WeCross Documentation

发布 ***v1.0.0-rc4***

WeCross Community

2020 年 **08** 月 **17** 日

Contents

由微众银行自主研发并完全开源的分布式商业区块链跨链协作平台。该平台能解决业界主流的区块链产品间接口不互通、无法协作的问题，以及区块链系统无法平行扩展、计算能力和存储容量存在瓶颈等问题。WeCross作为未来分布式商业区块链互联的基础架构，秉承公众联盟链多方参与、共享资源、智能协同和价值整合的理念，致力于促进跨行业、机构和地域的跨区块链价值交换和商业合作，实现了高效、通用和安全的区块链跨链协作机制。

4S平台优势

跨链业务高效协同
Synergetic

根据"一次适配，随处可用"原则，提炼跨链交互必需的"核心接口子集"，设计通用数据结构和网络协议，解决因设计目标不同而导致的各平台接口差异性难题。

跨链操作安全可信
Secure

引入CA身份认证机制，对通信链路进行加密加固，严格限制访问权限，设计多维度的默克尔证明机制，以及多种原子事务机制，保障跨链交互全流程数据的可信性。

跨链网络分层可扩展
Scalable

设计跨链路由协议与模块，支持多条区块链分布式互联，承载各种拓扑架构，支持多层次纵深跨链协作。同时有多方共建、共治的治理架构，实现跨链网络的可持续扩展。

跨链接入高效便捷
Speedy

设计通用SDK、交互式控制台以及可视化浏览器等全套开发组件，简化跨链交互操作流程，还有"所见即所得"的运维工具，一键发起跨链操作。

核心技术

UBI通用区块链接口
Universal Blockchain Interface

通用的区块链数据协议，抽象提炼主流区块链共通的核心数据结构与资源定义，使多种区块链平台可以用统一的数据协议进行交互，极大程度减小区块链平台之间的交互难度。

HIP异构链互联协议
Heterogeneous Interchain Protocol

主流区块链平台通用的网络交互协议及统一的交互模式，只需简便适配，即可实现异构区块链平台的连通。

CHAPTER 1

平台介绍

1.1 基本介绍

区块链作为构建未来价值互联网的重要基础设施，深度融合分布式存储、点对点通信、分布式架构、共识机制、密码学等前沿技术，正在成为技术创新的前沿阵地。全球主要国家都在加快布局区块链技术，用以推动技术革新和产业变革。经过行业参与者十年砥砺前行，目前区块链在底层技术方案上已趋于完整和成熟，国内外均出现可用于生产环境的区块链解决方案。其所面向的创新应用场景覆盖广泛，已在对账与清结算、跨境支付、供应链金融、司法仲裁、政务服务、物联网、智慧城市等众多领域落地企业级应用。

在广泛的场景应用背后，来自于性能、安全、成本、扩展等方面的技术挑战也愈发严峻。目前不同区块链应用之间互操作性不足，无法有效进行可信数据流通和价值交换，各个区块链俨然成为一座座信任孤岛，很大程度阻碍了区块链应用生态的融合发展。未来，区块链想要跨越到真正的价值互联网，承担传递信任的使命，开启万链互联时代，需要一种通用、高效、安全的区块链跨链协作机制，实现跨场景、跨地域不同区块链应用之间的互联互通，以服务数量更多、地域更广的公众群体。

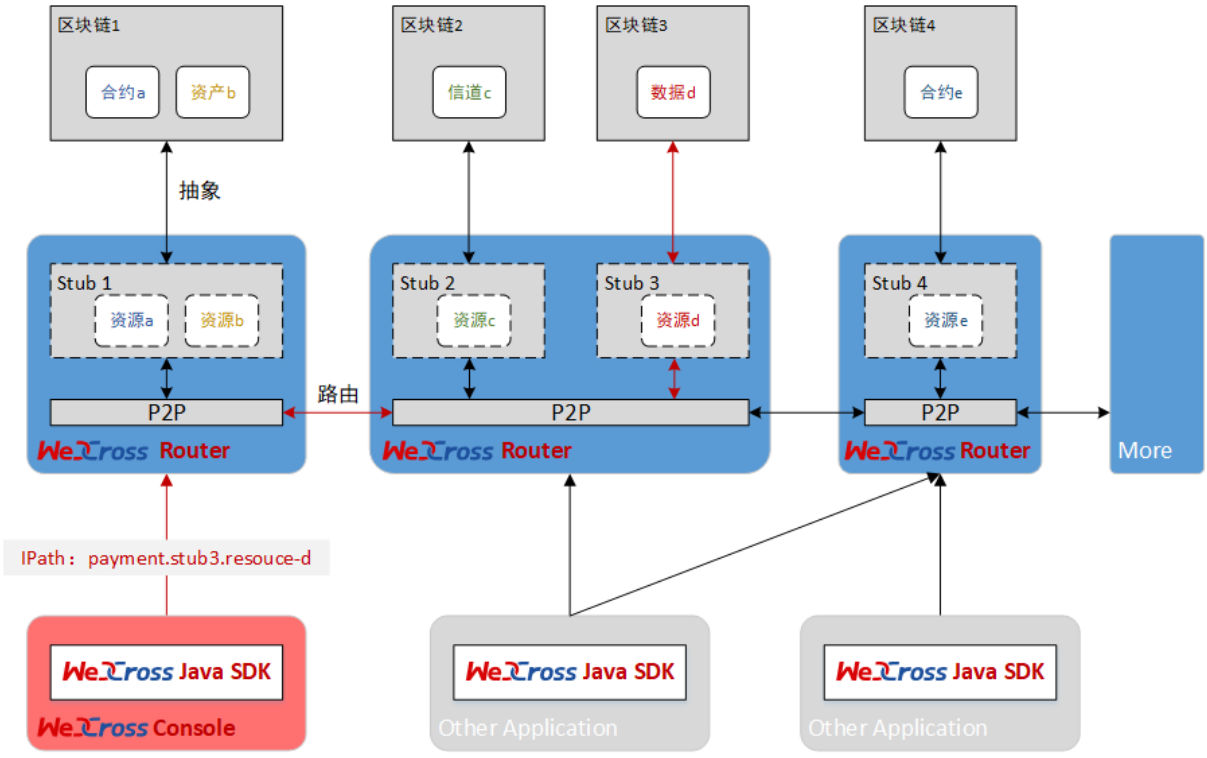
作为一家具有互联网基因的高科技、创新型银行，微众银行自成立之初即高度重视新兴技术的研究和探索，在区块链领域积极开展技术积累和应用实践，不断致力于运用区块链技术提升多机构间的协作效率和降低协作成本，支持国家推进关键技术安全可控战略和推动社会普惠金融发展。微众银行区块链团队基于一揽子自主研发并开源的区块链技术方案，针对不同服务形态、不同区块链平台之间无法进行可信连接与交互的行业痛点，研发区块链跨链协作平台——WeCross，以促进跨行业、机构和地域的跨区块链信任传递和商业合作。

WeCross 着眼应对区块链行业现存挑战，不局限于满足同构区块链平行扩展后的可信数据交换需求，还进一步探索异构区块链之间因底层架构、数据结构、接口协议、安全机制等多维异构性导致无法互联互通问题的有效解决方案。作为未来区块链互联的基础设施，WeCross 秉承多方参与、共享资源、智能协同和价值整合的理念，面向公众完全开源，欢迎广大企业及技术爱好者踊跃参与项目共建。

1.2 关键词

- **跨链路由（WeCross Router）**
  - 与链对接，对链上的资源进行抽象
  - 向外暴露统一的接口
  - 将调用请求路由至对应的区块链

- **控制台（WeCross Console）**
  - 命令行式的交互
  - 查询跨链信息，发送调用请求
- **跨链 SDK（WeCross Java SDK）**
  - WeCross开发工具包，供开发者调用WeCross
  - 集成于各种跨链APP中，提供统一的调用接口
  - 与跨链路由建立连接，调用跨链路由
- **跨链资源（Resource）**
  - 各种区块链上内容的抽象
  - 包括：合约、资产、信道、数据表
- **跨链适配器（Stub）**
  - 跨链路由中对接入的区块链的抽象
  - 跨链路由通过配置Stub与相应的区块链对接
  - FISCO BCOS需配置FISCO BCOS Stub、Fabric需配置Fabric Stub
- **IPath（Interchain Path）**
  - 跨链资源的唯一标识
  - 跨链路由根据IPath将请求路由至相应区块链上
  - 在代码和文档中将IPath简称为path
- **跨链分区**
  - 多条链通过跨链路由相连，形成跨链分区
  - 跨链分区有唯一标识，即IPath中的第一项（`payment.stub3.resource-d`的`payment`）



1.3 更多资料

- WeCross白皮书
- WeCross官网

CHAPTER 2

程序版本

2.1 下载程序

2.1.1 下载WeCross

提供三种方式，根据网络环境选择合适的方式进行下载。

方式1：命令下载

```
bash <(curl -sL https://github.com/WeBankFinTech/WeCross/releases/download/
↪resources/download_wecross.sh)
```

方式2：命令下载（源码编译模式）

```
# 默认下载master分支
bash <(curl -sL https://github.com/WeBankFinTech/WeCross/releases/download/
↪resources/download_wecross.sh) -s

# 下载特定版本下的
bash <(curl -sL https://github.com/WeBankFinTech/WeCross/releases/download/
↪resources/download_wecross.sh) -s -t v1.0.0-rc4
```

方式3：手动下载

- 国内资源：点击下载，MD5
- github release（下载最新版本的 `WeCross.tar.gz`）

  手动下载后解压

  ```
  tar -zxvf WeCross.tar.gz
  ```

解压后，目录下包含`WeCross/`文件夹。

2.1.2 下载WeCross控制台

同样提供三种方式，根据网络环境选择合适的方式进行下载。

方式1：命令下载

```
bash <(curl -sL https://github.com/WeBankFinTech/WeCross/releases/download/
↪resources/download_console.sh)
```

方式2：命令下载（源码编译模式）

```
# 默认下载master分支
bash <(curl -sL https://github.com/WeBankFinTech/WeCross/releases/download/
↪resources/download_console.sh) -s

# 下载特定版本下的控制台
bash <(curl -sL https://github.com/WeBankFinTech/WeCross/releases/download/
↪resources/download_console.sh) -s -t v1.0.0-rc4
```

方式3：手动下载

- 国内资源：点击下载，MD5
- github release（下载最新版本的 WeCross-Console.tar.gz）

  手动下载解压

  ```
  tar -zxvf WeCross-Console.tar.gz
  ```

下载后，目录下包含WeCross-Console/文件夹。

2.1.3 下载WeCross Demo

同样提供两种方式，根据网络环境选择合适的方式进行下载。

方式1：命令下载

```
# 下载最新demo
bash <(curl -sL https://github.com/WeBankFinTech/WeCross/releases/download/
↪resources/download_demo.sh)

# 下载特定版本下的demo
bash <(curl -sL https://github.com/WeBankFinTech/WeCross/releases/download/
↪resources/download_demo.sh) -t v1.0.0-rc4
```

方式2：手动下载

- 国内资源：点击下载，MD5
- github release（下载最新release下的demo.tar.gz）

  手动下载解压

  ```
  tar -zxvf demo.tar.gz
  ```

下载后，目录下包含demo/文件夹。

2.2 v1.0.0-rc4

(2020-08-18)

功能

- 两阶段事务框架：基于框架进行开发，实现多条异构链间的原子操作
- 跨链资源动态管理：通过API动态部署、更新跨链资源，无需编辑配置文件

新增

- 两阶段事务框架：框架、示例、逻辑和API等
- 代理合约：支持在运行时通过API对跨链资源进行部署和更新
- 更多的Demo
  - 两阶段 Demo
  - 跨 FISCO BCOS 群组 Demo
  - 跨 FISCO BCOS 国密与非国密链 Demo

更新

- 区块头同步逻辑更新：去除区块头落盘、router重启拉取最新区块头
- HTLC更新：使用WeCross-Console替代ledger-tool来初始化资产

2.3 v1.0.0-rc3

(2020-06-16)

新增

- Driver新增异步API定义：asyncCall、asyncSendTransaction，采用异步的方式调用插件接口

更新

- P2P通信：Router间的通信更新为异步的方式
- RPC接口：将性能较差的spring boot tomcat替换成netty的http server
- HTLC：适配Driver的异步API，采用异步的方式进行调用

2.4 v1.0.0-rc2

(2020-05-12)

新增

- 账户管理：用户账户统一由Router管理
- HTLC事务：支持同/异构链之间基于HTLC合约完成跨链转账
- Stub插件化：FISCO BCOS Stub和Fabric Stub通过jar包方式引入
- 安全通讯：WeCross SDK和Router之间采用TLS协议通讯
- 跨链Demo: 支持快速搭建WeCross Demo，体验简单跨链调用

更新

- 跨链接口：跨链调用需要指定账户名
- 跨链合约：跨链合约的参数类型和返回值类型限定为字符串数组
- 配置文件：主配置新增TLS以及HTLC配置项，Stub配置移除账户配置项
- 使用脚本：部署脚本、配置脚本以及证书生成脚本适配新的配置项

2.5 v1.0.0-rc1

(2019-12-30)

功能

- 接入区块链

  - 适配 FISCO BCOS
  - 适配 Fabric
- 统一接口：跨链路由将各种区块链的操作接口进行抽象，向外暴露统一的调用API
- 路由请求：跨链路由可自动将调用请求路由至相应的区块链
- 交易验证：向FISCO BCOS的链发交易时，能验证交易上链后的Merkle证明

架构

- 跨链路由：对接不同区块链的服务，对区块链的调用接口进行统一的抽象，彼此互连，将操作请求路由至相应链
- 跨链SDK：Java语言的API，用统一的接口向不同的链发请求
- 控制台：方便的操作终端，方便进行查询和发送请求

工具

- 跨链分区搭建脚本
- 接入FISCO BCOS和Fabric的配置框架生成脚本

CHAPTER 3

快速入门

本章介绍WeCross所需的软硬件环境配置，以及为用户提供了快速入门WeCross的教程。

3.1 环境要求

3.1.1 硬件

WeCross负责管理多个Stub并与多条链通讯，同时作为Web Server提供RPC调用服务，为了能保证服务的稳定性，尽量使用推荐配置。

配置	最低配置	推荐配置
CPU	1.5GHz	2.4GHz
内存	4GB	8GB
核心	4核	8核
带宽	2Mb	10Mb

3.1.2 支持的平台

- Ubuntu 16.04及以上
- CentOS 7.2及以上
- MacOS 10.14及以上

3.1.3 软件依赖

WeCross作为Java项目，需要安装Java环境包括：

- JDK8及以上
- Gradle 5.0及以上

WeCross提供了多种脚本帮助用户快速体验，这些脚本依赖`openssl`, `curl`, `expect`，使用下面的指令安装。

```
# Ubuntu
sudo apt-get install -y openssl curl expect tree

# CentOS
sudo yum install -y openssl curl expect tree

# MacOS
brew install openssl curl expect tree
```

3.2 快速体验

我们提供跨链Demo帮助用户快速体验并理解WeCross的原理。

下载 Demo

执行命令下载Demo，若下载较慢，可选择更多下载方式。

```
cd ~
# 下载WeCross demo合集，生成demo目录，目录下包含各种类型的demo
bash <(curl -sL https://github.com/WeBankFinTech/WeCross/releases/download/
↪resources/download_demo.sh)
```

注解:

- MacOS用户若出现“无法打开”，“无法验证开发者”的情况，可参考 FAQ问题3 的方式解决

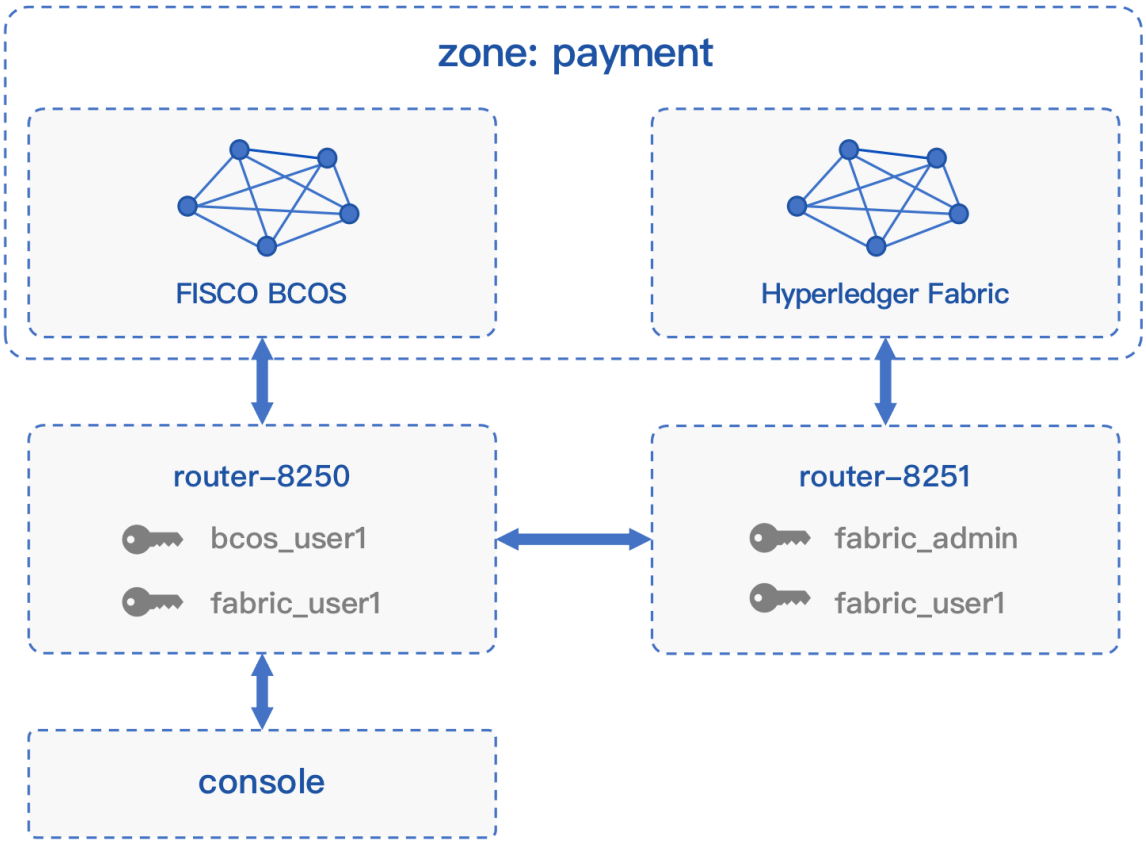
Demo 场景

1. Demo：跨平台 FISCO BCOS & Fabric
   - 跨链资源操作
   - 跨链转账（HTLC）
   - 跨链存证（2PC）
2. Demo：跨多群组
3. Demo：跨国密、非国密

实际情况下，WeCross的场景不仅限于两条链，用户可配置接入多条各种类型的区块链。

3.2.1 跨平台 FISCO BCOS & Fabric

此Demo搭建了一个WeCross跨链网络，连接FISCO BCOS和Hyperledger Fabric区块链。用户可通过WeCross控制台，对不同的链上资源进行操作。



网络部署

在已下载的demo目录下进行操作

```
cd ~/demo

#清理旧demo环境
bash clear.sh

# 运行部署脚本，第一次运行需耗时10-30分钟左右
bash build.sh # 若出错，可用 bash clear.sh 清理后重试
```

注解:

- MacOS用户若出现“无法打开”，“无法验证开发者”的情况，可参考 FAQ问题3 的方式解决

部署成功后会输出Demo的网络架构，FISCO BCOS和Fabric通过各自的WeCross Router相连。（输入Y，回车，进入WeCross控制台）

```
[INFO] Success! WeCross demo network is running. Framework:

      FISCO BCOS                  Fabric
     (4node pbft)             (first-network)
    (HelloWorld.sol)             (sacc.go)
           |                         |
           |                         |
    WeCross Router <----------> WeCross Router
(127.0.0.1-8250-25500)      (127.0.0.1-8251-25501)
```

(continues on next page)

(续上页)

```
            |
            |
    WeCross Console

Start WeCross Console? [Y/n]
```

跨链资源操作

查看资源

进入控制台，用`listResources`命令查看WeCross跨连网络中的所有资源。可看到有两个资源:

- payment.bcos.HelloWorld
  - 对应于FISCO BCOS链上的HelloWorld.sol合约
- payment.fabric.sacc
  - 对应于Fabric链上的sacc.go合约

```
[WeCross]> listResources
path: payment.bcos.HelloWorld, type: BCOS2.0, distance: 0
path: payment.fabric.sacc, type: Fabric1.4, distance: 1
total: 2
```

查看账户

用`listAccounts`命令查看WeCross Router上已存在的账户，操作资源时用相应账户进行操作。

```
[WeCross]> listAccounts
name: fabric_user1, type: Fabric1.4
name: bcos_user1, type: BCOS2.0
name: bcos_default_account, type: BCOS2.0
name: fabric_default_account, type: Fabric1.4
total: 4
```

操作资源：payment.bcos.HelloWorld

- 读资源
  - 命令：`call path 账户名 接口名 [参数列表]`
  - 示例：`call payment.bcos.HelloWorld bcos_user1 get`

```
# 调用HelloWorld合约中的get接口
[WeCross]> call payment.bcos.HelloWorld bcos_user1 get
Result: [Hello, World!]
```

- 写资源
  - 命令：`sendTransaction path 账户名 接口名 [参数列表]`
  - 示例：`sendTransaction payment.bcos.HelloWeCross bcos_user1 set Tom`

```
# 调用HelloWeCross合约中的set接口
[WeCross]> sendTransaction payment.bcos.HelloWorld bcos_user1 set Tom
Txhash  : 0x7e747198f553cb2e90e729b52179533dc4321e520b0f11b83b1f0e81fa7ff716
BlockNum: 5
Result  : []      // 将Tom给set进去

[WeCross]> call payment.bcos.HelloWorld bcos_user1 get
Result: [Tom]     // 再次get，Tom已set
```

操作资源：payment.fabric.sacc

跨链资源是对各个不同链上资源的统一和抽象，因此操作的命令是保持一致的。

- 读资源

```
# 调用mycc合约中的query接口
[WeCross]> call payment.fabric.sacc fabric_user1 get a
Result: [10] // 初次get，a的值为10
```

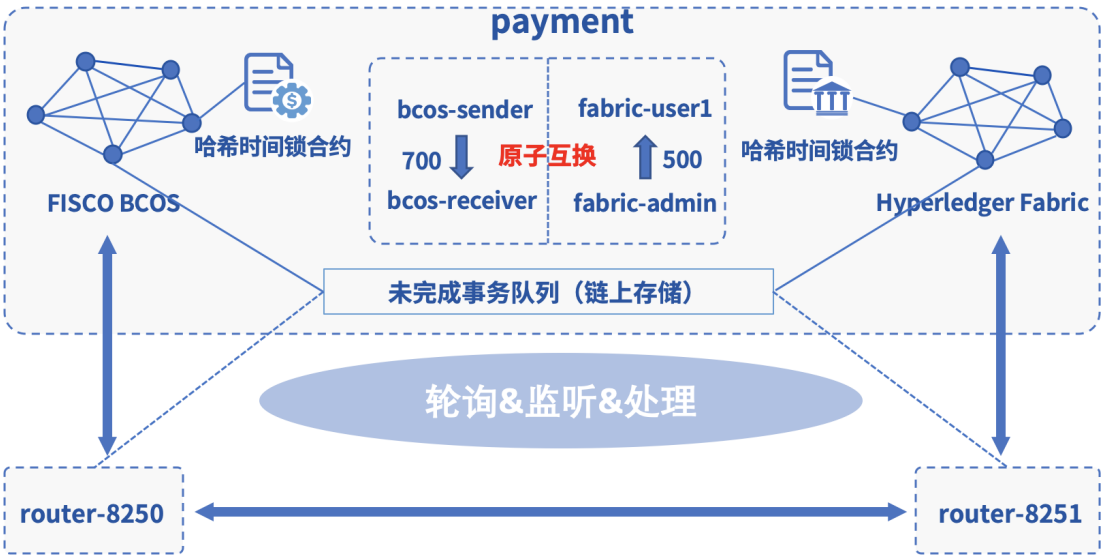
- 写资源

```
# 调用sacc合约中的set接口
[WeCross]> sendTransaction payment.fabric.sacc fabric_user1 set a 666
Txhash  : eca4ecacf7b159c1499d6c190fcaf9fd7348bdb96cdbf35cd29b34ac9bd8e518
BlockNum: 7
Result  : [666]

[WeCross]> call payment.fabric.sacc fabric_user1 get a
Result: [666] // 再次get，a的值变成666

# 退出WeCross控制台
[WeCross]> quit # 若想再次启动控制台，cd至WeCross-Console，执行start.sh即可
```

WeCross Console是基于WeCross Java SDK开发的跨链应用。搭建好跨链网络后，可基于WeCross Java SDK开发更多的跨链应用，通过统一的接口对各种链上的资源进行操作。

跨链转账

WeCross支持多种事务机制。此跨链转账的demo是哈希时间锁定机制（HTLC）的举例。WeCross基于其HTLC框架实现了**异构链之间资产的原子互换**，如下图所示：



部署哈希时间锁合约

可通过脚本`htlc_config.sh`完成相关部署，并体验跨链转账。

```
# 请确保demo已搭建完毕，并在demo根目录执行，耗时5分钟左右
bash htlc_config.sh
```

跨链转账涉及两条链、两个用户、四个账户，两条链上的资产转出者各自通过WeCross控制台创建一个转账提案，之后router会自动完成跨链转账。

创建转账提案

跨链转账需在跨链的两端都提交转账提案，提交后，router自动实现跨链转账。

- BCOS链资产转出者提交提案

BCOS链连的是router-8250，启动与该router连接的控制台。

```
cd ~/demo/WeCross-Console
bash start.sh
# 先查看接收方余额
[WeCross]> call payment.bcos.htlc bcos_sender balanceOf␣
↪0x2b5ad5c4795c026514f8317c7a215e218dccd6cf
Result: [0]
# 创建转账提案
[WeCross]> newHTLCProposal payment.bcos.htlc bcos_sender␣
↪bea2dfec011d830a86d0fbeeb383e622b576bb2c15287b1a86aacdba0a387e11␣
↪9dda9a5e175a919ee98ff0198927b0a765ef96cf917144b589bb8e510e04843c true␣
↪0x55f934bcbe1e9aef8337f5551142a442fdde781c␣
↪0x2b5ad5c4795c026514f8317c7a215e218dccd6cf 700 2000010000 Admin@org1.example.com␣
↪User1@org1.example.com 500 2000000000
# 输出
Txhash: a0c48eb7d1ca3a01ddf3563aeb6a1829f23dd0d778e7de2ce22406d1e84ba00f
BlockNum: 12
Result: create a htlc proposal successfully
# 退出当前控制台
[WeCross]> quit
```

