20 | Scavenge：基于copy的垃圾回收算法

2021-12-13 海纳

《编程高手必学的内存知识》 课程介绍 >

讲述：海纳

时长 20:00 大小 18.32M

你好，我是海纳。

上一节课中，我们讲到 GC 算法大致可以分为两大类：引用计数法和基于可达性分析的算法。在基于可达性分析的 GC 算法中，最基础、最重要的一类算法是基于 copy 的 GC 算法（后面简称 copy 算法）。

Copy 算法是最简单实用的一种模型，也是我们学习 GC 算法的基础。而且，它被广泛地使用在各类语言虚拟机中，例如 JVM 中的 Scavenge 算法就是以它为基础的改良版本。所以，掌握 copy 算法对我们后面的学习是至关重要的。

这一节课，我们就从 copy 算法的基本原理开始讲起，再逐步拓展到 GC 算法的具体实现。这些知识将帮助你对 JVM 中 Scavenge 的实现有深入的理解，并且让你正确地掌握

Scavenge GC 算法的参数调优。

最简单的 copy 算法

基于 copy 的 GC 算法最早是在 1963 年，由 Marvin Minsky 提出来的。这个算法的基本思想是把某个空间里的活跃对象复制到其他空间，把原来的空间全部清空，这就相当于是把活跃的对象从一个空间搬到新的空间。因为这种复制具有方向性，所以我们把原空间称为 From 空间，把新的目标空间称为 To 空间。

分配新的对象都是在 From 空间中，所以 From 空间也被称为**分配空间（Allocation Space）**，而 To 空间则相应地被称为**幸存者空间（Survivor Sapce）**。在 JVM 代码中，这两套命名方式都会出现，所以搞清楚这点比较有好处。我们这节课为了强调拷贝进行的方向，选择使用 From 空间和 To 空间来分别指代两个空间，而尽量不使用分配空间和幸存者空间的说法。

最基础的 copy 算法，就是把程序运行的堆分成大小相同的两半，一半称为 From 空间，一半称为 To 空间。当创建新的对象时，都是在 From 空间里进行内存的分配。等 From 空间满了以后，垃圾回收器就会把活跃对象复制到 To 空间，把原来的 From 空间全部清空。然后再把这两个空间交换，也就是说 To 空间变成下一轮的 From 空间，现在的 From 空间变成 To 空间。具体过程如图所示：

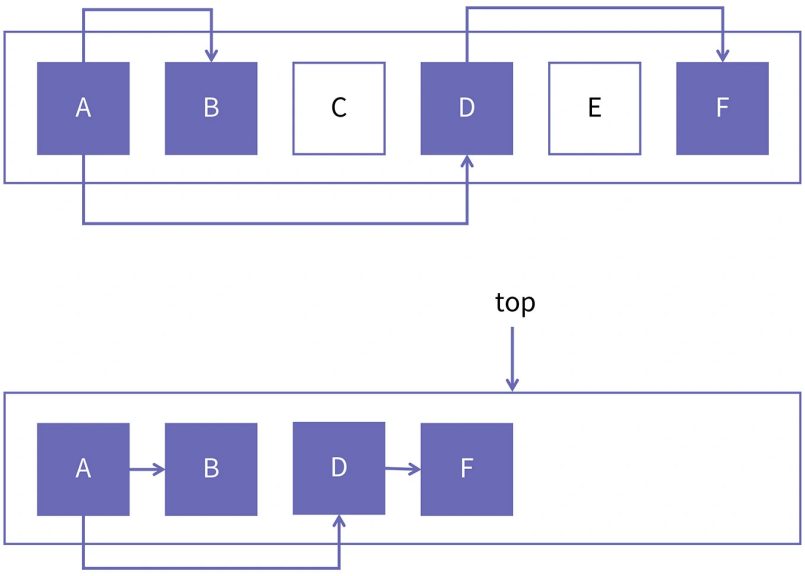

图1

可以看到，上图中的 from 空间已经满了，这时候，如果想再创建一个新的对象是无法满足的。此时就会执行 GC 算法将活跃对象都拷贝到新的空间中去。

假设 A 对象作为根对象是存活的，而 A 引用了 B 和 D，所以 B 和 D 是活跃的；D 又引用了 F，所以 F 也是活跃的。这时候，要是已经没有任何地方引用 C 对象和 E 对象，那么 C 和 E 就是垃圾了。当 GC 算法开始执行以后，就会把 A、B、D、F 都拷贝到 To 空间中去。拷贝完成后，From 空间就清空了，并且 From 空间与 To 空间相互交换。

