# Java基础

## IO流

1. 了解IO流吗，能说一下常用的IO流吗

答：常用的IO流有FileInputStream，ByteArrayInputStream，BufferedInputStream，InputStreamReader等

1. 能具体说下这些常用的IO流的用法吗？还这些IO流是否线程安全？

答：以FileWriter举例，通过将File类型的对象读入到内存中，然后以字符（InputStream则是字节）数组的形式从内存中读取出来。这些IO流都是线程不安全的，如果采取多线程进行文件复制，因为文件的读取和写入都是从同一内存区域拿取，我们的所有步骤都必须保证原子性，必须要使用同步块或者同步方法（不能使用volatile关键字，它只能保证可见性和同步性，但无法保证原子性）进行修饰，否则会出现“串行”

1. 说说字节流（Input、Output）和字符流（Reader、Writer）的优缺点？

答：字符流牛逼就是了。字节流是来自JDK1.0版本的产物，复杂逻辑相较于新版本的字符流要差很多，读写效率也低很多。字符流是JDK1.2的产物，对流进行了大量的优化，无论是读写速度，完整性亦或是模式来说，字符流都要优于字节流。字节流被保存下来的原因是字节流妹有编码问题，但是字符流有，所以再涉及到文件编码的时候，字符流会当场裂开。还有一个就是字节流在某些特定的条件下，读写速度要快于字符流。

1. IO流操作用了哪些设计模式。用这些设计模式有什么好处？（跳转到设计模式篇）

答：装饰器模式。用一个简单的流进行文件的读写，只能做到简单的文件复制，哪怕是对文件进行格式上的修改都会很困难，所以使用装饰器模式，每次创建特殊的流都可以把基础流传进去，然后添加特殊的属性，使得基础流变得多样性，但是前提条件，一定要实现基础流的公共接口，因为要保证都隶属于改接口的操作，保证一致性。

1. 如果让你读取一个体积庞大的CSV文件，你将如何处理。

答：首先，将文件引用载入对应的流处理对象（FileReader）中，读取该文件的引用。

其次，记上游标，自己从第？行开始到第？行结束，每次读取完清空缓冲区。

最后，在处理完成后关闭流，释放内存，让对象得以被回收，避免造成OOM

## 集合

1. 你知道的集合都有哪些。

答：Collection子接口List，Set，Queue，相关的实现。Map的实现

1. 能否详细的讲解一下常用的List或Set的实现。

答：

ArrayList：由数组维护，是个线程不安全的数组，能用于快速的查找数据，也能快速的增加或修改数据（不产生扩容的情况下），但是插入数据到某个位置却很慢，默认的初始数组大小为10，在使用add方法时会去校验扩容的情况，扩容grow()方法，每次扩容原大小\*1.5。

LinkedList：由链表维护，是个线程不安全的链表，能用于快速的插入，增加，不适合遍历快速查找节点，没有默认的初始值大小，也不必担心扩容问题，每次只会增加一个。LinkedList不仅实现了List接口同时还是先了Queue接口，能够实现先进先出（FIFO）的原则。

HashSet：由HashMap进行维护，创建HashMap使用的默认大小16，如果使用集合进行导入的话，计算出默认的大小，指定的阈值也与集合本身的size相同，避免HashMap自行扩容。HashSet只需要HashMap的Key，value取的是默认情况下的一个常量Object。HashSet为了保证键不重复，所以会进行Hash计算，然后通过equals方法比对两个对象是否相等，若相等，则插入失败。

TreeSet：同样是有TreeMap进行维护的，基本情况与HashSet相同，才用的是红黑树（自平衡排序二叉树）进行排序。

以上就是collection下的子类相关集合。

1. 有哪些集合是线程安全的？为什么线程安全。

答：Vector，stack，hashtable，enumeration。

上述的几个集合都是被synchronize修饰的，性能极差，不推荐使用，具体的优化方案可以

见多线程篇。

1. Java集合的快速失败机制（“fail-fast”）是什么？

答：是当集合在遍历的过程中，一旦有其他线程同样操作了这个集合导致集合发生变更，那么，迭代器就会立即停止遍历并抛出异常（ConcurrentModificationException）。而产生这个问题主要的原因就是迭代器有一个modCount的计数标记，每一次都会与遍历之前的计数标记expectModeCount进行比对，是否正确。一旦修改了比对就会失败，也就会抛出此异常。

如何解决呢：

Method-1.使用迭代器本省进行遍历操作。Iterator去遍历，那么每次都能比对成功，但是不推荐这么做

Method-2.所有涉及到的modCount运算的地方都要加上同步块。

Method-3.采用复制对象的形式创建一个副本，操作副本当操作完成后，将副本同步到原有集合中。

1. 简单介绍一下常用Map集合。

答：

HashMap：采用数组+链表+红黑树的方法实现key-value的存储。先将元素进行Hash计算，然后通过equals的方法比对是否是同一个key，若不是则使用单节点链表避免Hash碰撞，当链路中保存的数据超过阈值（HashMap的阈值默认为8）且数组长度大于64时，采用红黑树（自平衡排序二叉树）进行保存。HashMap初始大小为16，有一个默认的阈值，当大小达到阈值（默认情况下的0.75倍）时，put元素会产生resize扩容，扩容的大小为原有大小的2倍。

LinkedHashMap：属于HashMap的子类，具有HashMap相同的属性，但是采用的是双向链表结构，能够确定key-value的插入顺序。同时可以实现对链表的顺序访问（插入时的顺序）

TreeMap：与HashMap类似，同样的初始值，同样的扩容机制，但不同的是，TreeMap使用了红黑树（自平衡排序二叉树）进行维护，能够按照指定的顺序查找到对应的数据，是一个有序的Map集合。

以上就是Map下实现类相关的集合。

1. 简述一下HashMap的底层实现原理。

答：HashMap是基于Hash表来实现的集合。HashMap本质上是一个“散列列表”，HashMap的基本存储是先通过计算Hash值确定当前数组的下标。并将值的引用保存下来的。但是过程中难免会出现Hash碰撞，一旦产生了碰撞，就证明一个区域有两个不同的值，这两个值需要存储下去，那么就通过key-value链表来进行保存。在JDK8后，HashMap会对节点有个阈值，当超过这个阈值后，就是用红黑树进行存储。

当查找元素时，会先进行Hash计算，找到对应的下标，然后通过equals进行key值的比对，然后得到相应的value。

1. 简单介绍一下HashMap的扩容机制。

答：HashMap扩容是在当插入元素后，现有长度超过了HashMap本身预计的长度（无参构造情况下是16\*0.75），那么会将自己数组的容量扩展至2倍，然后对老的数组进行数据重排，是的分布更加均匀。如果是在是有有参数的构造情况下，HashMap会对数组进行初始化大小，并且扩容阈值也等于初始化的大小，能减小扩容产生的风险。

1. 什么是扰动？HashMap怎么减少产生Hash碰撞？

答：扰动就是指让整个Hash值参与运算而不是只是用低位的Hash值去运算，把这个运算的过程称之为扰动。

HashMap在JDK1.8版本中，采用了一次位运算，一次异或运算是的分布来实现分布。具体代码如下：

(h = key.hashCode()) ^ (h >>> 16);

而在JDK7版本中，采用了4次位运算，5次异或运算。

1. 简单介绍HashMap1.8与1.7做了那些优化。

答：hashmap相较于1.7主要从三个方面入手来说。

Hash散列计算的方案：因为采用1.7的逻辑运算，就是同一个只反复的取hash，但这种情况会导致hash值的高8位始终不会去参与计算，而且计算过程非常的冗长，而1.8采用了让高位参与运算，然后利用原数组长度+计算结果进行重新定位，减少了扩容所需要的时间，不在需要进行所有元素的重新计算，提高了扩容效率。

链表头插法改尾插法：在1.7中会导致链表的元素倒排，责在多线程场景下，同时触发扩容，那么会导致原来的头结点指向的下一个节点与原来的头结点形成环状，导致死循环，所以同步了扩容时的尾插法，解决了死循环问题

链表转红黑树：1.7中，链表太长会导致hashmap本身查询效率问题，那么为了优化这个查询问题，链表长度大于8以后，会选择转红黑树。红黑树的优势就在查询效率的一个提升，相较于其他平衡二叉树来说，他的平衡度较低，适合频繁的增删，也不会导致其频繁的旋转导致效率低下。

1. 1.8的hashmap中为什么使用了红黑树而不是平衡二叉树（AVL）或者常规二叉树。

答：若选择常规二叉树，二叉树对树的高度是没有保证的，所以在极端情况下是有可能退化成链表的，若选择平衡二叉树，可以解决链表退化问题，但是平衡二叉树中，最短路径叶子节点与最长路径叶子节点高度差不能超过1，这样就会导致在插入数据时频繁的翻转树，导致插入效率低下。而红黑树的最短路径与最长路径是不能超过两倍，是一个不严格的平衡二叉树，所以不会导致频繁翻转。对于hashmap来说，对象插入与读取都要考虑，所以综合考虑下来就以红黑树为主了。