- Fabric链资产转出者提交提案

Fabric链连的是router-8251，启动与该router连接的控制台。

```
cd ~/demo/WeCross-Console-8251
bash start.sh
# 先查看接收方余额
[WeCross]>  call payment.fabric.htlc fabric_admin balanceOf User1@org1.example.com
Result: [0]
# 创建转账提案
[WeCross]> newHTLCProposal payment.fabric.htlc fabric_admin␣
↪bea2dfec011d830a86d0fbeeb383e622b576bb2c15287b1a86aacdba0a387e11 null false␣
↪0x55f934bcbe1e9aef8337f5551142a442fdde781c␣
↪0x2b5ad5c4795c026514f8317c7a215e218dccd6cf 700 2000010000 Admin@org1.example.com␣
↪User1@org1.example.com 500 2000000000
# 输出
Txhash: 0x40ae8e2e284de813f8b071e0261e627ddc4d91e365e63f222638db9b1a70d05a
BlockNum: 10
Result: create a htlc proposal successfully
# 退出当前控制台
[WeCross]> quit
```

跨链资产转移

当两个资产转出者都创建完提案后，router开始执行调度，并完成跨链转账。一次跨链转账存在5-25s的交易时延，主要取决于两条链以及机器的性能。

查询转账结果

在各自的WeCross控制台查询资产是否到账。

- 查询BCOS链上资产接收者余额

```
cd ~/demo/WeCross-Console
bash start.sh

[WeCross]> call payment.bcos.htlc bcos_sender balanceOf␣
↪0x2b5ad5c4795c026514f8317c7a215e218dccd6cf
```

(continues on next page)

(续上页)

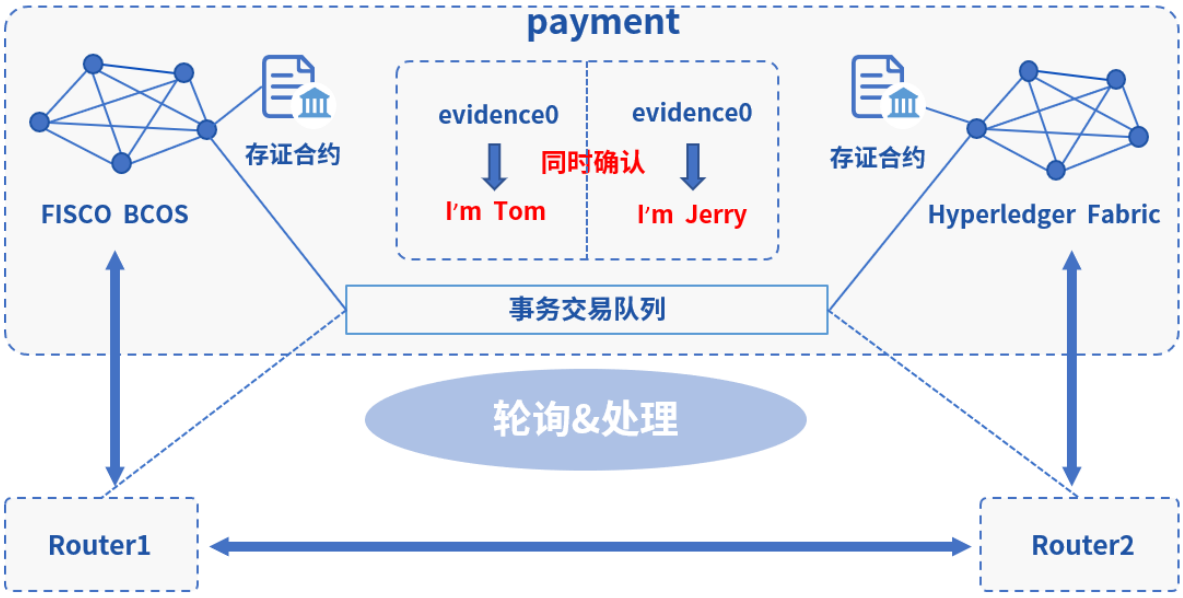
```
Result: [700]

# 退出当前控制台
[WeCross]> quit
```

- 查询Fabric链上资产接收者余额

```
cd ~/demo/WeCross-Console-8251
bash start.sh

[WeCross]>  call payment.fabric.htlc fabric_admin balanceOf User1@org1.example.com
Result: [500]

# 退出当前控制台
[WeCross]> quit
```

跨链存证

WeCross支持多种事务机制。此跨链转账的demo是两阶段事务机制（2PC）的举例。WeCross基于其2PC框架实现了**异构链之间证据的同时确认**，如下图所示：



部署跨链存证demo的合约

用一键脚本，在FISCO BCOS和Fabric上分别部署存证跨链的两个存证demo合约

```
cd ~/demo/
bash 2pc_config.sh
```

部署成功，输入Y进入控制台

```
[INFO] SUCCESS: 2PC evidence example has been deployed to FISCO BCOS and Fabric:

      FISCO BCOS                      Fabric
(payment.bcos.evidence)     (payment.fabric.evidence)
           |                             |
           |                             |
    WeCross Router <----------> WeCross Router
(127.0.0.1-8250-25500)       (127.0.0.1-8251-25501)
```

(continues on next page)

(续上页)

```
            |
            |
    WeCross Console

Start WeCross Console to try? [Y/n]
```

发起跨链事务（start）

发起一个跨链事务，指定事务涉及的跨链资源，此处FISCO BCOS和Fabric上各一个资源

- payment.bcos.evidence
- payment.fabric.evidence

```
# 发起事务，事务号100
[WeCross]> startTransaction 100 bcos_user1 fabric_user1 payment.bcos.evidence␣
↪payment.fabric.evidence
Result: success!

# 查看当前存证evidence0的内容
[WeCross]> call payment.bcos.evidence bcos_user1 queryEvidence evidence0
Result: [] # 未存入，空

[WeCross]> call payment.fabric.evidence fabric_user1 queryEvidence evidence0
Result: [] # 未存入，空
```

发送事务交易（exec）

事务开始后，通过execTransaction发送事务交易至向此事务涉及的资源，交易会被缓存入事务交易队列，在下一步commit时让所有交易同时被确认。

```
# 在FISCO BCOS链上进行存证，事务号100，序列号1，证据名：evidence0，内容：I'm Tom
[WeCross]> execTransaction payment.bcos.evidence bcos_user1 100 1 newEvidence␣
↪evidence0 "I'm Tom"
Result: [true]

# 在Fabric链上进行存证，事务号100，序列号1，证据名：evidence0，内容：I'm Jerry
[WeCross]> execTransaction payment.fabric.evidence fabric_user1 100 1 newEvidence␣
↪evidence0 "I'm Jerry"
Result: [newEvidence success]

# 可发送更多的操作
```

确认跨链事务（commit）

在此事务下缓存了一些列的事务交易后，通过commitTransaction让所有事务交易同时被确认，结束此事务。

```
# 确认事务，事务结束
[WeCross]> commitTransaction 100 bcos_user1 fabric_user1 payment.bcos.evidence␣
↪payment.fabric.evidence
Result: success!

# 查看当前存证内容，两条链都已完成存证
[WeCross]> call payment.bcos.evidence bcos_user1 queryEvidence evidence0
Result: [I'm Tom]

[WeCross]> call payment.fabric.evidence fabric_user1 queryEvidence evidence0
Result: [I'm Jerry]
```

回滚跨链事务（rollback）

在commit前，若不想让事务发生，则用rollbackTransaction回滚此事务，所有缓存的事务交易被丢弃，链上数据回退到事务发起前状态，事务结束。（每个事务要么commit，要么rollback）

```
# 查看当前存证evidence1的内容
[WeCross]> call payment.bcos.evidence bcos_user1 queryEvidence evidence1
Result: [] # 未存入，空

[WeCross]> call payment.fabric.evidence fabric_user1 queryEvidence evidence1
Result: [] # 未存入，空

# 发起另一个事务，事务号101
[WeCross]> startTransaction 101 bcos_user1 fabric_user1 payment.bcos.evidence␣
↪payment.fabric.evidence
Result: success!

# 向FISCO BCOS链发送事务交易，设置evidence1，内容为I'm TomGG
[WeCross]> execTransaction payment.bcos.evidence bcos_user1 101 1 newEvidence␣
↪evidence1 "I'm TomGG"
Result: [true]

# 向Fabric链发送事务交易，设置evidence1，内容为I'm JerryMM
[WeCross]> execTransaction payment.fabric.evidence fabric_user1 101 1 newEvidence␣
↪evidence1 "I'm JerryMM"
Result: [newEvidence success]

# 查看当前事务状态下的数据
[WeCross]> call payment.bcos.evidence bcos_user1 queryEvidence evidence1
Result: [I'm TomGG]

# 查看当前事务状态下的数据
[WeCross]> call payment.fabric.evidence fabric_user1 queryEvidence evidence1
Result: [I'm JerryMM]

# 尝试发送普通交易修改此事务下的资源，由于此资源处在事务状态中，被锁定，不可修改
[WeCross]> sendTransaction payment.bcos.evidence bcos_user1 newEvidence evidence1
↪"I'm TomDD"
Error: code(2031), message(payment.bcos.evidence is locked by unfinished␣
↪transaction: 101)

# 查看当前事务状态下的数据，未变化，普通交易修改失败，符合预期
[WeCross]> call payment.bcos.evidence bcos_user1 queryEvidence evidence1
Result: [I'm TomGG]

# 回滚操作！假设此事务不符合预期，需回滚至事务开始前状态，执行如下命令进行回滚，事务结束
[WeCross]> rollbackTransaction 101 bcos_user1 fabric_user1 payment.bcos.evidence␣
↪payment.fabric.evidence
Result: success!

# 再次查看当前存证evidence1的内容
[WeCross]> call payment.bcos.evidence bcos_user1 queryEvidence evidence1
Result: [] # 已回滚至开始状态

[WeCross]> call payment.fabric.evidence fabric_user1 queryEvidence evidence1
Result: [] # 已回滚至开始状态

# 退出当前控制台
[WeCross]> quit
```

此demo基于2PC框架实现，用户可根据业务需要基于框架开发自己的跨链应用，实现链间的原子操作。

清理 Demo

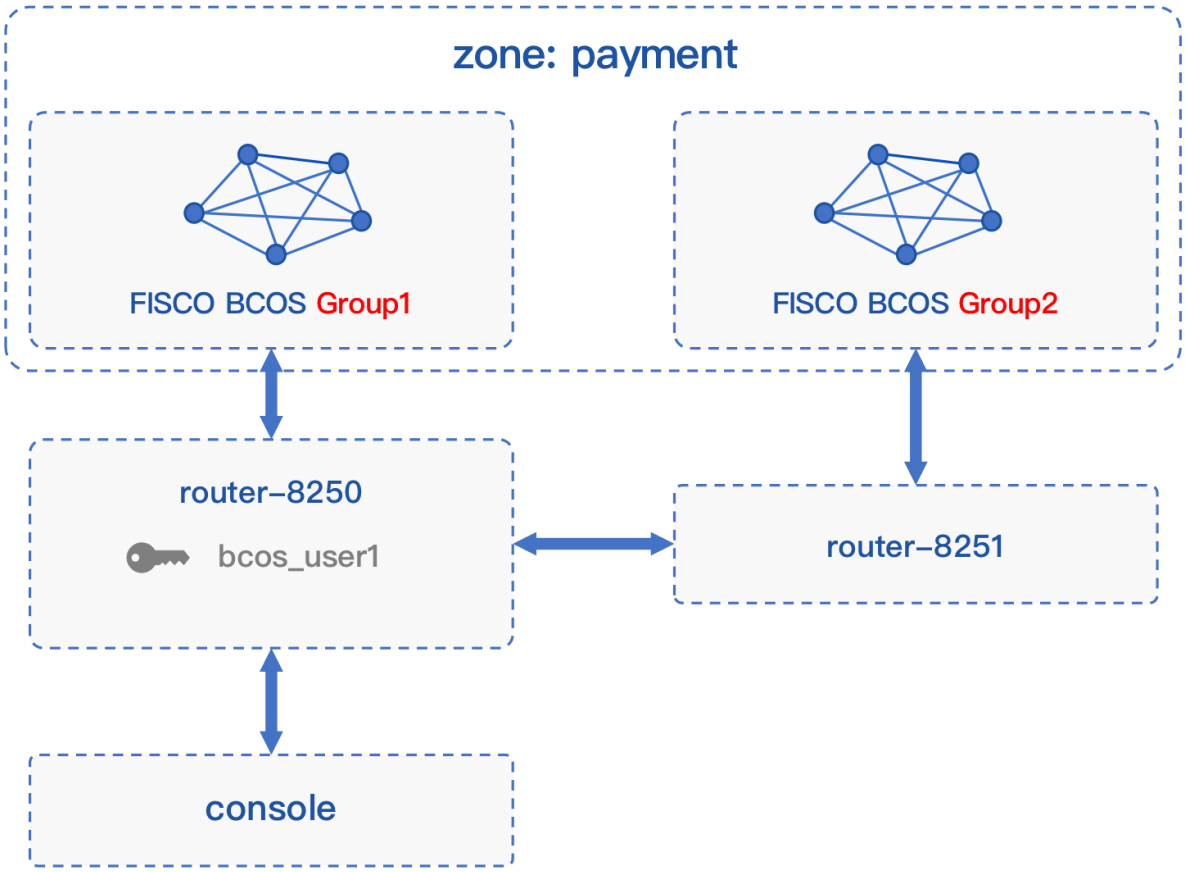
为了不影响其它章节的体验，可将搭建的Demo清理掉。

```
cd ~/demo/
bash clear.sh
```

至此，恭喜你，快速体验完成！可进入手动组网章节深入了解更多细节。

3.2.2 跨多群组

此Demo搭建了一个WeCross跨链网络，连接FISCO BCOS中的两个群组。用户可通过WeCross控制台，对不同群组的资源进行操作。实际情况下，WeCross接入链的场景不受限制，用户可配置接入多条各种类型的区块链。



网络部署

在已下载的demo目录下进行操作

```
cd ~/demo

#清理旧demo环境
bash clear.sh

# 运行部署脚本，第一次运行需耗时10-30分钟左右
bash build_cross_groups.sh # 若出错，可用 bash clear.sh 清理后重试
```

注解:

- MacOS用户若出现“无法打开”，“无法验证开发者”的情况，可参考 FAQ问题3 的方式解决

部署成功后会输出Demo的网络架构，FISCO BCOS和Fabric通过各自的WeCross Router相连。（输入Y，回车，进入WeCross控制台）

```
[INFO] Success! WeCross demo network is running. Framework:

                    FISCO BCOS
        Group 1                     Group 2
   (HelloWorldGroup1)          (HelloWorldGroup2)
           |                           |
           |                           |
    WeCross Router <----------> WeCross Router
(127.0.0.1-8250-25500)      (127.0.0.1-8251-25501)
           |
           |
    WeCross Console

Start console? [Y/n]
```

操作跨链资源

查看资源

进入控制台，用`listResources`命令查看WeCross跨连网络中的所有资源。可看到有两个资源:

- payment.group1.HelloWorldGroup1
  - 对应于Group1上的HelloWorld.sol合约
- payment.group2.HelloWorldGroup2
  - 对应于Group2上的HelloWorld.sol合约

```
[WeCross]> listResources
path: payment.group2.HelloWorldGroup2, type: BCOS2.0, distance: 1
path: payment.group1.HelloWorldGroup1, type: BCOS2.0, distance: 0
total: 2
```

查看账户

用`listAccounts`命令查看WeCross Router上已存在的账户，操作资源时用相应账户进行操作。

```
[WeCross]> listAccounts
name: bcos_user1, type: BCOS2.0
total: 1
```

操作资源：payment.group1.HelloWorldGroup1

- 读资源
  - 命令：`call path 账户名 接口名 [参数列表]`
  - 示例：`call payment.group1.HelloWorldGroup1 bcos_user1 get`

```
# 调用Group1 HelloWorld合约中的get接口
[WeCross]> call payment.group1.HelloWorldGroup1 bcos_user1 get
Result: [Hello, World!] // 初次get，值为Hello World!
```

- 写资源
  - 命令：`sendTransaction path 账户名 接口名 [参数列表]`
  - 示例：`sendTransaction payment.group1.HelloWorldGroup1 bcos_user1 set Tom`

```
# 调用Group1 HelloWeCross合约中的set接口
[WeCross]> sendTransaction payment.group1.HelloWorldGroup1 bcos_user1 set Tom
Txhash  : 0xae1f74b6320883600ef4c6885e0a9f9c06f367b6129cbda202cbe61cecac0ed0
BlockNum: 5
Result  : []      // 将Tom给set进去

[WeCross]> call payment.group1.HelloWorldGroup1 bcos_user1 get
Result: [Tom]     // 再次get，Tom已set
```

操作资源：payment.group2.HelloWorldGroup2

跨链资源是对各个不同链上资源的统一和抽象，因此操作的命令是保持一致的。

- 读资源

```
# 调用Group2 HelloWorld合约中的get接口
[WeCross]> call payment.group2.HelloWorldGroup2 bcos_user1 get
Result: [Hello, World!] // 初次get，值为Hello World!
```

- 写资源

```
# 调用Group2 HelloWeCross合约中的set接口
[WeCross]> sendTransaction payment.group2.HelloWorldGroup2 bcos_user1 set Jerry
Txhash  : 0xde7bf37a61478788727f9c5caccca96a478055fb440eee588d76e22cbc232bc5
BlockNum: 5
Result  : []      // 将Jerry给set进去

[WeCross]> call payment.group2.HelloWorldGroup2 bcos_user1 get
Result: [Jerry]     // 再次get，Jerry已set

# 检查Group1资源，不会因为Group2的资源被修改而改变
[WeCross]> call payment.group1.HelloWorldGroup1 bcos_user1 get
Result: [Tom]

# 退出WeCross控制台
[WeCross]> quit # 若想再次启动控制台，cd至WeCross-Console，执行start.sh即可
```

WeCross Console是基于WeCross Java SDK开发的跨链应用。搭建好跨链网络后，可基于WeCross Java SDK开发更多的跨链应用，通过统一的接口对各种链上的资源进行操作。

清理 Demo

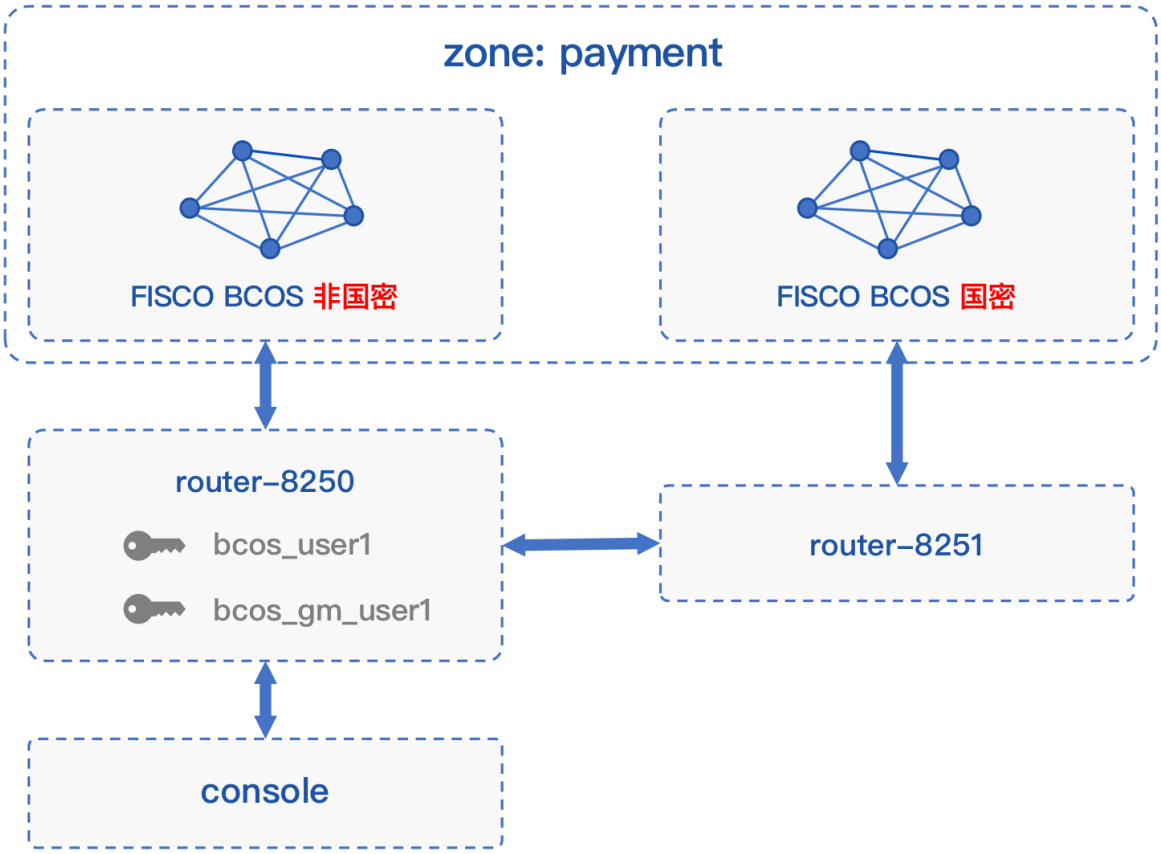
为了不影响其它章节的体验，可将搭建的Demo清理掉。

```
cd ~/demo/
bash clear.sh
```

至此，恭喜你，快速体验完成！可进入手动组网章节深入了解更多细节。

3.2.3 跨国密、非国密

此Demo搭建了一个WeCross跨链网络，连接FISCO BCOS的国密区块链和非国密区块链。用户可通过WeCross控制台，对不同链的链上资源进行操作。实际情况下，WeCross接入链的场景不受限制，用户可配置接入多条各种类型的区块链。



网络部署

在已下载的demo目录下进行操作

```
cd ~/demo

#清理旧demo环境
bash clear.sh

# 运行部署脚本，第一次运行需耗时10-30分钟左右
bash build_cross_gm.sh # 若出错，可用 bash clear.sh 清理后重试
```

注解:

- MacOS用户若出现“无法打开”，“无法验证开发者”的情况，可参考 FAQ问题3 的方式解决

部署成功后会输出Demo的网络架构，FISCO BCOS和Fabric通过各自的WeCross Router相连。（输入Y，回车，进入WeCross控制台）

```
[INFO] Success! WeCross demo network is running. Framework:

                    FISCO BCOS
        Normal                          Guomi
     (HelloWorld)                    (HelloWorld)
          |                               |
          |                               |
    WeCross Router <-----------> WeCross Router
(127.0.0.1-8250-25500)       (127.0.0.1-8251-25501)
          |
```

(continues on next page)

(续上页)

```
          |
    WeCross Console

Start console? [Y/n]
```

操作跨链资源

查看资源

进入控制台，用`listResources`命令查看WeCross跨连网络中的所有资源。可看到有两个资源:

- payment.bcos.HelloWorld
  - 对应于**非国密FISCO BCOS链**上的HelloWorld.sol合约
- payment.bcos_gm.HelloWorld
  - 对应于**国密FISCO BCOS链**上的HelloWorld.sol合约

```
[WeCross]> listResources
path: payment.bcos.HelloWorld, type: BCOS2.0, distance: 0
path: payment.bcos_gm.HelloWorld, type: GM_BCOS2.0, distance: 1
total: 2
```

查看账户

用`listAccounts`命令查看WeCross Router上已存在的账户，操作资源时用相应账户进行操作。

```
[WeCross]> listAccounts
name: bcos_user1, type: BCOS2.0
name: bcos_gm_user1, type: GM_BCOS2.0
total: 2
```

操作资源：payment.bcos.HelloWorld

- 读资源
  - 命令：`call path 账户名 接口名 [参数列表]`
  - 示例：`call payment.bcos.HelloWorld bcos_user1 get`

```
# 调用非国密链上HelloWorld合约中的get接口
[WeCross]> call payment.bcos.HelloWorld bcos_user1 get
Result: [Hello, World!] // 初次get，值为Hello World!
```

- 写资源
  - 命令：`sendTransaction path 账户名 接口名 [参数列表]`
  - 示例：`sendTransaction payment.bcos.HelloWorld bcos_user1 set Tom`

```
# 调用非国密链上HelloWeCross合约中的set接口
[WeCross]> sendTransaction payment.bcos.HelloWorld bcos_user1 set Tom
Txhash  : 0x5a70d7874c2f0e4a9eddf160db6d2a79b923afeb9fa95bdea368391079176b6b
BlockNum: 5
Result  : []      // 将Tom给set进去

[WeCross]> call payment.bcos.HelloWorld bcos_user1 get
Result: [Tom]     // 再次get，Tom已set
```

操作资源：payment.bcos_gm.HelloWorld

跨链资源是对各个不同链上资源的统一和抽象，因此操作的命令是保持一致的。

- 读资源

```
# 调用国密链上HelloWorld合约中的get接口
[WeCross]> call payment.bcos_gm.HelloWorld bcos_gm_user1 get
Result: [Hello, World!] // 初次get，值为Hello World!
```

- 写资源

```
# 调用国密链上HelloWeCross合约中的set接口
[WeCross]> sendTransaction payment.bcos_gm.HelloWorld bcos_gm_user1 set Jerry
Txhash  : 0xe74f237d6ad30e30755bb007bf543b6909238c65d72d4d5b62e29db6cd484aec
BlockNum: 5
Result  : []      // 将Jerry给set进去

[WeCross]> call payment.bcos_gm.HelloWorld bcos_gm_user1 get
Result: [Jerry]     // 再次get，Jerry已set

# 检查非国密链上的资源，不会因为国密链上的资源被修改而改变
[WeCross]> call payment.bcos.HelloWorld bcos_user1 get
Result: [Tom]

# 退出WeCross控制台
[WeCross]> quit # 若想再次启动控制台，cd至WeCross-Console，执行start.sh即可
```

