面向普适计算终端的 Java 虚拟机性能优化技术研究*

李　允[1]，罗　蕾[2]，雷昊峰[2]，熊光泽[2]

(1. 西南交通大学 计算机与通信工程学院，四川 成都 610031；2. 电子科技大学 计算机科学与工程学院，四川 成都 610054)

摘　要：分析了解释运行中利用线索化方法进行性能优化的技术,实现了以直接线索化方法为基础的 Java 虚拟机的解释器性能优化方案,并对嵌入式 Java 虚拟机的参考实现和 Java 的优化实现进行了性能对比。

关键词：普适计算；Java 虚拟机；线索化方法；直接线索化方法；性能优化

中图法分类号：TP316；TP273　　**文献标识码**：A　　**文章编号**：1001-3695(2005)03-0055-03

Performance Optimization Technology Java Virtual Machine for Pervasive Computing Terminals

LI Yun[1]，LUO Lei[2]，LEI Hao-feng[2]，XIONG Guang-ze[2]

(1. *School of Computer & Communications Engineering, Southwest Jiaotong University, Chengdu Sichuan* 610031, *China*; 2. *School of Computer Science & Engineering, University of Electronic Science & Technology, Chengdu Sichuan* 610054, *China*)

Abstract: The paper analyzes a performance optimization technology uses the threaded method in interpreting running modes and implements a performance optimization of Java VM based on direct threaded technology. The paper demonstrates the comparison performance of embedded Java VM and optimized VM based on the reference implementation of J2ME.

Key words: Pervasive Computing; Java VM; Threaded Method; Direct Threaded Method; Performance Optimization

1　引言

随着计算机技术的深入发展,逐渐呈现出了普适计算的模式[1,2]。普适计算的目的在于突破基于桌面系统的计算模式,使人们能够在任何情况下获得计算服务。在普适计算模式下,PDA 和移动电话、家用电器等通用设备,以及用于医疗、军事、娱乐等方面的专用设备将协同工作,以快速、高效和便捷的模式为人们提供服务。Java 由于其平台无关特性及其庞大的应用开发团队等显著特点,逐渐在移动终端领域形成了强大的需求,Java 技术本身也被称为普适 Java(Pervasive Java)。

为满足服务使用的便捷性,普适计算设备应该能够工作于移动环境。因此,与固定设备相比,普适计算设备主要具有资源有限等局限性。资源有限性主要是由于终端在设计方面存在以下需求:功耗低,重量轻和物理尺寸比较小。根据这些需求设计出来的设备所拥有的资源在以下方面存在不足:磁盘容量、物理内存大小、处理能力、缓存大小和屏幕大小。尽管随着技术的发展,普适计算终端的资源会得到不断增强,但固定设备同样也能从技术发展中获益,且固定设备不用考虑大小、功耗和重量等约束[3]。因此,资源有限性是普适计算终端的固有特性,技术的发展过程也不能弥补它与固定设备之间的差距。为此,形成了针对移动终端的、适合于嵌入式应用的 Java 技术——J2ME。但对于日益增长的复杂应用来说,J2ME 仍要求终端具有比较丰富的资源。因此,对终端的 Java 进行性能优化,不但能降低 Java 对资源的需求,满足更广泛普适应用的需要,还能有效降低 Java 应用的功耗情况,并使普适计算终端进行电子商务、电子银行和 3D 游戏等复杂应用成为可能。由于解释运行的虚拟机存在系统简单、容易实现等优势,且由于编译运行存在重复编译、编译过程需要消耗大量资源等方面的不足,使得嵌入式应用中大多采用 Java 的解释运行方式。

收稿日期：2004-02-09；修返日期：2004-05-23

基金项目：国家“863”计划资助项目(2002AA1Z2305)