## 多线程编程

1. 如何创建一个线程？实现这个线程有哪些方式？

答：我们可以通过Thread进行线程的创建和启动。

实现线程的方式有三种：

继承Thread类，并复写掉Thread下的run方法

实现Runnable接口，复写run方法

关于Callbale的一个误解，很多人会误以为Callable也是实现线程的方法，其实不然，Callable是依赖于FutureTask实现的一个接口，实际成为线程是FutureTask本身，而FutureTask本身是实现了Future接口和Runnable接口的，本质上还是实现Runnable接口

1. 并行和并发的区别是什么？并发编程的三要素是什么？如何保证线程运行时的安全性？

答：并行和并发最大的区别就是并行是在同一时刻发生的，但是可以同时执行，并不会产生相互影响。并发是指代同时产生两个任务，CPU能够处理，但是也是顺序的形式去执行，两个任务之间可能会相互受到牵制。

并发编程的三要素：

可见性：每个线程执行都是将主内存的数据拷贝到自己的工作内存区域，那么每个线程修改数据相对于其他线程来说是看不到的。

原子性：每一步操作对于CPU来说都是不可分割的，要么就全部执行，要么就全部不执行。

有序性：即两条不相关的命令会导致在处理的时候不按照代码编写的顺序去处理，我们把这种现象称为“重排序”，有序性就是指代代码的执行按照我们实际编写的顺序去执行

如何保证线程时运行的安全性：

线程执行首先保证线程之间不存在共享的数据，因为每个线程的工作空间是私有的，在提交到主内存时不能保证那个线程先提交。

其次，线程执行时，不能使用中间数据，因为线程实行代码无法保证顺序执行，很有可能被提前修改，这样会造成数据的异常修改。

最后，线程执行想要安全执行这段代码，必须要具有上述的三个特性以保证数据正确的被操作。

1. 简述一下线程和进程的区别。进程和线程有哪些区别？什么又是协程、管程。

答：线程是任务调度的最小单位，而进程是OS分配资源的最小单位。一个线程只能存在于某一个进程当中，但是一个进程可以拥有多个线程。

进程可以从OS中获取资源，而线程只能从进程中去获取资源，若存在多个线程的情况下，每条线程是共享进程的资源。但是如果有多条进程，那么每个进程的资源是相互独立的，不能共享。

在进程被杀死时，其他进程不会受到影响，但该进程下所有的线程都会停止，但是线程停止了，其他线程同样也不会受到影响，进程更不会受到影响。

在JDK17中引入了协程这一概念，而协程是比线程还要小的单位。也就是说，在单线程中可以拥有多个协程，是一种在非CPU（线程和进程都是CPU运行的单位）的运行单位，只针对某种语言而存在的一种概念。

1. 用户线程和守护线程有什么区别呢？

答：

用户线程是为我们执行日常逻辑创建的线程，能够进行数据的读和写

守护线程相当于“陪伴”，不参与实际的业务逻辑，只是作为一个陪同其他线程运行，没有实际逻辑计算，相当于一个定时器

守护线程在不会影响到JVM的关闭，一旦JVM退出，守护线程就有可能执行中断。

用户线程在没有正确执行完成之前，JVM关闭是会等待用户线程完全执行完在进行关闭。

1. 介绍一下线程的生命周期。

答：线程Thread在调用Start方法后进入就绪状态，等待CPU分配时间片。这里提一下为什么直接调用run方法无法启动线程，因为run方法只是一个能够自定义编写逻辑的方法，但是start方法会拖过JNI调用底层去开启线程，run方法不过只是一个被虚拟调用的方法。

Step1：通过Thread类start方法启动线程，进入线程等待区域等待CPU分配时间片。

Step2：获得CPU分配的时间片，进入运行状态。

Step3：如果过程中产生阻塞（Sleep），则会进入线程阻塞状态。

Step4：如果运行过程中，其他线程使用了synchronize修饰，那么会进入一种同步阻塞的状态，当前状态下不释放对象锁。

Step4：如果运行过程中需要等待（wait），线程同样会进入阻塞状态，与阻塞不同的是，该状态下会进行对象锁的释放，将修改信息提交回主内存（这也是为什么wait需要用synchronize进行修饰）

Step5：线程执行完成，main方法结束，线程运行过程中产生异常线程都会结束状态。

1. 简单介绍一下wait方法和sleep方法的不同。

答：

wait方法来源于对象，也就是所有类的父类Object，该方法调用JNI是的当前线程处于阻塞状态。使用wait方法时必须添加synchronized对象锁，主要原因就是禁止重排序，一旦对象被重排序，那么这个锁就可以在未执行任何逻辑业务之前把线程阻塞掉，导致混乱。

sleep方法来自于线程，是个静态方法，使当前运行的线程进入阻塞状态，sleep不会释放对象锁，因为他就没有对象锁，也不需要被对象锁（synchronized）所修饰，就算被修饰了，还是会保证其他线程被强行阻塞，因为sleep只针对线程，跟对象无关，所以不会对对象进行锁释放。

1. 简述一下synchronized关键字作用。

答：synchronized又叫做对象锁，synchronized是控制一个对象在多线程环境下执行一段代码段或者一个方法时只有一个线程在执行。synchronized是个重量级锁，该关键字可用于修饰代码段（方法块），方法，对象。synchronize能够保证在多线程执行的时候线程之间可见，哪怕是没有被释放对象锁也能看得到该数据被操作过，也能够满足happens-before原则顺序执行，且由于阻塞其他竞争对象的原因，是的该操作的过程是具有原子性的，不可再分割的。

1. 叙述一下synchronized锁升级

在JDK6版本以后，synchronized进行过锁的升级，采用了偏向锁的概念。所以一个对象对锁的概念保有了四种状态。无锁状态，偏向锁状态，轻量级锁状态，重量级锁状态。

无锁状态：简洁的说，就是一个线程处于未获得对象锁的一个状态，也就是锁竞争的初始状态

偏向锁：偏向锁是由于线程之间竞争时往往只有一个线程一直在获取锁，那么，当所有线程开始竞争锁，有一个线程获得了锁，那么就会为该线程的对象头加上threadID，标注这个是偏向这个线程的锁，当有其他线程开始竞争时，会去先比对threadID，如果是偏向该线程的锁，那么就获取锁继续执行，若不是，那么就要判断是否偏向锁的线程是否存活，若偏向锁线程未存活，则设置为无偏向状态，若存活，那就要判断偏向线程是否还需要锁住对象，若不需要，那么清除偏向锁，设置为无锁状态，若需要，升级为轻量级（自旋）锁。

轻量级锁：由于获取锁的时间段，且很快就释放了，那这种情况下，不可能也不肯牺牲这么大的代价去做一件小事，所以，轻量级锁就诞生了。轻量级锁就是在执行到执行代码区域时判断当前区域是否能够执行，若不能则在获取锁周边等待，若获取到了，那么就可以继续往下执行下去，这就是轻量级锁的核心概念CAS(compare and sweep比较，然后清除)，即若两值相等则继续执行，若两值不等则继续等待。

重量级锁：采用Monitor监控代码执行，也就是JDK6前的synchronized，调用底层（汇编）进行对字节文件的执行，是隶属于CPU层级的代码，每次调用代价都会很大，所以针对代码较长前不能被替换的时才会出现，一版情况下出现的都是轻量级锁。

1. synchronized为什么需要所升级，他解决了什么问题。

答：synchronized的锁升级是为了预防在较少线程争抢资源的前提下，用尽量少的资源去完成对线程操作的同步性。

首先，采用偏向锁机制，在对象头中添加对应得线程id（偏向锁会因为当前资源类拥有了hashcode而直接失效，编程轻量级锁，若在加锁过程中计算hashcode会直接变为重量级锁）可以保证若下次还是相同的线程进来的话，直接持锁上岗，不需要进行额外的加锁动作，从而提供CPU的资源利用率。

其次，轻量级锁是会解除偏向锁，采用CAS的原理对对象头中MarkWord记录的线程ID进行替换，若成功则继续持有锁，若不成功则进行自旋（参考CAS自旋的优缺点），当自旋达到一定程度后，会直接升级为重量级锁。



注：为什么偏向锁在计算了hashcode 后会直接失效呢，因为偏向锁占用了前54位用来保存线程id，那么一旦计算了hashcode，hashcode占用的31位是不可以被替换的，这种情况则不会允许添加偏向锁。

1. 简述一下volatile关键字的作用。

答：volatile是一个轻量级的同步机制，他能够保证在多线程执行时的可见性和有序性（指令重排序），volatile只能够修饰变量，能够使得在多线程操作这个变量的时各个线程之间可见，也能够是的线程操作该变量的时候不能够重排序操作，必须顺序执行，为happens-before原则提供实现的一个重要的关键字。

