计算机科学 1999Vol. 26№ 2

移动客户机的缓存同步算法*)

Cache Synchronization Algorithm for Mobile Clients

李 霖 周兴铭

(国防科技大学计算机学院 长沙 410073)

Abstract Client caching can be used to support mobile clients' (MC) disconnected operation. MC can cache only a subset of databases on server. We present a cache synchronization algorithm, called CS algorithm, to maintain consistency between MC's cache and server's databases. It can be proved that CS algorithm will make MC's cache consistent with the server and continue to obey the cache subset description(CCD).

Keywords Mobile computing, Cache, Synchronization, Mobile database

1. 移动客户机缓存简介

在移动数据库系统中,移动客户机(Mobile Client,简称 MC)的缓存技术可以用于支持 MC 的断接操作,即在 MC 与数据库服务器中断网络连接时,允许用户继续对数据库进行操作[1]。与服务器级复制相比,MC 的缓存属于次一级的地位,它与服务器级复制互相配合,共同支持 MC 对移动数据库的访问,从而提高移动数据库的可用性。

与服务器相比,MC 的存储容量仍然是有限的,使得 MC 难以容纳整个数据库系统。另一方面,即使 MC 能够放下整个服务器的数据库,一般用户也不会希望这么做,这既有安全方面的问题,也有代价的问题:将整个数据库的缓存与服务器完成同步将是一个代价高昂的任务。因此,我们提出的 MC 缓存机制允许用户只指定 MC 缓存服务器上数据库的一个子集。

为了保证 MC 断接操作的需要,在 MC 与服务器断接之前,缓存管理器 CM 需要与服务器进行一次缓存同步,使 MC 的缓存与服务器达成一致。但是,由于 MC 的缓存只是服务器上数据库的一个子集,因此不能简单地采用直接将服务器的更新事务日志传送给 MC,并由 MC 根据事务日志来更新本地缓存的方法。为此,我们专门为维护 MC 缓存的同步设计了一种基于更新事务操作日志的缓存同步算法—CS 同步算法。下面我们将具体介绍 CS 算法的细节。

MC 缓存的核心是缓存管理器 CM。CM 在 MC 上缓存服务器数据库的一个子集,当 MC 联机时,用户事务直接交给服务器完成,这时 MC 缓存不起作用;当 MC 断接时,CM 仿真服务器的功能,利用 MC 的缓存继续为用户提供数据库访问服务,并将其中的更新事务记录到 MC 的脱机事务日志中,等到下次 MC 与服务器重新连接时将其更新结果集成到服务器的数据库中。

为了维护 MC 的缓存,CM 需要管理一些必要的元数据(meta data),其中最重要的是客户机缓存描述定义,即 CCD(Client Cache Description)。CCD 描述了 MC 用户指定的缓存数据库子集,即:

$$CCD=\{(R_i,PS_i)\mid 1\leqslant i\leqslant n\}$$
$$PS_i=\{P_{ij}\mid 1\leqslant j\leqslant Q_i\}$$

其中,R_i 为 MC 要求缓存的服务器数据库中的关系;PS_i 为定义在 R_i 上的谓词集合,谓词演算可以是选取(select)或者投影(project),其中最多只能有一个投影谓词,而且其所指定的属性集合中必须包括关系的主键;n 表示 MC 缓存中的关系数,Q_i 表示 PS_i 中的谓词个数。MC 的缓存就由服务器数据库中各个缓存关系中满足该关系的所有谓词,即各个谓

*)本课题受九五国防预研项目"并行与分布式数据库"资助。
李霖 博士研究生,主要研究兴趣为移动计算与移动数据库系统。周兴铭 中科院院士,主要研究兴趣包括高性能机体系结构、并行与分布式数据库技术等。

词的交集的全部记录组成。

MC 的缓存主要用于 MC 的断接操作，因此，为了保证 MC 脱机状态的需要，CM 只需在 MC 主动断接之前与服务器完成一次缓存同步即可，即成批更新。

2. 缓存同步算法 CS

2.1 事务日志传送法的问题

由于在 MC 缓存上一次同步之后，服务器的数据库已经被一系列事务所更新，因此一种直观的缓存更新办法是直接把服务器的更新事务日志传送给 CM，由 CM 根据事务日志来更新 MC 缓存中的数据，如 Bayou 系统[2]。但是，我们将会看到，这种方法对于允许只缓存数据库的一个子集的 MC 缓存来说是不可行的。

