

# Java Hotspot G1 GC的一些关键技术

## (<https://tech.meituan.com/2016/09/23/g1.html>)



2016年09月23日

作者: 小亮

文章链接 (<https://tech.meituan.com/2016/09/23/g1.html>)

18946字



38分钟阅读

## 前言

G1 GC, 全称Garbage-First Garbage Collector, 通过-XX:+UseG1GC参数来启用, 作为体验版随着JDK 6u14版本面世, 在JDK 7u4版本发行时被正式推出, 相信熟悉JVM的同学们都不会对它感到陌生。在JDK 9中, G1被提议设置为默认垃圾收集器 (JEP 248)。在官网中, 是这样描述G1的:

> The Garbage-First (G1) collector is a server-style garbage collector, targeted for multi-processor machines with large memories. It meets garbage collection (GC) pause time goals with a high probability, while achieving high throughput. The G1 garbage collector is fully supported in Oracle JDK 7 update 4 and later releases. The G1 collector is designed for applications that:

- > \* Can operate concurrently with applications threads like the CMS collector.
- > \* Compact free space without lengthy GC induced pause times.
- > \* Need more predictable GC pause durations.
- > \* Do not want to sacrifice a lot of throughput performance.
- > \* Do not require a much larger Java heap.

从官网的描述中, 我们知道G1是一种服务器端的垃圾收集器, 应用在多处理器和大容量内存环境中, 在实现高吞吐量的同时, 尽可能的满足垃圾收集暂停时间的要求。它是专门针对以下应用场景设计的:

- \* 像CMS收集器一样, 能与应用程序线程并发执行。
- \* 整理空闲空间更快。
- \* 需要GC停顿时间更好预测。
- \* 不希望牺牲大量的吞吐性能。
- \* 不需要更大的Java Heap。

G1收集器的设计目标是取代CMS收集器, 它同CMS相比, 在以下方面表现的更出色:

- \* G1是一个有整理内存过程的垃圾收集器, 不会产生很多内存碎片。
- \* G1的Stop The World(STW)更可控, G1在停顿时间上添加了预测机制, 用户可以指定期望停顿时间。

有了以上的特性, 难怪有人说它是一款驾驭一切的垃圾收集器 (**G1: One Garbage Collector To Rule Them All** <http://www.infoq.com/articles/G1-One-Garbage-Collector-To-Rule-Them-All>)。

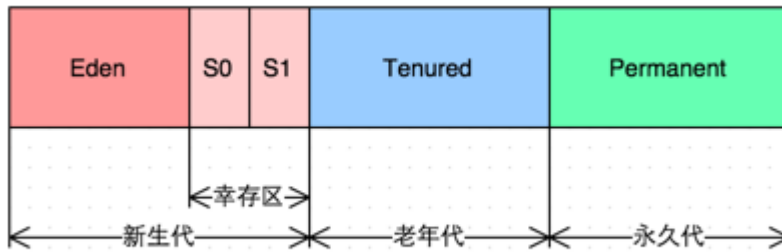
本文带大家来了解一下G1 GC的一些关键技术, 为能正确的使用它, 做好理论基础的铺垫。

## G1中几个重要概念

在G1的实现过程中, 引入了一些新的概念, 对于实现高吞吐、没有内存碎片、收集时间可控等功能起到了关键作用。下面我们就一起看一下G1中的这几个重要概念。

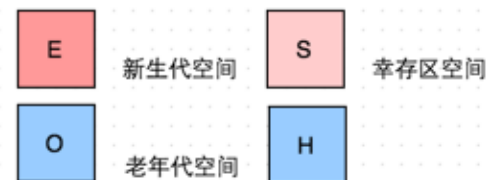
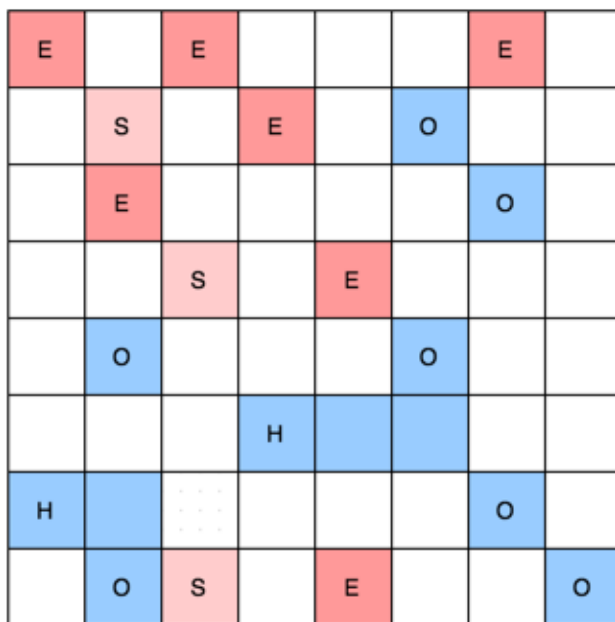
### Region

