腾讯云区块链服务平台 TBaaS

开发指南

腾讯云

文档目录

开发指南
长安链 · ChainMaker
合约开发
智能合约简介

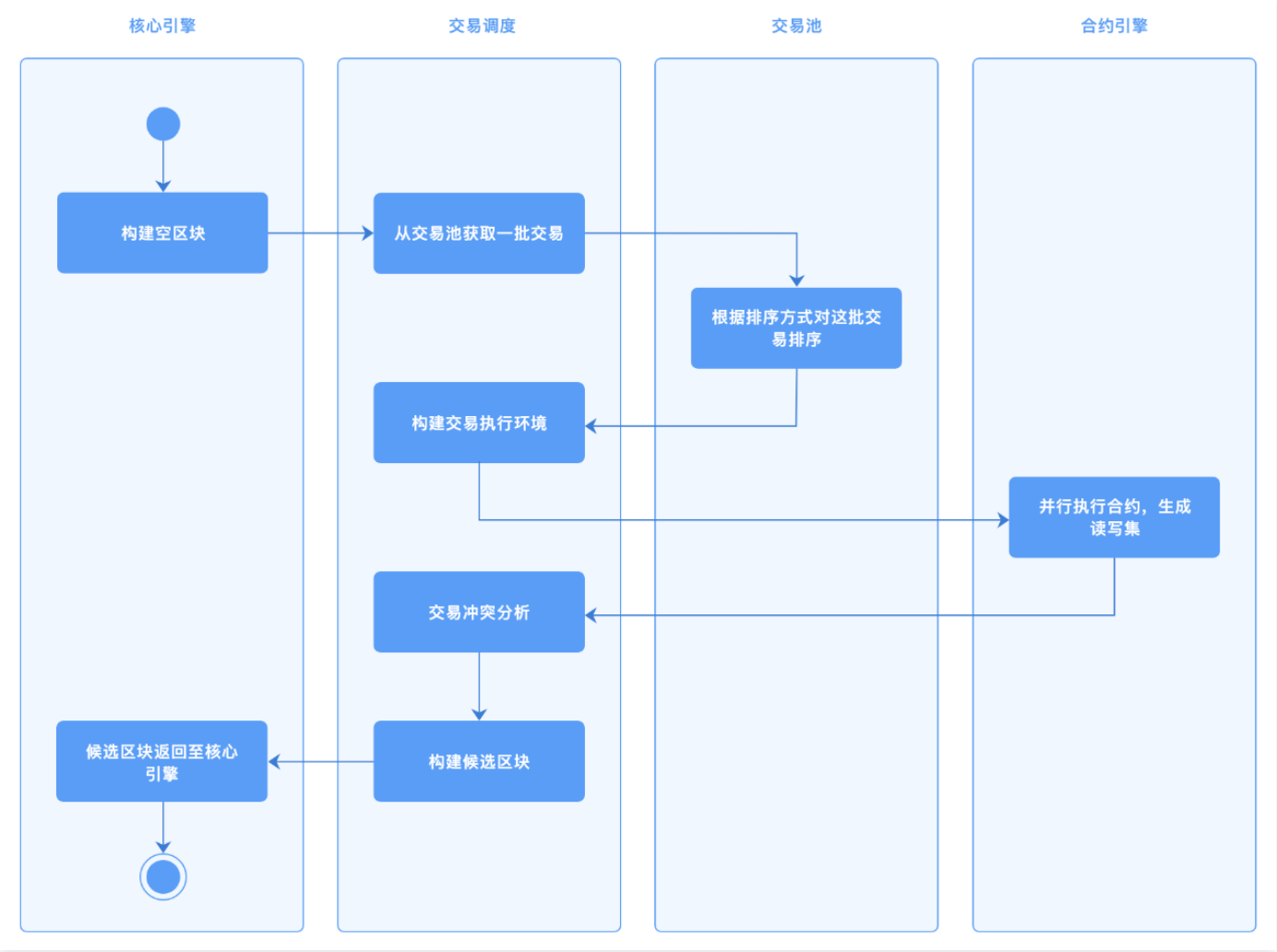
最近更新时间：2022-05-27 15:16:38

智能合约介绍

智能合约是一种计算机程序或交易协议，记录了交易条款信息、事件、行为，旨在减少对可信中间人的需求、仲裁和执行成本。在 ChainMaker 上，用户可以通过高级语言（C++、Go、Rust、Solidty）来编写智能合约，经过编译后，以 WASM、EVM 字节码的形式存储在区块链中。用户可以通过发送交易来触发执行智能合约中的代码。

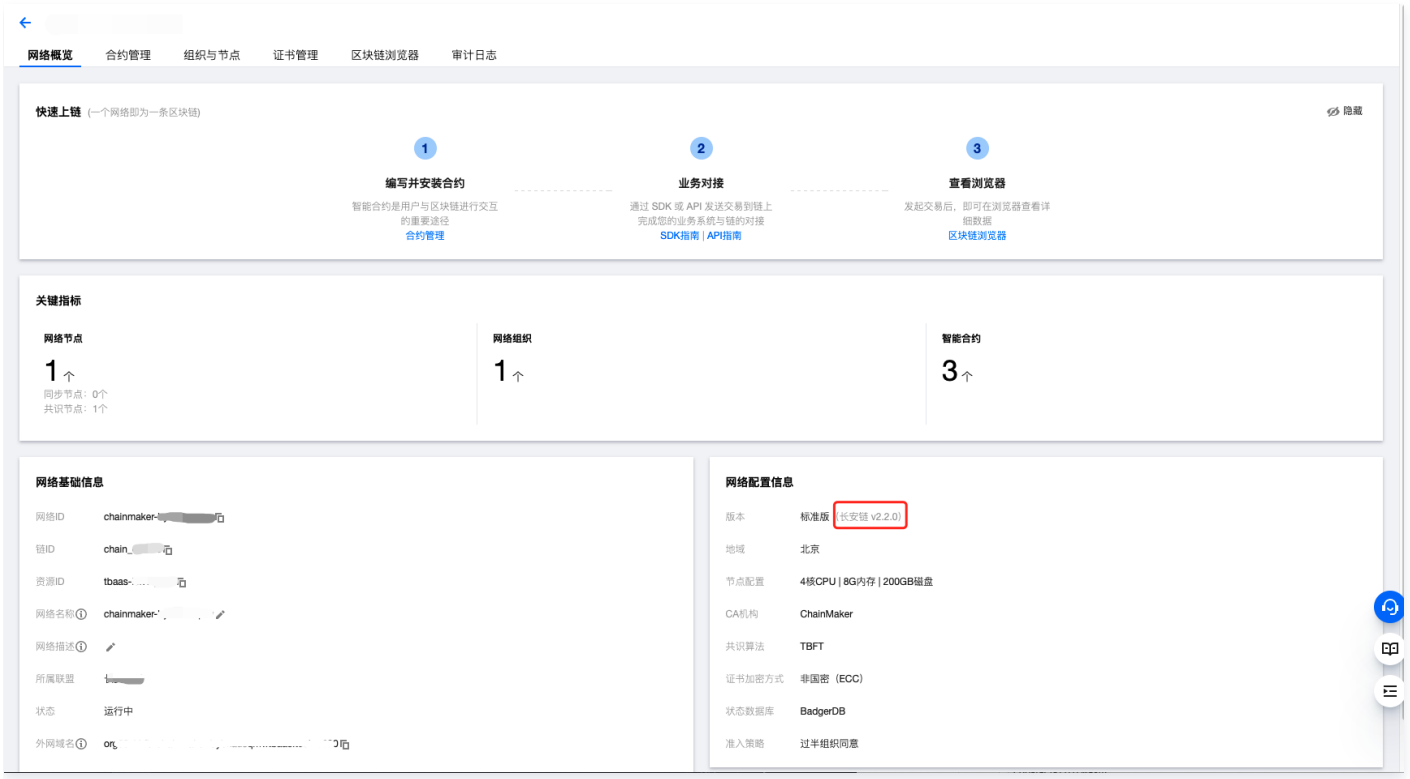
虚拟机为智能合约提供计算资源和运行容器。每个虚拟机都运行在隔离的环境中，确保资源访问安全性，只能修改属于该合约自身的状态记录。智能合约需要有执行终止条件，以限制对资源的消耗；终止条件可以是按照时间、指令数量、指令执行代价（类似 ETH gas）等方式。

长安链支持多种智能合约编程语言和虚拟机，为虚拟机提供统一的数据访问和密码算法访问接口。当一批交易通过调度器被发送至虚拟机时，虚拟机将解析交易中的智能合约调用参数，并且在运行时，通过数据访问接口获取运行时必要的数据，最后执行生成交易的读写集、交易执行结果和交易执行的日志信息。



长安链网络版本

不同版本的长安链网络在智能合约的编译、SDK 等方面略有差别，可以从区块链网络概览页右下角的**网络配置信息**中查看版本，并查看对应版本的智能合约开发文档。



字节码

长安链目前在软件上支持的虚拟机字节码包括两类：WASM（WebAssembly）和 EVM 字节码。

- WebAssembly 有一套完整的语义，实际上 WASM 是体积小且加载快的二进制格式，其目标就是充分发挥硬件能力以达到原生执行效率。WebAssembly 设计了一个非常规整的文本格式用来开发、调试、测试、优化。
- EVM 字节码是最初运用在以太坊上的一种虚拟机字节码，目前已经被广泛的运用在许多区块链平台上，有相对比较成熟的开发工具支持。

智能合约 SDK

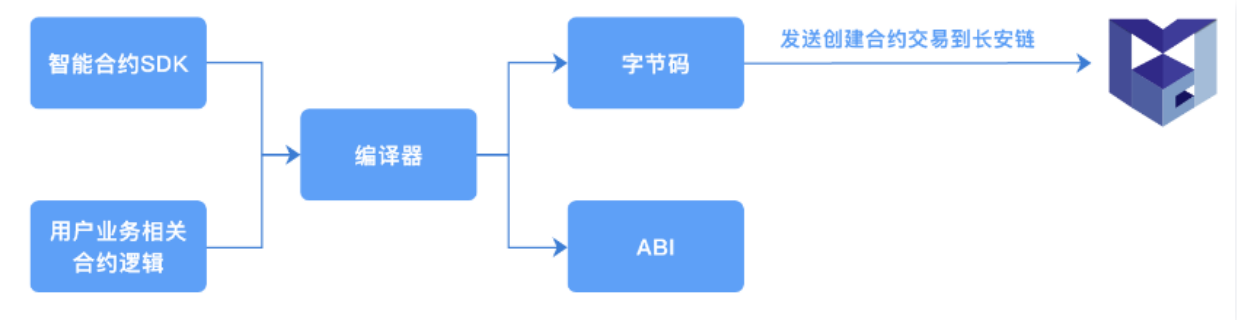
用户通过高级语言编写的智能合约一般情况而言，都需要存取区块链上的数据、API 支持，ChainMaker 为不同的高级语言提供了不同的 SDK。当然，这些 SDK 提供的基本能力是相同的，包括读取数据、写入数据、查询区块链的一些状态等。
不同语言的 SDK 受限于语言本身特性和编译器的支撑能力，例如 Go 语言支持函数同时返回多个数据，而 tinygo 编译器对垃圾回收支持存在缺陷，加上区块链系统本身为智能合约提供的运行内存大小受限、调用栈深度受限，用户编写合约时，需要注意这些特性。
目前 ChainMaker 已经支持的智能合约开发 SDK 包括 Rust、Go、C++。

智能合约生命周期管理

合约部署

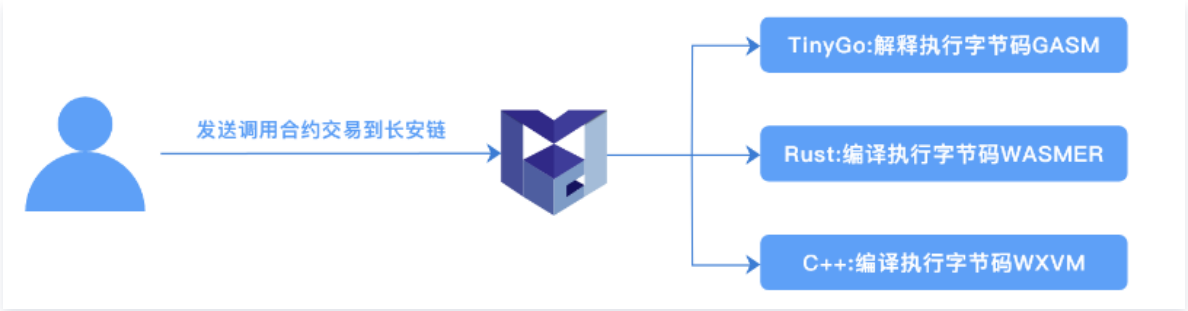
用户编写完成智能合约后，经过编译器编译为字节码，需要通过发送交易的形式部署到区块链上。发送的交易将被共识节点和同步节点接收和处理，在校验完成各项参数后，字节码将被存储在区块链数据库中。
在校验参数的过程中，如果下列校验出错，将把执行的错误信息记录在交易的执行结果中：

- 同一条链上不允许存在重名的合约
- 字节码不能为空
- 指定的智能合约执行引擎必须有效
- 版本信息不能为空



随后将调用执行合约的初始化方法：

- 对于 WASM 而言，将调用 init_contract() 方法，用户必须提供导出的 init_contract() 方法。



- 对于 EVM 而言，将调用构造方法。

合约升级

ChainMaker 支持对基于 WASM 和 EVM 的字节码进行升级。

- 对于 WASM，将调用 **upgrade()** 方法，用户必须提供导出的 **upgrade()** 方法。
- 对于 EVM，并不会调用任何方法，仅更新字节码。

合约升级也同样需要校验参数，如果下列校验出错，将把执行的错误信息记录在交易的执行结果中：

- 合约必须已经被部署成功
- 字节码不能为空
- 版本信息不能为空

合约注销（废止）

用户也可以注销已经被安装的合约，合约一旦被注销，将永远无法再次对合约发起任何操作。

智能合约事件

智能合约事件（contract event）是合约虚拟机中提供的一种智能合约向客户端发送通知消息的功能。当一笔交易触发了合约事件时，事件相关数据会进行广播并记录在区块当中。

- 事件的发送：用户可以在合约函数中指定合约事件 topic 和对应的合约事件数据，当一笔交易调用了包含合约事件的函数即可触发合约事件，ChainMaker 会向指定的 topic 发送事件数据，从而进行合约事件发送。
- 事件的订阅：用户可以使用 ChainMaker SDK 进行合约事件的订阅，如果用户对指定 topic 进行了订阅，当合约事件触发后，用户会收到对应 topic 的合约事件数据。
- 事件的存储：合约事件功能支持用户可配置存储，目前支持 mysql 的可配置存储。

v2.2.1 DockerGo 智能合约开发

最近更新时间：2022-06-28 17:02:15

本章节主要描述使用 Go 进行 ChainMaker 合约编写的方法，主要面向于使用 Go 进行 ChainMaker 的合约开发的开发者。Docker-go 合约运行在独立的 Docker VM 容器中，与长安链节点程序通过 UNIX Domain Socket 或者 TCP 通信。Docker-go 合约需在 Linux 环境下进行编译。

> ⚠ 注意
> 安装 DockerGo 合约时，合约名称必须跟编译合约时使用的合约名保持一致。

使用 Docker 镜像进行合约开发

ChainMaker 官方已经将容器发布至 docker hub 。

1. 拉取镜像

```
docker pull chainmakerofficial/chainmaker-docker-go-contract:v2.2.1
```

请指定您本机的工作目录 $WORK_DIR，例如 /data/workspace/contract，挂载到 docker 容器中以方便后续进行必要的一些文件拷贝。

```
docker run -it --name chainmaker-docker-go-contract -v $WORK_DIR:/home chainmakerofficial/chainmaker-
docker-go-contract:v2.2.1 bash
```

2. 编译合约，压缩合约

```
$ cd /home
$ tar xvf /data/contract_docker_go_template.tar.gz
$ cd contract_docker_go
$ ./build.sh
please input contract name, contract name should be same as name in tx:
<contract_name> #此处contract_name必须和交易中的合约名一致
please input zip file:
<zip_file_name> #建议与contract_name保持一致（不包含文件后缀）
```

编译、压缩好的文件位置如下：

```
/home/contract_docker_go/<contract_name>.7z
```

`<contract_name>.7z` 文件可在 TBaaS 控制台 上传并部署。

用户使用 Go（DockerGo）编写智能合约后，可以把源代码更新到 `main.go` 文件中并重新编译，可得到新的智能合约的压缩文件，并前往 TBaaS 控制台 上传并部署。更多关于使用 Go（DockerGo）开发长安链智能合约的详情，可参考长安链官网 使用 Go（DockerGo）进行智能合约开发 。

合约 API 列表

最近更新时间：2022-10-17 15:01:03

ChainMaker Go (DockerGo) 语言版本智能合约有丰富的 API 接口，供用户在撰写智能合约的时候与链进行交互，代码实现详情可以参考 API 接口代码实现 。

从逻辑方面划分，可将 API 划分为以下类型：

交易信息提取

接口	说明
GetCreatorOrgId() (string, error)	获取合约创建者所属组织ID
GetCreatorRole() (string, error)	获取合约创建者角色
GetCreatorPk() (string, error)	获取合约创建者公钥
GetSenderOrgId() (string, error)	获取交易发起者所属组织ID
GetSenderRole() (string, error)	获取交易发起者角色
GetTxId() (string, error)	获取交易ID
GetSenderPk() (string, error)	获取交易发起者公钥
GetBlockHeight() (int, error)	获取当前区块高度

账本交互

接口	说明
GetState(key string, field string) (string, error)	获取合约账户信息。该接口可从链上获取类别 “key” 下属性名为 “field” 的状态信息。
GetStateFromKey(key string) ([]byte, error)	获取合约账户信息。该接口可以从链上获取类别为key的状态信息
PutState(key string, field string, value string) error	写入合约账户信息。该接口可把类别 “key” 下属性名为 “filed” 的状态更新到链上。更新成功返回0，失败则返回1。
PutStateFromKey(key string, value string) error	写入合约账户信息。
DelState(key string, field string) error	删除合约账户信息。该接口可把类别 “key” 下属性名为 “name” 的状态从链上删除。
CallContract(contractName string, contractVersion string, args map[string][]byte) protogo.Response	跨合约调用。

参数处理

接口	说明
GetArgs() map[string][]byte	该接口将解析出的参数返还给用户。

其他辅助类

接口	说明
Log(message string)	该接口可记录事件日志。
EmitEvent(topic string, data []string)	发送合约事件
NewIterator(startKey string, limitKey string) (ResultSetKV, error)	新建key范围迭代器，key前闭后开，即：startKey <= dbkey < limitKey

NewIteratorWithField(key string, startField string, limitField string) (ResultSetKV, error)	新建field范围迭代器，key需相同，field前闭后开，即：key = dbdbkey and startField <= dbfield < limitField
NewIteratorPrefixWithKey(key string) (ResultSetKV, error)	新建指定key前缀匹配迭代器，key需前缀一致，即dbkey.startWith(key)
NewIteratorPrefixWithKeyField(key string, field string) (ResultSetKV, error)	新建指定field前缀匹配迭代器，key需相同，field前缀一致，即dbkey = key and dbfield.startWith(field)

合约示例

最近更新时间：2023-02-22 16:04:26

智能合约构成

ChainMaker Go (DockerGo) 语言的智能合约代码主要由以下接口构成：

```
package main

// sdk 代码中，有且仅有一个 main() 方法
func main() {
   // main() 方法中，下面的代码为必须代码，不建议修改 main() 方法当中的代码
   // 其中，TestContract 为用户实现合约的具体名称
    err := shim.Start(new(TestContract))
    if err != nil {
        log.Fatal(err)
    }
}

// 合约结构体，合约名称需要写入 main() 方法当中
type TestContract struct {
}

// 合约必须实现下面两个方法：
// InitContract(stub shim.CMStubInterface) protogo.Response
// InvokeContract(stub shim.CMStubInterface) protogo.Response

// 用于合约的部署和升级
// @param stub: 合约接口
// @return:     合约返回结果，包括 Success 和 Error
func (t *TestContract) InitContract(stub shim.CMStubInterface) protogo.Response {

    return shim.Success([]byte("Init Success"))

}

// 用于合约的调用
// @param stub: 合约接口
// @return:     合约返回结果，包括 Success 和 Error
func (t *TestContract) InvokeContract(stub shim.CMStubInterface) protogo.Response {

    return shim.Success([]byte("Invoke Success"))

}
```

代码入口

代码入口包名必须为 main。

```
package main

// sdk 代码中，有且仅有一个 main() 方法
func main() {
   // main() 方法中，下面的代码为必须代码，不建议修改 main() 方法当中的代码
   // 其中，TestContract 为用户实现合约的具体名称
    err := shim.Start(new(TestContract))
    if err != nil {
        log.Fatal(err)
```

```
    }
}
```

智能合约示例

存证合约示例

1. 存储文件哈希、文件名称、时间。
2. 通过文件哈希查询该条记录。

> ⚠ 注意
> - 实现合约接口的结构体不能包含任何字段，例如存证合约中的 FactContract；
> - 合约中不能定义全局变量；
>   因为在一个 vm-docker-go 合约实例的生命周期中，可能会处理多笔交易，这些交易共用全局变量和一个 Contract 接口实例（该示例中的 FactContract），不同交易之间会相互产生不可控的影响，导致合约业务不能正常运行。

```go
package main

import (
    "encoding/json"
    "log"
    "strconv"

    "chainmaker.org/chainmaker-contract-sdk-docker-go/pb/protogo"
    "chainmaker.org/chainmaker-contract-sdk-docker-go/shim"
)

type FactContract struct {
}

// 存证对象
type Fact struct {
    FileHash string `json:"FileHash"`
    FileName string `json:"FileName"`
    Time     int32  `json:"time"`
}

// 新建存证对象
func NewFact(FileHash string, FileName string, time int32) *Fact {
    fact := &Fact{
        FileHash: FileHash,
        FileName: FileName,
        Time:     time,
    }
    return fact
}

func (f *FactContract) InitContract(stub shim.CMStubInterface) protogo.Response {

    return shim.Success([]byte("Init Success"))

}

func (f *FactContract) InvokeContract(stub shim.CMStubInterface) protogo.Response {

    // 获取参数
    method := string(stub.GetArgs()["method"])
```

```
    switch method {
    case "save":
        return f.save(stub)
    case "findByFileHash":
        return f.findByFileHash(stub)
    default:
        return shim.Error("invalid method")
    }

}

func (f *FactContract) save(stub shim.CMStubInterface) protogo.Response {
    params := stub.GetArgs()

    // 获取参数
    fileHash := string(params["file_hash"])
    fileName := string(params["file_name"])
    timeStr := string(params["time"])
    time, err := strconv.Atoi(timeStr)
    if err != nil {
        msg := "time is [" + timeStr + "] not int"
        stub.Log(msg)
        return shim.Error(msg)
    }

    // 构建结构体
    fact := NewFact(fileHash, fileName, int32(time))

    // 序列化
    factBytes, _ := json.Marshal(fact)

    // 发送事件
    stub.EmitEvent("topic_vx", []string{fact.FileHash, fact.FileName})

    // 存储数据
    err = stub.PutStateByte("fact_bytes", fact.FileHash, factBytes)
    if err != nil {
        return shim.Error("fail to save fact bytes")
    }

    // 记录日志
    stub.Log("[save] FileHash=" + fact.FileHash)
    stub.Log("[save] FileName=" + fact.FileName)

    // 返回结果
    return shim.Success([]byte(fact.FileName + fact.FileHash))

}

func (f *FactContract) findByFileHash(stub shim.CMStubInterface) protogo.Response {
    // 获取参数
    FileHash := string(stub.GetArgs()["file_hash"])

    // 查询结果
    result, err := stub.GetStateByte("fact_bytes", FileHash)
    if err != nil {
        return shim.Error("failed to call get_state")
    }

    // 反序列化
```

```
    var fact Fact
    _ = json.Unmarshal(result, &fact)

    // 记录日志
    stub.Log("[find_by_file_hash] FileHash=" + fact.FileHash)
    stub.Log("[find_by_file_hash] FileName=" + fact.FileName)

    // 返回结果
    return shim.Success(result)
}

func main() {

    err := shim.Start(new(FactContract))
    if err != nil {
        log.Fatal(err)
    }
}
```

存证合约代码说明

参数名称	描述
InitContract	合约的初始化函数，在合约部署时被调用，在本合约中为空。
save	save 函数实现将文件相关信息记录到链上的功能。步骤如下： 1. save 函数先通过 [参数处理]API 接口 GetArgs 函数拿到时间，文件哈希和文件名字等信息。 2. 构造 Fact 结构，序列化为 byte 数据；且当序列化错误时调用 [其他辅助类]API 接口 Log 函数记录相应日志。 3. 通过调用 [账本交互]API 接口 PutStateByte 函数将数据记录到链上。 4. 最后通过调用 [其他辅助类]API 接口 Success 函数将操作结果记录到链上。
findByFileHash	通过文件哈希查询该条记录。步骤如下： 1. findByFileHash 通过 [参数处理]API 接口 GetArgs 函数拿到要查找的文件的文件哈希。 2. 通过 [账本交互]API 接口 GetStateByte 函数获取文件的信息；若失败则通过 [其他辅助类]API 接口 Error 函数将操作结果记录到链上，否则，通过 [其他辅助类]API 接口 Log 函数和 Success 函数记录相关日志和返回结果。

Solidity
智能合约开发

最近更新时间：2022-06-22 16:19:59

本章节主要描述使用 Solidity 进行 ChainMaker 合约编写的方法，主要面向于使用 Solidity 进行 ChainMaker 的合约开发的开发者。

> **注意**
> 安装 Solidity 合约时，需将 .bin 和 .abi 两个合约文件一起上传。

使用 Docker 镜像进行合约开发

ChainMaker 官方已经将容器发布至 docker hub 。

1. 拉取镜像
   代码示例如下：

```
docker pull chainmakerofficial/chainmaker-solidity-contract:2.0.0
```

   请指定您本机的工作目录 $WORK_DIR，例如 /data/workspace/contract，挂载到 docker 容器中以方便后续进行必要的一些文件拷贝。

```
docker run -it --name chainmaker-solidity-contract -v $WORK_DIR:/home chainmakerofficial/chainmaker-
solidity-contract:2.0.0 bash
# 或者先后台启动
docker run -d  --name chainmaker-solidity-contract -v $WORK_DIR:/home chainmakerofficial/chainmaker-
solidity-contract:2.0.0 bash -c "while true; do echo hello world; sleep 5;done"
# 再进入容器
docker exec -it chainmaker-solidity-contract bash
```

2. 编译合约
   代码示例如下：

```
cd /home/
# 解压缩合约SDK源码
tar xvf /data/contract_solidity_template.tar.gz
cd contract_solidity
# 编译token.sol合约
solc --abi --bin --hashes --overwrite -o . token.sol
```

   solc为编译命令， -abi选项指示生成abi文件，-bin指示生成字节码文件， -hashes指示生成函数签名文件， -overwrite指示如果生成文件已存在则覆盖， -o 指示编译生成的文件存放的目录。
   生成合约的字节码文件路径如下：

```
/home/contract_solidity/Token.bin
```

   `Token.bin` 文件可在 TBaaS 控制台 上传并部署。
3. 合约开发框架描述
   解压缩 contract_solidity_template.tar.gz 后，文件描述如下：

```
/home/contract_solidity# ls -l
total 4
-rw-rw-r-- 1 1000 1000 2816 Apr 29  2021 token.sol              # token合约
```

   用户使用 Solidity 编写智能合约后，可以把源代码更新到 `token.sol` 文件中并重新编译，可得到新的智能合约的字节码，并前往 TBaaS 控制台 上传并部署。更多关于使用 Solidity 开发长安链智能合约的详情，可参考长安链官网 使用 Solidity 进行智能合约开发 。

合约 API 列表

最近更新时间：2022-10-17 15:05:19

ChainMaker Solidity 语言版本智能合约有丰富的 API 接口，供用户在撰写智能合约的时候与链进行交互，代码实现详情可以参考 API 接口代码实现 。
从逻辑方面划分，可将 API 划分为以下类型：

交易信息提取

接口	说明
msg.data -> bytes	获取调用合约的完整数据
msg.sender -> address	获取消息发送者地址
tx.origin -> address	获取交易发送者地址
tx.gasprice -> uint	获取交易的gas价格

账本交互

接口	说明
blockhash(uint blockNumber) -> bytes32	获取指定区块高度的哈希值
block.gaslimit -> uint	获取当前区块的 gas 限制
block.number -> uint	获取当前区块的高度
block.timestamp -> uint	获取当前区块的时间戳

异常处理

接口	说明
assert(bool condition)	断言给定条件是否成立，如果不满足条件，则状态更改恢复
require(bool condition)	如果不满足条件，则返回
require(bool condition, string memory message)	如果不满足条件，则返回，并提供错误消息
revert()	中止执行并还原状态更改
revert(string memory reason)	中止执行并还原状态更改，并提供错误消息

数学和密码函数类

接口	说明
addmod(uint x, uint y, uint k) -> uint	计算(x+y)%k
mulmod(uint x, uint y, uint k) -> uint	计算(x*y)%k
keccak256(bytes memory) -> bytes32	计算给定输入的 Keccak-256 哈希
sha256(bytes memory) -> bytes32	计算给定输入的 sha256 哈希
ripemd160(bytes memory) -> bytes32	计算给定输入的 RIPEMD-160 哈希
ecrecover(bytes32 hash, uint8 v, bytes32 r, bytes32 s) -> address	从椭圆曲线签名中恢复与公钥关联的地址

ChainMake Solidity 语言版本智能合约完全兼容 EVM，Solidity 的具体使用详情可参见 Solidity 官方文档 。

合约调用

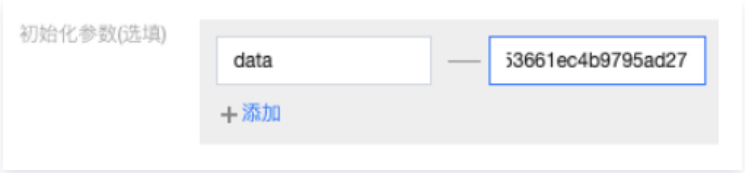
最近更新时间：2022-06-20 14:57:06

合约参数格式

当前 TBaaS 控制台暂支持两种 Solidity 合约参数的格式：**原始数据**、**ABI编码**。在安装或调用 Solidity 合约时，合约初始化参数、合约调用参数可以使用以上两种格式的任意一种。

- 当使用**原始数据**格式时，参数的 key 为参数名称，参数的 value 为参数值，如下图所示：

- 当使用**ABI编码**格式时，参数的 key 是一个固定的字符串 data，参数的 value 为 hex 格式的 ABI 编码，如下图所示：



长安链证书转换 EVM 地址

ChainMake Solidity 语言版本智能合约完全兼容 EVM，更多长安链证书与 EVM 地址的转换详情可参见 EVM 地址说明 。
在 证书管理 界面申请证书后，可根据获得的用户证书文件 user_sign.crt 获取该用户的 EVM 地址，代码示例如下：

```
import (
    "encoding/hex"
    "encoding/pem"
    "fmt"
    "io/ioutil"

    "chainmaker.org/chainmaker/common/v2/crypto/x509"
    "chainmaker.org/chainmaker/common/v2/evmutils"
    "github.com/ethereum/go-ethereum/accounts/abi"
)

func MakeAddrAndSkiFromCrtFilePath() {
    crtFilePath := "./user_sign.crt"

    crtBytes, err := ioutil.ReadFile(crtFilePath)
    if err != nil {
        fmt.Printf("fail to read the crt file:%v", err)
    }

    blockCrt, _ := pem.Decode(crtBytes)
    crt, err := x509.ParseCertificate(blockCrt.Bytes)
    if err != nil {
        fmt.Printf("fail to parse certificate:%v", err)
    }

    ski := hex.EncodeToString(crt.SubjectKeyId)
    addrInt, err := evmutils.MakeAddressFromHex(ski)
    if err != nil {
    fmt.Printf("fail to make address from hex:%v", err)
    }
```

```
    // 证书 SKI
    fmt.Printf("clientAddrSki: %s\n", ski)
    // EVM 地址（十进制）
    fmt.Printf("clientAddrInt: %s\n", addrInt.String())
    // EVM 地址
    fmt.Printf("clientEthAddr: 0x%x\n", addrInt.AsStringKey())
}
```

成功运行可以查看到如下图输出：

```
clientAddrSki: 65176a125555576d9a97b40a9a39a637f19dd75e5cc0043ab109419de32e6e64
clientAddrInt: 902584883230384040201055326396503673974016209567
clientEthAddr: 0x9e194e526ff154807d82c09cbb65e709bd48729f
```

ABI 编码示例

ChainMake Solidity 语言版本智能合约完全兼容 EVM，更多 ABI 编码的详情可参见 Solidity 官方文档 。

合约初始化

以 Token 合约为例，对合约初始化参数进行 ABI 编码。代码示例如下：

```
import (
    "encoding/hex"
    "fmt"
    "io/ioutil"
    "math/big"
    "strings"

    "chainmaker.org/chainmaker/common/v2/evmutils"
    "github.com/ethereum/go-ethereum/accounts/abi"
)

const (
    // 编译合约后获取的 abi 文件的路径
    tokenABIPath = "./testdata/token-evm-demo/token.abi"
    // 合约安装时调用 constructor ，调用方法设为空字符串
    function = ""
    // 入参 1：发行 EVM 地址，可根据用户证书转换取得
    clientAddr = "0x89f4090e315621696d6936453661ec4b9795ad27"
)

func testUserContractTokenEVMConstructor () {
    abiJson, err := ioutil.ReadFile (tokenABIPath)
    if err != nil {
        fmt.Printf ("fail to read the abi file:%v", err)
    }

    myAbi, err := abi.JSON (strings.NewReader (string (abiJson)))
    if err != nil {
        fmt.Printf ("fail to get abi object:%v", err)
    }

    addr := evmutils.BigToAddress (evmutils.FromHexString (clientAddr [2:]))

    dataByte, err := myAbi.Pack (function, addr)
    if err != nil {
        fmt.Printf ("fail to pack contract input:%v", err)
    }
```

```
    data := hex.EncodeToString (dataByte)

    pairs := map [string] string {
        "data": data,
    }

    // 编码后的参数
    fmt.Printf ("FuncParam %v\n", pairs)
}
```

输出结果如下：

```
FuncParam map [data:000000000000000000000000089f4090e315621696d6936453661ec4b9795ad27]
```

通过 TBaaS 控制台 安装合约 并填写对应初始化参数：

- key 取值：data
- value 取值：000000000000000000000000089f4090e315621696d6936453661ec4b9795ad27

合约调用

以 Token 合约为例，对 transfer 函数的函数名及调用参数进行 ABI 编码。代码示例如下：

```
import (
    "encoding/hex"
    "fmt"
    "io/ioutil"
    "math/big"
    "strings"

    "chainmaker.org/chainmaker/common/v2/evmutils"
    "github.com/ethereum/go-ethereum/accounts/abi"
)

const (
    // 编译合约后获取的 abi 文件的路径
    tokenABIPath = "./testdata/token-evm-demo/token.abi"
    // 调用方法
    function = "transfer"
    // 入参 1：转账 EVM 地址，可根据用户证书转换取得
    clientAddr = "0xa55f1e0cb68b0cc589906078237094bdb9715bfd"
    // 入参 2：转账金额
```

```
    amount = 200
)

func testUserContractTokenEVMTransfer () {
    abiJson, err := ioutil.ReadFile (tokenABIPath)
    if err != nil {
        fmt.Printf ("fail to read the abi file:%v", err)
    }

    myAbi, err := abi.JSON (strings.NewReader (string (abiJson)))
    if err != nil {
        fmt.Printf ("fail to get abi object:%v", err)
    }

    addr := evmutils.BigToAddress (evmutils.FromHexString (clientAddr [2:]))

    dataByte, err := myAbi.Pack (function, addr, big.NewInt (amount))
    if err != nil {
        fmt.Printf ("fail to pack contract input:%v", err)
    }

    data := hex.EncodeToString (dataByte)
    method := data [0:8]

    pairs := map [string] string {
        "data": data,
    }

    // 编码后的函数名
    fmt.Printf ("FuncName: %s\n", method)
    // 编码后的参数
    fmt.Printf ("FuncParam %v\n", pairs)
}
```

输出结果如下：

```
FuncName: a9059cbb
FuncParam map
[data:a9059cbb000000000000000000000000a55f1e0cb68b0cc589906078237094bdb9715bfd0000000000000000000000000000
0000000000000000000000000000000000000000c8]
```

长安链 SDK 调用示例

将 ABI 编码后的函数名及调用参数分别作为 method 与 params，代码示例如下：

```
    method := "a9059cbb"
    dataString :=
"a9059cbb000000000000000000000000a55f1e0cb68b0cc589906078237094bdb9715bfd0000000000000000000000000000000000
0000000000000000000000000000000000000000c8"
    params := []*common.KeyValuePair{
        {
            Key:   "data",
            Value: []byte(dataString),
        },
    resp, err := client.InvokeContract ("fact", method, "", params, -1, true)
    if err != nil {
        fmt.Printf ("fail to invoke contract:%v", err)
    }
```

云 API 调用示例

将 ABI 编码后的函数名及调用参数分别填入 FuncName 与 FuncParam 字段，代码示例如下：

```
    action_params = {
        "ClusterId": "chainmaker-txtxtxtxtx",
        "ChainId": "chain_txtxt",
        "ContractName": "fact",
        "FuncName": "a9059cbb",
        "FuncParam": "
{\"data\":\"a9059cbb000000000000000000000000a55f1e0cb68b0cc589906078237094bdb9715bfd0000000000000000000000000000000000000000000000000000000000000000c8\"}",
        "AsyncFlag": 0,
        "Action": "InvokeChainMakerContract",
        "Version": "2018-04-16",
        "Region": "ap-beijing"
    }
    # 实例化一个请求对象，根据调用的接口和实际情况，可以进一步设置请求参数
    params = json.dumps(action_params)
    req = models.InvokeChainMakerContractRequest()
    # 调用 InvokeChainMakerContractRequest 的 from_json_string 方法，使用 params 初始化 req 对象
    req.from_json_string(params)

    # 通过 client 对象调用想要访问的接口，需要传入请求对象
    resp = client.InvokeChainMakerContract(req)
    # 输出 json 格式的字符串回包
    print(resp.to_json_string())
```

合约示例

最近更新时间：2022-06-20 14:57:14

Token 合约代码示例

Token 合约代码示例如下，实现功能 ERC20。

```
/*
SPDX-License-Identifier: Apache-2.0
*/
pragma solidity >0.5.11;
contract Token {

    string public name = "token";      //  token name
    string public symbol = "TK";           //  token symbol
    uint256 public decimals = 6;            //  token digit

    mapping (address => uint256) public balanceOf;
    mapping (address => mapping (address => uint256)) public allowance;

    uint256 public totalSupply = 0;
    bool public stopped = false;

    uint256 constant valueFounder = 100000000000000000;
    address owner = address (0x0);

    modifier isOwner {
        assert (owner == msg.sender);
        _;
    }

    modifier isRunning {
        assert (!stopped);
        _;
    }

    modifier validAddress {
        assert (address (0x0) != msg.sender);
        _;
    }

    constructor (address _addressFounder) {
        owner = msg.sender;
        totalSupply = valueFounder;
        balanceOf [_addressFounder] = valueFounder;

        emit Transfer (address (0x0), _addressFounder, valueFounder);
    }

    function transfer (address _to, uint256 _value) public isRunning validAddress returns (bool success)
{
        require (balanceOf [msg.sender] >= _value);
        require (balanceOf [_to] + _value >= balanceOf [_to]);
        balanceOf [msg.sender] -= _value;
        balanceOf [_to] += _value;
        emit Transfer (msg.sender, _to, _value);
        return true;
    }
```

```
    function transferFrom (address _from, address _to, uint256 _value) public isRunning validAddress
returns (bool success) {
        require (balanceOf [_from] >= _value);
        require (balanceOf [_to] + _value >= balanceOf [_to]);
        require (allowance [_from][msg.sender] >= _value);
        balanceOf [_to] += _value;
        balanceOf [_from] -= _value;
        allowance [_from][msg.sender] -= _value;
        emit Transfer (_from, _to, _value);
        return true;
    }

    function approve (address _spender, uint256 _value) public isRunning validAddress returns (bool
success) {
        require (_value == 0 || allowance [msg.sender][_spender] == 0);
        allowance [msg.sender][_spender] = _value;
        emit Approval (msg.sender, _spender, _value);
        return true;
    }

    function stop () public isOwner {
        stopped = true;
    }

    function start () public isOwner {
        stopped = false;
    }

    function setName (string memory _name) public isOwner {
        name = _name;
    }

    function burn (uint256 _value) public {
        require (balanceOf [msg.sender] >= _value);
        balanceOf [msg.sender] -= _value;
        balanceOf [address (0x0)] += _value;
        emit Transfer (msg.sender, address (0x0), _value);
    }

    event Transfer (address indexed _from, address indexed _to, uint256 _value);
    event Approval (address indexed _owner, address indexed _spender, uint256 _value);
}
```

Token 合约代码说明

参数名称	描述
constructor	合约构造函数，在合约部署时被调用，将 _addressFounder 的余额设置为 valueFounder。
transfer	转账函数，该函数具有两个入参，接收者地址 _to 和转账金额 _value，该函数将调用者 msg.sender 的余额减去 _value，将接收者的余额加上 _value。
transferFrom	转账函数，该函数具有三个入参，转账者地址 _from，接收者地址 _to 和转账金额 _value，该函数将转账者 _from 的余额减去 _value，将接收者的余额加上 _value。
approve	批准函数，该函数具有两个入参，接收者地址 _spender 和 token 数量 _value，该函数将接收者 _spender 可以从调用者 msg.sender 处转出的 token 数量设置为 _value。
stop	停止函数，该函数执行后，transfer，transferFrom 和 approve 函数将不能再被执行。

start	启动函数，该函数执行后，transfer，transferFrom 和 approve 函数将可以再被执行。
setName	设置 token 名字函数，该函数具有一个入参 _name，将 token 的名字设置为 _name。
burn	销毁函数，该函数具有一个入参 _value，将调用者 msg.sender 的余额减去 _value。

Rust
智能合约开发

最近更新时间：2022-06-20 14:57:24

本章节主要描述使用 Rust 进行 ChainMaker 合约编写的方法，主要面向于使用 Rust 进行 ChainMaker 的合约开发的开发者。
Rust 安装及教程请参考 Rust 官网 。

使用 Docker 镜像进行合约开发

ChainMaker 官方已经将容器发布至 docker hub 。

1. 拉取镜像

```
docker pull chainmakerofficial/chainmaker-rust-contract:2.1.0
```

请指定您本机的工作目录 $WORK_DIR，例如 /data/workspace/contract，挂载到 docker 容器中以方便后续进行必要的一些文件拷贝。

```
docker run -it --name chainmaker-rust-contract -v $WORK_DIR:/home chainmakerofficial/chainmaker-rust-
contract:2.1.0 bash
# 或者先后台启动
docker run -d  --name chainmaker-rust-contract -v $WORK_DIR:/home chainmakerofficial/chainmaker-rust-
contract:2.1.0 bash -c "while true; do echo hello world; sleep 5;done"
# 再进入容器
docker exec -it chainmaker-rust-contract bash
```

2. 编译合约

```
cd /home/
tar xvf /data/contract_rust_template.tar.gz
cd contract_rust
make build
```

生成的合约字节码文件位置如下：

```
/home/contract_rust/target/wasm32-unknown-unknown/release/chainmaker_contract.wasm
```

`chainmaker_contract.wasm` 文件可在 TBaaS 控制台 上传并部署。

3. 合约开发框架描述
解压缩 contract_rust_template.tar.gz 后，文件描述如下：

```
chainmaker-contract-sdk-rust$ tree -I target
.
├── Cargo.lock # 依赖版本信息
├── Cargo.toml # 项目配置及依赖，参考：https://rustwasm.github.io/wasm-pack/book/cargo-toml-
configuration.html
├── Makefile    # build一个wasm文件
├── README.md   # 编译环境说明
├── src
|   ├── contract_fact.rs            # 存证示例代码
|   ├── easycodec.rs                # 序列化工具类
|   ├── lib.rs                      # 程序入口
|   ├── sim_context.rs              # 合约SDK主要接口及实现
|   ├── sim_context_bulletproofs.rs # 合约SDK基于bulletproofs的范围证明接口实现
|   ├── sim_context_paillier.rs     # 合约SDK基于paillier的半同态运算接口实现
|   ├── sim_context_rs.rs           # 合约SDK sql接口实现
```

```
└── vec_box.rs                  # 内存管理类
```

用户使用 Rust 编写智能合约后，可以把源代码更新到 `src/contract_fact.rs` 文件中并重新编译，得到新的智能合约的字节码，并前往 TBaaS 控制台 上传并部署。更多关于使用 Rust 进行开发长安链智能合约的详情，可参考长安链官网 使用 Rust 进行智能合约开发

合约 API 列表

最近更新时间：2022-10-17 15:04:36

ChainMaker Rust 语言版本智能合约有丰富的 API 接口，供用户在撰写智能合约的时候与链进行交互，代码实现详情可以参考 API 接口代码实现 。
从逻辑方面划分，可将 API 划分为以下类型：

交易信息提取

接口	说明
get_creator_org_id(&self) -> String	获取合约创建者所属组织 ID
get_creator_role(&self) -> String	获取合约创建者角色
get_creator_pub_key(&self) -> String	获取合约创建者公钥
get_sender_org_id(&self) -> String	获取交易发起者所属组织 ID
get_sender_role(&self) -> String	获取交易发起者角色
get_tx_id(&self) -> String	获取交易 ID
get_sender_pub_key(&self) -> String	获取交易发起者公钥
get_block_height(&self) -> u64	获取当前区块高度

账本交互

接口	说明
get_state(&self, key: &str, field: &str) -> Result<Vec<u8>, result_code>	获取合约账户信息。该接口可从链上获取类别 “key” 下属性名为 “field” 的状态信息。
get_state_from_key(&self, key: &str) -> Result<Vec<u8>, result_code>	获取合约账户信息。该接口可以从链上获取类别为 key 的状态信息
put_state(&self, key: &str, field: &str, value: &[u8]) -> result_code	写入合约账户信息。该接口可把类别 “key” 下属性名为 “filed” 的状态更新到链上。更新成功返回 0，失败则返回 1。
put_state_from_key(&self, key: &str, value: &[u8]) -> result_code	写入合约账户信息。
delete_state(&self, key: &str, field: &str) -> result_code	删除合约账户信息。该接口可把类别 “key” 下属性名为 “name” 的状态从链上删除。
delete_state_from_key(&self, key: &str) -> result_code;	删除合约账户信息。该接口可把类别 “key” 下属性名为 “name” 的状态从链上删除。
call_contract(&self, contract_name: &str, method: &str, param: EasyCodec) -> Result<Vec<u8>, result_code>;	跨合约调用。

参数处理

接口	说明
args(&self) -> &EasyCodec	该接口调用 getArgsMap() 接口，把 json 格式的数据反序列化，并将解析出的数据返还给用户。
arg(&self, key: &str) -> Result<Vec<u8>, String>	该接口可返回属性名为 “key” 的参数的属性值（输出格式为Result<Vec<u8>, String>）。
arg_as_utf8_str(&self, key: &str) -> String;	该接口可返回属性名为 “key” 的参数的属性值（输出格式为String）。

