码农求职小助手：Java虚拟机高频面试题

笔记本： 4-Java虚拟机
创建时间： 2019/9/7 21:32 **更新时间：** 2019/9/7 21:32
作者： pc941206@163.com

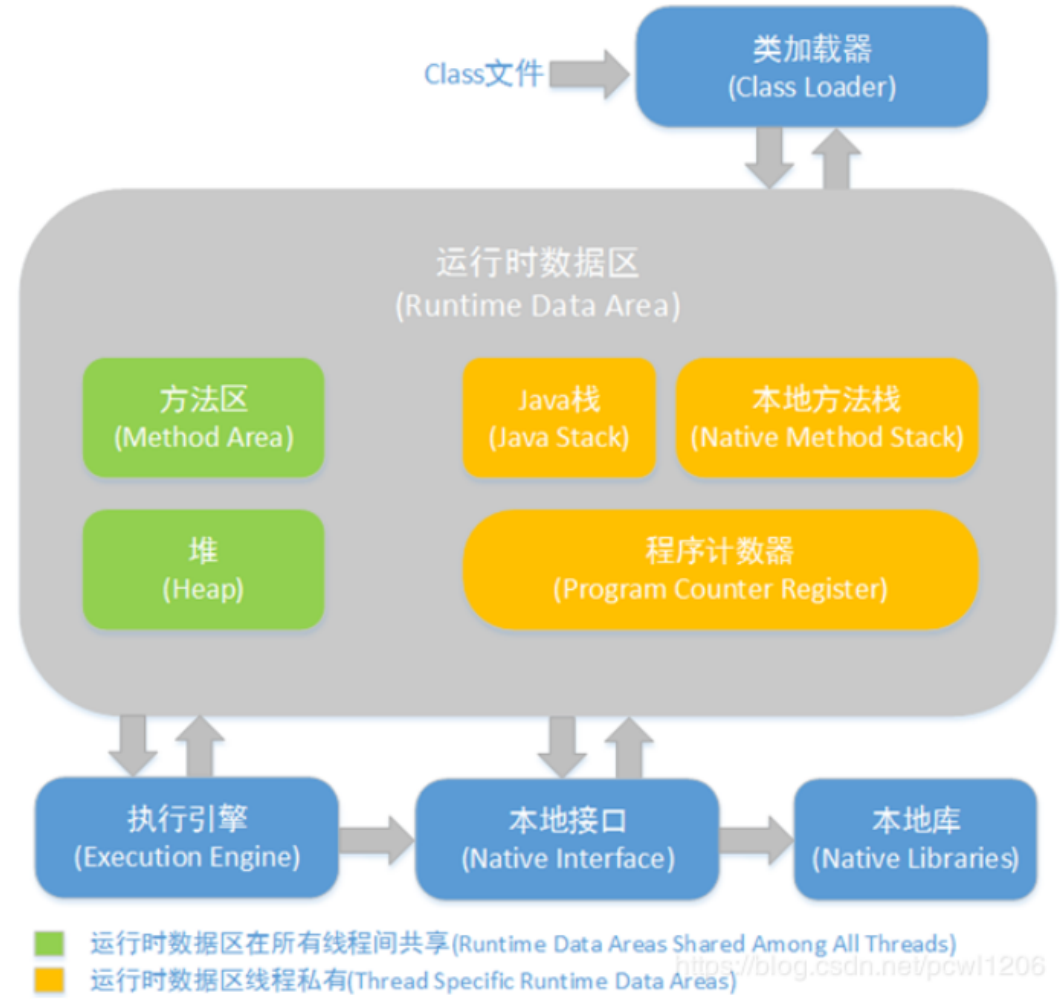
- 一、运行时数据区
  - 1、数据区基本介绍
    - 1.1 程序计数器
    - 1.2 Java 虚拟机栈
    - 1.3 本地方法栈
    - 1.4 Java 堆
    - 1.5 方法区
    - 运行时常量池
    - 1.6 直接内存
  - 2、堆和栈的细节说明
    - 2.1 数据类型
    - 2.2 栈是运行时的单位，而堆是存储的单位
    - 2.3 为什么要把堆和栈区分出来呢？栈中不是也可以存储数据吗？
    - 2.4 堆中存什么？栈中存什么？
    - 2.5 Java 中的参数传递时传值呢？还是传引用？
    - 2.6 Java 对象的大小
  - 3、对象的访问定位的两种方式
    - 3.1 使用句柄
    - 3.2 直接指针
    - 3.3 各自的优点
- 二、哪些垃圾需要回收
  - 2.1 引用计数法
    - 2.1.1 算法分析
    - 2.1.2 优缺点
    - 2.1.3 代码展示
  - 2.2 可达性分析算法
    - 垃圾回收从哪儿开始的呢？
  - 2.3 Java 中的引用你了解多少
    - 2.3.1 强引用
    - 2.3.2 软引用
    - 2.3.3 弱引用
    - 2.3.4 虚引用
  - 2.4 对象死亡（被回收）前的最后一次挣扎
  - 2.5 方法区如何判断是否需要回收
  - 2.6 内存泄漏

        - 内存泄漏的概念
        - Java 内存泄漏的根本原因?
        - 可能发生内存泄漏的情况
        - 尽量避免内存泄漏的方法?
- 三、常用的垃圾收集算法
    - 3.1 标记-清除算法（Mark-Sweep）
    - 3.2 复制算法(Copying)
    - 3.3 标记-整理算法(Mark-compact)
    - 3.4 分代收集算法
        - 为什么要分代?
        - 3.4.1 年轻代（Young Generation）的回收算法 (回收主要以 Copying 为主)
        - 3.4.2 年老代（Old Generation）的回收算法（回收主要以 Mark-Compact 为主）
        - 3.4.3 永久代（Permanent Generation）的回收算法
    - 3.5 其他问题
        - 3.5.1 如何处理碎片?
        - 3.5.2 浮动垃圾
- 四、常见的垃圾收集器
    - 4.1 Serial 收集器（复制算法)
    - 4.2 Serial Old 收集器(标记-整理算法)
    - 4.3 ParNew 收集器(停止-复制算法)
    - 4.4 Parallel Scavenge 收集器(停止-复制算法)
    - 4.5 Parallel Old 收集器(停止-复制算法)
    - 4.6 CMS(Concurrent Mark Sweep)收集器（标记-清除算法）
    - 4.7 G1 收集器【重点】
- 五、内存分配与回收策略
    - 5.1 GC 的触发时机【重点】
        - 1. Minor / Scavenge GC
        - 2. Full GC
    - 5.2 内存分配
- 六、JVM 监控工具
    - JDK 的可视化工具
    - 1、jvisualvm：虚拟机监视和故障处理平台
    - 2、JConsole
    - 3、JProfiler
    - 4、如何利用监控工具调优【结合具体的工具看下】
        - 4.1、堆信息查看
        - 4.2、线程监控
        - 4.3、热点分析
        - 4.4、快照
        - 4.5 内存泄露检查
    - JDK 的命令行工具
    - 1、jps： 查看当前 Java 进程

    - 2、jstat：显示虚拟机运行数据
    - 3、jmap：内存监控
    - 4、jhat：分析 heapdump 文件
    - 5、jstack：线程快照
    - 6、jinfo：虚拟机配置信息
    - 7、相关参数说明
        - 堆设置
        - 收集器设置
        - 垃圾回收统计信息
        - 并行收集器设置
        - 并发收集器设置
- 七、JVM 调优
    - 1、年轻代大小选择
    - 2、老年代大小选择
    - 3、吞吐量优先的应用
    - 4、较小堆引起的碎片问题
- 八、类文件结构
- 九、虚拟机类加载机制
    - 1、类加载各阶段作用
    - 2、类与类加载器【重要】
    - 3、双亲委派模型【非常重要】
        - 3.1 双亲委派模型的工作过程：
        - 3.2 使用双亲委派模型的好处：
        - 3.3 双亲委派模型的代码实现：
        - 3.4 怎么打破双亲委派模型机制？
        - 3.5 有哪些场景是需要打破双亲委派模型的？
- 十、虚拟机字节码执行引擎
    - 1、运行时栈帧结构
    - 2、方法调用
- 十一、早期优化和晚期优化
    - 1 早期优化
        - 1.1 解析与填充符号表
    - 2 晚期优化
        - 2.1 解释器与编译器
        - 2.2 编译对象与触发条件
        - 2.2 编译优化技术
- 十二、Java 内存模型
    - 1、概述
    - 2、Java 内存模型
        - 2.1 主内存与工作内存
        - 2.2 内存间的交互操作
- 十三、其他面试题
    - 1、说一下 Jvm 的主要组成部分？及其作用？
    - 2、说一下堆栈的区别？

一、运行时数据区

Java虚拟机在执行Java程序的过程中会把它所管理的内存划分为若干个不同的数据区域。这些区域都有各自的用途，以及创建和销毁时间，有的区域随着虚拟机进程的启动而存在，有些区域则依赖用户线程的启动和结束而建立和销毁。Java 虚拟机运行时数据区如下图所示：



其中方法区和堆是由所有线程共享的数据区，而Java虚拟机栈、本地方法栈和程序计数器是线程隔离的数据区。

Java虚拟机栈内存结构中的程序计数器、虚拟机栈和本地方法栈这三个区域随线程创建而生，随线程销毁而死，因此这三个区域的内存分配和回收是确定的，Java垃圾收集器重点关注的是Java虚拟机的堆内存和方法区内存。

1、数据区基本介绍

1.1 程序计数器

程序计数器（Program Counter Register）：是一块较小的内存空间，它可以看作是当前线程所执行的字节码的行号指示器。

字节码解释器工作时就是通过改变这个计数器的值来选取下一条需要执行的字节码指令。程序的分支、循环、跳转、异常处理、线程恢复等基础功能都需要依赖这个计数器来完成。

由于 Java 虚拟机的多线程是通过线程轮流切换并分配处理器执行时间的方式来实现的，在任何一个确定的时刻，一个处理器都只会执行一条线程中的命令。因此，为了线程

切换后能恢复到正确的执行位置，每条线程都需要有一个独立的程序计数器，各线程之间的计数器互不影响，独立存储，我们程这块内存区域为“线程私有”的内存。

此区域是唯一 一个虚拟机规范中没有规定任何 OutOfMemoryError 情况的区域。

1.2 Java 虚拟机栈

Java 虚拟机栈（Java Virtual Machine Stacks）：描述的是Java方法执行的内存模型：每个方法在执行的同时都会创建一个帧栈（Stack Frame）用于存储局部变量表、操作数栈、动态链接、方法出口等信息。每一个方法从调用直至执行完成的过程，就对应着一个栈帧在虚拟机栈中入栈到出栈的过程。它的线程也是私有的，生命周期与线程相同。

局部变量表存放了编译期可知的各种基本数据类型（boolean、byte、char、short、int、float、long、double）、对象引用和 returnAddress 类型（指向了一条字节码指令的地址）。

Java 虚拟机栈的局部变量表的空间单位是槽（Slot），其中64位长度的double 和 long 类型会占用两个Slot。局部变量表所需内存空间在编译期完成分配，当进入一个方法时，该方法需要在帧中分配多大的局部变量是完全确定的，在方法运行期间不会改变局部变量表的大小。

Java虚拟机栈有两种异常状况：如果线程请求的栈的深度大于虚拟机所允许的深度，将抛出StackOverflowError异常；如果扩展时无法申请到足够的内存，就会抛出OutOfMemoryError异常。

1.3 本地方法栈

本地方法栈（Native Method Stack）：与虚拟机栈所发挥的作用是非常相似的，它们之间的区别只不过是虚拟机栈为虚拟机执行Java方法（也就是字节码）服务，而本地方法栈则为虚拟机使用到的 Native 方法服务。

Java 虚拟机规范没有对本地方法栈中方法使用的语言、使用的方式和数据结构做出强制规定，因此具体的虚拟机可以自由地实现它。比如：Sun HotSpot 虚拟机直接把Java虚拟机栈和本地方法栈合二为一。

与Java虚拟机栈一样，本地方法栈也会抛出StackOverflowError和OutOfMemoryError异常。

1.4 Java 堆

Java堆（Java Heap）：是被所有线程所共享的一块内存区域，在虚拟机启动时创建。此内存区域的唯一目的就是：存放对象实例，几乎所有的对象实例都在这里分配内存。

Java 堆是垃圾收集器管理的主要区域，因此很多时候也被称做“GC”堆（Garbage Collected Heap）。从内存回收的角度看，由于现在收集器基本都采用分代收集算法，所以Java 堆中还可以细分为：新生代和老年代。从内存分配角度来看，线程共享的 Java 堆

中可能划分出多个线程私有的分配缓冲区（Thread Local Allocation Buffer, TLAB）。不过无论如何划分，都与存放的内容无关，无论哪个区域，存储的都仍然是对象实例，进一步划分的目的是为了更好地回收内存，或者更快地分配内存。

Java 虚拟机规定，Java 堆可以处于物理上不连续的内存空间中，只要逻辑上是连续的即可。在实现时，可以是固定大小的，也可以是可扩展的。如果在堆中没有完成实例分配。并且堆也无法扩展时，将会抛出 OutOfMemoryError 异常。

1.5 方法区

方法区（Method Area）：与 Java 堆一样，是各个线程共享的内存区域，它用于存储已被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。

虽然 Java 虚拟机规范把方法区描述为堆的一个逻辑部分，但是它却有一个别名叫做 Non-Heap（非堆），其目的应该就是与 Java 堆区分开来。

Java 虚拟机规范对方法区的限制非常宽松，除了和 Java 堆一样不需要连续的内存和可以选择固定大小或者可扩展外，还可以选择不实现垃圾收集。这个区域的内存回收目标主要是针对常量池的回收和对类型的卸载。

根据Java虚拟机规范规定，当方法区无法满足内存分配需求时，将抛出OutOfMemoryError异常。

- **运行时常量池**

运行时常量池（Runtime Constant Pool）：是方法区的一部分。Class 文件中除了有类的版本、字段、方法、接口等描述信息外，还有一些信息是常量池，用于存放编译期生成的各种字面量和符号引用，这部分内容将在类加载后进入方法区的运行时常量池中存放。

