JAVA 基础篇，面试必问的那些事

让程序性能优异的并发利器

线程池

创建参数和对工作机制的影响

线程池的创建各个参数含义

public ThreadPoolExecutor(int corePoolSize,int maximumPoolSize,long keepAliveTime,TimeUnit unit,BlockingQueue<Runnable> workQueue,ThreadFactory threadFactory,RejectedExecutionHandler handler)

corePoolSize

线程池中的核心线程数，当提交一个任务时，线程池创建一个新线程执行任务，直到当前线程数等于 corePoolSize；

如果当前线程数为 corePoolSize，继续提交的任务被保存到阻塞队列中，等待被执行；

如果执行了线程池的 prestartAllCoreThreads()方法，线程池会提前创建并启动所有核心线程。

maximumPoolSize

线程池中允许的最大线程数。如果当前阻塞队列满了，且继续提交任务，则创建新的线程执行任务，前提是当前线程数小于 maximumPoolSize

keepAliveTime

线程空闲时的存活时间，即当线程没有任务执行时，继续存活的时间。默认情况下，该参数只在线程数大于 corePoolSize 时才有用

TimeUnit

keepAliveTime 的时间单位

workQueue

workQueue 必须是 BlockingQueue 阻塞队列。当线程池中的线程数超过它的 corePoolSize 的时候，线程会进入阻塞队列进行阻塞等待。通过 workQueue，线程池实现了阻塞功能

workQueue

用于保存等待执行的任务的阻塞队列，一般来说，我们应该尽量使用有界队列，因为使用无界队列作为工作队列会对线程池带来如下影响。

1）当线程池中的线程数达到 corePoolSize 后，新任务将在无界队列中等待，因此线程池中的线程数不会超过 corePoolSize。

2）由于 1，使用无界队列时 maximumPoolSize 将是一个无效参数。

3）由于 1 和 2，使用无界队列时 keepAliveTime 将是一个无效参数。

4）更重要的，使用无界 queue 可能会耗尽系统资源，有界队列则有助于防止资源耗尽，同时即使使用有界队列，也要尽量控制队列的大小在一个合适的范围。

所以我们一般会使用，ArrayBlockingQueue、LinkedBlockingQueue、SynchronousQueue、PriorityBlockingQueue。

threadFactory

创建线程的工厂，通过自定义的线程工厂可以给每个新建的线程设置一个具有识别度的线程名，当然还可以更加自由的对线程做更多的设置，比如设置所有的线程为守护线程。

Executors 静态工厂里默认的 threadFactory，线程的命名规则是“pool-数字-thread-数字”。

RejectedExecutionHandler

线程池的饱和策略，当阻塞队列满了，且没有空闲的工作线程，如果继续提交任务，必须采取一种策略处理该任务，线程池提供了 4 种策略：

（1）AbortPolicy：直接抛出异常，默认策略；

（2）CallerRunsPolicy：用调用者所在的线程来执行任务；

（3）DiscardOldestPolicy：丢弃阻塞队列中靠最前的任务，并执行当前任务；

（4）DiscardPolicy：直接丢弃任务；

当然也可以根据应用场景实现 RejectedExecutionHandler 接口，自定义饱和策略，如记录日志或持久化存储不能处理的任务。

合理地配置线程池

要想合理地配置线程池，就必须首先分析任务特性

要想合理地配置线程池，就必须首先分析任务特性，可以从以下几个角度来分析。

•任务的性质：CPU 密集型任务、IO 密集型任务和混合型任务。

•任务的优先级：高、中和低。

•任务的执行时间：长、中和短。

•任务的依赖性：是否依赖其他系统资源，如数据库连接。

性质不同的任务可以用不同规模的线程池分开处理。

CPU 密集型任务应配置尽可能小的线程，如配置 Ncpu+1 个线程的线程池。由于 IO 密集型任务线程并不是一直在执行任务，则应配置尽可能多的线程，如 2*Ncpu。

混合型的任务，如果可以拆分，将其拆分成一个 CPU 密集型任务和一个 IO 密集型任务，只要这两个任务执行的时间相差不是太大，那么分解后执行的吞吐量将高于串行执行的吞吐量。如果这两个任务执行时间相差太大，则没必要进行分解。可以通过 Runtime.getRuntime().availableProcessors()方法获得当前设备的 CPU 个数。

请概述 ConcurrentHashMap

基本概述

ConcurrentHashMap 是线程安全的 Map，在 1.7 和 1.8 中实现有所不同。

JDK 1.7 中，采用分段锁的机制，实现并发的更新操作，底层采用数组+链表的存储结构，包括两个核心静态内部类 Segment 和 HashEntry。

①、Segment 继承 ReentrantLock（重入锁） 用来充当锁的角色，每个 Segment 对象守护每个散列映射表的若干个桶；

②、HashEntry 用来封装映射表的键-值对；

③、每个桶是由若干个 HashEntry 对象链接起来的链表。

JDK 1.8 中，采用 Node + CAS + Synchronized 来保证并发安全。取消类 Segment，直接用 table 数组存储键值对；当 HashEntry 对象组成的链表长度超过 TREEIFY_THRESHOLD 时，链表转换为红黑树，提升性能。底层变更为数组 + 链表 + 红黑树。

在 1.8 里

①、重要的常量：

private transient volatile int sizeCtl;

当为负数时，-1 表示正在初始化，-N 表示 N - 1 个线程正在进行扩容；

当为 0 时，表示 table 还没有初始化；

当为其他正数时，表示初始化或者下一次进行扩容的大小。

②、数据结构：

Node 是存储结构的基本单元，实现了 Map 中的 Entry 接口，用于存储数据；

TreeNode 继承 Node，但是数据结构换成了二叉树结构，是红黑树的存储结构，用于红黑树中存储数据；

TreeBin 是封装 TreeNode 的容器，提供转换红黑树的一些条件和锁的控制。

③、存储对象时（put() 方法）：

1.如果没有初始化，就调用 initTable() 方法来进行初始化；

2.如果没有 hash 冲突就直接 CAS 无锁插入；

3.如果需要扩容，就先进行扩容；

4.如果存在 hash 冲突，就加锁来保证线程安全，两种情况：一种是链表形式就直接遍历到尾端插入，一种是红黑树就按照红黑树结构插入；

5.如果该链表的数量大于阀值 8，就要先转换成红黑树的结构

6.如果添加成功就调用 addCount() 方法统计 size，并且检查是否需要扩容。

④、扩容方法 transfer()：默认容量为 16，扩容时，容量变为原来的两倍。

helpTransfer()：调用多个工作线程一起帮助进行扩容，这样的效率就会更高。

⑤、获取对象时（get()方法）：

1.计算 hash 值，定位到该 table 索引位置，如果是首结点符合就返回；

2.如果遇到扩容时，会调用标记正在扩容结点 ForwardingNode.find()方法，查找该结点，匹配就返回；

3.以上都不符合的话，就往下遍历结点，匹配就返回，否则最后就返回 null。

附加：为什么 hashmap1.8 不直接使用红黑树而还要保留链表

因为插入时红黑树需要进行左旋，右旋操作，而单链表不需要，在数量较少时，红黑树并没有表现出比链表更好的查询效率，而且在占用空间上，红黑树的节点比链表的节点更大，是链表的两倍。

附加：为什么大于 8 个的时候才转换红黑树

按照 JDK 源码里解释是这样的：

TreeNodes 占用空间是普通 Nodes 的两倍，所以只有当 bin 包含足够多的节点时才会转成 TreeNodes，而是否足够多就是由 TREEIFY_THRESHOLD 的值决定的。当 bin 中节点数变少时，又会转成普通的 bin。TREEIFY_THRESHOLD 的值是个空间和时间的权衡。