首先，因为 MC 的缓存只是服务器数据库的一个子集（一般情况下是一个很小的子集），如果将服务器的已提交事务日志全部传送给 CM，则日志中将包含许多与 MC 缓存无关的更新事务及其操作，浪费了大量宝贵的无线网络带宽；如果在向 CM 传送更新事务日志时，服务器根据每个更新事务 t 的操作范围和 MC 缓存的子集描述 CCD 来判断是否 t 与 MC 的缓存相关，则可以避免传送不必要的事务日志，但这又要求服务器承担较大的处理开销，如果要求同步的 MC 较多，会使服务器难以承受。

其次，即使服务器可以将所有已提交事务的日志传送给 CM，CM 可能也无法只根据接到的更新事务日志完成对 MC 缓存的更新，我们至少可以找到以下两种异常情况：

1. MC 缓存可能不包含某些相关更新事务所读的数据。例如，如果 MC 缓存只包含定单表 ORDER，则服务器上的更新操作：

```
Update ORDER
    Set QUANTITY=QUANTITY-10
    Where UID in(select UID from USER where
        NAME='John')
```

将不能在 MC 的缓存上执行，因为 MC 的缓存中不包含用户表 USER 的数据。

2. 某些更新事务即使能够在 MC 的缓存上执行，但其执行结果却可能破坏缓存子集描述 CCD 的有效性。例如，假设 MC 的缓存包括定单表 ORDER 中所有产品数量大于 50 的部分，即 $\sigma_{QUANTITY>50}$(ORDER)。服务器的更新操作：

```
Update ORDER
    Set QUANTITY=QUANTITY-10
```

将所有定单的产品数量增加 10。在 MC 的缓存上，虽然该更新操作可以执行，但执行结果却使 MC 缓存中对 ORDER 子集的定义不再有效，因为服务器上执行更新操作后，原来产品数量小于 50 但大于 40 的定单记录也将满足 MC 缓存中 ORDER 子集的定义，但是却没有传送给 MC 的缓存；换句话说，现在 MC 的缓存中 ORDER 的子集已经遗漏了一些记录。

因此，允许子集定义的 MC 缓存不能简单地采用直接传送服务器更新事务日志的方法来进行同步。为解决 MC 缓存的同步问题，我们设计了一种易于实现的缓存同步算法，称作 CS 算法（Cache Synchronization algorithm）。

2.2 CS 算法描述

CS 算法假定服务器采用基于时间戳的并发控制机制，因此每个更新事务将拥有一个时间戳，并且服务器上数据库的每个记录都附加有一个更新时间戳 ts，用于记录更新（插入或修改）该记录的最后一个已提交事务的时间戳。如果服务器采用的是基于 2PL（两阶段加锁协议）的并发控制机制，我们只需为每个提交的更新事务附加上一个时间戳，在每个关系中增加一个更新时间戳属性，并在事务提交时顺便修改被更新事务的更新时间戳，同样可以满足 CS 算法的要求。

在服务器上，每当一个更新事务提交时，服务器将该事务的更新操作序列记录到一个更新事务日志中。显然，在更新事务日志中所有事务都按照时间戳的顺序排列，服务器用 LTS 表示更新事务日志中的最新时间戳。

一个更新事务对数据库的更新包括插入（insert）、修改（update）与删除（delete）等三种基本操作。CS 算法将只需要传送已提交事务日志中的修改与删除操作部分，即忽略所有的插入操作。

为了保证服务器上更新事务的修改与删除操作能够在 MC 缓存上顺利执行，服务器在执行更新事务并记录其操作日志时，将事务中的所有集合更新操作都转换为单记录更新操作序列。因此，在服务器的事务日志中，所有的数据库修改与删除操作都是单记录操作，不再包括带有查询条件的集合更新操作。例如，在执行一个集合删除操作：

```
delete from ORDER
where QUANTITY<50
```

时，服务器在更新事务日志中记录为 $d(o,x_1)$，$d(o,x_2)$，…，$d(o,x_n)$，其中 o 表示关系 ORDER 的标识

符，x_i 为关系 ORDER 中记录的主键值，且 $x_1, x_2, \cdots, x_n$ 分别代表关系 ORDER 中所有满足条件 QUANTITY<50 的记录。类似地，对于一个集合修改操作：

```
update ORDER
    set QUANTITY=QUANTITY+10
    where QUANTITY<40
```

