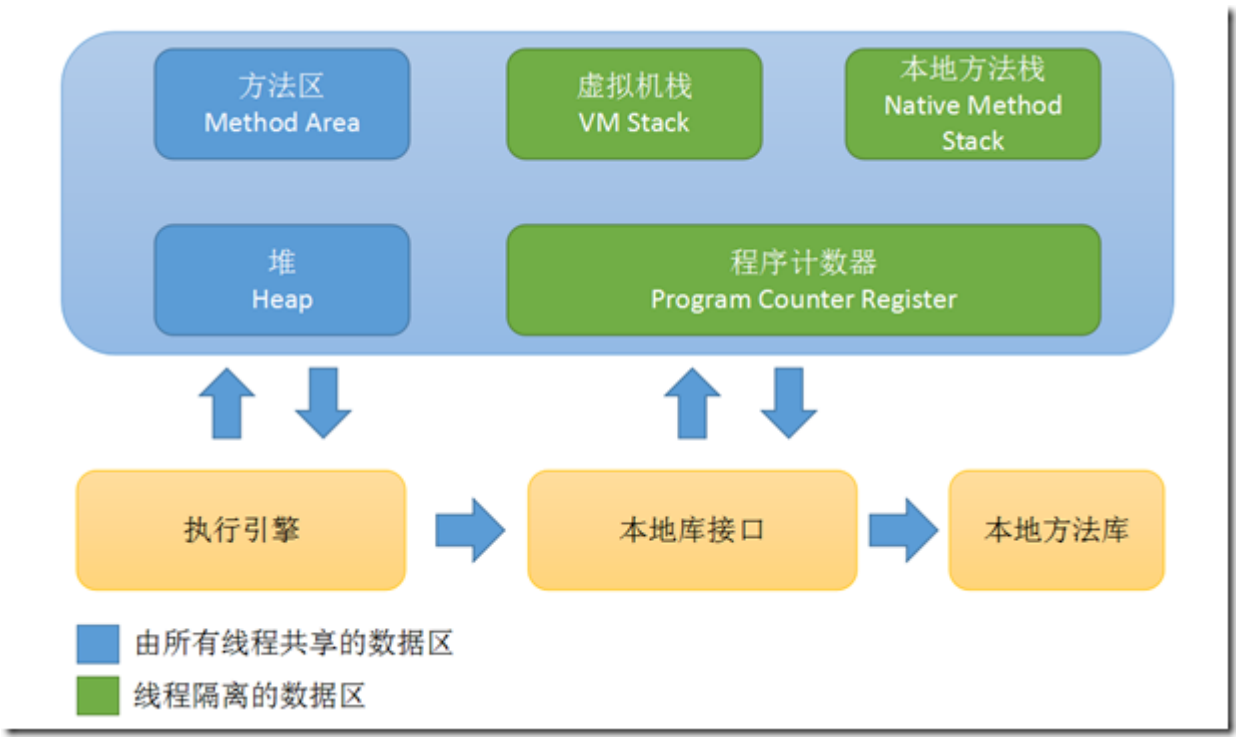
一、运行时数据区域

1、线程隔离的数据区

- 程序计数器（Program Counter Register）
  - 当前线程所执行代码的行号指示器
  - 每个线程有一个独立的程序计数器
- 虚拟机栈（VM Stack）
  - 线程私有，生命周期等同于线程
  - 每个方法在执行的同时会创建一个帧栈
  - 储存局部变量表、操作数栈、动态链接、方法出口等信息。最重要的是局部变量表，也是大家常常讨论的“栈”空间。
- 本地方法栈（Native Method Stack）
  - 本地方法栈类似于虚拟机栈，只不过虚拟机栈为虚拟机执行的java方法服务，本地方法栈为虚拟机使用到的Native方法服务



Notice：

1. HotPot 的实现合并了本地方法栈和虚拟机栈

2、由线程共享的数据区

- 堆（Heap）
  - 线程共享，虚拟机不关闭生命就不会结束。因此需要对此空间进行管理，是垃圾收集机制（GC）的主要区域。
  - 绝大部分对象实例及数组都要在堆上分配内存
- 方法区（Method Area）

    - 线程共享，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。
- 运行时常量池（Runtime Constant Pool）
    - 方法区的一部分，用于存放编译器生成的各种字面量和符号引用
- 直接内存（Direct Memory）
    - 直接分配堆外内存，不属于虚拟机运行时数据区。
    - 避免在Java堆和Native堆中来回复制数据，在某些场景能显著提高性能。如NIO中的通道（Channel）与缓冲区（Buffer）。

