如何实现一个垃圾收集器

初开
我本具足，超越范式。http://chukai.link

关注他

18 人赞同了该文章

原文：Do It Yourself (OpenJDK) Garbage Collector

作者：Aleksey，Red Hat，OpenJDK开发者，shipilev.net/

译者：初开，http://chukai.link

介绍

构建高可靠、高性能、可观察、可调试，有故障预测的运行时非常难，但造一个简单的轮子还是可以的。我们今天实现一个简单的垃圾收集器。Roman Kennke曾使用本文的早期版本，参加了FOSDEM 2019的演讲“20分钟构建GC”。虽然生产的实现很复杂，但原理是类似的。

为了改善阅读体验，请确保你了解过垃圾收集器的基本原理。本文虽然有一些基础知识和实现细节，但并不是入门的地方。可以阅读GC Handbook的第一章，或了解维基百科Tracing garbage collection。

1.构建块

现在实现新的GC要简单得多，很多其他GC实现的模块可以复用。

1.1 Epsilon GC

OpenJDK 11的JEP 318：“Epsilon：无操作垃圾收集器（实验版）”，是为不需要或禁止GC的场景提供最小实现，可以阅读JEP以获得更多信息。

从实现的角度来看，“垃圾收集器”这个术语用词不当，正确的术语是“自动内存管理器”，它实际上负责内存的分配和回收。

Epsilon GC仅实现了“分配”部分，所以我们可以在它上面来实现回收功能。

1.1.1 分配

Epsilon GC的核心是分配算法，分配到堆，或者分配到线程局部分配缓冲区（TLAB）。因为没有回收，所以它没怎么扩展TLAB。

1.1.2 屏障

一些垃圾收集器需要维护GC不变量，方法是在访问堆时强制执行GC屏障，所有并发收集器，包括分代收集器都是如此。Epsilon不需要屏障，但运行时和编译器仍然需要知道，哪怕是空实现，也要假装把它们打包起来。而OpenJDK 11的JEP-304：“垃圾收集接口”的定义使插入屏障更简单。

正因为Epsilon的GC屏障是空实现，所有的琐碎工作：加载，存储，CAS，数组复制，被委托给默认屏障。如果要开发一个仍然不需要障碍的GC，可以简单地重用Epsilon已有的。

1.1.3 JVM工具

实现GC的最后一项工作是能将它整合到JVM的各种监视工具中：MX bean、诊断命令等等。Epsilon 已经为我们处理了这个问题。

1.2 运行时和GC

1.2.1 Roots

垃圾收集器通常需要知道哪些部分包含对堆的引用，即GC Roots，包括线程堆栈和局部变量（JIT编译代码），本地类和类加载器，JNI句柄等等。GC Roots虽然听起来很复杂，但是在Hotspot中，每个VM子系统都会跟踪这些位置，我们可以用现有的GC实现。

1.2.2 对象遍历器

垃圾收集器需要遍历对象的引用，运行时已经提供了遍历器，无需自己实现。你将在下面的 `obj→oop_iterate` 中看到这些。

1.2.3 转发数据

垃圾收集器需要记录GC后的对象的新地址，称之为转发数据或转发地址，有几个地方存储地址值：

复用对象中的“markword”字段（Serial, Parallel等）。当GC暂停时，对对象字段的所有访问都停止，没有线程知道我们偷偷在“markword”中的存了数据。
维护单独的本地转发表（ZGC，C4等）。这将完全隔离GC与运行时和应用程序的其余部分，只有GC才会知道存在转发表。并发GC为了避免混淆通常采用这种方案。
在对象上添加另一个字段（Shenandoah等）。这是结合前两种方法，让运行时和应用程序与现有标头一起工作，同时保持转发数据。

1.2.4 标记数据

垃圾收集器需要在某处记录可达性标记。同样，有几种方法可以存储它：

复用对象中的“markword”（Serial, Parallel等）。同样，在GC暂停时，使用“markword”来编码“已标记”属性。然后，如果需要遍历所有活动对象，可以利用堆可解析性遍历。
维护单独的标记数据结构（G1，Shenandoah等）。这通常使用位图，将Java堆的每N个字节映射到1位标记位图。通常，Java对象按8个字节对齐，因此标记位图将64位堆映射到1位标记位图，占本机堆大小的1/64。这在扫描堆以获取对象实例（尤其是稀疏对象）时很管用：遍历位图通常比遍历分析堆快得多。
在引用本身（ZGC，C4，其他）中编码标记信息。这需要与应用协调以从引用中去除标记信息，或者做其他工作以保持正确性。换句话说，它需要GC屏障或更多的GC工作。

2.大计划

最容易在Epsilon上实现的GC算法是LISP2风格的标记-整理算法。该算法的原理在相关的维基百科条目或GC手册的第3.2章中给出。下面概述了算法的部分原理，可以阅读维基百科或GC手册进一步了解。

算法的关键是滑动GC：它通过将对象滑动到堆的开头来移动对象。它具有以下特征：

维护分配顺序。这对于控制内存布局非常有用，如果你是控制狂会很开心，但缺陷是，没法得到引用位置。
对象数是O(n)。但是，这种线性需要付出代价，每个GC循环需遍历堆4次。
它不需要任何额外的堆内存！无需保留活动对象的堆内存，因此即使是占有率99.9%的堆也可以操作。如果我们实现其他算法，例如标记-复制，我们需要重新划分堆的结构，并为复制预留一些空间，这超出了本文的范围。
通过一些技巧，当GC未启用时，可以不产生空间和时间开销。密集整理分配区域，这用来形容

