timeout：请求的超时时间，超过该时间将中断请求。

Response Type：指定响应数据的类型，如 JSON、文本等。

Transform Request：对请求数据进行处理的转换函数。

Transform Response：对响应数据进行处理的转换函数。

Auth：配置请求的身份验证信息，包括用户名和密码。

Proxy：设置请求的代理服务器。

Interceptors：拦截器，允许在发送请求或响应返回之前进行拦截和处理。

## **get请求和post请求有什么区别**

语义：GET请求用于从服务器获取数据，而POST请求用于向服务器提交数据。

参数位置：GET请求的参数以查询字符串的形式附加在URL的末尾，而POST请求的参数包含在请求体中。

请求长度限制：GET请求的参数长度通常有限制，因为URL长度有限制（通常为几千个字符），而POST请求没有明确的长度限制。

安全性：GET请求的参数会附加在URL上，因此在浏览器的历史记录、代理服务器日志等地方都会留下明文记录，相对来说不太安全；而POST请求的参数在请求体中，相对更安全一些。

幂等性：GET请求是幂等的，多次请求同一个URL获取的结果是一样的，不会对服务器产生副作用；而POST请求不是幂等的，每次请求可能对服务器进行状态修改或资源创建。

缓存：GET请求可以被浏览器缓存，如果请求的URL和参数与之前的请求相同，则可以直接从缓存中获取响应；而POST请求默认是不会被浏览器缓存的。

## **响应返回的数据一般是什么格式的**

JSON（JavaScript Object Notation）：JSON是一种轻量级的数据交换格式，易于阅读和编写，也易于解析和生成。在Web开发中，JSON是最常用的数据格式之一，常用于前后端之间的数据交互。

XML（eXtensible Markup Language）：XML是一种标记语言，用于描述和传输结构化数据。在过去的Web开发中，XML较为常见，但随着JSON的流行，使用XML的场景有所减少。

文本（Plain Text）：有时候，响应返回的数据可能是纯文本格式，如普通的文本内容或CSV（逗号分隔值）等。

HTML（Hypertext Markup Language）：当请求的是一个网页时，响应返回的数据通常是HTML格式，用于展示网页内容。

二进制（Binary）：有时候，响应返回的数据可能是二进制格式，如图片、音频、视频等文件。

## **购物车最多能支持多少数据**

硬件资源：购物车的数据量受到服务器硬件资源的限制。服务器的内存、处理器性能和存储容量都会对购物车的数据量产生影响。

数据库性能：购物车的数据存储通常使用数据库来管理。数据库的性能、索引设计和查询优化等方面都会影响购物车的数据处理能力。

网络传输：购物车数据在前端和后端之间进行传输。网络带宽、延迟和数据传输效率等因素也会影响购物车数据的处理速度。

业务需求：购物车数据量的上限也会根据具体业务需求进行设定。一般来说，购物车的数据量应该足够满足用户需求，同时不过度消耗系统资源。

## **购物车有五个商品支付的时候突然有一个商品没库存了怎么去控制事务**

库存检查：在进行支付前，对购物车中的每个商品进行库存检查。可以通过查询库存信息或调用库存服务接口来确认商品的库存情况。

悲观锁：在进行库存检查时，使用悲观锁锁定相关商品的库存记录，确保其他并发操作无法修改库存。如果商品库存不足，事务将无法继续执行。

回滚事务：如果发现有商品库存不足，需要回滚整个支付事务，确保购物车中的所有商品都没有被扣减库存。回滚事务将撤销之前的数据库操作，使数据回到事务开始前的状态。

事务管理器：使用事务管理器来控制支付事务的提交和回滚。事务管理器可以在支付过程中监控事务状态，如果发现库存不足的情况，自动回滚事务。

事务消息：可以使用事务消息队列，将购物车中的支付请求发送到消息队列中。库存服务订阅消息队列，对支付请求进行处理，并在处理过程中进行库存检查。如果库存不足，事务消息可以自动回滚支付事务。

## **控制事务用那个注解**

@Transactional：这是Spring框架提供的注解，用于控制事务的提交和回滚。通过在方法或类上添加@Transactional注解，可以将该方法或类中的操作包装成一个事务。当方法执行过程中发生异常时，事务会被回滚，数据库操作会撤销到事务开始前的状态。

@Transactional(rollbackFor = Exception.class)：这是在Spring框架中使用@Transactional注解时的一种常见配置。rollbackFor属性指定了在哪些异常发生时触发事务回滚，默认情况下，只有RuntimeException及其子类异常会触发回滚。通过设置rollbackFor = Exception.class，可以让所有类型的异常都触发事务回滚。

## **try-catch块能跟throw一起用吗**

可以

在Java中，可以在try块中编写可能抛出异常的代码，并在catch块中捕获并处理这些异常。如果在try块中的代码执行过程中发生了异常，可以使用throw语句抛出一个新的异常对象，然后在catch块中捕获并处理该异常。

## **SpringBoot怎么实现分页的**

当前页数 每页条数 数据总数 总页数

## **Vue的生命周期**

beforeCreate：在实例被创建之前调用。此时实例的数据观测和事件机制尚未初始化。

created：在实例创建完成后调用。此时实例已完成数据观测，可以访问数据、计算属性、方法等，但尚未挂载到DOM上。

beforeMount：在实例挂载到DOM元素之前调用。此时模板编译已完成，但尚未将模板渲染成最终的DOM结构。

mounted：在实例挂载到DOM元素后调用。此时实例已经被挂载到DOM上，可以操作DOM元素。

beforeUpdate：在数据更新之前调用。当数据变化时，此钩子在重新渲染之前被调用，可以在此阶段进行状态更新。

updated：在数据更新之后调用。此时DOM已更新，可以对更新后的DOM进行操作。

beforeDestroy：在实例销毁之前调用。此时实例还可以访问，可以进行一些清理工作，如取消定时器、解绑事件等。

destroyed：在实例销毁之后调用。此时实例及其所有的指令、事件监听器等都已被销毁。

除了上述的核心生命周期钩子外，还有一些其他钩子函数，如activated（keep-alive组件激活时调用）、deactivated（keep-alive组件停用时调用）等，用于特定的场景和功能。