WeCross Console是基于WeCross Java SDK开发的跨链应用。搭建好跨链网络后，可基于WeCross Java SDK开发更多的跨链应用，通过统一的接口对各种链上的资源进行操作。

清理 Demo

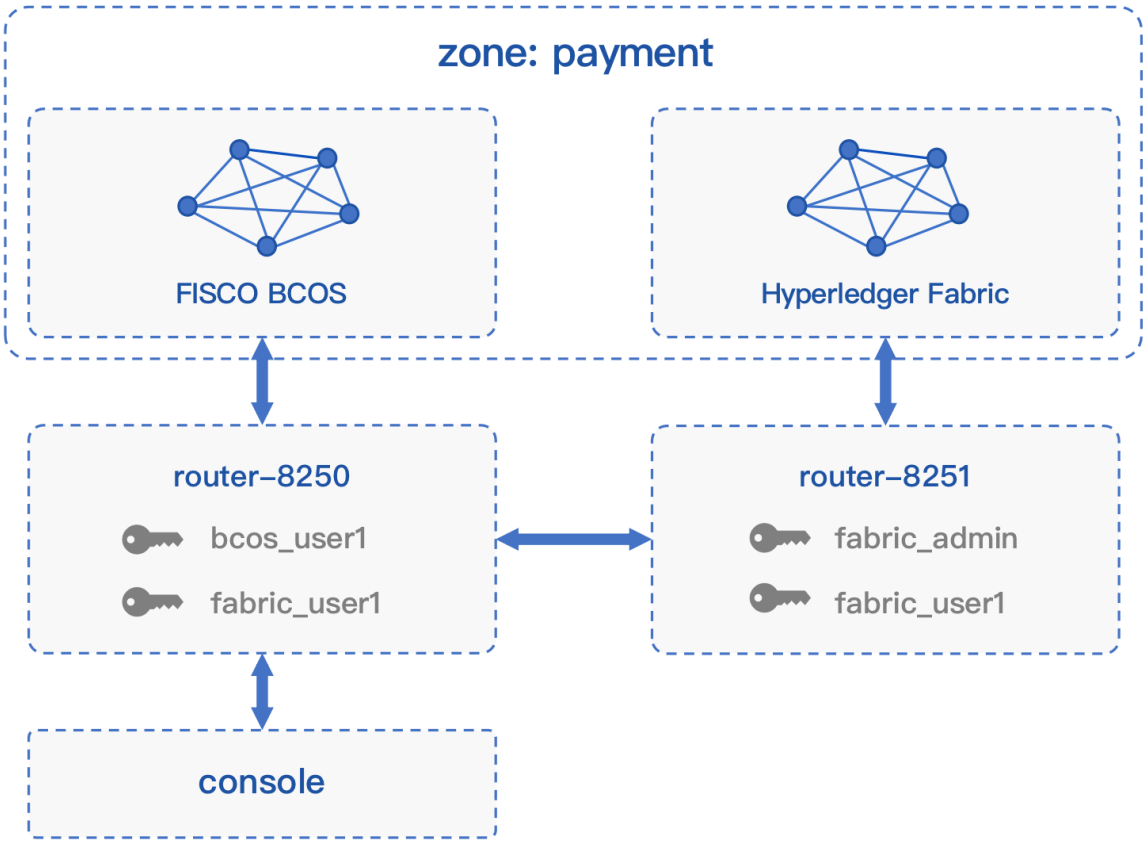
为了不影响其它章节的体验，可将搭建的Demo清理掉。

```
cd ~/demo/
bash clear.sh
```

至此，恭喜你，快速体验完成！可进入手动组网章节深入了解更多细节。

3.3 手动组网

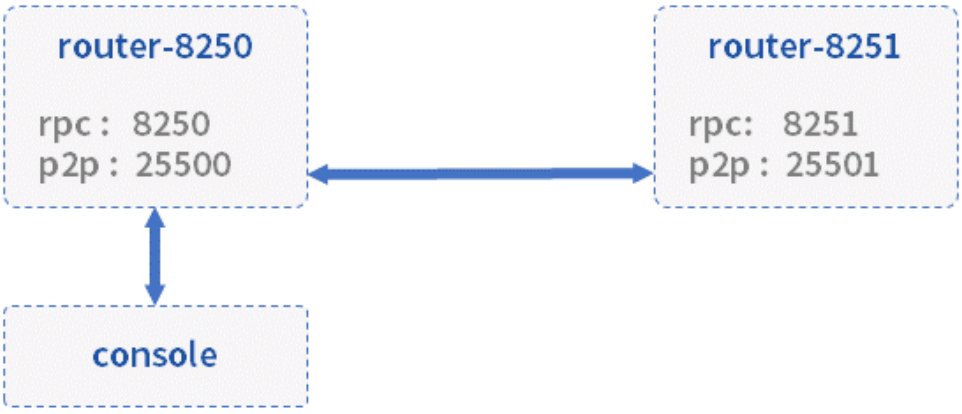
本文档介绍以手动的方式，一步步搭建一个与Demo相同的跨链网络。



3.3.1 快速部署

本章指导完成跨链路由的部署。

- **跨链路由**：与区块链节点对接，并彼此互连，形成跨链分区，负责跨链请求的转发
- **跨链控制台**：查询和发送交易的操作终端



操作以~/wecross/目录下为例进行。若Demo未清理，请先清理Demo环境。

```
mkdir -p ~/wecross/ && cd ~/wecross/
```

部署 WeCross Router

下载WeCross，用WeCross中的工具生成跨链路由，并启动跨链路由。

下载WeCross

WeCross中包含了生成跨链路由的工具，执行以下命令进行下载（提供三种下载方式，可根据网络环境选择合适的方式进行下载），程序下载至当前目录WeCross/中。

```
bash <(curl -sL https://github.com/WeBankFinTech/WeCross/releases/download/
↪resources/download_wecross.sh)
```

生成跨链路由

本例将构建两个跨链路由。首先创建一个ipfile配置文件，将需要构建的两个跨链路由信息（ip:rpc_port:p2p_port）按行分隔，保存到文件中。

注：请确保机器的8250，8251, 25500，25501端口没有被占用。

```
cd ~/wecross
vim ipfile

# 在文件中键入以下内容
127.0.0.1:8250:25500
127.0.0.1:8251:25501
```

生成好ipfile文件后，使用脚本build_wecross.sh生成两个跨链路由。

```
# -f 表示以文件为输入
bash ./WeCross/build_wecross.sh -n payment -o routers-payment -f ipfile

# 成功输出如下信息
[INFO] Create routers-payment/127.0.0.1-8250-25500 successfully
[INFO] Create routers-payment/127.0.0.1-8251-25501 successfully
[INFO] All completed. WeCross routers are generated in: routers-payment/
```

注解:

- -n 指定跨链分区标识符(zone id)，跨链分区通过zone id进行区分，可以理解为业务名称。
- -o 指定输出的目录，并在该目录下生成一个跨链路由。
- -f 指定需要生成的WeCross跨链路由的列表，包括ip地址，rpc端口，p2p端口，生成后的router已完成互联配置。

在routers-payment目录下生成了两个跨链路由。

```
tree routers-payment/ -L 1
routers-payment/
├── 127.0.0.1-8251-25501
├── 127.0.0.1-8252-25502
└── cert
```

生成的跨链路由目录内容如下，以127.0.0.1-8250-25500为例。

```
# 已屏蔽lib目录，该目录存放所有依赖的jar包
tree routers-payment/127.0.0.1-8250-25500/ -I "lib"
routers-payment/127.0.0.1-8250-25500/
```

(continues on next page)

(续上页)

```
├── add_account.sh     # 账户生成脚本
├── add_chain.sh       # 区块链配置文件创建脚本
├── apps
│   └── WeCross.jar    # WeCross路由jar包
├── build_wecross.sh
├── conf               # 配置文件目录
│   ├── accounts       # 账户配置目录
│   ├── application.properties
│   ├── chains         # 区块链配置目录，要接入不同的链，在此目录下进行配置
│   ├── log4j2.xml
│   ├── ca.crt         # 根证书
│   ├── ssl.crt        # 跨链路由证书
│   ├── ssl.key        # 跨链路由私钥
│   ├── node.nodeid    # 跨链路由nodeid
│   └── wecross.toml   # WeCross Router主配置文件
├── create_cert.sh     # 证书生成脚本
├── download_wecross.sh
├── plugin             # 插件目录，接入相应类型链的插件
│   ├── bcos-stub-gm.jar
│   ├── bcos-stub.jar
│   └── fabric-stub.jar
├── start.sh           # 启动脚本
└── stop.sh            # 停止脚本
```

启动跨链路由

```
# 启动 router-8250
cd ~/wecross/routers-payment/127.0.0.1-8250-25500/
bash start.sh    # 停止: bash stop.sh

# 启动 router-8251
cd ~/wecross/routers-payment/127.0.0.1-8251-25501/
bash start.sh    # 停止: bash stop.sh
```

启动成功，输出如下：

```
WeCross booting up .........
WeCross start successfully
```

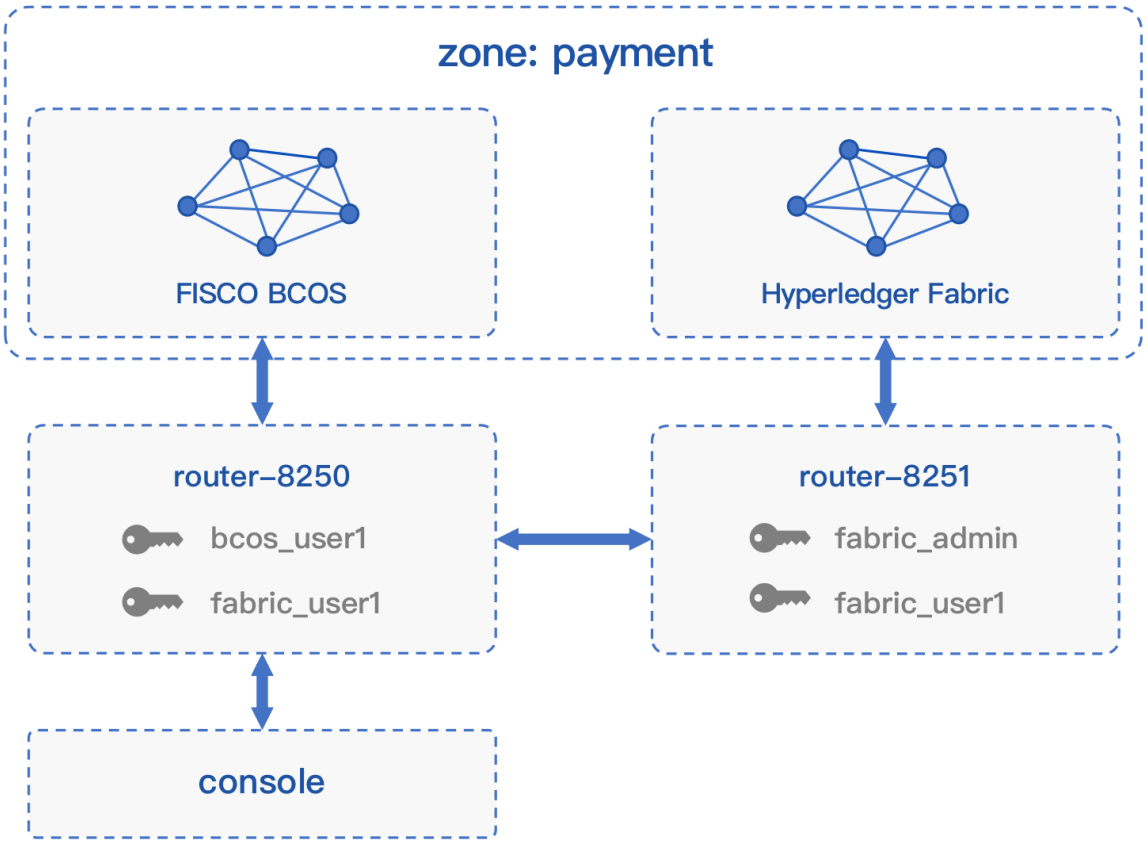
如果启动失败，检查8250, 25500端口是否被占用。

```
netstat -napl | grep 8250
netstat -napl | grep 25500
netstat -napl | grep 8251
netstat -napl | grep 25501
```

3.3.2 接入区块链

完成了WeCross的部署，如何让它和一条真实的区块链交互，相信优秀的您一定在跃跃欲试。本节包括

- 接入BCOS链：在router-8250上接入，配置交易发送账户
- 接入Fabric链：在router-8251上接入，配置交易发送账户



搭建区块链

在接入区块链前，先给出BCOS和Fabric链的搭建过程。

搭建BCOS链

FISCO BCOS官方提供了一键搭链的教程，详见单群组FISCO BCOS联盟链的搭建

详细步骤如下：

- 脚本建链

```
# 创建操作目录
mkdir -p ~/wecross/bcos && cd ~/wecross/bcos

# 下载build_chain.sh脚本
curl -LO https://github.com/FISCO-BCOS/FISCO-BCOS/releases/download/v2.6.0/build_
↪chain.sh && chmod u+x build_chain.sh

# 搭建单群组4节点联盟链
# 在fisco目录下执行下面的指令，生成一条单群组4节点的FISCO链。请确保机器的30300~30303，20200~
↪20203，8545~8548端口没有被占用。
# 命令执行成功会输出All completed。如果执行出错，请检查nodes/build.log文件中的错误信息。
bash build_chain.sh -l "127.0.0.1:4" -p 30300,20200,8545
```

- 启动所有节点

```
bash nodes/127.0.0.1/start_all.sh
```

启动成功会输出类似下面内容的响应。否则请使用netstat -an | grep tcp检查机器的30300~30303，20200~20203，8545~8548端口是否被占用。

```
try to start node0
try to start node1
try to start node2
try to start node3
node1 start successfully
node2 start successfully
node0 start successfully
node3 start successfully
```

搭建Fabric链

为方便Fabric链的搭建，WeCross Demo包中提供了Fabric链搭建脚本。若下载较慢，可选择更多下载方式。

```
mkdir -p ~/wecross/fabric && cd ~/wecross/fabric

# 下载Demo包，拷贝其中的Fabric demo链环境
bash <(curl -sL https://github.com/WeBankFinTech/WeCross/releases/download/
↪resources/download_demo.sh)
cp demo/fabric/* ./

# 搭建
bash build.sh # 若出错，执行 bash clear.sh 后重新 bash build.sh
```

搭建成功，查看Fabric链各个容器运行状态。

```
docker ps
```

可看到各个容器的状态：

```
CONTAINER ID        IMAGE
↪                                                  COMMAND                  CREATED
↪     STATUS              PORTS                               NAMES
3b55f9681227        dev-peer1.org2.example.com-mycc-1.0-
↪26c2ef32838554aac4f7ad6f100aca865e87959c9a126e86d764c8d01f8346ab   "chaincode -
↪peer.add..."   13 minutes ago      Up 13 minutes
↪ dev-peer1.org2.example.com-mycc-1.0
2d8d660c9481        dev-peer0.org1.example.com-mycc-1.0-
↪384f11f484b9302df90b453200cfb25174305fce8f53f4e94d45ee3b6cab0ce9   "chaincode -
↪peer.add..."   13 minutes ago      Up 13 minutes
↪ dev-peer0.org1.example.com-mycc-1.0
b82b0b8dcc0f        dev-peer0.org2.example.com-mycc-1.0-
↪15b571b3ce849066b7ec74497da3b27e54e0df1345daff3951b94245ce09c42b   "chaincode -
↪peer.add..."   14 minutes ago      Up 14 minutes
↪ dev-peer0.org2.example.com-mycc-1.0
441ca8a493fc        hyperledger/fabric-tools:latest
↪                                   "/bin/bash"              14 minutes ago
↪     Up 14 minutes                                 cli
de0d32730926        hyperledger/fabric-peer:latest
↪                                   "peer node start"        14 minutes ago
↪     Up 14 minutes       0.0.0.0:9051->9051/tcp    peer0.org2.example.com
ad98565bfa57        hyperledger/fabric-peer:latest
↪                                   "peer node start"        14 minutes ago
↪     Up 14 minutes       0.0.0.0:10051->10051/tcp  peer1.org2.example.com
bf0d9b0c54bf        hyperledger/fabric-peer:latest
↪                                   "peer node start"        14 minutes ago
↪     Up 14 minutes       0.0.0.0:8051->8051/tcp    peer1.org1.example.com
b4118a65f01a        hyperledger/fabric-orderer:latest
↪                                   "orderer"                14 minutes ago
↪     Up 14 minutes       0.0.0.0:7050->7050/tcp    orderer.example.com
```

(continues on next page)

(续上页)

```
fcf1bfe17dbe        hyperledger/fabric-peer:latest                                  ␣
↪                                   "peer node start"         14 minutes ago  ␣
↪     Up 14 minutes       0.0.0.0:7051->7051/tcp      peer0.org1.example.com
```

接入FISCO BCOS链

添加账户

在router中添加用于向链上发交易的账户。账户配置好后，可通过跨链网络向相应的链发交易，交易可被router转发至对应的链上。

添加BCOS账户

所配置的账户可用于向BCOS2.0类型的链发交易。

```
# 切换至对应router的目录下
cd ~/wecross/routers-payment/127.0.0.1-8250-25500/

# 用脚本生成BCOS账户：账户类型（BCOS2.0），账户名（bcos_user1）
bash add_account.sh -t BCOS2.0 -n bcos_user1
```

生成的bcos_user1文件目录如下：

```
tree conf/accounts/bcos_user1/
conf/accounts/bcos_user1/
├── 0xxxxxxxxxxxxxxxx.key
└── account.toml
```

添加Fabric账户

所配置的账户可用于向Fabric1.4类型的链发交易。

```
# 用脚本生成Fabric账户：账户类型（Fabric1.4），账户名（fabric_user1）
bash add_account.sh -t Fabric1.4 -n fabric_user1
cp ~/wecross/fabric/certs/accounts/fabric_user1/* conf/accounts/fabric_user1/   # 拷
贝 Fabric链的证书，具体说明请参考《跨链接入》章节
```

生成的fabric_user1文件目录如下：

```
tree conf/accounts/fabric_user1/
conf/accounts/fabric_user1/
├── account.crt
├── account.key
└── account.toml
```

配置接入FISCO BCOS链

为router添加需要接入的链配置。

生成配置文件

切换至跨链路由的目录，用 add_chain.sh 脚本在conf目录下生成bcos的配置文件框架。

```
cd ~/wecross/routers-payment/127.0.0.1-8250-25500
 # -t 链类型，-n 指定链名字
bash add_chain.sh -t BCOS2.0 -n bcos
```

执行成功。如果执行出错，请查看屏幕打印提示。

```
Chain "bcos" config framework has been generated to "conf/chains/bcos"
```

生成的目录结构如下：

```
tree conf/chains/bcos/
conf/chains/bcos
├── WeCrossProxy
│   └── WeCrossProxy.sol # 代理合约
└── stub.toml            # chain配置文件
```

配置BCOS节点连接

- 拷贝证书

```
cp ~/wecross/bcos/nodes/127.0.0.1/sdk/* conf/chains/bcos/
```

- 修改配置

```
vim conf/chains/bcos/stub.toml
```

如果搭FISCO BCOS链采用的都是默认配置，那么将会得到一条单群组四节点的链，群组ID为1，可连接至节点0的channel端口20200，则配置如下（参考此处获取更详尽的配置说明）：

```
[common]
    name = 'bcos'
    type = 'BCOS2.0' # BCOS

[chain]
    groupId = 1 # default 1
    chainId = 1 # default 1

[channelService]
    caCert = 'ca.crt'
    sslCert = 'sdk.crt'
    sslKey = 'sdk.key'
    timeout = 300000  # ms, default 60000ms
    connectionsStr = ['127.0.0.1:20200']
```

部署代理合约

代理合约是插件与链交互的入口，执行命令进行部署。

```
cd ~/wecross/routers-payment/127.0.0.1-8250-25500

java -cp 'conf/:lib/*:plugin/*' com.webank.wecross.stub.bcos.normal.proxy.
↪ProxyContractDeployment deploy chains/bcos bcos_user1 # deploy conf下的链配置位置 账
户名
```

部署成功，输出

```
SUCCESS: WeCrossProxy:xxxxxxxx has been deployed! chain: chains/bcos
```

启动路由

启动跨链路由加载已配置的跨链资源。

```
cd ~/wecross/routers-payment/127.0.0.1-8250-25500

# 若WeCross跨链路由未停止，需要先停止
bash stop.sh

# 重新启动
bash start.sh
```

检查日志，可看到刷出已加载的跨链资源，ctrl + c 退出。

```
tail -f logs/info.log |grep "active resources"

2020-08-17 15:04:10.802 [Thread-3] INFO  WeCrossHost() - Current active resources:␣
↪payment.bcos.WeCrossProxy(local)
2020-08-17 15:04:20.824 [Thread-3] INFO  WeCrossHost() - Current active resources:␣
↪payment.bcos.WeCrossProxy(local)
2020-08-17 15:04:30.841 [Thread-3] INFO  WeCrossHost() - Current active resources:␣
↪payment.bcos.WeCrossProxy(local)
```

接入Fabric链

添加账户

在router中添加用于向链上发交易的账户。

添加Fabric账户

Fabric账户需配置多个

- admin账户：必配，一个admin账户，用于接入此Fabric链
- 机构admin账户：选配，每个Fabric的Org配置一个admin账户，用于在每个Org上部署chaincode，此例子中用于部署代理合约和sacc合约。
- 用户账户：选配，用于往链上发交易。

相关操作如下

```
# 切换至对应router的目录下
cd ~/wecross/routers-payment/127.0.0.1-8251-25501/

# 用脚本生成Fabric账户配置：账户类型（Fabric1.4），账户名（fabric_admin）
# 接入Fabric链，需要配置一个admin账户
bash add_account.sh -t Fabric1.4 -n fabric_admin

# 拷贝 Fabric链的证书，具体说明请参考《跨链接入》章节
cp ~/wecross/fabric/certs/accounts/fabric_admin/*  conf/accounts/fabric_admin/

# 为Fabric链的每个Org都配置一个admin账户，此处有两个org（Org1和Org2），分别配两个账户
# 配Org1的admin
bash add_account.sh -t Fabric1.4 -n fabric_admin_org1
cp ~/wecross/fabric/certs/accounts/fabric_admin_org1/*  conf/accounts/fabric_admin_
↪org1/

# 配Org2的admin
bash add_account.sh -t Fabric1.4 -n fabric_admin_org2
cp ~/wecross/fabric/certs/accounts/fabric_admin_org2/*  conf/accounts/fabric_admin_
↪org2/

# router-8250上配置的用户账户直接拷贝也可用
cp -r ~/wecross/routers-payment/127.0.0.1-8250-25500/conf/accounts/fabric_user1␣
↪conf/accounts/
```

- 修改配置

生成的账户配置，默认的mspid为Org1的，需将Org2的账户的mspid配置为Org2的。此处只需修改fabric_admin_org2账户的配置。详细配置操作说明请参考此处

```
vim conf/accounts/fabric_admin_org2/account.toml
```

内容为

```
[account]
    type = 'Fabric1.4'
    mspid = 'Org2MSP' # 配置为Org2MSP
    keystore = 'account.key'
    signcert = 'account.crt'
```

目前配置了四个账户，若此router需要向BCOS的链发交易，也可配置BCOS的账户。账户配置与接入的链无关，router间自动转发交易至相应的链。账户目录结构如下：

```
tree conf/accounts/
conf/accounts/
├── fabric_admin
│   ├── account.crt
│   ├── account.key
│   └── account.toml
├── fabric_admin_org1
│   ├── account.crt
│   ├── account.key
│   └── account.toml
├── fabric_admin_org2
│   ├── account.crt
│   ├── account.key
│   └── account.toml
└── fabric_user1
    ├── account.crt
    ├── account.key
    └── account.toml
```

配置接入Fabric链

为router添加需要接入的链配置。

生成配置文件

切换至跨链路由的目录，用 add_chain.sh 脚本在conf目录下生成Fabric的配置文件框架。

```
cd ~/wecross/routers-payment/127.0.0.1-8251-25501
# -t 链类型，-n 指定链名字
bash add_chain.sh -t Fabric1.4 -n fabric
```

执行成功。如果执行出错，请查看屏幕打印提示。

```
SUCCESS: Chain "fabric" config framework has been generated to "conf/chains/fabric"
```

生成的目录结构如下：

```
tree conf/chains/fabric/
conf/chains/fabric
├── WeCrossProxy
│   └── proxy.go            # 代理chaincode
└── stub.toml           # chain配置文件
```

配置Fabric节点连接

- 拷贝证书

```
# 证书具体说明请参考《跨链接入》章节
cp ~/wecross/fabric/certs/chains/fabric/* conf/chains/fabric/
```

- 修改配置

```
vim conf/chains/fabric/stub.toml
```

相关配置项使用默认即可。（参考此处获取更详尽的配置说明）

```
[common]
    name = 'fabric'
    type = 'Fabric1.4'

[fabricServices]
    channelName = 'mychannel'
    orgUserName = 'fabric_admin'
    ordererTlsCaFile = 'orderer-tlsca.crt'
    ordererAddress = 'grpcs://localhost:7050'

[orgs]
    [orgs.Org1]
        tlsCaFile = 'org1-tlsca.crt'
        adminName = 'fabric_admin_org1'
        endorsers = ['grpcs://localhost:7051']

    [orgs.Org2]
        tlsCaFile = 'org2-tlsca.crt'
        adminName = 'fabric_admin_org2'
        endorsers = ['grpcs://localhost:9051']
```

部署代理chaincode

执行命令，部署代理chaincode

```
cd ~/wecross/routers-payment/127.0.0.1-8251-25501

java -cp 'conf/:lib/*:plugin/*' com.webank.wecross.stub.fabric.proxy.
↪ProxyChaincodeDeployment deploy chains/fabric # deploy conf下的链配置位置
```

部署成功

```
SUCCESS: WeCrossProxy has been deployed to chains/fabric
```

启动路由

启动跨链路由加载配置好的跨链资源。

```
cd ~/wecross/routers-payment/127.0.0.1-8251-25501

# 若WeCross跨链路由未停止，需要先停止
bash stop.sh

# 重新启动
bash start.sh
```

3.3.3 部署WeCross控制台

WeCross提供了控制台，方便用户进行跨链开发和调试。可通过脚本build_console.sh搭建控制台。

下载WeCross控制台

执行如下命令进行下载（提供三种下载方式，可根据网络环境选择合适的方式进行下载），下载完成后在当前目录下生成WeCross-Console目录。

```
cd ~/wecross/
bash <(curl -sL https://github.com/WeBankFinTech/WeCross/releases/download/
↪resources/download_console.sh)
```

