(19)中华人民共和国国家知识产权局

(12)发明专利申请

(10)申请公布号 CN 102243596 A
(43)申请公布日 2011.11.16

(21)申请号 201110227094.2

(22)申请日 2011.08.09

(71)申请人 复旦大学
地址 200433 上海市杨浦区邯郸路 220 号

(72)发明人 杨珉 周波 张源

(74)专利代理机构 上海正旦专利代理有限公司 31200
代理人 陆飞 盛志范

(51)Int.Cl.
G06F 9/445(2006.01)
G06F 9/455(2006.01)

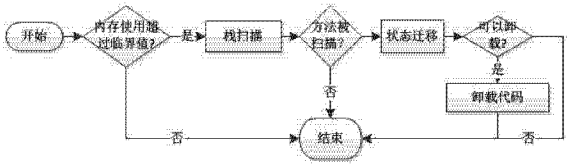
权利要求书 1 页 说明书 5 页 附图 1 页

(54)发明名称

一种自适应动态代码卸载方法

(57)摘要

本发明属于 Java 虚拟机设计技术领域，具体为一种自适应动态代码卸载方法。该方法包括：方法栈遍历，从虚拟机栈上获得每个方法的栈帧数据；方法状态迁移，依据所设计的状态机和得到的方法栈信息，做出正确的方法卸载状态迁移；代码卸载，虚拟机遍历整个方法列表，获取每个方法的卸载状态，如果其状态满足状态机所设定的终止态，则对方法的代码进行卸载，将其所占内存释放。本发明借助已有的垃圾回收机制，在垃圾回收过程中动态获取方法的状态，根据用户设置自适应地实现方法的卸载，消除了为获得较为准确的热、冷方法而进行程序剖析的额外开销，提高了程序执行速度，并且获得了比较好的代码卸载效果。



CN 102243596 A

1. 一种自适应动态代码卸载方法，其特征在于分为三个阶段：方法栈遍历、方法状态迁移和代码卸载；

在方法栈遍历阶段，从虚拟机栈上获得每个方法的栈帧数据；在方法状态迁移阶段，依据所设计的状态机、通过第一阶段得到的方法栈信息进一步获得的方法当前卸载状态以及方法在垃圾回收过程中是否被扫描信息，做出正确的方法卸载状态迁移；在代码卸载阶段，虚拟机遍历整个方法列表，获取每个方法的卸载状态，如果其状态满足状态机所设定的终止态，则对方法的代码进行卸载，将其所占内存释放。

2. 根据权利要求 1 所述的自适应动态代码卸载方法，其特征在于所述方法栈遍历，是在程序使用内存量超过阀值后启动，使用虚拟机已有的栈扫描技术，通过栈指针信息获得每个方法栈帧的数据，这些数据为下一阶段提供数据操作接口。

3. 根据权利要求 2 所述的自适应动态代码卸载方法，其特征在于所述方法状态迁移由两次迁移组成；

第一次迁移发生在完成对每个栈帧扫描结束之前，每个方法的初始状态都设置为“初始态”，如果方法在栈上被扫描到，则方法状态变为“已扫描”；如果该方法在本次扫描过程中在栈上出现多次，则其状态会停留在“已扫描”；如果该方法在上一次扫描时状态设置为“即将卸载”，那么这次扫描，该方法再次被扫描到，其状态就还原为“初始态”；

第二次迁移发生在代码卸载之前，本次迁移的对象为所有方法；对于在第一阶段扫描到方法，其状态为“初始态”或者“已扫描”，这些方法在此次不会被卸载；对于没有扫描到的方法，其状态为上一次该方法被扫描到时设置的状态，如果该方法状态为“即将卸载”，则该方法状态迁移为“已卸载”；如果为“初始态”则迁移为“即将卸载”，“已扫描”迁移为“初始态”。

4. 根据权利要求 3 所述的自适应动态代码卸载方法，其特征在于所述代码卸载，是根据第二阶段对方法的状态标识，遍历整个方法列表，对所有状态为“已卸载”的方法进行代码卸载；已卸载的方法状态将持续为“已卸载”直到其被重新编译，重新编译后，方法状态被重新设置为“初始态”；代码卸载后的方法，其代码空间被释放，给其他的方法或者数据使用。

