**Java面试题大全**

**一.JavaSE 部分**

**基础部分**

**Java中基本数据类型有哪些？**

byte：8位，最大存储数据量是255，存放的数据范围是-128~127之间。

short：16位，

int：32位，最大数据存储容量是2的32次方减1，数据范围是负的2的31次方到正的2的31次方减1。

long：64位，最大数据存储容量是2的64次方减1，数据范围为负的2的63次方到正的2的63次方减1。

float：32位，

double：64位，

boolean：只有true和false两个取值。

char：16位，存储Unicode码，用单引号赋值。

**Integer 和 int的区别**

**int**是基本数据类型，变量中直接存放数值，变量初始**化时值是0**

**Integer**是引用数据类型，变量中存放的是该对象的引用，变量**初始化时值时null**

Integer是int类型的包装类，将int封装成Integer，符合java面向对象的特性，可以使用各种方法比如和其他数据类型间的转换

Integer和int的深入对比：

1. 两个通过new生成的Integer对象，由于在堆中地址不同，所以永远不相等
2. int和Integer比较时，只要数值相等，结果就相等，因为包装类和基本数据类型比较时，会自动拆箱，将Integer转化为int
3. 通过new生成的Integer对象和非通过new生成的Integer对象相比较时，由于前者存放在堆中，后者存放在Java常量池中，所以永远不相等
4. 两个非通过new生成的Integer对象比较时，如果两个变量的数值相等且在-128到127之间，结果就相等。这是因为给Integer对象赋一个int值，java在编译时，会自动调用静态方法valueOf()，根据java api中对Integer类型的valueOf的定义，对于-128到127之间的整数，会进行缓存，如果下次再赋相同的值会直接从缓存中取，即享元模式

**String和StringBuilder和StringBuffer区别**

**三者**底层都是char[]存储数据，JDK1.9之后使用的是byte[] ，因为往往我们存储都是短字符串，使用byte[]这样更节约空间。

由于String底层的char[]有final修饰，因此每次对String的操作都会在内存中开辟空间，生成新的对象，所以String不可变

StringBuilder和StringBuffer是可变字符串，没有final修饰，适合字符串拼接，另外StringBuffer是线程安全的，方法有synchronized修饰，但是性能较低，StringBuilder是线程不安全的，方法没有synchronized修饰，性能较高

**String a = "A" 和 String a = new String("A") 创建字符串的区别**

String a = "A" 首先去常量池找 “A”，如果有，会把a指向这个对象的地址 ，如果没有则在栈中创建一个char型的值'A',堆中创建一个String对象object，值为"A",接着object会被存放进字符串常量池中，最后将a指向这个对象的的地址

new String("A") ： 如果常量池中么有“A”就会走上面相同的流程先创建“A”，然后在堆中创建一个String对象，它的值共享栈中已有的char值“A”。

**下面代码创建了几个对象**

* String s = "a" +"b" + "c" + "d";这条语句创建了几个对象？

创建了一个对象，因为相对于字符串常量相加的表达式，编译器会在编译期间进行优化，直接将其编译成常量相加的结果。

* String s； 创建几个对象？ 没有创建对象。
* String a = "abc"; String b = "abc"; 创建了几个对象

创建了一个对象，只是在第一条语句中创建了一个对象，a和b都指向相同的对象"abc"，引用不是对象

**== 和 equals 的区别是什么**

==比较对象比较的是地址，对于Object对象中的equals 方法使用的也是 == ，比较的是对象的地址，默认情况下使用对象的equals比较Object中的equals方法，也就是比较地址，如果要实现自己的比较方式需要复写equals 方法。

对于包装类比如：Integer都是复写过equals方法，比较的是int 值。

**final 和 finally 和 finalize 的区别**

当用final修饰类的时，表明该类不能被其他类所继承。当我们需要让一个类永远不被继承，此时就可以用final修饰

finally作为异常处理的一部分，它只能用在try/catch语句中，并且附带一个语句块，表示这段语句最终一定会被执行（不管有没有抛出异常），经常被用在需要释放资源的情况下

finalize()是在java.lang.Object里定义的，也就是说每一个对象都有这么个方法。这个方法在gc启动，该对象被回收的时候被调用。其实gc可以回收大部分的对象（凡是new出来的对象，gc都能搞定，一般情况下我们又不会用new以外的方式去创建对象），所以一般是不需要程序员去实现finalize的。

**JDK 和 JRE 有什么区别？**

JRE(Java Runtime Enviroment) ：是Java的运行环境，JRE是运行Java程序所必须环境的集合，包含JVM标准实现及 Java核心类库

JDK(Java Development Kit) ：是Java开发工具包，它提供了Java的开发环境(提供了编译器javac等工具，用于将java文件编译为class文件)和运行环境(提 供了JVM和Runtime辅助包，用于解析class文件使其得到运行)。JDK是整个Java的核心，包括了Java运行环境(JRE)，一堆Java工具tools.jar和Java标准类库 (rt.jar)。

**面向对象四大特性**

抽象 : 是将一类对象的共同特征总结出来构造类的过程,包括数据抽象和行为抽象两方面,抽象只关注对象的哪些属性和行为,并不关注这此行为的细节是什么 - 举例：定义一个persion类，了就是对人这种事物的抽象

封装：对数据的访问只能通过已定义的接口,封装就是隐藏一切可隐藏的东西,只向外界提供最简单的编程接口，比如在Java中，把不需要暴露的内容和实现细节隐藏起来，或者private修饰，然后提供专门的访问方法，如JavaBean。 - 生活举例：电脑主机就是把主板等封装到机壳，提供USB接口，网卡接口，电源接口等。 JavaBean就是一种封装。

继承：新类(子类，派生类)继承了原始类的特性，子类可以从它的父类哪里继承方法和实例变量，并且类可以修改或增加新的方法使之更适合特殊的需要。

多态：多态是指允许不同类的对象对同一消息做出响应。对象的多种形态，当编译时类型和运行时类型不一样，就是多态，意义在于屏蔽子类差异

**方法覆盖和重载**

方法的覆盖是子类和父类之间的关系，方法的重载是同一个类中方法之间的关系。 覆盖只能由一个方法，或只能由一对方法产生关系；方法的重载是多个方法之间的关系。 覆盖要求参数列表相同；重载要求参数列表不同。

**普通类和抽象类**

抽象类不能被实例化, 需要通过子类实例化 抽象类可以有构造函数，被继承时子类必须继承父类一个构造方法，抽象方法不能被声明为静态。 抽象方法只需申明，而无需实现，抽象类中可以允许普通方法有主体 含有抽象方法的类必须申明为抽象类 抽象的子类必须实现抽象类中所有抽象方法，否则这个子类也是抽象类

**接口和抽象类**

定义接口使用interface，定义抽象类使用abstract class

接口有全局常量，抽象方法，（java8后：静态方法，默认方法）

抽象类有构造方法，抽象方法，普通方法

接口和类是实现关系，抽象类和类是继承关系

**IO流**

**你知道BIO,NIO,AIO么？讲一下你的理解**

BIO （Blocking I/O）：同步阻塞I/O 模式，**以流的方式处理数据**，数据的读取写入必须阻塞在一个线程内等待其完成。**适用于连接数目比较小且固定的架构**

NIO （New I/O）：同时支持阻塞与非阻塞模式，**以块的方式处理数据**，**适用于连接数目多且连接比较短（轻操作）的架构，比如聊天器**

AIO （ Asynchronous I/O）：异步非阻塞I/O 模型，**适用于连接数目多且连接比较长（重操作）的架构**

**java 中四大基础流**

InputStream : 输入字节流, 也就是说它既属于输入流, 也属于字节流 ，

OutputStream: 输出字节流, 既属于输出流, 也属于字节流

Reader: 输入字符流, 既属于输入流, 又属于字符流

Writer: 输出字符流, 既属于输出流, 又属于字符流

**读文本用什么流，读图片用什么流**

文本用字符输入流，读图片用字节输入流

**字符流和字节流有什么区别**

字符流适用于读文本，字节流适用于读图片，视频，文件等。

字节流操作的基本单元为字节；字符流操作的基本单元为Unicode码元。

字节流默认不使用缓冲区；字符流使用缓冲区。

字节流通常用于处理二进制数据，实际上它可以处理任意类型的数据，但它不支持直接写入或读取Unicode码元；字符流通常处理文本数据，它支持写入及读取Unicode码元

**BufferedInputStream 用到什么设计模式**

主要运用了俩个设计模式，适配器和装饰者模式

**带缓冲区的流**

BufferedInputStream 带缓冲区的字节输入

BufferedOutputStream 带缓冲区的输出流

BufferedReader : 带缓冲区的字符输入流

BufferedWriter : 带缓冲区的字符输出流

**集合篇**

**说一下Java中的集合体系**

Collection接口

List：

* ArrayList：底层数据结构是数组，查询性能高，增删性能低
* Vector：底层数据结构是数组，查询性能高，增删性能低
* LinkedList：底层数据结构是双向链表，查询性能低，增删性能高

Set：

* HashSet：无序不重复的，使用HashMap的key存储元素，判断重复依据是hashCode()和equals()
* TreeSet：有序不重复的，底层使用TreeMap的key存储元素，排序方式分为自然排序，比较器排序

Map接口

* HashMap：key的值没有顺序，线程不安
* TreeMap：key的值可以自然排序，线程不安全
* HashTable：它的key和value都不允许为null，线程安全
* Properties：它的key和value都是String类型的，线程安全

**HashMap和HashTable的区别**

HashMap和HashTable都是实现了Map接口的集合框架，他们的区别

* HashTable是线程安全的，它的实现方法都加了synchronized关键字，因此它的性能较低
* HashMap是线程不安全的，它实现方法没有加synchronized，因此它的性能较高
* HashMap的key和value都允许为null，HashTable中的key和value都不能为null,如果不考虑线程安全，建议使用HashMap，如果需要考虑线程安全的高并发实现，建议使用ConcurrentHashMap

**ArrayList和LinkedList区别**

都属于线性结构，ArrayList是基于数组实现的，**开辟的内存空间要求联系**，可以根据索引随机访问元素性能高，但是插入和删除元素性能差，因为这会涉及到移位操作

LinkedList是基于双链表实现的，**开配的内存空间不要求连续，因此不支持索引，查找元素需要从头查找，因此性能差，但是添加删除只需要改变指针指向即可，性能高**. LinkedList会增加内存碎片化，增加内存管理难度

根据实际需要，如果项目中使用查找较多，使用ArrayList，如果使用增删较多，请使用LinkedList

**ArrayList和Vector区别**

ArrayList是线程不安全的，Vector相反是线程安全的，方法加了同步锁，线程安全但是性能差，ArrayList底层数组容量不足时，会自动扩容0.5倍，Vector会自动扩容1倍

**一个User的List集合，如何实现根据年龄排序**

第一种方式，让User类实现Comparable接口，覆写compareTo方法，方法中自定义根据年龄比较的算法

第二种方式，调用Collections.sort方法，传入一个比较器，覆写compare方法，方法中自定义根据年龄比较的算法

**HashMap底层用到了那些数据结构？**

JDK1.7及其之前：数组，链表 ; JDK1.8开始：数组，链表，红黑树

**什么是Hash冲突**

哈希冲突，也叫哈希碰撞，指的是两个不同的值，计算出了相同的hash，也就是两个不同的数据计算出同一个下标，通常解决方案有：

* 拉链法，把哈希碰撞的元素指向一个链表
* 开放寻址法，把产生冲突的哈希值作为值，再进行哈希运算，直到不冲突
* 再散列法，就是换一种哈希算法重来一次
* 建立公共溢出区，把哈希表分为基本表和溢出表，将产生哈希冲突的元素移到溢出表

**HashMap为什么要用到链表结构**

当我们向HashMap中添加元素时，会先根据key进行哈希运算，把hash值模与数组长度得到一个下标，然后将该元素添加进去。但是如果产生了哈希碰撞，也就是不同的key计算出了相同的hash值，这就出问题了，因此它采用了拉链法来解决这个问题，将产生hash碰撞的元素，挂载到链表中

**HashMap为什么要用到红黑树**

当HashMap中同一个索引位置出现哈希碰撞的元素多了，链表会变得越来越长，查询效率会变得越来越慢。因此在JDK1.8之后，当链表长度超过8个，会将链表转坏为红黑树来提高查询

**HashMap链表和红黑树在什么情况下转换的？**

当链表的长度大于等于8，同时数组的长度大于64，链表会自动转化为红黑树,当树中的节点数小于等于6，红黑树会自动转化为链表

**HashMap在什么情况下扩容？HashMap如何扩容的？**

HashMap的数组初始容量是16，负载因子是0.75，也就是说当数组中的元素个数大于12个，会成倍扩容

tips:为啥子是0.75：负载因子过小容易浪费空间，过大容易造成更多的哈希碰撞，产生更多的链表和树，因此折衷考虑采用了0.75

为啥子是成倍扩容：需要保证数组的长度是2的整数次幂

为嘛数组的长度必须是2的整数次幂：我们在存储元素到数组中的时候，是通过hash值模与数组的长度，计算出下标的。但是由于计算机的运算效率，加减法>乘法>除法>取模，取模的效率是最低的。开发者们为了让你用的开心，也是呕心沥血。将取模运算转化成了与运算，即数组长度减1的值和hash值的与运算，以此来优化性能。但是这个转化有一个前提，就是数组的长度必须为2的整数次幂

**HashMap是如何Put一个元素的**

首先，将key进行hash运算，将这个hash值与上当前数组长度减1的值，计算出索引。此时判断该索引位置是否已经有元素了，如果没有，就直接放到这个位置

如果这个位置已经有元素了，也就是产生了哈希碰撞，那么判断旧元素的key和新元素的key的hash值是否相同，并且将他们进行equals比较，如果相同证明是同一个key，就覆盖旧数据，并将旧数据返回，如果不相同的话

再判断当前桶是链表还是红黑树，如果是红黑树，就按红黑树的方式，写入该数据，

如果是链表，就依次遍历并比较当前节点的key和新元素的key是否相同，如果相同就覆盖，如果不同就接着往下找，直到找到空节点并把数据封装成新节点挂到链表尾部。然后需要判断，当前链表的长度是否大于转化红黑树的阈值，如果大于就转化红黑树，最后判断数组长度是否需要扩容。

**HashMap是如何Get一个元素的**

首先将key进行哈希运算，计算出数组中的索引位置，判断该索引位置是否有元素，如果没有，就返回null,如果有值，判断该数据的key是否为查询的key，如果是就返回当前值的value

如果第一个元素的key不匹配，判断是红黑树还是链表,如果是红黑树，就就按照红黑树的查询方式查找元素并返回,如果是链表，就遍历并匹配key，让后返回value值

**你知道HahsMap死循环问题吗**

HashMap在扩容数组的时候，会将旧数据迁徙到新数组中，这个操作会将原来链表中的数据颠倒，比如a->b->null，转换成b->a->null

这个过程单线程是没有问题的，但是在多线程环境，就可能会出现a->b->a->b....，这就是死循环

在JDK1.8后，做了改进保证了转换后链表顺序一致，死循环问题得到了解决。但还是会出现高并发时数据丢失的问题，因此在多线程情况下还是建议使用ConcurrentHashMap来保证线程安全问题

**说一下你对ConcurrentHashMap的理解**

ConcurrentHashMap，它是HashMap的线程安全，支持高并发的版本

在jdk1.7中，它是通过分段锁的方式来实现线程安全的。意思是将哈希表分成许多片段Segment，而Segment本质是一个可重入的互斥锁，所以叫做分段锁。

在jdk1.8中，它是采用了CAS操作和synchronized来实现的，而且每个Node节点的value和next都用了volatile关键字修饰，保证了可见性

**二.JavaEE&框架&中间件**

**数据库基础**

**平局值用什么，分组用什么**

统计平局值：avg , 分组：group by

**两个相同列的结果集求并集用什么**

union 并集 , union all(允许重复并集)

**完整查询SQL中的关键字的定义顺序**

SELECT 列名 FROM 表1 JOIN 表2 ON 条件 WHERE 条件 GROUP BY 列名 HAVING 条件 ORDER BY 列名 LIMIT

**完整的多表JOIN查询，SQL中关键字的执行顺序**

FROM --> ON --> JOIN --> WHERE --> GROUP BY --> HAVING --> ORDER BY --> LIMIT

**员工表employee字段有: id, username, amount ,deptname .**

* 求每个部门总人数怎么做 ,

select 部门名,count(id) from employee group by deptname

* 求每个部门总工资怎么做?

select 部门名,sum(amount) from employee group by deptname

**Spring部分**

**介绍一下Spring**

Spring是一个开源的轻量级控制反转和面向切面编程的容器框架。轻量级是说它开发使用简单，功能强大。控制反转是指将对象的创建，销毁控制交给ioc容器，方便解耦合，降低维护难度，面向切面编程是指将相同的逻辑横向抽取出来，可以对一些通用业务如事务，日志进行集中管理

**说下Spring框架的组成**

1. CoreContain核心容器模块：
   1. spring-core：提供框架的基本组成部分，包括 IoC 和依赖注入功能
   2. spring-beans：提供 BeanFactory，工厂模式
   3. context：提供国际化，事件传播，资源加载等功能
   4. spring-ExpressionLanguage：提供表达式语言
2. Web模块
   1. Web：提供面向web的基本功能和面向web的应用上下文
   2. Web-MVC：为web应用提供模型视图控制（MVC）
   3. Web-Socket：在 web 应用程序中提供客户端和服务器端之间通信的方式
   4. **Web-Portlet：模块提供了用于Portlet环境的MVC实现**
3. 数据/集成模块
   1. JDBC：包含了Spring对JDBC数据访问进行封装的所有类
   2. ORM：为对象-关系映射提供交互层
   3. OXM：提供对Object/XML映射实现的抽象层
   4. JMS：主要包含了一些制造、消费和消息的功能
   5. **Transaction：为实现特殊接口类以及所有的 POJO 支持编程式和声明式的事务管理**
4. 其他模块
   1. AOP：提供了面向切面编程相关实现
   2. Aspects：模块提供了与AspectJ的集成，是一个功能强大的AOP框架
   3. I**nstrumentation：提供了class instrumentation 的支持和类加载器classloader的实现**
   4. **Messaging：为 STOMP 提供支持**
   5. Test：支持使用JUnit和TestNG对Spring组件进行测试

**什么是Spirng的IOC**

IOC控制反转，把对象的创建，属性设置，初始化，销毁等工作交给Spirng的IOC容器去管理，解放程序员的劳动力。

对象被注册到Spring的IOC容器中，使用的时候从容器中获取即可，非常方便。

它通过依赖注入，将需要的外部资源注入到组件中，使用IOC使得对象之间的耦合度降低，资源变得容易管理，从而使得代码更加优雅

**你对AOP的理解**

AOP，Aspect Oriented Programming 英文首字母缩写，意为面向切面编程，是Spring的核心思想之一

AOP是对OOP(面向对象编程)的一种补充，能够做到很多面向对象无法做到的事情，比如需要在所有方法执行前开启事务，打印日志，如果使用面向对象来编程，将会产生大量重复代码，而使用AOP，**可以将那些与业务无关，却为业务模块所共同调用的逻辑封装起来，一次解决这些问题**。而这些重复的代码，一般统称为横切逻辑代码

使用AOP，在不改变原有业务逻辑的情况下，**实现解耦合，避免横切逻辑代码重复**

AOP的使用场景包括日志记录，性能统计，安全控制，事务处理，异常处理等等

**它是基于动态代理实现的，分为JDK动态代理和CGLIB动态代理。JDK动态代理只支持实现了接口的类 ，CGLIB支持没有实现接口的类。Spring默认使用JDK动态代理，如果被代理类没有实现接口，会选择CGLIB动态代理**

**Spring的Bean懒加载和非懒加载有什么区别**

懒加载：需要使用对象的时候才创建，节省资源，但不利于提前发现错误

非懒加载，也叫迫切加载，容器启动时就创建对象，消耗资源，但有利于提前发现错误

spring中默认时迫切加载，即在项目启动时，spring会扫描符合条件的所有bean并将其初始化

如果需要懒加载，可以使用@Lazy注释或者xml中配置属性default-lazy-init="true"

