元数据模块文档

2016013258 王泽宇 2016013270 李帅 2016013276 纳鑫

引言

经过存储模块部分的设计后，我们对我们数据库系统的总体设计做了一些细化。此文档中包含我们目前已有的类的设计，接口的说明，以及元数据的操作流程。

环境说明

运行环境

项目使用Java语言开发，JDK版本:1.8.0_201

第三方库

本项目使用了*JUnit4*库用于测试，包括了一下jar包

- hamcrest-core，版本1.3
- junit，版本4.13

元数据管理模块设计

支持的功能

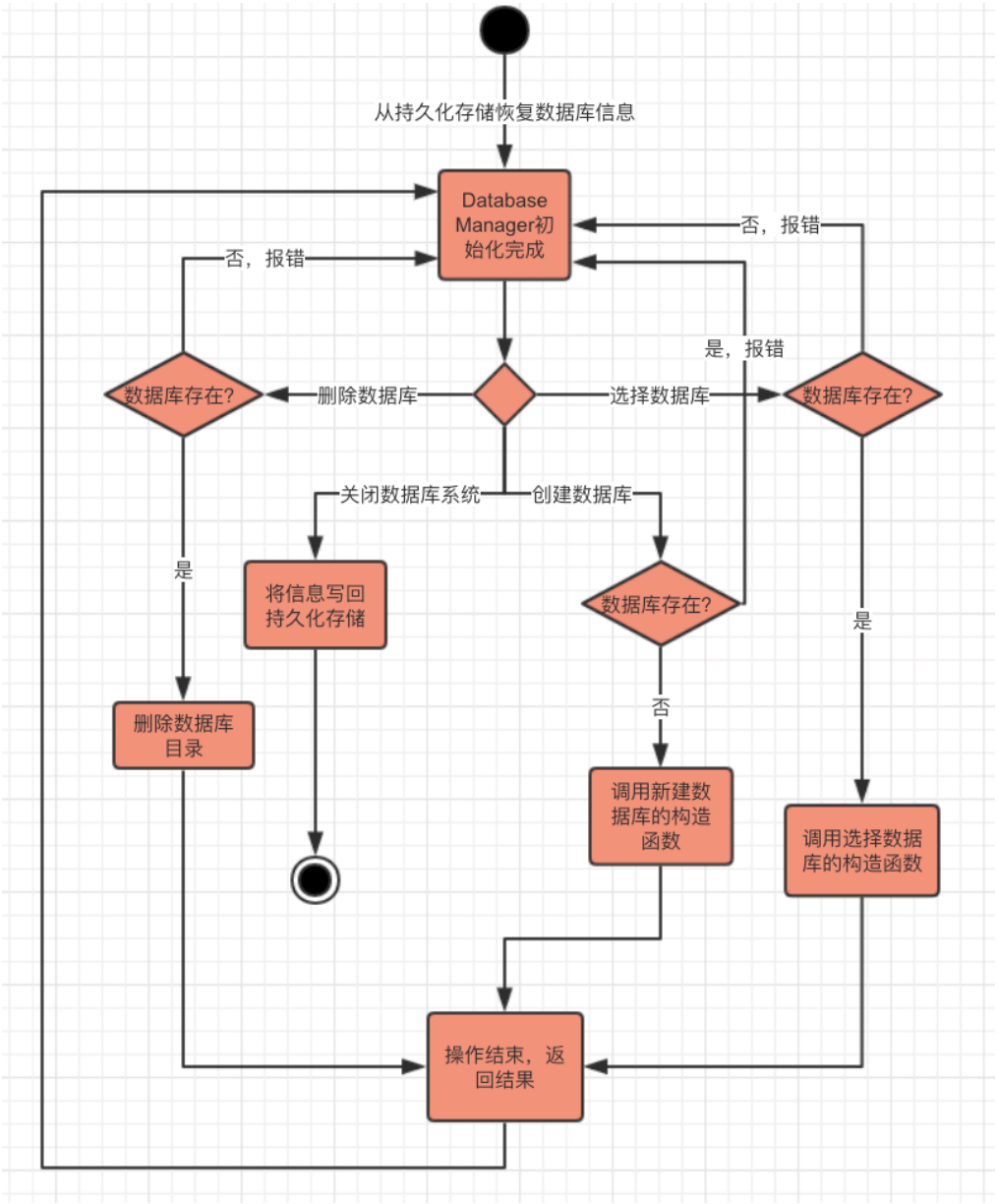
NaiveDB的元数据管理模块支持多个数据库、多张表存在，具体如下：

- 创建、删除一个数据库
