轻量级 Java 对象剖析机制

设计文档 v1.3

作者：张显龙

指导教师：史晓华

北京航空航天大学计算机学院

2020 年 3 月 2 日

目录

1. 相关背景及设计思路

1.1. 相关背景

一些 Java 对象在逻辑上有着有限的生命周期。当这些对象所要做的事情完成了，我们希望他们会被回收掉。但是如果有一系列对象因为程序员疏忽持有了某个对象的引用，那么在我们期待这个对象生命周期结束的时候被收回的时候，它是不会被回收的。在这种情况下它还会持续占用内存直到所有被持有的引用释放，这就造成了一种隐式的内存泄漏。当这种情况持续发生时，我们的内存会很快被消耗殆尽。因此我们需要一个轻量级的机制去判断对象的“冷热”情况。同时，由于在某些应用场景下带宽及运算能力受限，我们不得不对收集到的信息在线分析，因此我们设计了一个低负载的在线 Java 对象生存周期剖析工具。

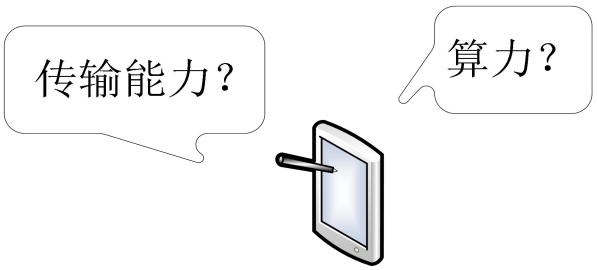

图 1.1 需要轻量级的在线机制

1.2. 设计思路

要实现一个在线的低负载的 Java 对象剖析工具，我们首先要明确我们需要哪些信息。根据对“冷”对象产生原因的分析，我们可以把“冷”对象定义为：经过一段时间未被使用且未被垃圾回收机制回收的对象。有了“冷”对象的定义，那么我们可以显然知道了我们要设计剖析工具需要的信息主要为对象的使用情况与时间戳。

然而，如果对 object 的布局进行修改，则会对系统造成巨大的负载，并且要修改的系统源码较多，因此我们需要通过其他数据结构来维护该信息。观察 ART 中 GC 的实现中有位图来记录一些对象生存信息，我们可以借助这种思想使用位图，来描述对象访问信息，位图用 1bit 来描述堆上的对应 64bit，因此空

间压缩比为 1：64，约不到堆整体的 2%，当对象产生访问信息时，我们通过插桩代码对位图对应位置进行标志。之后，我们再定义时间：我们将设置一个可配置的 GC_K，表示该机制通过判断在 GC_K 之间是否发生了访问信息。若未发生访问信息，则在根集可达性分析时判断该对象已经“冷”掉，进行相关处理。同时为了维护对象与分配点的信息，我们需要加入一个 hashtable 进行对象与分配点之间的关系维护。在设计中，我们考虑设置一个**对象大小阈值**，根据对象大小来判断一个对象是否需要被检测。由于实际应用环境中，受关注的 Java 内存泄漏对象基本都是大对象（例如大于 1K 的对象），因此这个可设定**对象大小阈值对工具的性能和实用性有较大的提升。**在整个机制的实现中，我们在所有 jitted code 针对所有对象都插桩了记录访问代码，但在 runtime 端的监测则根据对象大小来决定是否维护相关信息，这样实现的好处在于，**调整监测策略改变阈值时不需要重新编译被监控程序**，假设在实际应用中我们发现 profiler 对系统造成的负载较大，还可以自适应的调整对象阈值来缓解负载。

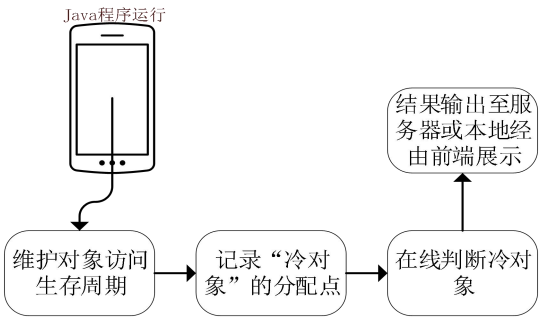

图 1.2 系统示意图

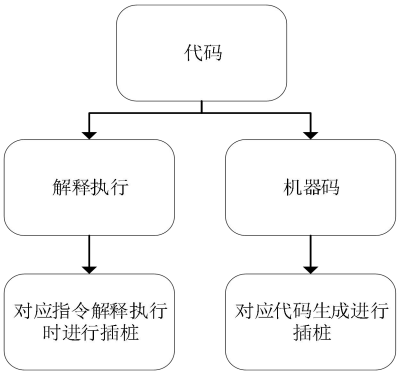

图 1.3 GC 机制与代码插桩

Android 自 7.0 开始，采用 AOT+JIT 的混合编译形式，通过观察系统代码，发现其共用一个代码生成器，因此要实现对于对象访问标记代码插桩，我们只需

要插桩在解释器部分对应的 bytecode 和代码生成器中对应指令的代码生成即可。通过插桩访问代码，实现了对于位图的标记。并且我们打算支持对于监测对象大小的配置选项，使得系统更灵活。

