存储管理器的设计与实现

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 郝玉琨

指导教师 张坤龙

2010 年 6 月 15 日

摘　要

TStore 是一个支持事务处理的数据存储引擎，它主要由事务处理器和存储管理器组成。论文设计并实现了 TStore 的存储管理器。

存储管理器包含空间管理、缓冲管理、存取方法管理三个模块。空间管理模块实现了页管理、槽页管理、空闲空间管理和存储单元管理，为其他模块提供文件读写支持。缓冲管理模块实现了改进的 LRU 缓冲调度算法，能够提高整个系统的读写性能。存储管理模块实现了 B-link 树索引数据结构，能够快速响应多用户的并发访问，提高系统的整体性能。

为了管理系统中大量的对象实例，存储管理器还实现了基于“键-值”映射的对象库，可以支持系统中各种对象的快速提取和持久化。

论文对存储管理器的实现进行了测试。黑盒测试的结果表明，系统各项功能都能正确执行。

关键字：存储引擎；存储管理器；空间管理；缓冲管理；存取方法管理

ABSTRACT

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 storage manager.

Storage manager is made up of space manager, buffer manager and accessing methods manager. Space manager provides the functions of page managing, slotted page managing, free space managing and storage container managing, and it provides interface for data reading and writing. Buffer manager implements an improved LRU algorithm, and it will improve the performance of data reading and writing. Access method manager implements B-link tree index data structure, and it gives fast response to multi-user accessing, improves the performance of the whole system.

Storage Manager also implements an object registry based "Key-Value" schema, which is efficient in managing a large number of object instances in the system. The object registry supports fast retrieving and storing of an object.

This paper finally gives a test of the storage manager. Result of Black-box Test shows that the functions mentioned above performing correctly in accordance with the design principles.

Key words： storage engine; storage manager; space manage; buffer manage; access method manage

目 录

第一章　绪论

随着互联网的发展，网络上积累的数据呈爆炸式的增长，数据处理压力不断增大的同时,也为专业人员进一步研究存储引擎提供了机遇。如何从存储管理的工作原理出发，研究出性能更稳定，并发支持度更高的存储引擎，成为开发人员必须要思考的问题。TStore 存储引擎就是这样一个支持事务处理的存储引擎，本篇论文介绍的是TStore存储引擎的一个重要组成部分：存储管理器。

1.1 存储引擎与存储管理器

存储引擎[1]，也称数据库引擎，是数据库管理系统[2]中的一个软件模块，负责插入、查找、更新、删除数据。

存储引擎已经被研究了多年，大致有如下两种：一种作为独立的产品，比如Berkeley DB[3]；一种作为数据库管理系统中的一个组成部分。后一种又可被细分为两类，一类如 SQL Server、Oracle 的存储引擎组件，存储引擎作为一个固定的成员，内嵌在数据库管理系统中，另一类如Mysql 的 MyISAM、InnoDB、MERGE等[4]，用户可以根据不同的使用环境进行选择。

与国际上众多知名产品相比，国内的发展情况相对滞后，比较著名的数据库管理系统有中国人民大学的金仓数据库，华中科技大学的达梦数据库，但是存储引擎还没有作为一个独立的系统来被重视和研究。

存储管理器，是存储引擎的一个重要的组成部分。存储管理器所要实现的功能包括空间管理、缓冲管理和存取方法管理[4]。

存储管理器为存储引擎的其他部分提供了数据存储相关的支持，并且保证了良好的数据存储效率。存储引擎内部涉及存储相关的操作，都需要利用存储管理器提供的接口来完成。

1.2 TStore 存储引擎

TStore 存储引擎是基于传统的存储引擎的概念，结合事务处理相关理论，实现的一个支持事务处理，允许多线程并发访问的新型存储引擎。

TStore 存储引擎有如下特色：

1、 事务性：TStore 存储引擎支持 ACID 事务处理特性。

2、 多线程：TStore 存储引擎是多线程的，支持多个线程同时读写。

3、 先写日志：TStore 存储引擎实现了 WAL 恢复管理策略。

4、 基于加锁的并发控制：TStore 存储引擎实现了共享锁、更新锁、互斥锁来管理并发访问。

5、 B-link 树索引：TStore 存储引擎实现了 B-link 树索引结构，它支持并发的读取、插入和删除数据。

6、 死锁检测：TStore 存储引擎支持死锁预防，设置了周期性检查锁表的后台线程，负责中断死锁事务。

TStore 存储引擎内部包含两个大的功能单元：事务处理和存储管理。其中，事务处理单元的功能包括事务管理、锁管理和日志管理；存储管理单元的功能包括空间管理、缓冲管理和存取方法管理。TStore 存储引擎的体系结构如图 1-1 所示。

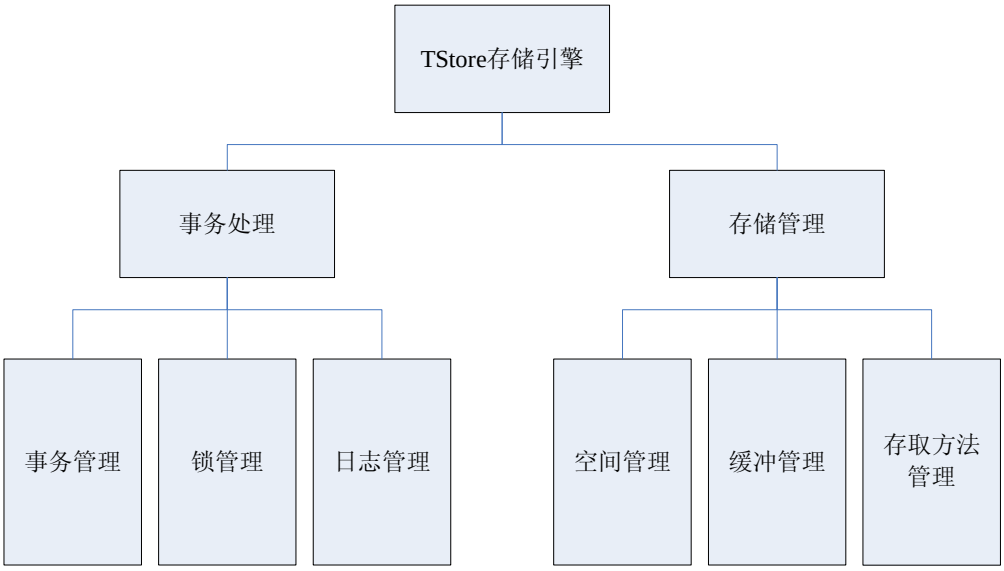

图 1-1　TStore 存储引擎体系结构

1.3 论文组织结构

第一章是绪论，概要地介绍本篇论文的研究背景，研究内容，以及所做的工作。

第二章是存储管理器的设计，从体系结构，模块划分等几个方面，介绍存储管理器的设计。

第三章是存储管理器的实现，根据设计阶段完成的模块划分结果，用流程图、类图、伪代码等多种形式，详细介绍存储管理器的实现细节；

第四章是存储管理器的测试，采用黑盒测试的方式，针对系统划分的各个模块进行功能测试；

第五章是总结与展望，总结本篇论文的成果，简要分析不足之处，并展望今后进一步的研究工作。

第二章　存储管理器的设计

2.1 功能需求

存储管理器所要实现的基本功能就是能够为输入数据分配合适的存储空间，并能够保证存储的数据能被快速地访问[9]。在本论文中，存储管理器还必须能够支持事务处理，在多用户条件下，保证系统运行的正确性和高效性。

存储管理器需要实现的功能包括空间管理、缓冲管理、存取方法管理，每个模块的具体需求如下：

空间管理模块要能够根据不同的用户需求，分配合理的存储空间，维护整个系统的空间分配信息，并且，能够根据用户的需求，在合适的时间完成存储空间的整理。

缓冲管理模块要能够提供设计良好的缓冲调度策略，保证在大多数的情况下都能保持良好的性能，避免由于大量的缓冲页命中失效而引起的系统性能的下降。

存取方法管理模块需要能够提供高效的数据存取方法，保证用户能快速地存取数据，并且，在多用户并发访问的条件下，要保证能够正确、高效地完成数据访问。

2.2 开发环境

存储管理器在开发过程中主要用到的工具和标准如表 2-1 所示。

表 2-1　开发环境

名称描述	具体内容
开发语言	C#
架构模型	C/S
开发平台	Microsoft Windows XP professional 版本 2002 Service Pack 2 .Net Framework 3.5
IDE 工具	Visual Studio 2008

2.3 体系结构

基于存储管理器所要完成的三大功能，本系统由三个模块组成：空间管理、缓冲管理和存取方法管理，如图 2-1 所示。

2.4 对象库设计

本系统为了便于管理系统中大量的对象实例，完成对象基本的序列化和反

序列化，设计了基于“键-值”映射的对象库。

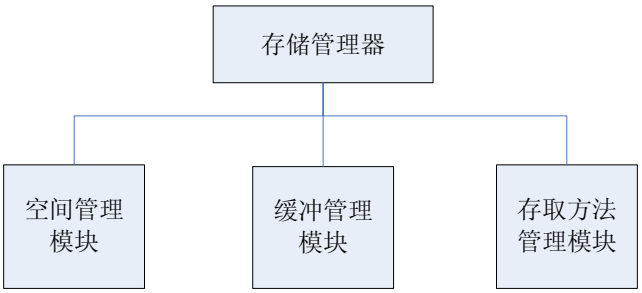

图 2-1　存储管理器体系结构

“键-值”映射是这样的一种机制：将某一确定的对象作为“值”，为该对象分配一个确定的标记作为“键”，然后在对象库中，将这一组“键-值”对应关系保存起来。当有需要的时候，可以在对象库中，使用“键”找到对应的“值”；或者，使用“值”找到对应的“键”。对象库模型的“键-值”映射如图 2-2 所示。

Key1	Value1
Key2	Value2
Key3	Value3
……	……
KeyN	ValueN

图 2-2　对象库“键-值”映射机制

论文设计了 Object Registry 来负责完成上述的键值映射机制，系统中其他的所有的对象管理相关模块，都在 Object Registry 上扩展功能。

在面向对象程序中，对象的序列化和反序列化是必不可少的一项操作，出于简化系统实现的目的，本系统专门将对象的序列化和反序列化的操作封装在对象库中。

具体设计为：当一个对象被序列化时，“键”存储在比特流的头部，在对象库中保存了该对象的类型，而“键”则是该类型在对象库的对应。当反序列化时，先读出比特流头部的“键”，然后用“键”在对象库中找到对应的“值”，即对象的类型，然后按照该类型将对象包装出来。

2.5 空间管理模块设计

根据空间管理的不同内容，可以将其细化为四个子模块，即页管理、槽页管理、空闲空间管理和存储单元管理。

2.5.1 页管理子模块

数据库系统存储的基本单位是叫做页的一个连续的比特集合。页管理子模块负责对系统的页进行管理。

论文将页包含在一个叫做存储容器的逻辑单元内，默认地将存储容器映射为操作系统的文件。

一个页被设计为存储容器中的一个确定大小的块。页管理子模块封装了将页映射到存储容器的过程。页管理子模块维护页面大小的信息，将读入/写出页的工作交给叫做 PageFactory 的类来完成。这样的设计，可以使所有的页被页管理子模块集中管理，并且新建页的类型注册不会引起页管理子模块的改动，同时，也为系统其它模块的执行提供了便利。比如：缓冲管理子模块可以在不做任何修改的情况下，随着页管理子模块的改动而与不同的分页策略协同工作。

页管理子模块不需要管理页内部存储的内容，它只是维护页面最基本的信息，比如：校验和，页编号，页面 Lsn，页类型等。这样，其他模块可以扩展基本的页类型，并且根据不同的需要增加附加特性。

