校园项目

EasySQL 技术文档

EasySQL Technical documents

数据库的简易实现。内容包括
数据的可靠性和数据恢复
两段锁协议（2PL）实现可串行化调度
MVCC
两种事务隔离级别（读提交和可重复读）
死锁处理
简单的表和字段管理
SQL 解析
基于 socket 的 server 和 client

目　录

第一章 项目整体介绍

1.1 项目来源

EasySQL 根据 Githhub 上的 NYADB2 项目改编而来，NYADB2 项目为 Golang 语言实现的简易的数据库，代码可读性强。为进一步理解数据库，提升编程技巧与逻辑思考能力，因此将其改编成了一个 Java 版数据库。

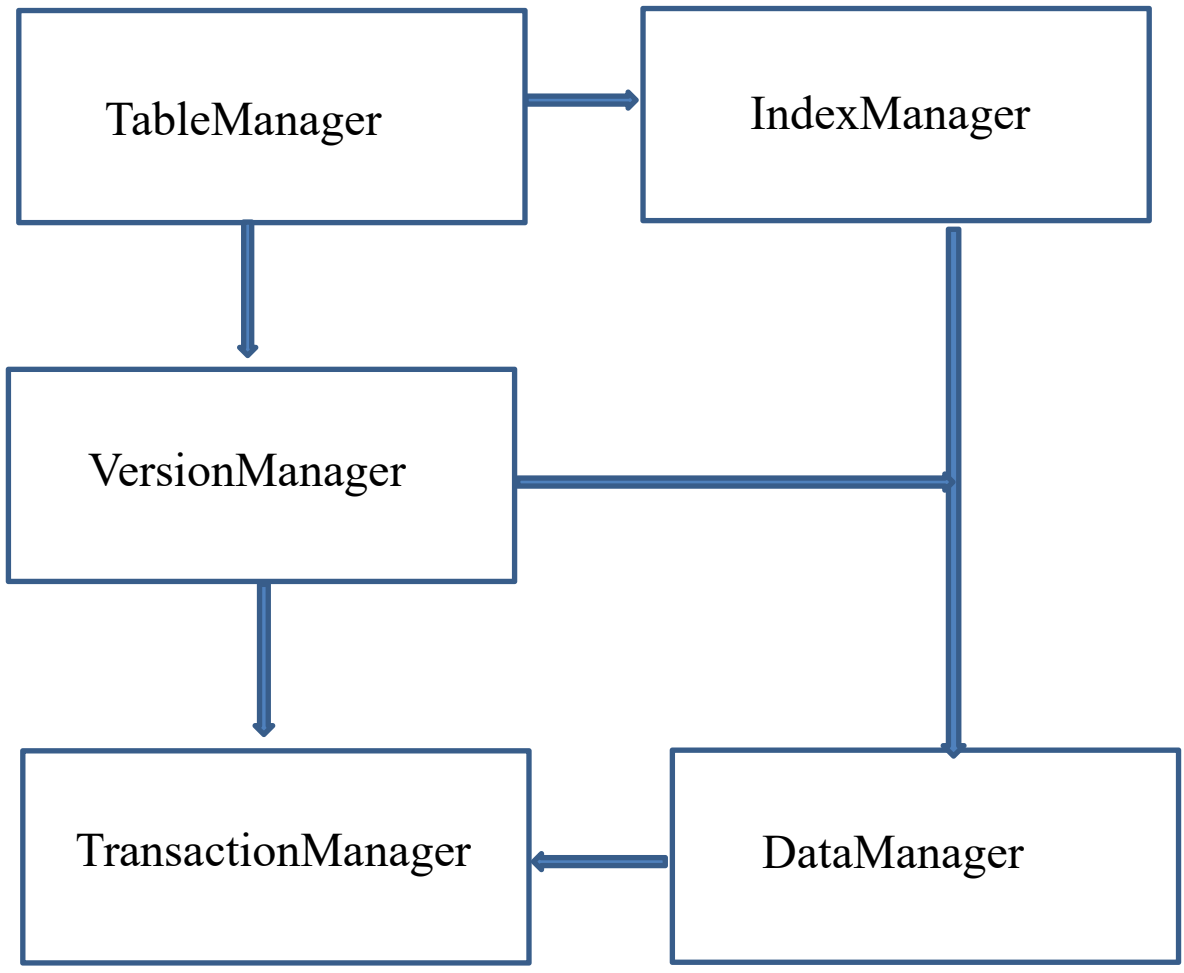
1.2 项目架构

1.1.1 整体架构

整体分为客户端与服务器端两部分。客户端与服务器采用 Socket 交互。客户端的职责很简单，读取用户输入并发送到服务器端，服务器端解析用户输入 SQL 语句，返回结果给用户端。

1.1.2 服务器端架构

1.1.2.1 模块依赖图



1.1.2.2 模块简介

TranactionManager 通过维护 XID 文件来维护事务的状态，并提供接口供其他模块来查询某个事务的状态。

DataManager 直接管理数据库 DB 文件和日志文件。DM 的主要职责有：1）分页管理 DB 文件，并进行缓存；2）管理日志文件，保证在发生错误时可以根据日志进行恢复；抽象 DB 文件数据项为 DataItem 供上层模块使用，并提供缓存。

VersionManager 基于两段锁协议实现了调度序列的可串行化，提供死锁检测功能，并实现了 MVCC 以消除读写阻塞。同时实现了两种隔离级别。

IndexManager 实现了基于 B+ 树的索引，BTW，目前 where 只支持已索引字段。

TableManager 实现了对字段和表的管理。同时，解析 SQL 语句，并根据语句操作表。

第二章　事务管理模块

2.1 功能介绍

本模块通过对 XID 文件的操作维护事务状态，对其他模块提供事务状态查询，事务状态修改功能。

2.2 XID 文件定义

XID 文件记录事务的状态。其中，用文件开头八个字节表示当前数据库记录事务数目。之后每个事务在文件中占用一个字节。因此，第 i 个事务状态在 XID 文件中 i-1+8 位置记录。0 号事务不记录，因为他被定义为总是已提交状态。

2.3 事务状态定义

每个事务有三个状态：

1. active 处于该状态下表示该事务处于活跃状态，状态码为 0。
2. commited 处于该状态下表示该事务处于已被提交状态，状态码为 1。
3. abort 处于该状态下表示该事务处于已被提交状态，状态码为 2。

2.4 TranactionManager

TransactionManager 定义该模块提供的功能：

```
public interface TransactionManager {
    public long begin();//开启新事务

    public void commit(long xid);//提交事务

    public void abort(long xid);//取消事务

    public boolean isActive(long xid);//事务是否进行态

    public boolean isCommitted(long xid);//事务是否已提交

    public boolean isAbort(long xid);//事务是否已取消

    public void close();//关闭事务
```

```
@    public static TransactionManager create(String path) {...}

@    public static TransactionManager open(String path) {...}
}
```

定义事务文件创建与打开两个方式。本项目文件操作采用 JavaNIO API，通过 FileChannel 进行读写操作。

创建具体实现类中，会对 XID 文件进行一次检查。当文件大小小于 8 字节或文件大小与文件描述事务数目不符，结束系统。

```
public class TransactionManagerImpl implements TransactionManager {
    private RandomAccessFile file;
    private FileChannel fc;
    private ReentrantLock lock = new ReentrantLock();
    private long XIDCounter;
    public TransactionManagerImpl(RandomAccessFile frw, FileChannel fc)
    private void checkXIDFile() {...}

    private long getXIDPosition(long xid) { return XID_HEADER_LENGTH + 
    private void updateXID(long xid, byte status) {...}
    @Override
    public long begin() {...}
    private boolean checkXIDStatus(long xid,byte status) {...}
    private void updateXIDFileHeader() {...}
    @Override
    public void commit(long xid) { updateXID(xid,XID_COMMITED_STATUS);

    @Override
    public void abort(long xid) { updateXID(xid,XID_ABORT_STATUS); }

    @Override
    public boolean isActive(long xid) { return checkXIDStatus(xid, XID_

    @Override
    public boolean isCommitted(long xid) { return checkXIDStatus(xid, X

    @Override
    public boolean isAbort(long xid) { return checkXIDStatus(xid, XID_A

    @Override
    public void close() {...}
}
```

事务管理类创建时根据 XID 文件记录当前最大事务 id。在每个事务开启时对其加锁，保证每个事务唯一性。对事务状态更新后，用 force 强制刷回 XID 文件，保证后续版本控制等模块使用。

第三章　数据管理模块

3.1 引用计数法缓存框架

EasySQL 采用引用计数法对页，数据项等数据进行缓存。通用引用计数缓存框架如下：

```
public abstract class BaseCache<T> {
    private HashMap<Long,T> cache;
    private HashMap<Long,Boolean> isGetting;
    private HashMap<Long,Integer> reference;
    private final long maxResource;
    private AtomicLong resourcesCount=new AtomicLong( initialValue: 0);
    private Lock lock;
//     根据key获取实际资源操作
    protected abstract T getByKeyOfCache(long key) throws Exception;
//     释放缓存,同时写回资源
    protected abstract void releaseByKeyOfCache(T key);

    public BaseCache(long maxResource) {...}
//     根据key从缓存中获取资源
    public T get(long key) {...}
//     从缓存中释放资源 若释放后引用数为0,将其写回
    public void release(long key) {...}
//关闭缓存   写回所有资源
    public void close() {...}
}
```

提供 get，release，close 方法。由具体实现模块实现从真实资源读入缓存或刷回操作。

get 方法逻辑为上锁，判断是否有其他线程在获取资源。若有，释放锁等待 1ms 并自旋，若没有则查看缓存是否有该资源，若有返回该资源并释放锁，没有则将该资源标记为获取中并释放锁。之后调用 getByKeyOfCache 获取实际资源，并将该资源移除获取中标记。放入缓存，资源数加一，并将引用计数加一。若失败，移除获取中标记。

release 方法中判断释放本次引用后该资源是否引用数为 0，若是将其刷回。Close 方法为将 cache 中所有数据刷回，并从缓存中移除。

3.2 页面缓存

3.2.1 页面抽象

EasySQL 数据相关信息保存在.db 文件中。对.db 文件采用分页管理，默认每页大小为 8k。将每页抽象为一个 Java 类，即 Page 类：

```
public class PageImpl implements Page {

//    页号
    private int pageNumber;
//    该页数据
    private byte[] data;
//    该页是否为脏页
    private boolean isDirty;
    //    pageCache
    private PageCache pageCache;
    private Lock lock;
    public PageImpl(int pageNumber, byte[] data, PageCache pageCache) {...}
//    设置该页为脏页
    @Override
    public void setDirty(boolean dirty) { isDirty = true; }
//    判断是否脏页
    @Override
    public boolean isDirty() { return isDirty; }
//  获取页号
    @Override
    public int getPageNumber() { return pageNumber; }
//获取该页数据
    @Override
    public byte[] getData() { return data; }
//释放该页
    @Override
    public void release() { pageCache.release( page: this); }
}
```

