

以太坊智能合约性能测试实验报告

0. 环境配置

caliper有两种安装方式，一种是通过npm，一种是通过docker

0.1 前置条件

- node.js需要V14 LTS或者V16 LTS才能从NPM安装caliper CLI
- 需要Docker版本20.10.11或更高才能使用caliper

可能需要以下该工具，具体取决于您绑定到的 SUT 和版本

- python3, g++和git (用于在绑定期间获取和编译一些包)

Pre-requisites

- Node.js v14 LTS or v16 LTS is required to install the Caliper CLI from NPM:
- Docker version 20.10.11 or later is required for use with the Caliper docker image

The following tools may be required depending on which SUT and version you bind to

- python3, make, g++ and git (for fetching and compiling some packages during bind)

1.运行示例项目

1.1 本地NPM安装和运行

在安装之前，需要从GitHub中克隆caliper-benchmark项目。以下命令需要在caliper-benchmark目录下执行。

注意：这是为项目安装caliper的强烈建议方法。将项目依赖项保留在本地可以更轻松地设置多个caliper项目。在运行新的基准测试之前，全局依赖项每次都需要重新绑定（以确保正确的全局依赖项）。

1. 像安装任何其他 NPM 软件包一样安装caliper CLI。强烈建议明确指定版本号，例如：
`@hyperledger/caliper-cli@0.5.0`
2. 将 CLI 绑定到所需的平台开发工具包（例如，使用开发工具包）。caliper支持以下版本的以太坊版本：`1.2.1` `1.3`
3. 使用适当的参数调用本地 CLI 二进制文件（使用 `npx`）。

命令如下：

```
npm install --only=prod @hyperledger/caliper-cli@0.5.0

npx caliper bind --caliper-bind-sut ethereum

npx caliper launch manager \
--caliper-workspace . \
--caliper-benchconfig benchmarks/scenario/simple/config.yaml \
--caliper-networkconfig networks/ethereum/1node-clique/networkconfig.json
```

这里指定的参数意义是：

- `--caliper-workspace` 指定工作区的路径
- `--caliper-networkconfig` 指定网络配置文件的路径
- `--caliper-benchconfig` 指定基准配置文件的路径

如果一切配置正确，输入以上命令后，caliper就会开始运行示例的**simple**合约，并使用实例中预制好的工作负载。

测试结果会生成在caliper-benchmark目录下的report.html文件中，同时也会直接显示在控制台中

Caliper report

Summary of performance metrics

Name	Succ	Fail	Send Rate (TPS)	Max Latency (s)	Min Latency (s)	Avg Latency (s)	Throughput (TPS)
open	1000	0	50.1	27.64	2.07	14.93	21.1
query	1000	0	100.1	0.00	0.00	0.00	100.1
transfer	50	0	5.1	6.89	2.08	4.48	3.5

Benchmark round: open

Test description for the opening of an account through the deployed contract.

```
rateControl:
  type: fixed-rate
  opts:
    tps: 50
```

Performance metrics for open

Name	Succ	Fail	Send Rate (TPS)	Max Latency (s)	Min Latency (s)	Avg Latency (s)	Throughput (TPS)
open	1000	0	50.1	27.64	2.07	14.93	21.1

