Garbage Collectors In Hotspot (Base OpenJDK 8)


feihui 关注
0.56 2019.10.29 06:32:50 字数 3,976 阅读 11,165

> JVM 的成功离不开其强大的垃圾收集功能，极大地避免内存泄漏（当然并不能完全避免，例如：例如将局部对象放入全局变量中）。那么今天我们就来了解下这其中的神奇之处。

我们先来看几个概念

image.png

Conservative GC VS Exact GC

Conservative GC -- **保守式** GC

简单来说，指的是“不能识别指针和非指针的 GC”。
优点：保守式GC的好处是相对来说实现简单些，而且可以方便的用在对GC没有特别支持的编程语言里提供自动内存管理功能。
缺点：

1. 会有部分对象本来应该已经死了，但有疑似指针指向它们，使它们逃过GC的收集。
2. 由于不知道疑似指针是否真的是指针，所以它们的值都不能改写；移动对象就意味着要修正指针。换言之，对象就不可移动了（一种办法可以在使用保守式GC的同时支持对象的移动，那就是增加一个间接层，不直接通过指针来实现引用，而是添加一层“句柄”（handle）在中间，所有引用先指到一个句柄表里，再从句柄表找到实际对象）。

Exact GC -- **精准式** GC

能精确地识别指针和非指针的“正确的根”(exact roots)来执行 GC 的
需要“语言处理程序的 支援”
优点：能精确处理垃圾
缺点：在性能和空间上都有所成本

1. 打标签
2. 只在制定位置存放对象指针
3. 从外部记录下类型信息，存成映射表。现在三种主流的高性能JVM实现，HotSpot、JRockit和J9都是这样做的。其中，HotSpot把这样的数据结构叫做OopMap，JRockit里叫做livemap，J9里叫做GC map。Apache Harmony的DRLVM也把它叫GCMap。

此外，还有一种半保守式 GC -- 栈上保守，堆内精准。

> 在HotSpot中，对象的类型信息里有记录自己的OopMap，记录了在该类型的对象内什么偏移量上是什么类型的数据。所以从对象开始向外的扫描可以是准确的；这些数据是在类加载过程中计算得到的。
> 每个被JIT编译过后的方法也会在一些特定的位置记录下OopMap，记录了执行到该方法的某条指令的时候，栈上和寄存器里哪些位置是引用。这样GC在扫描栈的时候就会查询这些OopMap就知道哪里是引用了。这些特定的位置主要在：
>
> 1. 循环的末尾
> 2. 方法临返回前 / 调用方法的call指令后

3. 可能抛异常的位置

这种位置被称为“安全点”（safepoint）。之所以要选择一些特定的位置来记录OopMap，是因为如果对每条指令（的位置）都记录OopMap的话，这些记录就会比较大，那么空间开销会显得不值得。选用一些比较关键的点来记录就能有效的缩小需要记录的数据量，但仍然能达到区分引用的目的。因为这样，HotSpot中GC不是在任意位置都可以进入，而只能在safepoint处进入。 而仍然在解释器中执行的方法则可以通过解释器里的功能自动生成出OopMap出来给GC用。

```
0x000000011857882e: je     0x00000001185788ad  ; OopMap{rdi=Oop rsi=Oop off=308}
                                                ;*goto
                                                ; - java.lang.String::hashCode@45 (line 1470)
```

OopMap is a structure that records where object references (OOPs) are located on the Java stack. Its primary purpose is to find GC roots on Java stacks and to update the references whenever objects are moved within the Heap.There are three kinds of OopMaps:

1. OopMaps for **interpreted methods**. They are computed lazily, i.e. when GC happens, by analyzing bytecode flow. The best reference is the source code (with lots of comments), see generateOopMap.cpp. InterpreterOopMaps are stored in OopMapCache.
2. OopMaps for **JIT-compiled methods**. They are generated during JIT-compilation and kept along with the compiled code so that VM can quickly find by instruction address the stack locations and the registers where the object references are held.
3. OopMaps for generated **shared runtime stubs**. These maps are constructed manually by the developers - authors of these runtime stubs.

During GC JVM walks through all thread stacks. Each stack is parsed as a stream of stack frames. The frames are either interpreted or compiled or stubs. Interpreted frames contain information about Java method and bci (bytecode index). OopMapCache helps to find an OopMap corresponding to the given method and bci. The method of a compiled frame is discovered by instruction address lookup.

Reference Counting VS Reference Tracing

Reference Counting — **引用计数**

优点: 实现简单,对象引用发生变更时更新计数器,当 counter 为 0 时即可释放
缺点: 不能解决循环依赖问题，即使对象已不再使用 (仍然使用这种方式的语言，会通过标志阴影区域 -- 即 counter 不为 0 的对象来进一步处理)

