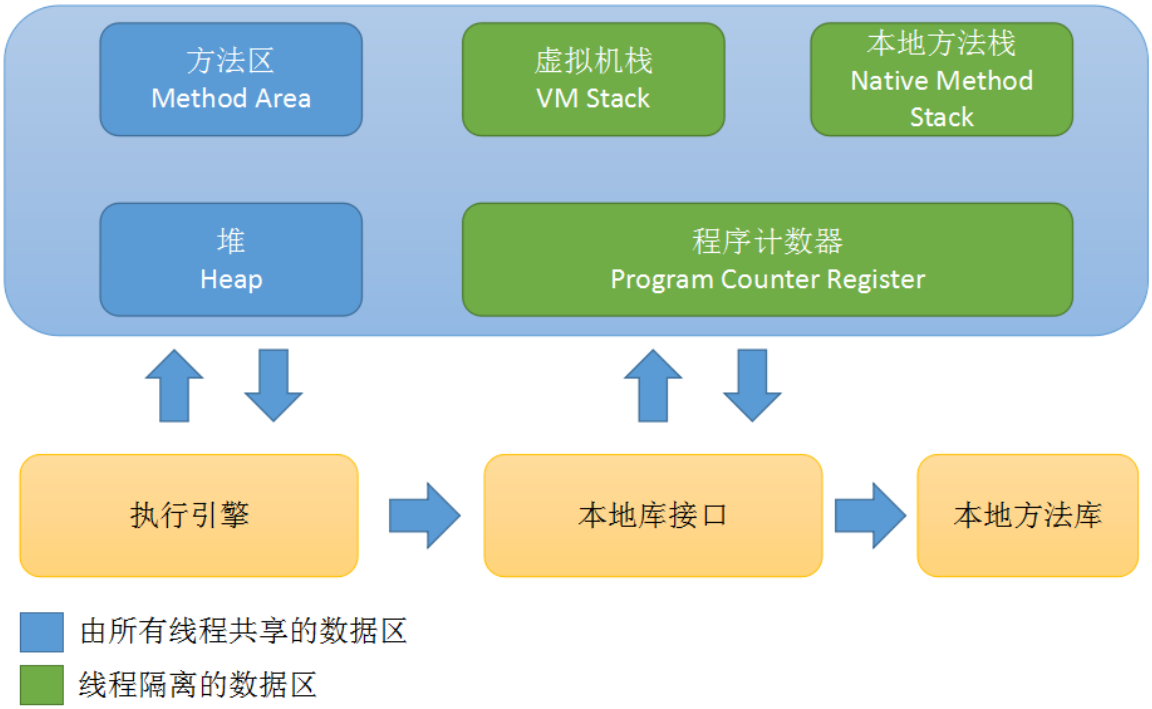
文章已收录我的仓库：Java学习笔记与免费书籍分享

Java垃圾回收机制与垃圾收集器

前言



首先来思考一个问题：

- 哪些内存需要回收？

  首先对于栈空间里的内存是无须回收的，因为这一部分内存会随着退出作用域而自动释放，**事实上，主要回收的内存是方法区中废弃的常量和不再使用类元数据与堆中未被引用的对象，这一部分内存我们定义为“垃圾”。**对于方法区中内存的回收是在回收堆内存时由虚拟机顺带回收的。

接下来，让我们一起来学习JVM是如何回收垃圾的，在此之前，约定在无特殊声明的情况下，默认的虚拟机为当前主流的HotSpot。

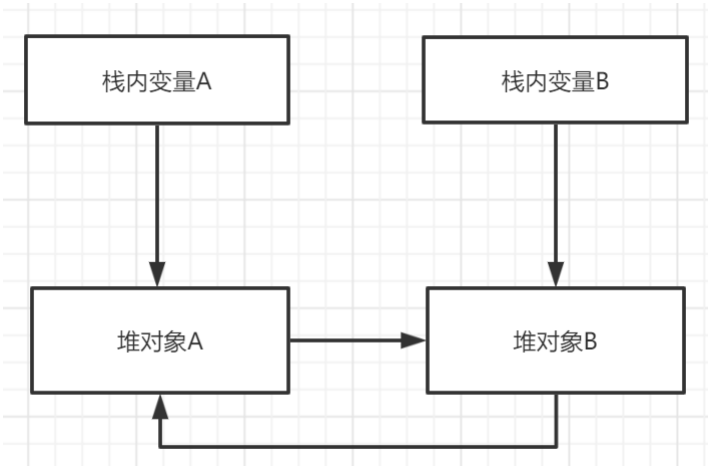
判定对象是否存活(标记)

只有当一个对象不再被使用后，JVM才可以将其回收，那么如何判定一个对象是否存活呢?

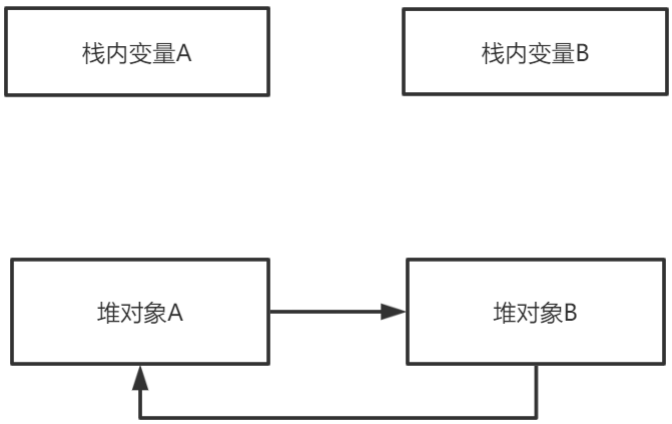
引用计数法

引用计数法是一个非常经典的方法，其原理也很简单，即为每一个对象维护一个引用计数，每当该对象被引用时，则自增引用计数，当局部变量修改引用时或局部变量被销毁，递减相应对象的引用计数，当一个对象的引用计数为0时，则代表该对象无引用，判定其为死亡状态。

引用计数的优点在于其实现简单，判定效率也高，例如C++中的智能指针或Python中的垃圾回收都采用了这种方法，但Java几乎所有的主流虚拟机都没有采用这个简单的方法，这是因为引用计数无法保证完全正确，例如经典的**循环引用**问题。



若此时局部变量放弃对对象的引用，这两个堆对象仍然无法被回收，因为它们之间存在互相引用：

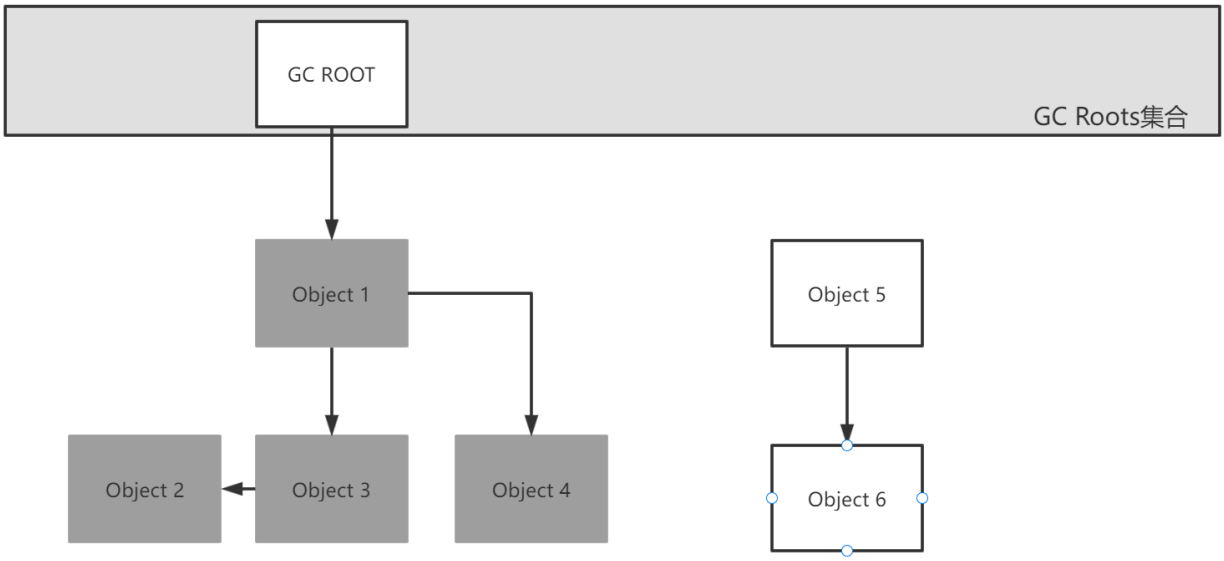



```
public class Test {
    Object instance = null;
    public static void main(String[] args) {
            var a = new Test();
            var b = new Test();
            a.instance = b;
            b.instance = a;
            /*即时移除对对象的引用，引用计数方法仍然无法回收对象*/
            a = null;
            b = null;
    }
}
```

可达性分析

算法思想

该算法通过一系列被称为“GC Roots”的根对象作为起点集合，依次扫描这些节点，根据引用关系向下搜索，搜索过程走过的路径称为引用链，如果某一个对象从任何一个GC Roots开始到该对象都不可达时，则判定该对象是不可能再被使用的。



