从测试案例来看,blockchain的主要功能点有下面几点.

1. import.
2. GetLastBlock的功能.
3. 如果有多条区块链,可以选取其中难度最大的一条作为规范的区块链.
4. BadHashes 可以手工禁止接受一些区块的hash值.在blocks.go里面.
5. 如果新配置了BadHashes. 那么区块启动的时候会自动禁止并进入有效状态.
6. 错误的nonce会被拒绝.
7. 支持Fast importing.
8. Light vs Fast vs Full processing 在处理区块头上面的效果相等.

可以看到blockchain的主要功能是维护区块链的状态, 包括区块的验证,插入和状态查询.

名词解释:

什么是规范的区块链

因为在区块的创建过程中,可能在短时间内产生一些分叉, 在我们的数据库里面记录的其实是一颗区块树.我们会认为其中总难度最高的一条路径认为是我们的规范的区块链. 这样有很多区块虽然也能形成区块链,但是不是规范的区块链.

数据库结构:

```
区块的hash值和区块头的hash值是同样的么。所谓的区块的Hash值其实就是Header的区块值。
// key -> value
// + 代表连接

"LastHeader"  最新的区块头 HeaderChain中使用
"LastBlock"   最新的区块头 BlockChain 中使用
"LastFast"    最新的快速同步的区块头

"h"+num+"n" -> hash  用来存储规范的区块链的高度和区块头的hash值

"h" + num + hash -> header 高度+hash值 -> 区块头

"h" + num + hash + "t" -> td  高度+hash值 -> 总难度

"H" + hash -> num  区块体hash -> 高度

"b" + num + hash -> block body   高度+hash值 -> 区块体

"r" + num + hash -> block receipts  高度 + hash值 -> 区块收据

"l" + hash -> transaction/receipt lookup metadata
```

key	value	说明	插入	删除
"LastHeader"	hash	最新的区块头HeaderChain中使用	当区块被认为是当前最新的一个规范的区块链头	当有了更新的区块链头或者是分叉的兄弟区块链替代了它
"LastBlock"	hash	最新的区块头BlockChain中使用	当区块被认为是当前最新的一个规范的区块链头	当有了更新的区块链头或者是分叉的兄弟区块链替代了它
"LastFast"	hash	最新的区块头BlockChain中使用	当区块被认为是当前最新的规范的区块链头	当有了更新的区块链头或者是分叉的兄弟区块链替代了它
"h"+num+"n"	hash	用来存储规范的区块链的高度和区块头的hash值 在HeaderChain中使用	当区块在规范的区块链中	当区块不在规范的区块链中
"h" + num + hash + "t"	td	总难度	WriteBlockAndState当验证并执行完一个区块之后(不管是不是规范的)	SetHead方法会调用。这种方法只会在两种情况下被调用，1是当前区块链包含了badhashs，需要删除所有从badhashs开始的区块， 2.是当前区块的状态错误，需要Reset到genesis。
"H" + hash	num	区块的高度在HeaderChain中使用	WriteBlockAndState当验证并执行完一个区块之后	SetHead中被调用，同上

"b" + num + hash	block body	区块数据	WriteBlockAndState or InsertReceiptChain	SetHead中被删除，同上
"r" + num + hash	block receipts	区块收据	WriteBlockAndState or InsertReceiptChain	同上
"l" + txHash	{hash,num,TxIndex	交易hash可以找到区块和交易	当区块加入规范的区块链	当区块从规范的区块链移除

