# JVM

## 1、JVM内存结构

根据JVM规范，JVM内存共分为堆、虚拟机栈、方法区、本地方法栈和程序计数器。 其中堆和方法区为线程共享，程序计数器、虚拟机栈和本地方法栈为线程私有。堆是对象分配内存的区域；方法区主要存放类信息、常量、静态变量、编译器编译后的代码等数据；程序计数器用来存储指向下一条指令的地址，即将要执行的指令代码；虚拟机栈是方法执行的内存区，每个方法执行时会在虚拟机栈中创建栈帧。

## 2、JVM内存模型（JMM）

JMM是一组抽象的概念并不真实存，JMM有三大特性：

（1）可见性

①线程解锁前必须把共享变量的值刷会主内存

②线程加锁前，必须读取主内存的最新值到自己的工作内存

③加锁解锁是同一把锁。由于JVM运行程序的实体是线程，而每个线程创建时JVM都会为其创建一个工作内存(或栈空间)，工作内存是每个线程的私有区域，而java内存模型规定所有的变量都存储在主内存，主内存是共享内存区域，所有线程都可以访问，但线程对变量的操作必须在工作内存中进行，首先要将变量从主内存拷贝到自己的工作内存，然后对变量进行操作，操作完成后再写会主内存，各个线程的工作内存中存储着主内存的变量副本拷贝，线程之间的传值必须依靠主内存来完成。

（2）原子性

某个线程正在做某个具体的业务时，中间不可以被分割，需要整体完整，要么同时成功，要么同时失败。

（3）有序性

计算机在执行程序时，为了提高性能，编译器和处理器尝尝会对指令进行重排序。一般分为编译器优化重排，指令并行的重排和内存系统的重排。处理器在进行指令重排时必须考虑指令之间的数据依赖性。在多线程环境中线程交替执行，由于指令重排，两个线程中使用的变量能否保证一致性是无法确定的。所以必须保证有序性。

## 3、双亲委派机制？

JDK9之前，双亲委派机制是如果一个类加载器收到了类加载的请求，它首先不会自己去尝试加载这个类，而是把这个请求委派给父类加载器去完成，每个层级的类加载器都是如此，因此所有的加载请求最终都应该委派到启动类加载器，只有当父类加载器反馈自己无法加载时，子类加载器才尝试进行加载。JDK9之后，在委派给父类加载器前，先判断该类是否能够归属于某一个系统模块中，如果可以找到这样的归属关系，就要优先委派给负责那个模块的加载器完成加载。作用就是为了保证每个类在程序中只加载一次并防止核心类被恶意篡改。

JDK9之前，类加载器包含启动类加载器、扩展类加载器、应用程序类加载器和自定义类加载器。在JDK9之后，由于模块化的引入，类加载器也发生了很大的变化。扩展类加载器被重命名为平台类加载器，平台类加载器和应用类加载器也不在继承自URLClassLoader，启动类加载器(BootClassLoader)、平台类加载器、应用程序类加载器全部继承于BuiltinClassLoader，启动类加载器也有了部分的java实现。

## 4、你觉得双亲委派这个名字有没有什么问题?

？？？？？？？

## 5、打破双亲委派的例子,举个例子?怎么打破?

最常见的打破双亲委派机制就是JAVA中涉及SPI(Service Provider Interface)的加载，如JDNI、JDBC等，由于实现类是第三方厂家，启动类加载器不可能对其进行加载，为了解决这个问题，java采用引入一个线程上下文类加载器加载的机制，通过Thread类的setContextClassLoader()方法进行设置，这是一种父类加载器去请求加载子类加载器完成类加载的行为，于是便打破了双亲委派机制。还有OSGI热部署和JDK1.1的自定义类加载器重写了loadClass();

## 6、你现在用什么 GC 收集器?

现在用的是JDK8，所以默认用的是Parallel GC。

## 7、Eden 区的对象回收策略?

Eden区主要采用复制算法，当Eden区内存不足时，会触发Minor GC进行垃圾回收，将存活的对象放入Survivor区或者符合条件直接到老年代。

## 8、CMS 的 STW 情况?

CMS垃圾回收器主要分为四个阶段：①初始标记 ②并发标记 ③重新标记④并发清除。

其中初始标记和重新标记仍然需要STW。

①初始标记：仅仅是标记GC ROOTS能直接关联到的对象，速度很快。

②并发标记：从GC ROOTS的直接关联对象开始遍历整个对象图的过程，虽然这个过程很长但并不需要停顿用户线程，可以与垃圾收集线程一起进行。

③重新标记：为了修正并发标记期间因用户程序继续运作而导致标记产生变动的那一部分对象的标记记录，在这个阶段的停顿时间稍微长一些。

④并发清除：清理删除标记阶段判断已死的对象，由于不需要移动存活对象，所以这个阶段也可以与用户线程同时并发的。

## 9、CMS 什么情况会触发 Full GC?

由于CMS无法处理浮动垃圾，所以在并发标记和并发清除阶段会有新的垃圾产生，当前的垃圾回收并不能回收这些新产生的垃圾，只能留到下一次垃圾收集时再清理，所以CMS收集器不能像其他收集器那样等到老年代完全被填满时再进行垃圾回收，必须预留一部分空间供并发收集时的程序运行使用。此时若CMS运行期间预留的内存无法满足程序分配新对象的需要，这时候虚拟机就会启动后备方案，采用Serial Old GC进行FULL GC。而且由于CMS采用标记清除算法，会导致内存碎片化而无法找到足够大的连续空间来分配给大对象，此时不得不提前触发FULL GC。

## 10、新对象一定会在新生代 new 出来么?

不一定，可能会存在栈上分配或者直接分配到老年代 -XX:PretenureSizeThreshold用于指定大于该设置值的对象直接老年代分配(此参数仅支持Serial和ParNew垃圾回收器)。

## 11、堆是分配对象存储的唯一选择吗？（栈上分配）

不是，如果经过逃逸分析(-XX:+DoEscapeAnalysis)后发现一个对象并没有逃逸出方法的话，那么就有可能被优化为栈上分配。这样就无需在堆上分配内存，也无需进行垃圾回收。还有基于OpenJDK定制的TaoBaoVM，其中创新的GCIH(GC Invisible heap)技术实现了Off-heap，将生命周期较长的java对象从堆中移到堆外，达到降低GC的回收频率和提升GC的回收效率的目的。

## 12、Java8有什么新的GC 算法?

Java8将方法区的PermGen更改为MetaSpace。JDK8u40之后G1提供了并发的类卸载支持，才被官方宣布为全功能的垃圾回收器。

## 13、G1垃圾回收器。

G1开创了收集器面向局部收集的设计思路和基于Region的内存布局形式。是JDK9默认的垃圾回收器。G1将java堆划分为多个大小相等的Region，每个Region都可以根据需要，扮演Eden区、Survivor区和老年代空间。并且存在一个特殊的Humongous区域，专门用来存储大对象（G1认为只要大小超过Region大小的一半即视为大对象），G1会跟踪每个Region的价值，然后在后台维护一个优先级队列，G1就会优先处理回收价值收益最大的Region。 G1的运作过程大概分为四个步骤：

①初始标记：仅仅只是标记一下GC ROOTS能直接关联到的对象，并修改TAMS(每个Region都有两个TAMS指针，在回收时TAMS指针以上的默认不回收)指针的值，让下一阶段用户线程并发运行时，能正确地在可用的Region中分配新对象，有STW。

②并发标记：从GC ROOTS开始对堆中对象进行可达性分析，找出回收的对象，耗时较长但可以并发执行，对象图扫描完成后，还需要重新处理SATB(原始快照)记录下的在并发时有引用变动的对象，无需STW。

③最终标记：用于处理并发标记阶段结束后仍遗留下的最后的STAB记录，需要STW。

④筛选回收：更新Region统计数据，对各Region的回收价值进行排序，根据用户选择的停顿时间进行计划，将多个Region组成回收集进行回收，需要STW。

## 14、哪些情况会出现OOM?

① java.lang.StackOverflowError：方法调用过多造成栈溢出

public static void main(String[] args) {  
 *stackOverFlow*();  
 }  
 private static void stackOverFlow() {  
 *stackOverFlow*();  
 }

② java.lang.OutOfMemoryError: Java heap space 堆内存溢出。可能大对象较多。

public static void main(String[] args) {  
 String str = "aaaaaaa";  
 while (true){  
 str += str +new Random().nextInt(111111)+new Random().nextInt(222);  
 str.intern();  
 }  
 }

③ java.lang.OutOfMemoryError: GC overhead limit exceeded

超过了98%的时间来做GC并且回收了不到2%的堆内存，连续多次GC都只回收了不到2%的极端情况下抛出。

public static void main(String[] args) {  
 int i = 0;  
 List<String> list = new ArrayList<>();  
 try {  
 while (true) {  
 list.add(String.*valueOf*(++i).intern());  
 }  
 } catch (Throwable e) {  
 System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*i=" + i);  
 e.printStackTrace();  
 throw e;  
 }  
}

④ java.lang.OutOfMemoryError: Direct buffer memory

参数配置：-Xms10m -Xmx10m -XX:+PrintGCDetails -XX:MaxDirectMemorySize=5m

导致原因：写NIO程序时经常使用ByteBuffer来读取或者写入数据，这是一种基于通道Channel与缓冲区Buffer的I/O方式，它可以使用Native函数库直接分配堆外内存，然后在通过一个存储在java堆里面的DirectByteBuffer对象作为这块内存的引用进行操作。这样能在一些场景中显著提高性能，避免在java堆和Native堆中来回复制数据。

