How actually card table and writer barrier work?

Card Table 和 Writer barrier 实际上是如何运作的?

Asked 11 years, 3 months ago Modified 2 years, 11 months ago Viewed 13k times

36

I am reading some materials about garbage collection in Java in order to get to know more deeply what really happens in GC process.

我正在阅读一些关于 Java 垃圾收集的材料，以便更深入地了解 GC 流程中真正发生的事情。

I came across on the mechanism called "card table". I've Googled for it and haven't found comprehensive information. Most of explanations are rather shallow and describes it like some magic.

我遇到了一种叫做 “card table” 的机制。我已经用谷歌搜索了它，但没有找到全面的信息。大多数解释都相当肤浅，像某种魔术一样描述它。

My question is: How card table and write barrier works? What is marked in card tables? How then garbage collector knows that particular object is referenced by another object persisted in older generation.

我的问题是：card table 和 write barrier 是如何工作的？牌表中标记了什么？那么垃圾回收器如何知道特定对象被老一代中保留的另一个对象引用。

I would like to have some practical imagination about that mechanism, like I was supposed to prepare some simulation.

我想对那个机制有一些实际的想象，就像我应该准备一些模拟一样。

java garbage-collection

Share Improve this question Follow

edited Jan 14, 2022 at 12:18

asked Oct 3, 2013 at 8:41

Dawid

803 ● 2 ● 9 ● 18

2 To get into such a deep theory, you can try this book gchandbook.org/contents.html. It provides quiet comprehensive overview.

要深入了解如此深入的理论，您可以尝试 gchandbook.org/contents.html 这本书。它提供安静的全面概述。 – Mikhail Oct 3, 2013 at 8:55

3 Answers

Sorted by: Highest score (default)

48

I don't know whether you found some exceptionally bad description or whether you expect too many details, I've been quite satisfied with the explanations I've seen. If the descriptions are brief and sound simplistic, that's because it really is a rather simple mechanism.

我不知道你是发现了一些特别糟糕的描述，还是你期望太多的细节，我对我看到的解释相当满意。如果描述简短且听起来简单，那是因为它确实是一个相当简单的机制。

As you apparently already know, a generational garbage collector needs to be able to enumerate old objects that refer to young objects. It would be correct to scan all old objects, but that destroys the advantages of the generational approach, so you have to narrow it down. Regardless of how you do that, you need a write barrier - a piece of code executed whenever a member variable (of a reference type) is assigned/written to. If the new reference points to a young object and it's stored in an old object, the write barrier records that fact for the garbage collect. The difference lies in how it's recorded. There are exact schemes using so-called remembered sets, a collection of *every* old object that has (had at some point) a reference to a young object. As you can imagine, this takes quite a bit of space.

正如您显然已经知道的那样，分代垃圾回收器需要能够枚举引用年轻对象的旧对象。扫描所有旧对象是正确的，但这会破坏分代方法的优势，因此您必须缩小范围。无论你怎么做，你都需要一个写屏障 - 每当分配/写入成员变量（引用类型）时执行的一段代码。如果新引用指向一个年轻的对象，并且它存储在一个旧对象中，则写屏障会记录该事实以进行垃圾回收。区别在于它的录制方式。有一些使用所谓的 remembered sets 的精确方案，即具有（在某个时候）引用年轻对象的*每个*旧对象的集合。可以想象，这需要相当大的空间。

The card table is a trade-off: Instead of telling you which objects *exactly* contains young pointers (or at least did at some point), it groups objects into fixed-sized buckets and tracks which buckets contain objects with young pointers. This, of course, reduces space usage. For correctness, it doesn't really matter how you bucket the objects, as long as you're consistent about it. For efficiency, you just group them by their memory address (because you have that available for free), divided by some larger power of two (to make the division a cheap

bitwise operation).

card 表是一种权衡：它不是告诉您哪些对象*确切*地包含年轻指针（或者至少在某个时候包含），而是将对象分组到固定大小的存储桶中，并跟踪哪些存储桶包含具有年轻指针的对象。当然，这减少了空间使用。为了正确起见，如何对对象进行存储并不重要，只要您始终如一即可。为了提高效率，您只需按它们的内存地址（因为您免费提供）对它们进行分组，然后除以 2 的较大幂数（使除法成为一种廉价的按位运算）。

Also, instead of maintaining an explicit list of buckets, you reserve some space for each possible bucket up-front. Specifically, there is an array of N bits or bytes, where N is the number of buckets, so that the `i`th value is 0 if the `i`th bucket contains no young pointers, or 1 if it does contain young pointers. This is the card table proper. Typically this space is allocated and freed along with a large block of memory used as (part of) the heap. It may even be embedded in the start of the memory block, if it doesn't need to grow. Unless the entire address space is used as heap (which is very rare), the above formula gives numbers starting from `start_of_memory_region >> K` instead of 0, so to get an index into the card table you have to subtract the start of the start address of the heap.

