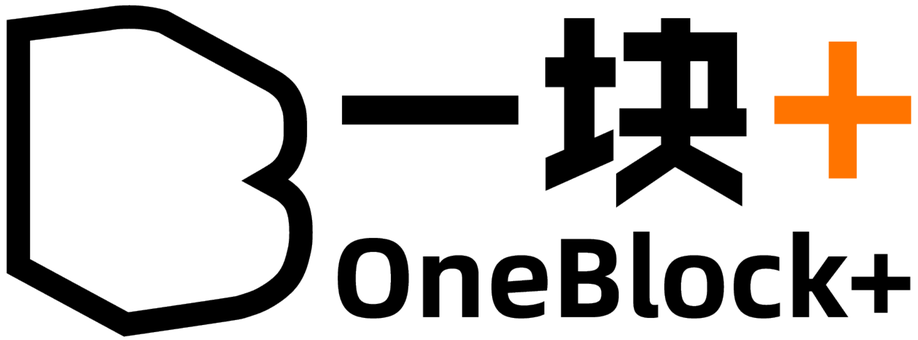
波卡跨链核心技术介绍

Alex 徐杨 | 2020-07-26

感谢



关于作者

- 徐杨
- 拓链科技CTO，HashKey Custody CTO
- https://github.com/alexxuyang/substrate-dex
- https://github.com/alexxuyang/RSA-rust

扫一扫上面的二维码图案，加我微信

说明

本文中所有带下划线的文字，都可以链接到代码库，
方便大家结合代码，再去深入理解其中的逻辑

文中所有的内容，均为作者自己的理解，可能有偏差
或错误，欢迎大家沟通并指正

牵涉到代码的地方，建议大家回去后慢慢阅读与理解

版本说明

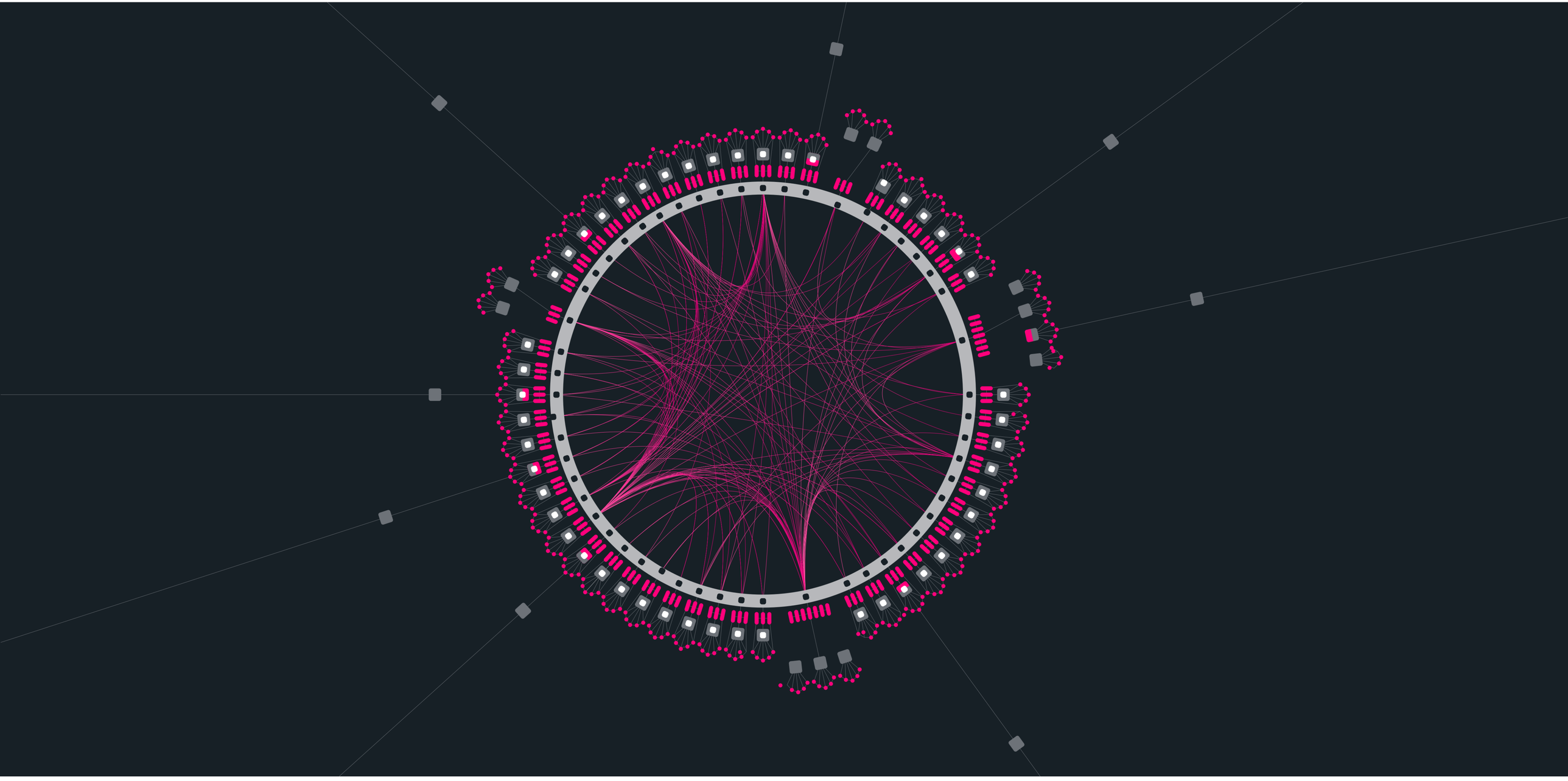
- 2020-06-25
- cumulus@05a83e8bdbcf4c216e8632431aa49dd996ce4c3f
- polkadot@91f3e68f5750dff7bddedeb846ff55447e8cf8f8
- substrate@740115371b096af4ca2c3b2c860d95ba801cffcb

为什么要了解波卡跨链机制

- 对跨链协议本身感兴趣
- 帮助更好的将parachain通过slot接入relay chain
- polkadot是学习如何深入使用substrate最好的参考

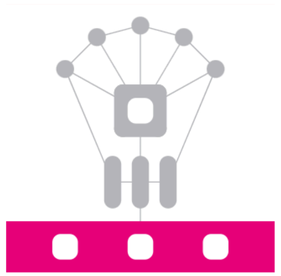
我们在理解波卡的跨链概念时，会遇到的常见问题是

- 收集人（collator），到底在干些什么？它和parachain、relay chain的关系是什么？
- 跨链，到底是什么东西在从parachain跨到relay chain?
- 平均每个链十个validators，为什么会保障安全性?
- 所谓的pooled security，到底在讲什么？
- 与跨链共识相关的各种数据，那些上链了？那些没有上链？设计的依据是什么？
- 波卡基于substrate而建，哪些模块适合放到波卡？哪些适合放到substrate?
- 会什么需要使用纠删码、VRF这些技术?
- 波卡整个生态的tps，能达到多少？为什么？



图片来源于网络

波卡架构

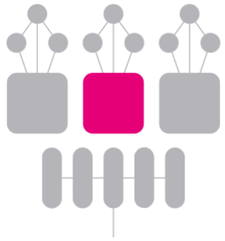


Relay Chain

Polkadot的核心， 负责网络的安全性、 共识和跨链互操作性。

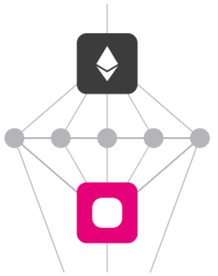
Parachains

主权区块链，可以拥有自己的代币， 并针对特定场景优化其功能。



Parathreads

类似于parachains ，但采用即用即付的收费模式。对于不需要持续连接网络的区块链来说更加经济。

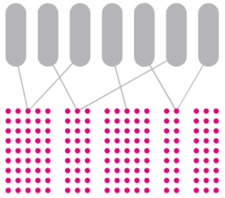


Bridges

允许parachains和parathreads连接Ethereum和Bitcoin等外部网络并与之通信。

图片来源于网络

共识参与者



Nominators

通过选择可信赖的validators，并抵押dots来保护Relay Chain。

Validators

通过抵押dots，验证collators的证据，并与其他validators达成共识来保护Relay Chain。

Collators

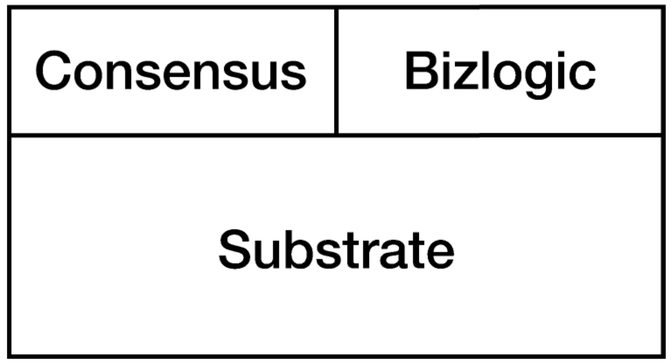
通过收集用户的分片交易，并为validator提供证明来维护分片。

Fishermen

监控网络并将不良行为报告给validators。 Collators和任何parachain全节点都可以扮演fisherman的角色。

Polkadot与Substrate的关系

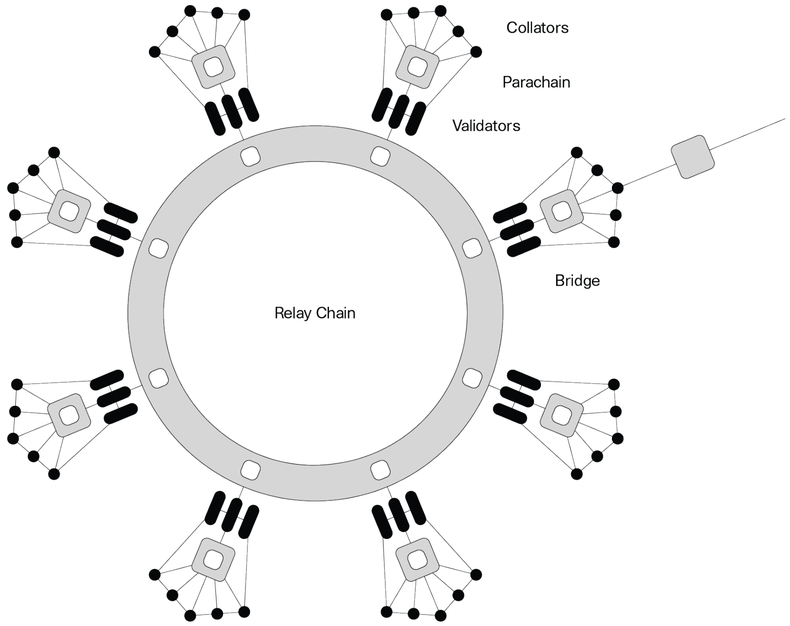
- 为了让生态伙伴快速搭建parachain，parity将polkadot中generic的逻辑抽象出来，形成了substrate
- 基于substrate之上，再去构建polkadot
- 在波卡的代码库中，会有两大定制化的逻辑代码
- 一是共识：collation、collator、candidate receipt、commitment、executor、chunks、gossip、network protocol、availability store等
- 二是业务逻辑：parachains生命周期管理、执行管理、ixo投资人映射、众筹等



= Polkadot

跨链协议流程 Availability and Validity

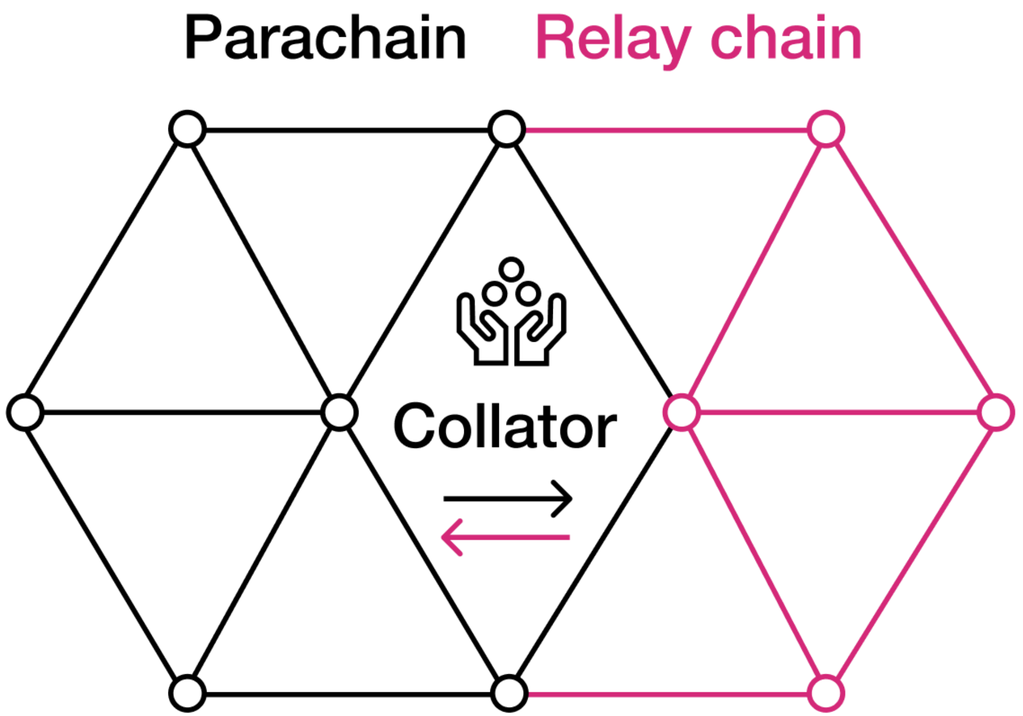
- 1. 平行链（parachains）阶段
- 2. 中继链（relay-chain）提交阶段
- 3. 可用性子协议
- 4. 在 GRANDPA 机制中执行有效性的二次检查
- 5. 钓鱼人（fisherman）的异议程序
- 6. 调用拜占庭容错的确定性小工具 （finality gadget）加强链的稳定性



