# 基础

## 为什么说使用枚举来实现单例是最好的选择

枚举可解决线程安全问题：枚举在经过javac的编译之后，会被转换成形如public final class T extends Enum的定义。

枚举可避免反序列化破坏单例：在序列化的时候Java仅仅是将枚举对象的name属性输出到结果中，反序列化的时候则是通过java.lang.Enum的valueOf方法来根据名字查找枚举对象。同时，编译器是不允许任何对这种序列化机制的定制的，因此禁用了writeObject、readObject等方法。普通的Java类的反序列化过程中，会通过反射调用类的默认构造函数来初始化对象。所以，即使单例中构造函数是私有的，也会被反射给破坏掉。由于反序列化后的对象是重新new出来的，所以这就破坏了单例。但是，枚举的反序列化并不是通过反射实现的。所以，也就不会发生由于反序列化导致的单例破坏问题。

## 常用的集合主要在java.util包下面：

主要接口有List,Set,Map,Queue,Deque,SortedMap,SortedSet

List:

线程不安全：ArrayList, LinkedList, AttributeList, RoleList

线程安全：CopyOnWriteArraylist ,Stack, Vector

Set:

线程不安全：EnumSet, HashSet, LinkedHashSet, TreeSet

线程安全：ConcurrentSkipListSet, CopyOnWriteArraySet

Map:

线程不安全：EnumMap, HashMap, IdentityHashMap, LinkedHashMap, Properties, TreeMap

线程安全：ConcurrentHashMap, ConcurrentSkipListMap, Hashtable

SortedMap继承于map:

ConcurrentSkipListMap, TreeMap

SortedSet继承于set:

ConcurrentSkipListSet, TreeSet

queue:

PriorityQueue, （优先级队列）

ConcurrentLinkedQueue, （基于链表的并发队列）

DelayQueue, （延期阻塞队列）（阻塞队列实现了BlockingQueue接口）

ArrayBlockingQueue, （基于数组的并发阻塞队列）

LinkedBlockingQueue, （基于链表的FIFO阻塞队列）

PriorityBlockingQueue, （带优先级的无界阻塞队列）

SynchronousQueue （并发同步阻塞队列）

Deque:

LinkedBlockingDeque, （基于链表的FIFO双端阻塞队列）

ArrayDeque, （数组双端队列）

## JVM内存划分

JDK1.7时JVM内存划分为程序计数器，虚拟机栈，本地方法栈，堆，方法区，JDK1.8将方法区移入到了直接内存中，称为元空间。

**程序计数器**主要有两个作用：在多线程的情况下，程序计数器用于记录当前线程执行的位置。

字节码解释器通过改变程序计数器来依次读取指令，从而实现代码的流程控制，如：顺序执行、选择、循环、异常处理。

**虚拟机栈：**用于描述Java方法执行的状态，每个方法在执行的同时都会创建一个栈帧用于存储局部变量表，操作数栈，动态链接，方法出口等信息。

局部变量表主要存放了编译器可知的各种数据类型（boolean、byte、char、short、int、float、long、double）、对象引用（reference类型，它不同于对象本身，可能是一个指向对象起始地址的引用指针，也可能是指向一个代表对象的句柄或其他与此对象相关的位置）。

Java方法有两种返回方式：return 语句，抛出异常

**本地方法栈：**虚拟机栈的作用非常相似，区别是：虚拟机栈为虚拟机执行Java方法（也就是字节码）服务，而本地方法栈则为虚拟机使用到的Native方法。

**堆：**在虚拟机启动时创建，唯一目的就是存放对象实例，所有的对象实例以及数组都要在堆上分配。Java堆还可以细分为：新生代和老年代：再细致一点有：eden区、s0区、s1区都属于新生代，tentired 区属于老年代。

**方法区：**是各个线程共享内存区域，用于存储已被虚拟机加载的类信息，常量，静态变量，即时编译器编译后的代码等数据。这个区域的内存回收目标主要是针对常量池的回收和对类型的卸载。HotSpot 虚拟机中方法区也常被称为 “永久代。

**直接内存**

直接内存并不是虚拟机运行时数据区的一部分，也不是虚拟机规范中定义的内存区域，但是这部分内存也被频繁地使用。而且也可能导致 OutOfMemoryError 异常出现。

## Java内存模型

Java内存模型（Java Memory Model ,JMM）就是一种符合内存模型规范的，目的是解决由于多线程通过共享内存进行通信时，存在的原子性、可见性（缓存一致性）以及有序性问题。

Java内存模型规定了所有的变量都存储在主内存中，每条线程还有自己的工作内存，线程的工作内存中保存的变量是主内存中拷贝过来的，线程对变量的所有操作都必须在工作内存中进行，而不能直接读写主内存。不同的线程之间也无法直接访问对方工作内存中的变量，线程间变量的传递均需要自己的工作内存和主存之间进行数据同步进行。

而JMM就作用于工作内存和主存之间数据同步过程。他规定了如何做数据同步以及什么时候做数据同步。

**Java内存模型的实现**

在Java中提供了一系列和并发处理相关的关键字，比如volatile、synchronized、final、concurren包等。其实这些就是Java内存模型封装了底层的实现后提供给程序员使用的一些关键字。

## JVM调优

调优工具：Jconsole,jProfile,visualVM

如何调优：

堆信息查看：

可查看堆空间大小分配（年轻代，年老代，持久代）

提供即时的垃圾回收功能

垃圾监控

有了堆信息查看方面的功能，我们一般可以顺利解决以下问题：

年老代年轻代大小划分是否合理

内存泄漏

垃圾回收算法设置是否合理

查看FullGC的频率

Linux使用jstat命令查看jvm的GC情况，通常运行命令如下：

jstat -gc 12538 5000

即会每5秒一次显示进程号为12538的java进成的GC情况

性能调优建议： jvm调优没有一个固定模板配置说必须如何操作，它需要根据系统的情况不同对待。 但是可以有如下建议：

1、初始化内存和最大内存尽量保持一致，避免内存不够用继续扩充内存。最大内存不要超过物理内存，例如内存8g，你可以设置最大内存4g/6g但是不能超过8g否则加载类的时候没有空间会报错。

2、gc/full gc频率不要太高、每次gc时间不要太长、根据系统应用来定。

## 对象在内存中的初始化过程

而 JVM 虚拟机执行 class 字节码的过程可以分为七个阶段：加载、验证、准备、解析、初始化、使用、卸载。

类的加载指的是将类的.class文件以二进制数据读入到内存中，将其放在运行时数据区的方法区内，然后在堆区创建一个Class对象，用来封装类在方法区内的数据结构。然后会对文件进行JVM规范校验，代码逻辑校验。校验通过以后，会在堆中为类的静态变量及成员变量分配内存，并将其初始化为默认值，接着执行构造代码块，对对象进行初始化，在堆中为类的对象分配内存。

private static int a = 3;在准备阶段会被a设为默认值0，执行初始化时把3赋予了a。静态代码块从上到下执行

变量覆盖：ParentClass subclass = new SubClass();

子类跟父类有相同普通变量的情况下，subClass.protectedField输出的是父类定义的变量

**类的初始化顺序**

1. 父类--静态变量

2. 父类--静态初始化块

3. 子类--静态变量

4. 子类--静态初始化块

5. 父类--变量

6. 父类--初始化块

7. 父类--构造器

8. 子类--变量

9. 子类--初始化块

10.子类--构造器

## 类加载机制

Java 虚拟机自带的加载器：根类加载器；扩展类加载器；系统类加载器

用户自定义的类加载器必须继承java.lang.ClassLoader类

自定义类加载器分为以下几个步骤：

1、定义类。继承ClassLoader

2、重写ClassLoader的 findClass方法

ClassLoader 里面有三个重要的方法 loadClass()、findClass() 和 defineClass()。

loadClass() 方法是加载目标类的入口，它首先会查找当前 ClassLoader 以及它的双亲里面是否已经加载了目标类，如果没有找到就会让双亲尝试加载，如果双亲都加载不了，就会调用 findClass() 让自定义加载器自己来加载目标类。ClassLoader 的 findClass() 方法是需要子类来覆盖的，不同的加载器将使用不同的逻辑来获取目标类的字节码。拿到这个字节码之后再调用 defineClass() 方法将字节码转换成 Class 对象。

自定义类加载器不易破坏双亲委派规则，不要轻易覆盖 loadClass 方法。否则可能会导致自定义加载器无法加载内置的核心类库。在使用自定义加载器时，要明确好它的父加载器是谁，将父加载器通过子类的构造器传入。如果父类加载器是 null，那就表示父加载器是「根加载器」。

**什么是双亲委派机制？**

双亲委派机制的前提是双亲委派模型，它要求除顶层启动类加载器外其余类加载器都应该有自己的父类加载器；类加载器之间通过复用关系来复用父加载器的代码。当需要加载一个类时，首先会检查请求加载的类是否已经被加载过；若没有被加载过：递归调用父类加载器的loadClass()；方法进行加载，如果父类加载器和启动类加载器均无法加载请求，则调用自身的加载功能。

实现双亲委派模型的代码都集中在java.lang.ClassLoader的loadClass()方法中；

**双亲委派模型的优点：**

Java类伴随其类加载器具备了带有优先级的层次关系，确保了在各种加载环境的加载顺序。

保证了运行的安全性，防止不可信类扮演可信任的类。

**什么情况下我们需要破坏双亲委派模型；**

双亲委派模型很好地解决了各个类加载器的基础类统一问题(越基础的类由越上层的加载器进行加载)，基础类之所以被称为“基础”，是因为它们总是作为被调用代码调用的API。但是，如果基础类又要调用用户的代码，那该怎么办呢。

## 什么时候一个对象会被GC？

被GC的对象首先会被标记为是否可回收，一般采用有向图的方式记录和管理堆(heap)中的所有对象，从根节点向下检索，不能检索到的则视为这个对象不可用，并标记为可回收。标记后的对象并不会立即回收，对象没有覆盖finalize()方法,或者finalize()方法已经被虚拟机调用过此对象会被直接回收。否则这个对象将会放置在一个队列之中,并在稍后由一个由虚拟机线程去执行其finalize()方法，做一些清除前的工作，例如资源释放等；直至下一次垃圾回收动作发生时才会真正回收对象占用的内存空间。

任何一个对象的finalize()方法都只会被系统自动调用一次,如果对象面临下一次回收,它的finalize()方法不会被再次执行

**垃圾收集器**

垃圾回收器的基本原理是什么？垃圾回收器可以马上回收内存吗？有什么办法主动通知虚拟机进行垃圾回收

答： 对于GC来说，当程序员创建对象时，GC就开始监控这个对象的地址、大小以及使用情况。通常，GC采用有向图的方式记录和管理堆(heap)中的所有对象。通过这种方式确定哪些对象是"可达的"，哪些对象是"不可达的"。当GC确定一些对象为"不可达"时，GC就有责任回收这些内存空间。可以。程序员可以手动执行System.gc()，通知GC运行，但是Java语言规范并不保证GC一定会执行。

新生代的垃圾回收器包括Serial、ParNew、Parallel Scavenge，老年代的垃圾回收器包括CMS、Serial Old、Parallel Old。其中新生代的三种垃圾回收器都采用了复制算法，并且会触发STOP THE WORLD。CMS使用标记清除算法，Serial Old、Parallel Old使用标记整理算法。

**Serial收集器**是一个**单线程收集器**（-XX：+UseSerialGC），这个“单线程”不只是说它只会使用一个CPU或者一条线程去完成垃圾收集工作，更重要的是在它进行垃圾收集时，必须暂停其他所有的工作线程，直到它垃圾收集结束。

**ParNew收集器**是Serial**收集器的多线程**版本（-XX:+UseParNewGC -XX:ParallelGCThreads），它能够与CMS收集器配合工作，因此，在运行在Server模式下的虚拟机中，ParNew收集器是首选的新生代收集器。

