走进JVM

> ## 本节目标
>
> 1. JVM简介
> 2. Java内存区域与内存溢出异常
> 3. 垃圾回收器与内存分配策略
> 4. 常用JVM性能监控与故障处理工具
> 5. Java内存模型

1. JVM简介

1.1 JVM概念

1).虚拟机简介：

JVM(Java Virtual Machine的简称。意为Java虚拟机。)：

虚拟机：

指通过软件模拟的具有完整硬件功能的、运行在一个完全隔离的环境中的完整计算机系统。 常见的虚拟机：

JVM、VMwave、Virtual Box

2).JVM和其他两个虚拟机的区别： a.VMwave与VirtualBox是通过软件模拟物理CPU的指令集,物理系统中会有很多的寄存器

b.JVM则是通过软件模拟Java字节码的指令集,JVM中只是主要保留了PC寄存器，其他的寄存器都进行了裁剪

JVM是一台被定制过的现实当中不存在的计算机。

1.2 Java和JVM发展简史

1).20世纪Java发展：

1996年SUN JDK 1.0时发布：Classic VM 纯解释运行，使用外挂进行JIT(编译器)

1997年JDK 1.1发布：

AWT、内部类、JDBC、RMI、反射(Java的核心)

RMI:远程方法调用(Remote Method Invocation)。能够让在某个java虚拟机上的对象像调用本地对象一样调用另一个java 虚拟机中的对象上的方法。

1998年JDK 1.2发布：Solary Exact VM(仅存在了很短的时间)

JIT和解释器混合执行 Accurate Memory Management 精确内存管理，数据类型敏感 提升GC性能

JDK1.2开始成为Java2(J2SE,J2EE,J2ME出现),并且加入了Swing Collection。

2).二零零几年：

2000年JDK1.3：HotSpot作为默认虚拟机发布

2002年JDK1.4：Classic VM退出历史舞台

1.4更新内容：Assert，正则表达式，NIO，IPV6，日志API，加密类库，异常链，XML解析器等。

2004年JDK1.5即JDK5,Java5（很重要的一个版本）

Java5更新内容：泛型，注解，装箱，枚举，可变长参数，Foreach循环。

虚拟机层面：改进了Java内存模型(JMM)，提供了JUC并发包。

JDK1.6 java6：

更新内容：脚本编程的支持(动态语言支持)，JDBC4.0，Java编译器API，微型Http服务器API等。

虚拟机层面：锁与同步，垃圾收集，类加载等算法的改动

3).二零一几年：

2011年JDK1.7/Java7发布： G1收集器(Update4才正式发布) 加强对非Java语言的调用支持 升级类加载器架构 64位系统压缩指针 NIO2.0

2014年Java8发布： Lamda表达式 语法增强 Java类型注释等

4).Java与JVM发展历史中的大事件：

使用最广泛的JVM为HotSpot HotSpot最早为Longview Technologies开发，被SUN收购

2006年，Java开源，并建立OpenJDK HotSpot成为SUN JDK和OpenJDK中所带的虚拟机

2008年，Oracle收购BEA 得到JRockit VM

2010年Oracle收购SUN 得到HotSpot

Orcale宣布在JDK8时整合HotSpot和JRockit VM，优势互补 在HotSpot的基础上移植JRockit的优秀特性

2. Java内存区域与内存溢出异常

2.1 运行时数据区域

JVM会在执行Java程序的过程中把它管理的内存划分为若干个不同的数据区域。这些数据区域各有各的用处，各有各的创建与销毁时间，有的区域随着JVM进程的启动而存在，有的区域则依赖用户线程的启动和结束而创建与销毁。一般来说，JVM所管理的内存将会包含以下几个运行时数据区域

线程私有区域:程序计数器、Java虚拟机栈、本地方法栈

线程共享区域:Java堆、方法区、运行时常量池

2.1.1 程序计数器(线程私有)

程序计数器是一块比较小的内存空间，可以看做是当前线程所执行的字节码的行号指示器。

如果当前线程正在执行的是一个Java方法，这个计数器记录的是正在执行的虚拟机字节码指令的地址；如果正在执行的是一个Native方法，这个计数器值为空。

程序计数器内存区域是唯一一个在VM规范中没有规定任何OOM情况的区域！

什么是线程私有?

由于JVM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，因此在任何一个确定的时刻，一个处理器(多核处理器则指的是一个内核)都只会执行一条线程中的指令。因此为了切换线程后能恢复到正确的执行位置，每条线程都需要独立的程序计数器，各条线程之间计数器互不影响，独立存储。我们就把类似这类区域称之为"线程私有"的内存。

2.1.2 Java虚拟机栈(线程私有)

虚拟机栈描述的是Java方法执行的内存模型 : 每个方法执行的同时都会创建一个栈帧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，就对应一个栈帧在虚拟机栈中入栈和出栈的过程。声明周期与线程相同。

之前我们一直讲的栈区域实际上就是此处的虚拟机栈，再详细一点，是虚拟机栈中的局部变量表部分。

局部变量表 : 存放了编译器可知的各种基本数据类型(8大基本数据类型)、对象引用。局部变量表所需的内存空间在编译期间完成分配，当进入一个方法时，这个方法需要在帧中分配多大的局部变量空间是完全确定的，在执行期间不会改变局部变量表大小。

此区域一共会产生以下两种异常:

1. 如果线程请求的栈深度大于虚拟机所允许的深度(-Xss设置栈容量)，将会抛出StackOverFlowError异常。
2. 虚拟机在动态扩展时无法申请到足够的内存，会抛出OOM(OutOfMemoryError)异常

2.1.3 本地方法栈(线程私有)

本地方法栈与虚拟机栈的作用完全一样，他俩的区别无非是本地方法栈为虚拟机使用的Native方法服务，而虚拟机栈为JVM执行的Java方法服务。

在HotSpot虚拟机中，本地方法栈与虚拟机栈是同一块内存区域。

2.1.4 Java堆(线程共享)

Java堆(Java Heap)是JVM所管理的最大内存区域。Java堆是所有线程共享的一块区域，在JVM启动时创建。此内存区域存放的都是对象实例。JVM规范中说到:"**所有的对象实例以及数组都要在堆上分配**"。

Java堆是垃圾回收器管理的主要区域，因此很多时候可以称之为"GC堆"。根据JVM规范规定的内容，Java堆可以处于物理上不连续的内存空间中。Java堆在主流的虚拟机中都是可扩展的(-Xmx设置最大值,-Xms设置最小值)。

如果在堆中没有足够的内存完成实例分配并且堆也无法再拓展时，将会抛出OOM

2.1.5 方法区(线程共享)

方法区与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。在JDK8以前的HotSpot虚拟机中，方法区也被称为"永久代"(JDK8已经被元空间取代)。

永久代并不意味着数据进入方法区就永久存在，此区域的内存回收主要是针对常量池的回收以及对类型的卸载。

JVM规范规定：当方法区无法满足内存分配需求时，将抛出OOM异常。

2.1.6 运行时常量池(方法区的一部分)

运行时常量池是方法区的一部分，存放字面量与符号引用。

字面量 : 字符串(JDK1.7后移动到堆中) 、final常量、基本数据类型的值。

符号引用 : 类和结构的完全限定名、字段的名称和描述符、方法的名称和描述符。

2.2 Java堆溢出

Java堆用于存储对象实例，只要不断的创建对象，并且保证GC Roots到对象之间有可达路径来避免来GC清除这些对象，那么在对象数量达到最大堆容量后就会产生内存溢出异常。

上节中已经讲到了，可以设置JVM参数-Xms:设置堆的最小值、-Xmx:设置堆最大值。下面我们来看一个Java堆OOM的测试

范例：观察Java Heap OOM

```
/**
 * JVM参数为:-Xmx20m -Xms20m -XX:+HeapDumpOnOutOfMemoryError
 * @author 38134
 *
 */
public class Test {
    static class OOMObject {

    }
    public static void main(String[] args) {
        List<OOMObject> list =
                new ArrayList<>();
        while(true) {
            list.add(new OOMObject());
        }
    }
}
```

Java堆内存的OOM异常是实际应用中最常见的内存溢出情况。当出现Java堆内存溢出时，异常堆栈信息"java.lang.OutOfMemoryError"会进一步提示"Java heap space"。当出现"Java heap space"则很明确的告知我们，OOM发生在堆上。

此时要对Dump出来的文件进行分析，以MAT为例。分析问题的产生到底是出现了内存泄漏(Memory Leak)还是内存溢出(Memory Overflow)

内存泄漏 : 泄漏对象无法被GC

内存溢出 : 内存对象确实还应该存活。此时要根据JVM堆参数与物理内存相比较检查是否还应该把JVM堆内存调大；或者检查对象的生命周期是否过长。

以上是我们处理Java堆内存的简单方法，处理具体这类问题需要的工具以及知识我们放到下面第四小节具体来说。

2.3 虚拟机栈和本地方法栈溢出

由于我们HotSpot虚拟机将虚拟机栈与本地方法栈合二为一，因此对于HotSpot来说，栈容量只需要由-Xss参数来设置。

关于虚拟机栈会产生的两种异常:

- 如果线程请求的栈深度大于虚拟机所允许的最大深度，会抛出StackOverFlow异常
- 如果虚拟机在拓展栈时无法申请到足够的内存空间，则会抛出OOM异常

范例:观察StackOverFlow异常(单线程环境下)

```
/**
 * JVM参数为:-Xss128k
 * @author 38134
 *
 */
public class Test {
    private int stackLength = 1;
    public void stackLeak() {
        stackLength++;
        stackLeak();
    }
    public static void main(String[] args) {
        Test test = new Test();
        try {
            test.stackLeak();
        } catch (Throwable e) {
            System.out.println("Stack Length: "+test.stackLength);
            throw e;
        }
    }
}
```

出现StackOverflowError异常时有错误堆栈可以阅读，比较好找到问题所在。如果使用虚拟机默认参数，栈深度在多多数情况下达到1000-2000完全没问题，对于正常的方法调用（包括递归），完全够用。

如果是因为多线程导致的内存溢出问题，在不能减少线程数的情况下，只能减少最大堆和减少栈容量的方式来换取更多线程。

范例：观察多线程下的内存溢出异常

```
/**
 * JVM参数为:-Xss2M
 * @author 38134
 *
 */
public class Test {

    private void dontStop() {
        while(true) {

        }
    }
    public void stackLeakByThread() {
        while(true) {
            Thread thread = new Thread(new Runnable() {
                @Override
                public void run() {
                    dontStop();
                }
            });
            thread.start();
        }
    }
```

```
    public static void main(String[] args) {
        Test test = new Test();
        test.stackLeakByThread();
    }
}
```

以上代码运行需谨慎。先记得保存手头所有工作。

3. 垃圾回收器与内存分配策略

上一节讲了Java运行时内存的各个区域。对于程序计数器、虚拟机栈、本地方法栈这三部分区域而言，其生命周期与相关线程有关，随线程而生，随线程而灭。并且这三个区域的内存分配与回收具有确定性，因为当方法结束或者线程结束时，内存就自然跟着线程回收了。因此我们本节课所讲的有关内存分配和回收关注的为Java堆与方法区这两个区域。

3.1 如何判断对象已"死"

Java堆中存放着几乎所有的对象实例，垃圾回收器在对堆进行垃圾回收前，首先要判断这些对象哪些还存活，哪些已经"死去"。判断对象是否已"死"有如下几种算法

3.1.1 引用计数法

引用计数描述的算法为:

给对象增加一个引用计数器，每当有一个地方引用它时，计数器就+1；当引用失效时，计数器就-1；任何时刻计数器为0的对象就是不能再被使用的，即对象已"死"。

引用计数法实现简单，判定效率也比较高，在大部分情况下都是一个不错的算法。比如Python语言就采用引用计数法进行内存管理。

但是，在主流的JVM中没有选用引用计数法来管理内存，最主要的原因就是引用计数法无法解决对象的循环引用问题

范例：观察循环引用问题

```
/**
 * JVM参数 :-XX:+PrintGC
 * @author 38134
 *
 */
public class Test {
    public Object instance = null;
    private static int _1MB = 1024 * 1024;
    private byte[] bigSize = new byte[2 * _1MB];
    public static void testGC() {
        Test test1 = new Test();
        Test test2 = new Test();
        test1.instance = test2;
        test2.instance = test1;
        test1 = null;
        test2 = null;
        // 强制jvm进行垃圾回收
```

```
        System.gc();
    }
    public static void main(String[] args) {
        testGC();
    }
}
```