图片来源于网络

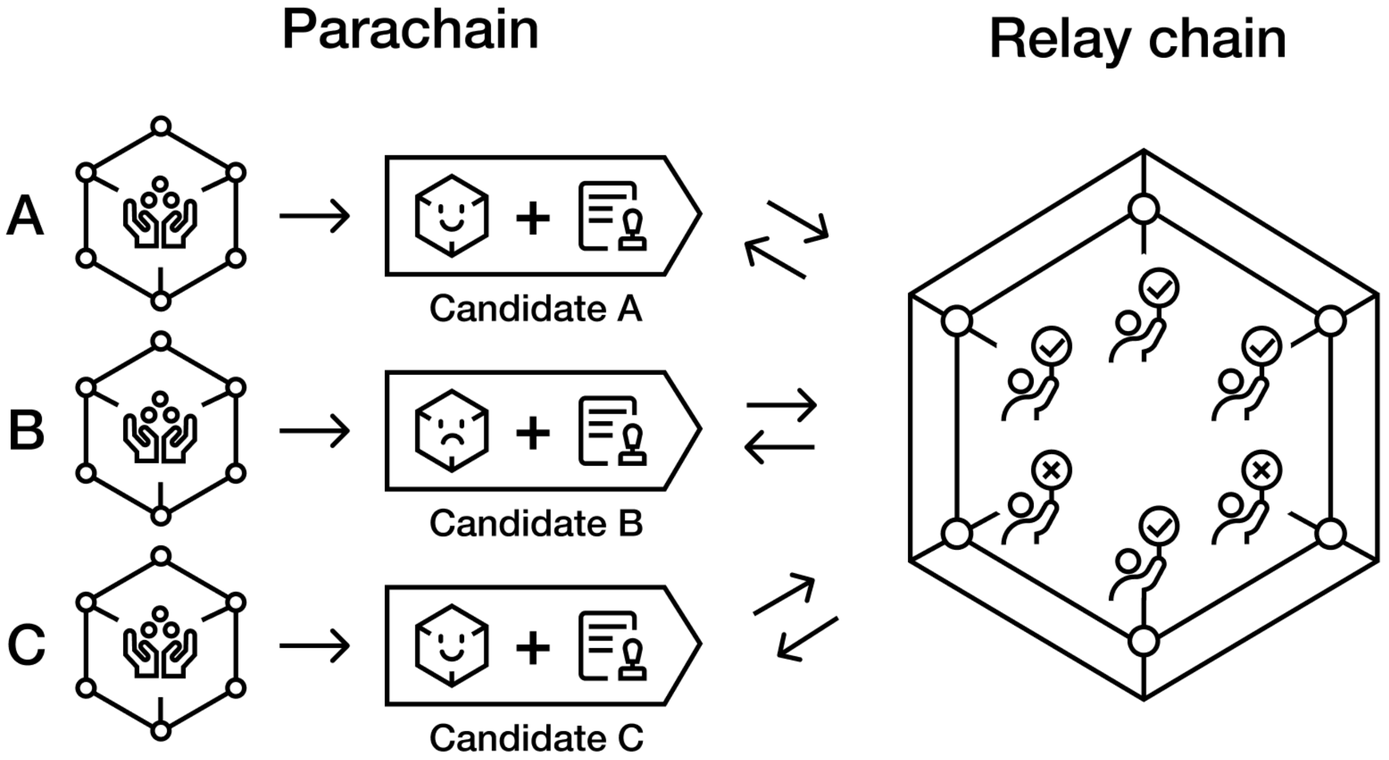
1. 平行链阶段 - 网络架构

- 收集人是一个独立的程序，它进入了两个链的p2p网络，一是parachain，二是relay chain
- 收集人是将信息从parachain摆渡到relay chain的中间节点



1. 平行链阶段 - 整体流程

- 平行链阶段是指平行链的收集人（collator）向当前分配给该平行链的一组验证人（validators）提出一个候选区块
- 收集人节点活跃在一个独特的平行链上，并提交状态转换的建议以及其有效性证明 validity proof
- 候选区块是指平行链的收集人（collator）提出的新区块，它可能是有效的，也可能是无效的
- 在被纳入中继链之前必须经过relay chain的validators的有效性检查



1. 平行链阶段 - collator激励

平行链只需要一个诚实的收集人来提交区块

从relay chain的角度看，是不需要收集人抵押dot的

对收集人的经济奖励（如果有），需要在parachain上去实现

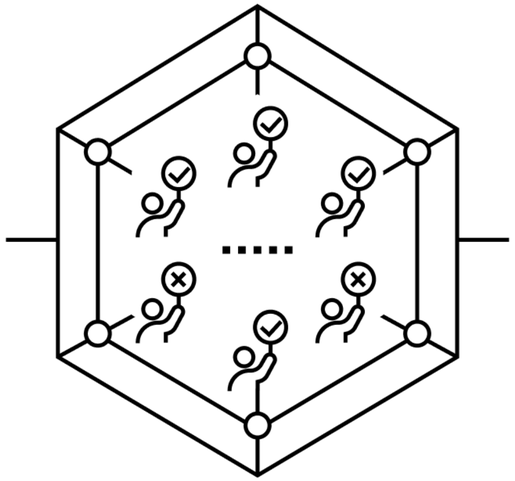
1. 平行链阶段 - 测试方法

在polkadot的代码仓库中，有一个adder模块，用来模拟一个最简单的parachain

在实际使用过程中，parachain需要参考cumulus代码库，将其接入relay chain

2. 中继链阶段 - slot

- collator生成了候选区块后，通过p2p网络，将其传送给relay chain的validators
- 在relay chain的每个区块slot中，会对每个parachain分配若干个validators
- validators_per_parachain = (validator_count - 1) / parachain_count
- 它们会负责验证候选区块



2. 中继链阶段 - duty_roaster

- 每个parachain分配的validators
- 基于链上的一个随机数（基于之前81个区块的数据生成）
- 再按照右侧的这个shuffle算法
- 进行乱序分配的
- 这里并没有使用VRF，和这个文章的描述是有出入的

```
1058        let mut roles_val : Vec<Chain> = (0..validator_count).map( f: |i : usize | match i {
1059            i : usize if i < parachain_count * validators_per_parachain => {
1060                let idx : usize = i / validators_per_parachain;
1061                Chain::Parachain(parachains[idx].0.clone())
1062            }
1063            _ => Chain::Relay,
1064        }).collect::<Vec<_>>();
```

```
1086        // shuffle
1087        for i : i32 in 0..(validator_count.saturating_sub(1)) {
1088            // 4 bytes of entropy used per cycle, 32 bytes entropy per hash
1089            let offset : usize = (i * 4 % 32) as usize;
1090
1091            // number of roles remaining to select from.
1092            let remaining : usize = sp_std::cmp::max( v1: 1, v2: (validator_count - i) as usize);
1093
1094            // 8 32-bit ints per 256-bit seed.
1095            let val_index : usize = u32::decode( input: &mut &seed[offset..offset + 4])
1096                .expect( msg: "using 4 bytes for a 32-bit quantity") as usize % remaining;
1097
1098            if offset == 28 {
1099                // into the last 4 bytes - rehash to gather new entropy
1100                seed = BlakeTwo256::hash( s: seed.as_ref());
1101            }
1102
1103            // exchange last item with randomly chosen first.
1104            roles_val.swap( a: remaining - 1, b: val_index);
1105        }
```

2. 中继链阶段 - VRF

- VRF (Verifiable Random Function)，babe中确定出块validator
- hash函数：result = hash(info)
- result = hash(SK, info)
- result = vrf_hash(SK, info)
- proof = vrf_proof(SK, info)
- vrf_verify(PK, info, result, proof) -> T/F
- 其中，SK/PK是validator的key pair
- 这样，既可以验证info与result的关联关系，也可以知道是哪个validator签署的

2. 中继链阶段 - VRF在波卡中的应用

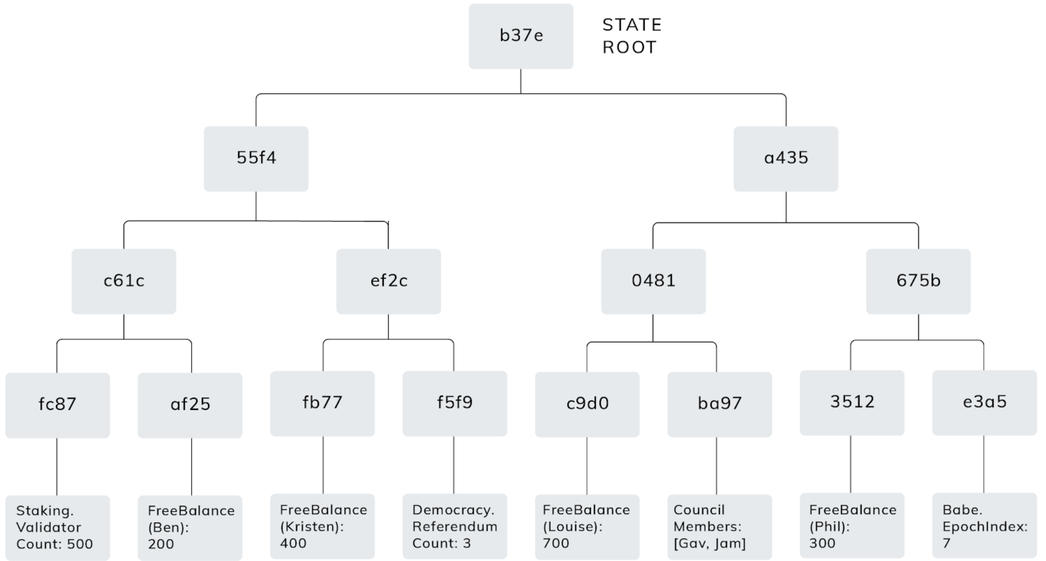
- result = hash(SK, info)
- result = vrf_hash(SK, info)
- proof = vrf_proof(SK, info)
- vrf_verify(PK, info, result, proof) -> T/F
- result是经过vrf计算后的结果，在polkadot中，result需要小于一个阈值才算抽中签，参考代码1、代码2
- info的值，在每个slot中，是一个全网固定的值
- 在polkadot中，info称作transcript，是将：一个随机数（每个epoch中固定，将N-2个epoch前的随机数拼接的Hash，代码1、代码2）、slot number、epoch index，这三个元素拼在一起
- 因为validator是否选中来出块，是根据vrf计算而定的，所以一个slot，会有一个、多个、或零个节点在vrf过程中胜出
  - 一个：正常情况（primary slot leaders)
  - 多个：多个validators出块，最后，需要grandpa来确定哪个分叉为最终链
  - 零个：使用round robin确定一个出块节点（secondary slot leaders）

2. 中继链阶段

1. 接收candidate block
2. 使用STF(State Transition Function)验证状态转移是否正确
3. 广播验证结果commitment & block给其它validators
4. 一旦超过一半的validators认可，便准备candidate receipt
5. 使用纠删码技术保存block数据
6. 发送交易到节点交易池
7. 打包节点将candidate receipt打包进入区块

2.2 STF - 默克尔树

- 平行链的状态保存在一个称为默克尔树的结构中
- 所有的状态都是该树的叶子
- 例如，Phil - FreeBalance：300
- 相邻的两个节点取其hash值，以此递归往上
- 直到状态根（state root）- b37e