TStore 系统与一般的存储引擎相比，增加了对事务处理的支持，因而系统为每个页分配一个锁存，用它管理对该页的并发访问。这样，页管理器需要调用锁管理器的接口。

2.5.2 槽页管理子模块

槽页管理子模块提供一种增强版的页，它允许在页上的特定位置插入、更新、删除数据。

槽页提供了一个内部允许分段，而整体上又统一的页。槽页内的每一条分段数据被称作一条记录。

通过这样一个基础的结构，其他模块能够在此之上实现更加复杂的功能。

为了达到在槽页内快速访问记录的目的，系统为每个槽页分配一个叫做 Slotted Page 的计数标识。

页面内的每条记录会被分配一个 Slot Number，即槽号。槽号从 0 开始计数，比如：页面内的第一条记录可以使用 0 来访问。

另外，为了存储记录数据，每个槽被设计为要存储一组 short 型的标记值。槽页管理子模块可以通过这些标记值向客户端传递一些信息。

系统会为每个槽分配一个 Slot Position 指针，槽管理子模块可以利用 Slot Position 在指定的位置插入、更新、删除记录。

删除记录仅仅是一个逻辑操作，即逻辑上该记录已经被删除，但是物理上的数据未发生变化。槽管理子模块有专门的接口，在一定的时间将标记为删除

的记录从物理上删除。

2.5.3 空闲空间管理子模块

空闲空间管理，实际上就是为新数据找到合适的位置存储。

论文使用空间映射页来支持空闲空间管理。在每个实例中，系统会专门分配一个空间映射页。当前实例所有可用的闲置页，均在空间映射页中有对应标记。当实例需要申请新的页面时，客户端先在空间映射页中查找未被标记使用的页，然后加上标记，就申请空间成功。

空闲空间管理子模块作为空间管理的基础模块，被系统中的其他模块大量地调用。

在 B-link 树模块中，每一页在空间映射页中有一个位指针对应，通过查看位指针，即可知道该指针对应的页面是否已经被分配。又如：在元组容器中，系统在空间映射页中为每个页面分配两个位来标记该页面的使用情况[5]，即：“11”表示该页已满，“10”表示该页已经使用 2/3，“01”表示该页已经使用 1/3，“00”表示该页为空。

2.5.4 存储单元管理子模块

为了管理系统存储空间下的各个存储单元，论文设计了存储容器的概念。

存储容器，是存储管理操作的基本单位。而事实上，存储容器可以认为是对数据库系统中的文件的封装，即：不管是磁盘上的数据文件，还是外接的其他设备，在存储管理器中，都被当作是一个存储容器来对待。这样的设计屏蔽了底层的文件实现，对外提供通用一致的读写接口，如图 2-3 所示。

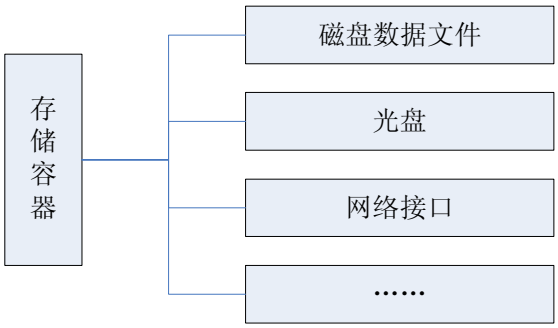

图 2-3　存储容器对文件的封装

存储单元管理子模块维护一个存储容器的实例库，其中存储着对象实例与对应标记的映射关系。通过存储单元管理子模块可以将一个存储容器实例映射为对应的标记值，同样，也可以使用标记得到对应的存储容器的实例。

2.6 缓冲管理模块设计

缓冲管理模块是存储管理器的核心模块之一，它的主要任务是将磁盘页缓存在内存中，减少数据访问过程中读写硬盘的次数，提高系统的效率。

论文为缓冲管理器分配了一个大小固定的缓冲池，缓冲池中的内容随着缓冲页的读入、写出而不停地变化[7]。为了实现方便，缓冲池用一个内存页的链表来实现。

2.6.1 改进的 LRU 算法

设计缓冲管理模块的核心是设计一个性能良好的缓冲页调度算法，以保证将最经常使用的页保留在内存中。为此，本系统实现了一个改良的 LRU 调度算法，如图 2-4 所示。

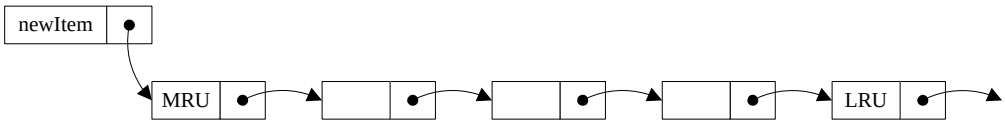

图 2-4　LRU 调度算法链表结构

算法维护着一个包含所有缓冲页的链表，链表的头部表示最近最少使用的页，标记为 LRU 端，链表的尾部表示最近最多使用的页面，标记为 MRU 端。这样的设计基于假设：如果一个页被频繁地访问，那么它很快会被再次访问。每当某一页被访问时，它被移动到链表的 MRU 端，这样经过一段时间，那些最频繁访问的页面就聚集到链表的 MRU 一侧。

考虑到一种特殊的情况：一个客户端读取了相当大数目的一些临时页，上述的那种算法可能导致缓冲页频繁替换，导致系统效率急剧下降。

为了避免出现这样的情况，论文对传统的 LRU 算法加以改进，在每个页上设置一个临时页标记位。当缓冲管理模块读到页面上的这个标记位时，如果该标记被置位，则将该页放置在 LRU 端，而不是 MRU 端。

缓冲管理模块在实现时，需要用到页管理模块提供的接口去实例化新页、读取页、写出脏页。另外，为了支持 WAL（先写日志）协议，缓冲管理器需要在所有页面相关的日志被写入磁盘之后，才可以将该页面写出。

2.7 存取方法管理模块设计

在本文中，存取方法管理实际上主要是对索引进行管理，因而，存取方法管理模块实际上就是索引管理模块。本文仅实现了一种索引数据结构：B-link 树。

2.7.1 B-link 树

论文将 B-link[2]树结构设计为一个拥有确定数目的页的存储容器。容器中

第一页是头页，第二个页面是空间映射页面，第三个页面被分配为树的根结点，根结点永远不会发生变化。在存储管理器中，B-link 树的每一个结点对应着一个页，本文中提到的每一个结点，实际上都是一个页面。

在所有的结点层次中，页面被链接到它右侧的兄弟页。论文规定：树形结构中，叶结点与根结点之间的叫做索引结点。

在叶结点上，设置一个额外的标记叫做最高键值[4]。在索引结点上，最后一项充当最高键值的角色。在结点中的所有项都能保证其值小于等于最高键值。由于叶结点的特殊结构，其最后一项的值可能和最高键值不同。

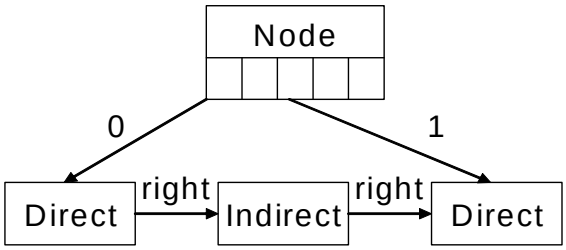

图 2-5　直接孩子页与非直接孩子页

在索引结点中，每一项对应一个指向孩子结点的指针。孩子结点包含的所有项的值均小于等于其在索引结点中对应项的值。

一个直接相连的孩子结点可以由父亲结点通过一个指针访问。一个非直接的孩子结点要通过直接孩子结点的向右邻接指针来访问，如图 2-5 所示。

如果是直接孩子结点，那么孩子结点中的最高键值将与索引结点中的对应项一致。如果孩子结点有一个邻接的非直接相连结点，那么，孩子结点的最高键值将小于索引结点中的对应项的值。

除了根结点之外，所有的结点必须至少含有两项（除了叶结点的最高键值）。

根结点允许有一个右兄弟。这种情况下，导致树的高度增加。

在每一层的最右侧的一项，系统设置了特殊的一项，这一项可以当作是逻辑上的无穷大。初始时，B-link 树结构仅包含无穷大项。随着树的生长，结点的分裂，无穷大项被推向了每一层的最右侧结点中的最后一项。论文规定这一项不能被删除。

2.7.2 结点的结构

在每个层次上，所有的结点从左向右链接在一起。在任意一层，最右侧的指针被设置为空。

叶结点拥有图 2-6 所示的结构。

在一个叶结点内，highkey 是额外设置的一项，可能与结点内最右的一项相同，也可能不同。可能引起 highkey 发生改变的操作是分裂、合并和结点重构。所有页面内的键值都保证小于等于 highkey。

图 2-6　B-link 树叶结点内部结构

最后一个键就是 highkey。在每一层的最右侧的结点拥有一个特殊的 highkey，值为 INFINITY。

每个索引项包含一个指向孩子结点的指针。这个孩子结点包含所有键值均小于等于该键值。

本系统为每个结点增加了一些限制：容量最大的项，其大小不能超过页面大小的 1/8。除了根结点之外，每个结点至少包含两个键。

2.7.3 值相关操作

1、 查找

查找操作是 B-link 树最基本的操作之一，是其他复杂操作的基础。

根据 B-link 树的结构，论文设计的查找操作执行如下：

1) 从根结点开始，扫描页面内的每一项，如果当前项的值小于查找值，则向后遍历，否则，转 2)；
2) 如果当前项的值不小于查找值，则根据当前项锁包含的指针，指向其对应的孩子结点，依次类推，直到扫描到达树叶结点。
3) 在树叶结点中扫描各项，如果找到与查找值相同的项，则返回该项，否则返回空值，表示页面未找到；

2、 插入

插入操作需要在查找的基础上完成，并且在插入过程中，需要考虑页面的向上溢出的情况。

插入操作的执行如下：

1) 执行查找操作，找到合适的插入位置；
2) 将插入项插入到指定的位置，如果该项插入之后，造成了页面向上溢出，即页面使用饱和，则需要分裂结点；

3、 删除

删除操作也需要在查找的基础上完成。在删除过程中，需要考虑页面向下溢出的情况。

删除操作的执行如下：

1） 执行查找操作，在 B-link 树中找到删除项所在的位置；

2） 将删除项从该位置删除，如果该项被删除之后，造成页面向下溢出，造成页面使用过少，则需要合并相邻结点。

2.7.4 结构相关操作

1、 分裂

分裂即将一个结点分裂成两个，新生成的结点用指针链接到旧的结点上。实际上，分裂操作是将原来一个页上的数据，分摊到两个页面上。

当一个结点被分裂时，原有各项在旧结点和新生成的结点之间重新分配。旧结点的右指针指向新生成的结点。旧结点中的 highkey 发生变化，与结点中的最大键值相同。这样 highkey 会随着结点中的键删除而发生变化，如图 2-7 所示。

