JVM GC篇 — 一般原理与垃圾收集算法

前言

GC一般原理

- 手动管理内存
- 对象已死?
  - 引用计数法
  - 可达性分析算法
  - 引用与对象生死
  - 并发的可达性分析

垃圾收集算法

- 分代收集理论
- 内存区域的划分
  - 新生代（Eden Space）
  - 存活区（Survivor spaces）
  - 老年代（Old Gen）
  - 永久代（Perm Gen）
  - 元数据区（MataSpace）
- 垃圾收集算法设计与应用
  - 标记-清除算法（Mark and Sweep）
  - 清理过程中的一个小问题
  - 标记-清除-整理算法（Mark-Sweep-Compact）
  - 复制算法（Copying）

总结

学习资料

前言

我们都知道 Java 程序员不用像 C++ 程序员一样手动申请和释放对象空间，这是因为 JVM 垃圾回收器的存在。为什么需要垃圾回收呢？因为空间是有限的，而我们运行程序完成各种各样的计算需要申请空间。但是空间不是无限的，如果我们的空间不够申请怎么办？这个时候就要释放内存，删除掉一些不用的无效的空间，来腾出空间来创建我们需要的对象。这就像我们的衣柜一样，我们买衣服回来放在衣柜里面，但是衣柜不是无限大的，所以当衣柜放不下的时候，我们就要清理丢掉小了的或者我们不穿的衣服，这样为新的衣服腾出空间。Java 自带GC会自动清理内存空间，而C++这种没有GC的语言，就需要使用手动清理空间了。全自动不用程序员操心的确很好，如果放仍不管也会引起其他的问题，那么怎么清理起来才是正确且高效。这就是一个有意思的问题，这一小节我们将从 GC 的一般原理和垃圾回收算法开始，逐步展开 Java GC 的世界。

GC一般原理

手动管理内存

有C++编程经验或者了解计算机原理的同学很容易就能理解，内存分配和内存释放两个概念，计算机程序在执行过程中，需要有地方存放输入参数、中间变量以及运算结果。通过前面的学习我们知道，这些运行时数据都放在堆栈内存中，栈中的数据的空间会随着栈的创建分配，随着栈空间销毁而释放。但是如果业务处理代码中需要使用到堆内存，这个时候就要注意了。因为空间是有限的，所以使用完的空间要即时释放这样才不会造成内存溢出的问题。因此C++程序员需要手动调用方法完成内存空间的释放。这种内存管理方式我们称为手动内存管理。

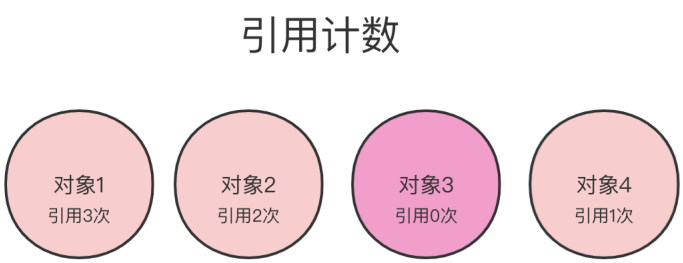
这种管理方式的优点就是简单且高效，因为用完的空间能被立刻释放，直接提高内存空间的使用率。但是缺点也是很明显的，就是如果一旦操作的人多了，容易出现操作不统一，造成内存资源抢占错误或内存错误释放的问题。并且内存空间的管理难度是随着操作方数量地增加而直线上升。在大型复杂

的项目中，多线程的内存操作是不避免的，因此手动内存管理的弊端被不断放大。 开发人员就想为什么不设计一个垃圾回收器自动的收集垃圾，这样既可以**减少程序员代码量，也可以尽可能的避免内存溢出**。因此GC顺势而来，其实GC的历史比 Java 还要久远，第一个带有动态分配和GC的语言并不是Java，1960年诞生的Lisp是第一门使用内存动态分配和垃圾收集的编程语言。

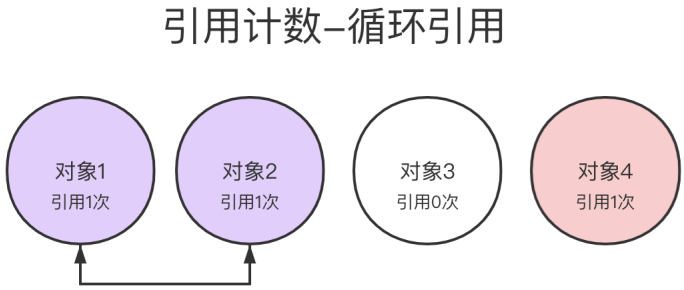
对象已死?