传统的GC收集器将连续的内存空间划分为新生代、老年代和永久代（JDK 8去除了永久代，引入了元空间Metaspace），这种划分的特点是各代的存储地址（逻辑地址，下同）是连续的。如下图所示：



传统GC内存布局

而G1的各代存储地址是不连续的，每一代都使用了n个不连续的大小相同的Region，每个Region占有一块连续的虚拟内存地址。如下图所示：



g1 GC内存布局

在上图中，我们注意到还有一些Region标明了H，它代表Humongous，这表示这些Region存储的是巨大对象（humongous object, H-obj），即大小大于等于region一半的对象。H-obj有如下几个特征：

- \* H-obj直接分配到了old gen，防止了反复拷贝移动。
- \* H-obj在global concurrent marking阶段的cleanup 和 full GC阶段回收。
- \* 在分配H-obj之前先检查是否超过 initiating heap occupancy percent和the marking threshold, 如果超过的话，就启动global concurrent marking，为的是提早回收，防止 evacuation failures 和 full GC。

为了减少连续H-objs分配对GC的影响，需要把大对象变为普通的对象，建议增大Region size。

一个Region的大小可以通过参数-XX:G1HeapRegionSize设定，取值范围从1M到32M，且是2的指数。如果不设定，那么G1会根据Heap大小自动决定。相关的设置代码如下：

```

// share/vm/gc_implementation/g1/heapRegion.cpp
// Minimum region size; we won't go lower than that.
// We might want to decrease this in the future, to deal with sma
ll
// heaps a bit more efficiently.
#define MIN_REGION_SIZE ( 1024 * 1024 )
// Maximum region size; we don't go higher than that. There's a g
ood
// reason for having an upper bound. We don't want regions to get
too
// large, otherwise cleanup's effectiveness would decrease as the
re
// will be fewer opportunities to find totally empty regions afte
r
// marking.
#define MAX_REGION_SIZE ( 32 * 1024 * 1024 )
// The automatic region size calculation will try to have around
this
// many regions in the heap (based on the min heap size).
#define TARGET_REGION_NUMBER 2048
void HeapRegion::setup_heap_region_size(size_t initial_heap_size,
size_t max_heap_size) {
    uintx region_size = G1HeapRegionSize;
    if (FLAG_IS_DEFAULT(G1HeapRegionSize)) {
        size_t average_heap_size = (initial_heap_size + max_heap_siz
e) / 2;
        region_size = MAX2(average_heap_size / TARGET_REGION_NUMBER,
(uintx) MIN_REGION_SIZE);
    }
    int region_size_log = log2_long((jlong) region_size);
    // Recalculate the region size to make sure it's a power of
    // 2. This means that region_size is the largest power of 2 tha
t's
    // <= what we've calculated so far.
    region_size = ((uintx)1 << region_size_log);
    // Now make sure that we don't go over or under our limits.
    if (region_size < MIN_REGION_SIZE) {
        region_size = MIN_REGION_SIZE;
    } else if (region_size > MAX_REGION_SIZE) {
        region_size = MAX_REGION_SIZE;
    }
}

```

## SATB

全称是Snapshot-At-The-Beginning，由字面理解，是GC开始时活着的对象的一个快照。它是通过Root Tracing得到的，作用是维持并发GC的正确性。那么它是怎么维持并发GC的正确性的呢？根据三色标记算法，我们知道对象存在三种状态：\* 白：对象没有被标记到，标记阶段结束后，会被当做垃圾回收掉。\* 灰：对象被标记了，但是它的field还没有被标记或标记完。\* 黑：对象被标记了，且它的所有field也被标记完了。

由于并发阶段的存在，Mutator和Garbage Collector线程同时对对象进行修改，就会出现白对象漏标的情况，这种情况发生的前提是：\* Mutator赋予一个黑对象该白对象的引用。\* Mutator删除了所有从灰对象到该白对象的直接或者间接引用。