- 获取系统的所有数据库
- 切换数据库
- 创建、删除一张表
- 将数据库与表的元信息进行持久化存储，并且能够在启动时恢复信息

处理流程

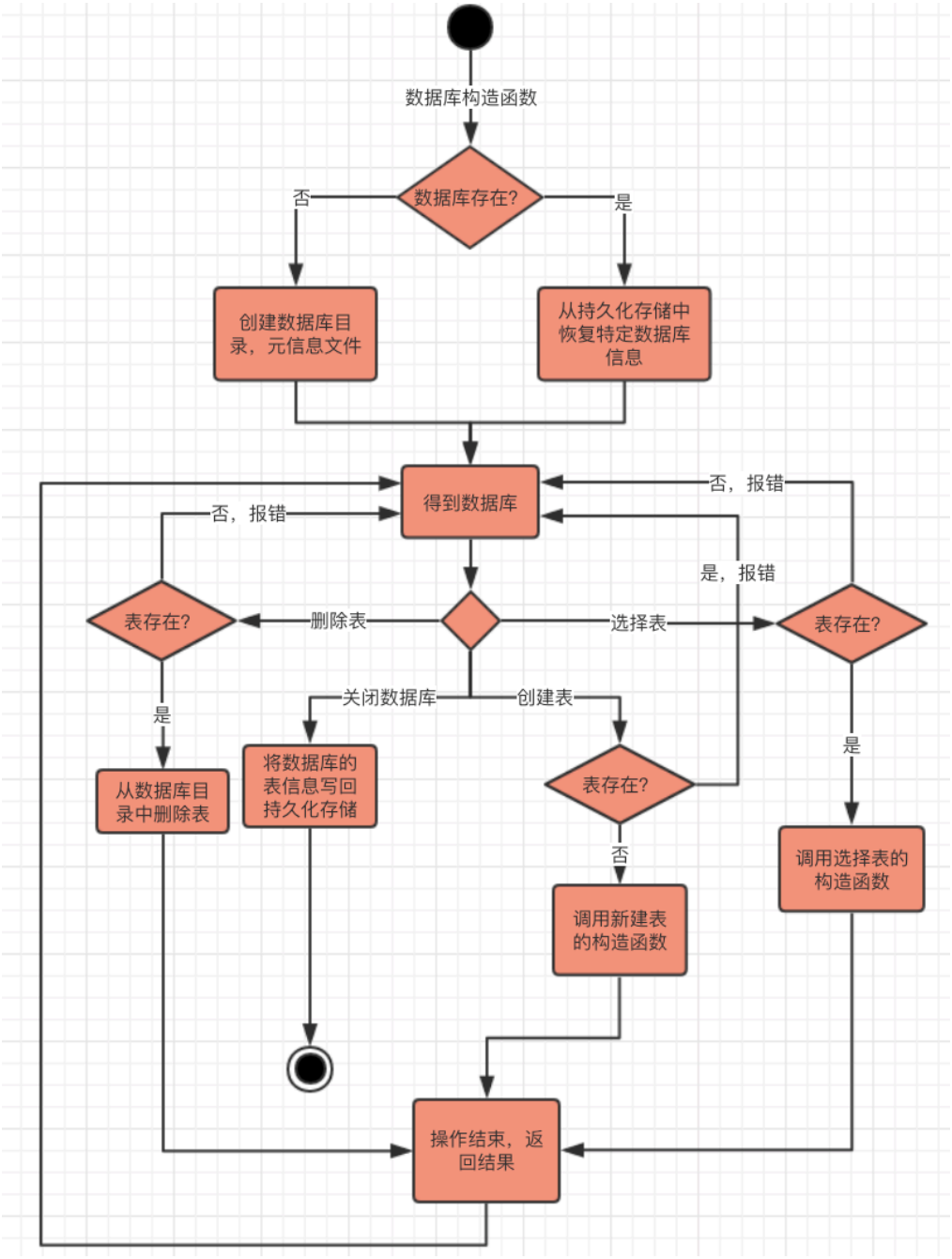
元数据管理一共分为两个部分，即对多数据库的管理和对多张表的管理

数据库元信息处理流程



对于数据库元信息的管理，都在`DatabaseManger`类中进行，在后文中有对此类的详细说明。数据库系统启动后首先调用此类的`initial`函数进行初始化，加载持久化的元信息，包括所有数据库名。之后接收具体操作，这里有四种，创建、删除、切换和关闭。关闭即关闭整个数据库系统，将元信息写回持久化存储；创建和切换都调用了`Database`类的构造函数，得到一个数据库；删除则直接将数据库对象的目录删除。