Notice：

1. 为什么GC的主要区域是Heap：动态创建的对象都在Heap上为其实例分配内存。Stack在方法和局部代码调用完成后释放帧栈空间，但Heap生命周期与虚拟机相同，内存不能自动释放。C/C++中程序员往往在代码中显示调用 free/delete 释放堆中对象的空间，操作繁琐，且容易发生内存泄漏。java 虚拟机的自动垃圾收集机制则能自动释放不需要用到的对象实例，实现堆内存的自动管理。
2. 已加载类的基本信息和方法储存在方法区
3. 常量（final）和静态变量（static）储存在方法区。问题：局部变量声明final，储存在哪个区域（堆、方法区、虚拟机栈）。
4. Object 和 Array 的实例在堆中分配内存，并在相应的位置（如栈）创建引用。没有有效引用将被GC。
5. HotPot 的实现把方法区合并到了堆的永久代区（Permanent Generation）。
6. 如何实现方法区是虚拟机的技术实现细节，但是使用永久代实现方法区现在看来并不是一个好主意，因为这样更容易遇到内存泄漏问题。JDK 1.7 的 HotPot 中，已经把原本放在永久代的字符串常量池移出。

三、堆栈溢出异常

1、对应参数：

- -Xms 堆的最小值
- -Xmx 堆的最大值
- -Xss 栈容量
- -Xoss 本地方法栈（HotPot 无效，因为没有设置单独的本地方法栈）
- -XX:PermSize=10M -XX:MaxPermSize=10M 永久代
- -XX:MaxDirectMemorySize=10M 直接内存
- -XX:+HeapDumpOnOutOfMemoryError 让虚拟机在出现内存溢出异常时Dump出当前的内存堆转储快照以便事后分析。

2、Java 堆溢出

- java.lang.OutOfMemoryError：Javaheap space
- 要解决这个区域的异常，一般的手段是先通过内存映像分析工具（如Eclipse Memory Analyzer）堆Dump出来的堆转储快照进行分析，重点是确认内存中的对象是否是必要的，也就是要分清楚到底是内存泄露还是内存溢出。
- 如果是内存泄露，可进一步通过工具查看泄露对象到GCRoots的引用链。于是就能找到泄露对象是通过怎样的路径与GCRoots相关联并导致垃圾收集器无法对他们进行回收的。掌握了泄露对象的类型信息及GC Roots引用链信息，就可以比较准确的定位到泄露代码的位置。
- 如果不存在泄露，换句话说，就是内存中的对象确实都是必须要存活的，那就应当检查虚拟机的堆参数（-Xms和-Xmx），与机器的物理内存对比看是否还可以调大，从代码上检查是否存在某些对象生命周期过长，持有状态过长的情况，尝试减少程序运行期的内存消耗。

```java
package com.susu.study.jvm;

import java.util.ArrayList;
import java.util.List;

/**
 * @Args: VM Args:-Xms20m -Xmx20m -XX:+HeapDumpOnOutOfMemoryError
 * -Xms 堆的最小值 -Xmx 堆的最大值 -XX:+HeapDumpOnOutOfMemoryError 让虚拟机在出现内存溢出异常时Dump出当前的内存堆转储快照以便事后分析。
 * @Description Java堆溢出 java.lang.OutOfMemoryError: Java heap space
 * @author: 01369674
 * @date: 2018/4/18
 */
public class HeapOOM {
    static class OOMObject{
    }

    public static void main(String[] args) {
        List<OOMObject> list = new ArrayList<OOMObject>();
        int count = 0;
        //循环创建对象直至内存溢出
        try {
            while(count<Integer.MAX_VALUE){
                list.add(new OOMObject());
                count++;
            }
        }catch (Throwable e) {
            System.out.println("count:"+count);
        }
    }
}

/*
* java.lang.OutOfMemoryError: Java heap space
* Dumping heap to java_pid643456.hprof ...
* Heap dump file created [8831270 bytes in 0.068 secs]
 */
```

