第31卷 第16期 计 算 机 工 程 2006年8月
Vol.31 № 16 Computer Engineering August 2006
·软件技术与数据库· 文章编号：1000—3428(2006)16—0052—03 文献标识码：A 中图分类号：TP 311

J2ME MIDP中RMS的设计实现与性能优化

李 莹，王 昕，毛迪林，高传善
（复旦大学计算机科学与工程系，上海 200433）

摘 要：移动信息设备框架（MIDP）是J2ME中一套重要的类库。作为MIDP的一部分，记录管理系统（RMS）实现了在手机等微小型系统上持久存储的能力。在Intel开发的虚拟机(XORP)和开发的CLDC的基础上，实现了MIDP中的RMS模块。该文介绍了对记录管理系统的设计思想，比较了两种不同设计方案的优劣，并介绍了实现细节。进一步分析了在性能优化方面做的工作，并将最终结果与Sun的实现作了比较。

关键词：J2ME；移动信息设备框架；记录管理系统；性能优化；记录

RMS Design and Performance Tuning in J2ME MIDP

LI Ying, WANG Xin, MAO Dilin, GAO Chuanshan
(Department of Computer Science and Engineering, Fudan University, Shanghai 200433)

【Abstract】 Mobile information device profile (MIDP) is an important class library in J2ME technology. As part of MIDP, record management system (RMS) provides a mechanism for persistent storage on small, handheld devices with limited capabilities. This paper designs and implements RMS of MIDP based on Intel's virtual machine named XORP, as well as own developed CLDC. This paper first compares the two initial designs of RMS, and then gives a description of the final implementation. After that, it presents the performance tuning techniques, and compares both pre- and post-tuned performance with Sun's WTK.

【Key words】 J2ME; Mobile information device profile (MIDP); Record management system (RMS); Performance tuning; Record

Sun公司的Jave2体系结构由3个不同层次组成：J2EE，J2SE，J2ME。J2EE强化支持企业内部的扩充API；J2SE定位在客户端的应用上；J2ME适合有限空间和有限处理器的微小型设备。

J2ME的体系结构可以分为4层。最底层是Java虚拟机。Sun使用的是K Virtual Machine (KVM)，我们使用的是Intel的XORP。次底层是配置层（configuration layer），它定义了Java虚拟机的功能和特定类别设备上可用的Java类别库的最小集；第2层是框架层（Profile Layer），它定义了特定系列设备上可用的应用程序接口（API）的最小集；最高层是MIDP层(Mobile Information Device Profile)，它是一个Java API集合，处理诸如使用者界面、持久存储和网络连接这样的问题。

记录管理系统是MIDP的一部分，提供了一组用于组织和操作设备资料库的类别和界面。它为MIDP应用程序(MIDlet)提供了一种跨多个调用持久存储数据的机制。这种持久存储机制可被视为一种简单的面向记录的数据库模型。

1 RMS包分析

RMS包位于javax.microedition.rms，共有10个类：其中，1个public class，4个interface，5个exception。

在RMS中，有两个重要的概念：Record Store，Record。

1.1 RecordStore

一个Record Store由一些记录的集合组成，供一个或多个MIDlet调用。

Record Store在平台相关位置被创建。一个MIDlet Suite中的MIDlet可创建多条不同名的record store。删除MIDlet suite将删除所有与此MIDlet相关的record store。同一个MIDlet suite中record store可共享。若在创建时给予权限，record store也可通过API显式地在不同MIDlet suite中共享。

每个RecordStore命名唯一，规则为MIDlet suite名加上RecordStore名，大小写敏感，最长32个字符。在同一个MIDlet Suite中，Record store的命名必须唯一。

API中并没有要求互锁。Record store的实现保证所有单个的操作都是不可分的、可串行的。但是，当一个MIDlet使用多线程来读写一个record store时，控制读写的过程应该由MIDlet本身来完成。

Record Store使用长整型来记录最后修改的时间/日期，用整型来记录版本信息。

1.2 Record

Record是由一串byte组成的。在一个record store中，每个record使用recordId唯一的标识。RecordId是一个整数，被用作记录的主键。第1个记录创建的时候，recordId为1，以后每个recordId递增1。

