TBASE 事务管理子系统的设计与实现

学　　院　计算机科学与技术

专　　业　计算机科学与技术

年　　级　2007 级

姓　　名　杨晓科

指导教师　张坤龙

2011 年 6 月 10 日

摘　要

TBASE 是一个用于基础教学的小型关系型数据库管理系统，它是一个完整的数据库管理系统，实现了数据库存储管理、查询优化和事务管理三大子系统。本论文设计并实现了 TBASE 的事务管理子系统。

事务管理子系统包含三大模块：锁管理、日志管理和恢复管理。锁管理模块实现了 SLock 和 XLock 两种形式的锁。采用两阶段加锁策略，通过一个锁表，处理所有并发事务，以便维护系统一致性；日志管理模块使用 WAL（写优先日志协议），靠日志记录即时记录所有对数据库系统的更新。恢复管理模块通过使用开始日志记录、提交日志记录、检查点日志记录等五种类型的日志记录来实现在系统崩溃后将系统进行恢复，以便系统可以正确的反映提交的事务和未提交的事务的状态。

论文对最终的事务管理子系统进行了黑盒测试。测试结果表明系统可以正常的处理并发运行的事务，并且可以在系统崩溃后恢复到正确的状态。

关键字：事务；封锁；日志；恢复；并发控制

ABSTRACT

TBASE is a small database management system for basic education. It is a complete database management system, implementing three major subsystems, which are data storage management, query optimization and transaction management. This paper designs and implements the transaction management subsystem of TBASE.

Transaction management subsystem consists of three modules: the lock management, log management and recovery management. Lock management module implements two forms of lock, that is, SLock and XLock. It takes a two-phase locking policy, and uses a lock table to handle all concurrent transactions, so it can maintenance the consistency of the system; log management module takes the WAL (write log priority protocol), writing a log record whenever the database system has any updates. Recovery Manager module uses five different types of log records, such as the start logging records, submitted log records and the checkpoint log records and so on, to recovery the system after it crashes, so that the system can reflect the state of the submitted transaction and the uncommitted transaction.

This paper takes a black box testing for the final transaction management subsystem. Test result shows that the system can handle concurrent transactions that are running normally, and can recovery to the correct state after a system crashes.

Keywords: transaction; lock; log; recovery; concurrency control

目　录

第一章 绪论

1.1 数据库管理系统

数据库（Database）是指长期保存在计算机的存储设备上、并按照某种模型组织起来的、可以被各种用户或应用共享的数据的集合。这种数据集合中的数据都是结构化的，而且尽可能的不出现重复，并且通常以最优方式为们某个特定组织的多用应用服务；数据的存储独立于使用它的应用程序，对于数据的增、删、改和检索由统一的系统进行管理和控制，而这个系统就是数据库管理系统。

数据库管理系统[6][17]（database management system）是操纵和管理数据库的大型软件，用于建立、使用和维护数据库，简称 DBMS。它对数据库进行统一的管理和控制，以保证数据库的安全性和完整性。用户通过 DBMS 访问数据库中的数据， 数据库管理员，也就是常说的 DBA，也通过 DBMS 对数据库进行维护。

现在，随着计算机软硬件技术的飞速发展，以及计算机系统在各个行业的广泛应用，数据库管理系统在实践中得到了广泛的应用。

目前国内外比较常见的数据库管理系统有 My SQL，Sybase，DB2，Oracle 等等，随着世界信息化程度的不断加深，数据库管理系统早已成为一个企业不可或缺的工具。

一个完整的数据库管理系统应该包括以下三个模块：存储模块，事务模块和查询优化模块，这三个模块相互联系、相互作用，组成一个不可分割的整体。

存储管理可以分为：文件和存取方法，缓冲区管理，磁盘空间管理。文件爱你和存取方法主要负责管理文件的存储结构；缓冲区管理部分的责任是把页从磁盘取入主存以响应读请求；磁盘空间管理部分则主要是分配、回收、读和写页面。

事务模块由锁管理、日志管理和恢复管理组成。锁管理部分跟踪对锁的请求，当数据库对象可用时，在该对象上授权加锁；日志管理部分记录数据库所有的变化；恢复管理部分负责维护日志，在系统崩溃后把系统恢复到一致性的状态。

查询优化模块是由分析器、优化器、计划执行器和操作求解器四个部分组成。这四个部分主要是针对用户提出的请求，经过语法分析，查询优化等最终生成一个高效的执行计划。

一个数据库管理系统必须拥有自己的数据模型，即隐藏了许多低级存储细节的高级数据描述结构的集合，例如实体-联系（ER）模型，使用它用户就可以定义要存储的数据。

数据库管理系统中的数据通常可以描述成三级抽象，分别是：概念模式、物

理模式和外模式，如图 1-1 所示。

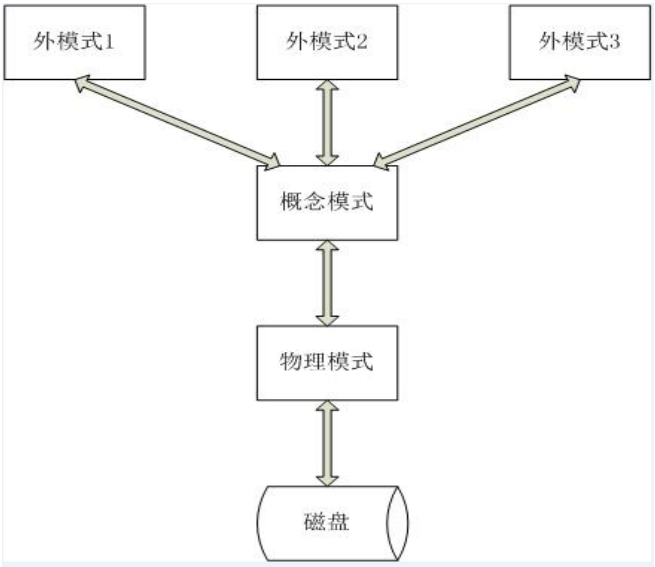

图 1-1 DBMS 中抽象的级别

物理模式描述存储细节。本质上看，物理模式描述了概念模式中关系在磁盘和磁带等二级存储上实际是如何存储的。

概念模式是以 DBMS 数据模型的形式描述存储的数据。它描述存储在数据库中的所有关系，这些关系包含实体信息和联系信息。

外模式，也称为 DBMS 的数据模型，允许为单个用户和用户组定制数据存取，一个数据库可以有多个外模式，每一个外模式由一个或者多个视图和来自概念模式的关系组成。

使用数据库管理系统有着很多的有点：

第一：数据独立性；DBMS 提供数据的抽象视图，把应用程序和数据细节分开。

第二：有效的数据存取；DBMS 使用各种复杂的技术来有效的存储和检索数据。

第三：数据完整性和安全性；总是通过 DBMS 来存取数据，可以增强数据的完整性约束。同时，通过实行存取控制，可以确保数据对不同用户的可见性。

第四：数据管理；多个用户共享数据，可以使得数据得到几种管理。

第五：并发存取和故障恢复；并发调度数据，所有用户在存取数据时感觉就像只有一个用户在操作一样。而且用户可以免受系统故障的影响。

第六：减少应用程序开发时间；DBMS 支持很多重要功能，而且这些功能对于很多存取 DBMS 数据的应用程序来说是通用的。

1.2 TBASE 概述

TBASE 是一个简单的多用户小型关系数据库管理系统（RDBMS），主要目的是为了教学。T 即取 TJU 或者 Teaching 之首字母，BASE 即基础之义。

TBASE 拥有一个标准的 RDBMS 的全部特征，包含并发控制[2]，锁，日志和恢复，缓冲区，堆文件，空间管理，访问方法[3]（B-树），关系操作符，查询优化和用户接口组件。

TBASE 数据库管理系统总体上分为三大模块：存储管理，查询优化和事务管理[1]，其模块划分可以表示成图 1-2 所示。

与其他数据库管理系统相比有如下特点：

第一，提供了 B 数索引；

第二，支持事务处理；

第三，使用写日志优先[5]；

第四，支持并发控制。

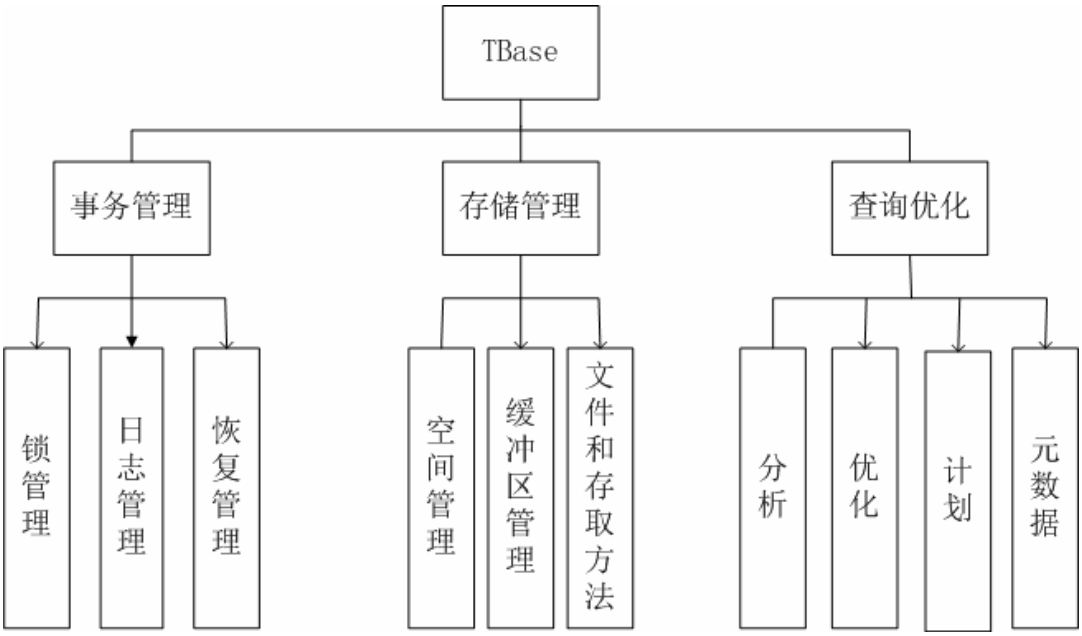

图 1-2 TBASE 数据库管理系统模块划分

1.3 事务管理子系统

事务管理子系统，是 TBASE 数据库管理系统的一个重要组成部分。事务管理子系统要实现的功能包括日志管理，锁管理[9]和恢复管理[7]。事务是访问和更新数据库中各种数据的一个基本执行单元，是一个整体。事务也是恢复和并发控

制的基本单元。它具有四个基本的属性，即，原子性、一致性[10]、隔离型、持久性。也就是通常所说的 ACID 特性。

原子性（A）：所有的操作要么全部都被执行，要么都不被执行，用户不需要担心不完全的事务（如当系统崩溃发生时）造成的后果。

一致性（C）：个单独执行的事务，即在它不和其他事务并发执行的情况下，必须能够保证数据库的一致性。

隔离性（I）：个事务的执行不能被其他事务干扰。即数据库管理系统将若干个事务交叉执行，用户不需要考虑并发执行的其他事务的影响，就能理解一个事务。

持久性（D）：一旦数据库管理系统通知用户事务已经成功执行，那么即时在数据更新存入磁盘之前发生了系统故障，事务处理的结果也能够被永久地保持住。

事务管理子系统就是要确保事务的 ACID 四个特性，以便实现并发控制，以及在系统发生故障时能够进行恢复。其主要目的就是提高事务处理的效率，保证其正确性。在事务管理子系统的三大模块中，锁模块的作用就是满足事务的原子性和隔离性；日志被用于确保事务的持久性；一致性则由恢复管理来完成。