对于第一个条件，在并发标记阶段，如果该白对象是new出来的，并没有被灰对象持有，那么它不会被漏标呢？Region中有两个top-at-mark-start（TAMS）指针，分别为prevTAMS和nextTAMS。在TAMS以上的对象是新分配的，这是一种隐式的标记。对于在GC时已经存在的白对象，如果它是活着的，它必然会被另一个对象引用，即条件二中的灰对象。如果灰对象到白对象的直接引用或者间接引用被替换了，或者删除了，白对象就会被漏标，从而导致被回收掉，这是非常严重的错误，所以SATB破坏了第二个条件。也就是说，一个对象的引用被替换时，可以通过write barrier 将旧引用记录下来。

```
// share/vm/gc_implementation/g1/g1SATBCardTableModRefBS.hpp
// This notes that we don't need to access any BarrierSet data
// structures, so this can be called from a static context.
template <class T> static void write_ref_field_pre_static(T* field, oop newVal) {
    T heap_oop = oopDesc::load_heap_oop(field);
    if (!oopDesc::is_null(heap_oop)) {
        enqueue(oopDesc::decode_heap_oop(heap_oop));
    }
}
// share/vm/gc_implementation/g1/g1SATBCardTableModRefBS.cpp
void G1SATBCardTableModRefBS::enqueue(oop pre_val) {
    // Nulls should have been already filtered.
    assert(pre_val->is_oop(true), "Error");
    if (!JavaThread::satb_mark_queue_set().is_active()) return;
    Thread* thr = Thread::current();
    if (thr->is_Java_thread()) {
        JavaThread* jt = (JavaThread*)thr;
        jt->satb_mark_queue().enqueue(pre_val);
    } else {
        MutexLockerEx x(Shared_SATB_Q_lock, Mutex::_no_safepoint_check_flag);
        JavaThread::satb_mark_queue_set().shared_satb_queue()->enqueue(pre_val);
    }
}
```

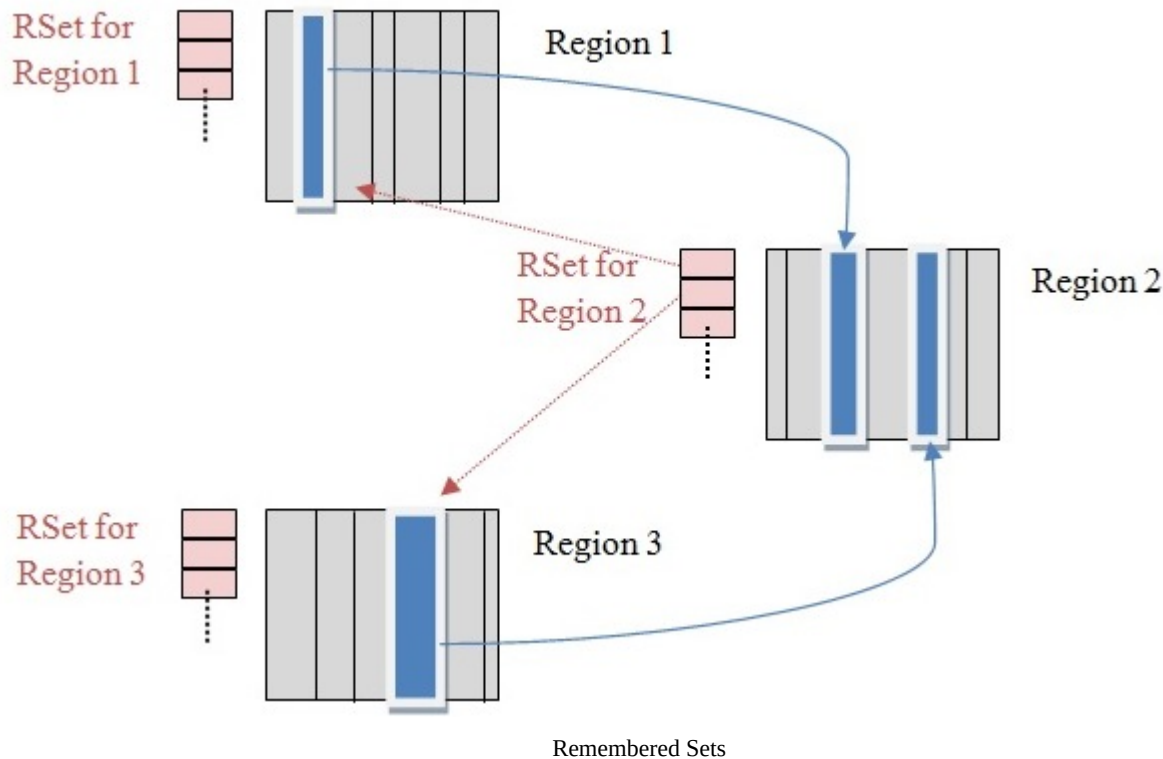
SATB也是有副作用的，如果被替换的白对象就是要被收集的垃圾，这次的标记会让它躲过GC，这就是float garbage。因为SATB的做法精度比较低，所以造成的float garbage也会比较多。

## RSet

全称是Remembered Set，是辅助GC过程的一种结构，典型的空间换时间工具，和Card Table有些类似。还有一种数据结构也是辅助GC的：Collection Set（CSet），它记录了GC要收集的Region集合，集合里的Region可以是任意年代的。在GC的时候，对于old->young和old->old的跨代对象引用，只要扫描对应的CSet中的RSet即可。逻辑上说每个Region都有一个RSet，RSet记录了其他

Region中的对象引用本Region中对象的关系，属于points-into结构（谁引用了我的对象）。而Card Table则是一种points-out（我引用了谁的对象）的结构，每个Card 覆盖一定范围的Heap（一般为512Bytes）。G1的RSet是在Card Table的基础上实现的：每个Region会记录下别的Region有指向自己的指针，并标记这些指针分别在哪些Card的范围内。这个RSet其实是一个Hash Table，Key是别的Region的起始地址，Value是一个集合，里面的元素是Card Table的Index。

