Java GC 笔记总纲

一 . 基础知识

1 . 1 常见的内存溢出

> 堆溢出
> 元空间溢出 ， 元数据区的内存溢出
> 直接内存溢出
> 虚拟机栈和本地方法栈溢出
> 运行时常量池溢出
> 方法区的内存溢出

1 . 2 内存溢出的常见原因

1 . 3 系统的线程划分

> 串行收集器
　　：用单线程处理所有垃圾回收工作 ， 效率高
　　：数据量比较小（100M左右）；单处理器下并且对响应时间无要求的应用
> 并行收集器
　　：“对吞吐量有高要求”，多CPU、对应用响应时间无要求的中、大型应用

> 并发处理器：
　：对响应时间有高要求”，多CPU、对应用响应时间有较高要求的中、大型应用

1 .4 Java 的四种引用类型

- 强引用
- 软引用（SoftReference）
- 弱引用（WeakReference）
- 虚引用（PhantomReference）

1）强引用
　　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

软引用（SoftReference）

二 . 虚拟机

2 . 1 Java 虚拟机

Java 虚拟机，是一个可以执行 Java 字节码的虚拟机进程 ， 它允许Java 查询在多个任意平台使用 ， 但是跨平台的是 Java 程序(包括字节码文件) ， 而不是 JVM

> 类加载器，在 JVM 启动时或者类运行时将需要的 class 加载到 JVM 中。

> 内存区，将内存划分成若干个区以模拟实际机器上的存储、记录和调度功能模块，如实际机器上的各种功能的寄存器或者 PC 指针的记录器等。

> 执行引擎，执行引擎的任务是负责执行 class 文件中包含的字节码指令，相当于实际机器上的 CPU 。

> 本地方法调用，调用 C 或 C++ 实现的本地方法的代码返回结果。

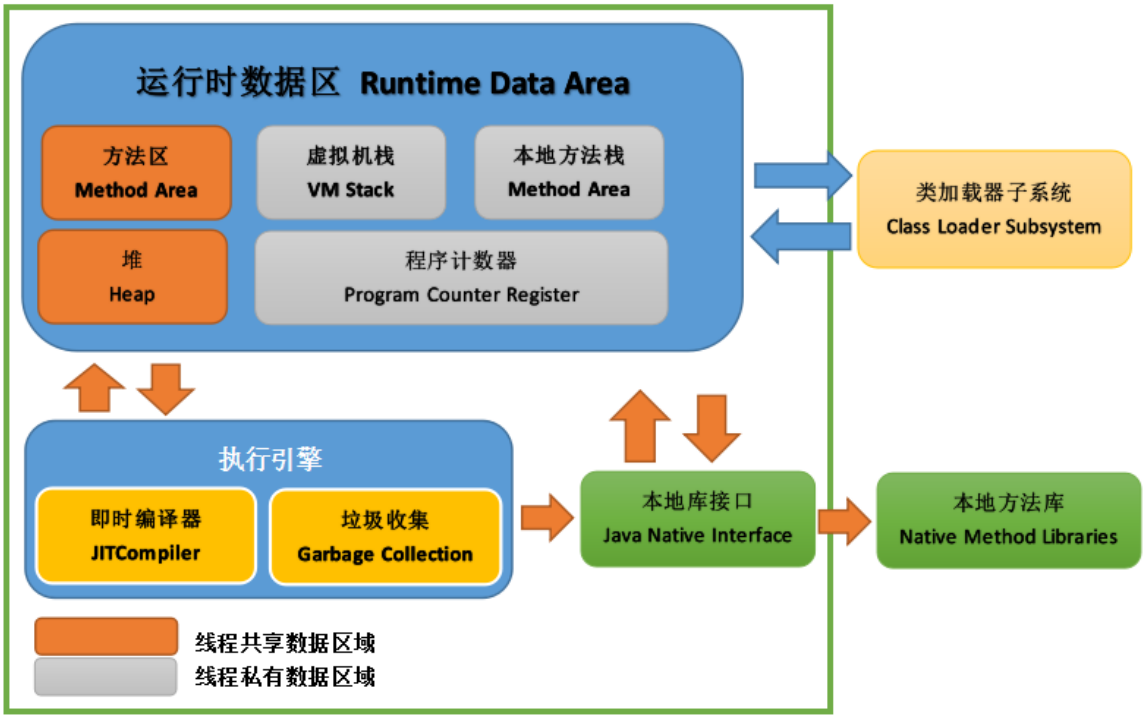
// 运行时数据区
> 程序计数器： Java 线程私有，类似于操作系统里的 PC 计数器，它可以看做是当前线程所执行的字节码的行号指示器。
- 如果线程正在执行的是一个 Java 方法，这个计数器记录的是正在执行的虚拟机字节码指令的地址；如果正在执行的是 Native 方法，这个计数器值则为空（Undefined）

