**创智链区块链应用系统**

**用户手册**

**北京新创智链科技有限公司**

目录

[1、引言 3](#_Toc30103752)

[1.1编写目的 3](#_Toc30103753)

[1.2概念解释 3](#_Toc30103754)

[2、关于区块链应用系统 4](#_Toc30103755)

[2.1 gichain 4](#_Toc30103756)

[2.1.1 使用方法 5](#_Toc30103757)

[2.3 区块链基础 5](#_Toc30103758)

[2.3.1 transaction/事务 6](#_Toc30103759)

[2.3.2 状态 6](#_Toc30103760)

[2.3.1 区块 7](#_Toc30103761)

[2.3.1 调用 8](#_Toc30103762)

[2.3.1 收据 8](#_Toc30103763)

[2.3 智能合约 8](#_Toc30103764)

[2.2.1 组织机构 9](#_Toc30103765)

[2.2.2 BNF范式 9](#_Toc30103766)

[2.2.3 合约标记 10](#_Toc30103767)

[2.4 合约规范 20](#_Toc30103768)

[2.4.1 BNF范式定义 20](#_Toc30103769)

[2.4.2 包结构规范 23](#_Toc30103770)

[2.4.3 白名单列表 24](#_Toc30103771)

# 1、引言

## 1.1编写目的

本文档提供的对象主要是区块链应用用户。本文档对区块链应用系统主要功能和操作进行介绍，为用户操作、使用区块链应用系统软件提供参考和指导。

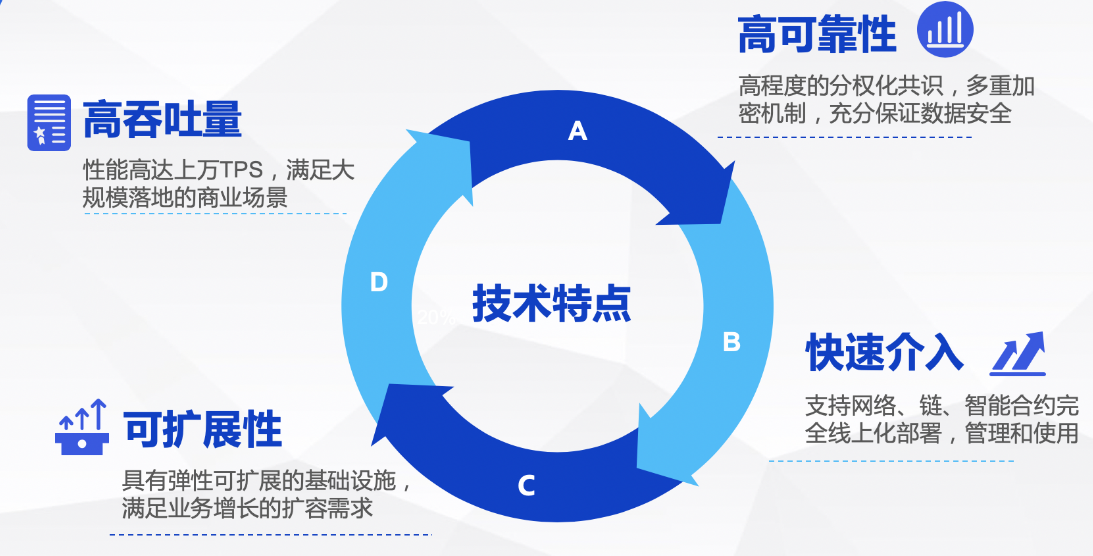
## 1.2概念解释

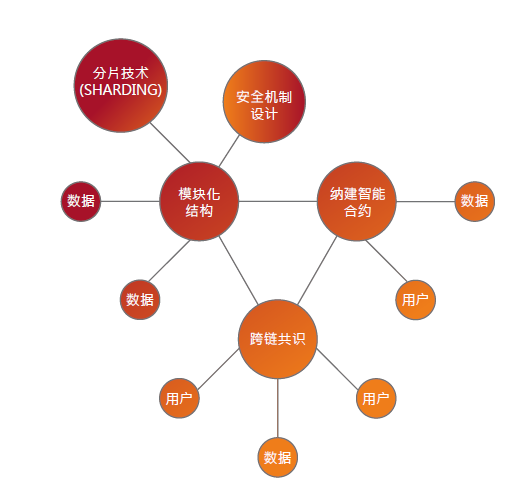
什么是gichain。

gichain是以Tendermint为基础开发的区块链体系，以系统安全性为依归，进行技术创新，实现物与物、人与物之间高效的价值信息传递，打造快速应用，具备高性能及高扩展性的平台。

什么是smcrunsvc？

Smcrunsvc是运行在docker容器中的合约应用程序，每一个组织的合约都会运行在单独的容器中。在创世组织中包含了创世合约以及基础合约。





# 2、关于区块链应用系统

## 2.1 gichain

命令运行格式如下：

ABCI application

Usage:

gichain [command]

Available Commands:

help Help about any command

start Run the ABCI application

unsafe\_reset\_all (unsafe) Remove all the data

version Show version info

Flags:

-h, --help help for gichain

Use "gichain [command] --help" for more information about a command.

