实验 3：数据定义的实现

一、实验目的

1. 掌握 Rucbase 元数据管理的实现方法。
2. 掌握 Rucbase 数据定义语句的实现方法。

二、相关知识

1. 元数据管理
2. 数据定义语言

三、实验内容

本实验包括 1 项任务。

任务 1：元数据管理实现

补全 SmManager 类，实现元数据管理功能，使 Rucbase 能够支持下列语句和命令：

CREATE TABLE 语句

DROP TABLE 语句

show tables 命令

desc 命令

SmManager::open_db

```
void open_db(const std::string& db_name);
```

1.方法声明：

方法名：open_db

返回类型：void

功能：打开数据库，找到数据库对应的文件夹，并加载数据库元数据和相关文件

参数列表：

db_name	
类型	const std::string&
含义	数据库名称，与文件夹同名

2.方法实现思路：

1. 调用 `is_dir(db_name)`检查数据库目录是否存在，如果不存在则抛出 `DatabaseNotFoundError` 异常
2. 使用 `chdir(db_name.c_str())`进入数据库目录，切换失败则抛出 `UnixError` 异常
3. 打开数据库元数据文件 `DB_META_NAME`（"db.meta"），使用 `std::ifstream` 读取文件流
4. 使用重载的`>>`操作符将元数据反序列化到 `db_`对象中：`ifs >> db_`
5. 遍历 `db_.tabs_`中的所有表元数据：
    - 使用 `rm_manager_->open_file(tab_name)`打开表文件，返回 `RmFileHandle` 智能指针
    - 使用 `fhs_.emplace(tab_name, ...)`将表文件句柄存储到映射中
    - 遍历该表的所有索引元数据 `tab_meta.indexes`
    - 使用 `ix_manager_->get_index_name(tab_name, index.cols)`获取索引文件名
    - 使用 `ix_manager_->open_index(tab_name, index.cols)`打开索引文件
    - 使用 `ihs_.emplace(index_name, ...)`将索引句柄存储到映射中
6. 保持当前工作目录在数据库目录中，不切换回父目录

3.实现难点：

需要理解目录切换策略的设计：`create_db()`在创建数据库后切换回父目录，而 `open_db()`在打开数据库后保持在数据库目录中。这样设计使得后续的

`flush_meta()`、`show_tables()`等操作能在正确的工作目录下进行。同时要正确使用 `emplace()`方法管理智能指针，避免不必要的拷贝。

SmManager::close_db

```
void close_db();
```

1.方法声明：

方法名：close_db

返回类型：void

功能：关闭数据库并把数据落盘

参数列表：无

2.方法实现思路：

1. 调用 `flush_meta()`将数据库元数据刷新到磁盘，确保最新的元数据被保存
2. 遍历索引句柄映射 `ihs_`中的所有索引文件：
    - 对每个索引句柄，调用 `ix_manager_->close_index(entry.second.get())`关闭索引文件
    - `close_index()`内部会调用 `buffer_pool_manager_->flush_all_pages()`将该索引文件的所有脏页刷新到磁盘
3. 调用 `ihs_.clear()`清空索引句柄映射
4. 遍历表文件句柄映射 `fhs_`中的所有表文件：
    - 对每个表文件句柄，调用 `rm_manager_->close_file(entry.second.get())`关闭表文件
    - `close_file()`内部会调用 `buffer_pool_manager_->flush_all_pages()`将该表文件的所有脏页刷新到磁盘

5. 调用 `fhs_.clear()`清空表文件句柄映射

3.实现难点：

需要按照正确的顺序关闭文件：先刷新元数据，再关闭索引文件，最后关闭表文件，以保证数据一致性。同时要理解智能指针的使用：使用 `entry.second.get()`获取 `unique_ptr` 的原始指针传递给关闭函数，但句柄的生命周期由 `unique_ptr` 管理，`clear()`会自动释放内存。

SmManager::drop_table

```
void drop_table(const std::string& tab_name, Context* context);
```

1.方法声明：

方法名：drop_table

返回类型：void

功能：删除指定的表，包括表文件、索引文件和元数据

参数列表：

	tab_name	context
类型	const std::string&	Context*
含义	要删除的表名	查询上下文（本实验未使用）

2.方法实现思路：

1. 调用 `db_.is_table(tab_name)`检查表是否存在，如果不存在则抛出 `TableNotFoundError` 异常
2. 调用 `db_.get_table(tab_name)`获取表的元数据引用

3. 删除该表的所有索引：
    - 遍历 tab.indexes 中的所有索引元数据
    - 使用 ix_manager_->get_index_name(tab_name, index.cols)获取索引文件名
    - 在 ihs_映射中查找该索引的句柄：auto ih_iter = ihs_.find(index_name)
    - 如果找到索引句柄（ih_iter != ihs_.end()），先调用 ix_manager_->close_index(ih_iter->second.get())关闭索引文件
    - 从 ihs_映射中移除该索引句柄：ihs_.erase(ih_iter)
    - 调用 ix_manager_->destroy_index(tab_name, index.cols)删除索引文件
4. 删除表文件：
    - 在 fhs_映射中查找该表的文件句柄：auto fh_iter = fhs_.find(tab_name)
    - 如果找到文件句柄（fh_iter != fhs_.end()），先调用 rm_manager_->close_file(fh_iter->second.get())关闭表文件
    - 从 fhs_映射中移除该表文件句柄：fhs_.erase(fh_iter)
    - 调用 rm_manager_->destroy_file(tab_name)删除表文件
5. 从元数据对象 db_.tabs_中删除该表的元数据：db_.tabs_.erase(tab_name)
6. 调用 flush_meta()将更新后的元数据持久化到磁盘

3.实现难点：

需要严格按照正确的顺序进行删除操作：先关闭并删除所有索引文件，再关闭并删除表文件，最后删除元数据并刷新到磁盘。必须先从内存映射中移除句柄再删除物理文件，避免访问已删除的文件。同时要使用迭代器进行查找和删除，提高效率并避免重复查找。

四、实验总结

本次实验完善了 SmManager 类，实现元数据管理功能，覆盖了从解析到执行的端到端流程，代码结构清晰，分析器与执行器的分工明确。当前实现满足实验要求的功能性目标；后续可在执行计划选择与性能优化、更多容错与更丰富的 SQL 支持上继续扩展。