Chain33 KVMVCC存储插件介绍

复杂美科技 张振华

目录

[一、Chain33架构及存储插件 1](#_Toc683413819)

[1.1 Chain33架构 2](#_Toc1645960930)

[1.2 Chain33架构数据层的存储插件 2](#_Toc1783696119)

[1数据读写层插件 3](#_Toc569361362)

[2数据存储层插件 4](#_Toc2035943945)

[二、MAVL和MPT数据存储格式介绍 5](#_Toc189529868)

[2.1概念介绍 5](#_Toc839274967)

[1.Hash 5](#_Toc1968224495)

[2.Hash List 5](#_Toc127645300)

[3.AVL树（平衡二叉树） 6](#_Toc2071504556)

[4.Merkle树 6](#_Toc1590209840)

[2.2.Merkle树在区块链中的应用 7](#_Toc521080854)

[2.3.MAVL树 9](#_Toc1760815266)

[2.5.MPT树 10](#_Toc1172065493)

[1.Trie树 10](#_Toc595897861)

[2. PatriciaTrie 11](#_Toc712572886)

[3. Merkle Patricia Tree 11](#_Toc2105530236)

[4. StateDB 14](#_Toc1200744282)

[三、KVMVCC数据存储格式介绍 17](#_Toc1922743104)

[3.1 为何要设计KVMVCC的数据存储格式 17](#_Toc1038112981)

[3.2 KVMVCC的数据存储格式设计 17](#_Toc1258099725)

[四、KVMVCC与MAVL/MPT的对比总结 21](#_Toc250608039)

# 一、Chain33架构及存储插件

**Chain33是由复杂美开发并开源的领先公链架构。**

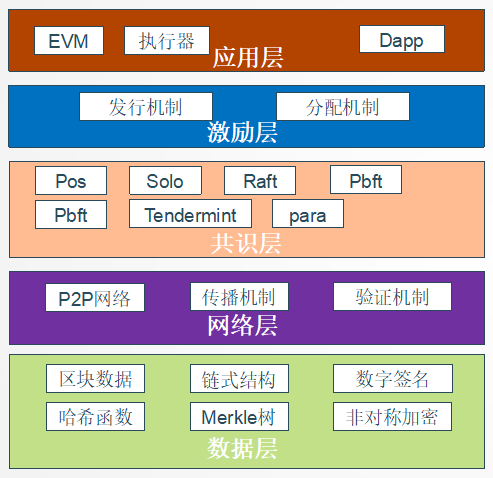
**代码及文档信息如下：**

**开源代码：https://github.com/33cn**

**开发者文档：<https://chain.33.cn/>**

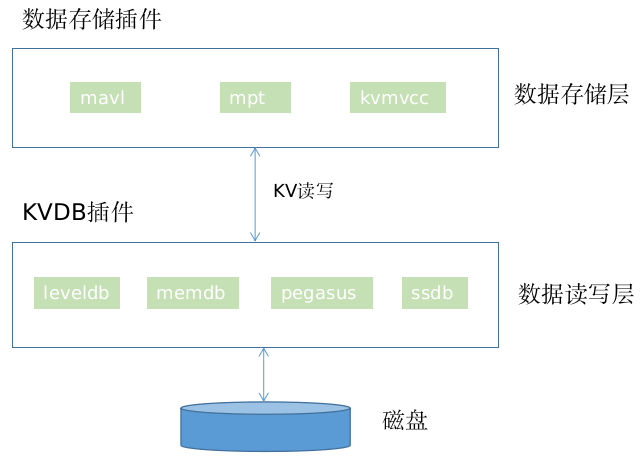
## 1.1 Chain33架构

Chain33基于插件式的架构设计，这样做的好处是把扩展功能从内核中剥离出来，使内核相对精简，容易实现，进而更稳定，而扩展插件以一种松耦合的方式和内核结合。



## 1.2 Chain33架构数据层的存储插件

Chain33对于区块链状态数据的存储又可以分为两层：数据存储层和数据读写层。



数据存储层定义了区块链状态数据的存储数据结构，负责数据的组织和处理。

数据读写层则定义了具体用哪一种key-value数据库来进行key-value数据持久化到文件系统及从文件系统中读取。一般要求key-value数据库提供性能很高的随机写数据到磁盘的能力和性能尚可的从磁盘随机读数据的能力(区块链程序写区块数据到磁盘的场景更多，从磁盘读数据的场景较少，所以对写数据能力要求更高)。

这两层的处理效率共同影响了区块链整体的数据读写处理性能。

### 1数据读写层插件

从Chain33开源的代码库中可以看到，Chain33支持如下的key-value数据库：

db

├── go\_badger\_db.go

├── go\_level\_db.go

├── go\_mem\_db.go

├── go\_pegasus.go

└── go\_ssdb.go

这些插件的简要介绍如下：

#### MemDB

是Chain33内置的一个简单实现的内存key-value数据库，用于开发自测使用。

#### LevelDB

是Google实现的一个非常高效的键值对数据库，其中键值都是二进制的，目前能够支持十亿级别的数据量，在这个数据量下还有非常高的性能。

特点：  
1、key和value都是任意长度的字节数组；  
2、entry（即一条K-V记录）默认是按照key的字典顺序存储的，当然开发者也可以重载这个排序函数；  
3、提供的基本操作接口：Put()、Delete()、Get()、Batch()；  
4、支持批量操作以原子操作进行；  
5、可以创建数据全景的snapshot(快照)，并允许在快照中查找数据；  
6、可以通过前向（或后向）迭代器遍历数据（迭代器会隐含的创建一个snapshot）；  
7、自动使用Snappy压缩数据；  
8、可移植性；

限制：  
1、非关系型数据模型（NoSQL），不支持sql语句，也不支持索引；  
2、一次只允许一个进程访问一个特定的数据库；  
3、没有内置的C/S架构，但开发者可以使用LevelDB库自己封装一个server；

#### SSDB

替代Redis数据库, Redis的100 倍容量。

LevelDB网络支持, 使用C/C++开发，有Go语言版本。

Redis API兼容, 支持Redis客户端。

适合存储集合数据, 如list, hash, zset...

