## 1、jvm布局：

jdk1.6版本JVM布局分为：heap（堆）,method（方法区，包含常量池）,stack（虚拟机栈）,native stack（本地方法栈）,程序计数器共五大区域。

其中方法区包含运行时常量池。**堆和方法区是线程共享的**，虚拟机栈和本地方法栈、程序计数器是随线程而建的。

**类信息放在方法区**

### 1.1、堆：

**储存对象信息和数组。**对象信息/数组包括****对象头，实例数据和对齐填充****共三个区域；

1.1.1、对象头包括二/三部分内容：

一是**类型指针**，即对象指向它的类元数据的指针，通过这个指针来确定那个类的实例（指向方法区储存的对象类型数据）；

二是用于存储对象自身的**运行时数据**，如哈希码，gc分代年龄，锁状态标志，线程持有的锁，偏向线程id，偏向时间戳等（ps:不了解这些，后续学习）；

三是如果存储的是数组，对象头除了包含以上俩个内容外还必须得有数组的长度数据。

其中类型指针还有一个知识点是对象的访问定位，对象的访问方式主流的有俩种，**句柄和直接指针**。这俩种区别与特点是：

①、在对象被移动时（垃圾收集时会移动对象即内存整理），句柄只需修改句柄中的对象实例数据的指针，栈中的reference(引用）不需要修改。而直接指针需要修改。

②、直接指针的优势就是速度更快，因为少了一次指针定位的时间开销。

-- 具体句柄和直接指针图解详见书本第49页。

1.1.2、**实例数据包括实例化对象存储的数据。**

1.1.3、对齐填充：**对齐填充并不是必然存在的，也没有特殊含义**。由于hotspot甲鱼的臀部（规定）对象的大小必须是8字节的整数倍。而对象头部分正好是8字节的倍数（1倍或者2倍），因此，当对象实例数据部分没有对齐时就需要通过对齐填充来补全。

1.1.4、对象创建分配内存有俩种方式：**指针碰撞（无内存泄漏）和空间列表（内存不连续，分配时找到一块足够大的内存划分给对象实例）；选择哪种方式创建由垃圾收集器是否有压缩整理功能决定。**

### 1.2、方法区：

**存储已被jvm加载的类信息(类名，访问修饰符，字段描述，方法描述等)、常量、静态变量、即时编译器编译后的代码（不清楚什么是即时编译器编译后的代码）等数据。**也就是说 final/static 的“基本数据类型变量”数据和指针放在方法区，没有final/static 的“基本数据类型变量”数据和指针放在虚拟机栈中。jdk1.6及以前方法区和堆独立区域，jdk1.7方法区中的字符串常量池放在堆中，jdk1.8删除方法区改成元空间（还没太了解）。

### 1.2.1、运行时常量池

：存储编译期间生成的各种字面量和符号引用。其中有个知识点是string的instern()方法。

详情见http://blog.csdn.net/hupoling/article/details/62423613的总结。

### 1.3、虚拟机栈：

其实严格来说虚拟机栈包含局部变量表、操作数栈、动态链接、方法出口等信息。其中局部变量表存储八种基本数据类型（byte,boolean,char,short,int,float,long,double）和对象引用。平常讨论的栈都是指的局部变量。

### 1.4、本地方法栈：

本地方法栈和虚拟机栈作业是十分相似的，只不过虚拟机栈是为虚拟机执行java方法（也就是字节码）服务，而本地方法栈是为虚拟机执行native方法服务。**sun hotspot虚拟机本地方法栈和虚拟机栈合二为一**。

### 1.5、程序计数器：

可以看做是当前线程执行字节码的行号指示器。此区域是jvm规范中唯一没有规定任何OOM情况的区域。

### 直接内存：

1. 在jdk1.4之后新加入了NIO，可以使用native函数库直接分配堆外内存（本机内存），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。避免了io在java堆和native堆来回复制数据提示对鞋性能。直接内存大小默认是java堆xmx（不知道为什么这样设计 容易误导）；

## 3、OOM

### 3.1、堆溢出：

Xmx:最大堆内存，Xms:

最小堆内存，当Xmx=Xms时堆不扩展。-XX:+HeapDumpOnOutOfMemoryError ；实例化大量对象可测试堆溢出。

3.1.1、OOM解决：增大xmx扩大堆内存

### 3.2、栈溢出：

栈溢出有俩种情况:

3.2.1、栈的深度超过最大深度限制，抛出StackOverflowError异常

3.2.2、栈扩展时内存不足，抛出OOM；

Xss:每个栈的大小；Xoss：本地方法栈大小，不过实际上xoss无效，栈容量只由-Xss参数设置。

3.2.3、实测中3.2.1说法有点不全。单线程下只有一个栈，所以只可能出现**StackOverflowError，**无论是栈的深度太大还是每个栈帧过大到账内存不足；多线程下会出现OOM,不断建线程时会出现内存不足；

### 3.2.4、OOM/StackOverflowError解决：

3.2.4.1、OOM:由于**栈内存=操作系统内存 - 堆内存 - 方法区内存 - 程序计数器内存，**所有可以**减小堆内存来扩大栈内存大小。**栈内存大小影响系统并发线程量（栈内存>=每个栈的大小xss\*线程量）;具体设置由n台服务器，每台服务器m个cpu,则最大线程量=n\*m;每个栈的大小xss<=栈内存/n\*m; **注意一下有个问题 ，这个公式没有直接内存？**

3.2.4.2、StackOverflowError深度：如果使用jvm默认设置，栈的深度大多数情况下可达到**1000~2000**,足以在日常开发中使用。注意避免代码中存在超过1000的方法嵌套。每个方法嵌套对应一个栈帧。

3.2.4.3、StackOverflowError栈帧大小：单线程下避免代码中存在大量基本类型或对象引用。

3.2.4.4、多线程下假设每个栈帧特大，jvm是抛出OOM还是StackOverflowError？（待考察研究）

3.3、方法区OOM：-XX:PermSize:最小方法区内存大小；-XX：MaxPermSize:最大方法区内存大小。

3.3.1**、OOM解决：避免大量的string.intern()；避免大量的动态java**（jsp,java反射）。

3.4、本机直接内存溢出：-XX:MaxDirectMemorySize：最大直接内存；默认为Xmx

3.4.1、OOM解决：使用NIO分配本机内存多注意是否超过MaxDirectMemorySize；平常开发容易忽略直接内存；

## 1.垃圾回收条件

jvm垃圾回收分为俩点，一是对象的内存回收也就是堆内存回收，二是字面量（运行时常量池）和类信息内存回收也就是方法区内存回收。

1.1 对象内存回收（heap）

如何判断对象是否可以回收，市场上有引用计数**[算法](http://lib.csdn.net/base/datastructure" \o "算法与数据结构知识库" \t "http://blog.csdn.net/hupoling/article/details/_blank)**和可达性分析算法两种方法 :

1.1.1 引用计数算法：简单来说就是，一个对象A被引用一次，对象A的引用次数就加一。当对象A的引用次数为0（也就是没有其他地方引用此对象）时可以回收。然而这里存在个问题就是 对象之间互相引用 A->B,B->A时 AB无法回收。不过微软公司的com（componet object model）技术用的就是这个算法，而目前流行的主流的jvm没有选用此算法的。

1.1.2 可达性分析算法：简单来说就是，通过一系统成为Gc Roots的对象为起始点,当从GC Roots到达不了对象A时，则对象A是不可用的，可回收的。

### java语言中，GC Roots包括四种对象：

1.1.2.1 虚拟机栈（栈帧中的本地变量表）中引用的对象

1.1.2.2 本地方法栈中（native方法）引用的对象

1.1.2.3 方法区中类静态属性引用的对象

1.1.2.4 方法区中常量引用的对象

之所以是上面四种，总结为：**GC管理的主要区域是Java堆，一般情况下只针对堆进行垃圾回收。方法区、栈和本地方法区不被GC所管理,因而选择这些区域内的对象作为GC roots**

