Substrate Runtime Recipes

本文将介绍substrate的使用示例，这些示例演示了在开发自定义Runtime模块时可用的不同特性和功能。

准备工作

需要预先安装substrate

```
$ curl https://getsubstrate.io -sSf | bash
```

创建substrate应用案例

在本教程中，我们会使用Polkadot UI来与我们的节点进行交互，刚开始我们需要使用官方给的案例来修改，先下载官方案例:

```
$ substrate-node-new substrate-example <name>
```

如果你需要扩展`substrate-node-template`，你需要修改`substrate-example/runtime/src/lib.rs`。

在原始文件上加入这两行:

```
mod runtime_example;
impl runtime_example::Trait for Runtime {}
```

然后修改`construct_runtime!()`宏在最后引入`RuntimeExample`。

```
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,
        RuntimeExample: runtime_example::{Module, Call, Storage},
    }
);
```

最后，我们在与`lib.rs`相同的文件夹下创建一个名为`runtime_example.rs`的文件。

更新Runtime模块

你可以将以下任何runtime示例粘贴到该`runtime_examples.rs`文件中，并使用以下命令编译新的运行时二进制文件：

```
$ cd substrate-example
$ cargo build --release
```

你应该在开始新链之前删除旧链：

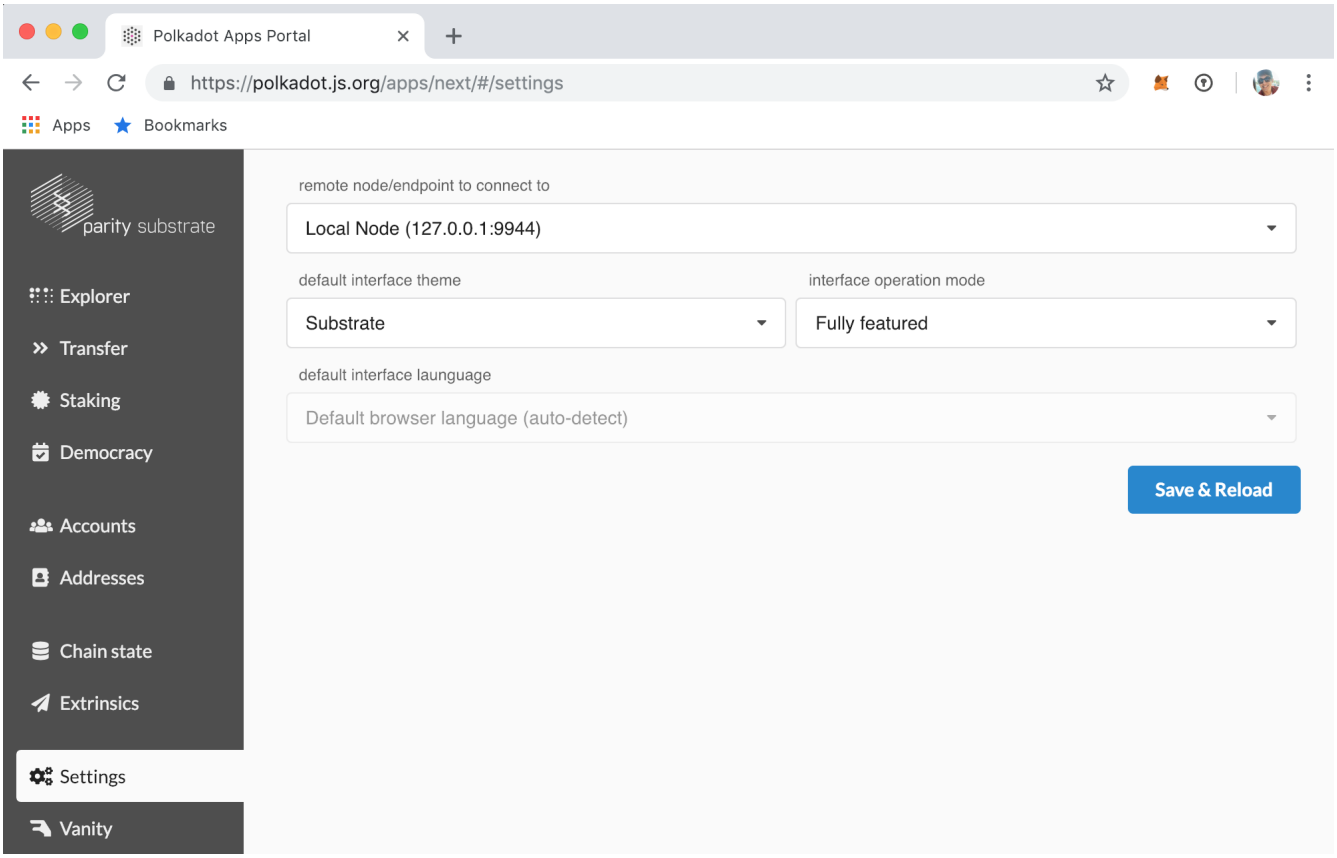
```
$ substrate purge-chain --dev
$ ./target/release/substrate-example --dev
```