配置控制台

```
cd ~/wecross/WeCross-Console
# 拷贝连接router所需的TLS证书，从生成的routers-payment/cert/sdk目录下拷贝
cp ~/wecross/routers-payment/cert/sdk/* conf/ # 包含：ca.crt、node.nodeid、ssl.
↪crt、ssl.key
# 拷贝配置文件，并配置跨链路由RPC服务地址以及端口。此处采用默认配置，默认连接至本地8250端口。
cp conf/application-sample.toml conf/application.toml
```

重要:

- 若搭建WeCross的IP和端口未使用默认配置，需自行更改WeCross-Console/conf/application.toml，详见 控制台配置。

启动控制台

```
bash start.sh
```

启动成功则输出如下信息，通过help可查看控制台帮助

```
================================================================================
Welcome to WeCross console(v1.0.0-rc4)!
Type 'help' or 'h' for help. Type 'quit' or 'q' to quit console.
================================================================================
```

测试功能

```
# 查看连接的router当前支持接入的链类型
[WeCross]> supportedStubs
[BCOS2.0, GM_BCOS2.0, Fabric1.4]

# 退出控制台
[server1]> quit
```

更多控制台命令及含义详见控制台命令。

3.3.4 部署跨链资源

WeCross支持直接通过WeCross-Console部署跨链资源。

```
cd ~/wecross/WeCross-Console/
```

部署 Fabric 跨链资源

WeCross 支持通过 WeCross-Console 向指定的Fabric链上部署chaincode。

- 配置chaincode代码（部署sacc为例）
  - WeCross-Console的chaincode存放目录：`conf/contracts/chaincode/`
  - sacc代码放入目录：`conf/contracts/chaincode/sacc`（目录名sacc为chaincode的名字）
  - sacc目录中放置chaincode代码：sacc.go （代码名任意）

WeCross-Console中已默认存放了sacc，目录结构如下。

```
tree conf/contracts/chaincode/sacc
conf/contracts/chaincode/sacc
├── policy.yaml
└── sacc.go
```

- 启动控制台

部署chaincode相关的账户在router-8251，将Console配置为连接router-8251

```
cd ~/wecross/WeCross-Console
vim conf/application.toml
```

配置为

```
[connection]
    server =  '127.0.0.1:8251' # 连接 router 8251
    sslKey = 'classpath:ssl.key'
    sslCert = 'classpath:ssl.crt'
    caCert = 'classpath:ca.crt'
```

启动控制台

```
bash start.sh
```

- 部署chaincode

为不同的Org分别安装（install）相同的chaincode

> 参数：ipath（xxx.yyy.zzz，xxx.yyy为指定的链，zzz为chaincode名），机构admin账户，机构名，chaincode代码工程目录，指定一个版本，chaincode语言

```
[WeCross]> fabricInstall payment.fabric.sacc fabric_admin_org1 Org1 contracts/
↪chaincode/sacc 1.0 GO_LANG
Result: Success
[WeCross]> fabricInstall payment.fabric.sacc fabric_admin_org2 Org2 contracts/
↪chaincode/sacc 1.0 GO_LANG
Result: Success
```

实例化（instantiate）指定chaincode

> 参数：ipath，admin账户，对应的几个Org，chaincode代码工程目录，指定的版本，chaincode语言，背书策略（此处用默认），初始化参数

```
[WeCross]> fabricInstantiate payment.fabric.sacc fabric_admin ["Org1","Org2"]␣
↪contracts/chaincode/sacc 1.0 GO_LANG default ["a","10"]
Result: Instantiating... Please wait and use 'listResources' to check. See router
↪'s log for more information.
```

instantiate请求后，需等待1min左右。用`listResources`查看是否成功。若instantiate成功，可查询到资源`payment.fabric.sacc`。

```
[WeCross]> listResources
path: payment.fabric.sacc, type: Fabric1.4, distance: 0
total: 1

[WeCross]> quit // 退出控制台
```

部署 BCOS 跨链资源

WeCross 支持通过 WeCross-Console 向指定的BCOS链上部署合约。部署步骤如下。

- 配置合约代码

  - 以HelloWorld合约为例
  - WeCross-Console 的合约存放目录：`conf/contracts/solidity/`

目录下已有HelloWorld合约文件，若需部署其它合约，可将合约拷贝至相同位置。

```
tree conf/contracts/solidity/
conf/contracts/solidity/
└── HelloWorld.sol
```

- 启动控制台

部署合约相关的账户在router-8250，将Console配置为连接router-8250

```
cd ~/wecross/WeCross-Console
vim conf/application.toml
```

配置为

```
[connection]
    server =  '127.0.0.1:8250' # 连接 router 8250
    sslKey = 'classpath:ssl.key'
    sslCert = 'classpath:ssl.crt'
    caCert = 'classpath:ca.crt'
```

启动控制台

```
cd ~/wecross/WeCross-Console/
bash start.sh
```

- 部署合约

用`bcosDeploy`命令进行部署。

参数：ipath，代码目录，合约名，设置一个版本号

```
[WeCross]> bcosDeploy payment.bcos.HelloWorld bcos_user1 contracts/solidity/
↪HelloWorld.sol HelloWorld 1.0
Result: 0x1b557d68ebc51ed5b12438ff1666f8111718f47a
```

用`listResources`可查看此资源已部署

```
[WeCross]> listResources
path: payment.bcos.HelloWeCross, type: BCOS2.0, distance: 0
```

3.3.5 操作跨链资源

查看资源

进入控制台，用`listResources`命令查看WeCross跨连网络中的所有资源。可看到有两个资源:

- `payment.bcos.HelloWorld`
  - 对应于FISCO BCOS链上的HelloWorld.sol合约
- `payment.fabric.sacc`
  - 对应于Fabric链上的sacc.go合约

```
[WeCross]> listResources
path: payment.bcos.HelloWorld, type: BCOS2.0, distance: 0
path: payment.fabric.sacc, type: Fabric1.4, distance: 1
total: 2
```

查看账户

用listAccounts命令查看WeCross Router上已存在的账户，操作资源时用相应账户进行操作。

```
[WeCross]> listAccounts
name: fabric_user1, type: Fabric1.4
name: bcos_user1, type: BCOS2.0
total: 2
```

操作资源：payment.bcos.HelloWorld

- 读资源
  - 命令：call path 账户名 接口名 [参数列表]
  - 示例：call payment.bcos.HelloWorld bcos_user1 get

```
# 调用HelloWorld合约中的get接口
[WeCross]> call payment.bcos.HelloWorld bcos_user1 get
Result: [Hello, World!]
```

- 写资源
  - 命令：sendTransaction path 账户名 接口名 [参数列表]
  - 示例：sendTransaction payment.bcos.HelloWeCross bcos_user1 set Tom

```
# 调用HelloWeCross合约中的set接口
[WeCross]> sendTransaction payment.bcos.HelloWorld bcos_user1 set Tom
Txhash  : 0x7e747198f553cb2e90e729b52179533dc4321e520b0f11b83b1f0e81fa7ff716
BlockNum: 5
Result  : []      // 将Tom给set进去

[WeCross]> call payment.bcos.HelloWorld bcos_user1 get
Result: [Tom]     // 再次get，Tom已set
```

操作资源：payment.fabric.sacc

跨链资源是对各个不同链上资源的统一和抽象，因此操作的命令是保持一致的。

- 读资源

```
# 调用mycc合约中的query接口
[WeCross]> call payment.fabric.sacc fabric_user1 get a
Result: [10] // 初次get，a的值为10
```

- 写资源

```
# 调用sacc合约中的set接口
[WeCross]> sendTransaction payment.fabric.sacc fabric_user1 set a 666
Txhash  : eca4ecacf7b159c1499d6c190fcaf9fd7348bdb96cdbf35cd29b34ac9bd8e518
BlockNum: 7
Result  : [666]

[WeCross]> call payment.fabric.sacc fabric_user1 get a
Result: [666] // 再次get，a的值变成666

# 退出WeCross控制台
[WeCross]> quit # 若想再次启动控制台，cd至WeCross-Console，执行start.sh即可
```

恭喜，你已经完成了整个WeCross网络的体验。相信优秀的你已经对WeCross有了大致的了解。接下来，你可以基于WeCross Java SDK开发更多的跨连应用，通过统一的接口对各种链上的资源进行操作。

CHAPTER 4

操作手册

4.1 配置文件

本节描述WeCross Router的配置。

4.1.1 配置结构

WeCross Router的配置位于conf目录下，分为：

- 主配置（wecross.toml）：配置Router连接等信息
- 链配置（chains/<chain_name>/stub.toml）：配置连接至对应区块链、链上资源
- 账户配置（accounts/<account_name>/account.toml）：配置可用于发交易账户的公私钥等信息

配置的目录结构如下：

```
# 这是conf目录下标准的配置结构，Router配置连接了两条链，分别叫bcos和fabric
.
├── log4j2.xml    // 日志配置文件，无需更改
├── accounts
│   ├── bcos_user1
│   │   └── account.toml  // 账户配置
│   └── fabric_user1
│       └── account.toml  // 账户配置
├── chains
│   ├── bcos
│   │   └── stub.toml           // 链配置
│   └── fabric
│       └── stub.toml      // 链配置
└── wecross.toml                    // 主配置
```

4.1.2 主配置

主配置为 conf/wecross.toml，配置示例如下：

```
[common]
    zone = 'payment'
    visible = true

[chains]
    path = 'classpath:chains'

[rpc] # rpc ip & port
    address = '127.0.0.1'
    port = 8250
    caCert = 'classpath:ca.crt'
    sslCert = 'classpath:ssl.crt'
    sslKey = 'classpath:ssl.key'

[p2p]
    listenIP = '0.0.0.0'
    listenPort = 25500
    caCert = 'classpath:ca.crt'
    sslCert = 'classpath:ssl.crt'
    sslKey = 'classpath:ssl.key'
    peers = ['127.0.0.1:25501']
    threadNum = 500


#[[htlc]]
#    selfPath = 'payment.bcos.htlc'
#    account1 = 'bcos_default_account'
#    counterpartyPath = 'payment.fabric.htlc'
#    account2 = 'fabric_default_account'
```

跨链服务配置有五个配置项，分别是[common]、[chains]、[rpc]、[p2p]以及[test]，各个配置项含义如下：

- [common]通用配置
  - zone：字符串；跨链分区标识符；通常一种跨链业务/应用为一个跨链分区
  - visible：布尔；可见性；标明当前跨链分区下的资源是否对其他跨链分区可见
- [chains] 链配置
  - path：字符串；链配置的根目录；WeCross从该目录下去加载各个链的配置
- [rpc]RPC配置
  - address：字符串；RPC服务监听地址，通常设置为本机IP地址
  - port：整型；WeCross Router的RPC端口；WeCross Java SDK调用Router的端口
  - sslOn：SSl开关，默认true
  - caCert：字符串；WeCross Router根证书路径
  - sslCert：字符串；WeCross Router证书路径
  - sslKey：字符串；WeCross Router私钥路径
  - threadNum：rpc线程数，默认16
  - threadQueueCapacity：任务队列容量，默认10000
- [p2p] 组网配置
  - listenIP：字符串；P2P服务监听地址；一般为'0.0.0.0'
  - listenPort ： 整型；P2P服务监听端口；WeCross Router之间交换消息的端口
  - caCert ： 字符串；WeCross Router根证书路径

  - sslCert：字符串；WeCross Router证书路径
  - sslKey：字符串；WeCross Router私钥路径
  - peers：字符串数组；peer列表；需要互相连接的WeCross Router列表
  - threadNum：p2p线程数，默认16
  - threadQueueCapacity：任务队列容量，默认10000
- `[htlc]`htlc配置(可选)
  - selfPath：本地配置的htlc合约资源路径
  - account1：可调用本地配置的htlc合约的账户
  - counterpartyPath：本地配置的htlc合约的对手方合约路径
  - account2：可调用对手方htlc合约的账户

注：

1. WeCross启动时会把`conf`目录指定为classpath，若配置项的路径中开头为`classpath:`，则以`conf`为相对目录。
2. `[p2p]`配置项中的证书和私钥可以通过create_cert.sh脚本生成。
3. 若通过build_wecross.sh脚本生成的项目，那么已自动帮忙配置好了`wecross.toml`，包括P2P的配置，其中链配置的根目录默认为`chains`。

4.1.3 链配置

链配置是Router连接每个区块链的配置：

- 指定链名

在`chains/<chain_name>/stub.toml`目录下，通过目录名`<chain_name>`指定链名。

- 区块链链接信息

在`stub.toml`中配置与区块链交互所需链接的信息。

- 跨链资源

在`stub.toml`中配置需要参与跨链的资源。

WeCross启动后会在`wecross.toml`中所指定的`chains`的根目录下去遍历所有的一级目录，目录名即为chain的名字，不同的目录代表不同的链，然后尝试读取每个目录下的`stub.toml`文件。

目前WeCross支持的Stub类型包括：FISCO BCOS和Fabric。

配置FISCO BCOS

请参考：FISCO BCOS 2.0插件配置

配置Fabric

请参考：Fabric 1.4插件配置

4.1.4 账户配置

在Router中配置账户，与链进行交互。配置操作包括：

- 指定账户名（`accounts/<account_name>/account.toml`下，通过目录名`<account_name>`指定账户名）
- 指定账户类型（用于BCOS、Fabric等，在`account.toml`中配置）
- 指定账户其它信息（密钥等，在`account.toml`中配置）

WeCross启动后会在`accounts`的根目录下去遍历所有的一级目录，目录名即为账户的名字，不同的目录代表不同的链，然后尝试读取每个目录下的`account.toml`文件。

配置不同类型的账户，与不同类型的链进行操作。目前WeCross支持的类型包括：FISCO BCOS和Fabric。

配置FISCO BCOS

请参考：FISCO BCOS 2.0账户配置

配置Fabric

请参考：Fabric 1.4账户配置

4.2 控制台

控制台是WeCross重要的交互式客户端工具，它通过WeCross-Java-SDK与WeCross 跨链代理建立连接，实现对跨链资源的读写访问请求。控制台拥有丰富的命令，包括获取跨链资源列表，查询资源状态，以及所有的JSON-RPC接口命令。

4.2.1 控制台命令

控制台命令可分为两类，普通命令和交互式命令。

普通命令

普通命令由两部分组成，即指令和指令相关的参数：

- **指令**: 指令是执行的操作命令，包括获取跨链资源列表，查询资源状态指令等，其中部分指令调用JSON-RPC接口，因此与JSON-RPC接口同名。 **使用提示： 指令可以使用tab键补全，并且支持按上下键显示历史输入指令。**
- **指令相关的参数**: 指令调用接口需要的参数，指令与参数以及参数与参数之间均用空格分隔。与JSON-RPC接口同名命令的输入参数和获取信息字段的详细解释参考JSON-RPC API。

交互式命令

WeCross控制台为了方便用户使用，还提供了交互式的使用方式，比如将跨链资源标识赋值给变量，初始化一个类，并用`.command`的方式访问方法。 详见：交互式命令

4.2.2 常用命令链接

普通命令

- 状态查询
  - *listResources*：查看资源列表
  - *detail*：查看资源详情
  - *listAccounts*：查看账户列表
  - *supportedStubs*：查看连接的router支持接入的链类型
- 资源调用
  - *call*：调用链上资源，用于查询，不触发出块
  - *sendTransaction*：发交易，用于改变链上资源，触发出块

- 资源部署
  - BCOS：*bcosDeploy*、*bcosRegister*
  - Fabric：*fabricInstall*、*fabricInstantiate*、*fabricUpgrade*
- 跨链事务
  - *startTransaction*：开始两阶段事务
  - *execTransaction*：发起事务交易
  - *callTransaction*：读取事务过程中的数据
  - *commitTransaction*：提交事务，确认事务执行过程中所有的变动
  - *rollbackTransaction*：撤销本次事务的所有变更时
- 跨链转账
  - *newHTLCProposal*：创建转账提案

交互式命令

- 初始化资源实例: *WeCross.getResource*
- 访问资源UBI接口: *[resource].[command]*

4.2.3 快捷键

- `Ctrl+A`：光标移动到行首
- `Ctrl+E`：光标移动到行尾
- `Ctrl+R`：搜索输入的历史命令
- ↑： 向前浏览历史命令
- ↓： 向后浏览历史命令
- `tab`： 自动补全，支持命令、变量名、资源名、账户名以及其它固定参数的补全

4.2.4 控制台响应

当发起一个控制台命令时，控制台会获取命令执行的结果，并且在终端展示执行结果，执行结果分为2类：

- **正确结果:** 命令返回正确的执行结果，以字符串或是json的形式返回。
- **错误结果:** 命令返回错误的执行结果，以字符串或是json的形式返回。
- **状态码:** 控制台的命令调用JSON-RPC接口时，状态码参考这里。

4.2.5 控制台配置与运行

重要: 前置条件：部署WeCross请参考 快速部署。

获取控制台

可通过脚本download_console.sh获取控制台。

```
cd ~ && mkdir -p wecross && cd wecross
# 获取控制台
bash <(curl -sL https://github.com/WeBankFinTech/WeCross-Console/releases/download/
↪resources/download_console.sh)
```

执行成功后，会生成WeCross-Console目录，结构如下：

```
├── apps
│   └── wecross-console.jar   # 控制台jar包
├── conf
│   ├── application-sample.toml    # 配置示例文件
│   └── log4j2.xml             # 日志配置文件
├── download_console.sh        # 获取控制台脚本
├── lib                        # 相关依赖的jar包目录
├── logs                       # 日志文件
└── start.sh                   # 启动脚本
```

配置控制台

控制台配置文件为 conf/application-toml，启动控制台前需配置

```
cd ~/wecross/WeCross-Console
# 拷贝配置sample
cp conf/application-sample.toml conf/application.toml

# 拷贝连接router所需的TLS证书，从生成的routers-payment/cert/sdk目录下拷贝
cp ~/wecross/routers-payment/cert/sdk/* conf/ # 包含：ca.crt、node.nodeid、ssl.
↪crt、ssl.key

# 配置
vim conf/application.toml
```

配置与控制台r与某个router的连接

```
[connection]
    server =  '127.0.0.1:8250' # 对应router的ip和rpc端口
    sslKey = 'classpath:ssl.key'
    sslCert = 'classpath:ssl.crt'
    caCert = 'classpath:ca.crt'
```

启动控制台

在WeCross已经开启的情况下，启动控制台

```
cd ~/wecross/WeCross-Console
bash start.sh
# 输出下述信息表明启动成功
================================================================================================
Welcome to WeCross console(1.0.0-rc4)!
Type 'help' or 'h' for help. Type 'quit' or 'q' to quit console.

================================================================================================
```

4.2.6 普通命令

以下所有跨链资源相关命令的执行结果以实际配置为准，此处只是示例。

help

输入help或者h，查看控制台所有的命令。

```
[WeCross]> help
--------------------------------------------------------------------------------
↪----------
quit                                Quit console.
supportedStubs                      List supported stubs of WeCross router.
listAccounts                        List all accounts stored in WeCross router.
listLocalResources                  List local resources configured by WeCross␣
↪server.
listResources                       List all resources including remote resources.
status                              Check if the resource exists.
detail                              Get resource information.
call                                Call constant method of smart contract.
sendTransaction                     Call non-constant method of smart contract.
callTransaction                     Call constant method of smart contract during␣
↪transaction.
execTransaction                     Call non-constant method of smart contract␣
↪during transaction.
startTransaction                    Start a 2pc transaction.
commitTransaction                   Commit a 2pc transaction.
rollbackTransaction                 Rollback a 2pc transaction.
getTransactionInfo                  Get info of specified transaction.
getTransactionIDs                   Get transaction ids of 2pc.
bcosDeploy                          Deploy contract in BCOS chain.
bcosRegister                        Register contract abi in BCOS chain.
fabricInstall                       Install chaincode in fabric chain.
fabricInstantiate                   Instantiate chaincode in fabric chain.
fabricUpgrade                       Upgrade chaincode in fabric chain.
genTimelock                         Generate two valid timelocks.
genSecretAndHash                    Generate a secret and its hash.
newHTLCProposal                     Create a htlc transfer proposal .
checkTransferStatus                 Check htlc transfer status by hash.
WeCross.getResource                 Init resource by path and account name, and␣
↪assign it to a custom variable.
[resource].[command]                Equal to: command [path] [account name].

--------------------------------------------------------------------------------
↪----------
```

注：

- help显示每条命令的含义是：命令 命令功能描述
- 查看具体命令的使用介绍说明，输入命令 -h或–help查看。例如：

```
[WeCross]> detail -h
--------------------------------------------------------------------------------
↪----------
Get the resource information
Usage: detail [path]
--------------------------------------------------------------------------------
↪----------
```

supportedStubs

显示router当前支持的插件列表。

```
[WeCross]> supportedStubs
[BCOS2.0, GM_BCOS2.0, Fabric1.4]
```

listAccounts

显示所有已配置的账户列表。

```
name: fabric_user1, type: Fabric1.4
name: fabric_default_account, type: Fabric1.4
name: bcos_user1, type: BCOS2.0
name: bcos_sender, type: BCOS2.0
name: bcos_default_account, type: BCOS2.0
total: 5
```

listLocalResources

显示router配置的跨链资源。

```
[WeCross]> listLocalResources
path: payment.bcos.htlc, type: BCOS2.0, distance: 0
path: payment.bcos.HelloWeCross, type: BCOS2.0, distance: 0
total: 2
```

listResources

查看WeCross跨链代理本地配置的跨链资源和所有的远程资源。

```
[WeCross]> listResources
path: payment.bcos.htlc, type: BCOS2.0, distance: 0
path: payment.bcos.HelloWeCross, type: BCOS2.0, distance: 0
path: payment.fabric.htlc, type: Fabric1.4, distance: 1
path: payment.fabric.abac, type: Fabric1.4, distance: 1
total: 4
```

status

查看跨链资源的状态，即是否存在于连接的router中。

参数:

- path：跨链资源标识。

```
[WeCross]> status payment.bcos.HelloWeCross
exists
```

detail

查看跨链资源的详细信息。

参数:

- path：跨链资源标识。

```
[WeCross]> detail payment.bcos.HelloWeCross
ResourceDetail{
 path='payment.bcos.HelloWeCross',
 distance=0',
 stubType='BCOS2.0',
 properties={
  BCOS_PROPERTY_CHAIN_ID=1,
  BCOS_PROPERTY_GROUP_ID=1,
  HelloWeCross=0x708133d132372727ce3848a16d47ab4daf77698c
 },
 checksum='0xb452f3d12c91b6cd93e083a518d2ea2cffbcf3d8b971221a5224f07a3be5e41a'
}

[WeCross]> detail payment.fabric.abac
ResourceDetail{
 path='payment.fabric.abac',
 distance=1',
 stubType='Fabric1.4',
 properties={
  PROPOSAL_WAIT_TIME=120000,
  CHAINCODE_TYPE=GO_LANG,
  CHANNEL_NAME=mychannel,
  CHAINCODE_NAME=mycc
 },
 checksum='c77f0ac3ead48d106d357ffe0725b9761bd55d3e27edd8ce669ad8b470a27bc8'
}
```

call

调用智能合约的方法，不涉及状态的更改，不发交易。

参数：

- path：跨链资源标识。
- accountName：交易签名账户，router上配置的账户名（`listAccounts`命令可查询）。
- method：合约方法名。
- args：参数列表。

```
[WeCross]> call payment.bcos.HelloWeCross bcos_user1 get
Result: [Talk is cheap, Show me the code]
```

sendTransaction

调用智能合约的方法，会更改链上状态，需要发交易。

参数：

- path：跨链资源标识。
- accountName：交易签名账户，router上配置的账户名。
- method：合约方法名。
- args：参数列表。

```
[WeCross]> sendTransaction payment.bcos.HelloWeCross bcos_user1 set hello wecross
Txhash  : 0x66f94d387df2b16bea26e6bcf037c23f0f13db28dc4734588de2d57a97051c54
BlockNum: 2219
Result  : [hello, wecross]
```

bcosDepoly

FISCO BCOS 合约部署命令，成功返回部署的合约地址，失败返回错误描述

```
bcosDeploy -h
--------------------------------------------------------------------------------
----------
Deploy contract and register contract info to CNS in BCOS chain
Usage: bcosDeploy [Path] [Account] [Source file path] [Class name] [Version]
Path -- e.g: [zone.chain.res], specify which the path to be deployed
Account -- Choose an account to send transaction
Source file path -- The solidity source code file path, e.g: HelloWorld.sol
Class name -- The contract to be deploy
Version -- The contract version
Example:
    bcosDeploy payment.bcos.HelloWorld bcos_user1 contracts/solidity/HelloWorld.
sol HelloWorld 1.0
```

参数：

- Path: 跨链资源标示，用于标记部署的合约资源
- Account: 部署合约发送交易的账户
- Source file path: 部署的合约路径，支持绝对路径/相对路径
- Class Name: 部署的合约名
- Verseion: 部署合约版本号，**注意：同一合约版本号唯一，再次部署相同的版本号会失败**

示例：

```
[WeCross]> bcosDeploy payment.my_bcos_chain.HelloWorld my_bcos_account contracts/
solidity/HelloWorld.sol HelloWorld 1.0
Result: 0x79a178e71dc77fbccd31d464c114c95403a31e00
```

bcosRegister

FISCO BCOS 注册已有合约为跨链资源，成功返回Success，失败返回错误描述

```
bcosRegister -h
--------------------------------------------------------------------------------
----------
Register contract info to CNS in BCOS chain
Usage: bcosRegister [Path] [Account] [Source file path] [Contract address]␣
[Version]
Path -- e.g: [zone.chain.res], specify which the path to be register
Account -- Choose an account to send transaction
Source file path -- The solidity source code/solidity abi file path, e.g:␣
HelloWorld.sol or HelloWorld.abi
Contract address -- contract address
Version -- The contract version
Example:
    bcosRegister payment.bcos.HelloWorld bcos_user1 contracts/solidity/HelloWorld.
sol 0x2c8595f82dc930208314030abc6f5c4ddbc8864f 1.0
    bcosRegister payment.bcos.HelloWorld bcos_user1 /data/app/HelloWorld.abi␣
0x2c8595f82dc930208314030abc6f5c4ddbc8864f 1.0
--------------------------------------------------------------------------------
----------
```

