

源码分析报告二——MVStore流程设计分析

1. 前言

笔者认为，对于一个数据库，最基本的操作就是对数据的增删改查。因此本报告将主要讲述MVStore的增删改查机制和数据结构。同时，因为H2以内存存储为特色，所以笔者还会讲述MVStore的内存存储部分过程。

2. 内存数据库表层操作

MVStore中有一个MVStore类，实际上我们就可以把它当作一个内存数据库。让我们来看下面的这个例子：

```
public static void main(String[] args) throws Exception {
    // open the store (in-memory if fileName is null)
    MVStore s = MVStore.open(null);
    MVMap<Integer, String> map_1 = s.openMap("firstMap");
    MVMap<String, Integer> map_2 = s.openMap("secondMap");
    map_1.put(1, "spring");
    map_1.put(3, "fall");
    map_1.put(3, "autumn");
    map_2.put("English", 59);
    map_2.remove("English");
    System.out.println(map_1.get(1));
    System.out.println(map_1.get(4));
    System.out.println(map_2.get("Math"));
    System.out.println(map_2.get("English"));
    s.commit();
    s.close();
}
```

我们实例化了一个名叫s的属于MVStore类的对象。我们来看open方法的定义：

```
public static MVStore open(String fileName) {
    HashMap<String, Object> config = new HashMap<>();
    config.put("fileName", fileName);
    return new MVStore(config);
}
```

open方法创建了一个名叫config的哈希表。在表中放入了一个键值对。这个哈希表将传入到MVStore的构造方法中去。那么我们来看这个构造方法：

```
MVStore(Map<String, Object> config) {
    compressionLevel = DataUtils.getConfigParam(config, "compress", 0);
    String fileName = (String) config.get("fileName");
    FileStore<?> fileStore = (FileStore<?>) config.get("fileStore");
    boolean fileStoreShallBeOpen = false;
}
```

```

        if (fileStore == null) {
            if (fileName != null) {
                ...
            }
            fileStoreShallBeClosed = true;
        } else {
            ...
        }
        this.fileStore = fileStore;
        keysPerPage = DataUtils.getConfigParam(config, "keysPerPage", 48);
        backgroundExceptionHandler =
(UncaughtExceptionHandler)config.get("backgroundExceptionHandler");
        if (fileStore != null) {
            ...
        } else {
            autoCommitMemory = 0;
            meta = openMetaMap();
        }
        onVersionChange(currentVersion);
    }

```

这段代码比较长，笔者只挑一部分来说明。实际上由于我们采用了内存模式（`fileName`为`null`），因此没有运行的代码笔者在此省略掉。在此之前，该构造方法还初始化了一部分成员变量，但就目前而言不是很重要。

我们注意到有一个名叫`fileStore`的对象。在此处它的值为`null`，后期我们将会注意它。在这里还需要注意的是一个名叫`meta`的成员变量（在代码块外就已被初始化为`null`）。它所属的类为`MVMap`。`MVMap`和`HashMap`类似，是`MVStore`存储的一个基础结构。`meta`可以说是`MVMap`的`MVMap`。之后我们将会展示`meta`的功能。

`MVStore`类对象成功建立之后，我们打算创建两个对象`map_1`和`map_2`。它们都调用了`MVStore`中的`OpenMap`方法。

```

public <K, V> MVMap<K, V> openMap(String name) {
    return openMap(name, new MVMap.Builder<>());
}

```

从这段代码我们可以看到，我们实际上是执行的另外一个同名方法。不过该方法的具体过程在此不作具体讲述，我们只需要知道它的功能是：返回和名字对应的表，如果没有则当场创建一个以该名字命名的表。在这里，数据库里并没有这两个表，于是便创建了两个空表。

接着就是增删改查操作。从例子中我们可以看到，增和改调用的是`MVMap`里的`put`方法，`MVMap`类中的`put`方法定义如下：

```

public V put(K key, V value) {
    DataUtils.checkArgument(value != null, "The value may not be null");
}

```

```
        return operate(key, value, DecisionMaker.PUT);
    }
}
```

这段代码首先检查了`value`：`value`不能为空，然后真正起作用的是`operate`方法。注意到`operate`方法里面有一个参数`DecisionMaker.PUT`，我们可以猜测：`operate`方法应该可以完成多种对`MVMap`的操作，并且由`DecisionMaker`决定执行什么操作。

```
/**
 * Add, replace or remove a key-value pair.
 * ...
 */
public V operate(K key, V value, DecisionMaker<? super V> decisionMaker) {
    IntValueHolder unsavedMemoryHolder = new IntValueHolder();
    int attempt = 0;
    while(true){...}
}
```

在此给出`operate`方法的部分代码。由注释我们可以看到，`operate`可以增删改`MVMap`对象中的键值对。不用看`remove`方法的代码，我们也可以知道其一样调用了`operate`方法。若不放心，大可验证一下：

```
public V remove(Object key) {
    return operate((K)key, null, DecisionMaker.REMOVE);
}
```

而对于`get`方法（根据`key`查找`value`），同样也是调用了`operate`方法，只是参数不同罢了。最后`commit`和`close`方法将保存更改持久化数据、关闭数据库，而此处我们使用的是内存模式，程序结束后所有的数据都会丢失，这两个方法的有无影响不大。并且这两个方法调用的方法相当多，我们仅简单介绍。

首先是`commit`方法，此部分代码如下所示：

```
public long commit() {
    return commit(x -> true);
}

private long commit(Predicate<MVStore> check) {
    if(canStartStoreOperation()) {
        storeLock.lock();
        try {
            if (check.test(this)) {
                return store(true);
            }
        } finally {
            unlockAndCheckPanicCondition();
        }
    }
    return INITIAL_VERSION;
}
```