使用`Polkadot UI`来与节点交互

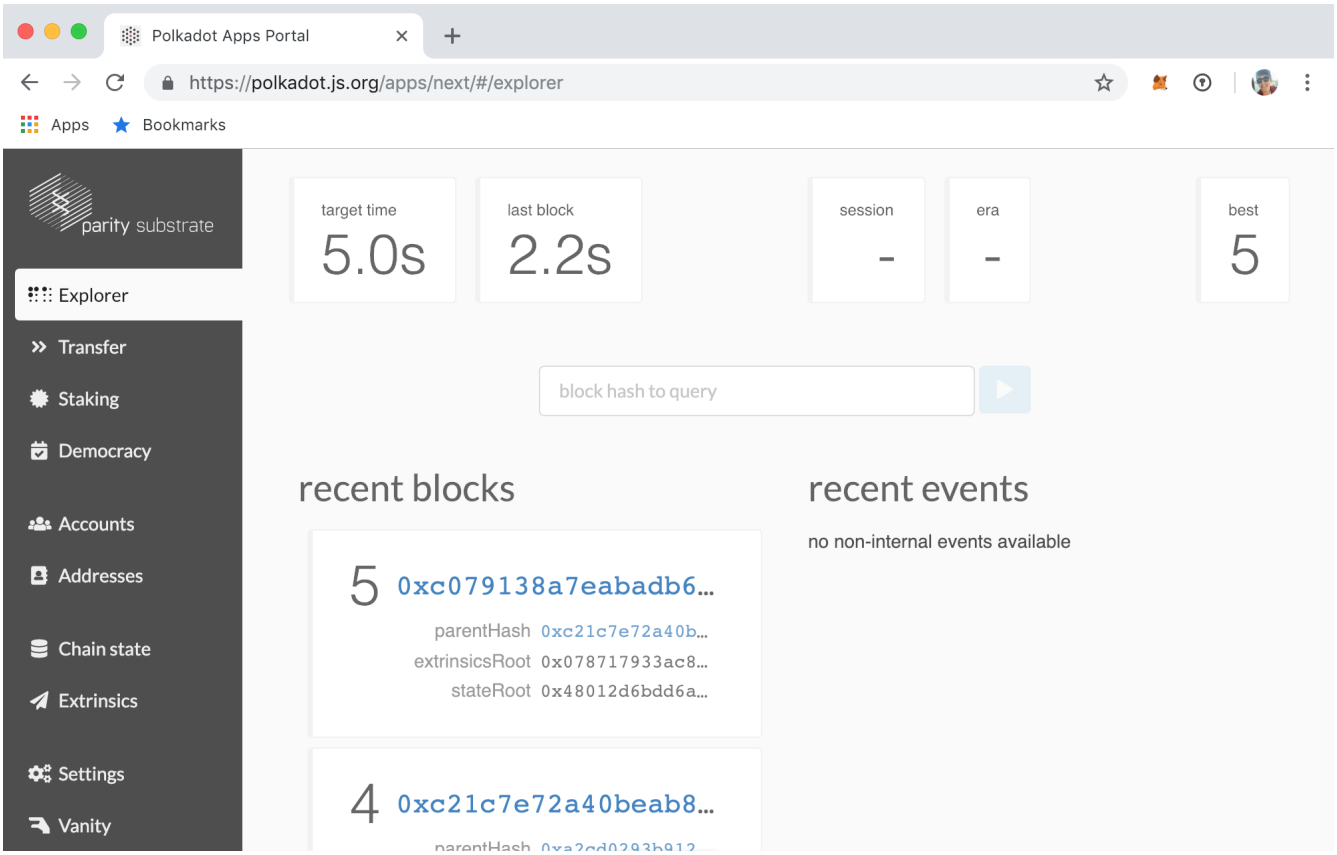
为了简化与自定义Substrate Runtime的交互，我们将利用Polkadot JS UI for Substrate.

默认情况下，此UI配置为与公共`Substrate test-network BBQ Birch`进行交互。要将其连接到本地节点，只需转到：

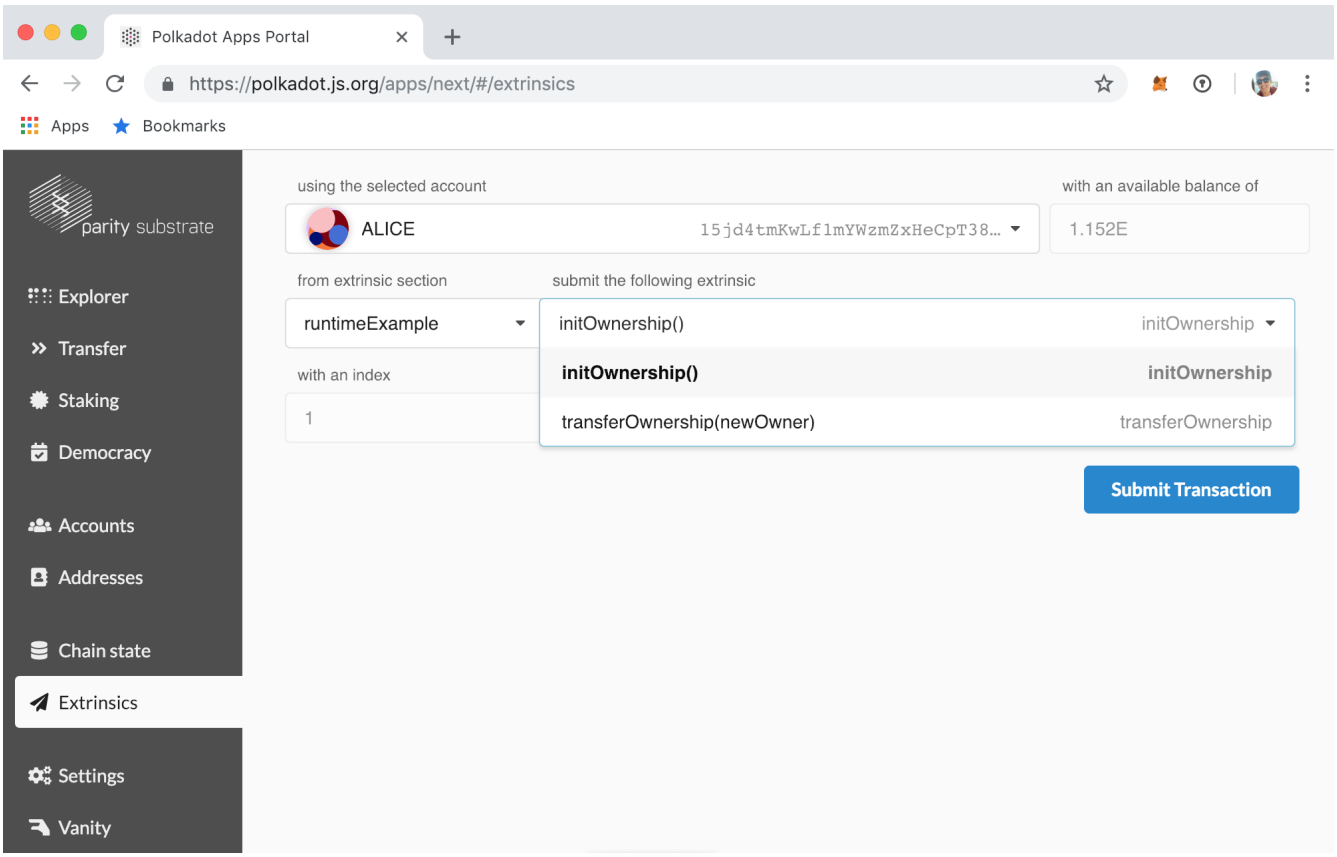
```
Settings > remote node/endpoint to connect to > Local Node (127.0.0.1:9944)
```