下图表示了RSet、Card和Region的关系（[出处](http://www.infoq.com/articles/tuning-tips-G1-GC) <http://www.infoq.com/articles/tuning-tips-G1-GC>）：



上图中有三个Region，每个Region被分成了多个Card，在不同Region中的Card会相互引用，Region1中的Card中的对象引用了Region2中的Card中的对象，蓝色实线表示的就是points-out的关系，而在Region2的RSet中，记录了Region1的Card，即红色虚线表示的关系，这就是points-into。而维系RSet中的引用关系靠post-write barrier和Concurrent refinement threads来维护，操作伪代码如下（[出处](http://hllvm.group.iteye.com/group/topic/44381) <http://hllvm.group.iteye.com/group/topic/44381>）：

```
void oop_field_store(oop* field, oop new_value) {
    pre_write_barrier(field);           // pre-write barrier: for
    maintaining SATB invariant           // the actual store
    *field = new_value;                  // post-write barrier: fo
    post_write_barrier(field, new_value); // tracking cross-region reference
}
```

post-write barrier记录了跨Region的引用更新，更新日志缓冲区则记录了那些包含更新引用的Cards。一旦缓冲区满了，Post-write barrier就停止服务了，会由Concurrent refinement threads处理这些缓冲区日志。RSet究竟是怎么辅助GC的呢？在做YGC的时候，只需要选定young

generation region的RSet作为根集，这些RSet记录了old->young的跨代引用，避免了扫描整个old generation。而mixed gc的时候，old generation中记录了old->old的RSet，young->old的引用由扫描全部young generation region得到，这样也不用扫描全部old generation region。所以RSet的引入大大减少了GC的工作量。

## Pause Prediction Model

Pause Prediction Model 即停顿预测模型。它在G1中的作用是：>G1 uses a pause prediction model to meet a user-defined pause time target and selects the number of regions to collect based on the specified pause time target.

G1 GC是一个响应时间优先的GC算法，它与CMS最大的不同是，用户可以设定整个GC过程的期望停顿时间，参数-XX:MaxGCPauseMillis指定一个G1收集过程目标停顿时间，默认值200ms，不过它不是硬性条件，只是期望值。那么G1怎么满足用户的期望呢？就需要这个停顿预测模型了。G1根据这个模型统计计算出来的历史数据来预测本次收集需要选择的Region数量，从而尽量满足用户设定的目标停顿时间。停顿预测模型是以衰减标准偏差为理论基础实现的：

```
// share/vm/gc_implementation/g1/g1CollectorPolicy.hpp
double get_new_prediction(TruncatedSeq* seq) {
    return MAX2(seq->davg() + sigma() * seq->dsd(),
                seq->davg() * confidence_factor(seq->num()));
}
```

在这个预测计算公式中：davg表示衰减均值，sigma()返回一个系数，表示信赖度，dsd表示衰减标准偏差，confidence\_factor表示可信度相关系数。而方法的参数TruncateSeq，顾名思义，是一个截断的序列，它只跟踪了序列中的最新的n个元素。

在G1 GC过程中，每个可测量的步骤花费的时间都会记录到TruncateSeq（继承了AbsSeq）中，用来计算衰减均值、衰减变量，衰减标准偏差等：

```
// src/share/vm/utilities/numberSeq.cpp

void AbsSeq::add(double val) {
    if (_num == 0) {
        // if the sequence is empty, the davg is the same as the value
        _davg = val;
        // and the variance is 0
        _dvariance = 0.0;
    } else {
        // otherwise, calculate both
        _davg = (1.0 - _alpha) * val + _alpha * _davg;
        double diff = val - _davg;
        _dvariance = (1.0 - _alpha) * diff * diff + _alpha * _dvariance;
    }
}
```

比如要预测一次GC过程中，RSet的更新时间，这个操作主要是将Dirty Card加入到RSet中，具体原理参考前面的RSet。每个Dirty Card的时间花费通过`_cost_per_card_ms_seq`来记录，具体预测代码如下：

```
// share/vm/gc_implementation/g1/g1CollectorPolicy.hpp

double predict_rs_update_time_ms(size_t pending_cards) {
    return (double) pending_cards * predict_cost_per_card_ms();
}

double predict_cost_per_card_ms() {
    return get_new_prediction(_cost_per_card_ms_seq);
}
```