引用计数法

越是看起来简单的东西，计算机实现起来越是复杂。就比如这个例子，如果判断一个对象已经死亡了。开发小哥哥想到了一个好办法，这个对象我创建的不用了他就死亡了。但是这个方案放在GC上可行么？GC并不知道一个对象什么时候不会在被使用了，所以判断一个对象是否存活，这成为GC设计的第一个困难。遇到困难，正面面对。从对象的引用关系上下手我们很容易想到一个办法。**先创建一个引用计数器，如果这个这个对象有个一地方引用了我们把引用计数器加一，如果不再引用了引用计数器减一**。当某个对象的引用计数器为0的时候就意味着没有地方引用这个对象，即对象已经“死亡”了，可以进行垃圾回收了。

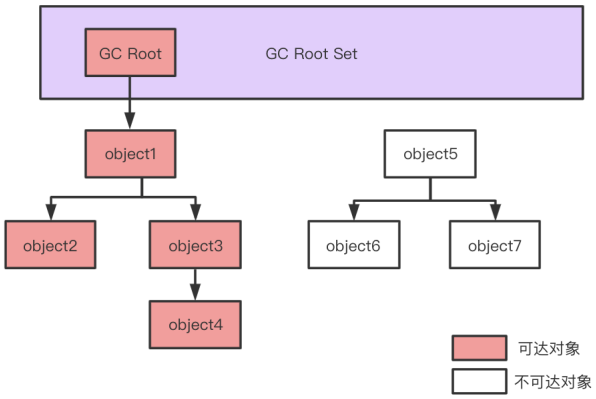


但是这个看似很完美的方案存在弊端嘛？没错就是`循环引用`的问题，上面这个图是一个对象引用，但是实际中我们创建的对象引用关系可比这个复杂多了。如果对象之间的相互引用，即便其他引用全部移除。引用计数器依旧不为0对象依旧无法回收。如下图其实对象1和对象2已经没有其他的引用了，并且这两个对象相互依赖可以直接进行垃圾回收，但是对象1和对象2的引用不为0，垃圾回收器判断他们还是“存活”状态，从而无法回收这两个对象。



可达性分析算法

引用计数法，虽然乍一看还不错但是有一个很严重的问题`循环引用`，因此基于引用计数的方式并不准确。那么有没有一个准确且性能也不错的方法来判断对象存活呢？办法不是灵光一线蹦出来的，是基于现实的情况推敲出来的。我们在引用计数法遇到的问题是循环引用。那我们是否可以从对象间互相引用的角度来思考是否能判断对象是否存活呢？答案是可以的。可达性分析，我们可以通过一些列`GC Roots`的**根对象作为起始点集合，从这些起始点开始，根据引用关系向下搜索，搜索过程所走过的路径称为“引用链”（Reference Chain），如果某些对象不可达，则证明次对象是不可能在被使用的**。通俗点来说就是通过从一个固定根节点然后遍历所有的关联对象的方式判断某一个对象是否存活。



其中GC Root 包含以下几种：

- 在虚拟机栈（栈帧中的本地方法变量表）中引用的对象。
- 在方法区中类静态属性引用的对象，譬如Java类中引用类型静态变量。
- 在方法去中常量引用的对象，譬如字符串常量池（String table）里的引用。
- 在本地方法栈中 JNI（即通常所说的Native方法）引用对象。
- Java 虚拟机内部的引用，如基本数据类型对应的 Class 对象，一些常驻的异常对象等，还有系统类加载器。
- 所有被同步锁（synchronzie关键字）持有对象。
- 反映Java虚拟机内部情况的 JMXBean，JVMTT中注册的回调、本地代码缓存等。

除了这些固定的 GC Roots 集合以外，**根据用户所选用的垃圾收集器以及当前回收的内存区域不同，还可以有其他对象“临时性”地加入**，共同构成完整 GC Roots 集合。

引用与对象生死

无论前面通过引用计数法判断对象引用数量，还是通过可达性分析算法判对象是否引用链可达来判断对象是否存活，这都和引用离不开关系，在`JDK1.2`之前的版本中，引用定义非常传统，**如果reference类型的数据中存储的数值代表的是另外一块内存的起始地址，就称该 refrence 数据代表某块内存、某个对象的引用。**这种定义并没有什么不对，但是现在看过于狭隘了。引用状态非黑即白，对于一个对象的描述只有两种状态即`被引用`和`未被引用`。如果是一种对象在内存充足的时候可以保留，但是在内存紧张的时候可以抛弃的对象，那么这种引用将很难描述，系统中很多缓存对象都符合这种场景。因此在`JDK1.2`之后，Java对引用的概念进行了扩充，将引用分为`强引用（Strongly Refrence）`、`软引用（Soft Refrence）`、`弱引用（Week Refrence）`、`虚引用（Phantom Refrence）`，这四种引用关系逐渐减弱。引用的定义通过这四种描述的补充，四种引用关系可以更好的描述引用，更好的协助垃圾回收器判断对象存活状态进行垃圾回收。其中引用与GC动作的描述如下：

- `强引用`是最传统的“引用”的定义，指在程序代码之中普遍存在的引用赋值，**这种引用关系下的对象无论在什么情况下都不会被垃圾回收器回收。**
- `软引用`是用来描述一些`还有用，但非必须`的对象。只被软引用关联着的对象，在系统发生**内存溢出之前，会把这些对象列入到回收范围之中进行第二次回收**。如果这次回收还没有足够的内存，才会抛出OOM异常，`JDK1.2版本之后`提供`SoftRefrence`类来实现软引用。
- `弱引用`也是用来描述那些非必须对象，但是他的强度比软引用要更弱一些。**被软引用关联的对象只能存活到下一次垃圾收集发生为止，不管当前内存是否足够都会被回收。**`JDK1.2`版本之后提供`WeekRefrence`类来实现弱引用。
- `虚引用`也称为“幽灵引用”和“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获取一个对象实例。为一个对象设置虚拟引用关联的唯一目的只是为了**能在这个对象被回收时收到一个系统通知。**在`JDK1.2`版本之后提供了`PhantomRefrence`类来实现虚引用。

