

-XX:MaxMetaspaceSize=128m

正如大家所知，[JDK 8 Early Access](#)版已经提供下载。这使开发者可以体验Java8的新特性。其中之一，是Oracle从JDK7发布以来就一直[宣称](#)的要完全移除永久代空间。例如，字符串内部池，已经在JDK7中从永久代中移除。JDK8的发布将宣告它的终结。这篇文章将会分享到目前为止对 PermGen 继任者：**Metaspace**的了解。我们将通过运行一个存在类元数据对象“泄漏”的程序，来对比HotSpot1.7与HotSpot1.8（b75，译者注：翻译文章时已经到b118）的运行时行为。待Java 8正式发布之后，才会提供最终的规范，优化参数和相关文档。

元空间（Metaspace）：

一种新的内存空间的诞生

JDK8 HotSpot JVM 使用本地内存来存储类元数据信息并称之为：**元空间（Metaspace）**；这与[Oracle JRockit](#) 和[IBM JVM's](#)很相似。这将是一个好消息：意味着不会再有[java.lang.OutOfMemoryError: PermGen](#)问题，也不再需要你进行调优及监控内存空间的使用.....但请等等，这么说还为时过早。在默认情况下，这些改变是透明的，接下来我们的展示将使你仍然要关注类元数据内存的占用。请一定要牢记，这个新特性也不能神奇地消除类和类加载器导致的内存泄漏。你需求使用不同的方法以及遵守新的命名约定来追踪这些问题。我推荐大家阅读有关PermGen移除总结和[Jon对此的评论](#)。

总结如下：

PermGen 空间的状况

- 这部分内存空间将全部移除。
- JVM的参数：PermSize 和 MaxPermSize 会被忽略并给出警告（如果在启用时设置了这两个参数）。

Metaspace 内存分配模型

- 大部分类元数据都在本地内存中分配。
- 用于描述类元数据的“klasses”已经被移除。

Metaspace 容量

- 默认情况下，类元数据只受可用的本地内存限制（容量取决于32位或是64位操作系统的可用虚拟内存大小）。

- 新参数（MaxMetaspaceSize）用于限制本地内存分配给类元数据的大小。如果没有指定这个参数，元空间会在运行时根据需要动态调整。

Metaspace 垃圾回收

- 对于僵死的类及类加载器的垃圾回收将在元数据使用达到“MaxMetaspaceSize”参数的设定值时进行。
- 适时地监控和调整元空间对于减小垃圾回收频率和减少延时是很有必要的。持续的元空间垃圾回收说明，可能存在类、类加载器导致的内存泄漏或是大小设置不合适。

Java 堆内存的影响

- 一些杂项数据已经移到Java堆空间中。升级到JDK8之后，会发现Java堆空间有所增长。

Metaspace 监控

- 元空间的使用情况可以从HotSpot1.8的详细GC日志输出中得到。
- Jstat 和 JVisualVM两个工具，在我们使用b75版本进行测试时，已经更新了，但是还是能看到老的PermGen空间的出现。

前面已经从理论上充分说明，下面让我们通过“泄漏”程序进行新内存空间的观察.....

PermGen vs. Metaspace 运行时比较

为了更好地理解Metaspace内存空间的运行时行为，我们建立了一个类元数据泄漏程序。可以从此处下载[源代码](#)。

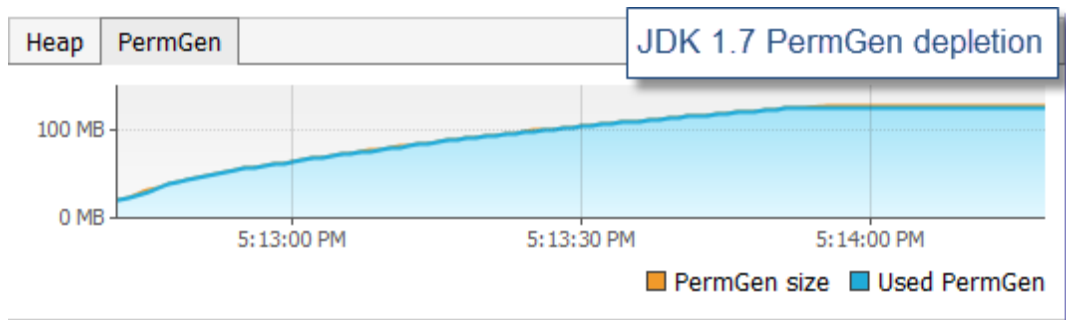
将进行以下几种场景的测试：

- 使用JDK1.7运行Java程序，监控并耗尽设定的128MB大小的PermGen内存空间。
- 使用JDK1.8 (b75)运行Java程序，监控新Metaspace内存空间的动态增长和垃圾回收过程。
- 使用JDK1.8 (b75)运行Java程序，模拟耗尽通过“MaxMetaspaceSize”参数设定的128MB大小的Metaspace内存空间。

JDK 1.7 @64-bit – PermGen 耗尽测试

- Java程序中包括5万次可配置迭代过程
- Java堆大小为1024 MB

- Java的PermGen空间为128 MB (-XX:MaxPermSize=128m)



可以从上面的JVisualVM的截图看出：当加载超过3万个类之后，PermGen被耗尽。我们也能通过程序和GC的输出观察耗尽的过程。

[html] [view plaincopy](#)

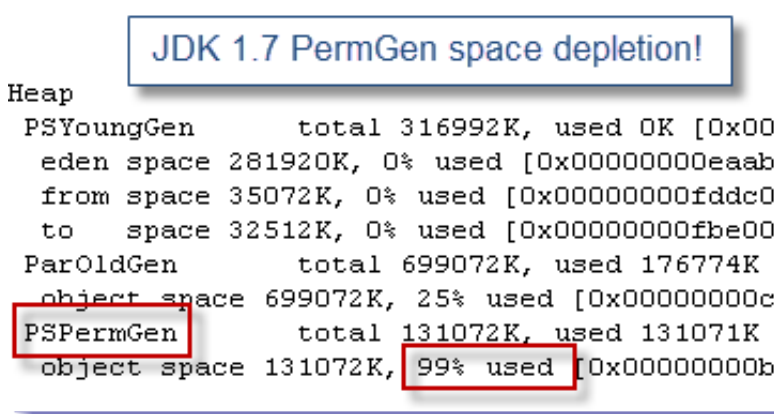


在CODE上查看代码片



派生到我的代码片

1. Class metadata leak simulator
2. Author: Pierre-Hugues Charbonneau
3. <http://javaeasupportpatterns.blogspot.com>
4. ERROR: java.lang.OutOfMemoryError: PermGen space

