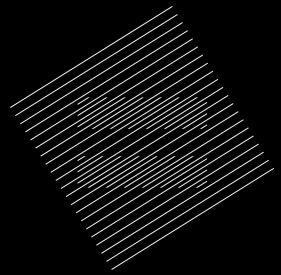
Substrate 中的 Rust 设计模式



孙凯超
Parity 工程师
@kaichaosun

Rust China Conf
2020

Overview

- 去中心应用的演进
- Substrate 的基本介绍
- 以及使用的常见 Rust 设计模式
- 技术生态
- 未来和展望

专用的区块链

Bitcoin 已经发展了10多年，

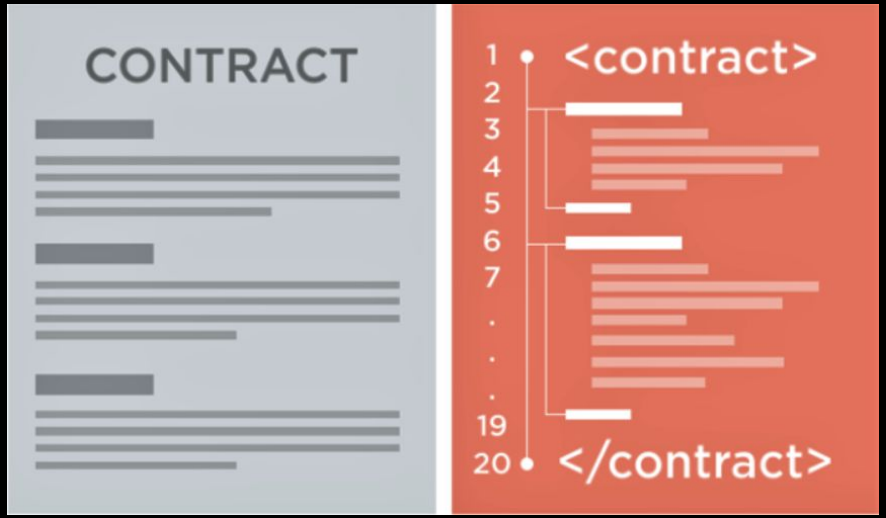
- 代表了早期去中心应用的探索和金融去中心的尝试
- 愿景还没完全达到，没有完全的去中心，全球支付的效率低
- 新时代的使命，价值存储
- 功能单一，创新成本高

图片来源:unsplash.com/@harrisonkugler

通用的区块链 - 智能合约

Ethereum 丰富了 Bitcoin 的脚本语言，运行在准图灵完备的虚拟机之上，

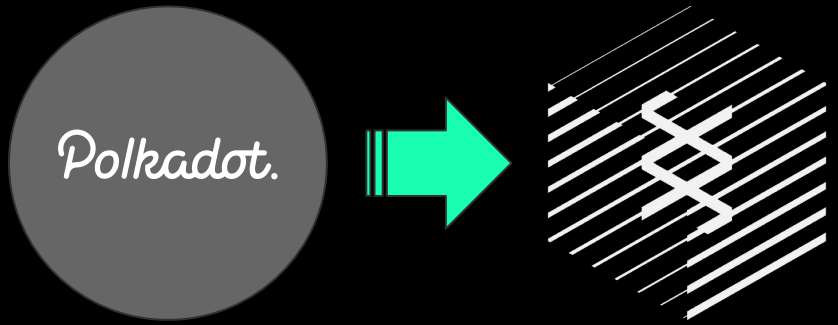
- 降低了去中心应用的创新成本
- 高可用，7*24 逻辑执行
- 无国界的协作成为可能
- 合约应用与平台绑定，如安全性隐患，成本高，用户体验差



