**接口幂等?**

幂等性原本是数学上的概念，用在接口上就可以理解为：同一个接口，多次发出同一个请求，必须保证操作只执行一次。  
调用接口发生异常并且重复尝试时，总是会造成系统所无法承受的损失，所以必须阻止这种现象的发生。  
比如下面这些情况，如果没有实现接口幂等性会有很严重的后果：  
支付接口，重复支付会导致多次扣钱  
订单接口，同一个订单可能会多次创建。

接口幂等性就是用户对于同一操作发起的一次请求或者多次请求的结果是一致的，不会因为多次点击而产生了副作用。举个最简单的例子，那就是支付，用户购买商品后支付，支付扣款成功，但是返回结果的时候网络异常，此时钱已经扣了，用户再次点击按钮，此时会进行第二次扣款，返回结果成功，用户查询余额返发现多扣钱了，流水记录也变成了两条．．．,这就没有保证接口的幂等性

**幂等是系统的接口对外的一种承诺(而不是实现)， 承诺只要调用接口成功， 外部多次调用对系统的影响是一致的。这里的多次调用强调是指接口一致，参数一致的情况。**

**为什么需要幂等:**

**幂等是为了保证重试机制不会带来数据重复，数据不一致等异常情况。**

**不管在单机还是分布式系统中，都存在因为网络抖动无法收到成功应答；重试补偿机制；用户无意识点击发起多次请求（业务意义上其实是同一条数据记录）等情况带来的相同参数多次调用相同接口的情况。**

**如何做到幂等**

**1、有些接口本身就是幂等的，例如查询接口，当然这是在查询数据没有做变更的情况下；删除操作一般会带有待删除数据的唯一标识，最终结果都是删除，也是幂等的。**

**2、对于创建新数据的情况，采用唯一业务单号，业务上的唯一条件约束。**

**3、在执行更新操作的时候，可以先查再决定是否更新，但这因为是非原子操作，所以在分布式系统中会存在问题。**

**4、对外提供服务的接口如何做到幂等**

**对外提供接口为了支持幂等调用，接口有两个字段必须传，一个是来源source，一个是来源方序列号seq，这个两个字段在己方系统里面作为唯一标识符，后续在己方系统中使用以上的方法来保证幂等。**

##### **概念**

接口的幂等是系统对外的一种承诺,承诺只要调用成功,外部多次调用对系统的影响是一致的.

##### **为什么要是有幂等**

在网络波动时用户无意识多次点击会发送相同的请求,在重试机制下要保证数据的一致,不能出现重复数据,也不能多次调用相同接口.

##### **如何实现幂等**

有些接口本身就是幂等的比如查询接口,实在没有新增和修改时,删除也是幂等的他的最终结果也是删除.

在新增数据时采用唯一的单号,唯一的约束

### **1.1什么是接口幂等性**

接口幂等性就是用户对于同一操作发起的一次请求或者多次请求的结果是一致的，不会因为多次点击而产生了副作用。举个最简单的例子，那就是支付，用户购买商品后支付，支付扣款成功，但是返回结果的时候网络异常，此时钱已经扣了，用户再次点击按钮，此时会进行第二次扣款，返回结果成功，用户查询余额返发现多扣钱了，流水记录也变成了两条．．．,这就没有保证接口的幂等性

**spring mvc的执行流程?**

1.用户发送请求至前端控制器DispatcherServlet  
2.DispatcherServlet收到请求调用处理器映射器HandlerMapping。  
3.处理器映射器根据请求url找到具体的处理器，生成处理器执行链HandlerExecutionChain(包括处理器对象和 处理器拦截器)一并返回给DispatcherServlet。  
4.DispatcherServlet根据处理器Handler获取处理器适配器HandlerAdapter执行HandlerAdapter处理一系列 的操作，如：参数封装，数据格式转换，数据验证等操作  
5.执行处理器Handler(Controller，也叫页面控制器)。  
6.Handler执行完成返回ModelAndView  
7.HandlerAdapter将Handler执行结果ModelAndView返回到DispatcherServlet  
8.DispatcherServlet将ModelAndView传给ViewReslover视图解析器  
9.ViewReslover解析后返回具体View  
10.DispatcherServlet对View进行渲染视图（即将模型数据model填充至视图中）。  
11.DispatcherServlet响应用户。

**spring mvc是什么时候把请求的参数映射到实体类的(其实这道题考的还是sringmvc的运行**

**原理,只不过是更细节的运行原理)?**

**mybatis的执行器有哪些?**

#### **Mybatis有三种基本的Executor执行器:**

#### ****SimpleExecutor、ReuseExecutor、BatchExecutor。****

**SimpleExecutor：**每执行一次update或select，就开启一个Statement对象，用完立刻关闭Statement对象。

**ReuseExecutor：**执行update或select，以sql作为key查找Statement对象，存在就使用，不存在就创建，用完后，不关闭Statement对象，而是放置于Map内，供下一次使用。简言之，就是重复使用Statement对象。

**BatchExecutor：**执行update（没有select，JDBC批处理不支持select），将所有sql都添加到批处理中（addBatch()），等待统一执行（executeBatch()），它缓存了多个Statement对象，每个Statement对象都是addBatch()完毕后，等待逐一执行executeBatch()批处理。与JDBC批处理相同。

作用范围：Executor的这些特点，都严格限制在SqlSession生命周期范围内。

Mybatis中如何指定使用哪一种Executor执行器？

答：在Mybatis配置文件中，可以指定默认的ExecutorType执行器类型，也可以手动给DefaultSqlSessionFactory的创建SqlSession的方法传递ExecutorType类型参数。

**mybatis运行原理?**

在MyBatis运行开始时需要先通过Resources加载全局配置文件.下面需要实例化SqlSessionFactoryBuilder构建器.帮助SqlSessionFactory接口实现类DefaultSqlSessionFactory.

在实例化DefaultSqlSessionFactory之前需要先创建XmlConfigBuilder解析全局配置文件流,并把解析结果存放在Configuration中.之后把Configuratin传递给DefaultSqlSessionFactory.到此SqlSessionFactory工厂创建成功.

由SqlSessionFactory工厂创建SqlSession.

每次创建SqlSession时,都需要由TransactionFactory创建Transaction对象,同时还需要创建SqlSession的执行器Excutor,最后实例化DefaultSqlSession,传递给SqlSession接口.

根据项目需求使用SqlSession接口中的API完成具体的事务操作.

如果事务执行失败,需要进行rollback回滚事务.

如果事务执行成功提交给数据库.关闭SqlSession

到此就是MyBatis的运行原理

**mybatis如何自己实现一个分页插件(这道题考的其实还是mybatis的执行器和mybatis运行原理)?**

Mybatis使用RowBounds对象进行分页，它是针对ResultSet结果集执行的内存分页，而非物理分页。可以在sql内直接书写带有物理分页的参数来完成物理分页功能，也可以使用分页插件来完成物理分页。

分页插件的基本原理是使用Mybatis提供的插件接口，实现自定义插件，在插件的拦截方法内拦截待执行的sql，然后重写sql，根据dialect方言，添加对应的物理分页语句和物理分页参数。

**mybatis接口方法和xml标签如何映射?**

1.注册驱动！

2.建立connection！

3.创建操作语句statement！

4.执行statement！

5.封装结果resultset！

Mybatis可以把Mapper.xml文件直接映射到对应的接口，调用接口方法会自动去Mapper.xml文件中找到对应的标签，

这个功能就是利用java的动态代理在binding包中实现的

一、注册Mapper

在初始化时会把获取到的Mapper接口注册到MapperRegistry，注册的时候创建一个Mapper代理工厂，

这个工厂通过JDK的代理创建一个执行对象，创建代理需要的InvocationHandler为MapperProxy

二、获取接口对象

从knownMappers中根据接口类型取出对应的代理创建工厂，用该工厂创建代理。

三、调用接口方法

调用代理方法会进入到MapperProxy的public Object invoke(Object proxy, Method method, Object[] args)方法

四丶最终执行sql会进入到MapperMethod中execute方法

mybatis是通过xml文件中mapper标签内的namespace属性来获得接口的全限定名来映射到xml

mapper.xml中的<mappr>标签配置了namespace属性,值为`接口的全限定名`,将接口与mapper.xml相关联,

而接口中的方法和xml中的sql语句则通过id连接,xml中标签的id对应接口中的方法名

**线程和进程的关系?**

1、进程：进程在操作系统中可以独立运行，作为资源分配的基本单位。表示运行中的程序。

2、线程：线程是进程中的一个实例，作为系统调度和分派的基本单位。是进程中的一段序列，能够完成进程中的一个功能。

3、进程和线程的区别：

（1）同一个进程可以包含多个线程，一个进程中至少包含一个线程，一个线程只能存在于一个进程中。

（2）同一个进程下的所有线程能够共享该进程下的资源。（系统运行时会为每个进程分配不同的内存区域，

但不会为线程分配内存。线程只能共享它所属进程的资源。）

（3）进程结束后，该进程下的所有线程将销毁，而一个线程的结束不会影响同一进程下的其他线程。

（4）线程是轻量级的进程，它的创建和销毁所需要的时间比进程小得多，所有操作系统的执行功能都是通过创建线程去完成的。

（5）线程在执行时是同步和互斥的，因为他们共享同一个进程下的资源。

（6）在操作系统中，进程是拥有系统资源的独立单元，它可以拥有自己的资源。

一般而言，线程不能拥有自己的资源，但是它能够访问其隶属进程的资源。

**线程有哪些状态?**

线程通常都有五种状态，创建、就绪、运行、阻塞和死亡。

创建状态：在生成线程对象，并没有调用该对象的start方法，这是线程处于创建状态。

就绪状态：当调用了线程对象的start方法之后，该线程就进入了就绪状态，

但是此时线程调度程序还没有把该线程设置为当前线程，此时处于就绪状态：

在线程运行之后，从等待或者睡眠中回来之后，也会处于就绪状态。

运行状态：线程调度程序将处于就绪状态的线程设置为当前线程，此时线程就进入了运行状态，

开始运行run函数当中的-代码。

阻塞状态：线程正在运行的时候，被暂停，通常是为了等待某个时间的发生(比如说某项资源就绪)之后再继续运行。

sleep,suspend，wait等方法都可以导致线程阻塞。

死亡状态：如果一个线程的run方法执行结束或者调用stop方法后，该线程就会死亡。

**线程的常用方法以及相似方法之间的区别(例如:sleep和yeild之类的)?**

Thread.currentThead（）：获取当前线程对象

getPriority（）：获取当前线程的优先级

setPriority（）：设置当前线程的优先级

注意：线程优先级高，被CPU调度的概率大，但不代表一定会运行，还有小概率运行优先级低的线程。

isAlive（）：判断线程是否处于活动状态 （线程调用start后，即处于活动状态）

join（）：调用join方法的线程强制执行，其他线程处于阻塞状态，等该线程执行完后，其他线程再执行。有可能被外界中断产生InterruptedException 中断异常。

sleep（）：在指定的毫秒数内让当前正在执行的线程休眠（暂停执行）。休眠的线程进入阻塞状态。

yield（）：调用yield方法的线程，会礼让其他线程先运行。（大概率其他线程先运行，小概率自己还会运行）

interrupt()：中断线程

wait（）：导致线程等待，进入堵塞状态。该方法要在同步方法或者同步代码块中才使用的

notify（）：唤醒当前线程，进入运行状态。该方法要在同步方法或者同步代码块中才使用的

notifyAll（）：唤醒所有等待的线程。该方法要在同步方法或者同步代码块中才使用的

线程睡眠：sleep

说明:让当前线程暂停一段时间，让出处理器资源，进入阻塞状态。

当睡眠时间达到，该进程将会重新进入就绪状态。

线程让步：yield

说明:只有其他线程优先级与当前线程优先级相同或者更高，才会获得执行机会。

区别:

1.sleep()方法暂停线程不会理会优先级，而yield只会给优先级相同或者更高的线程。

2.sleep()方法让线程进入阻塞状态，而yield()方法让线程再次进入就绪状态，不会阻塞。

可能该执行完yield()方法，立即再次获得处理器资源被执行。

D

3.sleep()方法声明抛出异常InterruptedException异常，所 以调用sleep()方法时要么捕获该异常，

要么显式声明抛出异常。而yield()方法则没有声明抛出任何异常。

4.sleep()比yield()有更好的移植性，通常不建议使用yield()方法来控制并发线程的执行。

**什么是线程安全?**

线程安全就是多线程访问时，采用了加锁机制，当一个线程访问该类的某个数据时，进行保护，其他线程不能进行访问直到该线程读取完，其他线程才可使用。不会出现数据不一致或者数据污染

**sychronazed监听的是this可以解决线程安全的问题吗?**

synchronized监听的是this(即对象的实例),

如果创建了一个类的两个实例对象，这时存在两把锁，

创建了两个线程，一个线程可以访问这个类中的synchronized方法，另一个线程也可以访问这个类中的synchronized方法，这时如果没有共享的变量,是线程安全的,如果有同一个共享变量，就会造成线程不安全

**new String("abc")会创建几个对象,说一下原理(JVM内存模型)?**

1个或2个

解释:

当JVM遇到上述代码时，会先检索常量池中是否存在“abc”

1.如果不存在“abc”这个字符串，则会先在常量池中创建这个一个字符串。然后再执行new操作，

会在堆内存中创建一个存储“abc”的String对象，对象的引用赋值给str2。此过程创建了2个对象。

2.如果检索常量池时发现已经存在了对应的字符串，那么只会在堆内创建一个新的String对象，此过程只创建了1个对象。

**锁的膨胀(升级)过程?**

偏向锁：只有一个线程进入临界区；

轻量级锁：多个线程交替进入临界区;

重量级锁：多个线程同时进入临界区；

偏向锁，轻量级锁都是乐观锁，重量级锁是悲观锁。

无锁->偏向锁->轻量级锁->重量级锁(锁升级的过程不可逆)

偏向锁，轻量级锁都是乐观锁，重量级锁是悲观锁。

一个对象刚开始实例化，没有任何线程来访问它的时候，它是可偏向的，意味着，它现在认为只可能有一个线程来访问它，所以当第一个线程来访问它的时候，它会偏向这个线程，此时，对象持有偏向锁。偏向第一个线程，这个线程在修改对象头成为偏向锁的时候使用CAS操作，并将对象头中的ThreadID改成自己的ID，之后再次访问这个对象时，只需要对比ID，不需要再使用CAS在进行操作。

一旦有第二个线程访问这个对象，因为偏向锁不会主动释放，所以第二个线程可以看到对象是偏向状态,这时表明在这个对象上已经存在竞争了，检查原来持有该对象锁的线程是否依然存活，如果挂了，则可以将对象变为无锁状态，然后重新偏向新的线程，如果原来的线程依然存活，则马上执行那个线程的操作栈，检查该对象的使用情况，

