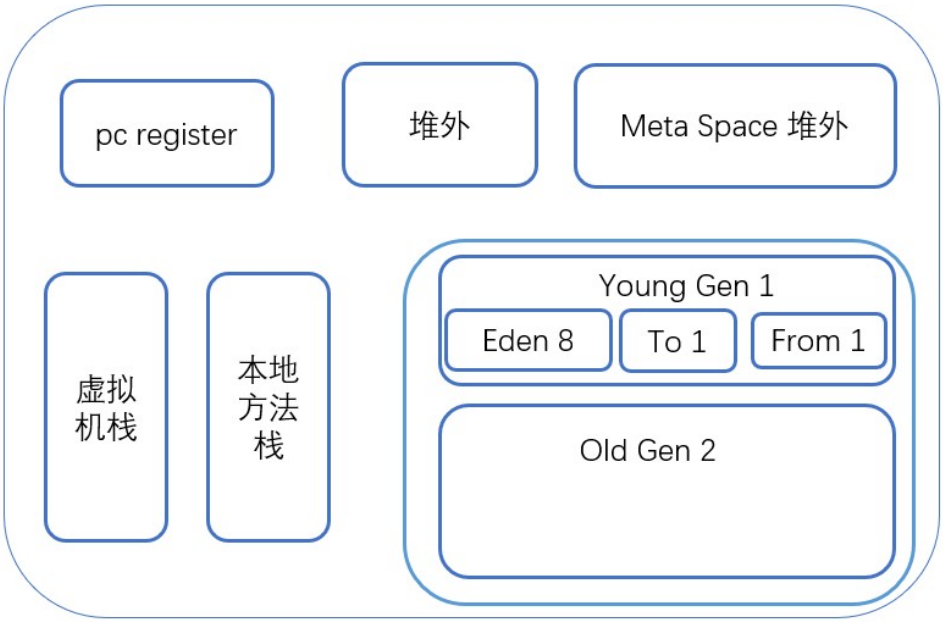
我们来聊聊JVM的GC吧

> 你的时间在哪里，你的成就就会在哪里

前言

在21年的时候写了一篇《JVM学习笔记(一) 初遇篇》，我们回忆一下这篇文章的内容, 我们讲了JVM运行时区域的组成，也就是一个JVM进程将申请到的区域划分了几部分，但是这个问题没有统一答案，在JDK 8下面有不同的垃圾回收器，不同的垃圾回收器策略不一样，比如JDK 8 默认的垃圾回收器，Parallel Scavenge(作用于年轻代)和Parallel Old(作用于老年代)，这两个垃圾回收器作用于堆，除此之外，我们知道JDK 8下面还有元空间，元空间在堆外，所以在JDK 8下面，JVM进程将申请的区域分割成以下几部分:

- 程序计数器
- 虚拟机栈
- 本地方法栈
- 堆
  - 年轻代(Young, 里面进一步分为eden space 、from space 、to space，内存分配比例为，8:1:1)
  - 老年代
- 元数据区域(Meta space)
- 堆外内存(通过Unsafe去申请)



在IDEA里面对应的启动类加上-XX:+PrintGCDetails，控制台会输出如下格式的GC日志:

```
Heap
 PSYoungGen      total 152576K, used 23609K [0x0000000716600000,
0x0000000721000000, 0x00000007c0000000)
  eden space 131072K, 18% used
[0x0000000716600000,0x0000000717d0e458,0x000000071e600000)
  from space 21504K, 0% used
[0x000000071fb00000,0x000000071fb00000,0x0000000721000000)
  to   space 21504K, 0% used
[0x000000071e600000,0x000000071e600000,0x000000071fb00000)
 ParOldGen       total 348160K, used 0K [0x00000005c3200000, 0x00000005d8600000,
0x0000000716600000)
  object space 348160K, 0% used
[0x00000005c3200000,0x00000005c3200000,0x00000005d8600000)
 Metaspace       used 4312K, capacity 4752K, committed 4992K, reserved 1056768K
  class space    used 481K, capacity 546K, committed 640K, reserved 1048576K
```

通过上面的JVM运行时区域图，我想可以就可以回答为什么JVM带GC的为什么还会有内存泄漏，原因在于在Java中我们还是可以自己手动的分配释放内存，也就是通过Unsafe:

```
// 申请内存
Unsafe.getUnsafe().allocateMemory();
// 释放内存
Unsafe.getUnsafe().freeMemory();
```

探究其实现的话，Unsafe.allocateMemory的实现是一个native函数:

```
private native long allocateMemory0(long bytes);
```

在Unsafe.cpp里面我们可以看到对应的实现:

```
UNSAFE_ENTRY(jlong, Unsafe_AllocateMemory(JNIEnv *env, jobject unsafe, jlong size))
  UnsafeWrapper("Unsafe_AllocateMemory");
  size_t sz = (size_t)size;
  if (sz != (julong)size || size < 0) {
    THROW_0(vmSymbols::java_lang_IllegalArgumentException());
  }
  if (sz == 0) {
    return 0;
  }
  sz = round_to(sz, HeapWordSize);
  void* x = os::malloc(sz, mtInternal);
  if (x == NULL) {
    THROW_0(vmSymbols::java_lang_OutOfMemoryError());
  }
  //Copy::fill_to_words((HeapWord*)x, sz / HeapWordSize);
  return addr_to_java(x);
UNSAFE_END
```

这样做的好处在哪呢，直接绕开JVM，向操作系统直接申请内存。我们已知的是在堆里面是由JVM管理的，那么在回收内存的时候到现在为止都无可避免的要Stop The World(尽管ZGC的设计目标是停顿时间不超过10ms，停顿时间不会随着堆的大小，或者活跃对象的增加)。

