# 初识substrate区块链框架

## 6.1 Substrate & Polkadot简介

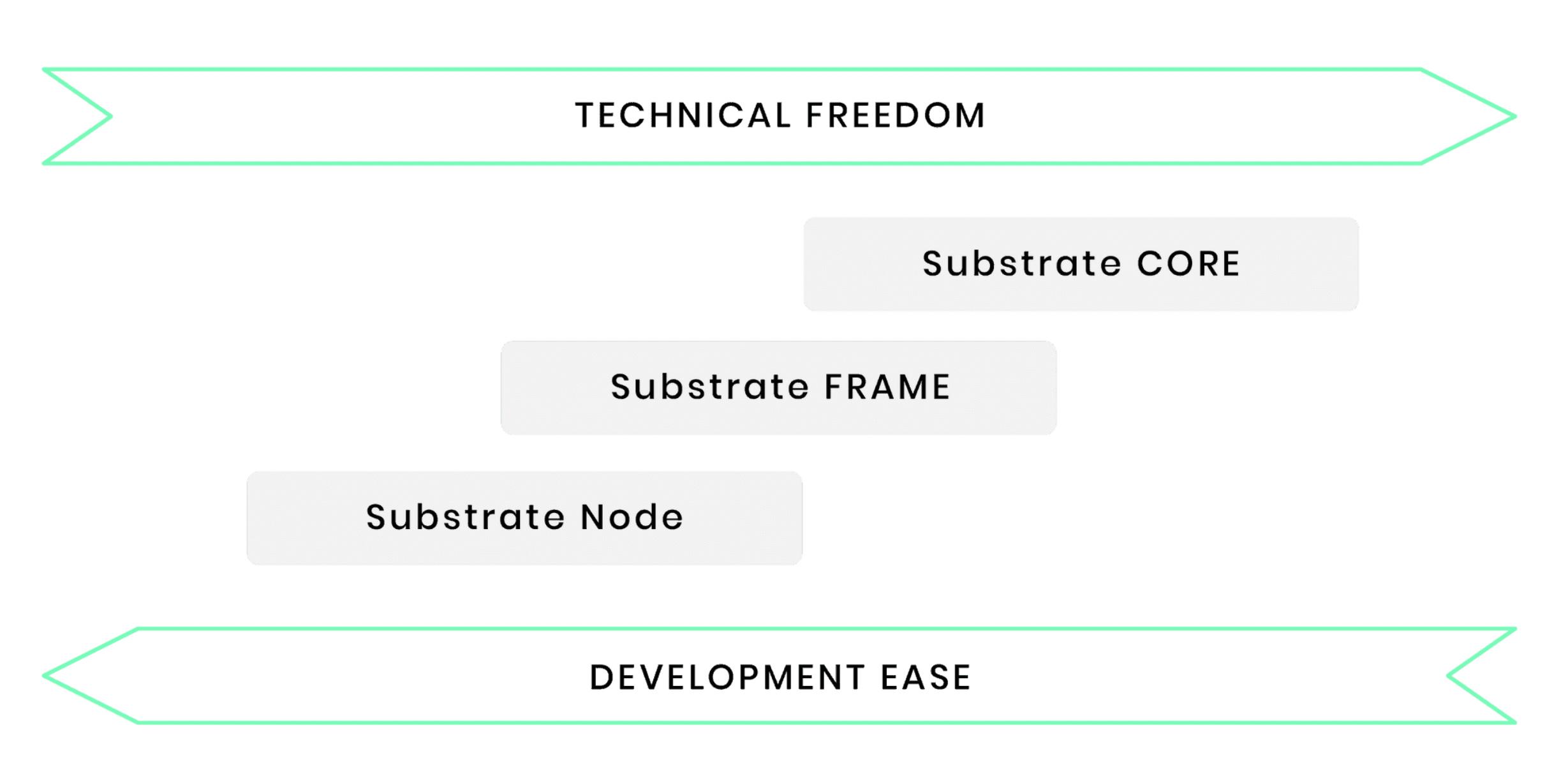
### 6.1.1 Substrate概述

Substrate是一个用于构建区块链的开源、模块化以及可扩展的框架，使你能够过组合自定义或预构建组件来创建专门构建的区块链。Substrate也为下一代异构多链网络Polkadot提供动力多支柱。

使用substrate的方式主要有以下几种：

1. 使用Substrate节点：开发者可以运行已经设计好的substrate节点，并配置genesis区块，在这种方式下，开发者只需要提供一个JSON文件就可以启动自己的区块链。这个JSON文件允许你配置构成Substrate节点运行时的模块的genesis状态。
2. 使用substrate frame：开发者可以使用Frame轻松创建自己的自定义运行时，因为Frame是用来构建底层节点的。使用Frame还可以配置数据类型，也可以从模块库中选择甚至是添加自己定义的模块。
3. 使用substrate core：整个FRAME系统可以忽略不计，运行时可以从头开始设计和实现。如果运行时可以与 Substrate 节点的抽象块创作逻辑兼容，那么您只需从 Wasm blob 构造一个新的genesis区块，并使用现有的基于 Rust 的 Substrate 客户端启动链即可。如果不是，则需要更改客户端的块创作逻辑，甚至可能更改标头和块序列化格式。在开发工作方面，这是迄今为止使用Substrate最困难的方式，但也给了你最大的创新自由。

Substrate允许开发人员在设计决策的每一步中在技术自由和易于开发之间做出选择,下图说明了这种灵活性的本质。



### 6.1.2 Polkadot概述

Polkadot 是将多个专用区块链连接到一个统一网络中的下一代区块链协议。parity团队在开发跨链基础设施Polkadot项目过程中，提炼出了Substrate框架。Substrate是独立于Polkadot的区块链系统框架，而基于Substrate以及Polkadot的区块链项目已经越来越多，整个Polkadot生态也是愈发强大。

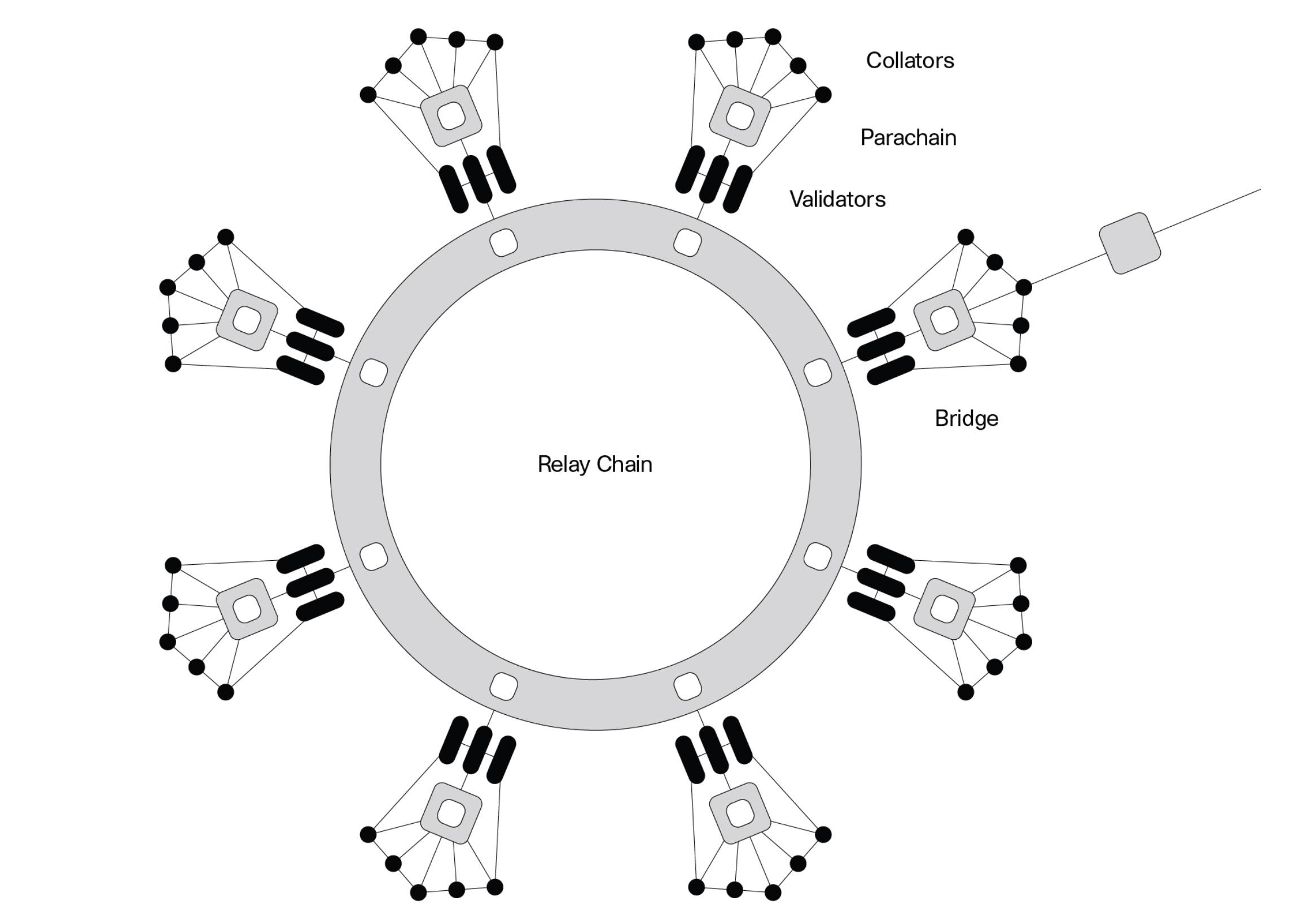
Polkadot将实现一个用户可以控制的、完全分散的网络。Polkadot旨在连接私有和联盟链，公共和无许可网络，以及尚未创建的未来技术。Polkadot促进了互联网，独立区块链可以通过Polkadot中继链以无信任的方式交换信息和交易。

Polkadot 使创建和连接分散式应用程序、服务和机构变得前所未有的容易。通过授权创新者构建更好的解决方案，我们寻求将社会从对破碎网络的依赖中解放出来，在这个网络中，其大型机构不能违反我们的信任。

### 6.1.3 Polkadot特性

Polkadot网络使用分片模型，其中分片 （ 称为"平行链" ） 允许并行处理事务，而不是按顺序处理。网络中的每个平行链都有一个唯一的状态转换函数（STF）。Polkadot有一个中继链作为系统的主链。基于Polkadot的设计，只要一个链的逻辑可以编译成Wasm并遵循中继链API，那么它就可以作为副链连接到Polkadot网络。

平行链构建区块并将其建议给中继链上的验证者，其中区块在被添加到最终链之前经过严格的可用性和有效性检查。由于中继链提供了安全保障，整理者（这些平行链的完整节点）没有任何安全责任，因此不需要强大的激励系统。这就是整个网络如何与发生的许多交易保持同步的方式。



为了与想要使用自己的最终确定过程的链（例如比特币）进行交互，Polkadot具有提供双向兼容性的桥式平行链，这意味着可以在不同的平行链之间进行交易。

交叉共识消息传递格式 （XCM） 允许平行链相互发送任何类型的消息。中继链的共享安全性和验证逻辑为无信任消息传递提供了环境，从而打开了真正的互操作性。

## 6.2 Substrate原理进阶

### 6.2.1 主要概念

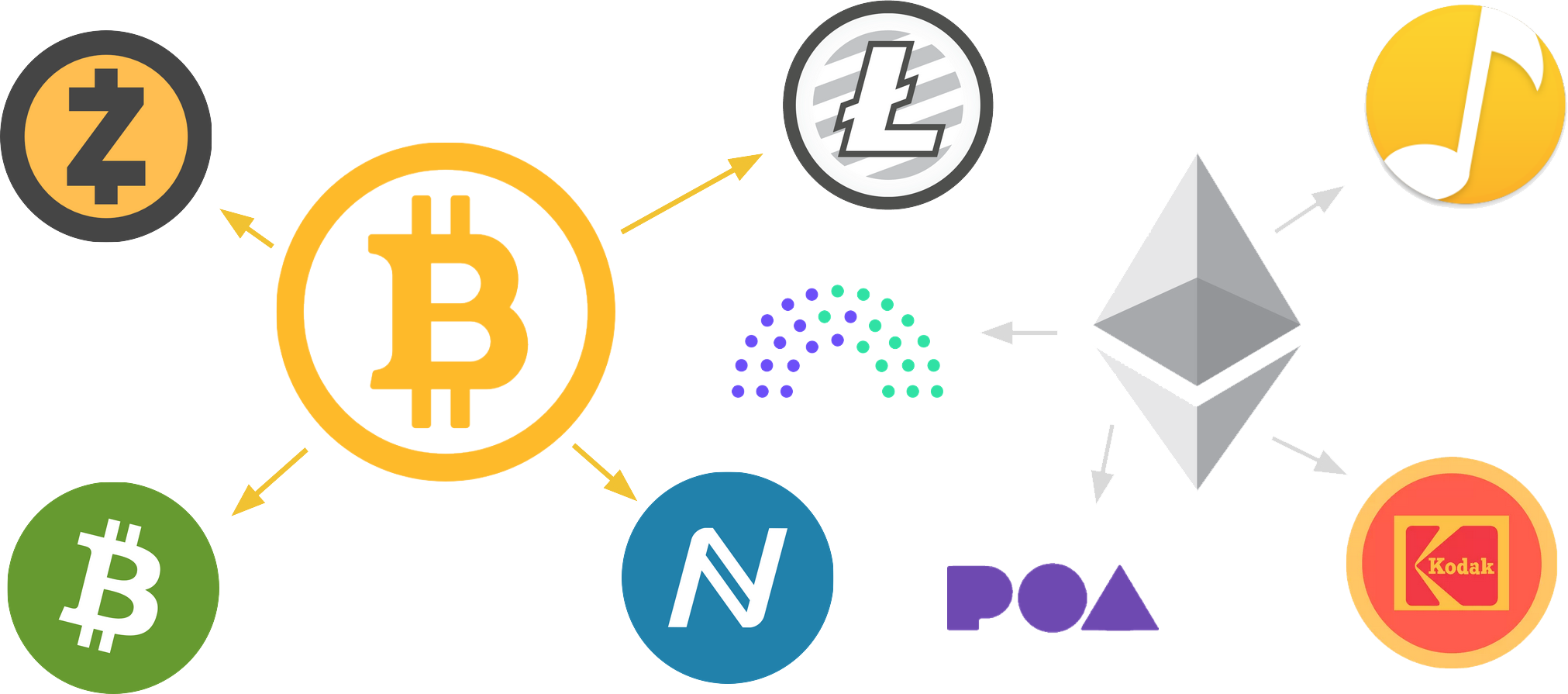
* **区块链基础**

区块链本质上就是一种有多个“节点”组成的一种不可篡改且可追踪溯源的哈希链条。这些节点是区块链网络上相互独立运行的计算机。为了实现区块链功能，这些节点需要以下特征与功能:

* 存储
* P2P 网络
* 共识机制
* 交易处理
* 以及其他...