### 1.2 方法区内存回收：

jvm规范不要求虚拟机在方法区实现垃圾收集（因为性价比低，回收一次回收的空间实际情况很小），但是jvm还是有方法区回收的。

1.2.1 字面量回收：如果一个字面量没有被引用就回被回收；

1.2.2 类信息回收：类信息回收需要同时满足3个条件：

1.2.2.1 java堆中不存在此类的实例；

1.2.2.2 加载该类的classloader已经被回收

1.2.2.3 该类对应的java.lang.Class对象没有任何地方引用，也就是说无法再任何地方通过反射访问该类的方法。

1.2.3 其他常量池的东西 暂不清楚。

### 1.3 四大类型引用：

自jdk1.2之后（jdk1.1引用分的种类不够明确），jvm中还有几种类型引用比较特殊；。分别是 强引用、软引用、弱引用、虚引用。

1.3.1 强引用：Object obj=new Object();只要强引用还存在，垃圾收集器永远不会回收被引用的对象。假如obj=null或者obj引用的栈帧出栈（不存在）就会被回收。

1.3.2 软引用：将要发生内存溢出溢出之前，将会把软引用的对象进行回收。SoftReference类实现软引用。

1.3.3 弱引用：只能生存到下一次gc之前。WeakReference类实现弱引用。

1.3.4 虚引用：这个需要注意，虚引用无法用来取得一个对象的实例。虚引用不会对对象的生存时间有任何影响，只是能在这个对象被回收时收到一个系统通知。推荐这篇文章更加了解虚引用：<http://www.mamicode.com/info-detail-988201.html。>

对于String类型的虚引用，因为Java中存在内建的常量池，所以使用String类型的虚引用时其实在常量池中是存在对应的强引用的，所以就算在finalize中不发生逃逸，gc也是不能成功的。

关于四种引用推荐一篇文章里面有详细例子：http://blog.csdn**[.NET](http://lib.csdn.net/base/dotnet" \o ".NET知识库" \t "http://blog.csdn.net/hupoling/article/details/_blank)**/u011277123/article/details/53908315

## 1.4 finalize() 对象自救

对象在可达性分析之后还需要进行一些逻辑。哪怕对象在可达性分析之后没有发现与GcRoots的引用链，jvm还需要进行下面几步(需要俩次标记)：

1.4.1 没有发现引用链，进行第一次标记

1.4.2 判断对象是否要执行finalize()方法：当对象没有覆盖finalize()方法或者已经被执行过一次finalize()方法；则此对象不执行finalize()方法，否则进行执行finalize()方法（把对象放在F-Quene队列中）；

**jvm自动建立的，低优先级的Finalizer线程去执行放在F-Quene的对象里的finalize()方法。**

1.4.3 进行第二次标记，判断对象是否有引用链，没有则回收。

所以如果对象自救（第一次标记没有引用链，第二次标记有引用链），只有在覆盖finalize（）方法激活对象（让对象被引用）。

## 垃圾收集算法：

1. jvm垃圾收集可以分为四种（严格来说是三种）：标记-清除算法；标记-整理算法；复制算法；分代收集算法。

### 2.1 标记-清除算法：

先把可回收内存标记，然后在清除掉；缺点是：会存在内存碎片导致内存泄漏。

### 2.2 标记-整理算法：

先把可回收内存标记，然后让存活的对象都向一端移动，最后清除点端边界以外的内存；缺点：比较标记-清除时间增长；

### 2.3 复制算法：

把内存平分为两份s1,s2；保证只用一个内存区s1，另一个为空s2；把存活的对象复制到为空的那个内存区域s2，然后在清除掉这个区域s1；缺点：内存减半

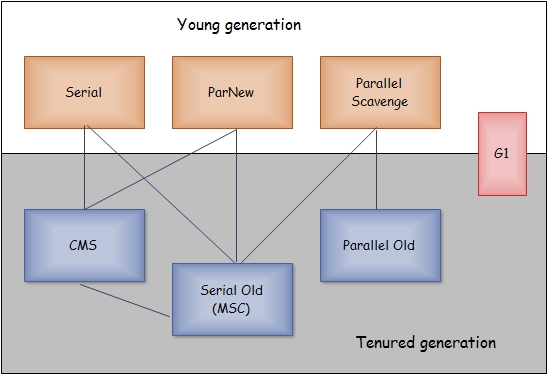
2.4 分代收集算法：

根据jvm特性，年轻代实际每次gc时存活对象较少，故用推荐复制算法；年老代存活对象较多，并且没有其他内存为年老代分配担保（分配担保：举个栗子：年老代为年轻代进行分配担保，当年轻代minor gc内存不足时【比如有个大对象obj1】obj1会放到年老代中；而年老代没有其他空间为其分担了）所以推荐标记-整理算法。