1.4 论文组织结构

第一章是绪论部分，主要介绍数据库管理系统的原理、体系结构、目前发展状态，以及 TBASE 数据库管理系统的体系结构和 TBASE 事务管理子系统的概况。

第二章是 TBASE 事务管理子系统的简介，主要是功能需求、软硬件开发环境以及事务管理子系统的架构。

第三章是 TBASE 事务管理子系统的设计与实现，主要讲解事务管理子系统三大模块各部分的设计原理与具体实现方法。

第四章是 TBASE 事务管理子系统的测试，主要是使用黑盒测试方法来实测系统的正确性。

第五部分是总结与展望，主要是对本论文的一些看法、未来的研究方向以及可能带来的影响。

最后是参考文献，外文资料及相应的中文参考和致谢。

第二章 TBASE 事务管理子系统体系结构

2.1 功能需求

数据库管理系统的一个重要任务就是进行数据的并发存取调度，以保证所有用户感觉不到同时还有其他用户在对数据进行存取。这是很重要的，因为通常一个数据库会被许多用户所共享，这些用户都是独立地向数据库管理系统提出他们的请求，而且不能期望数据库能处理那些由其他用户并发做出的改变。数据库管理系统让用户赶紧他们的程序正在由数据库管理系统按照某种顺序一个接一个地独立地执行。例如，一个往账户中存款的程序提交给数据库管理系统，同时又有另一个程序从该账户中取款，这两个程序中的任一个都可以被数据库管理系统首先运行，但它们的所有步骤都不能以相互作用的形式来交叉执行。

此外，数据库管理系统必须保护用户免于系统故障的影响，即，当系统在崩溃后重新启动时，要确保所有数据（和当前应用程序的状态）能恢复到一致的状态。例如，如果一个旅行代理商提出一个预订请求，且数据库管理系统应答说预订已经完成，此时如果系统崩溃，应确保预订信息不会丢失。另一方面，如果数据库管理系统还没有对请求应答，即在修改时发生了崩溃，则当系统重新启动时，未完成的修改则应该被取消。

事务管理子系统就是为解决诸如此类的问题而设计的，它所要实现的基本功能就是在若干个事务交叉执行的时候能够确保系统并发执行的结果和串行执行的结果相同，保证系统运行的正确性和高效性；同时还要求在系统出现故障时能够通过有效出措施确保数据的一致性。

事务管理子系统需要实现的功能包括锁管理、日志管理和恢复管理，每一个模块的具体需求如下：

锁管理模块要能够根据已经写好的加锁策略[13]，在适当的操作上适当的时间加合适的锁，加锁的粒度根据需求的不同而变化，确保并发执行的事务各个操作不会发生冲突。

日志管理模块要及时记录事务对数据库进行的各项修改，并且要保存在稳定的存储上，以便在系统发生异常或者崩溃的时候，能够幸免于难，为数据的恢复提供依据，确保事物的持久性。

恢复管理模块要能够在系统崩溃后重启时，读取日志文件中的信息，经过分析和取消崩溃时尚未提交的事务来确保事务的一致性。

事务管理模块要能够和 TBASE 其他两个模块（即，存储管理和查询优化）

很好的融合，确保整个系统正确、高效、稳定的运行。

2.2 开发环境

事务管理子系统在整个开发过程中环境如表 2-1 所示。

表 2-1 开发环境

名称	具体描述
开发语言	C#
架构模型	C/S
开发平台	Microsoft Windows 7 ultimate .Net Framework 3.5
IDE 工具	Visual Studio 2008

系统主要采用的编程语言是 C#，原因是 C#简单易用，编程方便，同时与 COM 集成，简化了很多编程上的工作，并且系统开发者本人在各种编程语言中相对来说更习惯于使用次语言。

2.3 体系结构

基于前面所描述的功能需求，对于事务管理子系统来说要完成三大模块的功能：锁管理、日志管理和恢复管理，其体系结构详细表示如图 2-1 所示。

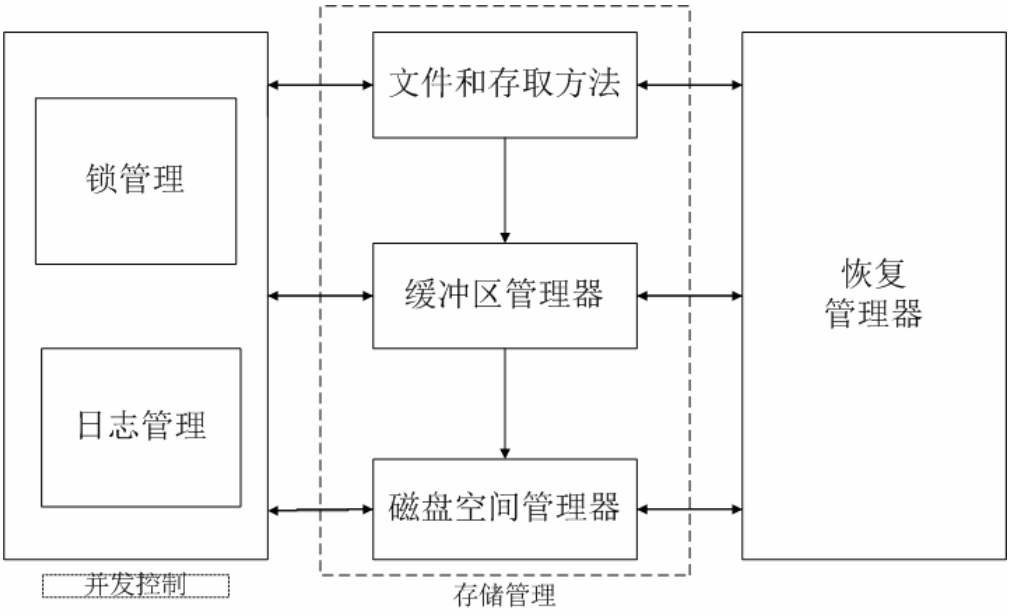

图 2-1 TBASE 事务管理子系统体系结构

图 2-1 中，虚线框中部分表示存储管理子系统，两边为事务管理子系统。存储管理子系统总体上可以分为三个部分：文件和存取方法，缓冲区管理器和空间管理器。事务管理子系统分为并发控制和恢复管理，其中并发控制一般分为锁管理和日志管理。由图可知，并发控制，即所管理和日志管理和存储管理三个部分都有交互，恢复管理也是如此。本论文将此实现为锁管理，日志管理和恢复管理三部分，图中不仅显示了三大模块，而且指示了它们主要交互的对象。

第三章　TBASE 事务管理子系统的设计与实现

3.1 锁管理模块

3.1.1 锁管理的原理

数据库管理系统中处理事务加锁的部分即称为锁管理模块[12]。锁管理模块维护着一个锁表，这是一个以数据对象标识为码的哈希表。数据库管理系统也在事务中维护每个事务的描述信息，该记录中包含一个指向事务拥有的锁列表的指针。在请求锁之前要检查这个列表，以确定不会对同一个锁请求两次。

对数据进行并发存取调度需要使用一个加锁协议，也就是每一个事务需要遵守的规则集合，以确保及时有几个事务在交叉工作，但它们锁产生的结果等同于所有事务按照某种欵全能型顺序执行的结果。锁是一种用来控制对数据库对象进行存取的机制。数据库管理系统通常支持两种类型的锁：共享锁和互斥锁。一个对象的共享锁可以由两个不同事务同时获得，而对象上的互斥锁则确保没有其他事务能获得该对象上的任何锁。

加锁表中的每一项针对某个数据对象，它包括下面的信息：拥有某个数据对象的事务数目（共享锁可能会超过一个），锁的自然属性（兼容或者互斥），和一个指向加锁请求队列的指针。

事务 T 在读写数据库对象 O 之前，必须先获得 O 的共享锁或者互斥锁，并且将它保持到最终提交或者中止。当事务需要某个对象的锁时，需要将请求提交给锁管理器。

如果请求的是共享锁，当前请求队列为空，而且该对象没有处于互斥锁状态时，锁管理器将同意加锁请求，并更新对象的相应锁表数据项。

如果请求的是互斥锁，并且没有事务拥有该对象的锁，那么锁管理器可以同意加锁请求，并更新该对象的相应锁表数据项。

对于其他情况，加锁请求不能立刻得到满足，加锁请求将被添加到该对象的加锁请求队列中，同时挂起请求加锁的事务，等待需要的锁直到能够得到满足，如果时间过长则可以选择结束重启该事务。

当事务被中止或者提交时，会释放所有自己拥有的锁。

加锁和解锁命令被实现成原子操作，使用的方法是当锁管理器并发执行时，需要使用系统提供的线程锁机制来实现。

3.1.2 锁管理模块的设计

1 锁模式

论文中使用两种锁模式来实现加锁，分别是共享锁(SLock)和互斥锁(XLock)，它们的兼容性如图 3-1 所示。

	SLock	XLock
SLock	✓	
XLock		

图 3-1 TBASE 使用的锁相容矩阵

图 3-1 中，一共使用了两种类型的锁：共享锁（SLock）和互斥锁（XLock）。其中 SLock 和 SLock 是兼容的，XLock 与 SLock 和 XLock 都不兼容，即多个事务可以同时获得一个对象的共享锁，但是只能有一个事务获得某个对象的互斥锁。

2 内部设计架构

本论文中，在锁管理模块中维护了一个锁表，该表跟踪每一个事务已经锁住的页和正在等待要锁的页。锁表是一个哈希表，表中的每一个锁实体代表一个被锁住的页。其结果如图 3-2 所示。

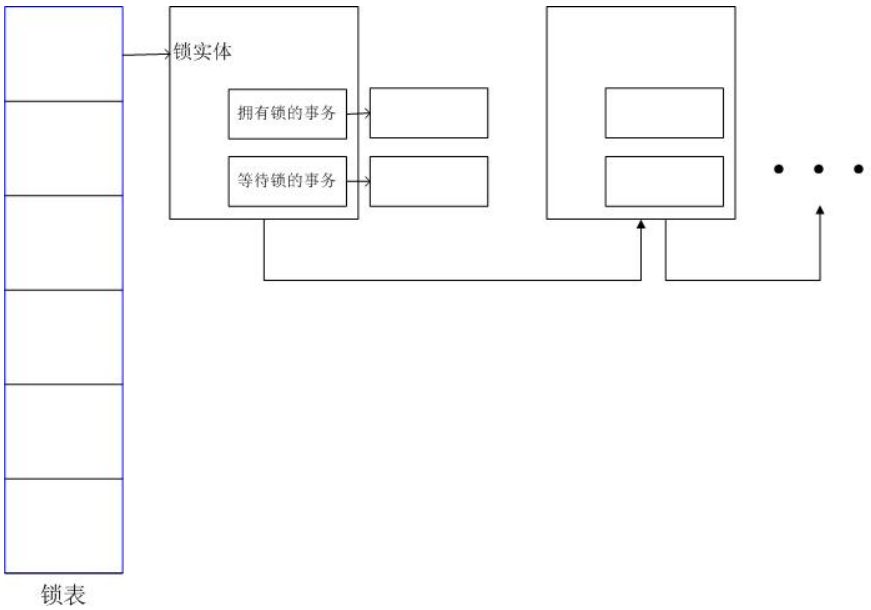

图 3-2 锁管理器中锁表的结构

3 程序执行流程图：

一个事务在读或者写数据库中的数据前必须申请该对象上的锁，事务请求加

锁的整个过程如图 3-4 所示。

图 3-4 中，一个事务在请求一个锁时，首先查看该对象上是否已经有锁，若没有则直接请求自己需要的锁，如果申请的是共享锁，则可以直接申请，否则，先申请共享锁，然后将共享锁升级为排它锁；如果该对象上已经有锁了，则检查申请锁的类型与已经存在的锁是否兼容，如果兼容则直接申请，否则将该事物置入等待队列中，之后检查等待是否超时，如果没有超时，则重新检查申请的锁是否与已经存在的锁兼容，否则直接结束该事务。

3.1.3 锁管理模块的实现

在锁管理（即并发控制）这一块，论文实现了四个类，包括 LockTable，ConcurrencyMgr，LockAbortException 和 Transaction 四个类。锁管理器子模块的类图如图 3-3 所示。