客户端API支持的语言包括: [C++](http://ssdb.io/docs/cpp/), [PHP](http://ssdb.io/docs/zh_cn/php/), Python, [Java](http://ssdb.io/docs/java/), Go

持久化的队列服务。

主从复制, 负载均衡。

Pegasus  
是小米云存储团队开发的一个分布式Key-Value存储系统，最初的动机是弥补HBase在可用性和性能上的不足。Pegasus系统的Server端完全采用C++语言开发，使用 [PacificA](https://www.microsoft.com/en-us/research/publication/pacifica-replication-in-log-based-distributed-storage-systems/) 协议支持强一致性，使用 [RocksDB](https://github.com/facebook/rocksdb) 作为单机存储引擎。

#### Badger

是一个基于LSM tree的高速数据库引擎，宣称加载数据库要比LevelDB快2.5–111倍，随机lookup要快1.6-14倍；比RocksDB快3.5倍，对于值的大小从128B到16KB，数据加载Badger比LevelDB快0.86 - 14倍。

Chain33实际部署时，可以根据不用的应用场景（比如存储类型、读写性能要求等）进行配置选择不同的key-value数据库作为数据读写层，Chain33默认配置选择LevelDB。

### 2数据存储层插件

从Chain33开源的代码库（https://github.com/33cn/chain33）中可以看到，Chain33默认支持的数据存储层插件是mavl：

store

├── init

└── mavl

从Chain33开源的插件代码库（https://github.com/33cn/plugin）中可以看到，Chain33还可以支持MPT、KVDB、MVCCKVDB等多种数据存储层插件实现：

store/

├── kvdb

├── kvmvcc

├── kvmvccmavl

└── mpt

其中kvdb为Chain33内部开发测试用，只保留最新版本的kv数据，不能查历史版本的kv数据；kvmvccmavl为从mavl向kvmvcc切换过渡使用。

对于MPT、MAVL、KVMVCC几种实现插件实现方式的介绍和对比，在下面章节详细展开。

# 二、MAVL和MPT数据存储格式介绍

## 2.1概念介绍

首先介绍几个概念，方便大家理解。

区块链数据存储，广泛用到Hash、Merkle树等概念。

### 1.Hash

Hash是一个把任意长度的数据映射成固定长度数据的函数。

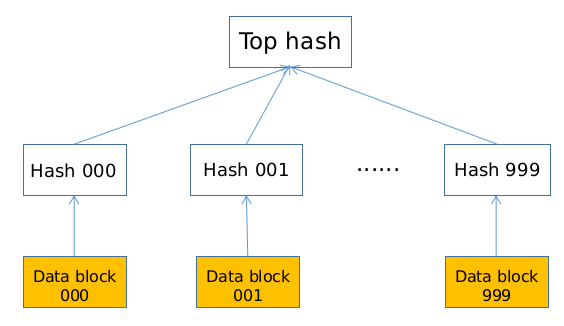
例如，对于数据完整性校验，最简单的方法是对整个数据做Hash运算得到固定长度的Hash值，然后把得到的Hash值公布在网上，这样用户下载到数据之后，对数据再次进行Hash运算，比较运算结果和网上公布的Hash值进行比较，如果两个Hash值相等，说明下载的数据没有损坏。可以这样做是因为输入数据的学细微改变就会引起Hash运算结果的面目全非，并且根据Hash值反推原始输入数据的特征是困难甚至不可行的。

### 2.Hash List

在P2P网络中作数据传输的时候，会同时从多个机器上下载数据，而且很多机器可以认为是不稳定或者不可信的。为了校验数据的完整性，更好的办法是把大的文件分割成小的数据块(例如，分割成2K为单位的数据块)。这样的好处是，如果小块数据在传输过程中损坏了，那么只要重新下载这一小块数据就行了，不用重新下载整个文件。

怎么确定小的数据块没有损坏呢？只需要为每个数据块做Hash。

比如BT下载的时候，在下载到真正数据之前，我们会先下载一个Hash列表。把每个小块数据的Hash值拼到一起，然后对这个长字符串在作一次Hash运算，这样就得到Hash列表的根Hash(Top Hash或者Root Hash)。下载数据的时候，首先从可信的数据源得到正确的根Hash，就可以用它来校验Hash列表了，然后通过校验后的Hash列表校验数据块。



### 3.AVL树（平衡二叉树）

由前苏联的数学家Adelse-Velskil和Landis在1962年提出的高度平衡的二叉树，根据科学家的英文名也称为AVL树。它具有如下几个性质：

（1）可以是空树。

（2）假如不是空树,任何一个结点的左子树与右子树都是平衡二叉树,并且高度之差的绝对值不超过1。

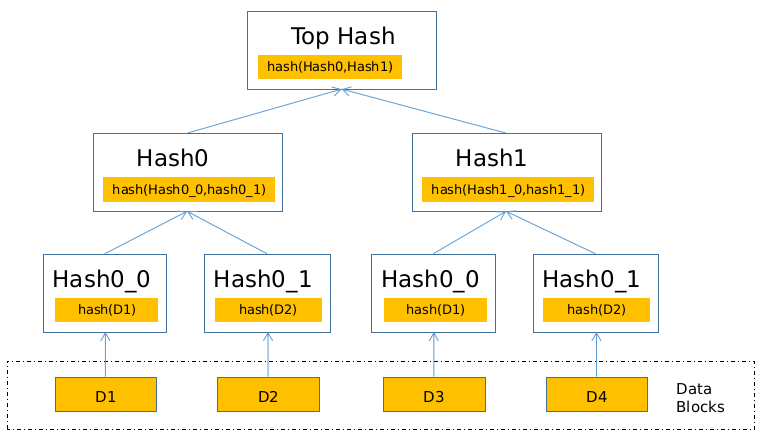
平衡树是同种元素序列情况下的深度最小的二叉排序树。这可以减少二叉树元素查找的深度，从而提升平均查找效率。付出的代价是在插入元素时，需要维护树的平衡，可能要通过旋转来调整元素的位置。

### 4.Merkle树

通常也被称作Hash Tree，顾名思义，就是存储hash值的一棵树。Merkle树的叶子是数据块(例如，文件或者文件的集合)的hash值。非叶节点是其对应子节点串联字符串的 hash。

Merkle Tree可以看做Hash List的泛化(Hash List 可以看作一种特殊的Merkle Tree，即树高为2的多叉Merkle Tree)。

在最底层，和哈希列表一样，我们把数据分成小的数据块，有相应地哈希和它对应。但是往上走，并不是直接去运算根哈希，而是把相邻的两个哈希合并成一个字符串，然后运算这个字符串的哈希，这样每两个哈希就结婚生子，得到了一个“子哈希”。如果最底层的哈希总数是单数，那到最后必然出现一个单身哈希，这种情况就直接对它进行哈希运算，所以也能得到它的子哈希。于是往上推，依然是一样的方式，可以得到数目更少的新一级哈希，最终必然形成一棵倒挂的树，到了树根的这个位置，这一代就剩下一个根哈希了，我们把它叫做Merkle Root。