Resource utilization for open

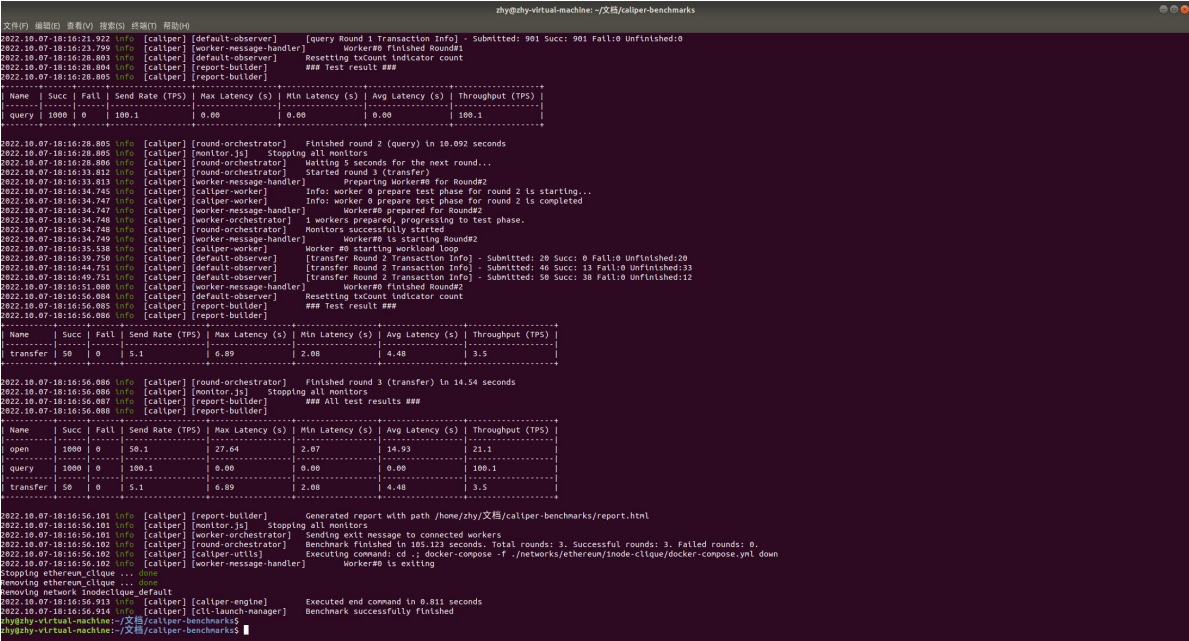
Benchmark round: query

Test description for the query performance of the deployed contract.

```
rateControl:
  type: fixed-rate
  opts:
    tps: 100
```

Performance metrics for query

Name	Succ	Fail	Send Rate (TPS)	Max Latency (s)	Min Latency (s)	Avg Latency (s)	Throughput (TPS)
query	1000	0	100.1	0.00	0.00	0.00	100.1



2. 配置caliper

由上面的启动命令可以知道，caliper有以下几个关键的配置文件：网络配置文件network-config，基准测试配置文件benchmark-config，根据启动命令可以找到这些配置文件所在的路径。

2.1 网络配置文件

参考资料: <https://hyperledger.github.io/caliper/v0.5.0/ethereum-config/>

因为我们使用的是以太坊，所以网络配置文件可以在 `caliper-benchmarks/networks/ethereum/1node-clique` 中找到，文件名为 `networkconfig.json`。内容如下

```
networkconfig.json
~/文档/caliper-benchmarks/networks/ethereum/1node-clique

{
  "caliper": {
    "blockchain": "ethereum",
    "command": {
      "start": "docker-compose -f ./networks/ethereum/1node-clique/docker-compose.yml up -d && sleep 6",
      "end": "docker-compose -f ./networks/ethereum/1node-clique/docker-compose.yml down"
    },
    "ethereum": {
      "url": "ws://localhost:8546",
      "contractDeployerAddress": "0xc0A8e4D217eB85b812aeb1226fAb6F588943C2C2",
      "contractDeployerAddressPassword": "password",
      "fromAddress": "0xc0A8e4D217eB85b812aeb1226fAb6F588943C2C2",
      "fromAddressPassword": "password",
      "transactionConfirmationBlocks": 12,
      "contracts": {
        "simple": {
          "path": "./src/ethereum/simple/simple.json",
          "estimateGas": true,
          "gas": {
            "open": 45000,
            "query": 100000,
            "transfer": 70000
          }
        }
      }
    }
  }
}
```

