一、深浅拷贝(对象)

```
Cloneable:CloneNotSupportedException
```

只有子类实现了Cloneable接口后才可以使用Object类提供的clone方法。

```
protected native Object clone() throws CloneNotSupportedExceptio
n;
```

要想让对象具有拷贝功能，必须实现Cloneable接口 标识接口，表示此类允许被clone，并且在类中自定义clone调用Object类提供的继承权限clone方法

1.1浅拷贝：对象值拷贝

对于浅拷贝而言，拷贝出来的对象仍然保留原有对象的所有引用。

> **对于基本类型而言：只是基本类型的值**
> **对于引用类型而言：只是值（地址） 栈上创建一个变量指向同一块堆内存**

问题：牵一发而动全身
只要任意一个拷贝对象（或原对象）中的引用发生改变，所有对象均会收到影响。（同时改会出现什么问题） 类似于线程共享资源？？

1.2深拷贝：新对象

拷贝出来的对象产生了所有引用的新的引用。
特点：修改任意一个对象，不会对其他对象产生影响。

1.3如何实现深拷贝

1.包含的其他类继续实现Cloneable接口，并且调用clone方法（递归调用）
2.使用序列化（***）**

使用序列化进行深拷贝时，无序再实现Cloneable接口，只需实现Serializable接口
特点：修改任意一个对象，不会影响其他对象。

1.3延迟拷贝：浅拷贝+深拷贝

二、JVM简单

虚拟机：VMWare/Virtual Box
通过软件模拟的具有完整硬件功能，运行再完全隔离环境中的计算机系统。
（通过软件模拟的一个系统）

JVM是通过软件模拟Java字节码的指令集,JVM只保存了PC寄存器，而普遍的虚拟机有很多的寄存器。

2000年JDK1.3：HotSpot 作为默认虚拟机发布至今为默认虚拟机。
不同虚拟机是实现不同。

Scala Kotto均可在上面运行

三、Java内存划分-共6块内存（bulingbuling）

3.1线程私有内存：（linux线程内容）

线程私有内存，每个线程都有，彼此之间完全隔离

3.1.1程序计数器

程序计数器是比较小的内存空间，当前线程所执行的字节码的行号指示器。
若当前线程执行的是java方法，计数器记录的是正在执行的JVM字节指令地址；
若当前线程执行的是native方法，计数器数值为空。

程序计数器是唯一块不会产生OOM异常的区域。

3.1.2虚拟机栈

-Xss设置栈容量

虚拟机描述java方法执行的内存模型，每个方法执行的同时都会创建一个栈帧来存储局部变量表，操作数栈，方法出口等信息。每个方法从调用到执行完毕的过程，对应一个栈帧的入栈出栈过程。

生命周期与线程相同：在创建线程时同时创建线程的虚拟机栈，线程执行结束，虚拟机栈与线程一同被回收。

此区域一共会产生两种异常：
1.若线程请求的栈深度大于JVM允许的深度（-Xss设置栈容量），抛出StackOverFlowErro异常。（单线程常常产生）
2.虚拟机正在进行栈的动态扩容，若无法申请到足够内存，抛出OutOfMemoryErro异常（多线程）

3.1.3本地方法栈

本地方法（native）执行的内存模型。

HotSpot虚拟机中，本地方法栈与虚拟机栈时同一块内存区域

3.2线程共享内存：

所有线程共享此内存空间，并且此内存空间对所有线程可见。

3.2.1堆（GC堆）--对象，数组

Java(堆 Java Heap)是JVM管理的最大内存区域。所有**线程共享此内存**，此内存中存放的都是对象实例以及数组

Java堆是垃圾回收器管理的最主要内存区域。java堆可以处于物理上不连续的内存空间。

-Xmx设置堆最大值

-Xms设置堆的最小值

如果在堆中没有足够的内存完成对象实例分配，并且堆无法再次扩展时抛出OOM异常。（最频繁发生！）

OOM:

内存溢出：内存中的对象确实还应该存活，但由于堆内存不够用而产生的异常。

内存泄漏：无用对象无法被GC 扩展内存后还是会报错OOM异常

3.2.2方法区（静态）（线程共享）

用于存储已被**JVM以加载的类信息，常量，静态变量等数据**，JDK8以前，方法区也叫永久代，JDK8之后称为元空间（Mate Space）之前所说的静态常量区

类定义的信息在方法区放着。

public static ……（符号引用）也正在方法区放着

对象产生：符号引用->类->具体引用

执行Test test = new Test();