ByteBuffer.allocate(capability)：分配JVM堆内存，属于GC管辖范围内，由于需要拷贝所以速度相对较慢。

ByteBuffer.allocateDirect(capability)：分配OS本地内存，不属于GC管辖范围，由于不需要内存拷贝所以速度相对较快。

但如果不断分配本地内存，堆内存很少使用，那么JVM就不需要执行GC，DirectByteBuffer对象就不会被回收，这时候堆内存充足，但是本地内存可能已经用光，再次分配本地内存就会导致OutOfMemoryError，程序便直接崩溃。

public static void main(String[] args) {  
 System.*out*.println("默认1/4");  
 System.*out*.println("配置的maxDirectMemory\t" + (sun.misc.VM.*maxDirectMemory*() / (double) 1024 / 1024) + "MB");  
 *//配置5M，但是分配6M* ByteBuffer byteBuffer = ByteBuffer.*allocateDirect*(6 \* 1024 \* 1024);  
}

⑤ java.lang.OutOfMemoryError: unable to create new native thread

在高并发请求服务器时，经常出现。

导致情况：应用创建了太多的线程，一个应用进程创建多个线程，超过系统承载极限；服务器并不能允许应用程序创建那么多的线程，linux系统默认允许单个进程可以创建的线程数为1024个，超过就会报unable to create new native thread

解决办法：想办法降低应用程序创建的线程数量，分析应用是否真的需要那么多线程；对于有的应用，确实需要创建很多线程，远超过linux系统的默认1024个限制，可以通过修改linux服务器配置，扩大linux默认限制。

public static void main(String[] args) {  
 for (int i = 0; ; i++) {  
 System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*i=" + i);  
 new Thread(()->{  
 try {  
 Thread.*sleep*(Integer.*MAX\_VALUE*);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 },""+i).start();  
 }  
}

⑥ java.lang.OutOfMemoryError: Metaspace

参数设置：-XX:MetaspaceSize=8m -XX:MaxMetaspaceSize=8m

Java8及以后采用Metaspace来代替永久代。

Metaspace是方法区在HotSpot中的实现，他与持久代最大的区别在于Metaspace使用的是本地内存而不是虚拟机内存，Metaspace存放了以下信息：虚拟机加载的类信息，常量池，静态变量，即时编译后的代码。

static class OOMTest{}  
public static void main(String[] args) {  
 while (true) {  
 Enhancer enhancer = new Enhancer();  
 enhancer.setSuperclass(OOMTest.class);  
 enhancer.setUseCache(false);  
 enhancer.setCallback((MethodInterceptor) (o, method, objects, methodProxy) -> methodProxy.invokeSuper(o,args));  
 enhancer.create();  
 }  
}

## 15、堆中内存分配比例?

默认情况下新生代占整个堆的1/3，老年代占堆的2/3，可以通过参数-XX:NewRatio来控制。在新生代中Eden区占8/10，两个Survivor区各占1/10，可以通过-XX:SurvivorRatio参数进行设置。

## 16、新生代 GC 工作流程?

首先新对象基本都会在Eden区的TLAB上进行分配，TLAB不足会在公共的Eden上分配，若还是内存不足会进行一次Minor GC，然后将Eden区和From区存活的对象放到To区，并进行年龄+1，当survivor中对象年龄达到了MaxTenuringThreshold的值对象或survivor中相同年龄所有的对象大小的总和等于survivor的一半时年龄大于或者等于该年龄的对象进入老年代。

## 17、大对象进入老年代,这个大对象是有多大?

大于参数-XX:PretenureSizeThreshold的设置值的对象直接老年代分配。

## 18、怎么排查OOM?

先通过内存映像工具对Dump出来的堆转储快照进行分析，如VisualVM、JProfiler，然后分清楚到底是出现了内存泄漏还是内存溢出。

如果是内存泄漏，可进一步通过工具查看泄漏对象到GC Roots的引用链。这样就能够找到泄漏的对象是通过怎么样的路径与GC Roots相关联的导致垃圾回收机制无法将其回收。掌握了泄漏对象的类信息和GC Roots引用链的信息，就可以比较准确地定位泄漏代码的位置。

如果不存在泄漏，那么就是内存中的对象确实必须存活着，那么此时就需要通过虚拟机的堆参数（-Xmx和-Xms）来适当调大参数；从代码上检查是否存在某些对象存活时间过长、持有时间过长的情况，尝试减少运行时内存的消耗。

## 19、根节点枚举

在HotSpot虚拟机中，当类加载完成时，hotSpot就会把对象内什么偏移量上是什么类型的数据计算出来，存到一组OopMap的数据结构中，这样收集器在扫描中就可以直接得到，并不需要真正的一个不漏的从方法区等GC ROOTS开始查找。

## 20、如何处理跨代引用？

垃圾收集器在新生代中建立了名为记忆集的数据结构，用以避免把整个老年代加进GC ROOTS的扫描范围。卡表是目前最常用的一种记忆集，其中定义了记忆集的记忆精度、与堆内存的映射关系。Hotspot中512字节代表一个卡页，数据变脏记则为1，表示存在跨代引用，需要加入GC ROOTS一并扫描。

## 21、如何维护卡表数据？

在hotspot中采用写屏障来完成卡表状态的更新，除了G1，其他垃圾收集器都采用的是写后屏障。为了处理伪共享问题，JDK7后hotspot还增加了-XX:+UseCondCardMark参数用来决定是否开启卡表更新的条件判断。即更新前先判断是否能更新。

## 22、如何保证回收的正确性？

三色标记：

①黑色：表示对象已经被垃圾回收器访问且所有的引用都已经扫描。

②灰色：表示对象已经被垃圾回收期访问但有引用还未被扫描。

③白色：表示对象尚未被垃圾回收器访问。

有两种解决方案：增量更新和原始快照。

增量更新是当一个黑色对象插入新的指向白色对象的引用时，就将这个新插入的引用记录下来，并发结束后再将这些记录过的引用关系中的黑色对象为根重新扫描一次。

原始快照是当灰色对象要删除指向白色对象的引用关系时，就将这个要删除的引用记录下来，并发结束后再将这些记录过的灰色对象为根重新扫描一次。CMS基于增量更新，而G1和Shenandoah则是采用原始快照来实现。

## 23、G1的优点和缺点？

G1最大的优点在于可以由用户指定期望的停顿时间，使得G1在不同的应用场景中取得吞吐量和延迟之间的平衡。并且在垃圾收集时不会产生内存碎片。缺点在于基于Region的设计导致G1的卡表实现更为复杂，并且每个Region都有一份卡表，占用了约20%甚至更多的堆内存。在CMS中的写屏障采用同步方式，而G1由于更为复杂则将其实现放在了类似于消息队列的结构中，再进行异步处理。

## 24、Shenandoah垃圾回收器

Shenandoah是第一款不由Oracle公司开发的垃圾回收器。由RedHat公司开发。Shenandoah也是基于Region的堆内存布局，有着Humongous Region，默认的回收策略也是优先回收价值最高的Region，但Shenandoah没有实现分代收集，支持并发的整理算法，摒弃了记忆集采用了连接矩阵的全局数据结构来纪律跨Region的引用关系，降低了维护记忆集的损耗和伪共享的发生概率。Shenandoah的工作进程大致分为9个阶段。

①初始标记：与G1一样，首先标记与GC ROOTS直接关联的对象，需要STW。

②并发标记：与G1一样，遍历对象图，标记出全部可达的对象

③最终标记：与G1一样，处理剩余SATB，并在这个阶段统计回收价值最高的Region，组成回收集，有一小段STW。

④并发清理：用于清理那些整个区域内一个存活对象都没有的Region。

⑤并发回收：把回收集里面的存活对象复制一份到其他未被使用的Region中，通过读屏障和转发指针来解决转移对象。

⑥初始引用更新：把旧对象的引用修正到复制后的新地址。会产生非常短暂的STW。

⑦并发引用更新：按照内存物理地址的顺序，线性的将引用类型的旧地址改为新地址。

⑧最终引用更新：修正存在GC ROOTS中的引用，需要STW。

⑨并发清理：此时回收集中无存活对象，直接清理。

## 25、Shenandoah的转发指针

之前在并发中移动对象通常是在原有的内存上设置保护陷阱，一旦访问旧对象就会进入预设好的异常处理机制在把访问转发到新对象上，但如果没有操作系统的支持，需要从用户态切换到内核态代价很高。Brooks提出了转发指针，在原有对象布局结构的最前面统一增加一个新的引用字段，正常情况下指向自己。并通过CAS来保证并发时对象的访问正确性，防止多线程下写入无效（复制副本，更新字段，收集器更新引用就会导致更新无效）。

## 26、ZGC

ZGC是一款基于Region内存布局的、暂时不设分代的，使用了读屏障、染色指针和内存多重映射等技术来实现可并发的标记整理算法，以低延迟为首要目的一款垃圾回收器。

与G1和Shenandoah不同的是，ZGC的Region具有动态性，比如动态创建和销毁，动态的区域容量。在x64平台下，ZGC的Region分为大中小三种容量。

小Region：固定2MB，存放小于256KB的小对象。

中Region：固定32MB，存放大于等于256KB小于4MB的对象。

大Region：不固定但必须为2MB的整数倍，用于存放4MB及以上的大对象，每个大Region只会存放一个大对象。大Region不会被重分配。

ZGC的工作进程大致可分为四个阶段：

①并发标记：与G1和Shenandoah一样，并发标记是遍历对象图做可达性分析阶段，但ZGC的标记在指针上而不是在对象上进行的，标记阶段会更新染色指针的Marked0、Marked1标志位。

②并发预备重分配：这个阶段需要根据特定的查询条件统计得出本次收集过程要清理那些Region，这些Region将组成重分配结果集。ZGC每次都会回收都会扫描所有的Region，用范围更大的扫描成本省去G1中记忆集的维护成本。