Reference Tracing — **引用追踪**

优点: 能解决循环以来的问题
缺点: 实现稍微复杂,需要从 ROOT 遍历对象,性能有较大的损耗

Minor GC vs Major GC vs Full GC

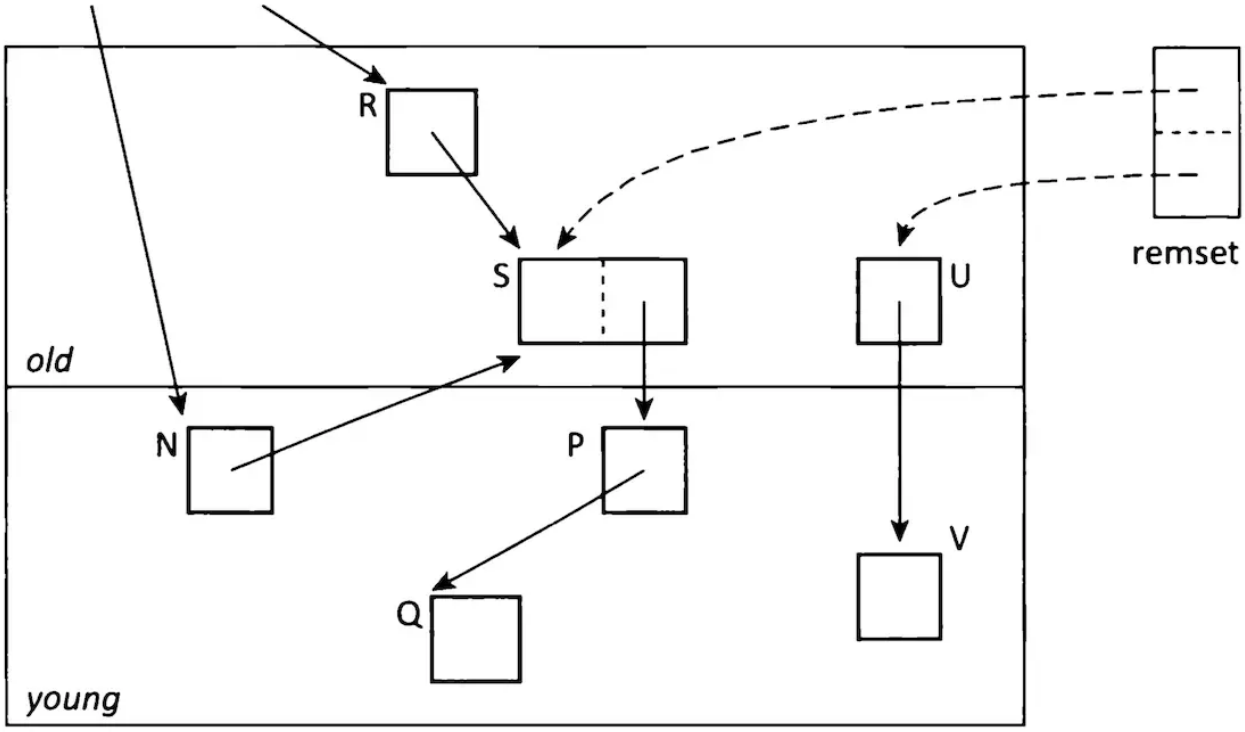
Minor GC is cleaning the Young space.
Major GC is cleaning the Old space.
Full GC is cleaning the entire Heap – both Young and Old spaces.

> But you must take note of that it is a bit more complex and confusing. To start with – many Major GC are triggered by Minor GC, so separating the two is impossible in many cases. On the other hand – modern garbage collection algorithms like G1 perform partial garbage cleaning so, again, using the term '**Mixed GC**' is only partially correct. **Instead of worrying about whether the GC is called Major or Full GC, you should focus on finding out whether the GC at hand stopped all the application threads or was able to progress concurrently with the application threads**.

Remembered Set

Generational garbage collection requires tracking of references from objects in older generations to objects in younger generations. In the example, the remembered set(remset below) records the location of any objects(or fields) that may container an inter-generational pointer of interest to the

garbage collector, in this case S and U. At the same time, generational collector needs a similar copy write barrier to detect any inter-generational references created by promotion.



pasted-image-2.png

图中的 remset **为什么不直接指向** young **对象呢？因为假如直接指向** young **对象，当** young **对象移动了，就需要遍历** old **来修改其中对象对** young **对象的引用。**

```
write_barrier(obj, field, new_obj){
    if(obj >= $old_start &&
       new_obj < $old_start &&
       obj.remembered == FALSE
    )
        $rs[$rs_index] = obj
        $rs_index++
        obj.remembered = TRUE
    *field = new_obj
}
```

Garbage Collection Roots:

1. Local variables
2. Active threads

3. Static fields
4. JNI references

Promotion

In generational garbage collection, objects will be move to an older generation after they survives several times in younger generation,

Floating Garbage

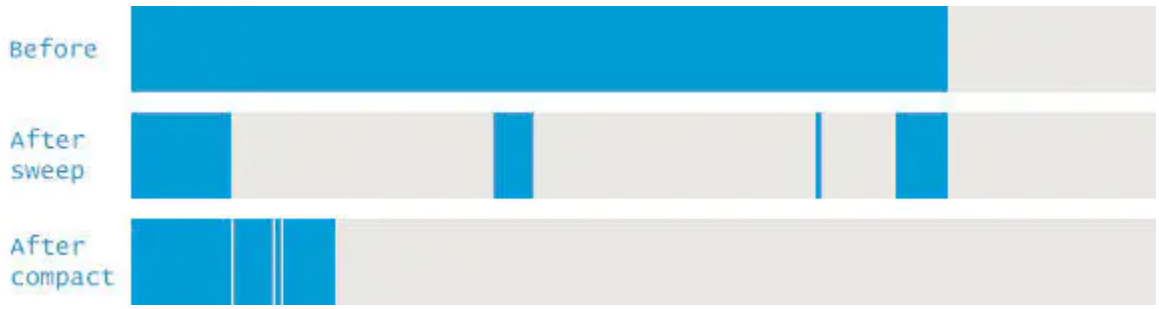
The concurrent collector, like all the other collectors in HotSpot, is a tracing collector that identifies at least all the reachable objects in the heap. In the parlance of Jones and Lins it is an incremental update collector. Because application threads and the garbage collector thread run concurrently during a major collection, objects that are traced by the garbage collector thread may subsequently become unreachable by the time collection finishes. Such unreachable objects that have not yet been reclaimed are referred to as floating garbage. The amount of floating garbage depends on the duration of the concurrent collection cycle and on the frequency of reference updates, also known asmutations, by the application. Furthermore, since the young generation and the tenured generation are collected independently, each acts a source of roots to the other. As a rough rule of thumb, try increasing the size of the tenured generation by 20% to account for the floating garbage. Floating garbage in the heap at the end of one concurrent collection cycle is collected during the next collection cycle.

Stop-The-World

All mutator threads need to be suspended at this moment. It is important for collector threads to do something correctly, such as marking alive object, copying alive objects and so on, but at the same time, application threads pause to process task any more and this really impact some interactive applications like web, so how to control and reduce pause time is very important for a GC algorithm.

