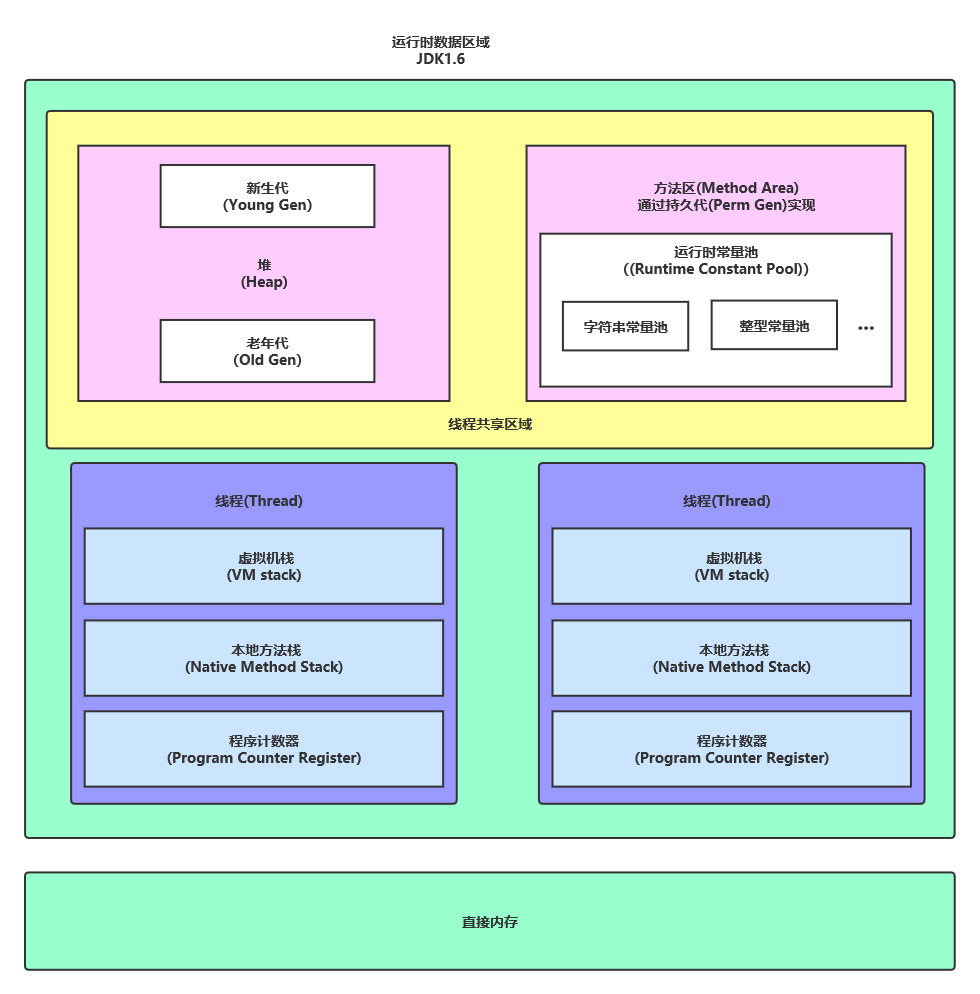
下面所述的java虚拟机特性，都是基于使用最为广泛的HotSpot虚拟机

1. JVM内存管理

不同于C/C++需要维护每一个对象在内存中的生命周期，java将内存控制权交给了java虚拟机，实现了自动化的内存管理，下面我们就来介绍关于关于HotSpot的内存区域

* 1. 运行时数据区域
     1. 概述

Jvm在执行java程序的过程中将其管理的内存划分为若干个不同的数据区域，请注意，从jdk1.8起，数据区域略有不同



从图中我们可以看到，HotSpot的运行时数据区主要被划分为**堆(Heap)、方法区(Method Area)、运行时常量池(Runtime Constant Pool)、java虚拟机栈(VM stack)、本地方法栈(Native Method Stack)、程序计数器(Program Counter Register)、直接内存(Direct Memory)**等几个部分。

**堆(Heap)、方法区(Method Area)**都是所有线程共享的部分，不同的线程拥有各自的**java虚拟机栈(VM stack)、本地方法栈(Native Method Stack)、程序计数器(Program Counter Register)** ，**直接内存(Direct Memory)**不属于jvm运行时数据区，不由jvm管理和回收，需要我们手动的回收，但它却会被频繁的使用，因此也放到内存模型中讲解

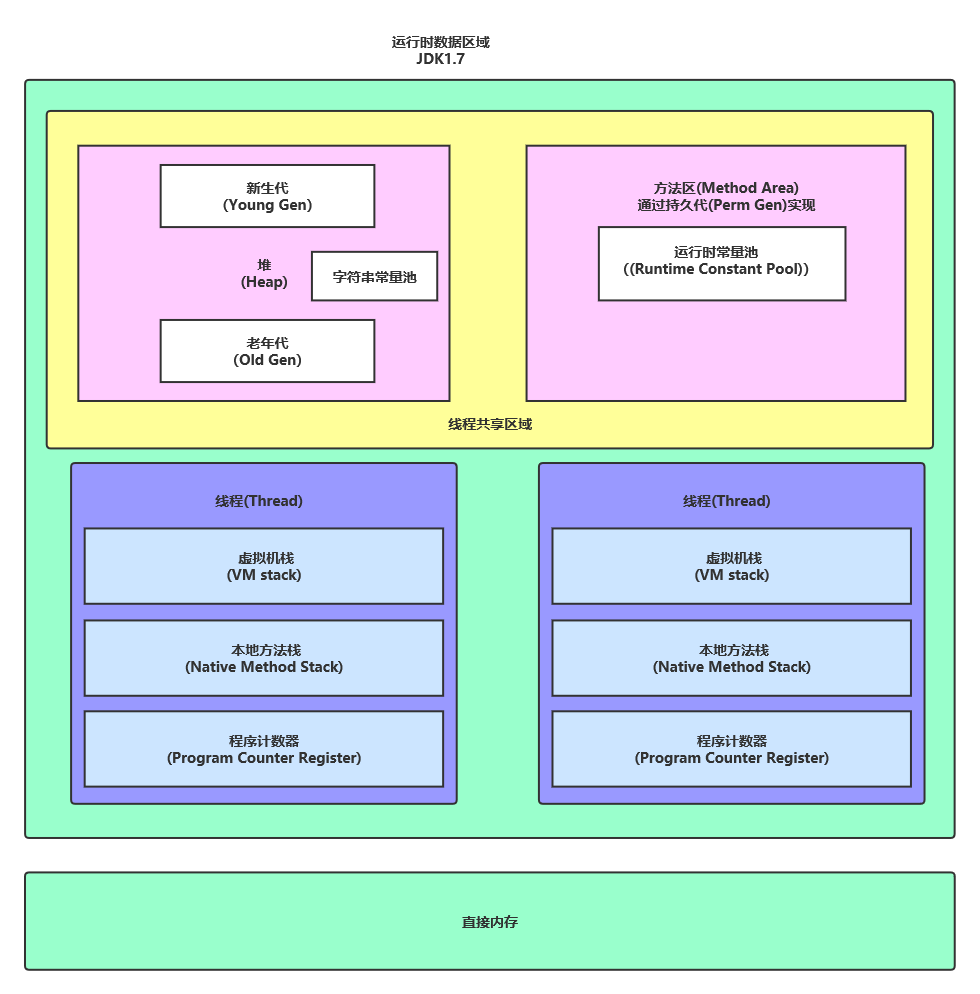
* + 1. 版本变动

#### 1.1.2.1 java1.6到1.7

在1.6版本中，**运行时常量池(Runtime Constant Pool)**被放在方法区中，**字符串常量池(String Constant Pool)**在**运行时常量池(Runtime Constant Pool)**中

从1.7版本起

1. **符号引用(Symbols)**转移到了native heap
2. **字面量(interned strings)**转移到了 heap，**字符串常量池(String Constant Pool)**在JDK7中从永久代中转移到**堆(heap)**中，但永久代仍存在于JDK1.7中，并没完全移除
3. **类的静态变量(class statics)**转移到了heap



在Hotspot中，方法区只是在逻辑上独立，物理上还是包含在堆区中。本来堆中只有新生代和老年代，HotSpot的设计团队为了让堆中的GC收集器可以像管理java堆一样来管理方法区，省去编写方法区内存管理代码的工作，才将GC分代收集扩展至方法区，这才有了永久代

intern()的作用

当常量池中不存在"abc"这个字符串的引用，将这个对象的引用加入常量池，返回这个对象的引用。

当常量池中存在"abc"这个字符串的引用，返回这个对象的引用；

String str1 = new String("hello") + new String("world");  
str1.intern();  
String str2 = "helloworld";  
System.*out*.println(str1 == str2);//JDK6及之前的版本是false，从JDK7起是true  
System.*out*.println(str1.intern() == str2);//true

因为JDK 6对于首次遇到的字符串，intern()会复制一份到常量池并返回其引用，此时str1是堆中的对象地址，而str2是方法区中字符串常量池的地址，所以为false。

而JDK7将字符串常量池移动到堆中了，不会拷贝字符串到永久代了，而是将堆中的这个对象的引用直接存到常量池中，减少内存开销然后返回，还是同一个(都是堆中的地址)，所以为true，如下图：

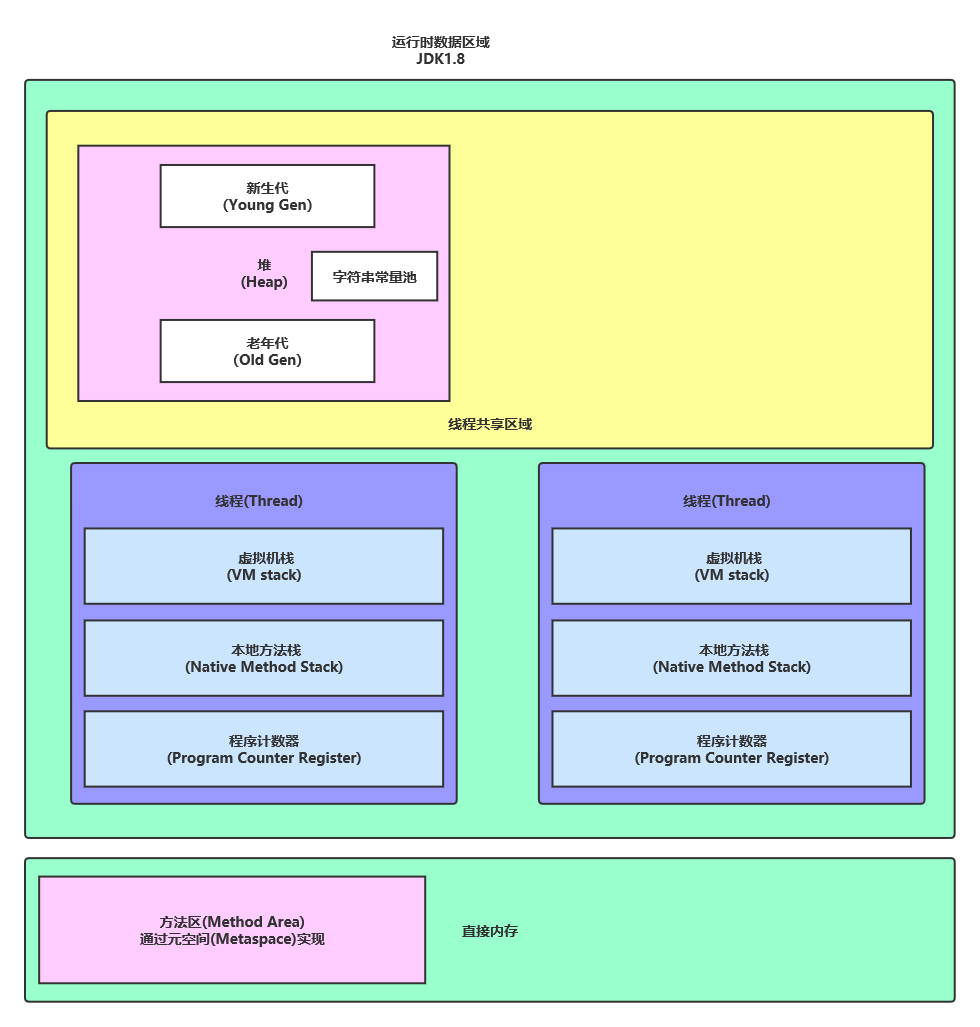


#### 1.1.2.2 java1.7到1.8

从jdk1.8起，**方法区(Method Area)**不再由**永久代(PermGen)**实现，而是通过**元空间(Metaspace)**实现，HotSpot开发团队将永久代中的剩余内容(主要是类型信息)全部移到元空间中。**元空间(Metaspace)**并不在虚拟机中，而是使用**直接内存(Direct Memory)**。请注意，1.8并没有移除方法区，而是更改了方法区的实现方式。永久代被彻底移除后， ‑XX:MaxPermSize 参数失去了意义，取而代之的是-XX:MaxMetaspaceSize

那么改动的原因是什么呢？

1. **为融合HotSpot VM与 JRockit VM**
2. **难以设置永久代的大小，永久代内存经常不够用或发生内存泄露，抛出OutOfMemoryError异常**
3. **永久代的GC回收效率低下**



<https://blog.csdn.net/Mrsssswan/article/details/87938193>

<http://warrentalk.site/2017/07/13/%E4%BB%8EJava6%E5%88%B0Java8-%E4%BD%A0%E5%BA%94%E8%AF%A5%E7%9F%A5%E9%81%93%E7%9A%84JVM%E6%96%B0%E7%89%B9%E6%80%A7/>

### 1.1.3 java堆

对于绝大多数应用来说，java堆是jvm五个运行时数据区中占有内存最大的一块了，绝大多数的对象实例都是在堆中分配的。它在虚拟机启动时被创建，由所有内存共享。它分为**新生代(Young Gen)**和**老年代(Old Gen)**，GC的主要工作区域就在这里。当java heap因为没有充足内存而无法完成实例分配并且无法扩展时，将会抛出OutOfMemoryError

**类的成员变量、静态变量、数组都是在java的堆内存里面分配空间的**。这里需要特别说明的是，当我们用new创建一个数组时，虚拟机会在**堆**内存中分配一个数组并返回它的引用(reference)，所以在局部变量表中数组和对象一样存储的也是引用地址，该引用都指向堆内存。类的普通成员变量，无论是基本类型还是引用类型，都是作为new出来的对象的属性存储在堆中的。类的静态变量，是跟随Class对象存储在堆中的

**局部变量和方法参数都是在虚拟机栈的局部变量表中分配内存的。**需要特别指出，如果局部变量是reference类型，那么它指向的对象是存储在java堆中的，可以被各个线程共享的。但是reference类型本身是存储在java虚拟机栈的局部变量表中的，是线程私有的

### 1.1.4 方法区

方法区是java6、7、8三个版本HotSpot主要改动的部分。同样的它也是所有内存共享的区域，它用于存储已被虚拟机加载的类信息、常量、静态变量(1.7起在堆中)等数据

说到方法区，就不得不提起永久代。由于java堆是采用分代设计的，在jdk8以前，HotSpot团队选择将收集器的分代设计扩展至方法区，也就是使用永久代来实现方法区，这么做的好处在于HotSpot垃圾收集器可以像管理java堆一样管理方法区的内存，省去了专门为方法区编写内存管理代码的工作。但根据以后的表现来看，这么做是不合适的。因此在jdk7中，HotSpot开发团队将字符串常量池、静态变量等数据移到堆中，到了jdk8时，彻底废弃了永久代的概念，改为使用在本地内存中实现的元空间来实现方法区，把永久代中还剩余的内容(主要是类型信息)转移到元空间中

### 1.1.5 运行时常量池

运行时常量池(Runtime Constants Pool)是方法区的一部分，Class文件中除了有类的版本、字段、方法、接口等信息描述外，还有一项信息是常量池表，它用于存放编译期间生成的各种字面量和符号引用，在类加载后，这部分内容将被放到方法区的运行时常量池中

### 1.1.6 直接内存

直接内存(Direct Memory)并不属于运行时数据区，也就是说它不属于jvm。但由于这部分内存也被频繁使用，因此放在这里讲解

在JDK4中新加入的NIO类引入了一种基于通道和缓冲区的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

JDK8中方法区的实现—元空间是存储在直接内存中的

直接内存虽然使用了本地内存，其大小不受虚拟机大小的限制，但最终大小还是受到系统总内存等因素的限制，并不是无限大的

### 1.1.7 元空间

参照上节直接内存，元空间是实现在直接内存中的，这样保证了元空间很难出现内存不够的情况，因为他是直接使用的系统内存。元空间用于存储部分方法区的数据，如类和类加载器的元数据信息。而符号引用和字符串常量等都存储在堆内存中

### 1.1.8 虚拟机栈

Java虚拟机栈(Java Virtual Machine Stack)是线程私有的，它描述的是java方法执行的线程内存模型。每个方法被执行的时候，Java虚拟机都会同步创建一个栈帧(Stack Frame)用于存储局部变量表、操作数栈、动态连接、方法出口等信息。每一个方法从调用到执行完毕，都对应一个栈帧从入栈到出栈的过程

局部变量表存放了编译期可知的各种Java基本数据类型如boolean、byte、char、short、int、float、long、double等，也包括对象引用(这里只是引用类型，对象的实体存储在堆中，可以把它理解为一个指向对象起始地址的引用指针)。和ReturnAddress(指向一条字节码指令的地址)类型

### 1.1.9 本地方法栈

本地方法栈与虚拟机栈发挥的作用是一致的，区别在于虚拟机栈为执行JAVA方法服务，而本地方法栈为执行Native方法服务

### 1.1.10 程序计数器

程序计数器(Program Counter Register)是一块较小的内存空间，各个线程都有各自的程序计数器。它可以被看做是当前线程所执行字节码的行号指示器，字节码解释器在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，它是程序控制流的指示器，分支、循环、跳转、异常处理、线程恢复等功能都依赖这个计数器完成

如果线程正在执行的是一个java方法，那么程序计数器记录的是正在执行的虚拟机字节码指令的地址。如果正在执行的是一个Native方法，则计数器值应为空

**此内存区域是唯一一个在Java虚拟机规范中没有规定任何OutOfMemoryError情况的区域**

* 1. HotSpot对象探秘

### 1.2.1对象的创建

一个java对象的创建，一般要经过如下几个步骤

1. **检查常量池中是否有对象所属类的符号引用，如果没有，说明类还未定义，抛出ClassNotFoundException**
2. **检查符号引用所代表的类是否已被加载、解析和初始化过。如果还没有，就先进行类的加载**
3. **根据方法区中存储的类信息确定创建该类对象需要的内存大小。一个对象的内存大小在类定义完就可以确定，且一个类实例化的所有对象其占用内存都是一样的**
4. **在堆中划分一块空闲的内存空间给该类对象**

主要有两种方式：

1. **指针碰撞：**

如果垃圾处理器采用了复制算法或标记-整理算法，那么堆中的内存是规整的，使用过的内存被放在一边，空闲的内存被放在另一边，中间放着一个指针作为分界点。当需要分配内存时，只需要移动指针即可

1. **空闲列表**

如果垃圾处理器采用标记-清除算法，那么堆中的空闲区域和已使用区域交错，虚拟机就需要维护一个列表记录那些区域是空闲区域。在分配内存时从列表中找到一块足够大的空间划分给对象实例，并更新列表上的记录

**JVM究竟采用哪种内存分配方法，取决于它使用了哪种垃圾收集器**

1. **虚拟机将分配到的内存空间都初始化为对应值(如果不特别指定的话都初始化为各数据类型的零值)，这一步操作保证了对象的字段在java代码中不赋初值就可以直接使用**
2. **设置对象头中的信息**
3. **调用构造函数进行初始化**

