Miniob Storage Overview

以下默认全部在 storage 模块文件夹下，如果跨出去了会额外说。

对于值的更新之类的需要去 observer 下面 common 下面的 value 文件修改

注意下面并不是给出了一个类的所有接口函数，如果发现要用但是没有还是需要移步类文件查看

DB

总体 db.h 是最上层的，代表一个数据库对象，内部变量：

```
  string                        name_;                ///< 数据库名称
  string                        path_;                ///< 数据库文件存放的目录
  unordered_map<string, Table *> opened_tables_;      ///< 当前所有打开的表
  unique_ptr<BufferPoolManager>  buffer_pool_manager_; ///< 当前数据库的buffer pool管理器
  unique_ptr<LogHandler>         log_handler_;         ///< 当前数据库的日志处理器
  unique_ptr<TrxKit>             trx_kit_;             ///< 当前数据库的事务管理器

  /// 给每个table都分配一个ID，用来记录日志。这里假设所有的DDL都不会并发操作，所以相关的数据都不上锁
  int32_t next_table_id_ = 0;

  LSN check_point_lsn_ = 0;  ///< 当前数据库的检查点LSN。会记录到磁盘中。
```

比较重要的接口：

```
  /**
   * @brief 根据表名查找表
   */
  Table *find_table(const char *table_name) const;
  /**
   * @brief 根据表ID查找表
```

```
   */
  Table *find_table(int32_t table_id) const;
```

还有一些像是 `create_table, drop_table` 之类的接口，不过不太用得到，主要是用于调用找到 table 去执行 table 里面的操作的。

Table

接下来就是 `Table` 类，table 层面中含 table_meta 和 table

先说 `TableMeta` 类，table_meta 中是操作和读取元数据，一些重要的变量是：

```
  int32_t                table_id_ = -1;  // 表的唯一 id
  std::string            name_;  // 表的名字
  std::vector<FieldMeta> trx_fields_; // 存储事务字段
  std::vector<FieldMeta> fields_;  // 存储所有字段的信息，包括系统字段。
  std::vector<IndexMeta> indexes_; // 存储表的所有索引信息
  StorageFormat          storage_format_; // 表的数据存储格式
```

相对重要的接口函数是：

```
RC add_index(const IndexMeta &index);  // 添加一个新的索引到表中
int32_t             table_id() const { return table_id_; } // 返回表的唯一
id
  const char         *name() const;  // 返回表名称
  const FieldMeta    *trx_field() const; // 返回事务字段元数据
  const FieldMeta    *field(int index) const; // 返回指定下标的字段元数据
  const FieldMeta    *field(const char *name) const; // 返回指定名称的字段
元数据
  const FieldMeta    *find_field_by_offset(int offset) const; // 返回指定
偏移的字段元数据
  auto                field_metas() const -> const std::vector<FieldMeta>
*{ return &fields_; }  //返回所有的字段元数据列表
  auto                trx_fields() const -> std::span<const FieldMeta>; //
返回事务字段的元数据列表
  const StorageFormat storage_format() const { return storage_format_; }
//返回存储格式

  int field_num() const;  // sys field included

  const IndexMeta *index(const char *name) const;  // 根据名称返回索引的元
数据
```

```
  const IndexMeta *find_index_by_field(const char *field) const; //根据字段名称返回索引的元数据
  const IndexMeta *index(int i) const; // 根据下标返回索引的元数据
  int              index_num() const; // 返回表中索引的数量

  int record_size() const; //返回表中记录的大小
```

再说 `Table` 类:

```
  Db                *db_ = nullptr;  // 指向所属 db 类对象的指针
  string             base_dir_; // 存储表数据文件的基目录路径，用于确定表数据文件的存储位置
  TableMeta          table_meta_; // 存储表的元数据信息
  DiskBufferPool    *data_buffer_pool_ = nullptr;  /// 指向表数据文件关联的 buffer pool
  RecordFileHandler *record_handler_   = nullptr;  /// 指向记录操作对象，用于处理表中的记录插入、删除、更新等操作，table的删除插入等都会调用 RecordFileHandler 的插入删除更新
  vector<Index *>    indexes_; // 存储表中所有索引对象的指针，用于管理和操作表的索引，支持高效的查询和检索
```