**Spring的依赖注入方式有哪些**

方式一：**setter方式注入，通过反射调用无参构造方法生成对象，再通过对于的setter方法注入配置的值，支持注解和xml两种实现方式**

方式二：**构造器方式注入，通过反射调用有参构造方法生成对象，支持注解和xml两种实现方式**

注解实现方式：**@Autowired，它是默认按类型匹配的、@Resource，它是默认按名字匹配的**

**说一下定义切面相关的注解**

@Aspect：定义切面

@Pointcut：定义切点 = cn.xx.service.\*

@Before：前置通知，在目标方法运行之前运行

@After：后置通知，在目标方法运行结束之后运行（无论方法正常结束还是异常结束）

@AfterReturning：返回通知，在目标方法正常返回之后运行

@AfterThrowing：异常通知，在目标方法出现异常以后运行

@Around：动态代理，手动推进目标方法运行

**Bean的四种注册方式**

方式一：普通注册方式，直接通过class注册

方式二：简单静态工厂方式注册

方式三：简单实例工厂方式注册

方式四：FactoryBean方式注册

**注册Bean的注解有哪些**

@Controller/@RestController 一般用于定义控制层的类

@Service 一般用于定义服务层的类

@Repository 一般用于定义持久层类

@Component 定义一般类

@Configuration 定义配置类

**IOC的启动流程有了解过吗**

当Spring启动时，IOC容器会加载Spring的配置文件，包括XML配置或者注解，然后解析这些Bean并把相关定义信息封装成BeanDefinition对象，通过Bean注册器BeanDefinitionRegistry注册到IOC容器，也就是一个ConcurrentHashMap中

此时会找出所有的单例且非惰性加载的bean，根据其BeanDefinition进行Bean的实例化，它会判断如果bean中有方法覆盖，就使用JDK反射创建Bean，否则使用CGLIB方式生成代理。然后把实例化好的Bean缓存到一个ConcurrentHashMap中

**Bean的生命周期讲一下**

从宏观的角度来说就是：**实例化 ，属性注入，初始化，使用，销毁。**更细的生命周期如下

1. 实例化：**如果是单例且迫切加载的bean，在Spring容器启动时就会根据BeanDefinition进行实例化，如果时设置了懒加载或者多例模式的bean，在用的时候才会实例化**
2. 属性赋值：**通过BeanDeifinition找到当前Bean所依赖的其他Bean，如果容器中有就直接拿过来，如果没有就根据创建流程区创建依赖的bean，然后通过反射给依赖的字段注入值**
3. **然后会调用BeanPostProcessor的前置处理器，对于@Autowired和@Transcational就是基于BeanPostProcessor来实现的。**
4. 接着会看Bean是否实现InitializingBean ，如果有会触发其afterPropertiesSet方法的调用
5. 接着是调用我们自定义的bean的init-method方法，此时会调用执行
6. 然后是调用BeanPostProcessor的后置处理
7. 容器正常关闭，Bean进行销毁，会先调用实现了DisposableBean的destory方法。
8. 接着调用我们指定的bean的destroy-method方法，此时会调用执行

**单例多例的区别**

单例和多例属于对象模式，单例模式指对象在整个系统中只存在一份，多例模式则可以有多个实例。

在spring的ioc容器中的bean默认都是单例的，如果需要使用多例，可以通过修改scope属性：scope="prototype"

如果一个bean是单例模式的，在处理多次请求的时候，在ioc容器中只实例化一个bean，这个对象会被保存在一个map中，当有请求来的时候，会先从map中查看，如果有就直接使用这个对象，没有才会实例化新的对象。

如果是多例(prototype)模式的bean，每次请求来的时候，会直接实例化新的bean，没有map缓存的过程。

**Spring的Bean被指定为prototype以及singleton有什么区别**

这两者分别指的是多例和单例模式，singleton即单例模式，指对象在整个系统中只存在一份；prototype即多例模式系统中可以有多个实例。

如果一个bean是单例模式的，在处理多次请求的时候，在ioc容器中只实例化一个bean，这个对象会被保存在一个map中，当有请求来的时候，会先从map中查看，如果有就直接使用这个对象，没有才会实例化新的对象。

如果是多例模式的bean，每次请求来的时候，会直接实例化新的bean，没有map缓存的过程。

在spring的ioc容器中的bean默认都是单例的，如果需要使用多例，可以指定scope属性：scope="prototype"

**BeanFactory和ApplicationContext有什么区别**

**BeanFactory接口是IOC容器的核心接口，定义了管理bean的最基本方法，比如实例化，配置，管理，获取bean的方法**

**ApplicationContext接口是BeanFactory接口的子接口，除了继承BeanFactory中所有管理bean的方法，还拥有环境、国际化、资源、事件等服务相关的接口**

**BeanFactory是延迟加载，ApplicationContext是迫切加载**

**BeanFactory和FactoryBean的区别**

BeanFactory接口是IOC容器的核心接口，定义了管理bean的最基本方法，比如实例化，配置，管理，获取bean的方法

**FactoryBean是IOC容器创建bean的一种形式，可以通过实现此接口来创建实例化过程比较复杂的bean**

**IOC容器是如何保证Bean的单例的？**

IOC容器会将单例模式的bean放入一个ConcurrentHashMap中，需要这个bean时直接到这个map中获取，如果没有找到才会实例化这个bean。而ConcurrentHashMap本身时线程安全的，也就保证了Bean是单例的

**Spring如何解决Bean的循环依赖**

**循环依赖分为三种，构造器注入循环依赖 ，setter方式注入循环依赖，多例模式Bean的循环依赖。而Spring解决了单例bean的setter注入循环依赖**

**setter循环依赖的解决主要使用了三级缓存**

* 一级缓存，**用来缓存已经实例化好的bean，即单利Bean缓存池**
* 二级缓存，**用来缓存正在创建的bean**
* 三级缓存，**用来缓存创建bean的实例工厂ObjectFactory**

假设有两个bean，A依赖B，B依赖A

**当实例化好A，在属性注入环境，发现A依赖了B，会先将正在创建的A的实例工厂ObjectFactory放入三级缓存，然后去创建B的实例。**

**走Bean的实例化流程创建B，在B的属注入环节发现，B依赖了A，这个时候就会去三级缓存中，找到A的创建工厂ObjectFactory获取A的实例，并注入到B中。此时B就初始化好了，然后将B实例放入一级缓存。最后将B实例注入到A中，A也就创建好了**

**在getBean的时候，如果单利Bean缓存池没有Bean,就会走二级缓存尝试获取，如果也没有，就会走三级缓存拿到Bean的ObjectFacory创建Bean,然后把Bean放入二级缓存。**

**Spring构造器注入能循环依赖吗**

构造注入不能解决循环依赖的原因是：**如果A的构造其中依赖了B B的构造器中又依赖了A 在getSingleton中三级缓存需要调用getObject()构造器，来构造提早暴露但未设置属性的bean，此时就会产生无限递归创建**

多例模式下Bean是不做缓存的，所以就没法暴露ObjectFactory，也就没办法解决循环依赖

**说几个Spring的IOC的容器工厂类**

BeanFactory：IOC容器顶层接口，提供了Bean获取的基础方法

DefaultListableBeanFactory：是整个 bean 加载的核心部分，Spring 注册及加载Bean 的默认实现

ApplicationContext：除了实现IOC基本功能外，还扩展了国际化支持，资源访问，事件发布

ClasspathXmlApplicationContext：从classpath中获取XML配置

**你知道Spring的AOP主要基于什么设计模式实现吗**

AOP的实现原理是基于动态代理，动态代理就是在运行时期动态的为原生类生成代理类以达到代码增强的目的，且代理类是持有原生类的，可以在代理类中调用原生类以及做一些增强业务。

动态代理分为JDK动态代理和CGLIB代理，CGLIB代理需要导入相关的jar包，两者的区别是JDK动态代理要求目标类需要实现至少一个接口。而CGLIB则是基于继承进行代理，原生类可以不实现任何接口

Spring中默认采用JDK动态代理，如果原生类没有实现任何接口，Spring会选择CGLIB代理，或者你可以在配置文件中强制指定使用CGLIB代理

**你知道@Autowaire自动注入的实现原理吗？**

**自动注入是通过BeanPostProcessor 后置处理器**AutowiredAnnotationBeanPostProcessor完成的，**在Bean实例化过程中，触发了AutowiredAnnotationBeanPostProcessor的postProcessPropertyValues方法的调用执行，它就会扫描当前类中是否有@Autowired注解**，**然后得到自动注入依赖的bean的类型，并去容器中得到依赖的bean实例，如果没有就走Bean的实例化流程创建依赖的Ban，然后反射进行字段赋值。**

**你知道@Transcational注解的实现原理吗？**

分为两个动作把，第一个是解析@Transcational注解，在Sping中有个后置处理器InfrastructureAdvisorAutoProxyCreator，在Bean的初始化过程中，它负责解析标记了@Transcational注解的类，生成代理。还创建了 TransactionAttributeSource ，它是对事务注解的封装，以及 TransactionInterceptor 事务拦截器。

在执行业务方法的时候，代码会进入事务拦截器TransactionInterceptor去执行事务相关的代码，TransactionInterceptor主要是通过调用TranscationManagerment的事务API，而TranscationManagerment又是调用connection的事务API完成事务操作。

**Javaweb基础**

**常见Http状态码**

200 成功返回状态

301 永久重定向，被请求的资源永久移动到新位置

302 临时重定向，被请求的资源临时移动到新的位置，项目中使用了oauth2，对目标资源访问无权限时就会见到，它是会重定向到授权地址

401 无权限访问

403 禁止访问，服务器已经接收到请求，但拒绝执行

404 找不到该资源

500 服务器内部错误 zuul找不到服务名就会见到

503 服务器内部错误 服务器维护或者过载

504 网关超时

**Servlet的生命周期**

Servlet 生命周期可被定义为从创建直到毁灭的整个过程。以下是 Servlet 遵循的过程：

* Servlet 初始化后调用 **init ()** 方法。
* Servlet 调用 **service()** 方法来处理客户端的请求。
* Servlet 销毁前调用 **destroy()** 方法。
* 最后，Servlet 是由 JVM 的垃圾回收器进行垃圾回收的。

**什么是过滤器？怎么创建一个过滤器**

过滤器：在请求发送之后，处理之前对请求的一次拦截，可以更改请求状态或者参数值等。

创建过滤器：实现filter接口，重写doFilter方法，最后在web.xml中配置过滤器

**讲一下Session的工作原理**

服务端的session id会自动写入客户端的cookie中，每次请求客户端回自动把cookie带入后台，后台自动根据cookie中的sessionid就能找到session

**Session和cookie有什么区别**

session和cookie都是为了弥补http协议的无状态特性，解决会话问题

session是以ConcurrentHashMap结构存储在服务器端，同时生成一个sessionid返回客户端并存放到cookie中

cookie是将数据存储在客户浏览器端

session占用服务器的性能，但安全性较高，使用cookie减轻服务器的压力，但有被用户篡改风险因此安全性较低

**说说preparedStatement和Statement的区别**

statement的sql语句使用字符串拼接，很容易出错，而preparedStatement使用？作为占位符，不容易出错易于维护

statement不对sql语句作处理，直接交给数据库，而preparedStatement支持预编译，事先将编译好的sql语句放到数据库端，相当于缓存，因此效率更高

statement有sql注入风险，preparedStatement没有sql注入风险

**请求转发和重定向的区别**

转发是一次请求，可以共享同一组request和response，重定向是多次请求，不能共享同一组request和response

转发地址栏不会发生变化，重定向地址栏会发生变化

转发不能到外部应用，重定向可以到尾部应用

如果我们需要数据共享，使用转发，如果需要访问内部资源(WEB-INF)，使用转发，如果需要跨域到外部资源，必须使用重定向

**get和post请求的区别**

最直观的区别，get把参数包含在url中，post是把参数放到request body中

post相对于get更安全,post发送的数据更大，get有url的长度限制

post更发送更多的数据类型，get只能发送ASCII字符

在restful中，get一般用户查询搜索数据，post一般用户添加或者修改数据

**JSP的原理**

jsp的本质就是servlet,每个JSP文件都回被编译成一个Serverlet去执行，在该Serverlet会对JSP中的动态内容进行替换，静态部分是标准的html，动态部分是java程序

**SpringMVC部分**

**SpringMVC怎么样设定重定向和转发的**

重定向是指将用户从当前请求重新定向到一个视图页面，或者是一个handler处理请求，以前的request域中信息全部失效，同时地址栏会发生变化，它是客户端行为

转发是指将用户从当前请求转发给另一个视图页面或者handler处理请求，以前的request域可以共享，地址栏不会发生变化，它是服务器行为

springmvc默认是使用转发方式跳转的，且会默认经过视图解析器,我们也可以通过指定，转发时在返回值前面加"forward:"，重定向时在返回值前面加"redirect:"，且此时就不会再经过视图解析器了

**SpringMVC如何对时间格式的参数进行格式化**

第一种需求，后台接收前台页面返回的string类型时间，要转换成的Date类型数据，可以使用@DateTimeFormat注解来接收参数

第二种需求，后台将Date类型数据返回给前台页面，默认是返回时间戳，如果想要优雅的格式，可以在模型的Date字段或get方法上使用@JsonFormat注解

**SpringMVC常用的注解有哪些**

@Controller：用来标识一个类是控制器类

@RequestMapping：用来映射请求路径和参数

@ResponseBody：将返回值放到responsebody中，通常返回json或者xml格式数据

@RequestBody：将前台请求参数转换成对象

@PathVariable：接收路径参数，通常用在restful接口中

@RestController：@Controller和@ResponseBody的组合注解

@ControllerAdvice：运用aop的思想，对全局做一些处理，比如结合@ExceptionHandler做全局异常捕获

**如何定义SpringMVC的拦截器**

SpringMVC 的拦截器主要用于拦截用户的请求并做相应的处理，通常应用在权限验证、判断登录等功能上

第1步，定义拦截器：可以实现 HandlerInterceptor 接口来自定义拦截器，接口定义了三个方法，preHandler方法是在请求到达处理器之前执行，postHandler方法是在请求经过处理器之后、解析试图之前执行，afterCompletion方法是在视图渲染之后、返回客户端之前执行

第2步，配置拦截器：在springmvc的配置文件xml中，配置所有拦截路径，以及需要放行的路径

**HandlerInterceptor和HandlerInterceptorAdapter的区别**

HandlerInterceptor是接口，我们可以实现该接口来定义拦截器，HandlerInterceptorAdapter是抽象类，它实现了HandlerInterceptor接口的子接口AsyncHandlerInterceptor，我们可以继承该类来定义拦截器，它简化拦截器的实现，默认preHandler返回true

**SpringMVC的执行原理**

**1.Http请求：客户端请求提交到DispatcherServlet-前端控制器**

**2.寻找处理器：由DispatcherServlet调用HandlerMapping-处理器映射器，根据url找到对应的的Handler**

**3.调用处理器：DispatcherServlet指定HandlerAdapter-处理器适配器去调用Handler**

**4.调用业务处理和返回结果：Handler调用业务逻辑处理完成后，返回ModelAndView**

**5.处理视图映射并返回模型： DispatcherServlet查询一个或多个ViewResoler-视图解析器，找到ModelAndView指定的视图**

**6.Http响应：将结果显示到客户端**

**SpringMVC的Controller是单例还是多例，有没有并发安全问题，如何解决**

在spring中，bean默认都是单例的，controller也是交给spring容器管理的一个bean，因此它也是单例的。

单例的好处是减少了创建对象和垃圾回收的时间，节省了内存资源，但同时单例会造成线程不安全的问题，因为当所有请求访问同一个controller实例，controller中的成员变量是所有线程公用的，某个线程如果修改了这个变量，别的请求再来拿这个变量就编程修改后的值了

要解决这个问题，最直接有效的方式就是不要在controller中定义成员变量，如果你非要定义成员变量，两种方式

第一种，可以给controller上加注解@Scope("prototype")，将controller设置为多例模式，每次请求都重新实例化一个controller

第二种，使用ThreadLocal变量，让每一个线程都有自己独立的变量

**RequestMapping 和 GetMapping有什么区别**

@Getmapping是一个组合注解，即是@RequestMapping(method = RequestMethod.GET)的缩写，意思是只接收get请求的方法

@Requestmapping如果没有指定请求方式，可以接收get，put等各种类型的请求

**SpringBoot部分**

**相比Spring,Spring Boot有哪些优点**

Springboot是一个基于spring的框架，对spring做了大量简化，使开发流程更快，更高效

它大量简化maven依赖，管理了大量的基础依赖

基于注解配置（JavaConfig），无需xml配置

内嵌Tomcat，部署流程简单

打包和部署更加灵活，允许独立运行

**SpringBoot如何做全局异常处理**

可以使用@ControllerAdvice注解，编写一个全局异常处理类，再自定义一个方法使用@ExceptionHandler来捕获具体的异常并作相应的处理

通常情况下后台向前台返回结果时，会把结果封装成包含有错误码，错误信息以及数据本身的json数据，因此我们可以使用自定义异常类，自定义枚举错误码，在捕获全局异常后，向前台返回一个包含错误码的信息

**@SpringBootApplication注解的含义**

@SpringBootApplication是SprnigBoot项目的核心注解，目的是开启自动配置，并表示该类为主启动类。它包含三个子标签

* @ComponentScan注解：**开启ioc自动扫描注解，默认扫描当前包及其子包中@Controller,@Service等，并把这些bean加载到ioc器中**
* @EnableAutoConfiguration注解：**启用springboot自动配置，自动所有扫描classpath目录下面所有jar中的spring.factories文件实现配置类批量注册**
* @SpringBootConfiguration注解：**标志该类为springboot配置类**

**spring-boot-starter-parent的作用**

这是SpringBoot的父工程，它的作用是帮我们管理了很多的基础jar包，同时它继承了spring-boot-dependencies，在spring-boot-dependencies项目中通过<dependencyManagement>管理了大量的依赖，同时通过<properties>维护了这些依赖的版本号

但是在项目中，还需要通过<dependencies> 去导入具体的依赖才能使用

**spring-boot-starter-web的作用**

此项目是Springboot和Springmvc整个的jar包，构建了web项目的基本环境，集成了日志，tomcat，springmvc，json支持等等

**SpringBoot中如何读取配置**

方式一：使用@Value读取配置文件

方式二：使用@ConfigurationProperties读取配置文件

**SpringBoot中日志的level有哪些**

日志级别从低到高分别为：

TRACE < DEBUG <INFO <WARN < ERROR

如果设置为 WARN，则低于 WARN 的信息都不会输出

Spring中默认使用INFO级别输出到控制台

**SpringBoot中如何管理事务**

事务(transaction)是指业务逻辑上对数据库进行的一系列持久化操作，要么全部成功，要么全部失败。

在Springboot中，可以通过xml配置和注解配置

**xml方式通过配置DataSourceTransactionManager和transactionManager实现**

**注解方式配置通过在主启动类上加上@EnableTransactionManagement开启事务管理器，在具体的实现层service类上加上@Transactional 实现事务**

**SpringBoot自动配置原理**

在启动类上我们会打上： @SpringBootApplication 注解，它是一个组合标签，包括：

* SpringBootConfuration ,本质是一个 Configuration ，代表Spring的配置类。
* IOC自动扫描的注解 ，ComponentScan 会去扫描类上是否有：@Component ,@Respository ,@Service @Controller ,如果有，就会把这个类自动注册到Spring容器中。
* EnableAutoConfiguration ：就是启动SpringBoot自动配置的注解

在 @EnableAutoConfiguration 注解中，注册了一个选择器，其中有一个方法会去返回很多的自动配置的的全限定名，这些类会自动注册到Spring容器中，

那它是怎么去找到这些所谓的自动配置类的呢？

**他会通过Spring的SPI接口，也就是通过一个SpringFactoryLoader去扫描 classpath中的所有的jar包中的 MET-INF/spring.factories 中的自动配置类**，比如： DispatchServlert就对应了DispatchServlertAutoConfiguration自动配置类 ， 它通过@Bean+方法的方式注册了一个 DispatchServlert 到Spring容器中了

**SpringBoot启动流程**

1.开启秒表计时

2.starting监听器，

3.处理应用参数

4.加载环境对象

5.打印横幅

6.创建Spring容器对象：AnnotationConfigApplicationContext

7.容器刷新的前置工作

8.刷新容器 ，这里会执行spring的ioc属性容器的refrsh方法,Bean的加载，初始化等都在这个里面，Tomcat的启动也在这个方法里面。

9.刷新容器后置工作

10.秒表停止

11.started事件

12.调用runner

13.running.listeners.