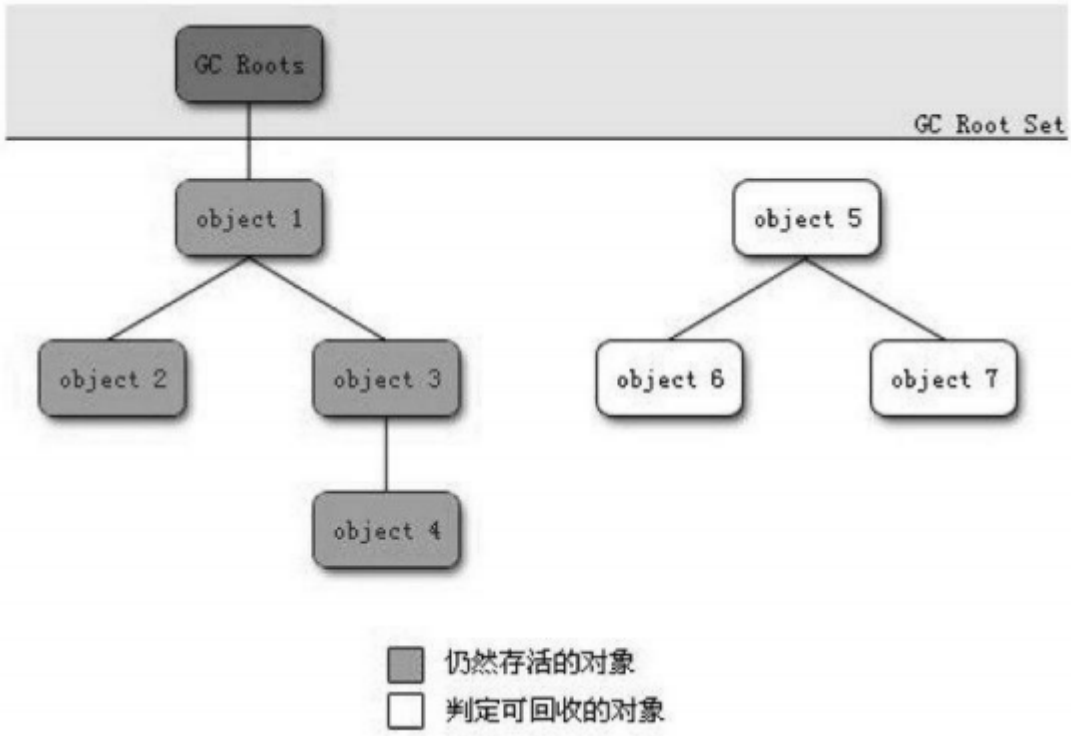
```
[GC (System.gc())  6092K->856K(125952K), 0.0007504 secs]
```

从结果可以看出，GC日志包含" 6092K->856K(125952K)"，意味着虚拟机并没有因为这两个对象互相引用就不回收他们。即VM并不使用引用计数法来判断对象是否存活。

3.1.2 可达性分析算法

在上面我们讲了，Java并不采用引用计数法来判断对象是否已"死"，而采用"可达性分析"来判断对象是否存活(同样采用此法的还有C#、Lisp-最早的一门采用动态内存分配的语言)。

此算法的核心思想为 : 通过一系列称为"GC Roots"的对象作为起始点，从这些节点开始向下搜索，搜索走过的路径称之为"引用链"，当一个对象到GC Roots没有任何的引用链相连时(从GC Roots到这个对象不可达)时，证明此对象是不可用的。以下图为例:



对象Object5-Object7之间虽然彼此还有关联，但是它们到GC Roots是不可达的，因此他们会被判定为可回收对象。

在Java语言中，可作为GC Roots的对象包含下面几种:

1. 虚拟机栈(栈帧中的本地变量表)中引用的对象
2. 方法区中类静态属性引用的对象
3. 方法区中常量引用的对象
4. 本地方法栈中JNI(Native方法)引用的对象

在JDK1.2以前，Java中引用的定义很传统：如果引用类型的数据中存储的数值代表的是另一块内存的起始地址，就称这块内存代表着一个引用。这种定义有些狭隘，一个对象在这种定义下只有被引用或者没有被引用两种状态。

我们希望能描述这一类对象：当内存空间还足够时，则能保存在内存中；如果内存空间在进行垃圾回收后还是非常紧张，则可以抛弃这些对象。很多系统中的缓存对象都符合这样的场景。

在JDK1.2之后，Java对引用的概念做了扩充，将引用分为强引用(Strong Reference)、软引用(Soft Reference)、弱引用(Weak Reference)和虚引用(Phantom Reference)四种，这四种引用的强度依次递减。

> 1. 强引用：强引用指的是在程序代码之中普遍存在的，类似于"Object obj = new Object()"这类的引用，只要强引用还存在，垃圾回收器永远不会回收掉被引用的对象实例。
> 2. 软引用：软引用是用来描述一些还有用但是不是必须的对象。对于软引用关联着的对象，在系统将要发生内存溢出之前，会把这些对象列入回收范围之中进行第二次回收。如果这次回收还是没有足够的内存，才会抛出内存溢出异常。在JDK1.2之后，提供了SoftReference类来实现软引用。
> 3. 弱引用：弱引用也是用来描述非必需对象的。但是它的强度要弱于软引用。被弱引用关联的对象只能生存到下一次垃圾回收发生之前。当垃圾回收器开始进行工作时，无论当前内容是否够用，都会回收掉只被弱引用关联的对象。在JDK1.2之后提供了WeakReference类来实现弱引用。
> 4. 虚引用：虚引用也被称为幽灵引用或者幻影引用，它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。为一个对象设置虚引用的唯一目的就是能在这个对象被收集器回收时收到一个系统通知。在JDK1.2之后，提供了PhantomReference类来实现虚引用。

生存还是死亡?

即使在可达性分析算法中不可达的对象，也并非"非死不可"的，这时候他们暂时处在"缓刑"阶段。要宣告一个对象的真正死亡，至少要经历两次标记过程：如果对象在进行可达性分析之后发现没有与GC Roots相连接的引用链，那它将会被第一次标记并且进行一次筛选，筛选的条件是此对象是否有必要执行finalize()方法。当对象没有覆盖finalize()方法或者finalize()方法已经被JVM调用过，虚拟机会将这两种情况都视为"没有必要执行"，此时的对象才是真正"死"的对象。

如果这个对象被判定为有必要执行finalize()方法，那么这个对象将会被放置在一个叫做F-Queue的队列之中，并在稍后由一个虚拟机自动建立的、低优先级的Finalizer线程去执行它（这里所说的执行指的是虚拟机会触发finalize()方法）。finalize()方法是对象逃脱死亡的最后一次机会，稍后GC将对F-Queue中的对象进行第二次小规模标记，如果对象在finalize()中成功拯救自己(只需要重新与引用链上的任何一个对象建立起关联关系即可)，那在第二次标记时它将会被移除出"即将回收"的集合；如果对象这时候还是没有逃脱，那基本上它就是真的被回收了。

范例:对象自我拯救

```
public class Test {
    public static Test test;
    public void isAlive() {
        System.out.println("I am alive :)");
    }
    @Override
    protected void finalize() throws Throwable {
        super.finalize();
        System.out.println("finalize method executed!");
        test = this;
    }
    public static void main(String[] args)throws Exception {
        test = new Test();
        test = null;
        System.gc();
```

```
            Thread.sleep(500);
            if (test != null) {
                test.isAlive();
            }else {
                System.out.println("no,I am dead :(");
            }
            // 下面代码与上面完全一致，但是此次自救失败
            test = null;
            System.gc();
            Thread.sleep(500);
            if (test != null) {
                test.isAlive();
            }else {
                System.out.println("no,I am dead :(");
            }
        }
    }
```

从上面代码示例我们发现，finalize方法确实被JVM触发，并且对象在被收集前成功逃脱。

但是从结果上我们发现，两个完全一样的代码片段，结果是一次逃脱成功，一次失败。这是因为，**任何一个对象的finalize()方法都只会被系统自动调用一次**，如果相同的对象在逃脱一次后又面临一次回收，它的finalize()方法不会被再次执行，因此第二段代码的自救行动失败。

3.2 回收方法区

方法区(永久代)的垃圾回收主要收集两部分内容 : 废弃常量和无用的类。

回收废弃常量和回收Java堆中的对象十分类似。以常量池中字面量(直接量)的回收为例，假如一个字符串"abc"已经进入了常量池中，但是当前系统没有任何一个String对象引用常量池的"abc"常量，也没有在其他地方引用这个字面量，如果此时发生GC并且有必要的话，这个"abc"常量会被系统清理出常量池。常量池中的其他类(接口)、方法、字段的符号引用也与此类似。

判定一个类是否是"无用类"则相对复杂很多。类需要同时满足下面三个条件才会被算是"无用的类" :

> 1. 该类所有实例都已经被回收(即在Java堆中不存在任何该类的实例)
> 2. 加载该类的ClassLoader已经被回收
> 3. 该类对应的Class对象没有在任何其他地方被引用，无法在任何地方通过反射访问该类的方法

JVM可以对同时满足上述3个条件的无用类进行回收，也仅仅是"可以"而不是必然。在大量使用反射、动态代理等场景都需要JVM具备类卸载的功能来防止永久代的溢出。

3.3 垃圾回收算法

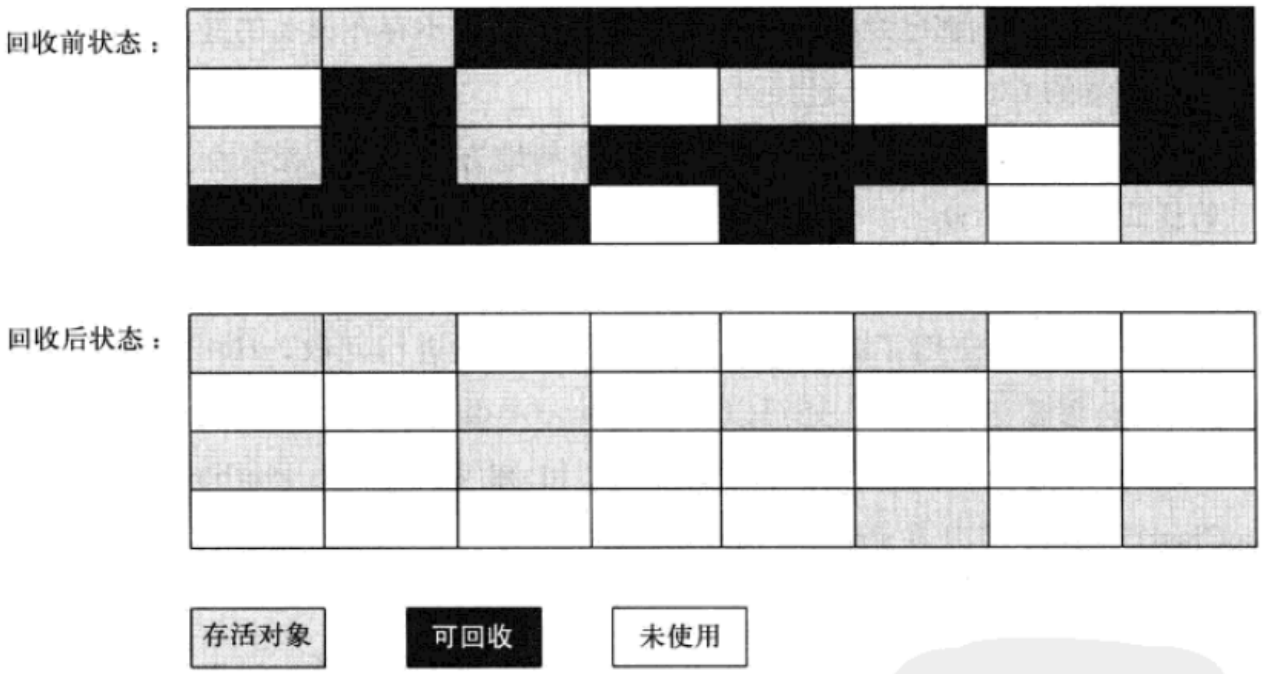
3.3.1 标记-清除算法

"标记-清除"算法是最基础的收集算法。算法分为"标记"和"清除"两个阶段 : 首先标记出所有需要回收的对象，在标记完成后统一回收所有被标记的对象(标记过程见3.1.2章节)。后续的收集算法都是基于这种思路并对其不足加以改进而已。

"标记-清除"算法的不足主要有两个 :

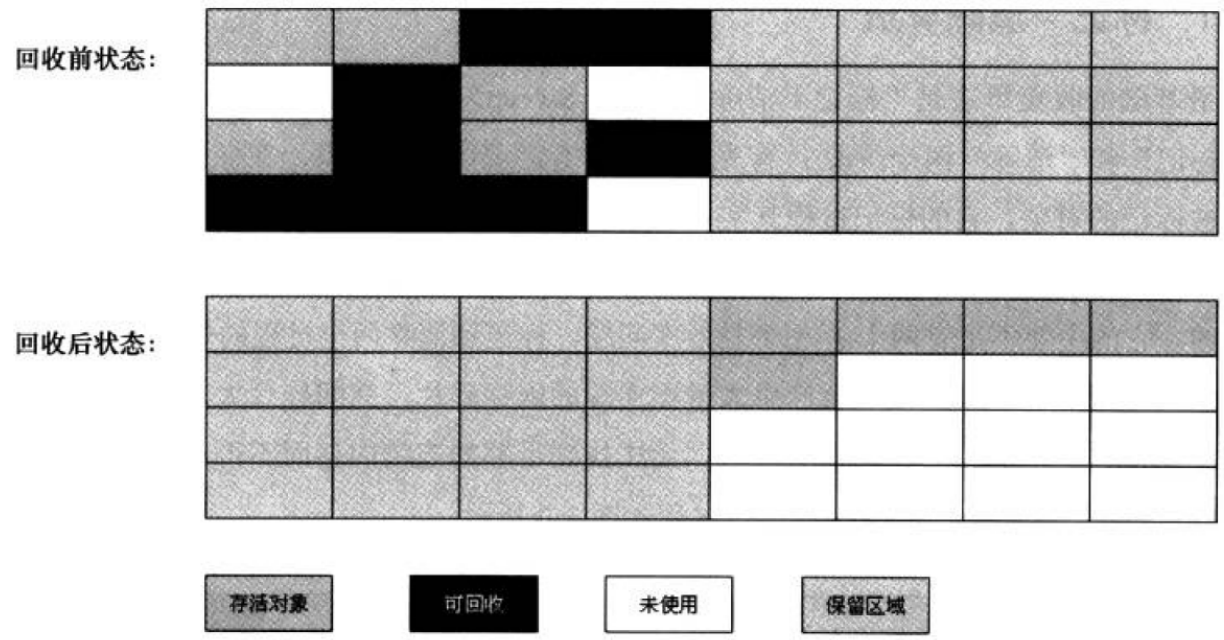
1. 效率问题 : 标记和清除这两个过程的效率都不高

2. 空间问题 : 标记清除后会产生大量不连续的内存碎片，空间碎片太多可能会导致以后在程序运行中需要分配较大对象时，无法找到足够连续内存而不得不提前触发另一次垃圾收集。