对于 Page 的 release 方法，当缓存中的该页为脏页才会被刷回，达到节省资源的目的。该逻辑在 PageCache 中实现。

普通的 Page 页用该页起始两个字节表示该页可用空间偏移量。在 Page 工具类 XPage 中定义了对 Page 数据获取偏移，插入数据等操作。

```java
public class XPage {
    public static byte[] initRaw() {...}

    public static void setFreeSpaceOffset(byte[] raw, short offset) {...}
//    获取Page偏移
    public static short getFreeSpaceOffset(Page pg) { return getFreeSpaceOffset(pg.getData()); }

    private static short getFreeSpaceOffset(byte[] data) { return Parser.parseShort(Arrays.copyOfRange(data, from: 0, to: 2)); }

    // 将raw插入pg中， 返回插入位置
    public static short insert(Page pg, byte[] raw) {...}
    // 获取页面的空闲空间大小
    public static int getFreeSpace(Page pg) { return PAGE_SIZE - (int)getFreeSpaceOffset(pg.getData()); }
    // 将raw插入pg中的offset位置，并将pg的offset设置为较大的offset
    public static void recoverInsert(Page pg, byte[] raw, short offset) {...}
    // 将raw插入pg中的offset位置，不更新update
    public static void recoverUpdate(Page pg, byte[] raw, short offset) {...}
}
```

获取某页偏移只需解析该页前两个字节。若进行了页面插入修改等操作，会将页面设为脏页。Insert 方法定义了在某页面插入操作：先设置该页为脏页，获取该页偏移量并将数据覆盖并修改偏移量。RecoverInsert 方法定义了数据恢复时数据重新插入操作。该操作首先设置该页为脏页，并将数据覆盖到对应位置上，视情况修改偏移量。RecoverUpdate 方法操作与 RecoverInsert 类似。值得注意的时，一条数据不允许横跨多页的情况，这种逻辑在 PageIndex 页面管理中实现。

3.2.2 页面缓存实现

```java
public class PageCacheImpl extends BaseCache<Page> implements PageCache {
    private final RandomAccessFile file;
    private final FileChannel fc;
    private final int maxResource;
//    缓存中的总页数
    private AtomicInteger pageNumbers;
    private final Lock lock;
    public PageCacheImpl(RandomAccessFile file, FileChannel fc, int maxResource) {...}

    private void flush(Page page) {...}
    public void truncateByBgno(int maxPgno) {...}
    private long getPageOffset(int pageNo) { return (pageNo - 1) * PAGE_SIZE; }
    @Override
    public int newPage(byte[] pageData) {...}

    @Override
    public Page getPage(int pageNo) { return get(pageNo); }

//    根据页号获取Page
    @Override
    protected Page getByKeyOfCache(long key) {...}

    @Override
    protected void releaseByKeyOfCache(Page page) {...}

    @Override
    public void close() {...}

    @Override
    public void release(Page page) { release(page.getPageNumber()); }

    @Override
    public int getMaxPageNo() { return pageNumbers.intValue(); }
```

getByKeyOfCache 方法为从磁盘中读取一页的具体操作。通过获取页号在.db 文件中偏移，通过加锁保证将其安全读入缓存。

flush 方法为将页刷回磁盘操作。该操作通过获取要刷回操作页号获取其在.db 文件中的偏移量，通过加锁保证安全性将其强制刷盘。该方法与 releaseByKeyOfCache 配合，当某页引用计数为 0 的时候，判断该页是不是脏页，若是则调用 flush 方法将其刷回磁盘。

对于每一页，页号就是其在缓存中的 key。

3.2.3 首页功能

EasySQL 中，首页是特殊的一页，该页不写入业务数据项，实现了判断数据库是否正常关闭功能。通过在 100-107，108-115 两段位置加入校验码方式判断是否数据库是否被正常关闭。当数据管理模块启动时，会在 100-107 位置加入随机的 8 字节校验码，在该模块被正常关闭时会在 108-115 位置复制该校验码。这样，判断数据库有没有被正常关闭只需校验这两段校验码是否一致。若不一致，将采用

日志进行数据恢复。

3.3 日志文件与恢复策略

EasySQL 在数据库非正常关闭情况下，再次启动时可以根据.log 日志文件恢复数据。

3.3.1 日志文件格式

[XCheckSum][log1][log2][...][][logN]([BadTail])

.log 文件前四位 XCheckSum 为总校验和，是对后续所有日志数据部分计算的校验和。BadTail 为在非正常关闭下某条可能没来得及写完的日志，通过该条日志校验和来判断。校验和参考了字符串 Hash 算法，Hash 种子为 1331，计算时当前位校验和为上一位校验和*Seed+当前位数。每条日志格式：

[Size][CheckSum][Data]

Size 占用四字节表示 Data 段字节数。CheckSum 为该条日志数据部分的校验和。日志功能实现如下：

```
public class LoggerImpl implements Logger {
    private RandomAccessFile file;
    private FileChannel fc;
    private Lock lock;
    private long fileSize;
    private int xCheckSum;
//  逐条解析日志的指针
    private long logPosition;
    public LoggerImpl(RandomAccessFile file, FileChannel fc, int xCheckSum) {...}
    public LoggerImpl(RandomAccessFile file, FileChannel fc) {...}
//重新打开日志文件时初始化日志尾指针，检查总校验和，检查BadTail 并将其移除
    public void init() {...}
    // 检查并移除bad tail
    private void checkAndRemoveTail() {...}
//    逐条解析并返回日志数据 在BadTail处返回null
    private byte[] internNext() {...}
//    校验和计算
    private int calChecksum(int checkValue, byte[] logData) {...}
//将某条数据封装为日志格式插入日志文件,并更新总校验和
    @Override
    public void log(byte[] data) {...}
//解析出日志数据部分
    @Override
    public byte[] next() {...}
//    更新总校验和
    private void updateXCheckSum(byte[] logData) {...}
//  包装数据为日志格式
    private byte[] wrapLog(byte[] data) {...}
//    截断Badtail
    @Override
    public void truncate(long x) {...}
    @Override
    public void rewind() { logPosition=4; }
    @Override
    public void close() {...}
```

3.3.2 恢复策略

EasySQL 通过版本控制模块控制了事务之间的隔离性。正在进行的事务，不会读取到未提交事务产生的数据，也不会修改任何未提交事务修改或产生的数据。因此恢复策略为重做所有已结束事务（comitted or abort），撤销未完成的事务(active)。对于每条日志记录的数据部分，规定其数据格式：

[LogType][Xid][PageNo][PageOffset][(OldData)Data]

其中，LogType 用一字节标记了该日志记录的操作为插入或是更新操作。LogType=0 表示插入操作，LogType=1 表示更新操作。对于更新操作，会在日志记录中保存该操作原本的数据 OldData。Xid 表示该日志记录的产生事务，PageNo 表示该数据在.db 中的页号，PageOffset 表示数据在该页中的偏移量。

当首页验证错误时，执行恢复策略。两种日志类型类定义：

```
static class InsertLogInfo {
    long xid;
    int pgno;
    short offset;
    byte[] raw;
}

static class UpdateLogInfo {
    long xid;
    int pgno;
    short offset;
    byte[] oldRaw;
    byte[] newRaw;
}
```

恢复策略：从前往后逐条解析出日志，若该日志记录操作事务为已结束事务，将其重做。对于插入操作，将数据重新插到对应位置。对于更新操作，将新数据插入到对于位置。然后从后往前解析日志，对于未结束事务，撤销其操作。具体逻辑为对于插入操作，将数据项设为不可见后插到对应位置，数据项的具体格式会在 3.5 节提及。对于更新操作，将旧数据写到对应位置。

```java
private static void undoTransactions(TransactionManager tm, Logger log, PageCache pageCache) {
    Map<Long, List<byte[]>> logCache = new HashMap<>();
    log.rewind();
    while(true) {
        byte[] logData = log.next();
        if(logData == null) break;
        if(isInsertLog(logData)) {
            InsertLogInfo li = parseInsertLog(logData);
            long xid = li.xid;
            if(tm.isActive(xid)) {
                if(!logCache.containsKey(xid)) {
                    logCache.put(xid, new ArrayList<>());
                }
                logCache.get(xid).add(logData);
            }
        } else {
            UpdateLogInfo xi = parseUpdateLog(logData);
            long xid = xi.xid;
            if(tm.isActive(xid)) {
                if(!logCache.containsKey(xid)) {
                    logCache.put(xid, new ArrayList<>());
                }
                logCache.get(xid).add(logData);
            }
        }
    }

    // 对所有active log进行倒序undo
    for(Map.Entry<Long, List<byte[]>> entry : logCache.entrySet()) {
        List<byte[]> logs = entry.getValue();
        for (int i = logs.size()-1; i >= 0; i --) {
            byte[] logData = logs.get(i);
            if(isInsertLog(logData)) {
                doInsertLog(pageCache, logData, UNDO_FLAG);
            } else {
                doUpdateLog(pageCache, logData, UNDO_FLAG);
            }
        }
        tm.abort(entry.getKey());
    }
}
```

Undo 逻辑

```java
private static void redoTranscations(TransactionManager tm, Logger logger, PageCache pageCache) {
    logger.rewind();
    while(true) {
        byte[] log = logger.next();
        if(log == null) break;
        if(isInsertLog(log)) {
            InsertLogInfo li = parseInsertLog(log);
            long xid = li.xid;
            if(!tm.isActive(xid)) {
                doInsertLog(pageCache, log, REDO_FLAG);
            }
        } else {
            UpdateLogInfo xi = parseUpdateLog(log);
            long xid = xi.xid;
            if(!tm.isActive(xid)) {
                doUpdateLog(pageCache, log, REDO_FLAG);
            }
        }
    }

}
```

Redo 事务逻辑

```java
private static void doUpdateLog(PageCache pageCache, byte[] log, int flag) {
    int pgno;
    short offset;
    byte[] raw;
    if(flag == REDO_FLAG) {
        UpdateLogInfo xi = parseUpdateLog(log);
        pgno = xi.pgno;
        offset = xi.offset;
        raw = xi.newRaw;
    } else {
        UpdateLogInfo xi = parseUpdateLog(log);
        pgno = xi.pgno;
        offset = xi.offset;
        raw = xi.oldRaw;
    }
    Page pg = null;
    try {
        pg = pageCache.getPage(pgno);
    } catch (Exception e) {
        Exit.systemExit(e);
    }
    try {
        XPage.recoverUpdate(pg, raw, offset);
    } finally {
        pg.release();
    }
}
```

恢复策略更新日志对应逻辑

```java
private static void doInsertLog(PageCache pageCache, byte[] log, int flag) {
    InsertLogInfo li = parseInsertLog(log);
    Page pg = null;
    try {
        pg = pageCache.getPage(li.pgno);
    } catch(Exception e) {
        Exit.systemExit(e);
    }
    try {
        if(flag == UNDO_FLAG) {
            DataItem.setDataItemRawInvalid(li.raw);
        }
        XPage.recoverInsert(pg, li.raw, li.offset);
    } finally {
        pg.release();
    }
}
```

