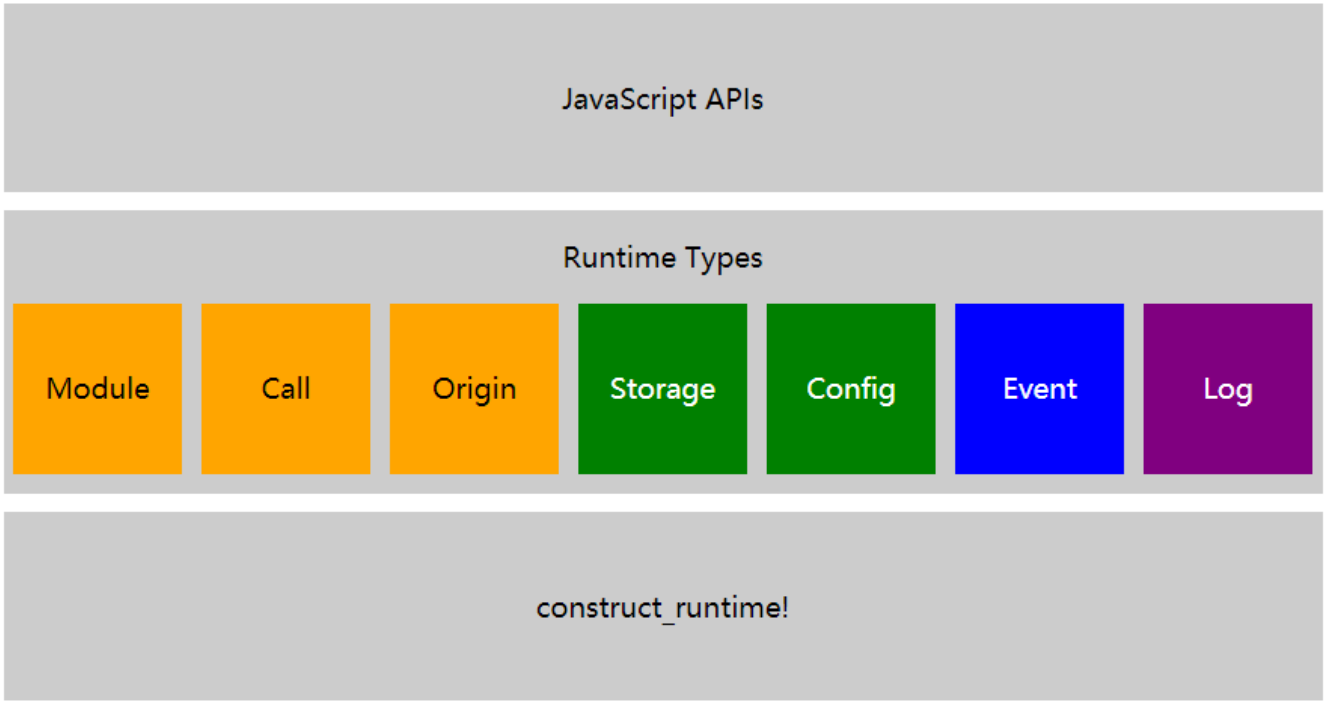
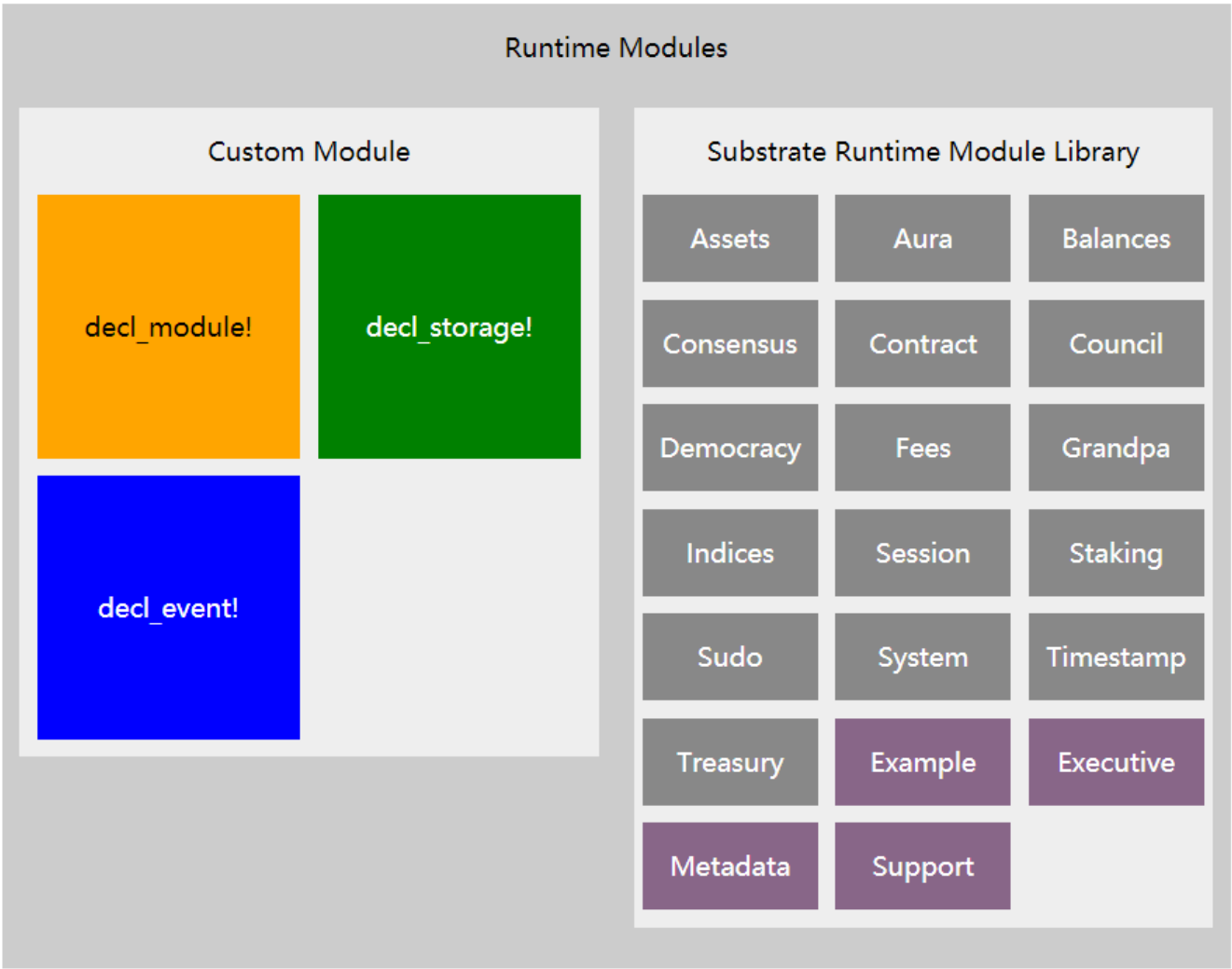
Substrate Runtime解析