如果UI连接成功，你应该可以转到"资源管理器"选项卡并查看正在运行的块生产过程。



然后，你可以在runtimeExample下的Extrinsics选项卡中与自定义函数进行交互。

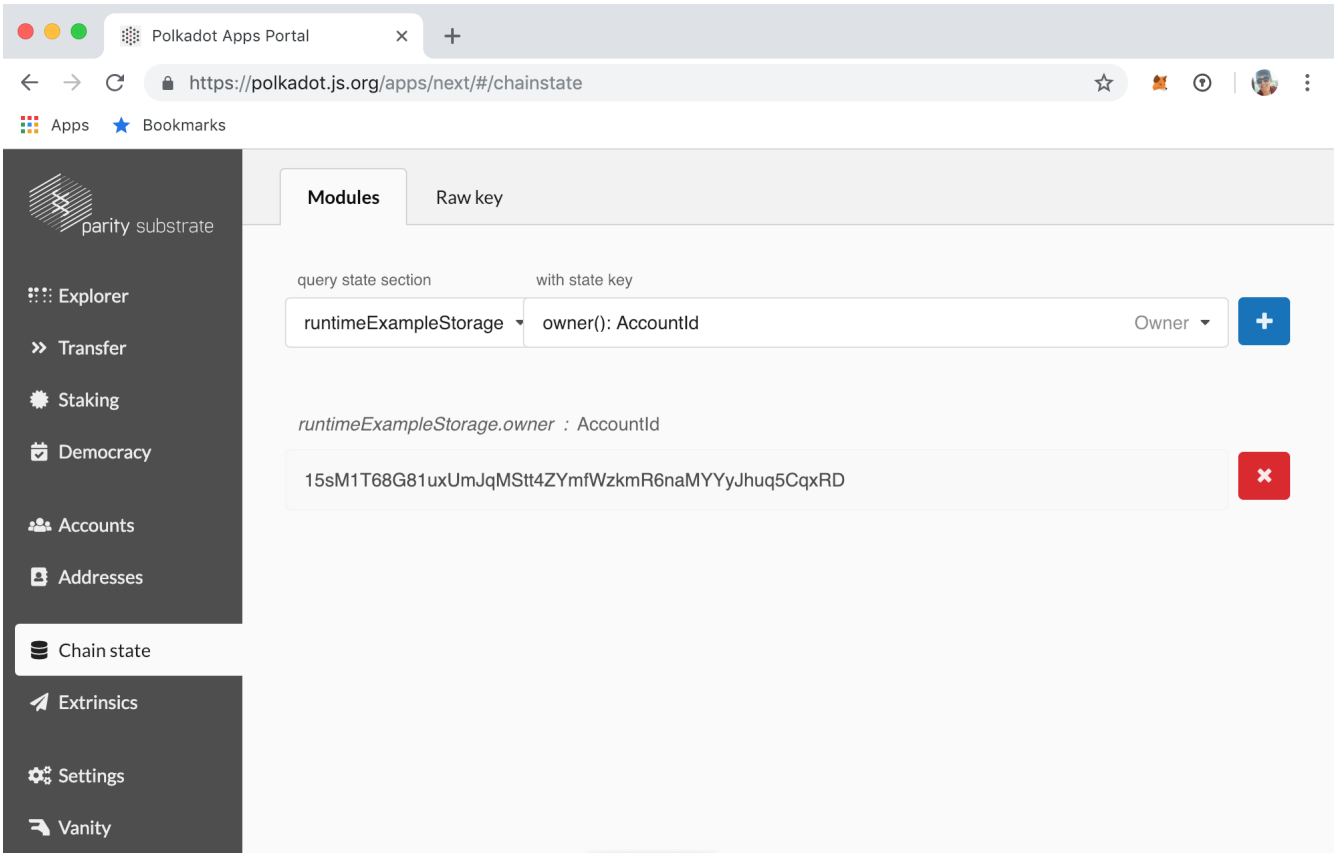


查看存储变量

如果你需要查看你自定义的变量。

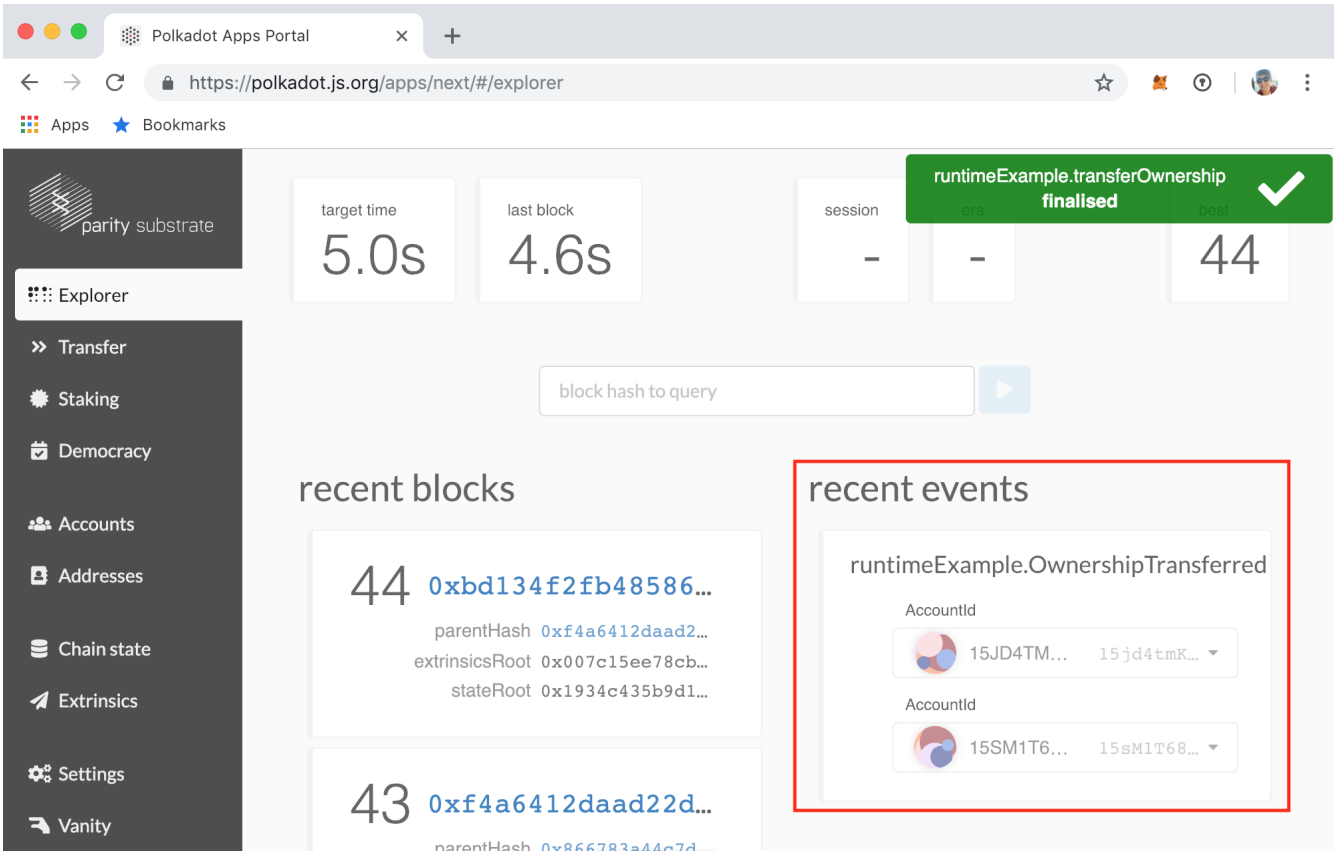
```
Chain State > runtimeExampleStorage > (variable name)
```