2. 详细设计

2.1. 相关开发环境

开发系统：Ubuntu-18.04

测试设备：Pixel 骁龙 821 处理器 4GB RAM

Android 版本：android-10.0.0_r2

build target： aosp_sailfish-userdebug

target build type：release

2.2. 对 Android 系统的修改

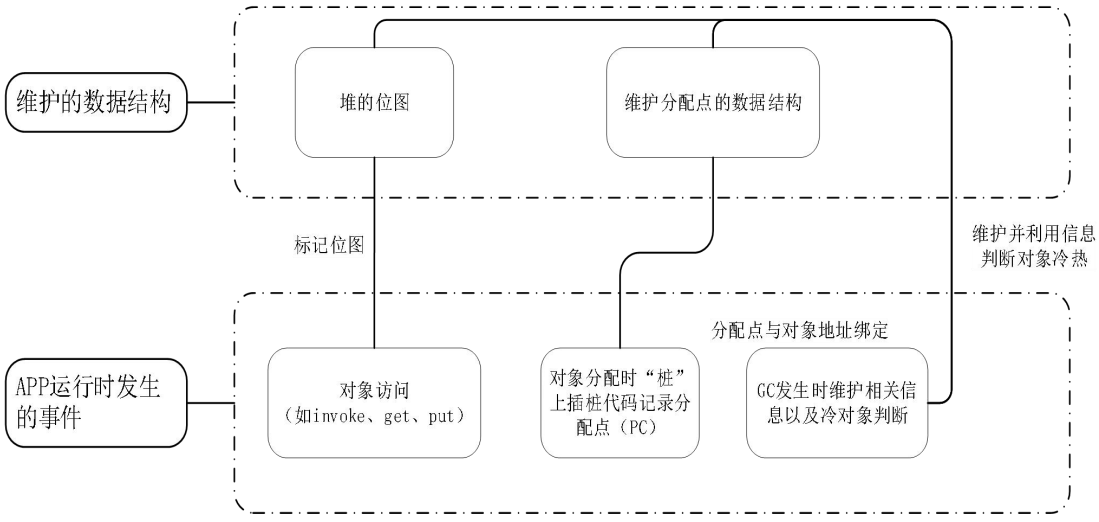

图 2.1 系统功能概述

根据前面的设计思路所示，我们需要修改 Android 系统中的 android/art/rutime 中的代码，在主 GC concurrent_copying 中插桩代码以及解释器中插桩代码实现对对象在运行时的行为活动进行监测。同时需要修改 android/art/compiler 中的代码，保证编译生成本地代码时生成位图访问代码。需要添加的功能如上一节所述，主要包含了 APP 运行时发生的时间维护和在

rumtime 中数据结构的维护。

工具所修改的 ART 相关文件如下图，增加的详细代码可以在该项目的私人 github 仓库中看到与原来的 ART 的具体对比。

主要修改文件：

/art/runtime/leakleak/leakleak.h
/art/runtime/leakleak/leakleak.cc
/art/runtime/gc/collector/concurrent_copying.cc
/art/runtime/gc/collector/concurrent_copying-inl.h
/art/runtime/gc/accounting/space_bitmap.cc
/art/runtime/gc/accounting/space_bitmap.h
/art/runtime/arch/arm64/quick_entrypoints_arm64.S
/art/runtime/gc/heap-inl.h
/art/runtime/gc/heap.h
/art/runtime/gc/heap.cc
/art/runtime/interpreter/interpreter_common.cc
/art/runtime/interpreter/interpreter_common.h
/art/runtime/runtime.cc
/art/runtime/thread.cc
/art/runtime/thread.h
/art/runtime/gc/space/large_obj_space.cc
/art/jit/jit_code_cache.cc
/art/compiler/optimizing/code_generator_arm64.cc
/art/compiler/optimizing/code_generator_arm64.h
/art/runtime/entrypoint/quick/quick_alloc_entrypoints.cc
/art/runtime/interpreter/mterp/arm64/array.S
/art/runtime/interpreter/mterp/arm64/object.S

代码设计采用单例模式，设计了一个 LeakTrace 类，功能包括了上述系统机制中的大部分功能，其他部分数据维护在“class heap”或者“class thread”中维护，LeakTrace 主要目的帮助在运行时维护对象相关信息以及查找怀疑的对象。

类中主要包含的成员和函数如下：

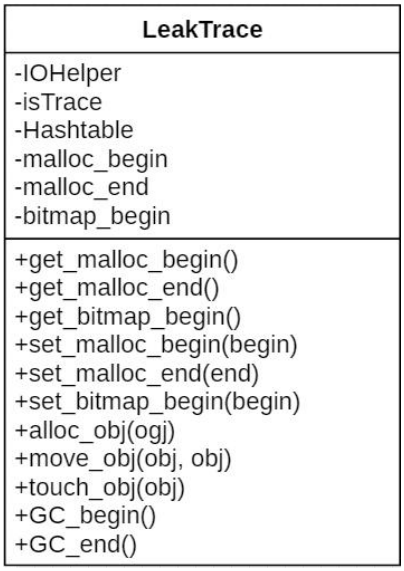

图 2.3 LeakTrace 类

其中 Hashtable 是用来维护分配点与对象的关系，在设置监测对象大小大于 128byte 的情况下，空间压缩比为 16：1，最坏情况下占堆大小的 6.25%。并且实际上往往更多的对象大于阈值，并且堆并不会长时间保持满状态，因此实际内存负载更低。Bitmap 则在堆初始化是进行构造，在 LeakTrace 类只持有 bitmap 的相关信息。