索引结点的分裂与叶结点的分裂类似，只是不需要设置额外的 highkey。

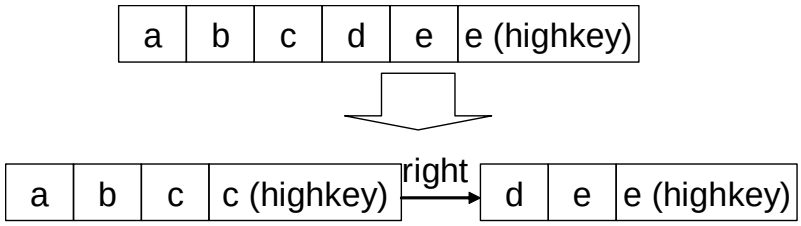

图 2-7　B-link 树结点的分裂

2、 合并

合并是分裂的逆操作，它连接两个已经存在的结点，并创建一个新的单独的结点。

出于提高并发度的考虑，空间映射页面更新被当作一个独立的只重做操作，在合并操作完成之后被日志记录。

3、 连接

连接操作从父结点向孩子结点创建一个连接，如图 2-8 所示。

操作的执行如下：

1） 新建索引记录；

2） 在父结点中找到左孩子结点对应的索引记录；

3） 将上个孩子指针指向右孩子；

4） 在上面的索引记录前面插入新的索引记录。

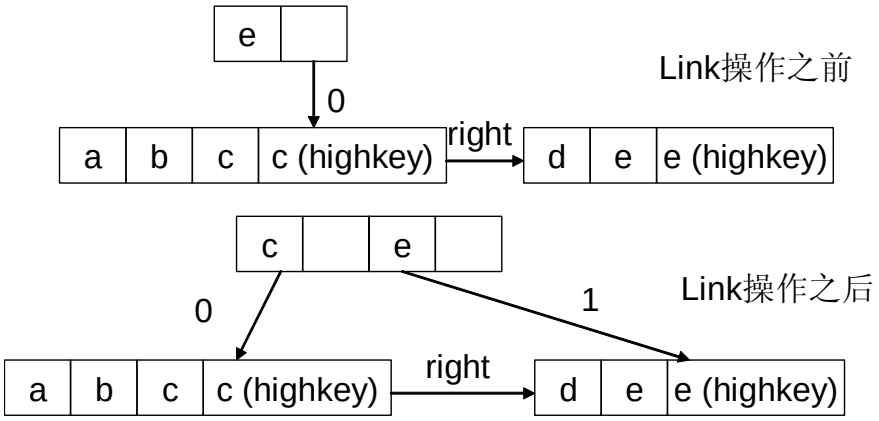

图 2-8　B-link 树结点链接操作

4、 解连接

解连接操作是连接操作的逆操作。

操作的执行如下：

1） 在父结点中删除包含指向孩子结点指针的索引记录

2） 将下一条索引记录的孩子指针指向下层的左侧的孩子结点

5、 结点重构

论文实现这个操作时，仅仅是将拥挤结点中的一项移动到它的相邻结点，而不是大范围地将两个结点中的键重构，如图 2-9 所示。

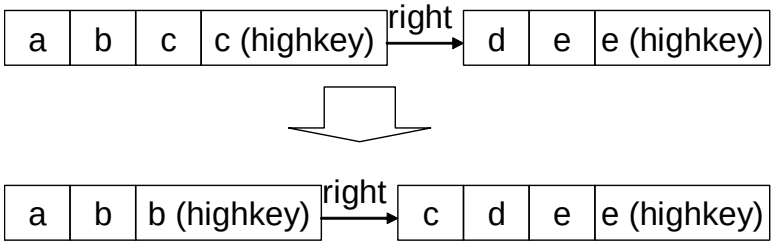

图 2-9　B-link 树结点重构操作

结点重构操作被日志记录为一个多页面只重做记录。

6、 增加树高度

论文规定，当这一操作触发时，之前根结点一定发生了分裂，且必有一个右兄弟。

在这一操作过程中，生成一个新的结点，并且根结点中的信息被复制到该结点中，然后根结点重新被初始化，将其孩子指针指向新建结点作为它的左孩

子，同时旧的右邻接结点作为它的右孩子，如图 2-10 所示。

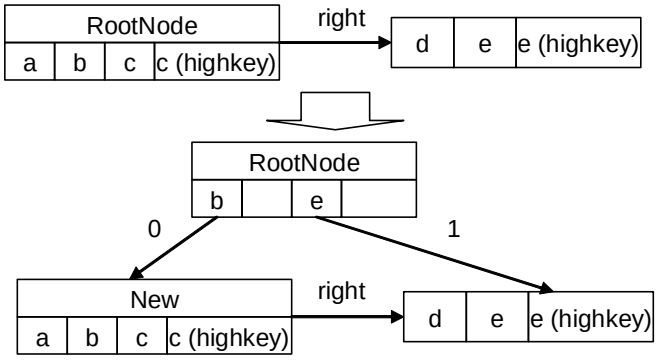

图 2-10　B-link 树增加树高度操作

7、 降低树高度

论文规定，根结点中只包含一个孩子结点，并且该孩子结点没有邻接结点时，执行降低树高度的操作。

操作执行如下：

1） 将根结点的唯一的孩子结点的信息复制到根结点中；

2） 将该孩子结点删除，回收其存储空间。

第三章　存储管理器的实现

本章主要介绍存储管理器的详细实现，将深入到方法的层次对各个功能模块的关键技术做深入地介绍。

出于代码管理方便，存储管理器的代码从整体上划分为接口和实现两个部分。接口部分主要是各个功能模块定义各自必须要实现的接口，并且包含了除主要功能实现之外的其他必要准备。实现部分则是各个功能模块的核心代码实现。这样的代码管理，使得系统最核心的代码被很好地封装了起来，能够被外界访问的仅仅是接口部分。

项目文件下的 API 命名空间即接口定义部分，IMPL 命名空间即核心代码实现部分。后续的每个功能模块的代码都会在 API 和 IMPL 两个命名空间下部署。

3.1 对象库的实现

存储管理器实现的对象库有两个重要的功能：首先，它提供了一种“键-值”映射机制，便于管理系统中出现的各种类对象；另外，它封装了对象序列化和反序列化的过程，通过操作对象库，可以存储或者提取一个指定的类对象。

根据系统的设计，系统在存储一个对象的时候，会先把对象的类型码保存在该对象所在的数据段的头两个字节。在从一段比特流中提取数据的时候，先读出对象的类型码，然后从对象库中用类型码得到对象的类型信息，再根据类型信息将对象包装出来。综上，对象库核心维护的是一个类型信息表。

3.1.1 对象库的核心数据结构

论文实现的对象库核心是一个叫做 typeRegistry 的数组，所有类型信息都保存在这个数组中，typeRegistry 数组的定义如图 3-1 所示。

```
internal ObjectDefinition[] typeRegistry
                                = new ObjectDefinition[Int16.MaxValue];
```

图 3-1　typeRegistry 数组定义代码

其中，typeRegistry 的类型为 ObjectDefinition，该类型是一个基类，它拥有两个子类 SingletonObjectDefinition 和 FactoryObjectDefinition。

在实现的时候，考虑到对象可能划分为单体类和工厂类两种，而且，两种类型在对象序列化和反序列化的时候的操作有明显的不同，故而采用了上述的实现方式。

3.1.2 键-值映射方式的实现

由前文知道，对象库里面维护的实际上是类型信息。考虑对象的提取过

程，论文实现的“键-值映射”中的“键”即是类型码，“值”即是类型定义。

类型码用一个 int16 型的 typecode 来实现，由于对象库只是维护类型信息，因而在相当程度上保证了对象库的轻巧，所以 typeRegistry 的容量仅仅为 2^{16}。

类型定义则是由 ObjectDefinition 类实现。系统将 ObjectDefinition 定义为一个抽象类，规定了基本的操作。按照上面的描述，SingletonObjectDefinition 和 FactoryObjectDefinition 两个类根据不同的情况，实现了两种类型的对象定义。

ObjectDefinition 的类图如图 3-2 所示。

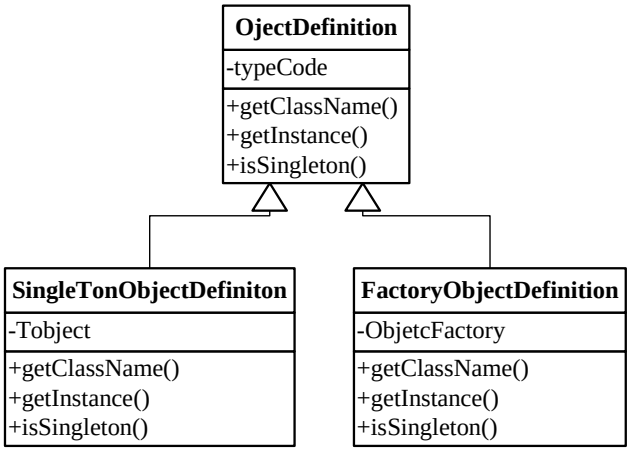

图 3-2 ObjectDefinition 子模块类图

3.2 空间管理模块实现

根据第二章的设计，空间管理模块将细化为四个子模块来实现，分别是页管理、槽页管理、空闲空间管理和存储单元管理。对应到代码实现上，每个子模块都是一个单独的命名空间，分别对应为 PageMgr、SlottedPageMgr、FreeSpaceMgr 和 StorageMgr。

3.2.1 PageMgr

PageMgr 命名空间下的类主要协作完成页管理的功能。

3.2.1.1 PageMgr 的类组织

PageMgr 命名空间下的类主要分作两部分，一部分属于 API 命名空间下，包括 Page、PageException、PageFactory、PageId、PageManager、PageReadException 六个类，完成了 PageMgr 下基本的接口定义。其中：

Page 是所有页面实现的基类，定义了最基本的页面属性和操作。

PageException 定义了页相关的异常处理。

PageFactory 控制页对象的实例化，或者是从内存中实例化出一个页面，或者是从比特流中包装出一个页面。

PageId 专门为系统提供了定义良好的页面编号：一个页面的 Id 由 containerId 和 pageNumber 两部分组成。

PageReadException 专门定义了读取页面过程中的异常处理。

PageManager 定义了页管理器需要实现的全部接口。PageManager 负责实例化各种类型的页面，以及从指定的存储容器中提取页面和向指定的存储容器中存储页面。

另一部分属于 IMPL 命名空间下，包括 RawPage 和 PageManagerImpl 两个类，实现了 API 中定义的接口，是页管理器的核心实现部分。其中：

RawPage 实现了抽象类 Page，给出了 Page 的一个基本的实现。

PageManagerImpl 类实现接口 PageManager，将其中定义的方法全部实现。

3.2.1.2 PageMgr 的关键结构

页是 TStore 系统中最为重要的一个概念，API 命名空间中的 Page 类完成了对 Page 的定义，如图 3-3 所示。

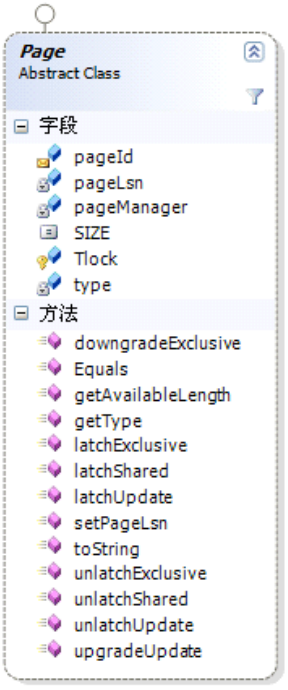

图 3-3　Page 类关系图

TStore 系统是支持并发处理的存储引擎，因而，在 Page 的定义中增加了对

加锁的支持，由上面的类图可知，每个页面上包含一个类型为 ILatch 的成员变量 Tlock，由它负责对该页面的并发控制。同时，Page 的定义中还增加了 latchExclusive 、 latchShared 、 latchUpdate 、 unlatchExclusive 、 unlatchShared、unlatchUpdate、upgradeUpdate 七个方法，在页面被并发访问时，这些方法会被调用。

3.2.1.3 PageMgr 的方法定义

PageManager 是一个接口，仅仅定义了几个通用的方法，而系统把实际的页面读写操作交给 PageFactory 来完成，这样的好处在于 PageManager 可以用来管理所有的页而不用关心每种页不同的类型。