Epsilon很贴切：它将持续从整理指针开始分配。但这也是它的缺点：堆开始时的一些死对象会频繁移动。
它不需要任何新的屏障来防止EpsilonBarrierSet受影响。

为简单起见，本文的GC实现将完全暂停，而非分代和单线程。在这种情况下，使用标记位图来存储标记数据，并重用标记字来存储转发数据。

3 实现GC的关键

一口气阅读完整实现太复杂了，我们逐个击破它。

3.1 准备

GC通常需要做一些准备工作，阅读以下注释，它们应该不难：

```
{
  GCTraceTime(Info, gc) time("Step 0: Prologue", NULL);
    //提交标记位图内存。 这样做有几个好处
   //循环之前：如果没有发生GC，则不会占用内存
   //在第一次调用时清理并映射未使用的位图部分
   //提交到零页，提高稀疏堆的性能。
  if (!os::commit_memory((char*)_bitmap_region.start(), _bitmap_region.byte_size(), fa
    log_warning(gc)("Could not commit native memory for marking bitmap, GC failed");
    return;
  }

    //我们不需要可解析堆来让算法工作，但我们想让线程放弃现有的TLAB。
  ensure_parsability(true);

  //通知运行时的各个部分我们将开始GC。
  CodeCache::gc_prologue();
  BiasedLocking::preserve_marks();

 //在标记期间将重置派生指针。
 //清除并激活表。
  DerivedPointerTable::clear();
}
```

由于我们使用标记位图来跟踪哪些对象可以访问，因此需要在使用之前清理。或者，在这种情况下，当我们追求从不占用资源直到GC循环命中，则需要首先将位图提交到内存。这带来了一些优势，至少在Linux上，大多数位图都映射到零页，特别是对于稀疏堆。GC完成后，线程需要放弃现有的TLAB并向GC请求新的TLAB。

运行时的某些部分，特别是处理Java堆引用的部分，会被GC过程修改，因此需要通知它们GC即将开始，请准备/保存某些状态。

3.2 标记

一旦准备好所有的组件，GC暂停标记就非常简单，这是许多GC实现的第一步。

```
{
  GCTraceTime(Info, gc) time("Step 1: Mark", NULL);

//标记堆栈和对象闭包（OopClosure）。通过对象闭包
//扫描对象的引用，标记它们，并推送新标记对象
//到堆栈以进行下一步处理。
  EpsilonMarkStack stack;
  EpsilonScanOopClosure cl(&stack, &_bitmap);

  // 在Roots上设置标记
  process_roots(&cl);
  stat_reachable_roots = stack.size();

  // 扫描堆的其余部分，终止是有保证的，最终会标记所有可访问对象。
  while (!stack.is_empty()) {
    oop obj = stack.pop();
    obj->oop_iterate(&cl);
    stat_reachable_heap++;
  }

    //标记完成后，就没有其他派生指针了。
  DerivedPointerTable::set_active(false);
}
```

就像图遍历一样：从最初的可到达顶点开始，遍历所有边，记录访问过的顶点，并执行此操作，直到走完未访问的顶点。在GC中，“顶点”是对象，“边”是它们之间的引用。

从技术上讲，可以使用递归来遍历，但对于任意图来说是一个糟糕的主意，比如有10亿个节点的话。因此，为了限制递归深度，我们使用记录堆栈来记录遍历后的对象。可达对象的初始值来自GC Roots。不要纠结process_roots，稍后会解释，先假设它遍历了所有引用。标记位图既用于跟踪标记（已访问对象的集合），也最终输出：所有可达对象的集合。

实际工作是由EpsilonScanOopClosure完成的，它会遍历给定对象的所有引用，如下：

```
class EpsilonScanOopClosure : public BasicOopIterateClosure {
private:
  EpsilonMarkStack* const _stack;
  MarkBitMap* const _bitmap;

  template <class T>
  void do_oop_work(T* p) {
    // p是指向oop的内存位置的指针，从中加载值，如果有必要，进行解压
    T o = RawAccess<>::oop_load(p);
    if (!CompressedOops::is_null(o)) {
      oop obj = CompressedOops::decode_not_null(o);

      //发现了对象。看看它是否已经标记。如果没有，
      //标记并将其推到标记堆栈上以进行进一步遍历。
      if (_bitmap->par_mark(obj)) {
         _stack->push(obj);
      }
    }
  }
};
```