哈希树的优势在于，它能够对大量的数据内容迅速作出高效且安全的验证。假设一个hash tree中有n个叶子节点，如果想要验证其中一个叶子节点是否正确（即该节点数据属于源数据集合并且数据本身完整），所需哈希计算的时间复杂度是是O（log(n)），相比之下hash list大约需要时间复杂度O(n)的哈希计算，hash tree的表现无疑是优秀的。

Merkle Tree和Hash List的主要区别：

Merkle Tree可以直接下载并立即验证其中的一个分支。因为可以将文件切分成小的数据块，这样如果有一块数据损坏，仅仅重新下载这个数据块就行了。如果文件非常大，Merkle tree可以一次下载一个分支，然后立即验证这个分支，如果分支验证通过，就可以下载数据了。而Hash list只有下载整个hash list才能验证。相对于Hash List，Merkle Tree的明显的一个好处是可以单独拿出一个分支来(作为一个小树)对部分数据进行校验，这个很多使用场合就带来了Hash List所不能比拟的方便和高效。

## 2.2.Merkle树在区块链中的应用

要了解 Merkle 树的使用，先要了解一下区块链中每个区块的数据结构，下面以比特币的数据结构为例说明。

a) 区块：

数据通过称之为区块(block)的文件，永久记录在数字货币网络上。它们好比是一个银行交易账本。新的区块会被添加到记录(区块链)的末端，而且一旦书写就很难修改或移除。

b) 区块结构：

|  |  |  |
| --- | --- | --- |
| 字节大小 | 名称 | 说明 |
| 4字节 | 区块大小 | 用字节表示的该字段之后的区块大小 |
| 80字节 | 区块头 | 组成区块头的几个字段 |
| 1-9(可变整数) | 交易计数器 | 交易的数量 |
| 可变的 | 交易 | 记录在区块里的交易信息 |

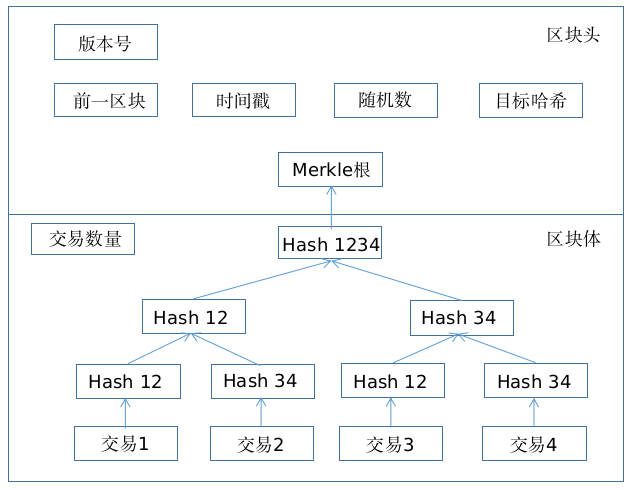
c) 区块头:

区块头由三组区块元数据组成。第一组引用父区块哈希值的数据，这组元数据用于将该区块与区块链中前一区块相连接。第二组元数据，即难度、时间戳和nonce，与挖矿竞争相关 。第三组元数据是Merkle树根(一种用来有效地总结区块中所有交易的数据结构)

d) 区块头结构:

|  |  |  |
| --- | --- | --- |
| 大小 | 名称 | 说明 |
| 4字节 | 版本 | 版本号,用于跟踪软件/协议的更新 |
| 32字节 | 父区块哈希值 | 引用区块链中父区块的哈希值 |
| 32字节 | Merkle根 | 该区块中交易的Merkle 树根的哈希值 |
| 4字节 | 时间戳 | 该区块产生的近似时间(精确到秒的 Unix 时间戳) |
| 4字节 | 难度目标 | 该区块工作量证明算法的难度目标 |
| 4字节 | Nonce | 用于工作量证明算法的计数器 |

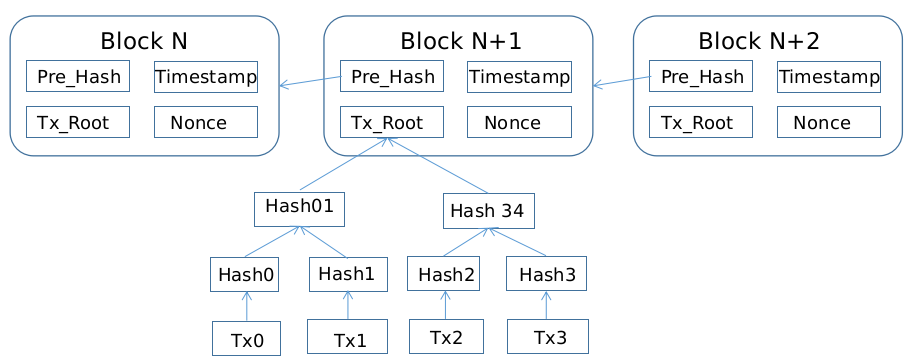
如下图,数据区块由区块头和区块体两部分组成:



从图中我们可以看出 Merkle 树被应用在了交易的存储上。每笔交易都会生成一个hash 值，然后不同的hash值向上继续做hash运算，最终生成唯一的Merkle根。并把这个 Merkle 根放入数据区块的区块头。利用Merkle树的特性，以确保每一比交易都不

可伪造和没有重复交易。

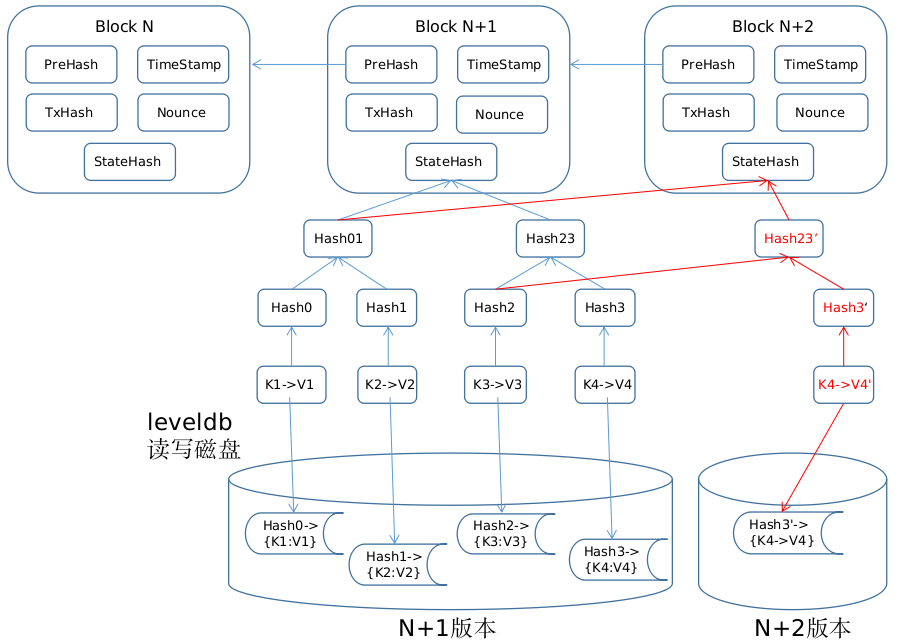
还可以从下图从整体上认识一下Merkle树在区块中的位置:



## 2.3.MAVL树

MAVL树是一个简化了的二叉平衡树，数据只保存在叶子节点上，分支节点只存储密匙。MAVL树的设计吸取 Merkle树和avl平衡二叉树各自的优势从而简化了算法同时保持了Merkle散列轨迹的短路。

Chain33默认的状态数据存储StateDB以Mavl结构进行组织，并最终以kv格式存储到kvdb（比如leveldb）中。



StateDB就是一棵MAVL树，在设计上并不会每生成一个块的时候都复制一份，而是重用树节点，在新块生成的时候，只改变发生状态变化的节点，然后引用原来未变化的节点，重新计算变化节点到根路径上所有节点的哈希，即相当于生成一颗新树（State Root Hash发生了变化）：

有了StateDB的这种设计，在区块链代码中的业务实现逻辑就会非常简单，如果需要修改状态数据（账户余额等），直接操作内存中的StateDB即可，在块生成之前，系统会根据变更的信息，重新生成哈希，并将变更写到数据库。

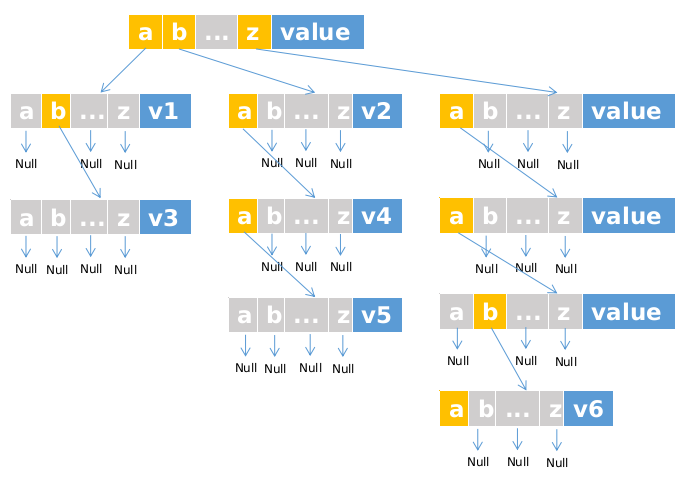
StateDB的这种设计使用起来非常方便，就像平行宇宙，当数据发生变化时，复用不变数据，再加上变化数据，生成一个新的MAVL树。

## 2.5.MPT树

Merkle Patricia Tree (MPT)是以太坊用来存储区块数据的核心数据结构。最简单理解是一个倒置的树形结构，每个节点可能有若干个子节点。它源自于Trie结构，又分别继承了PatriciaTrie和MerkleTree的优点，并基于内部数据的特性，设计了全新的节点体系和插入/载入机制。

### 1.Trie树

又称为字典树或者前缀树(prefix tree)，属于查找树的一种，其中的键通常是字符串，常用于存储Key-value数据结构。它与平衡二叉树的主要不同点包括：每个节点数据所携带的key不会存储在Trie的节点中，而是通过该节点在整个树形结构里的位置来体现；同一个父节点的子节点共享该父节点的key作为它们各自key的前缀，节点对应的key是根节点到该节点路径上的所有节点key值前后拼接而成，节点的value值就是该key对应的值。因此根节点key为空；待存储的数据只存于叶子节点中，非叶子节点帮助形成叶子节点key的前缀。如果key是英文单词，trie的每个节点就是一个长度为27的指针数组，index0-25代表a-z字符，26为标志域。



上图存储的数据如下：

[‘a’] = V1

[‘ab’] = V3

[‘b’] = V2

[‘ba’] = V4

[‘baa’] = V5

[‘zaab’] = V6

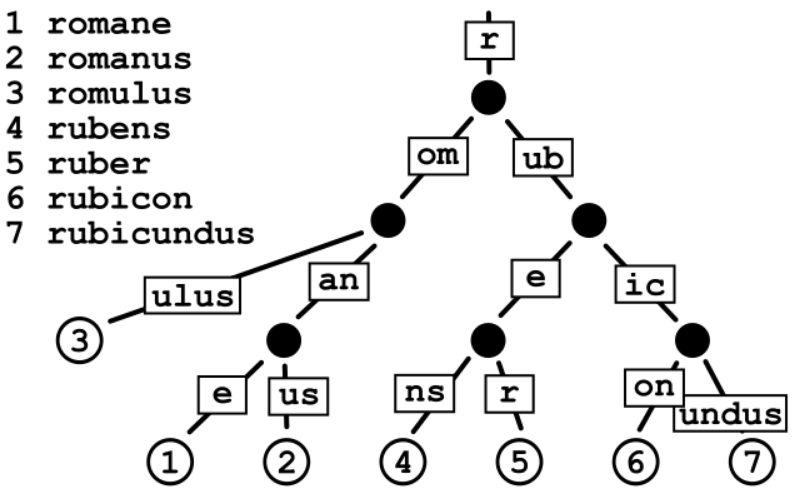
从上面可以看出zaab这个key，没有和任何其他key共享字段，但是却产生了6层，这种无用的深度增加有什么方法减呢？Patricia Tree就可以解决这个问题

### PatriciaTrie

又被称为RadixTree或紧凑前缀树(compact prefix tree)，是一种空间使用率经过优化的Trie。

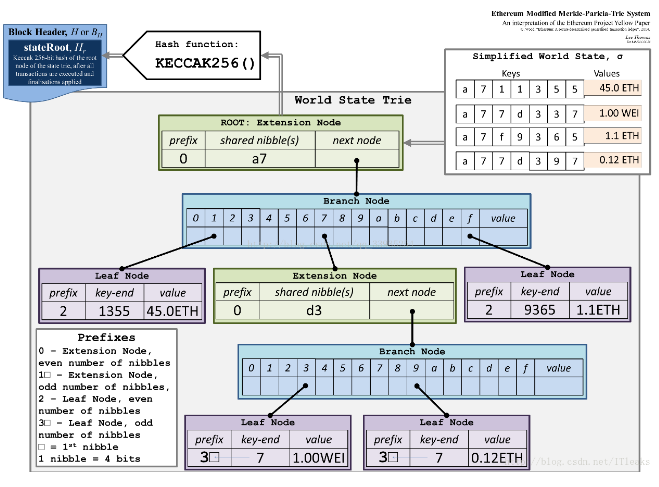
上面tries出现的问题的根本原因是每个前置节点只能表示一个字母，key有多长，树的深度就会多长，不管这个key有没有和其他key共享部分key。因而允许一个节点表示变长的key就可以解决这个深度，