例如上图中 Object 5 和 Object 6 就是已死的对象，通过可达性分析算法则可以避免对象循环引用的问题，例如即时对象5和对象6互相引用，由于与GC Roots之间没有引用链，它们仍然会被回收。

固定可作为GC Roots的对象包括以下几种：

- 本地栈中具有引用的对象：例如 `Object A = new Object();`，这里的变量A就是一个GC ROOTS，可以看一看前面引用计数的图，这里的A就是一个栈变量，但其指向了一个堆变量，当退出作用域时，下一次扫描将不会检测到该引用链，具体看下面的OopMap内容。
- 具有引用的静态类型对象：例如 `public static Object obj = new Object() ;`，这里的obj也会被作为GC ROOTS。
- 方法区中具有常量引用的对象：例如 `private String str = "Hello World!";`，此时str引用了常量池中的对象，str会被作为GC ROOTS以判定常量池中对象是否废弃。
- 在本地方法栈中JNI引用的对象：该对象可能是一个C或C++指针，不属于Java本地代码，这类对象如果引用了Java堆中对象同样也会被作为GC ROOTS。
- Java虚拟机内部的引用：例如基本数据类型引用了对于的class对象。

如果总结来看，除了堆中对象不会作为GC ROOTS，几乎所有引用了堆对象或常量池的对象都会被作为GC ROOTS，但这也不是绝对的，即使是堆中对象也是可能会作为根对象的，这在后面将跨代引用时会提到。

算法步骤

可达性分析主要采用三色标记法，按照对象是否被访问分成三种颜色：

1. 白色：表示尚未遍历该对象，如果结束时对象仍为白色，则表示对象不可达。
2. 灰色：表示已经遍历该对象，但还没有遍历该对象的引用，下一次遇到该对象时会继续遍历该对象的引用。

3. 黑色：已经遍历该对象，并且所有引用都扫描过了，下一次遇到该对象时会直接跳过。

整个算法用代码表示大概就像下面这样(自己模拟的，非官方)：

```
class AccessibilityAnalysis {
    /* key 为对象， value 为对象的引用集合*/
    private Map<Object, List<Object>> objMap = null;
    /* key 为对象， value 为对象的颜色, 0-白色 1-灰色 2-黑色*/
    private Map<Object, Integer> colors = null;
    /*保存结果的集合*/
    private Set<Object> result = null;

    public AccessibilityAnalysis(Map<Object, List<Object>> objMap,
Map<Object, Integer> colors) {
        this.objMap = objMap;
        this.colors = colors;
        this.result = new HashSet<>();
        for (Object obj : objMap.keySet()) {
            this.colors.put(obj, 0);
        }
    }

    public Set start() {
        for (Object obj : objMap.keySet()) {
            /*对每个根节点开始算法*/
            dfs(obj, colors);
        }
        for (var it : colors.entrySet()) {
            //这里标记可达的，也可以标记不可达，排除掉GC ROOTS本身
            if (it.getValue() == 2 && !objMap.containsKey(it.getKey())) {
                result.add(it.getKey());
            }
        }
        return result;
    }

    public void dfs(Object obj, Map<Object, Integer> colors) {
        int color = colors.get(obj);
        /*黑色或灰色表示已搜索过，略过*/
        if (color == 2 || color == 1) {
            return;
        }
        /*开始搜索，设置为灰色*/
        colors.put(obj, 1);
        for (Object nextObj : objMap.get(obj)) {
            dfs(nextObj, colors);
        }
        /*搜索完毕，设置为黑色*/
        colors.put(obj, 2);
        return;
    }
}
```

上述代码是在单线程模式下运行的，在全部节点遍历完之后再获取结果，事实上只需要两种颜色即可；但JVM允许并发标记，此时每个线程遍历完之后就会尝试标记，必须要将灰色节点信息传递给其他线程，以免其他线程直接标记该节点。

对象复活

一个对象即使被标记成“死亡”状态也不是一定会被回收的，这涉及JVM回收垃圾的具体步骤。

要真正回收一个垃圾，至少要经历两次标记过程，第一次则运行三色标记法对对象进行标记，随后JVM会进行一次筛选，判定该对象是否有必要执行**finalize**方法，**判定的依据是该对象是否重写方法并且没有执行过该方法**，如果没有必要执行，则直接进行回收；如果确实需要执行，该对象会被放置在F-Queue队列中，由JVM创建一条低优先级的线程去执行，**稍后**JVM会对队列中的对象进行小规模的标记，JVM统计在finalize中的GC ROOTS而避免重新遍历所有GC ROOTS，如果对象在finalize方法中重新关联了一条引用链，那么该对象将会逃脱第二次标记，从而复活。

要注意JVM并不等待finalize成功执行后才去标记，而是等待一小段时间后直接开始标记，而不管方法是否被执行，JVM不能接受一个“垃圾”还疯狂的占用程序运行资源。此外，根据上述判定依据，一个对象不能重复复活，因为finalize方法只会被执行一次。

```
public class Test {
    private static Test test = null;

    public void say() {
        System.out.println("我还活着");
    }

    @Override
    protected void finalize() throws Throwable {
        test = this;
    }

    public static void main(String[] args) throws InterruptedException {
        test = new Test();
        test = null;
        System.gc();
        /*finalize优先级很低，等待finalize执行*/
        Thread.sleep(200);
        if (test == null) {
            System.out.println("kill");
        } else {
            test.say();
        }


        test = null;
        System.gc();
        Thread.sleep(200);
        /*不允许第二次逃脱*/
```

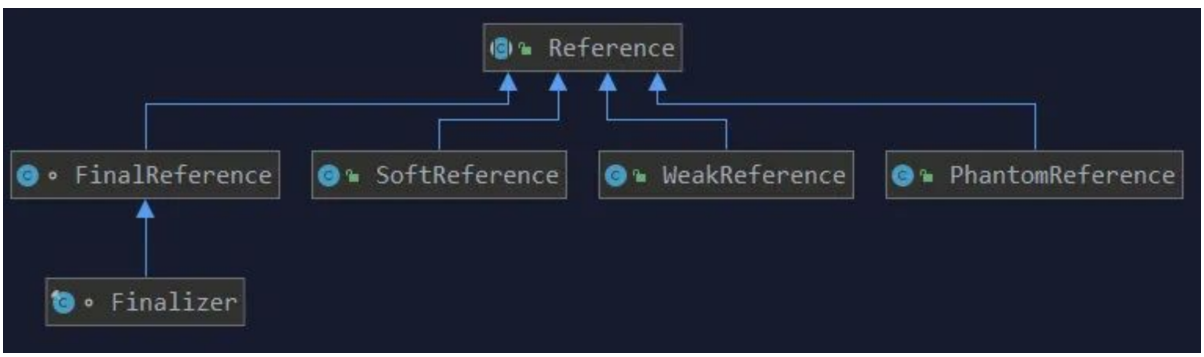
```
        if (test == null) {
            System.out.println("kill");
        } else {
            test.say();
        }
    }
}

/*
sout:
我还活着
kill
*/
```

不过，finalize方法已经被废弃了，这会影响程序性能，try - finally 是更好的选择。

引用概念的完善

垃圾收集过程十分依赖于引用链，但在之前，一个对象只有“被引用”和“未被引用”两种状态，对于一些“食之无味，弃之可惜”的对象却无能为力，这与我们生活中的某些现象很类似，例如你还有一些作业要写，你想把主要的任务完成，但对于一些次要的任务，你期望如果有时间就写，没时间就算了，这是一种动态的抉择，而不是处于一定写或一定不写的状态。Java中某些时候也需要这种对象，我们期望还有空间可用时就保留它，否则就清楚它，为此，Java完善了引用的概念：

- 强引用，绝大多数的引用都是强引用，例如“var obj = new Object();”这类引用关系，强引用只要存在，垃圾收集器就永远不会收集该对象。
- 软引用，软引用即我们上面所描述的一种关系，表示对象有用但非必须，如果JVM还有空间则不会清除对象，一旦JVM空间不足则会回收这部分内存。可以通过把引用赋值给SoftReference来将其转换为软引用，例如“SoftReference sr = new SoftReference(obj);”，此时obj便成为了软引用，后续可以通过sr.get()获得对象。
- 弱引用，这是比软引用更弱的引用，被弱引用关联的对象只能只能存活到下一次GC开始，下一次GC开始时，无论如何都会自动回收掉该内存。类似于软引用，Java提供了WeakReference类来实现弱引用。
- 虚引用，也称“幽灵引用”，一个对象持有虚引用时，就好比没有任何引用一样，随时都可能被回收，持有虚引用的唯一目的就是在该对象被GC时收到一个系统通知，虚引用可以通过PhantomReference类来实现。



垃圾回收算法

上述判定对象存活的过程被称为**标记过程**，当我们标记一个对象后，接下来就要开始真正的垃圾回收过程了。

不过在这之前，要说说经典的分代假说理论，毕竟，所有的垃圾回收算法都是基于这些理论的：

- 弱分代假说（Weak Generational Hypothesis）：绝大多数对象都是朝生夕灭的。
- 强分代假说（Strong Generational Hypothesis）：熬过越多次垃圾收集过程的对象就越难以消亡。
- 跨代引用假说（Intergenerational Reference Hypothesis）：跨代引用相对于同代引用来说仅占极少数。