多种多样引用关系的衍生很像我们平时业务开发过程中，因为某种状态的局限性从而对其进行扩展。从原本的最基本的`被引用`，拓展为`强引用`、`软引用`、`弱引用`和`虚引用`四种形式。这四种引用影响垃圾回收器的行为，帮助其高效判断并回收垃圾，从而提高内存空间的利用效率。

引用类型	对垃圾回收器的影响	回收时机
强引用（strongly refrence）	无论什么情况下都不会进行回收	不回收
软引用（soft refrence）	内存溢出之前，会把被该引用对象类如回收范围	内存溢出前回收
弱引用（week refrence）	被引用对象存活到下次垃圾回收为止	下一次垃圾回收时回收
虚引用（phantom refrence）	对被引用对象生命周期不产生影响，对象被回收时提供系统通知	无关联

前面我们看过了对象的引用关系和引用关系对垃圾回收器的影响。细心的朋友应该发现了上面的引用关系是可达对象。那不可达对象怎么处理呢？直接回收吗？不是的。不可达对象会先被判“死缓”，待回收对象会先判断是否需要执行`finalize()`方法，如果对象没有重写finalize()方法或者已经执行过finalize()方法，都会视为不需要执行。如果需要执行，对象会被放入一个队列中，依次执行finalize()方法。在执行 finalize() 方法，对象还可以抢救下自己，**只要重新和任何一个引用或者对象建立关联，那么这个对象就会被成功复活**。但是如果在这个阶段没有“成功自救”那就真的被回收了。当然没有finalize() 方法，那就会被直接被回收。finalize()看似是一个很棒的设计，但是这里也会产生很多的问题。可能会因为队列中对象不能即时处理造成对象没法回收从而造成OOM，甚至如果 finalize() 中有恶性循环，会导致整个内存回收子系统崩溃。**finalize()方法自 JDK9 开始被废弃。**

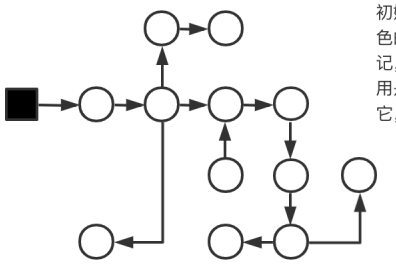
> 想要深入理解 finalize 机制的可以看这篇文章，虽然是全英文的但看起来压力并没有那么大
> debugging-to-understand-finalizer

并发的可达性分析

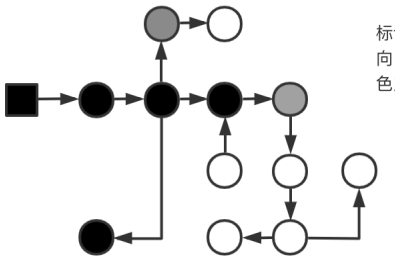
很多东西沾上了并发，事情的复杂度也就上去了但是仔细梳理下来也没那么复杂。可达性分析算法理论上要求全过程都是基于能保证一致性的快照中才能进行分析的，这样就意味着需要STW，冻结所有应用线程。在枚举根节点因为GC Roots 是极少数，STW是能接受的。而堆整个堆进行可达性分析时STW，对于并发的垃圾收集器来说是不能接受的。因此实现并发的可达性分析至关重要。可达性算法归根到底就是图的遍历，那么这个问题可以转化为，**如果在引用处于变化状态下，完成基本准确的遍历。**为什么是基本的呢？因为边是处于变化状态的，基本不可能标记出来的结果百分百准确。在这个场景中，我们要保证一个最基本的原则：可以错标，但是不能漏标记。**如果错标也就会产生一些浮动垃圾，漏标就会导致对象消失会影响应用线程。**那我们要怎么在引用变化的过程中进行动态标记呢？这里我们引入`三色标记（Tri-color Marking）`，在遍历对象图过程中遇到的对象，我们按照**“是否访问过”**这个条件标记成下面三种颜色。

- **白色：**表示尚未被垃圾收集器标访问过。在标记开始时，所有的对象都是白色的，**当标记结束时，如果对象还是白色的那表明对象不可达**。
- **黑色：**表示对象已经被垃圾收集器访问过，并且每个这个对象的所有引用都已经扫描过了，如果一个对象是黑色的那它就是**有效的可达对象。**黑色对象不可能直接指向某个白色对象。所以黑色对象不需要重新扫描。
- **灰色：**表示对象已经被垃圾收集器扫描过，但是**至少存在一个引用还没有被扫描过。**

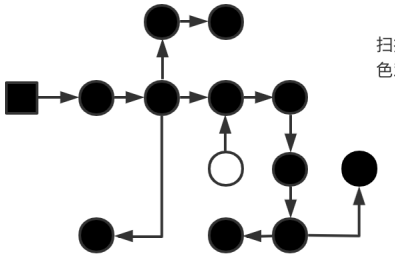
在引用关系不变化应用线程冻结的情况下，整个图遍历过程会由几个点开始，然后图会像水波纹一样各个节点从白变成黑。