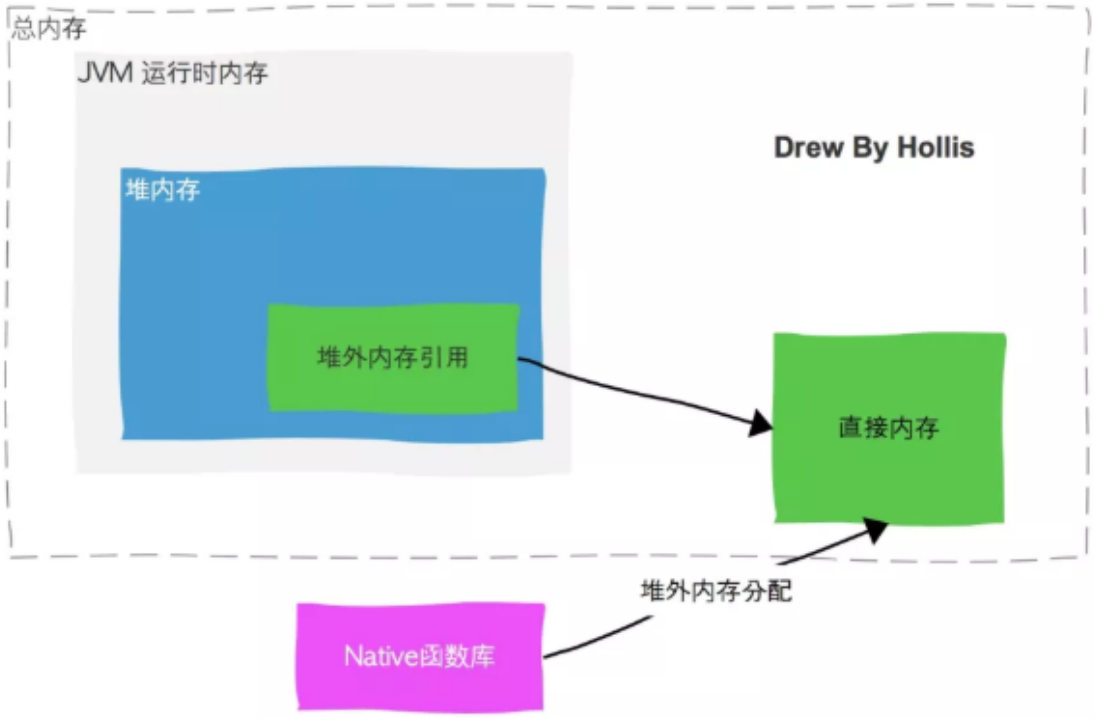
Java 虚拟机对 Class 文件每一部分（自然也包括常量池）的格式都有严格的规定，每一个字节用于存储哪种数据都必须符合规范上的要求才会被虚拟机认可、装载和执行。

1.6 直接内存

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 虚拟机规范中定义的内存区域。但是这部分内存也频繁地使用，而且也可能导致 OutOfMemoryError 异常。将其放到这里一起进行讲解。

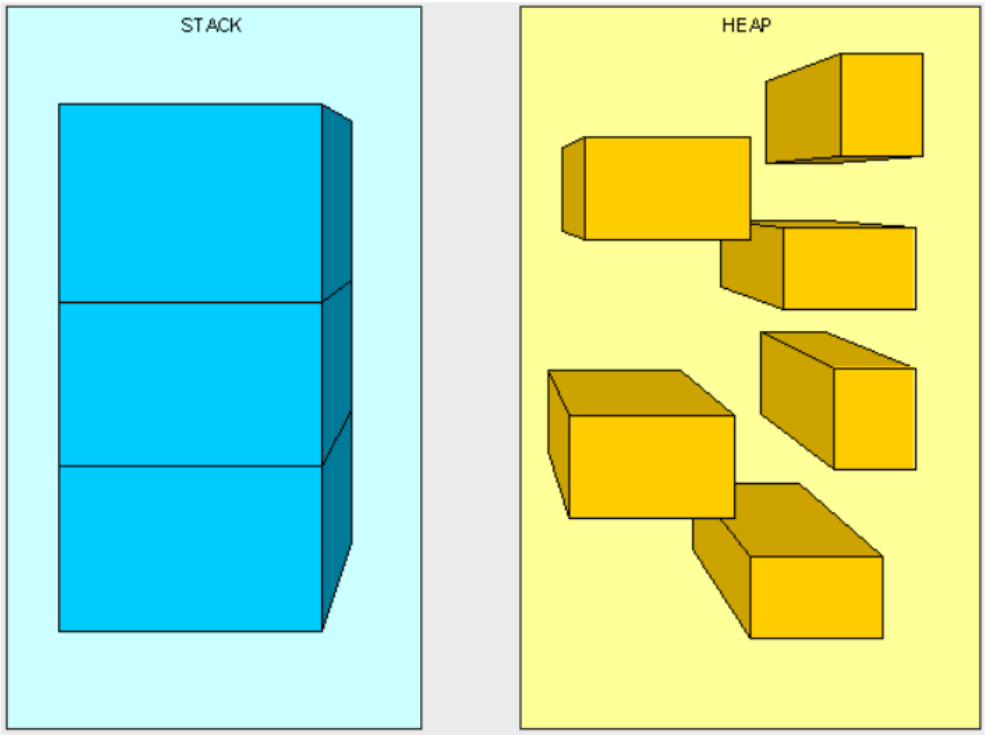
在 JDK1.4 中新加入了 NIO(New Input/Output) 类，引入了一种基于通道（Channel）与缓冲区（Buffer）的 I/O 方式，它可以使用Native函数库直接分配堆外内存，然后通过一个存储在Java堆中的DirectByteBuffer对象作为这块内存的引用进行操作。

本地直接内存的分配不会受到Java堆大小的限制，但是，既然是内存，肯定还是会受到本机总内存大小以及处理器寻址空间的限制。如果各个内存区域总和大于物理内存限制，从而导致动态扩展时出现 OutOfMemoryError 异常。



2、堆和栈的细节说明

堆和栈是程序运行的关键，很有必要把他们的关系说清楚。



> **堆和栈的区别是什么?**
>
> 堆和栈（虚拟机栈）是完全不同的两块内存区域，一个是线程独享的，一个是线程共享的。二者之间最大的区别就是存储的内容不同：**堆中主要存放对象实例。 栈（局部变量表）中主要存放各种基本数据类型、对象的引用。**

2.1 数据类型

Java 虚拟机中，数据类型可以分为两类：基本类型和引用类型。基本类型的变量保存原始值，即：它代表的值就是数值本身；而引用类型的变量保存引用值。“引用值”代表了某个对象的引用，而不是对象本身，对象本身存放在这个引用值所表示的地址的位置。

> 基本类型包括：byte、short、int、long、char、float、double、Boolean、returnAddress
>
> 引用类型包括：类类型、接口类型和数组

2.2 栈是运行时的单位，而堆是存储的单位

栈解决程序的运行问题，即程序如何执行，或者说如何处理数据。堆解决的是数据存储的问题，即数据怎么放、放在哪儿。

在 Java 中一个线程就会相应有一个线程栈与之对应，这点很容易理解，因为不同的线程执行逻辑有所不同，因此需要一个独立的线程栈。而堆则是所有线程共享的。栈因为是运行单位，因此里面存储的信息都是跟当前线程（或程序）相关信息的。包括局部变量、程序运行状态、方法返回值等等；而堆只负责存储对象信息。

2.3 为什么要把堆和栈区分出来呢？栈中不是也可以存储数据吗？

第一，从软件设计的角度看，**栈代表了处理逻辑**，而**堆代表了数据。**这样分开，使得处理逻辑更为清晰。**分而治之**的思想。这种隔离、模块化的思想在软件设计的方方面面都有体现。

第二，堆与栈的分离，使得堆中的内容可以被多个栈**共享**（也可以理解为多个线程访问同一个对象）。这种共享的收益是很多的。一方面这种共享提供了一种有效的数据交互方式(如：共享内存)，另一方面，堆中的共享常量和缓存可以被所有栈访问，节省了空间。

第三，栈因为运行时的需要，比如保存系统运行的上下文，需要进行地址段的划分。由于栈只能向上增长，因此就会限制住栈存储内容的能力。而堆不同，堆中的对象是可以根据需要动态增长的，因此栈和堆的拆分，使得**动态增长成为可能**，相应栈中只需记录堆中的一个地址即可。

第四，**面向对象就是堆和栈的完美结合。**其实，面向对象方式的程序与以前结构化的程序在执行上没有任何区别。但是，面向对象的引入，使得对待问题的思考方式发生了改变，而更接近于自然方式的思考。当我们把对象拆开，你会发现，对象的属性其实就是数据，存放在堆中；而对象的行为（方法），就是运行逻辑，放在栈中。我们在编写对象的时候，其实即编写了数据结构，也编写的处理数据的逻辑。不得不承认，面向对象的设计，确实很美。

2.4 堆中存什么？栈中存什么？

堆中存的是对象。栈中存的是基本数据类型和堆中对象的引用。一个对象的大小是不可估计的，或者说是可以动态变化的，但是在栈中，一个对象只对应了一个4btye的引用（堆栈分离的好处）。

- **为什么不把基本类型放堆中呢?**

因为其占用的空间一般是1~8个字节——需要空间比较少，而且因为是基本类型，所以不会出现动态增长的情况——长度固定，因此栈中存储就够了，如果把它存在堆中是没有什么意义的（还会浪费空间，后面说明）。可以这么说，基本类型和对象的引用都是存放在栈中，而且都是几个字节的一个数，因此在程序运行时，它们的处理方式是统一的。但是基本类型、对象引用和对象本身就有所区别了，因为一个是栈中的数据一个是堆中的数据。最常见的一个问题就是，Java 中参数传递时的问题。

2.5 Java 中的参数传递时传值呢？还是传引用?

要说明这个问题，先要明确两点：

> 1. 不要试图与 C 进行类比，Java 中没有指针的概念。
> 2. 程序运行永远都是在栈中进行的，因而参数传递时，只存在传递基本类型和对象引用的问题。不会直接传对象本身。

明确以上两点后：Java在方法调用传递参数时，因为没有指针，所以它都是进行传值调用（这点可以参考 C 的传值调用）。因此，很多书里面都说 Java 是进行传值调用，这点没有问题，而且也简化的 C 中复杂性。

但是传引用的错觉是如何造成的呢？在运行栈中，**基本类型和引用的处理是一样的，都是传值**，所以，如果是传引用的方法调用，也同时可以理解为“传引用值”的传值调用，即引用的处理跟基本类型是完全一样的。但是当进入被调用方法时，被传递的这个引用的值，被程序解释（或者查找）到堆中的对象，这个时候才对应到真正的对象。如果此时进行修改，修改的是引用对应的对象，而不是引用本身，即：修改的是堆中的数据。所以这个修改是可以保持的了。

对象，从某种意义上说，是由基本类型组成的。**可以把一个对象看作为一棵树，对象的属性如果还是对象，则还是一颗树（即非叶子节点），基本类型则为树的叶子节点。**程序参数传递时，被传递的值本身都是不能进行修改的，但是，如果这个值是一个非叶子节点（即一个对象引用），则可以修改这个节点下面的所有内容。

堆和栈中，栈是程序运行最根本的东西。程序运行可以没有堆，但是不能没有栈。而堆是为栈进行数据存储服务，说白了堆就是一块共享的内存。不过，正是因为堆和栈的分离的思想，才使得Java的垃圾回收成为可能。

Java中，栈的大小通过 -Xss 来设置，当栈中存储数据比较多时，需要适当调大这个值，否则会出现 java.lang.StackOverflowError 异常。常见的出现这个异常的是无法返回的递归，因为此时栈中保存的信息都是方法返回的记录点。

2.6 Java 对象的大小

基本数据的类型的大小是固定的，这里就不多说了。对于非基本类型的 Java 对象，其大小就值得商榷。在 Java 中，一个空 Object 对象的大小是 8 byte，这个大小只是保存堆中一个没有任何属性的对象的大小。看下面语句：

```
Object ob = new Object();
```

这样在程序中完成了一个 Java 对象的生命，但是它所占的空间为：4 byte + 8 byte。4 byte 是上面部分所说的 Java 栈中保存引用的所需要的空间。而那 8 byte 则是 Java 堆中对象的信息。因为所有的 Java 非基本类型的对象都需要默认继承 Object 对象，因此不论什么样的 Java 对象，其大小都必须是大于 8 byte。有了 Object 对象的大小，我们就可以计算其他对象的大小了。

```
Class NewObject {

        int count;

        boolean flag;

        Object ob;
}
```

其大小为：空对象大小(8 byte) + int 大小(4 byte) + Boolean 大小(1 byte)+空 Object 引用的大小(4 byte) = 17byte。但是因为 Java 在对对象内存分配时都是以 8 的整数倍来分，因此大于 17 byte 的最接近 8 的整数倍的是 24，因此此对象的大小为 24 byte。

这里需要注意一下基本类型的包装类型的大小。因为这种包装类型已经成为对象了，因此需要把它们作为对象来看待。包装类型的大小至少是12 byte（声明一个空 Object 至少需要的空间），而且 12 byte 没有包含任何有效信息，同时，因为 Java 对象大小是 8 的整数倍，因此一个基本类型包装类的大小至少是 16 byte。这个内存占用是很恐怖的，它是使用基本类型的 N 倍（N > 2），有些类型的内存占用更是夸张（随便想下就知道了）。因此，可能的话应尽量少使用包装类。在 JDK5.0 以后，因为加入了自动类型装换，因此，Java 虚拟机会在存储方面进行相应的优化。

3、对象的访问定位的两种方式

Java 程序通过栈上的引用数据来操作堆上的具体对象。目前主流的对象访问方式有：句柄 和 直接指针。

3.1 使用句柄

如果使用句柄的话，那么 Java 堆中将会划分出一块内存来作为句柄池，引用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。

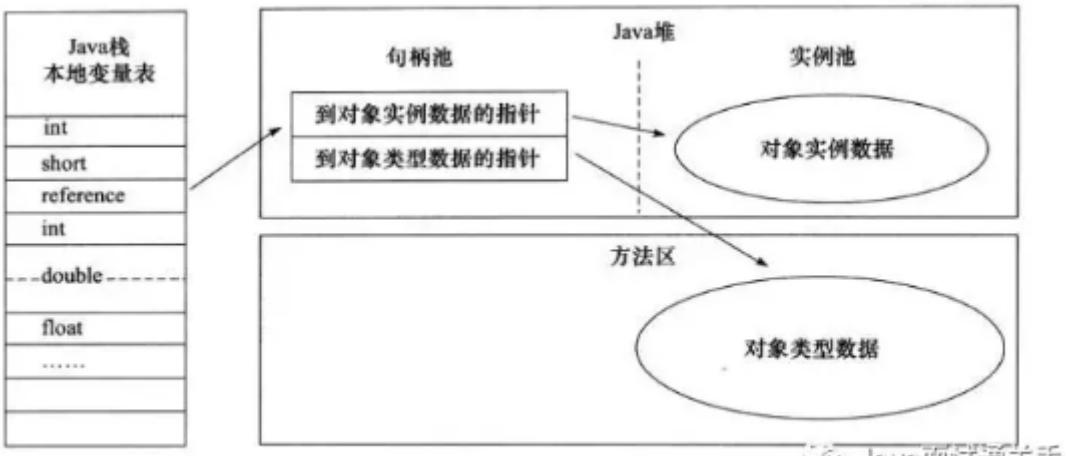


图 2-2　通过句柄访问对象

3.2 直接指针

如果使用直接指针访问，那么 Java 堆对象的布局中就必须考虑如何防止访问类型数据的相关信息，reference 中存储的直接就是对象的地址。

3.3 各自的优点

1、使用句柄来访问的最大好处是引用中存储的是稳定的句柄地址，在对象被移动时只会改变句柄中的实例数据指针，而引用本身不需要修改；

2、使用直接指针访问方式最大的好处就是速度快，它节省了一次指针定位的时间开销。

二、哪些垃圾需要回收

大家都知道JVM的内存结构包括五大区域：**程序计数器、虚拟机栈、本地方法栈、堆区、方法区。**其中程序计数器、虚拟机栈、本地方法栈3个区域随线程而生、随线程而灭，因此这几个区域的内存分配和回收都具备确定性，就不需要过多考虑回收的问题，因为方法结束或者线程结束时，内存自然就跟随着回收了。而Java堆区和方法区则不一样!这部分内存的分配和回收是动态的，正是垃圾收集器所需关注的部分。

垃圾收集器在对堆区和方法区进行回收前，首先要确定这些区域的对象哪些可以被回收，哪些暂时还不能回收，这就要用到判断对象是否存活的算法!