其他辅助类

接口	说明
ok(&self, value: &[u8]) -> result_code	该接口可记录用户操作成功的信息，并将操作结果记录到链上。
error(&self, body: &str) -> result_code	该接口可记录用户操作失败的信息，并将操作结果记录到链上。
log(&self, msg: &str)	该接口可记录事件日志。查看方式为在链配置的 log.yml 中，开启 vm:debug 即可看到类似：gasm log>> + msg
emit_event(&mut self, topic: &str, data: &Vec<String>) -> result_code	发送合约事件
new_iterator(&self,start_key: &str,limit_key: &str) -> Result<Box<dyn ResultSet>, result_code>	新建 key 范围迭代器，key 前闭后开，即：startKey <= dbkey < limitKey
new_iterator_with_field(&self, key: &str, start_field: &str, limit_field: &str) -> Result<Box<dyn ResultSet>, result_code>	新建 field 范围迭代器，key 需相同，field 前闭后开，即：key = dbdbkey and startField <= dbfield < limitField
new_iterator_prefix_with_key(&self, key: &str) -> Result<Box<dyn ResultSet>, result_code>	新建指定 key 前缀匹配迭代器，key 需前缀一致，即 dbkey.startWith(key)
new_iterator_prefix_with_key_field(&self, key: &str, field: &str) -> Result<Box<dyn ResultSet>, result_code>	新建指定 field 前缀匹配迭代器，key 需相同，field 前缀一致，即 dbkey = key and dbfield.startWith(field)

合约示例

最近更新时间：2023-02-22 16:04:27

智能合约构成

ChainMaker Rust 语言的智能合约代码主要由以下接口构成：

```
/*
Copyright (C) BABEC. All rights reserved.
SPDX-License-Identifier: Apache-2.0
一个 ChainMaker 的 Rust 版本智能合约主要包括以下函数：
*/
use crate::easycodec::*;
use crate::sim_context;
use sim_context::*;

// 安装合约时会执行此方法，必须
#[no_mangle]
pub extern "C" fn init_contract() {
    // 安装时的业务逻辑，内容可为空
    sim_context::log("init_contract");
}

// 升级合约时会执行此方法，必须
#[no_mangle]
pub extern "C" fn upgrade() {
    // 升级时的业务逻辑，内容可为空
    sim_context::log("upgrade");
    let ctx = &mut sim_context::get_sim_context();
    ctx.ok("upgrade success".as_bytes());
}

// 对 SDK 暴露的函数
// 对外暴露 test1 方法，供用户由 SDK 调用
#[no_mangle]
pub extern "C" fn test1() {}
// 对外暴露 test2 方法，供用户由 SDK 调用
#[no_mangle]
pub extern "C" fn test2() {}

// 其他函数，不对外暴露
fn test3() {}
```

智能合约示例

存证合约示例

可实现以下功能：

1. 存储文件哈希、文件名称和时间等信息。
2. 通过文件哈希查询该条记录。

```
use crate::easycodec::*;
use crate::sim_context;
use sim_context::*;

// 安装合约时会执行此方法，必须
#[no_mangle]
```

```
pub extern "C" fn init_contract() {
    // 安装时的业务逻辑，内容可为空
    sim_context::log("init_contract");
}

// 升级合约时会执行此方法，必须
#[no_mangle]
pub extern "C" fn upgrade() {
    // 升级时的业务逻辑，内容可为空
    sim_context::log("upgrade");
    let ctx = &mut sim_context::get_sim_context();
    ctx.ok("upgrade success".as_bytes());
}

struct Fact {
    file_hash: String,
    file_name: String,
    time: i32,
    ec: EasyCodec,
}

impl Fact {
    fn new_fact(file_hash: String, file_name: String, time: i32) -> Fact {
        let mut ec = EasyCodec::new();
        ec.add_string("file_hash", file_hash.as_str());
        ec.add_string("file_name", file_name.as_str());
        ec.add_i32("time", time);
        Fact {
            file_hash,
            file_name,
            time,
            ec,
        }
    }

    fn get_emit_event_data(&self) -> Vec<String> {
        let mut arr: Vec<String> = Vec::new();
        arr.push(self.file_hash.clone());
        arr.push(self.file_name.clone());
        arr.push(self.time.to_string());
        arr
    }

    fn to_json(&self) -> String {
        self.ec.to_json()
    }

    fn marshal(&self) -> Vec<u8> {
        self.ec.marshal()
    }

    fn unmarshal(data: &Vec<u8>) -> Fact {
        let ec = EasyCodec::new_with_bytes(data);
        Fact {
            file_hash: ec.get_string("file_hash").unwrap(),
            file_name: ec.get_string("file_name").unwrap(),
            time: ec.get_i32("time").unwrap(),
            ec,
        }
    }
```

```
}

// save 保存存证数据
#[no_mangle]
pub extern "C" fn save() {
    // 获取上下文
    let ctx = &mut sim_context::get_sim_context();

    // 获取传入参数
    let file_hash = ctx.arg_as_utf8_str("file_hash");
    let file_name = ctx.arg_as_utf8_str("file_name");
    let time_str = ctx.arg_as_utf8_str("time");

    // 构造结构体
    let r_i32 = time_str.parse::<i32>();
    if r_i32.is_err() {
        let msg = format!("time is {:?} not int32 number.", time_str);
        ctx.log(&msg);
        ctx.error(&msg);
        return;
    }
    let time: i32 = r_i32.unwrap();
    let fact = Fact::new_fact(file_hash, file_name, time);

    // 事件
    ctx.emit_event("topic_vx", &fact.get_emit_event_data());

    // 序列化后存储
    ctx.put_state(
        "fact_ec",
        fact.file_hash.as_str(),
        fact.marshal().as_slice(),
    );
}

// find_by_file_hash 根据 file_hash 查询存证数据
#[no_mangle]
pub extern "C" fn find_by_file_hash() {
    // 获取上下文
    let ctx = &mut sim_context::get_sim_context();

    // 获取传入参数
    let file_hash = ctx.arg_as_utf8_str("file_hash");

    // 校验参数
    if file_hash.len() == 0 {
        ctx.log("file_hash is null");
        ctx.ok("".as_bytes());
        return;
    }

    // 查询
    let r = ctx.get_state("fact_ec", &file_hash);

    // 校验返回结果
    if r.is_err() {
        ctx.log("get_state fail");
        ctx.error("get_state fail");
        return;
    }
```

```
    let fact_vec = r.unwrap();
    if fact_vec.len() == 0 {
        ctx.log("None");
        ctx.ok("".as_bytes());
        return;
    }

    // 查询
    let r = ctx.get_state("fact_ec", &file_hash).unwrap();
    let fact = Fact::unmarshal(&r);
    let json_str = fact.to_json();

    // 返回查询结果
    ctx.ok(json_str.as_bytes());
    ctx.log(&json_str);
}
```

存证合约代码说明

参数名称	描述
init_contract	合约的初始化函数，在合约部署时被调用，在本合约中打印了日志"init_contract"。
upgrade	合约升级时调用的函数。
save	save 函数实现将文件哈希和文件名称记录到链上的功能。步骤如下： 1. save 函数先通过 get_sim_context 和 [交易信息提取]API 接口 arg_as_utf8_str 函数拿到时间，文件哈希和文件名字等信息。 2. 构造 Fact 结构体并进行序列化；且当序列化错误时调用 [其他辅助类]API 接口 log 函数和 error 函数记录相应日志。 3. 通过调用 [其他辅助类]API 接口 emit_event 函数发送标识为 topic_vx 的合约事件。 4. 通过调用 [账本交互]API 接口 put_state 函数将文件等信息记录到链上。
find_by_file_hash	通过文件哈希查询该条记录。步骤如下： 1. find_by_file_hash 通过 get_sim_context 和 [参数处理]API 接口 arg_as_utf8_str 函数拿到要查找的文件的文件哈希。 2. 通过 [账本交互]API 接口 get_state 函数获取文件的信息；若失败则通过 [其他辅助类]API 接口 error 函数将操作结果记录到链上，否则，通过 [其他辅助类]API 接口 log 函数和 ok 函数记录相关日志并返回结果。

C++
智能合约开发

最近更新时间：2022-06-20 14:57:45

本章节主要描述使用 C++ 进行 ChainMaker 合约编写的方法，主要面向于使用 C++ 进行 ChainMaker 的合约开发的开发者。

使用 Docker 镜像进行合约开发

ChainMaker 官方已经将容器发布至 docker hub 。

1. 拉取镜像

```
docker pull chainmakerofficial/chainmaker-cpp-contract:2.1.0
```

请指定您本机的工作目录 $WORK_DIR，例如 /data/workspace/contract，挂载到 docker 容器中以方便后续进行必要的一些文件拷贝。

```
docker run -it --name chainmaker-cpp-contract -v $WORK_DIR:/home chainmakerofficial/chainmaker-cpp-
contract:2.1.0 bash
# 或者先后台启动
docker run -d --name chainmaker-cpp-contract -v $WORK_DIR:/home chainmakerofficial/chainmaker-cpp-
contract:2.1.0 bash -c "while true; do echo hello world; sleep 5;done"
# 再进入容器
docker exec -it chainmaker-cpp-contract bash
```

2. 编译合约

```
cd /home/
tar xvf /data/contract_cpp_template.tar.gz
cd contract_cpp
make clean
emmake make
```

生成的合约字节码文件位置如下：

```
/home/contract_cpp/main.wasm
```

`main.wasm` 文件可在 TBaaS 控制台 上传并部署。

3. 合约开发框架描述

解压缩 contract_cpp_template.tar.gz 后，文件描述如下：

- chainmaker
  - basic_iterator.cc：迭代器实现
  - basic_iterator.h：迭代器头文件声明
  - chainmaker.h：sdk 主要接口头文件声明，详情见 SDK API 描述
  - context_impl.cc：与链交互接口实现
  - context_impl.h：与链交互头文件声明
  - contract.cc：合约基础工具类
  - error.h：异常处理类
  - exports.js：编译合约导出函数
  - safemath.h：assert 异常处理
  - syscall.cc：与链交互入口
  - syscall.h：与链交互头文件声明
- pb

- **contract.pb.cc**：与链交互数据协议
- **contract.pb.h**：与链交互数据协议头文件声明

- **main.cc**：用户写合约入口
- **Makefile**：常用 build 命令

用户使用 C++ 编写智能合约后，可以把源代码更新到 `main.cc` 文件中并重新编译，可得到新的智能合约的压缩文件，并前往 TBaaS 控制台 上传并部署。更多关于使用 C++ 开发长安链智能合约的详情，可参考长安链官网 使用 C++ 进行智能合约开发 。

合约 API 列表

最近更新时间：2022-10-17 15:04:16

ChainMaker C++ 语言版本智能合约有丰富的 API 接口，供用户在撰写智能合约的时候与链进行交互，代码实现详情可以参考 API 接口代码实现 。
从逻辑方面划分，可将 API 划分为以下类型：

账本交互

接口	说明
bool get_object(const std::string& key, std::string* value){}	获取key为"key"的值。
bool put_object(const std::string& key, const std::string& value) {}	存储key为"key"的值，注意key长度不允许超过64，且只允许大小写字母、数字、下划线、减号、小数点符号。
bool delete_object(const std::string& key) {}	删除key为"key"的值。
bool call(const std::string &contract, const std::string &method, EasyCodecItems *args, std::string *resp){}	跨合约调用。

参数处理

接口	说明
bool arg(const std::string& name, std::string& value){}	该接口可返回属性名为 “name” 的参数的属性值。需要注意的是通过arg接口返回的参数，全部为字符串，合约开发者有必要将其他数据类型的参数与字符串做转换，包括atoi、itoa、自定义序列化方式等。

其他辅助类

接口	说明
void success(const std::string& body) {}	返回成功的结果
void error(const std::string& body) {}	返回失败结果
void log(const std::string& body) {}	输出日志事件。查看方式为在链配置的log.yml中，开启vm:debug即可看到类似：wxvm log>> + msg
bool emit_event(const std::string &topic, int data_amount, const std::string data, ...)	发送合约事件

合约示例

最近更新时间：2023-02-22 16:04:27

智能合约构成

ChainMaker C++ 语言的智能合约代码主要由以下接口构成：

```
#include "chainmaker/chainmaker.h"

using namespace chainmaker;

class Counter : public Contract {
public:
    void init_contract() {}
    void upgrade() {}
};

// 在创建本合约时，调用一次 init 方法。ChainMaker 不允许用户直接调用该方法。
WASM_EXPORT void init_contract() {
    // 安装时的业务逻辑，可为空

}

// 在升级本合约时，对于每一个升级的版本调用一次 upgrade 方法。ChainMaker 不允许用户直接调用该方法。
WASM_EXPORT void upgrade() {
    // 升级时的业务逻辑，可为空

}

// 对 SDK 暴露的函数
// 对外暴露 test1 方法，供用户由 SDK 调用
WASM_EXPORT void test1() {}
// 对外暴露 test2 方法，供用户由 SDK 调用
WASM_EXPORT void test2() {}
```

智能合约示例

存证合约示例

可实现如下功能：

1. 存储文件哈希、文件名称和时间等信息。
2. 通过文件哈希查询该条记录。

```
#include "chainmaker/chainmaker.h"

using namespace chainmaker;

class Counter : public Contract {
public:
    void init_contract() {}
    void upgrade() {}
    // 保存
    void save() {
        // 获取 SDK 接口上下文
        Context* ctx = context();
        // 定义变量
        std::string time;
```

```
        std::string file_hash;
        std::string file_name;
        std::string tx_id;
        // 获取参数
        ctx->arg("time", time);
        ctx->arg("file_hash", file_hash);
        ctx->arg("file_name", file_name);
        ctx->arg("tx_id", tx_id);
        // 发送合约事件
        // 向 topic:"topic_vx"发送 2 个 event 数据，file_hash,file_name
        ctx->emit_event("topic_vx",2,file_hash.c_str(),file_name.c_str());
        // 存储数据
        ctx->put_object("fact"+ file_hash,  tx_id+" "+time+" "+file_hash+" "+file_name);
        // 记录日志
        ctx->log("call save() result:" + tx_id+" "+time+" "+file_hash+" "+file_name);
        // 返回结果
        ctx->success(tx_id+" "+time+" "+file_hash+" "+file_name);
    }

    // 查询
    void find_by_file_hash() {
        // 获取 SDK 接口上下文
        Context* ctx = context();

        // 获取参数
        std::string file_hash;
        ctx->arg("file_hash", file_hash);

        // 查询数据
        std::string value;
        ctx->get_object("fact"+ file_hash, &value);
        // 记录日志
        ctx->log("call find_by_file_hash()-" + file_hash + ",result:" + value);
        // 返回结果
        ctx->success(value);
    }

};

// 在创建本合约时，调用一次 init 方法。ChainMaker 不允许用户直接调用该方法。
WASM_EXPORT void init_contract() {
    Counter counter;
    counter.init_contract();
}

// 在升级本合约时，对于每一个升级的版本调用一次 upgrade 方法。ChainMaker 不允许用户直接调用该方法。
WASM_EXPORT void upgrade() {
    Counter counter;
    counter.upgrade();
}

WASM_EXPORT void save() {
    Counter counter;
    counter.save();
}

WASM_EXPORT void find_by_file_hash() {
    Counter counter;
    counter.find_by_file_hash();
```

```
}
```

存证合约代码说明

参数名称	描述
init_contract	合约的初始化函数，在合约部署时被调用，在本合约中为空。
upgrade	合约升级时调用的函数，在本合约中为空。
save	save 函数实现将文件哈希和文件名称记录到链上的功能。步骤如下： 1. save 函数先通过 [参数处理]API 接口 arg 函数拿到时间，文件哈希和文件名字等信息。 2. 通过调用 [其他辅助类]API 接口 emit_event 函数发送标识为 topic_vx 的合约事件。 3. 通过调用 [账本交互]API 接口 put_object 函数将文件等信息记录到链上。
find_by_file_hash	通过文件哈希查询该条记录。步骤如下： 1. find_by_file_hash 通过 context 和 [参数处理]API 接口 arg 函数拿到要查找的文件的文件哈希。 2. 通过 [账本交互]API 接口 get_object 函数获取文件的信息，通过 [其他辅助类]API 接口 log 函数和 success 函数记录相关日志并返回结果。

TinyGo
智能合约开发

最近更新时间：2022-06-20 14:58:11

本章节主要描述使用 Go 进行 ChainMaker 合约编写的方法，主要面向于使用 Go 进行 ChainMaker 的合约开发的开发者。为了最小化 wasm 文件尺寸，应使用 TinyGO 编译器。

> ⚠ 注意
> - 当前在 v2.2.1 版本的长安链网络中，仅支持通过 TBaaS 控制台对存量 TinyGo 合约进行升级，不再支持 TinyGo 合约创建。
> - 新用户请使用 Go（DockerGo）进行智能合约开发， Go（TinyGo）仅供老用户进行智能合约升级。

使用 Docker 镜像进行合约开发

ChainMaker 官方已经将容器发布至 docker hub 。

1. 拉取镜像

```
docker pull chainmakerofficial/chainmaker-go-contract:2.1.0
```

请指定您本机的工作目录 $WORK_DIR，例如 /data/workspace/contract，挂载到 docker 容器中以方便后续进行必要的一些文件拷贝。

```
docker run -it --name chainmaker-go-contract -v $WORK_DIR:/home chainmakerofficial/chainmaker-go-
contract:2.1.0 bash
# 或者先后台启动
docker run -d  --name chainmaker-go-contract -v $WORK_DIR:/home chainmakerofficial/chainmaker-go-
contract:2.1.0 bash -c "while true; do echo hello world; sleep 5;done"
# 再进入容器
docker exec -it chainmaker-go-contract bash
```

2. 编译合约

```
cd /home/
# 解压缩合约SDK源码
tar xvf /data/contract_go_template.tar.gz
cd contract_tinygo
# 编译main.go合约
sh build.sh
```

生成合约的字节码文件位置如下：

```
/home/contract_tinygo/main.wasm
```

`main.wasm` 文件可在 TBaaS 控制台 上传并部署。

3. 合约开发框架描述

解压缩 contract_go_template.tar.gz 后，文件描述如下：

```
/home/contract_tinygo# ls -l
total 64
-rw-rw-r-- 1 1000 1000    56 Jul  2 12:45 build.sh             # 编译脚本
-rw-rw-r-- 1 1000 1000  4149 Jul  2 12:44 bulletproofs.go      # 合约SDK基于bulletproofs的范围证明接口实
现
-rw-rw-r-- 1 1000 1000 18871 Jul  2 12:44 chainmaker.go        # 合约SDK主要接口及实现
-rw-rw-r-- 1 1000 1000  4221 Jul  2 12:44 chainmaker_rs.go     # 合约SDK sql接口实现
-rw-rw-r-- 1 1000 1000 11777 May 24 13:27 easycodec.go         # 序列化工具类
```

```
-rw-rw-r-- 1 1000 1000  3585 Jul  2 12:44 main.go              # 存证示例代码
-rwxr-xr-x 1 root root 65122 Jul  6 07:22 main.wasm            # 编译成功后的wasm文件
-rw-rw-r-- 1 1000 1000  1992 Jul  2 12:44 paillier.go          # 合约SDK基于paillier的半同态运算接口实现
```

用户使用 Go（TinyGo）编写智能合约后，可以把源代码更新到 `main.go` 文件中并重新编译，可得到新的智能合约的字节码，并前往 TBaaS 控制台 上传并部署。更多关于使用 Go（TinyGo）开发长安链智能合约的详情，可参考长安链官网 使用 Go（TinyGo）进行智能合约开发 。

合约 API 列表

最近更新时间：2022-10-17 15:06:57

ChainMaker Go（TinyGo）语言版本智能合约有丰富的 API 接口，供用户在撰写智能合约的时候与链进行交互，代码实现详情可以参考 API 接口代码实现。

> **注意**
> - 当前在 v2.2.1 版本的长安链网络中，仅支持通过 TBaaS 控制台对存量 TinyGo 合约进行升级，不再支持 TinyGo 合约创建。
> - 新用户请使用 Go（DockerGo）进行智能合约开发， Go（TinyGo）仅供老用户进行智能合约升级。

从逻辑方面划分，可将 API 划分为以下类型：

交易信息提取

接口	说明
GetCreatorOrgId() (string, ResultCode)	获取合约创建者所属组织ID
GetCreatorRole() (string, ResultCode)	获取合约创建者角色
GetCreatorPk() (string, ResultCode)	获取合约创建者公钥
GetSenderOrgId() (string, ResultCode)	获取交易发起者所属组织ID
GetSenderRole() (string, ResultCode)	获取交易发起者角色
GetTxId() (string, ResultCode)	获取交易ID
GetSenderPk() (string, ResultCode)	获取交易发起者公钥
GetBlockHeight() (string, ResultCode)	获取当前区块高度

账本交互

接口	说明
GetState(key string, field string) (string, ResultCode)	获取合约账户信息。该接口可从链上获取类别 “key” 下属性名为 “field” 的状态信息。
GetStateFromKey(key string) ([]byte, ResultCode)	获取合约账户信息。该接口可以从链上获取类别为key的状态信息
PutState(key string, field string, value string) ResultCode	写入合约账户信息。该接口可把类别 “key” 下属性名为 “filed” 的状态更新到链上。更新成功返回0，失败则返回1。
PutStateFromKey(key string, value string) ResultCode	写入合约账户信息。
DeleteState(key string, field string) ResultCode	删除合约账户信息。该接口可把类别 “key” 下属性名为 “name” 的状态从链上删除。
CallContract(contractName string, method string, param map[string][]byte) ([]byte, ResultCode)	跨合约调用。

参数处理

接口	说明
Args() []*EasyCodecItem{}	该接口调用 getArgsMap() 接口，把 json 格式的数据反序列化，并将解析出的数据返还给用户。
Arg(key string) ([]byte, ResultCode)	该接口可返回属性名为 “key” 的参数的属性值（输出格式为[]byte）。

ArgString(key string) (string, ResultCode)	该接口可返回属性名为 “key” 的参数的属性值（输出格式为String）。

其他辅助类

接口	说明
SuccessResult(msg string)	该接口可记录用户操作成功的信息，并将操作结果记录到链上。
ErrorResult(msg string)	该接口可记录用户操作失败的信息，并将操作结果记录到链上。
LogMessage(msg string)	该接口可记录事件日志。查看方式为在链配置的log.yml中，开启vm:debug即可看到类似：gasm log>> + msg
EmitEvent(topic string, data ...string) ResultCode	发送合约事件
NewIterator(startKey string, limitKey string) (ResultSetKV, ResultCode)	新建key范围迭代器，key前闭后开，即：startKey <= dbkey < limitKey
NewIteratorWithField(key string, startField string, limitField string) (ResultSetKV, ResultCode)	新建field范围迭代器，key需相同，field前闭后开，即：key = dbdbkey and startField <= dbfield < limitField
NewIteratorPrefixWithKey(key string) (ResultSetKV, ResultCode)	新建指定key前缀匹配迭代器，key需前缀一致，即dbkey.startWith(key)
NewIteratorPrefixWithKeyField(key string, field string) (ResultSetKV, ResultCode)	新建指定field前缀匹配迭代器，key需相同，field前缀一致，即dbkey = key and dbfield.startWith(field)

合约示例

最近更新时间：2023-02-22 16:04:27

智能合约构成

ChainMaker Go (TinyGo) 语言的智能合约代码主要由以下接口构成：

> ⚠ 注意
> - 当前在 v2.2.1 版本的长安链网络中，仅支持通过 TBaaS 控制台对存量 TinyGo 合约进行升级，不再支持 TinyGo 合约创建。
> - 新用户请使用 Go（DockerGo）进行智能合约开发， Go（TinyGo）仅供老用户进行智能合约升级。

```
/*
Copyright (C) BABEC. All rights reserved.
SPDX-License-Identifier: Apache-2.0
一个 ChainMaker 的 TinyGO 版本智能合约主要包括以下函数：
*/
package main
// 安装合约时会执行此方法，必须
//export init_contract
func initContract() {
    // 此处可写安装合约的初始化逻辑
}

// 升级合约时会执行此方法，必须
//export upgrade
func upgrade() {
    // 此处可写升级合约的逻辑
}

// sdk 代码中，有且仅有一个 main() 方法
func main() {
// 空，不做任何事。仅用于对 tinygo 编译支持
}

// 对 SDK 暴露的函数
// 对外暴露 test1 方法，供用户由 SDK 调用
//export test1
func test1 () {}
// 对外暴露 test2 方法，供用户由 SDK 调用
//export test2
func test2 () {}

// 其他函数，不对外暴露
func test3 () {}
```

代码入口

```
func main() { // sdk 代码中，有且仅有一个 main() 方法
    // 空，不做任何事。仅用于对 tinygo 编译支持
}
```

对链暴露方法写法

```
//export method_name
```

```
func  method_name(): 不可带参数，无返回值
```

例如：对链暴露 init_contract 函数。

```
//export init_contract
func init_contract() {

}
```

其中 init_contract、upgrade 方法必须有且对外暴露。示例如下：

- init_contract：创建合约会执行该方法
- upgrade： 升级合约会执行该方法

```
// 安装合约时会执行此方法，必须。ChainMaker 不允许用户直接调用该方法。
//export init_contract
func init_contract() {
}
// 升级合约时会执行此方法，必须。ChainMaker 不允许用户直接调用该方法。
//export upgrade
func upgrade() {
}
```

智能合约示例

存证合约示例

1. 存储文件哈希、文件名称、时间和该交易的 ID。
2. 通过文件哈希查询该条记录。

```
/*
Copyright (C) BABEC. All rights reserved.

SPDX-License-Identifier: Apache-2.0

一个 文件存证 的存取示例 fact

*/

package main

import (
    "chainmaker.org/contract-sdk-tinygo/sdk/convert"
)

// 安装合约时会执行此方法，必须
//export init_contract
func initContract() {
    // 此处可写安装合约的初始化逻辑

}

// 升级合约时会执行此方法，必须
//export upgrade
func upgrade() {
    // 此处可写升级合约的逻辑

}
```

```
// 存证对象
type Fact struct {
    fileHash string
    fileName string
    time     int32 // second
    ec       *EasyCodec
}

// 新建存证对象
func NewFact(fileHash string, fileName string, time int32) *Fact {
    fact := &Fact{
        fileHash: fileHash,
        fileName: fileName,
        time:     time,
    }
    return fact
}

// 获取序列化对象
func (f *Fact) getEasyCodec() *EasyCodec {
    if f.ec == nil {
        f.ec = NewEasyCodec()
        f.ec.AddString("fileHash", f.fileHash)
        f.ec.AddString("fileName", f.fileName)
        f.ec.AddInt32("time", f.time)
    }
    return f.ec
}

// 序列化为 json 字符串
func (f *Fact) toJson() string {
    return f.getEasyCodec().ToJson()
}

// 序列化为 cmec 编码
func (f *Fact) marshal() []byte {
    return f.getEasyCodec().Marshal()
}

// 反序列化 cmec 为存证对象
func unmarshalToFact(data []byte) *Fact {
    ec := NewEasyCodecWithBytes(data)
    fileHash, _ := ec.GetString("fileHash")
    fileName, _ := ec.GetString("fileName")
    time, _ := ec.GetInt32("time")

    fact := &Fact{
        fileHash: fileHash,
        fileName: fileName,
        time:     time,
        ec:       ec,
    }
    return fact
}

// 对外暴露 save 方法，供用户由 SDK 调用
//export save
func save() {
    // 获取上下文
    ctx := NewSimContext()
```

```
    // 获取参数
    fileHash, err1 := ctx.ArgString("file_hash")
    fileName, err2 := ctx.ArgString("file_name")
    timeStr, err3 := ctx.ArgString("time")

    if err1 != SUCCESS || err2 != SUCCESS || err3 != SUCCESS {
        ctx.Log("get arg fail.")
        ctx.ErrorResult("get arg fail.")
        return
    }

    time, err := convert.StringToInt32(timeStr)
    if err != nil {
        ctx.ErrorResult(err.Error())
        ctx.Log(err.Error())
        return
    }

    // 构建结构体
    fact := NewFact(fileHash, fileName, int32(time))

    // 序列化：两种方式
    jsonStr := fact.toJson()
    bytesData := fact.marshal()

    //发送事件
    ctx.EmitEvent("topic_vx", fact.fileHash, fact.fileName)

    // 存储数据
    ctx.PutState("fact_json", fact.fileHash, jsonStr)
    ctx.PutStateByte("fact_bytes", fact.fileHash, bytesData)

    // 记录日志
    ctx.Log("【save】 fileHash=" + fact.fileHash)
    ctx.Log("【save】 fileName=" + fact.fileName)
    // 返回结果
    ctx.SuccessResult(fact.fileName + fact.fileHash)
}

// 对外暴露 find_by_file_hash 方法，供用户由 SDK 调用
//export find_by_file_hash
func findByFileHash() {
    ctx := NewSimContext()
    // 获取参数
    fileHash, _ := ctx.ArgString("file_hash")
    // 查询 Json
    if result, resultCode := ctx.GetStateByte("fact_json", fileHash); resultCode != SUCCESS {
        // 返回结果
        ctx.ErrorResult("failed to call get_state, only 64 letters and numbers are allowed. got key:" +
"fact" + ", field:" + fileHash)
    } else {
        // 返回结果
        ctx.SuccessResultByte(result)
        // 记录日志
        ctx.Log("get val:" + string(result))
    }

    // 查询 EcBytes
    if result, resultCode := ctx.GetStateByte("fact_bytes", fileHash); resultCode == SUCCESS {
```

```
        // 反序列化
        fact := unmarshalToFact(result)
        // 返回结果
        ctx.SuccessResult(fact.toJson())
        // 记录日志
        ctx.Log("get val:" + fact.toJson())
        ctx.Log("【find_by_file_hash】 fileHash=" + fact.fileHash)
        ctx.Log("【find_by_file_hash】 fileName=" + fact.fileName)
    }
}

func main() {

}
```

存证合约代码说明

参数名称	描述
init_contract	合约的初始化函数，在合约部署时被调用，在本合约中为空。
upgrade	合约升级时调用的函数，在本合约中为空。
save	save 函数实现将文件相关信息记录到链上的功能。步骤如下： 1. save 函数先通过 [交易信息提取]API 接口 GetTxId 函数拿到交易哈希。 2. 通过 [参数处理]API 接口 ArgString 函数拿到时间，文件哈希和文件名字等信息。 3. 构造 Fact 结构，序列化为 byte 数据；且当序列化错误时调用 [其他辅助类]API 接口 LogMessage 函数记录相应日志。 4. 通过调用 [账本交互]API 接口 PutState 函数将数据记录到链上。5. 通过调用 [其他辅助类]API 接口 SuccessResult 函数将操作结果记录到链上。
findByFileHash	通过文件哈希查询该条记录。步骤如下： 1. findByFileHash 通过 [参数处理]API 接口 ArgString 函数拿到要查找的文件的文件哈希。 2. 通过 [账本交互]API 接口 GetState 函数获取文件的信息；若失败则通过 [其他辅助类]API 接口 ErrorResult 函数将操作结果记录到链上，否则，通过 [其他辅助类]API 接口 LogMessage 函数和 SuccessResult 函数记录相关日志和返回结果。

v1.2.0
TinyGo
智能合约开发

最近更新时间：2022-06-20 14:58:33

使用 Go（TinyGo）进行智能合约开发

本章节主要描述使用 Go 进行 ChainMaker 合约编写的方法，主要面向于使用 Go 进行 ChainMaker 的合约开发的开发者。为了最小化 wasm 文件尺寸，应使用 TinyGO 编译器。

使用 Docker 镜像进行合约开发

ChainMaker 官方已经将容器发布至 docker hub 。

1. 拉取镜像

```
docker pull chainmakerofficial/chainmaker-go-contract:1.2.0
```

请指定您本机的工作目录 $WORK_DIR，例如 /data/workspace/contract，挂载到 docker 容器中以方便后续进行必要的一些文件拷贝。

```
docker run -it --name chainmaker-go-contract -v $WORK_DIR:/home chainmakerofficial/chainmaker-go-
contract:1.2.0 bash
# 或者先后台启动
docker run -d  --name chainmaker-go-contract -v $WORK_DIR:/home chainmakerofficial/chainmaker-go-
contract:1.2.0 bash -c "while true; do echo hello world; sleep 5;done"
# 再进入容器
docker exec -it chainmaker-go-contract /bin/sh
```

2. 编译合约

```
cd /home/
# 解压缩合约SDK源码
tar xvf /data/contract_go_template.tar.gz
cd contract_tinygo
# 编译main.go合约
sh build.sh
```

生成合约的字节码文件位置为：

```
/home/contract_tinygo/main.wasm
```

`main.wasm` 文件可在 TBaaS 控制台 上传并部署。

3. 合约开发框架描述

解压缩 contract_go_template.tar.gz 后，文件描述如下：

```
/home/contract_tinygo# ls -l
total 64
-rw-rw-r-- 1 1000 1000    56 Jul  2 12:45 build.sh              # 编译脚本
-rw-rw-r-- 1 1000 1000  4149 Jul  2 12:44 bulletproofs.go       # 合约SDK基于bulletproofs的范围证明接口实
现
-rw-rw-r-- 1 1000 1000 18871 Jul  2 12:44 chainmaker.go         # 合约SDK主要接口及实现
-rw-rw-r-- 1 1000 1000  4221 Jul  2 12:44 chainmaker_rs.go      # 合约SDK sql接口实现
-rw-rw-r-- 1 1000 1000 11777 May 24 13:27 easycodec.go          # 序列化工具类
-rw-rw-r-- 1 1000 1000  3585 Jul  2 12:44 main.go               # 存证示例代码
-rwxr-xr-x 1 root root 65122 Jul  6 07:22 main.wasm             # 编译成功后的wasm文件
```

```
-rw-rw-r-- 1 1000 1000  1992 Jul  2 12:44 paillier.go          # 合约SDK基于paillier的半同态运算接口实现
```

用户使用 Go（TinyGo）编写智能合约后，可以把源代码更新到 `main.go` 文件中并重新编译，可得到新的智能合约的字节码，并前往 TBaaS 控制台上传并部署。更多关于使用 Go（TinyGo）开发长安链智能合约的详情，可参考长安链官网 使用 Go（TinyGo）进行智能合约开发 。

合约 API 列表

最近更新时间：2022-10-17 15:07:59

ChainMaker Go（TinyGo）语言版本智能合约有丰富的 API 接口，供用户在撰写智能合约的时候与链进行交互，代码实现详情可参见 API 接口代码实现 。从逻辑方面划分，可将 API 划分为以下类型：

交易信息提取

接口	说明
GetCreatorOrgId() (string, ResultCode)	获取合约创建者所属组织 ID
GetCreatorRole() (string, ResultCode)	获取合约创建者角色
GetCreatorPk() (string, ResultCode)	获取合约创建者公钥
GetSenderOrgId() (string, ResultCode)	获取交易发起者所属组织 ID
GetSenderRole() (string, ResultCode)	获取交易发起者角色
GetTxId() (string, ResultCode)	获取交易 ID
GetSenderPk() (string, ResultCode)	获取交易发起者公钥
GetBlockHeight() (string, ResultCode)	获取当前区块高度

账本交互

接口	说明
GetState(key string, field string) (string, ResultCode)	获取合约账户信息。该接口可从链上获取类别 “key” 下属性名为 “field” 的状态信息。
GetStateFromKey(key string) ([]byte, ResultCode)	获取合约账户信息。该接口可以从链上获取类别为 key 的状态信息
PutState(key string, field string, value string) ResultCode	写入合约账户信息。该接口可把类别 “key” 下属性名为 “filed” 的状态更新到链上。更新成功返回 0，失败则返回 1。
PutStateFromKey(key string, value string) ResultCode	写入合约账户信息。
DeleteState(key string, field string) ResultCode	删除合约账户信息。该接口可把类别 “key” 下属性名为 “name” 的状态从链上删除。
CallContract(contractName string, method string, param map[string]string) ([]byte, ResultCode)	跨合约调用。

参数处理

接口	说明
Args() []*EasyCodecItem{}	该接口调用 getArgsMap() 接口，把 json 格式的数据反序列化，并将解析出的数据返还给用户。
Arg(key string) (string, ResultCode)	该接口可返回属性名为 “key” 的参数的属性值。

其他辅助类

接口	说明
SuccessResult(msg string)	该接口可记录用户操作成功的信息，并将操作结果记录到链上。
ErrorResult(msg string)	该接口可记录用户操作失败的信息，并将操作结果记录到链上。

LogMessage(msg string)	该接口可记录事件日志。查看方式为在链配置的 log.yml 中，开启 vm:debug 即可看到类似：gasm log>> + msg
EmitEvent(topic string, data ...string) ResultCode	发送合约事件
NewIterator(startKey string, limitKey string) (ResultSetKV, ResultCode)	新建 key 范围迭代器，key 前闭后开，即：startKey <= dbkey < limitKey
NewIteratorWithField(key string, startField string, limitField string) (ResultSetKV, ResultCode)	新建 field 范围迭代器，key 需相同，field 前闭后开，即：key = dbdbkey and startField <= dbfield < limitField
NewIteratorPrefixWithKey(key string) (ResultSetKV, ResultCode)	新建指定 key 前缀匹配迭代器，key 需前缀一致，即 dbkey.startWith(key)
NewIteratorPrefixWithKeyField(key string, field string) (ResultSetKV, ResultCode)	新建指定 field 前缀匹配迭代器，key 需相同，field 前缀一致，即 dbkey = key and dbfield.startWith(field)

合约示例

最近更新时间：2023-02-22 16:04:28

智能合约构成

ChainMaker Go (TinyGo) 语言的智能合约代码主要由以下接口构成：

```
/*
Copyright (C) BABEC. All rights reserved.
SPDX-License-Identifier: Apache-2.0
一个 ChainMaker 的 TinyGO 版本智能合约主要包括以下函数：
*/
package main
// 安装合约时会执行此方法，必须
//export init_contract
func initContract() {
    // 此处可写安装合约的初始化逻辑
}

// 升级合约时会执行此方法，必须
//export upgrade
func upgrade() {
    // 此处可写升级合约的逻辑
}

// sdk 代码中，有且仅有一个 main() 方法
func main() {
// 空，不做任何事。仅用于对 tinygo 编译支持
}

// 对 SDK 暴露的函数
// 对外暴露 test1 方法，供用户由 SDK 调用
//export test1
func test1 () {}
// 对外暴露 test2 方法，供用户由 SDK 调用
//export test2
func test2 () {}

// 其他函数，不对外暴露
func test3 () {}
```

代码入口

```
func main() { // sdk 代码中，有且仅有一个 main() 方法
    // 空，不做任何事。仅用于对 tinygo 编译支持
}
```

对链暴露方法写法

```
//export method_name
func  method_name(): 不可带参数，无返回值
```

例如：对链暴露 init_contract 函数。

```
//export init_contract
func init_contract() {
```

```
}
```

其中 init_contract、upgrade 方法必须有且对外暴露。示例如下：

- init_contract：创建合约会执行该方法
- upgrade： 升级合约会执行该方法

```
// 安装合约时会执行此方法，必须。ChainMaker 不允许用户直接调用该方法。
//export init_contract
func init_contract() {
}
// 升级合约时会执行此方法，必须。ChainMaker 不允许用户直接调用该方法。
//export upgrade
func upgrade() {
}
```

智能合约示例

存证合约示例

1. 存储文件哈希、文件名称、时间和该交易的 ID。
2. 通过文件哈希查询该条记录。

```
/*
Copyright (C) BABEC. All rights reserved.

SPDX-License-Identifier: Apache-2.0

一个 文件存证 的存取示例 fact

*/

package main

import (
    "chainmaker-contract-sdk-tinygo/convert"
)

// 安装合约时会执行此方法，必须
//export init_contract
func initContract() {
    // 此处可写安装合约的初始化逻辑

}

// 升级合约时会执行此方法，必须
//export upgrade
func upgrade() {
    // 此处可写升级合约的逻辑

}

// 存证对象
type Fact struct {
    fileHash string
    fileName string
    time     int32 // second
    ec       *EasyCodec
```

```
}

// 新建存证对象
func NewFact(fileHash string, fileName string, time int32) *Fact {
    fact := &Fact{
        fileHash: fileHash,
        fileName: fileName,
        time:     time,
    }
    return fact
}

// 获取序列化对象
func (f *Fact) getEasyCodec() *EasyCodec {
    if f.ec == nil {
        f.ec = NewEasyCodec()
        f.ec.AddString("fileHash", f.fileHash)
        f.ec.AddString("fileName", f.fileName)
        f.ec.AddInt32("time", f.time)
    }
    return f.ec
}

// 序列化为json字符串
func (f *Fact) toJson() string {
    return f.getEasyCodec().ToJson()
}

// 序列化为cmec编码
func (f *Fact) marshal() []byte {
    return f.getEasyCodec().Marshal()
}

// 反序列化cmec为存证对象
func unmarshalToFact(data []byte) *Fact {
    ec := NewEasyCodecWithBytes(data)
    fileHash, _ := ec.GetString("fileHash")
    fileName, _ := ec.GetString("fileName")
    time, _ := ec.GetInt32("time")

    fact := &Fact{
        fileHash: fileHash,
        fileName: fileName,
        time:     time,
        ec:       ec,
    }
    return fact
}

// 对外暴露 save 方法，供用户由 SDK 调用
//export save
func save() {
    // 获取上下文
    ctx := NewSimContext()

    // 获取参数
    fileHash, err1 := ctx.Arg("file_hash")
    fileName, err2 := ctx.Arg("file_name")
    timeStr, err3 := ctx.Arg("time")
```

```
    if err1 != SUCCESS || err2 != SUCCESS || err3 != SUCCESS {
        ctx.Log("get arg fail.")
        ctx.ErrorResult("get arg fail.")
        return
    }

    time, err := convert.StringToInt32(timeStr)
    if err != nil {
        ctx.ErrorResult(err.Error())
        ctx.Log(err.Error())
        return
    }

    // 构建结构体
    fact := NewFact(fileHash, fileName, time)

    // 序列化：两种方式
    jsonStr := fact.toJson()
    bytesData := fact.marshal()

    //发送事件
    ctx.EmitEvent("topic_vx", fact.fileHash, fact.fileName)

    // 存储数据
    ctx.PutState("fact_json", fact.fileHash, jsonStr)
    ctx.PutStateByte("fact_bytes", fact.fileHash, bytesData)

    // 记录日志
    ctx.Log("【save】 fileHash=" + fact.fileHash)
    ctx.Log("【save】 fileName=" + fact.fileName)
    // 返回结果
    ctx.SuccessResult(fact.fileName + fact.fileHash)
}

// 对外暴露 find_by_file_hash 方法，供用户由 SDK 调用
//export find_by_file_hash
func findByFileHash() {
    ctx := NewSimContext()
    // 获取参数
    fileHash, _ := ctx.Arg("file_hash")
    // 查询Json
    if result, resultCode := ctx.GetStateByte("fact_json", fileHash); resultCode != SUCCESS {
        // 返回结果
        ctx.ErrorResult("failed to call get_state, only 64 letters and numbers are allowed. got key:" +
"fact" + ", field:" + fileHash)
    } else {
        // 返回结果
        ctx.SuccessResultByte(result)
        // 记录日志
        ctx.Log("get val:" + string(result))
    }

    // 查询EcBytes
    if result, resultCode := ctx.GetStateByte("fact_bytes", fileHash); resultCode == SUCCESS {
        // 反序列化
        fact := unmarshalToFact(result)
        // 返回结果
        ctx.SuccessResult(fact.toJson())
        // 记录日志
        ctx.Log("get val:" + fact.toJson())
```

```
        ctx.Log("【find_by_file_hash】 fileHash=" + fact.fileHash)
        ctx.Log("【find_by_file_hash】 fileName=" + fact.fileName)
    }
}

func main() {

}
```

存证合约代码说明

参数名称	描述
init_contract	合约的初始化函数，在合约部署时被调用，在本合约中为空。
upgrade	合约升级时调用的函数，在本合约中为空。
save	save 函数实现将文件相关信息记录到链上的功能。 1. save 函数先通过 [交易信息提取]API 接口GetTxId函数拿到交易哈希。 2. 通过 [参数处理]API 接口Arg函数拿到时间，文件哈希和文件名字等信息。 3. 构造 stone 结构，序列化为 byte 数据；且当序列化错误时调用 [其他辅助类]API 接口 LogMessage 函数记录相应日志。 4. 通过调用 [账本交互]API 接口 PutState 函数将数据记录到链上。 5. 通过调用 [其他辅助类]API 接口 SuccessResult 函数将操作结果记录到链上。
findByFileHash	通过文件哈希查询该条记录。 1. findByFileHash 通过 [参数处理]API 接口 Arg 函数拿到要查找的文件的文件哈希。 2. 通过 [账本交互]API 接口 GetState 函数获取文件的信息；若失败则通过 [其他辅助类]API 接口 ErrorResult 函数将操作结果记录到链上，否则，通过 [其他辅助类]API 接口 LogMessage 函数和 SuccessResult 函数记录相关日志和返回结果。

Solidity
智能合约开发

最近更新时间：2022-06-20 14:59:03

本章节主要描述使用 Solidity 进行 ChainMaker 合约编写的方法，主要面向于使用 Solidity 进行 ChainMaker 的合约开发的开发者。

使用 Docker 镜像进行合约开发

ChainMaker 官方已经将容器发布至 docker hub 。

1. 拉取镜像

代码示例如下：

```
docker pull chainmakerofficial/chainmaker-solidity-contract:1.2.0
```

请指定您本机的工作目录 $WORK_DIR，例如 /data/workspace/contract，挂载到 docker 容器中以方便后续进行必要的一些文件拷贝。

```
docker run -it --name chainmaker-solidity-contract -v $WORK_DIR:/home chainmakerofficial/chainmaker-
solidity-contract:1.2.0 bash
# 或者先后台启动
docker run -d --name chainmaker-solidity-contract -v $WORK_DIR:/home chainmakerofficial/chainmaker-
solidity-contract:1.2.0 bash -c "while true; do echo hello world; sleep 5;done"
# 再进入容器
docker exec -it chainmaker-solidity-contract /bin/sh
```

2. 编译合约

代码示例如下：

```
cd /home/
# 解压缩合约SDK源码
tar xvf /data/contract_solidity_template.tar.gz
cd contract_solidity
# 编译token.sol合约
solc --abi --bin --hashes --overwrite -o . token.sol
```

生成合约的字节码文件路径如下：

```
/home/contract_solidity/Token.bin
```

`Token.bin` 文件可在 TBaaS 控制台 上传并部署。

3. 合约开发框架描述

解压缩 contract_solidity_template.tar.gz 后，文件描述如下：

```
/home/contract_solidity# ls -l
total 4
-rw-rw-r-- 1 1000 1000 2816 Apr 29  2021 token.sol              # token合约
```

用户使用 Solidity 编写智能合约后，可以把源代码更新到 `token.sol` 文件中并重新编译，可得到新的智能合约的字节码，并前往 TBaaS 控制台 上传并部署。更多关于使用 Solidity 开发长安链智能合约的详情，可参考长安链官网 使用 Solidity 进行智能合约开发 。