初始标记状态，只有GC Root是黑色的，其他的对象都还没有被标记，所以是白色的。注意图中的引用是有向的。如果没有对象引用它，对象将被回收。

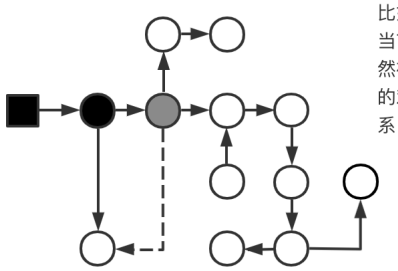


标记过程中，黑色会像水波纹一样向白色推进，灰色将成为白色和黑色之间的分割线。

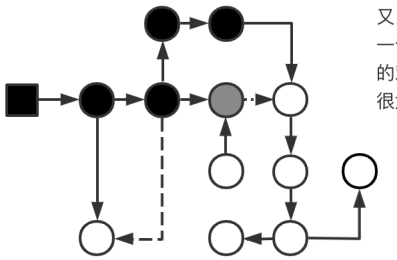


扫描完成黑色对象为存活对象，白色对象为凋亡对象。

上面这几张图是在应用线程停止的情况下，如果是在回收线程与应用线程并发，应用线程不断的修改引用关系的情况下，会发生什么情况呢？就像上面我们提到的，我们可以容忍少量的错标（**原本应该回收的对象，但是标记成了黑色**）但不可以容忍漏标（**原本不该被回收的对象，因为没有标记上黑色而被回收**）。来我们一起看看下面两个漏标的情况。



比如这种情况，在标记过程中，当前标记对象的某个引用关系突然被切断，同时一个已经标记过的对象和该对象又建立引用关系，这样就漏标了一个对象。



又比如这种情况，漏标的就不是一个对象了，而是一个对象链上的对象，这么多对象消失这就是很危险的了。

通过上面的图我们不难发现两种情况都有一个相同问题：**删除了灰色对象到某个白色对象之间的所有引用，并且同时一个标记过的黑色对象指向了这个白色或者这个白色对象所在链上的白色对象**，归纳下来就是两个相同的动作：

- 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者间接引用。

当这两个条件同时成立时候，就会产生消失对象。所以只要能破坏两个当中的一个条件问题就解决了。那解决方案有下面两个：

1. 增量更新（Increment Update）：破坏第一个条件，如果发现有从黑色对象指向白色对象的引用，那么就把这个这个引用记录下来，在最后标记STW的时候，以黑色对象为根重新标记一遍，**简单来说就是，黑色对象一旦新插入了白色对象的引用，那么就把这个黑色对象变成灰色，最后在遍历一次。CMS采用的就是这种方式。**
2. 原始快照（Snapshot At The Beginning， SATB）：破坏第二个条件，如果发现灰色对象删除指向白色对象的引用，就把这个引用记录下来，并且在最后标记的时候，以当时的灰色对象为根，按照原来的引用关系再标记一次。**简单来说，无论引用关系是否删除，就按照刚开始扫描那一刻的快照进行遍历，G1 采用的是这种方式。**

> 我之前一直在思考一个问题，新分配或新创建的对象怎么办？CMS中在并发标记时候，新晋升的对象刚进入老年代一定是白的，这怎么处理？后来我发现在并发预处理（Concurrent perclean）阶段，会标记一遍晋升对象。G1 的话则采用TAMS（Top at Mark Start）技术默认标记新分配的对象。这样就不会漏标新对象了。

垃圾收集算法

我们都知道我们对象实例是分配在堆空间中是一块连续的内存，我们会在在创建内存的时候都是在堆空间上分配一个块内存。由于实际情况的不同，每个实例对象的生命周期都不一样。有的对象可能刚分配使用完成就会被回收，有的可能会存在很长一段时间。这样垃圾回收器多次工作下来内存中可能存在多个内存间隙而导致新的对象无法继续分配。这明显不能充分利用有限的堆内存空间装尽可能多的对象。如果将一块完整的堆内存空间切分成多个逻辑空间，每个空间放生命周期不一样的对象，在每个逻辑空间使用不一样的收集策略是否就能尽可能的提高空间的使用率呢？当然是可以的，我们一起往下看。

分代收集理论

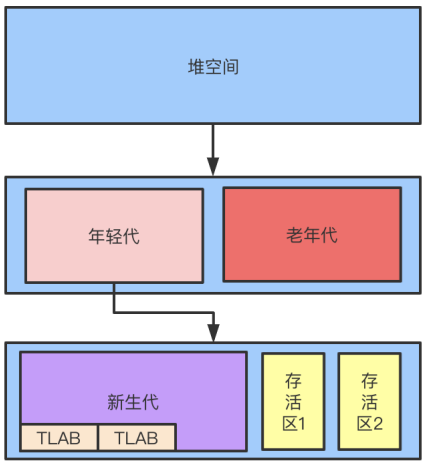
当前商业虚拟机的垃圾收集器，大多数都遵循了“分代收集”（Generational Collection） 的理论进行设计，分代收集虽然是理论，但是实质上是一套符合大多数程序实际运行情况的经验法则，它建立在两代假说的基础之上：

1. 弱分代假说（Week Generational Hyoothesis）：绝大多数对象那个都是朝生夕灭的。
2. 强分代假说（Strong Generational Hyoothesis）：熬过多次垃圾回收过程的对象就越难消亡。

这两个分代假说共同奠定了多款常用的垃圾回收器的一致设计原则：**收集器应该将 Java 划分出不同的区域，然后将回收对象依据其年龄（即熬过的垃圾回收过程的次数）分配到不同的区域之中存储。**

