



轻量化内存泄露检测工具

设计文档 v0.1

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

2019 年 12 月 20 日

目录

| | |
|---|----|
| 1. 相关背景及设计思路..... | 3 |
| 1.1. 相关背景 | 3 |
| 1.2. 设计思路 | 3 |
| 2. 详细设计..... | 4 |
| 2.1. 相关开发环境 | 4 |
| 2.2. 对 Android 系统的修改..... | 5 |
| 2.3. Compiler 与 Interpreter 的主要修改 | 7 |
| 2.4. 对象分配代码修改 | 7 |
| 2.5. Runtime 的主要修改 | 8 |
| 3. 性能测试及功能展示..... | 9 |
| 3.1. 功能实验 | 9 |
| 3.2. 性能实验 | 10 |
| 3.3. 总结 | 11 |

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的疏忽,被其他对象持有了其引用(但并不使用它们),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知道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。

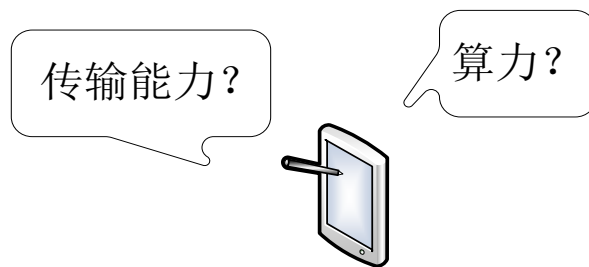


图 1.1 需要轻量级的在线机制

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,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。

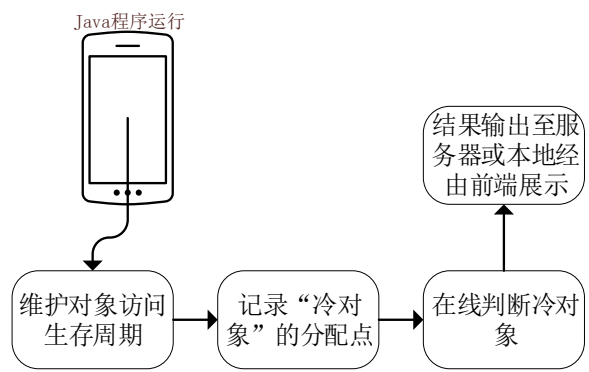


图 1.2 系统示意图

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，因此对于对象访问标记的代码插桩，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。

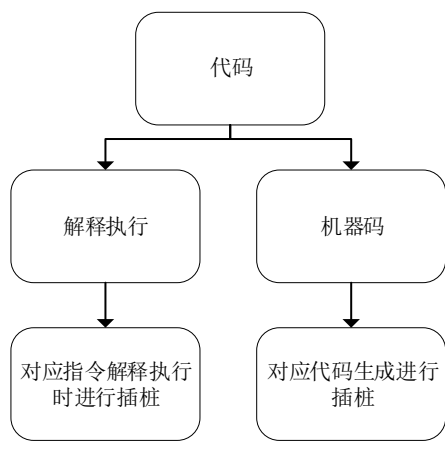


图 1.3 GC 机制与代码插桩

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-8.0.0_r2

```
build target: aosp_sailfish-userdebug
```

```
target build type: release
```

2.2. 对 Android 系统的修改

根据前面的设计思路所示，我们需要修改 Android 系统中的 android/art/runtime 中的代码，在主 GC concurrent_copying 中插桩代码以及解释器中插桩代码同时要修改 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 runtime 中数据结构的维护。

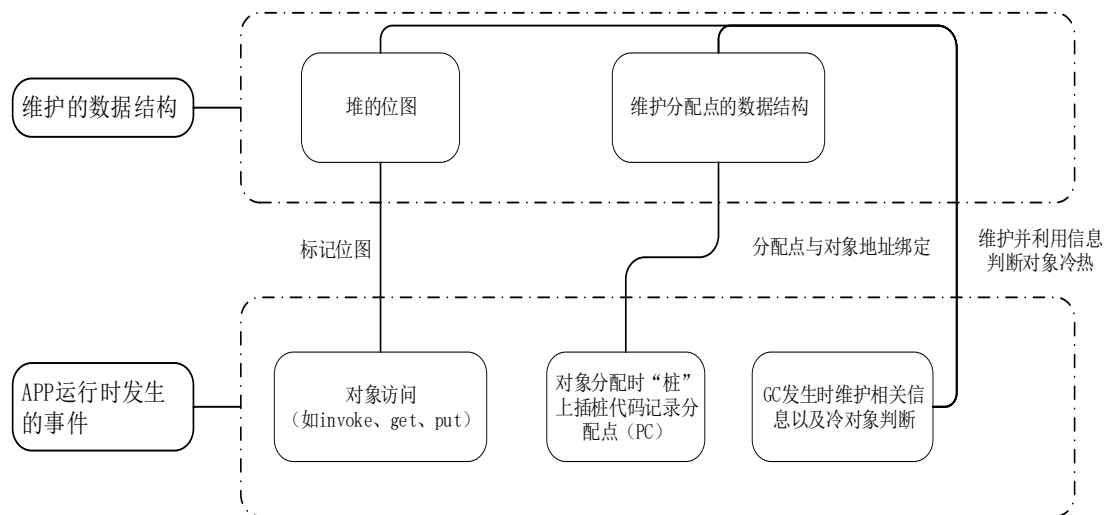


图 2.1 系统功能概述

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_common.cc |
| /art/runtime/interpreter_common.h |
| /art/runtime/runtime.cc |
| /art/runtime/thread.cc |
| /art/runtime/thread.h |