1. volatile能够把某些操作变成原子操作吗？

答：这是个比较特殊的点，volatile只要用来修饰原子类，就能使得该原子类在多线程中操作不会出现文件。但是这是因为原子类中每一个操作都是原子的，所以跟volatile无关。比较神奇的就是volatile用来修饰long和double，能使得long和double的操作变成原子的。

由于double和long都是64位的，所以每次操作时会变成两部分操作，如果用volatile修饰那么，就是原子的，因为对64位地址的读写是原子的（很迷，没懂，但是oracle java spec上有可以自行考证一下）

1. volatile的实际应用。

答：单例模式--双重锁校验。

这个模式下为什么要用到volatile呢，因为存在不回去竞争对象锁的线程，当这些线程获得CPU时间片后就会是一个半吊子初始化的对象，所以为了禁止重排序导致外部线程拿到了一个半吊子对象，所以必须加上volatile来修饰这个单例对象。

1. volatile和synchronized有什么区别吗。

答：

volatile是个轻量级锁，他能保证被修饰的对象操作是在各个线程是可见的且操作是不会被重排序。synchronized是一个重量级锁，他能够实现所有volatile实现的功能，且能够保证对象操作的原子性。

volatile只能够修饰对象，而synchronized能修饰对象，方法（注意修饰方法是锁住的是被调用的对象），和方法块。

volatile不会造成线程的阻塞，但是synchronized会阻塞与其竞争对象的线程。

volatile在使用时不会被编译器进行优化，而synchronized会被优化，甚至还会有锁升级。

1. volatile怎么实现可见性和重排序的。

答：volatile通过在每次读写操作加上相关的内存屏障来保证每次写入的内容和读取的内容都是和主内存中的内容是一致的，从而保证了可见性。内存屏障是指一种屏障指令，对CPU和编译器对指令前和后所发出的内存操作的一个排序约束。

内存屏障会加在所有的读写语句之间，会隔断每个操作指令的执行，使得在同一时刻下，在任意一道屏障之间只有一条指令会被执行，所以保证了不能被重排序。

写屏障（Store Memory Barrier）：内存屏障之前所有的写操作都要会写到主内存。

读屏障（Load Memory Barrier）：所有的读操作都会在读屏障之后发生，保证可见性。

读屏障会加在所有读操作之前，读屏障会让当前线程重新读取主内存中的数据，而写屏障会加在所有写操作之后，保证每次写操作完成之后，会将已经写好的数据刷新到主内存当中，从而实现了可见性。屏障就像一道障碍一样，没有对应得通行证就没法通过屏障，从而禁止后续的操作排到操作之前。

1. volatile不可以重排序是绝对吗。

答：

|  |  |  |  |
| --- | --- | --- | --- |
| 第一个操作 | 第二个操作：普通读 | 第二个操作：volatile读 | 第二个操作：volatile写 |
| 普通读 | 可以重排 | 可以重排 | 不可以重排 |
| volatile读 | 不可以重排 | 不可以重排 | 不可以重排 |
| volatile写 | 可以重排 | 不可以重排 | 不可以重排 |

当volatile修饰的变量与普通变量相互交缠时，还是会出现重排序，但是必须遵守happens-before规则，volatile修饰的内容所有读操作之后，都不可以进行重排序，而所有的写操作之前不可以重排序

注意：上述说的重排序是将第一个操作和第二个操作颠倒后是否还满足happens-before原则

# 并发编程（JUC）

## Java并发基础与原理

1. Java线程内内存模型（JMM Java Memory Model）

答：每个线程在操作数据的时候都是将数据的副本从主内存空间拷贝到自己的工作空间，在自己的工作空间操作完成了以后，在提交到主内存中。所以每个线程的工作空间都是相互独立的，不可见的。

缓存一致性协议（mesi）或总线嗅探机制是JMM用来保证主内存和副本内存之间读写的一致性。

1. 介绍一下多线程并发的三个特性。

答：

可见性：可见性指代的是，线程和线程之间资源都是来自于自己的工作空间，并不是直接操作主内存的，也就是线程1操作自己工作空间的数据相对于线程2来说是不可见的。

原子性：原子性是指这一步操作是最小的，不可在分割了，要么全部执行，要么全部不执行。也就是说这一句或一段语句具有原子性的话，那么，这一句或一段话执行不会被同类型的线程所打断，要么线程完全执行，要么异常退出，不存在执行一半后去执行其他相同类型的线程。

有序性（重排序）：这个特性来自于编译器自发的对线程并发操作的优化，遵守happen-before原则，也就是如果某一句话语之前执行的部分无关请速度很快，那么就会把这句话的执行顺序提前，就造成了重排序，重排序在某些情况会造成初始化没完成就将对象交出去了，导致外部线程拿到的是个半吊子的数据。

1. 简述一下happens-before原则和as-if-serial原则。

答：这两个原则都是面向重排序（针对CPU原语，即汇编）时编译器给出的一些约束。实际上只要最终结果一致就行，用户程序员不担心内部是否被重排序，只要最终结果没被影响即可。

happens-before原则：多线程情况下，给编写者提供一个约束，操作A必然先于操作B执行。操作A在线程A中，操作B在线程B中。那么由于happens-before原则，线程A中操作A一定会先于线程B的操作B，无论如何重排序，都是操作A先于操作B。

as-if-serial原则：单线程的情况下，数据之间存在相互依赖关系，那么编译器不会进行重排序，给编写者提供一个约束，操作A必须执行与操作B之前，那么操作A不管怎么执行，一定是会先于操作B。

举例：

主线程执行对象实例化操作，然后对对象进行set的方法调用。

那么由于as-if-serial原则，那么在单线程执行模式下，实例化操作也一定会先于set操作，不能够出现set操作先于实例化操作。

同样的在多线程条件下，由于happens-before原则，一个线程实例化对于其他线程可见，那么set操作在使用时（假定线程操作实例化已经完成，且被volatile修饰）那么实例化操作的写入主内存一定是先于set操作从主内存中读取对象。

1. 知道happens-before的六项规则吗？Happens-before与as-if-serial有什么区别。

答：简单概括就是。线程执行时顺序的，解锁先于加锁，写入大于读取，礼让先于阻塞，中断先于中断检测，创建先于回收。

程序执行时顺序的，后面的操作需要用到前面的结果，前面操作先于后面的任何操作。

获取锁时，第一个获取锁的解锁操作一定先于第二个获取锁的加锁操作。

Volatile控制域下的操作，写入操作一定先于读取操作。

一个线程的启动操作（start方法）一定是优先于线程任务执行的内容

Join方法执行返回成功让当前线程阻塞，则礼让的线程执行先于当前线程执行join返回成功。

线程使用interrupt方法先于线程检测到被中断的指令。

new一定是先于这个对象被回收时调用finalize

happens-before与as-if-serial最大的区别就是在于，happens-before针对的是不同线程之间也会生效，但是as-if-serial是针对当前执行的线程生效。他们都是为了实现再不改变程序执行结果的情况下提高并发效率。

1. Synchronized有什么作用。

答：synchronized是java中的一个关键字，被其申明的可以有方法和方法块。被声明的方法在同一资源类下，多个线程只能有一个synchronized方法可以执行，从而来实现锁的机制。Synchronized底层是应用了monitor进行对方法或方法块的监视（重量级锁）。往往一个monitorEnter对应两个monitorExit，主要的原因是在于能够在代码块出现异常的情况下，完成锁的释放。极端条件下会出现（即手动抛出异常）一个monitorEnter对应一个monitorExit。

1. 为什么每个对象他都能成为锁对象。

答：因为在java中，所有对象都来自于object，而object拥有一个object monitor，通过这个可以用来实现object的加锁机制，所以只有当对象再被加锁的情况下，object monitor会发挥他的作用，使得对象头中的owner记录当前使用的线程。这也是为什么wait和notify等方法必须要在synchronized中执行，因为你需要明确竞争的对象 ，从而来对不同的线程进行记录谁在使用当前对象。

1. 什么是公平锁，什么又是非公平锁。为什么定义锁大多都是非公平的。

答：公平锁是指，每个线程进入锁竞争环节时会进行排队等待，依照顺序执行队伍中的任务。而非公平锁则是反之，当开始锁竞争时，会先于当前队列中排名靠前的线程进行竞争，当竞争失败后，转入队列中排队。定义锁时，选择非公平锁的原因大致有两点：

非公平锁很大可能是同一个线程获得锁资源，从而可以减少CPU线程间上下切换从而提高效率。

非公平锁可以减少CPU切换，能更好的利用CPU的空闲时间。

1. 什么是unsafe类，他解决了什么问题。

答：unsafe类是一个可以直接操作内存的一个类，由于本身是不安全的，所以起名unsafe。因为java是不能直接操作内存的，而unsafe实现直接操作内存，执行cpu源语，帮助代码更容易实现CAS功能，CAS能有效的避免锁的重大开销，采用自旋的方式来保证实现轻量级锁。

