Casper

Casper 简介与概览

Casper 是知名开源区块链项目以太坊 (Ethereum) [1] 的共识算法，是以太坊转型为全面 PoS (Proof-of-Stake) 的基础理论支持和实现，同时也是以太坊 2.0 计划 [2] 的一部分。

起源

早在比特币 (Bitcoin) 诞生不久的 2011 年，就有对 PoW (Proof-of-Work) 挖矿出块的讨论，持反对观点的人认为这是一种无意义、浪费资源的行为。有人提出 PoS 权益证明，以程序算法的形式出块并分配收益，而不是通过 CPU、GPU 等硬件计算出某种符合条件的值来出块。

在 2016 年 9 月的以太坊 DevCon 2 上，以太坊创始人 Vitalik 分享了他的最新研究成果《以太坊紫皮书》[3]。在紫皮书中，Vitalik 提出了以太坊 2.0 的一系列目标，其中就包括以太坊的 PoS 方案 Casper (另一个主要功能是通过分片提升系统的可扩展性)。

在 2017 年的 DevCon3 上，Vitalik 和以太坊基金会的其他小伙伴也有大量的议题是关于 Casper 的最新进展 [4]。

原理与目标

前面说到，PoW 机制需要消耗大量的电力去维持区块链系统出块，根据 2017 年底的数据统计 [5]，比特币挖矿目前占用全球 0.13% 的用电量，超过 159 个国家的年均用电量，并且在过去 1 个月内涨幅超过 29.98%。按照目前的增长速度 。2020 年 2 月比特币挖矿将消耗全球所有的电量。

PoW 也是现在绝大部分区块链系统的基础机制。在 PoW 模式下的，用户消耗真实的电力用于产生区块获得分红。而在 PoS 模式下，用户使用区块链系统本身的虚拟代币进行挖矿，参与 PoS 的代币数量相当于 PoW 算力，达到了同样的出块效果却不用消耗真实的电力。（有点像股权分红）

以太坊的紫皮书中有如下几个 Casper 最终形态的设计目标：

1. (Efficiency via proof of stake) 通过 PoS 提升效率：共识机制不需要挖矿，降低电力浪费，并且可以满足大量可持续的以太币发行需求。

2. (Fast block time) 快速的出块率：在不影响安全的前提下，出块速度达到最大。
3. (Finality) 不可修改性：一旦出块并发布，一定时间之后这个块将会“finalized”（编程语言中的 不可修改 关键词）。
4. (Scalability) 可扩展性：区块链不必在全部节点行运行，相对的，所有的节点只需要保持区块的一部分数据。这样既可以提高单个节点的处理能力，也能提高整体吞吐率。
5. (Decentralization) 去中心化：这个系统不需要依赖任何普通用户无法部署的“超级节点”。
6. (Inter-shard communication) 跨分片通信：应用在不同的分片上能够互相通信。特别的，一个应用能够跨越多个分片存在。
7. (Recovery from Connectivity Catastrophe) 从连接中断中恢复：这个系统需要能够在一半以上的节点突然掉线的情况下恢复并运行。
8. (Censorship resistance) 抗审查：这个协议能够防止大部分的见证人联合起来提供的虚假交易信息。

简单地说，Casper 设计为一个投注对赌模型。每个参与 PoS 的用户投入以太币进入资金池，每次下注猜测最终加入到主链的块，猜中即可获得分红，猜错会扣除一部分手续费。长期下来，每个用户的收益与其投入的代币数量成正比。

总统选举例子

为了便于读者理解，我们以美国总统大选为例。（以下事例纯属虚构）

这里是 2016 年美国大选，每个公民仅有一票，只能选择投给特朗普和希拉里其中一人，假定投票有如下规则：

- 投票对象与最后当选总统人相符，投票者会获得美元奖励。
- 投票持续 10 天，投票人在结束前都可以更改自己的选票。
- 你每天可以询问身边的 20 个公民的投票意向，并且假定他们的回答是诚实的。
- 所有公民都是驱利的，即会选择回报期望值最高的策略。