我们在这里看到了与锁相关操作。MVStore作为数据库引擎，自然要处理多用户、多线程的情况，但这不是我们要讨论的重点。除开这些，我们很容易注意到，这部分的核心是store方法，但我们并不打算仔细研究该方法。

close方法也是如此：我们可以看到，实际上起作用的是closeStore方法。这两个方法都相当复杂，我们不多作展开。

```
public void close() {  
    closeStore(true, 0);  
}
```

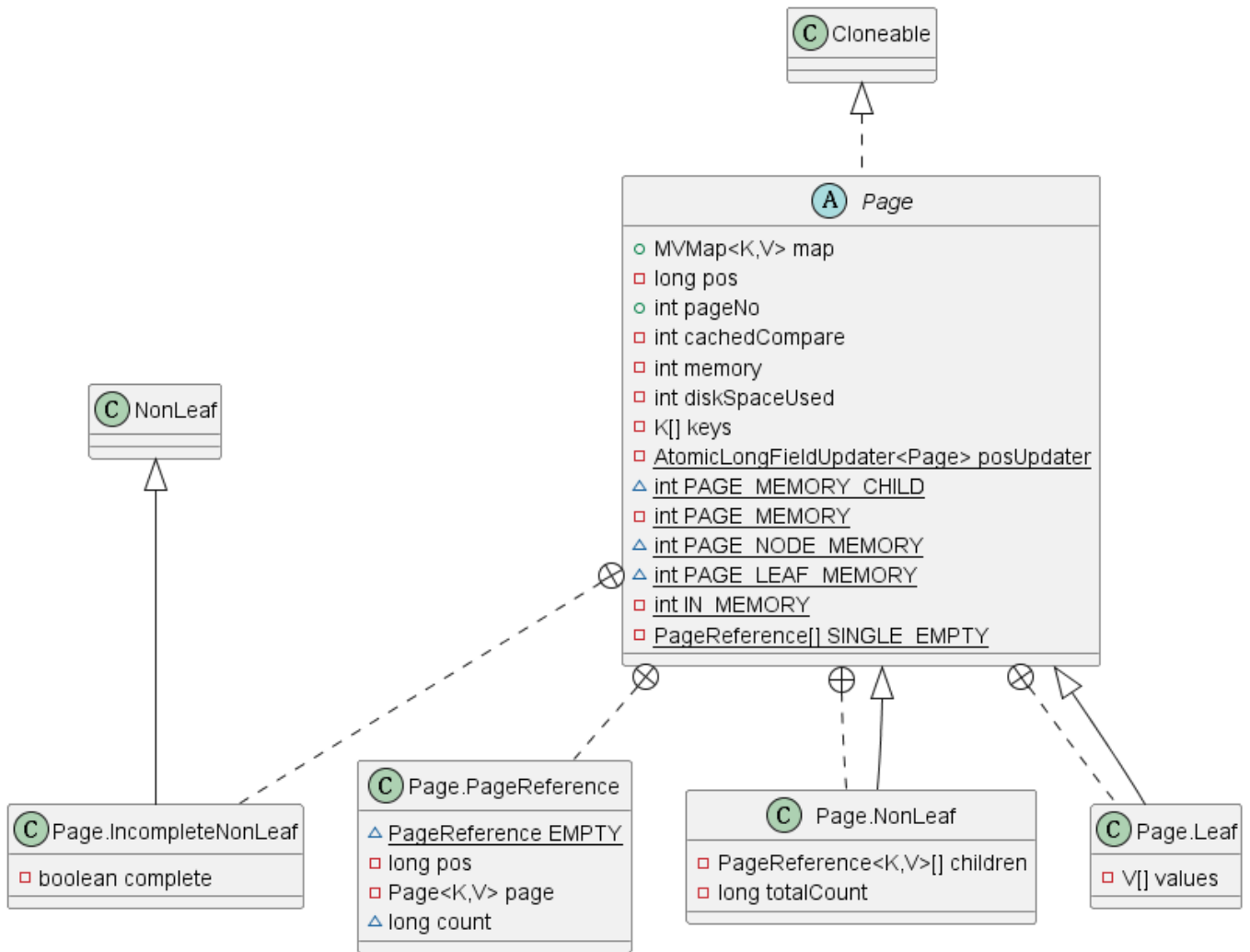
2. MVStore底层结构

2.1 MVStore的数据结构

笔者认为，数据结构的设计在整个项目中算是比较底层的内容，对于具体的数据存放地址、指针寻址等内容是比较枯燥乏味的，在此和地址有关的成员变量、方法将会略过——更重要的是如何用面向对象的方式讲述本部分内容。并且此部分更侧重讲述这些类地成员变量，较少提及相关的方法。

2.1.1 Page和MVMap

MVMap默认采用B+树结构进行存储，关于B+树的内容，读者可以查阅相关资料。我们都知道，树这种数据结构需要结点。在MVStore中，B+树的结点采用Page类进行保存。Page类是抽象类，其下有三个子类：Nonleaf、Leaf、incompleteLeaf。

