大规模信息系统构建技术导论

3180105079 李昊

2022 年 5 月 20 日

目录

1. 引言

本次开发的是一个分布式 MiniSQL 系统，我主要负责 MiniSQL 系统的 Record Manager， Index Manager， Buffer Manager 模块设计以及客户端的设计。

1.1 系统目标

本次系统开发中，Zookeeper 用于实现服务器与客户端，服务器与服务器通信，总体设计中 MiniSQL 系统运行于每个服务器内部，由 Zookeeper 将指令分发到各服务器，MiniSQL 系统的主要工作是接受指令进行工作并返回结果信息。

(1) Record Manager 模块：

Record Manager 负责管理记录表中数据的数据文件。主要功能为实现数据文件的创建与删除（由表的定义与删除引起）、记录的插入、删除与查找操作，并对外提供相应的接口。其中记录的查找操作要求能够支持不带条件的查找和带一个条件的查找（包括等值查找、不等值查找和区间查找）。

数据文件由一个或多个数据块组成，块大小应与缓冲区块大小相同。一个块中包含一条至多条记录，为简单起见，只要求支持定长记录的存储，且不要求支持记录的跨块存储。

(2) Index Manager：

Index Manager 负责 B+树索引的实现，实现 B+树的创建和删除（由索引的定义与删除引起）、等值查找、插入键值、删除键值等操作，并对外提供相应的接口。

B+树中节点大小应与缓冲区的块大小相同，B+树的叉数由节点大小与索引键大小计算得到。

(3) Buffer Manager：

Buffer Manager 负责缓冲区的管理，主要功能有：

1. 根据需要，读取指定的数据到系统缓冲区或将缓冲区中的数据写出到文 件
2. 实现缓冲区的替换算法，当缓冲区满时选择合适的页进行替换
3. 记录缓冲区中各页的状态，如是否被修改过等
4. 提供缓冲区页的 pin 功能，及锁定缓冲区的页，不允许替换出去

为提高磁盘 I/O 操作的效率，缓冲区与文件系统交互的单位是块，块的大小应为文件系统与磁盘交互单位的整数倍，一般可定为 4KB 或 8KB。

（4）client:

1. 接收用户命令行或者文件输入的 sql 语句。
2. 根据副本和负载均衡的相关配置，得出该 sql 语句对应的 MiniSQL 服务器。
3. 将该指令同步到 zookeeper 的对应服务器结点上。

1.2 系统环境

操作系统 windows10

编程语言 python 3.9

2. 详细设计

2.1 Record Manager 模块

为了节约空间，数据库文件存储采用的是字节存储，因此通过 encode 和 decode 函数，采用 ASCLL 编码的方式对数据进行存储。

```python
def convert_str_to_bytes(attributes):
    attr_list = list(attributes)
    for index, item in enumerate(attr_list):
        if isinstance(item, str):
            attr_list[index] = item.encode('ASCII')
    return tuple(attr_list)


def convert_bytes_to_str(attributes):
    attr_list = list(attributes)
    for index, item in enumerate(attr_list):
        if isinstance(item, bytes):
            attr_list[index] = item.decode('ASCII').rstrip('\00')
    return tuple(attr_list)
```

图表 1 字符串和字节转换函数

本模块主要实现数据的增删改查功能，具体实现首先需要得到确定表的名称，从而确定元数据，然后根据元数据确定每个数据的 size 大小，根据 size 逐个遍历，从而定位数据位置。

因此定义了两个类 RecordManager 和 Record 两个类，其中 RecordManager

类作为对外接口，读入表的名称和需要修改的值，调动 Record 类中的具体方法进行插入。

RecordManager 的函数以 insert 为例，数据存储在 minisql 文件下，以 table_name.table 作为文件名分别存储每个表，然后调用 Record 的构造函数传入表的路径，最后调用 Record 类里的 insert 函数插入数据，最后返回插入后的字节位置。

```
@classmethod
def insert(cls, table_name, fmt, attributes):
    """
        insert the given record into a suitable space,
        and return the offset of the inserted record
    """
    file_path = cls.file_dir + table_name + '.table'
    record = Record(file_path, fmt)
    position = record.insert(attributes)
    return position
```