当 hashCode 离散性很好的时候，树型 bin 用到的概率非常小，因为数据均匀分布在每个 bin 中，几乎不会有 bin 中链表长度会达到阈值。

但是在随机 hashCode 下，离散性可能会变差，然而 JDK 又不能阻止用户实现这种不好的 hash 算法，因此就可能导致不均匀的数据分布。

不过理想情况下随机 hashCode 算法下所有 bin 中节点的分布频率会遵循泊松分布，

一个 bin 中链表长度达到 8 个元素的概率为 0.00000006，几乎是不可能事件。所以，之所以选择 8，不是拍拍屁股决定的，而是根据概率统计决定的。

当然网上也有这样的说法：

红黑树的平均查找长度是 log(n)，如果长度为 8，平均查找长度为 log(8)=3，链表的平均查找长度为 n/2，当长度为 8 时，平均查找长度为 8/2=4，这才有转换成树的必要；链表长度如果是小于等于 6，6/2=3，而 log(6)=2.6，虽然速度也很快的，但是转化为树结构和生成树的时间并不会太短。

请概述 volatile

volatile 关键字的作用主要有两点：

多线程主要围绕可见性和原子性两个特性而展开，使用 volatile 关键字修饰的变量，**保证了其在多线程之间的可见性**，即每次读取到 volatile 变量，一定是最新的数据。但是 volatile 不能保证操作的原子，对任意单个 volatile 变量的读/写具有原子性，但类似于++这种复合操作不具有原子性。。

代码底层在执行时为了获取更好的性能会对指令进行重排序，多线程下可能会出现一些意想不到的问题。**使用 volatile 则会对禁止重排序**，当然这也一定程度上降低了代码执行效率。

同时在内存语义上，当写一个 volatile 变量时，JMM 会把该线程对应的本地内存中的共享变量值刷新到主内存，当读一个 volatile 变量时，JMM 会把该线程对应的本地内存置为无效。线程接下来将从主内存中读取共享变量。

在 Java 中对于 volatile 修饰的变量，编译器在生成字节码时，会在指令序列中插入内存屏障来禁止特定类型的处理器重排序问题、强制刷新和读取。

在具体实现上，volatile 关键字修饰的变量会存在一个“lock:”的前缀。它不是一种内存屏障，但是它能完成类似内存屏障的功能。Lock 会对 CPU 总线和高速缓存加锁，可以理解为 CPU 指令级的一种锁。

同时该指令会将当前处理器缓存行的数据直接写会到系统内存中，且这个写回内存的操作会使在其他 CPU 里缓存了该地址的数据无效。

请概述 AQS

是用来构建锁或者其他同步组件的基础框架，比如 ReentrantLock、ReentrantReadWriteLock 和 CountDownLatch 就是基于 AQS 实现的。

它使用了一个 int 成员变量表示同步状态，通过内置的 FIFO 队列来完成资源获取线程的排队工作。它是 CLH 队列锁的一种变体实现。它可以实现 2 种同步方式：独占式，共享式。

AQS 的主要使用方式是继承，子类通过继承 AQS 并实现它的抽象方法来管理同步状态，同步器的设计基于模板方法模式，所以如果要实现我们自己的同步工具类就需要覆盖其中几个可重写的方法，如 tryAcquire、tryReleaseShared 等等。

这样设计的目的是同步组件（比如锁）是面向使用者的，它定义了使用者与同步组件交互的接口（比如可以允许两个线程并行访问），隐藏了实现细节；同步器面向的是锁的实现者，它简化了锁的实现方式，屏蔽了同步状态管理、线程的排队、等待与唤醒等底层操作。这样就很好地隔离了使用者和实现者所需关注的领域。

在内部，AQS 维护一个共享资源 state，通过内置的 FIFO 来完成获取资源线程的排队工作。该队列由一个一个的 Node 结点组成，每个 Node 结点维护一个 prev 引用和 next 引用，分别指向自己的前驱和后继结点，构成一个双端双向链表。

同时与 Condition 相关的等待队列，节点类型也是 Node，构成一个单向链表。

synchronized 的实现原理

Synchronized 在 JVM 里的实现都是基于进入和退出 Monitor 对象来实现方法同步和代码块同步，虽然具体实现细节不一样，但是都可以通过成对的 MonitorEnter 和 MonitorExit 指令来实现。

对同步块，MonitorEnter 指令插入在同步代码块的开始位置，当代码执行到该指令时，将会尝试获取该对象 Monitor 的所有权，即尝试获得该对象的锁，而 monitorExit 指令则插入在方法结束处和异常处，JVM 保证每个 MonitorEnter 必须有对应的 MonitorExit。

对同步方法，从同步方法反编译的结果来看，方法的同步并没有通过指令 monitorenter 和 monitorexit 来实现，相对于普通方法，其常量池中多了 ACC_SYNCHRONIZED 标示符。

JVM 就是根据该标示符来实现方法的同步的：当方法被调用时，调用指令将会检查方法的 ACC_SYNCHRONIZED 访问标志是否被设置，如果设置了，执行线程将先获取 monitor，获取成功之后才能执行方法体，方法执行完后再释放 monitor。在方法执行期间，其他任何线程都无法再获得同一个 monitor 对象。

synchronized 使用的锁是存放在 Java 对象头里面，具体位置是对象头里面的 MarkWord，MarkWord 里默认数据是存储对象的 HashCode 等信息，但是会随着对象的运行改变而发生变化，不同的锁状态对应着不同的记录存储方式。在具体优化上，从 1.6 开始引入了偏向锁、自旋锁等机制提升性能。

什么是 CAS 操作，缺点是什么？

CAS 的基本思路就是，如果这个地址上的值和期望的值相等，则给其赋予新值，否则不做任何事儿，但是要返回原值是多少。每一个 CAS 操作过程都包含三个运算符：一个内存地址 V，一个期望的值 A 和一个新值 B，操作的时候如果这个地址上存放的值等于这个期望的值 A，则将地址上的值赋为新值 B，否则不做任何操作。

CAS 缺点：

ABA 问题：

比如说一个线程 one 从内存位置 V 中取出 A，这时候另一个线程 two 也从内存中取出 A，并且 two 进行了一些操作变成了 B，然后 two 又将 V 位置的数据变成 A，这时候线程 one 进行 CAS 操作发现内存中仍然是 A，然后 one 操作成功。尽管线程 one 的 CAS 操作成功，但可能存在潜藏的问题。从 Java1.5 开始 JDK 的 atomic 包里提供了一个类 AtomicStampedReference 来解决 ABA 问题。

循环时间长开销大：

对于资源竞争严重（线程冲突严重）的情况，CAS 自旋的概率会比较大，从而浪费更多的 CPU 资源，效率低于 synchronized。

只能保证一个共享变量的原子操作：

当对一个共享变量执行操作时，我们可以使用循环 CAS 的方式来保证原子操作，但是对多个共享变量操作时，循环 CAS 就无法保证操作的原子性，这个时候就可以用锁。

性能的奠基之石，SQL 优化

Mysql 索引类型和区别

普通索引：即一个索引只包含单个列，一个表可以有多个单列索引
唯一索引：索引列的值必须唯一，但允许有空值
复合索引：即一个索引包含多个列
聚集索引（聚簇索引）：innodb,数据和索引放到一起
非聚集索引：myisam，数据和索引文件分开存放

事务的四大特性

如果一个数据库声称支持事务的操作，那么该数据库必须要具备以下四个特性：

⑴ 原子性（Atomicity）

原子性是指事务包含的所有操作要么全部成功，要么全部失败回滚，这和前面两篇博客介绍事务的功能是一样的概念，因此事务的操作如果成功就必须要完全应用到数据库，如果操作失败则不能对数据库有任何影响。

⑵ 一致性（Consistency）

一致性是指事务必须使数据库从一个一致性状态变换到另一个一致性状态，也就是说一个事务执行之前和执行之后都必须处于一致性状态。

拿转账来说，假设用户 A 和用户 B 两者的钱加起来一共是 5000，那么不管 A 和 B 之间如何转账，转几次账，事务结束后两个用户的钱相加起来应该还得是 5000，这就是事务的一致性。