恢复策略插入日志对应逻辑

3.4 页面管理

该子模块缓存了每一页的空闲空间，当有数据插入.db 文件时，根据本子模块提供的页面管理快速获取一个具有足够空间的页进行操作。实现思路是将每一页按剩余空间大小映射到不同的子区间并缓存起来，根据需要空间大小从页面管理模块申请对应页。默认划分到 40 个子区间中。

```java
public class PageIndex {
//  将每页按剩余空间分到40个区间里

    private Lock lock;
    private LinkedList<PageInfo>[] lists;

    public PageIndex() {
        lock = new ReentrantLock();
        lists = new LinkedList[PAGE_INTERVALS_NUMBERS+1];
        for (int i = 0; i < lists.length; i++) {
            lists[i] = new LinkedList<>();
        }
    }

    public PageInfo select(int spaceSize) {
        lock.lock();
        try {
            int nums = spaceSize / PAGE_INTERVALS_SIZE;
            if (nums < PAGE_INTERVALS_NUMBERS) {
                nums++;
            }
            while (nums <= PAGE_INTERVALS_NUMBERS) {
                if(lists[nums].size() == 0) {
                    nums ++;
                    continue;
                }
                return lists[nums].removeFirst();
            }
            return null;
        } finally {
            lock.unlock();
        }
    }
```

```
    public void add(int pgNo, int freeSpace) {
        lock.lock();
        try {
            int num = freeSpace / PAGE_INTERVALS_SIZE;
            lists[num].add(new PageInfo(pgNo, freeSpace));
        }finally {
            lock.unlock();
        }
    }
}
```

具体实现

选取一页时会将其从页面管理中移除，因此同一页不会发生并发写操作。当完成插入后需要调用 add 方法将该页还给页面管理系统。

3.5 数据项 DataItem

数据项在 EasySQL 中被抽象为 DataItem 类。其他模块传递数据项地址由 DataManager 模块获取 DataItem 对象。

规定 EasySQL 中的数据项格式：

[Valid][Size][Data]

其中，Valid 占用一个字节，Valid=0 表示该数据项有效，Valid=1 表示该数据项不可见。Size 占用两个字节，表示 Data 大小。因为 DataItem 本质为 Page 里 data 中的一段，为使各模块能直接对 DataItem 进行操作，同时起到节省内存的作用，封装了一个共享数组类，这使得其他模块可以直接操作 Page 里的 data。

```
public class SubArray {
    public byte[] raw;
    public int start;
    public int end;

    public SubArray(byte[] raw, int start, int end) {
        this.raw = raw;
        this.start = start;
        this.end = end;
    }
}
```

为了更简易表示数据项位置，采用 uid 表示数据项位置。uid 为一个 64 位 long 型值，其高 32 位表示页号，低 16 位表示偏移量。

DataItem 类的实现：

```java
public class DataItemImpl implements DataItem {
    private SubArray raw;
    private byte[] oldRaw;
    private Lock rLock;
    private Lock wLock;
    private DataManagerImpl dm;
    private long uid;
    private Page page;

    public DataItemImpl(SubArray raw, byte[] oldRaw, Page page,long uid,DataManagerImpl dm ) {...}
    @Override
    public SubArray data() { return new SubArray(raw.raw,  start: raw.start+DATAITEM_DATA_OFFSET, raw.end); }
    public boolean isValid() { return raw.raw[raw.start+DATAITEM_VALID_OFFSET] == (byte)0; }

    @Override
    public void before() {...}

    @Override
    public void unBefore() {...}

    @Override
    public void after(long xid) {...}

    @Override
    public void release() {
        dm.releaseDataItem(this);
    }

    @Override
    public void lock() { wLock.lock(); }
```

```java
    public static byte[] wrapDataItemRaw(byte[] raw) {
        byte[] valid = new byte[1];
        byte[] size = Parser.short2Byte((short) raw.length);
        return Bytes.concat(valid, size, raw);
    }
//    解析页中offset处数据项
    public static DataItem parseDataItem(Page pageNo, short offset, DataManagerImpl dm) {
        byte[] raw = pageNo.getData();
        short dataSize = Parser.parseShort(Arrays.copyOfRange(raw,  from: offset + DATAITEM_SIZE_OFFSET,  to: offset + DATAITEM_DATA_OFFSET));
        short rawSize = (short) (dataSize + DATAITEM_DATA_OFFSET);
        long uid = Parser.parseUID(pageNo.getPageNumber(), offset);
        return new DataItemImpl(new SubArray(raw, offset,  end: offset+rawSize),new byte[rawSize],pageNo,uid,dm);
    }
    public static void setDataItemRawInvalid(byte[] raw) {
        raw[DATAITEM_VALID_OFFSET] = (byte) 1;
    }
```

```
    @Override
    public void unlock() { wLock.unlock(); }

    @Override
    public void rLock() { rLock.lock(); }

    @Override
    public void rUnLock() { rLock.unlock(); }

    @Override
    public Page page() { return page; }

    @Override
    public long getUid() { return uid; }

    @Override
    public byte[] getOldRaw() { return oldRaw; }

    @Override
    public SubArray getRaw() { return raw; }
}
```

DataItem 存有 DataManager 引用，这是因为其释放需要依赖 DataManager，因为其实现了抽象缓存类，具体在 3.6 节介绍。其他模块对 DataItem 进行修改时，要首先调用 before 方法，该方法对 DataItem 加写锁，将其所在页置为脏页，并将 DataItem 复制一份备份。若在修改期间需要撤销修改调用 unBefore 方法，该方法将数据恢复为备份数据并解开写锁。修改后需要调用 after 方法，该方法将本次修改操作原来数据与修改后的数据记录到更新日志并调用日志系统的写入方法确保日志的写入。这个流程保证了 DataItem 修改的原子性，同时提供了修改过程中的回滚操作。

当其他模块使用完DataItem后,需要调用release方法,该方法让DataManager模块释放一次对 DataItem 的引用，该引用为 0 时释放一次对该 DataItem 所在页的引用。

3.6 DataManager

DataManager 是数据管理模块对外层提供 api 的类，同时也是 DataItem 的缓存。每个 DataItem 的 key 是他的 uid。

```java
public static DataManager create(String path, long mem, TransactionManager tm) {
    PageCache pageCache = PageCache.create(path, mem);
    Logger logger = Logger.create(path);

    DataManagerImpl dm = new DataManagerImpl(pageCache,logger, tm);
    dm.initFirstPage();
    return dm;
}

public static DataManager open(String path, long mem, TransactionManager tm) {
    PageCache pageCache = PageCache.open(path, mem);
    Logger logger = Logger.open(path);
    DataManagerImpl dm = new DataManagerImpl(pageCache, logger, tm);
    if(!dm.loadCheckPageOne()) {
        Recover.recover(tm, logger, pageCache);
    }
    dm.fillPageIndex();
    FirstPage.setValidCheckPage(dm.firstPage);
    dm.pageCache.flushPage(dm.firstPage);
    return dm;
}
```

DataManager 的两个创建方法。Create 方法会创建 db 文件，log 文件，同时初始化首页。Open 方法中除了加载 db 文件，log 文件外，还会对首页进行安全验证，在发现异常后启动恢复策略。

```java
public class DataManagerImpl extends BaseCache<DataItem> implements DataManager {
    TransactionManager tm;
    PageCache pageCache;
    Logger logger;
    PageIndex pageIndex;
    Page firstPage;
    public DataManagerImpl( PageCache pageCache, Logger logger,TransactionManager tm) {...}
//     根据uid获取dataItem
    @Override
    public DataItem read(long uid) throws Exception {...}
//     打开时加载第一页
    boolean loadCheckPageOne() {...}
//     创建初始化第一页
    public void initFirstPage() {...}
//     初始化pageIndex
    void fillPageIndex() {...}
//     生成update日志
    public void logDataItem(long xid, DataItem dataItem) {...}
//     从页面管理中获取页，记录插入日志，将数据写入缓存并将页还回去
    @Override
    public long insert(long xid, byte[] data) throws Exception {...}
//  从PageCache中获取对应页，从对应页及对应偏移中获取DataItem
    @Override
    protected DataItem getByKeyOfCache(long uid) {...}
//  对应页引用减一
    @Override
    protected void releaseByKeyOfCache(DataItem key) { key.page().release(); }
//     释放资源，安全关闭日志，页面缓存，首页安全关闭。
    @Override
    public void close() {...}
//     uid引用减一
    public void releaseDataItem(DataItemImpl dataItem) {
        release(dataItem.getUid());
    }
}
```

DataManager 实现

fillPageIndex 方法在 DataManager 打开时调用，遍历所有页并将他们纳入页面管理。对于 Insert 方法，该方法从页面管理中拿到足够插入数据的页，写入插入日志，并将数据记入对应页缓存，将该页交还页面管理模块。对于 getByKeyOfCache 方法，该方法通过传入的 uid，从 PageCache 中获取对应页，在该页中对应偏移处解析获取 DataItem。对于 releaseByKeyOfCache 方法，该方法在无对目标 DataItem 引用时调用，在 DataItem 所在页执行一次 PageCache 的 release 方法。对于 close 方法，在关闭日志，页面缓存的同时，会执行首页安全退出逻辑。

DataManager 对上层提供了查找与插入功能,对 DataItem 提供了更新功能，并未显式提供更新功能。这是因为数据项的更新需要依赖版本控制模块，DataManager 只需能满足版本控制模块需求设置 Xmax 即可，删除功能由版本控制模块提供，修改完 Xmax 再插入一条新版本记录即可。

第四章　版本控制模块

4.1 LockTable

若同一时间多个事务组成的序列需要对同一条记录 X 进行操作，此时就需要实现调度序列的可串行化。若某条事务想对 X 记录进行修改操作时，先要进行判断是否需要对本事务加锁以及加锁是否会产生死锁。该逻辑在 LockTable 模块实现。

LockTable 类维护了每个事务持有的记录，以及该事务等待的记录等信息。

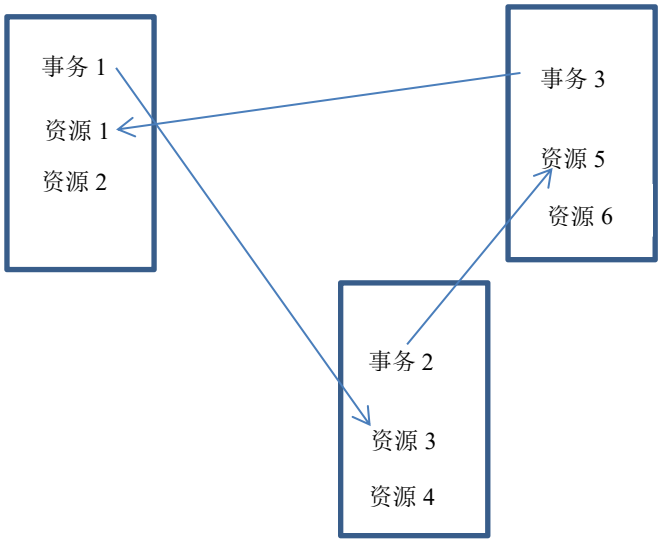
```
public class LockTable {
//    事务持有的资源记录
    private Map<Long, List<Long>> transHadUIDMap;
//    资源被事务持有的记录
    private Map<Long, Long> uidHasHadByTrans;
//    哪些事务在等待此资源
    private Map<Long, List<Long>> uidWaitedTransMap;
//    事务阻塞在哪个锁
    private Map<Long, Lock> transWaitingLockMap;
//    事务正在等待的资源
    private Map<Long, Long> transWaitingUID;
    private Lock lock;

    public LockTable() {...}
//    不需要等待返回null，否则返回锁，如果死锁抛异常
    public Lock add(long xid,long uid) throws  Exception {...}

    private void selectNewXID(long uid) {...}

    public void remove(long xid) {...}
    private void removeFromList(Map<Long, List<Long>> listMap, long uid0, long uid1) {...}
    private Map<Long, Integer> xidStamp;
    private int stamp;
    private boolean hasDeadLock() {...}
    private boolean dfs(long xid) {...}

    private void putIntoList(Map<Long, List<Long>> listMap, long xid, long uid) {...}
    private boolean isInList(Map<Long, List<Long>> listMap, long xid, long uid1) {...}

}
```