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。**LeakTrace** 类中主要包含的成员和函数如下：

| LeakTrace |
|--|
| -IOHelper -isTrace -Hashtable -malloc_begin -malloc_end -bitmap_begin |
| +get_malloc_begin() +get_malloc_end() +get_bitmap_begin() +set_malloc_begin(begin) +set_malloc_end(end) +set_bitmap_begin(begin) +alloc_obj(obj) +move_obj(obj, obj) +touch_obj(obj) +GC_begin() +GC_end() |

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，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，来进行位图的修改。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

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，通过比较进行跳转，在一定程度上降低了负载。

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` 的值保存到我们在增加的字段上。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。

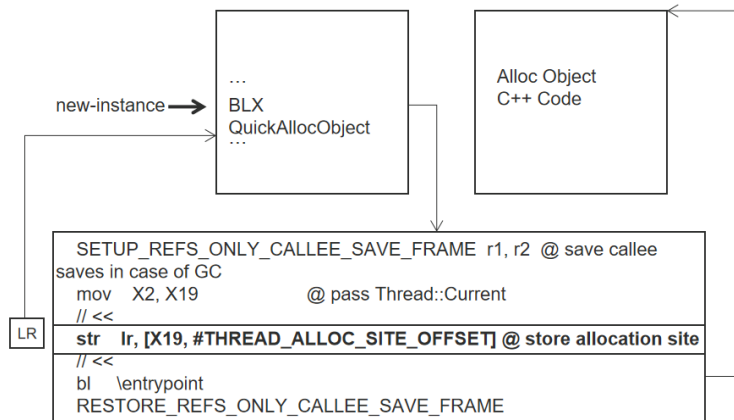


图 2.4 对象分配点

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。这里有一个显然的问题是在维护 hashtable 时我们要借助到 ART 中的锁机制，但是由于本身在 GC 过程中很多操作是要持有锁的，因此我们在这里加锁不会造成过大的负载。

代码修改位置位于 `/android/art/runtime/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。同时在 GC 结束时如果需要进行输出，则对保留的结果进行输出。在具体在 GC 里，我们维护 GC 时的对象移动操作，与在需要判定时进行相关维护。

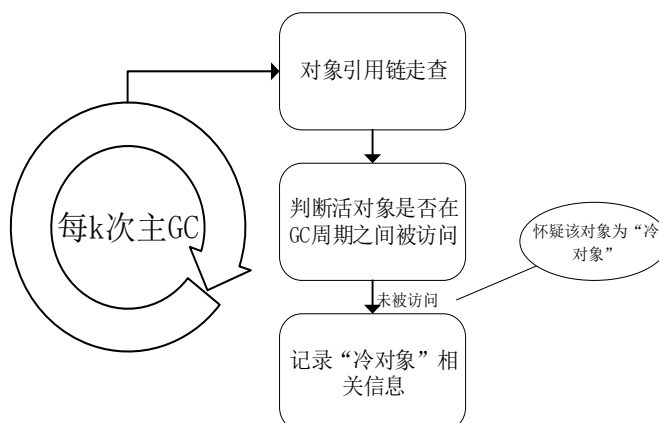


图 2.4 GC 机制

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
{
    String name;
    int id;
    int[] something;


    void set_id(int x){
        id = x;
    }
    ListLeak(){
        name = "zhang";
        id = 1;
        something = new int[100];
    }
    static List<ListLeak> list ;
    public static void main( String[] args ){
        list = new ArrayList<ListLeak>();
        int N = 300000;
        ListLeak o = new ListLeak();  内存泄漏点
        list.add(o);
        for(int i = 1; i < N; i++){
            ListLeak p = new ListLeak();
            list.add(p);
        }
        for(int i = 0; i < 10; i++){
            for(int k = 1; k < N; k++){
                list.get(k).set_id(k);
            }
            for(int k = 0; k < N/5; k++){
                ListLeak q = new ListLeak();
            }
            System.out.println("add List: "+i);
        }
    }
}
```

图 2.5 插桩代码

在图 2.5 的例子程序中，ListLeak 对象在主程序的第一个循环中，被创建了 30 万个，并被加入类型为 ArrayList 的 list 表中。在主程序的第二个循环中，list 中的对象被频繁访问，但第一个被加入 的 ListLeak 对象一直未被任何程序访问。该对象即为我们前文所述的“冷对象”，存在泄漏的较大肯定性。图 2.6 展示了我们的内存泄漏监测工具在程序运行过程中给出的可能泄漏信息，可以看到在程序运行过程中一共触发了主 GC 6 次，并发现一个泄漏点（地址 317194528，分配点 PC 地址 1893556244. 通过比对 method 编译后的运行区间，可以知道该 PC 就是 main 函数的第一个分配点 PC）。