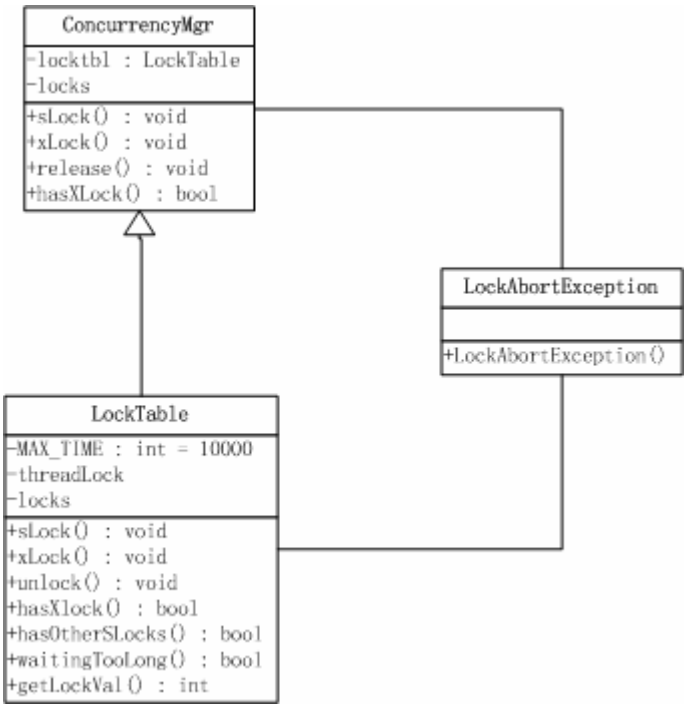

图 3-3 锁管理器子模块的类图

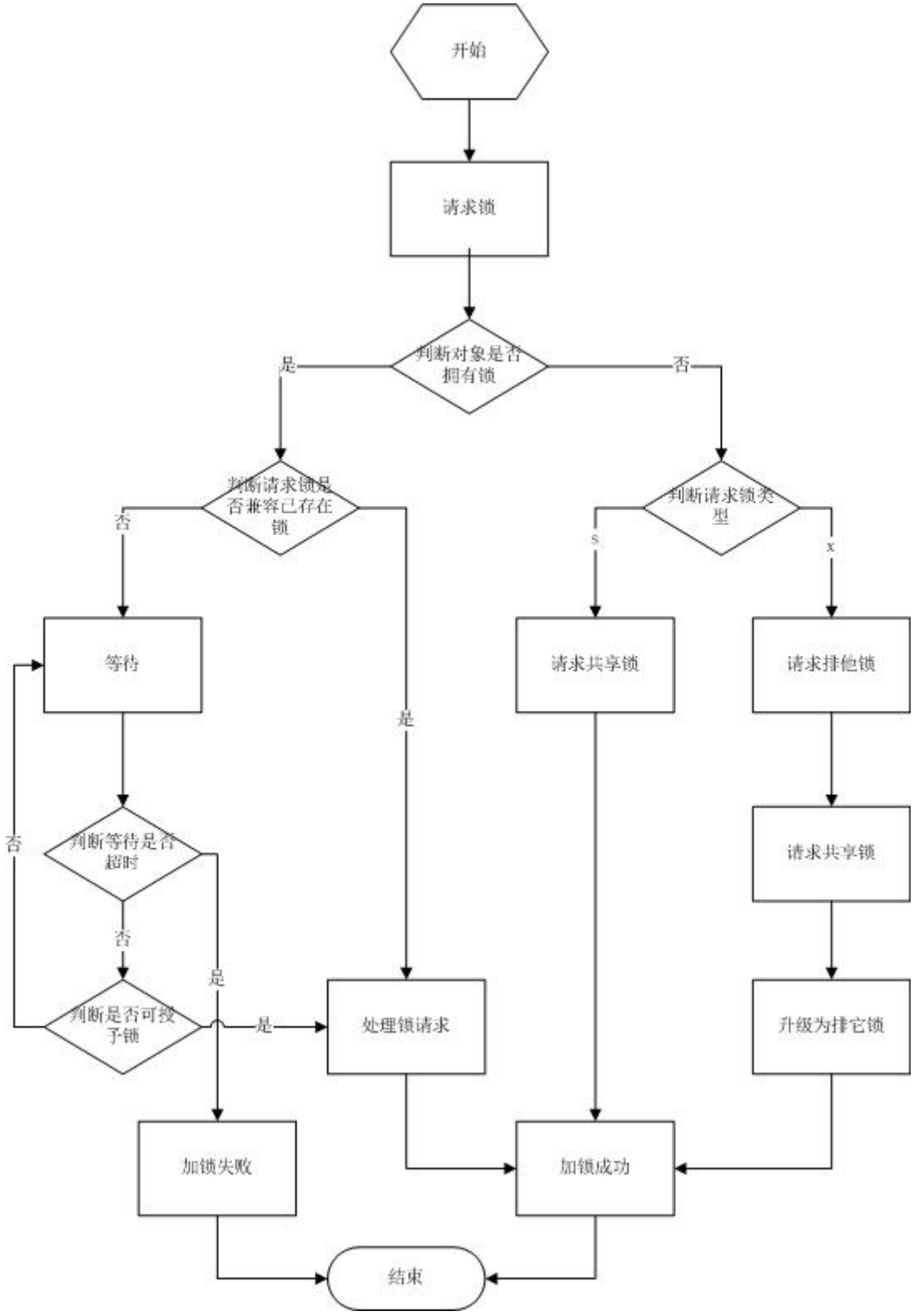

图 3-4 事务请求锁及加锁的流程

下面分别就个各类的成员变量及方法进行讲解

1 LockTable

(1) 概述

LockTable 是锁表，提供了 block 的加锁和解锁的方法。如果一个事务请求一个引起与现存锁冲突的锁，那么该事务会被放到等待链表。所有的 block 只有一个等待链表。当一个 block 上的最后一个锁解锁时，所有的事务被从等待链表中移除并重新调度，如果其中的一个事务发现它所等待的锁仍然处于加锁状态，那么它把自己放回等待链表。

对 Block 加锁的方法是：在 Block 和一个 integer 之间建立一个映射，这个 integer 标记了该 Block 是 SLock 还是 XLock。SLock 的 integer 值是非负的，XLock 统一标记为-1。

而且，对 SLock 的编号的方式是：初始化的 sLock 是 0，在添加到 LockTable 中的时候，所有的 SLock 的编号成邻接有序排列。在想 LockTable 中添加一个 SLock 的时候，会修改其 integer 值，Value=表中前一个 SLock 的 Value+1。锁表类的字符和方法如图 3-5 所示。

图 3-5 锁表类

(2) 成员变量

MAX_TIME：此变量是事务等待的时间，一旦时间超过 MAX_TIME，将会超出一个异常，它的值设置为 10000 毫秒。

Locks：此变量为锁表，记录每一个块上拥有的锁。

(3) 成员方法

- **public void** sLock(Block blk);

描述：在指定的 block 上获得一个共享锁，如果该方法被调用时有一个互斥

锁存在，那

么调用线程将被放到等待链表直到锁被释放，如果在一定时间段（当前为 10 秒）之后该线程仍然在等待链表中，那么将抛出一个异常。

实现：1.调用 System 的 currentTimeMillis()获得以毫秒为单位的当前时间并赋值给 long 类型变量 timestamp。

2.使用一个 while 循环，循环条件为两个成员方法的调用，一是 hasXlock(blk)即在指定的 block 上有互斥锁，二是！waitingTooLong(timestamp)即以 timestamp 为开始时间计算没有超过等待时间的限制，两个条件同时满足则进入循环 wait(MAX_TIME)线程等待 10 一个 MAX_TIME 的时间段。

3.跳出 while 循环之后调用 hasXlock(blk)检测指定的 block 上是否仍然有互斥锁，如果是的话则说明在线程等待一定时间段之后仍处在等待链表中，抛出异常。

4.此时共享锁可以获得，调用成员方法 getLockVal(blk)获得该 block 对应的整数值并赋值给 val。

5.调用 Map 对象的 put 方法用 val+1 替换 val。

- **public void** xLock(Block blk)；

描述：在指定的 block 上获得一个互斥锁，如果该方法被调用时有任意类型的一个锁存

在，那么调用线程将被放到等待链表直到所有的锁被释放，如果在一定时间段（当前为 10 秒）之后该线程仍然在等待链表中，那么将抛出一个异常。

实现：1.调用 System 的 currentTimeMillis()获得以毫秒为单位的当前时间并赋值给 long 类型变量 timestamp。

2.使用一个 while 循环，循环条件为两个成员方法的调用，一是 hasOtherSLocks(blk)即在指定的 block 上有其他共享锁，二是！waitingTooLong(timestamp)即以 timestamp 为开始时间计算没有超过等待时间的限制，两个条件同时满足则进入循环 wait(MAX_TIME)线程等待 10 一个 MAX_TIME 的时间段。

3. 跳出 while 循环之后调用 hasOtherSLocks(blk)检测指定的 block 上是否仍然有其他共享锁，如果是的话则说明在线程等待一定时间段之后仍处在等待链表中，抛出异常。

4.此时互斥锁可以获得，调用 Map 对象的 put 方法将 blk，-1 放到 locks 中。

- **public void** unlock(Block blk)；

描述：释放指定 block 上的一个锁，如果这个锁是该 block 上的最后一个锁，那么等待

着的事务将被通知。

实现：1.调用成员方法 getLockVal(blk)获得该 block 对应的整数值并赋值给 val。

2.如果 val 大于 1，说明该 block 上有不止一个共享锁，此时释放一个锁，因此

用 val-1 来替换所表中的 val；否则从锁表中删除该 block 对应的项并通知所有等待线程。

- **private boolean** hasXlock(Block blk);

描述：判断指定 block 上是否有互斥锁。

实现：调用成员方法 getLockVal()得到该 block 在锁表中对应的整数值，如果该值为负即

小于 0，那么该 block 上有互斥锁。

- **private boolean** hasOtherSLocks(Block blk);

描述：判断指定 block 上是否有其他的共享锁。

实现：调用成员方法 getLockVal()得到该 block 在锁表中对应的整数值，如果该值大于 1，

说明该 block 上有多与 1 个共享锁，即该 block 上有其他共享锁。

- **private boolean** waitingTooLong(**long** starttime);

描述：判断一个线程是否等待太长时间。

实现：调用 System 的 currentTimeMillis()获取以秒为单位的当前时间，减去参数开始时

间 starttime，如果差值大于系统设定的最大时间 MAX_TIME，则返回 ture 即超时。

- **private int** getLockVal(Block blk);

描述：获得 block 在所表中对应的整数值。

实现：1.调用 Map 的 get 方法获得锁表中 blk 对应的整数值赋值给 Integer 对象 ival。

2.如果 ival 为空，即该 block 没有加锁，返回 0，否则返回该整数值。

2 ConcurrencyMgr

(1) 概述

ConcurrencyMgr 是事务的并发管理器，每一个事务有自己的并发管理器，并发管理器跟踪事务当前拥有哪些锁，并且根据需要与全局的锁表进行交互。它包含了关于 Block 和 String 的映射，对加油不同锁的 Block ，用“S”和“X”来标记。ConcurrencyMgr 在加 xLock 的时候，不是直接加上 xLock，而是先加上 sLock，然后再不 sLock 升级为 xLock。并发管理器定义的字段及方法如图 3-6 所示。

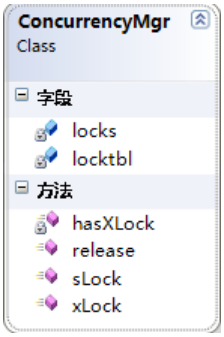

图 3-6 并发控制类

(2) 成员变量

Locktbl：此变量为一个全局的锁表，所有的事务都共享这同一个锁表。

Locks：

(3) 成员方法

- **public void** sLock(Block blk)；

描述:在 block blk 上获得一个共享锁（如果有必要），如果该事务当前在该 block 上没有

锁，那么该方法将会向锁表请求一个共享锁。

实现：1.调用 Map 的 get()方法获得成员参数 locks 中 blk 对应的值并判断是否为空。

2.值为空说明该事务当前在该block上没有锁，调用LockTable的sLock()方法请求一个共享锁，并在 locks 中添加该锁 blk，“S”。

- **public void** xLock(Block blk);

描述：在 block blk 上获得一个互斥锁，如果该事务当前在该 block 上没有互斥锁，那

么该方法将首先在该 block 上获得一个共享锁（如果有必要），然后升级为互斥锁。

实现：1.调用成员方法 hasXLock()判断该 block 上是否有互斥锁。

2.如果没有的话，调用 sLock()获得该 block 上的共享锁，调用 LockTable 的 xLock()方法获得该 block 上的互斥锁，并将 blk，“x”添加到 locks 中。

- **public void** release();

描述：通过请求锁表解锁每一个锁来释放所有的锁。

实现：1.使用一个 for 循环，对于 locks 的键值集合中的每一个 block，调用 LockTable 的 unlock()方法释放其上的锁。

2.清空 locks。

- **private boolean** hasXLock(Block blk);