由于堆内存是线程共享的，因此在内存分配的过程中也会出现并发问题，如A对象的内存还为分配完，B对象也分配与之相同的内存。HotSpot才用了两种方式来处理并发

1. 才用CAS + 失败重试保证更新操作的原子性
2. 在JDK6中引入了本地线程分配缓冲(TLAB)，把分配内存的动作按线程划分在不同的内存中进行，哪个线程需要分配内存，就在对应线程的本地缓冲区中分配。这样就不存在竞争的情况，可以大大提升分配效率，当Buffer容量不够的时候，再重新从Eden区域申请一块新的区域继续使用，这个申请动作还是需要原子操作的。

### 1.2.2对象的内存布局

#### 1.2.2.1对象的内存结构

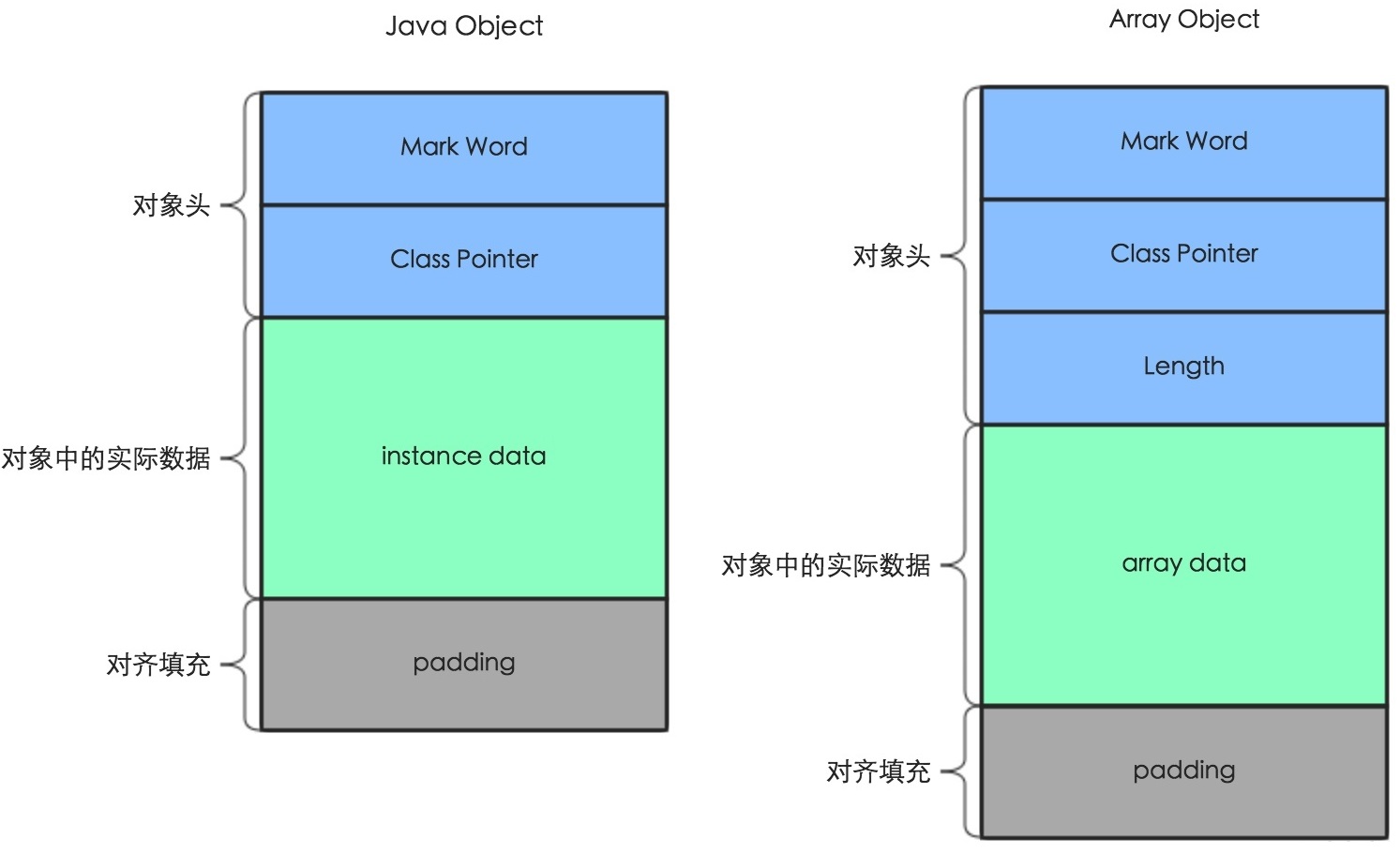
在HotSpot虚拟机中，对象在堆内存中可以被划分为三个部分：**对象头(Header)、实例数据(Instance Data)和对齐填充(Padding)**。

java里面数据类型主要分为原生类型，类，接口，数组这几种。原生类型包括了int、long、double、char、byte等原生类型。原生类型对应的包装类型如Integer、Long、Double这些在java里面归类到类(class)里面，这里为了描述方便把接口(interface)也归类到类(class)里面。所以总的来说java的数据类型主要归为三类**原生类型，类(class)，数组**。其中只有类和数组在java虚拟机里面是对象的形式的创建的，所以我们说的对象通常就是指类和数组的实例。

**对象头(Header)**包括两类信息：

1. **Mark Word：**用于存储对象运行时数据，如：哈希码(Hash Code)、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，这部分数据长度在32位虚拟机和64位虚拟机中大小分别为32bit和64bit
2. **Class Pointer：**类型指针中存储了指向对象类型信息的指针。其占用空间大小和Mark Word相同

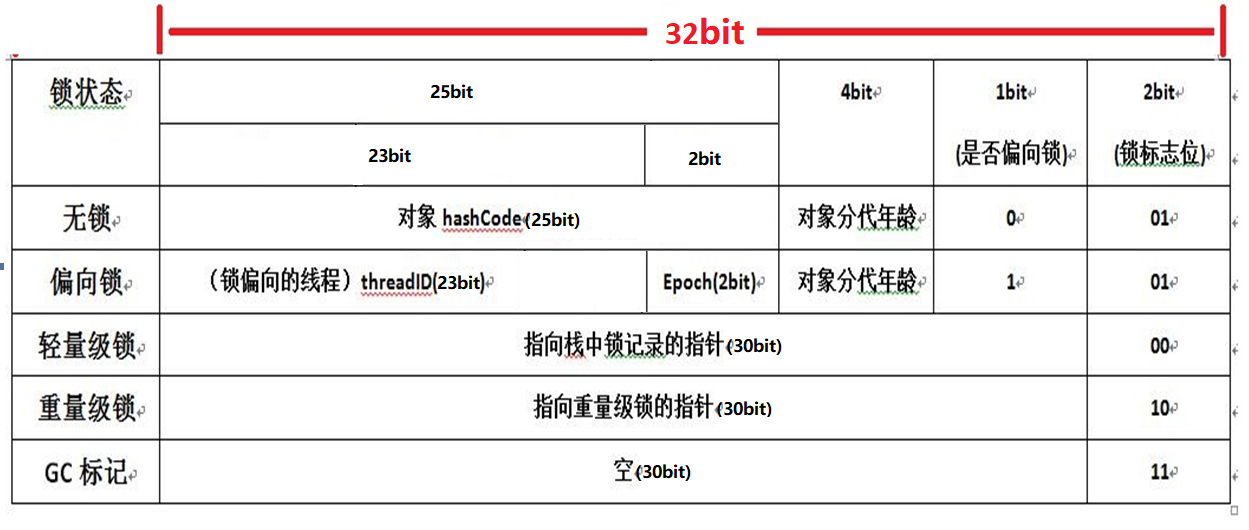
数组对象基本和类对象一致，唯一的区别在于数组对象头中多了一块用于记录数组长度的数据，长度为4字节(32bit)，如下图：



**实例数据(Instance Data)** 是对象真正存储的有效信息，也是我们程序代码里面定义的各种类型字段的内容，无论是在父类中继承下来的还是在子类中定义的都要记录下来。

**对齐填充(Padding)** 并不是必须的，hotspot 虚拟机中要求对象的存储空间必须是8个字节的整数倍，如果一个对象大小不足8个字节的整数倍，就需要用对齐填充进行补充。它没有特别的涵义，仅仅起到占位符的作用

下图分别是32位虚拟机的Mark Word和64位虚拟机的Mark Word：





#### 1.2.2.2 Synchronized锁膨胀

从JAVA6开始，synchronized锁分为三种状态：偏向锁，轻量级锁和重量级锁状态，这几个状态会随着线程竞争情况逐渐升级。当某个对象被Synchronzied关键字锁住后：

##### 1.2.2.2.1偏向锁

当线程A访问锁资源时，如果该锁并未被任何线程持有(无锁状态)，那么将在MarkWord中通过CAS操作存储线程A的ID和时间戳Epoch(真实地生成一个时间戳代价还是蛮大的，因此这里应当理解为一种类似时间戳的id)，同时将偏向锁标识设为1。此后，线程A如果再次访问锁资源，只需要比较当前线程的threadID和MarkWord中的threadID是否一致，如果一致，则无需使用CAS来加锁、解锁(可重入)

如果线程B尝试访问该锁资源，首先判断当前锁的MarkWord中是否标记为偏向锁。这里由于线程A已获取偏向锁，接下来线程B将判断拥有偏向锁的线程A是否还存活

如果线程A不存活，重新设置偏向锁标记为0**(线程A执行完毕后，不会主动去释放偏向锁，等到竞争出现才释放锁)**，接着使用cas替换偏向锁threadID为线程B的ID，锁不升级，仍为偏向锁

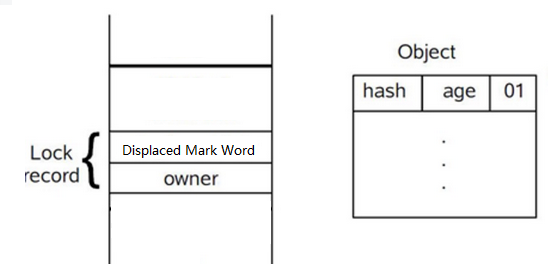
如果线程A还存活，首先暂停线程A，马上执行线程A的操作栈，检查锁的使用情况，如果线程A仍然需要持有偏向锁，则偏向锁升级为轻量级锁**（偏向锁就是这个时候升级为轻量级锁的）**，最后唤醒线程A。如果线程A不需要使用了，则将实例对象恢复成无锁状态，然后重新偏向线程B。

偏向锁假定将来只有一个线程会申请锁，减少无竞争且只有一个线程使用锁的情况下，使用轻量级锁产生的性能消耗

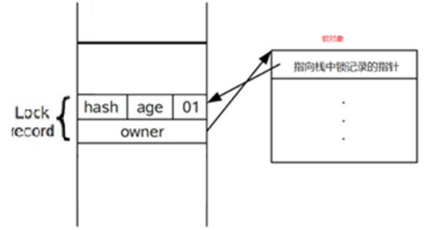
##### 1.2.2.2.2轻量级锁

首先在当前线程的桢栈中创建一块lock record空间，这个lock record分为两个部分：

1. **Displaced Mark Word：**用于存储mark word的拷贝
2. **Owner：**指向锁对象的mark word，用于识别被锁住的对象是哪一个



接着将锁住对象的mark word拷贝到**Displaced Mark Word**中用于CAS操作(CAS需要内存位置(V)、预期原来值(A)以及期待更新值(B))，这里的**Displaced Mark Word**就作为预期原来值(A)，将**Owner**指向锁对象的mark word也用于CAS操作，作为内存位置(V)，通过CAS操作尝试将锁对象的mark word的前62bit(或前30bit，取决于虚拟机位数)替换为lock record的内存地址。



如果CAS操作失败，尝试自旋重新操作，当自旋超过一定的次数，或者一个线程在持有锁，一个在自旋，又有第三个线程访问时，说明当前场景不适合轻量级锁，将膨胀为重量级锁

如果CAS操作成功，则线程成功获取轻量级锁，接着执行锁内代码。当执行完毕释放锁时，使用CAS将Displaced Mark Word中存储的拷贝替换回到锁对象的mark word中，若替换成功，则解锁完成。若替换不成功，表示在当前线程持有锁的这段时间内，其他线程也竞争过锁，并且锁已经膨胀成了重量锁(重量级锁就是传统的synchronzied)，那么就要去唤醒一个被挂起的线程去重新竞争锁

* 1. 垃圾回收机制

HotSpot的运行时数据区域中，程序计数器、本地方法栈、虚拟机方法栈3个区域随线程而生，也随线程而灭。这三个区域的内存分配和回收是在类结构确定下来的时候就基本确定，因此在这三个区域基本不需要考虑垃圾回收的问题。而JAVA堆和方法区的内存分配和回收则具有相当的不确定性，需要我们重点关注其垃圾回收，后续的垃圾回收都是针对这两个区域进行的内存管理

* + 1. 对象存活判断

在进行垃圾回收之前，我们首先需要确定哪些对象需要被回收

* + - 1. 引用计数法

引用计数法是在每一个对象中添加一个引用计数器，当有一个地方引用它时，引用计数器值就加1，当引用失效时，引用计数器减1，当计数器为0时，就认定对象已死亡。

这种方法虽然占用了一点额外的内存空间，但其最大的优点在于简单，判定效率高。但它很难解决对象的循环引用问题，如：

CycleReference aReference = new CycleReference();  
CycleReference bReference = new CycleReference();  
//设置循环  
aReference.instance = bReference;  
bReference.instance = aReference;  
//将引用置空，使堆中的对象不可达  
aReference = null;  
bReference = null;  
//建议虚拟机进行垃圾回收  
System.*gc*();

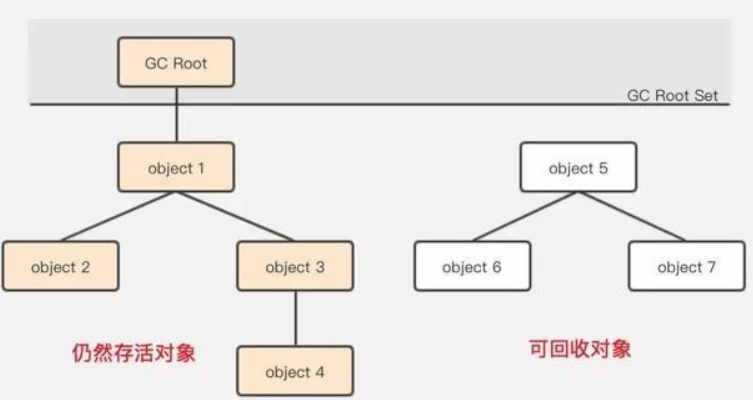
如果采用引用计数法，那么即使上述的两个堆中对象已不可访问，但垃圾回收器仍然无法回收他们，因为他们互相引用对方，其引用计数不为零

正因为有这样的缺陷，HotSpot并没有采用这种方式来判断对象是否存活

redis设计过程中是采用了引用计数法来进行内存回收的

* + - 1. 可达性分析算法

当前的主流商用程序语言都是采用可达性分析算法判定对象是否存活的。该算法从GC Roots开始向下搜索，搜索所走过的路径称为引用链。当一个对象到GC Roots没有任何引用链相连时，则证明此对象是不可达对象



在JAVA中，可作为GC Root的对象包括以下几种：

1. 虚拟机栈(栈帧中的本地变量表)中引用的对象。如：各线程中使用到的参数、局部变量、临时变量等
2. 类静态属性引用的对象
3. 常量引用的对象，如：字符串常量池里的引用
4. 本地方法栈中(Native方法)引用的对象
5. 所有被Synchronized关键字持有的对象

下面我们将介绍可达性分析算法的一些细节

* + - * 1. 准确式GC or 保守式GC

根搜索算法中需要在堆外查找引用，事实上它们都是不明确的根。以栈举例，栈帧里装的有int、double这样的数值，也有引用类型变量的地址，而这些值在GC看来就是一堆位排列，无法分辨究竟是不是引用，所以才叫不明确的根。

根据是否能够识别引用，GC分为准确式gc和保守式gc，目前主流的java虚拟机都是采用准确式GC

保守式GC：不能识别指针和非指针的GC

由于保守式GC 无法精确识别引用，因此只能采取一种保守的态度，把所有可疑的引用均当作指针，识别可疑引用的条件如下：

1. **是不是正确的对齐值（32位CPU，引用是4字节对齐）**
2. **是否指着堆内**
3. **是否指着对象的开头**

**优点：**

1. 实现简单，不需要依赖于语言处理程序的帮助，可以很方便的接入语言处理程序而无须再进行大量的改写

**缺点：**

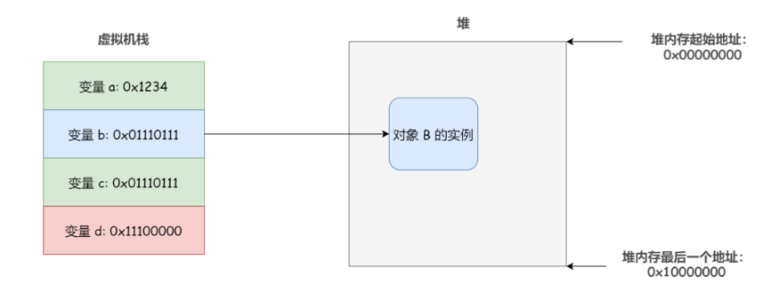
1. 识别指针和非指针需要耗费成本
2. 错误识别的指针会被当成活动的对象，包括其子对象，可能会造成垃圾对象压迫堆，导致堆缓慢增大
3. 能够使用的GC算法有限，所有包含需要移动对象的算法（标记复制、标记整理）均不可用。因为在移动或者压缩的过程中需要移动对象，对象的内存地址变了，就需要修改指针的指向，如果有一个非指针被误认为是指针，然后被修改了，就会引发意想不到的bug

我们假设有变量a、b、c、d

因为引用是一个地址，JVM 中地址是 32 位的，也就是 8 位的 16 进制，很明显 a 是一个 4 位 16 进制，不能作为引用（这里的专业术语叫对齐检查），因此，保守式GC可以判断出变量a不是指针

但对于变量b和c，因为变量都是32位的，保守式GC无法判断出他们是否为指针，因此，只能将二者都当做指针来看待

假设恰巧b和c的值相同，但b是引用类型，c是int类型，b指向堆中对象B的实例，如果此时我们执行b=null，很显然，对象B不再有引用指向它，因此其应该作为垃圾回收掉，但由于c变量的值和b变量相同，因此即使c实际上不是指针，保守式GC也只能怀疑变量c是指向对象B的指针，因此不能回收对象B。随着时间推移，这样的情况累积下去，最终导致堆缓慢变大，最后压迫堆



同样的，如果使用标记复制和标记整理类型的算法，需要拷贝对象，假设此时正在进行一场MinorGC，对象B需要从Eden区复制到Survivor区，那么他们的引用值就不再是0x01110111了，因此需要修改指向其的引用的值。但是由于保守式GC无法识别变量b、c是否为引用，只能将其都当做引用看待，最后会导致int类型的变量c被修改，产生意想不到的bug

准确式GC：能识别指针和非指针的GC

想要准确识别这些不确定的根有很多种方式，但是它们都有一个共通点，就是需要语言处理程序的支援，而得到语言处理程序支援准确识别了引用的这种GC，就是准确式GC

**优点：**

1. 堆中只存在活跃对象，而不存在错误识别导致压迫堆的问题
2. 兼容绝大部分GC算法

**缺点：**

1. 实现复杂，需要依赖于语言处理程序的帮助，比如OopMap
   * + - 1. 根节点枚举

上面我们说到了，要进行可达性分析，必须先获取GC Roots，GC Roots主要在全局性的引用(常量、类静态属性)和执行上下文(虚拟机栈和本地方法栈中的引用)中。尽管目标明确，但查找起来并不是一件容易的事，因为栈上的本地变量表里面只有一部分数据是 Reference 类型的（它们是我们所需要的），而那些非 Reference 类型的数据对我们而言毫无用处，但我们还是不得不对整个栈全部扫描一遍，这是对时间和资源的一种浪费。**迄今为止，所有垃圾回收器(即使是CMS、G1)在根节点枚举这一步都需要暂停用户线程**，即stop the world(STW)，如果是逐个查找GC Roots必然要消耗不少时间，导致过长的STW