PatriciaTrie里如果存在一个父节点只有一个子节点，那么这个父节点将与其子节点合并。这样可以缩短Trie中不必要的深度，大大加快搜索节点速度。

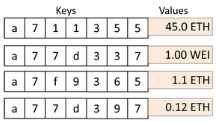


### Merkle Patricia Tree

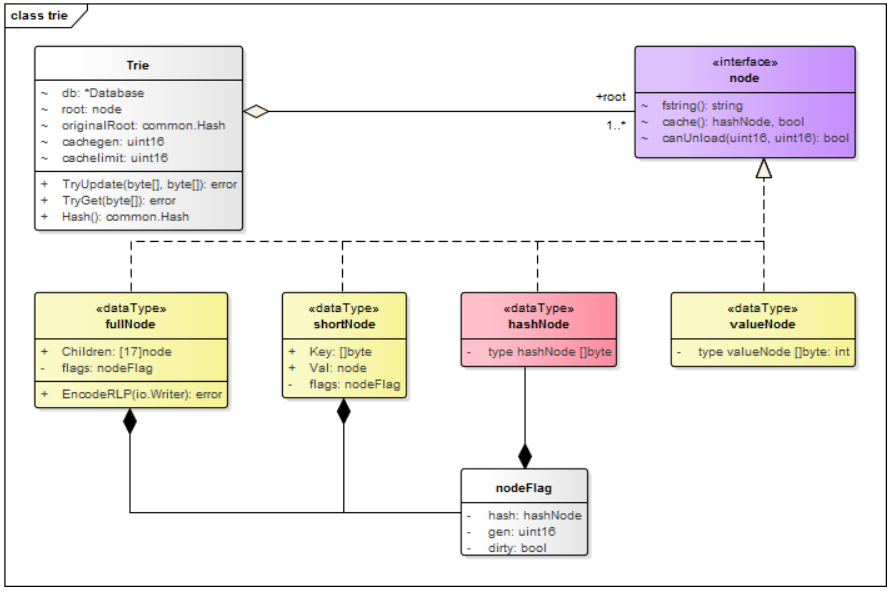
以太坊官方的MPT树结构图：



上图存储的key-value如下：



在以太坊中实现的MPT模型如下：



Merkle Patricia Tree有4种类型的节点，各种节点的介绍如下：

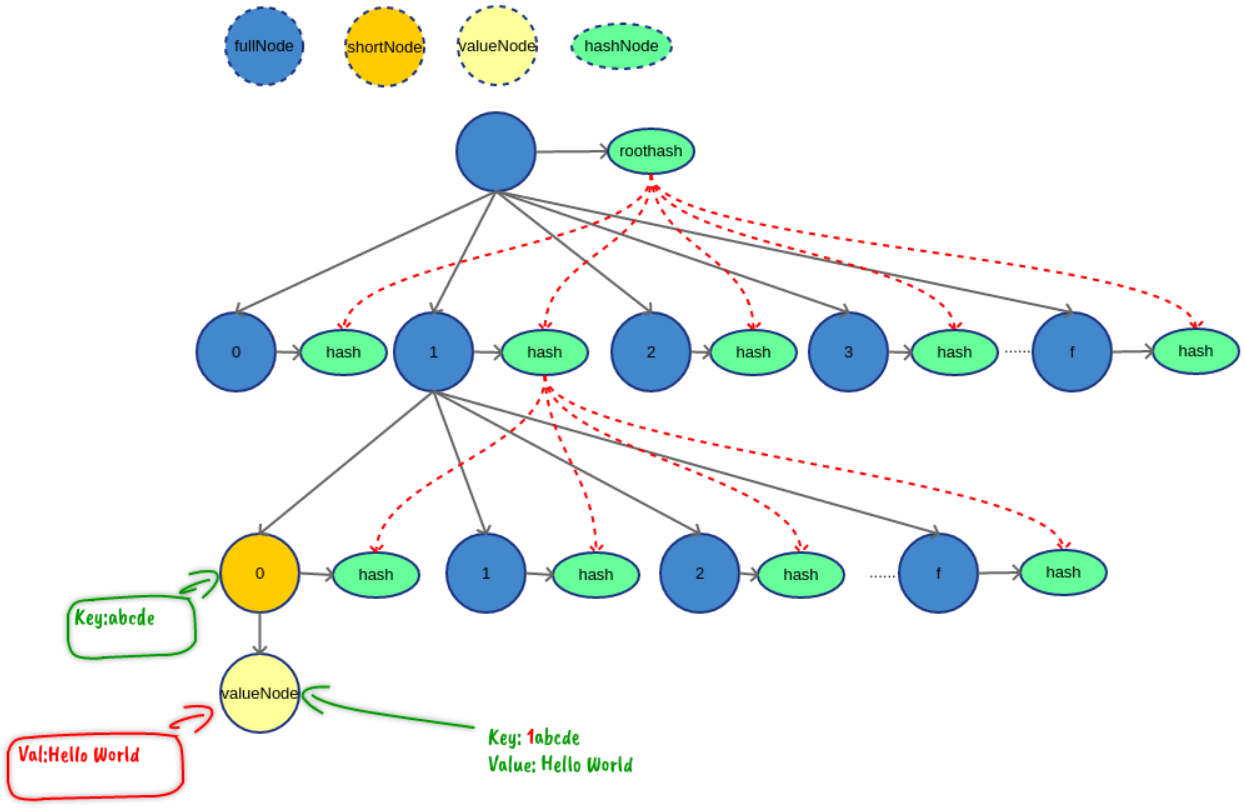
fullNode 也叫分支节点（对应上面MTP树结构图中的branch node），是一个可以携带多个子节点的父(枝)节点。它有一个容量为17的node数组成员变量Children，数组中前16个空位分别对应16进制(hex)下的0-f；第17位，留给该fullNode的数据部分。因为MPT树中的key被编码成一种特殊的16进制的表示，再加上最后的value，所以分支节点是一个长度为17的list，前16个元素对应着key中的16个可能的十六进制字符，如果有一个[key,value]对在这个分支节点终止，最后一个元素代表一个值，即分支节点既可以搜索路径的终止也可以是路径的中间节点。分支节点的父亲必然是extension node。

shortNode 也叫扩展节点（对应上面MTP树结构图中的extension node），是一个仅有一个子节点的节点。它的成员变量Val指向一个子节点，而成员Key是一个任意长度的字符串(字节数组[]byte)。对应PatriciaTrie的压缩节点。也是[key，value]的一个键值对，但是这里的value是其他节点的hash值，通过hash链接到其他节点。

valueNode 也叫叶子节点（对应上面MTP树结构图中的leaf node），不带子节点。在使用中，valueNode就是所携带数据部分的RLP哈希值，表示为[key,value]的一个键值对。和前面的英文字母key不一样，这里的key都是16编码出来的字符串，每个字符只有0-f 16种，value是RLP编码的数据。

