事务处理器的设计与实现

学　　院 计算机科学与技术

专　　业 计算机科学与技术

年　　级 2006 级

姓　　名 谭龙飞

指导教师 张坤龙

2010 年 6 月 15 日

摘　要

随着信息量的增加，数据库需要处理的数据与日俱增，存储引擎对数据库影响也越来越大，各种各样的存储引擎层出不穷。TStore 是一个支持事务处理的数据存储引擎，包括事务处理器和存储管理器两部分。论文设计并实现了 TStore 的事务处理器。

事务处理器包括锁管理、日志管理和事务管理三大模块。锁管理模块实现了 Latch 和 Lock 两种锁，Latch 锁维护内部一致性，保护临界资源。Lock 锁采用两阶段加锁策略，管理并发。日志管理模块采用写日志优先协议，如实地记录系统的变更。事务管理模块实现了事务的创建、提交、回滚以及恢复。

论文对实现的事务处理器进行了测试。单元测试减少了系统 Bug，提高了系统的稳定性。集成测试测试了各模块的接口，确保了系统的各项功能都正常实现。

关键字：存储引擎；事务处理器；锁管理；日志管理；事务管理

ABSTRACT

With the increase of information, database needs to deal with more and more data, and storage engine has a growing impact on the database, and there are all kinds of storage engine. TStore is a storage engine which supports transaction processing. It mainly consists of transaction processor and storage manager. This paper designs and implements the TStore transaction processor.

Transaction processor includes lock manager, log manager and transaction manager. Lock manage provides latch and lock. Latch maintains internal consistency, protects critical resources. Lock uses Two-Phase Locking, and manages concurrency. Log manager uses Write Ahead Log Protocol, and records the changes of system. Transaction manager provides the functions of create transaction、submit transaction、rollback transaction and recovery transaction.

After achieving transaction processor, paper also gives a test of the system. Unit testing reduces the system bugs, and improves stability of the system. Integration testing tests the interfaces of each part, and ensures the functions of the system.

Key words： storage engine; transaction processor; lock manage; log manage; transaction manage

目　录

第一章　绪论 ……1

1.1 存储引擎 1

1.2 事务处理器 1

1.3 论文的研究内容 2

1.4 论文的组织结构 2

第二章　TStore 的体系结构……3

2.1 TStore 系统特色.....3

2.2 TStore 系统的模块结构.....3

2.3 TStore 系统的代码组织.....4

第三章　锁管理模块 ……6

3.1 Latch 锁 6

3.2 Lock 锁 11

第四章　日志管理模块 ……17

4.1 日志管理模块的设计 17

4.2 日志管理模块的实现 18

第五章　事务管理模块 ……23

5.1 事务管理模块的设计 23

5.2 事务管理模块的实现 24

第六章　系统测试 ……28

6.1 单元测试 28

6.2 集成测试 28
第七章 总结 30
参考文献 31
外文资料
中文译文
致 谢

第一章　绪论

1.1 存储引擎

数据库中的数据用各种不同的技术存储在文件(或者内存)中。这些技术中的每一种技术都使用不同的存储机制、索引技巧、锁定水平并且最终提供不同的功能和能力。通过选择不同的存储引擎，能够获得额外的速度或者功能，从而改善应用的整体性能。这些不同的技术以及配套的相关功能在数据库中被称作存储引擎[1]。

随着数据库管理系统要处理的数据越来越大，存储引擎性能好坏对整个系统产生着巨大的影响。基于这样的原因，产业界和学术界都花费大量的精力研究如何提高存储引擎的性能，经过多年的努力，产生了大量的存储引擎，如：Berkeley DB[2]、InnoDB、MERGE、ISAM、MyISAM[3]等。根据是否支持事务处理，可以把存储引擎分为两大类：一类是不支持事务处理的存储引擎，如 MyISAM；一类是支持事务处理的存储引擎，如 InnoDB。

1.2 事务处理器

事务是一个最小的工作单元，不论成功与否都作为一个整体进行工作。 事务处理的主要特征是，任务要么全部完成，要么都不完成。在写入一些记录时，要么写入所有记录，要么什么都不写入。如果在写入一个记录时出现了一个失败，那么在事务处理中已写入的其他数据就会回滚（rollback）。事务可能由很多单个任务构成。

事务应该具有 4 个属性：原子性、一致性、隔离性、持续性。这四个属性通常称为 ACID 特性。

原子性（Atomicity）：事务的所有操作是原子工作单元；对于其数据修改，要么全都执行，要么全都不执行。原子性消除了系统处理操作子集的可能性。

一致性（Consistency）：数据从一种正确状态转换到另一种正确状态。事务在完成时，必须使所有的数据都保持一致。在相关数据库中，所有规则都必须应用于事务的修改，以保持所有数据的完整性。当事务结束时，所有的内部数据结构都必须是正确的。

隔离性（Isolation）：由并发事务所作的修改必须与任何其他并发事务所作的修改隔离。查看数据时数据所处的状态，要么是事务修改它之前的状态，要么是事务修改它之后的状态。简单的理解就是，防止多个并发更新彼此干扰。事务在操作数据时与其他事务操作隔离。隔离性一般是通过加锁的机制来实现的。

持久性（Durability）：事务完成之后，它对于系统的影响是永久性的。已提交的更改即使在发生故障时也依然存在。

事务处理器的作用是管理事务，使之能满足事务的 ACID 特性及其它要求的

特性，目的是提高事务处理效率和保证其正确性。事务处理器包括锁管理模块、日志管理模块和事务处理模块三个功能模块。锁管理模块的作用是使用适当的锁，使事务满足原子性和隔离性；日志模块的作用是记录事务对系统的修改，当系统故障时能恢复事务对系统的影响，支持事务的持久性；而一致性则由事务管理模块完成，当事务提交时需要进行一致性检查，不能通过则放弃事务。

1.3 论文的研究内容

本论文主要设计并实现了事务处理器。该系统是整个 TStore 存储引擎的重要组成部分，对存储引擎的整体构建有着很重要的意义。论文的研究内容主要分为以下两部分：

（一）设计事务处理器。通过学习事务处理相关书籍、阅读相关论文以及分析现有事务处理开源代码，了解了事务处理器的一般体系结构以及事务处理方面的算法和技术等知识，根据需求把它们运用到本事务处理器。

（二）实现了事务处理器。根据事务处理器的设计，编程实现了事务处理器。本论文使用的高级编程语言是 C#。在本过程中，学习了 C#编程知识，并掌握了一些编程技巧和编程规范。

1.4 论文的组织结构

论文第一章讲述了存储引擎、事务处理器等背景知识，以及本论文的研究内容。第二章讨论了 TStore 存储引擎的特色、模块结构以及代码组织结构。第三、第四、第五章是论文的核心所在，分别讨论了锁管理模块、日志管理模块、事务管理模块的概念设计和代码实现。第六章讲述了系统测试，包括单元测试和集成测试。第七章总结了本论文的主要工作。

第二章　TStore 的体系结构

2.1 TStore 系统特色

TStore 是一个支持事务处理的存储引擎，用 C#编写的。相比较其他存储引擎，它有如下特点：

1. 支持事务处理。TStore 存储引擎有专门的事务处理模块，并且能够支持事务的 ACID 特性。
2. 支持多线程访问。TStore 存储引擎有专门的并发控制模块，能够多线程的读写数据。
3. 支持写日志优先。TStore 存储引擎采用写日志优先的策略，如此系统崩溃时可以保证事务恢复。
4. 使用基于锁的并发控制。TStore 存储引擎使用行级别的共享锁、更新锁、互斥锁管理并发。
5. 提供了多个隔离级。TStore 存储引擎支持读已提交（read committed）、可重复读（repeatable read）和串行化（serializable）三个事务隔离级。
6. 提供了 B 树索引。TStore 存储引擎提供 B 树索引。
7. 支持表结构存取。TStore 存储引擎支持表，但为了灵活性考虑，仅仅把表行当做一块数据，而表行内部的结构可以自由选择。
8. 使用了两种类别的锁：latch 和 lock。Latch 用来保证内部的一致性，lock 用来管理并发。
9. 用死锁检测。TStore 存储引擎用专门的死锁检测线程，出现死锁时采用放弃某个事务的方法来解决死锁。

当然，它也用不足之处，TStore 存储引擎不支持 SQL 语言，并且还不支持分布式事务。

2.2 TStore 系统的模块结构

TStore 存储引擎包含两大部分：存储管理器和事务处理器。存储管理器由工作组其他同学负责。事务处理器分为三大模块：锁管理模块、日志管理模块和事务管理模块。锁模块包括 latch 和 lock 两个子模块，latch 模块解决的内部数据结构的一致性，lock 模块解决的是并发问题。日志模块解决的是日志的结构以及如何写日志，如何保持日志等问题，本存储引擎在日志模块采用的是先写日志（Write Ahead Log）的策略。事务模块解决的是如何创建事务，如何提交事务，如何恢复事务等问题，本存储引擎在事务模块采用的是 ARIES[4]算法。

在图 2-1 中，TStore 存储引擎包括两部分：存储管理器和事务处理器。事务处理器分为三大模块：锁管理模块、日志管理模块和事务处理模块。锁管理包括 Latch 锁和 Lock 锁两个子模块。

