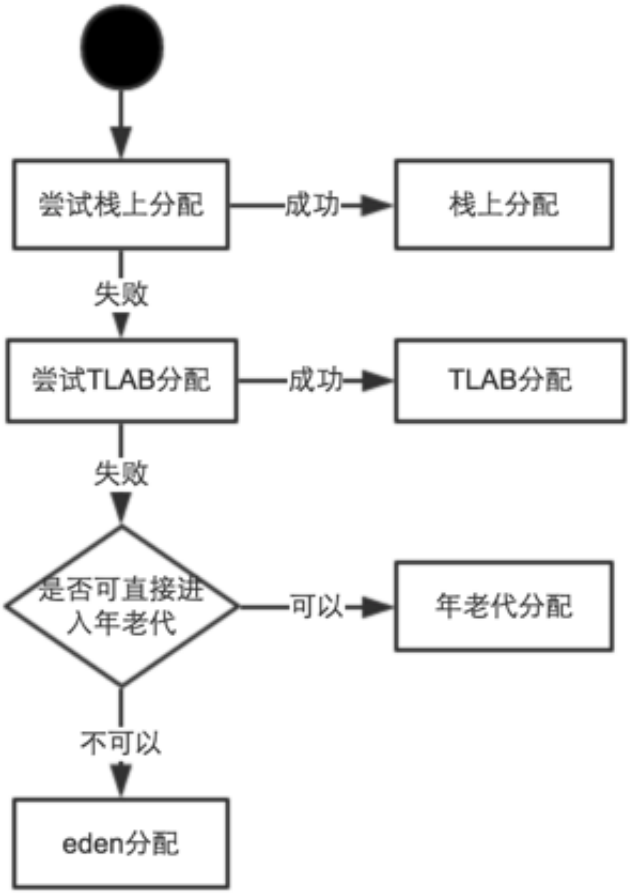

编者按：昨天，在“JVM 研究”微信群里，占小狼同学发来一篇文章：JVM源码分析之线程局部缓存TLAB（http://www.jianshu.com/p/cd85098cca39）。

以下内容主要根据大家的问题和RednaxelaFX、你假笨的分享整理。

TLAB的目的是在为新对象分配内存空间时，让每个Java应用线程能在使用自己专属的分配指针来分配空间，均摊对GC堆（Eden区）里共享的分配指针做更新而带来的同步开销。

TLAB只是让每个线程有私有的分配指针，但底下存对象的内存空间还是给所有线程访问的，只是其它线程无法在这个区域分配而已。当一个TLAB用满（分配指针top撞上分配极限end了），就新申请一个TLAB，而在老TLAB里的对象还留

在原地什么都不用管——它们无法感知自己是否是曾经从TLAB分配出来的，而只关心自己是在Eden里分配的。

TLAB简单来说本质上就是**【三个指针】**：start，top 和end （实际实现中还有一些额外信息但这里暂不讨论）。

其中start 和end 是占位用的，标识出Eden 里被这个TLAB 所管理的区域，卡住Eden 里的一块空间说其它线程别来这里分配了。而top 就是里面的分配指针，一开始指向跟start 同样的位置，然后逐渐分配，直到再要分配下一个对象就会撞上end 的时候就会触发一次TLAB refill。

要注意 TLAB 这个词其实有两层意思：一个是指存在管理 Java 线程的元数据对象JavaThread 里的ThreadLocalAllocBuffer 对象，它持有上述三个指针，仅用于管理用而不存储对象自身；另一个是指在 Eden 中分配出来的、被一个线程的ThreadLocalAllocBuffer 所管理的一块空间，这才是实际存放对象的地方。本讨论不特地指出的时候会自由混用这两层意思，把它们当作一个整体来看待。

TLAB refill包括下述几个动作：

将当前 TLAB 抛弃（retire）掉。这个过程中最重要的动作是将 TLAB 末尾尚未分配给 Java 对象的空间（浪费掉的空间）分配成一个假的“filler object”（目前是用 int[] 作为 filler object）。这是为了保持 GC 堆可以线性 parse（heap parseability）用的。

从 Eden 新分配一块裸的空间出来（这一步可能会失败）。

将新分配的空间范围记录到 ThreadLocalAllocBuffer 里。

将当前 TLAB 抛弃（retire）掉。这个过程中最重要的动作是将 TLAB 末尾尚未分配给 Java 对象的空间（浪费掉的空间）分配成一个假的“filler object”（目前是用 int[] 作为 filler object）。这是为了保持 GC 堆可以线性 parse（heap parseability）用的。