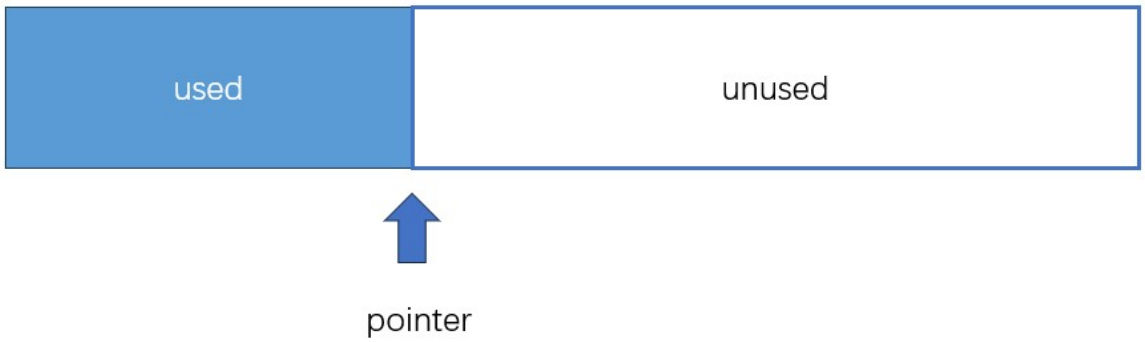
简称STW，所谓STW也就是整个应用程序都会被暂停。那么从这一点来推断选择将数据放到堆外，我们就可以在GC时减少回收停顿对于应用的影响。除此之外，通常在I/O通信的过程中，会存在堆内内存到堆外内存的数据拷贝操作，对于需要频繁进行内巡检数据拷贝且生命周期比较短的暂存数据，都建议放到堆外内存。

据说值类型的到来会让Java内存占用更低，这是Valhalla 项目努力的方向，之前看过一些回答据说C#使用值类型(struct)可以做到无gc，但是我对C#并没有那么多的了解。说会Unsafe的allocateMemory，直接申请内存相对于JVM内存会更快吗?

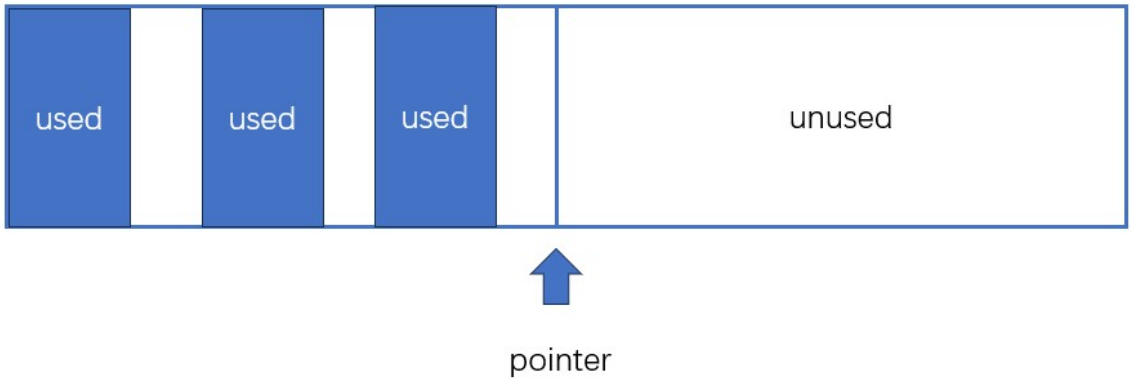
new 对象的分配内存步骤

在JVM中new 对象是如何分配内存的呢? 在为这个对象分配内存之前，首先要加载这个类并且计算出来这个对象需要占据多少内存，类相关的信息会进入到元空间里面，我们这里先只关心对象在堆内部的存储，堆内部的存储包括对象头、对象体以及内存对齐填充， 内存对齐不熟悉请参看《JMH探索学习笔记(一)》里面详细讲了内存对齐的意义，Java默认按8字节对齐，也就是说一个对象所占用的内存大小一定回收8的倍数，在Java里面不能自由的控制这种对齐，Java领域大名鼎鼎的Disruptor和Netty就有效的利用了内存对齐技术来做加速。

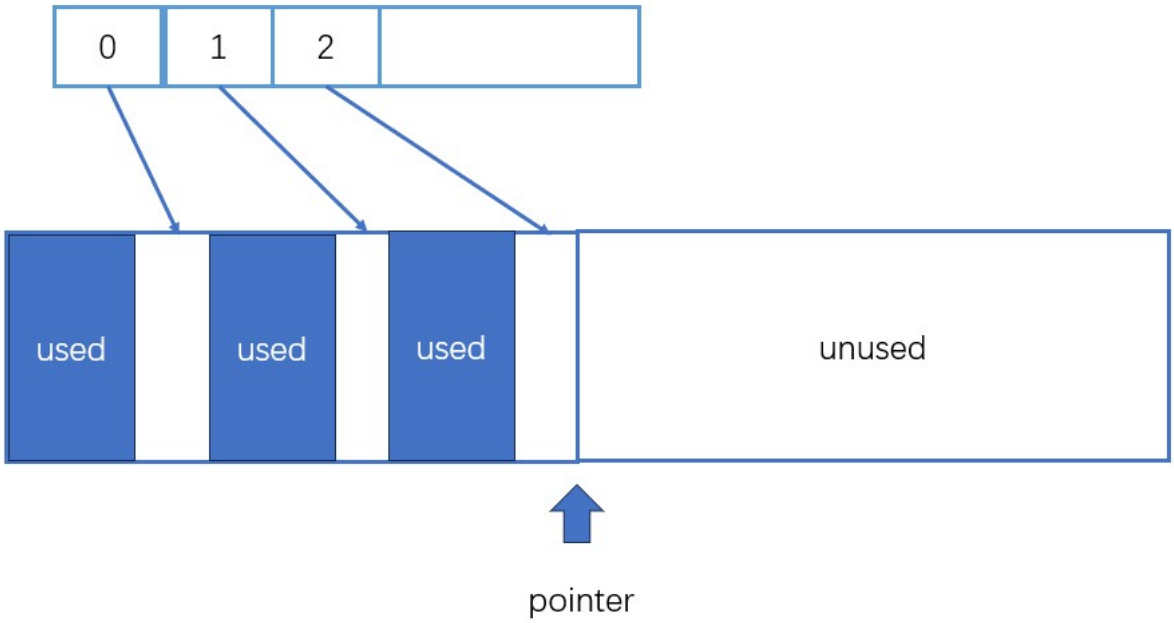
那如果让我们自己来设计分配内存，最简单的思路如下图所示:



首先我们有一个指针，每次分配内存的时候，指针总是指向已使用内存的边界，每次分配内存先计算所需内存的大小，然后CAS更新上图的指针，标记已使用内存的边界，但是内存往往不会这么整齐，原因在于指针在移动的过程，指针左侧已使用内存的部分内存可能已经被回收了，所以真实的情况更接近于下面的图:



如果按照这种方法分配内存，我们可以直观的看到内存没有被充分利用，这种简单的内存分配方法也被称为撞针分配，也称线性分配。这种分配策略存在明显的缺陷，于是我们就需要打第一道补丁，也就是freeList，也就是记录内存被释放区域的内存地址，下次再分配内存的时候优先从这个FreeList里面寻找合适的内存大小进行分配，如果找不到合适的区域，再进行撞针分配。



这个方案看起来完整了一些，但是我们目前讨论的都只有一个线程请求分配释放内存，目前的应用是多线程的情况下，都从主内存中分配，CAS更新重试过于频繁就会导致效率低下，我们需要接着在这个方案上打补丁，在当下的应用中，一般都使用线程池来复用线程，在这种情况下，一般来说每个线程分配的内存是比较稳定的，这里稳定的意思是每次分配对象的大小，每轮GC分配区间的分配对象个数以及总大小。所以我们可以考虑每个线程每个线程分配内存后，将这块内存保留下来，用于下次分配，这样就不用每次都从主内存分配了。

如果能估算每轮GC内每个线程使用的内存大小，则可以提前分配内存给线程，这样就更能提高分配效率。这种内存分配方式在Java里面就是TLAB(Thread Local Allocate Buffer)。

这里我们不考虑栈上分配，讨论这个会让问题变得复杂，我们这里只考虑无法在栈上分配需要共享的对象。对于Hotspot JVM实现，所有的GC算法的实现都是一种对于堆内存管理，也就是都实现了一种堆的抽象，都实现了接口 CollectedHeap。当为一个对象分配内存空间的时候，如果当前线程的TLAB大小足够，那么从线程当前的TLAB中分配；如果不够，但是当前TLAB剩余空间小于最大浪费空间限制(这是一个动态的值)， 则从堆上(一般是Eden区)重新申请一个新的TLAB进行分配，粗略的说，可以理解为也就是看再申请一个TLAB划算不划算。如果不划算，就直接在TLAB外进行分配，TLAB外的分配策略，不同的GC算法不同。

这么一看相对于JVM提前申请好的内存，Unsafe.alloteMemory如果没有做池化其实内存分配速度是赶不上JVM分配的，想来优势就只有能够避免GC停顿带来的影响了。

说回GC

一般我们提到的GC的语义大致有三种:

- Garbage Collection 垃圾收集技术
- Garbage Collector: 垃圾收集器
- Garbage Collecting: 垃圾收集动作

在维基百科中，对Garbage Collection 的定义是: “In computer science, garbage collection (GC) is a form of automatic memory management. The *garbage collector* attempts to reclaim memory that was allocated by the program，but is no longer referenced”， 也就是在说，在计算机科学中，垃圾收集技术是一种自动内存管理的一种形式。垃圾回收器尝试回收已分配给程序但不再被使用的内存。

所以在我看来垃圾收集技术(Garbage Collection)是包括垃圾收集器(Garbage Collector)和垃圾收集动作(Garbage Collecting) 垃圾收集动作的。 那么在我看来收集动作由两个部分组成，一个是识别垃圾，一个是收集垃圾。垃圾收集器是垃圾回收动作的实现。

