JAVA面试题准备

基础题目

Java线程的状态

新建（newThread） 就绪（runnable）调用了该对象的start()方法。该状态的线程位于可运行线程池中，等待被线程调度选中，获取cpu 的使用权 。 运行（running）获得了cpu 时间片（timeslice） ，执行程序代码 死亡（dead）线程run()、main() 方法执行结束，或者因异常退出了run()方法，则该线程结束生命周期。死亡的线程不可再次复生。 堵塞（blocked）即让出了cpu timeslice，暂时停止运行。直到线程进入可运行(runnable)状态，才有机会再次获得cpu timeslice 转到运行(running)状态

进程和线程的区别，进程间如何通讯，线程间如何通讯

进程的通讯

信号量

消息队列

套接字Socket

共享内存

管道通信

线程的通讯：

实现两个线程分别交替的打印自增的共享变量，奇/偶；

```
wait/notify 等待
Volatile 内存共享
CountDownLatch 并发工具  相当于synchronized
CyclicBarrier 并发工具  初始确定准备数量，满足条件后往下执行。
```

HashMap的数据结构是什么？如何实现的。和HashTable，ConcurrentHashMap的区别

数组+链表+红黑树（1.8新增）；

并发bug1，死循环(java8中不存在了)：put操作引起了扩容，导致对链表进行了倒序处理；

并发bug2，数据丢失：并发可能会少数据。（如果在修改时并发读取，ConcurrentModificationException）

并发bug3，数据重复

Cookie和Session的区别

比较的简单。

索引有什么用？如何建索引?

show index from 表名;

如何建索引：ALTER TABLE <表名> ADD INDEX 索引名[可选] (<字段>);

ArrayList是如何实现的，ArrayList和LinedList的区别？ArrayList如何实现扩容。

数组、链表；数组初始长度为10；需要扩容时1.5倍。

equals方法实现

Object中默认比较的是地址值；一般会根据业务需要进行重写，并且为了保证在容器中顺利使用，要对hashcode()进行重写。

面向对象

抽象，继承，多态，封装

线程状态，BLOCKED和WAITING有什么区别

BLOCKED状态 线程处于BLOCKED状态的场景。当前线程在等待一个monitor lock，比如等待执行synchronized代码块或者使用synchronized标记的方法。 在synchronized块中循环调用Object类型的wait方法，如下是样例

```
synchronized(this){
    while (flag){
        obj.wait();
    }
    // some other code
}
```

WAITING状态 线程处于WAITING状态的场景。

调用Object对象的wait方法，但没有指定超时值。 调用Thread对象的join方法，但没有指定超时值。 调用LockSupport对象的park方法。 提到WAITING状态，顺便提一下TIMED_WAITING状态的场景。

TIMED_WAITING状态 线程处于TIMED_WAITING状态的场景。

调用Thread.sleep方法。 调用Object对象的wait方法，指定超时值。 调用Thread对象的join方法，指定超时值。 调用LockSupport对象的parkNanos方法。 调用LockSupport对象的parkUntil方法。

sleep和wait的区别:

调用sleep()方法的过程中，线程不会释放对象锁，但是会让出CPU。Thread 方法。

wait（），会四方锁对象，进入等待此对象的等待锁定池，对此对象调用notify()方法后，本线程才进入对象锁定池准备。synchronized {}中，object方法。

JVM如何加载字节码文件

从虚拟机的视角看，java虚拟机的运行时内润被划分为五个区域：线程共享的方法区和堆；线程私有的PC寄存器，java方法栈，本地方法栈。首先需要将编译后的class文件加载到虚拟机中，java类文件存放在方法区内。

从硬件视角来看，Java 字节码无法直接执行。因此，Java 虚拟机需要将字节码翻译成机器码。

在 HotSpot 里面，上述翻译过程有两种形式：第一种是解释执行，即逐条将字节码翻译成机器码并执行；第二种是即时编译（Just-In-Time compilation，JIT），即将一个方法中包含的所有字节码编译成机器码后再执行。