**Parallel Scavenge**这也是一个并行的新生代垃圾收集器，不同于其他收集器（以尽可能缩短垃圾收集时用户线程的停顿时间为目的），它是唯一一个以达到一个可控制的吞吐量为目标的垃圾收集器。

**CMS收集器**的GC周期由6个阶段组成。其中4个阶段(名字以Concurrent开始的)与实际的应用程序是并发执行的，而其他2个阶段需要暂停应用程序线程。

初始标记：为了收集应用程序的对象引用需要暂停应用程序线程，该阶段完成后，应用程序线程再次启动。

并发标记：从第一阶段收集到的对象引用开始，遍历所有其他的对象引用。

并发预清理：改变当运行第二阶段时，由应用程序线程产生的对象引用，以更新第二阶段的结果。

重标记：由于第三阶段是并发的，对象引用可能会发生进一步改变。因此，应用程序线程会再一次被暂停以更新这些变化，并且在进行实际的清理之前确保一个正确的对象引用视图。这一阶段十分重要，因为必须避免收集到仍被引用的对象。

并发清理：所有不再被应用的对象将从堆里清除掉。

并发重置：收集器做一些收尾的工作，以便下一次GC周期能有一个干净的状态。

一个常见的误解是,CMS收集器运行是完全与应用程序并发的。我们已经看到，事实并非如此，即使“stop-the-world”阶段相对于并发阶段的时间很短。

应该指出，尽管CMS收集器为老年代垃圾回收提供了几乎完全并发的解决方案，然而年轻代仍然通过“stop-the-world”方法来进行收集。对于交互式应用，停顿也是可接受的，背后的原理是年轻代的垃圾回收时间通常是相当短的。

CMS收集器，我们会面临两个主要的挑战，可能需要进行调优：堆碎片;对象分配率高

CMS收集器并没有任何碎片整理的机制。

第二个挑战就是应用的对象分配率高。如果获取对象实例的频率高于收集器清除堆里死对象的频率，并发算法将再次失败。

G1算法：

G1把整个堆划分为一个一个等大小的区域（region）。内存的回收和划分都以region为单位；

增量收集把对堆空间划分成一系列内存块，使用时，先使用其中一部分（不会全部用完），垃圾收集时 把之前用掉的部分中的存活对象再放到后面没有用的空间中

回收步骤：

初始标记（Initial Marking） stop the world

并发标记（Concurrent Marking）

最终标记暂停（Final Marking Pause） stop the world

存活对象计算及清除（Live Data Counting and Cleanup）

**GC机制和原理**

**GC分哪两种， Minor GC 和Full GC有什么区别？什么时候会触发Full GC？**

Minor GC:

Eden区域满了，或者新创建的对象大小 > Eden所剩空间

当JVM无法为一个新的对象分配空间时会触发Minor GC，比如当Eden区满了。

执行Minor GC操作时，不会影响到永久代。

质疑常规的认知，所有的Minor GC都会触发“全世界的暂停(stop-the-world)”，停止应用程序的线程。对于大部分应用程序，停顿导致的延迟都是可以忽略不计的。

Major GC:清理永久代，但是由于很多MojorGC 是由MinorGC 触发的，所以有时候很难将MajorGC 和MinorGC区分开。

FullGC：是清理整个堆空间—包括年轻代和永久代。Full GC的时候会先触发Minor GC。

新生代直接晋升到老年代的大对象超过了老年代的剩余空间，引发Full GC。

Perm永久代空间不足会触发Full GC，可以让CMS清理永久代的空间。

System.gc()引起的Full GC，可以设置DisableExplicitGC来禁止调用System.gc引发Full GC

**分别采用什么算法？**

Eden跟Survivor0,Survivor1采用的垃圾回收方式是著名的**“停止-复制（Stop-and-copy）”**清理法（将Eden区和一个Survivor中仍然存活的对象拷贝到另一个Survivor中）

老年代用的算法是标记-整理算法（串行收集器）

关于方法区即永久代的回收，永久代的回收有两种：常量池中的常量，无用的类信息，常量的回收很简单，没有引用了就可以被回收。对于无用的类进行回收，必须保证3点：

1. 类的索引实例都已经被回收。2.加载类的ClassLoader已经被回收。3.类对象的Class对象没有被引用（即没有通过反射引用该类的地方）

**YGC的时候，有些对象可能会发生晋升，如果晋升失败怎么处理？**

标记阶段时，会把对象和对应的对象头数据保存在两个栈中，如果晋升失败的话，就把该对象的对象头复原，然后提早进行一次老年代的回收，防止下次进行YGC的时候发生晋升失败

## 内存泄漏的排查案例

把堆dump下来再用MAT等工具进行分析，也可以用更轻量级的在线分析，用jmap查看存活的对象情况（jmap -histo:live [pid]），定位到一个HashTable有问题后用神器btrace跟踪Hashtable.put调用的堆栈。

## jvm中有以下三种锁(由上到下越来越“重量级”)：偏向锁,轻量级锁,重量级锁

**偏向锁**：适用于只有一个线程访问同步块场景。加锁和解锁不需要额外的消耗。

**轻量级锁**： 可以有多个线程交替进入临界区，在竞争不激烈的时候，稍微自旋等待一下就能获得锁。这也是CAS的实现方式。竞争的线程不会阻塞，提高了程序的响应速度。但是会消耗CPU

**重量级锁**：比如Synchronized它会利用java对象的监视器锁（monitor）实现，同一临界区内只允许一个线程操作。线程竞争不使用自旋，不会消耗CPU。

## CAS原理及缺点

CAS是通过unsafe类的compareAndSwap方法实现的， 它将内存位置的内容与给定值进行比较，只有在相同的情况下，将该内存位置的内容修改为新的给定值。

compareAndSwap方法的第一个参数是要修改的对象，第二个参数是对象中要修改变量的偏移量，第三个参数是修改之前的值，第四个参数是预想修改后的值。通过将偏移量与修改之前的值做比较，则将将偏移量修改为预想修改的值，并返回true.

当然CAS一定要volatile变量配合

unsafe.compareAndSwapInt(this, valueOffset, expect, update);

compareAndSet这个方法主要调用unsafe.compareAndSwapInt这个方法，这个方法有四个参数，其中第一个参数为需要改变的对象，第二个为偏移量(即之前求出来的valueOffset的值)，第三个参数为期待的值，第四个为更新后的值。整个方法的作用即为若调用该方法时，value的值与expect这个值相等，那么则将value修改为update这个值，并返回一个true，如果调用该方法时，value的值与expect这个值不相等，那么不做任何操作，并范围一个false。

**CAS的缺点：**

1.CPU开销较大：在并发量比较高的情况下，如果许多线程反复尝试更新某一个变量，却又一直更新不成功，循环往复，会给CPU带来很大的压力。

2.不能保证代码块的原子性：CAS机制所保证的只是一个变量的原子性操作，而不能保证整个代码块的原子性。比如需要保证3个变量共同进行原子性的更新，就不得不使用Synchronized了。

3.ABA问题：这是CAS机制最大的问题所在。

如线程1从内存X中取出A，这时候另一个线程2也从内存X中取出A，并且线程2进行了一些操作将内存X中的值变成了B，然后线程2又将内存X中的数据变成A，这时候线程1进行CAS操作发现内存X中仍然是A，然后线程1操作成功。虽然线程1的CAS操作成功，但是整个过程就是有问题的。比如链表的头在变化了两次后恢复了原值，但是不代表链表就没有变化。

所以JAVA中提供了AtomicStampedReference/AtomicMarkableReference来处理会发生ABA问题的场景，主要是在对象中额外再增加一个标记来标识对象是否有过变更。

**共享与独占的区别在于：**

CHL队列中的节点的模式是独占还是共享，当一个线程成功修改了stat状态，表示获取了锁，如果线程所在的节点为SHARED，将开始一个读锁传递的过程，从头结点，向队列后续节点传递唤醒，直到队列结束或者遇到了EXCLUSIVE的节点，等待所有激活的读操作完成，然后进入到独享模式

公平与非公平的区别就在于线程第一次获取锁时，也就是执行修改stat操作时，是进队列还是直接修改状态，这是基本的工作机制

## 死锁产生的四个条件：

1）互斥。共享资源同时只能被一个线程访问。

2）占有且等待。线程T1在取得共享资源A的时候，请求等待资源B的时候并不释放资源A。

3）不可抢占。其他线程不能强行抢占线程的资源。

4）循环等待条件。线程T1在持有资源A1，同时在请求等待获取资源B，线程T2在持有资源B,然后在请求等待线程T1的持有资源，形成了交叉闭环申请。

处理死锁的方法可有以下4种：

1）死锁预防。由于互斥这个条件无法避免。操作系统本身所具有的特点。可对死锁产生的其他三个条件进行破坏。首先条件2，占有且等待，可以一次性申请所有的资源，可以破坏掉占有且等待。条件三不可抢占。当线程去请求其他资源时，如果获取不到锁，可以主动释放自己的锁，这样不可抢占的条件也被破坏掉了。条件四循环等待条件。可以对申请的资源进行编号，按序访问，这样线性的去申请资源，则不会造成交叉循环。

2）死锁避免。就不用去破坏形成死锁的其他条件。在资源动态分配的过程中。用某种方法判断防止进入不安全状态。从而避免发生死锁。可以使用银行家算法。死锁避免的算法会导致系统开销的增加。

3）死锁检测。死锁预防和死锁检测都是死锁发生之前的预防策略。死锁检测是通过系统设置的检测机构及时的判断当前系统是否处于死锁状态，并精确的确定当前死锁相关的进程和资源，执行死锁解除策略

4）死锁解除。这是与死锁检测结合使用的。它使用的方式是剥夺。就是将进程所占有的资源强行收回，分配给其他进程。

避免死锁最简单的方法就是阻止循环等待条件，将系统中所有的资源设置标志位、排序，规定所有的进程申请资源必须以一定的顺序（升序或降序）做操作来避免死锁。

**死锁排查**

1 、 使用top命令查看cpu占用资源较高的PID

2、 通过jps 找到当前用户下的java程序PID：执行 jps -l 能够打印出所有的应用的PID，找到有一个PID和这个cpu使用100%一样的ID！！就知道是哪一个服务了。

3、 使用 pidstat -p 1 3 -u -t

4 、 找到cpu占用较高的线程TID

5、 将TID转换为十六进制的表示方式

6、 通过jstack -l 输出当前进程的线程信息

7、 查找 TID对应的线程(输出的线程id为十六进制)，找到对应的代码

**Synchronized实现原理**

Synchronized的作用主要有三个：（1）确保线程互斥的访问同步代码（2）保证共享变量的修改能够及时可见（3）有效解决重排序问题。

同步代码块synchronized(对象)原理:每个对象有一个监视器锁（monitor）。当monitor被占用时就会处于锁定状态，线程会执行monitorenter指令时尝试获取monitor的所有权，通过monitorexit释放monitor所有权。

同步方法原理：

不过相对于普通方法，其常量池中多了ACC\_SYNCHRONIZED标示符。JVM就是根据该标示符来实现方法的同步的：当方法调用时，调用指令将会检查方法的 ACC\_SYNCHRONIZED 访问标志是否被设置，如果设置了，执行线程将先获取monitor，获取成功之后才能执行方法体，方法执行完后再释放monitor。在方法执行期间，其他任何线程都无法再获得同一个monitor对象。 其实本质上没有区别，只是方法的同步是一种隐式的方式来实现，无需通过字节码来完成。

## Lock与synchronized的区别

1. Lock的加锁和解锁都是由java代码配合native方法（调用操作系统的相关方法）实现的，而synchronize的加锁和解锁的过程是由JVM管理的

2. 当一个线程使用synchronize获取锁时，若锁被其他线程占用着，那么当前只能被阻塞，直到成功获取锁。而Lock则提供超时锁和可中断等更加灵活的方式，在未能获取锁的条件下提供一种退出的机制。

3. 一个锁内部可以有多个Condition实例，即有多路条件队列，而synchronize只有一路条件队列；同样Condition也提供灵活的阻塞方式，在未获得通知之前可以通过中断线程以    及设置等待时限等方式退出条件队列。