由于主流垃圾处理器都采用的是准确式GC，因此可以准确地识别出引用和非引用，这是怎么做到的呢？就是通过引入OopMap，OopMap全名叫ordinary object pointer map，即普通对象指针map。

这其实就是一种空间换时间的方式，在某个时间点把栈上代表引用的位置全部记录在oopMap中，这样到真正 gc 的时候就可以直接读取，而不用再一点一点的扫描了

我们都知道，在 JVM中，一个线程为一个栈，一个栈由多个桢栈(关于帧栈的定义可以参见**运行时帧栈**一节)组成，一个桢栈对应一个方法，一个栈帧中就可能有多个 OopMap。 gc 发生时，程序首先运行到最近的一个**安全点**停下来，然后更新自己的 OopMap ，记下栈上哪些位置代表着引用。

当需要枚举根节点时，只需要递归遍历每个栈帧的 OopMap ，通过其中记录的被引用对象的内存地址，即可找到GC Roots，这样就可以避免在根节点枚举阶段发生过久的STW

安全点safe-point

每次生成OopMap必然要带来额外的开销，如果每次遇到了会改变引用关系，或者说会改变OopMap内容的指令，都生成对应的OopMap，那么将会生成海量的OopMap，需要消耗大量的存储空间

那么是否可以指定某个特定的程序位置，要求当程序执行到这个位置时，再生成OopMap，那么这个点就被称为安全点safe-point

有了安全点，意味着程序不能在程序指令流的任意位置停下来进行gc，必须在安全点停下来才能进行gc，因此安全点不宜太少导致gc等待间隔过长，也不宜过多导致内存负荷加大，其原则就是可以让程序一直安全的跑下去，不能让太多的死对象占据内存。因此遇到长时间执行的指令的时候就给它 GC 一下，类如方法的调用、循环跳转、异常跳转等，遇到类似的执行才会生成 Safepoint

那么此时又有一个问题产生了，safepoint是对于线程来说的，那么现代程序都是多线程程序，如果没有发生gc时，多个线程到达safepoint的时间可以不一致，但是如果需要进行gc时，多个线程必须都要执行到safepoint并停顿在安全点，否则没有办法获取到同一时刻每个线程的OopMap。那么如何在gc的时候保证所有的线程都跑到最近的安全点并停顿下来呢？这里主要有如下两个方式

抢先式中断

这种方式不需要线程的执行代码主动配合，在gc发生时，jvm首先把所有用户线程全部中断，如果发现有用户线程中断的地方不在safepoint上，那么就恢复这条线程，让它跑到最近的安全点上再中断。现在几乎没有虚拟机采用抢先式中断来暂停线程以响应gc事件

主动式中断

这种方式是当发生gc需要中断线程时，不直接中断线程，而是设置一个标志位，每个线程在执行过程中需要不断轮询这个标志位，一旦发生中断标志为真，线程就自己在最近的安全点上主动中断挂起

由此可见，这个轮询标志位的操作在线程执行过程中会不断运行，这要求它必须足够高效，否则会影响线程执行效率，HotSpot采用内存保护陷阱机制，将轮询标志位的操作精简到只有一条汇编指令的程度

安全区域safe-region

安全点机制看起来已经很完美了，但是在实际情况中，线程并不总是处于running状态的，如果用户线程处于sleep状态或者blocked状态，在主动式中断的情况下，用户线程将无法走到安全点去中断挂起自己，对于这种情况，必须引入安全区域safe-region来解决

safe-region能够确保在某一段代码片段中，引用关系(即OopMap需要存储的内容)不会发生变化，因此在这个区域的任意地方开始gc都是安全的。我们完全可以把安全区域看做扩展版的安全点，典型的安全区域如：线程被挂起，或者线程执行JNI函数等等。

当用户线程执行到安全区域的代码时，首先会标识自己已经进入safe-region，如果此时发生gc，jvm就不必管处于safe-region中的线程。当线程要离开safe-region时，它会检查jvm是否已经完成了根节点枚举或者其他需要暂停用户线程的阶段，如果未完成，线程就必须等待，直到收到可以离开的信号

* + - * 1. 并发的可达性分析

可达性分析算法理论上要求全过程都基于一个能保障一致性的快照中进行才能进行分析，因此必须冻结用户线程的运行。

我们已经通过根节点枚举优化了查找GC Roots的过程，但接下来的”标记”阶段，如果堆过大，依然会造成用户线程停顿时间较长的问题，如果能够消减这部分时间，那么收益也将是巨大的

三色标记

首先我们需要弄清楚为什么必须在一个能保障一致性的快照中进行才能进行对象图的遍历，为此我们引入三色标记来辅助推导：

1. **白色：**对象尚未被垃圾收集器访问过。在可达性分析刚开始时，所有对象都是白色的，当可达性分析结束时，仍然白色的对象则代表不可达
2. **灰色：**表示对象已经被垃圾收集器访问过，但该对象上仍存在至少一个引用还没有被扫描过
3. **黑色：**表示对象及对象的所有引用都被已经扫描过了，当可达性分析结束时，所有黑色对象则代表本次分析出的应存活对象

由此，三色标记的遍历过程如下：

1. **初始时，所有对象都在 【白色集合】中；**
2. **将GC Roots 直接引用到的对象 和 跨代引用对象 挪到 【灰色集合】中；**
3. **递归执行：  
   3.1. 将本对象 引用到的 其他对象 全部挪到 【灰色集合】中；  
   3.2. 将本对象 挪到 【黑色集合】里面。**
4. **重复步骤3，直至【灰色集合】为空时结束。**
5. **结束后，仍在【白色集合】的对象即为GC Roots 不可达，可以进行回收。**

三色标记的缺陷

由此我们可以推断，如果在一个不能保证一致性的快照中进行分析，即垃圾收集器和用户线程并行工作时，如果可能会出现错误标记，主要有如下两种错误。

1. **浮动垃圾：**把白色对象错误标记为黑色，那没关系，不过是产生一些浮动垃圾罢了，下次清理收集就好了；
2. **对象消失：**把黑色对象错误标记为白色对象，将会导致将原本应存活的对象给清除了，这是非常致命的错误

三色标记缺陷的解决方案

Wilson于1994年证明了，当且仅当以下两个条件满足时，会误将原本黑色的对象错误标记为白色对象，导致应存活对象被清除：

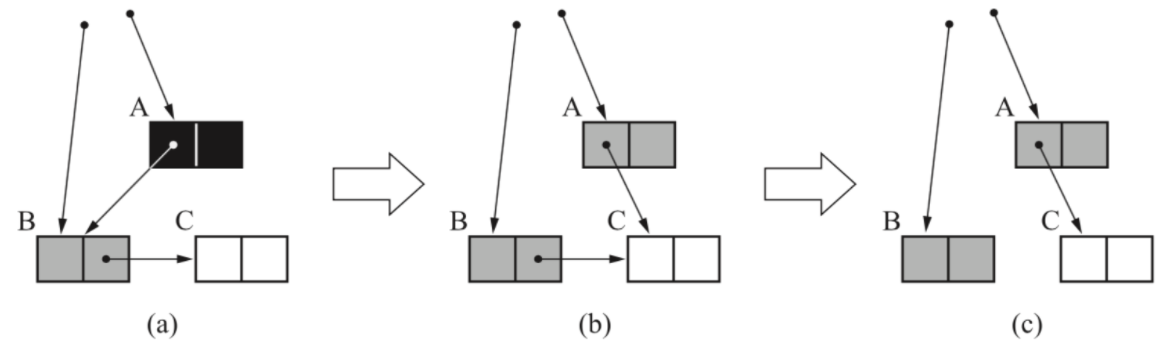
1. **赋值器插入了一条或多条从黑色对象到白色对象的新引用**
2. **赋值器删除了全部从灰色对象到该白色对象的直接或间接引用**

因此，如果希望用户线程和收集器并发工作，我们只需要破坏以上两个条件中的一个即可。由此分别产生了两种方案：增量更新和原始快照

增量更新

增量更新破坏第一个条件，即**赋值器插入了一条或多条从黑色对象到白色对象的新引用**

当黑色对象插入新的指向白色对象的引用时，就将这个新插入的引用记录下来，等待并发标记过后，再以该黑色对象为根，重新扫描一次。换言之，黑色对象一旦插入了一条或多条指向白色对象的新引用，就将黑色对象标记为灰色，等待下次重新扫描，如下图：



典型的应用如老年代垃圾回收器CMS，CMS分为四个阶段：

1. **初始标记：**这一阶段主要是根节点枚举，找出所有的GC Roots和新生代引用老年代产生的跨代引用对象，这一阶段会发生STW
2. **并发标记：**用户线程和垃圾回收线程并行执行，这一阶段会产生浮动垃圾
3. **重新标记：**通过增量更新来重新标记，这一阶段会发生STW
4. **并发清除**：清理删除掉标记阶段判定的已死亡对象

原始快照（SATB）

原始快照破坏第二个条件，即**赋值器删除了全部从灰色对象到该白色对象的直接或间接引用**

当灰色对象要删除指向白色对象的引用关系时，就将这个要删除的引用关系记录下来，并发标记结束后，再将以该灰色对象为根，重新扫描一次。换言之，就是无论引用关系删除与否，都会按照刚开始扫描那一刻的对象图快照来进行搜索

比如将引用B到D 的引用改为B到C时，通过写前屏障，会把B到D 的引用记录下来，保证还可以通过B对象遍历到D对象，在最终标记时，再以B对象为根，重新扫描一次

典型的如G1

cms和G1二次扫描的对比

* 1. **CMS采用增量更新**

**优点：**浮动垃圾少，因为所有黑色对象插入白色对象引用后，黑色对象都会变灰，需要重新扫描

**缺点：**由于跨代引用的存在(cms的卡表并不会记录新生代引用老年代的引用)，导致cms在remark阶段需要扫描整个新生代，并将新生代存活的作为GCRoots的一部分，因此CMS remark会比较慢

CMS的remark需要重新扫描mod-union table里的dirty card外加整个根集合，而此时整个young gen（不管对象死活）都会被当作根集合的一部分，因而CMS remark有可能会非常慢

* 1. **G1采用原始快照**

**优点：**二次标记阶段耗时较短，因为它只是以灰色对象为根，再进行依次扫描

**缺点：**按照最开始的对象图快照进行扫描，意味着可能某个黑s色对象已经不再活跃了，却仍被当作活跃对象，会产生较多的浮动垃圾

* + 1. 垃圾收集算法
       1. 分代收集

HotSpot的垃圾处理器，大多遵循了分代收集的规则进行设计。分代收集主要建立在两个假说上：

1. **弱分代假说：绝大多数对象都是朝生夕灭的**
2. **强分代假说：熬过越多次垃圾收集的对象就越难以消亡**

根据这两个假说，收集器应当将JAVA堆划分为两个区域，然后将所有对象依据其年龄(熬过垃圾收集的次数)分配到不同的区域中存储

HotSpot将java堆分为两个区域，即**新生代(Young Generation)**和**老年代(Old Generation)**，正是因为分代划分，才有了Minor GC、Major GC、Full GC等划分，进而也发展出了**标记-清除、标记-复制、标记-整理**等算法

* + 1. **对于新生代的GC一般被称为Minor GC**
    2. **对于老年代的GC一般被称为Major GC**
    3. **对整个新生代和部分老年代的GC被称为Mixed GC**
    4. **对整个堆进行GC被称为Full GC**

**Full GC通常是和Major GC等价的**，因为许多 Major GC 是由 Minor GC 触发的，所以很多情况下将这两种 GC 分离是不太可能的

因为分代收集，我们也产生了一些问题，比如：假设现在需要对新生代进行一次Minor GC，似乎我们只需要获取新生代的GC Roots即可，但实际情况并不是这样，因为新生代对象完全可能被老年代对象所引用，这就是跨代引用为题。为了少量的跨代引用，我们不得不在固定GC Roots外，在额外加入所有老年代对象以保证可达性分析的正确性，这无疑为Minor GC带来了很大的性能负担

接下来我们将介绍一种记忆集的机制，来帮助我们解决这个问题。

其大致原理是：在新生代中建立一个全局的数据结构，即记忆集(Remembered Set)，这个结构将老年代划分为若干小块，标识出老年代在哪一块存在存在跨代引用，此后只需要将存在跨代引用的小块老年代中的对象加入到GC Roots进行扫描。虽然该方法需要在改变对象引用关系时维护记忆集(Remembered Set)数据的正确性，会增加一些运行时开销，但比起扫描整个老年代还是划算的

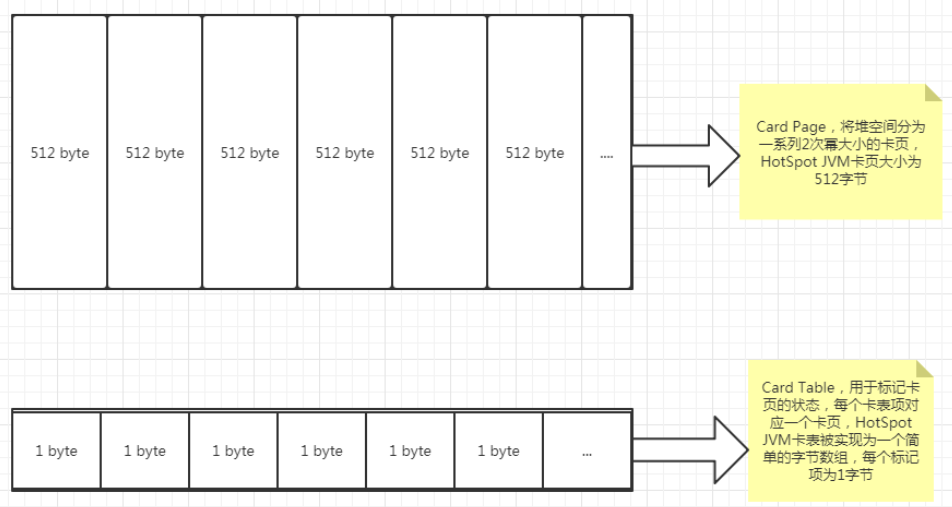
* + - * 1. 记忆集和卡表

刚才我们说到了，在MinorGC中的跨代引用需要通过记忆集来解决，其实不止MinorGC，所有涉及区域收集行为的垃圾处理器，都存在跨区域引用的问题，典型的如G1，为了方便后续理解G1，这里介绍一下记忆集(Remembered Set)

主流的记忆集实现方式是通过卡表(Card Table)实现的，卡表代表的是一块内存区域，它意味着记录存在跨区域引用的数据的最小粒度是一小块内存区域，HotSpot是通过一个字节数组来表示卡表(Card Table)：

**CARD\_TABLE[this address >> 9] = 0;**

**CARD\_TABLE**数组的每个元素都映射了一块特定的内存区域，这个内存区域被称为卡页(Card Page)。>> 9相当于除以512，由此可以看出HotSpot中一个卡页的大小为512字节。



HotSpot将整个堆划分为一个个大小为 512 字节的卡页(Card Page)，并且维护一个卡表(Card Table)数组，用来存储每张卡页(Card Page)的一个标识位，这个标识位代表对应的卡页(Card Page)是否存有跨区域指针。如果存在，那么我们就认为这张卡是脏的，在进行 gc 的时候，我们便可以不用扫描整个堆，而是在卡表(Card Table)中寻找脏卡，并将脏卡对应内存区域中的对象加入到 GC Roots 里。

之所以使用字节数组实现卡表而不是bitmap，是因为现代计算机最小都是按照字节寻址的，如果需要操作bit，就必须多消耗几条shift+mask指令，导致操作bitmap的速度比直接操作字节数组要慢，最终选择字节数组实现，是一种以空间换时间的策略

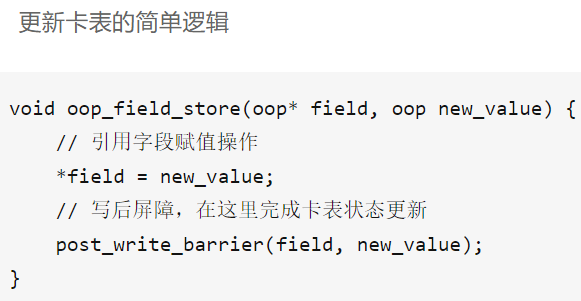
* + - * 1. 写屏障Write Barrier

那么如何维护Card Table的状态呢？简而言之，就是怎么把卡表(Card Table)的某个元素变脏呢？

首先我们需要明确，在有其他区域中的对象引用了本区域对象，即发生跨代引用时，本区域对象对应的卡页(Card Page)就应该变脏

HotSpot采用写屏障(Write Barrier)来维护卡表中元素的状态位，写屏障可以看做是虚拟机层面做的，对于“引用字段赋值”这一操作的AOP切面，类似spring aop，在引用对象赋值时会产生一个环形(Around)通知，在赋值前的写屏障被称为写前屏障(Pre- Write Barrier)，在赋值后的则叫写后屏障(Post- Write Barrier)

下面就是一段更新卡表(Card Table)状态的简化逻辑：



每次对引用进行赋值，写屏障都会带来额外的开销，不过这个开销相对于MinorGC扫描整个老年代来说还是低的。

除了写屏障的开销外，卡表还要面临“伪共享”的问题。我们都知道，现代cpu操作缓存的单位是“缓存行”(cacheline)，如果CPU希望读取一个被缓存的变量x，那么CPU需要读取变量x所在的整个缓存行cacheline。

假设cpu缓存行大小为64字节，而卡表是比特数组，单个元素的大小是1字节，那么64个卡表元素将共享同一个缓存行，这64个卡表元素对应的卡页(Card Page)大小为64\*512byte=32KB。那么只要多个线程更新的对象正好处在这32KB的内存空间中，那么就会出现单个缓存行cacheline的并发读写问题，触发缓存一致性协议(写回、无效化或者同步)，导致性能降低

我们一般解决为共享问题是通过向共享变量的前后插入padding留白，来将其他变量挤出缓存行，但这样会造成大量的工作内存浪费，如果我们对卡表使用此方式，相当于卡表数据的每一个元素都单独占有一个缓存行，这将会使工作内存浪费到一个不可接受的地步

最简单的解决方案是，采用有条件的写屏障，即先检查卡表标记，如果卡表元素未被标记过，才将其标记为脏，如下：

if (CARD\_TABLE [this address >> 9] != 0)

CARD\_TABLE [this address >> 9] = 0;

这种方式可以减少伪共享 出现的概率，但是由于多了一个if判断，最终是否能够提高性能，需要在具体的应用中测试，jvm默认不打开此if，可以通过参数-XX:+UseCondCardMark打开

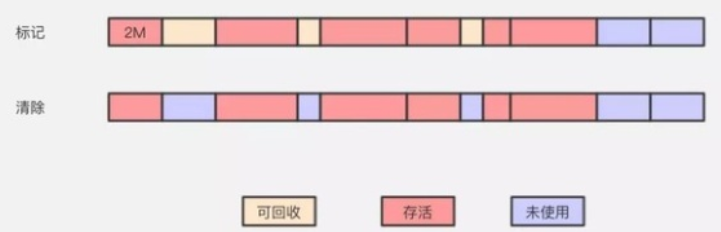
* + - * 1. 标记-清除算法

标记-清除算法是一种最基础的垃圾回收算法，它分为两个步骤：

1. **把内存区域中的这些对象进行标记，哪些属于可回收标记出来**
2. **在标记完成后统一回收可回收的对象**

这个算法相当简单，但它有两个很大的缺点：

1. **执行效率不稳定。**假设堆中有大量对象需要回收，此时就必须要进行大量的标记和清除操作，即标记和清除两个过程的效率随对象数量的增长而降低
2. **内存空间碎片化。**简单地进行标记、清除，会产生大量不连续的内存空间，一旦需要分配较大对象而无法找到足够的连续内存时，就需要提前触发下一次的垃圾收集操作



