Substrate Overview

- Git仓库
- **此版本详解针对V1.0版本**

Introduction

Substrate是一个区块链开发框架，是由Gavin Wood在2018年9月提出的。substrate框架将区块链系统进行了模块化的拆分，形成了用于共识、网络和其他配置的模块化组件，此外substrate底层实现了一套通用的状态转换过程(STF)，STF主要实现于SRML模块，SRML模块为区块链开发实现了绝大部分的底层结构，你只要按照它的标准进行开发，你可以快速开发出一条只专注于业务的区块链，而无需去考虑底层的实现。

Core Datatypes

Substrate底层定义了一套数据类型，要想使用它们，必须先定义并且实现特定接口。

这些数据类型中的每一个都对应于Rust中的trait，SRML模块提供了对这些数据类型的基本实现。

Type	Desc
Hash	对数据加密的模块，大小为256位
BlockNumber	有效块的数量，大小为32位
DigestItem	一种必须能够编码与共识和change-tracking相关的许多"hard-wired"的选择，同时能够编码在Runtime的特定模块中任何数量的"soft-coded"的变量
Digest	一组DigestItem，它编码与light-client相关的所有信息
Header	区块头，包含parent hash,storage root以及the extrinsics trie root,digest和区块号
Extrinsic	区块内数据类型，表示能够被区块识别的外部数据。通常涉及一个或多个签名以及某种编码指令(例如，用于转译资金所有权和调用智能合约)
Block	区块，由Header和Extrinsic组成

Usage

有三种方式可以使用substrate:

1. **直接使用substrate原生链**

> 通过运行预先设计好的substrate节点并且配置创世块的初始化信息，从而快速运行一条区块链。这种方式最为便捷，但是提供了最少量的可定制性，主要允许个人更改各种运行时模块的创建参数，例如balances，staking，block-period，fees和governance。

2. **重构SRML模块**

> Polkadot RelayChain的使用方式。通过重构原生SRML模块或新增自定义SRML模块，并且可能会重写substrate客户端的区块生成逻辑。这给了开发者极大的自由度，可以根据情况自行选择是否需要更改区块生成逻辑。

3. **通用方式**

> 开发自由度最高的方式，等于使用substrate现有模块进行重构，生成属于自己的区块链。可以忽略SRML模块，自己定制Runtime模块。如果逻辑的复杂度不影响区块生成的逻辑，那可以直接用wasm blob构建一个新的Genesis块，并且可以直接使用现有的基于Rust的substrate客户端启动区块链。如果逻辑过于复杂已经与现有区块生成逻辑不兼容，那就要对块生成逻辑进行更改，甚至可能改变区块头以及区块序列化的方式。

Glossary

Substrate和Polkadot代码库中使用的术语表。

Author (aka Block Author, Block Producer)

负责创建区块的节点，也就是矿工。

Adaptive Quorum Biasing (AQB)

法定人数动态化的方法，一种指定Proposal区块成功所需的人数的方法，取决于选民投票率。积极偏见意味着需要的投票数量远超投票数。消极偏见则相反。这是一种不可预知的机制，避免了严格的法定人数的要求。

AfG

“Al's Finality Gadget”的内部代号，以发明它的Alistair Stewart命名。(GRANDPA)

Aggregation

在“模块聚合”的上下文中使用，将来自多个运行时模块的类似类型组合成单个enum类。目前有五种:

- `Log` (可扩展的区块头项)
- `Event` (事件类型)
- `Call` (方法调用)
- `Origin` (函数的调用者)
- `Metadata` (允许对上述类型进行自我检查的信息)

Approval Voting

投票系统，投票人可以根据需要的投票数选出尽可能多的候选人。总票数最高的候选人获胜，但需要注意:

- 投票给所有候选人等同于弃票
- 可以通过投票选举所有其他候选人来“反对”一名候选人

Authority

共识的参与者，即区块验证人，AuthorityId用于识别共识参与者，在POS链中，权限值取决于token加权的提名/投票系统。

Aura (aka "Authority Round")

具有非即时终结性的确定性共识协议，即区块验证人轮流出块。

Aurand

在Aura中选择验证人时，采用随机的方式，增加了安全性。

Aurand/Ouroboros

Aurand的扩展，即验证人们在有限的slots中竞争。

Aurand+GRANDPA

混合共识，将区块生成与区块最终确定进行了分离，由GRANDPA算法进行最终区块确定。

Blind Assignment of Blockchain Extension (BABE)

