http://hllvm.group.iteye.com/group/topic/39376

[讨论] HotSpot VM Serial GC的一个问题

LeafInWind 2014-01-12
看了一段时间的HotSpot GC代码,一直有一个问题,就是新生代上的对象收集: 栈上引用的对象最终似乎都是通过FastScanClosure::do_oop(oop* p)方法移动并修改栈上指针指向移动后新位置的,但这个方法中仅仅移动了被p直接引用的对象,那么被p间接引用的对象呢,它们是如果被移动的,相应的指向它们的指针又是如何修改的. 注意到putfiled方法在实现过程中调用了do_oop_store,其中会将CardTable置dirty. 不知两者之间有关系吗?

RednaxelaFX 2014-01-13
好,多谢开帖！欢迎以后也多来HLLVM群组参与讨论.

普及一下背景知识. 楼主问的是HotSpot VM中的Serial GC,特别是其中的minor GC的实现细节. Minor GC只收集young generation,而使用Serial GC时这个young generation的实现类叫做DefNewGeneration.
关于这些DefNew啊ParNew啊啥的名字的历史,可以参考我之前写的一帖:
http://hllvm.group.iteye.com/group/topic/37095#post-242695

FastScanClosure只在DefNewGeneration的收集中有用到.
HotSpot VM里有很多以*-Closure方式命名的类. 它们其实是封装起来的回调函数. 为了让GC的具体逻辑与对象内部遍历字段的逻辑能松耦合,这部分都是通过回调函数来连接到一起的. ScanClosure与FastScanClosure都可用于DefNewGeneration的扫描.

顺带一提,之前jianglei3000在HLLVM群组发表过一篇新生代回收调试的一些心得,说的正好就是Serial GC的minor GC,值得配合本文一读.

==========

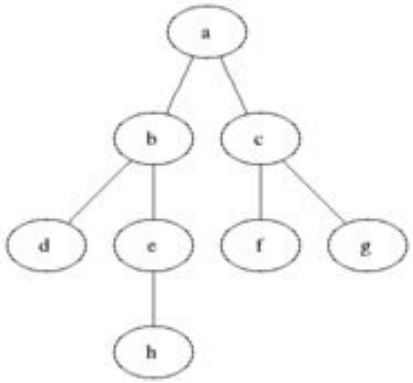
然后从基本入手. HotSpot VM Serial GC的minor GC使用的是Cheney算法的变种,所以先理解基本的Cheney算法有助理清头绪.

说啥都没代码直观,先上代码: 星期天陪老婆去理发的时候有空闲写了个Cheney算法的伪代码:
https://gist.github.com/rednaxelafx/8412637
这个伪代码采用近似C++的语法,其实稍微改改就能实际运行.

这个算法的原始论文是C. J. Cheney在1970年发表的: A nonrecursive list compacting algorithm
该算法在很多书上都有讲解,例如在《The Garbage Collection Handbook》第44页,第4章4.1小节、《ガベージコレクションのアルゴリズムと実装》第4章4.4小节. 每个版本的算法描述都稍微不同,我的伪代码也跟这两本书写的方式稍微不同,但背后要表达的核心思想是一样的就OK了.

Tracing GC的核心操作之一就是从给定的根集合出发去遍历对象图. 对象图是一种有向图,该图的节点是对象,边是引用. 遍历它有两种典型顺序: 深度优先(DFS)和广度优先(BFS).

广度优先遍历的典型实现思路是三色遍历: 给对象赋予白、灰、黑三种颜色以标记其遍历状态:
* 白色: 未遍历到的对象
* 灰色: 已遍历到但还未处理完的对象(意味着该对象尚有未遍历到的出边)
* 黑色: 已遍历完的对象
从Wikipedia引用一张动画图来演示三色遍历:



遍历过程:
1. 一开始,所有对象都是白色的;
2. 把根集合能直接碰到的对象标记为灰色. 在只有一个根对象的地方就只要把那个对象标记为灰色. 但GC通常不是只有一个特定根对象,而是有一个集合的引用作为根,这些引用能直接碰到的对象都要标记为灰色.
3. 然后逐个扫描灰色对象的出边,把这些边能直接碰到的对象标记为灰色. 每当一个对象的所有出边都扫描完了,就把这个对象标记为黑色.
4. 重复第3步直到不再有灰色对象,遍历结束.

这黑白灰要怎么跟实际实现联系起来呢?基本算法会使用一个队列(queue)与一个集合(set):

C++代码

```
1.  void breadth_first_search(Node* root_node) {
2.    // 记录灰色对象的队列
3.    Queue<Node*> scanning;
4.    // 记录黑色对象的集合
5.    Set<Node*>   scanned;
6.
7.    // 1. 一开始对象都是白色的
8.
9.    // 2. 把根对象标记为灰色
10.   scanned.add(root_node);
11.   scanning.enqueue(root_node);
12.
13.   // 3. 逐个扫描灰色对象的出边直到没有灰色对象
14.   while (!scanning.is_empty()) {
15.     Node* parent = scanning.dequeue();
16.     for (Node* child : parent->child_nodes() { // 扫描灰色对象的出边
17.       // 如果出边指向的对象还没有被扫描过
18.       if (child != nullptr && !scanned.contains(child)) {
19.         scanned.add(child);      // 记录下它已经被扫描到了
20.         scanning.enqueue(child); // 也把该对象放进灰色队列里等待扫描
21.       }
22.     }
23.   }
24. }
```

在这种伪代码中,队列scanning与集合scanned组合起来记录了对象的颜色:
* 所有没进入过scanning队列的对象是白色的;
* 进入了scanning队列与scanned集合,但还没离开scanning队列的对象是灰色的;
* 离开了scanning队列并且在scanned集合里的对象是黑色的.