如果仍然需要持有偏向锁，则偏向锁升级为轻量级锁，（ 偏向锁就是这个时候升级为轻量级锁的）。

如果不存在使用了，则可以将对象恢复成无锁状态，然后重新偏向。

轻量级锁认为竞争存在，但是竞争的程度很轻，一般两个线程对于同一个锁的操作都会错开，或者说稍微等待一下（自旋），

另一个线程就会释放锁。 但是当自旋超过一定的次数，或者一个线程在持有锁，一个在自旋，又有第三个来访时，轻量级锁膨胀为重量级锁，

重量级锁使除了拥有锁的线程以外的线程都阻塞，防止CPU空转

无锁（当前对象还没被使用）

偏向锁（当一个线程获取了锁就设置一个锁标识，这个标识存在对象头中,当有俩个以上的线程竞争锁的时候升级为轻量级锁）

轻量级锁(当一个线程自旋10次还是没有获取锁就升级为——》重量级锁）

**如何创建自定义注解?**

1. @Documented – 表示使用该注解的元素应被javadoc或类似工具文档化，它应用于类型声明，类型声明的注解会影响客户端对注解元素的使用。如果一个类型声明添加了Documented注解，那么它的注解会成为被注解元素的公共API的一部分。

　2. @Target – 用于描述修饰对象的范围，范围取值于ElementType这个枚举类:

TYPE(类、接口（包括注释类型）或枚举声明)、type

METHOD((方法声明)、 method

CONSTRUCTOR(构造方法声明)、constructor

FIELD(字段声明)、fifld

ANNOTATION\_TYPE(注释类型声明)、annotation\_type

LOCAL\_VARIABLE(局部变量声明)、local\_variable

PACKAGE(包声明)、package

PARAMETER(参数声明)。parameter

如果Target元注解不存在，那么该注解就可以使用在任何程序元素之上。

　　3. @Inherited – 表示一个注解类型会被自动继承，如果用户在类声明的时候查询注解类型，同时类声明中也没有这个类型的注解，那么注解类型会自动查询该类的父类，这个过程将会不停地重复，直到该类型的注解被找到为止，或是到达类结构的顶层（Object）。

　　4. @Retention – 表示注解类型保留时间的长短，范围取值于RetentionPolicy这个枚举类：

　　（1）CLASS 编译器将把注释记录在类文件中，但在运行时 VM 不需要保留注释

　　（2）RUNTIME  编译器将把注释记录在类文件中，在运行时 VM 将保留注释，因此可以反射性地读取

　　（3）SOURCE 编译器要丢弃的注释

**说一下多态(如果只强调重写重载的话那基本就凉了)?**

多态:

在java中 多态是指一种事务的多种状态

当继承关系发生后 我们用父类的引用指向子类的对象

当类实现接口 我们用接口的引用指向实现类对象

子级对象可以调用父级的部分特性 但是父级调用不到子级的特性

多态还包含重载和重写

多态:

多态是建立在封装和继承基础之上；也可以理解为子类就是父类（比如玫瑰是花，电视是电子产品，但是花不单指玫瑰，电子产品不单指电视），

因此多态的意思就是：父类的引用可以指向子类的对象；但子类的引用不能指向父类的对象

多态也取决于它是早绑定还是晚绑定，方法重载是早绑定，处于编译期间，在编译的时候java就知道他的参数和类型等；

而多态是晚绑定，就是在运行的时候才知道，而编译的时候是不知道的。

好处：提高了代码的复用性

提高了代码的维护性

提高了代码的扩展性

弊端

不能访问子类特有功能

重载:

重载是发生在同一类中,且方法名相同.

参数列表不同: 数据类型 个数不同 顺序不同

与返回值无关,与修饰符无关,

调用重载的方法由实际参数来决定的

重载是多态的一种体现

重写:

写是发生在继承关系中,当父类的方法无法满足子类的需求时,就在子类中重写父类的方法,重写也叫方法的覆盖

子类重写父类方法时,子类的返回值,方法名,参数列表必须完全一致,方法体不一致

子类不能重写父类中priavte修饰的和static修饰的方法

子类方法的访问权限不能小于父类;

子类方法不能声明比父类更大的异常.

重写也是java多态的一种体现,重载也是

**treeset 如何去重?**

可以实现排序及去重：如果compareTo返回0，说明是重复的，返回的是自己的某个属性和另一个对象的某个属性的差值,如果是负数，则往前面排，如果是正数，往后面排；reeset在排序中包含了去重,因为重复的key不会被添加到map中

TreeSet底层是基于TreeMap实现的,而TreeMap底层的数据结构是一个红黑树.TreeMap内置默认的Comparator比较器

当我们向TreeSet添加元素(add)时,会调用TreeMap的put方法,首先会通过Comparator比较器比较key的值在树中是否存在，若存在则替换掉，若不存在则插入其中。

**Arraylist RandomAccess接口的作用?**

RandomAccess接口这个空架子的存在，是为了能够更好地判断集合是否ArrayList或者LinkedList，从而能够更好选择更优的遍历方式，提高性能！

**序列化反序列化的作用?**

#### 1、序列化和反序列化的定义：

#### 1、序列化和反序列化的定义：

#### (1)Java序列化就是指把Java对象转换为字节序列的过程

#### Java反序列化就是指把字节序列恢复为Java对象的过程。

#### (2)序列化最重要的作用：在传递和保存对象时.保证对象的完整性和可传递性。对象转换为有序字节流,以便在网络上传输或者保存在本地文件中。

#### 反序列化的最重要的作用：根据字节流中保存的对象状态及描述信息，通过反序列化重建对象。

#### 总结：核心作用就是对象状态的保存和重建。（整个过程核心点就是字节流中所保存的对象状态及描述信息）

#### 2、json/xml的数据传递：

#### 在数据传输(也可称为网络传输)前，先通过序列化工具类将Java对象序列化为json/xml文件。

#### 在数据传输(也可称为网络传输)后，再将json/xml文件反序列化为对应语言的对象

#### 3、序列化优点：

#### ①将对象转为字节流存储到硬盘上，当JVM停机的话，字节流还会在硬盘上默默等待，等待下一次JVM的启动，把序列化的对象，通过反序列化为原来的对象，并且序列化的二进制序列能够减少存储空间（永久性保存对象）。

#### ②序列化成字节流形式的对象可以进行网络传输(二进制形式)，方便了网络传输。

#### ③通过序列化可以在进程间传递对象。

#### 4、序列化算法需要做的事：

#### ① 将对象实例相关的类元数据输出。

#### ② 递归地输出类的超类描述直到不再有超类。

#### ③ 类元数据输出完毕后，从最顶端的超类开始输出对象实例的实际数据值。

#### ④ 从上至下递归输出实例的数据。

**spring 如何为我们创建对象(生命周期)?**

Spring对Bean进行实例化（相当于程序中的new Xx()）

Spring将值和Bean的引用注入进Bean对应的属性中

如果Bean实现了BeanNameAware接口，Spring将Bean的ID传递给setBeanName()方法  
**（实现BeanNameAware清主要是为了通过Bean的引用来获得Bean的ID，一般业务中是很少有用到Bean的ID的**）

如果Bean实现了BeanFactoryAware接口，Spring将调用setBeanDactory(BeanFactory bf)方法并把BeanFactory容器实例作为参数传入。  
**（实现BeanFactoryAware 主要目的是为了获取Spring容器，如Bean通过Spring容器发布事件等）**

如果Bean实现了ApplicationContextAwaer接口，Spring容器将调用setApplicationContext(ApplicationContext ctx)方法，把y应用上下文作为参数传入.  
**(作用与BeanFactory类似都是为了获取Spring容器，不同的是Spring容器在调用setApplicationContext方法时会把它自己作为setApplicationContext 的参数传入，而Spring容器在调用setBeanDactory前需要程序员自己指定（注入）setBeanDactory里的参数BeanFactory )**

如果Bean实现了BeanPostProcess接口，Spring将调用它们的postProcessBeforeInitialization（预初始化）方法  
**（作用是在Bean实例创建成功后对进行增强处理，如对Bean进行修改，增加某个功能）**

如果Bean实现了InitializingBean接口，Spring将调用它们的afterPropertiesSet方法，作用与在配置文件中对Bean使用init-method声明初始化的作用一样，都是在Bean的全部属性设置成功后执行的初始化方法。

**（作用与6的一样，只不过6是在Bean初始化前执行的，而这个是在Bean初始化后执行的，时机不同 )**

经过以上的工作后，Bean将一直驻留在应用上下文中给应用使用，直到应用上下文被销毁

如果Bean实现了DispostbleBean接口，Spring将调用它的destory方法，作用与在配置文件中对Bean使用destory-method属性的作用一样，都是在Bean实例销毁前执行的方法。

**spring bean的作用域?**

singleton：SpringIOC容器只会创建该Bean的唯一实例；

prototype：每次请求都创建一个实例；

request：每次HTTP请求都会产生一个新的bean。需要注意的是，该作用域仅在基于Web的Spring ApplicationContext情形下有效，以下的session和global Session也是如此；

Session：每次会话创建一个实例；

global session：全局HttpSession中，容器会返回该bean的同一个实例。

**spring创建出来的对象(单例对象)是线程安全的吗?**

不是线程安全的。但是单例bean也不表示就一定线程不安全，这要看bean中有没有多线程共享变量。如果都是线程私有数据，那也是线程安全的，反之就是不安全的！

**如何保证spring的bean线程安全的呢?**

Spring中的Bean默认是单例模式的，框架并没有对bean进行多线程的封装处理。  
　　实际上大部分时间Bean是无状态的（比如Dao） 所以说在某种程度上来说Bean其实是安全的。  
　　但是如果Bean是有状态的 那就需要开发人员自己来进行线程安全的保证，最简单的办法就是改变bean的作用域 把 "singleton"改为‘prototype’这样每次请求Bean就相当于是 new Bean() 这样就可以保证线程的安全了。

有状态就是有数据存储功能  
无状态就是不会保存数据

在一般情况下，只有无状态的Bean才可以在多线程环境下共享，在Spring中，绝大部分Bean都可以声明为singleton作用域，因为Spring对一些Bean中非线程安全状态采用ThreadLocal进行处理，解决线程安全问题。

**threadlocal底层实现(这道题有可能会延伸出来 强弱软虚四种引用类型)?**

ThreadLocal是一个解决线程并发问题的一个类，用于创建线程的本地变量，我们知道一个对象的所有线程会共享它的全局变量，所以这些变量不是线程安全的，我们可以使用同步技术。但是当我们不想使用同步的时候，我们可以选择ThreadLocal变量。

每个线程都会拥有他们自己的Thread变量，他们可以使用get/set方法去获取他们的默认值或者在线程内部改变他们的值。ThreadLocal实例通常是希望他们同线程状态关联起来是private static属性。

　　　底层实现主要是存有一个map，以线程作为key，泛型作为value，可以理解为线程级别的缓存。每一个线程都会获得一个单独的

map。

强引用：不会被垃圾回收，宁肯抛出outofMemoryError错误也不会回收

软引用SoftReference：内存不足的时候会被优先回收，回收之后还是内存不足才会抛出outofmemory异常。适合用于缓存对象。

弱引用WeakReference：一个对象如果只有弱引用，那么就会被垃圾回收。常常用于hash表中，因为hash表允许存储java对象的引 用，如果用强引用的话只要hash表存在，那么键值对象就不会回收。如果某个hash表生命周期很长，键值又 多会很消耗内存。

虚引用： 虚引用是每次垃圾回收的时候都会被回收，通过虚引用的get方法永远获取到的数据为null，因此也被成为幽灵引用。

虚引用主要用于检测对象是否已经从内存中删除。

）

1.强： 我们常用的一个引用 = new 对象（）就是强引用.

只要有引用指向它，就不会被回收。

2.软： 内存空间不够时，软引用会被回收。

3.弱： 只要触发GC就会被回收。

4.虚： 虚引用是给写JVM虚拟机的人管理堆外内存用的，普通开发人员基本用不到

**spring 是如何解决循环依赖的?**

Spring是通过递归的方式获取目标bean及其所依赖的bean的；

Spring实例化一个bean的时候，是分两步进行的，首先实例化目标bean，然后为其注入属性。

结合这两点，也就是说，Spring在实例化一个bean的时候，是首先递归的实例化其所依赖的所有bean，直到某个bean没有依赖其他bean，此时就会将该实例返回，然后反递归的将获取到的bean设置为各个上层bean的属性的。

我们初始化一个Bean时，先调用Bean的构造方法，然后这个对象就在内存中存在了，

然后保存这个对象，当循环依赖产生时，直接拿到之前保存的对象，于是循环依赖就被终止了，依赖注入也就顺利完成了。

Spring通过三级缓存来解决循环依赖

为什么需要三级缓存，总而言之考虑到的是扩展性，预留扩展接口

循环依赖和AOP结合的时候，也能保证依赖的是代理对象，是通过SmartInstantiationAwareBeanPostProcessor的getEarlyBeanReference接口回调来实现的

**mysql事物的隔离级别?**

mysql默认的隔离级别是Repeatable read

Serializable (串行化)：可避免脏读、不可重复读、幻读的发生。

Repeatable read (可重复读)：可避免脏读、不可重复读的发生。

Read committed (读已提交)：可避免脏读的发生。

Read uncommitted (读未提交)：最低级别，任何情况都无法保证。

**什么是赃读 幻读 重复读?**

**1. 脏读**：脏读就是指当一个事务正在访问数据，并且对数据进行了修改，而这种修改还没有提交到数据库中，这时，另外一个事务也访问这个数据，然后使用了这个数据。

**2. 不可重复读**：是指在一个事务内，多次读同一数据。在这个事务还没有结束时，另外一个事务也访问该同一数据。那么，在第一个事务中的两 次读数据之间，由于第二个事务的修改，那么第一个事务两次读到的的数据可能是不一样的。这样就发生了在一个事务内两次读到的数据是不一样的，因此称为是不 可重复读。例如，一个编辑人员两次读取同一文档，但在两次读取之间，作者重写了该文档。当编辑人员第二次读取文档时，文档已更改。原始读取不可重复。如果 只有在作者全部完成编写后编辑人员才可以读取文档，则可以避免该问题。

**3. 幻读**: 是指当事务不是独立执行时发生的一种现象，例如第一个事务对一个表中的数据进行了修改，这种修改涉及到表中的全部数据行。 同时，第二个事务也修改这个表中的数据，这种修改是向表中插入一行新数据。那么，以后就会发生操作第一个事务的用户发现表中还有没有修改的数据行，就好象 发生了幻觉一样。例如，一个编辑人员更改作者提交的文档，但当生产部门将其更改内容合并到该文档的主复本时，发现作者已将未编辑的新材料添加到该文档中。 如果在编辑人员和生产部门完成对原始文档的处理之前，任何人都不能将新材料添加到文档中，则可以避免该问题。

脏读：一个事务读到了另外一个事务未提交的数据。

幻读：一个事务读到了另外一个事务已经提交的insert数据，导致多次查询结果不一致。

不可重复读：一个事务读到了另外一个事务已经提交的update数据，导致多次查询结果不一致。

**哪个隔离级别会产生赃读?**

Read uncommitted (读未提交)：最低级别，任何情况都无法保证。

以上是SQL-92标准中定义的四种隔离级别。在MySQL中，默认的隔离级别是REPEATABLE-READ（可重复读），并且解决了幻读问题。简单的来说，mysql的默认隔离级别解决了脏读、幻读、不可重复读问题。

**哪个隔离级别会产生幻读?**

Repeatable read (可重复读)：可避免脏读、不可重复读的发生。

Read committed (读已提交)：可避免脏读的发生。

Read uncommitted (读未提交)：最低级别，任何情况都无法保证。

**哪个会产生重复读?**

Read committed (读已提交)：可避免脏读的发生。

Read uncommitted (读未提交)：最低级别，任何情况都无法保证。

**查询语句的执行顺序?**

1. 从from开始
2. where 阶段
3. group by分组
4. having 筛选
5. select 查询挑拣计算列
6. order by排序显示