这些技术涉及领域很广，因此通常需要专家团队来开发。因此，大多数区块链项目都不是从头开始开发的。相反，这些项目是从现有的区块链代码库中分叉而来。例如：

* 莱特币(Litecoin), ZCash, Namecoin, Bitcoin Cash 等等是基于比特币库（Bitcoin）分叉而来；
* Quorum, POA Network, KodakCoin, Musicoin 等等是基于以太坊库分叉而来。



以这种方式构建区块链存在严重的局限性，因为这些现有平台在设计时并未把修改考虑进去。

### 6.2.2 交易池(Trasaction Pool)概念及原理

交易池包含所有已被本地节点接收并验证的广播到网络的交易，包括已签名和未签名的交易。

交易池需要检查交易是否有效。注意交易的有效性不是硬性规定在交易池逻辑里，而是由runtime决定的。检查交易有效性的例子有：

* 检查交易索引 (nonce) 是否正确。
* 检查帐户是否有足够的资金来支付相关费用。
* 检查签名是否有效。

交易池还定期检查池内现有交易的有效性。如果发现交易无效或已过期，则该交易将从池中删除。

如果交易是有效的，交易队列会将交易分为两组：

* 就绪队列 - 包含所有可放到新的待处理区块中的交易。 对于随 FRAME 构建的 Runtime，所有交易必须严格遵循就绪队列中的顺序。
* 未来队列 - 包含所有可能在未来变成有效的交易。 例如，一个交易可能有一个对其账户来说过高的索引 (nonce) 值。 此交易将在未来队列中等待，直到之前的交易上传至区块链上。

注意：是可以设计一个自定义 Runtime 来删除严格的交易顺序要求。这样就可以让全节点在交易传播过程和打包区块过程上实施不同的策略。

### 6.2.3 交易优先级

在有效的交易结构体中交易优先级决定了就绪队列中的交易顺序。当某个节点成为下一个区块生成者时，它将在下一个区块把交易按优先级别从高到低排序，直到达到区块的重量或长度限制。

交易优先级定义了当一个交易可解锁多个依赖交易时，所应有的线性排序。 例如，如果我们有两个（或更多）交易的依赖性得到满足，那么我们会使用优先级来选择它们的处理顺序。

对于用 FRAME 构建的 Runtime，优先级定义为交易要支付的 fee (费用)。 例如：

* 如果我们从不同的发送者那里收到2个交易（而且nonce=0时），我们通过 priority 来确定哪个交易更为重要，并优先把它打包进区块中。
* 如果我们从同一个发送方收到2 个相同nonce的交易，那么只会有一个交易会被打包到链上。 我们使用优先级来选择fee较高的交易，并把它储存到交易池中。

注意该交易池并不知道费用、账户、或签名 － 它只处理交易的有效性和 优先级、requires和provides参数这些抽象概念。 所有其他详细信息都是由 Runtime通过validate\_transaction函数定义的。

### 6.2.4 账户

Substrate 使用多组公钥/私钥对来表示网络的参与者。区块链系统包含不同角色的参与者，比如验证者和普通用户。举例来说，Substrate 节点使用提名权益证明 (NPOS) 算法来选择验证者。验证者和提名者可能持有大量资金，，所以 Substrate 中的 Staking 模块引入了抽象账户体系，以便尽可能保障资金的安全

这些抽象是：

* 隐匿密钥（Stash Key）：隐匿账户用来存放大量资金，因此它的私钥应尽可能安全地放在冷钱包中。
* 控制密钥（Controller Key）：控制账户可通过签名来控制隐匿账户的行为，比如用户的支付偏好；他应该仅保留用来支付交易费的最低额度的资金。由于它可能影响验证者的设置，应该保证私钥安全，但也需要在日常维护时使用。
* 会话密钥（Session Keys）：会话密钥是保存在验证者客户端中的“热”密钥，用于对特定操作进行签名。它们不应持有资金。

### 6.2.5 账户秘钥

密钥对代表一个账户，并可以控制资金，类似于其他区块链中的普通账户。在 Substrate 的 Balances 模块中，账户必须有不低于最低金额（保证金）的余额来确保账户保留在存储中。账户密钥可以自由定义，在 runtime 中才有实际意义。隐匿账户和控制账户中，密钥是以用途区分的，与底层的加密差异无关。从密码学的角度来看，创建隐匿密钥或控制密钥与所有普通账户密钥没区别。

隐匿密钥是隐匿账户的公钥/私钥对。这个账户就像一个“储蓄账户”，你不应该经常与它进行交易。因此，私钥应以最安全的方式来处理，例如在安全层或用安全硬件来加以保护。

隐匿密钥处于离线状态，它指定一个控制账户，根据隐匿账户的资金权重，来做出与支付无关的决定。它还可以指定一个代理账户代表它在治理中进行投票。

控制密钥是定义控制帐户的公/私钥对。在 Substrate 的 NPOS 模型中，控制密钥用来代表某个人验证或提名。

控制密钥用来进行奖励收取等相关设置，如果是验证者，则用于设置其会话密钥。控制账户只需要支付交易费，所以只应持有最小限度的资金。

永远不要用控制密钥从其存储账户中支出资金。然而，控制账户的行为可能会导致花销，因此仍应妥善处理。

### 6.2.6 Session Key

会话密钥是验证者用来签署和共识相关消息的 "热密钥"，不应作为控制资金的账户密钥，仅应该用于特定用途。它们可以定期更改，只需要控制账户对会话公钥签名，并向外广播来创建一个新的证书。会话密钥同样可以自由定义，在 runtime 中才有实际意义。

创建会话密钥的过程实际上是验证操作者必须证明某个密钥，可以代表其会话帐户 (抵押) 和提名人。为此需要使用自己的控制密钥对该密钥签名，从而创建证书。然后，他们在链上发布一笔包含该会话证书的交易，来告知区块链该会话密钥代表了他们的控制密钥。

Substrate 提供了让验证者来管理他们的会话密钥的会话模块。

你可以声明任意数量的会话密钥。例如，Substrate 的默认节点使用 BABE、GRANDPA 和 "I'm Online" 三个会话密钥。其他链可多可少，这取决于链需要验证者执行哪些操作。

这些不同的会话密钥可以使用相同的加密技术，但 runtime 逻辑中的服务是完全不同的。为了防止错误的操作使用错误的密钥，使用 Rust 强生类型来封装这些密钥，让它们彼此不兼容，确保它们仅被用于预期目的。

如果会话密钥被泄露，攻击者可提交导致节点权益损失的行为。会话密钥应该定期更改 (例如每次会话中)，可通过rotate\_keys RPC。

### 6.2.7 链下工作机

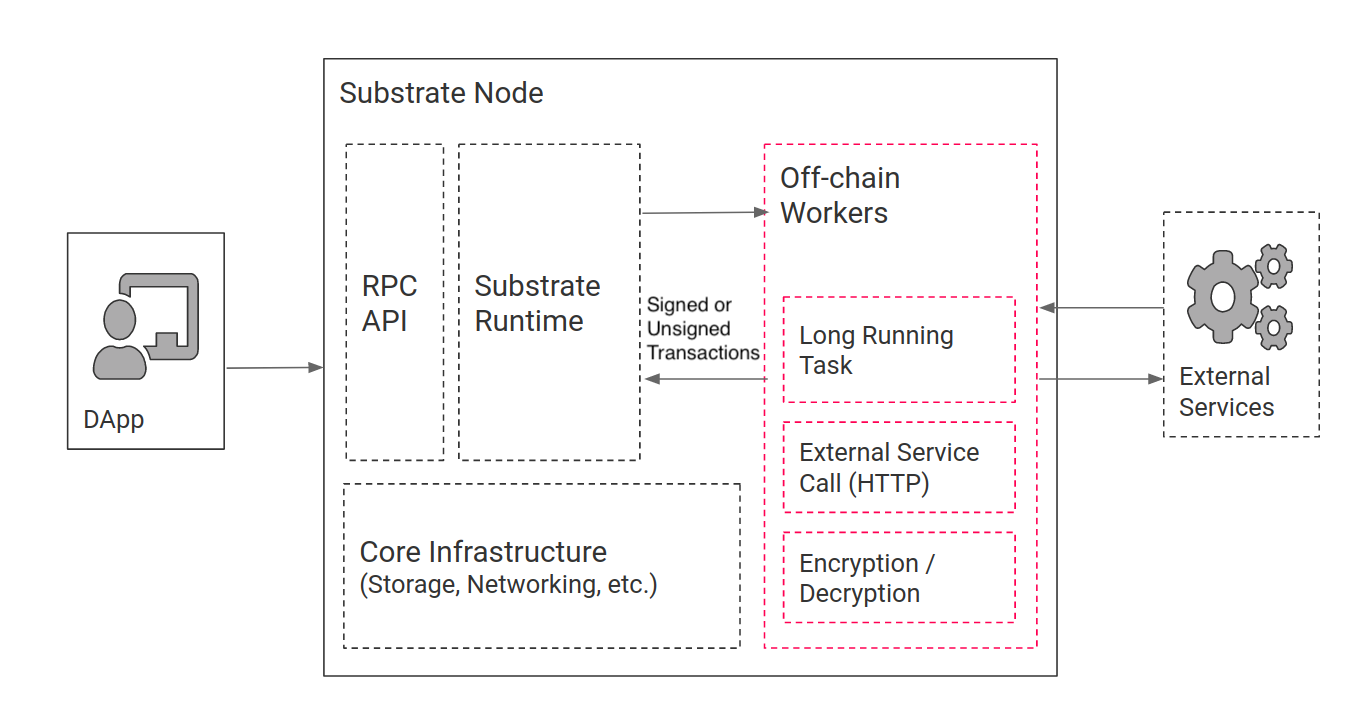
* **概述**

很多时候我们需要先查询或处理链下数据，再将其提交到链上。常规做法是使用预言机（Oracles）。预言机是一种外部服务，通常用于监听区块链事件，并根据条件触发任务。当这些任务执行完毕，执行结果会以交易的形式提交至区块链。虽然这种方法可行，但在安全性、可扩展性，和基本效率方面仍然存在一些缺陷。

为了让链下数据整合更安全、更高效，Substrate 提供了链下功能。

* 链下工作机(OCW) 子系统允许执行运行时间长并且非确定性的任务（如网络请求、数据的加密/解密和签名、随机数生成、CPU密集型计算、链上数据的枚举/汇总等），他们的执行时间可能超过出块时间。
* 链下存储 提供 Substrate 节点的本地存储，既可以被链下工作机访问（包括读写），也可以被链上逻辑访问（通过链下索引写，但不能读）。对于 Worker 线程之间通信，以及存储用户/节点不需要在整个网络上达成共识的数据，是非常好的选择。
* 链下索引 允许 runtime 如果选择加入，可以独立于 OCW 直接写入链下存储，作为链上逻辑的本地/临时存储，也是对链上状态的补充。

链下功能运行于自己的 Wasm 执行环境中，在 Substrate runtime 之外。分离这些关键点确保了区块生产不会受到链下任务长时间运行的影响。然而，链下功能与 runtime 声明在相同的代码中，可以很轻松访问链上状态并进行计算。



* **链下工作机**

链下工作机可以使用扩展 API 与外部世界沟通。

* 能够向链上提交交易submit transactions（已签名或未签名）发布计算结果。
* 全功能 HTTP 客户端，使链下工作机可访问和获取外部数据。
* 访问本地密钥库，来签署、验证声明（statements）或交易。
* 在所有链下工作机之间共享的另一个本地 键值数据库。
* 安全的本地熵源（entropy），用于随机数生成。
* 访问节点的精确 本地时间。
* 休眠和恢复工作的能力。

OCW 可以从 runtime 实现的一个特殊函数内启动， fn offchain\_worker(block: T::BlockNumber) 。将结果传回链上，OCW 可以提交签名或无签名交易，将数据包含在后续区块中。

值得注意的是，链下工作机的交易不受常规交易验证约束。所以需要另外实现一套交易验证机制 (例如投票，取平均值，检查提交人签名或简单地 "信任")，以确定哪些信息能够记录在链上。

* **链下存储**

顾名思义，该存储不在链上。可以通过链下工作机（读取和写入）和链上逻辑（仅写，请参阅下面的链下索引）来访问它。该存储在区块链网络之间不存在，不需要对其共识计算。

由于在每个区块导入过程中，都会有一个链下工作机产生，所以无论何时都可能存在多个链下工作机在运行。因此，与多线程编程环境类似，在访问存储时，也有实用程序互斥锁定存储，以保证数据的一致性。

链下存储充当各链下工作机之间以及链外逻辑和链上逻辑之间相互通信的桥梁，还可以通过远程过程调用（RPC）来读取，因此适合存储无限增长的数据，不会过度消耗链上存储。

* **链下索引**

区块链环境的存储主要指链上状态，但它的成本很高（因为被填充到网络中每个节点），不建议用于历史数据或用户生成的数据，这些数据随着时间的推移无限增长。

为此，我们有链下存储。除了可被 OCW 访问，Substrate 还包含了一个叫做"链下索引"的功能，允许 runtime 独立于 OCW 直接写入链下存储。节点在启动 Substrate 时，必须通过 --enable-offchain-indexing 选择开启这个数据的持久化。

与 OCW 不同的是，OCW 在初始区块链同步过程中并不执行，而链下索引则每次处理区块都在填充存储，所以数据始终是一致的，对于每个启用索引的节点来说，数据会完全相同。

