说明文档

实验框架

0. TL; DR

直接阅读 `main.cc` 中单元测试函数

几乎所有存储都使用 `std::shared_ptr` 封装

1. Field

实现了 Int / Float / Double 三种数据类型，都继承了 Field 的接口

```
virtual uint8_t *store(uint8_t *dst) const = 0;
virtual const uint8_t *load(const uint8_t *src) = 0;
virtual FieldType type() const = 0;
virtual size_t size() const = 0;
virtual std::string info() const = 0;
```

其中 `load` 与 `store` 将数据写入/读出内存，不带类型标识

`type` 返回数据类型也即 `FieldType`， `size` 返回数据所占空间大小

`info` 和 `operator` 为工具，可以提升开发速度

2. Record

存储了多个 Field，构成数据表中的一行，不带元数据

也即本质为 `std::vector<Field>`，增加了工具函数封装为类

提供 `load` 与 `store` 接口，依次调用 Field 对应的方法进行存储

3. Table

数据表的抽象，存储表名与表头，并基于此实现数据库的增删改查

```
Entry insert(std::shared_ptr<Record> record);
bool update(Entry dst, std::shared_ptr<Record> record);
bool remove(Entry dst);
std::vector<std::shared_ptr<Record>> select();
```

其中 Entry 为存储位置，也即 Page ID 与 Slot ID 所组成的 `std::pair`

对于插入操作，可以自行决定写入文件中的位置，并返回对应 Entry

框架暂时不考虑查找时的过滤，也即均为全量搜索

4. Instance

对接 Parser 和 Table 的接口

```
Instance(std::string path = "data");
bool create(std::string tbName, Header header);
bool drop(std::string tbName);
```

构造函数会遍历对应目录，读取对应数据表信息并存入 `_tables` 中

`create` 和 `drop` 分别创建/删除新数据表

5. Filesystem

提供了一份 Page 和 Slot 的参考实现，也可以不参考

所有 Entry 格式均为 `std::pair<size_t, size_t>`，不会关注具体空间分配

> 如果你想应用这份代码，请完成 BufPageManager 中 `getPage` 和 `writeBack` 的代码填空，函数描述已经在代码中给出详细解释
>
> 这一部分框架的测试代码在 `testfilesystem.cpp` 中，跑通测试代码且输出与 `refer.out` 一致，即视为该部分功能基本正确

TASK 1 STORAGE

你需要使用上课所学的知识完成数据库存储部分的设计

对于 Field，你需要增加 StringField 并完成对应 IO 接口

对于 Table，需要完成以下接口

```
Entry insert(std::shared_ptr<Record> record);
bool update(Entry dst, std::shared_ptr<Record> record);
bool remove(Entry dst);
std::vector<std::shared_ptr<Record>> select();
```

其中 `select` 为全量查找

对于 Instance，需要完成以下接口

```
Instance(std::string path = "data");
bool create(std::string tbName, Header header);
bool drop(std::string tbName);
```

测试流程详见 `main.cc`，后续会持续更新单元/集成测试