1. 解释一下自旋的定义。

答：自旋是一种自定义非阻塞的实现机制，由于在执行某段代码时需要单独执行，但是加上synchronized的代价太大，不仅会导致其他线程阻塞还会造成资源消耗大，而且执行的代码段速度很快，这样造成了极大的资源浪费。而自旋就是指我不采取阻塞，而是采取在需要加上synchronized的周围做循环等待，等到执行完成，获取到资源，则继续向下执行，极大的减少了资源的消耗。

1. 说一下CAS的优缺点。

答：CAS（compare and swap）是来自于CPU源语级别的操作指令，是通过比较当前值是否正确在判断是否能赋值成功的一个概念。

优点：降低了锁的量级，采用非阻塞的方法使得程序不会阻塞，也能保证数据的最终一致性。

缺点：因为偏向锁的缘故，容易导致cpu空转，同时还会连带ABA问题，导致中间被修改过，但是结果没变，导致结果虽然最终一致，但中间的改动却被忽略了。

1. 简述一下AQS的原理。

答：AQS是抽象的队列同步器，它是一个抽象的队列，抽象了大多数锁实现的一个基本方案。它采用CHL队列的形式保存已经入队的节点，并将需要执行的线程封装到节点当中。

AQS采用双向链表进行实现，他需要针对当前执行节点（也就是head指向的节点）进行虚拟占位，通过后续节点对前一个占位节点的状态进行修改，来保证前一个节点能够准确的被唤醒，同时还为了避免从头部开始遍历，但是过程中仍然在插入元素导致无法遍历到队列尾部。

1. AQS帮助我们解决了什么问题。

答：AQS作为一个队列同步器首先帮助我们实现了对锁的一个具体抽象内容。

首先，AQS可以实现公平与非公平锁，保证各各个线程都能获取到资源，雨露均沾。相较于原来的同步块或者锁，是通过notify随机唤醒一个线程来保证线程的执行，也就是说并不能做到雨露均沾。

其次，AQS通过自定实现方式提供排它锁和共享锁这个功能，优化了读多写少的场景，在读操作并发量较高的情况，synchronized同步块仍然会被阻塞，这样效率相对地下，可以采用共享锁来提高读的效率。

## 并发容器与原子类的应用

1. 原子引用是什么，解决了什么问题。

答：原子引用是对当前对象进行一个封装，通过底层的unsafe类来实现非阻塞的锁机制，帮助在竞争激烈的情况下，提高CPU的利用率。原子引用通过加volatile实现对象的线程间可见来保证多个线程在操作同一资源类的情况下内容能够及时的刷新到主内存中，在通过CPU的底层操作实现CAS（compare and swap）自旋锁的模式实现非阻塞的锁竞争。

1. 原子引用怎么解决ABA问题。

答：ABA问题就指的是一个对象原有的值为A，多个线程竞争的过程中，有一个线程改为了B，但是由于其他原有又改回了A，过程中已经被修改过了，但是最终的结果却没有变更，使得其他线程误以为对象的原有值没经过改变，并进行更新成功。这说明原子类一般情况下只能保证最终一致性，但无法保证过程中没有过修改。解决的方案则是使用戳记流水或版本号来标记当前值是否被修改过，或者通过修改标记来判断当前过程中是否有其他线程修改过来解决ABA问题。相关的原子类就有AtomicStampedReference（戳记流水）和AtomicMarkableReference（修改标记）。

1. 原子更新类是什么，有什么作用，解决了什么问题。

答：原子更新类只会更新这个类中对指定的字段，以一种线程安全的方式来更新非线程安全的对象的一个字段。他通过对某个高频竞争的字段实现非阻塞的锁定，从而更细粒度的对字段进行加锁，避免了多个线程访问同一资源类的时候访问不同的字段，都要去参与所竞争，减少了锁竞争的资源消耗，实现了细粒度的锁定。

1. 说一下LongAdder和LongAccumulator为什么相较于AtomicLong要快。

答：LongAdder是AtomicLong的增强类，在高并发的情况下，自旋锁会引起很多线程处于空转的状态从而引起CPU突然升高，为了解决这个问题，LongAdder采用了热点分散的方法有一个base值作为初始状态，一个cell数组作为扩容点，当base达到一定访问量了以后，会开启cell数组，并让其他的线程命中cell中的数组，从而分散了数据访问，减少了各个线程自旋空转的现象。最终通过value = base + Σi=0Cell[i]，进行求和给出最终结果。

1. 简述一下JDK1.7与JDK1.8的ConcurrentHashMap实现有什么区别。

## 线程池

1. 什么是线程池，线程池有什么作用。

答：线程池是用来管理多个线程的一个类。Java为了避免无节制的创建线程，没有办法很好的管理线程的使用而产生的一个管理工具。

线程池是帮助客户程序员来管理线程的申请和使用的。一般情况下我们想使用多线程进行并发操作时，一味的去创建线程，但是并没有关注过线程是否能够及时回收从而导致CPU长期飙高。所以线程池应运而生，它是用来管理线程的创建和回收，以及线程任务的执行。它可以通过有限线程数量和阻塞队列来帮助我们用最小的代价实现并发操作。我们可以设定线程池的核心线程数和最大线程数来执行相关任务，当我们的并发的任务到达极限时，可以投入阻塞队列中，稍后执行，这样来节省CPU资源，减少线程的申请和回收提高线程利用率。

1. 简述一下线程池的工作过程。

答：线程池在创建之初我们需要定义一些基本参数：

corePoolSize：核心线程数

maximumPoolSize：最大可执行线程数

keepAliveTime：每个最大可执行线程数存活时间

timeUnit：时间单位

workQueue：阻塞队列

threadFactory：线程创建工厂

handler：拒绝策略

核心线程是一些常驻的线程资源，不会主动消失。它是用来执行往线程池中投入的任务的中坚力量。当核心线程数不足以满足执行的任务的数量时，会将新来的任务投入阻塞队列中，保证任务不会丢失。当阻塞队列到达极限后，线程池会逐步提高线程数至最大线程数，用来满足任务的处理。假设在最大线程数拉满后还是不断的有任务往阻塞队列里塞，那么就会触发拒绝策略，从而导致在默认拒绝策略下代码报错和任务丢失。

注意：最大线程数是处于空闲状态后，一定时间（keepAliveTime）会被直接销毁，而核心线程数不会。还有为什么不先开启最大线程数而是走阻塞队列，因为创建线程是需要消耗大量的CPU资源的，同时线程回收也一样 ，那么我们就应该有限避免线程的创建，不到万不得已才触发。

1. 在jdk1.8中所提供的创建默认线程池的方法有哪些，分别有什么用。

答：一共有4种：

newFixedThreadPool：创建一个固定数量的线程池

newCachedThreadPool：创建一个单缓存任务的线程池

newScheduledThreadPool：创建一个定时执行的任务队列池，与常规线程池无异，但是往阻塞队列里边添加内容时会根据对应的堆二叉树进行排序。注意，SecheduledThread不适合使用ExecutorService去接收，更适合用ScheduledExecutorService.schedule

newWorkStealingPool：创建一个forkjoinpool的线程池

1. 简要的说下线程池的优缺点。
2. 为什么不推荐让我们直接使用JDK自带的线程池初始化方式。
3. 常规线程池与ForkJoinPool有哪些区别。各自的优劣说一下。
4. ThreadLocal与线程池会出现什么问题，同时ThreadLocal为什么采用弱引用。

答：

1. ThreadLocal是怎么解决 内存溢出问题的。

答：ThreadLocal内存溢出是因为ThreadLocal是一个弱引用，当发生GC时，弱引用就会被回收，而线程中ThreadLocalMap的key就会变为null，这样会存在非常多的value不为null但key为null的无法获取的数据，所以在ThreadLocal中强制我们在ThreadLocal中使用完毕后一定要进行remove，这样才能够基本保证能够定时清空线程中ThreadLocalMap有问题的内容。同时为了保险，ThreadLocal在再添加元素的时候也会去检测是否存在key为null的值，来保证使用过程中清空对应的无效内容。

## 并发工具

# JVM性能调优

## 面向对象

1. 什么是对象，谈谈你对new Object()的理解？

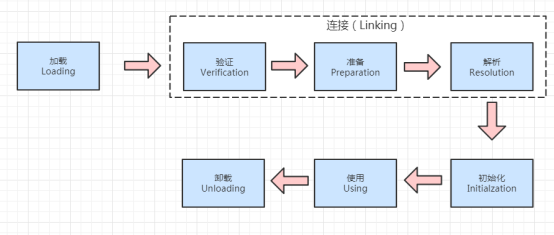
答：new Object是将实例放到栈内存中的一个操作，一个常规的对象大小（没有任何实例数据的情况下）为16字节。而一个对象分为三部分，对象头（Header），实例数据（Instance Data），对齐填充（Padding）组成。

对象头分为两部分组成对象标记（MarkWord）和类型指针（Class Pointer）。对象标记里边记录了HashCode、GC标记和次数以及锁的标记和次数。为了GC回收和锁的升级提供了一些响应的数据存储，用来完成标记一个对象