服务器将其记录为 $u(o, x_1, r_1), u(o, x_2, r_2), \cdots, u(o, x_n, r_n)$，其中 x_i 表示所有被更新记录的主键值，r_i 表示更新后的记录值。因为数据库服务器在执行更新事务时，最终总是会归结于对一系列单记录的更新，因此服务器将更新事务日志记录为单个记录更新操作序列的过程并不需要特殊处理，目前所有的商业 DBMS 产品都支持此功能。在后面的 CS 算法正确性讨论中，我们将看到把集合更新操作转换为单记录操作序列的必要性。

与服务器类似，MC 缓存中的每个记录也带有更新时间戳 ts，用于在 MC 重新联机并将脱机事务交给服务器执行时判断可能发生的更新事务冲突。CM 还记录 MC 缓存的更新时间戳 MC_NTS，它等于缓存中所有记录更新时间戳的最大值，即本地缓存已代表了服务器上所有时间戳小于或等于 MC_NTS 的已提交更新事务的执行结果。当 MC 的缓存初建时，服务器在向 CM 发送所有满足 CCD 描述的数据库记录的同时，附带当时更新事务日志的最新时间戳 LTS，CM 在建立缓存之后，将 MC_NTS 的初值设置为 LTS。

如前所述，在 MC 与服务器断接之前，缓存管理器 CM 将发起与服务器的一次缓存同步过程，即开始执行 CS 同步算法。CS 缓存同步算法由服务器和 CM 相互配合完成，其具体步骤如下：

1）CM 将 MC 缓存的子集描述 CCD 以及 MC 缓存的更新时间戳 MC_NTS 发送给服务器。

2）对于更新事务日志中所有时间戳大于 MC_NTS 的更新事务，服务器将其中所有与 CCD 相关的修改与删除操作按时间戳 ts 的次序传送给 CM；其中，对某个修改操作 u(o,x,r)（或删除操作 d(o,x)），如果关系 o 在 CCD 中定义，则说明该修改操作 u（或删除操作 d）与 CCD 相关。

3）CM 接到这些更新事务的修改/删除操作后，在本地缓存上依次执行每个操作：

● 对于删除操作 d(o,x)：若 x 在本地缓存中，则从缓存中删除 x；否则，忽略操作 d。

● 对于修改操作 u(o,x,r)：若 x 在本地缓存中，则将记录 x 更新为 r 的值，如果更新后 x 不再满足 CCD 定义，则从缓存中删除 x；否则，忽略操作 u。

CM 完成所有操作之后，向服务器发送执行完毕的确认消息。

4）接到 CM 的确认消息之后，对缓存描述 CCD 中的每一个关系 R_i，执行下述过程：

a）服务器根据 CCD 中描述的 R_i 的谓词集合 PS_i，再加上一个选择谓词 $\sigma_{ts>MC\_NTS}$，构造一个更新记录查询 Q_i，用以确定并找出 MC 缓存中 R_i 的所有需要更新的记录。设谓词集合 PS_i 中投影谓词指定的属性集合为 A（如果 PS_i 中没有投影谓词，则 A 包含关系 R_i 的所有属性），选择谓词 $s_1, \cdots, s_k$，则更新记录查询 Q_i 可以表示为：

```
select A
    from Ri
    where s1∧s2∧ ∧sk∧(ts>MC_NTS)
```

然后，服务器执行 Q_i，并将查询结果发送给 CM。

b）CM 接到这些记录后，根据记录的主键，将每个记录插入缓存或者更新缓存中已有的记录。

5）待 CCD 中的每一个关系 R_i 处理完之后，服务器将已提交事务日志中的最新时间戳 LTS 发送给 CM。

6）接到服务器发送的 LTS 后，CM 将本地的缓存更新时间戳 MC_NTS 更新为 LTS 的值，并结束本次缓存同步过程。算法结束！

2.3 CS 算法的正确性

下面我们将证明 CS 缓存更新算法的正确性，即：CM 与服务器执行一次 CS 缓存同步算法之后，MC 的缓存确实与服务器上数据库的最新版本达成一致，并且继续保持缓存子集描述 CCD 的有效性。所谓 CCD 的有效性，是指在缓存同步之后，MC 的缓存记录集合仍然等于按照 CCD 描述的服务器数据库子集。