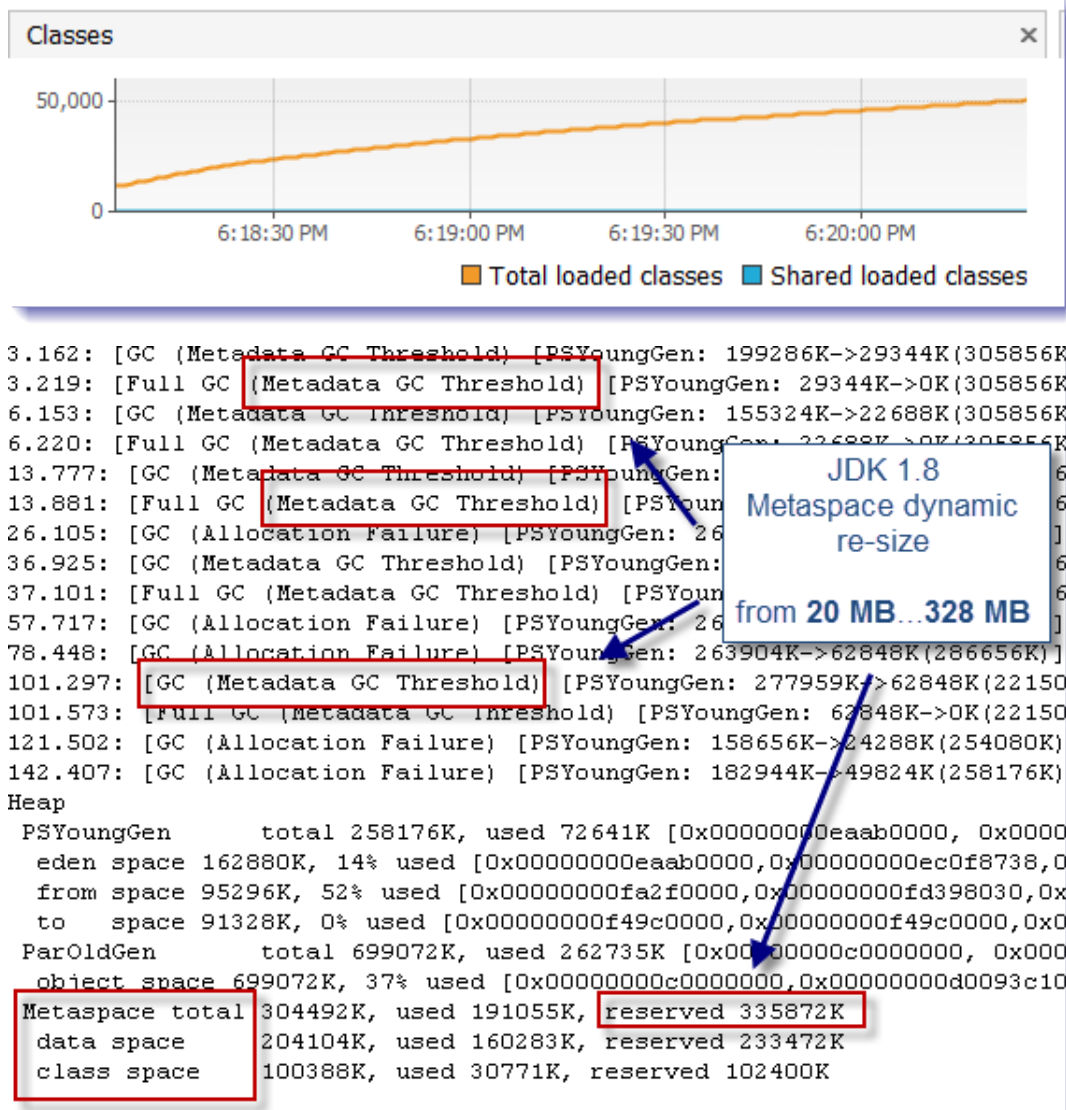


下面我们使用HotSpot JDK 1.8 JRE来执行程序。

JDK 1.8 @64-bit – Metaspace大小动态调整测试

- Java程序中包括5万次可配置迭代过程
- Java堆大小为1024 MB

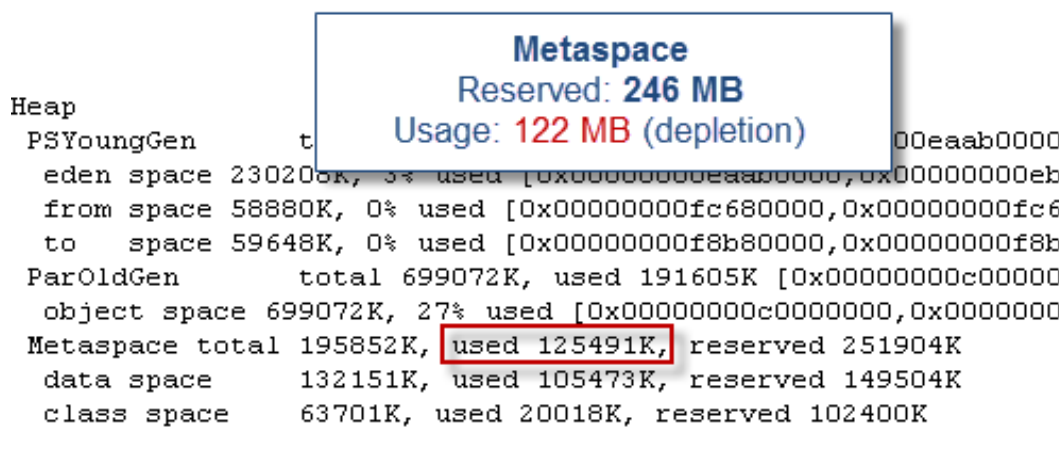
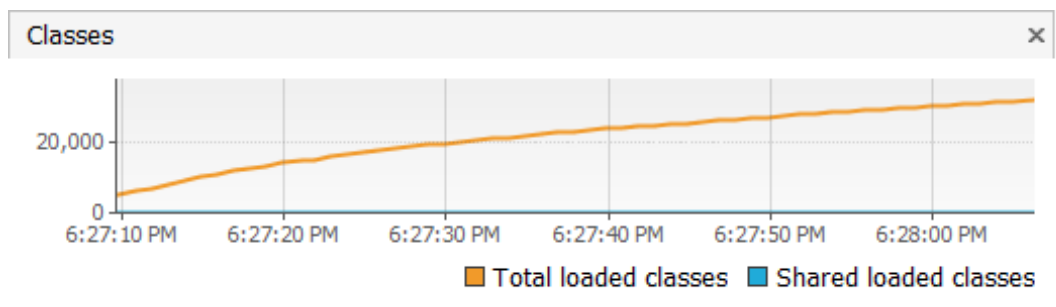
- Java的Metaspace空间：不受限制（默认）