**Mybatis部分**

**MyBatis中${}取值和#{}取值的区别**

{}能够防止SQL注入，因为底层使用PreparedStatement对象，预编译，性能较高

${}不能防止SQL注入，因为底层使用Statement对象，不会预编译而是拼接字符串，性能较低

能使用#{}时尽量使用#{}，如果需要动态传入表名或者字段名需要用 ${}比如，像 ORDER BY 时只能使用${}

**MyBatis关联查询中，延迟加载和饥饿加载的区别**

延迟加载，是先从单表查询，需要使用关联数据的时候才发起关联查询，不用的时候不查询关联的数据，又叫懒加载,饥饿加载，是在查询时将关联的数据立即查询出来加载进内存，不管用不用

**MyBatis对象关联查询和集合关联查询怎么做**

单个关联对象用associate ，适用于多对一的关联查询，使用javaType来定义实体类型，集合用collection，适用于一对多的关联查询，使用ofType来定义集合的泛型类型

**MyBatis一级缓存和二级缓存的区别**

缓存，是指将从数据库查询出的数据存放在缓存中，下次使用相同查询时不必再从数据库查询，而是直接从缓存中读取，从而减轻数据库查询的压力，提高性能

mybaits中的一级缓存，是SqlSession级别，默认开启，使用同一个SqlSession发送相同的SQL时命中；它的生命周期和SqlSession一致，当调用SqlSession.close()方法时会释放缓存

mybatis中的二级缓存，是namespace级别，默认不开启，执行同一个namespace的相同statement，发送相同的SQL时命中；它的生命周期是程序结束

当SQL中执行了update()、delete()、insert()操作，则缓存中的数据都会清空

**MyBaits的Mapper接口没有实现类为社么可以用@Autowired直接注入**

动态代理**，赋值给mapper接口引用的对象其实是一个代理对象**，**这个代理对象是由 JDK 动态代理创建的。在解析mapper的时候，mybatis会通过java反射，获取到接口所有的方法**

当调用接口中方法时，将通过接口全限定名+方法名对应找到映射文件中namespace和id匹配的sql，然后将执行结果返回

**在MyBatis如何动态修改SQL**

使用Mybatis的拦截器可以做到

**MyBatis的动态SQL标签有哪些？**

if标签：条件判断

choose、when、otherwise标签：选择结构，类似java中的switch

trim标签：对包含的内容加上前缀，后缀

where标签：主要是用来简化SQL语句中where条件判断的，能智能的处理and or，不必担心多余导致语法错误

foreach标签：遍历元素

**Mybatis的mapper如何传递多个参数**

方式一，可以使用map进行传参，SQL中使用map的key来引用取值

方式二，可以在SQL中使用#{param1}，#{param2}...来引用取值，它是根据mapper接口对应方法中形参的顺序进行匹配的，不管接口方法的参数名字叫个啥，SQL都只能使用param1，param2，等来取值

方式三，可以使用@Param注解，给mapper接口方法的参数命名，在SQL中直接使用取的名字来引用

**Mybatis，关联对象查询，使用嵌套子查询和JOIN连表有什么区别**

嵌套子查询，指的是在查询一个主对象的时候，使用单表查询，在resultmap中额外发送一个子sql查询关联对象，然后映射给主对象

连表join查询，指的是查询一个主对象的时候，使用join连表的方式把主对象和关联对象的数据一次性查出来，用resultmap映射结果

他们的区别，join连表查询只发一条sql就能把数据查询出来，嵌套子查询会有一个n+1的问题，就是说如果主查询出来n条数据，那么会额外发送n条子sql去查询对应的关联对象，加上主查询那1次，也就是n+1次，因此它的性能相对较低的，一般我们会使用join连表查询

**为什么要使用连接池**

对数据库的操作都需要取得连接，使用完都需要关闭连接，如果每次操作需要打开关闭连接，这样系统性能很低下。连接池就可以动态的管理这些连接的申请，使用和释放，我们操作数据库只需要在连接池里获取连接，使用完放回连接池，这样大大节省了内存，提高效率。

数据库连接池的原理主要分为三部分

* 第一，连接池的建立，在系统初始化时建立几个连接对象以便使用。
* 第二，连接池的管理，客户请求连接数据库时，首先查看连接池中是否有空闲连接，如果有直接分配，如果没有就等待，直到超出最大等待时间，抛出异常
* 第三，连接池的关闭，当系统关闭时，连接池中所有连接关闭

**Redis部分**

**讲一下你理解的Redis，为什么Redis很快**

Redis是一种高性能的，开源的，C语言编写的非关系型数据库，可以对关系型数据库起到补充作用，同时支持持久化，可以将数据同步保存到磁盘

说Redis很快是相对于关系型数据库如mysql来说的，主要有以下因素

* 第一，数据结构简单，所以速度快
* 第二，直接在内存中读写数据，所以速度快
* 第三，采用多路IO复用模型，减少网络IO的时间消耗，避免大量的无用操作，所以速度快
* 第四，单线程避免了线程切换和上下文切换产生的消耗，所以速度快

**你常用的Redis的数据存储结构有哪些，他们的使用场景分别是什么**

Redis存储形式是键值对，支持value形式包括String，List，Set，ZSet，Hash。

String可以用作缓存，计数器，防攻击，验证码、登录过期等，List可以用来做队列，秒杀等，Set可以用来去重

**Redis每种存储结构说 4 个命令吧**

1.String

* set key value 设置值
* get key 取值
* mset key value key value... 设置多个值
* mget key key 获取多个值
* incr key 将key中的值自增1
* decre key 将key中的值自减1

2.List

* lpush key value value... 从最左边设置值
* rpush key value value... 从最右边设置值
* lrange key start stop 查询key中指定区间的元素
* lpop key 移出并返回key中最左边的元素
* rpop key 移出并返回key中最右边的元素

3.Set

* sadd key value value 添加元素
* smembers key 返回集合key中的所有元素
* srem key member 删除集合key中member元素
* scard key 查询集合key中的元素数量

4.ZSet

* zadd key score value (score value)... 添加元素
* zcard key 查询集合key中元素数量
* zcount key min max 返回有序集合key中score 在min和max之间的元素
* zrange key start stop 返回有序集合key中索引在start和stop之间的元素

5.Hash

* hset key field value 添加元素
* hget key field 获取key集合中field键对应的值
* hmset key field value (field value)... 添加元素并批量添加子键值对
* hmget key field field 获取key集合中所有的子键值对

**你们项目是怎么用Redis的**

使用的是Springboot整合的redis，主要用来解决前后端分离后前后端会话问题，以及验证码的问题

**怎么防止Redis宕机数据丢失问题**

通过对Redis持久化，把内存中的数据和命令，保存一份到磁盘中做备份，当Redis发生宕机，重启服务器的时候，会从磁盘重新加载备份的数据，从而解决数据丢失问题

**Redis持久化是什么？有几种方式**

将内存中的数据备份到磁盘的过程，就叫作持久化

Redis持久化主要有两种方式，RDB和AOF，可以通过修改redis.conf进行配置

RDB是记录数据快照，而AOF是记录写命令的

**Redis有了AOF持久化为什么还要RDB？**

AOF和RDB各有所长

* **RDB是记录数据快照，它的优点是只产生一个持久化文件，体积相对较小，启动恢复速度快，备份方便，它的缺点是没办法做到数据百分百不丢失，因为它是每隔一定时间保存一次**
* **AOF是记录写命令，它的优点是格式清晰，容易理解，数据更安全，采用append模式即使持久化过程中宕机，也不影响已经保存的数据，它的缺点是文件体积较大，恢复速度慢**

根据实际需要来选择，通常二者可以结合来使用

**Redis内存不够了怎么办？**

方式一：增加物理内存

方式二：使用淘汰策略，删掉一些老旧数据

方式三：集群

**你们Redis用在哪些业务上？用的什么存储结构**

主要用做缓存,比如：验证码，分类缓存，数据字典缓存，权限数据缓存，登录信息缓存等。

String类型的存储结构用的比较多，并且使用了Json格式进行序列化。

**淘汰策略有哪些？你们用的哪种**

* volatile-lru ：从已设置过期时间的数据集中挑选最近最少使用的数据淘汰
* volatile-ttl：从已设置过期时间的数据集中挑选将要过期的数据淘汰
* volatile-random：从已设置过期时间的数据集中任意选择数据淘汰
* allkeys-lru：从数据集中挑选最近最少使用的数据淘汰
* allkeys-random：从数据集中任意选择数据淘汰
* no-enviction：不使用淘汰

**Redis事务和Mysql事务的区别**

Mysql的事务是基于日志，记录修改数据前后的状态来实现的，而Redis的事务是基于队列实现的

Mysql中的事务满足原子性：即一组操作要么同时成功，要么同时失败，

Redis中的事务不满足原子性，即一组操作中某些命令执行失败了，其他操作不会回滚

因此对于比较重要的数据，应该存放在mysql中

**使用Redis如何实现消息广播**

Redis是使用发布订阅来实现广播的

订阅者通过 SUBSCRIBE channel命令订阅某个频道 ， 发布者通过 PUBLISH channel message向该频道发布消息，该频道的所有订阅者都可以收到消息

**为什么要使用Redis做缓存**

一个字，快。

缓存它指的是将数据库的数据同步到内存中，客户端获取数据直接从内存中获取。由于内存读写速度大于磁盘，而使用缓存能减少磁盘读取，大大提高查询性能。

我们一般会将经常查询的，不会经常改变的热点数据，保存到缓存中，提高响应速度

**缓存的执行流程**

1.客户端发起查询请求

2.判断缓存中是否有数据

* 如果有，直接返回
* 如果没有，就从数据库查询，再把数据同步到缓存

3.返回数据给客户端

**你们怎么保证Redis和Mysql的一致性**

我们在代码中控制，如果数据库做是写操作，直接把redis中的对应数据删除，下次查询数据会重新写入缓存。

我们的业务对一致性要求不是很高，因此采用了先操作mysql，后删除redis。在写数据库和删除缓存行代码之间如果有查询请求依然会查询到Redis中的老数据，但是这种情况非常极端，而且我们的业务也能容忍这种短暂的脏数据。

我还知道其他方案，比如延迟双删 ， 监听Mysql事务日志自动同步Redis等。

**SpringCache常用注解**

@EnableCaching：打在主启动类上，开启缓存功能

@Cacheable：打在方法上，表示该方法会开启缓存，打在类上，表示类中所有的方法都开启缓存，方法的返回值会自动写入缓存。如果缓存中已经有数据，方法将不会被调用，而是拿着缓存数据直接返回给客户端。

@CacheEvict：搭载类或者方法上，会将缓存清除

@CachePut：更新缓存

@Caching：组合操作，要应用于方法的多个缓存操作

@CacheConfig：打在类上，共享的一些常见缓存设置

**了解缓存击穿，穿透，雪崩吗？怎么处理？**

缓存击穿：缓存中没有，数据库中有的数据，由于某种原因比如缓存过期了，同时并发用户特别多，一时间都往数据库中读取数据

* 解决方案：**加互斥锁，只能允许一个线程访问数据库，然后其他线程就可以往内存中拿**

缓存穿透：客户端频繁请求一个缓存和数据库中都没有数据，导致数据库压力大。

* 解决方案：**布隆过滤器来判断数据库中有没有这个key**

缓存雪崩：缓存重启，或者大量key失效，导致大量并发打到数据库

* 解决方案：**为key设置不同的过期时间**

**Redis的主从有什么优点，和缺点？**

优点是读写分离，分担了读的压力，同时能起到备份作用，防止数据丢失

缺点是不能分担写的压力，主的单点故障没有解决，存储没有得到扩容

**解释一下Redis的哨兵模式。哨兵的不足？**

**当主服务器中断服务后，可以将一个从服务器升级为主服务器 ，以便继续提供服务**

**哨兵就是用来监控主从服务器，实现故障恢复功能的。**它会不断的检查主服务器和从服务器的健康状态，当某个服务器出现问题时，可以向管理员发起通知。如果主服务器不可用时，会自动选择一个从服务器作为新的主服务器，并让其他的从服务器从新的主服务器复制数据

**哨兵也是主从模式，没有解决写的压力，只减轻了读的压力，而且存储也得不到扩容**

**Redis的cluster集群怎么存储数据的？**

Redis Cluster集群采用哈希槽 (hash slot)的方式来分配的。它默认分配了16384个槽位，当我们set一个key 时，会用CRC16算法得到所属的槽位，然后将这个key 分到对应区间的节点上

**什么情况下Redis集群不可用？**

Redis Cluster有一个容错机制，如果半数以上的主节点与故障节点通信都超时了，就会认为该节点故障了，自动触发故障转移操作，故障节点对应的从节点升级为主节点。

**但是如果某个主节点挂了，又没有从节点可以使用，那么整个Redis集群就不可用了、**

**Redis存储结构底层有没有了解？什么是SDS**

简单动态字符串，是Redis自己封装的字符串结构。它记录了字节数组buf，字节数组中用到的字节数len，以及未使用的字节数free。

* 为了解决二进制安全问题，定义了len来表示已有字符串长度
* 为了防止缓冲区溢出，在分配内存的时候做了预留空间free
* 内存惰性释放，多余的内存加入free做预留，优化了内存频繁分配
* 针对不同的String长度定制了不同的SDS结构

**Redis如何模拟队列和栈，用什么命令**

**list控制同一边进，同一边出就是栈；list控制一边进，另一边出就是队列**

**Redis存储单个对象怎么存，存储对象集合怎么存**

单个对象可以使用String，也可以使用hash

集合对象可以使用hash，以便可以快速的通过field来取值

**你们Redis用来做什么？使用的什么结构？**

登录信息login，使用的是String结构存储

手机验证码code，使用的是String结构

课程分类course\_type ，使用的是String结构

购物车保存，使用的是Hash结构

**统计全国高考前20名用什么？**

Zrevrangebyscore

**从100个VIP用户中随机抽取5名怎么做？**

Srandmember

**RabbitMQ**

**RabbitMQ的使用场景**

rabbitMQ消息队列可以用来

* 做任务的异步处理，提高程序的相应时间
* 提高系统稳定性，通过手动确认机制，当消费者出现故障，只要没有确认签收，请求的数据都不会丢失可以再次处理
* 服务解耦，生产者通过MQ与消费者交互
* 消除峰值，通过异步处理，消息传到MQ直接返回，接着等待排队处理，避免了线路拥堵

**RabbitMQ如何防止消息丢失**

首先，RabbitMQ的消息确认机制，默认是自动签收，也就是说消息一旦被消费者接收，就自动签收，消息就从队列里清除了。**因此对于重要的消息，不容丢失的数据，我们需要设置在消费完成后手动签收**

**其次，我们可以将消息持久化，避免消息在消费前MQ宕机，网络问题等造成的消息丢失**

**RabbitMQ的交换机有哪几种**

Fanout：广播，将消息交给所有绑定到交换机的队列

Direct：定向，把消息交给符合指定routing key的队列

Topic：通配符，把消息交给符合routing pattern的队列

**消息是如何从发送者到达消费者的（RabbitMQ工作流程）**

分为消息发送和消息接收两个步骤

* 消息发送：**生产者和Broker建立TCP连接，创建信道。通过信道将消息发送给Broker，由Exchange将消息进行转发到指定的队列**
* 消息接收：**消费者和Broker建立TCP连接 ，创建信道 ，然后监听指定的队列，当有消息到达队列时，Broker默认将消息推送给消费者，消费者就能接收到消息**

**如何防止消息重复消费**

重复消费，一般时由于消费者消费成功后，在给MQ确认签收的时候出现了网络波动，MQ没有接到确认，就会继续给消费者投递之前的消息，造成消费者接收到了两条一样的消息。

我们可以通过实现消息的幂等性来避免这种情况，比如说**让生产者给每个消息携带一个唯一的id，消费者获取消息后根据这个id去查询数据库，如果不存在就正常消费，如果存在了就证明该消息被消费过，直接丢弃**

**RabbitMQ消息投递失败，你们怎么处理**

我们可以**设置confirm回调和 returned 回调**

比如说，可以在发送消息的时候，把消息详情包括交换机名，路由键，都保存到一个表中，状态设置为发送中，如果在confirm方法中ack为false,代表发送到交换机失败 ，就把这个记录状态修改为发送失败

**然后我们创建一个定时任务定时扫表，去读取发送失败的数据并重新发送，为了优化性能，我们设置重试次数3次，如果3次都失败了，我们可以采取人工干预**

**ElasticSearch**

**Lucene创建索引原理**

**Lucene是基于倒排索引原理来实现的**

* 首先，将原文档进行分词处理，形成一个个单独的单词，
* 然后取出标点符号以及停词，形成词元，
* 再将词元做一些语言相关的处理，比如变成小写，转换时态，单复数形式等等，
* 将得到的词创建一个字典，**按照字母顺序排序，合并相同的词，最终生成一个倒排索引文档**

**ES的keyword和text区别**

keyword：不分词，直接建立索引，支持模糊查询，精确查询，聚合查询

text：分词后建立索引，支持模糊查询，精确查询，不支持聚合查询

keyword通常用于通常用于存储年龄，性别，邮编，邮箱号码等等，**直接将完整数据保存的场景**

text通常存储全文搜索的数据，**例如地址，文章内容的保存**

**ES的优势**

**ES是基于Lucene的开源搜索引擎，它解决了原生Lucene使用的不足，优化了Lucene的调用方式**

* 分布式的实时文件存储，每个字段都被索引并可被搜索
* 支持实时分析搜索
* 可以扩展到上百台服务器，处理PB级结构化或非结构化数据
* 通过简单的 RESTful API、可以跟各种语言的客户端甚至命令行进行交互
* 上手非常容易，只需很少的学习就可以在生产环境中使用

**Lucene/ES为什么那么快(ES用到什么数据结构)**

传统搜索比如mysql的like关键字查询，它的搜索方式就是全文扫表，查询性能很低

ES是基于Lucene的全文检索引擎，它采用的是倒排索引结构，在存储时先对文档进行分词，再做一些标点符号去除，大小写时态转换等优化处理，最后按照字母顺序去重排序，形成一个倒排索引文档，我们在检索时，就可以通过二分查找的方式找到目标值

**ES的分层结构，index下面是什么**

Index：**索引库**，包含有一堆相似结构的文档数据，类比Mysql中的数据库

Type：**类型**，它是index中的一个逻辑数据分类，类比Mysql中的表

Document：**文档**：是ES中的最小数据单元，通常用json结构标识，**类比Mysql中的一行数据**

Field：**字段**：**类比Mysql中的一个列**

**从ES7.0开始，Type被干掉了，从此库表合一即一个Index中只有一个默认的Type**

**讲几个ES中的查询对象：比如TermQuery**

TermQuery：匹配关键字查询（关键词不分词）

MatchQuery：匹配关键字查询（关键字分词后）

BooleanQuery：按条件查询

matchAllQuery：匹配所有文档查询

rangeQuery：查询指定范围内的数据

**你简单描述一下DSL语法**

DSL是一种以json形式标识的，由ES提供的一种查询语言，它由两部分组成，DSL查询和DSL过滤。

**DSL过滤类似于模糊查询，DSL查询类似于精确查询**

**你说一下 match和term的区别？**

term：不会对搜索词进行分词处理，而是作为一个整体与目标字段进行匹配，若完全匹配，则可查询到

match：会将搜索词分词，再与目标查询字段进行匹配，若分词中的任意一个词与目标字段匹配上，则可查询到

**你使用过ES的哪些聚合查询？**

**指标聚合，比如求和，求最大值，最小值，平均数**

**数量统计聚合，计算满足条件数据的总条数，相当于sql中的count**

**去重聚合，它会计算非重复的数据个数，相当于sql中的distinct**

**桶聚合**，它会将某个field的每个唯一值当成一个桶，并计算每个桶内的文档个数，**相当于sql中的group by**

**最高权值聚合**，它会匹配每组前n条数据，**相当于sql中的group by后取出前n条**

**ES高亮怎么做的？**

**使用HighlightBuilder对关键字作高亮处理，由于我们项目使用的是SpringBoot整合ES的jar包，结果没有进行高亮处理，我们使用ElasticsearchTemplate的queryForPage方法来获取结果，再手动进行分页封装返回前台**

**你们ES和数据库的数据一致性怎么做的**

代码控制的，数据库做了写操作，直接更新ES中的数据，**我知道可以通过 Logstash 中数据和ES的数据自动同步。**

**ES分片机制了解吗**

ES的索引库由多个分片 shard组成，shard分为primary shard主shad和replica shard 副本,主shard承担写请求，replica副本的数据从primary复制而来，同时分担读请求，primary shard的数量设定了就不能修改，replica数量可以修改。

**描述一下ES添加文档的过程**

(1) 客户端请求一个协调节点coordinating node

(2) 协调节点根据算法选择一个primary shard: 算法 hash(document\_id) % (num\_of\_primary\_shards)

(3) 对应的primary shard 所在节点保存完数据后，将数据同步到replica node。

(4) 协调节点coordinating node 发现 primary node 和所有 replica node 都搞定之后返回结果给客户端

**数据节点存储数据详细流程：**

(1) 当分片所在的节点接收到来自协调节点的请求后，会将请求写入到Memory Buffer，然后定时（默认是每隔1秒）写入到Filesystem Cache，这个从Momery Buffer到Filesystem 　　Cache的过程就叫做refresh

(2) 当然在某些情况下，存在Momery Buffer和Filesystem Cache的数据可能会丢失，ES是通过translog的机制来保证数据的可靠性的。其实现机制是接收到请求后，同时也会写入到translog中，当Filesystem cache中的数据写入到磁盘中时，才会清除掉，这个过程叫做flush；