其中 add 方法实现了某事务对某资源的获取过程需不需要阻塞的逻辑。

```
public Lock add(long xid,long uid) throws  Exception {
    lock.lock();
    try {
        if (isInList(transHadUIDMap, xid, uid)) {
            return null;
        }
        if (!uidHasHadByTrans.containsKey(uid)) {
            uidHasHadByTrans.put(uid, xid);
            putIntoList(transHadUIDMap,xid,uid);
            return null;
        }
        transWaitingUID.put(xid, uid);
        putIntoList(uidWaitedTransMap,uid,xid);
        if (hasDeadLock()) {
            transWaitingUID.remove(xid);
            removeFromList(uidWaitedTransMap,uid,xid);
            throw Error.DeadlockException;
        }
        UnReentrantLock l = new UnReentrantLock();
        l.lock();
        transWaitingLockMap.put(xid, l);
        return l;
    } finally {
        lock.unlock();
    }
}
```

若当前事务已持有目标资源或目标资源未由任何事务持有，不需要阻塞直接返回。否则把当前事务正在等待获取该资源的信息到阻塞信息表中。此时进行死锁检测，若产生了死锁则将刚才事务等待信息删除。死锁检测逻辑将在后文提及。若无死锁则将当前事务上一个不可重入锁，并将其返回。之后当前事务仍需获取一次该锁并解锁才可执行对资源的修改，但因为该锁为不可重入锁，事务会阻塞等待资源被释放。

由于某时资源只会被一个事务持有，死锁检测逻辑为以某一持有资源的事务出发，dfs 资源等待图。若有死锁产生，为事务等待资源图产生了环。当然资源等待图不一定是连通图。每轮检测时对该轮遍历过的事务打上版本戳，若同一轮中某个事务产生了两次版本戳则有死锁，移除最后添加的事务等待关系即可。



死锁示意图

```java
private Map<Long, Integer> xidStamp;
private int stamp;
private boolean hasDeadLock() {
    xidStamp = new HashMap<>();
    stamp = 1;
    for(long xid : transHadUIDMap.keySet()) {
        Integer s = xidStamp.get(xid);
        if(s != null && s > 0) {
            continue;
        }
        stamp ++;
        if(dfs(xid)) {
            return true;
        }
    }
    return false;
}
```

当某事务被提交或被撤销时，需要释放该事务持有的资源。逻辑为遍历该事务持有的资源，将各个资源交给并唤醒等待各个资源的被阻塞的第一个事务，即解开该事务的不可重入锁。

```java
public void remove(long xid) {
    lock.lock();
    try {
        List<Long> l = transHadUIDMap.get(xid);
        if(l != null) {
            while(l.size() > 0) {
                Long uid = l.remove( index: 0);
                selectNewXID(uid);
            }
        }
        transWaitingUID.remove(xid);
        transHadUIDMap.remove(xid);
        transWaitingLockMap.remove(xid);
    }finally {
        lock.unlock();
    }
}
```

```java
private void selectNewXID(long uid) {
    uidHasHadByTrans.remove(uid);
    List<Long> xids = uidWaitedTransMap.get(uid);
    if (xids == null) {
        return;
    }
    assert xids.size() > 0;
    while(xids.size() > 0) {
        long xid = xids.remove( index: 0);
        if(!transWaitingLockMap.containsKey(xid)) {
            continue;
        } else {
            uidHasHadByTrans.put(uid, xid);
            Lock lo = transWaitingLockMap.remove(xid);
            transWaitingUID.remove(xid);
            lo.unlock();
            break;
        }
    }
    if (xids.size() == 0) {
        uidWaitedTransMap.remove(uid);
    }
}
```

4.2 调度序列可串行化

若某条事务想对 X 记录进行修改操作时，先要进行判断是否需要对本事务加锁

以及加锁是否会产生死锁。若当前事务没有持有该资源，会给当前事务上一个不可重入锁。当前事务就会阻塞，等到别的事务释放该资源。即以下逻辑。因此，配合 LockTable，实现了调度序列的可串行化。

```
if(l != null) {
    l.lock();
    l.unlock();
}
```

4.3 版本控制与事务隔离级别

4.3.1 Entry

通过不可重入锁调度序列可串行化会导致事务间的相互阻塞，在一定情况下会导致死锁，显然效率比较低。为了提高事务处理效率，引入了 MVCC 进行版本控制，这也能实现调度序列可串行化。事务在第三章恢复策略提到了正在进行的事务不会读取到其他未提交事务产生的数据，也不会修改其他未提交事务修改产生的数据，除了调度序列串行化外还可由 MVCC 事务隔离实现。同时，MVCC 降低了事务的阻塞概率，例如，某个事务 T1 想修改 X 的值，此时他会创建 X 的新版本，另外一个事务 T2 想读取 X 的值，就会读取旧版本 X。

DataManager 模块向上层提供了数据项 DataItem 的概念，版本控制模块 VersionManager 通过控制 DataItem 中的版本头维护多个版本记录。VersionManager 将一条记录包装为 Entry 进行版本控制。Entry 格式：

[Xmin][Xmax][Data]

对应的类定义为：

```
public class Entry {
    private long uid;
    private DataItem dataItem;
    private VersionManagerImpl vm;

    public static Entry newEntry(VersionManagerImpl vm, DataItem dataItem, long uid) {...}
    public static Entry loadEntry(VersionManagerImpl vm, long uid) throws Exception {...}

    public static byte[] wrapEntryRaw(long xid, byte[] data) {...}

    public void release() { vm.releaseEntry(this); }

    public void remove() { dataItem.release(); }

    public byte[] data() {...}

    public long getXmin() {...}

    public long getXmax() {...}

    public void setXmax(long xid) {...}

    public long getUid() { return uid; }
}
```

其中，Xmin 记录了本条记录创建事务 XID，Xmax 记录了对这条记录进行删除的事务 XID。因此，创建记录时对一条记录封装版本控制信息逻辑为：

```
public static byte[] wrapEntryRaw(long xid, byte[] data) {
    byte[] xmin = Parser.long2Byte(xid);
    byte[] xmax = new byte[8];
    return Bytes.concat(xmin, xmax, data);
}
```

通过 data 方法获取 Entry 中的数据，按 Entry 格式解析出数据：

```
public byte[] data() {
    dataItem.rLock();
    try {
        SubArray origin = dataItem.data();
        byte[] data = new byte[origin.end - origin.start - ENTRY_DATA_OFFSET];
        System.arraycopy(origin.raw, srcPos: origin.start+ENTRY_DATA_OFFSET,data, destPos: 0,data.length);
        return data;
    }finally {
        dataItem.rUnLock();
    }
}
```

在设置 Xmax 的时候，由于 Entry 实质上是对 DataItem 的修改，在修改前需要调用 before，修改后调用 after 即可。

```
public void setXmax(long xid) {
    dataItem.before();
    try {
        SubArray origin = dataItem.data();
        System.arraycopy(Parser.long2Byte(xid), srcPos: 0,origin.raw, destPos: origin.start+ENTRY_DELETER_XID_OFFSET, length: 8);
    }finally {
        dataItem.after(xid);
    }
}
```

加载 Entry 通过调用 DataManager 的 read 方法读取到 DataItem，将其封装为 Entry 即可。

```
public static Entry loadEntry(VersionManagerImpl vm, long uid) throws Exception {
    DataItem dataItem = vm.dm.read(uid);
    return newEntry(vm, dataItem, uid);
}
```

4.3.2 MVCC 与事务隔离级别

EasySQL 对同一资源写写操作会被 2PL 锁阻塞，为了减少阻塞概率，消除读写阻塞，引入了 MVCC 机制。在 VersionManager 中，为每条记录维护了多个版本。举个例子，若事务 T1 打算对 X 记录更新，会对其上锁，创建更新的版本 X3。若另外一个事务 T2 想读取 X 的值，从 X3-X 顺序去遍历，发现 X3 创建事务未提交，查找到下一条 X。这是因为记录的查找依赖索引模块，最新插入的记录总是会被最先查找到。　　并且，通过 2PL 锁保证了某个事务不

会去修改另外事务修改但并未提交的数据项。

4.3.3 Transaction 类

EasySQL 将每个事务抽象为 Transaction 类，类中保存该事务的 xid，隔离级别 level，事务出现异常后记录下的异常信息，并记录了可重复读事务创建时的初始快照。

```
public class Transaction {
    public long xid;
    public int level;
    public Map<Long, Boolean> snapshot;
    public Exception err;
    public boolean autoAborted;

    public static Transaction newTransaction(lon

    public boolean isInSnapshot(long xid) {...}
}
```

4.3.4.1 读已提交

在读已提交隔离级别下，规定了事务只能读到已提交事务创建的数据，具体可见判定逻辑为目标记录创建事务已经提交并且没有事务删除该记录或删除该记录的事务尚未提交或该记录由当前事务创建并未删除。因此读已提交的可见性判断代码逻辑为：

```
private static boolean readCommitted(TransactionManager tm, Transaction transaction, Entry entry) {
    long xmin = entry.getXmin();
    long xmax = entry.getXmax();
    if (xmin == transaction.xid&&xmax==0) {
        return true;
    }
    if (tm.isCommitted(xmin)) {
        if (xmax == 0) {
            return true;
        }
        if (xmax != transaction.xid && !tm.isCommitted(xmax)) {
            return true;
        }
    }
    return false;
}
```

4.3.4.2 可重复读

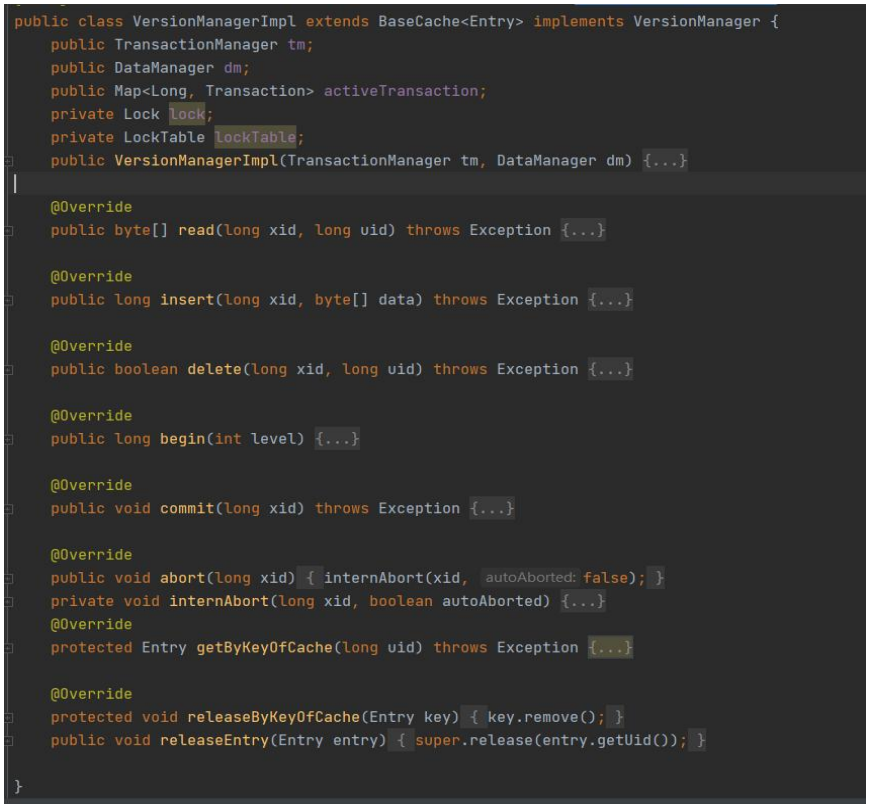
可重复读的可见性判断逻辑稍微复杂一点。具体可见判定逻辑为目标记录的创建事务若在当前事务之前创建并提交且没有事务删除该记录，或将其删除的事务在当前事务之后，或将其删除的事务在当前事务启动时处于活跃状态，该记录创建事务在当前事务之后，以及该记录为当前事务创建并未删除。

```
private static boolean repeatableRead(TransactionManager tm, Transaction transaction, Entry entry) {
    long xid = transaction.xid;
    long xmin = entry.getXmin();
    long xmax = entry.getXmax();
    if (xid == xmin && xmax == 0) {
        return true;
    }
    if (tm.isCommitted(xmin) && xmin < xid && !transaction.isInSnapshot(xmin)) {
        if (xmax == 0) {
            return true;
        }
        if (xmax != xid) {
            if (!tm.isCommitted(xmax) || xmax > xid || transaction.isInSnapshot(xmax)) {
                return true;
            }
        }
    }
    return false;
}
```

