glua编程语言

Contents:

简单介绍

glua 是一种区块链的智能合约编程语言，是一种静态强类型编程语言。

入门Tutorial

1. 开发环境

glua可以用来编写智能合约放入区块链然后调用，也可以作为区块链的event回调脚本执行.

在开发链上合约时，可以在区块链测试链上开发，编译glua合约源码文件生成字节码文件，然后把字节码注册到cchain测试链，获取到一个合约地址，然后调用这个合约的某个API；

在开发本地脚本时，可以在测试链上先编译glua脚本源码文件生成字节码文件，然后注册字节码文件为某个合约的某个event的回调脚本，当区块链同步到关联的event时就触发执行回调脚本。

当要发布到正式链上时步骤和上面类似，测试链主要用来开发时使用。

具体编译，注册合约，调用合约，注册脚本等的说明另见相关说明文档。

你需要什么：

- 最新版本的底层钱包节点程序（内置glua编译器和解释器）
- 一款你喜欢的编辑器

你还需要什么：

- 需要连上区块链网络，并同步完成所有的数据块（正式链）
- 一个拥有足够代币的账户，并保持钱包打开并处于解锁状态（正式链）

2. 你的第一个glua程序

```
type Person = {
    id: string default "123",
    name: string default "glua",
    age: int default 24
}

var M = Contract<Person>()

function M:init()
    let p = Person()
    pprint(p.id, p.name, p.age)
end

return M
```

3. 选择一个编辑器

我们提供了一个glua的IDE供开发使用，并提供和区块链的集成可以用来开发智能合约。但是也可以用各人自己喜欢的编辑器比如Visual Studio Code, Vim等编写，没有特别要求。 但是还是更推荐使用我们提供的IDE工具，因为是为合约的开发定制化的开发工具，集成了很多的功能，可以为你省去了很多命令行操作的繁琐动作。

4. 开始用glua编写智能合约

基本语法

Contents:

类型

这一章描述glua语言的类型系统

基本类型

- 包括nil, string, boolean, number, integer, function, table, Map, Array, Stream 几种基本类型
- nil: 表示空数据
- string: 字符串，代码里字面量用双引号或者单引号或者[[和]]包围起来，比如”hello world”，’hello’，[[hello world]]。[[和]]包括起来的是跨行字符串
- boolean: 布尔类型，表示真或者假，true或者false
- integer: 整数类型，比如123,45这样的，支持64bit整数，整数范围 `-9223372036854775808` 到 `9223372036854775807`
- number: 浮点数类型，表示小数，实现是64bit的double类型，值的范围是 `-2^1024` 到 `2^1024` ，精度15位小数. 同时编译期integer类型可以隐式转换成number类型，number类型不能隐式转换成integer类型
- function: 函数类型，函数可以作为变量的值，可以作为参数和返回值，支持闭包，使用起来类似其他类型的值，但是可以调用
- Map<T>: 哈希表类型，T代表哈希表的值类型，是一个键值对，键的类型是string类型，每一个键值对的值类型都是T代表的类型, Map<T>类型是table类型的子类型

Map<T>类型可以通过table模块中的函数进行操作，可以通过中括号下标或者点号加属性名进行读写访问，比如a['name'], a.name等

- Array<T>: 列表类型，T代表列表的值类型，列表中每一项都是T代表的类型，Array<T>类型是table类型的子类型

  Array<T>类型可以通过table模块中的函数进行操作，可以通过中括号下标进行读写访问，比如a[1]等
- Stream 二进制字节流类型，表示一个二进制比特的流

比如

```
let a = {}                    -- 这是一个空哈希表，a变量的类型自动推导为Map<object>
a['name'] = 'glua'           -- 修改a指向的哈希表中的键值对
let b = { age: 2 }              -- b变量类型是Map<int>
let c={ ['name']='China', age=5000, address: 'China' }            -- c变量类型是Map<object>类型，因为值类型有多种不同类型，所以自动推导为Map<object>
let d = [ 1, 2, 3 ]             -- d变量是一个包含3个值的列表，类型是Array<int>
```

- record: 编译期的自定义数据结构，类似C语言的struct结构体，里面可以自定义属性，程序编译完成，在运行时record类型的变量表现相当于table类型

比如

```
type Person = {
    name: string,
    age: int default 24,
    age2: int = 24  -- record属性默认值既可以用default也可以用=区分
}
```

- table: Map<T>和Array<T>以及record类型的父类型，可以将其他三种类型隐式转换或者通过totable函数显式转换成table类型，也可以将table类型隐式转换成具体子类型，Map类型可以转换成record类型， 其他不同子类型之间不能直接互相转换
- union: 表示在编译期可能是几种不同类型中的某一种类型，比如 let a : int | string | number 表示是int或string或number类型
- literal type: Literal Type类型是一种枚举类型，可以将多个字符串或者数字或者布尔值或者nil字面量作为枚举类型的每一项，

给枚举类型赋值时必须用兼容的枚举类型或者枚举类型中的字面量值，否则编译期报错

比如

```
type Gender = "male" | "female"
var gender: Gender
gender = "male"    -- Correct
gender = "Chinese" -- Error: 编译错误
```

- object: 编译期类型，是所有类型的父类型，变量被声明为object可以被赋值给任何类型的值，并且访问object类型的变量的属性是在编译期允许的。

比如

```
var a: object = 1
a = "hello"
pprint(a.name)  -- 编译期通过，运行时失败
```

静态类型系统

- 编译期会进行变量类型，函数调用，操作符使用，函数声明，record结构等进行静态类型推导和分析，对于类型不正确的使用编译期会报错。
- 对于变量，会分析变量在不同代码位置的类型，进而分析在不同位置是否使用错误，比如类型是否使用正确，或者是否初始化后才使用
- 对于函数定义，会分析参数类型和返回类型，以及在函数体中进行静态类型分析
- 对于函数调用，会分析当前上下文中，是否函数类型和使用的实参类型匹配
- 对于record类型，会分析每个属性字段的类型
- 变量和函数参数的类型声明，以及构造函数等用到类型的地方，会检查类型的存在下，兼容性等

全局变量和局部变量

- 不允许创建新的全局变量，也不允许对_ENV, _G全局变量做修改，只允许新创建local变量，语法是类似local a = 123; local a,b = “name”, ‘age’这样。如果创建局部函数，可以类似local function abc() return “abc”; end
- 局部变量声明可以同时声明变量类型，比如local a: string = “hello”
- 局部变量的值在编译期需要和局部变量的编译期类型声明一致，比如local a = 1; a = “hi” 这样会编译期报错.
- 为了安全因素，限制智能合约中的每个函数不能有超过128个局部变量，并且限制每个局部变量的符号长度不能超过128个字符(函数的参数，包括self，也算入局部变量的数量中)
- 变量名称、参数名、event名称的符号不允许使用关键字的名字，尤其注意 do/end/then等容易被误用为变量名/参数名/event名称的关键字
- 局部变量可以用local, var, let三个关键字来声明
- var关键字和local 关键字等效，都是用来声明可变局部变量，可变局部变量在接下来的可见作用域代码中可以被使用和修改
- let关键字用来声明不可变局部变量，声明的局部变量只能被初始化值，不能修改值，但是不可变的只是变量本身，变量指向的值如果是table或record类型依然可以改变这个table的内容

比如:

```
var a = 'hello'
local b = 'hello'
let c = 'hello'
var d: object = 'hello'
a = 'glua'  -- 正确
a = 123  -- 错误，a类型不能改变
b = 'glua'  -- 正确
b = 123 -- 错误，b类型不能改变
c = 'glua' -- 错误
d = 123  -- 正确，d变量编译期类型的object类型，可以用数字类型的值赋值
```

类型声明

声明变量和声明函数的参数时可以同时声明变量类型，函数参数如果不显示声明类型则默认是object类型

例如：

```
let a: string = "123"
var b: G1                           -- G1是某个record类型
let b: int | string | Person     --   Person是某个record类型，这种声明表示b是int或string或Person类型，也就是union类型
let c ?: int                         --  这种表示声明的变量c是int或nil类型的，相当于 int | nil的union类型

let function add(a: number, b: number, c ?: Array<number>)
    return a + b
end
```

声明类型时也可以声明函数类型，函数的签名类型语法是 (ArgTypeName1, ...) => RetTypeName

例如

```
let a: (int, int, int) => string
```

还可以将变量或者函数参数的类型声明为Function，表示这个变量/参数的类型是函数，能接受任何函数类型的值，不论参数数量，参数类型和返回值类型。 例如

```
let function add(a: number, b: number)
    return a + b
end
let add2: Function = add
let r = add2('123')    -- 这段代码编译期类型没问题，但是运行时会报类型错误
```

record自定义类型

- record类型是类似C语言的struct的语法，可以自定义带有若干个属性字段的结构体，每个属性字段都有各自的类型，不同属性可以有不同类型， 属性的类型也可以是其他的record类型，也就是可以嵌套使用record类型。

  比如:

```
type Address = {
    province: string,
    city: string
}

type Person = {
    name: string,
    age: int default 24,
    age2: int = 24,  -- record属性默认值既可以用default也可以用=区分
    address: Address,
    parent_names: Array<string>
}
```

- record的初始化和赋值可以用table来赋值，record类型的变量或者函数返回值也可以赋值给table类型的变量，不需要额外做类型转换， record的属性和属性值与table的键值一一对应。

比如:

```
type Person = {
    name: string,
    age: int default 24
}

type Address = {
    province: string,
    city: string
}

var p1: Person = {name='glua'} -- 用Map<T>值给record类型的变量赋值，会做自动类型转换
var p2: Address = p1           -- 编译报错，p1是Person类型变量，不能赋值给Address类型的变量
var p3: table = p1             -- 正确，record类型的变量或值可以赋值给table类型的变量
```

- record类型只是对变量的类型声明，但是在运行时实际还是table类型。 record如果用构造函数（类型构造函数的名称和类型名称同名）创建，可以使用一个table值作为参数初始化，没有赋值的属性会使用属性默认值。如果用table类型的值比如{}进行类型转换赋值给record类型，则不涉及构造函数调用，record类型的属性默认值不起作用。

  合约的storage的初始化是直接由区块链初始化的，record类型的default属性值不对它起作用

比如:

```
type Person = {
    name: string,
    age: int default 24,
    age2: int = 24  -- record属性默认值既可以用default也可以用=区分
}
let p1 = Person()                          -- p1值是{name: nil, age: 24, age2: 24}
let p2 = Person({name='glua'})             -- p2值是{name: 'glua', age: 24, age2: 24}
let p3 = Person({name='glua', age=100})    -- p3值是{name: 'glua', age: 100, age2: 24}
let p4: Person = {}                        -- p4值是{}
```

- record定义时可以带有若干个泛型参数，泛型参数在record的属性的类型中可以用来定义属性类型。带有泛型参数的record类型， 需要用具体类型实例化后才能用来声明变量或者函数参数类型。