假如对象内可以记录遍历状态的话,那算法可以变为只用一个队列:

C++代码

```
1.  void breadth_first_search(Node* root_node) {
2.    // 记录灰色对象的队列
3.    Queue<Node*> scanning;
4.
5.    // 1. 一开始对象都是白色的
6.
7.    // 2. 把根对象标记为灰色
8.    root_node->set_marked();
9.    scanning.enqueue(root_node);
10.
11.   // 3. 逐个扫描灰色对象的出边直到没有灰色对象
12.   while (!scanning.is_empty()) {
13.     Node* parent = scanning.dequeue();
14.     for (Node* child : parent->child_nodes() { // 扫描灰色对象的出边
15.       // 如果出边指向的对象还没有被扫描过
16.       if (child != nullptr && !child->is_marked()) {
17.         child->set_marked();       // 记录下它已经被扫描到了
```

```
      scanning.enqueue(child); // 也把该对象放进灰色队列里等待扫描
    }
   }
  }
}
```

上面两个版本是有且只有一个特殊对象作为根对象的有向图的广度优先遍历. 假如不是这样而是有一个集合的引用作为根的话,算法会变为:

C++代码

```
void breadth_first_search(Graph* graph) {
  // 记录灰色对象的队列
  Queue<Node*> scanning;

  // 1. 一开始对象都是白色的

  // 2. 把根集合的引用能碰到的对象标记为灰色
  // 由于根集合的引用有可能有重复,所以这里也必须
  // 在把对象加入队列前先检查它是否已经被扫描到了
  for (Node* node : graph->root_edges()) {
    // 如果出边指向的对象还没有被扫描过
    if (node != nullptr && !node->is_marked()) {
      node->set_marked();       // 记录下它已经被扫描到了
      scanning.enqueue(child); // 也把该对象放进灰色队列里等待扫描
    }
  }

  // 3. 逐个扫描灰色对象的出边直到没有灰色对象
  while (!scanning.is_empty()) {
    Node* parent = scanning.dequeue();
    for (Node* child : parent->child_nodes() { // 扫描灰色对象的出边
      // 如果出边指向的对象还没有被扫描过
      if (child != nullptr && !child->is_marked()) {
        child->set_marked();      // 把它记录到黑色集合里
        scanning.enqueue(child); // 也把该对象放进灰色队列里等待扫描
      }
    }
  }
}
```

有没有觉得这算法的样子跟GC越来越像了?

Cheney算法正如上面说的一样,用一个队列来实现对象图的遍历. 比较完整的伪代码可以参考我发这节开头给的链接,其中核心的部分抽取出来如下:

C++代码

```
void garbage_collect(Heap* heap) {
  Semispace* to_space = heap->to_space();

  // 记录灰色对象的队列: 从scanned到to_space->top()
  address scanned = to_space->bottom();

  // 1. 一开始对象都是白色的

  // 2. 把根集合的引用能碰到的对象标记为灰色
  // 由于根集合的引用有可能有重复,所以这里也必须
  // 在把对象加入队列前先检查它是否已经被扫描到了
  for (Object** refLoc : heap->root_reference_locations()) {
    Object* obj = *refLoc;
    if (obj != nullptr) {
      if (!obj->is_forwarded()) {
```

```
      // 记录下它已经被扫描到了,也把该对象放进灰色队列里等待扫描
      size_t size = obj->size();
      address new_addr = to_space->allocate(size);

      // address Semispace::allocate(size_t size) {
      //   if (_top + size < _end) {
      //     address new_addr = _top;
      //     _top += size;
      //     return new_addr;
      //   } else {
      //     return nullptr;
      //   }
      // }

      // to_space->allocate()移动了to_space->top()指针,
      // 等同于scanning.enqueue(obj);

      copy(/* to */ new_addr, /* from */ obj, size);
      Object* new_obj = (Object*) new_addr;
      obj->forward_to(new_obj);  // 设置转发指针(forwarding pointer)
      *refLoc = new_obj;         // 修正指针指向新对象
    } else {
      *refLoc = obj->forwardee(); // 修正指针指向新对象
    }
  }
}

// 3. 逐个扫描灰色对象的出边直到没有灰色对象
while (scanned < to_space->top()) {
  Object* parent = (Object*) scanned;
  // 扫描灰色对象的出边
  for (Object** fieldLoc : parent->object_fields()) {
    Object* obj = *fieldLoc;
    // 如果出边指向的对象还没有被扫描过
    if (obj != nullptr) {
      if (!obj->is_forwarded()) {    // 尚未被扫描过的对象
        // 记录下它已经被扫描到了,也把该对象放进灰色队列里等待扫描
        size_t size = obj->size();
        address new_addr = to_space->allocate(size);

        // to_space->allocate()移动了to_space->top()指针,
        // 等同于scanning.enqueue(obj);

        copy(/* to */ new_addr, /* from */ obj, size);
        Object* new_obj = (Object*) new_addr;
        obj->forward_to(new_obj);    // 设置转发指针(forwarding pointer)
        *fieldLoc = new_obj;         // 修正指针指向新对象
      } else {                       // 已经扫描过的对象
        *fieldLoc = obj->forwardee(); // 修正指针指向新对象
      }
    }
  }
  scanned += parent->size();
  // 移动scanned指针等同于scanning.dequeue(parent);
}
}
```

它的设计非常精妙:

1. 它使用一块连续的地址空间来实现GC堆,并将其划分为2个半分空间(semispace),分别称为from-space与to-space. 平时只用其中一个,也就是from-space;
2. 逐个扫描指针,每扫描到一个对象的时候就把它从from-space拷贝到to-space,并在原来的对象里记录下一个转发指针(forwarding pointer),记住该对象被拷贝到哪里了. 要知道一个对象有没有被扫描(标记)过,只要看该对象是否有转发指针即可;
3. 每扫描完一个指针就顺便把该指针修正为指向拷贝后的新对象. 这样,对象的标记(mark)、整理(compaction)、指针的修正就合起来在一步都做好了;
4. 它不需要显式为扫描队列分配空间,而是复用了to-space的一部分用作隐式队列. 用一个scanned指针来区分to-space的对象的颜色: 在to-space开头到scanned指针之间的对象是黑色的,在scanned指针到to-space已分配对象的区域的末尾之间的对象是灰色的. 如何知道还有没有灰色对象呢?只要scanned追上了to-space已分配对象区域的末尾就好了.
这种做法也叫做“两手指”(two-finger): “scanned”与“free”. 只需要这两个指针就能维护隐式扫描队列. “free”在我的伪代码里就是to_space->top().