3.3.2 复制算法(新生代回收算法)

"复制"算法是为了解决"标记-清理"的效率问题。它将可用内存按容量划分为大小相等的两块，每次只使用其中的一块。当这块内存需要进行垃圾回收时，会将此区域还存活着的对象复制到另一块上面，然后再把已经使用过的内存区域一次清理掉。这样做的好处是每次都是对整个半区进行内存回收，内存分配时也就不需要考虑内存碎片等复杂情况，只需要移动堆顶指针，按顺序分配即可。此算法实现简单，运行高效。算法的执行流程如下图 :



现在的商用虚拟机(包括HotSpot都是采用这种收集算法来回收新生代)

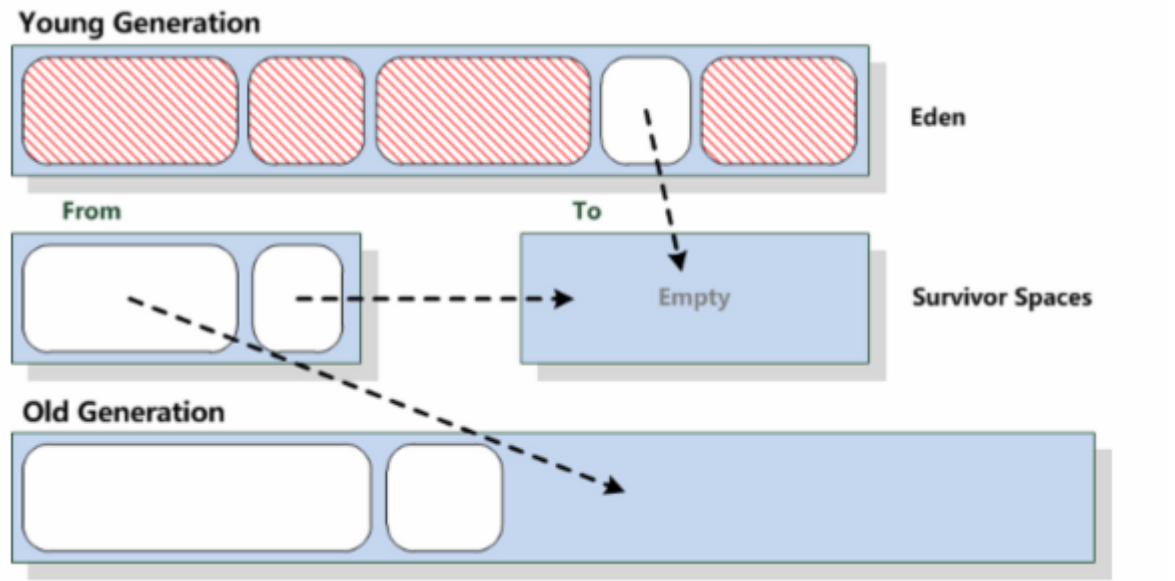
新生代中98%的对象都是"朝生夕死"的，所以并不需要按照1：1的比例来划分内存空间，而是将内存(新生代内存)分为一块较大的Eden(伊甸园)空间和两块较小的Survivor(幸存者)空间，每次使用Eden和其中一块Survivor（两个Survivor区域一个称为From区，另一个称为To区域）。当回收时，将Eden和Survivor中还存活的对象一次性复制到另一块Survivor空间上，最后清理掉Eden和刚才用过的Survivor空间。

当Survivor空间不够用时，需要依赖其他内存(老年代)进行**分配担保。**

HotSpot默认Eden与Survivor的大小比例是8：1，也就是说Eden:Survivor From : Survivor To = 8:1:1。所以每次新生代可用内存空间为整个新生代容量的90%,而剩下的10%用来存放回收后存活的对象。

HotSpot实现的复制算法流程如下:

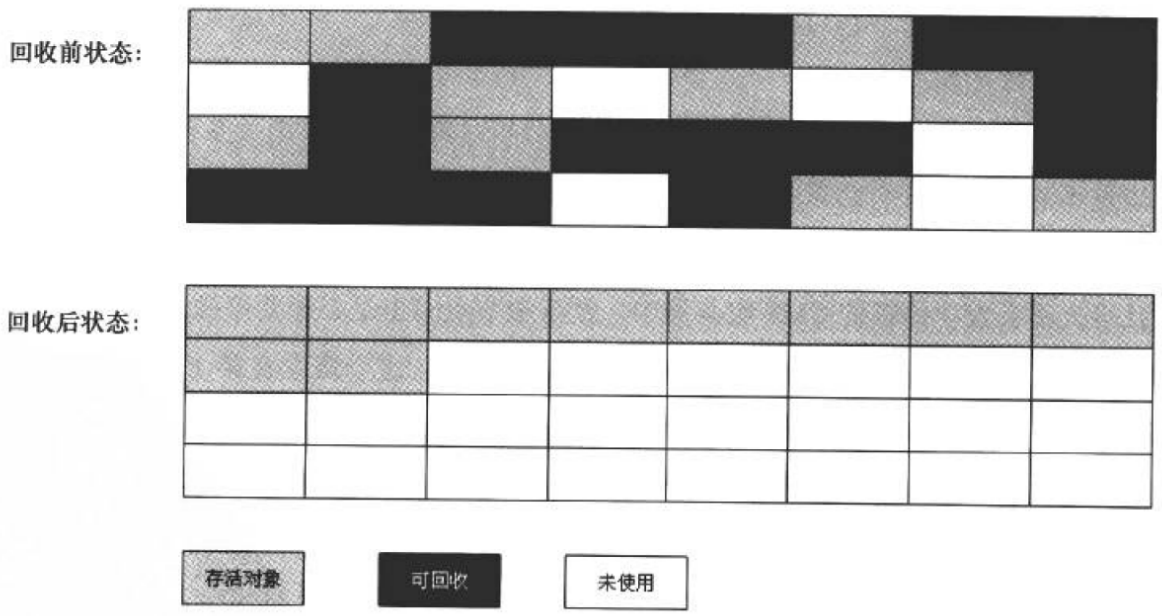
> 1. 当Eden区满的时候,会触发第一次Minor gc,把还活着的对象拷贝到Survivor From区；当Eden区再次触发Minor gc的时候,会扫描Eden区和From区域,对两个区域进行垃圾回收,经过这次回收后还存活的对象,则直接复制到To区域,并将Eden和From区域清空。
> 2. 当后续Eden又发生Minor gc的时候,会对Eden和To区域进行垃圾回收,存活的对象复制到From区域,并将Eden和To区域清空。
> 3. 部分对象会在From和To区域中复制来复制去,如此交换15次(由JVM参数MaxTenuringThreshold决定,这个参数默认是15),最终如果还是存活,就存入到老年代



3.3.3 标记-整理算法(老年代回收算法)

复制收集算法在对象存活率较高时会进行比较多的复制操作，效率会变低。因此在老年代一般不能使用复制算法。

针对老年代的特点，提出了一种称之为"标记-整理算法"。标记过程仍与"标记-清除"过程一致，但后续步骤不是直接对可回收对象进行清理，而是让所有存活对象都向一端移动，然后直接清理掉端边界以外的内存。流程图如下:



3.3.4 分代收集算法

当前JVM垃圾收集都采用的是"分代收集(Generational Collection)"算法，这个算法并没有新思想，只是根据对象存活周期的不同将内存划分为几块。

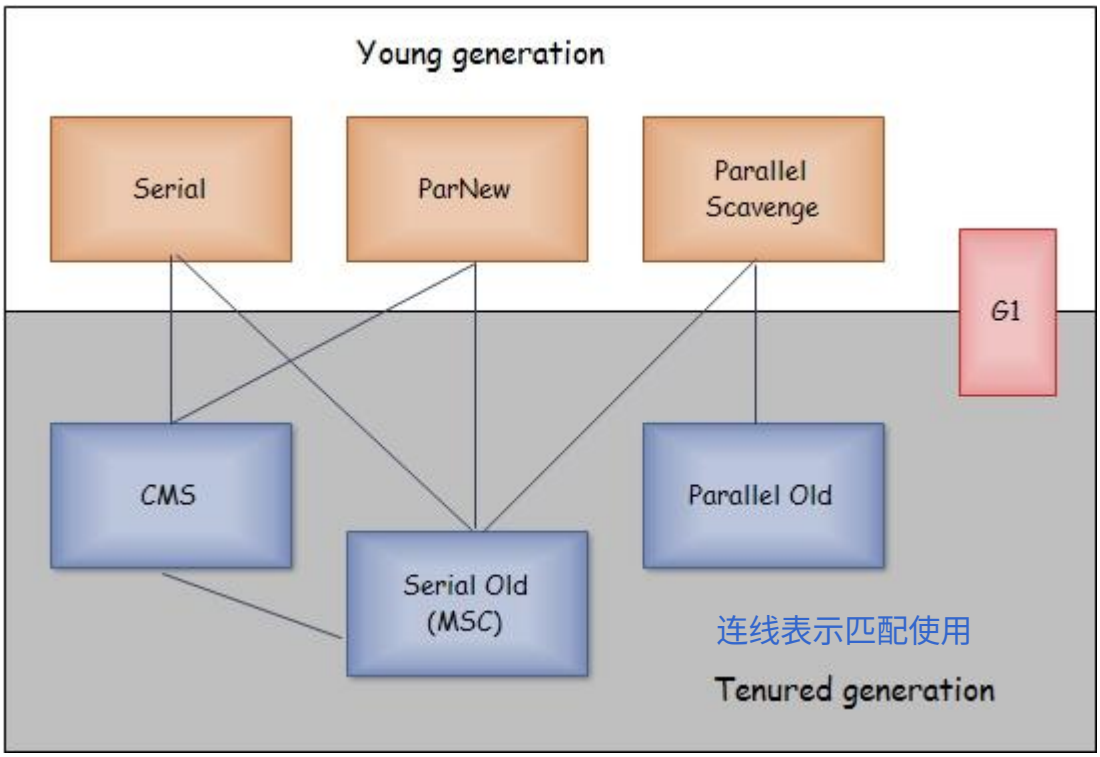
一般是把Java堆分为新生代和老年代。在新生代中，每次垃圾回收都有大批对象死去，只有少量存活，因此我们采用复制算法；而老年代中对象存活率高、没有额外空间对它进行分配担保，就必须采用"标记-清理"或者"标记-整理"算法。

面试题 : 请问了解Minor GC和Full GC么，这两种GC有什么不一样吗

> 1. Minor GC又称为新生代GC : 指的是发生在新生代的垃圾收集。因为Java对象大多都具备朝生夕灭的特性，因此Minor GC(采用复制算法)非常频繁，一般回收速度也比较快。
> 2. Full GC 又称为 老年代GC或者Major GC : 指发生在老年代的垃圾收集。出现了Major GC，经常会伴随至少一次的Minor GC(并非绝对，在Parallel Scavenge收集器中就有直接进行Full GC的策略选择过程)。Major GC的速度一般会比Minor GC慢10倍以上。

3.4 垃圾收集器 我们的垃圾回收器都有那些

如果说上面我们讲的收集算法是内存回收的方法论，那么垃圾收集器就是内存回收的具体实现。以下讲的收集器基于JDK1.7的G1收集器之后的HotSpot虚拟机，这个JVM包含的所有收集器如下图所示:



上图展示了7种作用于不同分代的收集器，如果两个收集器之间存在连线，就说明他们之间可以搭配使用。所处的区域，表示它是属于新生代收集器还是老年代收集器。在讲具体的收集器之前我们先来明确三个概念:

这两个反了

- 并行(Parallel) : 指多条垃圾收集线程并行工作，用户线程仍处于等待状态 在多核cpu上
- 并发(Concurrent) : 指用户线程与垃圾收集线程同时执行(不一定并行，可能会交替执行)，用户程序继续运行，而垃圾收集程序在另外一个CPU上。
- 吞吐量:就是CPU用于运行用户代码的时间与CPU总消耗时间的比值。

$$吞吐量 = 运行用户代码时间 / (运行用户代码时间 + 垃圾收集时间)$$

衡量你的代码处理能力

例如：虚拟机总共运行了100分钟，其中垃圾收集花掉1分钟，那吞吐量就是99%。

3.4.1 Serial收集器(新生代收集器,串行GC)

单线程GC

Serial收集器是最基本、发展历史最悠久的收集器，曾经（在JDK 1.3.1之前）是虚拟机新生代收集的唯一选择。

- **特性:**

  这个收集器是一个单线程的收集器，但它的“单线程”的意义并不仅仅说明它只会使用一个CPU或一条收集线程去完成垃圾收集工作，更重要的是在它进行垃圾收集时，必须暂停其他所有的工作线程，直到它收集结束(Stop The World).