比如：

```
type Person<A> = {  -- A就是Person泛型的泛型参数，是用来代表未知类型的类型变量
    name: string,
    address: A      -- 将address属性声明为A类型，则表示address属性的类型根据Person泛型实例化时的A的具体类型来确定
}
let p = Person<string>()  -- 这里先用string类型替换Person泛型的A类型变量替换产生一个新类型Person<string>，然后调用这个新类型的构造函数
```

- record的成员函数的定义，不能用function <varname>.<funcname> (...) ... end的语法,只能用function <varname>:<funcname> (...) ... end的语法

比如:

```
type Person = {
    name: string,
    age: int
}
var p1 = {}
function p1.sayHi1(self)    -- 正确
end
function p1:sayHi2()    -- 正确
end

var p2 = Person()
function p2.sayHi1(self)    -- 编译错误，p2是record类型的，不能用varname.funcName的方式定义成员函数
end
function p2:sayHi2()        -- 正确
end

p1:sayHi1()                 -- 正确
p1:sayHi2()                 -- 正确
p2:sayHi2()                 -- 正确
```

- 没有特定说明的record的语法，和Map<object>语法一致,也可以用varname.propertyName和varname['propertyName']，varname["propertyName"]的方式来读取和修改record类型变量的属性
- record定义的语法是:

```
type RecordName = { PropName: PropTypeName, … }
```

或

```
type RecordName <GenericType1, … > = { PropName: PropTypeName, … }
```

例如:

```
type Person = {
    name: string,
    age: int,
    mobile ?: string default ''                  --  这表示属性mobile的类型是string | nil，并且默认值是空字符串
}
type G1<T1, T2, T3> = { a: T1, b: T2, c: T3, count: int }
```

- record可以用来定义其他名称的新record，同时可以有新的泛型参数（可选的）

语法如下：

```
type RecordName = RecordNameExisted < Type1, … >
```

或

```
type RecordName = RecordNameExisted
```

或

```
type RecordName <GenericType1, … > = RecordNameExisted < Type1, … >
```

或

```
type RecordName <GenericType1, … > = RecordNameExisted
```

例如:

```
type G2<T> = G1<int, T, string>
type G3 = G1<string> -- 编译报错，G1需要3个类型变量
type G4 = string
type G5 = G4
```

- record类型在定义后自动产生一个同名函数作为构造函数，可以可选地接受一个Map<T>类型的参数， 参数Map<T>中有的属性覆盖record类型的属性默认值，合并后的Map<T>当成record类型作为构造函数的返回值

  调用类型的构造函数也可以省略括号直接传一个map字面量字面量作为参数

例如:

```
type Person = { name: string, age: int default 100 }
let p1 = Person()                          -- p1 的值是 {name: nil, age: 100}
let p2 = Person({name: "glua"})            -- p2 的值是{name: "glua", age: 100}

let p3 = Array<Person> (  [ {name: "glua"}, {name: "China", age: 10000} ] )   -- 这里括号不能省略,因为参数是array字面量不是map字面量
let p4 = Array<Person> { name: "hello", age: 100 }
```

Literal Type类型

- Literal Type类型是一种枚举类型，可以将多个字符串或者数字或者布尔值或者nil字面量作为枚举类型的每一项，

给枚举类型赋值时必须用兼容的枚举类型或者枚举类型中的字面量值，否则编译期报错。

- 运行期枚举类型表现出的类型和运行时具体的值的类型一致，也就是说，枚举类型是编译期类型，运行期不存在枚举类型
- 枚举类型没有构造函数，这点和record类型不一样
- 语法

```
type EnumTypeName = LiteralValue1 | LiteralValue2 ...

LiteralValue1 可以使用 LiteralString | LiteralNumber | LiteralInteger | LiteralBool | nil 其中任何一种字面量
```

- 例如

```
type Gender = "male" | "female"
var gender: Gender
gender = "male" -- Correct
gender = "Chinese" -- Error: 编译错误

function a1(p: true)
end

function a2(p: 'Chinese')
end

let a3: string = gender -- 枚举类型的变量可以显式降级到枚举值的类型

-- Literal Type类型也可以拼接
type Cat = "whiteCat" | "blackCat"
type Dog = "husky" | "corgi"
type Pets = Cat | Dog
-- 等效于 type Pets = "whiteCat" | "blackCat" | "husky" | "corgi"

type Falsy = "" | 0 | false | nil
```

泛型

- 支持在record类型定义时使用泛型来定义，在属性的类型中当做类型来使用。
- 带有泛型的record类型需要用实际类型实例化后才可以用来声明变量类型
- 定义新record类型时支持泛型，支持多个泛型参数

语法

```
type RecordName <GenericName1, … > = { PropName: TypeName, … }
```

例如：

```
type G1<T1, T2, T3> = {    -- 这里T1，T2，T3是泛型的类型变量，这里定义了个泛型类型G1，具体使用时可以用具体类型分别替代T1，T2，T3，产生一个新类型
    id: string,
    a: T1,
    b: T2,
    c: T3
}
```

- 泛型类型部分实例化和类型重定义
- 带有泛型的record类型，可以不使用全部泛型的类型变量，只替换其中部分类型变量产生新泛型类型

语法

```
type RecordName {<GenericName1, … > } = RecordNameExisted { <TypeName1, … >  }
```

例如：

```
type G2<T> = G1<int, T, string>          -- 定义一个有类型变量T的新泛型G2，这个泛型是用int,T,string分别替代G1泛型中的三个类型变量产生的新类型
type G3 = G2<string>                     -- 定义一个新类型G3，这个类型是用string替代G2泛型中的类型变量产生的新类型
```

- 泛型的实例化指用具体类型替代泛型中的类型变量，比如G2<string>就是对G2泛型的实例化
- 泛型实例化后可以直接在变量/函数参数的类型声明中使用，也可以直接用在构造函数中

比如:

```
type G2<T> = { name: string, info: T }
let a1: G2<string> = { name: 'glua', info: 'hello' }
let a2 = G2<int>({ name: 'glua', info: 123 })
```

Map类型

Map<T>类型是内置的一个基本类型,表示哈希表。

字面量语法是{}表示空Map<T>，也可以用形如 { key1 = value1, key2 = value2, key3: value3, ['key4'] = value4, 'key5': value5, ... }的形式初始化一个table值。

Map<T>可以用点操作符或者中括号的方式修改和查询其中某个索引的值，比如:

```
let a = { name='glua', age=1 }      -- 因为'glua'和1的类型不一样，a变量的类型自动推导为Map<object>类型
let a1 = { name: 'glua', age: 1 }   -- 这里a1和a两种初始化Map的方式等价
let a2 = { name: 'glua', address: 'China' }   -- 因为'glua'和'China'的类型都是string，所以a2变量的类型自动推导为Map<string>
let a3 = {name: 'glua', 'age': 1 }   -- 这种方式和a，a1等价
a['abc'] = 'China'     -- 给a插入/修改索引'abc'对应的值
a.abc = 'China'     -- 效果同a['abc'] = 'China'
let b1 = a.name      -- 读取a的'name'索引对应的值赋值给新变量b1
let b2 = a['name']    -- 同 let b2 = a.name
```

Map<T>类型的值的增删改查可以使用内置模块table模块来操作

例如:

```
var a = {name='glua'}
let b1 = a.name  -- 获取哈希表a中的'name'这个key映射的值
let b2 = a['name']  -- 获取哈希表a中的'name'这个key映射的值的另一种方式
a.name = 'China'  -- 修改或增加哈希表a中'name'这个key映射为'China'
a['name'] = 'China'  -- 修改或增加哈希表a中'name'这个key映射为'China'的另一种方式
var k: string, v: string = '', ''
for k, v in pairs(a) do
    pprint(k, v)  -- 遍历哈希表a，k和v分别是哈希表a的每一项的key和value
end
```

数组类型

- glua中数组类型是用动态数组实现的，长度可变
- 数组类型是可以存储若干个同类型值的数据结构，类型声明语法是Array<T>，其中T用具体某个类型替换，
- 数组字面量语法是:

```
[ value1, value2, … ]
```

例如：

```
let a: Array<int> = [ 1, 2, 3 ]     -- 注意，这里``>``和``=``两个符号不能连接在一起
let b: int = a[1]
```

- 数组类型和数组字面量支持嵌套和保护table字面量

例如:

```
type Person = { name: string, age: int default 0 }

let p1 = Person({name='p1'})
let p2 = Person({name='p2', age=24})
let persons: Array<Array<Person> > = [ [p1, p2], [p2, p3], [p3] ]       -- 注意，这里``>``和
``>``不能连接在一起，否则会被识别为``>>``
```

- Array<T>类型的值的增删改查可以使用内置模块table模块来操作

例如:

```
var a = [1,2,3]
table.append(a, 4)  -- 添加值到数组
a[3] = 100  -- 修改数组第3个元素
let array_length1 = table.length(a)  -- 获取数组长度
let array_length2 = #a  -- 另一种获取数组长度的方式
table.remove(a, 2)  -- 删除a数组中第2个元素（1-based索引）

var k: int, v: int = 0, 0
for k, v in pairs(a) do
    pprint(k, v)  -- 遍历数组a，k和v分别是a中每一项的索引和值，索引从1开始
end
```

table,record,Map<T>,Array<T> 类型间的转换

- table作为以上三个类型的父类型, Map<?>类型要可以用来初始化record类型

两边类型完全一样肯定接受

左值类型	右值类型	是否接受
table	record	true
table	Map<T>	true
table	Array<T>	true
record	table	true
record	Map<T>	true
record	Array<T>	false
record	其他类型的record	false
Map<T1>	table	true
Map<T1>	record	false
Map<T1>	空Map<object>	true
Map<T1>	Map<T2 where T2 extends T1>	true
Map<T1>	Map<T2 where T1 extends T2>	false
Map<T1>	Array<?>	false
Array<T1>	table	true
Array<T1>	空Map<object>	true
Array<T1>	record	false
Array<T1>	Array<T2 where T2 extends T1>	true
Array<T1>	Array<T2 where T1 extends T2>	false
Array<T1>	Map<?>	false

内置record类型

- Contract<S>类型，用于声明合约变量类型,S使用当前合约的storage类型替换

内置的Contract类型的代码实现如下：

```
type Contract<S> = {
    id: string,
    name: string,
    storage: S
}
```

使用例子如下：

```
type Storage = {
    author_name: string,
    age: int
}
let M: Contract<Storage> = {}
function M:init()
    self.storage.age = 100
    self.storage.author_name = 'glua'
    -- 这里self.id和self.name, self.storage.author_name是字符串类型，self.storage.age 是整数

end
return M
```