描述：判断该事务当前在 block 上是否有互斥锁。

实现：1.调用 Map 的 get 方法获得 locks 中 blk 对应的值并赋给字符串变量 locktype。

2.如果 locktype 不等于空，即该事务在 block 上有锁，并且 locktype 的值等于“x”则说明该事务当前在 blk 上有互斥锁。

3 Transaction

(1) 概述

确保所有的事务是串行的，可恢复的，并且满足ACID原则，根据其成员变量可以画出如图3-7所示的结构图：

图 3-7 Transaction 成员变量的结构

在这个结构中，缓冲区列表是 Transaction 的核心，恢复管理器和并发管理器的操作对象都是缓冲区列表。事务号是为了标记每一个事务，而 nextTxNum 把事务串起来。

(2) 成员方法

- Commit()和 rollback()

这两个方法的结构基本上是一样的，都是先解订所有与事务相关的 Block，然后调用 RecoveryMgr 中的方法，写好相关的提交、回滚日志，最后将事务 cognitive 等待队列中清除。

- Recover()

此方法则是在解订了所有 Block 之后，直接调用 RecoveryMgr 中的方法做恢复。

- getInt(Block blk，int offset)

此方法和 getString()，setInt()，setString()三个方法是相类似的，这里只拿 getint()和 setint()方法来讲解，其他不再一一说明。

从指定的 Block 中的指定位置读取数据的流程是：

1> 加锁；这里由于是“读”，所以加的是共享锁。

2> 取出对应 Block 的 Buffer；

3> 利用 Buffer 取出数据。

这里用 Buffer 而不是直接 IO 是为了解决速度问题。

- setint(Block blk, int offset, int val)

向指定的 Block 的指定位置写入值的流程是：

1> 加锁；这里是“写”，所以加的是互斥锁，即当前状态下是禁止再有读写请求的。

2> 取出对应 Block 的 Buffer；

3> 在写入新值之前，读取 offset 上的旧值，利用 RecoveryMgr 写入一条更新日志；

4> 利用 Buffer 写入新值。

在写入新值的时候，要把旧值写入更新记录，一遍以后做崩溃恢复。

- Size(string filename)

描述：返回指定文件中的块数，filename 是指定文件的名字。

实现：这个方法在执行操作之前，首先获得一个“文件末尾”上的 SLock。

- append(string filename，PageFormatter fmtr)

描述：插入一个新块到指定文件的末尾，并且返回一个对它的引用。

实现：这个方法在执行操作之前，首先获得一个“文件末尾”上的 XLock。

3.2 日志管理模块

3.2.1 日志管理的原理

日志是数据库管理系统对其所执行操作的历史记录，保存在稳定存储的记录文件上，是事务管理子系统的重要组成部分。日志在系统崩溃中能够完整的保存下来，以便在恢复时被恢复管理模块调用。日志通常会在不同的磁盘上维护多个版本，以便增强事务的持久性。

需要写入一条日志记录的操作有：

更新一页：修改某页后，一条类型为更新的记录被添加到日志尾。

提交：当一个事务决定提交时，会强制写一个包含事务标识的类型为 commit 的日志记录。强制写就是指日志记录被添加到日志，并且将含有提交记录的日志尾写入稳定存储。

中止：当一个事务被中止时，包含事务标识的中止日志记录被添加到日志中，并且开始取消这个事务。

结束：在中止或者提交事务时，除写中止或者提交日志记录外，还需要进行其他一些操作。在所有这些操作完成后，包含事务标识的结束日志记录被添加到日志中。

每一条日志记录都具有一些字段：preLSN、transID 和 type。属于某个事务的所有日志记录集合构成一个链表，通过 preLSN 字段可以快速获得前一条记录；当添加新的日志记录时需要更新这个链表。

3.2.2 日志管理模块的设计

TBASE 的日志管理是基于 ARIES。它使用 WAL（写优先日志），并且支持缓冲区管理器强制写策略的使用。日志被维护在一个单独的文件中，对于每一条日志记录，系统都给予一个唯一的标识符，称为日志序列码（log sequence number，简称 LSN）。通过日志序列码，就可以通过磁盘存取操作得到日志记录。LSN 必须以单调增加的凡是来生成。本论文中日志是顺序文件，可以无限增长，LSN 仅仅是从日志开头起的一个偏移量。LSN 的值是由恢复管理器分配的。

1 写优先日志法

日志管理模块采用传统的写优先日志（Write –ahead logging，即 WAL）法。WAL 的具体要求如下：

(1) 任何对数据库对象的修改首先写入日志；

(2) 在将更新的数据库对象写入磁盘前，日志中的记录必须强制地先写入稳定的存储。

WAL 是确保系统从崩溃中进行恢复时所有更有更新日志记录都可用的最主要规则。如果事务进行更新操作并提交，非强制方法意味着在随后某个时刻发生崩溃时某些更新结果还没有写入磁盘。如果没有存储更新记录，那也就没有办法保证已提交事务的更新数据可以在崩溃中幸免。

2 日志记录的插入

日志管理模块不知道日志记录的类型，而仅仅解释日志记录为字节序列，日志管理模块主要和恢复管理器交互。

日志管理器插入一条日志记录的流程图如图 3-8 所示。

图 3-8 中，插入一条日志记录时，首先要检查该记录的长度，如果该记录的长度过长,即该块已经没有足够空间插入此记录，则把块的内容保存到日志文件中，并重新添加一个块，把记录内容重新插入到块内，如果没有满则直接把日志记录放入缓冲区，直到最后写入磁盘，插入成功。

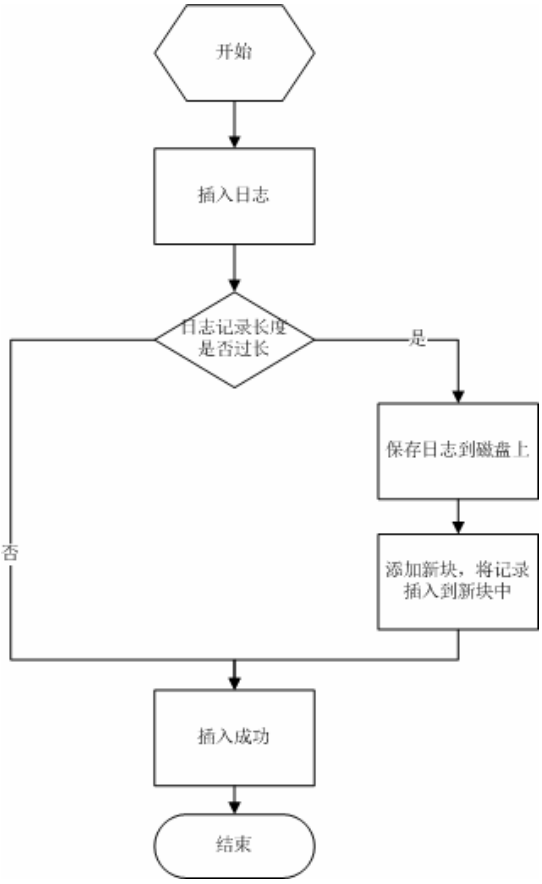

图 3-8 插入日志的流程

3 存储日志记录的页格式

在每一个存放日志记录的页内，在初始化的时候，在每个页的开头四个字节都先写入一个 LAST_POS 值，初始为 0，随着记录的插入其值便开始指向最后一个日志记录的位置，即最后一个日志记录的偏移量。每一个日志记录都是定长的，每一个日志记录的最后四个字节设置一个整数，preLSN，指向前一个日志记录的偏移量，具体如图 3-9 所示：

4 个字节，用于指向最后一条日志记录	
内容	指向前一个日志记录的指针（0）
内容	指向前一个日志记录的指针（int）
•••	
内容	指向前一个日志记录的指针（int）
内容	指向前一个日志记录的指针（int）

图 3-9 保存日志记录的页的结构

图 3-9 中，整个方框代表存放日志记录的一页，每一行（除了第一行）表示

一条日志记录，日志记录的内容随着日志记录的类型的不同而变化，每一条日志记录的最后四个字节即为一个 preLSN。首先，在初始化的时候，在每个 Block 的开头都先写入一个 LAST_POS 值为 0，它的值随着日志记录的插入而变化，总是指向最后一条日志记录，这么做最大的好处就是在插入记录时可以直接找到要插入的位置，而不用再遍历找到要插入的位置。然后，就开始写入日志记录。关键的部分是在写完日志之后。在写完每条日志之后，会加上一个便宜量，这个偏移量用来记录前一条记录的 offset。

4 日志记录迭代器

论文为日志记录创建了一个迭代器，用于以逆序遍历所有的日志记录，由于在每一个日志记录的最后四个字节保存着指向前一个日志记录的指针，所以可以很方便的从当前日志记录移动到下一个日志记录。

3.2.3 日志管理模块的实现

日志管理模块中实现了三个类，依次是 BasicLogRecord，LogIterator，LogMgr。

在日志模块中最核心的部分是日志的记录方式。日志记录在写入文件的时候是从前往后记录的，但是在读的时候却是以相反的方向进行的。在论文中采用一种独特的日志记录方式来实现这种功能。如前面图 3-9 所示。

在详细介绍整个日志管理模块之前，首先详细讲解一下下面用以添加记录，确定偏移量，以及读取记录的方法，现在可以详细查看图 3-10 中添加日志记录的代码。

```
public int append(Object[ ] rec){
        lock (this){
                int recsize = ConstantValue.INT_SIZE;
                foreach (Object obj in rec)
                    recsize += size(obj);
                if (currentpos + recsize >= ConstantValue.BLOCK_SIZE){
                    flush();
                    appendNewBlock();
                }
                foreach (Object obj in rec)
                    appendVal(obj);
                finalizeRecord();
                    return currentLSN();
            }
}
```

图 3-10 添加日志记录

从图 3-10 可以看出，这里的“记录”是记录内容和 offset 的结合体，在计算空间的时候，会自动加上在记录结尾 offset 所需要的空间。

再看图 3-11 中所示的代码，在写完一条记录值之后，currentpos 的值是真正记录结束后的位置，不包含前一条记录的 offset。系统中利用 finalizeRecord()来设置 offset。

```
private void finalizeRecord(){
        mypage.setInt(currentpos, getLastRecordPosition());
        setLastRecordPosition(currentpos);
        currentpos += ConstantValue.INT_SIZE;
}
```

图 3-11 设置偏移量

以上的三条语句：

1> 把 LAST_POS 的值卸载记录的后面，实际上就是给 offset 赋值

2> 把 currentpos 赋值给 LAST_POS

3> 给 currentpos 加上 4，跳过 offset

通过以上的介绍，很容易明白记录 offset 的过程，

接下里将详细讲解具体实现中的各个类以及其成员变量和函数。 首先展示一下日志管理器子模块的类图，如图 3-12 所示。

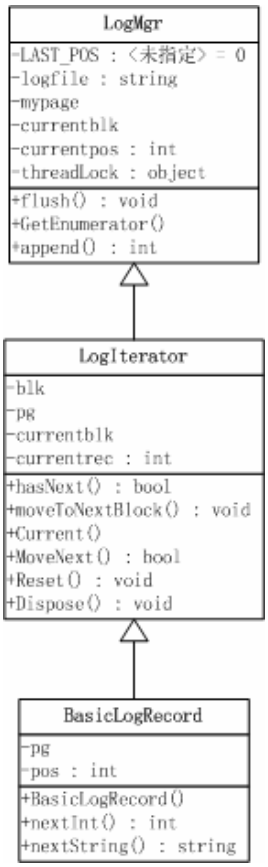

图 3-12 日志管理模块类图

1 BasicLogRecord

（1）概述

BasicLogRecord 提供的是一种基础的日志记录。它所完成的功能就是定位 Page，定位 Offset，然后提供读写记录的方法，但它本身并不知道这些值是什么，成员方法 nextInt()和 nextString()顺序地读这些值，客户端有必要知道日志记录中有多少个值和这些值的类型。基本日志记录类定义的字段和方法如图 3-13 所示。

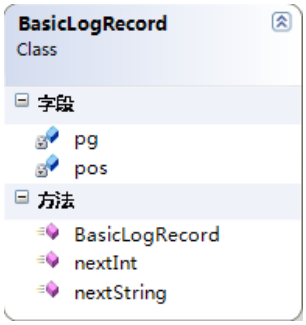

图 3-13 基本日志记录类

（2）成员变量

pg：此变量为日志记录所在的页。

pos;此变量为日志记录在页中的偏移量。

（3）成员方法

- **public** BasicLogRecord(Page pg, **int** pos)；

描述：构造函数，根据指定的页和指定的页内偏移创建一条日志记录。

实现：将参数列表中的参数值赋值给成员参数。

- **public int** nextInt();

