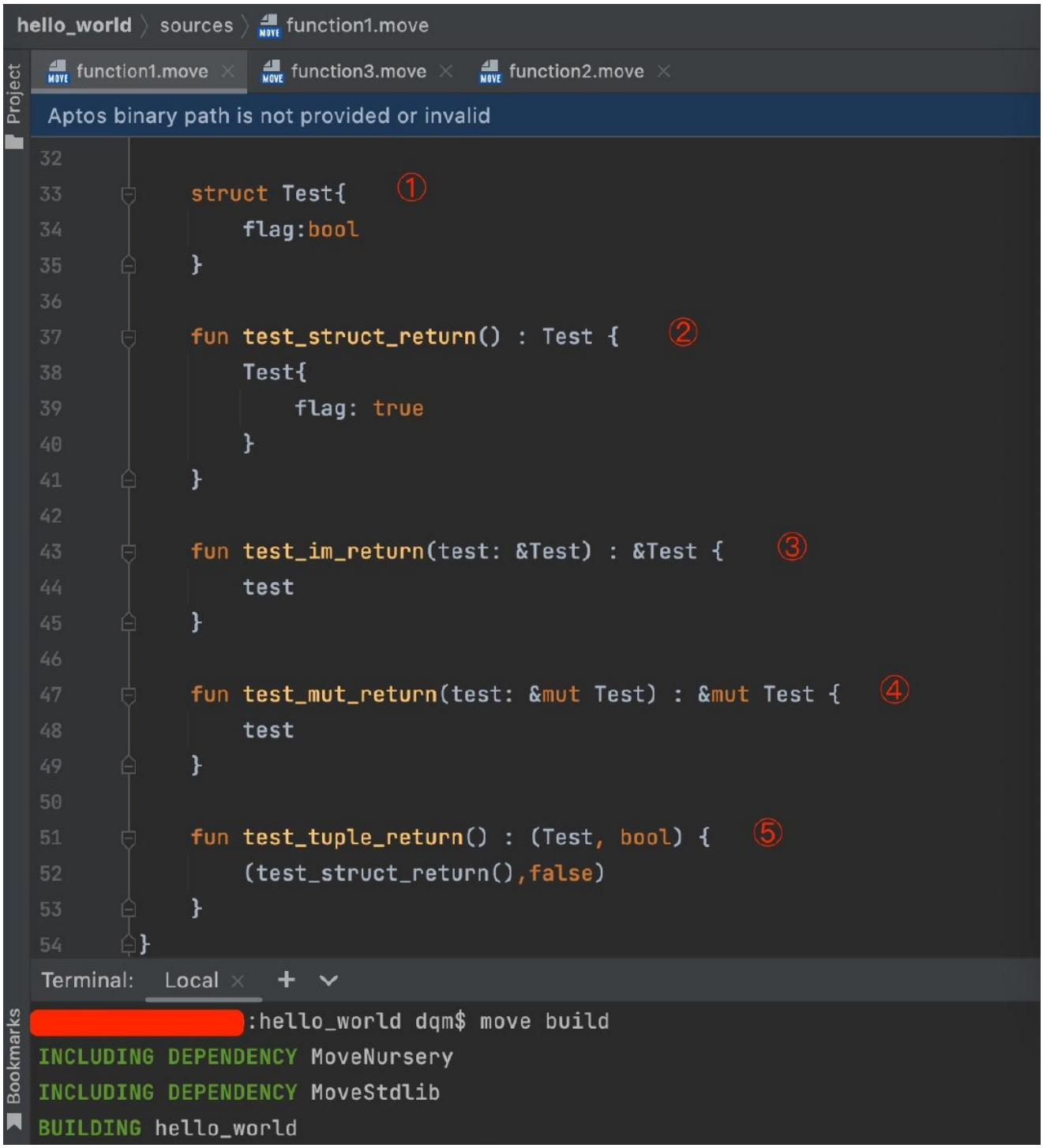
Move 高级语法

返回 Struct & 引用 & 元组

对比 Solidity，Move函数的返回值也很有特点。

Solidity的函数最早只能返回基本数据类型，后面增加了返回结构体的支持。但是，Move函数的返回值更加自由，能返回任何形式的数据，例如 `immutable ref`、`mutable ref`、tuple元组。



上面是Move函数返回各种类型的数据的例子，简单说明一下：

- ① 定义了一个strut
- ② 函数返回了一个struct
- ③ 函数返回了一个immutable的引用，以为着数据只读
- ④ 函数返回了一个immutable的引用，意味着数据可以被修改
- ⑤ 函数返回了一个tuple类型，调用了 ② 的函数

Patterns

> 设计模式 - https://move-by-example.com/core-move/patterns/index.html#patterns)

These patterns are closely related to the following Move language features :

- Resource oriented(资源导向)

    - Resource can only be stored, transfered, destroyed or dropped restricted to the abilities.
    - Resource can only be created, readed field, modified field in the module where it is defined.（资源只能在定义它的模块中创建、读取字段、修改字段）
- The global storage mechanism（全局存储机制）
    - The global storage can only be accessed（访问） through the `move_to`, `move_from`, `borrow_global`, `borrow_global_mut` functions.
    - Objects in the global storage can only be accessed in the module where it is defined.

Capability 模式

> https://move-by-example.com/core-move/patterns/capability.html#capability)

The `Capability` pattern can be used for access controll. It is the most widely .used pattern in Move smart contracts.
Capability 模式可用于访问控制，它是 Move smart contracts 中使用最广泛的模式

Why this pattern?

- Limited by that `no storage for modules to save data.``
    - 因为 module （合约）是不能储存数据的，所有的数据都被以资源的形式储存在 account 账户下面

How to use?

- construct a capability resource and `move_to` the receiptor.
- 定义一个 Capability 的 type，比如 ownerCapability 表示是一个 module 的拥有者，把这个资源对象发送给其他人，别人有了这个资源对象，就相当于有了权限操作 module 里的内容。

注意，是一个凭证，相当于是一个权限证明，不能 Copy, 不能 Drop 丢弃！ 要不就会有安全问题！

1. 下面的例子：将 Capability 存储到用户账户下的 Storage

```
module examples::capability {
  use std::signer;
  const ENotPublisher: u64 = 0;
  const ENotOwner: u64 = 1;

  struct OwnerCapability has key, store {}
  public entry fun init(sender: signer) {
    assert!(signer::address_of(&sender) == @examples, ENotPublisher);
    move_to(&sender, OwnerCapability{} );
  }
  // Only user with Capability can call this function.
  public entry fun grant_role() acquires OwnerCapability {
    let cap = borrow_global<OwnerCapability>(signer::address_if(&signer));
    // do s.th. with the capability ...
  }
}
```

2. 将 struct 传递给其他合约，其他合约可以通过这个对象 struct 来调用响应的函数。

```
module examples::capability {
  // same as above ...
```

```
  // The returned `OwnerCapability` can be stored in other modules.
  public entry fun get_owner_cap(sender: signer): OwnerCapability {
    assert!(signer::address_of(&sender) == @examples, ENotPublisher);
    OwnerCapability {}    // retured !
  }

  // module with `OwnerCapability` can call this Function !!
  public fun grant_role_with_cap(to: address, _cap: &OwnerCapability) acquires {
    // do s.th. ....
  }
```

官网例子（仔细看注释）

```
module examples::capability {
    use std::signer;

    const ENotPublisher: u64 = 0;
    const ENotOwner: u64 = 1;    // Error Code.

    struct OwnerCapability has key, store {}

    /// init and publish an OwnerCapability to the module owner.
    /// 初始化并向模块所有者发布 OwnerCapability
    public entry fun init(sender: signer) {
        // 该函数的执行人需要是模块 owner，否则 Error NotPublisher
        assert!(signer::address_of(&sender) == @examples, ENotPublisher);
        // 将该资源 move_to 到 sender 地址下
        move_to(&sender, OwnerCapability {})
    }

    /// mint to `_to` with the OwnerCapability.
    public fun mint_with_capability(_amount: u64, _to: address, _cap:
&OwnerCapability) {
        // mint and deposit to `_to` ,  mint and 存款
    }

    /// mint entry function. Only signer with OwnerCapability can call this
function.  mint 入口函数，只有拥有 OwnerCapability 的签名者才能调用此函数。
    public entry fun mint(sender: &signer, to: address, amount: u64) acquires
OwnerCapability {
        // 如果调用者 sender 不具备 OwnerCapability ，则 Error Not Owner
        assert!(exists<OwnerCapability>(signer::address_of(sender)), ENotOwner);

        // 接受 borrow_global 的返回值，borrow_global 会从全局状态中暂借出 owner 的
OwnerCapability 能力。
        let cap = borrow_global<OwnerCapability>(signer::address_of(sender));
        mint_with_capability(amount, to, cap);
    }
}
```

`OwnerCapability`：定义了（属主 owner 有？） key、store 能力（为全局存储操作的键值）

Offer 模式

> https://move-by-example.com/core-move/patterns/offer.html#offer)

The `Offer` pattern is used to transfer new objects to others, such as capability delegation, etc.
Offer 模式用于将新对象传递给其他对象，例如能力委托等。