合约 API 列表

最近更新时间：2022-10-17 15:08:47

ChainMaker Solidity 语言版本智能合约有丰富的 API 接口，供用户在撰写智能合约的时候与链进行交互，代码实现详情可以参考 API 接口代码实现 。
从逻辑方面划分，可将 API 划分为以下类型：

交易信息提取

接口	说明
msg.data -> bytes	获取调用合约的完整数据
msg.sender -> address	获取消息发送者地址
tx.origin -> address	获取交易发送者地址
tx.gasprice -> uint	获取交易的 gas 价格

账本交互

接口	说明
blockhash(uint blockNumber) -> bytes32	获取指定区块高度的哈希值
block.gaslimit -> uint	获取当前区块的 gas 限制
block.number -> uint	获取当前区块的高度
block.timestamp -> uint	获取当前区块的时间戳

异常处理

接口	说明
assert(bool condition)	断言给定条件是否成立，如果不满足条件，则状态更改恢复
require(bool condition)	如果不满足条件，则返回
require(bool condition, string memory message)	如果不满足条件，则返回，并提供错误消息
revert()	中止执行并还原状态更改
revert(string memory reason)	中止执行并还原状态更改，并提供错误消息

数学和密码函数类

接口	说明
addmod(uint x, uint y, uint k) -> uint	计算 (x+y)%k
mulmod(uint x, uint y, uint k) -> uint	计算 (x*y)%k
keccak256(bytes memory) -> bytes32	计算给定输入的 Keccak-256 哈希
sha256(bytes memory) -> bytes32	计算给定输入的 sha256 哈希
ripemd160(bytes memory) -> bytes32	计算给定输入的 RIPEMD-160 哈希
ecrecover(bytes32 hash, uint8 v, bytes32 r, bytes32 s) -> address	从椭圆曲线签名中恢复与公钥关联的地址

ChainMake Solidity 语言版本智能合约完全兼容 EVM，Solidity 的具体使用详情可参见 Solidity 官方文档 。

合约调用

最近更新时间：2022-10-17 15:09:15

长安链证书转换 EVM 地址

ChainMaker Solidity 语言版本智能合约完全兼容 EVM，更多长安链证书与 EVM 地址的转换详情可参见 EVM 地址说明 。
在“证书管理”界面申请证书后，可根据获得的用户证书文件 user_sign.crt 获取该用户的 EVM 地址，代码示例如下：

```
import (
    "encoding/hex"
    "encoding/pem"
    "fmt"
    "io/ioutil"

    "chainmaker.org/chainmaker-go/common/crypto/x509"
    "chainmaker.org/chainmaker-go/common/evmutils"
    "github.com/ethereum/go-ethereum/accounts/abi"
)

func MakeAddrAndSkiFromCrtFilePath() {
    crtFilePath := "./user_sign.crt"

    crtBytes, err := ioutil.ReadFile(crtFilePath)
    if err != nil {
        fmt.Printf("fail to read the crt file:%v", err)
    }

    blockCrt, _ := pem.Decode(crtBytes)
    crt, err := x509.ParseCertificate(blockCrt.Bytes)
    if err != nil {
        fmt.Printf("fail to parse certificate:%v", err)
    }

    ski := hex.EncodeToString(crt.SubjectKeyId)
    addrInt, err := evmutils.MakeAddressFromHex(ski)
    if err != nil {
    fmt.Printf("fail to make address from hex:%v", err)
    }

    // 证书 SKI
    fmt.Printf("clientAddrSki: %s\n", ski)
    // EVM 地址（十进制）
    fmt.Printf("clientAddrInt: %s\n", addrInt.String())
    // EVM 地址
    fmt.Printf("clientEthAddr: 0x%x\n", addrInt.AsStringKey())
}
```

成功运行可以查看到如下图输出：

```
clientAddrSki: 65176a125555576d9a97b40a9a39a637f19dd75e5cc0043ab109419de32e6e64
clientAddrInt: 902584883230384040201055326396503673974016209567
clientEthAddr: 0x9e194e526ff154807d82c09cbb65e709bd48729f
```

ABI 编码示例

ChainMake Solidity 语言版本智能合约完全兼容 EVM，更多 ABI 编码的详情可参见 Solidity 官方文档 。

合约初始化

以 Token 合约为例，对合约初始化参数进行 ABI 编码。代码示例如下：

```
import (
    "encoding/hex"
    "fmt"
    "io/ioutil"
    "math/big"
    "strings"

    "chainmaker.org/chainmaker-go/common/evmutils"
    "github.com/ethereum/go-ethereum/accounts/abi"
)

const (
    // 编译合约后获取的 abi 文件的路径
    tokenABIPath = "./testdata/token-evm-demo/token.abi"
    // 合约安装时调用 constructor ，调用方法设为空字符串
    function = ""
    // 入参 1：发行 EVM 地址，可根据用户证书转换取得
    clientAddr = "0x89f4090e315621696d6936453661ec4b9795ad27"
)

func testUserContractTokenEVMConstructor () {
    abiJson, err := ioutil.ReadFile (tokenABIPath)
    if err != nil {
        fmt.Printf ("fail to read the abi file:%v", err)
    }

    myAbi, err := abi.JSON (strings.NewReader (string (abiJson)))
    if err != nil {
        fmt.Printf ("fail to get abi object:%v", err)
    }

    addr := evmutils.BigToAddress (evmutils.FromHexString (clientAddr [2:]))

    dataByte, err := myAbi.Pack (function, addr)
    if err != nil {
        fmt.Printf ("fail to pack contract input:%v", err)
    }

    data := hex.EncodeToString (dataByte)

    pairs := map [string] string {
        "data": data,
    }

    // 编码后的参数
    fmt.Printf ("FuncParam %v\n", pairs)
}
```

输出结果如下：

```
FuncParam map [data:00000000000000000000000089f4090e315621696d6936453661ec4b9795ad27]
```

通过 TBaaS 控制台安装合约并填写对应初始化参数：

- key 取值：data
- value 取值：00000000000000000000000089f4090e315621696d6936453661ec4b9795ad27

合约调用

以 Token 合约为例，对 transfer 函数的函数名及调用参数进行 ABI 编码。代码示例如下：

```
import (
    "encoding/hex"
    "fmt"
    "io/ioutil"
    "math/big"
    "strings"

    "chainmaker.org/chainmaker-go/common/evmutils"
    "github.com/ethereum/go-ethereum/accounts/abi"
)

const (
    // 编译合约后获取的 abi 文件的路径
    tokenABIPath = "./testdata/token-evm-demo/token.abi"
    // 调用方法
    function = "transfer"
    // 入参 1：转账 EVM 地址，可根据用户证书转换取得
    clientAddr = "0xa55f1e0cb68b0cc589906078237094bdb9715bfd"
    // 入参 2：转账金额
    amount = 200
)

func testUserContractTokenEVMTransfer () {
    abiJson, err := ioutil.ReadFile (tokenABIPath)
    if err != nil {
        fmt.Printf ("fail to read the abi file:%v", err)
    }

    myAbi, err := abi.JSON (strings.NewReader (string (abiJson)))
    if err != nil {
        fmt.Printf ("fail to get abi object:%v", err)
    }

    addr := evmutils.BigToAddress (evmutils.FromHexString (clientAddr [2:]))

    dataByte, err := myAbi.Pack (function, addr, big.NewInt (amount))
    if err != nil {
        fmt.Printf ("fail to pack contract input:%v", err)
    }

    data := hex.EncodeToString (dataByte)
    method := data [0:8]

    pairs := map [string] string {
        "data": data,
    }

    // 编码后的函数名
    fmt.Printf ("FuncName: %s\n", method)
    // 编码后的参数
    fmt.Printf ("FuncParam %v\n", pairs)
}
```

输出结果如下：

```
FuncName: a9059cbb
FuncParam map
[data:a9059cbb000000000000000000000000a55f1e0cb68b0cc589906078237094bdb9715bfd000000000000000000000000000000000000000000000000000000000000000000000000000000000000c8]
```

长安链 SDK 调用示例

将 ABI 编码后的函数名及调用参数分别作为 method 与 params，代码示例如下：

```
    method := "a9059cbb"
    params := map [string] string {
        "data":
"a9059cbb000000000000000000000000a55f1e0cb68b0cc589906078237094bdb9715bfd000000000000000000000000000000000000000000000000000000000000000000000000000000000000c8",
    }
    resp, err := client.InvokeContract ("fact", method, "", params, -1, true)
    if err != nil {
        fmt.Printf ("fail to invoke contract:%v", err)
    }
```

云 API 调用示例

将 ABI 编码后的函数名及调用参数分别填入 FuncName 与 FuncParam 字段，代码示例如下：

```
    action_params = {
        "ClusterId": "chainmaker-txtxtxtxtx",
        "ChainId": "chain_txtxt",
        "ContractName": "fact",
        "FuncName": "a9059cbb",
        "FuncParam": "
{\"data\":\"a9059cbb000000000000000000000000a55f1e0cb68b0cc589906078237094bdb9715bfd000000000000000000000000000000000000000000000000000000000000000000000000000000000000c8\"}",
        "AsyncFlag": 0,
        "Action": "InvokeChainMakerContract",
        "Version": "2018-04-16",
        "Region": "ap-beijing"
    }
    # 实例化一个请求对象，根据调用的接口和实际情况，可以进一步设置请求参数
    params = json.dumps (action_params)
    req = models.InvokeChainMakerContractRequest ()
    # 调用 InvokeChainMakerContractRequest 的 from_json_string 方法，使用 params 初始化 req 对象
    req.from_json_string (params)

    # 通过 client 对象调用想要访问的接口，需要传入请求对象
    resp = client.InvokeChainMakerContract (req)
    # 输出 json 格式的字符串回包
    print (resp.to_json_string ())
```

合约示例

最近更新时间：2022-06-20 14:59:31

Token 合约代码示例

Token 合约代码示例如下，实现功能 ERC20。

```
/*
SPDX-License-Identifier: Apache-2.0
*/
pragma solidity >0.5.11;
contract Token {

    string public name = "token";      //  token name
    string public symbol = "TK";           //  token symbol
    uint256 public decimals = 6;            //  token digit

    mapping (address => uint256) public balanceOf;
    mapping (address => mapping (address => uint256)) public allowance;

    uint256 public totalSupply = 0;
    bool public stopped = false;

    uint256 constant valueFounder = 100000000000000000;
    address owner = address (0x0);

    modifier isOwner {
        assert (owner == msg.sender);
        _;
    }

    modifier isRunning {
        assert (!stopped);
        _;
    }

    modifier validAddress {
        assert (address (0x0) != msg.sender);
        _;
    }

    constructor (address _addressFounder) {
        owner = msg.sender;
        totalSupply = valueFounder;
        balanceOf [_addressFounder] = valueFounder;

        emit Transfer (address (0x0), _addressFounder, valueFounder);
    }

    function transfer (address _to, uint256 _value) public isRunning validAddress returns (bool success)
{
        require (balanceOf [msg.sender] >= _value);
        require (balanceOf [_to] + _value >= balanceOf [_to]);
        balanceOf [msg.sender] -= _value;
        balanceOf [_to] += _value;
        emit Transfer (msg.sender, _to, _value);
        return true;
    }
```

```
    function transferFrom (address _from, address _to, uint256 _value) public isRunning validAddress
returns (bool success) {
        require (balanceOf [_from] >= _value);
        require (balanceOf [_to] + _value >= balanceOf [_to]);
        require (allowance [_from][msg.sender] >= _value);
        balanceOf [_to] += _value;
        balanceOf [_from] -= _value;
        allowance [_from][msg.sender] -= _value;
        emit Transfer (_from, _to, _value);
        return true;
    }

    function approve (address _spender, uint256 _value) public isRunning validAddress returns (bool
success) {
        require (_value == 0 || allowance [msg.sender][_spender] == 0);
        allowance [msg.sender][_spender] = _value;
        emit Approval (msg.sender, _spender, _value);
        return true;
    }

    function stop () public isOwner {
        stopped = true;
    }

    function start () public isOwner {
        stopped = false;
    }

    function setName (string memory _name) public isOwner {
        name = _name;
    }

    function burn (uint256 _value) public {
        require (balanceOf [msg.sender] >= _value);
        balanceOf [msg.sender] -= _value;
        balanceOf [address (0x0)] += _value;
        emit Transfer (msg.sender, address (0x0), _value);
    }

    event Transfer (address indexed _from, address indexed _to, uint256 _value);
    event Approval (address indexed _owner, address indexed _spender, uint256 _value);
}
```

Token 合约代码说明

参数名称	描述
constructor	合约构造函数，在合约部署时被调用，将 _addressFounder 的余额设置为 valueFounder。
transfer	转账函数，该函数具有两个入参，接收者地址 _to 和转账金额 _value，该函数将调用者 msg.sender 的余额减去 _value，将接收者的余额加上 _value。
transferFrom	转账函数，该函数具有三个入参，转账者地址 _from，接收者地址 _to 和转账金额 _value，该函数将转账者 _from 的余额减去 _value，将接收者的余额加上 _value。
approve	批准函数，该函数具有两个入参，接收者地址 _spender 和 token 数量 _value，该函数将接收者 _spender 可以从调用者 msg.sender 处转出的 token 数量设置为 _value。

stop	停止函数，该函数执行后，transfer，transferFrom 和 approve 函数将不能再被执行。
start	启动函数，该函数执行后，transfer，transferFrom 和 approve 函数将可以再被执行。
setName	设置 token 名字函数，该函数具有一个入参 _name，将 token 的名字设置为 _name。
burn	销毁函数，该函数具有一个入参 _value，将调用者 msg.sender 的余额减去 _value。

Rust
智能合约开发

最近更新时间：2022-06-20 14:59:36

本章节主要描述使用 Rust 进行 ChainMaker 合约编写的方法，主要面向于使用 Rust 进行 ChainMaker 的合约开发的开发者。
Rust 安装及教程请参考 Rust 官网 。

使用 Docker 镜像进行合约开发

ChainMaker 官方已经将容器发布至 docker hub

1. 拉取镜像

```
docker pull chainmakerofficial/chainmaker-rust-contract:1.2.0
```

请指定您本机的工作目录 $WORK_DIR，例如 /data/workspace/contract，挂载到 docker 容器中以方便后续进行必要的一些文件拷贝

```
docker run -it --name chainmaker-rust-contract -v $WORK_DIR:/home chainmakerofficial/chainmaker-rust-
contract:1.2.0 bash
# 或者先后台启动
docker run -d  --name chainmaker-rust-contract -v $WORK_DIR:/home chainmakerofficial/chainmaker-rust-
contract:1.2.0 bash -c "while true; do echo hello world; sleep 5;done"
# 再进入容器
docker exec -it chainmaker-rust-contract /bin/sh
```

2. 编译合约

```
cd /home/
tar xvf /data/contract_rust_template.tar.gz
cd contract_rust
wasm-pack build
```

生成的合约字节码文件位置如下：

```
/home/contract_rust/target/wasm32-unknown-unknown/release/chainmaker_contract.wasm
```

`chainmaker_contract.wasm` 文件可在 TBaaS 控制台 上传并部署。

3. 合约开发框架描述
解压缩 contract_rust_template.tar.gz 后，文件描述如下：

```
chainmaker-contract-sdk-rust$ tree -I target
.
├── Cargo.lock # 依赖版本信息
├── Cargo.toml # 项目配置及依赖，参考：https://rustwasm.github.io/wasm-pack/book/cargo-toml-
configuration.html
├── Makefile   # build一个wasm文件
├── README.md  # 编译环境说明
├── src
|   ├── contract_fact.rs            # 存证示例代码
|   ├── easycodec.rs                # 序列化工具类
|   ├── lib.rs                      # 程序入口
|   ├── sim_context.rs              # 合约SDK主要接口及实现
|   ├── sim_context_bulletproofs.rs # 合约SDK基于bulletproofs的范围证明接口实现
|   ├── sim_context_paillier.rs     # 合约SDK基于paillier的半同态运算接口实现
|   ├── sim_context_rs.rs           # 合约SDK sql接口实现
```

```
|   └── vec_box.rs                  # 内存管理类
```

用户使用 Rust 编写智能合约后，可以把源代码更新到 `src/contract_fact.rs` 文件中并重新编译，得到新的智能合约的字节码，并前往 TBaaS 控制台 上传并部署。更多关于使用 Rust 进行开发长安链智能合约的详情，可参考长安链官网 使用 Rust 进行智能合约开发

合约 API 列表

最近更新时间：2022-10-17 15:09:55

ChainMaker Rust 语言版本智能合约有丰富的 API 接口，供用户在撰写智能合约的时候与链进行交互，代码实现详情可以参考 API 接口代码实现 。
从逻辑方面划分，可将 API 划分为以下类型：

交易信息提取

接口	说明
get_creator_org_id(&self) -> String	获取合约创建者所属组织 ID
get_creator_role(&self) -> String	获取合约创建者角色
get_creator_pub_key(&self) -> String	获取合约创建者公钥
get_sender_org_id(&self) -> String	获取交易发起者所属组织 ID
get_sender_role(&self) -> String	获取交易发起者角色
get_tx_id(&self) -> String	获取交易 ID
get_sender_pub_key(&self) -> String	获取交易发起者公钥
get_block_height(&self) -> i32	获取当前区块高度

账本交互

接口	说明
get_state(&self, key: &str, field: &str) -> Result<Vec<u8>, result_code>	获取合约账户信息。该接口可从链上获取类别 “key” 下属性名为 “field” 的状态信息。
get_state_from_key(&self, key: &str) -> Result<Vec<u8>, result_code>	获取合约账户信息。该接口可以从链上获取类别为 key 的状态信息
put_state(&self, key: &str, field: &str, value: &[u8]) -> result_code	写入合约账户信息。该接口可把类别 “key” 下属性名为 “filed” 的状态更新到链上。更新成功返回 0，失败则返回 1。
put_state_from_key(&self, key: &str, value: &[u8]) -> result_code	写入合约账户信息。
delete_state(&self, key: &str, field: &str) -> result_code	删除合约账户信息。该接口可把类别 “key” 下属性名为 “name” 的状态从链上删除。
delete_state_from_key(&self, key: &str) -> result_code;	删除合约账户信息。该接口可把类别 “key” 下属性名为 “name” 的状态从链上删除。
call_contract(&self, contract_name: &str, method: &str, param: EasyCodec) -> Result<Vec<u8>, result_code>;	跨合约调用。

参数处理

接口	说明
arg(&self, key: &str) -> Result<String, String>	该接口调用 getArgsMap() 接口，把 json 格式的数据反序列化，并将解析出的数据返还给用户。
arg_default_blank(&self, key: &str) -> String;	该接口可返回属性名为 “key” 的参数的属性值。

其他辅助类

接口	说明
ok(&self, value: &[u8]) -> result_code	该接口可记录用户操作成功的信息，并将操作结果记录到链上。
error(&self, body: &str) -> result_code	该接口可记录用户操作失败的信息，并将操作结果记录到链上。
log(&self, msg: &str)	该接口可记录事件日志。查看方式为在链配置的 log.yml 中，开启 vm:debug 即可看到类似：gasm log>> + msg
emit_event(&mut self, topic: &str, data: &Vec<String>) -> result_code	发送合约事件
new_iterator(&self,start_key: &str,limit_key: &str) -> Result<Box<dyn ResultSet>, result_code>	新建 key 范围迭代器，key 前闭后开，即：startKey <= dbkey < limitKey
new_iterator_with_field(&self, key: &str, start_field: &str, limit_field: &str) -> Result<Box<dyn ResultSet>, result_code>	新建 field 范围迭代器，key 需相同，field 前闭后开，即：key = dbdbkey and startField <= dbfield < limitField
new_iterator_prefix_with_key(&self, key: &str) -> Result<Box<dyn ResultSet>, result_code>	新建指定 key 前缀匹配迭代器，key 需前缀一致，即 dbkey.startWith(key)
new_iterator_prefix_with_key_field(&self, key: &str, field: &str) -> Result<Box<dyn ResultSet>, result_code>	新建指定 field 前缀匹配迭代器，key 需相同，field 前缀一致，即 dbkey = key and dbfield.startWith(field)

合约示例

最近更新时间：2023-02-22 16:04:28

智能合约构成

ChainMaker Rust 语言的智能合约代码主要由以下接口构成：

```
/*
Copyright (C) BABEC. All rights reserved.
SPDX-License-Identifier: Apache-2.0
一个 ChainMaker 的 Rust 版本智能合约主要包括以下函数：
*/
use crate::easycodec::*;
use crate::sim_context;
use sim_context::*;

// 安装合约时会执行此方法，必须
#[no_mangle]
pub extern "C" fn init_contract() {
    // 安装时的业务逻辑，内容可为空
    sim_context::log("init_contract");
}

// 升级合约时会执行此方法，必须
#[no_mangle]
pub extern "C" fn upgrade() {
    // 升级时的业务逻辑，内容可为空
    sim_context::log("upgrade");
    let ctx = &mut sim_context::get_sim_context();
    ctx.ok("upgrade success".as_bytes());
}

// 对 SDK 暴露的函数
// 对外暴露 test1 方法，供用户由 SDK 调用
#[no_mangle]
pub extern "C" fn test1() {}
// 对外暴露 test2 方法，供用户由 SDK 调用
#[no_mangle]
pub extern "C" fn test2() {}

// 其他函数，不对外暴露
fn test3() {}
```

智能合约示例

存证合约示例，实现功能

1、存储文件哈希、文件名称和时间等信息。
2、通过文件哈希查询该条记录

```
use crate::easycodec::*;
use crate::sim_context;
use sim_context::*;

// 安装合约时会执行此方法，必须
#[no_mangle]
pub extern "C" fn init_contract() {
    // 安装时的业务逻辑，内容可为空
```

```
    sim_context::log("init_contract");
}

// 升级合约时会执行此方法，必须
#[no_mangle]
pub extern "C" fn upgrade() {
    // 升级时的业务逻辑，内容可为空
    sim_context::log("upgrade");
    let ctx = &mut sim_context::get_sim_context();
    ctx.ok("upgrade success".as_bytes());
}

struct Fact {
    file_hash: String,
    file_name: String,
    time: i32,
    ec: EasyCodec,
}

impl Fact {
    fn new_fact(file_hash: String, file_name: String, time: i32) -> Fact {
        let mut ec = EasyCodec::new();
        ec.add_string("file_hash", file_hash.as_str());
        ec.add_string("file_name", file_name.as_str());
        ec.add_i32("time", time);
        Fact {
            file_hash,
            file_name,
            time,
            ec,
        }
    }
    fn get_emit_event_data(&self) -> Vec<String> {
        let mut arr: Vec<String> = Vec::new();
        arr.push(self.file_hash.clone());
        arr.push(self.file_name.clone());
        arr.push(self.time.to_string());
        arr
    }
    fn to_json(&self) -> String {
        self.ec.to_json()
    }
    fn marshal(&self) -> Vec<u8> {
        self.ec.marshal()
    }
    fn unmarshal(data: &Vec<u8>) -> Fact {
        let ec = EasyCodec::new_with_bytes(data);
        Fact {
            file_hash: ec.get_string("file_hash").unwrap(),
            file_name: ec.get_string("file_name").unwrap(),
            time: ec.get_i32("time").unwrap(),
            ec,
        }
    }
}

// save 保存存证数据
#[no_mangle]
pub extern "C" fn save() {
    // 获取上下文
```

```
    let ctx = &mut sim_context::get_sim_context();
    // 获取传入参数
    let file_hash = ctx.arg_default_blank("file_hash");
    let file_name = ctx.arg_default_blank("file_name");
    let time_str = ctx.arg_default_blank("time");
    // 构造结构体
    let r_i32 = time_str.parse::<i32>();
    if r_i32.is_err() {
        let msg = format!("time is {:?} not int32 number.", time_str);
        ctx.log(&msg);
        ctx.error(&msg);
        return;
    }
    let time: i32 = r_i32.unwrap();
    let fact = Fact::new_fact(file_hash, file_name, time);
    // 事件
    ctx.emit_event("topic_vx", &fact.get_emit_event_data());
    // 序列化后存储
    ctx.put_state(
        "fact_ec",
        fact.file_hash.as_str(),
        fact.marshal().as_slice(),
    );
}

// find_by_file_hash 根据 file_hash 查询存证数据
#[no_mangle]
pub extern "C" fn find_by_file_hash() {
    // 获取上下文
    let ctx = &mut sim_context::get_sim_context();
    // 获取传入参数
    let file_hash = ctx.arg_default_blank("file_hash");
    // 校验参数
    if file_hash.len() == 0 {
        ctx.log("file_hash is null");
        ctx.ok("".as_bytes());
        return;
    }
    // 查询
    let r = ctx.get_state("fact_ec", &file_hash);
    // 校验返回结果
    if r.is_err() {
        ctx.log("get_state fail");
        ctx.error("get_state fail");
        return;
    }
    let fact_vec = r.unwrap();
    if fact_vec.len() == 0 {
        ctx.log("None");
        ctx.ok("".as_bytes());
        return;
    }
    // 查询
    let r = ctx.get_state("fact_ec", &file_hash).unwrap();
    let fact = Fact::unmarshal(&r);
    let json_str = fact.to_json();
    // 返回查询结果
    ctx.ok(json_str.as_bytes());
    ctx.log(&json_str);
```

```
}
```

存证合约代码说明

参数名称	描述
init_contract	合约的初始化函数，在合约部署时被调用，在本合约中打印了日志"init_contract"。
upgrade	合约升级时调用的函数。
save	save 函数实现将文件哈希和文件名称记录到链上的功能。步骤如下： 1. save 函数先通过 get_sim_context 和 [交易信息提取]API 接口 arg_default_blank 函数拿到时间，文件哈希和文件名字等信息。 2. 构造 Fact 结构体并进行序列化；且当序列化错误时调用 [其他辅助类]API 接口 log 函数和 error 函数记录相应日志。 3. 通过调用 [其他辅助类]API 接口 emit_event 函数发送标识为 topic_vx 的合约事件。 4. 通过调用 [账本交互]API 接口 put_state 函数将文件等信息记录到链上。
findByFileHash	通过文件哈希查询该条记录。步骤如下： 1. find_by_file_hash 通过 get_sim_context 和 [参数处理]API 接口 arg_default_blank 函数拿到要查找的文件的文件哈希。 2. 通过 [账本交互]API 接口 get_state 函数获取文件的信息；若失败则通过 [其他辅助类]API 接口 error 函数将操作结果记录到链上，否则，通过 [其他辅助类]API 接口 log 函数和 ok 函数记录相关日志并返回结果。

C++
智能合约开发

最近更新时间：2022-06-20 14:59:49

本章节主要描述使用 C++ 进行 ChainMaker 合约编写的方法，主要面向于使用 C++ 进行 ChainMaker 的合约开发的开发者。

使用 Docker 镜像进行合约开发

ChainMaker 官方已经将容器发布至 docker hub 。

1. 拉取镜像

```
docker pull chainmakerofficial/chainmaker-cpp-contract:1.2.0
```

请指定您本机的工作目录 $WORK_DIR，例如 /data/workspace/contract，挂载到 docker 容器中以方便后续进行必要的一些文件拷贝。

```
docker run -it --name chainmaker-cpp-contract -v $WORK_DIR:/home chainmakerofficial/chainmaker-cpp-
contract:1.2.0 bash
# 或者先后台启动
docker run -d --name chainmaker-cpp-contract -v $WORK_DIR:/home chainmakerofficial/chainmaker-cpp-
contract:1.2.0 bash -c "while true; do echo hello world; sleep 5;done"
# 再进入容器
docker exec -it chainmaker-cpp-contract /bin/sh
```

2. 编译合约

```
cd /home/
tar xvf /data/contract_cpp_template.tar.gz
cd contract_cpp
make clean
emmake make
```

生成的合约字节码文件位置如下：

```
/home/contract_cpp/main.wasm
```

`main.wasm` 文件可在 TBaaS 控制台 上传并部署。

3. 合约开发框架描述

解压缩 contract_cpp_template.tar.gz 后，文件描述如下：

- chainmaker
  - basic_iterator.cc：迭代器实现
  - basic_iterator.h：迭代器头文件声明
  - chainmaker.h：sdk 主要接口头文件声明，详情见 SDK API 描述
  - context_impl.cc：与链交互接口实现
  - context_impl.h：与链交互头文件声明
  - contract.cc：合约基础工具类
  - error.h：异常处理类
  - exports.js：编译合约导出函数
  - safemath.h：assert 异常处理
  - syscall.cc：与链交互入口
  - syscall.h：与链交互头文件声明
- pb

- **contract.pb.cc**：与链交互数据协议
- **contract.pb.h**：与链交互数据协议头文件声明

- **main.cc**：用户写合约入口
- **Makefile**：常用 build 命令

用户使用 C++ 编写智能合约后，可以把源代码更新到 `main.cc` 文件中并重新编译，可得到新的智能合约的压缩文件，并前往 TBaaS 控制台 上传并部署。更多关于使用 C++ 开发长安链智能合约的详情，可参考长安链官网 使用 C++ 进行智能合约开发 。

合约 API 列表

最近更新时间：2023-02-21 14:47:43

ChainMaker C++ 语言版本智能合约有丰富的 API 接口，供用户在撰写智能合约的时候与链进行交互，代码实现详情可以参考 API 接口代码实现 。
从逻辑方面划分，可将 API 划分为以下类型：

账本交互

接口	说明
bool get_object(const std::string& key, std::string* value){}	获取key为"key"的值。
bool put_object(const std::string& key, const std::string& value){}	存储key为"key"的值，注意key长度不允许超过64，且只允许大小写字母、数字、下划线、减号、小数点符号。
bool delete_object(const std::string& key) {}	删除key为"key"的值。
bool call(const std::string& contract, const std::string& method, const std::map & args, Response* response){}	跨合约调用。

参数处理

接口	说明
bool arg(const std::string& name, std::string& value){}	该接口可返回属性名为 “name” 的参数的属性值。需要注意的是通过arg接口返回的参数，全部为字符串，合约开发者有必要将其他数据类型的参数与字符串做转换，包括atoi、itoa、自定义序列化方式等。

其他辅助类

接口	说明
void success(const std::string& body) {}	返回成功的结果
void error(const std::string& body) {}	返回失败结果
void log(const std::string& body) {}	输出日志事件。查看方式为在链配置的log.yml中，开启vm:debug即可看到类似：wxvm log>> + msg
bool emit_event(const std::string &topic, int data_amount, const std::string data, ...)	发送合约事件

合约示例

最近更新时间：2023-02-22 16:04:28

智能合约构成

ChainMaker C++ 语言的智能合约代码主要由以下接口构成：

```
#include "chainmaker/chainmaker.h"

using namespace chainmaker;

class Counter : public Contract {
public:
    void init_contract() {}
    void upgrade() {}
};

// 在创建本合约时，调用一次init方法. ChainMaker不允许用户直接调用该方法.
WASM_EXPORT void init_contract() {
    // 安装时的业务逻辑，可为空

}

// 在升级本合约时，对于每一个升级的版本调用一次upgrade方法. ChainMaker不允许用户直接调用该方法.
WASM_EXPORT void upgrade() {
    // 升级时的业务逻辑，可为空

}

// 对 SDK 暴露的函数
// 对外暴露 test1 方法，供用户由 SDK 调用
WASM_EXPORT void test1() {}
// 对外暴露 test2 方法，供用户由 SDK 调用
WASM_EXPORT void test2() {}
```

智能合约示例

存证合约示例

可实现如下功能：

1. 存储文件哈希、文件名称和时间等信息。
2. 通过文件哈希查询该条记录。

```
#include "chainmaker/chainmaker.h"

using namespace chainmaker;

class Counter : public Contract {
public:
    void init_contract() {}
    void upgrade() {}
    // 保存
    void save() {
        // 获取SDK 接口上下文
        Context* ctx = context();
        // 定义变量
        std::string time;
```

```
        std::string file_hash;
        std::string file_name;
        std::string tx_id;
        // 获取参数
        ctx->arg("time", time);
        ctx->arg("file_hash", file_hash);
        ctx->arg("file_name", file_name);
        ctx->arg("tx_id", tx_id);
        // 发送合约事件
        // 向topic:"topic_vx"发送2个event数据，file_hash,file_name
        ctx->emit_event("topic_vx",2,file_hash.c_str(),file_name.c_str());
        // 存储数据
        ctx->put_object("fact"+ file_hash,  tx_id+" "+time+" "+file_hash+" "+file_name);
        // 记录日志
        ctx->log("call save() result:" + tx_id+" "+time+" "+file_hash+" "+file_name);
        // 返回结果
        ctx->success(tx_id+" "+time+" "+file_hash+" "+file_name);
    }

    // 查询
    void find_by_file_hash() {
        // 获取SDK 接口上下文
        Context* ctx = context();

        // 获取参数
        std::string file_hash;
        ctx->arg("file_hash", file_hash);

        // 查询数据
        std::string value;
        ctx->get_object("fact"+ file_hash, &value);
        // 记录日志
        ctx->log("call find_by_file_hash()-" + file_hash + ",result:" + value);
        // 返回结果
        ctx->success(value);
    }

};

// 在创建本合约时，调用一次init方法. ChainMaker不允许用户直接调用该方法.
WASM_EXPORT void init_contract() {
    Counter counter;
    counter.init_contract();
}

// 在升级本合约时，对于每一个升级的版本调用一次upgrade方法. ChainMaker不允许用户直接调用该方法.
WASM_EXPORT void upgrade() {
    Counter counter;
    counter.upgrade();
}

WASM_EXPORT void save() {
    Counter counter;
    counter.save();
}

WASM_EXPORT void find_by_file_hash() {
    Counter counter;
    counter.find_by_file_hash();
```

```
}
```

存证合约代码说明

参数名称	描述
init_contract	合约的初始化函数，在合约部署时被调用，在本合约中为空。
upgrade	合约升级时调用的函数，在本合约中为空。
save	save 函数实现将文件哈希和文件名称记录到链上的功能。 1. save 函数先通过[参数处理]API 接口 arg 函数拿到时间，文件哈希和文件名字等信息。 2. 通过调用 [其他辅助类]API 接口 emit_event 函数发送标识为 topic_vx 的合约事件。 3. 通过调用 [账本交互]API 接口 put_object 函数将文件等信息记录到链上。
findByFileHash	通过文件哈希查询该条记录。 1. find_by_file_hash 通过 context 和 [参数处理]API 接口 arg 函数拿到要查找的文件的文件哈希。 2. 通过 [账本交互]API 接口 get_object 函数获取文件的信息，通过 [其他辅助类]API 接口 log 函数和 success 函数记录相关日志并返回结果。

证书申请
证书申请 CSR 生成指南

最近更新时间：2025-05-28 11:24:42

操作场景

本文介绍对应长安链区块链网络**非国密 ECC 证书**和**国密 SM2证书**申请证书请求文件 CSR 生成的步骤，请结合您的实际情况通过以下两种方式生成 CSR：

- 非国密 ECC 证书申请 CSR
- 国密 SM2 证书申请 CSR

操作步骤

非国密 ECC 证书申请 CSR

1. 前往 OpenSSL 官网，下载 openssl 并配置安装。
2. 下载 cmecccsr 工具 并解压。
3. 执行以下命令，生成对应文件。

```
sh ecccsr.sh
```

该命令会生成以下四个文件：

- `user_ecc_sign.key`：为用户证书对应私钥，需安全保存，支持在 SDK 中使用。
- `user_ecc_sign.csr`：用于在 TBaaS 控制台 申请用户证书。
- `user_ecc_tls.key`：为用户 tls 证书对应私钥，需安全保存，支持在 SDK 中使用。
- `user_ecc_tls.csr`：用于在 TBaaS 控制台 申请用户 tls 证书。

工具说明

以下为工具中主要使用的命令：

1. 生成用户证书对应私钥和 CSR 文件
   - **生成密钥对**：生成的 `temp` 文件为用户证书对应私钥。

     ```
     openssl ecparam -name prime256v1 -genkey -out temp
     ```

   - **生成用户证书 CSR 文件**：命令中使用的 `openssl_user.cnf` 文件已包含在下载工具中。

     ```
     openssl req -batch -config openssl_user.cnf -key temp -new -sha256 -out user_ecc_sign.csr
     ```

   - **转换私钥格式**：将已生成的 `temp` 私钥转换为 `pkcs#8` 格式的 `user_ecc_sign.key` 文件，后续用于 chainmaker-sdk 的配置和识别。

     ```
     openssl pkcs8 -topk8 -in temp -nocrypt -out user_ecc_sign.key
     ```

2. 生成用户 tls 证书对应私钥和 CSR 文件
   - **生成密钥对**：生成的 `temp` 文件为用户tls证书对应私钥。

     ```
     openssl ecparam -name prime256v1 -genkey -out temp
     ```

   - **生成用户 tls 证书 CSR 文件**：命令中使用的 `openssl_user.cnf` 文件已包含在下载工具中。

     ```
     openssl req -batch -config openssl_user.cnf -key temp -new -sha256 -out user_ecc_tls.csr
     ```

   - **转换私钥格式**：将已生成的 `temp` 私钥转换为 `pkcs#8` 格式的 `ser_ecc_tls.key` 文件，后续用于 chainmaker-sdk 的配置和识别。

```
openssl pkcs8 -topk8 -in temp -nocrypt -out user_ecc_tls.key
```

国密 SM2 证书申请 CSR

1. 前往 GmSSL 官网，下载 gmssl 并配置安装。
2. 下载 cmsm2csr 工具 并解压。
3. 执行以下命令，生成对应文件。

```
sh sm2csr.sh
```

该命令会生成以下四个文件：

- `user_sm2_sign.key`：为用户证书对应私钥，需安全保存，支持在 SDK 中使用。
- `user_sm2_sign.csr`：用于在 TBaaS 控制台 申请用户证书。
- `user_sm2_tls.key`：为用户 tls 证书对应私钥，需安全保存，支持在 SDK 中使用。
- `user_sm2_tls.csr`：用于在 TBaaS 控制台 申请用户 tls 证书。

工具说明

以下为工具中主要使用的命令：

1. 生成用户证书对应私钥和 CSR 文件
   - **生成密钥对**：生成的 `temp` 文件为用户证书对应私钥。

     ```
     gmssl ecparam -name sm2p256v1 -genkey -out temp
     ```

   - **生成用户证书 CSR 文件**：命令中使用的 `gmssl_user.cnf` 文件已包含在下载工具中。

     ```
     gmssl req -batch -config gmssl_user.cnf -key temp -new -sm3 -out user_sm2_sign.csr
     ```

   - **转换私钥格式**：将已生成的 `temp` 私钥转换为 `pkcs#8` 格式的 `user_sm2_sign.key` 文件，后续用于 chainmaker-sdk 的配置和识别。

     ```
     gmssl pkcs8 -topk8 -in temp -nocrypt -out user_sm2_sign.key
     ```

2. 生成用户 tls 证书对应私钥和 CSR 文件
   - **生成密钥对**：生成的 `temp` 文件为用户tls证书对应私钥。

     ```
     gmssl ecparam -name sm2p256v1 -genkey -out temp
     ```

   - **生成用户 tls 证书 CSR 文件**：命令中使用的 `gmssl_user.cnf` 文件已包含在下载工具中。

     ```
     gmssl req -batch -config gmssl_user.cnf -key temp -new -sm3 -out user_sm2_tls.csr
     ```

   - **转换私钥格式**：将已生成的 `temp` 私钥转换为 `pkcs#8` 格式的 `user_sm2_tls.key` 文件，后续用于 chainmaker-sdk 的配置和识别。

     ```
     gmssl pkcs8 -topk8 -in temp -nocrypt -out user_sm2_tls.key
     ```

长安链原生 SDK 证书配置示例

您可参考以下代码，在长安链原生 SDK chainmaker-sdk-go 中使用生成的私钥及已下载的证书配置 sdk_config.yml 文件，以非国密 ECC 证书为例。

Go SDK

```
chain_client:
  # 链ID
```

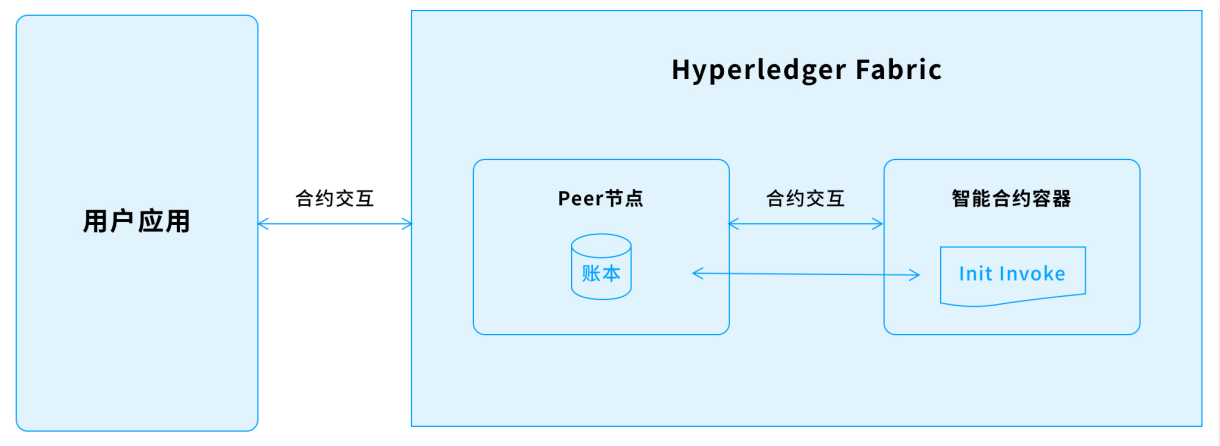
```
chain_id: "chain_txtxt"
# 组织ID
org_id: "orgtxtxtxt.chainmaker-txtxtxtxtx"
# 客户端用户tls私钥路径
user_key_file_path: "./user_ecc_tls.key"
# 客户端用户tls证书路径
user_crt_file_path: "./user_tls.crt"
# 客户端用户交易签名私钥路径(若未设置，将使用user_key_file_path)
user_sign_key_file_path: "./user_ecc_sign.key"
# 客户端用户交易签名证书路径(若未设置，将使用user_crt_file_path)
user_sign_crt_file_path: "./user_sign.crt"

nodes:
  - # 节点地址，格式为：IP:端口:连接数
    node_addr: "orgtxtxtxt.chainmaker-txtxtxtxtxt.baas.tech:8080"  #外网域名
    # 节点连接数
    conn_cnt: 1
    # RPC连接是否启用双向TLS认证
    enable_tls: true
    # 信任证书池路径
    trust_root_paths: # 包含组织根证书ca.crt的目录
      - "./ca"
    # 节点 TLS hostname
    tls_host_name: "common1-orgtxtxtxt.chainmaker-txtxtxtxtx"
```

Hyperledger Fabric 合约开发 智能合约简介

最近更新时间：2023-11-20 17:25:55

概述

在 Fabric 中，Chaincode（链码）也称为智能合约，是使用高级语言编写的一段代码，主要用于读取和操作账本上的数据。智能合约是一座连接用户应用和 Fabric 账本的桥梁。用户的应用可直接和智能合约打交道，智能合约可直接和 Fabric 账本进行交互。
一个完整的 Fabric 区块链应用包含用户的应用程序和用户编写的智能合约两部分。用户的应用程序通过区块链网络的 Peer 节点，调用智能合约。用户智能合约通过区块链网络的 Peer 节点，操作账本数据。如下图所示：



在此架构中，智能合约主要负责封装与账本直接交互的过程（包括按照用户指定的逻辑存储与查询账本数据），供用户应用程序调用。用户的应用程序主要负责以下职责：

- 用户的应用程序根据业务逻辑，负责生成需要存储在区块链网络上的数据。
- 用户的应用程序根据业务逻辑，从区块链网络上获取到相关数据，进行业务操作。

在 Fabric 中，用户编写的智能合约运行在隔离的沙盒环境中，目前主要以独立的 docker 容器展现，安装在区块链网络中的 Peer 节点上。Peer 节点通过启动 docker 容器，运行智能合约。当智能合约容器启动后，智能合约被调用时，智能合约会与启动智能合约容器的 Peer 节点建立交互，实现在对应 Peer 节点上账本相关数据的操作。不同的智能合约命名空间不一样，互相之间的数据也是独立的，智能合约只能访问到属于自己命名空间的键值对。如果一个智能合约想访问其他智能合约的数据，可以通过在智能合约内部调用其他智能合约的方式来实现。
Fabric 中的 Peer 节点提供了调用智能合约相关服务的接口。用户的应用程序可以通过使用此服务和 Fabric Peer 进行交互。通过与智能合约容器进行交互，完成应用程序和智能合约之间的交互。用户的应用程序可以通过以下两种方式使用 Fabric Peer 提供的服务：

- 利用 Fabric 提供的 SDK 与 Fabric Peer 进行交互，详情请参见 Fabric SDK 。
- 使用 腾讯云 SDK 调用相关的 TBaaS 云 API 接口 与 TBaaS 中的 Fabric Peer 进行交互。

注意事项

编写智能合约的时候，需要注意以下事项：

- **智能合约对非确定性代码处理。**
  在 Fabric 的交易中，交易请求将根据用户的配置，发送到多个组织的多个节点进行背书确认。如果智能合约中包含非确定性代码（例如随机数），将导致每个背书节点的运行结果不一致，无法达成共识。用户在编写智能合约时，建议不要在智能合约中包含随机函数。
- **智能合约运行无状态性。**
  编写智能合约时，建议不要在智能合约内部使用全局变量，建议设计为无状态性。在 Fabric 交易中，交易请求将根据用户的配置，发送到多个组织的多个节点进行背书确认。如果智能合约业务逻辑对全局变量有依赖，可能会导致每个背书节点运行结果不一致，无法达成共识。
- **智能合约初始化代码。**
  编写智能合约时，需要实现 Init 函数，此函数在智能合约实例化和升级的操作中，均会默认被执行一次。如果 Init 函数中的内容只能执行一次，升级的智能合约则不能包含此部分的 Init 代码。例如，在正常使用过程中，已将 Init 函数中设置的需写入账本的初始键值对 KEY1/VALUE1 修改为 KEY1/VALUE1_NEW，但是升级的智能合约默认调用 Init 函数，导致账本中的 KEY1 被更新为 VALUE1。为了避免产生此类似问题，建议将相关的 Init 内容写成独立的函数，供 Invoke 调用，在升级智能合约时，则无需变动 Init 的代码。

v2.3
Golang
合约打包说明

最近更新时间：2023-11-20 17:25:55

Go 语言合约示例包请参见 fabric-contract-go 。

Go 语言合约打包步骤

1. 编写 Go 语言合约文件。
2. 初始化 go.mod，执行命令 `go mod init fabric-contract-go-demo` 。
3. 在 go.mod 文件中添加 fabric-contract-api-go 依赖，示例如下：

```
module fabric-contract-go-demo

require github.com/hyperledger/fabric-contract-api-go v1.1.0

go 1.14
```

4. 使用命令 `go mod vendor` 将合约依赖下载到 vendor 目录中。
5. 在合约根目录中执行命令 `zip -r xxx.zip *` 将合约进行打包。

合约 API 列表

最近更新时间：2023-11-20 17:25:55

Fabric Go 语言版本智能合约有丰富的 API 接口，代码实现详情请参见 API 接口代码实现 。
从逻辑方面划分，可将 type 为 ChaincodeStub 的 API 划分为以下类型：

