

# Move语言基础(一)

Numen Cyber

Speaker: 王伟波



# Move语言简介

- Move是一种安全、沙盒式和经过形式化验证的编程语言，最初由facebook开发，它的第一个用例是Diem区块链(当时名字叫Libra)，Move为其实现提供了基础。Move允许开发人员编写灵活管理和转移数字资产的程序，同时提供安全保护，防止对那些链上资产的攻击。不仅如此，Move也可用于区块链世界之外的开发场景。
- Move的诞生从[Rust](#)中吸取了灵感，Move也是因为使用具有移动(move)语义的资源类型作为数字资产(例如货币)的显式表示而得名。



# Move语言特点

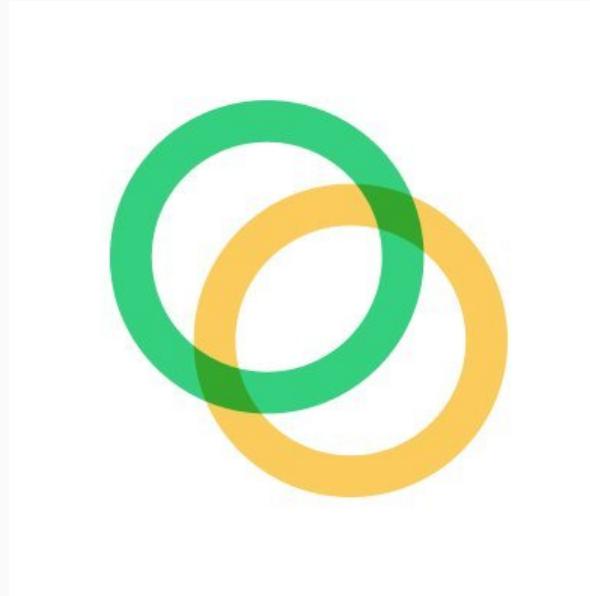
- 没有动态调用 (no re-entrancy)
- 不会混淆别名和可变性(类似RUST)
- 强制类型/内存/资源安全 (通过字节码校验)
- 鲁棒性
- 对整数溢出免疫
- Move Prover 形式化验证,确保合约的安全性



# 使用Move语言的L1网络



SUI



CEL  
O



APTOS



# 模块和脚本(Modules and Scripts)

- Move有两种不同类型的程序: Modules和Scripts。模块(Modules, 相当于智能合约)是定义结构类型以及对这些类型进行操作的函数的库。结构类型定义Move的全局存储的模式, 模块函数定义更新存储的规则。模块本身也存储在全局存储中。脚本(Scripts)是可执行的入口点, 类似于传统语言中的主函数main。脚本通常调用已发布模块的函数来更新全局存储。Scripts是临时的代码片段, 不发布到全局存储中。
- 一个Move源文件(或编译单元)。可能包含多个模块和脚本。但是发布模块或执行脚本都是独立的VM操作。



```
script {  
    <use>*  
    <constants>*  
    fun <identifier><[type parameters: constraint]*>([identifier: type]*) <function_body>  
}
```



```
module <address>::<identifier> {  
    (<use> | <friend> | <type> | <function> | <constant>)*  
}
```



# 示例

```
script {
    use std::debug;
    const ONE: u64 = 1;
    fun main(x: u64)
    {
        let sum = x + ONE;           debug::
        print(&sum)
    }
}
```

```
module 0x42::Test {
    struct Example has copy, drop { i: u64 }
    use std::debug;
    const ONE: u64 = 1;
    public fun print(x: u64)
    {
        let sum = x + ONE;
        let example = Example { i: sum };
        debug::print(&sum)
    }
}
```



# 安装move开发环境

- git clone <https://github.com/move-language/move.git>
- cd move ./scripts/dev\_setup.sh -ypt
- source ~/.profile
- cargo install --path language/tools/move-cli
- move -help