4. synchronize对线程的同步仅提供独占模式，而Lock即可以提供独占模式，也可以提供共享模式

Lock类

lock: 在 java.util.concurrent 包内。共有三个实现：

ReentrantLock

ReentrantReadWriteLock.ReadLock

ReentrantReadWriteLock.WriteLock

主要目的是和 synchronized 一样， 两者都是为了解决同步问题，处理资源争端而产生的技术。功能类似但有一些区别。

区别如下：

1.lock 更灵活，可以自由定义多把锁的枷锁解锁顺（synchronized 要按照先加的后解顺序）

2.提供多种加锁方案，lock 阻塞式, trylock 无阻塞式, lockInterruptily 可打断式， 还有 trylock 的带超时时间版本

3.本质上和监视器锁（即 synchronized 是一样的）

4.能力越大，责任越大，必须控制好加锁和解锁，否则会导致灾难。

5.和 Condition 类的结合。

6.性能更高，对比如下图：

## synchronized和ReentrantLock的区别

synchronized是关键字，ReentrantLock是类，这是二者的本质区别。

ReentrantLock比synchronized的扩展性体现在几点上：

（1）ReentrantLock可以对获取锁的等待时间进行设置，这样就避免了死锁

（2）ReentrantLock可以获取各种锁的信息

（3）ReentrantLock可以灵活地实现多路通知

另外，二者的锁机制其实也是不一样的。ReentrantLock底层调用的是Unsafe的park方法加锁，synchronized操作的应该是对象头中mark word，这点我不能确定。

## ThreadPoolExecutor

ThreadPoolExecutor 继承 AbstractExecutorService；AbstractExecutorService 实现 ExecutorService， ExecutorService 继承 Executor

public ThreadPoolExecutor(int corePoolSize,

int maximumPoolSize,

long keepAliveTime,

TimeUnit unit,

BlockingQueue<Runnable> workQueue,

ThreadFactory threadFactory,

RejectedExecutionHandler handler) {...}

corePoolSize : 核心线程数的大小

maximumPoolSize : 线程池中允许的最大线程数

keepAliveTime : 空闲线程允许的最大的存活时间

unit : 存活时间的单位

workQueue : 阻塞任务队列

threadFactory : 线程工厂用来创建线程

handler : 拒绝策略，针对当队列满了时新来任务的处理方式

**线程池的执行过程：**

在刚刚创建线程池的时候，内部线程的数量是 0，当首个任务进行添加的时候，会根据参数的配置进行线程的创建，并随着任务数的增加，会逐渐创建新的线程直到线程数达到核心线程的大小。这时，如果再添加任务会将任务放置到阻塞队列当中,如果队列装不下了这时会判断当前线程的数量是否达到了最大线程数，如果未达到，新添加的任务会由最大重新创建线程并立马执行。如果队列满了，而且已经达到最大线程数了，这时再添加任何就会由拒绝策略来处理，默认的拒绝策略是抛出异常。核心线程数不会被关闭，会一直等待任务到来。

## 线程池原理

原文：<https://mp.weixin.qq.com/s/LExpStGw7XXp0M1E_T0yKA>

https://mp.weixin.qq.com/s/DJPuYVhRGRXkkUGMuDdrSw

根据源码可以发现整个线程池大致分为 3 个部分，1. 是创建 worker 线程，2. 添加任务到 workQueue; 3.worker 线程执行具体任务

线程池每个线程会包装成worker，用来循环执行take任务。

第一步是生产者，也就是任务提供者他执行了一个 execute() 方法，本质上就是往这个内部队列里放了一个任务。

之前已经创建好了的 Worker 线程会执行一个 while 循环 ---> 不停的从这个 内部队列里获取任务。(这一步是竞争的关系，都会抢着从队列里获取任务，由这个队列内部实现了线程安全。)

获取得到一个任务后，其实也就是拿到了一个 Runnable 对象(也就是 execute(Runnabletask) 这里所提交的任务)，接着执行这个 Runnable 的 run() 方法，而不是 start()，为什么线程池在调度的时候执行的是 Runnable 的 run() 方法，而不是 start() 方法呢？线程池这个场景中却恰好就是要利用它只是一个普通方法调用。如果我们往一个核心、最大线程数为 2 的线程池里丢了 1000 个任务，那么它会额外的创建 1000 个线程，同时每个任务都是异步执行的，一下子就执行完毕了。

所谓线程池本质是一个hashSet。多余的任务会放在阻塞队列中。

只有当阻塞队列满了后，才会触发非核心线程的创建。所以非核心线程只是临时过来打杂的。直到空闲了，然后自己关闭了。

线程池提供了两个钩子（beforeExecute，afterExecute）给我们，我们继承线程池，在执行任务前后做一些事情。

线程池原理关键技术：锁（lock,cas）、阻塞队列、hashSet（资源池）

## 线程池遇到未捕获异常会怎样？

Java中线程执行的任务接口java.lang.Runnable如果 run() 方法中抛出了RuntimeExceptio将会怎么处理了？通常java.lang.Thread对象运行设置一个默认的异常处理方法：java.lang.Thread.setDefaultUncaughtExceptionHandler(UncaughtExceptionHandler)而这个默认的静态全局的异常捕获方法是直接输出异常堆栈。当然，我们可以覆盖此默认实现，只需要一个自定义的java.lang.Thread.UncaughtExceptionHandler接口实现即可。

而在线程池中却比较特殊。默认情况下，线程池 java.util.concurrent.ThreadPoolExecutor 会Catch住所有异常， 当任务执行完成(java.util.concurrent.ExecutorService.submit(Callable))获取其结果 时(java.util.concurrent.Future.get())会抛出此RuntimeException。也就是说当我们想线程池 ThreadPoolExecutor(java.util.concurrent.ExecutorService)提交任务时， 如果不理会任务结果（Feture.get()），那么此异常将被线程池吃掉。可以通过继承ThreadPoolExecutor并覆盖ThreadPoolExecutor.afterExecute 方法，我们才能捕获到任务的异常(RuntimeException)。

线程池有未抓取异常时不会导致后续任务中断，但是没有异常信息的输出，难以定位问题，对此，有三种方案：

）自己封装一个RunnableWrapper、CallableWrapper，将传入的任务包裹其中，在run方法中try-catch，并统一对未抓取异常进行异常处理/输出；

）继承ThreadPoolExecutor的方式来创建线程池，重写其afterExecute(Thread, Throwable)方法，判断第二个参数非空时获得其异常信息，统一进行处理；（注意：仅对 executor.execute(..) 启动的任务有效，executor.submit(..)的无效）

）创建线程池时，传入自定义的ThreadFactory，在其创建线程时，使用setUncaughtExceptionHandler(..)设置每个线程的未抓取异常处理句柄，统一进行处理；（注意：仅对 executor.execute(..) 启动的任务有效，executor.submit(..)的无效）

## 线程间有哪些通信方式？

线程间的通信模式有多种,可以是等待通知模式，共享内存模式，管道通信模式

等待通知模式可以使用wait() 和通知 notify() 方法来进行线程间通讯，也可以使用join()方法，在 join 线程完成后会调用 notifyAll() 方法，是在 JVM 实现中调用。。

共享内存模式可以使用volatile共享内存，也可以使用类似于CountDownLatch，CyclicBarrier，Semaphore这样的并发工具进行通信。

管道通信模式可以使用PipedWriter，PipedReader，writer.connect(reader)

也可以使用阻塞队列BlockingQueue来共享数据

## ThreadLocal

实现：每个Thread 都持有一个 TreadLocalMap 类型的变量（该类是一个轻量级的 Map，功能与 map 一样，区别是桶里放的是 entry 而不是 entry 的链表。功能还是一个 map 。）以本身为 key，以目标为 value。

主要方法是 get() 和 set(T a)，set 之后在 map 里维护一个threadLocal -> a，get 时将 a 返回。ThreadLocal 是一个特殊的容器。

# 框架及服务

## 代理商管理系统总结

代理商管理系统操作流程

从kafka获取订单数据（订单数据会重复推送），重复推送的数据会先查询缓存有没有条数据，有的话直接以这个订单id为key，设置分布式锁，查询数据库是否有这条数据，并且判断订单状态为未下单，邮编匹配（根据地址进行匹配），代理选择（根据国家选择代理，然后再根据地区选择合适的代理公司），更新数据库，释放锁。如果这个环节邮编未能匹配到或未选择到代理，之后还可以手动选择，下单（下单会根据选择的代理商调用对应的下单流程，点击下单后，先是获取分布式锁，获取失败直接返回失败，成功的话首先更新订单状态，然后调用邮编，地址校验接口，组装json数据的报文，进行下单，下单成功后保存账单及标签数据，修改订单状态）。

## 慧眼风控系统总结：

**项目描述：**

线下贷前审批系统，项目包含app端，pc端及后端，主要包含用户管理，系统管理，订单管理，日常管理，贷款营销，贷款调查，贷款审核，贷款审批等模块

系统管理：主要包含设备轨迹管理。

订单管理：可以查看订单流程，可查询和操作订单状态。

日常管理：包括月度目标划分，工作日清等。工作分配一般就是贷款调查分配，工作日清包括工作成果和工作总结录入。

贷款营销：包括客户申请登记，征信查询，调查预约及任务分配。

贷款调查：分为贷款问卷，调查详情，贷款评分，审批报告，多媒体信息采集模块

贷款问卷：根据不同行业为用户提供标准的调查模型，调查分为借款人及企业信息、担保人信息、行业信息三大部分组成，调查以问卷的方式进行标准化作业。借款主体问卷包括个人姓名住址，名下是否有车辆或房产，家里有多少人，家庭收入大约多少之类的，企业信息一般就是企业信息。行业信息比如建材这一行业，可以问从事这一行的时间，淡旺季的月销售额，库存，销售模式，进货量等情况。

调查详情：调查详情是对调查问卷的整理和还原，整理后的信息以可编辑表格的形式呈现，用户可通过整理后表格对客户调查信息进行核对和修改。

贷款评分：根据调查问卷、调查详情、贷款申请等信息使用系统中的分析模型进行解析，获得“贷款综合评分”、“综合偏离度”、“建议最高额度”三项结果,并对风险点、偏离度超过限制的数据项进行提示，为贷款审批提供参考依据。比如说个人这一方面也有评分，男的评分比女的高，已婚的评分比未婚的高，评分规则可以修改，并且每一项占总评分的比例也可以修改。

审批报告：系统根据贷款详情自动生成电子版调查报告（pdf版）

**主要负责**用户管理，订单管理，贷款调查模块（贷款问卷，调查详情，贷款评分，审批报告）。

**项目主要流程**：订单录入，然后调查预约，调查预约后会有专门的客户经理上门进行调查，调查以问卷的方式进行，问卷内容是根据不同行业为用户提供标准的调查模型，它主要分为三个模块：借款人及企业信息、担保人信息、行业信息。借款主体问卷包括个人姓名住址，名下是否有车辆或房产，家里有多少人，家庭收入大约多少之类的，企业信息一般就是企业信息。行业信息比如建材这一行业，可以问从事这一行的时间，淡旺季的月销售额，库存，销售模式，进货量等情况。问卷信息会持久化到数据库，然后在问卷提交的时候做一些整理到调查详情里面。调查详情是对调查问卷的整理和还原，整理后的信息以可编辑表格的形式呈现，用户可通过整理后表格对客户调查信息进行核对和修改。一般做调查问卷时会从客户那里拿一些纸质的证明，比如银行流水，出货订单这些，这些都可以在调查详情里面录入。贷款详情之后就是贷款评分环节，根据调查问卷、调查详情、贷款申请等信息使用系统中的分析模型进行解析，获得“贷款综合评分”、“综合偏离度”、“建议最高额度”三项结果,并对风险点、偏离度超过限制的数据项进行提示，为贷款审批提供参考依据。比如说个人这一方面也有评分，男的评分比女的高，已婚的评分比未婚的高，评分规则可以修改，并且每一项占总评分的比例也可以修改。

最后就是整理成审批报告，审批人审批之后会到放款中心，这个系统的流程就结束了。

## 代码中有用到什么设计模式

**单例模式**

**装饰者模式**有用到，一般将发送和接收请求写一个基础类，再有一个类用于拼装和解析请求参数。

还有像spring使用的代理模式，JDBC使用的桥接模式，这些虽然不是自己实现的，但都在代码里面有用到过。

## redis在使用过程中遇到什么问题

由于平时比较注意redis的使用，key的值不会太长，数据量也没存多大的。之前的系统也不是类似于新闻，基本不存在热点数据这些，因此在生产中使用的redis也没出啥问题，有次倒是宕过机，但也是因为机器故障宕调的，还有就是缓存穿透问题，也没什么大碍。

## 各框架使用时遇到的问题

**Dubbo在使用过程中都遇到了些什么问题？ 如何解决的？**

比较常遇到就是因为服务端操作比较繁琐，造成超时的问题，之前做这个系统时因为有个对账模块因为数据量大偶尔会这样，这个要么优化代码逻辑，要么配置长一点的超时时间。

另一个就是因为传递的对象没有实现序列化接口而报错，印象比较深刻的就是list的sublist方法返回的list是未实现序列化的，这个要么换其它数据格式，要么转换成实现了序列化的集合。