(3)在flush过程中，内存中的缓冲将被清除，内容被写入一个新段，段的fsync将创建一个新的提交点，并将内容刷新到磁盘，旧的translog将被删除并开始一个新的translog。 flush触发的时机是定时触发（默认30分钟）或者translog变得太大（默认为512M）时；

**详细描述一下Elasticsearch获取文档的过程**

(1) 客户端请求一个协调节点coordinating node

(2) coordinate node 根据算法hash(document\_id) % (num\_of\_primary\_shards)，将请求转发到对应的 node，此时会使用 round-robin随机轮询算法，在 primary shard 以及其所有 replica 中随机选择一个，让读请求负载均衡

(3) 接收到请求的 node 返回 document 给调节点 coordinate node。

(4) coordinate node 返回 document 给客户端。

搜索被执行成一个两阶段过程，我们称之为 Query Then Fetch；

**详细描述一下Elasticsearch搜索过程**

(1) 在初始查询阶段时，查询会广播到索引中每一个分片拷贝（主分片或者副本分片）。

(2) 每个分片在本地执行搜索并构建一个匹配文档的大小为 from + size 的优先队列。PS：在搜索的时候是会查询Filesystem Cache的，但是有部分数据还在Memory Buffer，所以搜索是近实时的。

(3) 每个分片返回各自优先队列中所有文档的 ID 和排序值给协调节点，协调节点它合并这些值到自己的优先队列中来产生一个全局排序后的结果列表。

(4) 接下来就是 取回阶段，协调节点辨别出哪些文档需要被取回并向相关的分片提交多个 GET 请求。每个分片加载并 丰富 文档，如果有需要的话，接着返回文档给协调节点。一旦所有的文档都被取回了，协调节点返回结果给客户端。

**详细描述一下Elasticsearch更新和删除文档的过程**

删除和更新也都是写操作，但是Elasticsearch中的文档是不可变的，因此不能被删除或者改动以展示其变更；　磁盘上的每个段都有一个相应的.del文件。当删除请求发送后，文档并没有真的被删除，而是在.del文件中被标记为删除。该文档依然能匹配查询，但是会在结果中被过滤掉。当段合并时，在.del文件中被标记为删除的文档将不会被写入新段。

在新的文档被创建时，Elasticsearch会为该文档指定一个版本号，当执行更新时，旧版本的文档在.del文件中被标记为删除，新版本的文档被索引到一个新段。旧版本的文档依然能匹配查询，但是会在结果中被过滤掉。

**ES有几种节点类型？他们的作用分别是什么**

**分为主节点，node.master =true , 数据节点node.data =true , 负载均衡节点(node.data =false,node.master=false),**

node.master=true,代表该节点有成为主资格 ,主节点的主要职责是和集群操作相关的内容，如创建或删除索引，跟踪哪些节点是群集的一部分，并决定哪些分片分配给相关的节点。一般会把主节点和数据节点分开

node.data=true,数据节点主要是存储索引数据的节点，主要对文档进行增删改查操作，聚合操作等,数据节点对CPU,IO,内存要求较高，优化节点的时候需要做状态监控，资源不够时要做节点扩充

当主节点和数据节点配置都设置为false的时候，该节点只能处理路由请求，处理搜索，分发索引操作等，从本质上来说该客户节点表现为智能负载平衡器。配置：mode.master=false,mode.data=false

**ES集群的三种颜色代表什么**

绿色，黄色，红色，绿色代表集群健康，所有的主备分片都得到分配，如果有备分片没有node去分配，集群是黄色，黄色和绿色都是可用状态，如果有主分片的节点down机，集群不可写数据，呈现红色，代表集群不健康。

**你们项目怎么使用ES**

**我们使用的是spring-boot-start-data-elasticsearch这个库来操作ES，用在大数据的搜索场景，比如商品的发布，搜索功能。**

**SpringSeucity**

**说一下security中的的filter**

SecurityContextPersistenceFilter：请求开始会从SecurityContextRepository中获取SecurityContext对象并设置给SecurityContextHolder，在请求完处理成后将SecurityContextHolder持有的SecurityContext再保存到配置好的SecurityContextRepository中，同时清除SecurityContextHolder中的SecurityContext

UsernamePasswordAuthenticationFilter：默认拦截“/login”登录请求，将请求中的认证信息包括用户名，密码封装成UsernamePasswordAuthenticationToken，然后调用AuthenticationManager的认证方法进行认证

BasicAuthenticationFilter：处理 HTTP 请求的 BASIC 授权标头，如果身份验证成功，就把生成的Authentication对象放入SecurityContextHolder。如果设置了记住我，下次访问就不会走这里来了

RememberAuthenticationFilter：记住我，调用RememberMeServices的autoLogin方法自动登录

AnonymousAuthenticationFilter：匿名filter，检测SecurityContextHolder有没有Authentication对象，如果没有，就会创建一个AnonymousAuthenticationToken并保存到SecurityContextHolder

ExceptionTranslationFilter：处理filter链中的所有AccessDeniedException和AuthenticationException

FilterSecurityInterceptor：继承自AbstractSecurityInterceptor，通过调用AccessDecisionManager.decide方法进行授权

**说一下security的认证原理**

首先，请求会经过**UsernamePasswordAuthenticationFilter拦截，请求的用户名密码会封装成UsernamePasswordAuthenticationToken**，**过滤器将token提交给认证管理器AuthenticationManager进行认证**

然后，**认证管理器调用AuthenticationProvider进行认证，AuthenticationProvider再调用UserDetailsService获取到数据库中存储的用户信息UserDetails，然后调用密码编码器对密码进行比较，认证成功后封装Authentication**

再后来，**请求回到UsernamePasswordAuthenticationFilter，调用SecurityContextHolder将Authentication对象封装成SecurityContext并保存到SecurityContextHolder中**

最后，**请求回到SecurityContextPersistenceFilter，它会调用SecurityContextRepository将SecurityContext对象存储起来，再清理掉SecurityContextHolder中的信息**

**三.微服务部分**

**相关概念**

**什么是集群**

**集群使将应用复制成多个相同的应用**，一起来工作，从而提高工作能力。即将多个应用程序分散在不同的服务器，每个服务器都独立运行相同的代码。可以分散服务器压力解决高并发的问题，同时也能预防单节点故障，即一台服务器故障不影响其他服务器正常运行，但没有解决单体应用代码臃肿，业务复杂，维护性差等等问题

**什么是负载均衡**

使用了集群后，解决高并发同时有一个新的问题，就是客户端的请求如何分配到多台服务。**因此需要通过负载均衡器，比如Nginx，使用负载均衡算法比如轮询、权重、随机等等将请求路由到不同的服务器**

**什么是分布式**

**分布式是将应用按照业务类型拆分成多个子应用**，每个子应用部署在不同的服务器上单独运行，子应用之间通过API相互调用。

可以分散服务器压力解决高并发问题，同时可以解决单体应用代码臃肿、业务复杂、维护性差等等问题，但是不能防止单节点故障，比如一个子应用故障，整个应用就能不完整运行

**集群和分布式的区别，分别解决什么问题**

**集群是将一个应用程序复制多份，部署在多台服务器上**，每个服务器中的程序都是完整的，可以独立运行

**分布式是将一个应用程序拆分成多个子程序，分别部署在多台服务器上**，每个服务器中的程序都是不完整的，所有服务器需要相互通信相互协调才能完成最终的业务

**集群能解决高并发问题，同时能防止单节点故障，即一台服务器宕机不影响其他服务器的正常运行**

**分布式也能解决高并发问题，但不能防止单节点故障，即一台服务器宕机了，整体业务就无法完成**

集群无法解决项目本身的代码臃肿、业务复杂等等问题，分布式能降低模块之间的耦合

实际应用中，我们可以将分布式和集群相结合，比如分布式某个子程序的负载很高，可以单独对这个子程序做集群

**说一下你理解的微服务**

**微服务也是一个分布式系统，它将单体应用进行细粒度拆分，形成多个微服务，每个服务独立运行，每个服务也都可以有自己的数据库，服务之间使用HTTP通信，互相协调完成整个系统的业务。**

**它的优点是服务之间解耦合**，不同的服务可以有不同的编程语言，技术选型多元化，支持敏捷开发

**他的缺点是分布式事务很复杂，部署麻烦，技术成本高，服务间通信对性能也有一定的损耗**

**什么是CAP理论 ， 哪些技术用到AP，哪些用到CP**

**CAP理论指的是，在一个分布式系统中，一致性，可用性，分区容错性，三个要素最多只能同时实现两点。**

**分区容错性是分布式系统的内在要求，因此我们通常会在一致性和可用性之间做取舍。**

满足CP，也就是**满足一致性和容错性，舍弃可用性**，如果系统允许有段时间失效就可以考虑。常见的如Redis，Nacos，ZooKeeper

满足AP，也就是**满足可用性和容错性，舍弃一致性**，如果系统允许出现短暂时间的不一致可以考虑。常见的如MySQL，Eureka

**什么是强一致性和最终一致性**

**强一致性是只数据在多个副本中总数实时同步的，如果能容忍数据在多个副本中在一定的延迟时间内同步，则是弱一致性**

**最终一致性则不要求数据什么时候同步，但是最终会同步即可**。通常情况下我们在分布式领域选择会牺牲了强一致性，会采用最终一致性

**什么是Base理论**

**Base指的是基本可用，软状态，最终一致性。它是对CAP中的AP的扩展**，**意思是说当出现故障部分服务不可用时，要保证核心功能可用，允许在一段时间内数据不一致，但最终要保证一致性。满足Base理论的事务也叫柔性事务**

**SpringCloud**

**讲一下你们公司微服务解决方案**

我司正在使用的是第一代微服务方案，Springcloud Netflix全家桶。

它是使用Eureka做服务注册与发现，也就是解决服务之间通信问题，

使用Ribbon/OpenFeign做客户端的负载均衡，也就是解决将请求路由到微服务集群的问题，

使用Hystrix断路器的熔断、降级来解决单节点故障，

使用Zuul做服务网关，将它作为整个微服务的大门，来实现登录、权限检查等业务，

使用Config分布式配置中心，来统一管理配置所有微服务的配置文件，

使用Bus消息总线给各个微服务广播消息，可以实现各个微服务配置的自动刷新，

使用Sleuth链路追踪，来实时监控各个微服务建的调用关系，快速定位故障节点

**说一说Spring Cloud有哪些常用组件**

Eureka：做服务注册与发现，用来解决服务之间通信问题，

Ribbon/OpenFeign：用做客户端的负载均衡，也就是解决将请求路由到微服务集群的问题，

Hystrix：断路器，它的熔断、降级策略用来解决单节点故障，

Zuul：做服务网关，它是整个微服务的大门，可以用来实现登录、权限检查等业务，

Config：分布式配置中心，用来统一管理配置所有微服务的配置文件，

Bus：消息总线，用来给各个微服务广播消息，可以实现各个微服务配置的自动刷新，

Sleuth：链路追踪，用来实时监控各个微服务建的调用关系，快速定位故障节点

**Spring Cloud的优缺点？**

微服务相对单体应用来说

优点

* 服务之间无耦合，代码简单方便开发维护，服务之间升级维护互不影响
* 轻量级HTTP通信机制，不同的服务可以采用不同的编程语言
* 有极强的扩展能力,业务量大的服务可以再次拆分服务，或者也可以集群部署
* 支持时下流行的敏捷开发并做了优化

缺点

* 分布式事务繁琐
* 部署麻烦，开发人员的学习成本高
* 技术成本高，开发人员需要花更多的时间学习相关技术
* 微服务间的通信存在对性能的损耗问题

**什么是服务注册**

Eureka是一个服务组测与发现的组件，翻译成人话就是管理所有微服务的通讯录的组件。**它包含注册中心，客户端两部分组成。客户端在启动的时候会向注册中心发送一条自我介绍信息，比如端口，ip等等，在注册中心就会保存一张所有微服务的通讯录。这就叫服务注册**

**什么是服务发现**

微服务会定期的从客户端拉取一份微服务通讯录，到本地缓存起来，默认是30s一次。当一个微服务向另一个微服务发起调用，直接根据本地的通讯录找到对方的服务名，发送HTTP请求。这个就叫服务发现

**什么是服务续约**

**微服务会定时（默认30s）发送心跳请求，告诉注册中心，自己还处于存活状态，那么服务中心就不会将其从清单中删除，否则，当微服务宕机或者网络故障等因素，没有在规定时间（默认90s）内提交心跳请求，注册中心就会将它从通讯录中删除。**

**如果服务挂了，注册中心要等到90s后剔除，那么在剔除前的这段时间内，挂掉的服务有可能还是会被调用，怎么处理？**

第一，**可以修改注册中心剔除服务时间，同时加快服务续约心跳请求的频率**

第二，**可以使用Hystrix的熔断降级机制，当某个服务不可访问，快速失败，并返回托底数据**

第三。**重试,提供者集群**

**你知道EurekaClient服务发现和服务续约每隔30s做一次请求是用什么技术实现的吗？**

**使用了ScheduledThreadPoolExecutor线程池定时任务来实现**

服务发现是先判断是否开启了服务发现功能（默认是开启的），获取定时任务的间隔时间（默认是30s），然后初始化服务发现的定时任务，间隔时间可以在yml中修改

服务续约是先判断是否开启服务注册功能（默认是开启的），获取定时任务间隔时间（默认是30s），然后初始化心跳请求的定时任务，间隔时间可以在yml中修改

**Ribbon是什么，Ribbon的工作原理讲一下**

**Ribbon是一个客户端负债均衡器，它可以按照负债均衡算法，向多个服务发起调用。当一个微服务有多个集群时，就可以使用它做请求负载均衡，通常结合RestTemplate来使用**

**说一下 Ribbon的工作原理**

**消费者会30/次注册中心拉取服务注册清单缓存到本地**，**当消费者需要调用一组提供者集群服务时**，**Ribbon会根据提供者服务名，在本地缓存的服务地址清单里找到这一组服务的通讯地址，然后按照负债均衡算法（默认是轮询），选择其中的一个通讯地址，发起http调用服务。**

Ribobn**内部通过LoadBalancerInterceptor拦截RestTemplate发起的请求**，然后交给RibbonLoadBalancerClient**负载均衡客户端做负载均衡**，RibbonLoadBalancerClient**把选择服务的工作交给**ILoadBalancer**负载均衡器** ，ILoadBalancer**会调用** IRule**负载均衡算法类来选择服务**。之后RibbonLoadBalancerClient**把选择好的服务交给LoadBalancerRequest去发请求。**

**Ribbon有哪些负载均衡算法，怎么配置**

RoundRobinRule：**简单轮询**，ribbon默认规则

AvailabilityFilteringRule：忽略短路状态和并发过高的服务器

WeightedResponseTimeRule：根据服务器响应时间作为**权重，响应时间越长权重越小**

ZoneAvoidanceRule：根据区域选择

BestAvailableRule：忽略短路的服务器，选择并发较低的服务器

RandomRule：**随机选择一个可用服务器**

Retry：**重试机制的选择逻辑**

**OpenFeign和Ribbon的区别**

**OpenFeign整合了Ribbon和Hystrix,屏蔽了Ribbon拼接URL，参数的细节，使用声明式编程，让服务调用变得更加简单，OpenFiegn底层也是走的Ribbon的负载均衡策略。推荐使用OpenFeign**

**OpengFiegn的工作流程**

首先，当程序启动时，**@EnableFeignClient会扫描@FeignClient注解的接口，并交给Spring容器管理。**

**当发起请求时，会使用jdk动态代理，并为每个方法都生成相应的RequestTemplate，同时封装http信息，包括url和请求参数等，最后把RestTemplate交个HttpClient发送请求，使用ribbon的负载均衡发起调用**

**为什么要使用Eureka 为什么要使用Ribbon 为什么要使用config配置中心**

在微服务系统中，各个服务之间是需要进行网络通信的，那么他们相互调用就得知道对方的通信地址。**eureka就是专门来做做服务注册与发现，解决服务之间通信问题的**

当一个微服务做了集群，也就是同一个服务名会对应多个地址，那么我们在调用的时候，应该调用哪一个就成了问题，Ribbon是一个负债均衡器，它可以按照负债均衡算法，向多个服务发起调用。**当一个微服务有多个集群时，就可以使用它做请求的分发**

在微服务系统中，服务数量很多，而每个服务都有自己的配置文件，管理起来很麻烦。**用了配置中心就可以帮我们集中管理配置文件，它支持本地配置文件，也支持将配置文件放到远程仓库如git集中管理**

**为什么Feign的客户端接口没有写实现类也可以直接被依赖注入**

**自动注入的实例其实是一个jdk动态代理对象，Feign会为每个方法生成相应的requestTemplate**，它根据服务名找到对应的服务，根据返回值类型、形参列表匹配相应的接口，然后封装url、请求参数，最后生成request请求，使用Ribbon负载均衡发起调用

**介绍一下Hystrix**

Hystrix意为熔断器，它可以将出现故障的服务，通过熔断、降级等手段隔离开，这样不影响整个系统的主业务。**它可以防止由单节点异常导致整个微服务故障，如果遇到故障时，快速失败，熔断的同时可以返回兜底数据达到服务降级的目的**

**什么是熔断，什么是降级**

熔断，**是对服务链路的一种保护机制，当链路上的某个服务不可访问时，服务就会触发降级返回拖地数据，同时当失败率到达一个阈值，就标记该服务为短路状态，当请求访问时直接熔断。直到检查到该服务能正常访问时，就快速恢复**

降级，是当某个服务不可访问时，我们返回一些事先准备好的数据给客户端，比如说，友情提示服务暂不可用，请骚后重试，这样用户体验就上去了

**什么是资源隔离？**

**指的是限制某一个分布式服务的资源使用，可以理解为限流，也就是限制某个服务的请求数量。它包括线程池隔离和信号量隔离**

**线程池隔离，是指用一个线程池来存储当前请求，可以通过设置线程池最大线程数和最大排队队列数来限制请求数量**

**信号量隔离：是指用一个计数器来记录当前有多少个线程在运行，请求进来计数器就增加1，超过最大信号量，就直接返回**

**资源隔离：信号量和线程池的区别**

**线程池方式是异步处理，它与调用线程不是同一个线程**

**信号量方式是同步处理，与调用线程是同一个线程**

**线程池方式由于需要排队，调度，线程切换，因此开销较大，信号量方式无需切换线程，开销较小**

**对于CAP理论，Eureka选择的是AP还是CP？它保证了一致性还是可用性？**

CAP理论指的是，一个分布式系统中，一致性，可用性，分区容错性，三个要素只能同时实现两点。**Eureka选择的是AP，它是弱一致性的，保证了可用性和分区容错性，放弃了数据一致性。**也就是说当多个Eureka之间不可通信时，需要保证服务可用，正常提供服务注册发现功能，但是网络恢复后最终还是会同步的。

**说一下Eureka的自我保护**

**为了防止服务被误删除，Eureka不会立即删除过时的服务数据**。这种机制可能会导致客户端从注册中心获取到已经下线的服务并发起调用而导致错误，因此在开发阶段我们可以关闭自我保护机制。在生产环境中，我们需要打开自我保护，**因为它可以防止因为网络波动，服务没有及时续约而造成的服务误删除问题。**

**你们项目是如何做服务降级的？**

比如在秒杀业务中，需要实时从redis中查询库存，**通过设置hystrix的最大信号量，以此来防止redis雪崩。**当并发过高，请求数超过最大信号量，触发降级，直接向客户端返回兜底数据：”活动太火爆啦，请骚后重试“

**Zuul有哪几类Filter,他们的执行顺序是怎么样的？**

zuul按照执行顺序，分为**pre前置过滤，route路由过滤，post后置过滤，error异常后过滤**

**正常流程是请求先经过前置过滤器，到达路由过滤器进行路由，路由到各种微服务执行请求，返回结果后经过后置过滤，返回用户**

**异常流程，如果再整个过程中出现异常，都会进入error异常过滤器，处理完毕后经过post过滤器返回用户，如果error自己出现异常，最终也会通过post过滤器返回用户，如果post过滤器出现异常，也会跳转到error过滤器，然后直接返回用户**

**在Zuul中做登录检查如何实现？**

**可以通过继承ZuulFilter抽象类，自定义pre类型的过滤器，shouldFilter方法中可以定义需要放行的资源，run方法中检查请求头中的token信息，如果没有token，就响应到客户端未登录的信息，并组织filter继续往后执行**