# 编写第一个Move 模块

```
language / documentation / tutorial / step_1 / basiccoin / sources / -- firstmove.move
1 module 0xCAFE::BasicCoin {
2     struct Coin has key {
3         value: u64,
4     }
5     public fun mint(account: signer, value: u64) {
6         move_to(&account, Coin { value })
7     }
8 }
9
```

Test.move

```
1 [package]
2 name = "BasicCoin"
3 version = "0.0.0"
4
```

Move.toml



# 给模块加单元测试

```
#[test(account = @0xC0FFEE)]
fun test_mint_10(account: signer) acquires Coin {
    let addr = signer::address_of(&account);
    mint(account, 10);
    // Make sure there is a `Coin` resource under `addr` with
    // a value of `10`.
    // We can access this resource and its value since we
    // are in the
    // same module that defined the `Coin` resource.
    assert!(borrow_global<Coin>(addr).value == 10, 0);
}
```



# 全局存储

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

```
/// Struct representing the balance of each address.  
struct Balance has key {  
    coin: Coin // same Coin from Step 1  
}
```



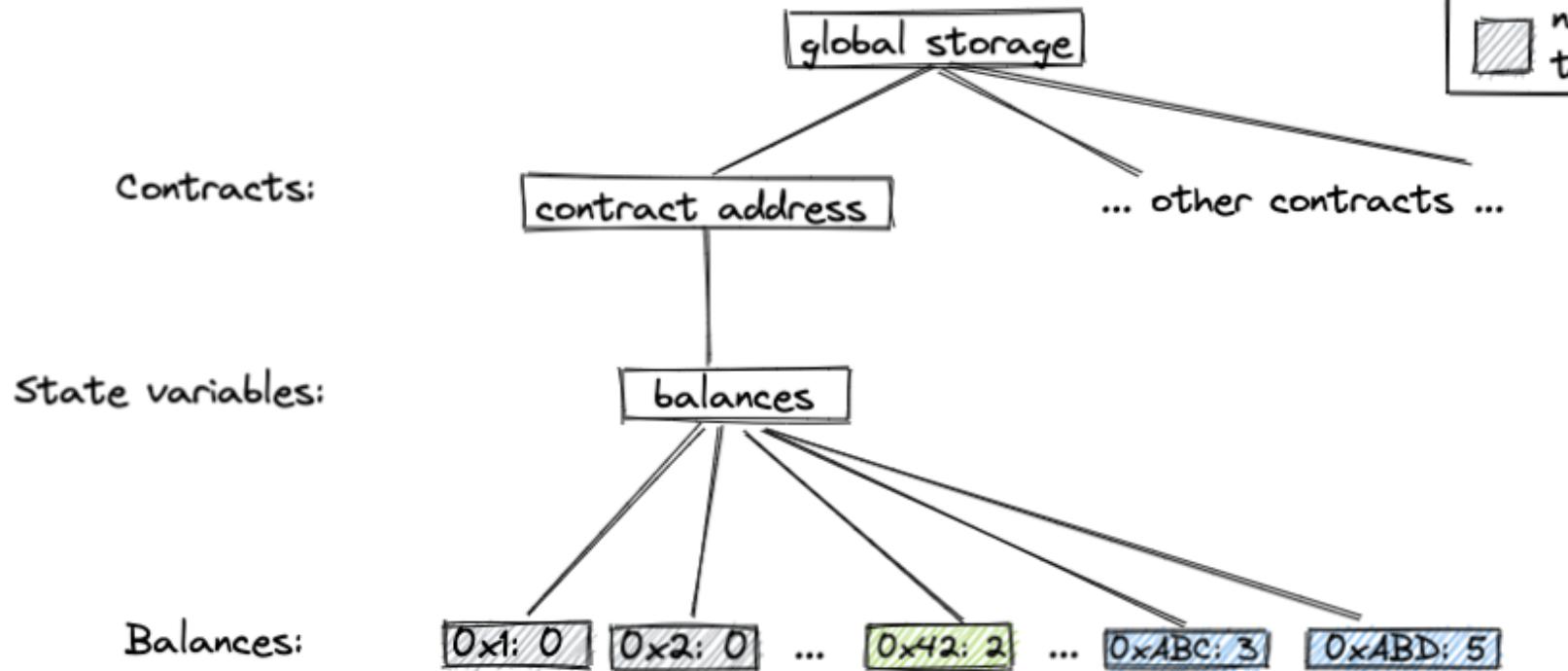
# 全局存储 - 操作 (Global Storage - Operators)