PageManager 定义了如下七个方法，参见图 3-4。

getInstance 返回一个根据指定类型实例化的页。

getLatchFactory 返回当前的 PageFactory 对应的 LatchFactory，这个 LatchFactory 用来在页面上加锁。

getPageSize 返回当前 PageManager 管理下的所有页的大小之和。

getRawPageType 返回当前页的类型码。

getUsablePageSize 返回当前 PageManager 管理下的所有页的可用空间的大小。

retrieve 使用 PageId，从指定的存储容器中提取指定页面。

store 将页面保存到指定的存储容器中。

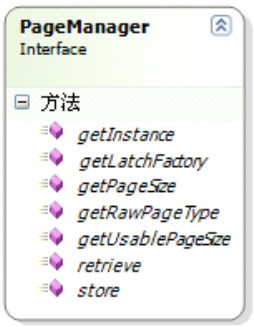

图 3-4　PageManager 类关系图

3.2.1.4 PageMgr 的关键方法实现

对于 PageManager 来说，最重要的是 retrieve 和 store 两个方法，以下仅对这两个方法做详细介绍。

1、 retrieve

retrieve 方法执行的伪代码如下：

1) 用 pageId 获取一个存储容器的实例 container；
2) 新建一个 byte 型的数组 data，用 pegeId 计算数据相关参数，并使用存储容器提供的接口，从 container 中读取数据到 data 中；
3) 计算 data 的校验和，如果计算值与存储的校验和一致，则用 pageId 和 data 包装出一个新的页面，否则报错，程序终止。

2、 store

store 方法执行的流程如下：

1) 由输入页面的 pageId 获取一个存储容器的实例 container；
2) 新建一个 ByteBuffer 对象 bb，然后将 page 中的数据暂存到 bb 中，并且计算 page 中数据的校验和，一并存储在 bb 中；
3) 使用 container 提供的方法，将 bb 中的数据保存到磁盘的指定位置。

3.2.2 SlottedPageMgr

SlottedPageMgr 命名空间下的类负责槽页管理。

3.2.2.1 SlottedPageMgr 的类组织

SlottedPageMgr 在 API 命名空间下包括 SlottedPage 和 SlottedPage Manager 两个类，在 IMPL 命名空间下是与 API 对应的 SlottedPageImpl 类和 SlottedPageManagerImpl 类。

由于槽页的特殊性质，SlottedPageMgr 的核心在于实现一个结构合理、性能良好的 Slotted Page，因而存储管理器将 SlottedPage 作为核心的单元，SlottedPageManager 仅仅是维护 SlottedPage 的类型信息。

3.2.2.2 SlottedPageMgr 的关键结构

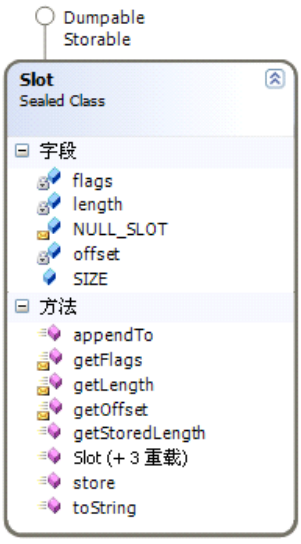

图 3-5　Slot 类关系图

1、 Slot

在 SlottedPageImpl 中首先定义了 Slot 类，对槽页中最基本的单位进行了规定，如图 3-5 所示。

一个 Slot 对象是 SlotPage 中的 SlotTable 中的一项，核心的内容包括 offset、flags、length 三项。其中，offset 表示该 Slot 的物理偏移，flags 是由 SlottedPageImpl 设置的标记值，length 表示该 Slot 中包含的数据的长度。

Slot 拥有自己的 store 方法，仅仅是将 Slot 对象中的 offset、flags、length 三个属性存储在给定的 ByteBuffer 中。

2、 SlotTable

SlotTable 维护了基本的 Slot 的信息，SlottedPage 通过维护这样一个 SlotTable 来对 slot 进行管理，SlotTable 的代码如图定义图 3-6。

```
[NonSerialized]
private List<Slot> SlotTable = new List<Slot>();
```

图 3-6　SlotTable 定义代码

SlotTable 用一个泛型的链表来实现，表中的每一个元素均为 Slot 类型。

3.2.2.3 SlottedPageMgr 的方法定义

在 API 命名空间下，将 SlottedPage 实现为一个继承自 Page 的抽象类。在该抽象类中，定义了 SlottedPage 需要支持的对外接口，如图 3-7 所示。

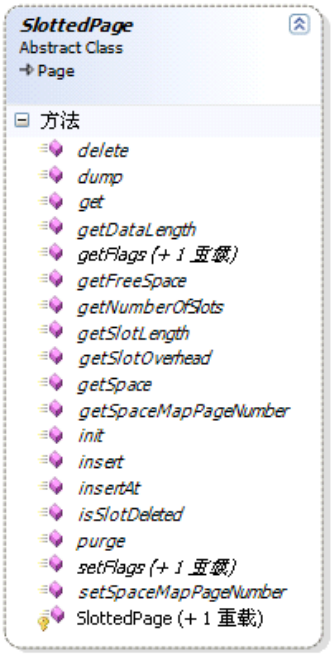

图 3-7　SlottedPage 类关系图

insert 支持向槽页中直接插入数据，而 insertAt 方法则支持从指定位置向槽页中插入数据。

setFlags 和 getFlags 方法支持在指定的槽上设置、获取标记值。

delete 和 purge 提供了两种删除指定的槽的方法，其中，delete 方法仅仅是逻辑删除，在要删除的槽上标记删除，而 purge 则会物理删除，真正地释放该槽的空间。

3.2.2.4 SlottedPageMgr 的关键方法实现

1、 insert

insert 直接向页面插入元组，如果在插入过程中发现有已经标记删除的元组，会将这些项重新利用。

insert 方法的流程图如图 3-8 所示。

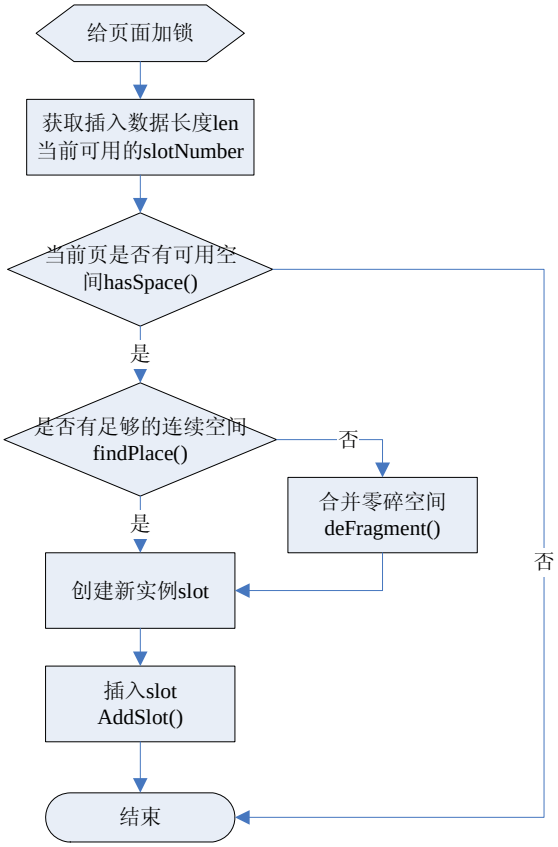

图 3-8　insert 方法流程图

由上述流程图所示，在调用 findPlace 方法的过程中，会将已标记删除的元组的空间释放。在调用 deFragment 方法时，会将已经释放的空间整理成连续的空间。

2、 insertAt

insertAt 方法与 insert 的区别在于它是在指定的位置插入新元组。本方法在实现中特殊之处有两点：

1) 需要对插入位置加互斥锁，阻止其他线程抢占；

2) 根据指定的位置，计算需要的页面空间。

insertAt 的部分代码如图 3-9。

```
validateLatchHeldExclusively();
int requiredSpace = 0;
if (slotNumber == numberOfSlots)
{
    requiredSpace = calculateSlotLength(len);
}
else
{
    if (isSlotDeleted(slotNumber))
    {
        requiredSpace = len;
    }
    else if (replaceMode)
    {
        int currentLen = getDataLength(slotNumber);
        requiredSpace = len - currentLen;
    }
    else
    {
        requiredSpace = calculateSlotLength(len);
    }
}
```

图 3-9　insertAt 方法的关键代码

3、 delete

delete 方法仅仅是将指定位置的元组标记删除，其实现代码如图 3-10。

```
public override void delete(int slotNumber)
{
    validateLatchHeldExclusively();
    validateSlotNumber(slotNumber, false);
    if (isSlotDeleted(slotNumber))
    {
        return;
    }
    freeSpace += getDataLength(slotNumber);
    deletedSlots++;
    slotTable[slotNumber] = Slot.NULL_SLOT;
}
```

图 3-10　delete 方法实现代码

4、 purge

purge 与 delete 的区别在于：delete 只是逻辑删除，而 purge 则是物理删除，会真正地将指定元组的空间释放。

由图 3-11 所示的 purge 方法的实现代码，purge 在删除指定项时，会改变标记删除计数器 deletedSlots，并且将该项从 SlotTable 中删除。

```
if (isSlotDeleted(slotNumber))
 {
    deletedSlots--;
 }
 freeSpace += getSlotLength(slotNumber);
```

图 3-11　purge 方法的关键代码

3.2.3 FreeSpaceMgr

FreeSpaceMgr 命名空间下的类主要负责系统空闲空间的分配和回收。

3.2.3.1 FreeSpaceMgr 的类组织

FreeSpaceMgr 在 API 命名空间下包括 6 个类，分别为 FreeSpaceChecker、FreeSpaceCursor 、 FreeSpaceManagerException 、 FreeSpaceMapPage 、FreeSpaceScan 和 FreeSpaceManager。其中：

FreeSpaceChecker 定义为一个检测类，即在分配一个页的时候，检测该页是否满足需要的大小。

FreeSpaceCursor 是一个接口，定义了在一个存储容器内查找、更新空闲空间信息需要的方法。

FreeSpaceManagerException 是一个异常类，针对整个空间管理模块。

FreeSpaceMapPage 定义为空闲空间管理中空间映射页的抽象类。

FreeSpaceScan 提供了在一个存储容器中访问一个非空页面的接口。

FreeSpaceManager 是接口类，定义了空闲空间管理必需的方法。

在 IMPL 命名空间下只有 FreeSpaceManagerImpl 类，实现了上述的 FreeSpaceManager 接口，包含了空闲空间管理的核心代码实现。

3.2.3.2 FreeSpaceMgr 的关键结构

1、 SpaceMapPage

SpaceMapPage，即空间映射页，是空闲空间管理的重要结构[6]，根据系统的设计，空闲空间管理子模块维护的所有页都会在 SpaceMapPage 中有一个对应的标记。系统可以通过简单地操作 SpaceMapPage，实现对页面的监控和操作。

API 命名空间下的 FreeSpaceMapPage 是抽象类，定义了两个基本的方法：

1） getSpaceBits 提取当前页面在 SpaceMapPage 中的对应标记。

2） setSpaceBits 更新当前页面在 SpaceMapPage 中的对应标记。

FreeSpaceManagerImpl 内嵌抽象类 SpaceMapImpl 继承自 FreeSpaceMapPage，提供了基本的空间映射页的实现，同时，提供了一部分抽象方法，允许不同种类的空间映射页实现。