类似于Aura协议，不同的是采用的方式是VRF而不是指定列表来选择验证人，验证人在某个时间插槽内选择一条链进行区块生成。

Block

区块，由Header和Extrinsics组成，将两部分与一些加密信息进行合并。块通过父指针（作为父代的加密摘要实现）排列到树中，并且通过"fork-choice规则"将树修剪成List。

Byzantine Fault Tolerance (BFT)

分布式计算机系统在面对一部分有缺陷的节点或参与者时保持运行的能力。 "拜占庭"指的是缺陷的最终级别，这些节点被认为是主动恶意和协调而不仅仅是离线或错误。通常，BFT系统保持功能，最多约三分之一的拜占庭节点。

Byzantine Failure

在需要达成共识的系统中，由拜占庭故障导致的系统服务丢失（即系统中的组件发生故障以及组件是否发生故障的信息不完整）。

Consensus

共识模块。

Consensus Engine

对于（一个真实的）规范链构成共识的手段。在substrate中特指的是一组trait的实现，即对多个待选区块中选择一个合法区块。

Consensus Algorithms

一种算法能够确保一组不一定相互信任的参与者可以就某些计算达成共识。

Crypto Primitives

签名规则与hash算法，应用于:

- The blockchain: 区块生成时进行hash计算
- State: 存储被编码为trie，允许对其使用散列算法的任何给定状态进行加密引用。
- Consensus: 验证者对区块签名

- Transaction authentication(交易验证): 在Runtime中使用交易与账户时，账户为一个数字身份。

在SRML模块中，主要使用的是SR25519和ED25519。

Council

委员会是SRML模块治理原则之一，并且是从一些固定期限的批准投票中选出的代表团体。该委员会主要作为一个机构来优化和检查/平衡更具包容性的公民投票制度。它有很多权力:

- 引入具有移除或倒置的公民投票（仅在一致时）AQB
- 取消公投（仅在一致时）
- 放弃他们的权利

Database Backend

交易数据持久化。

Digest

区块头的可扩展字段，对仅用于链同步的轻节点所需的信息进行编码。

Dispatch

使用预定义的参数集执行函数。SRML模块中，这特指"运行时调度"系统，一种获取一些纯数据的方法（该类型称为按惯例调用）并解释它以便在Runtime模块中调用已发布的函数并传入相应的参数。这些已发布的函数采用一个称为origin的附加参数，允许函数安全地确定其调用者。

Equivocating

在共识机制内投票或以其他方式支持多个互斥选项。这基本上被认为是拜占庭行为。

Ethash

POW算法之一，以太坊使用。

Events

事件记录，在SRML模块中，Event类型是可合并的类型之一，每个模块单独定义，并且这些数据类型聚合在一起形成可以表示所有模块类型的单个整体枚举类型。事件通过内存中存储项集实现，这些存储项在块执行后立即检查并在块初始化期间重置。

Executor

在Runtime模块调用外部函数的方法。Substrate有两个Executor，一个是Native，一个是Wasm。

Wasm Executor

一个Executor，它使用Wasm二进制文件和Wasm解释器来执行所需的调用。这往往很慢，但保证正确。

Native Executor

一个Executor，它使用当前内置和本机编译的运行时来执行所需的调用。如果代码与on-chain Wasm代码兼容，那么它将更加快速和正确。版本不正确将导致共识错误。

Extrinsic

块中数据。有两种类型：一种是交易数据(通常是签名过的)以及内部固有数据。

Existential Deposit

在SRML余额模块中，这是账户可能具有的最小余额。无法使用少于某个最小金额的余额创建账户，如果账户的余额低于金额，则会从系统中完全删除。

> **Transferring Small Amounts**
>
> 只要发送方和接收方都保持至少存在性存款的平衡，转移（包括非常少量的转移）将直观地起作用。但是，当发送方或接收方余额非常低时，尝试将小额余额从一个帐户转移到另一个帐户可能会产生稍微不直观的后果。如果转移的金额少于存在性存款且目标帐户以前不存在，则转移将“成功”，而不实际创建和记入目标帐户;这似乎基本上只是来自发件人的转移余额。如果转移使发件人低于存在余额，则其帐户将被删除。通过这种方式，传输甚至可以“成功”导致发件人帐户被完全删除，而接收者帐户永远不会被创建。 中间件取决于确保最终用户了解和/或保护这些边缘情况。