数据结构

```
// BlockChain represents the canonical chain given a database with a genesis
// block. The Blockchain manages chain imports, reverts, chain reorganisations.
// BlockChain 表示了一个规范的链,这个链通过一个包含了创世区块的数据库指定. BlockChain管理了链的插入,还原,重建等操作.
// Importing blocks in to the block chain happens according to the set of rules
// defined by the two stage Validator. Processing of blocks is done using the
// Processor which processes the included transaction. The validation of the state
// is done in the second part of the Validator. Failing results in aborting of
// the import.
// 插入一个区块需要通过一系列指定的规则指定的两阶段的验证器.
// 使用Processor来对区块的交易进行处理. 状态的验证是第二阶段的验证器. 错误将导致插入终止.
// The BlockChain also helps in returning blocks from **any** chain included
// in the database as well as blocks that represents the canonical chain. It's
// important to note that GetBlock can return any block and does not need to be
// included in the canonical one where as GetBlockByNumber always represents the
// canonical chain.
// 需要注意的是GetBlock可能返回任意不在当前规范区块链中的区块,
// 但是GetBlockByNumber 总是返回当前规范区块链中的区块.
type BlockChain struct {
    config *params.ChainConfig // chain & network configuration

    hc            *HeaderChain      // 只包含了区块头的区块链
    chainDb       ethdb.Database    // 底层数据库
    rmLogsFeed    event.Feed        // 下面是很多消息通知的组件
    chainFeed     event.Feed
    chainSideFeed event.Feed
```

```
    chainHeadFeed event.Feed
    logsFeed      event.Feed
    scope         event.SubscriptionScope
    genesisBlock  *types.Block      // 创世区块

    mu      sync.RWMutex // global mutex for locking chain operations
    chainmu sync.RWMutex // blockchain insertion lock
    procmu  sync.RWMutex // block processor lock

    checkpoint       int          // checkpoint counts towards the new checkpoint
    currentBlock     *types.Block // Current head of the block chain 当前的区块头
    currentFastBlock *types.Block // Current head of the fast-sync chain (may be ab
ove the block chain!) 当前的快速同步的区块头.

    stateCache   state.Database // State database to reuse between imports (contain
s state cache)
    bodyCache    *lru.Cache     // Cache for the most recent block bodies
    bodyRLPCache *lru.Cache     // Cache for the most recent block bodies in RLP en
coded format
    blockCache   *lru.Cache     // Cache for the most recent entire blocks
    futureBlocks *lru.Cache     // future blocks are blocks added for later process
ing  暂时还不能插入的区块存放位置.

    quit    chan struct{} // blockchain quit channel
    running int32         // running must be called atomically
    // procInterrupt must be atomically called
    procInterrupt int32          // interrupt signaler for block processing
    wg            sync.WaitGroup // chain processing wait group for shutting down

    engine    consensus.Engine  // 一致性引擎
    processor Processor // block processor interface  // 区块处理器接口
    validator Validator // block and state validator interface // 区块和状态验证器接
口
    vmConfig  vm.Config //虚拟机的配置

    badBlocks *lru.Cache // Bad block cache  错误区块的缓存.
}
```

构造,NewBlockChain 使用数据库里面的可用信息构造了一个初始化好的区块链. 同时初始化了以太坊默认的 验证器和处理器 (Validator and Processor)

```
// NewBlockChain returns a fully initialised block chain using information
// available in the database. It initialises the default Ethereum Validator and
// Processor.
func NewBlockChain(chainDb ethdb.Database, config *params.ChainConfig, engine conse
nsus.Engine, vmConfig vm.Config) (*BlockChain, error) {
    bodyCache, _ := lru.New(bodyCacheLimit)
```

```
    bodyRLPCache, _ := lru.New(bodyCacheLimit)
    blockCache, _ := lru.New(blockCacheLimit)
    futureBlocks, _ := lru.New(maxFutureBlocks)
    badBlocks, _ := lru.New(badBlockLimit)

    bc := &BlockChain{
        config:       config,
        chainDb:      chainDb,
        stateCache:   state.NewDatabase(chainDb),
        quit:         make(chan struct{}),
        bodyCache:    bodyCache,
        bodyRLPCache: bodyRLPCache,
        blockCache:   blockCache,
        futureBlocks: futureBlocks,
        engine:       engine,
        vmConfig:     vmConfig,
        badBlocks:    badBlocks,
    }
    bc.SetValidator(NewBlockValidator(config, bc, engine))
    bc.SetProcessor(NewStateProcessor(config, bc, engine))

    var err error
    bc.hc, err = NewHeaderChain(chainDb, config, engine, bc.getProcInterrupt)
    if err != nil {
        return nil, err
    }
    bc.genesisBlock = bc.GetBlockByNumber(0)  // 拿到创世区块
    if bc.genesisBlock == nil {
        return nil, ErrNoGenesis
    }
    if err := bc.loadLastState(); err != nil { //加载最新的状态
        return nil, err
    }
    // Check the current state of the block hashes and make sure that we do not hav
e any of the bad blocks in our chain
    // 检查当前的状态,确认我们的区块链上面没有非法的区块.
    // BadHashes是一些手工配置的区块hash值, 用来硬分叉使用的.
    for hash := range BadHashes {
        if header := bc.GetHeaderByHash(hash); header != nil {
            // get the canonical block corresponding to the offending header's numb
er
            // 获取规范的区块链上面同样高度的区块头,如果这个区块头确实是在我们的规范的区块
链上的话,我们需要回滚到这个区块头的高度 - 1
            headerByNumber := bc.GetHeaderByNumber(header.Number.Uint64())
            // make sure the headerByNumber (if present) is in our current canonica
l chain
            if headerByNumber != nil && headerByNumber.Hash() == header.Hash() {
                log.Error("Found bad hash, rewinding chain", "number", header.Numbe
```

```
r, "hash", header.ParentHash)
                bc.SetHead(header.Number.Uint64() - 1)
                log.Error("Chain rewind was successful, resuming normal operation")
            }
        }
    }
    // Take ownership of this particular state
    go bc.update()
    return bc, nil
}
```