2.1 引用计数法

2.1.1 算法分析

引用计数是垃圾收集器中的早期策略。在这种方法中，堆中每个对象实例都有一个引用计数。当一个对象被创建时，就将该对象实例分配给一个变量，该变量计数设置为1。当任何其它变量被赋值为这个对象的引用时，计数加1（a = b,则b引用的对象实例的计数器+1），但当一个对象实例的某个引用超过了生命周期或者被设置为一个新值时，对象实例的引用计数器减1。任何引用计数器为0的对象实例可以被当作垃圾收集。当一个对象实例被垃圾收集时，它引用的任何对象实例的引用计数器减1。

2.1.2 优缺点

优点：引用计数收集器可以很快的执行，交织在程序运行中。对程序需要不被长时间打断的实时环境比较有利。

缺点：无法检测出**循环引用**。如父对象有一个对子对象的引用，子对象反过来引用父对象。这样，他们的引用计数永远不可能为0。

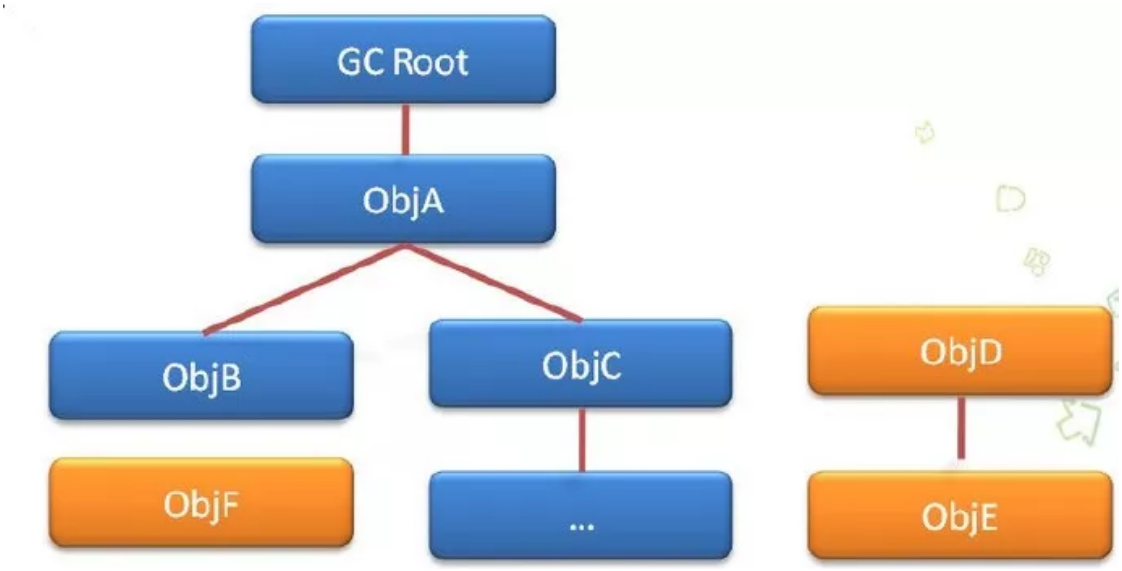
2.1.3 代码展示

```
public  class   demo{

    public  static  void   main(String[]   args){
        MyObject    object1 = new   MyObject();
        MyObject    object2 = new   MyObject();

        object1.object = object2;
        object2.object = object1;

        object1 = null;
        object2 = null;
    }
}

class   MyObject{

   MyObject    object;
}
```

这段代码是用来验证引用计数算法不能检测出循环引用。最后面两句将 object1 和 object2 赋值为null，也就是说 object1 和 object2 指向的对象已经不可能再被访问，但是由于它们互相引用对方，导致它们的引用计数器都不为 0，那么垃圾收集器就永远不会回收它们。

2.2 可达性分析算法

可达性分析算法是从离散数学中的图论引入的，程序把所有的引用关系看作一张图，从一个节点 GC ROOT 开始，寻找对应的引用节点，找到这个节点以后，继续寻找这个节点的引用节点，当所有的引用节点寻找完毕之后，剩余的节点则被认为是没有被引用到的节点，即无用的节点，无用的节点将会被判定为是可回收的对象。

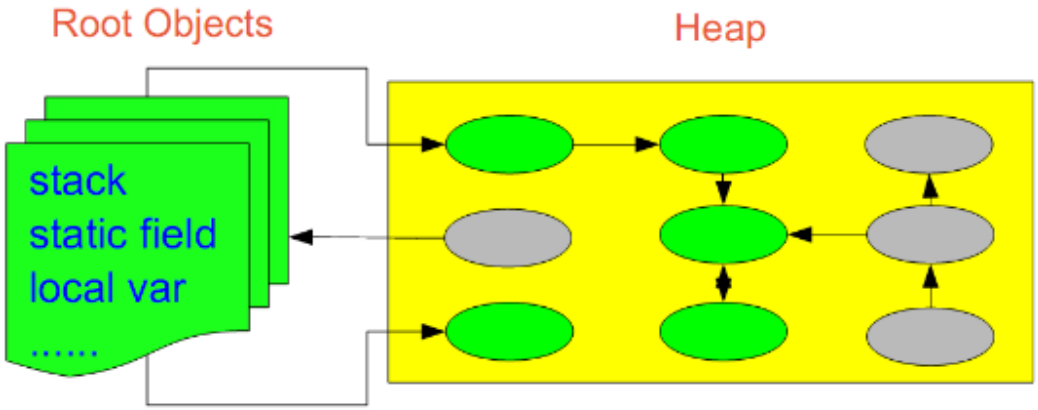


在 Java 语言中，可作为 GC Roots 的对象包括下面几种：

> a) 虚拟机栈中引用的对象（栈帧中的本地变量表）；
>
> b) 方法区中类静态属性引用的对象；
>
> c) 方法区中常量引用的对象；
>
> d) 本地方法栈中 JNI（Native方法）引用的对象。

- **垃圾回收从哪儿开始的呢?**

即，从哪儿开始查找哪些对象是正在被当前系统使用的。上面分析的堆和栈的区别，其中栈是真正进行程序执行地方，所以要获取哪些对象正在被使用，则需要**从 Java 栈开始。**同时，一个栈是与一个线程对应的，因此，如果有多个线程的话，则必须对这些线程对应的所有的栈进行检查。



同时，除了栈外，还有系统运行时的寄存器等，也是存储程序运行数据的。这样，**以栈或寄存器中的引用为起点，我们可以找到堆中的对象，又从这些对象找到对堆中其他对象的引用，这种引用逐步扩展，最终以 null 引用或者基本类型结束，这样就形成了一颗以 Java 栈中引用所对应的对象为根节点的一颗对象树。如果栈中有多个引用，则最终会形成多颗对象树。**在这些对象树上的对象，都是当前系统运行所需要的对象，不能被垃圾回收。而其他剩余对象，则可以视为无法被引用到的对象，可以被当做垃圾进行回收。因此，垃圾回收的起点是一些根对象（java栈, 静态变量, 寄存器...）。

2.3 Java 中的引用你了解多少

无论是通过引用计数算法判断对象的引用数量，还是通过可达性分析算法判断对象的引用链是否可达，判定对象是否存活都与“引用”有关。在Java语言中，将引用又分为**强引用、软引用、弱引用、虚引用**4种，这四种引用强度依次逐渐减弱。

2.3.1 强引用

在程序代码中普遍存在的，类似 Object obj = new Object() 这类引用，只要强引用还存在，垃圾收集器永远不会回收掉被引用的对象。

2.3.2 软引用

用来描述一些还有用但并非必须的对象。对于软引用关联着的对象，在系统将要发生内存溢出异常之前，将会把这些对象列进回收范围之中进行第二次回收。如果这次回收后还没有足够的内存，才会抛出内存溢出异常。

2.3.3 弱引用

也是用来描述非必需对象的，但是它的强度比软引用更弱一些，被弱引用关联的对象只能生存到下一次垃圾收集发生之前。当垃圾收集器工作时，无论当前内存是否足够，都会回收掉只被弱引用关联的对象。

2.3.4 虚引用

也叫幽灵引用或幻影引用，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构成影响，也无法通过虚引用来取得一个对象实例。它的作用是能在这个对象被收集器回收时收到一个系统通知。

罗列这四个概念的目的是为了说明，**无论引用计数算法还是可达性分析算法都是基于强引用而言的。**

2.4 对象死亡（被回收）前的最后一次挣扎

即使在可达性分析算法中不可达的对象，也并非是“非死不可”，这时候它们暂时处于“缓刑”阶段，**要真正宣告一个对象死亡，至少要经历两次标记过程。**

第一次标记： 如果对象在进行可达性分析后发现没有与 GC Roots 相连接的引用链，那它将会被第一次标记；

第二次标记： 第一次标记后接着会进行一次筛选，筛选的条件是此对象是否有必要执行 finalize() 方法。在 finalize() 方法中没有重新与引用链建立关联关系的，将被进行第二次标记。

第二次标记成功的对象将真的会被回收，如果对象在 finalize() 方法中重新与引用链建立了关联关系，那么将会逃离本次回收，继续存活。

2.5 方法区如何判断是否需要回收

方法区存储内容是否需要回收的判断可就不一样了。方法区主要回收的内容有：**废弃常量** 和 **无用的类。**

对于废弃常量也可通过引用的可达性来判断。

但是对于无用的类则需要同时满足下面3个条件：

> 1、该类所有的实例都已经被回收，也就是 Java 堆中不存在该类的任何实例；
>
> 2、加载该类的 ClassLoader 已经被回收；
>
> 3、该类对应的 java.lang.Class 对象没有在任何地方被引用，无法在任何地方通过反射访问该类的方法。

关于类加载的原理，也是阿里面试的主角，面试官也问过我比如：能否自己定义 String，答案是不行，因为 jvm 在加载类的时候会执行双亲委派。

2.6 内存泄漏

https://blog.csdn.net/anxpp/article/details/51325838

- **内存泄漏的概念**

在 Java 中，内存泄漏就是存在一些不会再被使用确没有被回收的对象，这些对象有下面两个特点：

1、这些对象是可达的，即在有向图中，存在通路可以与其相连；

2、这些对象是无用的，即程序以后不会再使用这些对象。

如果对象满足这两个条件，这些对象就可以判定为 Java 中的内存泄漏，这些对象不会被 GC 所回收，然而它却占用内存。

- **Java 内存泄漏的根本原因?**

长生命周期的对象持有短生命周期对象的引用就很可能发生内存泄漏，尽管短生命周期对象已经不再需要，但是因为长生命周期持有它的引用而导致不能被回收，这就是 Java 中内存泄漏的发生场景。

- **可能发生内存泄漏的情况**

1、静态集合类引起的内存泄漏

2、当集合里面的对象属性被修改后，再调用 remove() 方法时不起作用

3、监听器：释放对象的时候没有删除监听器

4、各种连接：比如数据库连接（dataSourse.getConnection()），网络连接(socket) 和 io 连接，除非其显式的调用了其 close() 方法将其连接关闭，否则是不会自动被 GC 回收的

5、内部类：内部类的引用是比较容易遗忘的一种，而且一旦没释放可能导致一系列的后继类对象没有释放

6、单例模式：单例对象在初始化后将在 JVM 的整个生命周期中存在（以静态变量的方式），如果单例对象持有外部的引用，那么这个对象将不能被 JVM 正常回收，导致内存泄漏。

- **尽量避免内存泄漏的方法?**

1、尽量不要使用 static 成员变量，减少生命周期；

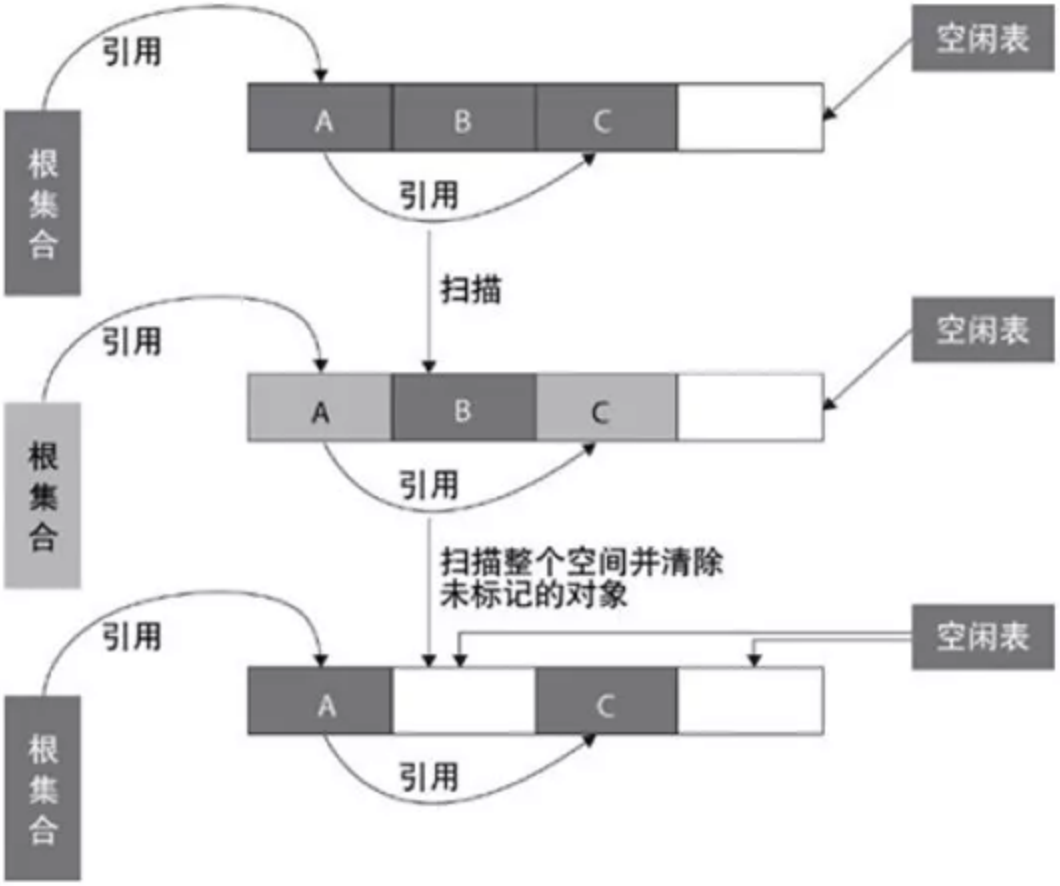
2、及时关闭资源；

3、不用的对象，可以手动设置为 null。

三、常用的垃圾收集算法

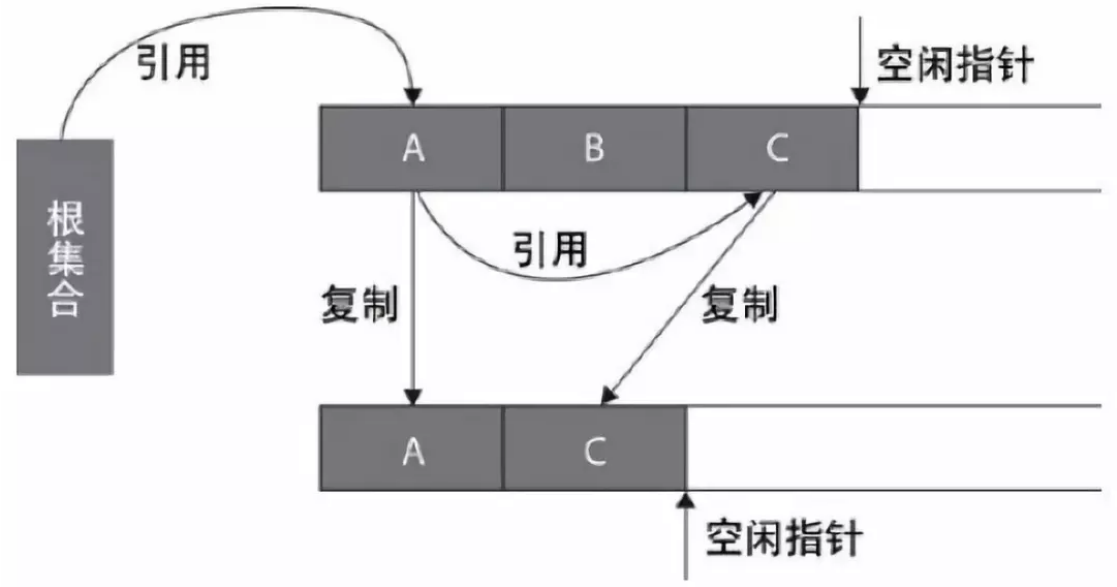
3.1 标记-清除算法（Mark-Sweep）

标记-清除算法采用从根集合（GC Roots）进行扫描，对存活的对象进行标记，标记完毕后，再扫描整个空间中未被标记的对象，进行回收，如下图所示。**标记-清除算法不需要进行对象的移动，只需对不存活的对象进行处理，在存活对象比较多的情况下极为高效，但由于标记-清除算法直接回收不存活的对象，因此会造成内存碎片。**