操作符	描述	出错
<code>move_to&lt;T&gt;(&amp;signer, T)</code>	在 <code>signer.address</code> 下发布 <code>T</code>	如果 <code>signer.address</code> 已经存在 <code>T</code>
<code>move_from&lt;T&gt;(address): T</code>	从 <code>address</code> 下删除 <code>T</code> 并返回	如果 <code>address</code> 下没有 <code>T</code>
<code>borrow_global_mut&lt;T&gt;(address): &amp;mut T</code>	返回 <code>address</code> 下 <code>T</code> 的可变引用 <code>mutable reference</code>	如果 <code>address</code> 下没有 <code>T</code>
<code>borrow_global&lt;T&gt;(address): &amp;T</code>	返回 <code>address</code> 下 <code>T</code> 的不可变引用 <code>immutable reference</code>	如果 <code>address</code> 下没有 <code>T</code>
<code>exists&lt;T&gt;(address): bool</code>	返回 <code>address</code> 下的 <code>T</code>	永远不会

每个指令的参数 `T` 都具有 [key 能力](#)。然而，类型 `T` 必须在当前模块中声明。这确保资源只能通过当前模块暴露的 API 来操作。指令在存储 `T` 类型资源的同时，使用 [address](#) 或 [&signer](#) 表示账户地址。



