基于 Flash 实现 Java Card 事务带来的问题及其解决方案①

The Problem Using Flash in Java Card Transaction and its' Solution

李 栋 杨义先 (北京邮电大学国家重点实验室 100876)

摘要:通常 Java Card 使用的可读写存储介质是 EEPROM,由于写 Flash 的特殊性,导致使用 Flash 来实现事务时,断电时系统可能不能恢复到事务开始前的状态。本文针对 Flash 在断电恢复时出现的问题提出了相应的解决方案,并且得到了很好的实践验证。

关键词:Transaction(事务) EEPROM Flash 掉电

1 引言

Transaction(事务)的概念,我们在数据库操作中经常遇到。当我们要对一系列的相关数据进行操作时,因为其相关性,希望操作的结果是所有操作都按照我们的期望完成了。如果有一个操作发生了问题,那么其他操作的正确性就不能保证了,因此所有的数据要恢复到其原来的状态。在这种情况下,我们就要使用事务的机制,将以上的这些相关操作都放到一个事务中,如果所有操作成功了,那么我们就提交事务;如果有任何一个操作出现了问题,就要取消这个事务。

从上面的描述中可以看出,事务具有 Atomicity(原子性)的性质。原子性就是指一个操作或者成功,或者失败,如果失败就要恢复到操作之前的状态。

在 Java Card Specification 中规定了其事务和原子性的特点,因此需要在 Java Card 中实现事务的机制。

实现 Java Card 需要的存储介质包括 EPROM(E2P),ROM 和 RAM。E2P 是可重复擦写的存储介质;ROM 是可读而不可写的存储介质,用来存放实现 Java Card 的程序代码;RAM 是动态的可擦写的存储介质,用来实现 Java Card 的虚拟机的堆栈操作。在智能卡上经常使用的存储介质还有 Flash,它是类似于 E2P 的一种存储介质,因为其相同体积的存储容量比 E2P 大很多,而其造价比较便宜,所以在很多应用中可以替代 E2P。

2 事务的实现策略

事务的实现通常有两种方式(备份新值和备份旧值)。当我们需要对一些永久数据进行更新操作时,将这些操作放在同一个事务中。如果在事务的执行过程中发生了智能卡断电等事件时,我们就可以取消事务来恢复到其原始的状态。

为了达到这种目的,可以在更新数据时,首先将数据的原始值备份到事务缓存区中,然后再去更新数据,这样我们可以根据事务缓存区中备份的数据原始值来恢复数据到其原始状态。这种方式称为"备份旧值"的实现方式。在事务提交时,需要完成的工作就相当简单了,只需要清除事务缓存区即可。

另外,可以采用"备份新值"的实现方式。在更新数据时,将数据的新值存放到事务缓存区,对数据本身不作更新。这种方式对事务取消的处理相当简单,因为数据的原始值没有发生改变,只需清除事务缓存区即可。当事务提交时,处理工作就要复杂一些了,需要用事务缓存区的新值来更新数据。

事务缓存区是由一条条的记录组成的。每条记录

① 国家自然科学基金资助(60372094)

由标志位,备份数据的长度,备份数据的起始地址,备份数据的内容这几部分组成。标志位的作用在于区分一条记录是否有效。

表 1　事务缓存区的结构

	标志位	备份数据的长度	备份数据的起始地址	备份数据的内容
记录 0				
记录 1				
记录 2				
…				
…				
…				
记录 N				

使用“备份新值”的实现方式存在一个问题,如果一个事务过程中的操作需要读取某个永久数据(指存在于 E2P 中的数据)的值,那么我们需要查看这个永久数据是否存在于事务缓存区,以便读取其最新的状态,因而其实现比较复杂。

下面,来比较这两种实现方式的区别:

表 2　两种实现方式的比较

	备份新值	备份旧值
事务提交时的操作	复杂	简单
事务取消时的操作	简单	复杂
事务缓存区可以使用的存储介质	RAM,E2P,Flash	E2P,Flash
实现的复杂性	复杂	简单

其中值得注意的一点是,采用“备份新值”的事务实现方式可以使用 RAM 来做事务缓存区,将记录备份存放在 RAM 中。而使用“备份旧值”的实现方式却不能使用 RAM 来做事务缓存区。为什么有这种区别呢?