图表 2 insert 函数

Record 类的函数以 insert 为例，首先，将传入的数值修改为 bytes 类型，然 后调用_parse_header()函数，向 BufferManager 申请第一个 block，这个 block 用于存储元数据，然后调用_calc 函数计算数据所在的 block 位置，然后向 BufferManager 申请锁，找到第一个空闲的位置插入数据，最后返回位置。

```python
def insert(self, attributes):
    """Insert the given record"""
    record_info = convert_str_to_bytes(attributes) + (b'1', -1)  # valid bit, next free space
    self.first_free_rec, self.rec_tail = self._parse_header()
    if self.first_free_rec >= 0:  # There are space in free list
        first_free_blk, local_offset = self._calc(self.first_free_rec)
        block = self.buffer_manager.get_file_block(self.filename, first_free_blk)
        with pin(block):
            data = block.read()
            records = self._parse_block_data(data, first_free_blk)
            next_free_rec = records[self.first_free_rec][-1]
            records[local_offset] = record_info
            new_data = self._generate_new_data(records, first_free_blk)
            block.write(new_data)
        position = self.first_free_rec
        self.first_free_rec = next_free_rec
    else:  # No space in free list, append the new record to the end of file
        self.rec_tail += 1
        block_offset, local_offset = self._calc(self.rec_tail)
        block = self.buffer_manager.get_file_block(self.filename, block_offset)
        with pin(block):
            data = block.read()
            records = self._parse_block_data(data, block_offset)
            records.append(record_info)
            new_data = self._generate_new_data(records, block_offset)
            block.write(new_data)
        position = self.rec_tail
    self._update_header()
    return position
```

图表 3 insert 函数

2.2 Index Manager 模块

索引采用了 B+树进行制作，设计了 Node 类和 IndexManager 类进行控制，Node 类定义了节点和具体操作，IndexManager 类调用 Node 类的方法增删改查数据，同时更新 B+树。

首先对 Node 节点进行了定义。Key_struct 对键值进行了定义，需要用到辅助函数将所有键值转换为元组类型，然后定义了元数据，由三个 int 类型数据组成：是否为叶节点，next_deleted 状态，子节点位置。

```python
class Node:
    key_struct = Struct(fmt)  # the struct to pack/unpack keys
    meta_struct = Struct('<3i')  # 3 ints: self.next_deleted, self.is_leaf, len(self.keys)
    n = (BufferManager.block_size - 16) // (4 + key_struct.size)
```

图表 4 Node 类数据结构

```python
def __init__(self, is_leaf, keys, children, next_deleted=0):
    self.is_leaf = is_leaf
    self.keys = _convert_to_tuple_list(keys)
    self.children = list(children)
    self.next_deleted = next_deleted
```

图表 5 Node 类初始化

然后调用插入，分裂，左移，右移函数对节点内的数据进行更新。

```
def insert(self, key, value):···

def split(self, new_block_offset):···

def fuse_with(self, other, parent, divide_point):···

def transfer_from_left(self, other, parent, divide_point):···

def transfer_from_right(self, other, parent, divide_point):···
```