这些生命周期钩子函数可以让我们在不同的阶段执行相应的操作，比如在数据初始化阶段做一些初始化操作，或者在组件销毁前清理资源。通过钩子函数，可以更好地控制组件的生命周期和行为。

## **created跟mounted的区别**

created和mounted是Vue组件生命周期中的两个阶段，它们之间有一些区别。

created是在组件实例被创建之后调用的钩子函数。在created阶段，组件实例已经被创建，可以访问到实例的数据、方法、计算属性等，但是组件尚未被挂载到DOM上，此时组件的模板编译和DOM渲染都还未开始。

mounted是在组件被挂载到DOM上之后调用的钩子函数。在mounted阶段，组件已经被挂载到DOM上，可以通过this.$el访问到组件的根DOM元素，可以对DOM进行操作、绑定事件等。此阶段适合执行需要访问DOM的操作，比如初始化第三方库、绑定全局事件等。

可以总结如下：

created阶段适合进行组件的初始化操作，比如初始化数据、调用API获取数据、监听事件等。此时组件尚未挂载到DOM上，无法进行DOM操作。

mounted阶段适合进行与DOM相关的操作，比如修改DOM样式、绑定事件、操作第三方库等。此时组件已经挂载到DOM上，可以进行DOM操作。

需要注意的是，mounted阶段表示组件已经完全初始化并挂载到DOM上，是访问和操作DOM的最佳时机。而created阶段是在组件实例创建后执行的，此时可能还没有生成完整的DOM结构，因此不适合进行与DOM相关的操作。

## **SpringBoot怎么实现统一异常处理**

在Spring Boot中实现统一异常处理可以通过以下步骤：

创建一个自定义异常类，用于表示业务相关的异常。该异常类可以继承自RuntimeException或其他合适的异常类，或者实现Exception接口。

javaCopy codepublic class BusinessException extends RuntimeException {

// 构造方法和其他自定义逻辑

// ...

}

创建一个全局异常处理器类，用于捕获和处理所有的异常。该类需要使用@ControllerAdvice注解进行标记，并定义相应的异常处理方法。

javaCopy code@ControllerAdvice

public class GlobalExceptionHandler {

@ExceptionHandler(Exception.class)

public ResponseEntity<Object> handleException(Exception ex) {

// 处理异常的逻辑

// ...

return new ResponseEntity<>(responseBody, HttpStatus.INTERNAL\_SERVER\_ERROR);

}

@ExceptionHandler(BusinessException.class)

public ResponseEntity<Object> handleBusinessException(BusinessException ex) {

// 处理业务异常的逻辑

// ...

return new ResponseEntity<>(responseBody, HttpStatus.BAD\_REQUEST);

}

}

在上述代码中，@ControllerAdvice注解表示这是一个全局异常处理器类。@ExceptionHandler注解用于标记具体的异常处理方法，可以根据不同的异常类型编写相应的处理逻辑，并返回合适的响应。

可以在全局异常处理器中使用@ResponseBody注解将响应数据直接返回给客户端，或者返回一个自定义的响应实体类。

通过以上步骤，当发生异常时，全局异常处理器会捕获异常并执行相应的处理逻辑，返回给客户端合适的响应。可以根据具体的业务需求，在异常处理器中添加更多的异常处理方法，并根据异常类型进行不同的处理。

需要注意的是，全局异常处理器的优先级是高于其他异常处理方式的，可以确保所有异常都经过统一的处理逻辑。

## **数据库中in与exist的区别**

在数据库中，IN和EXISTS是两种常用的查询条件，它们的主要区别如下：

IN运算符：IN用于在查询中指定一个值列表，并检查被查询的列是否在该列表中。IN条件通常用于简单的值匹配查询。

示例：

sqlCopy code

SELECT \* FROM table\_name WHERE column\_name IN (value1, value2, value3);

这将返回满足列 column\_name 值等于 value1、value2 或 value3 的行。

IN运算符适用于已知的固定值列表，并且可以一次匹配多个值。

EXISTS子查询：EXISTS用于在查询中指定一个子查询，并检查子查询的结果是否存在。EXISTS条件通常用于确定是否存在满足特定条件的记录。

示例：

sqlCopy code

SELECT column1 FROM table1 WHERE EXISTS (SELECT column2 FROM table2 WHERE condition);