Runtime架构





Substrate Runtime Module Library(SRML)

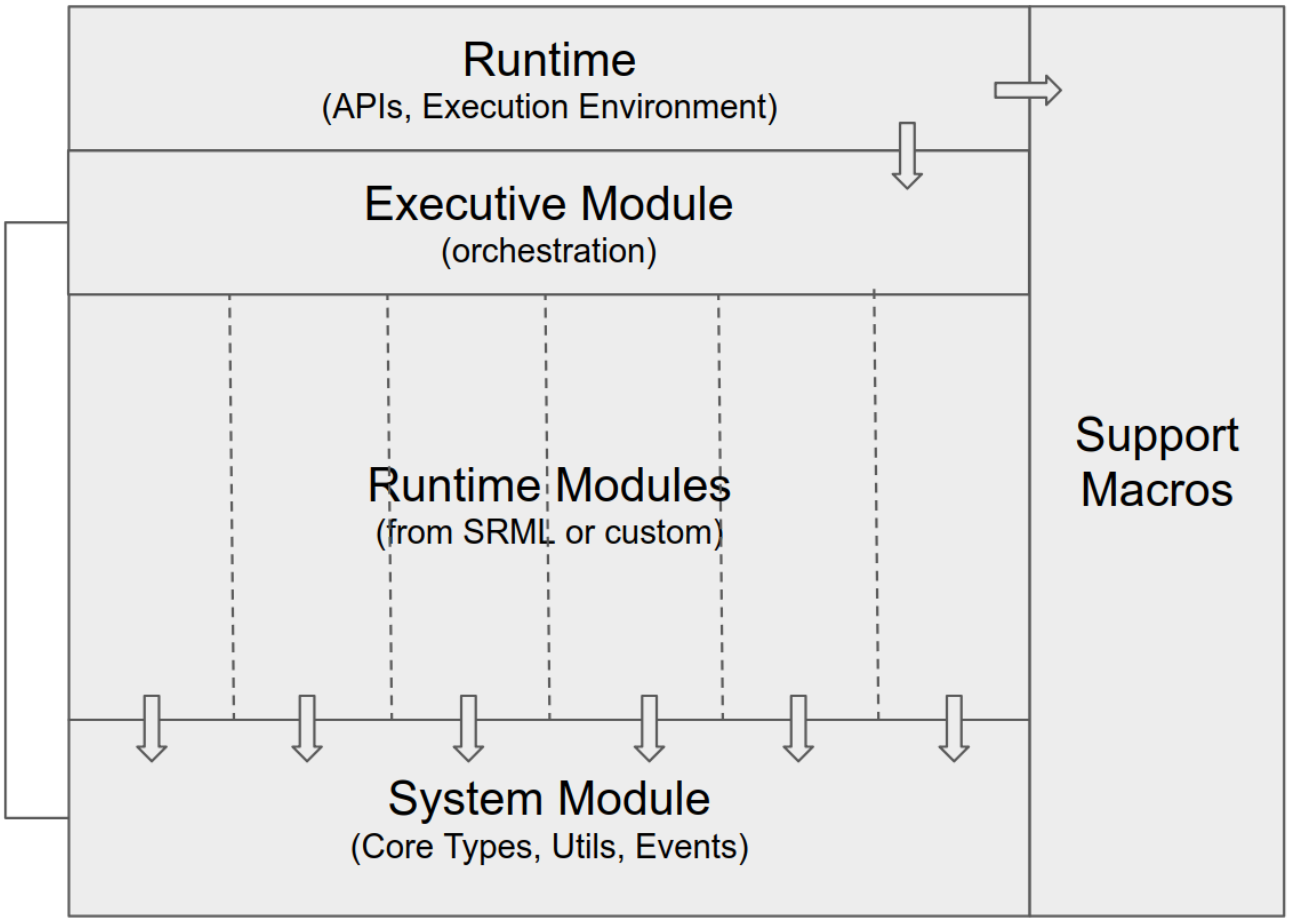
SRML是一组runtime模块。

What are runtime modules

Substrate Runtime由很多小组件组成来实现关注点的分离。这些组件都称为Runtime模块。Runtime模块将一组函数(可分派的外部调用，公有或私有的，可变或不可变的)，存储项以及可能的事件打包在一起。我们可以将SRML模块视为可用于创建其他包或运行时的包。(类似node.js的npm)