**在Zuul中如何做限流？**

方式一：**可以通过继承ZuulFilter抽象类自定义pre过滤器，加上限流算法，来实现**

方式二：**可以通过hystrix的资源隔离模式，设置线程池最大连接数或者最大信号量来实现**

方式三：**常用，Ratelimit，使用令牌桶算法。。**。

**配置中心解决什么问题？**

在分布式系统中，服务数量很多，而每个服务都有自己的配置文件，管理起来很麻烦。配置中心是个好东西，**可以帮我们集中管理配置文件，它支持本地配置文件，也支持将配置文件放到远程仓库如git集中管理。**

**EureakServer的搭建流程**

第一步，**导入eureka-server依赖，以及springboot的web环境依赖。**

第二布，主启动类上打注解，**@EnableEurekaServer**，开启eureka服务端功能

第三步，**yml配置文件中，配置注册中心的端口号，主机名，注册中心地址**

**Ribbon的整合流程**

第一步，导入ribbon依赖

第二部，**给RestTemplate的Bean定义方法上，加上注解@LoadBalanced，让这个restTemplate有负载均衡的功能**

第三步，**修改restTemplate调用服务的url，将目标主机名换成目标服务名**

**Feign的整合流程**

第一步，导入openfeign依赖

第二部，主配置类加注解，**@EnableFeignClients，开启feign支持**

第三步，**定义feign客户端接口，并加上注释@FeignClient("目标服务名")，接口中定义方法，该方法与目标服务的对应方法的方法名，返回值类型，形参列表，url路径要一致**

**Hystrix的整合流程**

* 第一步，导入hystrix依赖
* 第二部，主启动类加注解，**@EnableCircuitBreaker，开启熔断功能**
* 第三步，在需要开启熔断功能的方法上，加注解@HystrixCommand(fallbackMethod="xxx")，xxx是降级方法
* 第四步，定义降级方法，方法名需要和fallbackMethod的值一致，形参列表和返回值类型需要和目标方法一致

feign整合Hystrix：

* 第一步，**yml中配置，feign.hystrix.enable=true，开启hystrix功能**
* 第二部，**@FeignClient标签中，定义fallback或者fallbackFactory，指定降级类**
* 第三步，

如果是fallback，就实现feign接口，并覆写接口中的方法作为降级方法

如果是fallbackFactory，就实现FallbackFactory接口，同时指定泛型为feign接口，覆写create方法，返回一个feign接口的匿名内部类，类中写降级方法

**Zuul的整合流程**

第一步，导入zuul依赖

第二步，主启动类上加注解@EnableZuulProxy，开启zuul功能

第三步，yml中配置，统一访问前缀prefix，禁用通过服务名方式访问服务ignoredServices，配置路由routes指定某个服务使用某个路径来访问

**ConfigServer的整合流程**

配置中心服务端配置：

第一步，导入config-server依赖

第二步，主启动类加注解，@EnableConfigServer，开启配置中心

第三步，配置文件中，配置远程仓库地址，仓库账号密码

客户端配置：

第一步，导入config-client依赖

第二步，创建bootstrap.yml配置文件，配置中心地址config.uri，要拉取的配置文件名name，环境名profile

**你们微服务项目的技术栈描述一下**

前端门户系统：HTML + JQuery + CSS

前端管理系统：VUE + ElementUI

后端系统：基于SpringCloud微服务框架（Eureka+OpenFeign+Hystrix+Zuul+Config）

+MyBatisPlus+SpringMVC+Redis+ElasticSearch+RabbitMQ+AlicloudOSS

**浏览器发起一个请求，在你的微服务项目中的怎么去执行的？**

浏览器发起的所有请求首先通过Nginx，通过负载均衡算法，路由给zuul集群，然后通过zuul前置过滤，作登录校验后，它会从配置中心拉取的通讯地址中，根据url匹配到对应的服务，然后使用ribbon发起restful调用。微服务间也可以通过feign相互调用，最终执行完任务，返回浏览器

**说下Ribbon和Feign的区别呢**

Ribbon和Feign都是SpringCloud Netflix中实现负载均衡的组件，不同点在于

**Ribbon是需要我们手动构建http请求**，根据目标服务名通过负载均衡算法直接调用目标服务，

Feign是采用接口的方式，将需要调用的目标服务方法定义成抽象方法，路径，服务名，形参列表，返回值类型需要保持一致。我们只需要调用接口中的方法就可以了。它会自动帮我们生成jdk动态代理，为每个方法生成RequestTemplate并封装url和请求参数，使用负载均衡算法发起调用

Ribbon的实现方式，一般配合RestTemplate发起http请求，我们需要在注册RestTemplate的Bean的方法上加@LoadBalanced，使它具有负载均衡的能力

Feign的实现方式，是在主启动类上加@EnableFeignClients，在客户端接口上加注解@FeignClient

**Spring,SpringBoot和SpringCloud的关系以及区别**

Spring是一个开源的轻量级控制反转和面向切面编程的容器框架。轻量级是说它开发使用简单，功能强大。控制反转是指将对象的创建，销毁控制交给ioc容器，方便解耦合，降低维护难度，面向切面编程是指将相同的逻辑横向抽取出来，可以对一些通用业务如事务，日志进行集中管理。

Springboot是一个基于spring的框架，对spring做了大量简化，使开发流程更快，更高效。比如它大量简化maven依赖，基于注解配置（JavaConfig）无需XML，内嵌Tomcat，部署流程简单，打包和部署更加灵活，允许独立运行

SpringCloud是基于SpringBoot实现的，用于微服务架构中管理和协调服务的，它是一系列框架的有序集合，它为开发者提供了一系列工具，例如服务发现与注册，配置中心，网关，负载均衡，熔断器，链路追踪等等，让微服务架构落地变得更简单

**分布式事务**

**什么是分布式事务，**

分布式事务**，指的是在分布式环境中，一个请求可能涉及到对多个数据库的写操作，要保证多数据库的一致性就需要用到分布式事务**

**分布式事务你知道哪些解决方案？ 这些方案如何选型**

常见的分布式事务解决方案，**2PC，TCC，可靠消息最终一致性，最大努力通知**

2PC，它将整个事务流程分为两个阶段，**P指的是准备阶段，C指的是提交阶段。它是一个阻塞协议，不适用于并发较高，事务生命周期长的分布式事务。**

TCC，**它是基于补偿性事务的AP系统的一种实现，补偿也就是说先按照预定方案执行，如果失败了就走补偿方案。**它可以自己定义数据操作的粒度，但是对应用的侵入性强，可以用在登录送积分，送优惠券等等场景

可靠消息最终一致性，**指的是当事务发起方执行完本地事务后，就发出一条消息通知其他参与方，并且他们一定能接收到消息并处理事务。适合执行周期长，并且实时性要求不高的场景**

最大努力通知，**是在不影响主业务的情况下，尽可能的保证数据的一致性，它适用于一些最终一致性敏感度低的业务，比如支付结果通知**

**什么是2pc**

2PC，是将整个事务流程分为两个阶段，**P指的是准备阶段，C指的是提交阶段。**它常见的标准有XA，JTA，Seata

由DTP模型定义事务管理器TM和资源管理器RM之间通讯的接口规范叫做XA，它规定的交互方式是酱紫的：应用程序AP通过TM提交和回滚事务，TM通过XA接口来通知RM数据库事务的开始，结束，提交，回滚

2PC能保证分布式事务的原子性，但是也有很多缺陷

比如，在**第一阶段，如果参与者迟迟不回复协调者，就会造成事务的阻塞，性能不好**

比如，在**第二阶段，如果事务协调者发出提交事务指令后宕机，一部分参与者收到消息提交了事务，另一部分没有收到消息没有提交事务，这就会导致数据不一致**

再比如，在**第二阶段，如果事务协调者发出提交事务指令后宕机，收到指令的参与者也宕机了，我们就不能确定事务的执行结果，究竟有没有提交**

**Seata相比传统2PC有什么区别，以及优点？**

Seata是由阿里中间件团队发起的开源项目Fescar更名而来，是一个开源的分布式事务框架，它通过对本地关系数据库的分支事务协调，来驱动完成全局事务

Seata的**主要优点是性能好，不会长时间占用链接资源，对业务零入侵**

与传统的2PC的区别主要两方面

在架构层次方面，**传统的2PC方案的RM本质就是数据库自身，而Seata的RM是以jar包形式作为中间件层部署在应用程序上**

在两阶段提交上方面，**传统2PC方案是在第二阶段完成才释放资源，而Seata是在第一阶段就将本地事务提交，提高了效率**

**Seata的TC,TM,RM的含义，以及作用？**

TC：事务协调器，它是独立的中间件，需要独立部署运行，它维护全局事务的运行状态，接收TM指令发起全局事务的提交与回滚，负责与RM通信协调各各分支事务的提交或回滚

TM：事务管理器，TM需要嵌入应用程序中工作，它负责开启一个全局事务，并最终向TC发起全局提交或全局回滚的指令

RM：控制分支事务，负责分支注册、状态汇报，并接收事务协调器TC的指令，驱动分支事务的提交和回滚

**你知道TCC吗，它有什么样的优缺点？**

TCC**是基于补偿型事务的AP系统的一种实现。**补偿指的先按照事先预定的方案去执行，如果失败了就走补偿方案

它的优点是异步执行效率高，它能对分布式事务中的各个资源分别锁定，分别提交与释放

它的缺点是对应用的侵入性强，改动成本高，实现难度大

####

**解释一下Seata的工作原理**

Seata有三个角色：

* **TM任务管理器，负责开启，提交，回滚事务的发起，**
* **TC事务协调器 ，接收TM的指令通知RM提交或者回滚事务**
* **RM资源管理器，控制着分支事务的提交和回滚**

假设有服务A需要调用服务B，且两个服务都需要修改各自的数据库,A服务作为程序入口充当TM和RM，B服务控制着分支事务充当RM。

* A服务的TM向TC申请开启一个全局事务，全局事务创建成功并生成一个全局唯一的XID
* A服务的RM向TC注册分支事务，并将其纳入XID对应全局事务的管辖
* A服务执行分支事务，写undolog日志，向TC上报事务状态
* 当调用B服务时，B服务的RM向TC注册分支事务，该分支事务执行，然后写undolog,向TC上报事务状态
* 服务执行完毕A服务的TM向TC发送commit或者rollback指令
* TC接收到指令，向参与事务的RM发送指令
* 事务参与者RM受到commit指令，删除undolog日志。 如果是rollback指令就根据undolog回滚

**你能简单描述一下你在项目中是如何集成Seata的吗**

事务协调器：安装并启动Seata客户端

主业务端：

* 第一步，导入Seata依赖
* 第二步，yml中配置事务组名，同时需要添加配置文件file.conf，registry.conf，需要注意yml中事务组名与file.comf中的事务组名一致
* 第三步，配置DataSource，需要适用Seata对DataSource进行代理
* 第四步，数据库中添加undo log日志表
* 第五步，业务方法上加注解@GlobalTransactional(rollbackFor = Exception.class)注解

事务参与者：

* 前四步与主业务端相同，第五步不需要了

**没有Seata或者TCC这些事务框架,你可以怎么处理事务?**

不用框架就要自己实现,如果业务要求强一致性这个不太好做,需要协调多个数据库的同时提交和回滚.如果是业务不要求强一致性,我可以参照TCC思想 ,可以考虑自己实现异步写数据库方案,如果失败可以做补偿.当然这个要根据业务特性来,很多大公司都是自己封装事务框架.

**分布式锁**

**你说一下什么是分布式锁**

分布式锁是在分布式/集群环境中解决多线程并发造成的一系列数据安全问题.所用到的锁就是分布式锁，这种锁需要被多个应用共享才可以，通常使用Redis和zookeeper来实现。

**分布式锁有哪些解决方案**

**常用的三种方案**

基于数据库实现：通常基于主键，或者唯一索引来实现分布式锁,但是性能比较差，一般不建议使用

基于Redis ：可以使用setnx来加锁 ，但是需要设置锁的自动删除来防止死锁，所以要结合expire使用.为了保证setnx和expire两个命令的原子性，可以使用set命令组合。

另外释放锁在finallly中调用del删除锁，而删除锁前需要判断该锁是否是当前线程加的锁以免误删除锁，需要通过get获取锁然后进行判断，但是需要保证get判断或和del删除锁的原子性，可以使用LUA脚本实现。

总之自己封装Redis的分布式锁是很麻烦的，我们可以使用Redissoin来实现分布式锁，Redissoin已经封装好了。

基于zookeeper ： 使用临时顺序节点实现，线程进来都去创建临时顺序节点,第一个节点的创建线程获取到锁，后面的节点监听自己的上一个节点的删除事件，如果第一个节点被删除，释放锁第二个节点就成为第一个节点，获取到锁。

在项目中可以使用curator，这个是Apache封装好的基于zookeeper的分布式锁方案。

**Redis如何实现分布式锁，用什么命令**

**可以使用setnx来加锁** ，但是**需要设置锁的自动删除来防止死锁**，所以要结合expire使用.**为了保证setnx和expire两个命令的原子性，可以使用set命令组合。**

**Redis实现分布式锁可能会出现什么问题，如何解决**

添加锁和设置过期时间可以使用set命令进行组合，达到原子性加锁

需要用lua解决删除和判断锁的原子性，否则可能会删除掉别人的锁。

Redis集群环境中，**redis节点挂掉可能会导致加锁失败，可以使用Redisson的红锁来解决。**

**你项目中怎么使用分布式锁的**

自己封装Redis的分布式锁是很麻烦的，我们可以**使用Redissoin来实现分布式锁，**Redissoin已经封装好了

**了解Redission的看门狗原理吗？**

Redisson对分布式锁进行了封装**，对于锁超时问题，它提供了看门狗进行锁时间的续期**，底层使用了定时任务每10s检查一下，如果业务还未执行完成，未释放锁，就进行超时时间续期。

**你在项目中如果使用ZK实现分布式锁的？**

基于zookeeper ： 使用临时顺序节点实现，线程进来都去创建临时顺序节点,第一个节点的创建线程获取到锁，后面的节点监听自己的上一个节点的删除事件，如果第一个节点被删除，释放锁第二个节点就成为第一个节点，获取到锁。

在项目中可以使用curator，这个是Apache封装好的基于zookeeper的分布式锁方案。

**四.技能提升**

**多线程**

**创建线程是几种方式**

方式一：继承Thread类，覆写run方法，创建实例对象，调用该对象的start方法启动线程 方式二：创建Runnable接口的实现类，类中覆写run方法，再将实例作为此参数传递给Thread类有参构造创建线程对象，调用start方法启动

方式三：**创建Callable接口的实现类，类中覆写call方法，创建实例对象，将其作为参数传递给FutureTask类有参构造创建FutureTask对象，再将FutureTask对象传递给Thread类的有参构造创建线程对象，调用start方法启动**

Thread有单继承的局限性，Runnable和Callable三避免了单继承的局限，使用更广泛。**Runnable适用于无需返回值的场景，Callable使用于有返回值的场景**

**Thread的start和run的区别**

**start是开启新线程， 而调用run方法是一个普通方法调用，还是在主线程里执行。**没人会直接调用run方法

**sleep 和 wait的区别**

第一，**sleep方法是Thread类的静态方法，wait方法是Object类的方法**

第二：**sleep方法不会释放对象锁，wait方法会释放对象锁**

第三：**sleep方法必须捕获异常，wait方法不需要捕获异常**

**线程的几种状态**

**新建状态：线程刚创建，还没有调用start方法之前**

**就绪状态：也叫临时阻塞状态，当调用了start方法后，具备cpu的执行资格，等待cpu调度器轮询的状态**

**运行状态：就绪状态的线程，获得了cpu的时间片，真正运行的状态**

**冻结状态：也叫阻塞状态，指的是该线程因某种原因放弃了cpu的执行资格，暂时停止运行的状态，比如调用了wait，sleep方法**

**死亡状态：线程执行结束了，比如调用了stop方法**

**Synchronized 和 lock的区别**

他们都是用来解决并发编程中的线程安全问题的，不同的是

* **synchronized是一个关键字，依靠Jvm内置语言实现，底层是依靠指令码来实现；Lock是一个接口，它基于CAS乐观锁来实现的**
* **synchronized在线程发生异常时，会自动释放锁，不会发生异常死锁，Lock在异常时不会自动释放锁，我们需要在finally中释放锁**
* **synchronized是可重入，不可判断，非公平锁，Lock是可重入，可判断的，可手动指定公平锁或者非公平锁**

**你知道AQS吗**

AQS：AbstractQuenedSynchronizer抽象的队列式同步器。是除了java自带的synchronized关键字之外的锁机制，它维护了一个volatile修饰的 int 类型的，state（代表共享资源）和一个FIFO线程等待队列（多线程争用资源被阻塞时会进入此队列）。

工作思想是如果被请求的资源空闲，也就是还没有线程获取锁，将当前请求资源的线程设置为有效的工作线程，并将共享资源设置为锁定状态，如果请求的资源被占用，就将获取不到锁的线程加入队列。

**悲观锁和乐观锁**

悲观锁和乐观锁，指的是看待并发同步问题的角度

* 悲观锁认为，**对同一个数据的并发操作，一定是会被其他线程同时修改的。**所以在每次操作数据的时候，都会上锁，这样别人就拿不到这个数据。如果不加锁，并发操作一定会出问题。用阳间的话说，就是总有刁民想害朕
* 乐观锁认为，对同一个数据的并发操作，是不会有其他线程同时修改的。它不会使用加锁的形式来操作数据，而是**在提交更新数据的时候，判断一下在操作期间有没有其他线程修改了这个数据**

**悲观锁一般用于并发小，对数据安全要求高的场景,乐观锁一般用于高并发，多读少写的场景，通常使用版本号控制，或者时间戳来解决.**

**你知道什么是CAS嘛**

CAS,compare and swap的缩写，**中文翻译成比较并交换**。**它是乐观锁的一种体现，CAS 操作包含三个操作数 —— 内存位置（V）、预期原值（A）和新值(B)。 如果内存位置的值与预期原值相匹配，那么处理器会自动将该位置值更新为新值 。否则，处理器不做任何操作。**

**Synchronized 加非静态和静态方法上的区别**

实例方法上的锁，锁住的是这个对象实例，它不会被实例共享，也叫做对象锁

静态方法上的锁，锁住的是这个类的字节码对象，它会被所有实例共享，也叫做类锁

**Synchronized(this) 和 Synchronized (User.class)的区别**

Synchronized(this) 中，this代表的是该对象实例，不会被所有实例共享

Synchronized (User.class)，代表的是对类加锁，会被所有实例共享

**Synchronized 和 volatitle 关键字的区别**

这两个关键字都是用来解决并发编程中的线程安全问题的，不同点主要有以下几点

第一：volatile的实现原理，是在每次使用变量时都必须重主存中加载，修改变量后都必须立马同步到主存；synchronized的实现原理，则是锁定当前变量，让其他线程处于阻塞状态

第二：volatile只能修饰变量，synchronized用在修饰方法和同步代码块中

第三：volatile修饰的变量，不会被编译器进行指令重排序，synchronized不会限制指令重排序

第四：volatile不会造成线程阻塞，高并发时性能更高，synchronized会造成线程阻塞，高并发效率低

第五：**volatile不能保证操作的原子性，因此它不能保证线程的安全**，synchronized能保证操作的原子性，保证线程的安全

**synchronized 锁的原理**

synchronized是基于JVM内置锁实现，**通过内部对象Monitor(监视器锁)实 现，基于进入与退出Monitor对象实现方法与代码块同步，监视器锁的实现依赖 底层操作系统的Mutex lock（互斥锁）实现**，它是一个重量级锁性能较低，涉及到用户态到内核态的切换，会让整个程序性能变得很差。

**因此在JDK1.6及以后的版本中**，增加了锁升级的过程，依次为无锁，偏向锁，轻量级锁，重量级锁。而且还增加了锁粗化，锁消除等策略，这就节省了锁操作的开销，提高了性能

**synchronized 锁升级原理**

每个对象都拥有对象头，对象头由Mark World ，指向类的指针，以及数组长度三部分组成，锁升级主要依赖Mark Word中的锁标志位和释放偏向锁标识位。

* 偏向锁（无锁）

**大多数情况下锁不仅不存在多线程竞争，而且总是由同一线程多次获得。**偏向锁的目的是在某个线程 获得锁之后（线程的id会记录在对象的Mark Word锁标志位中），**消除这个线程锁重入（CAS）的开销，看起来让这个线程得到了偏护。(第二次还是这个线程进来就不需要重复加锁，基本无开销)**，**如果自始至终使用锁的线程只有一个，很明显偏向锁几乎没有额外开销，性能极高。**

* 轻量级锁（CAS）：

**轻量级锁是由偏向锁升级来的**，**偏向锁运行在一个线程进入同步块的情况下，当第二个线程加入锁争用的时候，偏向锁就会升级为轻量级锁自旋锁）；没有抢到锁的线程将自旋，获取锁的操作。**轻量级锁的意图是在没有多线程竞争的情况下，通过CAS操作尝试将MarkWord锁标志位更新为指向LockRecord的指针，减少了使用重量级锁的系统互斥量产生的性能消耗。

