**Java基础**

**讲讲你所遇到过的异常**

​ ConcurrentmodificationException

​ 这个异常是在多线程操作集合的时候产生的写并发写入的异常原因是因为操作没有加锁的对象例如ArrayList，HashSet，HashMap的时候，具体表现在高并发的添加上

**HashMap**

​ HashMap核心变量：

​ Node<K,V>：链表节点，包含了key、value、hash、next指针四个元素

​ table：Node<K,V>类型的数组，里面的元素是链表，用于存放HashMap元素的实体

​ loadFactor：负载因子

​ size：记录了放入HashMap的元素个数

​ threshold：阈值，决定了HashMap何时扩容，以及扩容后的大小，一般等于table大小乘以

​ loadFactor

**HashMap有几个构造方法？**

​ 答：4个

​ 分别是

//此方法用于设置HashMap的初始化长度，还有负载因子  
public HashMap(int initialCapacity, float loadFactor) {    
   ...    
   this.loadFactor = loadFactor;    
   this.threshold = tableSizeFor(initialCapacity);    
}    
​  
//此方法可以定义初始化长度  
public HashMap(int initialCapacity) {    
   this(initialCapacity, DEFAULT\_LOAD\_FACTOR);    
}    
    
//默认初始化值  
public HashMap() {    
   this.loadFactor = DEFAULT\_LOAD\_FACTOR; // all other fields defaulted    
}    
    
​  
public HashMap(Map<? extends K, ? extends V> m) {    
   this.loadFactor = DEFAULT\_LOAD\_FACTOR;    
   putMapEntries(m, false);   
}

**HashMap的源码，实现原理，JDK8中对HashMap做了怎样的优化。**

在JDK1.6，JDK1.7中，HashMap采用数组+链表实现 ，而JDK1.8中，HashMap采用数组+链表+红黑树实现，当链表长度超过阈值（8）时，将链表转换为红黑树，这样大大减少了查找时间。

​ 核心重点：1.6,1.7 数组+链表

​ 1.8 数组+链表 长度超过8 -----> 转换为红黑树

**HaspMap扩容是怎样扩容的，为什么都是2的N次幂的大小。**

​ 为了能让 HashMap 存取高效，尽量较少碰撞，Hash 值的范围值-2147483648到2147483648，前后加起来大概40亿的映射空间。用之前还要先做对数组的长度取模运算，这个数组下标的计算方法是“ (n - 1) & hash ”。（n代表数组长度）。这也就解释了 HashMap 的长度为什么是2的幂次方。

**HashMap，HashTable，ConcurrentHashMap的区别。**

​ **HashTable**

​ 底层数组+链表实现，无论key还是value都不能为null，线程安全，实现线程安全的方式是在修改数据时锁住-整个HashTable，效率低，ConcurrentHashMap做了相关优化

​ 初始size为11，扩容：newsize = olesize\*2+1

​ 计算index的方法：index = (hash & 0x7FFFFFFF) % tab.length

**HashMap**

底层数组+链表实现，可以存储null键和null值，线程不安全

初始size为16，扩容：newsize = oldsize\*2，size一定为2的n次幂

扩容针对整个Map，每次扩容时，原来数组中的元素依次重新计算存放位置，并重新插入

插入元素后才判断该不该扩容，有可能无效扩容（插入后如果扩容，如果没有再次插入，就会产生无效扩容）

当Map中元素总数超过Entry数组的75%，触发扩容操作，为了减少链表长度，元素分配更均匀

计算index方法：index = hash & (tab.length – 1)

ConcurrentHashMap

1.6时底层采用分段的数组+链表实现，线程安全

通过把整个Map分为N个Segment，可以提供相同的线程安全，但是效率提升N倍，默认提升16倍。(读操作不加锁，由于HashEntry的value变量是 volatile的，也能保证读取到最新的值。)

1.8版本时采用数组+链表+红黑树

取消segments字段，直接采用transient volatile HashEntry<K,V> table保存数据，采用table数组元素作为锁，从而实现了对每一行数据进行加锁，进一步减少并发冲突的概率。

Hashtable的synchronized是针对整张Hash表的，即每次锁住整张表让线程独占，ConcurrentHashMap允许多个修改操作并发进行，其关键在于使用了锁分离技术

有些方法需要跨段，比如size()和containsValue()，它们可能需要锁定整个表而而不仅仅是某个段，这需要按顺序锁定所有段，操作完毕后，又按顺序释放所有段的锁