* + - * 1. 标记-复制算法

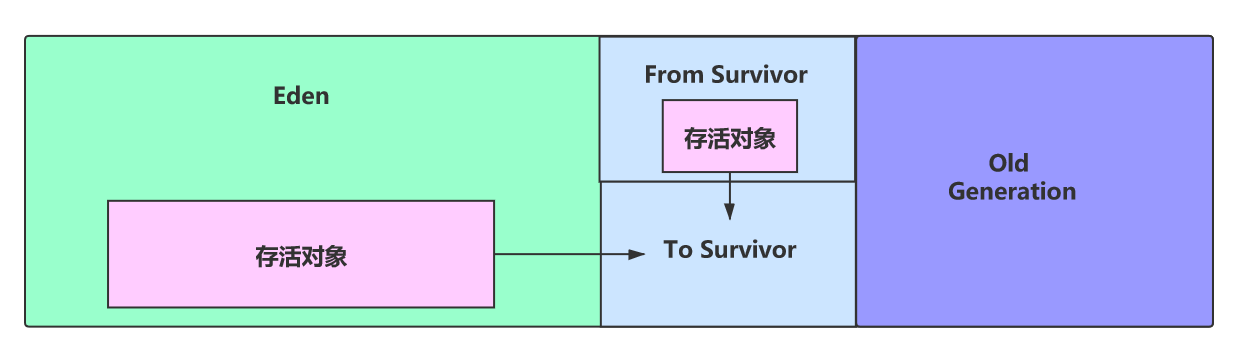
为了解决**标记-清除算法**面对大量可回收对象时执行效率大幅降低的问题，有人提出了标记-复制算法。

标记-复制算法将新生代划分为大小相等的两块，每次只使用其中的一块，当某一块内存用完了，就将还活着的对象复制到另一块内存上面，然后再一次性将已使用的内存块全部清理掉。

这种算法解决了**标记-清除算法**的两个缺点，面对绝大多数对象是需要回收的情况，**标记-复制算法**仅仅需要付出复制少量存活对象的代价就能完成垃圾回收，并且解决了内存碎片化的问题。但是**标记-复制算法**最大的问题在于它将内存大小缩小了一半，空间浪费很大



为了优化**标记-复制算法**，HotSpot将新生代分为一个较大的Eden区和两个较小的Survivor区，每次内存分配仅仅使用Eden区和一个Survivor区。当发生垃圾回收时，HotSpot将Eden区和已使用From Survivor区中还存活的对象一次性复制到另一个空置的To Survivor区，然后直接清理掉Eden区和已使用的From Survivor区，最后“From”和“To”会交换他们的角色。HotSpot默认的Eden区和Survivor区的大小比例是8:1，这意味着每次新生代可分配内存为整个新生代的90%，仅仅浪费了10%的空间，大大提高了内存空间利用率



当然，并不能确保每次垃圾回收时空置的Survivor区能容纳下Eden区和已使用Survivor区的存活对象，这就需要老年代对其进行分配担保，如果容纳不下，就需要暂时进入老年代中

* + - * 1. 标记-整理算法

**标记-复制算法**在对象存活率较高时需要进行大量复制，会大幅降低效率，因此在老年代中很难使用这种算法。为了应对老年代的垃圾回收，有人提出了**标记-整理算法**

**标记-整理算法**的标记过程仍然和**标记-清除算法**一致，但后续步骤不是直接对可回收对象进行清理，而是让所有存活的对象都向一端移动，然后直接清理掉端边界以外的内存



标记整理算法一方面在标记-清除算法上做了升级，解决了内存碎片的问题，也规避了复制算法只能利用一半内存区域的弊端。看起来很美好，但从上图可以看到，它对内存变动更频繁，需要整理所有存活对象的引用地址，在效率上比复制算法要差很多

老年代占据着2/3的堆内存空间，只有在 Major GC 的时候才会进行清理，每次 GC 都会触发“Stop-The-World”，即整个应用停止运行，其他所有除了垃圾收集收集器线程之外的线程都被挂起。这是难以接受的

* + - * 1. 总结

把 Java 堆分为**新生代**和**老年代**，这样就可以根据各个年代的特点采用最适当的收集算法。在**新生代**中，每次垃圾收集时都发现有大批对象死去，只有少量存活，那就选用**标记-复制算法**，只需要付出少量存活对象的复制成本就可以完成收集。而老年代中因为对象存活率高、没有额外空间对它进行分配担保，就必须使用**标记-清除算法**或者**标记-整理算法**来进行回收



Survivor 区相当于是 Eden 区和 Old 区的一个缓冲

1、为啥需要Survivor 区？

直接 Eden 到 Old 不好了吗，为啥要这么复杂。想想如果没有 Survivor 区，Eden 区每进行一次 Minor GC，存活的对象就会被送到老年代，老年代很快就会被填满。而有很多对象虽然一次 Minor GC 没有消灭，但其实也并不会蹦跶多久，或许第二次，第三次就需要被清除。这时候移入老年区，很明显不是一个明智的决定。

所以，Survivor 的存在意义就是减少被送到老年代的对象，进而减少 Major GC 的发生。Survivor 的预筛选保证，只有经历16次 Minor GC 还能在新生代中存活的对象，才会被送到老年代。

2、为啥需要俩Survivor 区？

设置两个 Survivor 区最大的好处就是解决内存碎片化。

我们先假设一下，Survivor 如果只有一个区域会怎样。Minor GC 执行后，Eden 区被清空了，存活的对象放到了 Survivor 区，而之前 Survivor 区中的对象，可能也有一些是需要被清除的。问题来了，这时候我们怎么清除它们？在这种场景下，我们只能标记清除，而我们知道标记清除最大的问题就是内存碎片，在新生代这种经常会消亡的区域，采用标记清除必然会让内存产生严重的碎片化。因为 Survivor 有2个区域，所以每次 Minor GC，会将之前 Eden 区和 From 区中的存活对象复制到 To 区域。第二次 Minor GC 时，From 与 To 职责对换，这时候会将 Eden 区和 To 区中的存活对象再复制到 From 区域，以此反复。

这种机制最大的好处就是，整个过程中，永远有一个 Survivor space 是空的，另一个非空的 Survivor space 是无碎片的。那么，Survivor 为什么不分更多块呢？比方说分成三个、四个、五个?显然，如果 Survivor 区再细分下去，每一块的空间就会比较小，容易导致 Survivor 区满，两块 Survivor 区可能是经过权衡之后的最佳方案。

* + 1. 常用垃圾收集器
       1. jvm的client模式和Server模式

将JDK安装好后，输入java –version，我们不仅可以看到版本号，还能看到：



这个Server VM代表JVM是以Server模式运行的，相对的，JVM还存在一种client模式，这两个模式最主要的差别在于：-Server模式启动时，速度较慢，但是一旦运行起来后，性能将会有很大的提升

**通常用于做服务器的时候我们用服务端模式，如果你的电脑只是运行一下java程序，就客户端模式就可以了。**

如何修改jvm运行模式呢？

**64位系统默认在 JAVA\_HOME/jre/lib/amd64/jvm.cfg**

**32在目录JAVA\_HOME/jre/lib/i386/jvm.cfg**

修改:

-server KNOWN  
-client IGNORE

如果希望以client模式启动，将两个**单词**对调一下即可

* + - 1. Serial收集器

Serial收集器是HotSpot中历史最悠久的垃圾回收器，它在JDK1.3.1之前甚至是垃圾收集器的唯一选择

Serial收集器是针对**新生代(Young Generation)**的单线程垃圾回收器，它采用**标记-复制算法**。Serial收集器在对新生代进行垃圾回收时，会导致**”Stop The World”**，即挂起所有工作线程，整个应用暂停服务，直到垃圾收集结束。试想一下，如果你的电脑每运行一小时，就会暂停响应五分钟，你是什么样的心情？因此**”Stop The World”**应当是尽力避免的

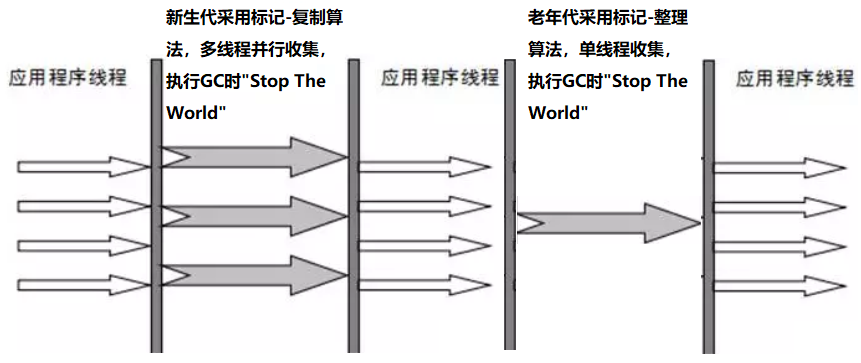
尽管Serial收集器会导致**”Stop The World”**，但是它简单而高效，对于内存较小的、单核心或核心数较少的运行环境，Serial收集器由于没有线程交互的开销，可以获得最高的单线程收集效率。正是因此，**HotSpot仍选择Serial收集器作为Client模式下的默认新生代垃圾回收器**。

* + - 1. Serial Old收集器

Serial Old收集器是Serial收集器的**老年代(Old Generation)**版本，它采用**标记-整理算法**，同样的，它也是单线程运行。它具有和Serial类似的特性，因此，这个收集器也**主要是运行在 Client 模式下HotSpot虚拟机默认的老年代(Old Generation)垃圾收集器。**

* + - 1. ParNew收集器

ParNew收集器实际上就是Serial收集器的多线程版本，它的很多特性都和Serial相同，甚至ParNew收集器复用了很多Serial收集器的代码。**ParNew垃圾收集器是激活CMS收集器后的默认新生代垃圾收集器**



在单核心处理器环境中，ParNew由于存在线程交互开销，其效果不如Serial收集器，但随着处理器技术的更新换代，现在的处理器通常有更多的核心数，在这种情况下，ParNew收集器的效率要远高于Serial收集器

在JDK5推出老年代垃圾收集器CMS后，ParNew通常被用于与CMS配合使用。直至G1出现，G1收集器是面向全堆的垃圾回收器，不需要其他新生代收集器配合工作，自JDK9起，官方就不再推荐ParNew+CMS的组合了，转而推荐使用G1进行替代

* + - 1. Parallel Scavenge收集器

Parallel Scavenge收集器是一款面向**新生代(Young Generation)**的多线程垃圾回收器，它采用**标记-复制算法**。那么它和ParNew有什么区别呢？

Parallel Scavenge收集器可以进行垃圾回收策略的自适应调节，它关注的重点在于控制应用的吞吐量。Parallel Scavenge收集器提供了两个参数用于精确控制吞吐量

**1.-XX:MaxGCPauseMillis：**控制最大垃圾收集停顿时间，即”Stop The World”停顿时间，应该设为一个大于0的毫秒数。收集器将尽力保证垃圾回收花费不超过设置时间，但并非设置得越小越好，这是以牺牲整体吞吐量和新生代空间为代价的。设置得越小，新生代越小，垃圾收集越频繁，吞吐量越低。

**2.-XX:GCTimeRatio：**该参数通常被用于控制吞吐量大小，值在0-100之间，默认值为99。

Parallel Scavenge收集器是JDK7和JDK8中的默认新生代垃圾收集器，从JDK9起被G1取代，G1成为默认收集器

* + - 1. Parallel Old收集器

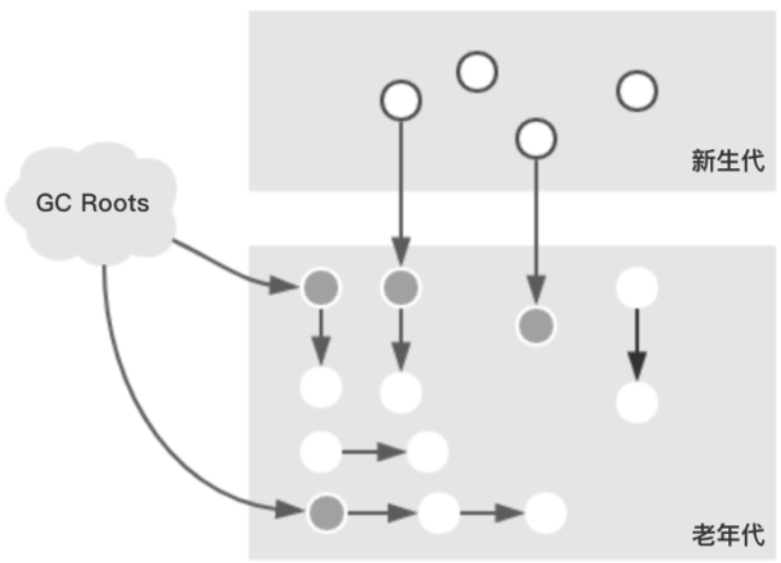
顾名思义，Parallel Old收集器是Parallerl Scavenge收集器的老年代版本，基于标记-整理算法，支持多线程并行收集，该收集器是在JDK6以后提供的，在此之前，Parallerl Scavenge收集器一直处在一个很尴尬的地位，如果新生代选择了Parallerl Scavenge收集器，那么老年代除了Serial Old以外别无选择(CMS无法与Parallerl Scavenge收集器配合)，由于Serial Old单线程的拖累，这个组合在多处理器环境下的吞吐量往往不太理想

* + - 1. CMS收集器

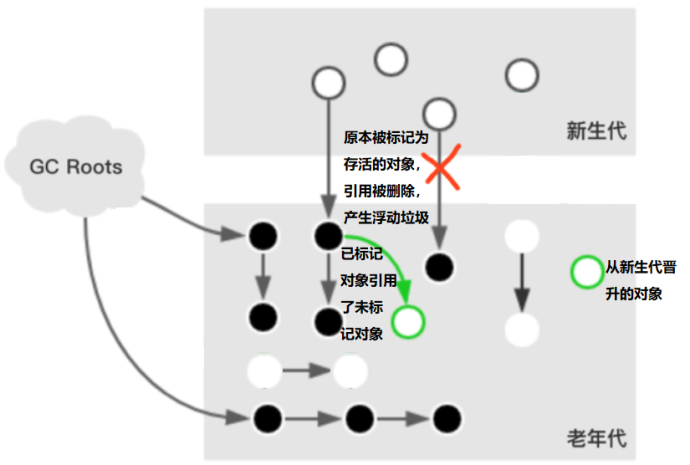
JDK5发布时，HotSpot推出了一款具有划时代意义的垃圾回收器—CMS，它第一次实现了，让垃圾收集线程和用户线程同时工作。**CMS(Concurrent Mark Sweep)收集器是一种基于多线程的“标记-清除”算法，以获取最短回收停顿时间为目标的老年代垃圾收集器。**它可以分为四个步骤：

1. **初始标记(CMS initial mark)：**这个阶段是标记从Gc Roots中直接可达的老年代对象(根节点枚举)以及新生代引用的老年代对象(跨代引用，因为卡表只记录老年代引用新生代的关系，所以这里没有用卡表，而是会扫描整个新生代，并把新生代的有效存活对象加入GC Roots)。就是下图中灰色的点。**这个过程会引起Stop The World**，好在它的初始标记速度很快(参见[根节点枚举](#根节点枚举))。

可以确定的是初始标记会扫描新生代，cms的源码参见<https://hub.nuaa.cf/YouDiSN/notes/blob/master/jvm/cms源码分析（补充内容）.md>



1. **并发标记(CMS concurrent mark)：**根据初始标记的对象，开始遍历对象图，标记所有可达的对象，这个过程耗时较长但可以和用户线程并发执行。由于用户线程并不会停止，因此会出现**对象从新生代晋升到老年代**、**老年代对象引用被改变**等情况



1. **重新标记(CMS remark)：**重新标记是为了修正并发标记阶段，因为用户线程并行运行导致的变动，**这个过程会引起Stop The World**，这个阶段的耗时比初始标记长一些，但远短于并发标记阶段(参见[增量更新](#增量更新))
2. **并发清除(CMS concurrent sweep)：**清理删除掉标记阶段判定的已死亡对象，由于标记-清除算法不需要移动存活对象，因此这个阶段是可以与用户线程并行执行的

由于会导致STW的初始标记和重新标记耗时较短，所以总体上来说，CMS收集线程是可以和用户线程并行执行的

**CMS是老年代垃圾收集器，一般情况下会有ParNew来配合执行(默认情况下也是ParNew)，在JDK9中被标记为不推荐使用，并在未来的版本中可能会被放弃**

缺点：

1. **空间碎片问题：**使用标记-清除算法，因而会导致大量内存空间碎片，当碎片空间太多以至于无法分配大对象时，会提前触发Full GC
2. **浮动垃圾问题：**由于并发标记阶段垃圾回收线程和用户线程并行执行，用户线程在运行的过程中会自然产生垃圾，这部分垃圾出现在标记结束后，CMS线程无法在当次回收中处理他们，只好等下一次垃圾回收时再清理，这一部分垃圾被称为浮动垃圾
3. **对处理器资源很敏感：**虽然CMS只会造成很短的停顿，但由于它占用了一部分CPU资源，它仍会降低应用的响应速度

前面我们介绍过关于跨代引用的处理，CMS是采用[卡表](#记忆集和卡表)来解决跨代引用的，但是在CMS中，卡表只记录老年代对象是否引用了新生代对象。也就是说，当新生代发生MinorGC时，不再需要将整个老年代都加入GC Roots扫描，而是先遍历老年代记录的卡表，将脏卡加入GC Roots

可是CMS不是老年代垃圾回收器吗？为什么我们记录卡表只记录老年代的卡表呢？新生代也会引用老年代对象的啊？

**我们都知道老年代区域比新生代大，在ParNew进行MinorGC时，如果扫描整个老年代，这是件很消耗性能的事情。而老年代卡表则可以避免老年代的全扫描**

**CMS的卡表是Point out模式的(G1是Point in)，也就是说，cms的卡表记录了”我引用了谁”，因此老年代引用新生代的关系是被记录在老年代卡表中的**

**如果CMS要进行 Major GC，即要清理老年代中的对象，那么当然也可以存在该对象被新生代对象所引用的情况，然后就需要扫描整个新生代，那么是否需要新生代卡表呢？**

**HotSpot 实际上并没有使用新生代卡表，因为新生代的引用变化率非常高，GC 比较频繁，而年轻代采用的大多是标记复制算法，经过一次MinorGC以后，会有大量新生代对象消亡，并且存活下来的原来脏卡中的跨代对象，也很容易被复制到另一个卡页中，导致新生代中大量脏卡变为非脏卡，也就是每次MinorGC以后，需要大量更新新生代中存储的卡表(新生代引用老年代)，而老年代由于采用标记清除算法，不需要移动老年代对象，并且老年代对象存活时间一般比较长，因此不会有这方面的担心**

在重新标记阶段，将整个新生代都加入GC Roots扫描，是CMS的增量更新算法 比 G1的原始快照SATB算法要慢的主要原因，因为SATB相当于始终对GC开始的那一刻的内存快照进行可达性分析，因而在重新标记阶段不需要考虑新生代的跨代引用(因为在初始标记阶段已经考虑过跨代引用了，SATB中初始标记和重新标记应当认为是在相同的内存快照中执行的)

* + - 1. G1收集器

Garbage first 垃圾收集器是目前垃圾收集器理论发展的最前沿成果，它从JDK6进入实验状态，在JDK7 update4中被Orcale认定为成熟。它的目标是替换掉JDK5中发布的CMS收集器，并在JDK9中替代Parallel Scavenge+Parallel Old组合成为Server模式下的默认垃圾处理器