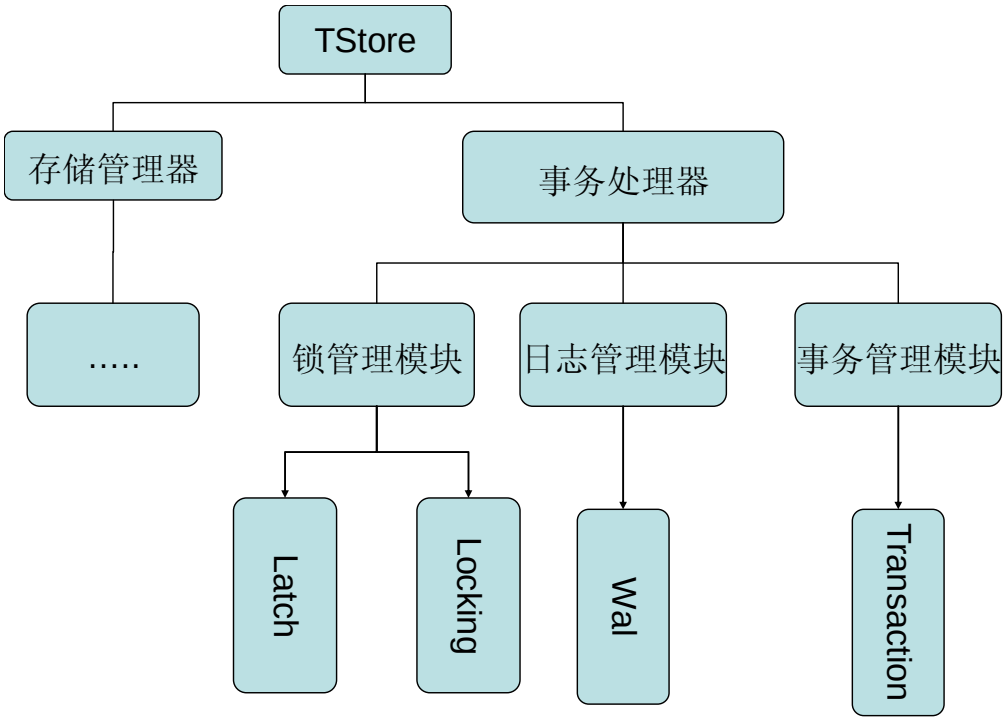

图 2-1　TStore 存储引擎的模块

2.3 TStore 系统的代码组织

本系统的代码是通过包和类组织起来的，整体来说分为两部分：

（一）TStore.Common：Common 部分封装的是一些简单、在 Rss 部分需要大量调用的类和函数，共包含 15 个文件夹，50 个 C#类，如：Event、Platform、Register、Logging、Mcat、Dumpable 等。

（二）TStore.Rss（Relational Storage System）：此部分是事务处理系统的主要实现部分，共包括 8 个文件，62 个 C#类。主要有四个模块：Latch、Locking、Wal 和 Transaction。

在图 2-2 中，TStore.Common 封装的是基本类，是其它类需要大量使用。TStore.Rss.Api.Latch 、 TStore.Rss.Api.Locking 、 TStore.Rss.Api.Wal 、 TStore.Rss.Api.TTransaction 封装的是接口， TStore.Rss.Impl.Latch 、 TStore.Rss.Impl.Locking 、 TStore.Rss.Impl.Wal 、 TStore.Rss.Impl.TransactionImpl 封装的是对应接口的实现。

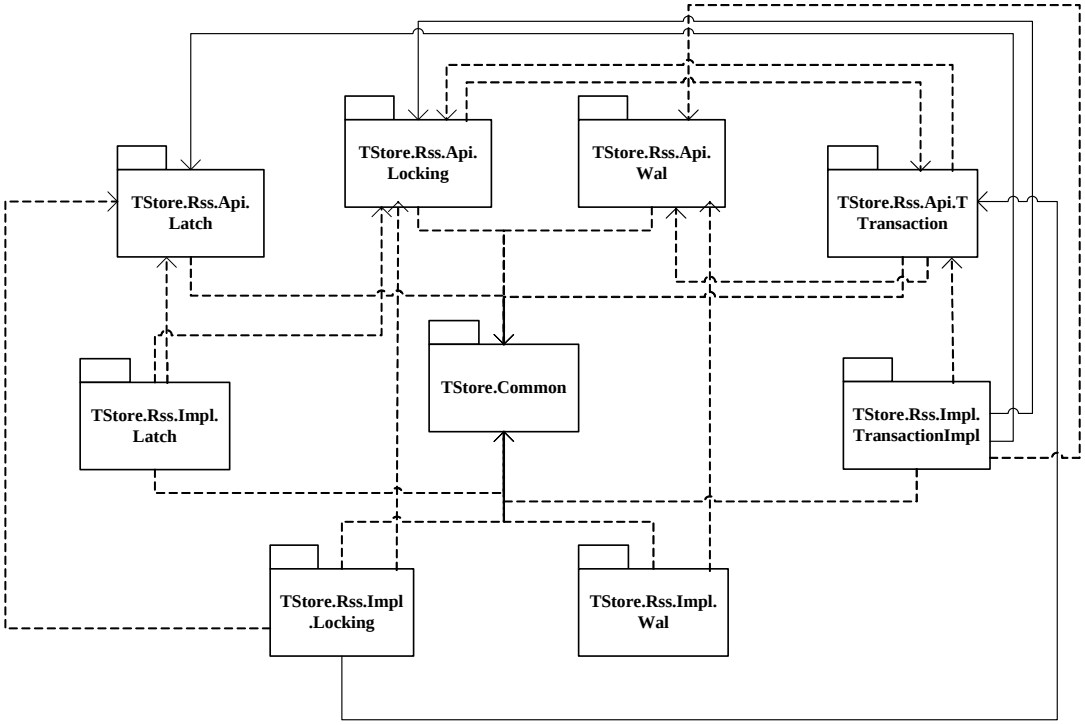

图 2-2　TStore 存储引擎代码组织图

第三章　锁管理模块

锁管理是事务处理器的重要组成部分。本系统中有两种类型的锁：latch 和 lock。Latch 称为轻量级的锁，是一种很有效的锁，用来维持内部一致性，保护临界资源。Latch 能够实现原子性，因为它保护的数据要么改变，要么不改变。lock 是重量级的锁，它的功能是管理各个线程，实现系统并发。因而 lock 能极大的提高系统的性能。

3.1 Latch 锁

3.1.1 latch 锁的设计

本系统中，共有三种模式的 latch 锁[5]：共享锁（share mode）、排它锁（exclusive mode）和更新锁（update mode）。它们的相容性如下：共享锁能与共享锁相容，排它锁不能与其它锁相容，更新锁能与共享锁相容，但共享锁不能与更新锁相容。除此之外，它们还有锁的升级降级关系：更新锁能升级为排它锁，相对应的排它锁能降级为更新锁，更新锁能降级为共享锁。

3.1.2 Latch 锁的实现

（一）类组织结构

本部分代码主要分为两个包：TStore.Rss.Api.Latch 和 TStore.Rss.Impl.Latch。TStore.Rss.Api.Latch 共有 ILatch、LatchException 和 LatchFactory 三个类，Ilatch 和 LatchFactory 定义的是接口，LatchException 实现的是异常处理，是基本类。TStore.Rss.Impl.Latch 是 latch 锁实现的主体部分，共有 LatchFactoryImpl、NewReadWriteUpdateLatch、ReadWriteLatch 和 ReadWriteUpdateLatch 四个类。NewReadWriteUpdateLatch、ReadWriteLatch 和 ReadWriteUpdateLatch 实现的是三个不同的 latch 锁，ReadWriteLatch 最简单，只实现了读写锁；ReadWriteUpdateLatch 较复杂，实现了读锁、写锁和更新锁；NewReadWriteUpdateLatch 最复杂，它还实现了意向锁，与 lock 锁联系起来了。

在图 3-1 中，ILatch 是一个接口，ReadWriteLatch、ReadWriteUpdateLatch、NewReadWriteUpdateLatch 分别实现了它。LatchFactoryImpl 实现了 LatchFactory，同时它还调用了 ReadWriteLatch 、ReadWriteUpdateLatch 、NewReadWriteUpdateLatch 的成员函数。LatchException 实现的是异常处理。