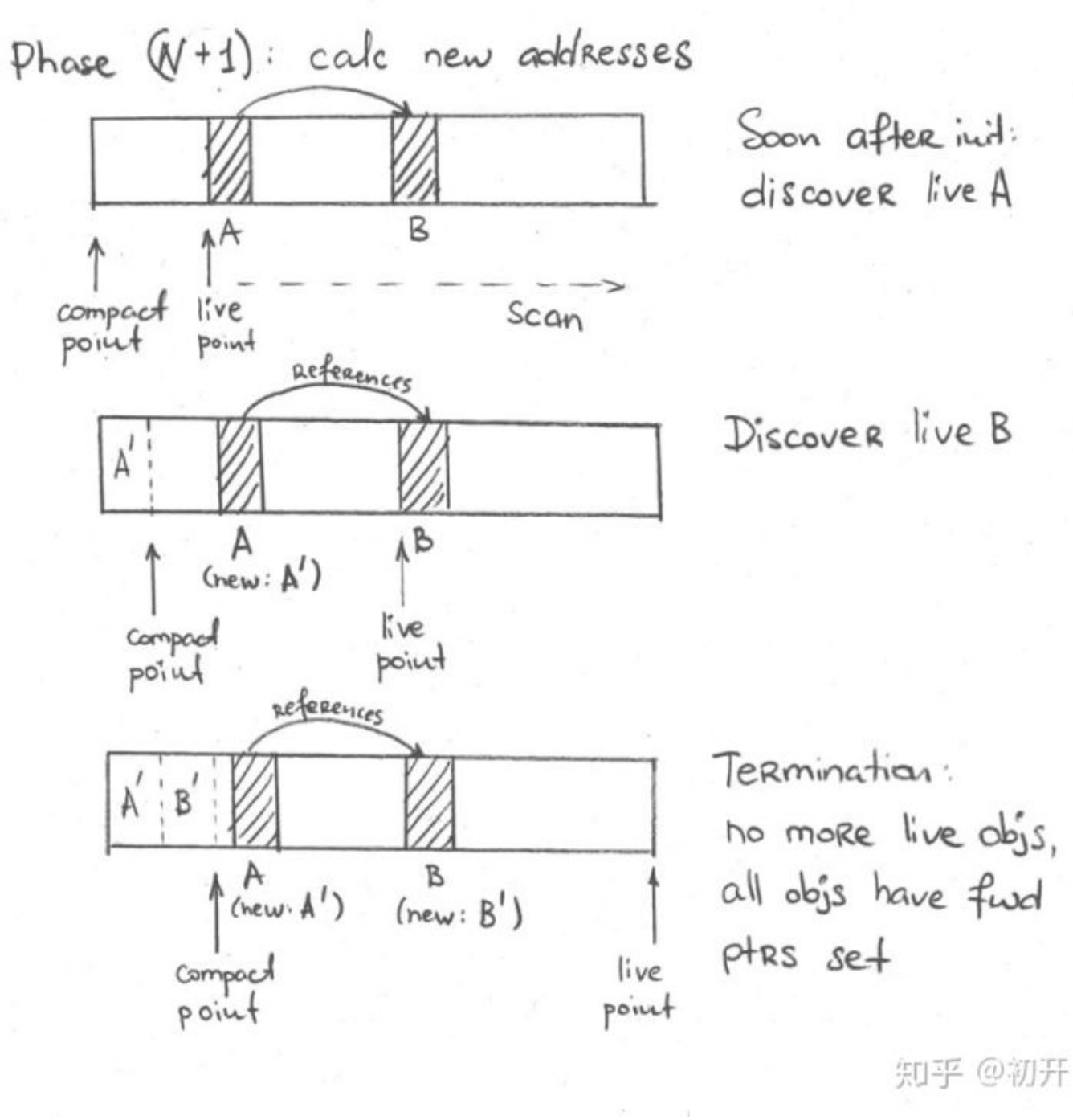
完成此步骤后，_bitmap包含活动对象设置的位。然后进一步遍历他们，如下：

```
//遍历标记位图，并在标记对象上调用它引用的对象闭包。
//走一个（非常稀疏的）可解析的堆，这要快得多，
//但占用位图的堆大小的1/64。
void EpsilonHeap::walk_bitmap(ObjectClosure* cl) {
   HeapWord* limit = _space->top();
   HeapWord* addr = _bitmap.get_next_marked_addr(_space->bottom(), limit);
   while (addr < limit) {
     oop obj = oop(addr);
     assert(_bitmap.is_marked(obj), "sanity");
```

```
        cl->do_object(obj);
        addr += 1;
        if (addr < limit) {
          addr = _bitmap.get_next_marked_addr(addr, limit);
        }
      }
  }
```

3.3 计算新位置

这部分也很简单。



计算新的:

```
/我们将用对象头中的markword存储转发信息（对象的新位置），其中一些markword需要谨慎保存。
//这是一个维护这些特殊markword列表的实用程序。
PreservedMarks preserved_marks;
//新分配空间的顶部。
HeapWord* new_top;

{
  GCTraceTime(Info, gc) time("Step 2: Calculate new locations", NULL);
```

```
  //遍历所有活动对象，计算新地址
  //并存储到markword。可选择保留一些已有的markword
    EpsilonCalcNewLocationObjectClosure cl(_space->bottom(), &preserved_marks);
    walk_bitmap(&cl);

   //在计算地址后，就有了新分配空间的头部指针。
   // 我们还不能使用它，因为有些“在堆中”的断言检查对象
    //在当前的“头部”。
    new_top = cl.compact_point();
    stat_preserved_marks = preserved_marks.size();
  }
```

这里唯一的缺陷是我们将新地址存储在Java对象的markword中，而这些markword可能已经有数据了，例如锁的信息。但这样的markword很少见，如果需要，可以单独存储，这是PreservedMarks做的事情。

