浙江大学

《数据库系统》
实验报告

作业名称	：	MiniSQL
姓　　名	：	王晓宇
学　　号	：	3220104364
电子邮箱	：	3220104364@zju.edu.cn
联系电话	：	19550222634
授课教师	：	孙建伶
助　　教	：	聂俊哲/石宇新

2024 年 6 月 8 日

实验名称 6 RECOVERY MANAGER

实验目的

> *Recovery Manager 负责管理和维护数据恢复的过程，包括：*
>
> - *日志结构的定义*
> - *检查点CheckPoint的定义*
> - *执行Redo、Undo等操作，处理插入、删除、更新，事务的开始、提交、回滚等日志，将数据库恢复到宕机之前的状态*
>
> *出于实现复杂度的考虑，同时为了避免各模块耦合太强，前面模块的问题导致后面模块完全无法完成，同组成员的工作之间影响过深，我们将Recovery Manager模块单独拆了出来。另外为了减少重复的内容，我们不重复实现日志的序列化和反序列化操作，实现一个纯内存的数据恢复模块即可。*

实验环境

> *1. 操作系统：Windows 11 23H2*
>
> *2. 数据库管理系统：MiniSQL*
>
> *3. 工具：CLion 2023.3.3*
>
> *4. 工程管理：Git/GitHub*

实验流程

设计思路

数据恢复是一个很复杂的过程，需要涉及系统的多个模块。以InnoDB为例，在其恢复过程中需要`redo log`、`binlog`、`undo log`等参与，这里把InnoDB的恢复过程主要划分为两个阶段：第一阶段主要依赖于`redo log`的恢复，而第二阶段需要`binlog`和`undo log`的共同参与。

第一阶段，数据库启动后，InnoDB会通过`redo log`找到最近一次`checkpoint`的位置，然后根据`checkpoint`相对应的`LSN`开始，获取需要重做的日志，接着解析日志并且保存到一个哈希表中，最后通过遍历哈希表中的`redo log`信息，读取相关页进行恢复。

在该阶段中，所有被记录到`redo log`但是没有完成数据刷盘的记录都被重新落盘。然而，InnoDB单靠`redo log`的恢复是不够的，因为数据库在任何时候都可能发生宕机，需要保证重启数据库时都能恢复到一致性的状态。这个一致性的状态是指此时所有事务要么处于提交，要么处于未开始的状态，不应该有事务处于执行了一半的状态。所以我们可以通过`undo log`在数据库重启时把正在提交的事务完成提交，活跃的事务回滚，保证了事务的原子性。此外，只有`redo log`还不能解决主从数据不一致等问题。

第二阶段，根据`undo`中的信息构造所有未提交事务链表，最后通过上面两部分协调判断事务是否需要提交还是回滚。InnoDB使用了多版本并发控制(MVCC)以满足事务的隔离性，简单的说就是不同活跃事务的数据互相是不可见的，否则一个事务将会看到另一个事务正在修改的数据。InnoDB借助`undo log`记录的历史版本数据，来恢复出对于一个事务可见的数据，满足其读取数据的请求。

log_rec.h相关

在我们的实验中，日志在内存中以`LogRec`的形式表现，定义于`src/include/recovery/log_rec.h`.

这里我们实现以下函数:

- `CreateInsertLog()`：创建一条插入日志
- `CreateDeleteLog()`：创建一条删除日志
- `CreateUpdateLog()`：创建一条更新日志
- `CreateBeginLog()`：创建一条事务开始日志
- `CreateCommitLog()`：创建一条事务提交日志
- `CreateAbortLog()`：创建一条事务回滚日志

LocRecType类

```
enum class LogRecType {
    kInvalid,
    kInsert,
    kDelete,
    kUpdate,
    kBegin,
    kCommit,
    kAbort,
};
```

`enum`数据类型的`LogRecType`存储了六种有效事务状态类型：插入(Insert)、删除(Delete)、更新(Update)、开始(Begin)、提交(Commir)、取消(Abort).