loadLastState, 加载数据库里面的最新的我们知道的区块链状态. 这个方法假设已经获取到锁了.

```
// loadLastState loads the last known chain state from the database. This method
// assumes that the chain manager mutex is held.
func (bc *BlockChain) loadLastState() error {
    // Restore the last known head block
    // 返回我们知道的最新的区块的hash
    head := GetHeadBlockHash(bc.chainDb)
    if head == (common.Hash{}) { // 如果获取到了空.那么认为数据库已经被破坏.那么设置区块
链为创世区块.
        // Corrupt or empty database, init from scratch
        log.Warn("Empty database, resetting chain")
        return bc.Reset()
    }
    // Make sure the entire head block is available
    // 根据blockHash 来查找block
    currentBlock := bc.GetBlockByHash(head)
    if currentBlock == nil {
        // Corrupt or empty database, init from scratch
        log.Warn("Head block missing, resetting chain", "hash", head)
        return bc.Reset()
    }
    // Make sure the state associated with the block is available
    // 确认和这个区块的world state是否正确.
    if _, err := state.New(currentBlock.Root(), bc.stateCache); err != nil {
        // Dangling block without a state associated, init from scratch
        log.Warn("Head state missing, resetting chain", "number", currentBlock.Numb
er(), "hash", currentBlock.Hash())
        return bc.Reset()
    }
    // Everything seems to be fine, set as the head block
    bc.currentBlock = currentBlock

    // Restore the last known head header
    // 获取最新的区块头的hash
    currentHeader := bc.currentBlock.Header()
```

```
    if head := GetHeadHeaderHash(bc.chainDb); head != (common.Hash{}) {
        if header := bc.GetHeaderByHash(head); header != nil {
            currentHeader = header
        }
    }
    // header chain 设置为当前的区块头.
    bc.hc.SetCurrentHeader(currentHeader)

    // Restore the last known head fast block
    bc.currentFastBlock = bc.currentBlock
    if head := GetHeadFastBlockHash(bc.chainDb); head != (common.Hash{}) {
        if block := bc.GetBlockByHash(head); block != nil {
            bc.currentFastBlock = block
        }
    }

    // Issue a status log for the user 用来打印日志.
    headerTd := bc.GetTd(currentHeader.Hash(), currentHeader.Number.Uint64())
    blockTd := bc.GetTd(bc.currentBlock.Hash(), bc.currentBlock.NumberU64())
    fastTd := bc.GetTd(bc.currentFastBlock.Hash(), bc.currentFastBlock.NumberU64())

    log.Info("Loaded most recent local header", "number", currentHeader.Number, "ha
sh", currentHeader.Hash(), "td", headerTd)
    log.Info("Loaded most recent local full block", "number", bc.currentBlock.Numbe
r(), "hash", bc.currentBlock.Hash(), "td", blockTd)
    log.Info("Loaded most recent local fast block", "number", bc.currentFastBlock.N
umber(), "hash", bc.currentFastBlock.Hash(), "td", fastTd)

    return nil
}
```

goroutine update的处理非常简单. 定时处理future blocks.

```
func (bc *BlockChain) update() {
    futureTimer := time.Tick(5 * time.Second)
    for {
        select {
        case <-futureTimer:
            bc.procFutureBlocks()
        case <-bc.quit:
            return
        }
    }
}
```