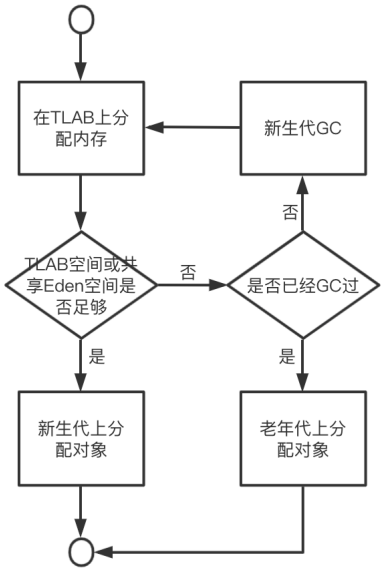
内存区域的划分

内存的划分也不是一蹴而就的也是慢慢发展来的，在前面的介绍Java 内存结构的小节中我们也简单的介绍了堆中的逻辑划分，这里我们展开聊聊这样划分背后的设计。根据前面的分代收集理论可以按照对象生命周期，将堆空间划分为生命周期短和生命周期长的区域即年轻代和老年代。其中为了更好的提升垃圾回收效率年轻代还划分为伊甸区 和 存活区。为了避免在分配内存空间时线程之间的竞争，伊甸区域为每个线程分配一小块内存空间，确保在线程并发创建对象时空间上的竞争这就是TLAB（Thread Local Allocation Buffer）。大致的内存区域划分图如下。



新生代（Eden Space）

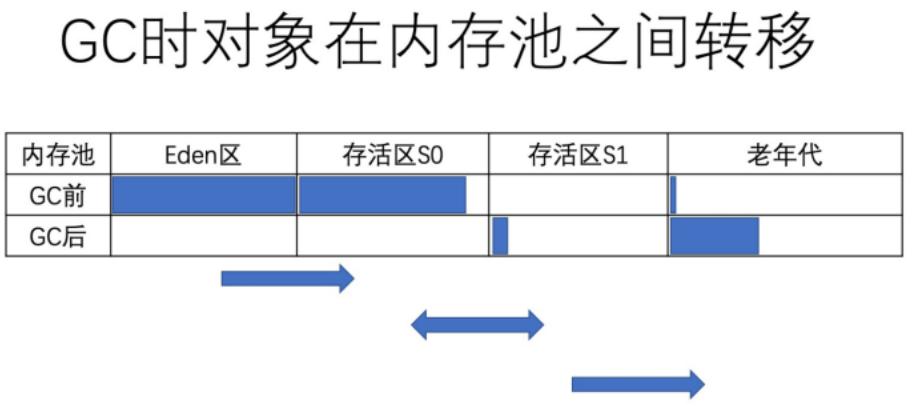
Eden Space，也叫做伊甸区，是内存中的一个区域，用来分配新的对象，通常会有多个线程同时创建多个对象，所以 Eden 区被划分为多个**线程本地分配缓冲区**（Thread Local Allocation Buffer，简称 TLAB）。通过这种缓冲区划分，大部分对象直接由JVM在对应线程的 TLAB 中分配，避免与其他线程同步操作。如果 TLAB 中没有足够的内存空间，就会在共享 Eden 区（Shared Eden Space）之中进行分配。如果共享的 Eden 区，也没有足够的空间，就会触发一次年轻代的 GC 来释放内存空间，如果 GC 之后 Eden 区依旧没有足够的内存空间，则对象就会被分配到老年代空间（Old Generation）。



当 Eden 区进行垃圾回收的时候，GC 将从 GC Roots 开始把所有的关联的对象都过一遍，并标记为存活对象。**标记完成后会将所有存活的对象都会被复制到存活区（Survivor spaces）**，这个时候就可以认为Eden区域是空的，就可以重新进行对象的分配，这个算法叫做标记复制算法（Mark and Copy）。

存活区（Survivor spaces）

Eden区旁边两个就是存活区（Survivor space），成为 `from空间` 和 `to空间`。需要着重强调的是任意一个时刻总有一个存活区是`空的（Empty）都是to空间`。每次的年轻代的 GC **都会把from区中的存活对象和Eden区中的存活对象复制到**`to区`中，from和to角色切换from变成to，to变成from。



存活的对象会在存活区中来回复制。**复制一次对象存活年龄+1**，按照强分代假设，存活超过一定时间的对象很可能会存活更长时间。这类对象当存活年龄超过`当年龄超过提升阈值(tenuring threshold)`，就会被`提升（Promotion）`至老年代区域。当然这个阈值参数是可以调整的，可以通过参数`-XX:+MaxTenuringThreshold`来指定上限。如果设置`-XX:+MaxTenuringThreshold=0`，对象不会在存活区之间复制会直接提升到老年代。JVM 中这个阈值的默认值是`15个GC周期`，如果存活区空间不够存放对象，`提升（Promotion）`也可能更早地执行。其中**存活区和Eden区的默认比例是 1：1：8 。**

老年代（Old Gen）

老年代的GC实现要复杂得多。**老年代的内存空间通常会更大，里面的产生垃圾对象的概率也更小，老年代GC发生的频率比年轻代小很多**。同时，因为预期老年代的对象大部分都是存活的，所以不再使用标记和复制（Mark and Copy）算法。而是采用移动对象的方式来实现最小内存碎片。老年代空间的清理算法通常是建立在不同的基础上的。原则上执行以下这些步骤：

- 通过标志位（marked bit），标记所有通过 GC Roots 可达对象；
- 删除不可达对象；
- 整理老年代空间中的内容，方法是所有的存活对象复制，从老年代空间开始的地方依次存放。