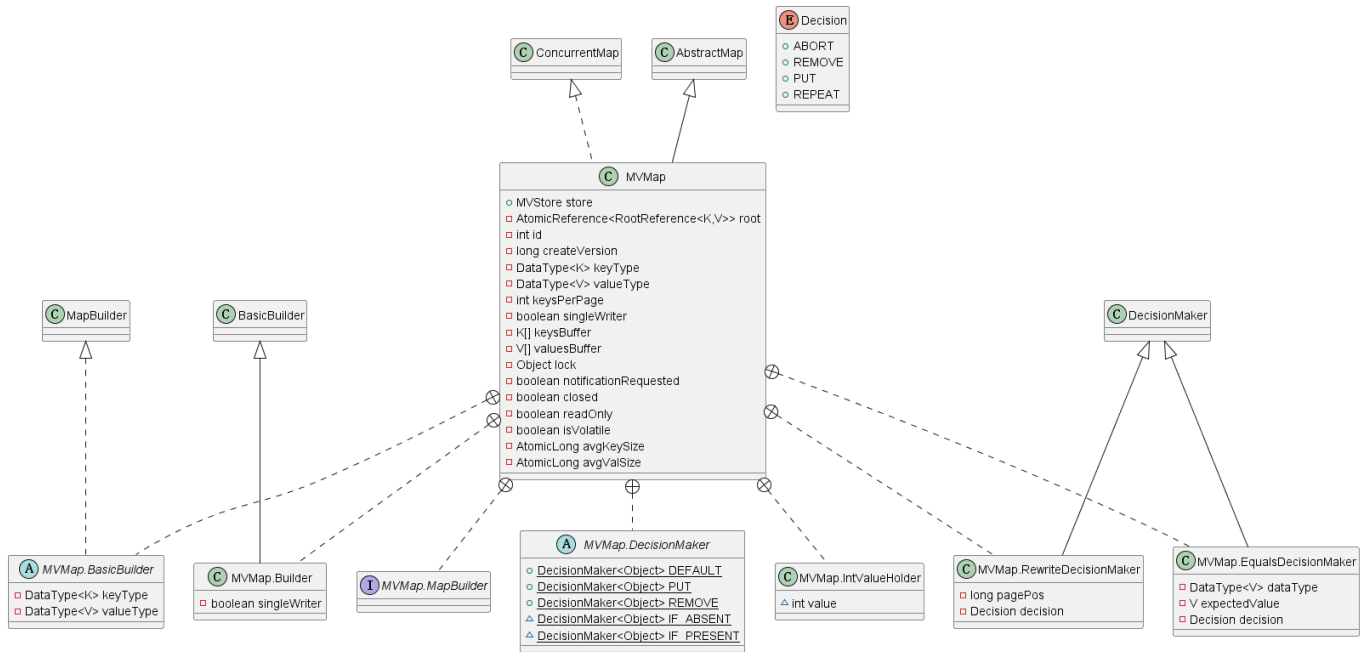


Page及其子类的成员变量（PlantUML绘制）

对于每一个Page，都有一个指向自身所在的MVMMap类成员变量`map`。`keys`以数组的形式保存键。Page中还有一个名叫`PageReference`的类，其作用相当于指针（引用），指向的就是内部的成员变量`page`。同时还需要记录以所指向的`page`为根的子树所包含的数据总数`count`。

而对于B+树，我们可以将它的结点分为两种：叶子节点和内部结点（非叶子结点）。于是这就产生了两个子类：`Leaf`和`NonLeaf`。对于叶子结点而言，只需要记录`value`数组存储值；对于内部结点，也就是`NonLeaf`，它需要用`PageReference`类数组保存孩子结点的引用，也就是`children`。`totalCount`用来保存孩子结点数目。而`incompleteNonLeaf`是`NonLeaf`的泛化，在对Page进行复制的时候，复制的Page可能不需要孩子结点引用（或者保持原状），`incompleteNonLeaf`起到一个过渡的作用，是一个中间过程产生的类。

说完Page，那么由Page构成的MVMMap就比较清楚了。MVMMap的UML类图如下：



MVMap及其相关类的成员变量（PlantUML绘制）

图中有许多构造器（Builder），这些不在我们的考虑范围之内；而DecisionMaker系列之前也有提及，它和operate方法一起，集成了对MVMap的数据操作。

MVMap类继承了AbstractMap类，实现了ConcurrentMap类接口。和Page类似，MVMap的成员变量中的store也是一个引用；root指向的是B+树的根结点（Page）。在这个层级，MVMap被赋予了id：这也很好理解，毕竟MVStore所存储的MVMap可以不止一个。比较有趣的是keysPerPage这个成员变量，在源码中它被默认赋予初值48。这也意味着每一个Page默认最多存储48个键，也就是说，MVMap的B+树默认是49阶的。我们可以找一个例子简单地验证一下：

```

MVStore s = MVStore.open(null);
MVMap map_1 = s.openMap("48keys");
MVMap map_2 = s.openMap("49keys");
for(int i = 0; i < 48; i++)
    map_1.put(i, "Hello");
for(int i = 0; i < 49; i++)
    map_2.put(i, "World");
  
```

我们创建了两个MVMap，一个含有48组数据，另一个含有49组。

```

map_1 = {MVMap@758} ""
  store = {MVStore@757}
  root = {AtomicReference@915} "RootReference(1387228415, v=0, owner=0, holdCnt=0, keys=48, append=0)"
    value = {RootReference@923} "RootReference(1387228415, v=0, owner=0, holdCnt=0, keys=48, append=0)"
      root = {Page$Leaf@925} "id: 1387228415\npos: 0\n0:Hello 1:Hello 2:Hello 3:Hello 4:Hello 5:Hello 6:Hello 7:Hello 8:Hello 9:Hello ...视图
  
```

48keys的数据结构

```

map_2 = {MVMap@759} ""
  > store = {MVStore@757}
  > root = {AtomicReference@983} "RootReference(748658608, v=0, owner=0, holdCnt=0, keys=49, append=0)"
    > value = {RootReference@991} "RootReference(748658608, v=0, owner=0, holdCnt=0, keys=49, append=0)"
      > root = {Page$NonLeaf@993} "id: 748658608\npos: 0\n[0] 24 [0]" ...视图
        > children = {Page$PageReference[2]@997}
          > 0 = {Page$PageReference@1000} "Cnt:24, pos:0 leaf, page:{id: 167185492\npos: 0\n0:World 1:World 2:World 3:World 4:W...
          > 1 = {Page$PageReference@1001} "Cnt:25, pos:0 leaf, page:{id: 592179046\npos: 0\n24:World 25:World 26:World 27:World..."