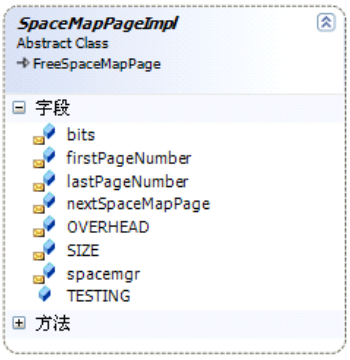

图 3-12　SpaceMapPageImpl 类关系图

由上可知，存储管理器实现的 SpaceMapPage，核心是叫做 bits 的 byte 型的数组，全部的页面使用信息保存在这个数组中。同时，还维护了一些指针，便于在 SpaceMapPage 上进行操作：

firstPageNumber，指向空闲空间管理器维护的第一个 page。

lastPageNumber，指向最后一个 page。

nextSpaceMapPage，指向下一个 SpaceMapPage。

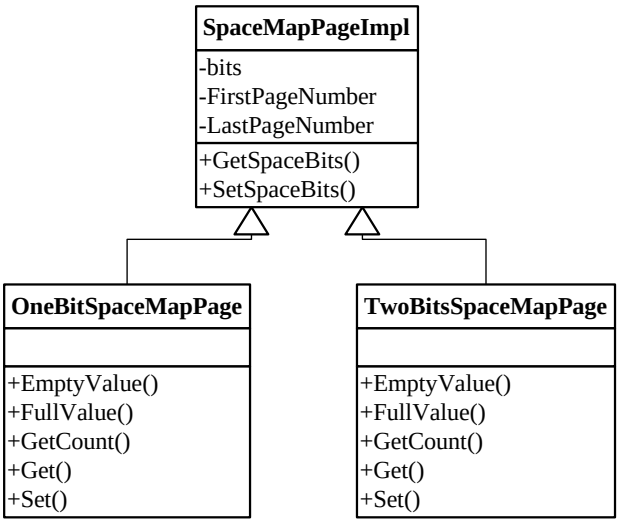

图 3-13　SpaceMapPage 实现子模块类图

论文提供了两种 SpaceMapPage 的实现：第一种为每个页面设置一位的位指针，通过设置 0 或 1 表示该页是否被使用；另一种为每个页面设置两位的标记，根据标记值的不同，反映出页面的使用状况，如图 3-13 所示。

3.2.3.3 FreeSpaceMgr 的方法定义

在 FreeSpaceManager 类中，定义了如下的七个操作：

CreateContainer：创建一个拥有指定名称的容器，并且将该容器用指定的 id 在对象库中注册。

ExtendContainer：向空闲空间管理器的容器中添加一个段。段是一类页面的集合，也是空间管理器分配空间的基本单位。

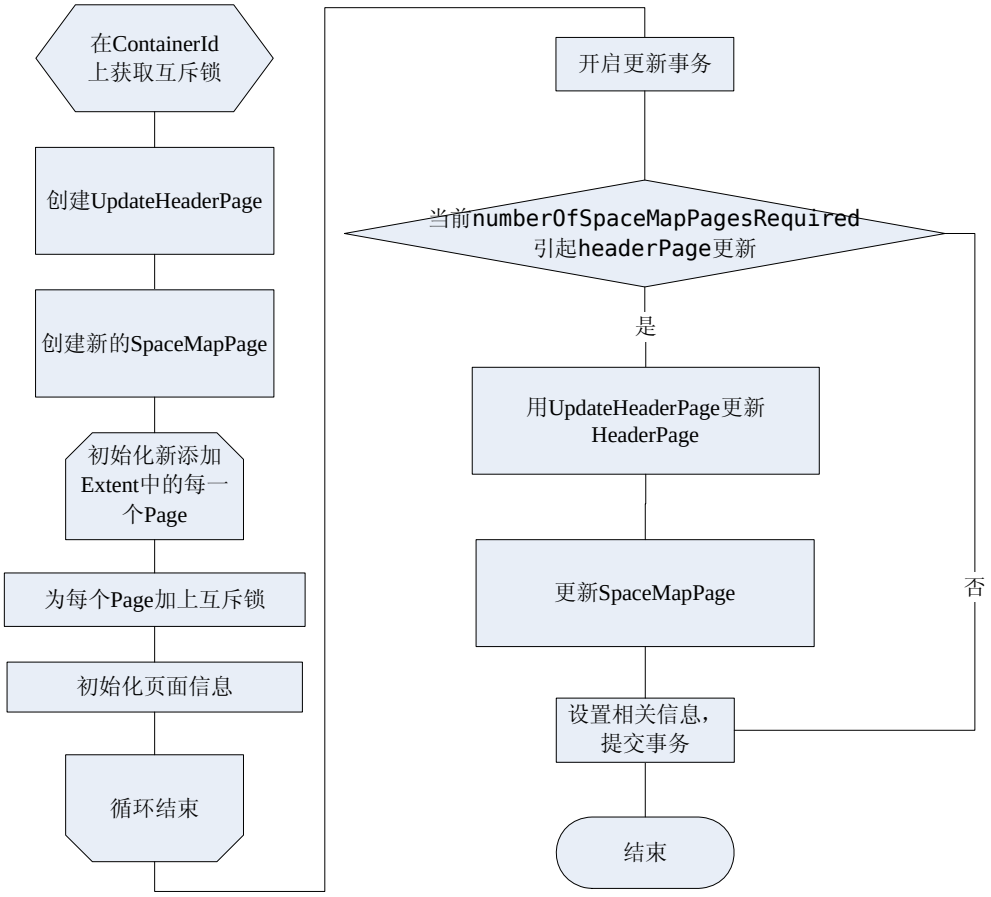

图 3-14　ExtendContainer 方法流程图

DropContainer：释放一个已经存在的容器。

GetSpaceCursor：获取一个游标，便于在容器内部访问空闲空间的信息。

GetPooledSpaceCursor：获取一个在容器内访问空闲空间信息的游标，与 GetSpaceCursor 不同的是，本方法可能会返回一个合并的游标。

ReleaseSpaceCursor：将已经获取的游标释放。

OpenScan：开启一个扫描，扫描空闲空间管理器包含的容器下所有的非空页面。扫描将会按照在容器中存储的顺序将页面依次返回。

3.2.3.4 FreeSpaceMgr 的关键方法实现

本节重点说明 ExtendContainer 方法。

在调用本方法之前，方法调用者必须先在空间管理器维护的 container 上面获取共享锁。

ExtendContainer 方法的流程图如 3-14 所示。

3.2.4 StorageMgr

3.2.4.1 StorageMgr 的类组织

StorageMgr 在 API 命名空间下包括五个类，分别为 StorageContainer、StorageContainerFactory 、 StorageContainerInfo 、 StorageException 和 StorageManager，其中：

StorageContainer 定义了基本的存储容器，屏蔽底层的具体实现，向其他模块提供通用的存储容器。

StorageContainerFactory 被用来生成存储容器的实例。

StorageContainerInfo 保存一个活的存储容器的基本信息。

StorageException 定义了 StorageMgr 相关的异常。

StorageManager 管理一个存储容器的对象库，支持对存储容器实例以“键-值”映射的方式访问。

在 IMPL 命名空间下包括 FileStorageContainer、FileStorageContainer Factory 和 StorageManagerImpl 三个类。

FileStorageContainer 实现了 StorageContainer 接口，基于 Storage Container 定义出 File 的概念。

FileStorageContainerFactory 负责创建基于 StorageContainer 实现的文件实例。

StorageManagerImpl 实现了 API 中的 StorageManager 接口，是存储单元管理器具体的代码实现。

3.2.4.2 StorageMgr 的关键结构

由前面的设计可知，StorageManager 在管理存储容器的时候，支持对存储容器以“键-值”映射的方式访问。为了达到这样的目标，StorageManager 的维护了一个 HashTable 类型的 map，如图 3-15 所示。

```
Hashtable map = new Hashtable();
```

图 3-15　StorageManager 核心结构代码

C#语言提供的 HashTable 类型，允许向其中插入任意的“键-值”对，并且可以在需要的时候用“键”提取需要的“值”。

向 map 中添加信息的实现如图 3-16 所示。

```
map.Add(id, new StorageContainerHolder(id, container));
```

图 3-16　StorageManager 添加存储单元示例代码

从 map 中提取值的实现如图 3-17 所示。

```
containerHolder = (StorageContainerHolder)map[id];
```

图 3-17　StorageManager 读取存储单元示例代码

3.2.4.3 StorageMgr 的方法定义

在 StorageManager 类中，定义了五个操作，分别是：

register：为一个存储容器实例分配一个 int 型的 id，并且把实例与 id 的对应关系保存起来。

getInstance：利用存储容器注册过的 id，提取出一个存储容器的实例。

remove：关闭并且移出一个指定的存储容器。

shutdown：关闭所有的存储容器。

getActiveContainers：返回一个表，包含系统中所有的活动存储容器。

3.2.4.4 StorageMgr 的关键方法实现

根据前面的说明，可知 StorageManagerImpl 中实现的方法都是基于其关键结构Hashtable类型的map展开的，以下重点介绍register和remove两个方法。

1、 register

register 方法负责向 StorageManager 注册新的存储单元，原理上比较简单，使用 map 自身提供的方法即可实现，要注意的是：一个 StorageManager 实例是被多个线程共享，在多线程并发访问的过程中，需要在 map 上加锁，保证数据的一致性。

```
public void register(int id, StorageContainer container)
{
    lock (map)
    {
        map.Add(id, new StorageContainerHolder(id, container));
    }
}
```

图 3-18　register 实现代码

2、 remove

remove 方法实现了从 StorageManager 中移除存储单元的功能，其操作的流程如图 3-19 所示。

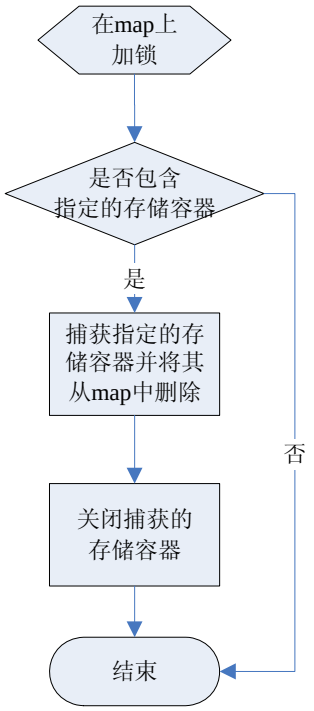

图 3-19　remove 方法流程图

3.3 缓冲管理模块实现

缓冲管理器的实现放在了 BufferMgr 命名空间下面。

3.3.1 BufferMgr 的类组织

BufferMgr 在 API 命名空间下包括如下的四个类：

BufferAccessBlock，当一个页面被暂存在内存中时，需要使用 BufferAccessBlock 将该页面包装起来，然后就可以开始对该页面的各种操作。

DirtyPageInfo 其中包含脏页的相关信息。

BufferManagerException 类定义了 BufferManager 相关的异常。

BufferManager 负责维护缓冲池，作为一个接口，定义了缓冲管理器需要实现的基本操作。

在 IMPL 命名空间下只包含一个 BufferManagerImpl 类，实现了 BufferManager 接口，包含了详细的实现细节。

3.3.2 BufferMgr 的关键结构

以下介绍 BufferManager 中的两个重要的结构：BufferControlBlock 和 BufferPool。其中 BufferControlBlock 是在缓冲管理中被实际操作的单位，对缓冲池的页的各种操作都是通过操作 BufferControlBlock 来实现的。BufferPool，即缓冲池，是缓冲管理最为关键的结构。缓冲管理的各种操作和调度算法，都在在缓冲池设计良好的结构之上得以实现的。

1、 BufferControlBlock

BufferControlBlock 在概念上包装着一个在内存中缓冲的页，维护着该页的相关信息。这些信息包括：该页在缓冲池中位置、该页被标记的次数、引起该页改动的最老动作的日志标号、该页是否为脏页、该页是否被写出磁盘。