**说一下你对分布式的理解?**

分布式系统（distributed system） 是建立在网络之上的软件系统。

内聚性：是指每一个数据库分布节点高度自治，有本地的数据库管理系统。

透明性：是指每一个数据库分布节点对用户的应用来说都是透明的，看不出是本地还是远程。

在一个分布式系统中，一组独立的计算机展现给用户的是一个统一

的整体，就好像是一个系统似的。分布式系统作为一个整体对用户提供服务，而整个系统的内部的协作对用户来说是透明的，用户就像是

+

指使用一个Mysql一样，如：分布式Mysql中间件Mycat ，来处理大并发大数据量的构架。

* **一、分布式系统的特点**

分布式系统最大的特点是可扩展性，它能够适应需求变化而扩展。企业级应用需求经常随时间而不断变化，这也对企业级应用平台提出了很高的要求。企业级应用平台必须要能适应需求的变化，即具有可扩展性。比如移动互联网2C应用，随着互联网企业的业务规模不断增大，业务变得越来越复杂，并发用户请求越来越多，要处理的数据也越来越多，这个时候企业级应用平台必须能够适应这些变化，支持高并发访问和海量数据处理。分布式系统有良好的可扩展性，可以通过增加服务器数量来增强分布式系统整体的处理能力，以应对企业的业务增长带来的计算需求。

分布式系统的核心理念是让多台服务器协同工作，完成单台服务器无法处理的任务，尤其是高并发或者大数据量的任务。分布式系统由独立的服务器通过网络松散耦合组成的。每个服务器都是一台独立的PC机，服务器之间通过内部网络连接，内部网络速度一般比较快。因为分布式集群里的服务器是通过内部网络松散耦合，各节点之间的通讯有一定的网络开销，因此分布式系统在设计上尽可能减少节点间通讯。此外，因为网络传输瓶颈，单个节点的性能高低对分布式系统整体性能影响不大。比如，对分布式应用来说，采用不同编程语言开发带来的单个应用服务的性能差异，跟网络开销比起来都可以忽略不计。因此，分布式系统每个节点一般不采用高性能的服务器，而是性能相对一般的普通PC服务器。提升分布式系统的整体性能是要通过横向扩展（增加更多的服务器），而不是纵向扩展（提升每个节点的服务器性能）。

* 二、分布式系统设计理念

1. 分布式系统对服务器硬件要求很低
2. 分布式系统强调横向可扩展性
3. 分布式系统不允许单点失效
4. 分布式系统尽可能减少节点间通讯开销
5. 分布式系统应用服务最好做成无状态的

**说一下你对微服务的理解?**

微服务是独立的服务共同组成整个系统，其次是单个部署每个跑在自己的进程中，每个服务为独立的业务开发，是分布式管理的，非常强调隔离性，1.分布式组成的系统，2.按照业务而不是按照技术来划分组织，3.他是有生命的产品，而不是项目，4.他是强服务个体，弱通信 5.自动化运维，6.具有高度的容错性，7.可以快速的液化和迭代，

**你们都有哪些微服务,服务之间是怎么划分的?**

微服务的特性：小型的、独立的、可运行的应用程序，组合成一个大的功能，这样做可以使服务的复用性更高，那直接就可以拿过用，并且每个功能都被称为一项服务，可以单独构建和部署，

松耦合   
就是服务与服务之间的影响要尽量减少，想象一下如果如果服务之间做到了松耦合，那么就意味着修改一个服务就不需要修改另一个服务。这一点对与实现微服务来说是很重要的。

高内聚  
 我们需要将相关的行为收集在一起，避免修改一个功能需要修改多个服务才能实现。

限界上下文  
任何一个给定的领域都包含多个界限上下文，每个界限上下文都中的东西都分为两个部分，一部分不需要与外部通信，另外一部分则需要、每个上下文都有明确的接口，该接口决定了它会暴露哪些模型给其他上下文。

**你们用到了cloud的哪些组件?**  
Eureka：服务注册于发现。

· Feign：基于动态代理机制，根据注解和选择的机器，拼接请求 url 地址，发起请求。

· Ribbon(瑞本no)：实现负载均衡，从一个服务的多台机器中选择一台。

· Hystrix(黑死 chui 可 s)：提供线程池，不同的服务走不同的线程池，实现了不同服务调用的隔离，避免了服务雪崩的问题。

Zuul：网关管理，由 Zuul 网关转发请求给对应的服务

第一个是既然是众多的服务 第一个服务就涉及到一个叫做注册中心 主要承担作用就是吧各种服务注册到这里边去,以后要调用直接从这里边拿服务

第二个既然有服务那同一个服务可能会注册多个所有第二个组件也就叫做 ribbon或者feign 他是做负载均衡调用的 主要的作用就是从注册中心拿到一堆的服务列表 比如说我要拿到一个用户服务 拿到他有可能是三个或者五个 这个时候就要按照一定的负载均衡策略 然后完成调用 这个操作就是交给ribbon或者feign去做的

第三个除了做正常的调用以外 我们还有可能做以下的事情 比如说在调这个服务的时候 这个服务突然挂掉了这个时候为了保证 服务的一个健壮性 就要用到另一个组件 hystrix 他主要就是做熔断和降级以及限流用的

第四个我们众多服务当中他可能有一些配置 而这些配置如果把他分散到不同的项目当中 不同的微服务当中 管理起来是不好管理的 所以要有一个统一的配置中心 configure

第五个就是网关 就像我们有n多个微服务 而这些服务最终是要暴露给前端调用的 而前端如果是一个服务就有一个地址 他是非常不好管理的 这个时候我们让他统一地址 这是第一个作用 我们还可以对要访问的微服务做统一鉴权等等的操作 而这些操作统统都交给微服务的网关去处理 也就是用zuul来进行操作 这就是springcloud的五大组件  
Spring Cloud Conifg  
Eureka  
Hystrix  
Ribbon  
Zuul  
Feign  
Spring Cloud Stream  
Spring Cloud Sleuth  
Consul  
Spring Cloud Gateway

**ribbon的负载均衡策略？**

| **策略类** | **命名** | **描述** |
| --- | --- | --- |
| RandomRule | 随机策略 | 随机选择server |
| RoundRobinRule | 轮询策略 | 按照顺序选择server（ribbon默认策略） |
| RetryRule | 重试策略 | 在一个配置时间段内，当选择server不成功，则一直尝试选择一个可用的server |
| BestAvailableRule | 最低并发策略 | 逐个考察server，如果server断路器打开，则忽略，再选择其中并发链接最低的server |
| AvailabilityFilteringRule | 可用过滤策略 | 过滤掉一直失败并被标记为circuit tripped的server，过滤掉那些高并发链接的server（active connections超过配置的阈值） |
| ResponseTimeWeightedRule | 响应时间加权重策略 | 根据server的响应时间分配权重，响应时间越长，权重越低，被选择到的概率也就越低。响应时间越短，权重越高，被选中的概率越高，这个策略很贴切，综合了各种因素，比如：网络，磁盘，io等，都直接影响响应时间 |
| ZoneAvoidanceRule | 区域权重策略 | 综合判断server所在区域的性能，和server的可用性，  轮询选择server并且判断一个AWS Zone的运行性能是否可用，  剔除不可用的Zone中的所有server |

**Eureka底层实现?**

服务启动后向Eureka注册，Eureka Server会将注册信息向其他Eureka Server进行同步，当服务消费者要调用服务提供者，则向服务注册中心获取服务提供者地址，然后会将服务提供者地址缓存在本地，下次再调用时，则直接从本地缓存中取，完成一次调用。

**erurka挂了怎么办?**

或者通过ribbon负载均衡多部署几个服务

如果只有一个服务就只能重启服务

如果经常挂掉可能是代码有问题

**谈一下你对spring的理解?**

Spring是一个基于IOC和AOP结构的轻量级框架,也是最受欢迎的企业级Java应用程序开发框架。从大小与开销两方面而言Spring都是轻量的。（完整的spring框架也可以在一个大小只有1M的jar文件里发布，并且spring所需的处理开销也是微不足道的。）此外spring是非入侵式的（典型的：spring应用中的对象不依赖于spring的特定类）。

spring通过一种称作控制反转（IOC）的技术促进了低耦合。当应用了IOC，一个对象依赖的其他对象会通过被动的方式传递进来，而不是这个对象自己创建或者查找依赖对象。

Spring支持面向切面（AOP）的编程，并且把应用业务逻辑和系统服务分开

spring包含并管理应用对象的配置和生命周期，在这个意义上它是一种容器。你可以配置你的每个bean如何被创建（基于一个可配置的原型，你的bean可以创建一个单独的实例或者每次需要都是生成一个新的实例）以及他们是如何互相关联的。

spring可以将简单的组件配置、组合成为复杂的应用。在spring中，应用对象被声明式的组合（bean都在一个xml文件里。)spring也提供了很多基础功能（事务管理、持久化框架集成等），将应用逻辑的开发留给开发者。

**动态代理/以及区别?**

**JDK动态代理：**利用反射机制生成一个实现代理接口的匿名类，在调用具体方法前调用InvokeHandler来处理。  
**CGlib动态代理：**利用ASM（开源的Java字节码编辑库，操作字节码）开源包，将代理对象类的class文件加载进来，通过修改其字节码生成子类来处理。

**区别：**JDK代理只能对实现接口的类生成代理；CGlib是针对类实现代理，对指定的类生成一个子类，并覆盖其中的方法，这种通过继承类的实现方式，不能代理final修饰的类。

**aspectj？**

* AspectJ 是一个代码生成工具（Code Generator）。
* AspectJ 语法就是用来定义代码生成规则的语法。
* AspectJ 有自己的语法编译工具，编译的结果是 Java Class 文件，运行的时候，classpath 需要包含 AspectJ 的一个 jar 文件（Runtime lib）。

作为AOP的具体实现之一的AspectJ，它向Java中加入了连接点（Join Point）这个新概念，其实它也只是现存的一个Java概念的名称而已。它向Java语言中加入少许新结构：切点（pointcut）、通知（Advice）、类型间声明（Inter-typedeclaration）和方面（Aspect）。 切点和通知动态地影响程序流程，类型间声明则是静态的影响程序的类等级结构，而方面则是对所有这些新结构的封装.

**mybatis 嵌套查询和嵌套结果有哪些区别?**

* + 嵌+套查询的弊端：即嵌套查询的N+1问题  
    尽管嵌套查询大量的简化了存在关联关系的查询，但它的弊端也比较明显：即所谓的N+1问题。关联的嵌套查询显示得到一个结果集，然后根据这个结果集的每一条记录进行关联查询。  
    现在假设嵌套查询就一个（即resultMap 内部就一个association标签），现查询的结果集返回条数为N，那么关联查询语句将会被执行N次，加上自身返回结果集查询1次，共需要访问数据库N+1次。如果N比较大的话，这样的数据库访问消耗是非常大的！所以使用这种嵌套语句查询的使用者一定要考虑慎重考虑，确保N值不会很大。

嵌套结果查询：  
嵌套语句的查询会导致数据库访问次数不定，进而有可能影响到性能。

嵌套结果的查询：即对于一对多，多对多，多对一的情况的查询，Mybatis通过联合查询，将结果从数据库内一次性查出来，然后根据其一对多，多对一，多对多的关系和ResultMap中的配置，进行结果的转换，构建需要的对象。

**浏览器禁用掉cookie session还能用吗?怎么用?**

* Session URL重写，保证在客户端禁用或不支持COOKIE时，仍然可以使用Session
* session机制。session机制是一种服务器端的机制，服务器使用一种类似于散列表的结构（也可能就是使用散列表）来保存信息。
* 当程序需要为某个客户端的请求创建一个session时，服务器首先检查这个客户端的请求里是否已包含了一个session标识（称为session id），如果已包含则说明以前已经为此客户端创建过session，服务器就按照session id把这个session检索出来使用（检索不到，会新建一个），如果客户端请求不包含session id，则为此客户端创建一个session并且生成一个与此session相关联的session id，session id的值应该是一个既不会重复，又不容易被找到规律以仿造的字符串，这个session id将被在本次响应中返回给客户端保存。 保存这个session id的方式可以采用cookie，这样在交互过程中浏览器可以自动的按照规则把这个标识发挥给服务器。一般这个cookie的名字都是类似于 SEEESIONID。但cookie可以被人为的禁止，则必须有其他机制以便在cookie被禁止时仍然能够把session id传递回服务器。

**如何解决mysql死锁？**

MySQL有两种死锁处理方式：

1.等待，直到超时（innodb\_lock\_wait\_timeout=50s）。

2.发起死锁检测，主动回滚一条事务，让其他事务继续执行（innodb\_deadlock\_detect=on）。

由于性能原因，一般都是使用死锁检测来进行处理死锁。

**死锁检测:**

**死锁检测的原理是构建一个以事务为顶点、锁为边的有向图，判断有向图是否存在环，存在即有死锁。**

**回滚:**

**检测到死锁之后，选择插入更新或者删除的行数最少的事务回滚，基于 information\_schema.innodb\_trx**

**表中的 trx\_weight 字段来判断。**

**收集死锁信息：**

1. 利用命令 SHOW ENGINE INNODB STATUS查看死锁原因。
2. 调试阶段开启 innodb\_print\_all\_deadlocks，收集所有死锁日志。

**减少死锁：**

1. 使用事务，不使用 lock tables 。
2. 保证没有长事务。
3. 操作完之后立即提交事务，特别是在交互式命令行中。
4. 如果在用 (SELECT ... FOR UPDATE or SELECT ... LOCK IN SHARE MODE)，尝试降低隔离级别。
5. 修改多个表或者多个行的时候，将修改的顺序保持一致。
6. 创建索引，可以使创建的锁更少。
7. 最好不要用 (SELECT ... FOR UPDATE or SELECT ... LOCK IN SHARE MODE)。

如果上述都无法解决问题，那么尝试使用 lock tables t1, t2, t3 锁多张表

**数据库如何做数据库优化?**

1、优化索引、sql语句、分析慢查询   
2、设计表的时候严格根据数据的设计规范来设计数据库   
3、使用缓存，吧经常访问到的数据而且不需要变化的数据放到缓存中   
4、使用固态硬盘   
5、采用MYSQL内部自带的表分区技术，吧数据分层到不同的文件中，能够提高餐盘的读写效率   
6、垂直分表，吧一些不经常用到的数据放到一个表中，节约磁盘的I/O   
7、主从分离读写，采取主从复制把数据库的读操作和写操作分离出来   
8、数据库分表分机器（数据特变大的），主要的原理就是数据路由  
9、选择合适的表引擎，对参数地上的优化   
10、进行加购级别的缓存，静态化和分布式。   
11、不采用全文索引吗，用什么搜什么   
12、采用更快的处分方式，例如NoSql储存经常访问的数据