此外，您无需维护明确的存储桶列表，而是预先为每个可能的存储桶预留一些空间。具体来说，有一个包含 N 位或字节的数组，其中 N 是桶的数量，因此如果第 `i`个桶不包含年轻指针，则第 i 个值为 0，如果它包含年轻指针，则第 `i`个值为 1。这是正确的牌桌。通常，此空间与用作堆（一部分）的大块内存一起分配和释放。如果它不需要增长，它甚至可以嵌入到内存块的开头。除非整个地址空间都用作堆（这种情况非常罕见），否则上面的公式给出从 `>> K 开始start_of_memory_region`数字而不是 0，因此要将索引放入 card 表中，您必须减去堆开始地址的开头。

In summary, when the write barrier finds that the statement `some_obj.field = other_obj;` stores a young pointer in an old object, it does this:

总之，当写入屏障发现语句 `some_obj.field = other_obj;` 将年轻指针存储在旧对象中时，它会执行以下操作：

```
card_table[(&old_obj - start_of_heap) >> K] = 1;
```

Where `&old_obj` is the address of the object that now has a young pointer (which is already in a register because it was just determined to refer to an old object). During minor GC, the garbage collector looks at the card table to determine which heap regions to scan for young pointers:

其中 `&old_obj` 是现在有一个年轻指针的对象的地址（它已经在 register 中，因为它刚刚被确定为引用一个旧对象）。在次要 GC 期间，垃圾回收器会查看 card 表以确定要扫描哪些堆区域以查找年轻的指针：

```
for i from 0 to (heap_size >> K):
    if card_table[i]:
        scan heap[i << K .. (i + 1) << K] for young pointers
```

Share Improve this answer Follow

edited Oct 26, 2014 at 11:55

answered Oct 3, 2013 at 9:22

user395760

1 If the card table indicates that some references held in some chunk of gen1 storage have been updated, how would the system identify which parts of that chunk held references? Does the system need to keep and traverse a linked list of all gen1 objects to find the starting addresses of any objects that are at least partially within the touched block?

如果 card 表指示 gen1 存储的某个块中保存的某些引用已更新，系统如何识别该块的哪些部分保存了引用？系统是否需要保留并遍历所有 gen1 对象的链接列表，以查找至少部分位于触摸块内的任何对象的起始地址？

– supercat Sep 20, 2018 at 19:13

Serious question: Now after the GC runs, the card table is reset. What happens if the old object did not update a reference to the young object, and then the next GC cycle kicks in??? will the young object be wiped, despite that the old object might still be referencing or needing it???

严重的问题：现在 GC 运行后，卡表被重置。如果旧对象没有更新对年轻对象的引用，然后下一个 GC 周期开始，会发生什么情况???尽管旧对象可能仍在引用或需要它，但 Young 对象是否会被擦除???

– Ahmad Tn Oct 17, 2019 at 14:58

@AhmadTn: why would the card table be reset when the GC runs? For a young collection I'd expect the card table to be kept intact.

@AhmadTn：为什么 GC 运行时 card 表会被重置？对于年轻的收藏，我希望牌桌保持完整。 – devoured elysium Nov 23, 2019 at 3:46

am i correct about this situation? if there is no pointer to young object in the old object (previously it has but now it doesnt anymore). The card table entry for it be reseted in next young collection because the GC couldnt find any

我对这种情况的看法正确吗？如果旧对象中没有指向 young 对象的指针（以前有，但现在不再有）。它的 card 表条目将在 next young collection 中重置，因为 GC 找不到任何

– Foxie Flakey Jun 24, 2022 at 13:23

Sometime ago I have written an article explaining mechanics of young collection in HotSpot JVM. Understanding GC pauses in JVM, HotSpot's minor GC

20

前段时间，我写了一篇文章，解释了 HotSpot JVM 中 young collection 的机制。 [了解 JVM 中的 GC 暂停，HotSpot 的次要 GC]

> Principle of dirty card write-barrier is very simple. Each time when program modifies reference in memory, it should mark modified memory page as dirty. There is a special card table in JVM and each 512 byte page of memory has associated one byte entry in card table.
>
> 脏卡写屏障的原理非常简单。每次程序修改内存中的引用时，它都应该将修改后的内存页标记为脏。JVM 中有一个特殊的 card 表，每个 512 字节的内存页在 card 表中都有一个关联的字节条目。
>
> Normally collection of all references from old space to young would require scanning through all objects in old space. That is why we need write-barrier. All objects in young space have been created (or relocated) since last reset of write-barrier, so non-dirty pages cannot have references into young space. This means we can scan only object in dirty pages.
>
> 通常，从 old space 到 young 的所有引用的集合需要扫描 old space 中的所有对象。这就是为什么我们需要 write-barrier。自上次重置 write-barrier 以来，young space 中的所有对象都已被创建（或重新定位），因此非脏页面不能引用 young space。这意味着我们只能扫描脏页中的对象。

Share Improve this answer Follow

edited Jun 20, 2020 at 9:12
Community Bot
1 ● 1

answered Oct 4, 2013 at 4:25
Alexey Ragozin
8,379 ● 1 ● 25 ● 23

16

For anyone who is looking for a simple answer:

对于正在寻找简单答案的任何人：

In JVM, the memory space of objects is broken down into two spaces:

在 JVM 中，对象的内存空间分为两个空间：

- Young generation (space): All new allocations (objects) are created inside this space.