LogRec日志记录结构体

> *内存日志结构。这里可以不考虑消耗内存空间的优化，实现一种能够用于所有类型日志的日志结构。*

```
struct LogRec {
  LogRec() = default;

  LogRecType type_{LogRecType::kInvalid};
  lsn_t lsn_{INVALID_LSN};
  lsn_t prev_lsn_{INVALID_LSN};
  txn_id_t txn_id_{INVALID_TXN_ID};
  KeyType ins_key_,del_key_,old_key_,new_key_;
  [[maybe_unused]] ValType
ins_val_{},del_val_{},old_val_{},new_val_{};

  LogRec(LogRecType type,lsn_t lsn,lsn_t prev_lsn,txn_id_t
txn_id):
  type_(type),lsn_(lsn),prev_lsn_(prev_lsn),txn_id_(txn_id)
  {}

  /* used for testing only */
  static std::unordered_map<txn_id_t, lsn_t> prev_lsn_map_;
  static lsn_t next_lsn_;
};
```

变量类型	变量名称	含义
LogRecType	type_	该日志的事务状态类型(如插入、删除)
lsn_t	lsn_	该日志的LSN号
lsn_t	prev_lsn_	关于该事务的前一个日志的LSN号
txn_id_t	txn_id_	该日志的对应的事务id
KeyType	ins_key_	对于插入类型的日志来说，代表它的插入位置
KeyType	del_key_	对于删除类型的日志来说，代表它的删除位置
KeyType	old_key_	对于更新类型的日志来说，代表它旧值的删除位置
KeyType	new_key_	对于更新类型的日志来说，代表它新值的更新位置
ValType	ins_val	对于插入类型的日志来说，代表它的插入值
ValType	del_val_	对于删除类型的日志来说，代表它的删除的值
ValType	old_val_	对于更新类型的日志来说，代表它的删除的旧值
ValType	new_val_	对于更新类型的日志来说，代表它的更新的新值

其他变量(用作测试)

```
1 std::unordered_map<txn_id_t, lsn_t> LogRec::prev_lsn_map_ = {};
2 lsn_t LogRec::next_lsn_ = 0;
```

变量名称	含义
prev_lsn_map_	unorder_map类型变量，连接事务号和前LSN号
next_lsn_	静态成员变量，用来记录log更新过程中递增的LSN号

作为全局变量，可以很好地维护LSN和对应`Map`,便于测试查看

CreateInsertLog

函数实现：

```
1 static LogRecPtr CreateInsertLog(txn_id_t txn_id, KeyType
  ins_key, ValType ins_val) {
2   lsn_t prev_lsn;
3
   if(LogRec::prev_lsn_map_.find(txn_id)==LogRec::prev_lsn_map_.en
  d()) {
4     prev_lsn = INVALID_LSN ;
```

```
    LogRec::prev_lsn_map_.emplace(txn_id,LogRec::next_lsn_);
  }else{
    prev_lsn = LogRec::prev_lsn_map_[txn_id];
    LogRec::prev_lsn_map_[txn_id] = LogRec::next_lsn_;
  }
  LogRec log =
LogRec(LogRecType::kInsert,LogRec::next_lsn_++,prev_lsn,txn_id);
  log.ins_key_ = std::move(ins_key);
  log.ins_val_ = ins_val;
  return std::make_shared<LogRec>(log);
}
```

插入日志：

1. 首先在全部日志中查找是否存在该事务的其他日志：
    - 如果没有找到，说明该日志是该事务的第一条，这种条件下，这条日志的`prev_lsn`应该是`INVALID_LSN`，标志事务的开始。此时向`prev_lsn_map_`应该是插入而不是更新元素`(txn_id,LogRec::next_lsn_)`
    - 如果找到，说明该日志不是该事务的第一条，这种条件下，这条日志的`prev_lsn`应该是上一条该事务的日志的`lsn`，标志承接上一条日志，属于同一事务。此时`prev_lsn_map_`应该更新元素`(txn_id,LogRec::next_lsn_)`
2. 利用我们编写好的LogRec赋值函数去更新该日志的相关信息（包括事务状态类型、`lsn`、`prev_lsn`、`txn_id`）：

```
LogRec(LogRecType type,lsn_t lsn,lsn_t prev_lsn,txn_id_t
txn_id):

 type_(type),lsn_(lsn),prev_lsn_(prev_lsn),txn_id_(txn_id
)
  {}
```