首先知道是那个类（存放方法区）

最后的具体引用->符号引用（从方法区中取出）

JVM规范：方法区无法满足内存分配需求时抛出,OOM

3.2.3运行时常量池（方法区的一部分）

运行时常量池方法区的一部分，存放字面量与符号引用。

字面量：字符串常量（**JDK1.7移到堆中**），final常量，**基本数据类型值。**

符号引用：类、字段、方法的完全限定名，名称，描述符。（**类和结构的完全限定名、字段的名称和描述符、方法的名称和描述符。）**

完全限定名：包.类名 权限修饰符。
字面量：10（基本数据类型）

3.3Java堆溢

4.、垃圾回收器与内存分配策略

4.1如何判断对象是否已死

4.1.1引用计数法

算法思想：
给每个对象附加一个引用计数器，每当有一个地方引用此对象时，计数器+1，每当有一个引用失效时，计数器-1；在任意时刻，只要计数器值为0的对象就是不能再被使用的，即对象已死。

引用计数法实现简单，判定效率也比较高，在大部分情况下都是一个不错的算法。比如Python语言就采用引用计 数法进行内存管理。

但是在主流的JVM中没有选用引用计数法来管理内存，**最主要的原因就是引用计数法无法解决对象的循环引用问题。 我中有你你中有我。**
JVM参数 :-XX:+PrintGC ： 打印JVM垃圾回收信息。

4.1.2 可达性分析算法

上面我们讲了，java并不采用引用计数法来判断对象是否已 "死",而采用“可达性分析法”来判断对象是否存活（同样采用此法的还有C#、Lisp-早的一门采用动态内存分配的语言)。
核心思想：通过一系列“GC Roots的对象作为起点。从这些点向下搜索对象。搜索走过的路径，称为”引用链“，当一个对象到任意一个GC Roots对象没有任何的引用链相连时（从GC Roots到对象不可达），证明此对象已死。

Java中能作为GC Roots的对象包含以下四种：
1.虚拟机栈中引用的对象（方法中的对象）

2.类静态变量引用的对象（静态变量）　---局部变量（类的普通对象）除外
3.常量引用的对象　（常量）
4.本地方法栈中引用的对象
(在JDK1.2以前，Java中引用的定义很传统：如果引用类型的数据中存储的数值代表的是另一块内存的起始地址，就 称这块内存代表着一个引用。**这种定义有些狭隘**，一个对象在这种定义下**只有被引用或者没有被引用两种状态。)**

我们希望能描述这一类对象：当内存空间还足够时，则能保存在内存中；如果内存空间在进行垃圾回收后还是非常 紧张，则可以抛弃这些对象。很多系统中的缓存对象都符合这样的场景。

**********在JDK1.2之后，Java对引用的概念做了扩充**
将引用分为强引用(Strong Reference)
软引用(Soft Reference)、
弱引用(Weak Reference)和
虚引用(Phantom Reference)四种，这四种引用的强度依次递减。

1.强引用：强引用指的是代码中普遍存在的，类似Object obj = new Object();[直接new出来的]
在JVM中只要强引用还存在，**垃圾回收器永远不会回收此对象实例。**

2.软引用：用来表述一些有用但不必须的对象。对于仅被软引用指向的对象，**在系统将要发生内存溢出之前，会将所以软引用对象进行垃圾回收。若内存够用，这些对象仍然保留。**在JDk1.2之后提供SoftReference来实现软引用。

3.弱引用：　弱引用也是用来描述非必需对象的。弱引用**强度比软引用更差一点。**仅被弱引用关联的对象最多**只能生存到下一次GC开始之前。**当垃圾回收器开始工作时，无论当前内存是否够用。**都会回收掉仅被弱引用关联的对象。**JDK1.2之后，使用WeakReference来实现弱引用。

4.虚引用：也被称为叫做幽灵引用，或者幻影引用，是最弱的一种引用关系。一个对象是否有虚引用的存在，完全不会对其生存时间产生影响。也无法通过虚引用来取得一个对象实例，**为一个对象设置虚引用的唯一目的就是能在这个对象被回收器时收到一个系统通知。在JDK1.2之后，提供了PhantomReference类来实现虚引用。（仅仅是一个通知，GC)**

4.2对象自我拯救（已经过期了）

```
protected void finalize() throws Throwable { }
```

在可达性分析算法中不可达的对象，也并非”非死不可“，所有不可达的对象处于”缓刑“阶段

要宣告已一个对象的彻底死亡需要经历两次标记过程。
1.第一次标记：若对象在进行可达性分析之后发现到GC Roots不可达，此对象会进行一次finalize()。筛选的条件是此对象是否有必要执行finalize()方法，**当对象没有覆盖finalize()方法或者finalize()方法已经被JVM调用过（不是我们自己调用的），那么JVM会对此对象彻底宣告死亡。**