一种自适应动态代码卸载方法

技术领域

[0001] 本发明属于 Java 虚拟机设计技术领域，具体涉及一种自适应动态代码卸载方法，包括代码生成、代码管理和垃圾回收等。

背景技术

[0002] 如今，手机、平板电脑等嵌入式设备等正在步入一个空前繁荣的时代，运行在这些设备上的程序性能直接关系到用户体验，决定着相应产品的市场控制力，也影响着技术发展的方向和进程。程序的性能一方面取决于程序本身的实现，另一方面也来自于对设备资源的高效合理的利用。对于嵌入式设备而言，由于成本的影响，它们的硬件资源往往比个人电脑（PC）更为有限。因此，高效的资源管理技术对嵌入式设备而言显得更为重要。

[0003] 内存管理是计算机中的一项重要技术。对于虚拟机而言，垃圾回收是内存管理的成熟机制。垃圾回收器通常需要对虚拟机的堆内存进行整理、回收，压缩，移动等操作，以实现有限内存的合理利用。内存管理除了管理应该程序所申请的内存之外，还包括虚拟机为编译程序而申请的代码存储内存。即时编译器（Just-In-Time Compiler，JIT）是虚拟机用来将字节码编译为机制码的组件，其实现字节码向机器码转变的过程被称为代码生成过程。经过代码生成过程得到程序机器码其本质上是一组连续的带有机器指令编码的内存。在执行这些指令之前，虚拟机为这些指令分配内存空间，然后将指令载入内存，读取指令至 CPU 相应电子以执行。通常情况下，即时编译器生成的机制码所占空间会比原有字节码所占空间增加六倍以上，这种现象被称为代码膨胀。对于硬件资源丰富的设备而言，代码膨胀对设备上运行程序的性能的影响可能不大，但是，对于嵌入式设备而言，代码膨胀所增加的内存需求对程序的性能有很大影响。

[0004] 为了更好地利用有限的内存资源，通过分析程序方法的调用特性，人们提出了代码卸载 (Code Unloading) 技术。当应用程序所需内存达到某一个临界值时，虚拟机就应该考虑对内存进行回收，代码卸载即在此时发生。在应用程序运行过程中，并不是所有的方法都会被执行到，相反，很多方法都不会被执行或者只被执行一次或几次（这些方法称为“冷方法”），更多的程序执行时间被用来反复执行为数不多的几个方法（称为“热方法”）。对于冷方法，虚拟机在代码卸载时将其所占用的内存释放或移至外存，以便将更多的热方法载入内存，以减少由于内存容量有限致使热方法的反复加载而增加的时间消耗，从而提升程序的性能，改善用户体验。

[0005] 代码卸载需要合理的方法淘汰策略，如果策略得当，移出的方法为更热的方法提供更多的内存，就能减少热方法的反复加载，使程序更快的执行；如果策略失当，移出的方法随后又被多次调用，或者移入的方法不是真正的热方法，那么进行代码卸载所需的开销作用明显，反而使程序执行时间变长。因此，符合所运行程序行为的卸载策略非常重要，而要做到较为准确地分析程序的运行时行为，就必须获得程序的运行时数据，即动态数据，并在此基础上实现可以自适应的代码卸载机制。目前常见的动态寻找热方法的技术是程序剖析（Profiling），通常借助插桩（Instrumentation）和采样（Sampling）两种方式。前者需

要在源程序中添加剖析代码，用以统计方法被调用的次数；后者虽然不需要修改源程序，但需要每隔一定时间对程序的方法栈进行扫描以获得当前正在执行的方法。两种方式都需要一定的额外开销，属于重量级的剖析方法。然而，这些重量级的方法在嵌入式设备，尤其是资源受限的嵌入式设备不适用。在这些设备上，能否实现不进行重量级剖析而又能较为准确地反应程序的动态行为的剖析方法对于代码卸载就显得至关重要。

发明内容

[0006] 本发明的目的在于提供一种能够减少程序剖析开销、提高程序执行速度的动态自适应代码卸载方法，并在 Android 操作系统内置虚拟机 Dalvik 上将其实现。