3. 更新日志应该保存的键值对，这里保存插入的位置和新值，`RollBack`要将该键值对删除

CreateDeleteLog

函数实现：

```
static LogRecPtr CreateDeleteLog(txn_id_t txn_id, KeyType
del_key, ValType del_val) {
  lsn_t prev_lsn;

 if(LogRec::prev_lsn_map_.find(txn_id)==LogRec::prev_lsn_map_.en
d()) {
    prev_lsn = INVALID_LSN ;
    LogRec::prev_lsn_map_.emplace(txn_id,LogRec::next_lsn_);
  }else{
    prev_lsn = LogRec::prev_lsn_map_[txn_id];
    LogRec::prev_lsn_map_[txn_id] = LogRec::next_lsn_;
  }
  LogRec log =
LogRec(LogRecType::kDelete,LogRec::next_lsn_++,prev_lsn,txn_id);
  log.del_key_ = std::move(del_key);
  log.del_val_ = del_val;
  return std::make_shared<LogRec>(log);
}
```

删除日志：

1. 首先在全部日志中查找是否存在该事务的其他日志：
    - 如果没有找到，说明该日志是该事务的第一条，这种条件下，这条日志的`prev_lsn`应该是`INVALID_LSN`，标志事务的开始。此时向`prev_lsn_map_`应该是插入而不是更新元素`(txn_id,LogRec::next_lsn_)`
    - 如果找到，说明该日志不是该事务的第一条，这种条件下，这条日志的`prev_lsn`应该是上一条该事务的日志的`lsn`，标志承接上一条日志，属于同一事务。此时`prev_lsn_map_`应该更新元素`(txn_id,LogRec::next_lsn_)`
2. 利用我们编写好的LogRec赋值函数去更新该日志的相关信息（包括事务状态类型、`lsn`、`prev_lsn`、`txn_id`）：

```
LogRec(LogRecType type,lsn_t lsn,lsn_t prev_lsn,txn_id_t
txn_id):

 type_(type),lsn_(lsn),prev_lsn_(prev_lsn),txn_id_(txn_id
)
  {}
```

3. 更新日志应该保存的键值对，这里保存删除的位置和旧值，`RollBack`要将该键值对加进去

CreateUpdateLog

函数实现：

```
static LogRecPtr CreateUpdateLog(txn_id_t txn_id, KeyType
old_key, ValType old_val, KeyType new_key, ValType new_val) {
  lsn_t prev_lsn;

 if(LogRec::prev_lsn_map_.find(txn_id)==LogRec::prev_lsn_map_.en
d()) {
    prev_lsn = INVALID_LSN ;
    LogRec::prev_lsn_map_.emplace(txn_id,LogRec::next_lsn_);
  }else{
    prev_lsn = LogRec::prev_lsn_map_[txn_id];
    LogRec::prev_lsn_map_[txn_id] = LogRec::next_lsn_;
  }
  LogRec log =
LogRec(LogRecType::kUpdate,LogRec::next_lsn_++,prev_lsn,txn_id);
  log.old_key_ = std::move(old_key);
  log.new_key_ = std::move(new_key);
  log.new_val_ = new_val;
  log.old_val_ = old_val;
  return std::make_shared<LogRec>(log);
}
```

更新日志：

1. 首先在全部日志中查找是否存在该事务的其他日志：

- 如果没有找到，说明该日志是该事务的第一条，这种条件下，这条日志的`prev_lsn`应该是`INVALID_LSN`，标志事务的开始。此时向`prev_lsn_map_`应该是插入而不是更新元素`(txn_id,LogRec::next_lsn_)`
- 如果找到，说明该日志不是该事务的第一条，这种条件下，这条日志的`prev_lsn`应该是上一条该事务的日志的`lsn`，标志承接上一条日志，属于同一事务。此时`prev_lsn_map_`应该更新元素`(txn_id,LogRec::next_lsn_)`