This pattern is intraoduced because the restriction that in Move, `move_to` requires a `signer`, which means that an account cannot directly send an object to another account. So we need to use the `Offer` pattern to achieve this.
这种模式是内部引入的，因为在 Move 中，move_to 需要 `signer` 签署，这意味着 A 帐户不能直接无许可地将对象发送到 B 帐户（因为需要 `move_to` 到 B 账户下面去，而 `move_to` 这个动作需要 B 账户来 sign 签署）。所以我们需要使用 Offer pattern 来实现这一点。

发送者将要发送的对象放在一个 Offer 里面，然后 receptor 接收 Offer 对象，再放到自己的账户下面。

例子 1：
小伦想发送给小明一个 `AdminCapability`，但是小伦不知道小明什么时候在家 (小明无法实时 sign 签署)，所以不能贸然寄给小明。所以他们俩约定了一个流程：

- 1. 小伦将 `AdminCapability` 包裹在 `Offer` 里，里面写上了小明的 `address`（`receipt: to`）
- 2. 目标接收人（小明）如果想获取 `AdminCapability` 的话，需要自己去申请打开 Offer 包裹，即 claim 调用 `accept_role`，然后合约会检查小明的地址（`address_of(sender)`）是不是 Offer 包裹里写入的 address .（即：只有小明才能打开这个写着小明 address 的 Offer 包裹）

```
module examples::offer {
  struct Offer<T: key+store> has key, store {
    receipt: address,
    offer: T,
  }

  /// The owner can grant the `admin` role to another address `to`:
  public entry fun grant_role_offer(sender: &signer, to: address) acquires
OwnerCapability {
    assert!(exists<OwnerCapability>(signer::address_of(sender)), ENotOwner);
    move_to<Offer<AdminCapability>>(sender,
      Offer<AdminCapability> {
        receipt: to,
        offer: AdminCapability {},
      }
    );
  }

  /// The receiptor accept an `offer` from `grantor` and save it in his account.
  public entry fun accept_role(sender: &signer, grantor: address) acquires Offer {
    assert!(exists<Offer<AdminCapability>>(grantor), ENotGrantor);
    let Offer<AdminCapability> { receipt, offer: admin_cap =
move_from<Offer<AdminCapability>>(grantor);
    assert!(receipt == signer::address_of(sender), ENotReceipt);  // Attention.
    move_to<AdminCapability>(sender, admin_cap);
  }
}
```

真正的 Wrapper —— 把别的合约里的资源 Wrapper 起来自己用。

```
module examples::coin {
  struct Coin has key, store {
    value: u64,
  }
  struct MintCapability has key, store {}
  public fun mint(amount: u64, _cap: &MintCapability): Coin {
    Coin { value: amount }
  }
}

module examples::wrapper {
  use examples::coin::{Self, Coin, MintCapability};
  struct MintCapabilityWrapper has key {
    mint_cap: MintCapability,
  }
  public entry fun mint(sender: &signer, amount: u64) acquires MintCapabilityWrapper
{
    // check if the sender has can mint with own policy
    // 这边省略了一些检查-- 检查 sender 是否符合政策 -- 在可 mint 的白名单里
    // ..

    // 符合 ： 发放 mint 能力:
    let wrapper = borrow_global<MintCapabilityWrapper>(@examples);
    let coin = coin::mint(amount, &wrapper.mint_cap);
  }

}
```

官网例子，和上面的差不多。

```
module examples::offer {
    use std::signer;

    const ENotPublisher: u64 = 0;   // Error codes.
    const ENotOwner: u64 = 1;
    const ENotReceipt: u64 = 2;
    const ENotGrantor: u64 = 3;

    struct OwnerCapability has key, store {}

    struct AdminCapability has key, store {}

    // 限制 offer struct 传入的 T 需要有 key + store abilities （个人理解）
    struct Offer<T: key + store> has key, store {
        receipt: address,
        offer: T,
    }

    // 和上面 Capability 一样，初始化并向模块所有者发布 OwnerCapability
    public entry fun init(sender: signer) {
        assert!(signer::address_of(&sender) == @examples, ENotPublisher);
        move_to(&sender, OwnerCapability {})
    }

    // 将 capability 授予给 `to`，即 receipt，
    // return 一个 Offer struct with AdminCapability.
```

```
    public fun grant_role_with_capability(to: address, _cap: &OwnerCapability):
Offer<AdminCapability> {
        Offer<AdminCapability> {
            receipt: to,
            offer: AdminCapability {},
        }
    }

    /// The owner can grant the admin role to another address `to`
    // owner 可以将 AdminCapability (admin role) 授予另一个地址`to`
    public entry fun grant_role_offer(sender: &signer, to: address) acquires
OwnerCapability {
        assert!(exists<OwnerCapability>(signer::address_of(sender)), ENotOwner);
        // 1. 签名者把自己的 OwnerCapability 揪出来
        let cap = borrow_global<OwnerCapability>(signer::address_of(sender));
        // 2. 将 OwnerCapability 放入，取回一个 AdminCapability 能力类型的 offer
        let offer = grant_role_with_capability(to, cap);
        // 3. 将 offer （AdminCapability能力） 塞入自己的地址（move_to ）
        move_to<Offer<AdminCapability>>(sender, offer);
    }

    /// Entry function for `receiptor` to accept an offer from `grantor` and save it
in their account.
    // `receiptor` 接受来自 `grantor` 的 offer 并将其保存在他们的帐户中
    public entry fun accept_role(receiptor: &signer, grantor: address) acquires
Offer {
        assert!(exists<Offer<AdminCapability>>(grantor), ENotGrantor);
        let Offer<AdminCapability> { receipt, offer: admin_cap } =
move_from<Offer<AdminCapability>>(grantor); // 移出
        assert!(receipt == signer::address_of(receiptor), ENotReceipt);
        move_to<AdminCapability>(receiptor, admin_cap);
    }
}
```

Wrapper 模式

> https://move-by-example.com/core-move/patterns/wrapper.html#wrapper

如上 Offer 类型，其局限是某个地址下只能存储一个资源，**无法分发给很多人**，所以实际中更常用的是 Wrapper 模式。

The Wrapper pattern is used to store an arbitrary number of a given type, or store objects defined in other modules.

> Wrapper 模式用于存储任意数量的给定类型，或存储其他模块中定义的对象

Why this pattern?

1. Limited by that an account can only have one resource of a given type.
    1. 一个账户只能拥有一个给定类型的资源。
2. Limited by that one can't `move_to` or `move_from` a resource outside the module.
    1. 不能 `move_to` 或 `move_from` 模块外的资源，使用了 Wrapper 后，可以再包一层，这样就能在 module 之外操作别人的 Resource 了.....

How to use?

1. Use a `vector` or a `table` to store the objects.
    1. 使用 `vector` 或者 `table` 数据结构来存储多个 Object，来分发给多个地址。
2. Wrapper an object in a new struct directly.

Applications:

- `NFTGalley::NFTGalley` in StarCoin framework .
- `token::TokenStore` in Aptos framework.

如下示例：

- `offers: Table<address, T>` : offers 里可以存放多个 address ， 也就是说： 可以把比如 `AdminCapability` 对象分发给多个地址。
- 重复调用 `grant_admin_offer(to: address)` , 将 `address, AdminCapability {}` 添加到 `&offer_store.offers` 里去
- address 来 claim 调用 accept_role 申请 `AdminCapability` 时， 合约会判断这个 address 在不在 `&store.offers` 的名单里，如果在名单里，就 `table::remove(&mut store.offers, to);` 弹出一个 `AdminCapability{}` 给 claimer

```
module examples::wrapper {
    use std::signer;
    use extensions::table::{Self, Table};

    const ENotPublisher: u64 = 0;
    const ENotOwner: u64 = 1;
    const ENotReceipt: u64 = 2;
    const ENotGrantor: u64 = 3;
    const EOfferExisted: u64 = 4;

    struct OwnerCapability has key, store {}

    struct AdminCapability has key, store {}

    /// The wrapper pattern.
    struct OfferStore<phantom T> has key, store {
        offers: Table<address, T>,
    }

    public entry fun init(sender: signer) {
        assert!(signer::address_of(&sender) == @examples, ENotPublisher);
        move_to(&sender, OwnerCapability {})
    }

    /// The owner can grant the admin role to another address `to`
    public entry fun grant_admin_offer(sender: &signer, to: address) acquires
OfferStore
    {
        assert!(exists<OwnerCapability>(signer::address_of(sender)), ENotOwner);
        if (!exists<OfferStore<AdminCapability>>(signer::address_of(sender))) {
            move_to<OfferStore<AdminCapability>>(sender, OfferStore<AdminCapability>
{
                offers: table::new(),
            });
        };
```

```
        let offer_store = borrow_global_mut<OfferStore<AdminCapability>>
(signer::address_of(sender));
        assert!(!table::contains<address, AdminCapability>(&offer_store.offers, to),
EOfferExisted);
        table::add(&mut offer_store.offers, to, AdminCapability {});
    }

    /// Entry function for `sender` to accept an offer from `grantor` and save it in
their account.
    public entry fun accept_role(sender: &signer, grantor: address)
    acquires OfferStore {
        assert!(exists<OfferStore<AdminCapability>>(grantor), ENotGrantor);
        let to = signer::address_of(sender);
        let store = borrow_global_mut<OfferStore<AdminCapability>>(grantor);
        assert!(table::contains<address, AdminCapability>(&store.offers, to),
EOfferExisted);
        let admin_cap = table::remove(&mut store.offers, to);
        move_to<AdminCapability>(sender, admin_cap);
    }
}
```

Witness 模式

> https://move-by-example.com/core-move/patterns/witness.html#witness)

Witness is a pattern that is used for confirming the ownership of a type. To do so, one passes a drop instance of a type. Coin relies on this implementation.
Witness 是一种用于确认类型所有权（the ownership of a type）的模式。
为此，需要传递一个类型的 drop 实例。 Coin 依赖于这个实现。

Why this pattern?