因此，重分配集只是决定了里面的存活对象会被重新复制到其他的Region，里面的Region会被释放。

③并发重分配：重分配是ZGC的核心阶段，这个过程要把重分配集中的存活对象分配到新的Region上，并为重分配集中的每个Region维护一个转发表，记录从旧对象到新对象的转换关系。基于染色指针的支持，ZGC收集器仅从引用上就明确得知一个对象是否处于重分配集中，如果用户线程此时并发访问了位于重分配集中的对象，这个访问就会被预置的内存屏障所截获，然后立即根据Region上的转发表记录将访问转发到新复制的对象上，并同时修正更新引用的值，使其直接指向对象。只要复制完就可以释放Region的内存，仅留下转发表就可以。

④并发重映射：重映射所做的就是修正整个堆中指向重分配集中旧对象的所有引用，但并不迫切，只是为了释放转发表的内存，所有此阶段合并到了下一次的垃圾回收的并发标记中完成，节省了遍历一遍对象图的开销，一旦所有所有指针都被修正之后，原来记录新旧对象关系的转发表就可以被释放。

## 27、ZGC的染色指针的三大优势

染色指针(地址由高到低Finalizable、Remapped、Marked1、Marked0)

64位操作系统 46 45 44 43

①染色指针可以使得一旦某个Region的存活对象被移走，这个Region立即就能够被释放掉和重用掉，而不必等待整个堆中所有指向该Region的引用都被修正后才清理。

②染色指针可以大幅减少在垃圾回收过程中内存屏障的使用数量，设置内存屏障，尤其是写屏障的目的通常是为了记录对象引用的变动情况，如果将这些直接维护在指针中，显然就可以省去一些专门的记录操作。

③染色指针可以作为一种可扩展的存储结构用来记录更多与对象标记、重定位过程相关数据，以便日后进一步提高性能。

注：为了染色指针的寻址，衍生除了多重映射技术来将多个不同的虚拟内存地址映射到用一个物理内存上。

# 数据库相关

## 1、JPA和MyBatis的区别?讲一下对象字段映射的细节？

JPA是一种标准，常见的实现由Hibernate和Spring data JPA

Mybatis是持久仓框架的一种

①JPA是对象和对象之间的映射，而Mybatis是对象和结果集的映射

②JPA的移植性较好，不用关心使用什么数据库，而Myabtis自己写sql语句，项目移植时需要重新编写sql

③当需要修改字段的时候mybatis修改起来比较费事，而JPA就相对简单

## 2、MySQL各种索引的一些差异？

MySql目前主要有FULLTEXT、HASH、BTREE、RTREE

①FULLTEXT：全文检索，目前仅支持MyISAM引擎支持。目前只有CHAR、VARCHAR、TEXT列上可以创建全文索引，FULLTEXT索引是按照分词原理建立索引的。

②HASH：为某一列或者几列建立HASH索引，就会利用这一列或者几列的值通过一定的算法计算出一个HASH值，对应一行或几行的数据。

③BTREE：BTREE索引就是一种将索引值按照一定的算法，存入一个树形的数据结构中。采用的是B+树

④RTREE：很少使用，仅支持geometry数据类型，较BTREE来说，RTREE的优势在于范围查找。

## 3、什么是聚簇索引?什么是非聚簇索引?

是否为聚簇索引实际上就是指的就是B+树的具体实现方式。聚簇索引的叶子结点就是数据结点，而非聚簇索引的叶子结点仍然是索引结点，只不过有指向对应数据块的指针。在mysql中InnoDB引擎是聚簇索引，而MyISAM是非聚簇索引。

## Innodb 是怎么保证崩溃恢复能力的?

一般情况下事务的提交涉及redo log 和 bin log。当commit命令执行时，首先会进入commit prepare阶段，这个阶段事务中新生成的redo log会被刷到磁盘，并将回滚状态设置为prepared状态。然后再进入commit阶段，InnoDB释放回滚锁，设置redo log提交状态，bin log持久化到磁盘，然后存储引擎提交。假如在事务的prepare阶段之前崩溃，此时的redo log和bin log都在内存中，此时本次事务的相关操作都会消失，不影响数据的一致性。假如在prepare阶段之后，commit阶段之前崩溃。此时redo log已经在磁盘中。bin log没有刷新到磁盘因此会消失。恢复时，读取磁盘中的redo log，但由于redo log还是prepare状态，就要判断bin log是否完整，若bin log完整则提交事务，若bin log不完整则回滚事务。若在commit阶段之后崩溃，此时redo log和bin log均在磁盘中，直接从redo log恢复。崩溃恢复的时候，会按顺序扫描 redo log：如果碰到既有 prepare、又有 commit 的 redo log，就直接提交；如果碰到只有 prepare、而没有 commit 的 redo log，就拿着 XID 去 bin log 找对应的事务。

## Innodb的优缺点?

**优点：**①InnoDB遵循ACID模式设计，支持事务，具有回滚和保护用户数据的崩溃恢复能力。

②InnoDB提供了行级锁，大大增加了多用户的并发性能。

③InnoDB采用聚簇索引的主键索引，减少了数据的查询次数。

**缺点：**①全文索引较慢、不支持FULLTEXT类型索引

②InnoD不保存表的具体行数，也就是说select conut(\*) from table时，要进行全表扫描

③delete table 时，InnoD不会重新建表，而是一行一行的删除

④InnoDB的行锁也不是绝对的，如果执行一个SQL时Mysql不知道要扫描的范围，InnoDB同样也会锁全表

## 6、explain 里面有哪些字段?

expain出来的信息有10列，分别是id、select\_type、table、type、possible\_keys、key、key\_len、ref、rows、Extra

①id：select查询序列号，id值越大优先级越高，越先被执行。

②select\_type：表示查询的类型。

(1) SIMPLE(简单SELECT，不使用UNION或子查询等)

(2) PRIMARY(子查询中最外层查询，查询中若包含任何复杂的子部分，最外层的select被标记为PRIMARY)

(3) UNION(UNION中的第二个或后面的SELECT语句)

(4) DEPENDENT UNION(UNION中的第二个或后面的SELECT语句，取决于外面的查询)

(5) UNION RESULT(UNION的结果，union语句中第二个select开始后面所有select)

(6) SUBQUERY(子查询中的第一个SELECT，结果不依赖于外部查询)

(7) DEPENDENT SUBQUERY(子查询中的第一个SELECT，依赖于外部查询)

(8) DERIVED(派生表的SELECT, FROM子句的子查询)

(9) UNCACHEABLE SUBQUERY(一个子查询的结果不能被缓存，必须重新评估外链接的第一行)

③table：输出的结果集的表，有时候可能是简称，不是真实的表

④type：表示MySQL在表中找到所需行的方式，又称“访问类型”。

ALL：Full Table Scan， MySQL将遍历全表以找到匹配的行

index: Full Index Scan，index与ALL区别为index类型只遍历索引树

range:只检索给定范围的行，使用一个索引来选择行

ref: 表示上述表的连接匹配条件，即哪些列或常量被用于查找索引列上的值

eq\_ref: 类似ref，区别就在使用的索引是唯一索引，对于每个索引键值，表中只有一条记录匹配，简单来说，就是多表连接中使用primary key或者 unique key作为关联条件

const、system: 当MySQL对查询某部分进行优化，并转换为一个常量时，使用这些类型访问。如将主键置于where列表中，MySQL就能将该查询转换为一个常量，system是const类型的特例，当查询的表只有一行的情况下，使用system

NULL: MySQL在优化过程中分解语句，执行时甚至不用访问表或索引，例如从一个索引列里选取最小值可以通过单独索引查找完成。

⑤possible\_keys：表示查询时，可能使用的索引

⑥Key：显示Mysql实际决定使用的索引，必然包含在possible\_keys

⑦key\_len：表示索引中使用的字节数

⑧ref：列与索引的比较，表示上述表的连接匹配条件

⑨rows：估算的结果集行数

⑩Extra：执行情况的描述和说明。

Using where:不用读取表中所有信息，仅通过索引就可以获取所需数据，这发生在对表的全部的请求列都是同一个索引的部分的时候，表示mysql服务器将在存储引擎检索行后再进行过滤

Using temporary：表示MySQL需要使用临时表来存储结果集，常见于排序和分组查询，常见 group by ; order by

Using filesort：当Query中包含 order by 操作，而且无法利用索引完成的排序操作称为“文件排序”

-- 测试Extra的filesort

explain select \* from emp order by name;

Using join buffer：改值强调了在获取连接条件时没有使用索引，并且需要连接缓冲区来存储中间结果。如果出现了这个值，那应该注意，根据查询的具体情况可能需要添加索引来改进能。

Impossible where：这个值强调了where语句会导致没有符合条件的行（通过收集统计信息不可能存在结果）。

Select tables optimized away：这个值意味着仅通过使用索引，优化器可能仅从聚合函数结果中返回一行

No tables used：Query语句中使用from dual 或不含任何from子句

## 7、mysql 一个 SQL 查询很慢,从哪些地方进行优化?

①查看字段是否建立了索引，假如没有，考虑建个索引。

①查看是否使用了select \* from table，如果有用具体字段代替，不要返回用不到的字段。

②应尽量避免全表扫描，首先应考虑在 where和order by相关的列上建立索引

③避免在 where 子句中对字段进行 null 值判断，否则将导致引擎放弃使用索引而进行全表扫描。

④索引字段上不要使用不等，索引字段上使用（！= 或者 < >）判断时，会导致索引失效而转向全表扫描。

⑤尽量避免在 where 子句中使用 or 来连接条件，否则将导致引擎放弃使用索引而进行全表扫描。

例如： select \* from t where num=10 or num=20

我们可以这样查询：select \* from t where num=10 union all select \* from t where num=20

⑥尽量避免在 where 子句中对字段进行表达式操作，这将导致引擎放弃使用索引而进行全表扫描。

## 8、自增 ID 和 UUID 的区别?

自增ID的插入数据模式，符合递增插入的场景。每次插入一条新记录，都是追加操作，都不涉及到挪动其他记录，也不会触发叶子节点的分裂，不会导致数据页的分裂，UUID在添加和删除时，都有可能导致页分裂和页合并。从性能和存储空间方面考量，自增ID往往是更合理的选择。