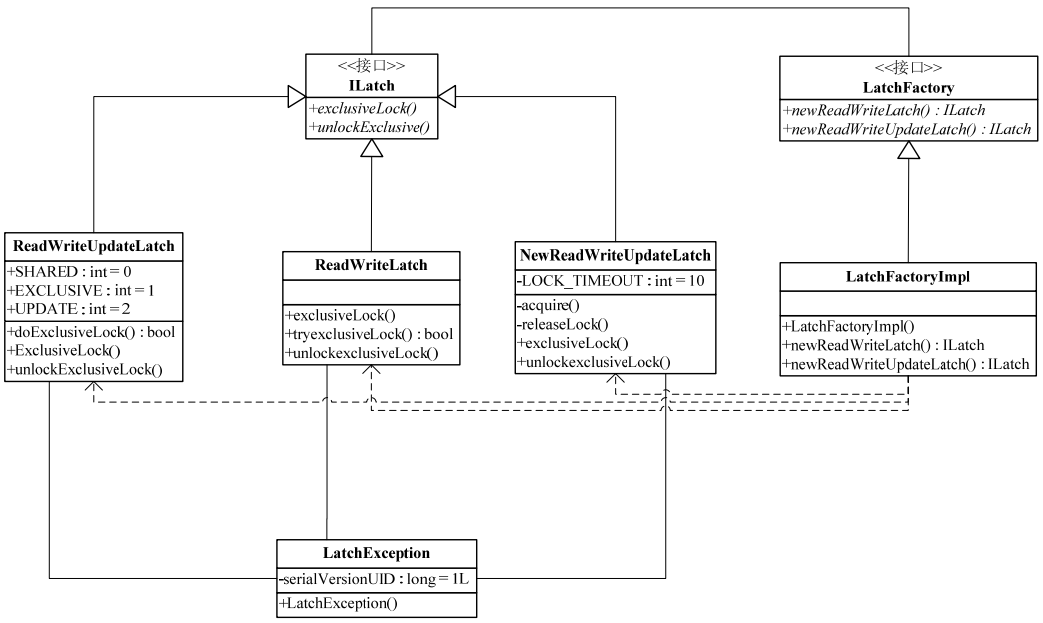

图 3-1　Latch 锁模块类图

（二）类成员函数简介

本模块的成员函数很多，无法一一介绍，选择一些关键的介绍如下：

1. tryExclusiveLock()：仅当请求锁在调用期间未被另一个线程保持时获的排它锁。加锁成功返回 true
2. exclusiveLock()：请求排它锁
3. exclusiveLockInterruptibly()：请求排它锁，允许其它线程中断
4. unlockExclusive()：释放排它锁
   (以上四个 API 共享锁和更新锁类似，不再介绍)
5. upgradeUpdateLock()：拥有更新锁时，升级为排它锁；否则出错，直接返回
6. downgradeExclusiveLock：拥有排它锁时，降级为更新锁；否则出错，直接返回
7. downgradeUpdateLock()：拥有更新锁时，降级为共享锁；否则出错，直接返回

（三）类成员函数实现示例

本模块的成员函数太多，实现无法一一表述，选择成员函数 exclusiveLock()的实现介绍。本部分共实现了三种 Latch 锁，exclusiveLock()也有三种实现。

1）ReadWriteLatch 类实现成员函数 exclusive()。在图 3-2 中，用 ReadWriteLockSim 类的 EnterWriteLock()方法实现 exclusive()方法。

```
public class ReadWriteLatch:ILatch
{
    private readonly ReaderWriterLockSlim rwlock=new ReaderWriterLockSlim();
    public void exclusiveLock() {
            rwlock.EnterWriteLock();
     }
}
```

图 3-2　ReadWriteLatch 类实现成员函数 exclusive()示意图

```
void exclusiveLock() {
    doExclusiveLock(true, false)
}
Boolean doExclusiveLock(Boolean wait，Boolean Interrupt)
{
    if(没线程拥有该对象的锁)
        //授予锁
        exclusiveCount = 1;
    if(本线程拥有该对象的排它锁)
        //授予锁
        exclusiveCount++;
    //除了以上两种情况，不能立即授予锁
    if(!wait)
        //加锁失败
        return false;
    加入等待队列;
    for( ; ; )
    {
        LockSupport.park();      //等待
        if(本线程拥有了排它锁)
            return true；//成功
        if(允许其它线程中断  &&  有线程中断)
            退出等待队列;
            return false；//失败
    } }
```

图 3-3　ReadWriteUpdateLatch 类实现成员函数 exclusive()示意图

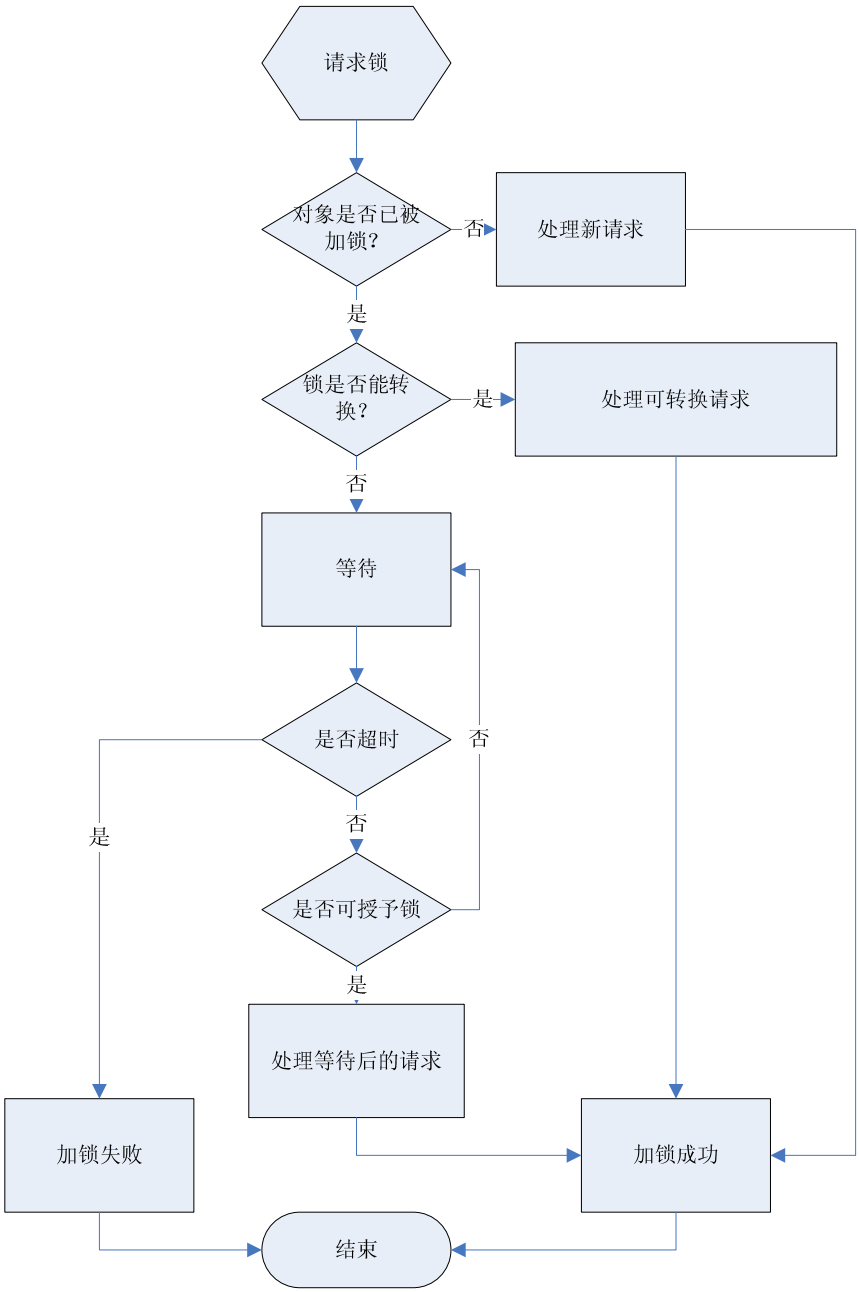

图 3-4　NewReadWriteUpdateLatch 类实现成员函数 exclusive()流程图

2）ReadWriteUpdateLatch 类实现成员函数 exclusive()。在图 3-3 中，ReadWriteUpdate 类中，exclusive()是根据概念设计来实现的，当请求的锁与现有锁相容时，授予锁；否则，不授予锁。doEeclusive()有两个参数，第一个表示是否允许等待，第二个表示是否允许其它程序中断。

3）NewReadWriteUpdateLatch 类实现成员函数 exclusive()。在图 3-4 中，NewReadWriteUpdateLatch 类请求排它锁有四种情况：一种情况是当前无锁，可以直接授予;，一种情况是当前有锁，但转换后也可以立即授予锁；第三种情况是不能立即授予锁，必须等待，等待后可以授予锁；最后一种情况是必须等待，但等待后超时了，不能授予锁。前三种情况加锁成功，最后一种情况加锁失败。

（四）使用代码示例

1)获得排它锁。在图 3-5 中，Properties 是可以设置一些参数，如事务日志的保存路径。在代码中我们先生成锁的工厂实例 latchFactory，使用工厂实例获得锁实例 latch，然后就可以请求排它锁。

```
Properties = new Properties();
platform = new PlatformImpl(properties);
latchFactory = new LatchFactoryImpl(platform, properties);
latch = latchFactory.newReadWriteUpdateLatch();
latch.exclusiveLock();
{
      //do something
}
latch.unlockExclusive();
```

图 3-5　获得排它锁示意图

2)升级锁，由更新锁升级为排它锁。在图 3-6 中，更新锁可以升级，升级后为排它锁，于是解锁时必须用排它锁解锁法。

```
latch.updateLock();
latch.upgradeUpdateLock();//升级为排它锁
latch.unlockExclusive();
```

图 3-6　更新锁升级示意图

3）更新锁与共享锁共处。在图 3-7 中，根据概念设计，更新锁可以与共享锁相容，但共享锁不能与更新锁相容。

```
latch.updateLock();
latch.sharedLock();
latch.sharedLock();
latch.unlockShared();
latch.unlockShared();
latch.unlockUpdate();
```