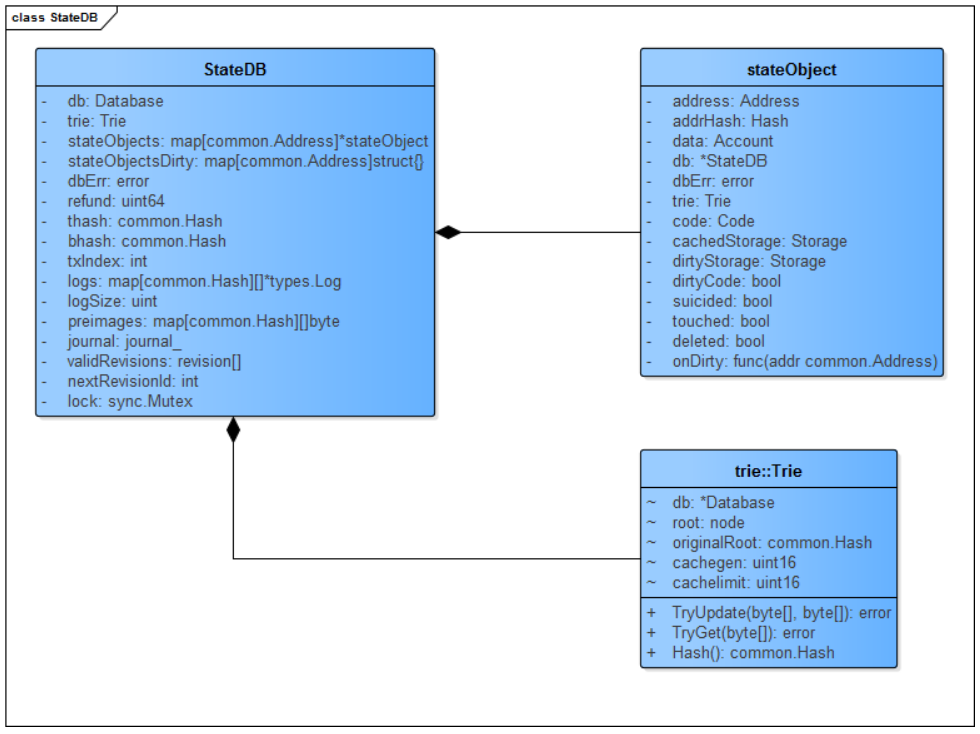
hashNode 在MPT中，hashNode几乎不会单独存在，而是以nodeFlag结构体的成员(nodeFlag.hash)的形式，被fullNode和shortNode间接持有。它为此节点以及下层节点所对应的哈希值，与Merkle树中节点的哈希用法相同。而root节点的hashNode，则为整个树的哈希。