- Stream类型，用于作为字节流类型使用，是一个内置的record类型，但是Stream类型的对象访问其中的成员函数只能用冒号不能用点号的形式

内置的Stream的类型签名如下：

```
type Stream = {
    pos: () => int,  -- 获取字节流中当前位置
    eof: () => bool, -- 获取字节流是否到结尾
    current: () => int,  -- 获取当前字节（转成int类型），如果已经读取结束了无法读取当前字节，返回-1
    next: () => bool,    -- 如果字节流还没有到结尾，pos递进1步，返回true，否则返回false
    reset_pos: () => nil,   -- 重组字节流的当前读取位置到起始位置
    size: () => int,        -- 获取字节流的长度
    push: (int) => nil,     -- 把参数取最后一字节（转成C风格的char类型），加入字节流
    push_string: (string) => nil  -- 把参数字符串中每一个字节加入字节流
}
```

使用例子如下：

```
let a1 = Stream()
a1:push(123)    -- 对于Stream类型，不能使用a1.push(a1, 123)这种点号访问成员函数的方法
a1:push_string('abc')
let s1 = a1:size()
pprint("a1 size is ", s1)
pprint('a1 is ', tostring(a1), tojsonstring(a1))
pprint('a1 pos is', a1:pos())

var c = a1:pos()

var s = ''
while c < a1:size() do
    s = s .. tostring(a1:current())
    pprint("a1[", c, "]=", a1:current())  -- 应该依次输出 a1[0]=123   a1[1]=97    a1[2]=98
a1[3]=99
    c = c + 1
    a1:next()
end
```

语句

- 一句代码结尾的分号如果不会引起语法歧义，可以省略（一般省略掉）
- 函数可以返回一个值或者不返回值,比如

```
let abc = function (a: number)
    if a > 0 then
        return
    else
        return 1
    end
end
```

- `--` 两个连着的减号后面的本行内容是注释
- `--[[` 和 `]]` 中间的多行文本也是注释

关键字

glua语法中使用到的关键字不可以作为变量名，函数名，event名等使用

关键字列表如下：

```
if  else    elseif   while   until   repeat
end     function     true   false     then
return  and     or    not     nil     local  offline
in  goto  for     do  break    emit
```

运算操作符

- `+` 加法
- `-` 减法
- `*` 乘法
- / 浮点数除法
- // 除法，但是结果向下取整
- % 取模运算，除法余数
- ^ 幂运算
- ~= 不等于比较
- = 赋值运算
- `..` 字符串连接操作符，可以用来连接2个字符串
- == 相等比较
- > 大于
- < 小于
- >= 大于等于
- <= 小于等于
- `&` bit与运算
- `|` bit或运算
- ~ bit取反运算
- `>>` bit右移
- `<<` bit左移

- and 且，比如true and false, and和or操作符连接的多个表达式，如果不是单符号表达式，为了避免优先级导致的编译错误，请给and/or两边的复杂表达式加上一对( )，比如 (2>1) and (1>2)
- or 或，比如true or nil
- not 取布尔相反运算
- # 取数组或字符串长度运算符，比如#array

函数

函数定义

可以定义匿名函数，也可以定义命名函数（但是不能定义全局函数名称），也可以创建闭包

比如

```
let abc = function(n: number)
        return n + 1
        end
let function add2(n: number)
    return n+2
end

let M = {}
function M:sayHi()
    if 2 < 1 then
        return
    end
    print('Hi')
end
```

函数的代码块中可以有return语句，表示返回0个或一个值给调用者，也可以没有return语句。但是return语句后同条件分支后面，本函数体不能有其他语句

比如

```
let abc = function (n: number)
    return n+1
    pprint(n)             -- 这里会编译报错，return语句后不应该有其他语句
end
```

函数的参数声明可以带有参数类型声明，语法形如：(name: string, age: int)，不带类型声明时这个参数的编译期类型是object类型

更简化的函数表达式

以上函数定义的语法也可以定义匿名函数，但是有时候一些简单匿名函数写起来不方便，所以提供更简化的函数表达式语法

- 单表达式函数

函数体只能是单表达式，并且必须和函数参数在同一行

语法规则如下:

```
Args => Expr
```

比如:

```
let a1 = (a: number, b: number) => a + b  -- a1类型是(number, number) => number
let a2 = a1(1, 2) -- result is 3
```

- 多行表达式函数

函数体可以是多个语句，并且不限制必须在同一行

语法规则如下:

```
Args => do
    Block
end
```

比如:

```
let a3 = (a: number, b: number) => do
    pprint(a + b)
    return a + b
end -- a3类型是(number, number) => number
let a4 = a3(1, 2) -- 输出3并a4设为3
```

函数调用

- 函数调用的语法有两种，一是函数对象或者变量后跟着()，可以传入若干个参数，比如

```
var result = sayHi('glua', 123)
```

- 另一种函数调用语法是，如果函数调用只有一个参数并且这个参数是字符串字面量或者table字面量，则可以省略此处小括号,比如

```
var result = print 'hello'
var result2 = pprint {hello: 2}
```

- 函数调用中如果return了返回值，会返回给调用者，比如上面例子中就是把返回值赋值给result。
- 函数可能在函数体代码中的不同分支分别有return语句，运行时会根据实际运行结果返回第一个return的值。 编译期会返回各return语句的值的编译期类型进行union分析出这个函数的返回类型。比如:

```
let function hello(n: number)   -- 因为这个函数的函数体内的各return语句的返回值分别是
int,string,int，所以这个函数的返回类型是 int | string
    if n > 10 then
        return n            -- 这条return语句返回int类型
    elseif n > 3 then
        return 'hello'      -- 这条return语句返回string类型
    else
        return 0            -- 这条return语句返回int类型
    end
end
```

- 函数定义时有个语法，比如function M:init() print(self.name) end，这种冒号语法，相当于用点号，但是自动加上一个self参数表示M对象， 调用这样函数时也是类似，M:init()调用，运行时会自动把M对象作为self参数放入M.init函数的第一个参数中。但是要注意，合约的API函数定义，只支持用冒号语法。 比如

```
var t1 = {name: 'glua'}
function t1:sayHello()
```

- 函数调用时需要函数定义时的参数类型和实际调用时使用的变量或者值的类型一致，否则会报错

控制流语法

可以使用两个减号表示-后面的当前行内容是注释，也就是不作为代码处理

if/else

条件判断语法，后面跟着真假值，比如nil/false都是假值，其他的值都是真值，else表示不满足情况下执行的代码，是可选的语句，例如

```
var a = {}
if false then
    print('false')
elseif nil then
    print('ni')
elseif 2 > 1 then
    print('2>1')
elseif a then
    print('a')
else
    print('else')
end

或者
if a then
    print('a')
end
```

for

循环遍历语句，

有2种for语法，第一种是for v = e1, e2, e3 do block end 形式，其中e1是v的初始值，e2是v的结束值（v超过e2时结束循环)，e3是每次遍历对v的增加的值，e3可以是负数，e3是可选的，默认是1

比如

```
for v=1,10,2 do
    print(v)
end
```

还有一种for语法是for var_1, ···, var_n in f, step do block end 这种形式的， var_1到var_n是若干个用来循环的变量名， 每次遍历都把step和var_1到var_n的值作为参数传给函数f，结果赋值给var_1,..., var_n.一直循环直到f(step, var_1, ..., var_n)的返回值是nil为止, 比如

```
var a

let f = function(s: number, v: number)
    if not v then
        return 1
    elseif v > 10 then
        return nil
    else
        return v + s
    end
end

for a in f, 2 do
    print(a)
end

let t1 = [1,2,3]
for k: int, v: int in pairs(t1) do   -- 这里的pairs的用来遍历Map<T>, Array<T>, table的全局函数，按key排序遍历
    pprint(k, v)
end
```

while/break

语法结构while exp do block end，满足exp条件就可以继续执行block的代码块，其中也可以使用break语句来跳出循环

比如

```
var a = 1
while a < 10 do
    a = a + 1
    print(a)
    if a > 8 then
        break
    end
end
```

repeat

语法结构repeat block until exp，重复执行block的代码块，直到满足exp为真值，其中也可以使用break语句来跳出循环

比如

```
local a = 1
repeat
    a = a + 1
    print(a)
until a >= 10
```

goto

可以通过 ::labelName::的语法定义label，然后函数中其他位置通过goto labelName进行跳转，从而实现控制流的无条件转移.

比如

```
var i = 0
::s1::
do
    print(i)
    i = i+1
end
if i>3 then
    goto end_of_file
end

goto s1

::end_of_file::
print("this is end")
```

and

逻辑操作符，如果左右2个表达式都是真值，结果才是真值，比如true and false结果是false, true and true结果才是true

or

逻辑操作符，左右2个表达式有一个是真值，结果就是真值，比如true of false结果是真值，false or false结果是false

not

逻辑操作符，和右边的表达式的布尔值相反，比如not false值为true

内置全局函数

函数名 函数类型签名（返回类型 （参数函数列表）） 描述

print: (...) => nil 标准输出参数，遇到table或者函数，输出 类型名: 0000000000000000

pprint: (...) => nil 标准输出参数tojsonstring化后的结果

type: (object) => string 根据运行时参数类型输出参数的类型对应的字符串，返回number, string, table, function, boolean，遇到其他类型的参数，返回前面5种中对应的字符串

比如:

```
let a1 = 1
let a2 = type(a1) -- a2值是'number'，运行时的int和number类型，type函数返回结果都是'number'
let b1: object = 1
let b2 = type(b1) -- b2值是'number'，因为使用的是运行时类型
type Person = {}
let c1 = Person()
let c2 = type(c1) -- c2值是'table'，运行时的table和record类型，type函数返回结果都是'table'
```

require: (string) => object 引用其他的模块，不当成合约引用，被引用的模块加载后return的结果作为require函数的结果

import_contract: (string) => table 引用合约，参数是合约的名称字符串，返回合约对应的table

emit: (string, string) => nil 抛出event事件，由区块链记录

exit: (object) => object 结束本次运行，参数是结束码

pairs: (table) => object 返回table的迭代器，遍历顺序是先遍历数组部分再遍历哈希表部分,哈希表部分的数字key在string的key前遍历，同样是string类型的key的，短字符串在长字符串前，同样长度字符串的key，按ASCII字符序从小到大

ipairs: (table) => object 返回table的数组部分的迭代器

error: (...) => object 报错

getmetatable: (table) => table

tostring: (object) => string 把参数转成字符串，对于table和函数，返回 类型名: 0

tojsonstring: (object) => string 把参数转成json字符串，对于函数，返回function: 0，对于table中的嵌套table，如果有循环引用，使用'address'替代值.json化时对于哈希表会按key的字符序先按长度从小到大再从左到右依次序比较各字符的字符序。对于空table，返回"[]"

tonumber: (object) => number 把参数转成number，遇到字符串，从字符串中读取数字，遇到无法解析的字符串或table或函数，返回nil

tointeger: (object) => int 把参数转成整数，遇到字符串，从字符串中读取整数，遇到无法解析的字符串或table或函数，返回nil

todouble: (object) => number 把参数转成number类型,遇到无法解析的值返回nil

toboolean: (obj) => bool 把参数obj转成一个布尔类型的值，obj如果是false或nil返回false，否则返回true

totable: (object) => table 把参数当成table使用，如果参数不是table返回nil，主要用于编译期静态类型转换

next: (...) => object 将迭代器递进到下一步

rawequal: (object, object) => bool 直接比较两个值是否是同一个值（==比较会优先使用__eq__元函数来比较）

rawlen: (object) => int 直接获取一个table的数组部分长度

rawget: (object, object) => object 直接获取一个table的某个属性

rawset: (object, object, object) => nil 直接设置一个table的某个属性

select: (...) => object 当index为数字将返回所有index大于index的参数:如：select(2,"a","b") 返回 "b". 当index为"#"，则返回参数的总个数(不包括index)

setmetatable: (table, table) => nil 设置table的元表

模块

可以使用require '模块名称'来加载模块,比如

```
let math = require 'math'  -- math变量的值是math模块的对象
```