2. 利用我们编写好的LogRec赋值函数去更新该日志的相关信息（包括事务状态类型、`lsn`、`prev_lsn`、`txn_id`）：

```
LogRec(LogRecType type,lsn_t lsn,lsn_t prev_lsn,txn_id_t txn_id):

 type_(type),lsn_(lsn),prev_lsn_(prev_lsn),txn_id_(txn_id)
  {}
```

3. 更新日志应该保存的键值对，这里保存更新的位置和新旧值，`RollBack`要将该键值对更换为之前的旧值

CreateBeginLog

函数实现：

```
static LogRecPtr CreateBeginLog(txn_id_t txn_id){
  lsn_t prev_lsn;

 if(LogRec::prev_lsn_map_.find(txn_id)==LogRec::prev_lsn_map_.end()) {
    prev_lsn = INVALID_LSN ;
    LogRec::prev_lsn_map_.emplace(txn_id,LogRec::next_lsn_);
  }else{
    prev_lsn = LogRec::prev_lsn_map_[txn_id];
    LogRec::prev_lsn_map_[txn_id] = LogRec::next_lsn_;
  }
  LogRec log = LogRec(LogRecType::kBegin,LogRec::next_lsn_++,prev_lsn,txn_id);
  return std::make_shared<LogRec>(log);
}
```

开始日志：

1. 首先在全部日志中查找是否存在该事务的其他日志：
    - 如果没有找到，说明该日志是该事务的第一条，这种条件下，这条日志的`prev_lsn`应该是`INVALID_LSN`，标志事务的开始。此时向`prev_lsn_map_`应该是插入而不是更新元素`(txn_id,LogRec::next_lsn_)`
    - 如果找到，说明该日志不是该事务的第一条，这种条件下，这条日志的`prev_lsn`应该是上一条该事务的日志的`lsn`，标志承接上一条日志，属于同一事务。此时`prev_lsn_map_`应该更新元素`(txn_id,LogRec::next_lsn_)`
2. 利用我们编写好的LogRec赋值函数去更新该日志的相关信息（包括事务状态类型、`lsn`、`prev_lsn`、`txn_id`）：

```
1  LogRec(LogRecType type,lsn_t lsn,lsn_t prev_lsn,txn_id_t
   txn_id):
2
    type_(type),lsn_(lsn),prev_lsn_(prev_lsn),txn_id_(txn_id
   )
3    {}
```

CreateCommitLog

函数实现：

```
1  static LogRecPtr CreateCommitLog(txn_id_t txn_id){
2    lsn_t prev_lsn;
3
    if(LogRec::prev_lsn_map_.find(txn_id)==LogRec::prev_lsn_map_.en
   d()) {
4      prev_lsn = INVALID_LSN ;
5      LogRec::prev_lsn_map_.emplace(txn_id,LogRec::next_lsn_);
6    }else{
7      prev_lsn = LogRec::prev_lsn_map_[txn_id];
8      LogRec::prev_lsn_map_[txn_id] = LogRec::next_lsn_;
9    }
10   LogRec log =
   LogRec(LogRecType::kCommit,LogRec::next_lsn_++,prev_lsn,txn_id);
11   return std::make_shared<LogRec>(log);
12 }
```

提交日志：

1. 首先在全部日志中查找是否存在该事务的其他日志：
    - 如果没有找到，说明该日志是该事务的第一条，这种条件下，这条日志的`prev_lsn`应该是`INVALID_LSN`，标志事务的开始。此时向`prev_lsn_map_`应该是插入而不是更新元素`(txn_id,LogRec::next_lsn_)`
    - 如果找到，说明该日志不是该事务的第一条，这种条件下，这条日志的`prev_lsn`应该是上一条该事务的日志的`lsn`，标志承接上一条日志，属于同一事务。此时`prev_lsn_map_`应该更新元素`(txn_id,LogRec::next_lsn_)`
2. 利用我们编写好的LogRec赋值函数去更新该日志的相关信息（包括事务状态类型、`lsn`、`prev_lsn`、`txn_id`）：

```
1 LogRec(LogRecType type,lsn_t lsn,lsn_t prev_lsn,txn_id_t
  txn_id):
2
   type_(type),lsn_(lsn),prev_lsn_(prev_lsn),txn_id_(txn_id
  )
3    {}
```