3.2 复制算法(Copying)

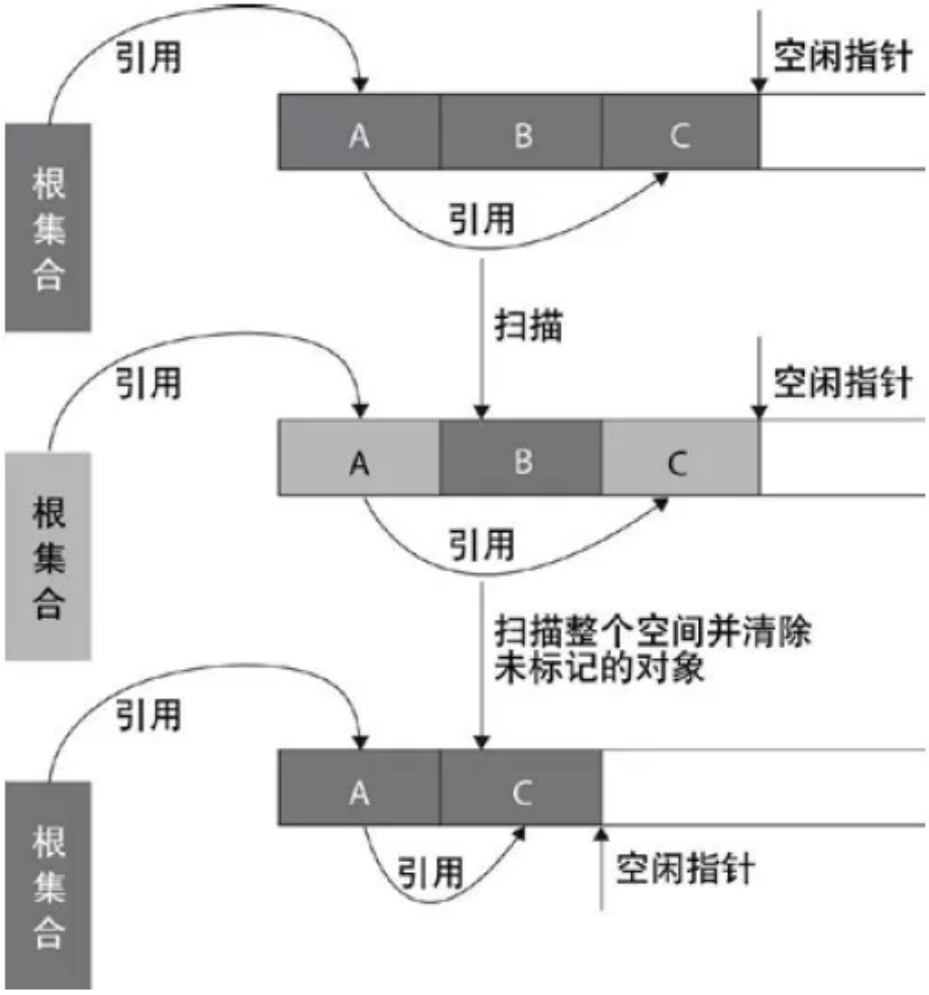
复制算法的提出是为了克服句柄的开销和解决内存碎片的问题。它开始时把堆分成 一个对象面和多个空闲面， 程序从对象面为对象分配空间，当对象满了，基于 copying 算法的垃圾 收集就从根集合（GC Roots）中扫描活动对象，并将每个活动对象复制到空闲面(使得活动对象所占的内存之间没有空闲洞)，这样空闲面变成了对象面，原来的对象面变成了空闲面，程序会在新的对象面中分配内存。



3.3 标记-整理算法(Mark-compact)

标记-整理算法采用标记-清除算法一样的方式进行对象的标记，但在清除时不同，在回收不存活的对象占用的空间后，会将所有的存活对象往左端空闲空间移动，并更新对应的指针。**标记-整理算法是在标记-清除算法的基础上，又进行了对象的移动，因此成本更高，但是却解决了内存碎片的问题。**

具体流程见下图：



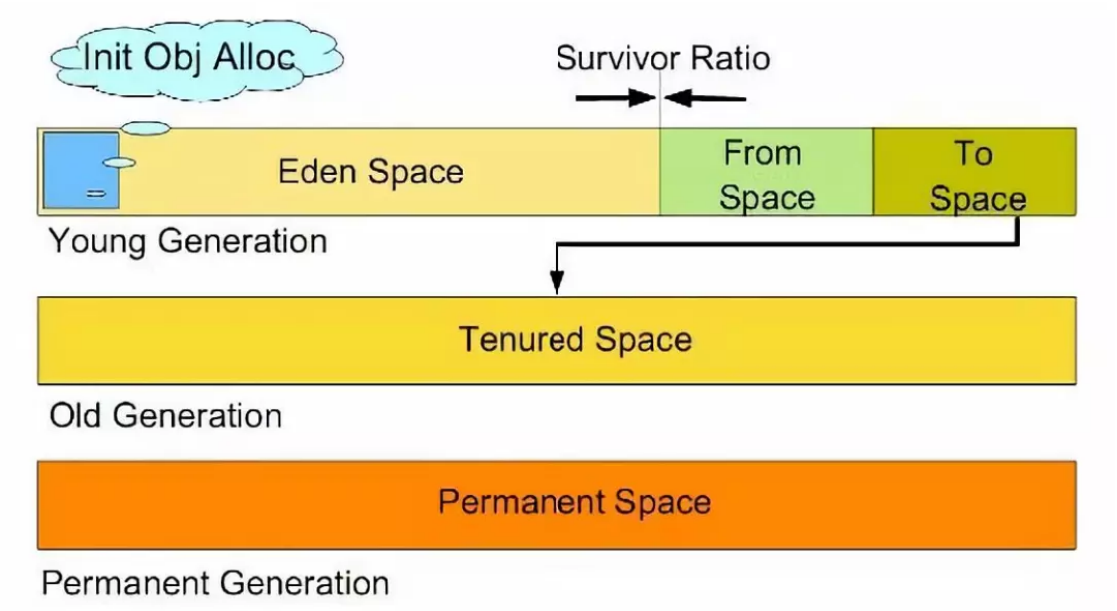
3.4 分代收集算法

> **为什么要分代?**
>
> 分代的垃圾回收策略，是基于这样一个事实：**不同的对象的生命周期是不一样的。**因此，不同生命周期的对象可以采取不同的收集方式，以便提高回收效率。
>
> 在 Java 程序运行的过程中，会产生大量的对象，其中有些对象是与业务信息相关，比如 Http 请求中的 Session 对象、线程、Socket 连接，这类对象跟业务直接挂钩，因此生命周期比较长。但是还有一些对象，主要是程序运行过程中生成的临时

> 变量，这些对象生命周期会比较短，比如：String 对象，由于其不变类的特性，系统会产生大量的这些对象，有些对象甚至只用一次即可回收。
>
> 试想，在不进行对象存活时间区分的情况下，每次垃圾回收都是对整个堆空间进行回收，花费时间相对会长，同时，因为每次回收都需要遍历所有存活对象，但实际上，对于生命周期长的对象而言，这种遍历是没有效果的，因为可能进行了很多次遍历，但是他们依旧存在。因此，分代垃圾回收采用分治的思想，进行代的划分，把不同生命周期的对象放在不同代上，不同代上采用最适合它的垃圾回收方式进行回收。

分代收集算法是目前大部分JVM的垃圾收集器采用的算法。**它的核心思想是根据对象存活的生命周期将内存划分为若干个不同的区域。**一般情况下将堆区划分为老年代（Tenured Generation）和新生代（Young Generation），在堆区之外还有一个代就是永久代（Permanet Generation）。

老年代的特点是每次垃圾收集时只有少量对象需要被回收，而新生代的特点是每次垃圾回收时都有大量的对象需要被回收，那么就可以根据不同代的特点采取最适合的收集算法。



3.4.1 年轻代（Young Generation）的回收算法 (回收主要以 Copying 为主)

1、所有新生成的对象首先都是放在年轻代的。年轻代的目标就是尽可能快速的收集掉那些生命周期短的对象。

2、新生代内存按照 8:1:1 的比例分为一个 eden 区和两个 survivor(survivor0, survivor1)区。大部分对象在Eden区中生成。回收时先将 eden 区存活对象复制到一个 survivor0 区，然后清空 eden 区，当这个 survivor0 区也存放满了时，则将 eden 区和 survivor0 区存活对象复制到另一个 survivor1 区，然后清空 eden 区 和这个 survivor0 区，此时 survivor0 区是空的，然后将survivor0 区和 survivor1 区交换，即保持 survivor1 区为空， 如此往复。

3、 当 survivor1 区不足以存放 eden区 和 survivor0区 的存活对象时，就将存活对象直接存放到老年代。若是老年代也满了就会触发一次Full GC(Major GC)，也就是新生

代、老年代都进行回收。

4、新生代发生的 GC 也叫做 Minor GC，MinorGC 发生频率比较高(不一定等 Eden 区满了才触发)。

3.4.2 年老代（Old Generation）的回收算法（回收主要以 Mark-Compact 为主)

1、在年轻代中经历了 N 次垃圾回收后仍然存活的对象，就会被放到年老代中。因此，可以认为年老代中存放的都是一些生命周期较长的对象。

2、内存比新生代也大很多(大概比例是1:2)，当老年代内存满时触发 Major GC 即 Full GC，Full GC 发生频率比较低，老年代对象存活时间比较长，存活率标记高。

3.4.3 永久代（Permanent Generation）的回收算法

用于存放静态文件，如 Java 类、方法等。永久代对垃圾回收没有显著影响，但是有些应用可能动态生成或者调用一些 class，例如 Hibernate 等，在这种时候需要设置一个比较大的永久代空间来存放这些运行过程中新增的类。永久代也称方法区，具体的回收可参见上文 2.5 节。

3.5 其他问题

3.5.1 如何处理碎片?

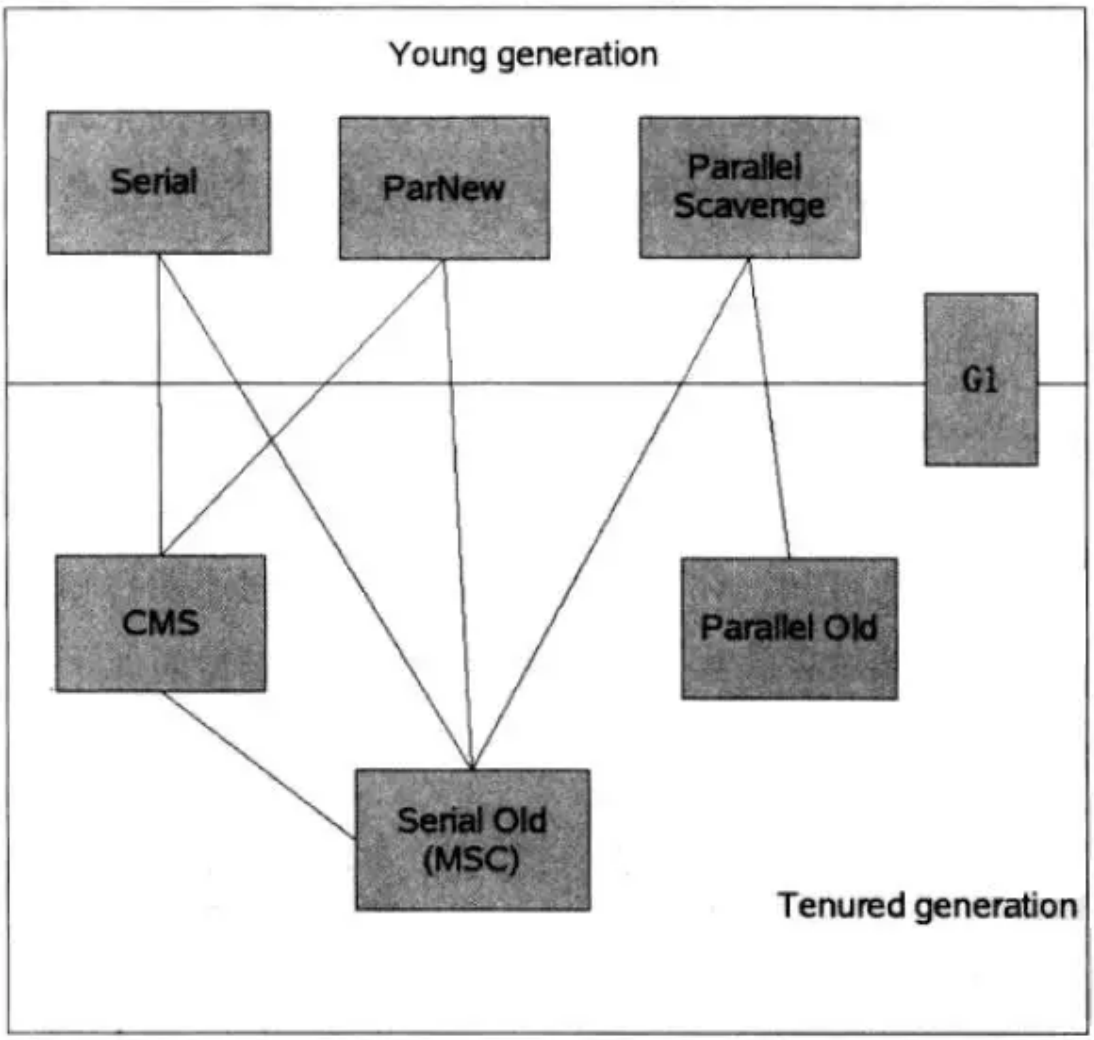
由于不同 Java 对象存活时间是不一定的，因此，在程序运行一段时间以后，如果不进行内存整理，就会出现零散的内存碎片。碎片最直接的问题就是会导致无法分配大块的内存空间，以及程序运行效率降低。所以，在上面提到的基本垃圾回收算法中，“复制”方式和“标记-整理”方式，都可以解决碎片的问题。