- **应用场景:**

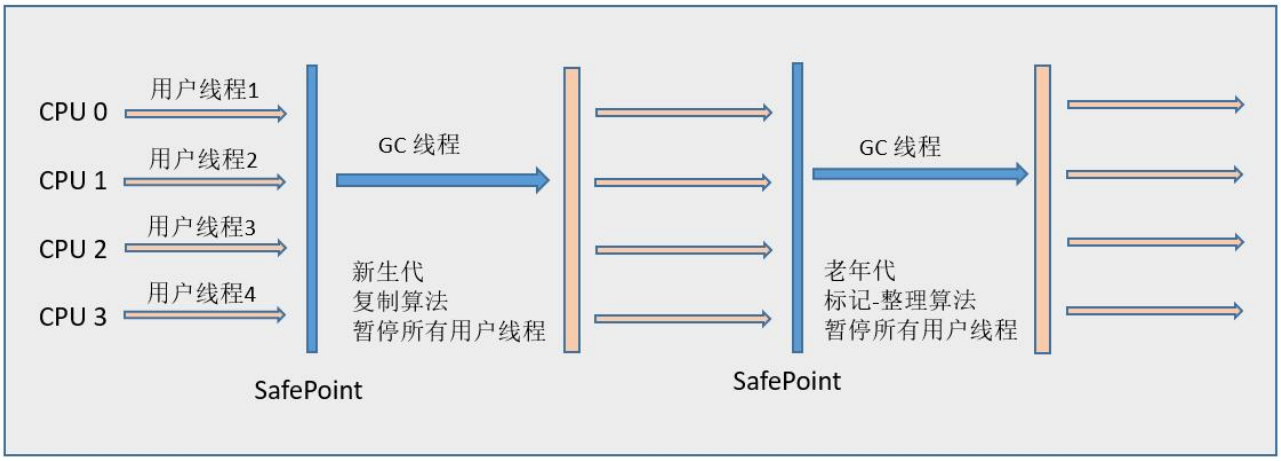
我们Java可以开发桌面程序

Serial收集器是虚拟机运行在Client模式下的默认新生代收集器。

我们Java编程server模式，jdk1.8之后，没有client模式。

- **优势:**

简单而高效（与其他收集器的单线程比），对于限定单个CPU的环境来说，Serial收集器由于没有线程交互的开销，专心做垃圾收集自然可以获得最高的单线程收集效率。 实际上到现在为止 : 它依然是虚拟机运行在Client模式下的默认新生代收集器



3.4.2 ParNew收集器(新生代收集器,并行GC)

ParNew收集器其实就是Serial收集器的多线程版本，除了使用多条线程进行垃圾收集之外，其余行为包括Serial收集器可用的所有控制参数、收集算法、Stop The World、对象分配规则、回收策略等都与Serial收集器完全一样，在实现上，这两种收集器也共用了相当多的代码。

- **特性：**

Serial收集器的多线程版本

- **应用场景：**

ParNew收集器是许多运行在Server模式下的虚拟机中首选的新生代收集器。

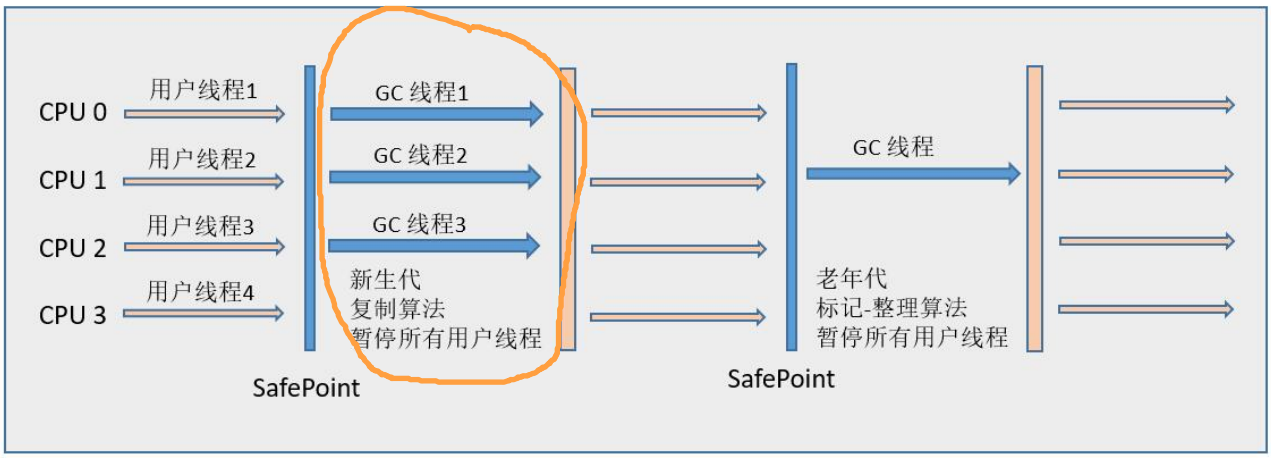
作为Server的首选收集器之中有一个与性能无关的很重要的原因是：除了Serial收集器外，目前只有它能与CMS收集器配合工作。 在JDK 1.5时期，HotSpot推出了一款在强交互应用中几乎可认为有划时代意义的垃圾收集器——CMS收集器，这款收集器是HotSpot虚拟机中第一款真正意义上的并发收集器，它第一次实现了让垃圾收集线程与用户线程同时工作。 不幸的是，CMS作为老年代的收集器，却无法与JDK 1.4.0中已经存在的新生代收集器Parallel Scavenge配合工作，所以在JDK 1.5中使用CMS来收集老年代的时候，新生代只能选择ParNew或者Serial收集器中的一个。

- **对比分析:**

与Serial收集器对比:

ParNew收集器在单CPU的环境中绝对不会有比Serial收集器更好的效果，甚至由于存在线程交互的开销，该收集器在通过超线程技术实现的两个CPU的环境中都不能百分之百地保证可以超越Serial收集器。

然而，随着可以使用的CPU的数量的增加，它对于GC时系统资源的有效利用还是很有好处的。



3.4.3 Parallel Scavenge收集器(新生代收集器,并行GC)

- **特性:**

Parallel Scavenge收集器是一个**新生代收集器**，它也是使用**复制算法**的收集器，又是**并行**的多线程收集器。

Parallel Scavenge收集器使用两个参数控制吞吐量:

```java
```

XX:MaxGCPauseMillis 控制最大的垃圾收集停顿时间 XX:GCRatio 直接设置吞吐量的大小 ```

直观上，只要最大的垃圾收集停顿时间越小，吞吐量是越高的，但是**GC停顿时间的缩短是以牺牲吞吐量和新生代空间作为代价的。**比如原来10秒收集一次，每次停顿100毫秒，现在变成5秒收集一次，每次停顿70毫秒。停顿时间下降的同时，吞吐量也下降了。

- **应用场景:**

停顿时间越短就越适合需要与用户交互的程序，良好的响应速度能提升用户体验，而高吞吐量则可以高效率地利用CPU时间，尽快完成程序的运算任务，主要适合在后台运算而不需要太多交互的任务。

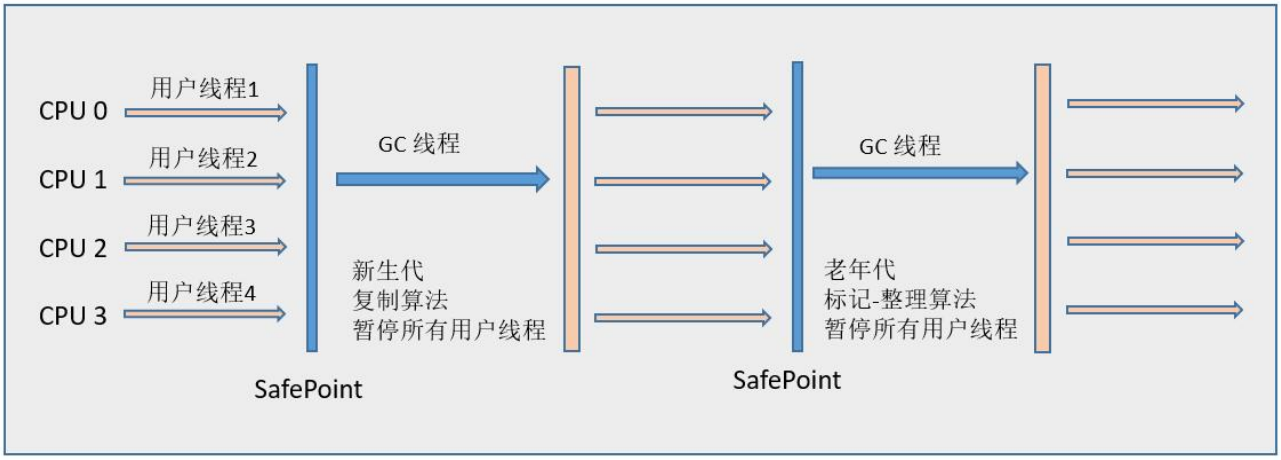
- **对比分析:**
  - **Parallel Scavenge收集器 VS CMS等收集器:** Parallel Scavenge收集器的特点是它的关注点与其他收集器不同，CMS等收集器的关注点是尽可能地缩短垃圾收集时用户线程的停顿时间，而Parallel Scavenge收集器的目标则是达到一个**可控制的吞吐量**（Throughput）。 由于与吞吐量关系密切，Parallel Scavenge收集器也经常称为"吞吐量优先"收集器。
  - **Parallel Scavenge收集器 VS ParNew收集器:** Parallel Scavenge收集器与ParNew收集器的一个重要区别是它具有自适应调节策略。
- **GC自适应的调节策略**:

  Parallel Scavenge收集器有一个参数-`XX:+UseAdaptiveSizePolicy`。当这个参数打开之后，就不需要手工指定新生代的大小、Eden与Survivor区的比例、晋升老年代对象年龄等细节参数了，虚拟机会根据当前系统的运行情况收集性能监控信息，动态调整这些参数以提供最合适的停顿时间或者最大的吞吐量，这种调节方式称为GC自适应的调节策略（GC Ergonomics）。

> 注：工作流程图同ParNew

3.4.4 Serial Old收集器(老年代收集器，串行GC)

- **特性：** Serial Old是Serial收集器的**老年代版本**，它同样是一个**单线程收集器**，使用**标记 - 整理**算法。
- **应用场景：**
  - **Client模式** Serial Old收集器的主要意义也是在于给Client模式下的虚拟机使用。
  - **Server模式** 如果在Server模式下，那么它主要还有两大用途：一种用途是在JDK 1.5以及之前的版本中与Parallel Scavenge收集器搭配使用，另一种用途就是作为CMS收集器的后备预案，在并发收集发生Concurrent Mode Failure时使用。

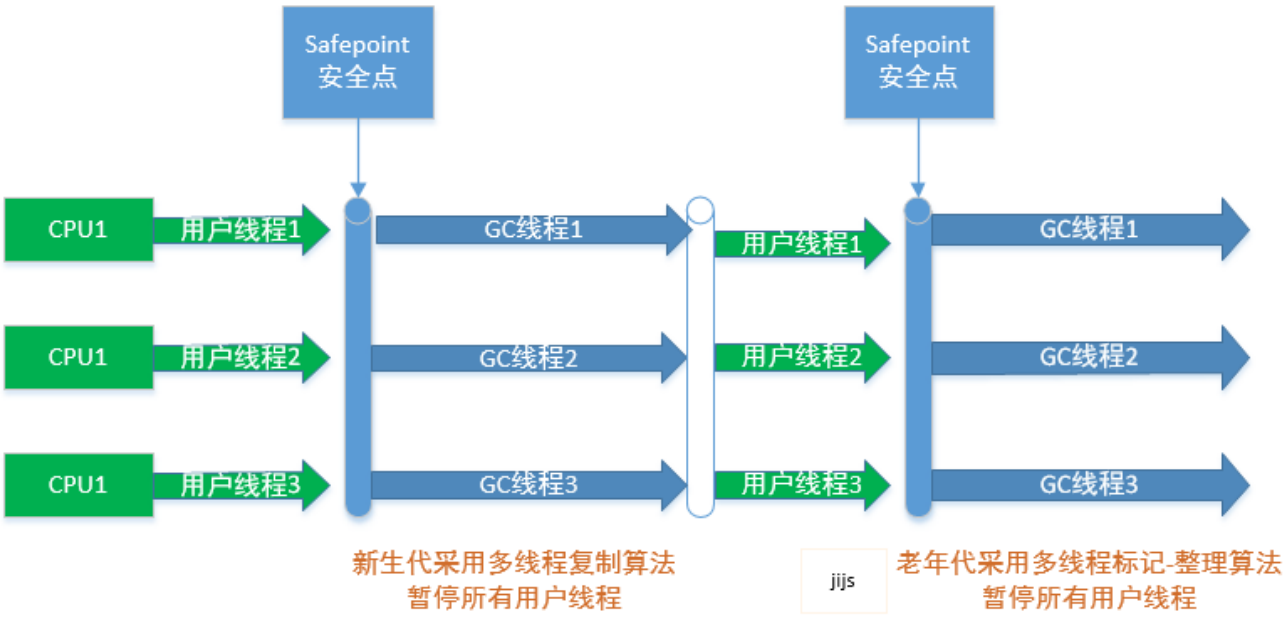


3.4.5 Parallel Old收集器(老年代收集器，并行GC)

- **特性：** Parallel Old是Parallel Scavenge收集器的**老年代版本**，使用**多线程**和**"标记 - 整理"**算法。
- **应用场景：**