## 类加载过程

1. 简要描述一下类加载的过程？

答：类加载过程是指一个class文件通过类加载子系统加载到方法区（永生代）的一个过程。主要分为加载->连接->初始化->使用->卸载等几个步骤，连接又分为验证->准备->解析三个步骤。



1. 类加载时，连接过程主要完成了什么事情。

答：连接过程分为三个部分，验证，准备，解析。

验证：验证待加载的class文件验证其是否被修改过，比如文件格式，字节码格式等。

准备：将class文件的类对象初始化出来，并附上初始值，想int类型赋上0，引用类型赋上null等。

解析：解析是将符号引用（全类名）解析为直接引用（内存地址）。

1. 类加载器有哪些。

答：常用的类加载器有BootStrapClassLoader（启动类加载器）、ExtClassLoader（扩展类加载器），AppClassLoader（应用类加载器）。

BootStrapClassLoader加载jre/lib下的jar内容，属于启动加载，非java代码实现的内容。

ExtClassLoader属于扩展类加载器，加载jre/lib/ext下java自身的扩展内容。

AppClassLoader主要加载我们自己实现的类对象，加载路径是classpath指定的路径。

1. 什么是双亲委派机制，为什么要使用双亲委派机制。

答：双亲委派机制是指一个class类加载时，会先由父类加载器去尝试加载，若不存在则将其交给子类加载器，最终交给自定义加载器。

具体的加载过程是先将class文件交给应用类加载器，然后应用类加载器再将其交给扩展类加载器，扩展类加载器再将其先交给启动类加载器，由启动类加载器优先加载，若启动类无法加载则交给扩展类加载器，扩展类加载器无法加载最后才会交给应用类加载器。

双亲委派机制可以避免类的重复加载，同时还可以避免核心API被篡改。

1. Tomcat为什么要自定义一个WebAppClassLoader。如何破坏双亲委派机制。

答：Tomcat为了保证多实例部署的情况，则将应用类加载器进行了扩展，从而使运行时所使用的类得到隔离，保证同名的类不被判断为重复加载。

破坏双亲委派机制只需要进行继承应用类加载器，在此基础之上，去复写loadclass方法，是的不去向上寻找自己的父类加载器，也就是一个class文件只需要应用类加载器去加载，不需要扩展类加载器去加载，跳过扩展类加载器，这样也就破坏了双亲委派机制。

## 运行时内存模型

1. 简要的描述一下jvm在运行时的内存模型。

答：jvm在运行过程中，他的内存模型分为，堆、栈、方法区、程序计数器。

堆：用于存放对象实体是线程中共享部分，分为新生代与老年代。新生代分为eden区和survivor区

栈：分为两个部分，本地方法栈与方法栈，都是属于线程私有的。本地方法栈用于存放java中一些native修饰的方法。方法栈则是我们自己内部实现的方法，通过栈帧来保存方法的嵌套。

方法区：方法区也称作非堆内容，它是线程共享的。方法区保存了一些类信息、常量、静态变量。

程序计数器：用于记录代码执行到那一段，也就是程序执行的行数。

1. 虚拟机栈（方法栈）由那些部分组成，分别的作用是什么。

答：每个线程的虚拟机栈都是私有的，而一个栈帧代表着一个执行的方法。一个栈帧中包含局部变量，操作数栈，方法返回地址，动态连接，附加信息等内容组成。

局部变量表与操作数栈一般是配合使用，局部变量表，操作数栈是用来保存了操作过程中产生的值与数据，在通过相应的指令进行计算，都在操作数栈完成，而局部变量表是用来保存相关的局部变量的信息。

1. 堆以及堆中各个区域相关的内容。

答：堆存放的主要是对象实例，数组，当执行字节码操作时，会把对象放置堆中。堆中分为新生代与老年代，新生代又分为三个区域，Eden区，Survivor0区，Survivor1区，对应的比例是8:1:1。

新生代：存放创建的新对象，其中Eden区存放各个创建的对象，Survivor0和Survivor1是保存一些逃过GC回收的一些对象。

老年代：存放一些逃过GC回收15次（对象头中会记录GC次数）对象，这些对象会从新生代中转向老年代。其中一些巨大的对象创建时发现Survivor0和Survivor1无法存放，这样会直接转向老年代。Eden也是同理，一个对象足够大，会直接从Eden往老年代存放。

## 垃圾回收器

1. 常见GC共有几种类型。

答：共分为youngGC，oldGC，fullGC三种类型。

youngGC：也称作MinorGC，主要是回收新生代的一些垃圾对象。

oldGC：也称作MajorGC，负责老年代GC回收，其中只有CMS垃圾收集器会单独对老年代进行回收，其他的垃圾收集器都是等到整堆回收时才会触发。

fullGC：整堆回收，同时也会对方法区（永久代，非堆内容）进行回收。

1. 常见的垃圾回收器有哪些

答：串行垃圾回收器（Serial）、并行垃圾回收器（Parallel）、并发标记扫描垃圾回收器（CMS）、G1垃圾回收器（Garbage-First）四种。

有区别于young gc，old gc，串行垃圾回收器与并行垃圾回收器又可以区分为

1. 为

# Mybatis/Spring Data Jpa

# Mysql

## 索引

1. 什么是索引？索引有哪些数据结构？

答：索引是可以帮我们快速查找数据的一种数据结构。常用的索引有Hash表和B+树（多路平衡查找树），innoDB默认情况下使用的是B+树。

1. Mysql添加索引为什么会变快，而索引为什么要采用B+树这种数据结构，而不是别的数据结构，例如：平衡二叉树，红黑树，B树

答：mysql存储数据到磁盘上是随机存储的，所以在查询的时候需要通过redo.log去定位相关的数据位置，未添加索引时mysql只能依次遍历所有元素，没遍历一个元素，mysql就会与磁盘进行一次IO操作，当数据体量非常大时，mysql进行IO的次数会急剧上升造成查询的时间增加。所以我们通过添加索引可以使得我们想要查找的目标元素去通过数据结构快速查找到目标数据，从而省去了多次IO所消耗的资源和开销。

为什么需要使用B+树

首先，存储索引的时候mysql不会存相关的数据，而是每一组索引的开头索引元素，通过这个进行排序，每一级都有严格的排序，然后通过折半查找进行快速定位，把这些内容加载到内存中，每一次查找只需要查询到叶子节点然后加载到内存中进行取数，所以整个过程只进行一次IO即可。

二叉树：当我们顺序添加元素时，二叉树会退化成为一个链表，不仅索引本身不生效，而且索引本身也会占用一定的空间。

红黑树：本身是二叉平衡树，当数据体量增大时，红黑树的高度会很高，当查询叶子节点的情况下，我们的IO次数一样会提高

B树：B树是对红黑树的横向扩展，先以每一个根节点为目标索引元素，在定位了目标元素后，再去叶子节点查找目标数据。而B+树是在该基础上进行了一定的改进。

1. 当数据体量本身很大的情况下，此时我去添加索引会发生什么事情。

答：数据体量本身很大的情况下添加所以，mysql需要去生成对应的B+数，此时第一次添加索引会导致数据库进行一次全表扫描，耗时会非常的长，资源消耗也会巨大。

1. Hash表索引和B+树有什么区别，谈谈各自的优缺点？

答：Hash表是将数据进行Hash值计算，获得相应的键值，在进行回表查询（使用非聚簇索引查询了到大量不相关的列而导致需要获得其聚簇索引在进行一次查询称之为回表查询）获得实际数据。B+树是一个多路平衡查找树，每次查找都从根节点开始查找其叶子节点最后在通过判断是否需要进行回表查询来获取数据。

Hash表索引可以很快的得到等值查询的结果，但是在范围查询上几乎失效。

由于Hash索引会计算哈希值，所以再进行范围查询的时候需要一个一个去比对，没有正确的顺序，导致很难查询到范围。

Hash索引不支持排序，因为本身就是散列存储，并没有顺序。

Hash索引无论什么条件下都必须执行回表查询，但是B+树却不需要，可以通过聚簇索引或者覆盖索引判断是否需要回表。

Hash索引在性能上不一定稳定，一旦出现大量重复值，会导致产生大量hash碰撞（就是计算出来的hash码相同，不能确定其是否是同一条数据）从而增大系统开销。

1. 在B+树的索引里，为什么可以支持区间查询。为什么范围查询的索引时走时不走。

答： B+树的叶子节点在innoDB是一个双向连表（B+树本身不是），此时，当命中该条索引，回去先查询相等的值，然后通过向后遍历叶子节点的值就能快速查询出当前命中索引范围内的数据。此时要注意的时，mysql底层是有一个查询优化器，当你的范围查询查询到了一个相对较大的范围（innoDB底层去查询多个数据页从而导致io次数太多），且不会命中其他索引的情况下，会变成全表扫描。此时若还想命中当前索引的话有以下解决方案。