数据表元信息处理流程

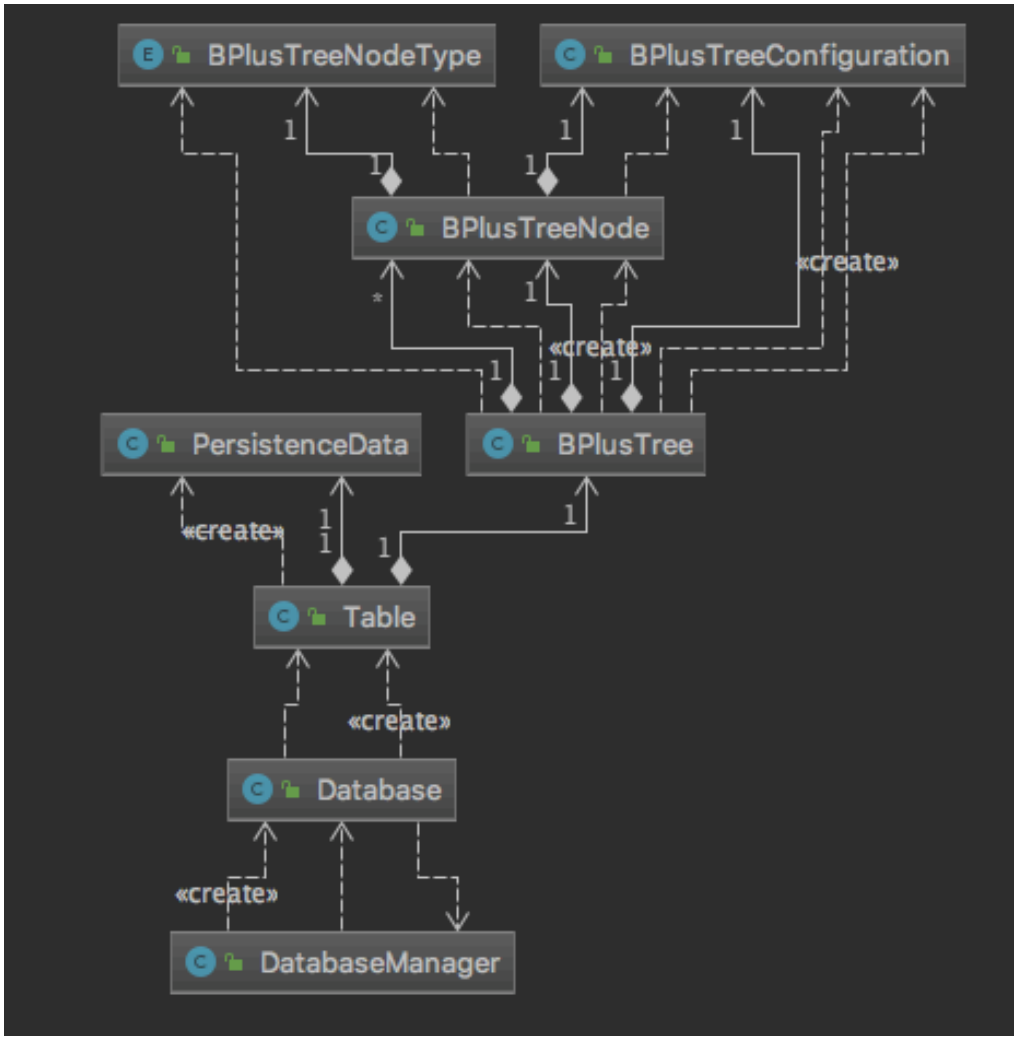


对于表的元信息管理，都在`Database`类的实例中进行，因为每一张表都存在于一个具体的数据库中。在切换或创建数据库后，调用了其构造函数，根据情况判断是创建还是切换，之后得到了数据库的对象。此对象的实例提供了创建、删除、选择和关闭四个操作。关闭即关闭此数据库，将元信息重新写回持久化存储；创建和选择表调用`Table`类的构造函数，得到一张表；删除则清空此数据库下某张特定的表。