图片来源于网络

2.2 STF - 验证

- 默克尔树具有一种方便的属性：如果某些值发生更改，则仅查看新修改的值，以及该树中未受影响数据根Hash，即可验证更改

- 基于此属性，验证程序可以验证状态转换，而无需访问整个状态。 它只需要：

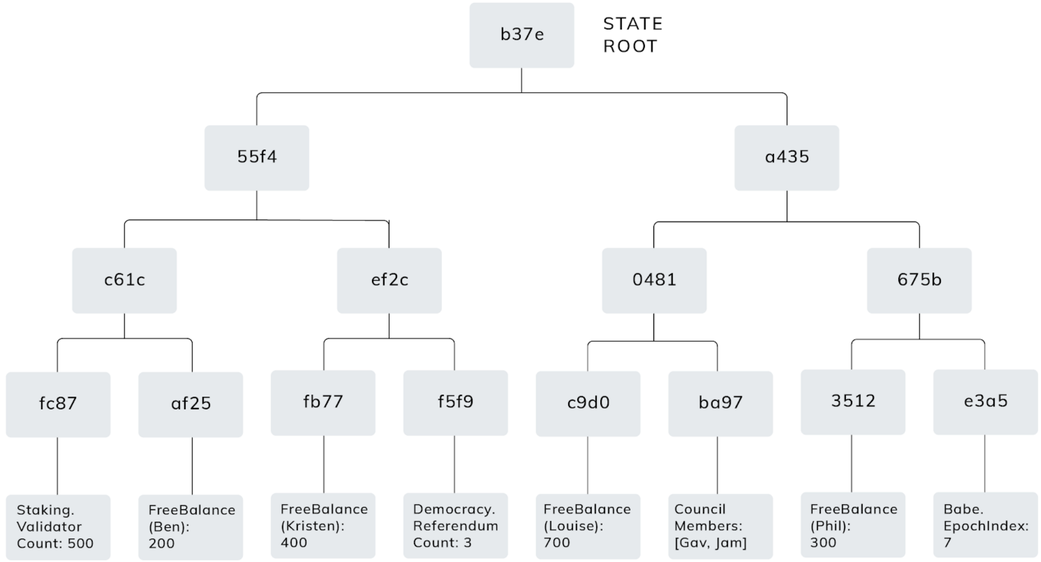
  - 该区块所修改的平行链数据库中的值

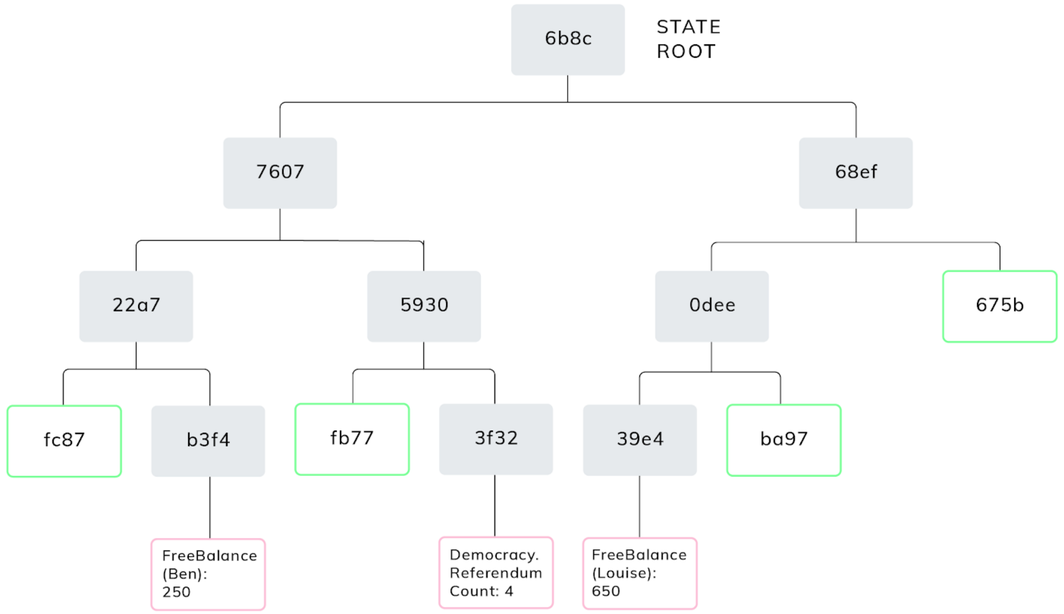
  - 区块数据（交易列表：状态转换）

  - Merkle树中未受影响数据根哈希

- 这套信息构成了有效性的证明。 哈希值具有固定的长度

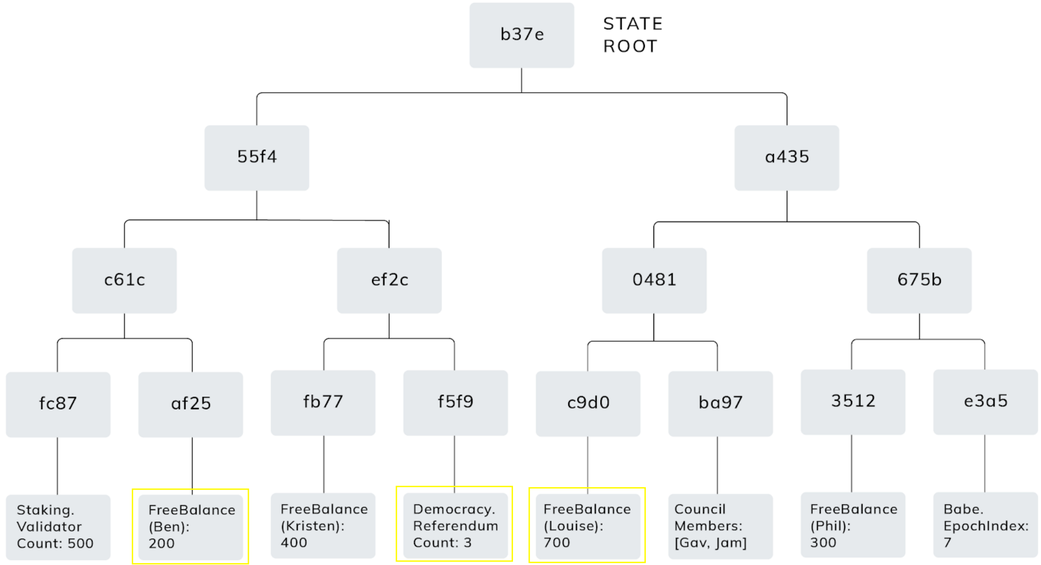
- 未修改的值有多大都没有关系； 哈希足以表示它们

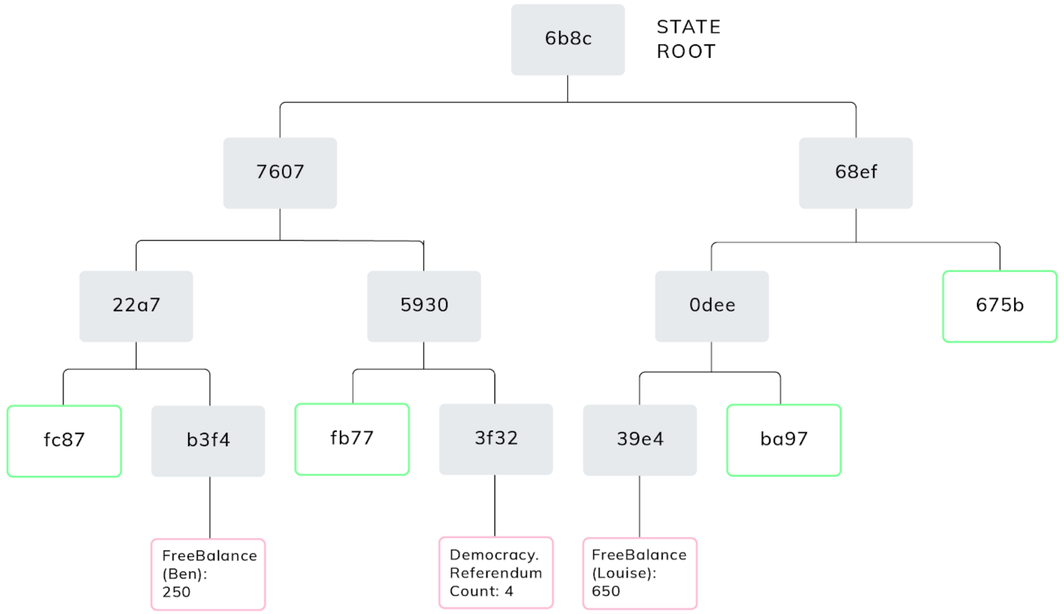




2.2 STF - 验证过程

- 我们来看看relay chain的验证的具体过程
- collator提供以下数据：
  - 1 该区块所修改的平行链数据库中的值，黄框的内容
  - 2 区块（状态转换列表）
  - 3 Merkle树中未受影响数据根哈希，绿框的内容
- relay chain的validator，基于1、3，可以计算出parent block的状态根 S1
- relay chain的validator，使用parachain的STF wasm镜像，基于修改的状态数据 - 1，执行区块交易 - 2，保证只修改了“1”中的内容，同时得到新的值，红框的内容 - 4
- 基于3、4计算当前block的状态根 S2
- 将（S1，S2）与collator传送过来的数据进行对比，相同则表示验证成功

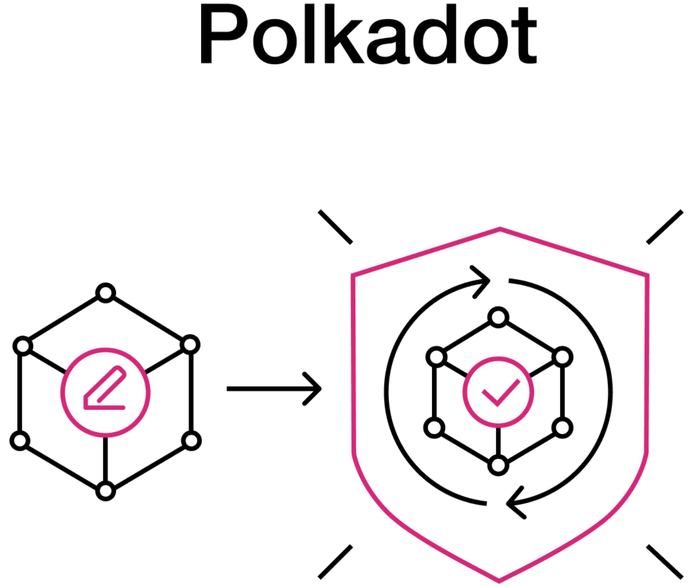




图片来源于网络

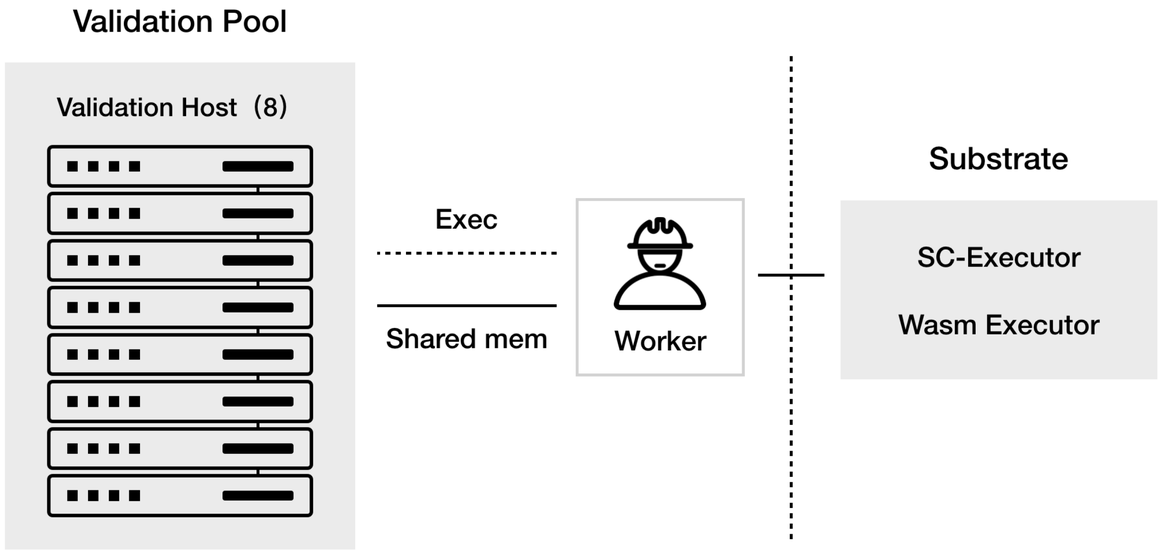
2.2 trust free shared security

- Polkadot验证程序不会检查平行链状态中的每个值
- 而只会检查已修改的值，以确保修改有效
- Polkadot不保证有效状态，它保证了有效的状态转换
- 如果一条链以有效状态加入Polkadot网络并在Polkadot安全性的保护下执行其所有转换，则它将具有有效状态



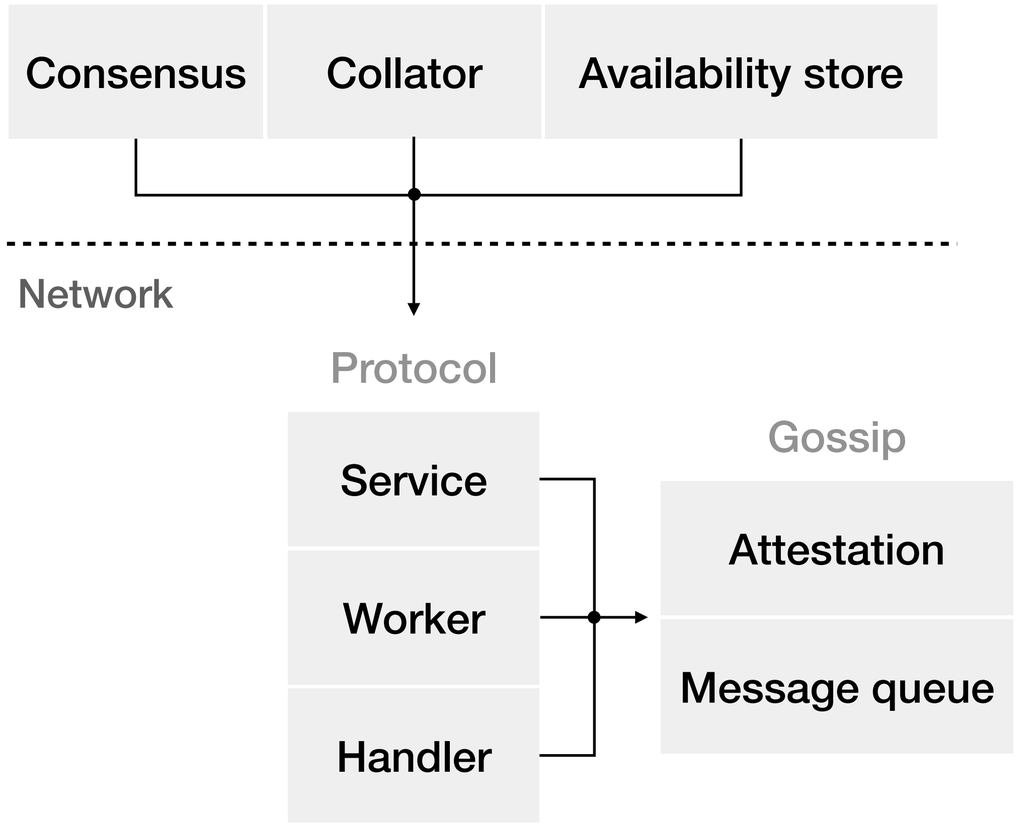
2.2 STF - executor

- validator最终使用executor验证器，来做STF验证
- validator上有一个validation_pool
- pool中最多启动八个validation host
- 在有验证工作时，host会通过exec创建单独进程worker，args：validation_worker
- host与worker之间通过shared memory通信
- worker最终调用了substrate->sc-executor->wasm_executor来做stf的验证



2.3 广播验证结果 - network设计

- 在substrate的network协议之上，波卡基于自身的共识逻辑，构建了一套定制化的网络协议
- 主要由两个组件组成，protocol 和 gossip
- protocol的核心是worker，是所有消息的核心路由
- handler是大多逻辑的具体实现层，供protocol使用
- service包装了protocol，并对外提供各种trait实现
- gossip使用了substrate的gossip engine来发送消息
- gossip实现了两个主要功能
- attestation 和 message queue

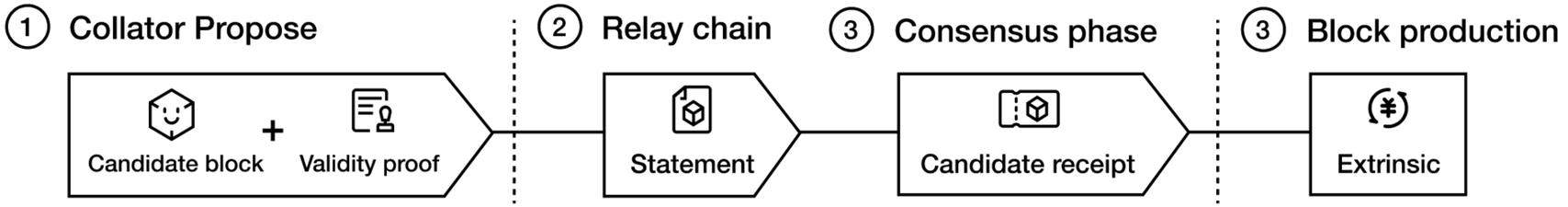


2.4 候选回执(candidate receipt)

- 对于每个parachain的一个区块，最终在relay chain上链的是这个候选回执
- candidate receipt是大小恒定的数据
- 每个参数都是ID、Hash、签名
- 这样在parachain扩展的时候，是一个线性的过程
- 区块数据、状态数据、纠删码数据等，都保存在本地的一个key-value数据库中，availability-store