减少IO次数，使用聚簇索引的方式可以减少IO次数，

提高范围查询的区分度，使得查询出来的数据尽可能的在数据页的范围内。

1. 什么事聚簇索引？什么是覆盖索引？

答：

聚簇索引：在B+树种，有些叶子节点会只存储Key值，有些节点会存储其Key值和整行数据。存储了整行数据就说明这个Key是唯一的，则称之为聚簇索引。主键就是典型的聚簇索引。如果不存在主键，Mysql会以第一个遇到的唯一非空字段进行聚簇索引的建立，如果以上两个都不存在，Mysql怎会隐式的创建一个键作为聚簇索引。

覆盖索引：如果当前查询所需要的字段都能在当前数据中找到，那么就把这种索引叫做覆盖索引，覆盖索引不需要进行回表查询。（这种也是三星索引的判断标准之一）

1. 联合索引是什么，为什么要注意联合索引的顺序问题？

答：联合索引就是多个字段组合起来成为一个索引，这个索引就称为联合索引。要使得联合索引生效则不需遵循联合索引创建时的顺序，也就是“最左原则”。Mysql对联合索引的顺序是有强制要求的，比如一个索引具有name，age，sex三个字段，联合索引就是先去查询name列的key，在使用结果去查询age列的key，以此类推。

注：联合索引不宜超过三个，本身该查询就类似于级联查询，进行三次异常级联不亚于连表三次以上

1. 为什么要建立联合索引，联合索引的效率为什么会比单体索引的效率高？为什么要遵守最左原则。

答：联合索引是以多个联合条件为一个节点进行排序的（底层B+树必须是有序的），在查询时会会从最左边的进行匹配，然后再在匹配出来的结果上让第二个元素进行匹配，直至条件中没有改联合索引的字段位置从而获得结果集在进行非索引字段的匹配。而单挑索引只会匹配一次，匹配到的结果集直接去进行非索引字段的匹配所以，联合索引的查询效率会比单体索引的要高。

Mysql在匹配索引时会有一个最优索引策略，每次只会使用一个索引进行查询，所以单体索引就算有多个也不能多次命中。

最左原则是为了满足mysql在匹配联合索引的情况下的一种准则，需要数据区分度大的字段会放到左边，依次从左向右排列。在mysql底层索引的结构中，联合索引匹配原则是从左至右依次匹配的，所以为了更容易命中联合索引。

注意：最左原则必须要命中最左边第一个元素该联合索引才会生效，因为mysql索引的底层数据结构中，所有叶子节点都是通过链表连接起来的一个有序的数据集，当最左边的字段命中之后就可以锁定范围了，之后继续向后遍历即可。当没有命中第一条数据时，其向后遍历的难度大大增加，所以在mysql的决策下是不会去命中该索引的。

1. 建立索引需要注意些什么？什么情况下会导致索引失效？

答：建立索引首先就是查询高频关键字，这类关键字因为使用频率非常高，所以建立索引能加快高频查询字段的查询询速度。其次，就是非空唯一的常用字段，这类字段可以有效的避免回表查询，加快查询的速度。最后，如果建立的是联合索引，按照查询频率一次向后排列，因为联合索引遵循最左原则，查询时先以最左边的字段进行索引筛选，而且联合索引不需要所有的字段都必须在条件列表里，而是是否存在这个关键字，一旦存在联合索引就会触发。

索引失效的情况如下：

使用了不等于，或（or）等诸如此类的取反关键字，所有会失效

参与了各种列之间的运算

范围查询命中索引，但是区分度不大导致mysql不会决策去走该条索引

使用了like条件，但是通配符却需要前后匹配，这种情况会进行全表扫描

数据少到必须全表扫描，这种情况比较常见于单机测试

1. 当一条sql查询命中了索引，简述一下索引查询数据的过程。

答：先将数据去\*.MYI文件查找，通过折半查找快速定位到所存储的数据地址，然后获取到数据本身（只有索引部分），若想获得全部数据，则需要再次进行通过唯一值查询。

## 事务

1. 什么是事务？ACID分表代表着什么？

答：事务就是执行一些列操作，在执行过程中要么就完全成功，要么就完全失败，而这一过程我们把它叫做事务。事务在执行时必须满足ACID四个特新，否则在执行时会出现各种各样的问题。

A=Atomicity（原子性）

事务的执行是最小粒度的，不可在分割的，要么就全部完成，要么就全部失败，保证自己是一个整体

C=Consistency（一致性）

数据从一个一致性的状态转移到另一个一致性的状态，不会存在中间状态

I=Isolation（隔离性）

一个事务从开始到最终提交这个过程对于其他事务是不可见的（也会有例外情况，例如隔离级别为读未提交时，隔离性几乎失效）

D=Durability（持久性）

事务一旦提交，那么这个数据就是永久存在的（除非你把它删了），不会影响到事务本身

1. Mysql的锁了解吗？具体有哪些锁？

答：Mysql为了保证在并发情况下，数据不一致，所以使用了锁机制来保证数据的一致性。再次通过两个维度来说明mysql锁的作用，先以读写方面来看：

共享锁：S锁，读锁，加锁后数据只能被执行读操作，无法执行更新或删除操作。一条数据可以被重复加读锁，但是不能加写锁，也就是排它锁。

排它锁：X锁，写锁，加锁了以后能对数据进行读写更新等操作，加了写锁就不能在加其他的锁了。

1. 多个事务并发会产生什么现象？怎么解决这些问题？

答：多个事务并发会产生一下三种情况：

脏读（不满足隔离性）：

A事务读取了B事务未提交前的内容，A事务理应读取B事务提交后的内容，但是却读取到了未完全提交的内容，那么则称之为脏读

不可重复读（不满足一致性）：

A事务（有两次查询）与B事务操作数据时，A事务第一次查询了B事务提交前的数据，当B事务提交后，A事务第二次查询时结果与第一次不一致。

幻读（不满足隔离性）：

A事务读取一个范围数据，但是B事务向这个范围内插入了一条数据，就产生了“怎么还有有一条数据没做修改的‘幻觉’”，我们把这种幻觉叫做幻读。

为了解决以上三种情况的出现，使用事务的隔离级别

READ UNCOMMITTED ---- 读未提交

该隔离级别就是允许其他事务读取当前事务未提交的内容，放弃了事务所必须的隔离性。是产生脏读的罪魁祸首。使用排它锁实现，对需要读写的数据加上排它锁，能够写数据，不能够加读锁，但是排它锁本身有读功能。

READ COMMITTED ---- 读已提交

该隔离级别只允许其他事务读取自己已经提交的内容，否则只能读取该事物未提交所有事务之前的内容，事务的操作过程对其他事务不可见。可解决脏读造成读取其他事务提交一部分的情况。使用瞬间共享锁和排它锁实现，写入数据必须加写锁，读取数据用读锁，读取数据完成后立马释放读锁，不需要等待事务处理完成。

REPEATABLE READ ---- 可重复读

可重复读保证了数据在执行过程中不允许被其他事务所修改，只能等到当前操作数据的事务完全提交后才能进行下一个事务的操作。使用排它锁和共享锁共同实现，读取数据加上读锁，写数据加上写锁，但是与读已提交（READ COMMITTED）不同的是，读锁要等到事务提交后才进行释放，所以避免了还在读数据的过程中被其他事务修改并提交

SERIALIZABLE ---- 串行执行

所有事务的执行都遵循串行执行，不允许插队，不允许重排序。类似于Java中的Synchronize同步块，效率极低。

1. InnoDB默认的隔离级别是什么。

答：默认情况下，InnoDB提供的隔离级别是可重复读。

1. 说一下InnoDB里的MVCC（Multi-Version Concurrency Control）是什么。

答：字面意思，多重版本控制。在某一时刻多个用户去读取MySQL中的数据只是当前版本的一个快照，当数据被修改完成了以后，不会删除快照，而是将它标记成“已过时”，然后将新的数据生成新的快照，然后定期进行过时数据的清理。这样一个数据就会存在多个版本了，但是至始至终只存在一个最新的版本的数据。提高了并发访问MySQL的吞吐量。

MVCC的实现是通过每张表新增两个隐藏的列，数据行版本号（DB\_TRX\_ID）和删除版本号（DB\_ROLL\_PT）通过版本号进行读写控制。

CURD四种情况：

C（create，插入操作insert）：

DB\_TRX\_ID取当前版本号，DB\_ROLL\_PT取值为null

U（update，更新操作update）：

新的数据DB\_TRX\_ID取当前版本号，DB\_ROLL\_PT取值为null。旧的数据DB\_TRX\_ID版本号不变，DB\_ROLL\_PT取新数据的版本号

D（delete，删除操作delete）：

DB\_TRX\_ID值不变，DB\_ROLL\_PT取当前事务版本号，标记为已过时。

R（retrieve，查询操作select）：

DB\_TRX\_ID版本号必须小于或等于当前版本号，DB\_ROLL\_PT为null或者大于当前事务（这里指查询事务）版本号，保证查询到的数据是其他事务之前，被删除之前的数据。