这将返回满足子查询条件的记录。

EXISTS子查询通常用于判断一个查询中是否存在符合某种条件的记录，可以根据子查询的结果动态决定主查询的结果。

关于区别，主要有以下几点：

IN运算符是针对一个列与一个固定值列表进行匹配，而EXISTS是针对一个子查询的结果进行判断。

IN适用于已知的固定值列表，而EXISTS适用于动态的条件判断。

IN通常用于简单的值匹配查询，而EXISTS通常用于判断是否存在满足特定条件的记录

## **spring作用域**

Singleton（默认）：在整个应用程序中只存在一个实例。每次请求该Bean时，都返回同一个实例。

Prototype：每次请求该Bean时，都创建一个新的实例。

Request：每个HTTP请求都会创建一个新的实例，该实例仅在当前请求范围内可见。不同的请求之间不共享同一个实例。

Session：每个HTTP会话都会创建一个新的实例，该实例在整个会话期间可见。不同的会话之间不共享同一个实例。

Global Session：在基于portlet的web应用程序中，每个全局HTTP会话都会创建一个新的实例。

Application：在整个web应用程序上下文中只存在一个实例，类似于Singleton作用域，但适用于Web环境。

WebSocket：每个WebSocket连接都会创建一个新的实例，该实例在连接期间可见。

## **断电问题**

以上级为主的对账（数据比对）

## **spring中事务的传播特性**

在 Spring 中，事务传播行为（Transaction Propagation Behavior）定义了一个方法调用中发生事务的方式。以下是事务的七种传播特性： REQUIRED（默认）：如果当前存在事务，则加入到当前事务中执行，如果当前没有事务，则创建一个新的事务。 SUPPORTS：如果当前存在事务，则加入到当前事务中执行，如果当前没有事务，则以非事务方式执行。 MANDATORY：如果当前存在事务，则加入到当前事务中执行，如果当前没有事务，则抛出异常。 REQUIRES\_NEW：无论当前是否存在事务，都创建一个新的事务并在它自己的事务中执行，如果有当前事务，则将其挂起。 NOT\_SUPPORTED：以非事务方式执行操作，如果当前存在事务，则将其挂起。 NEVER：以非事务方式执行操作，如果当前存在事务，则抛出异常。 NESTED：如果当前存在事务，则在嵌套事务中执行，嵌套事务独立于外部事务，如果当前没有事务，则创建一个新的事务。 这些传播特性可以在 @Transactional 注解中的 propagation 属性中指定，例如 @Transactional(propagation = Propagation.REQUIRED)。 需要注意的是，事务传播特性仅在方法之间嵌套调用时才会产生影响。如果方法直接被同一个类中的另一个方法调用，而不是通过代理对象调用，则事务传播特性不会生效。 希望能够解答你的问题！如果还有其他疑问，请随时提问。

## **HTTP状态码：**

HTTP（Hypertext Transfer Protocol）是用于在网络上传输超文本的协议，它使用状态码来表示请求和响应的处理结果。以下是一些常见的 HTTP 状态码及其含义：

1xx Informational（信息性状态码）：服务器正在处理请求，需要进一步操作。

100 Continue：客户端应继续发送请求。

101 Switching Protocols：服务器要求切换协议。

2xx Success（成功状态码）：请求已成功处理。

200 OK：请求成功。

201 Created：请求已被创建，并且成功。

204 No Content：服务器成功处理请求，但未返回内容。

3xx Redirection（重定向状态码）：需要进一步操作以完成请求。

301 Moved Permanently：被请求的资源已永久移动到新位置。

302 Found：被请求的资源暂时移动到新位置。

304 Not Modified：资源未修改，可使用缓存的版本。

4xx Client Errors（客户端错误状态码）：请求包含语法错误或无法完成请求。

400 Bad Request：请求有语法错误或无法被服务器理解。

401 Unauthorized：请求需要身份验证。

404 Not Found：请求的资源未找到。

405：是方法不对post，get

5xx Server Errors（服务器错误状态码）：服务器在处理请求时发生内部错误。

500 Internal Server Error：服务器遇到了意料不到的情况，无法完成请求。

502 Bad Gateway：作为网关或代理服务器的服务器从上游服务器收到无效响应。

503 Service Unavailable：服务器当前无法处理请求，一般用于过载维护情况。

## **连接过程：**

客户端根据：

1. 域名解析

发起TCP的3次握手

Web浏览器向Web服务器发送http请求报文 一旦建立了TCP连接，Web浏览器就会向Web服务器发送请求命令。例如：GET/sample/hello.jsp HTTP/1.1。 一个HTTP请求报文由请求行（request line）、请求头部（headers）、空行（blank line）和请求数据（request body）4个部分组成。

HTTP 连接过程可以分为以下几个步骤：

解析 URL：客户端解析目标网址（URL），提取出主机名、端口号、路径等信息。

建立 TCP 连接：使用解析得到的主机名和端口号，客户端与服务器之间建立 TCP 连接。这是一个三次握手的过程，确保双方都能正确地进行通信。

发起 HTTP 请求：客户端向服务器发送 HTTP 请求。请求中包含请求方法（GET、POST 等）、请求头部（例如 Accept、User-Agent 等）以及请求体（对于 POST 请求）。

服务器处理请求：服务器收到请求后会根据请求的内容进行处理，可能涉及查询数据库、读取文件或执行其他逻辑操作。

服务器返回响应：服务器将处理结果封装成 HTTP 响应，包含响应状态码、响应头部（例如 Content-Type、Content-Length 等）以及响应体（存放实际的响应内容）。

接收响应：客户端接收到服务器返回的响应。

处理响应：客户端根据响应状态码判断请求是否成功，若成功则解析响应头部和响应体，提取所需的数据。

关闭连接：HTTP 是无状态的协议，每个请求和响应都是独立的，完成一次通信后，客户端与服务器之间的 TCP 连接可以关闭。

# 什么叫跨域：

axios发送的请求地址和当前发送者的ip和端口不一样，不是同源的叫跨域。

# 乐观锁：timestamp和vertion是解决数据丢失，数据覆盖

# 可重复读：幻读问题=》一个事务进行保存，第二个可以删除，看不见可以进行删除

# 串行化： 一个客户端操作a表，别的客户端就不能操作a表，用commit操作之后才可以操作，但是可以操作b表

# 一致性：转账，总额是500，转账之后也是500。

# Dubbon的运行原理

下面是 Dubbo 的运行原理的详细解释：

@DubboReference(timeout = 10000,check = false,retries = 4)

Dubbo 的详细运行原理：

服务接口定义：首先，您需要定义服务接口，该接口定义了服务提供者将要暴露给消费者使用的方法和参数。

注册中心：Dubbo 使用注册中心来管理服务提供者和消费者之间的关系。注册中心是服务注册和发现的中心化存储，允许服务提供者将自己注册到中心，并允许消费者查询可用的服务提供者。常用的注册中心包括 ZooKeeper、Etcd、Consul 等。

服务提供者注册：服务提供者在启动时，将自己的服务信息注册到注册中心。这包括服务接口、IP 地址、端口号等信息。注册中心将这些信息存储起来，以便消费者能够发现和调用服务。