`get_new_prediction`就是我们开头说的方法，现在大家应该基本明白停顿预测模型的实现原理了。

## GC过程

讲完了一些基本概念，下面我们就来看看G1的GC过程是怎样的。

### G1 GC模式

G1提供了两种GC模式，Young GC和Mixed GC，两种都是完全Stop The World的。\* Young GC：选定所有年轻代里的Region。通过控制年轻代的region个数，即年轻代内存大小，来控制young GC的时间开销。\* Mixed GC：选定所有年轻代里的Region，外加根据global concurrent marking统计得出收集收益高的若干老年代Region。在用户指定的开销目标范围内尽可能选择收益高的老年代Region。

由上面的描述可知，Mixed GC不是full GC，它只能回收部分老年代的Region，如果mixed GC实在无法跟上程序分配内存的速度，导致老年代填满无法继续进行Mixed GC，就会使用serial old GC（full GC）来收集整个GC heap。所以我们可以知道，G1是不提供full GC的。

上文中，多次提到了global concurrent marking，它的执行过程类似CMS，但是不同的是，在G1 GC中，它主要是为Mixed GC提供标记服务的，并不是一次GC过程的一个必须环节。global concurrent marking的执行过程分为四个步骤：

- \* 初始标记（initial mark，STW）。它标记了从GC Root开始直接可达的对象。
- \* 并发标记（Concurrent Marking）。这个阶段从GC Root开始对heap中的对象标记，标记线程与应用程序线程并行执行，并且收集各个Region的存活对象信息。
- \* 最终标记（Remark，STW）。标记那些在并发标记阶段发生变化的对象，将被回收。
- \* 清除垃圾（Cleanup）。清除空Region（没有存活对象的），加入到free list。

第一阶段initial mark是共用了Young GC的暂停，这是因为他们可以复用root scan操作，所以可以说global concurrent marking是伴随Young GC而发生的。第四阶段Cleanup只是回收了没有存活对象的Region，所以它并不需要STW。

Young GC发生的时机大家都知道，那什么时候发生Mixed GC呢？其实是由一些参数控制着的，另外也控制着哪些老年代Region会被选入CSet。

- \* G1HeapWastePercent：在global concurrent marking结束之后，我们可以知道old gen regions中有多少空间要被回收，在每次YGC之后和再次发生Mixed GC之前，会检查垃圾占比是否达到此参数，只有达到了，下次才会发生Mixed GC。
- \* G1MixedGCLiveThresholdPercent：old generation region中的存活对象的占比，只有在此参数之下，才会被选入CSet。
- \* G1MixedGCCountTarget：一次global concurrent marking之后，最多执行Mixed GC的次数。
- \* G1OldCSetRegionThresholdPercent：一次Mixed GC中能被选入CSet的最多old generation region数量。

除了以上的参数，G1 GC相关的其他主要的参数有：

| 参数                                 | 含义  |
|------------------------------------|---|
| -XX:G1HeapRegionSize=n             | 设置Region大小，并非最终值  |
| -XX:MaxGCPauseMillis               | 设置G1收集过程目标时间，默认值200ms，不是硬性条件  |
| -XX:G1NewSizePercent               | 新生代最小值，默认值5%  |
| -XX:G1MaxNewSizePercent            | 新生代最大值，默认值60%   |
| -XX:ParallelGCThreads              | STW期间，并行GC线程数   |
| -XX:ConcGCThreads=n                | 并发标记阶段，并行执行的线程数   |
| -XX:InitiatingHeapOccupancyPercent | 设置触发标记周期的 Java 堆占用率阈值。默认值是45%。这里的java堆占比指的是non_young_capacity_bytes，包括old+humongous |