Mark and Sweep/Copy/Compact

pasted-image.jpg



pasted-image.jpg

pasted-image.jpg

Mark — three-color / triple-color

Assign colors to the objects:

1. White: not yet visited
2. Gray: visited, but references are not scanned yet
3. Black: visited, and fully scanned

一般实现如下：一开始所有对象均为白色 (隐式)，将能从 Root 直接访问的对象标记为标灰入队(其属性引用还未处理)，依次取出队中元素处理其属性引用 -- 这个时候有两种操作：广度优先和深度优先，以广度优先为例，由于该对象已经完全被处理标黑，而其中访问到的属性对象若不为黑色则标灰入队，重复处理知道队列为空，最后就只剩下白色对象(不可达)和黑色对象(可达)。

我们再来看看并发标记，由于标记线程在标记的同时应用线程有可能改变对象之间的关系，会导致漏标对象(黑色对象指向白色对象)，因此我们需要一些措施来解决这个问题 -- 例如：Barrier，简单来说就说是在修改对象间的引用关系时记录下这种改变：

Algorithm 15.1: Grey mutator barriers.

(a) Steele [1975, 1976] barrier

```
1 atomic Write(src, i, ref):
2     src[i] ← ref
3     if isBlack(src)
4         if isWhite(ref)
5             revert(src)
```

(b) Boehm *et al* [1991] barrier

```
1 atomic Write(src, i, ref):
2     src[i] ← ref
3     if isBlack(src)
4         revert(src)
```

(c) Dijkstra *et al* [1976, 1978] barrier

```
1 atomic Write(src, i, ref):
2     src[i] ← ref
3     if isBlack(src)
```

Algorithm 15.2: Black mutator barriers.

(a) Baker [1978] barrier

```
1 atomic Read(src, i):
2     ref ← src[i]
3     if isGrey(src)
4         ref ← shade(ref)
5     return ref
```

(b) Appel *et al* [1988] barrier

```
1 atomic Read(src, i):
2     if isGrey(src)
3         scan(src)
4     return src[i]
```

(c) Abraham and Patel [1987] / Yuasa [1990] barrier

```
1 atomic Write(src, i, ref):
2     if isGrey(src)||isWhite(src)
3         shade(src[i])
```

pasted-image.png

Sweep -- free list / forward pointer

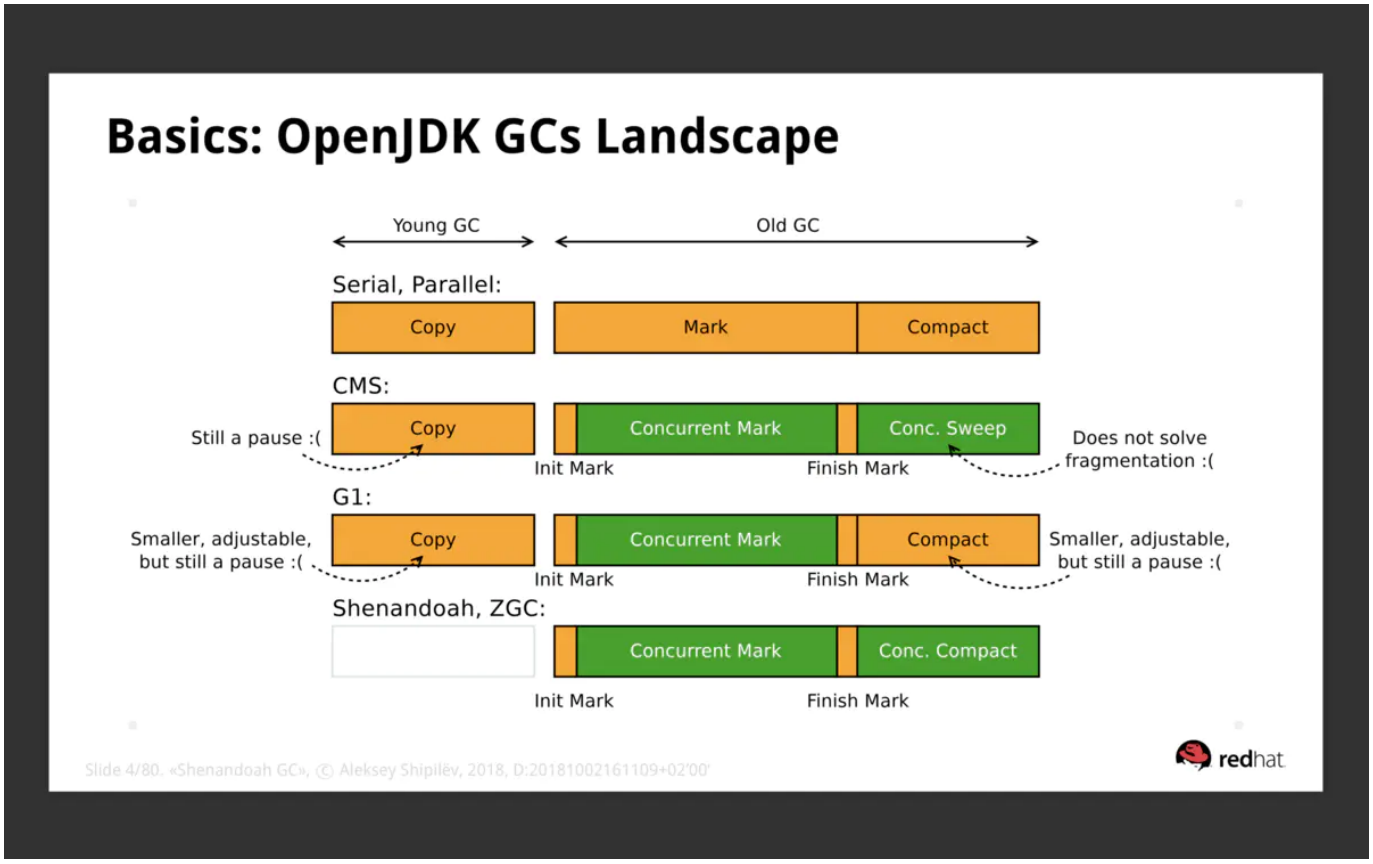
关于垃圾对象内存的处理，有两种方式：

1. 一种方式是释放垃圾对象，维护一个空闲内存列表，从该链表上分配和归还内存，这种方式实现非常简单，但显而易见的内存碎片问题(这是单纯用链表来管理内存怎都无法避免的一个问题，除非你能保证所有对象大小相同；而多级链表也只是在一定程度上缓解内存碎片)。
2. 一种方式是移动存活对象，这种方式需要而外的数据结构(例如：forward pointer) 来支持移动，并且还需要保留一定的空闲空间，在内存使用率上较低，但能有效得避免内存碎片。此外，有些优化在移动对象时也会尝试尽量把相关的对象放在一起，这样就可以利用空间局部性充分利用缓存。