> 虚拟机栈（栈内存）：Java线程私有，虚拟机栈描述的是 Java 方法执行的内存模型

> 本地方法栈 ：和 Java 虚拟机栈的作用类似，区别是该区域为 JVM 提供使用 Native 方法的服务

> 堆内存（线程共享）：所有线程共享的一块区域，垃圾收集器管理的主要区域
- 每个方法在执行的时候，都会创建一个栈帧用于存储局部变量、操作数、动态链接、方法出口等信息。
- 每个方法调用都意味着一个栈帧在虚拟机栈中入栈到出栈的过程。

> 方法区（线程共享）：各个线程共享的一个区域，用于存储虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据



2.2 内存堆

2.3 内存栈

2.4 Java 内存堆和栈区别

栈内存用来存储基本类型的变量和对象的引用变量；堆内存用来存储Java中的对象，无论是成员变量，局部变量，还是类变量，它们指向的对象都存储在堆内存中。

栈内存归属于单个线程，每个线程都会有一个栈内存，其存储的变量只能在其所属线程中可见，即栈内存可以理解成线程的私有内存；堆内存中的对象对所有线程可见。堆内存中的对象可以被所有线程访问。

如果栈内存没有可用的空间存储方法调用和局部变量，JVM 会抛出 java.lang.StackOverFlowError 错误；如果是堆内存没有可用的空间存储生成的对象，JVM 会抛出 java.lang.OutOfMemoryError 错误。

栈的内存要远远小于堆内存，如果你使用递归的话，那么你的栈很快就会充满。-Xss 选项设置栈内存的大小，-Xms 选项可以设置堆的开始时的大小。

2 . 6 HotSpot虚拟机

HotSpot 虚拟机将其物理上分为了2个部分 ：

> 新生代（young generation）
: 绝大多数最新被创建的对象会被分配到这里
: 对象从这个区域消失的过程我们称之为”minor GC“

-> 新生代三空间
：一个伊甸园空间（Eden ）
：两个幸存者空间（Survivor ）
创建后待伊甸园 -- 第一次GC --> 其中一个幸存者空间 -- 不断堆积--> 饱和后移动到第二个幸存者空间 --> 清空饱和空间 --> 几轮后剩下的放入老年代

> 老年代（old generation）
: 对象没有变得不可达，并且从新生代中存活下来，会被拷贝到这
: 对象从老年代中消失的过程，我们称之为”major GC“（或者”full GC“）

> card table
: 存在于老年代 ，512 byte,记录老年代对新生代的应用
: 由一个 write barrier

> 持久代（ permanent generation ） 又名 方法区（method area）
: 保存类常量以及字符串常量

> 加快缓存分配
: bump-the-pointer
- 跟踪在伊甸园空间创建的最后一个对象 ,放在顶部，下次创建查找该对象
: TLABs（Thread-Local Allocation Buffers）
- 该方案为每一个线程在伊甸园空间分配一块独享的空间，这样每个线程只访问他们自己的TLAB空间，再与bump-the-pointer技术结合可以在不加锁的情况下分配内存