可重复读下会发生版本跳跃导致产生多条有效记录问题。例如事务 T1 开始，之后事务 T2 开始，T2 修改了 X 产生了新的有效记录 X2 并提交，T1 也想修改 X，此时新记录 X2 对 T1 不可见，因此会获取 X，对 X 修改产生新的有效记录 X3，此时数据库就产生了两条数据，这是不允许的。因此需要先对 X2 进行撤销，需要将 T2 撤销，再将新的有效记录 X3 保存下来。某条记录对当前事务发生了版本跳跃的逻辑是，若该记录的 Xmax 已经是提交状态并且 Xmax 在当前事务之后或是在当前事务的初始快照中，则需要对 Xmax 对应的事务撤销后再执行插入新记录操作。

4.4 VersionManager

VersionManager 提供了对上层模块的功能调用，同时实现了对 Entry 的缓存，Entry 的 uid 即其 key。其中 getByKeyOfCache 的实现逻辑为调用 DataManager 获取 DataItem 并将其封装为 Entry，releaseByKeyOfCache 是调用 Entry 的 release 方法，实际上是 DataItem 的 release 流程。VersionManager 的实现如下：



其中 Read 方法根据 uid 获取对应的 Entry，若 Entry 对当前事务可见则将其数据部分返回，若不可见返回 null。Insert 方法则简单的将 data 封装为 Entry，交给 DataManager 插入即可。Delete 方法涉及到对 Entry 的数据项的修改，因此需要 LockTable 对该操作进行串行化处理。根据 uid 获取对应的 Entry，若该 Entry 不可见，直接返回即可。若可见，通过 LockTable 管理安全地获取对应 Entry，设置 Xmax 即可。

Begin 方法表示开启一个事务，通过 TransactionManager 获取一个 xid，将其放入活跃事务快照即可。Commit 方法表示提交一个事务，从快照中删除该活跃事务，同时释放 LockTable 中该事务持有的资源并通过 TransactionManager 提交事务。Abort 方法同样的释放该事务持有的资源，将其状态设为撤销即可。

VersionManager 只有插入，查找与删除功能，这是因为若要对某条记录进行更新，只需要调用 VM 的删除功能设置其 Xmax，之后再将新数据插入即可。

第五章　索引管理模块

5.1 索引节点定义

EasySQL 采用 B+树索引，类似 MyISAM 存储引擎保存各个数据项。叶子节点保存记录 key 及其地址，非叶节点保存 key 区间信息即子节点地址。节点与数据项一样存在 db 文件中。每个节点格式如下：

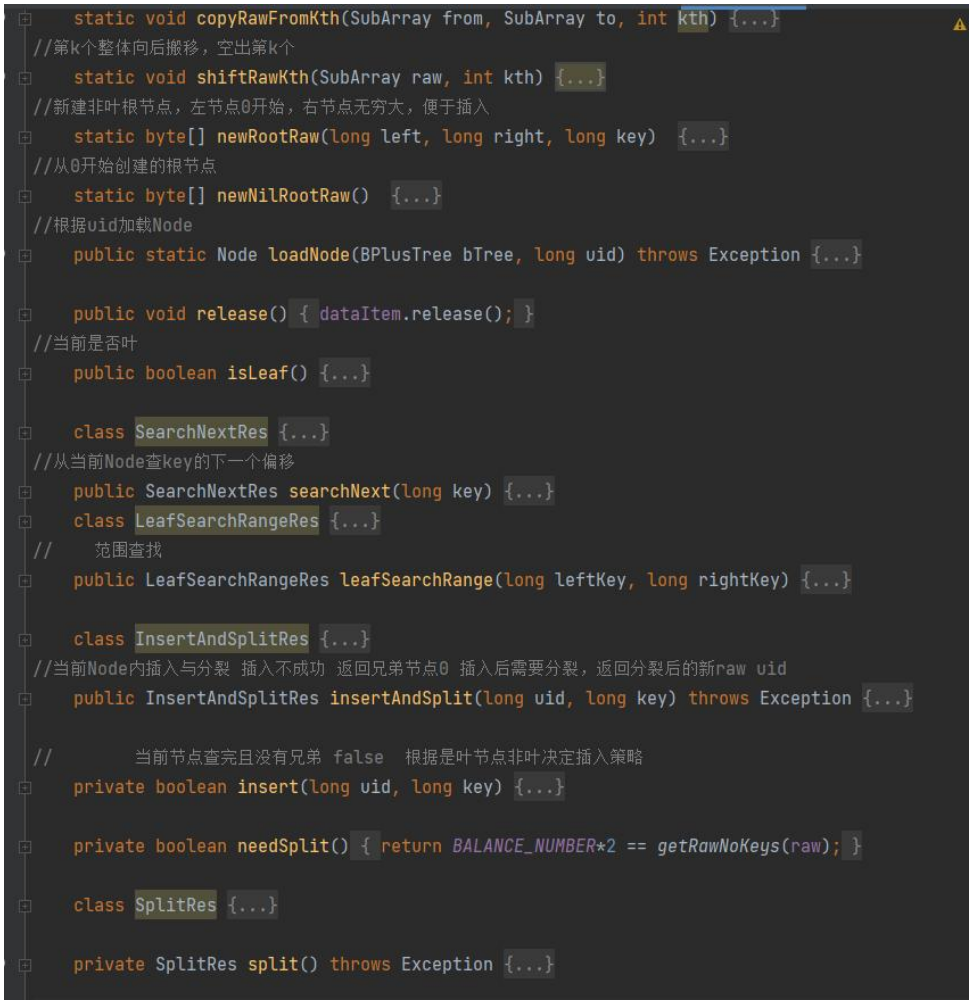
[LeafFlag][KeyNumber][SiblingUid]

[Uid0][Key0][Uid1][KEY1]...[UidN][KeyN]

一字节的 LeafFlag 表示该节点是否为叶节点。两字节的 KeyNumber 表示该节点中包含 key 的个数，八字节的 SiblingUid 表示该节点右兄弟节点地址。每个八字节的 key 都有一个八字节 uid 记录其子节点地址，该子节点的所有 key 都小于这个 key。非根节点的最小 key 数为 32，每个节点大小固定，为 1+2+8+2*8*（32*2+1+1）=1067 字节

其中，65 为节点最大 key 数，留一个 key 是为了下文提到的节点分裂逻辑。节点类：

```
public class Node {
    static final int IS_LEAF_OFFSET = 0;
    static final int NO_KEYS_OFFSET = IS_LEAF_OFFSET+1;
    static final int SIBLING_OFFSET = NO_KEYS_OFFSET+2;
    static final int NODE_HEADER_SIZE = SIBLING_OFFSET+8;

    static final int BALANCE_NUMBER = 32;
    static final int NODE_SIZE = NODE_HEADER_SIZE + (2*8)*(BALANCE_NUMBER*2+2);

    BPlusTree tree;
    DataItem dataItem;
    SubArray raw;
    long uid;
//设置leaf位
    static void setRawIsLeaf(SubArray raw, boolean isLeaf) {...}
//
    static boolean getRawIfLeaf(SubArray raw) { return raw.raw[raw.start + IS_LEAF_OFFSET] == (byte)1; }
//设置当前节点key数
    static void setRawNoKeys(SubArray raw, int noKeys) {...}

    static int getRawNoKeys(SubArray raw) {...}
//当前节点兄弟节点
    static void setRawSibling(SubArray raw, long sibling) {...}

    static long getRawSibling(SubArray raw) {...}
//第k个key的uid
    static void setRawKthSon(SubArray raw, long uid, int kth) {...}

    static long getRawKthSon(SubArray raw, int kth) {...}
//第k个的key值
```



EasySQL 中,B+树是事务无关的，以超级事务执行，这是因为需要 B+树找到所有符合条件的记录。因此某条事务留下很多记录，就算其回滚了依然可以通过索引找到他们。

5.2 B+树定义

B+树对外提供插入与查找功能。没有删除功能是因为 VersionManager 通过 DataManager 提供的 before 与 after 保证了对应记录的安全修改，即设置对应记录的 Xmax 即可。没有更新功能也是因为可以通过 VersionManager 的删除功能与插入功能实现。

```
public class BPlusTree {
    DataManager dm;
    long bootUid;
    DataItem bootDataItem;
    Lock bootLock;

    public static long create(DataManager dm) throws Exception {...}

    public static BPlusTree load(long bootUid, DataManager dm) throws Exception {...}
//dataItem保存rootUid 读之
    private long rootUid() {...}
//赋新根给root
    private void updateRootUid(long left, long right, long rightKey) throws Exception {...}
// 搜叶节点位置
    private long searchLeaf(long nodeUid, long key) throws Exception {...}
//比key大的第一个
    private long searchNext(long nodeUid, long key) throws Exception {...}

//    范围搜
    public List<Long> search(long key) throws Exception {...}

    public List<Long> searchRange(long leftKey, long rightKey) throws Exception {...}
// 递归插入 最后更新头
    public void insert(long key, long uid) throws Exception {...}

    class InsertRes {...}

    private InsertRes insert(long nodeUid, long uid, long key) throws Exception {...}

    private InsertRes insertAndSplit(long nodeUid, long uid, long key) throws Exception {...}

    public void close() { bootDataItem.release(); }
}
```