Cheney算法GC工作时,to-space中各指针的样子如下:

```
|[ 已分配并且已扫描完的对象 ]|[ 已分配但未扫描完的对象 ]|[ 未分配空间 ]|
^                          ^                          ^             ^
bottom                     scanned                    top           end
```

在GC结束时,不需要对原本的from-space做什么清理动作,只要把它的分配指针(top)设回到初始位置(bottom)即可. 之前在里面的对象就当作不存在了. 自然,也就不需要清理其中设置了的转发指针.

要想看图解Cheney算法的工作过程的,请参考这个幻灯片:
http://www.ps.uni-saarland.de/courses/gc-ws01/slides/copying_gc.pdf
我现在懒得画图...这个幻灯片已经画得足够好了,直接看它吧.
(不过它所图解的过程跟我的伪代码有一点点细微的差异,别介意...)

后面的回复LeafInWind问道:

> **LeafInWind 写道**
> 不知理解是否有误,感觉while的条件应该是*scanned==null.

while的条件就是scanned < to_space->top(). 或者说条件是scanned != to_space->top()也行. 反正就是: 如果scanned还没追上to_space->top(),继续循环.
scanned是一个指针,*scanned是一个Object实体. 只有指针可以等于null,对象实体就是一个实在的东西,不可能是“null”.

Cheney算法是一个非常非常简单且高效的GC算法. 看前面我写的伪代码就可以有直观的感受它有多简单. 它的实现代码恐怕比简易mark-sweep还简单.
但为啥很多简易的VM宁可采用mark-sweep而不用Cheney算法的copying GC呢?

因为mark-sweep GC的常规实现不移动对象,而copying GC必须移动对象. 移动对象意味着使用GC的程序(术语叫做mutator)需要做更多事情,例如说要能准确定位到所有的指针,以便在对象移动之后修正指针. 很多简易VM都偷懒不想记住所有指针的位置,所以无法支持copying GC.

关于找指针的问题,我之前也写过一点: 找出栈上的指针/引用

有一种叫做Bartlett风格的mostly-copying GC可以弱化定位指针的需求: 根集合之中不需要准确定位指针,只要保守的扫描; 在GC堆内的对象则必须要能准确定位其中的指针. 有兴趣的可以参考其原始论文: Mostly-Copying Garbage Collection Picks Up Generations and C++

Cheney算法的简单优雅之处来自它通过隐式队列来实现广度优先遍历,但它的缺点之一却也在此: 广度优先的拷贝顺序使得GC后对象的空间局部性(memory locality)变差了. 但是如果要改为真的深度优先顺序就会需要一个栈,无论是隐式(通常意味着递归调用)或者是显式.
使用递归实现的隐患是容易爆栈,有没有啥办法模拟深度优先的拷贝顺序但不用栈呢?这方面有很多研究. 其中一种有趣的做法是IBM的hierarchical copying GC,参考: Improving Locality with Parallel Hierarchical Copying GC

==========

相比基本的Cheney算法,HotSpot VM Serial GC有什么异同呢?

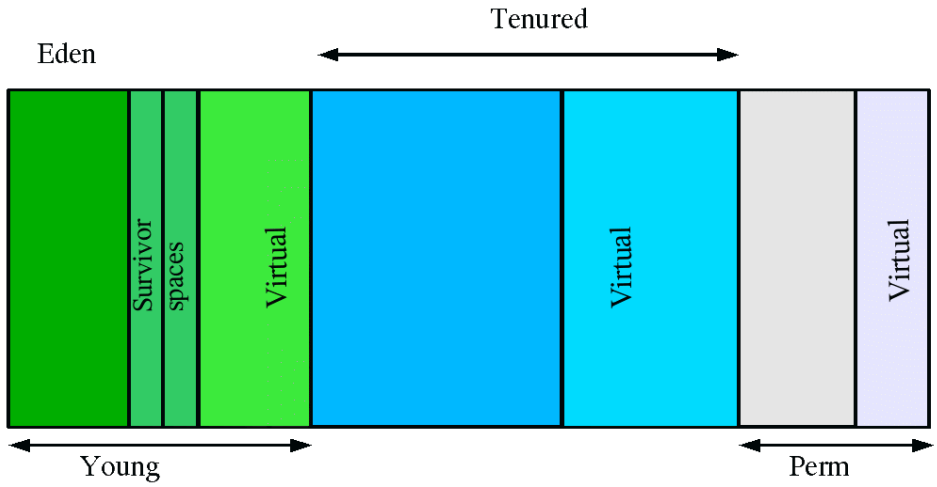
相同点:
1. 使用广度优先遍历;
2. 使用隐式队列;
3. copy等同mark + relocate (compact) + remap (pointer fixup)三件事一步完成.

在HotSpot VM里,copying GC用了scavenge这个名字,说的是完全相同的事.

相异点:
1. 基本Cheney算法不分代,而HotSpot的GC分两代
2. 基本Cheney算法使用2个半分空间(semispace),而HotSpot的GC在young generation使用3个空间——1个eden与两个survivor space. 注意这两个survivor space就与semispace的作用类似.

在G1 GC之前,所有HotSpot VM的GC堆布局都继承自1984年David Ungar在Berkeley Smalltalk里所实现的Generation Scavenging. 参考论文: Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm

GC堆布局:



那我们一点点把基本的Cheney算法映射过来.

* 基本Cheney算法用from-space和to-space,而HotSpot VM的DefNewGeneration有三个空间,eden space、from-space、to-space. 后者的eden space + from-space大致等于前者的from-space,而后者的to-space + old gen的一部分大致等于前者的to-space.

* 拷贝对象的目标空间不一定是to-space,也有可能是old generation,也就是对象有可能会从young generation晋升到old generation.
为了实现这一功能,对象头的mark word里有一小块地方记录对象的年龄(age),也就是该对象经历了多少次minor GC. 如果扫描到一个对象,并且其年龄大于某个阈值(tenuring threshold),则该对象会被拷贝到old generation; 如果年龄不大于那个阈值则拷贝到to-space.
要留意的是,基本Cheney算法中2个半分空间通常一样大,所以可以保证所有from-space里活着的对象都能在to-space里找到位置. 但HotSpot VM的from-space与to-space通常比eden space小得多,不一定能容纳下所有活的对象. 如果一次minor GC的过程中,to-space已经装满之后还遇到活对象要拷贝,则剩下的对象都得晋升到old generation去. 这种现象叫做过早晋升(premature tenuring),要尽量避免.