在BufferControlBlock中并不包含真正的Page，它拥有一个指针pageId，指向其对应的页面。

BufferControlBlock 结构如图 3-20 所示。

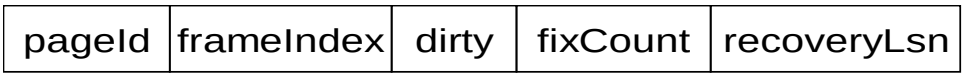

图 3-20　BufferControlBlock

2、 BufferPool

BufferPool 作为缓冲管理器最为核心的数据结构，在实现过程中并不是使用一个单纯的结构来实现的，而是包括三个子模块构成的，如图 3-21 所示。

```
private Page[] bufferpool;
private BufferHashBucket[] bufferHash;
private  List<BufferControlBlock> lru = new
                            TStoreLinkedList<BufferControlBlock>();
```

图 3-21　BufferPool 实现的部分代码

Page 类型的数组 bufferpool 是缓冲管理器中真正保存页面的地方。在这个数据中的每一项叫做一帧。

bufferHash 提供了用哈希的方式对缓冲池中的帧进行访问，这样大大地提高了对缓冲池的帧进行访问的效率。在 BufferHashBucket 中封装的是 BufferControlBlock 类型的链表，经过这样的封装，在对帧访问之前的所有操作，都可以只用轻量级的 BufferControlBlock 来实现，大大地提高了系统运行的效率。

lru 是系统自己实现的一个 BufferControlBlock 类型的链表，在这个链表中，维护了系统对缓冲池中各帧的访问信息，缓冲管理器实现的 LRU 调度算法[13]就是依靠这一链表结构来实现的。其中，链表的头部为 LRU 端，链表的尾部为 MRU 端。

3.3.3 BufferMgr 的方法定义

在 API 中的 BufferManager 接口，定义了缓冲管理器必须要实现的十个方法：

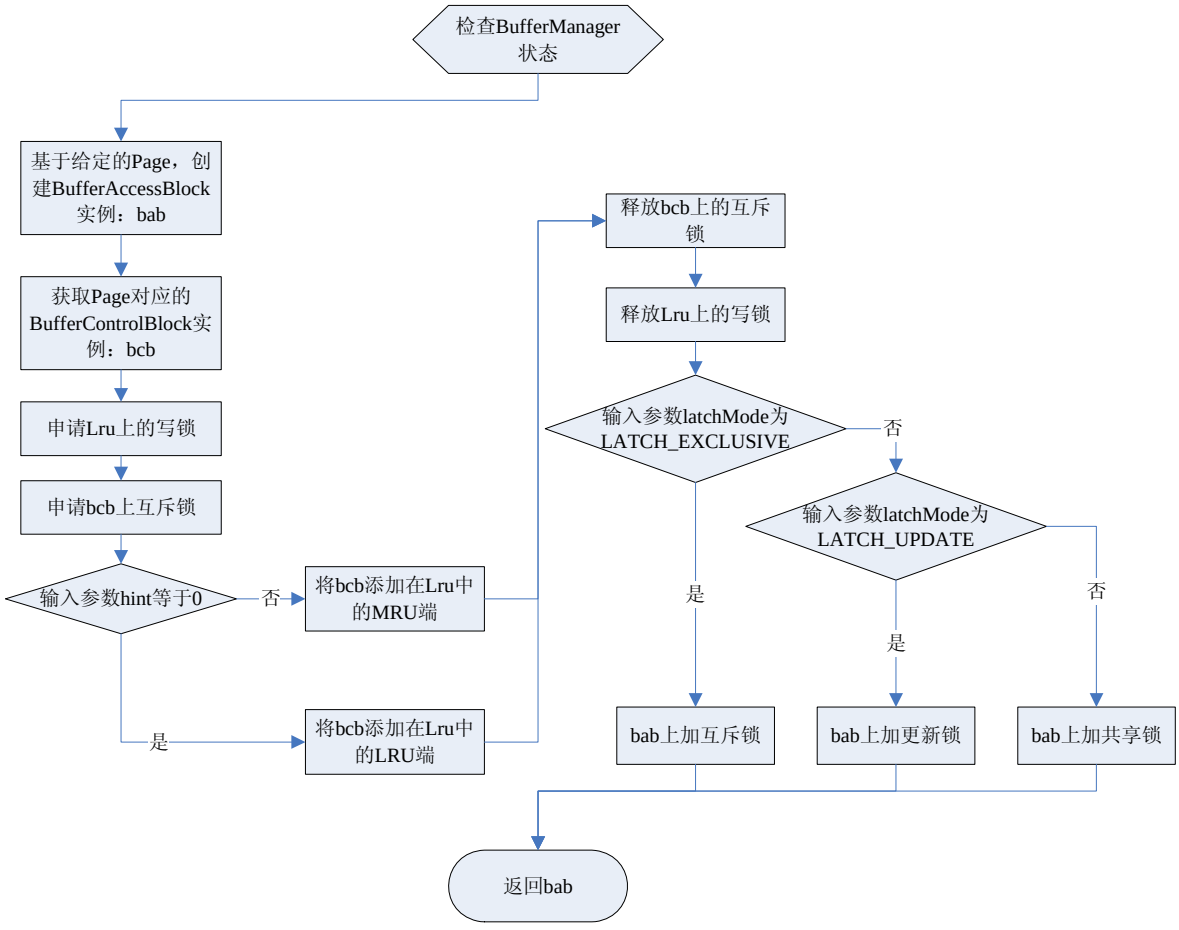

图 3-22　fix 方法流程图

start 开始一个缓冲管理器实例，这可能会开启后台的一个新的线程。

shutdown 关闭一个缓冲管理器实例，后台所有相关的线程都必须关闭；同时，要将缓冲池中所有的脏页写回磁盘。

fixShared 标记内存中的一个页，如果在内存中未找到，需要从硬盘中读取该页；然后在该页上加共享锁。

fixExclusive 标记内存中的一个页，如果在内存中未找到，需要从硬盘中读取该页；然后再该页上加互斥锁。

fixForUpdate 标记内存中的一个页，如果在内存中未找到，需要从硬盘中读取该页；然后在该页上加更新锁。

getDirtyPages 返回缓冲池中所有的脏页。

invalidateContainer 当有存储容器失效时，需要在该容器包含的页面上进行标记；这种情况可能由文件删除而引起。

writeBuffers 将缓冲池中的帧写入磁盘。

updateRecoveryLsns 将当前页面的 recoveryLsn 与缓冲池中的信息进行同步。

3.3.4 BufferMgr 的关键方法实现

在 BufferManagerImpl 中，fixShared、fixExclusive 和 fixForUpdate 都是将不同的参数传递给 fix 方法来实现的，图 3-22 即为 fix 方法的流程图。

```
if (hint == 0) //检查页面指示标记，不是临时页面
{
   if (lru.getLast() != nextBcb)//判断当前页是否在MRU端
   {
       if (nextBcb.isMemberOf(lru))
       {
           lru.remove(nextBcb);//将当前页从原位置移除
       }
       lru.addLast(nextBcb);//将当前页放到MRU端
    }
}
else//检查页面指示标记，是临时页面
{
    if (lru.getFirst() != nextBcb) //判断当前页是否在LRU端
    {
        if (nextBcb.isMemberOf(lru))
        {
           lru.remove(nextBcb); //将当前页从原位置移除
        }
        lru.addFirst(nextBcb);//将当前页放到LRU端
     }
}
```

图 3-23　缓冲管理器页面调度算法部分代码

在用 bab 创建了目标页对应的 bcb 之后，可以保证该页已经被装进哈希表中，

但是不一定在Lru中。在这种情况下，该页能够被客户端发现，但是不一定能被缓冲管理模块中的页面替换逻辑发现。但是，这样却不会引起任何的问题，因为：

1、如果该页不是第一被访问，那么它就已经存在于Lru表中，那么它就能够被缓冲管理模块的页面置换逻辑发现。

2、另外的仅有的可能是，该页刚刚被读入，或者它是新创建的页，在这种情况下，该页不可能是脏页，并且其fixCount至少为一，这样就不会被当作脏页写出去。

图3-23所示的部分代码，显示了缓冲管理器所实现的缓冲管理策略。

3.4 存取方法管理模块实现

出于典型性的考虑，存储管理器的存取方法管理器仅仅实现了索引管理器；并且，为了更好地说明TStore系统对并发访问的支持，仅仅实现了B-link树一种索引结构。

存取方法管理器的实现放在了IndexMgr命名空间下面。

3.4.1 IndexMgr的类组织

IndexMgr在API命名空间下包括五个类：

IndexContainer定义了一个接口，以用来控制一个索引。

IndexScan 实现了在指定的索引上的向前扫描，该扫描不断获取索引上的下一个值，直到索引的末尾。

IndexException定义了在IndexMgr下所有的异常。

UniqueConstraintViolationException 是为保证索引的唯一性约束而专门定义的异常类。

IndexManager 定义了一组方法，支持创建一个新的索引，并支持在已存在的索引上获取实例。

在IMPL命名空间下仅包含一个BTreeIndexManagerImpl类，实现了支持并发访问的B-link树索引。

3.4.2 BTreeIndexManage

1、 IndexItem

IndexItem表示在一个B-link树页中的一项。所有的索引结点和树叶结点都是包含着若干个IndexItem，但是在索引结点和在树叶结点的IndexItem的内容会有所不同。在索引结点，一个索引项包含一个键值 key，一个指向孩子结点的指针 childPageNumber，以及一个定位参数 location。在树叶结点中，一个索引项只包含一个键值key和一个定位参数location。

IndexItem继承自PartialIndexItem。PartialIndexItem类定义了轻量级的 IndexItem，在那些只需要检测索引项的孩子指针的情况下，PartialIndex

Item 能够有效地节省耗费。同时，PartialIndexItem 包含了叶结点标记 isLeaf，用来区分索引结点和叶结点。索引项的类图如 3-24 所示。

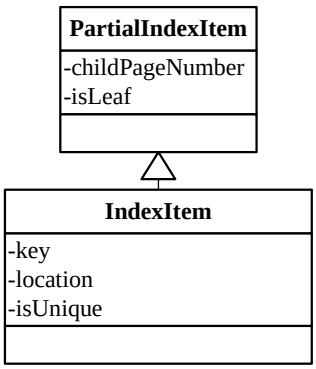

图 3-24　索引项类图

2、 BTreeNode

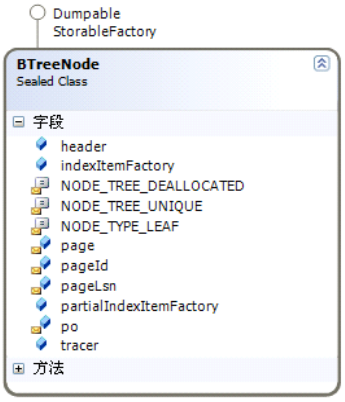

图 3-25　BTreeNode 类关系图

BTreeNode 定义了基本的 B-link 树的结点。

BTreeNode 的类关系图如 3-25 所示。

```
        static readonly int SIZE = TypeSize.INTEGER * 5;
        public int leftSibling = -1;
        public int rightSibling = -1;
        public int keyCount = 0;
        public int keyFactoryType = -1;
        public int locationFactoryType = -1;
```

图 3-26　BTreeNodeHeader 成员变量定义部分代码

在每一个 B-link 树的结点中，都包含一个 header，header 中包含了该结点的一些重要信息。系统用一个 BTreeNodeHeader 类来实现了 header，并且向外