筛选成功（对象覆写了finalize()方法，并且未被执行过），会将此对象放到F-Queue(唤醒队列),如果对象在
finalize()中成功自救（此对象与GC roots建立联系），则对象会在第二次标记时被移除回收集合，成功存活；若对象在finalize()中任然与GC Roots不可达，宣告死亡。

4.3回收方法区

方法区（永久代）的垃圾回收主要回收的是两部分：
1.废弃的常量
2.无用的类

回收废弃常量和回收Java堆中的对象十分类似。以常量池中字面量(直接量)的回收为例，假如一个字符串"abc"已经 进入了常量池中，但是当前系统没有任何一个String对象引用常量池的"abc"常量，也没有在其他地方引用这个字面 量，如果此时发生GC并且有必要的话（方法区（不够用了才会执行）），这个"abc"常量会被系统清理出常量池。常量池中的其他类(接口)、方法、 字段的符号引用也与此类似。

判断一个类是无用类，必须同时满足以下三个条件
1.该类的所有实例都已经被回收（JAVA堆中不存在该类的任何实现）
2.加载该类的类加载器已经被回收
3.该类的Class对象没有在任何其他地方被引用，也无法通过反射访问该类的所有内容。

JVM可以对同时满足上述3个条件的无用类进行回收，也仅仅是"可以"而不是必然。在大量使用反射、动态代理等场景都需要JVM具备类卸载的功能来防止永久代的溢出。

4.5垃圾回收算法

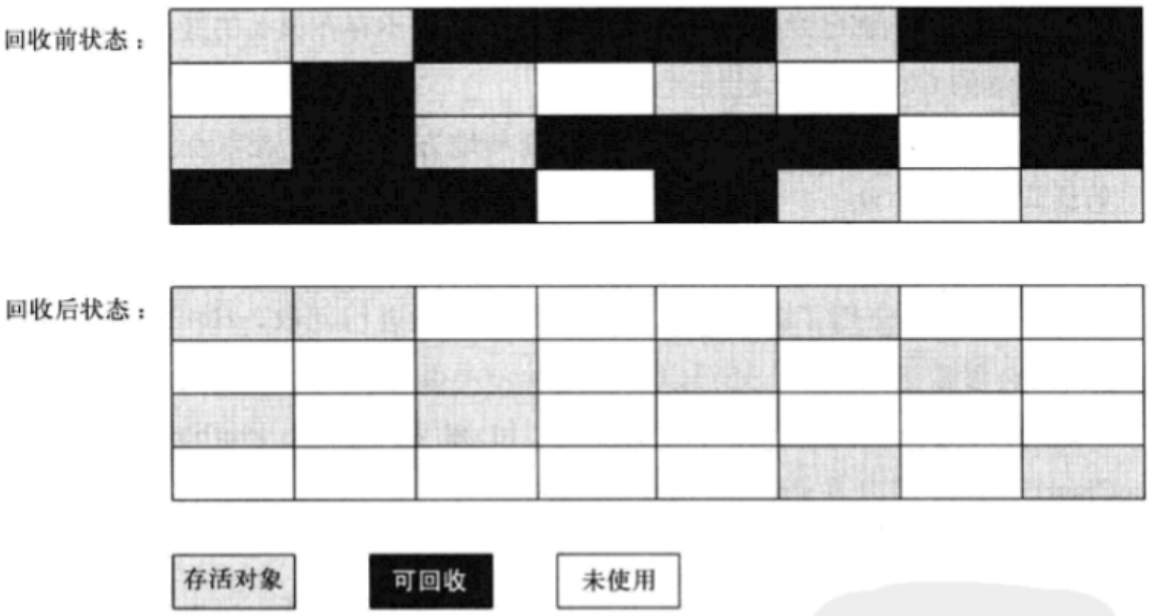
4.5.1标记-清除算法

- **算法核心思想**，

整个算法分为标记清除-两个阶段，
标记阶段标记出此垃圾回收需要的回收对象。清除阶段一次性清除所有带标记的对象。

- **导致的问题：Java不采用标记清除算法**

标记与清除两个阶段效率都不高，空间问题，标记清除算法会产生大量的不连续空间，导致的问题是，若程序中需要分配较大连续对象时，由于空间碎片较多时因此无法找到连续的内存空间而不得再触发GC。