在注重吞吐量以及CPU资源敏感的场合，都可以优先考虑Parallel Scavenge加Parallel Old收集器。

这个收集器是在JDK 1.6中才开始提供的，在此之前，新生代的Parallel Scavenge收集器一直处于比较尴尬的状态。原因是，如果新生代选择了Parallel Scavenge收集器，老年代除了Serial Old收集器外别无选择（Parallel Scavenge收集器无法与CMS收集器配合工作）。由于老年代Serial Old收集器在服务端应用性能上的"拖累"，使用了Parallel Scavenge收集器也未必能在整体应用上获得吞吐量最大化的效果，由于单线程的老年代收集中无法充分利用服务器多CPU的处理能力，在老年代很大而且硬件比较高级的环境中，这种组合的吞吐量甚至还不一定有ParNew加CMS的组合"给力"。直到Parallel Old收集器出现后，"吞吐量优先"收集器终于有了比较名副其实的应用组合。



3.4.6 CMS收集器（老年代收集器，并发GC）

- **特性：** CMS（Concurrent Mark Sweep）收集器是一种以获取最短回收停顿时间为目标的收集器。目前很大一部分的Java应用集中在互联网站或者B/S系统的服务端上，这类应用尤其重视服务的响应速度，希望系统停顿时间最短，以给用户带来较好的体验。CMS收集器就非常符合这类应用的需求。

CMS收集器是基于“标记—清除”算法实现的，它的运作过程相对于前面几种收集器来说更复杂一些，整个过程分为4个步骤：

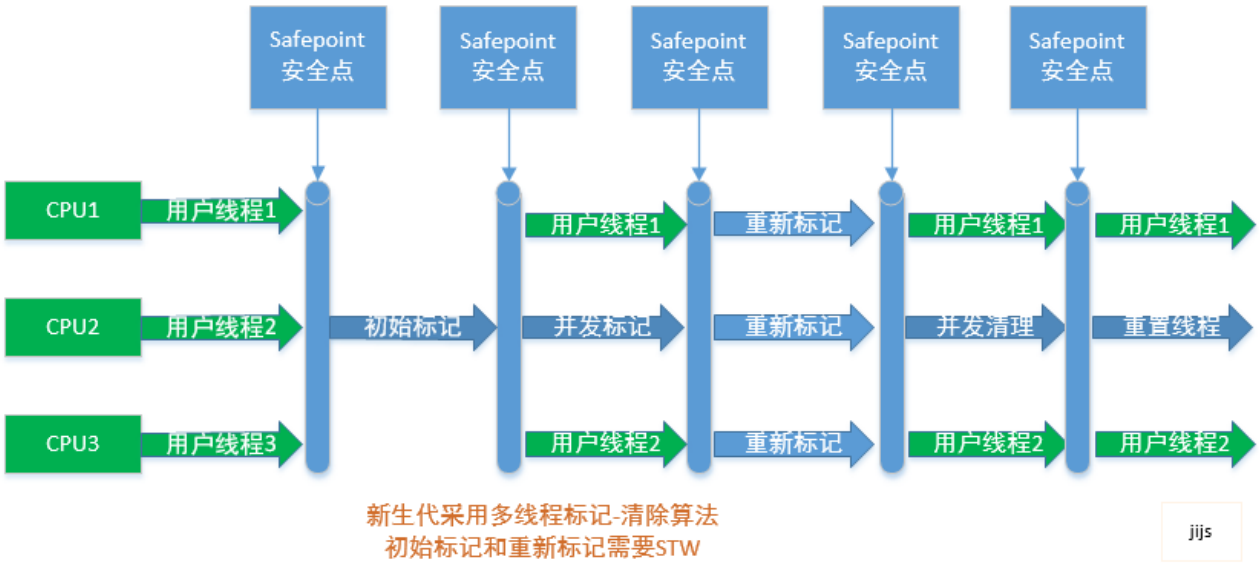
> - 初始标记（CMS initial mark） 初始标记仅仅只是标记一下GC Roots能直接关联到的对象，速度很快，需要“Stop The World”。 不处理任何用户线程
> - 并发标记（CMS concurrent mark） 并发标记阶段就是进行GC Roots Tracing的过程。
> - 重新标记（CMS remark） 重新标记阶段是为了修正并发标记期间因用户程序继续运作而导致标记产生变动的那一部分对象的标记记录，这个阶段的停顿时间一般会比初始标记阶段稍长一些，但远比并发标记的时间短，仍然需要“Stop The World”。
> - 并发清除（CMS concurrent sweep） 并发清除阶段会清除对象。

由于整个过程中耗时最长的并发标记和并发清除过程收集器线程都可以与用户线程一起工作，所以，从总体上来说，CMS收集器的内存回收过程是与用户线程一起并发执行的。

- **优点：** CMS是一款优秀的收集器，它的主要优点在名字上已经体现出来了：**并发收集、低停顿。**
- **缺点：**

> - CMS收集器对CPU资源非常敏感 其实，面向并发设计的程序都对CPU资源比较敏感。在并发阶段，它虽然不会导致用户线程停顿，但是会因为占用了一部分线程（或者说CPU资源）而导致应用程序变慢，总吞吐量会降低。 CMS默认启动的回收线程数是（CPU数量+3）/ 4，也就是当CPU在4个以上时，并发回收时垃圾收集线程不少于25%的CPU资源，并且随着CPU数量的增加而下降。但是当CPU不足4个（譬如2个）时，CMS对用户程序的影响就可能变得很大。
> - CMS收集器无法处理浮动垃圾 CMS收集器无法处理浮动垃圾，可能出现“Concurrent Mode Failure”失败而导致另一次Full GC的产生。由于CMS并发清理阶段用户线程还在运行着，伴随程序运行自然就还会有新的垃圾不断产生，这一部分垃圾出现在标记过程之后，CMS无法在当次收集中处理掉它们，只好留待下一次GC时再清理掉。这一部分垃圾就称为“浮动垃圾”。也是由于在垃圾收集阶段用户线程还需要运行，那也就还需要预留有足够的内存空间给用户线程使用，因此CMS收集器不能像其他收集器那样等到老年代几乎完全被填满了再进行收集，需要预留一部分空间提供并发收集时的程序运作使用。要是CMS

> 运行期间预留的内存无法满足程序需要，就会出现一次“Concurrent Mode Failure”失败，这时虚拟机将启动后备预案：临时启用Serial Old收集器来重新进行老年代的垃圾收集，这样停顿时间就很长了。
- CMS收集器会产生大量空间碎片 CMS是一款基于“标记—清除”算法实现的收集器，这意味着收集结束时会有大量空间碎片产生。空间碎片过多时，将会给大对象分配带来很大麻烦，往往会出现老年代还有很大空间剩余，但是无法找到足够大的连续空间来分配当前对象，不得不提前触发一次Full GC。



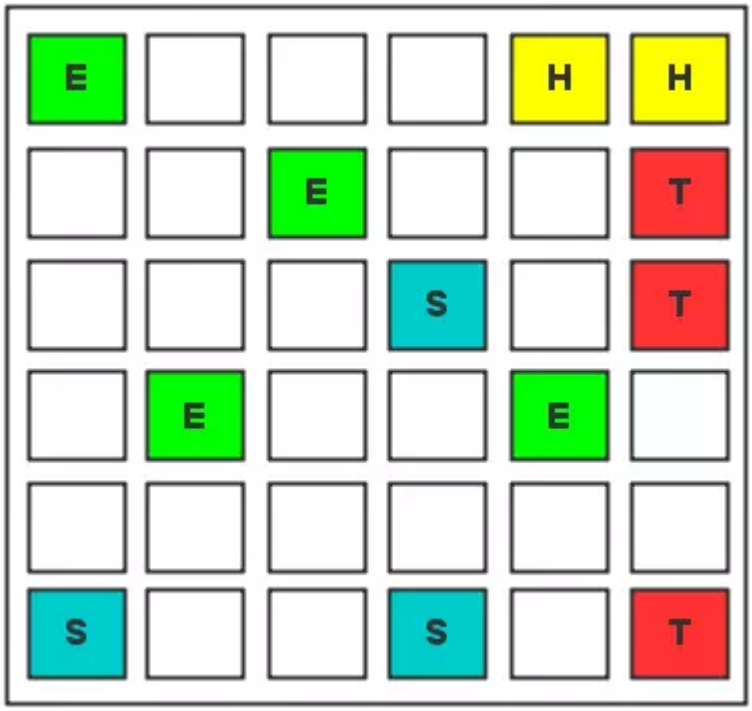
3.3.7 G1收集器(唯一一款全区域的垃圾回收器)

G1（Garbage First）垃圾回收器是用在heap memory很大的情况下，把heap划分为很多很多的region块，然后并行的对其进行垃圾回收。

G1垃圾回收器在清除实例所占用的内存空间后，还会做内存压缩。

G1垃圾回收器回收region的时候基本不会STW，而是基于 most garbage优先回收(整体来看是基于"标记-整理"算法，从局部(两个region之间)基于"复制"算法) 的策略来对region进行垃圾回收的。无论如何，G1收集器采用的算法都意味着

结果如下图：

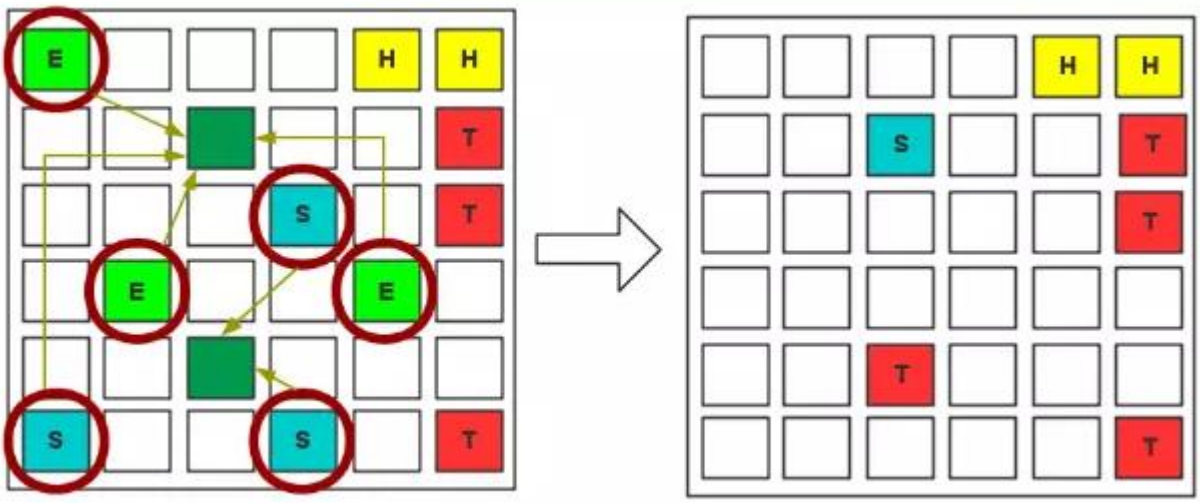


一个region有可能属于Eden，Survivor或者Tenured内存区域。图中的E表示该region属于Eden内存区域，S表示属于Survivor内存区域，T表示属于Tenured内存区域。图中空白的表示未使用的内存空间。G1垃圾收集器还增加了一种新的内存区域，叫做Humongous内存区域，如图中的H块。这种内存区域主要用于存储大对象-即大小超过一个region大小的50%的对象。