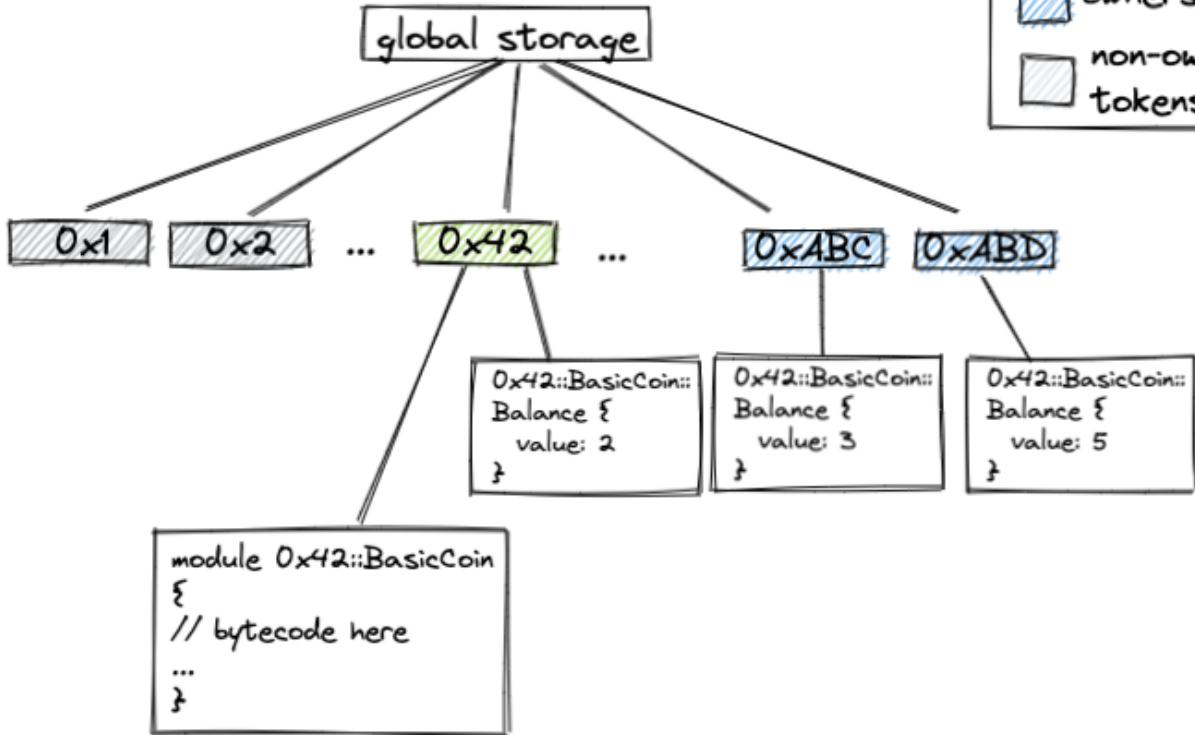
# Solidity Blockchain State





# Move Blockchain State

Addresses:



Resources:

Modules:



# 数据类型

- 内置类型
  - bool
  - unsigned integers: u8, u16, u32, u64, u128, u256
  - asset owner/tx sender: address
  - Strings: std::string::String (UTF8), std::ascii::String (ASCII)
- Collection types: vector<T>
- 用户自定义类型:

```
struct MyStruct {  
    int_field: u64,  
    struct_field: AnotherStruct  
}
```

```
struct AnotherStruct {  
    bool_field: bool  
}
```



# 向量 (Vector)

- `vector<T>` 是 Move 提供的唯一原始集合类型。`vector<T>` 是类型为 T 的同构集合，可以通过从"末端"推入/弹出（出栈/入栈）值来增长或缩小。（与 Rust 一样，向量（vector）是一种可以存放任何类型的可变大小的容器，也可称为动态数组）
- `vector<T>` 可以用任何类型 T 实例化。例如，`vector<u64>`、`vector<address>`、`vector<0x42::MyModuel::MyResource>` 和 `vector<vector<u8>>` 都是有效的向量类型。



# 向量 (Vector)

```
use std::vector;
let v = vector::empty<u64>();
vector::push_back(&mut v, 5);
vector::push_back(&mut v, 6);
assert!(*vector::borrow(&v, 0) == 5, 42);
assert!(*vector::borrow(&v, 1) == 6, 42); assert!(vector::pop_back(&mut v) ==
6, 42); assert!(vector::pop_back(&mut v) == 5, 42);
```



# 签名者 (Signer)

- 签名者 (signer) 是 Move 内置的资源类型。签名者 (signer) 是一种允许持有者代表特定地址 (address) 行使权力的能力 (capability)。可以将原生实现 (native implementation) 视为：

```
struct signer has drop { a: address }
```
- Move 程序可以使用地址字面量 (literal) 创建任何地址 (address) 值，
- signer 有点像 Unix UID，因为它表示一个通过 Move 之外的代码 (例如，通过检查加密签名或密码) 进行身份验证的用户。
- 但是，signer 值是特殊的，因为它们不能通过字面量或者指令创建 —— 只能通过 Move 虚拟机 (VM) 创建。在虚拟机运行带有 signer 类型参数的脚本之前，它会自动创建 signer 值并将它们传递给脚本：



```
script {
    use std::signer;
    fun main(s: signer) {
        assert!(signer::address_of(&s) == @0x42, 0);
    }
}
```

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



std::signer 标准库模块为 signer 提供了两个实用函数：

### 函数

signer::address\_of(&signer):  
address

signer::borrow\_address(&signer):  
&address

### 描述

返回由 &signer 包装的地址值。

返回由 &signer 包装的地址的引用。



# 引用(references)

- Move 支持两种类型的引用：不可变引用 & 和可变引用 &mut。不可变引用是只读的，不能修改相关值（或其任何字段）。可变引用通过写入该引用进行修改。Move的类型系统强制执行所有权规则，以避免引用错误。

