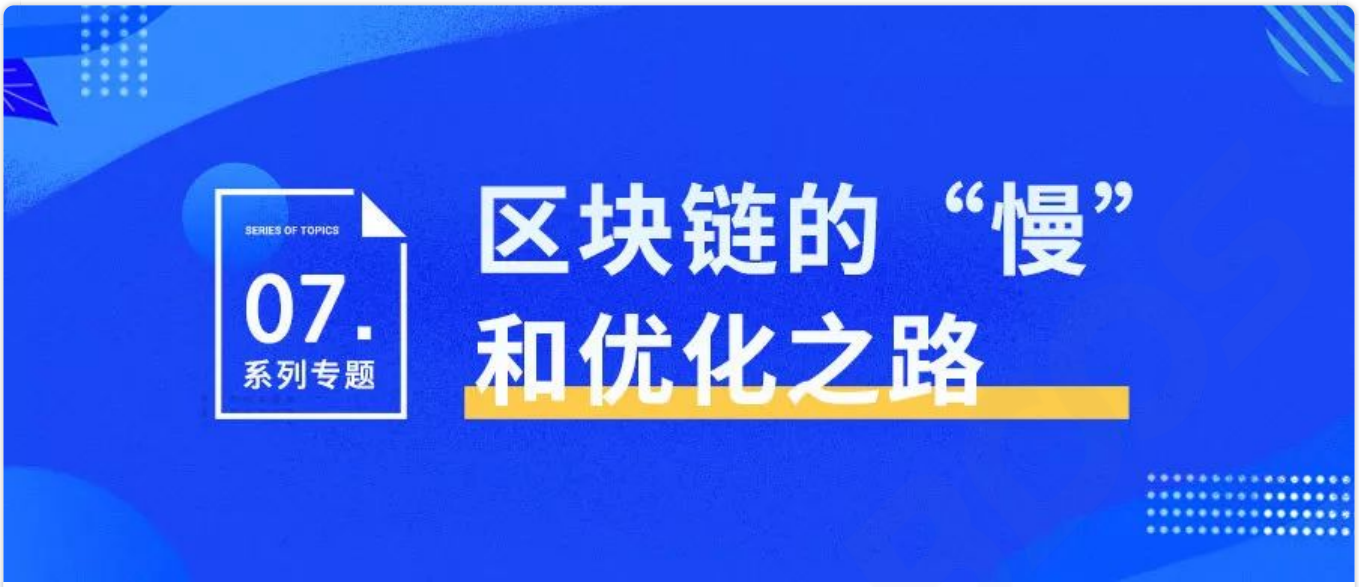
FISCO BCOS可并行合约开发框架（附实操教程）

原创 石翔 FISCO BCOS开源社区 2019-05-16



FISCO BCOS

系列专题 | 区块链的“慢”和优化之路（7）

可并行合约开发框架

作者：石翔

石翔

FISCO BCOS核心开发者

再小的性能问题都是问题

AUTHOR | 作者

本专题系列文章追到现在，也许你会想问，FISCO BCOS的并行到底怎么用？作为专题的完结篇，本文就来揭晓“庐山真面目”，并教你上手使用FISCO BCOS的并行特性！

FISCO BCOS提供了可并行合约开发框架，开发者按照框架规范编写的合约，能够被FISCO BCOS节点并行地执行。

并行合约的优势有：

- **高吞吐：**多笔独立交易同时被执行，能最大限度利用机器的CPU资源，从而拥有较高的TPS
- **可拓展：**可以通过提高机器的配置来提升交易执行的性能，以支持不断扩大业务规模