- Benifit from the Move type system, that a type can only be created by the module that defines it.
- 类型 type 只能由定义它的模块 module 创建。
- 得益于 Move 的特性 —— 一个类型实例化的时候，只能在定义这个类型的 Module 里面进行实例化。
- Witness 对象一定是创建对象的合约或者是授权的合约才能获取这个实例。

How to use?

- Define a public function with generic type argument, and a function argument of that type.
- 定义具有泛型类型参数的公共函数，以及该类型的函数参数。

Security programming

- **No copy ability** and public constructor function for the Witness type.
- Witness 类型没有 copy 能力和 public 构造函数。

Applications

- A third party library can provide public functions but limit the invoking.
- 第三方库可以提供公共功能但限制调用。
- In an open game, a hero module authorization other modules to increase a hero's expericence.

- 在一个开放游戏中，一个英雄模块授权（某个可信任可约）其他模块来增加一个英雄的经验。

如下例子，Hacker 想要攻击这个合约：

1. Hacker 知道 Publisher 还没来得及 `publish_coin`，黑客想利用这个时间差抢先发币：
2. Hacker 想利用 `examples::framework` 的 `publish_coin()` 和 `examples::xcoin` 的 `X` 类型来发行货币
3. 但是得益于 Move 的特性 —— 类型实例化的时候，只能在定义这个类型的 Module 里面进行实例化。
    1. 也就是说，publish() 中的类型 T 在 Module `examples::xcoin` 中被实例化为 `X`，其只能在 examples Module 中被实例化(应该是)， 所以外部 Module 无法调用 `X` 进行实例化
4. `publish_coin` 和 `publish_coin_v2` 传值/传引用，可以达到相同的效果。

```
module examples::framework {
    /// Phantom parameter T can only be initialized in the `create_guardian`
    /// function. But the types passed here must have `drop`.
    // Phantom param `T` 只能在 `create_guardian()` 中初始化，这里传递的类型必须有
`drop`。
    struct Coin<phantom T: drop> has key, store {
        value: u128,
    }

    /// The first argument of this function is an actual instance of the
    /// type T with `drop` ability. It is dropped as soon as received.
    // 该函数的第一个参数是具有“drop”能力的类型 T 的实际实例。 它一收到就被丢弃。
    public fun publish_coin<T: drop>(_witness: T) {
        // register this coin to the registry table
    }

    public fun publish_coin_v2<T: drop>(_witness: &T) {
        // register this coin to the registry table, it's also work well.
    }

}

/// Custom module that makes use of the `guardian`.
module examples::xcoin {
    use examples::framework;    // Use the `guardian` as a dependency.

    struct X has drop {}

    /// Only this module defined X can call framework::publish_coin<X>
    // 只有这个模块定义的 X 可以调用 framework::publish_coin<X>
    public fun publish() {
        framework::publish_coin<X>(X {});
    }
}

// Hack it !
module hacker::hacker {
        use examples::framework;    // Use the `guardian` as a dependency.
        use examples::xcoin::X;
        public fun publish() {
          coin::publish_coin<X>( X { } ); // Illegal, X can not be constructed here.
        }
}
```

Hot Potato 模式

> 烫手山芋，笑死 🤣

- Hot Potato is a name for a struct that has no abilities, hence it can only be packed and unpacked in its module.
- struct with no abilities, 它只能在其 module 中打包和解包。
- In this struct, you must call function B after function A in the case where function A returns a potato and function B consumes it.
- 在这个结构中，你必须在函数 A 之后调用函数 B，以防 A 返回一个 potato 而 B 消费它。

该 Design Pattern 实现的逻辑：

- 可以控制函数们的执行顺序：
- 比如说我做了一个基础库，在不知道调用者会干什么的时候，通过 Hot Potato 模式，让别人在预先设定的顺序下去调用我现有的函数。

如下例子：

- 定义了一个圆圆的土豆，它既不能直接吃，又不能存起来(马铃薯会发芽)，只能传递给下一个人 (函数)。
- 如果调用 `get_potato` 函数，该函数返回一个 `Potato {}` 对象，用户是没法处理的
- 所以用户必须在其后调用 `consume_potato` 来消费这个 `Potato {}` 对象
- 如此，就实现了 `get_potato -> consume_potato` 这样的一个函数执行顺序。

```
module examples::hot_potato {

    /// Without any capability, the `sender` can only call `consume_potato`.
    struct Potato {}

    /// When calling this function, the `sender` will receive a `Potato` object.
    /// The `sender` can do nothing with the `Potato` such as store, drop, except
    /// passing it to `consume_potato` function.
    public fun get_potato(_sender: &signer): Potato {
        Potato {}
    }

    public fun consume_potato(_sender: &signer, potato: Potato) {
        // do nothing
        let Potato {} = potato;
    }
}
```

Move 可见性

- private：当前Module可见（Move函数的默认可见性<这点跟Solidity不一样，Solidity 默认是public的，move 如此能很好的避免函数的泄露）
- friend：当前address下被授权的Module可见（可以理解为address级别的可见性，类比Java的package级别的可行性。Solidity通常使用Modifier来控制调用权限，这依赖开发者的个人能力，容易引发安全隐患，被黑客绕过）
- public：所有Module可见，但是用户不能直接作为交易使用（跟Solidity不一样，Solidity是调用public的合约函数发起链上交易）
- entry或者script：交易的入口（可以理解为可编程的交易，在讲Module的时候再详细介绍）

以上4种可见性，可见范围逐渐放大。

friend：`friend` 的作用是控制「相同address下，跨Module的函数调用」，这样精细化的访问控制，能更好的避免函数泄露的问题。下来是`friend`使用的例子。

```
... 等会写上
```

generic 泛型

When calling a generic function, one can specify the type arguments for the function's type parameters in a list enclosed by a pair of angle brackets.
调用泛型函数时，可以在由一对尖括号括起来的列表中指定函数类型参数的类型参数。

```
fun foo() {
    let x = id<bool>(true);
}

// Generic Structs
fun foo() {
    let foo = Foo<bool> { x: true };
    let Foo<bool> { x } = foo;
}


//  infer the type 自动类型推断
    let x = id(true);
    //        ^ <bool> is inferred

    let foo = Foo { x: true };
    //           ^ <bool> is inferred


// 不能自动推断时必须手动指定：
    fun foo() {
        // let v = vector::new();
        //                   ^ The compiler cannot figure out the element type.

        let v = vector::new<u64>();
        //                 ^~~~ Must annotate manually.
    }

// However, the compiler will be able to infer the type if that return value is used
later in that function
    fun foo() {
```

```
        let v = vector::new();
        //                  ^ <u64> is inferred
        vector::push_back(&mut v, 42);   // 42, so <u64> is inferred
    }
```

If you do not specify the type arguments, Move's type inference will supply them for you.

Unused Type Params

> (未使用的类型参数)

For a struct definition, an unused type parameter is one that does not appear in any field defined in the struct, but is checked statically at compile time. Move allows unused type parameters so the following struct definition is valid:
“未使用的类型参数” 是 —— 没有出现在结构体定义的任何字段中，但在编译时会静态检查的类型参数。Move语言允许未使用的类型参数，因此以下结构定义是有效的：

```
struct Foo<T> {
    foo: u64  // `T` is unused
}
```

This can be convenient when modeling certain concepts. Here is an example:
这在对某些概念进行建模时会很方便。这是一个例子：

```
address 0x2 {
  module m {
    // Currency Specifiers 货币说明符
    struct Currency_CNY {}   // RMB
    struct Currency_USD {}   // dollars

    // A generic coin type that can be instantiated using a currency specifier type.
可以使用 `货币说明符`(指定某种货币类型) 来实例化的 `通用 coin type` 。
    // e.g. Coin<Currency_CNY>, Coin<Currency_USD> etc.
    struct Coin<Currency> has store {
        value: u64
    }

    // Write code generically about all currencies 有关所有货币的通用代码
    public fun mint_generic<Currency>(value: u64): Coin<Currency> {
        Coin { value }
    }

    // Write code concretely about one currency 具体编写某种货币(如 USD) 的代码
    public fun mint_concrete(value: u64): Coin<Currency_USD> {
        Coin { value }
    }
  }
}
```

Phantom（幻影）

举个 rust 中的例子：

```
// 带有 `A` 和 phantom type `B` 的泛型结构体：
struct PhantomTuple<A, B>(A,PhantomData<B>);

// 注意：对于泛型 `A` 会分配存储空间，但 `B` 不会。
// 因此，`B` 不能参与运算。
```

phantom type parameter 是一种在运行时（runtime）不出现，而在（且只在）编译期进行静态方式检查的 parameter。

在上面的例子中，虽然 `struct Coin` 被要求有 `store` 能力，但 `Coin<Currency_CNY>` 和 `Coin<Currency_USD>` 都没有 `store` 能力。