图 3-7　更新锁与共享锁共处示意图

3.2 Lock 锁

3.2.1 Lock 锁的设计

（一）加锁协议

本事务处理器使用的加锁协议是两阶段加锁[6]。根据此加锁协议，事务管理锁分为严格的两阶段。第一阶段是获得封锁阶段，也称为扩展阶段。在这阶段，事务可以申请获得任何数据项上的任何类型的锁，但是不能释放任何锁。第二阶段是释放封锁阶段，也称为收缩阶段。在这阶段，事务可以释放任何数据项上的任何类型的锁，但是不能再获得任何锁。两阶段封锁协议实现了事务集的串行化调度，但同时，一个事务的失败可能会引起一连串事务的回滚，并且两阶段封锁协议并不保证不会发生死锁。

（二）多隔离级

本事务处理系统提供了三个事务隔离级：读已提交（read committed）、可重复读（repeatable read）和串行化（serializable）。

1. 读已提交：允许不可重复读取，但不允许脏读取。这可以通过“瞬间共享读锁”和“排他写锁”实现。读取数据的事务允许其他事务继续访问该行数据，但是未提交的写事务将会禁止其他事务访问该行。
2. 可重复读：禁止不可重复读取和脏读取，但是有时可能出现幻影数据。这可以通过“共享读锁”和“排他写锁”实现。读取数据的事务将会禁止写事务（但允许读事务），写事务则禁止任何其他事务。
3. 串行化：提供严格的事务隔离。它要求事务序列化执行，事务只能一个接着一个地执行，但不能并发执行。如果仅仅通过“行级锁”是无法实现事务序列化的，必须通过其他机制保证新插入的数据不会被刚执行查询操作的事务访问到。

隔离级别越高，越能保证数据的完整性和一致性，但是对并发性能的影响也越大。所以应该根据具体的使用环境选择不同的隔离级，使功能和性能都得到最优化。

（三）lock 锁的模式、相容性及锁转换

在本系统中，共有七种模式的 lock 锁[7]：

1. 没有锁（None）：对锁没有授权，表示对资源没有请求；
2. 意向共享锁（Intent Share Lock，简称 IS 锁）：将要对较细粒度对象加共享锁；
3. 意向排它锁（Intent Exclusive Lock，简称 IX 锁）：将要对较细粒度对象加排它锁；

4. 共享锁（Share Lock，简称 S 锁）：对粗粒度对象加共享锁；
5. 共享意向排它锁（Share Intent Exclusive Lock，简称 SIX 锁）：对粗粒度对象加共享锁，对其下的细粒度对象加排它锁，本质上是 S 锁和 IX 锁的组合；
6. 更新锁（Update Lock，简称 U 锁）；对粗粒度对象加更新锁；
7. 排它锁（Exclusive Lock，简称 X 锁）；对粗粒度对象加排它锁。

在表 3-1 中，如果一个锁的请求类型和被授予另一个事务的类型是相容的（Y），则该锁立即获得。如果不相容（N），那么新申请的锁必须等待另一个事物释放该锁。

表 3-1　各模式 Lock 锁的相容矩阵图

Mode	NONE	IS	IX	S	SIX	U	X
NONE	Y	Y	Y	Y	Y	Y	Y
Intent Shared	Y	Y	Y	Y	Y	N	N
Intent Exclusive	Y	Y	Y	N	N	N	N
Shared	Y	Y	N	Y	N	N	N
Shared Intent Exclusive	Y	Y	N	N	N	N	N
Update	Y	N	N	Y	N	N	N
Exclusive	Y	N	N	N	N	N	N

表 3-2　锁转换矩阵图

Mode	NONE	IS	IX	S	SIX	U	X
NONE	NONE	IS	IX	S	SIX	U	X
Intent Shared	IS	IS	IX	S	SIX	U	X
Intent Exclusive	IX	IX	IX	SIX	SIX	X	X
Shared	S	S	SIX	S	SIX	U	X
Shared Intent Exclusive	SIX	SIX	SIX	SIX	SIX	SIX	X
Update	U	U	X	U	SIX	U	X
Exclusive	X	X	X	X	X	X	X

本系统还支持锁转换，表 3-2 是锁转换矩阵图。在转换矩阵图中，如果一个锁以一种类型被批准，又以另一个类型被请求，则新请求被看作以两种类型中最大的类型申请。

（四）死锁

死锁是指两个或两个以上的进程在执行过程中,因争夺资源而造成的一种互相等待的现象,若无外力作用,它们都将无法推进下去.此时称系统处于死锁状态或系统产生了死锁。本事务处理器使用的是基于锁的并发控制，因而也会出现死锁。处理死锁问题，本系统使用了一个简单的死锁检测程序[8]，简单介绍如下。

每个锁维持两个队列：授予列和等待列。授予列保存的是授予锁的事务，等待列保存的是等待锁的事务。对授予集合 G 中的每一个锁，以及非空等待队列 W，检测程序如下运行：对每个 g∈G 和 w∈W，如果 mode(g)（表示 g 请求锁的模式）与 mode(w)不相容，则在图中加上边<w,g>。此外，如果 w`在队列 w 中处于 w 之后，并且不相容，则在图中加上边<w`,w>。算法照此循环，选择新锁，直到等待队列中所有锁都被检查过。这样就完成了等待图的构建。如果发现等待图中有环，说明存在死锁，则从环中随机选择一个事务回滚，如此可解决死锁。

3.2.2 Lock 锁的实现

（一）类组织结构

本部分代码分为两个包：TStore.Rss.Api.Locking 和 TStore.Rss.Impl.Locking。TStore.Rss.Api.Locking 包共有十个类，主要是接口和异常类以及一些简单的类。如 LockManager 定义的就是获得某个对象上的锁、对锁进行降级、查找锁、释放锁等锁操作的接口；LockDeadlockException 定义的是死锁异常；LockModeClass 是一个简单类，定义的是五种不同的锁模式以及它们的相容矩阵和转换矩阵。TStore.Rss.Impl.Locking 中共有五个类，是 Lock 锁实现的主体部分，实现了锁的各种操作，同时还实现了死锁的检测。

在图 3-8 中，LockModeClass 定义了五种不同的锁模式以及它们的相容矩阵和转换矩阵，是各个类都需要使用的。LockManagerFactoryImpl 实现了接口 LockMgrFactory，是一个工厂类。LockManagerImpl 实现了接口 LockManager 和接口 LockHandle，是 Lock 锁实现的主体部分。

（二）类成员函数简介

本模块的成员函数很多，无法一一介绍，选择一些关键的介绍如下：

1. acquire(Object owner, Object lockable, LockMode mode, LockDuration duration, int timeout, LockInfo lockInfo)：请求锁。owner 是锁的拥有者，lockable 是加锁的对象，mode 是加锁的模式，duration 是拥有锁的时间长度，timeout 代表着是否等待锁，lockInfo 是加锁的其它信息
2. downgrade(Object owner, Object lockable, LockMode downgradeTo)：把 owner 在 lockable 上拥有的锁降级为 downgradeTo

3. release(Object owner, Object lockable, bool force)：释放 owner 在 lockable 上拥有的锁，force 代表是否强制释放
4. findLock(Object owner, Object lockable)：查找 owner 在 lockable 上拥有的锁
5. start()：开始 Lock 锁管理实例
6. shutdown()：关闭 Lock 锁管理实例
7. setDeadlockDetectorInterval(int seconds)：设置死锁检测线程运行的时间间隔