- 交易信息提取
- 账本交互
- 参数处理
- 其他辅助类

交易信息提取

接口	说明
GetBinding() ([]byte, error)	返回交易的 binding 信息
GetChannelID() string	获取当前的通道名称
GetCreator() ([]byte, error)	获取交易提交者信息
GetDecorations() map[string][]byte	获取交易的额外信息
GetSignedProposal() (*pb.SignedProposal, error)	获取交易提案相关数据
GetTransient() (map[string][]byte, error)	获取交易的临时信息
GetTxID() string	获取交易的交易 ID
GetTxTimestamp() (*timestamp.Timestamp, error)	获取交易时间戳

账本交互

接口	说明
PutState(key string, value []byte) error	在账本中添加或者更新一对键值
GetState(key string) ([]byte, error)	获取指定键对应的值
DelState(key string) error	在账本中，删除对应的键值
GetStateByRange(startKey, endKey string) (StateQueryIteratorInterface, error)	查询指定范围内的键值
GetStateByRangeWithPagination(startKey, endKey string, pageSize int32, bookmark string) (StateQueryIteratorInterface, *pb.QueryResponseMetadata, error)	分页查询指定范围内的键值
GetStateByPartialCompositeKey(objectType string, attributes []string) (StateQueryIteratorInterface, error)	查询匹配局部复合键的所有键值
GetStateByPartialCompositeKeyWithPagination(objectType string, keys []string, pageSize int32, bookmark string) (StateQueryIteratorInterface, *pb.QueryResponseMetadata, error)	分页查询匹配局部复合键的所有键值
GetQueryResult(query string)(StateQueryIteratorInterface, error)	查询状态数据库，需要支持富查询功能的状态数据库
GetQueryResultWithPagination(query string, pageSize int32, bookmark string) (StateQueryIteratorInterface, *pb.QueryResponseMetadata, error)	分页查询状态数据库，需要支持富查询功能的状态数据库
GetHistoryForKey(key string) (HistoryQueryIteratorInterface, error)	返回对应键的所有历史值
SetStateValidationParameter(key string, ep []byte) error	设置特定键的背书策略

GetStateValidationParameter(key string) ([]byte, error)	获取特定键的背书策略
GetPrivateData(collection, key string) ([]byte, error)	获取指定私有数据集中的键的值
GetPrivateDataHash(collection, key string) ([]byte, error)	获取指定私有数据集中的键的值的 hash
PutPrivateData(collection string, key string, value []byte) error	设置指定私有数据集中键的值
DelPrivateData(collection, key string) error	删除指定私有数据集中对应的键
SetPrivateDataValidationParameter(collection, key string, ep []byte) error	设置指定私有数据集中键的背书策略
GetPrivateDataValidationParameter(collection, key string) ([]byte, error)	获取指定私有数据集中键的背书策略
GetPrivateDataByRange(collection, startKey, endKey string) (StateQueryIteratorInterface, error)	获取指定私有数据集中特定范围键的键值
GetPrivateDataByPartialCompositeKey(collection, objectType string, keys []string) (StateQueryIteratorInterface, error)	获取指定私有数据集中匹配局部复合键的键值
GetPrivateDataQueryResult(collection, query string) (StateQueryIteratorInterface, error)	获取指定私有数据集中特定查询的键值，需要支持富查询功能的状态数据库

参数处理

接口	说明
GetArgs() [][]byte	获取智能合约中调用参数
GetArgsSlice() ([]byte, error)	获取智能合约中调用参数
GetStringArgs() []string	获取智能合约中调用参数
GetFunctionAndParameters() (function string, params []string)	获取智能合约调用的函数名和参数， 默认第一个参数为函数名

其他辅助类

接口	说明
CreateCompositeKey(objectType string, attributes []string) (string, error)	组合属性，形成复合建
SplitCompositeKey(compositeKey string) (string, []string, error)	拆分复合键成一系列属性
InvokeChaincode(chaincodeName string, args [][]byte, channel string) pb.Response	调用其它智能合约 Invoke 方法
SetEvent(name string, payload []byte) error	设置发送的事件

合约示例

最近更新时间：2023-11-20 17:25:55

基本示例

本示例以一个基本的智能合约用例为例，只包含智能合约的必须部分，没有实现任何业务逻辑。

```
package main

import (
   "errors"
   "fmt"
   "strconv"

   "github.com/hyperledger/fabric-contract-api-go/contractapi"
)

// ABstore Chaincode implementation
type ABstore struct {
   contractapi.Contract
}


func main() {
   cc, err := contractapi.NewChaincode(new(ABstore))
   if err != nil {
      panic(err.Error())
   }
   if err := cc.Start(); err != nil {
      fmt.Printf("Error starting ABstore chaincode: %s", err)
   }
}
```

官方示例

Hyperledger Fabric 提供了很多官方智能合约样例，详情请参见 fabric 官方示例 。本示例以 Hyperledger Fabric 官方提供的 ABstore 样例为例。该示例的 Init 函数用于初始化两个 key/value 键值对，Invoke 函数用于根据不同业务逻辑进行细分调用，最终调用以下业务逻辑接口：

1. Init：用于初始化键值对。
2. Invoke：用于 key 之间的 value 转移。
3. Delete：用于删除一个键值对。
4. Query：用于查询 key 所对应的值。

Init 函数示例

Init 函数在智能合约实例化以及升级的时候会被调用。本例通过调用 API PutState 将数据写到账本中。具体代码如下：

```
// Init用于初始化两个键值对，用户输入的参数为KEY1_NAME, VALUE1, KEY2_NAME, VALUE2
func (t *ABstore) Init(ctx contractapi.TransactionContextInterface, A string, Aval int, B string, Bval
int) error {
   fmt.Println("ABstore Init")
   var err error
   // Initialize the chaincode
   fmt.Printf("Aval = %d, Bval = %d\n", Aval, Bval)
   // Write the state to the ledger
   err = ctx.GetStub().PutState(A, []byte(strconv.Itoa(Aval)))
   if err != nil {
      return err
```

```
    }

    err = ctx.GetStub().PutState(B, []byte(strconv.Itoa(Bval)))
    if err != nil {
        return err
    }

    return nil
}
```

Invoke 函数示例

业务逻辑 invoke 函数主要用于实现业务逻辑中的资产转移。本例中通过调用 API GetState 获取到 KEY 对应的资产总值，通过调用用户业务逻辑实现资产转移，通过调用 API PutState 将用户最终资产写入账本。具体代码如下：

```
// invoke实现两个键之间的value转移，输入为KEY1_NAME, KEY2_NAME, VALUE
func (t *ABstore) Invoke(ctx contractapi.TransactionContextInterface, A, B string, X int) error {
    var err error
    var Aval int
    var Bval int
    // Get the state from the ledger
    // TODO: will be nice to have a GetAllState call to ledger
    Avalbytes, err := ctx.GetStub().GetState(A)
    if err != nil {
        return fmt.Errorf("Failed to get state")
    }
    if Avalbytes == nil {
        return fmt.Errorf("Entity not found")
    }
    Aval, _ = strconv.Atoi(string(Avalbytes))

    Bvalbytes, err := ctx.GetStub().GetState(B)
    if err != nil {
        return fmt.Errorf("Failed to get state")
    }
    if Bvalbytes == nil {
        return fmt.Errorf("Entity not found")
    }
    Bval, _ = strconv.Atoi(string(Bvalbytes))

    // Perform the execution
    Aval = Aval - X
    Bval = Bval + X
    fmt.Printf("Aval = %d, Bval = %d\n", Aval, Bval)

    // Write the state back to the ledger
    err = ctx.GetStub().PutState(A, []byte(strconv.Itoa(Aval)))
    if err != nil {
        return err
    }

    err = ctx.GetStub().PutState(B, []byte(strconv.Itoa(Bval)))
    if err != nil {
        return err
    }

    return nil
}
```

Delete 函数示例

业务逻辑 delete 函数主要用于实现业务逻辑中的账户删除功能，本例通过调用 API DelState 删除对应账户。具体代码如下：

```
// delete用于从账本中删除指定的键，输入为KEY_NAME
func (t *ABstore) Delete(ctx contractapi.TransactionContextInterface, A string) error {

   // Delete the key from the state in ledger
   err := ctx.GetStub().DelState(A)
   if err != nil {
      return fmt.Errorf("Failed to delete state")
   }

   return nil
}
```

Query 函数示例

业务逻辑 query 函数主要用于实现业务逻辑中的账户查询功能，本例通过调用 API GetState 查询对应账户的资产。具体代码如下：

```
// query主要是查询键对应的值，输入为KEY_NAME
func (t *ABstore) Query(ctx contractapi.TransactionContextInterface, A string) (string, error) {
   var err error
   // Get the state from the ledger
   Avalbytes, err := ctx.GetStub().GetState(A)
   if err != nil {
      jsonResp := "{\"Error\":\"Failed to get state for " + A + "\"}"
      return "", errors.New(jsonResp)
   }

   if Avalbytes == nil {
      jsonResp := "{\"Error\":\"Nil amount for " + A + "\"}"
      return "", errors.New(jsonResp)
   }

   jsonResp := "{\"Name\":\"" + A + "\",\"Amount\":\"" + string(Avalbytes) + "\"}"
   fmt.Printf("Query Response:%s\n", jsonResp)
   return string(Avalbytes), nil
}
```

Java
合约打包说明

最近更新时间：2023-11-20 17:25:56

Java 语言合约示例包请参见 fabric-contract-java 。

Java 合约打包

1. 在 pom.xml 文件中添加如下所示的本地仓库配置：

```
<repositories>
    <repository>
        <id>my-local-repo</id>
        <url>file://${project.basedir}/repository</url>
        <snapshots>
            <enabled>true</enabled>
            <updatePolicy>never</updatePolicy>
            <checksumPolicy>ignore</checksumPolicy>
        </snapshots>
        <releases>
            <enabled>true</enabled>
            <updatePolicy>never</updatePolicy>
            <checksumPolicy>ignore</checksumPolicy>
        </releases>
    </repository>
    <repository>
        <id>central</id>
        <url>https://repo.maven.apache.org/maven2</url>
        <snapshots>
            <enabled>true</enabled>
            <updatePolicy>never</updatePolicy>
            <checksumPolicy>ignore</checksumPolicy>
        </snapshots>
        <releases>
            <enabled>true</enabled>
            <updatePolicy>never</updatePolicy>
            <checksumPolicy>ignore</checksumPolicy>
        </releases>
    </repository>
</repositories>
```

2. 在 pom.xml 文件中添加如下所示的 maven 插件配置，注意使用指定的版本。

```
<build>
    <plugins>
        <plugin>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.1</version>
            <configuration>
                <source>${java.version}</source>
                <target>${java.version}</target>
            </configuration>
        </plugin>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-shade-plugin</artifactId>
            <version>3.1.0</version>
```

```
            <executions>
                <execution>
                    <phase>package</phase>
                    <goals>
                        <goal>shade</goal>
                    </goals>
                    <configuration>
                        <descriptorRefs>
                            <descriptorRef>jar-with-dependencies</descriptorRef>
                        </descriptorRefs>
                        <finalName>chaincode</finalName>
                        <transformers>
                            <transformer

implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransformer">
                                <mainClass>chaincode.example.SimpleChaincode</mainClass>
                            </transformer>
                        </transformers>
                        <filters>
                            <filter>
                                <!-- filter out signature files from signed dependencies, else
repackaging
                                     fails with security ex -->
                                <artifact>*:*</artifact>
                                <excludes>
                                    <exclude>META-INF/*.SF</exclude>
                                    <exclude>META-INF/*.DSA</exclude>
                                    <exclude>META-INF/*.RSA</exclude>
                                </excludes>
                            </filter>
                        </filters>
                    </configuration>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

3. 在 pom.xml 中添加相关依赖，示例如下：

```
<dependencies>
    <dependency>
        <groupId>org.hyperledger.fabric-chaincode-java</groupId>
        <artifactId>fabric-chaincode-protos</artifactId>
        <version>2.3.1</version>
    </dependency>
    <dependency>
        <groupId>org.hyperledger.fabric-chaincode-java</groupId>
        <artifactId>fabric-chaincode-shim</artifactId>
        <version>2.3.1</version>
        <exclusions>
            <exclusion>
                <groupId>com.github.everit-org.json-schema</groupId>
                <artifactId>org.everit.json.schema</artifactId>
            </exclusion>
        </exclusions>
    </dependency>
    <dependency>
        <groupId>org.projectlombok</groupId>
```

```
        <artifactId>lombok</artifactId>
        <version>1.18.16</version>
    </dependency>
</dependencies>
```

4. 使用如下命令将合约依赖下载到 repository 目录中。

```
  mvn dependency:copy-dependencies -DoutputDirectory=repository -Dmdep.useRepositoryLayout=True -
Dmdep.copyPom=True
```

5. 在合约根目录中使用 `zip -r xxx.zip *` 命令将合约进行打包。

合约 API 列表

最近更新时间：2023-11-20 17:25:56

Fabric Java 语言版本智能合约有丰富的 API 接口，具体的代码实现可以参考 API 接口代码实现 。从逻辑上划分，可将 API 接口分为以下类型：

- 交易信息提取
- 账本交互
- 参数处理
- 其他辅助类

交易信息提取

接口	说明
byte[] getBinding()	返回交易的 binding 信息
String getChannelId()	获取当前的通道名称
byte[] getCreator()	获取交易提交者信息
SignedProposal getSignedProposal()	获取交易提案相关数据
Map<String, byte[]> getTransient()	获取交易的临时信息
String getTxId()	获取交易的交易 ID
Instant getTxTimestamp()	获取交易时间戳

账本交互

接口	说明
void putState(String key, byte[] value)	在账本中添加或者更新一对键值
void putStringState(String key, String value)	在账本中添加或者更新一对键值
byte[] getState(String key)	获取指定键对应的值
String getStringState(String key)	获取指定键对应的值
void delState(String key)	在账本中删除对应的键值
QueryResultsIterator<KeyValue> getStateByRange(String startKey, String endKey)	查询指定范围内的键值
QueryResultsIteratorWithMetadata<KeyValue> getStateByRangeWithPagination(String startKey, String endKey, int pageSize, String bookmark)	分页查询指定范围内的键值
QueryResultsIterator<KeyValue> getStateByPartialCompositeKey(String compositeKey)	查询匹配局部复合键的所有键值
QueryResultsIterator<KeyValue> getStateByPartialCompositeKey(String objectType, String... attributes)	查询匹配局部复合键的所有键值
QueryResultsIterator<KeyValue> getStateByPartialCompositeKey(CompositeKey compositeKey)	查询匹配局部复合键的所有键值
QueryResultsIteratorWithMetadata<KeyValue> getStateByPartialCompositeKeyWithPagination(CompositeKey compositeKey, int pageSize, String bookmark)	分页查询匹配局部复合键的所有键值
QueryResultsIterator<KeyValue> getQueryResult(String query)	查询状态数据库，需要支持富查询功能的状态数据库

QueryResultsIteratorWithMetadata<KeyValue> getQueryResultWithPagination(String query, int pageSize, String bookmark)	分页查询状态数据库，需要支持富查询功能的状态数据库
QueryResultsIterator<KeyModification> getHistoryForKey(String key)	返回对应键的所有历史值
void setStateValidationParameter(String key, byte[] value)	设置特定键的背书策略
byte[] getStateValidationParameter(String key)	获取特定键的背书策略
byte[] getPrivateData(String collection, String key)	获取指定私有数据集集中的键的值
String getPrivateDataUTF8(String collection, String key)	获取指定私有数据集集中的键的值
byte[] getPrivateDataHash(String collection, String key)	获取指定私有数据集集中的键的值的 hash
void putPrivateData(String collection, String key, byte[] value)	设置指定私有数据集集中键的值
void putPrivateData(String collection, String key, String value)	设置指定私有数据集集中键的值
void delPrivateData(String collection, String key)	删除指定私有数据集集中对应的键
void setPrivateDataValidationParameter(String collection, String key, byte[] value)	设置指定私有数据集集中键的背书策略
byte[] getPrivateDataValidationParameter(String collection, String key)	获取指定私有数据集集中键的背书策略
QueryResultsIterator<KeyValue> getPrivateDataByRange(String collection, String startKey, String endKey)	获取指定私有数据集集中特定范围键的键值
QueryResultsIterator<KeyValue> getPrivateDataByPartialCompositeKey(String collection, String compositeKey)	获取指定私有数据集集中匹配局部复合键的键值
QueryResultsIterator getPrivateDataByPartialCompositeKey(String collection, CompositeKey compositeKey)	获取指定私有数据集集中匹配局部复合键的键值
QueryResultsIterator<KeyValue> getPrivateDataByPartialCompositeKey(String collection, String objectType, String... attributes)	获取指定私有数据集集中匹配局部复合键的键值
QueryResultsIterator<KeyValue> getPrivateDataQueryResult(String collection, String query)	获取指定私有数据集集中特定查询的键值，需要支持富查询功能的状态数据库

参数处理

接口	说明
List<byte[]> getArgs()	获取智能合约中调用参数
List<String> getStringArgs()	获取智能合约中调用参数
String getFunction()	获取智能合约调用的函数名，默认第一个参数为函数名
List<String> getParameters()	获取智能合约调用的参数

其他辅助类

接口	说明
CompositeKey createCompositeKey(String objectType, String... attributes)	组合属性，形成复合键
CompositeKey splitCompositeKey(String compositeKey)	拆分复合键成一系列属性
Response invokeChaincode(final String chaincodeName, final List<byte[]> args, final String channel)	调用其它智能合约 invoke 方法

Response invokeChaincode(String chaincodeName, List<byte[]> args)	调用其它智能合约 invoke 方法
Response invokeChaincodeWithStringArgs(String chaincodeName, List<String> args, String channel)	调用其它智能合约 invoke 方法
Response invokeChaincodeWithStringArgs(String chaincodeName, List<String> args)	调用其它智能合约 invoke 方法
Response invokeChaincodeWithStringArgs(final String chaincodeName, final String... args)	调用其它智能合约 invoke 方法
void setEvent(String name, byte[] payload)	设置发送的事件
ChaincodeEvent getEvent()	获取发送的事件

合约示例

最近更新时间：2023-11-20 17:25:56

基本示例

本示例以一个基本的智能合约用例为例，只包含智能合约的必须部分，没有实现任何业务逻辑。

```
/*
 * SimpleAssetDemo implements a simple chaincode to manage an asset
 */
public class SimpleAssetDemo extends ChaincodeBase {
   /*
    * Init is called during chaincode instantiation to initialize any data.
    */
    @Override
   public Response init(ChaincodeStub stub) {
   }
   /*
    * Invoke is called per transaction on the chaincode. Each transaction is
    * either a 'get' or a 'set' on the asset created by Init function. The 'set'
    * method may create a new asset by specifying a new key-value pair.
    */
    @Override
   public Response invoke(ChaincodeStub stub) {
   }

   public static void main(String[] args) {
        new SimpleAssetDemo().start(args);
   }
}
```

官方示例

Hyperledger Fabric 提供了很多官方的智能合约样例，具体请参考 fabric 官方示例 。本示例以 Hyperledger Fabric 官方提供的 ABstore 样例为例。该示例的 init 函数用于初始化两个 key/value 键值对，invoke 函数用于根据不同业务逻辑进行细分调用，最终调用以下业务逻辑接口：

- invoke：用于 key 之间的 value 转移。
- delete：删除一个键值对。
- query：查询 key 所对应的值。

init 函数示例

init 函数在智能合约实例化以及升级的时候会被调用。本例通过调用 API getFunction 和 getParameters 获取到用户输入参数。在获取用户输入参数后，通过调用 API putStringState 将数据写到账本中。具体代码如下：

```
/*
 * init函数用于初始化两个键值对，用户输入的参数为KEY1_NAME, VALUE1,
 * KEY2_NAME, VALUE2
 */
@Override
public Response init(ChaincodeStub stub) {
    try {
        _logger.info("Init java simple chaincode");
        List<String> args = stub.getParameters();
        if (args.size() != 4) {
            newErrorResponse("Incorrect number of arguments. Expecting 4");
        }
        // Initialize the chaincode
```

```
        String account1Key = args.get(0);
        int account1Value = Integer.parseInt(args.get(1));
        String account2Key = args.get(2);
        int account2Value = Integer.parseInt(args.get(3));

        _logger.info(String.format("account %s, value = %s; account %s, value %s", account1Key,
account1Value, account2Key, account2Value));
        stub.putStringState(account1Key, args.get(1));
        stub.putStringState(account2Key, args.get(3));

        return newSuccessResponse();
    } catch (Throwable e) {
        return newErrorResponse(e);
    }
}
```

invoke 函数示例

invoke 函数对用户的不同的智能合约业务逻辑进行拆分。本例通过调用 API getFunction 和 getParameters 获取到用户的具体业务类型和参数，根据用户的不同业务类型，分别调用不同的业务函数，如 invoke，delete 和 query 函数。具体代码如下：

```
// invoke把用户调用的function细分到几个子function, 包含invoke, delete和query
@Override
public Response invoke(ChaincodeStub stub) {
    try {
        _logger.info("Invoke java simple chaincode");
        String func = stub.getFunction();
        List<String> params = stub.getParameters();
        if (func.equals("invoke")) {
            return invoke(stub, params);
        }
        if (func.equals("delete")) {
            return delete(stub, params);
        }
        if (func.equals("query")) {
            return query(stub, params);
        }
        return newErrorResponse("Invalid invoke function name. Expecting one of: [\"invoke\",
\"delete\", \"query\"]");
    } catch (Throwable e) {
        return newErrorResponse(e);
    }
}
```

业务逻辑 invoke 函数示例

业务逻辑 invoke 函数主要用于实现业务逻辑中的资产转移。本例中通过调用 API getStringState 获取到 KEY 对应的资产总值，通过调用用户业务逻辑实现资产转移，通过调用 API putStringState 将用户最终资产写入账本。具体代码如下：

```
// invoke实现两个键之间的value转移，输入为KEY1_NAME, KEY2_NAME, VALUE
private Response invoke(ChaincodeStub stub, List<String> args) {
    if (args.size() != 3) {
        return newErrorResponse("Incorrect number of arguments. Expecting 3");
    }
    String accountFromKey = args.get(0);
    String accountToKey = args.get(1);

    String accountFromValueStr = stub.getStringState(accountFromKey);
    if (accountFromValueStr == null) {
```

```
            return newErrorResponse(String.format("Entity %s not found", accountFromKey));
        }
        int accountFromValue = Integer.parseInt(accountFromValueStr);

        String accountToValueStr = stub.getStringState(accountToKey);
        if (accountToValueStr == null) {
            return newErrorResponse(String.format("Entity %s not found", accountToKey));
        }
        int accountToValue = Integer.parseInt(accountToValueStr);

        int amount = Integer.parseInt(args.get(2));

        if (amount > accountFromValue) {
            return newErrorResponse(String.format("not enough money in account %s", accountFromKey));
        }

        accountFromValue -= amount;
        accountToValue += amount;

        _logger.info(String.format("new value of A: %s", accountFromValue));
        _logger.info(String.format("new value of B: %s", accountToValue));

        stub.putStringState(accountFromKey, Integer.toString(accountFromValue));
        stub.putStringState(accountToKey, Integer.toString(accountToValue));

        _logger.info("Transfer complete");

        return newSuccessResponse("invoke finished successfully", ByteString.copyFrom(accountFromKey + ": "
    + accountFromValue + " " + accountToKey + ": " + accountToValue, UTF_8).toByteArray());
    }
```

delete 函数示例

业务逻辑 delete 函数主要用于实现业务逻辑中的账户删除功能，本示例通过调用 API delState 删除对应账户。具体代码如下：

```
// delete用于从账本中删除指定的键，输入为KEY_NAME
private Response delete(ChaincodeStub stub, List<String> args) {
    if (args.size() != 1) {
        return newErrorResponse("Incorrect number of arguments. Expecting 1");
    }
    String key = args.get(0);
    // Delete the key from the state in ledger
    stub.delState(key);
    return newSuccessResponse();
}
```

query 函数示例

业务逻辑 query 函数主要用于实现业务逻辑中账户查询功能，本示例通过调用 API getStringState 查询对应账户的资产。具体代码如下：

```
// query主要是查询键对应的值，输入为KEY_NAME
private Response query(ChaincodeStub stub, List<String> args) {
    if (args.size() != 1) {
        return newErrorResponse("Incorrect number of arguments. Expecting name of the person to query");
    }
    String key = args.get(0);
    //byte[] stateBytes
    String val = stub.getStringState(key);
    if (val == null) {
```

```
            return newErrorResponse(String.format("Error: state for %s is null", key));
        }
        _logger.info(String.format("Query Response:\nName: %s, Amount: %s\n", key, val));
        return newSuccessResponse(val, ByteString.copyFrom(val, UTF_8).toByteArray());
    }
```

v1.4
合约 API 列表（Go）

最近更新时间：2021-11-19 10:36:21

Fabric Go 语言版本智能合约有丰富的 API 接口，代码实现详情可以参考 API 接口代码实现 。
从逻辑方面划分，可将 type 为 ChaincodeStub 的 API 划分为以下类型：

- 交易信息提取
- 账本交互
- 参数处理
- 其他辅助类

交易信息提取

接口	说明
GetBinding() ([]byte, error)	返回交易的 binding 信息
GetChannelID() string	获取当前的通道名称
GetCreator() ([]byte, error)	获取交易提交者信息
GetDecorations() map[string][]byte	获取交易的额外信息
GetSignedProposal() (*pb.SignedProposal, error)	获取交易提案相关数据
GetTransient() (map[string][]byte, error)	获取交易的临时信息
GetTxID() string	获取交易的交易 ID
GetTxTimestamp() (*timestamp.Timestamp, error)	获取交易时间戳

账本交互

接口	说明
PutState(key string, value []byte) error	在账本中添加或者更新一对键值
GetState(key string) ([]byte, error)	获取指定键对应的值
DelState(key string) error	在账本中，删除对应的键值
GetStateByRange(startKey, endKey string) (StateQueryIteratorInterface, error)	查询指定范围内的键值
GetStateByRangeWithPagination(startKey, endKey string, pageSize int32, bookmark string) (StateQueryIteratorInterface, *pb.QueryResponseMetadata, error)	分页查询指定范围内的键值
GetStateByPartialCompositeKey(objectType string, attributes []string) (StateQueryIteratorInterface, error)	查询匹配局部复合键的所有键值
GetStateByPartialCompositeKeyWithPagination(objectType string, keys []string, pageSize int32, bookmark string) (StateQueryIteratorInterface, *pb.QueryResponseMetadata, error)	分页查询匹配局部复合键的所有键值
GetQueryResult(query string)(StateQueryIteratorInterface, error)	查询状态数据库，需要支持富查询功能的状态数据库
GetQueryResultWithPagination(query string, pageSize int32, bookmark string) (StateQueryIteratorInterface, *pb.QueryResponseMetadata, error)	分页查询状态数据库，需要支持富查询功能的状态数据库
GetHistoryForKey(key string) (HistoryQueryIteratorInterface, error)	返回对应键的所有历史值

SetStateValidationParameter(key string, ep []byte) error	设置特定键的背书策略
GetStateValidationParameter(key string) ([]byte, error)	获取特定键的背书策略
GetPrivateData(collection, key string) ([]byte, error)	获取指定私有数据集中的键的值
GetPrivateDataHash(collection, key string) ([]byte, error)	获取指定私有数据集中的键的值的 hash
PutPrivateData(collection string, key string, value []byte) error	设置指定私有数据集中键的值
DelPrivateData(collection, key string) error	删除指定私有数据集中对应的键
SetPrivateDataValidationParameter(collection, key string, ep []byte) error	设置指定私有数据集中键的背书策略
GetPrivateDataValidationParameter(collection, key string) ([]byte, error)	获取指定私有数据集中键的背书策略
GetPrivateDataByRange(collection, startKey, endKey string) (StateQueryIteratorInterface, error)	获取指定私有数据集中特定范围键的键值
GetPrivateDataByPartialCompositeKey(collection, objectType string, keys []string) (StateQueryIteratorInterface, error)	获取指定私有数据集中匹配局部复合键的键值
GetPrivateDataQueryResult(collection, query string) (StateQueryIteratorInterface, error)	获取指定私有数据集中特定查询的键值，需要支持富查询功能的状态数据库

参数处理

接口	说明
GetArgs() [][]byte	以 byte 数组的形式获取智能合约中调用参数
GetArgsSlice() ([]byte, error)	以 byte 切片的形式获取智能合约中调用参数
GetStringArgs() []string	以字符串数组的形式获取智能合约中调用参数
GetFunctionAndParameters() (function string, params []string)	获取智能合约调用的函数名和参数， 默认第一个参数为函数名

其他辅助类

接口	说明
CreateCompositeKey(objectType string, attributes []string) (string, error)	组合属性，形成复合建
SplitCompositeKey(compositeKey string) (string, []string, error)	拆分复合键成一系列属性
InvokeChaincode(chaincodeName string, args [][]byte, channel string) pb.Response	调用其它智能合约 Invoke 方法
SetEvent(name string, payload []byte) error	设置发送的事件

合约示例（Go）

最近更新时间：2024-01-15 14:42:01

智能合约构成

Go 语言的智能合约代码主要由以下接口构成：

```
// Chaincode interface must be implemented by all chaincodes. The fabric runs
// the transactions by calling these functions as specified.
type Chaincode interface {
        // Init is called during Instantiate transaction after the chaincode container
        // has been established for the first time, allowing the chaincode to
        // initialize its internal data

        Init(stub ChaincodeStubInterface) pb.Response
        // Invoke is called to update or query the ledger in a proposal transaction.
        // Updated state variables are not committed to the ledger until the
        // transaction is committed.
        Invoke(stub ChaincodeStubInterface) pb.Response
}
```

- 接口 Init 主要用于智能合约初始化和升级智能合约时调用此接口，初始化相关的数据。
- 接口 Invoke 主要用于实现智能合约中的内部业务逻辑。用户可以根据实际需求，实现相关的业务。
- 在实现过程中，用户可以调用 ChaincodeStubInterface 的 API 接口和链上进行交互。

智能合约示例

基本示例

本示例以一个基本的智能合约用例为例，只包含智能合约的必须部分，没有实现任何业务逻辑。

```
package main
import (
    "github.com/hyperledger/fabric/core/chaincode/shim"
    "github.com/hyperledger/fabric/protos/peer"
)

// SimpleAssetDemo implements a simple chaincode to manage an asset
type SimpleAssetDemo struct {
}

// Init is called during chaincode instantiation to initialize any data.
func (t *SimpleAssetDemo) Init(stub shim.ChaincodeStubInterface) peer.Response {
    return shim.Success(nil)
}

// Invoke is called per transaction on the chaincode. Each transaction is
// either a 'get' or a 'set' on the asset created by Init function. The 'set'
// method may create a new asset by specifying a new key-value pair.
func (t *SimpleAssetDemo) Invoke(stub shim.ChaincodeStubInterface) peer.Response {
    return shim.Success(nil)
}

func main() {
    err := shim.Start(new(SimpleAssetDemo))
    if err != nil {
        fmt.Printf("Error starting SimpleAssetDemo chaincode: %s", err)
    }
```

```
}
```

官方示例

Hyperledger Fabric 提供了很多官方智能合约样例，具体请参考 fabric 官方示例 。本示例以 Hyperledger Fabric 官方提供的 example02 样例为例。该示例的 Init 函数用于初始化两个 key/value 键值对，Invoke 函数用于根据不同业务逻辑进行细分调用，最终调用以下业务逻辑接口：

- invoke：用于 key 之间的 value 转移。
- delete：用于删除一个键值对。
- query：用于查询 key 所对应的值。

Init 函数示例

Init 函数在智能合约实例化以及升级的时候会被调用。在实现 Init 函数的过程中，可使用 Go 语言版本的合约 API 列表 来对参数和账本进行操作。本例通过调用 API GetFunctionAndParameters 获取到用户输入参数。在获取用户输入参数后，通过调用 API PutState 将数据写到账本中。具体代码如下：

```
// Init用于初始化两个键值对，用户输入的参数为KEY1_NAME, VALUE1, KEY2_NAME, VALUE2
func (t *SimpleChaincode) Init(stub shim.ChaincodeStubInterface) pb.Response {
    fmt.Println("ex02 Init")
    // 调用API GetFunctionAndParameters 获取用户输入参数
    _, args := stub.GetFunctionAndParameters()
    var A, B string    // Entities
    var Aval, Bval int // Asset holdings
    var err error

    if len(args) != 4 {
        return shim.Error("Incorrect number of arguments. Expecting 4")
    }

    // Initialize the chaincode
    A = args[0]
    Aval, err = strconv.Atoi(args[1])
    if err != nil {
        return shim.Error("Expecting integer value for asset holding")
    }
    B = args[2]
    Bval, err = strconv.Atoi(args[3])
    if err != nil {
        return shim.Error("Expecting integer value for asset holding")
    }
    fmt.Printf("Aval = %d, Bval = %d\n", Aval, Bval)

    // 调用API PutState把数据写入账本
    err = stub.PutState(A, []byte(strconv.Itoa(Aval)))
    if err != nil {
        return shim.Error(err.Error())
    }

    err = stub.PutState(B, []byte(strconv.Itoa(Bval)))
    if err != nil {
        return shim.Error(err.Error())
    }

    return shim.Success(nil)
}
```

Invoke 函数示例

Invoke 函数对用户的不同的智能合约业务逻辑进行拆分。本例通过调用 API GetFunctionAndParameters 获取到用户的具体业务类型和参数，根据用户的不同业务类型，分别调用不同的业务函数，如 invoke，delete 和 query 函数。具体代码如下：

```
// Invoke把用户调用的function细分到几个子function, 包含invoke,delete和query
func (t *SimpleChaincode) Invoke(stub shim.ChaincodeStubInterface) pb.Response {
    fmt.Println("ex02 Invoke")
    // 调用API GetFunctionAndParameters获取用户输入的业务类型和参数
    function, args := stub.GetFunctionAndParameters()
    if function == "invoke" {
        // Make payment of X units from A to B
        return t.invoke(stub, args)
    } else if function == "delete" {
        // Deletes an entity from its state
        return t.delete(stub, args)
    } else if function == "query" {
        // the old "Query" is now implemtned in invoke
        return t.query(stub, args)
    }

    return shim.Error("Invalid invoke function name. Expecting \"invoke\" \"delete\" \"query\"")
}
```

invoke 函数示例

业务逻辑 invoke 函数主要用于实现业务逻辑中的资产转移。本例中通过调用 API GetState 获取到 KEY 对应的资产总值，通过调用用户业务逻辑实现资产转移，通过调用 API PutState 将用户最终资产写入账本。具体代码如下：

```
// invoke实现两个键之间的value转移，输入为KEY1_NAME, KEY2_NAME, VALUE
// Transaction makes payment of X units from A to B
func (t *SimpleChaincode) invoke(stub shim.ChaincodeStubInterface, args []string) pb.Response {
    var A, B string    // Entities
    var Aval, Bval int // Asset holdings
    var X int          // Transaction value
    var err error

    if len(args) != 3 {
        return shim.Error("Incorrect number of arguments. Expecting 3")
    }

    A = args[0]
    B = args[1]

    // API GetState获取对应账户的资产
    Avalbytes, err := stub.GetState(A)
    if err != nil {
        return shim.Error("Failed to get state")
    }
    if Avalbytes == nil {
        return shim.Error("Entity not found")
    }
    Aval, _ = strconv.Atoi(string(Avalbytes))

    Bvalbytes, err := stub.GetState(B)
    if err != nil {
        return shim.Error("Failed to get state")
    }
    if Bvalbytes == nil {
        return shim.Error("Entity not found")
```

```
    }
    Bval, _ = strconv.Atoi(string(Bvalbytes))

    // 执行具体业务逻辑，这里是对应资产进行转移
    // Perform the execution
    X, err = strconv.Atoi(args[2])
    if err != nil {
        return shim.Error("Invalid transaction amount, expecting a integer value")
    }
    Aval = Aval - X
    Bval = Bval + X
    fmt.Printf("Aval = %d, Bval = %d\n", Aval, Bval)

    // API PutState将对应资产写入账本
    // Write the state back to the ledger
    err = stub.PutState(A, []byte(strconv.Itoa(Aval)))
    if err != nil {
        return shim.Error(err.Error())
    }

    err = stub.PutState(B, []byte(strconv.Itoa(Bval)))
    if err != nil {
        return shim.Error(err.Error())
    }

    return shim.Success(nil)
}
```

delete 函数示例

业务逻辑 delete 函数主要用于实现业务逻辑中的账户删除功能，本例通过调用 API DelState 删除对应账户。具体代码如下：

```
// delete用于从账本中删除指定的键，输入为KEY_NAME
// Deletes an entity from state
func (t *SimpleChaincode) delete(stub shim.ChaincodeStubInterface, args []string) pb.Response {
    if len(args) != 1 {
        return shim.Error("Incorrect number of arguments. Expecting 1")
    }

    A := args[0]

    // API DelState删除特定的账户
    // Delete the key from the state in ledger
    err := stub.DelState(A)
    if err != nil {
        return shim.Error("Failed to delete state")
    }

    return shim.Success(nil)
}
```

query 函数示例

业务逻辑 query 函数主要用于实现业务逻辑中的账户查询功能，本例通过调用 API GetState 查询对应账户的资产。具体代码如下：

```
// query主要是查询键对应的值，输入为KEY_NAME
// query callback representing the query of a chaincode
func (t *SimpleChaincode) query(stub shim.ChaincodeStubInterface, args []string) pb.Response {
    var A string // Entities
```

```
    var err error

    if len(args) != 1 {
        return shim.Error("Incorrect number of arguments. Expecting name of the person to query")
    }

    A = args[0]
    // API GetState查询特定的账户
    // Get the state from the ledger
    Avalbytes, err := stub.GetState(A)
    if err != nil {
        jsonResp := "{\"Error\":\"Failed to get state for " + A + "\"}"
        return shim.Error(jsonResp)
    }

    if Avalbytes == nil {
        jsonResp := "{\"Error\":\"Nil amount for " + A + "\"}"
        return shim.Error(jsonResp)
    }

    jsonResp := "{\"Name\":\"" + A + "\",\"Amount\":\"" + string(Avalbytes) + "\"}"
    fmt.Printf("Query Response:%s\n", jsonResp)
    return shim.Success(Avalbytes)
}
```

国密算法支持说明（Go）

最近更新时间：2024-03-25 15:02:09

简介

在业务场景中，用户希望在智能合约层增加对国密算法 SM2，SM3 的支持。TBaaS 已在智能合约层引入了国密算法的能力，方便用户快速在智能合约层使用国密算法。

TBaaS 智能合约对国密算法的支持，主要体现在以下两个方面：

- TBaaS 提供了一个单独的用户业务层工具 gmtool，用户可以使用 gmtool 实现公私钥生成、加密、解密、签名以及验签等功能。
- TBaaS 在 Go 语言智能合约中，额外引入了 gmssl 包，增加 SM2 以及 SM3 算法的 API 接口，用户可以方便的在智能合约中集成国密算法。

以下将分别对 gmtool 和 Go 智能合约的 gmssl 包进行说明。

国密算法工具 gmtool

Gmtool 工具是国密算法的用户业务层工具，主要实现了公私钥生成、加密、解密、签名、验签等功能，您可访问 gmtool 进行下载。

使用说明

命令	描述	参数
genkey	生成 SM2 算法的公私钥对（加密、签名的公私钥对使用同一个算法）	pkout：新公钥的文件路径。 skout：新私钥的文件路径。
encrypt	加密	pkin：用于加密的公钥的文件路径，可以是 genkey 生成的公钥文件路径。 plaintext：要加密的明文文件路径。 cipherout：生成的密文的指定文件路径。
decrypt	解密	skin：解密使用的私钥的文件路径。 cipherin：需要解密的密文文件路径。
sign	签名	skin：签名使用的私钥的文件路径。 message：需要签发的信息的文件路径。 signature：生成的签名的指定文件路径。
verify	验签	pkin：验签使用的公钥文件路径。 message：验证的信息所在文件路径。 signature：验证的签名所在文件路径。

示例

- **genkey**
  执行以下命令，生成的新公钥存在 pk.sm2 文件中，新私钥存在 sk.sm2 文件中。公私钥均使用 PKCS8 协议存成 PEM 格式。

  > **说明**
  > 本示例使用同一套公私钥来执行加解密以及签名验签。在生产环境中，加密和签名的公私钥对必须都单独生成，不能混用。

  ```
  ./gmtool genkey -pkout=pk.sm2 -skout=sk.sm2
  ```

- **encrypt**
  执行以下命令，工具会使用 pk.sm2 中存储的公钥对 plain 文件中的内容进行加密，生成的密文存在文件 cipher.sm2 中。密文经过 base64 编码，需要解码后才能作为国密算法库解密算法的入参，但可以直接作为 gmtool 工具的解密功能入参。

  ```
  ./gmtool encrypt -pkin=pk.sm2 -plaintext=plain -cipherout=cipher.sm2
  ```

- **decrypt**
  执行以下命令，工具会解析文件 sk.sm2 中的私钥，对文件 cipher.sm2 中的密文进行解密，解密结果在控制台展示。

```
./gmtool decrypt -skin=sk.sm2 -cipherin=cipher.sm2
```

- **sign**
  执行以下命令，工具会解析文件 sk.sm2 中的私钥，对 message 文件中的内容用 SM2 签名算法和 SM3 哈希算法进行签名，并把签名存在文件 sig.sm2 中。签名已经过 base64 编码，可以直接被 gmtool 工具的验签功能验证，但需要解码后才能作为国密算法库验签算法的入参。
  工具中的摘要片段固定使用“1234567812345678”与入参 message 拼接，暂不支持自定义。这里与 chaincode 中的接口说明对应。

> **说明**
> 根据GM/T 0009-2012《SM2密码算法使用规范》，用户身份标识ID的长度为16字节，其默认值从左到右依次为1234567812345678。

```
./gmtool sign -skin=sk.sm2 -message=message -signature=sig.sm2
```

- **verify**
  执行以下命令，在控制台显示验签结果。通过则为 “Valid Signature”，不通过为 “Invalid Signature”。

```
./gmtool verify -pkin=pk.sm2 -message=message -signature=sig.sm2
```

Go 语言智能合约 gmssl 包接口说明

Go 语言智能合约 gmssl 包 支持 SM2，SM3 算法的相关接口，用户可以很方便的在智能合约中直接 import gmssl 包，使用相关的接口。

gmssl 包支持接口

私钥相关函数

接口类型	函数名	说明
非成员函数	GeneratePrivateKey(alg string, args map[string]string, engine *Engine) (*PrivateKey, error)	功能：生成私钥。
		参数： • alg：string 类型的算法名称，SM2 算法固定为 “EC”。 • args：map[string]string 类型算法参数，SM2 算法固定为： `map[string]string{ "ec_paramgen_curve": "sm2p256v1", "ec_param_enc": "named_curve", }` • engine：SM2 加密或签名算法固定入参为 nil。
		输出： • 私钥：用于解密或签名，可导出成员 PublicKey 作为公钥进行加密。 • 错误信息。
	NewPrivateKeyFromPEM(pem string, password string) (*PrivateKey, error)	功能：从 PEM 格式字符串中读取私钥。
		参数： • pem：string 类型，符合 PEM 格式的字符串。 • password：string 类型，若私钥 PEM 经过加密保护，则在这里传入加密时使用的口令。若没有加密，则传入空字符串“”。
		输出： • 私钥：用于解密或签名，可导出成员 PublicKey 作为公钥进行加密。 • 错误信息。

成员函数	(*PrivateKey) GetPEM(cipher string, password string) (string, error)	功能：生成加密保护的私钥 PEM 格式字符串。
		参数： • cipher：string 类型，加密私钥 PEM 的算法，可选 “SM4”、“AES256” 及其它 openssl 支持的对称加密算法。 • password：string 类型，加密私钥使用的口令，用于扩展生成对称加密的密钥。
		输出： • PEM：string 类型。 • 错误信息。
	(*PrivateKey) GetUnencryptedPEM() (string, error)	功能： 生成不加密的私钥 PEM 格式字符串。
		输出： • PEM：string 类型。 • 错误信息。
	(*PrivateKey) GetPublicKeyPEM() (string, error)	功能：从私钥中抽取对应的公钥。
		输出： • PEM：string 类型，公钥的 PEM 格式字符串，未经加密。 • 错误信息。
	(*PrivateKey) Decrypt(alg string, ciphertext []byte, engine *Engine) ([]byte, error)	功能：解密，用私钥 sk 从密文中获取明文。
		参数： • alg：string 类型，算法名称。SM2 加密算法固定入参为 “sm2encrypt-with-sm3”。 • ciphertext：[]byte 类型，密文（由 gmtool 工具或者调用公钥成员函数 Encrypt 生成）。 • engine：SM2 算法固定入参 nil。
		输出： • 明文：[]byte 类型明文字符串。 • 错误信息。
	(*PrivateKey) Sign(alg string, dgst []byte, engine *Engine) ([]byte, error)	功能：针对入参中的信息摘要 dgst 生成签名。
		参数： • alg：string 类型，算法名称。SM2 签名算法固定入参为 “sm2sign”。 • dgst：[]byte 类型，信息摘要。将要签名的信息和 ComputeSM2IDDigest 函数生成的部分摘要，用选定的哈希算法算出最终摘要，作为此项入参。 • engine： SM2 算法固定入参为 nil。
		输出： • 签名：[]byte 类型。 • 错误信息。
	(*PrivateKey) ComputeSM2IDDigest(id string) ([]byte, error)	功能：对公钥和入参 id 生成特定摘要片段，用于签名时与信息生成最终摘要。
		参数： id：用户入参 id，可看做用户选择的公钥的一部分，在签名和验签时 id 相同才能正确验证。在 fabric 国密算法证书系统和 gmtool 工具中，默认使用 “1234567812345678” 作为 id，目前不支持自定义设置。
		输出： 摘要片段：[]byte 类型，一对特定公私钥对通常只有一个特定的摘要片段，可以预先算好，后续使用时直接取用结果。该摘要片段在签名时与被签名的信息拼接后使用哈希算法算出完整摘要。

		错误信息。

公钥相关函数

接口类型	函数名	说明
非成员函数	NewPublicKeyFromPEM(pem string) (*PublicKey, error)	功能：从 PEM 格式的字符串中读取解析出公钥。 参数：pem：string 类型，PEM 格式的公钥字符串。 输出： • 公钥：可用于加密或验签。 • 错误信息。
成员函数	(*PublicKey) GetPEM() (string, error)	功能：生成存有公钥信息的 PEM 格式字符串。 输出： • PEM：string 类型，可用于公钥的传输和保存。 • 错误信息。
	(*PublicKey) Encrypt(alg string, plaintext []byte, engine *Engine) ([]byte, error)	功能：加密，生成密文。 参数： • alg：string 类型，算法名称，SM2 加密算法固定入参为“sm2encrypt-with-sm3”。 • plaintext：[]byte 类型，明文字符串。 • engine：SM2 算法固定入参为 nil。 输出： • 密文：[]byte 类型。 • 错误信息。
	(*PublicKey) Verify(alg string, dgst []byte, sig []byte, engine *Engine) error	功能：验证摘要 dgst、签名 sig 是否与调用的公钥对应。 参数： • alg：string 类型，算法名称。SM2 签名算法固定入参为“sm2sign”。 • dgst：[]byte 类型，信息摘要。 • sig：[]byte 类型，签名。 • engine：SM2 签名算法固定入参 nil。 输出： 错误信息：若错误信息为 nil，则验证通过，否则验签不通过。
	(*PublicKey) ComputeSM2IDDigest(id string) ([]byte, error)	功能：对公钥和入参 id 生成特定摘要片段，用于签名时与信息生成最终摘要。与私钥的同名成员函数相同。 参数：id：string 类型，默认入参“1234567812345678”，暂不支持自定义。 输出： • 摘要片段：[]byte 类型。 • 错误信息。

哈希算法相关函数

接口类型	函数名	说明
非成员函数	NewDigestContext(alg string) (*DigestContext, error)	功能：初始化所选哈希算法。

		参数： alg：string 类型，算法名称，国密算法 SM3 算法入参为 “SM3”，其他算法按 openssl 管理入参。
		输出：哈希算法实例。
成员函数	(*DigestContext) Reset() error	功能：充值哈希算法，清除接入算法的字符串。
	(*DigestContext) Update(data []byte) error	功能：把 data 拼接到输入的数据末尾。
		参数：data：要拼接的新数据。
	(*DigestContext) Final() ([]byte, error)	功能：用已经用 Update 拼接入得数据生成摘要。
		输出： • 摘要：[]byte 类型，使用 Update 传入的所有数据生成的摘要。 • 错误信息。

国密算法使用说明（Go）

最近更新时间：2024-01-15 14:42:01

操作场景

用户可以利用 TBaaS 提供的国密算法的能力，便捷的在智能合约中使用国密算法。在智能合约中使用国密算法一般分为以下两个步骤：

1. 利用 TBaaS 提供的 gmtool 生成国密算法公私钥，同时使用生成的私钥对用户数据签名。
2. 利用 TBaaS 提供的 Go 语言智能合约包 gmssl，在智能合约中直接 import gmssl，编写国密算法相关的业务操作，例如验证签名等。

操作步骤

Gmtool国密算法公私钥生成以及数据签名

1. 执行以下命令，使用 gmtool 生成国密算法公私钥对。