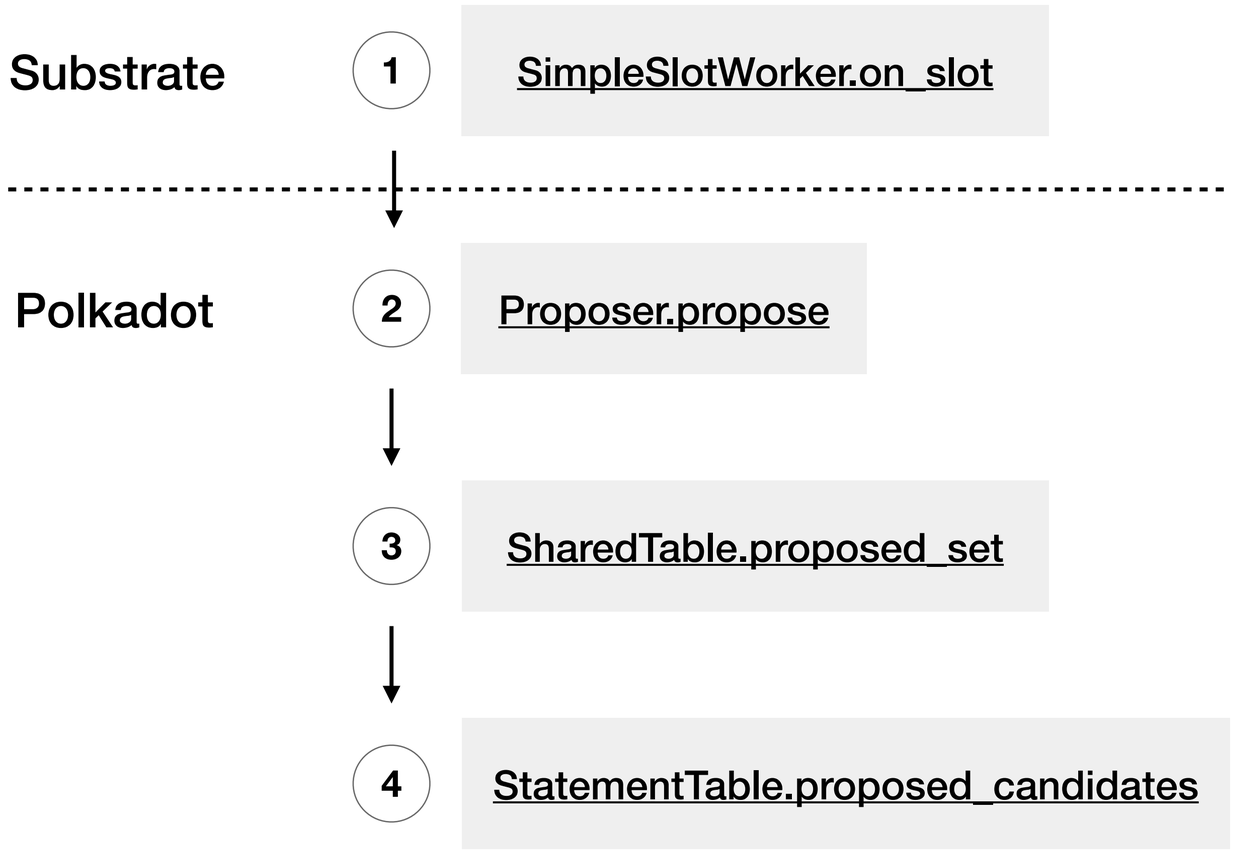
Candidate Receipt

- parachain的ID
- collator的ID 和 签名
- 父区块的candidate receipt的Hash
- 纠删码的默克尔根
- 外发消息的默克尔根
- 区块Hash
- parachain在执行该区块前的状态根
- parachain在执行该区块后的状态根



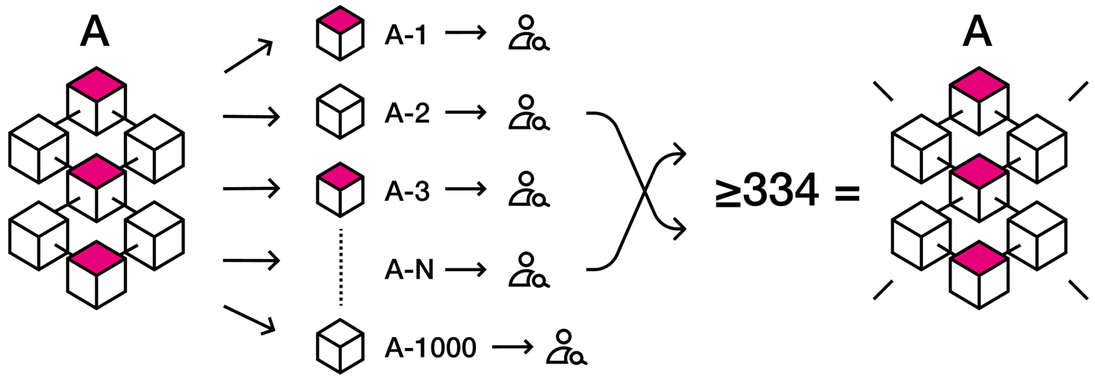
2.7 打包区块

- 一旦超过一半的validators认可，便准备candidate receipt
- 每个slot，回调substrate中的on_slot函数
- 波卡验证组件中的block_production
- SharedTable保存了一轮共识中，所有的过程数据validated、承诺数据StatementTable、candidate receipt的进展status、availability_store引用
- StatementTable保存了一轮共识中，validator的签名数据、不当行为misbehavior数据、candidate的投票数据；所有数据都是hash、sig、ID
- 区块数据、状态数据、纠删码数据等，都保存在本地的一个key-value数据库中，availability-store
- 对于babe中赢得slot的validator A而言，它的交易池中会有很多candidate receipt，需要保证：
  - candidate receipt的parent candidate已经收录在之前的区块
  - A的本地availability_store中已经保存了该candidate的纠删码chunk



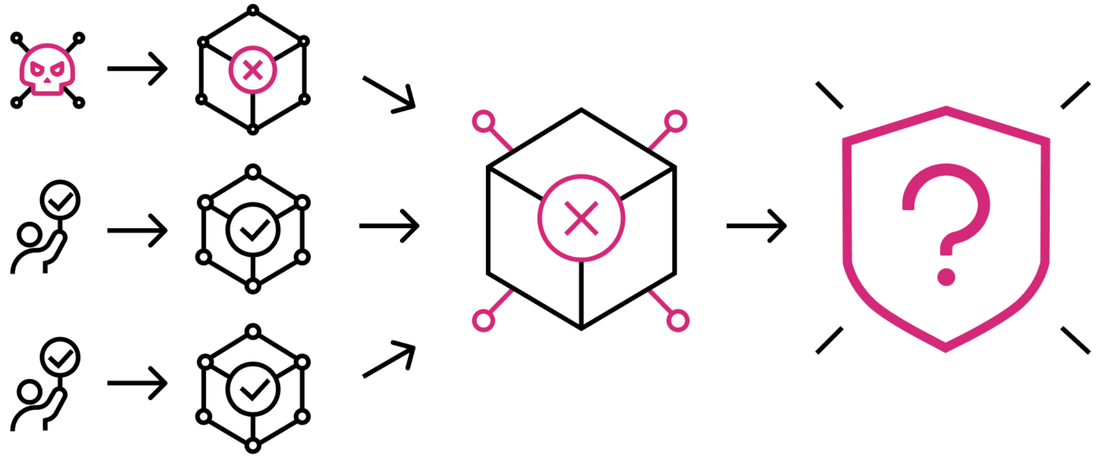
3. 可用性子协议 - 纠删码

- 在candidate receipt生成后、进入交易池之前，validators需要保存区块（block） + 有效性证明（proof of validity）数据
- 生成candidate receipt的validator，同时生成纠删码数据chunks
- 纠删码会将一个数据A分散成n份分片数据，A1…A100。当超过某个阈值（34）的分片组合在一起时，就可以还原出原始数据A
- 多项式插值算法，可以用在这样的场景：知道两个点可以确定一条线，知道三个点可以确定一条抛物线，知道四个点可以确定一条三次曲线
- 在polkadot中，假设总共有1000个validators，纠删码chunks也发送到了这1000个节点上，那么最终需要超过1/3的节点 - 334个节点的chunks就可以恢复出原始数据
- 收到candidate receipt以及纠删码的验证者，会将candidate receipt包括在中继链交易池中，block author可以再将其包含在一个块中



4-6. 更多的安全保障

- 由于波卡只会分配10-20个validators给一个parachain
- 超过1/2的确认便会将某parachain的candidate receipt打包
- 所以这里还是会小概率的存在validators合谋做坏事的情况
- 所以，在将candidate receipt打包到区块后
- 还需要更多的安全机制设计，来保证安全
  - 二次验证：将出块（babe）与最终确认（grandpa）分开，提供一个可以做二次验证的时机
  - 监督机制：fisherman



4-6. 更多的安全保障

1000个节点中，10个一组，分出100组，有六个节点是坏节点，
他们被分到一组的概率是多少?

C(4/994) / C(10/1000) = 1.53e-13

有十个节点是坏节点，至少六个被分到一组的概率是多少?

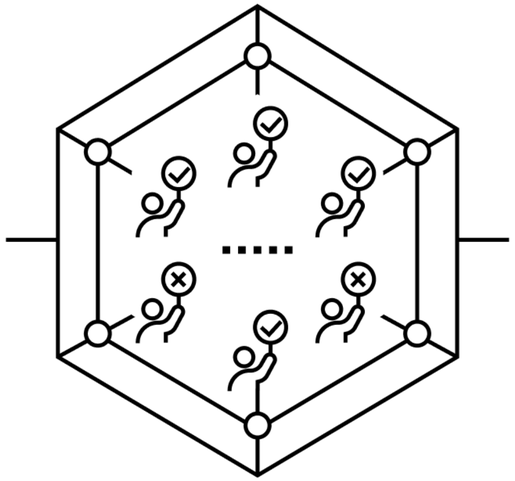
(C(6/10) * C(4/994) + C(7/10) * C(3/994) + C(8/10) * C(2/994) + C(9/10) * C(1/994) + C(10/10)) /
C(10/1000) = 3.23e-11

6秒一个区块，大概要6000年

坏节点数翻个倍，概率提升2个数量级

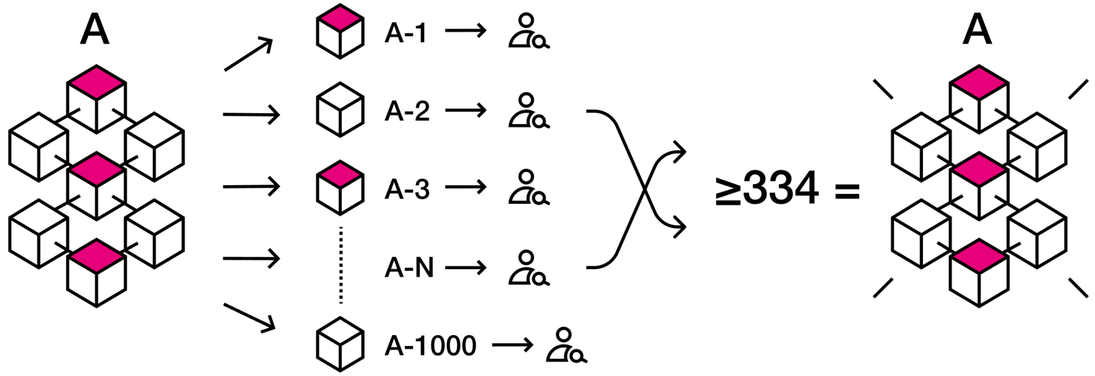
4-6. 更多的安全保障 - 二次验证 - STF

- 当relay chain生成了一个新区块后，会启动一个二次验证的会话
- relay chain会随机挑选validator对区块中所有的candidate receipt进行验证
- 验证过程包括：
  - 通过纠删码chunks恢复出block data、validity proof
  - 按照前文STF验证的过程，做一次重新的验证
  - 最终比对是否可以生成相同的candidate receipt



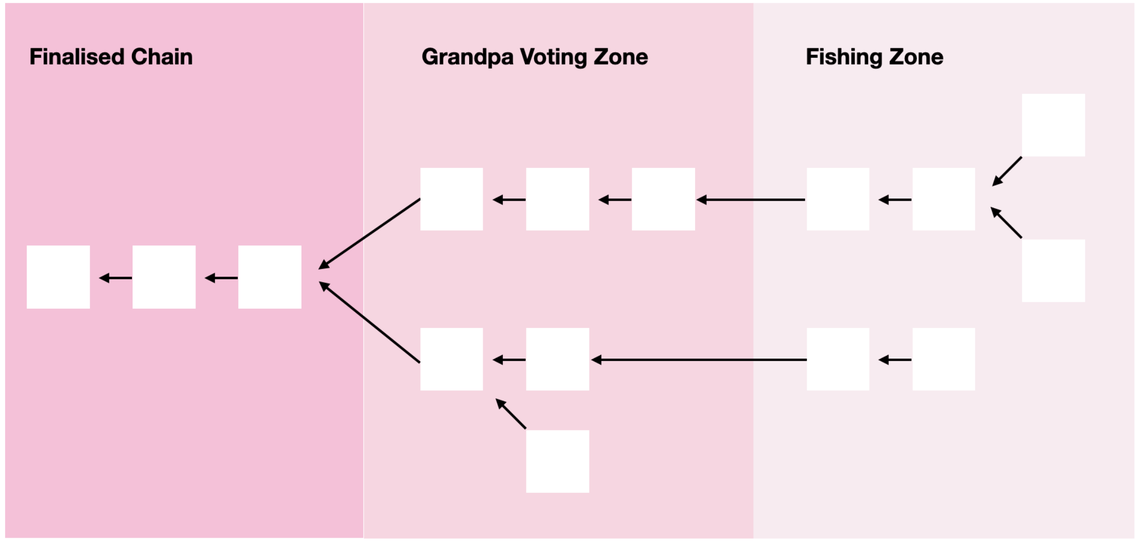
4-6. 更多的安全保障 - 二次验证 - 纠删码有效性验证

- 当relay chain生成了一个新区块后，会将block广播给所有的validators
- 每个validators接收到新区块后，会执行一个import block的操作
- 该操作中，会对区块中的每个candidate receipt，查看其对应的纠删码chunk是否保存在本地
- 如果没有保存在本地，则在p2p网络中发出一个warning信息，表明该receipt对应的chunk发生了遗漏
- 如果超过1/3的validators，都对某个receipt发出了warning信息，则该区块立即作废