### 2.1.1 使用方法

1. gichain start

启动gichain应用程序，已经集成到在部署安装包脚本中，此处不再赘述

1. gichain unsafe\_reset\_all

**注意：此操作是不安全的，只有在开发中或者你可以承担的起丢失区块数据的风险，才可以进行此操作。**

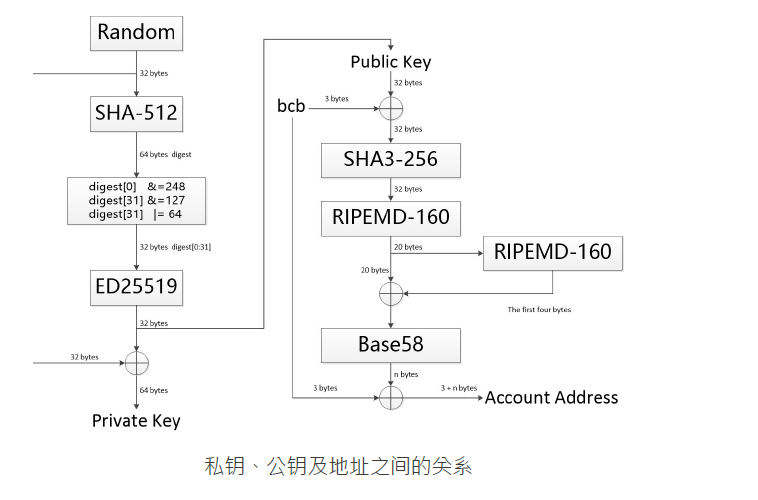
此命令删除gichain的应用数据库的数据信息（.appstate.db）

1. gichain version

查看gichain的当前版本

## 2.3 区块链基础

程序员对于区块链这个概念应该并不难理解，这是因为大多数复杂的算法与协议 (哈希 ，椭圆曲线密码学 ，点对点网络（P2P） 等) 都只是用于提供特定的功能和承诺。对于智能合约开发程序员来说，只需接受这些既有的特性与功能，不必关心底层技术的实现方案。



### 2.3.1 transaction/事务

区块链是一种全球共享的事务性数据库，这意味着每个人都可加入网络来阅读数据库中的记录。如果你想改变数据库中的某些东西，你必须创建一个被所有其他人所接受的事务（在区块链世界又被称为transaction）。事务一词意味着你想做的（假设您想要同时更改两个值），要么一点没做，要么全部完成。此外，当你的事务被应用到数据库时，其他事务不能修改数据库。

举个例子，设想一张表，列出所有账户的余额。如果你请求从一个账户转移到另一个账户，数据库的事务特性确保了如果从一个账户扣除金额，它总被添加到另一个账户。如果由于某些原因，无法添加金额到目标账户时，源账户也不会发生任何变化。

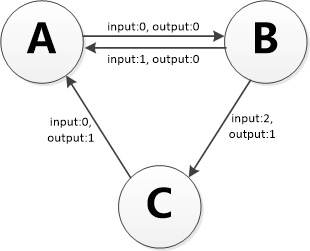
此外，transaction总是由发送人（创建者）签名。这样，就可非常简单地为数据库的特定修改增加访问保护机制。

### 2.3.2 状态

gichain参考了Ethereum、Fabric、Tendermint、Cosmos等开源区块链方案，借鉴了其中一些优秀思想。

gichain在本质上是一个基于transaction的状态机(transaction-based state machine)。在计算机科学中，状态机是一个包含一组状态集（states）、一个起始状态（start state）、一组输入符号集（alphabet）、一个映射输入符号和当前状态到下一状态的转换函数（transition function）的计算模型。

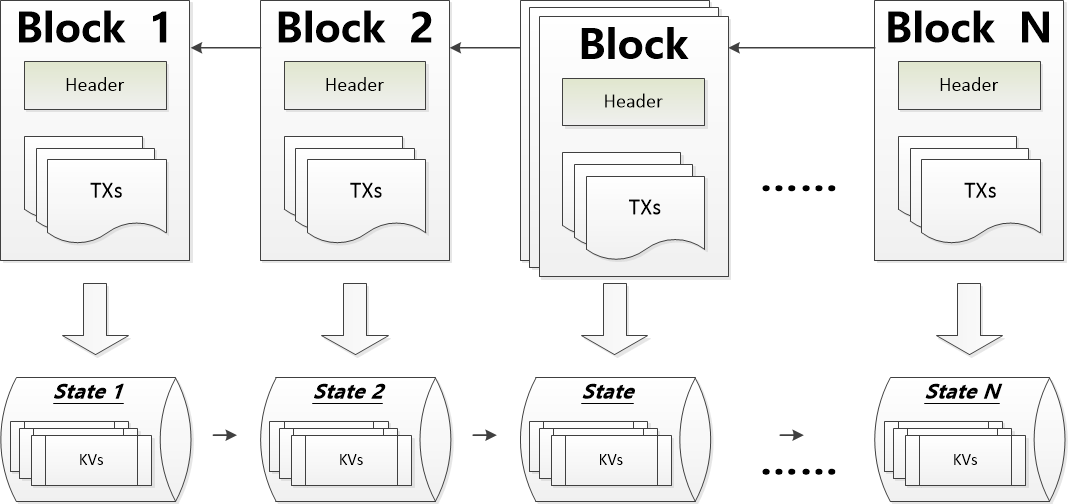
在gichain其中状态集由状态数据库进行表达，起始状态被称为创世状态（genesis state），输入的符号集就是在区块链领域常说的transaction（transaction，简称tx），状态转换函数就是智能合约。