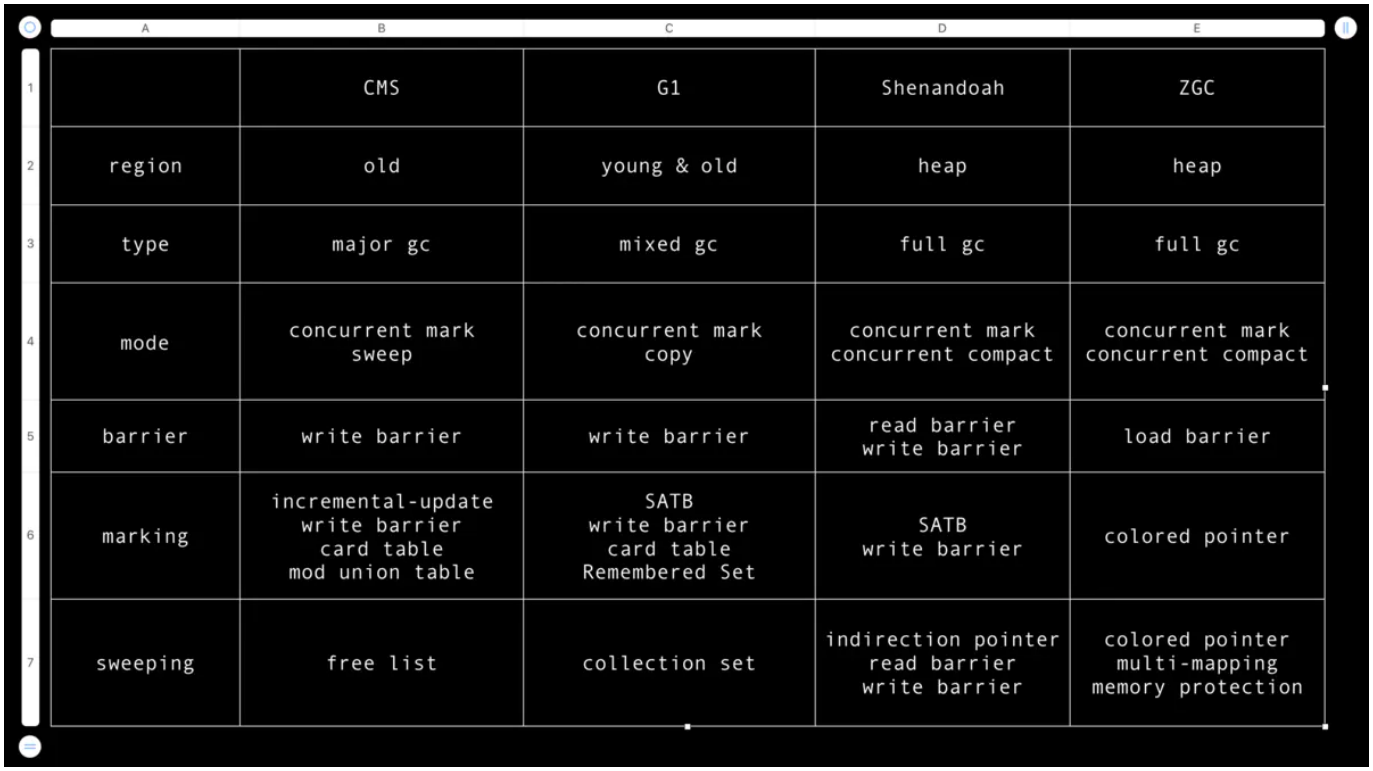
上面提到并发标记，这儿也简单提及下并发整理。同样地，由于整理线程和应用线程并发执行，那么应用线程有可能访问到移动过程的对象，那么这个时候同样需要一些措施来解决这个问题，例如：indication pointer + read / write / compare barrier，或者 memory protect 等。

> 垃圾收集是一个非常大的主题，推荐该领域中的一本神书《the garbage collection handbook》，非常值得一看。

我们再来看看 OpenJDK 中的 GC 实现：



pasted-image.png



	CMS	G1	Shenandoah	ZGC
region	old	young & old	heap	heap
type	major gc	mixed gc	full gc	full gc
mode	concurrent mark sweep	concurrent mark copy	concurrent mark concurrent compact	concurrent mark concurrent compact
barrier	write barrier	write barrier	read barrier write barrier	load barrier
marking	incremental-update write barrier card table mod union table	SATB write barrier card table Remembered Set	SATB write barrier	colored pointer
sweeping	free list	collection set	indirection pointer read barrier write barrier	colored pointer multi-mapping memory protection

image.png

下面介绍下当前 OpenJDK 中的 GC 或者是试验性 GC(简单聊下自己的总结，详情可参考列出的论文)：

CMS

> 参考论文：《A Generational Mostly-concurrent Garbage Collector》

相较于之前的 Parallel GC 以及更早的 GC，通过并发标记来减少停顿时间(标记对象所花费的时间往往大于移动对象的时间)，使用 Card Table 来减少 YGC 中的作为 GC Root 的老年代的范围，同时使用 Union Mod Table 来记录 YGC / OGC 同时发生是对 Card Table 的修改。

G1

> 参考论文：《Garbage-First Garbage Collection》

相对于 CMS，G1 的一大特是各代内存区域只是逻辑上连续，这有个非常大的好处在于 -- 对象从 from 到 to，可以不做移动，只需改变区域属性即可，每块区域还有 live map 记录当前区域存活对象数量，选择存活最少垃圾最大的区域进行收集，通过 remember set 记录区域之间的关系