年轻代垃圾收集

在G1垃圾收集器中，年轻代的垃圾回收过程使用复制算法。把Eden区和Survivor区的对象复制到新的Survivor区域。

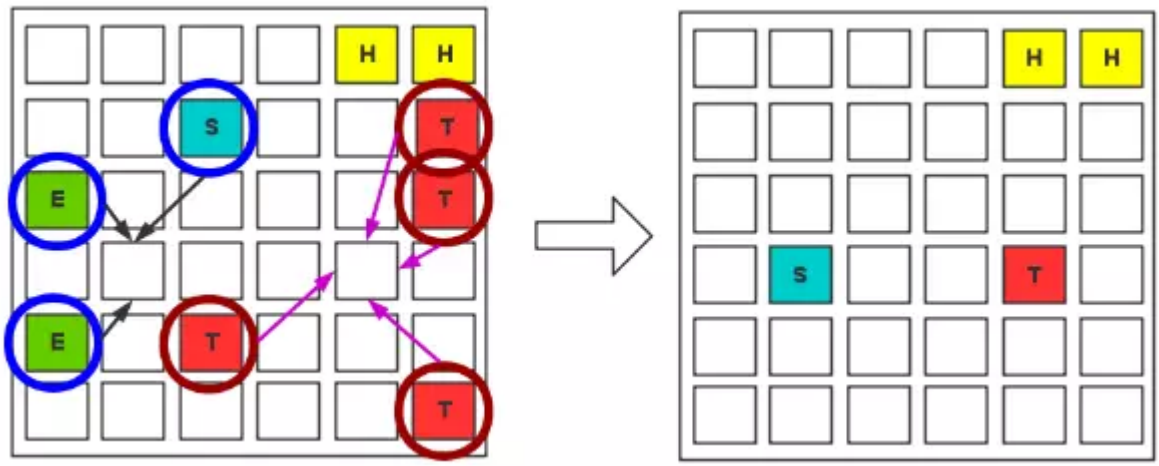
如下图：



老年代垃收集

对于老年代上的垃圾收集，G1垃圾收集器也分为4个阶段，基本跟CMS垃圾收集器一样，但略有不同：

- 初始标记(Initial Mark)阶段 - 同CMS垃圾收集器的Initial Mark阶段一样，G1也需要暂停应用程序的执行，它会标记从根对象出发，在根对象的第一层孩子节点中标记所有可达的对象。但是G1的垃圾收集器的Initial Mark阶段是跟minor gc一同发生的。也就是说，在G1中，你不用像在CMS那样，单独暂停应用程序的执行来运行Initial Mark阶段，而是在G1触发minor gc的时候一并将年老代上的Initial Mark给做了。
- 并发标记(Concurrent Mark)阶段 - 在这个阶段G1做的事情跟CMS一样。但G1同时还多做了一件事情，就是如果在Concurrent Mark阶段中，发现哪些Tenured region中对象的存活率很小或者基本没有对象存活，那么G1就会在这个阶段将其回收掉，而不用等到后面的clean up阶段。这也是Garbage First名字的由来。同时,在该阶段，G1会计算每个 region的对象存活率，方便后面的clean up阶段使用 。
- 最终标记(CMS中的Remark阶段) - 在这个阶段G1做的事情跟CMS一样, 但是采用的算法不同，G1采用一种叫做SATB(snapshot-at-the-begining)的算法能够在Remark阶段更快的标记可达对象。
- 筛选回收(Clean up/Copy)阶段 - 在G1中，没有CMS中对应的Sweep阶段。相反 它有一个Clean up/Copy阶段，在这个阶段中,G1会挑选出那些对象存活率低的region进行回收，这个阶段也是和minor gc一同发生的,如下图所示:



G1（Garbage-First）是一款面向**服务端应用**的垃圾收集器。HotSpot开发团队赋予它的使命是未来可以替换掉JDK 1.5中发布的CMS收集器。 如果你的应用追求低停顿，G1可以作为选择；如果你的应用追求吞吐量，G1并不带来特别明显的好处。

3.4 理解GC日志

每一种收集器的日志形式都是由它们自身的实现所决定的，换而言之，每个收集器的日志格式都可以不一样。但虚拟机设计者为了方便用户阅读，将各个收集器的日志都维持一定的共性，例如以下两段典型的GC日志:

```
[GC [DefNew: 3324K->152K(3712K), 0.0025925 secs] 3324K->152K(11904K), 0.0031680 secs]

[Full GC [Tenured: 0K->210K(10240K), 0.0149142 secs] 4603K->210K(19456K), [Perm :
2999K->2999K(21248K)], 0.0150007 secs] [Times: user=0.01 sys=0.00, real=0.02 secs]
```

GC日志开头的“[GC”和“[Full GC”说明了这次垃圾收集的停顿类型，而不是用来区分新生代GC还是老年代GC的。如果有“Full”，说明这次GC是发生了Stop-The-World的。 例如下面这段新生代收集器ParNew的日志也会出现“[Full GC”（这一般是因为出现了分配担保失败之类的问题，所以才导致STW）。如果是调用System.gc()方法所触发的收集，那么在这里将显示“[Full GC (System)”。

```
[Full GC 283.736: [ParNew: 261599K->261599K(261952K), 0.0000288 secs]
```

接下来的“[DefNew”、“[Tenured”、“[Perm”表示GC发生的区域，这里显示的区域名称与使用的GC收集器是密切相关的，例如上面样例所使用的Serial收集器中的新生代名为“Default New Generation”，所以显示的是“[DefNew”。如果是ParNew收集器，新生代名称就会变为“[ParNew”，意为“Parallel New Generation”。如果采用Parallel Scavenge收集器，那它配套的新生代称为“PSYoungGen”，老年代和永久代同理，名称也是由收集器决定的。

后面方括号内部的“3324K->152K(3712K)”含义是“GC前该内存区域已使用容量-> GC后该内存区域已使用容量 (该内存区域总容量)”。而在方括号之外的“3324K->152K(11904K)”表示“GC前Java堆已使用容量 -> GC后Java堆已使用容量 (Java堆总容量)”。

再往后，“0.0025925 secs”表示该内存区域GC所占用的时间，单位是秒。有的收集器会给出更具体的时间数据，如“[Times: user=0.01 sys=0.00, real=0.02 secs]”，这里面的user、sys和real与Linux的time命令所输出的时间含义一致，分别代表用户态消耗的CPU时间、内核态消耗的CPU事件和操作从开始到结束所经过的墙钟时间（Wall Clock Time）。CPU时间与墙钟时间的区别是，墙钟时间包括各种非运算的等待耗时，例如等待磁盘I/O、等待线程阻塞，而CPU时间不包括这些耗时，但当系统有多CPU或者多核的话，多线程操作会叠加这些CPU时间，所以读者看到user或sys时间超过real时间是完全正常的。

3.5 内存分配与回收策略

之前章节我们一直在讲JVM的内存回收，现在我们来看看JVM如何给对象分配内存。

3.5.1 对象优先在Eden分配

大多数情况下，对象在新生代Eden区中分配。当Eden区没有足够的空间进行分配时，虚拟机将发生一次Minor GC

范例 : 观察新生代GC

```
/**
 * JVM参数如下:
 * -XX:+PrintGCDetails
 * -XX:+UseSerialGC(使用Serial+Serial Old收集器组合)
 * -Xms20M -Xmx20M -Xmn10M(设置新生代大小)
 * -XX:SurvivorRatio=8(Eden:Survivor = 8 : 1)
 * @author 38134
 *
 */
public class Test {
    private static final int _1MB = 1024 * 1024;
    public static void testAllocation() {
        byte[] allocation1,allocation2,allocation3,allocation4;
        allocation1 = new byte[2 * _1MB];
        allocation2 = new byte[2 * _1MB];
        allocation3 = new byte[2 * _1MB];
        // 出现Minor GC
        allocation4 = new byte[4 * _1MB];
    }
    public static void main(String[] args) throws Exception{
        testAllocation();
    }
}
```

运行上述程序看到gc日志如下:

```
[GC (Allocation Failure) [DefNew: 7316K->624K(9216K), 0.0038519 secs] 7316K-
>6768K(19456K), 0.0038901 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
Heap
 def new generation   total 9216K, used 4802K [0x00000000fec00000, 0x00000000ff600000,
0x00000000ff600000)
  eden space 8192K,  51% used [0x00000000fec00000, 0x00000000ff014930,
0x00000000ff400000)
  from space 1024K,  60% used [0x00000000ff500000, 0x00000000ff59c198,
0x00000000ff600000)
  to   space 1024K,   0% used [0x00000000ff400000, 0x00000000ff400000,
0x00000000ff500000)
 tenured generation   total 10240K, used 6144K [0x00000000ff600000, 0x0000000100000000,
0x0000000100000000)
   the space 10240K,  60% used [0x00000000ff600000, 0x00000000ffc00030,
0x00000000ffc00200, 0x0000000100000000)
 Metaspace       used 2776K, capacity 4486K, committed 4864K, reserved 1056768K
  class space    used 296K, capacity 386K, committed 512K, reserved 1048576K
```

执行allocation4 = new byte[4 * _1MB];语句会发生一次Minor GC，这次GC的结果是7316K->624K(9216K-新生代总可用空间)，而总内存占用量几乎没有减少(因为allocation1、allocation2、allocation3这三个对象都还存活，虚拟机几乎没有找到可回收对象)。这次GC发生的原因是给allocation4分配内存的时候，发现Eden已经被占用了6MB，剩余空间不足以分配allocation4所需的4MB内存。因此发生Minor GC。GC期间虚拟机又发现已有的三个2MB对象无法全部放到survivor空间(空闲的To空间只有1MB)，所以只好通过分配担保机制提前转移到老年代。

本次GC结束后，4MB的allocation4对象成功分配在Eden中；剩余的三个对象(allocation1、allocation2、allocation3)被转移到老年代。上述的GC日志可以证实这一点。

3.5.2 大对象直接进入老年代

所谓的大对象是指，需要大量连续空间的Java对象，最典型的大对象就是那种很长的字符串以及数组(上述代码中的byte[] 数组就是典型的大对象)。大对象对虚拟机的内存分配是一个坏消息，经常出现大对象容易导致内存还有不少空间时就提前触发GC以获取足够的连续空间来放置大对象。

虚拟机提供了一个-XX:PretenureSizeThreshold参数，令大于这个设置值的对象直接在老年代分配。**这样做的目的在于避免Eden区以及两个Survivor区之间发生大量的内存复制(新生代采用复制算法收集内存)**

范例： 观察大对象直接进入老年代

```
/**
 * JVM参数如下:
 * -XX:+PrintGCDetails
 * -XX:+UseSerialGC(使用Serial+Serial Old收集器组合)
 * -Xms20M -Xmx20M -Xmn10M(设置新生代大小)
 * -XX:SurvivorRatio=8(Eden:Survivor = 8 : 1)
 * -XX:PretenureSizeThreshold = 3145728(此时不能写3MB)
 * @author 38134
 *
 */
public class Test {
    private static final int _1MB = 1024 * 1024;
    public static void testAllocation() {
        byte[] allocation;
```

```
        allocation = new byte[4 * _1MB];
    }
    public static void main(String[] args) throws Exception{
        testAllocation();
    }
}
```

```
Heap
 def new generation   total 9216K, used 1337K [0x00000000fec00000, 0x00000000ff600000,
0x00000000ff600000)
  eden space 8192K,  16% used [0x00000000fec00000, 0x00000000fed4e438,
0x00000000ff400000)
  from space 1024K,   0% used [0x00000000ff400000, 0x00000000ff400000,
0x00000000ff500000)
  to   space 1024K,   0% used [0x00000000ff500000, 0x00000000ff500000,
0x00000000ff600000)
 tenured generation   total 10240K, used 4096K [0x00000000ff600000, 0x0000000100000000,
0x0000000100000000)
   the space 10240K,  40% used [0x00000000ff600000, 0x00000000ffa00010,
0x00000000ffa00200, 0x0000000100000000)
 Metaspace       used 2774K, capacity 4486K, committed 4864K, reserved 1056768K
  class space    used 296K, capacity 386K, committed 512K, reserved 1048576K
```

执行上述代码之后，我们看到Eden空间几乎没有被使用，而老年代的10MB空间被占用了40%，也就是4MB的allocation对象就直接分配到老年代中，因为PretenureSizeThreshold被设置为3MB，因此超过3MB的对象都会直接在老年代进行分配。

3.5.3 长期存活的对象将进入老年代

既然虚拟机采用分代收集的思想来管理内存，那么内存回收时就必须能识别哪些对象应该放在新生代，哪些对象应该放在老年代中。为了做到这点，虚拟机给每个对象定义了一个对象年龄(Age)计数器。如果对象在Eden出生并经过一次Minor GC后仍然存活，并且能被Survivor容纳的话，将被移动到Survivor空间中，并且把对象年龄设为1.对象在Survivor空间中每"熬过"一次Minor GC，年龄就增加1岁，当它的年龄增加到一定程度(默认为15岁)，就将晋升到老年代中。对象晋升到老年代的年龄阈值，可以通过参数-XX:MaxTenuringThreshold设置。

下面我们分别设置MaxTenuringThreshold = 1 和 MaxTenuringThreshold = 15来观察。

范例 : 长期存活的对象进入老年代

```
/**
 * JVM参数如下:
 * -XX:+PrintGCDetails
 * -XX:+UseSerialGC(使用Serial+Serial Old收集器组合)
 * -Xms20M -Xmx20M -Xmn10M(设置新生代大小)
 * -XX:SurvivorRatio=8(Eden:Survivor = 8 : 1)
 * -XX:MaxTenuringThreshold=1
 * -XX:+PrintTenuringDistribution
 * @author 38134
 *
 */
public class Test {
    private static final int _1MB = 1024 * 1024;
```

```
    @SuppressWarnings("unused")
    public static void testAllocation() {
        byte[] allocation1,allocation2,allocation3;
        allocation1 = new byte[_1MB/4];
        // 什么时候进入老年代取决于XX:MaxTenuringThreshold的设置
        allocation2 = new byte[4 * _1MB];
        allocation3 = new byte[4 * _1MB];
        allocation3 = null;
        allocation3 = new byte[4 * _1MB];
    }
    public static void main(String[] args) throws Exception{
        testAllocation();
    }
}
```