根据gichain的状态机，我们从创世状态(genesis state)开始。这差不多类似于一片空白的石板，在网络中还没有任何transaction产生的状态。当transaction被执行后，这个创世纪状态就会转变成最终状态。在任何时刻，这个最终状态都代表着gichain当前的状态。

### 2.3.1 区块

gichain的状态由成千上万个transaction达成。这些transaction都被 “组团” 到一个个区块中。一个区块包含了一系列的transaction，每个区块都与它的前一个区块链接起来，这正是“区块链”这个词的来源，每一个区块都会导致状态机达到一个新的状态。



为了让一个状态转换成下一个状态，transaction必须是有效的（也就是促使采用区块链技术的不可抵赖特性需求）。为了让一个transaction被认为是有效的，它必须要经过一个验证过程，每一个transaction都必须由transaction发起人通过自己的私钥进行签名，并且在gichain的智能合约中校验满足一定的条件，才能被认为是有效的。

### 2.3.1 调用

智能合约可以通过消息调用的方式来调用其它智能合约。

每个transaction都可以包含多个按顺序执行的顶层消息调用，前一个消息调用的输出收据可以作为下一个顶层消息调用的

输入，在下一个顶层消息调用中执行一些逻辑处理代码。每一个顶层消息调用，这个消息调用会依次产生更多的跨合约的消息调用。

消息调用层数被限制为8层，为了预防死循环，各层消息调用不能形成环路。

### 2.3.1 收据

每一个消息调用除了返回结果以外，还可以返回执行合约业务逻辑过程中的日志数据，在这里我们把它称为收据，这些收据数据保存在区块上，可以很方便的进行检索。

## 2.3 智能合约

智能合约是一种计算机协议，其目的是以数字方式验证合约的协商。 他们不仅以与传统合同相同的方式定义与协议相关的规则和处罚，而且还可以自动执行这些义务。 如果满足预定义规则，则自动执行协议。 智能合约代码促进、验证和执行协议或transaction的协商或执行。 它是去中心化自动化的最简单形式。

​gichain的智能合约支持[Solidity](https://en.wikipedia.org/wiki/Solidity) 编程语言开发和go语言开发。 gichain的Solidity合约，完全与以太坊的EVM环境兼容，开发者可以非常方便的把以太坊中的智能合约移植到gichain区块链的BVM中，同时开发人员可以在具有Solidity的混合环境中构建，调试和执行智能合约。

​gichain区块链中的go语言合约，通过插件技术实现，主要解决当今的主流智能合约，无法编写复杂的业务逻辑（比如上万行代码），且大量合约之间的互相调用，使得合约的开发调试和维护升级的复杂度变得非常大的问题。

从V2.0版本开始支持独立开发与部署智能合约，智能合约采用golang进行开发。

V2.2版本开始支持solidity语言的智能合约，运行在bvm虚拟机中。

本章节只介绍golang进行开发的智能合约

### 2.2.1 组织机构

为gichain主链开发的智能合约按照组织的架构进行组织，为同一个组织机构开发的所有智能合约都将被集成在一个合约进程中供用户进行调用。

在一个智能合约中可以调用其它智能合约提供的接口，在那个接口对应的智能合约代码中又可以调用另一个智能合约的接口，gichain最多支持8级嵌套的跨合约调用，嵌套调用不允许包含环形调用结构。

gichain设定了一个基础组织，为该基础组织开发的智能合约将被集成到所有组织的合约进程中供用户调用，这个组织的设定主要是为了提供一些可以随时被任意组织跨合约调用执行的基础合约，例如gichain主链的基础通证模板合约。

### 2.2.2 BNF范式

本章对合约规范的描述采用巴科斯(BNF)范式。

巴科斯范式的英文缩写为BNF，它是以美国人巴科斯(Backus)和丹麦人诺尔(Naur)的名字命名的一种形式化的语法表示方法，用来描述语法的一种形式体系，是一种典型的元语言。又称巴科斯-诺尔形式(Backus-Naur form)。它不仅能严格地表示语法规则，而且所描述的语法是与上下文无关的。它具有语法简单，表示明确，便于语法分析和编译的特点。