3、虚拟机栈和本地方法栈溢出

- java.lang.StackOverflowError ：某线程中线程请求的栈深度大于虚拟机所允许的最大深度
- java.lang.OutOfMemoryError:unable to create new native thread：申请多线程时栈容量不够
- 因为操作系统分配给每个进程的内存是有限制的，比如32位Windows系统限制为2GB。虚拟机提供了参数来限制Java堆和方法区这两部分的最大值。剩余的内存为2GB（操作系统限制）减去Xmx（最大堆容量），再减去MaxPermSize（最大方法区容量），程序计数器消耗内存很小，可以忽略掉。如果虚拟机本身消耗的内存不计算在内，剩余的内存就是由虚拟机栈和本地方法栈瓜分掉了，每个线程分配到的栈容量越大，可以建立的线程数量自然就越少，建立线程时就越容易把内存耗尽。
- 所以，如果是建立过多线程导致的内存溢出，在不能减少线程数量或更换64位操作系统的情况下，可以通过减小最大堆和减小栈容量来换取更多的线程。如果没有这方面的处理经验，这种通过“减少内存”的手段来解决内存溢出的方式会很难想到。

```java
package com.susu.study.jvm;

/**
 * @Args: VM Args: -Xss128k
 * -Xss 设置虚拟机栈大小
 * @Description: 线程请求的栈深度大于虚拟机栈允许的最大深度，抛出 StackOverflowError 异常
 * @author: 01369674
 * @date: 2018/4/18
 */
public class JavaVMStackSOF {

    private int stackLength = 1;
    public void stackLeak(){
        stackLength++;
        stackLeak();
    }

    public static void main(String[] args) throws Throwable {
        JavaVMStackSOF oom = new JavaVMStackSOF();
        try{
            oom.stackLeak();
        }catch (Throwable e){
            System.out.println("stack length:"+oom.stackLength);
            throw e;
        }

    }
}

/*
 * stack length:2284
 * Exception in thread "main" java.lang.StackOverflowError
 */
```

```java
package com.susu.study.jvm;

/**
 * @Description: 虚拟机在扩展栈时无法申请到足够的内存空间，则抛出 OutOfMemoryError 异常
 * @Args: VM Args: -Xss2M
 * @author: 01369674
 * @date: 2018/4/18
 */
public class JavaVMStackOOM {
    private void dontStop(){
        int count=0;
        while(count<Integer.MAX_VALUE){
            count++;
        }
    }

    public void stackLeakByThread(){
        int count=0;
        while (count<Integer.MAX_VALUE){
            count++;
            Thread thread = new Thread(new Runnable() {
                public void run() {
                    dontStop();
                }
            });
            thread.start();
        }
    }

    public static void main(String[] args) {
        JavaVMStackOOM oom = new JavaVMStackOOM();
        oom.stackLeakByThread();
    }
}

/*
 * Exception in thread "main" java.lang.OutOfMemoryError: unable to create new native
thread
 */
```

4、方法区和运行时常量池溢出

- java.lang.OutOfMemoryError:PermGen space
- HotPot中方法区和常量池都存放在永久代。
- 运行时常量池：字符串和整型等常量池数据的存放方式在 JDK 1.7 中有一定的调整，因此表现与 JDK 1.6 会有所不同。如 intern()方法。
- 方法区溢出：方法区溢出也是一种常见的内存溢出异常，一个类要被垃圾回收器回收的条件是非常苛刻的。在经常生成大量Class的应用中，需要特别主要类的回收情况。比如：程序中使用了CGLib字节码增强和动态语言（Spring、Hibernate等主流框架）、大量JSP或动态生成JSP文件的应用（JSP第一次运行时需要编译为Java类），基于OSGi的应用（即使是同一个类文件被不同的类加载器加载也会被视为不同的类）等。

5、本机直接内存溢出

- java.lang.OutOfMemoryError
- DirectMemory容量可通过-XX:MaxDirectMemorySize指定，如果不指定，则默认与Java堆最大值（-Xmx指定）一样，可以直接通过反射获取Unsafe实例进行内存分配（Unsafe类的getUnsa()方法限制了只有引导类加载器才会返回实例，也就是设计者希望只有rt.jar中的类才能使用Unsafe的功能）。
- 由DirectMemory导致的内存溢出，一个明显的特征是在Heap Dump文件中不会看见明显的异常，如果发现OOM之后Dump文件很小，而程序中又直接或间接的使用了NIO，那就可以考虑检查一下是不是这方面的原因。

```
package com.susu.study.jvm;

import sun.misc.Unsafe;
import java.lang.reflect.Field;

/**
 * @Description: 直接内存区溢出，抛出 OOM,但是在 Heap Dump 文件中不会看见明显异常。注意使用 NIO的情
况。
 * @Args: -Xmx20M -XX:MaxDirectMemorySize=10M
 * @author: 01369674
 * @date: 2018/4/18
 */
public class DirectMemoryOOM {
    private static final int _1MB = 1024*1024;

    public static void main(String[] args) throws IllegalAccessException {
        Field unsafeField  = Unsafe.class.getDeclaredFields()[0];
        unsafeField.setAccessible(true);
        Unsafe unsafe = (Unsafe)unsafeField.get(null);
        int count=0;
        while (count<Integer.MAX_VALUE){
            unsafe.allocateMemory(_1MB);
        }
    }
}
```