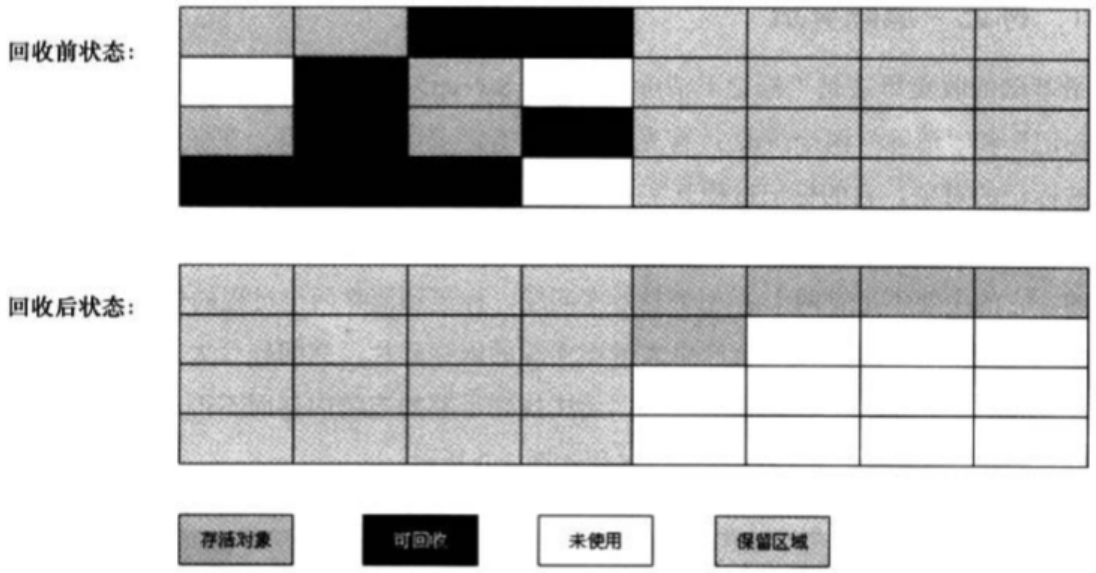
4.6复制算法（新生代垃圾回收算法）

- **核心思想：**

将内存按容量划分成大小相等的两块，每次只使用其中一块内存，当使用的内存需要进行垃圾回收时，会将此区域的所有存活的对象一次性复制到保留区域。然后将使用的内存块一次清理掉。