长时间的自旋操作是非常消耗资源的，一个线程持有锁，其他线程就只能在原地空耗CPU，执行不了任何有效的任务，这种现象叫做忙等（busy-waiting）

* 重量级锁：

**如果锁竞争情况严重，某个达到最大自旋次数(10次默认)的线程，会将轻量级锁升级为重量级锁，重量级锁则直接将自己挂起**，在JDK1.6之前，synchronized直接加重量级锁，很明显现在得到了很好的优化。

虚拟机使用CAS操作尝试将MarkWord更新为指向LockRecord的指针，如果更新成功表示线程就拥有该对象的锁；如果失败，会检查MarkWord是否指向当前线程的栈帧，如果是，表示当前线程已经拥有这个锁；如果不是，说明这个锁被其他线程抢占，此时膨胀为重量级锁。

**乐观锁的使用场景(数据库,ES)**

场景一：**ES中对version的控制并发写**。

场景二：**数据库中使用version版本号控制来防止更新覆盖问题。**

场景三：**原子类中的CompareAndSwap操作**

**AtomicInterger怎么保证并发安全性的**

**通过CAS操作原理来实现的**，就可见性和原子性两个方面来说

它的value值使用了volatile关键字修饰，也就保证了多线程操作时内存的可见性

Unsafe这个类是一个很神奇的类，而compareAndSwapInt这个方法可以直接操作内存，依靠的是C++来实现的，它调用的是Atomic类的cmpxchg函数。而这个函数的实现是跟操作系统有关的，比如在X86的实现就利用汇编语言的CPU指令lock cmpxchg，它在执行后面的指令时，会锁定一个北桥信号，最终来保证操作的原子性

**什么是重入锁，什么是自旋锁，什么是阻塞**

**可重入锁是指允许同一个线程多次获取同一把锁**，比如一个递归函数里有加锁操作

**自旋锁不是锁，而是一种状态，当一个线程尝试获取一把锁的时候，如果这个锁已经被占用了，该线程就处于等待状态，并间隔一段时间后再次尝试获取的状态，就叫自旋**

**阻塞，指的是当一个线程尝试获取锁失败了，线程就就进行阻塞**，这是需要操作系统切换CPU状态的

**你用过JUC中的类吗，说几个**

**Lock锁体系 ，ConcurrentHashMap ，Atomic原子类，如：AtomicInteger ；ThreadLoal ; ExecutorService**

**ThreadLocal的作用和原理**

ThreadLocal，翻译成中国话，叫做线程本地变量，它是为了解决线程安全问题的，它通过为每个线程提供一个独立的变量副本，来解决并发访问冲突问题 - 简单理解它可以把一个变量绑定到当前线程中，达到线程间数据隔离目的。

原理：ThredLocal是和当前线程有关系的，**每个线程内部都有一个ThreadLocal.ThreadLocalMap类型的成员变量threadLocals，它用来存储每个线程中的变量副本，key就是ThreadLocal变量，value就是变量副本。**

当我们调用get方法是，就会在当前线程里的threadLocals中查找，它会以当前ThreadLocal变量为key获取当前线程的变量副本

它的**使用场景比如在spring security中，我们使用SecurityContextHolder来获取SecurityContext**，比如在springMVC中，我们通过RequestContextHolder来获取当前请求，比如在 zuul中，我们通过ContextHolder来获取当前请求

**线程池的作用**

请求并发高的时候，如果没有线程池会出现线程频繁创建和销毁而浪费性能的情况，同时没办法控制请求数量，所以使用了线程池后有如下好处

* **主要作用是控制并发数量，线程池的队列可以缓冲请求**
* **线程池可以实现线程的复用效果**
* **使用线程池能管理线程的生命周期**

**Executors创建四种线程池**

* CachedThreadPool：可缓存的线程池，它在创建的时候，没有核心线程，线程最大数量是Integer最大值，最大空闲时间是60S
* FixedThreadPool：固定长度的线程池，它的最大线程数等于核心线程数，此时没有最大空闲时长为0
* SingleThreadPool：单个线程的线程池，它的核心线程和最大线程数都是1，也就是说所有任务都串行的执行
* ScheduledThreadPool：可调度的线程池，它的最大线程数是Integer的最大值，默认最长等待时间是10S，它是一个由延迟执行和周期执行的线程池

**线程池的执行流程**

corePoolSize，maximumPoolSize，workQueue之间关系。

1. 当线程池中线程数小于corePoolSize时，新提交任务将创建一个新线程(使用核心)执行任务，即使此时线程池中存在空闲线程。
2. 当线程池中线程数达到corePoolSize时(核心用完)，新提交任务将被放入workQueue中，等待线程池中任务调度执行 。
3. 当workQueue已满，且maximumPoolSize > corePoolSize时，新提交任务会创建新线程(非核心)执行任务。
4. 当workQueue已满，且提交任务数超过maximumPoolSize(线程用完，队列已满)，任务由RejectedExecutionHandler处理。
5. 当线程池中线程数超过corePoolSize，且超过这部分的空闲时间达到keepAliveTime时，回收这些线程。
6. 当设置allowCoreThreadTimeOut(true)时，线程池中corePoolSize范围内的线程空闲时间达到keepAliveTime也将回收。

**线程池执行流程 ：** 核心线程 => 等待队列 => 非核心线程 => 拒绝策略

**线程池构造器的7个参数**

* CorePoolSize：核心线程数，它是不会被销毁的
* MaximumPoolSize ：最大线程数，核心线程数+非核心线程数的总和
* KeepAliveTime：非核心线程的最大空闲时间，到了这个空闲时间没被使用，非核心线程销毁
* Unit：空闲时间单位
* WorkQueue：是一个BlockingQueue阻塞队列，超过核心线程数的任务会进入队列排队
* ThreadFactory：它是一个创建新线程的工厂
* Handler：拒绝策略，任务超过最大线程数+队列排队数 ，多出来的任务该如何处理取决于Handler

**线程池拒绝策略有几种**

拒绝策略，当线程池任务超过 最大线程数+队列排队数 ，多出来的任务该如何处理取决于Handler

1. AbortPolicy丢弃任务并抛出RejectedExecutionException异常；
2. DiscardPolicy丢弃任务，但是不抛出异常；
3. DiscardOldestPolicy丢弃队列最前面的任务，然后重新尝试执行任务；
4. CallerRunsPolicy由调用线程处理该任务

可以定义和使用其他种类的RejectedExecutionHandler类来定义拒绝策略。

**你知道ScheduledThreadPool使用场景吗**

这是带定时任务的线程池，EurekaClient拉取注册表&心跳续约就是使用的这个线程池。

**索引部分**

**什么是索引**

**索引是用来高效获取数据的存储结构如同字典的目录一样**，数据库的索引通常使用b+tree来实现，索引树的节点和数据地址相关联，查询的时候在索引树种进行高效搜索，然后根据数据地址获取数据。索引提高了搜索的效率同时增加了索引维护的成本，滥用索引也会降低insert,update,delete的性能。

**Mysql索引有哪些类型**

普通索引：允许重复的值

唯一索引：不允许有重复的值

主键索引：数据库自动为我们的主键创建索引，如果我们没有指定主键，它会根据没有null的唯一索引创建主键索引，否则会默认根据一个隐藏的rowId作为主键索引

全文索引，用来对文本域进行索引，比如text，varchar，只针对MyISAM有效

**索引方式有哪些**

B+树和hash，Myisam和innodb都不支持hash

**Mysql的索引结构原理**

采用了B+树的数据结构,采用B+树的原因，B+树是多叉树，适合存储大量数据，B+树的数据存储在叶子节点，内部节点只存键值，因此B+树每次查询都要走到叶子节点， 查询性能更稳定，同时它的非叶子节点只存储key，因此每个节点能存储更多的key，树的高度变的更低，查询性能更快，而且它的叶子节点能够形成一个链表，支持范围查询，排序 。

**InnoDB的索引结构和MyIsam的索引结构有什么区别**

他们都是用的B+树，不同的是

* **innodb的叶子节点存放的是数据，myisam的叶子节点存放的是数据的地址**
* **innodb中辅助索引的叶子节点存放的是主键索引的键值，myisam中辅助索引的叶子节点存放的也是数据的地址**
* **innodb的索引和数据都存放到一个文件中，myisam的索引和数据分别存放到不同的文件中**

**哪些列不适合创建索引**

不经常查询的列不适合创建索引

不出现在where中的字段不适合创建索引

离散度太低的字段不适合创建索引，比如性别

更新非常频繁的字段不适合创建索引

**哪些因素会造成索引失效**

模糊查询时，通配符放到左边的时候，会导致索引失效 比如 like ''%keyword%''

列是字符串类型，查询条件没有用引号，会导致索引失效

使用了or，in，not in，not exist， !=等，会导致索引失效

查询null值，会导致索引失效

还有mySQL认为全表扫描会比索引查找快，就不会使用索引，比如表里只有一条记录

**什么是辅助索引&什么是覆盖索引**

**除了主键索引之外的其他索引都叫辅助索引，也叫二级检索。辅助索引的叶子节点存储的是主键索引的键值，因此辅助索引扫描完之后还会扫描主键索引，这也叫回表**

**但是如果查询的列恰好包含在辅助索引的键值中，就不会再回表了，这也叫覆盖索引**

**InnoDB辅助索引的叶子节点也存数据吗**

InnoDB辅助索引的叶子节点存放的是，主键索引的键值

因此辅助索引扫描完还会扫描主键索引，也叫回表

但是如果查询的列恰好包含在辅助索引的键值中，就不会再回表了，这也叫覆盖索引

**组合索引的匹配原则**

组合索引向左匹配，我们应该优先选择组合索引，因为对覆盖索引命中率更高，查询性能更高，但是应该考虑列的顺序，因为组合索引会向左匹配

**Like一定会让索引失效吗**

不一定，比如：like "值%" 一样可以使用索引，向左匹配，而 like "%值"或 "\_值"就不能命中索引。

**索引创建的原则有哪些**

**查询较频繁的列应该考虑创建索引**

**不经常查询的列不适合创建索引**

**不出现在where中的字段不适合创建索引**

**离散度太低的字段不适合创建索引，比如性别**

**更新非常频繁的字段不适合创建索引**

**数据库优化**

**哪些因素可能会造成数据库性能问题**

不合理的商业需求，比如实时更新总注册人数，总交易额等等，应该考虑不要实时

对于热点数据的查询并发太高，应该考虑用缓存

数据库结构设计不合理，比如几十个字段集中在一张表，应该考虑分表

SQL语句有问题，比如太多JOIN，很多不需要的字段也要全部查询出来，应该考虑优化SQL

硬件和网络方面的影响

**Mysql的执行流程是怎么样的**

客户端发起SQL查询，首先通过连接器，它会检查用户的身份，包括校验账户密码，权限

然后会查询缓存，如果缓存命中直接返回，如果没有命中再执行后续操作，但是MySQL8.0之后已经删除了缓存功能

接下来到达分析器，主要检查语法词法，比如SQL有没有写错，总共有多少关键字，要查询哪些东西

然后到达优化器，他会以自己的方式优化我们的SQL

最后到达执行器，调用存储引擎执行SQL并返回结果

**优化SQL你从哪些方面着手**

不需要的字段就不要查询出来

小结果集驱动大结果集，将能过率更多数据的条件写到前面

in和not in尽量不要用，会导致索引失效

避免在where中使用or链接条件，这会导致索引失效

给经常要查询的字段建立索引

考虑如果不需要事务，并且主要查询的化，可以考虑使用MyISAM存储引擎

如果表数据量实在太庞大了，考虑分表

**如何去定位慢SQL**

通过druid连接池的内置监控来定位慢SQL

通过MySQL的慢查询日志查看慢SQL

通过show processlist，查看当前数据库SQL执行情况来定位慢SQL

**页面上发起的一个查询很慢，你怎么去优化**

首先看一下硬件和网络层面，有没有什么异常

然后分析代码有没有什么问题，算法有没有什么缺陷，比如多层嵌套循环

最后我们再定位到慢SQL，比如

* 通过druid连接池的内置监控来定位慢SQL
* 通过MySQL的慢查询日志查看慢SQL
* 通过show processlist，查看当前数据库SQL执行情况来定位慢SQL

定位到慢SQL再考虑优化该SQL，比如说

* 不需要的字段就不要查询出来
* 小结果集驱动大结果集，将能过率更多数据的条件写到前面
* in和not in尽量不要用，会导致索引失效
* 避免在where中使用or链接条件，这会导致索引失效
* 考虑如果不需要事务，并且主要查询的化，可以考虑使用MyISAM存储引擎

如果优化SQL后还是很慢，可以考虑给查询字段建索引来提升效率

如果建立索引了还是慢，看一下是不是数据量太庞大了，应该考虑分表了

**你如何看SQL有没有命中索引**

在SQL语句前加上explain，结果中的key就是实际用到的索引

**mysql存储引擎有哪些，有什么区别，如何选择**

主要有innodb，memory，myisam

innodb支持事务，速度相对较慢，支持外键，不支持全文索引

myisam 速度相对较快，支持全文索引，不支持外键，不支持事务，

memory不支持事务，基于内存读写，速度快，支持全文索引

如果对事务要求不高，而且是查询为主，考虑用myisam

如果对事务要求高，保存的都是重要的数据，建议使用innodb，它也是默认的存储引擎

如果数据频繁变化的，不需要持久化，可以使用memory

**下面SQL如何优化**

一个sql ： select sum(amount) from recharge ,来查询总充值，recharge 表数据量达到了上千万，怎么优化

可以考虑建个汇总表来统计总充值，总订单数，总人数等等等

或者采用日报表，月报表，年报表，使用定时任务进行结算的方式来统计

或者看数据能不能使用ES搜索引擎来优化，如果非得要在这个上千万的表中来查询，那就采用分表

**事务相关**

**什么是事务**

一组对数据库的操作，把这一组看成一个再给你，要么全部成功，要么全部失败。

举个栗子，比如A向B转账，A账户的钱少了，B账户的钱就应该对应增加，这就转账成功了，如果A账户的钱少了，由于网络波动等因素转账失败了，B账户的钱没有增加，那么A账户就应该恢复成原先的状态

**事务的四大特性**

原子性：指的是一个事务应该是一个最小的无法分割的单元，不允许部分成功部分失败，只能同时成功，或者同时失败

持久性：一旦提交事务，那么数据就应该持久化，保证数据不会丢失

隔离性：两个事务修改同一个数据，必须按顺序执行，并且前一个事务如果未完成，那么中间状态对另一个事务不可见

一致性：要求任何写到数据库的数据都必须满足预先定义的规则，它基于其他三个特性实现的

**InnoDB如何保证原子性和持久性的**

通过undo log 保证事务的原子性，redo log保证事务的持久性

undo log是回滚日志，记录的是回滚需要的信息，redo log记录的是新数据的备份

当事务开始时，会先保存一个undo log，再执行修改，并保存一个redo log，最后再提交事务。如果系统崩溃数据保存失败了，可以根据redo log中的内容，从新恢复到最新状态，如果事务需要回滚，就根据undo log 回滚到之前的状态

**事务并发问题有哪些**

脏读：事务A读到了事务B修改还未提交的数据

幻读，也叫虚读：事务A两次读取相同条件的数据，两次查询到的数据条数不一致，是由于事务B再这两次查询中插入或删除了数据造成的

不可重复读：事务A两次读取相同条件的数据，结果读取出不同的结果，是由于事务B再这两次查询中修改了数据造成的

第一类丢失更新：也叫回滚丢失，事务A和事务B更新同一条数据，事务B先完成了修改，此时事务A异常终止，回滚后造成事务B的更新也丢失了

第二类丢失更新：也叫覆盖丢失，事务A和事务B更新同一条数据，事务B先完成了修改，事务A再次修改并提交，把事务B提交的数据给覆盖了

**事务隔离级别有哪些，分别能解决什么问题**

读未提交：事务读不阻塞其他事务的读和写，事务写阻塞其他事务的写但不阻塞读，能解决第一类丢失更新的问题，

读已提交：事务读不会阻塞其他事务读和写，事务写会阻塞其他事务的读和写，能解决第一类丢失更新，脏读的问题

可重复读：事务读会阻塞其他事务的写但不阻塞读，事务写会阻塞其他事务读和写，能解决第一类丢失更新，脏读，不可重复读，第二类丢失更新问题

串行化：使用表级锁，让事务一个一个的按顺序执行，能解决以上所有并发安全问题

**MySql的InnoDB是如何保证原子性的**

利用了undo log实现的

undo log记录了这些回滚需要的信息，当事务执行失败或调用了rollback，导致事务需要回滚，就可以利用undo log中的信息将数据回滚到修改之前的样子

**MySql的InnoDB是如何保证持久性的**

利用了redo log实现的

redo log记录的是新数据的备份，在事务提交前，需要将Redo Log持久化，当系统崩溃时，可以根据redo Log的内容，将所有数据恢复到最新的状态

**说一下事务的执行流程(Undolog+Redolog)**

假设有A=1，B=2，两个数据，现在有个事务把A修改为3，B修改为4，那么事务的执行流程：

当事务开始时，会首先记录A=1到undo log，记录A=3到redo log，和记录B=2到undo log，记录B=4到redo log，然后再将redo log写入磁盘，最终事务提交

**解释一下事务并发丢失更新问题，·如何解决**

第一类丢失更新：也叫回滚丢失，事务A和事务B更新同一条数据，事务B先完成了修改，此时事务A异常终止，回滚后造成事务B的更新也丢失了

第二类丢失更新：也叫覆盖丢失，事务A和事务B更新同一条数据，事务B先完成了修改，事务A再次修改并提交，把事务B提交的数据给覆盖了

SQL标准中的四种隔离级别，读未提交，读已提交，可重复读，串行化，都能解决第一类数据更新丢失问题

对于第二类丢失更新问题，可以使用悲观锁也就是串行化来解决，也可以使用乐观锁的方式，比如加一个版本号管理来解决

**InnoDB事务隔离的实现原理是什么**

隔离的实现主要利用了读写锁和MVCC机制

读写锁，要求在每次读操作时需要获取一个共享锁，写操作时需要获取一个写锁。共享锁之间不会产生互斥，共享锁和写锁，写锁与写锁之间会产生互斥。当产生锁竞争时，需要等一个操作的锁释放，另一个操作才能获得锁

MVCC，多版本并发控制，它是在读取数据时通过一种类似快照的方式将数据保存下来，不同的事务看到的快照版本是不一样的，即使其他事务修改了数据，但是对本事务仍然是不可见的，它只会看到第一次查询到的数据

可重复读是只在事务开始的时候生成一个当前事务全局性的快照，而读提交则是每次执行语句的时候都重新生成一次快照

**数据库集群**

**Mysql主从解决什么问题，不能解决什么问题？**

MySQL主从同步，主负责写，从负责读，使用一主多从，能减轻读的压力

但是这不能解决写的压力和主库的单点故障，如果主库的写并发高，可以做成多个主库

**MySql主从复制原理？**

主要依靠binlog来实现的，它记录的是所有的DDL，DML，TCL操作

当主库的数据发生改变时，会将改变记录保存到binlog中

从库新开一个线程将binlog内容发送到从库

从库会发起一个I/O线程请求主库的binlog，并保存到中继日志中

从库新开一个SQL线程，读取中继日志并解析成具体操作，从而将主库更新的内容写到了从库中

**MySql主从配置步骤？**

安装mySQL主从客户端，并配置my.ini

主库需要配置授权从库使用的账号和权限，启动后可以通过show 主库名 status查看状态，我们需要记录File和Position的值，File是对应的binlog文件名，position是当前同步数据的最新行

从库需要配置主库链接信息，包括账号密码和binlog文件名和最新行，然后启动。通过show 从库名 status 检查同步状态，Slave\_IO\_Running 和 Slave\_SQL\_Running 的值都为YES，说明大功告成了