从上面的截图可以看到详细的GC输出日志，JVM Metaspace进行了动态扩展，本地内存的使用由20MB增长到328MB，以满足程序中不断增长的类数据内存占用需求。我们也能观察到JVM的垃圾回收事件—试图销毁僵死的类或类加载器对象。但是，由于我们程序的泄漏，JVM别无选择只能动态扩展Metaspace内存空间。程序能够运行5万次迭代，加载超过5万个类，而没有出现OOM事件。下面继续进行最后的一个测试场景：

JDK 1.8 @64-bit – Metaspace depletion

- Java程序中包括5万次可配置迭代过程
- Java堆大小为1024 MB
- Java的Metaspace空间：128MB (-XX:MaxMetaspaceSize=128m)



可以从上面的JVisualVM的截图看出：当加载超过3万个类之后，Metaspace被耗尽；与JDK1.7运行时非常相似。我们也能通过程序和GC的输出观察耗尽的过程。另一个有趣的现象是，保留的原生内存占用量是设定的最大大小两倍之多。这可能表明，如果可能的话，可微调元空间容量大小策略，来避免本地内存的浪费。

从Java程序的输出中看到如下异常。

[\[html\] view plaincopy](#)



在CODE上查看代码片



派生到我的代码片

1. Class metadata leak simulator
2. Author: Pierre-Hugues Charbonneau
3. <http://javaeesupportpatterns.blogspot.com>
4. ERROR: java.lang.OutOfMemoryError: Metadata space

完成！

正如预期的那样，像运行JDK1.7基线时一样，限定128 MB大小元空间时，并不能让程序完成50万次迭代。一种新的OOM错误被JVM抛出。上述OOM事件，是由

于元空间内存分配失败由JVM抛出的。

#metaspace.cpp

```
if (result == NULL) {
    // Try to clean out some memory and retry.
    result =
        Universe::heap()->collector_policy()->satisfy_failed_metadata_alloc
            loader_data, word_size, mdtype);

    // If result is still null, we are out of memory.
    if (result == NULL) {
        if (Verbose && TraceMetadataChunkAllocation) {
            gclog_or_tty->print_cr("Metaspace allocation failed for size "
                SIZE_FORMAT, word_size);
            if (loader_data->metaspace_or_null() != NULL) loader_data->metasp
                MetaspaceAux::dump(gclog_or_tty);
        }
        // -XX:+HeapDumpOnOutOfMemoryError and -XX:OnOutOfMemoryError suppo
        report_java_out_of_memory("Metadata space");
    }

    if (JvmtiExport::should_post_resource_exhausted()) {
        JvmtiExport::post_resource_exhausted(
            JVMTI_RESOURCE_EXHAUSTED_OOM_ERROR,
            "Metadata space");
    }
}
```

Metaspace
Depletion

一，什么是垃圾回收

首先，在了解G1之前，我们需要清楚的知道，垃圾回收是什么？简单的说垃圾回收就是回收内存中不再使用的对象。

垃圾回收的基本步骤

回收的步骤有2步：

1. 查找内存中不再使用的对象
2. 释放这些对象占用的内存

1,查找内存中不再使用的对象

那么问题来了，如何判断哪些对象不再被使用呢？我们也有2个方法：

1. 引用计数法

引用计数法就是如果一个对象没有被任何引用指向，则可视之为垃圾。这种方法的缺点就是不能检测到环的存在。

2.根搜索算法

根搜索算法的基本思路就是通过一系列名为“GC Roots”的对象作为起始点，从这些节点开始向下搜索，搜索所走过的路径称为引用链(Reference Chain)，当一个对象到GC Roots没有任何引用链相连时，则证明此对象是不可用的。

现在我们已经知道如何找出垃圾对象了，如何把这些对象清理掉呢？

2. 释放这些对象占用的内存

常见的方式有复制或者直接清理，但是直接清理会存在内存碎片，于是就会产生清理再压缩的方式。

总得来说就产生了三种类型的回收算法。

1.标记-复制

它将可用内存容量划分为大小相等的两块，每次只使用其中的一块。当这一块用完后，就将还存活的对象复制到另外一块上面，然后在把已使用过的内存空间一次理掉。它的优点是实现简单，效率高，不会存在内存碎片。缺点就是需要2倍的内存来管理。

2.标记-清理

标记清除算法分为“标记”和“清除”两个阶段：首先标记出需要回收的对象，标记完成之后统一清除对象。它的优点是效率高，缺点是容易产生内存碎片。

3.标记-整理

标记操作和“标记-清理”算法一致，后续操作不只是直接清理对象，而是在清理无用对象完成后让所有 存活的对象都向一端移动，并更新引用其对象的指针。因为要移动对象，所以它的效率要比“标记-清理”效率低，但是不会产生内存碎片。