SRML Architecture

Substrate源码附带了一组Runtime模块以及一些框架组件来帮助构建执行环境。下图显示了SRML及其支持库的体系结构组件。



有四个主要框架组件支持运行时模块:

System Module

系统模块为其他模块提供了底层的API和工具，可以将它当做SRML的标准库。比较特殊的是，系统模块定义了Substrate运行时的所有核心类型。它还定义了extrinsic events(Success/Failure)。所有的其他模块都是基于系统模块的。

Executive Module

Executive模块充当运行时的业务流程层。它将传入的外部调用分派给runtime中的各个模块。

Support Macros

辅助宏是Rust宏的集合，帮助我们实现模块的最常见组件。这些宏在runtime扩展以生成类型(`Module`,`Call`,`Store`,`Event`等)，运行时使用这些类型与模块进行通信。一些常见的辅助宏是`decl_module`,`decl_storage`,`decl_event`,`ensure`等。

Runtime

Runtime汇集了所有框架组件和模块。它扩展了辅助宏来获取每个模块的类型和特征实现。它还调用Executive模块来分发调用请求。

SRML Modules

SRML中还附带了一组预先定义好的模块，这些模块可以作为独立的功能包在runtime重用。例如，SRML中的`Balances`模块可以用于跟踪账户和余额，`Assets`模块可用于创建和管理可替代资产等。

你可以通过派生系统模块来自定义模块，也可以选择一个或多个SRML模块派生以使用他们的功能。

以下是相关模块:

- Assets
- Aura
- Balances
- Consensus
- Contract
- Council
- Democracy
- Finality Tracker
- Grandpa
- Indices
- Session
- Staking
- Sudo
- Timestamp
- Treasury

Runtime Types

Call Enum

在Substrate中，Call Enum列出了runtime模块中可调用函数。

每个模块都有自己的Call Enum，其中包含该模块的函数名和参数。然后构造于runtime，生成外部Call Enum作为每个模块的特定Call的集合，它从每个模块的枚举中引用这些单独的Call类型。

Module Call Enum

在单个模块级别中，一个叫`Call`的枚举类由宏`decl_module!`产生。以下是一个例子:

```
decl_module! {
    // Simple declaration of the `Module` type. Lets the macro know what its working on.
    pub struct Module<T: Trait> for enum Call where origin: T::Origin {
        fn deposit_event<T>() = default;
        /// Authenticates the sudo key and dispatches a function call with `Root` origin
        /// The dispatch origin for this call must be _Signed_.
        fn sudo(origin, proposal: Box<T::Proposal>) {...}
        /// Authenticates the current sudo key and sets the given AccountId as the new
sudo key
        /// The dispatch origin for this call must be _Signed_.
        fn set_key(origin, new: <T::Lookup as StaticLookup>::Source) {...}
    }
}
```

当`decl_module!`调用之后，会生成下面这个`Call`。

```
pub enum Call<T: Trait> {
    sudo(Box<T::Proposal>),
    set_key(<T::Lookup as StaticLookup>::Source),
}
```

你可以发现它列举了你定义的模块中所有可调用的函数名和参数。不包含`origin`，因为是所有外部可调用函数必须的。

注意`deposit_event`并没有形成任何变量，因为它不是一个真正的可调用函数，`decl_module!`内部对它进行了处理。

Runtime *Outer* Call Enum

上述为Runtime中每个模块生成一个Call Enum，然后将此枚举传递给`construct_runtime!`用于生成外部Call Enum的宏，该枚举列出了所有Runtime模块并引用了它们各自的`Call`对象。

例如，在默认`substrate-node-template`中的runtime，我们可以看到:

```
construct_runtime!(
    pub enum Runtime with Log(InternalLog: DigestItem<Hash, AuthorityId,
AuthoritySignature>) where
        Block = Block,
        NodeBlock = opaque::Block,
        UncheckedExtrinsic = UncheckedExtrinsic
    {
        System: system::{default, Log(ChangesTrieRoot)},
        Timestamp: timestamp::{Module, Call, Storage, Config<T>, Inherent},
        Consensus: consensus::{Module, Call, Storage, Config<T>, Log(AuthoritiesChange),
Inherent},
        Aura: aura::{Module},
        Indices: indices,
        Balances: balances,
        Sudo: sudo,
        Fees: fees::{Module, Storage, Config<T>, Event<T>},
        TemplateModule: template::{Module, Call, Storage, Event<T>},
    }
);
```

以上代码当经过宏调用后会生成下面这个`Call`:

```
pub enum Call {
    Timestamp(::srml_support::dispatch::CallableCallFor<Timestamp>),
    Consensus(::srml_support::dispatch::CallableCallFor<Consensus>),
    Indices(::srml_support::dispatch::CallableCallFor<Indices>),
    Balances(::srml_support::dispatch::CallableCallFor<Balances>),
    Sudo(::srml_support::dispatch::CallableCallFor<Sudo>),
    TemplateModule(::srml_support::dispatch::CallableCallFor<TemplateModule>),
}
```

外部`Call`收集了`construct_runtime!`中的所有模块暴露Call枚举的宏，因为只有这样的模块才会公开可调度的函数。 因此，它定义了区块链中的完整暴露可调度函数集合。