其中：

- `url` 是要连接到的节点的 URL。但是只支持web socket，HTTP连接是不允许的
- `contractDeployerAddress` 是部署者地址，如果没有特定或特定的需求，可以将其设置为等于[基准地址](#)。其私钥必须由与 [URL](#) 连接的节点持有，并且必须以校验和形式（同时具有小写和大写字母的节点）提供。
- `contractDeployerAddressPrivateKey`： [部署程序地址](#)的私钥。如果存在，则交易在caliper内签名，并将“raw”发送到以太坊节点。
- `contractDeployerAddressPassword`： 用于解锁[部署程序地址](#)的密码。如果没有解锁密码，则此密钥必须以空字符串形式存在。如果部署程序地址私钥存在，则不会使用此选项
- `fromAddressSeed`： 基准地址种子。网络配置可以使用固定的种子，并通过 [BIP-44](#) 密钥派生导出所需的地址。
- `fromAddressPrivateKey`： 基准测试地址私钥。如果存在，则交易在caliper内签名，并将“raw”发送到以太坊节点。
- `transactionConfirmationBlocks`： 确认块。它是适配器在警告caliper事务已在网络上成功执行之前将等待的块数。您可以自由地将其从1调整为所需的数量。请记住，在以太坊主网（PoW）中，可能需要12到20次确认才能将交易视为区块链中接受的交易。
- `contracts`： 合同配置。它是作为 json 对象提供的在运行基准测试之前要在网络上部署的合约的列表。您应该为每个合同提供一个 json 条目；键将表示用于调用该协定上的方法的协定标识符。

对于每个键，必须提供一个 JSON 对象，其中包含指向 `合约定义文件` 的字段——`path`

还强烈建议指定一个字段，该字段是一个对象，每个合约函数都有一个字段，您将在测试中调用该字段。这些字段的值应设置为执行事务所需的 gas 量。此数字不需要完全匹配，因为它用于设置交易的 gas 限制，因此，如果您的交易可能具有可变的 gas 成本，只需将此值设置为您希望在交易中看到的最高 gas 使用量即可。

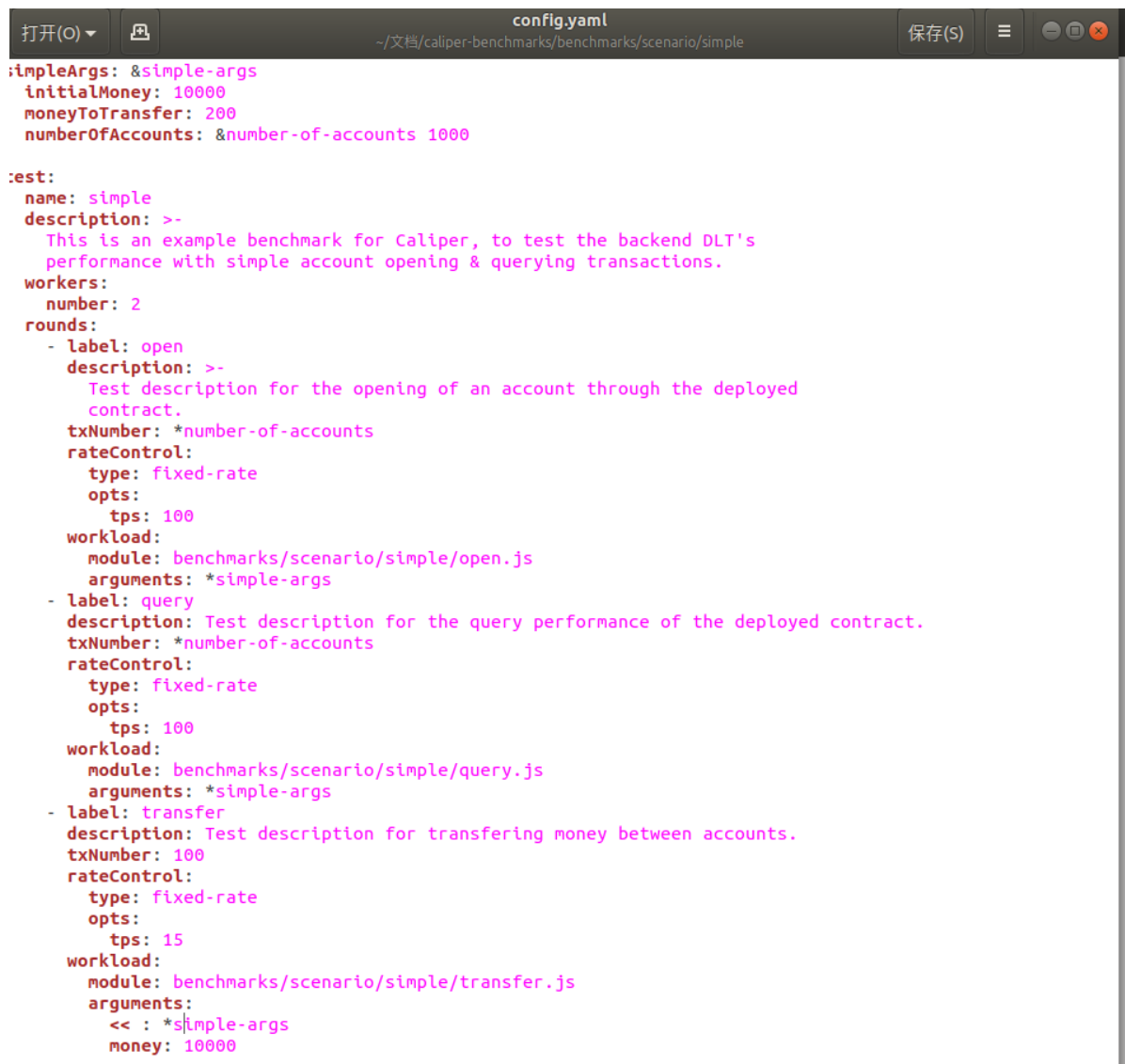
- 合约定义文件：合约定义文件是一个简单的JSON文件，其中包含部署和使用以太坊合约的基本信息。需要四个密钥：名字，abi，字节码和gas的量。合约定义文件是sol文件通过solidity编译生成的，通过编写配置文件，可以将编译结果筛选出需要的部分，即abi和字节码，并写入一个json文件中。也可以通过solcjs指令快捷生成abi和字节码。具体可见文末编译部分