CreateAbortLog

函数实现：

```
1  static LogRecPtr CreateAbortLog(txn_id_t txn_id) {
2    lsn_t prev_lsn;
3
   if(LogRec::prev_lsn_map_.find(txn_id)==LogRec::prev_lsn_map_.en
  d()) {
4      prev_lsn = INVALID_LSN ;
5      LogRec::prev_lsn_map_.emplace(txn_id,LogRec::next_lsn_);
6    }else{
7      prev_lsn = LogRec::prev_lsn_map_[txn_id];
8      LogRec::prev_lsn_map_[txn_id] = LogRec::next_lsn_;
9    }
10   LogRec log =
  LogRec(LogRecType::kAbort,LogRec::next_lsn_++,prev_lsn,txn_id);
11   return std::make_shared<LogRec>(log);
12 }
```

取消日志：

1. 首先在全部日志中查找是否存在该事务的其他日志：
    - 如果没有找到，说明该日志是该事务的第一条，这种条件下，这条日志的`prev_lsn`应该是`INVALID_LSN`，标志事务的开始。此时向`prev_lsn_map_`应该是插入而不是更新元素`(txn_id,LogRec::next_lsn_)`
    - 如果找到，说明该日志不是该事务的第一条，这种条件下，这条日志的`prev_lsn`应该是上一条该事务的日志的`lsn`，标志承接上一条日志，属于同一事务。此时`prev_lsn_map_`应该更新元素`(txn_id,LogRec::next_lsn_)`
2. 利用我们编写好的LogRec赋值函数去更新该日志的相关信息（包括事务状态类型、`lsn`、`prev_lsn`、`txn_id`）：

```
1 LogRec(LogRecType type,lsn_t lsn,lsn_t prev_lsn,txn_id_t
  txn_id):
2
   type_(type),lsn_(lsn),prev_lsn_(prev_lsn),txn_id_(txn_id
  )
3    {}
```

3. 对于该类事务返回，让上一层函数直接`RollBack`，不解释连招

recovery_manager.h相关

出于实现复杂度的考虑，我们将Recovery Manager模块独立出来，不考虑日志的落盘，用一个unordered_map简易的模拟一个KV Database，并直接在内存中定义一个能够用于插入、删除、更新，事务的开始、提交、回滚的日志结构。`CheckPoint`检查点应包含当前数据库一个完整的状态，该结构已帮大家实现好了。`RecoveryManager`则包含`UndoPhase`和`RedoPhase`两个函数，代表Redo和Undo两个阶段。

`LSN`称为日志的逻辑序列号(log sequence number)，在InnoDB存储引擎中，`LSN`占用8个字节。`LSN`的值会随着日志的写入而逐渐增大。

CheckPoint结构体

```
using KvDatabase = std::unordered_map<KeyType, ValType>;
using ATT = std::unordered_map<txn_id_t, lsn_t>;
struct CheckPoint {
    lsn_t checkpoint_lsn_{INVALID_LSN};
    ATT active_txns_{};
    KvDatabase persist_data_{};

    inline void AddActiveTxn(txn_id_t txn_id, lsn_t last_lsn) { active_txns_[txn_id] = last_lsn; }

    inline void AddData(KeyType key, ValType val) { persist_data_.emplace(std::move(key), val); }
};
```

数据类型	变量名称	含义
lsn_t	checkpoint_lsn_	`checkpoint`处的`LSN`值，此点及以后的`log`需要被处理
std::unordered_map<txn_id_t, lsn_t>	active_txns_	活跃事务`map`，与`txn_id`绑定，表示现在还在活跃的事务
std::unordered_map<KeyType, ValType>	persist_data_	`KVdatabase:value`与`key`绑定，表示数据库中现在已存的键值对

RecoveryManager类

本类实现了数据恢复的全过程，包括`Redo`,`Undo`两个阶段

Init

函数实现：

```
void Init(CheckPoint &last_checkpoint) {
      persist_lsn_ = last_checkpoint.checkpoint_lsn_;
      active_txns_ = std::move(last_checkpoint.active_txns_);
      data_ = std::move(last_checkpoint.persist_data_);
    }
```