根据以上规则，我们思考一下。如果你作为公民，准备投票给希拉里，但是第一天询问了身边 20 个人后发现 18 个人都是投票给特朗普的，为了获得最后的美元奖励，你会更改投票对象为特朗普吗？如果每个居民都如此考虑，那么会不会在几天之后快速出现所有人都选择特朗普的情况呢？

在这个模型下，假设第一天所有人，有 51%选择特朗普，49%选择希拉里，经过足够的天数迭代之后，所有的人都会因为利益而导向特朗普，这便是迭代收敛。

以太坊基金会的成员们，已经通过程序的方式在验证这个模型在各种情况下的可靠性[6]。

当然这个例子只是一个简化的模型，真实的 Casper PoS 模型需要考虑各种安全性、可用性和性能问题。

Casper 数据结构与投注出块

上一章讲到了 Casper 的基本情况，这一章讲一讲 Casper 的基础数据结构和投注流程。

为此，我们首先创建一个最小的 PoS 算法能够满足第一章设计目标中的第一点（PoS）和第二点（快速出块）。

如果把 PoS 比喻为一个大赌场，那么每个参与 PoS 的验证人就是赌徒，赌徒当然需要将代币作为赌资进行“投注”（deposit）才能进行参与。

验证人池

我们所接触的最重要的数据结构是验证人池，可以将验证人池理解为一个保存了所有参与 PoS 的验证人的集合，在使用 Go 重写 Casper 后，一个验证人可以用如下 Go 代码表示[3]。

```
type Validator struct {      // 投入的代币数量
      Deposit *big.Rat      // validator 加入的 dynasty
      Dynasty_start uint64
      // validator 退出的 dynasty
      Dynasty_end uint64
      // validator 的签名地址
      Addr string
      // 收回代币的地址
      Withdrawal_addr string
      // 该 validator 提交的上一个时间戳
      Prev_commit_epoch uint64}
```

包括了每个验证人参与 PoS 的代币数量，加入 PoS 的 dynasty（可以理解为周期版本号，验证人池变更的最小时间单位），退出的 dynasty，验证人的签名地址，收回代币的地址，以及该验证人参与 PoS 的前一个时间戳。

验证人池则由若干 Validator 组成，可以用集合（Set）表示。

投注

现存在一个“Casper 合约”，这个合约会保存并跟踪 “验证人池” (validator set)，该 Casper 合约被包含在创世块 (genesis block) 中并且没有权限要求 (公开的)，调用这个 Casper 合约是验证一个区块头部的第一步。因此，初始化状态的验证人池被定义在创世块并且能够被如下函数 (算子) 修改 (后文为了避免歧义，我们称 Casper 算法的重要操作函数为 Casper 算子) ：

deposit(bytes validation_code, bytes32 randao, address withdrawal_address)

该算子接受的参数如下：

- validation_code: 一串以太虚拟机 (EVM) 代码，用于验证区块和被其签名的一致性消息，可以看做是公钥。
- randao: 32 个 byte 的哈希值用于选取领导 (leader selection，详见下文) 。
- withdrawal_address: 用于收回投注资金的地址。

准确的说，validation_code 实现了一个智能合约，输入是 block header hash 和一个签名，签名有效返回 1，否则返回 0。这个机制允许每个验证人使用任意的签名方式，例如多重签名，或者 Lamport 签名抵抗量子计算机攻击。这段智能合约会使用 CALL_BLACKBOX 虚拟机操作码保证黑盒运行，不会被外部状态影响。

randao 是一个经常了连续 sha3 计算的随机值，可以认为是一个真正的纯随机 32 位字符串。每个验证人的 randao 都被保存在 Casper 合约中。

如果全部的参数都被接收，验证人池会在下下个时间戳增加一个验证人。例如，如果 deposit 在 n 时刻被调用，那么验证人池会在 n+2 时刻添加验证人，validation_code 的哈希值被用作验证人 ID (又称为 vchash)，每个验证人都会有一个独立的 ID。

出块

Casper 合约还有三个相关算子：

startWithdrawal(bytes32 vchash, bytes sig): 开始执行收回流程，需要传入验证人 ID 和能通过 validation_code 的私钥。
finishWithdrawal(bytes32 vchash): 收回代币到 deposit 时指定的收回地址，包括出块奖励和作弊惩罚。
getValidator(uint256 skips): 返回第 skips 的 validator 的 validation code，该验证人被指定创建下一个区块。如果一个 validator 无法创建下一个区块，则选择排在后面的一个 validator。