以上两种 GC，在整理或者复制对象仍然是 STW 的，下面介绍的两种 GC，STW 只发生在 GC Root Mark 和 GC Root Compact，下面两种的 GC 有很多相似之处，例如：分区域 / 尚未分代 / remap / 低暂停等，最大的不同之处在于实现：

1. Shenandoah 需要对象上而外增加 indirection pointer，这就以为需要修改对象内部布局，同时为了实现并发整理，还需要 read barrier / write barrier / compare barrier
2. ZGC 则是利用 pointer 四位高位 bit 来标记对象当前所处状态，同时通过 load barrier 和 memory protect(实际上个人感觉这部分的功能就类似于 write barrier) 来完成并发整理。其 mutil-map 实现的 remap 和 Shenandoah 实现的 remap，有更强的灵活性，毕竟对象 unmap 即可。

Shenandoah

> 参考论文：《Shenandoah An open-source concurrent compacting garbage collector for OpenJDK》

ZGC

> 参考论文：《The Pauseless GC Algorithm》

最后我们再来简单聊下 Pauseless GC，即使是 ZGC，在 GC Root Marking 和 Compacting 也是 STW 的，那有没有完全不需要暂停的 GC 呢？两种方式：

1. GC Root Marking 和 Compacting，可以一部分一部分的处理应用线程，例如：有十个应用线程，我先处理其中五个(暂停)，剩下的仍然在运行；之后再反过来。这种实现方法个人看来是一种伪实现。
2. 通过 return barrier 或者 memory protect 协调应用线程和 GC 线程的内存访问，例如：GC 线程从栈底向上处理，应用线程只处理当前帧。

> 到这你会发现，并发处理下虽然暂停时间是少了，但这却是需要应用线程也帮忙做一些垃圾回收工作(通过各种 barrier / memory protect，这种协调下带来的总的工作量并不少)。某种程度上来说是把之前暂停时间拿来回收用了，个人觉得这种优势体现在强交互业务上，把一次暂停分摊到一段时间内。