接下来，我将介绍如何编写FISCO BCOS并行合约，以及如何部署和执行并行合约。

预备知识

并行互斥

两笔交易是否能被并行执行，依赖于这两笔交易是否存在**互斥**。互斥，是指两笔交易各自**操作合约存储变量的集合存在交集**。

例如，在转账场景中，交易是用户间的转账操作。用transfer(X, Y) 表示从X用户转到Y用户的转账接口。互斥情况如下：

交易	互斥对象	交集	是否互斥
transfer(A, B) 和 transfer(A, C)	[A, B] 和 [A, C]	[A]	互斥，不可并行执行
transfer(A, B) 和 transfer(B, C)	[A, B] 和 [B, C]	[B]	互斥，不可并行执行
transfer(A, C) 和 transfer(B, C)	[A, C] 和 [B, C]	[C]	互斥，不可并行执行
transfer(A, B) 和 transfer(A, B)	[A, B] 和 [A, B]	[A, B]	互斥，不可并行执行
transfer(A, B) 和 transfer(C, D)	[A, B] 和 [C, D]	无	无互斥，可并行执行

此处给出更具体的定义：

- **互斥参数**：合约**接口**中，与合约存储变量的“读/写”操作相关的参数。例如转账的接口transfer(X, Y)，X和Y都是互斥参数。

- **互斥对象**：一笔**交易**中，根据互斥参数提取出来的、具体的互斥内容。例如转账的接口transfer(X, Y)，一笔调用此接口的交易中，具体的参数是transfer(A, B)，则这笔操作的互斥对象是[A, B]；另外一笔交易，调用的参数是transfer(A, C)，则这笔操作的互斥对象是[A, C]。

判断同一时刻两笔交易是否能并行执行，就是判断两笔交易的互斥对象是否有交集。相互之间交集为空的交易可并行执行。

FISCO BCOS

编写并行合约

FISCO BCOS提供了**可并行合约开发框架**，开发者只需按照框架的规范开发合约，定义好每个合约接口的互斥参数，即可实现能被并行执行的合约。当合约被部署后，FISCO BCOS会在执行交易前，自动解析互斥对象，在同一时刻尽可能让无依赖关系的交易并行执行。

目前，FISCO BCOS提供了solidity与预编译合约（点击可查看预编译合约架构设计）两种可并行合约开

发框架。

solidity合约的并行框架

编写并行的solidity合约，开发流程与开发普通solidity合约流程相同。在此基础上，只需将ParallelContract 作为需要并行的合约基类，并调用registerParallelFunction()，注册可以并行的接口即可。

先给出完整的举例。例子中的ParallelOk合约实现了并行转账的功能：

```
pragma solidity ^0.4.25;

import "./ParallelContract.sol";  // 引入ParallelContract.sol

contract ParallelOk is ParallelContract // 将ParallelContract 作为基类
{
    // 合约实现
    mapping (string => uint256) _balance;

    function transfer(string from, string to, uint256 num) public
{
        // 此处为简单举例，实际生产中请用SafeMath代替直接加减
        _balance[from] -= num;
        _balance[to] += num;
    }

    function set(string name, uint256 num) public
{
        _balance[name] = num;
    }

    function balanceOf(string name) public view returns (uint256)
{
        return _balance[name];
    }

    // 注册可以并行的合约接口
```

```
    function enableParallel() public
{
        // 函数定义字符串（注意","后不能有空格），参数的前几个是互斥参数（设计函数时
        registerParallelFunction("transfer(string,string,uint256)", 2
        registerParallelFunction("set(string,uint256)", 1); // 冲突参数
    }

    // 注销并行合约接口
    function disableParallel() public
{
        unregisterParallelFunction("transfer(string,string,uint256)")
        unregisterParallelFunction("set(string,uint256)");
    }
}
```

具体步骤如下：

step1

将ParallelContract作为合约的基类

```
pragma solidity ^0.4.25;

import "./ParallelContract.sol"; // 引入ParallelContract.sol

contract ParallelOk is ParallelContract // 将ParallelContract 作为基类
{
   // 合约实现

   // 注册可以并行的合约接口
   function enableParallel() public;

   // 注销并行合约接口
   function disableParallel() public;
}
```

step2

编写可并行的合约接口

合约中的public函数，是合约的接口。编写可并行的合约接口，是根据一定的规则，实现一个合约中的public函数。

确定接口是否可并行

可并行的合约接口，必须满足：

- 无调用外部合约
- 无调用其它函数接口

确定互斥参数

在编写接口前，先确定接口的互斥参数，接口的互斥即是对全局变量的互斥，互斥参数的确定规则为：

- 接口访问了全局mapping，mapping的key是互斥参数
- 接口访问了全局数组，数组的下标是互斥参数
- 接口访问了简单类型的全局变量，所有简单类型的全局变量共用一个互斥参数，用不同的变量名作为互斥对象