描述：返回当前记录中的下一个值，该值被假设为整数。

实现：1.调用 Page 的 getInt()方法，在页内指定偏移处读取一个整数。

2.将当前位置 pos 加一个整数长度 INT_SIZE。

3.返回结果。

- **public** String nextString();

描述：返回当前记录中的下一个值，该值被假设为字符串。

实现：1.调用 Page 的 getString()方法，在页内指定偏移处读取一个字符串。

2.将当前位置 pos 加上该字符串大小。

3.返回结果。

2 LogIterater

（1）概述

LogIterator，日志迭代器，提供逆向遍历日志文件中所有记录的功能。它所拥有的字段和方法如图 3-14 所示。

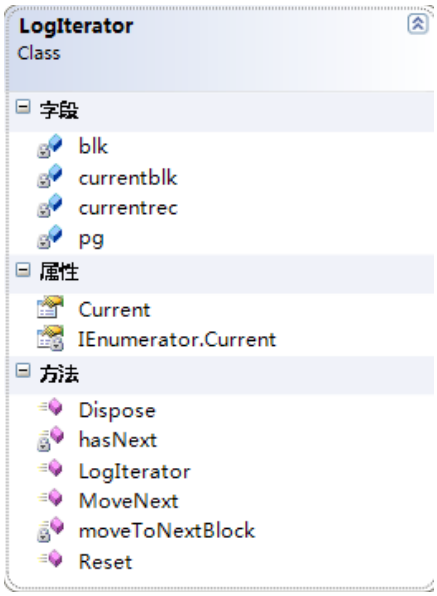

图 3-14 日志迭代器类

（2）成员参数

blk：此变量定义了一个块。

pg：此变量新建了一页。

currentrec;：此变量用于记录当前记录。

currentblk：此变量用于记录当前块。

（3）成员方法

- LogIterator(Block blk);

描述：构造函数，给日志文件的记录创建一个迭代器，并定位到最后一条日志记录的后面。

实现：1.将参数列表中的 blk 赋值给成员参数 blk。

2.将该块的内容读入 Page。

3.将当前记录设置为最后一条日志记录的后面一条，通过读取存于块开始位置的整数来完成。

- **public boolean** hasNext();

描述：用来判断当前日志记录是否是日志文件中最早的记录。

实现：当前记录 currentrec 大于 0 或者当前块号大于 0 时，当前记录不是最早的记录，逆向有下一条记录，返回 true。

- **public** BasicLogRecord Current();

描述：逆向移动到下一条日志记录，如果当前记录在其块中为最早的记录，那么该方法将会移动到一个更早的块，并返回一条日志记录。

实现：1.判断当前记录是否为块中最早的记录，即 currentrec 是否等于 0。

2.如果当前记录是块中最早的记录，则逆向移动到下一个块，通过调用成员方法 moveToNextBlock()来完成。

3.调用 Page 的 getInt()方法从 currentrec 位置处读取下一条记录的偏移，并作为 currentrec 的新值。

4.返回一条日志记录，Page 为 pg，pos 等于 current 加一个整数长度。

- **public void** remove();

描述：删除当前日志记录。

实现：抛出异常 UnsupportedOperationException。

- **private void** moveToNextBlock();

描述：逆向移动到下一个块，并且定位到块中最后一条记录的后面。

实现：1.新建一个 Block 引用，文件名等于当前块的文件名，块号等于当前块号减 1，赋值给 blk。

2.将该块的内容读入 Page。

3. 将当前记录设置为最后一条日志记录的后面一条，通过读取存于块开始位置的整数来完成。

3 LogMgr

（1）概述

LogMgr，底层的日志管理器，只是负责日志记录的读写，一条日志记录可以是任意顺序的整数值和字符串值的集合，它自己并不知道所写的日志的记录的内容是什么。所谓较高级的日志管理器是指恢复管理器，它可以明确的知道所写的日志记录的内容。图 3-15 展示了日志管理器的类。

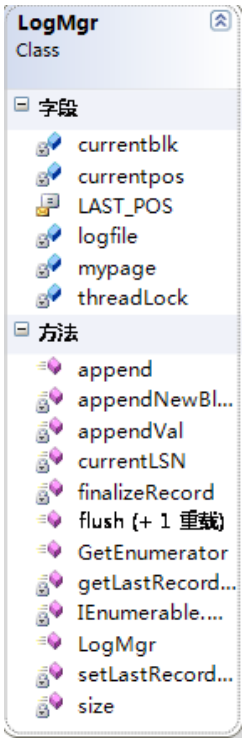

图 3-15 日志管理器类

（2）成员参数

LAST_POS; 指向一个页中最后一个整数的指针所指的位置。若值为0，则意味着这个指针指向那个页中的第一个值。

logfile：用于表示日志文件的名字。

mypage：新建的一页。

currentblk：此变量表示当前块

currentpos：此变量表示当前记录的位置

（3）成员方法

- **public** LogMgr(String logfile);

描述：为指定的日志文件创建日志管理器，如果该日志文件不存在，那么创建一个有一个空块的文件。

实现：1.将参数 logfile 赋值给成员参数 logfile。

2.调用 FileMgr 的 size()方法获得日志文件 logfile 的磁盘块数。

3.如果该日志文件磁盘块数为 0，即文件不存在，那么通过调用成员方法 appendNewBlock()来创建该文件。

4.如果该文件存在，创建一个磁盘块引用引用 logfile 的最后一个磁盘块并赋值给 currentblk，将该磁盘块内容读入 mypage，并将当前位置 currentpos 设置为该块中已用空间的后面可写的位置处。

- **public void** flush(**int** lsn);

描述：保证对应于指定 LSN 的日志记录被写到磁盘上，所有更早的日志记录同样也被写到磁盘上。

实现：调用成员方法 currentLSN()获得当前磁盘块号，如果指定的 LSN 大于等于当前 LSN 则调用成员方法 flush()来执行写操作。

- **public IEnumerator** <BasicLogRecord> GetEnumerator();

描述：返回一个日志记录的迭代器，从最近的块末尾开始逆向返回。

实现：1.调用 flush()成员方法将当前页写入日志文件。

2.新建一个 LogIterater 对象，将当前块 currentblk 作为参数，并将其返回。

- **public int** append(Object[] rec);

描述：向文件附加一条日志记录，记录包含任意的字符串和整数组成的数组，这个方法会在每一条日志记录的末尾存一个整数，这个整数是该记录的前一条记录的偏移，这些整数允许日志记录被逆向读取。

实现：1.定义记录大小 recsize 等于整数大小 INT_SIZE，因为要在记录末尾存一个整数。

2.使用一个 for 循环，对于该条日志记录的每一个值，recsize 增加该值的大小。

3.检测当前块是否可以容纳该记录，如果不可以，即 currentpos 加 recsize 大于等于块大小 BLOCK_SIZE，则将该块写入文件，并通过调用 appendNewBlock()来附加一个新块。

4.使用一个 for 循环，调用 appendVal()方法将记录中的每一个值写入。

5.调用 finalizeRecord()成员方法完成添加记录操作。

6.调用 currentLSN()获得当前磁盘块号并将其返回。

- **private void** appendVal(Object val);

描述：添加指定的值到 page 的当前位置 currentpos 处，然后给 currentpos 增加该值的大小。

实现：1.如果该值是字符串值，调用 page 的 setString()方法将该值存放到当前位置 currentpos 处，如果是整数值，调用 page 的 setInt()方法将该值存放到当前位置 currentpos 处。

2.调用 size()成员方法获取值 val 所占的空间大小，并增加 currentpos 的值。

- **private int** size(Object val);

描述：计算指定的整数或字符串所占的空间大小。

实现：如果该值为字符串类型，调用 String 的 length()方法和 STR_SIZE()方法来获得该字符串值的大小，并将其返回。如果是整数类型，则直接放回整数大小 INT_SIZE。

- **private int** currentLSN();

描述：返回最近的日志记录的 LSN，作为实现，这个 LSN 的值就是存储该记录的磁盘块的块号，因此同一磁盘块上的记录拥有相同的 LSN。

实现：放回当前磁盘块 currentblk 的块号 number。

- **private void** flush();

描述：将当前页写到日志文件。

实现：调用 page 的 write()方法将 mypage 的内容写到指定的磁盘块 currentblk 上。

- **private void** appendNewBlock();

描述：清空当前页，并将其附加到日志文件。

实现：1.调用 setLastRecordPosition()成员方法将最后一条记录的位置设置为 0，即没有记录。

2.将当前位置 currentpos 设置为整数大小 INT_SIZE，因为在块首要存放一个整数。

3.调用 page 的 append 方法将该 page 的内容作为一个新块附加到日志文件，并将返回到块号赋值给 currentblk。

- **private void** finalizeRecord();

描述：建立一个指向页内记录的循环指针链，在每一条日志记录的末尾添加一个整数用来存放它的前一条记录的偏移值，页首的 4 个字节包含一个整数值用来存放页内最后一条记录的偏移的偏移。

实现：1.调用 page 的 setInt()方法在记录的后面存放一个整数，该值由 getLastRecordPosition()获得，为插入该记录前存于 page 首部的末尾的偏移。

2.调用 setLastRecotdPosition()成员方法将 currentpos 的值存放到 page 首部。

3.currentpos 加一个整数大小 INT_SIZE。

- **private int** getLastRecordPosition();

描述：返回存在页首的页内最后一个整数的偏移。

实现：调用 page 的 getInt()方法返回偏移为 0 处的整数。

- **private void** setLastRecordPosition(**int** pos);

描述：设置存在页首的页内最后一个整数的偏移。

实现：调用 page 的 setInt()方法设置偏移为 0 处的整数值。

3.3 恢复管理模块

3.3.1 恢复管理的原理

TBASE 的恢复策略是基于 ARIES[13]的，ARIES 是一个常用的与并发控制方法相结合的恢复算法[7]。使用该方法，系统崩溃后调用恢复管理器，重新启动时分下面三个阶段恢复数据库。

分析：识别出缓冲池中的脏页（也就是已经进行了修改但是还没有写入磁盘的页）和崩溃时当前事务。

重做：从日志的某个合适点开始，重复所有操作，将数据库恢复到崩溃时所处的状态。

反做：取消没有提交的事务操作，使数据库只反应已提交事务的操作结果。

在 ARIES 恢复算法背后蕴含着下面三个基本原理。

写优先日志法[5]（Write-ahead logging）：热河对数据库对象的修改首先写入日志；在将更新的数据库对象写入磁盘前，日志中的记录必须先写入稳定的存储。

重做时重复历史：在崩溃后进行重新启动时，ARIES 重新追踪 DBMS 在崩溃前的所有操作，使系统恢复到崩溃时一样的状态。然后取消崩溃时还在执行的事务所做的操作（有效地取消它们的影响）。

恢复修改的记录数据：在反做某些事务时，如果出现对数据库对象的改变，则需要在日志中记录这些改变。这能够保障在重复进行重新启动时（由故障引起），不需要重复这些操作。

ARIES 恢复算法是一个非常简单、灵活的算法，而且还支持涉及比页级更低粒度锁（如记录级）的并发控制协议。但是由于时间和个人能力的限制，在本论文中，只实现了三个阶段中的第一和第三阶段，而省略了重做阶段，但是仍然能够达到当初的需求。

恢复管理器确保数据库的持久性。它在两种模式下操作。在正常的数据库操作期间运行于 Normal Mode，在崩溃重启期间运行于 Restart Mode 。

当运行于 Normal Mode 时，恢复管理器通过支持中止事务的回滚来支持事务的原子性。它也实现检查点来帮助从崩溃中恢复。恢复管理器也提供在正常操作模式下写日志记录的方法。Restart Mode 在崩溃之后被系统调用。

在正常操作期间，恢复管理器提供了把更新和提交操作的日志写到日志文件

的方法。也就是说，管理器关注更新和提交日志记录的格式。

恢复管理器真正的工作是在崩溃之后。重启程序关注三个阶段：分析，重做和取消，并且恢复系统到一致性状态。分析阶段恢复管理器从最近的检查点的日志记录开始进行处理。重做阶段重复历史，它执行所有被记录的操作。取消阶段从最后一个日志开始向后移动，删除所有在崩溃是还没有提交的事务的影响