这是因为 条件能力与泛型类型的规则, 因为 `Currency_CNY` 和 `Currency_USD` 没有 `store` 能力，尽管它们甚至没有在 `struct Coin` 的主体中使用，但这可能会导致一些不好的后果。例如，我们无法将 `Coin<Currency_CNY>` 放入全局存储的一个钱包中。

Phantom 类型参数解决了这个问题。未使用的类型参数可以标记为 Phantom 类型参数，不参与结构的能力推导。这样，在派生泛型类型的能力时，不考虑幻像类型参数的参数，从而避免了对虚假能力注释的需要。为了使这个宽松的规则合理，Move 的类型系统保证声明为 phantom 的参数

- 要么在结构定义中根本不使用，
- 要么仅用作也声明为 phantom 的类型参数的参数(没懂)

更正式地说，如果将类型用作 phantom 类型参数的输入参数，我们说该类型出现在 *phantom 位置*。有了这个定义，正确使用 phantom 参数的规则可以指定如下： **phantom 类型参数只能出现在 phantom 位置。** （不好理解，要结合下面的实例对比来看：

```
struct S1<phantom T1, T2> { f: u64 }
                  ^^
                  Ok : T1 does not appear inside the struct definition


struct S2<phantom T1, T2> { f: S1<T1, T2> }
                                  ^^
                                  Ok: T1 appears in phantom position
                                  // ↑ above S1<phantom T1, T2>
```

以下代码展示违反规则的示例：

```
struct S1<phantom T> { f: T }
                          ^
                          Error: Not a phantom position

struct S2<T> { f: T }  // OK

struct S3<phantom T> { f: S2<T> }
                             ^
                             Error: Not a phantom position
```

实例化 (Instantiation)

When instantiating a struct, the arguments to phantom parameters are excluded when deriving the struct abilities. For example, consider the following code:

实例化结构时，派生结构功能时会排除幻影参数的输入参数。例如，考虑以下代码：

```
struct S<T1, phantom T2> has copy { f: T1 }
struct NoCopy {}
struct HasCopy has copy {}
```

Consider now the type `S<HasCopy, NoCopy>`. Since `S` is defined with `copy` and all non-phantom arguments have copy then `S<HasCopy, NoCopy>` also has copy.
考虑类型 `S<HasCopy, NoCopy>`。由于 `S` 用 `copy` 定义，且所有非 phantom 参数具有 `copy` 能力，所以 `S<HasCopy, NoCopy>` 也具有 `copy` 能力。

实例-BaseCoin

https://github.com/move-language/move/blob/main/language/documentation/tutorial/step_6/BasicCoin/sources/BasicCoin.move

原来的 Coin 类型：

```
    struct Coin has store {
        value: u64
    }

    /// Struct representing the balance of each address.
    struct Balance has key {
        coin: Coin
    }
```

注意下面都是 `<CoinType>`：

```
struct Coin<phantom CoinType> has store {
        value: u64
}

struct Balance<phantom CoinType> has key {
        coin: Coin<CoinType>
}

/// Publish an empty balance resource under `account`'s address. This function must
be called before minting or transferring to the account.
public fun publish_balance<CoinType>(account: &signer) {
  let empty_coin = Coin<CoinType> { value: 0 };
  assert!(!exists<Balance<CoinType>>(signer::address_of(account)),
EALREADY_HAS_BALANCE);
  move_to(account, Balance<CoinType> { coin:  empty_coin });
}

/// Mint `amount` tokens to `mint_addr`. This method requires a witness with
`CoinType` so that the module that owns `CoinType` can decide the minting policy. 此
方法 requires `CoinType` 的 witness(见证人？)，以便拥有 `CoinType` 的模块可以决定铸币策略
public fun mint<CoinType: drop>(mint_addr: address, amount: u64, _witness: CoinType)
```

```
acquires Balance {
        // Deposit `total_value` amount of tokens to mint_addr's balance
        deposit(mint_addr, Coin<CoinType> { value: amount });
}
```

我们提供了一个名为 `MyOddCoin` 的小模块，它实例化 Coin 类型并自定义其 Transfer 策略：只能转移奇数个硬币。
only odd number of coins can be transferred

```
module NamedAddr::MyOddCoin {
    use std::signer;
    use NamedAddr::BasicCoin;

    struct MyOddCoin has drop {}

    const ENOT_ODD: u64 = 0;  // status_code ;

    public fun setup_and_mint(account: &signer, amount: u64) {
        BasicCoin::publish_balance<MyOddCoin>(account);
        BasicCoin::mint<MyOddCoin>(signer::address_of(account), amount, MyOddCoin
{});
    }

    public fun transfer(from: &signer, to: address, amount: u64) {
        // amount must be odd.
        assert!(amount % 2 == 1, ENOT_ODD);
        BasicCoin::transfer<MyOddCoin>(from, to, amount, MyOddCoin {});
    }

    /*
        Unit tests, 执行 move test 时会运行下面 2 个函数来进行单元测试
    */
    #[test(from = @0x42, to = @0x10)]
    fun test_odd_success(from: signer, to: signer) {
        setup_and_mint(&from, 42);
        setup_and_mint(&to, 10);

        // transfer an odd number of coins so this should succeed.
        transfer(&from, @0x10, 7);

        assert!(BasicCoin::balance_of<MyOddCoin>(@0x42) == 35, 0);
        assert!(BasicCoin::balance_of<MyOddCoin>(@0x10) == 17, 0);
    }

    #[test(from = @0x42, to = @0x10)]
    #[expected_failure]
    fun test_not_odd_failure(from: signer, to: signer) {
        setup_and_mint(&from, 42);
        setup_and_mint(&to, 10);

        // transfer an even number of coins so this should fail.
        transfer(&from, @0x10, 8);
    }
}
```

Vector

标准库中的 Vector 模块：

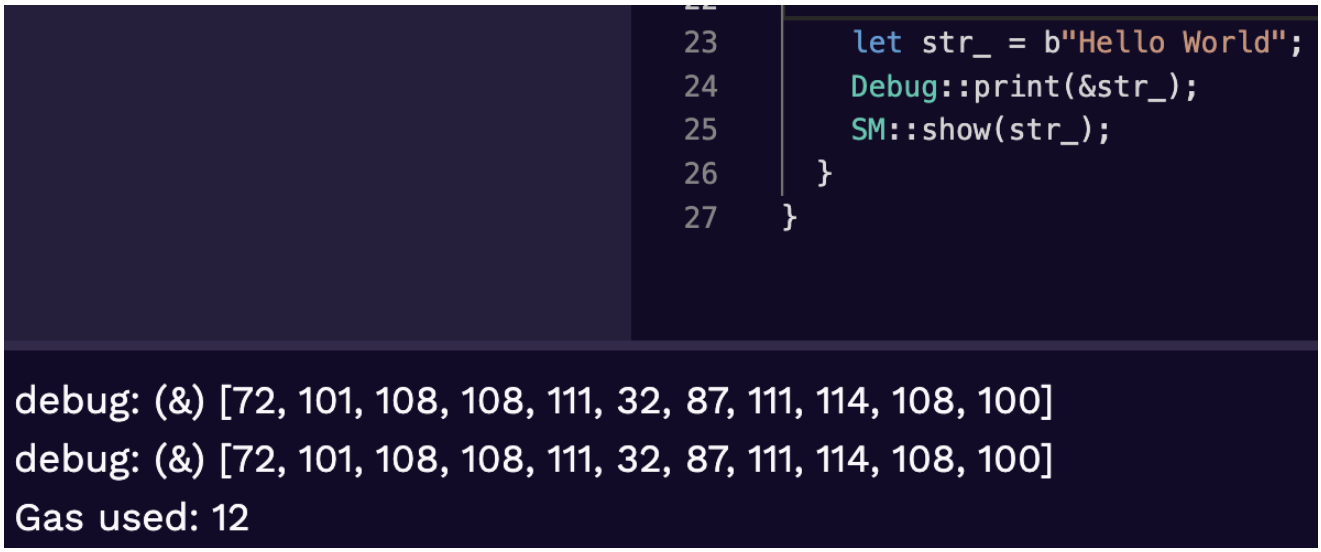
- Diem diem/diem
- Starcoin starcoinorg/starcoin

vector 是 Move 提供的泛型容器。Vector 功能实际上是由 VM 提供的，不是由 Move 语言提供的，使用它的唯一方法是使用标准库和 native 函数。

```
let str_ = b"hello";             // cast "hello" as Vector of Ascii
let v2 = Vector::empty<u64>();
Vector::push_back(&mut v2, 10);
```

1. 打印 “字符串”