确定参数类型和顺序

确定互斥参数后，根据规则确定参数类型和顺序，规则为：

- 接口参数仅限：string、address、uint256、int256（未来会支持更多类型）
- 互斥参数必须全部出现在接口参数中
- 所有互斥参数排列在接口参数的最前

```
mapping (string => uint256) _balance; // 全局mapping

// 互斥变量from、to排在最前，作为transfer()开头的两个参数
function transfer(string from, string to, uint256 num) public
{
    _balance[from] -= num;  // from 是全局mapping的key，是互斥参数
    _balance[to] += num; // to 是全局mapping的key，是互斥参数
}

// 互斥变量name排在最前，作为set()开头的参数
function set(string name, uint256 num) public
{
    _balance[name] = num;
}
```

step3

在框架中注册可并行的合约接口

在合约中实现 enableParallel() 函数，调用registerParallelFunction()注册可并行的合约接口。同时也需要实现disableParallel()函数，使合约具备取消并行执行的能力。

```
// 注册可以并行的合约接口
function enableParallel() public
{
    // 函数定义字符串（注意","后不能有空格），参数的前几个是互斥参数
    registerParallelFunction("transfer(string,string,uint256)", 2); //
    registerParallelFunction("set(string,uint256)", 1); // transfer接口
}

// 注销并行合约接口
function disableParallel() public
{
    unregisterParallelFunction("transfer(string,string,uint256)");
    unregisterParallelFunction("set(string,uint256)");
}
```

step4

部署/执行并行合约

用控制台或Web3SDK编译和部署合约，此处以控制台为例：

部署合约

```
[group:1]> deploy ParallelOk.sol
```

调用 enableParallel()接口，让ParallelOk能并行执行

```
[group:1]> call ParallelOk.sol 0x8c17cf316c1063ab6c89df875e96c9f0f5b2f
```

发送并行交易 set()

```
[group:1]> call ParallelOk.sol 0x8c17cf316c1063ab6c89df875e96c9f0f5b2f
```

发送并行交易 transfer()

```
[group:1]> call ParallelOk.sol 0x8c17cf316c1063ab6c89df875e96c9f0f5b2f
```

查看交易执行结果 balanceOf()

```
[group:1]> call ParallelOk.sol 0x8c17cf316c1063ab6c89df875e96c9f0f5b2f
80000
```

用SDK发送大量交易的例子，将在下文的例子中给出。

预编译合约的并行框架

编写并行的预编译合约，开发流程与开发普通预编译合约流程相同。普通的预编译合约以Precompile为基类，在这之上实现合约逻辑。基于此，Precompile的基类还为并行提供了两个虚函数，继续实现这两个函数，即可实现并行的预编译合约。

step1

将合约定义成支持并行

```
bool isParallelPrecompiled() override { return true; }
```

step2

定义并行接口和互斥参数

注意，一旦定义成支持并行，所有的接口都需要进行定义。若返回空，表示此接口无任何互斥对象。互斥参数与预编译合约的实现相关，此处涉及对FISCO BCOS存储的理解，具体的实现可直接阅读代码或询问相关有经验的程序员。

```
// 根据并行接口，从参数中取出互斥对象，返回互斥对象
std::vector<std::string> getParallelTag(bytesConstRef param) override
{
    // 获取被调用的函数名（func）和参数（data）
    uint32_t func = getParamFunc(param);
    bytesConstRef data = getParamData(param);

    std::vector<std::string> results;
    if (func == name2Selector[DAG_TRANSFER_METHOD_TRS_STR2_UINT]) //
    {
        // 接口为：userTransfer(string,string,uint256)
        // 从data中取出互斥对象
        std::string fromUser, toUser;
        dev::u256 amount;
        abi.abiOut(data, fromUser, toUser, amount);

        if (!invalidUserName(fromUser) && !invalidUserName(toUser))
        {
            // 将互斥对象写到results中
            results.push_back(fromUser);
            results.push_back(toUser);
        }
    }
    else if ... // 所有的接口都需要给出互斥对象，返回空表示无任何互斥对象

   return results;  //返回互斥
}
```

step3

编译，重启节点

手动编译节点的方法，参考FISCO BCOS技术文档：
https://fisco-bcos-documentation.readthedocs.io/zh_CN/latest/docs/manual/get_executable.html#id2