相对重要的接口函数为:

```
RC create(Db *db, int32_t table_id, const char *path, const char *name, const char *base_dir,
      span<const AttrInfoSqlNode> attributes, StorageFormat storage_format);
// 创建一个表
RC drop_table(const char *table_name) ; // 删除一个表
RC open(Db *db, const char *meta_file, const char *base_dir); //打开一个表，meta_file 保存表元数据的文件，比如 table.meta，basedir 是表文件的基目录路径
RC make_record(int value_num, const Value *values, Record &record);
// 根据给定的字段生成一个记录，value_num 字段的个数，values 每个字段的值，record    生成的记录数据
RC insert_record(Record &record); // 在当前的表中插入一条记录，索引中也一并插入
RC delete_record(const Record &record); // 表中删除一条记录，索引中一并删去
RC get_record(const RID &rid, Record &record); // 根据 rid 获取对应的 record
RC create_index(Trx *trx, const FieldMeta *field_meta, const char *index_name);
```

```
// 表上创建一个新的索引
RC get_record_scanner(RecordFileScanner &scanner, Trx *trx, ReadWriteMode
mode); // 函数通过初始化并返回一个记录扫描器
RC get_chunk_scanner(ChunkFileScanner &scanner, Trx *trx, ReadWriteMode
mode);
// 函数通过初始化返回一个 chunk 记录扫描器
RecordFileHandler *record_handler() const { return record_handler_; }
RC visit_record(const RID &rid, function<bool(Record &)> visitor);
// 可以在页面锁保护的情况下访问记录
RC Table::sync();  // flush
Index *find_index(const char *index_name) const;
Index *find_index_by_field(const char *field_name) const;
```

Field

可以看见上面调用了 `FieldMeta` 类，这个部分来源于 `storage/field/field_meta.h` ，这个是字段的定义，也就是列

```
  string   name_;  // 字段的名称
  AttrType attr_type_; // 字段的类型
  int      attr_offset_;  // 字段在一个 tuple 里面的偏移
  int      attr_len_;  // 字段占位长度
  bool     visible_; // 字段是否可见
  int      field_id_; // 字段的唯一标识符，通常表示是第几个
```

下面是一个示例：

```
// users 表，字段为 id,name,email,age
std::vector<FieldMeta> fields = {
    {"id", AttrType::INT, 0, sizeof(int), true, 1},
    {"name", AttrType::VARCHAR, sizeof(int), 100, true, 2},
    {"email", AttrType::VARCHAR, sizeof(int) + 100, 150, true, 3},
    {"age", AttrType::INT, sizeof(int) + 100 + 150, sizeof(int), true, 4}
};
```

比较重要的接口：

```
  const char *name() const; // 获取字段名称
  AttrType    type() const; // 获取类型
  int         offset() const;  // 获取偏移
  int         len() const;   // 获取长度
```

```
  bool        visible() const;  // 获取是否可见
  int         field_id() const;
  void      to_json(Json::Value &json_value) const; // 可以将当前 filed 转
换成一个 json value
  static RC from_json(const Json::Value &json_value, FieldMeta &field); //
从 json value 解析回 filed
```

接下来是 `Field` 类，变量为：

```
  const Table     *table_ = nullptr;  // 字段所属的表的指针
  const FieldMeta *field_ = nullptr;  // 字段的元信息指针
```

一些重要的接口函数：

```
const char *get_data(const Record &record); // 表示获取这个表中传入的这个
record 指定字段的值，以 char* 指针返回
void set_int(Record &record, int value);  // 用于将整数值设置到 record 的指
定字段中,注意这里会检查为 int类型以及长度是否合法
int  get_int(const Record &record); //用于从 record 的指定字段中获取整数值
void set_table(const Table *table) { this->table_ = table; }
void set_field(const FieldMeta *field) { this->field_ = field; }
```