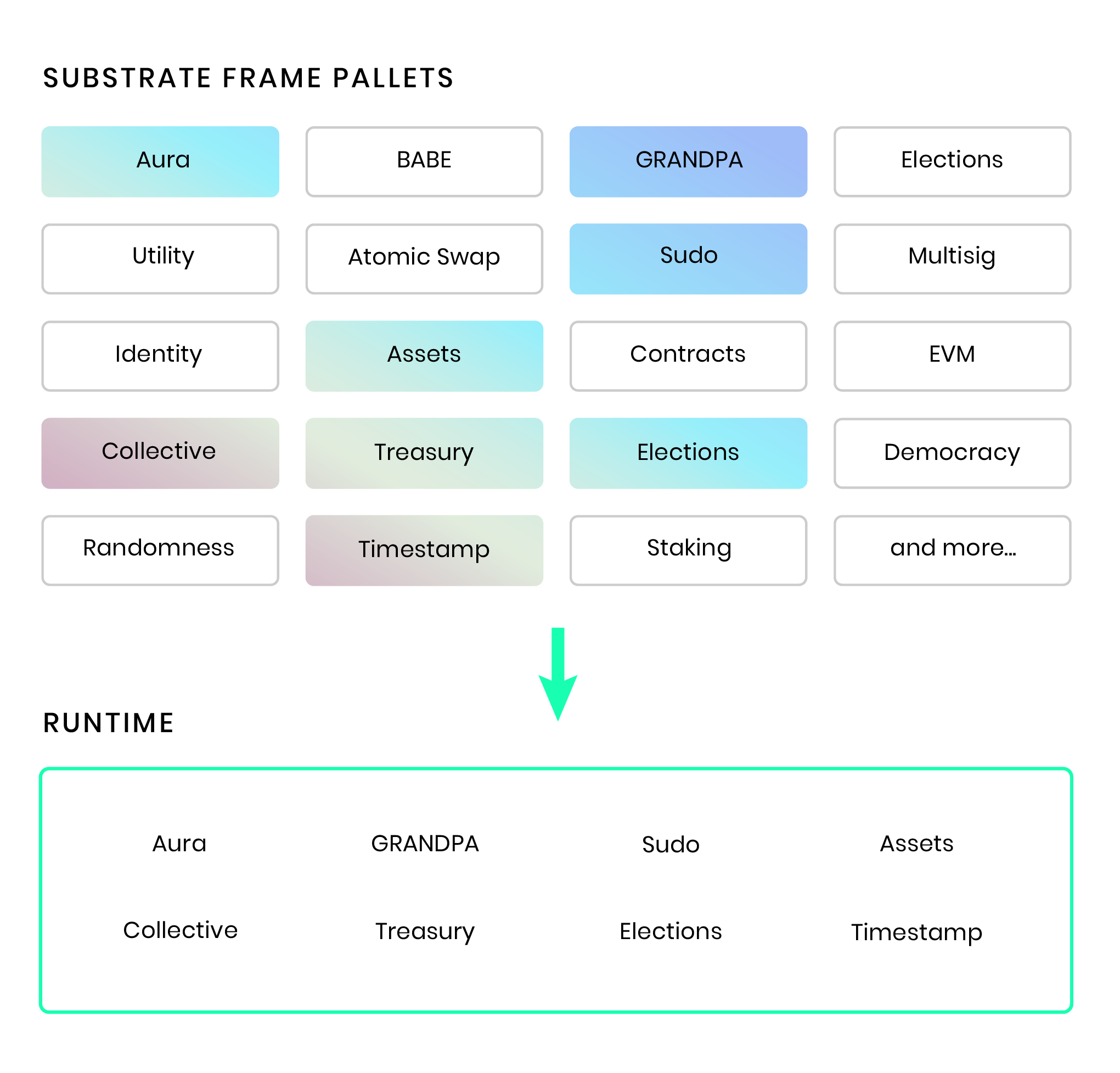
## 6.3 Runtime运行时开发

### 6.3.1 概述

Runtime 用来定义区块链的业务逻辑。在 Substrate 开发的区块链中，runtime 被称为”状态转换函数“；Substrate 开发人员在 runtime 中定义了用于表示区块链状态的存储项，同时也定义了允许区块链用户对该状态进行更改的函数。

为了无须分叉地升级，Substrate 采用了可编译成 WebAssembly (Wasm) 字节码的 runtime 形式。此外，Substrate 还对 runtime 必须实现的核心基本类型 进行定义。

核心 Substrate 代码库附带 FRAME 框架，FRAME 是 Parity 团队的 Substrate runtime 开发系统，已用在 Kusama 和 波卡 等链上。 FRAME 定义了额外的 runtime 基本类型，并提供了一个框架，让编写模块 ("pallet") 构建 runtime 变得十分容易。每个 pallet 用于封装特定于该域的逻辑，这些逻辑可表示为一组存储项、事件、错误和可调用函数的集合。 FRAME 开发人员可选择创建自己的 pallet ，也可以重用包括50多个 Substrate 随附 pallet 在内的现有资源。



本节大部分内容将会重点介绍使用如何使用 FRAME 进行 Substrate runtime 开发。但是请记住，FRAME 并不是 Substrate runtime 开发的唯一选择。

### 6.3.2 宏

* **介绍**

Substrate 的 runtime 宏使开发人员可以专注于 runtiem 逻辑，不必顾及链上变量的编码和解码，使得开发人员经常利用 runtime 宏带来的代码简洁性优化 runtime 开发。 但有时候，开发人员可能会碰到一些奇怪的错误，源于 runtime 宏新创建的结构体和类型，与宏内其它内容交互时产生。

本文的目的是对宏作一个概述，并对 runtime 工程师经常遇到的 Substrate 宏作详细解释。

* **宏基础知识**

在Rust语言中有四种类型的宏：

* 用 macro\_rules!定义的声明宏
* 作为 自定义生成的过程宏
* 作为 类属性的过程宏
* 作过 类函数的过程宏

大多数 Substrate runtime 宏都是使用声明宏或类函数宏定义的。

### 6.3.3 元数据

Substrate 区块链会将元数据暴露，以便与其交互。元数据根据不同的 pallet 来源分隔成各种模块，元数据为每个模块提供对外暴露的 存储项、extrinsic调用、事件、常量和错误的相关信息。Substrate 会自动生成这些元数据，并通过 RPC 函数让它可被调用。

区块链状态不断发展，Substrate 区块链的 runtime 是其一部分，因此区块链元数据是按区块存储的。请注意，查询旧区块 (例如使用存档节点) 获取的元数据可能是过时的，与区块链当前状态不兼容。正如升级文档所述，Substrate 链开发者可以认为，「仅当」链的 runtime spec\_version 发生了变化，链的元数据才会发生变化。

* **如何获取元数据**

从 Substrate 节点中获取元数据，可使用相应语言库或与语言无关的 HTTP 和 WebSocket API 这两种渠道。

**Rust**

获取元数据的最简单方法是查询自动生成的 JSON-RPC 函数 state\_getMetadata ， 这将返回 SCALE 编码字节的向量。 解码可使用 frame-metadata 和 parity-scale-codec 库。

一些有用的库（例如 substrate-subxt）能获取元数据并解码， 然后再使用 serde 将结构序列化为 JSON。如果你喜欢更直接地使用RPC，则可采用 JSONRPC 和 jsonrpsee Rust 库的相应接口。

**Javascript**

如果你使用的是 Javascript ，那么 polkadot-js/api 已经提供了与 Substrate 区块链进行交互的 API，其中包括 getMetadata 这个函数。

你可以尝试用以下代码块到 Substrate UI页面上获取元数据：

|  |
| --- |
| const { magicNumber, metadata } = await api.rpc.state.getMetadata();  console.log("Magic number: " + magicNumber);  console.log("Metadata: " + metadata.raw); |

**HTTP 和 WebSocket APIs**

Substrate 节点暴露了 JSON-RPC API ，可通过 HTTP 或 WebSocket 请求访问。从节点请求元数据的消息格式如下所示：

|  |
| --- |
| {  "id": 1,  "jsonrpc": "2.0",  "method": "state\_getMetadata",  "params": []  } |

params 可为空，如果需要获取指定区块链元数据，需要提供该块的哈希值作为参数。

|  |
| --- |
| {  "id": 1,  "jsonrpc": "2.0",  "method": "state\_getMetadata",  "params": ["0xca15c2f1e1540517697b6b5f2cc6bc0c60876a1a1af604269b7215970798bbed"]  } |

在上面的示例当中，

0xca15c2f1e1540517697b6b5f2cc6bc0c60876a1a1af604269b7215970798bbed是1,768,321区块的哈希值。

请求的返回结果格式如下：

|  |
| --- |
| {  "jsonrpc": "2.0",  "result": "0x6d6574610b7c1853797374656d011853797374656d3c1c4163636f756e7401010230543a3a4163636f756e744964944163...",  "id": 1  } |

result 字段包含了区块链元数据，是用 SCALE 编码的十六进制字符。上面的示例是从区块1,768,321返回的实际值，你可使用 WebSocket 客户端去查询节点作对比。 继续阅读以了解有关此编码 Blob 的格式及其解码格式的更多信息。

* **元数据格式**

本节将简要回顾以十六进制字符串表示的 SCALE 编码元数据，然后再更详细地查看元数据的解码格式。

JSON-RPC 的 state\_getMetadata 方法返回的十六进制 blob，以硬编码的魔术数字 0x6d657461 开头，在纯文本中为“元”。下一条数据(上例中0x0b) 为元数据版本，解码十六进制值 0x0b 会得到十进制值11，这是 Substrate元数据格式版本的编码，接下来的信息表示组成区块链 runtime 的 pallet 数量。上例中十六进制值 0x7c 代表十进制数字31，将其二进制(11111或十六进制的 0x1F)向左移动两位(1111100)，并 SCALE 编码后形成十六进制数值。

剩余的 Blob 对每个 pallet 的元数据进行了编码，我们将在下面对这部分内容展开详细说明。extrinsic 元数据不在本文的范围中

**解码后的元数据格式**

以下为解码后的元数据的精简版本：

|  |
| --- |
| {  "magicNumber": 1635018093,  "metadata": {  "V12": {  "modules": [  {  // ...  },  {  // ...  }  ],  "extrinsic": {  "version": 4,  "signedExtensions": [  "CheckSpecVersion",  "CheckTxVersion",  "CheckGenesis",  "CheckMortality",  "CheckNonce",  "CheckWeight",  "ChargeTransactionPayment"  ]  }  }  }  } |

如上所述，整数 1635018093 在纯文本中是代表“元”的“魔法数字”。剩余的数据内容分为两部分： modules部分和 extrinsic。modules 部分包含有关 runtime pallet 的信息，而 extrinsic 部分描述了 runtime 正在使用的 extrinsics 版本。不同的 extrinsic 版本可能具有不同格式，尤其是考虑到 signed extrinsics的时候。

**Modules**

以下为 modules 数组中单个元素的精简示例：

|  |
| --- |
| {  "name": "System",  "storage": {  // ..  },  "calls": [  // ..  ],  "events": [  // ..  ],  "constants": [  // ..  ],  "errors": [  // ..  ],  "index": 0  } |

每个元素都包含了它代表的 pallet 名称，以及一个 storage 对象、calls 数组、 event 数组和 errors 数组。

注意：如果 calls或 event为空，则为 null。 如果constants 或 errors 为空，则为空数组。

**Storage**

为了突出模块的存储相关元数据，以下是 modules 数组中单个元素的精简示例，

|  |
| --- |
| {  "name": "System",  "storage": {  "prefix": "System",  "items": [  {  "name": "Account",  "modifier": "Default",  "type": {  "Map": {  "hasher": "Blake2\_128Concat",  "key": "AccountId",  "value": "AccountInfo",  "linked": false  }  },  "fallback": "0x000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000",  "documentation": [  " The full account information for a particular account ID."  ]  },  {  "name": "ExtrinsicCount",  // ..  },  {  "name": "AllExtrinsicsLen",  // ..  }  ]  },  "calls": [/\*...\*/],  "events": [/\*...\*/],  "constants": [/\*...\*/],  "errors": [/\*...\*/],  "index": 0  } |

Pallet 中定义的每个存储项都有一个对应的元数据条目。例如，Account 存储项是在 frame-system 中由下面这段代码生成的：

|  |
| --- |
| decl\_storage! {  trait Store for Module<T: Trait> as System {  /// 某账户ID的完整账户信息。  pub Account get(fn account):  map hasher(blake2\_128\_concat) T::AccountId => AccountInfo<T::Index, T::AccountData>;  }  } |

存储元数据为区块链客户端提供了查询 Json-Rpc 的存储函数，以获取特定存储项内容。

**Calls**

模块元数据包含在 decl\_module! 的宏中定义的 runtime 可调用函数的信息。对于每个可调用函数，元数据包括：

name: 模块中函数的名称。

args: 函数定义里的参数，包括每个参数的名称和类型。

Documentation: 函数的文档说明。

举个例子，以下来自于Timestamp pallet：

|  |
| --- |
| decl\_module! {  pub struct Module<T: Trait> for enum Call where origin: T::Origin {  // ... snip  /// Set the current time.  ///  /// This call should be invoked exactly once per block. It will panic at the finalization  /// phase, if this call hasn't been invoked by that time.  ///  /// The timestamp should be greater than the previous one by the amount specified by  /// `MinimumPeriod`.  ///  /// The dispatch origin for this call must be `Inherent`.  #[weight = (  T::DbWeight::get().reads\_writes(2, 1) + 9\_000\_000,  DispatchClass::Mandatory  )]  fn set(origin, #[compact] now: T::Moment) {  // ... snip  }  }  } |

在元数据中表示如下：

|  |
| --- |
| "calls": [  {  "name": "set",  "args": [  {  "name": "now",  "ty": "Compact<T::Moment>"  }  ],  "documentation": [  " Set the current time.",  "",  " This call should be invoked exactly once per block. It will panic at the finalization",  " phase, if this call hasn't been invoked by that time.",  "",  " The timestamp should be greater than the previous one by the amount specified by",  " `MinimumPeriod`.",  "",  " The dispatch origin for this call must be `Inherent`."  ]  }  ], |

**Events**

这个元数据片段由 frame-system 中的以下列代码生成：

|  |
| --- |
| decl\_event!(  /// 系统模块事件  pub enum Event<T> where AccountId = <T as Trait>::AccountId {  /// extrinsic 成功完成。  ExtrinsicSuccess(DispatchInfo),  /// extrinsic 失败。  ExtrinsicFailed(DispatchError, DispatchInfo),  // ... snip  }  ) |

Substrate 的元数据将会如下描述这些事件：

|  |
| --- |
| "event": [  {  "name": "ExtrinsicSuccess",  "arguments": [  "DispatchInfo"  ],  "documentation": [  " An extrinsic completed successfully."  ]  },  {  "name": "ExtrinsicFailed",  "arguments": [  "DispatchError",  "DispatchInfo"  ],  "documentation": [  " An extrinsic failed."  ]  },  ], |

**Constants**

元数据包含任何的模块常量。pallet-babe 里：

|  |
| --- |
| decl\_module! {  /// BABE Pallet  pub struct Module<T: Trait> for enum Call where origin: T::Origin {  /// 一个纪元（时间段）所需要的槽数。我们将会话与纪元耦合在一起，  /// 也就是说，一旦新的纪元开始，我们就开始一个新的会话。  const EpochDuration: u64 = T::EpochDuration::get();  // ... snip  }  } |