```

49keys的数据结构

从上面两幅图我们可以看到，`map_1`的根结点的类型是`Leaf`，存储了48组数据；而`map_2`的根结点类型是`NonLeaf`，其下有两个孩子结点，由此得证。

`keysPerPage`这个数设置得相当大，这保证了MVMap树的层数一般不超过3层，查找效率很高。

此外，MVStore的Map还有R树的实现：`MVRTreeMap`。而这种数据结构并不常用，这里不展开阐述。

2.1.2 block和Chunk

MVStore可以将数据持久化到硬盘，将数据存储在一个文件里。因此，我们需要对存储文件的数据进行分析。

首先来看存储文件的格式：

```
[ file header 1 ] [ file header 2 ] [ chunk ] [ chunk ] ... [ chunk ]
```

比较奇特的是，存储文件有两个文件头。官方文档这样解释：文件头更新的时候写操作可能会失败，从而损坏一个文件头，因此用两个文件头以作备用。

首先来了解一下`block`。`block`并不是某一具体的对象或类。实际上，`block`只是对存储空间的一个划分：我们将存储文件划分成整数个块，每一块我们就可以称之为`block`。`block`的大小在文件头中有定义。

关于文件头，以下是一个文件头的描述：

```
H:2,block:2,blockSize:1000,chunk:1,clean:1,created:184a7c079cd,format:2,version:1,
fletcher:8e4ba0e2
```

分析以下这些字段的意思：

- H：值固定为2，代表H2。
- block：最新block的起始数。（chunk会更新，后面将会介绍）
- blockSize：block的大小（用16进制表示——实际上这里所有的数均为16进制，单位为byte）
- chunk：chunk的id号（通常与version一致）
- clean：这个字段取决于上一次使用数据库是否完全关闭。如果上次使用未完全关闭（此时没有该字段），则尝试恢复。
- format：文件格式号。
- version：最新版本号。
- fletcher：弗莱彻校验和。

同时，文件头自身也占一个block。这一点很容易验证：

Address	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f	Dump
00000000	48	3a	32	2c	62	6c	6f	63	6b	3a	39	2c	62	6c	6f	63	H:2,block:9,bloc
00000010	6b	53	69	7a	65	3a	31	30	30	30	2c	63	68	75	6e	6b	kSize:1000,chunk
00000020	3a	38	2c	63	72	65	61	74	65	64	3a	31	38	34	61	37	:8,created:184a7
00000030	63	30	37	39	63	64	2c	66	6f	72	6d	61	74	3a	32	2c	c079cd,format:2,
00000040	76	65	72	73	69	6f	6e	3a	38	2c	66	6c	65	74	63	68	version:8,fletch
00000050	65	72	3a	31	38	63	36	37	61	38	31	0a	00	00	00	00	er:18c67a81.....