1. MVCC底层是如何实现的。

答：在MVCC的实现就是指通过readView视图来维护一条数据的版本链，在可重复读和读并提交的隔离级别下才能进行。

ReadView是innoDB自己维护的一个视图，在可重复读的隔离级别下只会针对这一查询sql进行一次查询，之后相同的sql就不在进行readView的更新了。而在读并提交的隔离级别下，每一次查询sql都会重新生成readView，所以当一个事务提交以后我们还能看到最新提交的那个事务可以被其他事务查询到。

readView会帮助我们维护一个版本链，版本链是保存于undo.log之中，在可重复读的隔离级别下，每一次查询都会先去选取readView中的最新一条的事务ID，然后通过版本进行规则比对，知道查询到最新一个可见的数据为止，在同一个事务中进行的查询会以第一次查询生成的readView为主。

readView判断当前事务是否能够展示的的规则如下：

取当前readView中最新的事务id。

判断当前事务id是否处于最小值（未提交事务的最小值）和最大值（创建于readView中的最大事务id）中，大于当前事务最大值的事务id均为不可见状态，而小于当前最小值的事务均为可见状态。而处于最小值和最大值之间的闭区间的事务id分两种情况讨论。若当前事务ID处于未提交事务数组中，则不可见。若当前事务ID处于已提交事务数组中，那则为可见状态。

1. 在InnoDB中，如何使用MVCC和Next-Key lock来保证MySQL吞吐。

答：InnoDB在Repeatable Read（可重复读）的隔离级别的条件下，维护MVCC来控制并发情况下数据读取和写入数据的读写隔离，但是只是能够保证数据读取是多版本的，但是无法保证数据产生其他的情况，例如在读取时插入一条数据，下次在读取时读取的数据多了一条就产生幻读，那么我们需要Next-Key lock来保证查询范围内的数据不能够被操作，就保证当前隔离级别下，保证数据的一致性。

## SQL优化

1. Mysql有哪些引擎？

答：MySQL支持多种存储引擎，比如InnoD，,MyISAM，Memory，Archive等等。在大多数的情况下，直接选择使用InnoDB引擎都是最合适的，InnoDB也是MySQL的默认存储引擎。

1. Varchar和char有什么区别。

答：varchar是可变长度字符串，每次查询都要计算长度。但是char是一个定长类型，不需要做长度的计算，所以在查询时，char的查询效率要略高于varchar。

1. Mysql的binlong有几种模式？分别有什么作用？

答：statement，rows，mixed

Statement：记录单元为SQL语句，由于每一句SQL执行都是有上下文的，所以上下文也会被记录下来

Rows：记录每一行的改动，每一行数据发生变化，则使得记录会很多。Mysql默认情况下为Rows级别

Mixed：折中的方案，普通操作使用statement，除非statement无法记录，则使用rows

1. 对于大分页查询，一般怎么应对？

答：《阿里巴巴代码编写规约》上说明，limit的用法offset到第N行，取offset+n行数据，然后放弃前面的第N行数据，当offset非常大的时候，无异于全表扫描，那么我们可以使用无回表查询的方式，先查询到limit以内的数据，然后使用id进行查询如：

select \* from user where age > 20limit 10000,10

==>

select \* from user where id in (select id from user age > 20 limit 10000,10)

1. 简要说明一下三范式

答：

第一范式: 每个列都不可以再拆分.

第二范式: 非主键列完全依赖于主键,而不能是依赖于主键的一部分.

第三范式: 非主键列只依赖于主键,不依赖于其他非主键.

1. 谈谈你对mysql性能优化的理解

答：mysql的性能优化站在开发的角度我从以下几个角度去理解

从结构方面入手，优化表结构与业务的关联性，减少关联查询。增加读写分离避免读写过程中锁的开销。可以针对热点数据新增市面上一些高性能的分布式数据库，如redis，MongoDB等来缓解mysql性能开销，也可以适当的采取分库分表的策略，分库可以使得数据和业务模块之间良好的隔离，分表可以下降单表数据量级，提高查询速度。

从执行sql上入手，可以对慢sql进行日志记录，并使用mysql自带的执行计划分析sql执行过程中可以避免和优化的点，从而正确的使用索引。可以重点关注type，rows，key，从而提高sql的执行效率。在查询过程中要尽可能避免回表查询，使用聚簇索引或者实现覆盖索引，减少mysql I/O次数。

## InnoDB与MyISAM

1. Mysql的引擎有哪些，默认情况下的引擎是那个？

答：InnoDB，MyISAM，Memory三种。

InnoDB：提供了ACID事务的支持，同时还提供了行级锁和外键支持。现Mysql默认使用的引擎。

MyISAM：原MySQL默认引擎，但是相较于InnoDB缺少了行级锁和外键，同时也不支持ACID处理事务。

Memory：直接将数据存储到内存里，类似于Redis这种，但是安全性很难得到保障。

1. 在Mysql中，上述的引擎是针对与表还是针对于库亦或者是别的

答：是针对于mysql的表的，每张表都有属于自己的引擎模式，在建立表的情况下默认情况下是选择innoDB，只有在特殊标注情况下，才回去选择MyISAM和Memory

1. 简述一下InnoDB与MyISAM两个引擎有什么区别。

答：从MySQL常接触到的点进行比较

事务：InnoDB是支持事务的，但是MyISAM不支持事务。

索引：InnoDB的索引分为聚簇索引和非聚簇索引，MyISAM只有非聚簇索引。另外，MyISAM是不支持哈希索引的，但是支持全文索引（InnoDB不支持）。

存储文件：InnoDB存储两种文件（.frm表定义文件，.idb数据文件），但是MyISAM存储使用三种文件（.frm表定义文件，.myd数据文件，.myi索引文件）。

查询语句（select）：MyISAM的查询速度相对快一些（因为只需要维护三个文件即可）。InnoDB在并发相对较低的情况下略慢于MyISAM。

锁支持：InnoDB支持到行级锁，MyISAM只能支持到表级锁。

1. 说下InnoDB与MyISAM锁的区别。

答：

InnoDB的支持行级锁（row-level lock），冲突很少，但是开销很大。行级锁就是前面提到的行级共享锁和行级排他锁。

MyISAM的锁只能支持到表级锁（table-level lock），冲突非常多，开销很小。但是处理速度很慢，并发量会很低。

介绍一下页级锁来自BDB引擎，页级锁是行级锁和表级锁的折中方案，降低了冲突，同时耶降低了开销。

1. 简述一下InnoDB锁使用的算法。这些算法实现了什么

答：

Record lock：锁定单条记录，一般会针对唯一索引或主键进行加锁。

Gap lock：间隙锁，用于锁住一个范围，但是不包含数据本身。保证这个范围不能被其他数据修改

Next-Key lock：锁定一个范围，且包含数据本身。针对非唯一索引进行范围加锁，类似于记录锁和间隙锁的合体版。

默认环境下，隔离级别为可重复读，此情况下使用的算法是Next-Key lock，当索引为唯一索引或主键时，Next-Key lock会降级为Record lock，若是为一般索引，会先对索引本身加上锁（间隙锁Gap lock），然后对目标主键加上锁（记录锁Record lock），并且在下一个键值之间加上间隙锁，用来阻止幻读。间隙锁的范围是通过查询条件来确定的，如果采用范围查询的话，间隙锁就只会对该字段范围以内的索引（主键索引）加读锁，如果是等值查询，则会以其主键的范围进行加锁，所以主键是个很重要的东西，绝对不能丢弃。

这些算法的主要意义在于对MySQL是如何实现的行级锁进行了诠释，例如一般非唯一的普通索引，现在要通过这些字段操作数据，那么为了避免冲突，会进行对索引（索引本身也是数据，有点类似一个小规模的对象）的锁定，然后在进行操作，实现了一个针对不同索引进行不同的数据锁定模式。

这里要注意的是，MySql行级锁的实现并不会对记录加锁，加锁的对象时索引。当使用主键操作数据库时，会锁住主键索引。如果操作非聚簇索引时，先去锁住索引本身，再去锁定聚簇索引。当没有任何所有的情况，MySQL才会对所有的记录加锁。

1. 简述一下InnoDB引擎下，索引的存储结构。

答：在innoDB中，B+树类型的索引每一个叶子节点就是一个数据页，数据页中存放的的是数据页目录和数据组。而针对这些数据页的目录就是索引。下图展示了mysql一页数据中存放的内容。

图示

低可信度描述已自动生成

每页数据会提供自己最小的索引字段值（用于索引的排序）。此处只有叶子节点才会是此类型的数据页，而向上的节点则是只有页目录的形式，本身不存放用户数据。下图为B+树展示的内容（爷8会做动图，所以就将就一下），B+树的一个数据结构。

图形用户界面, 文本

描述已自动生成

这里特殊说明一下，只有主键索引的情况下，数据页中才会去保存数据的完整版，但是在联合索引和非主键索引的情况下，data数据集中存放的是主键，不会存放完整版，这也就是聚簇索引的来源。