合约定义

一个基本的合约的格式

```
type Storage = {
    -- 合约中storage的各属性定义在这里，比如 name: string
}

var M = Contract<Storage>()

function M:init()
    -- 这里加入合约初始化逻辑
    -- 合约的storage必须在这个函数里进行初始化
end

function M:on_deposit(num: int)
    -- 可选的转账到合约的回调函数，当用户转账到合约的时候会触发这个回调,这个函数如果不需要可以不用写
end

function M:on_destroy()
    -- 可选的合约被销毁时触发的回调函数
end

function M:on_upgrade()
    -- 可选的合约升级到正式合约时触发的回调函数
end

function M:demoApi1(arg1: string)
    -- 这里是示例的用户自定义API函数，一个合约可以有多个自定义API函数，demoApi1是这里的函数名，自定义
API函数自带一个self变量表示当前合约，
    -- 另外可以有0个参数或者有一个string类型的参数
end

return M   -- 这里是必须的，表示使用哪个对象代表本合约
```

合约全局变量

合约中可以通过caller和caller_address全局变量分别访问本次发起合约调用的用户的公钥和地址

合约全局方法

合约API函数

- 使用全局函数transfer_from_contract_to_address可以从当前合约（这个函数调用代码所在的合约）转账一定数额的某种资产给某个地址，第一个参数是目标地址（字符串），第二个参数是资产名称（比如HSR），第三个参数是转账数量的10万倍（int64类型），要求是正数

```
返回值
     0    转账成功
    -1   未知系统异常
    -2   Asset_symbol异常
    -3   合约地址非法
    -4   目标地址非法
    -5   账户余额不足支付转账金额
    -6   转账金额为负数
```

- 使用全局函数get_contract_balance_amount可以获取某个合约带精度的余额（精度为100000），第一个参数是合约地址（支持查询其他合约的余额），第二个参数是资产名称（比如HSR），返回带精度的合约余额（int64类型），如果出现错误或者合约不存在返回负数

```
返回值
    非负数  合约账户余额
    -1   资产id异常
    -2   合约地址异常
```

- 使用全局函数get_chain_now可以获取链上的当前时间，没有参数.

```
返回值
    正数   时间戳整数
    0       系统异常
```

- 使用全局函数get_chain_random可以获取链上的一个伪随机数字，但是同一个此链上的operation操作，不同节点不同时间执行返回结果都一样（实际是取操作发生的块上prev_secret_hash和本交易结合后的哈希）

```
返回值
    随机结果
```

- 使用全局函数get_header_block_num，可以获取上一个块的块号

```
返回值
    当前链最新块的序号
```

- 使用全局函数get_waited(num)，表示根据未来块的数据获取伪随机数,num是未来块的块号（但是未来需要再次调用，那个时候第num块已经是过去的块了就能知道结果了）

```
返回值
    正整数 结果值
    -1   目标块未到
    -2   设定的目标块不大于1
```

- 使用全局函数get_current_contract_address可以获取这个函数调用出现位置的合约地址，没有参数
- 全局变量caller存储着调用合约的用户的公钥，全局变量caller_address存储着调用合约的用户的账户地址
- 在转账到合约发生的时候，如果合约中定义了on_deposit(参数是转账金额)这个API，那么在转账发生后会调用这个API，并且保证转账和触发此API是原子性的，如果中途出现错误，整体回滚，转账失败。
- 使用语句emit EventName(arg: string)可以抛出事件，这里emit是关键字，EventName根据需要写入事件名称，由区块链记录下来，其他节点同步到emit触发的event时可以调用本地设置的回调
- 使用全局函数 is_valid_address(arg: string)可以检查一个地址字符串是否是合法的本区块链地址
- 使用全局函数get_transaction_fee() 可以获取一笔交易的手续费

```
返回值
    正整数 结果值
    -1   手续费资产id异常
    -2   系统异常
```

- 使用全局函数transfer_from_contract_to_public_account(to_account_name: string, asset_type: string, amount: int)可以从当前合约中转账到链上的账户名称，返回是否转账的状态

```
返回值
    0    转账成功
   -1   未知系统异常
   -2   Asset_symbol异常
   -3   合约地址非法
   -4   目标地址非法
   -5   账户余额不足支付转账金额
   -6   转账金额为负数
   -7   不存在指定账户名
```

如何在合约中调用一个链上已经存在的合约

可以通过import_contract函数引用其他的链上正式合约，返回代表这个被引用的合约的对象，从而可以通过这个返回的对象调用这个被引用的用户自定义API。

但是不能直接访问被引用合约的init/on_deposit/on_destroy/on_upgrade以及storage对象，只能通过调用API访问

比如:

```
let demo = import_contract 'demo'
demo:hello("China")  -- 这里调用了名称为demo的正式合约的hello函数API，使用"China"作为参数
```

合约的内置模块的使用

合约中可以直接使用内置库的模块，不需要进行require

table模块

table.concat(table, sep, start=1, end=table的数组部分长度) 把数组table中从第start项到第end项（包括第end项）每一项用sep分隔连接成一个字符串,返回拼接后的字符串

table.insert(table, pos, value) 在数组table的第pos个位置插入一个值value，如果只传2个参数table和value，则在table的数组部分的最后位置之后插入value，无返回值

table.append(table, value) 在数组的最后位置之后插入value，无返回值

table.length(table) 获取数组的数组部分的长度

table.remove(table, pos=table数组部分长度) 函数删除并返回table数组部分位于pos位置的元素. 其后的元素会被前移. pos参数可选, 默认为table长度, 即从最后一个元素删起，返回被删除的值

table.sort(table) 函数对给定的table进行升序排序.参数table中的元素需要类型一致，无返回值

```
> tbl = {"hsrha", "beta", "gamma", "delta"}
> table.sort(tbl)
> print(table.concat(tbl, ", "))
hsrha, beta, delta, gamma
```

全局函数rawlen 返回table中数组部分元素的个数

math模块

math.abs(n) 获取参数n的绝对值

math.ceil(n) 返回不小于n的最小整数

math.floor(n) 返回不超过n的最大整数

math.max(n1,n2,...) 返回参数列表中多个值的最大值，至少需要1个参数

math.maxinteger 常数，支持的最大整数

math.min(n1,n2,...) 返回参数列表中多个值的最小值，至少需一个参数

math.mininteger 常数，支持的最小整数

math.pi 常数，π值，3.1415...

math.sqrt(n) 获取第一个参数的平方根

math.tointeger(n) 把第一个参数str字符串转成整数，如果str本身是整数，直接转成整数，如果无法转换，返回nil

math.type(num) 判断第一个参数num是整数还是浮点数

string模块

string.split(str, sep) 把str按sep划分成多块，返回一个字符串数组

string.byte(s) 返回字符串首字符对应的ASCII数字

string.char(i1, i2, ...) 把多个整数对应的ascii字符构造成字符串返回

string.find(str, pattern, [init=1, [plain=nil]]) 在str字符串中查找模式字符串pattern，从str的第init个字符开始查找，plain表示是否把pattern当成普通文本字符串而不是模式字符串来查找，返会找到的第一个满足条件的子字符串的开始字符索引或者nil

模式字符串在可以用以下符号匹配源字符串中的一些子字符串

- .(点): 与任何字符配对
- %a: 与任何字母配对
- %c: 与任何控制符配对(例如n)
- %d: 与任何数字配对
- %l: 与任何小写字母配对

- %p: 与任何标点(punctuation)配对
- %s: 与空白字符配对
- %u: 与任何大写字母配对
- %w: 与任何字母/数字配对
- %x: 与任何十六进制数配对
- %z: 与任何代表0的字符配对
- %x(此处x是非字母非数字字符): 与字符x配对. 主要用来处理表达式中有功能的字符(^$()%.[]*+-?)的配对问题, 例如%%与%配对
- [数个字符类]: 与任何[]中包含的字符类配对. 例如[%w_]与任何字母/数字, 或下划线符号(_)配对
- [^数个字符类]: 与任何不包含在[]中的字符类配对. 例如[^%s]与任何非空白字符配对

例如:

```
let p1 = "%d%d:%d%d"
let s = "2016/11/11 11:11"
let a = string.find(s, p1)  -- a的结果是11，也就是子字符串"11:11"的第一个字符的索引
let b = string.sub(s, a)    -- b结果是"11:11"
let c = string.find(s, p1, 3)  -- c结果是12
let d = string.find(s, p1, 1, true)  -- d结果是nil，因为第四个参数是true，所以把p1当成普通字符串进行匹配
```

string.format(formatstring, ...args) 类似C语言的sprintf

例如:

```
let a = string.format("hello, %s, the number is %d", "China", 123)
```

string.gmatch(str, pattern) 返回在str字符串中遍历模式字符串pattern的迭代器

例如：

```
t = {}
s = "from=world, to=Lua"
var k = nil
var v = nil
for k, v in string.gmatch(s, "(%w+)=(%w+)") do
   t[k] = v
end
```

string.gsub(str, pattern, replacer, [n]) 把str中满足pattern模式的子串投用replacer字符串或者函数进行替换，如果提供了n,只进行前n个符合的字串的替换

string.len(str) 获取字符串长度

string.match(str, pattern, [init=1]) 在str找到第一个符合模式pattern的子串，从str的第init个字符开始查找

string.rep(str, n, [sep=”]) 返回str字符串重复n次的结果，间隔符是字符串sep

string.reverse(str) 返回字符串str的反转

string.sub(str, i, [,j=-1]) 获取str字符串的子字符串，从第i个字符开始，到第j个字符结束（包含第i和第j个字符），i和j可以是负数，表示从str反方向开始的第-i/-j个字符

string.upper(str) 把字符串str各字母字符转成大写后返回

time模块

time.add(timestamp, field, offset) 返回新时间戳，field是year/month/day/hour/minute/second其中某个字符串，offset是变化值，可以是正数、负数、零