加载（双亲委派模型）:查找字节流并据此创建类，【类加载器，进行字节流查找】

启动类加载器 C++ 负责加载最基础，最重要的类。JRE 的 lib 目录下 jar 包

拓展类加载器：加载相对次要、但又通用的类，比如存放在 JRE 的 lib/ext 目录下 jar 包中的类（以及由系统变量 java.ext.dirs 指定的类）

应用类加载器：应用程序路径下的类。

链接

将创建成的类合并至 Java 虚拟机中，使之能够执行的过程。它可分为验证、准备以及解析三个阶段。

> class文件中存储的具体信息格式不需要很清楚，但是我们要明白，class文件数据的存储顺序比如按照虚拟机规范的严格格式（前4个字节时魔数，接下来4个字节存放版本号，等等顺序，大小必须正确），否则在运行时无法正确加载。还有一个class文件对应唯一的一个类或接口的定义信息，我们可以在一个java文件中写2个类或者接口（内部类不算）但是编译出来的class文件等于所以class加上interface。
>
> 类加载器加载的类可以是磁盘上的class文件，也可以是网络或者解压得到的二级制文件（符合class格式），这里的class文件是一串二进制的字节流。
>
> 类的加载指的是将类的.class文件中的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这个类的java.lang.Class对象，用来封装类在方法区类的对象。

类加载的过程

Loading->Verfication->Preparation->**Resolution**->Initialization->Using->Unloading

加载验证准备初始化卸载的顺序时确定的。解析则不一定：有可能在初始化前或者初始化后（运行时绑定，又称动态绑定或者晚期绑定）。

加载

(1) 通过一个类的全限定名来获取定义此类的二进制字节流

(2) 将这个字节流所代表的静态存储结构转化为方法的运行时数据结构

(3) 在Java堆中生成一个代表这个类的java.lang.Class对象，作为方法区这些数据的访问入口。

链接

验证：

准备：

正式为类变量分配并设置类变量初始值（在方法区中进行分配进行，默认初始值是每一种类型的初始值）

解析：

解析阶段是虚拟机将常量池内的符号引用替换为直接引用的过程。解析动作主要针对类或接口、字段、类方法、接口方法、方法类型、方法句柄和调用点限定符7类符号引用进行

初始化

是类加载过程的最后一步，初始化阶段是执行类构造器()方法的过程，真正的设定变量的初始值（指定初始值）。

由于父类的()方法先于子类执行，所以父类的static语句先于子类执行，然后是父类的非static语句块和构造方法，接下来是非static语句块的构造方法。

new 指令的目标类，调用静态方法，字段所在类，初始化子类的父类，反射调用方法所在类，MethodHandle指向方法所在类，接口定义default方法直接，间接实现被调用时，初始化接口类。

JVM GC，GC算法

目前主流的垃圾回收器采取的是可达性分析算法。将一系列的GCRoots 作为初始的存活对象合计从该合计出发，探索所有能被该集合引用到的对象，并将其加入到集合中，也就是mark，未被探索到的对象便是死亡的，可以回收的。

GC Roots 由堆外向堆内的引用。包括但不限于：

java栈帧中的局部变量；

已经加载的静态变量；

JNI handles;

已经启动但是未停止的java线程。

存在的问题：

多线程环境下，其他线程可能会更新已经被访问过的对象引用。造成误报，或者漏报。

Stop-the-world 安全点

暂停时间，在垃圾回收是停止其他线程的工作，造成了回收所谓的 暂停时间 GC pause 。

safepoint机制，当java虚拟机收到 stop-the-world请求，会等所有线程到达安全点，然后允许请求stop-the-world的线程进行独占的工作。

另外，一旦垃圾回收线程喊出了安全词，其他线程会停下。这个稳定的执行状态，java虚拟机的堆栈不会发声变化，所以垃圾回收能够安全的进行可达性分析。

垃圾回收的三种方式

当标记完所有的存活对象时，我们便可以进行死亡对象的回收工作了。主流的基础回收方式可分为三种,分别为：会造成内存碎片的清除、性能开销较大的压缩、以及堆使用效率较低的复制。。

第一种是**清除**（sweep），即把死亡对象所占据的内存标记为空闲内存，并记录在一个空闲列表（free list）之中。当需要新建对象时，内存管理模块便会从该空闲列表中寻找空闲内存，并划分给新建的对象。