三 . 垃圾清理

3 . 1 垃圾回收的起因

> 程序员无法自动完成系统的GC ，GC 一般在以下环境被创建

大多数对象会很快变得不可达
只有很少的由老对象（创建时间较长的对象）指向新生对象的引用

3.2 垃圾回收中的概念

```
// stop-the-world
: Stop-the-world会在任何一种GC算法中发生
: Stop-the-world意味着 JVM 因为要执行GC而停止了应用程序的执行
: 当Stop-the-world发生时，除了GC所需的线程以外，所有线程都处于等待状态，直到GC任务完成
: GC优化很多时候就是指减少Stop-the-world发生的时间


// 分代回收
> 为什么垃圾回收要分代:
  : 不同的对象生命周期是不一样的 ，采用不同的收集方式，可以提高回收率

> 分代的方式 :
  : 年轻代
  : 老年代
  : 持久代

// 新生代 GC 和老年代 GC
新生代 : 一个 Eden 区 + 两个 Survivor 区
老年代 : 默认新生代(Young)与老年代(Old)的比例的值为 1:2 ，默认的 Eden:from:to=8:1:1

新生代GC（MinorGC/YoungGC）：指发生在新生代的垃圾收集动作，因为 Java 对象大多都具备朝生夕灭的特性，所以 MinorGC 非常频繁，一般回收速度也比较快。
老年代GC（MajorGC/FullGC）：指发生在老年代的 GC，出现了 MajorGC，经常会伴随至少一次的MinorGC（但非绝对的，在 Parallel Scavenge 收集器的收集策略里就有直接进行 MajorGC 的策略选择过程）。MajorGC 的速度一般会比 MinorGC 慢 10 倍以上。


// 触发分代回收的方式
Scavenge GC和Full GC。

Scavenge GC : 新对象生成  ，并且在 Eden 申请空间失败 ，即触发


// 垃圾收集器
新生代收集器
    -Serial 收集器
    - ParNew 收集器
        ?- ParNew 收集器，是 Serial 收集器的多线程版。
    - Parallel Scavenge 收集器

老年代收集器
    - Serial Old 收集器
        ?- Serial Old 收集器，是 Serial 收集器的老年代版本。
    - Parallel Old 收集器
        ?- Parallel Old 收集器，是 Parallel Scavenge 收集器的老年代版本。
    - CMS 收集器
```

```
新生代 + 老年代收集器
    - G1 收集器
    - ZGC 收集器


//  G1 和 CMS 的区别
```

- CMS ：并发标记清除。他的主要步骤有：初始收集，并发标记，重新标记，并发清除（删除）、重置。
- G1：主要步骤：初始标记，并发标记，重新标记，复制清除（整理）
- CMS 的缺点是对 CPU 的要求比较高。G1是将内存化成了多块，所有对内段的大小有很大的要求。
- CMS是清除，所以会存在很多的内存碎片。G1是整理，所以碎片空间较小。
- G1 和 CMS 都是响应优先把，他们的目的都是尽量控制 STW 时间。
- G1 和 CMS 的 Full GC 都是单线程 mark sweep compact 算法，直到 JDK10 才优化为并行的。

收集器	串行、并行or并发	新生代/老年代	算法	目标	适用场景
Serial	串行	新生代	复制算法	响应速度优先	单CPU环境下的Client模式
Serial Old	串行	老年代	标记-整理	响应速度优先	单CPU环境下的Client模式、CMS的后备预案
ParNew	并行	新生代	复制算法	响应速度优先	多CPU环境时在Server模式下与CMS配合
Parallel Scavenge	并行	新生代	复制算法	吞吐量优先	在后台运算而不需要太多交互的任务
Parallel Old	并行	老年代	标记-整理	吞吐量优先	在后台运算而不需要太多交互的任务
CMS	并发	老年代	标记-清除	响应速度优先	集中在互联网站或B/S系统服务端上的Java应用
G1	并发	both	标记-整理+复制算法	响应速度优先	面向服务端应用，将来替换CMS

3.3 垃圾回收的分类

```
> 分代回收


// JDK GC 的类型

Serial GC
  - 标记 - 清理 - 压缩
Parallel GC
Parallel Old GC (Parallel Compacting GC)
Concurrent Mark & Sweep GC (or “CMS”)
Garbage First (G1) GC
```