语法	类型	描述
&e	&T 其中 e: T 和 T 是非引用类型	创建一个不可变的引用 e
&mut e	&mut T 其中 e: T 和 T 是非引用类型	创建一个可变的引用 e
&e.f	&T 其中 e.f: T	创建结构 e 的字段 f 的不可变引用
&mut e.f	&mut T 其中 e.f: T	创建结构 e 的字段 f 的可变引用
freeze(e)	&T 其中 e: &mut T	将可变引用 e 转换为不可变引用



# 引用(references)

```
let s = S { f: 10 };
let f_ref1: &u64 = &s.f; // works
let s_ref: &S = &s;
let f_ref2: &u64 = &s_ref.f // also works
```

```
let x = 7;
let y: &u64 = &x;
let z: &&u64 = &y; // will not compile
```

语法	类型	描述
<code>&amp;e</code>	其中 e 为 <code>&amp;T</code> 或 <code>&amp;mut T</code>	读取 e 所指向的值
<code>*e1 = e2</code>	其中 <code>e1: &amp;mut T</code> 和 <code>e2: T</code>	用 e2 更新 e1 中的值



# 无法存储引用 References Cannot Be Stored

- 引用和元组是唯一不能存储为结构的字段值的类型，这也意味着它们不能存在于全局存储中。
- 这是 Move 和 Rust 之间的另一个区别，后者允许将引用存储在结构内。
- Move引用无法被序列化，但 每个 Move 值都必须是可序列化的。

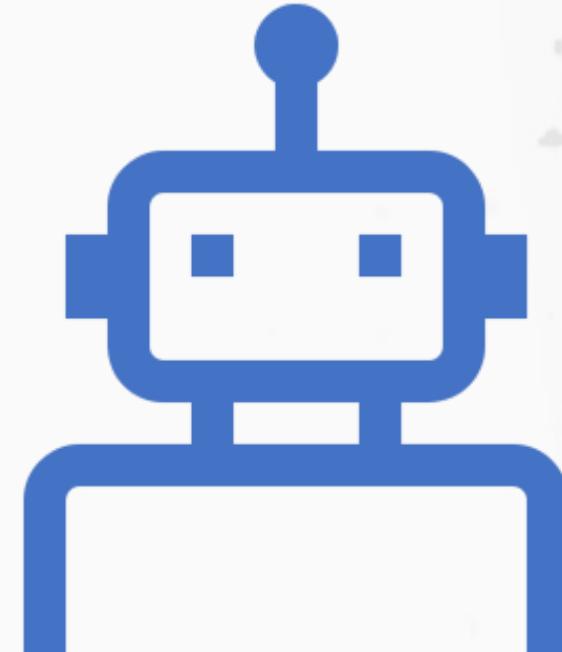
```
struct M{  
    id:u64,  
    lsok:bool,  
}
```

~~struct M{  
 id: &u64,  
 lsok:bool,  
}~~



# Move 函数可见性

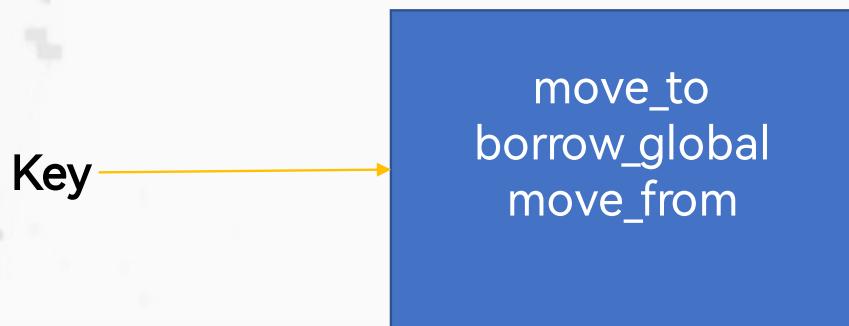
- // 可以可以从一个tx中被调用或者从另外一个模块被调用
- public entry fun i()
- 
- // 仅仅可以从一个tx中被调用，或者本模块
- entry fun j()
- 
- // 只能在本模块中被调用
- fun f()
- 
- // 可以被其他模块调用
- public fun g()
- // 可以被其他 在同一package“friend” 模块调用
- public(friend) fun h()