  Young generation （space）： 所有新的分配 （对象） 都在此空间内创建。

- Old generation (space): This is where long lived objects exist (and probably die)

  老一代（太空）：这是长寿对象存在的地方（并且可能会死亡）

The idea is that, once an object survives a few garbage collection, it is more likely to survive for a long time. So, objects that survive garbage collection for more than a threshold, will be promoted to old generation. The garbage collector runs more frequently in the young generation and less frequently in the old generation. This is because most objects live for a very short time.

这个想法是，一旦一个对象在几次垃圾回收中幸存下来，它就更有可能存活很长时间。因此，在垃圾回收中存活超过阈值的对象将被提升为老一代。垃圾回收器在 Young 代中运行得更频繁，而在 old 代中运行得不太频繁。这是因为大多数对象的生存期非常短。

We use generational garbage collection to avoid scanning of the whole memory space (like Mark and Sweep approach). In JVM, we have a **minor garbage collection** which is when GC runs inside the young generation and a **major garbage collection (or full GC)** which encompasses garbage collection of both young and old generations.

我们使用分代垃圾回收来避免扫描整个内存空间（就像 Mark 和 Sweep 方法一样）。在 JVM 中，我们有一个**次要的垃圾回收**，即 GC 在年轻一代内部运行，而另一个**主要垃圾回收（或完整的 GC）**包括年轻一代和老一代的垃圾回收。

When doing minor garbage collection, JVM follows every reference from the live roots to the objects in the young generation, and marks those objects as live, which excludes them from the garbage collection process. The problem is that there may be some references from the objects in the old generation to the objects in young generation, which should be considered by GC, meaning those objects in young generation that are referenced by objects in old generation should also be marked as live and excluded from the garbage collection process.

在执行次要垃圾回收时，JVM 会遵循从活动根到新生代中的对象的每个引用，并将这些对象标记为活动对象，从而将它们从垃圾回收过程中排除。问题在于，可能会有一些从老一代对象到新生代对象的引用，GC 应该考虑这些引用，这意味着那些被老一代对象引用的年轻代对象也应该标记为 live 并被排除在垃圾回收过程中。

One approach to solve this problem is to scan all of the objects in the old generation and find their references to young objects. But this approach is in contradiction with the idea of generational garbage collectors. (Why we broke down our memory space into multiple generations in the first place?)

解决此问题的一种方法是扫描老一代中的所有对象，并找到它们对年轻对象的引用。但这种方法与分代垃圾回收器的理念相矛盾。（为什么我们首先将内存空间分解为多代？

Another approach is using write barriers and card table. When an object in old generation writes/updates a reference to an object in the young generation, this action goes through something called write barrier. When JVM sees these write barriers, it updates the corresponding entry in the card table. Card table is a table, which each one of its entries correspond to 512 bytes of the memory. You can think of it as an array containing `0` and `1` items. A `1` entry means there is an object in the corresponding area of the memory which contains references to objects in young generation.

另一种方法是使用 write barriers 和 card table。当老一代对象写入/更新对年轻一代对象的引用时，此操作会通过称为写入屏障的东西。当 JVM 看到这些写入障碍时，它会更新 card 表中的相应条目。Card table 是一个表，它的每个条目对应 512 字节的内存。您可以将其视为包含 `0` 和 `1` 项的数组。`1` 条目表示内存的相应区域中有一个对象，其中包含对年轻一代中的对象的引用。

Now, when minor garbage collection is happening, first every reference from the live roots to young objects are followed and the referenced objects in young generation will be marked as live. Then, instead of scanning all of the old object to find references to the young objects, the card table is scanned. If GC finds any marked area in the card table, it loads the corresponding object and follows its references to young objects and marks them as live either.

现在，当发生次要垃圾回收时，首先会跟踪从活动根到年轻对象的每个引用，年轻一代中的引用对象将被标记为实时。然后，不是扫描所有旧对象以查找对年轻对象的引用，而是扫描 card 表。如果 GC 在 card table 中找到任何标记区域，它会加载相应的对象，并跟踪其对年轻对象的引用，并将它们标记为活动对象。

Share Improve this answer Follow

edited Sep 5, 2020 at 18:58

answered Jul 18, 2018 at 21:54

Hamid Mohayeji

4,265 ● 3 ● 50 ● 59

1 Serious question: Now after the GC runs, the card table is reset. What happens if the old object did not update a reference to the young object, and then the next GC cycle kicks in??? will the young object be wiped, despite that the old object might still be referencing or needing it???

严重的问题：现在 GC 运行后，卡表被重置。如果旧对象没有更新对年轻对象的引用，然后下一个 GC 周期开始，会发生什么情况???尽管旧对象可能仍在引用或需要它，但 Young 对象是否会被擦除???

– Ahmad Tn Oct 17, 2019 at 14:57

1 @AhmadTn why would the card table be reseted? it is alive and well for as long as the application is.

@AhmadTn为什么牌表会被重置？只要应用程序存在，它就一直存在。 – Eugene Feb 22, 2020 at 20:27