3 .4 常见得垃圾回收方式

3 . 5 垃圾回收的算法

> 应用计数
>
> ： 对一个对象有引用/移除 。 即添加/删除数量 ， 垃圾回收会回收数量为 0 的对象

> 标记清除
>
> ： 第一阶段从引用根节点开始标记所有被引用的对象
>
> ： 第二阶段遍历整个堆，把未标记的对象清除

> 复制（Copying）
>
> ： 将算法的内存空间分为相等的两个部分，回收时，遍历当前区域，将使用的对象复制到另外的区域

> 标记-整理（Mark-Compact）：
>
> ： 第一阶段从根节点开始标记所有被引用对象
>
> ： 第二阶段遍历整个堆，把清除未标记对象并且把存活对象“压缩”到堆的其中一块，按顺序排放

四 . 对象的创建

4 . 1 创建过程

1）检测类是否被加载
当虚拟机遇到 new 指令时，首先先去检查这个指令的参数是否能在常量池中定位到一个类的符号引用，并且检查这个符号引用代表的类是否已被加载、解析和初始化过。如果没有，就执行类加载过程。

2）为对象分配内存
类加载完成以后，虚拟机就开始为对象分配内存，此时所需内存的大小就已经确定了。只需要在堆上分配所需要的内存即可。
具体的分配内存有两种情况：
第一种情况是内存空间绝对规整
第二种情况是内存空间是不连续的。
- 对于内存绝对规整的情况相对简单一些，虚拟机只需要在被占用的内存和可用空间之间移动指针即可，这种方式被称为“指针碰撞”。
- 对于内存不规整的情况稍微复杂一点，这时候虚拟机需要维护一个列表，来记录哪些内存是可用的。分配内存的时候需要找到一个可用的内存空间，然后在列表上记录下已被分配，这种方式成为“空闲列表”。

多线程并发时会出现正在给对象 A 分配内存，还没来得及修改指针，对象 B 又用这个指针分配内存，这样就出现问题了。解决这种问题有两种方案：
- 第一种，是采用同步的办法，使用 CAS 来保证操作的原子性。
- 另一种，是每个线程分配内存都在自己的空间内进行，即是每个线程都在堆中预先分配一小块内存，称为本地线程分配缓冲（Thread Local Allocation Buffer，TLAB），分配内存的时候再TLAB上分配，互不干扰。可以通过 -XX:+/-UseTLAB 参数决定。

3）为分配的内存空间初始化零值

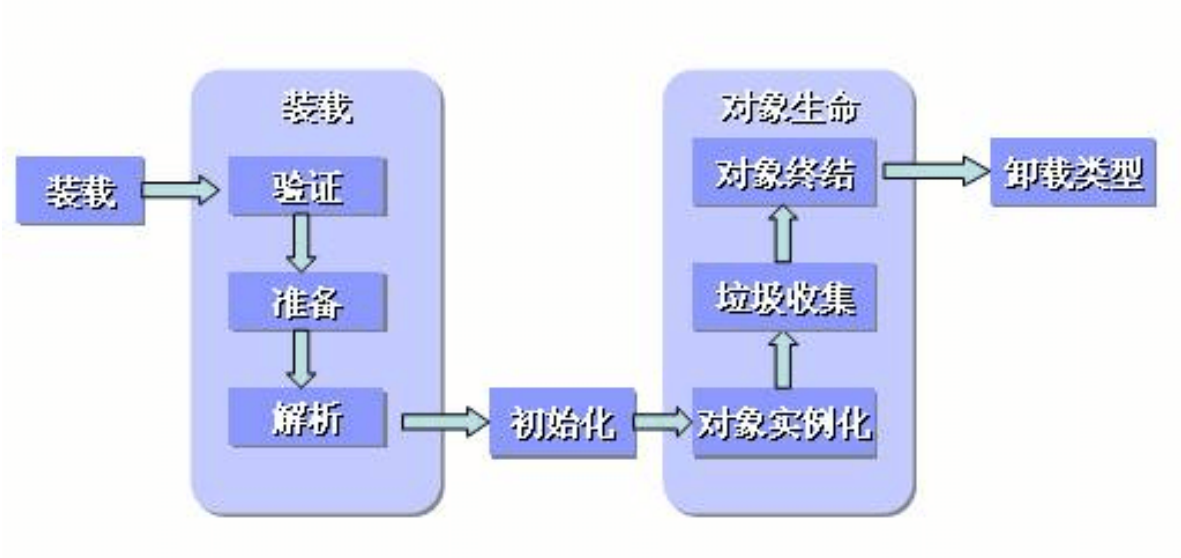
对象的内存分配完成后，还需要将对象的内存空间都初始化为零值，这样能保证对象即使没有赋初值，也可以直接使用。