其中，B+树创建时会新建一个空叶节点。调用 DataManager 的 insert 方法将其插入数据库中。加载时将根节点 uid 读入到 B+树类中即可。

5.2.1 B+树的查找

B+树对单一 Key 值的查找依赖于范围查找。范围查找中，先去查找范围区间左端点 key 所在的叶子节点。从根节点开始加载某个节点地址对应数据，若当前节点为叶子节点直接将该节点地址返回即可。若不是，从当前节点中查找比目标 key 大的第一个 key 对应的子节点地址，若目标 key 存在必会存在于对应子节点或其子节点中。若该节点中不存在这样的 key，则去其兄弟节点重复以上步骤。这样目标 key 若存在就找到了其所在的叶节点。从叶节点中找出>=区间左端点的第一个 key，然后往后遍历，找到所有符合条件的 key，若最右端点的 key 依然在范围内，再去其兄弟节点查找。如此，将他们对应的 uid 返回，也就完成了范围查找。

```
  范围搜
public List<Long> search(long key) throws Exception {
    return searchRange(key, key);
}

public List<Long> searchRange(long leftKey, long rightKey) throws Exception {
    long rootUid = rootUid();
    long leafUid = searchLeaf(rootUid, leftKey);
    List<Long> uids = new ArrayList<>();
    while(true) {
        Node leaf = Node.loadNode( bTree: this, leafUid);
        Node.LeafSearchRangeRes res = leaf.leafSearchRange(leftKey, rightKey);
        leaf.release();
        uids.addAll(res.uids);
        if(res.siblingUid == 0) {
            break;
        } else {
            leafUid = res.siblingUid;
        }
    }
    return uids;
}
```

```
// 搜叶节点位置
    private long searchLeaf(long nodeUid, long key) throws Exception {
        Node node = Node.loadNode( bTree: this, nodeUid);
        boolean isLeaf = node.isLeaf();
        node.release();
        if(isLeaf) {
            return nodeUid;
        } else {
            long next = searchNext(nodeUid, key);
            return searchLeaf(next, key);
        }
    }
```

```
//比key大的第一个
private long searchNext(long nodeUid, long key) throws Exception {
    while(true) {
        Node node = Node.loadNode( bTree: this, nodeUid);
        Node.SearchNextRes res = node.searchNext(key);
        node.release();
        if (res.uid != 0) {
            return res.uid;
        }
        nodeUid = res.siblingUid;
    }
}
```

5.2.2 B+树的插入

B+树中一个记录的插入是一个递归过程。从根节点出发，需要找到应当插入该记录的叶子节点位置。该过程与查找过程类似。如果当前节点已是叶节点，将执行节点插入分裂操作，并将分裂结果返回。若不是叶节点，根据 searchNext 找到下一个节点，再次执行此逻辑。在回溯过程中会获取子节点分裂情况。若子节点返回结果中含有分裂出的新节点，则在当前节点中将分裂出的 key 及其 uid 插入即可。为节点插入分裂操作流程为：在该节点上寻找 key 插入位置，即第一个大于等于目标 key 的位置。若这个位置在该节点最后，并且该节点没有兄弟节点，则将其插入该节点父节点的兄弟节点的子节点中。完成该节点 key 插入后，若该节点 key 值超越了阈值即 keyNumber=66，此时进行分裂逻辑。生成新节点，新节点为原节点中第 33-66 个 key 及其 uid，令原节点的兄弟节点为分裂出的新节点，新节点的 key 为其第一个 key 值。为确保安全，需要调用 DataItem 的 before after 对流程进行保护。这样返回结果就包含了分裂节点与新节点 key 信息，回溯过程中将新节点与新 key，执行插入分裂逻辑将其插入即可。这样最终到了根节点这里，若根节点发生了分裂，则将增加树高，新节点中将包含两个 key，一个是新节点第一个位置的 key 值，一个是无限大。到此，完成了 B+树的插入操作。

```
    }
// 递归插入 最后更新头
    public void insert(long key, long uid) throws Exception {
        long rootUid = rootUid();
        InsertRes res = insert(rootUid, uid, key);
        assert res != null;
        if(res.newNode != 0) {
            updateRootUid(rootUid, res.newNode, res.newKey);
        }
    }
```

```
private InsertRes insert(long nodeUid, long uid, long key) throws Exception {
    Node node = Node.loadNode( bTree: this, nodeUid);
    boolean isLeaf = node.isLeaf();
    node.release();
    InsertRes res = null;
    if(isLeaf) {
        res = insertAndSplit(nodeUid, uid, key);
    } else {
        long next = searchNext(nodeUid, key);
        InsertRes ir = insert(next, uid, key);
        if(ir.newNode != 0) {
            res = insertAndSplit(nodeUid, ir.newNode, ir.newKey);
        } else {
            res = new InsertRes();
        }
    }
    return res;
}
```

```
private InsertRes insertAndSplit(long nodeUid, long uid, long key) throws Exception {
    while(true) {
        Node node = Node.loadNode( bTree: this, nodeUid);
        Node.InsertAndSplitRes iasr = node.insertAndSplit(uid, key);
        node.release();
        if(iasr.siblingUid != 0) {
            nodeUid = iasr.siblingUid;
        } else {
            InsertRes res = new InsertRes();
            res.newNode = iasr.newSon;
            res.newKey = iasr.newKey;
            return res;
        }
    }
}
```

```java
//当前Node内插入与分裂 插入不成功 返回兄弟节点0 插入后需要分裂，返回分裂后的新raw uid
public InsertAndSplitRes insertAndSplit(long uid, long key) throws Exception {
    boolean success = false;
    Exception err = null;
    InsertAndSplitRes res = new InsertAndSplitRes();
    dataItem.before();
    try {
        success = insert(uid, key);
        if(!success) {
            res.siblingUid = getRawSibling(raw);
            return res;
        }
        if(needSplit()) {
            try {
                SplitRes r = split();
                res.newSon = r.newSon;
                res.newKey = r.newKey;
                return res;
            } catch(Exception e) {
                err = e;
                throw e;
            }
        } else {
            return res;
        }
    } finally {
        if(err == null && success) {
            dataItem.after( xid: 0);
        } else {
            dataItem.unBefore();
        }
    }
}
```

```java
private InsertRes insert(long nodeUid, long uid, long key) throws Exception {
    Node node = Node.loadNode( bTree: this, nodeUid);
    boolean isLeaf = node.isLeaf();
    node.release();
    InsertRes res = null;
    if(isLeaf) {
        res = insertAndSplit(nodeUid, uid, key);
    } else {
        long next = searchNext(nodeUid, key);
        InsertRes ir = insert(next, uid, key);
        if(ir.newNode != 0) {
            res = insertAndSplit(nodeUid, ir.newNode, ir.newKey);
        } else {
            res = new InsertRes();
        }
    }
    return res;
}
```

第六章 表管理模块

6.1 字段管理

EasySQL 将每个字段抽象为 Field 类，每个字段在数据库中也是一个数据项，其格式为：

[FieldName][TypeName][IndexUid]

其中，FieldName 和 TypeName 是一个字符串数组。Type 目前总共三种：int32 int64 string。字符串数组格式：

[StringLength][StringData]

如某字段建立了索引，则其 Field 记录的 IndexUid 记录了其 B+树根节点地址 uid。

```
public class Field {
//    Field位置
    long uid;
    private Table tb;
    String fieldName;
    String fieldType;
//    B树根节点位置
    private long index;
    private BPlusTree bt;

    public static Field loadField(Table tb, long uid) {...}

    public Field(long uid, Table tb) {...}

    public Field(Table tb, String fieldName, String fieldType, long index) {...}
//field的raw转为field对象
    private Field parseSelf(byte[] raw) {...}
```

```
    private void persistSelf(long xid) throws Exception {
        byte[] nameRaw = Parser.string2Byte(fieldName);
        byte[] typeRaw = Parser.string2Byte(fieldType);
        byte[] indexRaw = Parser.long2Byte(index);
        this.uid = ((TableManagerImpl)tb.tbm).vm.insert(xid, Bytes.concat(nameRaw, typeRaw, indexRaw));
    }
```

ParseSelf 方法提供了将 DataItem 转换为 Field 对象的方法，PersistSelf 方法为将 Field 对象通过 VersionManager 插入数据库持久化。

```java
//给表创建field
public static Field createField(Table tb, long xid, String fieldName, String fieldType, boolean indexed
    typeCheck(fieldType);
    Field f = new Field(tb, fieldName, fieldType, index: 0);
    if(indexed) {
        long index = BPlusTree.create(((TableManagerImpl)tb.tbm).dm);
        BPlusTree bt = BPlusTree.load(index, ((TableManagerImpl)tb.tbm).dm);
        f.index = index;
        f.bt = bt;
    }
    f.persistSelf(xid);
    return f;
}
```

CreateField 方法是 Field 的创建方法，TableManager 在创建表时会对每个字段调用此方法创建 Field 对象，并持久化 Field 及索引（需要的情况下）。

```java
public void insert(Object key, long uid) throws Exception {
    long uKey = value2Uid(key);
    bt.insert(uKey, uid);
}
```

Insert 方法将对应记录插入 B+树上。

```java
public FieldCalRes calExp(SingleExpression exp) throws Exception {
    Object v = null;
    FieldCalRes res = new FieldCalRes();
    switch(exp.compareOp) {
        case "<":
            res.left = 0;
            v = string2Value(exp.value);
            res.right = value2Uid(v);
            if(res.right > 0) {
                res.right --;
            }
            break;
        case "=":
            v = string2Value(exp.value);
            res.left = value2Uid(v);
            res.right = res.left;
            break;
        case ">":
            res.right = Long.MAX_VALUE;
            v = string2Value(exp.value);
            res.left = value2Uid(v) + 1;
            break;
    }
    return res;
}
```

CalExp 提供了在每个字段上解析 Where 条件表达式的方法。目前支持的条件为> < =三个条件。返回结果是一个范围区间。

6.2 表管理

EasySQL 将表抽象为 Table 类。Table 信息同样会作为一个数据项插入数据库中。该数据项格式为：

[TableName][NextTableUid][Field1][Field2]...[FieldN]

每次创建表时，都会创建一个.bt 文件，该文件表示当前库下第一张表的 uid。因此每张表之间形成了一个单链表。创建新表只需要将新表的 NextTableUid 指向之前的第一张表即可。当然，不做字段长度记录的原因是因为插入操作最终都会调用 DataManager 的操作，该操作自然会帮我们记录数据项的长度，每个 FieldUid 长度是固定的。Table 类如下：

```
public class Table {
    public TableManager tbm;
    public long uid;
    public String name;
    public byte status;
    public long nextUid;
    List<Field> fields = new ArrayList<>();

    public static Table loadTable(TableManager tbm, long uid) {...}
//根据create建表 按每个field初始化
    public static Table createTable(TableManager tbm, long nextUid, long xid, Create create) throws Exception {...}

    public Table(TableManager tbm, long uid) {...}

    public Table(TableManager tbm, String tableName, long nextUid) {...}

    private Table parseSelf(byte[] raw) {...}

    private Table persistSelf(long xid) throws Exception {...}

    public int delete(long xid, Delete delete) throws Exception {...}

    public int update(long xid, Update update) throws Exception {...}

    public String read(long xid, Select read) throws Exception {...}
//将insert中的values依次放到对应field上 将得到的数据项插入 若该field有索引 插入B+树
    public void insert(long xid, Insert insert) throws Exception {...}

//values-->数据raw
    private Map<String, Object> string2Entry(String[] values) throws Exception {...}
```

```
    private List<Long> parseWhere(Where where) throws Exception {...}

    class CalWhereRes {...}

    private CalWhereRes calWhere(Field fd, Where where) throws Exception {...}
//返回一条数据String
    private String printEntry(Map<String, Object> entry) {...}
//一条数据转为Entry
    private Map<String, Object> parseEntry(byte[] raw) {...}

    private byte[] entry2Raw(Map<String, Object> entry) {...}

    @Override
    public String toString() {...}
}
```

Table 中会保存该表字段列表。LoadTable 方法主要配合 parseSelf 方法从数据库对应位置解析表名，字段信息获取 Table 对象。

Create，insert，delete，update，read 方法都是用于根据 SQL 语句解析出的对应语句对象调用 VersionManager 提供的对应方法执行数据库操作。