4-6. 更多的安全保障 - 二次验证 - Fishing Zone

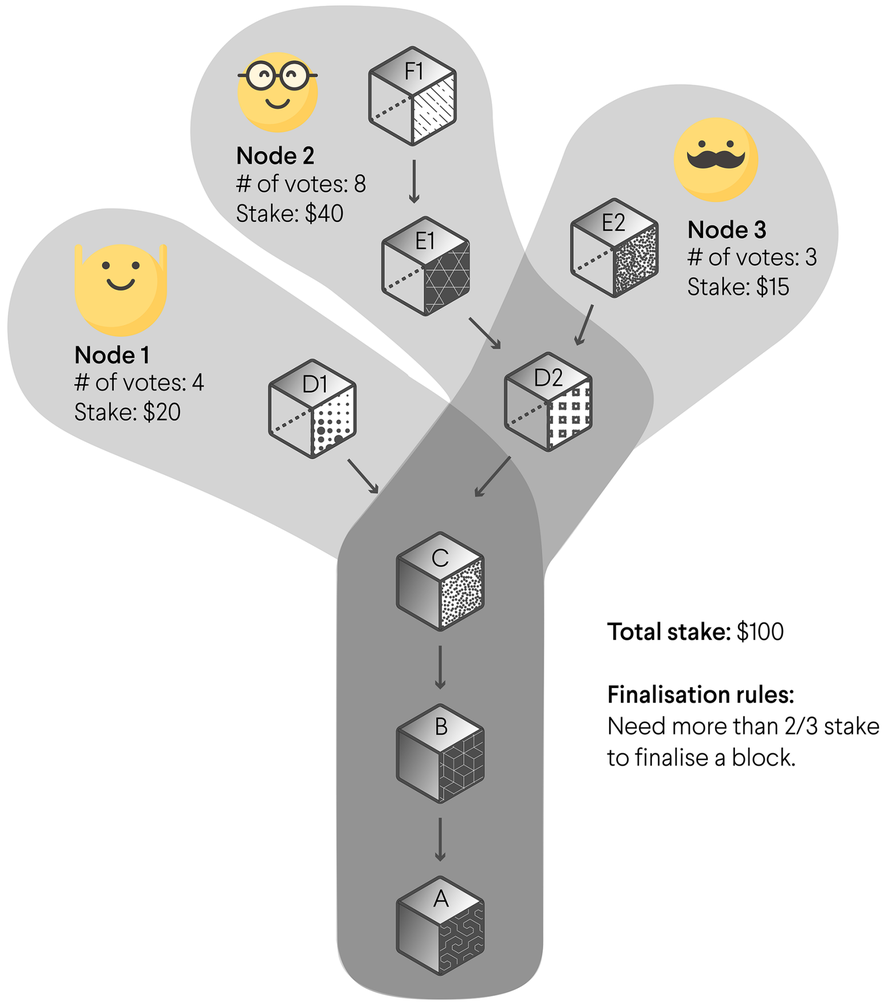
- 接下来，collators和fisherman，会执行更多的验证工作
- fisherman和collator一样，是parachain、relay chain？的节点
- 成为fisherman，首先需要抵押一定数量的dot
- fisherman负责验证区块中所有candidate receipt的有效性（validity）
- collator负责验证纠删码（chunks）的可用性（availability）
- 一旦发现问题，fisherman便会发出invalidity（抵押dot）、collator发出unavailability的消息
- 随着关于某个candidate receipt的invalidity、unavailability消息越来越多，需要更多的validators参与验证
- 最终，如果对于某个candidate receipt，超过1/3的validators都验证是invalidity或者unavailability，则该block作废



图片来源于网络

4-6. 更多的安全保障 - GRANDPA

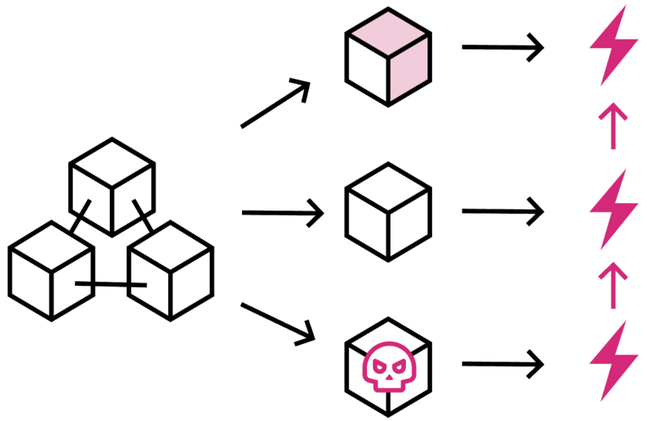
- GRANDPA模块，负责区块的最终性确认
- GRANDPA做最终性确认时，确认的是链，而不是区块
- 该协议可传递地应用投票，并且GRANDPA算法找到具有足够数量的投票的最高区块号，以将其视为最终投票。 此过程允许在一个回合中完成N个块的确认
- Polkadot中的所有parachain都遵循relay chain的终结性，未来的parablock必须始终以最终确认的、中继链中的candidate receipt为基础
- 一旦完成，该块将从共享的安全环境中受益，该安全环境允许链以无信任的方式彼此交互，而撤销该块意味着撤销relay chain的区块，那这是非常难的



图片来源于网络

4-6. 更多的安全保障 - 分叉（Fork）

- 前面讲过，如果超过1/3的validators在产块后，对某个candidate receipt验证时出错，则区块作废
- relay chain是使用babe出块的，它不停的在出块，所以“作废”操作，是在一个多分叉（fork）的环境下进行，这样事情又复杂了很多
- 在一个fork上有作恶行为的validator，要在所有的fork上进行惩罚
- 有争议的parablock没有出现在所有的fork中
- 有作恶行为的validator，不能创建fork以去掉自己的作恶行为
- 如果出块validator发现某fork上存在有争议的parablock，会导致该fork出现revert，则最好激励该validator在不包含该parablock的其它fork上构建区块



4-6. 更多的安全保障 - 本地争议（Local Disputes）

- 争议：已经生candidate receipt、打包后，在后续验证过程中出现了问题的情况
- 本地争议：对于当前fork中存在的争议
- 已经打包、且在验证期的parablock，可以有本地争议
- 对于存在争议的parablock，需要保证其block、code、proof都可用
- 争议可能会存在于之前的era，需要上一轮的validators set进行争议解决；这说明validator duty延长了一个era
- 争议解决后，在错误那一方的validators会被slash
- 区块和链会revert到争议前的位置；争议区块会进入黑名单

4-6. 更多的安全保障 - 远程争议（Remote Disputes）

- 远程争议：已在另一fork部分或全部解决的争议，该争议的结论、奖惩结论，都需要同步到其它的fork
- 但是，这样会带来很多的挑战
- 与本地争议一样，parablock所在fork的validators set负责解决并确定candidate的可用性
- 如果可用性验证没有通过，那么超时后（可用性验证失败），则会惩罚不支持争议的那些validators；这个结论也会同步到所有其它分支
- 如果可用性验证通过，那么可以将该远程争议replay到所有其它fork中，需要保证在所有fork中选择validators set的VRF是相同的
- 确保每个分支上都replay了该远程争议并都得到了相同的结论后，该远程争议结束
- 对于每个replay该争议的分支，按照上节执行本地争议的逻辑进行处理

关于波卡的TPS

- 每个parachain的tps是1000
- 总共有100个parachains
- relay chain有1000个validators，每个parachain分配10个validators
- 在每个relay chain的slot，每组validators只验证一个parachain
- relay chain 与 parachain的block time都是3秒
- ((1000 * 3) * 100) / 3 = 100,000 tps

相信讲到这里，大家对之前的问题，应该都有自己的理解或思路

参考文档链接

- The Path of a Parachain Block
- Availability and Validity
- polkadot-path-of-a-parachain-block（video）
- Availability and Validity by web3
- Parachain Allocation
- Security of the network
- Polkadot Consensus Part 1: Introduction
- Polkadot Consensus Part 2: GRANDPA
- Polkadot Consensus Part 3: BABE
- Validity Module

谢谢

附录：部分源码浅析

1. 平行链阶段

- run_collator@cumulus/test/parachain/src/service.rs
- 86-103行，启动一个parachain的节点
- 116行，定义了一个在parachain上发布区块的闭包，后面会使用到
- 127行，再启动一个collator（polkadot节点+collator逻辑）
- 128行传入的builder，可以生成一个context，做为在collator逻辑里回调parachain的入口
- 后面的produce_candidate就是通过该context进行回调的

```rust
/// Run a collator node with the given parachain `Configuration` and relaychain `Configuration`
///
/// This function blocks until done.
pub fn run_collator(
    parachain_config: Configuration,
    key: Arc<CollatorPair>,
    polkadot_config: polkadot_collator::Configuration,
    id: polkadot_primitives::parachain::Id,
) -> sc_service::error::Result<impl AbstractService> {
    let parachain_config = prepare_collator_config(parachain_config);

    let (builder, inherent_data_providers) = new_full_start!(parachain_config);
    inherent_data_providers
        .register_provider(sp_timestamp::InherentDataProvider)
        .unwrap();

    let block_announce_validator = DelayedBlockAnnounceValidator::new();
    let block_announce_validator_copy = block_announce_validator.clone();
    let service = builder
        .with_informant_prefix(format!("[{}] ", Color::Yellow.bold().paint("Parachain")))?
        .with_finality_proof_provider(|client, backend| {
            // GenesisAuthoritySetProvider is implemented for StorageAndProofProvider
            let provider = client as Arc<dyn StorageAndProofProvider<_, _>>;
            Ok(Arc::new(GrandpaFinalityProofProvider::new(backend, provider)) as _)
        })?
        .with_block_announce_validator(|_client| Box::new(block_announce_validator_copy))?
        .build_full()?;

    let registry = service.prometheus_registry();

    let proposer_factory = sc_basic_authorship::ProposerFactory::new(
        service.client(),
        service.transaction_pool(),
        registry.as_ref(),
    );

    let block_import = service.client();
    let client = service.client();
    let network = service.network();
    let announce_block = Arc::new(move |hash, data| network.announce_block(hash, data));
    let builder = CollatorBuilder::new(
        proposer_factory,
        inherent_data_providers,
        block_import,
        id,
        client,
        announce_block,
        block_announce_validator,
    );

    let polkadot_future = polkadot_collator::start_collator(
        builder,
        id,
        key,
        polkadot_config,
        Some(format!("[{}] ", Color::Blue.bold().paint("Relaychain"))),
    )
    .map(|_| ());
    service.spawn_essential_task("polkadot", polkadot_future);

    Ok(service)
}
```

1. 平行链阶段

- start_collator@polkadot/collator/src/lib.rs
- 436行，启动polkadot节点，这里启动是不带验证功能的fullnode
- 446行，启动了collator服务
- 通过polkadot的spawn_handle来启动
- 可见，collator服务是挂载在polkadot体系内部的服务

```rust
/// Async function that will run the collator node with the given `RelayChainContext` and `ParachainContext
/// built by the given `BuildParachainContext` and arguments to the underlying polkadot node.
pub async fn start_collator<P>(
    build_parachain_context: P,
    para_id: ParaId,
    key: Arc<CollatorPair>,
    config: Configuration,
    informant_prefix: Option<String>,
) -> Result<(), polkadot_service::Error>
where
    P: 'static + BuildParachainContext,
    P::ParachainContext: Send + 'static,
    <P::ParachainContext as ParachainContext>::ProduceCandidate: Send,
{
    if matches!(config.role, Role::Light) {
        return Err(
            polkadot_service::Error::Other("light nodes are unsupported as collator".into())
        .into());
    }

    if config.chain_spec.is_kusama() {...} else if config.chain_spec.is_westend() {
        let (service, client, handlers) = service::westend_new_full(
            config,
            Some((key.public(), para_id)),
            None,
            false,
            6000,
            None,
            informant_prefix,
        )?;
        let spawn_handle = service.spawn_task_handle();
        build_collator_service(
            spawn_handle,
            handlers,
            client,
            para_id,
            key,
            build_parachain_context
        )?.await;
    } else {
        let (service, client, handles) = service::polkadot_new_full(
            config,
            Some((key.public(), para_id)),
            None,
            false,
            6000,
            None,
            informant_prefix,
        )?;
        let spawn_handle = service.spawn_task_handle();
        build_collator_service(
            spawn_handle,
            handles,
            client,
            para_id,
            key,
            build_parachain_context,
        )?.await;
    }

    Ok(())
}
```

1. 平行链阶段

- build_collator_service@polkadot/collator/src/lib.rs
- 这是build_collator_service的第一部分
- 290行，创建了parachain_context，该部分内容定义在cumulus里，稍后介绍

```rust
#[cfg(not(feature = "service-rewr"))]
fn build_collator_service<SP, P, C, R, Extrinsic>(
    spawner: SP,
    handles: FullNodeHandles,
    client: Arc<C>,
    para_id: ParaId,
    key: Arc<CollatorPair>,
    build_parachain_context: P,
) -> Result<impl Future<Output = ()> + Send + 'static, polkadot_service::Error>
    where
        C: PolkadotClient<
            service::Block,
            service::TFullBackend<service::Block>,
            R
        > + 'static,
        R: ConstructRuntimeApi<service::Block, C> + Sync + Send,
        <R as ConstructRuntimeApi<service::Block, C>>::RuntimeApi:
            sp_api::ApiExt<
                service::Block,
                StateBackend = <service::TFullBackend<service::Block> as service::Backend<service::Block>>::State,
            >
            + RuntimeApiCollection<
                Extrinsic,
                StateBackend = <service::TFullBackend<service::Block> as service::Backend<service::Block>>::State,
            >
            + Sync + Send,
        P: BuildParachainContext,
        P::ParachainContext: Send + 'static,
        <P::ParachainContext as ParachainContext>::ProduceCandidate: Send,
        Extrinsic: service::Codec + Send + Sync + 'static,
        SP: Spawn + Clone + Send + Sync + 'static,
{
    let polkadot_network = handles.polkadot_network
        .ok_or_else(|| "Collator cannot run when Polkadot-specific networking has not been started")?;

    // We don't require this here, but we need to make sure that the validation service is started.
    // This service makes sure the collator is joining the correct gossip topics and receives the appropiate
    // messages.
    handles.validation_service_handle
        .ok_or_else(|| "Collator cannot run when validation networking has not been started")?;

    let parachain_context = match build_parachain_context.build(
        client.clone(),
        spawner,
        polkadot_network.clone(),
    ) {
        Ok(ctx) => ctx,
        Err(()) => {
            return Err("Could not build the parachain context!".into())
        }
    };
```

1. 平行链阶段

- build_collator_service@polkadot/collator/src/lib.rs
- 这是build_collator_service的第二部分
- 304行，notification_stream是波卡链上import block的stream
- 每次有block import后，会触发该stream的next()函数
- 333行，通过runtime api拿到validators list
- 335行，拿到当前parachain分配的validators
- 341行，通过函数collate去生成collation（区块）
- 350行，在reley chain网络上广播collation

```
301     let work = async move {
302         let mut notification_stream = client.import_notification_stream();
303
304         while let Some(notification) = notification_stream.next().await {
305             macro_rules! try_fr {...}
315
316             let relay_parent = notification.hash;
317             let id = BlockId::hash(relay_parent);
318
319             let network = polkadot_network.clone();
320             let client = client.clone();
321             let key = key.clone();
322             let parachain_context = parachain_context.clone();
323
324             let work = future::lazy(move |_| {
325                 let api = client.runtime_api();
326                 let global_validation = try_fr!(api.global_validation_schedule(&id));
327                 let local_validation = match try_fr!(api.local_validation_data(&id, para_id)) {
328                     Some(local_validation) => local_validation,
329                     None => return future::Either::Left(future::ok(())),
330                 };
331                 let downward_messages = try_fr!(api.downward_messages(&id, para_id));
332
333                 let validators = try_fr!(api.validators(&id));
334
335                 let targets = compute_targets(
336                     para_id,
337                     validators.as_slice(),
338                     try_fr!(api.duty_roster(&id)),
339                 );
340
341                 let collation_work = collate(
342                     relay_parent,
343                     para_id,
344                     global_validation,
345                     local_validation,
346                     downward_messages,
347                     parachain_context,
348                     key,
349                 ).map_ok(move |collation| {
350                     network.distribute_collation(targets, collation)
351                 });
352
353                 future::Either::Right(collation_work)
354             });
355
356             let deadlined = future::select(
357                 work.then(|f| f).boxed(),
358                 futures_timer::Delay::new(COLLATION_TIMEOUT)
359             );
360
361             let silenced = deadlined
362                 .map(|either| {...});
367
368             let future = silenced.map(drop);
369
370             tokio::spawn(future);
371         }
372     }.boxed();
373
374     Ok(work)
375 }
```