基于分代的假设

由于对象的存活时间有长有短，所以对于存活时间长的对象，减少被gc的次数可以避免不必要的开销。这样我们就把内存分成新生代和老年代，新生代存放刚创建的和存活时间比较短的对象，老年代存放存活时间比较长的对象。这样每次仅仅清理年轻代，老年代仅在必要时再做清理可以极大的提高GC效率，节省GC时间。

java垃圾收集器的历史

第一阶段，Serial（串行）收集器

在jdk1.3.1之前，java虚拟机仅仅能使用Serial收集器。Serial收集器是一个单线程的收集器，但它的“单线程”的意义并不仅仅是说明它只会使用一个CPU或一条收集线程去完成垃圾收集工作，更重要的是在它进行垃圾收集时，必须暂停其他所有的工作线程，直到它收集结束。

PS：开启Serial收集器的方式

-XX:+UseSerialGC

第二阶段，Parallel（并行）收集器

Parallel收集器也称吞吐量收集器，相比Serial收集器，Parallel最主要的优势在于使用多线程去完成垃圾清理工作，这样可以充分利用多核的特性，大幅降低gc时间。

PS:开启Parallel收集器的方式

-XX:+UseParallelGC -XX:+UseParallelOldGC

第三阶段，CMS（并发）收集器

CMS收集器在Minor GC时会暂停所有的应用线程，并以多线程的方式进行垃圾回收。在Full GC时不再暂停应用线程，而是使用若干个后台线程定期的对老年代空间进行扫描，及时回收其中不再使用的对象。

PS:开启CMS收集器的方式

-XX:+UseParNewGC -XX:+UseConcMarkSweepGC

第四阶段，G1（并发）收集器

G1收集器（或者垃圾优先收集器）的设计初衷是为了尽量缩短处理超大堆（大于4GB）时产生的停顿。相对于CMS的优势而言是内存碎片的产生率大大降低。

PS:开启G1收集器的方式

-XX:+UseG1GC

二，了解G1

G1的第一篇paper（附录1）发表于2004年，在2012年才在jdk1.7u4中可用。oracle

官方计划在jdk9中将G1变成默认的垃圾收集器，以替代CMS。为何oracle要极力推荐G1呢，G1有哪些优点？

首先，G1的设计原则就是简单可行的性能调优

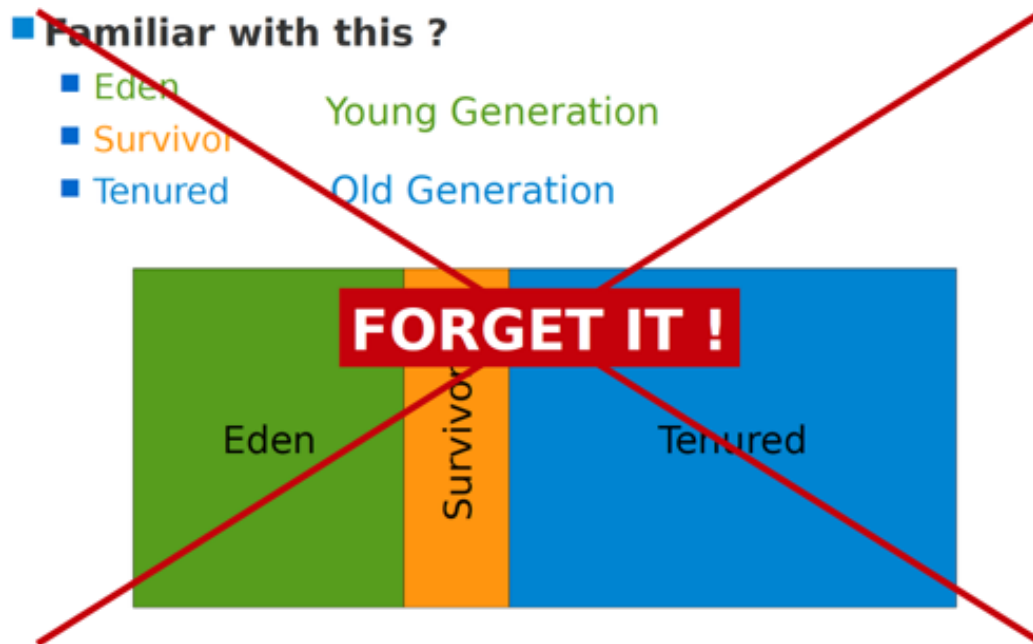
开发人员仅仅需要声明以下参数即可：

`-XX:+UseG1GC -Xmx32g -XX:MaxGCPauseMillis=200`

其中`-XX:+UseG1GC`为开启G1垃圾收集器，`-Xmx32g`设计堆内存的最大内存为32G，`-XX:MaxGCPauseMillis=200`设置GC的最大暂停时间为200ms。如果我们需要调优，在内存大小一定的情况下，我们只需要修改最大暂停时间即可。

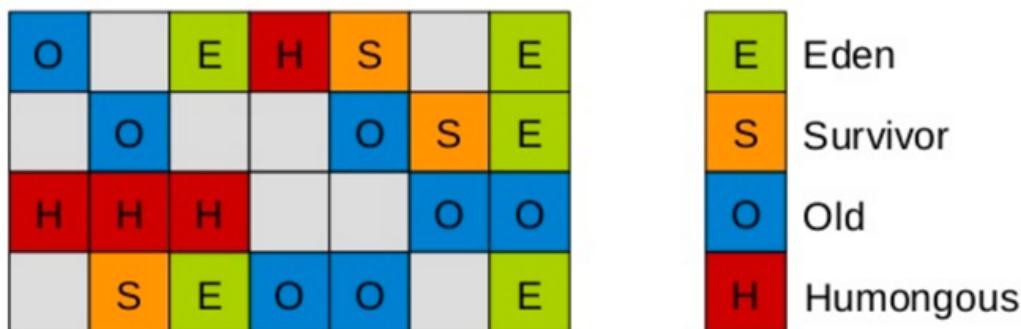
其次，G1将新生代，老年代的物理空间划分取消了。

这样我们再也不用单独的空间对每个代进行设置了，不用担心每个代内存是否足够。



20161222153407_691.png

取而代之的是，G1算法将堆划分为若干个区域（Region），它仍然属于分代收集器。不过，这些区域的一部分包含新生代，新生代的垃圾收集依然采用暂停所有应用线程的方式，将存活对象拷贝到老年代或者Survivor空间。老年代也分成很多区域，G1收集器通过将对象从一个区域复制到另外一个区域，完成了清理工作。这就意味着，在正常的处理过程中，G1完成了堆的压缩（至少是部分堆的压缩），这样也就不会有cms内存碎片问题的存在了。