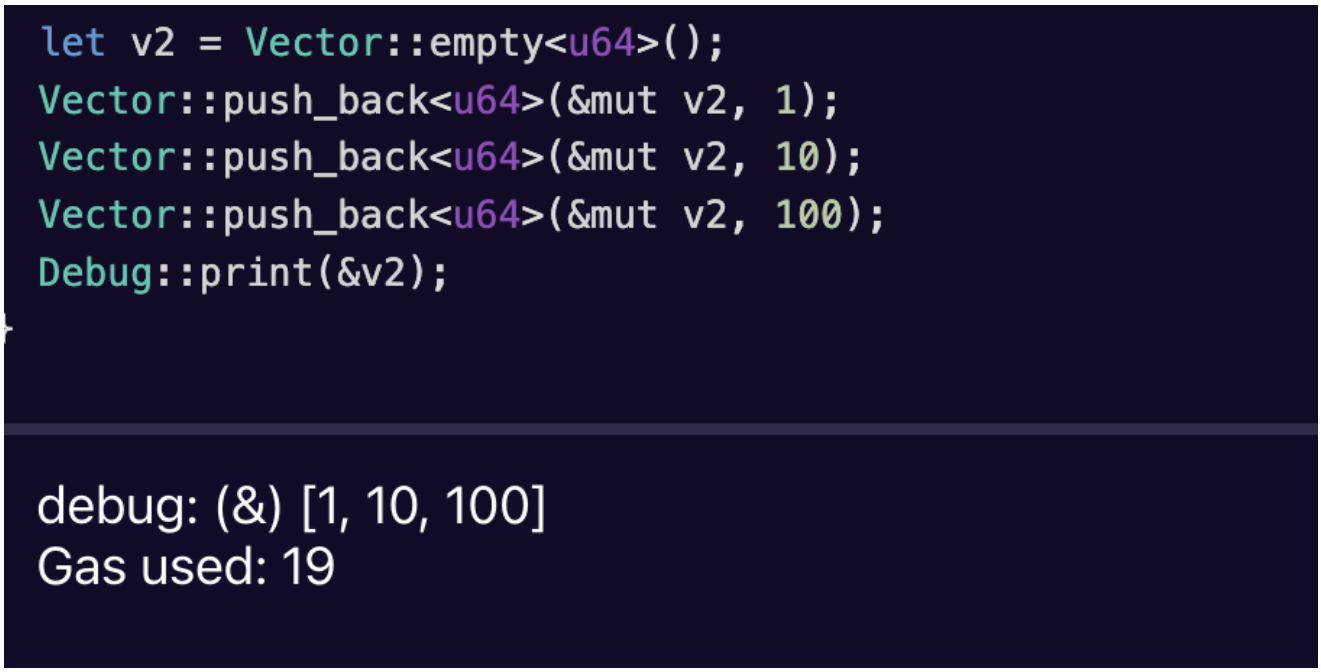
```
    let str_ = b"Hello World";
    Debug::print(&str_);
    SM::show(str_);   // Attention No &str_ , but str_ , why ?
```



1. 向 Vector Push Value：

```
script {
  use 0x1::Debug;
  use Sender::Math as SM;
  use 0x1::Vector;   // maybe lowcase as 'vector', need test ?

  fun testSum() {
    let v2 = Vector::empty<u64>();
    Vector::push_back<u64>(&mut v2, 1);
    Vector::push_back<u64>(&mut v2, 10);
    Debug::print(&v2);
  }
}
```



Vector 速查表

这是标准库中 Vector 方法的简短列表：

- 创建一个类型为 `<E>` 的空向量
  `Vector::empty<E>(): vector<E>;`
- 获取向量的长度
  `Vector::length<E>(v: &vector<E>): u64;`
- 将元素 e 添加到向量末尾
  `Vector::push_back<E>(v: &mut vector<E>, e: E);`
- 获取对向量元素的可变引用。不可变引用可使用 `Vector::borrow()`
  `Vector::borrow_mut<E>(v: &mut vector<E>, i: u64): &E;`
- 从向量的末尾取出一个元素
  `Vector::pop_back<E>(v: &mut vector<E>): E;`

Vector API

终止：即会不会产生 abort 类型的终止异常

`empty<T>(): vector<T>`	
`singleton<T>(t: T):vector<T>`	
`borrow<T>(v: &vector<T>, i:u64): &T`	返回在 index i 处对 T 的不可变引用

函数	描述	终止
vector::empty<T>(): vector<T>	创建一个可以存储T类型值的空数组	永不
vector::singleton<T>(t: T): vector<T>	创建一个包含t的大小为1的数组	永不中止
vector::push_back<T>(v: &mut vector<T>, t: T)	将t添加到v的尾部	永不中止
vector::pop_back<T>(v: &mut vector<T>): T	移除并返回v中的最后一个元素	如果v是空数组
vector::borrow<T>(v: &vector<T>, i: u64): &T	返回在索引i处对T的不可变引用	如果i 越界
vector::borrow_mut<T>(v: &mut vector<T>, i: u64): &mut T	返回在索引i处对T的可变引用	如果i越界

函数	描述	终止
vector::destroy_empty<T>(v: vector<T>)	销毁 v数组	如果v 不是空数组
vector::append<T>(v1: &mut vector<T>, v2: vector<T>)	将v2中的元素添加到v1的末尾	永不中止
vector::contains<T>(v: &vector<T>, e: &T): bool	如果e在数组v里返回true	永不中止
vector::swap<T>(v: &mut vector<T>, i: u64, j: u64)	交换数组v中第i个和第j个索引处的元素	如果i或j越界
vector::reverse<T>(v: &mut vector<T>)	就地反转数组v中元素的顺序	永不中止

函数	描述	终止
vector::index_of<T>(v: &vector<T>, e: &T): bool	如果e在索引i处的数组中，则返回(true, i)否则返回(false, 0)	永不中止
vector::remove<T>(v: &mut vector<T>, i: u64): T	移除数组v中的第i个元素，移动所有后续元素。这里是O(n)时间复杂度且保留了数组中元素的顺序。	如果 i 越界
vector::swap_remove<T>(v: &mut vector<T>, i: u64): T	将数组中的第i个元素与最后一个元素交换，然后弹出。这里是O(1)时间复杂度，但是不保留数组中的元素顺序。	如果i越界

Resource 资源

Move 的主要功能是提供了自定义 Resource 类型。Resource 类型为安全的数字资产编码具提供了丰富的可编程性

Resource 是一种特殊的 `结构体 struct` （是被限制了 ability 的 struct），可以在 Move 代码中定义和创建，也可以使用现有的 Resource。因此，我们可以像使用任何其它数据（比如向量或结构体）那样来管理数字资产。

Resource 只具有 `key` 和 `store` 2 种 ability，Resource 永远不能被复制，重用或丢弃。 Resource 类型只能由定义该类型的模块创建或销毁。这些检查由 Move 虚拟机通过字节码校验强制执行。Move 虚拟机将

拒绝运行任何尚未通过字节码校验的代码。

Resource 必须存储在账户下面，且一个账户同一时刻只能容纳一个资源

Resource can only be created, readed field, modified field in the module where it is defined.
The global storage mechanism

- The global storage can only be accessed through the `move_ to` , `move_from` , `borrow_ global` , `borrow_global_mut` functions.
- Objects in the global storage can only be accessed in the module where it is defined.
  - 也就是说对象（资源）被转移出去之后，是没法通过 `move_ to` , `move_from` 这些函数访问的。

总结下：

1. Resource 是一种 特殊 struct
2. Resource 只具有 key 和 store 2 种 ability，不能被复制，重用或丢弃
3. 一个账户同一时刻只能容纳一个 Resource（一个某类型的 Resource）
4. Resource 必需被 `使用` ，这意味着必须将新创建的 Resource `move` 到某个帐户下，从帐户移出的 Resource 必须被解构或存储在另一个帐户下。

存储 `store`

- `store` 能力允许具有这种能力的类型的值存在于全局存储中的结构体（资源）内部，但不一定作为全局存储中的顶级资源。如果一个值有 `store` 能力，则包含在该值内的所有值都要有 `store` 能力。

存储键 `key`

- `key` 能力允许将结构体用作存储标识符。换句话说， `key` 是一种作为顶级资源存储在全局存储中的能力。它允许在全局存储中创建、删除和更新资源。如果一个值有 `key` 能力，则包含在该值内的所有值都要有 `store` 能力。