一、对象已死吗

1、引用计数法

引用计数法是指给对象添加一个引用计数器，每当有一个地方引用它时，计数器值就加1；当引用失效时，计数器值就减1；任何时候计数器值为0就表示不会再被任何对象使用。

客观的说，引用计数法（Reference Counting）的实现简单，判断效率也很高，在大部分情况下都是一个不错的算法。但在主流的Java虚拟机里面没有使用引用计数法来管理内存，主要原因是它很难解决对象之间相互循环引用的问题。

引用计数无法解决下面两个对象相互引用但不可达的问题，但运行代码后发现对象实际上能够被GC。

```
public class ReferenceCountingGC {
2.    public Object instance=null;
3.    private static final int _1MB=1024*1024;
4.    private byte [] bigSize=new byte[2*_1MB];
5.
6.    public static void testGC(){
7.        ReferenceCountingGC objA=new ReferenceCountingGC();
8.        ReferenceCountingGC objB=new ReferenceCountingGC();
9.        objA.instance=objB;
10.        objB.instance=objA;
11.        objA=null;
12.        objB=null;
13.        System.gc();
14.    }
15.    public static void main(String []args){
16.        testGC();
17.    }
18.}
```

2、可达性分析算法

在主流的商用程序语言（Java、C#）的主流实现中，都是通过可达性分析（ReachabilityAnalysis）来判断对象是否存活的。这个算法的基本思路是通过一系列称为GC Roots的对象作为起始点，从这些节点开始向下搜索，搜索所走过的路径称为引用链（Reference Chain），当一个对象到GC Roots没有任何引用链（就是从GC Roots到这个对象不可达）时，则证明此对象是不可用的。