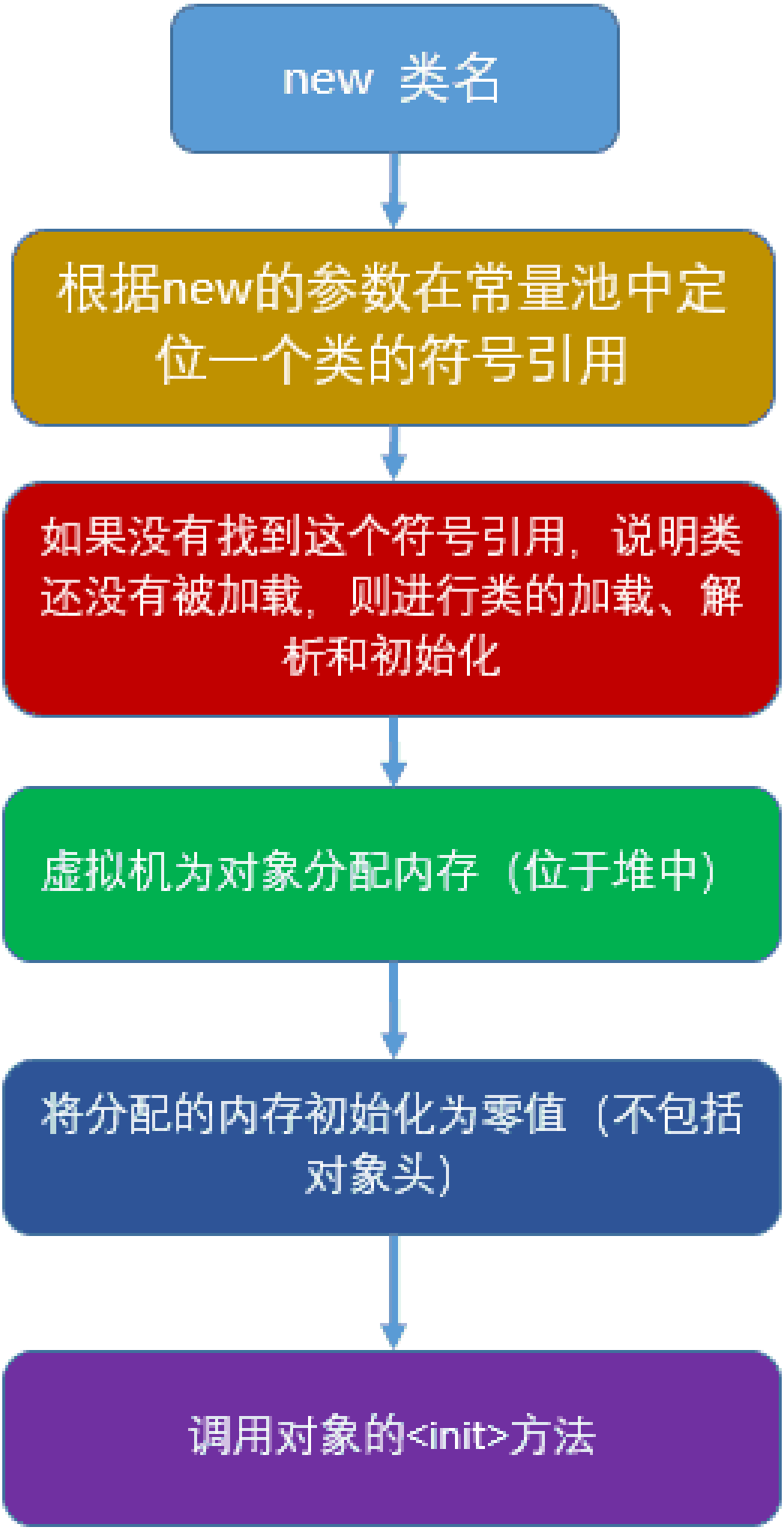
4）对对象进行其他设置

分配完内存空间，初始化零值之后，虚拟机还需要对对象进行其他必要的设置，设置的地方都在对象头中，包括这个对象所属的类，类的元数据信息，对象的 `hashcode` ，`GC` 分代年龄等信息。