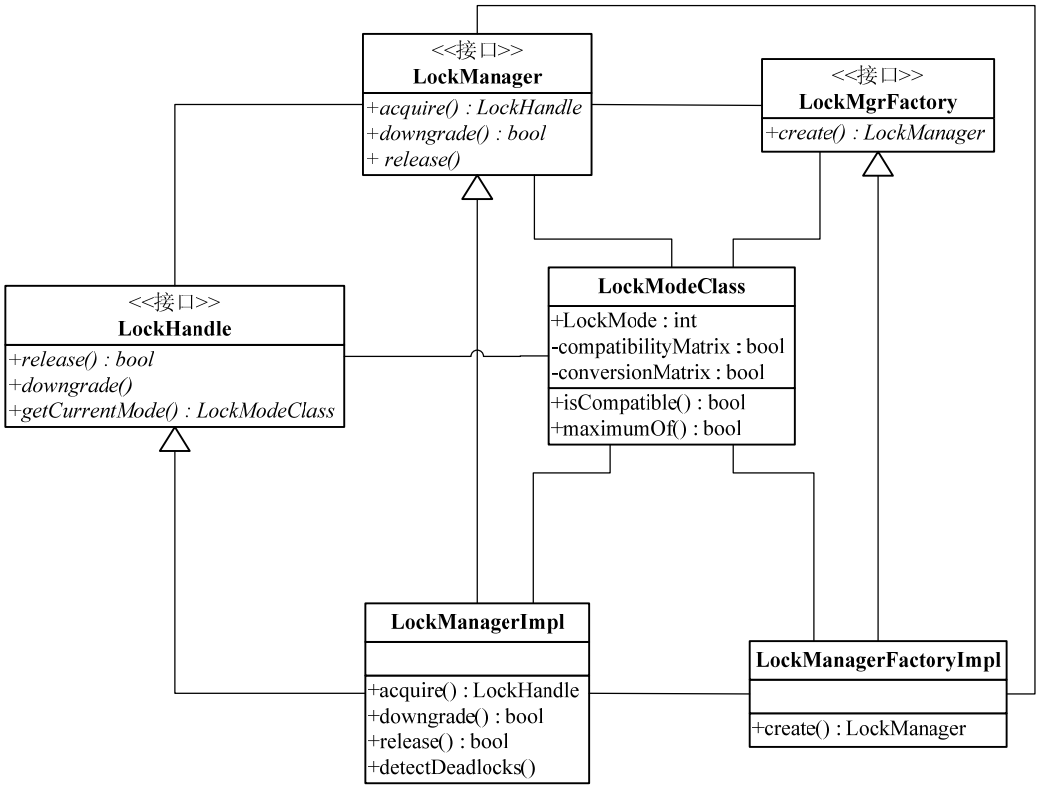

图 3-8　Lock 锁模块部分类类图

（三）类成员函数实现示例

本模块成员函数太多，它们的实现也无法一一表述，选择成员函数 acquire() 的实现介绍。在图 3-9 中，请求 Lock 锁有三种情况：第一种情况是对象未被加锁，此时可立即授予锁；第二种情况是对象已被加锁，但没有其它加锁请求在等待，此时检查请求锁的类型是否与当前锁的类型相容，相容则直接授予锁，不相容则需要等待；第三种情况是由其它加锁请求在等待，这种情况又分两类，一类是等待队列不能转换，则需要等待，一类是能转换，转换后检查是否能与当前锁相容，相容则授予锁，不相容则需要等待。等待超时，则加锁失败。

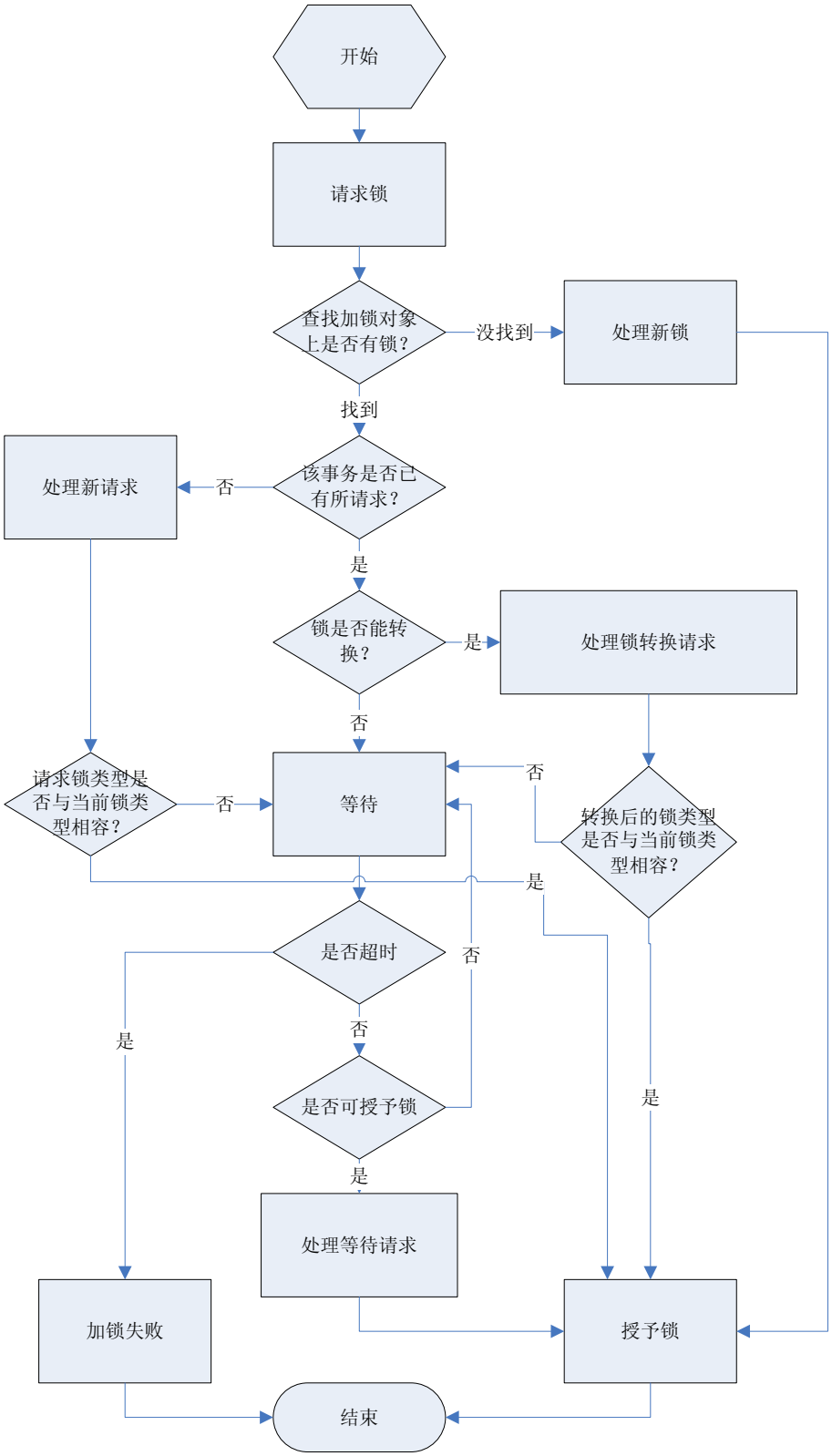

图 3-9　实现成员函数 acquire()流程图

（四）使用代码示例

1）建立 Lock 锁管理实例。在图 3-10 中，Properties 可以设置一些参数，如事务日志的存储路径。在代码中，先生成锁管理工厂 factory，再由锁管理工厂创建锁管理实例。

```
LockManagerFactoryImpl factory=new LockManagerFactoryImpl();
Properties properties = new Properties();
IPlatform platform = new PlatformImpl(properties);
LatchFactory latchFactory = new LatchFactoryImpl(platform, properties);
LockManager lockmgr=factory.  create(latchFactory, props);
```

图 3-10　建立 Lock 锁管理实例示意图

2)Lock 锁的获得和释放。在图 3-11 中，owner 是 Lock 锁的拥有者，lockname 是被加锁的对象，mode 是加锁的模式，duration 是拥有锁的时间长度，timeout 代表着是否等待锁，lockInfo 是加锁的其它信息。

```
LockManage lockmgr=  createLockManager();
Object owner=new Integer(1);
Object lockname=new Integer(10);
LockHandle handle = lockmgr.acquire(owner,lockname,LockMode.SHARED,
      LockDuration.MANUAL_DURATION, -1,null);//获得共享锁
//do something
handle.release(false);//释放锁
```

图 3-11　Lock 锁的获得和释放示意图

3)Lock 锁的降级。在图 3-12 中，通过 axquire()函数获得排它锁，然后使用 downgrade()函数，把排它锁降级为共享锁。

```
LockManage lockmgr=  createLockManager();
Object owner=new Integer(1);
Object lockname=new Integer(10);
LockHandle handle = lockmgr.acquire(owner,lockname,LockMode.EXCLUSIVE,
      LockDuration.MANUAL_DURATION, -1,null);//获得排它锁
handle.downgrade(LockMode.SHARED);//降级为共享锁
```

图 3-12　Lock 锁降级示意图

第四章　日志管理模块

日志是事务处理器所执行操作的历史集合，是事务处理器的重要组成部分。日志能保证事务的持久性，能够在系统崩溃时进行系统恢复。

从物理上来说，日志是保存在稳定存储上的连续的记录文件，它记录了系统的所有修改。每一条日志记录都有一个唯一的标识，称为日志顺序码（log sequence number，简称 LSN）。按照记录标识，就可以通过一个磁盘存取操作得到日志记录。LSN 必须以单调增加的方式来生成。

每条日志都具有下列一些字段：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。transId 字段存储的是生成日志记录的事务的标识。type 字段用来表明日志记录的类型。

4.1 日志管理模块的设计

（一）写优先日志协议

日志管理模块采取经典的写优先日志（Write Ahead Log，简称 WAL）协议[9]。写优先日志协议有如下要求：

1. 事务的所有操作都必须写入日志，并且在事务提交前日志必须保存到稳定的存储上；
2. 缓冲页面在修改前必须先写入日志，并且缓冲页面在写回磁盘前必须先把与此缓冲页面相关的日志记录保存到稳定的存储上；
3. 当缓冲页面正在修改，日志正在更新时，必须对此页面加排它锁（Exclusive Latch），以保证日志记录的顺序与产生日志的操作相一致。

于是，没被保存的日志记录可以被忽略，因为产生它的事务还未被提交。同时，日志的记录顺序反映了系统的变化顺序。如此，当系统出现故障时，可以使用日志对系统进行恢复。

（二）日志文件管理

除了日志记录外，系统还保存了日志控制信息（log control information），而且为了安全考虑，可以保存多份日志控制信息。

日志记录是不断增长的，但在实际上我们把日志分为了多个日志文件，同时我们还把多个日志文件组成一个日志组（Log Group）统一管理。每个日志组开始产生三个日志文件，当一个日志文件写满时产生下一个日志文件，保证同时有三个日志文件处于可写入状态。同时日志文件名为数字，并且单调递增，每次加一，如：1,2,3,4.....。如此定位一个日志记录首先要找到日志文件，再到日志文件中找到日志记录。于是 Lsn（日志顺序码）有两个参数，index 和 offset，index 为日志文件名，offset 为日志记录在日志文件中的位置。为了安全性，还可以设