[0007] 本发明调研了运行于 Android 操作系统之上的 Java 虚拟机中垃圾回收器，发现在进行内存回收过程中，回收器会遍历当前栈上每个方法；并试验发现，热方法的栈帧经常在垃圾回收时进行的栈遍历中被访问，而冷方法的栈帧在栈遍历时出现的频率很小。这些发现表明，方法的热度与其在栈上出现的频率有关系。通过分析，我们容易得出：在栈遍历时被访问次数多的方法很可能是热方法；或者说，紧挨着两次垃圾回收过程上，都没有出现在栈上的方法不是热方法的可能性比较大。这与程序运行的特征有关系，但是，多数情况下，因为热方法被调用的频率很高，因此其在栈上出现的次数也多。因此，本发明不使用前面所述的采样或插桩的剖析方法，而将类似采样的逻辑嵌入到垃圾回收执行的过程中，这样就消除了剖析所需的开销。垃圾回收越是频繁，即触发越频繁、触发相隔时间越短，说明程序对内存的需求越大，则热方法越是集中，越是可能出现在前后两次栈遍历中。

[0008] 本发明采用的技术方案为：设置一个方法卸载状态机，并为每一个方法设置卸载状态；在垃圾回收过程中，进行栈遍历；在栈遍历的过程中同时实现方法状态按状态机迁移；在垃圾回收结束前，依据方法当前的状态决定对方法进行卸载与否。

[0009] 本发明方法分为三个阶段：方法栈遍历，方法状态迁移和代码卸载。在方法栈遍历阶段同，本发明从虚拟机栈上获得每个方法的栈帧数据。在方法状态迁移阶段，本发明依据所设计的状态机、通过第一阶段得到的方法栈信息进一步获得的方法当前卸载状态以及方法在垃圾回收过程中是否被扫描信息，做出正确的方法卸载状态迁移。在代码卸载阶段，虚拟机遍历整个方法列表，获取每个方法的卸载状态，如果其状态满足状态机所设定的终止态，则对方法的代码进行卸载，将其所占内存释放。图 1 展示了该方案的流程图。下面详述本发明的完整过程，在程序运行过程中，虚拟机对内存使用量进行监控，当内存使用量没有超过一个设定的阀值（Threshold）时，程序不会触发代码卸载功能；当使用量超过阀值时，虚拟机随即进入代码卸载。下面分别进一步具体介绍。

[0010] 1、栈遍历，这是本发明的第一阶段。如图 1 中“栈扫描”阶段所示，程序使用内存量超过阀值（临界值）后，代码卸载启动第一步，栈扫描。本发明使用虚拟机已有的栈扫描技术，通过栈指针信息获得每个方法栈帧的数据。这些数据为下一阶段提供数据操作接口。

[0011] 2、状态迁移，这是本发明的第二阶段。该阶段由两次迁移组成，分别由图 2 中实线与虚线表示。第一次迁移发生在完成对每个栈帧扫描结束之前，过程如图 2 中虚线所示。每个方法的初始状态都设置为图 2 中的“初始态”，如果方法在栈上被扫描到，则方法状态变为“已扫描”；如果该方法在本次扫描过程中在栈上出现多次，则其状态会停留在“已扫描”；如果该方法在上一次扫描时状态设置为“即将卸载”，那么这次扫描，该方法再次被扫描到，

其状态就还原为“初始态”。本发明根据上文调研结果，作出合理假设，即如果一个方法在连续两次栈扫描中都没有出现，则其可能在最近一段时间内不是热方法，其是成为被卸载方法的候选。因此，本发明设计了第二次迁移，实现对连续两次没有被扫描到的方法卸载。第二次迁移发生在代码卸载之前，过程如图 2 中实线所示。本次迁移与第一次不同，其对象为所有方法。对于在第一阶段扫描到方法，其状态为“初始态”或者“已扫描”，依据图 2，这些方法在此次不会被卸载；对于没有扫描到的方法，其状态为上一次该方法被扫描到时设置的状态，依据图 2 实线箭头，如果该方法状态为“即将卸载”，则该方法状态迁移为“已卸载”；如果为“初始态”则迁移为“即将卸载”，“已扫描”迁移为“初始态”。