此时，top 指针指向了新的 From 空间，并且是可用内存的开始处。如果需要再分配新的对象的话，就会从 top 指针开始进行内存分配。

我们知道，GC 算法包括**对象分配和回收**两个方面。下面，我们分别从这两个方面对 copy 算法加以考察。先来看 copy 算法的内存分配是怎么做的。

对象分配

从上面的介绍里我们知道，在 From 空间里，所有的对象都是从头向后紧密排列的，也就是说对象与对象之间是没有空隙的。而所有的可用内存全部在 From 空间的尾部，也就是上图中 top 指针所指向的位置之后。

那么，当我们需要在堆里创建一个新的对象时，就非常容易了，只需要将 top 指针向后移动即可。top 指针始终指向最后分配的对象的末尾。每当新分配一个新对象时，只需要移动一次指针即可，这种分配效率非常高。

如果按这种方式进行新对象的创建，那么对象与对象之间可以保证没有任何空隙，因为后一个对象是顶着前一个对象分配的，所以，这种方式也叫做**碰撞指针（Bump-pointer）**。

了解了 copy 算法的内存分配过程后，我们再来看回收的过程。

Java 对象的内存布局

在进行垃圾回收之前，我们首先要识别出哪些对象是垃圾对象。上一节课我们讲过，如果一个对象永远不会再被使用到，那么我们就可以认为这个对象就是垃圾。识别一个对象是

否是垃圾的主要方法有两种：引用计数和基于可达性的方法。上一节课我们讲了引用计数，这节课，我们主要来看基于可达性分析，或者称为追踪（Tracing）的方法。

要想识别一个对象是不是垃圾，**Tracing 首先需要找到“根”引用集合**。所谓根引用指的是不在堆中，但指向堆中的引用。根引用包括了**栈上的引用、全局变量、JVM 中的 Handler、synchronized 对象等**。它的基本思想是把对象之间的引用关系构成一张图，这样我们就可以从根出发，开始对图进行遍历。能够遍历到的对象，是存在被引用的情况的对象，就是活跃对象；不能被访问到的，就是垃圾对象。

那怎么把对象之间的引用关系抽象成图呢？这就涉及到了 Java 对象的内存布局，我们先来看一下 Java 对象在内存中是什么样子的。

在 JVM 中，一个对象由对象头和对象体构成。其中，**对象头（Mark Word）在不同运行条件下会有不同的含义，例如对象锁标识、对象年龄、偏向锁指向的线程、对象是否可以被回收等等。而对象体则包含了这个对象的字段（field），包括值字段和引用字段**。

```
class A {
    int a;
    String b;
    Object c;
    double d;
    List e;
}
```

图2

每一个 Java 对象都有一个字段记录该对象的类型。我们把描述 Java 类型的结构称为 Klass。Klass 中记录了该类型的每一个字段是值类型，还是对象类型。因此，我们可以根据对象所关联的 Klass 来快速知道，对象体的哪个位置存放的是值字段还是引用字段。

如果是引用字段，并且该引用不是 NULL，那么就说明当前对象引用了其他对象。这样从根引用出发，就可以构建出一张图了。进一步地，我们通过图的遍历算法，就可以找到所有被引用的活对象。很显然，没有遍历到的对象就是垃圾。

通常来说，对图进行遍历有两种算法，分别是**深度优先遍历（Depth First Search，DFS）和广度优先遍历（Breadth First Search，BFS）**。接下来，我们一起看看这两种算法是如何完成图的遍历的。