2 RMS设计思想

下面比较几种RMS的实现以及它们的利弊。

2.1 固定长度的记录

对于record store和record的实现，方法是使用固定长度的记录(图1)。

基金项目：Intel公司基金资助项目“J2ME Class Libs with Small Footprint，Low Power and High Performance on XScale Processor”
作者简介：李 莹(1981—)，男，硕士生，主研方向：计算机网络和信息工程；王 昕，硕士生；毛迪林，讲师、博士；高传善，教授、博导
收稿日期：2005-10-13 **E-mail：**yingli@fudan.cn

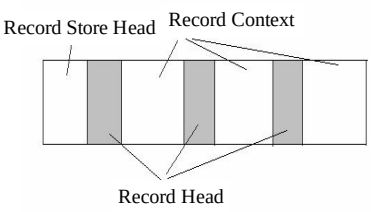

图 1 固定长度记录

对于固定长度的记录，在 record store 头部保存 record store 的相关信息，包括当前存在的记录个数、版本信息、签名、记录的统一长度等。记录着针对所有记录的整体信息。在 record store 的头部之后，也就是主体部分中，保存各个 record 的具体信息。Record 又包括记录头(record head)和 record 内容(record context)。由于每条记录采用固定长度，下一条记录的起始位置不需要指明，因此 record head 只需保存相关记录的记录号和实际记录长度。如果某条记录特别大，一块连续存储区域无法存下，需指明下一块固定长度存储区间的位置偏移量。record context 保存需要存储的记录的内容。

对于使用固定长度的记录，其好处是每条记录在 record store 中的偏移量是固定的。这样，就可以节省一个标志每条记录起始位置的 offset 信息，同时，记录的查找定位可以使用 Hash 方法快速地实现。另外，由于每条记录占据相同的固定空间，插入、删除操作相对比较简单，不涉及记录的移动、空间的再分配等问题。

与此同时，固定长度也有它的不足。比如，如果记录的长度相差很大，则短记录对存储空间的利用率很低。记录长度的参数很难配置。

考虑到 MIDP 在有限存储的环境中使用，而且记录的长度根据不同的应用相差很大，使用固定长度会造成很大的存储空间浪费，所以，我们没有使用这种方法。

2.2 可变长度的记录

在实现中，使用可变长度的记录(图 2)。

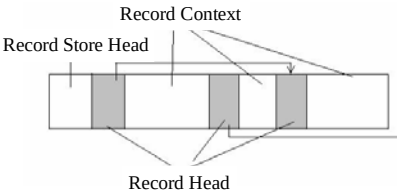

图 2 可变长度记录

对于可变长度记录，在 record store 头部存放 record store head 信息。除了包括当前记录总数、版本信息、签名、最后更新时间等全局信息外，record store head 还要维护两个链表的表头。两个链表分别是 record 节点链表和空闲节点链表。Record 节点链表的表头存储第 1 条记录的起始位置；空闲节点链表的表头存储第 1 块空闲存储区间的起点位置。初始时，二者都是 record store head 后的第 1 个字节。

每条 record 有一个 record head，它存有当前记录的记录号、下一条记录的起始位置（next offset）、记录的长度等信息。在 record head 之后分配一块连续的空间存储记录的内容（record context）。下一条记录可以通过上一条记录头中的 next offset 信息找到起始位置。

可变长度记录的优点是：存储空间根据记录需要而分配，不会造成浪费，空间利用率高。一条记录不会被分成多段。

缺点是查找记录时需要使用链表，效率相对较低，此外，频繁地插入删除可能造成存储空间中过多的小碎片，造成空间利用率下降，需要进行存储空间的整理。

尽管如此，在存储空间很紧张，但记录总数不多，遍历链表代价不大的情况下，可变长度的记录仍然是相对较好的选择。并且通过一定的优化，可以极大地减小遍历链表的代价。在我们的实现中，选择了可变长度的记录。

2.3 插入记录

插入记录可以通过 addRecord()方法实现。插入记录时，首先判断当前 record store 是否允许进行插入操作。如果可以，判断需要插入的内容是否合法。如果以上条件都满足，在空闲节点链表中查找第一个长度足够大的节点，将原空闲节点一分为二，前一半写入需要插入的节点的 record head 和 record context，并将相应的内容挂在记录节点链表的表头；后一半作为新的空闲节点，其长度为原长度减去新插入记录的长度，并在空闲节点链表上更新。