2.2 基准配置文件

参考资料：<https://hyperledger.github.io/caliper/v0.5.0/bench-config/>

<https://zhuanlan.zhihu.com/p/438726200>

基准配置文件的路径是 `caliper-benchmarks/benchmarks/scenario/simple` 文件名为 `config.yaml`



```
simpleArgs: &simple-args
  initialMoney: 10000
  moneyToTransfer: 200
  numberOfAccounts: &number-of-accounts 1000

test:
  name: simple
  description: >-
    This is an example benchmark for Caliper, to test the backend DLT's
    performance with simple account opening & querying transactions.
  workers:
    number: 2
  rounds:
    - label: open
      description: >-
        Test description for the opening of an account through the deployed
        contract.
      txNumber: *number-of-accounts
      rateControl:
        type: fixed-rate
        opts:
          tps: 100
      workload:
        module: benchmarks/scenario/simple/open.js
        arguments: *simple-args
    - label: query
      description: Test description for the query performance of the deployed contract.
      txNumber: *number-of-accounts
      rateControl:
        type: fixed-rate
        opts:
          tps: 100
      workload:
        module: benchmarks/scenario/simple/query.js
        arguments: *simple-args
    - label: transfer
      description: Test description for transferring money between accounts.
      txNumber: 100
      rateControl:
        type: fixed-rate
        opts:
          tps: 15
      workload:
        module: benchmarks/scenario/simple/transfer.js
        arguments:
          << : *simple-args
          money: 10000
```

该配置文件的大致意思是：

- 有两个工作进程执行基准测试（workers: number）
- 将有三轮测试（rounds中有三个）
- 第一轮以固定的100tps的发送速率发送 `number-of-accounts` 个TX
- TX的内容由工作负载（workload: module）决定
- 第二轮将以固定的100tps的发送速率发送 `number-of-accounts` 个TX
- TX的内容由工作负载（workload: module）决定
- 第三轮....