最后我们来看下如何进入安全点(我们都知道，执行 GC 前要求所有应用线程有达到 safepoint 来进入 STW，安全点狭隘的说就是字节码执行完毕，广义的说就是进入一个确定状态)：

```
void VMThread::execute(VM_Operation* op) {
    ...... //
    if (op->evaluate_at_safepoint() && !SafepointSynchronize::is_at_safepoint()) {
      SafepointSynchronize::begin();
      op->evaluate();
      SafepointSynchronize::end();
    } else {
      op->evaluate();
    }
    ...... //
}
```

```
void SafepointSynchronize::begin() {
  ...... //
  // Begin the process of bringing the system to a safepoint.
  // Java threads can be in several different states and are
  // stopped by different mechanisms:
  //
  //  1. Running interpreted
  //     The interpeter dispatch table is changed to force it to
  //     check for a safepoint condition between bytecodes.
  //  2. Running in native code
  //     When returning from the native code, a Java thread must check
  //     the safepoint _state to see if we must block.  If the
  //     VM thread sees a Java thread in native, it does
  //     not wait for this thread to block.  The order of the memory
  //     writes and reads of both the safepoint state and the Java
  //     threads state is critical.  In order to guarantee that the
  //     memory writes are serialized with respect to each other,
  //     the VM thread issues a memory barrier instruction
  //     (on MP systems).  In order to avoid the overhead of issuing
  //     a memory barrier for each Java thread making native calls, each Java
  //     thread performs a write to a single memory page after changing
  //     the thread state.  The VM thread performs a sequence of
  //     mprotect OS calls which forces all previous writes from all
  //     Java threads to be serialized.  This is done in the
  //     os::serialize_thread_states() call.  This has proven to be
  //     much more efficient than executing a membar instruction
  //     on every call to native code.
  //  3. Running compiled Code
  //     Compiled code reads a global (Safepoint Polling) page that
  //     is set to fault if we are trying to get to a safepoint.
  //  4. Blocked
  //     A thread which is blocked will not be allowed to return from the
  //     block condition until the safepoint operation is complete.
  //  5. In VM or Transitioning between states
  //     If a Java thread is currently running in the VM or transitioning
  //     between states, the safepointing code will wait for the thread to
  //     block itself when it attempts transitions to a new state.
  //
  _state            = _synchronizing;
  OrderAccess::fence();

  // Flush all thread states to memory
  if (!UseMembar) {
    os::serialize_thread_states();
  }

  // Make interpreter safepoint aware
  Interpreter::notice_safepoints();

  if (UseCompilerSafepoints && DeferPollingPageLoopCount < 0) {
    // Make polling safepoint aware
```

```
    guarantee (PageArmed == 0, "invariant") ;
    PageArmed = 1 ;
    os::make_polling_page_unreadable();
  }

  // Consider using active_processor_count() ... but that call is expensive.
  int ncpus = os::processor_count() ;

#ifdef ASSERT
  for (JavaThread *cur = Threads::first(); cur != NULL; cur = cur->next()) {
    assert(cur->safepoint_state()->is_running(), "Illegal initial state");
    // Clear the visited flag to ensure that the critical counts are collected properly.
    cur->set_visited_for_critical_count(false);
  }
#endif // ASSERT

  if (SafepointTimeout)
    safepoint_limit_time = os::javaTimeNanos() + (jlong)SafepointTimeoutDelay * MICROUNITS;

  // Iterate through all threads until it have been determined how to stop them all at a safepoint
  unsigned int iterations = 0;
  int steps = 0 ;
  while(still_running > 0) {
    for (JavaThread *cur = Threads::first(); cur != NULL; cur = cur->next()) {
      assert(!cur->is_ConcurrentGC_thread(), "A concurrent GC thread is unexpectly being suspended");
      ThreadSafepointState *cur_state = cur->safepoint_state();
      if (cur_state->is_running()) {
        cur_state->examine_state_of_thread();
        if (!cur_state->is_running()) {
           still_running--;
           // consider adjusting steps downward:
           //   steps = 0
           //   steps -= NNN
           //   steps >>= 1
           //   steps = MIN(steps, 2000-100)
           //   if (iterations != 0) steps -= NNN
        }
        if (TraceSafepoint && Verbose) cur_state->print();
      }
    }

    if (PrintSafepointStatistics && iterations == 0) {
      begin_statistics(nof_threads, still_running);
    }

    if (still_running > 0) {
      // Check for if it takes to long
      if (SafepointTimeout && safepoint_limit_time < os::javaTimeNanos()) {
        print_safepoint_timeout(_spinning_timeout);
      }

      // Spin to avoid context switching.
      // There's a tension between allowing the mutators to run (and rendezvous)
      // vs spinning.  As the VM thread spins, wasting cycles, it consumes CPU that
```

```
// a mutator might otherwise use profitably to reach a safepoint.  Excessive
// spinning by the VM thread on a saturated system can increase rendezvous latency.
// Blocking or yielding incur their own penalties in the form of context switching
// and the resultant loss of $ residency.
//
// Further complicating matters is that yield() does not work as naively expected
// on many platforms -- yield() does not guarantee that any other ready threads
// will run.   As such we revert yield_all() after some number of iterations.
// Yield_all() is implemented as a short unconditional sleep on some platforms.
// Typical operating systems round a "short" sleep period up to 10 msecs, so sleeping
// can actually increase the time it takes the VM thread to detect that a system-wide
// stop-the-world safepoint has been reached.  In a pathological scenario such as that
// described in CR6415670 the VMthread may sleep just before the mutator(s) become safe.
// In that case the mutators will be stalled waiting for the safepoint to complete and the
// the VMthread will be sleeping, waiting for the mutators to rendezvous.  The VMthread
// will eventually wake up and detect that all mutators are safe, at which point
// we'll again make progress.
//
// Beware too that that the VMThread typically runs at elevated priority.
// Its default priority is higher than the default mutator priority.
// Obviously, this complicates spinning.
//
// Note too that on Windows XP SwitchThreadTo() has quite different behavior than Sleep(0).
// Sleep(0) will _not yield to lower priority threads, while SwitchThreadTo() will.
//
// See the comments in synchronizer.cpp for additional remarks on spinning.
//
// In the future we might:
// 1. Modify the safepoint scheme to avoid potentally unbounded spinning.
//    This is tricky as the path used by a thread exiting the JVM (say on
//    on JNI call-out) simply stores into its state field.  The burden
//    is placed on the VM thread, which must poll (spin).
// 2. Find something useful to do while spinning.  If the safepoint is GC-related
//    we might aggressively scan the stacks of threads that are already safe.
// 3. Use Solaris schedctl to examine the state of the still-running mutators.
//    If all the mutators are ONPROC there's no reason to sleep or yield.
// 4. YieldTo() any still-running mutators that are ready but OFFPROC.
// 5. Check system saturation.  If the system is not fully saturated then
//    simply spin and avoid sleep/yield.
// 6. As still-running mutators rendezvous they could unpark the sleeping
//    VMthread.  This works well for still-running mutators that become
//    safe.  The VMthread must still poll for mutators that call-out.
// 7. Drive the policy on time-since-begin instead of iterations.
// 8. Consider making the spin duration a function of the # of CPUs:
//    Spin = (((ncpus-1) * M) + K) + F(still_running)
//    Alternately, instead of counting iterations of the outer loop
//    we could count the # of threads visited in the inner loop, above.
// 9. On windows consider using the return value from SwitchThreadTo()
//    to drive subsequent spin/SwitchThreadTo()/Sleep(N) decisions.

if (UseCompilerSafepoints && int(iterations) == DeferPollingPageLoopCount) {
   guarantee (PageArmed == 0, "invariant") ;
   PageArmed = 1 ;
   os::make_polling_page_unreadable();
```

```
        }

        // Instead of (ncpus > 1) consider either (still_running < (ncpus + EPSILON)) or
        // ((still_running + _waiting_to_block - TryingToBlock)) < ncpus)
        ++steps ;
        if (ncpus > 1 && steps < SafepointSpinBeforeYield) {
          SpinPause() ;     // MP-Polite spin
        } else
        if (steps < DeferThrSuspendLoopCount) {
          os::NakedYield() ;
        } else {
          os::yield_all(steps) ;
          // Alternately, the VM thread could transiently depress its scheduling priority or
          // transiently increase the priority of the tardy mutator(s).
        }

        iterations ++ ;
      }
      assert(iterations < (uint)max_jint, "We have been iterating in the safepoint loop too long");
    }
    assert(still_running == 0, "sanity check");

    if (PrintSafepointStatistics) {
      update_statistics_on_spin_end();
    }

    // wait until all threads are stopped
    while (_waiting_to_block > 0) {
      if (TraceSafepoint) tty->print_cr("Waiting for %d thread(s) to block", _waiting_to_block);
      if (!SafepointTimeout || timeout_error_printed) {
        Safepoint_lock->wait(true);  // true, means with no safepoint checks
      } else {
        // Compute remaining time
        jlong remaining_time = safepoint_limit_time - os::javaTimeNanos();

        // If there is no remaining time, then there is an error
        if (remaining_time < 0 || Safepoint_lock->wait(true, remaining_time / MICROUNITS)) {
          print_safepoint_timeout(_blocking_timeout);
        }
      }
    }
    ...... //
  }
```