Finality

一旦区块最终确定，结果不可逆转，即最终被确定的区块为最优区块。

Full Client

全节点，能够执行（并因此验证）所有逻辑以最大安全方式同步块链的节点。

Genesis Configuration

基于JSON的配置文件，可用于配置创世区块从而允许单个区块链运行时支持多个独立链。类似于Parity Ethereum中的“链规范”文件，并且当与Substrate Node一起使用时，可以被认为是Substrate的第三个和最高级别的使用范例。SRML模块中提供了一套根据配置生成创世区块的方法。

GRANDPA

基于GHOST的递归祖先导出前缀协议，也称为SHAFT（SHared Ancestry Finality Tool），最初称为AfG，是进行区块最终确认的算法。

Header

区块头，主要是加密信息的片段，用于汇总块。轻客户端使用此信息来获得链的最低安全但同步非常迅速。

HoneyBadgerBFT

具有异步“安全”和异步“活跃度”的即时终结一致性算法（如果经过调整）。最初被定为Polkadot可能探索的许多方向之一，但现在不太可能被使用。

Hybrid Consensus Protocol

将区块链共识拆分为Block Production和Finality Gadget。这使得链增长速度与概率“安全”共识一样快，如Ouroboros或Aurand，但具有与即时可靠性共识协议相同的安全保证级别。

Instant-Finality

一种非概率共识协议，可在块生产时立即提供最终保证，例如Tendermint和Rhododendron。这些往往是基于PBFT的，因此网络通信代价较高，且速度不快。

JSON-RPC

使用JSON协议在远程系统上调用函数的标准。对于Substrate，这是通过Parity JSONRPC包实现的。

JSON-RPC Core Crate

允许创建JSON-RPC服务器处理程序，并注册支持的方法。通过不同类型的传输协议（即WS，HTTP，TCP，IPC）暴露Substrate核心模块。

JSON-RPC Macros Crate

允许通过创建使用RPC方法名称注释的Rust Trait来简化代码中JSON-RPC服务器的创建过程，因此你只需实现方法名称。

JSON-RPC Proxy Crate

从二进制文件中公开一个简单的服务器（如TCP），前面有另一个二进制文件作为代理，将公开所有其他传输协议，并在到达上游服务器之前处理传入的RPC调用，从而可以实现缓存中间件（节省必须一直到节点），允许中间件，节点实例之间的负载平衡，或将帐户管理移动到处理签名事务的代理。这提供了在每个项目中嵌入整个JSON-RPC以及每个服务器的所有配置选项的替代方法。

JSON-RPC PubSub Crate

自定义（虽然是常规）扩展，对Dapp开发人员很有用。它允许"订阅"，以便服务器自动向客户端发送通知（而不是一直手动调用和轮询远程过程），但仅限于支持客户端和服务器之间持久连接的传输协议（即WS，TCP， IPC）。

Keystore

Substrate中的子系统，用于管理密钥以生成新区块。

Libp2p Rust

点对点网络库。基于协议实验室在Go和JavaScript中的Libp2p实现，以及它们（有点不完整）的规范。允许使用许多传输协议，包括WebSockets（可在Web浏览器中使用）。

Metadata

信息捕获允许外部代码调用Runtime数据结构。包括Runtime中每个模块的调度功能，事件和存储项的全面描述。

Nominated Proof-of-Stake (NPoS)

一种确定一组验证器（以及因此权限）的方法，这些验证器来自愿意将其利益提交给一个或多个创作/验证器节点的正确（非拜占庭）功能的多个账户。这最初是在Polkadot论文和短语中提出的一组赌注提名作为约束优化问题，最终给出了一组最大限度的验证者，每个验证者都有许多支持提名者的利益。削减和奖励是按比例进行的。

OAS3

OpenAPI规范3（OAS3），正式称为Swagger API规范，是用于创建JSON或YAML格式的Swagger文件的规范，可用于生成API参考文档，如https://substrate.readme.io/v1.0.0/reference

Origin

是谁调用了Runtime模块的函数，可以定制(新建)，有两个特殊的内置Origin:

- **Root**: 系统级别的起源，假设是无所不能的
- **签名**: 交易来源，包括签名者的账户标识

Practical Byzantine Fault Tolerance (pBFT)