3.5.2 浮动垃圾

由于在应用运行的同时进行垃圾回收，所以有些垃圾可能在垃圾回收进行完成时产生，这样就造成了“Floating Garbage”，这些垃圾需要在下次垃圾回收周期时才能回收掉。所以，并发收集器一般需要20%的预留空间用于这些浮动垃圾。

四、常见的垃圾收集器

下面一张图是HotSpot虚拟机包含的所有收集器：



> **串行处理器：**
> 适用情况：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。
> 缺点：只能用于小型应用
>
> **并行处理器：**
> 适用情况：“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。举例：后台处理、科学计算。
> 缺点：垃圾收集过程中应用响应时间可能加长
>
> **并发处理器：**
> 适用情况：“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。举例：Web 服务器/应用服务器、电信交换、集成开发环境。

4.1 Serial 收集器（复制算法)

新生代单线程收集器，标记和清理都是单线程，优点是简单高效。是 client 级别默认的GC方式，可以通过-XX:+UseSerialGC来强制指定。

4.2 Serial Old 收集器(标记-整理算法)

老年代单线程收集器，Serial 收集器的老年代版本。

4.3 ParNew 收集器(停止-复制算法)

新生代收集器，可以认为是 Serial 收集器的多线程版本，在多核 CPU 环境下有着比 Serial 更好的表现。

4.4 Parallel Scavenge 收集器(停止-复制算法)

并行收集器，追求高吞吐量，高效利用 CPU。吞吐量一般为 99%， 吞吐量= 用户线程时间/(用户线程时间+GC线程时间)。适合后台应用等对交互相应要求不高的场景。是 server 级别默认采用的GC方式，可用 -XX:+UseParallelGC 来强制指定，用 -XX:ParallelGCThreads=4 来指定线程数。

4.5 Parallel Old 收集器(停止-复制算法)

Parallel Old 收集器的老年代版本，并行收集器，吞吐量优先。

4.6 CMS(Concurrent Mark Sweep)收集器（标记-清除算法）

高并发、低停顿，追求最短 GC 回收停顿时间，cpu 占用比较高，响应时间快，停顿时间短，多核 cpu 追求高响应时间的选择。

CMS 是英文 Concurrent Mark-Sweep 的简称，是**以牺牲吞吐量为代价来获得最短回收停顿时间的垃圾回收器。**对于要求服务器响应速度的应用上，这种垃圾回收器非常适合。在启动 JVM 的参数加上“-XX:+UseConcMarkSweepGC”来指定使用 CMS 垃圾回收器。

CMS 使用的是标记-清除的算法实现的，所以在 gc 的时候会产生大量的内存碎片，当剩余内存不能满足程序运行要求时，系统将会出现 Concurrent Mode Failure，临时 CMS 会采用 Serial Old 回收器进行垃圾清除，此时的性能将会被降低。

- **整个过程分为四步：**

1、**初始标记**：记录下直接与 root 相连的对象【暂停所有的其他线程，速度很快】；

2、**并发标记**： 同时开启 GC 和用户线程，用一个闭包结构去记录可达对象。但在这个阶段结束，这个闭包结构并不能保证包含当前所有的可达对象。因为用户线程可能会不断的更新引用域，所以 GC 线程无法保证可达性分析的实时性。所以这个算法里会跟踪记录这些发生引用更新的地方。

3、**重新标记**： 重新标记阶段就是为了修正并发标记期间因为用户程序继续运行而导致标记产生变动的那一部分对象的标记记录。【这个阶段的停顿时间一般会比初始标记阶段的时间稍长，远远比并发标记阶段时间短】；

4、并发清除： 开启用户线程，同时 GC 线程开始对为标记的区域做清扫。

- **优缺点**

主要优点：并发收集、低停顿；

主要缺点：对 CPU 资源敏感、无法处理浮动垃圾、它使用的回收算法“标记-清除”算法会导致收集结束时会有大量空间碎片产生。

4.7 G1 收集器【重点】

G1 收集器在后台维护了一个优先列表，每次根据允许的收集时间，优先选择回收价值最大的 Region(这也就是它的名字 Garbage-First的由来。

- **垃圾回收的瓶颈**

传统分代垃圾回收方式，已经在一定程度上把垃圾回收给应用带来的负担降到了最小，把应用的吞吐量推到了一个极限。但是他无法解决的一个问题，就是 Full GC 所带来的应用暂停。在一些对实时性要求很高的应用场景下，GC 暂停所带来的请求堆积和请求失败是无法接受的。这类应用可能要求请求的返回时间在几百甚至几十毫秒以内，如果分代垃圾回收方式要达到这个指标，只能把最大堆的设置限制在一个相对较小范围内，但是这样有限制了应用本身的处理能力，同样也是不可接受的。

分代垃圾回收方式确实也考虑了实时性要求而提供了并发回收器，支持最大暂停时间的设置，但是受限于分代垃圾回收的内存划分模型，其效果也不是很理想。

为了达到实时性的要求（其实 Java 语言最初的设计也是在嵌入式系统上的），一种新垃圾回收方式呼之欲出，它既支持短的暂停时间，又支持大的内存空间分配。可以很好的解决传统分代方式带来的问题。

- **增量收集的演进**

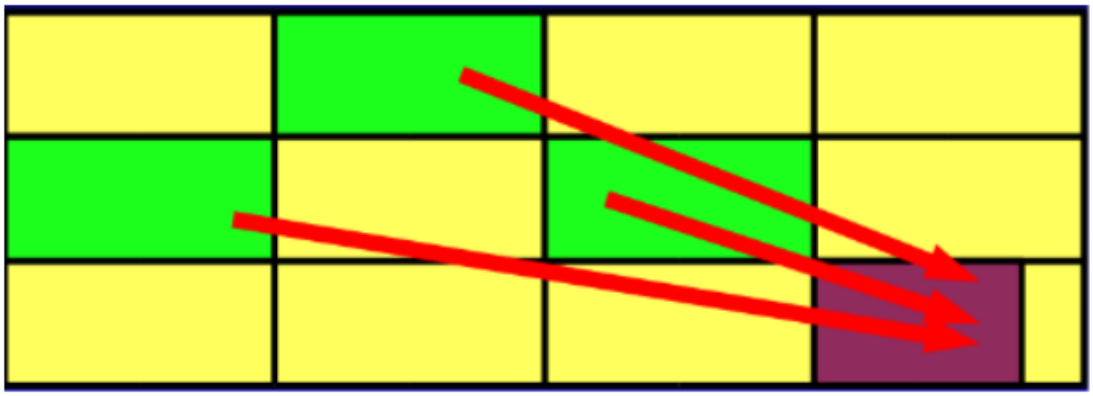
增量收集的方式在理论上可以解决传统分代方式带来的问题。增量收集把对堆空间划分成一系列内存块，使用时，先使用其中一部分（不会全部用完），垃圾收集时把之前用掉的部分中的存活对象再放到后面没有用的空间中，这样可以实现一直边使用边收集的效果，避免了传统分代方式整个使用完了再暂停的回收的情况。

当然，传统分代收集方式也提供了并发收集，但是他有一个很致命的地方，就是把整个堆做为一个内存块，这样一方面会造成碎片（无法压缩），另一方面他的每次收集都是对整个堆的收集，无法进行选择，在暂停时间的控制上还是很弱。而增量方式，通过内存空间的分块，恰恰可以解决上面问题。

- **G1 的设计目标**

> 1、支持很大的堆；
>
> 2、高吞吐量
> --支持多CPU和垃圾回收线程
> --在主线程暂停的情况下，使用并行收集
> --在主线程运行的情况下，使用并发收集
>
> 3、实时目标：可配置在 N 毫秒内最多只占用 M 毫秒的时间进行垃圾回收

当然G1要达到实时性的要求，相对传统的分代回收算法，在性能上会有一些损失。



G1 可谓博采众家之长，力求到达一种完美。它吸取了增量收集优点，**把整个堆划分为一个一个等大小的区域（region）。内存的回收和划分都以region为单位**；同时，它也吸取了 CMS 的特点，把这个垃圾回收过程分为几个阶段，分散一个垃圾回收过程；而且，G1也认同分代垃圾回收的思想，认为不同对象的生命周期不同，可以采取不同收集方式，因此，它也支持分代的垃圾回收。**为了达到对回收时间的可预计性，G1 在扫描了 region 以后，对其中的活跃对象的大小进行排序，首先会收集那些活跃对象小的 region，以便快速回收空间（要复制的活跃对象少了），因为活跃对象小，里面可以认为多数都是垃圾，所以这种方式被称为 Garbage First（G1）的垃圾回收算法，即：垃圾优先的回收。**

- **回收步骤：**

1、初始标记（Initial Marking）

G1 对于每个 region 都保存了两个标识用的 bitmap，一个为：previous marking bitmap，一个为：next marking bitmap，bitmap 中包含了一个 bit 的地址信息来指向对象的起始点。开始 Initial Marking 之前，首先并发的清空 next marking bitmap，然后停止所有应用线程，并扫描标识出每个 region 中 root 可直接访问到的对象，将 region 中 top 的值放入 next top at mark start（TAMS）中，之后恢复所有应用线程。触发这个步骤执行的条件为：

> G1 定义了一个 JVM Heap 大小的百分比的阀值，称为 h，另外还有一个 H，H 的值为(1-h) * Heap Size，目前这个 h 的值是固定的，后续 G1 也许会将其改为动态的，根据 jvm 的运行情况来动态的调整，在分代方式下，G1 还定义了一个u 以及 soft limit，soft limit 的值为 H-u * Heap Size，当 Heap 中使用的内存超过了 soft limit 值时，就会在一次 clean up 执行完毕后在应用允许的 GC 暂停时间范围内尽快的执行此步骤； 在 pure 方式下，G1 将 marking 与 clean up 组成一个环，以便 clean up 能充分的使用 marking 的信息，当 clean up 开始回收时，首先回收能够带来最多内存空间的 regions，当经过多次的 clean up，回收到没多少空间的 regions 时，G1 重新初始化一个新的 marking 与 clean up 构成的环。

2、并发标记（Concurrent Marking）

按照之前 Initial Marking 扫描到的对象进行遍历，以识别这些对象的下层对象的活跃状态，对于在此期间应用线程并发修改的对象的以来关系则记录到 remembered set logs 中，新创建的对象则放入比 top 值更高的地址区间中，这些新创建的对象默认状态即为活跃的，同时修改 top 值。

3、最终标记暂停（Final Marking Pause）

当应用线程的 remembered set logs 未满时，是不会放入 filled RS buffers 中的，在这样的情况下，这些 remebered set logs 中记录的 card 的修改就会被更新了，因此需要这一步，这一步要做的就是把应用线程中存在的 remembered set logs 的内容进行处理，并相应的修改 remembered sets，这一步需要暂停应用，并行的运行。

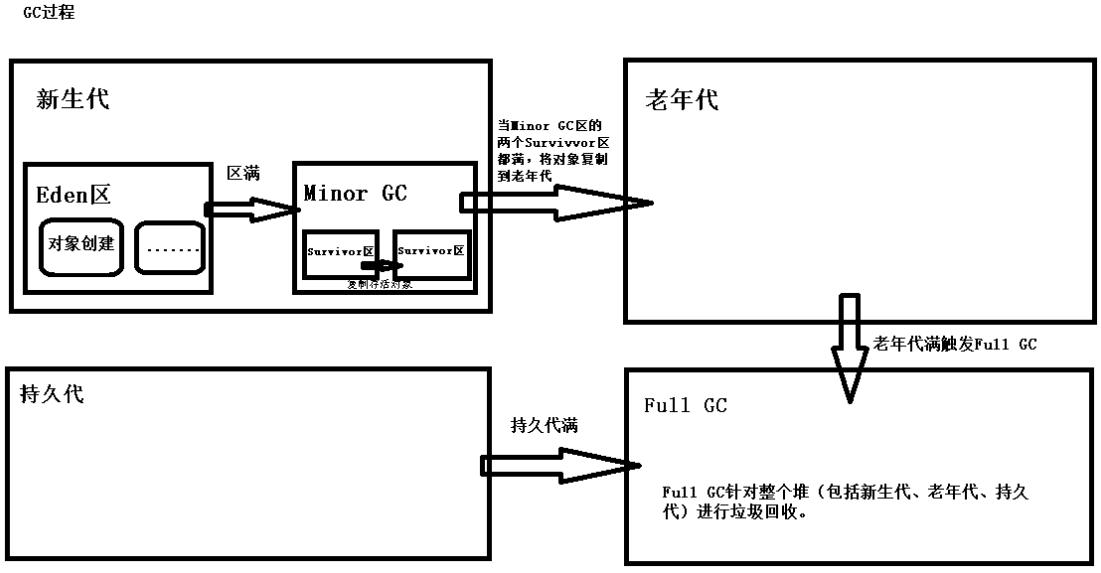
4、存活对象计算及清除（Live Data Counting and Cleanup）

值得注意的是，在G1中，并不是说Final Marking Pause执行完了，就肯定执行Cleanup这步的，由于这步需要暂停应用，G1为了能够达到准实时的要求，需要根据用户指定的最大的GC造成的暂停时间来合理的规划什么时候执行Cleanup，另外还有几种情况也是会触发这个步骤的执行的：

> G1 采用的是复制方法来进行收集，必须保证每次的 ” to space” 的空间都是够的，因此 G1 采取的策略是当已经使用的内存空间达到了 H 时，就执行 Cleanup 这个步骤； 对于 full-young 和 partially-young 的分代模式的 G1 而言，则还有情况会触发 Cleanup 的执行，full-young 模式下，G1 根据应用可接受的暂停时间、回收 young regions 需要消耗的时间来估算出一个 yound regions 的数量值，当 JVM 中分配对象的 young regions 的数量达到此值时，Cleanup 就会执行；partially-young 模式下，则会尽量频繁的在应用可接受的暂停时间范围内执行 Cleanup，并最大限度的去执行 non-young regions 的 Cleanup。

五、内存分配与回收策略

由于对象进行了分代处理，因此垃圾回收区域、时间也不一样。GC有两种类型：Scavenge GC 和 Full GC。



1，新生代：（1）所有对象创建在新生代的Eden区，当Eden区满后触发新生代的Minor GC，将Eden区和非空闲Survivor区存活的对象复制到另外一个空闲的Survivor区中。（2）保证一个Survivor区是空的，新生代Minor GC就是在两个Survivor区之间相互复制存活对象，直到Survivor区满为止。

2，老年代：当Survivor区也满了之后就通过Minor GC将对象复制到老年代。老年代也满了的话，就将触发Full GC，针对整个堆（包括新生代、老年代、持久代）进行垃圾回收。

3，持久代：持久代如果满了，将触发Full GC。

5.1 GC 的触发时机【重点】

1. Minor / Scavenge GC

Minor：[ˈmaɪnər]]

所有对象创建在新生代的 Eden 区，当 Eden 区满后触发新生代的 Minor GC，将 Eden 区和非空闲 Survivor 区存活的对象复制到另外一个空闲的 Survivor 区中。保证一个 Survivor 区是空的，新生代 Minor GC 就是在两个 Survivor 区之间相互复制存活对象，直到 Survivor 区满为止。