Exposing Call via the Metadata Endpoint

最后，当你运行substrate节点时，它将自动生成一个getMetadata的入口点，其中包含你runtime模块生成的对象。

例如，当getMetadata API的响应转换为JSON时，SRML sudo模块会列出以下调用。

```
"modules": [
    {
        "name": "sudo",
        "prefix": "Sudo",
        "calls": [
            {
                "name": "sudo",
                "args": [
                    {
                        "name": "proposal",
                        "type": "Proposal"
                    }
                ],
                "docs": [
                    "Authenticates the sudo key and dispatches a function call with
`Root` origin",
                    "",
                    "The dispatch origin for this call must be _Signed_."
                ]
            },
            {
                "name": "set_key",
                "args": [
                    {
                        "name": "new",
                        "type": "Address"
                    }
                ],
                "docs": [
                    "Authenticates the current sudo key and sets the given AccountId as
the new sudo key",
                    "",
                    "The dispatch origin for this call must be _Signed_."
                ]
            }
        ],
        ...
    },
    ...
]
```

然后，这可以用于生成JavaScript函数，来进行runtime中函数的调用。

Event Enum

Substrate runtime提供了事件类型来为用户和客户端提供信息。

Declaring Events

在Substrate中，使用`decl_event!`宏来声明事件。例如在SRML_sudo模块中声明了以下事件。

```
decl_event!(
    pub enum Event<T> where AccountId = <T as system::Trait>::AccountId {
        /// A sudo just took place.
        Sudid(bool),
        /// The sudoer just switched identity; the old key is supplied.
        KeyChanged(AccountId),
    }
);
```

The *Module* Event Enum

在编译的时候，`decl_event!`宏会为每个模块扩展并生成`RawEvent`这个枚举类。然后使用宏中指定的traits将事件类型生成为`RawEvent`的具体实现。

以下是宏扩展后SRML sudo模块生成的RawEvent和Event类型。

```
pub enum RawEvent<AccountId> {
    Sudid(bool),
    KeyChanged(AccountId),
}

pub type Event<T> = RawEvent<<T as system::Trait>::AccountId>;
```

The *Outer* Event Enum

除了每个模块的`Event`类型之外，还有一个`construct_runtime!`为整个runtime模块生成外部事件的宏。此外部事件类型合并了所有的模块的事件枚举。

例如，在Substrate提供的默认`substrate-node-template`中，我们可以看到以下声明，其中显示了`construct_runtime!`中声明的所有模块，只有部分模块具有`Event`类型，包括`System`,`Indices`,`Balances`,`Sudo`和`TemplateModule`。

```
construct_runtime!(
    pub enum Runtime with Log(InternalLog: DigestItem<Hash, AuthorityId,
AuthoritySignature>) where
        Block = Block,
        NodeBlock = opaque::Block,
        UncheckedExtrinsic = UncheckedExtrinsic
    {
        System: system::{default, Log(ChangesTrieRoot)},
        Timestamp: timestamp::{Module, Call, Storage, Config<T>, Inherent},
        Consensus: consensus::{Module, Call, Storage, Config<T>, Log(AuthoritiesChange),
Inherent},
        Aura: aura::{Module},
        Indices: indices,
        Balances: balances,
        Sudo: sudo,
        // Used for the module template in `./template.rs`
```

```
        TemplateModule: template::{Module, Call, Storage, Event<T>},
    }
);
```

当宏调用结束之后，会生成以下内容:

```
pub enum Event {
    system(system::Event),
    indices(indices::Event<Runtime>),
    balances(balances::Event<Runtime>),
    sudo(sudo::Event<Runtime>),
    template(template::Event<Runtime>),
}
```

Events Metadata

事件类型的原始数据:

```
{
    "metadata": {
        "MetadataV1": {
            "modules": [
                {
                    "name": "sudo",
                    "prefix": "Sudo",
                    "events": [
                        {
                            "name": "Sudid",
                            "args": [
                                "bool"
                            ],
                            "docs": [
                                " A sudo just took place."
                            ]
                        },
                        {
                            "name": "KeyChanged",
                            "args": [
                                "AccountId"
                            ],
                            "docs": [
                                " The sudoer just switched identity; the old key is
supplied."
                            ]
                        }
                    ]
                }
            ]
        }
    }
}
```

GenesisConfig Struct

创世块的配置是用来初始化区块链的。

Declaring Configurable Storage Items

在Substrate中，可以在genesis配置中初始化变量值。要启用此功能，需要将`config()`参数添加到runtime的存储模块的字段中。

例如，在SRML_sudo中:

```
decl_storage! {
    trait Store for Module<T: Trait> as Sudo {
        Key get(key) config(): T::AccountId;
    }
}
```

The *Module* GenesisConfig Type

当`decl_storage!`宏扩展之后，它会生成`GenesisConfig`类型，其中包含对声明`config()`的每个变量的引用。

以下是上述代码生成的:

```
pub struct GenesisConfig<T: Trait> {
    pub key: T::AccountId,
}
```

支持创世块配置的每个模块也将其`GenesisConfig`类型别名为`<ModuleName>Config`作为`construct_runtime!`中的一部分。

```
pub type SudoConfig = sudo::GenesisConfig<Runtime>;
```

The *Outer* GenesisConfig Struct

使用genesis配置声明变量之后，需要设置它的初始值。我们通过外部`GenesisConfig`类型执行此操作，该类型由`construct_runtime!`生成。

```
pub struct GenesisConfig {
    pub system: Option<SystemConfig>,
    pub timestamp: Option<TimestampConfig>,
    pub consensus: Option<ConsensusConfig>,
    pub indices: Option<IndicesConfig>,
    pub balances: Option<BalancesConfig>,
    pub sudo: Option<SudoConfig>,
}
```

此外部类型包括对每个模块的`GenesisConfig`类型的引用。

Setting the Genesis Configuration Values

现在我们已经建立了模块和外部`GenesisConfig`类型，我们需要设置它们的值。

你可以在`chain_spec.rs`文件中看到:

```
fn testnet_genesis(root_key: AccountId, ...) -> GenesisConfig {
    GenesisConfig {
        sudo: Some(SudoConfig
        {
            key: root_key,
        }),
        ...
    }
}
```

当你启动节点之后，将使用你配置的`GenesisConfig`来进行初始化。

注意:`GenesisConfig`不适用于WASM运行时环境，仅适用于Native环境。可配置变量被视为由WASM环境预先初始化。

Exposing the GenesisConfig Type

`GenesisConfig`类型不是通过Substrate元数据端点直接公开的。 相反，`GenesisConfig`用于设置变量的初始值，因此你可以通过查询变量来看是否初始化成功。

Runtime Macros

construct_runtime!

