下一代智能合约编程语言Move

什么是Move

> Move是一种新的智能合约的开发语言，它和C，C++一样都是一种编程语言，但是其主要用于编写智能合约，那么什么又是智能合约呢，以下是维基百科的定义，由此可知智能合约是指根据协定能够自动执行、控制活记录相关事件和行为的计算机程序或交易协议。这里我们可以先简单把智能合约理解成一个程序，区块链和智能合约更多内容之后会专门讲解（又开一个新坑）。可以做一个简单的类比，区块链就是一个操作系统，而智能合约就会运行在这个操作系统上的应用程序，不同的操作系统编写应用程序的语言也不同，在aptos上move就是编写智能合约的语言。

环境配置

还是那句话，工欲善其事，必先利其器。首先需要安装move的命令行工具move-cli, 使用以下命令就可以安装，前提是已经安装过cargo，关于cargo的安装可以自行google。

```
cargo install --path move/language/tools/move-cli
```

或

```
cargo install --git https://github.com/diem/move move-cli --branch main
```

安装完成后还需要配置环境变量，一般情况下move-cli会安装在~/.cargo/bin目录下，所以需要把这个路径加入到环境变量中去，然后使用以下命令，如果正常执行则说明安装成功。

```
move
```

然后使用以下命令新建一个move项目,用VS Code打开就可以开始使用move开始编程了。

```
move package new <package_name>
```

move中一些特殊概念

与其他的智能合约编程语言如Solidity不同，Move试图将脚本和模块分离开，脚本可以让开发者在事务中添加更多的逻辑而且在存储时间和资源时更加灵活，而模块则是让开发者去扩展区块链的功能以及通过诸多选项去定制智能合约，首先我们会先了解如何去编写Move的脚本。

基本类型

与其他的编程语言一样，Move也内置了一些基本的数据类型，但是由于是专门针对智能合约的开发场景，所以内置的数据类型与一般熟知的编程语言并不相同，Move内置的数据类型有整型（u8, u64, u128）,布尔类型boolean以及地址类型address。

整型

Move中整型又u8，u64和u128三种，以下代码说明了如何去声明变量以及赋值。

```
script {
    fun main() {
        // 定义一个空变量然后赋值
        let a: u8;
        a = 10;

        //定义一个变量并赋值
        let a: u64 = 10;

        //简单赋值
        let a = 10u128;

        //在函数或者表达式中可以讲整型视为不可变的值
        if (a < 10u8) {};

        //在一般情况下不需要指明类型

        if (a < 10) {};

    }
}

```

在需要比较值时不同类型的值要进行转化，转换成同一类型的值时才能进行比较，在Move中使用as关键字实现类型转换，不同类型的值是无法比较的，例子如下：

```
script {
    fun main() {
        let a: u8 = 10;
        let b: u64 = 100;
```

```
5
6       //只有相同类型的数据才能比较
7       if (a == (b as u8)) abort 11;
8
9       if ((a as u64 == b)) abort 11;
10    }
11 }
12
```

布尔类型Boolean

在Move中Boolean的定义与其他语言中布尔类型的定义别无二致，都是只有两个值true和false。

```
1 script {
2     fun main() {
3       let b: bool;
4       b = true;
5
6       let b: bool = true;
7
8       let b = true;
9       let b = false;
10    }
11 }
12
```

地址类型Address

地址类型算是编写智能合约语言相较于其他语言特有的类型，它代表了一个区块链的地址，用于发送代币和引入模块，在不同的区块链对于地址类型的定义也不同如在Diem中地址是16字节的16进制数，在dfinance的DVM中地址有"wallet1"的前缀。

```
1 script {
2     fun main() {
3       let addr: address;
4
5     //可以通过{{}}获取地址
6       addr = {{sender}};
7     // Diem中是16字节的16进制数
8       addr = 0x....;
9
10    // 在dfinance DVM中地址又“wallet1”的开头
11
12      addr = wallet1...;
```

```
13
14     }
15 }
16
```