Record

这里面首先有 RID 类，里面维护一个 page num 和一个 slot num，表示 record 在这个文件的第几个 page 的第几个 slot，函数接口有：

```
/**
   * 返回一个不可能出现的最小的RID
   * 虽然page num 0和slot num 0都是合法的，但是page num 0通常用于存放meta数
据，所以对数据部分来说都是
   * 不合法的. 这里在bplus tree中查找时会用到。
   */
  static RID *min()
  {
    static RID rid{0, 0};
    return &rid;
  }

  /**
   * @brief 返回一个“最大的”RID
```

```
   * 我们假设page num和slot num都不会使用对应数值类型的最大值
   */
  static RID *max()
  {
    static RID rid{numeric_limits<PageNum>::max(),
numeric_limits<SlotNum>::max()};
    return &rid;
  }
```

之后就是整个 record 的结构，`Record` 类：

```
  RID rid_; // rid 定位
  char *data_  = nullptr; // 所存储的数据
  int   len_   = 0;       /// 如果不是record自己来管理内存，这个字段可能是无效
的
  bool  owner_ = false;  /// owner_成员变量用来指示该对象是否拥有其指向的数据
(data_)的生命周期管理权
```

一些重要的函数：

```
  void set_data(char *data, int len = 0)
  {
    this->data_ = data;
    this->len_  = len;
  }
  void set_data_owner(char *data, int len)
  {
    ASSERT(len != 0, "the len of data should not be 0");
    this->~Record();
    this->data_  = data;
    this->len_   = len;
    this->owner_ = true;
  }
  RC copy_data(const char *data, int len)
  {
    ASSERT(len!= 0, "the len of data should not be 0");
    char *tmp = (char *)malloc(len);
    if (nullptr == tmp) {
      LOG_WARN("failed to allocate memory. size=%d", len);
      return RC::NOMEM;
    }
    memcpy(tmp, data, len);
```

```
    set_data_owner(tmp, len);
    return RC::SUCCESS;
  }
  RC new_record(int len)
  {
    ASSERT(len!= 0, "the len of data should not be 0");
    char *tmp = (char *)malloc(len);
    if (nullptr == tmp) {
      LOG_WARN("failed to allocate memory. size=%d", len);
      return RC::NOMEM;
    }
    set_data_owner(tmp, len);
    return RC::SUCCESS;
  }
  RC set_field(int field_offset, int field_len, char *data)
  {
    if (!owner_) {
      LOG_ERROR("cannot set field when record does not own the memory");
      return RC::INTERNAL;
    }
    if (field_offset + field_len > len_) {
      LOG_ERROR("invalid offset or length. offset=%d, length=%d, total
length=%d", field_offset, field_len, len_);
      return RC::INVALID_ARGUMENT;
    }
    memcpy(data_ + field_offset, data, field_len);
    return RC::SUCCESS;
  }
  void set_rid(const RID &rid) { this->rid_ = rid; }
  char       *data() { return this->data_; }
  RID       &rid() { return rid_; }
```

接下来就是 record manager 文件里面的定义，首先会明确磁盘中除了 page0 之外，每一个 page 的 header，也就是 PageHeader 类：

```
struct PageHeader
{
  int32_t record_num;        ///< 当前页面记录的个数
  int32_t column_num;        ///< 当前页面记录所包含的列数
  int32_t record_real_size;  ///< 每条记录的实际大小
  int32_t record_size;       ///< 每条记录占用实际空间大小(可能对齐)
  int32_t record_capacity;   ///< 最大记录个数
  int32_t col_idx_offset;    ///< 列索引偏移量
```

```
  int32_t data_offset;        ///< 第一条记录的偏移量

  string to_string() const;
};
```