2.3 工作负载

参考资料: <https://zhuanlan.zhihu.com/p/438726200>

<https://hyperledger.github.io/caliper/v0.5.0/workload-module/>


工作负载模块是 Caliper 基准测试的本质, 因为它们负责构建和提交 TX。将工作负载模块视为模拟 SUT 客户端的大脑, 决定在给定时刻提交哪种 TX。

工作负载模块是公开特定 API 的节点JS模块。实现没有进一步的限制, 因此可以实现任意逻辑 (使用更多任意组件)。

工作负载通常需要包含三个异步函数:

1. 初始化工作负载模块
2. 提交事务
3. 清理工作负载模块

示例中的工作负载配置路径为 `caliper-benchmarks/benchmarks/scenario/simple/utis`, 实际上根据基准测试的配置文件可以看出, 每一轮测试都有自己的工作负载, 对于示例来说就是 `open.js`、`query.js`和`transfer.js`, 打开其中任意一个工作负载文件, 都可以看到他们引入并使用了`utis`文件夹中的 `operation-base.js` 和 `simple-state.js` 两个文件。而 `operation-base.js` 又是继承自`caliper`内核中自带的 `workloadModuleBase` 类。



```
transfer.js
~/文档/caliper-benchmarks/benchmarks/scenario/simple

/*
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 * http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

'use strict';

const OperationBase = require('./utis/operation-base');
const SimpleState = require('./utis/simple-state');

/**
 * Workload module for transferring money between accounts.
 */
class Transfer extends OperationBase {

    /**
     * Initializes the instance.
     */
    constructor() {
        super();
    }

    /**
     * Create a pre-configured state representation.
     * @return {SimpleState} The state instance.
     */
    createSimpleState() {
        const accountsPerWorker = this.numberOfAccounts / this.totalWorkers;
        return new SimpleState(this.workerIndex, this.initialMoney, this.moneyToTransfer,
            accountsPerWorker);
    }

}
```

```

打开(O)  operation-base.js  保存(S)
~/文档/caliper-benchmarks/benchmarks/scenario/simple/utis
See the License for the specific language governing permissions and
limitations under the License.
/

use strict';

const { WorkloadModuleBase } = require('@hyperledger/caliper-core');

const SupportedConnectors = ['ethereum', 'fabric'];

/**
 * Base class for simple operations.
 */
class OperationBase extends WorkloadModuleBase {
  /**
   * Initializes the base class.
   */
  constructor() {
    super();
  }

  /**
   * Initialize the workload module with the given parameters.
   * @param {number} workerIndex The 0-based index of the worker instantiating the workload module.
   * @param {number} totalWorkers The total number of workers participating in the round.
   * @param {number} roundIndex The 0-based index of the currently executing round.
   * @param {Object} roundArguments The user-provided arguments for the round from the benchmark
   configuration file.
   * @param {ConnectorBase} sutAdapter The adapter of the underlying SUT.
   * @param {Object} sutContext The custom context object provided by the SUT adapter.
   * @async
   */
  async initializeWorkloadModule(workerIndex, totalWorkers, roundIndex, roundArguments, sutAdapter,
    sutContext) {
    await super.initializeWorkloadModule(workerIndex, totalWorkers, roundIndex, roundArguments,
      sutAdapter, sutContext);

    this.assertConnectorType();
    this.assertSetting('initialMoney');
    this.assertSetting('moneyToTransfer');
    this.assertSetting('numberOfAccounts');

    this.initialMoney = this.roundArguments.initialMoney;
    this.moneyToTransfer = this.roundArguments.moneyToTransfer;
    this.numberOfAccounts = this.roundArguments.numberOfAccounts;
    this.simpleState = this.createSimpleState();
  }
}
/**

```



```
simple-state.js
~/文档/caliper-benchmarks/benchmarks/scenario/simple/utlis

/*
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 * http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

'use strict';

const Dictionary = 'abcdefghijklmnopqrstuvwxyz';

/**
 * Class for managing simple account states.
 */
class SimpleState {
  /**
   * Initializes the instance.
   */
  constructor(workerIndex, initialMoney, moneyToTransfer, accounts = 0) {
    this.accountsGenerated = accounts;
    this.initialMoney = initialMoney;
    this.moneyToTransfer = moneyToTransfer;
    this.accountPrefix = this._get26Num(workerIndex);
  }

  /**
   * Generate string by picking characters from the dictionary variable.
   * @param {number} number Character to select.
   * @returns {string} string Generated string based on the input number.
   * @private
   */
}
```