服务消费者订阅：服务消费者在启动时，通过向注册中心订阅服务，获取所需的服务列表。注册中心将可用的服务提供者地址列表返回给消费者。

负载均衡：服务消费者从服务提供者的地址列表中选择一个合适的提供者进行调用。Dubbo 提供了多种负载均衡策略，如随机、轮询、最少活跃调用等，以确保请求能够均匀地分布到不同的服务提供者上。

远程调用：一旦服务消费者选择了要调用的服务提供者，它将通过网络将请求发送给提供者。Dubbo 使用高性能的通信框架（如 Netty）进行远程调用。

服务调用链处理：Dubbo 提供了拦截器机制，允许开发人员在服务调用链中进行拦截和处理。这些拦截器可以用于实现诸如日志记录、安全验证、性能监控等功能。

服务治理：Dubbo 提供了丰富的服务治理功能，包括容错处理、超时控制、路由规则、集群容错等。这些功能可以提高系统的稳定性和可靠性。

结果返回：服务提供者处理请求后，将结果返回给服务消费者。Dubbo 支持同步调用和异步调用，消费者可以根据需要选择合适的调用方式。

高可用性与故障转移：Dubbo 提供了多种故障转移策略，以应对服务提供者发生故障或不可用的情况。当服务提供者无法正常工作时，Dubbo 可以自动切换到其他

以上是 Dubbo 的基本运行原理。Dubbo 提供了一套完整的 RPC 通信框架，使得分布式系统中的服务能够高效、稳定地进行通信和调用

# Proxy.newInstance(); //作用是获取实现类对象

You:

# java 死锁 解决办法

ChatGPT:

Java中的死锁是指多个线程因为互相持有对方所需的资源而无法继续执行的状态。解决Java死锁的常见方法包括以下几种：

预防死锁：

避免一个线程同时获取多个锁。

通过按照相同的顺序获取锁来避免死锁。

使用定时锁，使用tryLock(timeout)方法尝试获取锁，并在一段时间后放弃。

检测与恢复死锁：

可以使用工具来检测死锁，例如JDK提供的jstack命令行工具。

一旦检测到死锁，可以通过中断其中一个或多个线程来恢复程序的正常执行。可以使用Thread.interrupt()方法来中断线程。

避免死锁：

可以使用资源分级的策略，确保线程按照固定的顺序获取资源。

使用锁的超时机制，当线程无法在指定时间内获取到锁时，放弃锁并释放已获取的资源。

重构代码：

分析代码中可能导致死锁的部分，并进行重构，减少锁的粒度或者使用更合适的并发控制机制。

使用并发集合类，如ConcurrentHashMap，它可以减少锁的竞争。线程安全

HashMap在一个线程中可能获取别的锁的实例

设计良好的并发控制策略：

使用并发库中提供的工具类，如java.util.concurrent包下的Lock接口、Semaphore、CountDownLatch等，来实现更安全的并发控制。

请注意，这些方法可以帮助减少死锁发生的可能性，但不能保证完全消除死锁。因此，在设计和编写并发代码时，应该尽量避免产生死锁的情况，同时合理地使用并发控制机制来保证程序的正确性和性能。

You:

# mysql 如何避免死锁

ChatGPT:

在MySQL中，避免死锁的关键在于合理设计数据库架构和优化查询语句，以减少锁竞争和提高并发性。下面是一些常见的方法和技巧来避免死锁：

减少事务的持锁时间：

尽量缩小事务的范围，只在必要时才开始事务，并尽早提交或回滚。

在事务中只锁定需要修改的数据行，而不是锁定整个表或范围。

尽量避免在事务中进行大量的查询操作。

统一访问顺序：

如果多个事务需要同时访问多个表，确保它们以相同的顺序访问这些表，可以减少死锁的可能性。

使用合适的索引和查询优化：

为经常被查询的列创建合适的索引，以提高查询效率，减少锁的持有时间。

优化查询语句，避免不必要的全表扫描和排序操作。

使用较小的事务隔离级别：

如果应用程序对数据一致性的要求不是非常高，可以将事务隔离级别设置为较低的级别（如Read Committed），以减少锁的持有时间和范围。

通过分库分表减少锁竞争：

将大表拆分成多个较小的表，或者将数据分散到多个数据库中，可以减少不同事务之间的锁竞争。

监控和处理死锁：

设置合适的超时时间，当事务等待锁的时间超过设定的阈值时，可以主动回滚事务并重试。

定期监控数据库的死锁情况，及时发现并解决死锁问题。

需要注意的是，避免死锁是一个复杂的任务，取决于具体的数据库架构和业务场景。因此，在设计数据库和编写查询语句时，需要仔细考虑锁的使用和并发控制机制，结合具体情况采取适当的措施来避免死锁的发生。

You:

# 哪些操作会发生数据库死锁，举例说明

ChatGPT:

数据库死锁通常发生在多个事务同时竞争相同的资源时，例如表、行、页或索引等。以下是几个可能导致数据库死锁的常见操作示例：

并发更新相同的数据行：

事务A锁定数据行X并更新，然后事务B锁定数据行X并更新。

这种情况下，事务A和事务B会互相等待对方释放锁，从而导致死锁。

交叉锁定多个资源：

事务A锁定数据行X并请求锁定数据行Y，同时事务B锁定数据行Y并请求锁定数据行X。

如果两个事务的锁定顺序不一致，会导致死锁的发生。

循环依赖的资源锁定：

事务A锁定数据行X并请求锁定数据行Y，同时事务B锁定数据行Y并请求锁定数据行Z，最后事务C锁定数据行Z并请求锁定数据行X。

这种循环依赖的资源锁定会导致循环等待，从而形成死锁。

长时间持有锁：

事务A锁定数据行X并在处理过程中长时间不释放锁，同时事务B请求锁定数据行X。

如果事务A长时间持有锁，事务B可能会等待过长时间，超过了设定的超时时间，从而导致死锁。

大规模数据更新：

当一个事务需要更新大量数据行时，它可能会锁定过多的资源，导致其他事务无法获取到所需的资源而发生死锁。

这些是一些常见的操作示例，可能导致数据库死锁的情况。具体是否发生死锁取决于并发操作的顺序、事务隔离级别、锁的使用方式等因素。因此，在设计数据库架构和编写业务逻辑时，需要合理规划事务的范围和顺序，并考虑并发控制机制，以避免死锁的发生。