1. 平行链阶段

- collate@polkadot/collator/src/lib.rs
- 178行，通过回调para_context的produce_candidate，生成一个区块数据
- produce_candidate定义在cumulus中
- 190行，使用collator的私钥对区块签名
- 196、205行，生成collation数据

```rust
pub async fn collate<P>(
    relay_parent: Hash,
    local_id: ParaId,
    global_validation: GlobalValidationSchedule,
    local_validation_data: LocalValidationData,
    downward_messages: Vec<DownwardMessage>,
    mut para_context: P,
    key: Arc<CollatorPair>,
)
    -> Result<parachain::Collation, Error>
    where
        P: ParachainContext,
        P::ProduceCandidate: Send,
{
    let (block_data, head_data) = para_context.produce_candidate(
        relay_parent,
        global_validation,
        local_validation: local_validation_data,
        downward_messages,
    ).map_err(Error::Collator).await?;

    let pov_block = PoVBlock {
        block_data,
    };

    let pov_block_hash: Hash = pov_block.hash();
    let signature = key.sign(&parachain::collator_signature_payload(
        &relay_parent,
        parachain_index: &local_id,
        &pov_block_hash,
    ));

    let info = parachain::CollationInfo {
        parachain_index: local_id,
        relay_parent,
        collator: key.public(),
        signature,
        head_data,
        pov_block_hash,
    };

    let collation = parachain::Collation {
        info,
        pov: pov_block,
    };

    Ok(collation)
}
```

1. 平行链阶段

- Collator@cumulus/collator/src/lib.rs
- Collator类型定义
- proposer_factory，是提议者实例（Proposer），可以propose一个区块
- inherent_data_providers，提供inherent数据的实例，包括babe、grandpa、timestamp等
- block_import，提供区块导入的逻辑
- 以上三个都是parachain上的对象实例，下面的network是relay chain的对象实例
- collator_network，relay chain的network对象

```rust
/// The implementation of the Cumulus `Collator`.
pub struct Collator<Block: BlockT, PF, BI> {
    proposer_factory: Arc<Mutex<PF>>,
    _phantom: PhantomData<Block>,
    inherent_data_providers: InherentDataProviders,
    collator_network: Arc<dyn CollatorNetwork>,
    block_import: Arc<Mutex<BI>>,
    wait_to_announce: Arc<Mutex<WaitToAnnounce<Block>>>,
}
```

1. 平行链阶段

- CollatorBuilder.build@cumulus/collator/src/lib.rs
- 366行，。。。。
- 372行，。。。。
- 380行，。。。。
- 384行，生成Collator实例并返回

```rust
348     fn build<PClient, Spawner, Extrinsic>(
349         self,
350         polkadot_client: Arc<PClient>,
351         spawner: Spawner,
352         polkadot_network: impl CollatorNetwork + Clone + 'static,
353     ) -> Result<Self::ParachainContext, ()>
354     where
355         PClient: ProvideRuntimeApi<PBlock>
356             + BlockchainEvents<PBlock>
357             + HeaderBackend<PBlock>
358             + Send
359             + Sync
360             + 'static,
361         PClient::Api: RuntimeApiCollection<Extrinsic>,
362         <PClient::Api as ApiExt<PBlock>>::StateBackend: StateBackend<HashFor<PBlock>>,
363         Spawner: Spawn + Clone + Send + Sync + 'static,
364         Extrinsic: codec::Codec + Send + Sync + 'static,
365     {
366         self.delayed_block_announce_validator
367             .set(Box::new(JustifiedBlockAnnounceValidator::new(
368                 polkadot_client.clone(),
369                 self.para_id,
370             )));
371
372         let follow =
373             match cumulus_consensus::follow_polkadot(self.para_id, self.client, polkadot_client) {
374                 Ok(follow) => follow,
375                 Err(e) => {
376                     return Err(error!("Could not start following polkadot: {:?}", e));
377                 }
378             };
379
380         spawner
381             .spawn_obj(Box::new(follow.map(|_| ())).into())
382             .map_err(|_| error!("Could not spawn parachain server!"))?;
383
384         Ok(Collator::new(
385             self.proposer_factory,
386             self.inherent_data_providers,
387             polkadot_network,
388             self.block_import,
389             Arc::new(spawner),
390             self.announce_block,
391         ))
392     }
393 }
```

1. 平行链阶段

- produce_candidate@cumulus/collator/src/lib.rs
- 这是produce_candidate的第一部分
- 186行，获得最新区块头
- 194行，基于最新区块头初始化提议者（Proposer）
- 199行，等待初始化完成
- 208行，获取inherent data，包括babe、grandpa、timestamp、authorship等数据
- 215行，调用proposer的propose方法来产生区块，具体实现在substrate的sc-basic-authorship中，这里

```rust
fn produce_candidate(
    &mut self,
    relay_chain_parent: PHash,
    global_validation: GlobalValidationSchedule,
    local_validation: LocalValidationData,
    downward_messages: Vec<DownwardMessage>,
) -> Self::ProduceCandidate {
    let factory = self.proposer_factory.clone();
    let inherent_providers = self.inherent_data_providers.clone();
    let block_import = self.block_import.clone();

    trace!(target: "cumulus-collator", "Producing candidate");

    let last_head = match HeadData::<Block>::decode(&mut &local_validation.parent_head.0[..]) {
        Ok(x) => x,
        Err(e) => {
            error!(target: "cumulus-collator", "Could not decode the head data: {:?}", e);
            return Box::pin(future::ready(Err(InvalidHead)));
        }
    };

    let proposer_future = factory.lock().init(&last_head.header);

    let wait_to_announce = self.wait_to_announce.clone();

    Box::pin(async move {
        let proposer = proposer_future.await.map_err(|e| {
            error!(
                target: "cumulus-collator",
                "Could not create proposer: {:?}",
                e,
            );
            InvalidHead
        })?;

        let inherent_data = Self::inherent_data(
            inherent_providers,
            global_validation,
            local_validation,
            downward_messages,
        )?;

        let Proposal {
            block,
            storage_changes,
            proof,
        } = proposer
            .propose(
                inherent_data,
                Default::default(),
                //TODO: Fix this.
                Duration::from_secs(6),
                RecordProof::Yes,
            )
            .await
            .map_err(|e| {
                error!(
                    target: "cumulus-collator",
                    "Proposing failed: {:?}",
                    e,
                );
                InvalidHead
            })?;
```

1. 平行链阶段

- produce_candidate@cumulus/collator/src/lib.rs
- 这是produce_candidate的第二部分
- 237行，产出了区块后，同时获得proof数据
- 248行，创建ParachainBlockData
- 259行，做block import的操作
- 272-278行，生成block candidate
- 280行，在parachain上广播该区块

```
237             let proof = proof.ok_or_else(|| {
238                 error!(
239                     target: "cumulus-collator",
240                     "Proposer did not return the requested proof.",
241                 );
242                 InvalidHead
243             })?;
244
245             let (header, extrinsics) = block.deconstruct();
246
247             // Create the parachain block data for the validators.
248             let b = ParachainBlockData::<Block>::new(
249                 header.clone(),
250                 extrinsics,
251                 proof.iter_nodes().collect(),
252             );
253
254             let mut block_import_params = BlockImportParams::new(BlockOrigin::Own, header);
255             block_import_params.body = Some(b.extrinsics().to_vec());
256             block_import_params.fork_choice = Some(ForkChoiceStrategy::LongestChain);
257             block_import_params.storage_changes = Some(storage_changes);
258
259             if let Err(err) = block_import
260                 .lock()
261                 .import_block(block_import_params, Default::default())
262             {
263                 error!(
264                     target: "cumulus-collator",
265                     "Error importing build block (at {:?}): {:?}",
266                     b.header().parent_hash(),
267                     err,
268                 );
269                 return Err(InvalidHead);
270             }
271
272             let block_data = BlockData(b.encode());
273             let header = b.into_header();
274             let encoded_header = header.encode();
275             let hash = header.hash();
276             let head_data = HeadData::<Block> { header };
277
278             let candidate = (block_data, parachain::HeadData(head_data.encode()));
279
280             wait_to_announce
281                 .lock()
282                 .wait_to_announce(hash, relay_chain_parent, encoded_header);
283
284             trace!(target: "cumulus-collator", "Produced candidate: {:?}", candidate);
285
286             Ok(candidate)
287         })
288     }
289 }
```

1. 平行链阶段

- 以上，我们整理了：
- collator的启动流程，从cumulus到polkadot
- collator包括了三个服务：PC node、RC node、collator service
- 区块如何产生的，由collator key签名
- 区块广播到parachain，network.annouce_block，代码在这里
- 区块广播到relaychain，network.distribute_collation，代码在这里和这里

2. 中继链阶段 - 验证服务

- 第二部分，我们来看看relay chain的验证服务部分
- 验证服务实例定义在这里：
- ParachainValidationInstances@polkadot/validation/src/validation_service/mod.rs
- client，是访问区块链服务的组件
- network，网络服务组件
- spawner：可以启动异步服务的handle
- live_instances：一个map，relay chain hash关联到验证实例
- validation_pool：验证hosts，用来提供实际验证的worker
- collation_fetch：获取collation的组件

```rust
/// Constructs parachain-agreement instances.
pub(crate) struct ParachainValidationInstances<N: Network, P, SP, CF> {
    /// The client instance.
    client: Arc<P>,
    /// The backing network handle.
    network: N,
    /// handle to spawner
    spawner: SP,
    /// Store for extrinsic data.
    availability_store: AvailabilityStore,
    /// Live agreements. Maps relay chain parent hashes to attestation
    /// instances.
    live_instances: HashMap<Hash, LiveInstance<N::TableRouter>>,
    /// The underlying validation pool of processes to use.
    /// Only `None` in tests.
    validation_pool: Option<ValidationPool>,
    /// Used to fetch a collation.
    collation_fetch: CF,
}
```

```rust
/// A live instance that is related to a relay chain validation round.
///
/// It stores the `instance_handle` and the `_table_router`.
struct LiveInstance<TR> {
    instance_handle: ValidationInstanceHandle,
    /// Make sure we keep the table router alive, to respond/receive consensus messages.
    _table_router: TR,
}
```

2. 中继链阶段 - 验证服务

- ParachainValidationInstances.get_or_instantiate@polkadot/validation/validation_service/mod.rs
- 验证服务的初始化，第一部分
- 359行，如果有实例则返回，注意这里构建的实例，是和parent_hash相关联的，每一个共识轮次，都会做一次初始化
- 365行，**从runtime api中得到validators数据**
- 366行，拿到本地keystore中可以给validators签名的key
- 368行，从runtime api拿到validators的任务表，任务是：relay chain / Parachain(ID)
- 370行，**通过validators、duty_roster计算分组信息，以及local_duty**
- 382、384行，从runtime拿到parachains信息
- 403行，验证runtime api的版本信息，并得到signing_context

```
351     async fn get_or_instantiate(
352         &mut self,
353         parent_hash: Hash,
354         keystore: &KeyStorePtr,
355         max_block_data_size: Option<u64>,
356     ) -> Result<ValidationInstanceHandle, Error> {
357         use primitives::Pair;
358
359         if let Some(instance : &LiveInstance<<...>::TableRouter> ) = self.live_instances.get( k: &parent_hash) {
360             return Ok(instance.instance_handle.clone());
361         }
362
363         let id : BlockId<Block> = BlockId::hash(parent_hash);
364
365         let validators = self.client.runtime_api().validators(&id)?;
366         let sign_with : Option<Arc<?>> = signing_key( validators: &validators[..], keystore);
367
368         let duty_roster = self.client.runtime_api().duty_roster(&id)?;
369
370         let (group_info : HashMap<Id, GroupInfo> , local_duty : Option<LocalDuty> ) = crate::make_group_info(
371             duty_roster,
372             authorities: &validators,
373             local_id: sign_with.as_ref().map( f: |k : &Arc<?> | k.public()),
374         )?;
375
376         info!(
377             "Starting parachain attestation session on top of parent {:?}. Local parachain duty is {:?}",
378             parent_hash,
379             local_duty,
380         );
381
382         let active_parachains = self.client.runtime_api().active_parachains(&id)?;
383
384         debug!(target: "validation", "Active parachains: {:?}", active_parachains);
385
386         // If we are a validator, we need to store our index in this round in availability store.
387         // This will tell which erasure chunk we should store.
388         if let Some(ref local_duty : &LocalDuty ) = local_duty {...}
400
401         let api : ApiRef<<...>::Api> = self.client.runtime_api();
402
403         let signing_context : SigningContext = if api.has_api_with::<dyn ParachainHost<Block, Error = ()>, _>(
404             at: &BlockId::hash(parent_hash),
405             pred: |version : u32 | version >= 3,
406         )? {
407             api.signing_context(&id)?
408         } else {
409             trace!(
410                 target: "validation",
411                 "Expected runtime with ParachainHost version >= 3",
412             );
413             SigningContext {
414                 session_index: 0,
415                 parent_hash,
416             }
417         };
```