提供接口对 header 中的信息进行访问。

图 3-26 所示的代码介绍了 BTreeNodeHeader 中所包含的信息。

3、 BTreeContext

BTreeContext 类定义了 B-link 树结点的上下文，在 B-link 树的各个操作中广泛地应用。

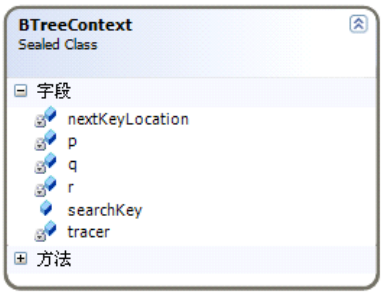

图 3-27　BTreeContext 类关系图

BTreeContext 的实例在方法中当做游标来使用，因为一个 BTreeContext 的实例包含了一个指向父结点的指针 p，一个指向当前结点的指针 q 和指向当前结点的右兄弟的指针 r。通过这几个指针的操作，可以完成对 B-link 树结点的遍历，如图 3-27 所示。

3.4.3 BTreeIndexManager 的操作类

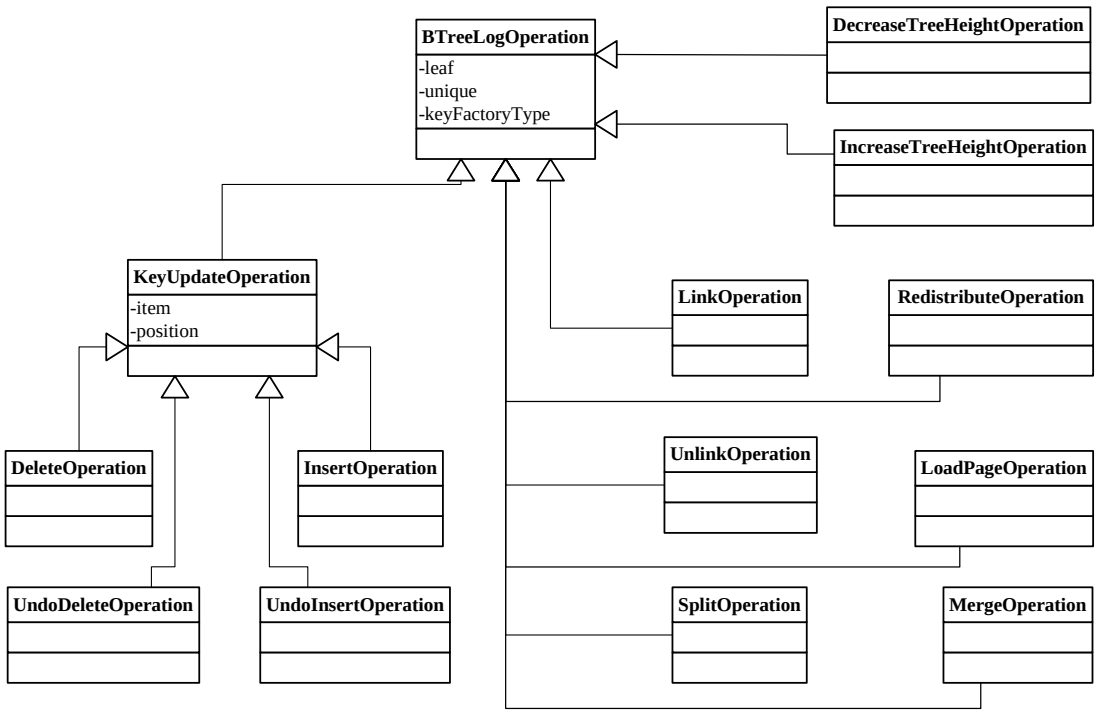

图 3-28　BTreeIndexManagerImpl 操作类类图

存储管理器实现的是支持事务处理的 B-link 树，因而，为了便于事务处理，在实现中将每一个操作都设置为一个类，在类中包含了一些附加信息，实现了对事务处理的支持。

BTreeLogOperation 是所有操作类的基类，BTreeIndexManagerImpl 下所有的操作类均继承自 BTreeLogOperation。

图 3-28 显示的是 BTreeIndexManagerImpl 中所有操作类的类图。

3.4.4 BTreeImpl 的关键方法实现

在 BTreeIndexManagerImpl 下的 BTreeImpl 实现了 B-link 树。以下介绍的是 B-link 树的重要操作的代码实现。

1、 查找

BTreeIndexManagerImpl 中将查找[3]操作分作两个阶段来实现：一是在索引结点上的遍历，二是在叶结点上的查找。

这两个阶段分别对应的是 readModeTraverse 和 doSearchAtLeafLevel。其中，readModeTraverse 返回的是包含搜索值的叶结点，doSearchAtLeafLevel 则是在叶结点中继续搜索，最后将查找的结果返回。

2、 插入

Insert 方法执行的流程如图 3-29 所示。

输入参数后，程序首先判断插入值是否合法，然后创建检查点。接下来调用 doInsert 方法，执行 B-link 树上的插入操作。插入成功，则返回，否则，为了保证数据的一致性，程序会执行事务回滚操作。

由流程图可知，在 B-link 树上真正的插入[3]操作在 doInsert 方法中执行，insert 方法只是在 doInsert 之外包装了事务的回滚操作。

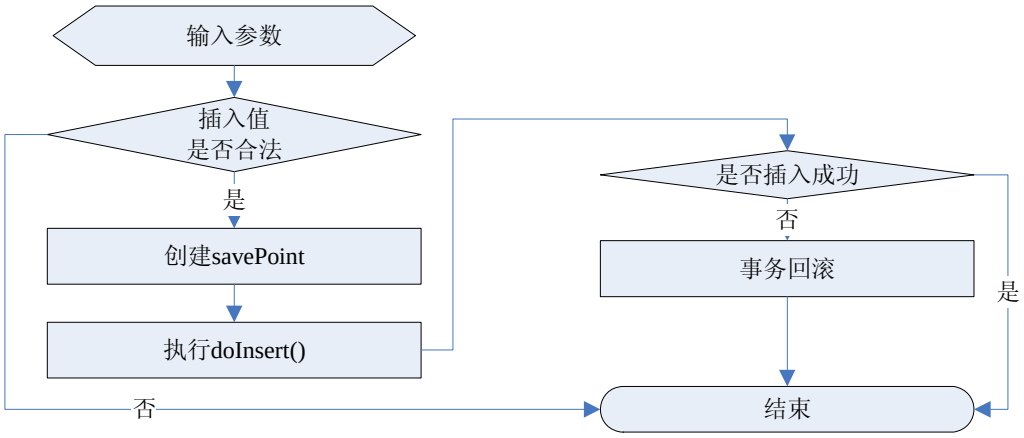

图 3-29　insert 方法流程图

doInsert 方法的伪代码如图 3-30 所示：

```
Insert(T, k, x) {
        update-mode-traverse(k, P);
        if (P is full) split(P);
        upgrade-latch(P);
        P` = the Page-id of the page that holds the record r` with
        the least key value greater than k;
        /* thus P` is P (when r` is found in P) or */
        /* the page next to P (otherwise) */
        X-latch(P`); lock-records(T, P, r, P`, r`);
        if (the exception "locks cannot be granted" is returned) {
                restart the insert operation;
        }
        search P for the position of insertion;
        if (a record with key value k is found) {
                terminate the insert operation; release the latches;
                return with the exception "uniqueness violation";
        } else {
                insert r into P;
                log(n, <T, insert, P, (k, x), Last-LSN(T)>);
                Page-LSN(P)= n; Last-LSN(T)= n;
                unlatch(P); unlatch(P_); unlock(r_);
                hold the X lock on r for commit duration;
        }
}
```

图 3-30　doInsert 方法伪代码

3、 删除

Delete 方法的实现采用了和 Insert 类似的方式，在 Delete 中包含了对 doDelete 方法的事务处理包装，而真正的删除[3]操作在 doDelete 方法中实现。

doDelete 方法执行的流程图如下：

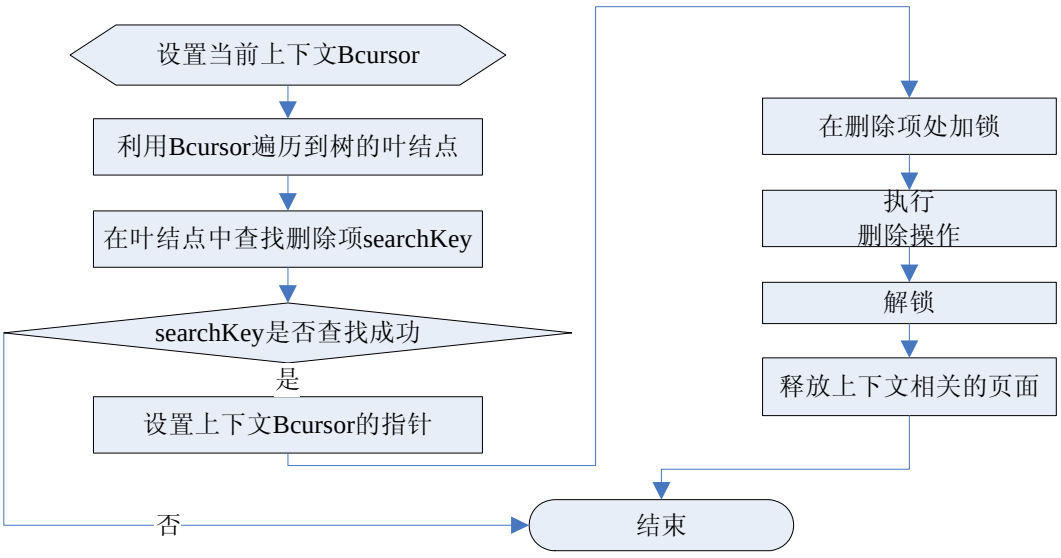

图 3-31　doDelete 方法流程图

第四章　存储管理器的测试

4.1 存储管理器测试概述

本系统在编码完成之后，使用黑盒测试的方法，对系统各个功能模块的正确性进行了检测。

存储管理器的测试的内容包括系统的对象库，以及空间管理模块、缓冲管理模块和存取方法管理模块。其中，空间管理模块的测试又根据其内部的模块划分，细化到页管理、槽页管理、空闲空间管理和存储单元管理四个子模块。

经过测试，系统的各个功能模块存在的问题均已经修复，系统能够在正确输入的前提下正常运行。

4.2 存储管理器对象库测试

由前文的设计介绍可知，存储管理器对象库的主要是以“键-值”映射的方式保存对象的信息，因此，在黑盒测试中，新建一个对象库，接着不断地以“键-值”对的形式向对象库中注册对象，然后再从对象库中用注册过的“键”获取注册对象，检测读出的对象是否与输入的对象相同。

对象库的部分测试代码如图 4-1 所示。

```
        factory.registerObjectFactory(3,
                                new MyStorable.MyStorableFactory());
        ByteBuffer buf = ByteBuffer.allocate(6);
        buf.putShort((short)3);//添加short型测试数
        buf.putInt(311566);//添加int型测试数
        buf.flip();//将数据写入磁盘
        MyStorable o = (MyStorable)factory.getInstance(buf);
        AdditionalFunctions.assert(311566 == o.value);//判断写入值
        buf = ByteBuffer.allocate(6);
        buf.putShort((short)3); //添加short型测试数
        buf.putInt(322575); //添加int型测试数
        buf.flip();//将数据写入磁盘
        o = (MyStorable)factory.getInstance(buf);
        AdditionalFunctions.assert(322575 == o.value); //判断写入值
```

图 4-1　ObjectRegistry 测试代码

利用上述的测试思路，本文完成了 TStore 存储管理器对象库的测试，对象库的功能实现正确，达到了设计预期目标。

4.3 空间管理模块测试

空间管理模块在实现的过程中分为页管理、槽页管理、空闲空间管理和存储单元管理四个子模块，在测试过程中，同样将测试工作分成四个方面。

4.3.1 页管理子模块测试

页管理的主要任务是为数据分配存储空间，保存数据。在测试过程中，测试逻辑会向系统申请存储空间，然后将测试数据写入指定的空间，最后检测从存储空间读取的数据是否与测试数据一致。