从 Eden 新分配一块裸的空间出来（这一步可能会失败）。

将新分配的空间范围记录到 ThreadLocalAllocBuffer 里。

TLAB refill不成功（Eden 没有足够空间来分配这个新 TLAB）就会触发YGC。

注意“撞上”指的是在某次分配请求中，top + new_obj_size >= end 的情况，也就是说在被判定“撞上”的时候，top 常常离end 还有一段距离，只是这之间的空间不足以满足新对象的分配请求new_obj_size 的大小。这意味着在触发 TLAB refill 的时候，有可能会浪费掉位于该 TLAB 末尾的一部分空间：该 TLAB 已经占用了这块空间，所以其它线程无法在这里分配 Java对象，但该 TLAB 要 refill的话，它自己也不会在这块空间继续分配 Java 对象，从应用层面看这块空间就浪费了。

每次分配 TLAB 的大小不是固定的，而是每个线程根据该线程启动开始到现在的历史统计信息来自己单独调整的。如果一个线程上跑的代码的内存分配速率非常高，则该线程会选择使用更大的 TLAB 以达到均摊同步开销的效果，反之亦然；同时它还会统计浪费比例，并且将其放入计算新 TLAB 大小的考虑因素当中，把浪费比例控制在一定范围内。

GC 很重要的一点是对 heap parseability 的依赖。GC 做某些需要线性扫描堆里的对象的操作时，需要知道堆里哪些地方有对象而哪些地方是空洞。一种办法是使用外部数据结构，例如 freelist 或者 allocation BitMap 之类来记录哪里有空洞；另一种办法是把空洞部分也假装成有对象，这样 GC 在线性遍历时会看到一个“对象总是连续分配的”的假象，就可以以统一的方式来遍历：遍历到一个对象时，通过其对象头记录的信息找出该对象的大小，然后跳到该大小之后就可以找到下一个对象的对象头，依此类推。HotSpot 选择的是后者的做法，假装成有对象的这种东西就叫做 filler object（填充对象）。

实现代码上，
TLAB 的慢速分配和重新申请空间的逻辑在这里：
http://hg.openjdk.java.net/jdk8u/jdk8u/hotspot/file/cf85f331361b/src/share/vm/gc_interface/collectedHeap.cpp#l264

申请好了空间并且 zero 完之后就会设置进 TLAB 里：

http://hg.openjdk.java.net/jdk8u/jdk8u/hotspot/file/cf85f331361b/src/share/vm/gc_interface/collectedHeap.cpp#l304

TLAB 里的 filler object 是这样用的：
http://hg.openjdk.java.net/jdk8u/jdk8u/hotspot/file/cf85f331361b/src/share/vm/memory/threadLocalAllocBuffer.cpp#l110

CollectedHeap 里的裸 allocate 动作是不关心分配的东西是什么类型的，只管在 GC 堆里看有没有地方可以分配，有的话 bump 分配指针并返回 bump 前的指针。TLAB 从 Eden 重新分配空间就是问 CollectedHeap 再 allocate 一块这样的裸的空间，然后把这块空间的首尾记录到自己的 start 和 end 里去。

另外建议把 TLAB 翻译为线程私有分配区，而不是线程局部分配缓存这样词对词的直译。毕竟 TLAB 并不是一个缓存，而且它的重点也不是局部，而是让那个分配指针成为线程私有的东西。

无论是加锁还是 CAS，HotSpot 的共享堆分配都是用碰撞指针（pointer bumping / bump-the-pointer）来做的。加锁跟 bump 不在一个层面上，不应该并列。锁或者 CAS 只是同步的机制，实际想要做的事情都一样是 bump pointer。如果在不需要与其它线程竞争的条件下，bump pointer 就不用同步保护。

例如在 TLAB 里，又例如在 PLAB 里，又例如在共享部分但在 safepoint 中没有竞争的情况下。

PLAB 也是个非常有趣的东西，提到 TLAB 的话也可以顺带说下。HotSpot 里的 TLAB 是只在 Eden 里分配的，用于给新建的小对象用。（本来其实也有考虑让 TLAB 在任意位置分配，但后来没实现）。PLAB 则是在 old gen 里分配的一种临时的结构。就是的promotion LAB。

在多 GC 线程并行做 YGC 的时候，大家都要为了晋升对象而在 old gen 里分配空间，于是 old gen 的分配指针就热起来了。大量的竞争会使得并行度降低，所以跟 TLAB 用同样的思路，old gen 在处理 YGC 的晋升对象的分配也一样可以用（GC）线程私有的分配区。这就是 PLAB。另外在 CMS 里 old gen 的剩余空间

不是连续的，而是有很多空洞。这些剩余空间是通过 freelist 来管理的。

如果 ParNew 要把对象晋升到 CMS 管理的 old gen，不优化的话就得在 freelist 上做分配。于是就可以通过类似 PLAB 的方式，每个 GC 线程先从 freelist 申请一块大空间，然后在这块大空间里线性分配（bump pointer）。这样就既降低了对分配指针/freelist的竞争，又可以降低 freelist 分配的频率而转为用线性分配。