- **导致问题：**

复制算法最大的问题在于内存利用率太低，所用的商用JVM都对复制算法进行了改进。

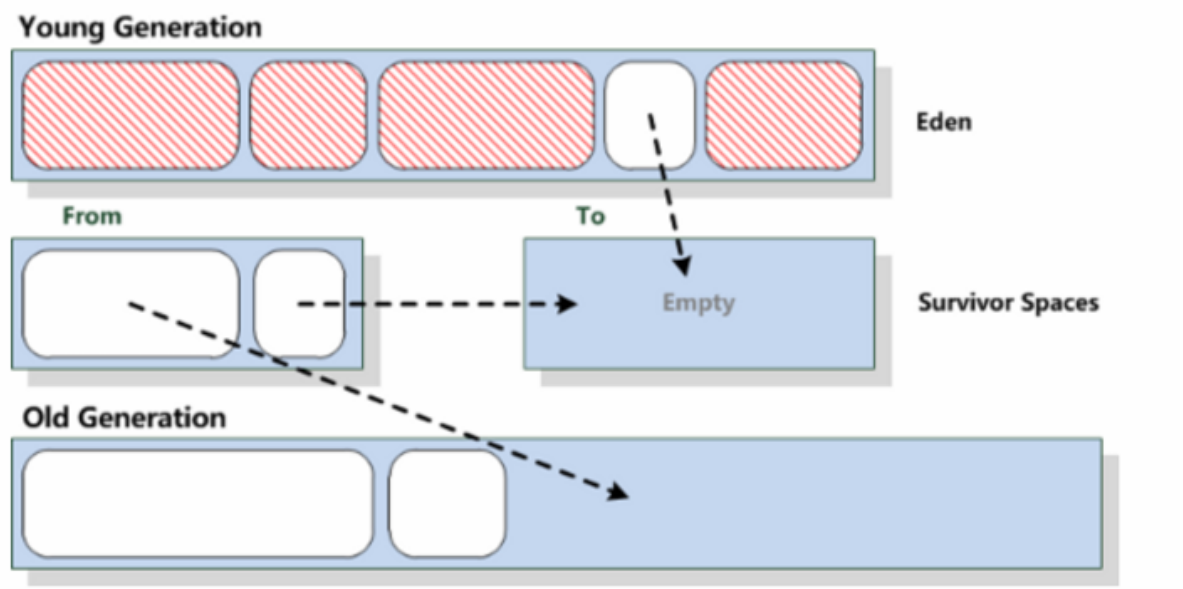


JVM改进后的复制算法：

新生代中98%的对象”朝生夕死“（存活时间非常短），所以并不需要按照1：1来划分内存空间。将内存（新生代内存）分为一块较大的Eden（伊甸园）
和两块较小的Survivor(幸存者)（一块称为From区，另一块称为To区）空间。每次只使用Eden区和其中一块Survivor。

HootSpot复制算法的流程：
内存区域划分不变，**执行流程：**
1.当Eden区满的时候，会触发第一次Minor GC，将所有存活对象拷贝到Survivor的From区域中，然后一次性清除Eden区；
2.当Eden区再次触发Minor GC，会扫描Eden区和From区，将这两块空间中的存活对象拷贝到To区，而后一次性清空Eden区和From区。
3.当后续Eden区再次发生MinorGC时，会对Eden和To区进行垃圾回收，存活对象移动到From区，后续流程类似，只是将From和To区来回作为保留区。

4.部分对象会在From与To区中来回复制，如此交换15次（MaxTenuringThreshold,默认为15）最终会存入老年代。 **（父债子偿）**
5.Survivor区域若无法放下所有存活对像，需要依赖其他内存，如老年代内存进行分配担保。



4.3标记整理算法（老年代垃圾回收算法）

为什么老年代不用复制算法，因为老年代不适用复制算法。（对象存活率较高，需要大量进行对象的复制，效率很低）

算法思想：

1.算法分为标记与整理两个阶段，标记过程与标记清除的过程一致。只是在整理的过程不一样。整理过程需要让所有存活对象向一段移动，而后直接清理掉存活对象边界以外的内存。

4.4分代回收算法（Java）******

了解JVM的分代回收算法：

算法核心思想，根据**对象的存活周期**将**内存**划分为以下两块，**新生代**、**老年代**。

新生代：每次GC都有大批对象死去，只有少量存活，因此采用复制算法；

老年代：对象存活率较高，没有额外空间对其分配担保，采用标记-整理算法；

面试题：minor GC ,Full GC

1.MinorGC称为新生代GC；指的是发生在新生代的垃圾回收。由于新生代对象大多存活周期较短，因此我们的MinorGC发生频率非常频繁，一般回收速度也较高。

2.FullGC称为老年代的垃圾回收或者MajorGC：指的实际发生在老年代的垃圾回收。出现了MajorGC通常会伴随至少一次的Minor GC（并非绝对），在Parallel Scavenge收集器中就有直接进行Full GC的策略选择过程)。 Major GC的速度一般会比Minor GC慢10倍以上。