其对应的实现示例如下图所示：



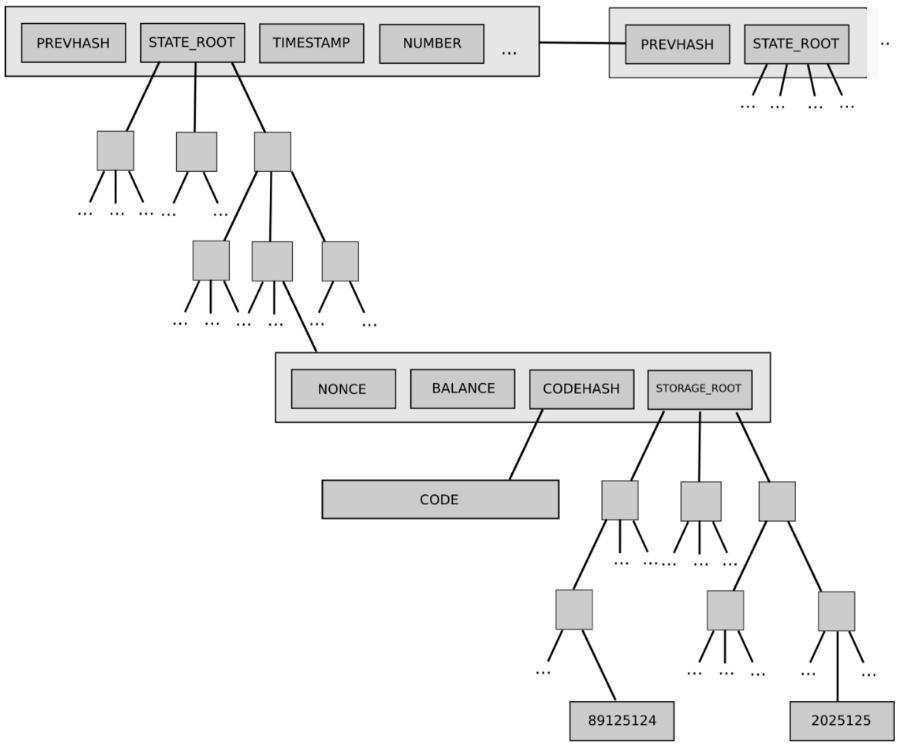
### StateDB

在以太坊中，设计了一个被称之为“世界状态”的对象，它用于保存系统内存在的所有账户的即时状态信息，可以直接查找。在以太坊中的实现模型定义如下：

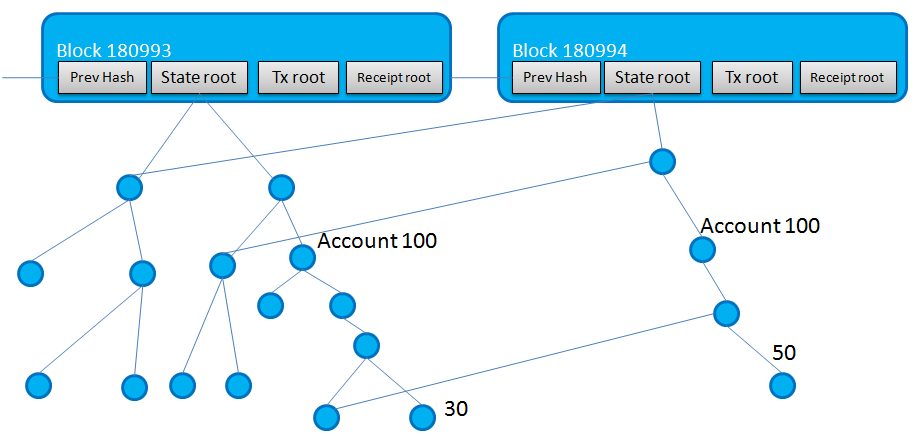


从上图中可以看出，StateDB中的stateObjects字典属性，保存了系统全局中所有账户的信息，所以称之为“世界状态”。

这棵状态树在区块中的存储示意如下图所示，STATE\_ROOT即为StateDB中的Root Hash：



StateDB就是一棵MPT树，在设计上并不会每生成一个块的时候都复制一份，而是重用树节点，在新块生成的时候，只改变发生状态变化的节点（通过写dirty标记），然后引用原来未变化的节点，重新计算变化节点到根路径上所有节点的哈希，即相当于生成一颗新树（State Root Hash发生了变化）：



有了StateDB的这种设计，在区块链代码中的业务实现逻辑就会非常简单，如果需要修改账户数据（余额变更、新增合约、废弃合约等），直接操作内存中的StateDB即可，在块生成之前，系统会根据变更的信息，重新生成哈希，并将变更写到数据库。

StateDB的这种设计使用起来非常方便，就像平行宇宙，当数据发生变化时，复用不变数据，再加上变化数据，生成一个新的MPT树。但是也存在一定的问题，因为它保存了所有的数据，随着时间的增长，这棵树会越来越大，占用越来越多的磁盘空间。

Chain33中，移植了以太坊的MPT树实现作为插件，也可以选择使用MPT树结构存储状态数据，和MAVL的效果比较类似。

# 三、KVMVCC数据存储格式介绍

## 3.1 为何要设计KVMVCC的数据存储格式

当前通过默克尔树（MAVL或者MPT）进行数据查询的缺陷在于，查询一笔交易的数据需要通过多次读操作来完成，例如，对于一颗20层的默克尔树，查询一个叶子节点的数据需要进行20次读操作来完成，导致数据查询的效率仅为普通数据库的查询效率的1/20，对于每秒能完成10万次读操作的系统，每秒仅能读取5000笔交易的数据，大幅限制了系统的读取性能。写数据时，同样要加载树型分支上的多个节点数据，并最终要在更新以后写入到磁盘，这里面的操作消耗也是比较大的。

于是，借鉴了数据库设计中的MVCC理念（Multi-Version Concurrency Control 多版本并发控制），我们设计了Chain33的KVMVCC的数据存储格式，用于改善MAVL或者MPT结构中存在的低效的问题，更好的满足区块链数据增长到一定规模后的保持较高的数据读写性能。

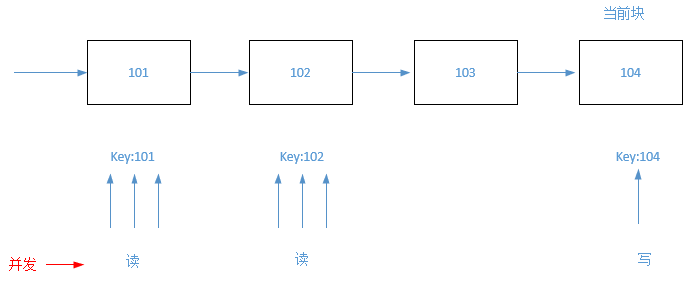
## 3.2 KVMVCC的数据存储格式设计

首先，区块链数据具有如下的特点：

（1）历史数据多（历史区块及交易数据），当前修改的数据少（当前正在生成的区块及交易数据）；

（2）历史数据不能修改，修改只发生在当前正在生成的区块上；

（3）读历史数据的比例高（对历史区块数据的读取），写数据的比例低（新生成区块中修改数据）；

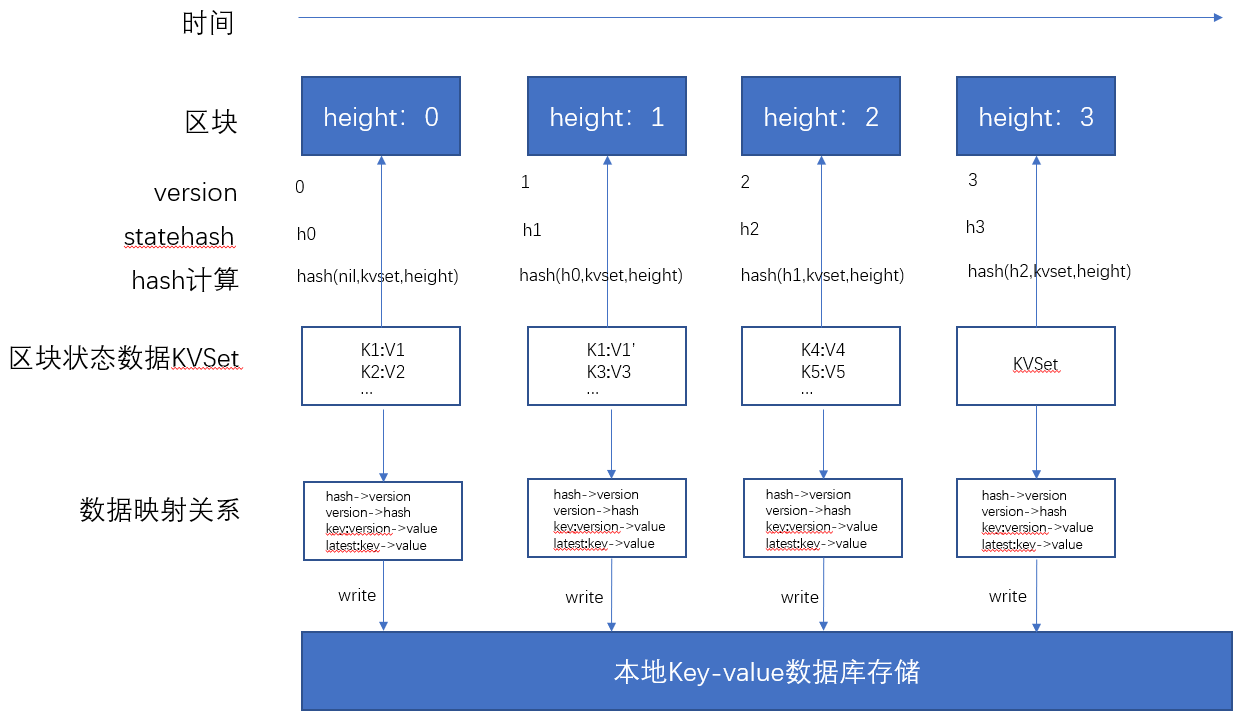


根据上图示意：

主键为Key的变量值在区块高度101、102中均发生变化，对应的在KVDB中有Key:101->value101和Key:102->value102两个历史版本数据，在当前区块104中正在写入新值Key:104->value104；