初始化状态：

- 更新现在的执行的日志号为`checkpoint`的日志号，标识`Redo`的作用开始序列号为此
- 更新现在的活跃的事务`ATT`，利用`move`直接移送所有权，省去中间变量的生成
- 更新数据库中的键值对，恢复到`CheckPoint`时的数据

RedoPhase

函数实现：

```
void RedoPhase() {
  for(auto log = log_recs_[persist_lsn_];
      ;log = log_recs_[++persist_lsn_]){
    //parse the log
      active_txns_[log->txn_id_] = log->lsn_;
    switch(log->type_){
      case(LogRecType::kInvalid):
        break;
      case(LogRecType::kInsert):
          data_[log->ins_key_] = log->ins_val_;
        break;
      case(LogRecType::kDelete):
          data_.erase(log->del_key_);
        break;
      case(LogRecType::kUpdate):
        data_.erase(log->old_key_);
        data_[log->new_key_]= log->new_val_;
        break;
      case(LogRecType::kBegin):
        break;
      case(LogRecType::kCommit):
          active_txns_.erase(log->txn_id_);
        break;
      case(LogRecType::kAbort):
          for(auto log_roll = log_recs_[active_txns_[log-
>txn_id_]];
              ;log_roll = log_recs_[log_roll->prev_lsn_] )
          {
            switch (log_roll->type_) {
              case(LogRecType::kInsert):
```

```
                    data_.erase(log_roll->ins_key_);
                  break;
                case(LogRecType::kDelete):
                  data_.emplace(log_roll->del_key_,log_roll->del_val_);
                  break;
                case (LogRecType::kUpdate):
                  data_.erase(log_roll->new_key_);
                  data_.emplace(log_roll->old_key_,log_roll->old_val_);
                  break;
                default:
                  break;
              }
              if(log_roll->prev_lsn_ == INVALID_LSN){
                break;
              }
            }
            active_txns_.erase(log->txn_id_);
          break;
        default:
          break;
      }
      if(persist_lsn_>=(lsn_t)(log_recs_.size()-1)){
        break;
      }
    }
}
```

重做事务：

从`persist_lsn_`开始遍历`lsn`直到做完所有事务

- 如果日志是非`Abort`的，那么按照日志本来的类型去做即可，大部分都是做到更改他的键值对
- 如果是`Abort`的，则需要根据现在的`LogRec`要回滚回去直至`Invaild`，做一次回滚操作，将旧值替换回去即可(这里介绍的较为笼统)

UndoPhase

函数实现：

```
void UndoPhase() {
  //abort all the active
  for(auto pair:active_txns_){
    for(auto log_roll = log_recs_[pair.second];
       ;log_roll = log_recs_[log_roll->prev_lsn_] )
    {
      if(log_roll == nullptr){
        break;
      }
      switch (log_roll->type_) {
        case(LogRecType::kInsert):
          data_.erase(log_roll->ins_key_);
          break;
        case(LogRecType::kDelete):
          data_.emplace(log_roll->del_key_,log_roll->del_val_);
          break;
        case (LogRecType::kUpdate):
          data_.erase(log_roll->new_key_);
          data_.emplace(log_roll->old_key_,log_roll->old_val_);
          break;
        default:
          break;
      }
      if(log_roll->prev_lsn_ == INVALID_LSN){
        break;
      }
    }
  }
  active_txns_.clear();
}
```

Undo事务：

对于每个活跃事务进行遍历：

- 找到活跃事务的最新日志，进行类似于`RollBack`的操作
- 直至找到`INVALID_LSN`代表该事务`Undo`成功

遇到的问题及解决方法

这次的数据库设计并没有引入事务模块，也就是我们的本次恢复模块相较于加入事务的数据库较为容易，测试较为独立，Debug较为顺利，现给出我遇到的某些问题和解决方法：

- 本次数据库模块设计出现了共享指针，要熟悉共享指针的使用。参考了以下博客：C++ 智能指针(共享指针、唯一指针、自动指针)-CSDN博客，便于理解。
- 在更新操作时要使用`erase`和`emplace`搭配的方法，用`[]`是过不了测试点的，这是个奇怪的问题，大概率是和他们对于`map`的底层操作有些许不同导致的