数据库优化是一个庞大的层面

1：从需求解决（不合理的需求，直接砍掉）

2：从开发解决（避免复杂的查询 结合需求）

3：从数据分析：查看olap oltp采用合适的硬件，内存 cpu 网络 磁盘等因素

4：选择合适的存储引擎

5：建立合适的索引

6：查询执行较慢的sql,查看 sql的执行计划分析执行慢的原因

**水平分表如何确定数据在哪个表里面?**

**sql如何优化?**

#### ****选择合适的存储引擎****

#### ****2. 优化字段的数据类型****

****3.为搜索字段添加索引****

#### ****4. 避免使用 Select \*****

#### ****5. 尽可能的使用 NOT NULL****

#### ****6. 固定长度的表会更快****

#### ****7. 使用垂直分割技术****

#### ****8. 使用EXPLAIN或DESC分析SELECT查询语句****

#### ****9. 不要使用 ORDER BY RAND() 随机排序****

以 **MySQL** 为例，常用的有两个存储引擎 **MyISAM** 和 **InnoDB**，它们各有利弊。

MyISAM 适合于一些需要大量查询的应用，但其对于有大量写操作并不是很好。甚至你只是需要update一个字段，整个表都会被锁起来，而别的进程，就算是读进程都无法操作直到更新操作完成。另外，MyISAM 对于 SELECT COUNT(\*) 这类的计算是非常快的。

InnoDB 支持行锁，于是在写操作比较多的时候，会更优秀。并且它还支持更多的高级应用，比如：事务和外键。

#### ****2. 优化字段的数据类型****

记住一个原则，越小的列会越快。对于大多数的数据库引擎来说，硬盘操作可能是最重大的瓶颈。所以，把你的数据变得紧凑会对这种情况非常有帮助，因为这减少了对硬盘的访问。

如果一个表只有几列数据（比如说字典表，配置表），那么，我们就没有理由使用 INT 来做主键，使用 MEDIUMINT, SMALLINT 或是更小的 TINYINT 会更经济一些。如果你不需要记录时间，使用 DATE 要比 DATETIME 好得多。当然，你也需要保留足够的扩展空间。

#### ****3. 为搜索字段添加索引****

索引并不一定就是给主键或是唯一的字段。如果在你的表中，有某个字段你会经常用来做搜索，那么最好是为其建立普通索引，除非你要搜索的字段是大的文本字段，那应该建立全文索引。

#### ****4. 避免使用 Select \*****

从数据库里读出越多的数据，查询就会越慢。并且，如果你的数据库服务器和WEB服务器是两台独立的服务器，这还会增加网络传输的负载。即使你要查询数据表的所有字段，也尽量不要用 \* 通配符，把所有的字段列出来也比 \* 通配符的查询效率高。

#### ****5. 尽可能的使用 NOT NULL****

除非你有很特别的原因去使用 NULL 值，你应该总是让你的字段保持 NOT NULL，并给一个默认值。 NULL 需要额外的空间，当然，并不是说你就不能使用NULL了，现实情况是很复杂的，有些情况下，你需要使用NULL值。

#### ****6. 固定长度的表会更快****

如果表中的所有字段都是固定长度的，整个表会被认为是 “static” 或 “fixed-length”。

表中如果没有VARCHAR、TEXT和BLOB这些类型的字段，就是固定长度的 。只要你包括了其中一个类型的字段，这个表就不是固定长度静态表，这样，MySQL 引擎会用另一种方法来处理。

固定长度的表会提高性能，因为MySQL搜寻得会更快一些，因为这些固定的长度是很容易计算下一个数据的偏移量，所以查询很快。而如果字段不是定长的，那么每次要找下一条的话，需要程序找到主键。

固定长度的表也更容易被缓存和重建。唯一的缺点是，固定长度的字段会浪费一些空间，因为定长的字段无论你用不用，都是要分配那么多的空间。可使用垂直分割技术，把表分割为两个，一个定长的，一个不是定长的。

#### ****7. 使用垂直分割技术****

“垂直分割”是把一张复杂的数据表按列分割成几张表的方法，这样可以降低表的复杂度和字段的数目，从而达到优化的目的。

例如：在user表中有一个字段是家庭地址，这个字段是可选字段，和其他的字段相比，你并不需要经常操作这个字段。那么，就可以把它放到另外一张表中， 这样会让user表有更好的性能。对于用户表来说，只有用户ID、用户名、密码、用户角色等会被经常使用。小一点的表总是会有好的性能。

#### ****8. 使用EXPLAIN或DESC分析SELECT查询语句****

使用 EXPLAIN 或 DESC 关键字分析SQL查询语句，可以知道MySQL是如何处理你的SQL语句的。

EXPLAIN 的结果会显示，SQL语句中索引的使用情况，要查询的数据是如何在表中进行搜索扫描和排序的等等。

#### ****9. 不要使用 ORDER BY RAND() 随机排序****

一些新手程序员，喜欢使用 ORDER BY RAND() 命令，来对查询结果进行随机排序，然后取出几条记录。但这却严重影响了性能，如果真的需要随机显示几条数据，可通过其他更好的方法来实现。

**mysql的explain(执行计划)?**

· EXPLAIN :模拟Mysql优化器是如何执行SQL查询语句的，从而知道Mysql是如何处理你的SQL语句的，分析你的查询语句或是表结构的性能瓶颈。

explain显示了mysql如何使用索引来处理select语句以及连接表。可以帮助选择更好的索引和写出更优化的查询语句。

## 2. Explain的作用

通过在select+sql语句前加上explain，我们可以看出：  
（1）表的读取顺序，id（table是id对应的表）  
（2）数据读取操作的操作类型，select type  
（3）哪些索引可以使用，possible\_keys  
（4）哪些索引被实际使用，key  
（5）表之间的引用，ref  
（6）每张表有多少行被优化器查询，rows

**聚集索引和非聚集索引?**

SQL Sever索引类型有：唯一索引，主键索引，聚集索引，非聚集索引。  
  
MySQL 索引类型有：唯一索引，主键（聚集）索引，非聚集索引，全文索引。

## 二.聚集索引

聚集（clustered）索引，也叫聚簇索引。

定义：数据行的物理顺序与列值（一般是主键的那一列）的逻辑顺序相同，一个表中只能拥有一个聚集索引。

## 三.非聚集索引

非聚集（unclustered）索引。

定义：该索引中索引的逻辑顺序与磁盘上行的物理存储顺序不同，一个表中可以拥有多个非聚集索引。

**如何保证redis和mysql双写数据一致性?**

## 懒加载

读取缓存步骤一般没有什么问题，但是一旦涉及到数据更新：数据库和缓存更新，就容易出现缓存和数据库间的数据一致性问题。不管是先写数据库，再删除缓存；还是先删除缓存，再写库，都有可能出现数据不一致的情况。举个例子：

1. 如果删除了缓存Redis，还没有来得及写库MySQL，另一个线程就来读取，发现缓存为空，则去数据库中读取数据写入缓存，此时缓存中为脏数据。
2. 如果先写了库，在删除缓存前，写库的线程宕机了，没有删除掉缓存，则也会出现数据不一致情况。
3. **延迟双删**
4. 在写库前后都进行redis.del(key)操作，并且第二次删除通过延迟的方式进行。
5. 方案一（一种思路，不严谨）具体步骤是：
6. 1）先删除缓存；
7. 2）再写数据库；
8. 3）休眠500毫秒（根据具体的业务时间来定）；
9. 4）再次删除缓存。
10. 方案二，异步延迟删除：
11. 1）先删除缓存；
12. 2）再写数据库；
13. 3）触发异步写人串行化mq（也可以采取一种key+version的分布式锁）；
14. 4）mq接受再次删除缓存。
15. 异步删除对线上业务无影响，串行化处理保障并发情况下正确删除。
16. （最终一致性和强一致性）
17. 如果对数据有强一致性要求，不能放缓存。

**redis数据类型(5种可能不是面试官想要的结果)?**

string 字符串（可以为整形、浮点型和字符串，统称为元素）

string类型的常用命令：   
自加：incr   
自减：decr   
加： incrby   
减： decrby  
list 列表（实现队列,元素不唯一，先入先出原则）

1. list类型支持的常用命令：   
   lpush:从左边推入   
   lpop:从右边弹出   
   rpush：从右变推入   
   rpop:从右边弹出   
   llen：查看某个list数据类型的长度  
   set 集合（各不相同的元素）
2. set类型支持的常用命令：   
   sadd:添加数据   
   scard:查看set数据中存在的元素个数   
   sismember:判断set数据中是否存在某个元素   
   srem:删除某个set数据中的元素  
   hash hash散列值（hash的key必须是唯一的）
3. hash数据类型支持的常用命令:   
   hset:添加hash数据   
   hget:获取hash数据   
   hmget:获取多个hash数据

sort set 有序集合

1. sort set和hash很相似,也是映射形式的存储:   
   zadd:添加   
   zcard:查询   
   zrange:数据排序

**redis 持久化策略/以及区别?**

* **二. 两种方式优缺点**
* **1. RDB方式**
* •优点：
* 1.RDB是一个单一的紧凑文件,它保存了某个时间点得数据集,非常适用于数据集的备份,比如你可以在每个小时报保存一下过去24小时内的数据,同时每天保存过去30天的数据,这样即使出了问题你也可以根据需求恢复到不同版本的数据集.  
  2.RDB是一个紧凑的单一文件,方便传送，适用于灾难恢复.  
  3.RDB在保存RDB文件时父进程唯一需要做的就是fork出一个子进程,接下来的工作全部由子进程来做，父进程不需要再做其他IO操作，所以RDB持久化方式可以最大化redis的性能.  
  4.与AOF相比,在恢复大的数据集的时候，RDB方式会更快一些.
* •缺点：
* 1.Redis意外宕机,可能会丢失几分钟的数据（取决于配置的save时间点）。RDB方式需要保存珍整个数据集，是一个比较繁重的工作，通常需要设置5分钟或者更久做一次完整的保存。  
  2.RDB 需要经常fork子进程来保存数据集到硬盘上,当数据集比较大的时候,fork的过程是非常耗时的,可能会导致Redis在一些毫秒级内不能响应客户端的请求.如果数据集巨大并且CPU性能不是很好的情况下,这种情况会持续更久。
* **2. AOF方式**
* **•优点**
* 1.使用AOF 会让Redis数据更加耐久: 你可以使用不同的fsync策略：无fsync,每秒fsync,每次写的时候fsync.使用默认的每秒fsync策略,Redis的性能依然很好(fsync是由后台线程进行处理的,主线程会尽力处理客户端请求),一旦出现故障，你最多丢失1秒的数据.  
  2.AOF文件是一个只进行追加的日志文件,所以不需要写入seek,即使由于某些原因(磁盘空间已满，写的过程中宕机等等)未执行完整的写入命令,你也也可使用redis-check-aof工具修复这些问题.  
  3.Redis 可以在 AOF 文件体积变得过大时，自动地在后台对 AOF 进行重写： 重写后的新 AOF 文件包含了恢复当前数据集所需的最小命令集合。 整个重写操作是绝对安全的，因为 Redis 在创建新 AOF 文件的过程中，会继续将命令追加到现有的 AOF 文件里面，即使重写过程中发生停机，现有的 AOF 文件也不会丢失。 而一旦新 AOF 文件创建完毕，Redis 就会从旧 AOF 文件切换到新 AOF 文件，并开始对新 AOF 文件进行追加操作。  
  4.AOF 文件有序地保存了对数据库执行的所有写入操作， 这些写入操作以 Redis 协议的格式保存， 因此 AOF 文件的内容非常容易被人读懂， 对文件进行分析也很轻松。 导出AOF 文件也非常简单： 举个例子， 如果你不小心执行了 FLUSHALL 命令， 但只要 AOF 文件未被重写， 那么只要停止服务器， 移除 AOF 文件末尾的 FLUSHALL 命令， 并重启 Redis ， 就可以将数据集恢复到 FLUSHALL 执行之前的状态。
* **•缺点**
* 1.对于相同的数据集来说，AOF 文件的体积通常要大于 RDB 文件的体积。  
  2.根据所使用的 fsync 策略，AOF 的速度可能会慢于 RDB 。 在一般情况下， 每秒 fsync 的性能依然非常高， 而关闭 fsync 可以让 AOF 的速度和 RDB 一样快， 即使在高负荷之下也是如此。 不过在处理巨大的写入载入时，RDB 可以提供更有保证的最大延迟时间。

redis有两种持久化方案：RDB和AOF ​ rdb类似一种数据的备份，就是将缓存中的数据冷备份到磁盘。需要恢复直接将改文件放到指定目录下即可。恢复速度快，数据丢失相 对较多。 ​ aof就是日志记录，将该redis中发生的所有数据变化记录下来。恢复时就可以通过aof中的日志进行恢复。因为是解析日志，恢复速度 慢，但是数据丢失少。 ​ 如果同时使用 RDB 和 AOF 两种持久化机制，那么在 redis 重启的时候，会使用 AOF 来重新构建数据，因为 AOF 中的数据更加完整。

**什么是restful风格?**

一、概述（百度百科）  
REST（英文：Representational State Transfer，简称REST）  
一种软件架构风格、设计风格，而不是标准，只是提供了一组设计原则和约束条件。它主要用于客户端和服务器交互类的软件。基于这个风格设计的软件可以更简洁，更有层次，更易于实现缓存等机制。

二、什么是RESTful  
REST 指的是一组架构（约束条件）和原则。满足这些（约束条件）和（原则）的应用程序或设计就是 RESTful。

三、restful有什么特点  
1.每一个URI代表一种资源，独一无二  
2.客户端和服务器之间，传递这种资源的某种表现层  
3.客户端通过四个HTTP动词，对服务器端资源进行操作，实现"表现层状态转化"。

**@Commpoent和@Bean的区别?**

@Component 和 @Bean 是两种使用注解来定义bean的方式。

@Component（和@Service和@Repository）用于自动检测和使用类路径扫描自动配置bean。注释类和bean之间存在隐式的一对一映射（即每个类一个bean）。这种方法对需要进行逻辑处理的控制非常有限，因为它纯粹是声明性的。

@Bean用于显式声明单个bean，而不是让Spring像上面那样自动执行它。它将bean的声明与类定义分离，并允许精确地创建和配置bean。

**servlet生命周期?**

1. 初始化阶段，调用init()方法

2. 响应客户请求阶段，调用service()方法

3. 终止阶段，调用destroy()方法

**limit优化(大表)?**

（1）、通过判断id的范围来分页  
select  id,my\_sn from big\_data where id>5000000 limit 10;  
也得到了分页的数据，但是我们发现如果id不是顺序的，也就是如果有数据删除过的话，那么这样分页数据就会不正确，这个是有缺陷的。  
（2）、通过连接查询来分页  
我们可以先查询500w条数据开始分页的那10个id，然后通过连接查询显示数据  
mysql> select b.id,b.my\_name from big\_data as b  inner join (select id from big\_data order by id limit 4500000,10) as  tmp on tmp.id=b.id;

我们测试不同起始端的分页数据

**如何查询mysql表中最后一条数据?**

select \* from test\_limit order by id DESC limit 1;

**重复扣款问题(坑比较多)?**

**Redis主从复制的过程?**

* 如果设置了一个从服务器，在连接时它发送了一个SYNC命令，不管它是第一次连接还是再次连接都没有关系。
* 然后主服务器开始后台存储，并且开始缓存新连接进来的修改数据的命令。当后台存储完成后，主服务器把数据文件发送到从服务器，
* 从服务器将其保存在磁盘上，然后加载到内存中。然后主服务器把刚才缓存的命令发送到从服务器。这是作为命令流来完成的，并且
* 和Redis协议本身格式相同。
* 你可以通过telnet自己尝试一下。在Redis服务器工作时连接到Redis端口，发送SYNC命令，会看到一个批量的传输，并且主服务器接收
* 的每一个命令都会通过telnet会话重新发送一遍。
* 当主从服务器之间的连接由于某些原因断开时，从服务器可以自动进行重连接。当有多个从服务器同时请求同步时，主服务器只进行一个后台存储。
* 当连接断开又重新连上之后，一般都会进行一个完整的重新同步，但是从Redis2.8开始，只重新同步一部分也可以。
* 设置主节点的地址和端口
* 建立套接字连接
* 发送PING命令
* 权限验证
* 同步
* 命令传播