2　基于解释执行的线索化优化技术

在程序的解释执行中,通常在解释器中采用 Switch-case 的解释处理方式,如下程序段通过 Switch-case 方式实现的解释器比较简单,易于移植和理解,但由于不少指令的解释程序比较简单(只有几条到十几条机制指令),使得 Switch-case 语句本身的执行开销太大,直接影响到解释执行的效率。

```
uint8 bytecodes[ ] = { OP LDC, OP ILOAD, OP ADD, OP ISTORE, . . . } ;
uint8 * pc = bytescodes;
while ( notEnd) {
    switch( * pc + + ) }
        case OP LDC:
          interpret LDC;
          break:
        case OP ILOAD:
          interpret ILOAD;
          break;
        case OP ISTORE:
          interpret ISTORE;
          break;
        case OP ADD:
          interpret ADD;
          break;
        …
    }
}
```

2.1 线索化的解释器

作为对传统的 Switch-case 解释执行方式的改进,线索化的解释器[4,5] (Threaded Interpreter) 最早在 Forth 语言解释器中得到应用。线索化的解释器意味着解释器对相邻字节码的解释执行过程通过一定的方式直接联系在一起,而 Switch-case 解释器每解释完一条字节码都要重新经过由 Switch 进行解释分发的过程。在 C/C++ 语言中,可以用 Goto 语句跳转到特定的标号。虽然 Goto 语句在结构化编程和面向对象编程的过程中都不提倡使用,但在实现解释器时,Goto 语句却可以发挥独到的作用。通常,Goto 跳转的地址在编译时就已经确定,不过,大多数 C 语言的编译器允许在程序中对标号寻址,使得 Goto 语句的跳转地址可以在运行时动态确定,如下程序段:

```
void * labels[ ] = { &&label1, &&label2, &&label3} ;
label1:
…
label2:
…
label3:
…
goto labels[ 1] ;   /* 跳转到 label2 * /
…
goto labels[ 1] ;   /* 跳转地址动态决定,i = 0 时跳转到 label1,以此类推 * /
```

线索化解释器正是基于这种特性,将所有字节码对应的解释程序的标号地址保存在一个标号数组中。然后,解释器以字节码的值作为标号数组的下标,直接跳转到对应的解释程序。在每个字节码解释执行完毕后,直接跳转到下一条字节码对应的解释程序。如下的伪码表示了线索化解释器的工作过程:

```
#define OP LDC      0
#define OP ILOAD    1
#define OP ISTORE   2
#define OP ADD      3
#define OP SUB      4
…
uint8 bytecodes[ ] = {
OP LDC,
OP ILOAD,
OP ADD
Op ISTORE,
…
} ;
void* labels[ ] = {
&&OP LDC LABEL,       /*  OP LDC = 0 * /
&&OP ILOAD LABEL,     /*  OP ILOAD = 1 * /
&&OP ISTORE LABEL,    /*  OP ISTORE = 2 * /
&&OP ADD LABEL,       /*  OP ADD = 3 * /
&&OP SUB LABEL,       /*  OP SUB = 4 * /
…
} ;
```

```
uint8* pc = bytescodes;
goto* labels[ * pc + + ]

OP LDC LABEL:
    interpret LDC;
    goto* labels[ * pc + + ] ;
OP ILOAD LABEL:
    interpret ILOAD;
    goto* labels[ * pc + + ] ;
OP ISTORE LABEL:
    interpret ISTORE;
    goto* labels[ * pc + + ] ;
OP ADD LABEL:
    interpret ADD;
    goto* labels[ * pc + + ] ;
OP SUB LABEL:
    interpret SUB;
    goto* labels[ * pc + + ] ;
    …
```

与 Switch-case 解释器相比,线索化解释器省掉了以下的开销:①每条字节码解释完毕后,到 Switch 的跳转指令;②Switch 在查找跳转表之前对字节码的值进行边界检查的指令。并且,由于许多常用字节码本身的解释程序很短(甚至不到 10 条机器指令),线索化解释器的优化效果相当明显。另外,线索化解释器也没有增加内存占用上的开销。通过实际的实验验证,线索化解释器的运行速度比 Switch-case 解释器快 7% ~10%。