2.3. Compiler 与 Interpreter 的主要修改

对于对象访问，我们需要在 Android 虚拟机中对应的解释器与编译器中插桩代码，以达到标记位图的效果。我们选择 iput，iget，aput，aget，invoke-virtual，invoke-direct 等 bytecode 指令作为访问标记。对应解释器中发现目标指令时在解释执行之前进行“访问对象”的位图标记。在编译器部分，我们修改/android/art/compiler/optimizing/code_generator.cc 中插桩代码，在将 IR 中对应 bytecode 所表示的代码生成部分插桩代码，判断生成的对象是否在 main_space 上，如果是在 main_space 的对象则进行位图的修改。具体生成的一段 arm64 汇编码如下所示：

```
0x00013aac: a90157f4      stp x20, x21, [sp, #16]
0x00013ab0: a9027bf6      stp x22, lr, [sp, #32]
0x00013ab4: d2a65810      mov x16, #0x32c00000
0x00013ab8: eb10003f      cmp x1, x16
0x00013abc: 5400032a      b.ge #+0x64 (addr 0x13b20)
0x00013ac0: d2a25810      mov x16, #0x12c00000
0x00013ac4: eb01021f      cmp x16, x1
0x00013ac8: 540002cd      b.le #+0x58 (addr 0x13b20)
0x00013acc: f944c670      ldr x16, [tr, #2440] ; 2440
0x00013ad0: eb01021f      cmp x16, x1
0x00013ad4: 54000260      b.eq #+0x4c (addr 0x13b20)
0x00013ad8: f904c661      str x1, [tr, #2440] ; 2440
0x00013adc: d10043ff      sub sp, sp, #0x10 (16)
0x00013ae0: f90003ef      str x15, [sp]
0x00013ae4: d2a25810      mov x16, #0x12c00000
0x00013ae8: cb100030      sub x16, x1, x16
0x00013aec: d343fe10      lsr x16, x16, #3
0x00013af0: 92400a0f      and x15, x16, #0x7
0x00013af4: d343fe10      lsr x16, x16, #3
0x00013af8: d28a0011      mov x17, #0x5000
0x00013afc: f2bccd51      movk x17, #0xe66a, lsl #16
0x00013b00: 8b100231      add x17, x17, x16
0x00013b04: d2800030      mov x16, #0x1
0x00013b08: 9acf2210      lsl x16, x16, x15
0x00013b0c: 39c0022f      ldrsb w15, [x17]
0x00013b10: 2a1001ef      orr w15, w15, w16
0x00013b14: 3900022f      strb w15, [x17]
0x00013b18: f94003ef      ldr x15, [sp]
0x00013b1c: 910043ff      add sp, sp, #0x10 (16)
```

图 2.4 部分插桩代码示例

在这里，我们还在 thread_local 中增加了一个字段，用于优化连续被访问的对象，通过比较如果连续访问同一对象则忽略插桩进行跳转，在一定程度上降低了负载。

在上述对象访问插桩代码中未对位图上锁，上锁会对操作带来较大负载，因此在系统实现中选择不进行锁操作。如果一定需要上锁，可以考虑借助 ARM 指令中的“ldrex”和“strex”来实现 4 字节的读写锁。

不对 BITMAP 访问加锁可能在多线程环境中对系统造成“误报”。考虑如下场景：插桩代码中通过“ldrsb”和“strb”读写对象地址对应的一个“字”，如果其他线程同时读取到同一个“字”，则会进行写操作被覆盖，从而丢掉了本次访问信息。但我们认为这种访问概率较低：首先，由于工具中可以设置被监测的对象，而通常较大对象（例如大于 1K 的对象）的内存泄漏才是被关注的焦点，而大对象在 BITMAP 中会独自占据超过 1 个字（对应 256 字节的对象）甚至多个字的空间，因此大对象之间不会产生“字”长度上的资源争夺；其次，工具只需要记录 N 次主 GC 之间是否发生过对象的读写访问，在此期间偶尔漏记不会对工具的漏报误报概率产生较大影响；最后，内存泄漏工具本身只是一种监控手段，牺牲部分精确性来保证性能在绝大多数应用环境下是值得的。