**G1从整体上来看是 标记-整理 算法，但从局部（两个Region之间）是复制算法**

在G1出现之前，包括CMS在内的众多垃圾收集器，其垃圾收集的目标范围要么是整个新生代(Minor GC)，要么是整个老年代(Major GC)，要么是整个堆(Full GC)。而G1打破了这个规则，它可以进行局部收集，其衡量标准已经不再是哪个分代，而是堆中哪块内存的垃圾最多，这就是G1的Mixed GC模式

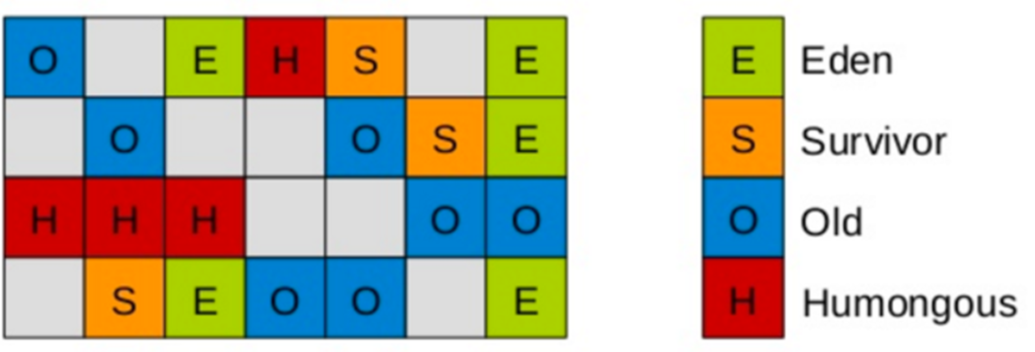
* + - * 1. 内存模型

和传统的垃圾收集器不同，G1不再坚持固定大小和固定数量的分代区域划分，而是把整个java堆划分为多个不同的Regions区域，默认情况下是2048个Regions。每个Regions的大小可以通过参数-XX:G1HeapRegionSize=n修改，取值范围为1MB~32MB，且必须为2的n次方，也就是说，取值仅可为1MB、2MB、4MB、8MB、16MB、32MB。

这些Regions区域可分为四类：

1. **Eden Regions：**新生代，G1收集器使用Eden Region来满足大多数内存分配需求
2. **Survivor Regions：**新生代，用于放置仍存活但不足以晋升老年代的对象
3. **Old Regions：**老年代，存放大龄对象
4. **Humongous Regions**：大对象区域，G1把该区域也当做老年代处理。当一个对象大于Region大小的50%，称为巨型对象，它就会独占一个或多个Region

G1并不要求相同类型的Regions在物理地址上相邻，但他们在逻辑上是连续的，如下图：



每个Regions的身份并不是固定不变的，每一个Regions都可以根据需要去扮演不同的角色。可能一个Eden Regions在经过一次垃圾回收后被划分为待使用区域，下一次它也可能变为Old Regions或者其他Regions。由此可以看出，G1会根据预测时间动态地改变各分代的大小

G1将以这些Region作为最小回收单元，即每次回收的内存大小都是Regions的整数倍，并且将大对象单独处理，这大大降低了Full GC发生的概率。当然，即使有了Humongous Regions，如果对象过大，找不到足够多的连续的Humongous Regions区，仍然会发生Full GC。

G1 并不会进行全区域的垃圾收集，它会跟踪每个Region被回收的价值，并维护一个优先级队列，每次优先处理回收价值最大的Region。

这种划分多个Region，并维护每个Region的回收价值优先队列的方式，尽可能高地提升了G1在有限的时间内回收垃圾的效率

* + - * 1. GC模式

**G1提供了两种GC模式，Young GC和Mixed GC，两种都是完全Stop The World的**

young GC

**选定所有年轻代里的Region。**通过控制年轻代的region个数，即年轻代内存大小，来控制young GC的时间开销。

Mixed GC

**选定所有年轻代里的Region，外加根据”全局并发标记”(global concurrent marking)统计得出收集收益高的若干老年代Region**。在用户指定的开销目标范围内尽可能选择收益高的老年代Region进行垃圾回收。

Mixed GC不是full GC，它只能回收部分老年代的Region，如果mixed GC实在无法跟上程序分配内存的速度，导致老年代填满无法继续进行Mixed GC，就会使用串行的full GC来收集整个GC heap(在java10中，改为并行收集)。

Mixed GC是由young GC触发的，其**”全局并发标记”(global concurrent marking)**的第一步—初始标记阶段是跟随young GC同步完成的，这样可以减少Mixed GC中造成的STW

* + - * 1. G1实现过程中需要解决的问题

G1看似非常完美，但是实现起来仍有如下问题需要解决

跨Regions引用问题(记忆集Rset)

在G1中，跨区域引用是非常常见的，我们通过记忆集(Rset)可以解决这一问题。**每一个Region将维护自己的记忆集(Rset)**，卡表会记录下在自己的Region区域内，哪些卡页存在其他Region指向自己Region区域的指针，这样在gc过程中需要枚举GC Roots时，只需要扫描**记忆集(Rset)**就可以得到跨区域引用，避免了整堆扫描。但是这样导致G1要占用更多的内存冗余，根据经验，G1大概需要耗费相当于堆大小的10%~20%用于维持收集器的工作

G1的**记忆集(Rset)**实现比传统的卡表实现更加复杂，有了Rset，部分区域回收成为可能

需要记录哪些跨代引用

首先我们需要分析哪种引用需要被G1记录在Rset中，只有本次GC中，不被回收的区域 引用了 被回收区域的引用，才需要被记录

1. **分区内部的引用**

**无需记录。**无论是新生代还是老年代的分区内部的引用，都不需要记录引用关系。因为是针对一个分区进行的垃圾回收，要么这个分区被回收，要么不被回收。

1. **新生代引用新生代**

**无需记录。**G1的两种回收模式（YGC/MIXED GC）都会全量处理新生代分区，所以整个新生代都会被回收。因此无需记录这种引用关系。

1. **新生代引用老年代**

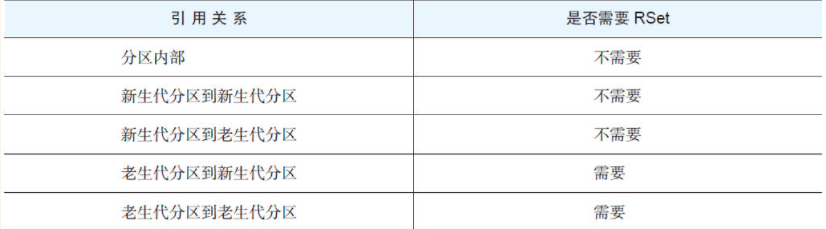
**无需记录。**G1的YGC只负责回收新生代，无需这个引用关系。混合GC时，G1会采用所有的survivor region中的对象作为GC roots，那么在遍历新生代分区时就能找到老年代分区了，无需这个引用关系。

1. **老年代引用新生代**

需要记录。YGC在回收新生代时，如果新生代的对象被老年代引用，那么需要标记为存活对象。即此时的根对象有两种，一个是栈空间/全局变量的引用，一个是老年代到新生代的引用。

1. **老年代引用老年代**

需要记录。混合GC时，只会回收部分老年代，被回收的老年代需要正确的标记哪些对象存活。



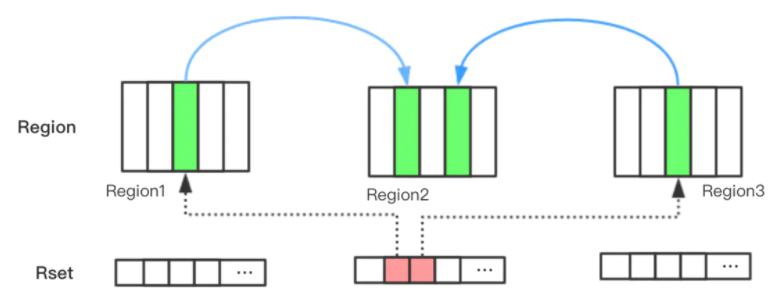
可以看出，cms和g1都需要记录老年代引用新生代，都不需要记录新生代引用老年代，但是g1还需要额外记录老年代引用老年代的跨代引用

G1中的Rset

G1的记忆集(Rset)采用的是Point-in模式(而CMS是采用Point-out模式的，即”我引用了谁”) ，即每个Region的记忆集RSet记录了其他Region中的对象引用本Region中对象的关系，假如有：

objectA.field = objectB

即A引用了B，或者说A中存在指向B的指针，那么会在对象B的Region的Rset中记录对象A的地址，如下图



Region1和Region3中的对象引用Region2中的对象，那么会在Region2的Rset中记录下这两个Region中的对象

采用Point-in模式后，对于垃圾回收的跨代引用会更好处理，试想一下如果老年代Region中的对象A引用了新生代Region中的对象B，那么在young GC时，就需要考虑到新生代对象B被老年代对象A跨代引用了，即新生代对象B所在的Region中需要记录跨代引用信息，这样一来就不需要再去扫描老年代Region获取跨代引用信息了

由于对当前region对象的引用可能很多，也可能很少，G1会根据数量的变化来改变数据结构以达到节省空间的目的，主要有三种数据结构

1. hash表

key是别的Regin的起始地址，在上例中，粉红色Rset的键值对中的key就是Region1和Region3的起始地址

value是一个集合，里面存储的元素是卡表的索引号。在上例中，就是Region1和Region3中绿色部分对应的卡表索引，即对象所在卡页的索引号

1. 细粒度表

略

1. 粗粒度位图

略

G1通过什么机制获知引用关系改变

明确需要记录哪些跨Regions引用后，G1又是在什么时机进行的记忆集(Rset)维护呢？

**G1采用写后屏障来维护记忆集**，在写变更时，通过插入一条额外的代码把引用关系放入到脏表队列(Dirty Card Queue)，即DCQ队列中，随后refine线程池中的线程取出DCQ队列的引用关系，更新RSet。比如，每一次将一个老年代对象的引用修改为指向新生代对象时，都会被写屏障捕获，并且记录下来

用户线程与收集器并发执行问题

用户线程和收集器线程并发执行可能会造成两个问题：

1. **修改引用：**在收集器线程并发标记过程中，修改了对象的引用，导致出现浮动垃圾或者对象消失现象
2. **新建对象：**在收集器线程并发标记过程中，创建了新的对象

修改对象引用关系(SATB算法)

这一点在三色标记的缺陷一节有说过，如果在一个不能保证一致性的快照中进行并发标记，即垃圾收集器和用户线程并行工作时，如果可能会出现错误标记，产生浮动垃圾和对象消失的情况，对于浮动垃圾，我们可以暂时不管，等待下次回收，但是对于对象消失，我们必须要采取措施避免出现

CMS采用增量更新来解决对象消失问题，而对于G1，则采用**原始快照(Snapshot-At-The-Beginning，简称SATB)**来解决

原始快照破坏第二个条件，即赋值器删除了全部从灰色对象到该白色对象的直接或间接引用。当灰色对象要删除指向白色对象的引用关系时，就将这个要删除的引用关系记录下来，并发标记结束后，再将以该灰色对象为根，重新扫描一次。换言之，就是无论引用关系删除与否，都会按照刚开始扫描那一刻的对象图快照来进行搜索

SATB采用最保守的做法，把变更前的引用对象记录下来，当作是存活对象，让其活过这一周期。因此 G1 的 SATB 会产生更多的浮动垃圾。但是换来的好处是不需要像 CMS 那样 remark，再走一遍 root trace 这种相当耗时的流程。

创建新的对象

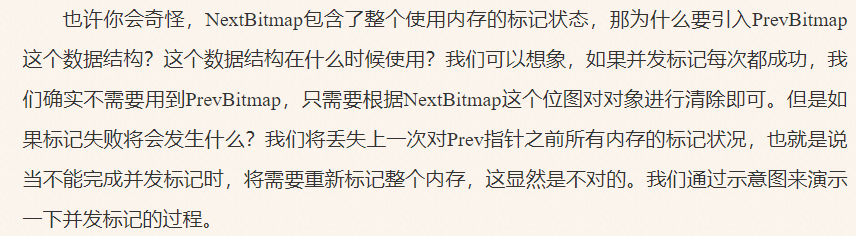
在全局并发标记(global concurrent marking)过程中，如果用户线程执行的时候不仅修改了对象引用关系，还新分配了新对象，G1 是如何找到并处理这些新对象的呢？

之前我们就介绍过，G1从整体上来看是 标记-整理 算法，但从局部（两个Region之间）是复制算法，无论是哪种算法，G1都会大大减少内存碎片的出现，因而新增对象是可以连续分配的，这样我们就可以通过几个指针来控制新对象的创建的内存区域位置

首先我们来介绍Region中的**2个bitmap**和**2个TAMS个指针(Top at Mark Start指针)** ，以及另外**3个内存位置标记指针**

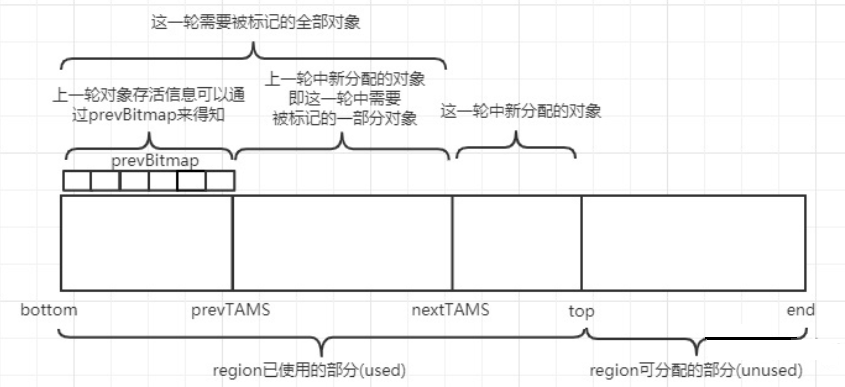
1. **Previous Bitmap：**记录的是第n-1轮(上一轮)Concurrent Marking 后的对象标记状态，由于第n-1轮concurrent marking已经完成，这个bitmap的信息可以直接使用。
2. **Next BitMap：**记录的是第n轮(当前这一轮)Concurrent Marking 的结果，尚未完成，不能使用。

看完这里你可能会有一个疑问，为什么我们需要PrevBitmap去记录上一轮并发标记的结果？这个答案可以在彭成寒的《Java核心技术系列·新一代垃圾回收器ZGC设计与实现》一书中关于G1介绍的一节中找到答案：



1. **Previous TAMS指针：**上一次全局并发标记(global concurrent marking)开始前已使用的内存分界点的地址，即上一次全局标记开始前的top位置
2. **Next TAMS指针：**全局并发标记(global concurrent marking)开始前已使用的内存分界点的地址，即本次全局标记开始前的top位置
3. **Bottom指针**：记录Region的起始地址
4. **Top指针**：指向当前内存成功分配后的地址
5. **End指针**：指向Region的结束地址

对应下图：



**[bottom, top)**是当前该region已用（used）的部分

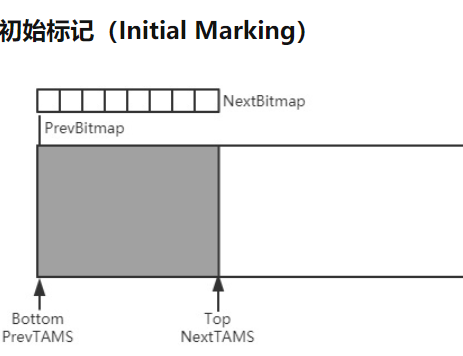
**[top, end)**是尚未使用的可分配空间（unused）

**[prevTAMS, nextTAMS)** 是上一轮标记到这一轮标记周期内新创建的对象

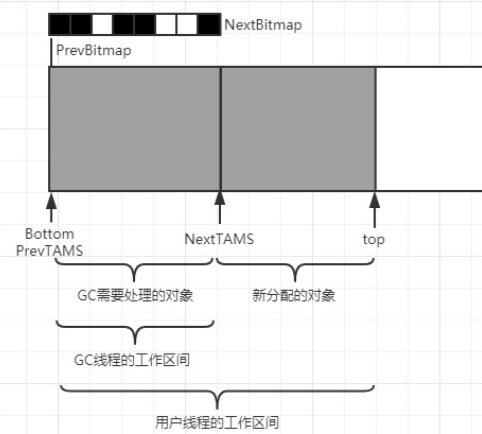
**[Bottom, PrevTAMS)**是上一轮全局并发标记中GC的工作区域，这部分内存在上一轮全局并发标记的标记结果将体现在prevBitmap中

**[Bottom, nextTAMS)** 本轮全局并发标记需要被标记的内存区域，这部分内存的并发标记结果将被放在nextBitmap中

**[nextTAMS, top)**是全局并发标记过程中新分配的对象地址，新分配的对象默认是存活的，并发标记开始前(即初始标记阶段)，G1将会让nextTAMS 指向 Top，如下图：



当进入并发标记阶段时，如果需要新分配对象时，就移动Top指针，分配内存区域，新分配的对象默认是存活的，如下图：



这样一来，GC线程的工作区间，如发生引用关系修改，则通过SATB算法去记录，并在最终标记阶段暂停用户线程(STW)后去处理。而新分配的对象，则放在NextTAMS和Top指针之间，仅供用户线程使用，避免了GC线程和用户线程并发操作这部分内存中的对象

如何建立可靠的停顿预测模型

略

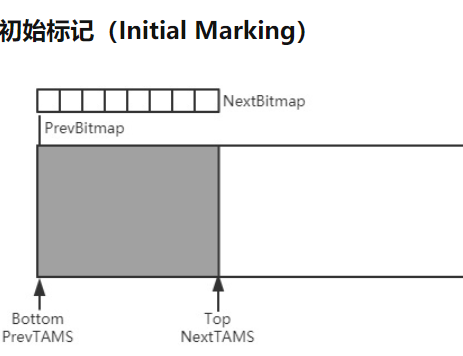
* + - * 1. 全局并发标记(global concurrent marking)的

全局并发标记发生在Mixed GC中，用于统计得出收集收益高的若干老年代Region，并通过并发可达性分析得出存活对象。帮助G1在用户指定的开销目标范围内尽可能选择收益高的老年代Region进行垃圾回收。

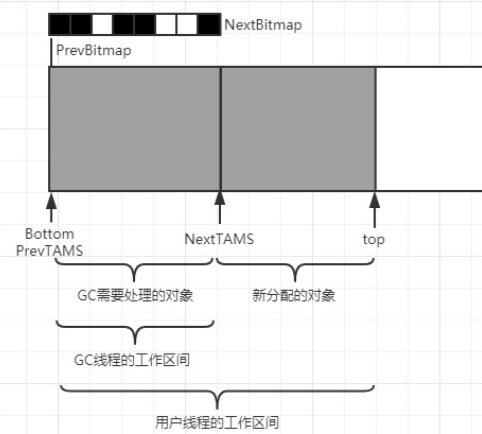
执行过程

和CMS类似，G1的垃圾回收流程也分为四步：

1. **初始标记(Initial Marking)：**标记GC Root能直接关联到的对象，同时修改TAMS指针的值(移动 nextTAMS 使其指向 Top，Top 实际上就是一个 Region 未分配区域和已分配区域的分界点，而PrevTAMS指向的是上一轮全局并发标记开始前的top位置，接下来G1将在Bottom和nextTAMS之间工作，并发标记过程中新分配的对象将放在nextTAMS和top指针之间)。**此阶段会发生STW**，但可以在Young GC造成的STW中同步执行(Mixed GC由Young GC触发)，因此不会有额外的停顿，下图展示了初始标记结束后的Region内存状态



1. **并发标记(Concurrent Marking):** 从GC Root开始进行并发的可达性分析，找出要回收的对象，这一阶段耗时较长，但可以和用户线程并发执行，下图展示了并发标记结束后的Region状态(NextBitMap的黑色部分表示还存活的内存区域在bitmap中的映射)