通过上面的描述可知，**老年代GC必须明确地进行整理，以避免内存碎片过多**。这也是`标记整理算法`。

永久代（Perm Gen）

在 Java8 之前有一个很特殊的空间，称为“永久代”（Permanent Generation）。这里存储数据（metadata）的地方，比如 class 信息等。此外，这个区域中也保存有其他的数据和信息。包括内部化的字符串（internalized strings）等等。实际上这块内存区域给开发这造成很多的麻烦，因为**很难去计算这块区域到底需要占用多少的空间**，预测失败的结果就是产生`java.lang.OutOfMemoryError: Permgen space`这种形式的错误。除非`OutOfMemoryError`确实是内存泄漏导致的，否则只能增加 permgen 的大小。

```Bash
-XX:MaxPermSize=256m //permGen 大小设置为256m，容易发生oom
```

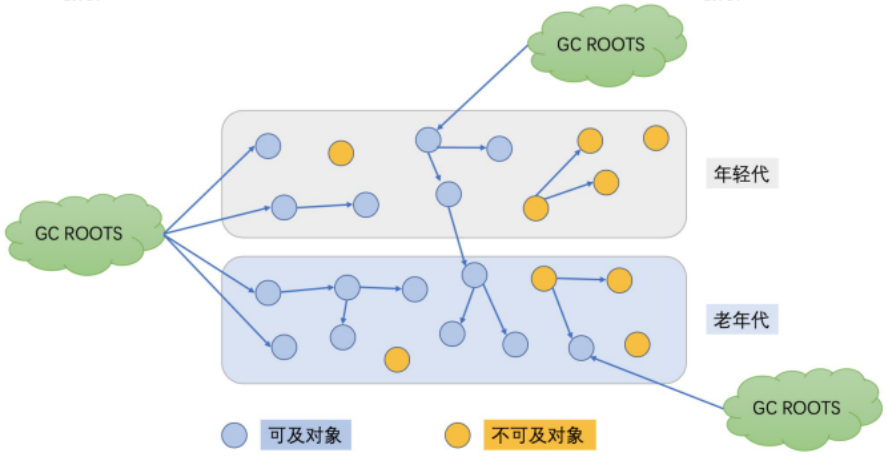
元数据区（MataSpace）

既然估算PermGen需要的空间那么复杂，Java8中索性直接删除了永久代（Permanent Generation）改用`MetaSpace`他俩本质上还是相同的。从此以后，Java 中很多杂七杂八的东西都放在普通的堆内存中。当然，像类定义（class definitions）之类的信息还是会被加载到 MetaSpace 中。**元数据区域位于本地内存（native memory）**，不再影响到普通的Java对象。`默认情况下，MetaSpace 的大小只受限于 Java 进程可用的本地内存。`这样的话就避免了 PermGen 因为预测不准确而OOM的尴尬了。但是自由也不是没有限制的，如果 MetaSpace 无限扩张失控，则可能会导致严重的程序性能问题，或者导致本地内存分配失败。为了避免这种事的发生我们还是要限制 MetaSpace 的大小。我们可以通过下面的方式限制其大小。

```Bash
-XX:MaxMetaspaceSize=512m // 设置metaSpace空间大小为512m
```

垃圾收集算法设计与应用

前面讲了很多的东西都是铺垫，但是空中楼阁不可能腾空而起，没有绿叶又哪来鲜花。我们来一起鸟瞰全局体会垃圾回收算法在设计的精妙之处。我们通过前面的梳理，我们对堆空间进行了逻辑上的划分，分成了`年轻代`和`老年代`，而年轻代存放的都是朝生夕死的对象。老年代则存放一些不容易被清理掉生命周期长的对象。年轻代又有两个幸存区配合垃圾回收，我们已经知道了我们内存布局了。接下来我们试着进行垃圾回收。首先我们要标记出已经“死亡”的对象，通过可达性算法分析我们很清楚标记出来的就是不可达的对象。年轻代和老年代根据我们清理目标空间的需要进行标记。



这个标记的过程中，即需要暂停所有应用线程以遍历所有对象的引用关系，因为我们无法追踪不断发生变化的引用关系。等确定这些引用关系后，应用线程又能继续执行。这个暂停的过程叫做`Stop The World pause（全线程暂停）`，简称为`STW`。线程也不可能在执行的过程中突然停下来，而是需要运行可以安全停下来的点，这些可以停下来的地方叫做`安全点（safe point）`，然后标记完成后 JVM 就可以安心处理垃圾了。

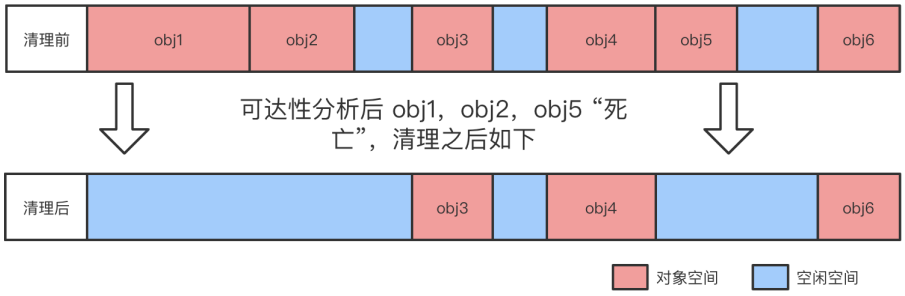
这里有一个有意思的问题：**如果一个新生代的对象有且只被一个老年代的对象引用，我们在年轻代进行GC时，我们该怎么判断该对象是存活的?**