该常量的元数据如下所示：

|  |
| --- |
| "constants": [  {  "name": "EpochDuration",  "type": "u64",  "value": "0x6009000000000000",  "documentation": [  " The number of \*\*slots\*\* that an epoch takes. We couple sessions to",  " epochs, i.e. we start a new session once the new epoch begins."  ]  },  // ...  ] |

元数据还包括 runtime 的 lib.rs 文件中定义的常量。举个例子，Kusama 中：

|  |
| --- |
| parameter\_types! {  pub const EpochDuration: u64 = EPOCH\_DURATION\_IN\_BLOCKS as u64;  } |

其中 EPOCH\_DURATION\_IN\_BLOCKS 是在 runtime/src/constants.rs 中定义的常数。

**Errors**

元数据会从 decl\_error! 宏获取所有可能的 runtime 错误。举个例子，frame-system 中：

|  |
| --- |
| decl\_error! {  /// 系统模块出错  pub enum Error for Module<T: Trait> {  /// 当前 runtime 和新 runtime 之间的描述名称不匹配  InvalidSpecName,  // ... snip  }  } |

这将会对外暴露以下元数据：

|  |
| --- |
| "errors": [  {  "name": "InvalidSpecName",  "documentation": [  " The name of specification does not match between the current runtime",  " and the new runtime."  ]  },  // ...  ] |

这些都是 extrinsic 提交或执行过程中可能发生的错误。

### 6.3.4 存储

Runtime 存储机制可使数据持久地存储在区块链中，并在 runtime 逻辑中能被访问。“存储应该是区块链 runtime 开发人员最为关注的点之一”，这种说法不言自明，因为区块链的主要目标之一就是为底层存储状态提供去中心化共识。此外，设计良好的存储系统可减少网络中节点的负载，从而降低区块链参与者的开销。Substrate 提供了分层的模块化存储API，使 runtime 开发人员能根据自身情况作出合适的存储决策。但同时请记住，区块链 runtime 存储的基本原则是尽可能少的使用链上存储。

* **存储项**

FRAME support中的 storage 模块使得 runtime 开发人员可灵活访问 Substrate 的存储API。 只要是Parity SCALE codec 编码支持的值都可以传入存储 API：

* Storage Value - 单一的值
* Storage Map - 一组键值对哈希映射
* Storage Double Map - 实现了两个键的映射，提供了高效删除第一个键相同的所有条目的重要功能。

存储项类型应取决于 runtime 使用该值的逻辑方式。

**存储值**

如果一个值需要在 runtime 里被视作为一个整体，则应该选用这种存储项类型，它可以是一个基础类型值、一个 结构体，或者是相关项的一个集合。将相关项包裹在一个共享结构体中，是减少存储读取次数的绝佳方法(非常重要的考虑因素)，但当结构体的大小增长到一定程度，它的使用成本可能远远超过减少存储读取次数所带来的好处。存储值也可用来存储集合，但 runtime 开发人员应当格外关注集合的大小。因为大型集合产生的存储成本就像大型的结构体一样高。此外，如果在 runtime 中迭代大型列表，可能会导致处理时间超出区块生产时间 —— 如果发生这种情况，区块链将停止产生区块，相当于区块链停止运行了。

**方法**

请参阅存储值的相关文档以获取 存储值对外暴露的完整方法列表 。 下面总结了其中一些最重要的方法：

* get() - 从存储中读取值。
* put(val) - 保存提供的值。
* mutate(fn) - 使用提供的函数修改值。
* take() - 加载值并把它从存储中移除。
* **存储映射**

映射数据结构非常适合管理随机访问的项目集，但并不适用于管理需要按顺序遍历的项目集。Substrate 中的 Storage Maps 通过键值对哈希映射来实现，这是大多数开发人员都熟悉的模式。为了给予区块链工程师更多控制权，Substrate 允许开发人员选择用于生成 map 中的 key 的哈希算法。请参阅进阶存储文档，以了解更多关于如何实现Substrate的存储映射的信息。

**方法**

Storage Maps 暴露了一系列 API 与存储值的 API 很类似。

* get - 从存储中查询对应 key 关联的值。 Docs: StorageMap#get(key), StorageDoubleMap#get(key1, key2)
* insert - 存储提供的值，并关联给定键。 Docs: StorageMap#insert(key, val), StorageDoubleMap#insert(key1, key2, val)
* mutate - 使用提供的函数来更改给定键的关联值。 Docs: StorageMap#mutate(key, fn), StorageDoubleMap#mutate(key1, key2, fn)
* take - 加载与给定键关联的值，并将其从存储中删除。 Docs: StorageMap#take(key), StorageDoubleMap#take(key1, key2)

**可迭代映射存储**

Substrate 映射存储可以遍历键和值。由于映射通常用于存储无大小限制的数据集(例如帐户余额)，如果在 runtime 中对整个映射进行遍历，很容易超出区块生产时间。此外，与访问本地列表元素相比，访问映射的元素需要更多数据库读取操作，因此遍历映射的时间明显比遍历列表的时间要多得多 。这并不是说在 runtime 中遍历映射是“错误的”。Substrate主要想表达的是"首要原则"，而不是简单粗暴的对与错。区块链 runtime 的高效是 Substrate 的首要原则之一，这些信息旨在帮助你了解所有 Substrate的存储功能，并在坚持首要原则的基础上使用它。

**可迭代映射存储方法**

Substrate 的可迭代映射存储接口定义了如下方法。要注意的是，在可迭代双映射存储中，iter 和 drain 方法要求传入一个参数，即第一个键：

* iter - 不按特定顺序枚举出映射中所有的元素。 如果在遍历映射时同时修改映射，则会返回异常结果。 Docs: IterableStorageMap#iter(), IterableStorageDoubleMap#iter(key1)
* drain - 从映射中移除所有元素，并以不固定的顺序遍历它们。 如果在运行 drain 方法时同时给映射增加新元素，则会返回未知结果。 Docs: IterableStorageMap#drain(), IterableStorageDoubleMap#drain(key1)
* translate - 使用提供的函数并以不特定顺序来转换映射中的所有元素。如果想从映射里删除一个元素，则需要在 translation 函数内返回 None。 Docs: IterableStorageMap#translate(fn), IterableStorageDoubleMap#translate(fn)

**哈希算法**

如上文所述，Substrate 映射存储的一个特性是，允许开发人员指定生成映射中 key 的哈希算法。 用于封装哈希计算逻辑的 Rust 对象称为"hasher"。 广义上讲，供 Substrate 开发人员使用的 "hasher" 可以用两种方式来描述：它们是否是密码学安全的，以及是否产生透明的输出。为了完整起见，下文也会介绍非透明哈希算法的特征。但是请记住，我们不推荐在基于 FRAME 的区块链内使用任何产生不透明输出的哈希算法。

**加密哈希算法**

加密哈希算法的原理，是使用密码学减少通过输入内容的控制对输出结果施加影响的情况。举个例子，即使输入只有简单的数字如1-10，加密哈希算法也会产生广泛分布的输出。因此，在用户能控制存储映射键的情况下，使用加密哈希算法尤其重要。否则，可能存在潜在的攻击载体，很容易被作恶者利用，来降低区块链网络的性能。其中一个推荐用加密哈希算法生成映射键的情景，就是用映射追踪用户余额。这时加密哈希算法尤其重要，攻击者就无法通过向多个连续帐号进行小额转账来轰炸系统。而如果没有加密哈希算法，产生的不平衡存储结构将影响系统性能。但加密哈希算法比非加密哈希算法更复杂且耗费资源更多，这就是 Substrate 允许开发人员自由选择使用时机的原因。

**透明哈希算法**

透明哈希算法是在给定输出结果的情况下，用于挖掘和验证输入内容的简单算法。Substrate 通过连接算法的输入与输出，使哈希算法变得透明。这使用户可以轻易获取 key 对应的原始未哈希的值，并重新哈希来验证。Substrate 的核心开发人员在基于 FRAME 的 runtime 中，已经废弃了不透明哈希算法的使用，此处的内容主要出于完整性的考虑而呈现。实际上，如果要获得 iterable map 的功能，则「必须」使用透明的哈希算法。

**常见的Substrate哈希算法**

下表列出了Substrate中使用的一些常见哈希算法，并指出了哪些是加密的和透明的：

|  |  |  |
| --- | --- | --- |
| Hasher | 密码学安全 | 是否透明 |
| Blake2 128 Concat | X | X |
| TwoX 64 Concat |  | X |
| Identity |  | X |

Identity hasher 封装了这样一个哈希算法，该算法的输出等于其输入(identity函数)。只有当key已经是加密哈希值时，才使用此类型的 hasher 。

* **声明存储项**

可使用 decl\_storage 宏轻松创建新的 runtime 存储项。以下示例了如何声明各种类型的存储项：

|  |
| --- |
| decl\_storage! {  trait Store for Module<T: Trait> as Example {  SomePrivateValue: u32;  pub SomePrimitiveValue get(fn some\_primitive\_value): u32;  // 类型可以使用通用的 `T: Trait`  pub SomeComplexValue: T::AccountId;  pub SomeMap get(fn some\_map): map hasher(blake2\_128\_concat) T::AccountId => u32;  pub SomeDoubleMap: double\_map hasher(blake2\_128\_concat) u32, hasher(blake2\_128\_concat) T::AccountId => u32;  }  } |

请注意，映射的存储项指定了将要使用的哈希算法。

**可见性**

在上面的示例中，除 SomePrivateValue 外，所有存储单元都通过 pub 关键字来表明它们公开可见。事实上区块链上的存储在 runtime「之外」始终公开可见，Substrate 存储项的可见性，仅影响到是否被 runtime「内 」其他 pallet 访问。

**Getter方法**

decl\_storage 宏提供了一个可选的 get 扩展，接收一个 getter 函数名作为参数，用来实现获取模块内存储项对应存储内容的getter方法。不使用它，你仍可获取存储项中的值，只是无法通过模块内定义 getter 方法的方式，需要使用 存储项的get方法。请记住，可选的 get 扩展仅影响从 Substrate 代码中访问该存储项的方式，而通过查询 runtime 存储状态的方式是一直能获取所有存储值的。

以下示例为存储值 SomeValue 实现 some\_value 的 getter 方法。现在这个模块访问存储值，除了可通过 SomeValue::get() 外，还可以通过 Self::some\_value() ：

|  |
| --- |
| decl\_storage! {  trait Store for Module<T: Trait> as Example {  pub SomeValue get(fn some\_value): u64;  }  } |

**默认值**

Substrate 允许自定义默认值，即当存储单元未设置返回值。默认值可以被 runtime 代码使用，但并不占用 runtime 的实际存储空间。

以下示例为映射中的所有条目指定默认值：

|  |
| --- |
| decl\_storage! {  trait Store for Module<T: Trait> as Example {  pub SomeMap: map u64 => u64 = 1337;  }  } |

**创世配置**

Substrate 的 runtime 存储 API 提供在创世区块初始化存储项的功能。创世存储配置 API 对外暴露了许多初始化存储的机制，这些机制在 decl\_storage 宏中都可以定义。这些机制催生 GenesisConfig 数据类型，它实现了接口 BuildModuleGenesisStorage特性 ，并且被添加到包含此存储项的模块里 (比如 Struct pallet\_balances::GenesisConfig)，被标记为创世配置的存储项在该数据类型上有一个相应的属性。为了能够使用模块的创世配置功能，在用 construct\_runtime macro 宏添加相应模块时，必须包含进 Config 元素。各个模块的 GenesisConfig 类型被聚合到整个 runtime 的 GenesisConfig 类型里。这一类型实现了 BuildStorage特性 (例如 Struct node\_template\_runtime::GenesisConfig )，该类型上的每个属性都对应着 runtime 某模块的 GenesisConfig。最终 runtime 的 GenesisConfig 通过 ChainSpec特性 的方式暴露出来。如果想获取关于使用 Substrate 的创世存储配置功能的完整示例，请参阅 Society pallet 中的 decl\_storage 宏，以及 Substrate 代码库附带的链规范文件中 Society pallet 存储的创世配置。继续阅读，详细了解这些功能。

**config**

当使用 decl\_storage 宏声明存储项时，可使用一个可选的 config 扩展名，该扩展名会在 pallet 的 GenesisConfig 数据类型中添加一个属性，属性的值即为创世区块存储项的初始值。config 扩展接收一个参数，用来表示在 GenesisConfig 数据类型上的属性名称。如果同时提供了 get扩展 ，则此参数是可选的( 属性的名称默认为 get 函数的名称)。

以下示例展示了如何将 config 扩展与名为 MyVal 的存储值一起使用，在所属模块的 GenesisConfig 数据类型上，创建一个名为 init\_val 的属性。随后又演示了如何使用 GenesisConfig 类型在区块链的创世区块中设置存储初始值。

在 my\_module/src/lib.rs中:

|  |
| --- |
| decl\_storage! {  trait Store for Module<T: Trait> as MyModule {  pub MyVal get(fn my\_val) config(init\_val): u64;  }  } |

在 chain\_spec.rs中:

|  |
| --- |
| GenesisConfig {  my\_module: Some(MyModuleConfig {  init\_val: 221u64 + SOME\_CONSTANT\_VALUE,  }),  } |

**build**

开发者通过 decl\_storage 宏的 config 扩展可在链描述文件中配置模块创世存储状态，而 build 扩展可在存储单元对应模块内实现同样的功能 (可以访问模块的私有函数)，和 config 一样，build 扩展接受单个参数，不同的是该参数是必需的，而且必须是闭包，闭包本质上就是一个函数。调用 build 闭包时需要传入一个参数，类型是本模块的 GenesisConfig (方便访问 GenesisConfig 类型的所有属性)。可以将 build 扩展和 config 扩展一起用于单个存储项，在这种情况下，当前 pallet 的 GenesisConfig 类型会有一个属性，属性名由config设置，对应的值在链描述中设置，存储项的初始值由 build 闭包的返回值确定。

下面是一个演示使用 build 设置存储项初始值的例子。该示例涉及两个存储项：一个是会员帐户 ID 的列表，另一个表示此列表中的特殊会员，即高级会员。会员列表由 config 扩展提供，会员列表由 config 扩展提供，而高级会员假设为成员列表中的第一个元素，通过 build 扩展来设置。

在 my\_module/src/lib.rs中:

|  |
| --- |
| decl\_storage! {  trait Store for Module<T: Trait> as MyModule {  pub Members config(orig\_ids): Vec<T::AccountId>;  pub Prime build(|config: &GenesisConfig<T>| config.orig\_ids.first().cloned()): T::AccountId;  }  } |