置多个日志组，把日志同时保存到多个位置。

日志记录生成时，首先存储在日志缓冲区中，当用户或系统要求保存日志记录时，才把日志记录保存到稳定存储上。保存日志时，首先要找到要写入的日志文件，当发现当前的日志文件已满时，移到下一个日志文件，并生成一个新日志文件，保证日志组同时有三个日志文件。然后找到日志文件的写入点，就可以写入日志记录了。

4.2 日志管理模块的实现

（一）类组织结构

本部分代码分为两个包：TStore.Rss.Api.Wal 和 TStore.Rss.Impl.Wal。TStore.Rss.Api.Wal 部分共育六个类，定义的是接口和异常类，LogException 是日志异常类，Lsn 定义的是日志顺序码（Log Sequence Number）的结构，LogRecord 接口定义的是对日志记录的操作，LogReader 接口定义的是对日志记录的读取操作，LogManager 接口定义是对日志的管理，如插入日志、保存日志、读入日志等，LogFactory 接口是生成 LogManger 实例的工厂。TStore.Rss.Impl.Wal 共有两个类，LogManagerImpl 和 LogFactoryImpl，是日志管理实现的主体部分，实现了日志的各种操作。

在图 4-1 中，Lsn 定义的是日志顺序码的结构和操作。LogFactoryImpl 实现了接口 LogFatory，是日志管理工厂类。LogManagerImpl 实现了接口 LogManager、LogReader 和 LogRecord，是实现日志管理的主体部分。

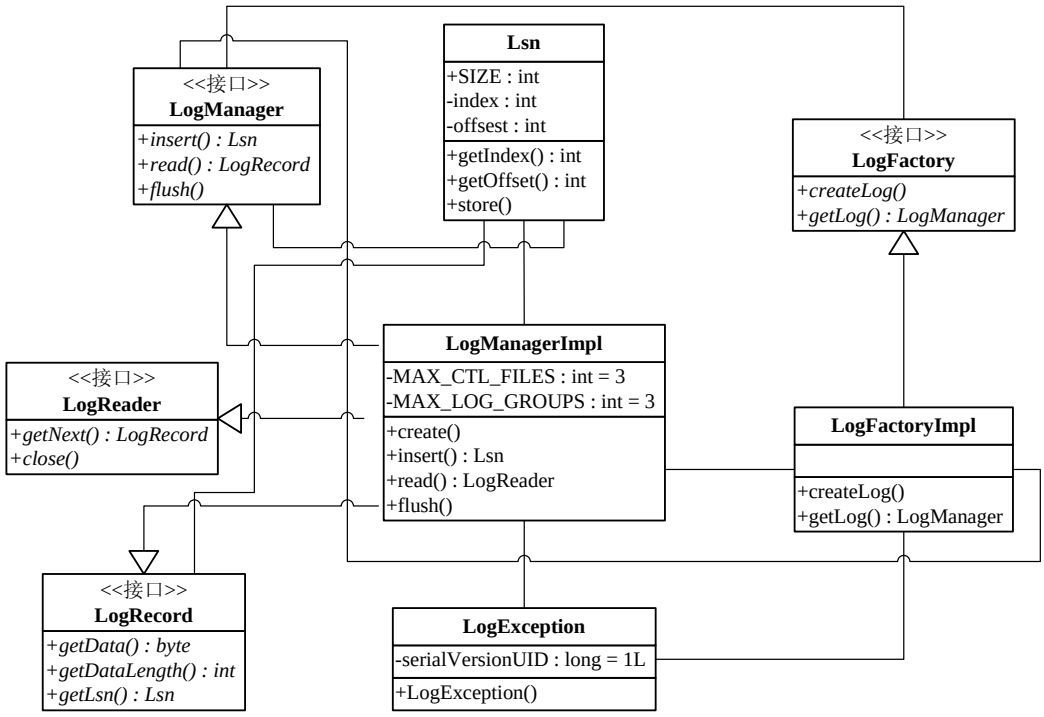

图 4-1　日志管理模块类图

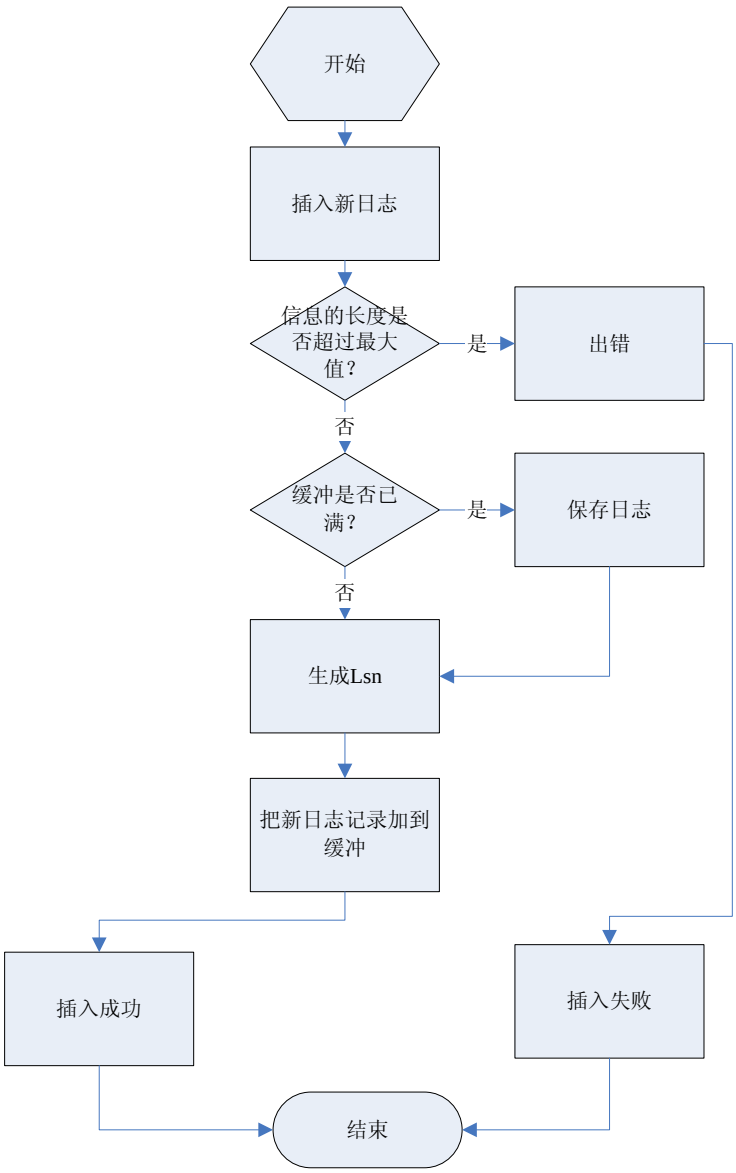

图 4-2　插入新日志的流程图

（二）类成员函数简介

本模块的成员函数很多，无法一一介绍，选择一些关键的介绍如下：

1. start()：开始日志管理实例
2. insert()：插入新的日志记录
3. getMaxLsn()：返回最大的 Lsn，也就是最后的那个 Lsn
4. getDurableLsn()：返回保存到稳定存储的日志记录的最大的 Lsn
5. getCheckpointLsn()：返回一个检查点的 Lsn
6. setCheckpointLsn()：生成一个检查点的日志记录
7. read()：读特定 Lsn 的日志记录
8. flush()：保存日志记录

9. getForwardScanningReader()：获得一个向前移动的日志游标[10]
10. getBackwardScanningReader()：获得一个向后移动的日志游标
11. shutdown()：关闭日志管理实例，释放分配的资源

（三）类成员函数实现示例

本模块类的成员函数太多，它们的实现也无法一一表述，选择成员函数 insert()的实现介绍。在图 4-2 中，插入新日志，首先检查信息的长度是否超过最大值，超过则出错，插入日志失败。不超过则再检查日志缓冲是否已满，满了则把缓冲中的日志记录保存到稳定存储。接着生成日志的 Lsn，最后把日志记录保存到日志缓冲中，日志插入成功。

（四）使用代码示例

1）创建日志实例。在图 4-3，properties 用来设置一些参数，如日志文件名。在代码中首先生成日志工厂，再由日志工厂创建日志实例。

2）获得已有日志实例。在图 4-4 中，properties 用来设置一些参数，如日志文件名。在代码中首先生成日志工厂，再用日志工厂打开已有的日志实例。获得日志实例后一定记得要关闭日志实例。

3）插入新的日志记录。在图 4-5 中，首先通过日志工厂获得日志实例，然后使用 insert()函数插入日志，最后关闭日志实例。

4）保存日志记录到稳定存储。在图 4-6 中，保存日志记录到稳定存储只要使用 flush()函数就行，没有参数时保存所有日志记录，有参数时只保存与 lsn 对应的日志记录。

```
LogFactory factory = new LogFactoryImpl();
Properties properties = new Properties();
properties.setProperty("log.ctl.1", "ctl.a");
properties.setProperty("log.ctl.2", "ctl.b");
properties.setProperty("log.groups.1.path", ".");
properties.setProperty("log.archive.path", ".");
properties.setProperty("log.group.files", "3");
properties.setProperty("log.file.size", "16384");
properties.setProperty("log.buffer.size", "16384");
properties.setProperty("log.buffer.limit", "4");
properties.setProperty("log.flush.interval", "30");
factory.createLog(properties);
```