此方法中的allocation1对象需要256K内存，Survivor可以容纳。当MaxTenuringThreshold = 1时，allocation1对象在第二次gc的时候进入老年代，新生代已使用的内存在第二次gc后干净的变成0k。

```
[GC (Allocation Failure) [DefNew: 5524K->880K(9216K), 0.0029366 secs] 5524K-
>4976K(19456K), 0.0029730 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC (Allocation Failure) [DefNew: 4976K->0K(9216K), 0.0008841 secs] 9072K-
>4975K(19456K), 0.0009006 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
Heap
 def new generation   total 9216K, used 4178K [0x00000000fec00000, 0x00000000ff600000,
0x00000000ff600000)
  eden space 8192K,  51% used [0x00000000fec00000, 0x00000000ff014930,
0x00000000ff400000)
  from space 1024K,   0% used [0x00000000ff400000, 0x00000000ff400000,
0x00000000ff500000)
  to   space 1024K,   0% used [0x00000000ff500000, 0x00000000ff500000,
0x00000000ff600000)
 tenured generation   total 10240K, used 4975K [0x00000000ff600000, 0x0000000100000000,
0x0000000100000000)
   the space 10240K,  48% used [0x00000000ff600000, 0x00000000ffadbce8,
0x00000000ffadbe00, 0x0000000100000000)
 Metaspace       used 2775K, capacity 4486K, committed 4864K, reserved 1056768K
  class space    used 296K, capacity 386K, committed 512K, reserved 1048576K
```

而当MaxTenuringThreshold = 15时，第二次gc后，allocation1对象还留在新生代Survivor空间，此时新生代仍然有404k占用。

3.5.4 动态对象年龄判定

为了能更好的适应不同程序的内存状况，JVM并不是永远要求对象的年龄必须达到MaxTenuringThreshold才能晋升老年代，如果在Survivor空间中相同年龄所有对象大小的总和大于Survivor空间的一半，年龄大于或等于该年龄的对象就可以直接进入老年代，无须等到MaxTenuringThreshold中要求的年龄。

范例 : 动态对象年龄判定

```
package jvm;
/**
```

```
 * JVM参数如下:
 * -XX:+PrintGCDetails
 * -XX:+UseSerialGC(使用Serial+Serial Old收集器组合)
 * -Xms20M -Xmx20M -Xmn10M(设置新生代大小)
 * -XX:SurvivorRatio=8(Eden:Survivor = 8 : 1)
 * -XX:MaxTenuringThreshold=15
 * @author 38134
 *
 */
public class Test {
    private static final int _1MB = 1024 * 1024;
    @SuppressWarnings("unused")
    public static void testAllocation() {
        byte[] allocation1,allocation2,allocation3,allocation4;
        allocation1 = new byte[_1MB / 4];
        // allocation1+allocation2大于Survivor空间的一半
        allocation2 = new byte[_1MB / 4];
        allocation3 = new byte[4 * _1MB];
        allocation4 = new byte[4 * _1MB];
        allocation4 = null;
        allocation4 = new byte[4 * _1MB];
    }
    public static void main(String[] args) throws Exception{
        testAllocation();
    }
}
```

```
[GC (Allocation Failure) [DefNew
Desired survivor size 524288 bytes, new threshold 1 (max 15)
- age   1:    1048576 bytes,    1048576 total
: 5780K->1024K(9216K), 0.0029212 secs] 5780K->5232K(19456K), 0.0029656 secs] [Times:
user=0.00 sys=0.00, real=0.00 secs]
[GC (Allocation Failure) [DefNew
Desired survivor size 524288 bytes, new threshold 15 (max 15)
: 5120K->0K(9216K), 0.0009421 secs] 9328K->5232K(19456K), 0.0009580 secs] [Times:
user=0.00 sys=0.00, real=0.00 secs]
Heap
 def new generation   total 9216K, used 4178K [0x00000000fec00000, 0x00000000ff600000,
0x00000000ff600000)
  eden space 8192K,  51% used [0x00000000fec00000, 0x00000000ff014930,
0x00000000ff400000)
  from space 1024K,   0% used [0x00000000ff400000, 0x00000000ff400000,
0x00000000ff500000)
  to   space 1024K,   0% used [0x00000000ff500000, 0x00000000ff500000,
0x00000000ff600000)
 tenured generation   total 10240K, used 5232K [0x00000000ff600000, 0x0000000100000000,
0x0000000100000000)
   the space 10240K,  51% used [0x00000000ff600000, 0x00000000ffb1c150,
0x00000000ffb1c200, 0x0000000100000000)
 Metaspace       used 2776K, capacity 4486K, committed 4864K, reserved 1056768K
  class space    used 296K, capacity 386K, committed 512K, reserved 1048576K
```

上述代码以及gc日志可以看出来，虽然我们设置了MaxTenuringThreshold=15，发现运行结果中Survivor空间占用仍为0%，而老年代比预期增加了6%。也就是说allocation1、allocation2对象都直接进入了老年代，而没有等到Age为15的临界值。**因为这两个对象加起来已经达到了512KB，并且它们是同年的，满足同年对象达到Survivor空间的一半规则。**我们只要注释掉其中一个对象的new操作，就会发现另外一个就不会晋升到老年代中去了。

3.5.5 空间分配担保

在发生Minor GC之前，虚拟机会检查**老年代最大可用的连续空间**是否大于**新生代所有对象的总空间**，

- 如果大于，则此次**Minor GC是安全的**
- 如果小于，则虚拟机会查看**HandlePromotionFailure**设置值是否允许担保失败。 如果HandlePromotionFailure=true，那么会继续检查老年代最大可用连续空间是否大于**历次晋升到老年代的对象的平均大小**，如果大于，则尝试进行一次Minor GC，但这次Minor GC依然是有风险的；如果小于或者HandlePromotionFailure=false，则改为进行一次Full GC。

上面提到了Minor GC依然会有风险，是因为新生代采用**复制收集算法**，假如大量对象在Minor GC后仍然存活（最极端情况为内存回收后新生代中所有对象均存活），而Survivor空间是比较小的，这时就需要老年代进行分配担保，把Survivor无法容纳的对象放到老年代。**老年代要进行空间分配担保，前提是老年代得有足够空间来容纳这些对象**，但一共有多少对象在内存回收后存活下来是不可预知的，**因此只好取之前每次垃圾回收后晋升到老年代的对象大小的平均值作为参考。**使用这个平均值与老年代剩余空间进行比较，来决定是否进行Full GC来让老年代腾出更多空间。

取平均值仍然是一种**概率性的事件**，如果某次Minor GC后存活对象陡增，远高于平均值的话，必然导致担保失败，如果出现了分配担保失败，**就只能在失败后重新发起一次Full GC。**虽然存在发生这种情况的概率，但**大部分时候都是能够成功分配担保**的，这样就避免了过于频繁执行Full GC。

4. 常用JVM性能监控与故障处理工具

4.1 JDK命令行工具

之前我们一直使用的"javac" "java"命令实际上都是JDK/bin目录下的命令行程序。除此之外，bin目录下还包含用于监控JVM和故障处理的工具。这些工具被Sun公司作为礼物送给JDK使用者，他们非常稳定且功能强大，能在处理应用程序性能问题、定位故障时发挥很大作用。

下面是bin目录下常用六大命令的名称以及用途。

命令名称	全称	用途
jps	JVM Process Status Tool	显示指定系统内所有的HotSpot虚拟机进程
jstat	JVM Statistics Monitoring Tool	用于收集Hotspot虚拟机各方面的运行数据
jinfo	Configuration Info for Java	显示虚拟机配置信息
jmap	JVM Memory Map	生成虚拟机的内存转储快照，生成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，让用户在浏览器上查看分析结果
jstack	JVM Stack Trace	显示虚拟机的线程快照

4.2 jps：虚拟机进程状态工具

jps是使用频率最高的JDK命令行工具，它可以列出正在运行的虚拟机进程，并显示虚拟机执行主类(main函数所在的类)名称以及这些进程的本地虚拟机唯一ID(Local Virtual Machine Identifier,LVMID)。其他的JDK工具大多需要输入jps查询到的LVMID来确定要监控的是哪一个虚拟机进程。

jps命令格式

jps 【options】【hostid】

options 有:

-q	只输出LVMID，省略主类的名称
-m	输出虚拟机进程启动时传递给主类main()函数的参数
-l	输出主类的全名，如果进程执行的是jar，输出jar路径
-v	输出虚拟机进程启动时JVM参数

范例 : 使用jps

```
$ jps -l
2976 Print
28648 sun.tools.jps.Jps
39036
```

hostid可用于查询开启了RMI服务的远程虚拟机进程状态，hostid为RMI注册表中注册的主机名

4.3 jstat：虚拟机统计信息监视工具

jstat（JVM Statistics Monitoring Tool）是用于监控虚拟机各种运行状态信息的命令行工具。它可以显示本地或远程虚拟机中的类装载、内存、垃圾收集、JIT编译等运行数据。 在没有GUI图形界面，只提供了纯文本控制台环境的服务器上，它将是运行期定位虚拟机性能问题的首选工具。

jstat命令格式为 :

`jstat [option vmid [interval count]]`

如果是本地虚拟机进程 VMID与LVMID是一致的，如果是远程虚拟机进程，那么VMID的格式应为:

`[protocol:][//] lvmind[@hostname [:port] /servername]`

参数interval和count代表查询间隔和次数，如果省略这两个参数，说明只查询一次。

选项option代表用户希望查询的虚拟机信息，主要分为三类 : 类装载、垃圾收集、运行期编译等状况，具体选项及作用见下表

选项	作用
-class	监视类装载、卸载数量、总空间以及装载类所耗费的时间
-gc	监视java堆状况，包括Eden区、两个Survivor区、年老代、永久代等的容量、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、最小空间
-gcutil	监视内容与-gc基本相同，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、最小空间
-gcold	监视老年代GC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、最小空间
-gcpermcapacity	输出永久代使用到的最大、最小空间
-compiler	输出JIT编译器编译过的方法、耗时等信息
-printcompilation	输出已经被JIT编译器编译的方法

范例 : 使用jstat

```
$ jstat -gcutil 2976
  S0     S1     E      O      M     CCS    YGC     YGCT    FGC    FGCT     GCT
  0.00   0.00   6.00   0.00  17.21  19.76      0    0.000     0    0.000    0.000
```

查询结果表明，此JVM进程新生代Eden(E,表示Eden)使用6%的空间，两个Survivor区(S0,S1表示Survivor0、Survivor1)里面都是空的，老年代(O,表示Old)和元空间(M,表示Meta)则分别使用了0%和17.21%的空间。程序运行以来共发生Minor GC(YGC,表示Young GC)0次，共耗时0秒；发生Full GC(FGC,表示Full GC)0次，共耗时0秒。所有GC总耗时(GCT,表示GC Time)0秒。

4.4 jinfo : Java配置信息工具

jinfo用于查看和调整虚拟机的配置参数.

jinfo的命令格式为:

`jinfo [option] pid`

范例 : 使用jinfo

```
$ jinfo -flags 2976
Attaching to process ID 2976, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 25.161-b12
Non-default VM flags: -XX:CICompilerCount=4 -XX:InitialHeapSize=134217728 -
XX:MaxHeapSize=2118123520 -XX:MaxNewSize=705691648 -XX:MinHeapDeltaBytes=524288 -
XX:NewSize=44564480 -XX:OldSize=89653248 -XX:+UseCompressedClassPointers -
XX:+UseCompressedOops -XX:+UseFastUnorderedTimeStamps -XX:-
UseLargePagesIndividualAllocation -XX:+UseParallelGC
Command line:  -Dfile.encoding=GBK
```

我们可以用jinfo -flags来查询线程的参数，其中的Non-default VM flags为虚拟机默认的设置参数，Command line为用户自行设置的参数

4.5 jmap : Java内存映像工具

jmap（Memory Map for Java）命令用于生成堆转储快照（一般称为heapdump或dump文件）。如果不使用jmap命令，要想获取Java堆转储快照，还有一些比较暴力的手段 : 譬如在之前用过的-XX:+HeapDumpOnOutOfMemoryError参数，可以让虚拟机在OOM异常出现之后自动生成dump文件。

jmap的作用并不仅仅为了获取dump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、当前使用的是哪种收集器等，

jmap的命令格式

`jmap [option] vmid`

option的常用选项及作用见下表

选项	作用
-heap	显示jvm heap详细信息
-histo	显示jvm heap中对象统计信息，包括类，实例数量，合计容量
-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

范例 : 观察java heap的内存使用情况

```
$ jmap -heap 2976
```

范例 : 观察heap中所有对象的情况，包括对象数量和所占空间大小

```
$ jmap -histo 2976
```

```
$ jmap -histo:live 2976
```

范例 : dump出所有对象文件可用于进一步分析

```
$ jmap -dump:format=b,file=heap.bin 2976
Dumping heap to C:\Users\38134\heap.bin ...
Heap dump file created
```

4.6 jhat : 虚拟机转存储快照分析工具

jhat(JVM Heap Analysis Tool)命令与jmap命令搭配使用，用于分析jmap生成的堆转储快照，jhat内置了一个微型的HTTP/HTML服务器，生成dump文件的分析结果后，可以在浏览器中查看。实际工作中使用jhat的几率并不大，因为分析dump文件的工作是一个耗时且消耗硬件资源的过程，一般不会在部署机器上分析；二是jhat分析功能相对来说比较简陋，我们可以用Visual VM或者MAT等工具实现比jhat更强大更专业的分析功能。

范例 : 使用jhat分析dump文件

```
$ jhat heap.bin
Reading from heap.bin...
Dump file created Fri May 25 14:48:01 CST 2018
Snapshot read, resolving...
Resolving 10105 objects...
Chasing references, expect 2 dots..
Eliminating duplicate references..
Snapshot resolved.
Started HTTP server on port 7000
Server is ready.
```