正是由于这些假说，JVM将**Java对象分为新生代(young)和老年代(old)，所有对象初始都作为新生代(下面对提到某些大对象除外)，当新生代熬过一定次数的垃圾收集后就会进入老年代，新生代与老年代被放置在不同的区域，根据不同年代采用不同的算法收集垃圾。**

标记 - 清除法

该算法分为标记和清除两个部分，首先标记所有不可达的对象，然后直接将这些对象清除，要注意这里的清除并不是真的置空，而是取消分配状态，然后将空闲块信息保存至空闲链表中，等到下次再分配。

清除的操作类似于操作系统中的空闲空间分配与回收，读者可自行了解空闲空间分配算法。

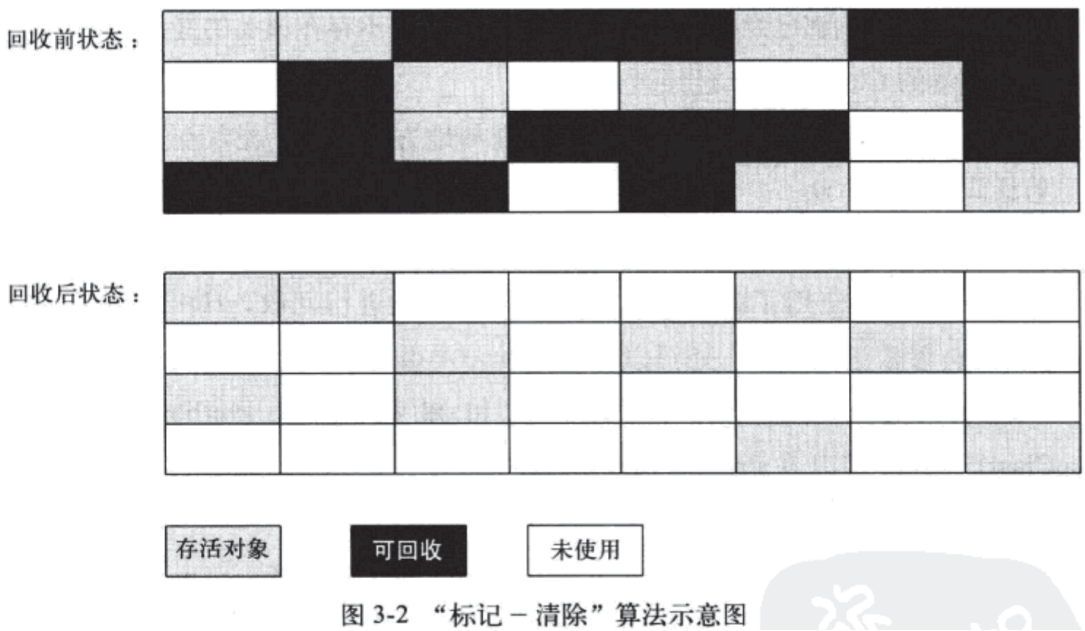


图 3-2 “标记 – 清除”算法示意图

标记清除算法的优点是实现简单，但缺点也是很明显的，根据分代假说，绝大多数对象都是朝生夕灭的，这意味着需要更多的时间去标记与清除这些对象，而且，如上图

所示，回收后很容易会导致外部碎片，这可能导致下一次分配难以进行。

标记 - 清除法既可以清理新生代也可以用以清理老年代，但由于其种种缺点，标记清除法事实上已经很少被单独使用了。

标记 - 复制法

标记 - 复制法解决了标记清除法的缺点，**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，当这一块的内存用完了，就将还存活着的对象复制到另外一块上面，然后再把已使用过的内存空间一次清理掉。**

如果内存中多数对象都是存活的，这种算法将会产生大量的内存间复制的开销，但对于多数对象都是可回收的情况，**算法需要复制的就是占少数的存活对象**，而且每次都是针对整个半区进行内存回收，分配内存时也就不用考虑有空间碎片的复杂情况，只要移动堆顶指针，按顺序分配即可。**这样实现简单，运行高效，几乎完美解决了标记 - 清除法带来的缺点。**

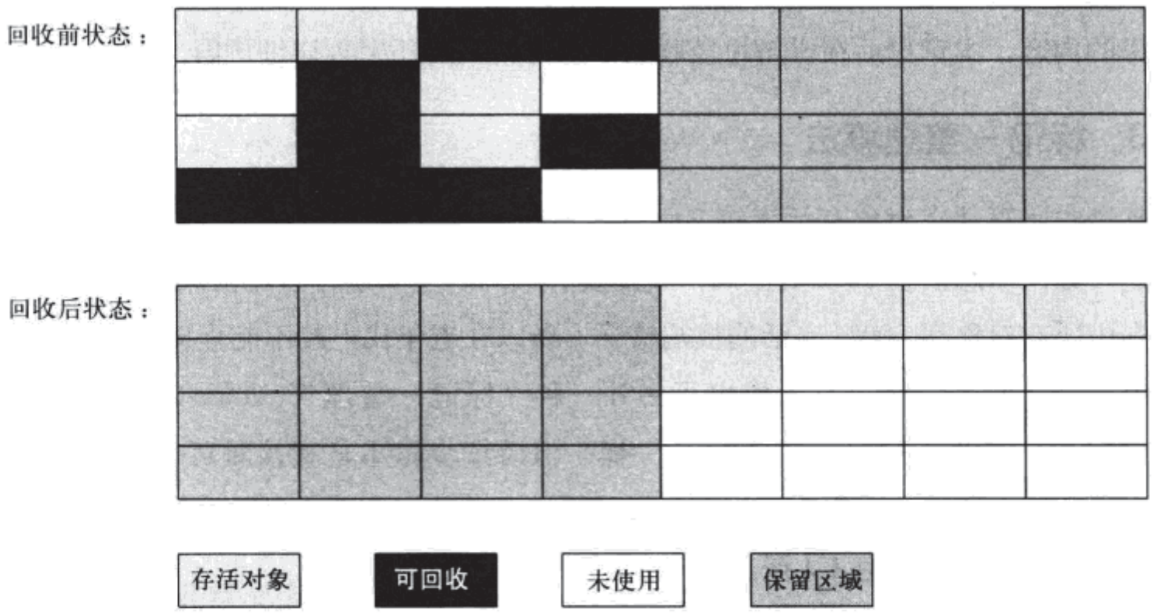


图 3-3　复制算法示意图

其缺陷也显而易见，这种复制回收算法的代价是将可用内存缩小为了原来的一半，**浪费了大量的空间**。为解决该问题，IBM公司曾仔细研究过对象存活的比例，发现大部分时候，仅有2%的对象存活下来，由于标记复制法复制的是存活对象，因此完全没有必要按照1：1的比例来分配空间，**为此有一种更优雅的标记复制法称为“Appel式回收法”，该算法将新生代区按照 8：1：1的比例分割。**

具体的，将新生代分为一块 Eden 区和 两块 Survivor 区，Eden 区与 Survivor 区大小比例为 8：1，当我们分配空间时，只是用 Eden 区和其中一块 Survivor 区，而保留另一块幸存者区作为复制空间，当发起垃圾回收时，将作为分配空间的 Eden 区和其中一块 Survivor 区中的存活对象复制到另一块 Survivor 区中，然后一次性清除 Eden 区和其中一块 Survivor 区(含不可达对象的区)，现在，该 Survivor 又被作为复制区域，等待下一次回收...

在并发复制过程中，对内存的分配会暂时放置在存活区中，这就保证了复制阶段不会出现错误。

但要注意，并不是所有的程序都能保证存活对象能全部装进 Survivor 区，当这种情况发生时，必须要借助老年代的空间以作为担保，**标记复制法必须要额外空间担保，因此，标记复制法通常只用于回收新生代。**如无特殊说明，后续标记复制法说的都是“Appel式回收法”。

这里还有一个问题，就是对于大对象如何解决？对于较大对象的复制是非常缓慢的，可能会导致较长时间的停顿，给予用户不好的体验，对于这种问题，通常大部分JVM规定超过一定大小的对象将直接进入老年代。

标记 - 整理法

老年代由于没有额外的空间担保，并且存活对象过大，复制效率低，无法使用标记复制算法，通常使用标记整理法。

由于没有额外的空间，因此只能在老年代内部实现，为了避免像标记清除法那样产生大量的外部碎片，标记整理法通过将标记存活的对象向空间一侧移动，然后一次性情况存活对象边界指针以外的区域。