选择出块人是通过完全的伪随机算法选择的，随机种子是一个全局的 globalRandao。

到目前为止，一个区块必须包含如下的额外字段：

	Seconds after block is publ			
	0	**BLOCK_TIME**	**BLOCK_TIME + SKIP_DELAY * 1**	**BLOCK_TIM SKIP_DELA**
Validators to accept blocks from		getValidator(0)	getValidator(0) getValidator(1)	getValidato getValidato getValidato

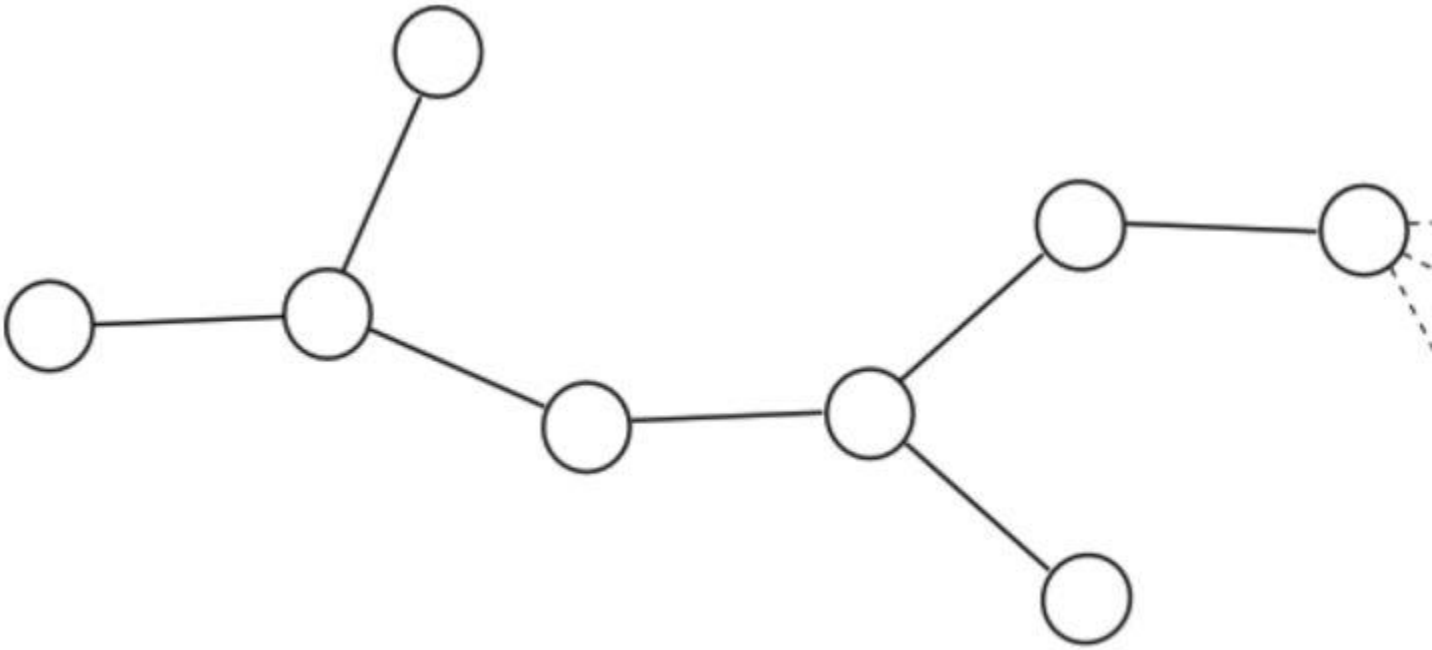


Figure 1: Demonstrating Table 1 graphi

创建具体区块所需要的最短时间可以简单计算为：GENESIS_TIME + BLOCK_TIME * <区块高度> + SKIP_TIME * <创世区块后所有的验证者 skip 数>。