Constructing a Runtime!

`construct_runtime!`宏是你用来定义所有的runtime模块的，包括内置的以及自定义的。

以下是一个例子:

```
construct_runtime!(
    pub enum Runtime with Log(InternalLog: DigestItem<Hash, Ed25519AuthorityId>) where
        Block = Block,
        NodeBlock = opaque::Block,
        UncheckedExtrinsic = UncheckedExtrinsic
    {
        System: system::{default, Log(ChangesTrieRoot)},
        Timestamp: timestamp::{Module, Call, Storage, Config<T>, Inherent},
        Consensus: consensus::{Module, Call, Storage, Config<T>, Log(AuthoritiesChange),
Inherent},
        Aura: aura::{Module},
        Indices: indices,
        Balances: balances,
        Sudo: sudo,
    }
);
```

Defining the Modules Used

在上面的示例中，我们仅使用SRML中的模块，但你可以看到有不同的语法方法可以定义模块及其类型的使用。 这些不同的模式由宏定义管理。

Naming Your Module

代码中的每一行就是一个模块定义:

```
System: system::{default, Log(ChangesTrieRoot)},
```

小写的system指向Rust模块，你可以在其中定义runtime的逻辑。大写的System是一个友好的名称，你可以自定义名称，该名称通过Runtime API公开。

Supported Types

`construct_runtime!` 宏支持以下类型:

- `Module`
- `Call`
- `Storage`
- `Event` or `Event<T>` (if the event is generic)
- `Origin` or `Origin<T>` (if the origin is generic)
- `Config` or `Config<T>` (if the config is generic)
- `Log`
- `Inherent`

No Types or `default`

你可以看到在 `Balances` 和 `Sudo` 模块中我们没有写任何类型调用，这种情况会采用默认的，默认为以下类型:

```
Module, Call, Storage, Event<T>, Config<T>
```

```
Balances: balances,
// 等同于
Balances: balances::{Module, Call, Storage, Event<T>, Config<T>},
```

如果你想要引入除了这些默认类型之外的，可以这么写:

```
System: system::{default, Log(ChangesTrieRoot)},
// 等同于
System: system::{Module, Call, Storage, Event<T>, Config<T>, Log(ChangesTrieRoot)},
```

When To Use Different Types

各种模块所公开的类型最终会为runtime提供调用。 大多数这些类型是由runtime开发中使用的其他宏自动生成的。

Module

Runtime的所有模块都需要 `Module` 类型。这种类型是通过 `decl_module!` 宏来生成的，这是定义模块公开的所有可调用函数。例如在 `Sudo` 模块中，它控制对你的链的 `admin` 的访问管理:

```
decl_module! {
    // Simple declaration of the `Module` type. Lets the macro know what its working on.
    pub struct Module<T: Trait> for enum Call where origin: T::Origin {
        fn deposit_event<T>() = default;

        fn sudo(origin, proposal: Box<T::Proposal>) {
            // This is a public call, so we ensure that the origin is some signed account.
            let sender = ensure_signed(origin)?;
            ensure!(sender == Self::key(), "only the current sudo key can sudo");

            let ok = proposal.dispatch(system::RawOrigin::Root.into()).is_ok();
            Self::deposit_event(RawEvent::Sudid(ok));
        }

        fn set_key(origin, new: <T::Lookup as StaticLookup>::Source) {
            // This is a public call, so we ensure that the origin is some signed account.
            let sender = ensure_signed(origin)?;
            ensure!(sender == Self::key(), "only the current sudo key can change the sudo
key");
            let new = T::Lookup::lookup(new)?;

            Self::deposit_event(RawEvent::KeyChanged(Self::key()));
            <Key<T>>::put(new);
        }
    }
}
```

Call

`Call` 类型也是由 `decl_module!` 宏来生成的，包含了一系列可调用函数和需要的参数。以下是 `Sudo` 模块的例子:

```
enum Call<T> {
    sudo(proposal: Box<T::Proposal>),
    set_key(new: <T::Lookup as StaticLookup>::Source),
}
```

Storage

`Storage` 模块是当你在你的模块中使用了 `decl_storage!` 宏就必须要定义，是用来将你自定义的数据写入链上的。

以下是 `Sudo` 模块的例子:

```
decl_storage! {
    trait Store for Module<T: Trait> as Sudo {
        Key get(key) config(): T::AccountId;
    }
}
```

Event

当你在模块中使用了 `decl_event!` 就需要定义这个类型，从而来找到你自定义的事件类型。

如果要公开模块定义的泛型类型，则需要使类型为通用（Event），如下所示:

```
/// An event in this module.
decl_event!(
    pub enum Event<T> where AccountId = <T as system::Trait>::AccountId {
        /// A sudo just took place.
        Sudid(bool),
        /// The sudoer just switched identity; the old key is supplied.
        KeyChanged(AccountId),
    }
);
```

否则，您可以简单地将Event用于非泛型类型。

Origin

只要模块声明要在模块中使用的自定义Origin枚举，就需要Origin类型。

对运行时的每个函数调用都有一个origin，它指定从哪里生成extrinsic。 在签名的外部（事务）的情况下，源包含调用者的标识符。 在Inherents的情况下，起源可以是空的。