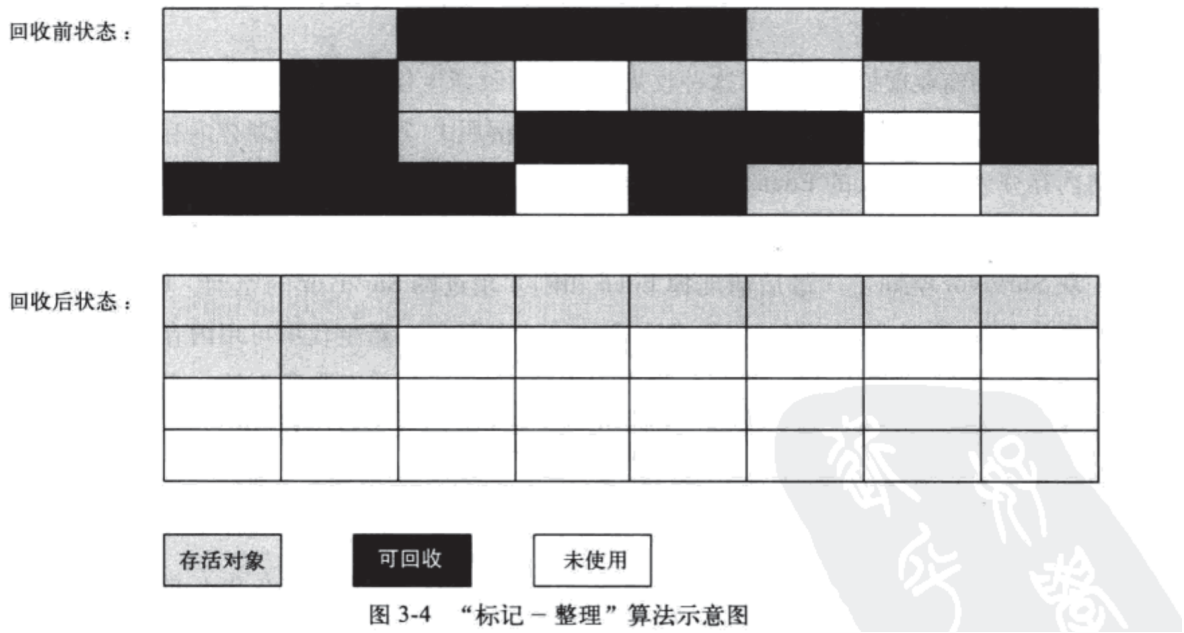


图 3-4 “标记－整理”算法示意图

由于JVM是基于页实现的，因此标记整理算法的实现是相对简单的，存活对象可以直接覆盖点死亡对象页，虽然实现简单，但由于涉及到大量内存或磁盘读写，并且由于老年代的特殊性(存活对象较多)，这会异常缓慢(我们不能移动死亡对象去整理，因为那样的话存活对象直接仍然有大量的碎片)，在整理的过程中，由于涉及到整个老年代的空间读写，JVM无法支持分配对象，因此必须要暂停所有线程(ZGC打破了牢笼)，由于暂停时间长，这种暂停被设计者形象的描述为“Stop The World”。

由于整理大对象需要较多时间，实际中也通常采用标记清除 + 标记整理法结合使用。

标记整理法有优点也有缺点。何时使用取决于具体的环境。

具体细节

JVM主动发起的几乎都是新生代的垃圾回收，而只有当空间不足时才会发起一次老年代的回收或者是

如何确定GC Roots？

难道虚拟机要遍历全部的栈空间以及方法区去确定GC ROOTS吗，这样做效率未免太低了些，HotSpot虚拟机采用一组数据结构OopMap(Ordinary Object Pointer，普通对象指针)，**OopMap是运行时生成的**，对于对象之间的引用，**虚拟机在类加载后就会把对象内所有数据计算出来**，如果是引用类型，则保存在OopMap中；而对于栈到堆的引用，**虚拟机在即时编译阶段(处于运行期)会扫描所有栈帧之间的数据**，并把引用保存到OopMap之中(**只会在安全点记录这一刻的栈帧信息**)，然后，JVM便可以直接扫描OopMap以确定GC ROOTS，所以，OopMap为JVM提供了准确式的垃圾收集。

来思考一下生成OopMap是否可以与用户线程并发运行，答案是可以，但实际成本过高，导致主流虚拟机几乎都是暂停用户线程生成OopMap以开始垃圾收集。要清楚即时编译器会扫描会扫描所有栈帧中的所有数据，考虑并发的情形，当程序进入一个方法时，即时编译器开始收集当前栈帧的所有信息，当信息被收集完成，此时程序恰好退出方法区，这意味着栈帧信息发生了变化，该方法出栈，这导致即时编译器收集到的全是无用、甚至是错误的信息，不仅增加空间成本，还要额外去处理这些无用的、错误的信息，这使得成本代价过于昂贵。

因此，在开始垃圾收集之前，必须要“STW”即暂停用户线程以确定GC ROOTS，这就引入了另一个问题，即何时暂停或者说在程序代码的哪个位置暂停？

安全点和安全区域

我们在确定 GCROOTS 时已经说明了暂停执行的必要性，由于暂停的原因之一是为了保证垃圾收集的安全性，**因此称程序暂停的位置为安全点。**有了安全点的设定，现在可以添加一条新的规则：**程序必须要执行到安全点的位置才可以发起垃圾收集，而不能随意发起。**这条规则隐含了一条信息，即在分配内存代码处必须要设置安全点以防内存不够发起垃圾回收。

那么除了在发起内存分配时设置安全点，其他地方该如何设置安全点呢？既不能让程序执行太长时间，这可能会导致内存占用率飙升，也不能频繁的执行，否则会影响性能。

> 安全点的选定基本上是以程序“是**否具有让程序长时间执行的特征**”为标准进行选定的——因为每条指令执行的时间都非常短暂，程序不太可能因为指令流长度太长这个原因而过长时间运行，“长时间执行”的最明显特征就是指令序列复用，例如方法调用、循环跳转、异常跳转等，所以具有这些功能的指令才会产生Safepoint。，

这是书上的原文，可能比较难理解，翻译一下就是为了避免线程长时间无法进入安全点，故而在长时间代码片段前后设置安全点。

例如一个超大的循环，一半会在循环回跳（重新进入循环或者说一次循环的末尾）时设定安全点，避免线程因为长时间无法到达安全点而导致停顿时间过长。

安全点一般都是由前端编译器生成的，前端编译器会插入关于安全点的字节码，在进入JVM之前，安全点就已经固定了。

一般会在如下几个位置选择安全点：

i. 循环的末尾
ii. 方法临返回前
iii. 调用方法之后
iv. 抛异常的位置

再来看**OpenJDK官方对安全点的定义：**

- 安全点是在程序执行期间的所有`GC Root`已知并且**所有堆对象的内容一致**的点。
- 从全局的角度来看，所有线程必须在GC运行之前在安全点阻塞。
- 从本地的角度来看，安全点是一个显着的点，它位于执行线程可能阻止GC的代码块中。 **大多数调用点都能当做安全点。**

总的来说，安全点就是指，当线程运行到这类位置时，**堆对象状态是确定一致的，引用关系不会轻易发生变化**，JVM可以安全地进行操作，如GC，偏向锁解除等。

现在问题是，由于GC发生时需要暂停线程(不只是单线程收集器，即使是多线程收集器某个阶段也需要暂停线程)，那么如何在 GC 发生时，使所有线程都跑到最近的 Safe Point 上再停下来？

主要有两种方式：

- 抢断式中断：在 GC 发生时，首先中断所有线程，如果发现线程未执行到 Safe Point，就恢复线程让其运行到 Safe Point 上，这种方式是由JVM主动控制的。

- 主动式中断：在 GC 发生时，JVM不直接操作线程中断，而是简单地设置一个标志，让各个线程执行时主动轮询这个标志，发现中断标志为真时就自己中断挂起，然后通知JVM自己到达安全点，一旦所有线程都准备好了，JVM就可以开始垃圾收集了。中断标志被设置在安全点处，因此当线程运行到安全点时，便会挂起自己。

安全区域又是什么？

Safe Point 是对正在执行的线程设定的。如果一个线程处于 Sleep 或中断状态，它就不能响应 JVM 的中断请求，再运行到 Safe Point 上。因此 JVM 引入了 Safe Region。**Safe Region 是指在一段代码片段中，引用关系不会发生变化**。在这个区域内的任意地方开始 GC 都是安全的。线程在进入 Safe Region 的时候先标记自己已进入了 Safe Region，等到被唤醒时准备离开 Safe Region 时，先检查能否离

开，如果 GC 完成了或未开始，那么线程可以离开，否则它必须等待直到收到安全离开的信号为止。

这意味着，休眠或中断代码附近必须被标志为安全区域，安全区域可以是一条代码(指令)，也可以是多条代码(指令)，如果不只一条代码，线程苏醒时可以继续运行，但运行到安全区域边界时，必须要判定GC是否正在进行中，是的话则不允许走出安全区域。