1. **最终标记(Final Marking):** 暂停用户线程，处理SATB记录下的少量有引用变动的对象，**此阶段会发生STW**
2. **筛选回收(Live Data Counting and Evacuation):**负责更新Regions的统计及记录，对各Regions的回收价值和成本进行排序，根据用户期望的停顿时间制定回收计划，可以自由选择任意的Region组成回收集，并将决定回收的Region中的存活对象复制到空的Region，再清理掉回收集的全部空间。这一阶段涉及到存活对象的移动，因此**此阶段会发生STW**
3. 如果G1判断此Region需要被回收，那么就需要拷贝**NextBitmap中黑色区域对应的内存空间中的对象** 加上 **NextTAMS指针和top之间的对象(新分配对象)** 到另一个空的Region中
4. 如果此Region可用空间较多，回收价值较低，

**初始标记阶段initial mark是共用了Young GC的暂停，这是因为他们可以复用root scan操作，所以可以说global concurrent marking是伴随Young GC而发生的。**

全局并发标记中的TAMS指针

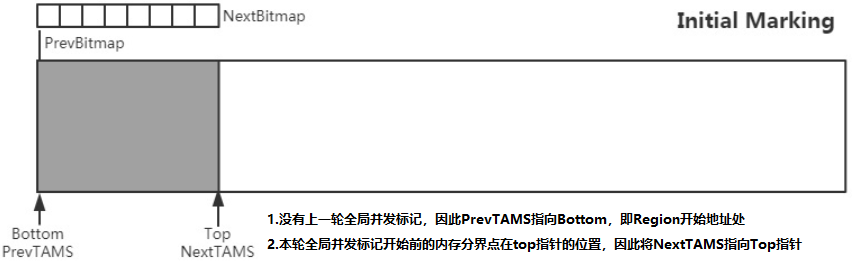
之前我们介绍了TAMS指针在全局并发标记中的作用，这次我们再总结回顾一下

1. **第1轮全局并发标记开始之前**

首先是一个空的Region开始被使用，此时触发了Mixed GC发生的条件，在一次Yong GC开始时，Mixed GC会开始执行，此时进入全局并发标记流程

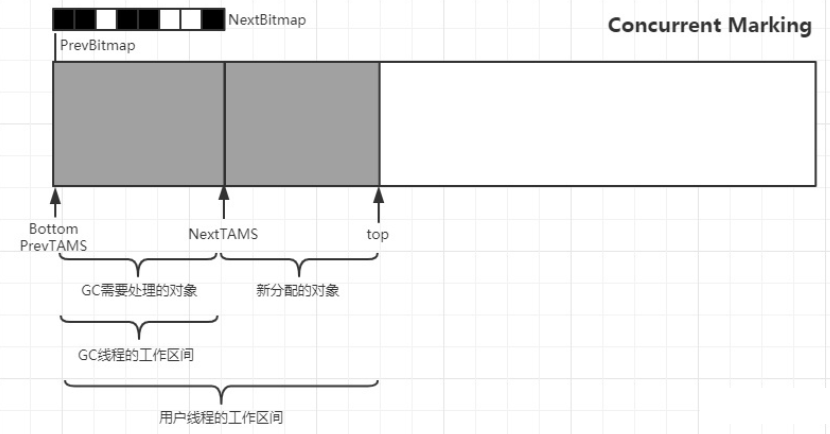
1. **第1轮全局并发标记的初始标记(Initial Marking)阶段**

Yong GC执行的过程中，同步执行初始标记流程，此阶段将会枚举GC Roots，并修改NextTAMS指向全局并发标记开始前已使用的内存分界点的地址，即Top指针此时指向的地址，如下图：



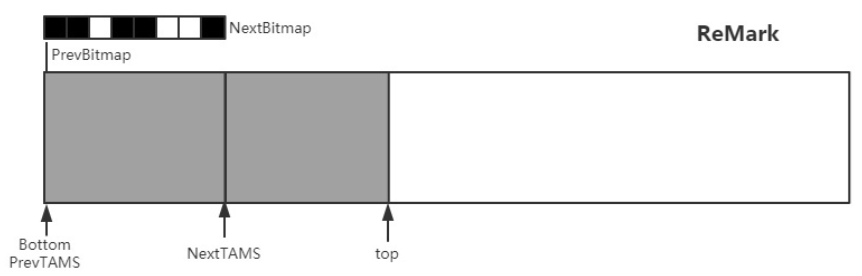
1. **第1轮全局并发标记的并发标记(Concurrent Marking)阶段**

G1的垃圾回收线程对[Bottom, NextTAMS)区间的内存区域执行可达性分析，将结果记录在NextBitmap中。用户线程并发执行，如果用户修改了[Bottom, nextTAMS)区间内对象的引用关系，则通过SATB算法记录，等到下一阶段再次标记；如果用户希望创建新的对象，G1将移动top指针，如下图：



1. **第1轮全局并发标记的最终标记阶段**

暂停用户线程，处理SATB记录下的少量有引用变动的对象，此阶段会发生STW，此时G1认为NextBitmap中黑色区域对应的对象和[NextTAMS, Top)区域的对象都是存活的，有浮动垃圾可以等下次再标记

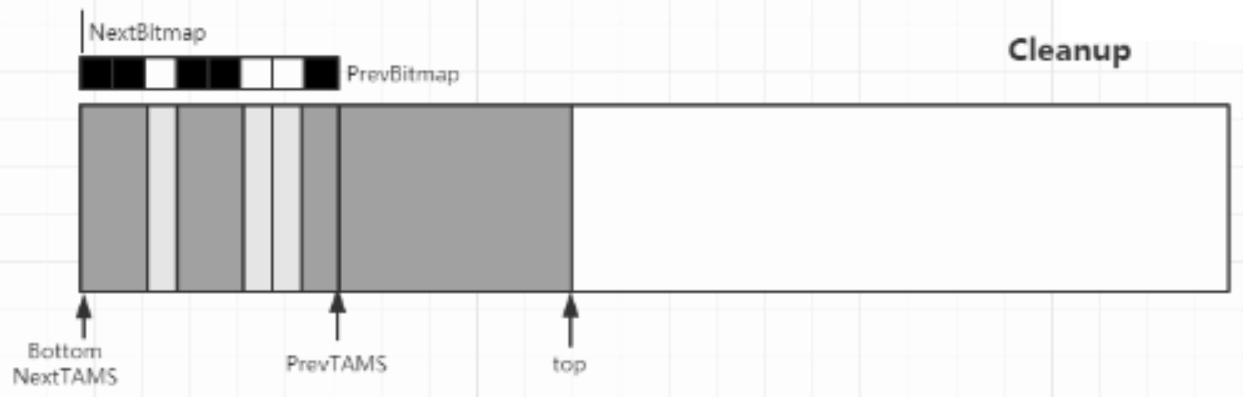


1. **第1轮全局并发标记的清理阶段**

此阶段会

1. 交换NextBitmap和PrevBitmap，并将NextBitmap置空
2. 交换NextTAMS指针和PrevTAMS指针的位置，此时PrevTAMS将指向并发标记开始前top指针所在的位置

如下图：

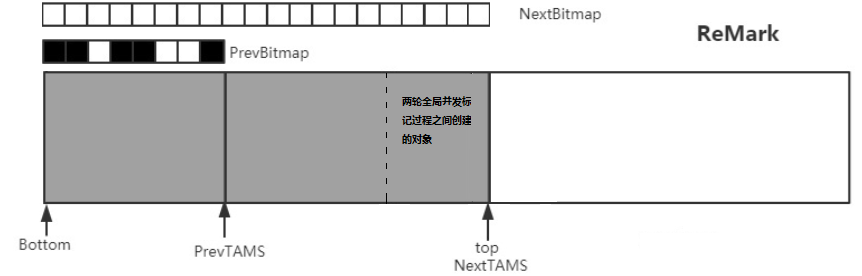


接下来G1根据回收价值和成本决定是否回收这个Region，如果回收了，那么回到第一步重新开始，如果不回收，进入下一步

1. **第2轮全局并发标记的初始标记阶段**

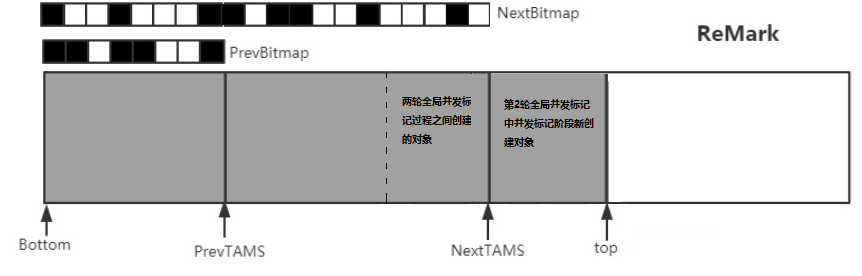
修改NextTAMS指向全局并发标记开始前已使用的内存分界点的地址，即Top指针此时指向的地址

给NextBitmap创建一个空的bitmap用于记录本次全局并发标记结果

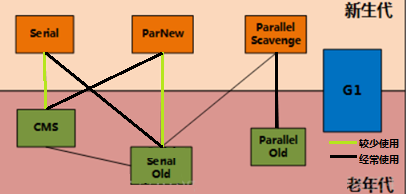
****

1. **第2轮全局并发标记的并发标记阶段**

GC线程对[Bottom, NextTAMS)之间的对象进行并发标记，如果有新对象创建，将在[NextTAMS,Top)区间内进行

****

#### 1.3.3.9 收集器的配合



**Client模式：**

Serial收集器是client模式下的新生代垃圾收集器，可以和serial Old、CMS(很少这么做)进行搭配

**Server模式：**

ParNew是激活CMS后的默认新生代垃圾收集器，常和CMS搭配，也可以和Serial Old进行搭配(很少这么做)

Parallel Scavenge(吞吐量优先收集器)早期与Serial old搭配使用(事实上Parallel Scavenge收集器本身实现了PS MarkSweep收集器，并非直接调用Serial old，只是PS MarkSweep和Serial Old的实现相同)，但性能受到其拖累，后常与Parallel Old配合使用

CMS在并发失败时会冻结用户线程，启用Serial Old重新进行老年代垃圾收集

**JDK1.7和1.8 server模式下默认使用Parallel Scavenge+Seriel Old组合，JDK9起默认使用G1收集器**

## 1.4内存分配机制

自动化内存管理的目标是解决两个问题：自动给对象分配空间、自动回收分配给对象的内存。关于内存回收，我们已经了解过了，接下来介绍内存的分配机制。

坦白地说，《java虚拟机规范》并未规定新生对象的创建和存储细节，其具体实现由使用哪种垃圾收集器以及相关JVM参数决定，而以下几项规则是使用Serial+Serial Old得出的，服务器端几乎不使用这种组合。因此，虽然以下几项规则大多是普适的，但也有部分收集器并不遵循这些规则，请谨慎参考

### 1.4.1对象优先在Eden分配

在大多数情况下，对象优先在Eden区分配，但并不绝对(如下面几节)。当Eden区空间不足时，虚拟机将发起一次Minor GC

### 1.4.2大对象直接进入老年代

大对象是指需要大量连续空间的JAVA对象，遇到大对象对于JVM来说是一个坏消息，更糟的消息是遇到一群”短命”的大对象。这种情况会导致JVM明明还有不少空间却要提前进行GC以获取连续的空间，且大对象在复制时需要更高的开销，因此在编写代码时就应尽量避免大对象

HotSpot提供了-XX:PretenureSizeThreshold参数，指定大于该值的对象直接在老年代分配，这么做的目的是为了避免在Eden区及两个Survivor区之间来回复制所造成的开销。

请注意！**-XX:PretenureSizeThreshold参数只对Serial和ParNew两款新生代收集器有效，其他收集器如Parallel Scavenge并不支持**，如果希望开启该功能，可以考虑ParNew+CMS组合

### 1.4.3大龄对象进入老年代

HotSpot虚拟机的大多数收集器都采用了分代收集的方式来管理堆内存，为此，JVM定义了一个age计数器存放在对象头中。每经历一次Minor GC后，若对象仍存活，则age+1，当其年龄达到一定阈值(默认为15)，就会被晋升到老年代中。晋升的阈值可以通过-XX:MaxTenuringThreshold参数设置

### 1.4.4动态对象年龄判定

为了更好地适应不同程序的内存情况，JVM并不是永远要求只有对象年龄大于-XX:MaxTenuringThreshold的对象才能晋升老年代。如果在Survivor空间中相同年龄所有对象的大小总和大于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

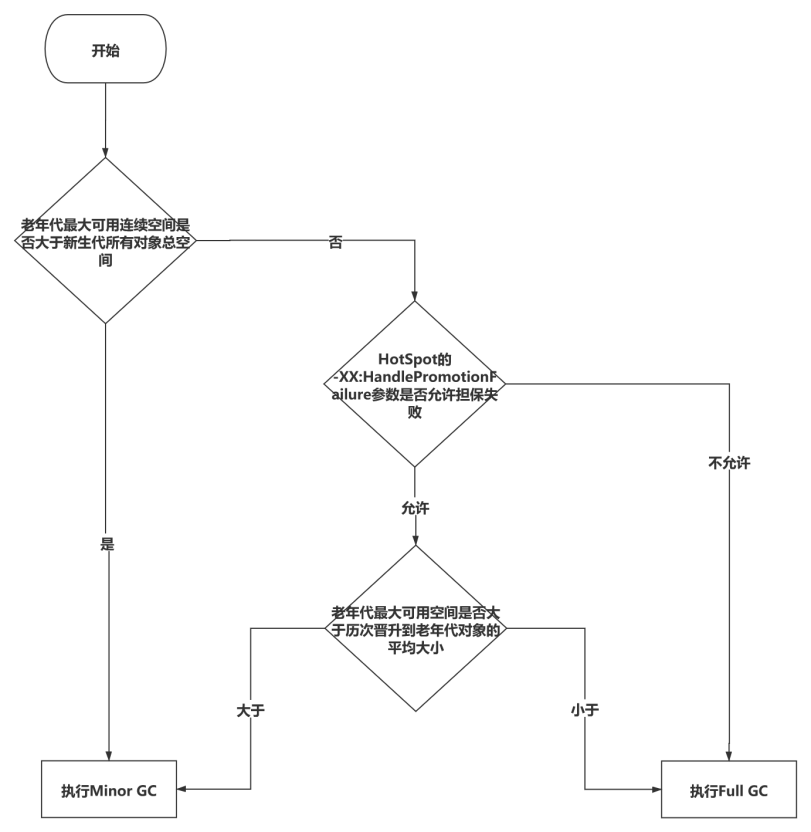
### 1.4.4空间分配担保

对于使用标记复制算法的新生代收集器，在发生Minor GC前，虚拟机必须先检查老年代最大可用连续空间是否大于新生代所有对象的大小。如果大于，那么这次Minor GC是绝对安全的。否则，JVM会先判断-XX:HandlePromotionFailure参数的值是否允许担保失败。如果允许，继续检查老年代最大可用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，则进行Minor GC，虽然可能失败但风险较小。如果小于，或者-XX:HandlePromotionFailure参数不允许担保，则由Minor GC改为Full GC

为什么要这么做呢？因为标记复制算法为了提高内存利用率，使用了一个Survivor空间作为轮换备份，因此在存活对象超过单个Survivor空间大小之后，就需要老年代来为其进行分配担保，将Survivor无法容纳的存活对象送入老年代，因此需要判断老年代是否可以容纳这些存活对象，如果无法容纳，就需要进行一次Major GC，实际上加上之前的Minor GC，也就是一次完整的Full GC了。

我们可以看到，判定流程中有一个是“继续检查老年代最大可用连续空间是否大于历次晋升到老年代对象的平均大小”，这其实也是赌概率，具有一定风险，如果赌错了，那就得老老实实地进行Full GC，这样一来兜的圈子就大了，但综合来看还是会把-XX:HandlePromotionFailure参数打开以避免Full GC过于频繁

需要注意的是，自JDK6 Update 24之后，-XX:HandlePromotionFailure参数就不再影响到虚拟机的空间担保策略了，相当于该参数默认开启无法关闭。



## 1.5 常用jvm参数

|  |  |  |
| --- | --- | --- |
| **参数** | **涵义** | **示例** |
| -Xms | 堆内存初始化大小，此值可以设置与-Xmx相同，以避免每次垃圾回收完成后JVM重新分配内存 | -Xms 1024m |
| -Xmx | 最大堆内存大小 | -Xmx 1024m |
| -Xmn | 设置年轻代大小，整个JVM内存大小=年轻代大小 + 年老代大小 + 持久代大小。持久代一般固定大小为64m，所以增大年轻代后，将会减小年老代大小。此值对系统性能影响较大，Sun官方推荐配置为整个堆的3/8 | -Xmn 2g |
| -Xss | 设置每个线程的堆栈大小。JDK5.0以后每个线程堆栈大小为1M，以前每个线程堆栈大小为256K。根据应用的线程所需内存大小进行调整。在固定物理内存下，减小这个值能生成更多的线程。但是操作系统对一个进程内的线程数还是有限制的，不能无限生成，一般在3000~5000左右 | -Xss 128k |
| -XX:NewRatio | 设置年轻代（包括Eden和两个Survivor区）与年老代的比值（除去持久代）。若设置为4，则年轻代与年老代所占比值为1：4 | -XX:NewRatio=4 |
| -XX:SurvivorRatio | 设置年轻代中Eden区与Survivor区的大小比值。若设置为4，则Survivor区与Eden区的比值为1:1:4，一个Survivor区占整个年轻代的1/6 | -XX:SurvivorRatio=4 |
| -XX:MaxPermSize | 设置永久代大小，JDK8移除了永久代，此参数失效 | -XX:MaxPermSize=16m |
| -XX:MetaspaceSize | 主要控制元空间GC发生的初始阈值，也就是最小阈值。即当使用的metaspace空间到达了阈值的时候，就会触发Metaspace的GC | -XX:MetaspaceSize=128m |
| -XX:MaxTenuringThreshold | 设置垃圾最大年龄。如果设置为0的话，则年轻代对象不经过Survivor区，直接进入年老代。 | -XX:MaxTenuringThreshold=0 |
| -XX:+UseSerialGC | 设serial收集器 |  |
| -XX:+UseParallelGC | 设置吞吐量优先收集器 |  |
| -XX:+UseParalledlOldGC | 设置吞吐量优先老年代收集器 |  |
| -XX:+UseConcMarkSweepGC | 设置cms收集器 |  |

Eg：

java -Xmx3550m -Xms3550m -Xss128k -XX:NewRatio=4 -XX:SurvivorRatio=4 -XX:MaxPermSize=16m -XX:MaxTenuringThreshold=0

## 2.常用的性能监控和故障处理工具

### 2.1 jps:虚拟机进程状况处理工具

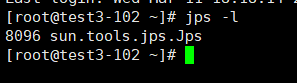
JSP(Jvm Process Status Tool)是一款常用的工具，可以用来查看虚拟机运行了哪些进程，并显示虚拟机执行主类名称以及其LVMID(进程id)。虽然功能较为单一，但确是使用频率最高的一款工具，命令格式：