在实际过程中，这意味着，一旦发布了某个区块，那么下一个区块的 0-skip 验证者会在 BLOCK_TIME 秒之后发布，同理，1-skip 验证者则在 BLOCK_TIME + SKIP_TIME 秒之后发布，以此类推。

如果一个验证者发布一个区块太早，其他的验证者会忽视该区块，直到在规定的时间之后，才会处理该区块（较小的 BLOCK_TIME 和较长的 SKIP_TIME 之间的不对称性，可以确保在正常情况下，区块的平均保留时间可以非常短，而在网络延迟更长的情况下，也可以保证网络的安全性)。

如果一个验证者创建了一个包括在链内的区块，他们得到的区块奖励相当于此周期内活动验证者集合内以太的总量乘上 REWARD_COEFFICIENT * BLOCK_TIME。因此，如果验证者总是正确的履行职责，REWARD_COEFFICIENT 本质上成为验证者的“预期每秒收益率”，乘上 3200 万得到近似的年化收益率。如果一个验证者创建了一个不包括在链内的区块，之后，在将来的任意时间（直到验证者调用 withdraw 函数为止）该区块标头可以作为一个“dunkle” 通过 Casper 合约的 includeDunkle 函数被包在链内；这使得验证者损失了相当于区块奖励的钱数（以及向包括 dunkle 在内的当事方提供一小部分罚款作为激励）。因此，验证者应当在确定该区块在链内的可能性超过 50%，才实际创建该区块。验证者的累计保证金，包括奖励和罚款，存储在 Casper 合约内。

“dunkle”机制的目的是为了解决权益证明中的“零赌注”的问题，其中，如果没有罚款，只有奖励，那么验证者将被物质激励，试图在每个可能的链上创建区块。在工作量证明场景下，创建区块需要成本，并且只有在“主链”上创建区块才有利可图。Dunkle 机制试图复制工作量证明中的经济理论，针对创建非主链上区块进行人工罚款，来代替工作量证明中电费用的“自然罚款”。

假设一个固定大小的验证者集合，我们可以很容易的定义分叉选择规则：计算区块数，最长链胜出。假设验证者集合可以变大和缩小，但是，该规则就不太适用了，因为作为少数支持的分叉的速度一个时期以后将像多数支持的分叉一样。因此，我们可以用计算的区块数代替定义的分叉选择规则，给每个区块一个相当于区块奖励的权重。因为区块奖励与积极验证的以太总量成正比，这确保了更积极验证以太的链得分增长速度更快。