如果我们仅仅扫描年轻代，这个对象一定是一个不可达对象，因为它被有且仅被一个老年代对象引用。难道我们进行一次年轻代GC连老年代也要扫一遍？如果扫，时间消耗太大，不扫，一定会出现这种误判的情况。那该怎么办？我们可以通过`找脏卡`的方式解决这个问题。什么是脏卡？这是 HotSpot 中一项叫卡表（card table）的技术中的一个名词，卡表是HotSpot 对记忆集（Remember Set）的实现。该技术**将整个堆划分为一个个大小为 512 字节的卡**，并且维护一个卡表，用来存储每张卡的一个标识位。这个标识位代表对应的卡**是否可能存有指向新生代对象的引用**。如果可能存在，那么我们就认为这张卡是**脏的**。在进行年轻代GC的时候，我们便可以**不用扫描整个老年代，而是在卡表中寻找脏卡**，并将**脏卡中的对象加入到年轻代GC的 GC Roots 里**。当完成所有**脏卡的扫描之后**，Java 虚拟机便会将所有**脏卡的标识位清零**。由于年轻代GC伴随着存活对象的复制，而复制需要更新指向该对象的引用。因此，**在更新引用的同时，我们又会设置引用所在的卡的标识位**。这个时候，我们可以确保脏卡中必定包含指向新生代对象的引用。

> 脏卡是包含指向新生代引用的卡，年轻代GC，不扫描老年代而是寻找脏卡并加入到年轻代GC Roots中，扫描完脏卡后表示位即脏卡去脏，复制算法后存活对象地址变化，重新老年代中引用变化是同时设置脏卡标志。这个过程随着GC周期不断循环下去。

标记-清除算法（Mark and Sweep）

标记完成后我们进行简单的清理可以得到类似下面的图。



可以看到上面图中，在一段连续的内存空间中，我们依次有`obj1到obj6`这6个对象。这6个对象之间不一定是紧紧靠在一起的，而是部分对象之间是有“间隙”的。在可达性分析算法标记过后，我们发现obj1，obj2, obj5为对象不可达即“死亡对象”。然后我们对这死亡的对象进行清理。清理完成后就得到了下面这个图（这里不考虑 finalize() 方法对垃圾回收过程产生的影响）。这个其实就是垃圾回收算法中的`标记-清除算法（Mark and Sweep）`，这个过程很简单简单分成下面两步：

- **通过可达性算法标记出不可达对象即“死亡对象”。**
- **“清理”不可达对象。**

这里有一个细节，这里的“清理”并不是真正的清理，而是**将死亡对象的内存空间地址记录到空闲表（free-list）上，然后直接使用这块空间进行空间分配。**

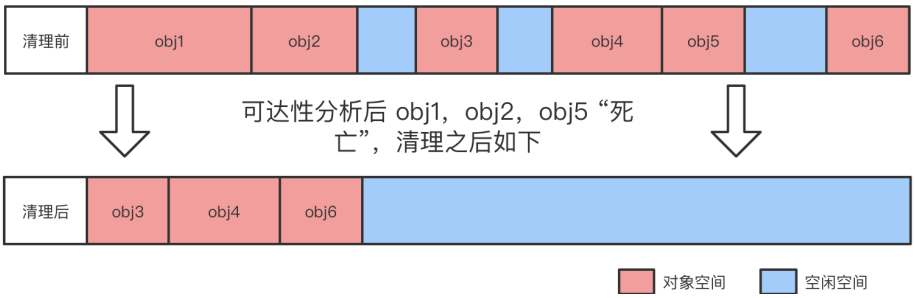
清理过程中的一个小问题

不知道你有没有发现一个问题，这里的空间内存虽然是记录在空闲表`free-list`上，但是新来一个对象可能不一定能分配的了。因为这个空间不是连续的，来一个对象可能比空间表 free-list 上所有的空闲空间都要大，但是所有的空闲空间加在一起又完全足够让这个对象进行分配，这个问题是不是就很难受？别急还有更难受的，别忘了我们可以通过GC也就是上面这个过程释放空间，GC完成后很顺利，刚好有一个对象被释放，一段连续的内存刚好够放下这个对象。我们放松一口气。这时又来了一个对象，这个对象和上一个对象遇到了同样的问题，这是我们该怎么办？GC？这时刚GC完成，应该是没有空间可以释放的，报OOM让程序员小哥哥解决？明明我们还有空间啊，只不过都是不连续的碎片空间。我们在这个场景中遇到的小问题就是**因为标记清除算法在清除完毕后不对空间进行整理，导致GC之后产生很**

多的不连续的碎片空间，这些碎片空间无法进行空间分配从而导致OOM的发生。我们该怎么解决这个办法呢?