上面的注释已经做很好的说明(代码也是挺多的)，那我们来关注以下三点：

> os::serialize_thread_states();

```
// Serialize all thread state variables
void os::serialize_thread_states() {
  // On some platforms such as Solaris & Linux, the time duration of the page
  // permission restoration is observed to be much longer than expected  due to
  // scheduler starvation problem etc. To avoid the long synchronization
  // time and expensive page trap spinning, 'SerializePageLock' is used to block
  // the mutator thread if such case is encountered. See bug 6546278 for details.
  Thread::muxAcquire(&SerializePageLock, "serialize_thread_states");
  os::protect_memory((char *)os::get_memory_serialize_page(),
                     os::vm_page_size(), MEM_PROT_READ);
  os::protect_memory((char *)os::get_memory_serialize_page(),
                     os::vm_page_size(), MEM_PROT_RW);
  Thread::muxRelease(&SerializePageLock);
}

jint os::init_2(void)
{
  ...... //
  if (!UseMembar) {
    address mem_serialize_page = (address) ::mmap(NULL, Bsd::page_size(), PROT_READ | PROT_WRITE, MAP_P
    guarantee( mem_serialize_page != MAP_FAILED, "mmap Failed for memory serialize page");
    os::set_memory_serialize_page( mem_serialize_page );
  }
  ...... //
}
```

我们都知道每个线程都有自己的工作内存(往往体现在缓存上)，为了实现状态值的可见性当 read write 变成 read only，将缓存刷新到主存中。

> Interpreter::notice_safepoints();

```
void TemplateInterpreter::notice_safepoints() {
  if (!_notice_safepoints) {
    // switch to safepoint dispatch table
    _notice_safepoints = true;
    copy_table((address*)&_safept_table, (address*)&_active_table, sizeof(_active_table) / sizeof(addre
  }
}
```

我们先来了解下 Hotspot 中三个 dispatch：

```
DispatchTable TemplateInterpreter::_active_table;
DispatchTable TemplateInterpreter::_normal_table;
DispatchTable TemplateInterpreter::_safept_table;
```

```
void InterpreterMacroAssembler::dispatch_next(TosState state, int step) {
  // load next bytecode (load before advancing r13 to prevent AGI)
  load_unsigned_byte(rbx, Address(r13, step));
  // advance r13
  increment(r13, step);
  dispatch_base(state, Interpreter::dispatch_table(state));
}

static address*   dispatch_table(TosState state) { return _active_table.table_for(state); }
```

我们之前介绍过，在 TemplateInterpreter 中，每个字节码对应一段汇编代码，那个这些汇编代码的地址则存储在 table 数组中，对应的数组下表即为字节码值(先不讨论与栈顶缓存有关的 tos)，正常情况下 _active_table 为 _normal_table，当需要进入 safe point 的时候，_active_table 则改为 _safept_table。

```
void TemplateInterpreterGenerator::generate_all() {
  ...... //
  // Bytecodes
  set_entry_points_for_all_bytes();
  set_safepoints_for_all_bytes();
}

void TemplateInterpreterGenerator::set_safepoints_for_all_bytes() {
  for (int i = 0; i < DispatchTable::length; i++) {
    Bytecodes::Code code = (Bytecodes::Code)i;
    if (Bytecodes::is_defined(code)) Interpreter::_safept_table.set_entry(code, Interpreter::_safept_en
  }
}

void TemplateInterpreterGenerator::generate_all() {
  ...... //
  { CodeletMark cm(_masm, "safepoint entry points");
    Interpreter::_safept_entry =
      EntryPoint(
        generate_safept_entry_for(btos, CAST_FROM_FN_PTR(address, InterpreterRuntime::at_safepoint)),
        generate_safept_entry_for(ctos, CAST_FROM_FN_PTR(address, InterpreterRuntime::at_safepoint)),
        generate_safept_entry_for(stos, CAST_FROM_FN_PTR(address, InterpreterRuntime::at_safepoint)),
        generate_safept_entry_for(atos, CAST_FROM_FN_PTR(address, InterpreterRuntime::at_safepoint)),
        generate_safept_entry_for(itos, CAST_FROM_FN_PTR(address, InterpreterRuntime::at_safepoint)),
        generate_safept_entry_for(ltos, CAST_FROM_FN_PTR(address, InterpreterRuntime::at_safepoint)),
        generate_safept_entry_for(ftos, CAST_FROM_FN_PTR(address, InterpreterRuntime::at_safepoint)),
        generate_safept_entry_for(dtos, CAST_FROM_FN_PTR(address, InterpreterRuntime::at_safepoint)),
        generate_safept_entry_for(vtos, CAST_FROM_FN_PTR(address, InterpreterRuntime::at_safepoint))
      );
  }
  ...... //
}
```

```
address TemplateInterpreterGenerator::generate_safept_entry_for(
        TosState state,
        address runtime_entry) {
  address entry = __ pc();
  __ push(state);
  __ call_VM(noreg, runtime_entry);
  __ dispatch_via(vtos, Interpreter::_normal_table.table_for(vtos));
  return entry;
}

IRT_ENTRY(void, InterpreterRuntime::at_safepoint(JavaThread* thread))
  ...... //
IRT_END


#define IRT_ENTRY(result_type, header)                               \
  result_type header {                                               \
    ThreadInVMfromJava __tiv(thread);                                \
    VM_ENTRY_BASE(result_type, header, thread)                       \
    debug_only(VMEntryWrapper __vew;)


class ThreadInVMfromJava : public ThreadStateTransition {
  ...... //
  ~ThreadInVMfromJava()  {
    trans(_thread_in_vm, _thread_in_Java);
    ...... //
  }
};

void trans(JavaThreadState from, JavaThreadState to)  { transition(_thread, from, to); }

static inline void transition(JavaThread *thread, JavaThreadState from, JavaThreadState to) {
  ...... //
  if (SafepointSynchronize::do_call_back()) {
    SafepointSynchronize::block(thread);
  }
  ...... //
}
```