深度优先搜索算法的实现

复制 GC 算法，最核心的就是如何实现复制。根据上面的描述，我们自己就可以很容易地写出一个基于深度优先搜索的算法，它的伪代码如下所示：

复制代码

```
1  void copy_gc() {
2      for (obj in roots) {
3          *obj = copy(obj);
4      }
5  }
6  obj * copy(obj) {
7      new_obj = to_space.allocate(obj.size);
8      copy_data(new_obj, obj, size);
9      for (child in obj) {
10         *child = copy(child);
11     }
12     return new_obj;
13 }
```

可以看到，算法的开始是从 roots 的遍历开始的，然后对每一个 roots 中的对象都执行 copy 方法（第 2~ 4 行）。copy 方法会在 To 空间中申请一块新的地址（第 7 行），然后将对象拷贝到 To 空间（第 8 行），再对这个对象所引用到的对象进行递归的拷贝（第 9~11 行），最后返回新空间的地址（第 12 行）。

在第 4 节课我们讲解栈的递归特性时，曾经对深度优先搜索的递归写法做过深入分析。拿上面的代码和第 4 课中的代码进行比较，我们会发现，上面的代码中缺少了对重复访问对象的判断。

考虑到有两个对象 A 和 B，它们都引用了对象 C，而且它们都是活跃对象，现在我们对这个图进行深度优先遍历。

图3

在遍历过程中，A 先拷到 to space，然后 C 又拷过去，这时候，空间里的引用是这种状态：

图4

A 和 C 都拷到新的空间里了，原来的引用关系还是正确的。但我们的算法在拷贝 B 对象的时候，先完成 B 的拷贝，然后你就会发现，此时我们还会把 C 再拷贝一次。这样，在 To 空间里就会有两个 C 对象了，这显然是错的。我们必须要想办法解决这个问题。

通常来说，在一般的深度优先搜索算法中，我们只需要为每个结点增加一个标志位 visited，以表示这个结点是否被访问过。但这只能解决重复访问的问题，还有一件事情我们没有做到：新空间中 B 对象对 C 对象的引用没有修改。这是因为我们在对 B 进行拷贝的时候，并不知道它所引用的对象在新空间中的地址。

解决这个问题的办法是**使用 forwarding 指针。也就是说每个对象的头部引入一个新的域（field），叫做 forwarding**。正常状态下，它的值是 NULL，如果一个对象被拷贝到新的空间里以后，就把它的新地址设到旧空间对象的 forwarding 指针里。

当我们访问完 B 以后，对于它所引用的 C，我们并不能知道 C 是否被拷贝，更不知道它被拷贝到哪里去了。此时，我们就可以在 C 上留下一个地址，告诉后来的人，这个地址已经变化了，你要找的对象已经搬到新地方了，请沿着这个新地址去寻找目标对象。这就是 forwarding 指针的作用。下面的图展示了上面描述的过程：

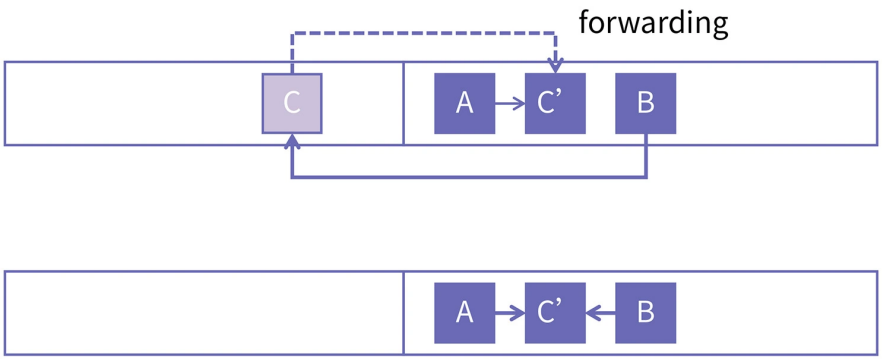

图5

如果你还不太明白，我再给你举一个形象点儿的例子：你拿到一张画，上面写着武穆遗书在临安皇宫花园里。等你去花园里找到一个盒子，却发现里面的武穆遗书已经不在了，里面留了另一幅画，告诉你它在铁掌峰第二指节。显然，有人移动过武穆遗书，并把新的地址告诉你了，等你第二次访问，到达临安的时候，根据新的地址就能找到真正的武穆遗书了。