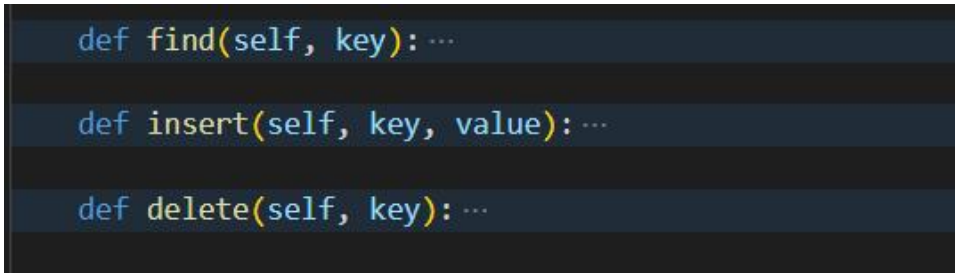
图表 6 Node 类主要方法

IndexManager 类定义了 Node，index_file_path,_manager,meta_struct 四个参数，首先向 BufferManager 申请一个 block，然后按照索引位置去找到元数据，还原成索引，如果该位置不存在索引那么就自己创建一个索引。

```
class IndexManager:
    def __init__(self, index_file_path, fmt):
        """specify the path of the index file and the format of the keys, return a index manager
        if the index file exists, read data from the file
        otherwise create it and initialize its header info
        multiple index manager on the same file MUSTN'T simultaneously exist"""
        self.Node = node_factory(fmt)
        self.index_file_path = index_file_path
        self._manager = BufferManager()
        self.meta_struct = Struct('<4i')  # total blocks, offset of the first deleted block, offset of the root node
        try:
            meta_block = self._manager.get_file_block(self.index_file_path, 0)
            with pin(meta_block):
                self.total_blocks, self.first_deleted_block, self.root, self.first_leaf = self.meta_struct.unpack(
                    meta_block.read()[:self.meta_struct.size])
        except FileNotFoundError:  # create and initialize an index file if not exits
            self.total_blocks, self.first_deleted_block, self.root, self.first_leaf = 1, 0, 0, 0
            with open(index_file_path, 'wb') as f:
                f.write(self.meta_struct.pack(self.total_blocks,
                                              self.first_deleted_block,
                                              self.root,
                                              self.first_leaf).ljust(BufferManager.block_size, b'\0'))
```

图表 7 IndexManager 类初始化

IndexManager 定义了 find，insert，delete 函数，根据键值定位到所在的 Node，然后返回 LeafIterator(用于遍历叶节点)，调用 Node 类里面的相关函数定位并进行处理。



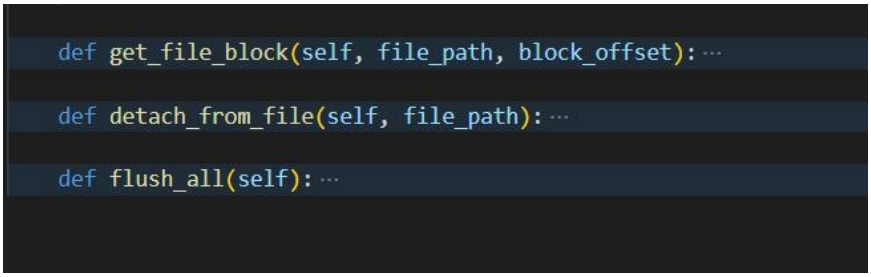
图表 8 IndexManager 类的主要方法

2.3 Buffer Manager 模块

本模块主要处理的是硬盘读写操作，设计了 Block 和 BlockManager 两个类进行控制，BlockManager 作为接口，规定了 block 的大小是 4096B，共有 1024 个 block。

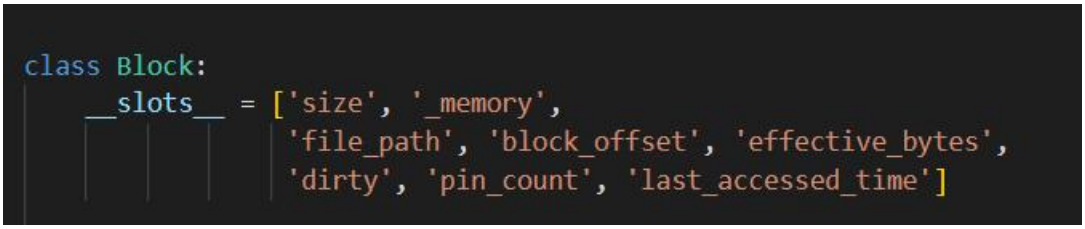
图表 9 block 块定义

Get_file_block 函数用于确定需要处理的 block 位于 file_path 的文件下的具体位置，detach 用于删除文件相关的缓存块，flush 用于把 block 里面的数据输出到硬盘中。