## 9、主键索引和普通索引的查询有什么区别？

如果语句是select \* from T where ID=500，即主键查询方式，则只需要搜索ID这棵B+树；

如果语句是select \* from T where k=5，即普通索引查询方式，则需要先搜索k索引树，得到ID的值为500，再到ID索引树搜索一次。这个过程称为回表。

也就是说，基于非主键索引的查询需要多扫描一棵索引树。因此，我们在应用中应该尽量使用主键查询。

## 10、自增 ID 申请完了会发生什么事情?

MySQL的自增主键ID是有限制的，它的最大值是2^31-1=4294967295。当自增主键达到最大，再次插入后，使用的自增主键还是4294967295，然后会报主键冲突的错误。

## 分库解决了什么问题?分表解决了什么问题?

①业务拆分，将一个大的业务拆分成多个小服务，服务之间通过RPC调用，不同服务的数据使用独立的数据库存储。

②为了应对高并发，读多写少，增加从库应对读流量，读少写多，分库分表，均衡写流量

③数据隔离，C端数据和B端数据分隔，核心数据与非核心数据分隔，使其不要相互影响

## 12、最左前缀原则

最左前缀原则是在联合索引中以最左边的为起点任何连续的索引都能匹配上，如创建(a,b,c)联合索引时，相当于创建了(a)单列索引，(a,b)联合索引以及(a,b,c)联合索引。

# Java基础

## 1、什么是RPC？

RPC 提供了一种远程通信机制，通过这一机制，在一台计算机上运行的程序可以顺畅地执行某个远程系统上的代码。

## 2、RPC 解决了现在的哪些问题？

解决无法在一个进程内，甚至一个计算机内通过本地调用的方式完成的需求，比如不同的系统间的通讯，甚至不同的组织间的通讯。由于计算能力需要横向扩展，需要在多台机器组成的集群上部署应用。

## 3、请谈谈你对volatile的理解

volatile是java虚拟机提供的轻量级的同步机制，基本上遵守了JMM的特性：保证可见性和禁止指令重排，但不保证原子性。

## 4、如何保证原子性？

①加Synchronized

②直接使用JUC下的AtomicInteger（不会出现写覆盖）

## 5、volatile如何保证有序性？

内存屏障：是一个CPU的指令，作用是保证特定操作的执行顺序和保证某些变量的内存可见性(强制刷新各种CPU的缓存数据)。

## 6、你在哪些地方用到过volatile？

单例模式下的DCL(Double Check lock)

public class Singleton {  
 private static volatile Singleton *instance* = null;  
 private Singleton() {  
 }  
 public static Singleton getInstance() {  
 if (*instance* == null) {  
 synchronized (Singleton.class) {  
 if (*instance* == null) {  
 *instance* = new Singleton();  
 }  
 }  
 }  
 return *instance*;  
 }  
}

## 7、什么是CAS?(讲一讲AtomicInteger为什么用CAS而不是synchronized)

CAS的全称是CompareAndSwap，它是一条CPU并发原语，他的功能是判断内存某个位置是否为预期值，如果是则更改为新的值，这个过程是原子的。CAS能保证原子性靠的是底层的Unsafe类，来自于JVM的rt.jar包内，由于JAVA无法直接访问底层系统，需要通过本地方法来访问，而基于Unsafe类就可以直接操作特定内存的数据。Unsafe类存在于sun.misc包下，其内部方法可以像C的指针一样采用offset直接操作内存。源码内部采用自旋锁进行实现。

var1：当前对象 var2：地址偏移量offset var4：update的值

public final int getAndAddInt(Object var1, long var2, int var4) {  
 int var5;  
 do {  
 var5 = this.getIntVolatile(var1, var2);  
 } while(!this.compareAndSwapInt(var1, var2, var5, var5 + var4));  
 return var5;  
}

compareAndSwapInt方法底层为汇编语言

## 8、CAS的缺点是什么？

①采用了自旋锁导致循环时间长开销大，长时间不成功CPU的开销会很大

②只能保证一个共享变量的原子操作

③存在ABA问题

## 9、原子类AtomicInteger的ABA问题是什么？

CAS算法实现一个重要的前提是需要取出内存中某时刻的数据并在当下比较并替换，那么在这个时间差内可能会导致数据的问题。比如一个线程一从内存中取出A，这个时候线程二也从内存中取出A，并且线程二进行操作将值变为B，然后又将值变为A，这时候线程一进行CAS的时候发现内存中仍然是A，然后线程一操作成功。但尽管线程一的CAS是成功的，但是不代表这个过程没有问题。

## 10、什么是原子更新？

为了解决其他的类也可以为原子类，java引出了AtomicReference<T>解决包装类的原子性问题。

## 11、如何解决ABA问题？

采用AtomicStampedReference，带时间戳的原子引用，每次CAS都会让stamp+1。

AtomicStampedReference<Integer> atomicInt = new AtomicStampedReference<>(100,1);

## 12、ArrayList是线程不安全的吗？如果是请写一个案例并给出解决方案

ArrayList底层是数组，默认初始值为10，添加时容量不够扩容为1.5倍，JDK8采用oldCapacity>>1。

由于ArrayList的方法没有加synchronized所以线程不安全。

Case：会报java.util.ConcurrentModificationExceptionList<String> list = new ArrayList<>();  
for (int i = 1; i <= 30; i++) {  
 new Thread(() -> {  
 list.add(UUID.randomUUID().toString().substring(0,8));  
 }, String.valueOf(i)).start();  
}

如何解决？①使用Vector<>();

②使用Collections.synchronizedList(new ArrayList<>());

③使用JUC包下的CopyOnWriteArrayList<>();

写时复制,读写分离的思想：CopyOnWriteArraylist往容器中添加元素的时候，不直接往Object[]中添加，而是先将Object[]进行Copy。复制一个新的容器Object[] newElements，然后在新的容器中添加元素，添加完元素后，再将原容器的引用指向新的容器(setArray(newElements))，这样做可以保证对CopyOnWirte容器进行并发的读，而不需要加锁。

CopyOnWriteArrayList的add方法

public boolean add(E e) {  
 final ReentrantLock lock = this.lock;  
 lock.lock();  
 try {  
 Object[] elements = getArray();  
 int len = elements.length;  
 Object[] newElements = Arrays.copyOf(elements, len + 1);  
 newElements[len] = e;  
 setArray(newElements);  
 return true;  
 } finally {  
 lock.unlock();  
 }  
}

注：HashMap和HashSet也是线程不安全的，HashSet的底层为HashMap

。JUC下有ConcurrentHashMap。创建HashSet时，会创建HashMap，初始容量16，负载因子0.75。value默认为一个Object对象

## 13、都有哪些锁？

①公平锁和非公平锁

公平锁：多个线程申请按照申请的顺序获取锁

非公平锁：多个线程获取锁不按照顺序，可以进行争抢，在多线程下有可能造成优先级翻转或饿死现象。再ReentrantLock中指定false或true，默认false，非公平锁。直接占有，占有失败再采用类似公平锁

非公平锁优点在于吞吐量比非公平的大

②可重入锁(递归锁)

指的是同一个线程外层函数获得锁之后，内层递归函数仍然能够获取该锁的代码，在同一个线程外层方法获取锁的时候，在进入内层方法会自动获取锁，也就是说线程可以进入任何一个它已经拥有的锁所同步的代码块。

③自旋锁

指尝试获取锁的线程不会立即阻塞，而是采用循环的方式尝试获取锁，这样的好处是减少线程上下文切换的消耗，缺点是循环会消耗CPU

④独占锁(写锁)/共享锁(读锁)/互斥锁

独占锁：指该锁一次只能被一个线程锁持有。 ReentrantLock的写锁

共享锁：指该锁可以被多个线程持有。 ReentrantLock的读锁

## 14、CountDownLatch、CyclicBarrier、Semaphore有没有使用过？

①CountDownLatch：具有计数器功能，等待其他线程执行完毕，主线程再继续执行。主要包括两个方法：countDown()计数器减1，await()等到计数器归0后唤醒等待线程

public static void main(String[] args) throws InterruptedException {  
 CountDownLatch countDownLatch = new CountDownLatch(6);  
 for (int i = 0; i <= 6; i++) {  
 new Thread(() -> {  
 System.*out*.println(Thread.*currentThread*().getName() + "走人");  
 countDownLatch.countDown();  
 }, String.*valueOf*(i)).start();  
 }  
 countDownLatch.await();  
 System.*out*.println("关门");  
}

②CyclicBarrier：循环屏障，让一组线程到达一个屏障时(同步点)被阻塞，直到最后一个线程到达屏障时，屏障才会开启，所有被屏障阻塞的线程才会继续干活，线程通过CyclicBarrier的await()方法。

public static void main(String[] args) {  
 CyclicBarrier cyclicBarrier = new CyclicBarrier(7, () -> {  
 System.*out*.println("人到齐，开会");  
 });  
 for (int i = 0; i < 7; i++) {  
 int curI = i;  
 new Thread(() -> {  
 System.out.println(curI + "到齐");  
 try {  
 cyclicBarrier.await();  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
 }, String.valueOf(i)).start();  
 }  
}

③Semaphore：信号量主要有两个目的，一个是用于多个共享资源的互斥使用，另一个用于并发线程数的控制。

public static void main(String[] args) {  
 *//模拟三个停车位* Semaphore semaphore = new Semaphore(3);  
 *//模拟六台车* for (int i = 1; i <= 6; i++) {  
 new Thread(() -> {  
 try {  
 semaphore.acquire();  
 System.out.println(Thread.currentThread().getName() + "抢到车位");  
 *//假设停三秒* TimeUnit.SECONDS.sleep(3);  
 System.out.println(Thread.currentThread().getName() + "离开车位");  
 } catch (Exception e) {  
 e.printStackTrace();  
 } finally {  
 semaphore.release();  
 }  
 }, String.valueOf(i)).start();  
 }  
}