思考题

> *本模块中，为了简化实验难度，我们将Recovery Manager模块独立出来。如果不独立出来，真正做到数据库在任何时候断电都能恢复，同时支持事务的回滚，Recovery Manager应该怎样设计呢？此外，CheckPoint机制应该怎样设计呢？*
>
> *注：如果完成了本模块，请在实验报告里完成思考题。思考题占本模块30%的分数，请尽量回答的详细些，比如具体到涉及哪些模块、哪些函数的改动，大致怎样改动。有能力、有时间的同学也可以挑战一下直接在代码上更改。*

Recovery Manager的设计：

- 由于这个模块的思路已经和数据恢复很契合了，就是通过日志来实现故障恢复，周期性地执行建立检查点和保存数据库状态。
- 当事务T在一个检查点之前提交，T对数据库所做的修改已写入数据库写入时间是在这个检查点建立之前或在这个检查点建立之时，在进行恢复处理时，没有必要对事务T执行重做操作。

    a. 从重新开始文件中找到最后一个检查点记录在日志文件中的地址，由该地址在日志文件中找到最后一个检查点记录。

    b. 由该检查点记录得到检查点建立时刻所有正在执行的事务清单ACTIVE-LIST。

        - 建立两个事务队列：UNDO-LIST和REDO-LIST。
        - 把ACTIVE-LIST暂时放入UNDO-LIST队列，REDO队列暂为空。

    c. 从检查点开始正向扫描日志文件，直到日志文件结束。

    - 如有新开始的事务 Ti ，把 Ti 暂时放入UNDO-LIST队列。
    - 如有提交的事务 Tj ，把 Tj 从UNDO-LIST队列移到REDO-LIST队列;直到日志文件结束。

    d. 对UNDO-LIST中的每个事务执行UNDO操作，对REDO-LIST中的每个事务执行REDO操作。

- 只需要保证在每个事务执行的时候都在通过日志维护执行顺序，并且保证检查点和数据保存同时写入磁盘，这样可以保证数据的一致性，便于我们的断电恢复。
- 另外，事务的回滚和Undo的操作是一样的，只需要将事务的日志“补偿”地反方向执行一次，即可得到事务刚开始的数据，达成事务回滚的操作。
- 总之大概率是不用改动此模块的，由于事务模块还没有实现，这里实现的恢复模块只能对串行化事务进行检查点策略的执行表现良好，当接入事务模块时，大概是把恢复日志管理与事务执行策略并行同步执行，另外提供回滚的接口给事务模块即可。

CheckPoint机制：

- 周期性地执行建立检查点和保存数据库状态。
    a. 将当前日志缓冲区中的所有日志记录写入磁盘的日志文件上
    b. 在日志文件中写入一个检查点记录
    c. 将当前数据缓冲区的所有数据记录写入磁盘的数据库中
    d. 把检查点记录在日志文件中的地址写入一个重新开始文件
- 恢复子系统可以定期或不定期地建立检查点,保存数据库状态，最好是采用不定期策略，因为定期的策略不容易控制每次写入磁盘的量，可能单次写入磁盘的操作用时很长，造成数据库的暂时不可用，可以根据日志现在的存储量来实行检查点策略。
- 操作的代码地方应该是在事务模块中，与写入日志的地方一致，统计写入日志的数量来做到不定期check的操作

总结、心得

不知不觉已经写到这里了，每一次打开Clion的时候都不知道自己能写到哪里，好在我有可靠的小组成员和我一起Debug、有悉心指导的聂俊哲&石宇新助教，在数据库学习中感谢孙建伶老师的悉心教导，模块6就写到这里，是一次酣畅淋漓的C++工程管理和数据库学习。

今天是6月8日高考第二日，祝全国考生以及母校平遥中学的学弟学妹们高考顺利，前程似锦！

春风得意马蹄疾，一日看尽长安花！