⑶ 隔离性（Isolation）

隔离性是当多个用户并发访问数据库时，比如操作同一张表时，数据库为每一个用户开启的事务，不能被其他事务的操作所干扰，多个并发事务之间要相互隔离。

即要达到这么一种效果：对于任意两个并发的事务 T1 和 T2，在事务 T1 看来，T2 要么在 T1 开始之前就已经结束，要么在 T1 结束之后才开始，这样每个事务都感觉不到有其他事务在并发地执行。

⑷ 持久性（Durability）

持久性是指一个事务一旦被提交了，那么对数据库中的数据的改变就是永久性的，即便是在数据库系统遇到故障的情况下也不会丢失提交事务的操作。

事务的隔离级别

不考虑事务的隔离性会发生的问题：

脏读

脏读是指在一个事务处理过程里读取了另一个未提交的事务中的数据。

当一个事务正在多次修改某个数据，而在这个事务中这多次的修改都还未提交，这时一个并发的事务来访问该数据，就会造成两个事务得到的数据不一致。例如：用户 A 向用户 B 转账 100 元，对应 SQL 命令如下

```
update account set money=money+100 where name='B';
(此时 A 通知 B)
update account set money=money - 100 where name='A';
```

当只执行第一条 SQL 时，A 通知 B 查看账户，B 发现确实钱已到账（此时即发生了脏读），而之后无论第二条 SQL 是否执行，只要该事务不提交，则所有操作都将回滚，那么当 B 以后再次查看账户时就会发现钱其实并没有转。

不可重复读

不可重复读是指在对于数据库中的某个数据，一个事务范围内多次查询却返回了不同的数据值，这是由于在查询间隔，被另一个事务修改并提交了。

例如事务 T1 在读取某一数据，而事务 T2 立马修改了这个数据并且提交事务给数据库，事务 T1 再次读取该数据就得到了不同的结果，发送了不可重复读。

不可重复读和脏读的区别是，脏读是某一事务读取了另一个事务未提交的脏数据，而不可重复读则是读取了前一事务提交的数据。

在某些情况下，不可重复读并不是问题，比如我们多次查询某个数据当然以最后查询得到的结果为主。但在另一些情况下就有可能发生问题，例如对于同一个数据 A 和 B 依次查询就可能不同，A 和 B 就可能打起来了……

虚读(幻读)

幻读是事务非独立执行时发生的一种现象。例如事务 T1 对一个表中所有的行的某个数据项做了从“1”修改为“2”的操作，这时事务 T2 又对这个表中插入了一行数据项，而这个数据项的数值还是为“1”并且提交给数据库。而操作事务 T1 的用户如果再查看刚刚修改的数据，会发现还有一行没有修改，其实这行是从事务 T2 中添加的，就好像产生幻觉一样，这就是发生了幻读。

幻读和不可重复读都是读取了另一条已经提交的事务（这点就脏读不同），所不同的是不可重复读查询的都是同一个数据项，而幻读针对的是一批数据整体（比如数据的个数）。

数据库提供的隔离级别

Read uncommitted (读未提交)：

读未提交，顾名思义，就是一个事务可以读取另一个未提交事务的数据。最低级别，任何情况都无法保证。

Read committed (读已提交)

顾名思义，就是一个事务要等另一个事务提交后才能读取数据。可避免脏读的发生，但是无法避免不可重复读。

Repeatable read (可重复读)

可重复读，就是在开始读取数据时，不再允许修改操作,可避免脏读、不可重复读的发生。但是无法无法避免幻读。

Serializable (串行化)

是最高的事务隔离级别，在该级别下，事务串行化顺序执行，可避免脏读、不可重复读、幻读的发生。就是以锁表的方式(类似于 Java 多线程中的锁)使得其他的线程只能在锁外等待，这种事务隔离级别效率低下，比较耗数据库性能，一般不使用。

以上四种隔离级别最高的是 Serializable 级别，最低的是 Read uncommitted 级别，当然级别越高，执行效率就越低。像 Serializable 这样的级别，就是以锁表的方式(类似于 Java 多线程中的锁)使得其他的线程只能在锁外等待，所以平时选用何种隔离级别应该根据实际情况。在 MySQL 数据库中默认的隔离级别为 Repeatable read (可重复读)。

在 MySQL 数据库中，支持上面四种隔离级别，默认的为 Repeatable read (可重复读)；而在 Oracle 数据库中，只支持 Serializable (串行化)级别和 Read committed (读已提交)这两种级别，其中默认的为 Read committed 级别。

MySQL 事务的实现原理

事务具有 ACID 四个特性。也即：原子性，一致性，隔离性，持久性。ACD 三个特性是通过 Redo log（重做日志）和 Undo log 实现的。 而隔离性是通过锁来实现的。

重做日志用来实现事务的持久性，即 D 特性。它由两部分组成：

①内存中的重做日志缓冲

②重做日志文件

在事务提交时，必须先将该事务的所有日志写入到 redo 日志文件中，待事务的 commit 操作完成才算整个事务操作完成。

Undo log，它可以实现如下两个功能：

1.实现事务回滚

2.实现 MVCC（多版本并发控制）

undo log 可以认为当 delete 一条记录时，undo log 中会记录一条对应的 insert 记录，反之亦然，当 update 一条记录时，它记录一条对应相反的 update 记录。

Sql 优化

常见步骤

环境方面

1、尽可能的使用高速磁盘和大内存

2、服务器使用 Linux，并且进行操作系统级别的调优，比如网络参数、避免使用 Swap 交换区等等

SQL 相关

1、先找到慢查询，慢查询日志，顾名思义，就是查询慢的日志，是指 mysql 记录所有执行超过 long_query_time 参数设定的时间阈值的 SQL 语句的日志。该日志能为 SQL 语句的优化带来很好的帮助。默认情况下，慢查询日志是关闭的，要使用慢查询日志功能，首先要开启慢查询日志功能。

- slow_query_log 启动停止技术慢查询日志
- slow_query_log_file 指定慢查询日志得存储路径及文件（默认和数据文件放一起）
- long_query_time 指定记录慢查询日志 SQL 执行时间得伐值（单位：秒，默认 10 秒）
- log_queries_not_using_indexes 是否记录未使用索引的 SQL
- log_output 日志存放的地方【TABLE】【FILE】【FILE,TABLE】

2、分析慢查询日志。慢查询的日志记录非常多，要从里面找寻一条查询慢的日志并不是很容易的事情，一般来说都需要一些工具辅助才能快速定位到需要优化的 SQL 语句，比如 Mysqldumpslow

3、SQL 本身优化，比如少用子查询，in 查询改关联查询、不使用外键与级联等等

4、反范式化设计，字段允许适当冗余、选择合适的字段存储长度等等

5、使用执行计划分析 SQL 语句，使用 EXPLAIN 关键字可以模拟优化器执行 SQL 查询语句，从而知道 MySQL 是如何处理你的 SQL 语句的。分析你的查询语句或是表结构的性能瓶颈，至少可以知道

- 表的读取顺序
- 数据读取操作的操作类型
- 哪些索引可以使用
- 哪些索引被实际使用
- 表之间的引用
- 每张表有多少行被优化器查询

比如，执行计划中的 ype 显示的是访问类型，是较为重要的一个指标，结果值从最好到最坏依次是：

system > const > eq_ref > ref > fulltext > ref_or_null > index_merge > unique_subquery > index_subquery > **range > index** > ALL