我们知道,RAM 在断电后数据都会被清除。E2P 和 FLASH 的数据在断电后不会被清除,而是保留为断电前的状态。

当突然断电时,RAM 上所有的数据都会丢失。如果采用“备份新值”在 RAM 上的实现方式,那么这些备份记录都会丢失,但是数据本身会保留其原始的状态,这恰恰符合事务取消的定义。如果采用“备份旧值”在 RAM 的实现方式,断电时,RAM 上的数据丢失了,也就是数据的原始状态丢失了;若这时要求恢复到数据的原始状态,就产生了矛盾。也就是说,“备份旧值”在 RAM 的实现方式,在事务取消时(掉电时),它不能将数据恢复到原始的状态。所以,“备份旧值”的事务实现方式不能使用 RAM 作为事务缓存区。

既然“备份新值”比“备份旧值”的实现方式复杂,那么为什么还会采用这种方式呢?原因在于其可以使用 RAM 作为事务缓存区。我们知道,写 RAM 的速度远远快于写 E2P 和 FLASH 的速度,而 Java Card 系统中会大量的使用到事务。如果使用 RAM 作为事务缓存区,那么 Java Card 的运行速度将会大大提高。但是,我们面临另外的问题,同等容量的 RAM 芯片面积要大于 E2P 或者 FLASH,因而其造价比较高,而且当前常用的 RAM 芯片容量都比较小,很难分出 1 到 2K 的空间来做事务缓存区。

实现事务的这两种方式,各有千秋。如果看重事务提交时的性能,那么需要使用“备份旧值”的方式;如果经常会出现事务取消的现象,那么采用“备份新值”的方式会取得更好的效果。

3　使用 Flash 实现事务带来的问题

Flash 与 E2P 相比,有价格低,容量大的优点。所以,在以后的应用中,Flash 会获得较大范围的应用。但是,使用 Flash 替代 E2P 作为存储介质,实现 Java Card 的事务机制会带来一些难以克服的问题。为什么呢?这时由于写 Flash 的特性造成的。

写 Flash 的特点如下:

(1) 只能单向写(从比特 1 到 0);

(2) 写的最小单位大于一个字节(以 Page(页面)为写的最小单位,一个 Page 由若干个字节组成)。

比如,要写 Flash 的一个字节,首先需要将这个字节所属的整个页面的内容拷贝到 RAM 中,然后在 RAM 中更新要写的那个字节,接着将 Flash 的这个页面的所有字节擦除为 0xFF,最后根据 RAM 的值,将 Flash 页面中的相应比特位从 1 写为 0。这样就完成了写 Flash 一个字节的工作。

使用 Flash 来代替 E2P 作为存储介质,如果使用“备份新值”的事务实现方式,那么事务缓存区可以使用 Flash 或者 RAM。如果使用“备份旧值”的方式来实现事务,那么事务缓存区只能使用 Flash。

当使用 Flash 作为事务缓存区,而且采用“备份旧值”的实现方式时,断电可能会对事务的恢复造成很大的麻烦。在某个特定的时刻发生断电时,事务就不能恢复到其原来的状态。

我们分析断电对事务造成的影响。断电发生的时机可以是:

断电现象 1:

正在“备份旧值”的时候,发生了断电。

断电现象 2:

完成了“备份旧值”,正在写备份记录的标志位,

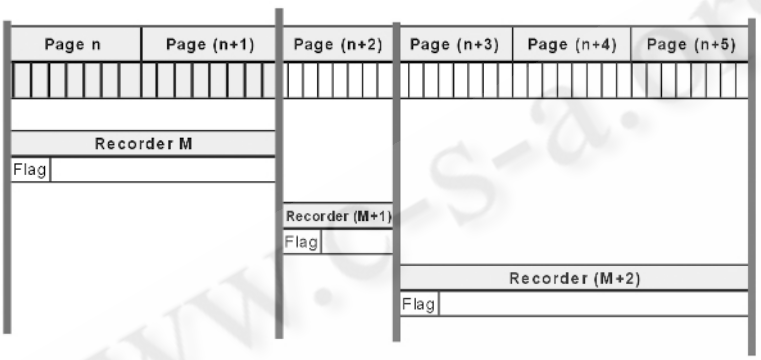

图 1　以页面为边界的事务缓存区结构

这时发生了断电。

对于“备份旧值”的实现方式,需要先在事务缓存区备份数据的旧值,然后才会更新数据。备份数据是通过一条条的记录来表示的。而在写入一条新的记录时,需要先写记录的“备份数据的长度”,“备份数据的起始地址”和“备份数据的值”这三个项目。当这三个项目成功写入之后,记录的标志位才会被设置。这时一条备份记录的建立工作才算完成。

只要一条记录的标志位没有被设置(来表示是否是一条有效记录),我们就不会更新数据的旧值。因此,当断电现象 1 发生时,数据还没有被更新。那么重新上电后,数据还是保持其原来的状态,这就满足了事务取消的要求。

4　解决方案

上述断电恢复的问题仅仅出现在使用 Flash 作为事务缓存区,而且使用“备份旧值”的事务实现方式这种特定的情况。因此,当使用 Flash 替代 E2P 作为存储介质时,事务的实现可以使用其它的实现方式来避免这个问题。

当我们使用“记录新值”的方式来实现事务,那么可以使用 RAM 来做事务缓存区,这就避开了使用 Flash 带来的难题。但是,我们知道 RAM 的造价比较昂贵,而且现有的芯片中 RAM 的存储空间比较小,很难拿出单独的几 K 的空间作为事务缓存区。而且作为 Java Card 的事务缓存区而言,其实现比较复杂。比如读取永久数据(存在于 E2P 或者 Flash 上的,掉电不会丢失的数据)时需要查找其是否存在于事务缓存区中,以便取得其最新的值。

所以,找到一种使用 Flash 作为事务缓存区的方案是必要的。

4.1　方案之一

一般 Flash 的一个 Page 包含有若干个字节,比如 8 个字节,16 个,32 个,64 个等。当 Page 包含的字节数目比较少时,比如 8 个或者 16 个,那么我们可以采取下面的方案 1。

方案 1:

(1) 每条记录都从一个 Page 开始;

(2) 记录占用的空间是整数个 Pages。

方案 1 如图 1 所示。

每条记录的起始都从一个 Page 开始,跨越整数个 Pages。这种方案会浪费一些空间,但是每条记录最多浪费的空间也不会超过 1 个 Page(反正 Flash 比较便宜,这些浪费算不上什么)。

这样就解决了断电现象 2 带来的问题,对本条记录的标志位的设置不会影响到上一条记录。这样在重新上电后,上一条记录会正确恢复,而本条记录是一条无效记录,而且本条记录指向的数据并没有被更新(因为只有在成功的写完标志位之后,我们认为旧的数据的备份完成了,之后才会去更新数据)。这样,事务取消的要求就得到了满足。

4.2　方案之二

上面的方案以每条记录浪费一个 Page 为代价,成功解决的写 Flash 的特性带来事务实现的难题。但是,当 Page 的单位比较大,比如是 64、128 时,采用上面的方法浪费的空间就太大了,这时我们只能采用二级缓存的方法。

方案 2:

二级缓存,将记录的标志位备份到二级缓存区。

所谓的二级缓存,指的是在设置一条记录标志位的时候,先将标志位所在的整个 Page 备份到二级缓存区,等二级记录成功写入后,然后再去设置本条记录的标志位。这样当发生断电现象 2 时,可以从二级缓存中恢复被破坏的上一条记录。