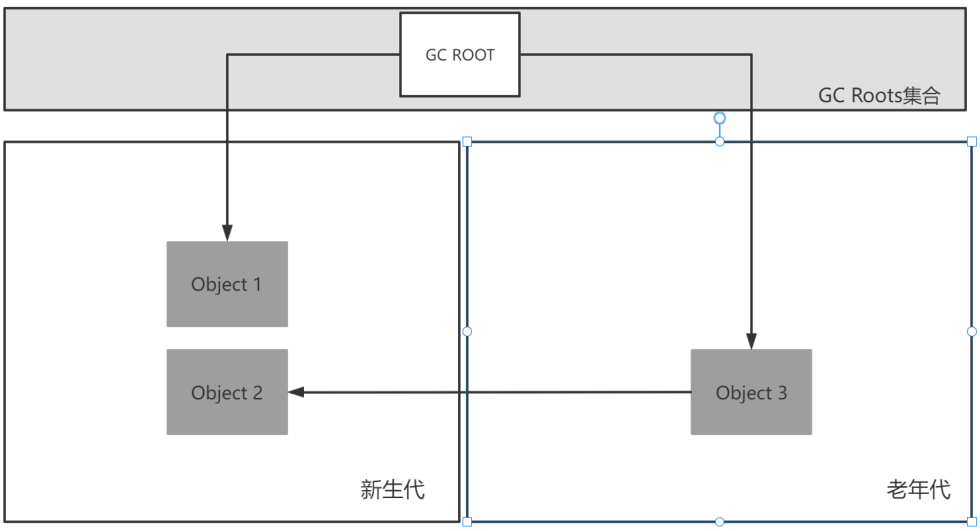
何时开始GC

在介绍了安全点之后，你可能会以为线程运行到安全点时就会发生GC，但其实并不是如此，安全点与GC之间是必要不充分关系，GC的发生只有在内存不足下才会发生，除非你手动调用。

对于经典垃圾收集器Serial来说，通常分配内存时会检测新生代内存是否足够，否则开始新生代垃圾收集，就像我们前面说的，一旦Survivor区无法装下需要复制的存活对象，JVM就会检查老年代是否有空间，如果老年代也没有空间，则会发起整堆回收(FULL GC)，如果还不行，则抛出异常。

记忆集与卡表

让我们回到标记阶段，现在让我们考虑一个新的问题：**跨代引用问题**。顾名思义，即老年代对象引用了新生代对象(通常不会说新生代引用老年代，因为GC几乎都是对新生代发起的)。考虑下图情形，JVM通常只对新生代进行垃圾回收，为了节省性能，JVM并不会扫描指向老年代的GC ROOTS，而只会对指向新生代的GC ROOTS进行可达性分析，如果是这样，那么下图中的 Obj 2 将会被错误的回收。

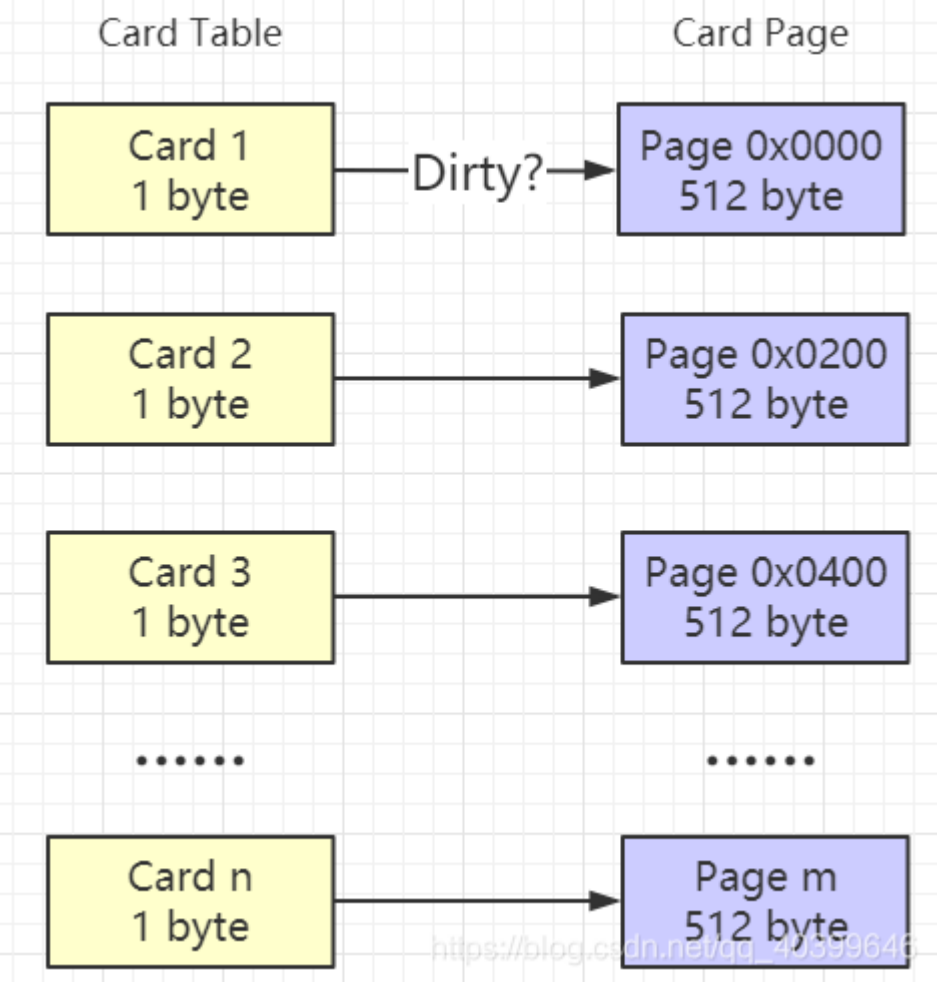


可是如果遍历所有老年代或遍历所有GC ROOTS进行新生代的回收，代价是非常昂贵的，为此，官方提出了记忆集的概念，**记忆集是一种用于记录从非收集区域指向收集区域的指针集合的数据结构。此处，记忆集将记录从老年代指向新生代的跨代指针。**

记忆集是一种抽象的数据结构，而卡表则是具体的实现。还有其他不同的实现，但卡表已经是目前的主流实现了。

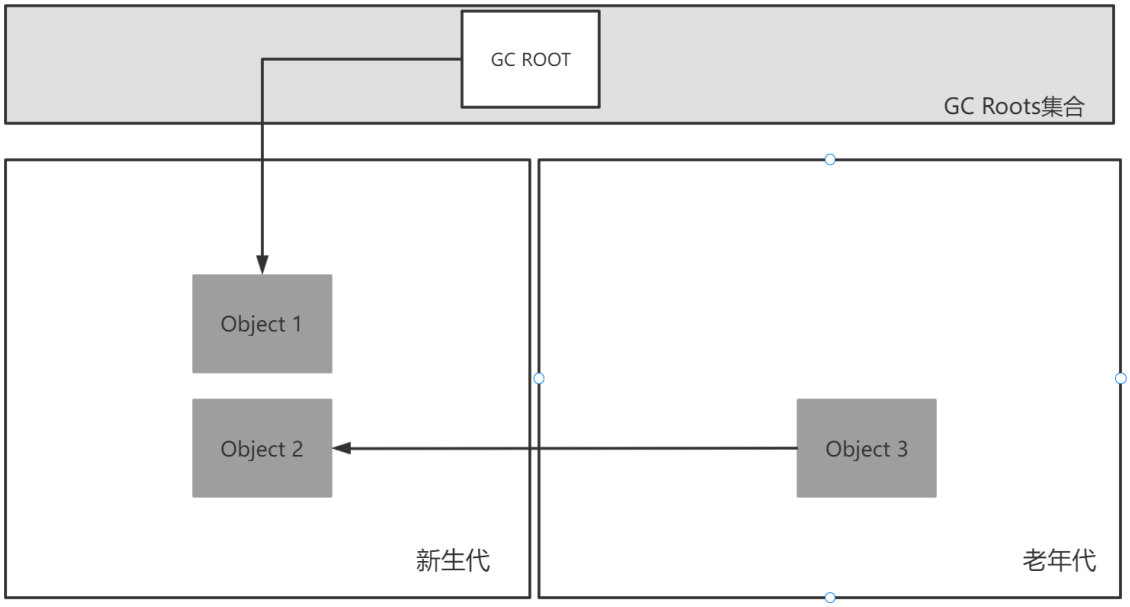
在hotspot虚拟机中，卡表是一个字节数组，数组的每一项对应着内存中的某一块连续地址的区域，如果该区域中有引用指向了待回收区域的对象，卡表数组对应的元素将被置为1，没有则置为0；例如如下代码，某块内存的地址向右移动9位(相当于除以512)定位到一个卡表元素，也就是说，内存中每512字节的连续区域会被定位到同一片卡表区域，如果卡表对应元素为1则代表该512个字节所在区域中有指向的指针。

```
CARD_TABLE [this address >> 9] = 0;
```



卡表是精确到一块内存区域的，只要这一块内存区域内有一个跨代引用则对应卡表都会被设置为1，没有精确到一个对象(一个卡表指向一个对象，对象内存在跨代引用则设置1)的原因是，对象通常是小而多的，这意味着需要更多的卡表项，增加了空间成本，而根据分代假说，跨代引用通常是很少的，因此稍微放松限制以节省内存是可以接受的。**现在，GC开始时，JVM还会检查卡表，如果卡表项为1，则遍历检查对应的内存区域，找出对应的跨代引用，并把它加入GC ROOTS之中。**

注意一个问题，即使像下图这样，并没有实际的引用指向 Obj 3，此时 Obj 3仍然会被加入 GC ROOTS，使得 Obj 2 不会被回收，但这是可以接受的，因为 Obj 2 很快就会进入老年代，跨代引用不存在了，Obj 2 和 Obj 3 会被一起回收。