在 chain\_spec.rs中:

|  |
| --- |
| GenesisConfig {  my\_module: Some(MyModuleConfig {  orig\_ids: LIST\_OF\_IDS,  }),  } |

**add\_extra\_genesis**

decl\_storage 宏的 add\_extra\_genesis 扩展定义了一个作用域，可同时使用 config 和 build 扩展，且无需将它们绑定到特定的存储项。可在 add\_extra\_genesis 作用域内，通过 config 向当前 pallet 的 GenesisConfig 数据类型添加一个属性，该属性可以在任意 build 闭包中使用。在该作用域内定义的 build 闭包可用于执行逻辑，而无需将该逻辑的返回值与特定存储项的值进行绑定。如果希望在模块内调用私有助手函数来设置多个存储项，或者在本模块中调用其他模块上定义的功能，都适合采用这种用法。

下例封装了与上面 build 例子中相同的用例：一个维护会员帐户ID列表以及指定高级会员的模块。 然而此处 add\_extra\_genesis 扩展定义的 GenesisConfig 属性，不绑定到特定存储项。除此之外，该扩展还定义了一个 build 闭包，用于本模块内调用私有函数来设置多个存储项值。 鉴于本示例的目的，私有助手函数 (initialize\_members) 的实现就留给大家想象了。

在 my\_module/src/lib.rs中:

|  |
| --- |
| decl\_storage! {  trait Store for Module<T: Trait> as MyModule {  pub Members: Vec<T::AccountId>;  pub Prime: T::AccountId;  }  add\_extra\_genesis {  config(orig\_ids): Vec<T::AccountId>;  build(|config| Module::<T>::initialize\_members(&config.members))  }  } |

在 chain\_spec.rs中:

|  |
| --- |
| GenesisConfig {  my\_module: Some(MyModuleConfig {  orig\_ids: LIST\_OF\_IDS,  }),  } |

* **访问存储项**

基于 Substrate 构建的区块链对外暴露了一个远程过程调用(RPC) 服务，该服务可用于查询区块链的链上存储。 可使用 Polkadot JS之类的软件库，用代码轻松地与 RPC 服务器进行交互并访问存储项。 Polkadot JS团队还维护着 Polkadot Apps UI，这是一个功能齐全的 Web 应用程序，用来与 Substrate 区块链交互和查询存储。请参阅进阶存储文档，以了解 Substrate 如何使用键值对数据库实现不同类型的存储项，以及如何通过 RPC 服务器直接查询此数据库。

* **最佳实践**

Substrate 的目标是提供一个灵活的框架，使人们可以构建适合其需求的区块链，Substrate 的核心开发者有意不以“对”或“错”的角度来思考如何构建区块链。这意味着，Substrate代码库遵循许多最佳实践，以长期促进安全、高效且可维护的区块链网络的建设。以下各节概述了使用 Substrate 存储的最佳实践，并描述了推动它们发展的首要原则。

**存储什么**

请记住，区块链 runtime 存储的基本原则是最大程度减少链上空间的使用。仅对「共识强依赖」的数据才应存储在 runtime中。如果可以，请尽可能使用如元数据哈希之后的结果来减少存储的数据量。例如，许多 Substrate 治理相关功能 (例如 Democracy pallet 的 propose 可调用函数) 允许网络参与者对可调用函数的「哈希」进行投票，哈希值大小是有界的，但函数本身内容大小可能无界。其中一种场景就是对 Runtime 进行升级时，可调用函数需要接收整个 runtime Wasm 数据块作为其参数。由于这些治理机制是 「在链上」实现的，让提案的状态达成共识需要的完整信息，也必须在链上存储-包括投票的 「内容」。不过，Substrate 的治理机制使用了一种巧妙的方式，将链上提案与其哈希值进行绑定，把提案相关完整数据上链的时间，推迟到了提案通过「之后」，意味着提案不会浪费存储空间。一旦提案通过，相关人员就会对实际的可调用函数本身(连同其所有参数) 进行重新哈希，与提案中的哈希值比较，确认无误后便可发起调用。另一种使用哈希最小化链上存储数据的模式，是在 IPFS 中存储元数据，链上仅需存储 IPFS 地址 (有界的哈希值) 就可以了。

哈希只是控制 runtime 存储大小机制的其中一种， 另一种例子是 边界。

**先验证，后写入**

发送交易之前的状态 Substrate 不会缓存，而是在调用时直接把更改上链。因此，如果交易失败，任何状态的更改都会保留下来。由于这个原因，在确定已满足所有前提条件之前，请勿进行任何存储更改。一般而言，用于修改存储的代码块应该构造如下：

|  |
| --- |
| {  // 这里是所有检查和抛出的代码块  // \*\* 以下无抛出 \*\*  // 这里是所有发出事件 & 存储  } |

不要用 runtime 存储来记录原子性操作的中间数据或瞬态数据，或操作失败时不再需要的数据。这并不意味着不能使用 runtim 存储来跟踪需要多个原子操作才能完成的动作，例如 Utility pallet 的多重签名功能。 在这个例子中，即便永远没有足够多的签名，让给定的函数调用实际发起，runtime 存储还是用来跟踪可调用函数获取的签名，每个签名都被视为多重签名操作中一个原子事件，只有满足了与该签名相关的所有前提条件，才会记录签名数据。

**创建边界**

约束存储项的大小在控制 runtime 存储的使用中非常有效，因此在整个 Substrate 代码库中多次重复使用了这种方法。一般而言，任何由用户操作产生的存储项都应设置边界。Utility pallet 多重签名功能就是其中一个例子，其中，与多重签名操作相关的签名者列表由多重签名参与者提供。要使多重签名的状态达成共识，必然用到签名者列表，因此也必然要将其存储在 runtime 中。但是，为了使 runtime 开发人员能够控制这些列表最大可占用的存储空间，Utility pallet 要求用户配置一个边界数字，该约束将作为写入存储的前提条件进行检查。

### 6.3.5 交易发起者(Origin)

可调用函数使用 runtime origin 来检查调用的来源。

* **Raw Origins**

Substrate定义了可在runtime模块中使用的三个raw origins：

|  |
| --- |
| pub enum RawOrigin<AccountId> {  Root,  Signed(AccountId),  None,  } |

* Root: 系统级的origin， 它拥有最高级别的权限。
* Signed: 交易origin， 它被某公钥签名过，包含了签名者的ID。
* None: origin缺失， 它需要由验证人同意或者由特殊模块验证之后才能被包含到区块中。
* **自定义Origin调用**

可在runtime中使用任意origin来构建调用。 例如：

|  |
| --- |
| // Root  proposal.dispatch(system::RawOrigin::Root.into())  // Signed  proposal.dispatch(system::RawOrigin::Signed(who).into())  // None  proposal.dispatch(system::RawOrigin::None.into()) |

* **自定义Origins**

除了 3 种核心Origin类型外，运行时间开发人员还能够定义自定义Origins。这些功能可用作授权检查运行中特定模块的内部功能，或定义围绕运行时间请求源的自定义访问控制逻辑。自定义Origins允许运行时间开发人员根据其运行时间逻辑指定有效的Origins。例如，最好将某些职能限制在特殊自定义Origins的访问，并仅授权来自集体成员的调度呼叫。使用自定义Origins的优点是，它为运行时开发人员提供了一种配置特权访问的方式，而不是向运行时间发送呼叫。

### 6.3.6 执行器

Substrate runtime 的执行由 Executive 模块协调。

与 FRAME 中其他模块不同，Executive 不是「runtime」模块， 而是一个普通的 Rust 模块，它负责调用区块链中各种 runtime 模块。

Executive 模块对外暴露了 execute\_block 函数，实现如下功能：

* 初始化区块
* 执行extrinsics
* 完结区块
* **验证交易**

区块开始执行前，检查签名交易有效性。 这个行为不会产生任何附带反应，仅用来确保无论交易能否打包到区块，都不会造成程序崩溃，因此将丢弃存储所作的更改。

* **执行区块**

有效交易队列只要不为空，Executive 模块就开始执行区块。

**初始化区块**

区块初始化时，System 模块和其他 runtime 模块都会先调用其 on\_initialize 函数，在交易执行前把模块定义的、需要前置的业务逻辑全部处理掉。 除了总是优先处理 System 模块外，其余模块的执行顺序遵循 construct\_runtime! 宏内的定义。

接下来初始检查，该步骤将验证区块头中父哈希是否正确，以及 extrinsics trie 的根是否包括全部 extrinsics。

**执行Extrinsics**

区块初始化完成以后，将按交易优先级顺序执行每个有效的 extrinsic。 Extrinsics 一定不能在 rutnime 逻辑中引起程序崩溃，否则用户可不受任何惩罚地消耗计算资源，攻击系统很容易。

当 extrinsic 执行时，修改将直接应用到存储，原有存储状态不会提前缓存。因此，在更改存储状态之前，runtime 开发人员应充分检查，确保 extrinsic 能执行成功。一旦 extrinsic 在执行过程中失败，存储更改无法回滚。

extrinsic 执行时触发的事件 也会写入存储。 因此，在完成所有待执行动作之前，不应该触发相关事件，避免事件触发后 extrinsic 才执行失败，导致事件不回滚。

**完结区块**

执行完队列中所有的 extrinsics 后，Executive 模块将调用每个模块的 on\_finalize 函数，以运行定义在区块末端的最终业务逻辑。 所有模块也将再次按照在 construct\_runtime! 宏中定义的顺序来执行，但 System 模块须等到最后完成。

接下来是最终检查，验证区块头中的摘要和存储根是否匹配计算结果。

### 6.3.7 事件

Substrate runtime 模块通过触发事件，向外部实体（例如用户，区块链链浏览器或dApps）通知 runtime 中的变化或状况。

模块所能触发的事件类型、事件中包含的信息，以及事件触发的时间都可以自定义。

* **声明事件**

Runtime 事件由 decl\_event! 宏创建。

|  |
| --- |
| decl\_event!(  pub enum Event<T> where AccountId = <T as Trait>::AccountId {  /// Set a value.  ValueSet(u32, AccountId),  }  ); |

Event 枚举类型需要在 runtime 的配置特性中进行声明。

|  |
| --- |
| pub trait Trait: system::Trait {  type Event: From<Event<Self>> + Into<<Self as system::Trait>::Event>;  } |

* **向runtime暴露事件**

模块中的事件需要暴露给 Substrate 的 runtime (/runtime/src/lib.rs)。

首先，要在模块的配置特性中定义事件类型：

|  |
| --- |
| // runtime/src/lib.rs  impl template::Trait for Runtime {  type Event = Event;  } |

然后再将该 Event 类型添加到 construct\_runtime! 宏里:

|  |
| --- |
| // runtime/src/lib.rs  construct\_runtime!(  pub enum Runtime where  Block = Block,  NodeBlock = opaque::Block,  UncheckedExtrinsic = UncheckedExtrinsic  {  // -- snap --  TemplateModule: template::{Module, Call, Storage, Event<T>},  //--增加这个------------------------------------->^^^^^^^^  }  ); |

* **保存事件**

Substrate 提供了一个默认实现，来存储在 decl\_module! 宏内定义的事件。

|  |
| --- |
| decl\_module! {  pub struct Module<T: Trait> for enum Call where origin: T::Origin {  // `deposit\_event` 的默认实现  fn deposit\_event() = default;  fn set\_value(origin, value: u64) {  let sender = ensure\_signed(origin)?;  // --snip--  Self::deposit\_event(RawEvent::ValueSet(value, sender));  }  }  } |

该函数默认从 FRAME 系统中调用 deposit\_event，把事件写入存储中。

函数会把该区块的事件存至 System 模块的 runtime 存储中，新区块开始后，System 模块都会自动删除上一个区块中存储的所有事件。

下游支持库如 Polkadot-JS api 直接支持通过默认实现保存的事件，但如果想用另外的方式，也可实现自定义的 deposit\_event 函数。

* **支持类型**

事件可以输出任何可被 Parity SCALE codec 编解码的类型。

如要使用诸如 AccountId 或 Balances 之类的Runtime泛型，则需要包含一个where 语句来定义这些类型，如上例所示。

* **监听事件**

Substrate RPC 不会直接暴露查询事件的端口。 如果使用默认实现保存事件，则可通过查询 System 模块的存储单元来查看当前区块的事件列表。还可以通过Polkadot-JS api 的 WebSocket 来订阅 runtime 事件。

### 6.3.8 错误处理

Runtime 代码应该显式且优雅地处理所有错误情况，也就是说 runtime 代码 必须“不抛出”，用 Rust 术语来讲，就是决不能"panic" 。一般来说，想要编写出“不抛出”的 Rust 代码，就要编写返回值为Result 类型的函数。Result 枚举类型有一个名为 Err 的变量，该变量可让函数传达执行失败的信息，以避免程序“panic”。 FRAME 系统中用于 runtime 开发的可调用函数「必须」返回DispatchResult类型，若可调用函数在执行时遇到错误，则该返回值 「应为」DispatchError。

每个FRAME pallet都可以使用decl\_error! 宏来自定义 DispatchError的内容。

|  |
| --- |
| // Errors 通知用户出了问题  decl\_error! {  pub enum Error for Module<T: Trait> {  /// 错误名称应该是描述性的。  InvalidParameter,  /// 错误应该有相关的有用文档。  OutOfSpace,  }  } |

为了能在 pallet 中触发自定义错误，pallet 的 module 部分必须要配置 Error 类型。

|  |
| --- |
| decl\_module! {  pub struct Module<T: Trait> for enum Call where origin: T::Origin {  // 如 pallet 用了 Errors ，则必须初始化  type Error = Error<T>;  /\* --snip-- \*/  }  } |

Pallet 模版演示了一些正确处理可调用函数内错误的方法。FRAME Support 模块还有个十分有用的ensure! 宏，可用于检查前提条件，在不满足时发出错误信息。

|  |
| --- |
| frame\_support::ensure!(param < T::MaxVal::get(), Error::<T>::InvalidParameter); |

### 6.3.9 交易费用

当出块人构造一个块时，必须要限制该区块的执行时间。区块体由一系列 extrinsics 组成。不同 extrinsic 执行所需的资源可能不同，Substrate 提供了一种称为“权重”的灵活机制，来代表执行某 extrinsic 所需的「时间」。 为了经济上的可持续发展并限制信息泛滥，部分交易（主要是用户发送的交易）在执行之前需要付费。