图表 10 BlockManager 主要方法

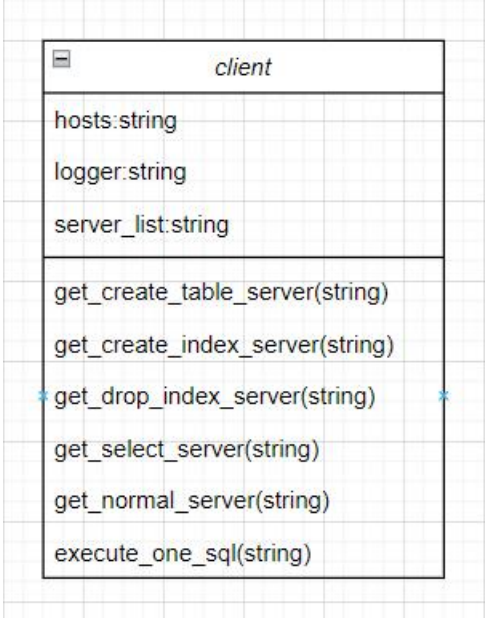
Block 下定义了一个元组__slots__存储 block 的大小，路径，偏移量等数据



图表 11 Block 数据结构

首先检查 data 的 size 需要小于 block 的 size，防止超出边界，然后在字节数组_memory 中对写入的数据进行修改，然后把数据记为脏数据，更新最后访问的时间，然后再 flush 函数中输出后，dirty 标记记为 false

2.4 client 模块



以下是方法的详细说明：

① get_create_table_server 方法。当用户输入create table 命令时，客户端会读取 zookeeper 中各个服务器 info 结点下的统计信息，选取存储数据较少的两个服务器，在对应 instruction 结点下生成指令结点。Minisql 程序处理完后，会将结果写入指令结点下的 result 结点，并且会在服务器下的 tables 结点和根目录下的 tables 结点创建该 table 结点，还会在 info 下更新相关统计信息。

② get_create_index_server 方法。create index 指令会在根目录下的 tables 结点，查找储存对应表的服务器，然后将指令同时写入这些服务器的对应结点中。处理完成后同样会写入result 结点，并在 info 结点下更新相关统计信息。

③ get_drop_index_server 方法。由于此指令只提供了 index 的 name，需要从根目录下的 indexes 查找对应的服务器。

④ get_select_server 方法。select 指令会从储存对应表的服务器中随机选取一台写入指令，实现简单的负载平衡。

⑤ get_normal_server 方法。找到表所对应的服务器。

⑥ execute_one_sql 方法，读取并处理指令。

3. 开发体会

MiniSQL 是在原有的基础上进行的开发，原本的 MiniSQL 构建上没有考虑到大规模处理的问题，数据是完全保存在内存中的，每次读写都是把整个文件读入，由于当时在单机处理数据规模较小，这个问题没有及时处理。在本次大规模系统开发中采用了课本上的 block 分块的方法，读写单位是 block，解决了大规模数据下内存不够用的问题。

在单机处理中不存在读写冲突的问题，因为所有数据只有单机访问，只需要简单的做到串行运行就可以了，但是在大规模的环境下，需要同时处理多台机的访问，因此对 block 新增了一个标志域作为锁，正在被访问的 block 域不能被其他的主机修改。

此外，我还对文件处理格式进行了修改，原本的数据没有处理格式的问题，输入输出都是通过 Unicode 编码实现的，在本次实验中用 python 的 encode 和 decode 函数进行了转换，将数据通过 ASCLL 编码进行存储，这样在大规模存储中可以有效地压缩空间。

开发中我感受到规模增大对于时间和空间提出了新的要求，原先低效率，占用大的方法在小规模下体现不出和高效率方法相比较的劣势，而在大规模的环境下就会暴露出来。因此在未来的开发中，为了避免编写的程序难以适应未来发展的需要，应该在设计的一开始就尽量采取高空间利用率、解决访问冲突的方案。