实际EpsilonCalcNewLocationObjectClosure已经做了：

```
class EpsilonCalcNewLocationObjectClosure : public ObjectClosure {
private:
  HeapWord* _compact_point;
  PreservedMarks* const _preserved_marks;

public:
  EpsilonCalcNewLocationObjectClosure(HeapWord* start, PreservedMarks* pm) :
                                      _compact_point(start),
                                      _preserved_marks(pm) {}

  void do_object(oop obj) {
    //记录对象的新位置：当前的整理指针（compact point）的位置（见本节开头图）。
    //如果对象当前位置和整理位置相同（这一判断对密集前缀中的对象一直为true），跳过它。
    if ((HeapWord*)obj != _compact_point) {
      markOop mark = obj->mark_raw();
      if (mark->must_be_preserved(obj)) {
        _preserved_marks->push(obj, mark);
      }
      obj->forward_to(oop(_compact_point));
    }
    _compact_point += obj->size();
  }
```

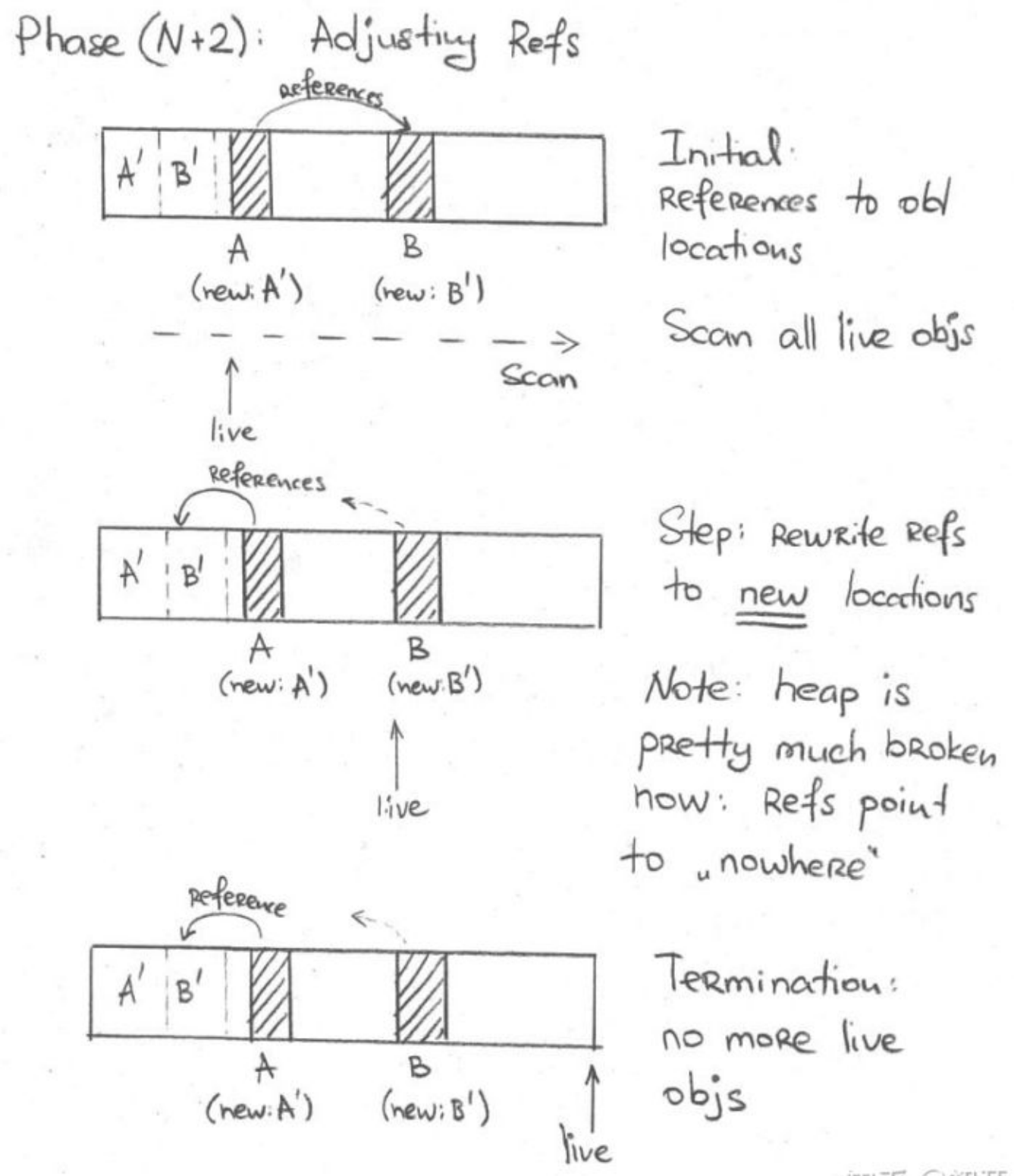
```
    HeapWord* compact_point() {
      return _compact_point;
    }
  };
```

forward_to是这里的关键部分：它将“转发地址”存储在对象的markword字中。后面将使用它。

> 译者注：密集前缀（dense prefix ）。由于上一次的的标记整理GC后，内存空间中，左边的对象密度很大，这部分被称为密集密集前缀。并且再一次整理时，这一部分也往往回收不了多少（因为大部分对象存活时间很短，而经过一次GC能活下来的，往往也能继续活着），如果剩余空间充足，为了提高性能往往会跳过左边密集部分的GC。