BNF范式表示语法规则的方式为：

- 非终结符用尖括号括起。

- 每条规则的左部是一个非终结符，右部是由非终结符和终结符组成的一个符号串，中间一般以::=分开。

- 具有相同左部的规则可以共用一个左部，各右部之间以竖线“|”隔开。

BNF范式中常用的元字符及其表示的意义如下：

1. 在双引号中的字符(例如"word")代表着这些字符本身。而double\_quote用来代表双引号本身。

2. 在双引号外的字(有可能有下划线)代表着语法部分。

3. 尖括号 < > 内包含的为必选项。

4. 方括号 [ ] 内包含的为可选项。

5. 大括号 { } 内包含的为可重复0至无数次的项。

6. 圆括号 ( ) 内包含的所有项为一组，用来控制表达式的优先级。

7. 竖线 | 表示在其左右两边任选一项，相当于"OR"的意思。

8. ::= 是“被定义为”的意思。

9. 空白字符 BNF范式定义中出现的空白字符间隔仅为排版需要，不作为规范的一部分。

### 2.2.3 合约标记

gichain智能合约代码采用标记法在合约代码的注释中对合约的元数据进行描述。

下面为gichain智能合约所使用的各种标记的详细语法描述。

#### 2.2.3.1 contract

标记**contract**用于标识合约名称，同一组织之下的合约名称必须唯一，请智能合约的开发者进行合理的规划。

标记**contract**为必须标记，但在整个合约代码中只能出现一次。

标记**contract**的BNF范式定义如下：

<contract标记语法> ::= "//@:contract:" <合约名称>

<合约名称> ::= <字母> | <合约名称> <字母数字串>

<字母数字串> ::= <字母> | <十进制数字> | <字母数字串> <字母> | <字母数字串> <十进制数字>

<字母> ::= "\_" | "-" | "." | <小写字母> | <大写字母>

<小写字母> ::= "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" |

"m" | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" |

"y" | "z"

<大写字母> ::= "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L" |

"M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" |

"Y" | "Z"

<十进制数字> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | 8 | "9"

标记**contract**之后的有效代码必须紧跟一个包含**sdk.ISmartContract**成员的合约类定义，这个合约类为智能合约的上下文环境，每次针对智能合约的消息调用都会自动创建这个合约类的一个实例，在这个实例上调用智能合约的方法。智能合约对外提供的方法必须属于这个类的成员函数。类名首字母必须大写。

示例如下：

//@:contract: mydonation

type Mydonation struct {

sdk sdk.ISmartContract

//Total donations received by donees

//@:public:store

donations map[types.Address]bn.Number // key=address of donee

}

#### 2.2.3.2 version

标记**version**用于标识合约的版本，在整个合约代码中只能出现一次。

标记**version**为必须标记，但在整个合约代码中只能出现一次。

标记**version**的BNF范式定义如下：

<version标记语法> ::= "//@:version:" <合约版本>

<合约版本> ::= <十进制数> |

<十进制数> "." <十进制数> |

<十进制数> "." <十进制数> "." <十进制数> |

<十进制数> "." <十进制数> "." <十进制数> "." <十进制数>

<十进制数> ::= <十进制数字> | <十进制数> <十进制数字>

<十进制数字> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | 8 | "9"

示例如下：

//@:version:1.0

注：gichain链不要求智能合约版本的具体规则，即合约版本既可以采用一段式（例如1、2、3），也可以采用两段式（1.0、1.1、1.2）、三段式（1.0.1、1.0.2、1.0.3）、四段式（1.0.1.102、1.0.1.103），但是同一智能合约的不同版本需要保持版本格式段数的一致性。

#### 2.2.3.3 organization

标记**organization**用于标识合约所属组织机构的ID。

标记**organization**为必须标记，但在整个合约代码中只能出现一次。

标记**organization**的BNF范式定义如下：

<organization标记语法> ::= "//@:organization:" <组织ID>

<组织ID> ::= <前缀码> <Base58字符串>

<前缀码> ::= "org"

<Base58字符串> ::= <Base58字符> | <Base58字符串> <Base58字符>

<Base58字符> ::= <十进制数字> | <大写字母> | <小写字母>

<十进制数字> ::= "1" | "2" | "3" | "4" | "5" | "6" | "7" | 8 | "9"

<大写字母> ::= "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "J" | "K" | "L" | "M" |

"N" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z"

<小写字母> ::= "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "m" |

"n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" |

"z"

示例如下：

//@:organization:orgBtjfCSPCAJ84uQWcpNr74NLMWYm5SXzer

#### 2.2.3.4 author

标记**author**用于标识合约作者的账户公钥。

标记**author**为必须标记，但在整个合约代码中只能出现一次。

标记**author**的BNF范式定义如下：

<author标记语法> ::= "//@:author:" <账户公钥>

<账户公钥> ::= <十六进制字符串>

<十六进制字符串> ::= <十六进制数字> | <十六进制字符串> <十六进制数字>

<十六进制数字> ::= "0" | 1" | "2" | "3" | "4" | "5" | "6" | "7" | 8 | "9" |

"A" | "B" | "C" | "D" | "E" | "F" |

"a" | "b" | "c" | "d" | "e" | "f"

示例如下：

//@:author:b37e7627431feb18123b81gif1f41ffd37efdb90513d48ff2c7f8a0c27a9d06c

注：十六进制公钥字符串长度必须等于64。

#### 2.2.3.5 constructor

标记constructor用于标识合约的上链初始化函数。

标记constructor之后的有效代码必须紧跟一个名为InitChain或UpdateChain的无参数函数，该函数将在gichain链上第一次部署(InitChain)或者升级(UpdateChain)这个合约的时候自动进行唯一一次调用，以完成智能合约在区块链上的初始化工作，例如初始化某些全局状态数据的初始值。

标记constructor为可选标记，但在整个合约代码中最多只能出现两次，一次标识 InitChain函数，一次标识UpdateChain函数。

标记constructor的BNF范式定义如下：

<author标记语法> ::= "//@:constructor"

示例如下：

//@:constructor

func (m \*Mydonation) InitChain() {

...