## GC日志

G1收集器的日志与其他收集器有很大不同，源于G1独立的体系架构和数据结构，下面这两段日志来源于美团点评的CRM系统线上生产环境。

## Young GC日志



我们先来看看Young GC的日志：

```

{Heap before GC invocations=12 (full 1):
  garbage-first heap   total 3145728K, used 336645K [0x000000007000
00000, 0x000000007c00000000, 0x000000007c00000000)
    region size 1024K, 172 young (176128K), 13 survivors (13312K)
  Metaspace           used 29944K, capacity 30196K, committed 30464K,
  reserved 1077248K
    class space       used 3391K, capacity 3480K, committed 3584K, res
erved 1048576K
2014-11-14T17:57:23.654+0800: 27.884: [GC pause (G1 Evacuation Pa
use) (young)
Desired survivor size 11534336 bytes, new threshold 15 (max 15)
- age   1:      5011600 bytes,      5011600 total
  27.884: [G1Ergonomics (CSet Construction) start choosing CSet, _
pending_cards: 1461, predicted base time: 35.25 ms, remaining tim
e: 64.75 ms, target pause time: 100.00 ms]
  27.884: [G1Ergonomics (CSet Construction) add young regions to C
Set, eden: 159 regions, survivors: 13 regions, predicted young re
gion time: 44.09 ms]
  27.884: [G1Ergonomics (CSet Construction) finish choosing CSet,
eden: 159 regions, survivors: 13 regions, old: 0 regions, predict
ed pause time: 79.34 ms, target pause time: 100.00 ms]
, 0.0158389 secs]
  [Parallel Time: 8.1 ms, GC Workers: 4]
    [GC Worker Start (ms): Min: 27884.5, Avg: 27884.5, Max: 278
84.5, Diff: 0.1]
    [Ext Root Scanning (ms): Min: 0.4, Avg: 0.8, Max: 1.2, Diff
: 0.8, Sum: 3.1]
    [Update RS (ms): Min: 0.0, Avg: 0.3, Max: 0.6, Diff: 0.6, S
um: 1.4]
    [Processed Buffers: Min: 0, Avg: 2.8, Max: 5, Diff: 5, S
um: 11]
    [Scan RS (ms): Min: 0.0, Avg: 0.1, Max: 0.1, Diff: 0.1, Sum
: 0.3]
    [Code Root Scanning (ms): Min: 0.0, Avg: 0.1, Max: 0.2, Dif
f: 0.2, Sum: 0.6]
    [Object Copy (ms): Min: 4.9, Avg: 5.1, Max: 5.2, Diff: 0.3,
Sum: 20.4]
    [Termination (ms): Min: 0.0, Avg: 0.0, Max: 0.0, Diff: 0.0,
Sum: 0.0]
    [GC Worker Other (ms): Min: 0.0, Avg: 0.4, Max: 1.3, Diff:
1.3, Sum: 1.4]
    [GC Worker Total (ms): Min: 6.4, Avg: 6.8, Max: 7.8, Diff:
1.4, Sum: 27.2]
    [GC Worker End (ms): Min: 27891.0, Avg: 27891.3, Max: 2789
2.3, Diff: 1.3]
    [Code Root Fixup: 0.5 ms]
    [Code Root Migration: 1.3 ms]
    [Code Root Purge: 0.0 ms]
    [Clear CT: 0.2 ms]
    [Other: 5.8 ms]
    [Choose CSet: 0.0 ms]
    [Ref Proc: 5.0 ms]
    [Ref Enq: 0.1 ms]

```

```

[Redirty Cards: 0.0 ms]
[Free CSet: 0.2 ms]
[Eden: 159.0M(159.0M)->0.0B(301.0M) Survivors: 13.0M->11.0M He
ap: 328.8M(3072.0M)->167.3M(3072.0M)]
Heap after GC invocations=13 (full 1):
  garbage-first heap   total 3145728K, used 171269K [0x000000007000
00000, 0x000000007c00000000, 0x000000007c00000000)
    region size 1024K, 11 young (11264K), 11 survivors (11264K)
  Metaspace           used 29944K, capacity 30196K, committed 30464K,
  reserved 1077248K
    class space       used 3391K, capacity 3480K, committed 3584K, res
erved 1048576K
}
[Times: user=0.05 sys=0.01, real=0.02 secs]

```

每个过程的作用如下：\* garbage-first heap total 3145728K, used 336645K

[0x00000000700000000, 0x000000007c0000000, 0x000000007c0000000) 这行表示使用了G1垃圾收集器, total heap 3145728K, 使用了336645K。\* region size 1024K, 172 young (176128K), 13 survivors (13312K) Region大小为1M, 青年代占用了172个（共176128K），幸存区占用了13个（共13312K）。\* Metaspace used 29944K, capacity 30196K, committed 30464K, reserved 1077248K class space used 3391K, capacity 3480K, committed 3584K, reserved 1048576K java 8的新特性, 去掉永久区, 添加了元数据区, 这块不是本文重点, 不再赘述。需要注意的是, 之所以有committed和reserved, 是因为没有设置MetaspaceSize=MaxMetaspaceSize。\* [GC pause (G1 Evacuation Pause) (young) GC原因, 新生代minor GC。\* [G1Ergonomics (CSet Construction) start choosing CSet, \_pending\_cards: 1461, predicted base time: 35.25 ms, remaining time: 64.75 ms, target pause time: 100.00 ms] 发生minor GC和full GC时, 所有相关region都是要回收的。而发生并发GC时, 会根据目标停顿时间动态选择部分垃圾对并多的Region回收, 这一步就是选择Region。\_pending\_cards是关于RSet的Card Table。predicted base time是预测的扫描card table时间。\* [G1Ergonomics (CSet Construction) add young regions to CSet, eden: 159 regions, survivors: 13 regions, predicted young region time: 44.09 ms] 这一步是添加Region到collection set, 新生代一共159个Region, 13个幸存区Region, 这也和之前的（172 young (176128K), 13 survivors (13312K)）吻合。预计收集时间是44.09 ms。\* [G1Ergonomics (CSet Construction) finish choosing CSet, eden: 159 regions, survivors: 13 regions, old: 0 regions, predicted pause time: 79.34 ms, target pause time: 100.00 ms] 这一步是对上面两步的总结。预计总收集时间79.34ms。\* [Parallel Time: 8.1 ms, GC Workers: 4] 由于收集过程是多线程并行（并发）进行, 这里是4个线程, 总共耗时8.1ms（wall clock time）\* [GC Worker Start (ms): Min: 27884.5, Avg: 27884.5, Max: 27884.5, Diff: 0.1] 收集线程开始的时间, 使用的是相对时间, Min是最早开始时间, Avg是平均开始时间, Max是最晚开始时间, Diff是Max-Min（此处的0.1貌似有问题）\* [Ext Root Scanning (ms): Min: 0.4, Avg: 0.8, Max: 1.2, Diff: 0.8, Sum: 3.1] 扫描Roots花费的时间, Sum表示total cpu time, 下同。\* [Update RS (ms): Min: 0.0, Avg: 0.3, Max: 0.6, Diff: 0.6, Sum: 1.4] [Processed Buffers: Min: 0, Avg: 2.8, Max: 5, Diff: 5, Sum: 11] Update RS (ms)是每