在Java语言中，可作为GC Roots的对象包括下面几种：

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

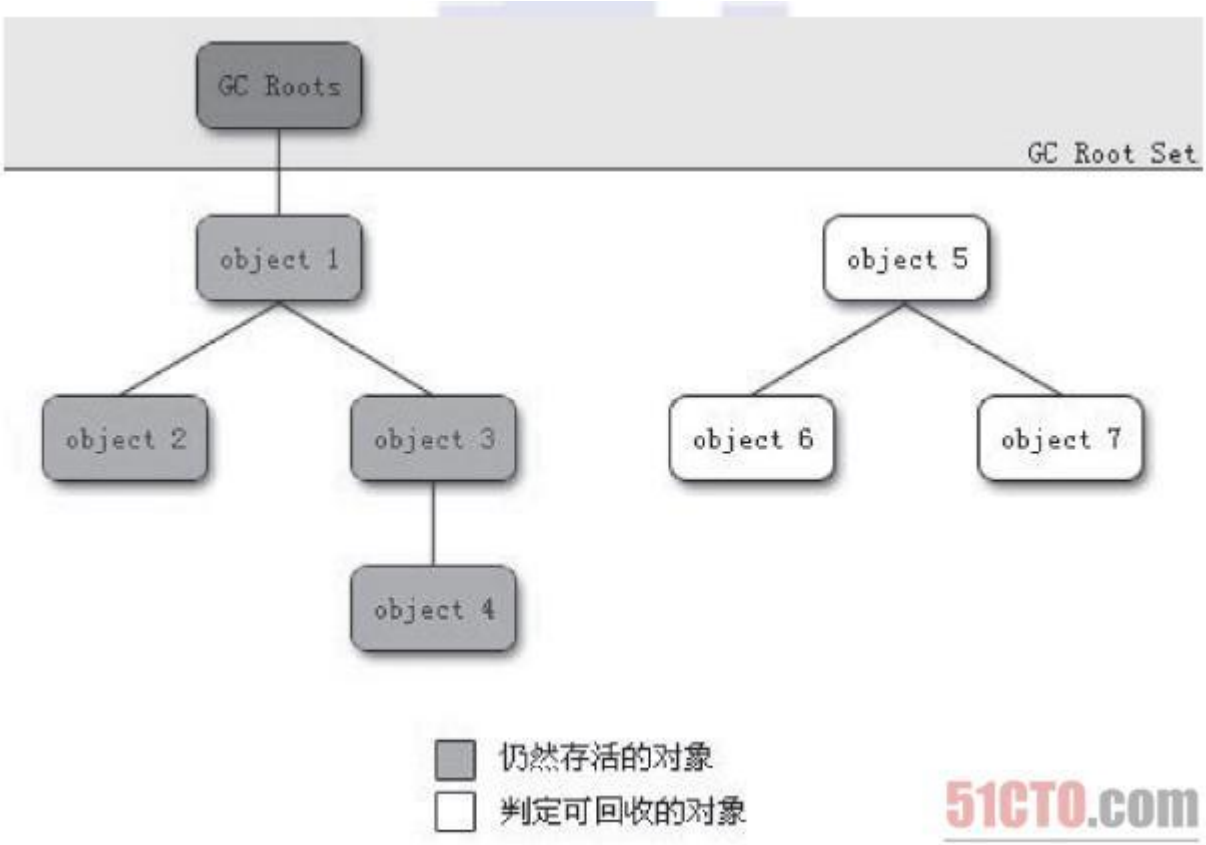


图 3-1　可达性分析算法判定对象是否可回收

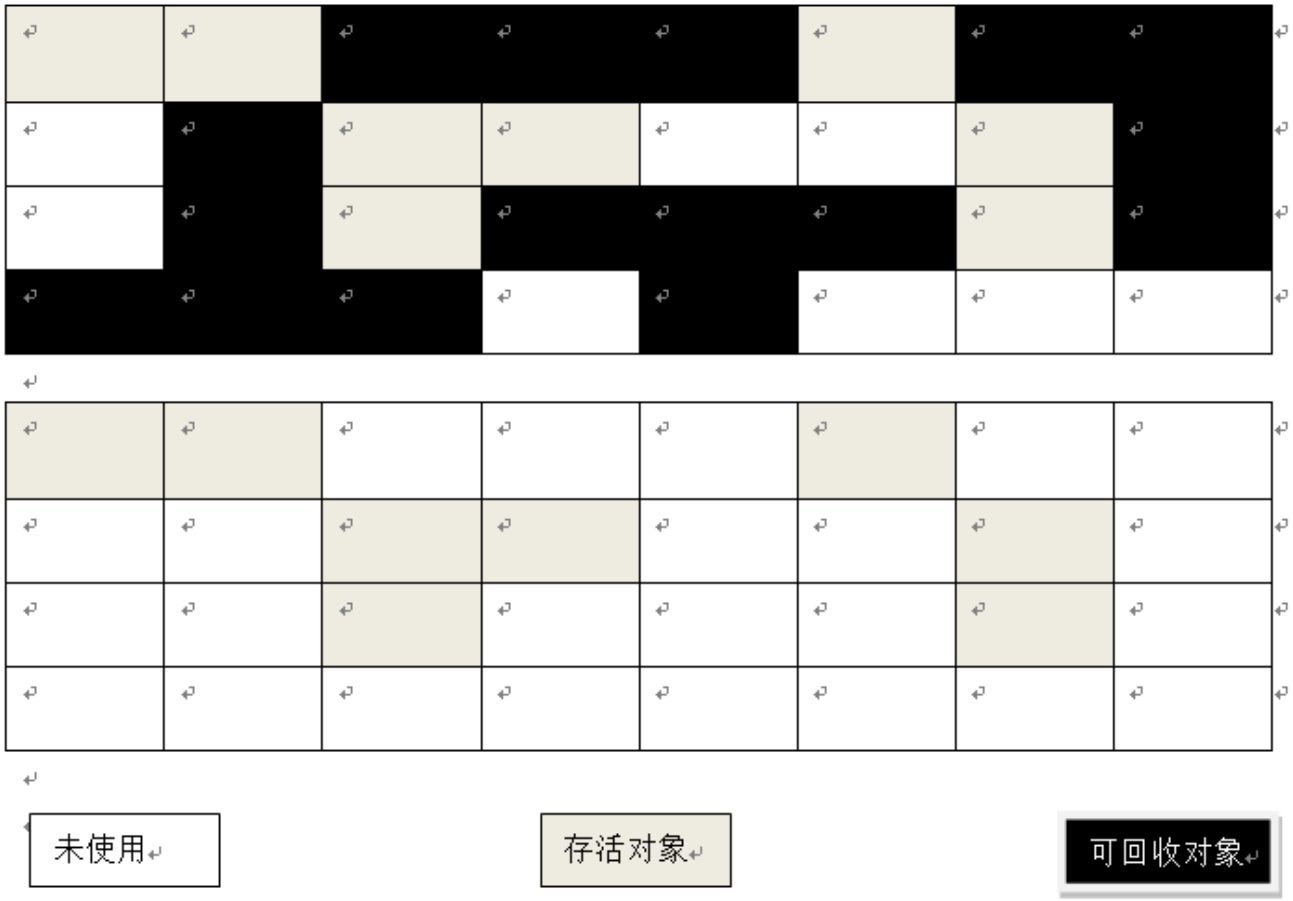
二、垃圾收集算法

1、标记-清除法

分为“标记”和“清除”两个阶段：首先标记出需要回收的所有对象，在标记完成后统一回收标记的对象，标记过程就是之前讲的通过引用计数法和可达性分析法进行判定。

它的主要不足有两个：

- 效率问题：标记和清除的效率都不高，
- 空间问题：标记清除之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在需要分配较大对象时，无法找到连续的内存空间而不得不提前触发另一次垃圾收集动作。