}

//@:constructor

func (m \*Mydonation) UpdateChain() {

...

}

#### 2.2.3.6 public:store

标记**public:store**用于标识一个状态变量，状态变量必须是通过标记```contract```标注的合约类的成员变量。

标记**public:store**之后的有效代码必须紧跟一个成员变量的定义。

标记**public:store**为可选标记，在合约类定义中可以出现多次，每次标识一个状态变量。

标记**public:store**的BNF范式定义如下：

<public:store标记语法> ::= "//@:public:store"

示例如下：

//@:contract: mydonation

type Mydonation struct {

sdk sdk.ISmartContract

...

//@:public:store:cache

totalDonations bn.Number

//@:public:store

donations map[types.Address]bn.Number

}

标记**public:store**标识的状态变量在合约代码中不能直接访问，根据gichain合约规范，gichain链提供的SDK配套工具会自动生成访问该状态变量的读写函数，示例如下：

//读取状态变量donations的函数

func (m \*Mydonation) \_donations(k types.Address) bn.Number {

temp := bn.N(0)

return \*m.sdk.Helper().StateHelper().GetEx(fmt.Sprintf("/donations/%v", k), &temp).(\*bn.Number)

}

//检测状态数据donations是否存在的函数

func (m \*Mydonation) \_chkDonations(k types.Address) bool {

return m.sdk.Helper().StateHelper().Check(fmt.Sprintf("/donations/%v", k))

}

//设置状态变量donations的函数

func (m \*Mydonation) \_setDonations(k types.Address, v bn.Number) {

m.sdk.Helper().StateHelper().Set(fmt.Sprintf("/donations/%v", k), &v)

}

//从状态数据库中删除状态变量donations的键值的函数

func (m \*Mydonation) \_delDonations(k types.Address) {

m.sdk.Helper().StateHelper().Delete(fmt.Sprintf("/donations/%v", k))

}

#### 2.2.3.7 public:store:cache

标记**public:store:cache**用于标识一个可缓存在内存的状态变量，状态变量必须是通过标记contract标注的合约类的成员变量。

标记**public:store:cache**之后的有效代码必须紧跟一个成员变量的定义。

标记**public:store:cache**为可选标记，在合约类定义中可以出现多次，每次标识一个状态变量。

标记**public:store:cache**的BNF范式定义如下：

<public:store:cache标记语法> ::= "//@:public:store:cache"

示例如下：

//@:contract:mydonation

type Mydonation struct {

sdk sdk.ISmartContract

...

//@:public:store:cache

totalDonations bn.Number

}

可缓存的状态变量在合约代码中不能直接访问，根据gichain合约规范，gichain提供的SDK配套工具会自动生成访问该状态变量的读写函数，示例如下：

//读取状态变量totalDonations的函数

func (m \*Mydonation) \_ totalDonations() bn.Number {

return \*m.sdk.Helper().StateHelper().McGetEx(

"/totalDonations", &bn.Number{V: big.NewInt(0)}).(\*bn.Number)

}

//检测状态数据totalDonations是否存在的函数

func (m \*Mydonation) \_chkTotalDonations () bool {

return m.sdk.Helper().StateHelper().McCheck("/totalDonations")

}

//设置状态变量totalDonations的函数

func (m \*Mydonation) \_setTotalDonations (v bn.Number) {

m.sdk.Helper().StateHelper().McSet("/totalDonations", &v)

}

//清除状态变量totalDonations的内存缓存的函数

func (m \*Mydonation) \_clrTotalDonations() {

m.sdk.Helper().StateHelper().McClear("/totalDonations")

}

//从状态数据库中删除状态变量totalDonations

func (m \*Mydonation) \_delTotalDonations() {

m.sdk.Helper().StateHelper().McDelete("/totalDonations")

}

#### 2.2.3.8 public:receipt

标记**public:receipt**用于标识合约中所有收据的定义。

标记**public:receip**t之后的有效代码必须紧跟一个名为 ```receipt``` 的接口定义。

标记**public:receipt**为可选标记，在整个合约代码中最多只能出现一次。

标记**public:receipt**的BNF范式定义如下：

<public:receipt标记语法> ::= "//@:public:receipt"

示例如下：

//@:public:receipt

type receipt interface {

...

emitTransferDonation(donee types.Address, value, balance bn.Number)

}

接口**receipt**之中定义的每一个发送收据的方法必须以**emit**开头。**emit**之后的第一个单词首字母转为小写后就是这个收据的名称，可以根据收据名称到gichain链上进行检索。根据gichain合约规范，gichain提供的SDK配套工具会自动生成发送收据函数的实现代码，示例如下：

//下面的函数是由配套工具自动生成的

func (m \*Mydonation) emitTransferDonation(donee types.Address, value, balance bn.Number) {

type transferDonation struct {

Donee types.Address `json:"donee"`

Value bn.Number `json:"value"`

Balance bn.Number `json:"balance"`

}

m.sdk.Helper().ReceiptHelper().Emit(

transferDonation{

Donee: donee,

Value: value,

Balance: balance,

},

)

}

//下面是发送收据的示例代码，位于Transfer函数的实现当中

func (m \* Mydonation) Transfer(to types.Address, value bn.Number) {

//实现trasnsfer的业务逻辑代码

...