一般来说，得保证查询至少达到 range 级别，要求能达到 ref。

优化 10 大策略

策略 1.尽量全值匹配

当建立了索引列后，能在 wherel 条件中使用索引的尽量所用。

策略 2.最佳左前缀法则

如果索引了多列，要遵守最左前缀法则。指的是查询从索引的最左前列开始并且不跳过索引中的列。

策略 3.不在索引列上做任何操作

不在索引列上做任何操作（计算、函数、(自动 or 手动)类型转换），会导致索引失效而转向全表扫描

策略 4.范围条件放最后

中间有范围查询会导致后面的索引列全部失效

策略 5.覆盖索引尽量用

尽量使用覆盖索引(指一个查询语句的执行只用从索引中就能够取得，不必从数据表中读取，)，减少 select *

策略 6.不等于要慎用

mysql 在使用不等于(!= 或者<>)的时候无法使用索引会导致全表扫描，如果一定要需要使用不等于，请用覆盖索引

策略 7.Null/Not 有影响

注意 null/not null 对索引的可能影响

1、自定义为 NOT NULL

在字段为 not null 的情况下，使用 is null 或 is not null 会导致索引失效

解决方式：覆盖索引

2、自定义为 NULL 或者不定义

Is not null 的情况会导致索引失效

解决方式：覆盖索引

策略 8.Like 查询要当心

like 以通配符开头('%abc...')mysql 索引失效会变成全表扫描的操作

解决方式：覆盖索引

策略 9.字符类型加引号

字符串不加单引号索引失效
解决方式：请加引号

策略 10.OR 改 UNION 效率高

解决方式：如果一定要用 OR，那么使用覆盖索引

Java 程序员必须知道的 JVM

JVM 内存区域

JVM 在执行 Java 程序的过程中会把它管理的内存分为若干个不同的区域，这些组成部分有些是线程私有的，有些则是线程共享的

线程私有的：程序计数器，虚拟机栈，本地方法栈

线程共享的：方法区，堆

程序计数器

较小的内存空间，当前线程执行的字节码的行号指示器；各线程之间独立存储，互不影响，此内存区域是唯一一个不会出现 OutOfMemoryError 情况的区域。

虚拟机栈

每个线程私有的，线程在运行时，在执行每个方法的时候都会打包成一个栈帧，存储了局部变量表，操作数栈，动态链接，方法出口等信息，然后放入栈。每个时刻正在执行的当前方法就是虚拟机栈顶的栈桢。方法的执行就对应着栈帧在虚拟机栈中入栈和出栈的过程。

本地方法栈

各虚拟机自由实现，本地方法栈 native 方法调用 JNI 到了底层的 C/C++(c/c++可以触发汇编语言，然后驱动硬件

方法区/永久代

用于存储已经被虚拟机加载的类信息，常量("zdy","123"等)，静态变量(static 变量)等数据，比如类信息就包括类的完整有效名、返回值类型、修饰符（public，private...）、变量名、方法名、方法代码、这个类型直接父类的完整有效名(除非这个类型是 interface 或是 java.lang.Object，两种情况下都没有父类)、类的直接接口的一个有序列表等等

堆

几乎所有对象都分配在这里，也是垃圾回收发生的主要区域

JVM 垃圾回收器

JVM 中是通过可达性分析算法判断对象是否可回收的。

这个算法的基本思想就是通过一系列的称为 “GC Roots” 的对象作为起点，从这些节点开始向下搜索，节点所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的话，则证明此对象是不可用的。

在垃圾对象回收上，有几种常用算法：

1. 标记-清除算法

标记-清除算法分为“标记”和“清除”阶段：首先标记出所有需要回收的对象，在标记完成后统一回收所有被标记的对象。但是会带来两个明显的问题：

1）效率问题

2）空间问题（标记清除后会产生大量不连续的碎片）

2. 复制算法

将内存分为大小相同的两块，每次使用其中的一块。当这一块的内存使用完后，就将还存活的对象复制到另一块去，然后再把使用的空间一次清理掉。这样就使每次的内存回收都是对内存区间的一半进行回收。

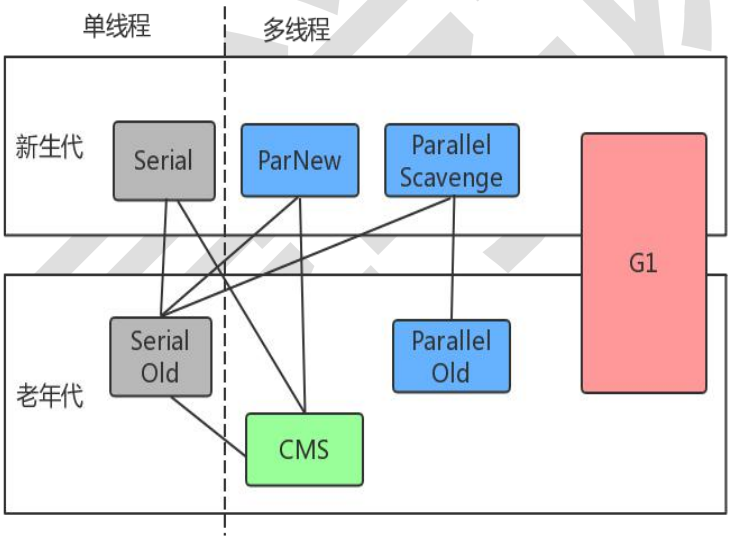
3. 标记-整理算法

根据老年代的特点特出的一种标记算法，标记过程仍然与“标记-清除”算法一样，但后续步骤不是直接对可回收对象回收，而是让所有存活的对象向一端移动，然后直接清理掉端边界以外的内存。

根据对象的生命周期，将 java 堆分为新生代和老年代，这样我们就可以根据各个年代的特点选择合适的垃圾收集算法。

比如在新生代中，每次收集都会有大量对象死去，所以可以选择复制算法，只需要付出少量对象的复制成本就可以完成每次垃圾收集。而老年代的对象存活几率是比较高的，而且没有额外的空间对它进行分配担保，所以我们必须选择“标记-清除”或“标记-整理”算法进行垃圾收集。

在具体的垃圾算法的实现上有几种垃圾回收器



Serial/Serial Old

最古老的，单线程，独占式，成熟，适合单 CPU　服务器

-XX:+UseSerialGC 新生代和老年代都用串行收集器

-XX:+UseParNewGC 新生代使用 ParNew，老年代使用 Serial Old

-XX:+UseParallelGC 新生代使用 ParallerGC，老年代使用 Serial Old

ParNew

和 Serial 基本没区别，唯一的区别：多线程，多 CPU 的，停顿时间比 Serial 少

-XX:+UseParNewGC 新生代使用 ParNew，老年代使用 Serial Old

除了性能原因外，主要是因为除了 Serial 收集器，只有它能与 CMS 收集器配合工作。

Parallel Scavenge（ParallerGC）/Parallel Old

关注吞吐量的垃圾收集器，高吞吐量则可以高效率地利用 CPU 时间，尽快完成程序的运算任务，主要适合在后台运算而不需要太多交互的任务。

所谓吞吐量就是 CPU 用于运行用户代码的时间与 CPU 总消耗时间的比值，即吞吐量=运行用户代码时间/（运行用户代码时间+垃圾收集时间），虚拟机总共运行了 100 分钟，其中垃圾收集花掉 1 分钟，那有吞吐效率就是 99%。-XX:+UseParallelOldGC 则会开启这一对组合，同时 Parallel Scavenge 还有一个自适应调节策略，就不需要手工指定新生代的大小（-Xmn）、Eden 与 Survivor 区的比例（-XX:SurvivorRatio）、晋升老年代对象年龄（-XX:PretenureSizeThreshold）等细节参数了，虚拟机会根据当前系统的运行情况收集性能监控信息，动态调整这些参数以提供最合适的停顿时间或最大的吞吐量。通过打开 -XX:+UseAdaptiveSizePolicy，只需要把基本的内存数据设置好（如-Xmx 设置最大堆），然后使用 MaxGCPauseMillis 参数（更关注最大停顿时间）或 GCTimeRatio 参数（更关注吞吐量）给虚拟机设立一个优化目标，那具体细节参数的调节工作就由虚拟机完成了。