恢复管理器也关注检查点，在设置检查点时，脏页表和事务表被写到日志文件中。崩溃后，分析阶段从最后一个检查点的日志记录开始。

3.3.2. 恢复管理模块的设计

恢复管理器为检查点、回滚和重启提供方法。恢复管理器主要和日志管理器交互。恢复管理器的一个功能就是产生不同操作（中止，更新，提交）的日志记录。恢复管理器创建被日志管理器写到日志文件中的日志记录。恢复管理器也和低级的缓冲区管理器和事务管理器交互。在事务回滚时被事务管理器调用。它为建立脏页表请求缓冲区管理器，为建立事务表而请求事务管理器。图 3-16 显示了恢复管理器和其他子系统的交互。

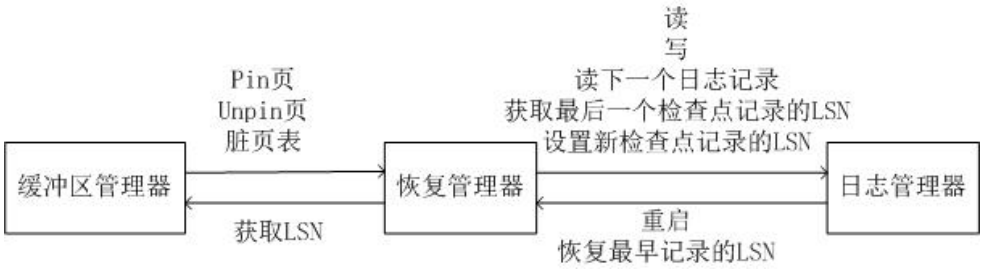

图 3-16 恢复管理器和其他子系统的交互

从图 3-14 中可以清楚地看到，恢复管理模块和日志管理模块有着很大的关系，前面在介绍日志管理模块时已经提过日志管理器自己并不知道自己锁写记录的内容是什么，而现在答案就很明显了，真正理解日志管理器锁写内容的是恢复管理器。同时日志管理器与缓冲区管理器也有很密切的联系。

在介绍 RecoveryMgr 之前，先对恢复时要用到的日志的各种情形做一个归纳，如图 3-17 所示。

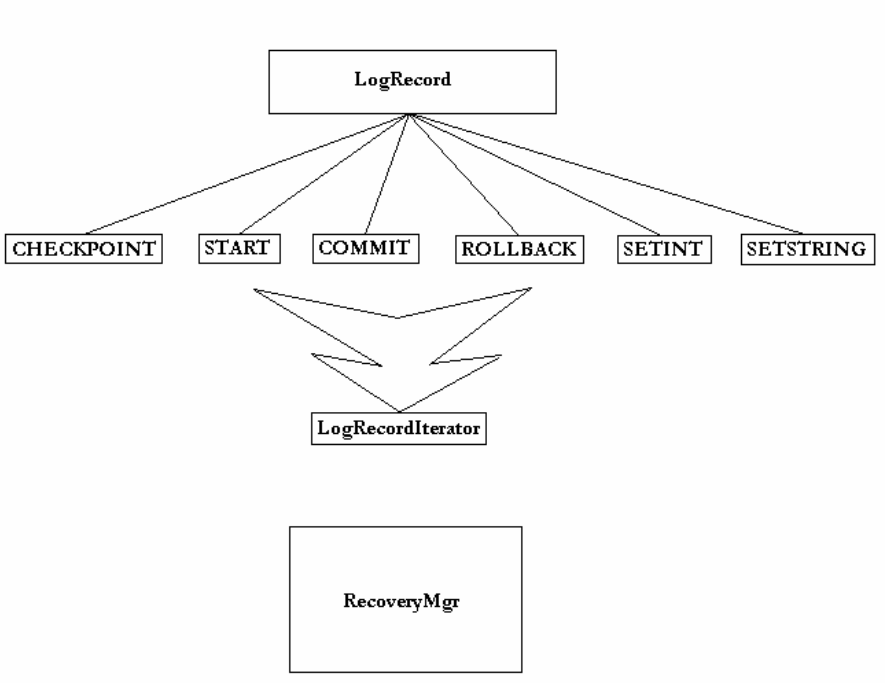

图 3-17 恢复管理器使用的日志记录示意图

从图 3-15 可以看出，恢复管理器要用到六种日志记录，分别为检查点日志，开始日志，提交日志，回滚日志，设置整型日志，设置字符串日志，这六种日志记录共同继承自同一个日志记录接口，然后通过日志记录迭代器来读取每种日志记录的的内容。Recovery 的工作机制了。

3.3.3 恢复管理模块的实现

首先来展示一下恢复管理器子模块的类图，如图 3-18 所示。

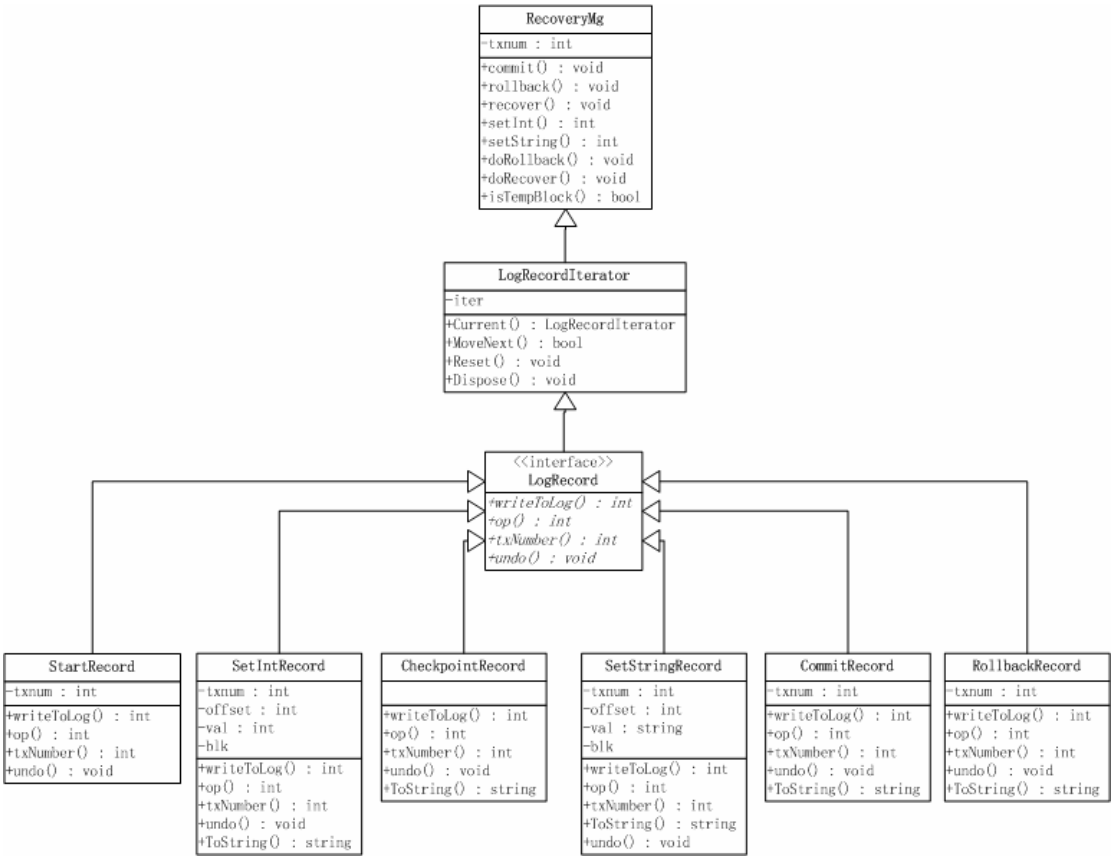

图 3-18 恢复管理子模块的类图

1 LogRecord

(1) 概述

这是一个接口，它提供了关于恢复的六种日志所需要的基本方法。它的类关系图如图 3-19 所示。

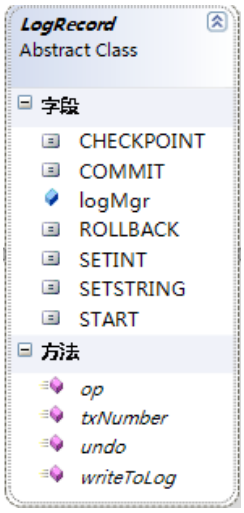

图 3-19 日志记录类

(2) 成员变量

它有六个成员变量，分别是 CHECKPOINT, START, COMMIT, ROLLBACK, SETINT, SETSTRING，它们是整型常量，值分别为 0，1，2，3，4，5。在后面依据它们的值来判读日志的类型。

(3) 成员方法

该类拥有 4 个成员方法，这四个方法分别被六种不同的日志记录所实现。在这里，恢复日志中的六种记录是变长的，各种不同的记录用在 LogRecord 中定义的常量做标记、区分。

在这六种记录中 SetInt 和 SetString 是比较特殊的，两者可以归作一类，下面以 SetInt 为例进行着重讲解。

2 SetIntRecord

(1) 概述

这是其中的一种日志记录，主要就是写一个设置整型的日志记录。其类关系图如图 3-20 所示。

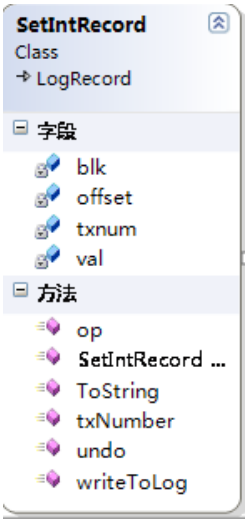

图 3-20 设置整型记录类

SetInt 的记录格式如图 3-21。

SETINT	TxID	Filename	Val	Offset	PreVal

图 3-21 SetInt 记录的格式

(2) 成员变量

该类有四个成员变量，分别为：txnum，offset，val，和 blk。此处可以用一句话来解释这四个成员变量的含义：在事务号为 txnum 的事务中，想在块 blk 的 offset 位置写入整数值 val。

(3) 成员方法

它有两个构造函数，分别做不同的用途：

- public SetIntRecord(int txnum, Block blk, int offset, int val)
- public SetIntRecord(BasicLogRecord rec)

第一个构造函数是利用写入的新值来赋值，是用来创建新的日志记录。

第二个构造函数是利用已有的记录来赋值，是在恢复的时候，即时建立一条记录，作恢复用的。

SetInt 中的 undo 方法需要特别说一下，详细代码如图 3-22 所示。

```
public override void undo(int txnum){
        BufferMgr buffMgr = TBaseConfig.bufferMgr();
        Buffer buff = buffMgr.pin(blk);
        buff.setInt(offset, val, txnum, -1);
        buffMgr.unpin(buff);
}
```

图 3-22 反做方法示意图

Undo就是把之前所做的修改取消，所以，方法的实现就是把之前存储在记录中的旧值写回原来的位置，取消完成的事务。这个方法实现过长如下：首先pin指定块的一个缓冲区，然后调用setInt()来重新存储被保存的值，最后unpin之前被pin的缓冲区。

3 LogRecordIterator

日志记录迭代器的实现比较简单，其实质就是封装了一个 LogIterator，在迭代上没有太多要讲述的。

在这里，需要强调的一点是，它在获取 next()的时候，利用一个 switch 语句，根据记录的 op（即存储的关于记录类型的标记）的不同，从而建立不同的日志记录。

4 RecoveryMgr

(1) 概述

RecoveryMgr 是恢复管理模块的核心部分，这个类的成员变量只有一个，即 txnum，它面向的是一个事务，每一个事务都有它自己的恢复管理器。RecoveryMgr 会伴随事务执行的各个阶段。恢复管理器的类关系图如图 3-23 所示。

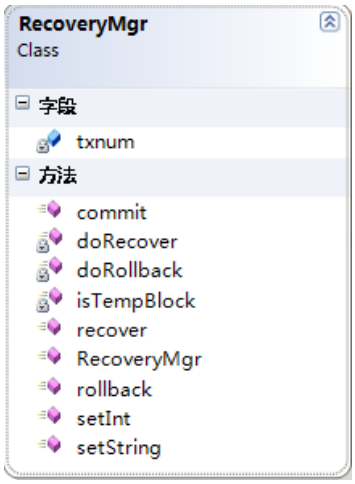

图 3-23 恢复管理器类

(2) 成员方法

➢ public RecoveryMgr(int txnum)

这是一个构造函数，为指定的事务创建一个恢复管理器。在前面的介绍中看到，RecoveryMgr 是 Transaction 的一个成员变量。在事务初始化的时候，即 RecoveryMgr 构建的时候，就已经开始写入 Start 记录了。