2.2 直接线索化的解释器

线索化解释器将字节码的处理程序的入口地址集中保存在标号数组中,解释执行字节码时需要至少两次内存访问:①读入 PC 指向的字节码;②以字节码作为下标,读入对应的处理程序的地址。直接线索化解释器[5] (Direct Threaded Interpreter) 为对线索化解释器的一种改进方案。在运行字节码之前将所有字节码所对应的处理程序的地址保存在一个数组中。在解释运行时不再访问原来的字节码数组,而是直接访问地址数组,因而在线索化解释器的基础上减少了一次内存访问的开销。直接线索化解释器的实现可以用如下所示的伪码表示:

```
#define OP LDC      0
#define OP ILOAD    1
#define OP ISTORE   2
#define OP ADD      3
#define OP SUB      4
…
uint8 bytecodes[ ] = {
OP LDC,
OP ILOAD,
OP ADD
Op ISTORE,
…
} ;
void* labels[ sizeof( bytecodes) ] ;
labels[ 0] = &&OP LDC LABEL;
/* 对应 bytecodes[ 0] = OP LDC * /
labels[ 1] = &&OP ILOAD LABEL;
/* 对应 bytecodes[ 1] OP ILOAD * /
labels[ 2] = &&OP ADD LABEL;
/* 对应 bytecodes[ 2] OP ADD * /
labels[ 3] = &&OP ISTORE LABEL;
/* 对应 bytecodes[ 3]  OP ISTORE * /
…
```

```
void* instructionLabel = &labels[ 0] ;
goto* instructionLabel + + ;
OP LDC LABEL:
    interpret LDC;
    goto* instuction Label + + ;
OP ILOAD LABEL:
    interpret ILOAD;
    goto* instuction Label + + ;
OP ISTORE LABEL:
    interpret ISTORE;
    goto * instuction Label + + ;
OP ADD LABEL:
    interpret ADD;
    goto * instuction Label + + ;

OP SUB LABEL:

    interpret SUB;

    goto * instuction Label + + ;

…
```

与 Switch-case 解释器执行过程相比,直接线索化解释器减少的开销包括一次跳转,一次访存和对字节码的值进行的边界检查操作。通过实际的实验结果表明,直接线索化解释器的速度比 Switch-case 解释器快 40% ~100%。但是,由于需要额外的内存空间来存放解释程序的地址数组,直接线索化解释器的内存开销要大于 Switch-case 解释器和线索化解释器(需要约 4 倍于字节码自身大小的内存空间)。直接线索化解释器在每条字节码解释完后仍然需要一条跳转指令,转到下一条字节码对应的解释程序入口。这样频繁的跳转对 CPU 的指令 Cache 和分支预测机构仍然会带来很大的影响。内联线索化的解释器[6] (Inlined Threaded Interpreter) 为直接线索化解释器的改进方案,将若干连续字节码对应的解释程序本身(而不是标号)复制到一个连续的地址空间中,然后运行这一段解释程序,这样就完全消除了在字节码解释程序间的访存和跳转指令的开销。可以看出,内联线索化解释器以一种简化的方式实现了类似 JIT 的代码生成工作。其代价是需要占用大量的内存(需要约数十倍于字节码自身大小的内存空间)。

3 普适计算终端 Java 虚拟机性能优化技术

3.1 直接线索化优化

采用直接线索化解释器作为 Java 虚拟机解释器的核心实现方案,其原因是:①Switch-case 方式实现的字节码解释器性能过于低下;②线索化解释器与 Switch-case 解释器相比性能有一定提高(7% ~10%),但提高幅度仍然有限;③内联线索化解释器与 Switch-case 解释器相比性能有很大提高(150% ~300%),但内存占用太大;④直接线索化解释器与 Switch-case 解释器相比性能有显著提高(40% ~100%),同时仍然保持较少的内存占用。