到这里，我们就可以将 copy gc 的算法彻底完成了，完整的算法伪代码如下所示：


```
void copy_gc() {
    for (obj in roots) {
        *obj = copy(obj);
    }
}
obj * copy(obj) {
    if (!obj.visited) {
        new_obj = to_space.allocate(obj.size);
        copy_data(new_obj, obj, size);
        obj.visited = true;
        obj.forwarding = new_obj;
        for (child in obj) {
```

```
            *child = copy(child);
        }
    }
    return obj.forwarding;
}
```

这样一来，我们就借助深度优先搜索算法完成了一次图的遍历。

我们说，除了深度优先搜索外，广度优先搜索也可以实现图的遍历。那这两种算法有什么区别呢？我们进一步来看。

深度优先对比广度优先

我们已经详细描述了基于深度优先的 copy 算法，为了对比它与广度优先的 copy 算法，我们使用一个例子来进行说明。我把这个例子所使用到的对象定义列在下面：

复制代码

```
class A {
    public B b1;
    public B b2;
    public A() {
        b1 = new B();
        b2 = new B();
    }
}
class B {
    public C c1;
    public C c2;
    public B() {
        c1 = new C();
        c2 = new C();
    }
}

class C {
}
```

假设，在 From 空间中对象的内存布局如下所示：

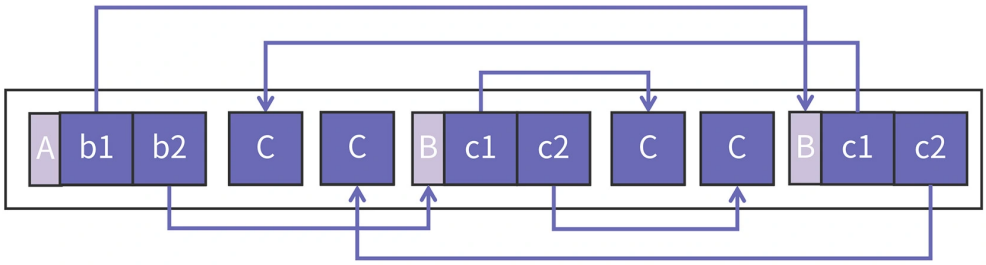

图6

请你注意，图中的空白部分是我为了让图更容易查看而故意加的，真实的情况是每个对象之间的空白是不存在的，它们是紧紧地挨在一起的。

接下来，我们从 A 对象开始深度优化遍历，那么第一个被拷贝进 To 空间的就是 A 对象，如下图所示：

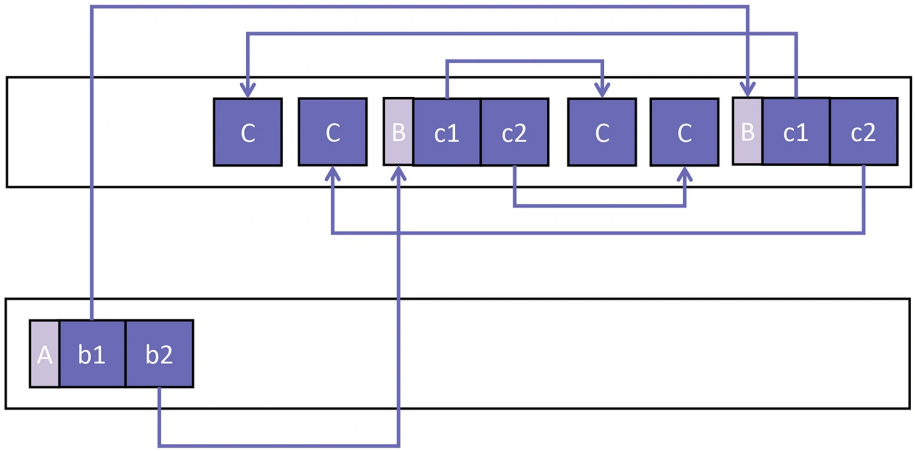

图7

然后对 A 进行扩展，先访问它属性 b1 所引用的对象，把 b1 所指向的对象拷贝到 To 空间，这一步完成以后，空间中对象的情况如下图所示：

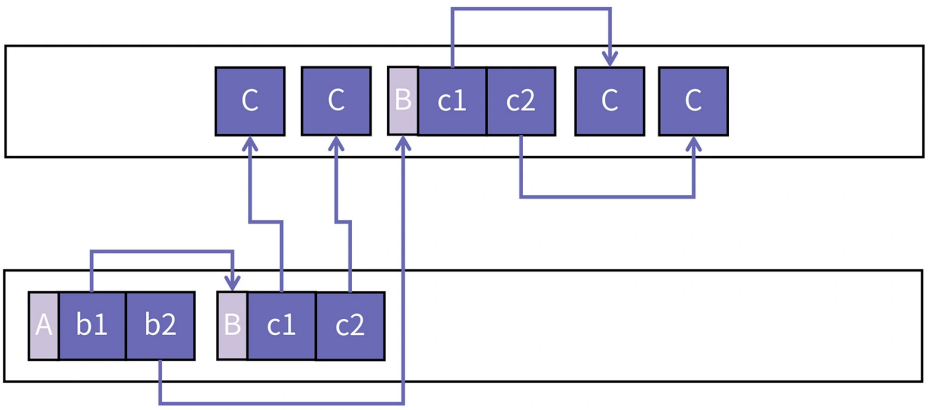

图8

接下来的这一步，是一个关键步骤，由于我们的算法是深度优先遍历，所以接下来会拷贝 c1 所引用的 C 对象，而不 b2 所引用的 B 对象。因为 C 的对象不包含指向其他对象的引用，所以，搜索算法拷贝完 C 对象以后就开始退栈。算法退到上一栈以后，就会继续搜索 B 对象中，c2 所引用的那个 C 对象。经过这两步操作以后，堆空间的情况如下所示：

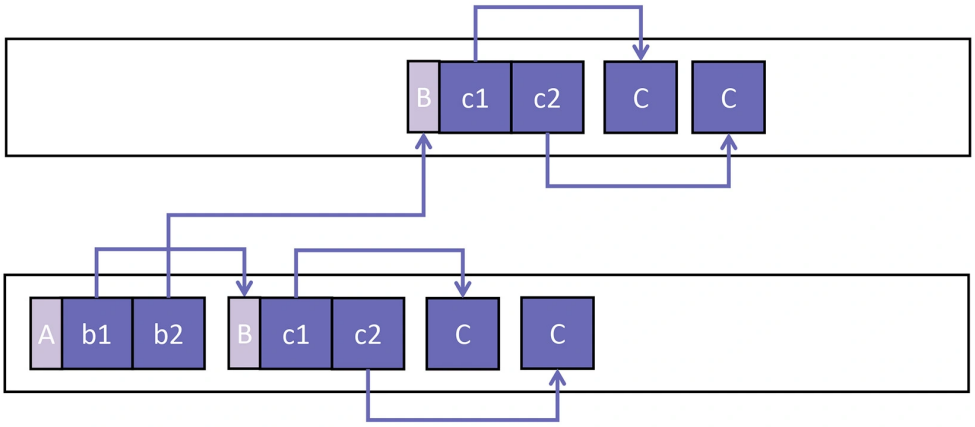

图9

这样 b1 所引用的 B 对象就搜索完了，算法会继续退栈，继续搜索 b2 所引用的 B 对象，当 b2 所引用的对象也全部搜索完成以后，再把 To 空间和 From 空间对调。我们就完成了一次 Copy GC 的全部过程。算法完成以后的堆空间的情况如下所示：