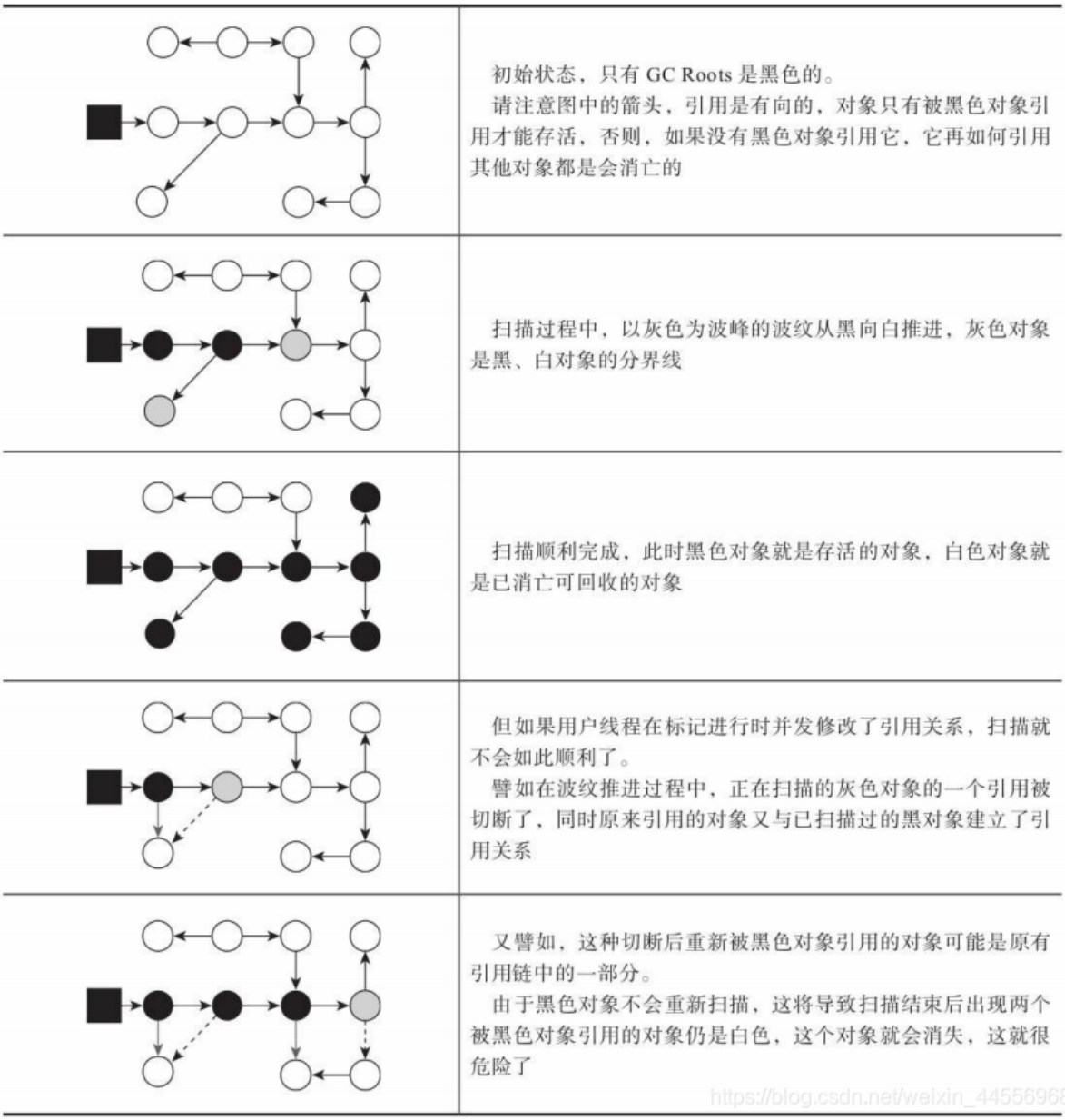
对老年代的回收呢？老年代肯定也存在跨代引用，但事实上解决这个问题是顺带解决的，由于对老年代发起回收时，往往都需要发起对新生代的回收(因此称为FULL GC)，对新生代的回收需要标记存活的对象，而此时顺带检查一下是否存在跨代引用即可，存在则加入老年代的GC ROOTS。

并发的可达性分析

我们想让GC过程与用户线程并发运行，来分析一些这可不可行。

- 根节点枚举是需要暂停的，其实也可以并发运行，但代价高的难以接受。
- 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**中才能够进行分析，但后来的科学家们发现标记阶段也是可以并发运行的，只是需要一定的代价进行修正，经过实践发现，这种代价是可以接受的。
- 一旦标记成功了，清除阶段是可以并发运行的，这不会造成任何问题。

我们主要分析并发的可达性分析，在 *深入理解JVM* 中有一张很形象的图：



如果是本该删除的对象最终没有删除，这种情况是可以接受的，等到下一次GC时回收即可；但对于本该存活的对象意外删除，这种情况是不可接受的。

来看最后两张图，黑节点增加了一条引用链，但由于灰节点删除了引用链，已经没办法访问该白节点了，现在，用户需要的对象却被我们意外的删除了，这会导致致命的错误，官方称这种情况为 “对象消失”。

Wilson于1994年在理论上证明了，当且仅当以下两个条件同时满足时，会产生“对象消失”的问 题，即原本应该是黑色的对象被误标为白色：

- **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；**
- **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，我们要解决并发扫描时的对象消失问题，只需破坏这两个条件的任意一个即可。由此分别 产生了两种解决方案：增量更新（Incremental Update）和原始快照（Snapshot At The Beginning， SATB）。

- 增量更新要破坏的是第一个条件，当黑色对象插入新的指向白色对象的引用关系时，就将这个新插入的引用记录下来，等并发扫描结束之后，再将这些记录过的

引用关系中的黑色对象为根，重新扫描一次。这可以简化理解为，黑色对象一旦新插入了指向白色对象的引用之后，它就变回灰色对象了。

- 原始快照要破坏的是第二个条件，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记录下来，在并发扫描结束之后，再将这些记录过的引用关系中的灰色对象为根，重新扫描一次，标记扫描到的对象为存活。这也可以简化理解为，无论引用关系删除与否，都会按照刚刚开始扫描那一刻的对象图快照进行搜索。

这是书上的原文，其实理解起来也很简单，**增量更新意味着重新扫描一遍用户更新的引用**，看看用户更新了哪条引用，顺着引用将用户指向的对象标记为存活，**也就是不改变原来的状态，单独进行一次额外的更新，即增量更新**；原始快照对被删除的引用重新遍历，可以抽象的理解为原来的链并没有删除，即一切还是原来的状态。

当进行二次扫描时(增量更新或原始快照)，不能再与用户并发的运行了，否则将会陷入死循环。因此这里并发不是绝对的，但二次扫描的时间是较为短暂的，是可以接受的。

现在还有一个问题是，如何额外记录下并发过程新的增量或旧的删除引用链，在发现这个问题的时候JVM已经比较成熟了，难以更改原来的架构了，因此需要考虑代理模式或装饰者模式，JVM采用了类似于动态代理技术，**即写屏障。**

要理解增加或删除引用都是发生对象与对象之间，栈上的局部引用不会对并发分析造成问题，而对象成员的复制原本的底层代码就像这样：

```
/**
* @param field 某对象的成员变量，如 D.fieldG
* @param new_value 新值，如 null
*/
void oop_field_store(oop* field, oop new_value) {
    *field = new_value; // 赋值操作
}
```

现在，通过JVM插入字节码，新的操作可能像下面一样(针对删除引用)：

```
void oop_field_store(oop* field, oop new_value) {
    pre_write_barrier(field); // 写屏障-写前操作
    *field = new_value; //(null)
}

void pre_write_barrier(oop* field) {
    oop old_value = *field; // 获取删除前的旧值
    remark_set.add(old_value); // 记录原来的引用对象
}
```