* 既然拷贝去的目标空间不一定是to-space,那原本Cheney算法里的隐式扫描队列会在哪里?
答案是既在to-space,也在old generation. 很简单,在这两个空间都记录它们的scanned指针(叫做“saved mark”),这俩空间各自原本也记录着它们的分配指针(“top”),之间的部分就用作扫描队列.

* Forwarding pointer安装在对象(oopDesc)头部的mark word(markOop)里. 只有在minor GC的时候才会把已拷贝的对象的mark word借用来放转发指针.

* 通过回调,把遍历逻辑与实际动作分离. 例如说,遍历根集合的逻辑封装在GenCollectedHeap::gen_process_strong_roots()、SharedHeap::process_strong_roots()里,遍历对象里的引用类型字段的逻辑封装在oopDesc::oop_iterate()系的函数里; 而实际拷贝对象的动作则由FastScanClosure::do_work()负责调用.

* 基本Cheney算法的“scanned”指针,在HotSpot Serial GC里是每个space的“saved mark”. 相关操作的函数名是: “save_marks()” / “set_saved_mark()”、“reset_saved_mark()”、“no_allocs_since_save_marks()” / “saved_mark_at_top()”

看看遍历循环的结束条件(循环条件的反条件):

C++代码

```
1.  bool saved_mark_at_top() const { return saved_mark_word() == top(); }
```

跟基本Cheney算法的循环条件 scanned != top() 一样.

为啥我的伪代码里scanned是个局部变量,而HotSpot里变成了每个空间的成员字段?因为使用回调函数来分离遍历逻辑与实际动作,代码结构变了,这个scanned指针也只好另找地方放来让需要访问它的地方都能访问到.

* HotSpot VM的分代式GC需要通过写屏障(write barrier)来维护一个记忆集合(remember set)——记录从old generation到young generation的跨代引用的数据结构. 具体在代码中叫做CardTable. 在minor GC时,old generation被remember set所记录下的区域会被看作根集合的一部分. 而在minor GC过程中,每当有对象晋升到old generation都有可能产生新的跨代引用.
所以FastScanClosure::do_work()里也有调用写屏障的逻辑: OopsInGenClosure::do_barrier().

* HotSpot VM要支持Java的弱引用. 在GC的时候有些特殊处理要做. 可以参考这帖:
http://hllvm.group.iteye.com/group/topic/27945?page=2#post-209812

* HotSpot VM的GC必须处理一些特殊情况,一个极端的例子是to-space和old generation的剩余空间加起来都无法容纳eden与from-space的活对象,导致GC无法完成. 这使得许多地方代码看起来很复杂. 但要了解主要工作流程的话可以先不关心这些旁支逻辑.

HotSpot代码与我的伪代码的几个对应关系:

HotSpot => 我的Cheney算法伪代码

* saved mark => scanned
* FastScanClosure => process_reference()
* GenCollectedHeap::gen_process_strong_roots()、SharedHeap::process_strong_roots() => 遍历根集合的循环逻辑
* FastEvacuateFollowersClosure => 遍历扫描队列的循环
* oopDesc::oop_iterate() => 遍历对象的引用类型字段的循环,Object::object_fields()
* DefNewGeneration::copy_to_survivor_space() => 拷贝对象的逻辑,Heap::evacuate()

其它应该都挺直观的?

==========

Oracle Labs的Maxine VM里也有过Cheney算法的实现,在这里:
https://kenai.com/projects/maxine/sources/maxine/content/com.oracle.max.vm/src/com/sun/max/vm/heap/sequential/semiSpace/SemiSpaceHeapScheme.java?rev=8747
也可以参考来看看. 纯Java实现的GC喔.

==========

顺带一提,早期V8的分代式GC中的minor GC也是基于Cheney算法,而且更接近原始的基本算法——只用2个半分空间. 最大的区别就是由于是分代式GC的一部分,它也跟HotSpot一样要处理对象晋升和写屏障.

Jay Conrod: A tour of V8: Garbage Collection(中文翻译: V8 之旅: 垃圾回收器)

就先写这么多了…

LeafInWind 2014-01-14
R神何时继续

RednaxelaFX 2014-01-14
> **LeafInWind 写道**
> R神何时继续

马上马上. 家里有点事结果没来得及把字码好. 楼主先看这个,我待会儿码字说明它是什么:
https://gist.github.com/rednaxelafx/8412637

LeafInWind 2014-01-14
看了伪代码,觉得最精妙的地方就在下面这段to-space中的宽度优先scan

Java代码

```
1.  // breadth-first scanning of object graph
2.  while (scanned < _to_space->top()) {
3.    Object* parent_obj = (Object*) scanned;
4.    foreach (Object** slot in parent_obj->obejct_fields()) {
5.      process_reference(slot);
6.    }
7.    scanned += parent_obj->size();
8.  }
```

由于对象是被顺序依次移动到to-space的,因此上述宽度优先的scan过程能保证所有间接引用都被扫描到移动到. 上述过程实质是一个队列的填充过程.
不知理解是否有误,感觉while的条件应该是*scanned==null.
另外仍然有两个问题:
1. 上述scan过程对应的hotspot代码是什么?
2. parent_obj->object_fields()在hotspot中有对应的直接实现吗?

LeafInWind 2014-01-14
其实被stack上root引用的对象从from区移动到to区的过程也蛮精妙的: 先移走,然后修改原对象的mark指向新位置,这样如果下次遇到一个root也指向该对象,就再不需要移动而可以直接修改root的指向了.

RednaxelaFX 2014-01-15
更新了我前面的回复. 楼主看看?

LeafInWind 2014-01-15
感谢R神精彩的讲解

RednaxelaFX 2014-01-15
> **LeafInWind 写道**
> 感谢R神精彩的讲解

有疑问的话欢迎继续提

LeafInWind 2014-01-15
> **引用**
>
> 后面的回复LeafInWind问道:
> LeafInWind 写道
> 不知理解是否有误,感觉while的条件应该是*scanned==null.
>
> while的条件就是scanned < to_space->top(). 或者说条件是scanned != to_space->top()也行. 反正就是: 如果scanned还没追上to_space->top(),继续循环.
> scanned是一个指针,*scanned是一个Object实体. 只有指针可以等于null,对象实体就是一个实在的东西,不可能是“null”.