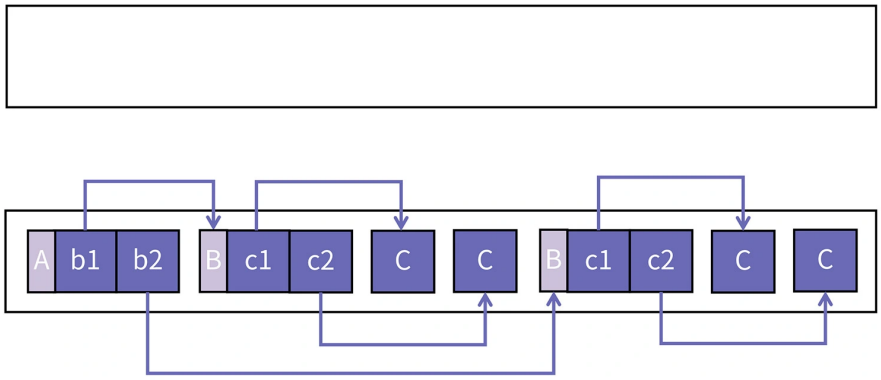

图10

从上面的图片中可以观察到一个特点：**To 空间中的对象排列顺序和 From 空间中的对象排列顺序发生了变化**。从图 5 和图 9 的箭头的样子可以看得出来，图 5 中的箭头是比较混乱的，而图 9 中的箭头则简约很多。因为箭头代表了引用关系，这就说明具有引用关系的对象在新空间中距离更近了。

我们在第 14 节课介绍过，因为 CPU 有缓存机制，所以在读取某个对象的时候，有很大概率会把它后面的对象也一起读进来。通常情况下，我们在写 Java 代码时，经常访问一个变量后，马上就会访问它的属性。如果在读 A 对象的时候，把它后面的 B 和 C 对象都能加载进缓存，那么，a.b1.c1 这种写法就可以立即命中缓存。

这是深度优先搜索的 copy 算法的最大的优点，同时，从代码里也能分析出它的缺点，那就是采用递归的写法，效率比较差。如果你熟悉数据结构的话，应该都知道，深度优先遍历也有非递归实现，它需要额外的辅助数据结构，也就是说需要手工维护一个栈结构。非递归的写法，可以使用以下伪代码表示：

复制代码

```
void copy_gc() {
  for (obj in roots) {
    stack.push(obj);
  }
  while (!stack.empty()) {
    obj = stack.pop();
    *obj = copy(obj);
    for (child in obj) {
      stack.push(child);
    }
  }
```

```
12 }
```

与深度优先搜索相对应的是广度优先搜索。它的优缺点刚好与深度优先搜索相反。如果使用广度优先算法将对象从 From 空间拷贝到 To 空间，那么有引用关系的对象之间的距离就会比较远，这将不利于业务线程运行期的缓存命中。它的优点则在于**可以节省 GC 算法执行过程中的空间，提升拷贝过程的效率**。这部分内容请你作为练习，自己推导一遍广度优先搜索以后的堆空间，你就能掌握的更好了。

广度优先算法节省空间的原理是：**使用 scanned 指针将非递归的广度优先遍历所需的队列，巧妙地隐藏在了 To 空间中**。我使用伪代码写出来，你就能理解了：

复制代码

```
void copy_gc() {
  for (obj in roots) {
    *obj = copy(obj);
  }
  while (to.scanned < to.top) {
    for (child in obj(scanned)) {
      *child = copy(child)
    }
    to.scanned += obj(scanned).size();
  }
}
```

其中，obj（scanned）代表把 scanned 所指向的对象强制转换为一个 obj。

综上所述，**深度优先搜索的非递归写法需要占用额外的空间，但有利于提高业务线程运行期的缓存命中率。而广度优先搜索则与其相反，它不利于运行期的缓存命中，但算法的执行效率更高**。所以 JDK6 以前的 JVM 使用了广度优先的非递归遍历，而在 JDK8 以后，已经把广度优先算法改为深度优先了，尽管这样做需要额外引用一个独立的栈。