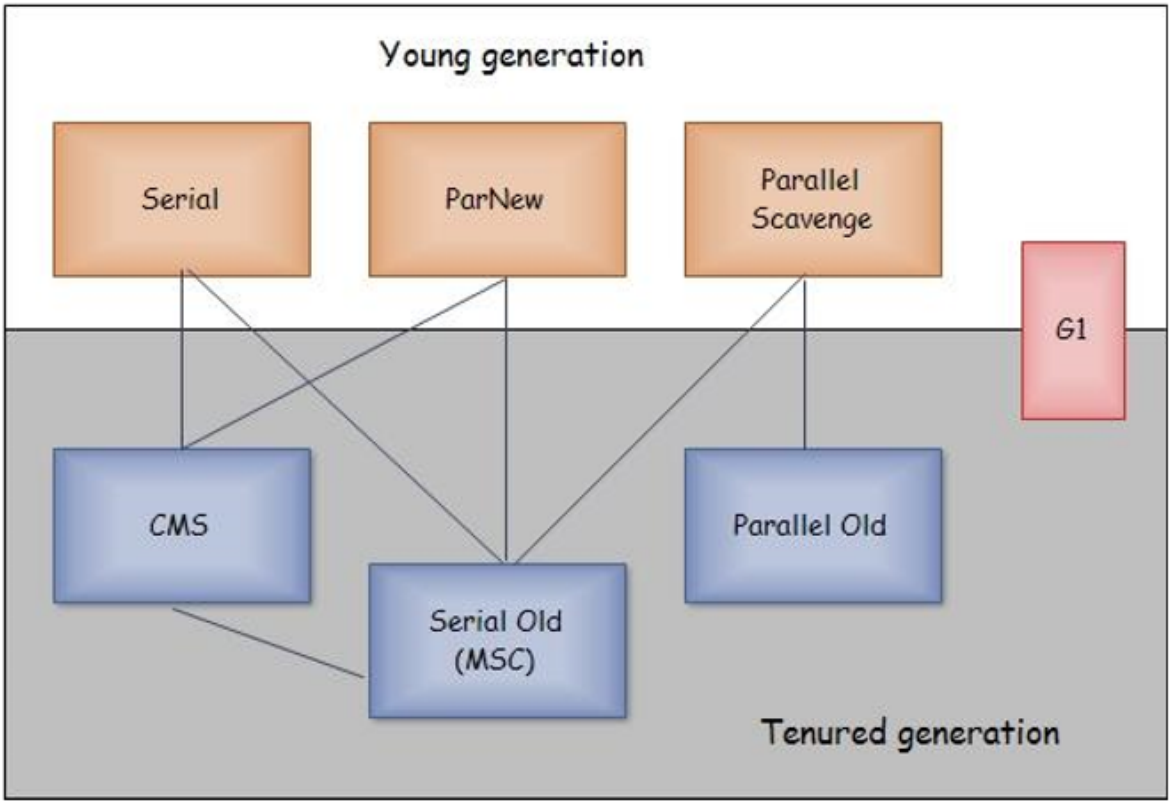
特例：大对象直接仍到老年代中，假如老年代也满了，就直接进行老年代的垃圾回收

5.垃圾收集器（根据JDK版本不同实现不同）—JDK8

(了解一下即可，JVM调优部分)
垃圾回收器也分两块（）

新生代垃圾回收器：Serial（串行收集器）、ParNew（并行）、 Parallel（并行）、Scavenge（并行）

老年代垃圾回收器：CMS 、Serial old、Parallel old
串行（Serial 、Serial old）：垃圾回收线程与用户线程在JVM中顺序执行（其中一个执行，都在等待）。
并行(Parallel、Scavenge、Parallel old）：多个垃圾回收线程一起执行，用户线程仍处于等待。
并发（CMS）：这才是真正意义上的垃圾回收线程与用户线程一起执行。CMS是第一款垃圾回收器
全区域垃圾回收器：G1(并发)
STW:当垃圾回收线程工作，用户线程处于等待状态



区域划分->判断对象是否存活->垃圾回收器->对象分配

5.对象分配策略

5.1对象优先在Eden区分配(垃圾回收效率高)

大多数情况下，对象在新生代的Eden区分配，当Eden区没有足够分配空间时，JVM发生一次Minor GC

-Xss:栈的大小

-Xms:堆的最小内存

-Xmx:堆的最大内存

-Xmn:新生代内存大小

-XX:+survivorRatio=8 (比率) -默认是8

5.2大对象直接进入老年代

设置大对象的标准：

-XX:PretenureSizeThreadhold = 字节大小（告诉JVM超过该字节大小的对象为大对象，直接进入老年代）

5.3长期存活对象进入老年代

JVM给堆中的每个对象定义了一个堆中年龄的计数器

若对象在Eden出生并且经历了一次MinorGC后任然存活并且能被Survivor容纳，将此对象的年龄置为1，此后对象再Survior区域经历一次MinorGC，年龄就增加一岁，当其年龄增加到一定程度（默认为15），此对象晋升到老年代

-XX:MaxTenuringThreshold

5.4动态年龄判定

动态年龄低于

JVM并不是永远要求对象的年龄必须达到MaxTenuringThreshold才能晋升老年代若Surivor空间中相同年龄的所有对象大小的总和大于Surivor空间（From 或To 区）的一半，此时年龄大于等于该年龄的**所有对象直接进入老年代。**无需等待到**MaxTenuringThreshold要求的年龄。**