你可以在`council motion`模块中看到自定义Origin的示例：

```
/// Origin for the council module.
#[derive(PartialEq, Eq, Clone)]
#[cfg_attr(feature = "std", derive(Debug))]
pub enum Origin {
    /// It has been condoned by a given number of council members.
    Members(u32),
}
```

Config

如果自定义的链上存储字段需要在创世块被初始化(添加`config()`)，则需要引入Config类型。

以下是一个例子:

```
GenesisConfig {
    ...
    sudo: Some(SudoConfig {
        key: root_key,
    }),
}
```

Log

如果你的模块需要生成日志那就需要Log类型。在模块中，你必须同时暴露一个`Log`类型和一个`RawLog enum`，`RawLog enum`定义了模块正在记录的内容。日志必须满足以下要求:

1. 所有支持的“系统”日志项的二进制表示应保持不变。 否则，本机代码将无法读取先前运行时版本生成的日志项。
2. Runtime永远不应删除对“系统”日志项的支持。 否则，本机代码将失去读取此类项目的能力，即使它们是由支持这些项目的版本生成的。

Consensus模块显示了如何实现日志的示例:

```
pub type Log<T> = RawLog<
    <T as Trait>::SessionKey,
>;

/// Add logs in this module.
#[cfg_attr(feature = "std", derive(Serialize, Debug))]
#[derive(Encode, Decode, PartialEq, Eq, Clone)]
pub enum RawLog<SessionKey> {
    /// Authorities set has been changed. Contains the new set of authorities.
    AuthoritiesChange(Vec<SessionKey>),
}

impl<SessionKey: Member> RawLog<SessionKey> {
    /// Try to cast the log entry as AuthoritiesChange log entry.
    pub fn as_authorities_change(&self) -> Option<&[SessionKey]> {
        match *self {
            RawLog::AuthoritiesChange(ref item) => Some(item),
        }
    }
}
```

使用`deposit_log()`函数来创建日志:

```
/// Deposit one of this module's logs.
fn deposit_log(log: Log<T>) {
    <system::Module<T>>::deposit_log(<T as Trait>::Log::from(log).into());
}
```

Inherent

如果模块提供了`ProvideInherent trait`的实现就需要Inherent类型。

如果你的模块想要自定义`inherent extrinsic`或者验证想要验证`inherent extrinsic`就需要这种自定义实现。如果你的模块需要额外的数据来创建`Inherent extrinsic`，则需要将数据作为`InherentData`传递到Runtime。

例如，在`Timestamp`模块中，`Inherent`数据用于创建`extrinsic`来设置给定区块的时间戳。作为创建区块节点，我们确认`Inherent extrinsic`中提出的时间在我们时钟的可接受时期内。

```
impl<T: Trait> ProvideInherent for Module<T> {
    type Inherent = T::Moment;
    type Call = Call<T>;

    fn create_inherent_extrinsics(data: Self::Inherent) -> Vec<(u32, Self::Call)> {
        let next_time = ::rstd::cmp::max(data, Self::now() + Self::block_period());
        vec![(T::TIMESTAMP_SET_POSITION, Call::set(next_time.into()))]
    }

    fn check_inherent<Block: BlockT, F: Fn(&Block::Extrinsic) -> Option<&Self::Call>>(
            block: &Block, data: Self::Inherent, extract_function: &F
    ) -> result::Result<(), CheckInherentError> {
        const MAX_TIMESTAMP_DRIFT: u64 = 60;
```

```
        let xt = block.extrinsics().get(T::TIMESTAMP_SET_POSITION as usize)
            .ok_or_else(|| CheckInherentError::Other("No valid timestamp inherent in
block".into()))?;

        let t = match (xt.is_signed(), extract_function(&xt)) {
            (Some(false), Some(Call::set(ref t))) => t.clone(),
            _ => return Err(CheckInherentError::Other("No valid timestamp inherent in
block".into())),
        }.into().as_();

        let minimum = (Self::now() + Self::block_period()).as_();
        if t > data.as_() + MAX_TIMESTAMP_DRIFT {
            Err(CheckInherentError::Other("Timestamp too far in future to accept".into()))
        } else if t < minimum {
            Err(CheckInherentError::ValidAtTimestamp(minimum))
        } else {
            Ok(())
        }
    }
}
```

decl_module!

Declaring a Module!

`decl_module!` 定义了你的可调用函数，作为访问你runtime的入口。这些函数应该协同工作，以构建一组通常独立的特性和功能，这些特性和功能将包含在区块链的最终runtime中。 宏的主要逻辑在这里定义。

SRML中每个不同的组件都是runtime模块的示例。

我们从最简单的`decl_module!`宏开始看:

```
decl_module! {
  pub struct Module<T: Trait> for enum Call where origin: T::Origin {
    fn set_value(origin, value: u32) -> Result {
      let _sender = ensure_signed(origin)?;
      <Value<T>>::put(value);
      Ok(())
    }
  }
}
```

注意这里写的`Value`默认已经在`decl_storage!`中声明，这里不再做相关介绍。

Declaration of the Module Type

`decl_module!`宏中第一行定义了`construct_runtime!`宏使用的`Module`类型。

```
pub struct Module<T: Trait> for enum Call where origin: T::Origin
```

这一行使用的是`decl_module!`宏中自定义的语法，不是标准的Rust。对于大多数模块开发，不用修改此行。

模块定义中用`T`来表示一个`Trait`类型。然后，模块内的函数可以使用此泛型来访问自定义类型。

枚举也定义为`Call`，这是`construct_runtime!`所要求的。将`decl_module!`中定义的函数分派到此枚举中。并明确定义函数名和参数。这种定义方式就是公开函数，允许下游API端与前端进行调用。