工作负载代码内容和如何编写可以看上面列出的参考资料。

目前来看，我们需要考虑的与工作负载相关内容，就是合约中每一个函数的名字、参数等

2.3.1 工作负载与合约中的函数的关系

示例程序非常简单，只有三个函数，其路径为 `caliper-benchmarks/src/ethereum/simple`

下图是合约代码，其文件名称为 `simple.sol`，我为了测试修改过一部分，所以与原本的示例程序有所不同。

可以看到三个函数分别是，`open` `query` 和 `transfer`，分别有两个参数、一个参数和四个参数（原本应该是三个，我自己又加了一个）

```
simple.sol
~/文档/caliper-benchmarks/src/ethereum/simple

pragma solidity >=0.4.22 <0.6.0;

contract simple {
  mapping(string => int) private accounts;

  function open(string memory acc_id, int amount) public {
    accounts[acc_id] = amount;
  }

  function query(string memory acc_id) public view returns (int amount) {
    amount = accounts[acc_id];
  }

  function transfer(string memory acc_from, string memory acc_to, int amount, string memory acc_id)
  public {
    accounts[acc_from] -= amount;
    accounts[acc_to] += amount;
    accounts[acc_id] += 1;
    accounts[acc_id] -= 1;
  }
}
```

打开它们对应的工作负载文件，可以看到一些共同点，就用 `transfer.js` 举例

```

    /**
     * @return {SimpleState} the state instance.
     */
    createSimpleState() {
        const accountsPerWorker = this.numberOfAccounts / this.totalWorkers;
        return new SimpleState(this.workerIndex, this.initialMoney, this.moneyToTransfer,
            accountsPerWorker);
    }

    /**
     * Assemble TXs for transferring money.
     */
    async submitTransaction() {
        const transferArgs = this.simpleState.getTransferArguments();
        await this.sutAdapter.sendRequests(this.createConnectorRequest('transfer', transferArgs));
    }
}

/**
 * Create a new instance of the workload module.
 * @return {WorkloadModuleInterface}
 */
function createWorkloadModule() {
    return new Transfer();
}

```

这是 transfer 函数的工作负载文件，与其他工作负载文件大同小异。他们都会创建一个 SimpleState 类的实例，并初始化一些数据，然后有一个异步函数 submitTransaction()，其中调用了 SimpleState 类实例对象的一个方法：getTransferArguments()，根据字面意思我们可以知道，这个方法与获取 transfer 函数的参数有关。在这之后工作负载文件又将利用包装好的 transfer 函数发起了请求。

打开 simple-state.js 文件，我们就可以看到工作负载中是如何确定合约中函数的参数的

```

/**
 * Get the arguments for transferring money between accounts.
 * @returns {object} The account arguments.
 */
getTransferArguments() {
    return {
        source: this._getRandomAccount(),
        target: this._getRandomAccount(),
        amount: this.moneyToTransfer,
        account: this._getRandomAccount()
    };
}
}

```

可以看到在 transfer.js 中调用的 SimpleState 类中的 getTransferArguments() 方法，返回了四个参数，这与合约代码 simple.sol 中相对应

```

function transfer(string memory acc_from, string memory acc_to, int amount, string memory acc_id)
public {
    accounts[acc_from] -= amount;
    accounts[acc_to] += amount;
    accounts[acc_id] += 1;
    accounts[acc_id] -= 1;
}
}

```

而这些参数的实际数据，其中账户是由 SimpleState 的成员函数 _getRandomAccount() 产生的，金额是写在基准配置文件，也就是 config.yaml 中的


```

    /
    _getAccountKey(index) {
        return this.accountPrefix + this._get26Num(index);
    }

    /**
     * Returns a random account key.
     * @return {string} Account key.
     * @private
     */
    _getRandomAccount() {
        // choose a random TX/account index based on the existing range, and restore the account name
        from the fragments
        const index = Math.ceil(Math.random() * this.accountsGenerated);
        return this._getAccountKey(index);
    }