扩容：段内扩容（段内元素超过该段对应Entry数组长度的75%触发扩容，不会对整个Map进行扩容），插入前检测需不需要扩容，有效避免无效扩容

**极高并发下HashTable和ConcurrentHashMap哪个性能更好，为什么，如何实现的**

​ ConcurrentHashMap 的性能仍然保持上升趋势，而 Hashtable 的性能则随着争用锁的情况的出现而立即降了下来。

​ 核心：

​ Hashtable使用 synchronized 作为锁，而在高并发情况下疯狂抢锁会损耗性能，ConcurrentHashMap使用锁桶（或段）。 ConcurrentHashMap将hash表分为16个桶（默认值），诸如get,put,remove等常用操作只锁当前需要用到的桶 ，所以他在高并发情况系会比HashTable快很多

**HashMap在高并发下如果没有处理线程安全会有怎样的安全隐患，具体表现是什么**

Hashmap在并发环境下，可能出现的问题：

​ 1、多线程put时可能会导致get无限循环，具体表现为CPU使用率100%；

​ 在HashMap每次put的时候都会检测，他的长度跟负载因子算出的阙值，如果在put的时候原来的Hash表放不下了，那么就会进行扩容， 扩容的长度为两倍，例如16的长度负载因子12，如果达到13那么就扩容到32，如果下次达到24那么就会再次扩容，他扩容的方式就是创建一个新的Hash表将原来的数据存储进去，这个过程就是一个数据的迁移过程，rehash ()的过程，多线程操作就有可能形成循环链表，如果使用get方法就会出现Infinite Loop的情况，在无限循环的过程中会造成cpu占满从而引起卡死，崩溃等等情况

​ 2、多线程put时可能导致元素丢失

​ 当多个线程同时执行addEntry(hash,key ,value,i)时，如果产生哈希碰撞，导致两个线程得到同样的bucketIndex去存储，就可能会发生元素覆盖丢失的情况

**ArrayList**

**ArrayList的初始化以及扩容的实现过程**

​ ArrayList的初始化他的容器大小时0，因为ArrayList在初始化的时候不会进行容器的初始化

​ 他会在第一次添加的时候给他的容器进行赋值，他的大小是10

​ 并且在他的扩容的时候是在add的时候容器大小不够触发，扩容为原来容器的1.5倍

**什么情况下你会使用ArrayList？什么时候你会选LinkedList？**

​ 在ArrayList中使用的是数组，那么他的查询效率就会高，因为他的查询时间复杂度为0（1），而LinkedList的采用的是双向链表，所以他的查询的时间复杂度为O（N），所以在查询比较多的情况下我们使用ArrayList，那么又来看一下使用LinkedList的场景，我们知道ArrayList的删的时候他会去进行一个位移的操作，在中间添加的时候他会查询并且位移，那么他的时间复杂度为O（n），而LinkedList采用双向链表，那么他可以从节点的上一个节点指向节点的下一个直接就进行了删除时间复杂度为O（1），所以在频繁的操作元素的时候使用LinkedList

**类**

**java中四种修饰符的限制范围。**

​ public

​ 不同包，非子类，只要项目内都可以访问

​ protected

​ 只有当前类和子类还有本包能够使用，

​ default

​ 只有当前类和本包能够使用

​ private

​ 只有本类能够使用

**Object类中的方法。**

1 registerNatives()   //私有方法  
   
 2 getClass()   //返回此 Object 的运行类。  
   
 3 hashCode()   //用于获取对象的哈希值。  
   
 4 equals(Object obj)     //用于确认两个对象是否“相同”。  
   
 5 clone()   //创建并返回此对象的一个副本。   
   
 6 toString()   //返回该对象的字符串表示。     
   
 7 notify()   //唤醒在此对象监视器上等待的单个线程。     
   
 8 notifyAll()     //唤醒在此对象监视器上等待的所有线程。     
   
 9 wait(long timeout)   //在其他线程调用此对象的 notify() 方法或 notifyAll() 方法，或       者超过指定的时间量前，导致当前线程等待。   
   
10 wait(long timeout, int nanos)   //在其他线程调用此对象的 notify() 方法或 notifyAll() 方法，或者其他某个线程中断当前线程，或者已超过某个实际时间量前，导致当前线程等待。  
​  
11 wait()   //用于让当前线程失去操作权限，当前线程进入等待序列  
​  
12 finalize()   //当垃圾回收器确定不存在对该对象的更多引用时，由对象的垃圾回收器调用此方法。

**动态代理**

**动态代理的两种方式，以及区别。**

​ **一般而言，动态代理分为两种，一种是JDK反射机制提供的代理，另一种是CGLIB代理。在JDK代理，必须提供接口，而CGLIB则不需要提供接口，在Mybatis里两种动态代理技术都已经使用了，在Mybatis中通常在延迟加载的时候才会用到CGLIB动态代理。**