参数：

- Path: 跨链资源标示，用于标记注册的合约资源

- Account: 注册合约发送交易的账户
- Source file path: 合约的ABI文件路径或者合约源码路径
- Contract address: 注册的合约地址
- Version: 合约版本号，**注意：同一部署合约版本号唯一，再次部署相同版本号的合约会失败**

示例：

```
> bcosRegister payment.my_bcos_chain.HelloWorld my_bcos_account contracts/solidity/
↪HelloWorld.sol 0x2c8595f82dc930208314030abc6f5c4ddbc8864f v1.0
Result: Success
```

fabricInstall

Fabric 安装链码命令，安装后需fabricInstantiate来启动链码

参数：

- path：跨链资源标识。
- account：被安装链码的endorser所属机构的admin账户
- orgName：被安装链码的endorser所属的机构
- sourcePath：链码工程所在目录，支持绝对路径和WeCross-Console的conf目录内的相对路径
- version：指定一个版本，fabricInstantiate时与此版本对应
- language：指定一个链码语言，支持GO_LANG和JAVA

```
[WeCross]> fabricInstall payment.fabric.sacc fabric_admin_org1 Org1 contracts/
↪chaincode/sacc 1.0 GO_LANG
Result: Success
[WeCross]> fabricInstall payment.fabric.sacc fabric_admin_org2 Org2 contracts/
↪chaincode/sacc 1.0 GO_LANG
Result: Success
```

fabricInstantiate

Fabric 启动（实例化）已安装的链码。此步骤前需先用fabricInstall向指定机构安装链码。

参数：

- path：跨链资源标识。
- account：指定一个发交易的账户
- orgNames：链码被安装的的机构列表
- sourcePath：链码工程所在目录，支持绝对路径和WeCross-Console的conf目录内的相对路径
- version：指定一个版本，与fabricInstall时的版本对应
- language：指定一个链码语言，支持GO_LANG和JAVA
- policy：指定背书策略文件，设置default为OR所有endorser
- initArgs：链码初始化参数

```
[WeCross]> fabricInstantiate payment.fabric.sacc fabric_admin ["Org1","Org2"]␣
↪contracts/chaincode/sacc 1.0 GO_LANG default ["a","10"]
Result: Instantiating... Please wait and use 'listResources' to check. See router
↪'s log for more information.
```

启动时间较长（1min左右），可用listResources查看是否已启动，若长时间未启动，可查看router的日志进行排查。

fabricUpgrade

Fabric 升级已启动的链码逻辑，不改变已上链的数据。此步骤前需先用fabricInstall向指定机构安装另一个版本的链码。

参数：

- path：跨链资源标识。
- account：指定一个发交易的账户
- orgNames：链码被安装的的机构列表
- sourcePath：链码工程所在目录，支持绝对路径和WeCross-Console的conf目录内的相对路径
- version：指定一个版本，与fabricInstall时的版本对应
- language：指定一个链码语言，支持GO_LANG和JAVA
- policy：指定背书策略文件，设置default为OR所有endorser
- initArgs：链码初始化参数

```
[WeCross]> fabricUpgrade payment.fabric.sacc fabric_admin ["Org1","Org2"]␣
↪contracts/chaincode/sacc 2.0 GO_LANG default ["a","10"]
Result: Upgrading... Please wait and use 'detail' to check the version. See router
↪'s log for more information.
```

升级时间较长（1min左右），可用`detail payment.fabric.sacc`查看版本号，若长时间升级完成，可查看router的日志进行排查。

genTimelock

跨链转账辅助命令，根据时间差生成两个合法的时间戳。

参数：

- interval：时间间隔

```
[WeCross]> genTimelock  300
timelock0: 1586917289
timelock1: 1586916989
```

genSecretAndHash

跨链转账辅助命令，生成一个秘密和它的哈希。

```
[WeCross]> genSecretAndHash
hash   : 66ebd11ec6cc289aebe8c0e24555b1e58a5191410043519960d26027f749c54f
secret: afd1c0f9c2f8acc2c1ed839ef506e8e0d0b4636644a889f5aa8e65360420d2a9
```

newHTLCProposal

新建一个基于哈希时间锁合约的跨链转账提案，该命令由两条链的资金转出方分别执行。

参数：

- path：跨链转账资源标识。

- accountName：资产转出者在router上配置的账户名。
- args：提案信息，包括两条链的转账信息。
  - hash：唯一标识，提案号，
  - secret：提案号的哈希原像
  - role：身份，发起方-true，参与方-false。发起方需要传入secret，参与方secret传null。
  - sender0：发起方的资金转出者在对应链上的地址
  - receiver0：发起方的资金接收者在对应链上的地址
  - amount0：发起方的转出金额
  - timelock0：发起方的超时时间
  - sender1：参与方的资金转出者在对应链上的地址
  - receiver1：参与方的资金接收者在对应链上的地址
  - amount1：参与方的转出金额
  - timelock1：参与方的超时时间，小于发起方的超时时间

```
[WeCross]> newHTLCProposal payment.bcos.htlc bcos_sender␣
↪88b6cea9b5ece573c6c35cb3f1a2237bf380dfbbf9155b82d5816344cdac0185 null false␣
↪Admin@org1.example.com User1@org1.example.com 200 2000010000␣
↪0x55f934bcbe1e9aef8337f5551142a442fdde781c␣
↪0x2b5ad5c4795c026514f8317c7a215e218dccd6cf  100 2000000000

Txhash: 0x244d302382d03985eebcc1f7d95d0d4eef7ff2b3d528fdf7c93effa94175e921
BlockNum: 2222
Result: [create a htlc transfer proposal successfully]
```

checkTransferStatus

根据提案号（Hash）查询htlc转账状态。

参数：

- path：跨链资源标识。
- accountName：交易签名账户，router上配置的账户名。
- method：合约方法名。
- hash：转账提案号。

```
[WeCross]> checkTransferStatus payment.bcos.htlc bcos_sender␣
↪dcbdf73ee6fdbe6672142c7776c2d21ff7acc6f0d61975e83c3b396a364bee93
status: succeeded!
```

startTransaction

写接口，开始两阶段事务

参数：

- transactionID：事务ID，类型为字符串，由用户指定，作为事务的唯一标识，后续所有的事务资源操作都必须指定该事务ID
- account_1 ... account_n：用于开始事务的账号列表，由于两阶段事务可能跨越多种区块链，多种区块链会使用不同类型的账号，因此开始事务时，需要为每种区块链指定至少一个账号，WeCross会使用相应类型的账号向链上发送开始事务交易，该账号列表仅用于开始事务，事务开始后，可以使用该账号列表以外的账号来发送事务交易

- path_1 . . . path_n：参与事务的资源路径列表，路径列表中的资源会被本次事务锁定，锁定后仅限本事务相关的交易才能对这些资源发起写操作，非本次事务的所有写操作都会被拒绝

```
[WeCross]> startTransaction 0001 bcos_user1 fabric_user1 payment.bcos.2pc payment.
↪fabric.2pc
Result: success!
```

execTransaction

写接口，发起事务交易

参数：

- path：资源路径
- account：交易账号
- transactionID：事务ID，该资源正在参与事务的ID
- seq：事务编号，本次操作的编号，每次事务交易唯一，要求递增
- method：接口名，同sendTransaction。需要注意的是，该接口需要在合约中配套以_revert结尾的回滚接口。
- args：参数，同sendTransaction

```
[WeCross]> execTransaction payment.bcos.2pc bcos_user1 0001 1 newEvidence key1␣
↪evidence1
Result: [true]

[WeCross]> execTransaction payment.fabric.2pc fabric_user1 0001 1 newEvidence key1␣
↪evidence1
Result: [newEvidence success]
```

callTransaction

读接口，查询事务中的数据

参数：

- path：资源路径
- account：交易账号
- transactionID：事务ID，该资源正在参与事务的ID
- method：接口名，同sendTransaction。
- args：参数，同sendTransaction

```
[WeCross]> callTransaction payment.bcos.2pc bcos_user1 0002 queryEvidence key1
Result: [evidence1]
```

commitTransaction

写接口，提交事务，确认事务执行过程中所有的变动

参数：

- transactionID：事务ID，待提交事务的ID

- account_1 ... account_n：用于提交事务的账号列表，由于两阶段事务可能跨越多种区块链，多种区块链会使用不同类型的账号，需要为每种区块链指定至少一个账号，WeCross会使用相应类型的账号向链上发送提交事务交易
- path_1 ... path_n：用于提交事务的路径列表

```
[WeCross]> commitTransaction 0001 bcos_user1 fabric_user1 payment.bcos.2pc payment.
↪fabric.2pc
```

rollbackTransaction

写接口，撤销本次事务的所有变更时

参数：

- transactionID：事务ID，待回滚事务的ID
- account_1 ... account_n：用于回滚事务的账号列表，由于两阶段事务可能跨越多种区块链，多种区块链会使用不同类型的账号，需要为每种区块链指定至少一个账号，WeCross会使用相应类型的账号向链上发送回滚事务交易
- path_1 ... path_n：用于回滚事务的路径列表

```
# 查看开始前的状态
[WeCross]> call payment.bcos.2pc bcos_user1 queryEvidence key2
Result: []

# 开始事务
[WeCross]> startTransaction 0002 bcos_user1 fabric_user1 payment.bcos.2pc payment.
↪fabric.2pc
Result: success!

# 执行事务
[WeCross]> execTransaction payment.bcos.2pc bcos_user1 0002 1 newEvidence key2␣
↪evidence2
Result: [true]

# 读事务数据
[WeCross]> callTransaction payment.bcos.2pc bcos_user1 0002 queryEvidence key2
Result: [evidence2]

# 回滚事务
[WeCross]> rollbackTransaction 0002 bcos_user1 fabric_user1 payment.bcos.2pc␣
↪payment.fabric.2pc
Result: success!

# 查看事务回滚后的状态，和开始前保持一致
[WeCross]> call payment.bcos.2pc bcos_user1 queryEvidence key2
Result: []
```

getTransactionInfo

读接口，查询事务信息

参数：

- transactionID：事务ID，待提交事务的ID
- account_1 ... account_n：如果涉及多条链需要多个账户
- path_1 ... path_n：参与事务的资源路径列表

```
[WeCross]> getTransactionInfo 0001 bcos_user1 fabric_user1 payment.bcos.2pc␣
↪payment.fabric.2pc
XATransactionInfo{
 transactionID='0001',
 status=1,
 startTimestamp=1596962127015,
 commitTimestamp=1596962353724,
 rollbackTimestamp=0,
 paths=[
  payment.fabric.2pc,
  payment.bcos.2pc
 ],
 transactionSteps=[
  XATransactionStep{
   seq=1,
   contract='0xd6cd8179b796f0fc04718364c232723833b8ca59',
   path='payment.bcos.2pc',
   timestamp='1596962287683',
   func='newEvidence(string,
   string)',
   args=
↪'0000000000000000000000000000000000000000000000000000000000000040000000000000000000000000000000000
↪'
  },
  XATransactionStep{
   seq=2,
   contract='null',
   path='payment.fabric.2pc',
   timestamp='1596962314',
   func='newEvidence',
   args='{
    "args":[
     "key1",
     "evidence1"
    ]
   }'
  }
 ]
}
```

getTransactionIDs

读接口，查询链上的事务ID

参数：

- account：交易账户
- path：指定需要查询的链
- option：选项。0全部事务，1已完成的事务，2未完成的事务

```
[WeCross]> getTransactionIDs payment.bcos.2pc bcos_user1 0
Result: [0001, 0002]

[WeCross]> getTransactionIDs payment.bcos.2pc bcos_user1 1
Result: [0001]

[WeCross]> getTransactionIDs payment.bcos.2pc bcos_user1 2
Result: [0002]
```

4.2.7 交互式命令

WeCross.getResource

WeCross控制台提供了一个资源类，通过方法getResource来初始化一个跨链资源实例，并且赋值给一个变量。 这样调用同一个跨链资源的不同UBI接口时，不再需要每次都输入跨链资源标识。

```
# myResource 是自定义的变量名
[WeCross]> myResource = WeCross.getResource payment.bcos.HelloWeCross bcos_user1

# 还可以将跨链资源标识赋值给变量，通过变量名来初始化一个跨链资源实例
[WeCross]> path = payment.bcos.HelloWeCross

[WeCross]> myResource = WeCross.getResource path bcos_user1
```

[resource].[command]

当初始化一个跨链资源实例后，就可以通过.command的方式，调用跨链资源的UBI接口。

```
# 输入变量名，通过tab键可以看到能够访问的所有命令
[WeCross]> myResource.
myResource.call              myResource.status
myResource.detail            myResource.sendTransaction
```

status

```
[WeCross]> myResource.status
exists
```

detail

```
[WeCross]> myResource.detail
ResourceDetail{
 path='payment.bcos.HelloWeCross',
 distance=0',
 stubType='BCOS2.0',
 properties={
  BCOS_PROPERTY_CHAIN_ID=1,
  BCOS_PROPERTY_GROUP_ID=1,
  HelloWeCross=0x9bb68f32a63e70a4951d109f9566170f26d4bd46
 },
 checksum='0x888d067b77cbb04e299e675ee4b925fdfd60405241ec241e845b7e41692d53b1'
}
```

call

```
[WeCross]> myResource.call get
Result: [hello, wecross]
```

sendTransaction

```
[WeCross]> myResource.sendTransaction set hello world
Txhash  : 0x616a55a7817f843d81f8c7b65449963fc2b7a07398b853829bf85b2e1261516f
BlockNum: 2224
Result  : [hello, world]
```

4.3 脚本介绍

为了方便用户使用，WeCross提供了丰富的脚本，脚本位于WeCross跨链路由的根目录下(如：`~/demo/routers-payment/127.0.0.1-8250-25500/`)，本章节将对这些脚步做详细介绍。

4.3.1 启动脚本

start.sh

启动脚本`start.sh`用于启动WeCross服务，启动过程中的完整信息记录在start.out中。

```
bash start.sh
```

成功输出：

```
Wecross start successfully
```

失败输出：

```
WeCross start failed
See logs/error.log for details
```

4.3.2 停止脚本

stop.sh

停止脚本`stop.sh`用于停止WeCross服务。

```
bash  stop.sh
```

4.3.3 构建WeCross脚本

build_wecross.sh

生成WeCross跨链路由网络

```
Usage:
    -n  <zone id>                   [Required]   set zone ID
    -l  <ip:rpc-port:p2p-port>      [Optional]   "ip:rpc-port:p2p-port" e.g:"127.0.
↪0.1:8250:25500"
    -f  <ip list file>              [Optional]   split by line, every line should␣
↪be "ip:rpc-port:p2p-port". eg "127.0.0.1:8250:25500"
    -c  <ca dir>                    [Optional]   dir of existing ca
    -o  <output dir>                [Optional]   default <your pwd>
    -z  <generate tar packet>       [Optional]   default no
    -T  <enable test mode>          [Optional]   default no. Enable test resource.
    -h  call for help
e.g
    bash build_wecross.sh -n payment -l 127.0.0.1:8250:25500
    bash build_wecross.sh -n payment -f ipfile
```

- **-n**：指定跨链分区标识
- **-l**：可选，指定生成一个跨链路由，与-f二选一，单行，如：192.168.0.1:8250:25500
- **-f**：可选，指定生成多个跨链路由，与-l二选一，多行，**不可有空行**，例如：

```
192.168.0.1:8250:25500
192.168.0.1:8251:25501
192.168.0.2:8252:25502
192.168.0.3:8253:25503
192.168.0.4:8254:25504
```

- **-c**：可选，指定跨链路由基于某个路径下的ca证书生成
- **-o**：可选，指定跨链路由生成目录，默认wecross/
- **-z**：可选，若设置，则生成跨链路由的压缩包，方便拷贝至其它机器
- **-T**：可选，若设置，生成的跨链路由开启测试资源
- **-h**：可选，打印Usage

4.3.4 添加账户脚本

add_account.sh

脚本add_account.sh用于在router中创建特定区块链的账户。

可通过-h查看帮助信息

```
Usage:
    -t <type>                               [Required] type of account, BCOS2.0 or␣
↪Fabric1.4
    -n <name>                               [Required] name of account
    -d <dir>                                [Optional] generated target_directory,␣
↪default conf/accounts/
    -h                                      [Optional] Help
```

- **-t**：账户类型，按照插件选择，如BCOS2.0，GM_BCOS2.0或Fabric1.4
- **-n**：指定账户名，发交易时使用，用区分不同的账户
- **-d**：账户目录，默认生成在conf/accounts下

不同的链有不同的操作方法，具体操作请查看（操作后，请重启router，让router重启加载配置）：

- BCOS2.0 账户配置
- Fabric1.4 账户配置

4.3.5 添加新接入链脚本

add_chains.sh

脚本add_chains.sh用于在router中创建特定区块链的连接配置

```
Usage:
    -t <type>                               [Required] type of chain, BCOS2.0 or␣
↪Fabric1.4
    -n <name>                               [Required] name of chain
    -d <dir>                                [Optional] generated target_directory,␣
↪default conf/stubs/
    -h                                      [Optional] Help
```

- **-t**：连接类型，按照插件选择，如BCOS2.0或Fabric1.4

- **-n**：连接名，账户名称
- **-d**：连接配置目录，默认生成在conf/chains/下

不同的链有不同的操作方法，具体操作请查看（操作后，请重启router，让router重启加载配置）：

- BCOS2.0 接入配置
- Fabric1.4 接入配置

4.3.6 创建P2P证书脚本

create_cert.sh

创建P2P证书脚本`create_cert.sh`用于创建P2P证书文件。WeCross Router之间通讯需要证书用于认证，只有具有相同`ca.crt`根证书的WeCross Router直接才能建立连接。

可通过-h查看帮助信息：

```
Usage:
    -c                                  [Optional] generate ca certificate
    -C <number>                         [Optional] the number of node certificate␣
↪generated, work with '-n' opt, default: 1
    -D <dir>                            [Optional] the ca certificate directory,␣
↪work with '-n', default: './'
    -d <dir>                            [Required] generated target_directory
    -n                                  [Optional] generate node certificate
    -t                                  [Optional] cert.cnf path, default: cert.cnf
    -h                                  [Optional] Help
e.g
    bash create_cert.sh -c -d ./ca
    bash create_cert.sh -n -D ./ca -d ./ca/node
    bash create_cert.sh -n -D ./ca -d ./ca/node -C 10
```

- **c**： 生成ca证书，只有生成了ca证书，才能生成节点证书。
- **C**：配合-n，指定生成节点证书的数量。
- **D**：配合-n，指定ca证书路径。
- **d**：指定输出目录。
- **n**：生成节点证书。
- **t**：指定`cert.cnf`的路径

CHAPTER 5

跨链接入

5.1 接入 FISCO BCOS 2.0

WeCross BCOS2 Stub 是 WeCross Router的插件，让Router具备接入FISCO-BCOS 2.0的能力。关于该插件包含下列方面内容：

- 跨链合约
- 插件安装
- 账户配置
- 插件配置
- 代理合约配置

重要： FISCO-BCOS版本需要 >= v2.4.0

5.1.1 跨链合约

BCOS2 Stub支持任意类型接口的`solidity`合约。

`HelloWorld`合约示例：

```
pragma solidity ^0.4.24;
pragma experimental ABIEncoderV2;

contract HelloWorld {
    string s = "Hello World!";

    function set(string memory _s) public returns (string memory) {
        s = _s;
        return s;
    }

    function get() public constant returns(string memory) {
        return s;
```

(continues on next page)

(续上页)

```
    }
}
```

5.1.2 插件安装

在生成router时，默认安装FISCO-BCOS stub插件，安装目录为router下的plugin目录：

```
plugin/
|-- bcos2-stub-gm-xxxx.jar     # 国密插件
|-- bcos2-stub-xxxx.jar        # 非国密插件
└-- fabric-stub.jar
```

用户如有特殊需求，可以自行编译，替换plugin目录下的插件。

手动安装

下载编译

```
git clone https://github.com/WeBankFinTech/WeCross-BCOS2-Stub.git
cd WeCross-BCOS2-Stub
bash gradlew build -x test
```

WeCross-BCOS2-Stub编译生成两个插件

- 国密插件
- 非国密插件

```
dist/apps
├── bcos2-stub-gm-xxxx.jar     # 国密插件
└── bcos2-stub-xxxx.jar        # 非国密插件
```

安装插件

```
cp dist/apps/* ~/wecross/routers-payment/127.0.0.1-8250-25500/plugin/
```

注：若router中配置了两个相同的插件，插件冲突，会导致router启动失败。

WeCross快速部署

5.1.3 账户配置

WeCross中账户用于交易签名，BCOS2 Stub支持pem和p12两种格式文件。

配置路径

WeCross Router账户配置位于conf/accounts/目录。每个账户使用单独目录配置，使用账户名称作为子目录名称，每个目录包含account.toml配置文件以及配置文件中配置的私钥文件，私钥文件可以为pem或者p12格式。

```
// 示例
conf/accounts/
|-- bcos_pem
|   |-- 0x5399e9ca7b444afb537a7a9de2762d17c3c7f63a.pem
|   `-- account.toml
└-- bcos_p12
```

(continues on next page)

(续上页)

```
    |-- 0x0ed9d10e1520a502a41115a4fc8b6e3edb201940.p12
    └-- account.toml
```

上面的示例中配置了两个账户，账户名称分别为: bcos_pem、bcos_p12

配置格式

```
// account.toml
[account]
    type = 'BCOS2.0'
    accountFile = '0x15469c4af049c6441f6ef3f8d22d44547031ebea.p12'
    password = '123456'
```

- type: 账户类型，GM_BCOS2.0或者BCOS2.0，GM_BCOS2.0表示国密账户，BCOS2.0表示非国密账户
- accountFile: 私钥文件
- password: p12文件密码，pem文件时忽略

配置步骤

在router目录下，用add_account.sh直接生成账户即可，无需其他手动配置。

```
bash add_account.sh -h

Usage:
    -t <type>                               [Required] type of account, BCOS2.0 or␣
↪Fabric1.4
    -n <name>                               [Required] name of account
    -d <dir>                                [Optional] generated target_directory,␣
↪default conf/accounts/
    -h                                      [Optional] Help
e.g
    bash add_account.sh -t BCOS2.0 -n my_bcos_account
    bash add_account.sh -t Fabric1.4 -n my_fabric_account
```

示例:

```
cd ~/wecross/routers-payment/127.0.0.1-8250-25500/

# 举例1：生成非国密账户,-t 指定使用BCOS2.0插件（非国密插件） -n 设置一个账户名
bash add_account.sh -t BCOS2.0 -n bcos_user1

# 举例2：生成国密账户,-t 指定使用GM_BCOS2.0插件（国密插件） -n 设置一个账户名
bash add_account.sh -t GM_BCOS2.0 -n bcos_gm_user1
```

账户生成至conf/accounts目录下

```
conf/accounts/
├── bcos_gm_user1
│   ├── account.key
│   └── account.toml
└── bcos_normal_user1
    ├── account.key
    └── account.toml
```

5.1.4 接入链配置

stub插件的配置文件为stub.toml，作用：

- 配置资源信息
- 配置SDK连接信息，与链进行交互

配置路径

```
conf/chains/bcos/
|-- ca.crt
|-- sdk.crt
|-- sdk.key
└-- stub.toml
```

配置格式

stub插件的配置文件stub.toml格式以及字段含义

```
[common]                    # 通用配置
    name = 'bcos'           # stub配置名称
    type = 'BCOS2.0'        # stub类型，`GM_BCOS2.0`或者`BCOS2.0`，`GM_BCOS2.0`国密类
型，`BCOS2.0`非国密类型

[chain]                     # FISCO-BCOS 链配置
    groupId = 1             # 连接FISCO-BCOS群组id，默认为1
    chainId = 1             # 连接FISCO-BCOS链id，默认为1

[channelService]            # FISCO-BCOS 配置
    caCert = 'ca.crt'       # 根证书
    sslCert = 'sdk.crt' # sdk证书
    sslKey = 'sdk.key'  # sdk私钥
    timeout = 5000          # SDK请求超时时间
    connectionsStr = ['127.0.0.1:20200','127.0.0.2:20200']     # 连接列表

# [[resources]] 资源列表
[[resources]]
    name = 'htlc'               # 资源名称
    type = 'BCOS_CONTRACT'  # 资源类型，BCOS_CONTRACT
    contractAddress = '0x7540601cce8b0802980f9ebf7aeee22bb4d73c22'   # 合约地址
```

重要:

- BCOS2 Stub当前只支持合约类型的资源

5.1.5 代理合约配置

配置完成后，在router目录下执行命令，部署代理合约，分为国密与非国密的部署。

非国密

```
Usage:
        java -cp 'conf/:lib/*:plugin/*' com.webank.wecross.stub.bcos.normal.proxy.
↪ProxyContractDeployment check [chainName]
```

(continues on next page)

(续上页)

```
        java -cp 'conf/:lib/*:plugin/*' com.webank.wecross.stub.bcos.normal.proxy.
↪ProxyContractDeployment deploy [chainName] [accountName]
        java -cp 'conf/:lib/*:plugin/*' com.webank.wecross.stub.bcos.normal.proxy.
↪ProxyContractDeployment upgrade [chainName] [accountName]
Example:
        java -cp 'conf/:lib/*:plugin/*' com.webank.wecross.stub.bcos.normal.proxy.
↪ProxyContractDeployment check chains/bcos
        java -cp 'conf/:lib/*:plugin/*' com.webank.wecross.stub.bcos.normal.proxy.
↪ProxyContractDeployment deploy chains/bcos bcos_user1
        java -cp 'conf/:lib/*:plugin/*' com.webank.wecross.stub.bcos.normal.proxy.
↪ProxyContractDeployment upgrade chains/bcos bcos_user1
```

参数:

- check: 检查代理合约是否部署
- deploy: 部署代理合约子命令
- upgrade: 更新大代理合约子命令，表示重新部署代理合约
- chainName: 链名称
- accountName: 发送交易的账户