如何更加通用化呢？

面向应用场景的区块链开发框架，

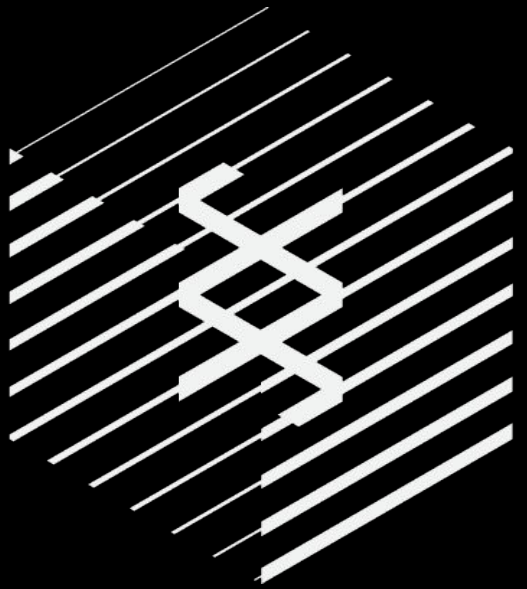
- 2017年，开始 Polkadot 跨链协议的设计
- 开发过程中抽离出 Substrate
- 快速开发一条应用链，针对业务场景进行优化，灵活可扩展
- 使用 Rust，WASM



Substrate 组件

开源、模块化、可扩展的区块链开发框架，涵盖了区块链的核心组件：

- 点对点网络传输和协议层
- 数据库层
- 共识协议
- 交易池
- Runtime 逻辑层
- ...



Substrate 组件

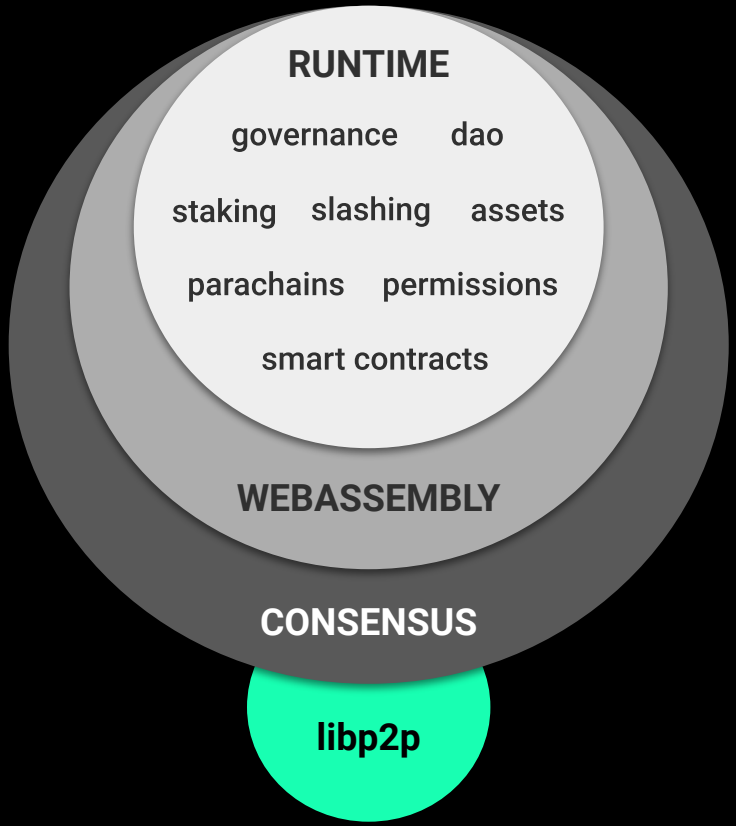
Rust

编译为Wasm

Wasm在链上存储

Rust

编译成本地可执行文件



Substrate 中 Rust 设计模式

Newtype 模式，

- 使用元组结构体封装数据
- 无需暴露内部数据的实现细节，向后兼容更容易
- 还可用于区分通用类型，新的类型，不易混淆
- zero-cost abstraction

```rust
pub struct OpaquePeerId(pub Vec<u8>);

impl OpaquePeerId {
    /// Create new `OpaquePeerId`
    pub fn new(vec: Vec<u8>) -> Self {
        OpaquePeerId(vec)
    }
}
```

Substrate 中 Rust 设计模式

Newtype 模式，

- 使用元组结构体封装数据
- 无需暴露内部数据的实现细节，向后兼容更容易
- 还可用于区分通用类型，新的类型，不易混淆
- zero-cost abstraction

```rust
pub struct OpaquePeerId(pub Vec<u8>);

impl OpaquePeerId {
    /// Create new `OpaquePeerId`
    pub fn new(vec: Vec<u8>) -> Self {
        OpaquePeerId(vec)
    }
}
```