You:

不对吧 第一个

ChatGPT:

非常抱歉，我之前给出的例子并不准确。在MySQL中，多个事务并发更新相同的数据行通常不会导致死锁。相反，数据库死锁往往发生在以下情况下：

循环依赖的资源锁定：

事务A锁定资源X并请求锁定资源Y，同时事务B锁定资源Y并请求锁定资源X。

这种循环依赖的资源锁定会导致死锁的发生。

间隙锁引起的死锁：

事务A锁定某个范围的数据行（例如通过范围查询或使用SELECT ... FOR UPDATE），事务B同时请求锁定这个范围内的数据行中的另外一部分。

这种情况下，事务A和事务B可能会互相等待对方释放锁，从而导致死锁。

长时间持有锁：

事务A锁定资源X并在处理过程中长时间不释放锁，同时事务B请求锁定资源X。

如果事务A长时间持有锁，事务B可能会等待过长时间，超过了设定的超时时间，从而导致死锁。

需要注意的是，数据库死锁的发生是由于竞争资源的方式和顺序导致的，具体情况可能因数据库管理系统和锁机制的实现细节而有所不同。因此，在设计数据库架构和编写业务逻辑时，需要理解所使用的数据库的特定行为，以避免潜在的死锁情况。同时，通过合理设置事务隔离级别、优化查询和索引，以及合理设计事务范围等措施，也能够减少死锁的发生。

You:

第二个说的不对吧

ChatGPT:

非常抱歉，我给出的关于间隙锁引起死锁的解释是不准确的。实际上，在MySQL中，间隙锁不会直接导致死锁。请允许我更正并提供更准确的解释。

在MySQL中，间隙锁（Gap Lock）是一种用于防止并发事务插入数据范围之间的数据的锁定机制。它可以防止其他事务在同一范围内插入新的数据行，从而保持数据的一致性。然而，间隙锁本身不会导致死锁。

死锁通常发生在多个事务之间存在循环等待资源的情况下。在MySQL中，死锁的发生可能与间隙锁和行锁的组合使用有关，具体取决于事务的操作和隔离级别。以下是可能导致死锁的一种常见情况：

范围查询与插入：

事务A执行范围查询并获取了间隙锁，事务B在该间隙内插入新的数据行。

然后，事务A尝试获取行锁以访问范围内的数据行，而事务B也尝试获取间隙锁来插入数据行。

如果这两个事务同时请求对方持有的锁，形成了循环等待，就可能导致死锁的发生。

需要注意的是，这种情况下的死锁是由于范围查询和插入操作的竞争造成的，而间隙锁本身并不直接导致死锁。此外，死锁还可能发生在其他复杂的并发操作场景中，具体情况取决于并发事务的交互方式和锁的使用方式。

为了避免死锁，可以考虑以下方法：

优化事务操作的顺序，确保事务按照相同的顺序获取锁。

使用较小的事务隔离级别，减少锁的竞争范围。

合理设计数据库架构和索引，以减少并发事务之间的锁竞争。

定期监测和分析死锁情况，并根据具体情况进行调整和优化。

再次为之前的不准确解释向您道歉，并感谢您的指正。

# redis主从复制

主从配置：master是从服务器

slave是主服务器 设置主服务 slaveOf 指定ip+端口

如果主服务宕机，设置当前服务器为主服务器：slaveof no one

# redis分片

###### **redis的一致性hash:**

基于编程方面的

哈希环的构建：

将所有 Redis 节点在一个范围为 0 到 2^32-1 的环上均匀分布，这个范围可以使用哈希函数生成。

将每个节点映射到哈希环上的位置。可以使用节点的哈希值（例如，IP 地址的哈希值）与哈希环进行映射。通常使用一致性哈希算法来选择节点在环上的位置。

数据存储和查找：

当需要存储数据时，使用相同的哈希函数计算键的哈希值，并在哈希环上顺时针查找最近的节点。将数据存储到该节点上。

当需要查找数据时，根据键的哈希值，使用一致性哈希算法在哈希环上查找最近的节点，并从该节点上获取数据。

添加或删除节点：

当需要添加一个新节点时，只需计算新节点的哈希值，在哈希环上找到该节点应插入的位置。

当需要删除节点时，将该节点从哈希环上移除，并将该节点上的数据重新分配到相邻节点。

# actuator监控

# Serciter安全性框架

# jar上传私服

<project>

<!-- 其他配置信息 -->

<!--上传到私服里面-->

<distributionManagement>

<repository>

<id>company-releases</id>

<url>http://43.143.175.64:8081/repository/maven-releases/</url>

</repository>

<snapshotRepository>

<id>company-snapshots</id>

<url>http://43.143.175.64:8081/repository/maven-snapshots/</url>

</snapshotRepository>

</distributionManagement>

</project>

## **配置settings**

然后需要在 maven的setting 文件中，配置上传的 server id 用户名 密码，记住 id 和 pom.xml 中 repository 下的id 保持一致

<!-- 连接公司私服--> <server> <id>company-releases</id> <username>userNormal</username> <password>qwe123</password> </server> <server> <id>company-snapshots</id> <username>userNormal</username> <password>qwe123</password> </server>

还需要配置镜像

<mirror> <id>localnexus</id> <mirrorOf>\*,!central</mirrorOf> <name>本地Nexus仓库</name> <url><http://43.143.175.64:8081//repository/maven-public/></url> </mirror>

上述配置中，你需要注意以下几点：

在 <distributionManagement> 节点中配置私服的信息，包括 repository ID、名称和 URL。确保将 repositoryId 设置为私服的ID，以便与插件配置中的 repositoryId 保持一致。