5）执行 `init` 方法

执行完上面的步骤之后，在虚拟机里这个对象就算创建成功了，但是对于 `Java` 程序来说还需要执行 `init` 方法才算真正的创建完成，因为这个时候对象只是被初始化零值了，还没有真正的去根据程序中的代码分配初始值，调用了 `init` 方法之后，这个对象才真正能使用。




new 类名
根据new的参数在常量池中定位一个类的符号引用
如果没有找到这个符号引用，说明类还没有被加载，则进行类的加载、解析和初始化
虚拟机为对象分配内存（位于堆中）
将分配的内存初始化为零值（不包括对象头）
调用对象的<init>方法

4.2 内存布局

对象的内存布局包括三个部分：

- 对象头：对象头包括两部分信息。
    - 第一部分，是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，线程持有的锁等等。
    - 第二部分，是类型指针，即对象指向类元数据的指针。
- 实例数据：就是数据。
- 对齐填充：不是必然的存在，就是为了对齐。

4.3 对象的访问定位

句柄定位：Java 堆会画出一块内存来作为句柄池，reference 中存储的就是对象的句柄地址，而句柄中包含了对象实例数据与类型数据各自的具体地址信息。

直接指针访问：Java 堆对象的不居中就必须考虑如何放置访问类型数据的相关信息，而 reference 中存储的直接就是对象地址。

4.4 类加载器

```
// 什么是类加载器
```

类加载器(ClassLoader)，用来加载 Java 类到 Java 虚拟机中 ，一般来说，Java 虚拟机使用 Java 类的方式如下：Java 源程序(.java 文件)在经过 Java 编译器编译之后就被转换成 Java 字节代码(.class 文件) ，类加载器，负责读取 Java 字节代码，并转换成 java.lang.Class 类的一个实例

```
// 发生的时期
```

- 1、遇到 new、getstatic、putstatic、invokestatic 这四条字节码指令时，如果类还没进行初始化，则需要先触发其初始化。
- 2、使用 java.lang.reflect 包的方法对类进行反射调用的时候，如果类还没进行初始化，则需要先触发其初始化。
- 3、当初始化了一个类的时候，如果发现其父类还没进行初始化，则需要先触发其父类的初始化。
- 4、当虚拟机启动时，用户需要指定一个执行的主类，即调用其 #main(String[] args) 方法，虚拟机则会先初始化该主类。
- 5、当使用 JDK7 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为 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。

```
// 加载Class 的方式
```

- 第一个阶段，加载(Loading)，是找到 .class 文件并把这个文件包含的字节码加载到内存中。
- 第二阶段，连接(Linking)，又可以分为三个步骤，分别是字节码验证、Class 类数据结构分析及相应的内存分配、最后的符号表的解析。
- 第三阶段，Initialization(类中静态属性和初始化赋值)，以及Using(静态块的执行)等。

五 . GC 监控

5 .1 什么时 GC 监控

```
GC 监控是指监控 JVM 执行 GC 的过程
例如 :
    > 何时一个新生代被移动到老年代，以及其中被花费的时间
    > stop the world 何时发生，执行了多长时间

> GC 访问的接口 : GUI / CUI 两大类
    : CUI GC 监控方法使用的独立的 jstat 的 CUI 应用
    : CUI 或者在启动的时候选择JVM 参数 verbosegc
    : GUI GC 由一个单独的图形化界面完成 : jconsole ,jvisualvm , Visual GC


jstat :
参数名称见附录

-verbosegc : 启动 Java 应用时可指定
```