2.4. 对象分配代码修改

对象分配时，我们需要得到对象分配点的 pc 值，在实际运行中，可以通过桩代码获得 lr 寄存器的值获得。因此，我们选择在堆分配对象时，在桩代码中插桩代码取得 lr 寄存器的值，在这里将 lr 寄存器的值保存到我们在 thread 中增加的字段。因为在 ART 中，X19 寄存器总是保存线程的地址，因此我们可以轻松的在代码中插桩代码将 pc 的值保存到我们增加的字段上。同时，在堆分配对象的出口获得当前 thread 我们增加的新字段的值即可。（注意通过 TLAB 的分配汇编码中也要进行插桩代码）

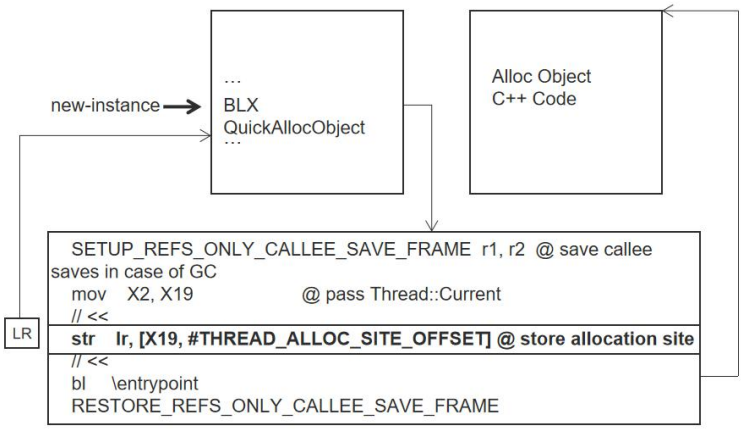

图 2.4 对象分配点

之后在 mirror::Object* Heap::AllocObjectWithAllocator(...)进行插桩代码，获取分配点的 PC 值并与对象地址进行绑定。同时为了获取 PC 值对应的方法，我们需要在 jit、aot 的地方插桩代码，记录方法的入口地址与大小，方便统计时进行聚类输出，以 JIT 为例，所有 JIT 代码都存在 JIT_Cache 中，在编译时我们可以轻松的获得方法地址与大小，可以选择将信息储存离线或在线进行分析，其他方法的入口点都可以通过类似方法进行 dump 或者在线储存分析。

2.5. Runtime 的主要修改

Runtime 中主要修改 GC 中的代码，对主 GC——Concurrent copying 进行修改，主要涉及 GC 发生时对象的维护，一个是移动对象时对于 hashtable 的动态维护，一个是对于在时间间隔触发时对于对象走查时的动态维护。根据 Android 10.0 中堆的实现特点。其中 Main_Space 由 Region_Space 构成，其中 Region_Space 由若干个 Region 构成，每个大小 256KB。我们对于每个 Region

构建一个 hashtable，进而减少不同线程之间对于锁的竞争。从而分担了并发 GC 给我们的处理机制带来的负载。

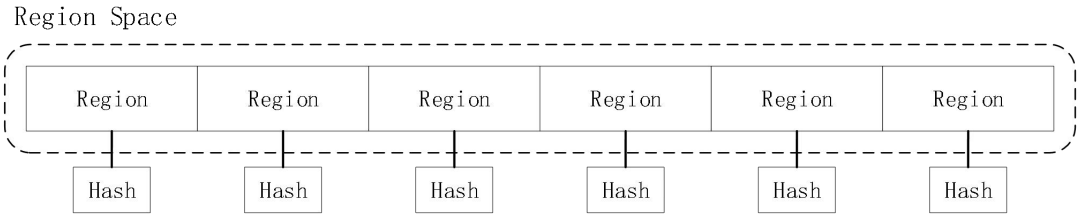

图 2.6 Hash 维护对象信息

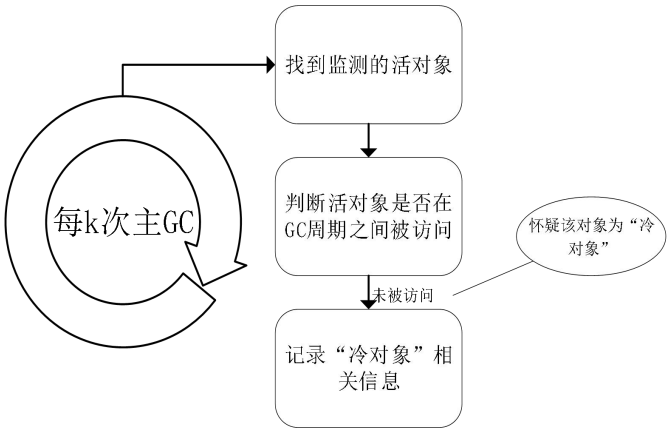

图 2.7 GC 机制

代码修改位置位于/android/art/rumtime/gc/collector/在 GC 开始时 RunPhases 中插桩代码，进行 GC 相关信息维护，开始阶段计算该次 GC 是第几次 GC，来判断是否在 GC 中需要进行对象检查，如果需要监测，则需要在从根出发的根集引用遍历时进行判断，通过判断对象对应的标记位图是否被标记和分配时处于的“时间点”来判断是否怀疑该对象“泄漏”。同时在 GC 结束时如果需要进行聚类汇总输出。在其他非监测的 GC 中，我们主要需要维护 GC 时的对象移动操作，与在需要判定时进行相关维护（如：地址与分配点的重新绑定）。对于已经回收的冗余信息，我们需要单独一个机制进行处理，因为在 heap 上不是所有对象都有一个显示的释放动作。在现有设计系统中我们考虑在走查 Mark 对象时候记录走过的对象，对于一定 GC 次数后都未被走到的对象我们认为其以及被释放，在数据结构中进行擦除（这里可以选择再建一个位图或者通过 hashtable 来进行维护，在监测对象大小阈值设置较大时，整体数量较少时 hashtable 更为合适，反之则位图更佳）。