最后`origin: T::Origin`为简化`decl_module`中函数的参数定义而进行的优化。我们只是说函数中使用的原始变量的类型是`Trait :: Origin`，它通常由`system`模块定义。

Functional Requirements

为确保模块按预期运行，在开发模块功能时需要遵循这些规则。

Must Not Panic（无错误）

在任何情况下，模块都不应该发生`panic`。runtime模块中的`panic`可能会导致`DDOS`攻击。如果你的runtime模块有发生`panic`的可能，恶意用户可以发送执行大量计算工作的事务，导致runtime时出现错误，然后由于发生了错误，避免支付了与该计算工作相关的任何费用。在`panic`之前完成的计算都没有收费，因此`panic`将始终恢复对存储的任何更改，包括付款。

这种攻击只会直接影响接收`extrinsic`的节点。该节点将计算一个`extrinsic`直到`panic`。`panic`之后，它会抛弃`extrinsic`，但仍然能够产生一次阻挡。这里的一个例外是`on_initialise`或`on_finalise`中的一个`panic`，它实际上会阻塞你的节点，因为它将无法生成一个块，因为这些函数总是为每个生成的块调用。

你应该提前检查可能的错误情况并正确地处理它们。如果你的状态“受到不可挽回的损害”（即不一致），你仍然应该避免`panic`。当你发现这种不一致时，最好的办法就是简单地让状态单独并尽早检测它，以最大限度来减少计算，从而减少任何影响经济行为的`DoS`向量。

状态不一致通常可以解决治理戳状态重新形成的问题。在可能的情况下引入某种“重置”以进行治理调用也可能有助于解决这些情况。

No Side-Effects On Error(无任何副作用的错误)

此函数必须全部完成(并返回`Ok(())`)或者必须没有副作用的返回`Err()`。

作为Substrate的开发人员，你必须明确设计你runtime时的逻辑与在以太坊上开发智能合约的区别。

在以太坊，如果你的交易任何时候失败(错误或没有gas等)，你的智能合约的状态将不受影响。但是，在Substrate上并非如此。一旦交易开始修改区块链的存储，这些更改就是永久性的，即使交易在runtime期间稍后失败也是如此。

这对于区块链系统是必要的，因为你可能想要跟踪用户的`nonce`或者为发生的任何计算减去`gas`。这两件事实际上都发生在以太坊状态转换函数中(用于失败的交易)，但你作为智能合约开发者，从来不必担心这些事情。

你必须意识到你对区块链状态所做的任何更改，并确保它遵循“先验证，最后写”模式。

Function Return

`Dispatchable`函数并不会返回一个值。相对的，它只能返回一个结果，当一切成功完成时接受`Ok(())`或者如果出现错误则接受`Err(&'static str')`。

如果你没有指定`Result`的返回值，它将由`decl_module!`自动添加`Ok(())`在最后。

因此，此函数定义等效于上面的示例：

```
decl_module! {
  pub struct Module<T: Trait> for enum Call where origin: T::Origin {
    fn set_value(origin, value: u32) {
      let _sender = ensure_signed(origin)?;
      <Value<T>>::put(value);
    }
  }
}
```

你仍然可以像往常一样在代码中的其他位置返回`Err()`。

Proportional Costs to Computation

确保对每个调用只执行一次，内存与磁盘存储与调用者支付的费用或强制调用的难度成正比。

如果你没有办法确保模块功能是否会在没有大量计算的情况下成功，那么你就会遇到经典的区块链攻击。防范这种攻击的正常方法是为操作附加价值。作为存储的第一个主要更改，从调用者的账户中保留当前金额（`Balances`模块具有针对此方案的保留功能）如果事实证明你无法继续操作，则此金额应足以支付实际执行的任何费用。

如果最终发现操作是正常的，因此支票的费用应由网络承担，那么你可以退还保留的资产。但是，如果操作结果无效并且计算被浪费，那么你可以将其烧毁或在其他地方遣返。

Check Origin

所有函数都使用`origin`来确定调用者。模块支持三种`origin`类型:

- Signed Extrinsic - ensure_signed(origin)?
- Inherent Extrinsic - ensure_inherent(origin)?
- Root - ensure_root(origin)?

你总是需要选择其中一个方法作为你在函数中做的第一件事，否则你的链将是可攻击的。

Reserved Functions

虽然你可以在自己的模块中为函数命名，但有一些功能名称是保留的，并且你可以在函数中进行特殊调用。

deposit_event()

如果你在模块中想要发出事件，则需要定义`deposit_event()`函数，这个函数会处理你在`decl_events!`宏中定义的事件。事件可以包含泛型，在这种情况下，你应该`deposit_event<T>()`函数。

`decl_module!`宏为`deposit_event()`提供了默认的实现，你可以通过简单的定义来访问它。

```
fn deposit_event() = default;

// or for events with generics
// fn deposit_event<T>() = default;
```

如果你想在`decl_module!`中发送事件，需要这样写:

```
decl_module! {
  pub struct Module<T: Trait> for enum Call where origin: T::Origin {
    fn deposit_event<T>() = default;
```

```
    fn set_value(origin, value: u32) -> Result {
      let sender = ensure_signed(origin)?;
      <Value<T>>::put(value);

      Self::deposit_event(Event::Set(sender, value));

      Ok(())
    }
  }
}
```

on_initialise() and on_finalise()