之后会有一个 RecordPageIterator 类，这是遍历一个页面中每条记录的 iterator，下面是类里面的变量的定义：

```
  RecordPageHandler *record_page_handler_ = nullptr; //负责处理一个页面中各种操作，比如插入记录、删除记录或者查找记录
  PageNum            page_num_            = BP_INVALID_PAGE_NUM;
  common::Bitmap     bitmap_;             ///< bitmap 的相关信息可以参考 RecordPageHandler 的说明
  SlotNum            next_slot_num_ = 0;  ///< 当前遍历到了哪一个slot
```

重要的函数为：

```
void init(RecordPageHandler *record_page_handler, SlotNum start_slot_num = 0);
// 初始化，表示从某个页面的哪个记录开始扫描，默认为 0
bool has_next(); //判断是否有下一个记录
RC next(Record &record); // 读取下一个记录到record中包括RID和数据，并更新下一个记录位置next_slot_num_
bool is_valid() const { return record_page_handler_ != nullptr; } // 判断是否合法
```

之后就是 RecordPageHandler 类，是一个基类，下面是它里面的定义：

```
  DiskBufferPool  *disk_buffer_pool_ = nullptr;  ///< 当前操作的buffer pool(文件)
  RecordLogHandler log_handler_;                 ///< 当前操作的日志处理器
  Frame *frame_ = nullptr;  ///< 当前操作页面关联的frame(frame的更多概念可以参考buffer pool和frame)
  ReadWriteMode rw_mode_     = ReadWriteMode::READ_WRITE;  ///< 当前的操作是否都是只读的
  PageHeader    *page_header_ = nullptr;                   ///< 当前页面上页面头
  char          *bitmap_      = nullptr;  ///< 当前页面上record分配状态信息bitmap内存起始位置
  StorageFormat storage_format_; // 存储格式
```

都是重要的函数，均已经有详细注释。

紧接着还有 RowRecordPageHandler 类，是 RecordPageHandler 的派生类，继承于它，表示为行存储。主要是针对行存，单独实现了 insert, delete, update 以及 get_record

还有负责处理 PAX 存储格式的页面中各种操作的，但是目前可能只用关注 Row 就行了。

之后就是 record_log.h，也就是 RecordLogHandler 类：

```
  LogHandler   *log_handler_    = nullptr;
  int32_t       buffer_pool_id_ = -1;
  int32_t       record_size_    = -1;
  StorageFormat storage_format_ = StorageFormat::ROW_FORMAT;
```

一些重要的函数接口都有详细注释在代码中。

Common

首先有 ConDesc 类，用于描述一个被比较的对象是什么形式的：

```
struct ConDesc
{
  bool  is_attr;      // 是否属性，false 表示是值
  int   attr_length;  // 如果是属性，表示属性值长度
  int   attr_offset;  // 如果是属性，表示在记录中的偏移量
  Value value;        // 如果是值类型，这里记录值的数据
};
```

DefaultConditionFilter：

```
  ConDesc  left_;
  ConDesc  right_;
  AttrType attr_type_ = AttrType::UNDEFINED;
  CompOp   comp_op_   = NO_OP;
  // A?B 形式这种的
```

重要的函数接口是 `filter`，它可以判断一个 record 是否符合这个条件，满足他的某个字段=一个固定的值之类的。

```
virtual bool filter(const Record &rec) const;
```

CompositeConditionFilter 相当于是多个条件的集合判断：

```
  const ConditionFilter **filters_      = nullptr;
  int                    filter_num_    = 0;
  bool                   memory_owner_ = false;  // filters_的内存是否由自
己来控制
```

这里的 filter 可以一次性判断多个条件，使用方式一样。

meta_util 没什么可说的，这一部分是拿来帮助写路径的，比如自动帮你加一个 `/.table,/.data` 这种的

之后是下面的 column 类和 chunk 类，column 就是字面定义，用来存储同一字段的信息。

chunk 就是一堆 column 组成，并且确保 column 的 size 等都要相同，从这个角度来看，chunk 就是一堆 record 不过按照列存的形式。