* **费用计算**

通过可调用函数的权重，和一系列可配置参数来计算函数调用最终费用。

**打包费用**

交易费用由两部分组成：

* length\_fee: 每字节的费用乘以编码后的 extrinsic 长度（以字节为单位）。查看 TransactionByteFee。
* weight\_fee: 基于 extrinsic 权重费用的函数，接收两个入参。第一个参数为 ExtrinsicBaseWeight，它在 runtime 中声明，应用于所有 extrinsics ，涵盖签名验证类的固定开销。第二个参数为可灵活使用的#[weight]注释，来标记 extrinsic 的复杂度。Runtime 必须定义 WeightToFee 来实现Convert功能把权重转换为 Currency 。

如上所述，可调用函数的最终费用为：

|  |
| --- |
| fee =  len(tx) \* length\_fee +  WeightToFee(weight) |

这个 费用 称之为“打包费”。 请注意，打包费是在实际调用 extrinsic 「之前」向发送方收取的，因此如 extrinsic 执行失败，费用仍会产生。如果账户中余额不足以支付费用，或支付费用后余额低于保留余额，则交易也不会执行，也不会扣除任何费用。 这种情况很少见，因为在区块中添加 extrinsic 之前，交易队列和区块构造逻辑已执行了相关检查。

**费用乘数因子**

相同输入，上述公式得出的费用始终相同。然而权重是动态的，并且根据 WeightToFee 的定义方式，使最终费用有一定程度的可变性。为了满足这个要求，Substrate 提供了：

* NextFeeMultiplier:存储在 Transaction Payment 模块中的可配置乘数。
* Trait.html#associatedtype.FeeMultiplierUpdate">FeeMultiplierUpdate: Runtime 的可配置参数，用于描述该乘数的变化方式。

NextFeeMultiplier 的类型为 Fixed64，可以表示固定浮点数。根据上述的打包费公式，最终版本将为：

|  |
| --- |
| fee =  len(tx) \* length\_fee +  WeightToFee(weight)  final\_fee = fee \* NextFeeMultiplier |

更新 NextFeeMultiplier 与更新 WeightToFee 带来的结果类似。交易支付模块中的 FeeMultiplierUplierUpdate 关联类型定义为 Convert<Fixed64, Fixed64>，应为：“它接收前一个乘数并返回下一个乘数”。

默认的更新函数受波卡网络启发，实现了定向调整，其中定义了区块权重的目标饱和度。如果前一个区块相对更饱和，则费用会略增加。同样，如果前一个区块的交易量少于目标交易量，则费用将稍微减少。

* **附加费用**

打包费在 extrinsic 执行之前必须是可计算的，因此只能使用固定逻辑。有些交易则需要使用其他策略来限制资源的使用，例如：

* 保证金：保证金用于某些链上事件触发后，要么退还要么罚没的一种费用。例如，runtime 开发人员希望实现一种保证金机制用于投票。此时，保证金会在投票结束之后正常退还。但如果投票人尝试作恶，则保证金会在投票结束后被罚没。
* 押金：押金是可返还的费用。例如，用户执行占用存储的操作，可能需要支付押金，如果后续操作释放了该存储空间，则返还用户押金将。
* 销毁：交易基于其逻辑在执行中销毁资金。例如，如果交易创建了新的存储条目，增加了链上状态的占用空间，则交易可能会因此销毁请求方发送的资金。
* 限制：runtime 开发人员可自由地对某些操作强制执行恒定的或可配置的限制。例如，默认的 Staking pallet 仅允许1个提名人提名16个验证人，以限制验证人选举过程的复杂性。

重点注意，如果在链中查询交易费用，它只返回打包费。

* **默认权重注释**

Substrate 中的所有可调用函数必须要指定权重。可使用标注系统，结合数据库读/写权重的固定值和基于基准测试的固定值。 最基本的示例如下所示：

|  |
| --- |
| #[weight = 100\_000]  fn my\_dispatchable() {  // ...  } |

请注意，空 extrinsic 被包含到区块存在成本，ExtrinsicBaseWeight 将自动添加到权重声明。

**数据库访问参数化**

为了使权重标注独立于已部署的数据库后端，我们把它们定义为常量，在标注中用于表示可调用函数的数据库使用情况：

|  |
| --- |
| #[weight = T::DbWeight::get().reads\_writes(1, 2) + 20\_000]  fn my\_dispatchable() {  // ...  } |

可调用函数增加了20,000权重，包含数据库的一个读取和两个写入，以及其他一些操作。通常我们所说的一次数据库访问，就是指访问一次声明在 decl\_storage! 代码块中的值。然而，由于某个值被访问了一次后就会被缓存，而再次访问则不会导致重复的数据库操作。因此，只有第一次访问会被计入权重。 这意味着：

* 多次读取相同的值只计算一次读取权重。
* 多次写入相同的值只计算一次写入权重。
* 多次读取相同值并随后对该值进行写入，只计算一次读取和一次写入的权重。
* 一次写入并随后跟着一次读取，只计算一次写入的权重。

**可调用函数的类**

可调用函数可分成三个类：Normal , Operational, 和 Mandatory ，如果未在权重标注中额外定义，则默认为 Normal ，开发人员可以指定可调用函数使用其他类，例如：

|  |
| --- |
| #[weight = (100\_000, DispatchClass::Operational)]  fn my\_dispatchable() {  // ...  } |

此元组形式的标注还可以指定最后一个参数，来确定是否向用户收取该标注权重的费用，默认为Pays::Yes ：

|  |
| --- |
| #[weight = (100\_000, DispatchClass::Normal, Pays::No)]  fn my\_dispatchable() {  // ...  } |

**Normal类的可调用函数**

此类的可调用函数表示常规用户触发交易，其消耗不能超过区块总权重上限的一定比例，比例可以通过检查 [AvailableBlockRatio](https://substrate. dev/rustdocs/v2.0.0/frame\_system/trait. Trait. html#associatedtype. AvailableBlockRatio) 来找到。Normal 类可调用函数将被发送到交易池中。

**Operational类可调用函数**

与 normal 类的可调用函数相反，operational 类的可调用函数「提供了」网络功能，而不是「使用了」网络功能。 这些类型的可调用函数有可能会消耗掉整个区块的权重上限，这就是说，他们不受 [AvailableBlockRatio](https://substrate. dev/rustdocs/v2.0.0/frame\_system/trait. Trait. html#associatedtype. AvailableBlockRatio) 的限制，将获得最高优先级，且无需支付 length\_fee。

**Mandatory类的可调用函数**

即使会导致区块超出权重上限，Mandatory 类的可调用函数也必定被包含进区块。这种可调用函数类只能应用于 inherents，且只能用在作为区块验证过程部分的函数上。

由于这种类型的可调用函数无视权重总是包含到区块中，因此验证过程必须防止恶意验证程序滥用函数，同时保证构建出有效且权重不会太高的区块。通常采用的方式是，确保操作始终非常轻，并且每个区块只能被包含一次。为了使恶意验证者更难滥用，在返回错误的情况下，该可调用函数将不能被包含到区块。它的存在出于一个前提：允许创建一个超权重区块比禁止创建任何区块更好。

**动态权重**

除固定权重和常量之外，权重计算还可以加入可调用函数的入参。简单运算输入参数应当可以轻松得出权重：

|  |
| --- |
| #[weight = FunctionOf(  |args: (&Vec<User>,)| args.0.len().saturating\_mul(10\_000),  DispatchClass::Normal,  Pays::Yes,  )]  fn handle\_users(origin, calls: Vec<User>) {  // Do something per user  } |

* **调用后权重校正**

受最终执行逻辑影响，可调用函数实际消耗的权重可能小于调用前的预估。权重校正的重要性在权重篇章里已经解释得非常清楚。为了校正权重，需在可调用函数中声明另一个返回类型，返回实际执行权重：

|  |
| --- |
| #[weight = 10\_000 + 500\_000\_000]  fn expensive\_or\_cheap(input: u64) -> DispatchResultWithPostInfo {  let was\_heavy = do\_calculation(input);  if (was\_heavy) {  // None 表示权重注释 "无校正"。  Ok(None.into())  } else {  // 返回实际消耗的权重  Ok(Some(10\_000).into())  }  } |

* **自定义费用**

也可调整自定义权重函数及打包费用函数，来创建定制化的权重费用系统。

**自定义权重**

可通过创建一种自定义的权重计算类型，来代替上述默认权重注释系统。这种类型必须实现下列特性：

* [WeighData<T>]：确定调度的权重。
* [ClassifyDispatch<T>]：确定调度的类。
* [PaysFee<T>]：确定可调度的发件人是否支付费用。

接着，Substrate 会将以上两个特性的输出信息绑定到 [DispatchInfo] 结构体中，并通过为所有Call 变量和不透明的 extrinsic 类型实现 GetDispatchInfo 特性来提供以上信息。这在 System 和 Executive 模块内部使用，你也许用不上。

ClassifyDispatch 和 WeightData 都是基于泛型 T 的，这里 T 最终会解析为可调用函数所有参数（除origin之外）的元组。作为演示，我们将构建一个结构体来计算权重 m \* len(args) ，其中 m 是给定的乘数，而 args 则是可调用函数的所有参数串联组成的元组。另外，如果交易的入参长度超过100个字节，则该交易的可调用函数类别被定为 Operational ；如果该交易经编码后长度大于10个字节，则交易发送方将需要支付费用。

|  |
| --- |
| use coded::Encode;  use sr\_primitives::weights::{DispatchClass, ClassifyDispatch, WeightData}  // self.0 是乘数 `m`  struct LenWeight(u32);  // 我们不太清楚 T 是什么。毕竟，不同调用有不同的参数， `T` 也会不同。  // 我们关心的是`T`是可以编码的。  // 根据定义，这一点永远为真，所有调度参数都可以编码  impl<T: Encode> WeighData<T> for LenWeight {  fn weigh\_data(&self, target: T) -> Weight {  let multiplier = self.0;  let encoded\_len = target.encode().len() as u32;  multiplier \* encoded\_len  }  }  impl<T: Encode> ClassifyDispatch<T> for LenWeight {  fn classify\_dispatch(&self, target: T) -> DispatchClass {  let encoded\_len = target.encode().len() as u32;  if encoded\_len > 100 {  DispatchClass::Operational  } else {  DispatchClass::Normal  }  }  }  impl<T: Encode> PaysFee<T> {  fn pays\_fee(&self, target: T) -> Pays {  let encoded\_len = target.encode().len() as u32;  if encoded\_len > 10 {  Pays::Yes  } else {  Pays::No  }  }  } |

权重计算函数也可以强制参数的最终类型，而不是将其定义为可编码的模糊类型，关于如何做到这一点， pallet-example 包含了示例。请注意，在这种情况下，代码大致如下所示。

|  |
| --- |
| struct CustomWeight;  impl WeighData<(&u32, &u64)> for CustomWeight {  fn weigh\_data(&self, target: (&u32, &u64)) -> Weight {  ...  }  }  // 给定调用:  decl\_module! {  #[weight = CustomWeight]  fn foo(a: u32, b: u64) { ... }  } |

这意味着 CustomWeight 只能与带有特定签名 (u32, u64) 的可调用函数一起使用，而 Lenweight 则通用得多，因为它没有对 <T> 这个泛型作任何严格的假设。

**自定义打包费用**

这是如何自定义打包费用的示例，必须在相应的模块中配置适当的关联类型。

|  |
| --- |
| use sr\_primitives::{traits::Convert, weights::Weight}  // 假设这是余额类型  type Balance = u64;  // 假设我们希望所有权重都用 `100+2\*w` 转换为费用  struct CustomWeightToFee;  impl Convert<Weight, Balance> for CustomWeightToFee {  fn convert(w: Weight) -> Balance {  let a = Balance::from(100);  let b = Balance::from(2);  let w = Balance::from(w);  a + b \* w  }  }  parameter\_types! {  pub const ExtrinsicBaseWeight: Weight = 10\_000\_000;  }  impl frame\_system::Trait for Runtime {  type ExtrinsicBaseWeight = ExtrinsicBaseWeight;  }  parameter\_types! {  pub const TransactionByteFee: Balance = 10;  }  impl transaction\_payment::Trait {  type TransactionByteFee = TransactionByteFee;  type WeightToFee = CustomWeightToFee;  type FeeMultiplierUpdate = TargetedFeeAdjustment<TargetBlockFullness>;  }  struct TargetedFeeAdjustment<T>(sp\_std::marker::PhantomData<T>);  impl<T: Get<Perquintill>> Convert<Fixed128, Fixed128> for TargetedFeeAdjustment<T> {  fn convert(multiplier: Fixed128) -> Fixed128 {  // 不要改变任何东西。把任何费用更新信息放在这里。  multiplier  }  } |

### 6.3.10 链下工作机

本文将从技术实现层面来介绍如何在 Substrate runtime 中使用链下工作机

* **在 Runtime 中使用链下工作机**

我们可为链下工作机单独创建一个 pallet ，来放置其业务逻辑。在本示例中，我们将此 pallet 称为：my\_offchain\_worker 。它属于 runtime 的一部分，所以源文件目录为：runtime/src/my\_offchain\_worker.rs。

首先，导入下列模块：

|  |
| --- |
| // 打印输出支持更好的调试  use support::{ debug, dispatch };  use system::offchain;  use sp\_runtime::transaction\_validity::{  TransactionValidity, TransactionLongevity, ValidTransaction, InvalidTransaction  }; |

然后，在 pallet 的配置特性中包括以下关联类型，用于链下工作机发送签名和无签名交易。

|  |
| --- |
| pub trait Trait: timestamp::Trait + system::Trait {  /// The overarching event type.  type Event: From<Event<Self>> + Into<<Self as system::Trait>::Event>;  type Call: From<Call<Self>>;  type SubmitSignedTransaction: offchain::SubmitSignedTransaction<Self, <Self as Trait>::Call>;  type SubmitUnsignedTransaction: offchain::SubmitUnsignedTransaction<Self, <Self as Trait>::Call>;  } |

接着，在 decl\_module 宏模块中，定义 offchain\_worker 函数，作为链下工作机的入口，在每次区块导入后运行。

|  |
| --- |
| decl\_module! {  pub struct Module<T: Trait> for enum Call where origin: T::Origin {  // --snip--  fn offchain\_worker(block: T::BlockNumber) {  debug::info!("Hello World.");  }  }  } |

出于安全原因，链下工作机默认无法直接访问用户密钥(包括开发环境)，只能访问应用子密钥。我们需要在 runtime 文件的顶部定义 KeyTypeId ，使其可以分组应用子密钥。定义代码如下所示：

|  |
| --- |
| // key 类型 ID 可以是任何4字符的字符串  pub const KEY\_TYPE: KeyTypeId = KeyTypeId(\*b"abcd");  // --snip--  pub mod crypto {  pub use super::KEY\_TYPE;  use sp\_runtime::app\_crypto::{app\_crypto, sr25519};  app\_crypto!(sr25519, KEY\_TYPE);  } |