➢ public void commit()

此方法的主要作用是写一个提交记录到日志，并且把它刷到磁盘上。具体做法是：先把事务锁住的页释放，然后写入 Commit 记录，最后在再立刻把写好的 Commit 记录写回稳定的存储。

➢ public void rollback()

写一个回滚记录到日志，并且把它刷到磁盘上。这里和上面的操作的形式差不多，只是增加了一个做 Rollback 的操作。

➢ public void recover()

从日志中恢复未完成的事务，然后写一个静态检查点记录到日志并把它刷到磁盘。和上面两个唯一不同的是：事务不需要释放钉住的页。很显然，一个需要被恢复的事务，就不再有块被 pin。

通过上面的三个方法，可以发现，在每次日志记录生成之后，就立马写回硬盘，这样保证了恢复日志的绝对安全，同时，降低了崩溃恢复的难度。在这里，并没有使用通常提到的“偷帧”和“非强制写页”的策略，而是采用了强制写页的方式。

➢ public int setInt(Buffer buff, int offset, int newval)

写一个 setint 记录到日志，并返回它的 lsn。这里基本上的原理还是和之前介绍的一样的：在写入新值之前，把旧值记录下来，做好恢复日志。只是，这里有一个不同的情况是，有可能事务是在对临时 Block 做 SetInt()修改操作。而按

照系统的规定，对临时 Block 的操作是不需要写入日志的，所以在判断 Block 为 Temp 之后，返回-1，而不是 lsn。

- private void doRollback()

在做回滚的时候，现在要在当前情况想产生一个日志的迭代器，以输出记录。然后，根据日志记录开始反做事务。而记录的 op 则是提供了判断某个事务是否被反做成功的标记：对某个事务被反做的时候，当读到的记录的 op 为 START 的时候，说明事务在 start 之后的操作均被反做，反做成功。LogIterator 提供日志记录的方法和事务执行的方向是相反的。

- private void doRecover()

做一个完全的数据库恢复。在这里，首先获取一个所有已提交事务号的数组，用以向其中添加 Commited 的事务的编号：doReover 的目的就是要让更多的事务 Commited，自然完成之后要把“成果”记录下来。

在利用迭代器遍历日志的时候，如果遇上了已提交的事务，会把相应的 Txnum 记录下来；如果发现某些记录的 Txnum 不再 CommitedTxs 集合中，则认为这些事务没有处理完成，这些事务会被反做；如果发现了有 CHECKPOINT，则停止对此事务的操作。

上面的 Rollback 和 Recovery，通过查询记录的形式，保证了事务的原子性。实现这些所用的方法实质上还是利用了 ARISE 算法的原理，只是论文将其实现简单化了，在写完一条日志后就直接刷到了硬盘里面，保证了 log 的完整、正确，这样的实现自然就比较简单，但是却不影响系统的可行性和正确性。

第四章　事务管理子系统的测试

4.1 系统测试概述

事务管理子系统完成之后，论文将它与存储管理子系统和查询优化子系统（另外数据库管理系统的另外两个必不可少的模块）进行了整合，，由于三个模块间相互交互、相互依赖的程度比较高，尤其是事务管理子系统与存储管理子系统之间的联系，所以只有将三个模块结合起来才能进行各项测试，之后论文采用黑盒测试的方法，来测试系统的各项功能是否能够正确执行。

在进行各项测试之前，首先在此简单介绍一下黑盒测试的原理：

黑盒测试，也称功能测试，它是通过测试来检测每个功能是否都能正常使用。在测试中，把程序看作一个不能打开的黑盒子，在完全不考虑程序内部结构和内部特性的情况下，对程序接口进行测试，它只检查程序功能是否按照需求规格说明书的规定正常使用，程序是否能适当地接收输入数据而产生正确的输出信息。黑盒测试着眼于程序外部结构，不考虑内部逻辑结构，主要针对系统的功能进行测试。

4.2 日志记录的测试

日志是记录外界对数据库进行的修改或更新，此模块主要测试对数据库的各项更新是否正确的生成了日志记录，能够生成日志记录的操作有一下几种：创建表，插入记录，修改记录，删除记录。下面就分别就创建表和插入记录两种情况来展示本系统可以可以正确的生成相应的日志

4.2.1 创建表

在创建表之前，首先创建一个空的数据库，创建完成之后，数据库所在目录如图 6-1 所示。

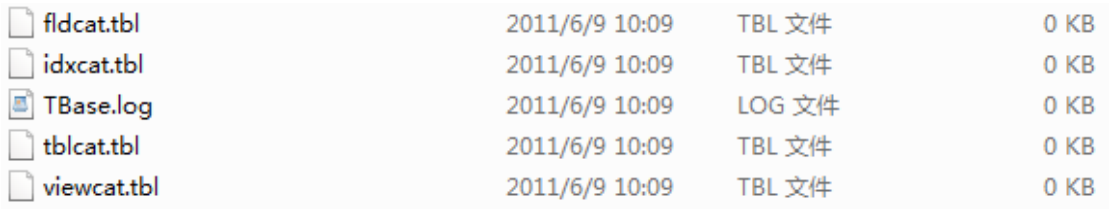

图 6-1 初始数据库文件

可以看到此时日志文件为空

然后在数据库中创建一个表，创建语句如下。

CREATE TABLE P (PPNO VARCHAR(3), PNAME VARCHAR(10), COLOR VARCHAR(8), WEIGHT INT);

结果在数据库所在目录下生成了一个日志文件，如图 6-2,6-3 所示。

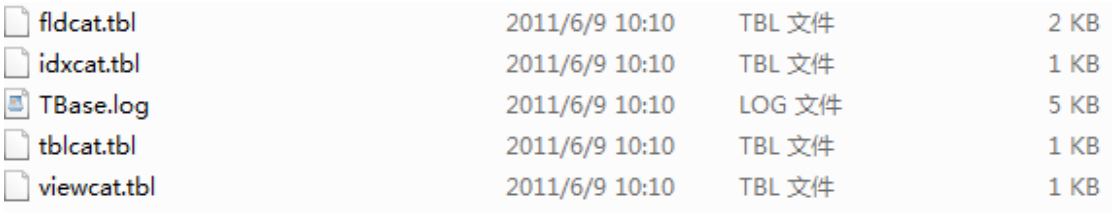

图 6-2 插入数据后数据库的变化

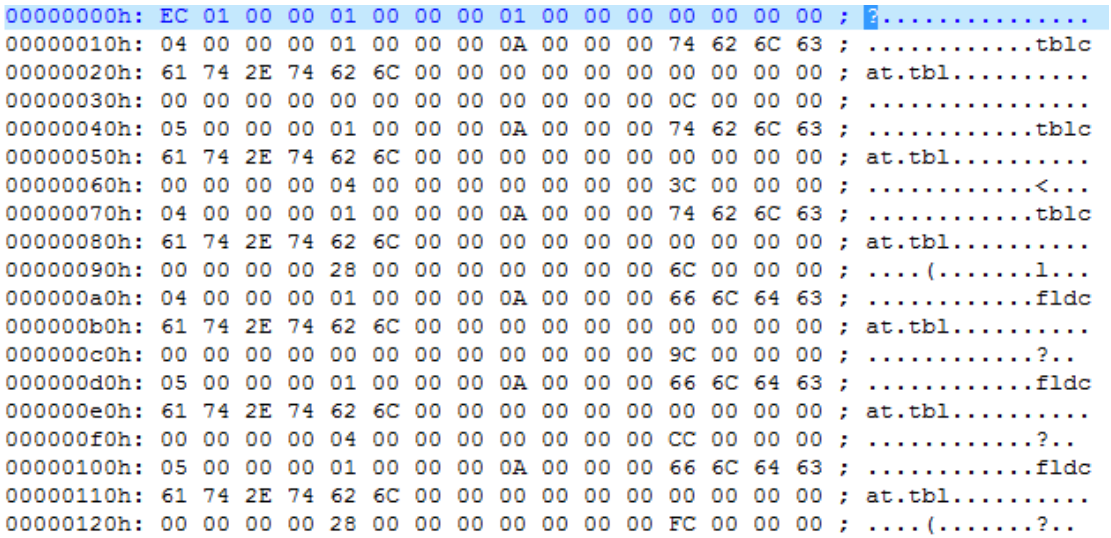

图 6-3 日志记录的内容

此时可以看到日志记录的大小发生了变化，而且 UltraEdit 查看可知日志文件有了具体的内容。

4.2.2 插入记录

然后向表中插入四条记录，插入语句如下。

INSERT INTO P(PPNO,PNAME,COLOR,WEIGHT) VALUES('P1','nut','red',12);

INSERT INTO P(PPNO,PNAME,COLOR,WEIGHT) VALUES('P2','bolt','green',17);
INSERT INTO P(PPNO,PNAME,COLOR,WEIGHT) VALUES('P3','screwdriver',
'blue',14);
INSERT INTO P(PPNO,PNAME,COLOR,WEIGHT) VALUES('P4','screwdriver',
'red',14);

此时在此查看是否有相应的日志记录产生，如图 6-4，6-5 所示。

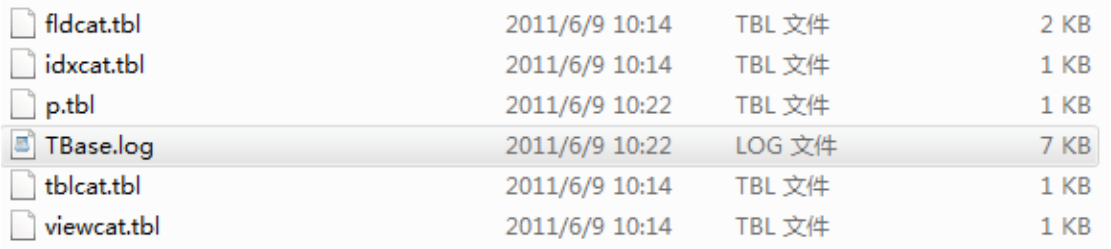

图 6-4 变化后的数据库

```
00000210h: 66 6C 64 63 61 74 2E 74 62 6C 00 00 74 62 6C 63 ; fldcat.tbl..tblc
00000220h: 61 74 2E 74 00 00 00 00 5C 00 00 00 00 00 00 00 ; at.t....\.......
00000230h: 00 00 00 00 05 00 00 00 01 00 00 00 0A 00 00 00 ; ................
00000240h: 66 6C 64 63 61 74 2E 74 62 6C 00 00 74 62 6C 63 ; fldcat.tbl..tblc
00000250h: 61 74 2E 74 00 00 00 00 80 00 00 00 00 00 00 00 ; at.t....€.......
00000260h: 30 00 00 00 04 00 00 00 01 00 00 00 0A 00 00 00 ; 0...............
00000270h: 66 6C 64 63 61 74 2E 74 62 6C 00 00 74 62 6C 63 ; fldcat.tbl..tblc
00000280h: 61 74 2E 74 00 00 00 00 A4 00 00 00 00 00 00 00 ; at.t....?......
00000290h: 60 00 00 00 04 00 00 00 01 00 00 00 0A 00 00 00 ; `...............
000002a0h: 66 6C 64 63 61 74 2E 74 62 6C 00 00 66 6C 64 63 ; fldcat.tbl..fldc
000002b0h: 61 74 2E 74 00 00 00 00 A8 00 00 00 00 00 00 00 ; at.t....?......
000002c0h: 90 00 00 00 04 00 00 00 01 00 00 00 0A 00 00 00 ; ?...............
000002d0h: 66 6C 64 63 61 74 2E 74 62 6C 00 00 66 6C 64 63 ; fldcat.tbl..fldc
000002e0h: 61 74 2E 74 00 00 00 00 AC 00 00 00 00 00 00 00 ; at.t....?......
000002f0h: C0 00 00 00 04 00 00 00 01 00 00 00 0A 00 00 00 ; ?...............
00000300h: 74 62 6C 63 61 74 2E 74 62 6C 00 00 66 6C 64 63 ; tblcat.tbl..fldc
00000310h: 61 74 2E 74 00 00 00 00 2C 00 00 00 00 00 00 00 ; at.t....,.......
00000320h: F0 00 00 00 05 00 00 00 01 00 00 00 0A 00 00 00 ; ?...............
00000330h: 74 62 6C 63 61 74 2E 74 62 6C 00 00 66 6C 64 63 ; tblcat.tbl..fldc
```