**Redis怎么解决脑裂问题?**

redis的配置文件中，存在两个参数

min-slaves-to-write 3  
min-slaves-max-lag 10  
?  
第一个参数表示连接到master的最少slave数量  
第二个参数表示slave连接到master的最大延迟时间  
如果连接到master的slave数量小于第一个参数，且ping的延迟时间小于等于第二个参数，那么master就会拒绝写请求，配置了这两个参数之后，如果发生集群脑裂，原先的master节点接收到客户端的写入请求会拒绝，就可以减少数据同步之后的数据丢失。

注意：较新版本的redis.conf文件中的参数变成了

min-replicas-to-write 3  
min-replicas-max-lag 10  
?  
redis中的异步复制情况下的数据丢失问题也能使用这两个参数

**Redis分布式锁?**

要介绍分布式锁，首先要提到与分布式锁相对应的是线程锁、进程锁。

线程锁：主要用来给方法、代码块加锁。当某个方法或代码使用锁，在同一时刻仅有一个线程执行该方法或该代码段。线程锁只在同一JVM中有效果，因为线程锁的实现在根本上是依靠线程之间共享内存实现的，比如synchronized是共享对象头，显示锁Lock是共享某个变量（state）。

进程锁：为了控制同一操作系统中多个进程访问某个共享资源，因为进程具有独立性，各个进程无法访问其他进程的资源，因此无法通过synchronized等线程锁实现进程锁

Redis可以在项目中充当分布式锁的功能，如果页面某个功能，需要进行加锁，那么可以使用sync关键字进行加锁，然后如果该项目分布式部署，那么sync只能锁住本项目的线程，无法跨项目加锁，这个时候，就需要用到分布式锁了，

首先，在程序开始的时候，在redis中通过setnx命令，设置一个变量，如果redis中没有这个变量，那么会创建成功，有的话会创建失败，（证明已经有人加锁在使用这个变量了）

当setnx 成功后，在设置一个过期时间， 超过这个过期时间，这个变量自动过期，锁也就会释放，这样可以避免程序宕机后，变量一直存在造成死锁问题

设置完成后，就可以执行业务逻辑了，当业务逻辑执行完之后，在delete删除这个变量，删除之后，下个线程就可以创建变量了

**介绍一下哨兵机制?**

哨兵机制是redis的集群方案，优于redis负责整个项目的数据读取，意外宕机后有很大的影响，所以集群哨兵就是保障redis运行的一个方案，首先准备两台redis，ip或者端口设置不同，避免冲突，例如6379 6380两台

1.配置slave(从库，奴隶的意思) 我们把6379作为主库，所以在conf中配置 slave从库，因此，两者的数据是一模一样的

2.搭建哨兵seof:127.0.0.1:6380 , 意思是6380是6379 的从库，当主库6379发生数据改变时，会自动同步到ntinel

sentinel.conf 配置文件中， 我们可以找到port 属性，这里是用来设置sentinel 的端口。

配置redis sentinel monitor mymaster 127.0.0.1 6380 2 意思是当redis宕机那个redis变成主库

3:哨兵的原理是 每秒都是想redis发送信息，如果消息得不到及时回应，那么就是宕机了，执行切换主库的操作

**查询的覆盖索引?**

所有的查询字段是索引的一部分

所有的查询返回字段在同一个索引中

由于所有出现在查询中的字段是索引的一部分， MongoDB 无需在整个数据文档中检索匹配查询条件和返回使用相同索引的查询结果。

因为索引存在于RAM中，从索引中获取数据比通过扫描文档读取数据要快得多

**rabbitmq如何解决消息丢失问题?**

从实际来讲，队列中消息丢失是能的，例如生产者发送了10个消息到队列每个消息增加一条数据到数据库，结果最终数据库只有7条数据，这就是“消息丢失”，首先咱们要找到消息是从哪里丢失的，

有三个可能，

1：生产者丢失了消息，消息并没有成功发送到队列，

2：其次rabbit队列自己弄丢了，如果在发送消息时，mq意外宕机，也有可能造成丢失问题，

3：消费者丢失了数据，在消费者执行消息过程中取到消息后，消费者宕机了，队列认为消息已经消费了，+-

如何解决 ，

第一种生产者丢失消息，可使用confirm确认机制可以保证，就是发送消息确认一下消息，表明消息是否发送到队列

或者定义rpc回调接口 当消息执行的时候，可以回调一发送端的方法，证明消息已经被消费了

第二种队列自己丢失，可通过设置消息的持久化解决，消息发送后，是存储在内存中的，宕机肯定是造成丢失，然后在声明消息时可配置消息持久化到硬盘，服务宕机是无法影响硬盘的，数据不会丢失，交换机，队列，消息，都要设置持久化

第三种在消费者丢失

与spring整合后，消费者的监听方法监听到消息后会执行相应的业务逻辑，如果执行业务的过程中报错，会默认队列消息回滚就是将消息重新扔回到队列然后再次发送执行，但如果是业务或者数据本身除了问题，那么会导致再次报错，消息回滚，重而导致死循环，所以我们执行业务的试试一般要进行try catch异常捕捉，如果发生异常了，可以将发生异常的消息记录下来 而不是回滚到队列，那么就需要工作人员去查看这些失败的消息是什么了，解决好问题，然后重新执行下消息就可以

总结，如果以上三种方案都采用的话，在发生消息丢失的可能就很小了，如果还有，那就是很小几率的问题了，基本不会发生，可以用其他第三方处理，发送消息到队列的时候，在手动将消息像redis缓存，或者其他存储空间发送存储，消费者消费消息之后，在将redis中的消息进行标记，最后如果真的有消息未处理丢失，可以查看redis中是否有数据没有被标记，那这些数据就是丢失的消息可以排查原因为什么会被丢失，是否是数据本身问题等

**mybatis一对多，多对多怎么实现?**

**mq怎么保证消息不会重复消费?**

**es读(通过id)/写数据的过程?**

**es搜索过程?**

**hashmap底层实现?**

HashMap是双列的键值对结构，继承了AbstractMap类，实现了Map接口效率非常高

HashMap底层变量

transient Node<K,V>[] table; map底层是node数组，node实现于Map.Entry,由hash key vlaue next组成

static final int DEFAULT\_INITIAL\_CAPACITY = 1 << 4; // aka 16默认的初始化数据长度

static final int TREEIFY\_THRESHOLD = 8; 默认的扩容树长度

static final float DEFAULT\_LOAD\_FACTOR = 0.75f;默认的负载因子，数组占用率超过0.75进行扩容

int threshold;//扩容长度 数组长度\*负载因子

当我们往HashMap中put元素的时候，判断数组是否为null如果为null调用resize()进行扩容初始化数组

在根据key的hashCode进行高位异或计算hash值，根据hash值&数组最大下标 得到这个元素在数组中的位置（即下标）， 高位异或是为了尽可能减少hash碰撞的情况

如果数组该下标上没有数据，则根据key value hash创建一个Node节点，放入数组下标中，如果该数组下标是有元素的，那么证明发生了hash碰撞就吧改下标元素拓展为链表结构，将新Node节点放在老Node节点中的next节点中。此时就完成了链表结构的拓展，但是如果新增节点的时候，hash碰撞较高的话会导致链表效率，也会影响性能，所以链表赋值的时候，进行了拓展如果链表超过8个长度的话，则将链表转为红黑树结构存储，从而提高查询效率

当数组占用率达到负载因子占用率的时候，数组长度双倍进行扩容，扩容之后，需要将老数组节点遍历重新根据hash&新数组最大下标进行计算下标，然后重新赋值计算node节点应该在的位置

从HashMap中get元素时，首先计算key的hashCode，找到数组中对应位置的某一元素，然后通过key的equals方法在对应位置的链表中找到需要的元素。

归纳起来简单地说，HashMap 在底层将 key-value 当成一个整体进行处理，这个整体就是一个 Entry 对象。HashMap 底层采用一个 Entry[] 数组来保存所有的 key-value 对，当需要存储一个 Entry 对象时，会根据hash算法来决定其在数组中的存储位置，在根据equals方法决定其在该数组位置上的链表中的存储位置；当需要取出一个Entry时，也会根据hash算法找到其在数组中的存储位置，再根据equals方法从该位置上的链表中取出该Entry。

**arrayList底层实现?**

首先ArrayList继承于AbstractList抽象类，实现了，List，和序列化的接口，

ArrayList的底层变量有，

默认的数组长度 private static final int DEFAULT\_CAPACITY = 10;

一个空数组 private static final Object[] EMPTY\_ELEMENTDATA = {};

底层数组 transient Object[] elementData;

集合长度 private int size;

ArrayList集合底层是一个Object类型的数组elementData，所以list可以放入任意类型的元素，

ArrayList无参构造中默认实例化了一个空的object数组给elementData赋值，所以初始长度为0，在add增加的元素方法中会有一个方法判断ensureCapacityInternal，如果elementData是空数组的话那么会自动增加一个元素长度为10的数组，装入数据后赋值给elementData，默认长度是10 是由DEFAULT\_CAPACITY这个变量规定，

ArrayList有参构造中会传入一个集合长度initialCapacity，这是会判断如果initialCapacity大于0则创建这个长度的Object数组赋值给elementData，如果initialCapacity 等于0则用默认的空数组赋值给elementData，如果小于0则会抛出异常Illegal Capacity:

ArrayList中的get方法，获取数组会传入下标，首先有个rangeCheck方法判断当前下标是否大于数组长度，如果大于报异常，如果小于则返回elementData【下标】

ArrayList中的add方法，会先判断是否是elementData是否是空数组，如果不是，则会elementData【size++】=值，如果数组长度不够用则会进行当前数组长度1.5倍扩容

新长度=老长度+老长度>>1 (右移1是除以2的意思)

int newCapacity = oldCapacity + (oldCapacity >> 1);

ArrayList的clear清除方法，就是一个for循环，终值是size，然后将elementData【i】设置成null

**CAS是什么?**

**CAS的ABA问题?怎么解决?**

**volatile关键字的作用?**

**synchronized的状态?**

JDK6之前只有两个状态：无锁、有锁（重量级锁），而在JDK6之后对synchronized进行了优化，新增了两种状态，总共就是四个状态：**无锁状态、偏向锁、轻量级锁、重量级锁**，其中无锁就是一种状态了。锁的类型和状态在对象头Mark Word中都有记录，在申请锁、锁升级等过程中JVM都需要读取对象的Mark Word数据。

**DCL(双检索)单例会有线程安全的问题吗?**

**创建线程的方式**

**继承 Thread 类**

Thread 类实现了 Runnable 接口并定义了操作线程的一些方法，我们可以通过创建类时继承 Thread类来创建一个线程。

具体实现：

（1）创建一个继承Thread的类ThreadDemo

（2）重新run()方法

调用步骤：

（1）创建ThreadDemo 类的对象t1

（2）执行t1.start() 方法来启动线程

**实现 Runnable 接口**

通过实现Runnable 接口来创建线程类 RThread，但是使用的时候，仍需要创建Thread 对象，把RThread的对象当成参数传入。

具体操作：

（1）实现Runnable 接口创建线程类 RThread

（2）重写run()方法

调用步骤：

（1）创建RThread 类的对象 rThread

（2）创建Thread类对象，并把rThread当成参数传入，相当于对rThread进行了封装。

（3）通过start()方法启动线程

**通过 ExecutorService 和 Callable\ 实现有返回值的线程**

我们需要在主线程中开启多个线程去执行一个任务，然后收集各个线程的返回结果并将最终结果进行汇总，这是就需要用到 Callable 接口。

具体步骤：

（1）创建一个类实现Callable接口

（2）重写 call() 方法

调用步骤：

（1）创建线程池

（2）创建接收结果的列表集合

（3）创建线程对象

（4）将线程对象提交到线程池中，并将返回结果接收

（5）将返回结果加入结果集合

（6）关闭线程池

**基于线程池的execute()，创建临时线程**

我们可以利用缓存策略使用线程池来创建线程

具体创建：

（1）创建线程池

（2）调用线程池的execute()方法

（3）采用匿名内部类的方法，创建Runnable对象，并重写run()方法

**线程池的状态?**

1.RUNNING：这是最正常的状态，接受新的任务，处理等待队列中的任务。线程池的初始化状态是RUNNING。线程池被一旦被创建，就处于RUNNING状态，并且线程池中的任务数为0。

2.SHUTDOWN：shutdown不接受新的任务提交，但是会继续处理等待队列中的任务。调用线程池的shutdown()方法时，线程池由RUNNING -> SHUTDOWN。

3.STOP：不接受新的任务提交，不再处理等待队列中的任务，中断正在执行任务的线程。调用线程池的shutdownNow()方法时，线程池由(RUNNING or SHUTDOWN ) -> STOP。

4.TIDYING：tidying所有的任务都销毁了，workCount 为 0，线程池的状态在转换为 TIDYING 状态时，会执行钩子方法 terminated()。因为terminated()在ThreadPoolExecutor类中是空的，所以用户想在线程池变为TIDYING时进行相应的处理；可以通过重载terminated()函数来实现。

当线程池在SHUTDOWN状态下，阻塞队列为空并且线程池中执行的任务也为空时，就会由 SHUTDOWN -> TIDYING。

当线程池在STOP状态下，线程池中执行的任务为空时，就会由STOP -> TIDYING。

5.TERMINATED：terminated线程池处在TIDYING状态时，执行完terminated()之后，就会由 TIDYING -> TERMINATED。

**大表sql如何优化?**

第一优化你的sql和索引；

第二加缓存，memcached,redis；

第三以上都做了后，还是慢，就做主从复制或主主复制，读写分离，可以在应用层做，效率高，也可以用三方工具，第三方工具推荐360的atlas,其它的要么效率不高，要么没人维护；

第四如果以上都做了还是慢，不要想着去做切分，mysql自带分区表，先试试这个，对你的应用是透明的，无需更改代码,但是sql语句是需要针对分区表做优化的，sql条件中要带上分区条件的列，从而使查询定位到少量的分区上，否则就会扫描全部分区，另外分区表还有一些坑，在这里就不多说了；

第五如果以上都做了，那就先做垂直拆分，其实就是根据你模块的耦合度，将一个大的系统分为多个小的系统，也就是分布式系统；

第六才是水平切分，针对数据量大的表，这一步最麻烦，最能考验技术水平，要选择一个合理的sharding key,为了有好的查询效率，表结构也要改动，做一定的冗余，应用也要改，sql中尽量带sharding key，将数据定位到限定的表上去查，而不是扫描全部的表；

**自定义线程池的7个参数?**

corePoolSize：线程池核心线程数量，核心线程不会被回收，即使没有任务执行，也会保持空闲状态。如果线程池中的线程少于此数目，则在执行任务时创建。

maximumPoolSize：池允许最大的线程数，当线程数量达到corePoolSize，且workQueue队列塞满任务了之后，继续创建线程。

keepAliveTime：超过corePoolSize之后的“临时线程”的存活时间。

unit：keepAliveTime的单位。

workQueue：当前线程数超过corePoolSize时，新的任务会处在等待状态，并存在workQueue中，BlockingQueue是一个先进先出的阻塞式队列实现，底层实现会涉及Java并发的AQS机制，有关于AQS的相关知识，我会单独写一篇，敬请期待。

threadFactory：创建线程的工厂类，通常我们会自顶一个threadFactory设置线程的名称，这样我们就可以知道线程是由哪个工厂类创建的，可以快速定位。

handler：线程池执行拒绝策略，当线数量达到maximumPoolSize大小，并且workQueue也已经塞满了任务的情况下，线程池会调用handler拒绝策略来处理请求。