```

10-07 20:26:42.907 2810 2824 D leakleak: ZHANG, GC_BEGIN and this is 1
10-07 20:26:42.976 2810 2824 D leakleak: ZHANG, GC_BEGIN and this is 2
10-07 20:26:43.092 2810 2824 D leakleak: ZHANG, GC_BEGIN and this is 3
10-07 20:26:43.792 2810 2824 D leakleak: ZHANG, GC_BEGIN and this is 4
10-07 20:26:44.641 2810 2824 D leakleak: ZHANG, GC_END and this is 4
10-07 20:26:44.641 2810 2824 W dalvikvm: ZHANG,this 317194528 LEAK and alloc at1893556244
10-07 20:26:44.646 2810 2824 D leakleak: ZHANG, GC_BEGIN and this is 5
10-07 20:26:45.380 2810 2824 D leakleak: ZHANG, GC_END and this is 5
10-07 20:26:45.381 2810 2824 W dalvikvm: ZHANG,this 317194528 LEAK and alloc at1893556244
10-07 20:26:45.385 2810 2824 D leakleak: ZHANG, GC_BEGIN and this is 6
10-07 20:26:46.286 2810 2824 D leakleak: ZHANG, GC_END and this is 6

```

图 2.6 泄漏分配点 pc

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 eembc，在 JIT 模式下运行 eembc，对比插桩代码前后的性能如下：

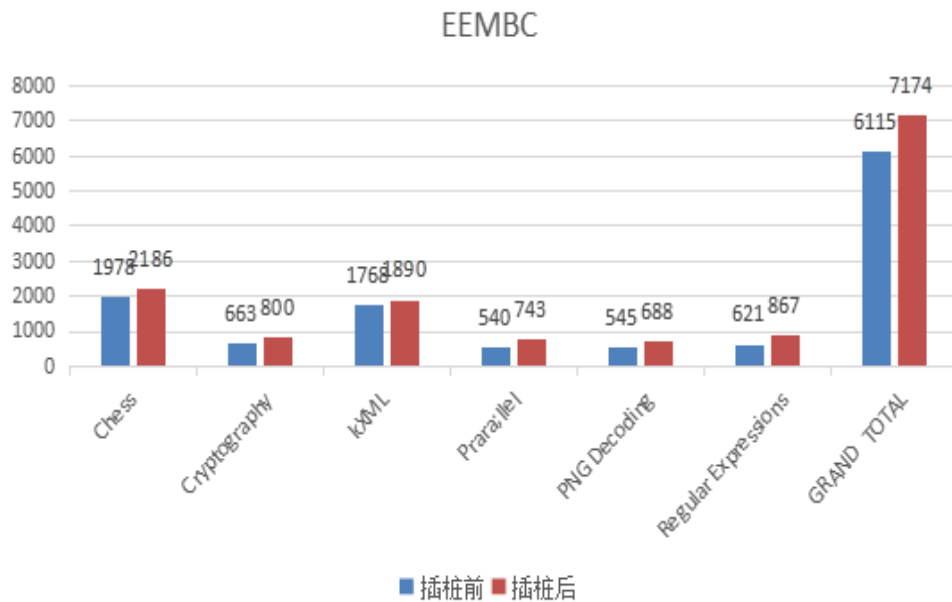


图 2.7 性能测试

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 85.24%，即对性能的负面影响为 14.76% (<15%)。该负载还有进一步优化的空间。

表 2.1 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 2.1 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. | Ori. | Prof. | Ori. | Prof. | Ori. | Prof. |

| | | | | | | | | | | | | | | |
|-------|------|------|-----|-----|------|------|-----|-----|-----|-----|-----|-----|------|------|
| Rnd1 | 517 | 563 | 289 | 272 | 448 | 431 | 167 | 196 | 135 | 191 | 148 | 239 | | |
| Rnd2 | 362 | 403 | 98 | 134 | 334 | 370 | 92 | 134 | 99 | 128 | 115 | 156 | | |
| Rnd3 | 364 | 408 | 91 | 131 | 332 | 363 | 93 | 159 | 98 | 123 | 114 | 157 | | |
| Rnd4 | 368 | 407 | 97 | 132 | 325 | 361 | 117 | 108 | 111 | 113 | 126 | 158 | | |
| Rnd5 | 367 | 405 | 88 | 131 | 329 | 365 | 71 | 146 | 102 | 133 | 118 | 157 | | |
| Total | 1978 | 2186 | 663 | 800 | 1768 | 1890 | 540 | 743 | 545 | 688 | 621 | 867 | 6115 | 7174 |

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。

3.3. 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 85.24%，即对性能的负面影响为 14.76%（<15%）。该负载还有进一步优化的空间。