3. 性能测试及功能展示

3.1. 功能实验

为了验证系统功能，我们设计了若干个“内存泄漏”的 Java 程序。通过将对象绑定到静态引用上，制造冷对象。并通过反复的对象分配操作刺激 GC，激发判断机制启动。一个例子如下：

```
public class ListLeak
{
        int id = 0;
        int id_1= 2;
        int id_2= 3;
        int id_3= 4;
        int id_4= 5;
        int id_5= 6;
        int id_6= 7;
        int id_7= 8;
        int id_8= 9;
        int id_9= 0;
        int id_10 =11;
        ListLeak next;
        int make_leak(int no){
                int j=no;
                end.next = new ListLeak();
                end = end.next;
                for(int i = 0; i < 5000; i++) {
                        j+=no;
                }
                return j;
        }
        static ListLeak end;
        public static void main(String[] args) throws InterruptedException {
                int N = 5000000;
                ListLeak o =  new ListLeak();
                end = o;
                int f=0;
                int[] arr;
                for(int i = 0; i < N; i++){
                        arr = new int[10];
                        arr[(i)%8] = arr[i%10]+N;
                        o.id = arr[(i)%10];
                        if(i%1000000==0){
                                System.gc();
                                Thread.sleep(1000);
                        }
                }
                long[] le = new long[100000];
                for(int i = 0; i < 500; i++) f += o.make_leak(i*i);
                for(int i = 0; i < N; i++){
                        arr = new int[10];
                        arr[(i)%8] = arr[i%10]+N;
                        o.id = arr[(i)%10];
                        if(i%1000000==0){
                                System.gc();
                                Thread.sleep(1000);
                        }
                }

                le[0]=N;

        }
}
```

大对象，及小泄漏对象

图 3.1 插桩代码

在实验中共触发主 GC 18 次，监测程序占用内存约 10MB 左右并发现了 3 个 JIT-Method 泄漏点，分别为大对象 long[],设计的泄漏对象 ListLeak，和用于触发 GC 的 int[],其中由解释器分配的 ListLeak 进行过监测，未发现泄漏。例子中的主方法都进行 JIT（osr）编译，其他检测到的怀疑点应来自其他库函数的 native code（如果存在误报，可能原因是指令选择插桩代码不全或存在 Data

Race 数据竞争现象）。

功能实验输出信息：

maybe leak at 0x70e191c8 and GC is 16 and class is java.lang.String

maybe leak at 0x44c00c94 and GC is 16 and from method void ListLeak.main(java.lang.String[])(osr) and class is ListLeak

maybe leak at 0x70e195a4 and GC is 16 and class is java.lang.String

maybe leak at 0x70d0730c and GC is 16 and class is java.lang.Class

maybe leak at 0x44c00c44 and GC is 16 and from method void ListLeak.main(java.lang.String[])(osr) and class is long[]

maybe leak at 0x71034dac and GC is 16 and class is java.lang.String

maybe leak at 0x70dccfe8 and GC is 16 and class is java.lang.Thread[]

maybe leak at 0x70e7dccc and GC is 16 and class is java.lang.Thread

maybe leak at 0x44c00b28 and GC is 16 and from method void ListLeak.main(java.lang.String[])(osr) and class is int[]

alloc obj tol:59898

alloc obj byte tol:4958920

diff pc tol:15

find new tol:59999

3.2. 性能实验

验证功能之后，需要对实现的机制进行性能负载测试。本次测试选用在 JIT 模式下运行 eembc，监测大于等于 48 字节的对象。需要特别指出，jitted code 中所有对象读写访问点均进行了插桩，所有 Java 对象，无论大小是多少，其被访问信息均得到了记录。但在 GC 中，工具根据预设的对象大小阈值仅对大于等于 48 字节的对象进行生命周期的跟踪记录。经统计在堆上分配的大于 48 字节的对象约为 3.7 万个，约堆上分配对象总数的 3.2%，对比插桩代码前后的性能如下：