其它的比如配置未生效，启动抱服务不可用，服务注册不上等等，一般都是环境或配置的问题，根据异常提示仔细检查一下就可以了。

kafka在使用过程中遇到什么问题

一般刚开始时问题比较多，大多也是配置出错原因导致的，比如，消费不到消息，可能是这个comsumer没有注册到zookeeper,或topic相关配置有问题，要么就是comsumer group配置有问题。线上也层出现过消息积压的情况，后面是增加消费者服务的方式解决了。

## Spring IOC容器是怎么实现对象的创建和依赖的：

容器启动时ResourceLoader从存储介质中加载Spring配置信息，并使用Resource表示这个配置文件的资源；BeanDefinitionReader会读取Resource所指向的配置文件资源，然后解析配置文件。配置文件中每一个<bean>解析成一个BeanDefinition对象，并保存到BeanDefinitionRegistry中；

容器扫描BeanDefinitionRegistry中的BeanDefinition；通过反射进行Bean实例化的工作；使用BeanWrapper完成Bean属性的设置工作；

单例Bean缓存池：Spring 在DefaultSingletonBeanRegistry类中提供了一个用于缓存单实例 Bean的缓存器，它是一个用HashMap实现的缓存器，单实例的Bean以beanName为键保存在这个HashMap中。

**IOC最大的优点是：**最小的代价和最小的侵入性使松散耦合得以实现。

原文：Spring IOC知识点一网打尽！ https://mp.weixin.qq.com/s/TYftOpe6J1Y4EDNDxKKqXg

## Spring AOP实现方式

Spring AOP编程底层通过的是动态代理技术，在调用的时候肯定用的是代理对象。那么Spring是怎么做的呢？我只需要写一个BPP，在postProcessBeforeInitialization或者postProcessAfterInitialization方法中，对对象进行判断，看他需不需要织入切面逻辑，如果需要，那我就根据这个对象，生成一个代理对象，然后返回这个代理对象，那么最终注入容器的，自然就是代理对象了。

Spring提供了BeanPostProcessor，就是让我们可以对有需要的对象进行“加工处理”啊！

BBP的全称叫做：BeanPostProcessor，一般我们俗称对象后处理器。简单来说，通过BeanPostProcessor可以对我们的对象进行“加工处理”。

Spring支持两种类型的事务管理：

编程式事务管理：这意味你通过编程的方式管理事务，给你带来极大的灵活性，但是难维护。

声明式事务管理：这意味着你可以将业务代码和事务管理分离，你只需用注解和XML配置来管理事务。

## Spring事务的传播属性

1) **REQUIRED** ，这个是默认的属性,如果存在一个事务，则支持当前事务。如果没有事务则开启一个新的事务。

2) **MANDATORY** 支持当前事务，如果当前没有事务，就抛出异常。

3) **NEVER** 以非事务方式执行，如果当前存在事务，则抛出异常。

4) **NOT\_SUPPORTED** 以非事务方式执行操作，如果当前存在事务，就把当前事务挂起。

5) **REQUIRES\_NEW** 新建事务，如果当前存在事务，把当前事务挂起。

6) **SUPPORTS** 支持当前事务，如果当前没有事务，就以非事务方式执行。

7) **NESTED** 支持当前事务，新增Savepoint点，与当前事务同步提交或回滚。嵌套事务一个非常重要的概念就是内层事务依赖于外层事务。外层事务失败时，会回滚内层事务所做的动作。而内层事务操作失败并不会引起外层事务的回滚。

PROPAGATION\_NESTED 与PROPAGATION\_REQUIRES\_NEW的区别

它们非常 类似,都像一个嵌套事务，如果不存在一个活动的事务，都会开启一个新的事务。使用PROPAGATION\_REQUIRES\_NEW时，内层事务与外层事务就像两个独立的事务一样，一旦内层事务进行了提交后，外层事务不能对其进行回滚。两个事务互不影响。两个事务不是一个真正的嵌套事务。同时它需要JTA 事务管理器的支持。

使用PROPAGATION\_NESTED时，外层事务的回滚可以引起内层事务的回滚。而内层事务的异常并不会导致外层事务的回滚，它是一个真正的嵌套事务。

## Spring事务失效情况

Spring事务失效的5种大的原因:

1. 如使用mysql且引擎是MyISAM，则事务会不起作用，原因是MyISAM不支持事务，可以改成InnoDB

2. 如果使用了spring+springmvc，要确保springmvc的注解不会扫描到service类，因为按照spring配置文件的加载顺序来讲，先加载springmvc配置文件，再加载spring配置文件，我们的事物一般都在srping配置文件中进行配置，如果此时在加载srpingMVC配置文件的时候，把servlce也给注册了，但是此时事物还没加载，也就导致后面的事物无法成功注入到service中。所以把对service的扫描放在spring配置文件中或是其他配置文件中。

3.查看事物配置中的excution表达式路径是否正确，查看rollback-for指定的异常类是否有。

4. @Transactional 注解开启配置，必须放到listener里加载，如果放到DispatcherServlet的配置里，事务也是不起作用的。

5. @Transactional 注解只能应用到 public 可见度的方法上。 如果你在 protected、private 或者 package-visible 的方法上使用 @Transactional 注解，它也不会报错，事务也会失效。

6.同一个service中嵌套事务只有前面那个会生效。我们知道Spring中事务管理是使用AOP代理技术实现的，目标对象自身并没有事务管理功能的，而是通过代理对象动态增强功能对事务进行增强的。因此当我们在同一个service类中通过一个方法调用另一个方法时，是通过目标对象this对象调用的，目标对象自身并没有事务管理功能，因此事务不能生效。

6. Spring团队建议在具体的类（或类的方法）上使用 @Transactional 注解，而不要使用在类所要实现的任何接口上。在接口上使用 @Transactional 注解，只能当你设置了基于接口的代理时它才生效。因为注解是 不能继承的，这就意味着如果正在使用基于类的代理时，那么事务的设置将不能被基于类的代理所识别，而且对象也将不会被事务代理所包装。

**Spring中同一个service中方法相互调用事务不生效问题解决方案**

我们知道Spring中事务管理是使用AOP代理技术实现的，目标对象自身并没有事务管理功能的，而是通过代理对象动态增强功能对事务进行增强的。因此当我们在同一个service类中通过一个方法调用另一个方法时，是通过目标对象this对象调用的，目标对象自身并没有事务管理功能，因此事务不能生效。

解决方案：

1.使用xml配置方式暴漏代理对象，然后在service中通过代理对象AopContext.currentProxy()去调用方法。

2.在java配置类上添加注解@EnableAspectJAutoProxy(exposeProxy = true)方式暴漏代理对象，然后在service中通过代理对象AopContext.currentProxy()去调用方法。

3.(不推荐)在service中自动装配service自身，然后同过装配对象调用。

## springMVC和struts2的区别有哪些?

（1）springmvc的入口是一个servlet即前端控制器（DispatchServlet），而struts2入口是一个filter过虑器（StrutsPrepareAndExecuteFilter）。

（2）springmvc是基于方法开发(一个url对应一个方法)，请求参数传递到方法的形参，可以设计为单例或多例(建议单例)，struts2是基于类开发，传递参数是通过类的属性，只能设计为多例。

（3）Struts采用值栈存储请求和响应的数据，通过OGNL存取数据，springmvc通过参数解析器是将request请求内容解析，并给方法形参赋值，将数据和视图封装成ModelAndView对象，最后又将ModelAndView中的模型数据通过reques域传输到页面。Jsp视图解析器默认使用jstl。

## SpringMVC工作流程描述

请求被Spring 前端控制Servelt DispatcherServlet捕获后，DispatcherServlet对请求URL进行解析，得到请求资源标识符，然后根据该URI，调用HandlerMapping获得对应的Handler

（包括Handler对象以及Handler对象对应的拦截器），最后以 HandlerExecutionChain对象的形式返回； DispatcherServlet 根据获得的Handler，获取一个合适的HandlerAdapter。

提取Request中的模型数据，填充Handler入参，HandlerAdapter会调用自己的handle方法，handle方法运用java的反射机制执行controller的具体方法。Handler执行完成后，向DispatcherServlet 返回一个ModelAndView对象；根据返回的ModelAndView，选择一个适合的ViewResolver（必须是已经注册到Spring容器中的 ViewResolver)返回给DispatcherServlet ； ViewResolver 结合Model和View，来渲染视图。将渲染结果返回给客户端。

## SpringBoot启动流程

当执行SpringApplication类的静态run方法时

1、创建SpringApplication实例：

将传入的启动类"com.example.ms.DemoApplication"放入Set集合中

判断是否为Web环境：存在（javax.servlet.Servlet && org.springframework.web.context.ConfigurableWebApplicationContext ）类则是

创建并初始化ApplicationInitializer列表 （spring.factories）

创建并初始化ApplicationListener列表  （spring.factories）

初始化主类mainApplicatioClass    (DemoApplication)

2、启动run方法

创建计时器StopWatch

配置awt系统属性

获取SpringApplicationRunListeners

启动SpringApplicationRunListeners

创建ApplicationArguments

创建并初始化ConfigurableEnvironment

打印Banner

创建ConfigurableApplicationContext

准备ConfigurableApplicationContext

刷新ConfigurableApplicationContext

容器刷新后动作

SpringApplicationRunListeners发布finish事件

计时器停止计时

## 优化你的Spring Boot

在SpringBoot的Web项目中，默认采用的是内置Tomcat，那么该如何优化容器的线程数，超时时间及jvm优化。

首先线程数是一个重点，初始线程数和最大线程数，初始线程数保障启动的时候，如果有大量用户访问，能够很稳定的接受请求。而最大线程数量用来保证系统的稳定性，而超时时间用来保障连接数不容易被压垮，如果大批量的请求过来，延迟比较高，不容易把线程打满。

在spring boot配置文件中application.yml，添加以下配置

server:

tomcat:

min-spare-threads: 20

max-threads: 100

connection-timeout: 5000

jvm优化是一个需要场景化的，没有什么太多特定参数，一般来说在server端运行都会指定如下参数

初始内存和最大内存基本会设置成一样的，具体大小根据场景设置，-server是一个必须要用的参数，至于收集器这些使用默认的就可以了，除非有特定需求。

1.使用-server模式

设置JVM使用server模式。64位JDK默认启动该模式

java -server -jar springboot-1.0.jar

2.指定堆参数

这个根据服务器的内存大小，来设置堆参数。

-Xms :设置Java堆栈的初始化大小

-Xmx :设置最大的java堆大小

java -server -Xms512m -Xmx768m -jar springboot-1.0.jar

设置初始化堆内存为512MB，最大为768MB。

3.远程Debug

在服务器上将启动参数修改为：

java -Djavax.net.debug=

ssl -Xdebug -Xnoagent -Djava.compiler=

NONE -Xrunjdwp:transport=

dt\_socket,server=y,suspend=

n,address=8888 -jar springboot-1.0.jar

这个时候服务端远程Debug模式开启，端口号为8888。

在IDEA中，点击Edit Configuration按钮，配置远程debug。

## SpringCloud之Fegin

@EnableFeignClients 注解开启 Feign 的功能

将@FeginClient注解，放在对应interface类上

@FeginClient参数介绍

name：指定FeignClient的名称，如果项目使用了Ribbon，name属性会作为微服务的名称，用于服务发现

url: url一般用于调试，可以手动指定@FeignClient调用的地址

decode404:当发生http 404错误时，如果该字段位true，会调用decoder进行解码，否则抛出FeignException

configuration: Feign配置类，可以自定义Feign的Encoder、Decoder、LogLevel、Contract

fallback: 定义容错的处理类，当调用远程接口失败或超时时，会调用对应接口的容错逻辑，fallback指定的类必须实现@FeignClient标记的接口

fallbackFactory: 工厂类，用于生成fallback类示例，通过这个属性我们可以实现每个接口通用的容错逻辑，减少重复的代码

path: 定义当前FeignClient的统一前缀

Spring Cloud 之 Feign 作为 HTTP 客户端调用远程服务，常见的异常主要有以下两类：

1. feign.FeignException: status 404 reading说明找不到被调用的方法，也就是你定义的 Feign 客户端接口与被调用接口不一致。一般都是配置问题。