实用的拜占庭容错（pBFT） 一种解决拜占庭将军问题的原始方法。这允许系统容忍来自其参与者的多达三分之一的拜占庭行为，每个节点具有O（2N^2）通信开销。

Probabilistic Finality

在基于概率最终性的链（例如比特币）中，网络参与者依赖于一些概率p，即所提出的块B将无限期地保留在规范链中，其中p接近1，因为在块B的顶部产生了更多的块。

Proof-of-Finality

可用于证明特定块已完成的一段数据。除非使用签名聚合，否则可能非常大。

Provable Finality

一些共识机制的目标是可证明的最终结果，即所有块都被保证在块包含时成为该链的规范块。在诸如轻客户端不具有完整链状态或与其他链通信的情况下，其中链间数据不是普遍分布的情况下，可证明最终性是可取的。

Rhododendron

Instant-Finality BFT一致性算法，是区块链的多种PBFT变种之一。其他包括Stellar的一致性算法和Tendermint。参见Rhododendron crate。

Runtime

区块执行逻辑，即状态转换函数。

SHAFT

SHAFT（Shared Ancestry Finality Tool）。GRANDPA

Stake-Weighted Voting

token的质押比值来决定投票权重。

State

可以在顺序块的执行之间读取并持久化的数据。状态存储在“Trie”中，这是一种加密不可变数据结构，使得增量摘要非常有效。在Substrate中，此trie作为简单的键/值映射暴露给Runtime模块，其中键和值都可以是任意字节数组。

SRML (Substrate Runtime Module Library)

一种可扩展，通用，可配置和模块化的系统，用于构建运行时并共享它们的可重用组件。从广义上讲，这对应于Substrate的第二种使用范例，比使用Substrate Core更高级别，但低于仅使用自定义配置运行Substrate Node的级别。

STF (State Transition Function)

状态转换函数，在Substrate中，这基本上等同于Runtime。

Storage Items

在SRML中，存储项是为运行时提供类型安全持久性数据的一种方法，通过编译器宏，奇偶校验码文件夹和状态API实现。所有存储项必须具有实现parity-codec :: Codec特征的类型，如果它们具有相应的JSON Genesis Configuration条目，则还必须具有serde特征。

Substrate

区块链框架，用于构建和部署可升级，模块化和高效区块链的框架和工具包。三种使用范例：Core、SRML and Node。

Substrate Core

三个Substrate使用范例中的最低级别，这是一个极简主义/纯粹的区块链构建框架，包含最低级别的基本功能，包括共识，块生成，链同步，JSON-RPC的I/O，Runtime，网络同步，数据库后端，遥测，沙盒和版本控制。

Substrate Execution Environment

运行WebAssembly代码以运行块的运行时环境。

Transaction

一种包含签名的Extrinsic，并且所有有效实例在链接时包含签名者一定数量的令牌。由于有效性可以有效确定，交易可以在网络上散布，具有合理的安全性来抵御DoS攻击，就像比特币和以太坊一样。

Transaction Era

可定义的时间段，表示为一个块号范围，其中交易可以有效地包含在块中。在收回帐户并且其（重放保护）随机数被重置为零的情况下，时代是反对交易重放攻击的后盾。时间可以在交易中有效地表达，并且仅花费两个字节。

Transaction Pool

尚未包含在块中但已确定为有效的交易集合。

Tagged Transaction Pool

“通用”交易池实现，允许Runtime指定给定交易是否有效，如何优先处理以及如何在依赖性和互斥性方面与其他交易相关。它的设计易于扩展，并且通用性足以使UTXO和基于账户的交易模块都可以轻松表达。

Trie (Patricia Merkle Tree)

一种不可变的加密数据结构，通常用于表示地图或项目集，其中:

- 需要数据集的加密摘要
- 使用数据集的增量更改重新计算摘要是很便宜的，即使数据集非常大

- 需要一个简明的证据，即数据集包含一些项目/对（或缺少它）

Validator

一个半信任（或不受信任但受到良好激励）的角色，有助于维护网络。在Substrate中，验证人员广泛地对应于运行共识系统的Authority。在Polkadot中，验证器还管理其他职责，例如保证数据可用性和验证链式候选块。

WebAssembly (Wasm)

基于虚拟机的执行架构。

Extrinsics

在Substrate中Extrinsic就是数据片段，主要有两种类型:交易和固有数据。

Transactions

交易通常指的是被一方或多方签名过的数据或行为，这些行为可能造成状态转换或资金流动，并且他们能够被预执行并进行广播，而不会是垃圾信息。

Inherent Extrinsics (aka Inherents)