R神的代码没有问题,是我对top的理解有误,top其实是随着对象的分配在不断的递减的,我把它当做end了. 另外本意是想说*scanned!=null,并且这里只是将null当做一个空间未被分配的标记,当然设这个标记还需要对空间初始化什么的,不好,所以还是R神的条件好.

RednaxelaFX 2014-01-15

> **LeafInWind 写道**
> R神的代码没有问题,是我对top的理解有误,top其实是随着对象的分配在不断的递减的,我把它当做end了

是说递增?

LeafInWind 2014-01-15

> **RednaxelaFX 写道**
> > **LeafInWind 写道**
> > R神的代码没有问题,是我对top的理解有误,top其实是随着对象的分配在不断的递减的,我把它当做end了
>
> 是说递增?

又看了代码,的确应该是递增

LeafInWind 2014-01-16

仔细看了hotspot的FastEvacuateFollowersClosure的代码,感觉好累,都是嵌套的宏定义,代码跳转都没有用了,不得不一点一点手动记下来,好累
另外也不知后缀 _v 和 _nv 是什么意义,n是表示narrow吗?

RednaxelaFX 2014-01-17

有_v与_nv后缀的方法,前者是virtual版,后者是non-virtual版.
HotSpot里有不少代码写得很纠结. 一方面在设计的时候想尽量灵活,所以很多函数都声明为虚函数; 另一方面某些调用频率非常高的函数如果是虚函数实在是太慢了,所以就有很hack的办法硬生生的把某些类虚函数又加上非虚版本.

这种细节在读代码了解大体实现思路的时候要尽量忽略掉…不然会看到吐的

LeafInWind 2014-01-20

今天仔细看了FastEvacuateFollowersClosure::do_void(),有一个问题！
首先,FastEvacuateFollowersClosure::do_void()代码如下:

C++代码

```
1.  do {
2.    _gch->oop_since_save_marks_iterate(_level, _scan_cur_or_nonheap,
3.                                       _scan_older);
4.  } while (!_gch->no_allocs_since_save_marks(_level));
```

对SerialGC来说,就是要分别调用DefNewGeneration和TenuredGeneration的oop_since_save_marks_iterate,而前者的代码如下

C++代码

```
1.  void DefNewGeneration::                                \
2.  oop_since_save_marks_iterate##nv_suffix(OopClosureType* cl) {   \
3.    cl->set_generation(this);                           \
4.    eden()->oop_since_save_marks_iterate##nv_suffix(cl);        \
5.    to()->oop_since_save_marks_iterate##nv_suffix(cl);          \
6.    from()->oop_since_save_marks_iterate##nv_suffix(cl);        \
7.    cl->reset_generation();                             \
8.    save_marks();                                       \
9.  }
```

调用to()的oop_since_save_marks_iterate当然没问题,这正是遍历引用链需要的;
调用from()的oop_since_save_marks_iterate也就算了,对象不会分配在from上,其top和save mark都来自上一次GC时的to区域,因此应该是相等的,故不会遍历;
但调用eden()的oop_since_save_marks_iterate似乎就有问题了:

- 除非对象在eden上分配时,修改top指针的同时也修改save mark,否则这里top不等于save mark,就将会有实际的遍历发生了啊

目前没有找到在eden区分配对象改变top的同时改变save mark的代码,不知是我不够耐心,还是我理解有误,存在其他机制.

RednaxelaFX 2014-01-21
哈哈,好问题. 其实前面我的说明里已经覆盖到了这个地方,但不像楼主那么仔细的读代码的话是不会发现这种问题的. Good job.

> **RednaxelaFX 写道**
> * HotSpot VM的GC必须处理一些特殊情况,一个极端的例子是to-space和old generation的剩余空间加起来都无法容纳eden与from-space的活对象,导致GC无法完成. 这使得许多地方代码看起来很复杂. 但要了解主要工作流程的话可以先不关心这些旁支逻辑.

就是这里. HotSpot VM的某些GC在“generational GC framework”里,它们最初实现的时候尽量做得比较通用,以便任何young generation可以跟old generation混在一起用. DefNew就是这个框架里的一个young generation的实现. 所以为了保证能处理任何可能性,DefNew里甚至能容忍在GC时在eden里分配对象...但因为没有实际代码真的用到了这个地方,说不定其实这个支持已经用不了了,只是代码还在那里...

LeafInWind 2014-01-21
谢谢R神的回复,又看了代码,终于找到了如下问题的答案

> **引用**
> 但调用eden()的oop_since_save_marks_iterate似乎就有问题了:

答案就在于GenCollectedHeap::do_collection方法中

C++代码

```
1.  // Do collection work
2.  {
3.    HandleMark hm;  // Discard invalid handles created during gc
4.    save_marks();   // save marks for all gens
5.    ...
6.    _gens[i]->collect(full, do_clear_all_soft_refs, size, is_tlab);
```

可以看到,在调用各个代的collect前,先调用了save_marks()方法,该方法会将所有代的所有space的save mark置为与top相等. 故collect方法中最后对eden和from的遍历不会有实际动作.

另外,下面的说法似乎有问题

> **引用**
> 调用from()的oop_since_save_marks_iterate也就算了,对象不会分配在from上,其top和save mark都来自上一次GC时的to区域,因此应该是相等的,故不会遍历;

DefNewGeneration::allocate_from_space就是在from区间直接分配对象.