Reset 方法 重置区块链.

```
// Reset purges the entire blockchain, restoring it to its genesis state.
func (bc *BlockChain) Reset() error {
    return bc.ResetWithGenesisBlock(bc.genesisBlock)
}

// ResetWithGenesisBlock purges the entire blockchain, restoring it to the
// specified genesis state.
func (bc *BlockChain) ResetWithGenesisBlock(genesis *types.Block) error {
    // Dump the entire block chain and purge the caches
    // 把整个区块链转储并清楚缓存
    if err := bc.SetHead(0); err != nil {
        return err
    }
    bc.mu.Lock()
    defer bc.mu.Unlock()

    // Prepare the genesis block and reinitialise the chain
    // 使用创世区块重新初始化区块链
    if err := bc.hc.WriteTd(genesis.Hash(), genesis.NumberU64(), genesis.Difficulty
()); err != nil {
        log.Crit("Failed to write genesis block TD", "err", err)
    }
    if err := WriteBlock(bc.chainDb, genesis); err != nil {
        log.Crit("Failed to write genesis block", "err", err)
    }
    bc.genesisBlock = genesis
    bc.insert(bc.genesisBlock)
    bc.currentBlock = bc.genesisBlock
    bc.hc.SetGenesis(bc.genesisBlock.Header())
    bc.hc.SetCurrentHeader(bc.genesisBlock.Header())
    bc.currentFastBlock = bc.genesisBlock

    return nil
}
```

SetHead将本地链回卷到新的头部。 在给定新header之上的所有内容都将被删除，新的header将被设置。 如果块体丢失（快速同步之后的非归档节点），头部可能被进一步倒回。

```
// SetHead rewinds the local chain to a new head. In the case of headers, everythin
g
// above the new head will be deleted and the new one set. In the case of blocks
// though, the head may be further rewound if block bodies are missing (non-archive
// nodes after a fast sync).
func (bc *BlockChain) SetHead(head uint64) error {
    log.Warn("Rewinding blockchain", "target", head)
```

```
    bc.mu.Lock()
    defer bc.mu.Unlock()

    // Rewind the header chain, deleting all block bodies until then
    delFn := func(hash common.Hash, num uint64) {
        DeleteBody(bc.chainDb, hash, num)
    }
    bc.hc.SetHead(head, delFn)
    currentHeader := bc.hc.CurrentHeader()

    // Clear out any stale content from the caches
    bc.bodyCache.Purge()
    bc.bodyRLPCache.Purge()
    bc.blockCache.Purge()
    bc.futureBlocks.Purge()

    // Rewind the block chain, ensuring we don't end up with a stateless head block
    if bc.currentBlock != nil && currentHeader.Number.Uint64() < bc.currentBlock.NumberU64() {
        bc.currentBlock = bc.GetBlock(currentHeader.Hash(), currentHeader.Number.Uint64())
    }
    if bc.currentBlock != nil {
        if _, err := state.New(bc.currentBlock.Root(), bc.stateCache); err != nil {
            // Rewound state missing, rolled back to before pivot, reset to genesis
            bc.currentBlock = nil
        }
    }
    // Rewind the fast block in a simpleton way to the target head
    if bc.currentFastBlock != nil && currentHeader.Number.Uint64() < bc.currentFastBlock.NumberU64() {
        bc.currentFastBlock = bc.GetBlock(currentHeader.Hash(), currentHeader.Number.Uint64())
    }
    // If either blocks reached nil, reset to the genesis state
    if bc.currentBlock == nil {
        bc.currentBlock = bc.genesisBlock
    }
    if bc.currentFastBlock == nil {
        bc.currentFastBlock = bc.genesisBlock
    }
    if err := WriteHeadBlockHash(bc.chainDb, bc.currentBlock.Hash()); err != nil {
        log.Crit("Failed to reset head full block", "err", err)
    }
    if err := WriteHeadFastBlockHash(bc.chainDb, bc.currentFastBlock.Hash()); err != nil {
        log.Crit("Failed to reset head fast block", "err", err)
    }
```

```
    return bc.loadLastState()
}
```