这两个函数每个区块都要执行。

可以不带参数的调用这些函数，也可以带上区块号作为参数来调用这些函数。

```
// The signature could also be: `fn on_initialise(n: T::BlockNumber)`
fn on_initialise() {
    // Anything that needs to be done at the beginning of the block.
    Self::my_function();
}

// The signature could also be: `fn on_finalise()`
fn on_finalise(n: T::BlockNumber) {
    // Anything that needs to be done at the end of the block.
    Self::my_function_with_blocknumber(n);
}
```

你可以使用`on_initialise()`来帮助你执行任何需要在调用runtime业务逻辑之前需要执行的任务。例如，为更新的存储架构执行一次性的存储元素迁移。你可以使用`on_finalise()`来帮助你清理任何不需要的存储项或重置下一个块的值。

注意：如果使用这些函数打印到控制台，输出将显示两次，因为在准备块时将调用一次，在导入时再次调用一次。但是，在区块链中，这些函数只能被调用一次。

Privileged Functions

特权函数是只能够在`origin`是`Root`的时候被调用。可以在`Consensus`模块中找到特权函数的示例，以进行runtime的升级:

```
/// Set the new code.
pub fn set_code(new: Vec<u8>) {
    storage::unhashed::put_raw(well_known_keys::CODE, &new);
}
```

请注意，此函数在函数输入的开头省略了`origin`参数。 decl_module！ 宏自动转换没有`origin`的函数来检查`origin`是否为`Root`。 因此，上面的函数等同于写：

```
/// Set the new code.
pub fn set_code(origin, new: Vec<u8>) -> Result {
    ensure_root(origin)?;
    storage::unhashed::put_raw(well_known_keys::CODE, &new);
    Ok(())
}
```

`Result`以及`Ok(())`会自动被添加。

不同的runtime有允许执行特权调用的不同原因。

因为它是特权，我们可以假设它是一次性操作，可以使用大量的处理/存储/内存，而不必担心可玩性或攻击情形。

通常，这样的函数将通过`Sudo`模块中的`sudo()`函数调用，该函数可以根据来自用户的提议构建`Root`调用。

decl_storage!

声明存储中与编解码器兼容的类型的强类型包装器。

示例

```
decl_storage! {
        trait Store for Module<T: Trait> as Example {
                Foo get(foo) config(): u32=12;
                Bar: map u32 => u32;
                pub Zed build(|config| vec![(0, 0)]): linked_map u32 => u32;
        }
}
```

使用`trait Store for Module<T: Trait>`来声明进行链上存储，`as Example`为存储类的名称。名称必须是唯一的：具有相同名称和相同内部存储项名称的另一个模块将发生冲突。

基本存储包括名称和类型;支持的类型是:

- Value:`Foo: type`:实现了StorageValue
- Map:`Foo: map hasher($hash) type => type`:使用`$hash`实现StorageMap，标识`Hashable` trait可用散列算法选择。`hasher($hash)`是可选的，默认是`blake2_256`。小心在map中插入`trie`中`$hash(module_name ++ " " ++ storage_name ++ encoding(key))`的每个`key`。如果key不可信（例如，可以由用户设置），则必须使用诸如`blake2_256`之类的密码加密器。 否则，存储中的其他值可能会受到影响。
- Linked Map:`Foo: linked_map hasher($hash) type => type`:和`Map`一样，但是实现了EnumarableStorageMap.
- Double Map:`Foo: double_map hasher($hash) u32, $hash2(u32) => u32`:用`$hash`和`$hash2`来实现`StorageDoubleMap`，表示`Hashable` trait中可用的散列算法的选择。`hasher($hash)`是可选的，默认为`blake2_256`。小心插入`trie`中的双重映射中的每个键对。 最终`key`计算如下：

```
$hash(module_name ++ " " ++ storage_name ++ encoding(first_key)) ++
$hash2(encoding(second_key))
```

如果第一个`key`是不可信的，那么需要使用`blake2_256`的`hash`函数。否则，可能会损害所有存储项的其他值。第二个`key`也是一样。

基本存储可以扩展如下：

`#vis #name get(#getter) config(#field_name) build(#closure): #type = #default;`

- `#vis`：设置结构的可见性，如pub或者nothing
- `#name`：存储项的名称，用作存储中的前缀
- `get(#getter)`：在Module中实现`#getter`函数
- `config(#field_name)`：如果设置了`get`，则`field_name`是可选的。将包含在`GenesisConfig`中。
- `build`：闭包调用来覆盖存储
- `#default`：为`none`时的返回值

存储项可以通过以下几种方式被访问:

- 结构: `Foo::<T>`
- `Store Trait`结构: `<Module<T> as Store>::Foo`
- 使用`getter`进行调用: `Module::<T>::foo()`

GenesisConfig

可以定义用于存储初始化的可选GenesisConfig结构，或者至少一个存储字段需要默认初始化（get和config或build），或者具体如下：

```
decl_storage! {
        trait Store for Module<T: Trait> as Example {

                // Your storage items
        }
        add_extra_genesis {
                config(genesis_field): GenesisFieldType;
                config(genesis_field2): GenesisFieldType;
                ...
                build(|_: &mut StorageOverlay, _: &mut ChildrenStorageOverlay, _:
&GenesisConfig<T>| {
                        // Modification of storage
                })
        }
}
```

这种结构可以通过`decl_runtime!`宏来公开为`Config`

Module with Instances

`decl_storage!`宏支持使用以下语法构建具有实例的模块(`DefaultInstance`类型是可选的):

```
trait Store for Module<T: Trait<I>, I: Instance=DefaultInstance> as Example {}
```

然后使用两个通用参数(即`GenesisConfig<T,I>`)生成`genesis`配置，并使用两个通用参数来访问存储项。例如`<Dummy<T, I>>::get()`或者`Dummy::<T, I>::get()`。