重要的函数接口，chunk 里面：

```
  Column *column_ptr(size_t idx) //返回第几列的指针
  {
    ASSERT(idx < columns_.size(), "invalid column index");
    return &column(idx);
  }
  int column_ids(size_t i) //返回对应列的唯一 id
  {
    ASSERT(i < column_ids_.size(), "invalid column index");
    return column_ids_[i];
  }
  Value get_value(int col_idx, int row_idx) const { return
columns_[col_idx]->get_value(row_idx); } //从 Chunk 中获得指定行指定列的
Value
```

column 里面：

```
Value get_value(int index) const;  // 获取 index 位置的列值
RC append(char *data, int count); // 向 Column 追加写入数据，count 要写入数
据的长度（这里指列值的个数，而不是字节），因为同一列中类型相等，所以值所占的字节
大小是已知的
```

Buffer

接下来就是比较关键的，也就是 buffer

画个结构图在这。

Page

```
struct Page
{
  LSN      lsn; // log sequence number
  CheckSum check_sum;
  char     data[BP_PAGE_DATA_SIZE];
  // data 数组中的前一部分存储了 BPFILEHEADER 结构，后一部分存储了数据
};
```

Frame

```
  friend class BufferPool;

  bool          dirty_ = false; // 判断是否为脏页
  atomic<int>   pin_count_{0}; // 原子整数，确保在多线程环境中线程安全地访问
和修改共享数据
  unsigned long acc_time_ = 0;  // 记录页面的访问时间，帮助实现 lru
  FrameId       frame_id_; /*
  buffer_pool_id
  page_num
*/
  Page          page_;

  /// 在非并发编译时，加锁解锁动作将什么都不做
  common::RecursiveSharedMutex lock_;

  /// 使用一些手段来做测试，提前检测出头疼的死锁问题
  /// 如果编译时没有增加调试选项，这些代码什么都不做
  common::DebugMutex           debug_lock_;
  intptr_t                     write_locker_          = 0;
  int                          write_recursive_count_ = 0;
  unordered_map<intptr_t, int> read_lockers_;
```

常用的函数接口：

```
int  buffer_pool_id() const { return frame_id_.buffer_pool_id(); }
void set_buffer_pool_id(int id) { frame_id_.set_buffer_pool_id(id); }
Page &page() { return page_; }
PageNum page_num() const { return frame_id_.page_num(); }
void access(); //更新当前内存页面的访问时间
LSN  lsn() const { return page_.lsn; }
void set_lsn(LSN lsn) { page_.lsn = lsn; }
void mark_dirty() { dirty_ = true; }
void clear_dirty() { dirty_ = false; }
bool dirty() const { return dirty_; }
char *data() { return page_.data; }
bool can_purge() { return pin_count_.load() == 0; }
void pin();
int unpin();
void write_latch(); //获取写锁，确保独占访问
void write_unlatch(); //释放写锁，允许其他线程访问
void read_latch(); // 获取读锁，允许多个线程同时读取
void read_unlatch(); // 释放读锁，允许其他线程读取
```

LruCache

也就是我们所使用的 lru，我们的 BPFrameManager 中的页面都先用双向链表将他们的指针串起来。

定义为：

```
  using SearchType = unordered_set<ListNode *, PListNodeHasher,
PListNodePredicator>;
  SearchType searcher_;
  ListNode  *lru_front_ = nullptr; //头节点
  ListNode  *lru_tail_  = nullptr; //尾节点
```

```
void lru_touch(ListNode *node) // 意味着访问节点
void lru_push(ListNode *node) // 节点放到开头
void lru_remove(ListNode *node) //移除一个节点
void foreach (function<bool(const Key &, const Value &)> func) // 传入一个
这种类型函数，在遍历的时候调用它
void remove(const Key &key)
void put(const Key &key, const Value &value)
bool get(const Key &key, Value &value)
```