3.4 调整指针

根据算法，再次遍历堆并重写新位置的引用：




```
{
  GCTraceTime(Info, gc) time("Step 3: Adjust pointers", NULL);

//遍历所有活动对象及其引用字段，并设置新地址。新地址在转发数据markword字段中，先处理堆对象。
  EpsilonAdjustPointersObjectClosure cl;
  walk_bitmap(&cl);
```

```
  // 对于VM roots，执行相同的操作，更新引用。
  EpsilonAdjustPointersOopClosure cli;
  process_roots(&cli);

  // 最后，确保之前保留的标记知道对象移动了。
  preserved_marks.adjust_during_full_gc();
}
```

移动的对象有两组引用: 来自堆中的其他对象和GC Roots，需要同时更新。一些保留标记也记录了对象引用，需要让它们自我更新。PreservedMarks自己知道该怎么做，因为它得到了记录的对象引用。

对象闭包（OopClosureœ）现在有两种类型：一种接受对象并遍历其内容，另一种更新位置。这是有一个小的性能优化：如果一个对象没有转发地址，它就不会更新，我们可以节省操作。

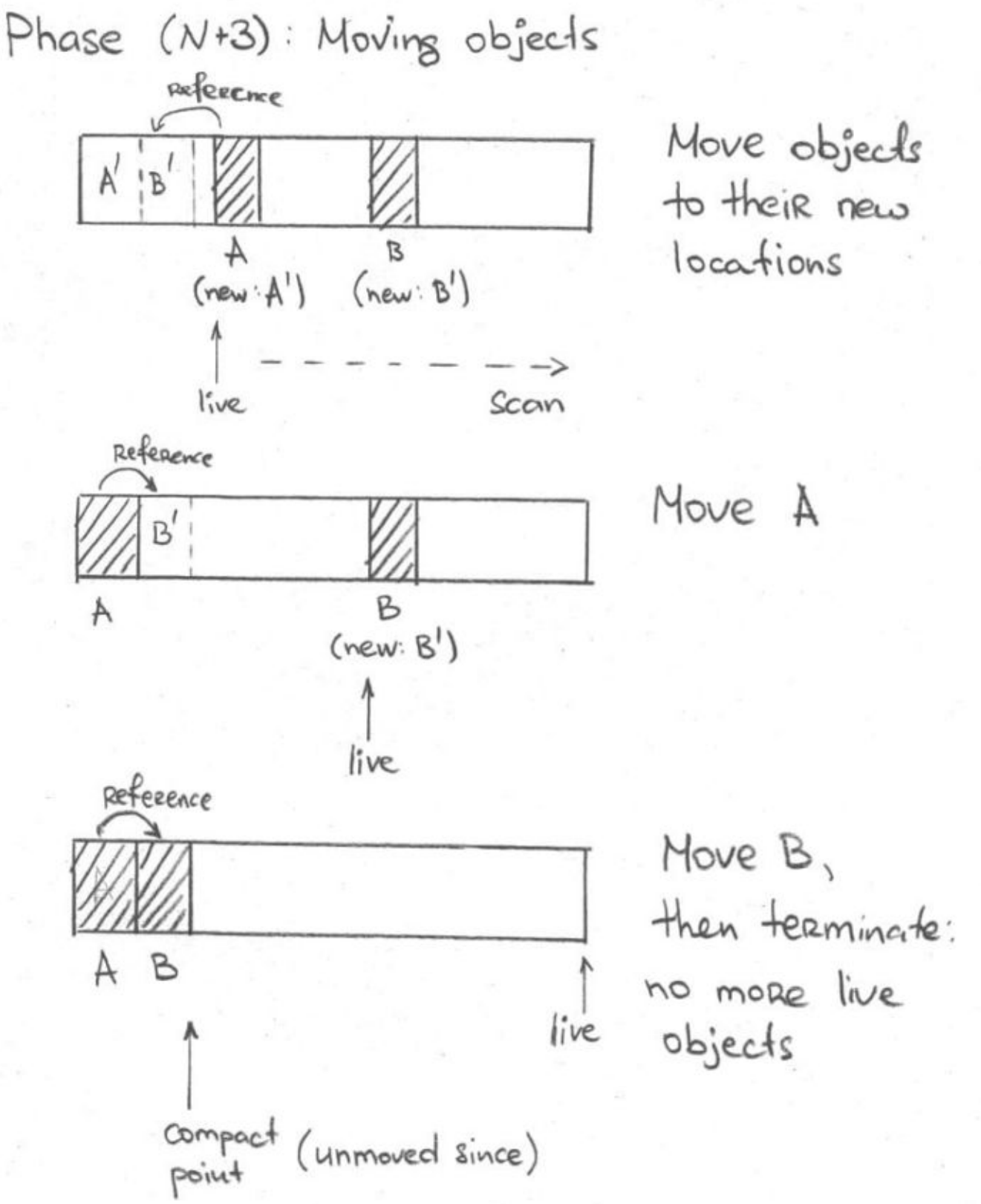
```
class EpsilonAdjustPointersOopClosure : public BasicOopIterateClosure {
private:
  template <class T>
  void do_oop_work(T* p) {
    // p是指向oop的内存位置的指针，从中加载值，如果有必要，进行解压
    T o = RawAccess<>::oop_load(p);
    if (!CompressedOops::is_null(o)) {
      oop obj = CompressedOops::decode_not_null(o);

      // 用转发地址重写当前对象的指针
      // 如果不需要则跳过
      if (obj->is_forwarded()) {
        oop fwd = obj->forwardee();
        assert(fwd != NULL, "just checking");
        RawAccess<>::oop_store(p, fwd);
      }
    }
  }
};

class EpsilonAdjustPointersObjectClosure : public ObjectClosure {
private:
  EpsilonAdjustPointersOopClosure _cl;
public:
  void do_object(oop obj) {
```

```
      // 遍历更新当前所有可用对象的引用
      obj->oop_iterate(&_cl);
    }
  };
```