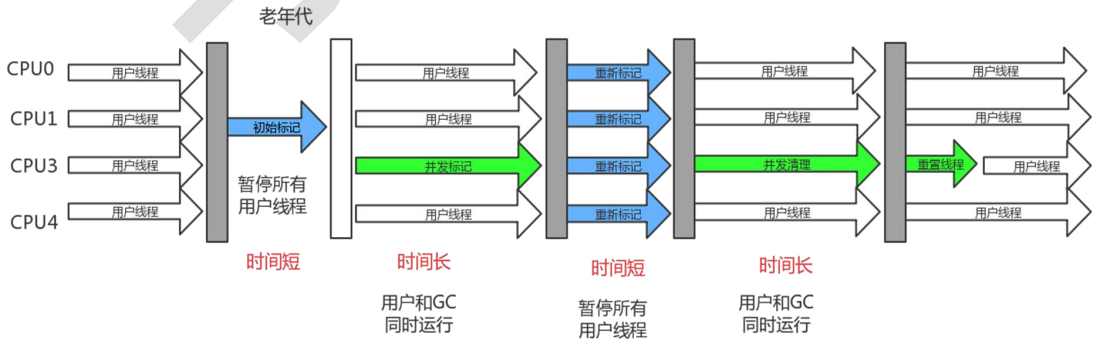
Concurrent Mark Sweep （CMS）

收集器是一种以获取最短回收停顿时间为目标的收集器。目前很大一部分的 Java 应用集中在互联网站或者 B/S 系统的服务端上，这类应用尤其重视服务的响应速度，希望系统停顿时间最短，以给用户带来较好的体验。CMS 收集器就非常符合这类应用的需求。

-XX:+UseConcMarkSweepGC ，一般新生代使用 ParNew，老年代的用 CMS，并发收集失败，转为 SerialOld 从名字（包含“Mark Sweep”）上就可以看出，CMS 收集器是基于“标记—清除”算法实现的，它的运作过程相对于前面几种收集器来说更复杂一些。

整个过程分为 4 个步骤，包括：

- **初始标记：**仅仅只是标记一下 GC Roots 能直接关联到的对象，速度很快，需要停顿（STW -Stop the world）。
- **并发标记：**从 GC Root 开始对堆中对象进行可达性分析，找到存活对象，它在整个回收过程中耗时最长，不需要停顿。
- **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，需要停顿(STW)。这个阶段的停顿时间一般会比初始标记阶段稍长一些，但远比并发标记的时间短。
- **并发清除：**不需要停顿。



优点：

由于整个过程中耗时最长的并发标记和并发清除过程收集器线程都可以与用户线程一起工作，所以，从总体上来说，CMS 收集器的内存回收过程是与用户线程一起并发执行的。

缺点：

CPU 资源敏感：因为并发阶段多线程占据 CPU 资源，如果 CPU 资源不足，效率会明显降低。

浮动垃圾：由于 CMS 并发清理阶段用户线程还在运行着，伴随程序运行自然就还会有新的垃圾不断产生，这一部分垃圾出现在标记过程之后，CMS 无法在当次收集中处理掉它们，只好留待下一次 GC 时再清理掉。这一部分垃圾就称为“浮动垃圾”。

由于浮动垃圾的存在，因此需要预留出一部分内存，意味着 CMS 收集不能像其它收集器那样等待老年代快满的时候再回收。

在 1.6 的版本中老年代空间使用率阈值(92%)

如果预留的内存不够存放浮动垃圾，就会出现 Concurrent Mode Failure，这时虚拟机将临时启用 Serial Old 来替代 CMS。

会产生空间碎片：标记 - 清除算法会导致产生不连续的空间碎片，CMS 只会删除无用对象，不会对内存做压缩，会造成内存碎片，这时候我们需要用到这个参数：

G1 垃圾回收器

主要是用在大内存和多处理器数量的服务器上。jdk9 中将 G1 变成默认的垃圾收集器。

G1 中重要的参数：

-XX:+UseG1GC 使用 G1 垃圾回收器

-XX:MaxGCPauseMillis=200 设置 GC 的最大暂停时间为 200ms

内部布局改变

G1 把堆划分成多个大小相等的独立区域（Region），每个 Region 大小为 2 的倍数，范围在 1MB-32MB 之间，可能为 1，2，4，8，16，32MB。所有的 Region 有一样的大小，JVM 生命周期内不会改变。整个堆被划分成 2048 左右个 Region。新生代和老年代不再物理隔离。Region 可以说是 G1 回收器一次回收的最小单元。

算法：标记—整理 （old，humongous） 和复制回收算法(survivor)。

Stop The World 现象

Stop the World 机制，简称 STW，主要指执行垃圾收集算法时,Java 应用程序的其他所有除了垃圾收集收集器线程之外的线程都被挂起。

此时，系统只能允许 GC 线程进行运行，其他线程则会全部暂停，等待 GC 线程执行完毕后才能再次运行。这些工作都是由虚拟机在后台自动发起和自动完成的，是在用户不可见的情况下把用户正常工作的线程全部停下来，这对于很多的应用程序，尤其是那些对于实时性要求很高的程序来说是难以接受的。我们 GC 调优的目标就是尽可能的减少 STW 的时间和次数。

JVM 中存在哪些引用？

1. 强引用

以前我们使用的大部分引用实际上都是强引用，这是使用最普遍的引用。如果一个对象具有强引用，那就类似于必不可少的生活用品，垃圾回收器绝不会回收它。当内存空 间不足，Java 虚拟机宁愿抛出 OutOfMemoryError 错误，使程序异常终止，也不会靠随意回收具有强引用的对象来解决内存不足问题。

2. 软引用（SoftReference）

如果一个对象只具有软引用，那就类似于可有可无的生活用品。如果内存空间足够，垃圾回收器就不会回收它，如果内存空间不足了，就会回收这些对象的内存。只要垃圾回收器没有回收它，该对象就可以被程序使用。软引用可用来实现内存敏感的高速缓存。

软引用可以和一个引用队列（ReferenceQueue）联合使用，如果软引用所引用的对象被垃圾回收，JAVA 虚拟机就会把这个软引用加入到与之关联的引用队列中。

3. 弱引用（WeakReference）

如果一个对象只具有弱引用，那就类似于可有可无的生活用品。弱引用与软引用的区别在于：只具有弱引用的对象拥有更短暂的生命周期。在垃圾回收器线程扫描它 所管辖的内存区域的过程中，一旦发现了只具有弱引用的对象，不管当前内存空间足够与否，都会回收它的内存。不过，由于垃圾回收器是一个优先级很低的线程， 因此不一定会很快发现那些只具有弱引用的对象。

弱引用可以和一个引用队列（ReferenceQueue）联合使用，如果弱引用所引用的对象被垃圾回收，Java 虚拟机就会把这个弱引用加入到与之关联的引用队列中。

4. 虚引用（PhantomReference）

"虚引用"顾名思义，就是形同虚设，与其他几种引用都不同，虚引用并不会决定对象的生命周期。如果一个对象仅持有虚引用，那么它就和没有任何引用一样，在任何时候都可能被垃圾回收。

虚引用主要用来跟踪对象被垃圾回收的活动。

在程序设计中除了强引用，使用软引用的情况较多，这是因为软引用可以加速 JVM 对垃圾内存的回收速度，可以维护系统的运行安全，防止内存溢出（OutOfMemory）等问题的产生

类加载机制

类从被加载到虚拟机内存中开始，到卸载出内存为止，它的整个生命周期包括：加载（Loading）、验证（Verification）、准备（Preparation）、解析（Resolution）、初始化（Initialization）、使用（Using）和卸载（Unloading）7 个阶段。其中验证、准备、解析 3 个部分统称为连接（Linking）

加载阶段

虚拟机需要完成以下 3 件事情：

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区的运行时数据结构。

3）在内存中生成一个代表这个类的 java.lang.Class 对象，作为方法区这个类的各种数据的访问入口。

验证

是连接阶段的第一步，这一阶段的目的是为了确保 Class 文件的字节流中包含的信息符合当前虚拟机的要求，并且不会危害虚拟机自身的安全。但从整体上看，验证阶段大致上会完成下面 4 个阶段的检验动作：文件格式验证、元数据验证、字节码验证、符号引用验证。