清除这种回收方式的原理及其简单，但是有两个缺点。一是会造成内存碎片。由于 Java 虚拟机的堆中对象必须是连续分布的，因此可能出现总空闲内存足够，但是无法分配的极端情况。

另一个则是分配效率较低。如果是一块连续的内存空间，那么我们可以通过指针加法（pointer bumping）来做分配。而对于空闲列表，Java 虚拟机则需要逐个访问列表中的项，来查找能够放入新建对象的空闲内存。

第二种是**压缩**（compact），即把存活的对象聚集到内存区域的起始位置，从而留下一段连续的内存空间。这种做法能够解决内存碎片化的问题，但代价是压缩算法的性能开销。

第三种则是**复制**（copy），即把内存区域分为两等分，分别用两个指针 from 和 to 来维护，并且只是用 from 指针指向的内存区域来分配内存。当发生垃圾回收时，便把存活的对象复制到 to 指针指向的内存区域中，并且交换 from 指针和 to 指针的内容。复制这种回收方式同样能够解决内存碎片化的问题，但是它的缺点也极其明显，即堆空间的使用效率极其低下。

什么情况会出现Full GC，什么情况会出现yong GC。

Full GC:

```
1.system.gc()
    会建议JVM进行垃圾回收，不代表一定会进行GC
2.老年代空间不足
    大对象直接进入老年代、长期存活的对象进入老年代。调优建议：让对象在 Minor GC 阶段被回收、让对象在新生代多存活一段时间以及不要创建过大的对象及数组。
3.Permanet Generation（永久代）空间满（JDK 1.7 及以前）
    方法区是用永久代实现的，永久代中存放的为一些 Class 的信息、常量、静态变量等数据，当系统中要加载的类、反射的类和调用的方法较多时，永久代可能会被占满。建议：可采用的方法为增大永久代空间或转为使用 CMS GC。
4.CMS GC时出现promotion failed（担保：s1s2--old）和concurrent mode failure.(老年代采用CMS，内存足够但是不连续，当一次性分配大的数组对象时候，因为一次性分配不足，出现 conrent error 并发错误，提前触发一次full GC)
5.统计得到的Minor GC晋升到旧生代的平均大小大于旧生代的剩余空间
```

JVM内存模型

程序计数器：线程私有。是一块较小的内存，是当前线程所执行的字节码的行号指示器。是Java虚拟机规范中唯一没有规定OOM（OutOfMemoryError）的区域。

Java栈：线程私有。生命周期和线程相同。是Java方法执行的内存模型。执行每个方法都会创建一个栈帧，用于存储局部变量和操作数（对象引用）。局部变量所需要的内存空间大小在编译期间完成分配。所以栈帧的大小不会改变。存在两种异常情况：若线程请求深度大于栈的深度，抛StackOverflowError。若栈在动态扩展时无法请求足够内存，抛OOM。

本地方法栈：线程私有。与Java栈类似，但是不是为Java方法（字节码）服务，而是为本地非Java方法服务。也会抛StackOverflowError和OOM。

Java堆：所有线程共享。虚拟机启动时创建。存放对象实力和数组。所占内存最大。分为**新生代**（Young区），**老年代**（Old区）。新生代分**Eden区**，**Survivor区**。Survivor区又分为**From space区和To Space区**。Eden区和Survivor区的内存比为8:1。 当扩展内存大于可用内存，抛OOM。

方法区：所有线程共享。用于存储已被虚拟机加载的类信息、常量、静态变量等数据。又称为非堆（Non – Heap）。方法区又称“永久代”。GC很少在这个区域进行，但不代表不会回收。这个区域回收目标主要是针对常量池的回收和对类型的卸载。当内存申请大于实际可用内存，抛OOM。

Java运行时数据区

程序计数器；Java栈；本地方法栈；Java堆；方法区；

运行时常量池：方法区的一部分，类的版本、字段、方法、接口等描述信息外，还有一项是常量池（Const Pool Table），用于存放**编译期生成的各种字面量和符号引用** 。并非预置入Class文件中常量池的内容才进入方法运行时常量池，运行期间也可能将新的常量放入池中，这种特性被开发人员利用得比较多的便是String类的intern()方法。