Mempool

有 MemPoolSimple，一般用来存储 Frame，定义中为：

```
list<T *> pools; // 存储所有的 Frame*
set<T *>  used; // 表示正在使用的 Frame*
list<T *> frees; // 表示 free 掉的 Frame*
int       item_num_per_pool; // 每个池多少个块
```

这是一个模板类，可以管理任何类型的对象。

还有就是 MemPoolItem，定义为：

```
pthread_mutex_t mutex;
string          name;
bool            dynamic;
int             size;
int             item_size;
int             item_num_per_pool;

list<void *> pools;
set<void *>  used;
list<void *> frees;
```

是一个通用的内存池类，管理的是原始内存块，而不是具体的对象。

示例：

```
MemPoolSimple<MyObject> myPool("MyObjectPool");
myPool.init(true, 5, 100);  // 初始化5个池，每个池100个对象
MyObject *obj = myPool.alloc();  // 分配一个对象
myPool.free(obj);  // 释放对象

MemPoolItem myPool("MyMemoryPool");
myPool.init(32, true, 5, 100);  // 初始化5个池，每个池100个32字节的内存块
void *mem = myPool.alloc();  // 分配一个32字节的内存块
myPool.free(mem);  // 释放内存块
```

常用的函数接口差不多，都是 `alloc()`，以及 `free(sth)`

BPFrameManager

```
  using FrameLruCache  = common::LruCache<FrameId, Frame *,
BPFrameIdHasher>;
  using FrameAllocator = common::MemPoolSimple<Frame>;

  mutex          lock_;
  FrameLruCache  frames_;
  FrameAllocator allocator_;
```

常用函数：

```
Frame *get(int buffer_pool_id, PageNum page_num); //获取指定页面
list<Frame *> find_list(int buffer_pool_id); //列出所有指定文件的页面
Frame *alloc(int buffer_pool_id, PageNum page_num); //分配一个新的页面
RC free(int buffer_pool_id, PageNum page_num, Frame *frame); // free 掉一
个页面
int purge_frames(int count, function<RC(Frame *frame)> purger); // 淘汰一
些页面，用来准备分配新的页面
```

DiskBufferpool

一个文件被划分成多个相同大小的页面，并在需要访问的时候，会从文件读取到内存中。DiskBufferPool 就负责管理磁盘文件，以及负责管理页面在文件与内存中的交互，比如读取、写回。

```
private:
  BufferPoolManager   &bp_manager_;      /// BufferPool 管理器
  BPFrameManager      &frame_manager_;   /// Frame 管理器
  DoubleWriteBuffer   &dblwr_manager_;   /// Double Write Buffer 管理器
  BufferPoolLogHandler log_handler_;     /// BufferPool 日志处理器

  int file_desc_ = -1;  /// 文件描述符
  /// 由于在最开始打开文件时，没有正确的buffer pool id不能加载header frame，
所以单独从文件中读取此标识
  int32_t       buffer_pool_id_ = -1;
  Frame        *hdr_frame_      = nullptr;  /// 文件头页面
  BPFileHeader *file_header_    = nullptr;  /// 文件头
  set<PageNum>  disposed_pages_;            /// 已经释放的页面

  string file_name_;  /// 文件名
```

```
  common::Mutex lock_;
  common::Mutex wr_lock_;

private:
  friend class BufferPoolIterator; //声明友元，BufferPoolIterator 可访问此类的 private 变量
```

常见函数接口：

```
RC open_file(const char *file_name); //根据文件名打开一个文件
RC close_file(); //关闭文件
RC get_this_page(PageNum page_num, Frame **frame); //根据文件ID和页号获取指定页面到缓冲区，返回页面句柄指针。
RC allocate_page(Frame **frame); // 在指定文件中分配一个新的页面，并将其放入缓冲区
RC dispose_page(PageNum page_num); // 释放某个页面，将此页面设置为未分配状态
RC purge_page(PageNum page_num); //释放指定文件关联的页的内存，刷新指定页面到磁盘(flush)，并且释放关联的Frame
RC flush_page(Frame &frame); // 刷新数据到double write buffer
RC unpin_page(Frame *frame); // unpin
RC write_page(PageNum page_num, Page &page) //刷新数据到磁盘
RC recover_page(PageNum page_num); //回放日志时处理page0中已被认定为不存在的page
RC redo_allocate_page(LSN lsn, PageNum page_num);
RC redo_deallocate_page(LSN lsn, PageNum page_num);
```