标记-清除-整理算法（Mark-Sweep-Compact）

面对上面的问题其实最简单的办法就是在清理完成后我们整理下内存空间，把还存活的对象再次排好。有了整理部分的加入后上面的过程变成下面这样：



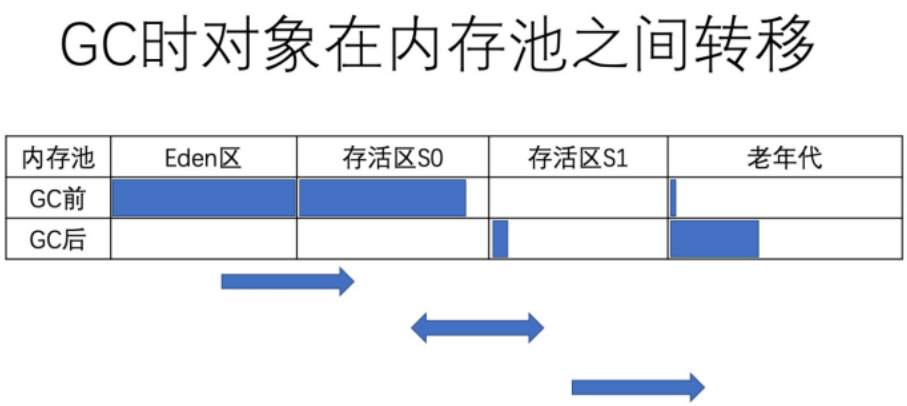
这个算法就是垃圾回收算法中的`标记-清除-整理算法（Mark-Sweep-Compact）`也称作`Compact压缩算法`。压缩算法分为下面三步，其中前面两步和标记-清除算法一致。

- **通过可达性算法标记出不可达对象即“死亡对象”。**
- **“清理”不可达对象。**
- **整理压缩对象空间。**

这个方案很完美，因为这个相较于标记-清理算法多了一步，这一步虽然压缩了对象，空闲空间也变成**连续的空间提升了空间的利用率**。但是多这一步的操作也为**增加了每次GC复杂度**增加每次的GC耗时。而老年代对象基本上都是存活对象且空间更大，发生GC的概率更低，因此**压缩算法更多的用在了老年代。**

复制算法（Copying）

上面的压缩算法的弊端是整理空间增加了GC的时间复杂度，有没有优化的方案呢？有的，在算法中我们经常会使用空间换时间的思想增加程序的空间复杂度来降低时间复杂度。这里我们可以直接**将所有的存活对象复制另外一个内存空间中，原本的内存空间直接清空。**复制过去的对象内存空间是连续的，我们还可以在新空出来的空间分配对象，这个复制过去的目标空间就是我们前面提到的`存活区（Survivor space）`，这个GC算法也就是我们前面提到的`标记复制算法`（Mark and Copy）



在实际使用中配合新生代Eden区和来回复制的两个Survivor存活区，实现高效的复制算法。因此复制算法分为以下三步：

- **通过可达性算法标记出不可达对象即“死亡对象”。**
- **复制存活对象到Survivor的to区。**
- **清空Eden区和Survivor的from区，并且from和to区对换。（逻辑上的清空和对换）**

复制算法的时间复杂度要明显优于压缩算法，因此复制算法更加适合“节奏更快”的年轻代。**复制算法是运用在年轻代的GC算法。**

> 部分收集（`Partial GC`）指目标不是完整收集整个 Java 堆的垃圾收集，其中又可划分：
> - 新生代收集（`Minor GC`/`Young GC`）：收集目标为新生代的垃圾收集。
> - 老年代收集（`Major GC`/`Old GC`）：目标为老年代的垃圾收集。

> - 混合收集（`Mix GC`）：目标为整个新生代以及部分老年代的垃圾收集，目前只有G1有这种行为。
>
> 整个堆收集（`Full GC`）：收集整个Java堆和方法区的垃圾收集。

总结

这一小节我们从最原始的手动管理内存开始，简单分析其利弊之后，我们逐步开始走向自动的垃圾回收机制，第一个垃圾回收机器并不是诞生在Java语言上，在进行垃圾回收之前我们要做的第一件事不是清理回收，而是判断一个对象是“活着”还是已经“死亡”。这里我们简单介绍了引用计数法。在用计数法解决不了循环引用之后，我们介绍了可达性分析算法，通过一些列GC Roots 判断某个对象是否可达从而判断这个对象是否存活。为了更好的进行垃圾回收，引用的两种状态引用和被引用显然不能很好的描述引用的类型状态，因此我们引入了强引用、软引用、弱引用和虚引用，顺便提了这几种引用对垃圾回收器行为的影响。接下来我们进入垃圾回收算法，在正式介绍算法之前我们聊了聊分代收集理论。为了更高效的进行垃圾回收，JVM依据对象的“存活时间”的长短，分为年轻代、老年代和永生代以及后续替代永生代的元数据区。并为了更好的配合垃圾回收器的工作又将年轻代切分成新生代和存活代。最后我们结合图文简单介绍标记清除、标记整理（压缩算法）和复制算法。简单明了的说明了各种算法的特点和工作的内存区域。我们还提到了一些细节不要忘了哦，我们即将被淘汰的 finalize() 方法对垃圾回收的一丢丢影响和即将在垃圾回收器中大放异彩的卡表设计。

今天这小节内容很基础但是梳理下来真的很多也画了不少图，按照自己的逻辑走一遍下来真的透彻了很多。九层之台，起于累土。加油～。

学习资料

- 极客时间专栏《深入理解 Java 虚拟机—垃圾回收（上）》
- 极客时间专栏《深入理解 Java 虚拟机—垃圾回收（下）》
- 常见的的GC算法（GC的背景与原理）
- 深入理解 Java 虚拟机（第三版）