图 6-5 改变后的日志记录的内容

可以看到日志记录的大小及日志文件的 内容均有变化，说明修改数据库的操作已经正确写下日志记录。

4.3 并发控制的测试

此模块主要通过同时向数据库中插入多条记录来测试系统是否能够处理并发的情形。此处演示向数据库中同时插入三条语句，具体插入语句如下。

INSERT INTO S(SSNO,SNAME,STATUS,SCITY) VALUES ('S1' , 'Andrew' , 20 , 'Beijing') ;

INSERT INTO P(PPNO,PNAME,COLOR,WEIGHT) VALUES('P1','nut','red',12);
INSERT INTO J(JJNO,JNAME,JCITY) VALUES('J1','radio','Beijing');

插入后事务执行结果如图 6-6，6-7 所示。

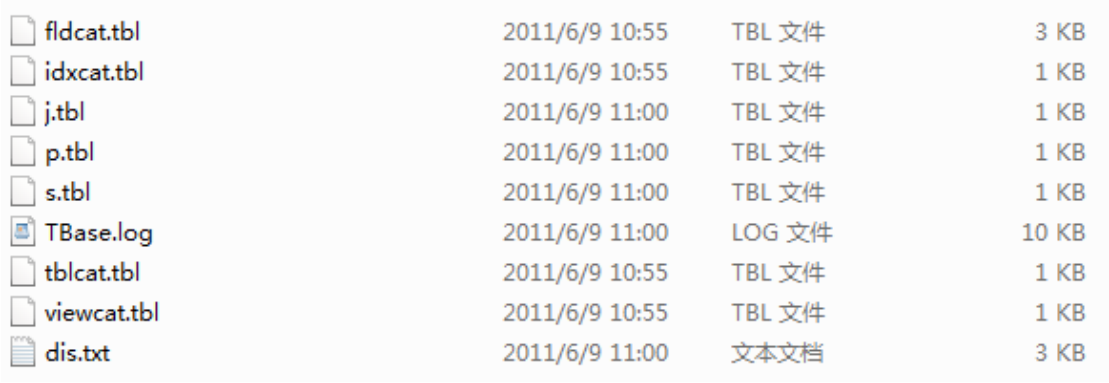

图 6-6 数据库的变化

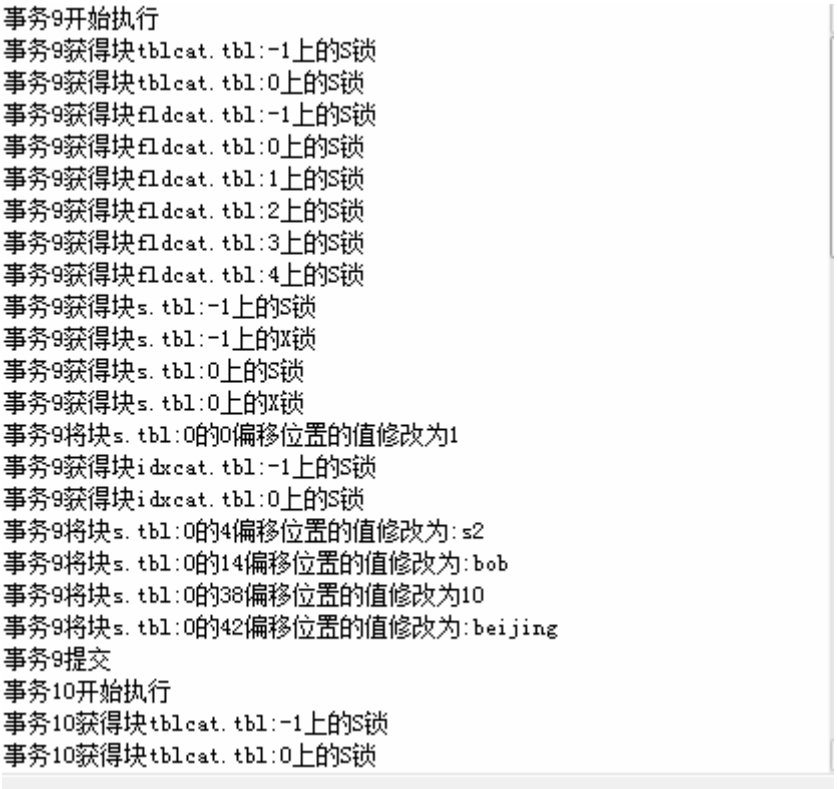

图 6-7 事务执行调度序列

由图 6-7 中可以看到，并发执行中事务的调度序列以及事务所执行的操作，从中可以明确看出事务在并发执行过程中没有出现冲突，可以正确的进行并发操作，由此可以证明本系统可以正确处理并发的情形。

第五章　总结与展望

5.1 总结

伴随着数据库行业的飞速发展，各行各业需要用到数据库的地方也越来越多，海量数据的处理对数据库的要求也越来越高，而且数据的种类也发生了很大的变化，所以数据库管理系统的作用也就愈加的重要，在面对众多客户同时访问数据库的情况下，数据库管理系统中事务管理的作用更是成为重中之重。

本论文的主要工作在于以下几个方面。

本论文就针对数据库管理系统中的事务管理子系统的，TBASE 设计的初衷就是为了用于教学。本论文和另外两篇论文（存储管理和查询优化）可以完整的结合成为一个小型数据库管理系统，能够实现一个数据库管理系统应该具有的全部基本功能，这对他人了解数据库，运用数据库提供了一个很好的平台，为以后研究数据库管理系统奠定坚实的基础。

论文对事务管理子系统的设计和实现做了比较详细的阐述，通过架构图，流程图和类的设计说明，使得读者更加容易把握整体，理解细节，弄明白整个系统。

论文深入的讲述了事务管理子系统，把它分为三大模块，分别是：锁管理模块，日志管理模块和恢复管理模块，对它们分别进行了详细的设计和实现，实现中使用了很多经典的算法。

1 锁管理模块：

第一：使用了两种锁模式，即共享锁和互斥锁，来保证几个不同事务在交叉工作时它们锁产生的结果等同于所有事务按照某种串行顺序执行的结果。

第二：使用了锁升级的策略，以此来简化两阶段加锁的实现，同时又能起到两阶段加锁的效果。

第三：使用了一个全局的锁表，来记录所有持有锁的事务，以此来保证所有事务在开始时，获取共享锁以读取数据对象，获取互斥锁以修改数据对象，然后在结束时释放所有这些锁，并且不会有冲突，确保数据库的一致性。

2 日志管理：

第一：设计了存放日志记录的也的格式。每一页开始的四个字节存放最后一个日志记录的偏移位置，日志记录是从后向前存储的，每条日志记录的最后四个字节存储前一个日志记录的偏移量。

第二：使用每一个日志记录的在页中的偏移量作为日志序列码（log sequence number，LSN），以此来唯一的确定一条日志记录，LSN 是单调递增的方式生成。

3 恢复管理模块：

第一：实现了检查点记录，开始记录，提交记录，设置整型记录和设置字符串记录五种日志记录，以此作为系统崩溃或者其他故障导致的恢复时的依据。

第二：实现了写优先日志的策略，使得对数据库的任何修改在写到稳定存储前，其修改的日志记录首先被写到稳定存储上。

第三：使用分析和反做两个阶段来完成在系统崩溃后重新启动时对数据库的恢复。

5.2 展望

本系统已经实现了最初的需求，完成了需要完成的功能，但是论文中还有一些不尽如人意的地方，以后的工作主要有以下两个方面：

1 锁管理模块

(1) 使用新类型的锁，即使用意向共享锁和意向排它锁，来实现多粒度锁协议。

(2) 增加死锁检查和死锁预防模块，避免死锁情形的发生，增加系统的稳定性和可靠性。

2 恢复管理模块

在恢复管理模块中添加重做这一个阶段，本系统的恢复是基于 ARIES 的，虽然可以正确的实现恢复，但是却简化了其实现。

本系统目前还只是一个简单的原型系统，论文中还存在的一些些不足之处，这些地方都需要投入大量时间和精力来改善，例如上面所提到的这些，在以后的时间里面会慢慢来弥补和完善。

参考文献

[1]C.MOHAN,B.LINDSAY,and R.OBERMARCK. Transcation Mannagement in the R* Distributed Database Management System. ACM Transactions on Database Systems, 1986, Vol.11,No.4, 378-396.

[2]Ibrahim Jaluta, Seppo Sippu and Eljas Soisalon-Soininen. Concurrency control and recovery for balanced B-link trees[J]. The VLDB Journal, 2005, 14(2): 257 – 277.

[3]Mohan, C. An Efficient Method for Performing Record Deletions and Updates UsingIndex Scans[A], Proc. 28th International Conference on Very Large Databases[C]. Hong Kong, 2002: 940-949.

[4]Douglas Comer. The Ubiquitous B-Tree[J]. ACM Computing Surveys, 1979, 11(2): 121-137.

[5]C.Mohan and Frank Levine. ARIES/IM: an efficient and high concurrency index management method with write-ahead logging[J]. ACM SIGMOD Record, 1992, 21(2): 371-380.

[6]加西亚(Hector Garcia-Molina)，沃尔曼(Jeffrey D. Ullman)，威德姆(Jennifer D. Widom). 数据库系统实现[M]. 北京：机械工业出版社，2001. 1-238

[7]David Lomet and Betty Salzburg. Access method concurrency with recovery[J]. ACM SIGMOD Record, 1992, 21(2): 351-360.

[8]M.M.Astrahan, M.W.Blasgen, D.D.Chamberlin et al. System R: Relational Approach to Database Management[J]. ACM Transactions on Database Systems, 1976, 1(2): 97-137.

[9]Philip L. Lehman, S. Bing Yao: Efficient Locking for Concurrent Operations on B-Trees[J], ACM Transactions on Database Systems, 1981, 6(4): 650-670.

[10] Boral, H. et. al., "Prototyping Bubba: A Highly Parallel Database System," IEEE Knowledge and Data Engineering, Vol. 2, No. 1, March, 1990.

[11]ELLIS, C.S. Concurrent search and insertion in 2-3 trees. Tech. Rep. 78-05-01, Dep. Computer Science, Univ. Washington, Seattle, May 1978.

[12]KUNG, H.T., AND SONG, SW. A parallel garbage collection algorithm and ita correctness proof.In Proc. 18th Ann. Symp. Foundations of Computer Science, IEEE, Oct. 1977, pp. 120-131.

[13]C. Mohan, D. Haderle, B. Lindsay et al. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging[J]. ACM Transactions on Database Systems, 1992, 17(1):94-162.

[14]特勒尔森 (Troelsen, Andrew). C#与.NET 3.5 高级程序设计[M].北京：人民邮电出版社，2009.1-376

[15]Jim Gray, Andreas Reuter. Transaction Processing: Concepts and Techniques[M]. American: Morgan Kaufmann Publishers, 1993, 269-458.

[16]Mark L.McAuliffe, Michael J. Carey and Marvin H. Solomon. Towards Effective and Efficient Free Space Management[J], ACM SIGMOD Record, 1996, 25(2): 389-40.

[17]罗摩克里希纳(Raghu Ramakrishnan)，格尔基(Johannes Gehrke)．数据库管理系统原理与设计（第三版）[M]．北京： 清华大学出版社，2004. 207-295.

致　　谢

论文的最终的成稿得益于众人的关心和指导。

首先，要感谢的是本论文我的指导老师张坤龙张老师，从进入他实验室的那一刻起，我就为张老师严谨、认真、勤劳的治学精神所感动。从选题到开题，从中期检查到现在，每一步都伴随着张老师耐心的指导和和蔼的笑容，让我从对数据库管理系统的迷宫中一步步逐渐找到门路，看到光芒。

其次，要感谢另外和我一起做同一系统其他两部分的同学，杨亚军和韩智攀，虽然我们做的是同一系统的三个不同的部分，但正是因为是同一系统它们之间千丝万缕的联系让我们能够经常在一起讨论，使我自己的设计变得更加清晰明了。

再次，要感谢实验室的师兄们，感谢你们把自己从前做毕设的经验和教训毫不保留的教给了我，让我能够少走很多弯路，同时也感谢你们对本论文的修改的指正。

最后，要感谢在我做毕业设计期间所有关心我的同学、朋友和老师，是你们让我可以安心的把全部精力都投入到毕业设计上，使我最终可以顺利完成毕业设计的各项任务。

在此，衷心地感谢所有关心和帮助过我的人，谢谢你们！