在这你可以查询变量的状态。如果尚未设置该值，则可能会返回`<unknown>`。



查看事件

有时候调用函数时会产生`Events`，你可以在`Explorer`选项卡下看到最近事件。



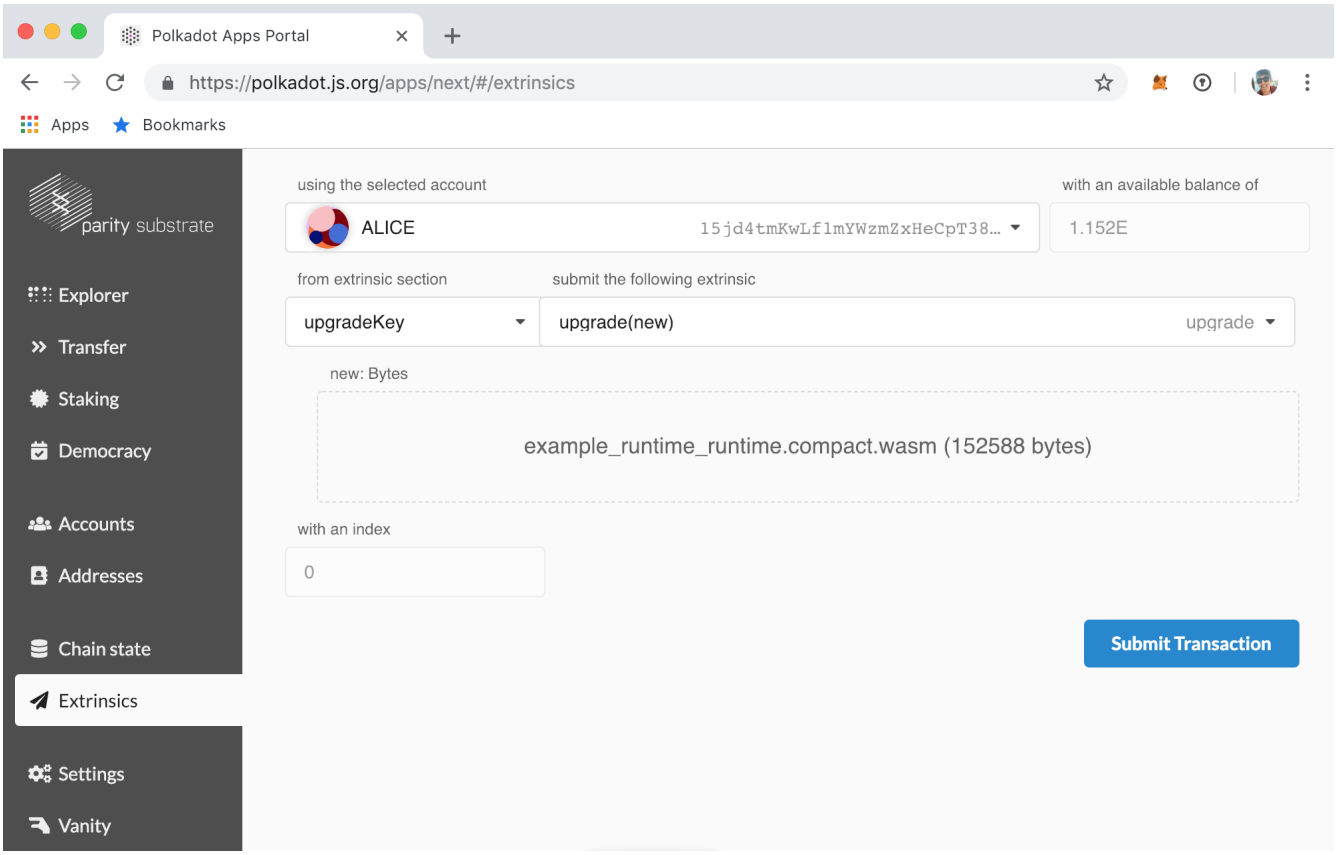
WASM Runtime更新

你可以使用Polkadot UI执行Runtime更新，而无需每次更新都要重启链。如果这样做，你不会在终端中显示Runtime信息，但你能够通过UI与链进行交互。要执行升级，要转到:

```
Extrinsics > Upgrade Key > upgrade(new)
```

在那里，你可以选择文件图片并上传运行`./build.sh`时生成的wasm文件。

```
substrate-example/runtime/wasm/target/wasm32-unknown-
unknown/release/node_runtime.compact.wasm
```



当更新成功之后，你可以刷新界面并看到更新的效果。

基本示例

可以在`srml/example`源码中和其他`SRML`模块中找到Substrate Runtime开发的一个很好的参考案例。以下示例旨在最小化特定功能的工作样本。

简单存储

创建一个简单的单值存储。

`runtime_example.rs`

```
use support::{decl_module, decl_storage, StorageValue, dispatch::Result};

pub trait Trait: system::Trait {}

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

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

```
    Value: u32;
  }
}
```

账户到值的映射

创建一个账户到值的map结构。

`runtime_example.rs`

```
use support::{decl_module, decl_storage, StorageMap, dispatch::Result};
use system::ensure_signed;

pub trait Trait: system::Trait {}

decl_module! {
        pub struct Module<T: Trait> for enum Call where origin: T::Origin {
                fn set_account_value(origin, value: u32) -> Result {
                        let sender = ensure_signed(origin)?;
                        <Value<T>>::insert(sender.clone(), value);
                        Ok(())
                }
        }
}

decl_storage! {
        trait Store for Module<T: Trait> as RuntimeExampleStorage {
                Value: map T::AccountId => u32;
        }
}
```

存储映射

创建一个KV映射结构。

`runtime_example.rs`

```
use support::{decl_module, decl_storage, StorageMap, dispatch::Result};

pub trait Trait: system::Trait {}

decl_module! {
        pub struct Module<T: Trait> for enum Call where origin: T::Origin {
                fn set_mapping(_origin, key: u32, value: u32) -> Result {
                        <Value<T>>::insert(key, value);
                        Ok(())
                }
        }
}

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

```
                Value: map u32 => u32;
        }
}
```