[0012] 3、代码卸载，这是本发明的第三阶段。如图 1 “代码卸载”所示，根据第二阶段对方法的状态标识，遍历整个方法列表，对所有状态为“已卸载”的方法进行代码卸载。已卸载的方法状态将持续为“已卸载”直到其被重新编译，重新编译后，方法状态被重新设置为“初始态”。代码卸载后的方法，其代码空间将被释放，从而给其他的方法或者数据使用。

[0013] 本发明的有益效果是：1)本发明设计实现了一个动态自适应代码卸载机制。2)本发明无需对程序进行重量级的剖析，只需对垃圾回收稍做修改，消除了程序剖析所需的花销。3)本发明使应用程序的内存需求量降低，使得很多因内存足迹(memory footprint)过大而难以在内存有限的设备上运行的程序得以在其上运行。4)本发明寻找热方法的策略可以推广至虚拟机其他功能组件，使其他组件减少程序剖析的开销，从而发现更多的优化机会。

附图说明

[0014] 图 1 为本发明所述代码卸载机制的流程设计图。

[0015] 图 2 为栈扫描与代码卸载共同作用下的方法卸载状态机设计图。

具体实施方式

[0016] 在具体实施过程中，我们发现本发明的第二个阶段(状态迁移)的第一次迁移可以与第一个阶段整合实现，从而减少一次为设置方法状态而进行的栈扫描；同样，第二次迁移可以与第三个阶段整合实现，从而减少一次为设置方法状态而进行的整个方法列表扫描。因此，在具体实施时，我们选择将第二阶段的两次迁移结合到第一阶段和第三阶段中，从而使实现更加高效。下面将以时间顺序介绍本发明方法的二个步骤的详细实现。

[0017] 步骤一、栈扫描与第一次状态迁移

本步骤目的在于依次扫描每个栈帧，得到操作每个栈帧所属方法的操作接口，再利用该接口设置方法的卸载状态。本发明使用虚拟机内置的栈扫描方法，在扫描过程中，得到每个栈帧所属方法的方法指针，结合图 2 所示状态机，设置方法的卸载状态。方法的卸载状态包括以下四种：初始态、已扫描、即将卸载和已卸载。每个方法的卸载状态在方法加载时被设置为初始态，如果在栈扫描过程中，某个方法存在于栈上(即被扫描到)，那么它的状态将按照图 2 中所示的虚线箭头指示进行迁移。按照状态机所示，如果一个方法处于“初始态”，且被扫描到，那么它的状态会立即被置为“已扫描”；如果一个方法在前一次垃圾回收中未被扫描到，即其当前状态为“即将卸载”，但是在此次垃圾回收栈扫描时被扫描，那么该方法的卸载状态会被还原为“初始态”；如果一个方法(递归方法或者在垃圾回收发生时，在

多个线程上同时执行的方法）在此次扫描中出现多次，那么该方法的卸载状态将保持在“已扫描”状态。栈扫描方法的伪代码如图 3 中 scanMethod 所示，其算法思想如下：

1）初始化方法卸载状态为初始态

2）栈扫描：对栈上每个方法

a）如果方法的卸载状态为初始态，则将其修改为已扫描；

b）如果方法的卸载状态为已扫描态，则将其修改为已扫描态；

c）如果方法的卸载状态为即将卸载态，则将其修改为初始态；

从上述算法思想可以看到，当一个方法被扫描到时，其状态按图 2 所示状态机从左到右依次做一回迁移。需要说明的是，如图 2 中虚线箭头所示的状态迁移并不是一个完整地状态机。因为图中虚线箭头至少没有终止状态。完整的状态变换还需要下一步骤的介入。

[0018] 步骤二、代码卸载与第二次状态迁移

经过步骤一，被扫描到的方法的卸载状态已经发生了改变，没有被扫描到的方法则保持其原有状态。依据方法的最后状态，本发明选择在扫描之后垃圾回收结束之前对方法进行卸载。卸载的过程的伪代码如附录中第二个方法 performCodeUnload 所示，其算法思想为：

对于方法列表中的每个方法：

1）如果该方法的卸载状态为即将卸载，则将方法的机器码地址修改为引导重编译的代码片段（Code Stub）的地址，将代码的卸载状态修改为已卸载，并将代码所占内存地址释放；