到这里，基于 copy 算法的原理，我们就全部讲完了。总体来看，基于 copy 的算法要将堆空间分成两部分：**一部分是 From 空间，一部分是 To 空间。不管什么时刻，总有一半空间是空闲的**。所以，它总体的空间利用率并不高。为了提升空间的利用率，Hotspot 对 copy 算法进行了改造，并把它称为 Scavenge 算法。那它是怎么实现的呢？

Scavenge 算法

我们知道，每次回收中能存活下来的对象占总体的比例都比较小。那么，我们就可以结合这个特点，把 To 空间设置得小一点，来提升空间的利用率。

Hotspot 在实现 copy 算法时做了一些改进。它将 From 空间称为 Eden 空间，To 空间在算法实现中则被分成 S0 和 S1 两部分，这样 To 空间的浪费就可以减少了。Java 堆的空间关系如下图所示：

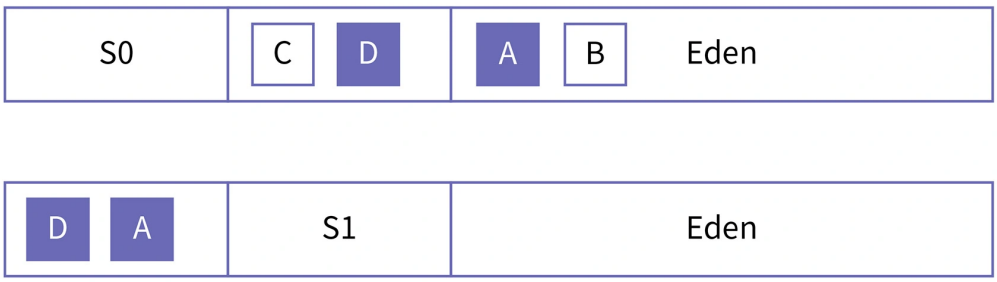

图11

在这张图里，Hotspot 的内存管理器在 Eden 空间中分配新的对象，每次 GC 时，如果将 S0 做为 To 空间，则 S1 与 Eden 合起来成为 From 空间。也就是说 To 空间这个空闲区域就大大减小了，这样可以提升空间的总体利用率。

Scavenge 算法是简单 copy 算法的一种改进。在这种算法中，人们习惯于把 S0 和 S1 称为**幸存者空间（Survivor Space）**。配置 Survivor 空间的大小是 JVM GC 中的重要参数，例如：-XX:SurvivorRatio=8，代表 Eden:S0:S1=8:1:1。

讲到这，我们清楚地了解了 Scavenge 算法的原理和来龙去脉。由此我们也容易推知，基于 Copy 的 GC 算法有以下特点：

1. 对象之间紧密排列，中间没有空隙，也就是没有内存碎片；
2. 分配内存的效率非常高。因为每次分配对象都是把指针简单后移即可，操作非常少，所以效率高；
3. 回收的效率取决于存活对象的多少，如果存活对象比较多，那么回收的效率就差，如果存活的对象少，则回收效率高。如果对象的生命周期比较短，也就是说存活的时候比较

短，那么在进行 GC 的时候，存活的对象就会比较少，这种情况下采用基于 copy 的 GC 算法是比较高效的；

4. 内存利用率并不高。因为在任一时刻总有一部分空间是无非被使用的，Scavenge 算法也只能缓解这个问题，而不能彻底解决，这是由算法的设计所决定的；
5. copy 算法需要搬移对象，所以需要业务线程暂停。

总结

这节课我们重点学习了基于 copy 的 GC 算法的基本原理和它的具体实现。

因为 copy 算法每一次都会搬移对象，在搬移的过程中就已经完成了内存的整理，所以对象与对象之间是没有空隙的，也就是说没有内存碎片。这同时也让内存分配的实现非常简单：**我们只需要记录一个头部指针，有分配空间的需求就把头部指针向后移就可以了**。因为后一个对象是顶着前一个对象分配的，所以，这种方式也叫做**碰撞指针**。