InsertChain,插入区块链, 插入区块链尝试把给定的区块插入到规范的链条,或者是创建一个分叉. 如果发生错误,那么会返回错误发生时候的index和具体的错误信息.

```
// InsertChain attempts to insert the given batch of blocks in to the canonical
// chain or, otherwise, create a fork. If an error is returned it will return
// the index number of the failing block as well an error describing what went
// wrong.
//
// After insertion is done, all accumulated events will be fired.
// 在插入完成之后,所有累计的事件将被触发.
func (bc *BlockChain) InsertChain(chain types.Blocks) (int, error) {
    n, events, logs, err := bc.insertChain(chain)
    bc.PostChainEvents(events, logs)
    return n, err
}
```

insertChain方法会执行区块链插入,并收集事件信息. 因为需要使用defer来处理解锁,所以把这个方法作为一个单独的方法.

```
// insertChain will execute the actual chain insertion and event aggregation. The
// only reason this method exists as a separate one is to make locking cleaner
// with deferred statements.
func (bc *BlockChain) insertChain(chain types.Blocks) (int, []interface{}, []*types
.Log, error) {
    // Do a sanity check that the provided chain is actually ordered and linked
    // 做一个健全的检查, 提供的链实际上是有序的和相互链接的
    for i := 1; i < len(chain); i++ {
        if chain[i].NumberU64() != chain[i-1].NumberU64()+1 || chain[i].ParentHash(
) != chain[i-1].Hash() {
            // Chain broke ancestry, log a messge (programming error) and skip inse
rtion
            log.Error("Non contiguous block insert", "number", chain[i].Number(), "
hash", chain[i].Hash(),
                "parent", chain[i].ParentHash(), "prevnumber", chain[i-1].Number(),
 "prevhash", chain[i-1].Hash())

            return 0, nil, nil, fmt.Errorf("non contiguous insert: item %d is #%d [
%x…], item %d is #%d [%x…] (parent [%x…])", i-1, chain[i-1].NumberU64(),
                chain[i-1].Hash().Bytes()[:4], i, chain[i].NumberU64(), chain[i].Ha
sh().Bytes()[:4], chain[i].ParentHash().Bytes()[:4])
        }
    }
```

```
// Pre-checks passed, start the full block imports
bc.wg.Add(1)
defer bc.wg.Done()

bc.chainmu.Lock()
defer bc.chainmu.Unlock()

// A queued approach to delivering events. This is generally
// faster than direct delivery and requires much less mutex
// acquiring.
var (
    stats         = insertStats{startTime: mclock.Now()}
    events        = make([]interface{}, 0, len(chain))
    lastCanon     *types.Block
    coalescedLogs []*types.Log
)
// Start the parallel header verifier
headers := make([]*types.Header, len(chain))
seals := make([]bool, len(chain))

for i, block := range chain {
    headers[i] = block.Header()
    seals[i] = true
}
// 调用一致性引擎来验证区块头是有效的.
abort, results := bc.engine.VerifyHeaders(bc, headers, seals)
defer close(abort)

// Iterate over the blocks and insert when the verifier permits
for i, block := range chain {
    // If the chain is terminating, stop processing blocks
    if atomic.LoadInt32(&bc.procInterrupt) == 1 {
        log.Debug("Premature abort during blocks processing")
        break
    }
    // If the header is a banned one, straight out abort
    // 如果区块头被禁止了.
    if BadHashes[block.Hash()] {
        bc.reportBlock(block, nil, ErrBlacklistedHash)
        return i, events, coalescedLogs, ErrBlacklistedHash
    }
    // Wait for the block's verification to complete
    bstart := time.Now()

    err := <-results
    if err == nil { // 如果没有错误. 验证body
        err = bc.Validator().ValidateBody(block)
    }
```

```
        if err != nil {
            if err == ErrKnownBlock { // 如果区块已经插入, 直接继续
                stats.ignored++
                continue
            }

            if err == consensus.ErrFutureBlock {
                // Allow up to MaxFuture second in the future blocks. If this limit