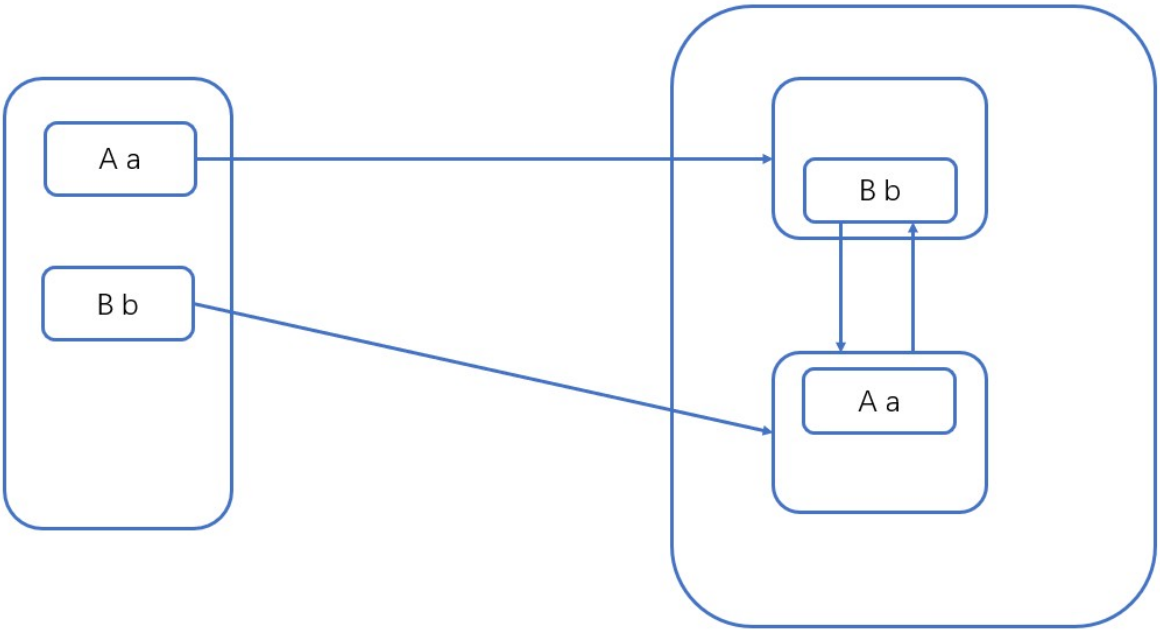
Garbage Collecting 垃圾收集动作

识别垃圾

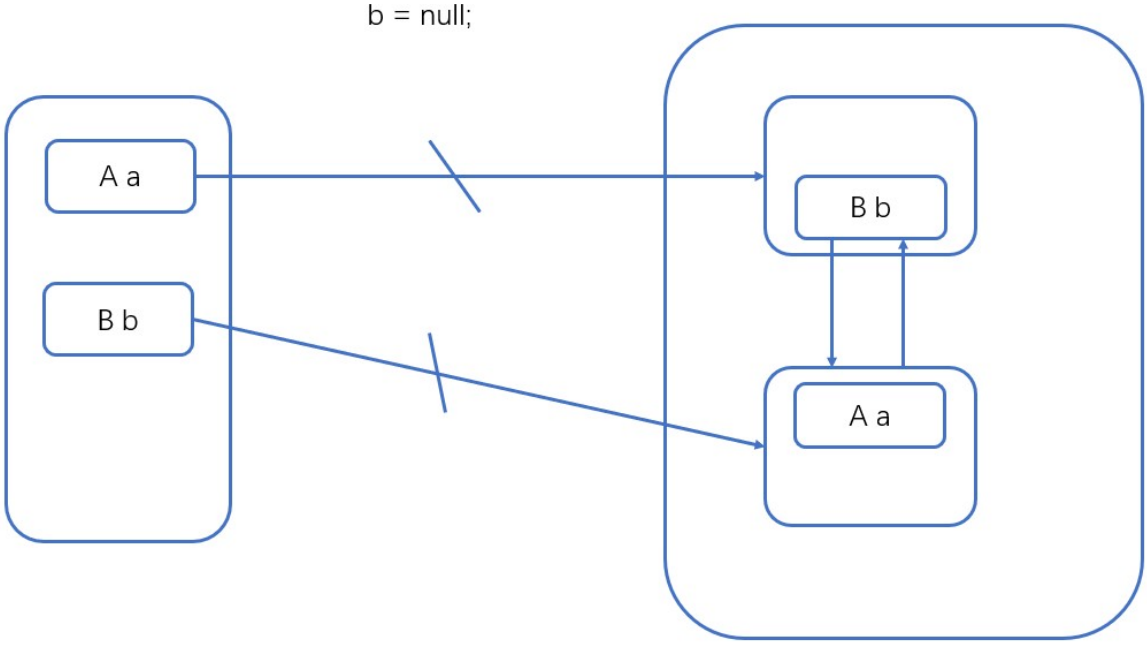
- 引用计数法(Reference Counting): 对每个对象的引用进行计数，每当有一个地方引用它时+1，引用失效-1。引用的计数放在对象头里，大于0的对象被认为是存活对象。在《JVM学习笔记(一) 初遇篇》里面我们提到，这种识别垃圾的算法，碰到循环引用的问题无法解决循环引用问题，所谓循环引用就是指，A类里面有一个成员变量是B类，B类有一个成员变量是A类，如下面代码所示:

```
public class A {
    public B b;
}
public class B {
    public A a;
}
public static void main(String[] args) {
  A a = new A();
  B b = new B();
  a.b = b;
  b.a = a;
  a = null;
  b = null;
}
```

现在a引用了b，引用+1，b引用了a，引用 + 1。虽然我们将a 和 b 都置为了null，但只是引用置为了null，我的脑海出现这样一幅图:



a = null;
b = null;



我在想这幅图的时候在想引用变量a为null的时候，会不会也将B指向的对象中的成员变量也置为null，于是取消了循环引用，但仔细想了想，引用变量a 指向 A实例在堆里面的地址，B的实例b中的成员变量a也指向这个地址，虽然引用变量a取消了这个引用，但是没有取消B的实例b中成员变量a的引用，因此还是造成了循环依赖。

但是现在来看，引用计数法是可以解决循环引用的，也就是Recycler 算法，但是在多线程环境下，引用计数变更也要进行昂贵的同步操作，性能较低，早期的编程语言会采用此算法