**SpringBoot的优点?**

1）优点

快速构建项目。

对主流开发框架的无配置集成。

项目可独立运行，无须外部依赖Servlet容器。

提供运行时的应用监控。

极大地提高了开发、部署效率。

与云计算的天然集成。

2）缺点

版本迭代速度很快，一些模块改动很大。

由于不用自己做配置，报错时很难定位。

网上现成的解决方案比较少。

**SpringBoot自动配置原理?**

Boot项目启动类配置SpringBootApplication注解。这个注解底层包含了@SpringBootConfiguration（ioc容器配置类），@EnableAutoConfiguration(自动化配置类),@ComponentScan（注解扫描类，实例化相关对象）当启动类执行后，启动类的注解，会默认扫描所有的类完成创建，这个时候，会扫描到spring-boot-autoconfigure这个jar包依赖，这个依赖中定义了大量技术的配置类，mvc，缓存，对列，mybatis应有尽有，每个类中都定义了Configuration注解，注解完成了各个类的创建，而且属性，都是从application配置文件中读取，并且进入属性注入的，所以，启动类启动后，扫描创建默认的配置类，然后配置类去检查application中有没有配置该技术，有的话。直接读取创建，完成了自动配置

**为什么使用springCloud框架(如果只是回答他的组件和他于springBoot的关系的话基本就凉了)?**

Spring Cloud 来源于 Spring，质量、稳定性、持续性都可以得到保证。

spirng Cloud 天然支持 Spring Boot，更加便于业务落地。

Spring Cloud 是 Java 领域最适合做微服务的框架。相比于其它框架，Spring Cloud 对微服务周边环境的支持力度最大。对于中小企业来讲，使用门槛较低。

Spring Cloud 是微服务架构的最佳落地方案。

与分布式系统相关的复杂性 – 包括网络问题，延迟开销，带宽问题，安全问题。

处理服务发现的能力 – 服务发现允许集群中的进程和服务找到彼此并进行通信。

解决冗余问题 – 冗余问题经常发生在分布式系统中。

负载平衡 – 改进跨多个计算资源（例如计算机集群，网络链接，中央处理单元）的工作负载分布。

减少性能问题 – 减少因各种操作开销导致的性能问题。

**什么是微服务/微服务的意义?**

在介绍微服务时，首先得先理解什么是微服务，顾名思义，微服务得从两个方面去理解，什么是"微"、什么是"服务"，

微，狭义来讲就是体积小、著名的"2 pizza 团队"很好的诠释了这一解释（2 pizza 团队最早是亚马逊 CEO Bezos提出来的，意思是说单个服务的设计，所有参与人从设计、开发、测试、运维所有人加起来 只需要2个披萨就够了 ）。 而所谓服务，一定要区别于系统，服务一个或者一组相对较小且独立的功能单元，是用户可以感知最小功能集。

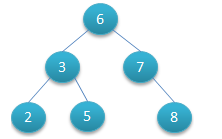
**项目中的难点/如何解决的?**

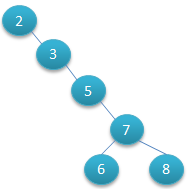
**分布式中事务有哪些问题?**

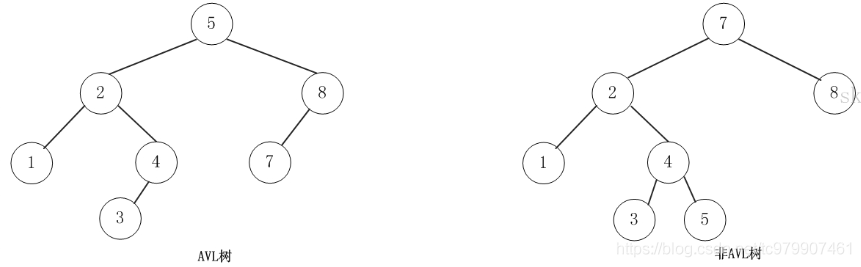
1. 阻塞式重试；在微服务架构中，阻塞式重试是比较常见的一种方式
2. 2PC、3PC 传统事务；
3. 使用队列，后台异步处理；
4. TCC 补偿事务；
5. 本地消息表（异步确保）；
6. MQ 事务。

**二叉树?平衡二叉树?红黑树?Btree?B+tree?**

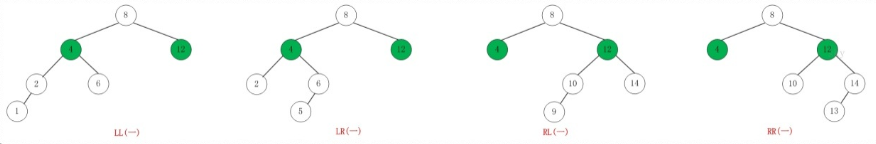
1: 二叉树:

二叉树具有以下性质：左子树的键值小于根的键值，右子树的键值大于根的键值。  
如下图所示就是一棵二叉查找树，  
  
对该二叉树的节点进行查找发现深度为1的节点的查找次数为1，深度为2的查找次数为2，深度为n的节点的查找次数为n，因此其平均查找次数为 (1+2+2+3+3+3) / 6 = 2.3次

二叉查找树可以任意地构造，同样是2,3,5,6,7,8这六个数字，也可以按照下图的方式来构造：  
  
但是这棵二叉树的查询效率就低了。因此若想二叉树的查询效率尽可能高，需要这棵二叉树是平衡的，从而引出新的定义——平衡二叉树，或称AVL树。

2) : 平衡二叉树（AVL树）:在符合二叉查找树的条件下，还满足任何节点的两个子树的高度最大差为1。下面的两张图片，左边是AVL树，它的任何节点的两个子树的高度差<=1；右边的不是AVL树，其根节点的左子树高度为3，而右子树高度为1；  


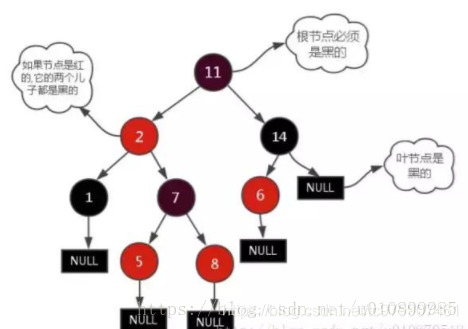
如果在AVL树中进行插入或删除节点，可能导致AVL树失去平衡，这种失去平衡的二叉树可以概括为四种姿态：LL（左左）、RR（右右）、LR（左右）、RL（右左）。它们的示意图如下：



这四种失去平衡的姿态都有各自的定义：

* LL：LeftLeft，也称“左左”。插入或删除一个节点后，根节点的左孩子（Left Child）的左孩子（Left Child）还有非空节点，导致根节点的左子树高度比右子树高度高2，AVL树失去平衡。
* RR：RightRight，也称“右右”。插入或删除一个节点后，根节点的右孩子（Right Child）的右孩子（Right Child）还有非空节点，导致根节点的右子树高度比左子树高度高2，AVL树失去平衡。
* LR：LeftRight，也称“左右”。插入或删除一个节点后，根节点的左孩子（Left Child）的右孩子（Right Child）还有非空节点，导致根节点的左子树高度比右子树高度高2，AVL树失去平衡。
* RL：RightLeft，也称“右左”。插入或删除一个节点后，根节点的右孩子（Right Child）的左孩子（Left Child）还有非空节点，导致根节点的右子树高度比左子树高度高2，AVL树失去平衡。

3):红黑树(弱平衡二叉树): 红黑树也属于平衡二叉树，但在每个节点增加一个存储位表示节点的颜色，非红即黑。通过对任何一条从根到叶子的路径上各个节点着色的方式的限制，红黑树确保没有一条路径会比其它路径长出两倍，因此，红黑树是一种弱平衡二叉树（由于是弱平衡，可以看到，在相同的节点情况下，AVL树的高度低于红黑树），相对于要求严格的AVL树来说，它的旋转次数少，所以对于搜索，插入，删除操作较多的情况下，我们就用红黑树。



4):平衡多路查找树(B-tree): B-Tree是为磁盘等外存储设备设计的一种平衡查找树。因此在讲B-Tree之前先了解下磁盘的相关知识。

系统从磁盘读取数据到内存时是以磁盘块（block）为基本单位的，位于同一个磁盘块中的数据会被一次性读取出来，而不是需要什么取什么。

InnoDB存储引擎中有页（Page）的概念，页是其磁盘管理的最小单位。InnoDB存储引擎中默认每个页的大小为16KB，可通过参数innodb\_page\_size将页的大小设置为4K、8K、16K，在MySQL中可通过如下命令查看页的大小：mysql> show variables like 'innodb\_page\_size'

**左旋(的过程/以及触发条件),右旋(的过程/以及触发条件),btree向上分裂?**

**ReenTrantLock?**

解释:ReentrantLock是java.util.concurrent.locks包中的一个类,是独占锁,为最后一个执行lock操作成功且为释放锁的线程锁拥有.因为它是基于AQS实现的,

特点:

1. ReenTrantLock可以指定是公平锁还是非公平锁; synchronized只能是非公平锁。

2. ReenTrantLock提供了一个Condition类，用来实现唤醒特定的线程; synchronized要么随机唤醒一个线程要么唤醒全部线程。

3. ReenTrantLock提供了一种能够中断等待锁的线程的机制。

**限流(令牌桶,以及具体实现)?**

限流是对某一时间窗口内的请求数进行限制，保持系统的可用性和稳定性，防止因流量暴增而导致的系统运行缓慢或宕机。常用的限流算法有令牌桶和和漏桶，而Google开源项目Guava中的RateLimiter使用的就是令牌桶控制算法。

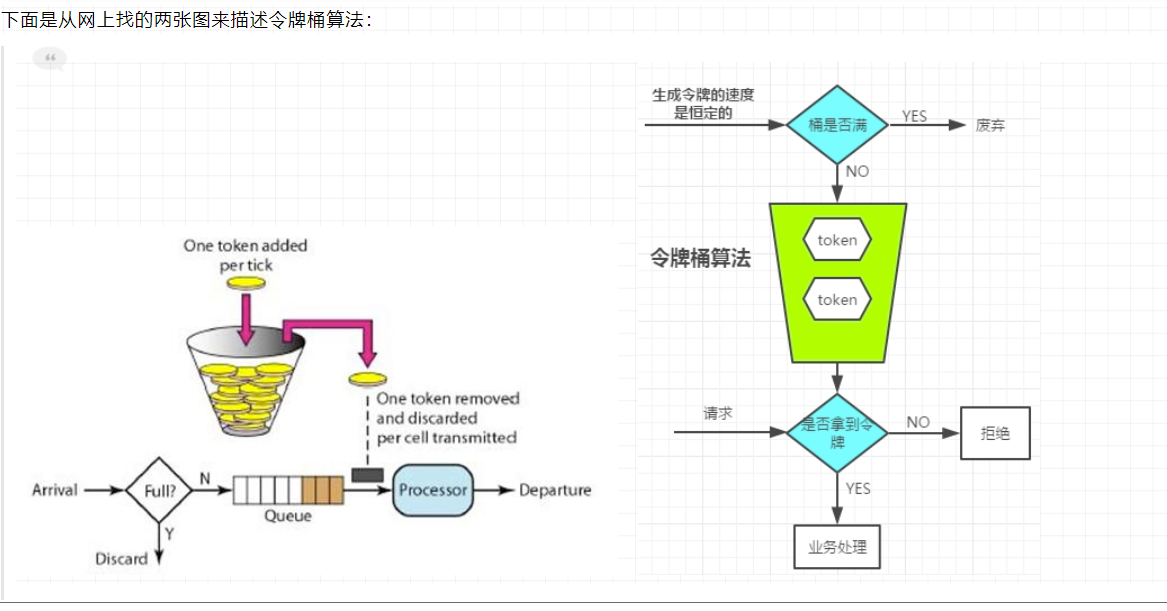
在开发高并发系统时有三把利器用来保护系统：缓存、降级和限流

1):缓存：缓存的目的是提升系统访问速度和增大系统处理容量

2):降级：降级是当服务器压力剧增的情况下，根据当前业务情况及流量对一些服务和页面有策略的降级，以此释放服务器资源以保证核心任务的正常运行

3):限流：限流的目的是通过对并发访问/请求进行限速，或者对一个时间窗口内的请求进行限速来保护系统，一旦达到限制速率则可以拒绝服务、排队或等待、降级等处理

令牌桶:生成令牌的速度是恒定的，而请求去拿令牌是没有速度限制的。这意味，面对瞬时大流量，该算法可以在短时间内请求拿到大量令牌，而且拿令牌的过程并不是消耗很大的事情。



**TCC补偿失败怎么处理?**

什么是TCC事务:TCC是Try、Confirm、Cancel三个词语的缩写，TCC要求每个分支事务实现三个操作 ：预处理Try、确认Confirm、撤销Cancel。Try操作做业务检查及资源预留，Confirm做业务确认操作，Cancel实现一个与Try相反的操作既回滚操作。TM首先发起所有的分支事务的try操作，任何一个分支事务的try操作执行失败，TM将会发起所有分支事务的Cancel操作，若try操作全部成功，TM将会发起所有分支事务的Confirm操作，其中Confirm/Cancel操作若执行失败，TM会进行重试。

**JVM内存模型?**

堆:堆是Java虚拟机所管理的内存最大一块。堆是所有线程共享的一块内存区域，在虚拟机启动时创建。

此内存区域唯一的目的就是存放对象实例。所有的对象实例都在这里分配内存

虚拟机栈 :与程序计数器一样，java虚拟机栈也是线程私有的，他的生命周期与线程相同。

本地方法栈:与虚拟机栈发挥的作用非常类似，他们之间的区别是虚拟机栈为虚拟机执行java方法服务,

而本地方法栈则为虚拟机使用到的native方法服务

程序计数器:程序计数器是一块较小的内存空间，他可以看做是当前线程所执行字节码的行号指示器。

在虚拟机的概念模型里，字节码解释器工作时就是通过改变这个计数器的值来选择下一条将要执行的字节码指令。

运行时常量池:运行时常量池是方法区的一部分。Class文件除了 有类的版本、字段、方法、接口等描述信息外，还有一项是常量池，

用于存放编译期生成的各种字面量和符号引用，这部分内容在类加载后进入方法区的运行时常量池。

对象创建:在语言层面上，创建对象（克隆，反序列化）通常只是一个new关键字。

Jvm是java虚拟机，在项目启动后，运行中产生的一些变量，全部 jvm中，jvm中区域划分为堆，栈，常量区，方法栈，等很多区域，但是有关于变量存储的，仅限于 堆栈，常量区中，我们项目中的基本数据类型，创建后，放在常量区中，也包括不用new创建的String, 在经过new实例化的创建的对象，默认在堆中开辟新的 空间存储在堆中，栈中放的是对象引用的指针，指针指向堆中

**类的加载机制(双亲委派)?**

一个编译后的class文件，想要在JVM中运行，就需要先加载到JVM中。java中将类的加载工具抽象为类加载器，而通过加载工具加载类文件的具体方式被称为双亲委派机制。

双亲:既然称为双亲委派机制，则双亲是必须存在的。ClassLoader类存在一个parent属性。因此可以配置双亲。这个双亲是指ExtClassLoader和AppClassLoader，在JDK中则是这样设置：ExtClassLoader.parent=null;

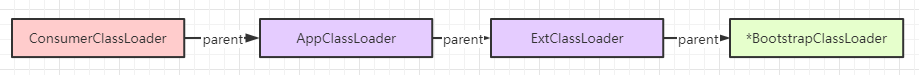
AppClassLoader.parent=ExtClassLoader

XXXClassLoader.parent=AppClassLoader

自定义的类加载器在构建时，如果未指定parent，则使用getSystemClassLoader()来获取父类加载器，而该方法默认获取的就是AppClassLoader。

委派: 双亲设置之后，便可以委派了，委派过程也就是类文件加载过程

以上的类加载方式，图示如下



**类的生命周期?**

概述:一个Java类从开始到结束整个生命周期会经历7个阶段：加载（Loading）、验证（Verification）、准备（Preparation）、解析（Resolution）、初始化（Initialization）、使用（Using）和卸载（Unloading）。其中验证、准备、解析三个部分又统称为连接（Linking）.