                // is exceeded the chain is discarded and processed at a later time
                // if given.
                // 如果是未来的区块, 而且区块的时间距离现在不是很久远. 那么存放起来.
                max := big.NewInt(time.Now().Unix() + maxTimeFutureBlocks)
                if block.Time().Cmp(max) > 0 {
                    return i, events, coalescedLogs, fmt.Errorf("future block: %v > %v", block.Time(), max)
                }
                bc.futureBlocks.Add(block.Hash(), block)
                stats.queued++
                continue
            }

            if err == consensus.ErrUnknownAncestor && bc.futureBlocks.Contains(block.ParentHash()) { 如果区块没有找到祖先 而在future blocks 包含了这个区块的祖先,那么也存放在future
                bc.futureBlocks.Add(block.Hash(), block)
                stats.queued++
                continue
            }

            bc.reportBlock(block, nil, err)
            return i, events, coalescedLogs, err
        }
        // Create a new statedb using the parent block and report an
        // error if it fails.
        var parent *types.Block
        if i == 0 {
            parent = bc.GetBlock(block.ParentHash(), block.NumberU64()-1)
        } else {
            parent = chain[i-1]
        }
        state, err := state.New(parent.Root(), bc.stateCache)
        if err != nil {
            return i, events, coalescedLogs, err
        }
        // Process block using the parent state as reference point.
        // 处理区块,生成交易,收据,日志等信息.
        // 实际上调用了state_processor.go 里面的 Process方法.
        receipts, logs, usedGas, err := bc.processor.Process(block, state, bc.vmCon
```

```
fig)
        if err != nil {
            bc.reportBlock(block, receipts, err)
            return i, events, coalescedLogs, err
        }
        // Validate the state using the default validator
        // 二次验证,验证状态是否合法
        err = bc.Validator().ValidateState(block, parent, state, receipts, usedGas)
        if err != nil {
            bc.reportBlock(block, receipts, err)
            return i, events, coalescedLogs, err
        }
        // Write the block to the chain and get the status
        // 写入区块和状态.
        status, err := bc.WriteBlockAndState(block, receipts, state)
        if err != nil {
            return i, events, coalescedLogs, err
        }
        switch status {
        case CanonStatTy:  // 插入了新的区块.
            log.Debug("Inserted new block", "number", block.Number(), "hash", block
.Hash(), "uncles", len(block.Uncles()),
                "txs", len(block.Transactions()), "gas", block.GasUsed(), "elapsed"
, common.PrettyDuration(time.Since(bstart)))

            coalescedLogs = append(coalescedLogs, logs...)
            blockInsertTimer.UpdateSince(bstart)
            events = append(events, ChainEvent{block, block.Hash(), logs})
            lastCanon = block

        case SideStatTy:  // 插入了一个forked 区块
            log.Debug("Inserted forked block", "number", block.Number(), "hash", bl
ock.Hash(), "diff", block.Difficulty(), "elapsed",
                common.PrettyDuration(time.Since(bstart)), "txs", len(block.Transac
tions()), "gas", block.GasUsed(), "uncles", len(block.Uncles()))

            blockInsertTimer.UpdateSince(bstart)
            events = append(events, ChainSideEvent{block})
        }
        stats.processed++
        stats.usedGas += usedGas.Uint64()
        stats.report(chain, i)
    }
    // Append a single chain head event if we've progressed the chain
    // 如果我们生成了一个新的区块头, 而且最新的区块头等于lastCanon
    // 那么我们公布一个新的 ChainHeadEvent
    if lastCanon != nil && bc.LastBlockHash() == lastCanon.Hash() {
        events = append(events, ChainHeadEvent{lastCanon})
```

```
    }
    return 0, events, coalescedLogs, nil
}
```