2. Read timed out executing 调用服务超时。使得远程调用的时间超过 Feign 的默认超时时间，便会抛出该异常。通过设置 Feign 的超时时间可解决问题。Feign 的调用分两层，Ribbon 的调用和 Hystrix 的调用，高版本的 Hystrix 默认是关闭的，所以设置 Ribbon 即可。

#请求处理的超时时间

#ribbon.ReadTimeout: 120000

portal-test-service.ribbon.ReadTimeout: 120000

#请求连接的超时时间

#ribbon.ConnectTimeout: 30000

portal-test-service.ribbon.ConnectTimeout: 30000

3. status 404 reading 之 Request method 'POST' not supported。使用 Feign 远程调用 Get 请求不支持通过 @RequestBody 注解传递参数导致。

添加 feign-httpclient 依赖即可

Fegin的follback必须实现fegin接口

@FeignClient(name = "user",url = "${user.url}",fallback = UserFeignFallback.class

/\*fallbackFactory = UserFeignFactory.class\*/)

public interface UserFeign {

@GetMapping("/{id}")

User getUserByID(@PathVariable("id") String id);

}

@Component

public class UserFeignFallback implements UserFeign {

@Override

public User getUserByID(String id) {

User user = new User();

user.setId("100");

user.setName("fallback 回调用户");

return user;

}

}

Spring Cloud Feign HTTP请求异常Fallback容错机制，它是基于Hystrix实现的，所以要通过配置参数feign.hystrix.enabled=true开启该功能，及其两种实现方式。

Fallback工厂方式引出了ErrorDecoder错误解码自定义处理，有三种方式，可根据实际请求选择，举一反三其他自定义配置也可以通过这种方式实现如：Decoder、Encoder、Logger

## SpringCloud之Ribbon

当我们使用Spring Cloud Ribbon实现客户端负载均衡的时候，通常都会利用@LoadBalanced来让RestTemplate具备客户端负载功能，从而实现面向服务名的接口访问。

以下配置，配置重试机制：

hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds：断路器的超时时间需要大于ribbon的超时时间，不然不会触发重试。

hello-service.ribbon.ConnectTimeout：请求连接的超时时间

hello-service.ribbon.ReadTimeout：请求处理的超时时间

hello-service.ribbon.OkToRetryOnAllOperations：对所有操作请求都进行重试

hello-service.ribbon.MaxAutoRetriesNextServer：切换实例的重试次数

hello-service.ribbon.MaxAutoRetries：对当前实例的重试次数

## SpringCloud之hyxtrix

JVM工具远程连接

jconsole与Jvisualvm远程连接

通常我们的web服务都输部署在服务器上的，在window使用jconsole是很方便的，相对于Linux就有一些麻烦了，需要进行一些设置。

1.查看hostname,首先使用

hostname -i

查看，服务器的hostname为127.0.0.1，这个是不对的，需要进行修改

2.修改hostname

修改/etc/hosts文件，将其第一行的“127.0.0.1 localhost.localdomain localhost”，修改为：“192.168.44.128 localhost.localdomain localhost”.“192.168.44.128”为实际的服务器的IP地

3.重启Linux，在服务器上输入hostname -i，查看实际设置的IP地址是否为你设置的

4.启动服务，参数为：

java -jar -Djava.rmi.server.hostname=192.168.44.128 -

Dcom.sun.management.jmxremote -Dcom.sun.management.jmxremote.port=911 -

Dcom.sun.management.jmxremote.ssl=false -

Dcom.sun.management.jmxremote.authenticate=false jantent-1.0-SNAPSHOT.jar

ip为192.168.44.128，端口为911 。

5.打开Jconsole，进行远程连接,输入IP和端口即可

同理，JvisualVm的远程连接是同样的，启动参数也是一样。

然后在本机JvisualVm输入IP：PORT，即可进行远程连接：如下图所示：

## 说一下Innodb和MyISAM的区别

MyISAM类型不支持事务处理等高级处理，而InnoDB类型支持。MyISAM类型的表强调的是性能，其执行数度比InnoDB类型更快，但是不提供事务支持，而InnoDB提供事务支持以及外部键等高级数据库功能。

InnoDB不支持FULLTEXT类型的索引

InnoDB 中不保存表的具体行数，也就是说，执行select count(\*) from table时，InnoDB要扫描一遍整个表来计算有多少行，但是MyISAM只要简单的读出保存好的行数即可。注意的是，当count(\*)语句包含 where条件时，两种表的操作是一样的。

对于AUTO\_INCREMENT类型的字段，InnoDB中必须包含只有该字段的索引，但是在MyISAM表中，可以和其他字段一起建立联合索引。

DELETE FROM table时，InnoDB不会重新建立表，而是一行一行的删除。

LOAD TABLE FROM MASTER操作对InnoDB是不起作用的，解决方法是首先把InnoDB表改成MyISAM表，导入数据后再改成InnoDB表，但是对于使用的额外的InnoDB特性(例如外键)的表不适用。

## MySQL中事物ACID的原理

**Mysql怎么保证一致性的**

ACID嘛，原子性(Atomicity/,ætə'mɪsɪtɪ/)、一致性(Consistency/kən'sɪst(ə)nsɪ/)、隔离性(Isolation/aɪsə'leɪʃ(ə)n/)、持久性(Durability/djuərə'biləti/)

ACID四大特性之中，C(一致性)是目的，A(原子性)、I(隔离性)、D(持久性)是手段，是为了保证一致性，数据库提供的手段。数据库必须要实现AID三大特性，才有可能实现一致性。

**Mysql怎么保证原子性的**

是利用Innodb的undo log，undo log名为回滚日志，是实现原子性的关键，当事务回滚时能够撤销所有已经成功执行的sql语句，他需要记录你要回滚的相应日志信息。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，便可以利用undo log中的信息将数据回滚到修改之前的样子。

**Mysql怎么保证持久性的**

是利用Innodb的redo log。正如之前说的，Mysql是先把磁盘上的数据加载到内存中，在内存中对数据进行修改，再刷回磁盘上。如果此时突然宕机，内存中的数据就会丢失。

怎么解决这个问题？简单啊，事务提交前直接把数据写入磁盘就行啊。但是这样太浪费资源了。于是，决定采用redo log解决上面的问题。当做数据修改的时候，不仅在内存中操作，还会在redo log中记录这次操作。当事务提交的时候，会将redo log日志进行刷盘(redo log一部分在内存中，一部分在磁盘上)。当数据库宕机重启的时候，会将redo log中的内容恢复到数据库中，再根据undo log和binlog内容决定回滚数据还是提交数据。

采用redo log的好处是进行刷盘比对数据页刷盘效率高，具体表现如下：redo log体积小，毕竟只记录了哪一页修改了啥，因此体积小，刷盘快。redo log是一直往末尾进行追加，属于顺序IO。效率显然比随机IO来的快。

**Mysql怎么保证隔离性的**

利用的是锁和MVCC机制。MVCC,即多版本并发控制(Multi Version Concurrency Control),一个行记录数据有多个版本对快照数据，这些快照数据在undo log中。如果一个事务读取的行正在做DELELE或者UPDATE操作，读取操作不会等行上的锁释放，而是读取该行的快照版本。

**事务隔离级别：**

**SQL 标准定义了四个隔离级别**：

•**READ-UNCOMMITTED(读取未提交)**： 最低的隔离级别，允许读取尚未提交的数据变更，可能会导致脏读、幻读或不可重复读

•**READ-COMMITTED(读取已提交):** 允许读取并发事务已经提交的数据，可以阻止脏读，但是幻读或不可重复读仍有可能发生

•**REPEATABLE-READ（可重读）**: 对同一字段的多次读取结果都是一致的，除非数据是被本身事务自己所修改，可以阻止脏读和不可重复读，但幻读仍有可能发生。

•**SERIALIZABLE(可串行化)**: 最高的隔离级别，完全服从ACID的隔离级别。所有的事务依次逐个执行，这样事务之间就完全不可能产生干扰，也就是说，该级别可以防止脏读、不可重复读以及幻读。

## SQL解析，SQL执行过程

客户端把语句发给服务器端执行，服务器会先在数据库的高速缓存中去查找,是否存在相同语句的执行计划。如果在数据高速缓存中,则服务器进程就会直接执行这个 SQL 语句,省去后续的工作。如果没有就会走下一步，进行语句合法性检查(data dict cache)，语言含义检查(data dict cache)，当语法、语义都正确后,系统就会对我们需要查询的对象加锁。并且检查所连接的用户是否有这个数据访问的权限。当语句与语法都没有问题,权限也匹配的话,服务器进程还是不会直接对数据库文件进行查询。服务器进程会根据一定的规则,对这条语句进行优化。然后再是语义执行，提取数据并返回。

简要流程：查缓存->语句合法性检查->给查询对象加锁->权限检查->sql语句优化->提取数据

## SQL优化考虑的步骤

1.数据库设计和表创建时就要考虑性能，简言之就是使用合适的数据类型,选择合适的索引，表字段避免null值出现，使用枚举或整数代替字符串类型；单表不要有太多字段；

索引：要根据查询有针对性的创建，考虑在WHERE和ORDERBY命令上涉及的列建立索引；应尽量避免在WHERE子句中对字段进行NULL值判断；字符字段只建前缀索引；字符字段最好不要做主键；不用外键，由程序保证约束；简单查询建单列索引，复杂查询建组合索引。使用组合索引时主意顺序和查询条件保持一致，同时删除不必要的单列索引；

2.SQL的编写需要注意优化

SQL优化方面主要应该做到这两点：SQL语句尽可能简单，大语句拆小语句，减少锁时间。查询尽量走索引。

尽量避免在WHERE子句中使用!=或<>操作符，避免模糊查询，避免select \*，OR改写成IN（OR的效率是n级别，IN的效率是log(n)级别，in的个数建议控制在200以内），对于连续数值，使用BETWEEN不用IN。尽量不用函数和触发器，在应用程序实现。使用连接查询（join）来代替子查询，但有些情况下使用可能使用子查询更快些。

还有一些就是条件顺序的优化：from后面的表关联，是自右向左解析的 ，这时就可以将记录最少的表放在最右边。而where条件的解析顺序是自下而上的。那些可以过滤掉最大数量记录的条件必须写在WHERE子句的末尾,还有就是表之间的连接条件必须写在其它Where条件之前。

3.分区

如果做到上面点都无优化效果，可以考虑引入分区，MySQL在5.1版引入的分区是一种简单的水平拆分，用户需要在建表的时候加上分区参数，MySQL实现分区的方式也意味着索引也是按照分区的子表定义，没有全局索引。

用户的SQL语句是需要针对分区表做优化，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，可以通过EXPLAINPARTITIONS来查看某条SQL语句会落在那些分区上，从而进行SQL优化

分区的好处：

可以让单表存储更多的数据

分区表的数据更容易维护，可以通过清楚整个分区批量删除大量数据，也可以增加新的分区来支持新插入的数据。另外，还可以对一个独立分区进行优化、检查、修复等操作

部分查询能够从查询条件确定只落在少数分区上，速度会很快

分区表的数据还可以分布在不同的物理设备上，从而搞笑利用多个硬件设备

可以使用分区表赖避免某些特殊瓶颈，例如InnoDB单个索引的互斥访问、ext3文件系统的inode锁竞争

可以备份和恢复单个分区

分区的限制和缺点

一个表最多只能有1024个分区

如果分区字段中有主键或者唯一索引的列，那么所有主键列和唯一索引列都必须包含进来

分区表无法使用外键约束

NULL值会使分区过滤无效

所有分区必须使用相同的存储引擎

4.分表

分表分为垂直拆分和水平拆分，通常以某个字段做拆分项。

5.分库

把一个数据库分成多个，建议做个读写分离就行了，真正的做分库也会带来大量的开发成本

## sharding-jdbc使用流程

1.配置DataSource(配置多个数据源)

2.使用TableRuleConfiguration配置要分库分表的表，并配置分库分表策略，根据订单取模的方式定位到相应的数据库。

3.使用ShardingDataSourceFactory创建整合数据源

sharding-jdbc使用注意事项

不能执行批量插入操作：不支持对不同分片规则的字段值进行批量插入操作，类似sql：insert into t\_order values(x,x,x,x),(x,x,x,x),(x,x,x,x)