time.tostr(timestamp) 把时间戳转成时间字符串， %yy-%m-%d %H:%M:%S格式

time.difftime(timestamp1, timestamp2) 比较2个时间戳的间隔的秒数

json模块

json.dumps(任意lua值) 将lua值转成json字符串

json.loads(字符串) 将json字符串转成lua值，如果失败，返回nil

utf8模块

utf8.char(…) 接受参数中若干个数字，返回对应的UTF8编码的字节序列

utf8.charpattern 常量，能匹配UTF8字节序列的一个字符串pattern模式

utf8.codes(s) 返回变量s中所有字符的迭代器，按utf8编码

utf8.codepoint(s, i, j) 返回字符串s在索引[i, j]范围内的utf8编码的字符

utf8.len(s, i, j) 返回字符串s在索引[i, j]范围内的按utf8编码的字符长度

utf8.offset(s, n, i) 返回s中第n个字符的从第i个字节开始的字节索引

一个合约的整个生命周期流程

- 编写合约
- 编译合约
- 注册合约到区块链上成为临时合约
- 升级合约成为正式合约/销毁临时合约
- 调用合约API
- 转账到合约

合约定义的约束

- 合约作为一个特殊的模块，合约中不能定义全局变量，不能修改_ENV, _G的值，可以通过import_contract ‘合约名称’来加载合约， 返回加载的合约模块信息，合约必须返回一个record类型的对象，表示合约的api，其中必须包含一个init函数。合约有id, name, storage等内置属性，注意不要用这些名字的API，否则会被覆盖掉。
- 合约代码中，合约对象作为一个record类型，必须在合约代码结尾return这个record对象,return的这个对象代表了本合约，如果合约中用到了合约的storage，因为语法有静态类型检查，所以需要给合约的storage属性声明一个类型
- 合约的id/name/storage这三个属性都是在执行时由区块链提供值的，并且这三个属性本身是只读的，但是storage属性的内容是可以改变的
- 合约的storage需要声明为record类型，storage的record类型的各属性的类型只能是int, number, bool, string, Map<int>, Map<number>, Map<bool>, Map<string>, Array<int>, Array<number>, Array<bool>, Array<string> 其中某一种
- 内置库有一个Contract<T>泛型可以作为合约类型的基类，具体使用时可以将合约要return的变量声明为Contract的实例类型（需要提供一个record类型作为合约storage的类型，作为Contract的类型变量）

比如

```
type Storage = {                -- 这里声明一个record类型，用来作为合约的storage的类型，名称自定义
    name: string,                   -- 这里只是类型声明，合约的self.storage还是要给每一项初始化赋值才能使用
    age: int,
    age2: int default 24,               -- 这里的default值，对于合约的storage初始值不起作用，因为合约的storage是由区块链进行初始化的
    error_property: int | string        -- 这个属性会导致编译错误，因为storage的record类型的属性有类型限制
}

let M = Contract<Storage>()    --- 这里声明合约的类型是Contract<Storage>，而Contract<T>是一个泛型类型，其中泛型参数类型T是其中record属性的类型

function M:init()                   -- 因为M变量是record类型，声明M的成员函数，只能用function M:funcName的方式，不能用function M.funcName的方式
    pprint("contract init running", self.id, self.name)        -- 这里的self指当前对象也就是这里的合约对象的值
    self.storage.name = 'hi'        -- 因为这里的storage的name属性作为string类型使用，所以上面的Storage类型的name属性要声明为string类型
    let storage = self.storage
    storage.age = 100               -- 即使把self.storage赋值给其他变量，storage的类型依然是上面声明的Storage类型，编译期会检查类型
end

function M:testSomeLogic()
    let contract2 = import_contract 'contract2'    -- 这里需要引用已经上链的合约名字，如果使用不存在的合约名字，编译期会报错
    contract2.storage.name = 'glua' -- 这会编译期报错，因为合约中不能直接操作引用的其他合约的storage
    self.storage.age = self.storage.age + 1
    if self.storage.age < 100 then
        transfer_from_contract_to_address('这里填入目标地址', 'HSR', 10000000)
    end
    self.name = 'hello'             -- 报错，合约的id/name/storage属性都是只读的属性，不可修改
end

function M:query()
    pprint('query demo')
end

return M
```

- 合约中不能直接操作引用的其他合约的storage，也不能调用合约本身或者其他合约的init,on_deposit,on_upgrade,on_destroy的API，编译期会报错
- 合约代码在编译时，会加载一次合约API外的代码，所以如果合约API外代码有运行时问题也会在编译合约时报错

本地脚本

本地脚本简单介绍

本地回调脚本同样适用本语言和语法，但是不需要遵循合约的格式，不需要脚本结尾返回一个table，也可以定义全局变量。

本地回调脚本执行时会按照脚本中代码顺序依次执行。

一个简单的接受合约event的脚本

```
pprint("got event ", event_type, " from contract ", contract_id（emit触发代码所在的合约ID）, "
param is ", param)
```

本地脚本的全局变量

```
truncated: emit抛出的参数是否被截断
param: emit时抛出的字符串格式的参数
contract_id: emit触发代码所在的合约ID
event_type: 字符串格式的事件类型
```

本地脚本的模块使用

本地脚本可以通过require函数引入内置库的模块，从而在脚本中使用这些模块的功能

比如

```
let http = require 'http'
http.listen('127.0.0.1', 3000)  -- 在3000端口监听http请求
```

如何将本地脚本绑定到链上合约

在区块链的控制台中或者界面中操作按钮操作。

使用命令”compile_script 脚本源文件路径”来编译

使用命令”add_script 脚本编译后的文件路径 描述字符串”来将本地脚本注册到链上

storage操作

storage简单介绍

每个智能合约在区块链中可以存储查询一些状态数据，这个功能称作storage。

在一个合约中的API函数中，可以用self.storage获取到当前合约的storage对象，import_contract加载的合约，禁止直接读写其storage

storage对象可以类似table操作，读取属性，写入属性，但是只能嵌套最多一层table，并且storage[某个属性名]值为某个类型的话，修改这个属性的值只能改为同类型的值，如果storage[某个属性名]值为table的话，这个table中不同属性的值的类型要一样，比如都是整数或者都是字符串。

对storage的修改操作不会立刻提交，而是在当前lua堆栈关闭的时候，如果没有发生过错误，就自动提交storage的变更（只提交变更而不是storage本身）

storage的基本类型

storage中各属性的类型可以使用int, number, bool, string, Stream, Map<int>, Map<number>, Map<bool>, Map<string>, Map<Stream>, Array<int>, Array<number>, Array<bool>, Array<string>, Array<Stream>这些类型

storage操作比如：

```
self.storage.name = "hi"
self.storage.age = 123
self.storage.name = 456 -- 错误，类型不能改变
let abc = self.storage.age -- 正确，读取storage属性值
self.storage.tt = {name: "hi", age: 2} -- 错误，嵌套table的属性值的类型要一样
self.storage.tt = {name: {name: "hi"}} -- 错误，storage中嵌套table多层不允许
self.storage.tt = {name: "hi", age: "2"} -- 正确
```

5. 把智能合约注册到区块链上使用

另见API说明文档

内置库

require 加载某个模块，比如local math = require ‘math’

内置模块有string, table, math, time, json，使用时不需要require

另外，非合约模式下还可以使用更多内置模块os, net, io, http, jsonrpc等

内置全局函数

函数名 函数类型签名（返回类型 （参数函数列表）） 描述

print: (...) => nil 标准输出参数，遇到table或者函数，输出 类型名: 0000000000000000

pprint: (...) => nil 标准输出参数tojsonstring化后的结果

type: (object) => string 根据运行时参数类型输出参数的类型对应的字符串，返回number, string, table, function, boolean，遇到其他类型的参数，返回前面5种中对应的字符串

比如:

```
let a1 = 1
let a2 = type(a1) -- a2值是'number'，运行时的int和number类型，type函数返回结果都是'number'
let b1: object = 1
let b2 = type(b1) -- b2值是'number'，因为使用的是运行时类型
type Person = {}
let c1 = Person()
let c2 = type(c1) -- c2值是'table'，运行时的table和record类型，type函数返回结果都是'table'
```

require: (string) => object 引用其他的模块，不当成合约引用，被引用的模块加载后return的结果作为require函数的结果

import_contract: (string) => table 引用合约，参数是合约的名称字符串，返回合约对应的table

emit: (string, string) => nil 抛出event事件，由区块链记录

exit: (object) => object 结束本次运行，参数是结束码

pairs: (table) => object 返回table的迭代器，遍历顺序是先遍历数组部分再遍历哈希表部分,哈希表部分的数字key在string的key前遍历，同样是string类型的key的，短字符串在长字符串前，同样长度字符串的key，按ASCII字符序从小到大

ipairs: (table) => object 返回table的数组部分的迭代器

error: (...) => object 报错

getmetatable: (table) => table

tostring: (object) => string 把参数转成字符串，对于table和函数，返回 类型名: 0

tojsonstring: (object) => string 把参数转成json字符串，对于函数，返回function: 0，对于table中的嵌套table，如果有循环引用，使用'address'替代值.json化时对于哈希表会按key的字符序先按长度从小到大再从左到右依次序比较各字符的字符序。对于空table，返回"[]"

tonumber: (object) => number 把参数转成number，遇到字符串，从字符串中读取数字，遇到无法解析的字符串或table或函数，返回nil

tointeger: (object) => int 把参数转成整数，遇到字符串，从字符串中读取整数，遇到无法解析的字符串或table或函数，返回nil

todouble: (object) => number 把参数转成number类型,遇到无法解析的值返回nil

toboolean: (obj) => bool 把参数obj转成一个布尔类型的值，obj如果是false或nil返回false，否则返回true

totable: (object) => table 把参数当成table使用，如果参数不是table返回nil，主要用于编译期静态类型转换

next: (...) => object 将迭代器递进到下一步

rawequal: (object, object) => bool 直接比较两个值是否是同一个值（==比较会优先使用__eq__元函数来比较）

rawlen: (object) => int 直接获取一个table的数组部分长度

rawget: (object, object) => object 直接获取一个table的某个属性

rawset: (object, object, object) => nil 直接设置一个table的某个属性

select: (...) => object 当index为数字将返回所有index大于index的参数:如：select(2,"a","b") 返回 "b". 当index为"#"，则返回参数的总个数(不包括index)

setmetatable: (table, table) => nil 设置table的元表

table模块

table.concat(table, sep, start=1, end=table的数组部分长度) 把数组table中从第start项到第end项（包括第end项）每一项用sep分隔连接成一个字符串,返回拼接后的字符串

table.insert(table, pos, value) 在数组table的第pos个位置插入一个值value，如果只传2个参数table和value，则在table的数组部分的最后位置之后插入value，无返回值

table.append(table, value) 在数组的最后位置之后插入value，无返回值

table.length(table) 获取数组的数组部分的长度

table.remove(table, pos=table数组部分长度) 函数删除并返回table数组部分位于pos位置的元素. 其后的元素会被前移. pos参数可选, 默认为table长度, 即从最后一个元素删起，返回被删除的值

table.sort(table) 函数对给定的table进行升序排序.参数table中的元素需要类型一致，无返回值

```
> tbl = {"hsrha", "beta", "gamma", "delta"}
> table.sort(tbl)
> print(table.concat(tbl, ", "))
hsrha, beta, delta, gamma
```