准备阶段

是正式为类变量分配内存并设置类变量初始值的阶段，这些变量所使用的内存都将在方法区中进行分配。这个阶段中有两个容易产生混淆的概念需要强调一下，首先，这时候进行内存分配的仅包括类变量（被 static 修饰的变量），而不包括实例变量，实例变量将会在对象实例化时随着对象一起分配在 Java 堆中。其次，这里所说的初始值“通常情况”下是数据类型的零值，假设一个类变量的定义为：

public static int value=123；

那变量 value 在准备阶段过后的初始值为 0 而不是 123，因为这时候尚未开始执行任何 Java 方法，而把 value 赋值为 123 的 putstatic 指令是程序被编译后，存放于类构造器＜clinit＞（）方法之中，所以把 value 赋值为 123 的动作将在后面的初始化阶段才会执行。

假设上面类变量 value 的定义变为：public static **final** int value=123；编译时 Javac 将会为 value 生成 ConstantValue 属性，在准备阶段虚拟机就会根据 ConstantValue 的设置将 value 赋值为 123。

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。

符号引用以一组符号来描述所引用的目标，符号可以是任何形式的字面量，只要使用时能无歧义地定位到目标即可。符号引用与虚拟机实现的内存布局无关，引用的目标并不一定已经加载到内存中。各种虚拟机实现的内存布局可以各不相同，但是它们能接受的符号引用必须都是一致的，因为符号引用的字面量形式明确定义在 Java 虚拟机规范的 Class 文件格式中。

直接引用可以是直接指向目标的指针、相对偏移量或是一个能间接定位到目标的句柄。直接引用是和虚拟机实现的内存布局相关的，同一个符号引用在不同虚拟机实例上翻译出来的直接引用一般不会相同。如果有了直接引用，那引用的目标必定已经在内存中存在。

初始化阶段

虚拟机规范则是严格规定了有且只有 5 种情况必须立即对类进行“初始化”（而加载、验证、准备自然需要在此之前开始）：

1）遇到 new、getstatic、putstatic 或 invokestatic 这 4 条字节码指令时，如果类没有进行过初始化，则需要先触发其初始化。生成这 4 条指令的最常见的 Java 代码场景是：使用 new 关键字实例化对象的时候、读取或设置一个类的静态字段（被 final 修饰、已在编译期把结果放入常量池的静态字段除外）的时候，以及调用一个类的静态方法的时候。

2）使用 java.lang.reflect 包的方法对类进行反射调用的时候，如果类没有进行过初始化，则需要先触发其初始化。

3）当初始化一个类的时候，如果发现其父类还没有进行过初始化，则需要先触发其父类的初始化。

4）当虚拟机启动时，用户需要指定一个要执行的主类（包含 main（）方法的那个类），虚拟机会先初始化这个主类。

5）当使用 JDK 1.7 的动态语言支持时，如果一个 java.lang.invoke.MethodHandle 实例最后的解析结果 REF_getStatic、REF_putStatic、REF_invokeStatic 的方法句柄，并且这个方法句柄所对应的类没有进行过初始化，则需要先触发其初始化。

初始化也是类加载过程的最后一步，前面的类加载过程中，除了在加载阶段用户应用程序可以通过自定义类加载器参与之外，其余动作完全由虚拟机主导和控制。到了初始化阶段，才真正开始执行类中定义的 Java 程序代码在准备阶段，变量已经赋过一次系统要求的初始值，而在初始化阶段，则根据程序员通过程序制定的主观计划去初始化类变量和其他资源，或者可以从另外一个角度来表达：初始化阶段是执行类构造器＜clinit＞（）方法的过程。＜clinit＞（）方法是由编译器自动收集类中的所有类变量的赋值动作和静态语句块（static{}块）中的语句合并产生的，编译器收集的顺序是由语句在源文件中出现的顺序所决定的。

＜clinit＞（）方法对于类或接口来说并不是必需的，如果一个类中没有静态语句块，也没有对变量的赋值操作，那么编译器可以不为这个类生成＜clinit＞（）方法。

虚拟机会保证一个类的＜clinit＞（）方法在多线程环境中被正确地加锁、同步，如果多个线程同时去初始化一个类，那么只会有一个线程去执行这个类的＜clinit＞（）方法，其他

线程都需要阻塞等待，直到活动线程执行＜clinit＞（）方法完毕。如果在一个类的＜clinit＞（）方法中有耗时很长的操作，就可能造成多个进程阻塞。所以类的初始化是线程安全的，项目中可以利用这点。

双亲委派模型

对于任意一个类，都需要由加载它的类加载器和这个类本身一同确立其在 Java 虚拟机中的唯一性.

从 Java 虚拟机的角度来讲，只存在两种不同的类加载器：

一种是启动类加载器（Bootstrap ClassLoader），这个类加载器使用 C++语言实现，是虚拟机自身的一部分；另一种就是所有其他的类加载器，这些类加载器都由 Java 语言实现，独立于虚拟机外部，并且全都继承自抽象类 java.lang.ClassLoader。

启动类加载器（Bootstrap ClassLoader）：这个类将器负责将存放在＜JAVA_HOME＞\lib 目录中的，或者被-Xbootclasspath 参数所指定的路径中的，并且是虚拟机识别的（仅按照文件名识别，如 rt.jar，名字不符合的类库即使放在 lib 目录中也不会被加载）类库加载到虚拟机内存中。启动类加载器无法被 Java 程序直接引用，用户在编写自定义类加载器时，如果需要把加载请求委派给引导类加载器，那直接使用 null 代替即可。

扩展类加载器（Extension ClassLoader）：这个加载器由 sun.misc.Launcher$ExtClassLoader 实现，它负责加载＜JAVA_HOME＞\lib\ext 目录中的，或者被 java.ext.dirs 系统变量所指定的路径中的所有类库，开发者可以直接使用扩展类加载器。

应用程序类加载器（Application ClassLoader）：这个类加载器由 sun.misc.Launcher $App-ClassLoader 实现。由于这个类加载器是 ClassLoader 中的 getSystemClassLoader（）方法的返回值，所以一般也称它为系统类加载器。它负责加载用户类路径（ClassPath）上所指定的类库，开发者可以直接使用这个类加载器，如果应用程序中没有自定义过自己的类加载器，一般情况下这个就是程序中默认的类加载器。

我们的应用程序都是由这 3 种类加载器互相配合进行加载的，如果有必要，还可以加入自己定义的类加载器。

双亲委派模型要求除了顶层的启动类加载器外，其余的类加载器都应当有自己的父类加载器。这里类加载器之间的父子关系一般不会以继承（Inheritance）的关系来实现，而是都使用组合（Composition）关系来复用父加载器的代码。

使用双亲委派模型来组织类加载器之间的关系，有一个显而易见的好处就是 Java 类随着它的类加载器一起具备了一种带有优先级的层次关系。例如类 java.lang.Object，它存放在 rt.jar 之中，无论哪一个类加载器要加载这个类，最终都是委派给处于模型最顶端的启动类加载器进行加载，因此 Object 类在程序的各种类加载器环境中都是同一个类。相反，如果没有使用双亲委派模型，由各个类加载器自行去加载的话，如果用户自己编写了一个称为 java.lang.Object 的类，并放在程序的 ClassPath 中，那系统中将会出现多个不同的 Object 类，Java 类型体系中最基础的行为也就无法保证，应用程序也将会变得一片混乱。

ClassLoader 中的 loadClass 方法中的代码逻辑就是双亲委派模型：

在自定义 ClassLoader 的子类时候，我们常见的会有两种做法，一种是重写 **loadClass** 方法，另一种是重写 **findClass** 方法。其实这两种方法本质上差不多，毕竟 loadClass 也会调用 findClass，但是从逻辑上讲我们最好不要直接修改 loadClass 的内部逻辑。我建议的做法是只在 findClass 里重写自定义类的加载方法。

loadClass 这个方法是实现双亲委托模型逻辑的地方，擅自修改这个方法会导致模型被破坏，容易造成问题。因此我们最好是在双亲委托模型框架内进行小范围的改动，不破坏原有