总体结构

在元数据模块的设计中，我们对NaiveDB原来的B+树模块和util模块进行了重构，B+树模块拆分为B+树类和Persistence类，即把持久化存储部分移出，并使用Table类对他们进行管理，Database包中的类则用于管理和数据库相关的内容；并把原有utils部分进行了重构。对于新增的其他包：Type见接口设计部分，Test包见测试设计。

Databse，Table包总体的结构



Table类中

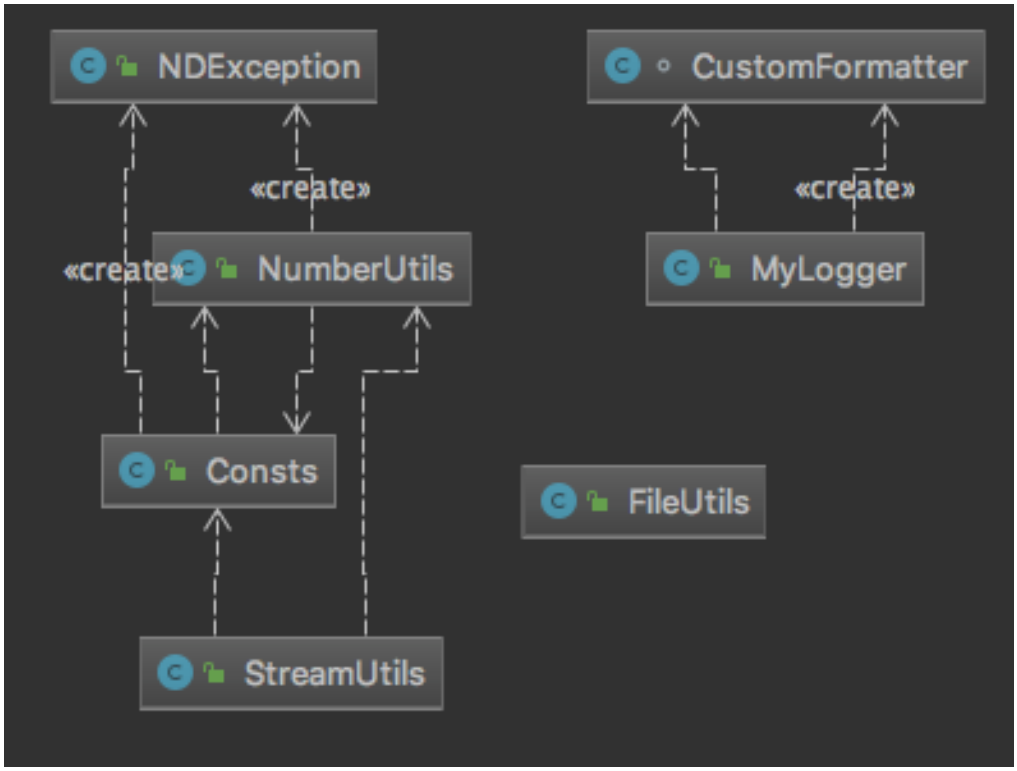
- 将Persistence实例作为对数据的管理
- 将B+树实例作为索引
- 元信息在Table类中进行维护

Database包中

- 使用DatabaseManager类的静态方法对所有数据库进行管理
- 使用Databse类表示具体数据库，并对表进行管理

utils类总体结构

我们将原有的utils进行了重构，结构如下



- 系统的所有异常目前都统一使用`NDException`表示
- 增加了系统自定义日志输出`MyLogger`
- 按照功能拆分了所有工具类

接口设计

基于已经完成的B+树部分，我们对其进行了一定拆分和优化，目前所有类的接口如下：

BPlusTree包

依然对外提供两个类：`BPlusTree`和`BPlusTreeConfiguration`，接口均与存储模块类似，这里不再赘述

Database包

外对提供两个类，`Database`和`DatabaseManager`。

DatabaseManager类

此类提供所有操作都是静态函数，内部变量也都是静态，使用类名调用操作即可。接口如下

- `initial()`，初始化数据库系统，每次启动时必须调用
- `close()`，关闭数据库系统，每次退出时必须调用
- `Database create(String)`，创建一个数据库，输入数据库名，返回创建好的数据库
- `Database get(String)`，获取一个已经创建过的数据库，返回数据库
- `void drop(String)`，删除一个数据库
- `ArrayList<String> getDatabases()`，获取目前所有的数据库