简易Token传输

如果我们想要通过Substrate实现简单的token传输，我们需要:

1. 设置总供应量
2. 在配置token时建立所有权
3. 使用我们的runtime函数协调token传输

```
decl_storage! {
  trait Store for Module<T: Trait> as Template {
    pub TotalSupply get(total_supply): u64 = 21000000; // (1)

    pub BalanceOf get(balance_of): map T::AccountId => u64; // (3)

    Init get(is_init): bool; // (2)
  }
}
```

同样，我们也需要设置事件来进行提醒。

```
decl_event!(
    pub enum Event<T> where AccountId = <T as system::Trait>::AccountId {
        // event for transfer of tokens
        // from, to, value
        Transfer(AccountId, AccountId, u64),
    }
);
```

为了将逻辑融合进我们的代码，我们需要这样写:

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

      // initialize the token
      // transfers the total_supply amount to the caller
      fn init(origin) -> Result {
        let sender = ensure_signed(origin)?;
        ensure!(Self::is_init() == false, "Already initialized.");

        <BalanceOf<T>>::insert(sender, Self::total_supply());

        <Init<T>>::put(true);

        Ok(())
      }
```

```
      // transfer tokens from one account to another
      fn transfer(origin, to: T::AccountId, value: u64) -> Result {
        let sender = ensure_signed(origin)?;
        let sender_balance = Self::balance_of(sender.clone());
        ensure!(sender_balance >= value, "Not enough balance.");

        let updated_from_balance = sender_balance.checked_sub(value).ok_or("overflow in
calculating balance")?;
        let receiver_balance = Self::balance_of(to.clone());
        let updated_to_balance = receiver_balance.checked_add(value).ok_or("overflow in
calculating balance")?;

        // reduce sender's balance
        <BalanceOf<T>>::insert(sender.clone(), updated_from_balance);

        // increase receiver's balance
        <BalanceOf<T>>::insert(to.clone(), updated_to_balance);

        Self::deposit_event(RawEvent::Transfer(sender, to, value));

        Ok(())
      }
  }
}
```

这个案例的完整代码可以在这找到:here.

在无限列表中添加/删除元素

如果列表的大小与我们访问数据的方式无关，那么实现很简单。

例如，假设我们有一个提议列表（在runtime定义为结构）。 当提案到期时，我们会将其从列表中删除，但不必更新已添加的其他提案的索引（如果我们在访问提案之前检查提案是否存在）。

我们可以将我们的提案存储在类似于初始示例的键值映射中：