​ 通过实现接口InvocationHandler，然后在他的invoke方法上面进行动态代理

​ CGLIB的类似于jdk的动态代理，但是不用提供接口

**JAVA库相关**

**Java序列化的方式。**

​ 1、实现Serializable接口

​ 2、实现Externalizable接口

**传值和传引用的区别，Java是怎么样的，有没有传值引用**

* 传值：传递的是值的副本。方法中对副本的修改，不会影响到调用方。
* 引用：传递的是引用的副本，共用一个内存，会影响到调用方。此时，形参和实参指向同一个内存地址。对引
* 用副本本身（对象地址）的修改

**一些较新的东西JDK8的新特性，流的概念及优势，为什么有这种优势**

​ 个人感觉，一项技术需要有生命周期地进行更新，所以会出现一些新式的东西，可能好用也有可能不好用，但是他的存在肯定是有意义的，像Lambda表达式和Stream流这是JDK8的新特性中比较火的两个特性，那么他们分别又做了什么呢？先来说说Lambda它体现了函数式编程这个概念，其实在很多地方已经有这个概念了，例如前端ES6的规范中，而且他不单单是一个概念，也是一个革新，他还能帮助我们简化代码，尤其是在编写匿名内部类的时候，这就是他为什么会产生（因为这个概念是已经有的，并且8的版本开发出来了，而且也能简洁代码，还能体现函数式编程），那么再来说一下Stream流的概念，

​ 流的概念数据流（data stream）最初是通信领域使用的概念，代表传输中所使用的信息的数字编码信号序列。然而，我们所提到的数据流概念与此不同。这个概念最初在1998年由Henzinger在文献87中提出，他将数据流定义为“只能以事先规定好的顺序被读取一次的数据的一个序列”。

​ 那么Stream流有什么优势呢？

（1）速度更快

（2）代码更少（增加了新的语法Lambda表达式）

（3）强大的Stream API

（4）便于并行

（5）最大化减少了空指针异常Optional

其中最为核心的为 Lambda 表达式与Stream API

​ 流注重的不是不是数据，而是计算，Stream他并不会自己存储元素，只能通过其他获取，而且他也不会修改源对象，只会产生新的数据，他的流式计算方便快捷，效率非常高，但是不适合作为存储容器，他更适合计算数据，数据筛选之类

**@transactional注解在什么情况下会失效，为什么**

1，检查方法是不是public

2，检查异常是不是unchecked异常

3，如果是checked异常也想回滚的话，注解上写明异常类型即可 @Transactional(rollbackFor=Exception.class)

**数据结构和算法**

**B+树**

**快速排序，堆排序，插入排序**

**一致性Hash算法，一致性Hash算法的应用**

**链表和数组的优缺点？**

**解决hash冲突的方法有哪些？**

**JVM**

**JVM的启动方式**

JVM有两种运行模式Server与Client。两种模式的区别在于

​ HotSpot 虚拟机默认使用Server模式启动

​ Client模式启动速度较快

​ Server模式启动较慢

​ 但是启动进入稳定期长期运行之后Server模式的程序运行速度比Client要快很多。这是因为Server模式启动的JVM采用的是重量级的虚拟机，对程序采用了更多的优化；而Client模式启动的JVM采用的是轻量级的虚拟机。所以Server启动慢，但稳定后速度比Client远远要快。

**JVM的内存结构**

**G1**

​ G1将新生代，老年代的物理空间划分取消了 ，取而代之的是，G1算法将堆划分为若干个区域（Region），它仍然属于分代收集器 。不过，这些区域的一部分包含新生代，新生代的垃圾收集依然采用暂停所有应用线程的方式，将存活对象拷贝到老年代或者Survivor空间。老年代也分成很多区域，G1收集器通过将对象从一个区域复制到另外一个区域，完成了清理工作。这就意味着，在正常的处理过程中，G1完成了堆的压缩（至少是部分堆的压缩），这样也就不会有cms内存碎片问题的存在了

**JVM方法栈的工作过程，方法栈和本地方法栈有什么区别**

​ 方法栈：存储栈帧，支持Java方法的调用、执行和退出

​ 本地方法栈：作用是支撑Native方法的调用、执行和退出

​ 本地方法栈Native方法是基于系统还有底层的实现，它并不是调用的Java代码

**JVM的栈中引用如何和堆中的对象产生关联**

​ 栈中的栈帧用来存储内存地址，通过PC计数器进行引用，它类似于C中的指针，可以帮助栈引用到堆中的堆内存对象

**说一下逃逸分析技术**

**逃逸分析**

​ 逃逸是指在某个方法之内创建的对象除了在方法体之内被引用之外，还在方法体之外被其它变量引用到；这样带来的后果是在该方法执行完毕之后，该方法中创建的对象将无法被GC回收。由于其被其它变量引用，由于无法回收，即称为逃逸。