Minor/Scavenge这种方式的 GC 是在年轻代的 Eden 区进行，不会影响到年老代。因为大部分对象都是从 Eden 区开始的，同时 Eden 区不会分配的很大，所以 Eden 区的 GC 会频繁进行。因而，一般在这里需要使用速度快、效率高的算法，使 Eden 去能尽快空闲出来。

2. Full GC

对整个堆进行整理，包括 Young、Tenured 和 Perm。Full GC 因为需要对整个堆进行回收，所以比 Minor GC 要慢，因此应该尽可能减少 Full GC 的次数。在对 JVM 调优的过程中，很大一部分工作就是对于 Full GC 的调节。

Minor 有如下原因可能导致 Full GC：

1、**调用 System.gc()**，会建议虚拟机执行 Full GC。只是建议虚拟机执行 Full GC，但是虚拟机不一定真正去执行。

2、**老年代空间不足**，原因：老年代空间不足的常见场景为大对象直接进入老年代、长期存活的对象进入老年代等。为了避免以上原因引起的 Full GC，应当尽量不要创建过大的对象以及数组。除此之外，可以通过 -Xmn 虚拟机参数调大新生代的大小，让对象尽量在新生代被回收掉，不进入老年代。还可以通过 -XX:MaxTenuringThreshold 调大对象进入老年代的年龄，让对象在新生代多存活一段时间；

3、**空间分配担保失败**：使用复制算法的 Minor GC 需要老年代的内存空间作担保，如果担保失败会执行一次 Full GC；

4、**JDK 1.7 及以前的永久代空间不足。**在 JDK 1.7 及以前，HotSpot 虚拟机中的方法区是用永久代实现的，永久代中存放的为一些 Class 的信息、常量、静态变量等数据。**当系统中要加载的类、反射的类和调用的方法较多时，永久代可能会被占满**，在未配置为采用 CMS GC 的情况下也会执行 Full GC。如果经过 Full GC 仍然回收不了，那么虚拟机会抛出 java.lang.OutOfMemoryError。为避免以上原因引起的 Full GC，可采用的方法为增大永久代空间或转为使用 CMS GC。

5、Concurrent Mode Failure 执行 CMS GC 的过程中，同时有对象要放入老年代，而此时老年代空间不足（可能是 GC 过程中浮动垃圾过多导致暂时性的空间不足），便会报 Concurrent Mode Failure 错误，并触发 Full GC。

5.2 内存分配

1、**对象优先在 Eden 区分配**：大多数情况下，对象在新生代 Eden 区分配，当 Eden 区空间不够时，发起 Minor GC。

2、**大对象直接进入老年代**：大对象是指需要连续内存空间的对象，最典型的大对象是那种很长的字符串以及数组。经常出现大对象会提前触发垃圾收集以获取足够的连续空间分配给大对象。**-XX:PretenureSizeThreshold，大于此值的对象直接在老年代分配，避免在 Eden 区和 Survivor 区之间的大量内存复制。**

3、**长期存活的对象将进入老年代**：为对象定义年龄计数器，对象在 Eden 出生并经过 Minor GC 依然存活，将移动到 Survivor 中，年龄就增加 1 岁，增加到一定年龄则移动到老年代中。-XX:MaxTenuringThreshold 用来定义年龄的阈值。

4、**动态对象年龄判定**：为了更好的适应不同程序的内存情况，虚拟机不是永远要求对象年龄必须达到了某个值才能进入老年代，如果 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，年龄大于或等于该年龄的对象就可以直接进入老年代，无需达到要求的年龄。

5、**空间分配担保**

（1）在发生 Minor GC 之前，虚拟机先检查老年代最大可用的连续空间是否大于新生代所有对象总空间，如果条件成立的话，那么 Minor GC 可以确认是安全的；

（2）如果不成立的话虚拟机会查看 HandlePromotionFailure 设置值是否允许担保失败，如果允许那么就会继续检查老年代最大可用的连续空间是否大于历次晋升到老年代对象的平均大小，如果大于，将尝试着进行一次 Minor GC；如果小于，或者 HandlePromotionFailure 设置不允许冒险，那么就要进行一次 Full GC。

六、JVM 监控工具

- **JDK 的可视化工具**

1、jvisualvm：虚拟机监视和故障处理平台

VisualVM（All- in- One Java Troubleshooting Tool） 是到目前为止，随 JDK发布的功能最强大的运行 监视和故障处理程序， 并且可以预见在未来一段时间内都是官方主力发展的虚拟机故障处理工具。官方在 VisualVM 的软件说明中写上了“ All- in- One” 的描述字样， 预示着它除了运行监视、故障处理外，还提供了很多其他方面的功能。
VisualVM 基于 NetBeans 平台开发， 因此它一开始就具备了插件扩展功能的特性， 通过插件扩展 支持，VisualVM 可以做到：

1、显示虚拟机进程及进程的配置和环境信息（ jps、 jinfo）；

2、监视应用程序的CPU、GC、堆、方法区及线程的信息（ jstat、 jstack）；

3、dump 及分析堆转储快照（jmap、 jhat）；

4、方法级的程序运行性能分析，找出被调用最多、运行时间最长的方法；

5、离线程序快照： 收集程序的运行时配置、线程 dump、 内存 dump 等信息建立 一个快照， 可以将快照 发送开发者处进行 Bug 反馈；

6、其他 plugins 的无限的可能性。

2、JConsole

JConsole 工具在 JDK/bin 目录下，启动 JConsole 后，将自动搜索本机运行的 jvm 进程，不需要 jps 命令来查询指定。双击其中一个 jvm 进程即可开始监控，也可使用“远程进程”来连接远程服务器。

用于对 JVM 中的内存、线程和类等进行监控；

3、JProfiler

商业软件，需要付费。功能强大。

4、如何利用监控工具调优【结合具体的工具看下】

- **观察内存释放情况、集合类检查、对象树**

4.1、堆信息查看

1、可查看堆空间大小分配（年轻代、年老代、持久代分配）

2、提供即时的垃圾回收功能、

3、垃圾监控（长时间监控回收情况）

4、查看堆内类、对象信息查看：数量、类型等

5、对象引用情况查看

- **有了堆信息查看方面的功能，我们一般可以顺利解决以下问题：**

1、年老代年轻代大小划分是否合理；

2、内存泄漏垃；

3、圾回收算法设置是否合理。

4.2、线程监控

1、线程信息监控：系统线程数量

2、线程状态监控：各个线程都处在什么样的状态下

3、Dump 线程详细信息：查看线程内部运行情况

4、死锁检查

4.3、热点分析

CPU 热点：检查系统哪些方法占用的大量 CPU 时间；

内存热点：检查哪些对象在系统中数量最大（一定时间内存活对象和销毁对象一起统计）这两个东西对于系统优化很有帮助。我们可以根据找到的热点，有针对性的进行系统的瓶颈查找和进行系统优化，而不是漫无目的的进行所有代码的优化。

4.4、快照

快照是系统运行到某一时刻的一个定格。在我们进行调优的时候，不可能用眼睛去跟踪所有系统变化，依赖快照功能，我们就可以进行系统两个不同运行时刻，对象（或类、线程等）的不同，以便快速找到问题。

举例说，我要检查系统进行垃圾回收以后，是否还有该收回的对象被遗漏下来的了。那么，我可以在进行垃圾回收前后，分别进行一次堆情况的快照，然后对比两次快照的对象情况。

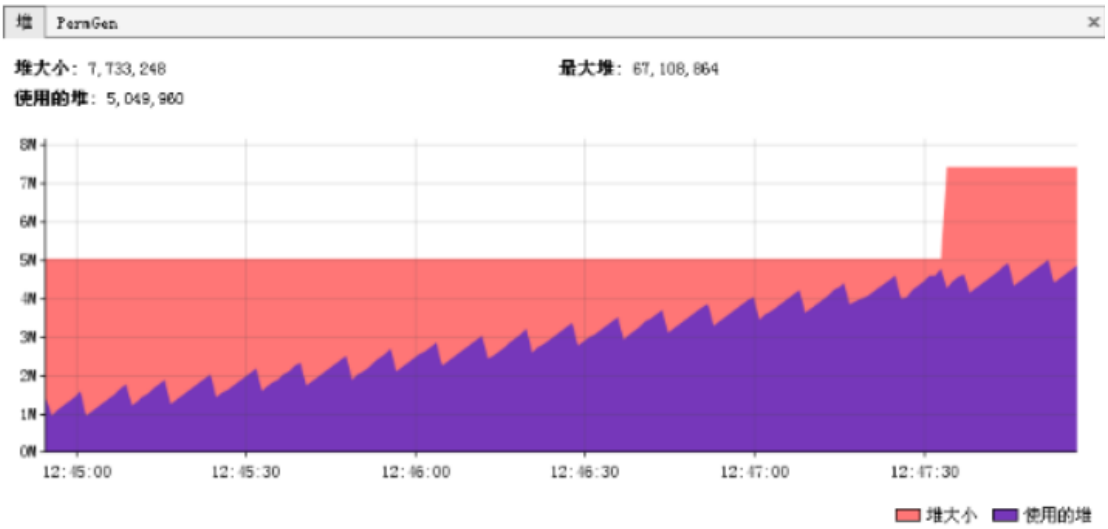
4.5 内存泄露检查

内存泄漏是比较常见的问题，而且解决方法也比较通用，这里可以重点说一下，而线程、热点方面的问题则是具体问题具体分析了。

内存泄漏一般可以理解为系统资源（各方面的资源，堆、栈、线程等）在错误使用的情况下，导致使用完毕的资源无法回收（或没有回收），从而导致新的资源分配请求无法完成，引起系统错误。内存泄漏对系统危害比较大，因为它可以直接导致系统的崩溃。

需要区别一下，内存泄漏和系统超负荷两者是有区别的，虽然可能导致的最终结果是一样的。内存泄漏是用完的资源没有回收引起错误，而系统超负荷则是系统确实没有那么多资源可以分配了（其他的资源都在使用）。

- **1、年老代堆空间被占满：异常： java.lang.OutOfMemoryError: Java heap space**



这是最典型的内存泄漏方式，简单说就是**所有堆空间都被无法回收的垃圾对象占满，虚拟机无法再在分配新空间。**

如上图所示，这是非常典型的内存泄漏的垃圾回收情况图。所有峰值部分都是一次垃圾回收点，所有谷底部分表示是一次垃圾回收后剩余的内存。连接所有谷底的点，可以发现一条由底到高的线，这说明，随时间的推移，系统的堆空间被不断占满，最终会占满整个堆空间。因此可以初步认为系统内部可能有内存泄漏。（上面的图仅供示例，在实际情况下收集数据的时间需要更长，比如几个小时或者几天）。

- **解决：**

这种方式解决起来也比较容易，一般就是根据垃圾回收前后情况对比，同时根据对象引用情况（常见的集合对象引用）分析，基本都可以找到泄漏点。

2、持久代被占满：异常：java.lang.OutOfMemoryError: PermGen space

Perm 空间被占满。无法为新的 class 分配存储空间而引发的异常。这个异常以前是没有的，但是在 Java 反射大量使用的今天这个异常比较常见了。**主要原因就是大量动态反射生成的类不断被加载，最终导致 Perm 区被占满。**更可怕的是，不同的 classLoader 即便使用了相同的类，但是都会对其进行加载，相当于同一个东西，如果有 N 个 classLoader 那么他将会被加载 N 次。因此，某些情况下，这个问题基本视为无解。当然，存在大量 classLoader 和大量反射类的情况其实也不多。

- **解决：**

> 1、-XX:MaxPermSize=16m
>
> 2、换用 JDK。比如 JRocket。

3、堆栈溢出：异常：java.lang.StackOverflowError

这个就不多说了，一般就是递归没返回，或者循环调用造成

4、线程堆栈满异常：Fatal：Stack size too small

java 中一个线程的空间大小是有限制的。JDK5.0 以后这个值是1M。与这个线程相关的数据将会保存在其中。但是当线程空间满了以后，将会出现上面异常。

- **解决：**

增加线程栈大小。-Xss:2M。但这个配置无法解决根本问题，还要看代码部分是否有造成泄漏的部分。

5、系统内存被占满异常：java.lang.OutOfMemoryError: unable to create new native thread

这个异常是由于操作系统没有足够的资源来产生这个线程造成的。系统创建线程时，除了要在 Java 堆中分配内存外，操作系统本身也需要分配资源来创建线程。因此，当线程数量大到一定程度以后，堆中或许还有空间，但是操作系统分配不出资源来了，就出现这个异常了。

分配给 Java 虚拟机的内存愈多，系统剩余的资源就越少，因此，当系统内存固定时，分配给 Java 虚拟机的内存越多，那么，系统总共能够产生的线程也就越少，两者成反比的关系。同时，可以通过修改 -Xss 来减少分配给单个线程的空间，也可以增加系统总共内生产的线程数。

- **解决：**

重新设计系统减少线程数量。线程数量不能减少的情况下，通过 -Xss 减小单个线程大小，以便能生产更多的线程。

- **JDK 的命令行工具**

1、jps： 查看当前 Java 进程

jps（Java Virtual Machine Process Status Tool） 是 JDK 提供的一个查看当前 Java 进程的小工具。

> **命令格式：jps[options] [hostid]**
>
> **options 选项：**
>
> -q：仅输出 VM 标识符，不包括 classname、jar name、arguments in main method
>
> **-m：输出 main method 的参数**
>
> **-l：输出完全的包名，应用主类名，jar 的完全路径名**
>
> **-v：输出虚拟机进程启动时的 JVM 参数**
>
> -V：输出通过 flag 文件传递到 JVM 中的参数(.hotspotrc 文件或 -XX:Flags= 所指定的文件
>
> -Joption：传递参数到 vm,例如：-J-Xms512m

> **hostid：**
>
> [protocol:][[//]hostname][:port][/servername]

2、jstat：显示虚拟机运行数据

https://blog.csdn.net/zhaozheng7758/article/details/8623549

jstat（JVM Statistics Monitoring Tool）用于收集 HotSpot 虚拟机各方面的运行数据。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。

> **jstat 命令格式为：jstat [option vmid[internal [s | ms] [count]]]**
>
> option：选项（我们一般使用：-gcutil：查看 gc 情况）
>
> vmid：-VM 的进程号，即当前的 java 进程号
>
> internal：间隔时间，单位为秒或者毫秒
>
> count：打印次数，如果缺省，则打印无数次
>
> 举例：jstat –gc 20445 1 20 ： 每 1 毫秒查询一次进程 20445 的垃圾回收情况，监控 20 次

> **options 选项：**
>
> -class：监视类装载、卸载数量、总空间及类装载在所耗费的时间
>
> -gc：监视 Java 堆状况，Eden 区、2 个 survivor 区、老年代、永久代等的容最、已用空间、GC 时间合计等信息
>
> -gccapacity：监视内容与 - gc 基本相同，但输出主要关注 Java 堆各个区域使用到的最大和最小空间
>
> -gcutil：监视内容与 - gc 慕本相同，但输出主要关注已使用空间占总空间的百分比
>
> -gccause：与 - gcutil 功能一样，但是会额外输出导致上一次 GC 产生的原因
>
> -gcnew：监视新生代 GC 的状况
>
> -gcnewcapacity：监视内容与 - gcnew 基本相同，输出主要关注使用到的最大和最小空间
>
> -gcold：视老年代 GC 的状况
>
> -gcoldcapacity：监视内容与 - gcold 基本相同，输出主要关注使用到的最大和最小空间
>
> -gcpermcapacity：输出永久代使用到的最大和最小空间
>
> -compiler：输出 JIT 编译器编译过的方法、娓时等信息
>
> -printcompilation：输出已经被 JIT 编译的方法