20161222153407_471.png

在G1中，还有一种特殊的区域，叫Humongous区域。如果一个对象占用的空间超过了分区容量50%以上，G1收集器就认为这是一个巨型对象。这些巨型对象，默认直接会被分配在年老代，但是如果它是一个短期存在的巨型对象，就会对垃圾收集器造成负面影响。为了解决这个问题，G1划分了一个Humongous区，它用来专门存放巨型对象。如果一个H区装不下一个巨型对象，那么G1会寻找连续的H分区来存储。为了能找到连续的H区，有时候不得不启动Full GC。

PS：在java 8中，持久代也移动到了普通的堆内存空间中，改为元空间。

对象分配策略

说起大对象的分配，我们不得不谈谈对象的分配策略。它分为3个阶段：

1. TLAB(Thread Local Allocation Buffer)线程本地分配缓冲区
2. Eden区中分配
3. Humongous区分配

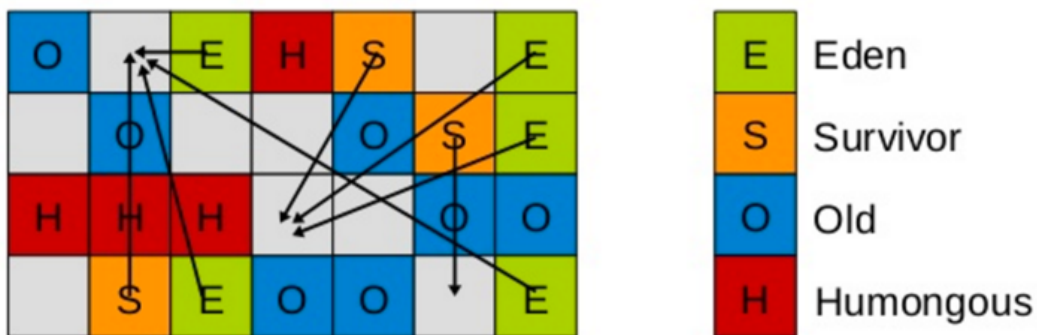
TLAB为线程本地分配缓冲区，它的目的为了使对象尽可能快的分配出来。如果对象在一个共享的空间中分配，我们需要采用一些同步机制来管理这些空间内的空闲空间指针。在Eden空间中，每一个线程都有一个固定的分区用于分配对象，即一个TLAB。分配对象时，线程之间不再需要进行任何的同步。

对TLAB空间中无法分配的对象，JVM会尝试在Eden空间中进行分配。如果Eden空间无法容纳该对象，就只能在老年代中进行分配空间。

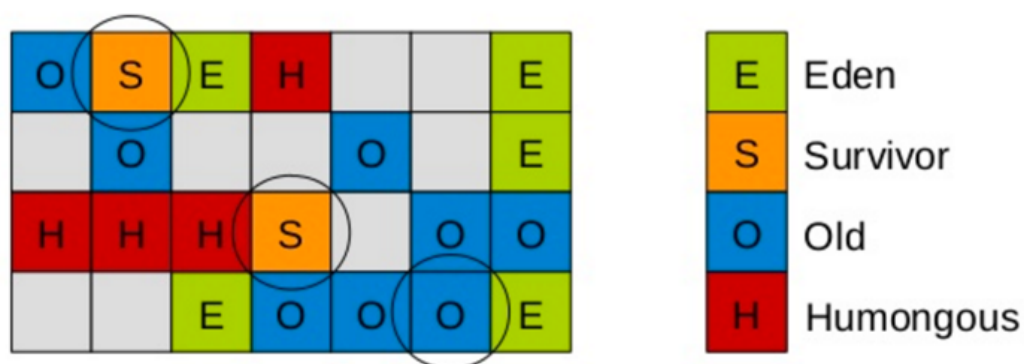
最后，G1提供了两种GC模式，Young GC和Mixed GC，两种都是Stop The World(STW)的。下面我们将分别介绍一下这2种模式。

三，G1 Young GC

Young GC主要是对Eden区进行GC，它在Eden空间耗尽时会被触发。在这种情况下，Eden空间的数据移动到Survivor空间中，如果Survivor空间不够，Eden空间的部分数据会直接晋升到老年代空间。Survivor区的数据移动到新的Survivor区中，也有部分数据晋升到老年代空间中。最终Eden空间的数据为空，GC停止工作，应用线程继续执行。