```
#[cfg_attr(feature = "std", derive(Serialize, Deserialize, Debug))]
#[derive(Encode, Decode, Clone, PartialEq, Eq)]
struct Proposal<Hash> {
    details: Hash,
}

decl_storage! {
    trait Store for Module<T: Trait> as Example {
        Proposals get(proposals): map u32 => Proposal<T::Hash>;
        LargestIndex get(largest_index): u32;
    }
}
```

要添加提案，我们会增加largest_index并在该索引处添加提案：

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

        fn add_proposal(hash: Hash) -> Result {
            // any necessary checks here

            // instantiate new proposal
            let prop = Proposal { details: hash.clone() };

            // increment largest index
            <LargestIndex<T>>::mutate(|count| count + 1);

            // add a proposal at largest_index
            let largest_index = Self::largest_index::get();
            <Proposals<T>>::insert(largest_index, prop);

            Ok(())
        }
    }
}
```

要删除提议，我们可以在相关索引处简单地调用`StorageMap`类型的`remove`方法。 在这种情况下，我们不需要更新其他提案的索引。 这是因为此示例的顺序无关紧要。

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

        fn remove_proposal(index: u32) -> Result {
            // any necessary checks here

            // remove proposal at the given index
            <Proposals<T>>::remove(index);

            Ok(())
        }
    }
}
```

因为我们没有在地图中更新其他提案的索引，所以我们必须在删除提案，改变提议或执行任何其他相关操作之前检查提案是否存在。

```
// index is the `u32` that corresponds to the proposal in the `<Proposals<T>>` map
ensure!(<Proposals<T>>::exists(index), "proposal does not exist at the provided index");
```

有关此模式的更广泛和完整的示例，请看这:SunshineDAO.

有序列表的swap和pop

当我们想要保留存储以使我们的列表即使在删除元素之后也不会继续增长，我们可以调用swap和pop方法:

1. 使用列表头部的元素（map中具有最高索引的元素）交换要删除的元素。
2. 删除最近放置在最高索引处的元素。
3. 减少LargestIndex值。

继续我们的示例，我们保持添加提议的相同逻辑（增加LargestIndex并在列表的头部插入条目）。 但是，我们从列表中删除元素时调用swap和pop函数:

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

        fn remove_proposal(index: u32) -> Result {
            // check that a proposal exists at the given index
            ensure!(<Proposals<T>>::exists(index), "A proposal does not exist at this
index");

            let largest_index = Self::largest_index::get();
            let proposal_to_remove = <Proposals<T>>::take(index);
            // swap
            if index != largest_index {
                let temp = <Proposals<T>>::take(largest_index);
                <Proposals<T>>::insert(index, temp);
                <Proposals<T>>::insert(largest_index, proposal_to_remove);
            }
            // pop
            <Proposals<T>>::remove(largest_index);
            <LargestIndex<T>>::mutate(|count| count - 1);

            Ok(())
        }
    }
}
```

Linked Map

要为更简单的代码交换性能，请使用`linked_map`数据结构。 除了`StorageMap`之外，通过实现`EnumarableStorageMap`，`linked_map`提供了一个方法头，它可以生成列表的头部，从而不必存储`LargestIndex`。 枚举方法还返回根据何时（键，值）对插入到映射中而排序的迭代器。

要使用`linked_map`，我们还需要导入`EnumerableStorageMap`。 这是`decl_storage`块中的新声明:

```
use support::{StorageMap, EnumerableStorageMap}; // no StorageValue necessary

decl_storage! {
    trait Store for Module<T: Trait> as Example {
        Proposals get(proposals): linked_map u32 => Proposal<T::Hash>;
        // no largest_index value necessary
    }
}
```

这是新的`remove_proposal`方法:

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

        fn remove_proposal(index: u32) -> Result {
            // check that a proposal exists at the given index
            ensure!(<Proposals<T>>::exists(index), "A proposal does not exist at this
index");

            let head_index = Self::proposals::head();
            let proposal_to_remove = <Proposals<T>>::take(index);
            <Proposals<T>>::insert(index, <Proposals<T>>::get(head_index));
            <Proposals<T>>::remove(head_index);

            Ok(())
        }
    }
}
```

唯一需要注意的是，这种实现会产生一些性能成本（仅与使用 `StorageMap` 和 `StorageValue` 相比），因为 `linked_map heap` 将整个映射分配为迭代器以实现 `enumerate` 方法.

Higher Order Arrays with Tuples and Maps

为了表示跨多个用户的多个项目的所有权，可以将元组与映射一起使用以模拟数组。

例如，考虑一种情况，其中持久存储跟踪社交网络图，其中每个用户（由AccountId表示）具有其他朋友的列表。 在这种情况下，使用二维数组会很方便：

```
SocialNetwork[AccountId][Index] -> AccountId
```

通过这种数据结构，我们可以通过调用来查看给定AccountId有多少朋友：

```
SocialNetwork[AccountId].length()
```

要在Substrate运行时存储的上下文中模拟此数据结构，我们可以使用元组和映射（在 `decl_storage! {}` 块中声明，如前面的示例所示）：

```
SocialNetwork get(my_friend): map (T::AccountId, u32) => T::AccountId;
SocialNetwork get(friends_count): map T::AccountId => u32;
```

在Substrate上管理运行时存储时，使用映射来模拟高阶数据结构的模式很常见。 要查看此模式，请查看the Substratekitties Collectables Tutorial.

String Storage (as Bytemap)

如何使用JavaScript将字符串存储在运行时中以将字符串转换为十六进制并且转换回来。

Substrate不直接支持字符串。 运行时存储用于存储运行时运行的业务逻辑的状态。 它不是存储UI所需的一般数据。 如果你真的需要将一些任意数据存储到你的运行时，你总是可以创建一个`bytearray (Vec <u8>)`，但更合乎逻辑的做法是将一个哈希存储到像`IPFS`这样的服务，然后用来为你的用户界面获取数据。

`runtime_example.rs`