LeafInWind 2014-01-21
再问一个问题

C++代码

```
1.  void GenCollectedHeap::
2.  gen_process_strong_roots(int level,
3.                           bool younger_gens_as_roots,
4.                           bool activate_scope,
5.                           bool collecting_perm_gen,
6.                           SharedHeap::ScanningOption so,
7.                           OopsInGenClosure* not_older_gens,
8.                           bool do_code_roots,
9.                           OopsInGenClosure* older_gens) {
10.   // General strong roots.
11.
12.   if (!do_code_roots) {
13.     SharedHeap::process_strong_roots(...);
14.   } else {
15.     bool do_code_marking = (activate_scope || nmethod::oops_do_marking_is_active());
16.     CodeBlobToOopClosure code_roots(not_older_gens, do_code_marking);
```

```
17.     SharedHeap::process_strong_roots(...);
18.   }
19.
20.   if (younger_gens_as_roots) {
21.     if (!_gen_process_strong_tasks->is_task_claimed(GCH_PS_younger_gens)){
22.       for (int i = 0; i < level; i++) {
23.         not_older_gens->set_generation(_gens[i]);
24.         _gens[i]->oop_iterate(not_older_gens);
25.       }
26.       not_older_gens->reset_generation();
27.     }
28.   }
29.   // When collection is parallel, all threads get to cooperate to do
30.   // older-gen scanning.
31.   for (int i = level+1; i < _n_gens; i++) {
32.     older_gens->set_generation(_gens[i]);
33.     rem_set()->younger_refs_iterate(_gens[i], older_gens);
34.     older_gens->reset_generation();
35.   }
36.
37.   _gen_process_strong_tasks->all_tasks_completed();
38. }
```

以上代码中
1. process_strong_roots负责将栈上的root直接引用的并且在DefNew中的对象移动到to区域或者年老代,但如果root引用的对象在年老代,就不动它了.
2. if (younger_gens_as_roots) {}对于DefNew没有意义,因为其level就是0.
3. for (int i = level+1; i < _n_gens; i++) {...}负责遍历年老代中dirty card的对应区域,作为根节点查找并移动DefNew中被引用的对象,同时清除dirty card,如果对象移动后仍然在DefNew中,还需将相应card 置dirty.

上面第3步正是minor GC的关键所在,因为本来第1步中栈上的root完全有可能引用年老代中的对象,但minor GC却忽略它们,而只通过第3步做一个模糊的遍历,否则年老代中的对象可能存在大量的引用,其引用链的遍历代价可能就很大了.

请问R神,我对上述三个步骤的总结以及对最后minor GC的理解是否正确.

RednaxelaFX 2014-01-22

> **LeafInWind 写道**
> 1. process_strong_roots负责将栈上的root直接引用的并且在DefNew中的对象移动到to区域或者年老代,但如果root引用的对象在年老代,就不动它了.

差不多.
SharedHeap::process_strong_roots()扫描所有一定是根的东西,包括
* Universe所持有的一些必须活着的对象,
* 所有JNI handles,
* 所有线程的栈,
* 所有当前被锁着的对象
* VM内实现的MBean所持有的对象
* JVMTI所持有的对象
* (可选): 所有已加载的类 或 所有已加载的系统类 (SystemDictionary)
* (可选): 所有驻留字符串(StringTable)
* (可选): 代码缓存(CodeCache)
* (可选): PermGen

不只是线程栈喔.

HotSpot VM的GC在minor GC的时候只收集young generation,不收集old generation和perm generation,不卸载类(class unloading),也不卸载. 所以在这种条件下,下述几项都是strong root:
* 所有已加载的类
* 所有驻留字符串

* 所有动态生成的代码(CodeCache)
* PermGen的remember set所记录的存在跨代引用的区域

在full GC的时候,上述一些项就不需要是strong root了.

然后,当只收集GC堆的一部分而不是收集全堆时,GenCollectedHeap::gen_process_strong_roots()在上述基础上会进一步把跨代引用当作strong root. 对Serial GC的minor GC来说,它收集young generation而不收集old/perm generation,所以后两者中的跨代引用就是strong root,所以要扫描后两者的remember set.

> **LeafInWind 写道**
> 2. if (younger_gens_as_roots) {}对于DefNew没有意义,因为其level就是0.

嗯没错.
HotSpot VM里唯一用到这个地方的就是CMS的并发收集,它只收集old generation,可选收集perm generation,不收集young generation. 所以它需要扫描young generation作为strong root. 可以参考CMSCollector::checkpointRootsInitialWork() 和 CMSCollector::do_remark_non_parallel().

> **LeafInWind 写道**
> 3. for (int i = level+1; i < _n_gens; i++) {...}负责遍历年老代中dirty card的对应区域,作为根节点查找并移动DefNew中被引用的对象,同时清除dirty card,如果对象移动后仍然在DefNew中,还需将相应card 置dirty.

差不多. 移动对象的时候**不会**把原本dirty的card清掉. 你想像一下,如果一个old generation里的对象原本有引用到young generation,那么被引用的对象如果被移动到to space了,它还是活在young generation里的,没必要清理掉old generation哪个对应的card.
其实每次移动对象都会检查一下是否产生了新的跨代引用(例如说有对象晋升到了old generation,而该对象还引用着在young generation里的对象). 如果有的话就得相应把card置dirty.

LeafInWind 2014-01-22

> **引用**
> 移动对象的时候不会把原本dirty的card清掉.

又看了代码,还是觉得原本的dirty card会清掉. 首先,理论上说,如果young gen中的对象被移到了to区域,其位置变了,因此原来的card应该清掉,然后将新位置对应card置dirty; 其次,如果对象被移动了old gen,那么card已经没有意义了,当然应该清掉.

看代码,第3步中最终调用的是CardTableModRefBS::non_clean_card_iterate_possibly_parallel,其代码如下

C++代码

```
1.  DirtyCardToOopClosure* dcto_cl = sp->new_dcto_cl(cl, precision(),
2.                                                   cl->gen_boundary());
3.  ClearNoncleanCardWrapper clear_cl(dcto_cl, ct);
4.
5.  clear_cl.do_MemRegion(mr);
```

从ClearNoncleanCardWrapper的命名看,它就应该会清card,然后看它的代码,有如下语句

C++代码

```
1.  if ((*cur_entry != CardTableRS::clean_card_val()) && clear_card(cur_entry)) {
```

当当前card not clean时,就会调用clear_card(cur_entry)将当前card清掉.

> **引用**
> 其实每次移动对象都会检查一下是否产生了新的跨代引用(例如说有对象晋升到了old generation,而该对象还引用着在young generation里的对象). 如果有的话就得相应把card置dirty.