在 <build> 节点下的 <plugins> 节点中配置 Maven Deploy 插件。它分为构建插件和部署插件两个部分。

构建插件部分设置 <skip>true</skip>，用于跳过默认的构建部署行为，避免重复部署。

部署插件部分通过添加 <executions> 节点来定义执行阶段，在 deploy 阶段触发插件的部署目标。

执行 mvn deploy 命令时，Maven 将会使用配置中指定的私服信息将构建产物上传到私服。



# 从私服中下载jar

<repositories>

<repository>

<id>company-releases</id>

<url>http://43.143.175.64:8081/repository/maven-public/</url>

<releases>

<enabled>true</enabled>

</releases>

<snapshots>

<enabled>true</enabled>

<updatePolicy>always</updatePolicy>

<checksumPolicy>fail</checksumPolicy>

</snapshots>

</repository>

</repositories>

# 一个项目引入另一个项目的模块

如下：

先创建A、B独立项目，然后再将A项目执行maven install打包，最后在B项目通过下面方式引入就可以了。

要注意版本，版本就是来源A项目POM文件的version。

<dependency>

<groupId>com.tszr</groupId>

<artifactId>A</artifactId>

<version>0.0.1-SNAPSHOT</version>

</dependency>

[IMG_290](javascript:void(0);)

[IMG_291](javascript:void(0);)

<project xmlns="http://maven.apache.org/POM/4.0.0"  
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">  
   <modelVersion>4.0.0</modelVersion>  
   <groupId>com.tszr</groupId>  
   <artifactId>A</artifactId>  
   <version>0.0.1-SNAPSHOT</version>  
   <name>A</name>  
   <description>A</description>  
      
   <parent>  
       <groupId>org.springframework.boot</groupId>  
       <artifactId>spring-boot-starter-parent</artifactId>  
       <version>2.1.3.RELEASE</version>  
   </parent>  
      
   <dependencies>  
       <dependency>  
           <groupId>org.springframework.boot</groupId>  
           <artifactId>spring-boot-starter-web</artifactId>  
       </dependency>  
   </dependencies>  
      
   <build>  
       <plugins>  
           <plugin>  
               <artifactId>maven-compiler-plugin</artifactId>  
               <configuration>  
                   <source>11</source>  
                   <target>11</target>  
               </configuration>  
           </plugin>  
       </plugins>  
   </build>  
</project>

[IMG_292](javascript:void(0);)

# 网关的作用是什么

统一管理微服务请求，权限控制、负载均衡、路由转发、监控、安全控制黑名单和白名单等

# vue跟jq的区别

1.

Vue.js 拥有丰富的插件生态系统，提供了许多官方和社区维护的插件，如路由器 (Vue Router)、状态管理器 (Vuex) 等，以满足不同的开发需求。

jQuery 也有大量的插件可供选择，但大部分是用于 DOM 操作、动画效果等方面的，相对于 Vue.js 的组件生态系统较为有限。

//1、原生JavaScript获得value值 var d1 = document.getElementById("demoId"); alert(d1.value);

//2、dom对象转换成jQuery对象 var $d2 = $(d1); alert(d2.val());

//3、jQuery对象转换成dom对象 var d3 = $d2[0];

2.

Vue.js 是一款现代的、基于组件化的 JavaScript 框架，采用了MVVM (Model-View-ViewModel) 架构模式。它通过数据绑定和组件化的方式来构建用户界面，将视图与数据状态进行关联。

jQuery 是一个功能强大的 JavaScript 库，主要用于简化 DOM 操作、事件处理、动画效果等。它基于传统的命令式编程风格，通过选择器和方法链来操纵 DOM 元素。

# \*http协议\*

\*什么是http协议？\*

http协议是基于tcp/ip的应用层协议,简单来说就是基于应用层的通信规范,双方要通信必须遵循这个规范,就是http协议\* \*\*Http协议工作原理?\*\* \*用户输入url,浏览器会先解析ip和端口号,然后与ip三次握手建立连接,服务器接受请求做处理进行响应,响应期间会四次挥手断开连接

\*长连接,短连接,无连接\*

\*长连接:\*一直建立连接,效率高,但是占用连接资源

\*短连接:\*是在无连接和长连接之间的,在一次连接内多次请求和多次响应,交互完数据之后断开连接,效率较高

\*无连接:\*建立连接后,请求一次相应一次就断开连接,效率很低

\*HTTP协议的特点是什么\*\*?\*

http协议是无状态的, 服务器端在处理相应请求后不会保留任何客户端的信息，每次请求都是独立的.

每一次交互都是请求和响应.