**加载**:加载过程就是把class字节码文件载入到虚拟机中，至于从哪儿加载，虚拟机设计者并没有限定，你可以从文件、压缩包、网络、数据库等等地方加载class字节码。

**验证**:验证的目的是确保class文件的字节流中信息符合虚拟机的要求，不会危害虚拟机安全，使得虚拟机免受恶意代码的攻击，这一步至关重要。

验证分为: 文件格式验证, 源数据验证, 字节码验证, 符号引用验证

**准备**:准备阶段的工作就是为类的静态变量分配内存并设为jvm默认的初值，对于非静态的变量，则不会为它们分配内存。静态变量的初值为jvm默认的初值，而不是我们在程序中设定的初值。(仅包含类变量,不包含实例变量).

**解析**:虚拟机将常量池中的符号引用替换为直接引用，解析动作主要针对类或接口，字段，类方法，方法类型等等。

**初始化**:在该阶段，才真正意义上的开始执行类中定义的java程序代码，该阶段会执行类构造器，并且在Java虚拟机规范中有明确的规定，在下面5种情况下必须对类进行初始化：遇到new、getstatic、putstatic、invokestatic这4条字节码指令时，如果类没有进行过初始化，则需要先触发其初始化。

**使用**:使用该类所提供的功能，其中包括主动引用和被动引用。

**1):主动引用：**通过new关键字实例化对象、读取或设置类的静态变量、调用类的静态方法。

通过反射方式执行以上三种行为。

初始化子类的时候，会触发父类的初始化。

作为程序入口直接运行时（也就是直接调用main方法）。

**2):被动引用：**

引用父类的静态字段，只会引起父类的初始化，而不会引起子类的初始化。

定义类数组，不会引起类的初始化。

引用类的常量，不会引起类的初始化。

**卸载**:从内存中释放，在我之前写的**[垃圾回收机制（GC）总结](https://link.zhihu.com/?target=http%3A//mp.weixin.qq.com/s%3F__biz%3DMzU3OTkyMDAxNg%3D%3D%26mid%3D2247483934%26idx%3D1%26sn%3D9f62ec5a72481af362b66a44fb27dbba%26chksm%3Dfd5f8be1ca2802f7f00f0621a5370349ade57802f9c89ea642c71bc93d52cd920f904d180327%26scene%3D21%23wechat_redirect" \t "_blank)**一文中有介绍到方法区内存回收中对类的回收条件

**OOM会发生在什么地方?**

堆内存

堆内存不足是最常见的发送OOM的原因之一。

如果在堆中没有内存完成对象实例的分配，并且堆无法再扩展时，

将抛出OutOfMemoryError异常，抛出的错误信息是“java.lang.OutOfMemoryError:Java heap space”。

当前主流的JVM可以通过-Xmx和-Xms来控制堆内存的大小，

发生堆上OOM的可能是存在内存泄露，也可能是堆大小分配不合理。

直接内存

直接内存虽然不是虚拟机运行时数据区的一部分，但既然是内存，就会受到物理内存的限制。

在JDK1.4中引入的NIO使用Native函数库在堆外内存上直接分配内存，但直接内存不足时，也会导致OOM。

Java虚拟机栈和本地方法栈

这两个区域的区别不过是虚拟机栈为虚拟机执行Java方法服务，

而本地方法栈则为虚拟机使用到的Native方法服务，在内存分配异常上是相同的。

在JVM规范中，对Java虚拟机栈规定了两种异常：

a. 如果线程请求的栈大于所分配的栈大小，则抛出StackOverFlowError错误，比如进行了一个不会停止的递归调用；

b. 如果虚拟机栈是可以动态拓展的，拓展时无法申请到足够的内存，则抛出OutOfMemoryError错误。

方法区

随着Metaspace元数据区的引入，方法区的OOM错误信息也变成了“java.lang.OutOfMemoryError:Metaspace”

对于旧版本的Oracle JDK，由于永久代的大小有限，而JVM对永久代的垃圾回收并不积极，

如果往永久代不断写入数据，例如String.Intern()的调用，在永久代占用太多空间导致内存不足，

也会出现OOM的问题，对应的错误信息为“java.lang.OutOfMemoryError:PermGen space”

**什么是垃圾?**

在 JVM 的眼中，垃圾就是指那些在堆中存在的，已经“死亡”的对象。

而对于“死亡”的定义，我们可以简单的将其理解为“不可能再被任何途径使用的对象”。

那怎样才能确定一个对象是存活还是死亡呢？这就涉及到了垃圾判断算法，其主要包括引用计数法和可达性分析法。

**如果确定一个对象是垃圾(引用计数,根可达)?**

引用计数算法:

给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;

当引用失效时,计数器值就减1;计数器为0,说明对象没有被使用

但是主流的java虚拟机里面没有选用引用计数法来管理内存,

其中最重要的原因是它很难解决对象之间互相循环引用的问题(objA和objB都有字段instance,赋值令objA.instance=ObjB和objB.instance=objA,除此之外,两者再无任何引用,但是因为它们互相引用着对方,导致它们的引用计数都不为0,于是引用计数算法无法通知GC收集器回收它们)

可达性分析算法:

通过一系列的称为”GC Roots”的对象作为起始点,从这些节点开始向下搜索,

搜索走过的路径称为引用链,当一个对象到GC Roots没有任何引用链相连,则证明此对象是不可用的

**你了解的垃圾回收算法?**

1.标记-清除算法

标记-清除算法采用从根集合（GC Roots）进行扫描，

对存活的对象进行标记，标记完毕后，再扫描整个空间中未被标记的对象，进行回收。

2.复制算法

对根集合进行扫描，对存活的对象进行标记，然后将标记的对象复制到新的内存，将旧内存清空

解决了效率问题，不会多次扫描，但是内存区域浪费了一半。

当分区的内存不够时，使用内存担保，多分一点内存，类似贷款。

3.标记-整理算法（标记-整理-清除算法）

因为复制算法中，对于内存的控制会引起内存担保，反倒使垃圾回收的效果变慢

先把存活的对象标记，将内存分成两块，被标记的对象往存活区走，没被标记的往另一侧，然后清除非存活区即可。

4.分代收集算法

分情况进行垃圾回收，对于新生代这种经常需要垃圾回收的，

选择复制算法，对于老年代这种有很多对象存活的选择标记-整理算法。

**你知道的垃圾回收器?**

Serial、parNew、ParallelScavenge、SerialOld、ParallelOld、CMS、G1

Serial是一个单线程的收集器，Serial的特点是它在进行垃圾收集时，必须“Stop the World”，

意思就是当这个垃圾收集器开始工作时，必须停止其他所有的工作线程。听起来似乎很不靠谱，

但是对于限定单个CPU的场景下，这种方式简单而高效。对于简单的桌面应用，分配给虚拟机的内存不会很大，

对于一两百兆的新生代，Serial的垃圾收集时间可以控制在一百毫秒以内，对于用户来说基本上是无影响的。

ParNew垃圾收集器是Serial的多线程版本，使用多条线程进行垃圾收集。除此之外，和Serial基本相同，

ParNew在多线程收集垃圾时依旧需要\*\*“Stop the World”\*\*。

ParNew可以使用-XX:ParallelGCThreads参数来限制垃圾收集的线程数量。

ParNew收集器在新生代使用复制算法

Parallel Scavenge也是新生代收集器，也同样是多线程的收集器，但是和ParNew不同，

Parallel Scavenge收集器关注的是一个可控制的吞吐量（Throughput）。

所谓吞吐量指的是CPU用于运行代码的时间和CPU总消耗的时间比例。

Parallel Scavenge采用复制算法。

一听名字就知道这是Serial收集器的老年代版本，是单线程收集器，采用标记-整理算法，其余的和新Serial基本相同。

Parallel Scavenge收集器的老年版本，多线程收集器，采用标记-整理算法，也是吞吐量优先。

CMS是老年代垃圾收集器，在收集过程中可以与用户线程并发操作。

它可以与Serial收集器和Parallel New收集器搭配使用。

CMS牺牲了系统的吞吐量来追求收集速度，适合追求垃圾收集速度的服务器上。

**G1是一款针对多处理器大容量内存的服务器端的垃圾收集器**

目标是在实现高吞吐量的同时，尽可能地满足垃圾收集暂停世家你的要求

G1在执行一些Java堆空间中的全区域操作时是和应用程序线程并发进行的，

因此减少了Java堆空间的中断比例

**你们项目用的哪种垃圾回收器组合?**

**spring boot的starter原理?**

starter是springBoot的一个重要部分。

通过starter，我们能够快速的引入一个功能，而无需额外的配置。

同时starter一般还会给我提供预留的自定配置选项，

我们只需要在application.properties中设置相关参数，就可以实现配置的个性化

**如何自定义一个springboot的starter?**

SpringBoot中的starter是一种非常重要的机制，能够抛弃以前繁杂的配置，将其统一集成进starter，

应用者只需要在maven中引入starter依赖，SpringBoot就能自动扫描到要加载的信息并启动相应的默认配置。

starter让我们摆脱了各种依赖库的处理，需要配置各种信息的困扰。

SpringBoot会自动通过classpath路径下的类发现需要的Bean，并注册进IOC容器。

SpringBoot提供了针对日常企业应用研发各种场景的spring-boot-starter依赖模块。

所有这些依赖模块都遵循着约定成俗的默认配置，并允许我们调整这些配置，即遵循“约定大于配置”的理念。

**java中的锁?**

1.乐观锁和悲观锁

乐观锁

顾名思义，就是很乐观，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的

时候会判断一下在此期间别人有没有去更新这个数据，可以使用版本号等机制。乐观锁适用于多读的应

用类型，这样可以提高吞吐量。

悲观锁

顾名思义，就是很悲观，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，

这样别人想拿这个数据就会阻塞，直到它拿到锁。传统的关系型数据库里边就用到了很多这种锁机制，

比如行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁。

2.独享锁/共享锁

独享锁也称独占锁，是指该锁每次只能被一个线程占有，共享锁则可以被多个线程使用。

3.互斥锁/读写锁

互斥锁：

互斥意味着一个锁某一时刻只能被一个线程持有，其它试图获取锁的线程都会被阻塞，

直至当前锁释放，该锁上的其它线程进入就绪状态，准备抢占锁，

读写锁：

包含了上文提到的独享锁/共享锁，写既是独享锁，也是共享锁，读远远大于写的场景非常需要用到它。

相比于传统的锁，读读是不互斥的只有涉及到写才会互斥，这样就比传统锁提高了cpu资源利用率，

可以说读写锁就是为了优化这种场景而设计的。

在DB中：一般读锁也叫共享锁，写锁称为独占锁。

4.行锁/表锁/页锁

MySQL的锁机制比较简单，其最显著的特点是不同的存储引擎支持不同的锁机制。

行锁，表锁，页锁主要是从锁的粒度上来进行划分的.

MySQL大致可归纳为以下3种锁：

表级锁：开销小，加锁快；不会出现死锁；锁定粒度大，发生锁冲突的概率最高，并发度最低。

行级锁：开销大，加锁慢；会出现死锁；锁定粒度最小，发生锁冲突的概率最低，并发度也最高。

页面锁：开销和加锁时间界于表锁和行锁之间；会出现死锁；锁定粒度界于表锁和行锁之间，并发度一般

5.间隙锁

MySQL中当我们用范围条件检索数据，并请求共享或排他锁时，InnoDB会给符合条件的已有数据记录的索引项加锁；

对于键值在条件范围内但并不存在的记录，叫做"间隙(GAP)"。

InnoDB也会对这个"间隙"加锁，这种锁机制就是所谓的间隙锁(Next-Key锁)。

6.公平锁/非公平锁

公平锁：是指多个线程按照申请锁的顺序来获取锁。

非公平锁：不是按照顺序来获取锁，这意味着它存在这线程饥饿或优先级反转问题。

7.偏向锁/轻量级锁/重量级锁

偏向锁:

是指锁一直被一个线程持有，再没有其它线程竞争该锁的情况下，当前线程可自由获取该锁，降低了获取锁的代价。

轻量级锁：

当锁处于偏向锁状态时，这时候有其它线程来尝试获取该锁，则发生了竞争态势，该锁会从偏向升级为轻量级锁，

其它线程会通过自旋来尝试获取锁，不会阻塞从而提高效率。不过需要注意的是，

轻量级锁并不是用来代替重量级锁的，它是状态的一种过渡。

重量级锁：

当锁处于轻量状态的时候，其它线程虽然是自旋，但自旋不会一直持续下去，

当自旋一定次数的时候，还没有获取到锁，就会进入阻塞，该锁膨胀为重量级锁。

重量级锁会让其他申请的线程进入阻塞，性能降低。

8.可重入锁

所谓重入锁，指的是以线程为单位，当一个线程获取对象锁之后，这个线程可以再次获取本对象上的锁，

而其他的线程是不可以的，它的意义在于防范死锁

9.分段锁

分段锁是一种锁的设计，而并不是指某一具体的锁，比如我们常见的ConcurrentHashMap，

它是并发安全高效的集合。 分段锁的目的是细化锁的粒度，尽量避免资源的竞争使线程阻塞。

10.自旋锁

是指当一个线程在获取锁的时候，如果锁已经被其它线程获取，那么该线程将循环等待，然后不断的判断锁是否能够被成功获取，直到获取到锁才会退出循环。

对于互斥锁，如果资源已经被占用，资源申请者只能进入睡眠状态。但是自旋锁不会引起调用者睡眠，如果自旋锁已经被别的执行单元保持，调用者就一直循环在那里看是否该自旋锁的保持者已经释放了锁，”自旋”一词就是因此而得名。