图 4-3　创建日志实例示意图

```
LogFactory factory = new LogFactoryImpl();
Properties properties = new Properties();
properties.setProperty("log.ctl.1", "ctl.a");
properties.setProperty("log.ctl.2", "ctl.b");
properties.setProperty("log.groups.1.path", ".");
properties.setProperty("log.archive.path", ".");
properties.setProperty("log.group.files", "3");
properties.setProperty("log.file.size", "16384");
properties.setProperty("log.buffer.size", "16384");
properties.setProperty("log.buffer.limit", "4");
properties.setProperty("log.flush.interval", "30");
LogMgr log = factory.openLog(properties);
try {
   // do some work
} finally {
if (log != null)
   log.close();
}
```

图 4-4　获得日志实例示意图

```
LogMgr log = factory.openLog(null);
try {
   String s = "hello world!";
   byte[] b = s.getBytes();
   Lsn lsn = log.insert(b, b.length);
} finally {
if (log != null)
   log.close();
```

图 4-5　插入新的日志记录示意图

```
String s = "hello world!";
byte[] b = s.getBytes();
Lsn lsn = log.insert(b, b.length);
log.flush(lsn);
```

图 4-6　保存日志记录到稳定存储示意图

5）读取日志记录。在图 4-7 中，通过.getForwardScanningReader()函数获得一个向前移动的日志游标。然后循环使用 getNext()移动到每一条日志记录。最后一定要关闭日志游标。

```
LogReader reader = log.getForwardScanningReader(null);
try {
    for (;;) {
        LogRecord rec = reader.getNext();
        if (rec == null) {
            break;
        }
        printRecord(rec);
    }
}
finally {
    if (reader != null)
        reader.close();
    }
}
```

图 4-7　读取日志记录示意图

6）记录检查点日志。在图 4-8 中，首先生成一条检查点日志记录，然后再插入新日志记录，最后保存新日志记录到稳定存储。

```
CheckpointRecord checkpointRec = new CheckpointRecord();
Lsn checkpointLsn = log.insert(checkpointRec.getData(),
checkpointRec.getLength());
logmgr.setCheckpointLsn(checkpointLsn);
logmgr.flush(checkpointLsn);
```

图 4-8　记录检查点日志示意图

第五章　事务管理模块

事务管理模块的作用是管理事务，需要能创建事务，提交事务或放弃事务，并且还能创建检查点，同时还需要提供恢复机制，在系统出错时恢复系统。

事务管理模块支持事务的 ACID 特性。通过反做中止的事务、重做已提交的事务以及与事务管理模块协同提交等策略，提供了要么都做要么都不做的特性，支持了 A（原子性）。利用在提交时，任何不能通过资源管理器一致性检测的事务都将被中止这一策略，实现了 C（一致性）。通过与锁管理模块相结合，对事务的某些对象适当的加锁，实现了事务的 I（隔离性）。在提交事务时，要把已提交事务的日志记录强制写入永久性存储器，以便在重启时重做最近已提交的工作，实现了 D（持久性）。

5.1 事务管理模块的设计

（一）ARIES

本模块是根据 ARIES[4]设计的。在本系统中，ARIES 算法有以下要求：

1. 写日志的策略必须是写日志优先
2. 事务修改的最小单元是页面（page）[11]
3. 每个页面要有一个页面顺序码（PageLSN），用来定位最后一个修改页面的日志记录
4. 在写检查点期间，事务管理器并不是保存所有的页面，仅仅保存被修改过的页面，也就是所谓的脏页
5. 在系统重启阶段，事务管理器负责通知缓冲管理需要恢复的赃页，缓冲管理负责恢复赃页
6. 日志管理能够顺序读取日志记录，或者直接通过 LSN 定位日志记录。同时，日志管理还要能产生检查点日志
7. 锁管理模块提供加锁和解锁接口，同时锁管理还要能强制释放锁，以解决死锁

（二）事务管理模块与锁管理模块的交互

锁管理模块是一个非常重要的模块，它能支持事务的隔离性，当多个用户使用系统时，事务必须要使用锁。下面是事务管理模块与锁管理模块的交互活动[12]：

1. 在事务开始之后，事务可以在任何时候通过锁管理模块提供的接口获得锁
2. 当使用可重复读隔离级时，上一条记录的更新锁可以降级为共享锁
3. 当事务回滚到保存点时，保存点之后获得的锁要全部释放
4. 当事务完成（提交或放弃）时，事务拥有的锁要全部释放

（三）事务的日志记录等级

事务管理模块与日志管理模块关系密切，在事务中会产生大量的日志。在事务管理模块中，有四个产生日志的操作：

1. 可重做操作(Redoable)日志：所用影响引擎页面的操作都必须产生此种日志，以备系统恢复
2. 事务无关操作(NonTransactionRelatedOperation)日志：这类日志记录的操作不会影响引擎页面，具体来是对各种容器的操作
3. 发布提交动作（PostCommitAction）日志：这类操作是事务无关操作的子类。这类动作是事务提交后的操作，一般是对容器的释放
4. 容器删除操作（ContainerDeleteOperation）日志：容器删除有两种，一是释放建好的容器，一是放弃建容器的操作

5.2 事务管理模块的实现

（一）类组织结构

本部分代码分为两部分：TStore.Rss.Api.TTransaction 和 TStore.Rss.Impl.TransactionImpl。TStore.Rss.Api.TTransaction 共有二十八个类，主要定义的是接口，如 Transaction 定义的是事务接口，主要操作有提交事务、获得锁、插入日志等。TStore.Rss.Impl.TransactionImpl 共有四个类，实现了 TStore.Rss.Api.TTransaction 包定义的接口，是事务管理实现的主体部分。

在图 5-1 中，TransactionalModeRegistryImpl 是事务注册类。TransactionManagerImpl 实现了 TransactionalMode、Transaction、TransactionManager、Savapoint、Redoable 等接口，是实现事务管理的主体部分。

（二）类成员函数简介

本模块的成员函数很多，无法一一介绍，选择一些关键的介绍如下：

1. getLastLsn()：获得本事务产生的最后一个 Lsn
2. logInsert()：插入本事务产生的日志记录
3. acquireLock(Lockable lockable, LockMode mode,LockDuration duration)：获得 lockable 上的锁，mode 为锁的模式，duration 为锁的持有期间长短
4. releaseLock(Lockable lockable)：释放 lockable 上的锁
5. createSavepoint()：创建事务保存点
6. commit()：提交事务
7. rollback(Savepoint sp)：回滚事务到保存点
8. abort()：放弃事务
9. begin(IsolationMode isolationMode)：开始一个新的事物

（三）类成员函数实现示例

本模块类的成员函数太多，它们的实现无法一一表述，选择成员函数 commit()

的实现介绍。在图 5-2 中，提交事务有四个阶段。首先是准备工作，做的是插入准备日志、记录事务此种状态并把事务状态写为准备提交。然后是插入事务提交日志记录。第三阶段是完成本事务的所有操作。最后阶段是释放资源。