解释器在运行字节码之前把字节码数组翻译为直接包含字节码对应解释程序入口的线索化代码数组,如图 1 所示。

实现以上翻译过程的代码可以用如下伪码表示(翻译程序按顺序读入每一个字节码,查找其对应的解释程序入口地址,再将地址写入线索化代码数组中):

```
void* opcode entries[ MAX OPCODE] ;   /* 保存字节码解释程序入口地址* /
/*   初始化解释程序入口地址表   * /
opcode entries[ OP CODE1] = &&OP1  ENTRY;
opcode entries[ OP CODE2] = &&OP2  ENTRY;
opcode entries[ OP CODE3] = &&OP3  ENTRY;
```

```
...
/*  将字节码翻译为 threaded code */
uint8* pc = bytecodes[];
int bytecodes length = length of bytecodes;
void** threadedPC = threadedCodes[];
while (bytecodes length--) {
    if (IS OPCODE(*pc)) {
        /*  该字节是操作码,翻译为解释程序入口地址 */
        *threadedPC = opcode entries[*pc];
    } else{
        /*  该字节是操作码的参数,直接保存 */
        *threadedPC = *pc;
    }
    threadedP C++;
    p C++;
}
```

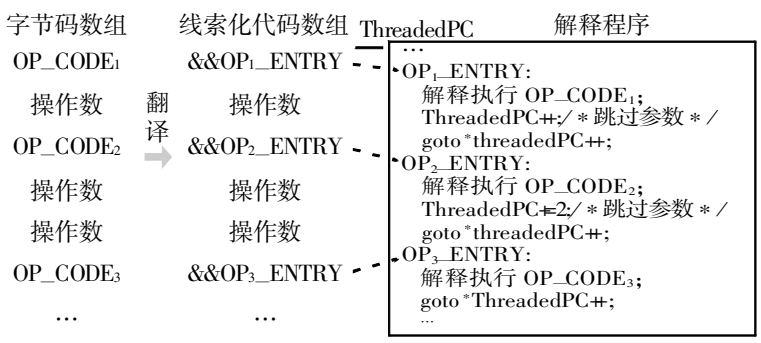

图 1　线索化代码数组

实际上,图 1 和上述的伪码只考虑了最简单的字节码翻译过程,即每个字节码对应线索化代码的一条数据(通常为 4 字节),其中操作码对应翻译程序标号地址,操作码后续的操作数则直接复制到线索化代码中。无论是操作码还是操作数,每一个字节码在翻译后都要占用 4 字节的存储空间。在实际的翻译过程中,有两个问题需要解决:①操作数的翻译;②相对跳转地址的处理。

在 32 位 CPU 上,标号地址(类型为 void*)长度为 32 位,占用四个字节的存储器。一条 Java 操作码的后续操作数长度从一字节到数十字节不等,这些操作数有的是 16 位整数,长为两个字节;有的是 32 位整数,长为四个字节。如果按照跳转地址动态确定的翻译方式,16 位的操作数在翻译后将占用两个线索化代码单元,即八个字节;而 32 位操作数则要占用 16 个字节。并且,在线索化代码运行时,还需要重新组装这些操作数。这样做既增加了无谓的内存占用,又降低了运行效率。因此,在翻译过程中,通常是将一个操作数的若干个字节码合并在一条线索化代码单元中,如图 2 所示。