​ 逃逸分析技术可以分析出某个对象是否永远只在某个方法、线程的范围内，并没有“逃逸”出这个范围，逃逸分析的一个结果就是对于某些未逃逸对象可以直接在栈上分配提高对象分配回收效率，对象占用的空间会随栈帧的出栈而销毁。

​ 总结：虽然他被引用之后导致了无法回收，但是它是随着栈的结束而结束掉的，而一个栈对应一个线程，这个线程一结束就会被销毁了

**GC的常见算法，CMS以及G1的垃圾回收过程，CMS的各个阶段哪两个是Stop the world的，CMS会不会产生碎片，G1的优势**

​

**标记清除和标记整理算法的理解以及优缺点**

​ 标记清除：就是将要被回收的内存对象进行标记，然后统一的去清除，但是在这个过程中会产生大量的不连续的内存碎片，并且标记然后清除的效率不高

​

​ 标记整理：它是将标记的 内存进行先标记然后整理然后再清除，所以他不会产生内存碎片，但是他所耗费的资源更多

​

**JVM如何判断一个对象是否该被GC，可以视为root的都有哪几种类型**

​ 是根据他的引用强度进行垃圾回收的，引用分为四种

​ 强引用(Strong)

​ 就是我们平时使用的方式 A a = new A();强引用的对象是不会被回收的

​ 软引用(Soft)

​ 在jvm要内存溢出(OOM)时，会回收软引用的对象，释放更多内存

​ 是用来描述一些还有用但并非必需的对象。对于软引用关联着的对象，在系统将要发生内存溢出异

​ 常之前，将会把这些对象列进回收范围之中进行第二次回收。

​ 弱引用(Weak)

​ 在下次GC时，弱引用的对象是一定会被回收的

​  也是用来描述非必需对象的，但是它的强度比软引用更弱一些，被弱引用关联的对象只能生存到下

​ 一次垃圾收集发生之前。当垃圾收集器工作时，无论当前内存是否足够，都会回收掉只被弱引用关

​ 联的对象。

​ 虚引用(Phantom)

​ 对对象的存在时间没有任何影响，也无法引用对象实力，唯一的作用就是在该对象被回收时收到一

​ 个系统通知

​ 视为root的有

​ 虚拟机栈(JVM stack)中引用的对象(准确的说是虚拟机栈中的栈帧(frames))

​ 方法区中类静态属性引用的对象

​ 本地方法栈(Native Stack)引用的对象

**什么情况下类会被回收掉**

​ 堆中不存在该类的任何实例

​ 加载该类的classloader已经被回收

​ 该类的java.lang.Class对象没有在任何地方被引用，也就是说无法通过反射再带访问该类的信息

**如何判断一个对象需要被回收**

​ 1、引用计数法

​ 引用计数法思路是这样的，给对象添加一个引用计数器，有地方引用时，计数器就加1；当引用失效时就减1；当计数为0的时候就判定对象需要被回收

引用计数法有一个难以解决的问题就是相互循环引用问题。这样就会造成无法回收，并且JVM用的是可达性分析

​ 2、可达性分析算法

​ 这个算法的基本思路是通过一些列称为“GC Roots”的对象作为起始点，从这些点开始向下搜索，搜索走过的路径称为引用链，当一个对象到GC Roots没有任何引用链相连时，则证明对象需要被回收.

​ 被视为Roots的对象有

​ 虚拟机栈中引用的对象

​ 方法区中类静态属性引用的对象

​ 方法区中常量引用的对象

​ 本地方法栈中JNI引用的对象

**强软弱虚引用的区别以及GC对他们执行怎样的操作**

​ 强引用一般在GC的时候都不会去回收他，所以强引用过多JVM就算抛出OOM异常也不会去回收他

​ 对象是虚可达到对象 ，虚引用对象在那时或者在以后的某一时间，它会将该引用加入队列， 所以他其实只是会发出一个通知，但是他在某一个时间段还是会加入队列，所以在每次GC他都会被清理但是每次都又会回归队列

**System.gc()可以进行垃圾回收么？**

​ System.gc()是一个Java提供的函数，看似他是进行Gc操作的，实际上它并不能进行Gc，因为Java并没有提供手动gc的任何方法，那么他的作用是什么呢？