这个移动我觉得应该是不需要检查跨代引用及修改card table的. 因为这些对象移动后都还在save mark和top之间,因此oop_since_save_mark_iterate方法能够完成对对象中所有oop field的scan,不需要借助card table. 另外从代码看,移动对象使用的始终是gc_barrier为false的FastScanClosure,因此也不会修改card table.

RednaxelaFX 2014-01-22
你得了解CardTable是用来干嘛的.

如果有这样的对象关系:

Java代码

```
1. Object young_gen_object = new Object();
2. old_gen_object.field = young_gen_object;
```

那么是old_gen_object所在的地方对应的card会被dirty掉,因为是它持有跨代引用.
至于young_gen_object具体在什么地址都没关系,只要在young generation里. 它对应的card不会被dirty.

所以在minor GC的时候,如果young generation里有对象移动到to-space了,那原本在old generation里的CardTable完全不受影响.

> **LeafInWind 写道**
> 这个移动我觉得应该是不需要检查跨代引用及修改card table的. 因为这些对象移动后都还在save mark和top之间,因此oop_since_save_mark_iterate方法能够完成对对象中所有oop field的scan,不需要借助card table.

但你不记得还会有下一次minor GC…下次就要用到新dirty的card了

> **LeafInWind 写道**
> 另外从代码看,移动对象使用的始终是gc_barrier为false的FastScanClosure,因此也不会修改card table.

你没看到这个:

C++代码

```
1. FastScanClosure fsc_with_gc_barrier(this, true);
```

以及后面FastEvacuateFollowersClosure evacuate_followers的构造、gch->gen_process_strong_roots()的参数都有用到带GC barrier的版本

LeafInWind 2014-01-22

> **R神 写道**
> 那么是old_gen_object所在的地方对应的card会被dirty掉,因为是它持有跨代引用.
> 至于young_gen_object具体在什么地址都没关系,只要在young generation里. 它对应的card不会被dirty.

这个R神是对的,的确是我理解错了. 但gen_process_strong_roots中如下这段代码

C++代码

```
1. older_gens->set_generation(_gens[i]);
2. rem_set()->younger_refs_iterate(_gens[i], older_gens);
3. older_gens->reset_generation();
```

我仍然认为是先清card,然后如果对象仍然在young gen中,则将相同的card置dirty,这是靠older_gens完成的,而如果对象移到old gen了,older_gens就不会再置了.

另外SharedHeap::process_strong_roots方法中,虽然not_older_gens和older_gens两个FastScanClosure都传过去了,但至少移动栈上引用时用的都是not_older_gens,因此即使对象移到了old gen,也不会去修改相应card.
但

> **R神 写道**
> 但你不记得还会有下一次minor GC…下次就要用到新dirty的card了

似乎又是对的,这里感觉有点想不通了

哈哈,想通了. SharedHeap::process_strong_roots方法中,对象移动到old gen时,的确不会将相应card置dirty,实际上它也置不了,还没遍历oop呢,是否是跨代引用都不知道. 但evacuate_followers.do_void()在处理old gen时使用的是fsc_with_gc_barrier,此时就会发现跨代引用并置相应card为dirty了.

RednaxelaFX 2014-01-23

> **LeafInWind 写道**
> > **RednaxelaFX 写道**
> > 那么是old_gen_object所在的地方对应的card会被dirty掉,因为是它持有跨代引用.
> > 至于young_gen_object具体在什么地址都没关系,只要在young generation里. 它对应的card不会被dirty.
>
> 这个R神是对的,的确是我理解错了. 但gen_process_strong_roots中如下这段代码

C++代码

```
1.  older_gens->set_generation(_gens[i]);
2.  rem_set()->younger_refs_iterate(_gens[i], older_gens);
3.  older_gens->reset_generation();
```

> 我仍然认为是先清card,然后如果对象仍然在young gen中,则将相同的card置dirty,这是靠older_gens完成的,而如果对象移到old gen了,older_gens就不会再置了.

这个理解是对的. 实际动作是先清了card再扫描它,扫描过程中如果发现活对象涉及跨代引用,那对应的card就会被再次dirty. 结果来看很多dirty card就还是dirty的.

LeafInWind 2014-01-24
这两天看了一下full gc,能否请教R神一个问题:
make-compact的第三步,调整所有的oop指向第二步中计算出的新位置,在SCAN_AND_ADJUST_POINTERS这个宏的开头,有如下代码:

C++代码

```
1.  HeapWord* q = bottom();
2.  HeapWord* t = _end_of_live;
3.  if (q < t && _first_dead > q &&
4.      !oop(q)->is_gc_marked()) {
5.    /* we have a chunk of the space which hasn't moved and we've
6.     * reinitialized the mark word during the previous pass, so we can't
7.     * use is_gc_marked for the traversal. */
8.    HeapWord* end = _first_dead;
9.    ...
10. }
```

我理解_first_dead是一个指向当前contiguous space中第一个dead obj的指针,那么如果q<_first_dead的话,q就应该是live obj,那么相应的“oop(q)->is_gc_marked()”这个条件就应该一定成立,那么上面这段代码就毫无意义了.
从代码中的注释看,[bottom, _first_dead)这段区间似乎有一点特殊,但不知特殊在哪里??不知R神能否指点一下,谢谢！

又看了一下代码,现在基本想明白了. [bottom, _first_dead)这段区间中的对象的确比较特殊,特殊就特殊在它们虽然活着,但compact的时候并不需要移动. 因此虽然在mark-compact的第一个阶段被标上了gc mark,但在第二阶段计算移动后新位置时又故意把gc mark去掉了. 代码如下:

C++代码

```
1.  HeapWord* CompactibleSpace::forward(...){
2.    ...
3.    if ((HeapWord*)q != compact_top) {
4.      q->forward_to(oop(compact_top));
5.      assert(q->is_gc_marked(), "encoding the pointer should preserve the mark");
6.    } else {  //q==compact_top说明当前对象不需要移动
7.      // if the object isn't moving we can just set the mark to the default
8.      // mark and handle it specially later on.
9.      q->init_mark();
10.     assert(q->forwardee() == NULL, "should be forwarded to NULL");
11.   }
12. }
```

所有在[bottom, _first_dead)这段区间中的对象虽然是活对象,但并没有gc mark. 但尽管没有gc mark,其中的oop还是需要adjust的. 所以才有SCAN_AND_ADJUST_POINTERS宏开头的一大段代码.
请教R神,上述理解有什么问题吗?！

LeafInWind 2014-01-25
还是有一个问题,一次full gc过后,存活对象的年龄要长一岁吗,如果长的话,那young gen里超过阈值的对象是否在full gc过程中也应该promote到old gen去了呢???