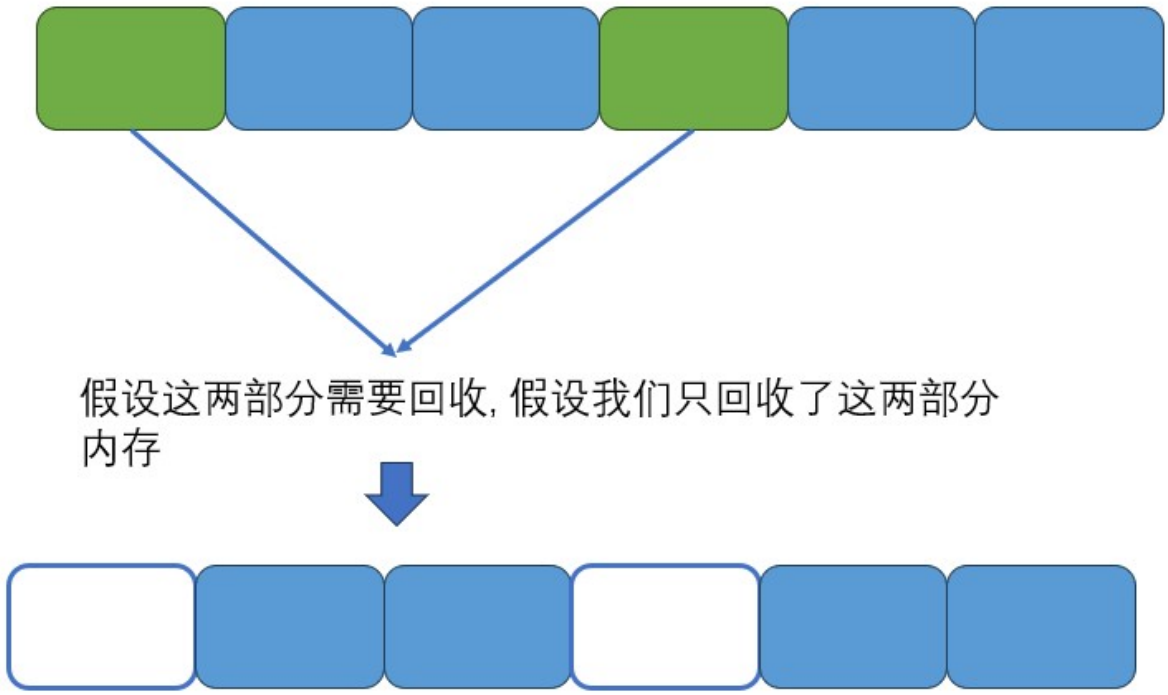
- 由于引用计数法的性能问题，JVM采用的是可达性分析算法，又称引用链算法(Tracing GC)：也就是从GC Roots开始进行搜索，可以被搜索的对象即可为可达对象，需要多次标记才能更加准确的确定不可达对象，整个连通图之外的对象就可以作为垃圾回收掉。 那哪些对象可以做为gc roots呢，在Java语言中，可作为GC Root的对象有以下几种, 这里只列出常见的，具体的可以参看参考文档[8]:
  - 虚拟机栈(栈帧中的本地变量表)中引用的对象
  - 类静态变量引用的对象
  - 类常量引用的对象
  - JNI的native方法栈中引用的对象

- JNI中的global对象
- 活着的线程所使用的对象

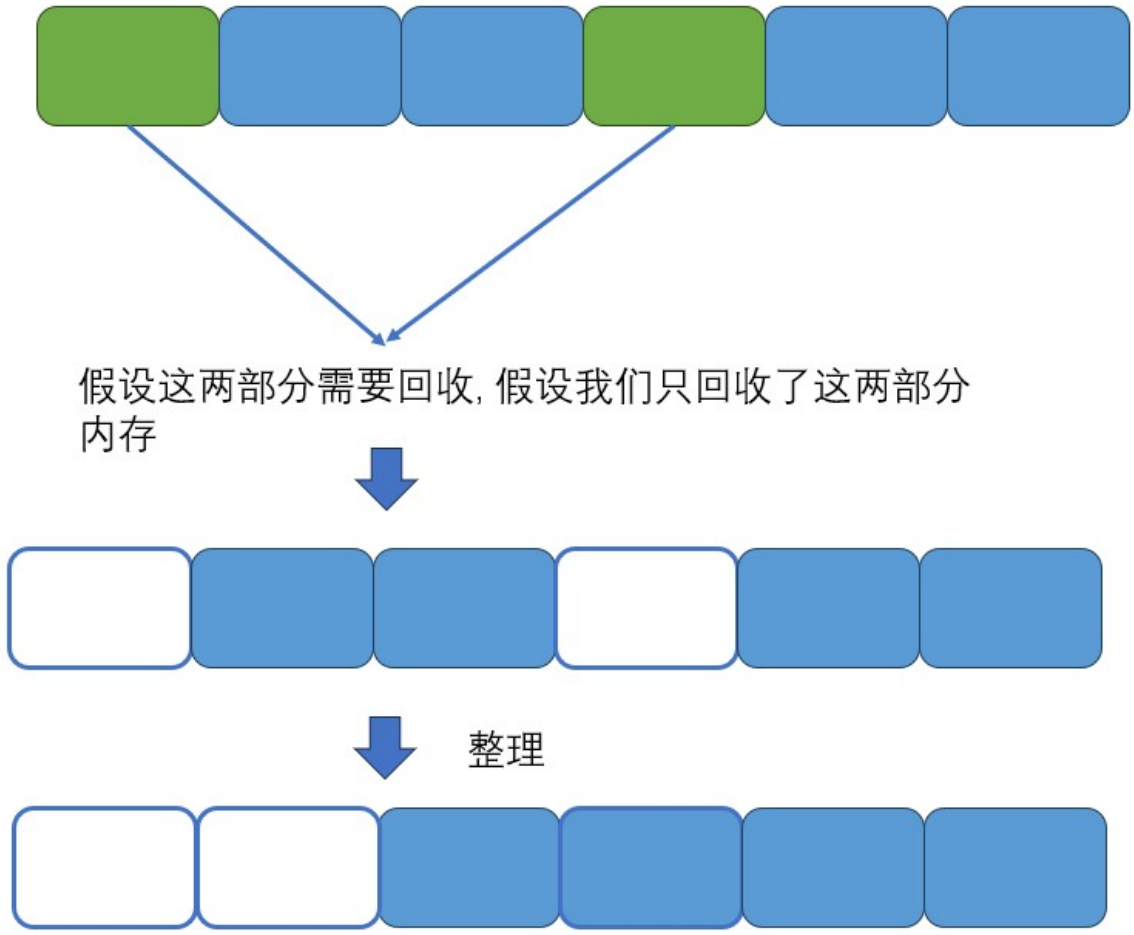
回收垃圾

标记清除算法

现在我们能识别垃圾了，那么下一个问题就是如何回收垃圾，一个选择是将不可达对象的内存释放出来就好:



这样的做法是简单，但是缺点也很明显，就是回收出来的内存碎片严重，如果我们需要一块连续的内存，而回收出来的内存区域不满足我们的需求怎么办，于是只好再打一个补丁，也就是整理, 对内存碎片进行整理:



这也就是"标记-清除"算法，CMS 垃圾回收器就是采用"标记-清除"，那在什么时候进行整理呢， 在CMS垃圾回收器中有两个参数进行控制:

- -XX:+UseCMSCompactAtFullCollection=true 在每次FullGC之后进行内存整理
- -XX:+CMSFullGCsBeforeCompaction = n CMSFullGCsBeforeCompaction 说的是，在上一次CMS并发GC执行过后，到底还要再执行多少次full GC才会做压缩。默认是0，也就是在默认配置下每次CMS GC顶不住了而要转入full GC的时候都会做压缩。

CMS GC在实现上分成前台收集(foreground collector)和后台收集(background collector)， 前台收集触发条件比较简单，一旦遇到对象分配但空间不够，就会直接触发GC来立即进行空间回收，采用的算法是mark sweep(标记-清除)，不压缩。在CMS发生foreground gc才是FullGC。

基于前台收集， 我们可以预设一个场景是在高并发下面，新生代的Eden区会被迅速填满，频繁触发Young GC，而在YoungGC没结束的时候，请求又到达了，这就导致每次YoungGC之后，还有大量对象存活，导致Survivor区放不下的问题，这会导致对象从年轻代进入老年代，如果CMSFullGCsBeforeCompaction 次数过大就会导致内存碎片过大，一旦连续空间不足以容纳这些对象就会触发FullGC，直到超过我们设置的次数才会去整理。 这个调优案例来自于参考资料[9] , 这篇文章虽然提供了一个调优案例但是在CMS GC的回收方式是没有讲CMS的两类收集类型, 也就是前台收集(foreground collector)和后台收集(background collector)。

那么基于这个案例来说，我们就可以做如下调优通过gc日志判断存活对象的大小，然后扩容Young区，我们可以在总的Heap大小不变的情况下，适当的增大Young区，一般情况下老年代的大小应当为活跃对象的2-3倍左右，后台收集稍微复杂点我们在下面的什么时候会触发GC一节单独讲。这个参数来自参考文档[6].

整理与复制算法

标记-整理算法的主要目的就是解决在非移动式回收器中都会存在的碎片化问题，也分为两个阶段，第一阶段与 Mark-Sweep 类似，第二阶段则会对存活对象按照整理顺序（Compaction Order）进行整理。主要实现有双指针（Two-Finger）回收算法、滑动回收（Lisp2）算法和引线整理（Threaded Compaction）算法等。CMS垃圾回收器算是标记-清除-整理，在某些情况

将空间分为两个大小相同的 From 和 To 两个半区，同一时间只会使用其中一个，每次进行回收时将一个半区的存活对象通过复制的方式转移到另一个半区。有递归（Robert R. Fenichel 和 Jerome C. Yochelson提出）和迭代（Cheney 提出）算法，以及解决了前两者递归栈、缓存行等问题的近似优先搜索算法。复制算法可以通过碰撞指针的方式进行快速地分配内存，但是也存在着空间利用率不高的缺点，另外就是存活对象比较大时复制的成本比较高。

Par New 和 Parallel Scavenge 采用的是标记复制算法，CMS垃圾回收器采用的是标记-清除-整理算法，在前台收集的时候是标记清除，后台收集的时候会触发标记-清除-整理。 宏观上来看，新生代的 Young GC、G1和ZGC都基于标记-清理-复制算法，但算法的实现不同导致了巨大的性能差异。

时间和空间开销

三种算法在是否移动对象、空间和时间方面的一些对比，假设存活对象数量为 *L*、堆空间大小为 *H*，则：

	移动对象	空间开销	时间开销
Mark-Sweep	否	低（有碎片）	mark 阶段与存活对象的数量成正比 *O(L)*，sweep 阶段与整堆大小成正比 *O(H)*
Mark-Compact	是	低（无碎片）	mark 阶段与存活对象的数量成正比 *O(L)*，compaction 阶段与存活对象的大小成正比 *O(L)*
Copying	是	高	与存活对象大小成正比 *O(L)*

声明图片来自于参考文档[6], 把标记(mark)、清除(sweep)、compaction(整理)、复制(copying)这几种动作的耗时放在一起看，大致有这样的关系:

$$\begin{cases} compaction \geq copying > mark > sweep \\ mark + sweep > copying \end{cases}$$

虽然 整理 与 复制 都涉及移动对象，但取决于具体算法，整理可能要先计算一次对象的目标地址，然后修正指针，最后再移动对象。复制则可以把这几件事情合为一体来做，所以可以快一些。另外，还需要留意 GC 带来的开销不能只看 收集 的耗时，还得看 分配 。如果能保证内存没碎片，分配就可以用撞针方式，只需要挪一个指针就完成了分配，非常快。而如果内存有碎片就得用 freelist 之类的方式管理，分配速度通常会慢一些。

Garbage Collector 垃圾收集器

现在让我们说回garbage Collector 垃圾收集器，垃圾回收器负责执行垃圾回收动作，上文我们已经讨论了两种JDK 8下面的垃圾回收器组合:

- Parallel Scavenge(年轻代) + Parallel Old(老年代) 吞吐量优先
- Par New(年轻代) + CMS (老年代) 延时优先

如果经常关注JDK的进展，看到延时优先应当会想起ZGC，ZGC来自 JEP 333 ZGC: A Scalable Low-Latency Garbage Collector (Experimental) , 也是低延时，而且还承诺暂停时间小于10ms，那这就跟CMS垃圾回收器的生态位有点重合，于是CMS垃圾回收器在JDK 被标记弃用，在JDK 14 被移除。 那么我能否既想要吞吐量优先的同时，又低延时呢，这也就是G1了(-XX:+UseG1GC):