Substrate 中 Rust 设计模式

Builder 模式，

- 用于构造复杂对象
- 将内部不同的属性进行了隔离
- 方便构造时校验和处理 side effect
- 链式调用

```rust
pub struct ValidTransaction {
    pub priority: TransactionPriority,
    pub requires: Vec<TransactionTag>,
    pub provides: Vec<TransactionTag>,
    pub longevity: TransactionLongevity,
    pub propagate: bool,
}
```

Substrate 中 Rust 设计模式

Builder 模式，

- 用于构造复杂对象
- 将内部不同的属性进行了隔离
- 方便构造时校验和处理 side effect
- 链式调用

```rust
pub struct ValidTransactionBuilder {
    prefix: Option<&'static str>,
    validity: ValidTransaction,
}

impl ValidTransactionBuilder {
    pub fn priority(mut self, priority: TransactionPriority) -> Self {
        self.validity.priority = priority;
        self
    }
    // -- snip --
    pub fn and_requires(mut self, tag: impl Encode) -> Self {
        self.validity.requires.push(match self.prefix.as_ref() {
            Some(prefix) => (prefix, tag).encode(),
            None => tag.encode(),
        });
        self
    }
    // --snip --
    pub fn build(self) -> ValidTransaction {
        self.validity
    }
}
```

Substrate 中 Rust 设计模式

Builder 模式，

- 用于构造复杂对象
- 将内部不同的属性进行了隔离
- 方便构造时校验和处理 side effect
- 链式调用

```
ValidTransaction::with_tag_prefix("StakingOffchain")
    .priority(T::UnsignedPriority::get().saturating_add(score[0].saturated_into()))
    .and_provides(era)
    .longevity(TryInto::<u64>::try_into(
            T::ElectionLookahead::get()).unwrap_or(DEFAULT_LONGEVITY)
    )
    .propagate(false)
    .build()
```

Substrate 中 Rust 设计模式

Macros 元编程，宏的定义方式有

- 声明式的宏
- 过程宏
  - 类函数
  - 派生
  - 属性式
- 应该使用哪种宏？

Substrate 中 Rust 设计模式

Macros 元编程

- 应该使用哪种宏？

	声明式宏	函数过程宏	属性宏
符合 Rust 标准语法	否	否	是
逻辑复杂时的可读性和维护性	差	较好	好
是否放在单独 crate	否	是	是

Substrate 中 Rust 设计模式

Macros 元编程，Substrate 定义的宏，

- 位于 frame_support crate
- decl_storage 定义存储单元（类函数）
- decl_module 包含可调用函数（声明式）
- decl_event 定义事件（声明式）
- decl_error 定义错误信息（声明式）
- construct_runtime 添加模块到 Runtime（类函数）
- FRAME v2，pallet（属性式）

Substrate Runtime 开发

使用**宏定义的 DSL**（Domain Specific Language）开发
一条承载特定业务的区块链应用，

- 最简单，也最普遍
- 开发时间缩短
- 代码更加简洁，提升效率
- 只需关注上层业务
- Rails, or Linux

Substrate Runtime 开发

以**存证应用**为例，在某一时间点验证数据文件的存在性和归属，最早是通过比特币网络带有时间戳的交易实现的。

```
decl_storage! {
    trait Store for Module<T: Trait> as PoeModule {
        Proofs get(fn proof): map hasher(twox_64_concat) Vec<u8> => (T::AccountId,
T::BlockNumber);
    }
}
```

Substrate Runtime 开发

```
decl_module! {
    pub struct Module<T: Trait> for enum Call where origin: T::Origin {
        // -- snip --

        #[weight = 0]
        pub fn create_claim(origin, claim: Vec<u8>) -> DispatchResult {
            let sender = ensure_signed(origin)?;

            ensure!(!Proofs::<T>::contains_key(&claim), Error::<T>::DuplicateClaim);

            Proofs::<T>::insert(&claim, (sender.clone(), system::Module::
<T>::block_number()));

            Self::deposit_event(RawEvent::ClaimCreated(sender, claim));

            Ok(())
        }

    }
}
```