**什么是垂直分表，垂直分库，水平分表，水平分库**

垂直分表，可以理解为按列分表，如果一个表的字段太多了，可以按照使用频率分成不同的表，优化查询性能。比如商品表可以分为商品类型表，商品详情表，商品促销表等等

垂直分库，为了减轻单个数据库压力，我们可以按照业务类型，拆分成多个数据库，比如分布式架构，不同的模块可以有不同的数据库

水平分表，可以理解为按行分表，如果一个表的数据有千万行，查询性能太低，可以拆分成10张小表，每张表保存一百万行数据

水平分库，我们做了水平分表后，表数量太多了也会影响数据库查询效率，我们可以将这些表分到多个数据库中

**分库分表后会出现哪些问题？怎么解决**

会产生分布式事务，以前本地事务就能结局的问题现在要用上Seata分布式事务

垂直分库后跨库查询会导致一个查询结果来源于两个库，可能要用到多线程调用多个库查询

水平分库后一个分页查询的某一页可能来自两个库，可以将两个库的数据合并之后再执行SQL

水平分表后不同的表出现主键重复，可以通过雪花算法来解决

两个库都用到同一个表，那这个公共表的维护可能要用到MySQL主从同步

**你们公司使用的是什么技术来水平分表？还可以有什么技术？有什么区别？**

使用的是sharding-jdbc来实现的，它是由java开发的关系型数据库中间件，读写分离，分库分表操作简单

TDDL，淘宝业务框架，复杂而且分库分表的部分还没有开源

Mycat，要安装额外的环境，不稳定用起来复杂

MySQL官方提供的中间件，不支持大数据量的分不分表，性能较差

**你们使用什么规则来分库分表的？还有哪些规则？**

垂直分库，按照业务进行垂直分库，比如课程表和用户表放到不同数据库

垂直分表，把多字段表拆分少量字段表，比如将课程表分为课程类型表，课程详情表，课程促销表等

水平分表，把海量数据表拆分为多个小表

把商品业务进行水平分库，可以对水平分库后每一个数据库服务器进行集群

**你从哪些方面去优化你的数据库？**

如果是并发高，可以考虑缓存，如果是数据量大可以考虑分库分表，具体如下：

首先应该考虑垂直分库，不同的业务使用不同的数据库

然后进行垂直分表，按照使用频率把字段多的表拆分成若干个表

对经常查询的列建立索引，提高查询效率

设计冗余字段，减少join表的次数

SQL优化，比如尽量使用索引查询

对热点数据应该考虑做缓存，比如首页展示汇总数据

从海量数据中查询数据应该考虑用全文检索

如果查询并发高，可以对mySQL做集群

如果数据量实在太大了，可以考虑水平分表，

水平分表后，表数量还是太多了，可以考虑水平分库

**Mysql的集群有哪些模式？**

一主一从；一主多从；双主；环形多主；级联同步

**单机优化到极致了，可以怎么优化？**

可以考虑做集群，比如一主多从模式，然后对应用做读写分离

**多机优化有哪些方式？**

分表，分库，主从同步

**解释一下分库分表的含义？**

垂直分表，可以理解为按列分表，如果一个表的字段太多了，可以按照使用频率分成不同的表，优化查询性能。比如商品表可以分为商品类型表，商品详情表，商品促销表等等

垂直分库，为了减轻单个数据库压力，我们可以按照业务类型，拆分成多个数据库，比如分布式架构，不同的模块可以有不同的数据库

水平分表，可以理解为按行分表，如果一个表的数据有千万行，查询性能太低，可以拆分成10张小表，每张表保存一百万行数据

水平分库，我们做了水平分表后，表数量太多了也会影响数据库查询效率，我们可以将这些表分到多个数据库中

**水平分表有哪些分表规则？**

按照区间范围分表，比如把用户按照年龄分为新生代表，青年代表，老年代表

按照时间分表，比如按照年来分表，比如登录日志，分成今年的表，去年的表。。

hash分表，通过将某一列的值比如id，通过一定的hash算法来算出对应那张表

雪花算法，通过雪花算法生成id，根据id来算出对应那张表

**能简单说一下你怎么使用shardingjdbc做读写分离的嘛**

首先导入相关的依赖

然后在配置文件中配置datasource，包括主从数据库的名字，主从数据库的连接信息，配置负载均衡

项目中就可以正常使用datasource了，自动做读写分离

**能简单说一下你怎么使用shardingjdbc做读分库分表的嘛**

首先，要改造数据库，比如水平分表，水平分库

在配置文件中，需要做如下配置

* datasource名字，多个数据源就配多个datasource
* 分库策略，比如按照哪一列分库，分库规则
* 分表策略，比如哪些库下面的哪些表，按照那一列分表，分表规则
* 配置公共的表

然后项目中就可以正常使用了

**JVM篇**

**你们用什么工具监控JVM**

jconsule, jvisualvm

**JVM类加载流程**

loading加载：class文件从磁盘加载到内存中

verification验证：校验class文件，包括字节码验证，元数据验证，符号引用验证等等

preparation准备：静态变量赋默认值，只有final会赋初始值

resolution解析：常量池中符号引用，转换成直接访问的地址

initializing初始化：静态变量赋初始值

**JVM类加载器有几种类型，分别加载什么东西，用到什么设计模式？**

1. BootStrap ClassLoader 启动类加载器，加载<JAVA\_HOME>\lib下的类
2. Extenstion ClassLoader 扩展类加载器，加载<JAVA\_HOME>\lib\ext下的类
3. Application ClassLoader 应用程序类加载器，加载Classpath下的类
4. 自定义类加载器

这里是用到了双亲委派模式，从上往下加载类，在这过程中只要上一级加载到了，下一级就不会加载了，这麽做的目的

* 不让我们轻易覆盖系统提供功能
* 也要让我们扩展我们功能。

**JVM组成，以及他们的作用**

运行时数据区：

* 堆：存放对象的区域，所有线程共享
* 虚拟机栈：对应一个方法，线程私有的，存放局部变量表，操作数栈，动态链接等等
* 本地方法栈：对应的是本地方法，在hotspot中虚拟机栈和本地方法栈是合为一体的
* 程序计数器：确定指令的执行顺序
* 方法区：存放虚拟机加载的类的信息，常量，静态变量等等，JDK1.8后，改为元空间

执行引擎：

* 即时编译器，用来将热点代码编译成机器码（编译执行）
* 垃圾收集，将没用的对象清理掉

本地方法库：融合不同的编程语言为java所用

**在JVM层面，一个线程是如何执行的**

线程执行，每个方法都会形成一个栈帧进行压榨保存到虚拟机栈中，方法调用结束就回出栈。调用过程中创建的变量在虚拟机栈，对象实例存放在堆内存中，栈中的变量指向了对中的内存。当方法执行完成就出栈，创建的变量会被销毁，堆中的对象等待GC。

**程序内存溢出了，如何定位问题出在哪儿？**

增加启动参数-XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=d:\ 可以把内存溢出的日志输出到文件，然后通过JVM监视工具VisualVM来分析日志，定位错误所在。在linux服务器也可以使用命令： jmap -dump 来下载堆快照。

**垃圾标记算法**

垃圾标记算法有：引用计数和可达性算法

* 引用计数 ： 给每一个对象添加一个引用计数器，每当有一个地方引用它时，计数器值加1；每当有一个地方不再引用它时，计数器值减1，这样只要计数器的值不为0，就说明还有地方引用它，它就不是无用的对象. 这种算法的问题是当某些对象之间互相引用时，无法判断出这些对象是否已死
* GC Roots ：找到一个对象作为 CG Root , 当一个对象到GC Roots没有任何引用链相连（GC Roots到这个对象不可达）时，就说明此对象是不可用的

**垃圾回收算法**

* 标记清除算法 ：分为标记和清除两个阶段，首先标记出所有需要回收的对象，标记完成后统一回收所有被标记的对象 ；缺点：标记和清除两个过程效率都不高；标记清除之后会产生大量不连续的内存碎片。
* 复制算法 ：把内存分为大小相等的两块，每次存储只用其中一块，当这一块用完了，就把存活的对象全部复制到另一块上，同时把使用过的这块内存空间全部清理掉，往复循环 ，缺点：实际可使用的内存空间缩小为原来的一半，比较适合
* 标记整理算法 ：先对可用的对象进行标记，然后所有被标记的对象向一段移动，最后清除可用对象边界以外的内存
* 分代收集算法 ：把堆内存分为新生代和老年代，新生代又分为Eden区、From Survivor和To Survivor。一般新生代中的对象基本上都是朝生夕灭的，每次只有少量对象存活，因此新生代采用复制算法，只需要复制那些少量存活的对象就可以完成垃圾收集；老年代中的对象存活率较高，就采用标记-清除和标记-整理算法来进行回收。

**垃圾回收器有哪些**

* 新生代：Serial ：一款用于新生代的单线程收集器，采用复制算法进行垃圾收集。Serial进行垃圾收集时，不仅只用一条线程执行垃圾收集工作，它在收集的同时，所有的用户线程必须暂停（Stop The World
* 新生代：ParNew : ParNew就是一个Serial的多线程版本`，其它与Serial并无区别。ParNew在单核CPU环境并不会比Serial收集器达到更好的效果，它默认开启的收集线程数和CPU数量一致，可以通过-XX:ParallelGCThreads来设置垃圾收集的线程数。
* 新生代：Parallel Scavenge(掌握) Parallel Scavenge也是一款用于新生代的多线程收集器，与ParNew的不同之处是，ParNew的目标是尽可能缩短垃圾收集时用户线程的停顿时间，Parallel Scavenge的目标是达到一个可控制的吞吐量.Parallel Old收集器以多线程，采用标记整理算法进行垃圾收集工作。
* 老年代：Serial Old ,Serial Old收集器是Serial的老年代版本，同样是一个单线程收集器，采用标记-整理算法。
* 老年代CMS收集器是一种以最短回收停顿时间为目标的收集器，以“最短用户线程停顿时间”著称。整个垃圾收集过程分为4个步骤
  + 初始标记：标记一下GC Roots能直接关联到的对象，速度较快
  + 并发标记：进行GC Roots Tracing，标记出全部的垃圾对象，耗时较长
  + 重新标记：修正并发标记阶段引用户程序继续运行而导致变化的对象的标记记录，耗时较短
  + 并发清除：用标记-清除算法清除垃圾对象，耗时较长

整个过程耗时最长的并发标记和并发清除都是和用户线程一起工作，所以从总体上来说，CMS收集器垃圾收集可以看做是和用户线程并发执行的。

* 老年代：Parallel Old ,Parallel Old收集器是Parallel Scavenge的老年代版本，是一个多线程收集器，采用标记-整理算法。可以与Parallel Scavenge收集器搭配，可以充分利用多核CPU的计算能力。
* 堆收集：G1 收集器, G1 收集器是jdk1.7才正式引用的商用收集器，现在已经成为jdk1.9默认的收集器。前面几款收集器收集的范围都是新生代或者老年代，G1进行垃圾收集的范围是整个堆内存，它采用“化整为零”的思路，把整个堆内存划分为多个大小相等的独立区域（Region）在每个Region中，都有一个Remembered Set来实时记录该区域内的引用类型数据与其他区域数据的引用关系（在前面的几款分代收集中，新生代、老年代中也有一个Remembered Set来实时记录与其他区域的引用关系），在标记时直接参考这些引用关系就可以知道这些对象是否应该被清除，而不用扫描全堆的数据

Jdk1.7.18新生代使用Parallel Scavenge,老年代使用Parallel Old

**Minor GC和Full GC**

新生代的回收称为Minor GC，新生代的回收一般回收很快，采用复制算法，造成的暂停时间很短 ，而Full GC一般是老年代的回收,并伴随至少一次的Minor GC，新生代和老年代都回收，而老年代采用标记-整理算法，这种GC每次都比较慢，造成的暂停时间比较长`，通常是Minor GC时间的10倍以上。尽量减少 Full GC

**JVM优化的目的是什么？**

优化程序的内存使用大小，以及减少CG来减少程序的停顿来提升程序的性能。

**堆怎么调，栈怎么调**

-Xms : 初始堆，1/64 物理内存

-Xmx : 最大堆，1/4物理内存

-Xmn :新生代大小

-Xss : 栈大小

**设计模式**

**什么是单例，如何实现**

一个类只能有一个实例，主要用于需要频繁使用的对象避免频繁初始化和销毁来提高性能，或者资源需要相互通信的环境

主要实现方式有，饿汉模式，懒汉模式，枚举，静态内部类

饿汉模式，是在类加载过程中就将这个单例对象实例化，需要将构造方法私有化，定义一个成员变量并new一个该类的实例作为初始值，提供一个公共的静态方法获取这个实例

懒汉模式，是在使用时才创建这个单例对象，需要将构造方法私有化，定义一个该类的成员变量不赋初始值，提供一个获取实例的公共静态方法。特别注意这个方法需要保证多线程环境下的并发安全性，可以通过DCL加volatile关键字来解决

枚举，直接在枚举中定义字段，它就是单例并且线程安全的

静态内部类，在类中搞一个静态内部类，在静态内部类中搞一个目标类的静态成员变量并且new一个实例作为初始值。然后在目标类中定义一个获取实例的静态方法，方法返回的就是静态内部类中的成员变量。这种方式能保证线程安全，也能实现延迟加载。缺点是这种方式传参不太方便

**模板模式的作用**

定义一个算法骨架，而将某个或多个具体的实现延迟到子类中，使得子类可以在不修改当前算法的结构情况下，重新定义当前算法的某些特定步骤

比如考试中所有考生的试卷都一样，答案由每个考生自己完成

**什么是适配器模式**

将不兼容的接口转换为可兼容的接口的中间类

比如HandlerInterceptorAdapter ，我们定义拦截器时不需要覆写HandlerInterceptor中的所有方法，因为适配器类帮我们做了空实现。但JDK1.8之后，给接口中增加了默认方法，可以有方法体，因此这些适配器类已经失去作用了

**什么是代理模式？有几种代理？**

不直接使用实际对象，通过调用代理对象间接调用实际对象，主要用作对实际对象的增强，分为静态代理，JDK动态代理，CGLIB动态代理

**JDK动态代理和CGLIB动态代理的区别？**

JDK动态代理是jdk提供的，我们可以直接使用，而CGLIB需要导入第三方库

JDK动态代理是利用反射机制生成一个实现代理接口的匿名类，在调用目标方法前调用InvokeHandler来处理

CGLIB动态代理是先加载目标类的class文件，然后修改其字节码生成子类来实现的

**常见的设计模式说一下**

单例模式：一个类只能有一个实例，分为饿汉模式（迫切加载）和懒汉模式（延迟加载）和枚举。

工厂模式：隐藏了产品的复杂创建过程，实现生产功能的复用，让产品生产更加高效。分为简单工厂（需要来回切换生产线），工厂方法（开设新的生产线），抽象工厂（制定创建产品的接口，让子工厂选择创建哪种产品）

在Spring中各种的BeanFactory创建bean都用到了

模板模式：定义一个算法骨架或者算法的流程，而不同的实例实现方式不同，将某个或多个具体的实现延迟到子类中，比如RedisTemplate实现了RedisOperations，ElasticSearchTemplate实现了ElasticsearchOperations

代理模式：不直接使用实际对象，通过调用代理对象间接调用实际对象，主要用作对实际对象的增强，分为静态代理，JDK动态代理，CGLIB动态代理比如Spring的AOP原理就是动态代理，当目标对象实现了接口会使用JDK动态代理，没有实现接口会使用CGLIB动态代理

适配器模式：将不兼容的接口转换为可兼容的接口的中间类，比如HandlerInterceptorAdapter ，我们定义拦截器时不需要覆写HandlerInterceptor中的所有方法，因为适配器类帮我们做了空实现。但JDK1.8之后，给接口中增加了默认方法，可以有方法体，因此这些适配器类已经失去作用了

观察者模式：当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新，比如Spring中的ApplicationListener

**数据结构**

**数据结构有哪几种分类**

按照逻辑结构分

* 集合：没有相互关系的一堆数据
* 线性结构：元素存在一对一的相互关系
* 树形结构：元素存在一对多的相互关系
* 图形结构：元素存在多对多的相互关系

按照物理结构分

* 顺序存储结构：用一组地址连续的存储空间依次存储线性表的数据元素，也叫顺序存储结构，比如数组
* 链接存储结构：用一组任意的存储空间来存储线性表中的数据元素，不要求相邻元素在物理位置上也相邻，比如链表
* 数据索引存储结构：建立附加的索引来标识节点的地址，通过索引，可以很快检索数据
* 数据散列存储结构：将数据元素的存储位置与关键字之间建立确定的对应关系，加快查找的速度，又叫hash存储

**数组和链表在内存中的存储结构有什么区别**

数组在内存中是一组连续的存储空间，它随机存取元素性能很高，但是插入和删除操作，需要移动其他元素，因此性能很低

链表在内存中的存储空间可以是不连续的，而在每一个元素中都保存相邻节点的指针，因此它的存储密度相对较小，查找的性能低，因为需要从第一个元素依次遍历，但是它的插入和删除操作性能很高，因为它不需要移动节点，只需要改变相邻节点指针就行了，同时它更容易造成内存的碎片化

**说一下散列存储(Hash存储) ， 什么是Hash冲突 ， 有什么解决方案**

散列存储，它通过把关键码的值映射到表中的一个位置，来提高查询的速度。而这个映射函数叫做散列函数。

哈希冲突，也叫哈希碰撞，指的是两个不同的值，计算出了相同的hash，也就是两个不同的数据计算出同一个下标，通常解决方案有：

* 拉链法，把哈希碰撞的元素指向一个链表
* 开放寻址法，把产生冲突的哈希值作为值，再进行哈希运算，直到不冲突
* 再散列法，就是换一种哈希算法重来一次
* 建立公共溢出区，把哈希表分为基本表和溢出表，将产生哈希冲突的元素移到溢出表

**说说 数组，链表，循环，嵌套循环的时间复杂度**

时间复杂度是用来度量算法执行的时间长短，通常我们用Ｏ(f(n))渐进时间复杂度来衡量，比如说

* 要在 hash 表中找到一个元素就是 O(1)
* 要在无序数组中找到一个元素就是 O(n)
* 访问数组的第 n 个元素是 O(1)
* 二分搜索的时间复杂度最好的情况是 O(1)，最坏情况（平均情况）下 O(log n)
* 访问链表的第 n 个元素是 O(n)
* 一个For循环是O(n)
* 两个For循环嵌套是O(n2)
* 三个Foreach嵌套是O(n3)

**JDK中线性结构的集合有哪些**

数组：按照顺序物理结构存储，ArrayList

链表：按照链式物理结构存储，LinkedList

栈：LIFO后进先出的线性存储结构，分为用数组实现的顺序栈，用链表实现的链栈

队列：FIFO先进先出的线性存储结构，分为顺序队列和链式队列

串：特殊的线性存储结构，String，StringBuffer，StringBuilder

**你说一下树形结构对比线性结构的优势**

线性结构，对于大量的输入数据，访问时间很长，效率很低，树形结构的优势在于它查找数据性能很高

**说一下树的分类，以及你对它们的理解**

树有二叉树，多叉树，他们特点如下

* 二叉树：树中任意节点最多只有两个分叉的树，它又分为二叉排序树，平衡二叉树，赫夫曼树，红黑树
* 二叉排序树，它是一个有序的二叉树，优势在于查找插入数据的性能很高，但是可能会出现倾斜而变成数组
* 平衡二叉树，二叉排序树进化形态，要求任何节点的两颗字数高度差不大于1。它的查询性能很高，但是每次增删元素，会重排序导致性能低
* 红黑树，自平衡二叉树，要求根节点和叶子节点是黑色，其他节点红黑交替，在任何一个子树中，从根节点向下走到空姐点的路径经过的黑节点数相同。从而保证了平衡。它的查询性能比平衡二叉树稍低，插入和删除元素的性能大幅提高。

多叉树：解决二叉树存储大规模数据时，深度过大而导致IO性能低，查询效率低的问题，常见有B树和B+树，字典树，后缀树等等

* B树，自平衡的树，一个节点可以存储多个key，和拥有key数量+1个分叉，适用于读写相对大的数据块，比如文件系统，数据库索引。因为相对二叉树来说，节点存储key越多，分叉越多，需要的节点越少，树高越矮，IO次数少，查询效率越高。
* B+树，B树升级版，它的内部节点只存储key，不存储具体数据，叶子节点存放key和具体数据。这就使得每个节点可以存更多的key，树的高度更低，查询更快，同时它每次查询都会到叶子节点，查询速度更稳定。并且所有的叶子节点会组成一个有序链表，方便区间查询

**有是二叉树为什么要出现多叉树**

因为二叉树在大规模的数据存储中，树会高的没谱，这会导致IO读写过于频繁，查询效率低下

多叉树可以解决这个问题，它每层可以存放更多的数据，因此能大幅度降低树的深度，提高查询性能

**B-tree和b+tree的区别**

一是节点存储内容上的区别：B树每个节点都可以存放key，存放数据，而B+树所有内部节点只存放key，叶子节点存放key和数据，因此它的节点能存放更多数据，降低树高，查询性能更快

二是B+树所有的叶子节点会构成一个链表结构，方便区间查找和排序

**说一下ES用到了什么数据结构**

ES是使用了数据索引存储结构，它是通过为关键字建立索引，通过索引找到对应的数据，这种索引也叫倒排索引，可以实现快速检索

**五.项目部分**

**浏览器输入一个域名，它是怎么去执行的？**

1. 首先带着域名去hosts文件中看有没有配置对应的本地域名，如果有就以配置的ip进行访问
2. 如果hosts没有配置，就会请求DNS服务器解析域名得到对应的IP然后发起访问
3. 这时候请求就会打到服务器上可能是Nginx也有可能直接打到Tomcat.