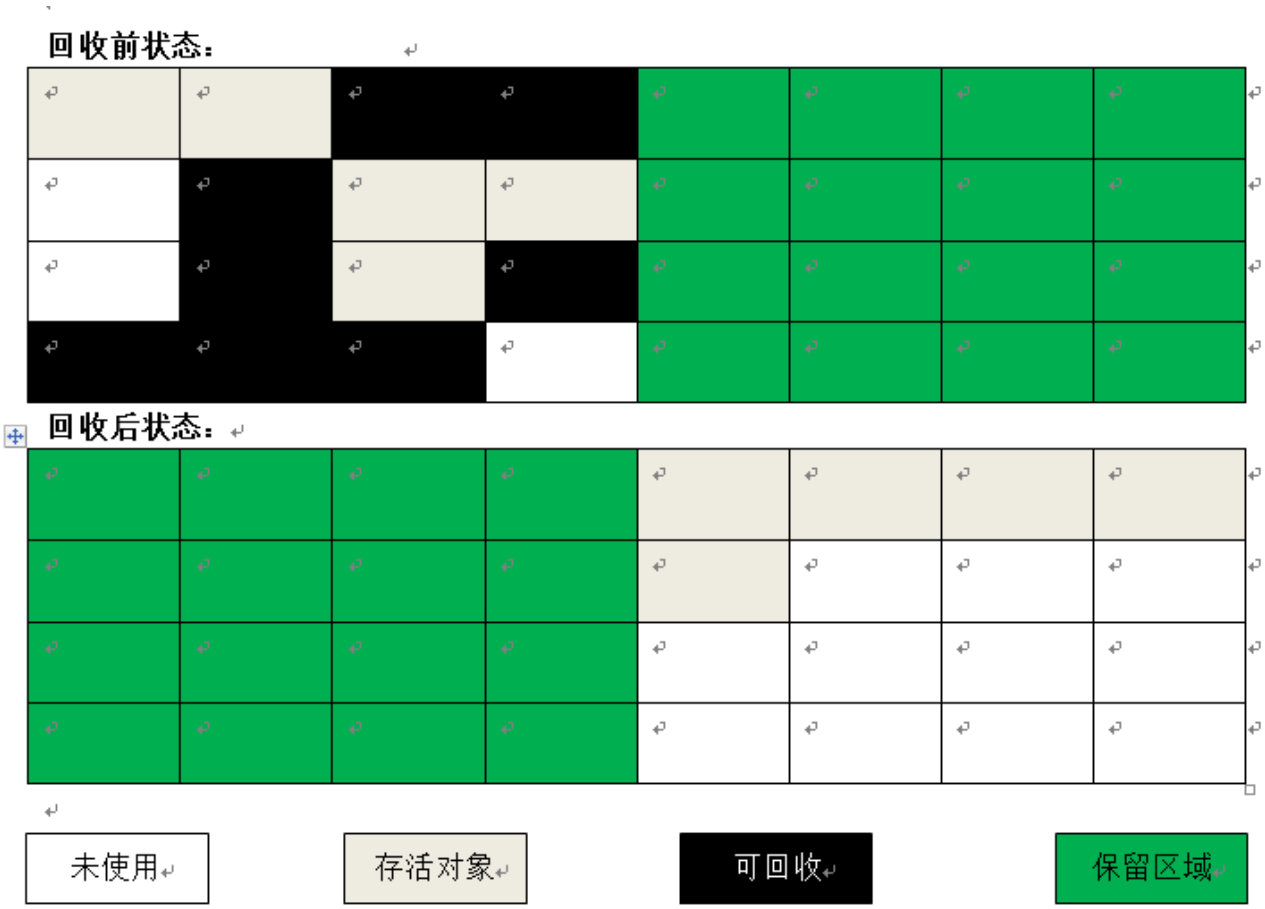


2、复制算法

将内存划分为大小相等的两块，每次只使用其中的一块。当这块内存用完了，就将还存活的对象复制到另一块内存上，然后把已使用过的内存空间一次清理掉。

优点：每次只对其中一块进行GC,不用考虑内存碎片的问题，并且实现简单，运行高效

缺点：内存缩小了一半



注：现在商用虚拟机都采用这种算法来回收新生代，因为新生代中98%的对象都是朝生夕死的，所以并不需要1：1的比例来划分内存空间，而是将内存分为一块较大的Eden空间和两块较小的Survivor空间，每次使用Eden和一块Survivor空间。当回收时，将Eden和Survivor中开存活的对象一次性复制到另一块Survivor空间上，最后清理掉Eden和刚才使用过的Survivor空间。