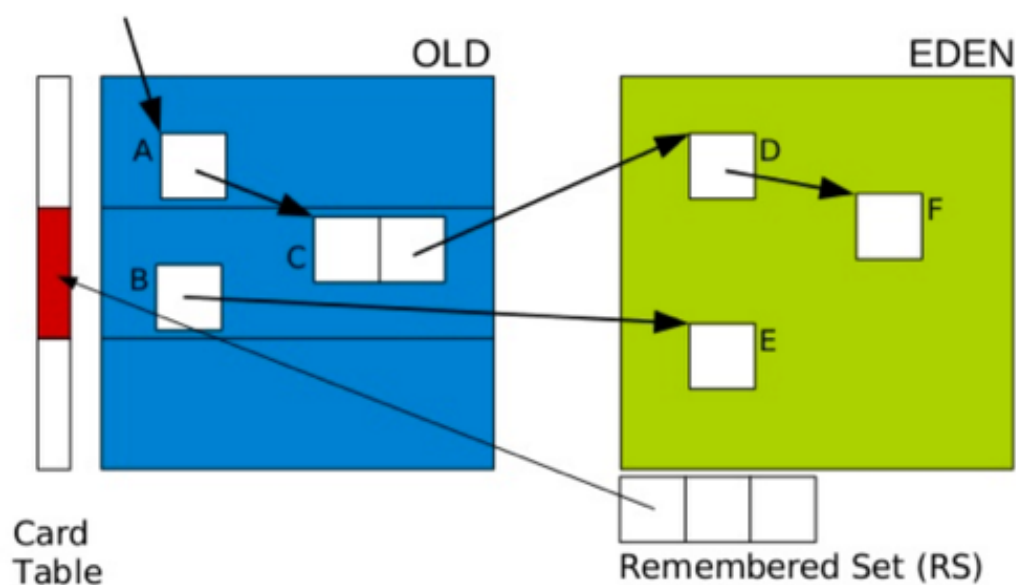


20161222153407_47.png



20161222153408_747.png

这时，我们需要考虑一个问题，如果仅仅GC 新生代对象，我们如何找到所有的根对象呢？老年代的所有对象都是根么？那这样扫描下来会耗费大量的时间。于是，G1引入了RSet的概念。它的全称是Remembered Set，作用是跟踪指向某个heap区内的对象引用。



在CMS中，也有RSet的概念，在老年代中有一块区域用来记录指向新生代的引用。这是一种point-out，在进行Young GC时，扫描根时，仅仅需要扫描这一块区域，而不需要扫描整个老年代。

但在G1中，并没有使用point-out，这是由于一个分区太小，分区数量太多，如果是用point-out的话，会造成大量的扫描浪费，有些根本不需要GC的分区引用也扫描了。于是G1中使用point-in来解决。point-in的意思是哪些分区引用了当前分区中的对象。这样，仅仅将这些对象当做根来扫描就避免了无效的扫描。由于新生代有多个，那么我们需要在新生代之间记录引用吗？这是不必要的，原因在于每次GC时，所有新生代都会被扫描，所以只需要记录老年代到新生代之间的引用即可。

需要注意的是，如果引用的对象很多，赋值器需要对每个引用做处理，赋值器开销会很大，为了解决赋值器开销这个问题，在G1 中又引入了另外一个概念，卡表（Card Table）。一个Card Table将一个分区在逻辑上划分为固定大小的连续区域，每个区域称之为卡。卡通常较小，介于128到512字节之间。Card Table通常为字节数组，由Card的索引（即数组下标）来标识每个分区的空间地址。默认情况下，每个卡都未被引用。当一个地址空间被引用时，这个地址空间对应的数组索引的值被标记为“0”，即标记为脏被引用，此外RSet也将这个数组下标记录下来。一般情况下，这个RSet其实是一个Hash Table，Key是别的Region的起始地址，Value是一个集合，里面的元素是Card Table的Index。

Young GC 阶段：

- 阶段1：根扫描

静态和本地对象被扫描

- 阶段2：更新RS

处理dirty card队列更新RS

- 阶段3：处理RS

检测从年轻代指向年老代的对象

- 阶段4：对象拷贝

拷贝存活的对象到survivor/old区域

- 阶段5：处理引用队列

软引用，弱引用，虚引用处理

四，G1 Mix GC

Mix GC不仅进行正常的新生代垃圾收集，同时也回收部分后台扫描线程标记的老年代分区。

它的GC步骤分2步：

1. 全局并发标记（global concurrent marking）
2. 拷贝存活对象（evacuation）

在进行Mix GC之前，会先进行global concurrent marking（全局并发标记）。

global concurrent marking的执行过程是怎样的呢？

在G1 GC中，它主要是为Mixed GC提供标记服务的，并不是一次GC过程的一个必须环节。global concurrent marking的执行过程分为五个步骤：

- 初始标记 (initial mark, STW)

在此阶段, G1 GC 对根进行标记。该阶段与常规的 (STW) 年轻代垃圾回收密切相关。

- 根区域扫描 (root region scan)

G1 GC 在初始标记的存活区扫描对老年代的引用, 并标记被引用的对象。该阶段与应用程序 (非 STW) 同时运行, 并且只有完成该阶段后, 才能开始下一次 STW 年轻代垃圾回收。

- 并发标记 (Concurrent Marking)

G1 GC 在整个堆中查找可访问的 (存活的) 对象。该阶段与应用程序同时运行, 可以被 STW 年轻代垃圾回收中断

- 最终标记 (Remark, STW)

该阶段是 STW 回收, 帮助完成标记周期。G1 GC 清空 SATB 缓冲区, 跟踪未被访问的存活对象, 并执行引用处理。

- 清除垃圾 (Cleanup, STW)

在这个最后阶段, G1 GC 执行统计和 RSet 净化的 STW 操作。在统计期间, G1 GC 会识别完全空闲的区域和可供进行混合垃圾回收的区域。清理阶段在将空白区域重置并返回到空闲列表时为部分并发。