```
use support::{decl_module, decl_storage, ensure, StorageValue, dispatch::Result};
use rstd::prelude::*;

pub trait Trait: system::Trait {}

const BYTEARRAY_LIMIT: usize = 500;

decl_module! {
  pub struct Module<T: Trait> for enum Call where origin: T::Origin {
    fn set_value(_origin, value: Vec<u8>) -> Result {
      ensure!(value.len() <= BYTEARRAY_LIMIT, "Bytearray is too large");
      <Value<T>>::put(value);
      Ok(())
    }
  }
}

decl_storage! {
  trait Store for Module<T: Trait> as RuntimeExampleStorage {
    Value: Vec<u8>;
  }
}
```

用于字符串存储的JavaScript辅助程序

我们将字符串存储为`bytearray`，它作为十六进制字符串输入到`Polkadot UI`中。 这些辅助函数允许你在浏览器控制台中将字符串转换为十六进制以及转换回来。

```
function toHex(s) {
    var s = unescape(encodeURIComponent(s))
    var h = '0x'
    for (var i = 0; i < s.length; i++) {
        h += s.charCodeAt(i).toString(16)
    }
    return h
}

function fromHex(h) {
    var s = ''
    for (var i = 0; i < h.length; i+=2) {
        s += String.fromCharCode(parseInt(h.substr(i, 2), 16))
    }
    return decodeURIComponent(escape(s))
}
```

添加事件

一个简单的添加机器，用于检查溢出并使用结果发出事件，而不使用存储。

你需要修改`lib.rs`。 添加`type Event = Event;`到`trait`实现，删除存储，并将事件添加到`construct_runtime!()`，如下所示：

`lib.rs`

```
impl runtime_example::Trait for Runtime {
        type Event = Event;
}

...
RuntimeExample: runtime_example::{Module, Call, Event},
...
```

`runtime_example.rs`

```
use support::{decl_module, decl_storage, decl_event, dispatch::Result};

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

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

        fn add(_origin, val1: u32, val2: u32) -> Result {
            let result = match val1.checked_add(val2) {
                Some(r) => r,
                None => return Err("Addition overflowed"),
            };
            Self::deposit_event(Event::Added(val1, val2, result));
            Ok(())
        }
    }
}

decl_event!(
    pub enum Event {
        Added(u32, u32, u32),
    }
);
```

自带的权限函数(`OnlyOwner`,`Ownable`)

权限函数的基本实现，只能由“owner”调用。 成功运行该函数时会发出一个事件。

你需要修改此示例的`lib.rs`. 添加`type Event = Event;`到trait实现，并将`Event <T>`添加到`construct_runtime () `宏：

`lib.rs`

```
impl runtime_example::Trait for Runtime {
        type Event = Event;
}

...
RuntimeExample: runtime_example::{Module, Call, Storage, Event<T>},
...
```

`runtime_example.rs`

```
use support::{decl_module, decl_storage, decl_event, StorageValue, dispatch::Result,
ensure};
use system::ensure_signed;

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

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

        fn init_ownership(origin) -> Result {
            ensure!(!<Owner<T>>::exists(), "Owner already exists");
            let sender = ensure_signed(origin)?;
            <Owner<T>>::put(&sender);
            Self::deposit_event(RawEvent::OwnershipTransferred(sender.clone(), sender));
            Ok(())
        }

        fn transfer_ownership(origin, newOwner: T::AccountId) -> Result {
            let sender = ensure_signed(origin)?;
            ensure!(sender == Self::owner(), "This function can only be called by the
owner");
            <Owner<T>>::put(&newOwner);
            Self::deposit_event(RawEvent::OwnershipTransferred(sender, newOwner));
            Ok(())
        }
    }
}

decl_storage! {
        trait Store for Module<T: Trait> as RuntimeExampleStorage {
                Owner get(owner): T::AccountId;
    }
}

decl_event!(
        pub enum Event<T> where AccountId = <T as system::Trait>::AccountId {
                OwnershipTransferred(AccountId, AccountId),
```

```
        }
);
```

哈希函数

Substrate使用`BlakeTwo256`算法为散列数据提供内置支持。 这是系统特性的一部分。 系统特征下的Hashing类型公开了一个名为hash的函数。 此函数接受字节数组（`Vec <u8>`）的引用，并生成`BlakeTwo256`哈希摘要。

在下面的代码片段中，我们的自定义模块有一个函数`get_hash`，它接受`Vec <u8>`参数数据并在其上调用哈希函数。

```
use runtime_primitives::traits::Hash;
use support::{decl_module, dispatch::Result};
use {system::{self}};

pub trait Trait: system::Trait {}

decl_module! {
  pub struct Module<T: Trait> for enum Call where origin: T::Origin {
    pub fn get_hash(_origin, data: Vec<u8>) -> Result {
      let _digest = <<T as system::Trait>::Hashing as Hash>::hash(&data);
      Ok(())
    }
  }
}
```

使用泛型存储结构

一个基本运行时，它通过泛型使用Rust原语类型和Substrate特定类型的组合来存储自定义嵌套结构。 我们还将向你展示如何在`Polkadot-UI`和`Substrate-UI`中导入和使用此自定义类型。

```
use support::{decl_module, decl_storage, StorageMap, dispatch::Result};

pub trait Trait: balances::Trait {}

#[derive(Encode, Decode, Default)]
pub struct Thing <Hash, Balance> {
    my_num: u32,
    my_hash: Hash,
    my_balance: Balance,
}

#[derive(Encode, Decode, Default)]
pub struct SuperThing <Hash, Balance> {
    my_super_num: u32,
    my_thing: Thing<Hash, Balance>,
}

decl_module! {
    pub struct Module<T: Trait> for enum Call where origin: T::Origin {
        fn set_mapping(_origin, key: u32, num: u32, hash: T::Hash, balance: T::Balance) ->
```