二级缓存区的要求其起始地址从 Page 开始,占用 2 个 Page 的长度。

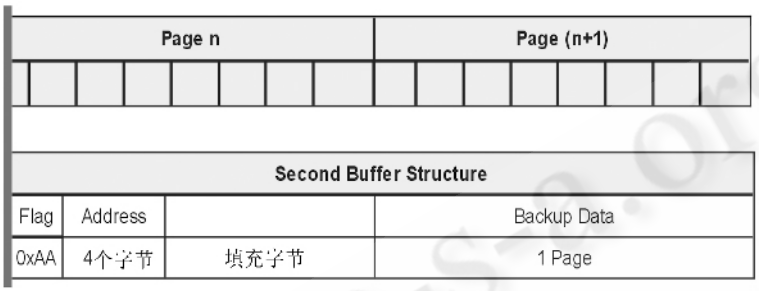

图 2　二级缓存区的结构

这样所有的记录也只浪费了 2 个 Page 的空间。但是,这样就又产生了新的问题。二级缓冲区写入的频率太高了,写每条记录时都需要写一次二级记录,而 Flash 的写寿命比 E2P 还低,这样会降低 Flash 的使用寿命。

为了避免二级缓存区频繁写入的问题,我们可以动态的改变二级缓存区的地址,以避免频繁对同一个位置进行写操作。为了确保二级缓存区产生的随机性,我们建立一张二级缓存区管理表来管理二级缓存区的产生。管理表由 64 项组成,每项存放每次动态生成的二级缓存区的起始位置。这样以后每次动态生成一个二级缓存区,首先与二级缓冲区管理表相比较。如果管理表中已经存在这个地址,表示这次新生成的二级缓存区在最近的 64 次中已经被使用过,那么需要重新生成一个二级缓存区。通过这种方式,我们把每次将二级记录写入同一位置的概率降低到原来的 1/64。

如表 3 所示,地址列表记录了最近 64 次使用的地址空间。为了确保在 64 这个周期中,不会将同一块地址空间两次作为二级缓存区来使用,那么我们需要采取类似循环文件(见 GSM11.11)的概念来管理这个地址列表。每次对管理表的项目的更新都是对最老的项目的更新。这样就可以确保我们在 64 周期内不会两次使用同一地址空间作为二级缓冲区。

表 3　二级缓存区管理表

	说明
Address 0	4 个字节的物理地址
Address 1	
Address 2	
…	
…	
Address 63	

5　结论

智能卡作为一种安全的存储介质,经常应用于银行、电信、身份认证等场合,所以事务的机制在智能卡平台上有着很重要的作用。尤其对于多应用的 Java Card 平台,其必须对数据的更新操作提供可靠的保证。在相同的条件下,相对于 E2P、Flash 能够以更低的代价提供更大的存储空间,满足了应用程序的多样化和巨大化的要求。因此,Flash 在将来有着更加广泛的应用。但是由于 Flash 的一些特殊性质,使得其在应用时要采用与众不同的方式来处理,这就带来了很多新的问题。

参考文献

1　Sun Microsystems, Inc. "Java Card 2.2 Runtime Environment (JCRE) Specification", Version 2.2, March 2002.

2　ETSI, "3GPP TS 11.11 Technical Specification", Version 8.6.0, Release 1999.

3　Jim Gray, the Transaction Concept: Virtues and Limitations, Very Large Data Bases, 7th International Conference, September 9 – 11, 1981, Cannes, France, Proceedings.

4　Jim Gray and Andreas Reuter, Transaction Processing: Concepts and Techniques, Morgan Kaufmann 1993, ISBN 1 – 55860 – 190 – 2.

5　Guthery, S. B., Internet Computing, IEEE,? Volume: 1,? Issue: 1,? Jan. – Feb. 1997, Pages: 57 – 59.