​ System.gc()他只是帮助我们提醒一下虚拟机需要回收垃圾了，但是他并不会一定去执行垃圾回收的操作，只是起到了一个提醒的作用

**Java类加载的过程**

加载：

​ 通过一个类的全限定名来获取定义此类的二进制字节流 ，将这个字节流所代表的静态存储结构转化为方

​ 法区域的运行时数据结构。 在Java堆中生成一个代表这个类的java.lang.Class对象，作为方法区域数据

​ 的访问入口

验证：

​ 验证阶段作用是保证Class文件的字节流包含的信息符合JVM规范，不会给JVM造成危害。如果验证失

​ 败，就会抛出一个java.lang.VerifyError异常或其子类异常。验证过程分为四个阶段 准备：

​  1.文件格式验证：验证字节流文件是否符合Class文件格式的规范，并且能被当前虚拟机正确的处理。

​  2.元数据验证：是对字节码描述的信息进行语义分析，以保证其描述的信息符合Java语言的规范。

​ 3.字节码验证：主要是进行数据流和控制流的分析，保证被校验类的方法在运行时不会危害虚拟机。

​ 4.符号引用验证：符号引用验证发生在虚拟机将符号引用转化为直接引用的时候，这个转化动作将在解

​ 析阶段中发生。

准备 :

​ 准备阶段为变量分配内存并设置类变量的初始化。在这个阶段分配的仅为类的变量（static修饰的变

​ 量），而不包括类的实例变量。对已非final的变量，JVM会将其设置成“零值”，而不是其赋值语句的值

​  pirvate static int size = 12;

​  那么在这个阶段，size的值为0，而不是12。 final修饰的类变量将会赋值成真实的值。

解析：

​ 解析过程是将常量池内的符号引用替换成直接引用。

​ 主要包括四种类型引用的解析。

​ 类或接口的解析、

​ 字段解析、

​ 方法解析、

​ 接口方法解析

初始化 ：

​ 在准备阶段，类变量已经经过一次初始化了，在这个阶段，则是根据程序员通过程序制定的计划去初始

​ 化类的变量和其他资源。这些资源有static{}块，构造函数，父类的初始化等。

​  至于使用和卸载阶段阶段，这里不再过多说明，使用过程就是根据程序定义的行为执行，卸载由GC完

​ 成。

使用 :

​ 新线程---程序计数器----jvm栈执行（对象引用）-----堆内存（直接引用）----方法区

卸载:

​ GC垃圾回收

**双亲委派模型的过程以及优势**

​

启动类(引导类)加载器 Bootstrap ClassLoader

​ 虚拟机的一部分，由c++实现。负责加载<JAVA\_HOME>/lib下的类库

扩展类加载器 Extension ClassLoader,

​ sun.misc.Launcher$ExtClassLoader.负责加载<JAVA\_HOME>/lib/ext下的类库

应用程序类加载器 Application ClassLoader

​ sun.misc.Launcher$AppClassLoader, 它是System.getClassLoader()的返回值，也称为系统类加载器。

​ 负责加载用户类路径上所指定的类库。如果应用程序没有自定义过类加载器

自定义类加载器：

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，而是把这个请求委托给父类加载器去完成，每一个层次的类加载器都是如此  
​  
如果从应用层类加载器加载，那么他会将任务交给父类（扩展类加载器），扩展类加载器将任务交给启动类加载器  
​  
如果启动类加载器无法完成加载任务（找不到需要加载的类）那么他就会将任务递交给子类，如果启动类加载器加载不了则交给扩展类加载器，拓展类加载到了那么就直接返回结果，如果还是加载不到则继续往下，如果一直加载不了那么就会无法加载此类找不到类  
​  
​  
java.lang.ClassNotFoundException

**常用的JVM调优参数**

​ **Trace跟踪参数**

​ 打开GC跟踪日志（每次执行GC的信息都能打印，获得执行时间，空间大小）：

​ -verbose:gc 或 -XX:+printGC 或 -XX:+printGCDetails

​ 类加载监控：（监控类加载的顺序）

​ -XX:+TraceClassLoading

​ **堆的分频参数**

​ -Xmx10M 指定最大堆，JVM最多能够使用的堆空间 （超过该空间引发OOM）

​ -Xms5M 指定最小堆，JVM至少会有的堆空间（尽可能维持在最小堆）

​ -Xmn 11M（new） 设置新生代大小

​ **栈的分配参数**

​ -Xss 每个线程都有独立的栈空间（几百k，比较小）

​ 需要大量线程时，需要尽可能减小栈空间

​ 栈空间太小-----StackOverFlow栈溢出（一般递归时产生大量局部变量导致）

**Java有没有主动触发GC的方式**