## 15、什么是阻塞队列？你知道的阻塞队列都有那些？

阻塞队列是一个在队列基础上又支持阻塞插入和阻塞移出的两个方法。我们不需要关心什么时候需要阻塞，什么时候唤醒线程。典型的应用生产者-消费者模式、线程池、消息中间件。阻塞队列有以下几个种类：

①ArrayBlockingQueue：基于数组机构的有界阻塞队列，此队列按照FIFO原则对元素进行排序。

②LinkedBlockingQueue：由链表结构组成的有界阻队列(大小为Integer.MAX\_VALUE)

③SynchronousQueue：不存储元素的阻塞队列，也即单个元素的队列，每一个put操作都必须等待一个take操作，否则不能继续添加元素，反之亦然。

public static void main(String[] args) {  
 *//默认非公平锁* BlockingQueue<String> queue = new SynchronousQueue<>();  
 new Thread(() -> {  
 try {  
 System.out.println(Thread.currentThread().getName() + "\tput 1");  
 queue.put("1");  
 System.out.println(Thread.currentThread().getName() + "\tput 2");  
 queue.put("2");  
 System.out.println(Thread.currentThread().getName() + "\tput 3");  
 queue.put("3");  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }, "AA").start();  
  
 new Thread(() -> {  
 try {  
 TimeUnit.SECONDS.sleep(3);  
 System.out.println(Thread.currentThread().getName() + "\t" + queue.take());  
 TimeUnit.SECONDS.sleep(3);  
 System.out.println(Thread.currentThread().getName() + "\t" + queue.take());  
 TimeUnit.SECONDS.sleep(3);  
 System.out.println(Thread.currentThread().getName() + "\t" + queue.take());  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
 }, "BB").start();  
}

## 16、请手写一个生产者消费者模式

一个初始值为0的变量，两个线程对其进行操作，一个加1一个减1

①传统版(synchronized+wait+notify)

②升级版(lock+await+signalAll)

public class ProConsumer {  
 public static void main(String[] args) {  
 ShareData shareData = new ShareData();  
 new Thread(() -> {  
 for (int i = 0; i < 5; i++) {  
 try {  
 shareData.increment();  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
 }  
 },"AAA").start();  
  
 new Thread(() -> {  
 for (int i = 0; i < 5; i++) {  
 try {  
 shareData.decrement();  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
 }  
 },"BBB").start();  
 }  
}  
  
class ShareData {  
 private int number = 0;  
 private Lock lock = new ReentrantLock();  
 private Condition condition = lock.newCondition();  
  
 public void increment() throws Exception {  
 lock.lock();  
 try {  
 *//判断* while (number != 0) {  
 *//等待 不能生产* condition.await();  
 }  
 *//干活* number++;  
 System.*out*.println(Thread.*currentThread*().getName() + "\t" + number);  
 *//通知唤醒* condition.signalAll();  
 } catch (Exception e) {  
 e.printStackTrace();  
 } finally {  
 lock.unlock();  
 }  
 }  
 public void decrement() throws Exception {  
 lock.lock();  
 try {  
 *//判断* while (number == 0) {  
 *//等待 不能生产* condition.await();  
 }  
 *//干活* number--;  
 System.*out*.println(Thread.*currentThread*().getName() + "\t" + number);  
 *//通知唤醒* condition.signalAll();  
 } catch (Exception e) {  
 e.printStackTrace();  
 } finally {  
 lock.unlock();  
 }  
 }  
}

③进阶版(BlockQueue)

public class ProConsumer {  
 public static void main(String[] args) throws Exception {  
 MyResource myResource = new MyResource(new ArrayBlockingQueue<>(10));  
 new Thread(() -> {  
 System.*out*.println(Thread.*currentThread*().getName() + "\t生产线程启动");  
 try {  
 myResource.myProduct();  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
 }, "Product").start();  
 new Thread(() -> {  
 System.*out*.println(Thread.*currentThread*().getName() + "\t消费线程启动");  
 try {  
 myResource.myConsumer();  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
 }, "Consumer").start();  
  
 TimeUnit.*SECONDS*.sleep(5);  
 System.*out*.println("生产五秒结束，停止生产");  
 myResource.setFLAG(false);  
 }  
}  
  
class MyResource {  
 private volatile boolean FLAG = true;  
 private AtomicInteger atomicInteger = new AtomicInteger();  
 private BlockingQueue<String> blockingQueue;  
  
 public MyResource(BlockingQueue<String> blockingQueue) {  
  
 this.blockingQueue = blockingQueue;  
 System.*out*.println(blockingQueue.getClass().getName());  
 }  
  
 public void myProduct() throws Exception {  
 String data;  
 boolean returnVal;  
 while (FLAG) {  
 data = atomicInteger.incrementAndGet() + "";  
 returnVal = blockingQueue.offer(data, 2L, TimeUnit.*SECONDS*);  
 if (returnVal) {  
 System.*out*.println(Thread.*currentThread*().getName() + "\t插入队列" + data + "成功");  
 } else {  
 System.*out*.println(Thread.*currentThread*().getName() + "\t插入队列" + data + "失败");  
 }  
 TimeUnit.*SECONDS*.sleep(1);  
 }  
 System.*out*.println(Thread.*currentThread*().getName() + "\t停止生产");  
 }  
  
 public void myConsumer() throws Exception {  
 String result;  
 while (FLAG) {  
 result = blockingQueue.poll(2L, TimeUnit.*SECONDS*);  
 if (result == null || result.equalsIgnoreCase("")) {  
 FLAG = false;  
 System.*out*.println(Thread.*currentThread*().getName() + "\t超时，消费退出");  
 System.*out*.println();  
 return;  
 }  
 System.*out*.println(Thread.*currentThread*().getName() + "\t消费队列" + result + "成功");  
 }  
 }  
  
 public void setFLAG(boolean FLAG) {  
 this.FLAG = FLAG;  
 }  
}

## 17、synchronized和lock有什么区别？Lock有什么好处？

①synchronized是关键字，属于JVM层面，底层使用的是monitor对象来完成(monitorenter、monitorexit指令，每个锁对象都有拥有一个锁计数器和一个指向持有该锁的线程的指针，当执行monitorenter时，若计数器为0则没有被其他线程所持有，JVM就会将该锁对象的持有线程设置为当前线程，并且将计数器+1；遇到monitorexit-1，0则代表释放)，而Lock为具体的类，是API层面的锁。

②synchronized不需要用户去手动释放锁，当synchronized代码执行完后系统会自动让线程释放对锁的占用。ReentrantLock则需要用户去手动释放，若没有释放则有可能出现死锁现象。

③synchronized不可以被中断，除非抛出异常或者正常运行完成。ReentrantLock可以中断。

④synchronized为非公平锁，ReentrantLock两者都可，默认非公平锁

⑤synchronized不能绑定条件，只能随机唤醒一个或者全部，ReentrantLock可以绑定条件，做到精确唤醒。

例：Lock的条件唤醒A打印5次、B打印10次、C打印15次，来10轮

public class ReentrantLockCondition {  
 public static void main(String[] args) {  
 MyResource myResource = new MyResource();  
 new Thread(() -> {  
 for (int i = 0; i < 10; i++) {  
 myResource.print5();  
 }  
 }, "AAA").start();  
 new Thread(() -> {  
 for (int i = 0; i < 10; i++) {  
 myResource.print10();  
 }  
 }, "BBB").start();  
 new Thread(() -> {  
 for (int i = 0; i < 10; i++) {  
 myResource.print15();  
 }  
 }, "CCC").start();

}  
}  
  
class MyResource {  
 *//A:1 B:2 C:3* private int number = 1;  
 private Lock lock = new ReentrantLock();  
 private Condition c1 = lock.newCondition();  
 private Condition c2 = lock.newCondition();  
 private Condition c3 = lock.newCondition();  
 public void print5() {  
 lock.lock();  
 try {  
 *//判断* while (number != 1) {  
 c1.await();  
 }  
 for (int i = 0; i < 5; i++) {  
 System.*out*.println(Thread.*currentThread*().getName() + "\t" + i);  
 }  
 number = 2;  
 c2.signal();  
 } catch (Exception e) {  
 e.printStackTrace();  
 } finally {  
 lock.unlock();  
 }  
 }  
 public void print10() {  
 lock.lock();  
 try {  
 *//判断* while (number != 2) {  
 c2.await();  
 }  
 for (int i = 0; i < 10; i++) {  
 System.*out*.println(Thread.*currentThread*().getName() + "\t" + i);  
 }  
 number = 3;  
 c3.signal();  
 } catch (Exception e) {  
 e.printStackTrace();  
 } finally {  
 lock.unlock();  
 }  
 }  
 public void print15() {  
 lock.lock();  
 try {  
 *//判断* while (number != 3) {  
 c3.await();  
 }  
 for (int i = 0; i < 10; i++) {  
 System.*out*.println(Thread.*currentThread*().getName() + "\t" + i);  
 }  
 number = 1;  
 c1.signal();  
 } catch (Exception e) {  
 e.printStackTrace();  
 } finally {  
 lock.unlock();  
 }  
 }  
}

## 18、创建线程的四种方式？

①继承Thread类

class MyThread extends Thread{  
 @Override  
 public void run() {  
   
 }  
 }

②实现Runnable接口

class MyThread implements Runnable {  
 @Override  
 public void run() {  
  
 }  
 }

③继承Callable接口

class MyThread implements Callable<Integer>{  
 @Override  
 public Integer call() throws Exception {  
 System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*come in Callable");  
 return 1024;  
 }  
 }

public class CallableDemo {  
 public static void main(String[] args) throws Exception {  
 FutureTask<Integer> futureTask = new FutureTask<>(new MyThread());  
 Thread t1 = new Thread(futureTask,"AA");  
 t1.start();  
 System.*out*.println(futureTask.get());  
 }  
 }