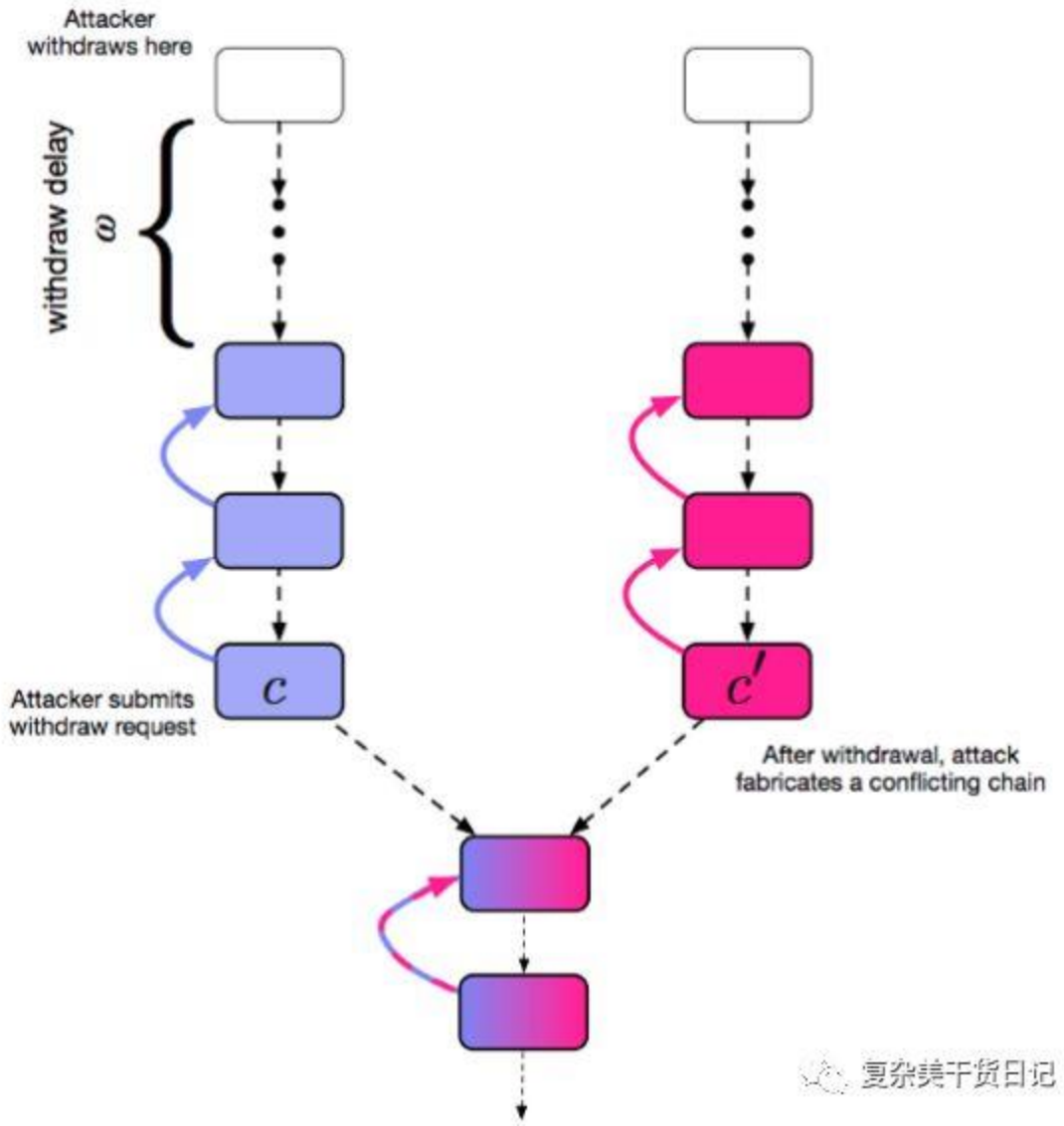
我们可以看出，这条规则可以用另一种方式很方便的理解：基于价值损失的分叉选择模型。该原则是：我们选择验证者赌最多价值的链，就是说，验证者承认除了上述的链外，所有其他链都会损失大量的资金。我们可以等同认为，这条链是验证者失去资金最少的链。在这样一个简单的模式中，很容易看出这如何简单的对应着区块权重为区块奖励的最长链。该算法是尽管简单，但用于 PoS 的实现来说也足够高效。

取钱

```
// Check if we can withdrawif this.dynasty <
this.validators[validator_index].Dynasty_end + 1 {
  return errors.New("validator not gone, can not withdraw")
}
end_epoch :=
this.dynasty_start_epoch[this.validators[validator_index].Dynasty_end + 1]if
this.current_epoch < end_epoch + Epoch(this.withdrawal_delay) {      return
errors.New("still within delay, can not withdraw")}
```

取钱比较简单，先进行 check，首先当前 dynasty 必须是在该验证人的 end_dynasty+1 之后，通过 dynasty 计算 end_epoch，然后检查当前 epoch 必须是要在 end_epoch 加上取钱所需要的延时之后。上述代码就是进行检查的过程。

那么取钱为什么要有那么长的延时呢，因为如果不这样，那么大部分验证者解除绑定，他们就可以恶意地创造包含之前的验证者集的第二条链。所以解除绑定保证金后必须要经过一个“解冻”期，才可收回。



上述检查成功以后，将该验证人的投注乘上一个比例因子获得取钱的金额，然后将该金额发送到对应验证人的取钱地址，最后在验证人池中删除验证人。

达成共识

对于一个分布式容错系统而言，核心问题就是所有进程最终达到一个相同的阶段，即“共识”。现实生活中的去中心化系统可能会出现任意的错误，共识就需要考虑很多异常情况。现在我们先考虑简单的情况，在没有 failure 的情况下，casper 是如何达成共识的。