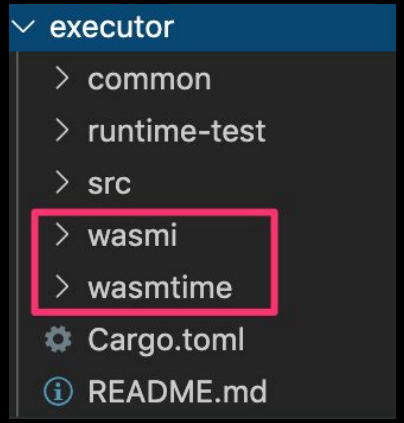
Substrate Runtime 开发

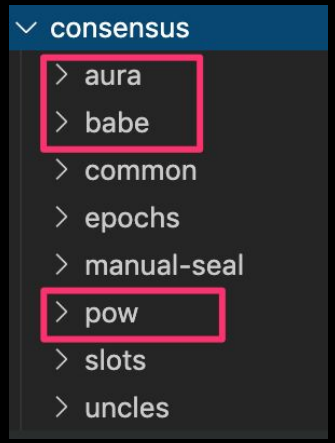
```
construct_runtime!(
    pub enum Runtime where
        Block = Block,
        NodeBlock = opaque::Block,
        UncheckedExtrinsic = UncheckedExtrinsic
    {
        // -- snip --
        // Include the custom logic from the template pallet in the runtime.
        PoeModule: poe::{Module, Call, Storage, Event<T>},
    }
);
```

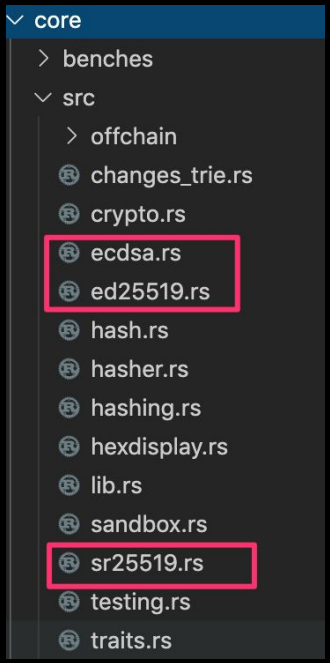
Substrate 中 Rust 设计模式

可复用模块和**接口抽象**，实现组件可插拔，

- P2P 通信协议
- 数据库
- WASM 虚拟机
- 共识协议
- 密码学







Substrate 共识协议开发

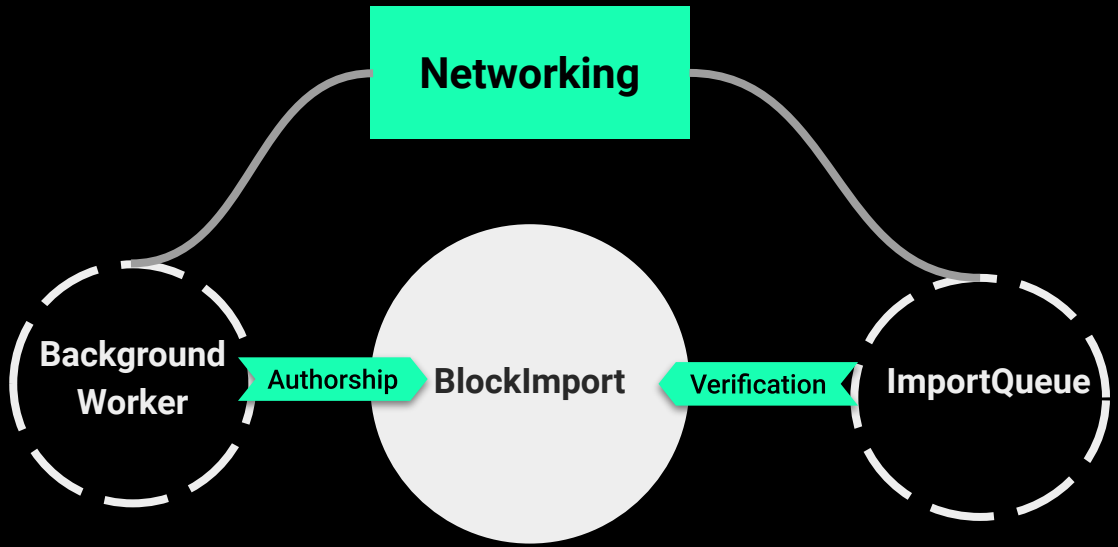
目前支持的共识有，

- 工作量证明（Proof of Work）
- 权威证明（Proof of Authority, Aura）
- 权益证明（Proof of Stake, Babe）
- 手动触发生成区块（Manual/instant seal）