Create 对象是 create SQL 语句的抽象。包含表名，字段，字段类型，索引信息。

```
public class Create {
    public String tableName;
    public String[] fieldName;
    public String[] fieldType;
    public String[] index;
}
```

Create 方法在数据库中初始化字段项，字段对象，最后将创建的表持久化到数据库。在 TableManager 调用此方法后会将这个新的头表记录到.bt 文件中。

```java
//根据create建表 按每个field初始化
public static Table createTable(TableManager tbm, long nextUid, long xid, Create create) throws Exception {
    Table tb = new Table(tbm, create.tableName, nextUid);
    for(int i = 0; i < create.fieldName.length; i ++) {
        String fieldName = create.fieldName[i];
        String fieldType = create.fieldType[i];
        boolean indexed = false;
        for(int j = 0; j < create.index.length; j ++) {
            if(fieldName.equals(create.index[j])) {
                indexed = true;
                break;
            }
        }
        tb.fields.add(Field.createField(tb, xid, fieldName, fieldType, indexed));
    }

    return tb.persistSelf(xid);
}
```

Insert 对象定义如下：

```java
public class Insert {
    public String tableName;
    public String[] values;
}
```

在 Table 类的 insert 方法中，将 Insert 对象按照每个字段格式转换为数据项，再调用 vm 将该数据项插入表中。同时对于建立了索引的字段，逐个维护其索引。

Delete 类如下：

```java
public class Delete {
    public String tableName;
    public Where where;
}
```

在 Delete 方法中根据 where 解析出需要删除的字段值范围，查到这些记录的 uid，调用版本控制器对其删除即可。

```java
public int delete(long xid, Delete delete) throws Exception {
    List<Long> uids = parseWhere(delete.where);
    int count = 0;
    for (Long uid : uids) {
        if(((TableManagerImpl)tbm).vm.delete(xid, uid)) {
            count ++;
        }
    }
    return count;
}
```

Update 类如下：

```java
public class Update {
    public String tableName;
    public String fieldName;
    public String value;
    public Where where;
}
```

Update 方法根据 where 解析出需要操作的记录，对这些记录依次调用 VersionManager 模块的 delete 方法，并创建新记录 VM 的插入方法插入数据库，将新纪录索引列插入对应 B+树即可。

```java
public int update(long xid, Update update) throws Exception {
    List<Long> uids = parseWhere(update.where);
    Field fd = null;
    for (Field f : fields) {
        if(f.fieldName.equals(update.fieldName)) {
            fd = f;
            break;
        }
    }
    if(fd == null) {
        throw Error.FieldNotFoundException;
    }
    Object value = fd.string2Value(update.value);
    int count = 0;
    for (Long uid : uids) {
        byte[] raw = ((TableManagerImpl)tbm).vm.read(xid, uid);
        if (raw == null) {
            continue;
        }
        ((TableManagerImpl)tbm).vm.delete(xid, uid);
        Map<String, Object> entry = parseEntry(raw);
        entry.put(fd.fieldName, value);
        raw = entry2Raw(entry);
        long uuid = ((TableManagerImpl)tbm).vm.insert(xid, raw);
        count ++;
        for (Field field : fields) {
            if(field.isIndexed()) {
                field.insert(entry.get(field.fieldName), uuid);
            }
        }
    }
```

Select 类定义如下：

```java
public class Select {
    public String tableName;
    public String[] fields;
    public Where where;
}
```

Read 方法先解析 Where 范围，再调用 VM 提供的 read 方法读出查到的记录即可。

```java
public String read(long xid, Select read) throws Exception {
    List<Long> uids = parseWhere(read.where);
    StringBuilder sb = new StringBuilder();
    for (Long uid : uids) {
        byte[] raw = ((TableManagerImpl)tbm).vm.read(xid, uid);
        if (raw == null) {
            continue;
        }
        Map<String, Object> entry = parseEntry(raw);

        sb.append(printEntry(entry)).append("\n");
    }
    return sb.toString();
}
```

CalWhere 方法根据 SQL 语句中的 where 条件获取范围。目前 EasySQL 最多支持两个表达式用 or 或 and 连接。

```
private CalWhereRes calWhere(Field fd, Where where) throws Exception {
    CalWhereRes res = new CalWhereRes();
    switch(where.logicOp) {
        case "":
            res.single = true;
            FieldCalRes r = fd.calExp(where.singleExp1);
            res.l0 = r.left; res.r0 = r.right;
            break;
        case "or":
            res.single = false;
            r = fd.calExp(where.singleExp1);
            res.l0 = r.left; res.r0 = r.right;
            r = fd.calExp(where.singleExp2);
            res.l1 = r.left; res.r1 = r.right;
            break;
        case "and":
            res.single = true;
            r = fd.calExp(where.singleExp1);
            res.l0 = r.left; res.r0 = r.right;
            r = fd.calExp(where.singleExp2);
            res.l1 = r.left; res.r1 = r.right;
            if (res.l1 > res.l0) {
                res.l0 = res.l1;
            }
            if (res.r1 < res.r0) {
                res.r0 = res.r1;
            }
            break;
        default:
            throw Error.InvalidLogOpException;
    }
    return res;
}
```

6.3 TableManager

服务器端将 SQL 语句转换为对应的各个类，调用 TableManager 对应的方法对数据库进行读写操作。TableManager 接口及实现类定义：

```java
public interface TableManager {
    BeginRes begin(Begin begin);
    byte[] commit(long xid) throws Exception;
    byte[] abort(long xid);

    byte[] show(long xid);
    byte[] create(long xid, Create create) throws Exception;

    byte[] insert(long xid, Insert insert) throws Exception;
    byte[] read(long xid, Select select) throws Exception;
    byte[] update(long xid, Update update) throws Exception;
    byte[] delete(long xid, Delete delete) throws Exception;

    public static TableManager create(String path, VersionManager vm, DataManager dm) {
        Booter booter = Booter.create(path);
        booter.update(Parser.long2Byte( value: 0));
        return new TableManagerImpl(vm, dm, booter);
    }

    public static TableManager open(String path, VersionManager vm, DataManager dm) {
        Booter booter = Booter.open(path);
        return new TableManagerImpl(vm, dm, booter);
    }
}
```

```java
public class TableManagerImpl implements TableManager {
    public VersionManager vm;
    public DataManager dm;
    private Booter booter;
    private Map<String, Table> tableCache;
    private Map<Long, List<Table>> xidTableCache;
    private Lock lock;

    public TableManagerImpl(VersionManager vm, DataManager dm, Booter booter) {
        this.vm = vm;
        this.dm = dm;
        this.booter = booter;
        this.tableCache = new HashMap<>();
        this.xidTableCache = new HashMap<>();
        lock = new ReentrantLock();
        loadTables();
    }
//所有表加入缓存
```

对于 begin，commit，abort 方法直接调用 VersionManager 提供的对应方法即可。TableManager 在创建时会从.bt 文件中将所有表转换为 Table 对象并加入缓存。create 方法除了调用 Table 的 create 方法外，将创建的表插入到表头即可。

```java
@Override
public byte[] create(long xid, Create create) throws Exception {
    lock.lock();
    try {
        if(tableCache.containsKey(create.tableName)) {
            throw Error.DuplicatedTableException;
        }
        Table table = Table.createTable( tbm: this, firstTableUid(), xid, create);
        updateFirstTableUid(table.uid);
        tableCache.put(create.tableName, table);
        if(!xidTableCache.containsKey(xid)) {
            xidTableCache.put(xid, new ArrayList<>());
        }
        xidTableCache.get(xid).add(table);
        return ("create " + create.tableName).getBytes();
    } finally {
        lock.unlock();
    }
}
@Override
```

其余增删查改通过从缓存中获取 Table 对象，调用对应操作完成。

第七章 服务器与客户端通信

7.1 前后端通信工具类

Package 是传输基本结构，通过客户端 Encoder 编码以及服务器端 Encoder 解码实现数据通信。其编解码规则为：

[Flag][Data]

Flag=0 表示这个 Package 中的 data 为数据，Flag=1 表示 data 为错误信息。

Package 基本定义如下：

```
public class Package {
    byte[] data;
    Exception err;

    public Package(byte[] data, Exception err) {
        this.data = data;
        this.err = err;
    }

    public byte[] getData() { return data; }

    public Exception getErr() { return err; }
}
```

Encoder 负责将 Package 编码与解码。

```
ublic class Encoder {
    public byte[] encode(Package pkg) {
        if(pkg.getErr() != null) {
            Exception err = pkg.getErr();
            String msg = "Intern server error!";
            if(err.getMessage() != null) {
                msg = err.getMessage();
            }
            return Bytes.concat(new byte[]{1}, msg.getBytes());
        } else {
            return Bytes.concat(new byte[]{0}, pkg.getData());
        }
    }
    public Package decode(byte[] data) throws Exception {
        if(data.length < 1) {
            throw Error.InvalidPkgDataException;
        }
        if(data[0] == 0) {
            return new Package(Arrays.copyOfRange(data, from: 1, data.length), err: null);
        } else if(data[0] == 1) {
            return new Package( data: null, new RuntimeException(new String(Arrays.copyOfRange(data, from: 1,
        } else {
            throw Error.InvalidPkgDataException;
        }
    }
```

Transporter 内部保存了客户端到服务器端的 Socket 连接 Channel，对信息编码后将其转换为十六进制字符串，末尾加上换行符由 Transporter 类写入 SocketChannel 发送出去，在读写时通过 BufferedReader 和 BufferedWriter 很方便的进行处理。Packager 封装了 Transporter 和 Encoder，直接调用其 send 与 receive 方法即可完成数据接受与发送。

```java
public class Transporter {
    private Socket socket;
    private BufferedReader reader;
    private BufferedWriter writer;
    public Transporter(Socket socket) throws IOException {
        this.socket = socket;
        this.reader = new BufferedReader(new InputStreamReader(socket.getInputStream()));
        this.writer = new BufferedWriter(new OutputStreamWriter(socket.getOutputStream()));
    }
    public void send(byte[] data) throws Exception {
        String raw = hexEncode(data);
        writer.write(raw);
        writer.flush();
    }
    public byte[] receive() throws Exception {
        String line = reader.readLine();
        if(line == null) {
            close();
        }
        return hexDecode(line);
    }
    public void close() throws IOException {
        writer.close();
        reader.close();
        socket.close();
    }

    private String hexEncode(byte[] buf) { return Hex.encodeHexString(buf,  toLowerCase: true)+"\n"; }
    private byte[] hexDecode(String buf) throws DecoderException {
        return Hex.decodeHex(buf);
    }
}
```

