# 二、Java内存区域与内存溢出异常

## 1 概述

Jvm虚拟机负责java程序内存的管理（申请与释放）

## 2 运行时数据区域

jvm虚拟机会将其管理的内存分为很多种，来满足程序运行的需要。

### 1 程序计数器

每个线程所执行的字节码的指令位置。通俗讲就是：线程指向到代码的哪个位置。

线程私有的。

Why： 每个线程都需要记录位置。

### 2 栈（java虚拟机栈）

1 每个线程都会有自己独立的栈，存放局部变量。先进后出，入栈出栈。

2 栈是有深度的：如果深度不足，则StackOverflowError

如果栈支持动态拓展，申请不到足够的内存来拓展的时候，则OutOfMerroryError

3 为什么多线程编程的时候局部变量不影响线程同步？

因为每个线程都有自己独立的栈，不共享；但堆不是，多个线程共享同一个堆空间。

4使用 –Xss 调整大小。

### 3 本地方法栈

Native方法使用的栈

### 4 堆

1 程序运行的对象实例和数组分配在堆上。

2 怎么进行垃圾回收呢？

垃圾回收算法。

3 为了更好地垃圾回收，使用分代手机，堆又细分为几块。

新生代： Eden，From Survivor, To Survivor

老生代：

永久代：

### 5 静态方法区

所有线程共享的。

1 存储jvm加载的类信息，编译的代码信息。

类加载器加载的类信息应该放在这里。

2 如果是Hotspot虚拟机，方法区就是永久代。

3 内部可以细分出一块 常量池。 字符串、static变量都放在常量池中。所有线程共享这一份。但是，好像在jdk1.7之后，jvm逐步去永久代，已经把 字符串常量池从永久代中移出了。

### 7 直接内存 （堆外内存）

NIO中新加入的类，不属于JVM管理。但JVM+直接内存不能大于操作系统内存。

总结：

1 程序运行，首先加载类(放入方法区)-》static变量（方法区中）-》字符串（运行时常量池中）-》程序运行-》基本类型的局部变量（放入线程栈中）-》生成引用类型的对象（线程栈中放入引用指针，对象实例占用空间在堆中申请）

2 每块数据区域在什么情况下都报什么异常。

# 三、垃圾收集器与内存分配策略

哪些对象可以回收？（判断对象死亡还是存活）

什么时候回收？

如何回收？ （垃圾收集算法）

## 3.2 对象已死吗？

怎么判断对象是否还在“存活”，哪些算法？

### 3.2.1 引用计数法

给对象添加一个引用计数器，当其为0时，对象不再可用，可以回收了。

缺点： 无法解决对象间相互引用的问题。

### 3.2.2 可达性分析算法

从GC Roots对象出发，向下搜索，没有达到的对象，则是不可用的对象。

1 GC ROOts对象不是一个，而是多个。

2 GC Roots对象怎么挑选？

### 3.2.3 扩展引用几种类型

1 为什么增加几种引用的类型？

刚开始，只有强引用。但我们希望能描述这样的一类对象，当内存空间还足够时，则保留在内存中；如果内存空间在进行垃圾收集后还是很紧张，则抛弃这些对象。这样，能使用这些引用实现缓存功能。

它们的回收时机不同：

强引用：

软引用：在系统即将发生内存溢出异常之前，将这些对象进行二次回收。

弱引用：会延长一个垃圾回收的寿命。在第二次垃圾回收的时候，进行回收。

虚引用：对象被回收时，收到一个系统通知。

### 3.2.5 回收方法区

方法区（HotSpot中的永久代）也可以通过设置jvm参数进行垃圾回收，回收废弃的常量和无用的类，但影响小，我们并不关注。

## 3.3 垃圾回收算法

回收算法的发展历程。

### 3.3.1 标记-清除算法（Mark-Sweep）

最简单，最原始；效率不高，产生大量的空间碎片。

### 3.3.2 复制算法

为了解决上面算法效率低，内存碎片的问题。

思想：把内存分为大小相同的两块，每次只使用其中一块。而用完需要垃圾回收的时候，把存活对象复制到另外一块上。

特点：简单高效；但浪费了一部分内存空间。

现代的商业虚拟机都使用这种算法来对新生代的内存进行回收，但因为新生代中对象98%朝生夕死，并不是1：1比例划分。

Eden：From Survivor：To Survivor = 8 ： 1 ： 1 （使用90%的新生代内存，浪费10%），并不能保证10%的Survivor内存够用，因为需要老年代进行分配担保。

### 3.3.3 标记-整理算法（Mark-Compact）

老年代对象存活率高，复制算法不适用。（拷贝对象大，效率低，浪费空间）

因此，有了 标记-整理算法。

### 3.3.4 分代收集算法

把前面的算法进行整合，根据对象的存活周期不同将内存划分为几块，每块采用合适的算法。

## 3.5 垃圾收集器

讨论的是HotSpot虚拟机使用的哪些垃圾收集器。

STW： 暂停用户工作线程的时间，越少越好

1 Serial 收集器 复制算法 单线程、Client模式适用

2 ParNew 收集器 复制算法 Serial的多线程模式 Server模式适用

3 CMS 收集器 ConcurrentMarkSweep 老年代

缩短STW时间为目的。3次标记，一次清除。

a 初始标记

4 G1 收集器

参考：

<https://ylgrgyq.github.io/jvm/2016/07/03/garbage-first-collector-understanding.html>