**jps [options] [hostid]**

options的选项为：

|  |  |
| --- | --- |
| **选项** | **作用** |
| **-q** | 只输出LVMID，忽略主类名称 |
| **-m** | 输出虚拟机进程启动时传递给主类main()函数的参数 |
| **-l** | 输出主类全名，如果执行的是jar包，输出jar路径 |
| **-v** | 输出虚拟机进程启动时的JVM参数 |

eg：



### 2.2 jstat:虚拟机统计信息监视工具

用于jvm虚拟机统计信息监控，如监控jvm内存使用率等数据

命令格式：

**jstat [options vmid [ interval [s | ms] [ count] ] ]**

1. options是可选参数，其功能可分为：类加载、垃圾收集、运行期编译三类，其可选列表如下：

|  |  |
| --- | --- |
| **选项** | **作用** |
| -class | 监视类加载数量、卸载数量、加载卸载总空间及装载耗费时间 |
| -gc | 1. **S0C ：**年轻代中第一个survivor的容量 (字节) 2. **S1C ：**年轻代中第二个survivor的容量 (字节) 3. **S0U ：**年轻代中第一个survivor目前已使用空间 (字节) 4. **S1U ：**年轻代中第二个survivor目前已使用空间 (字节) 5. **EC ：**年轻代中Eden区的容量 (字节) 6. **EU ：** 年轻代中Eden目前已使用空间 (字节) 7. **OC ：**Old代的容量 (字节) 8. **OU ：**Old代目前已使用空间 (字节) 9. **MC ：**方法区的容量 (字节) 10. **MU ：**方法区目前已使用空间 (字节) 11. **CCSC ：**压缩类空间大小(字节) 12. **CCSU ：**压缩类空间使用大小(字节) 13. **YGC ：**JVM从启动至今年轻代垃圾回收次数 14. **YGCT ：**JVM从启动至今年轻代垃圾回收消耗时间(s) 15. **FGC ：**JVM从启动至今老年代垃圾回收次数 16. **FGCT ：**JVM从启动至今老年代垃圾回收消耗时间(s) 17. **GCT ：**JVM从启动至今垃圾回收消耗总时间(s) |
| -gccapacity | 监视内容与-gc基本相同，但关注点在于各区域使用到的最大/最小容量 |
| -gcutil | 输出内容类似-gc，但关注已使用空间占全部空间的百分比 |
| -gccause | 与-gcutil功能一样，不过多了一项”导致上次垃圾收集的原因” |
| -gcnew | 监视新生代垃圾收集状况 |
| -gcnewcapacity | 监视内容与-gcnew基本相同，但关注点在于使用到的最大/最小容量 |
| -gcold | 监视老年代垃圾收集状况 |
| -gcoldcapacity | 监视内容与-gcold一致，但关注点在于使用到的最大/最小容量 |
| -gcpermcapacity | 输出永久代使用到的最大最小空间 |
| -compiler | 输出即时编译器编译过的方法、耗时等信息 |
| -printcompilation | 输出已经被即时编译的方法 |

### 2.3 jinfo:java配置信息工具

用于实时地查看和调整虚拟机的各项参数

命令格式：

**jinfo [option] pid**

|  |  |
| --- | --- |
| **选项** | **作用** |
| < no option > | 输出全部的参数和系统属性，eg: jinfo 4747 |
| -flag <name> | 输出对应名称的参数 |
| -flag +/- <name> | 开启或者关闭对应名称的参数 |
| -flag <name>=<value> | 设定对应名称的参数 |
| -flags | 输出全部的参数 |
| -sysprops | 输出系统属性 |

### 2.4 jmap:java内存映像工具

用于生成堆转储快照

命令格式：

**jmap [option] vmid**

|  |  |
| --- | --- |
| **选项** | **作用** |
| -dump:[live,]format=b,file=<filename> | 生成java堆转储快照，live可选，如果选中，则只dump出存活对象，eg：jmap -dump:live,format=b,file=myjmapfile.txt 19570 |

### 2.5 jhat:虚拟机堆转储快照分析工具

jhat命令往往被与jmap命令搭配使用，来分析jmap生成的堆转储快照。除非必要，一般来说这个命令很少使用，因为分析是一个很耗资源的命令，我们通常是将快照下载到其他机器上，使用别的工具进行分析，常用的工具是visualVM。

### 2.6 jstack:java堆栈跟踪工具

用于生成虚拟机当前时刻的线程快照。线程快照是指当前虚拟机中每一条线程正在执行的堆栈集合。生成该快照的目的通常是定位线程出现长时间停顿的原因，如死锁、死循环、请求外部资源导致的长时间挂起等

命令格式：

**jstack [option] vmid**

|  |  |
| --- | --- |
| **选项** | **作用** |
| -F | 当正常输出的请求不被响应时，强制输出线程堆栈 |
| -l | 除堆栈外，显示关于锁的附加信息 |
| -m | 如果调用到native方法时，可以显示c/c++的堆栈 |

## 3.调优案例

先介绍几个常用参数

|  |  |
| --- | --- |
| **参数** | **涵义** |
| -Xmx | 堆的最大 大小，等同于-XX:MaxHeapSize |
| -Xms | 堆的初始化初始化大小，也可以理解为堆的最小大小，如果需要进一步细化，初始化大小用-XX:NewSize，最大大小用-XX:MaxNewSize |
| -Xmn | 新生代的大小，这个值设定了也可以得到老年代大小 |
| -Xss | 线程栈大小 |

### 3.1 大内存硬件的部署策略

**场景：**对一个具有15w PV的在线文档网站进行了硬件升级，将32位centOS更换为64位，将4G内存提升为16G内存，原先的4G内存只分配了1.5G的堆内存给JVM，现在升级为16G后，使用-Xmx和-Xms参数将jvm的内存大小固定为12GB，选用64位JDK5，JVM虚拟机以服务端模式运行，选用吞吐量优先收集器(Parallel Scavenge)，运行了一段时间后发现十分不理想，网站经常出现不定期地失去响应

**分析：**经过检查发现，网站不定期失去响应是GC的”Stop The World”导致的，由于程序设计的原因，访问文档时会将整个文档读取到内存中，这导致堆中出现大量文档序列化产生的大对象，大多数大对象都在分配时就进入老年代了(虽然Parallel Scavenge不支持-XX:PretenureSizeThreshold参数，但仍有自己的算法去判断大对象)，没有在Minor GC中被清理，这种情况下，即使是12G的堆内存也很快被耗尽，导致每隔几分钟出现十几秒的间隔，用户体验极差。这种情况在硬件升级之前反而没那么严重，尝试将堆内存重新缩小到1.5GB，长时间停顿就减轻了，但这样过于浪费硬件资源。其实这个案例最大的问题是在于堆内存过大，每次Full GC耗时过长，而因为业务特殊性+程序设计不合理产生了大量的大对象，加剧了Full GC出现的频率，导致效果十分不理想

**调优思路：**目前，对于在大内存的硬件设备上部署应用，我们通常有两种方式：

1. **通过一个单独的Java虚拟机来管理大量的Java内存**
2. **在Linux上使用容器来区分隔离各JVM，使用多个虚拟机来利用硬件资源**
3. **单个JVM**

本案例采用了第一种方式，但是并没有使用比较前沿的如Shenandoah、ZGC等以控制延迟为目标的垃圾收集器。其实使用以控制延迟为目标的垃圾收集器是最值得优先考虑的方案，但并不是说使用别的垃圾收集器就不能控制延迟，只是需要考虑的问题更多

如果期望使用传统垃圾收集器来管理大内存，就必须要控制Full GC的频率，至少不能让Full GC经常在用户使用时出现。譬如控制Full GC在十几个小时甚至几天内不出现，或者在晚上通过定时任务触发Full GC甚至重启来清理JVM内存。当然还有很多其他问题需要解决，因此推荐使用第二种方式来管理大内存

1. **多个JVM**

现在常用的做法是在一台Linux服务器上部署多个容器(如docker等)来隔离各个JVM，这样一台服务器可以有多个JVM同时运行。在本案例中可以采取这种方式，运行多个web应用在不同的JVM中，然后再搭建一个负载均衡服务器来分配访问请求，这样可以显著降低Full GC消耗的时间

但是这样做也是有缺点的：

1. **各节点竞争资源。**这种现象在同一台机器部署多个相同应用的情况下更严重，最典型的就是磁盘竞争，各应用如果同时访问某个磁盘文件，可能造成IO异常
2. **很难利用资源池。**一般来说各应用都是自己用自己独立的连接池，这样可能导致一些应用的连接池满了，另外一些应用的连接池仍有空余，虽然可以使用JNDI来解决这个问题，但引入了复杂度且可能带来额外的性能消耗。譬如数据库连接池
3. **大量使用本地缓存的应用。**本地缓存存放在各应用内，如果应用大量使用了本地缓存，会导致每个逻辑节点上都有一份缓存，既浪费资源，也可能导致本地缓存不一致。可以考虑改为集中式缓存来解决问题

这个案例最终采用了第二种方式来部署，部署了5个应用，每个容器分配了2G内存，每个堆内存大小固定为1.5G，也就是说部署项目共使用了10G，另外采用了Ngnix服务器进行负载均衡。考虑到网站的压力主要在磁盘和内存访问，对CPU性能消耗并不大，替换为CMS收集器进行垃圾回收。采用这种方案进行部署后，情况大为改善，速度比起硬件升级前有较大提升

### 3.2 堆外内存溢出

**场景：**这是一个比较小型的项目，项目采用了CometD 1.1.1作为服务端推送框架，内存大小为4G，采用32位windows系统运行，在应用测试阶段发现服务端不定时抛出内存溢出异常，尝试将堆内存调大到1.6GB(最大只能调到1.6G)，基本没有效果，抛出内存异常反而更加频繁了。加入-XX:+HeapDumpOnOutOfMemoryError参数，抛出异常时竟然没有任何文件产生。无奈之下只好挂着jstat紧盯屏幕，Eden区、Survivor区、老年代的内存都很稳定，压力不大，垃圾收集并不频繁，但仍然不断抛出内存溢出异常，最后再系统日志中找到异常堆栈，发现是由直接内存溢出抛出的异常

**分析：**我们都知道，系统对于可识别的内存大小是有限制，对于单个进程可管理的内存大小也有限制。在32位操作系统下，32位系统最多可映射2^32 Byte (4G)的内存空间，也就是说给32位操作系统插上8G的内存是没有意义的。64位操作系统最多可以映射2^48 Byte(256TB)。

这仅仅是寻址空间大小，单个进程的内存管理大小也是有限制的，32位windows限制单个进程最大允许使用2GB空间，同一进程下的所有线程都共享这个内存空间，其余2GB为操作系统保留。当然也可以通过boot.ini进行设置，将单个进程的限制大小改到3GB。详见<https://docs.microsoft.com/zh-cn/previous-versions/sql/sql-server-2008-r2/ms189334(v=sql.105)?redirectedfrom=MSDN>

对于32位Linux系统，单进程默认限制为3GB，剩余1GB为操作系统内核保留

回到我们的案例中，案例中采用的是默认32位windows操作系统，也就是说，单个java进程的最大内存大小为2GB，其中1.6GB被划给了堆，而直接内存不属于运行时数据区域，也就是说，即使不算方法栈和程序计数器等区域，Direct Memory最多也只能使用0.4GB内存。虽然在垃圾收集时，虚拟机会对直接内存进行垃圾收集，但这只是清理堆内存时”顺带”做的。也就是说，堆内存不足时，可以主动通知垃圾收集器进行内存清理，而直接内存不足，是无法通知垃圾收集器清理垃圾的。尤其在调高堆内存之后，直接内存的可用空间更小，导致异常抛出更加频繁。而本案例中采用的CometD 1.1.1框架存在大量NIO操作，需要使用到大量的直接内存

因此在处理小内存系统或32位系统时，除了堆内存外，我们还需要注意以下区域可能占用较多内存：

1. **直接内存：**如本案例，当该区域空间不足时会抛出OutOfMemoryError或OutOfMerrorError：Direct buffer memory异常。可以通过-XX:MaxDirectMemorySize调整大小
2. **线程堆栈：**虚拟机栈空间不足，会抛出StackOverflowError(可能是线程请求的栈深度超过虚拟机所允许的深度)或OutOfMemoryError(如果jvm栈容量可以动态扩展，当栈扩展无法申请到足够的内存)异常
3. **Socket缓存区：**如果Socket比较多，内存不够分配，也会抛出IOException:Too many open files异常
4. **JNI代码：**JNI占用的内存也不是堆中的，是本地方法栈中分配的，如果调用JNI方法过多也可能发生异常
5. **虚拟机和垃圾收集器：**虚拟机和垃圾收集器也要消耗内存

### 3.3 不恰当的数据结构导致过长停顿

**场景：**有一个后台分析的RPC服务，采用ParNew+CMS组合，配置为-Xms4g -Xmx8g -Xmn1g。在某次更新之前，Minor GC的时间基本都维持在30毫秒以内。但更新后的Minor GC时间暴增到500毫秒。经了解后，本次更新是增加了一个定时分析功能，需要每十分钟加载一个约80MB的文件到内存中进行分析，这些数据在内存中会产生超过100W个Map.Entry<Long,Long>对象。

**分析：**ParNew采用的是标记复制算法，之前能维持30毫秒内的Minor GC是因为大部分对象是朝生夕灭的，需要复制的存活对象很少。进行代码更新后，800MB的Eden空间很快就满了(Eden:Survivor:Survivor =8:1:1)，但Minor GC发生时，占内存最大的HashMap仍然存活，那么复制算法的代价就会变得很大

**调优思路：**

1. 如果不修改程序，仅从GC调优的角度来考虑，可以考虑将survivor区设为0，让新生代存活对象第一次Minor GC后立即进入老年代，等待Major GC再去清理，这样一来会加剧Full GC发生的次数，导致程序不定期Stop The world
2. 要想从根本上解决问题，我们就需要先分析一下HashMap的空间效率。在HashMap<Long,Long>结构中，只有两个长整型是有效数据，共2×8个字节。长整型被包装成Long后，额外多了2×8字节的Mark Word和2×8字节的Klass指针。同样的，这两个Long组成Map.Entry，又多了16个字节的对象头，还有一个8字节的next字段和4字节的int型hash字段，为了对齐，还需要添加4字节的空白填充，最后HashMap还有对Entry的8字节引用。这样实际空间利用效率为

(2×8byte)的long /

[(2×8byte)的long + (2×16byte)的Long对象头+(16byte)的Entry对象头+(8byte)Entry的next指针+(4byte)Entry的hash+(4byte)Entry的对齐空白+(8byte)对Entry的引用]=18%

需要根据实际业务提高数据结构的空间利用效率，减轻垃圾收集器的负担

# 2.虚拟机执行子系统

## 2.1Class文件结构

Class文件是一组以8个字节为基础单位的二进制流，各数据项目严格按照顺序紧凑地排列在文件之中，中间不添加任何分隔符，这意味着Class文件的内容几乎全部是程序运行的必要数据，没有空隙存在。当遇到需要占用8个字节以上空间的数据项时，Class文件会按“高位在前”的方式分隔为若干个8字节进行存储

### 2.1.1无符号数和表

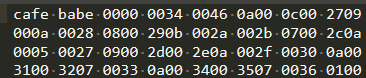
根据《Java虚拟机规范》的规定，Class文件格式采用一种类似C语言结构体的伪结构来存储数据，这种伪结构只有两种数据类型**：”无符号数”和表**。

1. **无符号数：**无符号数是基本的数据类型，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，数值量或按UTF8编码构成的字符串值
2. **表：**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。为了便于区分，所有表的命名都习惯性地以\_info结尾。表用于描述有层次关系的复合结构数据，整个Class文件也可以视作一张表

### 2.1.2魔数与Class文件的版本

每个Class文件的前4个字节被称为魔数(Magic Number)，魔数的作用是确定这个文件是否能被虚拟机接受，作用类似于文件的扩展名，但是由于文件的扩展名可以随意改动，基于安全考虑，采用魔数来识别文件。很多文件格式标准都采用魔数的方式进行身份识别，Class文件使用的魔数是16进制的”cafebabe”，即0xCAFEBABE。(1字节=8bit，可以存储8个2进制数字或2个16进制数字)

紧跟着魔数的4个字节存储的是Class文件的版本号，其中第5、6个字节是次版本号，第7、8个字节数主版本号，如下图，版本号为3\*16+4=52，java的版本号是从45开始的(即JDK1.0-1.1使用了45.0-45.3)，jvm拒绝执行超过自己版本号的Class文件，因为JVM能向前兼容而不能向后兼容



### 2.1.3运行时常量池

紧接着主次版本号之后的是运行时常量池，运行时常量池主要用于存放两种常量：

1. **字面量(Literal)：**字面量比较接近JAVA语言层面的常量概念，如文本字符串、被声明为final的常量值等
2. **符号引用(Symbolic References)：**符号引用以一组符号来描述所引用的目标，本质上是个字符串。符号引用可以是任何形式的字面量，只要使用时能无歧义地定位到目标即可

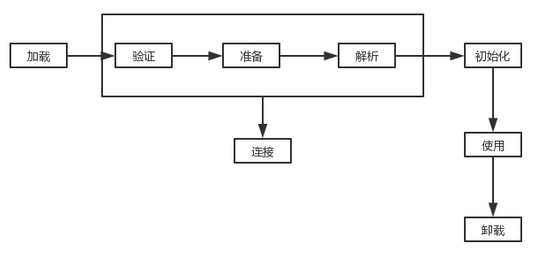
## 2.2虚拟机类加载机制

Class文件中的各种信息最终都需要加载到虚拟机中后才能被运行和使用，本节将介绍如何加载Class文件

### 2.2.1类的生命周期

一个类型从加载到虚拟机内存到卸载出内存为止，会经历如下几个步骤：

1. **加载(Loading)**
2. **验证(Verification)**
3. **准备(Preparation)**
4. **解析(Resolution)**
5. **初始化(Initialization)**
6. **使用(Using)**
7. **卸载(Unloading)**



其中，**加载、验证、准备、初始化**和**卸载**这五个阶段的开始执行顺序是确定的。而**解析**阶段则不一定，它在某些情况下可以在**初始化**之后再开始。需要注意，这里能确定的仅仅是开始顺序，因为这些阶段通常是交叉执行的，会在一个阶段执行的过程中调用、激活另一个阶段

### 2.2.2类的加载过程

#### 2.2.2.1加载(Loading)

在加载阶段，JVM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访问入口

Jvm规范中对这三个要求并没有规定得太具体，因此可以在这一点”间隙”上玩出各种”花样”来，比如第一条，它并没有规定二进制字节流必须从某个Class文件中获取，因此衍生了很多重要的技术：

1. 从ZIP压缩包获取，后来发展出JAR、EAR、WAR包
2. 从网络中获取，后来发展出WebApplet
3. 运行时生成，后来发展成反射技术
4. 由其他文件生成，典型的如JSP
5. 从数据库中读取，极为少见
6. 从加密文件中获取，典型的Class文件反编译保护措施

…

类加载既可以通过虚拟机内置的引导类加载器完成，也可以通过用户自定义的类加载器完成，开发者可以通过自定义类加载器来控制获取字节流的方式。

对于数组类而言，情况就不同了，数组类不通过类加载器创建，它是由虚拟机直接在内存中动态构造出来的。但这并不是说数组类和类加载器无关，数组的**元素的类型**还是需要依靠类加载器进行加载。

加载阶段结束后，被加载的二进制字节流就按照虚拟机设定的格式存储在方法区中了(还会在堆中产生一个Class对象供程序去访问方法区的类型数据)。

加载阶段和连接阶段的部分动作(如部分字节码文件格式验证动作)是交叉进行的，也就是说加载阶段尚未完成，连接阶段就可能已经开始。但二者的开始时间仍会保持固定的先后顺序

#### 2.2.2.2连接(Linking)

##### 2.2.2.2.1验证(Verification)

验证是连接阶段的第一步，这一阶段的目的是确保Class字节流中包含的信息符合《java虚拟机规范》的全部约束要求。

Java语言在编译的时候就拒绝诸如：数组越界访问、强转对象为未实现类型、跳转到不存在的代码行等行为，如果这么做了，编译时就会报错，根本无法生成Class文件。但由于JVM规范并未限制二进制字节流的来源，所以被加载的二进制字节流并不安全，如果不检查输入的二进制字节流，那么可能会因为载入了错误或恶意的字节流码导致JVM崩溃，因此验证字节码是JVM自我保护的一项必要措施