RednaxelaFX 2014-01-25

> **LeafInWind 写道**
> 还是有一个问题,一次full gc过后,存活对象的年龄要长一岁吗,如果长的话,那young gen里超过阈值的对象是否在full gc过程中也应该promote到old gen去了呢???

一次full GC过后,在理想情况下young generation里所有活的对象都被晋升到old generation了,不论age如何.
就是说只要old generation还有地方,young generation所有活对象都要拷贝过去.
HotSpot VM里对象的age只是经历的minor GC的次数.

LeafInWind 2014-01-25
感谢R神,之前没有细想,还以为是各个gen各自压缩的. 又看了代码,由于第二阶段计算移动后新位置的过程中使用的始终是同一个CompactPoint对象,因此按照CompactibleSpace::forward中的如下代码:

C++代码

```
1.  size_t compaction_max_size = pointer_delta(end(), compact_top);
2.  while (size > compaction_max_size) {
3.    // switch to next compaction space
4.    cp->space->set_compaction_top(compact_top);
5.    cp->space = cp->space->next_compaction_space();
6.    if (cp->space == NULL) {
7.      cp->gen = GenCollectedHeap::heap()->prev_gen(cp->gen);
8.      assert(cp->gen != NULL, "compaction must succeed");
9.      cp->space = cp->gen->first_compaction_space();
10.     assert(cp->space != NULL, "generation must have a first compaction space");
11.   }
12.   compact_top = cp->space->bottom();
13.   cp->space->set_compaction_top(compact_top);
14.   cp->threshold = cp->space->initialize_threshold();
15.   compaction_max_size = pointer_delta(cp->space->end(), compact_top);
16. }
```

应该是将old gen和young gen作为一个整体,将其中的存活对象依次**压满**old gen,然后是eden和from.
请教R神,不知我的补充是否正确.
另外,还想请教一下full gc和minor gc是什么关系,full gc是否会伴随minor gc.
还有,一次full gc后,对象的年龄是清零还是不变?

Pseudo-code that implements Cheney's algorithm for copying GC

// Pseudo-code that implements Cheney's algorithm

```
class Object {
 // remains null for normal objects
 // non-null for forwarded objects
 Object* _forwardee;

public:
 void forward_to(address new_addr);
 Object* forwardee();
 bool is_forwarded();
 size_t size();
 Iterator<Object**> object_fields();
};

class Heap {
 Semispace* _from_space;
 Semispace* _to_space;

 void swap_spaces();
 Object* evacuate(Object* obj);

public:
 Heap(address bottom, address end);

 address allocate(size_t size);
 void collect();
 void process_reference(Object** slot);
};

class Semispace {
 address _bottom;
 address _top;
 address _end;

public:
 Semispace(address bottom, address end);

 address bottom() { return _bottom; }
 address top()    { return _top; }
 address end()    { return _end; }

 bool contains(address obj);
 address allocate(size_t size);
 void reset();
};

void Object::forward_to(address new_addr) {
 _forwardee = new_addr;
}

Object* forwardee() {
 return _forwardee;
}

bool Object::is_forwarded() {
 return _forwardee != nullptr;
}

// Initialize the heap. Assuming contiguous address space
Heap::Heap(address bottom, address end) {
 size_t space_size = (end - bottom) / 2;
 address boundary = bottom + space_size;
 _from_space = new Semispace(bottom, boundary);
 _to_space   = new Semispace(boundary, end);
}

void Heap::swap_spaces() {
 // Swap the two semispaces.
```

```
 // std:swap(_from_space, _to_space);
 Semispace* temp = _from_space;
 _from_space = _to_space;
 _to_space = temp;

 // After swapping, the to-space is assumed to be empty.
 // Reset its allocation pointer.
 _to_space->reset();
}

address Heap::allocate(size_t size) {
 return _from_space->allocate();
}

Object* Heap::evacuate(Object* obj) {
 size_t size = obj->size();

 // allocate space in to_space and copy object to there
 address new_addr = _to_space->allocate(size);
 copy(/* to */ new_addr, /* from */ obj, size);

 // set forwarding pointer in old object
 Object* new_obj = (Object*) new_addr;
 obj->forward_to(new_obj);

 return new_obj;
}

void Heap::collect() {
 // The from-space contains objects, and the to-space is empty now.

 address scanned = _to_space->bottom();
  // scavenge objects directly referenced by the root set
 foreach (Object** slot in ROOTS) {
   process_reference(slot);
 }

 // breadth-first scanning of object graph
 while (scanned < _to_space->top()) {
   Object* parent_obj = (Object*) scanned;
   foreach (Object** slot in parent_obj->obejct_fields()) {
     process_reference(slot);
     // note: _to_space->top() moves if any object is newly copied into to-space.
   }
   scanned += parent_obj->size();
 }

 // Now all live objects will have been evacuated into the to-space,
 // and we don't need the data in the from-space anymore.

 swap_spaces();
}

void Heap::process_reference(Object** slot) {
 Object* obj = *slot;
 if (obj != nullptr && _from_space->contains(obj)) {
   Object* new_obj = obj->is_forwarded() ? obj->forwardee() // copied
                                         : evacuate(obj);   // not copied (not marked)

   // fixup the slot to point to the new object
   *slot = new_obj;
 }
}

Semispace::Semispace(address bottom, address end) {
 _bottom = bottom;
 _top    = bottom;
 _end    = end;
```

```
}

address Semispace::contains(address obj) {
 return _bottom <= obj && obj < _top;
}

address Semispace::allocate(size_t size) {
 if (_top + size <= end) {
   address obj = _top;
   _top += size;
 } else {
   return nullptr;
 }
}

void Semispace::reset() {
 _top = _bottom;
}
```