<http://www.open-open.com/lib/view/open1482392048975.html>

<http://ifeve.com/%E6%B7%B1%E5%85%A5%E7%90%86%E8%A7%A3g1%E5%9E%83%E5%9C%BE%E6%94%B6%E9%9B%86%E5%99%A8/>

## 3.6 内存分配与回收策略

1 如果启用了本地线程分配缓冲，对象将优先分配到线程的TLAB上。

2 对象优先在Eden区中分配

注意： -Xmn10m是配置新生代的内存大小为10m，而不是Eden。

如果-XX:SurvivorRation=8,则Eden区为8M，From和To Survivor区各自占 1M。新生代实际使用9M，浪费1MM。当对象在**Eden区**放不下时，会触发Minor GC。

3 大对象直接进入老年代。

何为大对象？有个参数可以配置： -XX:PretenureSizeThreshold=3145728 (3M)

4 长期存活的对象将进入老年代。

什么是长期存活？ Jvm 对象有个年龄计数器(Age)。 每经历过一次Minor Gc，age加1，当达到设置的阈值时候，则进入老年代。 -XX:MaxTenuringThreshold=10 (10岁，10次)

5 动态对象年龄判定

jvm并不永远要求年龄达到设置的阈值，就进入老年代。

如果在Survivor空间中，相同年龄的对象总和达到Survivor空间的一半，则大于等于该年龄的对象集体进入老年代。

6 空间分配担保

新生代发生Minor GC，不知道会有多少内存被回收，多少对象晋升为老年代。如果晋升为老年代的对象总大小大于了老年代最大可用的连续空间怎么办？ 这是不安全的。

因此，在Minor GC前，如果 小于老年代最大可用连续空间，则安全。

如果大于，则检查最大可用连续空间是否大于 晋升为老年代对象的平均大小，

如果大于，则尝试Minor GC；如果失败了，再进行Full GC。

如果小于，则 Full GC。

为了避免频繁的Full GC，一般打开分配担保。 -XX:-HandlePromotionFailure

# 七、虚拟机类加载机制

# 十二、Java内存模型与线程

## 2 硬件效率与一致性

1 CPU处理速度快，但只能逐个计算；内存读取速度慢，但可以存批量。

如果解决CPU与内存的速度矛盾？

每个处理器增加了高速缓存（cache 快取），内存中的运算单元批量放入高速缓存中，CPU对高速缓存进行读写。

## 3 Java内存模型

java的内存模型同上面的硬件模型类似。

1 java的内存模型是什么？

三个单位：线程、每个线程的工作内存、主内存。

所有的共享变量都放在主内存中；

每个线程只与自己的工作内存进行交互，不能直接读写住内存；

工作内存对主内存存在8种原子操作。

## 4 volatile 关键字

1 “赋值可见性”：当一个线程修改了这个变量的值，新值对其他线程来说是可以立刻得到的。

当普通共享变量修改后，只是在工作内存中修改，并不立刻同步到主内存中，因此其他线程获取的值仍然是旧的数据；

而volatile修饰的变量，当线程对其在工作内存中修改后，立刻同步到内内存；而其他线程读取这个变量的时候，强制从主内存刷新到工作内存，因此是立即可见的。

2 禁止指令重排序。

什么是指令重排序？

两段逻辑互不关联的代码，其执行顺序并不是按照源码的书写顺序来执行，在编译的时候会重排序。尤其如果在多个线程中执行时。

为什么要禁止？

## 5 线程的状态转换

线程的生命周期中有6中状态：

New、Runnable（Running+Ready）、Terminated

中间夹杂着 waiting(无限期等待) 、Timed Waiting(设置了时间参数)、

Blocked(阻塞，等待获取一个排他锁，进入同步区域)

# 十三、线程安全与锁优化

1 什么是线程安全？ 这个类是线程安全的，指的是什么？

当多个线程访问同一个对象时，无论怎么执行，都能获得正确的结果，就是线程安全的。

## 1 线程安全的实现方法

### 1 互斥同步（悲观锁）

1 synchronized

原理：编译后，会在同步块的前后形成monitorenter和monitorexit这两个字节码指令，这两个指令需要指定锁定和解锁的对象。

2 ReentrantLock

优点：1 等待可中断 2 线程可以公平 3 锁可以绑定多个条件，即可以绑定多个Condition对象。而synchronized只有一个隐含的对象实现wait()和notify()

### 2 非阻塞同步（乐观锁）

设计思想： 先进行操作，如果没有其他线程使用共享数据，则操作成功；如果共享数据在被使用，则产生了冲突，使用补偿措施（最常见的是循环重试，直到成功为止）。

随着硬件指令集的发展而产生。

乐观锁避免线程挂起。

CAS （Compare And Swap） 比较并交换，这是一个原子操作。

但CAS有一个逻辑漏洞：ABA问题。

## 2 锁优化

优化锁的性能

### 1 自旋锁和自适应自旋

什么是自旋锁？ 自己旋转，空转的锁。

为什么要空转？ 阻塞时间和线程切换的衡量。

互斥锁对性能最大的影响是阻塞的实现，挂起线程和恢复线程都需要转入内核态中完成，极大影响系统的并发性能。空转的时间可以由自旋次数设置。

自适应：是自旋的时间不再固定。

### 2 锁消除

编译器检测到不可能存在共享数据竞争的锁进行消除

### 3 锁粗化

### 4 轻量级锁

### 5 偏向锁