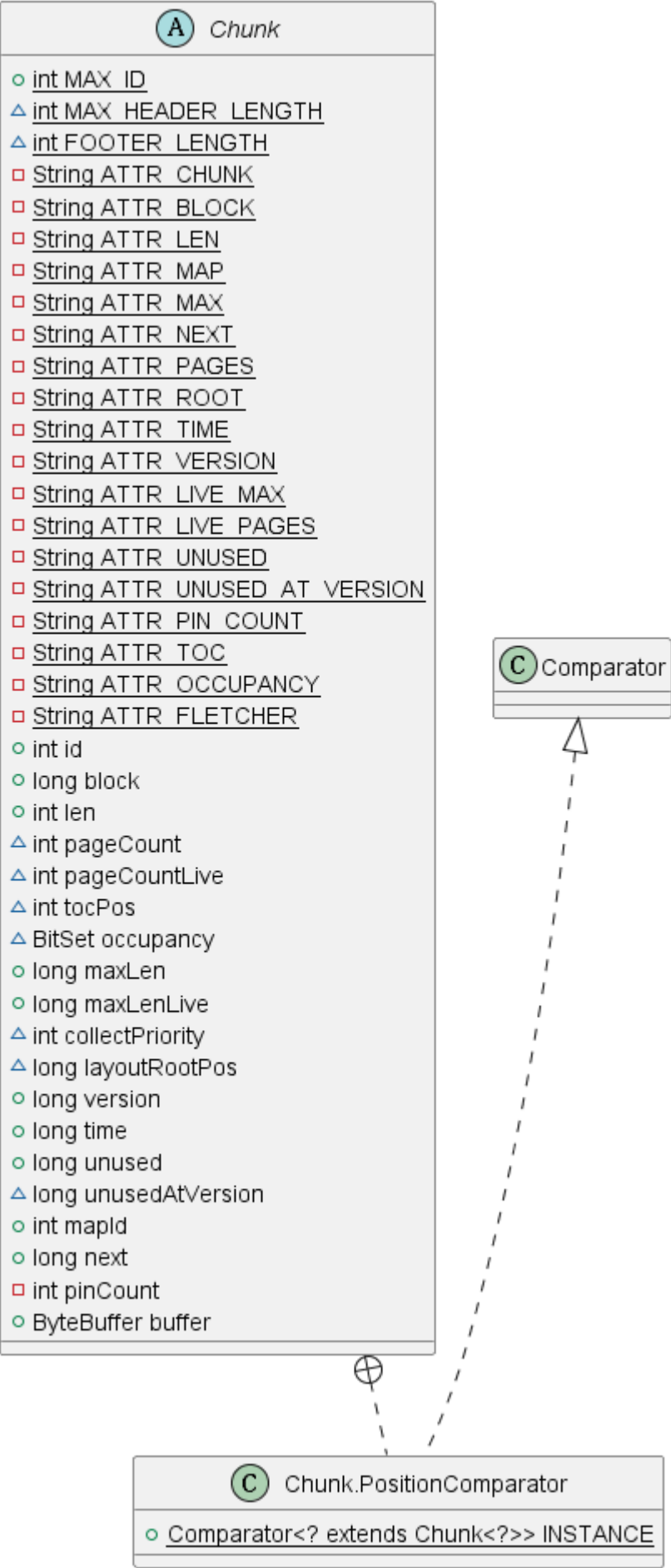
第一个文件头，首地址0x0000

00001000	48	3a	32	2c	62	6c	6f	63	6b	3a	39	2c	62	6c	6f	63	H:2,block:9,bloc
00001010	6b	53	69	7a	65	3a	31	30	30	30	2c	63	68	75	6e	6b	kSize:1000,chunk
00001020	3a	38	2c	63	72	65	61	74	65	64	3a	31	38	34	61	37	:8,created:184a7
00001030	63	30	37	39	63	64	2c	66	6f	72	6d	61	74	3a	32	2c	c079cd,format:2,
00001040	76	65	72	73	69	6f	6e	3a	38	2c	66	6c	65	74	63	68	version:8,fletch
00001050	65	72	3a	31	38	63	36	37	61	38	31	0a	00	00	00	00	er:18c67a81.....

第二个文件头，首地址0x1000

两个文件头首地址之差正好是一个block的大小。

文件头用来描述文件的重要属性，Chunk用来存储数据内容。Chunk的成员变量如下所示：



Chunk的成员变量 (PlantUML绘制)

注意到这里的Chunk为抽象类，它有两个继承类：**SFChunk**和**MFChunk**，分别代表单文件（SingleFile）和多文件（MultiFile）的Chunk。在这里我们只讨论单文件存储（这也是MVStore的默认持久化方式）的情形。

直接看Chunk的成员变量有点抽象，官方文档给出了Chunk在文件中的格式：

```
[ header ] [ page ] [ page ] ... [ page ] [ footer ]
```

我们可以看到，Chunk含有头（header）和脚（footer）。header的作用依旧是给出本Chunk的重要属性，footer则主要用于校验。以下给出一个header和footer的例子：

```
chunk:1,len:1,pages:6,max:420,map:3,root:4000010446,time:18,version:1,next:3,toc:4
5b
chunk:1,len:1,version:1,fletcher:90d3728c
```

在MVStore中，一个Chunk保存一个版本。MVStore的特色就是多版本存储，每个版本都是以Chunk的形式保存。可以粗略地理解为：有多少个版本，就有多少个Chunk。同时，MVStore采用日志化结构存储，在文件头中添加索引等方式寻址（例如指向最新版本的**version**字段）。实际上，**commit**方法更新数据库也是按Chunk保存的：每次commit相当于更新一次版本，也即增加一个Chunk。

2.1.3 元数据

元数据指的是描述数据的数据。在MVStore中有两个MVMap类型的元数据：**meta**和**layout**。笔者认为，直接给出图例能更方便地理解这两个对象：

```

▼ f meta = {MVMap@987} size = 4
  > "map.2" -> "name:48keys"
  > "map.3" -> "name:49keys"
  > "name.48keys" -> "2"
  > "name.49keys" -> "3"
```

meta示例

```

▼ f layout = {MVMap@1128} size = 11
  > "chunk.3" -> "chunk:3,block:4,len:1,pages:5,livePages:0,max:560,liveMax:0,map:3,next:5,root:c00000f2d0,time:567f40,unused:56f1ba,unusedAtVersion:5,version:3,toc:55e,occupancy:0"
  > "chunk.4" -> "chunk:4,block:5,len:1,pages:1,livePages:0,max:300,liveMax:0,map:3,next:6,root:100000029d2,time:567f40,unused:56815d,unusedAtVersion:4,version:4,toc:2db,occupancy:0"
  > "chunk.5" -> "chunk:5,block:6,len:1,pages:2,livePages:1,max:360,liveMax:60,map:3,next:7,root:14000003c52,time:56815d,unused:56815d,unusedAtVersion:5,version:5,toc:3e1,occupancy:02"
  > "chunk.6" -> "chunk:6,block:7,len:1,pages:5,livePages:0,max:760,liveMax:0,map:3,next:8,root:1800000f2d4,time:56f1ba,unused:5b77fc,unusedAtVersion:7,version:6,toc:76a,occupancy:0"
  > "chunk.7" -> "chunk:7,block:8,len:1,pages:1,livePages:0,max:600,liveMax:0,map:3,next:9,root:1c0000029d6,time:56ff53,unused:5b77fc,unusedAtVersion:7,version:7,toc:4e8,occupancy:0"
  > "chunk.8" -> "chunk:8,block:9,len:1,pages:5,livePages:0,max:960,liveMax:0,map:3,next:a,root:2000000f2d6,time:5b77fc,unused:66d1135,unusedAtVersion:8,version:8,toc:8c6,occupancy:0"
  > "chunk.9" -> "chunk:9,block:2,len:1,pages:5,livePages:0,max:960,liveMax:0,map:3,next:3,root:2400000f316,time:66d1135,unused:cc0228d,unusedAtVersion:9,version:9,toc:976,occupancy:0"
  > "meta.id" -> "1"
  > "root.1" -> "140000029c6"
```