接下来，我们重点研究了 Java 对象的内存布局，从这里我们知道，Java 对象并不只包含用户定义的字段，还包括了对象头和 Klass 指针。其中 Klass 用于描述 Java 对象的类型，它还记录了 Java 对象的布局信息，来指示对象中哪一部分是值，哪一部分是引用。

然后就是我们这节课的重点了。使用深度优先搜索算法对活跃对象进行遍历，在遍历的同时就把活跃对象复制到 To 空间中去了。活跃对象有可能被重复访问，所以人们使用 forwarding 指针来解决这个问题。**图的遍历分为深度优先搜索和广度优先搜索**，我们对两种做法都加以讲解，并对比了它们的特点：

深度优先搜索的递归写法实现简单，但效率差，非递归写法需要额外的辅助数据结构，但它能使业务线程运行时有更好的空间局部性，有利于提高缓存命中率。

广度优先搜索的实现可以借助 To 空间做为辅助队列，节约空间。但不利于业务线程的缓存命中率。

最后，我们展示了 Hotspot 中的真实做法，也就是 Scavenge 算法，并总结了 copy 算法的五个特点：**没有内存碎片、分配效率高、回收效率取决于存活对象比例、总的内存利用率不高和需要暂停**。

思考题

我们提到，copy 算法需要暂停业务线程，那么假设业务线程很多，我们应该怎么样通知业务线程停下来呢？提示：参考加餐二中提到的信号机制和第 5 节课所讲的协程切换时机。欢迎在留言区分享你的想法，我在留言区等你。

吊打面试官

- 请谈一下Scavenge算法。

这道题目是属于比较直接的题目。但经过了这节课的学习，你就知道了，这道题的本质其实是在考察图算法。所以回答的内容要包含对图算法的描述。

你可以分4点来回答，第一点你要回答，堆中的对象是如何抽象成图的；接着第二点呢，你要讲一讲深度优先和广度优先算法；第三点你可以谈谈，使用forwarding指针处理重复访问对象的问题；还有最后一点，如果你能说出Scavenge算法可以多线程执行，这将成为你面试中的加分项！那么它具体的做法是多个线程并发执行Scavenge，每个线程都有一个线程本地栈，用于进行深度优先搜索。只要做好并发控制，这种做法可以充分利用计算资源提升算法执行效率。

高频面试真题

极客时间

好啦，这节课到这就结束啦。欢迎你把这节课分享给更多对计算机内存感兴趣的朋友。我是海纳，我们下节课再见！

上一篇 19 | 垃圾回收：如何避免内存泄露？

下一篇 21 | 分代算法：基于生命周期的内存管理

精选留言 (3)

写留言

费城的二鹏
2021-12-13

通过信号量通知，然后在编译生成的代码中插入检查点。用于检查信号量，判断是否需要停顿。

作者回复: 检查点是对的。这种检查点的机制和信号有点像，都是业务线程要主动去处理。

一子三木
2021-12-13

safepoint?

展开

作者回复: 是的。但你知道safepoint是怎么发生作用的吗？

送过快递的码农
2021-12-13

老师，关于这个我有两个疑问
1，this是不是就是对象头部最顶端的内存地址
2，我们在复制算法执行完之后，对象地址发生改变。我们业务线程里面的栈里面的引用，啥时候变更的呢？比如Object obj = new Object(); 这个被多个线程栈里面都有这个地址，这个值更新的问题，是标志的时候，通过记一个列表，然后统一更新的么？这个感觉也...
展开

作者回复: 1. this是Java层面的概念。其实我们不应该去管它的地址是什么。但如果你非要使用Unsafe对它进行访问，那么，是的，它指向的是对象头部的地址。
2. 栈上的指针也是在对象拷贝阶段更新的呀。这和普通对象有什么不同呢？都是一个二维指针罢了。并不复杂，它没有什么特别之处。非要说有的话，那就是编译器要操心栈上哪些地方是引用，哪些地方是引用。这就是栈的OopMap，这个由编译器操心。内存管理器是不必操心的。