全局存储 （以 Starcoin 为例）

`Move` 模块并没有自己的存储。相反，全局存储（我们称之为区块链状态）由地址索引。每个地址下都有 `Move` 模块（代码）和 `Move` 资源。

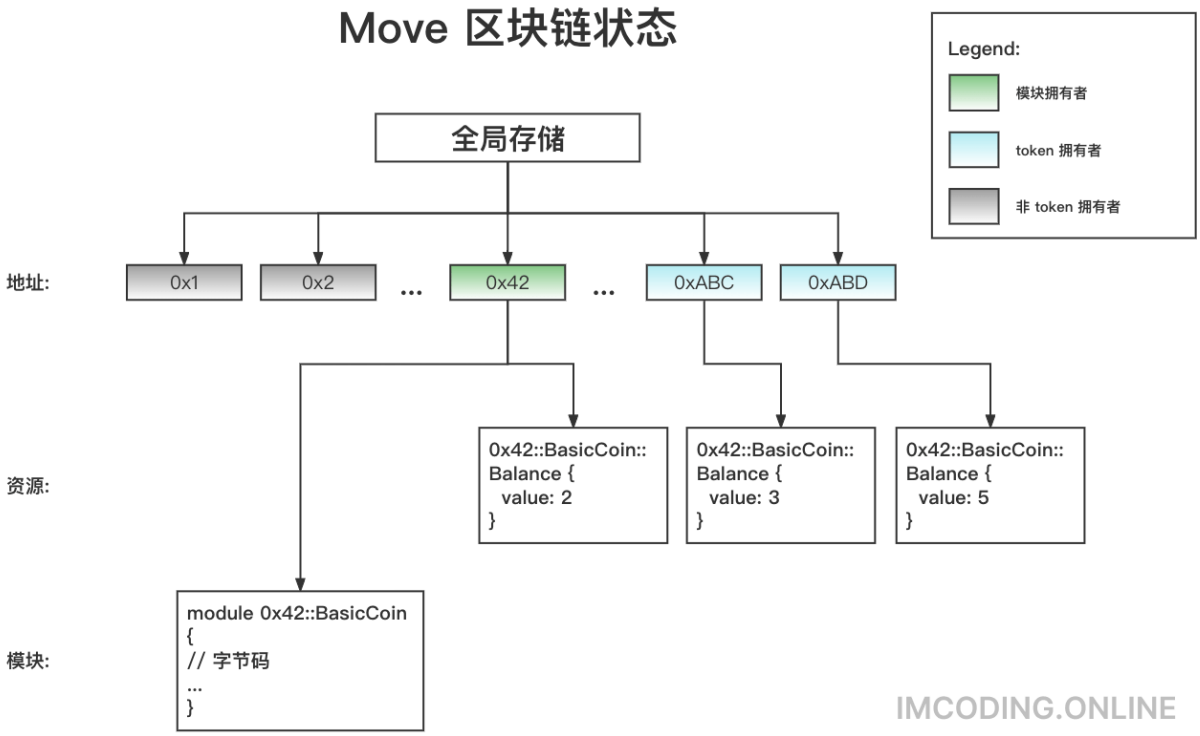
在伪代码中，全局存储看起来像：

```
struct GlobalStorage {
  resources: Map<(address, ResourceType), ResourceValue>
  modules: Map<(address, ModuleName), ModuleBytecode>
}
```

每个地址下的`Move`资源存储是一个类型到值的映射，也就是说每个地址下每种类型只能有一个值。这方便地为我们提供了按地址索引的原生映射：

```
module 0x42::BasicCoin {
  struct Balance has key {
    value: u64,
  }
}
```

在上面的模块中，我们定义了一个顶级资源 `Balance 余额`，它保存了每个地址持有的 token 数量。 此时，`Move` 区块链的状态大致如下所示：



signer 签署者

在开始使用 *Resource* 之前，我们需要了解 `signer` 类型以及这种类型存在的原因。

Signer 是一种原生的类似 Resource 的**不可复制**的类型，它包含了交易发送者的地址。 Signer 代表发送交易的人，**代表了发送者的权限**。换句话说，使用 signer 意味着可以访问发送者的地址和 Resource。它与 `signature` 没有直接关系，就 Move VM 而言，它仅表示发送者。

可以将其原生实现视为：

```
struct signer has drop {
  a: address,
}
```

Signer 类型不能在代码中创建，必须通过脚本调用传递

> Signer 只有一种 ability： `Drop` 。

> `use StarcoinFramework::Signer` ，是使用标准库下的 Signer module，Signer 是一种原生的类似 Resource 的不可复制的类型，它包含了交易发送者的地址。引入 signer 类型的原因之一是要明确显示哪些函数需要发送者权限，哪些不需要。因此，函数不能欺骗用户未经授权访问其 Resource。具体可参考源码。

```
script {
  // signer is an owned value
  fun main(account: signer) {
    let _ = account;
  } }
```

Signer 参数无需手动将其传递到脚本中，客户端（CLI）会自动将它放入你的脚本中。而且，signer 自始至终都只是引用，虽然标准库中可以访问签名者的实际值，但使用此值的函数是私有的，无法在其他任何地方使用或传递 signer 值。

一个交易脚本可以有任意数量的 `signer`， 只要这些 `signer` 参数在别的类型参数之前。换句话说，所有 `signer` 参数必须放在第一位：

```
script {
  use Std::Signer;

  fun main(s1: signer, s2: signer, x: u64, y: u8) {
    // ...
  }
}
```

这对于实现多方权限行为的 *多签脚本* 非常有用。

标准库中的 Signer 模块

`Std::Signer` 标准库模块提供了两个函数来从 `signer` 中获取地址。

- `address_of(&signer): address`：返回 signer 包装的地址
- `borrow_address(&signer): &address`：返回 signer 包装的地址的引用

> https://move-book.com/cn/resources/signer-type.html#%E6%A0%87%E5%87%86%E5%BA%93%E4%B8%AD%E7%9A%84-signer-%E6%A8%A1%E5%9D%97

原生类型离不开原生方法, signer 的原生方法包含在 `0x1::Signer` 模块中。

```
module Signer {
    // borrow_address: Borrows the address of the signer
    // Conceptually, you can think of the `signer` as being a resource struct
wrapper arround an address
    // 概念上讲，可以将 `signer` 视为 a resource struct wrapper arround an address:
    //   ```
    //   resource struct Signer { addr: address }
    //   ```
    // `borrow_address` borrows this inner field
    native public fun borrow_address(s: &signer): &address;

    // Copies the address of the signer
    public fun address_of(s: &signer): address {
        *borrow_address(s)
    }
}
```

使用 address_of：

```
    fun main(account: signer) {
        let _ : address = 0x1::Signer::address_of(&account);
    }
```

模块中的 Signer

```
module M {
  use 0x1::Signer;
  // let's proxy Signer::address_of
  public fun get_address(account: signer): address {
    Signer::address_of(&account)
  } }
```

> 使用 `&signer` 作为函数的参数说明该函数正在使用发送者的地址。

引入`signer`类型的原因之一是要明确显示哪些函数需要发送者权限，哪些不需要。因此，函数不能欺骗用户未经授权访问其 Resource。

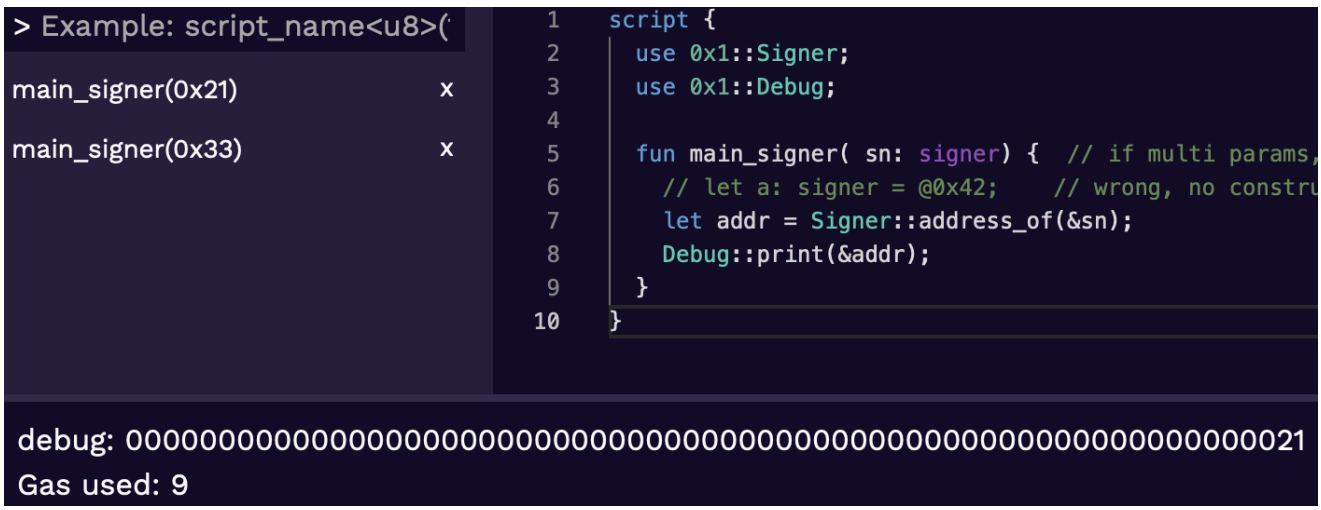
signer API

- `address_of ( &Signer ): address` 返回 signer 中的地址
- `borrow_address(&Signer): &address` 返回 SIgner 中地址的引用

```
script {
  use 0x1::Signer;
  use 0x1::Debug;

  fun main_signer( sn: signer) {  // if multi params, place signer first

    // let a: signer = @0x42;     // wrong, no constructor for signer
    let addr = Signer::address_of(&sn);
    Debug::print(&addr);
  }
}
```



- `Signer` 是 library
- `signer` 是数据类型
- 如果要穿多个参数，把 signer 类型放在第一个。
- signer 不能创建（没有构造函数），只能通过传递值调用。

```
public fun init(account: &signer){
    move_to(account, Counter{value:0});
}
```