Substrate 共识协议开发

区块生成算法的抽象，

- 区块生成
  sp_consensus_common::Proposer
- 区块导入
  sp_consensus_common::BlockImport
- 区块验证
  sp_consensus_common::import_queue::Verifier



Substrate 共识协议开发

```rust
pub trait Proposer<B: BlockT> {
    fn propose(
        self,
        inherent_data: InherentData,
        inherent_digests: DigestFor<B>,
        max_duration: Duration,
        record_proof: RecordProof,
    ) -> Self::Proposal;
}
```

```rust
pub trait BlockImport<B: BlockT> {
    fn import_block(
        &mut self,
        block: BlockImportParams<B, Self::Transaction>,
        cache: HashMap<CacheKeyId, Vec<u8>>,
    ) -> Result<ImportResult, Self::Error>;
}
```

```rust
pub trait Verifier<B: BlockT>: Send + Sync {
    fn verify(
        &mut self,
        origin: BlockOrigin,
        header: B::Header,
        justification: Option<Justification>,
        body: Option<Vec<B::Extrinsic>>,
    ) -> Result<(BlockImportParams<B, ()>, Option<Vec<(CacheKeyId, Vec<u8>)>>), String>;
}
```

Substrate 共识协议开发

PoW 共识，

- Mining worker 使用并生产区块
- 在多个线程里分别启动

```rust
pub struct Proposer<B, Block: BlockT, C, A: TransactionPool> {
    spawn_handle: Box<dyn SpawnNamed>,
    client: Arc<C>,
    parent_hash: <Block as BlockT>::Hash,
    parent_id: BlockId<Block>,
    parent_number: <<Block as BlockT>::Header as HeaderT>::Number,
    transaction_pool: Arc<A>,
    now: Box<dyn Fn() -> time::Instant + Send + Sync>,
    metrics: PrometheusMetrics,
    _phantom: PhantomData<B>,
    max_block_size: usize,
}
```

Substrate 共识协议开发

PoW 共识，

- ImportQueue 使用 PowVerifier 对象和 PowBlockImport 对象

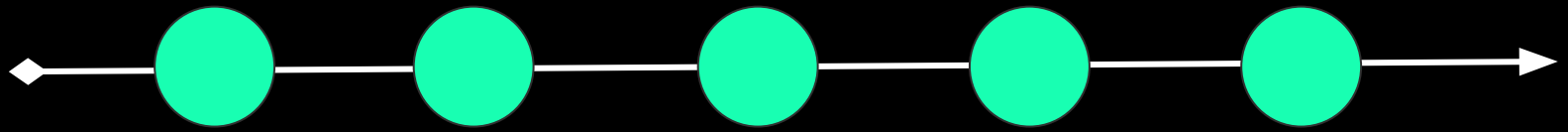
```rust
pub struct PowVerifier<B: BlockT, Algorithm> {
    algorithm: Algorithm,
    _marker: PhantomData<B>,
}

let verifier = PowVerifier::new(algorithm);
```

```rust
pub struct PowBlockImport<B: BlockT, I, C, S, Algorithm, CAW> {
    algorithm: Algorithm,
    inner: I,
    select_chain: S,
    client: Arc<C>,
    inherent_data_providers: sp_inherents::InherentDataProviders,
    check_inherents_after: <<B as BlockT>::Header as HeaderT>::Number,
    can_author_with: CAW,
}
```

Substrate 技术生态

- 开发者课程
- Workshop

- Web3训练营



- 文档 视频
- Seminar
- Playground
- Chat

- 黑客松
- 社区训练营

- Delivery Partner
- Builder Program

未来和展望...

More great features coming...

- FRAME v2
- Sassafras 共识
- Runtime 多线程
- 跨链消息（XCM）
- OCW 本地存储
- 优化数据迁移流程
- ……

Thanks.

官网文档：substrate.dev
知乎专栏：parity.link/zhihu

We are hiring! *@kaichaosun*