2）如果方法的卸载状态为初始态，则将其修改为即将卸载；

3）如果方法的卸载状态为已扫描，则将其修改为初始态。

[0019] 从上述算法思想可以看到：1）此次扫描到的方法必定不会被回收；2）此次垃圾回收没有扫描到的方法，如果其状态为即将卸载，那么，该方法会立刻被卸载；3）此次扫描到的方法不会被卸载，其卸载状态按图 2 所示状态机从右到左依次做一回迁移。

[0020] 附录

```
Void scanMethod(Method* method){
if(method->status == METHOD_TO_UNLOAD)
method->status = METHOD_INITIAL;
else if(method->status == METHOD_INITIAL)
method->status= METHOD_SCANED;
else if(method->status == METHOD_SCANED)
method->status = METHOD_SCANED;
}
VoidperformCodeUnload(){
  Method* method = startMethod;
while( method != NULL){
if(method->status == METHOD_TO_UNLOAD){//unload this method
method->codeAddr = (void*)codeStub;
method->status = METHOD_UNLOADED;
```

```
free(method->allocatedAddr);
//recycle of code memory
}
else if(method->status == METHOD_INITIAL)
method->status = METHOD_TO_UNLOAD;
else if(method->status == METHOD_SCANED)
method->status = METHOD_INITIAL;
method = nextMethod;
}
}
```

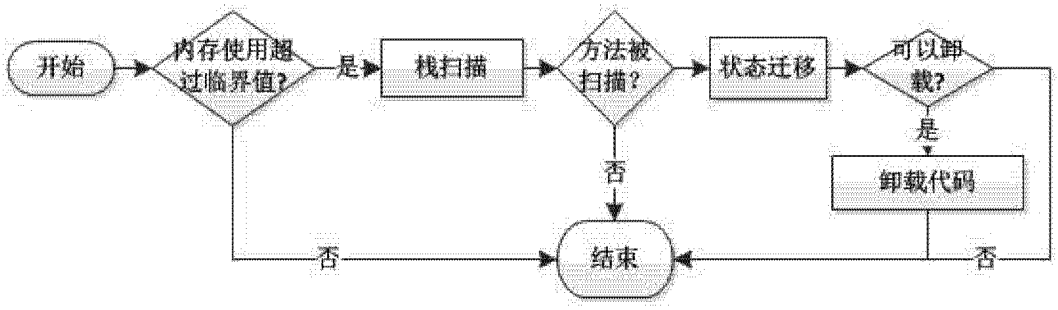

开始
内存使用超过临界值?
是
栈扫描
方法被扫描?
状态迁移
可以卸载?
是
否
卸载代码
否
结束
否

图 1

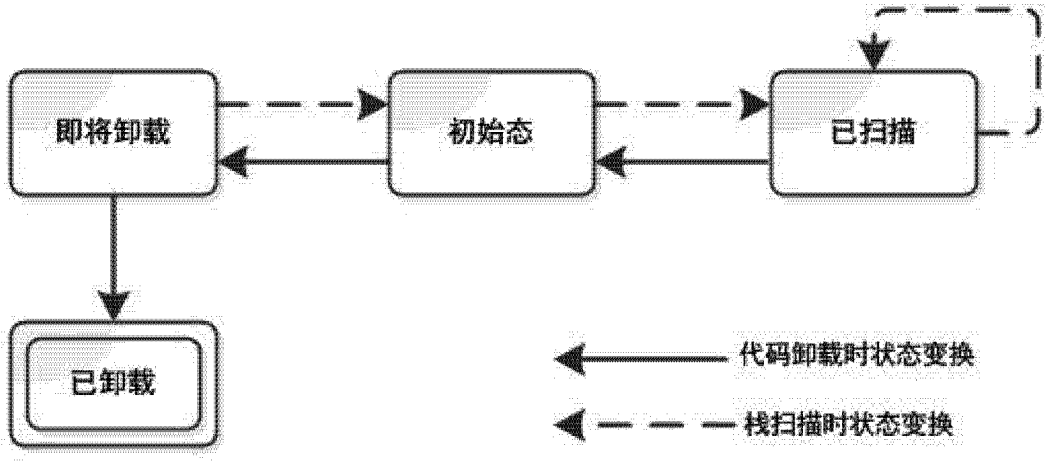

即将卸载
初始态
已扫描
已卸载
代码卸载时状态变换
栈扫描时状态变换

图 2