http协议应包括一个请求行,若干请求头,一个空行,请求实体

# \*String类常用的操作方法:\*

equals()方法:比较是否相等

length()方法:字符串的长度

trim()方法:去除字符串的前后空格

tocharArray()方法:字符串转数组

split()方法:把字符串按照规则分割,返回String类型数组

# \*常见端口号\*

\*tomcat\*:8080

\*vue\*：8080

\*nginx\*：80

\*https协议\*：443

\*http协议\*：80

\*redis\*:6379

\*zookeeper\*:2181

\*rabbitMq\*:5672

# \*Bean注入容器有哪些方式？\*

① 使用@Controller、@Service、@Repository、@Component 注解标注该类，然后启用@ComponentScan自动扫描

② 使用@Import 方法。

③ @Configuration与@Bean注解

# \*mybatis的优缺点？\*

\*优点：\*

\1. 易于上手和掌握

\2. sql写在xml里，便于统一管理和优化

\3. 解除sql语句和程序代码的耦合

\4. 解决了数据库字段和实体类字段不一致问题

\5. 提供xml标签，支持编写动态sql

\*缺点：\*

SQL语句工作量很大，尤其是字段多、关联表多时。更是如此

# 抽象类与接口的区别\*\*

1.\*\*一个类只能进行单继承，但可以实现多个接口。\*\*

2.\*\*有抽象方法的类，一定是抽象类，但是抽象类里面不一定有抽象方法。\*\*

3.\*\*接口里面所有的方法的默认修饰符为public abstract,接口里的成员变量默认修饰符\*\*

为：\*\*static final\*\*

# SpringMVC运行原理\*\*

1.\*\*客户端发送http请求，web服务器接收到这个请求,如果能匹配到DispatcherServlet的映射\*\*

路径\*\*(web.xml)文件,web容器交给DispatcherServlet进行处理\*\*

2.DispatcherServlet\*\*根据请求信息去找HandlerMapping(处理器映射器),HandlerMapping通\*\*

过对比映射\*\*,成功向DispatcherServlet返回一个Handler对象\*\*

3.DispatcherServlet\*\*继续将请求转发给HandlerAdapter(处理器适配器)，处理适配器根据包装\*\*

好的内容找到对应的资源对象\*\*,然后调用controller对应的方法\*\*

4.controller\*\*完成业务逻辑处理后,返回一个ModelAndView给前端控制器\*\*

5.\*\*前端控制器将ModelAndView传递到ViewResolver(视图解析器)进行解析ModelAndView,\*\*

找到相对应的视图

6.\*\*前端控制器将ModelAndView中的数据模型进行页面的视图渲染\*\*

# 双亲委派机制有什么优点：

1.避免了类的重复加载

\2. 保护了程序的安全性，防止核心的API被修改

一、你们的团队中是怎么使用git分支的

# Hotfixes:紧急修复分支

### 我们的git 分支分为 master > release （预生产环境） > dev（开发测试分支） > feature (feature 分支下 有最近增加模块分支)，每个开发人员在最近增加模块的这个分支下，有自己的分支，

合并分支时，我们先推到自己的分支，从自己的分支merge到feature分支，当开发工作基本完成，merge到dev 分支，进行测试，测试成功后，向我们的老大提出merge request ，申请成功merge到release分支，开发人员，测试人员，运营人员一起测试，测试成功后正式上线（由老大完成master 分支操作） ，上线后会进行TAG打版，出现特别大的漏洞会进行版本回退（回滚）

# 1.1 \*java Exception\*\*体系结构\*

java 异常是程序运行过程中出现的错误。Java把异常当作对象来处理，并定义一个基类java.lang.Throwable作为所有异常的超类。在Java API中定义了许多异常类,分为两大类，错误Error和异常Exception。其中异常类Exception又分为运行时异常(RuntimeException)和非运行时异常(非runtimeException)，也称之为不检查异常（Unchecked Exception）和检查异常（Checked Exception）。

1、 Error与Exception

Error是程序无法处理的错误，比如OutOfMemoryError、ThreadDeath等。这些异常发生时，Java虚拟机（JVM）一般会选择线程终止。

Exception是程序本身可以处理的异常，这种异常分两大类运行时异常和非运行时异常。程序中应当尽可能去处理这些异常。

2、 运行时异常和非运行时异常

a) 运行时异常: 都是RuntimeException类及其子类异常：

i. IndexOutOfBoundsException 索引越界异常

ii. ArithmeticException：数学计算异常

iii. NullPointerException：空指针异常

iv. ArrayOutOfBoundsException：数组索引越界异常

v. ClassNotFoundException：类文件未找到异常

vi. ClassCastException：造型异常（类型转换异常）

这些异常是不检查异常（Unchecked Exception），程序中可以选择捕获处理，也可以不处理。这些异常一般是由程序逻辑错误引起的。

非运行时异常:是RuntimeException以外的异常，类型上都属于Exception类及其子类。从程序语法角度讲是必须进行处理的异常，如果不处理，程序就不能编译通过。如：

IOException、文件读写异常

FileNotFoundException：文件未找到异常

EOFException：读写文件尾异常

MalformedURLException：URL格式错误异常

SocketException：Socket异常

SQLException：SQL数据库异常

# 这是springboot项目ServletContextListener监听器的实现

servetContextListen 监听器用于监听 Servlet 上下文的生命周期事件。Servlet 上下文是一个 Web 应用程序的运行环境，它在应用程序启动时创建，并在应用程序关闭时销毁。

依赖包

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>javax.servlet-api</artifactId>

<version>4.0.1</version>

<scope>provided</scope>

</dependency>

ServletContextListener 接口定义了两个方法：

contextInitialized(ServletContextEvent event): 当 Servlet 上下文被初始化时调用该方法。这通常发生在 Web 应用程序启动时，用于执行一些初始化操作，如加载配置文件、建立数据库连接等。

contextDestroyed(ServletContextEvent event): 当 Servlet 上下文被销毁时调用该方法。这通常发生在 Web 应用程序关闭时，用于执行一些清理操作，如关闭数据库连接、释放资源等。

通过实现 ServletContextListener 接口并在 web.xml 配置文件中进行相应的注册，可以在 Servlet 上下文的生命周期事件发生时执行自定义的逻辑。例如，你可以在 contextInitialized 方法中初始化应用程序的全局变量，或者在 contextDestroyed 方法中释放资源和保存状态。

下面是一个示例：

javaCopy codeimport javax.servlet.ServletContextEvent;  
import javax.servlet.ServletContextListener;  
@Configuration  
public class MyServletContextListener implements ServletContextListener {  
​  
   @Override  
   public void contextInitialized(ServletContextEvent event) {  
       // 在 Servlet 上下文初始化时执行的逻辑  
       System.out.println("ServletContext initialized.");  
       // 执行其他初始化操作...  
   }  
​  
   @Override  
   public void contextDestroyed(ServletContextEvent event) {  
       // 在 Servlet 上下文销毁时执行的逻辑  
       System.out.println("ServletContext destroyed.");  
       // 执行其他清理操作...  
   }  
}

在 web.xml 文件中配置监听器：

xmlCopy code<listener>  
   <listener-class>com.example.MyServletContextListener</listener-class>  
</listener>