HotSpot虚拟机默认Eden和Survivor的大小是8:1，也就是每次新生代可用空间是整个新生代容量的90%（80%+10%），只有10%的内存会被“浪费”。当然，98%可回收只是一般的情况的数据，我们没有办法保证每次回收都只有不多于10%的对象存活，当Survivor空间不够用时，需要依赖其他内存（这里指老年代）进行分配担保（Handle Promotion）。

内存的分配担保类似于银行贷款，如果我们的信誉好。在98%的情况下都能按时偿还，于是银行可能默认我们下次也能按时偿还贷款，只需要有一个担保人能保证如果我不能还款时，可以从他的账户扣钱，那银行就认为没有风险了。内存的分配担保也一样，如果另外一块Survivor空间没有足够空间存放上一次新生代收集下来的存活对象时，这些对象将直接通过分配担保机制进入老年代。

3、标记-整理法

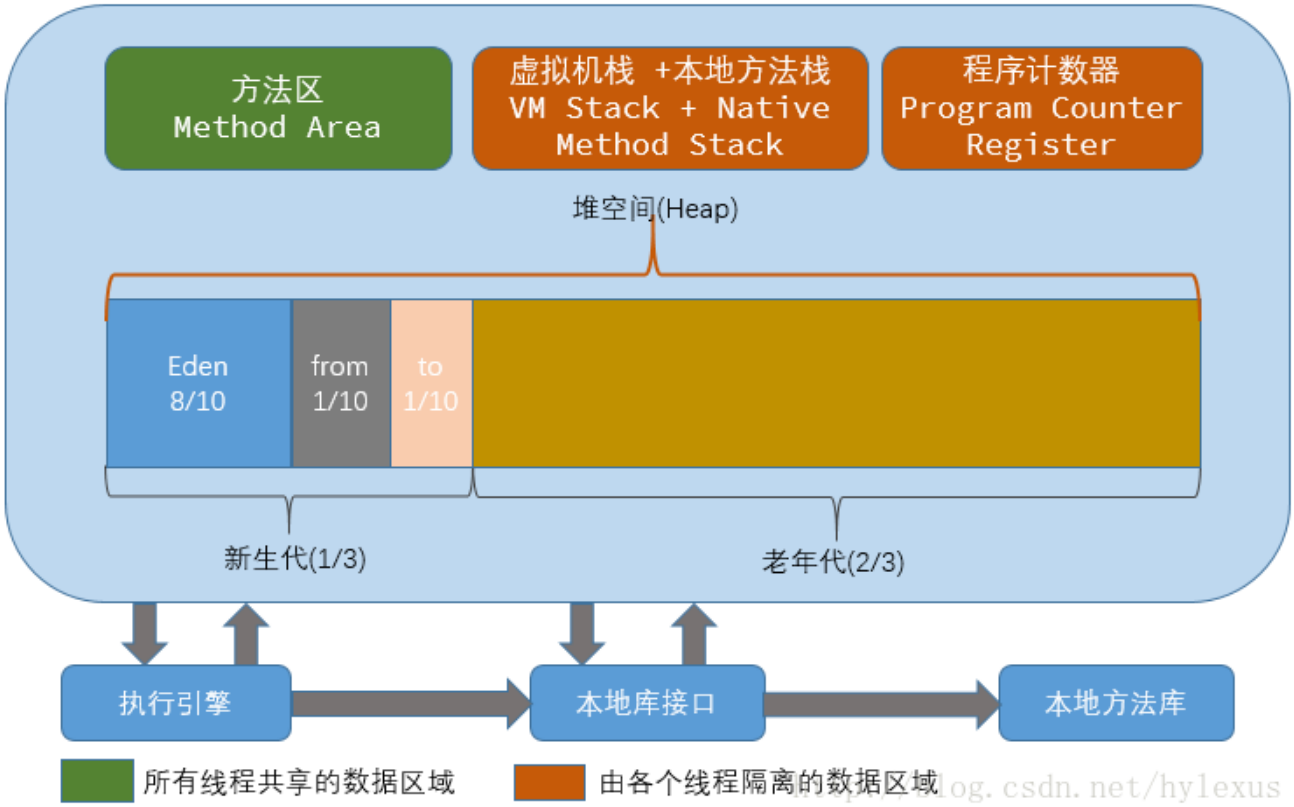
复制收集算法在对象存活率较高时就要进行较多的复制操作，效率将会变低。更关键的是，如果不想浪费另外50%的空间，就需要额外的空间进行分配担保，以应对被使用的内存中所有对象都100%存活的极端情况，所以老年代一般不能直接选用这种算法。