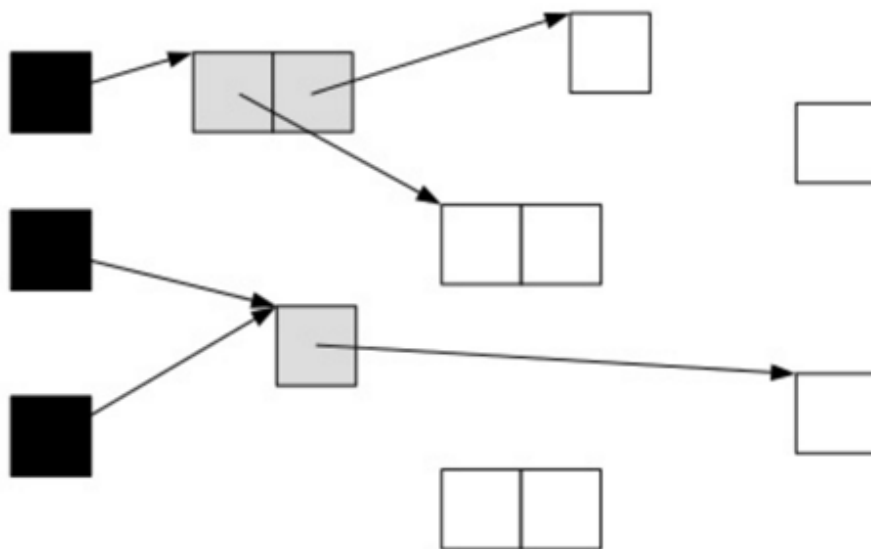
三色标记算法

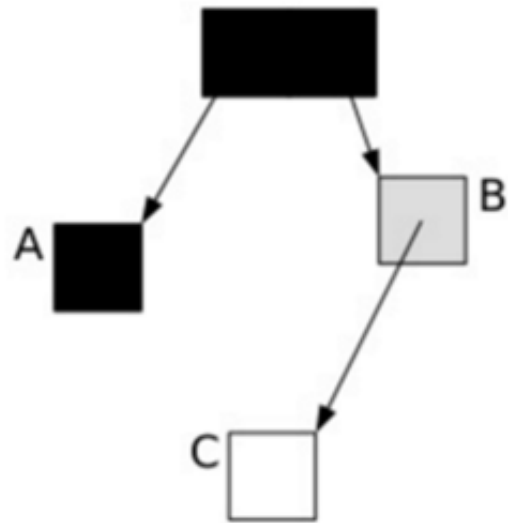
提到并发标记, 我们不得不了解并发标记的三色标记算法。它是描述追踪式回收器的一种有用的方法, 利用它可以推演回收器的正确性。首先, 我们将对象分成三种类型的。

- 黑色: 根对象, 或者该对象与它的子对象都被扫描
- 灰色: 对象本身被扫描, 但还没扫描完该对象中的子对象
- 白色: 未被扫描对象, 扫描完成所有对象之后, 最终为白色的为不可达对象, 即垃圾对象

当GC开始扫描对象时, 按照如下图步骤进行对象的扫描:

根对象被置为黑色, 子对象被置为灰色。





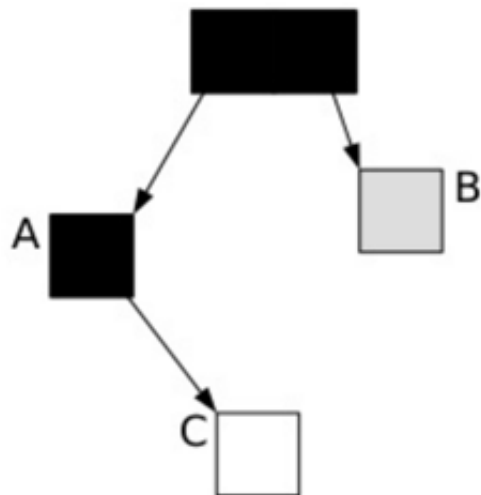
20161222153408_866.png

这时候应用程序执行了以下操作：

A.c=C

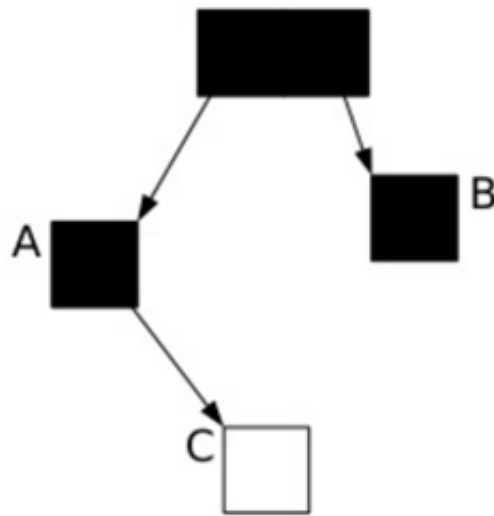
B.c=null

这样，对象的状态图变成如下情形：



20161222153408_118.png

这时候垃圾收集器再标记扫描的时候就会下图成这样：



20161222153408_812.png

很显然，此时C是白色，被认为是垃圾需要清理掉，显然这是不合理的。那么我们如何保证应用程序在运行的时候，GC标记的对象不丢失呢？有如下2中可行的方式：

1. 在插入的时候记录对象
2. 在删除的时候记录对象

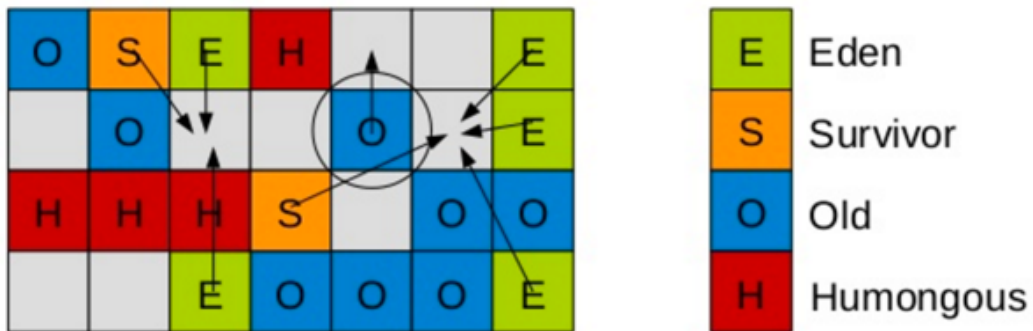
刚好这对应CMS和G1的2种不同实现方式：

在CMS采用的是增量更新（Incremental update），只要在写屏障（write barrier）里发现要有一个白对象的引用被赋值到一个黑对象的字段里，那就把这个白对象变成灰色的。即插入的时候记录下来。

在G1中，使用的是STAB（snapshot-at-the-beginning）的方式，删除的时候记录所有的对象，它有3个步骤：

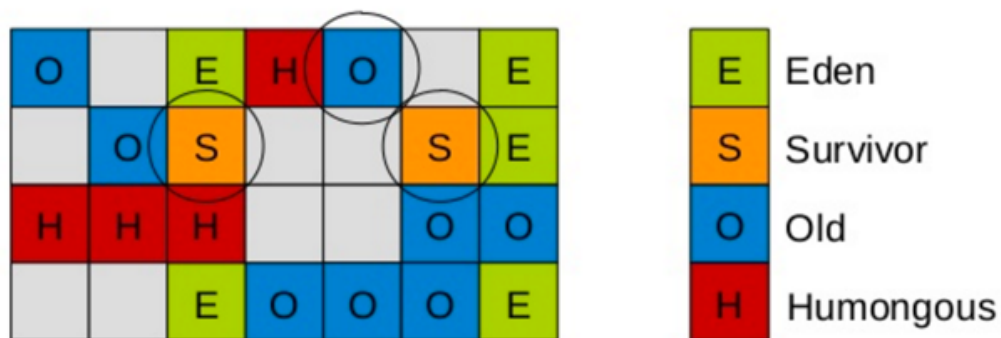
- 1，在开始标记的时候生成一个快照图标记存活对象
- 2，在并发标记的时候所有被改变的对象入队（在write barrier里把所有旧的引用所指向的对象都变成非白的）
- 3，可能存在游离的垃圾，将在下次被收集