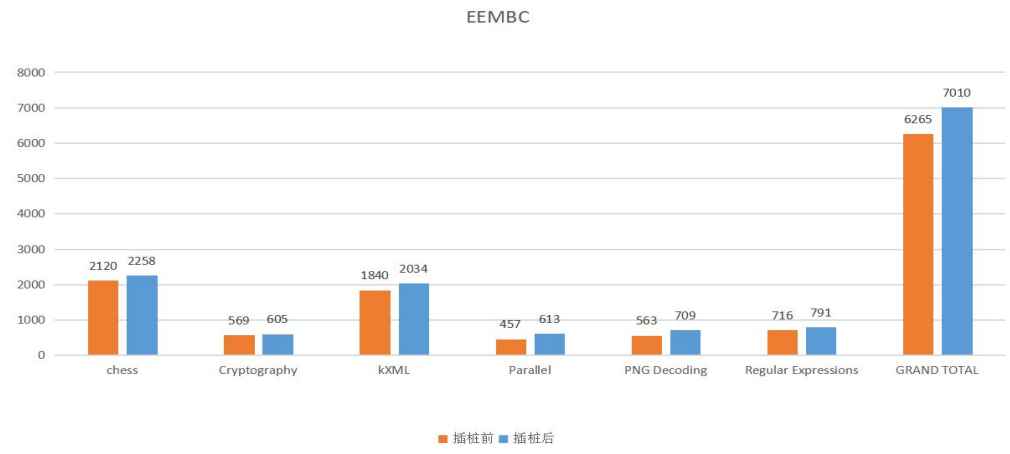

图 3.3 性能测试

经过对比，程序在插桩后的虚拟机上运行速度上大约是在原虚拟机上运行速度的 89.37%，即对**性能的负面影响为 10.63%**。该负载还有进一步优化的空间。

表 3.1 给出了详细的性能运行数据。其中，“Profiler”列的数据为打开内存泄漏监测工具后得到的性能数据，“Original”列的数据为原 ART Dalvik 虚拟机运行的数据。

表 3.1 EEMBC 性能测试数据

	Chess		Cryptography		kXML		Parallel		PNG Decoding		Regular Expressions		GRAND TOTAL	
	Ori.	Prof.	Ori	Prof.	Ori	Prof	Ori	Prof	Oril	Prof	Ori	Prof	Ori	Prof
Rnd1	517	544	289	191	448	464	167	186	135	219	148	184		
Rnd2	362	430	98	121	334	385	92	126	99	129	115	159		
Rnd3	364	449	91	94	332	386	93	65	98	122	114	143		
Rnd4	368	418	97	100	325	404	117	89	111	120	126	163		
Rnd5	367	417	88	99	329	395	71	147	102	119	118	142		
Total	1978	2258	663	605	1768	2034	540	613	545	709	621	791	6265	7010

此外，该内存监测工具在 Java 程序运行期间，除了文本格式的内存泄漏警告信息，不输出任何中间结果，不占用任何带宽（可以选择配置输出类中的方法与入口点）。下面的数据表示 EEMBC 在运行时的监测输出，共统计到分配点 330 个，其中怀疑有 5 个 JIT 分配点存在疑似冷对象的分配点。

性能实验输出信息：

info about: dalvikvm:

maybe leak at 0x44c3bcc0 and GC is 32 and from method void com.sun.mep.bench.png.ReverseFilter.<init>(byte[], com.sun.mep.bench.png.Header) and class is byte[]

maybe leak at 0x44c3cd48 and GC is 32 and from method void com.sun.mep.bench.png.Inflater.<init>(byte[], com.sun.mep.bench.png.Header) and class is byte[]

maybe leak at 0x44c34188 and GC is 32 and from method void com.motorola.bench.parallel.ParallelBench.cleanupAndVerify() and class is char[]

maybe leak at 0x7044bfcbe8 and GC is 32 and class is int[]

maybe leak at 0x71d3294c and GC is 32 and class is java.lang.Object[]

maybe leak at 0x71e0c6a8 and GC is 32 and class is java.lang.Object[]

maybe leak at 0x44c21988 and GC is 32 and from method org.kxml.parser.StartTag org.kxml.parser.XmlParser.parseStartTag() and class is char[]

maybe leak at 0x44c1f99c and GC is 32 and from method void org.kxml.parser.StartTag.<init>(org.kxml.parser.StartTag, java.lang.String, java.lang.String, java.util.Vector, boolean, boolean) and class is java.lang.String

maybe leak at 0x7044bf71a6 and GC is 32 and class is int[]
maybe leak at 0x7044bf714c and GC is 32 and class is int[]
maybe leak at 0x7044bff380 and GC is 32 and class is com.sun.mep.bench.Chess.Rook
maybe leak at 0x71e12fa4 and GC is 32 and class is java.util.jar.JarFile
maybe leak at 0x7044bff362 and GC is 32 and class is com.sun.mep.bench.Chess.Bishop
maybe leak at 0x7044bff358 and GC is 32 and class is com.sun.mep.bench.Chess.Bishop
maybe leak at 0x7044bff3f2 and GC is 32 and class is com.sun.mep.bench.Chess.Pawn
maybe leak at 0x7044bf3a8e and GC is 32 and class is org.kxml.parser.XmlParser
maybe leak at 0x7044c09cfa and GC is 32 and class is int[]
maybe leak at 0x7044bff6f6 and GC is 32 and class is com.sun.mep.bench.Chess.Evaluator
maybe leak at 0x7044c09cec and GC is 32 and class is int[]
maybe leak at 0x7044bff3b6 and GC is 32 and class is com.sun.mep.bench.Chess.King
maybe leak at 0x71d32604 and GC is 32 and class is java.lang.Object[]
maybe leak at 0x7044c0326c and GC is 32 and class is byte[]
maybe leak at 0x7044bff75e and GC is 32 and class is com.sun.mep.bench.Chess.AlphaBeta
maybe leak at 0x7044c02d36 and GC is 32 and class is int[]
maybe leak at 0x7044c0b3c2 and GC is 32 and class is byte[]
maybe leak at 0x71e12dac and GC is 32 and class is java.lang.String
alloc obj tol:37014
alloc obj byte tol:81952006
diff pc tol:330
find new tol:1237728