国密

```
Usage:
        java -cp 'conf/:lib/*:plugin/*' com.webank.wecross.stub.bcos.guomi.proxy.
↪ProxyContractDeployment check [chainName]
        java -cp 'conf/:lib/*:plugin/*' com.webank.wecross.stub.bcos.guomi.proxy.
↪ProxyContractDeployment deploy [chainName] [accountName]
        java -cp 'conf/:lib/*:plugin/*' com.webank.wecross.stub.bcos.guomi.proxy.
↪ProxyContractDeployment upgrade [chainName] [accountName]
Example:
        java -cp 'conf/:lib/*:plugin/*' com.webank.wecross.stub.bcos.guomi.proxy.
↪ProxyContractDeployment check chains/bcos
        java -cp 'conf/:lib/*:plugin/*' com.webank.wecross.stub.bcos.guomi.proxy.
↪ProxyContractDeployment deploy chains/bcos bcos_user1
        java -cp 'conf/:lib/*:plugin/*' com.webank.wecross.stub.bcos.guomi.proxy.
↪ProxyContractDeployment upgrade chains/bcos bcos_user1
```

参数：同上

5.1.6 参考链接

WeCross-BCOS2-Stub

FISCO BCOS 环境搭建参考

FISCO-BCOS JavaSDK文档

FISCO-BCOS Console文档

5.2 接入 Fabric 1.4

WeCross Fabric Stub 是 WeCross Router的插件，让Router具备接入Fabric 1.4的链的能力。其要点包括:

- 插件安装
- 接入配置：用于接入相应的Fabric 1.4链

- 账户配置：用于用相应账户发交易

5.2.1 插件安装

在生成router时，默认安装Fabric 1.4插件，安装目录为router下的plugin目录：

```
cd ~/wecross/routers-payment/127.0.0.1-8251-25501/
tree plugin/
plugin/
└── fabric1-stub-XXXXX.jar
```

用户如有特殊需求，可以自行编译，替换plugin目录下的插件。

手动安装

下载编译

```
git clone https://github.com/WeBankFinTech/WeCross-Fabric1-Stub.git
cd WeCross-Fabric1-Stub
bash gradlew assemble # 在 dist/apps/下生成fabric1-stub-XXXXX.jar
```

安装插件

```
cp dist/apps/fabric1-stub-XXXXX.jar ~/wecross/routers-payment/127.0.0.1-8250-25500/
↪plugin/
```

注：若router中配置了两个相同的插件，插件冲突，会导致router启动失败。

5.2.2 账户配置

在router中配置Fabric账户，用户可在sdk中指定router用相应的账号发交易。

完整配置

配置完成的账户如下，在accounts目录中

```
accounts/                                                    # router的账户目录，所有账户的
文件夹放在此目录下
└── fabric_admin                                          # 目录名即为此账户名，SDK发交易时，指定的即
为处的账户名
    ├── account.toml                                  # 账户配置文件
    ├── user.crt                                        # Fabric的用户证书
    └── user.key                                        # Fabric的用户私钥
```

其中 account.toml为账户配置文件

```
[account]
    type = 'Fabric1.4'                                  # 采用插件的名字
    mspid = 'Org1MSP'                                 # 账户对应机构的MSP ID
    keystore = 'user.key'                           # 账户私钥文件名字，指向与此文件相同目录下的私钥文
件
    signcert = 'user.crt'                           # 账户证书名字，指向与此文件相同目录下的证书文件
```

配置步骤

生成配置文件

为router生成某个账户的配置，在router目录下执行

```
cd ~/wecross/routers-payment/127.0.0.1-8251-25501/

# 举例1：生成名字为fabric_admin的账户配置 -t 指定使用Fabric1.4插件生成 -n 设置一个账户名
bash add_account.sh -t Fabric1.4 -n fabric_admin

# 举例2：生成名字为fabric_user1的账户配置 -t 指定使用Fabric1.4插件生成 -n 设置一个账户名
bash add_account.sh -t Fabric1.4 -n fabric_user1
```

生成后，conf/accounts目录下出现对应名字的账户目录，接下来需将相关账户文件拷贝入目录中。

拷贝账户文件

以fabric-sample/first-network的crypto-config为例，配置其中一个账户即可

- 若配置 Org1 的 Admin

```
# 拷贝用户私钥，命名为 user.key
cp ~/demo/fabric/fabric-sample/first-network/crypto-config/peerOrganizations/org1.
↪example.com/users/Admin@org1.example.com/msp/keystore/*_sk accounts/fabric_admin/
↪user.key
# 拷贝用户证书，命名为 user.crt
cp ~/demo/fabric/fabric-sample/first-network/crypto-config/peerOrganizations/org1.
↪example.com/users/Admin@org1.example.com/msp/signcerts/*.pem accounts/fabric_
↪admin/user.crt
```

- 若配置 Org1的 User1

```
# 拷贝用户私钥，命名为 user.key
cp ~/demo/fabric/fabric-sample/first-network/crypto-config/peerOrganizations/org1.
↪example.com/users/User1@org1.example.com/msp/keystore/*_sk accounts/fabric_user1/
↪user.key
# 拷贝用户证书，命名为 user.crt
cp ~/demo/fabric/fabric-sample/first-network/crypto-config/peerOrganizations/org1.
↪example.com/users/User1@org1.example.com/msp/signcerts/*.pem accounts/fabric_
↪user1/user.crt
```

编辑配置文件

编辑 account.toml

```
vim conf/accounts/<account_name>/account.toml
```

内容为

```
[account]
    type = 'Fabric1.4'                          # 采用插件的名字
    mspid = 'Org1MSP'                          # 账户对应机构的MSP ID
    keystore = 'user.key'                  # 账户私钥文件名字，指向与此文件相同目录下的私钥文
件
    signcert = 'user.crt'                  # 账户证书名字，指向与此文件相同目录下的证书文件
```

5.2.3 接入链配置

在router中配置需接入的链，访问链上资源。

完整配置

配置完成如下，在chains目录中

```
chains                                                    # router的stub的配置目
录，所有的stub都在此目录下配置
└── fabric                                                # 此链的名字，名字可任意指定，
与链类型无关
    ├── orderer-tlsca.crt                 # orderer证书
    ├── org1-tlsca.crt                        # 需要连接Org1的endorser的证书1，有则配
    ├── org2-tlsca.crt                        # 需要连接Org2的endorser的证书2，有则配
    └── stub.toml                                # stub配置文件
```

其中，stub.toml 为接入的链的配置文件

```
[common]
    name = 'fabric'
    type = 'Fabric1.4'

[fabricServices]
    channelName = 'mychannel'
    orgUserName = 'fabric_admin'
    orgUserAccountPath = 'classpath:accounts/fabric_admin'
    ordererTlsCaFile = 'orderer-tlsca.crt'
    ordererAddress = 'grpcs://localhost:7050'

[orgs]
    [orgs.Org1]
         tlsCaFile = 'org1-tlsca.crt'
         adminName = 'fabric_admin_org1'
         endorsers = ['grpcs://localhost:7051']

    [orgs.Org2]
         tlsCaFile = 'org2-tlsca.crt'
         adminName = 'fabric_admin_org2'
         endorsers = ['grpcs://localhost:9051']
```

配置步骤

生成配置文件

```
cd ~/wecross/routers-payment/127.0.0.1-8251-25501
bash add_chain.sh -t Fabric1.4 -n fabric # -t 链类型，-n 指定链名字

# 查看生成目录
tree conf/chains/fabric
```

生成的目录结构如下：

```
conf/chains/fabric
└── stub.toml           # chain配置文件
```

拷贝链证书

以fabric-sample/first-network的crypto-config为例

```
# 拷贝 orderer证书
cp ~/demo/fabric/fabric-sample/first-network/crypto-config/ordererOrganizations/
↪example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.
↪pem conf/chains/fabric/orderer-tlsca.crt
# 拷贝 Org1 证书
```

(continues on next page)

(续上页)

```
cp ~/demo/fabric/fabric-sample/first-network/crypto-config/peerOrganizations/org1.
↪example.com/peers/peer0.org1.example.com/tls/ca.crt conf/chains/fabric/org1-
↪tlsca.crt
# 拷贝 Org2 证书
cp ~/demo/fabric/fabric-sample/first-network/crypto-config/peerOrganizations/org2.
↪example.com/peers/peer0.org2.example.com/tls/ca.crt conf/chains/fabric/org2-
↪tlsca.crt
```

编辑配置文件

```
vim conf/chains/fabric/stub.toml
```

- 基础配置

```
[common]
    name = 'fabric'                                    # 指定的连接的链的名字，对应path中的
↪{zone}/{chain}/{resource}的chain
    type = 'Fabric1.4'                              # 采用插件的名字
```

- 配置链

```
[fabricServices]
    channelName = 'mychannel'
    orgUserName = 'fabric_admin' # 指定一个机构的admin账户，用于与orderer通信
    ordererTlsCaFile = 'orderer-tlsca.crt' # orderer证书名字，指向与此配置文件相同目录下的
证书
    ordererAddress = 'grpcs://localhost:7050' # orderer的url

[orgs] # 机构节点列表
    [orgs.Org1] # 机构1：Org1
         tlsCaFile = 'org1-tlsca.crt' # Org1的证书
         adminName = 'fabric_admin_org1' # Org1的admin账户，在下一步骤中配置
         endorsers = ['grpcs://localhost:7051'] # endorser的ip:port列表，可配置多个

    [orgs.Org2] # 机构2：Org2
         tlsCaFile = 'org2-tlsca.crt' # Org2的证书
         adminName = 'fabric_admin_org2' # Org2的admin账户，在下一步骤中配置
         endorsers = ['grpcs://localhost:9051'] # endorser的ip:port列表，可配置多个
```

配相关账户

stub.toml中涉及三个账户，采用账户配置给出的步骤配置入conf/accounts目录即可：

- fabric_admin：fabricServices的admin账户，用于与orderer通信，此处选择一个Org的admin账户作为此账户即可
- fabric_admin_org1：与Org1的endorser通信的账户，需配置Org1的admin
- fabric_admin_org2：与Org2的endorser通信的账户，需配置Org2的admin，**注意生成账户后需手动将默认的mspid改为 Org2MSP**

部署代理合约

配置完成后，在router目录下执行命令，部署代理合约

```
java -cp 'conf/:lib/*:plugin/*' com.webank.wecross.stub.fabric.proxy.
↪ProxyChaincodeDeployment deploy chains/fabric # deploy conf下的链配置位置
```

部署成功

```
SUCCESS: WeCrossProxy has been deployed to chains/fabric
```

代理合约部署完成后，即可启动router，接入链配置完成

5.2.4 部署跨链资源

用户可通过WeCross控制台部署和升级链码，相关操作见控制台说明部分：

- fabricInstall
- fabricInstantiate
- fabricUpgrade

CHAPTER 6

开发手册

6.1 跨链SDK开发应用

WeCross router向外部暴露了所有的UBI接口，开发者可以通过SDK实现这些接口的快速调用。

6.1.1 环境要求

重要:

- java版本

  要求 JDK8或以上

- WeCross服务部署

  参考 部署指南

6.1.2 Java应用引入SDK

通过gradle或maven引入SDK到java应用

gradle:

```
compile ('com.webank:wecross-java-sdk:1.0.0-rc3')
```

maven:

```
<dependency>
    <groupId>com.webank</groupId>
    <artifactId>wecross-java-sdk</artifactId>
    <version>1.0.0-rc3</version>
</dependency>
```

6.1.3 使用方法

示例代码如下：

```
try {
    // 初始化 Service
    WeCrossRPCService weCrossRPCService = new WeCrossRPCService();

    // 初始化Resource
    WeCrossRPC weCrossRPC = WeCrossRPCFactory.build(weCrossRPCService);
    Resource resource = ResourceFactory.build(weCrossRPC, "payment.bcos.
↪HelloWecross", "bcos_user1"); // RPC服务，资源的path，用哪个账户名操作此resource

    // 用初始化好的resource进行调用
    String[] callRet = resource.call("get");    // call 接口函数名 参数列表
    System.out.println((Arrays.toString(callRet)));

    // 用初始化好的resource进行调用
    String[] sendTransactionRet = resource.sendTransaction("set", "Tom"); //␣
↪sendTransaction 接口函数名 参数列表
    System.out.println((Arrays.toString(sendTransactionRet)));

} catch (WeCrossSDKException e) {
    System.out.println("Error: " + e);
}
```

6.2 WeCross Java SDK API

SDK API分为两大类型，一种是RPC接口，一种是资源接口，其中资源接口是对RPC接口进行了封装。

6.2.1 API列表

- RPC接口

  RemoteCall supportedStubs();

  RemoteCall listAccounts();

  RemoteCall listResources(Boolean ignoreRemote);

  RemoteCall detail(String path);

  RemoteCall call(Request request);

  RemoteCall call(String path, String account, String method, String... args);

  RemoteCall sendTransaction(Request request);

  RemoteCall sendTransaction(String path, String account, String method, String... args);

  RemoteCall callTransaction(String transactionID, String path, String account, String method, String... args);

  RemoteCall execTransaction(String transactionID,String seq, String path, String account, String method, String... args);

  RemoteCall startTransaction(String transactionID, String[] accounts, String[] paths);

  RemoteCall commitTransaction(String transactionID, String[] accounts, String[] paths);

  RemoteCall rollbackTransaction(String transactionID, String[] accounts, String[] paths);

  RemoteCall getTransactionInfo(String transactionID, String[] accounts, String[] paths);

RemoteCall customCommand(String command, String path, String account, Object... args);

RemoteCall getTransactionIDs(String path, String account, int option);

- 资源接口

Resource ResourceFactory.build(WeCrossRPC weCrossRPC, String path, String account)

boolean isActive();

ResourceDetail detail();

TransactionResponse call(Request request);

String[] call(String method);

String[] call(String method, String... args);

TransactionResponse sendTransaction(Request request);

String[] sendTransaction(String method);

String[] sendTransaction(String method, String... args);

6.2.2 RPC接口解析

supportedStubs

显示router当前支持的插件列表。

参数

- 无

返回值

- `StubResponse` - 响应包
  - `version`: `String` - 版本号
  - `errorCode`: `int` - 状态码
  - `message`: `String` - 错误消息
  - `data`: `Stubs` - 支持的插件列表

java示例

```
// 初始化RPC实例
WeCrossRPCService weCrossRPCService = new WeCrossRPCService();
WeCrossRPC weCrossRPC = WeCrossRPCFactory.build(weCrossRPCService);

// 调用RPC接口，目前只支持同步调用
StubResponse response = weCrossRPC.supportedStubs().send();
```

注 - 之后的java示例，会省去初始化WeCrossRPC的步骤。

listAccounts

显示所有已配置的账户列表。

参数

- 无

返回值

- `AccountResponse` - 响应包
  - `version`: `String` - 版本号
  - `errorCode`: `int` - 状态码
  - `message`: `String` - 错误消息
  - `data`: `Accounts` - 配置的账户列表

java示例

```
    AccountResponse response = weCrossRPC.listAccounts().send();
```

listResources

显示router配置的跨链资源。

参数

- `ignoreRemote`: `Boolean` - 是否忽略远程资源

返回值

- `ResourceResponse` - 响应包
  - `version`: `String` - 版本号
  - `errorCode`: `int` - 状态码
  - `message`: `String` - 错误消息
  - `data`: `Resources` - 配置的资源列表

java示例

```
    ResourceResponse response = weCrossRPC.listResources(true).send();
```

detail

获取资源详情。

参数

- `path`: `String` - 跨链资源标识

返回值

- ResourceDetailResponse - 响应包
  - version: String - 版本号
  - errorCode: int - 状态码
  - message: String - 错误消息
  - data: ResourceDetail - 资源详情

java示例

```
    ResourceDetailResponse response = weCrossRPC.detail("payment.bcos.HelloWeCross
↪").send();
```

call(无参数)

调用智能合约，不更改链状态，不发交易。

参数

- path: String - 跨链资源标识
- account: String - 账户名
- method: String - 调用的方法

返回值

- TransactionResponse - 响应包
  - version: String - 版本号
  - errorCode: int - 状态码
  - message: String - 错误消息
  - data: Receipt - 调用结果

java示例

```
    TransactionResponse transactionResponse =
        weCrossRPC
            .call(
                "payment.bcos.HelloWeCross","get")
            .send();
```

call(带参数)

调用智能合约，不更改链状态，不发交易。

参数

- path: String - 跨链资源标识
- account: String - 账户名
- method: String - 调用的方法
- args : String... - 可变参数列表

返回值

- TransactionResponse - 响应包
  - version: String - 版本号
  - errorCode: int - 状态码
  - message: String - 错误消息
  - data: Receipt - 调用结果

java示例

```
    TransactionResponse transactionResponse =
        weCrossRPC
            .call(
                "payment.bcos.HelloWeCross","get","key")
            .send();
```

sendTransaction(无参数)

调用智能合约，会改变链状态，发交易。

参数

- path: String - 跨链资源标识
- account: String - 账户名
- method: String - 调用的方法

返回值

- TransactionResponse - 响应包
  - version: String - 版本号
  - errorCode: int - 状态码
  - message: String - 错误消息
  - data: Receipt - 调用结果

java示例

```
    TransactionResponse transactionResponse =
        weCrossRPC
            .sendTransaction(
                "payment.bcos.HelloWeCross","set")
            .send();
```

sendTransaction(带参数)

调用智能合约，会改变链状态，发交易。

参数

- path: String - 跨链资源标识
- account: String - 账户名
- method: String - 调用的方法
- args : String... - 可变参数列表

返回值

- TransactionResponse - 响应包
  - version: String - 版本号
  - errorCode: int - 状态码
  - message: String - 错误消息
  - data: Receipt - 调用结果

java示例

```
    TransactionResponse transactionResponse =
        weCrossRPC
            .sendTransaction(
                "payment.bcos.HelloWeCross","set","value")
            .send();
```

callTransaction

获取事务中的状态数据，不发交易

参数

- transactionID:String - 事务ID
- path: String - 跨链资源标识
- account: String - 账户名
- method: String - 调用的方法
- args : String... - 可变参数列表

返回值

- TransactionResponse - 响应包
  - version: String - 版本号
  - errorCode: int - 状态码
  - message: String - 错误消息
  - data: Receipt - 调用结果

java示例

```
TransactionResponse transactionResponse =
    weCrossRPC
        .callTransaction(
            "0001","payment.bcos.2pc","queryEvidence","key1")
        .send();
```

execTransaction

执行事务，发交易

参数

- transactionID:String - 事务ID
- seq:String - 事务序列号，需递增
- path: String - 跨链资源标识
- account: String - 账户名
- method: String - 调用的方法
- args : String... - 可变参数列表

返回值

- TransactionResponse - 响应包
  - version: String - 版本号
  - errorCode: int - 状态码
  - message: String - 错误消息
  - data: Receipt - 调用结果

java示例

```
TransactionResponse transactionResponse =
    weCrossRPC
        .execTransaction(
            "0001","1","payment.bcos.2pc","newEvidence","key1","evidence1")
        .send();
```

startTransaction

开始事务，锁定事务相关资源，发交易

参数

- transactionID:String - 事务ID
- accounts:String[] - 账户列表，每条链需要一个发交易的账户
- paths: String[] - 跨链资源列表

返回值

- RoutineResponse - 响应包
  - version: String - 版本号
  - errorCode: int - 状态码
  - message: String - 错误消息
  - data: int - 调用结果

java示例

```
    RoutineResponse routineResponse =
        weCrossRPC
            .startTransaction(
                "0001", new String[]{"bcos_user1","fabric_user1"}, new String[]{
↪"payment.bcos.2pc", "payment.fabric.2pc"},)
            .send();
```

commitTransaction

提交事务，释放事务相关资源，发交易

参数

- transactionID:String - 事务ID
- accounts:String[] - 账户列表，每条链需要一个发交易的账户
- paths: String[] - 跨链资源列表

返回值

- RoutineResponse - 响应包
  - version: String - 版本号
  - errorCode: int - 状态码
  - message: String - 错误消息
  - data: int - 调用结果

java示例

```
    RoutineResponse routineResponse =
        weCrossRPC
            .commitTransaction(
                "0001", new String[]{"bcos_user1","fabric_user1"}, new String[]{
↪"payment.bcos.2pc", "payment.fabric.2pc"},)
            .send();
```

rollbackTransaction

回滚事务，释放事务相关资源，发交易

参数

- transactionID:String - 事务ID
- accounts:String[] - 账户列表，每条链需要一个发交易的账户
- paths: String[] - 跨链资源列表

返回值

- RoutineResponse - 响应包
  - version: String - 版本号
  - errorCode: int - 状态码
  - message: String - 错误消息
  - data: int - 调用结果

java示例

```
    RoutineResponse routineResponse =
        weCrossRPC
            .rollbackTransaction(
                "0001", new String[]{"bcos_user1","fabric_user1"}, new String[]{
↪"payment.bcos.2pc", "payment.fabric.2pc"},)
            .send();
```

getTransactionInfo

获取事务详情，不发交易

参数

- transactionID:String - 事务ID
- accounts:String[] - 账户列表，每条链需要一个发交易的账户
- paths: String[] - 跨链资源列表

返回值

- RoutineInfoResponse - 响应包
  - version: String - 版本号
  - errorCode: int - 状态码
  - message: String - 错误消息
  - data: String - 事务信息

java示例

```
    RoutineInfoResponse routineInfoResponse =
        weCrossRPC
            .getTransactionInfo(
                "0001", new String[]{"bcos_user1","fabric_user1"}, new String[]{
↪"payment.bcos.2pc", "payment.fabric.2pc"},)
            .send();
```

getTransactionIDs

获取事务ID列表，不发交易

参数

- path: String - 跨链资源标识
- account: String - 账户名
- option: int - 选项，0全部事务，1已完成的事务，2未完成的事务
-

返回值

- RoutineIDResponse - 响应包
  - version: String - 版本号
  - errorCode: int - 状态码
  - message: String - 错误消息
  - data: String[] - 事务ID列表

java示例

```
    RoutineIDResponse routineIDResponse =
        weCrossRPC
            .getTransactionIDs(
                "payment.bcos.2pc", "bcos_user1", 0)
            .send();
```

customCommand

自定义命令

参数

- command: String - 命令名称
- path: String - 跨链资源标识
- account: String - 账户名
- args: Object... - 可变参数
-

返回值

- CommandResponse - 响应包
  - version: String - 版本号
  - errorCode: int - 状态码
  - message: String - 错误消息
  - data: String - 调用结果

java示例

```
CommandResponse commandResponse =
    weCrossRPC
        .customCommand(
            "deploy", "payment.bcos.2pc", "bcos_user1")
        .send();
```

6.2.3 资源接口解析

ResourceFactory.build

初始化一个跨链资源

参数

- weCrossRPC : WeCrossRPC - RPC实例
- path: String - 跨链资源标识
- account: String - 账户名

返回值

- Resource - 跨链资源实例

java示例

```
    // 初始化RPC实例
    WeCrossRPCService weCrossRPCService = new WeCrossRPCService();
    WeCrossRPC weCrossRPC = WeCrossRPCFactory.build(weCrossRPCService);

    // 初始化资源实例
    Resource resource = ResourceFactory.build(weCrossRPC, path, account);
```

注 - 之后的java示例，会省去初始化Resource的步骤。

isActive

获取资源状态，`true`:可达，`false`:不可达。

参数

- 无

返回值

- `bool` - 资源状态

java示例

```
    bool status = resource.isActive();
```

detail

获取资源详情。

参数

- 无

返回值

- `ResourceDetail` - 资源详情

java示例

```
    ResourceDetail detail = resource.detail();
```

call

调用智能合约，不更改链状态，不发交易。

参数

- request: Request<TransactionRequest> - 请求体

返回值

- TransactionResponse - 响应包
  - version: String - 版本号
  - errorCode: int - 状态码
  - message: String - 错误消息
  - data: Receipt - 调用结果

java示例

```
    TransactionResponse transactionResponse = resource.call(request);
```

call(无参数)

调用智能合约，不更改链状态，不发交易。

参数

- method: String - 调用的方法

返回值

- String[] - 调用结果

java示例

```
    String[] result = resource.call("get");
```

call(带参数)

调用智能合约，不更改链状态，不发交易。

参数

- method: String - 调用的方法
- args: String... - 可变参数列表

返回值

- String[] - 调用结果

java示例

```
    String[] result = resource.call("get", "key");
```

sendTransaction

调用智能合约，会改变链状态，发交易。

参数

- request: Request<TransactionRequest> - 请求体

返回值

- TransactionResponse - 响应包
  - version: String - 版本号
  - errorCode: int - 状态码
  - message: String - 错误消息
  - data: Receipt - 调用结果

java示例

```
    TransactionResponse transactionResponse = resource.sendTransaction(request);
```

sendTransaction(无参数)

调用智能合约，会改变链状态，发交易。

参数

- method: String - 调用的方法

返回值

- String[] - 调用结果

java示例

```
    String[] result = resource.sendTransaction("set");
```

sendTransaction(带参数)

调用智能合约，会改变链状态，发交易。

参数

- `method`: `String` - 调用的方法
- `args`: `String...` - 可变参数列表

返回值

- `String[]` - 调用结果

java示例

```
    String[] result = resource.sendTransaction("set", "value");
```

6.2.4 RPC状态码

当一个RPC调用遇到错误时，返回的响应对象必须包含error错误结果字段，该字段有下列成员参数：

- errorCode: 使用数值表示该异常的错误类型，必须为整数。
- message: 对该错误的简单描述字符串。

标准状态码及其对应的含义如下：

6.3 WeCross Stub插件开发

本章内容介绍区块链接入WeCross的完整开发流程，用户可以根据本教程实现一个WeCross区块链`Stub`插件，通过该插件接入WeCross。

注解:

- Java编程语言

  WeCross使用Java实现，接入的区块链需要支持Java版本的SDK，要求开发人员具备Java开发能力。

- Gradle构建工具

  WeCross组件目前使用Gradle进行构建，因此假定用户能够使用Gradle，maven能达到相同的效果，用户可以自行探索。

6.3.1 创建Gradle项目

Gradle依赖分类

- Stub API定义
- Java SDK，区块链Java版的SDK
- 其他依赖，例如：toml解析工具

```
// Gradle 依赖示例
dependencies {
    // Stub接口定义Jar
    implementation 'com.webank:wecross-java-stub:1.0.0-rc4'
    // BCOS JavaSDK
    implementation 'org.fisco-bcos:web3sdk:2.4.0'
    // toml文件解析
    implementation
    'com.moandjiezana.toml:toml4j:0.7.2'
    // 其他依赖
}
```

新建Gradle工程，并且在`build.gradle`中添加依赖。

Gradle配置参考: WeCross-BCOS-Stub build.gradle

6.3.2 核心组件

`Stub`插件需要实现的组件接口：

- StubFactory: 创建其他组件
- Account: 账户，用于交易签名
- Connection: 与区块链交互
- Driver: 区块链数据的编解码

StubFactory

StubFactory功能

- 添加`@Stub`注解，指定插件类型
- 提供Account、Connection、Driver实例化入口

只有添加@Stub注解的插件才能被Wecross Router识别加载!