7.2 Server

Server 直接采用了 Java 的 Socket api。Server 启动一个 ServerSocket 监听端口，当有请求到来时直接把请求交给线程池处理。HandleSocket 实现了 Runable 接口，在 run 中定义其行为交给线程池运行即可。Exeutor 根据 sql 语句类型转换为对应对象，交给 TableManager 运行并将结果返回。EasySQL 对于 SQL 语句的解析非常简单，根据语义逐个逐个解析为对于语句对象。再不赘述。

```java
public class Server {
    private int port;
    TableManager tbm;
    public Server(int port, TableManager tbm) {
        this.port = port;
        this.tbm = tbm;
    }

    public void start() {
        ServerSocket serverSocket = null;
        try {
            serverSocket = new ServerSocket(port);
        } catch (IOException e) {
            e.printStackTrace();
            return;
        }
        log.info("Server listen to port:" + port);
        ThreadPoolExecutor tpe = new ThreadPoolExecutor( corePoolSize: 10,  maximumPoolSize: 20,
                keepAliveTime: 1L, TimeUnit.SECONDS, new ArrayBlockingQueue<>( capacity: 100), new ThreadPoolExecutor.CallerRunsPolicy());
        try {
            while (true) {
                Socket accept = serverSocket.accept();
                HandleSocket handleSocket = new HandleSocket(accept, tbm);
                tpe.execute(handleSocket);
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            try {
                serverSocket.close();
            } catch (IOException e) {}
        }
```

```java
public class HandleSocket implements Runnable{
    private Socket socket;
    private TableManager tbm;
    public HandleSocket(Socket socket, TableManager tbm) {
        this.socket = socket;
        this.tbm = tbm;
    }
    @Override
    public void run() {
        InetSocketAddress address = (InetSocketAddress) socket.getRemoteSocketAddress();
        log.info("Establish connetction: {} : {}", address.getAddress().getHostAddress(), address.getPort());
        Transporter transporter = null;
        try {
            transporter = new Transporter(socket);
        } catch (IOException e) {
            e.printStackTrace();
            try {
                socket.close();
            } catch (IOException ioException) {
                ioException.printStackTrace();
            }
            return;
        }
        Encoder encoder = new Encoder();
        Packager packager = new Packager(transporter, encoder);
        Executor exe = new Executor(tbm);
        while(true) {
            Package pkg = null;
            try {
                pkg = packager.receive();
            } catch(Exception e) {
```

```
                break;
            }
            byte[] sql = pkg.getData();
            byte[] result = null;
            Exception e = null;
            try {
                result = exe.execute(sql);
            } catch (Exception e1) {
                e = e1;
                e.printStackTrace();
            }
            pkg = new Package(result, e);
            try {
                packager.send(pkg);
            } catch (Exception e1) {
                e1.printStackTrace();
                break;
            }
        }
        exe.close();
        try {
            packager.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

```java
@Slf4j
public class Executor {
    private long xid;
    TableManager tbm;
    public Executor(TableManager tbm) {
        this.tbm = tbm;
        this.xid = 0;
    }
    public void close() {
        if(xid != 0) {
            log.info("Abnormal Abort: " + xid);
            tbm.abort(xid);
        }
    }

    public byte[] execute(byte[] sql) throws Exception {
        log.info("Server Execute: {}", new String(sql));
        Object stat = Parser.Parse(sql);
        if(Begin.class.isInstance(stat)) {
            if(xid != 0) {
                throw Error.NestedTransactionException;
            }
            BeginRes r = tbm.begin((Begin)stat);
            xid = r.xid;
            return r.result;
        } else if(Commit.class.isInstance(stat)) {
            if(xid == 0) {
                throw Error.NoTransactionException;
            }
            byte[] res = tbm.commit(xid);
            xid = 0;
            return res;
```

```
        } else if(Abort.class.isInstance(stat)) {
            if(xid == 0) {
                throw Error.NoTransactionException;
            }
            byte[] res = tbm.abort(xid);
            xid = 0;
            return res;
        } else {
            return execute2(stat);
        }
    }

    private byte[] execute2(Object stat) throws Exception {
        boolean tmpTransaction = false;
        Exception e = null;
        if(xid == 0) {
            tmpTransaction = true;
            BeginRes r = tbm.begin(new Begin());
            xid = r.xid;
        }
        try {
            byte[] res = null;
            if(Show.class.isInstance(stat)) {
                res = tbm.show(xid);
            } else if(Create.class.isInstance(stat)) {
                res = tbm.create(xid, (Create)stat);
            } else if(Select.class.isInstance(stat)) {
                res = tbm.read(xid, (Select)stat);
            } else if(Insert.class.isInstance(stat)) {
                res = tbm.insert(xid, (Insert)stat);
            } else if(Delete.class.isInstance(stat)) {
                res = tbm.delete(xid, (Delete)stat);
            } else if(Update.class.isInstance(stat)) {
                res = tbm.update(xid, (Update)stat);
```

```
            } else if(Update.class.isInstance(stat)) {
                res = tbm.update(xid, (Update)stat);
            }
            return res;
        } catch(Exception e1) {
            e = e1;
            throw e;
        } finally {
            if(tmpTransaction) {
                if(e != null) {
                    tbm.abort(xid);
                } else {
                    tbm.commit(xid);
                }
                xid = 0;
            }
        }
    }
}
```

服务器的启动根据命令行传入启动或创建数据库命令，服务器就启动起来了。代码入下：

```java
public class Launcher {
    public static void main(String[] args) throws ParseException {
        Options options = new Options();
        options.addOption( opt: "open",  hasArg: true,  description: "-open DBPath");
        options.addOption( opt: "create",  hasArg: true,  description: "-create DBPath");
        options.addOption( opt: "mem",  hasArg: true,  description: "-mem 64MB");
        CommandLineParser parser = new DefaultParser();
        CommandLine cmd = parser.parse(options,args);

        if(cmd.hasOption("open")) {
            openDB(cmd.getOptionValue("open"), parseMem(cmd.getOptionValue("mem")));
            return;
        }
        if(cmd.hasOption("create")) {
            createDB(cmd.getOptionValue("create"));
            return;
        }
        log.info("Usage: launcher (open|create) DBPath");
    }
    public static void createDB(String path) {
        TransactionManager tm = TransactionManager.create(path);
        DataManager dm = DataManager.create(path, DEFALUT_MEM, tm);
        VersionManager vm = new VersionManagerImpl(tm, dm);
        TableManager.create(path, vm, dm);
        tm.close();
        dm.close();
    }

     public static void openDB(String path, long mem) {
```

```
        TransactionManager tm = TransactionManager.open(path);
        DataManager dm = DataManager.open(path, mem, tm);
        VersionManager vm = new VersionManagerImpl(tm, dm);
        TableManager tbm = TableManager.open(path, vm, dm);
        new Server(DEFAULT_PORT, tbm).start();
    }

    public static long parseMem(String memStr) {
        if(memStr == null || "".equals(memStr)) {
            return DEFALUT_MEM;
        }
        if(memStr.length() < 2) {
            Exit.systemExit(Error.InvalidMemException);
        }
        String unit = memStr.substring(memStr.length()-2);
        long memNum = Long.parseLong(memStr.substring(0, memStr.length()-2));
        switch(unit) {
            case "KB":
                return memNum*KB;
            case "MB":
                return memNum*MB;
            case "GB":
                return memNum*GB;
            default:
                Exit.systemExit(Error.InvalidMemException);
        }
        return DEFALUT_MEM;
    }
}
```

7.3 Client

Client 的任务也很简单。在 Shell 中发送一个 SQL 语句给服务器，等待服务器返回结果即可。

```java
public class Client {
    private RoundTripper rt;
    public Client(Packager packager) {
        this.rt = new RoundTripper(packager);
    }
    public byte[] execute(byte[] stat) throws Exception {
        Package pkg = new Package(stat,  err: null);
        Package resPkg = rt.roundTrip(pkg);
        if(resPkg.getErr() != null) {
            throw resPkg.getErr();
        }
        return resPkg.getData();
    }
    public void close() {
        try {
            rt.close();
        } catch (Exception e) {
        }
    }
}
```

其中 roundTrip 方法返回传入 package 的服务器返回结果。

```java
public class RoundTripper {
    private Packager packager;

    public RoundTripper(Packager packager) { this.packager = packager; }

    public Package roundTrip(Package pkg) throws Exception {
        packager.send(pkg);
        return packager.receive();
    }

    public void close() throws Exception {
        packager.close();
    }
}
```

客户端的 Shell 通过 Scanner 循环读入用户传入的 SQL，调用 Client 的 execute 方法并将结果打印给用户。

```java
public class Shell {
    private Client client;

    public Shell(Client client) {
        this.client = client;
    }

    public void run() {
        Scanner sc = new Scanner(System.in);
        try {
            while(true) {
                System.out.print(":> ");
                String statStr = sc.nextLine();
                if("exit".equals(statStr) || "quit".equals(statStr)) {
                    break;
                }
                try {
                    byte[] res = client.execute(statStr.getBytes());
                    System.out.println(new String(res));
                } catch(Exception e) {
                    System.out.println(e.getMessage());
                }
            }
        } finally {
            sc.close();
            client.close();
        }
    }
}
```

这样，客户端的启动类只需要建立 Socket 连接并启动这个 Shell 就可以了。

```java
public class Shell {
    private Client client;

    public Shell(Client client) {
        this.client = client;
    }

    public void run() {
        Scanner sc = new Scanner(System.in);
        try {
            while(true) {
                System.out.print(":> ");
                String statStr = sc.nextLine();
                if("exit".equals(statStr) || "quit".equals(statStr)) {
                    break;
                }
                try {
                    byte[] res = client.execute(statStr.getBytes());
                    System.out.println(new String(res));
                } catch(Exception e) {
                    System.out.println(e.getMessage());
                }
            }
        } finally {
            sc.close();
            client.close();
        }
    }
}
```

```java
public class Launcher {
    public static void main(String[] args) throws IOException {
        Socket socket = new Socket(DEFAULT_SERVER_ADDRESS, DEFAULT_PORT);
        Encoder encoder = new Encoder();
        Transporter transporter = new Transporter(socket);
        Packager packager = new Packager(transporter, encoder);
        Client client = new Client(packager);
        Shell shell = new Shell(client);
        shell.run();
    }
}
```

使用示例

通过 mvn compile 编译项目

执行命令 mvn exec:java -Dexec.mainClass=”com.madara.server.Launcher” -Dexec.args=”-create D:/EasySQLTest/test” 创建数据库

```
F:\JavaSE\EasySQL>mvn exec:java -Dexec.mainClass="com.madara.server.Launcher" -Dexec.args="-create E:/EasySQLTest/test"
```

执行命令 mvn exec:java -Dexec.mainClass=”com.madara.server.Launcher” -Dexec.args=”-open D:/EasySQLTest/test” 打开数据库并开启服务器

```
F:\JavaSE\EasySQL>mvn exec:java -Dexec.mainClass="com.madara.server.Launcher" -Dexec.args="-open E:/EasySQLTest/test"
```

```
17:54:49.127 [com.madara.server.Launcher.main()] INFO com.madara.server.server.Server - Server listen to port:9999
```

执行命令 mvn exec:java -Dexec.mainClass=”com.madara.client.Launcher” 在命令行开启客户端 Shell 输入 SQL 语句。

```
F:\JavaSE\EasySQL>mvn exec:java -Dexec.mainClass="com.madara.client.Launcher"
```

支持的 SQL 语句：

create table xxx fieldName1 fieldType1[,fieldNameN fieldTypeN] (index fieldname1 [fieldnameN])。 示例：

```
:> create table table_test id int32,value int64,name string (index id value)
create table_test
```

begin isolation level read [repeatable|commited]

```
:> begin isolation level read committed
begin
```

abort or commit

insert into [tablename] values xx xx ...

```
:> insert into table_test values 1 10 'No 1'
insert
```

select field1[,field2...] or * from [tablename] [where ...]

```
:> select * from table_test
[1, 10, No 1]
```

update tablename set fieldname=xx where fieldname=xx

```
:> update table_test set value=100 where id=1
update 1
```

delete from tablename where fieldname=xx

```
:> delete from table_test where id=1
delete 1
```