3、jmap：内存监控

jmp（Memory Map for Java）生成虚拟机的内存转储快照（heapdump 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。

> **jmap 命令格式：jmap [option] vmid**
>
> 示例：jmap -dump:format=b,file=yhj.dump 20445
>
> **option 选项：**
>
> **-dump：生成 Java 堆转储快照。**格式为： - dump: [live]format=b,file=<filename>，其中 live 子参数说明是否只 dump 出存活的对象
>
> -finalizerinfo：显示在 F-Queue 中等待 Finalizer 线程执行 finalize 方法的对象。只在 Linux /Solaris 平台下有效
>
> -heap：显示 Java 堆详细信息，如使用哪种回收器、参数配置、分代状况等． 只在 Linux/Solaris 平台下有效
>
> -histo：显示堆中对象统计信息，包括类、实例数量和合计容量
>
> -permstat：以 ClassLoader 为统计口径显示永久代内存状态。只在 Linux/Solaris 平台下有效
>
> -F：当虚拟机进程对 - dump 选项没有响应时，可使用这个选项强制生成 dump 快照。只 Linux/Solaris 平台下有效

4、jhat：分析 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， 让用户可以在浏览器上查看分析结果。常与 jmap 搭配使用。

5、jstack：线程快照

jstack（Stack Trace for Java）显示虚拟机的线程快照。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。**线程出现停顿的时候通过 jstack 来查看各个线程的调用堆栈，可以知道没有响应的线程在后台正什么事情，或者等待着什么资源。**

6、jinfo：虚拟机配置信息

jinfo（Configuration Info for Java）显示虚拟机的配置信息。实时地查看和调整虚拟机的各项参数。不同操作系统支持程序不一样。

命令格式：jinfo [option] pid

7、相关参数说明

- **堆设置**

-Xms：初始堆大小

-Xmx：最大堆大小

-XX:NewSize=n：设置年轻代大小

-XX:NewRatio=n：设置年轻代和年老代的比值。如:为3，表示年轻代与年老代比值为 1：3，年轻代占整个年轻代年老代和的 1/4

-XX:SurvivorRatio=n：年轻代中 Eden 区与两个 Survivor 区的比值。注意 Survivor 区有两个。如：3，表示Eden：Survivor=3：2，一个Survivor区占整个年轻代的 1/5

-XX:MaxPermSize=n：设置持久代大小

- **收集器设置**

-XX:+UseSerialGC：设置串行收集器

-XX:+UseParallelGC：设置并行收集器

-XX:+UseParalledlOldGC：设置并行年老代收集器

-XX:+UseConcMarkSweepGC：设置并发收集器

- **垃圾回收统计信息**

-XX:+PrintGC：开启打印 gc 信息

-XX:+PrintGCDetails：打印 gc 详细信息

-XX:+PrintGCTimeStamps

-Xloggc:filename

- **并行收集器设置**

-XX:ParallelGCThreads=n：设置并行收集器收集时使用的CPU数。并行收集线程数

-XX:MaxGCPauseMillis=n：设置并行收集最大暂停时间

-XX:GCTimeRatio=n：设置垃圾回收时间占程序运行时间的百分比。公式为 1/(1+n)

- **并发收集器设置**

-XX:+CMSIncrementalMode：设置为增量模式。适用于单 CPU 情况

-XX:ParallelGCThreads=n：设置并发收集器年轻代收集方式为并行收集时，使用的 CPU 数。并行收集线程数

七、JVM 调优

1、年轻代大小选择

响应时间优先的应用：尽可能设大，直到接近系统的最低响应时间限制（根据实际情况选择）。在此种情况下，年轻代收集发生的频率也是最小的。同时，减少到达年老代的对象。

吞吐量优先的应用：尽可能的设置大，可能到达 Gbit 的程度。因为对响应时间没有要求，垃圾收集可以并行进行，一般适合8CPU以上的应用。

2、老年代大小选择

响应时间优先的应用：年老代使用并发收集器，所以其大小需要小心设置，一般要考虑并发会话率和会话持续时间等一些参数。如果堆设置小了，可以会造成内存碎片、高回收频率以及应用暂停而使用传统的标记清除方式；如果堆大了，则需要较长的收集时间。最优化的方案，一般需要参考以下数据获得：

1、并发垃圾收集信息

2、持久代并发收集次数

3、传统 GC 信息

4、花在年轻代和年老代回收上的时间比例。减少年轻代和年老代花费的时间，一般会提高应用的效率。

3、吞吐量优先的应用

一般吞吐量优先的应用都有一个很大的年轻代和一个较小的年老代。原因是，这样可以尽可能回收掉大部分短期对象，减少中期的对象，而年老代尽存放长期存活对象。

4、较小堆引起的碎片问题

因为年老代的并发收集器使用标记、清除算法，所以不会对堆进行压缩。当收集器回收时，他会把相邻的空间进行合并，这样可以分配给较大的对象。但是，当堆空间较小

时，运行一段时间以后，就会出现“碎片”，如果并发收集器找不到足够的空间，那么并发收集器将会停止，然后使用传统的标记、清除方式进行回收。如果出现“碎片”，可能需要进行如下配置：

1. -XX:+UseCMSCompactAtFullCollection：使用并发收集器时，开启对年老代的压缩。

2. -XX:CMSFullGCsBeforeCompaction=0：上面配置开启的情况下，这里设置多少次Full GC后，对年老代进行压缩

八、类文件结构

字节码 Class 类文件是由一些列字节码命令组成，用于表示程序中各种常量、变量、关键字和运算符号的语义等等。

Java 的 Class 类文件是一组由 8 位字节为基础单位的二进制流，各个数据项目严格按照顺序紧凑地排列在 Class 文件之中，中间没有添加任何分隔符，这使得 Class 文件中存储的内容几乎全部都是程序运行的必要数据，没有空隙存在。当遇到需要占用 8 位字节以上空间的数据项时，则会按照高位在前的方式分割若干个 8 位字节进行存储。

Java 虚拟机规定，Class 文件格式采用一种类似于 C 语言结构体系的伪结构来存储数据，这种伪结构中只有两种数据类型：**无符号数**和**表。**

1、无符号数：属于基本的数据类型，以 u1、u2、u4、u8 来分别代表 1 个字节、2 个字节、4 个字节和 8 个字节的无符号数，无符号数可以用来描述数字、索引引用、数量值或者按照 UTF-8 编码构成字符串值；

2、表：是由多个无符号数或者其他表作为数据项构成的复合数据类型，所有表都习惯地以"_info" 结尾。表用于描述有层次关系的复合结构的数据，整个 Class 文件本质上就是一张表。

- **Class 文件结构如下标所示：**

类型	名称	数量
u4	magic	1
u2	minor_version	1
u2	major_version	1
u2	constant_pool_count	1
cp_info	constant_pool	constant_pool_count - 1
u2	access_flags	1
u2	this_class	1
u2	super_class	1
u2	interfaces_count	1
u2	interfaces	interfaces_count
u2	fields_count	1
field_info	fields	fields_count
u2	methods_count	1
method_info	methods	methods_count
u2	attributes_count	1
attribute_info	attributes	attributes_count

Class 文件没有任何分隔符，严格按照上面结构表中的顺序排列。无论是顺序还是数量，甚至于数据存储的字节序这样的细节，都是被严格限定的，哪个字节代表什么含义，长度是多少，先后顺序如何，都不允许改变。

1、**魔数（magic）**：每个 Class 文件的头 4 个字节称为魔数（Magic Number），它的唯一作用是**确定这个文件是否为一个能被虚拟机接受的Class 文件，即判断这个文件是否符合 Class 文件规范。**

2、**文件的版本：minor_version 和 major_version**

3、**常量池：constant_pool_count 和 constant_pool**：常量池中主要存放两大类常量：字面量（Literal）和符号引用（Symbolic References）。

4、**访问标志：access_flags**：用于识别一些类或者接口层次的访问信息。包括：这个 Class 是类还是接口、是否定义了 Public 类型、是否定义为 abstract 类型、如果是类，是否被声明为了 final 等等。

5、**类索引、父类索引与接口索引集合：this_class、super_class和interfaces**

6、字段表集合：field_info、fields_count：字段表（field_info）用于描述接口或者类中声明的变量；fields_count 字段数目：表示Class文件的类和实例变量总数。

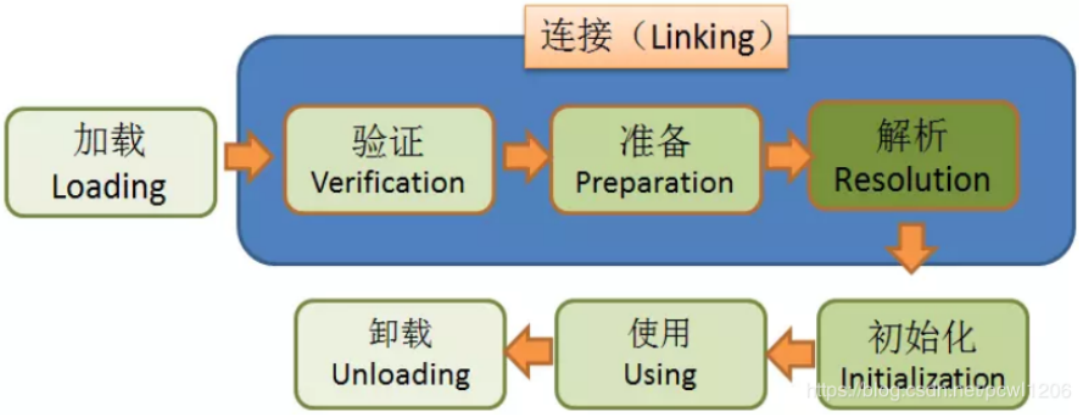
7、方法表集合：methods、methods_count

8、属性表集合：attributes、attributes_count

九、虚拟机类加载机制

虚拟机把描述类的数据从 Class 文件加载到内存，并对数据进行校验、转换解析和初始化，最终形成可以被虚拟机直接使用的 Java 类型，这就是虚拟机的类加载机制。

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、验证、准备、解析、初始化、使用、卸载** 7 个阶段。其中验证、准备、解析 3 个部分统称为连接，这7个阶段发生的顺序如下图所示：



1、类加载各阶段作用

- **1、加载**

在加载阶段，虚拟机需要完成以下三件事情：

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接口。

- **2、验证**

主要是为了确保 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。验证阶段大致上分为 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。

（1）文件格式校验【基于二进制字节流】：验证字节流是否符合 class 文件的规范，并且能被当前版本的虚拟机处理。只有通过这个阶段的验证后，字节流才会进入内存的方法区进行存储，所以后面的3个阶段的全部是基于方法区的存储结构进行的，不会再直接操作字节流；

（2）元数据验证：对字节码描述的信息进行语义分析，以保证其描述的信息符合 Java 语言规范的要求。目的是保证不存在不符合 Java 语言规范的元数据信息；

（3）字节码验证：该阶段主要工作是进行数据流和控制流分析，保证被校验类的方法在运行时不会做出危害虚拟机安全的行为；

（4）符号引用验证：最后一个阶段的校验发生在虚拟机将符号引用转化为直接引用的时候，这个转化动作将在连接的第三个阶段——解析阶段中发生。符号引用验证的目的是确保解析动作能正常执行。

- **3、准备**

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。实例化不是类加载的一个过程，类加载发生在所有实例化操作之前，并且类加载只进行一次，实例化可以进行多次。

初始值是默认值 0 或 false 或 null。 如果类变量是常量（final），那么会按照表达式来进行初始化，而不是赋值为 0。public static final int value = 123;

- **4、解析**

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。

- **5、初始化**

在准备阶段，变量已经赋过一次系统要求的初始值了，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。

2、类与类加载器【重要】

类加载器虽然只用于实现类的加载动作，但它在 Java 程序中起到的作用却远远不限于类加载阶段。**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**，每个类加载器，都拥有一个独立的类名称空间。换句话说：**比较两个类是否“相等”，只有在这两个类是由同一个类加载器加载的前提下才有意义，否则，即使这两个类来源于同一个Class文件，被同一个虚拟机加载，只要加载它们的类加载器不同，那么这两个类就必定不相等。**

类加载器就是执行上面类加载流程的一些类，系统默认的就有一些类加载器，从 JVM 的角度看，只有两种类加载器：

1、**启动类加载器(Bootstrap ClassLoader)**：这个类加载器是由 C++ 语言实现的，是虚拟机自身的一部分。负责将存在 <JAVA_HOME>\lib 目录中的，或者被 -Xbootclasspath 参数所指定的路径中的类库加载到虚拟机内存中。启动内加载器无法被 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启动类加载器，直接使用 null 即可；

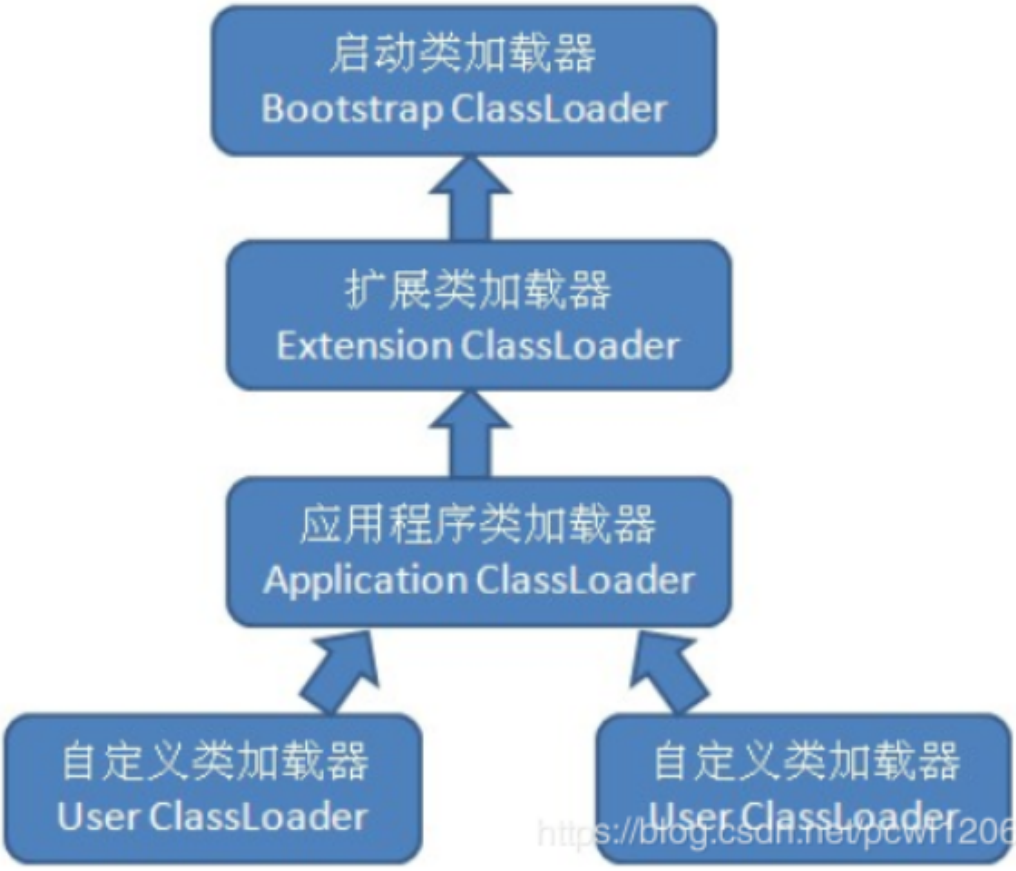
2、**其他类加载器**：由 Java 语言实现，独立于虚拟机外部，并且全都继承自抽象类 java.lang.ClassLoader。如**扩展类加载器**和**应用程序类加载器**：