谨慎修改分片规则字段，否则会因为路由规则会造成数据存在，却查不到数据的情况

用sharding来执行一条创建sql的语句，会自动路由到各个库去执行。

不适合 OLAP 的场景。

不适合事务强一致的要求。

数据处理大致可以分成两大类：联机事务处理OLTP（on-line transaction processing）、联机分析处理OLAP（On-Line Analytical Processing）。OLTP是传统的关系型数据库的主要应用，主要是基本的、日常的事务处理，例如银行交易。OLAP是数据仓库系统的主要应用，支持复杂的分析操作，侧重决策支持，并且提供直观易懂的查询结果。

OLTP 系统强调数据库内存效率，强调内存各种指标的命令率，强调绑定变量，强调并发操作；

OLAP 系统则强调数据分析，强调SQL执行市场，强调磁盘I/O，强调分区等。

## 索引连环炮

索引具体采用的哪种数据结构呢？

常见的MySQL主要有两种结构：Hash索引和B+ Tree索引，我们使用的是InnoDB引擎，默认的是B+树

既然你提到InnoDB使用的B+ 树的索引模型，那么你知道为什么采用B+ 树吗？这和Hash索引比较起来有什么优缺点吗？

哈希索引适合等值查询，但是无法进行范围查询

哈希索引没办法利用索引完成排序

哈希索引不支持多列联合索引的最左匹配规则

如果有大量重复键值的情况下，哈希索引的效率会很低，因为存在哈希碰撞问题

刚刚我们聊到B+ Tree ，那你知道B+ Tree的叶子节点都可以存哪些东西吗？

InnoDB的B+ Tree可能存储的是整行数据，也有可能是主键的值

那这两者有什么区别吗？

在 InnoDB 里，索引B+ Tree的叶子节点存储了整行数据的是主键索引，也被称之为聚簇索引。而索引B+ Tree的叶子节点存储了主键的值的是非主键索引，也被称之为非聚簇索引

那么，聚簇索引和非聚簇索引，在查询数据的时候有区别吗？

聚簇索引查询会更快？

为什么呢？

因为主键索引树的叶子节点直接就是我们要查询的整行数据了。而非主键索引的叶子节点是主键的值，查到主键的值以后，还需要再通过主键的值再进行一次查询

刚刚你提到主键索引查询只会查一次，而非主键索引需要回表查询多次。（后来我才知道，原来这个过程叫做回表）是所有情况都是这样的吗？非主键索引一定会查询多次吗？

通过覆盖索引也可以只查询一次

覆盖索引（covering index）指一个查询语句的执行只用从索引中就能够取得，不必从数据表中读取。也可以称之为实现了索引覆盖。

当一条查询语句符合覆盖索引条件时，MySQL只需要通过索引就可以返回查询所需要的数据，这样避免了查到索引后再返回表操作，减少I/O提高效率。

如，表covering\_index\_sample中有一个普通索引 idx\_key1\_key2(key1,key2)。

当我们通过SQL语句：select key2 from covering\_index\_sample where key1 = 'keytest';的时候，就可以通过覆盖索引查询，无需回表。

不知道的话没关系，想问一下，你们在创建索引的时候都会考虑哪些因素呢？

我们一般对于查询概率比较高，经常作为where条件的字段设置索引

那你们在创建联合索引的时候，需要做联合索引多个字段之间顺序你们是如何选择的呢？

我们把识别度最高的字段放到最前面

为什么这么做呢？

最左前缀匹配的原则，即最左优先，在检索数据时从联合索引的最左边开始匹配。

where子句中使用最频繁的一列放在最左边，因为MySQL索引查询会遵循最左前缀匹配的原则，即最左优先，在检索数据时从联合索引的最左边开始匹配。所以当我们创建一个联合索引的时候，如(key1,key2,key3)，相当于创建了（key1）、(key1,key2)和(key1,key2,key3)三个索引，这就是最左匹配原则

你们线上用的MySQL是哪个版本啊呢？

我们MySQL是5.7

那你知道在MySQL 5.6中，对索引做了哪些优化吗？

有一个比较重要的 ：Index Condition Pushdown Optimization

Index Condition Pushdown（索引下推）

MySQL 5.6引入了索引下推优化，默认开启，使用SET optimizer\_switch = 'index\_condition\_pushdown=off';可以将其关闭。官方文档中给的例子和解释如下：

people表中（zipcode，lastname，firstname）构成一个索引

SELECT \* FROM people WHERE zipcode='95054' AND lastname LIKE '%etrunia%' AND address LIKE '%Main Street%';

如果没有使用索引下推技术，则MySQL会通过zipcode='95054'从存储引擎中查询对应的数据，返回到MySQL服务端，然后MySQL服务端基于lastname LIKE '%etrunia%'和address LIKE '%Main Street%'来判断数据是否符合条件。

如果使用了索引下推技术，则MYSQL首先会返回符合zipcode='95054'的索引，然后根据lastname LIKE '%etrunia%'和address LIKE '%Main Street%'来判断索引是否符合条件。如果符合条件，则根据该索引来定位对应的数据，如果不符合，则直接reject掉。有了索引下推优化，可以在有like条件查询的情况下，减少回表次数。

你们创建的那么多索引，到底有没有生效，或者说你们的SQL语句有没有使用索引查询你们有统计过吗？

这个还没有统计过，除非遇到慢SQL的时候我们才会去排查

那排查的时候，有什么手段可以知道有没有走索引查询呢？

可以通过explain查看sql语句的执行计划，通过执行计划来分析索引使用情况

那什么情况下会发生明明创建了索引，但是执行的时候并没有通过索引呢？

一条SQL语句的查询，可以有不同的执行方案，至于最终选择哪种方案，需要通过优化器进行选择，选择执行成本最低的方案。

在一条单表查询语句真正执行之前，MySQL的查询优化器会找出执行该语句所有可能使用的方案，对比之后找出成本最低的方案。

这个成本最低的方案就是所谓的执行计划。优化过程大致如下：

1、根据搜索条件，找出所有可能使用的索引

2、计算全表扫描的代价

3、计算使用不同索引执行查询的代价

4、对比各种执行方案的代价，找出成本最低的那一个

## mybatis 的 dao 接口跟 xml 文件里面的sql 是如何建立关系的？

Mybatis在初始化SqlSessionFactoryBean的时候，找到mapperLocations路径去解析里面所有的XML文件。Mybatis会把每个SQL标签封装成SqlSource对象（根据SQL语句的不同，又分为动态SQL和静态SQL。其中，静态SQL包含一段String类型的sql语句；而动态SQL则是由一个个SqlNode组成）。然后会将SqlSource与全限定类名+方法名组成的ID封装在一个MappedStatement对象中，并将它缓存到Configuration#mappedStatements中，每一个SQL标签就对应一个MappedStatement对象。

当使用@Autowired注入这个Dao接口的时候，spring会通过JDK动态代理，返回了一个Dao接口的代理对象，这个代理对象的处理器是MapperProxy对象。调用Dao接口方法的时候，实际调用到代理对象的invoke方法，在这里，实际上调用的就是SqlSession里面的东西了。SqlSession通过statement全限定类型+方法名拿到MappedStatement 对象，然后

MappedStatement调用执行器Executor去具体SQL并返回。

## Redis有哪些数据类型？

字符串String、字典Hash、列表List、集合Set、有序集合SortedSet。

如果你是Redis中高级用户，还需要加上下面几种数据结构HyperLogLog、Geo、Pub/Sub。

如果你说还玩过Redis Module，像BloomFilter，RedisSearch，Redis-ML，面试官得眼睛就开始发亮了。

**String：**最常规的set/get操作，value可以是String也可以是数字。一般做一些复杂的计数功能的缓存。

**hash:**这里value存放的是结构化的对象，比较方便的就是操作其中的某个字段。

**List：**使用List的数据结构，可以做简单的消息队列的功能。另外还有一个就是，可以利用lrange命令，做基于redis的分页功能，性能极佳，用户体验好。

**Set:**所以可以做全局去重的功能。就是利用交集、并集、差集等操作，可以计算共同喜好，全部的喜好，自己独有的喜好等功能。

**sorted set：**sorted set多了一个权重参数score,集合中的元素能够按score进行排列。可以做排行榜应用，取TOP N操作，可以用来做延时任务，可以做范围查找。

**HyperLogLog** 提供不精确的去重计数方案，虽然不精确但是也不是非常不精确，标准误差是 0.81%，这样的精确度已经可以满足上面的 UV 统计需求了。

**Redis**可以执行发布/订阅模式(publish/subscribe), 该模式可以解耦消息的发送者和接收者,使程序具有更好的扩展性.

**Redis底层数据结构**

**简单动态字符串（simple dynamic string）SDS；链表；字典；跳跃表；整数集合；压缩列表；对象**

**字典**，又称为符号表（symbol table）、关联数组（associative array）或映射（map），是一种用于保存键值对的抽象数据结构。在字典中，一个键（key）可以和一个值（value）进行关联，字典中的每个键都是独一无二的。比如SET msg "hello world"创建这样的键值对（“msg”，“hello world”）在数据库中就是以字典的形式存储。Redis 字典所使用的哈希表由 dict.h/dictht 结构定义.Redis 中采用了连地址法（separate chaining）来解决键冲突。

**跳跃表**：跳跃表（skiplist）是一种有序数据结构，它通过在每个节点中维持多个指向其他节点的指针，从而达到快速访问节点的目的。

**整数集合**：整数集合是集合建的底层实现之一，当一个集合中只包含整数，且这个集合中的元素数量不多时，redis就会使用整数集合intset作为集合的底层实现。我们可以这样理解整数集合，他其实就是一个特殊的集合，里面存储的数据只能够是整数，并且数据量不能过大。

**压缩列表**：压缩列表是列表键和哈希键的底层实现之一。当一个列表键只把汗少量列表项，并且每个列表项要么就是小整数，要么就是长度比较短的字符串，那么Redis 就会使用压缩列表来做列表键的底层实现。

**SDS**

**优点：**

获取字符串长度的复杂度为O(1)

API是安全的，不会造成缓存溢出问题

修改字符串长度N次最多需要执行N次内存重新分配（由于SDS结构中有free字段的存在，因此缩小时，会把没有的char算到free中（惰性空间释放），而放大（空间预分配）时如果len长度<1M，则扩大后的len=原有len\*2，如果len长度>=1M，则扩大后的len=原有len+1M）

可以保存文本或者二进制数据（因为SDS获取字符串时是根据len长度算的，而不是根据'\0'结尾之前的字符串算）

可以使用一部分<String.h>库中函数（不需要重新实现）

**缺点：**

很多函数以值的形式返回新字符串，由于有时SDS要求创建一个占用更多空间的新字符串，所以大多数SDS的API调用像这样：s = sdscat(s,"Some more data");你可以看到s被用来作为sdscat的输入，但也被设为SDS API调用返回的值，因为我们不知道此调用是否会改变了我们传递的SDS字符串，还是会重新分配一个新的字符串。忘记将sdscat或者类似函数的返回值赋回到存有SDS字符串的变量的话，就会引起bug。

如果一个SDS字符串在你的程序中多个地方共享，当你修改字符串的时候，你必须修改所有的引用。但是，大多数时候，当你需要共享SDS字符串时，将字符串封装成一个结构体，并使用一个引用计数会更好，否则很容易导致内存泄露。

## Kafka的Leader选举机制？

Kakfa Broker集群受Zookeeper管理。所有的Kafka Broker节点一起去Zookeeper上注册一个临时节点，因为只有一个Kafka Broker会注册成功，其他的都会失败，所以这个成功在Zookeeper上注册临时节点的这个Kafka Broker会成为Kafka Broker Controller，其他的Kafka broker叫Kafka Broker follower。（这个过程叫Controller在ZooKeeper注册Watch）。这个Controller会监听其他的Kafka Broker的所有信息，如果这个kafka broker controller宕机了，在zookeeper上面的那个临时节点就会消失，此时所有的kafka broker又会一起去Zookeeper上注册一个临时节点，因为只有一个Kafka Broker会注册成功，其他的都会失败，所以这个成功在Zookeeper上注册临时节点的这个Kafka Broker会成为Kafka Broker Controller，其他的Kafka broker叫Kafka Broker follower。

## kafka零拷贝等技术是如何运用的？

kafka中的消费者在读取服务端的数据时，需要将服务端的磁盘文件通过网络发送到消费者进程，网络发送需要经过几种网络节点。如下图所示：

传统的读取文件数据并发送到网络的步骤如下：

（1）操作系统将数据从磁盘文件中读取到内核空间的页面缓存；

（2）应用程序将数据从内核空间读入用户空间缓冲区；

（3）应用程序将读到数据写回内核空间并放入socket缓冲区；

（4）操作系统将数据从socket缓冲区复制到网卡接口，此时数据才能通过网络发送。

通常情况下，Kafka的消息会有多个订阅者，生产者发布的消息会被不同的消费者多次消费，为了优化这个流程，Kafka使用了“零拷贝技术”，如下图所示：

“零拷贝技术”只用将磁盘文件的数据复制到页面缓存中一次，然后将数据从页面缓存直接发送到网络中（发送给不同的订阅者时，都可以使用同一个页面缓存），避免了重复复制操作

如果有10个消费者，传统方式下，数据复制次数为4\*10=40次，而使用“零拷贝技术”只需要1+10=11次，一次为从磁盘复制到页面缓存，10次表示10个消费者各自读取一次页面缓存。

**kafka元数据**有Broker,Topic,Partition,Producer,Consumer,Consumer Group这几个? 2.kafka元数据都存储在zookeepr服务器上, 所以需要使用zookeeper, 具体的消息不在zookeeper上?