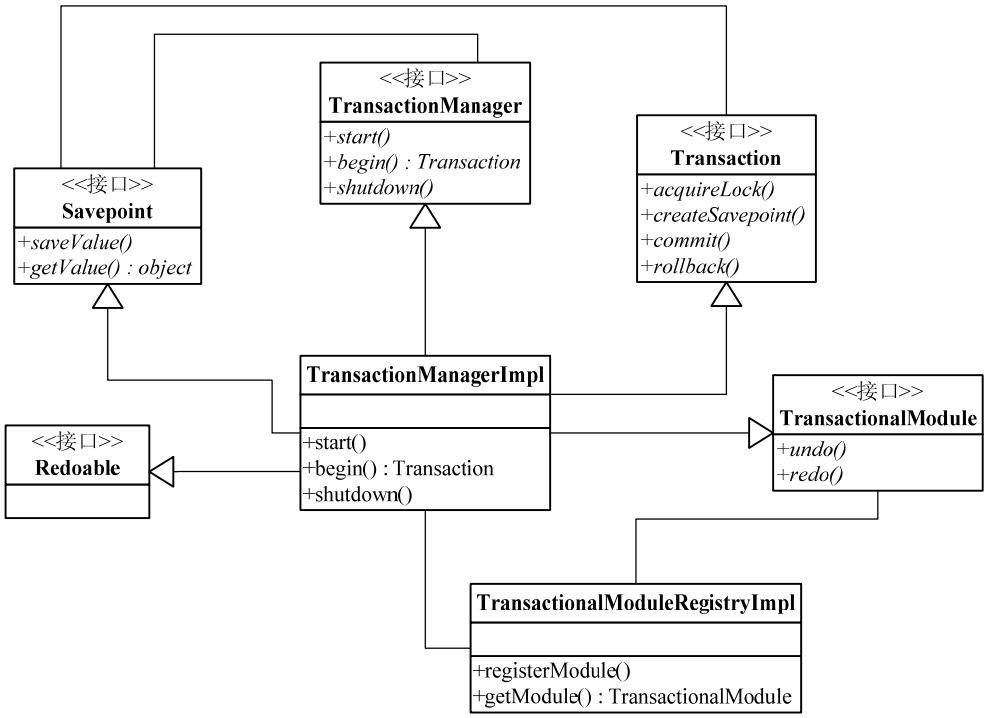

图 5-1　事务管理模块部分类类图

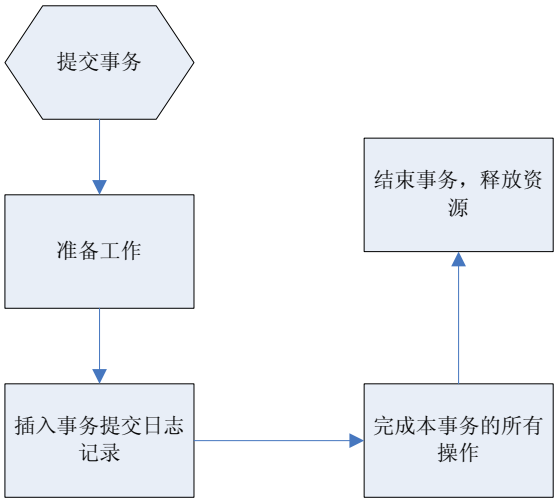

图 5-2　提交事务的流程图

（四）使用代码示例

1）创建、关闭事务管理实例。在图 5-3 中，事务管理实例的创建和关闭必须是成对的，当创建事务管理实例后，一定要调用关闭动作。

```
TransactionManagerImpl trxmgr = new TransactionManagerImpl();
try {
    trxmgr.start();
    //do something
} finally {
    trxmgr.shutdown();
}
```

图 5-3　创建、关闭事务管理实例示意图

2）开始新事务、提交或放弃事务。在图 5-4 中，首先，调用 begin()函数开始新事务，然后做一些事，做成功了则调用 commit()函数提交事务，失败了则调用 abort()函数放弃事务。

```
Transaction trx = trxmgr.begin(IsolationMode.SERIALIZABLE);
success=false;
try   {
    //do something
    success=true;
}
finally {
    if (success)
        trx.commit();
    else
        trx.abort();
}
```

图 5-4　开始新事务、提交或放弃事务示意图

```
Transaction trx = trxmgr.begin();
Savepoint sp = trx.createSavepoint(false);
//do something
trx.rollback(sp);
trx.commit();
```

图 5-5　创建事务保存点以及回滚到事务保存点示意图

3）创建事务保存点以及回滚到事务保存点。在图 5-5 中，创建事务保存点时通过调用函数 createSavepoint()来实现的，当要回滚到保存点时，调用 rollback()函数。

4）事务管理模块与锁管理模块的结合。在图 5-6 中，事务请求锁是通过调用函数 acquire()函数来实现的，acquire()函数共有三个参数，第一个参数是被锁的对象，第二个参数是锁的模式，第三个参数是锁的保持时间。事务对锁的降级是调用 downgradeLock()来实现的。

```
Transaction trx = trxmgr.begin();
trx.acquireLock(new ObjectLock(1,15), LockMode.SHARED,LockDuration.
    MANUAL_DURATION);
trx.acquireLock(new ObjectLock(1,15), LockMode.UPDATE,LockDuration.
    MANUAL_DURATION);
trx.downgradeLock(new ObjectLock(1, 15),LockMode.SHARED);
trx.commit();
```

图 5-6　事务请求锁示意图

第六章　系统测试

6.1 单元测试

单元测试是在软件开发过程中要进行的最低级别的测试活动，在单元测试活动中，软件的独立单元将在与程序的其他部分相隔离的情况下进行测试。单元测试是一种非常有效的测试活动，简单但能发现较多的错误，能为后期的调试节约很多工作。本系统的单元测试使用 C# 提供的类库 Microsoft.VisualStudio.TestTools.UnitTesting 自动生成。图 6-1 是一小段单元测试代码，其他的测试类似。

```
[TestMethod()]
public void getNextTest()
{
    int range = 4; // TODO: 初始化为适当的值
    WrappingSequencer target = new WrappingSequencer(range); // TODO: 初始化为适当的值
    int expected = 0; // TODO: 初始化为适当的值
    int actual;
    for (int i=0; i < 12; i++,expected++)
    {
        expected = expected % 4;
        actual = target.getNext();
        Assert.AreEqual(expected, actual);
    }
}
```

图 6-1　getNext()函数单元测试示意图

6.2 集成测试

集成测试，也叫组装测试或联合测试。在单元测试的基础上，将所有模块按照设计要求组装成为子系统或系统，进行集成测试。实践表明，一些模块虽然能够单独地工作，但并不能保证连接起来也能正常的工作。程序在某些局部反映不出来的问题，在全局上很可能暴露出来，影响功能的实现。

本系统的集成测试是按照功能进行组装的，如事务模块的集成测试，是测试能否生成新事务、能否对事务加锁、能否插入事务日志、能否提交事务等。如图 6-2，测试了事务加锁，以及事务锁的升级，同时还测试了事务保存点。

```
trx = trxmgr.begin(IsolationMode.SERIALIZABLE);
trx.acquireLock(lock1,LockMode.SHARED,LockDuration.COMMIT_DURATION);
Savepoint savepoint = trx.createSavepoint(false);
assertEquals(trx.hasLock(lock1), LockMode.SHARED);
trx.acquireLock(lock1,LockMode.UPDATE,LockDuration.COMMIT_DURATION);
assertEquals(trx.hasLock(lock1), LockMode.UPDATE);
trx.rollback(savepoint);
assertEquals(LockMode.UPDATE, trx.hasLock(lock1));
```

图 6-2　事务使用锁的集成测试示意图

第七章 总结

随着信息量的增加，数据库需要处理的数据与日俱增，存储引擎对数据库影响也越来越大，各种各样的存储引擎层出不穷。本论文设计并实现了存储引擎中的事务处理器。

首先，深入的了解事务处理器，对事务处理器进行了概念设计，把事务管理系统划分为三个模块：锁管理模块、日志管理模块和事务管理模块。再根据需求，把一些经典的算法运用到各个模块，如在锁管理模块中使用两阶段加锁策略，在日志管理模块中使用写日志优先协议，在事务管理模块中使用 ARIES[4]算法。

然后，根据设计实现了事务处理器。本系统全部采用 C#语言[13][14][15]编写。根据类设计的基本原则[16]，系统设计了大量的类。这样使得系统组织简单，条理清晰，并且还提高了代码的可读性和重用性。本系统共有 112 个类。

最后，还进行了系统测试。系统测试包括两类测试：单元测试和集成测试。单元测试是随着实现事务处理器进行的，由编译器自动生成。集成测试是按功能进行的，测试系统能否完成特定的功能。

参考文献

[1]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.

[2] Oracle Technology Network．Oracle Berkeley DB 11g [EB/OL]．http://www.oracle.com/technology/products/berkeley-db/index.html．2010

[3]Mysql Developer Zone．The MySQL 5.0 Archive Storage Engine[EB/OL]．http://dev.mysql.com/tech-resources/articles/storage-engine.html．2005．

[4]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.

[5]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.

[6]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.

[7]C.Mohan and D.Haderle. Algorithms for Flexible Space Management in Transaction Systems Supporting Fine-Granularity Locking[A]. In Proceedings of the International Conference on Extending Database Technology[C]. Cambridge, United Kingdom: 1994. 131-144.

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

[9]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.

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

[11]Mark L.McAuliffe, Michael J. Carey and Marvin H. Solomon. Towards Effective and Efficient Free Space Management[J]. ACM SIGMOD Record, 1996, 25(2): 389-400.

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

[13]内格尔 (Nagel，C) 等．C#高级编程(第 6 版)[M]．北京：清华大学出版社，2008．1－358．

[14]李泽，陈彬，唐俊翟等．C#函数适用手册[M]．北京：冶金工业出版社，2005．107－122．

[15]斯梅切尔．C#和.NET 2.0 实战[M]．北京：人民邮电出版社，2008．94－107．

[16]杨宏伟等．C#程序员开发手册[M]．北京：科学出版社，2005．113—244．

致　谢

本文的写作过程中得到了许多老师、同学的指导和帮助。

首先我要衷心的感谢我的导师张坤龙。本文的选题，研究工作以及论文的写作都是在张老师的悉心指导和帮助下才得已完成的。论文的顺利完成，得益于张老师对我的严格训练和谆谆教诲。张老师丰富的专业知识，独到的见解，严谨的治学态度，创新的学术思想，追求完美和一丝不苟的敬业精神深深的激起了我对知识的渴求，不但对我的学业帮助巨大，更将使我终身受益。

感谢李罡和胡静老师，他们为毕业生所写的《本科毕设论文写作注意事项 2010 版》，这给了我很大的帮助。

感谢郝玉琨同学，他与我一同设计和实现存储引擎，经常在探讨中给予我启发，并毫无保留的分享他收集到的资料。

还要感谢实验室的师兄们，在我的研究工作和论文写作过程中，他们为我提供材料，并且当我向他们请教时，他们总是耐心的为我讲解。

再次对所有帮助过的人表示感谢！