个线程花费在更新Remembered Set上的时间。 \* [Scan RS (ms): Min: 0.0, Avg: 0.1, Max: 0.1, Diff: 0.1, Sum: 0.3] 扫描CS中的region对应的RSet, 因为RSet是points-into, 所以这样实现避免了扫描old generation region, 但是会产生float garbage。 \* [Code Root Scanning (ms): Min: 0.0, Avg: 0.1, Max: 0.2, Diff: 0.2, Sum: 0.6] 扫描code root耗时。code root指的是经过JIT编译后的代码里, 引用了heap中的对象。引用关系保存在RSet中。 \* [Object Copy (ms): Min: 4.9, Avg: 5.1, Max: 5.2, Diff: 0.3, Sum: 20.4] 拷贝活的对象到新region的耗时。 \* [Termination (ms): Min: 0.0, Avg: 0.0, Max: 0.0, Diff: 0.0, Sum: 0.0] 线程结束, 在结束前, 它会检查其他线程是否还有未扫描完的引用, 如果有, 则“偷”过来, 完成后再申请结束, 这个时间是线程之前互相同步所花费的时间。 \* [GC Worker Other (ms): Min: 0.0, Avg: 0.4, Max: 1.3, Diff: 1.3, Sum: 1.4] 花费在其他工作上 (未列出) 的时间。 \* [GC Worker Total (ms): Min: 6.4, Avg: 6.8, Max: 7.8, Diff: 1.4, Sum: 27.2] 每个线程花费的时间和。 \* [GC Worker End (ms): Min: 27891.0, Avg: 27891.3, Max: 27892.3, Diff: 1.3] 每个线程结束的时间。 \* [Code Root Fixup: 0.5 ms] 用来将code root修正到正确的evacuate之后的对象位置所花费的时间。 \* [Code Root Migration: 1.3 ms] 更新code root 引用的耗时, code root中的引用因为对象的evacuation而需要更新。 \* [Code Root Purge: 0.0 ms] 清除code root的耗时, code root中的引用已经失效, 不再指向Region中的对象, 所以需要被清除。 \* [Clear CT: 0.2 ms] 清除card table的耗时。 \* [Other: 5.8 ms] [Choose CSet: 0.0 ms] [Ref Proc: 5.0 ms] [Ref Enq: 0.1 ms] [Redirty Cards: 0.0 ms] [Free CSet: 0.2 ms] 其他事项共耗时5.8ms, 其他事项包括选择CSet, 处理已用对象, 引用入ReferenceQueues, 释放CSet中的region到free list。 \* [Eden: 159.0M(159.0M)->0.0B(301.0M) Survivors: 13.0M->11.0M Heap: 328.8M(3072.0M)->167.3M(3072.0M)] 新生代清空了, 下次扩容到301MB。