④使用线程池

Executors.newFixedThreadPool(5);//固定大小线程池  
 Executors.newSingleThreadExecutor();//单个大小线程池  
 Executors.newCachedThreadPool();//可伸缩大小的线程池

## 19、Runnable和Callable的区别？

①Runnable无返回值，Callable有返回值

②Runnable不抛异常，Callable抛异常

## 20、为什么要用线程池，线程池的优势是什么？

线程池的主要工作控制运行的线程的数量，处理过程中将任务放到队列，然后在线程创建后启动这些任务，如果线程数量超过了最大数量，超出数量的线程排队等候，等待其他线程执行完毕，再从队列中取出任务来执行。

主要特点：线程复用；控制最大并发数；管理线程

主要优势：①降低资源消耗。通过复用已创建的线程降低线程创建和销毁造成的消耗。②提高响应速度。当任务到达时，任务可以不需要等到线程创建就立即执行。③提高线程的可管理性。线程是稀缺资源，如果无限制的创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一的分配，调优和监控。

## 21、线程池的几个重要参数？

线程池一共有七个核心参数。

①corePoolSize：线程池中的常驻核心线程数。

②maximumPoolSize：线程池能够容纳同时执行的最大线程数，此值必须大于1。

③keepAliveTime：多余的空闲线程的存活时间。当线程池的数量超过corePoolSize时，当空闲时间达到keepAliveTime值时，多余空闲线程会被销毁直到剩下corePoolSize个线程为止

④unit：keepAliveTime的单位

⑤workQueue（是一个BlockingQueue）：任务队列，存储被提交但尚未被执行的任务。当线程数目达到corePoolSize后，就会把到达的任务放到缓存队列中。

⑥threadFactory：表示生成线程池中工作线程的线程工厂，用于创建线程。一般默认即可。

⑦handler：拒绝策略，表示当队列满了并且工作线程大于等于线程池的maximumPoolSize时拒绝。

## 22、线程池的工作原理？

①创建线程池后，等待提交的任务。

②若正在运行的线程数小于corePoolSize，直接执行。

②若正在运行的线程数大于corePoolSize并且workQueue未满则放入队列等待执行。

③若队列已满但小于maximumPoolSize，创建非核心线程执行任务。

④若队列已满并且线程数等于maximumPoolSize，则会使用拒绝策略来执行

## 23、线程池的拒绝策略？

线程池一共有4种拒绝策略：

①AbortPolicy：默认的拒绝策略，直接抛出RejectedExecutionException.

②CallerRunsPolicy：该策略不会抛弃任务，也不会抛出异常，而是将任务退回到调用者，从而降低新任务的流量。

③DiscardOldestPolicy：抛弃队列中等待最久的任务，然后把当前任务加入队列中并尝试再次提交任务。

④DiscardPolicy：直接丢弃任务，不处理也不抛异常。

## 24、工作中如何使用线程池？

阿里手册中指出：线程池不允许使用Executors去创建，而是通过ThreadPoolExecutor的方式。

使用Executors返回的线程池对象的弊端：

①FixedThreadPool和SingleThreadPool由于允许的请求队列长度为Integer.MAX\_VALUE，有可能会堆积大量的请求，从而导致OOM。

②CachedThreadPool和ScheduledThreadPool由于允许的创建的线程数为Integer.MAX\_VALUE，有可能会创建大量的线程，从而导致OOM。

## 25、如何配置线程数最大数量？

看业务时CPU密集型还是IO密集型，CPU密集型是任务需要大量的运算，而没有阻塞，CPU一直全速运行；IO密集型并不是一直执行任务，则应尽可能多的配置线程。首先查询服务器的核数。

①CPU密集型：一般为CPU核数+1的线程池。

②IO密集型：CPU核数\*2或 CPU核数/(1-阻塞系数) 阻塞系数0.8-0.9

## 26、字符串常量池的知识。

public static void main(String[] args) {  
 String str1 = new StringBuilder("58").append("tongcheng").toString();  
 System.*out*.println(str1);  
 System.*out*.println(str1.intern());  
 System.*out*.println(str1.intern() == str1);  
 System.*out*.println();  
 String str2 = new StringBuilder("ja").append("va").toString();  
 System.*out*.println(str2);  
 System.*out*.println(str2.intern());  
 System.*out*.println(str2.intern() == str1);  
}

答案是什么？

58tongcheng java

58tongcheng java

true false jdk6之前 是两个false

"java"字符串在加载sun.misc.Version时进入了常量池，所以新new和在字符串常量池的不是一个。

Intern()方法：Intern是一个本地方法，它的作用是如果字符串常量池中已经包含一个等于此String对象的字符串，则返回代表池中这个字符串的String对象引用；否则，会将此String对象包含的字符串添加到常量池中，并返回此String对象的引用。（JDK6之前会首次遇到的字符串复制到永久代的字符串常量池，返回的也是永久代的字符串实例的引用，StringBuilder创建的字符串实例在java堆上，所以不是同一个引用）

## 27、LockSupport是什么？

LockSupport是用于创建锁和其他同步类的基本线程阻塞原语。 该类与使用它的每一个线程关联一个许可证，如果许可证可用则立即返回park，并在此过程消费；否则可能会阻止；如果尚未得到许可，则致电unpark获得许可。与Semaphore不同，许可证不会累积，最多只有一个。线程等待唤醒的加强版

其中park()和unpark()方法的作用分别是阻塞线程和解除阻塞线程。使用synchronized和lock时，线程先要获得并持有锁，必须在锁块(synchronized或lock)中，并且必须要先等待后唤醒，线程才能够被唤醒。而LockSupport则不需要。

public static void park() {  
 *UNSAFE*.park(false, 0L);  
}

public static void unpark(Thread thread) {  
 if (thread != null){

*UNSAFE*.unpark(thread);

}   
}

例：LockSupport的等待和唤醒

public static void main(String[] args) {  
 Thread t1 = new Thread(() -> {  
 System.out.println(Thread.currentThread().getName() + "\t----------->come in");  
 LockSupport.park();  
 System.out.println(Thread.currentThread().getName() + "\t----------->被唤醒");  
 }, "AAA");  
 t1.start();  
 try {  
 TimeUnit.*SECONDS*.sleep(2);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 Thread t2 = new Thread(() -> {  
 System.out.println(Thread.currentThread().getName() + "\t----------->come in");  
 LockSupport.unpark(t1);  
 }, "BBB");  
 t2.start();  
}

## 28、ReentrantLock的实现原理？(AQS)

ReentrantLock是基于AQS(AbstractQueueSynchronizer，抽象队列同步器)实现的，内部有一个抽象的Sync类继承了AQS。AQS是用来构建锁或者其他同步器组件的重量级基础框架以及整个JUC体系的基石，它使用一个volatile的int类型的成员变量来表示同步状态，通过内置的FIFO队列来完成资源获取的排队工作，将每条要去抢占资源的线程封装成一个Node节点来实现锁的分配，通过CAS完成对State值的修改。

锁和同步器的关系：锁是面向锁的使用者，同步器是面向锁的实现者。

加锁必定产生阻塞，有阻塞就得排队，排队必然要有某种形式的队列来进行管理。

## 29、AQS(AbstractQueuedSynchronizer)

在AQS中，有一个内部类Node，用来封装线程。Node结点中包含了next指针、pre指针、Thread线程、共享(SHARED)还是独占(EXCLUSIVE)和当前线程的waitStatus。waitStatus默认值为0。其中waitStatus有四种状态：

①CANCELLED：为1，表示获取锁的状态已经取消。

②SIGNAL：为-1，表示线程已经准备就绪，等资源释放。

③CONDITION：为-2，表示结点在等待队列中等待唤醒。

④PROPAGATE：为-3，表示在Share状态下才能被使用。

在Lock接口的实现类中，基本都是通过聚合一个AQS的子类(Sync)来完成线程的访问控制。而Sync类有NonfairSync和FairSync两个子类代表非公平锁和公平锁。公平锁和非公平锁的区别在于获取同步状态时(tryAcquire)会多加一个hasQueuePreDecessors()限制条件，hasQueuePreDecessors是公平锁加锁时判断队列中是否存在有效结点的方法。

## 30、AQS的具体执行流程？

以ReentrantLock的加锁解锁为例：

(1)当调用lock方法时会调用内部类sync的lock方法，具体实现为NonFairSync和FairSync的lock方法，NonFairSync的lock方法会先尝试采用CAS获取锁，若失败调用acquire方法，FairSync的lock则直接调用acquire。

(2)在acquire方法中有三个方法：tryAcquire、addWaiter和acquireQueued。

①首先会执行tryAcquire方法尝试进行获取锁，若失败，则调用addWaiter方法将线程加入队列中。

②addWaiter方法将当前线程封装成一个Node添加的队列中，AQS的队列时一个双向链表。若第一次添加则会先初始化队列，向队列中添加一个虚拟头节点或哨兵结点。然后将当前结点入队。

③acquiredQueued方法会再次调用tryAcquire方法尝试获取锁，若失败则调用shouldParkAfterFailedAcquire和parkAndCheckInterrupt()方法。在shouldParkAfterFailedAcquire会将虚拟头节点的waitStatus设置为Signal，然后在parkAndCheckInterrupt调用LockSupport.park将线程挂起，等待唤醒。

(3)当持有锁的线程unlock时，会调用sync的release方法，最后调用LockSupport.unpark，将state的值设置为0，表示没有线程持有锁。此时队列中的线程被唤醒进行获取锁，头节点出队，将下一个结点变为虚拟的头节点。