直接内存： JDK1.4加入了NIO，引入一种基于通道与缓冲区的I/O方式，它可以使用Native函数库直接分配堆外内存，然后通过一个存储在Java堆中的**DirectByteBuffer**对象作为这块内存的引用进行操作。因为避免了在Java堆和Native堆中来回复制数据，提高了性能。

事务的实现原理

ACID

原子性

InnoDB: redo log(重做日志)和undo log(回滚日志)。其中redo log用于保证事务持久性；undo log则是事务原子性和隔离性实现的基础。实现原子性的关键，是当事务回滚时能够撤销所有已经成功执行的sql语句。 InnoDB 实现回滚，靠的是undo log ： 当事务对数据库进行修改时，InnoDB 会生成对应的undo log ； 如果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。

undo log属于逻辑日志，它记录的是sql执行相关的信息。当发生回滚时，InnoDB会根据undo log的内容做与之前相反的工作：对于每个insert，回滚时会执行delete；对于每个delete，回滚时会执行insert；对于每个update，回滚时会执行一个相反的update，把数据改回去。

以update操作为例：当事务执行update时，其生成的undo log中会包含被修改行的主键(以便知道修改了哪些行)、修改了哪些列、这些列在修改前后的值等信息，回滚时便可以使用这些信息将数据还原到update之前的状态。

一致性

可以说，一致性是事务追求的最终目标：前面提到的原子性、持久性和隔离性，都是为了保证数据库状态的一致性。此外，除了数据库层面的保障，一致性的实现也需要应用层面进行保障。

实现一致性的措施包括：

- 保证原子性、持久性和隔离性，如果这些特性无法保证，事务的一致性也无法保证
- 数据库本身提供保障，例如不允许向整形列插入字符串值、字符串长度不能超过列的限制等
- 应用层面进行保障，例如如果转账操作只扣除转账者的余额，而没有增加接收者的余额，无论数据库实现的多么完美，也无法保证状态的一致

隔离性

隔离性追求的是并发情形下事务之间互不干扰。简单起见，我们仅考虑最简单的读操作和写操作(暂时不考虑带锁读等特殊操作)，那么隔离性的探讨，主要可以分为两个方面：

- (一个事务)写操作对(另一个事务)写操作的影响：锁机制保证隔离性
- (一个事务)写操作对(另一个事务)读操作的影响：MVCC保证隔离性

锁机制

首先来看两个事务的写操作之间的相互影响。隔离性要求同一时刻只能有一个事务对数据进行写操作，InnoDB通过锁机制来保证这一点。

锁机制的基本原理可以概括为：事务在修改数据之前，需要先获得相应的锁；获得锁之后，事务便可以修改数据；该事务操作期间，这部分数据是锁定的，其他事务如果需要修改数据，需要等待当前事务提交或回滚后释放锁。

按照粒度，锁可以分为表锁、行锁以及其他位于二者之间的锁。表锁在操作数据时会锁定整张表，并发性能较差；行锁则只锁定需要操作的数据，并发性能好。但是由于加锁本身需要消耗资源(获得锁、检查锁、释放锁等都需要消耗资源)，因此在锁定数据较多情况下使用表锁可以节省大量资源。MySQL中不同的存储引擎支持的锁是不一样的，例如MyIsam只支持表锁，而InnoDB同时支持表锁和行锁，且出于性能考虑，绝大多数情况下使用的都是行锁。

脏读、不可重复读和幻读 和 事务的隔离级别

SQL标准中定义了四种隔离级别，并规定了每种隔离级别下上述几个问题是否存在。一般来说，隔离级别越低，系统开销越低，可支持的并发越高，但隔离性也越差。隔离级别与读问题的关系如下：

隔离级别	脏读	不可重复读	幻读
Read Uncommitted 读未提交	可能	可能	可能
Read Committed 读已提交	不可能	可能	可能
Repeatable Read 可重复读	不可能	不可能	可能
Serializable 可串行化	不可能	不可能	不可能

MVCC 多版本的并发控制协议

RR解决脏读、不可重复读、幻读等问题，使用的是MVCC：MVCC全称Multi-Version Concurrency Control，即多版本的并发控制协议。下面的例子很好的体现了MVCC的特点：在同一时刻，不同的事务读取到的数据可能是不同的(即多版本)——在T5时刻，事务A和事务C可以读取到不同版本的数据。