6JDK命令行工具

6.1**********jps：JVM进程状态工具**********

列出正在运行的JVM进程，并且返回进程ID

常用的参数有以下： -q -l

jsp - l:输出主类全名，返回进程ID。

6.2jstat：JVM统计信息监视工具

显示本地或远程JVM中类装载，内存，垃圾回收等数据

jstat - gcutil PID:显示垃圾回收信息

6.3jinfo：JVM配置信息查看工具

jinfo - flags PID ： 查看JVM配置信息

6.4jmap：内存映像工具（查看Java堆具体信息）

jmap:用于生成堆转储快照（堆的快照）

jmap -heap PID：显示JVM堆的具体信息

jmap -histo PID：显示JVM中对象的统计信息

6.5jhat：虚拟机转存储快照分析工具

jhat(JVM Heap Analysis Tool)命令与jmap命令搭配使用，用于分析jmap生成的堆转储快照，

6.6********jstack Java堆栈跟踪工具 *********

jstack生成当前JVM线程快照，可用于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问题，当线程出现停顿时 就可以用jstack各个线程调用的堆栈情况

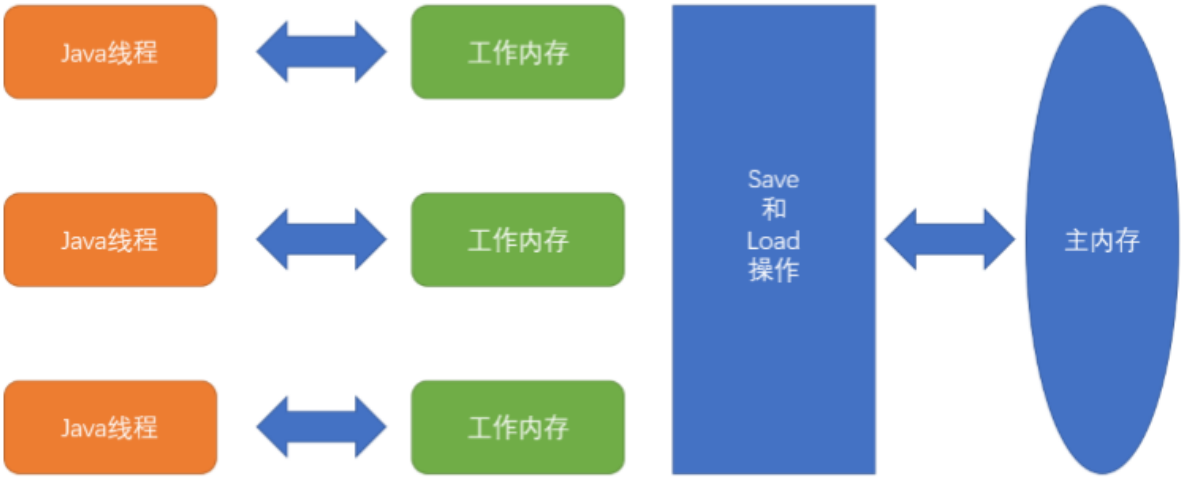
命令名称	全称	用途
jps	JVM Process Status Tool	显示指定系统内所有的HotSpot虚拟机进程
jstat	JVM Statistics Monitoring Tool	用于收集Hotspot虚拟机各方面的运行数据
jinfo	Configuration Info for Java	显示虚拟机配置信息
jmap	JVM Memory Map	生成虚拟机的内存转储快照，生成heapdump文件
jhat	JVM Heap Dump Browser	用于分析heapdump文件，它会建立一个HTTP/HTML服务器，让用户在浏览器上查看分析结果
jstack	JVM Stack Trace	显示虚拟机的线程快照

7.Java内存模型-基于线程的模型

JVM定义的主要目标为了定义程序中各个变量的访问规则
（JVM如何将变量从内存中取出以及如何将变量再写回内存等细节。此处的变量包括实例字段，静态字段与数组元素（线程共享资源）。）

7.1主内存与工作内存

工作内存：每个线程独有。
主内存：所有线程共享内存。（堆（大部分），方法区，运行时常量池）
JVM规定所有变量必须放在主内存中。每条线程都有自己的工作内存，线程的**工作内**存中保存了该线程使用到的**变量的主内存副本。**线程对变量的**所有操作（读取，赋值等）**都必须在**工作内存中进行**，不能直接读写主内存变量。不同的线程之间也**无法直接访问彼此的工作内存变量，线程间变量值的传递均需要通过主内存来完成。**