> The Garbage-First (G1) garbage collector is targeted for multiprocessor machines scaling to a large amount of memory. It attempts to meet garbage collection pause-time goals with high probability while achieving high throughput with little need for configuration. G1 aims to provide the best balance between latency and throughput using current target applications and environments whose features include:
>
> G1的是为多处理器机器，可以扩展到大量内存。它尝试尽可能(high probability 大概率，在中文里看起来怪怪的)实现垃圾收集暂停时间目标的同时实现高吞吐量，只需要一点配置(别调优了呗，你知道G1有多努力嘛)。G1在当前应用和环境下提供延迟和吞吐量的最佳平衡，包括:
>
> - Heap sizes up to tens of GBs or larger, with more than 50% of the Java heap occupied with live data.
>
>   堆大小可达数十GB或更大,其中超过50%的Java堆被存活数据占用。
> - Rates of object allocation and promotion that can vary significantly over time.
>
>   对象分配和晋升的速率可能随时间而显著变化。
> - A significant amount of fragmentation in the heap.
>
>   堆中存在大量的内存碎片。
> - Predictable pause-time target goals that aren't longer than a few hundred milliseconds, avoiding long garbage collection pauses.
>
>   可预测的暂停时间目标,不超过几百毫秒,避免长时间的垃圾收集暂停。

总体上看GC

从最基本的角度上来看，JVM中垃圾收集算法的核心功能可以被归纳为以下三点:

1. 当应用程序请求分配内存时,GC负责提供所需的内存。提供内存的过程应该尽可能快速。
2. GC需要识别出应用程序不再使用的内存。同样地,这个识别机制应该高效,不能占用过多的时间。这些无法访问的内存通常被称为垃圾。
3. GC将这些回收的内存再次提供给应用程序使用,最好能够 "及时" 提供,也就是说,要尽快完成这个过程。

有很多算法可以满足所有这些要求，但是不幸的是没有银弹，没有完美的算法，JDK提供了好几种垃圾回收器，每种垃圾回收器针对不同的场景进行了优化，们的实现大致决定了吞吐量、延迟和内存占用这三个主要性能指标中的一个或多个的行为,以及它们如何影响Java应用程序。

1. 吞吐量:表示在给定的时间单位内可以完成的工作量。在本讨论中,每单位时间执行更多收集工作的垃圾收集算法是更可取的,允许Java应用程序具有更高的吞吐量。
2. 延迟: 表示应用程序单个操作需要多长时间。专注于延迟的垃圾收集算法试图最小化对延迟的影响。在GC的上下文中,关键问题是它的操作是否会导致暂停、暂停的程度以及暂停的持续时间。
3. 内存占用: 在GC的上下文中,内存占用意味着GC正常运行所需的额外内存,超出了应用程序的Java堆内存使用量。纯粹用于管理Java堆的数据会占用应用程序的资源;如果GC(或更一般地说,JVM)使用的内存量更少,则可以为应用程序的Java堆提供更多内存。

这三个指标是相互关联的:高吞吐量的收集器可能会显著影响延迟(但会最小化对应用程序的影响),反之亦然。较低的内存消耗可能需要使用在其他指标上不太优化的算法。低延迟收集器可能会并发地或以小步骤的方式执行更多工作,作为应用程序执行的一部分,占用更多的处理器资源。

在JDK 18中提供了五种垃圾回收器，分别关注不同的性能指标，下表列出了他们的名称、关注领域以及所需属性的一些概念:

Garbage collector	Focus area	Concepts
Parallel	Throughput	Multithreaded stop-the-world (STW) compaction and generational collection
Garbage First (G1)	Balanced performance	Multithreaded STW compaction, concurrent liveness, and generational collection
Z Garbage Collector (ZGC) (since JDK 15)	Latency	Everything concurrent to the application
Shenandoah (since JDK 12)	Latency	Everything concurrent to the application
Serial	Footprint and startup time	Single-threaded STW compaction and generational collection

我使用Serial的时候给我的感觉确实是启动速度很快。

GC 类型

一般按回收的区域我们可以将GC分为两类，一类是回收堆，一类是回收元空间和堆外。

回收堆

按照是否收集整个堆，我们又可以将GC分为部分收集(Partial GC)和全量收集(Full GC)。部分收集只对某些分代/分区进行回收，那么为什么要分代呢，分代有什么好处呢？ 对传统的、基本的GC实现来说，由于它们在GC的整个工作过程中都要“stop-the-world”，如果能想办法缩短GC一次工作的时间长度就是件重要的事情。如果说收集整个GC堆耗时太长，那不如只收集其中的一部分？

于是就有好几种不同划分堆的方式来实现部分收集，而分代式GC就是其中的一个思路，这个思路基于的基本假设就是大部分对象的生命周期很短(die Yount)，而不在这个大部分的对象，也就是生命周期很长的对象则可能会存活很长时间。 这是对过往的很多应用行为分析得出的一个假设。基于这个假设，如果让新创建的对象都在年轻代分配，然后频繁收集年轻代，则大部分垃圾都能在young gc 中收集。由于年轻代的大小配置通常只占整个GC堆较小的部分，而且较高的对象死活率，让它非常适合使用复制算法来收集，这样不但能降低单次GC的时间长度，还可以提高GC的工作效率。这也就是ZGC走向分代的理由，Oracle的HotSpot VM里的G1 GC，在最初设计的时候是不分代的部分并发+增量式GC，而后来在实际投入生产的时候使用的却也是分两代的分代式GC设计。

我们接着回到部分收集和全量收集，Partial GC又可以进一步细分:

- Young GC：只收集年轻代的GC。
- Old GC: 只收集年老代的GC，只有CMS 的并发收集是这个模式。
- Mixed GC: 收集整个年轻代和部分年老代的GC，只有G1有这个模式。

Full GC就比较简单了: 全量收集的GC，对整个堆进行回收，STW的时间会比较长，一旦发生，影响比较大，也可以叫做Major GC。 各种Young GC的触发原因都是eden区要满了，Parallel Old GC的触发则是在要执行Young GC时候预测其晋升(promote)的对象(object)的总大小(size)超过老生代剩余容量大小。触发CMS GC初始标记(initial marking)是堆(Heap)的老年代使用比例超过某值。G1 GC初始标记的触发条件是Heap使用比率超过某值。Parallel Old GC（Full GC）之前会跑一次Parallel Young GC，主要是为了减轻Full GC 的负担。