如上，`init` 方法参数是一个 `&signer`， 意味着该方法必须是一个账户合法签名过后才可以调用

API

OK 回到 Resource。来看看 API：

name	description	abort
`move_ro<T>(&Signer, T)`	将 Resource T 发布给 Signer	if Signer 早已具有了 T
`move_from<T>(address): T`	删除 Signer 的 T 资源并返回	if Signer 没有 T
`borrow_global_mut<T>(address): &mut T`	返回 Signer 下 T 的可变引用	if Signer 没有 T
`borrow_global(address): &T`	返回 Signer 下 T 的不可变引用	if Signer 没有 T
`exists <T>(address): bool`	返回 address 下的是否具有 T 类型的资源	Never

create、move、Query Resource

① `move_to<T>(&signer, T)`：发布、添加类型为 T 的资源到 signer 的地址下。
② `exists<T>(address): bool`：判断地址下是否有类型为 T 的资源。。
③ `move_from<T>(addr: address): T` —— 从地址下删除类型为 T 的资源并返回这个资源。
④ `borrow_global< T >(addr: address): &T` —— 返回地址下类型为 T 的资源的不可变引用。
⑤ `borrow_global_mut< T >(addr: address): &mut T` —— 返回地址下类型为 T 的资源的可变引用。

```
// resouces/collection.move

// 一个模块里最主要的 Resource 通常跟模块取相同的名称（例如这里的 Collection）。遵循这个惯例，你的模块将易于阅读和使用。
module 0x1::collection{
  use 0x1::Vector;
  use 0x1::Signer;

  struct Item has store, drop {}

  // define resouce,
  // abilities of resouce only store & kkry
  struct Collection has store, key {
    items: vector<Item>,   // vector not Vector
  }
  // move resource to account.
  // 将资源发布到 account 下 !
  public fun start_collection(account: &signer){
    move_to<Collection>(account, Collection{
      items: Vector::empty<Item>()
    })
  }

  // judge exists ?
  public fun exists_at(at: address): bool {
    exists<Collection>(at)
  }
}
```

- `vector<Item>` 表示容器里面存放的是 `Item` 类型；

总结一下：

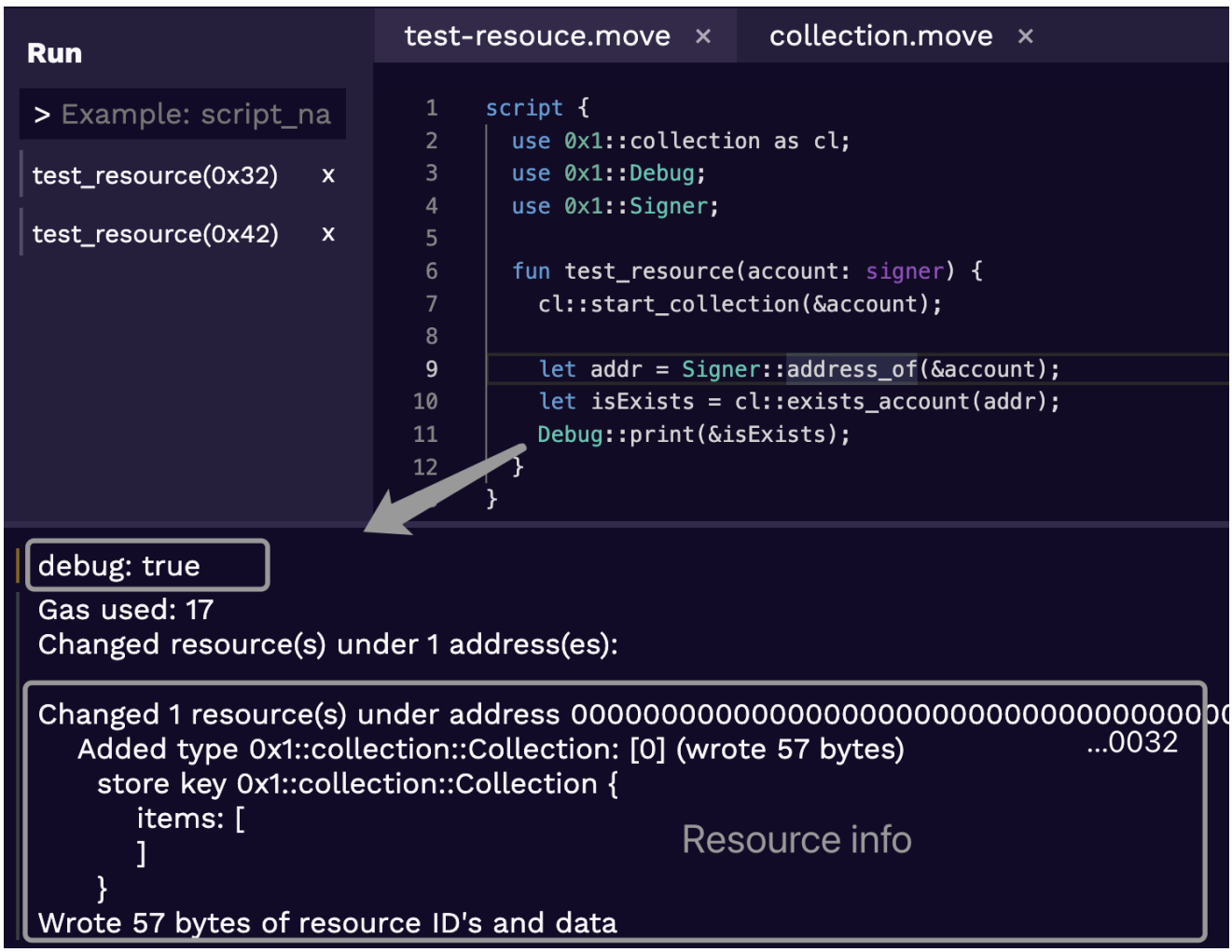
1. 你只能将 Resource 放在自己的帐户下。你无权访问另一个帐户的 `signer` 值，因此无法放置 Resource 到其它账户。
2. 一个地址下最多只能存储一个同一类型的 Resource。两次执行相同的操作是不行的，比如第二次尝试创建已有 Resource 将会导致失败。

测试：

```
// scripts/test-resource.move
script {
  use 0x1::collection as cl;
  use 0x1::Debug;
  use 0x1::Signer;

  fun test_resource(account: signer) {
    cl::start_collection(&account);

    let addr = Signer::address_of(&account);
    let isExists = cl::exists_account(addr);
    Debug::print(&isExists);
  }
}
```



Read, Modify Resource

- Read: `borrow_global<T: key>(addr: address): &T;`
  - 返回一个引用 reference
- Modify: `borrow_global_mut<T>(addr)`

```
module 0x1::collection{
  use 0x1::Vector;
  use 0x1::Signer;

  struct Item has store, drop {}

  // define resouce,
  // abilities of resouce only store & kkry
  struct Collection has store, key {
    items: vector<Item>,   // vector not Vector
  }
  // move resource
  public fun start_collection(account: &signer){
    move_to<Collection>(account, Collection{
      items: Vector::empty<Item>()
    })
  }

  // judge exists ?
  public fun exists_account(at: address): bool {
    exists<Collection>(at)
  }

  // modify
  // acquires return resource list
  public fun add_item(account: &signer) acquires Collection{
    // get the resource mutable quote
    let addr = Signer::address_of(account);
    let collection = borrow_global_mut<Collection>(addr);
    Vector::push_back(&mut collection.items, Item{});
  }

  // return resources length
  public fun size(account: &signer): u64 acquires Collection {
    let addr = Signer::address_of(account);
    let collection = borrow_global<Collection>(addr);
    Vector::length(&collection.items)
  }

}
```

测试：

① 取消 `cl::start_collection(&account);` 的注释，先创建

② 注释掉 `cl::start_collection(&account);` ，执行后续代码

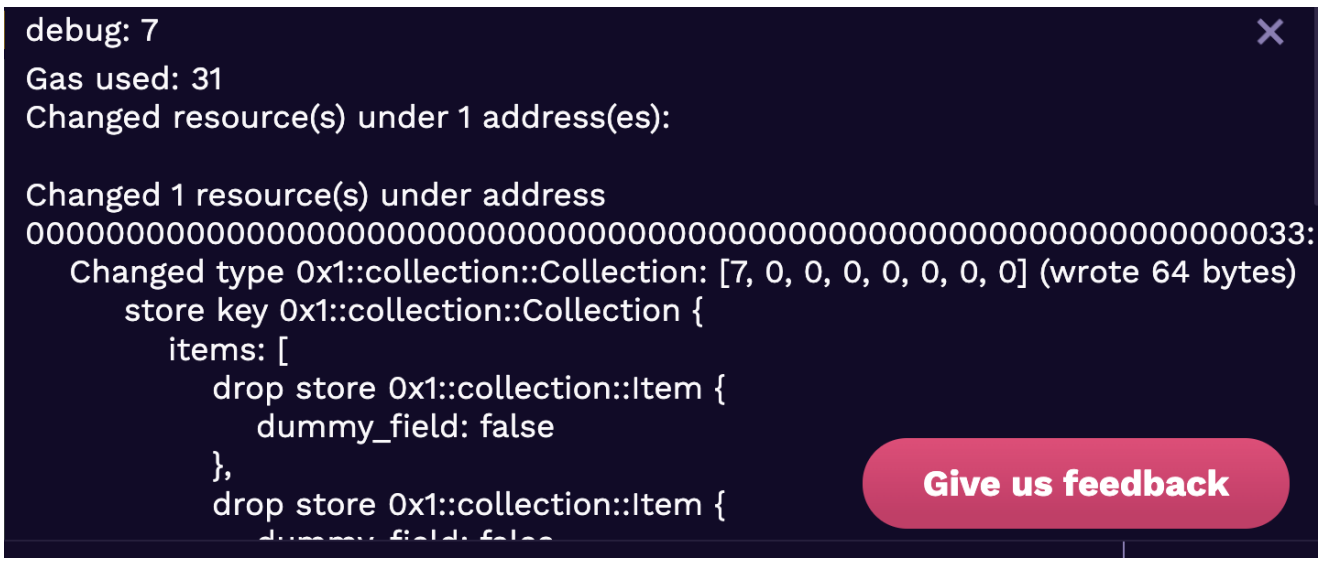
```
script {
  use 0x1::collection as cl;
```