//发送trasnsfer收据

d.emitTransferDonation(

to,

value,

balance,

)

}

#### 2.2.3.9 public:method

标记**public:method**用于标识合约的公开方法。

标记**public:method**之后的有效代码必须紧跟一个针对通过标记**contract**标注的合约类的成员函数定义（函数名称必须由大写字母开头），这个合约的成员函数可以通过gichain链的广播transaction来执行。

标记**public:method**为可选标记，在整个合约代码中可以出现零次、一次或多次。

标记**public:method**的BNF范式定义如下：

<public:method标记语法> ::= "//@:public:method:gas[" <燃料数量> "]"

<燃料数量> ::= ["-"] <十进制数>

<十进制数> ::= <十进制数字> | <十进制数> <十进制数字>

<十进制数字> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | 8 | "9"

注：

\* 燃料数量为正整数表示该方法调用消耗的燃料费用由transaction最初的发起者支付；

\* 燃料数量为0表示不需要支付手续费；

\* 燃料数量为负整数表示该方法调用消耗的燃料费用由当前智能合约的账户进行支付。

示例如下：

//@:public:method:gas[500]

func (m \*Mydonation) Transfer(to types.Address, value bn.Number) {

...

}

#### 2.2.3.10 public:interface

标记**public:interface**用于标识合约的公开接口。

标记**public:interface**之后的有效代码必须紧跟一个针对通过标记**contract**标注的合约类的成员函数定义（函数名称必须由大写字母开头），这个合约的成员函数可以从别的合约通过跨合约调用机制来执行。

标记**public:interface**为可选标记，在整个合约代码中可以出现零次、一次或多次。

标记**public:interface**的BNF范式定义如下：

<public:interface标记语法> ::= "//@:public:interface:gas[" <燃料数量> "]"

<燃料数量> ::= <十进制数>

<十进制数> ::= <十进制数字> | <十进制数> <十进制数字>

<十进制数字> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | 8 | "9"

示例如下：

//@:public:interface:gas[450]

func (m \*Mydonation) Transfer(to types.Address, value bn.Number) {

...

}

注：

- 燃料数量大于等于0。

标记**public:interface**标记**public:method**可以同时标注合约类的同一个成员函数，上面的例子实际上应该定义如下：

//@:public:method:gas[500]

//@:public:interface:gas[450]

func (m \*Mydonation) Transfer(to types.Address, value bn.Number) {

...

}

#### 2.2.3.11 import

标记**import**用于导入一个外部合约的接口原型，方便在当前合约中调用外部合约。

标记**import**为可选标记，在整个合约代码中可以出现多次，每次导入一个外部合约的跨合约调用接口，每一个外部合约只能导入一次。

标记**import**的BNF范式定义如下：

<import标记语法> ::= "//@:import:" <合约名称>

<合约名称> ::= <字母> | <合约名称> <字母数字串>

<字母数字串> ::= <字母> | <十进制数字> | <字母数字串> <字母> | <字母数字串> <十进制数字>

<字母> ::= "\_" | "-" | "." | <小写字母> | <大写字母>

<小写字母> ::= "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" |

"m" | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" |

"y" | "z"

<大写字母> ::= "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L" |

"M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" |

"Y" | "Z"

<十进制数字> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | 8 | "9"

以前面的mydonation慈善捐款合约为例，如果在其它合约中需要调用该合约，需要编写如下代码：

//本段代码在合约mycontract中

//@:import: mydonation

type mydonation interface {

Transfer(to types.Address, value bn.Number)

}

外部合约的接口名称自定义，只要遵循**golang**语法规范即可。

外部合约的接口在合约代码中不能直接访问，根据gichain合约规范，gichain提供的 **SDK**配套工具会自动生成访问该外部合约接口的实现函数，自动生成的代码示例如下：

//本段代码是由SDK配套工具自动生成的

//mydonation This is method of MyContract

func (m \*MyContract) mydonation () \*InterfacemydonationStub {

return &InterfacemydonationStub{}

}

//Transfer This is a method of InterfacemydonationStub

func (is \*InterfacemydonationStub) Transfer(to types.Address, value bn.Number) {

return

}

//run Transfer some receipts to destination contract

func (is \*InterfacemydonationStub) run(f func()) (\*InterfacemydonationStub) {

...

f()

...

return is

}

//contract Wrap the destination contract information

func (is \*InterfacemydonationStub) contract() IContract {

...

return contract

}

下面是在合约代码中调用外部合约的示例代码：

//本段代码在合约mycontract中

func (m \*MyContract)TransferTest(to types.Address, value bn.Number) {

m. mydonation().Transfer(to, value)

}

跨合约调用还支持向被调用合约传递收据，以下为示例代码（从合约 **contract2**中调用合约 **contract1**的接口）：

合约1，提供跨合约接口服务：

//@:contract:contract1

type C1 struct {

sdk sdk.ISmartContract

...