接下来讨论一下达成共识的过程，也就是一个区块的最终确定，主要是通过两轮的投票过程来确定一个区块。一次是 PREPARE，一次是 COMMIT，对应一个块的 justified 和

finalized，justified("审判")可以认为一个块被最终确定的前置条件，finalized 即为块的最终确定。

共识信息也是一个重要的数据结构，用 Go 语言可以表示为

```
type Consensus_message struct {      // 这个 hash 现在有多少 prepares (hash of message hash + view source) 在目前的 dynasty
      Cur_dyn_prepares map[uint32]big.Rat      // Bitmap of which validator IDs have already prepared
      Prepare_bitmap: num256[num][bytes32]      // 之前的 dynasty
      Prev_dyn_prepares map[uint32]big.Rat      // 是否是被确认过的 hash
      Ancestry_hash_justified map[uint32]bool
      // hash 有多少 commits
      Cur_dyn_commits map[uint32]big.Rat      // 之前的 dynasty
      Prev_dyn_commits map[uint32]big.Rat
}
```

主要是当前 dynasty 和前一个 dynasty 中 PREPARE 和 COMMIT 的数量，以及父区块是否被 justified 的信息。

PREPARE 消息解析出来是以下一些字段

[validator_index, epoch, ancestry_hash, source_epoch, source_ancestry_hash, sig] 记录验证人的序号，当前块和父块的时间戳和哈希值，签名等信息。

首先需要验证签名以及检查该 PREPARE 消息是第一次发出。然后在对应 epoch 上的共识信息中 PREPARE 字段中加上该验证人的投注金额，如果 PREPARE 的数目大于等于总投注金额的 2/3 那么该块就认为是 justified，即被审判。

COMMIT 消息与 PREPARE 消息类似，之前共识信息数据结构记录了父区块是否被 justified 的信息，这时就派上了用场。确认以后同样地在共识信息中 COMMIT 子段中加上该验证人的投注金额，如果 COMMIT 的数目大于等于总投注金额的 2/3，那么父区块就被认为是 finalized，即最终确定。

所以简单地总结一下，一个区块的确定经历两个过程

- 大多数(超过 2/3)验证人在周期 1 的时候给区块 1 进行了投票，区块 1 被审判(justified)。
- 大多数(超过 2/3)验证人在周期二的时候给区块 1 的子区块叫区块 2 进行了投票，所以在周期二区块 1 被确定(finalized)。

关于共识的达成想法还是很清晰的，通过大多数验证人对同一个区块的投票来最终确定一个块。然而，问题在于不像工作量证明中电力资源的“自然惩罚”，验证人的行为主观性更强，共识的最终形成需要严格定义惩罚方案，而这是 casper 共识算法的最重要部分，也决定了算法能否在真实系统中 work.

这一章主要讲的是 Casper 的惩罚机制。

在有了前几章提到的数据结构和 Casper 算子之后，PoS 已经基本成型。然而，缺乏惩罚措施会导致大量参与 PoS 的验证人作弊，从而使整个系统瘫痪。惩罚（Slash）的目的在于限制验证人作弊，作弊的代价将远远高于诚实工作，因此经济利益会驱使绝大部分验证人诚实工作。

我们的意图是使得 51%的攻击非常的昂贵，并且即使大部分的验证人一起协力也不能将已经确定的区块回滚，除非他们愿意承受极大的经济损失—这个经济损失如此的大，以至于当攻击成功了，这会提高底层加密货币的价格，因为市场会对总硬币供应的减少做出强烈的反应，而不是启用紧急硬分叉来纠正攻击。

惩罚条件需要满足的情况

惩罚条件需要满足下面两个情况：

1. 责任安全：如果两个冲突的哈希值被最终确定，那么至少能够被证实，存在至少三分之一的验证节点违反了惩罚条件
2. 合理的活跃性：必须存在一组至少有三分之二的验证节点发送的一组消息，并且必须在没有违反惩罚条件的情况下能确定一些新的哈希区块，除非有至少三分之一的验证器违反了惩罚条件。