和任何其他 pallet 一样，runtime 必须要实现该 pallet 的配置特性。我们进入位于 runtime/src/lib.rs 目录下的 runtime lib.rs 文件，作如下配置：

|  |
| --- |
| // 使用密钥定义来定义交易签名人。  type SubmitTransaction = system::offchain::TransactionSubmitter<  offchain\_pallet::crypto::Public, Runtime, UncheckedExtrinsic>;  impl offchain\_pallet::Trait for Runtime {  type Event = Event;  type Call = Call;  // 要在 runtime 使用的签名的交易  type SubmitSignedTransaction = SubmitTransaction;  // 要在 runtime 使用的无签名交易  type SubmitUnsignedTransaction = SubmitTransaction;  } |

随后为 runtime 实现 system::offchain::CreateTransaction 这个特性。继续在 lib.rs 文件中写入：

|  |
| --- |
| use sp\_runtime::transaction\_validity;  // --snip--  impl system::offchain::CreateTransaction<Runtime, UncheckedExtrinsic> for Runtime {  type Public = <Signature as Verify>::Signer;  type Signature = Signature;  fn create\_transaction<TSigner: system::offchain::Signer<Self::Public, Self::Signature>> (  call: Call,  public: Self::Public,  account: AccountId,  index: Index,  ) -> Option<(Call, <UncheckedExtrinsic as sp\_runtime::traits::Extrinsic>::SignaturePayload)> {  let period = 1 << 8;  let current\_block = System::block\_number().saturated\_into::<u64>();  let tip = 0;  let extra: SignedExtra = (  system::CheckVersion::<Runtime>::new(),  system::CheckGenesis::<Runtime>::new(),  system::CheckEra::<Runtime>::from(generic::Era::mortal(period, current\_block)),  system::CheckNonce::<Runtime>::from(index),  system::CheckWeight::<Runtime>::new(),  transaction\_payment::ChargeTransactionPayment::<Runtime>::from(tip),  );  let raw\_payload = SignedPayload::new(call, extra).ok()?;  let signature = TSigner::sign(public, &raw\_payload)?;  let address = Indices::unlookup(account);  let (call, extra, \_) = raw\_payload.deconstruct();  Some((call, (address, signature, extra)))  }  } |

如果链下工作机要使用无签名交易，需要在 contrast\_runtime! 宏下的 OffchainPallet 中添加额外参数 ValidateUnsigned，与其它 pallet 及其参数一起作为 runtime 的组成部分。另外，还需要为此单独编写自定义的 验证逻辑。

|  |
| --- |
| construct\_runtime!(  pub enum Runtime where  Block = Block,  NodeBlock = opaque::Block,  UncheckedExtrinsic = UncheckedExtrinsic  {  // --snip--  // 需要使用无签名交易  OffchainPallet: offchain\_pallet::{ Module, Call, Storage, Event<T>, transaction\_validity::ValidateUnsigned }  // 如果只需要使用签名交易，只需要：  // OffchainPallet: offchain\_pallet::{ Module, Call, Storage, Event<T> }  }  ); |

* **在 service.rs 文件中添加密钥**

我们用 KeyTypeId 指定了一个本地密钥库来存储应用子密钥，链下工作机可以通过访问它们来签发交易。可通过以下两种方式添加应用子密钥。

**选项 1（开发阶段）：添加第一个用户密钥作为应用子密钥**

在开发环境中，可将第一个用户的密钥添加为应用子密钥。在 node/src/service.rs 文件中作如下代码更新：

|  |
| --- |
| pub fn new\_full<C: Send + Default + 'static>(config: Configuration<C, GenesisConfig>)  -> Result<impl AbstractService, ServiceError>  {  // --snip--  // 克隆 Alice 的 key  let dev\_seed = config.dev\_key\_seed.clone();  // --snip--  let service = builder.with\_network\_protocol(|\_| Ok(NodeProtocol::new()))?  .with\_finality\_proof\_provider(|client, backend|  Ok(Arc::new(GrandpaFinalityProofProvider::new(backend, client)) as \_)  )?  .build()?;  // 添加以下部分，将密钥添加到密钥库  if let Some(seed) = dev\_seed {  service  .keystore()  .write()  .insert\_ephemeral\_from\_seed\_by\_type::<runtime::offchain\_pallet::crypto::Pair>(  &seed,  runtime::offchain\_pallet::KEY\_TYPE,  )  .expect("Dev Seed should always succeed.");  }  } |

完成后即可进行签名交易了。但此仅供 开发环境 使用。

**选项2：通过 CLI 添加应用子密钥**

在实际环境中，设置好 Substrate 节点后可通过命令行界面添加一个新的应用子密钥。操作如下所示：

|  |
| --- |
| # Generate a new account  $ subkey generate  # Submit a new key via RPC  $ curl http://localhost:9933 -H "Content-Type:application/json;charset=utf-8" -d \  '{  "jsonrpc":"2.0",  "id":1,  "method":"author\_insertKey",  "params": [  "<YourKeyTypeId>",  "<YourSeedPhrase>",  "<YourPublicKey>"  ]  }' |

如果命令和参数输入正确，则节点将返回如下 JSON 响应：

|  |
| --- |
| { "jsonrpc": "2.0", "result": null, "id": 1 } |

新密钥就成功添加到本地密钥库了。

* **签名交易**

现在我们已经做好用链下工作机签发签名交易的准备了。回到 my\_offchain\_worker.rs 这个 pallet 中：

|  |
| --- |
| decl\_module! {  pub struct Module<T: Trait> for enum Call where origin: T::Origin {  // --snip--  pub fn onchain\_callback(origin, \_block: T::BlockNumber, input: Vec<u8>) -> dispatch::Result {  let who = ensure\_signed(origin)?;  debug::info!("{:?}", core::str::from\_utf8(&input).unwrap());  Ok(())  }  fn offchain\_worker(block: T::BlockNumber) {  // 在这里，我们指定了下一个区块导入时要在链上调用的函数。  let call = Call::onchain\_callback(block, b"hello world!".to\_vec());  T::SubmitSignedTransaction::submit\_signed(call);  }  }  } |

定义了链上回调函数后，即可在链下工作机中，指定其在下一个区块导入阶段中执行。接下来我们就可以向节点提交签名交易了。

在Substrate代码库中查看 fn system::offchain::submit\_signed 的实现，我们能看到它是为本地密钥库中的每个密钥都调用了链上回调函数，但由于现在本地密钥库中只有一个密钥，因此只调用了一次。

* **无签名交易**

可通过以下代码将无签名交易发送回链上。

|  |
| --- |
| decl\_module! {  pub struct Module<T: Trait> for enum Call where origin: T::Origin {  // --snip--  pub fn onchain\_callback(\_origin, \_block: T::BlockNumber, input: Vec<u8>) -> dispatch::Result {  debug::info!("{:?}", core::str::from\_utf8(&input).unwrap());  Ok(())  }  fn offchain\_worker(block: T::BlockNumber) {  // 在这里，我们指定了下一个区块导入时要在链上调用的函数。  let call = Call::onchain\_callback(block, b"hello world!".to\_vec());  T::SubmitUnsignedTransaction::submit\_unsigned(call);  }  }  } |

默认所有无签名交易为无效，允许提交无签名交易需要在 my\_offchain\_worker.rs 中添加以下代码段：

|  |
| --- |
| decl\_module! {  // --snip--  }  impl<T: Trait> Module<T> {  // --snip--  }  #[allow(deprecated)]  impl<T: Trait> support::unsigned::ValidateUnsigned for Module<T> {  type Call = Call<T>;  fn validate\_unsigned(call: &Self::Call) -> TransactionValidity {  match call {  Call::onchain\_callback(block, input) => Ok(ValidTransaction {  priority: 0,  requires: vec![],  provides: vec![(block, input).encode()],  longevity: TransactionLongevity::max\_value(),  propagate: true,  }),  \_ => InvalidTransaction::Call.into()  }  }  } |

需要额外添加 deprecated 属性来禁用系统对无签名交易的警告提示，这是因为此部分 API 仍在变迁过程中，我们会在后期发布的 Substrate 版本对此更新。现在请暂时谨慎使用。

* **链上回调函数中的参数**

在链上回调时，回调函数名称及所有参数值会被一起哈希。回调函数将被存储，并在下一个区块导入期间被调用。如果发现哈希值已经存在，意味着一个具有相同参数集的函数曾经被调用过，那么签名交易将选择更高优先级的回调，而对于无签名交易，此回调函数将被忽略。

如果你的 pallet 经常进行链上回调，并且估计偶尔有重复参数集，则可从 offchain\_worker 函数中传入当前区块号作为额外参数。区块号只会一直增加，并能保证是唯一的。

* **获取外部数据**

从第三方 API 获取外部数据，请使用 my\_offchain\_worker.rs 中的 offchain::http 库，代码如下所示：

|  |
| --- |
| use sp\_runtime::{  offchain::http,  transaction\_validity::{  TransactionValidity, TransactionLongevity, ValidTransaction, InvalidTransaction  }  };  // --snip--  decl\_module! {  pub struct Module<T: Trait> for enum Call where origin: T::Origin {  // --snip--  fn offchain\_worker(block: T::BlockNumber) {  match Self::fetch\_data() {  Ok(res) => debug::info!("Result: {}", core::str::from\_utf8(&res).unwrap()),  Err(e) => debug::error!("Error fetch\_data: {}", e),  };  }  }  }  impl<T: Trait> Module<T> {  fn fetch\_data() -> Result<Vec<u8>, &'static str> {  // 指定请求  let pending = http::Request::get("https://min-api.cryptocompare.com/data/price?fsym=BTC&tsyms=USD")  .send()  .map\_err(|\_| "Error in sending http GET request")?;  // 等待返回  let response = pending.wait()  .map\_err(|\_| "Error in waiting http response back")?;  // 检查 HTTP 响应是否正常  if response.code != 200 {  debug::warn!("Unexpected status code: {}", response.code);  return Err("Non-200 status code returned from http request");  }  // 以 bytes 形式收取结果  Ok(response.body().collect::<Vec<u8>>())  }  } |

### 6.3.11 调试

所有软件开发，调试都是必不可少的，区块链也不例外。 通用的 Rust 调试工具，大部分同样也适用于 Substrate。 但由于 Substrate runtime 运行在 no\_std 环境中，调试有一些限制。

* **FRAME的调试 & 日志工具**

FRAME Supportdebug module 包含了可以从 runtime 代码中打印日志的宏和函数。

**从Native Runtime打印日志**

Native调试可以保证性能，使用 frame\_support::debug::native 模块中提供的宏。

|  |
| --- |
| pub fn do\_something(origin) -> DispatchResult {  print("Execute do\_something");  let who = ensure\_signed(origin)?;  let my\_val: u32 = 777;  Something::put(my\_val);  frame\_support::debug::native::debug!("called by {:?}", who);  Self::deposit\_event(RawEvent::SomethingStored(my\_val, who));  Ok(())  } |

frame\_support::debug::native::debug!宏避免了 Wasm runtime 的额外开销，但仅在 runtime 以 native 模式执行时才有效。 为了查看这些日志消息，必须给节点配置正确的日志 target ，默认与包含日志消息的 crate 名一致。

|  |
| --- |
| ./target/release/node-template --dev -lpallet\_template=debug |

debug! 宏接收了一个可选参数 target，于指定日志 target 名。

|  |
| --- |
| frame\_support::debug::native::debug!(target: "customTarget", "called by {:?}", who); |

**从Wasm Runtime打印日志**

Wasm runtime输出日志记录需要牺牲性能。

|  |
| --- |
| pub fn do\_something(origin) -> DispatchResult {  print("Execute do\_something");  let who = ensure\_signed(origin)?;  let my\_val: u32 = 777;  Something::put(my\_val);  frame\_support::debug::RuntimeLogger::init();  frame\_support::debug::debug!("called by {:?}", who);  Self::deposit\_event(RawEvent::SomethingStored(my\_val, who));  Ok(())  } |

* **Printable 特性**

Printable 特性提供了 runtime 在 no\_std 和 std 两种环境中进行打印的方式。 print 函数可在实现了Printable 特性的所有类型中使用。 Substrate 默认对部分类型 (u8, u32, u64, usize, &[u8], &str) 实现了此特性。 当然也可应用到自定义类型，以下示例用 node-template 作为代码模板，在一个 pallet 的 Error 类型上实现了 printable 特性。

|  |
| --- |
| use sp\_runtime::traits::Printable;  use sp\_runtime::print;  // pallet 的错误  decl\_error! {  pub enum Error for Module<T: Trait> {  /// 值为 None  NoneValue,  /// Value 达到最大值，不能再增加  StorageOverflow,  }  }  impl<T: Trait> Printable for Error<T> {  fn print(&self) {  match self {  Error::NoneValue => "Invalid Value".print(),  Error::StorageOverflow => "Value Exceeded and Overflowed".print(),  \_ => "Invalid Error Case".print(),  }  }  }  /// 不需要任何参数，尝试增加存储值，并可能抛出一个错误  pub fn cause\_error(origin) -> dispatch::DispatchResult {  // 检查它是否被签名，并获取签名者。也可以参考: ensure\_root 和 ensure\_none  let \_who = ensure\_signed(origin)?;  print("My Test Message");  match Something::get() {  None => {  print(Error::<T>::NoneValue);  Err(Error::<T>::NoneValue)?  }  Some(old) => {  let new = old.checked\_add(1).ok\_or(  {  print(Error::<T>::StorageOverflow);  Error::<T>::StorageOverflow  })?;  Something::put(new);  Ok(())  },  }  } |

使用RUST\_LOG环境变量来运行节点二进制文件以打印值。

|  |
| --- |
| RUST\_LOG=runtime=debug ./target/release/node-template --dev |

每次调用 runtime 函数时，值都会打印在终端或标准输出中。

|  |
| --- |
| 2020-01-01 tokio-blocking-driver DEBUG runtime My Test Message <-- str implements Printable by default  2020-01-01 tokio-blocking-driver DEBUG runtime Invalid Value <-- the custom string from NoneValue  2020-01-01 tokio-blocking-driver DEBUG runtime DispatchError  2020-01-01 tokio-blocking-driver DEBUG runtime 8  2020-01-01 tokio-blocking-driver DEBUG runtime 0 <-- index value from the Error enum definition  2020-01-01 tokio-blocking-driver DEBUG runtime NoneValue <-- str which holds the name of the ident of the error |