```
./gmtool genkey -pkout=pk.sm2 -skout=sk.sm2
```

其中，pk.sm2 是国密算法公钥文件，sk.sm2 是国密算法私钥文件。

2. 执行以下命令，使用 gmtool 生成的私钥对信息进行签名。

```
./gmtool sign -skin=sk.sm2 -message=message -signature=sig.sm2
```

其中，message 是信息文件包含要签名的信息，sig.sm2 是生成的文件包含签名信息，该签名信息是 base64 编码的。

智能合约国密算法示例

该示例中的 Init 函数直接返回成功，无其余操作。Invoke 函数会根据不同业务逻辑进行细分调用，最终调用 verify 业务逻辑接口，用于验证用户的签名是否正确。您可访问 智能合约代码 获得完整代码，以下将对代码中的重要函数进行分析。

Init 函数示例

Init 函数主要用于在智能合约实例化和升级的时候默认调用。在实现 Init 函数的过程中，可以使用 Go 语言版本的合约 API 来对参数和账本进行操作。以返回成功为例，示例代码如下：

```
// Init函数不包含具体业务，直接返回成功。
func (t *SimpleAsset) Init(stub shim.ChaincodeStubInterface) peer.Response {
    // Get the args from the transaction proposal
    return shim.Success(nil)
}
```

Invoke 函数示例

Invoke 函数可以对用户的不同的智能合约业务逻辑进行拆分。本示例以只实现了一种业务类型国密算法验签 verify 为例，介绍如何通过调用 API GetFunctionAndParameters 获取到用户的具体业务类型和参数，再分别调用不同的函数。

```
// Invoke把用户调用的function细分到几个子function, 这里只实现了verify
// Invoke is called per transaction on the chaincode.
func (t *SimpleAsset) Invoke(stub shim.ChaincodeStubInterface) peer.Response {
    // Extract the function and args from the transaction proposal
    fn, args := stub.GetFunctionAndParameters()

    var result string
    var err error

    switch fn {
    case "verify": {
        result, err = verify(stub, args)
    }
```

```
    default: {
        shim.Error("invalid function")
    }
    }
    if err != nil {
        return shim.Error(err.Error())
    }

    // Return the result as success payload
    return shim.Success([]byte(result))
}
```

业务逻辑 verify 函数示例

本示例以实现国密算法签名验证的业务逻辑 verify 函数为例。调用国密算法的接口 NewPublicKeyFromPEM 用于获取国密算法公钥，调用 ComputeSM2IDDiges，NewDigestContext，Reset，Update，Final 用于生成信息摘要，调用 Verify 用于验证签名是否正确。

```
// verify输入的参数是信息,签名base64以及公钥base64
// verify验证用户的签名是否正确
func verify(stub shim.ChaincodeStubInterface, args []string) (string, error) {
    if len(args) != 3 {
        return "", fmt.Errorf("Incorrect arguments. Expecting a message, a signature, and a public key")
    }

    pkPem, err := base64.StdEncoding.DecodeString(args[2])
    if err != nil {
        fmt.Println(err.Error())
        return "", err
    }
    // 获取国密算法公钥
    pk, err := gmssl.NewPublicKeyFromPEM(string(pkPem))
    if err != nil {
                fmt.Println(err.Error())
        return "", err
    }

    sig, err := base64.StdEncoding.DecodeString(args[1])
    if err != nil {
                fmt.Println(err.Error())
        return "", err
    }

    message := args[0]
    // 计算信息摘要
    zid, err := pk.ComputeSM2IDDigest(DEFAULT_USER_ID)
    if err != nil {
                fmt.Println(err.Error())
        return "", err
    }
    sm3ctx, err := gmssl.NewDigestContext("SM3")
    if err != nil {
                fmt.Println(err.Error())
        return "", err
    }
    sm3ctx.Reset()
    sm3ctx.Update(zid)
    sm3ctx.Update([]byte(message))
    tbs, err := sm3ctx.Final()
    if err != nil {
```

```
                fmt.Println(err.Error())
        return "", err
    }
    // 验证签名
    err = pk.Verify("sm2sign", tbs, sig, nil)
    if err == nil {
        fmt.Println("Valid Signature")
        return "Valid Signature.", nil
    } else {
        fmt.Println("Invalid Signature")
        return "Invalid signature:", nil
    }
}
```

同态加密支持说明（Go）

最近更新时间：2024-03-25 16:04:29

简介

在 Fabric 区块链网络中，用户可以把业务数据按照特定的业务逻辑上链。对于链上的数据，多个参与方之间均透明可见证。对于不能对链上其他组织直接开放的隐私数据，用户可以直接把相关数据的 hash 值存储在链上作为见证。但是存在如下情形，用户数据不能直接向其他组织直接开放，但其他组织又需要根据链上这些数据进行特定的运算背书。
针对以上场景，TBaaS 引入同态加密的能力，很好的保证了数据隐私性，链上透明性，以及数据可操作性。

TBaaS 的同态加密能力特点

TBaaS 的同态加密能力，主要体现在以下两个方面：

- TBaaS 提供了一个单独的用户工具 paitool， 用户可以使用这个工具做一些基本的同态公私钥生成，同态加密和同态解密操作。
- TBaaS 在 Go 语言智能合约中，额外引入了 paillier 包，增加了同态算法的 API 接口。用户可以通过 API 接口，实现同态公私钥生成、同态加密、同态解密、同态加法、同态减法以及部分同态乘法（密文和明文相乘）。

下面将分别对 paitool 和 Go 智能合约的 paillier 包进行说明。

同态加密算法工具 paitool

paitool 工具是同态加密 Paillier 算法的用户工具，主要实现了生成同态算法公私钥对，同态加密，同态解密3个功能，您可访问 paitool 进行下载。

使用说明

命令	描述	参数
genkey	生成 Paillier 算法的公私钥对	length：公钥长度，即安全等级，[]代表可选，默认为2048 pkout：新公钥的文件路径 skout：新私钥的文件路径
encrypt	加密	pkin：用于加密的公钥的文件路径，可以是 genkey 生成的公钥文件路径 plaintext：要加密的明文数字，十进制形式的数字。假设公钥值为 N，长度为 n bit，该数字范围只能取 (-N/2,N/2]，即明文的安全长度为 n - 2 bit（不算符号的长度） cipherout：生成的密文的指定文件路径
decrypt	解密	skin：解密使用的私钥的文件路径 cipherin：需要解密的密文文件路径

示例

- genkey
  执行以下命令，生成的新公钥存在 pk.pai 文件中，新私钥存在 sk.pai 文件中。
  公钥是一行16进制字符串，私钥是两行16进制字符串。示例中的公钥是 2048-bit 长的二进制串，16进制编码后为512位字符串。私钥是两段 1024-bit 长的二进制串，16进制编码后为两段256位字符串。

```
./paitool genkey [-length=2048] -pkout=pk.pai -skout=sk.pai
```

- encrypt
  执行以下命令，工具会使用 pk.pai 中存储的公钥对数字10进行加密，生成的密文存在文件 cipher.pai 中。
  密文是一个16进制字符串，可以直接作为 paitool 工具的解密功能入参，或者在调用 Paillier chaincode 的解密、同态运算接口时作为密文入参。

```
./paitool encrypt -pkin=pk.pai -plaintext=10 -cipherout=cipher.pai
```

- decrypt
  执行以下命令，工具会解析文件 sk.pai 中的私钥，对文件 cipher.pai 中的密文进行解密，解密后的明文结果会直接输出。

```
./paitool decrypt -skin=sk.pai -cipherin=cipher.pai
```

Go 语言智能合约 paillier 包接口说明

Go 语言智能合约 paillier 包是根据轻量同态加密 Paillier 算法实现的，该算法是由 Paillier Pascal 于1999年提出。用户在 TBaaS 中使用 Go 语言智能合约时，可直接 import paillier 包，使用相关的接口。该算法支持加法、减法、部分乘法，但对运算的输入和输出有范围限制，**即参与运算的数和运算结果的长度都不能比公钥长，否则会溢出。**

paillier 包支持接口

接口	功能	参数	输出
GenerateKey(rand io.Reader, length int) (*PrivateKey, error)	生成私钥	• rand：io.Reader，是随机数生成器，推荐使用 crypto/rand 库中的 rand.Reader • length：int，是公钥长度，也代表安全性，推荐使用大于2048的整数	• 私钥：用于解密，可导出成员 PublicKey 作为公钥进行加密 • 错误信息
Encrypt(pk *PublicKey, plaintext string) (string, error)	加密，生成密文	• pk：公钥 • plaintext：string 类型明文，格式需要是以 string 表示的整数，可以是负数	• 密文：string 类型 • 错误信息
Decrypt(sk *PrivateKey, ciphertext string) (string, error)	解密，用私钥 sk 从密文中获取明文	• sk：私钥 • ciphertext：string 类型密文	• 明文：string 类型表示的数字 • 错误信息
Neg(pk *PublicKey, ciphertext string) (string, error)	计算密文中被加密数字的相反数，生成其相反数的密文	• pk：公钥 • ciphertext：string 类型密文	• 密文：string 类型，其内容为原信息的相反数 • 错误信息
AddCipher(pk *PublicKey, cipher1 string, cipher2 string) (string, error)	计算 cipher1 和 cipher2 中明文的和，并生成和的新密文	• pk：公钥 • cipher1，cipher2：string 类型的密文	• 密文：string 类型 • 错误信息
Add(pk *PublicKey, cipher string, plain string) (string, error)	计算密文 cipher 中被加密的数与明文数字 plain 的和，并生成和的新密文	• pk：公钥 • cipher：string 类型密文 • plain：string 类型明文，以 tring 表示的一个数，可以为负数	• 密文：string 类型 • 错误信息
SubCipher(pk *PublicKey, cipher1 string, cipher2 string) (string, error)	计算 cipher1 和 cipher2 中明文的差（cipher1 − cipher2），并生成差的新密文	• pk：公钥 • cipher1，cipher2：string 类型的密文	• 密文：string 类型 • 错误信息
Sub(pk *PublicKey, cipher string, plain string) (string, error)	计算密文 cipher 中被加密的数与明文数字 plain 的差（cipher − plain），并生成差的新密文	• pk：公钥 • cipher：string 类型密文 • plain：string 类型明文，以 string 表示的一个数，可以为负数	• 密文：string 类型 • 错误信息
Mul(pk *PublicKey, cipher string, plain string) (string, error)	计算密文 cipher 中被加密的数与明文数字 plain 的乘积，并生成积的新密文	• pk：公钥 • cipher：string 类型密文 • plain：string 类型明文，以 string 表示的一个数，可以为负数	• 密文：string 类型 • 错误信息
GetPublicKeyHex(pk *PublicKey) string	生成公钥的16进制字符串表达方式	pk：公钥	string 类型的公钥字符串
GetPublicKeyFromHex(hex string) (*PublicKey, error)	从16进制字符串中恢复出公钥	string：16进制字符串	公钥
WritePublicKeyToFile(pk *PublicKey, file string)	将公钥写入文件	• pk：公钥	错误信息

error		• file：文件名	
ReadPublicKeyFromFile(file string) (*PublicKey, error)	从文件中读取公钥	file：文件名	• 公钥 • 错误信息
GetPrivateKeyHex(sk *PrivateKey) string	生成私钥的16进制字符串表达方式	sk：私钥	string 类型的私钥字符串
GetPrivateKeyFromHex(hex string) (*PrivateKey, error)	从16进制字符串中恢复出私钥	string：16进制字符串	私钥
WritePrivateKeyToFile(sk *PrivateKey, file string) error	将私钥写入文件	• sk：私钥 • file：文件名	错误信息
ReadPrivateKeyFromFile(file string) (*PrivateKey, error)	从文件中读取私钥	file：文件名	• 私钥 • 错误信息
GetCiphertextHex(cipher string) string	生成密文的16进制字符串表达方式	cipher：密文	string 类型的密文字符串
GetCiphertextFromHex(hex string) string	从16进制字符串中恢复出密文	string：16进制字符串	密文
WriteCiphertextToFile(cipher string, file string) error	将密文写入文件	• cipher：密文 • file：文件名	错误信息
ReadCiphertextFromFile(file string) (string, error)	从文件中读取密文	file：文件名	• 密文 • 错误信息

同态加密使用说明（Go）

最近更新时间：2024-01-15 14:42:01

操作场景

用户可以利用 TBaaS 提供的同态加密的能力，便捷的在智能合约中使用同态加密。使用同态加密可以分为以下三个步骤：

1. 使用 TBaaS 提供的 paitool 生成同态公私钥，同时使用生成的公钥对用户数据线下加密。
2. 使用 TBaaS 提供的 Go 语言智能合约包 paillier，编写同态相关的业务层操作，例如同态加、同态减、部分同态乘等。
3. 当业务逻辑完成后，用户可以从智能合约中获取到对应的同态加密数据，线下使用对应的同态私钥进行数据解密，从而可以获取链上业务操作完成后的真实数据。

操作步骤

Paitool 同态公私钥生成以及数据加密

1. 依次执行以下命令，使用 paitool 生成两组同态加密的公私钥对。

```
./paitool genkey -length=2048 -pkout=pk1.pai -skout=sk1.pai
./paitool genkey -length=2048 -pkout=pk2.pai -skout=sk2.pai
```

其中，pk1.pai, pk2.pai 是同态加密公钥文件，sk1.pai, sk2.pai 是同态加密私钥文件。

2. 依次执行以下命令，使用 paitool 对数字100和200分别用不同的公钥进行同态加密。

```
./paitool encrypt -pkin=pk1.pai -plaintext=100 -cipherout=cipher1.pai
./paitool encrypt -pkin=pk2.pai -plaintext=200 -cipherout=cipher2.pai
```

其中，cipher1.pai 是数字100同态加密后的密文数据，cipher2.pai 是数字200加密后的密文数据。

智能合约同态加密示例

Hyperledger Fabric 提供了很多官方的智能合约样例，具体请参考 fabric 官方示例 。本示例以 Hyperledger Fabric 官方提供的 example02 样例为例进行同态修改。该示例的 Init 函数用于初始化两个 key/value 键值对，其中 value 是用同态加密后的数据，Invoke 函数用于根据不同业务逻辑进行细分调用，最终调用以下业务逻辑接口：

- invoke：用于 key 之间的 value 转移。
- query：用于查询 key 所对应的值。

您可以访问 智能合约代码 获得完整代码，以下将对代码中的重要函数进行分析。

Init 函数示例

Init 函数主要用于在智能合约实例化和升级的时候默认调用。在实现 Init 函数的过程中，可以使用 Go 语言版本的合约 API 来对参数和账本进行操作。在这个示例中，通过调用 API GetFunctionAndParameters 获取到用户输入参数。在获取用户输入参数后，通过调用 API PutState 将数据写到账本中。

```
// Init函数用于初始化两个键值对，用户输入的参数为KEY1_NAME, VALUE1, KEY2_NAME, VALUE2,其中VALUE1和VALUE2都是同态加
密后的数据
// 在例子中，VALUE1是cipher1.pai的内容， VALUE2是ciphe2.pai的内容
func (t *SimpleChaincode) Init(stub shim.ChaincodeStubInterface) pb.Response {
    fmt.Println("ex02 Init")
    // 调用API GetFunctionAndParameters 获取用户输入参数
    _, args := stub.GetFunctionAndParameters()
    var A, B string    // Entities
    var Aval, Bval string // Asset holdings
    var err error

    if len(args) != 4 {
        return shim.Error("Incorrect number of arguments. Expecting 4")
    }
```

```
    // Initialize the chaincode
    A = args[0]
    Aval = args[1]

    B = args[2]
    Bval = args[3]

    // 调用API PutState把数据写入账本
    // Write the state to the ledger
    err = stub.PutState(A, []byte(Aval))
    if err != nil {
        return shim.Error(err.Error())
    }

    err = stub.PutState(B, []byte(Bval))
    if err != nil {
        return shim.Error(err.Error())
    }

    return shim.Success(nil)
}
```

Invoke 函数示例

Invoke 函数可以对用户的不同的智能合约业务逻辑进行拆分。本示例通过调用 API GetFunctionAndParameters 获取到用户的具体业务类型和参数，再分别调用不同的函数，如 invoke 和 query 函数。

```
// Invoke把用户调用的function细分到几个子function, 包含invoke和query
func (t *SimpleChaincode) Invoke(stub shim.ChaincodeStubInterface) pb.Response {
    fmt.Println("ex02 Invoke")
    // 调用API GetFunctionAndParameters获取用户输入的业务类型和参数
    function, args := stub.GetFunctionAndParameters()
    if function == "invoke" {
        // Make payment of X units from A to B
        return t.invoke(stub, args)
    } else if function == "query" {
        // the old "Query" is now implemtned in invoke
        return t.query(stub, args)
    }
    return shim.Error("Invalid invoke function name")
}
```

业务逻辑 invoke 函数示例

业务逻辑 invoke 函数主要用于实现业务逻辑中的资产转移。本示例中通过调用 API GetState 获取到 KEY 对应的同态加密资产总值，通过调用用户业务逻辑实现资产转移，通过调用 API PutState 将用户最终资产写入账本。
在此过程中，调用了同态加密的接口 GetPublicKeyFromHex 用于获取同态公钥，GetCiphertextFromHex 用于获取同态加密数据，Sub 用于同态密文和明文相减，Add 用于同态密文和明文相加以及 GetCiphertextHex 用于获取同态加密后的16进制密文数据。

```
// invoke实现两个键之间的value转移，输入为KEY1_NAME, KEY1_PUBKEYINHEX, KEY2_NAME，KEY2_PUBKEYINHEX，VALUE
// 在例子中，KEY1_PUBKEYINHEX是pk1.pai内容的base64, KEY2_PUBKEYINHEX是pk2.pai的内容base64
func (t *SimpleChaincode) invoke(stub shim.ChaincodeStubInterface, args []string) pb.Response {
    var A, B string    // Entities
    var Apkpem, Bpkpem []byte //public key in hex

    var Aval, Bval string // Asset encrypted holdings
    var X *big.Int          // Transaction value
    var err error
```

```
    if len(args) != 5 {
        return shim.Error("Incorrect number of arguments. Expecting 5")
    }

    A = args[0]
    Apkpem, err = base64.StdEncoding.DecodeString(args[1])
    if err != nil {
        return shim.Error("Failed to get Apkpem in hex")
    }

    B = args[2]
    Bpkpem, err = base64.StdEncoding.DecodeString(args[3])
    if err != nil{
        return shim.Error("Failed to get Bpkpem in hex")
    }
    X, isOK := new(big.Int).SetString(args[4], 10)
    if !isOK {
        return shim.Error("Fail to get X")
    }

    // Get the state from the ledger
    // API GetState获取对应账户的资产，这里的资产是同态加密后的数据
    Avalbytes, err := stub.GetState(A)
    if err != nil {
        return shim.Error("Failed to get state")
    }
    if Avalbytes == nil {
        return shim.Error("Entity not found")
    }
    Aval = string(Avalbytes)

    Bvalbytes, err := stub.GetState(B)
    if err != nil {
        return shim.Error("Failed to get state")
    }
    if Bvalbytes == nil {
        return shim.Error("Entity not found")
    }
    Bval = string(Bvalbytes)

    // 执行具体业务逻辑，这里是对应资产进行转移
    // Perform the execution
    // 调用同态接口GetPublicKeyFromHex获取公钥信息
    Apk, err := paillier.GetPublicKeyFromHex(string(Apkpem))
    if err != nil {
        return shim.Error("Fail to get A public key in PublicKey")
    }
    // 调用同态接口GetCiphertextFromHex获取同态密文信息
    Acipher, err := paillier.GetCiphertextFromHex(Aval)
    if err != nil {
        return shim.Error("Fail to get Ciphertext for A")
    }


    // 调用同态接口Sub执行密文和明文相减
    Aciphernew, err := paillier.Sub(Apk, Acipher, X.Text(10))
    if err != nil {
        return shim.Error("Fail to compute Aciphernew")
    }
    // 调用同态接口GetCiphertextHex获取同态密文16进制string
```

```
    Avalnew, err := paillier.GetCiphertextHex(Aciphernew)
    if err != nil {
        return shim.Error("Fail to get Avalnew")
    }

    Bpk, err := paillier.GetPublicKeyFromHex(string(Bpkpem))
    if err != nil {
        return shim.Error("Fail to get B public key in PublicKey")
    }

    Bcipher, err := paillier.GetCiphertextFromHex(Bval)
    if err != nil {
        return shim.Error("Fail to get Ciphertext for B")
    }
    // 调用同态接口Add执行密文和明文相加
    Bciphernew, err := paillier.Add(Bpk, Bcipher, X.Text(10))
    if err != nil {
        return shim.Error("Fail to compute Bciphernew")
    }

    Bvalnew, err := paillier.GetCiphertextHex(Bciphernew)
    if err != nil {
        return shim.Error("Fail to get Bvalnew")
    }

    // API PutState将对应资产写入账本
    // Write the state back to the ledger
    err = stub.PutState(A, []byte(Avalnew))
    if err != nil {
        return shim.Error(err.Error())
    }

    err = stub.PutState(B, []byte(Bvalnew))
    if err != nil {
        return shim.Error(err.Error())
    }

    return shim.Success(nil)
}
```

业务逻辑 query 函数示例

业务逻辑 query 函数主要用于实现业务逻辑中的账户查询功能，本示例通过调用 API GetState 查询对应账户的资产。

```
// query主要是查询键对应的值，输入为KEY_NAME
// query callback representing the query of a chaincode
func (t *SimpleChaincode) query(stub shim.ChaincodeStubInterface, args []string) pb.Response {
    var A string // Entities
    var err error

    if len(args) != 1 {
        return shim.Error("Incorrect number of arguments. Expecting name of the person to query")
    }

    A = args[0]

    // Get the state from the ledger
    Avalbytes, err := stub.GetState(A)
    if err != nil {
```

```
        jsonResp := "{\"Error\":\"Failed to get state for " + A + "\"}"
        return shim.Error(jsonResp)
    }

    if Avalbytes == nil {
        jsonResp := "{\"Error\":\"Nil amount for " + A + "\"}"
        return shim.Error(jsonResp)
    }

    return shim.Success(Avalbytes)
}
```

Paitool 同态数据解密

依次执行以下命令，通过智能合约从链上获取到最新的业务逻辑操作过的同态密文数据。拥有对应同态私钥的用户，可以解密出相应的明文。

```
./paitool decrypt -skin=sk1.pai -cipherin=cipher1new.pai
./paitool decrypt -skin=sk2.pai -cipherin=cipher2new.pai
```

其中，cipher1new.pai 对应的是智能合约中 KEY1_NAME 存储的值，cipher2new.pai 对应的是智能合约中 KEY2_NAME 存储的值。

零知识范围证明支持说明（Go）

最近更新时间：2024-01-15 14:42:01

简介

在 Fabric 区块链网络中，用户可以把业务数据按照特定的业务逻辑上链。链上存储 hash 数据具备了隐私性强的特点，但存在参与方难以对链上的 hash 数据直接操作的问题。存在如下场景，用户数据不能直接向其他组织直接开放，但其他组织又需要根据链上数据进行特定的运算背书，同时其他组织需要确定链上的数据在特定的数值范围内。例如，在用户转账场景中，对于加密的用户账户，参与方需要判断用户的余额大于等于转账金额，同时可以直接利用链上的加密数据进行加减操作。
针对以上问题，TBaaS 引入 Bulletproofs 零知识范围证明的能力，保证了数据隐私性、链上透明性和数据可操作性。Bulletproofs 零知识范围证明可以利用 Pedersen 承诺算法将明文数值隐藏，实现数据加密。同时 Pedersen 承诺是一个支持同态加法、减法、部分乘法运算的算法，因此在对已经制成承诺的数据进行运算时，不使用“解密-运算-加密”流程，而是直接对承诺进行计算。

TBaaS 的零知识范围证明能力特点

TBaaS 的零知识范围证明能力，主要体现在以下两个方面：

- TBaaS 提供了零知识范围证明使用的两个工具 Bulletproofs 和 Pedersen。Bulletproofs 工具提供生成范围证明、验证范围证明与承诺的有效性、范围证明的同态运算（加、减、乘）等功能。Pedersen 工具提供生成承诺、验证承诺、致盲因子的随机生成、同态加法、减法、乘法及求相反数运算。
- TBaaS 在 Go 语言智能合约中，额外引入了 Bulletproofs 包，增加了零知识范围证明 API 接口。用户可以通过在智能合约中使用 API 接口，实现验证范围证明与承诺的有效性、范围证明的同态运算（加、减、乘）等功能。

下面将分别对零知识范围证明工具 Bulletproofs 和 Pedersen 进行说明。

零知识范围证明工具 Bulletproofs

Bulletproofs 工具是零知识范围证明 Bulletproofs 算法的用户端工具。零知识范围证明算法在提供验证一个数值属于 [0，2^{64}) 的同时，不向验证者泄露此数值。该工具提供生成范围证明、验证范围证明与承诺的有效性、范围证明的同态运算（加、减、乘）等功能。您可访问 Bulletproofs 进行下载。

使用说明

命令	描述	输入参数	输出
prove	为输入的数字生成零知识证明，证明其为非负整数。	• value：无符号整数，证明该整数是非负数。 • opening：用于在 Pedersen 承诺和 Bulletproofs 证明中隐藏整数 value 的随机数，也用来验证某个 Pedersen 承诺中隐藏的整数是否是某个给定值时用到的致盲因子。（可选，若不给出则使用一个密码学随机数）	• proof：{proof}类型的字符串，是零知识范围证明的内容。 • commitment：{commitment}类型的字符串，是对应 proof 的承诺。 • opening：{opening}类型的字符串，用在 commitment 中作为致盲因子的部分，需要用户自行保存，在核对 commitment 中的数值时使用。
verify	验证已输入 Bulletproofs 证明和 Pedersen 承诺的有效性。	• proof：{proof}类型的数据，由 prove 或者 add、sub、mul 等同态运算命令生成的零知识范围证明。 • commitment：{commitment}类型数据，由 pedersen 工具、prove 命令或者 add、sub、mul 等同态运算命令生成的 Pedersen 承诺。	若 proof 与 commitment 相符，并且 commitment 中隐藏的值为非负整数，则显示“Valid proof”，否则显示“Invalid proof”。
add	证明的同态加法计算。支持“承诺 + 承诺”、“承诺 + 数值”两种模式。	• value1、value2：数字，分别在commitment1、commitment2两个承诺中。 • commitment1、commitment2：相加的两个承诺。 • opening1、opening2：分别为 commitment1和commitment2的致盲因子。 其中，commitment2、opening2是可选入参，如果同时给出了这两个入参，则使用“承诺 + 承诺”模式，否则使用“承诺 + 数值”模式。	• proof：value1 + value2为非负整数的证明。 • commitment：value1 + value2的承诺。 • opening：commitment 的致盲因子。
sub	证明的同态减法计算。支持“承诺 - 承	• value1、value2：数字，分别在commitment1、commitment2两个承诺中。	• proof：value1 - value2为非负整数的证明。

	诺”、“承诺 – 数值”两种模式。	• commitment1、commitment2：相减的两个承诺。 • opening1、opening2：分别为 commitment1和commitment2的致盲因子。 其中，commitment2、opening2是可选入参，如果同时给出了这两个入参，则使用“承诺 – 承诺”模式，否则使用“承诺 – 数值”模式。	• commitment：value1 – value2的承诺。 • opening：commitment 的致盲因子。
mul	证明的同态乘法计算。只有“承诺 * 数值”的模式。	• value1、value2：相乘的两个值。 • commitment1：包含 value1的承诺。 • opening1：commitment1的致盲因子。	• proof：value1 * value2为非负整数的证明。 • commitment：value1 * value2的承诺。 • opening：commitment 的致盲因子。

零知识范围证明工具 Pedersen

Pedersen 工具是 Pedersen 承诺的用户端工具。Pedersen 承诺是一个同态加密算法，可以隐藏明文，且与目标明文绑定。但 Pedersen 承诺不能进行解密，作为替代，它支持对明文、承诺（密文）关联性的验证。Pedersen 承诺中用于生成承诺和验证明文、承诺关联性的组件被称为致盲因子，又叫做 opening。该工具提供生成承诺、验证承诺及致盲因子的随机生成、同态加法、减法、乘法、求相反数运算的工具。您可访问 Pedersen 进行下载。

使用说明

命令	描述	输入参数	输出
commit	生成承诺。	• value：在承诺中绑定及隐藏的目标值。 • opening：用于隐藏 value 的随机数。若不给出则随机生成一个致盲因子。	• commitment：绑定了 value 的承诺。 • opening：用于生成 commitment 的致盲因子。
verify	验证 Pedersen 承诺的有效性。	• commitment：绑定了 Value 的 Pedersen 承诺。 • value：在承诺中绑定及隐藏的目标值。 • opening：用于隐藏 value 的随机数。若不给出则随机生成一个致盲因子。	Valid or Invalid。
genOpening	生成随机致盲因子。	–	opening：一个随机生成的44-byte 的 base64字符串（32-byte 二进制串）。该随机字符串可用于在生成承诺 ./pedersen commit 或生成证明 ./bulletproofs prove 命令中作为指定的随机数入参。
openingAdd	在 Pedersen 承诺的“承诺 + 承诺” 同态加法运算中更新致盲因子。	opening1、opening2：两个相加的致盲因子。	opening：opening1 + opening2
openingSub	在 Pedersen 承诺的“承诺 – 承诺” 同态减法运算中更新致盲因子。	opening1、opening2：两个相减的致盲因子。	opening：opening1 – opening2
openingMul	在 Pedersen 承诺的“承诺 * 数值” 同态乘法运算中更新致盲因子。	• value：参与乘法的数值。 • opening：参与乘法的致盲因子。	opening：opening * value

openingNeg	计算 opening 的相反数（mod p 数域内的相反数，其中 p 是一个大质数）。	opening：需要求相反数的目标致盲因子。	opening：– opening

Go 语言智能合约 Bulletproofs 包接口说明

用户在 TBaaS 中使用 Go 语言智能合约时，可直接 import bulletproofs 包，使用相关的接口。当前接口支持验证范围证明与承诺的有效性，同态加法、减法、乘法、求相反数运算。

Bulletproofs 包支持接口

接口	功能	参数	输出
BulletproofsVerify(proof []byte, commitment []byte) (bool, error)	验证范围证明与承诺的有效性。	• proof：672 – byte 二进制串，证明内容。 • commitment：32 – byte 二进制串，与 proof 对应的 Pedersen 承诺。	• 证明有效性：若证明与承诺通过验证，即 x 在[0, 264) 范围内，则返回 true，否则为 false。 • 错误信息
PedersenAddNum(commitment []byte, value uint64) ([]byte, error)	进行“承诺 + 数值”的同态运算，该运算不涉及致盲因子变化，链上、线下皆适用。	• commitment：32 – byte 二进制串，原承诺。 • value：无符号64位整数，参与同态运算的数值。	• 新承诺：32 – byte 二进制串，假设 commitment 所绑定值为 x，则新承诺绑定值为 “x + value”，致盲因子不变。 • 错误信息
PedersenAddCommitment(commitment1, commitment2 []byte) ([]byte, error)	进行“承诺 + 承诺”的同态运算，仅返回新承诺，虽然致盲因子发生变化，但运算过程及结果中不出现新旧致盲因子，适用于链上数据更新。	• commitment1：32 – byte 二进制串，绑定值为 x、致盲因子为 r 的承诺。 • commitment2：32 – byte 二进制串，绑定值为 y、致盲因子为 s 的承诺。	• 新承诺：32 – byte 二进制串，绑定值为 x + y、致盲因子为 r + s 的承诺。 • 错误信息
PedersenSubNum(commitment []byte, value uint64) ([]byte, error)	进行“承诺 – 数值”的同态运算，该运算不涉及致盲因子变化，链上、线下皆适用。	• commitment：32 – byte 二进制串，原承诺。 • value：无符号64位整数，参与同态运算的数值。	• 新承诺：32 – byte 二进制串，假设 commitment 所绑定值为 x，则新承诺绑定值为 x – value，致盲因子不变。 • 错误信息
PedersenSubCommitment(commitment1, commitment2 []byte) ([]byte, error)	进行“承诺 – 承诺”的同态运算，仅返回新承诺，虽然致盲因子发生了变化，但运算过程及结果中不出现新旧致盲因子，适用于链上数据更新。	• commitment1：32 – byte 二进制串，绑定值为 x、致盲因子为 r 的承诺。 • commitment2：32 – byte 二进制串，绑定值为 y、致盲因子为 s 的承诺。	• 新承诺：32 – byte 二进制串，绑定值为 x – y、致盲因子为 r – s 的承诺。 • 错误信息
PedersenMulNum(commitment1 []byte, value uint64) ([]byte, error)	进行“承诺 * 数值”的同态运算，仅返回新承诺，虽然致盲因子发生了变化，但运算过程及结果中不出现新旧致盲因子，适用于链上数据更新。	• commitment：32 – byte 二进制串，原承诺。 • value：无符号64位整数，参与同态运算的数值。	• 新承诺：32 –byte 二进制串，假设 commitment 所绑定值为 x、致盲因子为 r，则新承诺绑定值为 x * value，致盲因子为 r * value。 • 错误信息

零知识范围证明使用说明（Go）

最近更新时间：2024-11-12 09:13:42

操作场景

用户可以利用 TBaaS 提供的零知识范围证明的能力，便捷的在智能合约中使用零知识证明。在智能合约中使用零知识范围证明，可以分为以下四个步骤：

1. 使用 TBaaS 提供的零知识范围证明工具 Bulletproofs，对特定的数字生成零知识证明，证明其为非负整数。
2. 使用 TBaaS 提供的零知识范围证明工具 Bulletproofs，对步骤1的数字执行同态减法操作，并生成证明，确保对应的数字为非负整数。
3. 使用 TBaaS 智能合约将步骤1生成的承诺和证明存储到链上。
4. 使用 TBaaS 智能合约对特定的数字执行同态减法计算，并验证步骤2生成的证明和同态减承诺的有效性。

操作步骤

本文以使用零知识范围证明一个隐藏的值满足特定的条件为例，例如“大于等于18岁”。

1. 执行以下命令，使用 Bulletproofs 对特定的数字生成零知识证明，证明其为非负数。

```
bulletproofs prove -value=18
```

以返回成功为例，示例代码如下：

```
bulletproofs prove -value=18
proof:
jM0+O70Z0nBQJif0BBQHTkWqKskk06FkwwNqTv3B9RcqcElZXezAkhEaMdFwpZKLX09PS8ZOhNsOoGvpCizxTiKxyEmbsNJvuUZVuy
MzQq26voz1GRgn31oVk/oo8Voe8FavZiX1F9wwsBWzloAfaRrKvbgeNN2Tq7BCGVIvMi/1AddDWBv32aa4h18iYKYrE8OAixUh57lN
5mujdfUYCA6cZL0c8NccVJ5ila1E4dBgXAlxWPAaAC3LlJV4Tf0JaVjluYCw8PRGfO3u6AIjAVfJXkDfNOiOR9jjiUEVTgL6KipHY6
EHAZV+4hSKGoD0K2+fSMFkK0+4BC+bokedS1pD2ZiNSxBlUJ9psWa+FBOKHnesHp9N6WwypnIclKQ6Qu8O2Qtw7l/UE+lXswTO3/1S
eTTbMtQKxGLdYeEoM2rGZy9GQ3A/fq1H1H9410VlUGzjzVaez8wrtgMa/dwKJj7WHtZ8lhWdzcMsoBo3XuO7OZ6YN3XxdWH1AOjn4y
U7AA0IokI6lBIVRKZSVBVRyngxiQb7eulPH10xXW/OOzWwUCm6v/Iw5j3+OQR5y/AfT+2cm/Ar6AAqptnp8mLzId4NYm7QhsOAWvUu
UgQFFHnyVff4GBUFXdLOKJb9oJZOThmy4VeMWz5fL6Jwm0Mcn+uaGENZw4gwPtlg4dy3OHV6suIGykFUdK7eqWuP3+Sumf96NyGQRW
pa+1Qk1tU8aAAy5jlmcmlPToL9ST8DGqOBFNXyLr7GcdKS5aB+vQsGFtLOeh46A61jiYi7QzTbaDwj6tVRZdvNwkjUl36kWRloG2Kq
iQe4YR/PIw6sWrj0Aj0H9l2lgZGw5ph34jQdDEWSYibDh4Xr9h0q4yw0ZpBagy8pgVIxYVE3N39BCU4C
commitment: ********************************+bSfFX3GlhI=
opening: 04QToSmCYt3KaFqB6VMYDhh/aMKXh1G2b+xnS0IwYAY=
```

2. 执行以下命令，使用 Bulletproofs 对 步骤1 的数字执行同态减法操作，并生成证明，确保对应的数字为非负整数。

```
bulletproofs sub -value1=18 -commitment1=********************************+bSfFX3GlhI= -
opening1=04QToSmCYt3KaFqB6VMYDhh/aMKXh1G2b+xnS0IwYAY= -value2=18
```

以返回成功为例，示例代码如下：

```
bulletproofs sub -value1=18 -commitment1=********************************+bSfFX3GlhI= -
opening1=04QToSmCYt3KaFqB6VMYDhh/aMKXh1G2b+xnS0IwYAY= -value2=18
proof:
GG1kmQGw6ENSmhaG7LfApsixeJIxAfe5Bmns2x6Gwhq2uFWkiWyLYo5EjnRX14wvjHcpchjp9eE1uQh6X3r/Y3Iy53hw7VjIkO8GcJ
x+11t5iYxWfaquM0iVQcRy1OQ1BkELjeIT6alkXi0qnBnlq4Z1fOcw/ouFg7Se4vX+I2dT5CaxJq++ZBbhx7hjBRPsDZCsDmEoXJcC
VTT1uccKAU0yoV+Vm1/11hzbhlWEbQoFGWAqn8w8IoRHF1J4BTkEtdcYtWuM7I1ZDYaY8IpyJzzSAAMHk65/nQgARppB9A8CAZ8FBE
U6Oj/iKUQwaILjGG6xY8DngaFM3jg8A7PgOww9Hv7ao0hK26rIpnTF3/m6ezEcWBchdhR6JkDuBIIhqgbtShcZ0iQoUn1ZpOSQ6aC2
5mHqBnq3/eJfzp5q/nwgBZObeEppGLk+KvcPUlmYElyVjNWZBkfqnDx+MRSME1Y2enysm2hxvjkr+/hpW56y75l2JS9QxWcQHVFFVz
VHOnDdSraHARDNHq5bZiinlz8n98fmYVCDyXq+y36JK3BGrljuuBeRNSeZZqhKy/JpyH+vsaeDQLLW7FualiXbNgwsZMNtnO4w53vG
yfTps/w9uhh1jmUDi/obK3czP5c6lHDBkUv7i6uzvLz5+tYzvamv/zUOdSHoNn1gcPS7XnCkHEnQzW0rCLHv38MVYeboe10n52Tj2+
aTuyeGUEa5IgDiFQJhLx/w6v0DUWl5lNGC/Ma6WjH47TOf4jDEIzVLWB5VIfWNLW9PPFxlfhckJ23iPBZZEFNpjInUNQqnJCAYB9BQ
kyreyFHYlNHeNl6g6YJEQ1zmZBrGguewH8fbAKY1OJV6fuPfi6xG5aIqfik/o4TNNVa6K2sdNkKDBkAP
commitment: CIXe8Nptv/3VJrgu9xlRHSGC7HB5DGqIGis+HlUipWw=
opening: 04QToSmCYt3KaFqB6VMYDhh/aMKXh1G2b+xnS0IwYAY=
```

3. 使用 TBaaS 智能合约将 步骤1 生成的承诺和证明存储到链上，您可访问 Go 语言版本智能合约代码 进行下载。合约需存储用户在 步骤1 生成的承诺和证明。输入参数分别为 key，步骤1承诺，步骤1证明。
   以返回成功为例，示例代码如下：

```
func set(stub shim.ChaincodeStubInterface, args []string) (string, error) {
 if len(args) != 3 {
     return "", fmt.Errorf("incorrect arguments: expecting a user name, a committed value, and a
proof")
 }

 userName := args[0]
 commitment := args[1]
 proof := args[2]

 commitmentRaw, err := base64.StdEncoding.DecodeString(commitment)
 if err != nil {
     return "invalid input", fmt.Errorf("invalid input: fail to parse commitment")
 }
 proofRaw, err := base64.StdEncoding.DecodeString(proof)
 if err != nil {
     return "invalid input", fmt.Errorf("invalid input: fail to parse proof")
 }

 isValid, err := bulletproofs.BulletproofsVerify(proofRaw, commitmentRaw)
 if !isValid {
     return "invalid proof", fmt.Errorf("invalid proof")
 }

 err = stub.PutState(userName, []byte(commitment))
 if err != nil {
     return "", fmt.Errorf("failed to set asset: %s", args[0])
 }

 err = stub.PutState(userName + proofSuffix, []byte(proof))
 if err != nil {
     return "", fmt.Errorf("failed to set asset: %s", args[0] + proofSuffix)
 }
 return "success", nil
}
```

4. **使用 TBaaS 智能合约对特定的数字执行同态减法计算，减去特定的数字，例如18，并验证 步骤2 生成的证明和智能合约中同态减承诺的有效性，从而可以在链上数据加密的情况下，得到链上数据大于等于18。输入参数分别为 key，要比较的数字18，步骤2生成的 proof。**
   以返回成功为例，示例代码如下：

```
func subNumber(stub shim.ChaincodeStubInterface, args []string) (string, error) {
 if len(args) != 3 {
     return "", fmt.Errorf("incorrect arguments: expecting a user name, a number, and a proof")
 }

 proof, err := base64.StdEncoding.DecodeString(args[2])
 if err != nil {
     return "", fmt.Errorf("fail to parse the input proof")
 }

 commitment1, err := stub.GetState(args[0])
 if err != nil {
     return "", fmt.Errorf("failed to get asset: %s with error: %s", args[0], err)
 }
 if commitment1 == nil {
     return "", fmt.Errorf("asset not found: %s", args[0])
 }
```