责任的安全给我们带来了“经济确定数”的想法，如果两个互相矛盾的哈希值都得到了确定（比如说，分叉），那么我们有数学的证明说，一大系列的验证节点都违反了一些惩罚条件，我们可以向区块链提交这一项证据并且对这些违反了惩罚条件的验证节点们实行惩罚。

合理的活跃性基本上是说“算法不应该被‘卡住’，并且不应该确定任何的事情”。

同一时刻两次 prepare 惩罚

协议定义了一系列的惩罚条件，诚实的验证人遵循一个协议，保证不触发任何的条件（注意：有时候我们说“违反”一个惩罚条件，他的同义词是“触发”，把惩罚条件当作一个

法律，你不应该违反法律）。永远不要发送 PREPARE 的消息 2 次，这一点对于一个诚实的验证人来说一点都不难。

如果一个验证人发送了一个表单的签名信息：
["PREPARE", epoch, HASH1, epoch_source1]
与另一个表单的签名信息：
["PREPARE", epoch, HASH2, epoch_source2]
当 HASH1 != HASH2 或者 epoch_source1 != epoch_source2，但是 epoch 的值在两个信息中都是一样的，那么这个验证人的保证金就会被取消(slash)（例如：删除此保证金）

此条件可以用如下算子表示：

```
func double_prepare_slash(prepare1, prepare2 byte) error {
      //解析 prepare1 与 prepare2 -> ["PREPARE", epoch, HASH, epoch_source]
      [validator_index1, epoch1, sig1, epoch_source1] := parse(prepare1)
      [validator_index2, epoch2, sig2, epoch_source2] := parse(prepare2)
      //判断签名是否正确
      if (!hash(validator_index1) || !(validator_index2)) {            return
Slash
      }
      //判断 epoch 是否相同
      if (epoch1 != epoch2) {            return Slash
      }
      //判断 epoch_source 是否不同，如果相同表示违反了规则需要被惩罚
      if (epoch_source1 == epoch_source2) {            return Slash
      }
      ...      return nil}
```

prepare 和 commit 不一致惩罚

“PREPARE”和“COMMIT”是从传统的拜占庭容错共识理论中借用的术语。现在，我们来假设他们代表了两种不同类型的消息，在后面的协议内容中我们会介绍。你可以认为共识协议要求两轮不同的协议，其中 PREPARE 代表了一轮协议，COMMIT 代表了第二轮协议。

如果一个验证人发送了一个表单的 commit 签名信息：
["COMMIT", epoch1, HASH1]
与另一个表单的 prepare 签名信息：
["PREPARE", epoch2, HASH2, epoch_source]
当 epoch_source < epoch1 < epoch2，那么无论 HASH1 是否等于 HASH2，验证人都会被惩罚(slash)。该条件限定了，prepare 永远在 commit 之前，即对于一个 epoch 已经被

commit 之后，无法再对之前的 epoch 进行任何 prepare 操作。commit 已经成为确认的历史，验证人不应该篡改历史。

此条件可以用如下算子表示：

```
func double_prepare_slash(prepare_msg, commit_msg byte) error {
      //解析 prepare1 与 prepare2 -> ["PREPARE", epoch, HASH, epoch_source]
      [validator_index1, prepare_epoch, sig1, prepare_source_epoch] :=
parse(prepare_msg)
      [validator_index2, commit_epoch, sig2] := parse(commit_msg)
      //判断签名是否正确
      if (!hash(validator_index1) || !(validator_index2)) {            return
Slash
      }
      //判断 prepare 的 epoch 应该在 commit 的 epoch 之前
      if (prepare_source_epoch >= commit_epoch) {            return Slash
      }
      //判断 prepare 应该在 commit 之前
      if (commit_epoch >= prepare_epoch) {            return Slash
      }
      ...      return nil}
```

参考

[1] 最新以太坊紫皮书: https://docs.google.com/document/d/1maFT3cpHvwn29gLvtY4WcQiI6kRbN_nbCf3JlgR3m_8
[2] 老版以太坊紫皮书中文版: http://8btc.com/thread-40113-1-1.html
[3] 复杂美 Casper-Go 项目源代码
[4] Minimal Slashing Conditions: https://medium.com/@VitalikButerin/minimal-slashing-conditions-20f0b500fc6c