当你的 Web 应用程序启动或关闭时，上述监听器的方法将被调用，以执行你定义的相应逻辑。

# spring监听器的实现ApplicationListener<ContextRefreshedEvent>

在 Spring 框架中，除了 Servlet 监听器之外，还有一些特定于 Spring 的监听器可以用于处理 Spring 容器的生命周期事件和应用程序的事件。以下是几个常用的 Spring 监听器：

ApplicationListener：ApplicationListener 接口是 Spring 框架中用于监听应用程序事件的主要接口。你可以实现该接口来处理各种应用程序级别的事件，如上下文刷新事件、上下文关闭事件、自定义事件等。通过在 Spring 上下文中注册实现了 ApplicationListener 接口的 bean，你可以在特定事件发生时执行相应的逻辑。

ContextRefreshedEvent：ContextRefreshedEvent 事件在 Spring 上下文初始化或刷新时触发。你可以通过实现 ApplicationListener 接口并监听该事件，在应用程序初始化完成后执行一些初始化任务。

ContextClosedEvent：ContextClosedEvent 事件在 Spring 上下文关闭时触发。你可以通过实现 ApplicationListener 接口并监听该事件，在应用程序关闭前执行一些清理任务。

RequestHandledEvent：RequestHandledEvent 事件在 Spring MVC 应用程序中，每次请求处理完成后触发。你可以通过实现 ApplicationListener 接口并监听该事件，在每次请求完成后执行一些额外的处理，如日志记录、性能监控等。

这些 Spring 监听器提供了一种机制，可以在特定的 Spring 容器事件发生时执行自定义的逻辑。你可以通过实现相应的监听器接口，并使用 Spring 的配置机制将监听器注册到 Spring 上下文中。

以下是一个示例，展示如何使用 Spring 的 ApplicationListener 监听器：

javaCopy codeimport org.springframework.context.ApplicationListener;  
import org.springframework.context.event.ContextRefreshedEvent;  
import org.springframework.stereotype.Component;  
​  
@Component  
public class MyApplicationListener implements ApplicationListener<ContextRefreshedEvent> {  
​  
   @Override  
   public void onApplicationEvent(ContextRefreshedEvent event) {  
       // 在应用程序初始化或刷新时执行的逻辑  
       System.out.println("Application initialized or refreshed.");  
       // 执行其他初始化操作...  
   }  
}

在上述示例中，MyApplicationListener 类实现了 ApplicationListener<ContextRefreshedEvent> 接口，并在 onApplicationEvent 方法中处理 ContextRefreshedEvent 事件。通过将该类标记为 @Component，并在 Spring 配置中启用组件扫描，Spring 将自动将该监听器注册到应用程序的 Spring 上下文中。

通过使用 Spring 的监听器机制，你可以更好地集成和扩展 Spring 应用程序，以处理各种事件和生命周期情况。

# ORM框架是什么

ORM（对象关系映射）框架是一种软件工具或库，它的目标是在关系型数据库和面向对象编程语言之间建立映射，以便于开发人员使用面向对象的方式来操作数据库。

传统的关系型数据库使用表格和行来存储和组织数据，而面向对象编程语言使用类和对象来表示数据和行为。ORM 框架的主要任务是将这两种不同的数据模型之间进行转换和映射，使得开发人员可以使用面向对象的编程语言来操作数据库，而无需直接编写复杂的 SQL 查询语句。

ORM 框架通常提供以下功能：

对象关系映射：ORM 框架能够自动将对象模型和数据库表之间进行映射，将类和对象映射到数据库表和行，字段映射到属性或成员变量。

数据库操作：ORM 框架提供一组API或方法，用于执行数据库操作，如插入、更新、删除和查询数据，而无需直接编写 SQL 语句。

事务管理：ORM 框架通常提供事务管理功能，使得开发人员能够在应用程序中执行事务操作，并确保数据的一致性和完整性。

查询语言：ORM 框架通常提供一种查询语言（如 HQL、JPQL 或类似的语言），用于进行高级和复杂的数据库查询操作。

缓存管理：ORM 框架通常提供缓存机制，以提高性能和减少对数据库的访问次数。

# 获取xml

#### **IMG_293**

# 获取properties文件



# <context:annotation-config/>注解作用

当在 Spring 的配置文件中包含 <context:annotation-config/> 元素时，它会告诉 Spring 容器去扫描并启用各种注解的支持，以便使用注解来配置和管理 Spring 组件。

具体而言，<context:annotation-config/> 的作用如下：

启用注解扫描：该元素会启用 Spring 对组件扫描的支持，以自动检测和注册使用注解标注的 Spring 组件，如 @Component、@Service、@Repository 等。

启用自动装配：该元素会启用 Spring 的自动装配功能，使得通过 @Autowired、@Resource、@Inject 等注解可以自动完成依赖注入。

支持其他注解功能：该元素还启用了其他一些注解的支持，例如 @Value 用于属性注入、@Qualifier 用于指定注入的具体实例等。

# mysql的去重、组合、分组、连接、虚拟表

###### **去重复(DISTINCT)**

　　DISTINCT:用于返回唯一不同的值，主要是用于某一字段

　　语法

SELECT DISTINCT <列名>|\* FROM <表名>

###### **组合查询**

　　SQL组合查询可以使用以下关键字进行操作：

UNION(并集):比较两个查询的结果，返回两个集合所有非重复行。

INTERSECT(交集):比较两个查询的结果，返回由左右双侧输入查询输出的非重复行。

EXCEPT(差集):比较两个查询的结果，返回左侧查询集合中不包含左右集合交集部分的非重复行。

注意: 使用 EXCEPT 或 INTERSECT 的两个查询的结果集组合起来的基本规则: 所有查询中的列数和列的顺序必须相同；数据类型必须兼容。

　语法

[SQL查询表达式1]  
 UNION | INTERSECT | EXCEPT  
[SQL查询表达式2];