## global concurrent marking 日志

对于global concurrent marking过程, 它的日志如下所示:

```

66955.252: [G1Ergonomics (Concurrent Cycles) request concurrent c
ycle initiation, reason: occupancy higher than threshold, occupan
cy: 1449132032 bytes, allocation request: 579608 bytes, threshol
d: 1449
551430 bytes (45.00 %), source: concurrent humongous allocation]
2014-12-10T11:13:09.532+0800: 66955.252: Application time: 2.5750
418 seconds
66955.259: [G1Ergonomics (Concurrent Cycles) request concurrent
cycle initiation, reason: requested by GC cause, GC cause: G1 Hu
mongous Allocation]
{Heap before GC invocations=1874 (full 4):
garbage-first heap    total 3145728K, used 1281786K [0x0000000700
000000, 0x00000007c0000000, 0x00000007c0000000)
region size 1024K, 171 young (175104K), 27 survivors (27648K)
Metaspace             used 116681K, capacity 137645K, committed 137984
K, reserved 1171456K
class space           used 13082K, capacity 16290K, committed 16384K,
reserved 1048576K
66955.259: [G1Ergonomics (Concurrent Cycles) initiate concurrent
cycle, reason: concurrent cycle initiation requested]
2014-12-10T11:13:09.539+0800: 66955.259: [GC pause (G1 Humongous
Allocation) (young) (initial-mark)
.....
2014-12-10T11:13:09.597+0800: 66955.317: [GC concurrent-root-regi
on-scan-start]
2014-12-10T11:13:09.597+0800: 66955.318: Total time for which app
lication threads were stopped: 0.0655753 seconds
2014-12-10T11:13:09.610+0800: 66955.330: Application time: 0.0127
071 seconds
2014-12-10T11:13:09.614+0800: 66955.335: Total time for which app
lication threads were stopped: 0.0043882 seconds
2014-12-10T11:13:09.625+0800: 66955.346: [GC concurrent-root-regi
on-scan-end, 0.0281351 secs]
2014-12-10T11:13:09.625+0800: 66955.346: [GC concurrent-mark-star
t]
2014-12-10T11:13:09.645+0800: 66955.365: Application time: 0.0306
801 seconds
2014-12-10T11:13:09.651+0800: 66955.371: Total time for which app
lication threads were stopped: 0.0061326 seconds
2014-12-10T11:13:10.212+0800: 66955.933: [GC concurrent-mark-end,
0.5871129 secs]
2014-12-10T11:13:10.212+0800: 66955.933: Application time: 0.5613
792 seconds
2014-12-10T11:13:10.215+0800: 66955.935: [GC remark 66955.936: [G
C ref-proc, 0.0235275 secs], 0.0320865 secs]
[Times: user=0.05 sys=0.00, real=0.03 secs]
2014-12-10T11:13:10.247+0800: 66955.968: Total time for which app
lication threads were stopped: 0.0350098 seconds
2014-12-10T11:13:10.248+0800: 66955.968: Application time: 0.0001
691 seconds
2014-12-10T11:13:10.250+0800: 66955.970: [GC cleanup 1178M->632M(
3072M), 0.0060632 secs]
[Times: user=0.02 sys=0.00, real=0.01 secs]

```

```
2014-12-10T11:13:10.256+0800: 66955.977: Total time for which app
lication threads were stopped: 0.0088462 seconds
2014-12-10T11:13:10.257+0800: 66955.977: [GC concurrent-cleanup-s
tart]
2014-12-10T11:13:10.259+0800: 66955.979: [GC concurrent-cleanup-e
nd, 0.0024743 secs
```

这次发生global concurrent marking的原因是：humongous allocation，上面提过在巨大对象分配之前，会检测到old generation 使用占比是否超过了 initiating heap occupancy percent (45%)，因为  $1449132032(\text{used}) + 579608(\text{allocation request}) > 1449551430(\text{threshold})$ ，所以触发了本次 global concurrent marking。对于具体执行过程，上面的表格已经详细讲解了。值得注意的是上文中所说的initial mark往往伴随着一次YGC，在日志中也有体现：GC pause (G1 Humongous Allocation) (young) (initial-mark)。

## 后记

因为篇幅的关系，也受限于能力水平，本文只是简单介绍了G1 GC的基本原理，很多细节没有涉及到，所以说只能算是为研究和使用的同学打开了一扇门。一个日本人专门写了一本书《**彻底解剖「G1GC」 アルゴリズム**》(<http://tatsu-zine.com/books/g1gc>) 详细的介绍了G1 GC，这本书也被作者放到了GitHub上，详见参考文献5。另外，莫枢在这方面也研究的比较多，读者可以去 **高级语言虚拟机论坛** (<http://hllvm.group.iteye.com>) 向他请教，本文的很多内容也是我在此论坛上请教过后整理的。总而言之，G1是一款非常优秀的垃圾收集器，尽管还有些不完美（预测模型还不够智能），但是希望有更多的同学来使用它，研究它，提出好的建议，让它变的更加完善。

## 参考文献

- **Getting Started with the G1 Garbage Collector** (<http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/G1GettingStarted/index.html>)
- **请教G1算法的原理** (<http://hllvm.group.iteye.com/group/topic/44381>)
- **关于incremental update与SATB的一点理解** (<http://hllvm.group.iteye.com/group/topic/44529>)
- **Tips for Tuning the Garbage First Garbage Collector** (<http://www.infoq.com/articles/tuning-tips-G1-GC>)
- **g1gc-impl-book** (<https://github.com/authorNari/g1gc-impl-book>)
- **垃圾优先型垃圾回收器调优** (<http://www.oracle.com/technetwork/cn/articles/java/g1gc-1984535-zhs.html>)
- **Understanding G1 GC Logs** ([https://blogs.oracle.com/poonam/entry/understanding\\_g1\\_gc\\_logs](https://blogs.oracle.com/poonam/entry/understanding_g1_gc_logs))
- **G1: One Garbage Collector To Rule Them All** (<http://www.infoq.com/articles/G1-One-Garbage-Collector-To-Rule-Them-All>)