1. 在innoDB下的Buffer Pool是什么，起到了什么作用。

答：Buffer Pool是mysql在内存中开辟的一个区域，每次查询的数据有可能不同，但是也有可能相同，所以mysql会将数据直接保存到自己开辟的一个buffer pool（缓冲池）中来减少对磁盘IO的操作，每次读取的是一个数据页，下次查询的时候就不需要去访问磁盘，而直接访问内存即可。Buffer Pool运行流程如下

图示, 示意图

描述已自动生成

Buffer Pool作为缓冲池为高频查询的数据进行缓存，为低频率使用的数据进行最近使用频率的算法进行淘汰，减少mysql直接接触磁盘从而减少大量的IO操作。

1. 为什么要是用redo.log，redo.log能够帮助mysql完成什么操作。

答：假设，在innoDB模式下，若产生宕机，在buffer pool中的数据将会丢失（为持久化的保存咋内存中），那么为了保证宕机后还能正确执行，innoDB中引入了redo.log和binlog。

图形用户界面, 应用程序

描述已自动生成

一个事务的操作流程如下：

* 修改buffer pool中的数据，形成脏数据页
* 操作行为生成redo.log日志
* 将redo.log进行持久化操作
* 返回操作成功

为什么要去写redo.log呢，因为数据存储在磁盘中是随机存储的，也就是我们需要去查找这些数据并修改是很消耗性能的，而redo.log是一个读写的一个log文件而已，读写性能相较于随机读写是要快也方便的。不仅如此，redo.log作为一个操作日志的情况，我可以为各种意外情况提供一定的容灾机制，当我宕机的情况下，可以通过redo.log进行还原。

1. 在执行更新语句时，innoDB下如何去更新索引。

答：在buffer pool中有一块特别的区域，这块区域叫做change buffer。Change buffer中保存的是数据更新所执行的sql，当使用到sql对应的索引的情况下，索引会被读取到内存（buffer pool）中，此时再去将索引中的数据重新更新掉，避免了无用的读取索引修改时还能减少update产生的工作量。change buffer也会定时的持久化到磁盘中防止宕机的时候丢失数据。

# Redis

# Zookeeper

# Netty与Http

# Spring framework/SpringBoot

## Spring Framework

1. Spring Framework有哪些模块组成，具有哪些核心功能。

答：Spring Framework由web，mvc，bean，aop……组成，主要是一个轻量级且易于维护的框架。Spring Framework有六大特征。

核心内容：IOC，AOP，事件，数据绑定，数据校验，SqEl等

测试：测试单元（Junit）

数据访问：DAO支持，JDBC，ORM框架等

Web支持：SpringMVC和Spring webflux web

集成：集成消息队列JMS，邮件，任务调度（QuartZ），远程处理

语言：支持Grovy，kotlin等语言

1. 说一下对Spring Framework的核心IOC和AOP的理解

答：

IOC：Spring提供了IOC作为对实例对象统一化管理，不在需要我们自己去实例化，转而由IOC容器保存，相当于提供一个工厂，我们只需要向提供配置清单（XML）或者标记自己的类使得编译器知晓该对象需要进行实例化。

AOP：顾名思义，面向切面编程，将执行方法纵向分割，不对方法本身产生影响而对方法执行的前、中、后等纵向切割面提供编写。对方法执行的不同分支做不同的处理。

1. 说明下载Spring的IOC容器中Bean的生命周期。

答：Spring Bean首先是进行个体实例化（通过反射），然后判断Bean本身是否满足相应的三段式条件BeanNameAware、BeanFactoryAware、ApplicationContextAware顺序由小到大，完成以上三段式以后，加载BeanPostProcessor做前置处理（可参考逸百年项目自定义注解扫描），然后就是相对常见的InitializingBean实现afterPropertiesSet方法，然后到init-method指向的初始化方法，然后又BeanPostProcessor完成处理，将配置好的Bean注册到容器中。最后销毁。

特殊说明一下，PostConstract执行在InitializingBean之前，大致的顺序如下

@PostConstruct > afterPropertiesSet > initMethod

1. Bean的作用域有哪些？

答：Bean的创建大致可以分为三步，首先，创建实体，其次，为实体填充内容并且注册到IOC容器当中，最后，调用配置的init-method初始化方法进行对象的初始化。那么我们创建实体时可分为无参构造创建和有参构造创建两种情况。

Bean的作用域分为5种

Singleton：作用域Spring的IOC容器中，让Bean作为单例存在，只要容器能触及到的范围，都能够使用。在缺省配置的状态下，默认使用Singleton模式。使用get/set模式进行初始化传值或使用有参构造创建实体。

Prototype：作用域与Singleton模式相同，但不同的是，prototype模式下，所有Bean都不是单例，只是在获取或注入Bean时创建一个，创建模式与singleton模式相同。

Request：属于web容器（也就是contextListener），只能作用在在webApplicationContext容器中，每次接收到请求的时候会创建一个Bean，同时也可以被回收。

Session：同Request一样，只能作用于webApplicationContext容器中，但不同的是，Session下的Bean能被所有的Request所共享，但是不同情况下的Session使用的Bean不能被共享

GlobalSession：同样，只能作用于webApplicationContext容器中，简而言之，就是提供一个能被所有Session所共享的Bean

1. 在spring中是如何解决bean的循环依赖问题？

答：关于如何解决循环依赖问题：

首先，先认识Spring IOC容器的三级缓存

SingletonFactories（三级），earlySingletonObjects（二级），SingletonObjects（一级）

Spring首先从一级缓存去获取Bean对象，未获取到且对象处于创建中的一个状态，则去二级缓存中获取，若二级缓存也未能获取到且允许使用三级缓存，那么则会去三级缓存中获取，在三级缓存获取到了以后，将三级缓存的Bean对象迁移至二级缓存。

循环依赖就是指代着A对象指向B对象，B对象同时指向A对象，那么在创建Bean的时候就陷入了循环，无法清楚知道应该优先创建那个对象。

分三种情况讨论：

Situation1：使用构造方法创建对象

无法满足上述情况。因为在一级缓存中创建A对象时就必须要创建B的对象，此时会 去二，三级缓存中查找，未能找到则去创建B对象，但是此时B对象创建还未能发现A对象，所以进入死循环。

Situation2：使用get/set方法创建对象

能够满足上述情况。由于get/set方法是在一级缓存中进行，而对象的实例早在二级缓 存就创建了，所以，此时B对象的创建能够在二级缓存中发现A对象，那么B对象创 建成功，A对象值填充完成

Situation3：使用prototype模式进行对象创建

不能满足上述情况，无论使用哪种构造方法。由于prototype是在注入的过程中才会去 进行Bean的实例，而不是像单例一样，在初始化阶段就完成了空构造参数的创建。所 以在注入时，A对象先通过无参构造创建实例，在进行值填充，此时都在一级缓存中执 行，当值填充到达需要引用B对象时，则会去创建B对象，同时发现任何缓存中没有 创建好的A对象，那么创建失败

## SpringBoot

1. SpringBoot常用的核心注解有哪些？

答：

@SpringBootApplication：用来标记启动类，同时还可以确定类的扫描范围

@SpringBootConfiguration：SpringBoot整合了@Configuration文件配置的功能。

@Profile：用于确认从哪个配置文件读取（注意，必须满足application-\*.yml之类的）

@ConfigurationProperties：可用于提取配置文件中某项相同配置的前缀

1. SpringBoot的配置文件\*.properties与\*.yml有什么区别？application.properties与bootstrap.properties有什么不同？

答：yml配置能够简洁明了的看出配置的结构和关联性，简化了properties文件的配置。application.properties：在ApplicationContext中进行加载，然后自动配置到对应的配置类中。

bootstrap.properties：该文件优先于application文件，一般用于上下文配置是进行初始化某些信息时使用，且bootstrap中的内容不能被覆盖。

1. SpringBoot工程中为什么要在pom管理文件中继承spring-boot-starter-parent或者是导入spring-boot-starter-dependency这些依赖？

答：SpringBoot是一个高度集成性的一个框架，通过继承或导入问题中的这些依赖来实现依赖版本插件版本的统一化，同时也导入了相关的自动化配置，例如有些Starter不需要我们进行配置，只需要导入就能够完成启动和使用。

1. SpringBoot为什么通过Jar文件就可以单独启动对外暴露接口并访问？

答：因为SpringBoot内部已经实现了内嵌式tomcat或者jboss之类的能够对外暴露服务的容器，使用和可以通过一些简单的配置对这些内嵌式的容器进行一些列的配置。一旦启动就会加载这些容器实现不需要我们再去将已经打好的war文件放到容器中。

1. SpringBoot中的监视器（Actuator）使用过吗？能实现什么功能？

答：SpringBoot同时依赖actuator-starter来监视服务是否正常，以及服务当前的一些相关参数，以及指标。

1. 简述一下SpringBoot的启动过程。

答：

# Spring Cloud Netflix

# Spring Cloud Alibaba

# Rabbit MQ/Kafka

# 设计模式