2. 中继链阶段 - 验证服务

- ParachainValidationInstances.get_or_instantiate@polkadot/validation/validation_service/mod.rs
- 验证服务的初始化，第二部分
- 419行，构建SharedTable，当前共识round中，需要的各种数据，包括：验证者、分组、Key、context、availability_store？？？？？、validation_pool
- 431行，构建当前共识轮次中的Router，router负责将callation数据传送给network层，并进行后续广播
- 442-457行，spawn出一个任务**launch_work**，从collator获取collation数据，设置data到availability_store，并将collation广播给其它validator
- 451行，提供**collation_fetch** closure，在launch_work会回调
- 464行，构建一个tracker，让调用者能够访问table的数据，跟踪到进度
- 469-473行，构建一个LiveInstance，和parent_hash关联，并添加到map中

```
419         let table : Arc<SharedTable> = Arc::new( data: SharedTable::new(
420             validators: validators.clone(),
421             groups: group_info,
422             key: sign_with,
423             signing_context,
424             availability_store: self.availability_store.clone(),
425             max_block_data_size,
426             validation_pool: self.validation_pool.clone(),
427         ));
428
429         // The router will join the consensus gossip network. This is important
430         // to receive messages sent for the current round.
431         let router = match self.network.build_table_router(
432             table: table.clone(),
433             authorities: &validators,
434         ).await {
435             Ok(res) => res,
436             Err(e) => {
437                 warn!(target: "validation", "Failed to build router: {:?}", e);
438                 return Err(Error::CouldNotBuildTableRouter(format!("{:?}", e)))
439             }
440         };
441
442         if let Some((Chain::Parachain(id : Id ), index : u32 )) = local_duty.map( f: |d : LocalDuty | (d.
443             let n_validators = validators.len();
444             let availability_store : Store = self.availability_store.clone();
445             let client : Arc<P> = self.client.clone();
446             let collation_fetch : CF = self.collation_fetch.clone();
447             let router = router.clone();
448
449             let res : Result<(), SpawnError> = self.spawner.spawn(
450                 future: launch_work(
451                     collation_fetch: move || collation_fetch.collation_fetch( parachain: id, relay_parent:
452                     availability_store,
453                     router,
454                     n_validators,
455                     local_id: index,
456                 ),
457             );
458
459             if let Err(e : SpawnError ) = res {
460                 error!(target: "validation", "Failed to launch work: {:?}", e);
461             }
462         }
463
464         let tracker = ValidationInstanceHandle {
465             table,
466             started: Instant::now(),
467         };
468
469         let live_instance = LiveInstance {
470             instance_handle: tracker.clone(),
471             _table_router: router,
472         };
473         self.live_instances.insert( k: parent_hash, v: live_instance);
```

2. 中继链阶段 - 验证服务

- 定义、实现、调用runtime api的方式
- decl_runtime_apis@polkadot/primitives/src/parachain.rs
- impl_runtime_apis@polkadot/runtime/polkadot/src/lib.rs

```
363         let id : BlockId<Block> = BlockId::hash(parent_hash);
364
365         let validators = self.client.runtime_api().validators(&id)?;
366         let sign_with : Option<Arc<?>> = signing_key( validators: &validators[..], keystore);
367
368         let duty_roster = self.client.runtime_api().duty_roster(&id)?;
```

```
669 sp_api::decl_runtime_apis! {
670     /// The API for querying the state of parachains on-chain.
671     #[api_version(3)]
672     pub trait ParachainHost {
673         /// Get the current validators.
674         fn validators() -> Vec<ValidatorId>;
675         /// Get the current duty roster.
676         fn duty_roster() -> DutyRoster;
677         /// Get the currently active parachains.
678         fn active_parachains() -> Vec<(Id, Option<(CollatorId, Retriable)>)>;
679         /// Get the global validation schedule that all parachains should
680         /// be validated under.
```

```
1007     impl parachain::ParachainHost<Block> for Runtime {
1008         fn validators() -> Vec<parachain::ValidatorId> {
1009             Parachains::authorities()
1010         }
1011         fn duty_roster() -> parachain::DutyRoster {
1012             Parachains::calculate_duty_roster().0
1013         }
1014         fn active_parachains() -> Vec<(parachain::Id, Option<(parachain::CollatorId, parachain::Retr
1015             Registrar::active_paras()
1016         }
1017         fn global_validation_schedule() -> parachain::GlobalValidationSchedule {
1018             Parachains::global_validation_schedule()
1019         }
```

2. 中继链阶段 - 验证服务

- make_group_info@polkadot/validation/src/lib.rs
- 核心逻辑，180-195行，遍历roster、authorities，更新map，得到每个parachain分配的validators
- 同时，如果当前节点V是验证节点，那么计算出V的当前轮次的工作：LocalDuty
- 197-200行，计算每一个组需要的最小确认数，超过一半

```rust
/// Compute group info out of a duty roster and a local authority set.
pub fn make_group_info(
    roster: DutyRoster,
    authorities: &[ValidatorId],
    local_id: Option<ValidatorId>,
) -> Result<(HashMap<ParaId, GroupInfo>, Option<LocalDuty>), Error> {
    if roster.validator_duty.len() != authorities.len() {
        return Err(Error::InvalidDutyRosterLength {
            expected: authorities.len(),
            got: roster.validator_duty.len()
        });
    }

    let mut local_validation = None;
    let mut local_index : usize = 0;
    let mut map : HashMap<Id, GroupInfo> = HashMap::new();

    let duty_iter : Zip<Iter<?>, Iter<Chain>> = authorities.iter().zip( other: &roster.validator_duty);
    for (i : usize , (authority : &? , v_duty : &Chain )) in duty_iter.enumerate() {
        if Some(authority) == local_id.as_ref() {
            local_validation = Some(v_duty.clone());
            local_index = i;
        }

        match *v_duty {
            Chain::Relay => {}, // does nothing for now.
            Chain::Parachain(ref id : &Id ) => {
                map.entry( key: id.clone()).or_insert_with(GroupInfo::default)
                    .validity_guarantors
                    .insert( value: authority.clone());
            }
        }
    }

    for live_group : &mut GroupInfo in map.values_mut() {
        let validity_len : usize = live_group.validity_guarantors.len();
        live_group.needed_validity = validity_len / 2 + validity_len % 2;
    }


    let local_duty : Option<LocalDuty> = local_validation.map( f: |v : Chain | LocalDuty {
        validation: v,
        index: local_index as u32,
    });

    Ok((map, local_duty))
}
```

2. 中继链阶段 - 验证服务

- launch_work@polkadot/validation/validation_service/mod.rs
- 496行，通过collation_fetch从collators拿到一个Collation，**对collation的验证工作，也在这里做的**
- 504-509行，解包输出的数据，得到承诺commitments、纠删码erasure_chunks、验证数据available_data
- 511行，将commitments添加到collation_info得到receipt
- 512行，得到pov_block数据，区块数据
- 514行，将收据、available_data存入store，这里这个store是一个kv-db
- 525行，将纠删码数据存入store
- 533行，**将collation发送给其它validators**，在此之前，会对发送数据进行签名，并将其保存到shared_table

```
485  async fn launch_work<CFF, E>(
486      collation_fetch: impl FnOnce() -> CFF,
487      availability_store: AvailabilityStore,
488      router: impl TableRouter,
489      n_validators: usize,
490      local_id: ValidatorIndex,
491  ) where
492      E: std::fmt::Debug,
493      CFF: Future<Output = Result<(CollationInfo, FullOutput), E>> + Send,
494  {
495      // fetch a local collation from connected collators.
496      let (collation_info : CollationInfo , full_output : FullOutput ) = match collation_fetch().await {
497          Ok(res : (CollationInfo, FullOutput) ) => res,
498          Err(e : E ) => {
499              warn!(target: "validation", "Failed to collate candidate: {:?}", e);
500              return
501          }
502      };
503
504      let crate::pipeline::FullOutput {
505          commitments : CandidateCommitments ,
506          erasure_chunks : Vec<ErasureChunk> ,
507          available_data : AvailableData ,
508          ..
509      } = full_output;
510
511      let receipt : AbridgedCandidateReceipt = collation_info.into_receipt(commitments);
512      let pov_block : PoVBlock = available_data.pov_block.clone();
513
514      if let Err(e : Error ) = availability_store.make_available(
515          candidate_hash: receipt.hash(),
516          available_data,
517      ).await {
518          warn!(
519              target: "validation",
520              "Failed to make parachain block data available: {}",
521              e,
522          );
523      }
524
525      if let Err(e : Error ) = availability_store.clone().add_erasure_chunks(
526          candidate: receipt.clone(),
527          n_validators: n_validators as _,
528          chunks: erasure_chunks.clone(),
529      ).await {
530          warn!(target: "validation", "Failed to add erasure chunks: {}", e);
531      }
532
533      if let Err(e) = router.local_collation(
534          receipt,
535          pov_block,
536          chunks: (local_id, &erasure_chunks),
537      ).await {
538          warn!(target: "validation", "Failed to send local collation: {:?}", e);
539      }
540  }
```

2. 中继链阶段 - 验证服务

- 保存collation的数据结构
- SlotEntries@polkadot/network/src/legacy/collator_pool.rs
- SlotEntries枚举结构，通过定义了Blank、Pending、Awaiting三种值，以及当发生收到collation、添加接收人等事件时的状态流转，把collation和receiver连接了起来

```rust
#[derive(Debug)]
enum SlotEntries {
    Blank,
    // not queried yet
    Pending(Vec<Collation>),
    // waiting for next to arrive.
    Awaiting(Vec<oneshot::Sender<Collation>>),
}

impl SlotEntries {
    fn received_collation(&mut self, collation: Collation) {
        *self = match std::mem::replace(self, SlotEntries::Blank) {
            SlotEntries::Blank => SlotEntries::Pending(vec![collation]),
            SlotEntries::Pending(mut cs) => {
                cs.push(collation);
                SlotEntries::Pending(cs)
            }
            SlotEntries::Awaiting(senders) => {
                for sender in senders {
                    let _ = sender.send(collation.clone());
                }

                SlotEntries::Blank
            }
        };
    }

    fn await_with(&mut self, sender: oneshot::Sender<Collation>) {
        *self = match ::std::mem::replace(self, SlotEntries::Blank) {
            SlotEntries::Blank => SlotEntries::Awaiting(vec![sender]),
            SlotEntries::Awaiting(mut senders) => {
                senders.push(sender);
                SlotEntries::Awaiting(senders)
            }
            SlotEntries::Pending(mut cs) => {
                let next_collation = cs.pop().expect("empty variant is always `Blank`; qed");
                let _ = sender.send(next_collation);

                if cs.is_empty() {
                    SlotEntries::Blank
                } else {
                    SlotEntries::Pending(cs)
                }
            }
        };
    }
}
```

2. 中继链阶段 - 验证服务

- 前文讲到launch_work会调用router的local_collation，该函数最终会调用到distribute_validated_collation
- 1312-1320行，构建签名的statement并p2p广播
- 1324-1335行，构建区块数据pov_block并p2p广播
- 1338-1348行，构建纠删码chunk数据并p2p广播

```
1296  fn distribute_validated_collation(
1297      instance: &ConsensusNetworkingInstance,
1298      receipt: AbridgedCandidateReceipt,
1299      pov_block: PoVBlock,
1300      chunks: (ValidatorIndex, Vec<ErasureChunk>),
1301      gossip_handle: &impl GossipOps,
1302  ) {
1303      // produce a signed statement.
1304      let hash : Hash = receipt.hash();
1305      let validated : Validated = Validated::collated_local(
1306          receipt,
1307          collation: pov_block.clone(),
1308      );
1309
1310      // gossip the signed statement.
1311      {
1312          let statement : GossipStatement = crate::legacy::gossip::GossipStatement::new(
1313              relay_chain_leaf: instance.relay_parent,
1314              signed_statement: match instance.statement_table.import_validated(validated) {
1315                  None => return,
1316                  Some(s : SignedStatement ) => s,
1317              }
1318          );
1319
1320          gossip_handle.gossip_message(instance.attestation_topic, message: statement.into());
1321      }
1322
1323      // gossip the PoV block.
1324      {
1325          let pov_block_message = crate::legacy::gossip::GossipPoVBlock {
1326              relay_chain_leaf: instance.relay_parent,
1327              candidate_hash: hash,
1328              pov_block,
1329          };
1330
1331          gossip_handle.gossip_message(
1332              topic: crate::legacy::gossip::pov_block_topic( parent_hash: instance.relay_parent),
1333              message: pov_block_message.into(),
1334          );
1335      }
1336
1337      // gossip erasure chunks.
1338      for chunk : ErasureChunk in chunks.1 {
1339          let message = crate::legacy::gossip::ErasureChunkMessage {
1340              chunk,
1341              candidate_hash: hash,
1342          };
1343
1344          gossip_handle.gossip_message(
1345              topic: crate::erasure_coding_topic( candidate_hash: &hash),
1346              message: message.into(),
1347          );
1348      }
1349  }
```

2. 中继链阶段 - 验证服务

- 在验证服务中，有使用collation_fetch，这里我们来看看
- collation_fetch@polkadot/validation/src/collation.rs
- 函数主体是一个死循环loop，直到获取到正确的数据后才会返回
- 77行，通过collators.collate获得collation；这里最终也是使用前面讲述的SlotEntries机制来获取collation
- 79行，使用pipeline组件，对collation进行验证
- 91行，返回正确的数据

```
60      /// A future which resolves when a collation is available.
61      pub async fn collation_fetch<C: Collators, P>(
62          validation_pool: Option<crate::pipeline::ValidationPool>,
63          parachain: ParaId,
64          relay_parent: Hash,
65          collators: C,
66          client: Arc<P>,
67          max_block_data_size: Option<u64>,
68          n_validators: usize,
69      ) -> Result<(CollationInfo, crate::pipeline::FullOutput), C::Error>
70          where
71              P::Api: ParachainHost<Block, Error = sp_blockchain::Error>,
72              C: Collators + Unpin,
73              P: ProvideRuntimeApi<Block>,
74              <C as Collators>::Collation: Unpin,
75      {
76          loop {
77              let collation = collators.collate(parachain, relay_parent).await?;
78              let Collation { info, pov } = collation;
79              let res = crate::pipeline::full_output_validation_with_api(
80                  validation_pool.as_ref(),
81                  &*client,
82                  &info,
83                  &pov,
84                  &relay_parent,
85                  max_block_data_size,
86                  n_validators,
87              );
88
89              match res {
90                  Ok(full_output) => {
91                      return Ok((info, full_output))
92                  }
93                  Err(e) => {
94                      debug!("Failed to validate parachain due to API error: {}", e);
95
96                      // just continue if we got a bad collation or failed to validate
97                      collators.note_bad_collator(info.collator)
98                  }
99              }
100         }
101     }
```