如上图所示，屏幕显示"Server is ready"后，用户在浏览器中输入"http://localhost:7000"就可以看到分析结果

4.7 jstack : Java堆栈跟踪工具

jstack命令用于生成虚拟机当前时刻的线程快照。线程快照指的是当前虚拟机内的每一条线程正在执行的方法堆栈的集合，生成线程快照的作用是，可用于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待等问题，当线程出现停顿时 就可以用jstack各个线程调用的堆栈情况。

`jstack [option] vmid`

option的常用选项及作用见下表:

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	如果调用到本地方法的话，可以显示C/C++的堆栈

范例 : 使用jstack

`jstack -l 7596`

```
Found one Java-level deadlock:
=============================
"book-thread":
  waiting to lock monitor 0x0000000019fe0c38 (object 0x00000000d5fe57a8, a Pen),
  which is held by "pen-thread"
"pen-thread":
  waiting to lock monitor 0x0000000019fe3628 (object 0x00000000d5fe6978, a Book),
  which is held by "book-thread"

Java stack information for the threads listed above:
===================================================
"book-thread":
        at Print$2.run(Print.java:29)
        - waiting to lock <0x00000000d5fe57a8> (a Pen)
        - locked <0x00000000d5fe6978> (a Book)
        at java.lang.Thread.run(Thread.java:748)
"pen-thread":
        at Print$1.run(Print.java:16)
        - waiting to lock <0x00000000d5fe6978> (a Book)
        - locked <0x00000000d5fe57a8> (a Pen)
        at java.lang.Thread.run(Thread.java:748)

Found 1 deadlock.
```

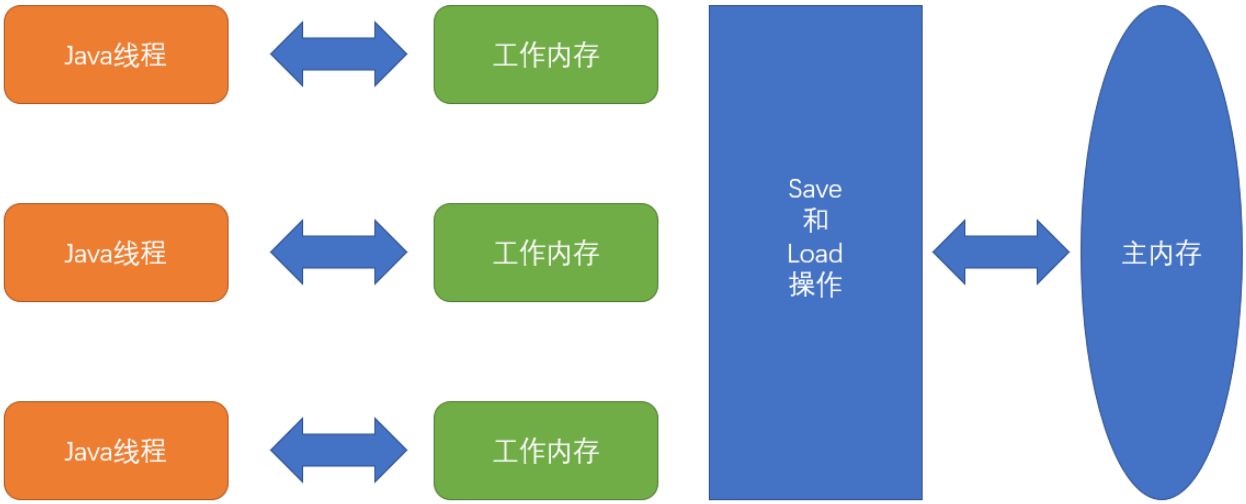
5. Java内存模型

JVM定义了一种Java内存模型(Java Memory Model,JMM)来屏蔽掉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的内存访问效果。在此之前，C/C++直接使用物理硬件和操作系统的内存模型，因此，会由于不同平台下的内存模型的差异，有可能导致程序在一套平台上*并发*完全正常，而在另一套平台上*并发*访问经常出错。

5.1 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JVM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。此处的变量包括**实例字段、静态字段和构成数组对象的元素**，但不包括局部变量和方法参数，因为后两者是线程私有的，不会被线程共享。

Java内存模型规定了所有的变量都存储在主内存中。每条线程还有自己的工作内存，线程的工作内存中保存了被该线程使用到的变量的主内存副本拷贝，线程对变量的所有操作(读取、赋值等)都必须在工作内存进行，而不能直接读写主内存中的变量。不同的线程之间也无法直接访问对方工作内存中的变量，线程间变量值的传递均需要通过主内存来完成。线程、主内存、工作内存三者的交互关系如下所示：



5.2 内存间交互操作

关于主内存与工作内存之间的具体交互协议，即一个变量如何从主内存中拷贝到工作内存、如何从工作内存同步回主内存之类的实现细节，Java内存模型中定义了如下8种操作来完成。JVM实现时必须保证下面提及的每一种操作的原子的、不可再分的。

- lock(锁定)：作用于主内存的变量，它把一个变量标识为一条线程独占的状态
- unlock(解锁)：作用于主内存的变量，它把一个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。
- 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。
- 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的变量值放入工作内存的变量副本中。
- 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。
- 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给工作内存的变量。
- store(存储)：作用于工作内存的变量，它把工作内存中一个变量的值传送到主内存中，以便后续的write操作使用。
- 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。

Java内存模型的三大特性：

- **原子性**：由Java内存模型来直接保证的原子性变量操作包括read、load、assign、use、store和read。大致可以认为，基本数据类型的访问读写是具备原子性的。如若需要更大范围的原子性，需要synchronized关键字约束。(即一个操作或者多个操作 要么全部执行并且执行的过程不会被任何因素打断，要么就都不执行)
- **可见性**：可见性是指当一个线程修改了共享变量的值，其他线程能够立即得知这个修改。volatile、synchronized、final三个关键字可以实现可见性。
- **有序性**：如果在本线程内观察，所有的操作都是有序的；如果在线程中观察另外一个线程，所有的操作都是无序的。前半句是指"线程内表现为串行"，后半句是指"指令重排序"和"工作内存与主内存同步延迟"现象。

Java内存模型具备一些先天的“有序性”，即不需要通过任何手段就能够得到保证的有序性，这个通常也称为 **happens-before** 原则。如果两个操作的执行次序无法从happens-before原则推导出来，那么它们就不能保证它们的有序性，虚拟机可以随意地对它们进行重排序。

下面就来具体介绍下happens-before原则（先行发生原则）：

- 程序次序规则：一个线程内，按照代码顺序，书写在前面的操作先行发生于书写在后面的操作
- 锁定规则：一个unLock操作先行发生于后面对同一个锁的lock操作
- volatile变量规则：对一个变量的写操作先行发生于后面对这个变量的读操作
- 传递规则：如果操作A先行发生于操作B，而操作B又先行发生于操作C，则可以得出操作A先行发生于操作C
- 线程启动规则：Thread对象的start()方法先行发生于此线程的每个一个动作
- 线程中断规则：对线程interrupt()方法的调用先行发生于被中断线程的代码检测到中断事件的发生
- 线程终结规则：线程中所有的操作都先行发生于线程的终止检测，我们可以通过Thread.join()方法结束、Thread.isAlive()的返回值手段检测到线程已经终止执行
- 对象终结规则：一个对象的初始化完成先行发生于他的finalize()方法的开始

也就是说，要想并发程序正确地执行，必须要保证原子性、可见性以及有序性。只要有一个没有被保证，就有可能会导致程序运行不正确。

5.3 volatile型变量的特殊规则

关键字volatile可以说是JVM提供的最轻量级的同步机制，但是它并不容易完全被正确理解和使用。JVM内存模型对volatile专门定义了一些特殊的访问规则。

当一个变量定义为volatile之后，它将具备两种特性。

第一：保证此变量对所有线程的可见性，这里的"可见性"是指：当一条线程修改了这个变量的值，新值对于其他线程来说是可以立即得知的。而普通变量做不到这一点，普通变量的值在线程间传递均需要通过主内存来完成。例如：线程A修改一个普通变量的值，然后向主内存进行回写，另外一条线程B在线程A回写完成之后再从主内存进行读取操作，新值才会对线程B可见。

关于volatile变量的可见性，经常会被开发人员误解。**volatile变量在各个线程中是一致的，但是volatile变量的运算在并发下一样是不安全的。**原因在于Java里面的运算并非原子操作。

范例：volatile变量自增操作

```
package com.company;
public class Main {
    public static volatile int num = 0;
    public static void increase() {
        num++;
    }
    public static void main(String[] args) {
       Thread[] threads = new Thread[10];
        for (int i = 0; i < 10; i++) {
            threads[i] = new Thread(new Runnable() {
                @Override
                public void run() {
                    for (int j = 0; j < 100; j++) {
                        increase();
                    }
                }
            });
```

```
            threads[i].start();
        }
        while (Thread.activeCount() > 2) {
            Thread.yield();
        }
        System.out.println(num);
    }
}
```

问题就在于num++之中，实际上num++等同于num = num+1。volatile关键字保证了num的值在取值时是正确的，但是在执行num+1的时候，其他线程可能已经把num值增大了，这样在+1后会把较小的数值同步回主内存之中。

由于volatile关键字只保证可见性，在不符合以下两条规则的运算场景中，我们仍然需要通过加锁(synchronized或者lock)来保证原子性。

1. 运算结果并不依赖变量的当前值，或者能够确保只有单一的线程修改变量的值
2. 变量不需要与其他的状态变量共同参与不变约束

如下代码这类场景就特别适合使用volatile来控制并发，当shutdown()方法被调用时，能保证所有线程中执行的doWork()方法都立即停下来。

```
volatile boolean shutdownRequested;

public void shutdown() {
    shutdownRequested = true;
}

public void work() {
    while(!shutdownRequested) {
        //do stuff
    }
}
```

第二：使用volatile变量的语义是禁止指令重排序。普通的变量仅仅会保证在该方法的执行过程中所有依赖赋值结果的地方都能获取到正确的结果，而不能保证变量赋值操作的顺序和程序代码中执行的顺序一致。

volatile关键字禁止指令重排序有两层意思：

1）当程序执行到volatile变量的读操作或者写操作时，在其前面的操作的更改肯定全部已经进行，且结果已经对后面的操作可见；在其后面的操作肯定还没有进行；

2）在进行指令优化时，不能将在对volatile变量访问的语句放在其后面执行，也不能把volatile变量后面的语句放到其前面执行。

举个简单的例子：

```
//x、y为非volatile变量
//flag为volatile变量
x = 2;         //语句1
y = 0;         //语句2
flag = true;   //语句3
x = 4;         //语句4
y = -1;        //语句5
```

由于flag变量为volatile变量，那么在进行指令重排序的过程的时候，不会将语句3放到语句1、语句2前面，也不会将语句3放到语句4、语句5后面。但是要注意语句1和语句2的顺序、语句4和语句5的顺序是不作任何保证的。

并且volatile关键字能保证，执行到语句3时，语句1和语句2必定是执行完毕了的，且语句1和语句2的执行结果对语句3、语句4、语句5是可见的。

范例 : 指令重排序

```
Map configOptions;
char[] configText;
volatile boolean initialized = false;

//假设以下代码在线程A执行
//模拟读取配置文件信息，当读取完成后将initialized设置为true以通知其他线程配置可用
configOptions = new HashMap();
configText = readConfigFile(fileName);
processConfigOptions(configText,configOptions);
initialized = true;

//假设以下代码在线程B执行
//等待initialized为true，代表线程A已经把配置信息初始化完成
while(!initialized) {
    sleep();
}
//使用线程A初始化好的配置信息
doSomethingWithConfig();
```

单例模式中的Double Check:

双重检验锁模式（double checked locking pattern），是一种使用同步块加锁的方法。程序员称其为双重检查锁，因为会有两次检查 instance == null，一次是在同步块外，一次是在同步块内。为什么在同步块内还要再检验一次？因为可能会有多个线程一起进入同步块外的 if，如果在同步块内不进行二次检验的话就会生成多个实例了。

```
public static Singleton getSingleton(){
        if(instance==null){ //Single Checked
            synchronized (Singleton.class){
                if(instance==null){ //Double Checked
                instance=new Singleton();
            }
        }
    }
        return instance;
}
```

这段代码看起来很完美，很可惜，它是有问题。主要在于instance = new Singleton()这句，这并非是一个原子操作，事实上在 JVM 中这句话大概做了下面 3 件事情。 **给 instance 分配内存** 调用 Singleton 的**构造函数来初始化**成员变量 **将instance对象指向分配的内存空间**（执行完这步 instance 就为非 null 了） 但是在 JVM 的即时编译器中存在指令重排序的优化。也就是说上面的第二步和第三步的顺序是不能保证的，最终的执行顺序可能是 1-2-3 也可能是 1-3-2。如果是后者，则在 3 执行完毕、2 未执行之前，被线程二抢占了，这时 instance 已经是非 null 了（但却没有初始化），所以线程二会直接返回 instance，然后使用，然后顺理成章地报错。 我们只需要将 instance 变量声明成 volatile 就可以了。

```
class Singleton{
    private volatile static Singleton instance = null;

    private Singleton() {

    }

    public static Singleton getInstance() {
        if(instance==null) {
            synchronized (Singleton.class) {
                if(instance==null)
                    instance = new Singleton();
            }
        }
        return instance;
    }
}
```