完成此步骤后，堆已经被玩坏了：引用指向了“错误”位置，因为对象本身没移动。让我们纠正下！

3.5 移动对象

按照算法步骤，将对象移动到新位置：



再次遍历堆并用EpsilonMoveObjects移动所有活动对象：

```
{
  GCTraceTime(Info, gc) time("Step 4: Move objects", NULL);
  // 将所有活动对象移动到新位置。引用已在前一步调整了。
  EpsilonMoveObjects cl;
  walk_bitmap(&cl);
  stat_moved = cl.moved();
  //将所有对象移动到对应位置后，我们可以移回分配空间的头部指针了。
  _space->set_top(new_top);
}
```

完成此操作后，可以将可分配空间的起点移回到整理区末尾，让GC循环结束后从这里开始分配新内存。

请注意，滑动GC意味着覆盖了现有对象，但由于在一个方向上扫描，这意味着覆盖的对象直接被复制到正确位置。因此，EpsilonMoveObjects只是将转发的对象移动到新的位置：

```
class EpsilonMoveObjects : public ObjectClosure {
public:
  void do_object(oop obj) {
//如果存在转发地址，将对象复制到新位置。
//这是最后一步了，所以重置markword字段，清除转发地址。
    if (obj->is_forwarded()) {
      oop fwd = obj->forwardee();
      assert(fwd != NULL, "just checking");
      Copy::aligned_conjoint_words((HeapWord*)obj, (HeapWord*)fwd, obj->size());
      fwd->init_mark_raw();
    }
  }
};
```

3.6 润色

GC已经结束，堆内存基本一致了，再润色下：

```
{
  GCTraceTime(Info, gc) time("Step 5: Epilogue", NULL);

  // 恢复所有特殊的markword。
```

```
  preserved_marks.restore();

  // 通知运行时GC已完成
  DerivedPointerTable::update_pointers();
  BiasedLocking::restore_marks();
  CodeCache::gc_epilogue();
  JvmtiExport::gc_epilogue();

  // 不再需要标记位图了
  if (!os::uncommit_memory((char*)_bitmap_region.start(), _bitmap_region.byte_size()))
    log_warning(gc)("Could not uncommit native memory for marking bitmap");
  }

  // 如果有要求，将内存还给操作系统，在大堆上会花一点时间。
  if (EpsilonUncommit) {
    _virtual_space.shrink_by((_space->end() - new_top) * HeapWordSize);
    _space->set_end((HeapWord*)_virtual_space.high());
  }
}
```

通知运行时的其他部分进行GC后清理和修复；恢复之前保存的特殊markword；清除标记位图。如果有必要，可以将请求的内存重置到新的分配点，从而将内存还给给操作系统！

4.将GC挂接到VM

4.1 Roots 遍历

还记得需要从VM中获取特殊的可隐式访问的引用吗？这是要求VM中的每个特殊子系统将其隐藏的引用从其余对象中移除。当前Hotspot中Root目录如下所示：

```
void EpsilonHeap::do_roots(OopClosure* cl) {
  // 需要告诉运行时我们将用1个线程遍历Roots
  StrongRootsScope scope(1);

  // 需要为一些特殊的Root类型适配对象闭包。
  CLDToOopClosure clds(cl, ClassLoaderData::_claim_none);
  MarkingCodeBlobClosure blobs(cl, CodeBlobToOopClosure::FixRelocations);
```

```
  // 遍历运行时Root的所有部分,有些Root在遍历时需要锁。
  {
    MutexLockerEx lock(CodeCache_lock, Mutex::_no_safepoint_check_flag);
    CodeCache::blobs_do(&blobs);
  }
  {
    MutexLockerEx lock(ClassLoaderDataGraph_lock);
    ClassLoaderDataGraph::cld_do(&clds);
  }
  Universe::oops_do(cl);
  Management::oops_do(cl);
  JvmtiExport::oops_do(cl);
  JNIHandles::oops_do(cl);
  WeakProcessor::oops_do(cl);
  ObjectSynchronizer::oops_do(cl);
  SystemDictionary::oops_do(cl);
  Threads::possibly_parallel_oops_do(false, cl, &blobs);
}
```