算法伪码如下：

```
判断 record store 允许插入操作
判断插入的记录合法
  if (record store 非空)
在空闲节点链表上查找可插入的第一个节点
插入节点，offset 放在 record 链表表头；
将空闲节点剩下的部分作为新的空闲节点；
  if (record store 为空)
在记录节点链表上插入第一个节点；
更新 record 链表表头；
更新空闲列表表头；
      更新 record store head；
```

2.4 删除记录

在删除记录时，首先判断 record store 是否打开，是否允许删除，然后找到需要删除的记录，将记录节点从记录节点链表上摘下，放入空闲节点链表。

伪码如下：

```
判断 record store 是否打开，是否允许删除
查找需要删除的记录是否存在
如果删除这一记录后 record store 还有记录
    将此节点从 record 链表中摘下
    将此节点放入空闲节点链表
 如果删除这一记录后 record store 无记录
     重置两个链表。一些特殊操作。
更新 record store head；
```

2.5 查找记录

查找记录时，顺序遍历记录节点链表，找到第 1 个符合查找条件的记录，或者到达链表尾，抛出记录未找到的异常。

2.6 更新记录

对于更新操作，首先找到需要更新的记录，执行一次插入新记录的操作，再执行一次删除旧记录的操作。

2.7 其他方法

根据 MIDP2.0 的 spec，按照 clean room 的原则，写出其他的公有、私有类的实现。

3 实现中的性能考虑

在实现完第 1 个版本后，将我们实现的 RMS 性能和 Sun 的版本作了比较，我们使用的 benchmark 工具是 open source 的 TaylorBenchmark，通过在其中添加控制循环次数和记录时间的语句，比较两种不同版本的性能。

起初，我们版本的性能并不是特别理想，所以做了代码性能优化。性能优化从以下几个方面展开：

(1)避免频繁地打开/关闭 record store

第 1 个版本很大的性能损失在于频繁地对 record store 进行打开和关闭操作。每个 record store 对应于一个文件，通常，一系列操作会针对一个固定的 record store 反复进行，这样，只需要在一系列操作前打开文件，维持文件打开的状态，执行一系列操作，与这个文件相关的操作全部结束后关闭文件。这样，只需要在打开 record store 时打开文件，并保持打开状态，直到关闭 record store 时关闭文件。

(2)减少 native 的调用和参数传递

使用 JNI 调用 native 方法是一种开销十分巨大的操作，所以，尽可能减少 native 调用，避免反复调用 native 或顺序调用一系列 native，尽可能减少 native 中需要传递的参数，以提高性能。

(3)将新插入的记录放在链表表头

我们发现操作记录时，往往对最近记录的操作比较频繁。比如，在手机游戏中，往往读取的是最近存储的记录；下载文件或播放视频时，往往也是读取最新存储的内容。所以，我们采取的策略是，添加记录时，将新记录放在链表的表头，而不是表尾，这样，每次查找记录、读取记录、或者删除记录时，就可以以最小的代价遍历链表，而不必每次都遍历到表尾才找到最新的记录。实际上，由于采取这种策略，链表平均遍历的节点数只占链表节点总数的很小一部分。

(4)减少不必要的 synchronize

synchronize 在 java 中是一个很耗时的操作，有些 synchronize 由类库来保证，有些 synchronize 由 application 来保证。

4 实验

我们的实验平台是 PIII 800, 384MB SDRAM，Win2k 操作系统。平台是 Intel 的 XORP，以及我们开发的 CLDC、MIDP，比较对象是 Sun 的 JVM 和 WTK。测试工具是开源的 Taylor Benchmark，并在其中加入循环控制和计时语句。

测试了添加新记录（add Records），顺序读取记录（Enumerated Record Read），随机读取（Random Read Record），删除记录（Delete Records）4 种操作。

在测试添加记录时，先建立一个空的 record store，每次为 record store 添加 100 条记录，每条记录 100B。共循环执行 10 次，并将 10 次的结果求平均值，分别得到 WTK 添加 100 条记录的平均时间，优化前的 RMS 添加 100 条记录的平均时间，优化后的 RMS 添加 100 条记录的平均时间。单位均为 millisecond。