2.1 **扩展类加载器**(Extension ClassLoader)：这个类加载器由 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。

2.2 **应用程序类加载器**(Application ClassLoader)：这个类加载器由 sun.misc.Launcher$AppClassLoder 实现。由于个类加载器是 ClassLoader 中的 getSystemClassLoader() 方法的返回值，所以一般也称之为系统类加载器。**它负责加载用户路径（ClassPath）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。**

3、双亲委派模型【非常重要】

应用程序一般是由上诉的三种类加载器相互配合进行加载的，如果有必要，还可以加入自己定义的类加载器，它们的关系如下图所示：



3.1 双亲委派模型的工作过程：

如果一个类加载器收到了类加载请求，它首先不会自己去尝试加载这个类，而是把这个请求委派给父类加载器去完成。每一个层次的类加载器都是如此，因此所有的加载请求最终都应该传送到顶层的启动类加载器中，只有当父类加载器反馈自己无法完成这个加载请求（它的搜索范围中没有找到所需的类）时，子加载器才会尝试自己去加载。

3.2 使用双亲委派模型的好处：

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 中，无论哪一个类加载器需要加载这个类，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（使用的是同一个类加载器加载的）。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 java.lang.Object 类，并放在程序的 ClassPath 中，那么系统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，应用程序也将变得一片混乱。

3.3 双亲委派模型的代码实现：

实现双亲委派的代码都集中在 java.lang.ClassLoader 的 loadClass() 方法中，逻辑清晰易懂：**先检查是否已经被加载过，若没有加载则调用父加载器的 loadClass() 方法，**

若父加载器为空则默认使用启动类加载器作为父类加载器。如果父类加载失败，抛出 ClassNotFoundException 异常后，再调用自己的 findClass() 方法进行加载。

补充：若要实现自定义类加载器，只需要继承 java.lang.ClassLoader 类，并且重写其 findClass() 方法即可。

3.4 怎么打破双亲委派模型机制?

1：自己写一个类加载器；

2：重写 loadClass() 方法

3：重写 findClass() 方法

这里最主要的是重写 loadClass 方法，因为双亲委派机制的实现都是通过这个方法实现的，先找父加载器进行加载，如果父加载器无法加载再由自己来进行加载，源码里会直接找到根加载器，重写了这个方法以后就能自己定义加载的方式了。

3.5 有哪些场景是需要打破双亲委派模型的?

https://www.jianshu.com/p/09f73af48a98

JNDI 服务，它的代码由启动类加载器去加载，但 JNDI 的目的就是对资源进行集中管理和查找，它需要调用独立厂商实现部部署在应用程序的 classpath 下的 JNDI 接口提供者(SPI, Service Provider Interface) 的代码，但启动类加载器不可能“认识”之些代码，该怎么办?

为了解决这个困境，Java 设计团队只好引入了一个不太优雅的设计：**线程上下文件类加载器(Thread Context ClassLoader)**。这个类加载器可以通过 java.lang.Thread 类的 setContextClassLoader() 方法进行设置，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，那么这个类加载器默认就是应用程序类加载器。**有了线程上下文类加载器，JNDI 服务使用这个线程上下文类加载器去加载所需要的 SPI 代码，也就是父类加载器请求子类加载器去完成类加载动作，这种行为实际上就是打通了双亲委派模型的层次结构来逆向使用类加载器，已经违背了双亲委派模型，但这也是无可奈何的事情。**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。

十、虚拟机字节码执行引擎

所有的 Java 虚拟机的执行引擎都是一致的：**输入的是字节码文件，处理的过程是字节码解析的等效过程，输出的是执行结果。**

1、运行时栈帧结构

图 8-1 栈帧的概念结构

2、方法调用

方法调用不等同于方法执行，方法调用的唯一的任务就是确定被调用方法的版本（即调用哪一个方法），暂时还不涉及方法内部的具体运行过程。

十一、早期优化和晚期优化

1 早期优化

以 Javac 编译器为例，编译过程大致可以分为 3 个过程：

1、解析与填充符号表的过程；

2、插入式注解处理器的注解处理过程；

3、分析与字节码生成过程。

图 10-4　Javac 的编译过程

1.1 解析与填充符号表

- **词法分析与语法分析**

词法分析器的作用是将 Java 源文件的字符流转变成对应的 Token 流。而语法分析器是将词法分析器分的 Token 流组件成更加结构化的语法树。

- **语义分析：**

1、标注检查

标注检查的检查内容包括：变量使用前是否已被声明、变量与赋值之间的数据类型是否能够匹配。

在标注检查步骤中一个重要的动作称为**常量折叠**，如果我们在代码中写了如下定义：int a = 1 + 2; 那么在语法树上仍然能看到字面量 1、2 以及操作符 +，但是在进过常量折叠之后，他们将会被折叠为字面量 3。

2、数据及控制流分析

数据流分析主要完成如下工作：

1、检查变量在使用前是否都已经被正确赋值；

2、保证 final 修饰的变量不会被重复赋值；

3、要确定方法的返回值类型。这里需要检查方法的返回值类型是否确定，并检查接受这个方法返回值的引用类型是否匹配，如果没有返回值，则不能有任何引用类型指向方法的这个返回值；

4、所有的 Checked Exception 都要捕获或者向上抛出；

5、所有的语句都要被执行到。这里会检查是否有语句出现在一个 return 方法的后面，因为在 return 方法后面的语句永远也不会被执行到。

控制流主要完成如下工作：

1、去掉无用的代码，比如永假的 if 代码块；

2、变量的自动转换，比如自动装箱拆箱；

3、去除语法糖。解语法糖的过程由 desugar() 方法触发，在 com.sun.tools.javac.comp.TransTypes和com.sun.tools.javac.comp.Lower 类中完成；

4、数据流及控制流的分析入口是 flow() 方法，具体操作由 com.sun.tools.javac.comp.Flow 类来完成。

2 晚期优化

2.1 解释器与编译器

解释器：程序可以迅速启动和执行，消耗内存小 （类似人工，成本低，到后期效率低）；

编译器：随着代码频繁执行会将代码编译成本地机器码 （类似机器，成本高，到后期效率高）。

> **为何 HotSpot 虚拟机要使用解释器与编译器并存的架构?**
>
> 在整个虚拟机执行架构中，解释器与编译器经常配合工作，两者各有优势：**当程序需要迅速启动和执行的时候，解释器可以首先发挥作用，省去编译的时间，立即执行。在程序运行后，随着时间的推移，编译器逐渐发挥作用，把越来越多的代码编译成本地代码之后，可以获取更高的执行效率。当程序运行环境中内存资源限制较大（如部分嵌入式系统），可以使用解释执行节约内存，反之可以使用编译执行来提升效率。**
>
> 解释执行可以节约内存，而编译执行可以提升效率。因此，在整个虚拟机执行架构中，解释器与编译器经常配合工作。

2.2 编译对象与触发条件

程序中的代码只有是“热点”代码时，才会被编译为本地代码。在运行过程中会被即时编译的“热点代码”有两类：**被多次调用的方法、被多次执行的循环体。**

判断一段代码是不是“热点”代码，是不是需要触发即时编译，这样的行为称为**热点探测**，目前主要的热点探测判定方式有两种：

1、**基于采样的热点探测**：虚拟机会周期性地检查各个线程的栈顶，如果发现某个（或某些）方法经常出现在栈顶，那这个方法就是“热点方法”。

2、**基于计数器的热点探测**：虚拟机会为每个方法（甚至是代码块）建立计数器，统计方法的执行次数，如果执行次数超过一定的阈值就认为它是“热点方法”。

2.2 编译优化技术

1、方法内联；

2、公共子表达式消除；

3、数组范围检查消除；

4、逃逸分析。。

十二、Java 内存模型

1、概述

处理器和内存不是同数量级，所以需要在中间建立中间层，也就是高速缓存，这会引出缓存一致性问题。在多处理器系统中，每个处理器都有自己的高速缓存，而它们又共享同一主内存（Main Memory），有可能操作同一位置引起各自缓存不一致，这时候需要约定协议在保证一致性。

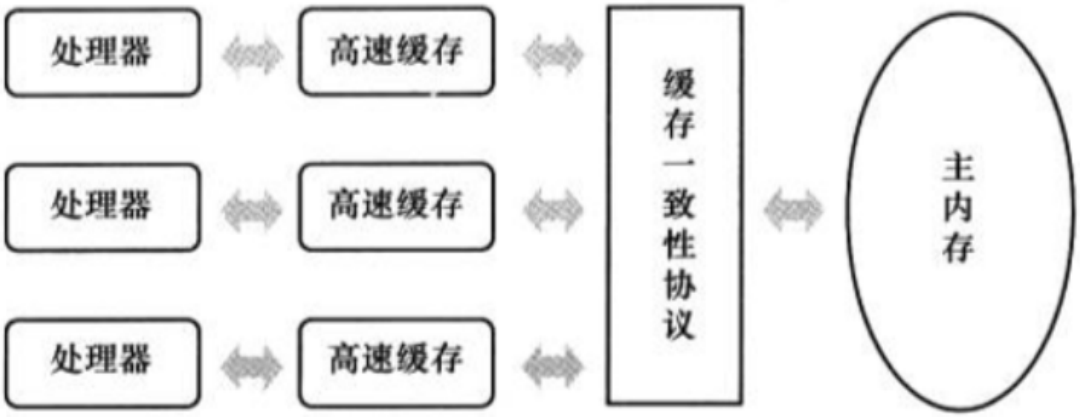


图 12-1　处理器、高速缓存、主内存间的交互关系

2、Java 内存模型

Java 内存模型(Java Memory Model，JMM)：屏蔽掉了各种硬件和操作系统的内存访问差异，以实现让 Java 程序在各种平台下都能达到一致性的内存访问效果。

2.1 主内存与工作内存

Java 内存模型的主要目标是定义程序中各个变量的访问规则，即在虚拟机中将变量存储到内存和从内存中取出变量这样的底层细节。

Java 内存模型规定了所有的变量都存储在主内存（Main Memory）中，每个线程有自己的工作线程（Working Memory），保存主内存副本拷贝和自己私有变量，不同线程不能访问工作内存中的变量。线程间变量值的传递需要通过主内存来完成。

线程、主内存、工作内存交互关系图如下所示：

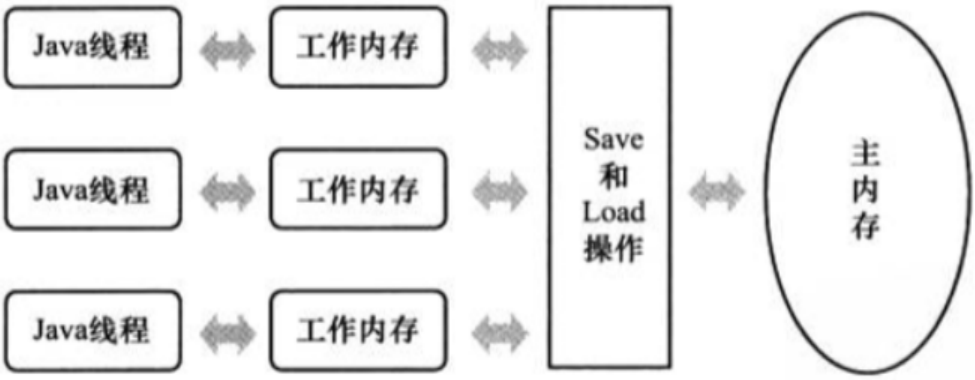


图 12-2　线程、主内存、工作内存三者的交互关系（请与图 12-1 对比）

2.2 内存间的交互操作

关于主内存与工作内存之间的具体的交互协议，即：**一个变量如何从主内存拷贝到工作内存、如何从工作内存同步主内存之类的实现细节**，Java内存模型中定义一下八种操作来完成：

1、**lock(锁定)**： 作用于主内存的变量。它把一个变量标志为一个线程独占的状态；

2、**unlock(解锁)**：作用于主内存的变量，它把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；

3、**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；

4、**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变量值放入工作内存的变量的副本中；

5、**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，每当虚拟机遇到一个需要使用到变量的值的字节码指令时将会执行这个操作；

6、**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值给工作内存的变量，每当虚拟机遇到需要给一个变量赋值的字节码时执行这个操作；

7、**store(存储)**：作用于工作内存的变量。它把一个工作内存中一个变量的值传递到主内存中，以便随后的write操作使用；

8、**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。

如果要把一个变量从工作内存复制到工作内存，那就要按顺序执行 read 和 load 操作，如果要把变量从工作内存同步回主内存，就要按顺序执行 store 和 write 操作。

上诉 8 种基本操作必须满足的规则：

1、不允许 read 和 load、store 和 write 操作之一单独出现；

2、不允许一个线程丢弃它的最近的 assign 操作，即变量在工作内存中改变之后必须把该变化同步回主内存；

3、不允许一个线程无原因地（没有发生过任何 assign 操作）把数据从线程的工作内存同步回主内存中；

4、一个新的变量只能在主内存中“诞生”，不允许在工作内存中直接使用一个未被初始化（load 或 assign）的变量，换句话说就是对一个变量实施 use 和 store 操作之前，必须执行过了 assign 和 load 操作；

5、一个变量在同一时刻只允许一条线程对其进行 lock 操作，但 lock 操作可以被同一线程重复执行多次，多次执行 lock 后，只有执行相同次数的 unlock，变量才会被解锁；

6、如果对一个变量执行 lock 操作，将会清空工作内存中此变量的值，在执行引擎使用这个变量前，需要重新执行 load 或 assign 操作初始化变量的值；

7、如果一个变量事先没有被 lock 操作锁定，则不允许对它执行 unlock 操作，也不允许去 unlock 一个被其他线程锁定主的变量；

8、对一个变量执行 unlock 操作之前，必须先把此变量同步回主内存中（执行 store 和 write 操作）。

十三、其他面试题

1、说一下 Jvm 的主要组成部分？及其作用?

1、类加载器（ClassLoader）

2、运行时数据区（Runtime Data Area）

3、执行引擎（Execution Engine）

4、本地库接口（Native Interface）

组件的作用： 首先通过类加载器（ClassLoader）会把 Java 代码转换成字节码，运行时数据区（Runtime Data Area）再把字节码加载到内存中，而字节码文件只是 JVM 的一套指令集规范，并不能直接交给底层操作系统去执行，因此需要特定的命令解析器执行引擎（Execution Engine），将字节码翻译成底层系统指令，再交由 CPU 去执行，而这个过程中需要调用其他语言的本地库接口（Native Interface）来实现整个程序的功能。

2、说一下堆栈的区别?

1、栈内存存储的是局部变量，而堆内存存储的是实体；

2、栈内存的更新速度要快于堆内存，因为局部变量的生命周期很短；

3、栈内存存放的变量生命周期一旦结束就会被释放，而堆内存存放的实体会被垃圾回收机制不定时的回收。