有一些扩展模块可以并行遍历。对于我们的单线程GC，简单的遍历就足够了。

4.2 安全点和GC暂停

由于GC是全局的，需要VM执行全局的GC暂停。在Hotspot中，这是通过一个新VM_Operation调用我们的GC代码，并要求VM线程执行它：

```
// 在安全点下执行GC循环的VM操作
class VM_EpsilonCollect: public VM_Operation {
private:
  const GCCause::Cause _cause;
  EpsilonHeap* const _heap;
  static size_t _last_used;
public:
  VM_EpsilonCollect(GCCause::Cause cause) : VM_Operation(),
                                            _cause(cause),
                                            _heap(EpsilonHeap::heap()) {};

  VM_Operation::VMOp_Type type() const { return VMOp_EpsilonCollect; }
  const char* name()             const { return "Epsilon Collection"; }
```

```
  virtual bool doit_prologue() {
//在管理备份存储之前需要获取堆锁。
//这也自然地序列化GC请求，并允许我们合并来自多个线程的连续分配失败请求。
//不需要处理自上次完成GC以来没有分配的分配失败。
//在开始下一个之前等待分配的1%的堆
    // GC seems to resolve most races.
    Heap_lock->lock();
    size_t used = _heap->used();
    size_t capacity = _heap->capacity();
    size_t allocated = used > _last_used ? used - _last_used : 0;
    if (_cause != GCCause::_allocation_failure || allocated > capacity / 100) {
      return true;
    } else {
      Heap_lock->unlock();
      return false;
    }
  }

  virtual void doit() {
    _heap->entry_collect(_cause);
  }

  virtual void doit_epilogue() {
    _last_used = _heap->used();
    Heap_lock->unlock();
  }
};

size_t VM_EpsilonCollect::_last_used = 0;

void EpsilonHeap::vmentry_collect(GCCause::Cause cause) {
  VM_EpsilonCollect vmop(cause);
  VMThread::execute(&vmop);
}
```

当所有线程想要立即执行GC时，这可以解决一些性能敏感的资源竞争，这通常在内存耗尽时发生。

4.3 分配失败

虽然最好显式的请求进入GC，但我们还希望在没有内存的情况下GC能对堆耗尽做出反应。allocate_work使用这个包装器，在分配失败时执行GC，这足够处理大多数调用场景了：

HeapWord* EpsilonHeap::allocate_or_collect_work(size_t size) {HeapWord* res = allocate_work(size);if (res == NULL && EpsilonSlidingGC) {vmentry_collect(GCCause::_allocation_failure);res = allocate_work(size);}return res;}好了，完成了。

5.构建

该模块应该适用于OpenJDK。

```
$ hg clone https://hg.openjdk.java.net/jdk/jdk/ jdk-jdk
$ cd jdk-jdk
$ curl https://shipilev.net/jvm/diy-gc/webrev/jdk-jdk-epsilon.changeset | patch -p1
```

然后构建OpenJDK：

```
$ ./configure --with-debug-level=fastdebug
$ make images
```

运行：

```
$ build/linux-x86_64-server-fastdebug/images/jdk/bin/java -XX:+UnlockExperimentalVMOpt
openjdk version "13-internal" 2019-09-17
OpenJDK Runtime Environment (build 13-internal+0-adhoc.shade.jdk-jdk-epsilon)
OpenJDK 64-Bit Server VM (build 13-internal+0-adhoc.shade.jdk-jdk-epsilon, mixed mode,
```

6.测试

如何确保GC实现没有问题？好吧，有一些东西：

断言。Hotspot的代码断言确实很多，因此运行fastdebug构建通常会在GC崩溃时显示断言失败。
内部验证。当前模块实现了GC循环中的最后一步，它可以遍历所有活动对象并验证是否正常。在

GC循环结束时，暴露给运行时和应用程序之前可以捕获异常错误。
测试。如果代码实际上没有运行，则断言和验证是没用的。需要有能提前运行的单元和集成测试。

例如，你可以通过以下方式验证模块是否正常：

```
$ CONF=linux-x86_64-server-fastdebug make images run-test TEST=gc/epsilon/
Building targets 'images run-test' in configuration 'linux-x86_64-server-fastdebug'
Test selection 'gc/epsilon/', will run:
* jtreg:test/hotspot/jtreg/gc/epsilon

Running test 'jtreg:test/hotspot/jtreg/gc/epsilon'
Passed: gc/epsilon/TestAlwaysPretouch.java
Passed: gc/epsilon/TestAlignment.java
Passed: gc/epsilon/TestElasticTLAB.java
Passed: gc/epsilon/TestEpsilonEnabled.java
Passed: gc/epsilon/TestHelloWorld.java
Passed: gc/epsilon/TestLogTrace.java
Passed: gc/epsilon/TestDieDefault.java
Passed: gc/epsilon/TestDieWithOnError.java
Passed: gc/epsilon/TestMemoryPools.java
Passed: gc/epsilon/TestMaxTLAB.java
Passed: gc/epsilon/TestPrintHeapSteps.java
Passed: gc/epsilon/TestArraycopyCheckcast.java
Passed: gc/epsilon/TestClasses.java
Passed: gc/epsilon/TestUpdateCountersSteps.java
Passed: gc/epsilon/TestDieWithHeapDump.java
Passed: gc/epsilon/TestByteArrays.java
Passed: gc/epsilon/TestManyThreads.java
Passed: gc/epsilon/TestRefArrays.java
Passed: gc/epsilon/TestObjects.java
Passed: gc/epsilon/TestElasticTLABDecay.java
Passed: gc/epsilon/TestSlidingGC.java
Test results: passed: 21
TEST SUCCESS
```