* **Substrate自带的print函数**

对于传统用例，Substrate 提供了额外工具来进行 Print 调试(或跟踪)。可使用 print函数 记录 runtime 执行的状态。

|  |
| --- |
| use sp\_runtime::print;  // --snip--  pub fn do\_something(origin) -> DispatchResult {  print("Execute do\_something");  let who = ensure\_signed(origin)?;  let my\_val: u32 = 777;  Something::put(my\_val);  print("After storing my\_val");  Self::deposit\_event(RawEvent::SomethingStored(my\_val, who));  Ok(())  }  // --snip-- |

可使用 RUST\_LOG 环境变量启动区块链，以查看打印日志。

|  |
| --- |
| RUST\_LOG=runtime=debug ./target/release/node-template --dev |

如果触发了错误，这些值将打印在终端或标准输出中。

|  |
| --- |
| 2020-01-01 00:00:00 tokio-blocking-driver DEBUG runtime Execute do\_something  2020-01-01 00:00:00 tokio-blocking-driver DEBUG runtime After storing my\_val |

* **If Std**

传统的 print 函数允许打印并实现 Printable 特性。 但是，在某些传统用例下，你可能不仅仅想要打印，或者不想仅因为调试而专门使用 Substrate 的一些特性。 这时候 if\_std! 宏 就十分有用。

使用此宏时要注意，仅当真正运行的是 runtime 的 native 版本时，宏内部代码才会执行。

|  |
| --- |
| use sp\_std::if\_std; // 将 if\_std! 宏导入到 scope |

println! 语句应放置在 if\_std 宏内。

|  |
| --- |
| decl\_module! {  // --snip--  pub fn do\_something(origin) -> DispatchResult {  let who = ensure\_signed(origin)?;  let my\_val: u32 = 777;  Something::put(my\_val);  if\_std! {  // 这段代码只有在启用 `std` 功能时才会被编译和执行。  println!("Hello native world!");  println!("My value is: {:#?}", my\_val);  println!("The caller account is: {:#?}", who);  }  Self::deposit\_event(RawEvent::SomethingStored(my\_val, who));  Ok(())  }  // --snip--  } |

每次调用runtime函数时，值都会打印在终端或标准输出中。

|  |
| --- |
| $ 2020-01-01 00:00:00 Substrate Node  2020-01-01 00:00:00 version x.y.z-x86\_64-linux-gnu  2020-01-01 00:00:00 by Anonymous, 2017, 2020  2020-01-01 00:00:00 Chain specification: Development  2020-01-01 00:00:00 Node name: my-node-007  2020-01-01 00:00:00 Roles: AUTHORITY  2020-01-01 00:00:00 Imported 999 (0x3d7a…ab6e)  # --snip--  -> Hello native world!  -> My value is: 777  -> The caller account is: d43593c715fdd31c61141abd04a99fd6822c8558854ccde39a5684e7a56da27d (5GrwvaEF...)  # --snip--  2020-01-01 00:00:00 Imported 1000 (0x3d7a…ab6e) |

### 6.3.12 测试

Runtime 测试通过模拟出 Substrate 的 runtime 环境，使开发者可以验证 runtime 模块的逻辑。

* **单元测试**

Substrate 采用了 Rust 现成单元测试 框架。运行测试的命令为：

|  |
| --- |
| cargo test <optional: test\_name> |

* **模拟Runtime环境**

在测试 Substrate runtime 模块之前，须先构建「模拟 runtime」环境。配置类型 Test 被定义为单元结构体，这个结构体实现了在模拟 runtime 中需要用到的所有配置特性。

|  |
| --- |
| #[derive(Clone, PartialEq, Eq, Debug)]  pub struct Test; |

如果 Test 实现了 balances::Trait，这一实现里面对于 Balance 类型可能使用了 u64 类型。

|  |
| --- |
| impl balances::Trait for TestRuntime {  type Balance = u64;  //..  } |

通过将 u64 赋值给 balances:: Balance 和 system:: AccountId，模拟 runtime 能大幅降低测试人员的工作量。 账户和余额的确认仅需要跟踪(AccountId: u64，Balance: u64)映射就可以了。

**模拟Runtime存储**

sp-io crate 暴露了 TestExternalities 的实现，经常被用来模拟测试中的存储。它是一个存储在内存的基于哈希映射的外部依赖，实现位于 substrate\_state\_machine] 称作 TestExternalities.

此示例演示了如何定义一个名为 ExtBuilder 的结构体，来构建 TestExternalities 的一个实例。

|  |
| --- |
| pub struct ExtBuilder;  impl ExtBuilder {  pub fn build() -> sp\_io::TestExternalities {  let mut storage = system::GenesisConfig::default().build\_storage::<TestRuntime>().unwrap();  sp\_io::TestExternalities::from(storage)  }  } |

为了给单元测试创建测试环境，我们将使用默认的创世配置调用 build 方法生成 TestExternalities。 然后，with\_externalities方法提供了一个 runtime 环境，我们可以在当中调用 pallet 的方法来测试存储、事件和错误行为是否符合预期。

|  |
| --- |
| #[test]  fn fake\_test\_example() {  ExtBuilder::build().execute\_with(|| {  // ...test conditions...  })  } |

外部依赖的自定义实现，使开发人员能够构建 runtime 环境，提供对外部节点功能的访问。 另外一个例子在 offchain 里可以找到, 它维护了自己的 Externalities 实现.

**创世配置**

前面展示的 ExtBuilder::build() 方法使用了默认的创世纪配置来构建模拟 runtime 环境。在大多数情况下，提前为测试设置好存储会比较方便些。

其中一个例子就是在测试前预先设置账户余额。

在对 system::Trait 的实现中，我们将 AccountId 设置为 u64，与前面提到的 Balance 一样。将(u64, u64)成对放入 balances 数组中，与 (AccountId, Balance) 一一配对，作为账户余额的预先配置。

|  |
| --- |
| pub fn build(self) -> sp\_io::TestExternalities {  GenesisConfig {  balances: Some(balances::GenesisConfig::<TestRuntime>{  balances: vec![  (1, 10),  (2, 20),  (3, 30),  (4, 40),  (5, 50),  (6, 60)  ],  vesting: vec![],  }),  }.build\_storage().unwrap().into()  } |

账户1有余额10，账户2有余额20，依此类推。

**区块生产**

对于验证依赖于区块时间线变化的预期行为，模拟区块生产非常有用。

一个简单的方法是以 System::block\_number() 为唯一输入，在所有模块的 on\_initialize 和 on\_finalize 方法之间递增 System 模块的区块号。虽然对于 runtime 代码来说，缓存对存储或系统模块的调用很重要，但测试环境工具应该优先考虑可读性，以方便将来维护。

|  |
| --- |
| fn run\_to\_block(n: u64) {  while System::block\_number() < n {  ExampleModule::on\_finalize(System::block\_number());  System::on\_finalize(System::block\_number());  System::set\_block\_number(System::block\_number() + 1);  System::on\_initialize(System::block\_number());  ExampleModule::on\_initialize(System::block\_number());  }  } |

仅当 ExampleModule 这个 pallet 的特性实现了 sr\_primitives::traits::{OnInitialize, OnFinalize} 特性 之后，才能够调用 on\_finalize 和 on\_initialize 两个方法，这两个方法分别负责在每个区块的块前和块后执行 runtime 方法的相应逻辑。

要在单元测试中使用此函数，

|  |
| --- |
| #[test]  fn my\_runtime\_test() {  with\_externalities(&mut new\_test\_ext(), || {  assert\_ok!(ExampleModule::start\_auction());  run\_to\_block(10);  assert\_ok!(ExampleModule::end\_auction());  });  } |

### 6.3.13 升级

无分叉升级是区块链开发框架 Substrate 的特性之一，通过将状态转换函数的定义（即 runtime 本身）作为元素，包括在区块链不断发展的 runtime 状态中，使此特性成为可能。这使网络维护者可以利用区块链的无信任、去中心化的共识能力，安全地改进 runtime 。

在 FRAME 系统的 runtime 开发中，System 库定义了 set\_code 调用，用于更新 runtime 的定义。链升级教程描述了 FRAME runtime 升级的细节，并演示了执行升级的两种机制，这两种升级都是严格意义上的「相加」，意味着它们通过「扩展」 runtime 的方式来「修改」 runtime，而非更新现有的 runtime 状态。如果 runtime 升级定义了对现有状态的改变，很可能需要执行 "存储迁移"。

* **Runtime版本管理**

为了让执行器能够选择合适的 runtime 执行环境，它需要知道本地和 Wasm runtime 的 spec\_name、spec\_version 和 authoring\_version。

Runtime 提供了runtime 版本结构体。下面是一个 runtime 版本结构体示例。

|  |
| --- |
| pub const VERSION: RuntimeVersion = RuntimeVersion {  spec\_name: create\_runtime\_str!("node-template"),  impl\_name: create\_runtime\_str!("node-template"),  authoring\_version: 1,  spec\_version: 1,  impl\_version: 1,  apis: RUNTIME\_API\_VERSIONS,  transaction\_version: 1,  }; |

* spec\_name: 不同 Substrate runtime 的标识符。
* impl\_name: 规范的实现者名。这对节点没有什么影响，只是为了区分不同实现团队的代码。
* authoring\_version: authorship 接口的版本，只有当授权接口的版本等于授权节点的本地 runtime ，授权节点才会尝试 author 块。
* spec\_version: runtime 规范的版本。一个完整的节点不会尝试使用它的本地 runtime 来替代链上的Wasm runtime ，除非 Wasm 和本地的所有 spec\_name、spec\_version 和 authoring\_version 都相同。
* impl\_version: 规范的执行版本。节点可以随意忽略，它只是表明代码不同而已，只要其他两个版本一致，那么即便代码可能不同，实现的功能却是一样的，非共识性的优化是唯一可以仅仅改变 impl\_version 的原因。
* transaction\_version: extrinsics 接口版本。在以下情况下它必须更新：extrinsic参数(数量、顺序或类型)改变、extrinsics 或 pallets 被移除、construct\_runtime! 宏中的 pallet 顺序「或」pallet 中的 extrinsic 顺序改变。一旦它被更新，spec\_version 也必须更新。
* apis 是一个支持的 runtime APIs 列表和其版本。

如上所述，执行器在选择执行之前，总是会验证原生 runtime 是否具有相同的共识驱动逻辑，与版本高低无关。

注意： runtime 版本是手动设置的。因此，如果 runtime 版本表述错误，执行者仍然会做出不恰当的决定。

**访问Runtime版本**

runtime 版本对于 FRAME runtime 上的应用和开发者大有用处，它通过 FRAME runtime 系统 的 state.getRuntimeVersion RPC 端点公开，该端点接受一个可选的区块标识符。大多数 FRAME 区块链开发者将使用 runtime 元数据，来了解 runtime 公开的 API 及其交互要求。runtime 元数据应该「仅」当链的 runtime spec\_version 改变时才会改变。

* **无分叉Runtime升级**

传统区块链在升级链的状态转换函数时，需要进行硬分叉。这需要停止节点并手动升级到最新的可执行文件。对于分布式生产网络来说，硬分叉升级协调是一个复杂的过程。

本页这些特性的最高目标，是允许 Substrate 区块链执行"无分叉 Runtime 升级"。这意味着，runtime 逻辑的升级可以实时进行，而不会导致网络分叉。

为了执行无分叉 runtime 升级，Substrate 用现有的 runtime 逻辑，将存储在链上的 Wasm runtime，更新为具有新逻辑的新共识冲突版。这个升级作为共识过程的一部分，被推送到网络上所有同步节点。一旦 Wasm runtime 被升级，执行者会看到本地 runtime spec\_name、spec\_version 或 authoring\_version 与新的 Wasm runtime 不再匹配，将回去执行规范的 Wasm runtime，任何执行过程中都不再使用本地 runtime。

* **存储迁移**

存储迁移是自定义的一次性函数，允许开发人员重新处理现有存储，以便将其转换为更新的目标。例如，设想一个 runtime 升级，将用户余额的数据类型从「无符号」整数改为「有符号」整数，在这种情况下，存储迁移将以无符号整数读取现有的值，并转换为有符号整数后写入。如果在需要迁移的时候不执行存储迁移，将导致 runtime 执行引擎误解代表 runtime 状态的存储值，从而导致未定义行为。Substrate runtime 存储迁移属于存储管理的一个类别，大致可称为"数据迁移"。

**使用FRAME进行存储迁移**

FRAME 存储迁移通过 OnRuntimeUpgrade 特性 实现，它指定了一个函数 on\_runtime\_upgrade。这个函数提供了一个钩子，允许 runtime 开发者指定逻辑，在 runtime 升级「后」、但任何外部甚至 on\_initialize 函数执行「前」立即运行。

**迁移准备**

为存储迁移做准备，就是了解 runtime 升级定义的更改。Substrate 存储库使用 D1-runtime-migration 标签来指定这种变化。

**编写迁移**

每个 runtime 迁移都不一样，但需遵循一些约定和最佳实践。

将迁移提取到可重用的函数中，并为它们编写测试。

在迁移中包含日志，以协助调试。

请记住，迁移是在升级后 runtime 的内容中执行的，这表示迁移代码可能需要包含废弃的类型，就像这个例子。

善用存储版本，使迁移更安全、更具声明性，就像这个例子。

**落实迁移**

默认情况下，FRAME 会根据 construct\_runtime! 宏中 pallet 出现的顺序来安排 on\_runtime\_upgrade 函数的执行顺序——具体来说，它们「逆向」(从上到下)运行。如有需要，FRAME 公开了一种以自定义顺序注入存储迁移的功能(见这里))。

FRAME 存储迁移将按照如下顺序运行。

1. frame\_system::on\_runtime\_upgrade
2. 自定义 on\_runtime\_upgrade, 如上所述
3. 所有 runtime 里 pallets 中定义的 on\_runtime\_upgrade 函数，按照上面描述的顺序。

**测试迁移**

测试存储迁移很重要，有很多程序可以提供协助。Substrate Debug Kit 包括一个 Remote Externalities 工具，它允许对实时链数据安全地进行存储迁移单元测试。Fork Off Substrate 脚本可以轻松创建链规范，用于引导本地测试链，以测试 runtime 升级和存储迁移。

### 6.3.14 预置模块FRAME介绍

Framework for Runtime Aggregation of Modularized Entities (FRAME) 是一组可简化 runtime 开发的模块（称为 pallet）和支持库。其中 pallet 指 FRAME 中的那些独立的功能模块，承载特定业务逻辑。

FRAME 提供了一些与 Substrate Primitives 交互的助手模块，而 Substrate Primitives 则提供了与核心客户端的交互接口。

下图展示了FRAME及其 support 库的总体架构：