}

//@:public:interface:gas[450]

func (c \*C1) Register() {

//I need receipt from caller

transfers := dw.sdk.Message().GetTransferToMe()

sdk.Require(transfers!=nil && len(transfers)==1,

types.ErrInvalidParameter, "Please transfer to me first")

value := transfers[0].Value

...

}

合约2，跨合约调用者：

//@:contract:contract2

type C2 struct {

sdk sdk.ISmartContract

...

}

//@:import:contract1

type mydonation interface {

Register()

}

//@:public:method:gas[450]

func (c \*C2) Register() {

...

c.contract1().run(func(){

c.sdk.Message().Contract().Account.TransferByName(

"CNY",

c.contract1().contract().Account().Address(),

bn.N(1000000000),

)

}).Register()

...

}

## **2.4 合约规范**

### 2.4.1 BNF范式定义

综合上面对合约标记的描述，下面定义合约规范的**BNF**范式：

<智能合约> ::= <合约定义代码文件> {<合约实现代码文件>} {<合约测试代码文件>}

<合约定义代码文件> ::= <代码包定义> <合约定义代码>

<合约实现代码文件> ::= <代码包定义> <合约实现代码>

<合约测试代码文件> ::= <代码包定义> <合约测试代码>

<代码包定义> ::= "package" <合约包名>

<合约包名> ::= 遵循glang语法规范，但不能为 std

<合约定义代码> ::= "import ("

<合约SDK包根路径>

{[包别名] <合约支撑代码包路径>}

")"

<合约类定义>

[<合约上链初始化函数定义>]

[<挖矿定义>]

[<合约收据定义>]

{<跨合约调用接口定义>}

{<合约公开函数定义>}

{<合约实现代码>}

<合约SDK包根路径> ::= double\_quote "blockchain/smcsdk/sdk" double\_quote

<合约支撑代码包路径> ::= 遵循golang代码包路径规范，遵循gichain合约规范的白名单与灰名单规范

<包别名> ::= 遵循golang代码规范的代码包别名，不允许使用 '.'

<合约实现代码> ::= 遵循golang代码规范的合约实现代码（不需要gichain合约标记的代码），包括类型定义、

常量定义、函数定义（注：不能包含全局变量定义，不允许使用 for 关键字，不允许递归

调用）

<合约测试代码> ::= 遵循golang单元测试规范的测试代码

<合约类定义> ::= "//@:contract:" <合约名称>

"//@:version:" <合约版本>

"//@:organization:" <组织ID>

"//@:author:" <账户公钥>

"type " <合约类名> " struct {"

" sdk sdk.ISmartContract"

{<状态变量定义>}

{<golang变量定义>}

"}"

<合约名称> ::= 参见<合约标记:contract>

<合约版本> ::= 参见<合约标记:version>

<组织ID> ::= 参见<合约标记:organization>

<账户公钥> ::= 参见<合约标记:author>

<合约类名> ::= <大写字母开头的标识符>

<golang变量定义> ::= 遵循golang代码规范的标准变量定义代码

<状态变量定义> ::= <基本状态变量定义> | <带缓存的状态变量定义>

<基本状态变量定义> ::= "//@:public:store"

<变量名称> ["\*"] <变量类型>

<带缓存的状态变量定义> ::= "//@:public:store:cache"

<变量名称> ["\*"] <变量类型>

<变量名称> ::= <标识符>

<变量类型> ::= <元数据类型> | <数组类型> | <映射表类型>

<合约上链初始化函数定义> ::= <部署函数> | <升级函数>

<部署函数> ::= "//@:constructor"

"func (" <合约对象定义> ") InitChain() {"

<上链代码>

"}"

<升级函数> ::= "//@:constructor"

"func (" <合约对象定义> ") UpdateChain() {"

<上链代码>

"}"

<挖矿定义> ::= "//@:public:mine"

"func (" <合约对象定义> ") Mine() int64 {"

<挖矿代码>

"}"

<合约对象定义> ::= <变量名称> "\*" <合约类名>

<上链代码> ::= <golang函数体>

注1：只允许访问状态变量

注2：sdk中不允许访问Message()和Tx()

<挖矿代码> ::= <golang函数体>

注1：sdk中不允许访问Message()和Tx()

<golang函数体> ::= 遵循golang代码规范的函数体实现代码，参见 <合约实现代码> 的定义

<合约收据定义> ::= "//@:public:receipt"

"type receipt interface {"

<收据函数名称> <函数入口参数定义>

"}"

<收据函数名称> ::= "emit" <大写字母开头的标识符>

<函数入口参数定义> ::= "(" <参数表> ")"

<参数表> ::= <参数定义> | <参数表> "," <参数定义>

<参数定义> ::= <变量名称> ["\*"] <变量类型>

<跨合约调用接口定义> ::= "//@:import:" <合约名称>

"type "<接口类名称>" interface {"

<接口函数名称> <函数入口参数定义> <函数返回定义>

"}"

<接口类名称> ::= <大写字母开头的标识符>

<接口函数名称> ::= <大写字母开头的标识符>

<函数返回定义> ::= <空> | ["\*"] <变量类型> | "(" <返回表> ")"