注释

为了使程序更加可读，注释是必不可少的，与大部分语言相同，Move的注释也有两种，单行注释和多行注释。

单行注释

```
1 script {
2     fun main() {
3         // 单行注释
4     }
5 }
6
```

多行注释

```
1 script {
2     fun main() {
3
4         /*多行注释多行注释多行注释多行注释多行注释多行注释
5         多行注释多行注释多行注释多行注释*/
6
7     }
8 }
9
```

表达式与作用域

> 在编程语言中，表达式是指拥有返回值的代码单元，例如一个有返回值的函数调用就是表达式。
> 表达式必须要用分号分隔

空表达式

在move中一个空括号代表一个空表达式，空表达式会被虚拟机自动忽略，以下就是一个空表达式的例子：

```
script {
    fun empty() {
        () //这是一个空表达式
    }
}

```

文字表达式

看下面的代码，每一行以分号结尾都是一个表达式。

```
script {
    fun main() {
        10;
        10 + 5;
        true;
        true != false;
        0x1;
        1; 2; 3
    }
}
```

这些都是最简单的表达式，那么为什么我们需要它们？我们又该如何利用它们呢？不知道还记不记得上一篇文章我们介绍基本数据类型时定义基本类型的变量，这就是对于表达式的运用，通过let关键字声明变量，然后将表达式的值存储在变量中：

```

script {
    fun main() {
        let a;
        let b = true;
        let c = 10;
        let d = 0x12;
        a = c
    }
}

```

整型的运算符

Move中提供了很多整型的运算符可以改变整型的值

运算符	含义
+	相加
-	相减
/	除
*	乘
%	除余
<<	左移
>>	右移
&	与
^	异或
|	或

"_"与标记未使用

在Move语言中每一个变量都必须被使用，否则代码无法编译通过，以下代码编译会报错

```
script {
    fun main() {
        let a = 1;
    }
}

```

其报错如下

```
warning[W09003]: unused assignment
  ┌─ ./sources/test.move:3:12
  │
3 │         let a = 1;
  │             ^ Unused assignment or binding for local 'a'. Consider
removing, replacing with '_', or prefixing with '_' (e.g., '_a')
  
```

报错提示也说了可以通下划线"_"代替a，以下代码就不会报错

```
script {
    fun main() {
```

```
        let _ = 1;
    }
}

```

在Move中允许定义一个变量两次，但是必须使用"_"将第一次定义的值标记为不使用，其代码如下:

```
script {
    fun main() {
        let a = 1;
        let a = 2;
        let _ = a;
    }
}

```

块表达式

一个由{}包围组成的代码块也是一个表达式，一个代码块可以包含表达式，也可以包含其他代码块，从某种意义上说函数体也是带有某种限制的代码块。

```
script {
    fun block() {
        { };
        { { }; };
        true;
        {
            true;

            { 10; };
        };
        { { { 10; }; }; };
    }
}

```

理解作用域

在Move中作用域就是由{}所包围的代码块,代码块可以相互嵌套且没有限制。

```
script {
    fun main() {
```

```
        //这是一个函数体
        {
            //这是函数体内的一个代码块
            {
                //这是代码块中的一个代码块
            };
        };
        {
            //函数体内的另一个代码块
        };
    }
}

```

变量的生命周期

对于变量而言，代码块内部可以访问外部的变量，但是外部却无法访问代码块内部的变量

```
script {
    fun main() {
        let a = 2;
        {
            let b = a + 2;
        };
        let c = b + 3;
    }
}

```

代码块的返回值

代码块也是一个表达式，一个表达式就有返回值，其返回值就是最后一个没有分号的表达式的值

```
script {
    fun main() {
        let a = {
            let c = 10;
            c * 100
        };
        let _ = a;
    }
}

```

控制流

Move与其他语言相同，也有控制流语句，主要是循环的while和loop，以及if。

if

Move语言的if与其他语言的使用并无什么特别大的差异。

```
script {
    fun main() {
      let a = true;
      let b = if (a) {
        10
      } else {
          20
          };
}