## 31、HashMap的容量为什么是2的幂次方？

在HashMap中计算index的位置采用的是k & (length -1)的位运算，只有保证length的长度为2的幂次方才能计算出0-length的所有位置。

## 32、HashMap计算HashCode的时候为什么要进行位移和异或运算？

由于计算index位置时采用与运算，所以导致高位的数字不论怎么变化都会得到相同的位置，为了充分利用高位进行位置计算进行了位移或者异或，减小哈希碰撞的概率。

## 33、HashMap的扩容机制？

在1.7中HashMap先扩容再添加元素，1.8时先添加元素在进行扩容操作。并且在1.7时采用头插法而1.8采用尾插法，多个线程同时对这个HashMap进行put操作，而察觉到内存容量不够，需要进行扩容时，多个线程会同时执行resize操作，由于采用头插法会改变链表的顺序，

## 34、HashMap的链表结点大于8一定会进行树化吗？

必须在当前总元素大于64的情况之下才会进行树化，否则会进行resize扩容。

## 35、HashMap的put方法的逻辑？

①如果数组为空则创建数组

②如果发现key为0则直接插入table[0]的位置

③计算key的哈希值和哈希值对应的下标位置

④查看是否有链表，有循环看是否有相同的key，有则直接进行赋值

⑤添加节点

## 36、JVM 是如何处理异常的？

在一个方法中如果发生异常，这个方法会创建一个异常对象，并转交给 JVM，该异常对象包含异常名称，异常描述以及异常发生时应用程序的状态。创建异常对象并转交给 JVM 的过程称为抛出异常。可能有一系列的方法调用，最终才进入抛出异常的方法，这一系列方法调用的有序列表叫做调用栈。

JVM 会顺着调用栈去查找看是否有可以处理异常的代码，如果有，则调用异常处理代码。当 JVM 发现可以处理异常的代码时，会把发生的异常传递给它。如果 JVM 没有找到可以处理该异常的代码块，JVM 就会将该异常转交给默认的异常处理器（默认处理器为 JVM 的一部分），默认异常处理器打印出异常信息并终止应用程序。

## 37、final、finally、finalize 有什么区别？

final可以修饰类、变量、方法，修饰类表示该类不能被继承、修饰方法表示该方法不能被重写、修饰变量表示该变量是一个常量不能被重新赋值。

finally一般作用在try-catch代码块中，在处理异常的时候，通常我们将一定要执行的代码方法finally代码块中，表示不管是否出现异常，该代码块都会执行，一般用来存放一些关闭资源的代码。

finalize是一个方法，属于Object类的一个方法，而Object类是所有类的父类，Java 中允许使用 finalize()方法在垃圾收集器将对象从内存中清除出去之前做必要的清理工作。

## 38、try-catch-finally中，如果catch中return了，finally还会执行吗？

会执行，在return前执行。在finally中改变返回值的做法是不好的，因为如果存在finally代码块，try中的return语句不会立马返回调用者，而是记录下返回值待finally代码块执行完毕之后再向调用者返回其值，然后如果在finally中修改了返回值，就会返回修改后的值。显然，在finally中返回或者修改返回值会对程序造成很大的困扰，C#中直接用编译错误的方式来阻止程序员干这种龌龊的事情，Java 中也可以通过提升编译器的语法检查级别来产生警告或错误。

public static int getInt() {  
 int a = 10;  
 try {  
 System.*out*.println(a / 0);  
 a = 20;  
 } catch (ArithmeticException e) {  
 a = 30;  
 return a;  
 */\*  
 \* return a 在程序执行到这一步的时候，这里不是return a 而是 return 30；这个返回路径就形成了  
 \* 但是呢，它发现后面还有finally，所以继续执行finally的内容，a=40  
 \* 再次回到以前的路径,继续走return 30，形成返回路径之后，这里的a就不是a变量了，而是常量30  
 \*/* } finally {  
 a = 40;  
 }  
 return a;  
}

## 39、类 ExampleA 继承 Exception，类 ExampleB 继承ExampleA。

①输出什么？

try {  
 throw new ExampleB("b");  
} catch (ExampleA e) {  
 System.*out*.println("ExampleA");  
} catch (Exception e) {  
 System.*out*.println("Exception");  
}

输出ExampleA

②输出什么？

class Annoyance extends Exception { }  
class Sneeze extends Annoyance { }

public static void main(String[] args) throws Exception {  
 try {  
 try {  
 throw new Sneeze();  
 } catch (Annoyance a) {  
 System.*out*.println("Caught Annoyance");  
 throw a;  
 }  
 } catch (Sneeze s) {  
 System.*out*.println("Caught Sneeze");  
 return;  
 } finally {  
 System.*out*.println("Hello World!");  
 }  
}

输出：

Caught Annoyance

Caught Sneeze

Hello World!

# Spring

## 1、IOC 的初始化流程?

## 2、SpringAOP的常用注解？

①@Before：前置通知，目标方法之前执行

②@After：后置通知，目标方法之后执行(始终执行)

③@AfterReturning：返回后通知，执行方法结束前执行(异常不执行)

④@AfterThrowing：异常通知，出现异常后执行

⑤@Around：环绕通知，环绕目标方法执行

## 3、AOP的全部执行顺序是什么？(boot1和boot2的AOP有什么不同)

①在Spring4时执行顺序为：

正常：@Around->@Before->@Around->@After->@AfterReturning

异常：@Around->@Before->@After->@AfterThrowing

②在Spring5时执行顺序为：

正常：@Around->@Before->@AfterReturning->@After->@Around

异常：@Around->@Before->@AfterThrowing->@After

## 4、Spring如何解决循环依赖？(或者直接问三级缓存)

循环依赖是指多个bean之间的相互依赖，形成闭环，比如A依赖B，B依赖A。Spring只能解决setter注入的单例Bean，否则会抛出BeanCurrentlyInCreationException的异常。

Spring内部通过三级缓存来解决循环依赖问题。

在DefaultSingletonBeanRegistry中有三个Map

一级缓存：singletonObjects，存放经历了完整生命周期的Bean对象。是一个ConcurrentHashMap，初始容量256

二级缓存：earlySingletonObjects，存放早期暴露出来的Bean对象，Bean的生命周期还未结束（属性还未填充完毕），是HashMap 初始16

三级缓存：singletonFactory，存放可以生成Bean的工厂，是HashMap 初始16。

以A依赖B，B依赖A为例

①首先创建beanA，调用getSingleton方法尝试从一级缓存中获取beanA，若找到直接返回，若将A添加到singletonsCurrentlyInCreation的Map中表示beanA正在创建，然后调用doCreate方法开始创建beanA。

②在doCreate方法中，先反射调用构造器创建beanA的实例，然后判断是否为单例，是否允许早起暴露，是否正在创建中，为true则将beanA放入singletonFactories三级缓存中。然后调用populateBean方法为beanA填充属性。

③对beanA进行属性填充时发现beanA依赖beanB，于是再调用doGetBean->getSingleton，在缓存中查找B，没有beanB进行B的创建，然后给beanB添加属性，将beanB添加到三级缓存。

④由于beanB依赖beanA，然后调用getSingleton()获取beanA。此时的beanA的创建工厂在三级缓存中，于是beanB拿到了beanA的实例，beanB顺利完成初始化，并将beanA从三级缓存移动到二级缓存。beanB由三级缓存直接存到一级缓存。

⑤beanB创建完成，beanA继续属性填充，由于beanB创建完成，beanA也创建完成，并调用addSingleton方法将beanA从二级缓存放入一级缓存。

## 5、为什么要包装一层ObjectFactory对象？(为什么有第三级缓存)

如果创建的Bean有对应的代理，那么其他对象注入时，注入的应该是代理对象，但是Spring无法提前知道这个对象是不是有循环依赖。在正常情况下，Spring都是在创建好完成品Bean之后才创建对应的代理。

此时Spring便有两种选择：①不管是否存在循环依赖都提前创建好代理对象，并将对象放入缓存，出现循环依赖时，直接从缓存中取出代理对象并注入。②不提前创建好代理对象，在出现循环依赖被其他对象注入时，才实时生成代理对象。这样在没有循环依赖的情况下，Bean就可以按照Spring的设计原则的步骤进行创建。Spring选择了第二种，在对象外边包装一层ObjectFactory，提前曝光ObjectFactory，在被注入时才在ObjectFactory.getObject方法内实时生成代理对象，并将代理好的对象放到二级缓存。

## 6、三级缓存能不能变为二级缓存？

可以，假如Spring选择第一种处理方式，在曝光半成品之时，就直接执行getEarlyBeanReference创建代理，并放入缓存到earlySingltonObjects。这样就不需要ObjectFactory来延迟执行getEarlyBeanReference，也就不需要singletonFactories这个缓存。但如果使用二级缓存，意味着Bean在构造完之后就创建代理对象，这样违背了Spring的设计原则。

因为Spring的AOP是在Bean的创建完成之后通过AnnotationAware

AspectJAutoProxyCreator后置处理器的postProcessAfterInitialization方法中对初始化后的Bean完成AOP代理。假如出现循环依赖，没有办法，只有先给Bean创建代理，但是在没有出现循环依赖的情况下，设计的初衷就是在Bean的生命周期的最后完成代理而不是实例化之后就立马代理。

# 计算机网络

## 1、七层网络模型

七层模型，也称OSI参考模型，分为应用层、表示层、会话层、传输层、网络层、数据链路层和物理层。

## 2、为什么有了七层还有五层的概念?

网络的统一需要有一个统一的标准，当时有OSI的七层网络模型和TCP/IP的四层结构，当时竞争时几乎所有人都认为OSI会赢，但是胜出者却是TCP/IP,由于大家习惯了使用OSI的理论知识，现在要用TCP/IP的结构会重新学很多东西，于是就开始用OSI的七层结构解释TCP/IP的四层结构，于是五层协议就由此而来。将OSI的应用层、表示层和会话层变为应用层为第五层。