程序运行中其他相关负载统计：

- 发现堆上分配对象约 123.7 万，实际监控对象约为 3.7 万。
- 工具的内存负载大约为 9.2MB，其中包含 bitmap 占 8MB（main_space 大小为 512MB），其他数据结构占内存峰值（如 hashtable 等）约占 1.2MB。
- GC（含新生代 GC）共发生 86 次，分别取最大值、平均数、中位数进行比较：（1）最大值：插桩后 GC 时间最大值为 131ms，插桩前最大值为 77ms。（2）平均数：插桩后 GC 时间平均数为 26ms，插桩前平均数为 17ms。（3）中位数：插桩后 GC 时间中位数为 16ms，插桩前中位数为 13ms。

3.3. 稳定性实验

进行性能实验后，需要对实现的机制进行稳定性测试。我们了选择 system_server 和 com.android.launcher3 进行监测，并统计相关信息。

在 com.android.launcher3 中经过 monkey 进行 50000 次随机操作，system_server 工作正常。输出的分析信息如下所示：

稳定性实验输出信息：

…

maybe leak at 0x71bc9df8 and GC is 32 and class is java.lang.Class

maybe leak at 0x71c7735c and GC is 32 and class is java.lang.String

maybe leak at 0x714a6508 and GC is 32 and class is java.lang.Class

maybe leak at 0x7133ebb0 and GC is 32 and class is java.util.concurrent.ThreadPoolExecutor

maybe leak at 0x71c783ac and GC is 32 and class is android.content.res.Configuration

maybe leak at 0x71cacb50 and GC is 32 and class is android.graphics.drawable.GradientDrawable

maybe leak at 0x715390b4 and GC is 32 and class is java.lang.String[]

maybe leak at 0x73d621a3a8 and GC is 32 and class is java.lang.Class

maybe leak at 0x714c9aac and GC is 32 and class is android.system.StructStat

maybe leak at 0x742f35ce0c and GC is 32 and class is com.android.internal.colorextraction.drawable.ScrimDrawable

maybe leak at 0x71c97774 and GC is 32 and class is java.lang.String

maybe leak at 0x743a52d38e and GC is 32 and class is int[]

maybe leak at 0x73d618622c and GC is 32 and class is int[]

maybe leak at 0x742f3bcf90 and GC is 32 and class is java.lang.String

maybe leak at 0x720cfe08 and GC is 32 and class is java.lang.String

maybe leak at 0x742f3c2034 and GC is 32 and class is com.android.systemui.qs.QuickStatusBarHeader

maybe leak at 0x73d618623a and GC is 32 and class is int[]

maybe leak at 0x73d62afd8c and GC is 32 and class is com.android.systemui.statusbar.notification.row.ExpandableNotificationRow

maybe leak at 0x742f34c2f4 and GC is 32 and class is java.lang.Class

maybe leak at 0x714c71ac and GC is 32 and class is char[]

maybe leak at 0x713ca818 and GC is 32 and class is java.lang.String

maybe leak at 0x71bd5944 and GC is 32 and class is android.animation.PropertyValuesHolder$FloatPropertyValuesHolder

maybe leak at 0x742f429d30 and GC is 32 and class is java.lang.String

maybe leak at 0x73d633dee4 and GC is 32 and class is java.lang.Class

maybe leak at 0x71cc26f8 and GC is 32 and class is android.graphics.drawable.VectorDrawable$VFullPath

maybe leak at 0x73d61862d2 and GC is 32 and class is int[]

maybe leak at 0x743bff96c6 and GC is 32 and class is java.math.BigInteger

maybe leak at 0x71e66064 and GC is 32 and class is java.lang.String

maybe leak at 0x71532634 and GC is 32 and class is android.system.StructStat

maybe leak at 0x743b6cc1be and GC is 32 and class is float[]

maybe leak at 0x742f2a4d40 and GC is 32 and class is java.lang.Class

JIT method: void androidx.gridlayout.widget.GridLayout$Axis$1.<init>(androidx.gridlayout.widget.GridLayout$Axis, androidx.gridlayout.widget.GridLayout$Arc[]) and code: 0x9c518450

alloc obj tol:57779

alloc obj byte tol:17649637

diff pc tol:1652

find new tol:224639

其中，共统计到分配点 1652 个，部分产生可能泄漏信息，但是分配点在

native 库中，因此仅能得到 PC 值。程序运行中其他相关表现为：

- 发现堆上分配对象约 22.4 万，实际监控对象约为 5.7 万。
- 工具的内存负载大约 18.8MB，其中包含 bitmap 占 16MB（main_space 大小为 1GB），其他数据结构占内存峰值（如 hashtable 等）约占 2.8MB。
- GC 共发生 39 次，分别取最大值、平均数、中位数进行比较：（1）最大值：插桩后 GC 时间最大值为 388ms，插桩前最大值为 163ms。（2）平均数：插桩后 GC 时间平均数为 128ms，插桩前平均数为 71ms。（3）中位数：插桩后 GC 时间中位数为 102ms，插桩前中位数为 58ms。