跟到最后使用的是 pthread_mutex_lock 来完成线程的 block

> os::make_polling_page_unreadable(); 这部分涉及两个知识点 -- signal handler / mmap

```
void os::make_polling_page_unreadable(void) {
  if( !guard_memory((char*)_polling_page, Bsd::page_size()) )
    fatal("Could not disable polling page");
};
```

```
bool os::guard_memory(char* addr, size_t size) {
  return bsd_mprotect(addr, size, PROT_NONE);
}

static bool bsd_mprotect(char* addr, size_t size, int prot) {
  // Bsd wants the mprotect address argument to be page aligned.
  char* bottom = (char*)align_size_down((intptr_t)addr, os::Bsd::page_size());

  // According to SUSv3, mprotect() should only be used with mappings
  // established by mmap(), and mmap() always maps whole pages. Unaligned
  // 'addr' likely indicates problem in the VM (e.g. trying to change
  // protection of malloc'ed or statically allocated memory). Check the
  // caller if you hit this assert.
  assert(addr == bottom, "sanity check");

  size = align_size_up(pointer_delta(addr, bottom, 1) + size, os::Bsd::page_size());
  return ::mprotect(bottom, size, prot) == 0;
}

jint os::init_2(void)
{
  // Allocate a single page and mark it as readable for safepoint polling
  address polling_page = (address) ::mmap(NULL, Bsd::page_size(), PROT_READ, MAP_PRIVATE|MAP_ANONYMOUS,
  guarantee( polling_page != MAP_FAILED, "os::init_2: failed to allocate polling page" );

  os::set_polling_page( polling_page );
  ...... //
  Bsd::install_signal_handlers();
  ...... //
}

void os::Bsd::install_signal_handlers() {
  ...... //
  set_signal_handler(SIGSEGV, true);
  ...... //
}

void os::Bsd::set_signal_handler(int sig, bool set_installed) {
  ...... //
  if (!set_installed) {
    sigAct.sa_flags = SA_SIGINFO|SA_RESTART;
  } else {
    sigAct.sa_sigaction = signalHandler;
    sigAct.sa_flags = SA_SIGINFO|SA_RESTART;
  }
  ...... //
}

void signalHandler(int sig, siginfo_t* info, void* uc) {
  ...... //
  JVM_handle_bsd_signal(sig, info, uc, true);
  ...... //
}
```

```
extern "C" JNIEXPORT int
JVM_handle_bsd_signal(int sig,
                      siginfo_t* info,
                      void* ucVoid,
                      int abort_if_unrecognized) {
  ...... //
  if ((sig == SIGSEGV || sig == SIGBUS) && os::is_poll_address((address)info->si_addr)) {
    stub = SharedRuntime::get_poll_stub(pc);
  }
  ...... //
  if (stub != NULL) {
    // save all thread context in case we need to restore it
    if (thread != NULL) thread->set_saved_exception_pc(pc);

    uc->context_pc = (intptr_t)stub;
    return true;
  }
  ...... //
}

address SharedRuntime::get_poll_stub(address pc) {
  ...... //
  if (at_poll_return) {
    assert(SharedRuntime::polling_page_return_handler_blob() != NULL,
           "polling page return stub not created yet");
    stub = SharedRuntime::polling_page_return_handler_blob()->entry_point();
  }
  ...... //
  return stub;
}

// Hotspot 初始化过程中被调用
void SharedRuntime::generate_stubs() {
  ...... //
  _polling_page_safepoint_handler_blob = generate_handler_blob(CAST_FROM_FN_PTR(address, SafepointSynch
  _polling_page_return_handler_blob    = generate_handler_blob(CAST_FROM_FN_PTR(address, SafepointSynch
  ...... //
}

void SafepointSynchronize::handle_polling_page_exception(JavaThread *thread) {
  ...... //
  state->handle_polling_page_exception();
}

void ThreadSafepointState::handle_polling_page_exception() {
  ...... //
  // This is a poll immediately before a return. The exception handling code
  // has already had the effect of causing the return to occur, so the execution
  // will continue immediately after the call. In addition, the oopmap at the
  // return point does not mark the return value as an oop (if it is), so
  // it needs a handle here to be updated.
  if( nm->is_at_poll_return(real_return_addr) ) {
    ...... //
```

```
      // Block the thread
      SafepointSynchronize::block(thread());
      ...... //
    }

    // This is a safepoint poll. Verify the return address and block.
    else {
      ...... //
      // Block the thread
      SafepointSynchronize::block(thread());
      ...... //
    }
  }
```

很亲切的 SafepointSynchronize::block 对吧！

```
0x000000010f895d4e: test %eax,-0xeb3bc54(%rip) # 0x0000000100d5a100; {poll}
```

理解了 SafepointSynchronize::begin 的逻辑，end 的逻辑就应该非常好理解。

知识扩展：

Signal Handler

The sigaction() system call is used to change the action taken by a process on receipt of a specific signal. (See signal(7) for an overview of signals.)

```
int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);

struct sigaction {
 void  (*sa_handler)(int);
 void  (*sa_sigaction)(int, siginfo_t *, void *);
 sigset_t  sa_mask;
 int sa_flags;
 void  (*sa_restorer)(void);
};
```

On some architectures a union is involved: do not assign to both sa_handler and sa_sigaction.

```
void handler(int sig, siginfo_t *info, void *ucontext) { … }
```

sig

The number of the signal that caused invocation of the handler.

info

A pointer to a siginfo_t, which is a structure containing further information about the signal, as described below.

ucontext

This is a pointer to a ucontext_t structure, cast to void *. The structure pointed to by this field contains signal context information that was saved on the user-space stack by the kernel; for details, see sigreturn(2). Further information about the ucontext_t structure can be found in getcontext(3). Commonly, the handler function doesn't make any use of the third argument.

mmap

mmap() creates a new mapping in the virtual address space of the calling process. The starting address for the new mapping is specified in addr. The length argument specifies the length of the mapping (which must be greater than 0)

```
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);

PROT_EXEC Pages may be executed.
PROT_READ Pages may be read.
PROT_WRITE Pages may be written.
PROT_NONE Pages may not be accessed.
```

```
int mprotect(void *addr, size_t len, int prot);
```

mprotect() changes the access protections for the calling process's memory pages containing any part of the address range in the interval [addr, addr + len -1]. addr must be aligned to a page boundary. If the calling process tries to access memory in a manner that violates the protections, then the kernel generates a SIGSEGV signal for the process.

```
PROT_NONE The memory cannot be accessed at all.
PROT_READ The memory can be read.
PROT_WRITE The memory can be modified.
PROT_EXEC The memory can be executed.
```