的稳定结构。同时，也避免了自己重写 loadClass 方法的过程中必须写双亲委托的重复代码，从代码的复用性来看，不直接修改这个方法始终是比较好的选择。
但是 Tomcat 中没有完全遵守双亲委派模型。

双亲委派模型的破坏

双亲委派很好地解决了各个类加载器的基础类的统一问题(越基础的类由越上层的加载器进行加载)，基础类之所以称为“基础”，是因为它们总是作为被用户代码调用的 API，但世事往往没有绝对的完美，如果基础类又要调用回用户的代码，那该怎么办？

比如 JDBC 原因是原生的 JDBC 中 Driver 驱动本身只是一个接口，并没有具体的实现，具体的实现是由不同数据库类型去实现的。

例如，MySQL 的 mysql-connector-.jar 中的 Driver 类具体实现的。 原生的 JDBC 中的类是放在 rt.jar 包的，是由启动类加载器进行类加载的，在 JDBC 中的 Driver 类中需要动态去加载不同数据库类型的 Driver 类，而 mysql-connector-.jar 中的 Driver 类是由独立厂商实现并部署在应用程序的 ClassPath 下的，那启动类加载器肯定是不能进行加载的，既然是自己编写的代码，那就需要由应用程序启动类去进行类加载。

于是，这个时候就引入线程上下文件类加载器(Thread Context ClassLoader)。有了这个东西之后，程序就可以把原本需要由启动类加载器进行加载的类，由应用程序类加载器去进行加载了。如果创建线程时还未设置，它将会从父线程中继承一个，如果在应用程序的全局范围内都没有设置过的话，那这个类加载器默认就是应用程序类加载器。

Java 中所有涉及 SPI 的加载动作基本上都采用这种方式，例如 JNDI、JDBC、JCE、JAXB 和 JBI 等。

双亲委派模型的 “被破坏”是由于用户对程序动态性的追求而导致的，这里所说的“动态性”指的是当前一些非常“热门”的名词：代码热替换（HotSwap）、模块热部署（HotDeployment）等等。

JVM 常用工具

jps

列出当前机器上正在运行的虚拟机进程，JPS 从操作系统的临时目录上去找（所以有一些信息可能显示不全）。

-q　:仅仅显示进程，
-m:输出主函数传入的参数. 下的 hello 就是在执行程序时从命令行输入的参数
-l: 输出应用程序主类完整 package 名称或 jar 完整名称.
-v: 列出 jvm 参数, -Xms20m -Xmx50m 是启动程序指定的 jvm 参数

jstat

是用于监视虚拟机各种运行状态信息的命令行工具。它可以显示本地或者远程虚拟机进程中的类装载、内存、垃圾收集、JIT 编译等运行数据，在没有 GUI 图形界面，只提供了纯文本控制台环境的服务器上，它将是运行期定位虚拟机性能问题的首选工具。

假设需要每 250 毫秒查询一次进程 13616 垃圾收集状况，一共查询 10 次，那命令应当是：jstat-gc 13616　250 10

常用参数：
-class (类加载器)
-compiler (JIT)
-gc (GC 堆状态)

-gccapacity (各区大小)
-gccause (最近一次 GC 统计和原因)
-gcnew (新区统计)
-gcnewcapacity (新区大小)
-gcold (老区统计)
-gcoldcapacity (老区大小)
-gcpermcapacity (永久区大小)
-gcutil (GC 统计汇总)
-printcompilation (HotSpot 编译统计)

jinfo

查看和修改虚拟机的参数

jinfo －sysprops 可以查看由 System.getProperties()取得的参数

jinfo －flag 未被显式指定的参数的系统默认值

jinfo －flags（注意 s）显示虚拟机的参数

jinfo －flag +[参数] 可以增加参数，但是仅限于由 java -XX:+PrintFlagsFinal －version 查询出来且为 manageable 的参数

jinfo －flag -[参数] pid 可以修改参数

Thread.getAllStackTraces();

jmap

用于生成堆转储快照（一般称为 heapdump 或 dump 文件）。jmap 的作用并不仅仅是为了获取 dump 文件，它还可以查询 finalize 执行队列、Java 堆和永久代的详细信息，如空间使用率、当前用的是哪种收集器等。和 jinfo 命令一样，jmap 有不少功能在 Windows 平台下都是受限的，除了生成 dump 文件的-dump 选项和用于查看每个类的实例、空间占用统计的-histo 选项在所有操作系统都提供之外，其余选项都只能在 Linux/Solaris 下使用。

jmap -dump:live,format=b,file=heap.bin <pid>

Sun JDK 提供 jhat（JVM Heap Analysis Tool）命令与 jmap 搭配使用，来分析 jmap 生成的堆转储快照。

jhat

jhat dump 文件名

后屏幕显示“Server is ready.”的提示后，用户在浏览器中键入 http://localhost：7000/就可以访问详情

使用 jhat 可以在服务器上生成堆转储文件分析（一般不推荐，毕竟占用服务器的资源，比如一个文件就有 1 个 G 的话就需要大约吃一个 1G 的资源）

jstack

（Stack Trace for Java）命令用于生成虚拟机当前时刻的线程快照。线程快照就是当前虚拟机内每一条线程正在执行的方法堆栈的集合，生成线程快照的主要目的是定位线程出现长时间停顿的原因，如线程间死锁、死循环、请求外部资源导致的长时间等待等都是导致线程长时间停顿的常见原因。

在代码中可以用 java.lang.Thread 类的 getAllStackTraces（）方法用于获取虚拟机中所有线程的 StackTraceElement 对象。使用这个方法可以通过简单的几行代码就完成 jstack 的大部分功能，在实际项目中不妨调用这个方法做个管理员页面，可以随时使用浏览器来查看线程堆栈。（并发编程中的线程安全课程中有具体的案例）

项目内存或者 cpu 占用率过高如何排查

一、在排查问题的过程中针对 CPU 的问题，使用以下命令组合来排查问题

1、查看问题进程，得到进程 PID：

top -c

2、查看进程里的线程明细，并手动记下 CPU 异常的线程 PID：

top -p PID -H

3、使用 jdk 提供 jstack 命令打印出项目堆栈：

jstack pid > xxx.log

线程 PID 转成 16 进制，与堆栈中的 nid 对应，定位问题代码位置。

二、针对内存问题，使用以下命令组合来排查问题：

1、查看内存中的存活对象统计，找出业务相关的类名：

jmap -histo:live PID > xxx.log

2、通过简单的统计还是没法定位问题的话，就输出内存明细来分析。这个命令会将内存里的所有信息都输出，输出的文件大小和内存大小基本一致。而且会导致应用暂时挂起，所以谨慎使用。

jmap -dump:live,format=b,file=xxx.hprof PID

3、 最后对 dump 出来的文件进行分析。文件大小不是很大的话，使用 jdk 自带的 jhat 命令即可：

jhat -J-mx2G -port 7170

4、dump 文件太大的话，可以使用 jprofiler 工具来分析。jprofiler 工具的使用，这里不做详细介绍，有兴趣可以搜索一下。

三、需要分析 GC 情况，可以使用以下命令：jstat -gc PID

框架源码，CRUD 和高级程序员的分水岭

谈谈依赖注入和面向切面

类似的面试题：谈谈你对 Spring 框架的理解、谈谈 Spring 中的 IOC 和 AOP 概念等等

spring 框架是一个开源而轻量级的框架，是一个 IOC 和 AOP 容器，spring 的核心就是控制反转（IOC）和面向切面编程（AOP）

控制反转（IOC）：是面向对象编程中的一种设计原则，用来降低程序代码之间的耦合度，使整个程序体系结构更加灵活，与此同时将类的创建和依赖关系写在配置文件里，由配置文件注入，达到松耦合的效果。与此同时 IOC 也称为 DI（依赖注入），依赖注入是一种开发模式；依赖注入提倡使用接口编程； 依赖注入使得可以开发各个组件，然后根据组件之间的依赖关系注入组装。