```
    proof1, err := stub.GetState(args[0] + proofSuffix)
    if err != nil {
        return "", fmt.Errorf("failed to get asset: %s with error: %s", args[0] + proofSuffix, err)
    }
    if proof1 == nil {
        return "", fmt.Errorf("asset not found: %s", args[0] + proofSuffix)
    }

    num, err := strconv.ParseInt(args[1], 10, 64)
    if err != nil {
        return "", fmt.Errorf("fail to recognize the number")
    }

    commitment1Raw, err := base64.StdEncoding.DecodeString(string(commitment1))
    if err != nil {
        return "", fmt.Errorf("fail to parse commitment1: " + err.Error())
    }
    proof1Raw, err := base64.StdEncoding.DecodeString(string(proof1))
    if err != nil {
        return "", fmt.Errorf("fail to parse proof1: " + err.Error())
    }

    ret, err := bulletproofs.BulletproofsVerify(proof1Raw, commitment1Raw)
    if err != nil {
        return "", fmt.Errorf("invalid proof1: " + err.Error())
    }
    if !ret {
        return "", fmt.Errorf("invalid proof1")
    }

    commitment, err := bulletproofs.PedersenSubNum(commitment1Raw, uint64(num))
    if err != nil {
        return "", fmt.Errorf("fail to compute sum of commitments: " + err.Error())
    }

    ret, err = bulletproofs.BulletproofsVerify(proof, commitment)
    if err != nil {
        return "", fmt.Errorf("invalid proof new: " + err.Error())
    }
    if !ret {
        return "", fmt.Errorf("invalid proof new")
    }

    return "success", nil
}
```

合约 API 列表（Java）

最近更新时间：2024-01-15 14:42:01

Fabric Java 语言版本智能合约有丰富的 API 接口，具体的代码实现可以参考 API 接口代码实现 。
从逻辑上划分，可将 API 接口分为以下类型：

- 交易信息提取
- 账本交互
- 参数处理
- 其他辅助类

交易信息提取

接口	说明
byte[] getBinding()	返回交易的 binding 信息
String getChannelId()	获取当前的通道名称
byte[] getCreator()	获取交易提交者信息
SignedProposal getSignedProposal()	获取交易提案相关数据
Map<String, byte[]> getTransient()	获取交易的临时信息
String getTxId()	获取交易的 ID 信息
Instant getTxTimestamp()	获取交易时间戳

账本交互

接口	说明
void putState(String key, byte[] value)	在账本中添加或者更新一对键值
void putStringState(String key, String value)	在账本中添加或者更新一对键值
byte[] getState(String key)	获取指定键对应的值
String getStringState(String key)	获取指定键对应的值
void delState(String key)	在账本中删除对应的键值
QueryResultsIterator<KeyValue> getStateByRange(String startKey, String endKey)	查询指定范围内的键值
QueryResultsIteratorWithMetadata<KeyValue> getStateByRangeWithPagination(String startKey, String endKey, int pageSize, String bookmark)	分页查询指定范围内的键值
QueryResultsIterator<KeyValue> getStateByPartialCompositeKey(String compositeKey)	查询匹配局部复合键的所有键值
QueryResultsIterator<KeyValue> getStateByPartialCompositeKey(String objectType, String... attributes)	查询匹配局部复合键的所有键值
QueryResultsIterator<KeyValue> getStateByPartialCompositeKey(CompositeKey compositeKey)	查询匹配局部复合键的所有键值
QueryResultsIteratorWithMetadata<KeyValue> getStateByPartialCompositeKeyWithPagination(CompositeKey compositeKey, int pageSize, String bookmark)	分页查询匹配局部复合键的所有键值

QueryResultsIterator<KeyValue> getQueryResult(String query)	查询状态数据库，需要支持富查询功能的状态数据库
QueryResultsIteratorWithMetadata<KeyValue> getQueryResultWithPagination(String query, int pageSize, String bookmark)	分页查询状态数据库，需要支持富查询功能的状态数据库
QueryResultsIterator<KeyModification> getHistoryForKey(String key)	返回对应键的所有历史值
void setStateValidationParameter(String key, byte[] value)	设置特定键的背书策略
byte[] getStateValidationParameter(String key)	获取特定键的背书策略
byte[] getPrivateData(String collection, String key)	获取指定私有数据集集中的键的值
String getPrivateDataUTF8(String collection, String key)	获取指定私有数据集集中的键的值
byte[] getPrivateDataHash(String collection, String key)	获取指定私有数据集集中的键的值的 hash
void putPrivateData(String collection, String key, byte[] value)	设置指定私有数据集集中键的值
void putPrivateData(String collection, String key, String value)	设置指定私有数据集集中键的值
void delPrivateData(String collection, String key)	删除指定私有数据集集中对应的键
void setPrivateDataValidationParameter(String collection, String key, byte[] value)	设置指定私有数据集集中键的背书策略
byte[] getPrivateDataValidationParameter(String collection, String key)	获取指定私有数据集集中键的背书策略
QueryResultsIterator<KeyValue> getPrivateDataByRange(String collection, String startKey, String endKey)	获取指定私有数据集集中特定范围键的键值
QueryResultsIterator<KeyValue> getPrivateDataByPartialCompositeKey(String collection, String compositeKey)	获取指定私有数据集集中匹配局部复合键的键值
QueryResultsIterator getPrivateDataByPartialCompositeKey(String collection, CompositeKey compositeKey)	获取指定私有数据集集中匹配局部复合键的键值
QueryResultsIterator<KeyValue> getPrivateDataByPartialCompositeKey(String collection, String objectType, String... attributes)	获取指定私有数据集集中匹配局部复合键的键值
QueryResultsIterator<KeyValue> getPrivateDataQueryResult(String collection, String query)	获取指定私有数据集集中特定查询的键值，需要支持富查询功能的状态数据库

参数处理

接口	说明
List<byte[]> getArgs()	获取智能合约中调用参数
List<String> getStringArgs()	获取智能合约中调用参数
String getFunction()	获取智能合约调用的函数名，默认第一个参数为函数名
List<String> getParameters()	获取智能合约调用的参数

其他辅助类

接口	说明
CompositeKey createCompositeKey(String objectType, String... attributes)	组合属性，形成复合键
CompositeKey splitCompositeKey(String compositeKey)	拆分复合键成一系列属性

Response invokeChaincode(final String chaincodeName, final List<byte[]> args, final String channel)	调用其它智能合约 invoke 方法
Response invokeChaincode(String chaincodeName, List<byte[]> args)	调用其它智能合约 invoke 方法
Response invokeChaincodeWithStringArgs(String chaincodeName, List<String> args, String channel)	调用其它智能合约 invoke 方法
Response invokeChaincodeWithStringArgs(String chaincodeName, List<String> args)	调用其它智能合约 invoke 方法
Response invokeChaincodeWithStringArgs(final String chaincodeName, final String... args)	调用其它智能合约 invoke 方法
void setEvent(String name, byte[] payload)	设置发送的事件
ChaincodeEvent getEvent()	获取发送的事件

合约示例（Java）

最近更新时间：2024-01-15 14:42:01

智能合约构成

Java 语言的智能合约代码，关键是在于实现以下接口：

```
/**
 * Defines methods that all chaincodes must implement.
 */
public interface Chaincode {
    /**
     *Called during an instantiate transaction after the container has been
     *established, allowing the chaincode to initialize its internal data
     */
    public Response init(ChaincodeStub stub);

    /**
     *Called for every Invoke transaction. The chaincode may change its state
     *variables.
     */
    public Response invoke(ChaincodeStub stub);
}
```

- 接口 init 主要是用于智能合约初始化和升级的时候调用这个接口，初始化相关的数据。
- 接口 invoke 主要是用于实现智能合约里的内部业务逻辑，用户可以根据需要，实现相关的业务。
- 在实现过程中，用户可以调用 ChaincodeStub 的 API 接口来和链上交互。

智能合约示例

基本示例

本示例以一个基本的智能合约用例为例，只包含智能合约的必须部分，没有实现任何业务逻辑。

```
/*
 * SimpleAssetDemo implements a simple chaincode to manage an asset
 */
public class SimpleAssetDemo extends ChaincodeBase {
    /*
     * Init is called during chaincode instantiation to initialize any data.
     */
    @Override
    public Response init(ChaincodeStub stub) {
    }
    /*
     * Invoke is called per transaction on the chaincode. Each transaction is
     * either a 'get' or a 'set' on the asset created by Init function. The 'set'
     * method may create a new asset by specifying a new key-value pair.
     */
    @Override
    public Response invoke(ChaincodeStub stub) {
    }

    public static void main(String[] args) {
        new SimpleAssetDemo().start(args);
    }
}
```

官方示例

Hyperledger Fabric 提供了很多官方的智能合约样例，具体请参考 fabric 官方示例 。本示例以 Hyperledger Fabric 官方提供的 chaincode_example02 样例为例。该示例的 init 函数用于初始化两个 key/value 键值对，invoke 函数用于根据不同业务逻辑进行细分调用，最终调用以下业务逻辑接口：

- invoke：用于 key 之间的 value 转移。
- delete：删除一个键值对。
- query：查询 key 所对应的值。

init 函数示例

init 函数在智能合约实例化以及升级的时候会被调用。在实现 init 函数的过程中，可使用 Java 语言版本的合约 API 列表 来对参数和账本进行操作。本例通过调用 API getFunction 和 getParameters 获取到用户输入参数。在获取用户输入参数后，通过调用 API putStringState 将数据写到账本中。具体代码如下：

```
    /*
     * init函数用于初始化两个键值对，用户输入的参数为KEY1_NAME, VALUE1,
     * KEY2_NAME, VALUE2
     */
    @Override
    public Response init(ChaincodeStub stub) {
        try {
            _logger.info("Init java simple chaincode");
            // 调用API getFunction获取当前的输入函数
            String func = stub.getFunction();
            if (!func.equals("init")) {
                return newErrorResponse("function other than init is not supported");
            }
            // 调用API getParameters 获取用户输入参数
            List<String> args = stub.getParameters();
            if (args.size() != 4) {
                newErrorResponse("Incorrect number of arguments. Expecting 4");
            }
            // Initialize the chaincode
            String account1Key = args.get(0);
            int account1Value = Integer.parseInt(args.get(1));
            String account2Key = args.get(2);
            int account2Value = Integer.parseInt(args.get(3));

            _logger.info(String.format("account %s, value = %s; account %s, value %s", account1Key,
account1Value, account2Key, account2Value));
            // 调用API putStringState 把数据写入账本
            stub.putStringState(account1Key, args.get(1));
            stub.putStringState(account2Key, args.get(3));

            return newSuccessResponse();
        } catch (Throwable e) {
            return newErrorResponse(e);
        }
    }
```

invoke 函数示例

invoke 函数对用户的不同的智能合约业务逻辑进行拆分。本例通过调用 API getFunction 和 getParameters 获取到用户的具体业务类型和参数，根据用户的不同业务类型，分别调用不同的业务函数，如 invoke，delete 和 query 函数。具体代码如下：

```
    // invoke把用户调用的function细分到几个子function, 包含invoke, delete和query
    @Override
    public Response invoke(ChaincodeStub stub) {
```

```
        try {
            _logger.info("Invoke java simple chaincode");
            // 调用API getFunction和getParameters获取用户输入的业务类型和参数
            String func = stub.getFunction();
            List<String> params = stub.getParameters();
            if (func.equals("invoke")) {
                return invoke(stub, params);
            }
            if (func.equals("delete")) {
                return delete(stub, params);
            }
            if (func.equals("query")) {
                return query(stub, params);
            }
            return newErrorResponse("Invalid invoke function name. Expecting one of: [\"invoke\",
\"delete\", \"query\"]");
        } catch (Throwable e) {
            return newErrorResponse(e);
        }
    }
```

业务逻辑 invoke 函数示例

业务逻辑 invoke 函数主要用于实现业务逻辑中的资产转移。本例中通过调用 API getStringState 获取到 KEY 对应的资产总值，通过调用用户业务逻辑实现资产转移，通过调用 API putStringState 将用户最终资产写入账本。具体代码如下：

```
    // invoke实现两个键之间的value转移，输入为KEY1_NAME, KEY2_NAME, VALUE
    private Response invoke(ChaincodeStub stub, List<String> args) {
        if (args.size() != 3) {
            return newErrorResponse("Incorrect number of arguments. Expecting 3");
        }
        String accountFromKey = args.get(0);
        String accountToKey = args.get(1);

        // API getStringState获取对应账户的资产
        String accountFromValueStr = stub.getStringState(accountFromKey);
        if (accountFromValueStr == null) {
            return newErrorResponse(String.format("Entity %s not found", accountFromKey));
        }
        int accountFromValue = Integer.parseInt(accountFromValueStr);

        String accountToValueStr = stub.getStringState(accountToKey);
        if (accountToValueStr == null) {
            return newErrorResponse(String.format("Entity %s not found", accountToKey));
        }
        int accountToValue = Integer.parseInt(accountToValueStr);

        int amount = Integer.parseInt(args.get(2));
        // 执行具体业务逻辑，这里对应资产进行转移
        if (amount > accountFromValue) {
            return newErrorResponse(String.format("not enough money in account %s", accountFromKey));
        }

        accountFromValue -= amount;
        accountToValue += amount;

        _logger.info(String.format("new value of A: %s", accountFromValue));
        _logger.info(String.format("new value of B: %s", accountToValue));
```

```
        // API putStringState将对应资产写入账本
        stub.putStringState(accountFromKey, Integer.toString(accountFromValue));
        stub.putStringState(accountToKey, Integer.toString(accountToValue));

        _logger.info("Transfer complete");

        return newSuccessResponse("invoke finished successfully", ByteString.copyFrom(accountFromKey +
": " + accountFromValue + " " + accountToKey + ": " + accountToValue, UTF_8).toByteArray());
    }
```

delete 函数示例

业务逻辑 delete 函数主要用于实现业务逻辑中的账户删除功能，本示例通过调用 API delState 删除对应账户。具体代码如下：

```
    // delete用于从账本中删除指定的键，输入为KEY_NAME
    // Deletes an entity from state
    private Response delete(ChaincodeStub stub, List<String> args) {
        if (args.size() != 1) {
            return newErrorResponse("Incorrect number of arguments. Expecting 1");
        }
        String key = args.get(0);
        // API delState删除特定的账户
        // Delete the key from the state in ledger
        stub.delState(key);
        return newSuccessResponse();
}
```

query 函数示例

业务逻辑 query 函数主要用于实现业务逻辑中账户查询功能，本示例通过调用 API getStringState 查询对应账户的资产。具体代码如下：

```
    // query主要是查询键对应的值，输入为KEY_NAME
    // query callback representing the query of a chaincode
    private Response query(ChaincodeStub stub, List<String> args) {
        if (args.size() != 1) {
            return newErrorResponse("Incorrect number of arguments. Expecting name of the person to
query");
        }
        String key = args.get(0);
        // API getStringState查询特定的账户
        String val  = stub.getStringState(key);
        if (val == null) {
            return newErrorResponse(String.format("Error: state for %s is null", key));
        }
        _logger.info(String.format("Query Response:\nName: %s, Amount: %s\n", key, val));
        return newSuccessResponse(val, ByteString.copyFrom(val, UTF_8).toByteArray());
    }
```

证书申请
v2.3
证书申请 CSR 生成指南

最近更新时间：2023-11-20 17:25:56

操作场景

本文介绍对应Fabric区块链网络 **非国密 ECC 证书** 和 **国密 SM2证书** 申请证书请求文件 CSR 生成的步骤，请结合您的实际情况通过以下两种方式生成 CSR：

- 非国密 ECC 证书申请 CSR
- 国密 SM2 证书申请 CSR

操作步骤

非国密 ECC 证书申请 CSR

1. 前往 OpenSSL 官网，下载 openssl 并配置安装。
2. 下载 cmecccsr 工具 并解压。
3. 执行以下命令，生成对应文件。

```
sh ecccsr.sh
```

该命令会生成以下四个文件：

- `user_ecc_sign.key`：为用户证书对应私钥，需安全保存，支持在 SDK 中使用。
- `user_ecc_sign.csr`：用于在 TBaaS 控制台 申请用户证书。
- `user_ecc_tls.key`：为用户 tls 证书对应私钥，需安全保存，支持在 SDK 中使用。
- `user_ecc_tls.csr`：用于在 TBaaS 控制台 申请用户 tls 证书。

工具说明

以下为工具中主要使用的命令：

1. 生成用户证书对应私钥和 CSR 文件
   - **生成密钥对**：生成的 `temp` 文件为用户证书对应私钥。

     ```
     openssl ecparam -name prime256v1 -genkey -out temp
     ```

   - **生成用户证书 CSR 文件**：命令中使用的 `openssl_user.cnf` 文件已包含在下载工具中。

     ```
     openssl req -batch -config openssl_user.cnf -key temp -new -sha256 -out user_ecc_sign.csr
     ```

   - **转换私钥格式**：将已生成的 `temp` 私钥转换为 `pkcs#8` 格式的 `user_ecc_sign.key` 文件，后续用于 fabric-sdk 的配置和识别。

     ```
     openssl pkcs8 -topk8 -in temp -nocrypt -out user_ecc_sign.key
     ```

2. 生成用户 tls 证书对应私钥和 CSR 文件
   - **生成密钥对**：生成的 `temp` 文件为用户 tls 证书对应私钥。

     ```
     openssl ecparam -name prime256v1 -genkey -out temp
     ```

   - **生成用户 tls 证书 CSR 文件**：命令中使用的 `openssl_user.cnf` 文件已包含在下载工具中。

```
openssl req -batch -config openssl_user.cnf -key temp -new -sha256 -out user_ecc_tls.csr
```

- **转换私钥格式：**将已生成的 `temp` 私钥转换为 `pkcs#8` 格式的 `ser_ecc_tls.key` 文件，后续用于 fabric-sdk 的配置和识别。

```
openssl pkcs8 -topk8 -in temp -nocrypt -out user_ecc_tls.key
```

国密 SM2 证书申请 CSR

1. 前往 GmSSL 官网，下载 gmssl 并配置安装。
2. 下载 cmsm2csr 工具 并解压。
3. 执行以下命令，生成对应文件。

```
sh sm2csr.sh
```

该命令会生成以下四个文件：

- `user_sm2_sign.key`：为用户证书对应私钥，需安全保存，支持在 SDK 中使用。
- `user_sm2_sign.csr`：用于在 TBaaS 控制台 申请用户证书。
- `user_sm2_tls.key`：为用户 tls 证书对应私钥，需安全保存，支持在 SDK 中使用。
- `user_sm2_tls.csr`：用于在 TBaaS 控制台 申请用户 tls 证书。

工具说明

以下为工具中主要使用的命令：

1. 生成用户证书对应私钥和 CSR 文件
   - **生成密钥对：**生成的 `temp` 文件为用户证书对应私钥。

```
gmssl ecparam -name sm2p256v1 -genkey -out temp
```

   - **生成用户证书 CSR 文件：**命令中使用的 `gmssl_user.cnf` 文件已包含在下载工具中。

```
gmssl req -batch -config gmssl_user.cnf -key temp -new -sm3 -out user_sm2_sign.csr
```

   - **转换私钥格式：**将已生成的 `temp` 私钥转换为 `pkcs#8` 格式的 `user_sm2_sign.key` 文件，后续用于 fabric-sdk 的配置和识别。

```
gmssl pkcs8 -topk8 -in temp -nocrypt -out user_sm2_sign.key
```

2. 生成用户 tls 证书对应私钥和 CSR 文件
   - **生成密钥对：**生成的 `temp` 文件为用户 tls 证书对应私钥。

```
gmssl ecparam -name sm2p256v1 -genkey -out temp
```

   - **生成用户 tls 证书 CSR 文件：**命令中使用的 `gmssl_user.cnf` 文件已包含在下载工具中。

```
gmssl req -batch -config gmssl_user.cnf -key temp -new -sm3 -out user_sm2_tls.csr
```

   - **转换私钥格式：**将已生成的 `temp` 私钥转换为 `pkcs#8` 格式的 `user_sm2_tls.key` 文件，后续用于 fabric-sdk 的配置和识别。

```
gmssl pkcs8 -topk8 -in temp -nocrypt -out user_sm2_tls.key
```

v1.4
证书申请 CSR 生成指南

最近更新时间：2023-11-20 17:25:56

操作场景

本文介绍对应区块链网络证书 ECC 和 SM2 申请 CSR 生成的步骤，请结合您的实际情况通过以下两种方式生成 CSR：

- ECC 证书申请 CSR
- SM2 证书申请 CSR

操作步骤

ECC 证书申请 CSR

1. 前往 OpenSSL 官网，下载 openssl 并配置安装。
2. 下载 ecccsr 工具 并解压。
3. 执行以下命令，生成对应文件。

```
sh ecccsr.sh
```

该命令会生成以下三个文件：

- `out.key`：为用户的私钥，需安全保存。
- `out.csr`：用于在 TBaaS 控制台 申请证书。
- `out_sk`：为 `out.key` 的 pkcs#8 格式，支持在 SDK 中使用。

工具说明

以下为工具中主要使用的命令：

- **生成密钥对**：生成的 `out.key` 文件为用户的私钥，需安全保存。

```
openssl ecparam -name prime256v1 -genkey -out out.key
```

- **生成 CSR 文件**：命令中使用的 `openssl_user.cnf` 文件已包含在下载工具中，无需变更内容。

```
openssl req -batch -config openssl_user.cnf -key out.key -new -sha256 -out out.csr
```

- **转换私钥格式**：将已生成的 `out.key` 私钥转换为 `pkcs#8` 格式的 `out_sk` 文件，用于 fabric-sdk 识别。

```
openssl pkcs8 -topk8 -in out.key -nocrypt -out out_sk
```

SM2 证书申请 CSR

1. 前往 GmSSL 官网，下载 gmssl 并配置安装。
2. 下载 sm2csr 工具 并解压。
3. 执行以下命令，生成对应文件。

```
sh sm2csr.sh
```

该命令会生成以下三个文件：

- `out.key`：为用户的私钥，需安全保存。
- `out.csr`：用于在 TBaaS 控制台 申请证书。
- `out_sk`：为 `out.key` 的 pkcs#8 格式，支持在 SDK 中使用。

工具说明

以下为工具中主要使用的命令：

- **生成密钥对**：生成的 `out.key` 文件为用户的私钥，需安全保存。

```
gmssl ecparam -name sm2p256v1 -genkey -out out.key
```

- **生成 CSR 文件**：命令中使用的 `gmssl_user.cnf` 文件已包含在下载工具中，无需变更内容。

```
gmssl req -batch -config gmssl_user.cnf -key out.key -new -sm3 -out out.csr
```

- **转换私钥格式**：将已生成的 `out.key` 私钥转换为 `pkcs#8` 格式的 `out_sk` 文件，用于 fabric-sdk 识别。

```
gmssl pkcs8 -topk8 -in out.key -nocrypt -out out_sk
```

Fabric 原生 SDK 示例

您可参考以下代码，在 Fabric 原生 SDK 中使用生成的私钥及已下载的证书：

Java SDK

```
Wallet wallet = Wallet.createFileSystemWallet("本地存储证书信息目录");
FileReader keyReader = new FileReader("pkcs#8 格式私钥证书文件");
FileReader certReader = new FileReader("TBaaS 上下载的证书");
wallet.put("证书标识", Identity.createIdentity("组织 MSP", certReader, keyReader));
```

Node.js SDK

```
var fs = require('fs-extra');
var fabric_client = new Fabric_Client();

var cert = fs.readFileSync('TBaaS 上下载的证书');
var priv = fs.readFileSync('pkcs#8 格式私钥证书文件');

fabric_client.createUser({
    username: '证书标识',
    mspid: '组织 MSP',
    cryptoContent: {
    privateKeyPEM: priv,
    signedCertPEM: cert
        }
    });
```

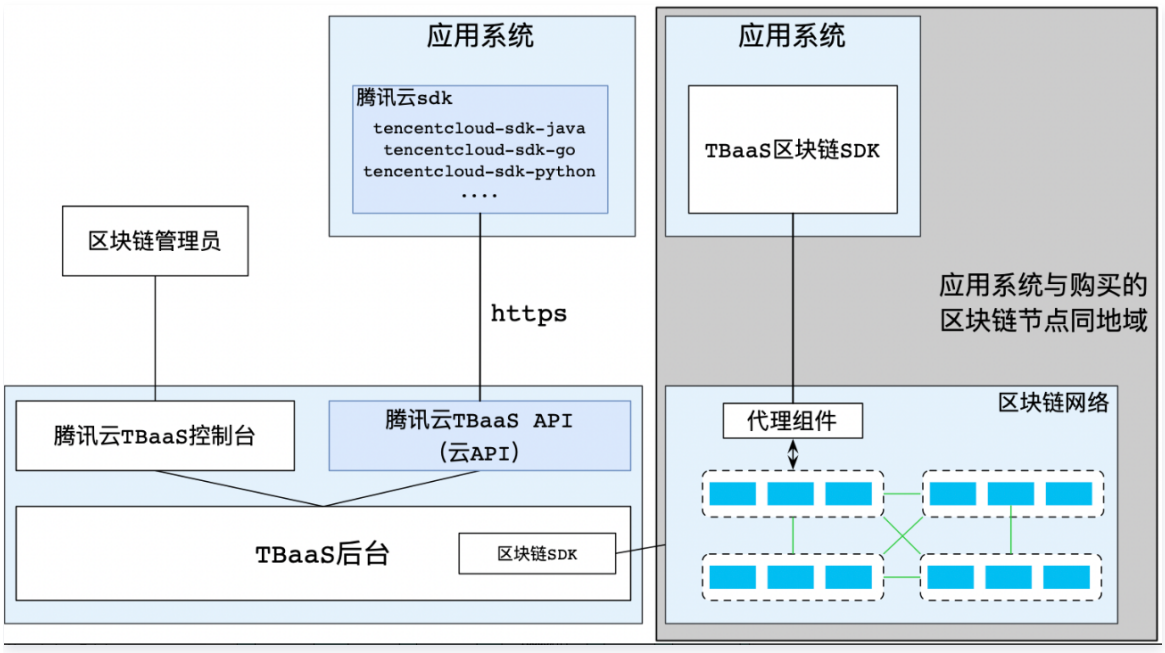
应用系统对接
对接说明

最近更新时间：2023-06-07 10:37:54

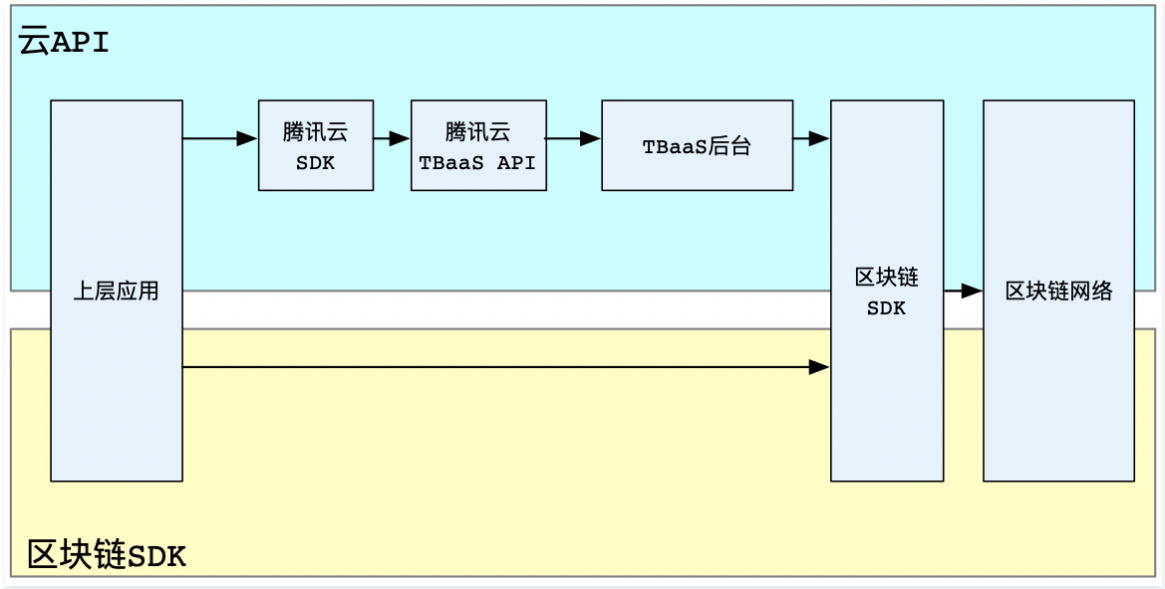
本文档介绍在购买腾讯云区块链服务平台（Tencent Blockchain as a Service，简称 TBaaS）产品后如何进行应用开发对接，即如何在应用系统上调用合约与区块链网络进行交互。

对接方式

合约是应用系统与区块链网络进行交互的唯一途径，因此不同的对接方式对应着不同的合约调用方式。如下图所示：



应用系统包括上层应用和腾讯云 SDK/区块链 SDK。目前用户的应用系统可以通过云 API 和区块链官方 SDK 调用 TBaaS 上区块链网络的合约。两种方式均通过区块链 SDK 与区块链网络进行交互，可认为云 API 是对区块链 SDK 的封装。两种调用方式的逻辑关系如下图所示：



对接方式对比

对接方式	优缺点	适用场景

云 API	配置简单，使用 SecretID 和 SecretKey 即可接入网络。功能较少，仅包含调用、查询和发起交易。	用户侧应用系统希望以最快的速度或者最小的开发时间对接云上的区块链网络。 用户侧应用系统对区块链的调用频率低于50笔每秒（TPS）。 用户侧应用系统无法固定部署在与区块链所在地域相同的云上私有网络（VPC）中或用户不打算购买腾讯云服务器（CVM）。 同时支持长安链 · ChainMaker、Fabric 区块链引擎。
区块链 SDK	配置繁琐，功能强大。 tbaas-fabric-sdk-java 在原生社区 SDK 上进行了封装，简化配置流程的同时，保留了丰富的功能支持。 chainmaker-sdk-go-demo 详细地介绍如何使用 SDK对接长安链网络，并提供完整的 Demo，帮助用户快速了解 SDK 对接的方式。	用户侧应用系统对区块链的调用频率超过50笔每秒（TPS）。 用户侧应用系统已经基于区块链原生的 SDK 进行了相关开发，迁移至云上时，希望改动尽可能少。 若需内网服务（需购买腾讯云服务器），用户侧服务器所在 VPC 与区块链网络在同一地域。例如，用户侧服务器 VPC 在广州，区块链网络也需要购买在广州，目前 TBaaS 仅支持同地域打通。 目前长安链 · ChainMaker 仅支持外网服务，Fabric 支持内网、外网服务。

选择对接方式

您可通过 云 API 或 区块链 SDK 两种方式对接网络。

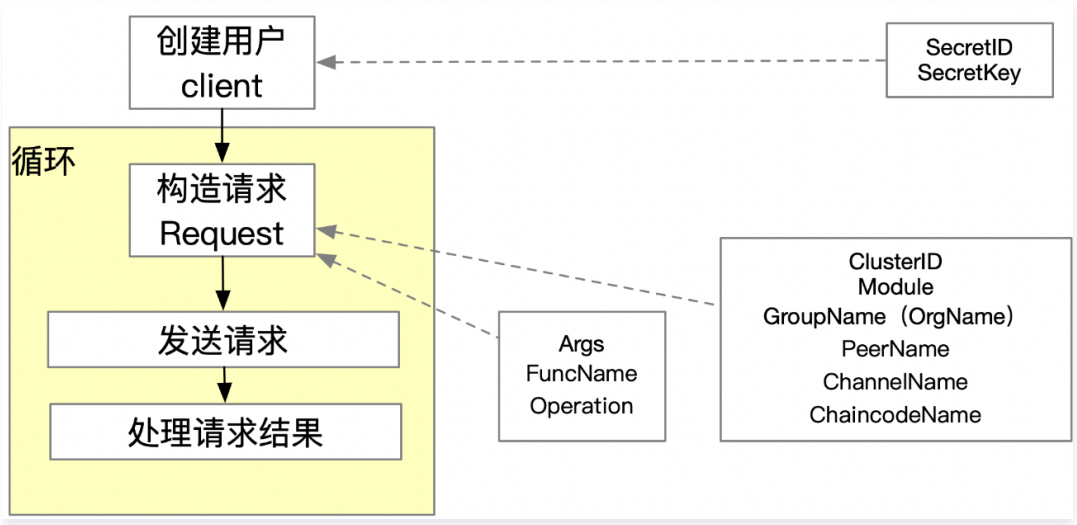
云 API 对接网络
云 API SDK
对接说明

最近更新时间：2022-04-21 14:34:28

云 API 为腾讯云向外提供服务的接口，应用系统需要根据语言集成对应的 SDK 调用云 API，SDK 的集成方法请参考以下对应链接：

- Tencent Cloud SDK 3.0 for Python
- Tencent Cloud SDK 3.0 for Java
- Tencent Cloud SDK 3.0 for PHP
- Tencent Cloud SDK 3.0 for Go
- Tencent Cloud SDK 3.0 for NodeJS
- Tencent Cloud SDK 3.0 for .NET

SDK 的使用流程如下图所示：



云 API 支持长安链 · ChainMaker、Fabric 调用。

获取账户信息

使用云 API 调用合约时除了需要网络、合约的相关参数，还需要提供购买 TBaaS 节点的账户信息，包括 SecretId 和 SecretKey。

> **说明**
> - 如果没有腾讯云账号，请先 注册新账号 。
> - 如果已有腾讯云账号，可以在 API 密钥管理 中获取 `SecretId` 和 `SecretKey` 。

调试接口

在应用系统调用接口之前，若需要对接口进行调试，推荐使用 API 3.0 Explorer 。该工具提供在线调用、签名验证、SDK 代码生成和快速检索接口等能力，能显著降低使用 TBaaS API 的难度。仅需要在页面上输入 API 密钥以及 请求结构 的必要参数。

Go SDK

最近更新时间：2022-10-13 18:13:58

简介

欢迎使用 TBaaS 产品开发者工具套件（SDK）3.0，SDK3.0 是云 API3.0 平台的配套工具。为方便 Go 开发者调试和接入 TBaaS 产品 API，这里向您介绍适用于 Go 的 TBaaS 产品开发工具包，并提供首次使用开发工具包的简单示例。让您快速获取 TBaaS 产品 Go SDK 并开始调用。

依赖环境

1. 依赖环境：Go 1.9版本及以上，并设置好 GOPATH 等必须的环境变量。
2. 通过腾讯云控制台开通 TBaaS 产品。
3. 获取 SecretID、SecretKey 以及调用地址（tbaas.tencentcloudapi.com）。

获取安装

安装 Go SDK 和第一次使用云 API 之前，用户需要在腾讯云控制台上申请并获取安全凭证。安全凭证包括 SecretID 和 SecretKey。SecretID 用于标识 API 调用者的身份，SecretKey 用于加密签名字符串和服务器端验证签名字符串的密钥。SecretKey 必须严格保管，避免泄露。

通过 go get 安装（推荐）

使用语言自带的工具安装 SDK：

```
go get -u github.com/tencentcloud/tencentcloud-sdk-go
```

通过源码包安装

1. 前往 Github 代码托管地址 下载最新代码。
2. 将获取到的源码包解压缩，并安装到 `$GOPATH/src/github.com/tencentcloud` 目录下。

接口列表

详细的接口列表请查看 API文档 。

示例

以新增交易（Invoke）接口为例：

```
package main
import (
        "fmt"
        "github.com/tencentcloud/tencentcloud-sdk-go/tencentcloud/common"
        "github.com/tencentcloud/tencentcloud-sdk-go/tencentcloud/common/errors"
        "github.com/tencentcloud/tencentcloud-sdk-go/tencentcloud/common/profile"
        tbaas "github.com/tencentcloud/tencentcloud-sdk-go/tencentcloud/tbaas/v20180416"
)

func main() {
    // 必要步骤：
    // 实例化一个认证对象，入参需要传入腾讯云账户密钥对secretId，secretKey。
    credential := common.NewCredential("secretId", "secretKey")
    // 实例化一个客户端配置对象，可以指定超时时间等配置
    cpf := profile.NewClientProfile()
    // SDK有默认的超时时间，非必要请不要进行调整。
    // 如有需要请在代码中查阅以获取最新的默认值。
    cpf.HttpProfile.ReqTimeout = 10
     // 设置访问域名
    // SDK会自动指定域名。通常是不需要特地指定域名的，但是如果您访问的是金融区的服务，
    // 则必须手动指定域名，例如云服务器的上海金融区域名： tbaas.ap-shanghai-fsi.tencentcloudapi.com
```

```
    cpf.HttpProfile.Endpoint = "tbaas.tencentcloudapi.com"
    // 实例化Tbaas的client对象
    // 第二个参数是地域信息，根据资源所属地域填写相应的地域信息，例如广州地域的资源可以直接填写字符串ap-guangzhou，或者引
用预设的常量
    client, _ := tbaas.NewClient(credential, regions.Guangzhou, cpf)
    // 实例化一个请求对象，根据调用的接口和实际情况，可以进一步设置请求参数
    request := tbaas.NewInvokeRequest()
    params := "
{\"Module\":\"transaction\",\"Operation\":\"invoke\",\"ClusterId\":\"251005746ctestenv\",\"ChaincodeName
\":\"pettycc1\",\"ChannelName\":\"pettyc1\",\"Peers\":
[{\"PeerName\":\"peer0.pettycorg.ctestenv\",\"OrgName\":\"pettycOrg\"}],\"FuncName\":\"invoke\",\"Args\"
:[\"a\",\"b\",\"10\"],\"AsyncFlag\":0,\"GroupName\":\"pettycOrg\"}"
    err := request.FromJsonString(params)
    if err != nil {
        panic(err)
    }
    // 通过client对象调用想要访问的接口，需要传入请求对象
    response, err := client.Invoke(request)
    // 处理异常
    if _, ok := err.(*errors.TencentCloudSDKError); ok {
        fmt.Printf("An API error has returned: %s", err)
        return
    }
    // 非SDK异常，直接失败。实际代码中可以加入其他的处理。
    if err != nil {
        panic(err)
    }
    // 打印返回的json字符串
    fmt.Printf("%s", response.ToJsonString())
}
```

Java SDK

最近更新时间：2025-05-28 11:24:42

简介

欢迎使用 TBaaS 产品开发者工具套件（SDK）3.0，SDK3.0 是云 API3.0 平台的配套工具。为方便 Java 开发者调试和接入 TBaaS 产品 API，这里向您介绍适用于 Java 的 TBaaS 产品开发工具包，并提供首次使用开发工具包的简单示例。让您快速获取 TBaaS 产品 Java SDK 并开始调用。

依赖环境

1. 依赖环境：JDK 7版本及以上。
2. 通过腾讯云控制台开通 TBaaS 产品。
3. 获取 SecretID、SecretKey 以及调用地址（tbaas.tencentcloudapi.com）。

获取安装

安装 Java SDK 和第一次使用云 API 之前，用户需要在腾讯云控制台上申请并获取安全凭证。安全凭证包括 SecretID 和 SecretKey。 SecretID 用于标识 API 调用者的身份，SecretKey 用于加密签名字符串和服务器端验证签名字符串的密钥。SecretKey 必须严格保管，避免泄露。

通过 Maven 安装（推荐）

Java SDK 推荐通过 Maven 安装。Maven 是 Java 的依赖管理工具，支持您项目所需的依赖项，并将其安装到项目中。关于 Maven 详细介绍可参考 Maven 官网。

1. 前往 Maven 官网 下载对应系统的 Maven 安装包，并进行安装。
2. 在 Maven pom.xml 添加以下内容，为您的项目添加 Maven 依赖项。

```
<dependency>
    <groupId>com.tencentcloudapi</groupId>
    <artifactId>tencentcloud-sdk-java</artifactId>
    <version>3.0.1</version>
</dependency>
```

> **说明：**
> <version> 标签中的版本号为参考示例，请在 Maven 仓库上找到最新的版本进行填写。

3. 引用方法可参考示例。

通过源码包安装

1. 前往 Github 代码托管地址 下载源码压缩包。
2. 将获取到的源码包解压缩到您项目合适的位置。
3. 将 vendor 目录下的 jar 包拷贝到 Java 可找到的路径中。
4. 引用方法可参考示例。

接口列表

详细的接口列表请查看 API文档。

示例

以新增交易（Invoke）接口为例：

```
import com.tencentcloudapi.common.Credential;
import com.tencentcloudapi.common.profile.ClientProfile;
import com.tencentcloudapi.common.profile.HttpProfile;
import com.tencentcloudapi.common.exception.TencentCloudSDKException;
// 导入对应产品模块的client
import com.tencentcloudapi.tbaas.v20180416.TbaasClient;
```

```
// 导入要请求接口对应的request response类
import com.tencentcloudapi.tbaas.v20180416.models.InvokeRequest;
import com.tencentcloudapi.tbaas.v20180416.models.InvokeResponse;

public class InvokeTransaction
{
    public static void main(String [] args) {
        try{
            // 实例化一个认证对象，入参需要传入腾讯云账户密钥对secretId，secretKey
            Credential cred = new Credential("secretId", "secretKey");
            // 设置访问域名
            // SDK会自动指定域名。通常是不需要特地指定域名的，但是如果您访问的是金融区的服务，
            // 则必须手动指定域名，例如云服务器的上海金融区域名： tbaas.ap-shanghai-fsi.tencentcloudapi.com
            HttpProfile httpProfile = new HttpProfile();
            httpProfile.setEndpoint("tbaas.tencentcloudapi.com");
            // 实例化Tbaas的client对象
            ClientProfile clientProfile = new ClientProfile();
            clientProfile.setHttpProfile(httpProfile);
            // 第二个参数是地域信息，根据资源所属地域填写相应的地域信息，例如广州地域的资源可以直接填写字符串ap-
guangzhou，或者引用预设的常量
            TbaasClient client = new TbaasClient(cred, "ap-guangzhou", clientProfile);
            // 实例化一个请求对象，根据调用的接口和实际情况，可以进一步设置请求参数
            String params = "
{\"Module\":\"transaction\",\"Operation\":\"invoke\",\"ClusterId\":\"251005746ctestenv\",\"ChaincodeName
\":\"pettycc1\",\"ChannelName\":\"pettyc1\",\"Peers\":
[{\"PeerName\":\"peer0.pettycorg.ctestenv\",\"OrgName\":\"pettycOrg\"}],\"FuncName\":\"invoke\",\"Args\"
:[\"a\",\"b\",\"10\"],\"AsyncFlag\":0,\"GroupName\":\"pettycOrg\"}";
            InvokeRequest req = InvokeRequest.fromJsonString(params, InvokeRequest.class);
            // 通过client对象调用想要访问的接口，需要传入请求对象
            InvokeResponse resp = client.Invoke(req);
            // 输出json格式的字符串回包
            System.out.println(InvokeResponse.toJsonString(resp));
        } catch (TencentCloudSDKException e) {
                System.out.println(e.toString());
        }
    }
}
```

Python SDK

最近更新时间：2022-10-13 18:10:55

简介

欢迎使用 TBaaS 产品开发者工具套件（SDK）3.0，SDK3.0 是云 API3.0 平台的配套工具。为方便 Python 开发者调试和接入 TBaaS 产品 API，这里向您介绍适用于 Python 的 TBaaS 产品开发工具包，并提供首次使用开发工具包的简单示例。让您快速获取 TBaaS 产品 Python SDK 并开始调用。

依赖环境

1. 依赖环境：Python 2.7到3.6版本。
2. 通过腾讯云控制台开通 TBaaS 产品。
3. 获取 SecretID、SecretKey 以及调用地址（tbaas.tencentcloudapi.com）。

获取安装

安装 Python SDK 和第一次使用云 API 之前，用户需要在腾讯云控制台上申请并获取安全凭证。安全凭证包括 SecretID 和 SecretKey。SecretID 用于标识 API 调用者的身份，SecretKey 用于加密签名字符串和服务器端验证签名字符串的密钥。SecretKey 必须严格保管，避免泄露。

通过 Pip 安装（推荐）

您可以通过执行以下命令，将腾讯云 API Python SDK 安装到您的项目中。如果您的项目环境尚未安装 pip，请参见 pip官网 进行安装。

```
pip install tencentcloud-sdk-python
```

> ⚠ 注意
> 如果您的项目环境中同时搭建 Python2 和 Python3 环境，在 Python3 环境下，请使用 pip3 命令进行安装。

通过源码包安装

1. 前往 Github 代码托管地址 下载最新代码。
2. 将获取到的源码包解压缩，并执行以下命令进行安装。

```
$ cd tencentcloud-sdk-python
$ python setup.py install
```

接口列表

详细的接口列表请查看 API文档。

示例

以新增交易（Invoke）接口为例：

```
# -*- coding: utf-8 -*-
from tencentcloud.common import credential
from tencentcloud.common.profile.client_profile import ClientProfile
from tencentcloud.common.profile.http_profile import HttpProfile
from tencentcloud.common.exception.tencent_cloud_sdk_exception import TencentCloudSDKException
# 导入对应产品模块的client models。
from tencentcloud.tbaas.v20180416 import tbaas_client, models
try:
    # 实例化一个认证对象，入参需要传入腾讯云账户密钥对secretId，secretKey
    cred = credential.Credential("secretId", "secretKey")

    # 配置访问域名
    # SDK会自动指定域名。通常是不需要特地指定域名的，但是如果您访问的是金融区的服务，
```

```
    # 则必须手动指定域名，例如云服务器的上海金融区域名： tbaas.ap-shanghai-fsi.tencentcloudapi.com
    httpProfile = HttpProfile()
    httpProfile.endpoint = "tbaas.tencentcloudapi.com"
    # 实例化Tbaas的client对象
    clientProfile = ClientProfile()
    clientProfile.httpProfile = httpProfile
    #第二个参数是地域信息，根据资源所属地域填写相应的地域信息，例如广州地域的资源可以直接填写字符串ap-guangzhou，或者引用
预设的常量
    client = tbaas_client.TbaasClient(cred, "ap-guangzhou", clientProfile)

    # 实例化一个请求对象，根据调用的接口和实际情况，可以进一步设置请求参数
    params =
'{"Module":"transaction","Operation":"invoke","ClusterId":"251005746ctestenv","ChaincodeName":"pettycc1"
,"ChannelName":"pettyc1","Peers":
[{"PeerName":"peer0.pettycorg.ctestenv","OrgName":"pettycOrg"}],"FuncName":"invoke","Args":
["a","b","10"],"AsyncFlag":0,"GroupName":"pettycOrg"}'
    req = models.InvokeRequest()
    # 调用InvokeRequest的from_json_string方法，使用params初始化req对象
    req.from_json_string(params)

    # 通过client对象调用想要访问的接口，需要传入请求对象
    resp = client.Invoke(req)

    # 输出json格式的字符串回包
    print(resp.to_json_string())

except TencentCloudSDKException as err:
    print(err)
```

云 API 命令行工具

最近更新时间：2024-11-12 09:13:42

简介

腾讯云命令行工具（TCCLI）是管理腾讯云资源的统一工具。通过 TCCLI，您可以快速轻松的调用腾讯云 API 来管理您的腾讯云资源。本文档指导您安装、配置 TCCLI，并通过 TCCLI 调用 TBaaS 对外提供的云 API 接口管理您的资源。

> **注意：**
> TCCLI 基于 Python 对云 API 的调用进行了封装，所以此方式同云 API 一样，可以对长安链 · ChainMaker、Fabric 进行操作，一般用于单次调用，不适用于集成到业务应用系统中。

安装 TCCLI

1. 安装 Python 环境和 Pip 工具。

   > **注意：**
   > Python 版本必须为 2.7 及以上版本，更多内容请参考 Python 主页 和 Pip 主页 。

2. 执行以下命令，安装 TCCLI。

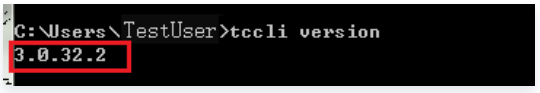
   > **说明：**
   > TCCLI 依赖于 TencentCloudApi Python SDK。如果 TencentCloudApi Python SDK 的版本号小于安装 TCCLI 版本，安装 TCCLI 时，将自动升级 TencentCloudApi Python SDK。

   ```
   pip install tccli
   ```

3. 完成 TCCLI 安装后，执行以下命令，检测是否安装成功。

   ```
   tccli version
   ```

   返回信息类似如下所示，则表示安装成功。

4. （可选）如果您使用的是 Linux 操作系统，执行以下命令，自动补全功能。

   ```
   complete -C 'tccli_completer' tccli
   ```

   > **说明：**
   > Windows 操作系统暂不支持此操作。

配置 TCCLI

使用腾讯云命令行工具，需进行以下初始化配置，使其完成使用 云 API 的必要前提条件。请根据实际需求，选择配置模式进行配置：

交互模式

执行 `tccli configure` 命令，进入交互模式快速配置。

```
$  tccli configure
TencentCloud API secretId [*afcQ]:************************************************************
```

```
TencentCloud API secretKey [*ArFd]:OxXj7khcV******SSYNABcdCc1LiArFd
region: ap-guangzhou
output[json]:
```

- secretId：云 API 密钥 SecretId。
- secretlKey：云 API 密钥 SecretKey。
- region：云产品地域，请切换至对应产品页面获取可用的 region。
- output：可选参数，请求回包输出格式，支持 [json table text] 三种格式，默认为 json。

更多信息请执行 `tccli configure help` 查看。

命令行模式

通过命令行模式您可以在自动化脚本中配置您的信息。

```
# set 子命令可以设置某一配置，也可同时配置多个。
tccli configure set secretId ************************************************************
tccli configure set region ap-guangzhou  output json
# get 子命令用于获取配置信息。
tccli configure get secretKey
secretKey = OxXj7khcV******SSYNABcdCc1LiArFd
# list 子命令打印所有配置信息。
tccli configure list
credential:
secretId =  ************************************************************
secretKey =  OxXj7khcV******SSYNABcdCc1LiArFd
configure:
region =  ap-guangzhou
output =  json
```

更多信息请执行 `tccli configure [list get set] help` 查看。

TCCLI 还支持多账户，方便您同时使用多种配置。以账户名 test 为例：

- 通过交互模式，指定账户名 test。

```
$  tccli configure --profile test
TencentCloud API secretId [*BCDP]:************************************************************
TencentCloud API secretKey [*ArFd]:OxXj7khcV******SSYNABcdCc1LiArFd
region: ap-guangzhou
output[json]:
```

- 通过命令行模式，指定账户名 test。