那么对于历史数据Key:101、Key:102可以有很多并发的读，对于Key:104有唯一的当前写，这些读写操作可以并发互不影响的执行，并且这些数据可以平铺存储在普通的KVDB中，读写只要根据块高作为版本号与Key构成唯一主键即可（Key:version或者Key:height），不需要构建树形结构来读取历史值和改变当前值，这对于数据读写的效率提升会有比较大的帮助。

KVMVCC的数据存储格式的思路如下：



1. Hash计算：

statehash=hash（prevstatehash，KVSet，height），包含了前一区块的状态Hash信息，本区块的状态数据KVSet信息，本区块的高度信息（也就是版本信息）。

1. 有以下对应关系会被存储到每个节点的数据库中：

hash->height(version)

height(version)->hash

key:height(version)->value

lastest:key->value

1. 数据查询：

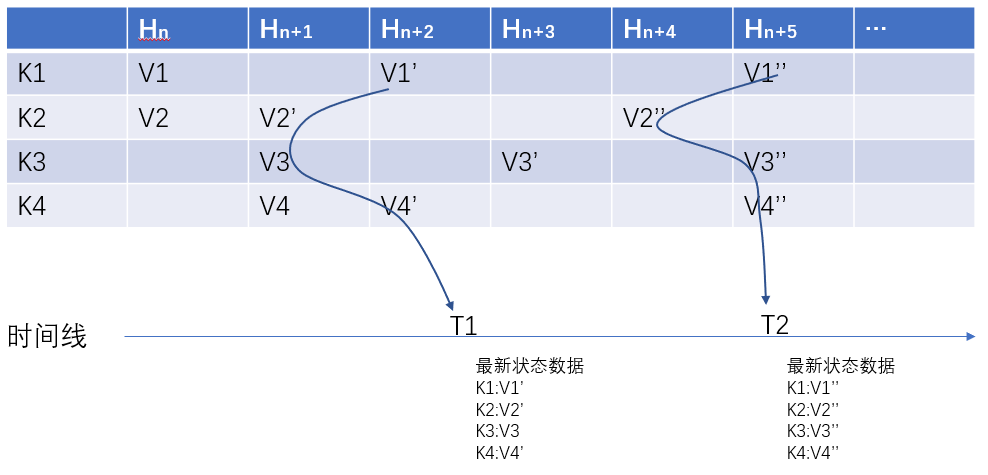
根据statehash可以查找到对应的height（version），根据height可以查找到对应高度时，具体key值对应的value值。

1. 数据验证：

对于特定高度height的KVSet，可以根据前一区块的hash值prevstatehash、KVSet、height进行Hash运算，如果hash值相符，则数据未被篡改，否则，数据被改动或者数据有误（高度有误，或者KVSet数据有误）。

1. 对于最新版本数据的维护：

特别的，当对于最新区块的key、value值进行存储时，同时保留（新增key）或者更新（已经有历史版本的key）key:latest->value的映射关系到本地key-value数据库中存储。当需要获得最新的批量数据时，可以根据latest前缀（可以自定义）来批量查询最新数据。由于通常的key-value数据库可以很好的支持前缀匹配查询，查询效率会比较高，远高于默克尔树存储结构的查询。



如上图所示：

当时间到达T1时，可以快速通过latest前缀得到最新状态数据集： {K1:V1’,K2:V2’,K3:V3,K4:V4’}

当时间到达T2时，可以快速通过latest前缀得到最新状态数据集： {K1:V1’’,K2:V2’’,K3:V3’’,K4:V4’’}

由于KVMVCC的数据存储结构，摒弃了树型结构的管理负担，写入和读取数据非常直接没有效率损耗，理论上可以接近数据读写层（LevelDB）的性能极限，从在个别性能较好的机器上的测试结果看，也可以印证了该推断。

# 四、KVMVCC与MAVL/MPT的对比总结

采用MAVL或者MPT的结构来存储状态数据的这种设计使用起来非常方便，但是也存在一定的问题：

（1）因为它保存了所有的数据，随着时间的增长，这棵树会越来越大，占用越来越多的磁盘空间。

（2）对于历史数据的读取，需要根据特定区块的State Root Hash建立起树形结构，并查找相关kv信息，并不能做到KV数据库的根据主键直接查询的快速简捷。

（3）数据变更时，需要对变更路径上的节点进行多次的数据读取，建立树型分支结构后，才能进行数据的更新；写入数据时，变化的树分支上的节点信息都要写入，消耗的操作还是比较多的。

简单来说，无论是MAVL还是MPT的数据存储格式，都是以树形结构来组织StateDB中的数据，通过StateHash来索引，对于数据状态变化的维护比较方便，缺点是随着数据量增加，效率不断下降，另外树型结构有很多不存储数据的中间节点，不能通过主键直接查找，需要按路径查找，导致查找效率慢；由于中间节点的存在，新增数据时，操作步骤也比较多，不直接。读写数据过程中对树型结构的维护是比较耗时的，会影响到处理效率。

而KVMVCC的存储格式，则通过数据版本控制，对数据存储时主键增加版本信息，可以实现数据的平铺存储，根据主键及版本快速查找数据，并且处理性能不随数据量增加而明显下降。

当然，KVMVCC相比MAVL和MPT也存在自己的不足，就是在对特定高度的状态数据进行查询及统计分析时，KVMVCC需要遍历数据库，该场景中存在比较严重的性能问题。也就是说MAVL和MPT比较方便做历史数据的SPV验证，但KVMVCC的特点决定了它不便于支持SPV验证。但并不是所有的区块链应用场景都需要做SPV验证的。

没有能适用于所有应用场景的完美方案，KVMVCC和MAVAL/MTP树型结构两者分别适用于不同的场景，开发者可根据具体业务需求进行选择。Chain33通过插件的方式提供了尽量多的可选择性，方便开发者。