<空> ::= 空白

<返回表> ::= <返回定义> | <返回表> "," <返回定义>

<返回定义> ::= [<变量名称>] ["\*"] <变量类型>

<合约公开函数定义> ::=[<合约公开方法标记>]

[<合约公开接口标记>]

"func (" <合约对象定义> ")" <公开函数名称> <函数入口参数定义> <函数返回定义> "{"

<golang函数体>

"}"

<合约公开方法标记> ::= "//@:public:method:gas[ "<燃料数量> "]"

<合约公开接口标记> ::= "//@:public:interface:gas[ "<燃料数量> "]"

<燃料数量> ::= 参见<合约标记:public:method>

<公开函数名称> ::= <大写字母开头的标识符>

<标识符> ::= <字母> | <标识符> <字母数字串>

<大写字母开头的标识符> ::= <大写字母> | <大写字母开头的标识符> <字母数字串>

<字母数字串> ::= <字母> | <十进制数字> | <字母数字串> <字母> | <字母数字串> <十进制数字>

<字母> ::= "\_" | <小写字母> | <大写字母>

<小写字母> ::= "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" |

"m" | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" |

"y" | "z"

<大写字母> ::= "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L" |

"M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" |

"Y" | "Z"

<十进制数字> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | 8 | "9"

<十进制数> ::= <十进制数字> | <十进制数> <十进制数字>

<元数据类型> := <golang内置类型> | <SDK内置类型> | <自定义数据结构>

<golang内置类型> ::= "int" | "int8" | "int16" | "int32" | "int64" |

"uint" | "uint8" | "uint16" | "uint32" | "uint64" |

"bool" | "string" | "byte"

<SDK内置类型> ::= <Address> | <HexBytes> | <Hash> | <PubKey> | <Number>

<Address> ::= "types.Address"

<HexBytes> ::= "types.HexBytes"

<Hash> ::= "types.Hash"

<PubKey> ::= "types.PubKey"

<Number> ::= "bn.Number"

<自定义数据结构> ::= "type " <标识符> " struct {"

{<变量名称> ["\*"] <变量类型>}

"}"

<数组类型> ::= <数组维度定义> [<数组维度定义>] <元数据类型>

<数组维度定义> ::= "[" [<十进制数>] "]"

<映射表类型> ::= <映射表类型1> | <映射表类型2>

<映射表类型1> ::= <映射表定义> [<映射表定义>] <元数据类型>

<映射表类型2> ::= <映射表定义> [<映射表定义>] <数组类型>

<映射表定义> ::= "map[" <元数据类型> "]"

### 2.4.2 包结构规范

下面定义合约包结构的**BNF**范式：

<智能合约包结构> ::= <合约定义代码文件> {<合约实现代码文件>} {<合约测试代码文件>}

<合约定义代码文件> ::= <合约代码目录> "/" <golang实现代码文件名>

<合约实现代码文件> ::= <合约代码目录> "/" <golang实现代码文件名>

<合约测试代码文件> ::= <合约代码目录> "/" <golang测试代码文件名>

<合约代码目录> ::= <GOPATH> "/" <合约根目录> "/" <合约名称> "/v" <合约版本> "/" <合约名称>

<GOPATH> ::= golang标准环境变量GOPATH指向的目录

<合约根目录> ::= "src/contract/" <组织ID> "/code"

<组织ID> ::= 参见<合约标记:organization>

<golang实现代码文件名> ::= <合法文件名> ".go"

<golang测试代码文件名> ::= <合法文件名> "\_test.go"

<任意文件名> ::= <合法文件名>|<非法文件名>

<非法文件名> ::= [<合法文件名>] "autogen" [<合法文件名>] | <任意文件名> "\_test"

<合法文件名> ::= 排除<非法文件名>以后所有操作系统认可作为文件名的字符串

注：规范定义中出现的空白符间隔仅为排版需要，不作为规范的一部分。

### 2.4.3 白名单列表

出于安全考虑，gichain智能合约限制为只允许导入行为明确且不会导致不同节点运行结果不一致的代码支撑包。这样的包包含在白名单列表中。

白名单列表中支持的代码包在智能合约中可以放心的使用。

白名单与灰名单的具体内容随智能合约**SDK** 及配套的gichain插件版本升级将会有所不同，详细信息请参见智能合约 **SDK** 及配套的gichain插件的相关文件。

下面是白名单列表：

//以下为可用的 golang 标识包

bytes

container/heap

container/list

container/ring

crypto

crypto/aes

crypto/cipher

crypto/des

crypto/hmac

crypto/md5

crypto/rc4

crypto/sha1

crypto/sha256

crypto/sha512

encoding

encoding/ascii85

encoding/asn1

encoding/base32

encoding/base64

encoding/binary

encoding/csv

encoding/gob

encoding/hex

encoding/json

encoding/pem

encoding/xml

errors

fmt

hash

hash/adler32

hash/crc32

hash/crc64

hash/fnv

index/suffixarray

math

math/big

math/bits

math/cmplx

reflect

regexp

regexp/syntax

sort

strconv

strings

unicode

unicode/utf16

unicode/utf8

//以下为 SDK 提供的标准包

blockchain/smcsdk/sdk