在 system_server 中经过 monkey 进行 50000 次随机操作，launcher 工作正常。输出的分析信息如下：

稳定性实验输出信息：

```
…
    maybe leak at    0x715ce39c and GC is 16 and class is int[]
    maybe leak at    0x715ce2ec and GC is 16 and class is char[]
    maybe leak at    0x715be8f0 and GC is 16 and class is long[]
    maybe leak at    0x71c77630 and GC is 16 and class is java.lang.String
    maybe leak at    0x743b649410 and GC is 16 and class is long[]
    maybe leak at    0x71ba6d38 and GC is 16 and class is int[]
    maybe leak at    0x71b78118 and GC is 16 and class is int[]
    maybe leak at    0x7161dfa4 and GC is 16 and class is java.util.jar.JarFile
    maybe leak at    0x71cc5680 and GC is 16 and class is int[]
    maybe leak at    0x713aed5c and GC is 16 and class is java.security.CryptoPrimitive[]
    maybe leak at    0x743b6cc1b0 and GC is 16 and class is float[]
    maybe leak at    0x71caced4 and GC is 16 and class is int[]
    maybe leak at    0x71c76664 and GC is 16 and class is int[][]
    maybe leak at    0x71c9d3c0 and GC is 16 and class is
android.graphics.drawable.VectorDrawable$VectorDrawableState
    maybe leak at    0x71b7858c and GC is 16 and class is java.lang.String
    maybe leak at    0x71534204 and GC is 16 and class is java.lang.String[]
    maybe leak at 0x9c4dd63c and GC is 16 and from method void
android.app.WindowConfiguration.<init>() and class is android.content.res.Configuration
    maybe leak at    0x7147f1a4 and GC is 16 and class is java.text.DecimalFormatSymbols
    maybe leak at 0x9c4f9260 and GC is 16 and from method android.animation.AnimatorSet
android.animation.AnimatorSet.clone() and class is android.animation.AnimatorSet$Node
    maybe leak at    0x71495384 and GC is 16 and class is java.lang.String[]
    maybe leak at    0x743acbf846 and GC is 16 and class is java.lang.String[]
    maybe leak at    0x7161a224 and GC is 16 and class is char[]
    maybe leak at    0x7161d9dc and GC is 16 and class is java.lang.String
    maybe leak at    0x7161dce4 and GC is 16 and class is java.lang.String
```

android.animation.PropertyValuesHolder$FloatPropertyValuesHolder

maybe leak at 0x714025a4 and GC is 16 and class is java.lang.String

maybe leak at 0x743c30c31c and GC is 16 and class is java.util.HashMap$Node[]

maybe leak at 0x7145e69c and GC is 16 and class is java.lang.String

maybe leak at 0x71baa344 and GC is 16 and class is char[]

maybe leak at 0x716176a8 and GC is 16 and class is java.lang.Object[]

maybe leak at 0x743c341070 and GC is 16 and class is java.util.concurrent.ConcurrentHashMap$Node[]

alloc obj tol:329799

alloc obj byte tol:92425379

diff pc tol:1213

find new tol:940455

其中，共统计到分配点 1213 个，部分产生可能泄漏信息，但是分配点在 native 库中，因此仅能得到 PC 值。程序运行中其他相关表现为：

- 发现堆上分配对象约 94 万，实际监控对象约为 32.9 万。
- 工具的内存负载大约 24.1MB，其中包含 bitmap 占 16MB（main_space 大小为 1GB），其他数据结构占内存峰值（如 hashtable 等）约占 8.1MB。
- GC 共发生 21 次，分别取最大值、平均数、中位数进行比较：（1）最大值：插桩后 GC 时间最大值为 651ms，插桩前最大值为 286ms。（2）平均数：插桩后 GC 时间平均数为 173ms，插桩前平均数为 117ms。（3）中位数：插桩后 GC 时间中位数为 155ms，插桩前中位数为 88ms。

3.4. 总结

当前版本实现了一个低负载的在线 Java 对象监测机制，可通过配置对特定应用进行监控并指定“冷”的 GC 次数。该内存监测工具在 Java 程序运行期间，除了文本格式的内存泄漏警告信息，不输出任何中间结果，不占用任何带宽。程序在插桩后的虚拟机上运行速度上大约是在原虚拟机上运行速度的 89.37%，即对性能的负面影响为 10.63%。该负载还有进一步优化的空间。

此外，目前工具仍处于原型状态，在可靠性、界面友好、输出信息分析能力上，仍有较大的提升余地。