**请求在你的项目中是怎么执行的？**

后端使用zuul网关，请求先到达zuul网关，zuul做登录检查，zuul网关底层整合ribbon把请求路由到下游微服务，服务之间使用OpenFeign进行通信。执行成功后原路返回结果。

**如果zuul网关挂了怎么办？**

可以做zuul集群，使用Nginx做负载均衡到zuul集群，然后Nginx可以采用双机主备，或者双机互备做集群防止单点故障。如果并发非常高可以加上LVS做负载。

**如果有人用脚本刷你们的短信接口怎么办**

首先，可以设置图形验证码，流量错峰

其次，可以获取请求的ip地址，手机号，发送时间，并保存到发送短信记录的日志中，对于短时间多次请求的ip地址，手机号，可以拦截不执行发送手机验证码

再次，可以设置单位时间内发送短信的总数量，比如设定1秒最多只发送10条验证码。但这种方式会降低并发性

**非对称加密，什么是数字签名**

非对称加密是一种算法，指的是加密和解密时使用不同的密钥，其中私钥不可公开，公钥可以公开。

数字签名就是在非对称加密的基础上，使用私钥加密，公钥解密，主要用来防止数据被篡改，实现安全传输的目的

**Oauth2的四种授权模式**

oauth协议是一个安全的开放授权标准，与传统的授权方式相比，它不会使第三方触及到用户的账号信息，比如用户名，密码。Oauth2有四种授权模式

一、授权码模式，它是功能最完整，流程最严密的授权模式

二、简化模式，直接从前端渠道获取token，容易受安全攻击

三、用户名密码模式，使用用户名和密码登录的应用，比如桌面APP

四、客户端凭证模式，用户直接向客户端认证，客户端以自己的名义向第三方索取服务

**要求每天早上 1点统计前一天的平台注册人数，怎么做**

使用定时任务每日结算即可。把结算的数据保存到一个统计表中

**使用Quzrtz定时任务做订单超时关单有什么问题**

数据量大的时候，定时任务扫描表性能会很差，而且多数都是空扫描，还有延迟问题，

对于我们的小型项目，可以使用quartz定时器，使用起来也很简单方便，但如果是高并发，比如秒杀等业务，可以使用RabbitMQ的延迟队列来实现，也可以使用Redis来做延迟队列。

**讲一下你做过的比较复杂的业务**

省略...

**什么是RBAC ， 相关表怎么设计的？**

RBAC：Role-Based Access Control首字母缩写，意为基于角色的访问控制。基本思想是对系统操作的各种权限不是直接授予具体的用户，而是在用户集合与权限集合之间建立一个角色集合。

将权限与角色相关联，用户通过成为适当角色的成员而得到这些角色的权限。极大地简化了权限的管理。这样管理都是层级相互依赖的，权限赋予给角色，而把角色又赋予用户，这样的权限设计很清楚，管理起来很方便。

实现RBAC，需要将用户对权限的多对多关系，转化为用户对角色，角色对权限的多对多关系，因此在数据库中，需要在用户，角色，权限中分别加入中间表，即用户表，用户和角色关系表，角色表，角色和权限关系表，权限表

**在VUE中，什么是MVVM**

MVVM，Model–View–ViewModel首字母缩写，是一种软件架构模式。

其中Model指的是模型，包括数据和一些基本操作

View指的是视图，页面渲染结果

ViewModel指的是模型与视图间的双向操作

MVVM的思想就是数据模型和视图的双向绑定，只要数据变化，视图会跟着变化，只要视图被修改，数据也会跟者变化

**讲几个VUE的指令**

v-text：给元素填充纯文本内容

v-html：给元素填充内容，与v-text的区别是它会把内容的html符号进行渲染

v-for：遍历数字，字符串，数组，对象

v-bind：将data中的数据绑定到标签上,作为标签的属性

v-model：创建双向绑定，表单的值被修改时会自动修改data中的数据，data中的值变化时页面也会被修改

v-show：根据表达式的真假值，切换元素的css属性

v-if：根据表达式的真假值，销毁或重建元素

v-on：绑定事件

**webpack的作用**

VUE项目需要打包后才能部署

首先，它可以将ES6等高级语法，编译成各个浏览器都认识的语法

其次，它可以将相互依赖的许多散碎文件搞成一个整体，提高网页访问的效率

再次，它可以将代码压缩，减小代码体积

**Vue中定义组件分为几种，有什么区别**

组件是一种自定义的元素标签，可以对功能封装，提高代码复用性，分为全局组件和局部组件两种

* 全局组件，是在所有vue挂载的标签中都有效，
* 局部组件，只在当前vue所挂载的标签中有效

**讲一下你用过ElementUI的哪些组件**

基础组件，比如按钮Button，图标Icon

表单组件：比如表单Form，单选框Radio，多选框Checkbox，输入框Input，选择器Select，级联选择器Cascader

其他组件：比如Dialog对话框，消息提示Message

**你们Redis做登录是怎么处理登录信息过期的？**

给保存在Redis中的token设置过期时间来处理登录过期的,为了防止已登录用户在访问后台时突然遭遇登录过期的情况，我们在后台接收到用户访问时，重新设置token的过期时间写入Redis，则用户访问期间就不会突然过期了

**讲一下你们的登录实现方案**

当用户第一次发起登录请求，后台生成一个token保存到Redis中

将生成的token返回给用户端

用户端使用用浏览器中的localStorage保存token

通过axios的拦截器，给每次请求的请求头都加上token

服务端收到token，就能在Redis中找到对应的数据

**三方登录流程讲一下**

1.用户发起微信登录请求

2.后端获取请求二维码的连接，重定向到扫码界面

3.用户使用微信扫一扫并同意授权

4.后端回调获取授权码，并将授权码作为参数，重定向到前端跳转页面

5.前端将授权码返回后端，后端根据授权码获取token

6.后端根据token获取openId

7.根据openId查询微信用户表

* 如果查到有用户信息，且已关联本地账户，就默认登录
* 如果有查到用户信息，但没有关联本地账户，就跳转本地账户绑定页面，
* 如果没有查到用户信息，就向微信平台发起请求查询用户基本信息，添加到微信用户信息表，再跳转本地账户绑定页面

8.执行绑定逻辑时，根据手机号判断是否有本地账户，如果有就直接绑定，如果没有就自动注册再绑定，绑定成功后就默认登录

**讲一下什么是非对称加密，什么是数字签名，数字签名的作用是什么？**

非对称加密是一种算法，指的是加密和解密时使用不同的密钥，其中私钥不可公开，公钥可以公开。

数字签名就是在非对称加密的基础上，使用私钥加密，公钥解密，主要用来防止数据被篡改，实现安全传输的目的

**京东的首页的商品分类，让你设计表，你怎么设计**

首先可以看出表的结构是自关联，三层的树状结构，分类表的字段可以有主键id，商品名，创建时间，修改时间，上架时间，下架时间，商品数量，排序，图标，父级id

**如何查询出树状结构的课程分类数据**

首先，在entity中加入子分类字段children

查询方式有四种

* 第一，使用嵌套for循环，循环体内查询每一层级的数据，并关联到children。当然这也可以使用递归函数来实现
* 第二，使用mybatis的嵌套查询，也就是主查询加额外子sql查询的方式
* 第三，使用mybatis的嵌套结果，也就是join连表查询的方式
* 第四，只使用一次查询，将所有数据查询出来，通过一种算法来实现：除了第一级，其他所有数据都关联到自己的父级分类，结果返回第一级数据就可以

第一，第二种方式，当层级多的时候查询性能极低，第三种方式一般只能查询两层结构，第四种方式性能最高，适用于数据量本身并不大但层级很多的场景

所有课程的数据本身体量小，层级多，因此采用了第四种方式。

**你们系统使用Redis缓存了哪些东西？用Redis的什么结构去存储的？**

登录信息login，使用的是String结构存储

手机验证码code，使用的是String结构

课程分类course\_type ，使用的是String结构

购物车保存，使用的是Hash结构

**课程发布流程讲一下**

发布课程两大步

第一步，将课程的状态改为上线并保存到数据库中，

第二步，将课程信息保存到ES中，方便门户网站展示

**你们课程相关的表是怎么设计的？主要的字段说一下**

我们按照字段的使用频次，垂直分表来设计，分为课程主表，课程详情表，课程类型表，课程市场详情表。

课程主表，包括主键id，课程名称，课程类型id，课程上下线状态，适用人群，课程等级，课程所属机构等，并且冗余了课程类型名，课程价格字段来提高前台的查询性能

课程详情表，包括课程简介，课程详情

课程市场详情表，包括课程价格，促销活动，活动过期时间

课程类型表，包括主键id，类型名，创建修改时间，课程数量，父级id

其中课程主表和课程详情表、课程主表和课程市场详情表，都是一对一的关系，他们采用相同的主键id来相互关联。课程主表和课程类型表是多对一的关系，在课程主表添加类型id来相互关联

**讲一下你们这个项目的主线业务**

我们项目分为两大版图，

入驻我们平台的培训机构，可以发布相关课程，入驻平台的企业，可以发布相关的就业招聘信息

门户网站的大众用户，可以选择培训机构发布的课程来进行学习，可以选择企业发布的招聘信息来就业

**你们项目最大并发是多少**

俺们项目是按照最高2000 QPS设计的，实际并发数运维在统计，俺也不太清楚

**你们项目最大表数量是多少**

俺们项目都有分库分表，按服务拆分多个数据库，对于有些数据量大的表，我们也是按照字段的使用频率，拆分成多个表，比如课程表拆分成课程主表，课程详情表，课程分类表等等。

但是有些表比如日志，流水相关的表，数据量还是很大的

**.说一下你们课程搜索的那个业务方法的大致逻辑**

首先，课程在发布的时候，就同时将课程信息存放到ES中，信息中包括了需要查询的字段，如课程标题，课程分类，课程等级，机构名，销量，浏览量，上线时间，价格等等

接下来，根据用户在前台发送的查询条件，在ES中搜索对应的课程，并作关键字高亮处理，排序和分页处理，然后返回前台

**项目并发高处理过不过来怎么办**

前端优化：

* 使用页面静态化技术由Nginx实现动静分离、
* CDN加速加快响应速度、
* 使用验证码使流量错峰等手段最大限度的降低并发

后端优化：

* Nginx+LVS负载，也可以多机房部署，分流
* 从架构上使用分布式、集群分散并发量，
* 从数据结构上使用缓存如Redis减少数据读写时间，
* 从处理方式上采用如RabitMQ队列实现异步响应，
* 资源隔离比如使用Hystrix的信号量隔离来限流，同时做好备用方案比如Hystrix的熔断降级策略等等

**讲一下你们的微服务授权方案 你还知道有哪些方案吗？**

我们使用的是SpringSecurity+Oauth2+JWT，认证服务器负责颁发token，资源服务器负责认证和授权

或者也可以将认证工作交给网关zuul，资源服务器只负责授权工作。

另外常见的授权方案还有，单点登录，用户只用在某个服务上登录，访问其他服务时就不需要登录了，这就要求每个面向用户的服务都必须于认证服务交互，会产生大量重复的工作

分布式会话，它是将用户认证信息存储在共享容器比如redis中，通常会以会话作为key，当用户访问微服务时，就从redis中获取认证信息。这对安全存储有较高的要求，复杂度高

讲一下你们微服务认证授权的整体流程

客户端访问认证服务器，认证服务器验证用户名密码，然后颁发token

客户端保存token，并且每次访问服务时都携带token

资源服务器接收到客户端请求，会验证token信息，认证通过后返回资源

你们为啥要用JWT

一个字，安全

我们做了认证授权后，每次客户端访问资源服务器，都需要远程调用认证服务器进行token的校验和授权，才能访问到资源。这是很好性能的，因此我们考虑将签名信息直接保存到客户端，那就不需要每次都向认证服务器认证授权了。

但是这有有一个新的问题，这些敏感数据赤裸裸的存到客户端不安全！而JWT就能解决这个问题。它支持非对称加密算法对信息加密，保证了信息安全

另外，JWT以json对象的形式传递信息，解析更方便

可以再令牌中定义内容，方便扩展

**Oauth2的授权模式有哪些，分别使用在什么场景？**

授权码模式：它是功能最完整、流程最严密的授权模式

简化模式：跳过授权码，直接再浏览器端申请令牌

用户名密码模式：客户向客户端提供用户名密码，建立在用户对客户端高度信赖的基础上

客户端模式：客户端以自己的名义，要求服务提供商提供服务

**Oauth2认证，如果Token过期了你们是怎么处理的**

首先，我们会在前端设置axios后置拦截，检查是否是token过期，判断一下如果返回401，就代表token过期了

然后从localStorage中获取刷新refresh\_token，并发送请求获取新的token

后台接收到前台的刷新token请求，拼接完整的刷新token的url，发送http请求获取到新的token并返回客户端

客户端收到新的token就把旧的token覆盖掉，最后把之前的请求再重新发送一次

**Oauth2认证，如果Token被盗了怎么办？**

首先，我们需要对token设置过期时间，这个时间可以根据需要设置短一点

然后，可以在token中加入客户身份标识，比如客户的ip地址，如果短时间内ip地址频繁变动，就标记为异常状态，并给用户发送信息，提示账户有风险

**秒杀的整体流程详细说一下**

秒杀的商品和库存是缓存到Redis的，库存使用信号量，做的是秒杀预减库存方案。用户发起秒杀，直接走Redis秒杀商品，满足资格就预减库存，然后预创订单写入Redis。整个秒杀流程是不做数据罗库的。

此时把订单号返回给客户端，用户带着订单号进入订单确认页面进行下单，用户确认下单，再把Redis中的预创订单写入订单数据，同时做库存同步。紧接着就是调用支付接口做支付。

**如果流量更高，比如：每秒10W请求，应该怎么处理**

Lvs+Nginx集群+下游服务集群。如果流量再高，就使用CDN分流。

**说一下支付超时处理方案？延迟队列和死信队列是什么意思？**

支付超时使用MQ延迟队列来处理，把消息投递到一个设置了过期时间的队列中，**达到过期时间消息会被转发给另外一个“死信队列”**

**设置了过期时间的队列就是延迟队列，过期的消息叫着死信消息，存放死信消息的队列叫死信队列。**

**整个秒杀流程你用到了哪些队列**

**下单业务中用到了一个队列，订单超时用到一个队列，支付结果处理用到一个队列。**

**秒杀成功，返回给用户的数据是什么？**

**预创订单号，前台通过这个订单号来进行下单。**

**你们怎么处理超卖**

Redisson分布式锁，信号量来保证库存不超卖

**如何提高接口的qps**

一方面：提高并发数

1.多线程,尽量用线程池 (线程个数：CPU核数 / (1 - 阻塞系数(IO密集型接近1，计算密集型接近0)))

2.适当调整连接数(Tomcat,Redis，Mysql等连接数)

3.集群

二方面：提高接口响应速度

1.减少和数据库交互，使用Redis代替

2.使用异步方案，比如MQ

3.使用并发编程，多个线程同时工作

4.减少服务的调用链

5.实在要连数据库，考虑数据库优化

**你们这个前后端分离项目是怎么部署的**

前后端分开部署，前端使用Nginx部署，

后端使用Springboot内嵌的tomcat部署，

分开部署后，通过代理解决前后端域名不一致的跨域问题

**前后端分离的好处**

第一，专人干专事，前后端同时开发，效率更高

第二，责任分离，避免了前后端相互踢皮球的现象

第三，前后端解耦合，一套后端可以处理不同的前端，包括app端，浏览器端

第四，分开部署，减轻了服务器压力

第五，页面显示东西再多也不怕，数据都是异步加载，就算后端服务器挂了，前端页面也能访问，虽然没有数据

第六，前端分离出去，后端写一套接口就可以适用于web,app端

**你们用什么做项目代码管理的**

使用主流的Git管理项目

**讲讲Git相对于SVN的区别**

第一。Git是每个攻城狮都有自己的版本库，可以在自己的库上任意操作提交代码

第二。Git在每个工程只产生一个.git目录，而SVN会在每个目录下都生成.svn目录

第三。Git能快速切换分支，且合并文件的速度比SVN快

第四。Git采用分布式版本库，内容完整性更好

**你们微服务项目怎么部署**

docker 容器 ，使用Jnekins做持续集成。

**讲几个Git的命令**

git clone：从远程仓库克隆项目到本地

git add：添加代码到本地仓库管理

git commit：提交add后的代码到本地仓库

git push：推送本地仓库文件到远程仓库

git pull：拉取远程仓库中的代码到本地仓库

**六.运维篇**

**linux**

**有使用过linux吗 ， 讲几个命令**

* 查看目录 : ls
* 切换目录： cd
* 拷贝：cp
* 远程拷贝 ：scp
* 移动 : mv
* 删除：rm
* 查看文本内容：cat
* 编辑文本： vi
* 查找：find
* 远程拷贝：scp
* 创建目录 ： mkdir
* 创建文件：touch

**Linux根目录下的几个核心目录**

* /bin : 二进制文件
* /dev : 设备文件
* /etc : 配置文件
* /home : 用户的主目录，在 Linux 中，每个用户都有一个自己的目录，一般该目录名是以用户的账号命名的
* /root: 该目录为系统管理员，也称作超级权限者的用户主目录。
* /sbin : s 就是 Super User 的意思，是 Superuser Binaries (超级用户的二进制文件) 的缩写，这里存放的是系统管理员使用的系统管理程序。

**周日凌晨零点零分定期备份 /user/backup到 /tmp 目录下，如何做?**

使用crontab即可做到，如下配置：

crontab -e 0 0 \* \* 7 /bin/cp /user/backup /tmp

**Linux中你怎么排查项目问题？查看项目日志你一般怎么做?**

查看tomcat日志，使用tail命令tail -n N filename.txt 。n是查看行数

**怎么查看进程**

ps -ef | grep 软件名

**常用的压缩命令**

使用 ： tar -zcvf压缩 ， tar -zxvf 解压缩

或者使用： zip 压缩成zip, unzip解压

**部署过项目么？大概讲一讲如何部署的**

单体应用的部署是比较简单的，前端打包上传使用Nginx，后端打包成war，可以使用Tomcat来部署，如果是SpringBoot的可以默认打包为jar，直接java -jar 启动。如果用到其他组件，比如Redis可以直接在服务器安装，然后项目指向其IP即可。

如果项目的组成部分比较多，比如：项目后端，前端，Redis,Mysql等等都涉及到，那么可以使用Docker来部署，这样更好管理应用之间的内存和资源分配。

**你们这个服务器的配置是怎么样的**

我们微服务有20个服务器，业务系统是8核CPU,16G内存，有些服务配置还要低一些，视频处理系统是12核CPU，24G内存。通过NFS方式共享20T硬盘。

**Docker**

**讲讲什么是Docker**

docker是一个容器技术，最大的好处是做资源的分配和管理，传统的linux部署项目不好管理内存等资源的分配，造成了应用之间资源竞争的情况，Dcoker的出现解决了这一问题。我们可以把我们的应用打包成Docker的镜像，然后启动成容器。容器和容器之间相互隔离也可以互相通信。就类似于有多个主机一样。

**讲几个Docker的命令**

docker images ：查看本地镜像

docker search : 搜索镜像

docker pull : 下载镜像

docker push : 上传镜像到仓库

docker rmi : 删除镜像

docker run : 创建并启动一个容器

docker ps : 查看容器

docker rm :删除容器

docker stop : 停止容器

docker kill :停止容器

docker start : 启动容器

docker exec -it 容器名 /bin/bash : 进入容器

docker exit :退出容器

docker cp : 拷贝文件到容器，或者从容器中拷贝文件到linux

docker logs : 查看容器的日志

**怎么把文件上传到容器中**

docker cp 或者在启动容器的时候增加 -v 做目录映射

**某个服务不可访问了你怎么排查**

服务不可访问，那就是容器出问题了，我会去找到对应的容器是不是挂了，或者使用docker logs 查看日志根据错误日志来排错。

**容器之间怎么通信**

使用容器IP通信，但是容器重启IP会变动，不建议

使用端口映射也可以通信，但是内网部署的应用不需要做端口映射，所以这个不建议用

使用--link 名字进行通信

使用桥接网络通信

**对于Redis和zuul网关你怎么部署**

首先肯定要下载一个redis的镜像， 对于zuul的镜像可以使用docker插件对zuul进行打包。

redis是不需要暴露给外网的，所以不要做端口映射，可以使用--link或桥接网络通信 ，而zuul是服务访问入口需要做端口映射进行外网部署。