所谓依赖，从程序的角度看，就是比如 A 要调用 B 的方法，那么 A 就依赖于 B，反正 A 要用到 B，则 A 依赖于 B。所谓倒置，你必须理解如果不倒置，会怎么着，因为 A 必须要有 B，才可以调用 B，如果不倒置，意思就是 A 主动获取 B 的实例：B b = new B()，这就是最简单的获取 B 实例的方法（当然还有各种设计模式可以帮助你去获得 B 的实例，比如工厂、Locator 等等），然后你就可以调用 b 对象了。所以，不倒置，意味着 A 要主动获取 B，才

能使用 B；到了这里，就应该明白了倒置的意思了。倒置就是 A 要调用 B 的话，A 并不需要主动获取 B，而是由其它人自动将 B 送上门来。

形象的举例就是：

通常情况下，假如你有一天在家里口渴了，要喝水，那么你可以到你小区的小卖部去，告诉他们，你需要一瓶水，然后小卖部给你一瓶水！这本来没有太大问题，关键是如果小卖部很远，那么你必须知道：从你家如何到小卖部；小卖部里是否有你需要的水；你还要考虑是否开着车去；等等等等，也许有太多的问题要考虑了。也就是说，为了一瓶水，你还可能需要依赖于车等等这些交通工具或别的工具，问题是不是变得复杂了？那么如何解决这个问题呢？

解决这个问题的方法很简单：小卖部提供送货上门服务，凡是小卖部的会员，你只要告知小卖部你需要什么，小卖部将主动把货物给你送上门来！这样一来，你只需要做两件事情，你就可以活得更加轻松自在：

第一：向小卖部注册为会员。

第二：告诉小卖部你需要什么。

这和 Spring 的做法很类似！Spring 就是小卖部，你就是 A 对象，水就是 B 对象

第一：在 Spring 中声明一个类：A

第二：告诉 Spring，A 需要 B

面向切面编程（AOP）将安全，事务等于程序逻辑相对独立的功能抽取出来，利用 Spring 的配置文件将这些功能插进去，实现了按照切面编程，提高了复用性；最主要的作用：可以在不修改源代码的情况下，给目标方法动态添加功能。

面向切面编程的目标就是分离关注点。什么是关注点呢？就是你要做的事，就是关注点。假如你是个公子哥，没啥人生目标，天天就是衣来伸手，饭来张口，整天只知道玩一件事！那么，每天你一睁眼，就光想着吃完饭就去玩（你必须要做的事），但是在玩之前，你还需要穿衣服、穿鞋子、叠好被子、做饭等等等等事情，这些事情就是你的关注点，但是你只想吃饭然后玩，那么怎么办呢？这些事情通通交给别人去干。在你走到饭桌之前，有一个专门的仆人 A 帮你穿衣服，仆人 B 帮你穿鞋子，仆人 C 帮你叠好被子，仆人 C 帮你做饭，然后你就开始吃饭、去玩（这就是你一天的正事），你干完你的正事之后，回来，然后一系列仆人又开始帮你干这个干那个，然后一天就结束了！

AOP 的好处就是你只需要干你的正事，其它事情别人帮你干。也许有一天，你想裸奔，不想穿衣服，那么你把仆人 A 解雇就是了！也许有一天，出门之前你还想带点钱，那么你再雇一个仆人 D 专门帮你干取钱的活！这就是 AOP。每个人各司其职，灵活组合，达到一种可配置的、可插拔的程序结构。

从 Spring 的角度看，AOP 最大的用途就在于提供了事务管理的能力。事务管理就是一个关注点，你的正事就是去访问数据库，而你不想管事务（太烦），所以，Spring 在你访问数据库之前，自动帮你开启事务，当你访问数据库结束之后，自动帮你提交/回滚事务！

Spring 优点：a:低侵入式设计，独立于各种应用服务器，b:依赖注入特点性将组件关系透明化，降低耦合度 c:与第三方框架具有良好的整合效果

解释 Spring 框架中 bean 实例化的流程

Spring 容器 从 XML 文件中读取 bean 的定义，并实例化 bean。
Spring 根据 bean 的定义填充所有的属性。
如果 bean 实现了 BeanNameAware 接口，Spring 传递 bean 的 ID 到 setBeanName 方法。
如果 Bean 实现了 BeanFactoryAware 接口， Spring 传递 beanfactory 给 setBeanFactory 方法。
如果有任何与 bean 相关联的 BeanPostProcessors，Spring 会在

postProcesserBeforeInitialization()方法内调用它们。
如果 bean 实现 IntializingBean 了，调用它的 afterPropertySet 方法，
如果 bean 声明了初始化方法，调用此初始化方法。
如果有 BeanPostProcessors 和 bean 关联，这些 bean 的 postProcessAfterInitialization() 方法将被调用。

Spring Bean 的生命周期

无非就是在 bean 实例化的流程的基础之上，增加了 Spring 容器销毁 Bean 的过程，包括了执行有@PreDestroy 注解的方法，然后是

Spring 在 Bean 创建过程中是如何解决循环依赖的?

循环依赖只会存在在单例实例中，多例循环依赖直接报错。
A 类实例化后，把实例放 map 容器中，A 类中有一个 B 类属性，A 类实例化要进行 IOC 依赖注入，这时候 B 类需要实例化，B 类实例化跟 A 类一样，实例化后方 map 容器中。B 类中有一个 A 类属性，接着 B 类的 IOC 过程，又去实例化 A 类，这时候实例化 A 类过程中从 map 容器发现 A 类已经在容器中了，就直接返回了 A 的实例，依赖注入到 B 类中 A 属性中，B 类 IOC 完成后，B 实例化就完全完成了，就返回给 A 类的 IOC 过程。这就是循环依赖的解决。

AOP 实现流程

1、aop:config 自定义标签解析
2、自定义标签解析时封装对应的 aop 入口类，类的类型就是 BeanPostProcessor 接口类型
3、Bean 实例化过程中会执行到 aop 入口类中
4、在 aop 入口类中，判断当前正在实例化的类是否在 pointcut 中，pointcut 可以理解为一个模糊匹配，是一个 joinpoint 的集合
5、如果当前正在实例化的类在 pointcut 中，则返回该 bean 的代理类，同时把所有配置的 advice 封装成 MethodInterceptor 对象加入到容器中，封装成一个过滤器链
6、代理对象调用，jdk 动态代理会调到 invocationHandler 中，cglib 型代理调到 MethodInterceptor 的 callback 类中，然后在 invoke 方法中执行过滤器链。

Spring 框架中如何基于 AOP 实现的事务管理?

事务管理，是一个切面。在 aop 环节中，其他环节都一样，事务管理就是由 Spring 提供的 advice，既是 TransactionInterceptor，它一样的会在过滤器链中被执行到，这个 TransactionInterceptor 过滤器类是通过解析<tx:advice>自定义标签得到的。

描述一下 SpringMvc 的整个访问或者调用流程。

1：发起请求到前端控制器(DispatcherServlet)
2：前端控制器请求 HandlerMapping 查找 Handler（ 可以根据 xml 配置、注解进行查找），处理器映射器 HandlerMapping 向前端控制器返回 Handler
3：前端控制器调用处理器适配器去执行 Handler
4：处理器适配器去执行 Handler
5：Handler 执行完成给适配器返回 ModelAndView,处理器适配器向前端控制器返回 ModelAndView（ModelAndView 是 springmvc 框架的一个底层对象，包括 Model 和 view）
6：前端控制器请求视图解析器去进行视图解析（根据逻辑视图名解析成真正的视图(jsp)），视图解析器向前端控制器返回 View
7:前端控制器进行视图渲染（ 视图渲染将模型数据(在 ModelAndView 对象中)填充到 request 域）
8：前端控制器向用户响应结果