固有数据。一般来说，固有者没有签名，也没有任何其他内在价值的加密指示。因此，他们不像交易那样得到广播。（从技术上来说，没有任何东西可以阻止substrate广播，但是没有基于费用的垃圾信息预防机制。）相反，它们代表的数据以自以为是的方式描述了许多有效信息之一。除此之外，它们被认为是“真实的”仅仅是因为足够多的验证者已经同意它们是合理的。

举一个例子，有固有的时间戳，它设置块的当前时间戳。这不是Substrate的固定部分，但确实是SRML的一部分 根据需要使用。没有签名可以从根本上证明一个块在给定时间以与签名可以“证明”转移某些特定资金的愿望完全相同的方式生成。相反，每个验证器的业务是确保他们在同意块候选有效之前将时间戳设置为合理的值。

另一个例子是SRML中用于确定和惩罚或停用离线验证器的“注释错过的提议”。同样，这仅仅是验证者意见的声明，并且由链的运行时逻辑决定对“意见”采取什么行动。

Blocks and the Extrinsic Trie

Extrinsics会被绑定在区块中并在Runtime时执行。区块头extrinsics_root中有一个显式字段，它是这一系列extrinsics的加密摘要。这有两个目的:它防止（在恶意方或其他方面）对区块头构建和分发后的一系列外在函数进行任何更改。其次，它提供了一种允许轻客户端节点简洁地验证任何给定的外部确实确实存在于块中的方法，只需要知道头部。

Extrinsics in the SRML

SRML为Extrinsic提供了两个基础泛型实现:UncheckedExtrinsic（基本变体）和UncheckedMortalExtrinsic（更复杂），后者是前者功能的超集。基本变体提供了在最多一个签名的SRML模型下的外部特征的最小但完整的实现以及可分派的调用。 更复杂的变体提供了这种功能以及编码“时代”的可能性，从而允许交易仅对特定范围的块有效。

这些实现对以下四种类型通用:

- `Address`: 交易中记录的发件人格式（如果有）。与Inherents无关。
- `Index`: 编码交易索引的类型。与Inherents无关。

- `Signature`: 签名，它既定义了保存签名的数据类型，又暗示了验证签名的逻辑。（目前`sr25519`签名已实施。）与Inherents无关。
- `Call`: 一种类型，用于通知链上逻辑应该调用哪些代码以执行此外部函数。一般来说，这必须实现特殊的调度trait。

当Executive和Block专用时，会创建SRML运行时内的进一步通用功能。这些确定了如何将地址转换为AccountId，并可以传入一个类型化的上下文数据来帮助实现这一点。

Block-authoring Logic

在Substrate中，区块链同步和块生成之间存在区别。仅涉及同步的全节点（和轻节点）不执行与块生成节点相同的功能。具体而言，除了完全同步之外，块生成节点还需要交易队列逻辑，固有交易知识和BFT一致逻辑。 BFT一致性逻辑是作为Substrate的核心元素提供的，可以忽略，因为它只在SDK下的authorities()API条目中公开。

Substrate中的交易队列逻辑被设计为尽可能通用，允许runtime通过initialise_block和apply_extrinsic调用来表示哪些交易适合包含在块中（全（同步）节点将仅运行execute_block）。但是，优先级和替换策略等更微妙的方面目前必须作为区块链创作逻辑的一部分进行“硬编码”。也就是说，Substrate的交易队列参考实现应该足以用于初始链实现。

固有extrinsic知识也是非常通用的，按照惯例，extrinsics的实际构造被委托给runtime中的“软代码”。 如果链中需要额外的extrinsics，那么:

- 需要更改块生成逻辑以将其提供给runtime
- runtime的intrinsic_extrinsics调用将需要使用这些额外的信息，以构造包含在块中的任何其他extrinsics。

Runtime and API

所有Substrate链都有runtime模块，runtime是WebAssembly二进制的blob，包含节点已知的许多入口函数(entry-point)。Substrate有两个核心入口点:execute_block和version。入口点(entry-point)分为API特征集和被表示为Trait(自己实现)。核心的两个入口点被实现于Core API trait。

可以提供Core以外的API来添加或支持区块链客户端的其他功能。有许多标准API能够帮助提供常见Substrate客户端组件的功能，包括runtime模块的metadata，块生成(block authoring)，交易队列。