5 .2 常见的 GC 监控工具

- jps : 虚拟机进程状况工具
  JVM Process Status Tool ，显示指定系统内所有的HotSpot虚拟机进程。
- jstat : 虚拟机统计信息监控工具
  JVM statistics Monitoring ，是用于监视虚拟机运行时状态信息的命令，它可以显示出虚拟机进程中的类装载、内存、垃圾收集、JIT编译等运行数据。
- jinfo : Java 配置信息工具
  JVM Configuration info ，这个命令作用是实时查看和调整虚拟机运行参数。
- jmap : Java 内存映射工具
  JVM Memory Map ，命令用于生成 heap dump 文件。
- jhat : 虚拟机堆转储快照分析工具
  JVM Heap Analysis Tool ，命令是与 jmap 搭配使用，用来分析 jmap 生成的 dump 文件。jhat 内置了一个微型 的HTTP/HTML 服务器，生成 dump 的分析结果后，可以在浏览器中查看。
- jstack : Java 堆栈跟踪工具
  Java Stack Trace ，用于生成 Java 虚拟机当前时刻的线程快照。
- HSDIS : JIT 生成代码反编译

// Java 自带
- JConsole : Java 监视与管理控制台

Java Monitoring and Management Console 是从 Java5 开始，在 JDK 中自带的 Java 监控和管理控制台，用于对 JVM 中内存，线程和类等的监控。

- VisualVM ：多合一故障处理工具

  JDK 自带全能工具，可以分析内存快照、线程快照、监控内存变化、GC变化等。

  特别是 BTrace 插件，动态跟踪分析工具。

// 其他

- MAT ：内存分析工具
- [GChisto](GC 日志分析工具 -- GChisto) ：一款专业分析 GC 日志的工具。

5.3 监控常用命令

```
// 获取 Java 程序使用的内存
Runtime#freeMemory() 方法，返回剩余空间的字节数。
Runtime#totalMemory() 方法，总内存的字节数。
Runtime#maxMemory() 方法，返回最大内存的字节数。
```

六. GC 优化

6.1 GC 优化的前提

```
> GC 优化永远是最后一项任务

> 原则 ：
    > 将转移到老年代的对象数量降到最少
        ：调整新生代空间的大小。
    > 减少 Full GC 的执行时间
        ： 你需要将老年代空间设定为一个“合适”的值
```

6.2 GC 优化的方案

```
> 使用  StringBuilder 或者StringBuffer 来替代String
> 尽量少的输出日志

GC 优化考虑的参数
```

6.3 GC 优化需要考虑的参数

定义	参数	描述
堆内存空间	-Xms	Heap area size when starting JVM启动JVM时的堆内存空间。
	-Xmx	Maximum heap area size堆内存最大限制
新生代空间	-XX:NewRatio	Ratio of New area and Old area新生代和老年代的占比
	-XX:NewSize	New area size新生代空间
	-XX:SurvivorRatio	Ratio of Eden area and Survivor area伊甸园空间和幸存者空间的占比

6 . 4 GC类型可选参数

分类	参数	备注
Serial GC	-XX:+UseSerialGC	
Parallel GC	-XX:+UseParallelGC -XX:ParallelGCThreads=value	
Parallel Compacting GC	-XX:+UseParallelOldGC	
CMS GC	-XX:+UseConcMarkSweepGC -XX:+UseParNewGC -XX:+CMSParallelRemarkEnabled -XX:CMSInitiatingOccupancyFraction=value -XX:+UseCMSInitiatingOccupancyOnly	
G1	-XX:+UnlockExperimentalVMOptions -XX:+UseG1GC	在JDK6中这两个参数必须同时使用

6 . 5 GC 优化过程

```
1 > 监控 GC 状态
2 > 分析监控结果 ， 考虑是否需要GC
3 > 调整 GC 类型 ， 分配存储空间
4 > 分析结果
```

八 其他

32 位 JVM 和 64 位 JVM 的最大堆内存分别是多少

```
理论上说上 32 位的 JVM 堆内存可以到达 2^32，即 4GB
4 位 JVM 允许指定最大的堆内存，理论上可以达到 2^64
```

直接内存(堆外内存)

> 直接内存(Direct Memory)，并不是虚拟机运行时数据区的一部分，也不是 Java 虚拟机规范中农定义的内存区域
> NIO(New Input/Output) 类，引入了一种基于通道(Channel)与缓冲区（Buffer）的 I/O 方式，它可以使用 native 函数库直接分配堆外内存，然后通脱一个存储在 Java 堆中的 DirectByteBuffer 对象作为这块内存的引用进行操作