根据老年代的特点，有人提出了另外一种“标记-整理”（Mark-Compact）算法，标记过程任然与“标记-清除”算法的标记过程相同，但后续步骤不是对可回收对象进行直接清理，而是让所有活的对象都移动到另一端，然后直接清理掉端边界以外的内存，

4、分代收集算法

当前商业虚拟机的垃圾收集都采用“分代收集”（GenerationalCollection）算法，这种算法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堆划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合的收集算法。在新生代中每次都有大量的对象死去，只有少量存活，那就采用复制算法，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、没有额外空间对它们进行分配担保，就必须使用“标记-清理”或“标记-整理”算法来进行回收。

5、HotPot 分代收集算法



对象将根据存活的时间被分为：年轻代、年老代、永久代。

年轻代:

所有新生成的对象首先都是放在年轻代的。年轻代的目标就是尽可能快速的收集掉那些生命周期短的对象。年轻代分三个区。一个Eden区，两个Survivor区(一般而言)。大部分对象在Eden区中生成。当Eden区满时，还存活的对象将被复制到Survivor区（两个中的一个），当这个Survivor区满时，此区的存活对象将被复制到另外一个Survivor区。同时，根据程序需要，Survivor区是可以配置为多个的（多于两个），这样可以增加对象在年轻代中的存在时间，减少被放到年老代的可能。

年老代:

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

持久代:

用于存放静态文件，如Java类、方法等。

Scavenge GC

一般情况下，Eden空间满时，就会触发Scavenge GC，清除Eden区非存活对象，并且把尚且存活的对象移动到Survivor区。然后整理Survivor的两个区。Eden区的GC会频繁进行，速度也很快。

Full GC

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

附录：

1. 程序计数器

- 程序计数器（Program Counter Register） 是一块较小的内存空间，它可以看作是当前线程所执行的字节码的行号指示器。在虚拟机的概念模型里，字节码解释器工作时就是通过改变这个计数器的值来选取下一条执行字节码指令。
- 每条线程都有一个独立的程序计数器。
- 如果执行的是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地址。如果是native方法，计数器为空。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。

2. Java虚拟机栈

- 同样是线程私有，描述Java方法执行的内存模型：每个方法在执行的同时都会创建一个栈帧（Stack Frame）用于存储局部变量表、操作数栈、动态链接、方法出口等信息。一个方法对应一个栈帧。
- 局部变量表存放了各种基本类型、对象引用和returnAddress类型（指向了一条字节码指令地址）。其中64位长度long 和 double占两个局部变量空间，其他只占一个。
- 规定的异常情况有两种：1.线程请求的栈的深度大于虚拟机所允许的深度，将抛出StackOverflowError异常；2.如果虚拟机可以动态扩展，如果扩展时无法申请到足够的内存，就抛出OutOfMemoryError异常。

3. 本地方法栈

- 和Java虚拟机栈很类似，不同的是本地方法栈为Native方法服务。

4. Java堆

- 是Java虚拟机所管理的内存中最大的一块。由所有线程共享，在虚拟机启动时创建。堆区唯一目的就是存放对象实例。
- 堆中可细分为新生代和老年代，再细分可分为Eden空间、From Survivor空间、To Survivor空间。
- 堆无法扩展时，抛出OutOfMemoryError异常

5. 方法区

- 所有线程共享，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。
- 当方法区无法满足内存分配需求时，抛出OutOfMemoryError

6. 运行时常量池

- 它是方法区的一部分，Class文件中除了有类的版本、字段、方法、接口等描述信息外，还有一项是常量池（Const Pool Table），用于存放编译期生成的各种字面量和符号引用。并非预置入Class文件中常量池的内容才进入方法运行时常量池，运行期间也可能将新的常量放入池中，这种特性被开发人员利用得比较多的便是String类的intern()方法。
- 当方法区无法满足内存分配需求时，抛出OutOfMemoryError

7. 直接内存

- 并不是虚拟机运行时数据区的一部分，也不是Java虚拟机规范中定义的内存区域。
- JDK1.4加入了NIO，引入一种基于通道与缓冲区的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。因为避免了在Java堆和Native堆中来回复制数据，提高了性能。
- 当各个内存区域总和大于物理内存限制，抛出OutOfMemoryError异常。