Declaring APIs

当使用Rust来构建runtime模块，impl_apis宏可用于简化新入口点的编组和分派。用其他语言编写的runtime需要手动实现这样的逻辑，包括数据类型编码/解码，通过预先分配的内存管理参数的传递和返回类型，并将入口点函数公开为publish API。 （有一个基于Substrate Simple Codec（编解码器）的特定ABI，用于对这些函数的参数进行编码和解码，并指定它们应在何处以及如何传递。）

Transaction Lifecycle

Substrate允许开发者自定义runtime模块(应用逻辑)，可以将任何类型的信息放入块中。尽管在Substrate Core和SRML模块中提供了一些基本功能，区块链开发者需要验证数据被打包上链之前的合法性。

本文会介绍Substrate提供的工具以及如何对自定义模块进行合法性检查。

块中的所有数据被称为extrinsic，并且extrinsic被分为两种类型:

交易(Transactions)是签名过的信息，可以认为是比特币或者以太坊交易。交易在网络中goosip并且可以被任意节点提交到区块中。

Inherents是未签名过的信息，其来源、真实性以及内容不是必然可证的，它们由生成区块的节点自行添加到区块中。如时间戳的修改、用于共识的一组验证者集合。这些是"真实的"，因为网络中的其他参与者通过验证并同意它们加入区块。Inherents数据不会在网络中传播，由区块生成者直接写入区块。但是，inherents仍会影响状态，因此你需要一种方法来验证它们是否会在最终确定之前按计划影响存储。

交易的来源主要是两种方式:

- 从网络中接收区块
- 自己生成区块

对于第一种情况，块通过`execute_block`运行，整个块成功或者失败。

第二种情况，节点在发布块之前需要做更多的检查。

1. 监听网络中的交易
2. 通过`validate_transaction`函数验证交易，并将合法交易写入交易池
3. 交易池对交易进行排序并且返回已经能够被写入区块的交易。我们用`BlockBuilder`来构造区块
4. `BlockBuilder`使用`apply_extrinsic`函数来执行交易并且将状态更改写入本地内存
5. 构造好的区块被发布到网络中(其他节点都运行`execute_block`)

在第二步中，`validate_transaction`会执行一些基本检查并且返回一个包含变量Valid,Invalid和Unknown的enum类。`validate_transaction`在runtime被调用，并检查签名是否有效以及随机数，但不检查是否有任何模块调用成功。`validate_transaction`单独检查交易，因此它不会捕获错误，例如双花。

实际上，可以使用`validate_transaction`来检查对模块的调用，因为它在runtime时能够被调用。但是，我们不建议这样做，因为它是一个潜在的DoS向量，因为网络中的所有交易都将传递给validate_transaction。validate_transaction函数应该专注于为池提供必要的信息以便对交易进行排序和优先级排序，并快速拒绝所有肯定无效或过时的交易。注意，该函数将被频繁调用（对于同一事务，也可能多次调用）。如果以正确的顺序执行，则`validate_transaction`也可能使那些可能通过`execute_block`的相关交易失败。

我们使用`validate_transaction`的输出来在交易池中对交易排序并且使用`BlockBuilder`构造区块。区块中的所有extrinsics会使用`apply_extrinsic`函数执行。它会进行所有模块调用并将状态转换应用于本地内存。如果它没有任何问题，那么块就会传播到网络，就像第一种情况一样，每个人都会运行execute_block。在本地，块被导入链而不运行execute_block（我们重用内存更改集）。

请记住，对于Substrate，token传输不是唯一的状态转换：任何信息都可以写入块。execute_block永远不应该发生混乱，如果对模块的调用返回Err，它将被最终确定，并且不会恢复对存储的更改。这是因为恢复更改将要求每个节点在每个块复制存储。

当你要修改存储时，最佳做法是执行必要的每项检查以确保调用成功，执行所有存储更改，最后发出事件，以便自己能够知道该函数未提前返回。具体案例可以查看`transfer`函数(在balances模块中)，所有的数据必须通过检查才能写入内存。

如果你正在自己定制模块，那么你需要确保无效的extrinsics永远不会出现panic并且在返回Err时永远不会修改模块状态。正确实现模块的方式应当确保能够惩罚恶意方提交无效extrinsics（防止DoS攻击）。

Substrate为开发者提供了极大的自由度，可以对区块和存储进行个性化构建。因此并非所有的错误都是可被Substrate工具包捕获的，因为runtime模块不做限制。开发者需要实现自己的错误检查机制，以防止非法操作。