`StubFactory`接口定义

```
public interface StubFactory {

    /** 创建Driver对象 */
    public Driver newDriver();

    /** 解析Connection配置stub.toml，创建Connection对象 */
    public Connection newConnection(String path);

    /** 解析Account配置account.toml，创建Account对象 */
    public Account newAccount(String name, String path);
}
```

BCOS Stub示例:

```
/** @Stub注解，插件类型：BCOS2.0 */
@Stub("BCOS2.0")
public class BCOSStubFactory implements StubFactory {

    @Override
    public Driver newDriver() {
        // 创建返回BCOSDriver对象
    }
```

(continues on next page)

(续上页)

```
    @Override
    public Connection newConnection(String path) {
        // 解析stub.toml配置, 创建返回BCOSConnection对象
    }

    @Override
    public Account newAccount(String name, String path) {
        // 解析account.toml账户配置，创建返回BCOSAccount对象
    }
}
```

Account

Account包含账户私钥，用于交易签名。

接口定义

```
public interface Account {

    /** 账户名称 */
    String getName();

    /** 账户类型 */
    String getType();

    /** 账户公钥 */
    String getIdentity();
}
```

Account由StubFactory对象newAccount接口创建，用户需要解析配置生成Account对象

- Account newAccount(String name, String path)
  - name: 账户名称
  - path: 配置文件account.toml所在目录路径

账户配置文件位于conf/accounts/目录，可以配置多个账户，每个账户置于单独的子目录。

```
# 目录结构, conf/accounts/账户名称/
conf/accounts
        └── bcos # 账户名称: bcos
            ├── 0x4c9e341a015ce8200060a028ce45dfea8bf33e15.pem # BCOS私钥文件
            └── account.toml # account.toml配置文件
```

```
# account.toml内容
[account]
    type = "BCOS2.0" # 必须项，账户类型，与插件@Stub注解定义的类型保持一致
    accountFile = '0x4c9e341a015ce8200060a028ce45dfea8bf33e15.pem' # 配置的私钥文件名
称
```

account.toml解析流程可以参考BCOS Stub account.toml解析

BCOS Stub示例

```
public class BCOSAccount implements Account {

    /** 名称 */
    private final String name;
```

(continues on next page)

(续上页)

```
    /** 类型 */
    private final String type;

    /** 公钥 */
    private final String publicKey;

    /** BCOS私钥对象，交易签名，加载配置的私钥文件生成 */
    private final Credentials credentials;

    public Credentials getCredentials() {
        return credentials;
    }
    /** 其他接口 */
}
```

Connection

Connection用于

- 为Driver提供统一发送接口，与区块链交互
- 获取配置的资源列表，资源在Connection初始化时从配置文件加载

接口定义

```
public interface Connection {

    /** 发送接口请求给区块链 */
    Response send(Request request);

    /** 获取资源列表，资源列表在stub.toml文件配置 */
    List<ResourceInfo> getResources();
}
```

接口列表

- getResources

获取资源列表，资源在Connection初始化时加载，表示Stub可以访问的区块链资源。

- send

提供统一的发送请求接口给Driver使用，Driver设置请求类型并将参数序列化生成Request，调用send接口，返回Response对象，Response包含返回值、错误信息以及返回内容。

```
/** Request请求对象，包含请求类型与请求内容 */
public class Request {

    /** 请求类型，用户自定义，区分不同的区块链请求 */
    private int type;

    /** 请求内容，序列化的请求参数，用户自定义序列化方式 */

    private byte[] data;

    /** 请求资源 */
    private ResourceInfo resourceInfo;
}

/** Response返回值对象，包含错误码、错误描述以及返回内容 */
public class Response {
```

(continues on next page)

(续上页)

```
    /** 返回状态 */
    private int errorCode;

    /** 返回错误描述 */
    private String errorMessage;

    /** 返回内容，序列化的返回参数，可以反序列化为返回对象 */
    private byte[] data;
}

/** 资源对象 */
public class ResourceInfo {

    /** 资源名称 */
    private String name;

    /** 资源类型，用户自定义 */
    private String stubType;

    /** 额外属性，用户自定义 */
    private Map<Object, Object> properties = new HashMap<Object, Object>();
}
```

Connection由StubFactory对象`newConnection`接口创建，解析配置生成Connection对象

- `Connnection newConnection(String path)`
  - path: 配置文件`stub.toml`所在目录路径

插件配置`stub.toml`

- 通用配置：插件名称、类型
- SDK配置：初始化JavaSDK，与区块链交互
- 资源列表：区块链可以访问的资源列表

插件配置默认位于`chains/`目录，可以配置多个stub，每个stub位于单独的子目录。

```
# 目录结构，conf/chains/stub名称/
conf/chains/
        └── bcos # stub名称：bcos
            └── stub.toml # stub.toml配置文件
            # 其他文件列表，比如：证书文件
```

`stub.toml`解析流程可以参考BCOS Stub stub.toml解析

BCOS示例

```
/** Request type定义，用户自定义 */
public class BCOSRequestType {
    // call
    public static final int CALL = 1000;
    // sendTransaction
    public static final int SEND_TRANSACTION = 1001;
    // 获取区块高度
    public static final int GET_BLOCK_NUMBER = 1002;
    // 获取BlockHeader
    public static final int GET_BLOCK_HEADER = 1003;
    // 获取交易Merkle证明
    public static final int GET_TRANSACTION_PROOF = 1004;
}

public class BCOSConnection implements Connection {
```

(continues on next page)

(续上页)

```
    /** BCOS SDK对象，调用区块链接口 */
    private final Web3jWrapper web3jWrapper;
    /** 为Driver提供统一发送接口，根据不同请求类型分别与区块链完成交互 */
    @Override
    public Response send(Request request) {
        switch (request.getType()) {
            /** call请求 */
            case BCOSRequestType.CALL:
                /** handle call request */
            /** sendTransaction请求 */
            case BCOSRequestType.SEND_TRANSACTION:
                /** handle sendTransaction request */
            /** 获取区块头 */
            case BCOSRequestType.GET_BLOCK_HEADER:
                /** handle getBlockeHeader request */
            /** 获取块高 */
            case BCOSRequestType.GET_BLOCK_NUMBER:
                /** handle getBlockNumber request */
            /** 获取交易Merkle证明 */
            case BCOSRequestType.GET_TRANSACTION_PROOF:
                /** handle getTransactionProof request */
            default:
                /** unrecognized request type */
        }
    }

    /** 获取资源列表 */
    @Override
    public List<ResourceInfo> getResources() {
        return resourceInfoList;
    }
}
```

BCOS stub.toml示例

```
[common]     # 通用配置
    name = 'bcos' # 名称，必须项
    type = 'BCOS2.0' # 必须项，插件类型，与插件@Stub注解定义的类型保持一致


[chain]      # BCOS链属性配置
    groupId = 1 # default 1
    chainId = 1 # default 1

[channelService]     # BCOS JavaSDK配置
    caCert = 'ca.crt'
    sslCert = 'sdk.crt'
    sslKey = 'sdk.key'
    timeout = 300000  # ms, default 60000ms
    connectionsStr = ['127.0.0.1:20200', '127.0.0.1:20201', '127.0.0.1:20202']

[[resources]]        # 资源配置列表
    name = 'HelloWeCross' # 资源名称
    type = 'BCOS_CONTRACT' # 资源类型，BCOS合约
    contractAddress = '0x8827cca7f0f38b861b62dae6d711efe92a1e3602' # 合约地址
```

Driver

Driver是Stub与WeCross Router交互的入口，用途包括:

- 发送交易

- 编解码交易
- 编解码区块
- 验证交易
- 其他功能

接口定义

```
public interface Driver {

    /** call或者sendTransaction请求,返回为true,其他类型返回false */
    public boolean isTransaction(Request request);

    /** 解码BlockHeader数据 */
    public BlockHeader decodeBlockHeader(byte[] data);

    /** 获取区块链当前块高 */
    public long getBlockNumber(Connection connection);

    /** 获取Block Header */
    public byte[] getBlockHeader(long blockNumber, Connection connection);

    /** 解析交易请求,请求可能为call或者sendTransaction */
    public TransactionContext<TransactionRequest> decodeTransactionRequest(byte[]␣
↪data);

        /** 调用合约,查询请求 */
    public TransactionResponse call(
            TransactionContext<TransactionRequest> request, Connection connection);

    /** 调用合约,交易请求 */
    public TransactionResponse sendTransaction(
            TransactionContext<TransactionRequest> request, Connection connection);

        /** 获取交易,并且对交易进行验证 */
    public VerifiedTransaction getVerifiedTransaction(
            String transactionHash,
            long blockNumber,
            BlockHeaderManager blockHeaderManager,
            Connection connection);
}
```

接口列表

- isTransaction 是否为请求交易
  - 参数列表:
    - Request request: 请求对象
  - 返回值:
    - request为call或者sendTransaction请求返回true，否则返回false
- getBlockNumber 获取当前区块高度
  - 参数列表:
    - Connection connection: Connection对象，发送请求
  - 返回值
    - 区块高度，负值表示获取区块高度失败
- getBlockHeader 获取区块头数据，区块头为序列化的二进制数据
  - 参数列表:

        * long blockNumber: 块高
        * Connection connection: Connection对象，发送请求
    - 返回值
        * 序列化的区块头数据，返回null表示获取区块头数据失败，可以使用decodeBlockHeader获取区块头对象

- decodeBlockHeader 解析区块头数据，返回区块头对象
    - 参数列表:
        * byte[] data: 序列化的区块头数据
    - 返回值
        * 区块头BlockHeader对象，返回null表示解析区块头数据失败

```
区块头对象
public class BlockHeader {
    /** 区块高度 */
    private long number;
    /** 上一个区块hash */
    private String prevHash;
    /** 区块hash */
    private String hash;
    /** 状态根，验证状态 */
    private String stateRoot;
    /** 交易根，验证区块交易 */
    private String transactionRoot;
    /** 交易回执根，验证区块交易回执 */
    private String receiptRoot;
}
```

- call、sendTransaction `call`与`sendTransaction`接口类似，前者用于查询状态，后者发送交易，修改区块链状态
    - 参数列表
        * TransactionContext request: 请求上下文，获取构造交易需要的数据，构造交易
        * Connection connection: Connection对象，发送请求
    - 返回值
        * TransactionResponse: 返回对象
    - 注意:
        * `sendTransaction`接口要对返回交易进行验证，各个区块链的验证方式有所不同，BCOS采用Merkle证明的方式对交易及交易回执进行验证

```
// 请求对象
public class TransactionRequest {
    /** 接口 */
    private String method;
    /** 参数 */
    private String[] args;
}

// 返回对象
public class TransactionResponse {
    // 返回状态，0表示成功，其他表示错误码
    private Integer errorCode;
    // 错误信息
    private String errorMessage;
    // 交易hash，sendTransaction时有效
```

(continues on next page)

(续上页)

```
    private String hash;
    // 区块高度，交易所在的区块的块高，sendTransaction时有效
    private long blockNumber;
    // 返回结果
    private String[] result;
}

// 交易请求上下文参数，获取构造交易需要的参数
public class TransactionContext<TransactionRequest> {
    // 交易请求
    private TransactionRequest data;
    // 账户，用于交易签名
    private Account account;
    // 请求资源，用于获取资源相关信息
    private ResourceInfo resourceInfo;
    // 区块头管理器，获取区块头信息
    private BlockHeaderManager blockHeaderManager;
}

// 区块头管理器
public interface BlockHeaderManager {
    // 获取当前块高
    public long getBlockNumber();
    // 获取区块头，阻塞操作
    public byte[] getBlockHeader(long blockNumber);
}
```

- getVerifiedTransaction 根据哈希和块高查询交易并校验交易，返回交易的请求与返回对象，校验交易方式与sendTransaction接口校验交易方式保持一致。
  - 参数列表
    - String transactionHash: 交易hash
    - long blockNumber: 交易所在区块高度
    - BlockHeaderManager blockHeaderManager: 区块头管理器，获取区块头
    - Connection connection: 发送请求
  - 返回值
    - VerifiedTransaction对象

```
public class VerifiedTransaction {
    /** 交易所在块高 */
    private long blockNumber;
    /** 交易hash */
    private String transactionHash;
    /** 交易调用的合约地址 */
    private String realAddress;
    /** 交易请求参数 */
    private TransactionRequest transactionRequest;
    /** 交易返回 */
    private TransactionResponse transactionResponse;
}
```

BCOS示例

这里给个完整的BCOS Stub发送交易的处理流程，说明Driver与Connection的协作，以及在BCOS中如何进行交易验证。

- BCOSDriver

```
@Override
public TransactionResponse sendTransaction(
        TransactionContext<TransactionRequest> request, Connection connection) {

    TransactionResponse response = new TransactionResponse();

    try {
        ResourceInfo resourceInfo = request.getResourceInfo();
        /** 合约的额外属性，BCOS构造交易需要这些参数，参考BCOS Stub.toml配置 */
        Map<Object, Object> properties = resourceInfo.getProperties();
        /** 获取合约地址 */
        String contractAddress = (String) properties.get(resourceInfo.getName());
        /** 获取群组Id */
        Integer groupId = (Integer) properties.get(BCOSConstant.BCOS_RESOURCEINFO_
↪GROUP_ID);
        /** 获取链Id */
        Integer chainId = (Integer) properties.get(BCOSConstant.BCOS_RESOURCEINFO_
↪CHAIN_ID);
        /** 获取块高 */
        long blockNumber = request.getBlockHeaderManager().getBlockNumber();
        BCOSAccount bcosAccount = (BCOSAccount) request.getAccount();
        /** 获取私钥 参考account.toml配置 */
        Credentials credentials = bcosAccount.getCredentials();

        /** 交易签名，使用credentials对构造的交易进行签名 */
        String signTx =
                SignTransaction.sign(
                        credentials,
                        contractAddress,
                        BigInteger.valueOf(groupId),
                        BigInteger.valueOf(chainId),
                        BigInteger.valueOf(blockNumber),
                        FunctionEncoder.encode(function));

        // 构造Request参数
        TransactionParams transaction = new TransactionParams(request.getData(),␣
↪signTx);
        Request req = new Request();
        /** Request类型 SEND_TRANSACTION */
        req.setType(BCOSRequestType.SEND_TRANSACTION);
        /** 参数JSON序列化 */
        req.setData(objectMapper.writeValueAsBytes(transaction));
        /** Connection send发送请求 */
        Response resp = connection.send(req);
        if (resp.getErrorCode() != BCOSStatusCode.Success) {
            /** Connection返回异常 */
            throw new BCOSStubException(resp.getErrorCode(), resp.
↪getErrorMessage());
        }
        /** 获取返回的交易回执，回执被序列化为byte[] */
        TransactionReceipt receipt =
                objectMapper.readValue(resp.getData(), TransactionReceipt.class);

        // Merkle证明，校验交易hash及交易回执，失败抛出异常
        verifyTransactionProof(
                receipt.getBlockNumber().longValue(),
                receipt.getTransactionHash(),
                request.getBlockHeaderManager(),
                receipt);

        /** 其他逻辑，构造返回 */
```

(continues on next page)

(续上页)

```
    } catch (Exception e) {
        /** 异常场景 */
        response.setErrorCode(BCOSStatusCode.UnclassifiedError);
        response.setErrorMessage(" errorMessage: " + e.getMessage());
    }

    return response;
}
```

- BCOSConnection

```
public class BCOSConnection implements Connection {
    /** BCOS JavaSDK 实例句柄 */
    private final Web3jWrapper web3jWrapper;

    @Override
    public Response send(Request request) {
        switch (request.getType()) {
            case BCOSRequestType.SEND_TRANSACTION:
                /** type: SEND_TRANSACTION */
                return handleTransactionRequest(request);
            /** 其他case场景 */
        }
    }

    /** 发送交易请求处理 */
    public Response handleTransactionRequest(Request request) {
        Response response = new Response();
        try {
            /** 参数JSON序列化，反序列化得到请求参数*/
            TransactionParams transaction =
                    objectMapper.readValue(request.getData(), TransactionParams.
↪class);
            /** 签名交易 */
            String signTx = transaction.getData();
            /** 调用BCOS RPC发送交易接口，获取回执以及Merkle证明 */
            TransactionReceipt receipt = web3jWrapper.
↪sendTransactionAndGetProof(signTx);

            /** 交易回执不存在 */
            if (Objects.isNull(receipt)
                    || Objects.isNull(receipt.getTransactionHash())
                    || "".equals(receipt.getTransactionHash())) {
                throw new BCOSStubException(
                        BCOSStatusCode.TransactionReceiptNotExist,
                        BCOSStatusCode.getStatusMessage(BCOSStatusCode.
↪TransactionReceiptNotExist));
            }

            /** 交易执行失败 */
            if (!receipt.isStatusOK()) {
                throw new BCOSStubException(
                        BCOSStatusCode.SendTransactionNotSuccessStatus,
                        StatusCode.getStatusMessage(receipt.getStatus()));
            }

            /** 交易正确执行 */
            response.setErrorCode(BCOSStatusCode.Success);
            response.setErrorMessage(BCOSStatusCode.
↪getStatusMessage(BCOSStatusCode.Success));
            /** 返回交易回执，回执JSON方式序列化 */
```

(continues on next page)

(续上页)

```
            response.setData(objectMapper.writeValueAsBytes(receipt));
        } catch (Exception e) {
            /** 异常情况 */
            response.setErrorCode(BCOSStatusCode.HandleSendTransactionFailed);
            response.setErrorMessage(" errorMessage: " + e.getMessage());
        }
        return response;
    }
}
```

6.3.3 生成Jar

Stub插件需要打包生成shadow jar才可以被WeCross Router加载使用，在Gradle中引入shadow插件。

shadow插件使用: Gradle Shadow Plugin

- 引入shadow插件

```
plugins {
    // 其他插件列表
    id 'com.github.johnrengelman.shadow' version '5.2.0'
}
```

- 添加打包task

```
jar.enabled = false
project.tasks.assemble.dependsOn project.tasks.shadowJar

shadowJar {
    destinationDir file('dist/apps')
    archiveName project.name + '.jar'
    // 其他打包逻辑
}
```

- 执行build操作

```
bash gradlew build
```

dist/apps目录生成jar文件

6.3.4 参考链接

WeCross-BCOS-Stub

WeCross-Fabric-Stub

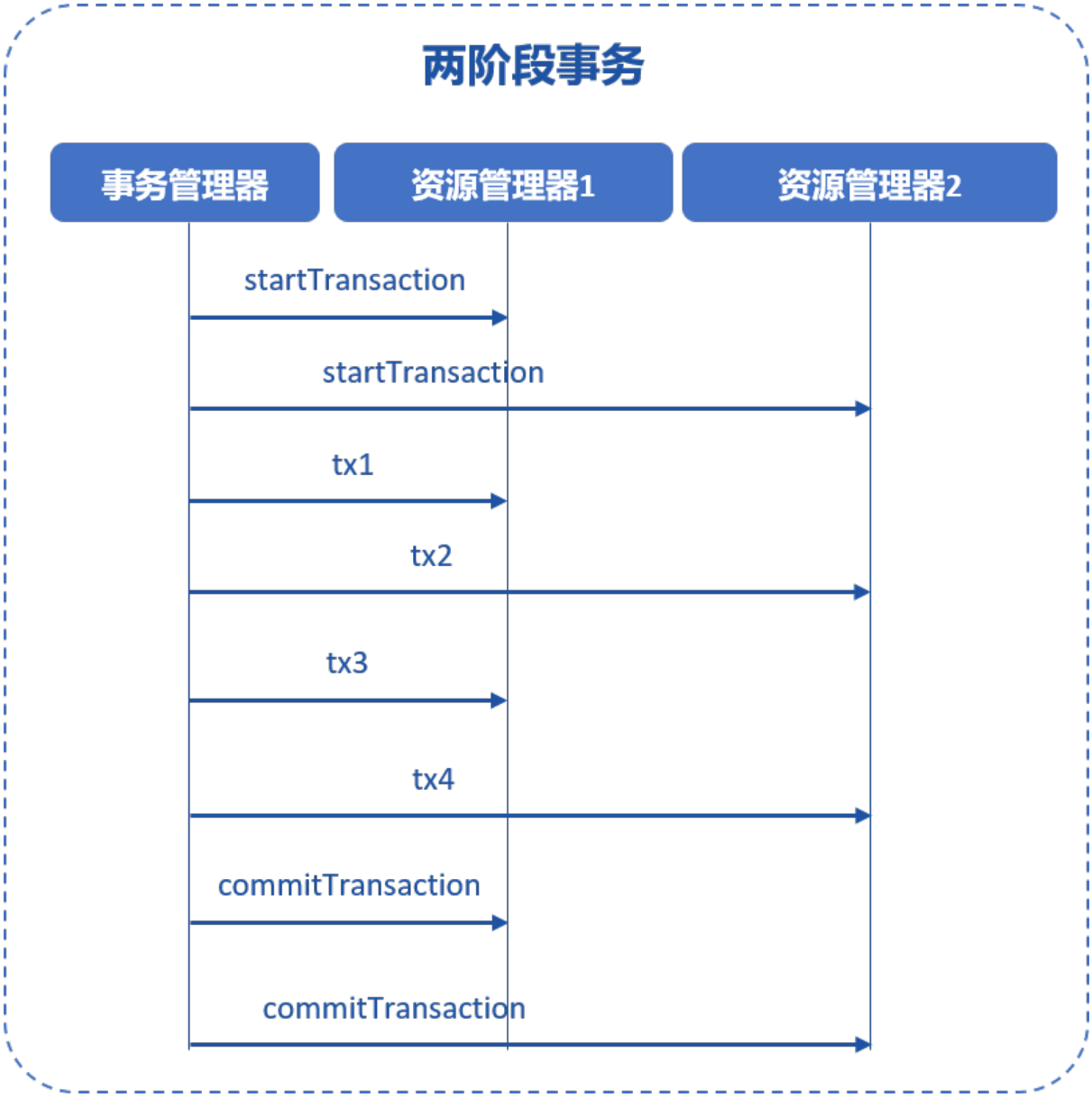
WeCross文档

CHAPTER 7

跨链事务

7.1 两阶段事务

两阶段事务是分布式数据库和分布式系统中常见的事务模型，两阶段事务的架构中包含事务管理器和资源管理器等多个角色。



在WeCross事务模型中，资源管理器是各个参与事务的区块链，事务管理器由WeCross的跨链路由实现，WeCross跨链路由对多个区块链上的资源进行事务的调度和管理。

用户使用WeCross SDK，可以同时操作多个跨链路由和多个链上的资源，参与一个两阶段事务，确保事务过程中所有资源的原子性。

7.1.1 准备工作

无论何种链、智能合约或链码，参与基本的两阶段事务流程时不需要提前做修改，但如果要支持两阶段事务的回滚操作，资源的实现中必须包含与正向交易接口相反的“反向”交易接口，以Solidity智能合约为例，为了支持两阶段事务的回滚操作需要做修改：

原Solidity合约包含一个正向交易接口，如转账：

```
function transfer(string memory from, string memory to, int balance) public {
        // balance check...
    balances[from] -= balance;
    balances[to] += balance;
}
```

注意，案例为了简明考虑，此处省略了余额和透支检查的逻辑，只列出了正向交易接口的关键逻辑

可以看出，正向交易接口的逻辑是将from的余额减少，等量增加to的余额，实现了余额从from转移到to的效果。当事务成功执行时，正向交易接口的执行结果将被认可，一旦事务出现异常，需要回滚时，Solidity合约需要提供一个反向交易接口。

反向交易接口的参数与正向交易接口相同，函数名增加_revert的后缀，表明这是一个用于两阶段事务的反向交易接口，当两阶段事务需要回滚时，WeCross会自动执行Solidity合约的反向交易接口，反向交易接口实现如下：

```
function transfer_revert(string memory from, string memory to, int balance) public
↪{
        // balance check...
        balance[from] += balance;
        balance[to] -= balance;
}
```

反向交易接口的逻辑与正向接口相反，将from的余额增加，等量减少to的余额，实现了余额从to转移到from的效果，在相同输入参数的情况下其行为和执行结果与正向交易接口相反。

WeCross跨链路由在事务执行失败时，会按照正向交易接口执行的顺序，以相反的顺序调用反向交易接口。

举例，两阶段事务过程中执行了一系列正向交易接口如下：

```
transfer1('from1', 'to1', 100)
```

(continues on next page)

(续上页)

```
transfer2('from2', 'to2', 200)
transfer3('from3', 'to3', 300)
```

当事务发生异常，WeCross会按以下顺序执行反向交易接口：

```
transfer3_revert('from3', 'to3', 300)
transfer2_revert('from2', 'to2', 200)
transfer1_revert('from1', 'to1', 100)
```

7.1.2 使用

两阶段事务无需事先配置即可使用，如果需要使用两阶段事务的回滚功能，则要求参与事务的资源都实现了反向交易接口。

7.1.3 控制台使用两阶段事务

开始两阶段事务

使用startTransaction命令，开始两阶段事务

```
startTransaction [transactionID] [account_1] ... [account_n] [path_1] ... [path_n]
```

参数解析：

- transactionID：事务ID，类型为字符串，由用户指定，作为事务的唯一标识，后续所有的事务资源操作都必须指定该事务ID
- account_1 … account_n：用于开始事务的账号列表，由于两阶段事务可能跨越多种区块链，多种区块链会使用不同类型的账号，因此开始事务时，需要为每种区块链指定至少一个账号，WeCross会使用相应类型的账号向链上发送开始事务交易，该账号列表仅用于开始事务，事务开始后，可以使用该账号列表以外的账号来发送事务交易
- path_1 … path_n：参与事务的资源路径列表，路径列表中的资源会被本次事务锁定，锁定后仅限本事务相关的交易才能对这些资源发起写操作，非本次事务的所有写操作都会被拒绝

例子：

开始一个事务，事务ID为100，账号为account，资源为zone.chain.res1

```
startTransaction 100 account zone.chain.res1
```

开始一个事务，事务ID为200，账号为account、fabric，资源为zone.chain.res1、zone.chain.res2

```
startTransaction 200 account fabric zone.chain.res1 zone.chain.res2
```