## 垃圾收集器：

共七中垃圾收集器，分别是serial、parnew、parallel scavenge和cms、serial old、 parallel old、G1；其中serial、parnew、parallel scavenge用于年轻代，cms、serial old、 parallel old用于年老代，g1用于年轻代和年老代。所有收集器都存在stop the world，不过在java发展中 一直在优化停顿时间。

先总结比较这七个收集器：



### 年轻代：

### 4.1.Serial:

适用于年轻代垃圾回收，复制算法，jdk1.3之前 推荐用于客户端模式(Client)下的虚拟机，属于单线程，无对stop the world优化，属于最老的年轻代垃圾收集器产品；

### 4.2 ParNew:

适用于年轻代垃圾回收，复制算法，jdk1.3发布，推荐用于服务端模式（Server）下的虚拟机，属于多线程，无对stop the world优化，其实就是Serial的多线程版本；是服务端开发的首先；

### 4.3 Parallel Scavenge：

适用于年轻代垃圾回收，复制算法，jdk1.4发布，属于多线程，无对stop the world优化，它是一个可以控制吞吐量的收集器，拥有**自适应调节策略**；需要注意一点的是，**gc停顿时间缩短是牺牲吞吐量和新生代空间来换取的。**

stop The World:gc时 需要停止所有线程进行垃圾回收；

**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)**

自适应调节策略：虚拟机会根据当前系统的运行情况收集性能监控信息，动态调整参数以提高最合适的停顿时间或最大吞吐量。

### 年老代

### 4.4 Serial old ：

适用于年老代垃圾回收，标记-整理算法，jdk1.5之前，**主要**用于client下的虚拟机,如果用在server模式下主要有俩个作用：一是jdk1.5以及之前用于与Parallel Scavenger搭配使用，二就是为cms提供后备预案，属于单线程，无stop the world优化。

### 4.5 Parallel Old：

适用于年老代垃圾回收，标记-整理算法，jdk1.6发布，属于多线程，注重于吞吐量控制，是为了Parallel Scavenge定制的；

### 4.6 CMS:

适用于年老代垃圾回收，**标记-清除算法**,jdk1.5发布，推荐server模式下，属于多线程，对stop the world有优化，CMS是一种以获取最短回收停顿时间为目标的收集器，也就是优化服务器响应速度。CMS分为4步：其中 初始化标记，重新标记 stop the world

4.6.1 **初始标记：**

仅仅只是标记GcRoots可以直接关联的对象；

4.6.2 **并发标记**：进行gcroots tracing（也就是 引用链搜索）；优化成并发

4.6.3 **重新标记**：修正并发标记因用户程序继续运行而导致标记产生变动那一部分对象的标记记录。

4.6.4 并发清除：并发清理标记的对象内存。

### 4.6.5 CMS缺点：

4.6.5.1 **CMS对cpu资源非常敏感，**默认启动的回收线程数是（cpu数量+3）/4；垃圾回收线程数量不少于25%的cpu资源，当cpu越大，线程数/cpu总量 越小；但是当cpu小于4个时 显得就不怎么合适了；

4.6.5.2 **无法处理浮动垃圾**，需要等下一次gc才能处理；并且还需要预留一部分空间提供并发收集时的用户程序运作使用，即启动阈值（-XX:CMSInitiatingOccupancyFraction 阈值百分比）；要是CMS运行期间预留的空间不满足程序需要，就会出现一次“**Concurrent Mode Failure**”失败，这是虚拟机启动后备预案：临时启用Serial Old，这样一来停顿时间就很长了，所以-XX:CMSInitiatingOccupancyFraction 设置太高容易导致大量Concurrent Mode Failure”失败，性能反而降低，采用SerialOld 来清理老年代

4.6.5.3 由于**用的是标记-清除算法，所以会出现内存碎片**；CMS提供-XX:UseCMSCompactAtFullCollection开关参数（默认为开），用于CMS收集器要进行FullGC时进行内存碎片合并整理，-XX:CMSFullGCsBeforeCompaction 用来设置执行多少次不压缩Full GC后跟着来一次带压缩的（默认为0，表示每次进入Full GC都进行碎片压缩）。