```

while

Move 语言中While的使用也与其他语言相同

```
script {
    fun main() {
      let i = 0;

      while(i < 5) {
       i = i + 1;
      };
}

```

无条件循环loop与break，continue

与其他语言不同，Move提供了一个无条件的循环loop，没有终止条件，不加干预会一直循环下去

```
script {
    fun main() {
        let i = 0;

        loop {
            i = i + 1;
        };
```

```
8
9           // UNREACHABLE CODE
10          let _ = i;
11      }
12  }
13
```

为了能够终止循环，Move提供来break和continue，其用法与其他语言相同

```
1  script {
2      fun main() {
3          let i = 0;
4
5          loop {
6              i = i + 1;
7
8              if (i / 2 == 0) continue;
9              if (i == 5) break;
10
11             // assume we do something here
12         };
13
14         0x1::Debug::print<u8>(&i);
15     }
16 }
17
```

abort

有时候一些条件触发，需要废弃函数执行，这也是智能合约常用的场景，Move提供了abort关键字。

```
1  script {
2      fun main(a: u8) {
3
4          if (a != 10) {
5              abort 0;
6          }
7          //如果a不等于10程序不会执行到这里
8      }
9  }
10
```

assert

assert是对abort的进一步包装

```
script {

    fun main(a: u8) {
        assert!(a == 10, 0);

        //如果a等于10程序会执行到这里
    }
}

```

模块

模块是开发者发布在自己地址的一系列函数，之前我们使用的脚本只能使用已经发布的模块或者标准库，标准库也就是发布在0x1地址的一系列模块。

模块是发布在发送者的地址的，标准库是发布在0x1地址的，当发布一个模块时它的函数并未执行，需要使用use script去执行模块。

模块是以module关键字开头，紧接着的是模块明，之后双括号里面就是模块的内容：

```
address 0x1 {
  module Math {
  //模块内容
   public fun sum(a: u64, b: u64): u64 {
     a + b
   }
}
}

```

模块是唯一的向其他人开放代码的方式，新的类型和资源只能定义在模块里。

默认的你的模块会在你的地址编译和发布，然而如果需要使用本地的模块（用作测试和开发）可以在模块文件中指定地址即可。

```
address 0x1 {
```

```
  module Math {
  //模块内容
  public fun sum(a: u64, b: u64): u64 {
    a + b
  }
}
}

```

引入模块

Move中默认的上下文是空的，这意味着你只能使用基本类型如interger，bool和address，而且智能操作这些基本类型和变量，为了实现更加复杂的功能，你可以引入已经发布的模块或者标准库。

直接引入

可以直接通过地址引入模块

```
script {
    fun main(a: u8) {
        0x1::Offer::assert!(a == 10, 1);
    }
}

```

在这个例子中我们使用了地址01x(标准库)的模块Offer中的方法assert!(expr: bool, code: u8)。

使用关键字use

为了使代码更加简洁，可以使用use关键字引入模块。

```
use 0x1::Vector;

```

引入模块后需要使用模块的内容需要使用::

```
script {
  use 0x1::Vector;
    fun main(a: u8) {
        let _ = Vector::empty<u64>();
    }
}
```

```
7
```

与此同时还可以引入指定的内容

```
script {
  //引入一个成员
  use 0x1::Signer::address_of;
  //引入多个成员
  use 01x::Vector::{
    empty,
    push_back
  }
}

```

为了避免命名冲突，可以在引入是使用as取别名

```
script {
  //引入一个成员
  use 0x1::Signer::address_of as addr;
  //引入多个成员
  use 01x::Vector::{
    empty as em,
    push_back as pu
  }
}

```

常量

Move支持定义脚本级别和模块级别的常量，一旦定义就无法更改。

```
script {

    use 0x1::Debug;

    const RECEIVER : address = 0x999;

    fun main(account: &signer) {
        Debug::print<address>(&RECEIVER);