2. 中继链阶段 - 验证服务

- 在pipeline组件中，核心入口是validate函数
- 217行，执行mode，如果有worker则使用spawn模式，否则是in-process模式
- 221行，调用wasm_executor的validate_candiate函数进行验证，传入的核心数据为：
  - parachain的STF的代码code
  - 参数：parent_head，height，block_data
  - 执行mode
- 验证结束后，227行，比对validator的验证结果，和collator传过来的结果，也就是区块头，是否一致
- 一切正常的话，234行，返回一个ValidatedCandidate数据

```rust
187  pub fn validate<'a>(
188      validation_pool: Option<&'_ ValidationPool>,
189      collation: &'a CollationInfo,
190      pov_block: &'a PoVBlock,
191      local_validation: &'a LocalValidationData,
192      global_validation: &'a GlobalValidationSchedule,
193      validation_code: &ValidationCode,
194  ) -> Result<ValidatedCandidate<'a>, Error> {
195      if collation.head_data.0.len() > global_validation.max_head_data_size as _ {
196          return Err(Error::HeadDataTooLarge(
197              collation.head_data.0.len(),
198              global_validation.max_head_data_size as _,
199          ));
200      }
201
202      let params = ValidationParams {
203          parent_head: local_validation.parent_head.clone(),
204          block_data: pov_block.block_data.clone(),
205          max_code_size: global_validation.max_code_size,
206          max_head_data_size: global_validation.max_head_data_size,
207          relay_chain_height: global_validation.block_number,
208          code_upgrade_allowed: local_validation.code_upgrade_allowed,
209      };
210
211      // TODO: remove when ext does not do this.
212      let fee_schedule = FeeSchedule {
213          base: 0,
214          per_byte: 0,
215      };
216
217      let execution_mode = validation_pool
218          .map(ExecutionMode::Remote)
219          .unwrap_or(ExecutionMode::Local);
220
221      match wasm_executor::validate_candidate(
222          &validation_code.0,
223          params,
224          execution_mode,
225      ) {
226          Ok(result) => {
227              if result.head_data == collation.head_data {
228                  let fees = validate_upward_messages(
229                      &result.upward_messages,
230                      fee_schedule,
231                      local_validation.balance,
232                  )?;
233
234                  Ok(ValidatedCandidate {
235                      pov_block,
236                      global_validation,
237                      local_validation,
238                      upward_messages: result.upward_messages,
239                      fees,
240                      processed_downward_messages: result.processed_downward_messages,
241                  })
242              } else {
243                  Err(Error::HeadDataMismatch)
244              }
245          }
246          Err(e) => Err(e.into()),
247      }
```

2. 中继链阶段 - 验证服务

- validate_candidate@polkadot/parachain/src/wasm_executor/validation_host.rs
- 验证实例池validate_pool中有n（8）个验证实例ValidationHost，每个host负责验证一个parachain的block
- validate_candidate是验证逻辑的入口
- 244行，启动一个worker，实际验证的工作，是在worker中做的
- host与work之间，是通过shared mem的方式传送数据的
- 245-267行，host将各种数据copy到shared mem中，包括：区块头、STF的code、params等
- 270行，发送ready事件给worker
- 273行，等待worker执行结束
- 291行，返回最终的结果，主要内容是新区块的区块头数据

```rust
234     pub fn validate_candidate(
235         &mut self,
236         validation_code: &[u8],
237         params: ValidationParams,
238         test_mode: bool,
239     ) -> Result<ValidationResult, Error> {
240         if validation_code.len() > MAX_CODE_MEM {
241             return Err(Error::CodeTooLarge(validation_code.len()));
242         }
243         // First, check if need to spawn the child process
244         self.start_worker(test_mode)?;
245         let memory = self.memory.as_mut()
246             .expect("memory is always `Some` after `start_worker` completes successfully");
247         {
248             // Put data in shared mem
249             let data: &mut[u8] = &mut **memory.wlock_as_slice(0)?;
250             let (mut header_buf, rest) = data.split_at_mut(1024);
251             let (code, rest) = rest.split_at_mut(MAX_CODE_MEM);
252             let (code, _) = code.split_at_mut(validation_code.len());
253             let (call_data, _) = rest.split_at_mut(MAX_RUNTIME_MEM);
254             code[..validation_code.len()].copy_from_slice(validation_code);
255             let encoded_params = params.encode();
256             if encoded_params.len() > MAX_RUNTIME_MEM {
257                 return Err(Error::ParamsTooLarge(MAX_RUNTIME_MEM));
258             }
259             call_data[..encoded_params.len()].copy_from_slice(&encoded_params);
260
261             let header = ValidationHeader {
262                 code_size: validation_code.len() as u64,
263                 params_size: encoded_params.len() as u64,
264             };
265
266             header.encode_to(&mut header_buf);
267         }
268
269         debug!("{} Signaling candidate", self.id);
270         memory.set(Event::CandidateReady as usize, EventState::Signaled)?;
271
272         debug!("{} Waiting for results", self.id);
273         match memory.wait(Event::ResultReady as usize, shared_memory::Timeout::Sec(EXECUTION_TIMEOUT
274             Err(e) => {...}
281             Ok(()) => {}
282         }
283
284         {
285             debug!("{} Reading results", self.id);
286             let data: &[u8] = &**memory.wlock_as_slice(0)?;
287             let (header_buf, _) = data.split_at(1024);
288             let mut header_buf: &[u8] = header_buf;
289             let header = ValidationResultHeader::decode(&mut header_buf).unwrap();
290             match header {
291                 ValidationResultHeader::Ok(result) => Ok(result),
292                 ValidationResultHeader::Error(message) => {
293                     debug!("{} Validation error: {}", self.id, message);
294                     Err(Error::External(message).into())
295                 }
296             }
297         }
298     }
```

2. 中继链阶段 - 验证服务

- start_worker@polkadot/parachain/src/wasm_executor/validation_host.rs
- 在前面一步validate_candidate中，会调用start_worker启动一个worker
- 211行，创建shared mem
- 212行，得到当前执行程序的path
- 215行，再次启动当前程序，单独的一个新进程，入参传入“WORKS_ARGS”，同时传入shared mem的file path，/dev/shm
- 223行，等待run_worker中的WorkerReady的事件
- 返回前，worker与memory都已经ready

```rust
impl ValidationHost {
    fn create_memory() -> Result<SharedMem, Error> {
        let mem_size = MAX_RUNTIME_MEM + MAX_CODE_MEM + 1024;
        let mem_config = SharedMemConf::default()
            .set_size(mem_size)
            .add_lock(shared_memory::LockType::Mutex, 0, mem_size)?
            .add_event(shared_memory::EventType::Auto)?  // Event::CandidateReady
            .add_event(shared_memory::EventType::Auto)?  // Event::ResultReady
            .add_event(shared_memory::EventType::Auto)?; // Event::WorkerReady

        Ok(mem_config.create()?)
    }

    fn start_worker(&mut self, test_mode: bool) -> Result<(), Error> {
        if let Some(ref mut worker) = self.worker {
            // Check if still alive
            if let Ok(None) = worker.try_wait() {
                // Still running
                return Ok(());
            }
        }
        let memory = Self::create_memory()?;
        let self_path = env::current_exe()?;
        debug!("Starting worker at {:?}", self_path);
        let mut args = if test_mode { WORKER_ARGS_TEST.to_vec() } else { WORKER_ARGS.to_vec() };
        args.push(memory.get_os_path());
        let worker = process::Command::new(self_path)
            .args(args)
            .stdin(process::Stdio::piped())
            .spawn()?;
        self.id = worker.id();
        self.worker = Some(worker);

        memory.wait(
            Event::WorkerReady as usize,
            shared_memory::Timeout::Sec(EXECUTION_TIMEOUT_SEC as usize),
        )?;
        self.memory = Some(memory);
        Ok(())
    }
```

2. 中继链阶段 - 验证服务

- run_worker@polkadot/parachain/src/wasm_executor/validation_host.rs
- 86行，open shared mem
- 94-104行，设置一个退出机制
- 106行，发送WorkerReady事件
- 110行，判断是否退出
- 115行，等待CandidateReady事件
- 131-140行，准备数据，code 和 call_data
- 142行，基于code执行call_data
- 150-151行，设置执行结果到shared mem
- 执行code的逻辑，使用到了substrate中的WasmExecutor，这里就不介绍了

```rust
pub fn run_worker(mem_id: &str) -> Result<(), String> {
    let mut memory = match SharedMem::open(mem_id) {
        Ok(memory) => memory,
        Err(e) => {...}
    };

    let exit = Arc::new(atomic::AtomicBool::new(false));
    // spawn parent monitor thread
    let watch_exit = exit.clone();
    std::thread::spawn(move || {
        use std::io::Read;
        let mut in_data = Vec::new();
        // pipe terminates when parent process exits
        std::io::stdin().read_to_end(&mut in_data).ok();
        debug!("{} Parent process is dead. Exiting", process::id());
        exit.store(true, atomic::Ordering::Relaxed);
    });

    memory.set(Event::WorkerReady as usize, EventState::Signaled)
        .map_err(|e| format!("{} Error setting shared event: {:?}", process::id(), e))?;

    loop {
        if watch_exit.load(atomic::Ordering::Relaxed) {
            break;
        }

        debug!("{} Waiting for candidate", process::id());
        match memory.wait(Event::CandidateReady as usize, shared_memory::Timeout::Sec(3)) {...}

        {
            debug!("{} Processing candidate", process::id());
            // we have candidate data
            let mut slice = memory.wlock_as_slice(0)
                .map_err(|e| format!("Error locking shared memory: {:?}", e))?;

            let result = {
                let data: &mut[u8] = &mut **slice;
                let (header_buf, rest) = data.split_at_mut(1024);
                let mut header_buf: &[u8] = header_buf;
                let header = ValidationHeader::decode(&mut header_buf)
                    .map_err(|_| format!("Error decoding validation request."))?;
                debug!("{} Candidate header: {:?}", process::id(), header);
                let (code, rest) = rest.split_at_mut(MAX_CODE_MEM);
                let (code, _) = code.split_at_mut(header.code_size as usize);
                let (call_data, _) = rest.split_at_mut(MAX_RUNTIME_MEM);
                let (call_data, _) = call_data.split_at_mut(header.params_size as usize);

                let result = validate_candidate_internal(code, call_data);
                debug!("{} Candidate validated: {:?}", process::id(), result);

                match result {
                    Ok(r) => ValidationResultHeader::Ok(r),
                    Err(e) => ValidationResultHeader::Error(e.to_string()),
                }
            };
            let mut data: &mut[u8] = &mut **slice;
            result.encode_to(&mut data);
        }
        debug!("{} Signaling result", process::id());
        memory.set(Event::ResultReady as usize, EventState::Signaled)
            .map_err(|e| format!("Error setting shared event: {:?}", e))?;
```

2. 中继链阶段 - 共识服务

- @polkadot/network/src/protocol/mod.rs
- 共识服务，主要由两个组件构成，一个是worker工作组件，另一个是ProtocolHandler
- Woker组件负责从网络、其它上层服务接受事件，并传送给各个具体的执行方，ProtocolHander是执行方之一
- api: 通过api接入runtime获取共识需要的数据
- executor：使用执行器执行一个后台任务
- gossip_handle：从p2p网络中获取/发送消息
- background_to_main_sender、background_receiver：

```rust
struct Worker<Api, Sp, Gossip> {
    protocol_handler: ProtocolHandler,
    api: Arc<Api>,
    executor: Sp,
    gossip_handle: Gossip,
    background_to_main_sender: mpsc::Sender<BackgroundToWorkerMsg>,
    background_receiver: mpsc::Receiver<BackgroundToWorkerMsg>,
    service_receiver: mpsc::Receiver<ServiceToWorkerMsg>,
    consensus_networking_receivers: FuturesUnordered<StreamFuture<ConsensusNetworkingReceiver>>,
}
```

```rust
struct ProtocolHandler {
    service: Arc<dyn NetworkServiceOps>,
    peers: HashMap<PeerId, PeerData>,
    // reverse mapping from validator-ID to PeerID. Multiple peers can represent
    // the same validator because of sentry nodes.
    connected_validators: HashMap<ValidatorId, HashSet<PeerId>>,
    consensus_instances: HashMap<Hash, ConsensusNetworkingInstance>,
    collators: crate::legacy::collator_pool::CollatorPool,
    local_collations: crate::legacy::local_collations::LocalCollations<Collation>,
    config: Config,
    local_keys: RecentValidatorIds,
}
```

```rust
struct ConsensusNetworkingInstance {
    statement_table: Arc<SharedTable>,
    relay_parent: Hash,
    attestation_topic: Hash,
    _drop_signal: exit_future::Signal,
}
```

AnV 6-2 Relay chain submission phase

- 上一节讲到，在relaychain上将collation广播出去
- on_remote_collation@polkadot/network/src/protocol/mod.rs
- 该函数做了很多检查
- 658行，这个是关键，调用了self.collators.on_collation，把collation放到了collation pool中

```rust
struct ProtocolHandler {
    service: Arc<dyn NetworkServiceOps>,
    peers: HashMap<PeerId, PeerData>,
    // reverse mapping from validator-ID to PeerID. Multiple peers can represent
    // the same validator because of sentry nodes.
    connected_validators: HashMap<ValidatorId, HashSet<PeerId>>,
    consensus_instances: HashMap<Hash, ConsensusNetworkingInstance>,
    collators: crate::legacy::collator_pool::CollatorPool,
    local_collations: crate::legacy::local_collations::LocalCollations<Collation>,
    config: Config,
    local_keys: RecentValidatorIds,
}
```

```rust
fn on_remote_collation(&mut self, remote: PeerId, relay_parent: Hash, collation: Collation) {
    let peer : &mut PeerData = match self.peers.get_mut( k: &remote) {
        None => { self.service.report_peer( peer: remote, value: cost::UNKNOWN_PEER); return }
        Some(p : &mut PeerData ) => p,
    };

    let (collator_id, para_id : Id ) = match peer.ready_and_collating_for() {
        None => {
            self.service.report_peer( peer: remote, value: cost::UNEXPECTED_MESSAGE);
            return
        }
        Some(x : (?, Id) ) => x,
    };

    let collation_para : Id = collation.info.parachain_index;
    let collated_acc = collation.info.collator.clone();

    let structurally_valid : bool = para_id == collation_para && collator_id == collated_acc;
    if structurally_valid && collation.info.check_signature().is_ok() {
        debug!(target: "p_net", "Received collation for parachain {:?} from peer {}",
            para_id, remote);

        if self.collators.collator_id_to_peer_id(&collator_id) == Some(&remote) {
            self.collators.on_collation(collator_id, relay_parent, collation);
            self.service.report_peer( peer: remote, value: benefit::GOOD_COLLATION);
        }
    } else {
        self.service.report_peer( peer: remote, value: cost::INVALID_FORMAT);
    }
}
```

AnV 6-2 Relay chain submission phase

- CollationPool@polkadot/network/src/legacy/collation_pool.rs
- CollationPool保存了collator的map
- 保存了某个parachain的collators的map
- 保存了接收到的collation数据

```rust
struct ParachainCollators {
    primary: CollatorId,
    backup: Vec<CollatorId>,
}

/// Manages connected collators and role assignments from the perspecti
#[derive(Default)]
pub struct CollatorPool {
    collators: HashMap<CollatorId, (ParaId, PeerId)>,
    parachain_collators: HashMap<ParaId, ParachainCollators>,
    collations: HashMap<(Hash, ParaId), CollationSlot>,
}
```

```rust
struct CollationSlot {
    live_at: Instant,
    entries: SlotEntries,
}

impl CollationSlot {...}

#[derive(Debug)]
enum SlotEntries {
    Blank,
    // not queried yet
    Pending(Vec<Collation>),
    // waiting for next to arrive.
    Awaiting(Vec<oneshot::Sender<Collation>>),
}
```