怎么样？现在试试运行fastdebug并构建验证。对于构建中途不崩溃这一点，请保持心态良好。

7.开始你的表演

让我们带上spring-petclinic，用Apache Bench加载，跑一下吧！由于程序几乎没有实时数据，因此分代和非分代GC都差不多。运行-Xlog:gc -XX:+UnlockExperimentalVMOptions -XX:+UseEpsilonGC -XX:+EpsilonSlidingGC：

```
Heap: 20480M reserved, 20480M (100.00%) committed, 19497M (95.20%) used
GC(2) Step 0: Prologue 2.085ms
GC(2) Step 1: Mark 51.005ms
GC(2) Step 2: Calculate new locations 71.207ms
GC(2) Step 3: Adjust pointers 49.671ms
GC(2) Step 4: Move objects 22.839ms
GC(2) Step 5: Epilogue 1.008ms
GC(2) GC Stats: 70561 (8.63%) reachable from roots, 746676 (91.37%) reachable from hea
GC(2) Heap: 20480M reserved, 20480M (100.00%) committed, 37056K (0.18%) used
GC(2) Lisp2-style Mark-Compact (Allocation Failure) 20479M->36M(20480M) 197.940ms
```

200毫秒？对于第一次运行的单线程GC来说还不错！你可以看到四个主要阶段的时间相同。事实上，如果你用不同的占比和大小的堆，那么就会出现：更多对象会让GC会显著变慢（有太多活动对象并不好玩），更大的堆意会导致较慢的GC（即使在稀疏堆上遍历很长的距离也会对吞吐量产生影响）。

为了进行比较，用分代GC来测下。例如，-Xlog:gc -XX:+UseSerialGC主要是年轻代GC：

```
GC(46) Pause Young (Allocation Failure) 575M->39M(1943M) 2.603ms
GC(47) Pause Young (Allocation Failure) 575M->39M(1943M) 2.606ms
GC(48) Pause Young (Allocation Failure) 575M->39M(1943M) 2.747ms
GC(49) Pause Young (Allocation Failure) 575M->39M(1943M) 2.578ms
```

哇，2毫秒,这是因为大多数对象在年轻一代中已经死亡，并几乎没有任何GC工作要做。如果我们禁用分代-Xlog:gc -XX:+UseSerialGC并强制使用Full GC，那么我们会看到速度慢了下来：

GC(3) Pause Full (Allocation Failure) 16385M->34M(18432M) 1969.694msGC(4) Pause Full (Allocation Failure) 16385M->34M(18432M) 2261.405msGC(5) Pause Full (Allocation Failure) 16385M->34M(18432M) 2327.577msGC(6) Pause Full (Allocation Failure) 16385M->34M(18432M) 2328.976ms还可以用其他GC策略，留给读者练习。

8.改进

可以从这里开始尝试，但它会占用现有的OpenJDK GC的工作（包括测试！），所以仅作为练习。还可以改进：

处理引用实现。当前实现忽略了软/弱/虚引用。也忽略了可终结对象的存在。从性能的角度来看，这并不理想，但从正确性来看是安全的，因为代码会将这些引用视为始终可访问，会更新为常规引用。进阶实现则涉及将共享的ReferenceProcessor连接到标记代码，并在标记结束后标记/清除这些存活/死亡的引用。
实现类卸载和其他VM清理。当前的实现从不卸载类，也不会清除堆无法访问的对象的内部VM数据结构。实现这一点需要注意弱/强Root，默认情况下只标记强Root，然后在标记完成后查看是否仍然标记了弱Root，清理死亡的Root。
并行化。并行版本的最简单方法是将堆划分给每个线程，并在这些区域内执行相同的顺序整理。这将留下区域之间的空白，因此需要修改分配方式以知道存在多个空白区域。
处理密集前缀。通常情况下，正常堆最终会出现对象“沉积”层，如果将堆的前缀区指定为不移动，避免计算地址和移动对象，可以提升性能。不过仍需要标记并调整指针。
将密集前缀扩展到所有分代。通过GC屏障，可以看出密集前缀的哪些部分变动少，从而减少标记和指针调整。最后，它会变成“分代”，它会通过前缀执行young GC，有时也可以执行Full GC来整理前缀。
从GC手册中获取GC算法，并尝试实现。

总结

实现GC既有趣又有教育意义，可能非常适合大学的GC课程。

产品化是一个繁琐且耗时的过程，因此更容易的是切换并调整现有的收集器。如果这个功能进一步开发，最终变为现有的串行或并行GC一样，没这个必要。

注：

不要混淆TLAB和java.lang.ThreadLocal。从GC的角度来看，ThreadLocals仍然是普通对象，并且它们不会被GC清除，除非Java代码有处理。
可以先看看闭包。
对象的新旧位置可以重叠。例如，可以将100字节对象滑动8个字节。复制例程将确保正确复制内容，见Copy::aligned_*conjoint*_words。
从GC的角度来看，java.lang.ref.Reference.referent只是另一个Java字段，除非我们以特殊方式遍历堆，否则它是强可访问的。可终结对象有自己的合成实例，FinalReference会保留它们。
这个mark-compact的并行版本是Shenandoah（从OpenJDK 8开始）和G1（从OpenJDK 10开始，JEP 307: "Parallel Full GC for G1"）中Full GC的实现。