发起事务交易

使用execTransaction命令，发起事务交易，execTransaction命令与sendTransaction命令类似，增加了transactionID和seq字段，其中，transactionID字段为开始两阶段事务（startTransaction）时填入的transactionID字段，seq字段为顺序字段，该字段每个事务交易唯一，且要求递增。

任何资源一旦参与了事务，就无法用sendTransaction来向该资源发送交易，必须使用execTransaction

```
execTransaction [path] [account] [transactionID] [seq] [method] [args]
```

参数解析：

- path：资源路径
- account：交易账号
- transactionID：事务ID，该资源正在参与事务的ID
- seq：事务编号，本次操作的编号，每次事务交易唯一，要求递增
- method：接口名，同sendTransaction
- args：参数，同sendTransaction

例子：

通过控制台，调用transfer接口：

```
execTransaction zone.chain.res1 account 100 1 transfer 'fromUserName' 'toUserName'␣
↪100 #调用事务资源zone.chain.res1的transfer接口

execTransaction zone.chain.res2 account 100 1 transfer 'fromUserName' 'toUserName'␣
↪100 #调用事务资源zone.chain.res2的transfer接口
```

提交事务

使用commitTransaction命令，提交事务，确认事务执行过程中所有的变动。

```
commiTransaction [transactionID] [account_1] ... [account_n] [path_1] ... [path_n]
```

参数解析：

- transactionID：事务ID，待提交事务的ID
- account_1 ... account_n：用于提交事务的账号列表，由于两阶段事务可能跨越多种区块链，多种区块链会使用不同类型的账号，需要为每种区块链指定至少一个账号，WeCross会使用相应类型的账号向链上发送提交事务交易
- path_1 ... path_n：用于提交事务的路径列表，此处填写所有参与了事务的链，无需精确到参与事务的资源，填入链的路径即可

例子：

通过控制台，执行一个完整的事务步骤

```
startTransaction 100 account zone.chain.res1 zone.chain.res2 #开始事务

execTransaction zone.chain.res1 account 100 1 transfer 'fromUserName' 'toUserName'␣
↪100 #调用事务资源zone.chain.res1的transfer接口
execTransaction zone.chain.res2 account 100 1 transfer 'fromUserName' 'toUserName'␣
↪100 #调用事务资源zone.chain.res2的transfer接口

commitTransaction 100 account zone.chain #提交事务
```

回滚事务

当事务的某个步骤执行失败，需要撤销本次事务的所有变更时，使用rollbackTransaction命令

```
rollbackTransaction [transactionID] [account_1] ... [account_n] [path_1] ... [path_
↪n]
```

参数解析：

- transactionID：事务ID，待回滚事务的ID
- account_1 ... account_n：用于回滚事务的账号列表，由于两阶段事务可能跨越多种区块链，多种区块链会使用不同类型的账号，需要为每种区块链指定至少一个账号，WeCross会使用相应类型的账号向链上发送回滚事务交易
- path_1 ... path_n：用于回滚事务的路径列表，此处填写所有参与了事务的链，无需精确到参与事务的资源，填入链的路径即可

例子：

通过控制台，执行一个完整的事务步骤

```
startTransaction 100 account zone.chain.res1 zone.chain.res2 #开始事务

execTransaction zone.chain.res1 account 100 1 transfer 'fromUserName' 'toUserName'␣
↪100 #调用事务资源zone.chain.res1的transfer接口
execTransaction zone.chain.res2 account 100 1 transfer 'fromUserName' 'toUserName'␣
↪100 #调用事务资源zone.chain.res2的transfer接口
execTransaction zone.chain.res2 account 100 1 set 'fromUserName' 'property' "true"␣
↪ #调用事务资源zone.chain.res2的set接口，假设该接口调用失败

rollbackTransaction 100 account zone.chain #回滚事务
```

7.2 哈希时间锁定

哈希时间锁合约（hash time lock contract，htlc）能够实现两条异构链之间资产的原子交换，即跨链转账。

如果你想**快速体验**跨链转账可以参考体验WeCross。

WeCross提供了Solidity和Golang版本的htlc基类合约，基于htlc基类合约可以轻松开发适用于不同资产类型的htlc应用合约。

本章节以FISCO BCOS和Hyperledger Fabric的**示例资产合约**为例，演示如何实现两条异构链的资产互换。

7.2.1 准备工作

要完成资产互换，需要在各自链上部署资产合约以及哈希时间锁合约，然后通过WeCross控制台创建跨链转账提案，router会根据提案信息自动完成跨链转账。

部署WeCross和控制台

- 以组网方式搭建两个router
- 搭建两个WeCross控制台，分别连接两个router

FISCO BCOS前期准备

部署合约

进入连接BCOS链的router的控制台。

```
bash start.sh

# 发行资产，拥有者是bcos_user1，资产名为htlc，最小单位1，发行数量100000000
[WeCross]> bcosDeploy payment.bcos.ledger bcos_user1 contracts/solidity/
↪LedgerSample.sol LedgerSample 1.0 token htlc 1 100000000
# 资产合约地址需要记录下来
Result: 0xf4fdcdfe0184644f09a1cfa16a945cc71a5d44ff

# 部署htlc合约
[WeCross]> bcosDeploy payment.bcos.htlc bcos_user1 contracts/solidity/
↪LedgerSampleHTLC.sol LedgerSampleHTLC 1.0
# htlc合约地址需要记录下来
Result: 0x22a83719f748da09845d91fe1a2f44437f0ad13b
```

资产授权

要完成跨链转账，资产拥有者需要将资产的转移权授权给哈希时间锁合约。

进入连接BCOS链的router的控制台。

```
bash start.sh

# approve [被授权者地址]（此处为自己的哈希时间锁合约地址），[授权金额]
[WeCross]> sendTransaction payment.bcos.ledger bcos_user1 approve␣
↪0x22a83719f748da09845d91fe1a2f44437f0ad13b 1000000
Txhash  : 0x718e948b0ab55697c61675253acfd580104e539c85e0fcb23c0686457ea429d4
BlockNum: 46
Result  : [true]
```

哈希时间锁合约初始化

需要将资产合约的地址和对手方的哈希时间锁合约地址保存到自己的哈希时间锁合约。

进入连接BCOS链的router的控制台。

```
bash start.sh

# init [己方资产合约地址] [对端哈希时间锁合约地址](此处约定Fabric的合约名为htlc，之后将以该名称
安装和初始化链码)
[WeCross]> sendTransaction payment.bcos.htlc bcos_user1 init␣
↪0xf4fdcdfe0184644f09a1cfa16a945cc71a5d44ff htlc
Txhash  : 0x7df25ce20e7db6f6bba836bf54c258bb5386873e14b57e74a2371ec367b31779
BlockNum: 51
Result  : [success]

# 查看bcos_user1的地址
[WeCross]> call payment.bcos.htlc bcos_user1 queryAddress
# bcos_user1的地址需要记录下来，发起转账提案时需要
Result: [0x55f934bcbe1e9aef8337f5551142a442fdde781c]

# 查看bcos_user1即owner余额，检查是否初始化成功
[WeCross]> call payment.bcos.htlc bcos_user1 balanceOf␣
↪0x55f934bcbe1e9aef8337f5551142a442fdde781c
[100000000]
```

###Fabric前期准备

进入连接fabric链的router的控制台。

部署合约

```
bash start.sh

# 在机构1安装资产合约链码
```

(continues on next page)

(续上页)

```
[WeCross]> fabricInstall payment.fabric.ledger fabric_admin_org1 Org1 contracts/
↪chaincode/ledger 1.0 GO_LANG
path: classpath:contracts/chaincode/ledger
Result: Success
# 在机构2安装资产合约链码
[WeCross]> fabricInstall payment.fabric.ledger fabric_admin_org2 Org2 contracts/
↪chaincode/ledger 1.0 GO_LANG
path: classpath:contracts/chaincode/ledger
Result: Success
# 实例化链码，为fabric_admin发行资产100000000
[WeCross]> fabricInstantiate payment.fabric.ledger fabric_admin ["Org1","Org2"]␣
↪contracts/chaincode/ledger 1.0 GO_LANG default ["token","htlc","100000000"]
Result: Query success. Please wait and use 'listResources' to check.

# 在机构1安装哈希时间锁合约链码
[WeCross]> fabricInstall payment.fabric.htlc fabric_admin_org1 Org1 contracts/
↪chaincode/htlc 1.0 GO_LANG
path: classpath:contracts/chaincode/htlc
Result: Success
# 在机构2安装哈希时间锁合约链码
[WeCross]> fabricInstall payment.fabric.htlc fabric_admin_org2 Org2 contracts/
↪chaincode/htlc 1.0 GO_LANG
path: classpath:contracts/chaincode/htlc
Result: Success
# 实例化哈希时间锁合约，需要写入[己方资产合约名，channel，以及BCOS的哈希时间锁合约地址]
[WeCross]> fabricInstantiate payment.fabric.htlc fabric_admin ["Org1","Org2"]␣
↪contracts/chaincode/htlc 1.0 GO_LANG default ["ledger","mychannel",
↪"0x22a83719f748da09845d91fe1a2f44437f0ad13b"]
Result: Query success. Please wait and use 'listResources' to check.
```

资产授权 fabric的示例资产合约通过创建一个托管账户实现资产的授权。

进入连接fabric链的router的控制台。

```
bash start.sh

# fabric_admin创建一个托管账户完成授权
[WeCross]> sendTransaction payment.fabric.ledger fabric_admin createEscrowAccount␣
↪1000000
Txhash  : d6a6241cbaac9cad768465213f3462f54c62e32f168c26bcce23d060315f0751
BlockNum: 1097
Result  : [HTLCoin-Admin@org1.example.com-EscrowAccount]

# 查看fabric_admin的地址
[WeCross]> call payment.fabric.htlc fabric_admin queryAddress
Result: [Admin@org1.example.com]

# 查看fabric_admin授权后的余额
[WeCross]> call payment.fabric.htlc fabric_admin balanceOf Admin@org1.example.com
Result: [99000000]
```

7.2.2 哈希时间锁合约配置

配置FISCO BCOS端router 在主配置文件`wecross.toml`中配置htlc任务。

```
[[htlc]]
     #直连链的哈希时间锁资源路径
    selfPath = 'payment.bcos.htlc'

    #确保已在router的accounts目录配置了bcos_default_account账户
```

(continues on next page)

(续上页)

```
    account1 = 'bcos_default_account'

    #对手方的哈希时间锁资源路径
    counterpartyPath = 'payment.fabric.htlc'

    #确保已在router的accounts目录配置了fabric_default_account账户
    account2 = 'fabric_default_account'
```

重要:

- 主配置中的htlc资源路径以及能访问这两个资源的账户请结合实际情况配置。

配置Fabric端router

在主配置文件`wecross.toml`中配置htlc任务。

```
[[htlc]]

     #直连链的的哈希时间锁资源路径
    selfPath = 'payment.fabric.htlc'

    #确保已在router的accounts目录配置了fabric_default_account账户
    account1 = 'fabric_default_account'

    #对手方的哈希时间锁资源路径
    counterpartyPath = 'payment.bcos.htlc'

    #确保已在router的accounts目录配置了bcos_default_account账户
    account2 = 'bcos_default_account'
```

重启两个router

```
bash stop.sh && bash start.sh
```

7.2.3 发起跨链转账

假设跨链转账发起方是FISCO BCOS的用户，发起方选择一个`secret`，计算 $$ hash = sha256(secret) $$ 得到`hash`，然后和Fabric的用户即参与方协商好各自转账的金额、账户以及时间戳。

协商内容

- FISCO BCOS的两个账户：
  - 资产转出者：0x55f934bcbe1e9aef8337f5551142a442fdde781c（BCOS链发行资产的地址）
  - 资产接收者：0x2b5ad5c4795c026514f8317c7a215e218dccd6cf
- FISCO BCOS转账金额：700
- Fabric的两个账户：
  - 资产转出者：Admin@org1.example.com （fabric链发行资产的地址）
  - 资产接收者：User1@org1.example.com
- Fabric转账金额500
- 哈希: 2afe76d15f32c37e9f219ffd0a8fcefc76d850fa9b0e0e603cbd64a2f4aa670d
- 哈希原像: e7d5c31dcc6acae547bb0d84c2af05413994c92599bff89c4abd72866f6ac5c6（协商时只有发起方知道）
- 时间戳t0：发起方的超时时间，单位s

- 时间戳t1： 参与方的超时时间，单位s

注解:

- 哈希原像和哈希可通过控制台命令 genSecretAndHash 生成。
- 两个时间戳可通过控制台命令 genTimelock 生成。
- 时间戳需要满足条件：t0 > t1 + 300 > now + 300

创建跨链转账提案

两条链的**资产转出者**通过WeCross控制台创建跨链转账提案，将协商的转账信息写入各自的区块链。

- 命令介绍
  - 命令：`newHTLCProposal`
  - 参数：`path, account(资产转出者账户名), hash, secret, role, sender0, receiver0, amount0, timelock0, sender1, receiver1, amount1, timelock`
- 注意事项
  - 其中下标为0的参数是发起方信息。
  - 发起方`secret`传入哈希原像，`role`传入true；参与方`secret`传入null，`role`传入false。
- 启动控制台
- 发起方创建转账提案

该步骤由发起方即BCOS链的资产转出者完成。

进入连接BCOS链的router的控制台。

```
bash start.sh

[WeCross]> newHTLCProposal payment.bcos.htlc bcos_user1␣
↪edafd70a27887b361174ba5b831777c761eb34ef23ee7343106c0b545ec1052f␣
↪049db09dd9cf6fcf69486512c1498a1f6ea11d33b271aaad1893cd590c16542a true␣
↪0x55f934bcbe1e9aef8337f5551142a442fdde781c␣
↪0x2b5ad5c4795c026514f8317c7a215e218dccd6cf 700 2000010000 Admin@org1.example.com␣
↪User1@org1.example.com 500 2000000000
```

- 参与方创建转账提案

该步骤由参与方即fabric链的资产转出者完成。

进入连接fabric链的router的控制台。

```
bash start.sh

[WeCross]> newHTLCProposal payment.fabric.htlc fabric_admin␣
↪edafd70a27887b361174ba5b831777c761eb34ef23ee7343106c0b545ec1052f null false␣
↪0x55f934bcbe1e9aef8337f5551142a442fdde781c␣
↪0x2b5ad5c4795c026514f8317c7a215e218dccd6cf 700 2000010000 Admin@org1.example.com␣
↪User1@org1.example.com 500 2000000000
```

结果确认

哈希时间锁合约提供了查询余额的接口，可通过WeCross控制台调用，查看两方的接收者是否到账。

- FISCO BCOS用户确认

```
[WeCross]> call payment.bcos.htlc bcos_user1 balanceOf␣
↪0x2b5ad5c4795c026514f8317c7a215e218dccd6cf
Result: [700]
```

- Fabric用户确认

```
[WeCross]> call payment.fabric.htlc fabric_admin balanceOf User1@org1.example.com
Result: [500]
```

注：跨链转账存在交易时延，取决于两条链以及机器的性能，一般需要5~25s完成转账。

CHAPTER 8

应用场景

8.1 数字资产跨链

区块链天然具有金融属性，有望为金融业带来更多创新。支付清算方面，在基于区块链技术的架构下，市场多个参与者维护的多个账本或区块链融合连通并实时交互，短短几分钟内就能完成现在两三天才能完成的支付、对账、清算任务，降低了跨行跨境交易的复杂性和成本；同时，区块链技术能够确保交易记录透明安全，方便监管部门追踪链上交易，快速定位高风险交易流向。数字票据和供应链金融方面，区块链技术可以有效解决中小企业融资难问题。目前的供应链金融很难惠及产业链上游的中小企业，因为他们跟核心企业往往没有直接贸易往来，金融机构难以评估其信用资质。基于区块链技术，可以建立一种联盟多链网络，涵盖核心企业、上下游供应商、金融机构等，核心企业发放应收账款凭证给其供应商，票据数字化上链后可在供应商之间跨链流转，每一级供应商可凭数字票据实现对应额度的融资。

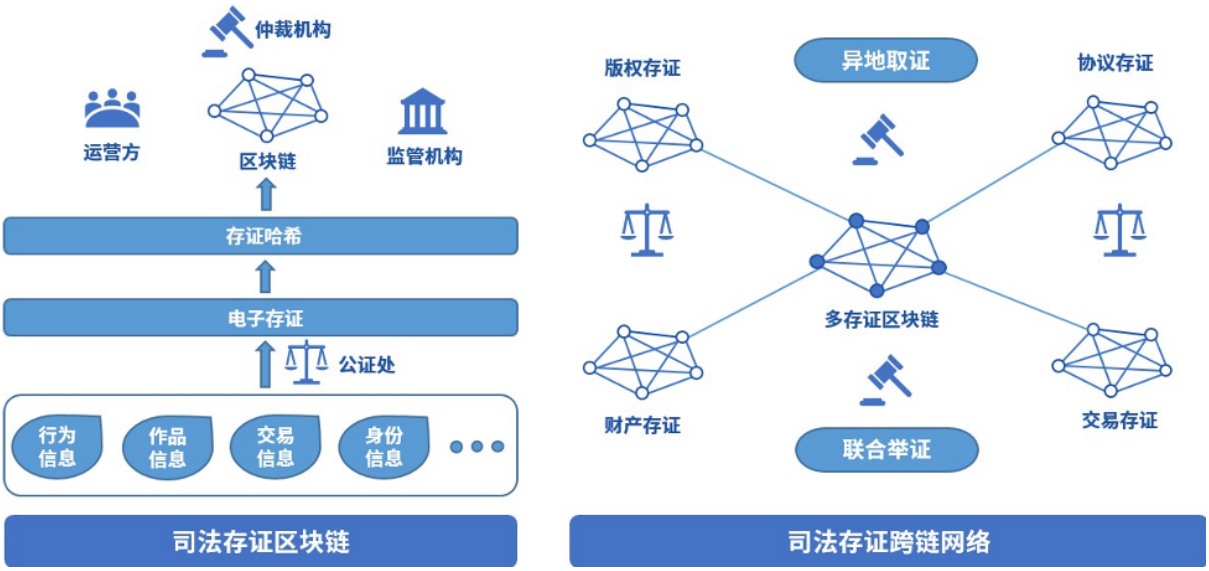
伴随着区块链在金融领域落地应用的飞速增长，多元化的数字资产场景和区块链应用带来了区块链资产相互隔离的问题，不同数字资产业务彼此搭建的区块链上的数字资产无法安全可信地实现互通，区块链上存在的数字资产价值越来越大，跨链的需求愈发迫切。

WeCross支持以多种网络拓扑模型搭建数字资产的跨链分区。在交易逻辑上，两阶段事务模型和HTLC事务模型将实现数字资产的去中心、去信任和不可篡改的转移。在安全防护上，加密和准入机制将保障数字资产转移的安全与可信。通过以上技术优势，WeCross将助力过去纸质形态的资产凭证全面数字化，让资产和信用层层深入传递到产业链末端，促进数字经济的发展。

8.2 司法跨域仲裁

随着数字经济高速发展，司法证据正逐步进入电子化时代。2017年9月，微众银行区块链团队与第三方存证公司合作，推出区块链司法存证与仲裁平台，开创将仲裁、法院等机构作为链上节点的先河，并于2018年2月，联合仲裁机构基于该平台出具业内首份裁决书，标志着区块链应用在司法领域的真正落地并完成价值验证；2018年6月，杭州互联网法院开始探求区块链在司法场景中的运用，进一步确立了区块链存证电子证据的合法性；2018年9月，北京互联网法院推出电子证据平台“天平链”，加速推动在网络空间治理的法治化进程。由于区块链司法应用能够极大缩减仲裁流程，仲裁机构得以快速完成证据核实，快速解决纠纷。

随着区块链应用在司法存证领域的普及，不同司法存证链之间连通的需求愈发强烈。但区块链的信任模型使得不同的司法存证链上的证据无法互通互信，当司法仲裁需要异地取证或是联合举证时，需要引入一个中心化的可信机构来进行协调，影响了区块链的实用价值。



WeCross跨链技术可以将各家存证链的证据统一抽象成证据资源，在不同的司法存证链之间可信地传输证据。WeCross可以搭建一个拥有多类型存证的存证链网络，在面向重大问题和重大纠纷时，去中心化地帮助各个链交互完备、可信和强有力的证据材料，帮助仲裁机构完成裁决。

8.3 个体数据跨域授权

随着WeIdentity、Hyperledger Indy等遵循DID协议的区块链身份认证系统出现，多个国家和地区开展了多中心化身份认证的实践与落地，多中心化身份认证目前市场需求巨大，加之政策鼓励支持，行业方兴未艾，处于高速发展的黄金时期。2019年2月27日，微众银行区块链团队与澳门政府设立的澳门科学技术发展基金签署合作协议，在智慧城市、民生服务、政务管理、人才培训等方面开展合作。双方合作的首个项目基于“WeIdentity”的实体身份标识及可信数据交换解决方案展开，这是区块链在粤港澳大湾区应用落地的重要一步。

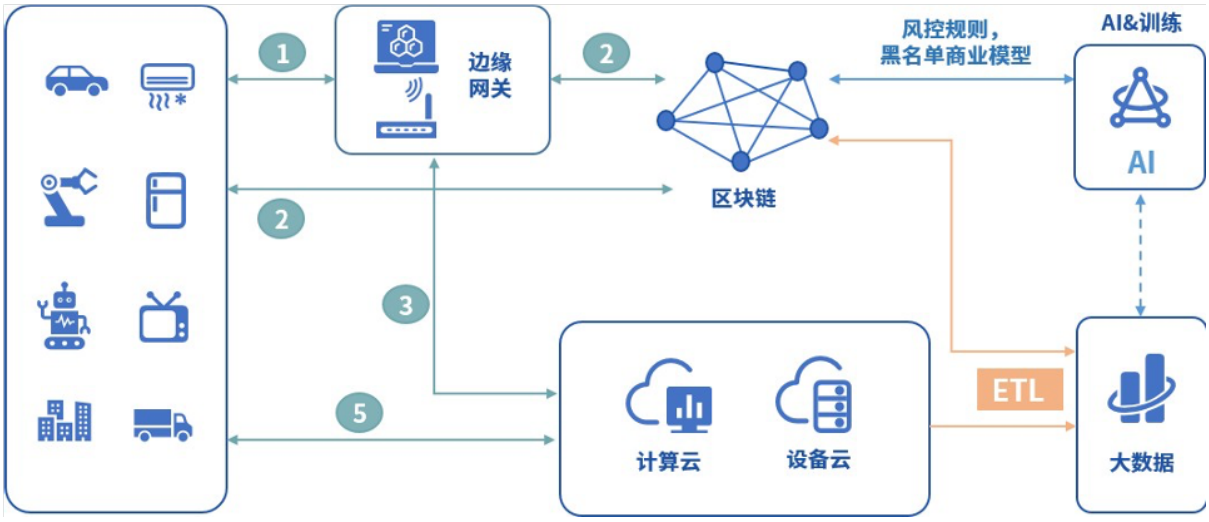
身份认证正向跨地域的方向发展，不同地域、业务和基于不同区块链平台的身份认证产品之间尚不能互认的现状造成信息的鸿沟，导致身份和资质等数据仍然局限在小范围的地域和业务内，无法互通。

WeCross 可以将多个不同架构、行业和地域的多中心化身份认证平台联结起来，帮助多中心化身份认证更好地解决数据孤岛、数据滥用和数据黑产的问题，在推进数据资源开放共享与信息流通，促进跨行业、跨领域、跨地域大数据应用，形成良性互动的产业发展格局上，发挥更大的作用。

8.4 物联网跨平台联动

随着智能穿戴、智能家居、无人机及以人脸识别等人工智能设备的普及，智能设备的类别越来越多，人机交互的频次也越来越高，物联网数据的类型和结构呈现多样化和复杂化的趋势。在5G时代，实现万物互联之后，数据和场景的复杂度更是呈几何倍数增长。区块链技术为物联网设备提供信任机制，保证所有权、交易等记录的可信性、可靠性及透明性，同时还可为用户隐私提供保障机制，从而有效解决物联网发展面临的大数据管理、信任、安全和隐私等问题，推进物联网向更加灵活化、智能化的形态演进。

目前物联网行业的区块链项目，有的旨在解决物联网碎片化严重、物联网产品没有标准化等痛点，有的则探索区块链在智能城市、基础设施、智能电网、供应链以及运输等领域的应用。然而，它们都面临着相同的困境。物联网设备硬件模块的选择和组合非常多样，对区块链平台的支持能力不尽相同，一旦硬件部署完成后难以更新，单一的区块链平台在连通多样化的物联网设备时必然会遇到瓶颈，无法全面满足所有物联网设备在多样化场景中的需求。



WeCross跨链技术支持物联网设备跨链平行扩展，可用于构建高效、安全的分布式物联网网络，以及部署海量设备网络中运行的数据密集型应用；WeCross跨链技术可以安全可信地融合连通多个物联网设备的区块链，在功能和安全上满足多样的场景需求。

CHAPTER 9

FAQ

9.1 问题1

下载速度太慢，下不下来。

回答

我们提供了多种下载方式，点击此处查看。

9.2 问题2

在搭建Demo时报错

```
========== ERROR !!! FAILED to execute End-2-End Scenario ==========

ERROR !!!! Test failed
```

回答

Fabric 的demo和机器上的Fabric网络冲突了，尝试用demo目录下的`clear.sh`用脚本清理机器上已有的Fabric网络。

9.3 问题3

在部署demo时，**MacOS用户**若出现“**无法打开**”，“**无法验证开发者**”的情况。

回答

MacOS对下载的包权限要求较为严格，必须同一个进程下载的才可执行，可采用如下方法解决：

```
# 清理环境
cd ~/demo/ && bash clear.sh && cd ~ && rm -rf demo
# 将三个步骤的命令拼成一条命令执行
bash <(curl -sL https://github.com/WeBankFinTech/WeCross/releases/download/
↪resources/download_demo.sh) && cd demo && bash build.sh
```

CHAPTER 10

社区

10.1 贡献代码

10.1.1 接入更多链

WeCross，目前适配了

- FISCO BCOS
- Fabric

WeCross一直在迭代，想接入什么链？一起来写代码吧

- 联系社区（issue） -> 手把手指导 -> 开发 -> 贡献PR -> 合入 -> WeCross！

10.1.2 更多

- 点Star！！！
- 提交代码(Pull Requests)。
- 提问和提交BUG