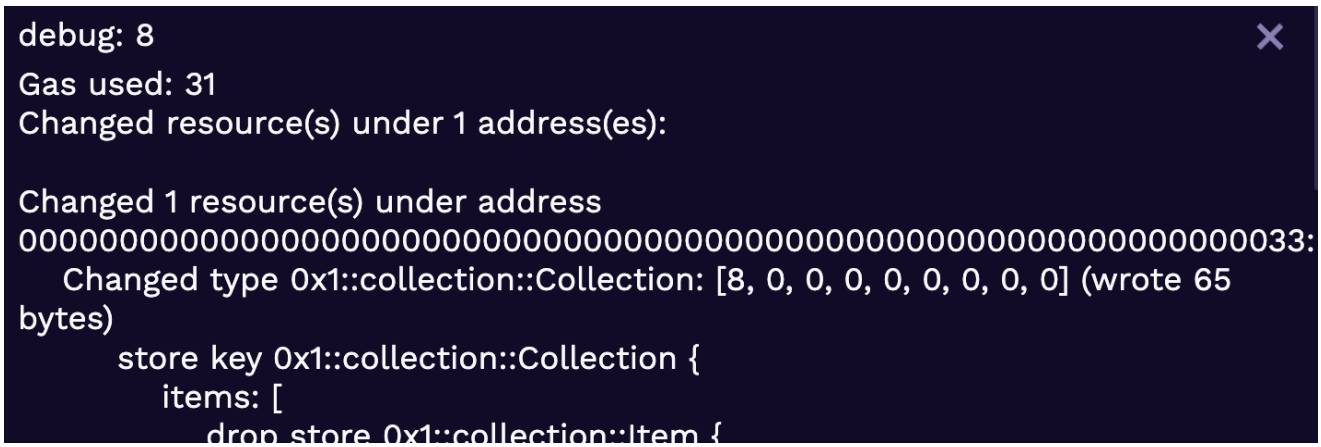
```
  use 0x1::Debug;
  use 0x1::Signer;

  fun test_resource(account: signer) {
    cl::start_collection(&account);

    // let addr = Signer::address_of(&account);
    // let isExists = cl::exists_account(addr);
    // Debug::print(&isExists);

    let addr = Signer::address_of(&account);
    cl::add_item(&account);
    let lsize = cl::size(&account);
    Debug::print(&lsize);
  }
}
```





- `*value_ref = *value_ref + v;`

```
// &mut b 修改资源:

module 0x42::BasicCoin {
  struct Balance has key {
    value: u64,
  }
```

```
  public fun increment(addr: address, v: u64) acquires Balance {
    let value_ref = &mut borrow_global_mut<Balance>(addr).value;
    *value_ref = *value_ref + v;
  }
}
```

Destroy Resource

```
module 0x1::collection{
  use 0x1::Vector;
  use 0x1::Signer;

  struct Item has store, drop {}
  // ...
  // Destroy
  public fun destroy(account: &signer) acquires Collection{
    let addr = Signer::address_of(account);
    // remove collection from address
    let collection = move_from<Collection>(addr);
    // DESTROY:
    let Collection{ items: _ } = collection;
  }
}
```

Advanced Data Structures

table

- table 是用结构体实现的键值对 Dict , 类似 Solidity 的 mapping
- table.move

`table.move` 的几个方法：

1. `upsert`： 没有就新增到 table 里，有就进行修改

simple_map

- simple_map.move

```
struct SimpleMap<Key, Value> has copy, drop, store {
  data: vector<Element<Key, Value>>,
}
struct Element<Key, Value> has copy, drop, store {
  key: Key,
  value: Value,
}
```

This module `simple_map` provides a solution for sorted maps, that is it has the properties that

1. Keys point to Values，键指向值
2. Each Key must be unique，每个 Key 是且必须是唯一的
3. A Key can be found within O(Log N) time，在 O(Log N) 时间内可以找到一个 Key
4. The data is stored as sorted by Key，按 Key 大小顺序存储

5. Adds and removals take O(N) time，添加和删除需要 O(N) 时间

property_map

- property_map.move
- property_map 是专为 Token 设计的数据结构
  - PropertyMap is a specialization of SimpleMap for Tokens.
  - It maps a String key to a PropertyValue that consists of type (string) and value (`vector<u8>`)
  - It provides basic on-chain serialization of primitive and string to property value with type information
  - It also supports deserializing property value to it original type.

```
struct PropertyMap has copy, drop, store {
    map: SimpleMap<String, PropertyValue>,
}

struct PropertyValue has store, copy, drop {
    value: vector<u8>,
    type: String,
}
```

Aptos versus Sui

Aptos Move中资源所有权(resource ownership)：

- 在 aptos move中只有被关键词 key 修饰的 Struct, 才可以做为 resource
- 通过 move_ to 操作符将 resource 绑定到 signer 的 address 上 (存到全局 table (BTreeMap) , 存储层是 rocksdb)

在 sui move 中只有被关键词 key 修饰的 Struct 且其第一个字段是名为 id 的 UID 类型，这样的 Struct 可以做为 object

Sui 禁用了 move *to 操作符，通过* `sui:transfer` *模块将 object 绑定到指定的，address 或 object* id 上 (sui move中没有获取 object 的需要，所以没有全局 table 的概念，也是存到 rocksdb)

```
// Bridge Info
struct Info has key {
  id: UID,
  as_paused: bool,
  xbtc_cap: Option<ID>,
}
```

owned 和 shared 所有权安全

Aptos 的 owned 和 shared 所有权安全
Aptos： owned 所有权安全主要集中在 move 的语法层面:

1. move module 的 public, public(friend), public(script) 等修饰符
2. resource 的四种 abilities (copy, drop, store and key)
   1. 用 key 修饰 struct， 可以通过 move_to 放到链上、放到全局 Storage 里面，它有相应的地址，可以通过地址把 Resource 进行加载。
   2. store 和 key 是对应的，作为 key 修饰的 struct 字段也可以存储到链上。

3. drop 设计的很妙，如果 struct 没有 drop 能力的话，必须通过合约提供的函数进行销毁。如果提供了 drop 能力，是可以自行销毁的，如果考虑不周会产生安全事故：
    1. Liquidswap 闪电贷：
        1. 函数是输入 2 整数，返回 2 个币，第三个数给了一个闪电贷的结构 struct（没有 drop 能力），如果不能消耗后面的结构的话，是不能调用成功的。保证了合约的安全。
3. resource 的拆解和组装只能在其所在的 module 之中进行：
    1. 还是 Liquidswap 闪电贷，如果提供了 drop 的话，任何人都能把 coin 导走。

Aptos： shared 所有权安全主要控制在开发者手中

- 全员共享 —— 实现简单，在 aptos move 合约中，可以创建 resource account 绑定合约的 resource, 使得合约的存储空间间接共享
- 局部共享：最容易爆出合约安全事故，需要开发者自己实现 ACL (Access Control List, 访问控制列表)，达到只允许某些 address 访问的精确控制

Sui： owned 和 shared 所有权安全：

Sui：Owned 所有权出了体现在下列 move 的语法层面外:

1. move module 的 public (friend) 等修饰符 （ 去掉了 public(script)
2. object 的四种 abilities (copy, drop, store and key)
3. object 的拆解和组装只能在其所在的 module 之中进行

> 还体现在 sui 本身对 object 的抽象， 默认的 object 就是 owned:

shared 所有权安全:

- 全员共享: sui 提供了shared object (读写) 和 freeze object (只读) **shared object (读写) 和 freeze object (只读)**
    - 局部共享:最容易爆出合约安全事故，需要开发者自己实现 ACL (Access Control List, 访问控制列表)，达到只允许某些address 访问的精确控制

其他差异

aptos 和 sui move编译器会检测到以下大多数差异性(同样的代码)

1. 合约调用：aptos move 传递的是基础类型参数，sui 除了要传递基础类型参数外，还会传递 object 的 id
2. vm适配层：这导致了 aptos move 和 sui move 的系统库不一样
3. 部署合约：aptos 用户合约地址是可以指定到用户地址，而 Sui 则将所有用户合约地址统一到 0x0 地址下
4. 所有权转移：aptos resource 不能直接转移到 receiver地址，而 sui object 可以直接转移到 receiver 地址
    1. Aptos 这个设计对于 Web3 用户是比较突兀的，需要双方都同意
    2. Sui 则不需要接收方同意，如果你不想要就直接删掉 （ 呃，）
5. 地址格式: aptos 地址是 32 字节，而 sui 地址是 20 字节
6. 合约升级: aptos 合约支持升级，而 sui 合约暂时目前不支持升级

Move 中常用的编程模式：