```

```
        // they can also be assigned to a variable

        let _ = RECEIVER;

        // but this code leads to compile error
        // RECEIVER = 0x800;
    }
}

```

函数

在Move中函数是唯一执行的地方，函数需要使用fun关键字声明，之后是函数名和参数列表，但括号内的是函数体。

```
fun function_name(arg1: u64, arg2: bool) {
    //函数体
}

```

> 在Move中函数名需要都小写然后通过下划线连接。

脚本中的函数

脚本中只包含一个main函数，这个函数带有参数是用来执行一次事务，限制比较多，没有返回值，可以用来执行已经发布的模块的函数，下面的脚本就是用来检查地址是否存在。

```
script {
  use 0x1::Account;
  fun main(addr: address) {
    assert!(Account::exists(addr));
  }
}

```

模块中的函数

模块是一系列解决一个或多个任务的函数和类型（之后会详细介绍）的集合。在这一部分我们会创建一个简单的Math模块，提供一系列基本的算术能力。

```
module Math {
  fun zero(): u8 {
```

```
    0
  }
}

```

> 需要注意的是函数可以用return来终止函数执行并且返回一个值，以上就是返回来一个零，这是我们在表达式中提到到，没有分号可以省略return关键字。

返回多个返回值

在Move中函数是可以返回多个返回值的：

```
module Math {
  public fun max(a: u8, b: u8): (u8, bool) {
    if (a > b) {
      (a, false)
    } else if {
      (b, false)
    } else {
      (a, true)
    }
  }
}

```

我们已经返回了两个值，那么又该如何接受那两个值呢:

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

  fun main(a: u8, b: u8) {
    let (max, is_equal) = Math::max(99, 100);
    assert(is_equal, 1);
    Debug::print<u8>(&max)
  }
}

```

函数的可见性

Move的函数主要可以分为两类，一类事private（默认的），一类事public，如果时public则是其他模块也可以访问，但是如果时private，则只能在定义函数的地方访问。例子如下，is_zero和zero函数都

是在Math模块内，但是zero是private，is_zero可以访问zero，但是其他模块不可以访问zero，而is_zero是public其他模块也可以访问。

```
module Math {

    public fun is_zero(a: u8): bool {
        a == zero()
    }

    fun zero(): u8 {
        0
    }
}

```

native函数

Move有一种特殊类型的函数native函数，native函数是实现Move没有提供的能力，实现的方式也有很多种，其室友虚拟机自己定义的，例子如下

```
module Signer {

    native public fun borrow_address(s: &signer): &address;

    // ... some other functions ...
}

```

结构体的定义

结构体只能在模块里定义，一struct关键字开头，之后时名字和双括号，双括号里面是是定义：

```
struct Name {
  FIELD1: TYPE1,
  FIELD@: TYPE2,
  ......
}

```

以下就是结构体定义的例子：

```
module M {
  struct Empty{}

  struct MyStruct{
    field1: address,
    field2: bool,
    field3: Empty
  }

  struct Example {
    field1: u8,
    field2: address,
    field3: u64,
    field4: bool,
    field5: bool,

    field5: MyStruct
  }
}

```

需要注意的是每一个定义的结构体都会变成一个新的类型，这个类型可以桶模块访问，如M::MyStruct。

递归定义在Move是不允许的，例如以下代码编译会报错：

```
struct MyStruct{
    field5: MyStruct
  }

```

结构体的使用

我们可以通过创建实例来使用结构体，可以通过它的定义来创建实例：

```
module Country {
  struct Country {
    id: u8,
    population: u64
  }

  public fun new_country(c_id: u8, c_population: u64): Country {
    let country = Country {
      id: c_id,
```

```
        population: c_population
    };
    country
  }
}

```

Move还允许使用更简洁的方式创建实例：

```
// ...
public fun new_country(id: u8, population: u64): Country {
    // id matches id: u8 field
    // population matches population field
    Country {
        id,
        population
    }

    // or even in one line: Country { id, population }
}

```

访问结构体的属性

> 只有模块内部才能访问结构体的属性，对于模块外部，结构体的属性是不可见的：

```
// ...
public fun get_country_population(country: Country): u64 {
    country.population // <struct>.<property>
}


```

结构结构体

可以通过let = 去解构一个结构体

```
module Country {

    // ...