回收Meta Space

在讨论这个区域的回收之前，让我们看看这个区里面会存储什么数据，Java 7之前字符串常量池被放到了Perm 区，所有被intern的String都会被存在这里，由于String.intern是不受控的，所以-XX:MaxPermSize的值也不太好设置，经常会出现Java.lang.OutOfMemoryError：PermGen space异常，所以在Java 7之后的常量池等字面量、类静态变量、符号引用等几项都被移动到Heap中，在JDK 8采用MetaSpace替代PermGen。

在最底层，JVM 通过 mmap 接口向操作系统申请内存映射，每次申请 2MB 空间，这里是虚拟内存映射，不是真的就消耗了主存的 2MB，只有之后在使用的时候才会真的消耗内存。申请的这些内存放到一个链表中 VirtualSpaceList，作为其中的一个 Node。

在上层，MetaSpace 主要由 Klass Metaspace 和 NoKlass Metaspace 两大部分组成。

- **Klass MetaSpace：** 就是用来存 Klass 的，就是 Class 文件在 JVM 里的运行时数据结构，这部分默认放在 Compressed Class Pointer Space 中，是一块连续的内存区域，紧接着 Heap。Compressed Class Pointer Space 不是必须有的，如果设置了 `-XX:-UseCompressedClassPointers`，或者 `-Xmx` 设置大于 32 G，就不会有这块内存，这种情况下 Klass 都会存在 NoKlass Metaspace 里。
- **NoKlass MetaSpace：** 专门来存 Klass 相关的其他的内容，比如 Method，ConstantPool 等，可以由多块不连续的内存组成。虽然叫做 NoKlass Metaspace，但是也其实可以存 Klass 的内容，上面已经提到了对应场景。

也就是MetaSpace负责存储类的元数据，类和其元数据的生命周期与其对应的类加载器相同，只要类的类加载器是存活的，在MetaSpace中的类元数据也是存活的，不能被回收。每个加载器有单独的存储空间，通过 ClassLoaderMetaspace 来进行管理 SpaceManager* 的指针，相互隔离的。

写在最后

这篇文章叫当我们说起GC调优时，我的预期是从本质入手，理解GC，然后就能从本质上定位出来问题在哪里，我不愿意直接告诉你一些结论，原因在于，在这个信息泛滥的时代，有一些被“奉为圭臬(nie)”的结论是错的，相比于结论，我更愿意更关注过程，我更关注思维模型，我有的时候觉得在技术是一片海洋，如果只是记结论，恐怕会淹没在技术的海洋里面，而追寻本质，探本穷源则是在技术的海洋里面建造自己的船，慢慢的这艘船越来越大，能够抵抗更大的风浪，带着在技术的海洋里面一直前进。

说到这里，我想起大学的时候自学的《高等代数》，里面总是从问题引出几个最简单的定理，然后借助这几个最简单的定理构建起一个代数系统，我有的时候也在想技术本身也应该是如此吧，借助最简单的规则构建起庞大的计算机系统。

参考资料

[1] 对于JVM，你就只知道堆和栈吗？ (https://segmentfault.com/a/1190000019561132)

[2] Why XX:MaxDirectMemorySize can't limit Unsafe.allocateMemory? https://stackoverflow.com/questions/29702028/why-xxmaxdirectmemorysize-cant-limit-unsafe-allocatememory

[3] 指针碰撞、空闲列表、TLAB是什么关系？ https://www.zhihu.com/question/476948066/answer/2036463259

[4] 新一代垃圾回收器ZGC的探索与实践 https://tech.meituan.com/2020/08/06/new-zgc-practice-in-meituan.html

[5] 为什么Java都保留了基本数据类型，但却始终不实现struct结构体？ https://www.zhihu.com/question/521910197/answer/3173982094

[6] Java中9种常见的CMS GC问题分析与解决 https://tech.meituan.com/2020/11/12/java-9-cms-gc.html

[7] Young , Tenured and Perm generation https://stackoverflow.com/questions/2070791/young-tenured-and-perm-generation

[8] Garbage Collection Roots https://help.eclipse.org/latest/index.jsp?topic=/org.eclipse.mat.ui.help/concepts/gcroots.html

[9] JVM实战（23）——内存碎片优化 https://blog.csdn.net/smart_an/article/details/135643685

[10] JVM调优——之CMS 常见参数解析 https://www.cnblogs.com/onmyway20xx/p/6605324.html

[11] JVM 源码解读之 CMS 何时会进行 Full GC http://www.disheng.tech/blog/jvm-%E6%BA%90%E7%A0%81%E8%A7%A3%E8%AF%BB%E4%B9%8B-cms-%E4%BD%95%E6%97%B6%E4%BC%9A%E8%BF%9B%E8%A1%8C-full-gc/

[12] G1 GC：一个神奇的 JVM 参数，减少你的内存消耗 http://www.disheng.tech/blog/g1-gc%E4%B8%80%E4%B8%AA%E7%A5%9E%E5%A5%87%E7%9A%84-jvm-%E5%8F%82%E6%95%B0%E5%87%8F%E5%B0%91%E4%BD%A0%E7%9A%84%E5%86%85%E5%AD%98%E6%B6%88%E8%80%97/

[13] HotSpot Virtual Machine Garbage Collection Tuning Guide https://docs.oracle.com/en/java/javase/17/gctuning/garbage-first-g1-garbage-collector1.html#GUID-0394E76A-1A8F-425E-A0D0-B48A3DC82B42

[14] Major GC和Full GC的区别是什么？触发条件呢？ https://www.zhihu.com/question/41922036/answer/93079526

[15] java的gc为什么要分代？ https://www.zhihu.com/question/53613423/answer/135743258