layout示例

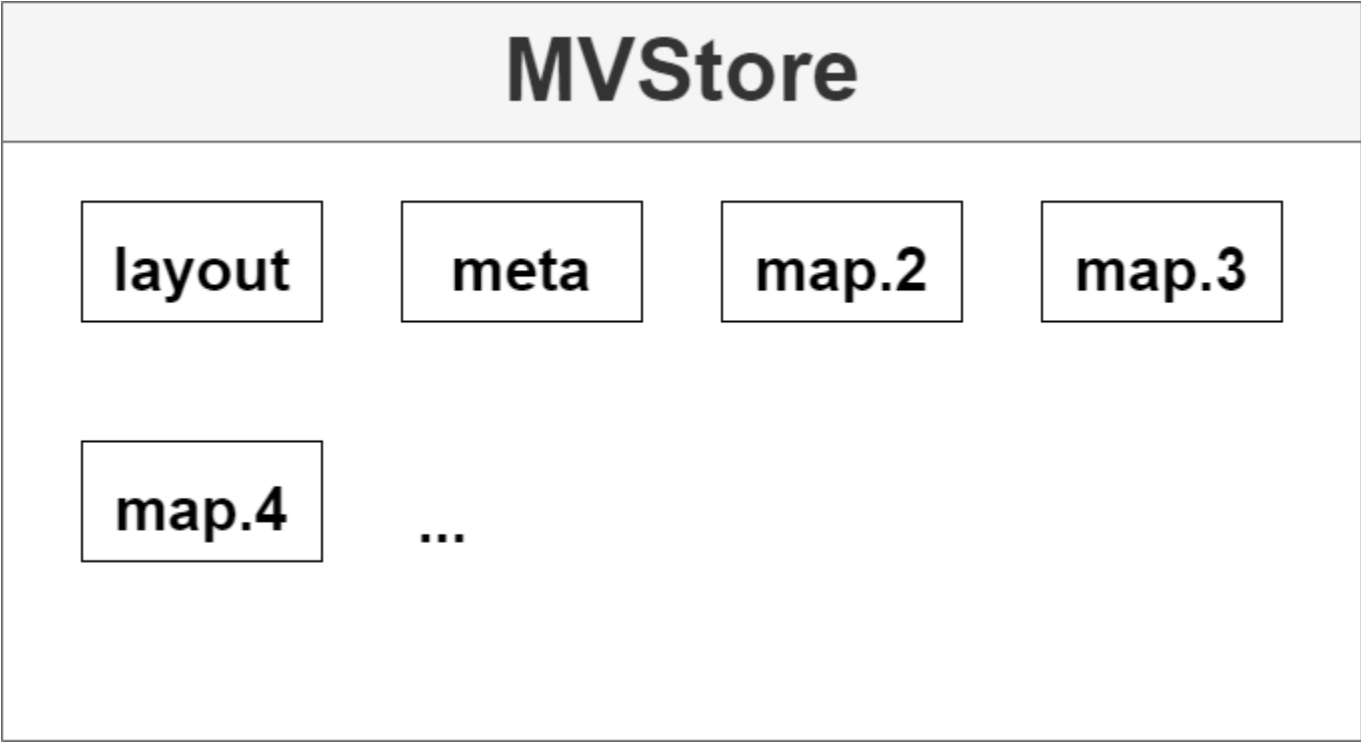
如上所示，**meta**是一个键值均为String类型的MVMap。它存储了所有MVMap数据的键值分别为**id->name**和**name->id**的键值对。（**id**：MVMap的id号；**name**：MVMap的名称）

而**layout**则更近一层，它描述的是文件存储的Chunk信息和MVMap信息（包括**meta**）。显而易见的是，**layout**在内存模式下不会存在。在图中我们可以看到，**layout**展示了**chunk.id**为2之后的所有Chunk信息（这

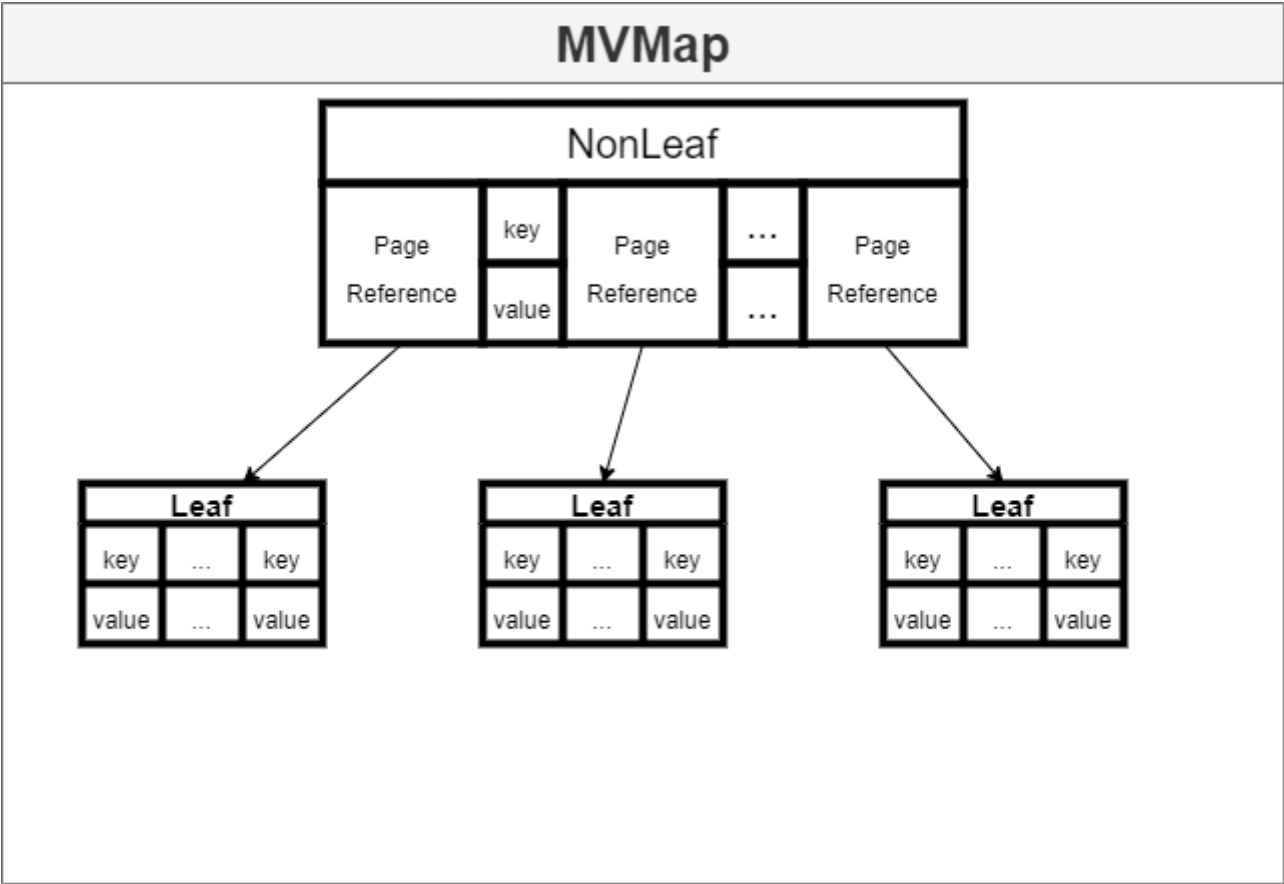
一点很好理解，因为`chunk.1`和`chunk.2`是两个文件头)。同时也解释了为什么`meta`存储的MVMap的id从2开始：`meta`的id为1。`root.id`表示每个MVMap的根结点，后面的`pos`表示地址，方便进行增删改查操作。