WriteBlockAndState,把区块写入区块链.

```
// WriteBlock writes the block to the chain.
func (bc *BlockChain) WriteBlockAndState(block *types.Block, receipts []*types.Rece
ipt, state *state.StateDB) (status WriteStatus, err error) {
    bc.wg.Add(1)
    defer bc.wg.Done()

    // Calculate the total difficulty of the block
    // 计算待插入的区块的总难度
    ptd := bc.GetTd(block.ParentHash(), block.NumberU64()-1)
    if ptd == nil {
        return NonStatTy, consensus.ErrUnknownAncestor
    }
    // Make sure no inconsistent state is leaked during insertion
    // 确保在插入过程中没有不一致的状态泄漏
    bc.mu.Lock()
    defer bc.mu.Unlock()
    // 计算当前区块的区块链的总难度.
    localTd := bc.GetTd(bc.currentBlock.Hash(), bc.currentBlock.NumberU64())
    // 计算新的区块链的总难度
    externTd := new(big.Int).Add(block.Difficulty(), ptd)

    // Irrelevant of the canonical status, write the block itself to the database
    // 和规范区块没有关系的状态, 写入数据库.  写入区块的hash 高度和对应的总难度.
    if err := bc.hc.WriteTd(block.Hash(), block.NumberU64(), externTd); err != nil
{
        return NonStatTy, err
    }
    // Write other block data using a batch.
    batch := bc.chainDb.NewBatch()
    if err := WriteBlock(batch, block); err != nil { // 写入区块
        return NonStatTy, err
    }
    if _, err := state.CommitTo(batch, bc.config.IsEIP158(block.Number())); err !=
nil {  //Commit
        return NonStatTy, err
    }
    if err := WriteBlockReceipts(batch, block.Hash(), block.NumberU64(), receipts);
 err != nil { // 写入区块收据
        return NonStatTy, err
    }
```

```
    // If the total difficulty is higher than our known, add it to the canonical ch
ain
    // Second clause in the if statement reduces the vulnerability to selfish minin
g.
    // Please refer to http://www.cs.cornell.edu/~ie53/publications/btcProcFC.pdf
    // 如果新的区块的总难度高于我们当前的区块，把这个区块设置为规范的区块.
    // 第二个表达式 ((externTd.Cmp(localTd) == 0 && mrand.Float64() < 0.5))
    // 是为了减少自私挖矿的可能性.
    if externTd.Cmp(localTd) > 0 || (externTd.Cmp(localTd) == 0 && mrand.Float64()
< 0.5) {
        // Reorganise the chain if the parent is not the head block
        // 如果这个区块的父区块不是当前的区块，说明存在一个分叉.需要调用reorg重新组织区块
链.
        if block.ParentHash() != bc.currentBlock.Hash() {
            if err := bc.reorg(bc.currentBlock, block); err != nil {
                return NonStatTy, err
            }
        }
        // Write the positional metadata for transaction and receipt lookups
        // "l" + txHash -> {blockHash,blockNum,txIndex}
        // 根据交易的hash值来找到对应的区块以及对应的交易。
        if err := WriteTxLookupEntries(batch, block); err != nil {
            return NonStatTy, err
        }
        // Write hash preimages
        // hash(Keccak-256) -> 对应的数据 这个功能是用来测试的。如果开启了dev模式,
        // 或者是 vmdebug参数,  如果执行 SHA3 指令就会添加Preimage
        if err := WritePreimages(bc.chainDb, block.NumberU64(), state.Preimages());
 err != nil {
            return NonStatTy, err
        }
        status = CanonStatTy
    } else {
        status = SideStatTy
    }
    if err := batch.Write(); err != nil {
        return NonStatTy, err
    }

    // Set new head.
    if status == CanonStatTy {
        bc.insert(block)
    }
    bc.futureBlocks.Remove(block.Hash())
    return status, nil
}
```