经过测试，更正了编码过程中发现的问题，页管理器达到了设计的目标，能够正常地运行，其测试结果如图 4-2 所示。

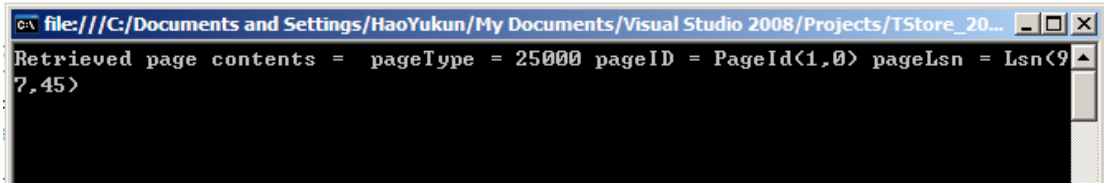

图 4-2　页管理子模块测试结果图

4.3.2 槽页管理子模块测试

槽页最大的特点在于页面内划分为若干个槽，并且支持向指定的槽中写入数据。为了检测槽页管理能否将数据写入指定的位置，以及系统在页面空间不足情况下的处理措施，设计如 4-3 所示的测试代码。

```
StringItem item2 = new StringItem("Dibyendu Majumdar, This is pretty foolish");
page.insertAt(1, item2, false);
assertEquals(page.getFreeSpace() == (page.getSpace() -
                  (item.getStoredLength() + item2.getStoredLength() +
                  SlottedPageImpl.Slot.SIZE * 2)));
assertEquals(page.getNumberOfSlots() == 2);
assertEquals(page.getDeletedSlots() == 0);
              ……
StringItem item4 = new StringItem("This is a long item, and should force the
                   page to be compacted - by removing holes");
page.insertAt(3, item4, false);
assertEquals(page.getFreeSpace() == (page.getSpace() -
                  (item2.getStoredLength() + item3.getStoredLength() +
                  item4.getStoredLength() + SlottedPageImpl.Slot.SIZE * 4)));
assertEquals(page.getNumberOfSlots() == 4);
assertEquals(page.getDeletedSlots() == 1);
```

图 4-3　槽页管理子模块测试代码示例

经过测试，槽页管理子模块能够按照设计预想输出指定的结果。

4.3.3 空闲空间管理子模块测试

根据空闲空间管理的实现，需要测试的主要是两方面：

1、测试其能否正常地创建、扩展、删除存储容器。

2、测试其维护的空间映射页能够正常地工作。

基于前面提到的思路，设计了图 4-4 所示的测试类。经测试，空闲空间管理

器运行正常，达到了设计的目的。

4.3.4 存储单元管理子模块测试

存储单元管理负责创建基本的存储单元实例，是存储管理器中最为基础却又最为重要的模块。

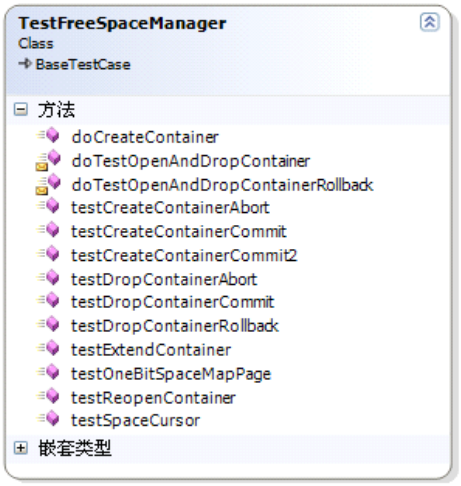

图 4-4　空闲空间管理子模块测试类类关系图

基于存储单元管理子模块的设计思路，在测试中主要检测的是存储单元的创建和向存储单元写入数据两项功能。

测试中的存储单元创建结果如图 4-5。

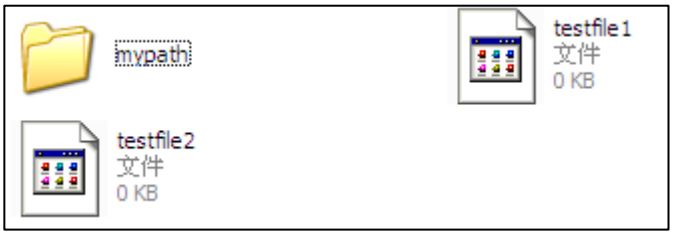

图 4-5　存储单元管理子模块部分测试结果

经过测试，存储单元管理器能够实现预期的结果，测试通过。

4.4 缓冲管理模块测试

缓冲管理模块是存储管理器的核心模块，其主要的功能是将系统频繁访问的页面暂存在内存中，有效减少系统的磁盘读写次数，提高系统的性能。

根据缓冲管理器的工作特性，测试的重点是缓冲帧的读写，以及在多线程同时访问缓冲池的情况下，测试系统的执行结果。

经过测试，缓冲管理器能够在单线程的条件下很好地完成读写操作，并且

能够在多线程的情况下，保持系统的正确运行。

测试结果如图 4-6 所示。

```
BufferManager Statistics:
bufferpoolsize = 3, dirtybuffers = 0, fixcount = 6021, cachehits = 4598, writers
leepinterval = 5000, hashtablesize = 53
: Validated that value of i in page PageId(1,1) is 3000
BufferManager Statistics:
bufferpoolsize = 3, dirtybuffers = 0, fixcount = 9023, cachehits = 7599, writers
leepinterval = 5000, hashtablesize = 53
T1 (1) Locking page in UPDATE mode
T2 (2) Trying to obtain shared latch on page
T2 (3) Obtained shared latch on page
T1 (4) Upgrading lock to Exclusive
T2 (5) Releasing shared latch on page
T1 (6) Lock upgraded to Exclusive
T1 (7) Downgrading lock to Update
T1 (8) Releasing Update latch on page
BufferManager Statistics:
bufferpoolsize = 3, dirtybuffers = 0, fixcount = 9026, cachehits = 7601, writers
leepinterval = 5000, hashtablesize = 53
```

图 4-6　缓冲管理模块部分测试结果

4.5 存取方法管理模块测试

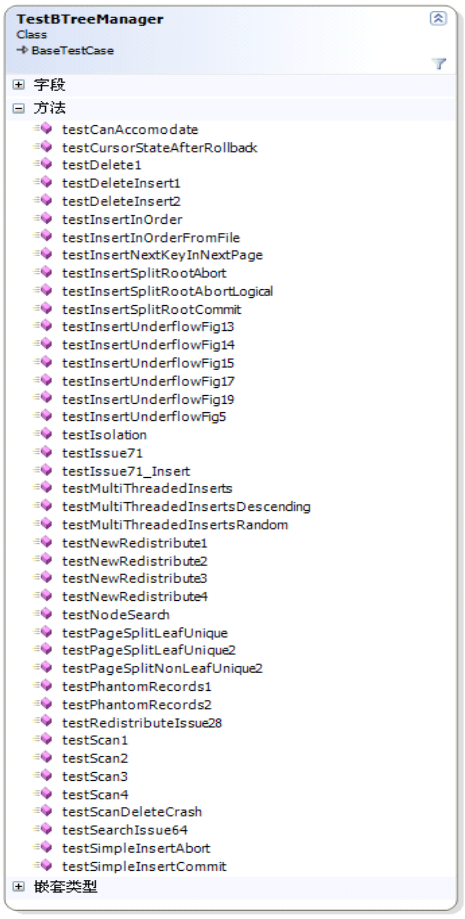

图 4-7　B-link 树测试类的类关系图

论文实现的存取方法管理模块即索引管理模块，并且在索引管理模块下，实现了一种支持事务处理的 B-link 树索引结构。

B-link 树索引结构包含若干重要的数据结构，并且，考虑到该索引结构的各项操作需要支持事务处理，因而，B-link 树的各项操作成为测试的重点。

B-link 树测试类的类关系图如图 4-7。

经过测试，TStore 系统实现的 B-link 树索引结构，能够在正确导入数据的情况下正确完成设计预期的功能。

第五章　总结与展望

TStore 存储引擎在传统的存储引擎的基础之上，增加了对事务处理的支持，并保证了一定规模的并发度。本文实现的存储管理器，虽然并非是事务处理的直接逻辑单元，但是从底层实现上，提供了对事务处理的支持，并且保证了良好的存储管理性能。论文在构建存储管理器的过程中，做了以下工作：

第一，改进了缓冲管理LRU算法。该系统在实现缓冲管理模块的过程中，考虑到了在瞬间大量临时页面扫描过程中，页面失效过快而导致缓冲池频繁替换，引起系统性能下降的特殊情况，采用了在页面上增加hint标记的做法，改进了传统的LRU缓冲管理调度算法。

第二，实现了支持并发访问的 B-link 树索引结构。该系统基于 Ibrahim Jaluta等人的算法描述，实现了一个支持并发访问的B-link树索引结构。该系统实现的B-link树索引结构，能够在任何情况下都能保证树的平衡，并且它支持在该索引结构上同时执行任意数目的操作。

第三，面向对象的程序设计。程序的主体部分使用 C#语言编写，在整个程序中所有的功能模块都用类进行封装，并且在运行时各个模块以对象的形式存在。通过面向对象的程序设计，提高了程序源码的可读性和重用性。

第四，大规模的项目编程。该系统实现了一个支持事务处理的存储引擎，实现部分的源代码达到两万余行，测试部分的源代码达到一万余行，设计的类和接口共计两百余个。

本文实现的存储管理器目前还只是一个简单的原型，还有很多的地方需要改善，比如：系统仅仅实现了一种B-link树索引结构，对于目前被广泛使用的哈希表还没有支持，存取方法的实现过于单一等等，这些都需要在以后的工作中进一步改进和完善。

参考文献

[1]维基百科．Storage Engine[EB/OL]．http://en.wikipedia.org/wiki/Storage_engine．2008．

[2]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．

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

[4]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．

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

[6]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．

[7]Hong-Tai Chou，David J．DeWitt．An Evaluation of Buffer Management Strategies for Relational Database Systems[C]．Proc．11th International Conference on Very Large Data Bases．Stockholm，1985：127－141．

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

[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]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．

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

[12]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．

[13]格雷(Gray，J)等．事务处理：概念与技术[M]．北京：机械工业出版社，2003．504－530．

[14]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．

[15]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．

[16]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：1994．131－144．

[17] Rakesh Agrawal, Anastasia Ailamaki et al．The Claremont Report on Database Research [C]．ACM SIGMOD Record．Berkeley，2008，37(3)：8－19．

致　谢

本文的写作过程中得到了很多老师、同学的帮助和指导。

感谢张坤龙老师，在张老师的一步步指导下，我完成了毕业设计的程序实现和论文的写作。跟随张老师的两年里，他在做学问和做人方面给予我莫大的启迪和指引。得益于张老师的长期培养，我有幸走进数据库管理系统的科研领域，体验到科学的魅力和做研究的快乐。

感谢谭龙飞同学，他与我共同完成了整个TStore存储引擎。我们一起讨论系统实现中遇到的问题，他经常能在关键时刻给予我启发。

感谢实验室的吴宗远和孙博师兄，在我的论文写作阶段，他们给予了我很多论文写作的建议，让我少走了不少弯路。

感谢我的挚友王旭、隋扬同学，在接近半年的毕业设计期间，是他们的关心和帮助，让我走出了情绪的低谷，顺利完成了毕业设计的各项任务。

再次对所有帮助过我的人表示衷心的感谢！