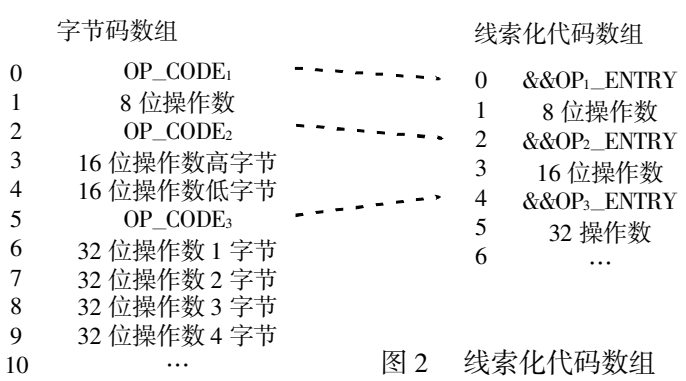

图 2　线索化代码数组

如图 2 所示,在翻译时对字节码操作数进行合并处理,可以在一定程度上降低线索化代码的内存占用,同时提高了运行效率。可以看出,合并操作数使得字节码数组与线索化代码数组的下标不再具有一一对应关系。对顺序执行的字节码而言,这种变化没有影响,但对于进行相对地址跳转的字节码而言,情况就发生了变化。以图 2 为例,假设字节码数组中下标为 2 的 OP_CODE_2 是一条跳转指令,其后的操作数是相对地址,值为 8,则 OP_CODE_2 的执行结果为跳转到下标为 10 的指令处。而在线索化代码中,OP_CODE_2 的解释程序则应该跳转到对应的指令,即下标为 6 的线索化代码处。因此,在对跳转指令进行翻译时,必须对操作数进行调整。

3.2　字节码替换优化

字节码替换主要包括以下几个方面的内容:

(1) 将某些不指定操作数类型的指令细分并改写为针对特定类型的指令。例如,LDC 指令将常量表中指定的常量压入 Java 堆栈,它的操作数(指定常量的下标)也是一个常量。在翻译代码时可以得知不同的 LDC 指令所装入常量的数据类型,并将 LDC 指令细分为 LDC_INT, LDC_FLOAT, LDC_STRING 等指令,以提高运行效率。

(2) 将常量操作数直接嵌入在线索化代码中。每一个 Java 方法都有自己的常量表,程序运行时通过 LDC 指令和一个下标操作数来将常量表中指定的常量压入 Java 堆栈。可以将长度不超过四个字节的常量直接嵌入到线索化代码中,如整型常量、字符串指针常量等,这样可以减少一次查表的开销。

(3) 合并某些功能相近或相同的指令。例如,BIPUSH 和 SIPUSH 指令分别向一个 8 位整数和 16 位整数压入堆栈,考虑到 8 位和 16 位整数在线索化代码和堆栈上实际都要占用四个字节,这两条指令可以由同一段解释程序来执行。

3.3　线索化代码存储空间优化

字节码翻译为线索化代码后,需要占用原字节码大小 3 ~ 4 倍的存储空间。相对内联线索化解释器而言,开销已经大大减小,但对于内存资源有限的移动终端而言,这样的开销仍然不可忽视。线索化代码存储空间优化技术即用来降低线索化代码的内存占用情况,其工作原理是:

(1) 开辟一块专门用于存放线索化代码的内存空间;

(2) Java 方法只在被运行之前才翻译为线索化代码;

(3) 通过 LRU(最近最久未用)表对已翻译为线索化代码的 Java 方法的被调用情况进行记录;

(4) 当存放线索化代码的内存空间不足以容纳新的需要翻译的 Java 方法时,在 LRU 表中查找最近最久未调用的线索化代码,并释放这段代码。有必要的话,重复这个过程直到可以容纳新的 Java 方法为止。

通常由于移动终端本身存储能力的限制,Java 应用代码的规模通常不会超过 100KB,大多数常用应用不会超过 30KB ~ 40KB。在这个前提下,通过 LRU 表来维护线索化代码的内存空间,能够大大降低线索化代码所占用的内存。

3.4　提高 CPU 指令 Cache 命中率

Java 虚拟机一共定义了 204 条指令,这些指令在 Java 程序中出现的频率是不一样的。根据 Radhakrishnan[7] 的统计,Java 程序中使用频率最高的 15 条指令,其运行次数占所有指令运行次数的 56% ~85%。这些使用频率最高的指令随程序不同而不完全一致。但总的来说,常量表访问指令、变量表访问指令、算术运行指令和分支指令的使用频率相对其他指令要高得多。为此,将使用频率高的指令的解释程序安排在相邻的地址空间中,可以提高 CPU 指令 Cache 的命中率,进而改善虚拟机的性能。

4　性能分析

Java 虚拟机性能优化是在嵌入式 Java——　(下转第 79 页)