reorgs方法是在新的链的总难度大于本地链的总难度的情况下，需要用新的区块链来替换本地的区块链为规范链。

```
// reorgs takes two blocks, an old chain and a new chain and will reconstruct the b
locks and inserts them
// to be part of the new canonical chain and accumulates potential missing transact
ions and post an
// event about them
// reorgs 接受两个区块作为参数，一个是老的区块链，一个新的区块链，这个方法会把他们插入
// 以便重新构建出一条规范的区块链。 同时会累计潜在会丢失的交易并把它们作为事件发布出去。
func (bc *BlockChain) reorg(oldBlock, newBlock *types.Block) error {
    var (
        newChain    types.Blocks
        oldChain    types.Blocks
        commonBlock *types.Block
        deletedTxs  types.Transactions
        deletedLogs []*types.Log
        // collectLogs collects the logs that were generated during the
        // processing of the block that corresponds with the given hash.
        // These logs are later announced as deleted.
        // collectLogs 会收集我们已经生成的日志信息，这些日志稍后会被声明删除(实际上在数
据库中并没有被删除)。
        collectLogs = func(h common.Hash) {
            // Coalesce logs and set 'Removed'.
            receipts := GetBlockReceipts(bc.chainDb, h, bc.hc.GetBlockNumber(h))
            for _, receipt := range receipts {
                for _, log := range receipt.Logs {
                    del := *log
                    del.Removed = true
                    deletedLogs = append(deletedLogs, &del)
                }
            }
        }
    )

    // first reduce whoever is higher bound
    if oldBlock.NumberU64() > newBlock.NumberU64() {
        // reduce old chain 如果老的链比新的链高。那么需要减少老的链，让它和新链一样高
        for ; oldBlock != nil && oldBlock.NumberU64() != newBlock.NumberU64(); oldB
lock = bc.GetBlock(oldBlock.ParentHash(), oldBlock.NumberU64()-1) {
            oldChain = append(oldChain, oldBlock)
            deletedTxs = append(deletedTxs, oldBlock.Transactions()...)

            collectLogs(oldBlock.Hash())
        }
    } else {
        // reduce new chain and append new chain blocks for inserting later on
```

```
        // 如果新链比老链要高，那么减少新链。
        for ; newBlock != nil && newBlock.NumberU64() != oldBlock.NumberU64(); newB
lock = bc.GetBlock(newBlock.ParentHash(), newBlock.NumberU64()-1) {
            newChain = append(newChain, newBlock)
        }
    }
    if oldBlock == nil {
        return fmt.Errorf("Invalid old chain")
    }
    if newBlock == nil {
        return fmt.Errorf("Invalid new chain")
    }

    for { //这个for循环里面需要找到共同的祖先。
        if oldBlock.Hash() == newBlock.Hash() {
            commonBlock = oldBlock
            break
        }

        oldChain = append(oldChain, oldBlock)
        newChain = append(newChain, newBlock)
        deletedTxs = append(deletedTxs, oldBlock.Transactions()...)
        collectLogs(oldBlock.Hash())

        oldBlock, newBlock = bc.GetBlock(oldBlock.ParentHash(), oldBlock.NumberU64(
)-1), bc.GetBlock(newBlock.ParentHash(), newBlock.NumberU64()-1)
        if oldBlock == nil {
            return fmt.Errorf("Invalid old chain")
        }
        if newBlock == nil {
            return fmt.Errorf("Invalid new chain")
        }
    }
    // Ensure the user sees large reorgs
    if len(oldChain) > 0 && len(newChain) > 0 {
        logFn := log.Debug
        if len(oldChain) > 63 {
            logFn = log.Warn
        }
        logFn("Chain split detected", "number", commonBlock.Number(), "hash", commo
nBlock.Hash(),
            "drop", len(oldChain), "dropfrom", oldChain[0].Hash(), "add", len(newCh
ain), "addfrom", newChain[0].Hash())
    } else {
        log.Error("Impossible reorg, please file an issue", "oldnum", oldBlock.Numb
er(), "oldhash", oldBlock.Hash(), "newnum", newBlock.Number(), "newhash", newBlock.
Hash())
    }
```

```
    var addedTxs types.Transactions
    // insert blocks. Order does not matter. Last block will be written in ImportCh
ain itself which creates the new head properly
    for _, block := range newChain {
        // insert the block in the canonical way, re-writing history
        // 插入区块 更新记录规范区块链的key
        bc.insert(block)
        // write lookup entries for hash based transaction/receipt searches
        // 写入交易的查询信息。
        if err := WriteTxLookupEntries(bc.chainDb, block); err != nil {
            return err
        }
        addedTxs = append(addedTxs, block.Transactions()...)
    }

    // calculate the difference between deleted and added transactions
    diff := types.TxDifference(deletedTxs, addedTxs)
    // When transactions get deleted from the database that means the
    // receipts that were created in the fork must also be deleted
    // 删除那些需要删除的交易查询信息。
    // 这里并没有删除那些需要删除的区块，区块头，收据等信息。
    for _, tx := range diff {
        DeleteTxLookupEntry(bc.chainDb, tx.Hash())
    }
    if len(deletedLogs) > 0 { // 发送消息通知
        go bc.rmLogsFeed.Send(RemovedLogsEvent{deletedLogs})
    }
    if len(oldChain) > 0 {
        go func() {
            for _, block := range oldChain { // 发送消息通知。
                bc.chainSideFeed.Send(ChainSideEvent{Block: block})
            }
        }()
    }

    return nil
}
```