主内存和工作内存中的资源有延迟，不是最新的

5.2内存见交互操作

关于主内存与工作内存之间的具体交互协议，即一个变量如何从主内存中拷贝到工作内存、如何从工作内存同步回 主内存之类的实现细节，Java内存模型中定义了如下8种操作来完成。JVM实现时必须保证下面提及的每一种操作的 原子的、不可再分的。

lock(锁定)：作用于主内存的变量，它把一个变量标识为一条线程独占的状态

unlock(解锁)：作用于主内存的变量，它把一个处于锁定状态的变量释放出来，释放后的变量才可以被其他线 程锁定。

read(读取)：作用于主内存的变量，它把一个变量的值从主内存传输到线程的工作内存中，以便随后的load动 作使用。

load(载入)：作用于工作内存的变量，它把read操作从主内存中得到的变量值放入工作内存的变量副本中。**use(使用)：**作用于工作内存的变量，它把工作内存中一个变量的值传递给执行引擎。

assign(赋值)：作用于工作内存的变量，它把一个从执行引擎接收到的值赋给工作内存的变量。

store(存储)：作用于工作内存的变量，它把工作内存中一个变量的值传送到主内存中，以便后续的write操作 使用。

write(写入)：作用于主内存的变量，它把store操作从工作内存中得到的变量的值放入主内存的变量中。

5.3内存三大特性

1. 原子性： 基本数据类型的访问读写是具备原子性的。如若需要更大范围的原子性，需要内键锁或Lock体系的支持（i++,i--等操作），由Java内存模型来直接保证的原子性变量操

作包括read、load、assign、use、store和read。大致可 以认为，基本数据类型的访问读写是具备原子性的。如若需要更大范围的原子性，需要synchronized关键字 约束。(即一个操作或者多个操作 要么全部执行并且执行的过程不会被任何因素打断，要么就都不执行)
2.可见性：可见性是指当一个线程修改了共享变量的值，其他线程能够立即得知这个修改。volatile、 synchronized、final三个关键字可以实现可见性。

3.有序性：如果在本线程内观察所有操作都是有序的，若在线程外观察另外一个线程所有操作都是无序的。
JMM具备先天的有序性，即无序通过任何手段就能保证的有序性。**称为JMM的happens-before原则。**若两个操作的次序无法从happens-before中推导出来，则无法保证其有序性，JVM可以随意对其进行重排序。

下面就来具体介绍下happens-before原则（先行发生原则）：
下面就来具体介绍下happens-before原则（先行发生原则）：
程序次序规则：一个线程内，按照代码顺序，书写在前面的操作先行发生于书写在后面的操作
定规则：一个unLock操作先行发生于后面对同一个锁的lock操作
volatile变量规则：对一个变量的写操作先行发生于后面对这个变量的读操作
传递规则：如果操作A先行发生于操作B，而操作B又先行发生于操作C，则可以得出操作A先行发生于操作C 线程启动规则：Thread对象的start()方法先行发生于此线程的每个一个动作 线程中断规则：对线程interrupt()方法的调用先行发生于被中断线程的代码检测到中断事件的发生
线程终结规则：线程中所有的操作都先行发生于线程的终止检测，我们可以通过Thread.join()方法结束、 Thread.isAlive()的返回值手段检测到线程已经终止执行
对象终结规则：一个对象的初始化完成先行发生于他的finalize()方法的开始

也就是说，要想并发程序正确地执行，必须要保证原子性、可见性以及有序性。只要有一个没有被保证，就有可能 会导致程序运行不正确。

要想并发程序正确的执行，必须同时保证原子性，可见性，有序性，只要有任意一个没有被保证，就有可能导致程序运行不正确。

6.3volatile变量的特殊规则（加内存屏障）

第一：保证此变量对所有线程的可见性。

当一条线程修改了这个变量的值，新值对于其他变量来说是可以立刻得知的。普通变量无法做到可见性。
volatile变量在各个线程中是一致的，但是volatile变量的运算在并发下一样是不安全的。
java运算操作并非原子性，必须结合内键锁或Lock体系来约束。

由于volatile只保证可见性，在不符合以下两条规则的场景下，**任然需要使用加锁来保证原子性**：

I.运算结果不依赖当前变量的值，或者能够确保只有单一的线程修改变量的值。
II.变量不需要与其他的状态变量共同参与不变约束。

并发场景下对于类似i++操作如何保证程序的正确结果：
1.加锁结合内键锁或Lock体系

2.使用原子类（java.util.current.atomic包下的所有类-内部使用CAS保证原子性）

第二：禁止指令重排
a.当程序执行到Volatile变量的读或写操作时，在其前面的操作更改肯定全部已经执行完毕且结果已经对后面的操作可见，在其后面的操作肯定还没有执行。
b.在进行指令优化时，不能将对volatile变量访问的语句放在其后面执行，也不能提前执行。

懒汉式单例模式线程安全问题：

1.在堆上分配空间
2.属性初始化
3.引用指向对象

Double check　Double Lock
双重检索模式(double checked locking pattern)