年轻代+年老代

### 4.7 G1：

jdk1.7发布，整体看是“**标记-整理算法**”，从局部（俩个Region之间）上来看是基于“复制”算法实现，也就是说没有内存碎片；G1将整个堆区分为多个大小固定的区域，并且跟踪这些区域里面的垃圾堆积程度，在后台去维护一个优先级列表，每次根据允许收集的时间，优先回收垃圾堆积程度最大区域，这就是G1(Garbage First)的由来

**如果应用追求低停顿，那G1现在可以作为一个尝试的选择，如果应该追求吞吐量，G1并不会带来什么特别的好处！！！**

## 内存分配策略：共有 4个策略

### 5.1 对象优先在Eden分配：

major gc时经常会伴随至少一次的minor gc，但并非绝对，比如说 Parallel Scavenge就是直接major gc没有伴随minor gc;

### 5.2 大对象直接进入老年代

-XX：PretenureSizeThreshold设置最大对象，单位B;PretenureSizeThreshold参数只对Serial和ParNew两款收集器有效；Parallel Scavenge收集器不需要设置；如果遇到必须使用此参数的场景，可以考虑ParNew+CMS组合；

### 5.3 长期存活的对象放入老年代:

虚拟机给每个对象定义了一个对象年龄（Age）计数器；gc一次Age+1,当Age大于一定程度（默认为15岁）就会被晋升到老年代；-XX:MaxTenuringThreshold：最大年龄；

### 5.4 动态对象年龄判定：

如果在Survivor空间中相同年龄所有对象大小的总和大于Survivor空间的一半，年龄大于或等于改年龄的对象就可以直接进入老年代，无须等到MaxTenuringThreshold中要求的年龄；

### 5.4空间分配担保

jdk6 update 24后，只要老年代的连续空间大于新生代对象总大小或者历次晋升的平均大小就会进行major gc，否则Full GC；

## 1.jdk的命令行工具简单总结介绍

1.1 jps: 获取虚拟机进程 vmid；jps -v ：显示虚拟机对应启动的参数

1.2 jinfo:获取虚拟机参数值和动态修改部分运行期可改的参数。jinfo [option] pid例如：jinfo -flag PermSize vmid;显示vid对应的虚拟机的方法区大小；jinfo -flag +/- name：添加或删除name属性；

1.3 jmap:java内存影像工具， jmap -heap vmid:显示堆的详细信息，如参数配置，分代状况;jmap -histo vmid:显示堆中对象统计信息，包括类，实例数量，合计容量。

1.4 jstat:虚拟机监控工具，jstat -gccause vmid:输出已使用空间占各自总空间的百分比；

1.5jhat:虚拟机堆转储快照分析工具；jstack：java堆栈跟踪工具；

2.jdk的可视化工具:jConsole和visualVm；其中VisualVM有个BTrace插件值得注意。

2.1 **BTrace 动态日志跟踪**：可以打印调用堆栈、参数、返回值、性能监视、定位连接泄漏、内存泄漏、解决多线程竞争问题。如生产上遇到问题时需要方法输入参数和返回输出参数，但开发时没有日记记录，平常做法就是补充上日志，然后在重现上线。而**BTrace可以在不停止jvm的情况下动态调试代码。**（ps:BTrace后续深入研究）

推荐一篇文章：http://huanghaifeng1990.iteye.com/blog/2121419

## 一、jvm调优思路：

第一步：jps 获取jvm id；

第二步：获取jvm垃圾收集器种类

第三步：查看gc次数和时间，分析原因来优化

gc次数频繁：①、内存回收率低导致短时间内回收次数多；②、内存大小太小;

gc时间长：①、内存过大；②、内存扩展导致时间长（固定内存大小）

选择适合的收集器也可大幅度优化jvm。

二、优化思路注意点：

1、64位jdk的性能测试结果普遍低于32位jdk；

2、64位jdk由于指针膨胀和数据类型对齐补白导致消耗的内存比32位大；

3、使用nio时，堆外内存不足导致内存溢出。