定理 1 缓存管理器 CM 与服务器执行一次 CS 缓存同步算法之后，MC 的缓存与服务器数据库的最新版本达成一致，并且继续保持缓存子集描述 CCD 的有效性。

证明：（提要）设服务器数据库中满足 CCD 定义的子集为 S_{CCD}。定理 1 实际上是要证明，在服务器与 CM 执行一次 CS 同步算法之后，以下三个断言成立：

（1）对 MC 缓存中的任意记录 r，若 r 也在 S_{CCD} 中（记为 r'），则 r=r'；

(2)对任意记录 r,若 r 在 MC 缓存中,则 r 也必在 S_{CCR}中;

(3)对任意记录 r,若 r 在 S_{CCR}中,则 r 必在 MC 的缓存中。

我们采用了归纳法证明。定理 1 得证。

如果我们不要求服务器将已提交事务的删除操作记录为单记录删除的话,CS 缓存同步算法的正确性将不能保证。例如,MC 的缓存包括关系 ORDER。如果服务器上时间戳大于 MC_NTS 的已提交事务日志中包含一个更新操作:

```
update ORDER
    set QUANTITY=QUANTITY+5
```

随后又有一个集合删除操作:

```
delete from ORDER
    where QUANTITY≥50
```

则在完成 CS 缓存同步算法之后,MC 的缓存将出现漏删现象:由于 CS 算法先执行删除操作,因此在同步之前 MC 的缓存中 QUANTITY 在 45～50 之间的 ORDER 记录将不会被删除,而这些记录在服务器上都将被删除。因此,在记录已提交事务日志时,服务器必须将所有的集合删除操作转化为单记录删除操作序列。

小结 针对允许只缓存服务器数据库部分子集的移动客户机缓存机制,本文介绍了一种缓存同步算法—CS 算法。CS 算法以更新事务操作日志的传送为基础,保证了 MC 的缓存在经过同步之后仍然保持与服务器上数据库之间的一致性,并且继续维护缓存子集描述 CCD 的有效性。同时,CS 算法只要求服务器向 CM 发送自上次缓存同步以来服务器更新事务的删除操作日志以及被更新的数据库记录,因而尽可能地降低了服务器与 MC 的处理开销,具有很高的执行效率。因此,CS 算法是一个正确、高效的缓存同步算法。

参 考 文 献

1 李霖,周兴铭 移动数据库中客户机的断接操作.计算机科学,1999,26(3)

2 Demers A, et al. The Bayou architecture: support for data sharing among mobile users. In Proc. IEEE Workshop on Mobile Computing Systems and Applications, Santa Cruz, California, 1994. 2～7

(上接第 4 页)

个 agent,给定一个名字,然后在一个标准的模式下挂起、恢复或终止这个 agent 的执行。

agent 转移:agent 应用中创建出的众多 agent 在不同类型的 agent 系统中自由移动的能力是标准化的主要目标之一,它规定了一个公共的基础构架。

agent 和 agent 系统的命名:agent 系统的互操作除了要求 agent 操作的标准化,其参数的语法和语义也必须标准化,特别是 agent 和 agent 系统命名也要标准化,它使得应用能够标识 agent 和 agent 系统,亦使得 agent 和 agent 系统能互相标识。

agent 系统类型和定位:只有能被某一目标 agent系统所支持,agent 才能向该 agent 系统迁移,因此,agent 迁移前必须要能从异地获取目标系统的类型信息,故 agent 系统的定位语法必须标准化。从 agent 系统彼此定位的角度来讲,定位语法也需标准化。

小结 本文的中心议题是 agent 的流动特性,我们认为流动 agent 是一独立的计算机程序,它可自主地在异构的网络上,按照一定的规程流动,寻求合适的计算资源、信息资源或软件资源,利用与这些资源同处一台主机或网络的优势,处理或使用这些资源,代表用户完成特定的任务。此外,我们列出了使用流动 agent 所带来的若干好处,给出了 agent 技术的几个前景较好的应用领域,并介绍了若干目前较为成熟的基于 Java 的流动 agent 系统。尽管基于 Java 的流动 agent 系统有许多共性,但不能实现互操作,为此本文最后介绍了 OMG 提出的 MASIF:流动 agent 技术的某些成份的标准化。agent 技术的标准化对该技术的影响将会是令人鼓舞的。(参考文献,略)