```
Result {
            let thing = Thing {
                            my_num: num,
                            my_hash: hash,
                            my_balance: balance
                        };
            <Value<T>>::insert(key, thing);
            Ok(())
        }

        fn set_super_mapping(_origin, key: u32, super_num: u32, thing_key: u32) -> Result
{
            let thing = Self::value(thing_key);
            let super_thing = SuperThing {
                            my_super_num: super_num,
                            my_thing: thing
                        };
            <SuperValue<T>>::insert(key, super_thing);
            Ok(())
        }
    }
}

decl_storage! {
    trait Store for Module<T: Trait> as RuntimeExampleStorage {
        Value get(value): map u32 => Thing<T::Hash, T::Balance>;
        SuperValue get(super_value): map u32 => SuperThing<T::Hash, T::Balance>;
    }
}
```

要通过UI访问此结构的值，你需要导入新类型的结构，以便UI了解如何解码它。

`Polkadot UI`

对于`Polkadot-UI`，您需要创建一个描述结构的JSON文件：

```
{
    "Thing": {
        "my_num": "u32",
        "my_hash": "Hash",
        "my_balance": "Balance"
    },
    "SuperThing": {
        "my_super_num": "u32",
        "my_thing": "Thing"
    }
}
```

转到:

```
Settings > developer > additional type definitions (JSON)
```

然后倒入json文件。

Substrate UI

对于`Substrate UI`，您需要利用`oo7-substrate`模块提供的注册功能：`addCodecTransform()`。

在控制台中，只需输入：

```
addCodecTransform('Thing<Hash,Balance>', { my_num: 'u32', my_hash: 'Hash', my_balance:
'Balance' });
addCodecTransform('SuperThing<Hash,Balance>', { my_super_num: 'u32', my_thing: 'Thing' });
```

你可以在`app.jsx`的`constructor()`函数中添加这些行，以在页面加载时自动导入这些类型。

Working with Custom Enums in Polkadot Apps

如果你想在Polkadot应用程序中注册自定义Substrate枚举，这是一个工作示例。

首先，你需要在Substrate模块（Rust）中定义枚举：

```
// Example 1: Simple enum
pub enum VoteKind {
    Approve,
    Reject,
    Slash,
}

// Example 2: Parametrized enum
pub enum ElectionStage<BlockNumber> {
    Announcing(BlockNumber),
    Voting(BlockNumber),
    Revealing(BlockNumber),
}
```

其次，在`Polkadot Apps`的`TypeScript`代码中注册以前的枚举。 此代码放在https://github.com/polkadot-js/apps/blob/master/packages/apps/src/index.tsx#L47附近。

```
import { BlockNumber } from '@polkadot/types';
import { Enum, EnumType } from '@polkadot/types/codec';

// Example 1: Simple enum

class VoteKind extends Enum {
  constructor (value?: any) {
    super([
      'Approve',
      'Reject',
      'Slash'
    ], value);
  }
}
```

```
typeRegistry.register({
  VoteKind
});

// Example 2: Parametrized enum

class Announcing extends BlockNumber { }
class Voting extends BlockNumber { }
class Revealing extends BlockNumber { }
class ElectionStage extends EnumType<Announcing | Voting | Revealing> {
  constructor(value?: any, index?: number) {
    super({
      Announcing,
      Voting,
      Revealing
    }, value, index);
  }
}

typeRegistry.register({
  Announcing,
  Voting,
  Revealing,
  ElectionStage
});
```

Print a message

有时我们需要在构建应用程序时打印调试消息。 此示例演示如何打印消息以调试Substrate运行时代码。

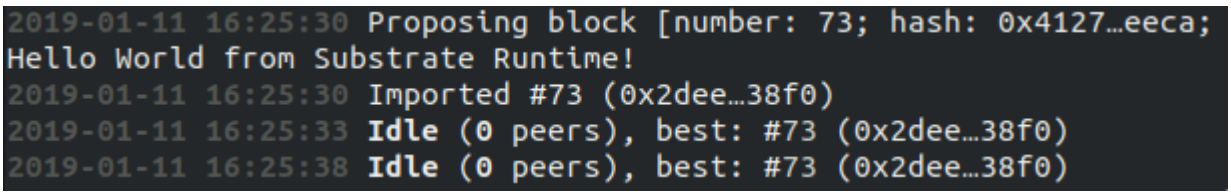
你需要在Substrate核心中包含`runtime_io`，以便在运行时模块中使用IO函数。

```
use support::{decl_module, dispatch::Result};
use runtime_io;

pub trait Trait: system::Trait {}

decl_module! {
  pub struct Module<T: Trait> for enum Call where origin: T::Origin {
    pub fn print_something(_origin) -> Result {
      // the following line prints a message on the node's terminal/console
      runtime_io::print("Hello World from Substrate Runtime!");
      Ok(())
    }
  }
}
```

执行此功能时，该消息将出现在运行Substrate节点的终端上。 以下屏幕截图显示了打印的消息。



余额转账

你可以使用 `Currency` 类型来实现安全的资金转移。请注意，此模块不适用存储类型。

```
use support::{decl_module, dispatch::Result, Currency, LockableCurrency};
use system::ensure_signed;

pub trait Trait: system::Trait {
    type Currency: LockableCurrency<Self::AccountId, Moment=Self::BlockNumber>;

    type Event: From<Event<Self>> + Into<<Self as system::Trait>::Event>;
}

decl_module! {
  pub struct Module<T: Trait> for enum Call where origin: T::Origin {
    fn transfer_proxy(origin, to: T::AccountId, value: T::Balance) -> Result {
      let sender = ensure_signed(origin)?;
      T::Currency::make_transfer(&sender, &to, value)?;

      Ok(())
    }
  }
}
```