编译之后，关闭节点，替换掉原来的节点二进制文件，再重启节点即可。

举例：并行转账

此处分别给出solidity合约和预编译合约的并行举例。

配置环境

该举例需要以下执行环境：

- Web3SDK客户端
- 一条FISCO BCOS链

若需要压测最大的性能，至少需要：

- 3个Web3SDK，才能产生足够多的交易
- 4个节点，且所有Web3SDK都配置了链上所有的节点信息，让交易均匀发送到每个节点上，才能让链接收足够多的交易

并行Solidity合约：ParallelOk

基于账户模型的转账，是一种典型的业务操作。ParallelOk合约，是账户模型的一个举例，能实现并行的转账功能。ParallelOk合约已在上文中给出。

FISCO BCOS在Web3SDK中内置了ParallelOk合约，此处给出用Web3SDK来发送大量并行交易的操作方法。

step1

用SDK部署合约、新建用户、开启合约的并行能力

```
# 参数：<groupID> add <创建的用户数量> <此创建操作请求的TPS> <生成的用户信息文件
java -cp conf/:lib/*:apps/* org.fisco.bcos.channel.test.parallel.paral
# 在group1上创建了 10000个用户，创建操作以2500TPS发送的，生成的用户信息保存在user
```

执行成功后，ParallelOk被部署到区块链上，创建的用户信息保存在user文件中，同时开启了ParallelOk的并行能力。

step2

批量发送并行转账交易

注意：在批量发送前，请将SDK的日志等级调整为ERROR，才能有足够的发送能力。

```
# 参数：<groupID> transfer <总交易数量> <此转账操作请求的TPS上限> <需要的用户信
java -cp conf/:lib/*:apps/* org.fisco.bcos.channel.test.parallel.paral

# 向group1发送了 100000比交易，发送的TPS上限是4000，用的之前创建的user文件里的用
```

step3

验证并行正确性

并行交易执行完成后，Web3SDK会打印出执行结果。TPS 是此SDK发送的交易在节点上执行的TPS。validation 是转账交易执行结果的检查。

```
Total transactions:  100000
Total time: 34412ms
TPS: 2905.9630361501804
Avg time cost: 4027ms
Error rate: 0%
Return Error rate: 0%
Time area:
0    < time <  50ms   : 0  : 0.0%
50   < time <  100ms  : 44  : 0.04400000000000004%
100  < time <  200ms  : 2617  : 2.617%
200  < time <  400ms  : 6214  : 6.214%
400  < time <  1000ms : 14190  : 14.19%
1000 < time <  2000ms : 9224  : 9.224%
2000 < time           : 67711  : 67.711%
validation:
   user count is 10000
   verify_success count is 10000
   verify_failed count is 0
```

可以看出，本次交易执行的TPS是2905。执行结果校验后，无任何错误(verify_failed count is 0)。

step4

计算总TPS

单个Web3SDK无法发送足够多的交易以达到节点并行执行能力的上限。需要多个Web3SDK同时发送交易。在多个Web3SDK同时发送交易后，单纯将结果中的TPS加和得到的TPS不够准确，需要直接从节点处获取TPS。

用脚本从日志文件中计算TPS

```
cd tools
sh get_tps.sh log/log_2019031821.00.log 21:26:24 21:26:59 # 参数：<日志文
```

得到TPS（3 SDK、4节点，8核，16G内存）

```
statistic_end = 21:26:58.631195
statistic_start = 21:26:24.051715
total transactions = 193332, execute_time = 34580ms, tps = 5590 (tx/s)
```