全局函数rawlen 返回table中数组部分元素的个数

math模块

math.abs(n) 获取参数n的绝对值

math.ceil(n) 返回不小于n的最小整数

math.floor(n) 返回不超过n的最大整数

math.max(n1,n2,...) 返回参数列表中多个值的最大值，至少需要1个参数

math.maxinteger 常数，支持的最大整数

math.min(n1,n2,...) 返回参数列表中多个值的最小值，至少需一个参数

math.mininteger 常数，支持的最小整数

math.pi 常数，π值，3.1415...

math.sqrt(n) 获取第一个参数的平方根

math.tointeger(n) 把第一个参数str字符串转成整数，如果str本身是整数，直接转成整数，如果无法转换，返回nil

math.type(num) 判断第一个参数num是整数还是浮点数

string模块

string.split(str, sep) 把str按sep划分成多块，返回一个字符串数组

string.byte(s) 返回字符串首字符对应的ASCII数字

string.char(i1, i2, ...) 把多个整数对应的ascii字符构造成字符串返回

string.find(str, pattern, [init=1, [plain=nil]]) 在str字符串中查找模式字符串pattern，从str的第init个字符开始查找，plain表示是否把pattern当成普通文本字符串而不是模式字符串来查找，返会找到的第一个满足条件的子字符串的开始字符索引或者nil

模式字符串在可以用以下符号匹配源字符串中的一些子字符串

- .(点): 与任何字符配对
- %a: 与任何字母配对
- %c: 与任何控制符配对(例如n)
- %d: 与任何数字配对
- %l: 与任何小写字母配对
- %p: 与任何标点(punctuation)配对
- %s: 与空白字符配对
- %u: 与任何大写字母配对
- %w: 与任何字母/数字配对
- %x: 与任何十六进制数配对

- %z: 与任何代表0的字符配对
- %x(此处x是非字母非数字字符): 与字符x配对. 主要用来处理表达式中有功能的字符(^$()%.[]*+-?)的配对问题, 例如%%与%配对
- [数个字符类]: 与任何[]中包含的字符类配对. 例如[%w_]与任何字母/数字, 或下划线符号(_)配对
- [^数个字符类]: 与任何不包含在[]中的字符类配对. 例如[^%s]与任何非空白字符配对

例如:

```
let p1 = "%d%d:%d%d"
let s = "2016/11/11 11:11"
let a = string.find(s, p1)  -- a的结果是11，也就是子字符串"11:11"的第一个字符的索引
let b = string.sub(s, a)    -- b结果是"11:11"
let c = string.find(s, p1, 3)  -- c结果是12
let d = string.find(s, p1, 1, true)  -- d结果是nil，因为第四个参数是true，所以把p1当成普通字符串进行匹配
```

string.format(formatstring, ...args) 类似C语言的sprintf

例如:

```
let a = string.format("hello, %s, the number is %d", "China", 123)
```

string.gmatch(str, pattern) 返回在str字符串中遍历模式字符串pattern的迭代器

例如：

```
t = {}
s = "from=world, to=Lua"
var k = nil
var v = nil
for k, v in string.gmatch(s, "(%w+)=(%w+)") do
   t[k] = v
end
```

string.gsub(str, pattern, replacer, [n]) 把str中满足pattern模式的子串投用replacer字符串或者函数进行替换，如果提供了n,只进行前n个符合的字串的替换

string.len(str) 获取字符串长度

string.match(str, pattern, [init=1]) 在str找到第一个符合模式pattern的子串，从str的第init个字符开始查找

string.rep(str, n, [sep=”]) 返回str字符串重复n次的结果，间隔符是字符串sep

string.reverse(str) 返回字符串str的反转

string.sub(str, i, [,j=-1]) 获取str字符串的子字符串，从第i个字符开始，到第j个字符结束（包含第i和第j个字符），i和j可以是负数，表示从str反方向开始的第-i/-j个字符

string.upper(str) 把字符串str各字母字符转成大写后返回

time模块

time.add(timestamp, field, offset) 返回新时间戳，field是year/month/day/hour/minute/second其中某个字符串，offset是变化值，可以是正数、负数、零

time.tostr(timestamp) 把时间戳转成时间字符串， %yy-%m-%d %H:%M:%S格式

time.difftime(timestamp1, timestamp2) 比较2个时间戳的间隔的秒数

json模块

json.dumps(任意lua值) 将lua值转成json字符串

json.loads(字符串) 将json字符串转成lua值，如果失败，返回nil

utf8模块

utf8.char(...) 接受参数中若干个数字，返回对应的UTF8编码的字节序列

utf8.charpattern 常量，能匹配UTF8字节序列的一个字符串pattern模式

utf8.codes(s) 返回变量s中所有字符的迭代器，按utf8编码

utf8.codepoint(s, i, j) 返回字符串s在索引[i, j]范围内的utf8编码的字符

utf8.len(s, i, j) 返回字符串s在索引[i, j]范围内的按utf8编码的字符长度

utf8.offset(s, n, i) 返回s中第n个字符的从第i个字节开始的字节索引

os模块

os.clock() 返回执行该程序CPU花去的时钟秒数

os.date(format, time) 参数时间戳可选，如果没有给出时间用当前时间，返回格式化的字符串或table类型的时间信息

os.difftime(t2, t1) 返回时间戳t2-t1的差值

os.execute(command) 执行系统命令command

os.exit(code, close) 退出进程，code是错误码，close表示是否退出当前虚拟堆栈

os.getenv(varname) 获取环境变量

os.remove(filename) 删除文件或空文件夹

os.rename(oldname, newname) 重命名文件

os.setlocale(locale, category) 设置当前程序的时区，category可选默认值all，表示locale作用的范围

os.time(table) table表示时间信息的table，默认是当前时间的信息，返回时间戳

os.tmpname() 返回一个临时文件名称

io模块

io.close(file) 关闭文件

io.flush() flush输出缓冲区

io.input(file) 读取模式打开文件

io.lines(filename) 读取模式打卡文件并返回遍历文件内容中各行的迭代器

io.open(filename, mode) 按指定模式打开文件，mode可选值为读取模式'r', 写入模式'w', 添加模式'a', 保留旧内容更新模式'r+', 抛弃旧内容更新模式'w+', 保留旧内容并且只能文件尾添加的更新模式'a+'，默认是'r'

io.read(read_mode) 读取当前打开输入文件的内容并返回，参数read_mode可以有多个可选值，不同值的作用如下：

```
"*all"      读取整个文件
"*line"     读取下一行
"*number"       读取一个数字
<num>     读取一个不超过<num>个字符的字符串
```

io.seek(pos ?: int) 设置或获取当前打开文件的读写位置,如果提供pos参数，就是修改当前打开文件的读写位置，如果不提供pos参数，则是返回当前打开文件的读写位置

io.write(content: string) 把content的内容写入当前打开的文件

例如:

```
let io = require 'io';
let lines = io.lines("test/in.txt")      -- 读取文件内容，每行文本内容放入lines(table类型)
let text = table.concat(lines, ',')
io.open("test/out.txt", "w")             -- 写入模式打开文件（然后当前的打开文件是test/out.txt)
io.write(text)                           -- 把text内容写入当前打开的文件（也就是test/out.txt)
let cur_pos = io.seek()                  -- 当前打开文件的读写位置是cur_pos
io.close()                               -- 关闭当前打开的文件
```

net模块

net.listen(address, port) TCP监听address地址的port端口，监听所有地址的port端口用‘0.0.0.0’, 返回TcpServer对象

net.connect(address, port) 发起TCP连接

net.accept(server: TcpServer) tcp监听端阻塞等待TCP客户端连接，返回TcpSocket

net.accept_async(server: TcpServer, handler: Function) tcp异步监听TCP客户端连接，当出现新连接时，使用连接socket触发handler函数

net.start_io_loop(server: TcpServer) 启动TCP异步服务端事件循环，如果使用accept_async异步TCP服务，需要之后调用这个函数

net.read(socket, count) 从socket中读取count个字节

net.read_until(socket, end: string) 从socket读取字节流直到遇到end，返回结果包含end

net.write(socket, data) 把字节流或字符串写入socket

net.close_socket(socket) 关闭socket连接

net.close_server(server) 关闭TcpServer

net.shutdown() 关闭整个IO事件循环

例如:

```
let server = net.listen("127.0.0.1", 3000)
while true do
    let ctx = net.accept(server)
    let data = net.read(ctx, 10)
    pprint(data)
end
```

http模块

http.listen(address: string, port: int) 监听address地址的PORT端口的HTTP请求

http.connect(address: string, port: int) 连接到HTTP服务器端（一般不需要直接用）

http.request(method: string, url: string, body: string, headers: table) 发送http请求，返回http回复

http.close(ctx) 关闭http请求上下文

http.accept(server: HttpServer) 等待http请求，返回http请求上下文ctx

http.accept_async(server: HttpServer, handler: Function) 异步监听http请求，当接收到新http请求时，使用HttpContext对象作为参数调用handler函数

http.start_io_loop(server: HttpServer) 启动http异步服务端事件循环，如果使用accept_async异步TCP服务，需要之后调用这个函数

http.get_req_header(ctx, key: string) 获取http请求中的头信息中key的值

http.get_res_header(ctx, key: string) 获取Http回复中头信息中key的值

http.get_req_http_method(ctx) 获取http请求中的HTTP方法（字符串）

http.get_req_path(ctx) 获取http请求中的path部分

http.get_req_http_protocol 获取http请求的HTTP协议（字符串）

http.get_req_body(ctx) 获取http请求中的body内容

http.set_res_header(ctx, key: string, value: string) 设置http回复中的头信息

http.write_res_body(ctx, data: string) 向http回复中追加写入数据

http.set_status(ctx, status_code: int, status_message: string) 设置http回复中的状态码和信息

http.get_status(ctx) 获取http回复的状态码

http.get_status_message(ctx) 获取http回复中的状态信息（字符串）

http.get_res_body(ctx) 获取http回复中的body内容

http.finish_res(ctx) 把http回复内容传给客户端，必须调用这个函数才会实际回复

下面给出一个最简单的阻塞式HTTP模块的使用例子（注意这只是阻塞式API的代码例子，不建议直接使用）：

```
let http = require 'http'
let net = require 'net'

let res = http.request('GET', "http://www.gov.cn/", '', {
    Accept="text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8",
    ["User-Agent"]="Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/51.0.2704.103 Safari/537.36"
})
pprint(http.get_res_header(res, "Content-Length"))
pprint(http.get_res_body(res))
http.close(res)

let server = http.listen("0.0.0.0", 3000)

pprint("listening on 0.0.0.0:3000\n")

-- async api

let function handler(ctx)
    let net = require 'net'
    pprint("got new connection", ctx)
    -- pprint('get req body', http.get_req_body(ctx), '\n')
    net.write(ctx, "HTTP/1.1 200 OK\r\nContent-Type:text/html; utf-8\r\nContent-
Length:5\r\n\r\nhello")
    net.close_socket(ctx)
end

net.accept_async(server, handler)
net.start_io_loop(server)

pprint("starting sync http server")

while true do
    let ctx = http.accept(server)
    pprint('get req body', http.get_req_body(ctx), '\n')
    http.write_res_body(ctx, "hello world")
    http.set_status(ctx, 200, 'OK')
    http.set_res_header(ctx, "Content-Type", "text/html; utf-8")
    http.finish_res(ctx)
    http.close(ctx)
end
```

jsonrpc模块

暂时没有提供

内置全局函数

函数名 函数类型签名（返回类型 （参数函数列表）） 描述

print: (...) => nil 标准输出参数，遇到table或者函数，输出 类型名: 0000000000000000

pprint: (...) => nil 标准输出参数tojsonstring化后的结果

type: (object) => string 根据运行时参数类型输出参数的类型对应的字符串，返回number, string, table, function, boolean，遇到其他类型的参数，返回前面5种中对应的字符串