# 能力 (Abilities)

- copy :允许此类型的值被复制
- drop:允许此类型的值被弹出/丢弃
- store:允许此类型的值存在于全局存储的某个结构体中
- key:允许此类型作为全局存储中的键(具有 key 能力的类型才能保存到全局存储中)





# 能力 (Abilities)

- 要声明一个 struct 具有某个能力，它在结构体名称之后，在字段之前用 has <ability> 声明。例如：
  - struct Ignorable has drop { f: u64 }
  - struct Pair has copy, drop, store { x: u64, y: u64 }

```
// A struct without any abilities
struct NoAbilities {}

struct WantsCopy has copy {
    f: NoAbilities, // ERROR 'NoAbilities' does not have 'copy'
}
```



# 程序包(packages)

包允许 Move 程序员更轻松地重用代码并在项目之间共享。Move 包系统允许程序员轻松地：

- 定义一个包含 Move 代码的包
- 通过命名地址参数化包
- 在其他 Move 代码中导入和使用包并实例化命名地址
- 构建包并从包中生成相关的编译源代码
- 使用围绕已编译 Move 工件的通用接口

a\_move\_package

- Move.toml (required) (需要的)
- sources (required) (需要的)
- examples (optional, test & dev mode) (可选的, 测试 & 开发者模式)
- scripts (optional) (可选的)
- doc\_templates (optional) (可选的)
- tests (optional, test mode) (可选的, 测试模式)

模式)



# 泛型 (generics)

- 泛型可用于定义具有不同输入数据类型的函数和结构体。这种语言特性被称为参数多态。
- 函数和结构体都可以在其signatures, 中采用类型参数列表，由一对尖括号括起来 <...> 。

```
fun id<T>(x: T): T { // this type annotation is  
unnecessary but valid (x: T) }
```

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

```
struct Foo<T> has copy, drop { x: T }  
struct Bar<T1, T2> has copy, drop { x: T1, y:  
vector<T2>, }
```

```
fun foo() {  
    let foo = Foo<bool> { x: 0 }; // error! 0 is not a  
bool  
    let Foo<address> { x } = foo; // error! bool is  
incompatible with address
```



# 标准库

- Move本身有很多标准库
- APTOS, SUI自身有根据自己的需要添加了很多标准库
- 标注库的安全性需要考虑

## Index

- [0x1::ascii](#)
- [0x1::bcs](#)
- [0x1::bit\\_vector](#)
- [0x1::error](#)
- [0x1::fixed\\_point32](#)
- [0x1::hash](#)
- [0x1::option](#)
- [0x1::signer](#)
- [0x1::string](#)
- [0x1::type\\_name](#)
- [0x1::vector](#)



# MOVE CTF CHALLENGE



# 解题思路

1.本题主要是解出 data1 data2的数据

2.通过右边的代码片段，我们可以知道data2的长度为9

3. 我们通过22-24行代码，可以通  
过脚本计算出data2的值，  
 $a_{11}$ ,  
 $a_{12}, a_{13}, a_{21}, a_{22}, a_{23}, a_{31}, a_{32},$   
 $a_{33}$

4. 计算出 $a_{11}, a_{12}, a_{13}, a_{21}, a_{22}, a_{23}, a_{31}, a_{32}, a_{33}$ 之后，我们通过一次赋值给 $c_{11}, c_{21}, c_{31}$ 以及上面的那三个方程组，就可以计算出 $data_1$ 的值



总

对Move语言有了总体的认识

熟悉了Move语言的基本语法

对Move语言的特点有了新的体会



**Thank you**  
for your attention

Contact us at

---

📞 +65 6355 5555

✉️ Contact@numencyber.com

Find us at

---