并行预编译合约：DagTransferPrecompiled

与ParallelOk合约的功能一样，FISCO BCOS内置了一个并行预编译合约的例子（DagTransferPrecompiled），实现了简单的基于账户模型的转账功能。该合约能够管理多个用户的存款，并提供一个支持并行的transfer接口，实现对用户间转账操作的并行处理。

注意：DagTransferPrecompiled仅做示例使用，请勿直接运用于生产环境。

step1

生成用户

用Web3SDK发送创建用户的操作，创建的用户信息保存在user文件中。命令参数与parallelOk相同，不同的仅仅是命令所调用的对象是precompile。

```
java -cp conf/:lib/*:apps/* org.fisco.bcos.channel.test.parallel.preco
```

step2

批量发送并行转账交易

用Web3SDK发送并行转账交易。

注意：在批量发送前，请将SDK的日志等级请调整为ERROR，才能有足够的发送能力。

```
java -cp conf/:lib/*:apps/* org.fisco.bcos.channel.test.parallel.precor
```

step3

验证并行正确性

并行交易执行完成后，Web3SDK会打印出执行结果。TPS 是此SDK发送的交易在节点上执行的TPS。validation 是转账交易执行结果的检查。

```
Total transactions:  80000
Total time: 25451ms
TPS: 3143.2949589407094
Avg time cost: 5203ms
Error rate: 0%
Return Error rate: 0%
Time area:
0    < time <  50ms   : 0  : 0.0%
50   < time <  100ms  : 0  : 0.0%
100  < time <  200ms  : 0  : 0.0%
200  < time <  400ms  : 0  : 0.0%
400  < time <  1000ms : 403  : 0.50375%
1000 < time <  2000ms : 5274  : 6.592499999999999%
2000 < time           : 74323  : 92.90375%
validation:
    user count is 10000
    verify_success count is 10000
    verify_failed count is 0
```

可以看出，本次交易执行的TPS是3143。执行结果校验后，无任何错误(verify_failed count is 0)。

step4

计算总TPS

单个Web3SDK无法发送足够多的交易以达到节点并行执行能力的上限。需要多个Web3SDK同时发送交易。在多个Web3SDK同时发送交易后，单纯将结果中的TPS加和得到的TPS不够准确，需要直接从节点处获取TPS。

用脚本从日志文件中计算TPS

```
cd tools
sh get_tps.sh log/log_2019031311.17.log 11:25 11:30 # 参数：<日志文件> <i
```

得到TPS（3 SDK、4节点，8核，16G内存）

```
statistic_end = 11:29:59.587145
statistic_start = 11:25:00.642866
total transactions = 3340000, execute_time = 298945ms, tps = 11172 (tx
```

结果说明

本文举例中的性能结果，是在3SDK、4节点、8核、16G内存、1G网络下测得。每个SDK和节点都部署在不同的VPS中，硬盘为云硬盘。实际TPS会根据你的硬件配置、操作系统和网络带宽有所变化。

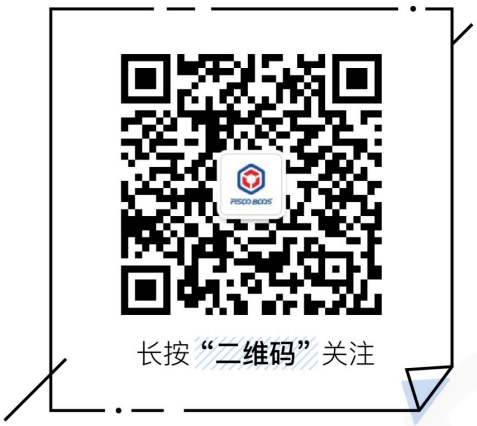
如您在部署过程中遇到阻碍或有问题需要咨询，可以进入FISCO BCOS官方技术交流群寻求解答。（进群请长按下方二维码识别添加小助手）

ID：fiscobcosfan

FISCO BCOS的代码完全开源且免费

下载地址↓↓↓

https://github.com/fisco-bcos



更多开发教程戳阅读原文

向右→给我一朵小花花

阅读原文