结果如图 3 所示。

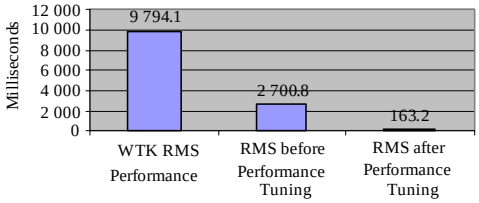

图 3 插入 100 条记录的平均时间

插入操作比较的另一个方面是文件的大小，也就是额外维护信息在总存储信息中占的比例，如果比例越小，则存储空间有效利用率越高。我们在进行 10 次插入操作，每次插入 100 条 100B 的记录后，比较 Record store 的大小，结果如图 4 所示。其中，有效内容为 100 * 100 * 10 = 100kB，其余部分为维护信息，如 record store head，record head 等。

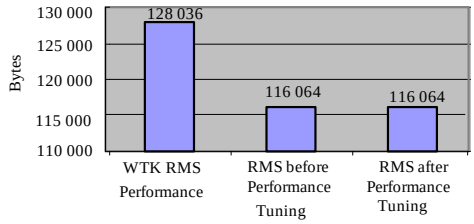

图 4 插入 1 000 条记录后的文件大小

对空的 record store 进行插入操作，和对已有不同数量的记录的 record store 进行插入操作，其时间代价也有可能会不相同。我们记录了插入第 1~第 100 条记录，第 101~第 200 条记录，直到第 901~第 1 000 条记录的 10 段时间。结果如图 5 所示。随着已存在的记录数的增多，WTK 的插入代价有上升的趋势。

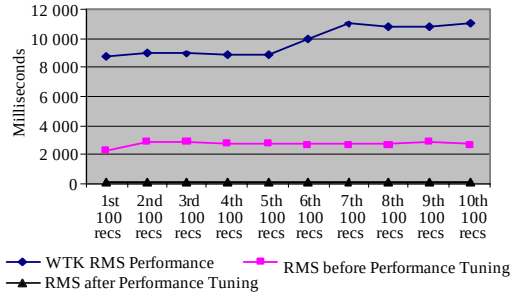

图 5 每次 100 条记录的插入时间代价趋势

对于读操作，我们分别测试了顺序读取和随机读取。

对于顺序读取，我们每次读取 100 条记录，每条记录 100 个字节。读取 10 次取平均值。

其时间比较如图 6 所示。

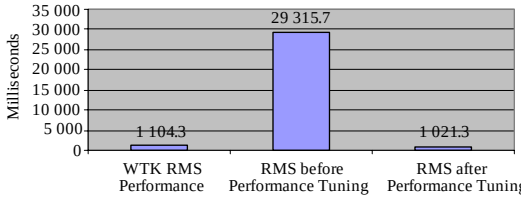

图 6 顺序读取 100 条记录的平均时间

对于随机读取，每次读取 100 条记录，每读取一条记录前先用系统当前时间作为 seed，生成一个随机数，并读取这条记录。测试 10 次取平均值。结果如图 7 示。

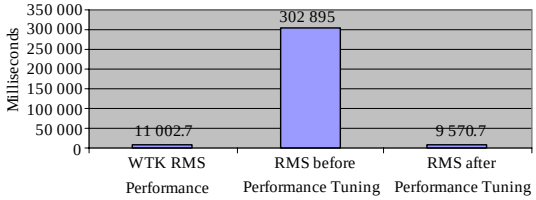

图 7 随机读取 100 条记录的平均时间

对于删除操作，我们每次删除 100 条记录，共操作 10 次取平均值，其结果如图 8 所示。

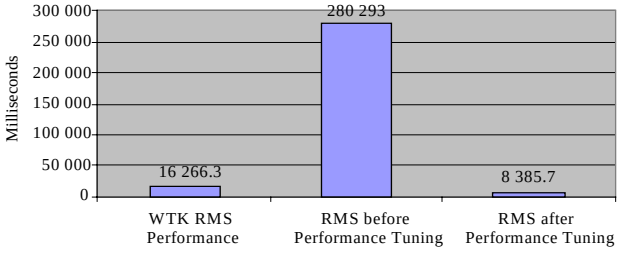

图 8 删除 100 条记录的平均时间

(下转第 69 页)