不必惊讶，JVM早就具备插入字节码的功能了，忘了提了，**卡表的变脏也是基于写屏障实现的。**

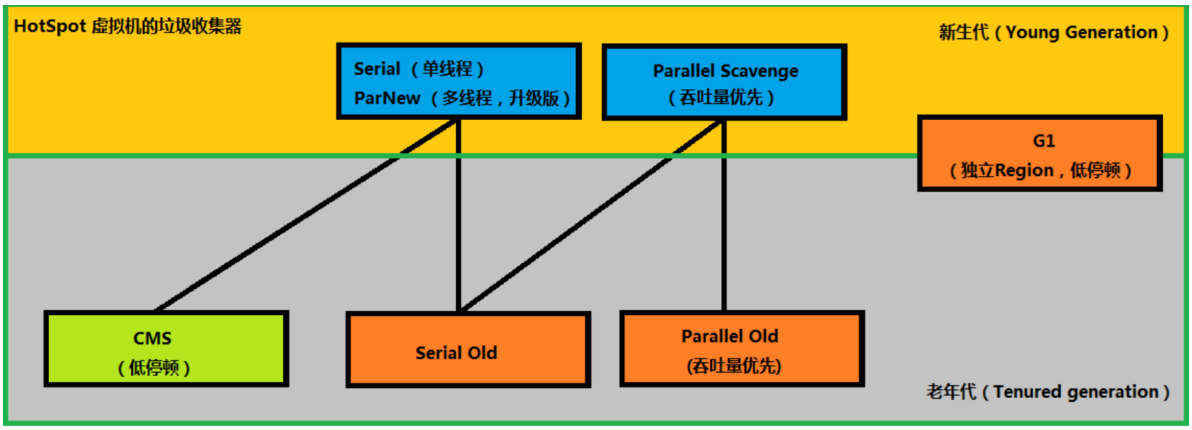
Eden 与 Survivo 大小对性能的影响

默认情况下，eden 与 survivo 区的比例是 8：1：1，我们来分析下调大/小 eden 区对垃圾收集的性能会有什么影响。

- 调大 Eden 区的比例
  - 当 Eden 区变得更大时，意味着 survivo 空间也变的更小了，我们可能需要更多的时间去收集新生代的垃圾，但同时，由于 survivo 变小，当 survivo 区占满时，我们不得不去占用老年代的空间，这就会给老年代带来更多的压力，如果老年代空间达到阈值，就会触发 FULL GC，因此，整个的结果是：**新生代回收的周期变得更长了，但回收时间也变长了。由于新生代可能过早移到老年代，老年代区压力过大，可能会提前回收，老年代的回收周期变短了，回收的频率和次数提高了，总体性能下降，总体停顿时间增加，但由于检测的比较及时，单次 STW 的时间会相应缩短。**
- 调小 Eden 区的比例的结果与调大 Eden 区的比例是相反的。

垃圾收集器

研究完理论之后，是时候来看看具体的实现了。



> 吞吐量是指用户应用程序线程用时占程序总用时的比例，即:
>
> $$吞吐量 = \frac{运行用户代码时间}{运行用户代码时间 + 垃圾收集时间}$$

Serial收集器

Serial收集器是HotSpot在Client模式下默认的新生代收集器，也是最基本、发展历史最悠久的收集器。该收集器属于串行收集器，这意味着当开始垃圾收集时必须要暂停所有线程。

Serial收集器对新生代采用标记复制法，GC时，所有用户都必须在安全点处停下来，当 Survivor 区不够时，就像我们之前说的，Serial收集器尝试使用老年代中的空

间，如果老年代空间不够，则发起一次FULL GC，FULL GC由 CMS 收集器或 Serial Old 收集器完成。

Serial收集器的优点是简单而高效，并且占用内存很小，但会带来较长时间的停顿，但对于客户端而言，这种停顿是可以接受的，直到现在，Serial收集器依旧是HotSpot在Client模式下默认的新生代收集器。

- 添加参数`-XX:+UseSerialGC`以显示指定Serial收集器。

ParNew收集器

ParNew收集器只是Serial收集器的一个升级版本，ParNew收集器多线程并发收集，但遗憾的是，这里的并发并不是指与用户线程共同运行，而是并发地清除垃圾。换句话说，ParNew收集器在GC时仍然要暂停用户线程，然后多线程地去清理垃圾，我们在之前讨论过并发性的可达性分析，那时候的并发主要是考虑与用户线程同时运行，要理清楚ParNew收集器并发收集的概念。

要注意并发收集的效率并不是一定就比串行的Serial收集器高，如果只是单核心模拟出来的并发，那还不如直接串行，毕竟线程切换也是重量级别的。但如果处理器核心较多，并发收集的效率就会明显高于串行收集。

Parallel Scavenge收集器

Parallel Scavenge收集器与 ParNew收集器十分相像，Parallel Scavenge收集器也是暂停式的，基于标记复制法多线程并发收集，但**评价Parallel Scavenge收集器是全自动的、吞吐量优先的收集器**。

我们已经提到过吞吐量的概念，Parallel Scavenge收集器允许用户指定一个比例，Parallel Scavenge收集器尽力的去保证程序的吞吐量接近该值，要想提高吞吐量，就得适当提升Eden区的空间，前面介绍过标记复制法的高效性，当我们提高了Eden区的空间时，下一次回收就能在较短的时间内回收更多的垃圾，从而提高吞吐量。

Parallel Scavenge收集器是在服务机下运行效率还不错的收集器。

- 添加参数`-XX:MaxGCPauseMillis=val`，val是指定的毫秒数，表示最大GC停顿时间，最小可为0，Parallel Scavenge只是会去尝试降低停顿时间，但Parallel Scavenge更关注吞吐量。
- 添加参数`-XX:GCTimeRatio=val`，val是 0 - 100 的值(百分之几)，表示吞吐量，当吞吐量与最大GC停顿时间冲突时，优先考虑吞吐量。
- 添加参数`-XX:+UseAdaptiveSizePolicy`表示让虚拟机自动调整 Eden 空间的大小、多大对象进入老年代、多就进入老年代等，虚拟机会尽力的保证一个吞吐量或停顿时间趋近于上述参数。

Serial Old收集器

SerialOld收集器是老年代收集器，这个收集器也是串行的，基于标记整理法回收老年代，回收时必须要“STW”，因此难以胜任服务器下的工作，SerialOld收集器一般只存在与客户端模式下。

Parallel Old收集器

Parallel Old收集器是服务器端可选的老年代收集器，该收集器的目的是为了提高FULL GC的吞吐量而生，为了实现这个目的，Parallel Old收集器采用并发的标记-整理法以清理老年代，但注意这里仍然要暂停用户线程，并发仅仅只是为了加快清理过程而非减少停顿时间，在多核处理下，这种方式确实能够提高FULL GC速度，从而提高吞吐量。

CMS收集器

CMS是一款老年代收集器，CMS第一次几乎真正实现了与用户并发运行，但是那个时候还没有解决如何实现真正意义上的并发标记整理法，因此**CMS采用的是标记清除法，这意味着即使CMS能够降低停顿时间，但依然会带来许多外部碎片。CMS的关注点就是低停顿，CMS期望能让用户线程尽可能的不停顿而完成GC。**

CMS运作分为4个步骤：

i. 初始标记(CMS-initial-mark)。初始标记生成GC ROOTS对象，并标记GC ROOTS能直接关联到的对象，正如我们前面说的，枚举GC ROOTS对象是需要暂停的，但鉴于已有技术优化，暂停时间会很短暂。
ii. 并发标记(CMS-concurrent-mark)。这一部分与我们在并发的可达性分析中一样，CMS采取的是增量更新的办法以消除并发问题。
iii. 重新标记(CMS-remark)。与我们在并发的可达性分析中一样，这一阶段CMS进行增量更新，重新遍历之前保存的引用以增量更新，这里需要“STW”。
iv. 并发清除(CMS-concurrent-sweep)。并发清除并没有什么问题，前提是我们已经正确的进行了标记。

在并发收集时，CMS必须要划分一些空间供正在运行的用户程序分配，CMS默认将这些空间标记为存活状态，这意味着这部分的空间产生的垃圾必须要等到下一次收集，这就是所谓的**“浮动垃圾”**，当这一部分预留空间被挤满时，CMS会停下所有工作，启动 Serial/Serial Old 收集器发起 FULL GC。当程序核心不够时，CMS就会运行缓慢，很可能会导致预留空间不够的情况，最后反而还是要调用串行收集器，弄巧成拙，CMS只推荐处理器核心 >= 4时使用。