时间	事务 A	事务 B	事务 C
T1	开始事务	开始事务	开始事务
T2	查询 zhangsan 的余额，结果为 100		
T3		修改 zhangsan 的余额，将余额由 100 改为 200	
T4		提交事务	
T5	查询 zhangsan 的余额，结果为 100		查询 zhangsan 的余额，结果为 200

MVCC最大的优点是读不加锁，因此读写不冲突，并发性能好。InnoDB实现MVCC，多个版本的数据可以共存，主要是依靠数据的隐藏列(也可以称之为标记位)和undo log。其中数据的隐藏列包括了该行数据的版本号、删除时间、指向undo log的指针等等；当读取数据时，MySQL可以通过隐藏列判断是否需要回滚并找到回滚需要的undo log，从而实现MVCC；隐藏列的详细格式不再展开。

持久性

InnoDB作为MySQL的存储引擎，数据是存放在磁盘中的，但如果每次读写数据都需要磁盘IO，效率会很低。为此，InnoDB提供了缓存(Buffer Pool)，Buffer Pool中包含了磁盘中部分数据页的映射，作为访问数据库的缓冲：当从数据库读取数据时，会首先从Buffer Pool中读取，如果Buffer Pool中没有，则从磁盘读取后放入Buffer Pool；当向数据库写入数据时，会首先写入Buffer Pool，Buffer Pool中修改的数据会定期刷新到磁盘中（这一过程称为刷脏）。

Buffer Pool的使用大大提高了读写数据的效率，但是也带了新的问题：如果MySQL宕机，而此时Buffer Pool中修改的数据还没有刷新到磁盘，就会导致数据的丢失，事务的持久性无法保证。

redo log被引入来解决这个问题：当数据修改时，除了修改Buffer Pool中的数据，还会在redo log记录这次操作；当事务提交时，会调用fsync接口对redo log进行刷盘。如果MySQL宕机，重启时可以读取redo log中的数据，对数据库进行恢复。redo log采用的是WAL（Write-ahead logging，**预写式日志**），所有修改先写入日志，再更新到Buffer Pool，保证了数据不会因MySQL宕机而丢失，从而满足了持久性要求。

写redo log 比直接修改数据库本身要快的原因：

（1）刷脏是随机IO，因为每次修改的**数据位置随机**，但写redo log是**追加操作**，属于顺序IO。

（2）刷脏是**以数据页（Page）为单位**的，MySQL默认页大小是16KB，一个Page上一个小修改都要整页写入；而redo log中只包含真正需要写入的部分，**无效IO大大减少**。

binlog 和 rodo log

（1）作用不同：redo log是用于crash recovery的，保证MySQL宕机也不会影响持久性；binlog是用于point-in-time recovery的，保证服务器可以基于时间点恢复数据，此外binlog还用于主从复制。

（2）层次不同：redo log是InnoDB存储引擎实现的，而binlog是MySQL的服务器层(可以参考文章前面对MySQL逻辑架构的介绍)实现的，同时支持InnoDB和其他存储引擎。

（3）内容不同：redo log是物理日志，内容基于磁盘的Page；binlog的内容是二进制的，根据binlog_format参数的不同，可能基于sql语句、基于数据本身或者二者的混合。

（4）写入时机不同：binlog在事务提交时写入；redo log的写入时机相对多元：

- 前面曾提到：当事务提交时会调用fsync对redo log进行刷盘；这是默认情况下的策略，修改innodb_flush_log_at_trx_commit参数可以改变该策略，但事务的持久性将无法保证。
- 除了事务提交时，还有其他刷盘时机：如master thread每秒刷盘一次redo log等，这样的好处是不一定要等到commit时刷盘，commit速度大大加快。

性能分析和调优

jstack

```
-l 长列表. 打印关于锁的附加信息,例如属于java.util.concurrent 的 ownable synchronizers列表.

-F 当’jstack [-l] pid’没有相应的时候强制打印栈信息

-m 打印java和native c/c++框架的所有栈信息.

-h | -help 打印帮助信息
```