验证阶段主要包括以下四个检验动作：文件格式验证、元数据验证、字节码验证、符号引用验证。

1. **文件格式验证**

这一阶段主要验证字节流是否符合Class文件规范，版本是否兼容等。目的是为了保证输入的字节流能正确地解析并存储于方法区内，通过了这个阶段，这段字节流才被允许进入方法区中存储，后面三个验证阶段都是基于方法区的存储结构上进行的，不会再直接读取操作字节流了

1. 是否以魔数0xCAFEBABE开头
2. 主次版本号是否在虚拟机的接受范围内
3. 检查常量tag标志，常量池中是否存在不支持的常量类型
4. 指向常量的各种索引是否有指向不存在的常量或不符合类型常量的情况

…

1. **元数据验证**

这一阶段是进行语义分析

1. 这个类是否有父类(出Object外都应该有父类)
2. 这个类是否继承了不被允许继承的类(final修饰)
3. 如果这个类不是抽象类，其是否实现了所有必要实现的方法
4. 类中的字段和方法是否和父类产生矛盾(如覆盖了父类的final字段，或出现不合规则的重载等)

…

1. **字节码验证**

这是验证阶段中最复杂的阶段，这一阶段要对类的方法体进行校验分析，确保被验证类的方法在运行时不会做出危害虚拟机安全的行为。如

1. 保证不会出现诸如”在操作栈放了一个int，使用时却按long类型加载到本地变量表中”这类操作
2. 保证任何跳转指令都不会跳转至方法体之外
3. 确保方法体的类型转换总是有效的

…

1. **符号引用验证**

最后一个阶段的校验行为发生在虚拟机将符号引用转化为直接引用的时候，这个动作将在连接的第三个阶段——解析阶段中发生(由此可以看出类加载的各阶段是交叉进行的)。

1. 是否可以通过符号引用的全限定名找到对应的类
2. 检查类、字段、方法的可访问性(private、protected、public)

…

验证阶段对于类的加载过程来说是一个非常重要，但却并非必须的阶段。如果程序运行的代码来源可信并且已被反复使用或验证过，就可以考虑使用-Xverify:none参数来关闭大部分类验证措施以缩短虚拟机的类加载时间

##### 2.2.2.2.2准备(Preparation)

准备阶段是正式为静态变量分配内存并设置变量初始值的阶段。理论上来说，静态变量所使用的内存都应当在方法区中分配，在JDK1.7以前，HotSpot使用永久代来实现方法区，此时的静态变量无论是在物理上还是逻辑上都是存储在方法区中，但JDK8及以后，HotSpot采用元空间实现方法区，而类变量(静态变量)会随着Class对象一起存放在Java堆中(参见1.1.2.1节)，这时候静态变量仅逻辑上存储在方法区了

“准备阶段是正式为静态变量分配内存并设置变量初始值的阶段”这句话有两点容易混淆：

1. 这里仅为静态变量分类内存，不包括实例变量，实例变量会在对象实例化时分配在java堆中
2. 设置初始值并非赋值，这里的初始值**通常**指各数据类型的零值，如public static int a=2;这句，在准备阶段过后，其初始值为0而非2，真正赋值需要到初始化阶段才会进行。不过当类变量被final修饰时，那么在准备阶段就会进行初始化赋值，这是因为final对象只能进行一次赋值。

##### 2.2.2.2.3解析(Resolution)

解析阶段需要做的事情是将常量池内的符号引用替换为直接引用。那么什么是符号引用什么是直接引用呢？

**符号引用：**符号引用是以一组符号来描述所引用的目标，符号可以是任何形式的字面量，只要使用时能无歧义地定位到目标即可。其和虚拟机的内存布局无关，不同虚拟机能接受的符号引用必须一致

**直接引用：**直接引用是可以直接指向目标的指针、偏移量或能直接定位到目标的句柄。直接引用是和虚拟机的内存布局直接相关的，同一个符号引用在不同虚拟机上翻译出来的直接引用一般不会相同。如果符号引用被解析为了直接引用，那么他们所指向的目标必定已存在于虚拟机的内存中

虚拟机并不是立即对所有的符号引用进行解析的，虚拟机会自行判断哪些类在类加载时就需要解析，哪些类在将要被使用前才解析

在验证阶段(Verification)的最后一个小阶段中，我们提到过，在解析阶段时会对能否解析进行验证

#### 2.2.2.3初始化(Initialization)

类的初始化阶段是类加载过程的最后一个阶段，这一阶段主要是对类变量和其他资源进行赋值和初始化操作。换种说法，初始化阶段就是执行 类构造器<clinit>()方法的过程，那么什么是**<clinit>**?

1. **<clinit>：类构造器。**在jvm第一次加载class文件时调用，包括静态变量初始化语句和静态块的执行。它是由**编译器自动收集类中的所有类变量的赋值动作和静态语句块（static）中的语句合并产生的，编译器收集的顺序是由语句在源文件中出现的顺序所决定的，静态语句块只能访问到定义在静态语句块之前的变量**。eg：

class Tests {  
 static {  
 *i* = 1;//对定义在静态代码块之后的变量可以赋值  
 System.*out*.println("m = " + *i*);//但不能访问  
 }  
 static int *i* = 2; //初始化开始前i值为0，结束后i值为2  
}

**JVM会保证在子类的<clinit>执行前，父类的<clinit>已执行完毕**，eg：

class Parent {  
 static {  
 *A* = 2;  
 }  
 static int *A* = 1;  
}  
  
class Child extends Parent {  
 public static int *B* = *A*;//B的值为1  
}

1. **实例构造器，即构造函数。构造函数调用之前会给所有非static字段分配内存并赋值，**在实例创建出来的时候调用，包括调用new操作符；调用Class或java.lang.reflect.Constructor对象的newInstance()方法；调用任何现有对象的clone()方法；通过java.io.ObjectInputStream类的getObject()方法反序列化。

**注意!应当避免在类变量或静态代码块中执行耗时过长的操作，因为<clinit>在多线程环境中会被加锁，如果多个线程同时初始化一个类，那么只会有一个线程执行<clinit>方法。如果执行了耗时过长的操作，那么可能造成多个线程阻塞**

**有很多人会误导说静态代码块比静态变量先初始化，实际上这是错误的，谁先谁后看他们在源文件中的顺序**

class SS {  
 public static volatile int *i* = *sout*();  
 static {  
 System.*out*.println("静态代码块执行");  
 *i* = 3;  
 }  
  
  
 public static int sout(){  
 System.*out*.println("静态变量初始化");  
 return 1;  
 }  
}

最终执行结果为：

静态变量初始化

静态代码块执行

### 2.2.3类加载器(Class Loader)

#### 2.2.3.1概述

Jvm开发设计团队有意将**加载阶段**中的”通过一个类的全限定名来获取定义此类的二进制字节流”这一动作放到JVM外部实现，以便让程序自己决定如何获取所需的类。实现这一动作的代码被称为”**类加载器(Class Loader)”**

类加载器虽然只实现了**加载阶段**的部分动作，但其在程序中的作用相当重要。即使是来自同一个Class文件的同一个类，如果被不同的类加载器加载了，那么这两个类必定不相等。每一个类加载器都有一个独立的类名称空间。**任意的一个类，都必须由加载它的类加载器和类本身共同确定其在JVM中的唯一性**

我们可以自定义自己的类加载器，虚拟机也有自己默认的类加载器，如果没有自定义类加载器，都将使用jvm默认类加载器

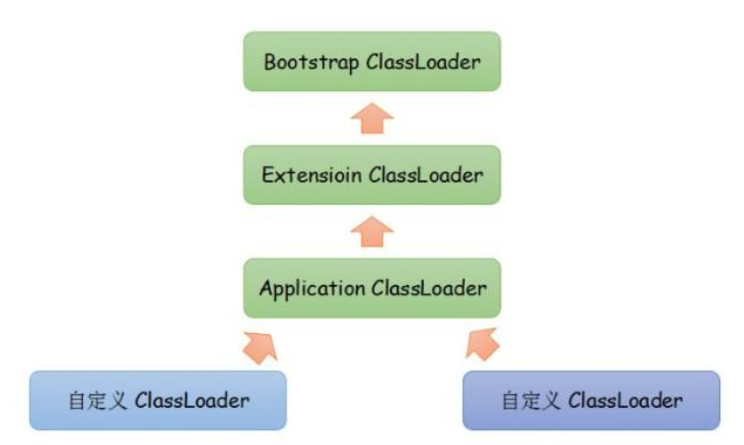
#### 2.2.3.2双亲委派

在HotSpot中存在三种类加载器：

1. **启动类加载器(BootStrap)：**这个类加载器使用C++实现，属于HotSpot的一部分，它无法被java程序直接引用。它负责加载<JAVA\_HOME>\lib目录下的类库
2. **扩展类加载器(Extension Class Loader)：**由java代码实现，可以被java程序直接使用。它负责加载<JAVA\_HOME>\ext目录下的类库。
3. **应用程序类加载器(Application Class Loader)：**由java代码实现，又被称为系统类加载器，由ClassLoader类中的getSystemClassLoader()方法即可获取该类加载器。她伏在加载所有用户路径(ClassPath)上的类库。如果程序没有自定义自己的类加载器，那么默认情况下就是使用应用程序类加载器

除了以上三种类加载器，用户还可以自定义自己的类加载器进行拓展

**双亲委派模型：**如果一个类加载器需要加载类，那么首先它会把这个类请求委派给父类加载器去完成，每一层都是如此，一直递归到顶层。当父加载器无法完成这个请求时，子类才会尝试去加载。这里的双亲其实就指的是父类，没有mother。父类也不是我们平日所说的那种继承关系，只是调用逻辑是这样。



这么做有什么优势呢？

如果用户自己写了一个java.lang.Object类，并放在程序的classpath当中，就会与rt.jar中的java.lang.Object重复，如果不使用双亲委派，那么系统中会出现多个版本的Object类，那么程序就会一片混乱。如果使用了双亲委派，那么所有的java.lang.Object都会被委托给BootStrap ClassLoader进行加载。

实现双亲委派的代码非常简单：

protected Class<?> loadClass(String name, boolean resolve) throws ClassNotFoundException{  
 synchronized (getClassLoadingLock(name)) {  
 // 首先，检查类是否已被加载  
 Class<?> c = findLoadedClass(name);  
 if (c == null) {  
 long t0 = System.*nanoTime*();  
 try {  
 if (parent != null) {

// 如果存在父加载器，由父加载器加载  
 c = parent.loadClass(name, false);  
 } else {  
 c = findBootstrapClassOrNull(name);  
 }  
 } catch (ClassNotFoundException e) {  
 // 如果父加载器抛出类未找到异常，说明父加载器无法加载  
 }  
  
 if (c == null) {  
 // 如果父加载器无法加载，就自行加载  
 long t1 = System.*nanoTime*();  
 c = findClass(name);  
  
 // this is the defining class loader; record the stats  
 sun.misc.PerfCounter.*getParentDelegationTime*().addTime(t1 - t0);  
 sun.misc.PerfCounter.*getFindClassTime*().addElapsedTimeFrom(t1);  
 sun.misc.PerfCounter.*getFindClasses*().increment();  
 }  
 }  
 if (resolve) {  
 resolveClass(c);  
 }  
 return c;  
 }  
}

#### 2.2.3.3如何打破双亲委派

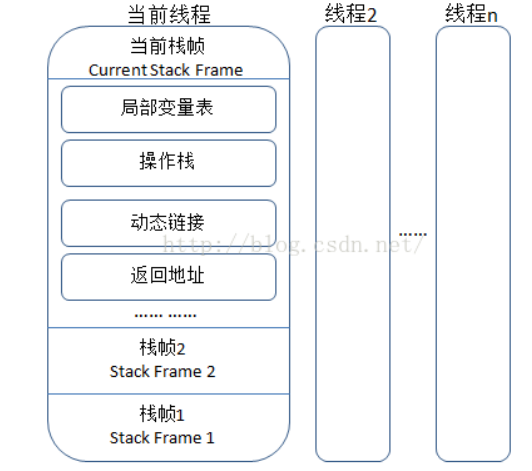
双亲委派是通过java.lang.ClassLoader的loadClass()方法进行的，如果想破坏双亲委派，只需要继承ClassLoader类，然后重写loadClass方法即可。如果想要实现自己的类加载逻辑而又不破坏双亲委派，只需要继承ClassLoader，然后重写findClass方法，默认的findClass如下：

protected Class<?> findClass(String name) throws ClassNotFoundException {  
 throw new ClassNotFoundException(name);  
}

在调用loadClass无法加载类后，默认会调用findClass加载类

## 2.3运行时帧栈结构

Jvm以方法作为最基本的执行单元，帧栈是用于支持jvm进行方法调用和方法执行背后的数据元素。帧栈存储了方法的局部变量表、操作数栈、动态连接和方法返回地址等信息。每一个方法从调用开始到执行结束，都对应着一个帧栈在虚拟机栈中从入栈到出栈



### 2.3.1局部变量表(Local Variables Table)

局部变量表用于存放方法的参数和方法内部定义的局部变量。局部变量表的容量以变量槽为最小单位，JVM并未规定变量槽的大小，只要求每个变量槽都应该能存放一个boolean、byte、char、short、int、float、reference(引用类型)或returnAddress(几乎已不用了)类型的数据。这意味着JVM允许变量槽的长度随着处理器、操作系统或虚拟机实现不同而发生变化

Reference引用类型表示对一个堆中实例对象的引用，JVM并未规定其长度，也没有说明它应该有怎样的结构，但一般来说，jvm至少要能通过reference类型做到两件事情：

1. **根据reference直接或间接地查找到对象在java堆中的数据起始地址或索引**
2. **根据reference直接或间接地查找到对象的数据类型在方法区中存储的类型信息(Class信息)**

当一个方法被调用时，jvm虚拟机会使用局部变量表来完成从参数值到参数列表的传递过程，即实参传递给形参。为了节省帧栈占用的内存，局部变量表中的变量槽是可以重用的，当某个方法体中定义的变量已超出其作用域范围，那么该变量的变量槽就可以交给其他变量来使用。这个行为还会影响GC，eg：

public static void main(String[] args) {  
 {  
 byte[] placeholder = new byte[64 \* 1024 \* 1024];  
 }  
 //即使已经离开了placeholder的作用域，其占用的空间仍不会释放  
 System.*gc*();  
}

这似乎与我们上面所说的变量槽可复用相悖，但仔细分析，可以发现其中的奥秘，这是因为自离开placeholder的作用域后，局部变量表仍未发生过读写操作，其内容还未被其他变量覆盖，所以作为GC Roots一部分的局部变量表还保持着对它的关联。因此只有当离开作用域后局部变量表第一次发生读写后，下一次GC时placeholder的空间才会被回收。在某些书籍中建议”把不使用的对象设置为null”并非一个无意义的操作。虽然大多数情况下不建议这么做，但是在某些极端场景下，这么做还是有用的。

### 2.3.2操作数栈(Operand Stack)

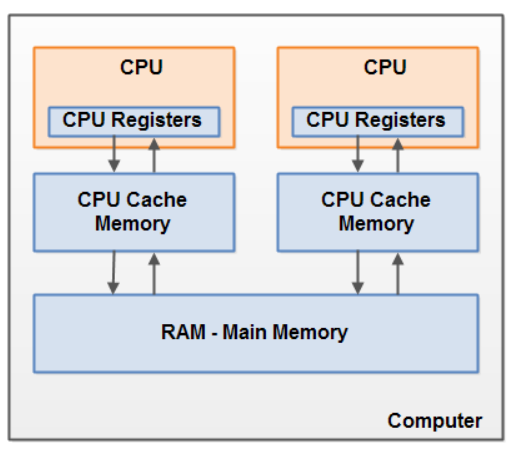
又称操作栈，它是一个后入先出(LIFO)的栈。当一个方法刚刚开始执行时，这个操作栈是空的。

# 3.JVM内存模型与并发

### 3.1 java内存模型JMM(Java Memory Model)

CPU运算速度和内存的读写速度之间存在巨大的鸿沟，所以现代计算机都不得不在二者之间加入一层或多层高速缓存，将运算数据复制到高速缓存中，待运算结束后同步回主内存。这很好地解决了CPU运算速度和内存读写速度之间的矛盾，但它引入了一个新的问题：**缓存一致性问题**

现代CPU往往通过增加核心数来横向扩展运算性能，每个核心都拥有自己的高速缓存，而他们又共享同一主内存，当多个核心的运算任务都涉及同一主内存区域的数据时，将可能导致各自的缓存数据不一致



以上情况因不同硬件和平台各有不同，JAVA虚拟机规范定义了一种JAVA内存模型(JMM)，试图屏蔽各硬件平台之间的差异，让Java程序在不同的环境下都能做到一致的内存访问效果。Java内存模型JMM控制了线程间的通信，它决定一个线程对共享变量的写入何时对另一个线程可见。它是对CPU、高速缓存等硬件的一个抽象。

JMM定义了主内存和工作内存的概念：

1. **主内存：**被多个线程之间共享的内存，多个线程之间的**共享变量**存储在主内存中。主要对应JVM堆中的对象实例数据部分，在JAVA中，**所以类的实例字段、静态字段和数组都存储在堆内存中**。它是上文硬件主内存的一个抽象
2. **工作内存：**又叫本地内存，它存储了多个**共享变量的副本**，这些副本不会被线程共享。主要对应虚拟机栈的局部变量表。**局部变量、方法定义的参数都存储在虚拟机栈中**。它是寄存器、高速缓存等硬件的抽象

### 3.2 主内存和工作内存的交互操作

Java内存模型定义了8种原子性操作：

1. **Lock锁定**  作用于主内存变量，将变量标志为一条线程所独占。该操作执行后会清空工作内存中此变量的值
2. **Unlock解锁**  作用于主内存变量，将处于锁定的变量释放出来。对一个变量执行unload之前，必须执行store和write操作将其同步回主存
3. **Read读取**  作用于主内存变量，它将一个变量的值从主内存传输到线程的工作内存中
4. **Load载入**  作用于工作内存变量，它把从主内存读取的变量值放入工作内存的副本中
5. **Use使用**  作用于工作内存变量，将工作内存变量值传递给执行引擎
6. **Assgin赋值**  作用于工作内存变量，将执行引擎的值传递给工作内存的变量
7. **Store存储** 作用于工作内存变量，它把工作内存变量传递到主内存中
8. **Write写入** 作用于主内存变量，把Store操作从工作内存得到的变量值放入主内存变量中

如果要把变量存主内存拷贝到工作内存，那么按顺序执行Read和load操作。

如果要把变量从工作内存同步回主内存，那么按顺序执行store和write操作。

这里只需要按顺序执行即可，并未要求严格的连续执行

### 3.3 voltile关键字

Volatile关键字是JVM提供的最轻量级的同步机制，当一个变量被声明为voltiole时，它将具备两个特征：

1. **保证此变量对所有线程的可见性。**这里的可见性指当一条线程修改了这个变量的值，新值对于其他线程可以立即得知。普通变量做不到这一点，普通变量的值在线程中传递需要通过主内存作为媒介。而voltile变量要求每个工作内存在使用其之前都必须从主内存刷新到最新的值，保证可以立即看到其他线程对该值的修改，并且要求每个工作内存在修改值之后，必须立即同步回主存，使其他线程能及时看到自己的修改
2. **禁止重排列优化。**编译器会对没有数据依赖的语句进行重排序优化，而voltile会禁止重排序，eg：

public void mySort() {  
 int x = 11;//语句1  
 int y = 12;//语句2  
 x = x + 5; //语句3  
 y = x \* x; //语句4  
}

你可能觉得上述语句的执行顺序为1234，但是由于重排序的存在，他可能会被优化为2134或1324等，但是由于数据依赖的存在，绝不会出现3在1之前，4在2、3之前。在单线程时，执行重排不会出现问题，但在多线程的环境下，指令重排可能会出现让人莫名的执行结果