死锁产生的原因是等待需要的资源而一直拿不到，就会导致线程一直处于等待中。比较专业的定义是：一组互相竞争资源的线程因互相等待，导致“永久”阻塞的现象。

并发程序一旦死锁，一般没有特别好的方法，很多时候我们只能重启应用。因此，解决死锁问题最好的办法还是规避死锁。

11.死锁

死锁产生的原因是等待需要的资源而一直拿不到，就会导致线程一直处于等待中。比较专业的定义是：一组互相竞争资源的线程因互相等待，导致“永久”阻塞的现象。

并发程序一旦死锁，一般没有特别好的方法，很多时候我们只能重启应用。因此，解决死锁问题最好的办法还是规避死锁。

12.分布式锁

很多时候我们需要保证一个方法在同一时间内只能被同一个线程执行。在单机环境中可以很容易解决，但是在分布式环境下就比较复杂了。分布式锁有三种方式:数据库，redis,zk

db设计思路：新建一张有method字段的表，method字段创建唯一索引，作插入操作以尝试获取锁，作删除操作以释放锁

redis设计思路：用到 setnx命令，expire，dtt命令。setnx （set if it not exist）的唯一性。

zk设计思路：利用zk的一个根目录下不能有重命子结点去获取锁，释放锁。

**Exrcutors创建线程池有什么缺点?**

**jvm?**

JVM(Java Virtual Machine，Java虚拟机)

Java程序的跨平台特性主要是指字节码文件可以在任何具有Java虚拟机的计算机或者电子设备上运行，

Java虚拟机中的Java解释器负责将字节码文件解释成为特定的机器码进行运行。

因此在运行时，Java源程序需要通过编译器编译成为.class文件。众所周知java.exe是javaclass文件的执行程序，

但实际上java.exe程序只是一个执行的外壳，它会装载jvm.dll

（windows下，下皆以windows平台为例，linux下和solaris下其实类似，为：libjvm.so），

这个动态连接库才是java虚拟机的实际操作处理所在。

JVM是JRE的一部分。它是一个虚构出来的计算机，是通过在实际的计算机上仿真模拟各种计算机功能来实现的。

JVM有自己完善的硬件架构，如处理器、堆栈、寄存器等，还具有相应的指令系统。Java语言最重要的特点就是跨平台运行。

使用JVM就是为了支持与操作系统无关，实现跨平台。所以，JAVA虚拟机JVM是属于JRE的，

而现在我们安装JDK时也附带安装了JRE(当然也可以单独安装JRE)。

**gc?**

GC就是垃圾回收，java这种语言是动态分配内存大小的，并且依靠垃圾回收机制来完成对分配内存空间的回收，

从而来避免内存溢出的问题，也在一定程度上降低了程序员工作的复杂度。

jvm中的GC采用了generation（分代回收）算法，因为大多数的对象存活的时间比较短，而少部分的对象才能够长时间存活。

因此，jvm将堆内存划分为年轻代（young generation）和年老代（old generation）。年轻代中的对象通常建立时间不久，且大部分生命周期也很短；年老代中的对象则已经创建比较久了，

其声明周期也相对年轻代比较长。按照上面的划分，jvm在做GC时也进行了区别对待，对年轻代GC会相对比较频繁，

且采用了copying(复制)算法；年老代的GC相对比较少，且采用的是tracing算法的一种，是标记-清除-压缩

**BIO和NIO说一下？**

BIO，面向流，只能读或者只能写，阻塞IO

NIO，面向缓冲区，可以同时进行读写，非阻塞IO

**Nginx的反向代理？**

实际运行方式是代理服务器接受网络上的连接请求。

它将请求转发给内部网络上的服务器，并将从服务器上得到的结果返回给网络上请求连接的客户端 。

代理服务器和原始服务器处于同一个局域网内。

优点：

反向代理服务器可以隐藏源服务器的存在和特征。

它充当互联网云和web服务器之间的中间层。

这对于安全方面来说是很好的，特别是当您使用web托管服务时。

**你用到过那些设计模式，在什么场景下用到的?**

**为什么要用Eureka?**

Eureka是Netfix的一个子模块,也是核心模块之一. Eureka是一个基于REST的服务,用于定位服务,以实现云端中间层服务发现和故障转移。

服务注册与发现对于微服务架构来说是非常重要的,有了服务发现与注册,只需要使用服务的标识符,就可以访问到服务,而不需要修改服务调用的配置文件了。

功能类似于dubbo的注册中心,比如Zookeeper

**Eureka的自我保护机制?**

1.Eureka在运行期间会统计心跳失败的比例，在15分钟内是否低于85%,如果出现了低于的情况，

Eureka Server会将当前的实例注册信息保护起来，同时提示一个警告，表明进入了保护模式

2.一旦进入保护模式，Eureka Server将会尝试保护其服务注册表中的信息，不再删除服务注册表中的数据。也就是不会注销任何微服务。

3.我们在开发测试阶段，需要频繁地重启发布，如果触发了保护机制，则旧的服务实例没有被删除，

这时请求有可能跑到旧的实例中，而该实例已经关闭了，这就导致请求错误，影响开发测试。

4.自我保护模式可以让集群更加健壮。故障恢复时，自动退出自我保护模式。

5.生产环境，不会频繁重启，所以，一定要把自我保护机制打开，否则网络一旦中断，就无法恢复

**Eureka如何保证数据一致性?**

Eureka满足AP原则，也就是各个节点平等，只要有节点挂掉，会立刻换至其他节点，保证服务可用，只不过查到的信息可能不是最新的。

但是在网络稳定后，当前实例新的注册信息会被同步到其他节点中。而Zookeeper为了保障每刻查询到的结果都是强一致的，

所以当master节点网络故障时，其他节点会重新进行leader选举，问题是选举时间过长，且选举时，这个集群都是不可用的。

**Zuul怎么实现路由的?**

**Zuul怎么拦截请求的?**

**Feign怎么做的请求转发的?**

1. 实现Feign提供的一个接口RequestInterceptor

2.在Feign请求A服务的注解FeignClient加上configuration = FeignConfiguration.class

3.然后就可以在A服务接收到Feign转发过来的请求头信息

**Git冲突怎么解决?**

1.首先我们文件如果有冲突的情况下，文件会显示红色。然后我们鼠标移到项目目录上点击右键，选择到team这个栏上。

这一栏是git的属性和处理方式。然选择到同步工作间也就是synchronize workspace

2.之后我们会进入到解决冲突的界面。然后我们点击双向红箭头的位置，这个意思显示所有有冲突的文件，

这样的目的是让我们更直观的看到文件冲突有些。

3.然后选择我们需要修改冲突的文件，然后鼠标点击右键，选择merge tool，然后右下角会出现两个代码对比界面。

记住左边的代码是服务器最新的代码，而右边就是你自己的代码，

这时候你需要去对比服务器代码和你的代码哪里不一致然后修改放在左边保存。

**什么是fastDFS，在项目中怎么用的?**

FastDFS 是用 c 语言编写的一款开源的分布式文件系统。

FastDFS 为互联网量身定制，充分考虑了冗余备份、负载均衡、线性扩容等机制，并注重高可用、高性能等指标，

使用 FastDFS很容易搭建一套高性能的文件服务器集群提供文件上传、下载等服务。

在项目中怎么使用，比如图片的上传

1） 指定图片绝对地址

2） 指定图片服务器地址；使用 Client.conf 配置文件

3） 编写UploadController

4） 在SpringMVC中添加文件解析器的配置

5）页面文件上传

6）编写FastDFS的简单封装工具

**介绍一下Tcc?**

Tcc是分布式事务框架，用于分布式事务的。

分布式事务就是针对两个以上的库操作数据事务管理的，

比如操作A库B库，当B库失败，也要把A库哪一步操作也要回滚。

Tcc其实是一个模板框架，是英文字母try,confirm,cnacel三个单词的缩写。

我们要操作的业务部分在try里面执行，执行的所有结果是否有失败的情况在confirm里面确认，如果有失败，则在cancel里面做回滚操作。

**Tcc补偿失败怎么处理?**

TCC 补偿失败的主要思想是针对每个操作，都要注册一个与其对应的确认和补偿（撤销）操作。

Try 阶段：尝试执行，完成所有业务检查（一致性），留必须业务资源（准隔离性）。

Confirm 阶段：确认执行，真正执行业务，不作任何业务检查，只使用Try阶段预留的业务资源，

Confirm 操作满足幂等性。要求具备幂等设计，Confirm 失败后需要进行重试。

Cancel 阶段：取消执行，释放 Try 阶段预留的业务资源 ，Cancel操作满足幂等性。

Cancel 阶段的异常和 Confirm 阶段异常处理方案基本上一致。

**mybatis和hibernate的区别?**

（1） hibernate是全自动，而mybatis是半自动

hibernate完全可以通过对象关系模型实现对数据库的操作，拥有完整的JavaBean对象与数据库的映射结构来自动生成sql。

而mybatis仅有基本的字段映射，对象数据以及对象实际关系仍然需要通过手写sql来实现和管理。

（2）hibernate数据库移植性远大于mybatis

hibernate通过它强大的映射结构和hql语言，大大降低了对象与数据库（Oracle、MySQL等）的耦合性，

而mybatis由于需要手写sql，因此与数据库的耦合性直接取决于程序员写sql的方法，

如果sql不具通用性而用了很多某数据库特性的sql语句的话，移植性也会随之降低很多，成本很高。

（3）hibernate拥有完整的日志系统，mybatis则欠缺一些

hibernate日志系统非常健全，涉及广泛，包括：sql记录、关系异常、优化警告、缓存提示、脏数据警告等；而mybatis则除了基本记录功能外，功能薄弱很多。

（4）mybatis相比hibernate需要关心很多细节

hibernate配置要比mybatis复杂的多，学习成本也比mybatis高。

但也正因为mybatis使用简单，才导致它要比hibernate关心很多技术细节。

mybatis由于不用考虑很多细节，开发模式上与传统jdbc区别很小，因此很容易上手并开发项目

但忽略细节会导致项目前期bug较多，因而开发出相对稳定的软件很慢，而开发出软件却很快。

hibernate则正好与之相反。但是如果使用hibernate很熟练的话，实际上开发效率丝毫不差于甚至超越mybatis。

（5）sql直接优化上，mybatis要比hibernate方便很多

由于mybatis的sql都是写在xml里，因此优化sql比hibernate方便很多。

而hibernate的sql很多都是自动生成的，无法直接维护sql；虽有hql，但功能还是不及sql强大，

见到报表等变态需求时，hql也歇菜，也就是说hql是有局限的；hibernate虽然也支持原生sql，但开发模式上却与orm不同，

需要转换思维，因此使用上不是非常方便。总之写sql的灵活度上hibernate不及mybatis。

（6）缓存机制上，hibernate要比mybatis更好一些

MyBatis的二级缓存配置都是在每个具体的表-对象映射中进行详细配置，这样针对不同的表可以自定义不同的缓存机制。

并且Mybatis可以在命名空间中共享相同的缓存配置和实例，通过Cache-ref来实现。

而Hibernate对查询对象有着良好的管理机制，用户无需关心SQL。所以在使用二级缓存时如果出现脏数据，系统会报出错误并提示。

Hibernate的二级缓存配置在SessionFactory生成的配置文件中进行详细配置，然后再在具体的表-对象映射中配置是那种缓存。

而MyBatis在使用二级缓存时需要特别小心。如果不能完全确定数据更新操作的波及范围，避免Cache的盲目使用。否则，脏数据的出现会给系统的正常运行带来很大的隐患。

**重复订单的问题?**

如何防止订单重复提交

1.用户每次进入提交订单页面的时候，调用发号器接口，在提交订单页面生成唯一的序号

2.用户提交订单的时候，传递该序号，然后后端通过redis的原子方法++，做判断，如何大于2，则代表请求重复，对该请求不做处理即可。

3.这样就可以防止用户重复提交订单。

4.并且整个流程中，不是很影响性能。

**TCP三次握手四次挥手?**

三次握手过程理解

第一次握手：主机 A 发送位码为 syn＝1,随机产生 seq number=1234567 的数据包到服务器，主机 B 由 SYN=1 知道，A 要求建立联机；

第二次握手：主机 B 收到请求后要确认联机信息，向 A 发 送 ack number=( 主 机 A 的

seq+1),syn=1,ack=1,随机产生 seq=7654321 的包

第三次握手：主机 A 收到后检查 ack number 是否正确，即第一次发送的 seq number+1,以及位码

ack 是否为 1，若正确，主机 A 会再发送 ack number=(主机 B 的 seq+1),ack=1，主机 B 收到后确认

13/04/2018 Page 163 of 283

seq 值与 ack=1 则连接建立成功.

四次挥手

TCP 建立连接要进行三次握手，而断开连接要进行四次。这是由于 TCP 的半关闭造成的。因为 TCP 连

接是全双工的(即数据可在两个方向上同时传递)所以进行关闭时每个方向上都要单独进行关闭。这个单

方向的关闭就叫半关闭。当一方完成它的数据发送任务，就发送一个 FIN 来向另一方通告将要终止这个

方向的连接。

1） 关闭客户端到服务器的连接：首先客户端 A 发送一个 FIN，用来关闭客户到服务器的数据传送，

然后等待服务器的确认。其中终止标志位 FIN=1，序列号 seq=u

2） 服务器收到这个 FIN，它发回一个 ACK，确认号 ack 为收到的序号加 1。

3） 关闭服务器到客户端的连接：也是发送一个 FIN 给客户端。

4） 客户段收到 FIN 后，并发回一个 ACK 报文确认，并将确认序号 seq 设置为收到序号加 1。

首先进行关闭的一方将执行主动关闭，而另一方执行被动关闭。

**熔断怎么处理?**

1） 启动Eureka-Server及LOADBALANCE-SERVICE，不要启动Bookingcar-Service

点击"LOADBALANCE-SERVICE"后面链接，进入负载均衡验证页面，输入请求参数/v1/lb/testport?name=bookingcar-service，我们会在页面上得到以下的错误提示

2） 在pom.xml里，添加对Hystrix的依赖

3）在ribbon-service入口程序RibbonServiceApplication.java处添加注解@EnableCircuitBreaker，开启熔断器功能

4） 在LoadBalanceService.java里，使用了@Service注解声明了LoadBalanceService类。

我们在LoadBalanceService类里通过@HystrixCommand注解引入对TestPort调用的熔断机制，

@HystrixCommand注解里可以添加回调函数，如@HystrixCommand(callbackCommand = "XXX“），

当无法调用TestPort服务时，熔断机制的回调函数就会发生作用，对错误进行快速处理。

5） 重新启动LOADBALANCE-SERVICE， 再次进入负载均衡验证页面，输入请求参数/v1/lb/testport?name=bookingcar-service，我们发现熔断机制已经发挥作用

**git分支?**

git 分支操作是掌握分布式版本控制系统的基础，当我们创建一个仓库并对仓库进行初始化之后，

该仓库会就会创建一个默认的 master 分支，而这个 master 分支可以被当做是最开始的主分支。

Git 分支的创建不限制于网络，具有本地特性，而且还可以根据我们的需求来创建多个分支，

这些分支可以是为了修复程序中的某个 bug 而存在 ，或者是为了增加当前程序的一些新特性都而使用，等等。

分支的创建能够很好的让我们的操作与开发主线进行隔离，特别是以分布式方式进行的开源项目会涉及到众多的人员，

为了不让我们的修改影响到主线程序，只有通过审核后的代码才能够合并到主线中去，所以通过使用分支是最理想的方法。

**静态化的具体流程?**

**那有过哪些和第三方系统对接的经验?**

**svn是怎么搭建的?**

**nginx几台，怎么搭建的?**

**商品超卖的情况怎么解决?**

**你在项目中遇到过什么印象比较深的问题吗？你是怎么解决的？解决的思路是什么?**

**如果你的经理给了你一个需求让你实现，你需要考虑到什么?**

**创建多个订单的时候怎么保证线程安全的?**

**订单的幂等性怎么保证?**

**创建订单怎么同时减少库存?**

**如何防止页面被人恶意攻击?**