比如:

```
let a1 = 1
let a2 = type(a1) -- a2值是'number'，运行时的int和number类型，type函数返回结果都是'number'
let b1: object = 1
let b2 = type(b1) -- b2值是'number'，因为使用的是运行时类型
type Person = {}
let c1 = Person()
let c2 = type(c1) -- c2值是'table'，运行时的table和record类型，type函数返回结果都是'table'
```

require: (string) => object 引用其他的模块，不当成合约引用，被引用的模块加载后return的结果作为require函数的结果

import_contract: (string) => table 引用合约，参数是合约的名称字符串，返回合约对应的table

emit: (string, string) => nil 抛出event事件，由区块链记录

exit: (object) => object 结束本次运行，参数是结束码

pairs: (table) => object 返回table的迭代器，遍历顺序是先遍历数组部分再遍历哈希表部分,哈希表部分的数字key在string的key前遍历，同样是string类型的key的，短字符串在长字符串前，同样长度字符串的key，按ASCII字符序从小到大

ipairs: (table) => object 返回table的数组部分的迭代器

error: (...) => object 报错

getmetatable: (table) => table

tostring: (object) => string 把参数转成字符串，对于table和函数，返回 类型名: 0

tojsonstring: (object) => string 把参数转成json字符串，对于函数，返回function: 0，对于table中的嵌套table，如果有循环引用，使用'address'替代值.json化时对于哈希表会按key的字符序先按长度从小到大再从左到右依次序比较各字符的字符序。对于空table，返回"[]"

tonumber: (object) => number 把参数转成number，遇到字符串，从字符串中读取数字，遇到无法解析的字符串或table或函数，返回nil

tointeger: (object) => int 把参数转成整数，遇到字符串，从字符串中读取整数，遇到无法解析的字符串或table或函数，返回nil

todouble: (object) => number 把参数转成number类型,遇到无法解析的值返回nil

toboolean: (obj) => bool 把参数obj转成一个布尔类型的值，obj如果是false或nil返回false，否则返回true

totable: (object) => table 把参数当成table使用，如果参数不是table返回nil，主要用于编译期静态类型转换

next: (...) => object 将迭代器递进到下一步

rawequal: (object, object) => bool 直接比较两个值是否是同一个值（==比较会优先使用__eq__元函数来比较）

rawlen: (object) => int 直接获取一个table的数组部分长度

rawget: (object, object) => object 直接获取一个table的某个属性

rawset: (object, object, object) => nil 直接设置一个table的某个属性

select: (...) => object 当index为数字将返回所有index大于index的参数:如：select(2,"a","b") 返回 "b". 当index为"#"，则返回参数的总个数(不包括index)

setmetatable: (table, table) => nil 设置table的元表

一些合约相关的API

合约API函数

- 使用全局函数transfer_from_contract_to_address可以从当前合约（这个函数调用代码所在的合约）转账一定数额的某种资产给某个地址，第一个参数是目标地址（字符串），第二个参数是资产名称（比如HSR），第三个参数是转账数量的10万倍（int64类型），要求是正数

```
返回值
     0   转账成功
    -1   未知系统异常
    -2   Asset_symbol异常
    -3   合约地址非法
    -4   目标地址非法
    -5   账户余额不足支付转账金额
    -6   转账金额为负数
```

- 使用全局函数get_contract_balance_amount可以获取某个合约带精度的余额（精度为100000），第一个参数是合约地址（支持查询其他合约的余额），第二个参数是资产名称（比如HSR），返回带精度的合约余额（int64类型），如果出现错误或者合约不存在返回负数

```
返回值
    非负数 合约账户余额
    -1   资产id异常
    -2   合约地址异常
```

- 使用全局函数get_chain_now可以获取链上的当前时间，没有参数.

```
返回值
    正数   时间戳整数
    0      系统异常
```

- 使用全局函数get_chain_random可以获取链上的一个伪随机数字，但是同一个此链上的operation操作，不同节点不同时间执行返回结果都一样（实际是取操作发生的块上prev_secret_hash和本交易结合后的哈希）

```
返回值
    随机结果
```

- 使用全局函数get_header_block_num，可以获取上一个块的块号

```
返回值
    当前链最新块的序号
```

- 使用全局函数get_waited(num)，表示根据未来块的数据获取伪随机数,num是未来块的块号（但是未来需要再次调用，那个时候第num块已经是过去的块了就能知道结果了）

```
返回值
    正整数 结果值
    -1   目标块未到
    -2   设定的目标块不大于1
```

- 使用全局函数get_current_contract_address可以获取这个函数调用出现位置的合约地址，没有参数
- 全局变量caller存储着调用合约的用户的公钥，全局变量caller_address存储着调用合约的用户的账户地址
- 在转账到合约发生的时候，如果合约中定义了on_deposit(参数是转账金额)这个API，那么在转账发生后会调用这个API，并且保证转账和触发此API是原子性的，如果中途出现错误，整体回滚，转账失败。
- 使用语句emit EventName(arg: string)可以抛出事件，这里emit是关键字，EventName根据需要写入事件名称，由区块链记录下来，其他节点同步到emit触发的event时可以调用本地设置的回调
- 使用全局函数 is_valid_address(arg: string)可以检查一个地址字符串是否是合法的本区块链地址
- 使用全局函数get_transaction_fee() 可以获取一笔交易的手续费

```
返回值
    正整数 结果值
    -1   手续费资产id异常
    -2   系统异常
```

- 使用全局函数transfer_from_contract_to_public_account(to_account_name: string, asset_type: string, amount: int)可以从当前合约中转账到链上的账户名称，返回是否转账的状态

```
返回值
    0   转账成功
   -1  未知系统异常
   -2  Asset_symbol异常
   -3  合约地址非法
   -4  目标地址非法
   -5  账户余额不足支付转账金额
   -6  转账金额为负数
   -7  不存在指定账户名
```

智能合约

Contents:

智能合约介绍

首先，我们所说的智能合约是在区块链(blockchain)和加密货币(cryptocurrencies)的上下文中.

智能合约(smart contract): 在blockchains和cryptocurrencies的上下文中，智能合约是

- 预先写好的代码逻辑（我们使用lua进行编写）
- 在分布式的存储平台上进行存储和调用（blockchain）
- 可以被运行在同一区块链上的节点执行
- 运行的结果会形成交易进行存储

简单点说，智能合约就是一段可执行的代码（它可以被合约编写者赋予各种各样的功能），它经过编译然后被存储在区块链上;然后根据合约的地址，区块链上的节点可以调用它实现相关的功能.

一个简单的互助保险合约

```
type Storage= {
    participant: Array<string>,
    amount: int,
    owner: string
}

var M: Contract<Storage> = {}

function M:init()
    self.storage.participant=[]
    self.storage.amount=0
    self.storage.owner=caller_address --记录创建者
    pprint("contract init")
    emit event("contract init")
end

function M:on_deposit(amount: int)
    local in_flag: bool = false  --判断是否已经参与过合约
    for k,v in pairs(self.storage.participant) do
        if caller_address == v then
            in_flag = true
            break
        end
    end

    if in_flag == false then
        self.storage.participant[#self.storage.participant+1] = caller_address  --记录参与者
    end

    self.storage.amount = tointeger(self.storage.amount+amount)              --记录金额
    local deposit_info:string = tostring(caller_address).." transfered in,amount
"..tostring(amount)..",sum "..tostring(get_contract_balance_amount()+amount)
    pprint("deposit info: ", deposit_info)
    emit event(deposit_info)
end

function M:handle(address:string)
    if self.storage.owner ~= caller_address then --只允许创建者进行理赔
        pprint("caller_address is not the contract owner")
        pprint("caller_address: " , caller_address, " contract_owner_address: ",
self.storage.owner)
        return
    end

    local in_flag: bool = false
    for k,v in pairs(self.storage.participant) do  --遍历参与者列表，只对在参与者列表中的地址进行理
赔
        if address == v then
            in_flag = true
            local amount:int = math.floor(self.storage.amount/2)
            if amount > 10000000 then   --最高赔付100HSR
                amount = 10000000
            end
            local res=transfer_from_contract_to_address(address,"HSR",amount)
            self.storage.amount =self.storage.amount-amount
            pprint("amount: ", amount)
            local pay_info = "pay to "..address..""..tostring(amount)
            pprint("pay info: ", pay_info)
            emit event("pay to "..address..""..tostring(amount))
        end
    end

    if in_flag == false then
        pprint("caller ", caller_address, " did not take part in this contract")
    end
end

function M:get_balance()
    pprint("contract balance: ", self.storage.amount)
end

function M:get_participator()
    pprint("contract participator:")
    for k,v in pairs(self.storage.participant) do
        pprint(v)
    end
```

```
end

return M
```

合约的特殊交互类型event

合约event简单介绍

合约中可以emit抛出事件，抛出的事件记录到区块链上，区块链网络上的节点同步到这些事件时，可以根据本地配置触发相应一些脚本。

emit中的参数对应本地回调脚本执行时的几个全局变量，truncated（emit抛出的参数是否被截断），param（emit时抛出的字符串格式的参数），contract_id（emit触发代码所在的合约ID），event_type(字符串格式的事件类型)。