## 3、为什么要三次握手？

TCP 需要 seq 序列号来做可靠重传或接收，而避免连接复用时无法分辨出 seq 是延迟或者是旧链接的 seq，因此需要三次握手来约定确定双方的 ISN（初始 seq 序列号）。

TCP A TCP B  
 1.CLOSED LISTEN  
 2.SYN-SENT -> <SEQ=100><CTL=SYN> -> SYN-RECEIVED  
 3.ESTABLISHED<- <SEQ=300><ACK=101><CTL=SYN,ACK><- SYN-RECEIVED  
 4.ESTABLISHED-> <SEQ=101><ACK=301><CTL=ACK> -> ESTABLISHED  
 5.ESTABLISHED-><SEQ=101><ACK=301><CTL=ACK><DATA>-> ESTABLISHED  
 Basic 3-Way Handshake for Connection Synchronization

第二行中，A 发送了SEQ = 100，标志位是 SYN；

第三行，B 发回了 ACK 101 与 SEQ 300，标志位是 SYN 与 ACK（两个过程合并了）。注意，ACK 是101意味着，B 希望接收到 101序列号开始的数据段。

第四行，A 返回了空的数据，SEQ 101， ACK 301，标志位为 ACK。至此，双方的开始 SEQ （也就是 ISN）号100与300都被确认接收到了。 第五行，开始正式发送数据包，注意的是 ACK 依旧是第四行的301，因为没有需要 ACK 的 SYN 了（第四行已经 ACK 完）。

## 4、四次挥手

由于TCP连接是全双工的，因此每个方向都必须单独进行关闭。这原则是当一方完成它的数据发送任务后就能发送一个FIN来终止这个方向的连接。收到一个 FIN只意味着这一方向上没有数据流动，一个TCP连接在收到一个FIN后仍能发送数据。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。

（1）TCP客户端发送一个FIN，用来关闭客户到服务器的[数据传送](https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E4%BC%A0%E9%80%81" \t "https://baike.baidu.com/item/%E5%9B%9B%E6%AC%A1%E6%8C%A5%E6%89%8B/_blank)。

（2）服务器收到这个FIN，它发回一个ACK，确认序号为收到的序号加1。和SYN一样，一个FIN将占用一个序号。

（3）服务器关闭客户端的连接，发送一个FIN给客户端。

（4）客户端发回ACK[报文](https://baike.baidu.com/item/%E6%8A%A5%E6%96%87" \t "https://baike.baidu.com/item/%E5%9B%9B%E6%AC%A1%E6%8C%A5%E6%89%8B/_blank)确认，并将确认序号设置为收到序号加1。

# Redis

## 1、Redis的传统五大数据类型及其使用场景？

①string：商品编号、订单号采用INCR命令生成；点赞数；阅读数

②list：有序有重复，微信订阅公众号

③hash：redis的hash<---->java Map<String,Map<K,V>>

Key field value 可以做购物车

④set：无需无重复 做微信抽奖小程序、朋友圈点赞、好友关注社交关系、 QQ内推可能认识的人

⑤zset(sorted set) 向有序集合中加入元素和分数

根据商品销售对商品排序显示、热搜、排行榜

注：redis后边新增了bitmap(位图)、HyperLogLog(统计)、GEO(地理)、Stream；命令不区分大小写但key区分大小写；help @类型名词 查看命令

## 2、Redis的除了做缓存你还用来见过什么用法？

①购物车 ②分布式锁

## 3、Redis做分布式锁的时候需要注意的问题？

①设置分布式锁一定要设置过期时间防止宕机导致无法获取锁。

②设置锁和设置过期时间一定要保证原子性。

③删除锁要防止误删除别人的锁。

④判断是否为自己的锁时一定要保证原子性防止出现加锁解锁不是一个客户端导致误解锁。官方推荐使用lua脚本(或者Redis事务)

⑤若业务未结束需要给缓存续期。Redisson采用watch dog策略每10s将续到30s。

## 4、如果Redis是单点部署的，会带来什么问题？

## 5、Redis的集群模式有没有问题？

## 6、Redis事务

(1)事务介绍：

①Redis的事务是通过MULTI、EXEC、DISCARD和WATCH这四个命令完成的。

②Redis的单个命令都是原子性的，所以这里确保事务性的对象是命令集合。

③Redis将命令集合序列化并确保处于一事务的命令集合连续且不被打断的执行。

④Redis不支持回滚操作。

(2)相关命令

①MULTI：用于标记事务的开始。Redis会将后续的命令逐个放入队列中，然后使用EXEC命令原子化地执行这个命令序列。

②EXEC：在一个事务中执行所有先前放入队列的命令，然后恢复正常的连接状态。

③DISCARD：清除所有先前在一个事务中放入队列的命令，然后恢复正常的连接状态。

④WATCH：当某个事务需要按条件执行时，就要使用这个命令将给定的键设置为受监控的状态。该命令可以实现Redis的乐观锁

⑤UNWATCH：清除所有先前为一个事务设置监控的键。

## 7、Redis的内存有多大？

如果不设置最大内存大小或者设置最大内存大小为0，在64位操作系统下不限制内存大小，32位操作系统最多使用3GB的内存。

可以在配置文件中设置：maxmemory <bytes> 建议最大内存的3/4

动态进行配置：config set maxmemory 1000

查看内存：info memory

## 8、Redis的缓存满了怎么办？

满了会报OOM：

OOM command not allowed when use memory > ‘maxmemory’

## 9、Redis的过期键淘汰策略？

如果一个键是过期的，到了过期时间不会马上就从内存中删除。具体有三种：

①定时删除 影响CPU ②惰性删除 占用内存

③定期删除：每隔一段时间进行一次删除操作。

## 10、Redis的缓存淘汰策略？(兜底方法，内存即将打满)

①noeviction：不会驱逐任何key 默认

②allkeys-lru：对所有key使用LRU算法进行删除 推荐

③volatile-lru：对所有设置过期时间的key使用LRU进行删除

④allkeys-random：对所有的key随机删除

⑤volatile-random：对所有设置过期时间的key随机删除

⑥allkey-lfu：对所有key使用LFU算法进行删除

⑦volatile-lfu：对所有设置过期时间的key使用LFU进行删除

⑧volatile-ttl：删除马上要过期的key

## 11、手写一个LRU算法(核心思想HashMap+双向链表)

## 12、Redis 缓存穿透 + 缓存雪崩 + 缓存击穿？

①缓存穿透：缓存穿透是指查询一个根本不存在的数据，缓存和数据库都不会命中。但在日常工作中出于容错的考虑，如果从数据库查不到数据则不写入缓存，导致不存在的数据都要到持久层去查询，失去了保护数据库的意义。

**解决：**缓存一个空对象(key null)或者使用布隆过滤器拦截(采用bitmap).在访问缓存和数据库之前，将存在的key用布隆过滤器提前保存起来，做第一层拦截，当收到一个对key请求时先用布隆过滤器验证是key否存在，如果存在在进入缓存层、存储层。

②缓存雪崩：由于缓存承载了大量的请求，有效的保护了数据库，但如果缓存由于某种原因宕机或者缓存时间在同一时间失效，大量请求直接访问数据库，数据库压力过大造成雪崩。

**解决：**缓存进行水平扩展(sentinel和cluster实现)；采用多级缓存；将缓存的过期时间设置为随机数。

③缓存击穿：在缓存失效的瞬间，有大量的线程用来重建缓存，造成后端负载过大，导致程序崩溃。

**解决：**使用分布式互斥锁，只允许一个线程重建缓存，其他线程等待重建缓存的线程执行完之后，重新从缓存获取数据即可。

# Mybatis

## 1、什么是Mybatis？

Mybatis是一个半ORM框架，内部封装了JDBC，开发时只需要关注SQL语句本身而不需要花费精力去处理加载驱动、创建连接、创建statement等复杂的过程。程序员只需要编写原生sql，可以严格控制sql的执行性 能，灵活度较高。

优点：①基于sql编写，不会对程序或者数据库的现有设计造成任何影响。②消除了JDBC的大量冗余代码。③能与各种数据库进行很好的集成。

缺点：①sql工作量较大。②sql语言依赖数据库，导致数据库的移植性较差，不能随意的更换数据库。

## 2、#{}和${}的区别是什么？

#{}是预编译处理，${}是字符串替换，Mybatis在处理#{}时，会将sql中的#{}替换为?号，Mybatis在处理${}时，就是把${}替换成变量的值。

## 3、Mybatis的一级、二级缓存？

一级缓存: 基于PerpetualCache的HashMap本地缓存，其存储作用域为 Session，当Session flush或close之后，该Session中的所有Cache就将清空，默认打开一级缓存。当在同一个sqlSession中执行两次相同的sql语句时，第一次执行完毕会将数据库中查询的数据写到缓存（内存），第二次查询时会从缓存中获取数据，不再去底层数据库查询，从而提高查询效率。

二级缓存与一级缓存其机制相同，默认也是采用PerpetualCache，HashMap存储，不同在于其存储作用域为Mapper(Namespace)，并且可自定义存储源，如 Ehcache。默认不打开二级缓存，要开启二级缓存，使用二级缓存属性类需要实现Serializable序列化接口(可用来保存对象的状态),可在它的映射文件中配置<cache/>;二级缓存的作用域是mapper的同一个namespace。不同的sqlSession两次执行相同的namespace下的sql语句，且向sql中传递的参数也相同，即最终执行相同的sql语句，则第一次执行完毕会将数据库中查询的数据写到缓存，第二次查询会从缓存中获取数据，不再去底层数据库查询，从而提高效率。

对于缓存数据更新机制，当某一个作用域(一级缓存 Session/二级缓存Namespaces)的进行了C/U/D 操作后，默认该作用域下所有 select 中的缓存将被 clear。