Database类

此类包含了所有数据库相关操作，获取实例通过`DatabaseManager`的`create`和`get`方法即可，接口如下

- `void close()`，关闭一个数据库，在使用完数据库后调用，写回数据库保存的元信息
- `Table createTable(String table_name, ArrayList<Pair<String, Type>> cols)`，创建一个表，输入表名和列信息，返回一个表对象
- `Table getTable(String table_name)`，获取一个已经创建了的表，返回一个表对象
- `void dropTable(String table_name)`，从数据库中删除某张表
- `ArrayList<String> getTables()`，获取数据库中所有的表

Table包

此包提供一个Table类，通过`Database`类的`createTable`和`getTable`即可，接口如下

- `void close()`，关闭table，将数据写回磁盘，使用完table后调用
- `void setPrimary(String)`，设置某列为主键，只能设置一次，参数为列名
- `void setNotNull(ArrayList<Boolean>)`，设置非空表，参数为一张布尔的表
- `ArrayList<String> getColNames()`，获取所有列名
- `ArrayList<Type> getColTypes()`，获取所有列类型
- `String getFileName()`，获取table的文件名前缀

说明：由于SQL语句部分未实现，目前Table类提供的接口并不完善，这里也是下一个模块工作的中心

Persistence包

此包提供一个`PersistenceData`类用于提供数据的持久化存储，接口如下

- 构造函数`PersistenceData(String filename, LinkedList<String> types)`，接受文件名和类型列表。
- `long add(LinkedList value)` 增加一条数据，并返回其行号
- `LinkedList get(long rowNum)`获取行号为`rowNum`的数据
- `remove(long rowNum)`返回行号为`rowNum`的数据
- `update(long rowNum, LinkedList value)`跟新一条数据，注意：旧数据的主键和新数据的主键必须一致。
- `close()`关闭数据库前务必调用此函数以将数据改变写回文件。

Type包

此包提供数据库类型对象，接口如下

- 构造函数`Type(int)`，`Type(int, int)`，`Type(String)`，分别用于创建普通类型，带一个参数的类型（这里只有String）和从*类型字符串*加载一个类型
- `String typeName()`，获取类型的*类型字符串*
- `int typeSize()`，获取类型的大小
- `boolean check(Object)`，类型检查，检查输入是否为此类型
- `int getType()`，获取类型的枚举量

utils包

相较于存储部分，这部分多了两个类`FileUtils`和`StreamUtils`两个工具，其中`FileUtils`目前只提供了递归删除一个文件夹的函数。`StreamUtils`提供了对`BufferedOutputStream`和`BufferedInputStream`实例的操作，例如从流中读取一个`Int`、`Long`、`Float`、`Double`、`String`类型，以及将它们写入流中。

元数据持久化结构

数据库部分

首先在`DatabaseManager`初始化时，会在当前运行目录下创建`data`文件夹和`data/db.meta`作为所有数据库的元信息。`db.meta`中保存了目前所有的数据库。 当创建了一个新的数据库，会在`data`文件夹下创建对应数据库名的文件夹，并在文件夹中创建`dbmeta`文件，其中保存了此数据库下所有的数据表。

表部分

使用某个数据库创建一张表，会在数据库对应目录下创建`表名+.meta`文件和`表名+.data`文件（还有其他的一些，用于B+树持久化）。其中`meta`后缀名的文件保存列类型，列名，列是否为空，主键等信息。`.data`后缀名保存数据。

测试部分

编写此模块时我们给系统添加了测试，目前测试框架和使用方法已经搭建完成，不过测试的编写还不完全。我们使用了`JUnit4`框架编写测试，运行`Test`包下的`TestRunner`类即可运行所有测试，测试类的编写按照`JUnit4`的规则即可，需要运行测试类需要在`TestRunner`的`main`函数中加入相应的`runClass`语句。 我们后期会加入更多测试并形成测试文档。

总结以及计划

元数据管理模块完成后，NaiveDB的整体框架基本形成，接下来我们编写对输入的SQL语句编译运行的模块，以及具体执行过程的函数，重点即为查询部分的编写。