触发合约event的方法

可以使用emit关键字的语法来触发合约event，每次执行到emit语句时触发一个这种事件类型的合约event

语法是:

```
emit EventName(EventArg)
```

比如:

```
emit hello("glua")  -- hello是emit抛出的事件名称，"glua"是参数
EventName最长支持49个字节长的字符串，EventArg最长支持1024个字节，超长截断
```

示例

```
-- Hello, this is a example contract

-- 这里是定义合约的storage的类型
type Storage = {
    name: string,
    age: int,
    money: number,
    is_man: bool,
    int_table: Map<int>,
    int_array: Array<int>
}

-- 声明一个合约类型的变量M
let M = Contract<Storage>()

-- 给合约定义一个初始化函数，这里M代表合约的变量
function M:init()
    pprint('contract demo init')
    self.storage['name'] = 'zhangsan'  -- 这里是给合约的storage各属性进行初始化操作，下面代码类似
    self.storage['age'] = 16
    self.storage['money'] = 1.1345
    self.storage['is_man'] = true
    self.storage['int_table'] = {a: 1,b: 2, c: 3,d: 4}
    self.storage['int_array'] = [5,6,7,8]
end

-- 给合约定义一个名为set的API，参数是名为name的字符串类型的参数
function M:set(name: string)
    pprint('contract demo set')
    self.storage['name'] = 'lisi'
    self.storage['age'] = 14
    self.storage['money'] = 5.3456
    self.storage['is_man'] = false
    self.storage['int_table'].a = 15
    self.storage['int_table'].f = 10
    table.remove(self.storage['int_array'], 1)
    pprint('after remove')
    self.storage['int_array'][1] = 99
    pprint('after set array by index')
    pprint(self.storage.int_array)
    self.storage.int_array[#self.storage.int_array+1] = 20
    table.insert(self.storage['int_array'], 19)
    pprint('insert value to array')
    -- 执行从合约账户转账到用户账户的操作

transfer_from_contract_to_address("HSRB3MmTLBoh8KSokqdk1pcN6zxBKxaVeUeRfffffffffffffffffffffffffff
 "HSR", 10000)
end

function M:get(arg: string)
    pprint('test3 contract demo get')
    pprint(self.storage['name'])  -- 这里是读取合约的storage中属性为name的值并输出
    pprint(self.storage['age'])
    pprint(self.storage['money'])
    pprint(self.storage['is_man'])
    pprint(self.storage['int_table'])
    pprint(self.storage['int_array'])
end

return M
```

HOWTOs

1. 怎么进行调试

默认情况下，合法的合约调用会在代理节点执行，但是普通节点也可以手动打开合约虚拟机，验证区块链中的合约调用，执行合约调用。

简单的调试方法可以通过pprint输出变量，对于基本类型pprint会输出多个参数值的字符串表示， 对于函数会输出函数的内存地址，对于table类型的值会转成json字符串输出。

之后会增加远程调试功能，在远程代理节点上运行过程中断点调试。

2. 怎么使用event

合约中可以通过emit EventName(EventArg)抛出，EventName是自定义event名称， event名称词法要求和变量名要求一样，但是最长支持49字符, EventArg是event的参数，要求值是最长1024字符的字符串

调用合约的操作执行过程中抛出的event，会被记录到区块链中，区块链本地节点可以设置监控某合约的某event的回调脚本， 然后在区块链同步到包含被监控的event的块时，会触发设置的脚本，回调脚本也是用glua编写。

3. 常见编译错误有哪些

- 变量类型和赋值不一致
- 函数调用的参数类型和实际传参类型不一致
- 函数参数类型和函数体中使用时的类型不一致，建议函数参数加上显式类型声明
- 一些代码块漏加 `end`
- 使用了没有申明过的变量
- 合约中定义了全局变量（合约中不允许定义新的全局变量只能读取，但是脚本中可以定义新全局变量）
- 对nil值进行一些不允许的操作符操作，比如加减乘除等
- 访问非table类型值的属性
- 其他

4. 对中文或者其他非英文的支持如何

变量名，函数名，event名称不可以使用中文，只能英文字符或者下划线开头，跟着若干个英文字符或者下划线或者数字， 但是字符串中的内容可以用中文或者其他语言的文字，支持unicode。

5. 支持多线程吗

因为glua主要是为了在区块链的节点上运行的，考虑到区块链上的一些特性尤其是为了达成共识，不支持多线程编程

6. glua中怎么使用随机数

提供两种获取随机数的方式

- 全局函数get_chain_random() 获取链上伪随机数，这个函数返回的随机数是可被推算出来的结果，仅用于只需要返回均匀分布的数字的地方

- 全局函数get_waited(blocknumber) 可以获取根据指定块的块号上的二进制内容产生的一个int32数字，参数可以是过去块也可以是未来块的块号要用这种方式获取随机数，可以调用全局函数get_header_block_num()获取到前一个块的块号, 然后用未来的某个块号（当前块号加上未来块的数量，大概10秒一个块）作为参数调用get_waited函数，如果执行的时候当前区块链还没有到这个块号，返回-1， 如果执行的时候当前区块链已经超过了这个块号，返回根据那个块上数据产生的一个int32数字。以彩票为例，设置get_waited的参数为预计开奖时间的未来块号， 然后在开奖前一段时间前允许投注，这时候get_waited参数的块号还没有到这个块，返回类型是-1，所有人都不知道到开奖时间后这个函数调用的返回结果会是多少。 当开始时间到了后，调用get_waited的返回结果固定下来一个int32类型的正数，并且以后任何节点每次用同一个参数调用结果都是固定的，随机数被确定下来了。 可以根据需要用这个返回值 (result % 10000) / 10000 来得到[0, 1)之间的精度4位小数的随机数

7. 怎么实现面向对象的类型继承和多态

可以使用table类型和record类型模拟对象，record类型有默认属性值，并且属性可以是有默认实现的函数

比如:

```
type Person = {
    id: string default "123",
    name: string default "glua",
    age ?: int = 24,  -- record属性默认值既可以用default也可以用=区分
    fn: (number, number) => number default
            function (a: number, b: number)
                return a + b
            end
}

let p = Person()
pprint(p.id, p.name, p.age)
let n = p.fn(p.age, 2016)
pprint(n)
```

如果需要实现类似面向对象语言中的类型继承和多态功能的话，可以实现一个extend函数，调用子record类型的构造函数后，用extend函数 接受子对象和父类型，在extend函数中创建新的父类型的对象，然后给把父类型对象中子类型对象没有的属性赋值给子类型对象。用这种 方法可以起到继承和多态的效果。目前没有给出标准的extend函数，给出一个示例实现:

```
let function extend(obj, parent_class)
    let parent = parent_class(obj)
    for k, v in pairs(parent) do
        obj[k] = v
    return totable(obj)
end

type A = {
    name1: string,
    age1: int default 100
}

type B = {
    name: string,
    age: int
}

let b = B()
extend(b, A)

-- or

let c = extend(B(), A)
```

还有一种实现类型继承和多态的方法是使用setmetatable和元表，这以后会给出更多文档和例子

词法规则

- 整数：64位有符号整数，格式比如

3 345 0xff 0xBEBADA

- 浮点数：64位浮点数，格式比如

3.0 3.1416 314.16e-2 0.31416E1 34e1

0x0.1E 0xA23p-4 0X1.921FB54442D18P+1

- 变量名：字母或下划线开头，内容包括字母或下划线或者数字，并且不能是关键字符号
- 字符串：行内字符串可以用2个单引号或者2个双引号前后包围，其中可以使用反斜杠”转义一些特殊字符，跨行或者行内字符串可以用[[和]]包围。

比如：

```
a = 'alo\n123"'
a = "alo\n123\""
a = '\97lo\10\04923"'
a = [[alo
123"]]
a = [==[
alo
123"]==]
```

- 关键字：

```
and       break     do        else      elseif    end
false     for       function  goto      if        in
local     nil       not       or        repeat    return
then      true      until     while     offline
```

- true/false/nil字面量
- 标点符号：

```
+     -     *     /     %     ^     #
&     ~     |     <<    >>    //
==    ~=    <=    >=    <     >     =
(     )     {     }     [     ]     ::
;     :     ,     .     ..    ...
```

语法规则

```
chunk ::= block

block ::= {stat} [retstat]

type ::= Name |
       '(' {type} [‘,’ type] ‘)’  ‘=>’ type

record ::= ‘type’ Name {‘<’ { Name [‘,’ Name ] } ‘>’} ‘=’
                    ‘{‘ {  Name ‘:’ type [  ‘,’  Name ‘:’ type  ]  } ‘}’

typedef ::=  ‘type’ Name {‘<’ { Name [‘,’ Name ] } ‘>’} ‘=’  Name {‘<’ { Name [‘,’ Name ] }
‘>’}

stat ::=  ‘;’ |
     varlist ‘=’ explist |
     functioncall |
     label |
     break |
     goto Name |
     record |
     typedef |
     do block end |
     while exp do block end |
     repeat block until exp |
     if exp then block {elseif exp then block} [else block] end |
     for Name ‘=’ exp ‘,’ exp [‘,’ exp] do block end |
     for namelist in explist do block end |
     function funcname funcbody |
     local function Name funcbody |
     local namelist [‘=’ explist]

retstat ::= return [explist] [‘;’]

label ::= ‘::’ Name ‘::’

funcname ::= Name {‘.’ Name} [‘:’ Name]

varlist ::= var {‘,’ var}

var ::=  Name | prefixexp ‘[’ exp ‘]’ | prefixexp ‘.’ Name

name ::= Name {‘:’ type } || Name

namelist ::= name {‘,’ name}

explist ::= exp {‘,’ exp}

exp ::=  nil | false | true | Numeral | LiteralString | ‘...’ | functiondef |
     prefixexp | tableconstructor | exp binop exp | unop exp

prefixexp ::= var | functioncall | ‘(’ exp ‘)’

functioncall ::=  prefixexp args | prefixexp ‘:’ Name args

args ::=  ‘(’ [explist] ‘)’ | tableconstructor | LiteralString

functiondef ::= function funcbody

funcbody ::= ‘(’ [parlist] ‘)’ block end

parlist ::= namelist [‘,’ ‘...’] | ‘...’

tableconstructor ::= ‘{’ [fieldlist] ‘}’

fieldlist ::= field {fieldsep field} [fieldsep]

field ::= ‘[’ exp ‘]’ ‘=’ exp | Name ‘=’ exp | exp

fieldsep ::= ‘,’ | ‘;’

binop ::=  ‘+’ | ‘-’ | ‘*’ | ‘/’ | ‘//’ | ‘^’ | ‘%’ |
     ‘&’ | ‘~’ | ‘|’ | ‘>>’ | ‘<<’ | ‘..’ |
     ‘<’ | ‘<=’ | ‘>’ | ‘>=’ | ‘==’ | ‘~=’ |
     and | or

unop ::= ‘-’ | not | ‘#’ | ‘~’
```

FAQs

- 合约的storage的初始化是直接由区块链初始化的，record类型的default属性值不对它起作用