Bufferpoolmanager

```
class BufferPoolManager final
{
public:
  BufferPoolManager(int memory_size = 0);
  ~BufferPoolManager();
  // 初始化双写缓冲区
  // @param dblwr_buffer 双写缓冲区对象
  // @return 操作结果代码
  RC init(unique_ptr<DoubleWriteBuffer> dblwr_buffer);
  // 创建一个新的文件，并将其关联到一个BufferPool
  // @param file_name 文件名
  // @return 操作结果代码
  RC create_file(const char *file_name);
  // 打开一个已存在的文件，并为其分配一个DiskBufferPool实例
```

```
  // @param log_handler 日志处理器对象，用于处理恢复操作
  // @param file_name 要打开的文件名
  // @param bp 返回的DiskBufferPool指针
  // @return 操作结果代码
  RC open_file(LogHandler &log_handler, const char *file_name,
DiskBufferPool *&bp);
  // 关闭并释放与给定文件名关联的DiskBufferPool
  // @param file_name 要关闭的文件名
  // @return 操作结果代码
  RC close_file(const char *file_name);
  // 将指定帧的数据刷新到磁盘中
  // @param frame 包含数据的帧
  // @return 操作结果代码
  RC flush_page(Frame &frame);
  // 获取帧管理器对象，用于管理和调度缓存中的帧
  BPFrameManager    &get_frame_manager() { return frame_manager_; }
  // 获取双写缓冲区对象
  DoubleWriteBuffer *get_dblwr_buffer() { return dblwr_buffer_.get(); }

  /**
   * @brief 根据ID获取对应的BufferPool对象
   * @details 在做redo时，需要根据ID获取对应的BufferPool对象，然后让
bufferPool对象自己做redo
   * @param id buffer pool id
   * @param bp buffer pool 对象
   */
  RC get_buffer_pool(int32_t id, DiskBufferPool *&bp);

private:
  // 帧管理器，用于管理缓存中的所有帧
  BPFrameManager frame_manager_{"BufPool"};
  // 双写缓冲区对象，用于提高数据持久化的可靠性
  unique_ptr<DoubleWriteBuffer> dblwr_buffer_;
  // 互斥锁，用于保护共享资源免受多线程并发访问的影响
  common::Mutex                               lock_;
  // 存储文件名到DiskBufferPool映射关系的哈希表
  unordered_map<string, DiskBufferPool *>  buffer_pools_;
  // 存储缓冲池ID到DiskBufferPool映射关系的哈希表
  unordered_map<int32_t, DiskBufferPool *> id_to_buffer_pools_;
  // 记录下一个可用的缓冲池ID，原子类型确保线程安全
  atomic<int32_t>                             next_buffer_pool_id_{1};  // 系
统启动时，会打开所有的表，这样就可以知道当前系统最大的ID是多少了
};
```

还有一个 double_write_buffer。

Trx

before，函数只用关注 insert, delete 之类的

Index

分为 index_meta，index，bplustreeindex，bplustree, bplustree_log,bplustree_log_entry, latch_memo,

Index 定义如下：

```
    IndexMeta index_meta_;  ///< 索引的元数据
    FieldMeta field_meta_;  ///< 当前实现仅考虑一个字段的索引
```

latch_memo 的定义：

```
    DiskBufferPool      *buffer_pool_ = nullptr;
    deque<LatchMemoItem> items_;
    vector<PageNum>      disposed_pages_;  /// 等待释放的页面
```

Persist

用来辅助文件读写操作之类的，不过实际没用到，但是看上去还是很方便的。

multi-index

index_scan operator 需要修改