具体消息是在本地磁盘上，zookeeper只是负责管理元数据

## Leader broker

在Kafka集群中会有一个或者多个broker，其中有一个broker会被选举为控制器（Kafka Controller），它负责管理整个集群中所有分区和副本的状态。当某个分区的leader副本出现故障时，由控制器负责为该分区选举新的leader副本。当检测到某个分区的ISR集合发生变化时，由控制器负责通知所有broker更新其元数据信息。当使用kafka-topics.sh脚本为某个topic增加分区数量时，同样还是由控制器负责分区的重新分配。

Kafka中的控制器选举的工作依赖于Zookeeper，成功竞选为控制器的broker会在Zookeeper中创建/controller这个临时（EPHEMERAL）节点

具备控制器身份的broker需要比其他普通的broker多一份职责，具体细节如下：

监听partition相关的变化。为Zookeeper中的/admin/reassign\_partitions节点注册监听器（某种XXXListener，后面的也是），用来处理分区重分配的动作。为Zookeeper中的/isr\_change\_notification节点注册监听器，用来处理ISR集合变更的动作。为Zookeeper中的/admin/preferred-replica-election节点添加监听器，用来处理优先副本的选举动作。

监听topic相关的变化。为Zookeeper中的/brokers/topics节点添加监听器，用来处理topic增减的变化。为Zookeeper中的/admin/delete\_topics节点添加监听器，用来处理删除topic的动作。

监听broker相关的变化。为Zookeeper中的/brokers/ids/节点添加监听器，用来处理broker增减的变化。

从Zookeeper中读取获取当前所有与topic、partition以及broker有关的信息并进行相应的管理。对于所有topic所对应的Zookeeper中的/brokers/topics/[topic]节点添加监听器，用来监听topic中的分区分配变化。

启动并管理分区状态机和副本状态机。

更新集群的元数据信息。

如果参数auto.leader.rebalance.enable设置为true，则还会开启一个名为“auto-leader-rebalance-task”的定时任务来负责维护分区的优先副本的均衡。

## offset

consumer的offset是怎么存储的呢？consumer有两种消息方式，一种是存放在broker的日志目录中，另一种方式是存放在zookeeper中。两种存放方式和你使用kafka-console-consumer命令使用的选项有关。如果使用的是bootstrap-server，那么就存放在broker；如果使用的是–zookeeper那么就存放在zookeeper。

broker存储offset是kafka从0.9版本开始，提供的新的消费方式。原因是zookeeper来存放，还是有许多弊端，不方便灵活控制，效率不高。

下面通过bootstrap-server的方式消费数据：

使用consumer命令消费消息，这里我消费的是分区0中的消息，offset从5开始：kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic demo1 --partition 0 --offset 5

打印所有的消费组名称：kafka-consumer-groups.sh --bootstrap-server localhost:9092 --list

查看消费明细：kafka-consumer-groups.sh --bootstrap-server localhost:9092 --desc --group console-consumer-33936

下面通过zookeeper的方式消费数据：kafka-console-consumer.sh --zookeeper localhost:2181/kafka1 --topic demo1 --from-beginning

打印所有的消费组名称：kafka-consumer-groups.sh --zookeeper localhost:2181/kafka1 --list

查看消费明细，可以看到，我已经消息到第11条数据了：kafka-consumer-groups.sh --zookeeper localhost:2181/kafka1 --desc --group console-consumer-69524

也可以通过zookeeper client来查看consumer的offset：get /kafka1/consumers/console-consumer-69524/offsets/demo1/0

Kafka对于offset的处理有两种提交方式：(1) 自动提交(默认的提交方式) (2) 手动提交(可以灵活地控制offset)

(1) 自动提交偏移量:

Kafka中偏移量的自动提交是由参数enable\_auto\_commit和auto\_commit\_interval\_ms控制的，当enable\_auto\_commit=True时，Kafka在消费的过程中会以频率为auto\_commit\_interval\_ms向Kafka自带的topic(\_\_consumer\_offsets)进行偏移量提交，具体提交到哪个Partation是以算法：partation=hash(group\_id)%50来计算的。

(2) 手动提交偏移量:

鉴于Kafka自动提交offset的不灵活性和不精确性(只能是按指定频率的提交)，Kafka提供了手动提交offset策略。手动提交能对偏移量更加灵活精准地控制，以保证消息不被重复消费以及消息不被丢失。

对于手动提交offset主要有3种方式：1.同步提交 2.异步提交 3.异步+同步 组合的方式提交

1.同步手动提交偏移量

同步模式下提交失败的时候一直尝试提交，直到遇到无法重试的情况下才会结束，同时同步方式下消费者线程在拉取消息会被阻塞，在broker对提交的请求做出响应之前，会一直阻塞直到偏移量提交操作成功或者在提交过程中发生异常，限制了消息的吞吐量。

2.异步手动提交偏移量+回调函数

异步手动提交offset时，消费者线程不会阻塞，提交失败的时候也不会进行重试，并且可以配合回调函数在broker做出响应的时候记录错误信息。对于异步提交，由于不会进行失败重试，当消费者异常关闭或者触发了再均衡前，如果偏移量还未提交就会造成偏移量丢失。

3.异步+同步 组合的方式提交偏移量

针对异步提交偏移量丢失的问题，通过对消费者进行异步批次提交并且在关闭时同步提交的方式，这样即使上一次的异步提交失败，通过同步提交还能够进行补救，同步会一直重试，直到提交成功。

Kafka具有存储功能，默认保存数据时间为7天或者大小1G，也就是说kafka broker上的数据超7天或者1G，就会被清理掉。这些数据存放在broker服务器上，以log文件的形式存在。

kafka容错机制

1.Replication机制

一般情况下partition的数量大于等于broker的数量，并且所有partition的leader均匀分布在broker上。Kafka从0.8开始提供partition级别的replication，可以在config/server.properties中配置，默认是1。每个partition都有一个唯一的leader，所有的读写操作都在leader上完成，replication批量从leader上pull数据。leader会track “in sync”的node list（ISR）。如果一个follower宕机，或者落后太多，leader将把它从”in sync” list中移除。

2.broker 容灾机制

1. controller 在 zookeeper 的 /brokers/ids/[brokerId] 节点注册 Watcher，当 broker 宕机时 zookeeper 会 fire watch

2. controller 从 /brokers/ids 节点读取可用broker

3. controller决定set\_p，该集合包含宕机 broker 上的所有 partition

4. 对 set\_p 中的每一个 partition

4.1 从/brokers/topics/[topic]/partitions/[partition]/state 节点读取 ISR

4.2 决定新 leader（如4.3节所描述）

4.3 将新 leader、ISR、controller\_epoch 和 leader\_epoch 等信息写入 state 节点

5. 通过 RPC 向相关 broker 发送 leaderAndISRRequest 命令

3.Leader Election机制

当leader宕机了，怎样在follower中选举出新的leader。

https://blog.csdn.net/wingofeagle/article/details/60966001

4.controller failover

 当 controller 宕机时会触发 controller failover。每个 broker 都会在 zookeeper 的 "/controller" 节点注册 watcher，当 controller 宕机时 zookeeper 中的临时节点消失，所有存活的 broker 收到 fire 的通知，每个 broker 都尝试创建新的 controller path，只有一个竞选成功并当选为 controller。

**Kafka存活包含两个条件：**

一是它必须维护与Zookeeper的session(这个通过Zookeeper的heartbeat机制来实现)

二是follower必须能够及时将leader的writing复制过来，不能“落后太多”

## MQ之间比较

RabbitMQ

结合erlang语言本身的并发优势，支持很多的协议：AMQP，XMPP, SMTP, STOMP，也正是如此，使的它变的非常重量级，更适合于企业级的开发。

性能较好，但是不利于做二次开发和维护。

ActiveMQ

历史悠久的开源项目，是Apache下的一个子项目。已经在很多产品中得到应用，实现了JMS1.1规范，可以和spring-jms轻松融合，实现了多种协议，不够轻巧（源代码比RocketMQ多），支持持久化到数据库，对队列数较多的情况支持不好。

​Kafka

Apache下的一个子项目，使用scala实现的一个高性能分布式Publish/Subscribe消息队列系统，具有以下特性：

快速持久化：通过磁盘顺序读写与零拷贝机制，可以在O(1)的系统开销下进行消息持久化；

高吞吐：在一台普通的服务器上既可以达到10W/s的吞吐速率；

高堆积：支持topic下消费者较长时间离线，消息堆积量大；

完全的分布式系统：Broker、Producer、Consumer都原生自动支持分布式，依赖zookeeper自动实现复杂均衡；

支持Hadoop数据并行加载，对于像Hadoop的一样的日志数据和离线分析系统，但又要求实时处理的限制，这是一个可行的解决方案。

Kafka通过Hadoop的并行加载机制统一了在线和离线的消息处理。Apache Kafka相对于ActiveMQ是一个非常轻量级的消息系统，除了性能非常好之外，还是一个工作良好的分布式系统。

## NIO和传统的IO有什么区别呢？

传统的IO，需要为每个连接创建一个线程，当并发的连接数量非常巨大时，线程所占用的栈内存和CPU线程切换的开销将非常巨大。使用NIO，不再需要为每个线程创建单独的线程，可以用一个含有限数量线程的线程池，甚至一个线程来为任意数量的连接服务。

IO是面向流的，一次可以读取一个或多个字节。NIO是缓冲区的，数据是先被 读/写到buffer中的，你可以控制读取什么位置的数据。

IO是阻塞式IO，当一条线程执行read()或者write()方法时，这条线程会一直阻塞直到读取到了一些数据或者要写出去的数据已经全部写出，在这期间这条线程不能做任何其他的事情。NIO在非阻塞式模式下，允许一条线程从channel中读取数据，通过返回值来判断buffer中是否有数据，如果没有数据，NIO不会阻塞。

Java NIO的selectors允许一条线程去监控多个channels的输入，你可以向一个selector上注册多个channel，然后调用selector的select()方法判断是否有新的连接进来或者已经在selector上注册时channel是否有数据进入。

## netty的线程模型么？

Netty使用的是Reactor线程模型，Netty使用EventLoop来处理连接上的读写事件，而一个连接上的所有请求都保证在一个EventLoop中被处理，一个EventLoop中只有一个Thread，所以也就实现了一个连接上的所有事件只会在一个线程中被执行。一个EventLoopGroup包含多个EventLoop，可以把一个EventLoop当做是Reactor线程模型中的一个线程，而一个EventLoopGroup类似于一个ExecutorService，

## Dubbo完整的一次调用链路介绍；

dubbo服务初始化以后，会将service接口生成代理对象，将接口的方法进行包装。每次调用都会调用到InvocationHandler的invoke()方法，此方法中会进行远程服务调用一些列复杂过程，诸如网络通信，编码，解码，序列化等，然后将结果返回。在InvokerInvocationHandler.invoker()方法中，最后调用invoker.invoke(new RpcInvocation(method,args))然后调用到FailfastClusterInvoker.invoke()，此对象持有接口服务的RegistryDirectory，里面含有远程提供者的具体信息，是在服务消费者初始化时，通过订阅zk相应节点得到的。此方法的调用过程比较复杂，会调用负载均衡算法，根据一定的策略，选取一个提供者，生成DubboInvoker对象,再进行DubboInvoke.invoke()的调用。进入HeaderExchangeClient.request()方法，HeaderExchangeClient里面封装了NettyClient，由此将请求交予netty框架发送。