```
# set/get/list 子命令指定账户名 test。
tccli configure set region ap-guangzhou  output json  --profile test
tccli configure get secretKey      --profile test
tccli configure list      --profile test
在调用接口时指定账户（以 cvm DescribeZones 接口为例）。
tccli cvm DescribeZones --profile test
```

使用 TCCLI

- 通过 `tccli tbaas Invoke` 命令，新增交易（支持同步模式和异步模式）。
- 通过 `tccli tbaas Query` 命令，查询交易。
- 通过 `tccli tbaas GetInvokeTx` 命令，查询 Invoke 异步调用结果。

示例

准备工作

通过 TBaaS 控制台创建并初始化合约。

> **说明：**
> 以转账交易合约为例，初始 a、b 两个账号值为100。

新增交易（同步模式）

请求：

```
tccli tbaas Invoke --Module 'transaction' --Operation 'invoke' --ClusterId '1251568418ndemo' --
ChaincodeName 'tylercc' --ChannelName 'cfirst' --FuncName 'invoke' --Peers
'[{"PeerName":"peer1.aliceorg.ndemo","OrgName":"AliceOrg"}]' --Args '["b", "a", "10"]'
```

返回：

```
{
  "Events": "AliceOrgpeer1.aliceorg.ndemo:VALID",
  "RequestId": "f305f44d-0697-48e2-b85d-e416c4******",
  "Txid": "a73158cc7b628fbcefc329d3fd5de9db9dfaa312528ed09cf4ab121ff6******"
}
```

参数说明：

操作	参数	说明
请求	Module	模块名，固定字段：transaction
	Operation	操作名，固定字段：invoke
	ClusterId	区块链网络 ID，可在区块链网络详情或列表中获取
	ChaincodeName	业务所属智能合约名称，可在智能合约详情或列表中获取
	ChannelName	业务所属通道名称，可在通道详情或列表中获取
	FuncName	该笔交易需要调用的智能合约中的函数名称
	Peers	对该笔交易进行背书的节点列表（包括节点名称和节点所属组织名称，详见数据结构一节），可以在通道详情中获取该通道上的节点名称及其所属组织名称
	Args	被调用的函数参数列表，示例中'["b", "a", "10"]'为b向a转账10
返回	Events	事件处理结果
	RequestId	请求 ID
	Txid	交易 ID

新增交易（异步模式）

请求：

```
tccli tbaas Invoke --Module 'transaction' --Operation 'invoke' --ClusterId '1251568418ndemo' --
ChaincodeName 'tylercc' --ChannelName 'cfirst' --FuncName 'invoke' --Peers
'[{"PeerName":"peer1.aliceorg.ndemo","OrgName":"AliceOrg"}]' --Args '["b", "a", "10"]' --AsyncFlag '1'
```

返回：

```
{
  "Events": "",
  "RequestId": "be95c1e4-ffb9-49e4-a1bb-60770c******",
  "Txid": "1c80cea264a447c9d4d0ba50adbe6e0ff1ac1db313f8d6c83370e410db******"
}
```

参数说明：

操作	参数	说明
请求	Module	模块名，固定字段：transaction
	Operation	操作名，固定字段：invoke
	ClusterId	区块链网络 ID，可在区块链网络详情或列表中获取
	ChaincodeName	业务所属智能合约名称，可在智能合约详情或列表中获取
	ChannelName	业务所属通道名称，可在通道详情或列表中获取
	FuncName	该笔交易需要调用的智能合约中的函数名称
	Peers	对该笔交易进行背书的节点列表（包括节点名称和节点所属组织名称，详见数据结构一节），可以在通道详情中获取该通道上的节点名称及其所属组织名称
	Args	被调用的函数参数列表，示例中'["b", "a", "10"]'为b向a转账10
	AsyncFlag	同步调用标识，可选参数，值为0或者不传表示使用同步方法调用，调用后会等待交易执行后再返回执行结果；值为1时表示使用异步方式调用，执行后会立即返回交易对应的Txid，后续需要通过GetInvokeTx查询该交易的执行结果。（对于逻辑较为简单的交易，可以使用同步模式；对于逻辑较为复杂的交易，建议使用异步模式，否则容易导致API因等待时间过长，返回等待超时）
返回	Events	事件处理结果，异步调用返回空字符串
	RequestId	请求 ID
	Txid	交易 ID

查询交易

请求：

```
tccli tbaas Query --Module 'transaction' --Operation 'query' --ClusterId 'l251568418ndemo' --
ChaincodeName 'tylercc' --ChannelName 'cfirst' --FuncName 'query' --Peers
'[{"PeerName":"peer1.aliceorg.ndemo","OrgName":"AliceOrg"}]' --Args '["a"]'
```

返回：

```
{
  "Data": [
      "110"
  ],
  "RequestId": "237b61b6-c75f-4927-9729-1e117a******"
}
```

参数说明：

操作	参数	说明
请求	Module	模块名，固定字段：transaction
	Operation	操作名，固定字段：query
	ClusterId	区块链网络 ID，可在区块链网络详情或列表中获取
	ChaincodeName	业务所属智能合约名称，可在智能合约详情或列表中获取
	ChannelName	业务所属通道名称，可在通道详情或列表中获取
	FuncName	该笔交易需要调用的智能合约中的函数名称
	Peers	对该笔交易进行背书的节点列表（包括节点名称和节点所属组织名称，详见数据结构一节），可以在通道详情中获取该通道上的节点名称及其所属组织名称
	Args	被调用的函数参数列表，示例中'["a"]'为查询a账号的值
返回	Data	查询结果
	RequestId	请求 ID

查询 Invoke 异步调用结果

请求：

```
tccli tbaas GetInvokeTx --Module 'transaction' --Operation 'query_txid' --ClusterId 'l251568418ndemo' --
ChannelName 'cfirst' --PeerName 'peer1.aliceorg.ndemo' --PeerGroup 'AliceOrg' --TxId
'1c80cea264a447c9d4d0ba50adbe6e0ff1ac1db313f8d6c83370e410db******'
```

返回：

```
{
  "TxValidationCode": 0,
  "RequestId": "6a1d26ba-b322-4aea-8db7-effc82******",
  "TxValidationMsg": "VALID"
}
```

参数说明：

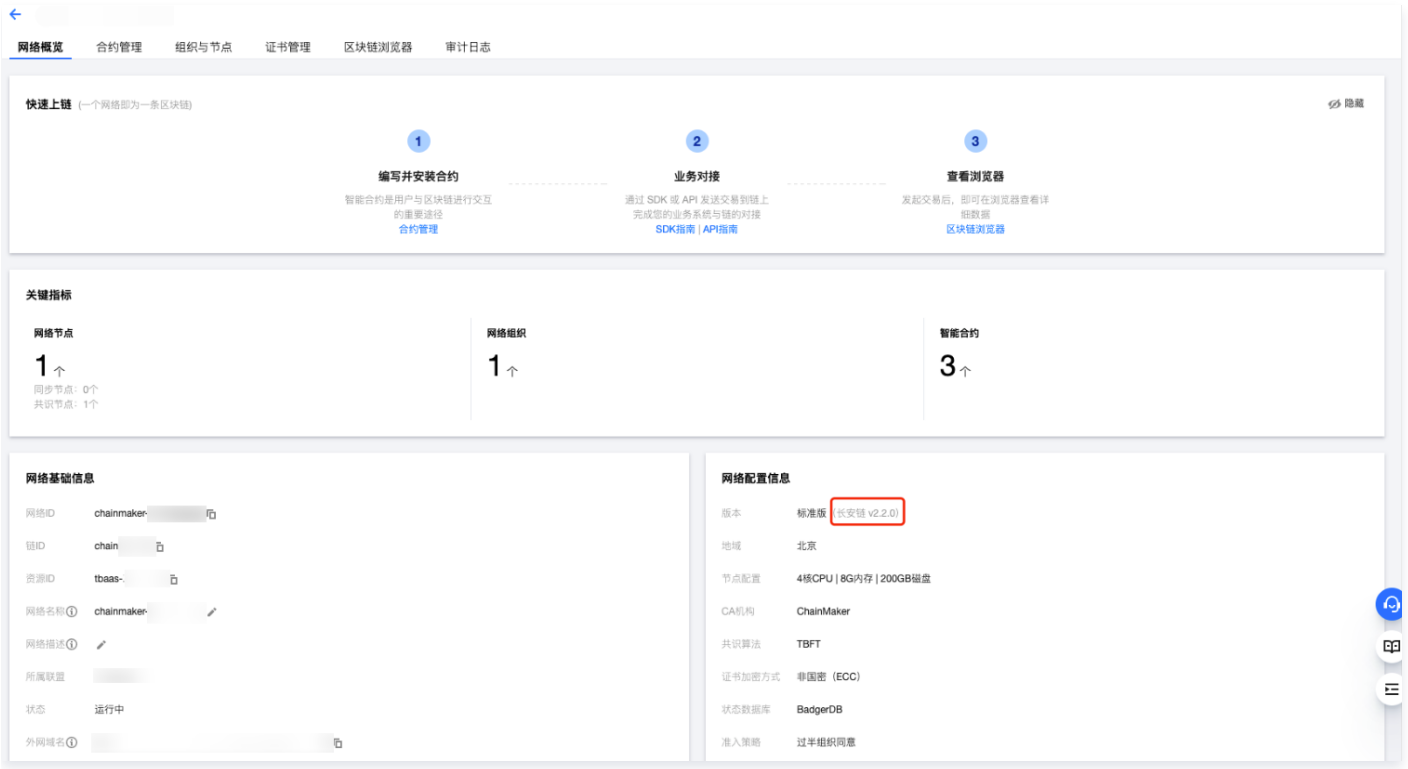
操作	参数	说明
请求	Module	模块名，固定字段：transaction
	Operation	操作名，固定字段：query_txid
	ClusterId	区块链网络 ID，可在区块链网络详情或列表中获取
	ChannelName	业务所属通道名称，可在通道详情或列表中获取
	PeerName	执行该查询交易的节点名称，可以在通道详情中获取该通道上的节点名称及其所属组织名称
	PeerGroup	执行该查询交易的节点所属组织名称，可以在通道详情中获取该通道上的节点名称及其所属组织名称
	TxId	事务 ID
返回	TxValidationCode	交易状态码
	RequestId	请求 ID

	TxValidationMsg	交易状态信息

区块链 SDK 对接网络
长安链 SDK 对接网络
对接 v2.2.1 网络

最近更新时间：2023-02-22 16:04:26

本文为您介绍如何使用长安链 SDK 对接 v2.2.1 版本的长安链网络，长安链网络版本可以从区块链网络概览页右下角的**网络配置信息**中查看。



下载 SDK

下载 chainmaker-sdk-go-demo，压缩包主要包含以下内容：

- **chainmaker-sdk-demo**：目录。
  - **config**：目录，存放 SDK 配置文件，后续证书也可放置在该目录下。
    - **config.yml**：SDK 配置文件。
  - **wasm、evm**：目录，为 SDK 调用合约示例代码。
  - **rust-fact.wasm**：文件，Rust 存证合约示例代码编译后字节码，可根据 智能合约开发（Rust） 指南编译得到，通过 TBaaS 控制台上传该文件可部署该合约。
  - **token.abi**：文件，Solidity 合约示例代码编译后 abi 文件。
  - **token.bin**：文件，Solidity 合约示例代码编译后字节码，可根据 智能合约开发（Solidity） 指南编译得到，通过 TBaaS 控制台上传该文件可部署该合约。
  - **token.sol**：文件，Solidity 合约示例代码。
- **chainmaker-sdk-go**：目录，存放长安链 Go 版本 SDK 源代码。
- **chainmaker-pb-go**：目录，存放长安链 pb-go 库源代码。
- **chainmaker-common**：目录，存放长安链 common 库源代码。

操作步骤

查看长安链 SDK 配置文件

长安链 SDK 配置文件 config.yml 如下所示：

```
chain_client:
  # 链 ID
  chain_id: "chain_txtxt"
  # 组织 ID
  org_id: "orgtxtxtxt.chainmaker-txtxtxtxtx"
  # 客户端用户私钥路径
  user_key_file_path: "../config/user_ecc_tls.key"
  # 客户端用户证书路径
  user_crt_file_path: "../config/user_tls.crt"
  # 客户端用户交易签名私钥路径（若未设置，将使用 user_key_file_path）
  user_sign_key_file_path: "../config/user_ecc_sign.key"
  # 客户端用户交易签名证书路径（若未设置，将使用 user_crt_file_path）
  user_sign_crt_file_path: "../config/user_sign.crt"
  # 同步交易结果模式下，轮训获取交易结果时的最大轮训次数，删除此项或设为<=0 则使用默认值 10
  retry_limit: 10
  # 同步交易结果模式下，每次轮训交易结果时的等待时间，单位：ms 删除此项或设为<=0 则使用默认值 500
  retry_interval: 500

  nodes:
    - # 节点地址，格式为：IP：端口：连接数
      node_addr: "orgtxtxtxt.chainmaker-txtxtxtxtx.tbaas.tech:8080"  #"外网域名：端口"
      # 节点连接数
      conn_cnt: 1
      # RPC 连接是否启用双向 TLS 认证
      enable_tls: true
      # 信任证书池路径，ca.crt 所在路径
      trust_root_paths:
        - "../config/"
      # TLS hostname
      tls_host_name: "common1-orgtxtxtxt.chainmaker-txtxtxtxtx"
  archive:
    # 数据归档链外存储相关配置
    type: "mysql"
    dest: "root:123456:localhost:3306"
    secret_key: xxx
  rpc_client:
    max_receive_message_size: 16 # grpc 客户端接收消息时，允许单条 message 大小的最大值 (MB)
    max_send_message_size: 16 # grpc 客户端发送消息时，允许单条 message 大小的最大值 (MB)
  pkcs11:
    enabled: false # pkcs11 is not used by default
    library: /usr/local/lib64/pkcs11/libupkcs11.so # path to the .so file of pkcs11 interface
    label: HSM # label for the slot to be used
    password: 11111111 # password to logon the HSM(Hardware security module)
    session_cache_size: 10 # size of HSM session cache, default to 10
    hash: "SHA256" # hash algorithm used to compute SKI
```

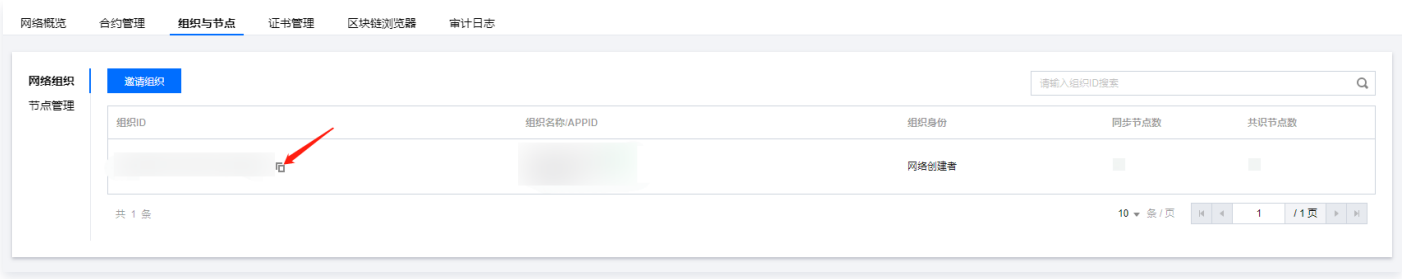
获取访问地址和链 ID

1. 登录 TBaaS 控制台 。
2. 选择左侧导航栏中的**长安链 > 区块链网络**，进入“区块链网络”页面。
3. 在“区块链网络”页面中，选择需查看的网络，单击**管理**进入“网络概览”页面。
4. 在“网络基础信息”模块找到链 ID 和外网域名，分别填写到配置文件的 `chain_id` 和 `node_addr` 字段。

获取组织 ID

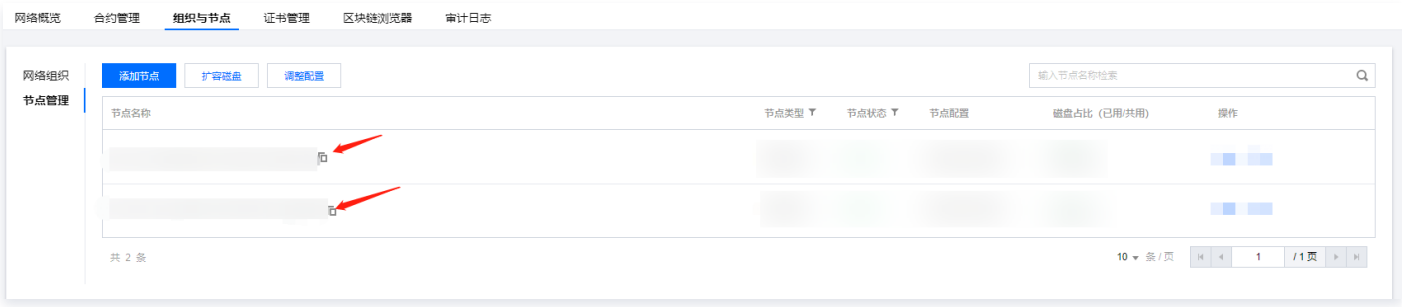
1. 单击**组织与节点**页签，进入“组织与节点”页面。

2. 找到当前组织 ID 信息，填写到配置文件的 `org_id` 字段。如下图所示：



获取节点名称

1. 在“组织与节点”页面单击**节点管理**，进入“节点管理”页面。
2. 找到当前组织的节点列表，选择一个节点并复制节点名称，填写到配置文件的 `tls_host_name` 字段。如下图所示：

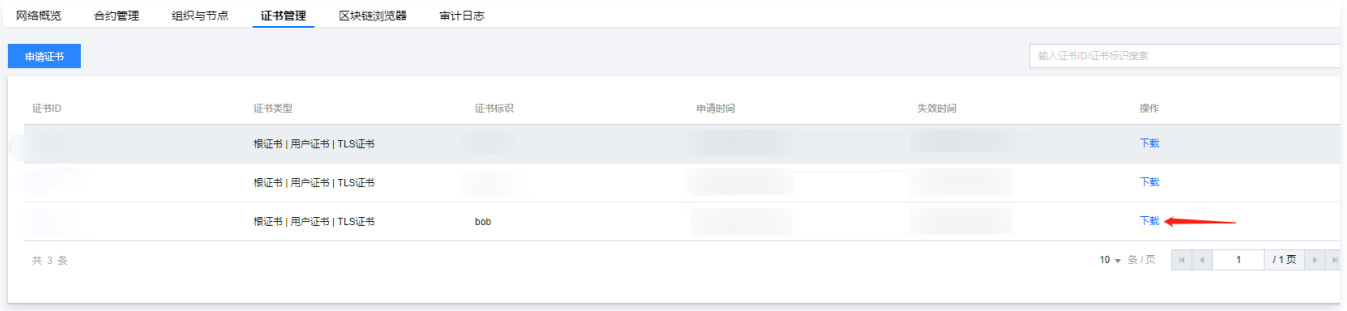


申请证书

1. 根据 证书申请 CSR 生成指南，生成证书 CSR 和相应私钥。
2. 以生成非国密 ECC 证书为例，运行 ecccsr.sh 脚本将会得到以下四种文件：
   - **user_ecc_sign.key**：用户证书私钥。
   - **user_ecc_sign.csr**：用于在 TBaaS 控制台 申请用户证书。
   - **user_ecc_tls.key**：用户 tls 证书私钥。
   - **user_ecc_tls.csr**：用于在 TBaaS 控制台 申请用户 tls 证书。

3. 在“证书管理”界面添加申请证书，填写证书标识，并上传 `user_ecc_sign.csr` 和 `user_ecc_tls.csr` 。如下图所示：

4. 申请成功后可以单击**下载**下载对应证书，压缩包包含如下文件：
   - **ca.crt**：组织根证书。
   - **user_sign.crt**：用户证书。
   - **user_tls.crt**：用户 tls 证书。



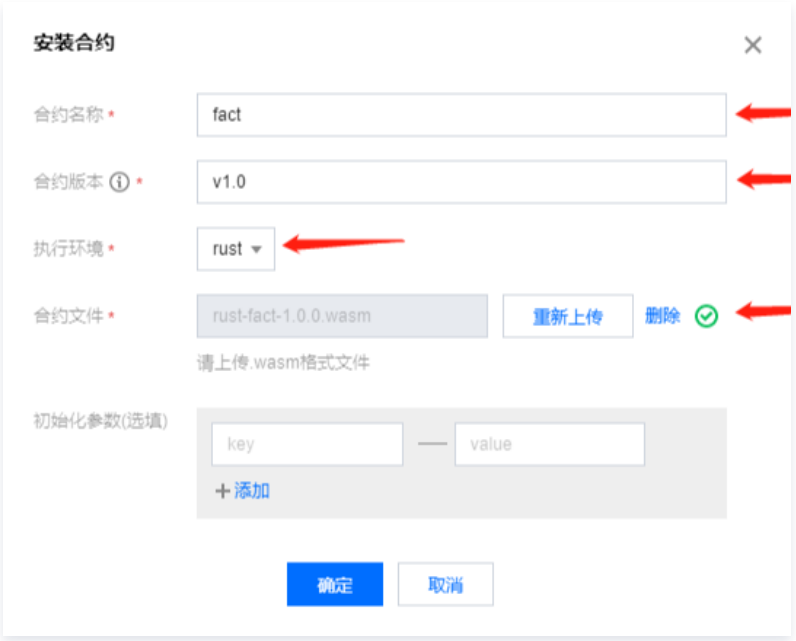
5. 将三个证书 `ca.crt` ， `user_sign.crt` ， `user_tls.crt` 和两个私钥 `user_ecc_sign.key` 和 `user_ecc_tls.key` 放置到 `chainmaker-sdk-go-demo/chainmaker-sdk-demo/config` 目录下。
6. 根据下表，修改 SDK 配置文件 config.yml 的证书路径和 CA 路径：

参数	值	说明
user_key_file_path	./config/user_ecc_tls.key	用户 tls 证书所对应私钥
user_crt_file_path	./config/user_tls.crt	用户 tls 证书
user_sign_key_file_path	./config/user_ecc_sign.key	用户证书所对应私钥
user_sign_crt_file_path	./config/user_sign.crt	用户证书
trust_root_paths	./config/	根 CA 证书 ca.crt 所在目录

部署合约

部署 wasm 合约

1. 单击**合约管理**页签，进入“合约管理”页面
2. 使用压缩包内的 `rust-fact.wasm` 上传，并且进行如下配置：
   - **合约名称**：填写 `fact` 。
   - **版本号**：填写 `v1.0` 。
   - **执行环境**：选择 `Rust` 。
   - **初始化参数（选填）**：保存为空。



3. 部署成功后，合约列表将会展示刚刚部署的合约。如下图所示：

部署 solidity 合约

1. 单击**合约管理**页签，进入“合约管理”页面
2. 根据 合约调用 (Solidity)，将下载的证书文件 user_sign.crt 转换为 EVM 地址
3. 使用压缩包内的 `token.bin` 上传，并且进行如下配置：
   - **合约名称**：填写 `token` 。
   - **版本号**：填写 `v1.0` 。
   - **执行环境**：选择 `Solidity` 。
   - **初始化参数（选填）**：key 为 `_addressFounder` ，value 为 EVM 地址。

4. 部署成功后，合约列表将会展示刚刚部署的合约。如下图所示：

运行 Demo

运行 wasm 合约调用 demo

1. 进入 `chainmaker-sdk-go-demo/chainmaker-sdk-demo/wasm` 目录。
2. 将 `main.go` 示例代码中的 `claimContractName` 修改为上述部署的合约的名字，示例如下：

```
var (
 claimContractName = "fact"
)
```

3. 执行以下命令：

```
go run main.go ../config/config.yml
```

成功运行可以查看到如下图输出：

```
the block height is 28invoke contract success, resp: [code:0]/[msg:OK]/[contractResult:result:"file_[redacted]" gas_used:[redacted] contract_event:<topic:"topic_vx" tx_id:"[redacted]36a7618f9bcda" contract_name:"fact" contract_version:"v1.0" event_data:"[redacted]e1376580b40018" event_data:"file_[redacted]" > ]
<nil>
QUERY claim contract resp: message:"SUCCESS" contract_result:<result:"{\"fileHash\":\"e[redacted]\",\"fileName\":\"file_[redacted]\",\"time\":162920227}" gas_used:[redacted] >
```

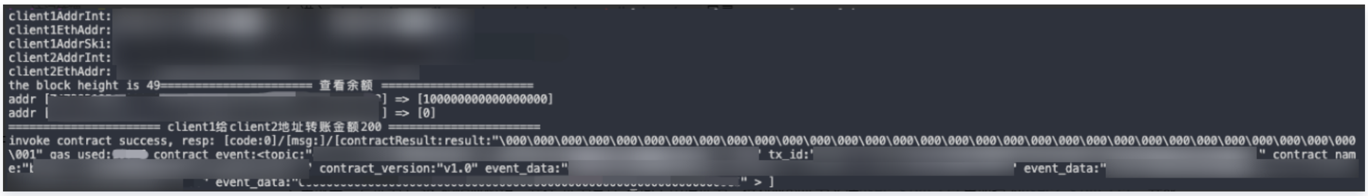
运行 evm 合约调用 demo

1. 进入 `chainmaker-sdk-go-demo/chainmaker-sdk-demo/evm` 目录。
2. 将 `main.go` 示例代码中的 `contractName` 修改为上述部署的 evm 合约的名字，示例如下：

```
var (
 contractName = "token"
```

```
)
```

3. 生成另一套证书 CSR 和相应私钥，并在“证书管理”界面上申请证书，将对应的证书文件 `user_sign.crt` 重命名为 `user2_sign.crt` ，并放置到 `chainmaker-sdk-go-demo/chainmaker-sdk-demo/config` 目录下。
4. 运行 `go run main.go ../config/config.yml`
成功运行可以查看到如下图输出：



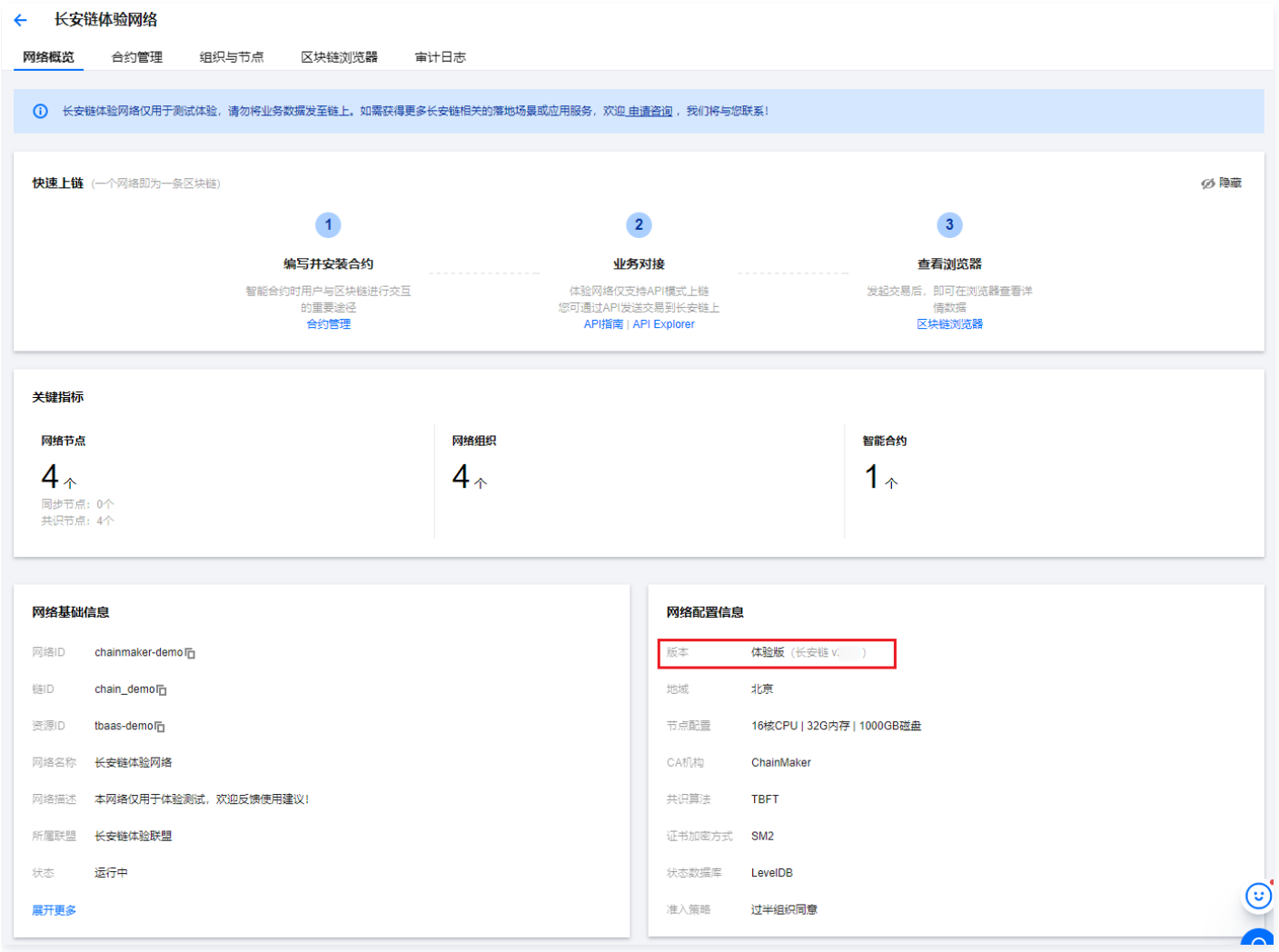
更多对接方式

长安链提供了多种语言的 SDK，包括 Go SDK、Java SDK 等，方便开发者根据需求进行选用。更多详情请参见 长安链 SDK 开发指南 。

对接 v1.2.0 网络

最近更新时间：2023-02-22 16:04:26

本文为您介绍如何使用长安链 SDK 对接 v1.2.0 版本的长安链网络，长安链网络版本可以从 区块链网络概览页 右下角的**网络配置信息**中查看。



下载 SDK

下载 chainmaker-sdk-go-demo，压缩包主要包含以下内容：

- **chainmaker-sdk-demo**：目录。
  - **config**：目录，存放 SDK 配置文件，后续证书也可放置在该目录下。
    - **config.yml**：SDK 配置文件。
  - **wasm、evm**：目录，为 SDK 调用合约示例代码。
  - **go-fact-1.2.0.wasm**：文件，存证合约示例代码编译后字节码，可根据 智能合约开发（Go） 指南编译得到，通过 TBaaS 控制台上传该文件可部署该合约。
  - **token.abi**：文件，Solidity 合约示例代码编译后 abi 文件。
  - **token.bin**：文件，Solidity 合约示例代码编译后字节码，可根据 智能合约开发（Solidity） 指南编译得到，通过 TBaaS 控制台上传该文件可部署该合约。
  - **token.sol**：文件，Solidity 合约示例代码。
- **chainmaker-sdk-go**：目录，存放长安链 Go 版本 SDK 源代码。

操作步骤

查看长安链 SDK 配置文件

长安链 SDK 配置文件 config.yml 如下所示：

```
chain_client:
  # 链 ID
  chain_id: "chain_txtxt"
  # 组织 ID
  org_id: "orgtxtxtxt.chainmaker-txtxtxtxtx"
  # 客户端用户私钥路径
  user_key_file_path: "../config/user_ecc_tls.key"
  # 客户端用户证书路径
  user_crt_file_path: "../config/user_tls.crt"
  # 客户端用户交易签名私钥路径（若未设置，将使用 user_key_file_path）
  user_sign_key_file_path: "../config/user_ecc_sign.key"
  # 客户端用户交易签名证书路径（若未设置，将使用 user_crt_file_path）
  user_sign_crt_file_path: "../config/user_sign.crt"

  nodes:
    - # 节点地址，格式为：IP：端口：连接数
      node_addr: "orgtxtxtxt.chainmaker-txtxtxtxtx.tbaas.tech:8080"  #"外网域名：端口"
      # 节点连接数
      conn_cnt: 1
      # RPC 连接是否启用双向 TLS 认证
      enable_tls: true
      # 信任证书池路径，ca.crt 所在路径
      trust_root_paths:
        - "../config/"
      # TLS hostname
      tls_host_name: "common1-orgtxtxtxt.chainmaker-txtxtxtxtx"
```

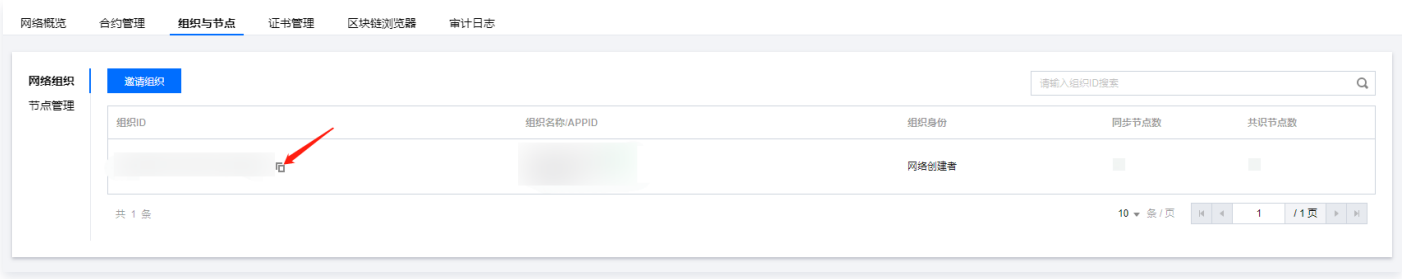
获取访问地址和链 ID

1. 登录 TBaaS 控制台 。
2. 选择左侧导航栏中的**长安链 > 区块链网络**，进入“区块链网络”页面。
3. 在“区块链网络”页面中，选择需查看的网络，单击**管理**进入“网络概览”页面。
4. 在“网络基础信息”模块找到链 ID 和外网域名，分别填写到配置文件的 `chain_id` 和 `node_addr` 字段。

获取组织 ID

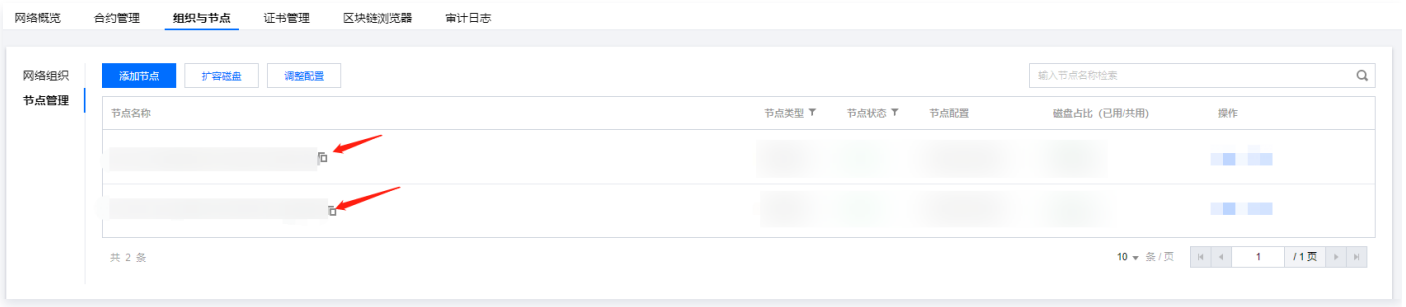
1. 单击**组织与节点**页签，进入“组织与节点”页面。

2. 找到当前组织 ID 信息，填写到配置文件的 `org_id` 字段。如下图所示：



获取节点名称

1. 在“组织与节点”页面单击**节点管理**，进入“节点管理”页面。
2. 找到当前组织的节点列表，选择一个节点并复制节点名称，填写到配置文件的 `tls_host_name` 字段。如下图所示：

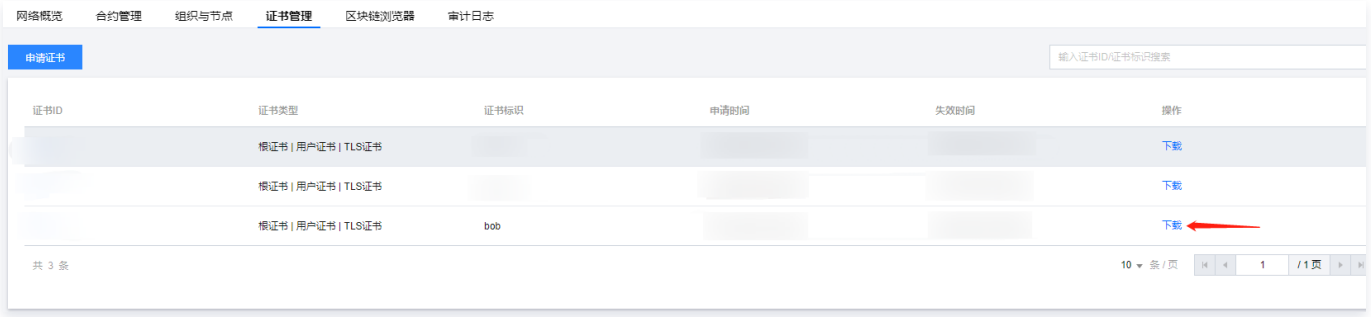


申请证书

1. 根据 证书申请 CSR 生成指南，生成证书 CSR 和相应私钥。
2. 以生成非国密 ECC 证书为例，运行 ecccsr.sh 脚本将会得到以下四种文件：
   - **user_ecc_sign.key**：用户证书私钥。
   - **user_ecc_sign.csr**：用于在 TBaaS 控制台 申请用户证书。
   - **user_ecc_tls.key**：用户 tls 证书私钥。
   - **user_ecc_tls.csr**：用于在 TBaaS 控制台 申请用户 tls 证书。

3. 在“证书管理”界面添加申请证书，填写证书标识，并上传 `user_ecc_sign.csr` 和 `user_ecc_tls.csr` 。如下图所示：

4. 申请成功后可以单击**下载**下载对应证书，压缩包包含如下文件：
   - **ca.crt**：组织根证书。
   - **user_sign.crt**：用户证书。
   - **user_tls.crt**：用户 tls 证书。



5. 将三个证书 `ca.crt` ， `user_sign.crt` ， `user_tls.crt` 和两个私钥 `user_ecc_sign.key` 和 `user_ecc_tls.key` 放置到 `chainmaker-sdk-go-demo/chainmaker-sdk-demo/config` 目录下。
6. 根据下表，修改 SDK 配置文件 config.yml 的证书路径和 CA 路径：

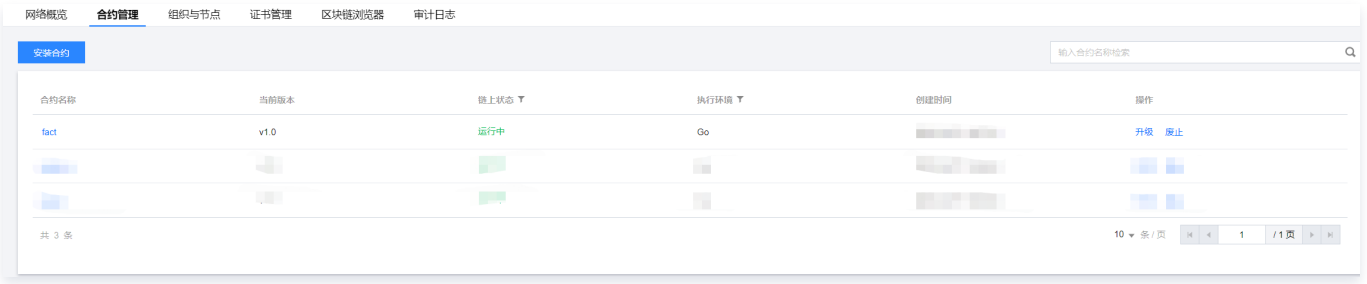
参数	值	说明
user_key_file_path	./config/user_ecc_tls.key	用户 tls 证书所对应私钥
user_crt_file_path	./config/user_tls.crt	用户 tls 证书
user_sign_key_file_path	./config/user_ecc_sign.key	用户证书所对应私钥
user_sign_crt_file_path	./config/user_sign.crt	用户证书
trust_root_paths	./config/	根 CA 证书 ca.crt 所在目录

部署合约

部署 wasm 合约

1. 单击**合约管理**页签，进入“合约管理”页面
2. 使用压缩包内的 `go-fact-1.2.0.wasm` 上传，并且进行如下配置：
   - **合约名称**：填写 `fact` 。
   - **版本号**：填写 `v1.0` 。
   - **执行环境**：选择 `Go` 。
   - **初始化参数（选填）**：保存为空。

3. 部署成功后，合约列表将会展示刚刚部署的合约。如下图所示：



部署 solidity 合约

1. 单击**合约管理**页签，进入“合约管理”页面
2. 根据 合约调用 (Solidity)，将下载的证书文件 user_sign.crt 转换为 EVM 地址，并根据该 EVM 地址生成合约初始化参数的 ABI 编码
3. 使用压缩包内的 `token.bin` 上传，并且进行如下配置：
   - **合约名称**：填写 `token` 。
   - **版本号**：填写 `v1.0` 。
   - **执行环境**：选择 `Solidity` 。
   - **初始化参数（选填）**：key 为 data，value 为 hex 格式的 ABI 编码。

4. 部署成功后，合约列表将会展示刚刚部署的合约。如下图所示：

运行 Demo

运行 wasm 合约调用 demo

1. 进入 `chainmaker-sdk-go-demo/chainmaker-sdk-demo/wasm` 目录。
2. 将 `main.go` 示例代码中的 `claimContractName` 修改为上述部署的合约的名字，示例如下：

```
var (
 claimContractName = "fact"
)
```

3. 执行以下命令：

```
go run main.go ../config/config.yml
```

成功运行可以查看到如下图输出：

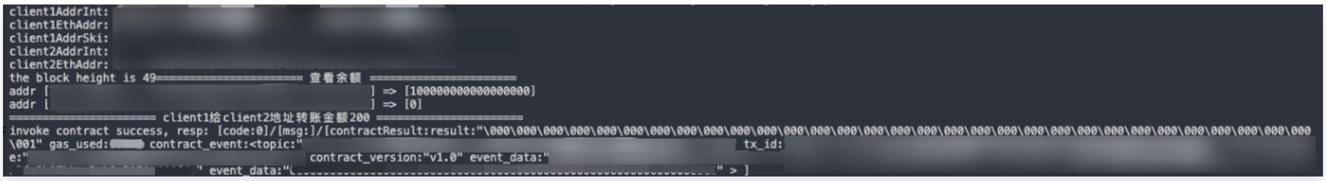
运行 evm 合约调用 demo

1. 进入 `chainmaker-sdk-go-demo/chainmaker-sdk-demo/evm` 目录。
2. 将 `main.go` 示例代码中的 `contractName` 修改为上述部署的 evm 合约的名字，示例如下：

```
var (
 contractName = "token"
```

```
)
```

3. 生成另一套证书 CSR 和相应私钥，并在“证书管理”界面上申请证书，将对应的证书文件 `user_sign.crt` 重命名为 `user2_sign.crt` ，并放置到 `chainmaker-sdk-go-demo/chainmaker-sdk-demo/config` 目录下。
4. 运行 `go run main.go ../config/config.yml`
成功运行可以查看到如下图输出：



更多对接方式

长安链提供了多种语言的 SDK，包括 Go SDK、Java SDK 等，方便开发者根据需求进行选用。更多详情请参见 长安链 SDK 开发指南 。

Fabric SDK 对接网络
对接 v2.3 网络

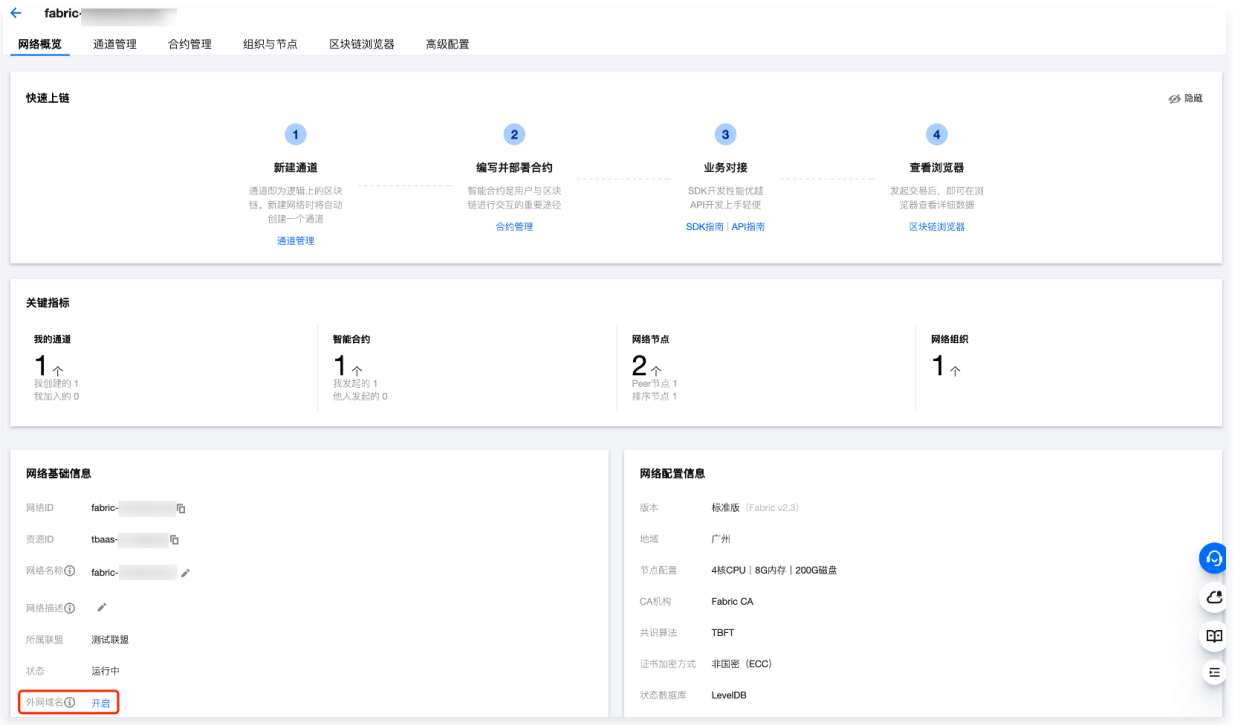
最近更新时间：2024-11-26 16:56:42

> **注意：**
> 区块链 SDK 对接网络调用方式支持 Fabric 网络。

若应用系统调用频率较高，则需要直接使用区块链 SDK 与区块链网络对接，相比云 API 调用方式性能更高。
除了支持社区版区块链 SDK（Java、NodeJS、Golang），TBaaS 对 Golang 版和 Java 版的社区区块链 SDK 进行了定制（tbaas-fabric-sdk-go 和 tbaas-fabric-sdk-java），简化了应用系统与区块链网络连接的流程。

1. 获取网络名和访问地址

1. 登录 TBaaS 控制台，选择左侧导航栏中的 **Fabric > 区块链网络**。
2. 在**区块链网络**页面，选择需查看的网络，单击**管理**进入网络概览页面。
3. 在**网络基础信息**模块找到网络名和访问地址，填写到配置文件的网络名和 url 相关字段。如下图所示：



2. 获取 MSP 和组织名

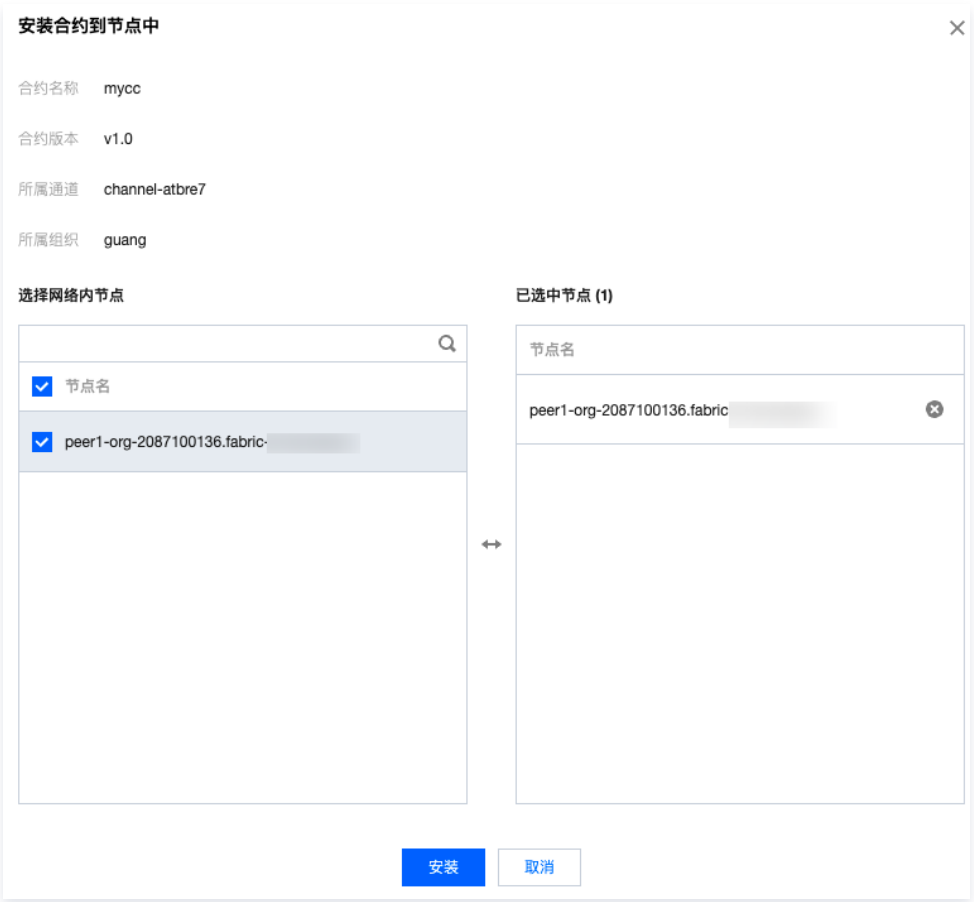
1. 在**组织与节点**页签，选择**网络组织**。
2. 找到当前组织信息，填写到配置文件的 MSP 相关字段；截取 `.` 左边的部分，如 `org-251005746`，填写到配置文件的组织名相关字段。如下图所示：

3. 获取节点名称

1. 在**组织与节点**页签，选择**节点管理**。
2. 找到当前组织的节点列表，选择一个节点并复制节点名称，填写到配置文件的关于节点名字段。如下图所示：

4. 部署合约

1. 在**合约管理**页签，单击**新建合约**。
2. 在**新建合约**中，填写如下基本信息：
   - 合约名称：填写 mycc
   - 合约版本：填写 v1.0
   - 选择合约语言：选择 Go
   - 上传合约压缩包
3. 单击**安装**，并选择 peer 节点。



4. 单击合约实例化，等待合约状态更新为"已实例化"即可。

5. 申请证书流程

1. 前往 OpenSSL 官网，下载 openssl 并配置安装。
2. 下载 ecccsr 工具，解压后执行 `sh ecccsr.sh`，该命令会生成以下四个文件：

- user_ecc_sign.key：为用户证书对应私钥，需安全保存，支持在 SDK 中使用。
- user_ecc_sign.csr：用于在 TBaaS 控制台 申请用户证书。
- user_ecc_tls.key：为用户 tls 证书对应私钥，需安全保存，支持在 SDK 中使用。
- user_ecc_tls.csr：用于在 TBaaS 控制台 申请用户 tls 证书。

3. 在**证书管理**界面添加申请证书，填写证书标识，并上传 user_ecc_sign.csr 和 user_ecc_tls.csr。如下图所示：

4. 申请成功后可以单击下载对应证书，压缩包包含如下文件：
   - ca.crt：组织根证书
   - user_sign.crt：用户证书
   - user_tls.crt：用户 tls 证书
5. 将三个证书 ca.crt，user_sign.crt，user_tls.crt 和两个私钥 user_ecc_sign.key 和 user_ecc_tls.key 放置到对应 SDK 的证书目录下并替换。修改 SDK 配置文件 config.yml 的相应的路径。

6. SDK 使用

6.1 tbaas-fabric-sdk-go

1. 下载进入 tbaas-fabric-sdk-go，进入 tbaas-fabric-sdk-go/fabric-sdk-demo 目录。
2. 将 main.go 示例代码中的 configFile，mspId，username，channelName 和 contractName 进行相应修改，示例如下：