G1(Garbage First)收集器

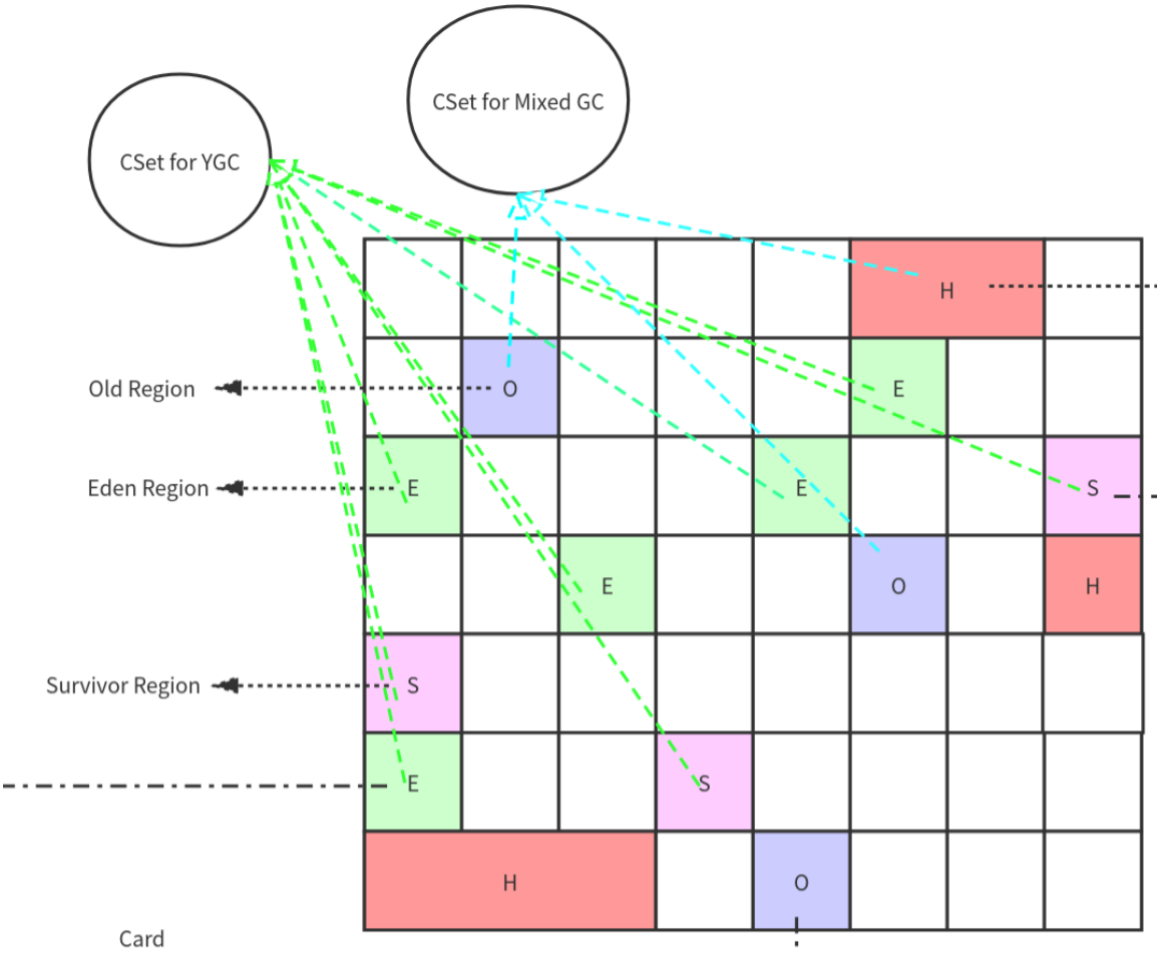
G1(Garbage First)垃圾收集器是当今垃圾回收技术最前沿的成果之一。早在JDK7就已加入JVM的收集器大家庭中，成为HotSpot重点发展的垃圾回收技术。同优秀的CMS垃圾回收器一样，**G1也是关注最小时延的垃圾回收器，也同样适合大尺寸堆内存的垃圾收集**，官方也推荐使用G1来代替选择CMS。G1最大的特点是引入分区的思

路，弱化了分代的概念，合理利用垃圾收集各个周期的资源，解决了其他收集器甚至CMS的众多缺陷。

G1垃圾收集器的设计原则是“首先收集尽可能多的垃圾(Garbage First)”，目标是为了尽量缩短处理超大堆（超过4GB）产生的停顿。

因此，G1并不会等内存耗尽（比如`Serial`串行收集器、`Parallel`并行收集器）或者快耗尽的时候才开始垃圾回收，而是在内部采用了**启发式算法**，在堆中找出具有高收集收益的分区（`Region`）进行收集。

G1淡化了分代的思想，它将整个堆分成相同大小的区，但G1没有放弃分代的思想，对于每个区，G1依然标记其年代，并基于不同年代采取不同的算法，值得注意的是，**G1将大对象单独分区**，即H区(Humongous区)，以避免对大对象标记整理的缓慢处理(直接GG)。这种分区不必在物理空间上连续，只需逻辑连续即可，这意味着G1可以动态的调整大小，当新生代空间不够时，G1会分配空闲的区，将他们组织成逻辑上的连续块，这使得G1的内存分配速率较高，并且这种做法消除了外部碎片，仅可能存在些许内部碎片。



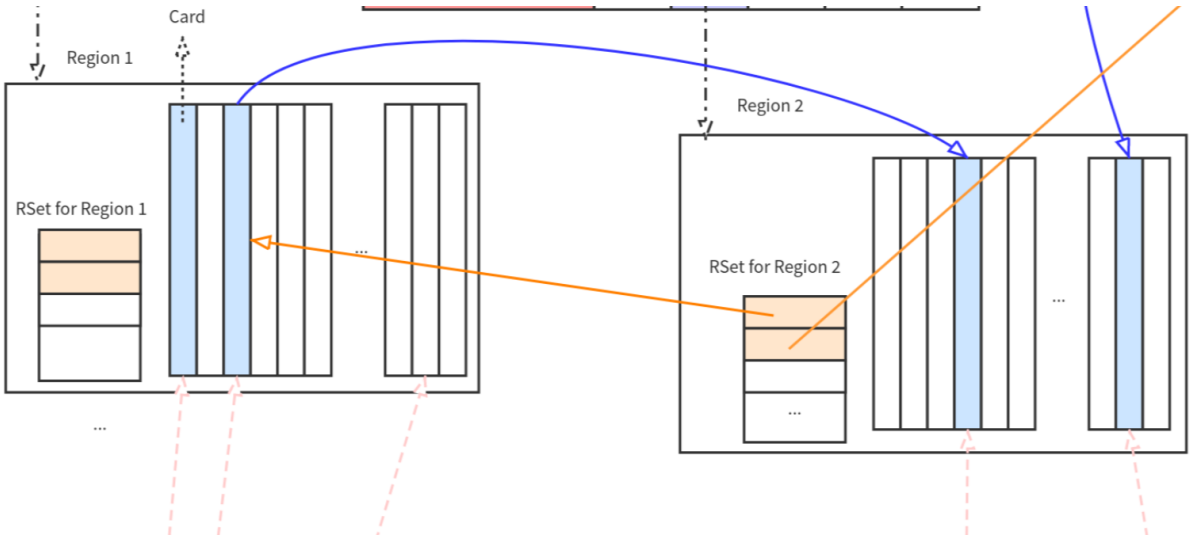
G1期望能够“**建立可预测的停顿时间模型**”，这意味着G1能够支持回收时间不超过N毫秒的决策，这与Parallel Scavenge收集器是不同的，Parallel Scavenge收集器只是去尝试，但G1期望能够尽力确保实现。

G1的收集是基于区的回收，G1可以额外关注每个区的状态，G1跟踪各个区垃圾堆积的价值大小，得出预测回收空间大小与回收所须时间的比例，在满足用户指定的停顿时间下，G1会优先回收能够回收最大空间的区，在之前，**停顿时间长主要是因为当**

老年代已经被占满了才会发起GC，而G1完全取消了这种模型，G1选择回收时不再是一味优先选择新生代了(但堆空间较大时仍然优先新生代)，而是根据预测的模型选择分区回收，这就使得几乎不会出现长时间的“STW”了。

如何确定区中可回收垃圾的价值是设计G1的难点，G1统计所有可能的选项，例如上一次回收前的卡表的脏页数量、上一次回收前区的分代以及使用空间大小等，然后结合上一次回收所耗费时间以及回收空间建立起一种联系，然后根据下一次回收前统计的数据，预测下一次的期望回收的时间，这种算法比我们想象的要复杂的多。

G1的一个分区被分为卡表以及512字节的块，G1的卡表更加复杂，G1的卡表更像是一个Map，key是该区被指向的区域，而Value指出是谁指向了自己，例如下图中Map可能存放 key = addr(Region 1) 和 对于Region1中的区域 指向 Region 2区域的索引号，即2。这种卡表占用更大的空间，但**更详细的信息利于每个分区的并发收集**。



G1的垃圾回收包括了以下几种回收机制：

- 年轻代收集。

  由于标记复制法的高效性，年轻代的收集并不需要与用户并发，G1采用与ParNew收集器相似的算法收集，年轻代的收集需要暂停用户线程。要注意的是，G1会动态调整 Eden 与 Survivor 区的大小以保证满足用户期望的停顿时间。

- 老年代收集/并发标记。

当堆空间达到一定阈值时（**默认45%**），便会周期性的触发并发标记。触发并发标记时会经历四个步骤：

i. 初始标记。这一阶段扫描GC ROOTS能够关联的对象，同时设置一块空间，供并发时用户分配新对象，这一部分空间默认标记存活。初始标记需要暂停，但G1并不立即开始工作，而是等到下一次年轻代收集暂停时一起运行。

ii. 并发标记。这与我们之前讲过的可达性分析类似。

iii. 重新标记。G1采用效率更高的原始快照进行二次标记，这一阶段结束后G1已经可以统计到堆中垃圾的“价值”了。

iv. 筛选。根据统计的数据预测回收的时间，根据用户期望停顿时间排序指定价值最大的回收计划。当计划生成后，G1 并不立即回收，而是等到新生代暂停时间进行“混合收集”。

- **混合收集**

并发周期结束后是混合垃圾回收周期，不仅进行年轻代垃圾收集，而且根据“计划”回收价值最大的老年代区。然后恢复到常规的年轻代垃圾收集，最终再次启动并发周期。

- 异常时回退为 Serial Old 发起 FULL GC。

同CMS一样。主要是清理过慢，导致用户无可用内存分配，此时会发起一次FULL GC。

G1几乎是一个“全能”的垃圾收集器，但G1非常依赖计算机资源，仅是每个分区保存的卡表就可能占用总内存的15%左右，通常，仅当内存 >= 16G时，内核数 >= 8时，使用G1才会有明显的效率提升。

> 对G1的了解主要来源于碎片化的信息统计，如有错误，望指出。