请求到达服务端以后，netty服务端会先对数据进行解码，解码之后进入NettyHandler的messageReceived()方法，然后进入到DubboProtocol内部类ExchangeHandlerAdapter的reply()方法，此invoker为包装类型，包含filter执行链、提供端接口实现类的包装类，执行invoke()方法，先调用filter执行链，然后调用AbstractProxyInvoker实现类的doInvoke()方法，进行调用的包装类的invokeMethod()方法，包装类持有具体实现类，至此调用完毕，将结果封装成RpcResult返回,回到HeaderExchangeHandler.receive()方法，调用channel.send()写到客户端。

## dubbo的服务暴露过程

Dubbo会在Spring实例化完bean之后，在刷新容器最后一步发布ContextRefreshEvent事件的时候，通知实现了ApplicationListener的ServiceBean类进行回调onApplicationEvent 事件方法，dubbo会在这个方法中调用ServiceBean父类ServiceConfig的export方法，而该方法真正实现了服务的（异步或者非异步）发布。

由服务配置类 ServiceConfig 进行初始化工作及服务暴露入口，首先进去执行该类的export()方法。

export方法先判断是否需要延迟暴露（这里我们使用的是不延迟暴露），然后执行doExport方法。

doExport方法先执行一系列的检查方法，然后调用doExportUrls方法。检查方法会检测dubbo的配置是否在Spring配置文件中声明，没有的话读取properties文件初始化。

doExportUrls方法先调用loadRegistries获取所有的注册中心url，然后遍历调用doExportUrlsFor1Protocol方法。对于在标签中指定了registry属性的Bean，会在加载BeanDefinition的时候就加载了注册中心。

原文：<https://blog.csdn.net/yanpenglei/article/details/80261762>

## Dubbo框架结构

服务层：

Dubbo最大的特点就是按照分层的方式来架构，使用这种方式可以实现各个层之间最大限度地松耦合。Dubbo的框架设计一共划分为10层，有服务接口层，配置层，服务代理层，集群层，监控层，服务注册层，远程调用层，信息交换层，网络传输层，数据序列化层。其中我觉得最主要的就是服务代理层，服务注册层与集群层，服务代理层会将根据你写的接口生成一个代理类，用于处理请求，而服务注册层则会将服务地址暴露给注册中心，请求过来后再由底层转发给各个代理实现类去处理，并响应。

服务接口层（Service）：该层是与实际业务逻辑相关的，根据服务提供方和服务消费方的业务设计对应的接口和实现。

配置层（Config）：对外配置接口，以ServiceConfig和ReferenceConfig为中心，可以直接new配置类，也可以通过spring解析配置生成配置类。

服务代理层（Proxy）：服务接口透明代理，生成服务的客户端Stub和服务器端Skeleton，以ServiceProxy为中心，扩展接口为ProxyFactory。

服务注册层（Registry）：封装服务地址的注册与发现，以服务URL为中心，扩展接口为RegistryFactory、Registry和RegistryService。可能没有服务注册中心，此时服务提供方直接暴露服务。

集群层（Cluster）：封装多个提供者的路由及负载均衡，并桥接注册中心，以Invoker为中心，扩展接口为Cluster、Directory、Router和LoadBalance。将多个服务提供方组合为一个服务提供方，实现对服务消费方来透明，只需要与一个服务提供方进行交互。

监控层（Monitor）：RPC调用次数和调用时间监控，以Statistics为中心，扩展接口为MonitorFactory、Monitor和MonitorService。

远程调用层（Protocol）：封将RPC调用，以Invocation和Result为中心，扩展接口为Protocol、Invoker和Exporter。Protocol是服务域，它是Invoker暴露和引用的主功能入口，它负责Invoker的生命周期管理。Invoker是实体域，它是Dubbo的核心模型，其它模型都向它靠扰，或转换成它，它代表一个可执行体，可向它发起invoke调用，它有可能是一个本地的实现，也可能是一个远程的实现，也可能一个集群实现。

信息交换层（Exchange）：封装请求响应模式，同步转异步，以Request和Response为中心，扩展接口为Exchanger、ExchangeChannel、ExchangeClient和ExchangeServer。

网络传输层（Transport）：抽象mina和netty为统一接口，以Message为中心，扩展接口为Channel、Transporter、Client、Server和Codec。

数据序列化层（Serialize）：可复用的一些工具，扩展接口为Serialization、 ObjectInput、ObjectOutput和ThreadPool。

## Zookeeper选举原理

Zookeeper的用途包括统一配置管理、统一命名服务、分布式锁、集群管理。

如何在zookeeper集群中选举出一个leader,zookeeper使用了三种算法,具体使用哪种算法,在配置文件中是可以配置的,对应的配置项是”electionAlg”,其中1对应的是LeaderElection算法,2对应的是AuthFastLeaderElection算法,3对应的是FastLeaderElection算法.默认使用FastLeaderElection算法.其他两种算法我没有研究过,就不多说了.

要理解这个算法,最好需要一些paxos算法的理论基础.

1) 数据恢复阶段

首先,每个在zookeeper服务器先读取当前保存在磁盘的数据,zookeeper中的每份数据,都有一个对应的id值,这个值是依次递增的,换言之,越新的数据,对应的ID值就越大.

2) 向其他节点发送投票值

在读取数据完毕之后,每个zookeeper服务器发送自己选举的leader（首次选自己）,这个协议中包含了以下几部分的数据:

a)所选举leader的id(就是配置文件中写好的每个服务器的id) ,在初始阶段,每台服务器的这个值都是自己服务器的id,也就是它们都选举自己为leader.

b) 服务器最大数据的id,这个值大的服务器,说明存放了更新的数据.

c)逻辑时钟的值,这个值从0开始递增,每次选举对应一个值,也就是说: 如果在同一次选举中,那么这个值应该是一致的 ; 逻辑时钟值越大,说明这一次选举leader的进程更新.

d) 本机在当前选举过程中的状态,有以下几种:LOOKING,FOLLOWING,OBSERVING,LEADING,顾名思义不必解释了吧.

服务器接受来自其他各个服务器的投票，并判断投票的有效性（包括检查是否是本轮投票、是否来自LOOKING状态的服务器）;

处理投票（服务器将自己的投票和收到的投票进行对比，先检查最大数据的id，大的服务器作为leader；如果ZXID相同，检查服务器id，服务器id大的作为leader；更新投票），将最终的投票重新发出去；

统计投票：每次投票后，服务器统计所有投票，判断是否有过半的服务器收到相同的投票信息；如果是，则选举出了新的leader，如果不是，重新开始投票；

改变服务器状态：follower将自己的状态改为FOLLOWING,leader将自己的状态改为LEADING。

以一个简单的例子来说明整个选举的过程.

假设有五台服务器组成的zookeeper集群,它们的id从1-5,同时它们都是最新启动的,也就是没有历史数据,在存放数据量这一点上,都是一样的.假设这些服务器依序启动,来看看会发生什么.

1) 服务器1启动,此时只有它一台服务器启动了,它发出去的报没有任何响应,所以它的选举状态一直是LOOKING状态

2) 服务器2启动,它与最开始启动的服务器1进行通信,互相交换自己的选举结果,由于两者都没有历史数据,所以id值较大的服务器2胜出,但是由于没有达到超过半数以上的服务器都同意选举它(这个例子中的半数以上是3),所以服务器1,2还是继续保持LOOKING状态.

3) 服务器3启动,根据前面的理论分析,服务器3成为服务器1,2,3中的老大,而与上面不同的是,此时有三台服务器选举了它,所以它成为了这次选举的leader.

4) 服务器4启动,根据前面的分析,理论上服务器4应该是服务器1,2,3,4中最大的,但是由于前面已经有半数以上的服务器选举了服务器3,所以它只能接收当小弟的命了.

5) 服务器5启动,同4一样,当小弟.

**对Paxos算法了解多少？**

Paxos是一种分布式一致性算法，

Paxos算法定义了三种角色：Acceptor、Proposer和Learner，其中Proposer是提议发起者，负责接收客户端请求，并将客户端的请求发送到Paxos集群中，以便决定这个值是否可以被批准；Acceptor是提议批准者，负责处理接收到的提议；Learner是学习者，只能学习到已经被批准的值，不能学习没有被批准的值。每一个进程都可以扮演其中任意一种角色。

Paxos完整算法核心过程包含prepare和accept两个阶段，可大致描述为：

1) prepare阶段：Proposer向Acceptor发起提议权申请请求（Prepare消息），Acceptor负责批准Proposer申请的提议权，如果同意就回复Promise消息

2) accept阶段：Proposer一旦获得提议权即可进一步提交变量取值（Accept消息），Acceptor负责批准Proposer提交的变量取值，如果接受就回复Accepted消息

## 内存泄露排查思路

1、使用top命令，查看占用内存较高的进程ID。

发现PID为3331的进程占用内存 90.6%。而且是一个Java进程，基本断定是程序问题。

2、使用jmap查看内存情况，并分析是否存在内存泄露。

jmap -heap 3331：查看java 堆（heap）使用情况

jmap -histo 3331：查看堆内存(histogram)中的对象数量及大小

jmap -histo:live 3331：JVM会先触发gc，然后再统计信息

jmap -dump:format=b,file=heapDump 3331：将内存使用的详细情况输出到文件

得到堆dump文件后，可以进行对象分析。如果有大量对象在持续被引用，并没有被释放掉，那就产生了内存泄露，就要结合代码，把不用的对象释放掉。

# 生产问题定位

## CPU高负载排查实践

首先利用 top-c 将系统资源使用情况实时显示出来 （ -c 参数可以完整显示命令）。接着输入 大写P 将应用按照 CPU 使用率排序，第一个就是使用率最高的程序。果不其然就是我们的一个 Java 应用。

利用 top-Hppid 然后输入 P 依然可以按照 CPU 使用率将线程排序。这时我们只需要记住线程的 ID 将其转换为 16 进制存储起来，通过 jstack pid>pid.log生成日志文件，利用刚才保存的 16 进制进程 ID 去这个线程快照中搜索即可知道消耗 CPU的线程在干啥了。如果你嫌麻烦，我也强烈推荐阿里开源的问题定位神器 arthas 来定位问题。比如上述操作便可精简为一个命令 thread-n3 即可将最忙碌的三个线程快照打印出来，非常高效。

最忙绿的线程是一个 GC 线程，也就意味着它在忙着做垃圾回收。

排查到这里，有经验的老司机一定会想到：多半是应用内存使用有问题导致的。于是我通过 jstat-gcutil pid20050 将内存使用、gc 回收状况打印出来（每隔 200ms 打印 50次）。

既然是初步定位是内存问题，所以还是得拿一份内存快照分析才能最终定位到问题。通过命令 jmap-dump:live,format=b,file=dump.hprof pid 可以导出一份快照文件。这时就得借助 MAT 这类的分析工具出马了。

使用MAT工具可以看出，在内存中存在一个非常大的字符串，而这个字符串正好是被这个定时任务的线程引用着。其实看上图中的引用关系及字符串的内容不难看出这是一个 insert 的 SQL 语句。

最终通过这个线程快照找到了具体的业务代码：他调用一个写入数据库的方法，而这个方法会拼接一个 insert 语句，其中的 values 是循环拼接生成

最终，对这循环的集合做了限定大小

## Java 应用性能优化

Java 应用性能优化是一个老生常谈的话题，典型的性能问题如页面响应慢、接口超时，服务器负载高、并发数低，数据库频繁死锁等。

将 Java 性能优化分为 4 个层级：应用层、数据库层、框架层、JVM 层，

每层优化难度逐级增加，涉及的知识和解决的问题也会不同。比如应用层需要理解代码逻辑，通过 Java 线程栈定位有问题代码行等；数据库层面需要分析 SQL、定位死锁等；框架层需要懂源代码，理解框架机制；JVM 层需要对 GC 的类型和工作机制有深入了解，对各种 JVM 参数作用了然于胸。

## 你是如何使用JDK来实现自己的缓存（支持高并发）？

使用HashMap以键值对的方式存储数据，值为entity对象（内部类），该对象包含value与过期时间。

创建定时器线程池，用于清除过期缓存，Put,get,remove,size方法要同步。