```
const (
    configFile = "config/config.yaml"
    // 组织MSPID
    mspId = "org-251005746.fabric-5x6wdaow3q"
    // 用户名
    username = "test"
    // 通道名
    channelName = "channel-9xjcb2"
    // 合约名
    contractName = "gotest"
)
```

3. 更新配置 yml 文件中相关的内容：

```
version: 1.0.0
```

```
client:
  # 客户端默认使用的组织
  organization: org-251005746
  logging:
    # sdk日志级别
    level: info
  tlsCerts:
    client:
      # 用户TLS私钥路径
      key:
        path: config/test/user_ecc_tls.key
      # 用户TLS证书路径
      cert:
        path: config/test/user_tls.crt
# 通道信息
channels:
  # 通道名
  channel-9xjcb2:
    # peer节点列表
    peers:
      # peer节点名
      peer1-org-251005746.fabric-5x6wdaow3q:
    # orderer节点列表
    orderers:
      - orderer1-org-251005746.fabric-5x6wdaow3q
# 组织信息
organizations:
  # 组织名
  org-251005746:
    # 组织mspId，形式为${orgName}.${clusterId}，orgName为组织名，clusterId为网络ID
    mspid:  org-251005746.fabric-5x6wdaow3q
    # 该组织下的节点列表
    peers:
    - peer1-org-251005746.fabric-5x6wdaow3q
    # 组织用户
    users:
      # 用户名
      test:
        # 用户私钥路径
        key:
          path: config/test/user_ecc_sign.key
        # 用户证书路径
        cert:
          path: config/test/user_sign.crt
# 节点信息
peers:
  # peer节点名
  peer1-org-251005746.fabric-5x6wdaow3q:
    # peer节点url
    url: grpcs://org-251005746-fabric-5x6wdaow3q.tbaas.tech:8080
    grpcOptions:
      # peer节点TLS中的SNI，使用节点名
      ssl-target-name-override: peer1-org-251005746.fabric-5x6wdaow3q
    # TLS CA证书路径
    tlsCACerts:
      path: config/test/ca.crt
entityMatchers:
  peer:
    # 需要配置为发起peer
    - pattern: (\w*)peer1-org-251005746.fabric-5x6wdaow3q(\w*)
```

```
      mappedHost: peer1-org-251005746.fabric-5x6wdaow3q
    # 多组织背书配置如下，前三项需配置为其他背书组织的PeerName，urlSubstitutionExp与orderer的相同
#     - pattern: peer1-org-251005762.fabric-5x6wdaow3q
#       mappedHost: peer1-org-251005762.fabric-5x6wdaow3q
#       sslTargetOverrideUrlSubstitutionExp: peer1-org-251005762.fabric-5x6wdaow3q
#       urlSubstitutionExp: grpcs://org-251005746-fabric-5x6wdaow3q.tbaas.tech:8080
  orderer:
    - pattern: (\w+)
      sslTargetOverrideUrlSubstitutionExp : "${1}"
      urlSubstitutionExp: grpcs://org-251005746-fabric-5x6wdaow3q.tbaas.tech:8080
```

4. 执行以下命令：

```
go run main.go
```

成功运行可以查看到如下图输出：

```
[fabsdk/core] 2023/09/11 06:06:24 UTC - cryptosuite.GetDefault -> INFO No default cryptosuite found, using default SW implementation
[fabsdk/fab] 2023/09/11 06:06:24 UTC - fab.detectDeprecatedNetworkConfig -> WARN Getting orderers from endpoint config channels.orderer is deprecated, use entity matchers to override orderer configuration
[fabsdk/fab] 2023/09/11 06:06:24 UTC - fab.detectDeprecatedNetworkConfig -> WARN visit https://github.com/hyperledger/fabric-sdk-go/blob/master/test/fixtures/config/overrides/local_entity_matchers.yaml for samples
[fabsdk/grpc] 2023/09/11 06:06:24 UTC - comm.(*GrpcLogger).Infoln -> INFO [core]parsed scheme: ""
[fabsdk/grpc] 2023/09/11 06:06:24 UTC - comm.(*GrpcLogger).Infoln -> INFO [core]scheme "" not registered, fallback to default scheme
[fabsdk/grpc] 2023/09/11 06:06:24 UTC - comm.(*GrpcLogger).Infoln -> INFO [core]ccResolverWrapper: sending update to cc: {[{org-251005746-fabric-5x6wdaow3q.tbaas.tech:8080  <nil> 0 <nil>}] <nil> <nil>}
[fabsdk/grpc] 2023/09/11 06:06:24 UTC - comm.(*GrpcLogger).Infoln -> INFO [core]ClientConn switching balancer to "pick_first"
[fabsdk/grpc] 2023/09/11 06:06:24 UTC - comm.(*GrpcLogger).Infoln -> INFO [core]Channel switches to new LB policy "pick_first"
[fabsdk/grpc] 2023/09/11 06:06:24 UTC - comm.(*GrpcLogger).Infoln -> INFO [core]Subchannel Connectivity change to CONNECTING
[fabsdk/grpc] 2023/09/11 06:06:24 UTC - comm.(*GrpcLogger).Infoln -> INFO [core]Subchannel picks a new address "org-251005746-fabric-5x6wdaow3q.tbaas.tech:8080" to connect
[fabsdk/grpc] 2023/09/11 06:06:24 UTC - comm.(*GrpcLogger).Infoln -> INFO [core]Channel Connectivity change to CONNECTING
[fabsdk/grpc] 2023/09/11 06:06:24 UTC - comm.(*GrpcLogger).Infoln -> INFO [core]Subchannel Connectivity change to READY
[fabsdk/grpc] 2023/09/11 06:06:24 UTC - comm.(*GrpcLogger).Infoln -> INFO [core]Channel Connectivity change to READY
[fabsdk/grpc] 2023/09/11 06:06:24 UTC - comm.(*GrpcLogger).Warningln -> WARN [core]Adjusting keepalive ping interval to minimum period of 10s
[fabsdk/grpc] 2023/09/11 06:06:24 UTC - comm.(*GrpcLogger).Warningln -> WARN [core]Adjusting keepalive ping interval to minimum period of 10s
[fabsdk/grpc] 2023/09/11 06:06:24 UTC - comm.(*GrpcLogger).Warningln -> WARN [core]Adjusting keepalive ping interval to minimum period of 10s
[fabsdk/grpc] 2023/09/11 06:06:25 UTC - comm.(*GrpcLogger).Warningln -> WARN [core]Adjusting keepalive ping interval to minimum period of 10s
[fabsdk/grpc] 2023/09/11 06:06:25 UTC - comm.(*GrpcLogger).Infoln -> INFO [core]parsed scheme: ""
[fabsdk/grpc] 2023/09/11 06:06:25 UTC - comm.(*GrpcLogger).Infoln -> INFO [core]scheme "" not registered, fallback to default scheme
[fabsdk/grpc] 2023/09/11 06:06:25 UTC - comm.(*GrpcLogger).Infoln -> INFO [core]ccResolverWrapper: sending update to cc: {[{org-251005746-fabric-5x6wdaow3q.tbaas.tech:8080  <nil> 0 <nil>}] <nil> <nil>}
[fabsdk/grpc] 2023/09/11 06:06:25 UTC - comm.(*GrpcLogger).Infoln -> INFO [core]ClientConn switching balancer to "pick_first"
[fabsdk/grpc] 2023/09/11 06:06:25 UTC - comm.(*GrpcLogger).Infoln -> INFO [core]Channel switches to new LB policy "pick_first"
[fabsdk/grpc] 2023/09/11 06:06:25 UTC - comm.(*GrpcLogger).Infoln -> INFO [core]Subchannel Connectivity change to CONNECTING
[fabsdk/grpc] 2023/09/11 06:06:25 UTC - comm.(*GrpcLogger).Infoln -> INFO [core]Subchannel picks a new address "org-251005746-fabric-5x6wdaow3q.tbaas.tech:8080" to connect
[fabsdk/grpc] 2023/09/11 06:06:25 UTC - comm.(*GrpcLogger).Infoln -> INFO [core]Channel Connectivity change to CONNECTING
[fabsdk/grpc] 2023/09/11 06:06:25 UTC - comm.(*GrpcLogger).Infoln -> INFO [core]Subchannel Connectivity change to READY
[fabsdk/grpc] 2023/09/11 06:06:25 UTC - comm.(*GrpcLogger).Infoln -> INFO [core]Channel Connectivity change to READY
submit transaction: ebd1ca8cc48525d465a954be4c45ecc996158900c2d6b827c9731fb55f165f37
Received block event: &fab.BlockEvent{Block:(*common.Block)(0xc000611700), SourceURL:"grpcs://org-251005746-fabric-5x6wdaow3q.tbaas.tech:8080"}
query b: 100
```

6.2 tbaas-fabric-sdk-java

1. 下载 tbaas-fabric-sdk-java 。
2. 按照 Readme 文档安装 jar 包。
3. 将 eccDemo.java 示例代码中的 wallet, identity, network, username 和 contract 进行相应修改。
4. 更新配置 yml 文件中相关的内容：

```
name: test-net
version: 1.0.0
client:
  # 客户端默认使用的组织
  organization: org-251005746
  logging:
    # sdk日志级别
    level: info
  tlsCerts:
    client:
      # 用户TLS私钥路径
      key:
        path: src/main/resources/crypto-config/eccUser/user_ecc_tls.key
      # 用户TLS证书路径
      cert:
        path: src/main/resources/crypto-config/eccUser/user_tls.crt
# 通道信息
channels:
```

```
  # 通道名
  channel-9xjcb2:
    # peer节点列表
    peers:
      # peer节点名
      peer1-org-251005746.fabric-5x6wdaow3q:
        endorsingPeer: true
        chaincodeQuery: true
        ledgerQuery: true
        eventSource: true
    # orderer节点列表
    orderers:
      - orderer1-org-251005746.fabric-5x6wdaow3q
# 组织信息
organizations:
  # 组织名
  org-251005746:
    # 组织mspId，形式为${orgName}MSP-${clusterId}，orgName为组织名，clusterId为网络ID
    mspid: org-251005746.fabric-5x6wdaow3q
    # 该组织下的节点列表
    peers:
      - peer1-org-251005746.fabric-5x6wdaow3q
    # 组织用户
    users:
      # 用户名
      eccUser:
        # 用户私钥路径
        key:
          path: src/main/resources/crypto-config/eccUser/user_ecc_sign.key
        # 用户证书路径
        cert:
          path: src/main/resources/crypto-config/eccUser/user_sign.crt
# 节点信息
peers:
  # peer节点名
  peer1-org-251005746.fabric-5x6wdaow3q:
    # peer节点url
    url: grpcs://org-251005746-fabric-5x6wdaow3q.tbaas.tech:8080
    grpcOptions:
      # peer节点TLS中的SNI，使用节点名
      hostnameOverride: peer1-org-251005746.fabric-5x6wdaow3q
    # TLS CA证书路径
    tlsCACerts:
      path: src/main/resources/crypto-config/eccUser/ca.crt
# 节点信息
orderers:
  # peer节点名
  orderer1-org-251005746.fabric-5x6wdaow3q:
    # peer节点url
    url: grpcs://org-251005746-fabric-5x6wdaow3q.tbaas.tech:8080
    grpcOptions:
      # peer节点TLS中的SNI，使用节点名
      hostnameOverride: orderer1-org-251005746.fabric-5x6wdaow3q
    # TLS CA证书路径
    tlsCACerts:
      path: src/main/resources/crypto-config/eccUser/ca.crt
entityMatchers:
  peer:
    - pattern: (\w*)peer1-org-251005746.fabric-5x6wdaow3q(\w*)
      mappedHost: peer1-org-251005746.fabric-5x6wdaow3q
```

```
    - pattern: (\w+)
      sslTargetOverrideUrlSubstitutionExp: "${1}"
      urlSubstitutionExp: grpcs://org-251005746-fabric-5x6wdaow3q.tbaas.tech:8080
  orderer:
    - pattern: (\w+)
      sslTargetOverrideUrlSubstitutionExp : "${1}"
      urlSubstitutionExp: grpcs://org-251005746-fabric-5x6wdaow3q.tbaas.tech:8080
```

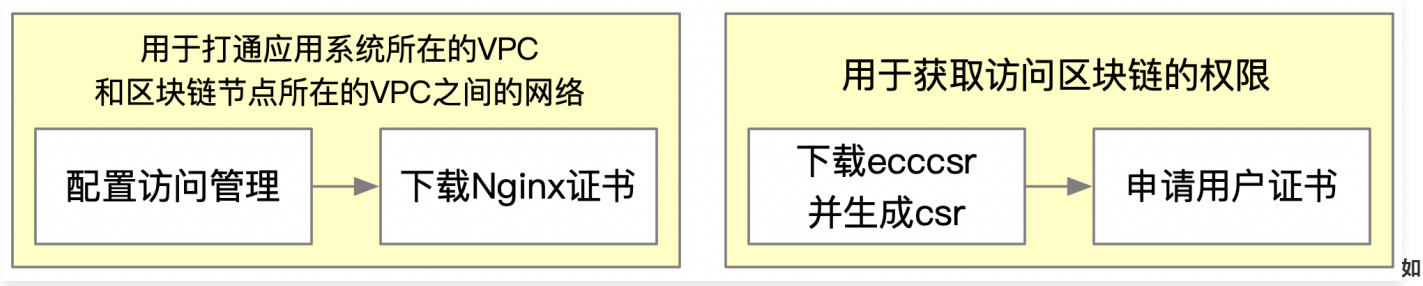
5. 运行 eccDemo.java。
6. 成功运行可以查看到输出为："交易后：20"。

对接 v1.4 网络

最近更新时间：2024-09-04 14:34:21

> **注意：**
> 区块链 SDK 对接网络调用方式支持 Fabric 网络。

若应用系统调用频率较高，则需要直接使用区块链 SDK 与区块链网络对接。这种情况下**应用系统应部署在与区块链网络同一地域内的云服务器 CVM 上**。在云 API 调用方式中，需要应用系统提供账户的 SecretID 和 SecretKey，用于认证授权访问权限。而在区块链 SDK 中，则需要应用系统在 TBaaS 控制台上申请用于访问的证书（节点证书和 nginx 证书）。如下图所示：



如果应用在开发测试中希望在本地访问区块链网络，则可以开启并使用网络的外网域名，使用该域名访问区块链网络的代理组件。这种访问方式仅适用于开发调试，在生产环境中推荐使用访问管理打通 VPC 的方式。
两种访问方式都需要在“访问管理”标签页中获取访问端地址，分别为“外网域名”和“访问端地址”。其区别可参考以下表格：

VPC访问	外网访问
使用内网地址，无法在本地访问。	使用公网域名，可在本地访问。
性能高。	性能低。
可用于生产环境。	一般只用于调试。
应用系统需要部署在与区块链节点同地域的VPC内，并在访问管理页面进行内网打通。	应用系统可部署在本地。

除了支持社区版区块链 SDK（Java、NodeJS、Golang），TBaaS 对 Java 版的社区区块链 SDK 进行了定制（tbaas-fabric-sdk-java），简化了应用系统与区块链网络连接的流程。

获取访问地址及证书

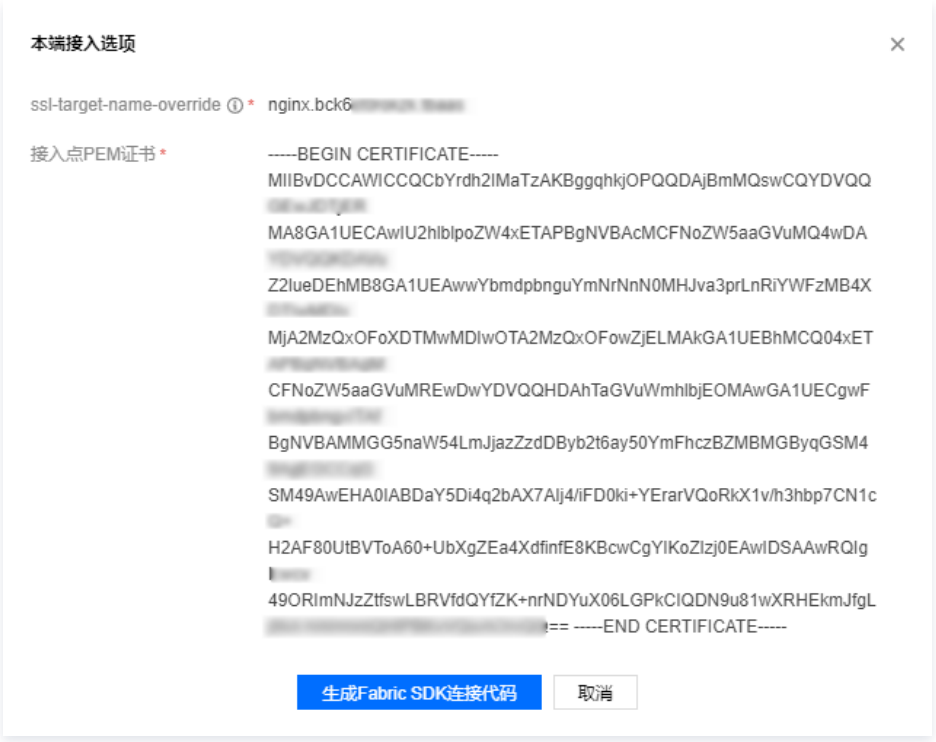
VPC 访问

1. 登录 TBaaS 控制台。
2. 选择左侧导航栏中的 **Fabric > 区块链网络**，进入区块链网络页面。
3. 在**区块链网络**中，选择需查看的网络，进入访问管理页面。
4. 在访问管理中，单击**新建**。在弹出的“新建链接”窗口中，参考以下信息进行创建：

- 名称：即链接标识。
- 选择访问端：即选择应用系统所在的 VPC 和子网。

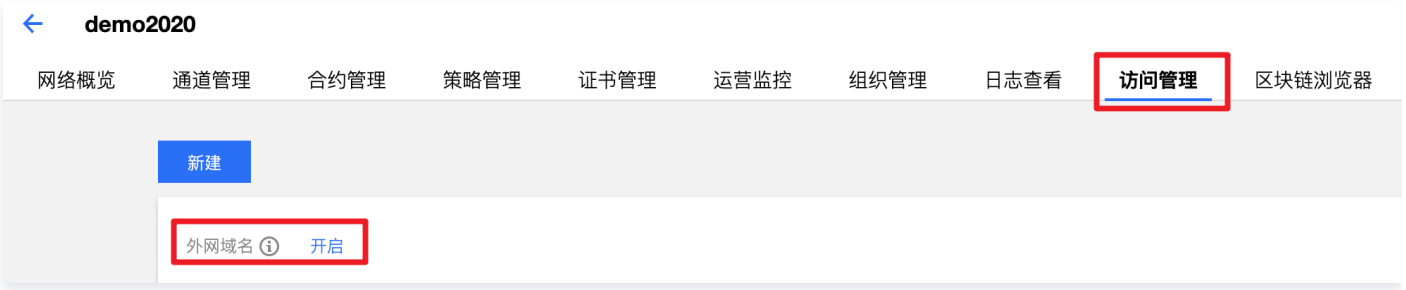
5. 创建成功后即可获取 VPC 访问地址（记为 PROXY_URL），即访问端地址（内网地址）。如下图所示：

在本端链接选项中单击**查看**，并下载 nginx 证书（记为TLS_CERT），保存在文件中。如下图所示：



外网访问（仅用于开发测试）

1. 登录 TBaaS 控制台。
2. 选择左侧导航栏中的 **Fabric > 区块链网络**，进入区块链网络页面。
3. 在**区块链网络**中，选择需查看的网络，进入访问管理页面。
4. 在访问管理中，单击外网域名右侧的**开启**。如下图所示：



获取外网域名后并单击**nginx 证书下载**。

5. 前往 OpenSSL 官网，下载 openssl 并配置安装。
6. 下载 ecccsr 工具，解压后执行 `sh ecccsr.sh` ，得到以下三个文件：
   - out.key
   - out.csr
   - out_sk

申请证书流程

1. 前往 认证信息 页面，查看企业名称。如下图所示：

2. 登录 TBaaS 控制台，选择左侧导航栏中的 **Fabric > 区块链网络**，进入区块链网络页面。
3. 在**区块链网络**中，选择需查看的网络，进入证书管理页面。
4. 在证书管理中单击**申请**，在“申请证书”弹窗中，填写认证信息中的企业名称。如下图所示：

5. 在**证书信息**页面上传通过 外网访问 获取的 `out.csr` 文件。如下图所示：

6. 下载 上一步 申请到的证书，记为 USER_CERT。如下图所示：

经过上述步骤后，得到了访问域名（PROXY_URL）、NGINX证书（TLS_CERT）、out_sk 和用户证书（USER_CERT），在后续的访问中需要使用到这些数据。除此之外，关于网络名、通道名、chaincodeName 等信息的获取方式，请参阅 对接说明及对接前准备 。

tbaas-fabric-sdk-java

下载 tbaas-fabric-sdk-java 。

以下代码示例为不同步骤的代码编写：

1. 配置基本参数。

```
// 通道名称
static String CHANNEL_NAME = "ydtest";
// 开启TLS
static boolean TLS_ENABLE = true;
// 是否通过代理进行访问区块链网络
static boolean PROXY_NETWORK = true;

// 开启TLS需要传证书
static String TLS_CERT = "cert/nginx.bcj4ew1lql10.tbaas.pem";

// "grpcs://"+PROXY_URL
static String PROXY_GRPC_URL = "grpcs://aliceorg-bcj4ew1lql10.tbaas.tech:8080";
```

```
// MSP_ID可以在控制台-组织管理查看
static String MSP_ID = "AliceOrgMSP-bcj4ew1lql10";

// https://cloud.tencent.com/document/product/663/38395
// out.csr、out_sk在本地生成，client.pem在控制台上传out.csr文件后，通过控制台下载
static String USER_KEY_FILE_PATH = "cert/out_sk_20200804";
static String USER_CERT_FILE_PATH = "cert/leyuchen@aliceorg.bcj4ew1lql10@client.pem";

// 控制台-运营监控可查到节点域名/名称
static String PEER_DOMAIN = "peer0.aliceorg.bcj4ew1lql10";
static String PEER_ENDPOINT = "peer0.aliceorg.bcj4ew1lql10:7051";

// 合约名称
static String CHAINCODE_NAME = "ccc";
// 仅在安装合约时需要填写版本号，调用合约不需要填写版本号，默认会调用最新部署版本的合约
private static final String CHAINCODE_VERSION = "";
// 仅在调用GO智能合约时需要添加合约路径
private static final String CHAINCODE_PATH = "";
```

2. 初始化用户并设置访问通道的默认用户。

```
FabricUser user = new FabricUser.Builder()
        .setKeyBytes(FileUtils.getResourceFileBytes(USER_KEY_FILE_PATH))
        .setCertBytes(FileUtils.getResourceFileBytes(USER_CERT_FILE_PATH)) //
FileUtils.getFileBytes("系统中文件的绝对路径")
        .setMspId(MSP_ID)
        .build();
ChannelContext.setDefaultUser(user);
```

3. 连接到通道。

```
Channel demoChannel = ChannelHandler.create()
        .setChannelName(CHANNEL_NAME)
        .addServiceDiscoveryNode(PEER_ENDPOINT)
        .setNetworkType(new ProxyNetworkContext(PROXY_GRPC_URL))
        .setTLSCertBytes(FileUtils.getResourceFileBytes(TLS_CERT))
        .init();
```

4. 创建 fabric 模板。

```
FabricTemplate fabricTemplate = FabricTemplate.getInstance();
```

5. 通过 fabric 模板快速获取通道内信息。

```
// 查询通道内可发现的peer节点
List peerList = fabricTemplate.findPeers(CHANNEL_NAME);
// 获取通道内参与的组织msp列表
List memberList = fabricTemplate.findMemberships(CHANNEL_NAME);
// 获取通道内可发现节点内的合约列表
List chainCodeList = fabricTemplate.findChainCodes(CHANNEL_NAME);
```

6. 通过 fabric 模板调用合约。

```
// 调用智能合约query函数
List queryArgs = Collections.singletonList("a");
FabricQueryResponse response = fabricTemplate.query(where(CHANNEL_NAME).has(CHAINCODE_ID)
        .callFunc("query").addArgs(queryArgs), Integer.class);
```

```
// 调用智能合约并完成交易
List invoke = Arrays.asList("a", "b", "1");
TransactOptions transactOptions = new TransactOptions()
        .waitForBlockEvent(false) // 设置为false则直接返回结果的future，不等交易从peer确认返回，默认值为true
        .eventCallback(Arrays.asList("TEST_EVENT_ID_1"), (b, e) -> LOGGER.debug(e.getChaincodeId() +
new String(e.getPayload())))
        .eventCallback(Arrays.asList("TEST_EVENT_ID_2"), (b, e) -> LOGGER.debug("这是不同的回调函数"));

FabricTransactResponse invokeResponse = fabricTemplate.transact(where(CHANNEL_NAME).has(CHAINCODE_ID)
        .callFunc("invoke").addArgs(invoke), transactOptions);
```

访问管理
访问管理概述

最近更新时间：2023-07-28 15:35:11

存在问题

如果您在腾讯云中使用到了云服务器、私有网络、云数据库等多项服务，这些服务由不同的人管理，但都共享您的云账号密钥，将存在如下问题：

- 您的密钥由多人共享，泄密风险高。
- 您无法限制其它人的访问权限，易产生误操作造成安全风险。

解决方案

您可以通过子账号实现不同的人管理不同的服务来规避以上的问题。默认情况下，子账号没有使用云服务的权利或者相关资源的权限。因此，需要创建策略来允许子账号使用他们所需要的资源或权限。

访问管理（Cloud Access Management，CAM）是腾讯云提供的一套 Web 服务，主要用于帮助用户安全管理腾讯云账户下资源的访问权限。通过 CAM，您可以创建、管理和销毁用户（组），并通过身份管理和策略管理控制指定用户可以使用的腾讯云资源。

当您使用 CAM 的时候，可以将策略与一个用户或一组用户关联起来，策略能够授权或者拒绝用户使用指定资源完成指定任务。有关 CAM 策略的更多基本信息，请参见 策略 。

若您不需要对子账户进行 TBaaS 区块链网络实例的访问管理，您可以跳过此章节，不会影响您理解其余文档及使用产品。

相关操作

CAM 策略必须授权使用一个或多个 TBaaS 区块链网络实例操作，或者必须拒绝使用一个或多个 TBaaS 区块链网络实例操作，同时还必须指定可以用于操作的资源（全部资源或部分资源），策略还可以包含操作资源所设置的条件。您可参考以下文档进行使用：

- 可授权的资源类型
- 授权策略语法
- 授权示例

可授权的资源类型

最近更新时间：2023-01-05 15:02:11

资源级权限指的是能够指定用户对哪些资源具有执行操作的能力。TBaaS 区块链网络实例部分支持资源级权限，即表示针对支持资源级权限的 TBaaS 区块链网络实例操作，您可以控制何时允许用户执行操作或是允许用户使用特定资源。
访问管理 CAM 中可授权的资源类型如下：

资源类型	授权策略中的资源描述方法
区块链网络实例相关	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId

本文将介绍当前支持资源级权限的 TBaaS 区块链网络实例 API 操作，以及每个操作支持的资源和条件密钥。指定资源路径时，您可以在路径中使用 * 通配符。

支持资源级授权的 API 列表

概览和联盟 API 列表

API 操作	资源路径
GetTbaasClusterSummary	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
GetTbaasClusterListByTag	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
GetTbaasClusterListSummary	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
GetTbaasClusterNumberSummary	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId

Fabric 区块链网络 API 列表

API 操作	资源路径
GetClusterList	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
GetGroupListPerAppid	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
ModifyClusterLink	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
GetClusterLinkDetail	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
GetClusterDetail	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
ApplyClientCert	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
GetChaincodeCompileLog	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
AddChannelPeer	qcs::tbaas:$region:$account:resource/*

	qcs::tbaas:$region:$account:resource/$resourceId
ModifyChannelVoteRate	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
CreateChannel	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
GetChaincodePvtDataList	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
InitializeChaincode	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
GetClusterType	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
Query	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
GetInvokeTx	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
Invoke	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
GetGroupListNoChannel	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
GetGroupListNoChaincode	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
GetGroupListPerChannel	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
GetTransactionDetail	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
CheckCertUserIdentity	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
GetGroupListPerCluster	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
GetClusterResourceInfo	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
ApplyUserCert	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
GetBlockTransactionListForUser	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
CheckCreateChaincode	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
CreatePrivateLink	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
GetBlockList	qcs::tbaas:$region:$account:resource/*

	qcs::tbaas:$region:$account:resource/$resourceId
GetLatesdTransactionList	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
GetBlockTransactionList	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
GetBlockDetail	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
CheckChaincodeChannel	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
ReissueCert	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
DownloadCert	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
CheckGroupClusterCreator	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
GetPrivateLinkDetail	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
ModifyPrivateLink	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
GetPrivateLinkList	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
DeletePrivateLink	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
GetUserVpcList	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
GetSrvLogDetail	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
GetLogDetail	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
GetChaincodeDetail	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
GetChaincodeList	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
GetChannelList	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
GetGroupListPerChaincode	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
GetChannelListPerChaincode	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
GetChannelListForInit	qcs::tbaas:$region:$account:resource/*

	qcs::tbaas:$region:$account:resource/$resourceId
GetEndorsementList	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
CreateEndorsement	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
GetEndorsementDetail	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
GetChaincodeListPerEndorsement	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
GetChaincodeListPerChannel	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
GetLogList	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
GetPeerListPerChaincode	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
GetPeerList	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
GetPeerListForInstall	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
GetPeerListForChannel	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
InstallChaincode	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
AddGroupForChaincode	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
ModifyClusterDescription	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
AllocateGrouptoMember	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
CreateChaincode	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
GetGroupListForCloudMonitor	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
GetClusterListForCloudMonitor	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
GetPeerListForCloudMonitor	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
GetChannelListForCloudMonitor	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
GetClusterSummary	qcs::tbaas:$region:$account:resource/*

	qcs::tbaas:$region:$account:resource/$resourceId
GetChannelDetail	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
GetSrvStatusList	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
GetPeerListPerChannel	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
SetChannelAnchorPeer	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
GetCertDetail	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
GetCertList	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
DownloadUserCert	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
GetTransactionDetailForUser	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
CreateChannelAndAdd	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
CreateChaincodeAndInstall	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
CheckResourceModify	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
CheckResourceRenew	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
ModifyResourceRenewFlag	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
PreFeeGetModifyPrice	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId

Bcos 区块链网络 API 列表

API 操作	资源路径
TransByDynamicContractHandler	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
SendTransactionHandler	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
GetBlockListHandler	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
DeployDynamicContractHandler	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId

GetTransByHashHandler	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
BlockByNumberHandler	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
GetTransListHandler	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId
GetClusterPreInfo	qcs::tbaas:$region:$account:resource/* qcs::tbaas:$region:$account:resource/$resourceId

不支持资源级授权的 API 列表

针对不支持资源级权限的区块链网络实例 API 操作，您仍可以向用户授予使用该操作的权限，但策略语句的资源元素必须指定为 `*` 。

API 操作	API 描述
GetTbaasKeySummary	TBaaS 关键指标概览
GetConsortiumList	获取联盟列表
CreateConsortium	创建联盟
GetConsortiumMembers	获取联盟成员列表
GetConsortiumDetail	获取联盟详情
GetConsortiumMemberAuthData	获取成员认证信息
InviteConsortiumMember	邀请成员加入联盟
ExitConsortium	联盟退出
GetEventList	获取事件列表
GetEventSummary	获取事件中心概览
GetChannelVotersForEvent	获取通道投票参与者信息
GetClusteMemberForEvent	获取事件中心网络成员
GetEventDetail	获取事件详情
GetConsortiumDtailForEvent	获取事件中心联盟详情
GetConsortiumMemberForEvent	获取事件中心联盟成员
GetChannelInviteesForEvent	获取通道被邀请成员信息
GetChannelDetailForEvent	获取事件中心通道概要
DealEventTask	处理代办任务
GetEventStepStatus	获取事件流程状态
GetClusteDetailForEvent	获取事件中心网络详情
GetUserAuthType	查询用户认证类型
PreFeeGetPrice	预付费询价
CheckResourceCreate	资源创建参数校验
GetChaincodeTemplate	查询 Chaincode 模板

GetCosUrl	获取 Cos 的下载链接
GetCosSign	查询 Cos 的 Sign
CompileChaincode	验证智能合约是否合法
ExportChaincodeDev	IDE 智能合约导出
UploadChaincodeDev	IDE 智能合约上传
RunChaincodeDev	IDE 智能合约执行
GetChaincodeDevAccessAuth	在线编辑器访问权限查询
AsynCheckChaincodeDev	IDE 智能合约异步操作结果查询
AsynCompileChaincodeDev	IDE 智能合约异步编译
BcosPreFeeGetPrice	Bcos 预付费询价
GetUserListHandler	Bcos 分页查询公私钥信息列表
UpdateKeyUserHandler	Bcos 修改私钥用户的备注

授权策略语法

最近更新时间：2023-07-28 15:34:11

策略语法

CAM 策略：

```
{
        "version":"2.0",
        "statement":
        [
           {
              "effect":"effect",
              "action":["action"],
              "resource":["resource"],
               "condition": {"key":{"value"}}
           }
       ]
}
```

- **版本 version**：必填项，目前仅允许值为"2.0"。
- **语句 statement**：用来描述一条或多条权限的详细信息。该元素包括 effect、action、resource、condition 等多个其他元素的权限或权限集合。一条策略有且仅有一个 statement 元素。
- **影响 effect**：必填项，描述声明产生的结果是“允许”还是“显式拒绝”。包括 allow（允许）和 deny（显式拒绝）两种情况。
- **操作 action**：必填项，用来描述允许或拒绝的操作。操作可以是 API（以 `tbaas:` 为前缀描述）。
- **资源 resource**：必填项，描述授权的具体数据。资源是用六段式描述，每款产品的资源定义详情会有所区别。
- **生效条件 condition**：必填项，描述策略生效的约束条件。条件包括操作符、操作键和操作值组成。条件值可包括时间、IP 地址等信息，有些服务允许您在条件中指定其他值。

TBaaS 区块链网络实例的操作

在 TBaaS 区块链网络实例策略语句中，您可以指定任意的 API 操作。对于区块链网络实例，请使用以 `tbaas:` 为前缀的 API 。例如 `tbaas:GetClusterList` 或者 `tbaas:CreateChannel` 。
如果您要在单个语句中指定多个操作的时候，请使用逗号将它们隔开，如下所示：

```
"action":["tbaas:action1","tbaas:action2"]
```

您也可以使用通配符指定多项操作。例如，您可以指定名字以单词 "Get" 开头的所有操作，如下所示：

```
"action":["tbaas:Get*"]
```

如果您要指定 TBaaS 区块链网络实例中所有操作，请使用 `*` 通配符，如下所示：

```
"action": ["tbaas:*"]
```

区块链网络实例的资源

每个 CAM 策略语句都有适用于自己的资源。
资源的一般形式如下：

```
qcs:project_id:service_type:region:account:resource
```

- **project_id**：描述项目信息，仅为了兼容 CAM 早期逻辑，无需填写。
- **service_type**：产品简称，如 TBaaS。

- **region**：地域信息，如 ap-guangzhou。
- **account**：资源拥有者的主账号信息，如 uin/653339763。
- **resource**：各产品的具体资源详情，如 resource/tbaas-xxx 或者 resource/*。

例如，您可以使用特定实例（tbaas-k05xdcta）在语句中指定它，如下所示：

```
"resource":[ "qcs::tbaas:ap-guangzhou:uin/653339763:resource/tbaas-k05xdcta"]
```

您还可以使用 `*` 通配符指定属于特定账户的所有实例，如下所示：

```
"resource":[ "qcs::tbaas:ap-guangzhou:uin/653339763:resource/*"]
```

您要指定所有资源，或者如果特定 API 操作不支持资源级权限，请在 resource 元素中使用 * 通配符，如下所示：

```
"resource": ["*"]
```

如果您想要在一条指令中同时指定多个资源，请使用逗号将它们隔开，如下所示为指定两个资源的例子：

```
"resource":["resource1","resource2"]
```

下表描述了 TBaaS 区块链网络实例能够使用的资源和对应的资源描述方法。

资源	授权策略中的资源描述方法
实例	`qcs::tbaas:$region:$account:resource/$resourceId` 其中： • $ 为前缀的单词均为代称。 • project 指项目 ID。 • region 指地域。 • account 指账户 ID。

授权示例

最近更新时间：2022-10-13 18:22:51

操作场景

您可以使用访问管理 CAM 策略让用户拥有在腾讯云区块链服务平台 TBaaS 控制台中查看和使用特定资源的权限。本文中的示例向您介绍如何使用控制台的特定策略。

操作示例

TBaaS 区块链网络实例的全读写策略

如果您希望用户拥有创建和管理 TBaaS 区块链网络实例的权限，您可以对该用户使用名称为：QcloudTBAASFullAccess 的策略。
具体操作步骤如下：
参考 授权管理，将预设策略 QcloudTBAASFullAccess 授权给用户。

TBaaS 区块链网络实例的只读策略

如果您希望用户拥有查询 TBaaS 区块链网络实例的权限，但是不具有创建、删除和修改的权限，您可以对该用户使用名称为：QcloudTBAASReadOnlyAccess 的策略。
具体操作步骤如下：
参考 授权管理，将预设策略 QcloudTBAASReadOnlyAccess 授权给用户。

授权用户拥有非资源级的 API 接口的操作权限策略

如果您希望用户拥有非资源级的 API 接口的操作权限，创建策略并将其关联到对应的用户。策略内容可参考以下策略语法进行设置：

```
{
    "version": "2.0",
    "statement": [
        {
            "action": [
                "tbaas:GetTbaasKeySummary",
                "tbaas:GetConsortiumList",
                "tbaas:CreateConsortium",
                "tbaas:GetConsortiumMembers",
                "tbaas:GetConsortiumDetail",
                "tbaas:GetConsortiumMemberAuthData",
                "tbaas:InviteConsortiumMember",
                "tbaas:ExitConsortium",
                "tbaas:GetEventList",
                "tbaas:GetEventSummary",
                "tbaas:GetChannelVotersForEvent",
                "tbaas:GetClusteMemberForEvent",
                "tbaas:GetEventDetail",
                "tbaas:GetConsortiumDtailForEvent",
                "tbaas:GetConsortiumMemberForEvent",
                "tbaas:GetChannelInviteesForEvent",
                "tbaas:GetChannelDetailForEvent",
                "tbaas:DealEventTask",
                "tbaas:GetEventStepStatus",
                "tbaas:GetClusteDetailForEvent",
                "tbaas:GetUserAuthType",
                "tbaas:PreFeeGetPrice",
                "tbaas:CheckResourceCreate",
                "tbaas:GetChaincodeTemplate",
                "tbaas:GetCosUrl",
                "tbaas:GetCosSign",
                "tbaas:CompileChaincode",
```

```
                "tbaas:ExportChaincodeDev",
                "tbaas:UploadChaincodeDev",
                "tbaas:RunChaincodeDev",
                "tbaas:GetChaincodeDevAccessAuth",
                "tbaas:AsynCheckChaincodeDev",
                "tbaas:AsynCompileChaincodeDev",
                "tbaas:BcosPreFeeGetPrice",
                "tbaas:GetUserListHandler",
                "tbaas:UpdateKeyUserHandler"
            ],
            "effect": "allow",
            "resource": "*"
        }
    ]
}
```

授权用户拥有特定 TBaaS 区块链网络实例的操作权限策略

如果您希望用户拥有特定 TBaaS 区块链网络实例操作权限，可将以下策略关联到该用户。以下策略允许用户拥有对资源 ID 为 `tbaas-xxx`，广州地域的 TBaaS 区块链网络实例的操作权限：

```
{
    "version": "2.0",
    "statement": [
        {
            "action": "tbaas:*",
            "resource": "qcs::tbaas:ap-guangzhou::resource/tbaas-xxx",
            "effect": "allow"
        }
    ]
}
```

授权用户拥有批量 TBaaS 区块链网络实例的操作权限策略

如果您希望用户拥有批量 TBaaS 区块链网络实例操作权限，可将以下策略关联到该用户。以下策略允许用户拥有对资源 ID 为 `tbaas-xxx`、`tbaas-yyy`，广州地域的 TBaaS 区块链网络实例的操作权限和对资源 ID 为 `tbaas-zzz`，北京地域的 TBaaS 区块链网络实例的操作权限。

```
{
    "version": "2.0",
    "statement": [
        {
            "action": "tbaas:*",
            "resource": [
                "qcs::tbaas:ap-guangzhou::resource/tbaas-xxx",
                "qcs::tbaas:ap-guangzhou::resource/tbaas-yyy",
                "qcs::tbaas:ap-beijing::resource/tbaas-zzz"
            ],
            "effect": "allow"
        }
    ]
}
```

授权用户拥有特定地域 TBaaS 区块链网络实例的操作权限策略

如果您希望用户拥有特定地域的 TBaaS 区块链网络实例的操作权限，可将以下策略关联到该用户。以下策略允许用户拥有对广州地域的 TBaaS 区块链网络实例的操作权限。

```
{
```

```
    "version": "2.0",
    "statement": [
        {
            "action": "tbaas:*",
            "resource": "qcs::tbaas:ap-guangzhou::*",
            "effect": "allow"
        }
    ]
}
```

授权用户拥有特定标签的 TBaaS 区块链网络实例的操作权限策略

如果您希望用户拥有特定地域的 TBaaS 区块链网络实例的操作权限，可将以下策略关联到该用户。以下策略允许用户拥有对广州地域的 TBaaS 区块链网络实例的操作权限。

```
{
    "version": "2.0",
    "statement": [
        {
            "effect": "allow",
            "action": "*",
            "resource": "*",
            "condition": {
                "for_any_value:string_equal": {
                    "qcs:tag": [
                        "qta&camtest"
                    ]
                }
            }
        }
    ]
}
```

自定义策略

如果您认为预设策略不能满足您的要求，您可以通过创建自定义策略达到目的。若按照资源进行授权，针对不支持资源级权限的 TBaaS 区块链网络实例 API 操作，您仍可以向用户授予使用该操作的权限，但策略语句的资源元素必须指定为 `*` 。
策略内容可参考以下策略语法进行设置：

```
{
    "version": "2.0",
    "statement": [
        {
            "action": [
                "Action"
            ],
            "resource": "Resource",
            "effect": "Effect"
        }
    ]
}
```

- Action 中内容换成您要进行允许或拒绝的操作。
- Resource 中内容换成您要授权的具体资源。
- Effect 中内容换成允许或者拒绝。