    // we'll return values of this struct outside
```

```
    public fun destroy(country: Country): (u8, u64) {

        // variables must match struct fields
        // all struct fields must be specified
        let Country { id, population } = country;

        // after destruction country is dropped
        // but its fields are now variables and
        // can be used
        (id, population)
    }
}

```

> 需要注意的是未使用的变量在Move中是被禁止的，如果解构结构体又不需要使用其属性，需要使用下划线：

```
module Country {
    // ...

    public fun destroy(country: Country) {

        // this way you destroy struct and don't create unused variables
        let Country { id: _, population: _ } = country;

        // or take only id and don't init `population` variable
        // let Country { id, population: _ } = country;
    }
}

```

之前提到外部的模块是无法直接访问结构体的属性的，所以为了外部模块能够访问定义的结构体的属性，需要提供相关的访问函数：

```
module Country {

    struct Country {
        id: u8,
        population: u64
    }

    public fun new_country(id: u8, population: u64): Country {
        Country {
```

```
            id, population
        }
    }

    // don't forget to make these methods public!
    public fun id(country: &Country): u8 {
        country.id
    }

    // don't mind ampersand here for now. you'll learn why it's
    // put here in references chapter
    public fun population(country: &Country): u64 {
        country.population
    }

    // ... fun destroy ...
}

```

Abilities

Move有一个非常灵活且可高度定制化的类型系统，每一个类型都可以被赋予四种能力去决定数据怎么被使用、删除活存储，这四种能力如下：

- Copy-可以被复制
- Drop-可以在结尾被删除
- Key-可以被全局存储设置为key
- Store-可以被全局存储
- 这篇文章会介绍Copy和Drop的能力，至于Key和Store能力在之后的文章详细介绍。在定义结构体的时候可以设置这四种能力：

```
module Library {
    struct Book has store, copy, drop {
      year: u64
    }

    struct Storage has {
      books: vector<Book>
    }

    struct Empty{}
    }

```

```
    // 如果没有设置这几种属性会怎么样呢:
    module Country {
        struct Country { id: u8, population: u64 }
    }

    public fun new_country(id: u8, population: u64) {
        Country(id, population) }
    }

    script { use {{sender}}::Country;
    fun main() { Country::new_country(1, 1000000); } }

    // 我嘛在Country中定义了一个结构体，然后通过一个方法暴露给外部，然后在外部调用这个方法来获取数据，但是返回值没有被使用，由于我们没有给这个结构体设置Drop属性，这时候不会自动丢弃返回值，会报错

    error: ┌── scripts/main.move:5:9 ─── │ 5 │     Country::new_country(1, 1000000); │     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ Cannot ignore values without the 'drop' ability. The value must be used │

    #### Drop
     //我们可以通过has Drop给结构体设置Drop能力

    module Country { struct Country has drop { // has id: u8, population: u64 } // ... }
    rust

    //当我们设置了Drop能力，脚本就能运行了:
    script { use {{sender}}::Country;

    fun main() {
        Country::new_country(1, 1000000); // value is dropped
    }
    > Drop只定义Drop的能力，解构不需要Drop能力。

    #### Copy
    // 我们已经学习了如何创建一个Country实例以及丢弃它，那么我们该如何实现拷贝呢，Move提供来copy关键字来实现拷贝，前提是需要已经赋予结构体copy能力。

    module Country { struct Country has drop, copy { // see comma here! id: u8, population: u64 } // ... }

    script { use {{sender}}::Country;

    fun main() {
        let country = Country::new_country(1, 1000000);
        let _ = copy country;
```

```
53      }
54  }
```