这样，G1到现在可以知道哪些老的分区可回收垃圾最多。当全局并发标记完成后，在某个时刻，就开始了Mix GC。这些垃圾回收被称作“混合式”是因为他们不仅仅进行正常的新生代垃圾收集，同时也回收部分后台扫描线程标记的分区。混合式垃圾收集如下图：



20161222153408_75.png

混合式GC也是采用的复制的清理策略，当GC完成后，会重新释放空间。



20161222153408_443.png

至此，混合式GC告一段落了。下一小节我们讲进入调优实践。

五，调优实践

MaxGCPauseMillis调优

前面介绍过使用GC的最基本的参数：

-XX:+UseG1GC -Xmx32g -XX:MaxGCPauseMillis=200

前面2个参数都好理解，后面这个MaxGCPauseMillis参数该怎么配置呢？这个参数从字面的意思上看，就是允许的GC最大的暂停时间。G1尽量确保每次GC暂停的时间都在设置的MaxGCPauseMillis范围内。那G1是如何做到最大暂停时间的呢？这涉及到另一个概念，CSet(collection set)。它的意思是在一次垃圾收集器中被收集的区域集合。

- Young GC：选定所有新生代里的region。通过控制新生代的region个数来控制young GC的开销。
- Mixed GC：选定所有新生代里的region，外加根据global concurrent marking统计

得出收集收益高的若干老年代region。在用户指定的开销目标范围内尽可能选择收益高的老年代region。

在理解了这些后，我们再设置最大暂停时间就好办了。首先，我们能容忍的最大暂停时间是有一个限度的，我们需要在这个限度范围内设置。但是应该设置的值是多少呢？我们需要在吞吐量跟MaxGCPauseMillis之间做一个平衡。如果MaxGCPauseMillis设置的过小，那么GC就会频繁，吞吐量就会下降。如果MaxGCPauseMillis设置的过大，应用程序暂停时间就会变长。G1的默认暂停时间是200毫秒，我们可以从这里入手，调整合适的时间。

其他调优参数

-XX:G1HeapRegionSize=n

设置的 G1 区域的大小。值是 2 的幂，范围是 1 MB 到 32 MB 之间。目标是根据最小的 Java 堆大小划分出约 2048 个区域。

-XX:ParallelGCThreads=n

设置 STW 工作线程数的值。将 n 的值设置为逻辑处理器的数量。n 的值与逻辑处理器的数量相同，最多为 8。

如果逻辑处理器不止八个，则将 n 的值设置为逻辑处理器数的 5/8 左右。这适用于大多数情况，除非是较大的 SPARC 系统，其中 n 的值可以是逻辑处理器数的 5/16 左右。

-XX:ConcGCThreads=n

设置并行标记的线程数。将 n 设置为并行垃圾回收线程数 (ParallelGCThreads) 的 1/4 左右。

-XX:InitiatingHeapOccupancyPercent=45

设置触发标记周期的 Java 堆占用率阈值。默认占用率是整个 Java 堆的 45%。

避免使用以下参数：

避免使用 -Xmn 选项或 -XX:NewRatio 等其他相关选项显式设置年轻代大小。固定年轻代的大小会覆盖暂停时间目标。

触发Full GC

在某些情况下，G1触发了Full GC，这时G1会退化使用Serial收集器来完成垃圾的清理工作，它仅仅使用单线程来完成GC工作，GC暂停时间将达到秒级别的。整个应用处于假死状态，不能处理任何请求，我们的程序当然不希望看到这些。那么发生Full GC的情况有哪些呢？

- 并发模式失败

G1启动标记周期，但在Mix GC之前，老年代就被填满，这时候G1会放弃标记周期。这种情形下，需要增加堆大小，或者调整周期（例如增加线程数-XX:ConcGCThreads等）。

- 晋升失败或者疏散失败

G1在进行GC的时候没有足够的内存供存活对象或晋升对象使用，由此触发了Full GC。可以在日志中看到(to-space exhausted)或者 (to-space overflow)。解决这种问题的方式是：

a,增加 -XX:G1ReservePercent 选项的值（并相应增加总的堆大小），为“目标空间”增加预留内存量。

b,通过减少 -XX:InitiatingHeapOccupancyPercent 提前启动标记周期。

c,也可以通过增加 `-XX:ConcGCThreads` 选项的值来增加并行标记线程的数目。

- 巨型对象分配失败

当巨型对象找不到合适的空间进行分配时,就会启动Full GC,来释放空间。这种情况下,应该避免分配大量的巨型对象,增加内存或者增大`-XX:G1HeapRegionSize`,使巨型对象不再是巨型对象。

由于篇幅有限,G1还有很多调优实践,在此就不一一列出了,大家在平常的实践中可以慢慢探索。最后,期待java 9能正式发布,默认使用G1为垃圾收集器的java性能会不会又提高呢?

结束语

我希望您能喜欢这篇较早的对Java8元空间的分析和实验文章。目前的观察有力地表明,适当的监控和调优是必须的,以此来避免诸如,过度的在元空间中的GC,或像我们测试场景下触发OOM的条件。后面的文章中可能包括,性能上的比较,以确定这一新特性是否有潜在的性能上的提升。

参考: [Java 8: From PermGen to Metaspace](#) from our [JCG partner](#) Pierre-Hugues Charbonneau at the [Java EE Support Patterns & Java Tutorial](#) blog.