// 对比
直接内存申请空间耗费更高的性能，当频繁申请到一定量时尤为明显
直接内存 IO 读写的性能要优于普通的堆内存，在多次读写操作的情况下差异明显

附录：

Jstat 参数名称

参数名称	描述
gc	输出每个堆区域的当前可用空间以及已用空间（伊甸园，幸存者等等），GC执行的总次数，GC操作累计所花费的时间。
gccapactiy	输出每个堆区域的最小空间限制（ms）/最大空间限制（mx），当前大小，每个区域之上执行GC的次数。（不输出当前已用空间以及GC执行时间）。
gccause	输出-gcutil提供的信息以及最后一次执行GC的发生原因和当前所执行的GC的发生原因
gcnew	输出新生代空间的GC性能数据
gcnewcapacity	输出新生代空间的大小的统计数据。
gcold	输出老年代空间的GC性能数据。
gcoldcapacity	输出老年代空间的大小的统计数据。
gcpermcapacity	输出持久带空间的大小的统计数据。
gcutil	输出每个堆区域使用占比，以及GC执行的总次数和GC操作所花费的事件。

列	说明	Jstat参数
S0C	输出Survivor0空间的大小。单位KB。	-gc -gccapacity -gcnew -gcnewcapacity
S1C	输出Survivor1空间的大小。单位KB。	-gc -gccapacity -gcnew -gcnewcapacity
S0U	输出Survivor0已用空间的大小。单位KB。	-gc -gcnew
S1U	输出Survivor1已用空间的大小。单位KB。	-gc -gcnew
EC	输出Eden空间的大小。单位KB。	-gc -gccapacity -gcnew -gcnewcapacity
EU	输出Eden已用空间的大小。单位KB。	-gc -gcnew
OC	输出老年代空间的大小。单位KB。	-gc -gccapacity -gcold -gcoldcapacity
OU	输出老年代已用空间的大小。单位KB。	-gc -gcold
PC	输出持久代空间的大小。单位KB。	-gc -gccapacity -gcold -gcoldcapacity -gcpermcapacity
PU	输出持久代已用空间的大小。单位KB。	-gc -gcold
YGC	新生代空间GC时间发生的次数。	-gc -gccapacity -gcnew -gcnewcapacity -gcold -gcoldcapacity -gcpermcapacity -gcutil -gccause
YGCT	新生代GC处理花费的时间。	-gc -gcnew -gcutil -gccause
FGC	full GC发生的次数。	-gc -gccapacity -gcnew -gcnewcapacity -gcold -gcoldcapacity -gcpermcapacity -gcutil -gccause
FGCT	full GC操作花费的时间	-gc -gcold -gcoldcapacity -gcpermcapacity -gcutil -gccause
GCT	GC操作花费的总时间。	-gc -gcold -gcoldcapacity -gcpermcapacity -gcutil -gccause
NGCMN	新生代最小空间容量，单位KB。	-gccapacity -gcnewcapacity
NGCMX	新生代最大空间容量，单位KB。	-gccapacity -gcnewcapacity
NGC	新生代当前空间容量，单位KB。	-gccapacity -gcnewcapacity

列	说明	Jstat参数
OGCMN	老年代最小空间容量，单位KB。	-gccapacity -gcoldcapacity
OGCMX	老年代最大空间容量，单位KB。	-gccapacity -gcoldcapacity
OGC	老年代当前空间容量制，单位KB。	-gccapacity -gcoldcapacity
PGCMN	持久代最小空间容量，单位KB。	-gccapacity -gcpermcapacity
PGCMX	持久代最大空间容量，单位KB。	-gccapacity -gcpermcapacity
PGC	持久代当前空间容量，单位KB。	-gccapacity -gcpermcapacity
PC	持久代当前空间大小，单位KB	-gccapacity -gcpermcapacity
PU	持久代当前已用空间大小，单位KB	-gc -gcold
LGCC	最后一次GC发生的原因	-gccause
GCC	当前GC发生的原因	-gccause
TT	老年化阈值。被移动到老年代之前，在新生代空存活的次数。	-gcnew
MTT	最大老年化阈值。被移动到老年代之前，在新生代空存活的次数。	-gcnew
DSS	幸存者区所需空间大小，单位KB。	-gcnew

参考文档

http://cmsblogs.com/?p=5140