2.2 MVStore的结构图

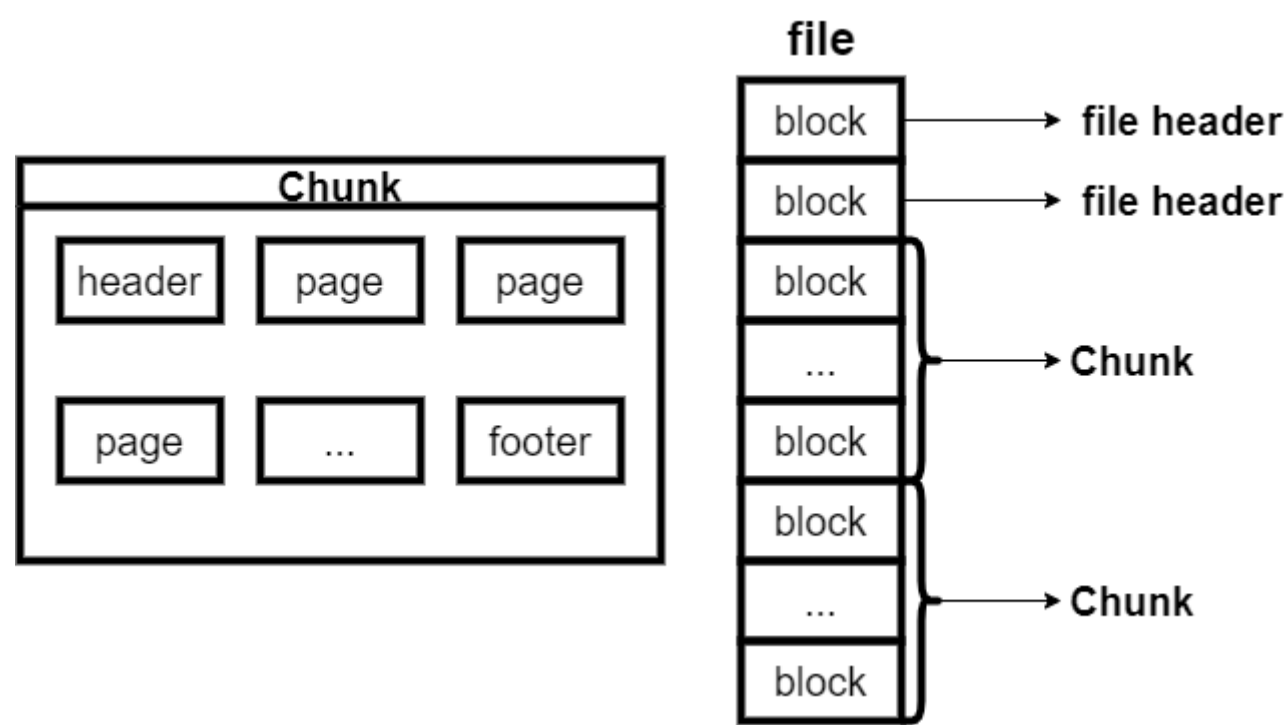
根据以上分析，我们可以大致画出MVStore的结构图。（只展示重要部分）



MVStore结构图



MVMap结构图



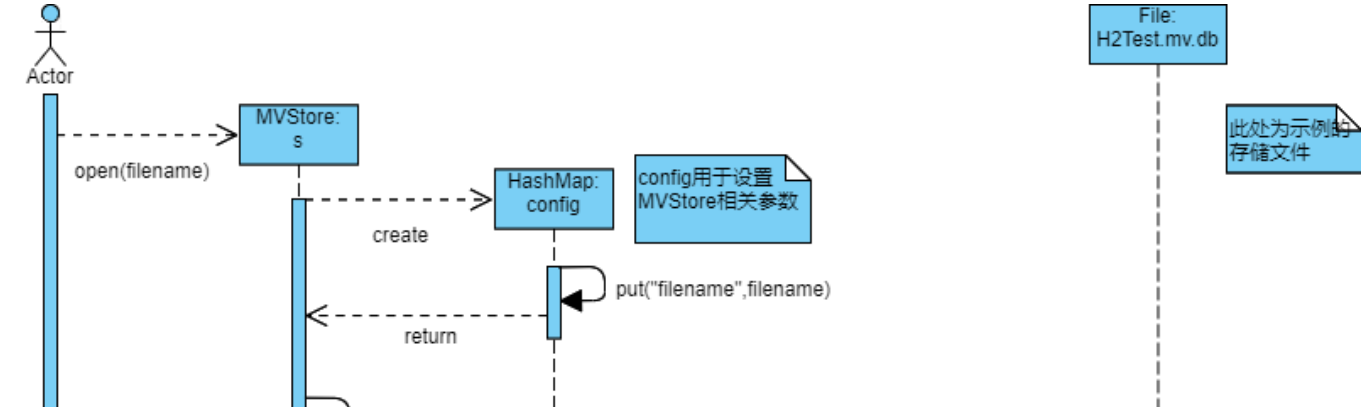
存储文件结构图

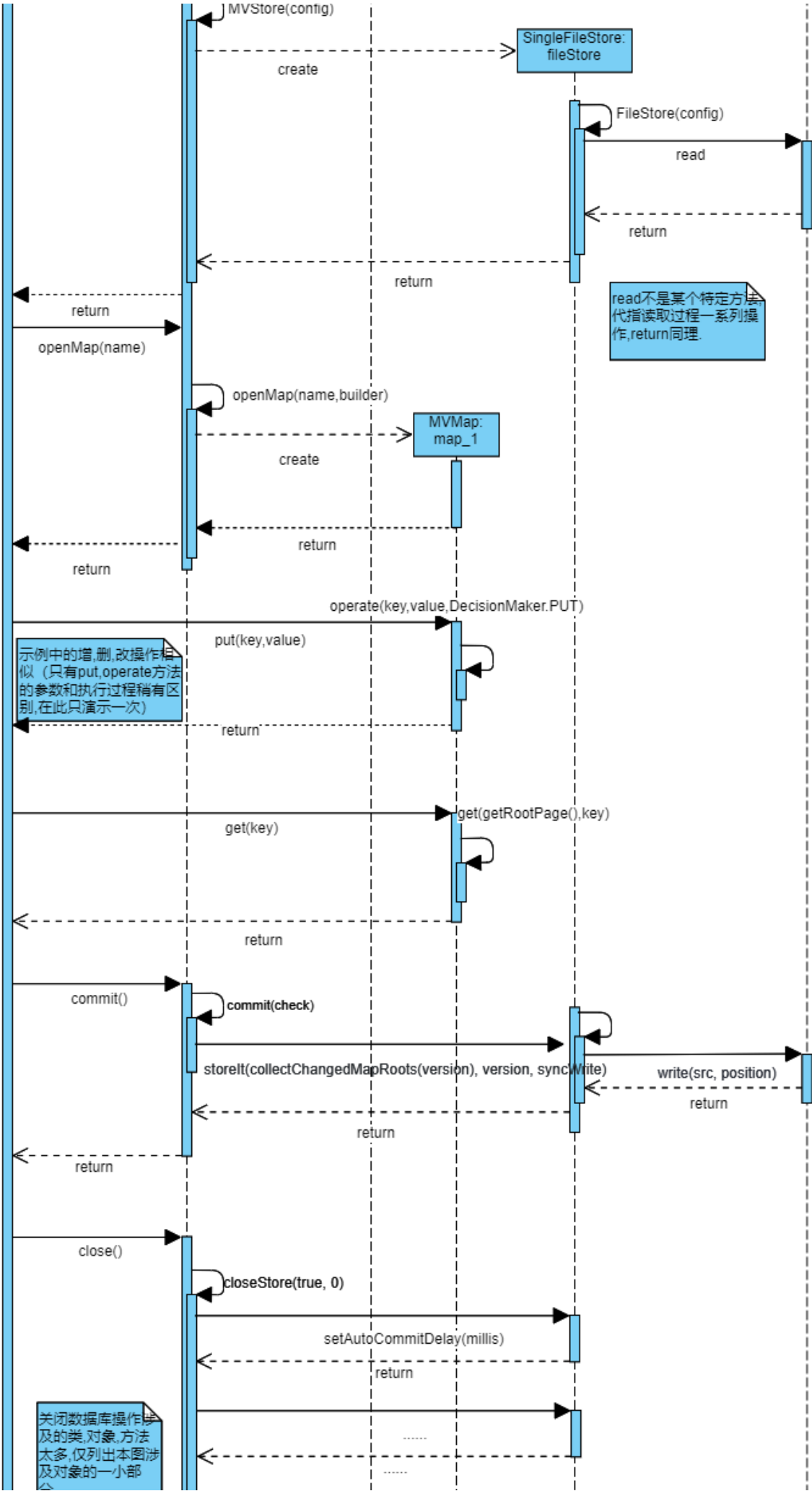
3.MVStore时序图

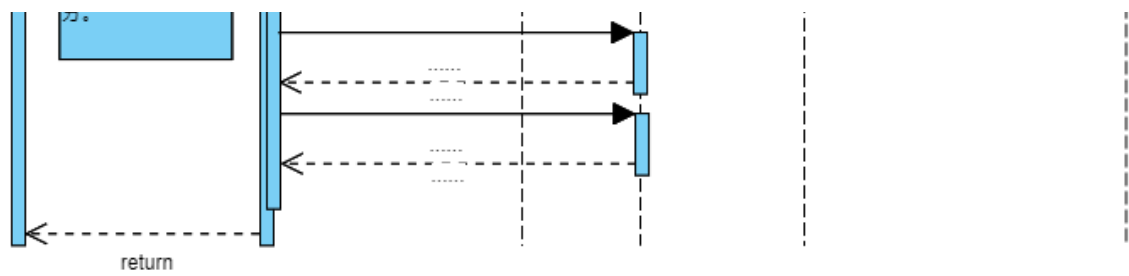
最后以一个简单例子为例，展示MVStore工作的时序图。

```
public static void main(String[] args) throws Exception {
    MVStore s = MVStore.open("E:/Java/H2Test.mv.db");
    MVMap<Integer, String> map_1 = s.openMap("map_1");
    map_1.put(2, "Winter");
    map_1.put(3, "Summer");
    map_1.put(3, "Autumn");
    map_1.remove(3);
    System.out.println(map_1.get(2));
    s.commit();
    s.close();
}
```

时序图如下所示：（忽略了许多类和操作方法，仅展示重要部分）







MVStore示例时序图