```

simple-state.js 中其他两个函数 open 和 query 的参数获取如下，这都与合约文件 simple.sol 中函数的参数一一对应

所以在测试前，需要配置好合约中需要测试的函数的工作负载，使他们的名字、参数都能一一对应。修改参数后，使用 `solidity` 对合约进行重新编译，并确保新生成的合约定义文件与网络配置文件绑定。

该功能是工作负载生成的主干。每次速率控制器启用下一个 TX 时，工作进程都会调用此函数。因此，为了能够跟上高频调度设置，尽可能保持此函数实现的效率至关重要。

```
let requestsSettings = [{
  contract: 'simple',
  verb: 'open',
  value: 1000000000000000000000,
  args: ['sfogliatella', 1000]
},{
  contract: 'simple',
  verb: 'open',
  value: 900000000000000000000,
  args: ['baba', 900]
}];

await this.sutAdapter.sendRequests(requestsSettings);
```

设置对象具有以下结构：

- `contract`：字符串。必填。协定的 ID（[即此处](#)指定的密钥）。
- `readonly`：布尔值。自选。指示请求是 TX 还是查询。默认为 `false`。
- `verb`：字符串。必填。要调用协定的函数的名称。
- `value`：数字。自选。Wei 中要传递给合同的支付函数的值参数。
- `args`：[]。自选。要以方法签名中显示的正确顺序传递给方法的参数列表。它必须是一个数组。

3. 编译合约.sol文件

编译合约代码需要使用 `solidity`

参考资料：<http://t.zoukankan.com/YpfBoI-g-p-14787678.html>

网络配置文件中说到，我们需要配置好合约定义文件，该文件中由四个部分，其中 `abi` 和 `字节码` 是 `solidity` 编译生成。

控制台打印编译后所有输出的结果，使用工具调整格式后如下：

```
{
  "contracts": {
    "Storage.sol": {
      "Storage": {
        "abi": [...],
        "devdoc": {"details": "Store & retrieve value in a variable..."},
        "evm": {"assembly": " /* \"Storage.sol\":147:354 contract Storage {\r... */\n mstore(0x40, 0x80)\n",
        "ewasm": {"wasm": "..."},
        "metadata": "{\"compiler\":{\"version\":\"0.8.4+commit.c7e474f2\"},\"language\":\"Solidity\",\"output\""},
        "storageLayout": {...},
        "userdoc": {
          "kind": "user",
          "methods": {},
          "version": 1
        }
      }
    }
  },
  "sources": {...}
}
```

编译后的输入输出json中各字段的含义，可以查看官方中文文档：<https://solidity-cn.readthedocs.io/zh/develop/using-the-compiler.html#id5>

3.1 使用脚本文件进行编译

`solidity` 编译会生成很多东西，但是我们需要一个 `js` 脚本文件筛选我们需要的东西

我创建了一个名为 `compile.js` 的脚本，脚本内容如下

```
const fs=require("fs");
const solc = require("solc");
const path = require("path");
const contractPath = path.resolve(__dirname, "../ethereum", "simple.sol");
const contractSource = fs.readFileSync(contractPath, "utf-8");
//预先定义编译源输入json对象
let jsonContractSource = JSON.stringify({
  language: 'Solidity',
  sources: {
    'simple.sol': { // 指明编译的文件名，方便获取数据
      content: contractSource, // 加载的合约文件源代码
    },
  },
},
```

```

        settings: { // 自定义编译输出的格式。以下选择输出全部结果。
            outputSelection: {
                '*': {
                    '*': [ '*' ]
                }
            }
        },
    });
    const result = JSON.parse(solc.compile(jsonContractSource));
    if(Array.isArray(result.errors) && result.errors.length){
        console.log(result.errors);
    }
    storageJson = {
        'abi': {},
        'bytecode': ''
    };
    //此时的simple.sol与输入的json对象中定义的编译文件名相同
    storageJson.abi = result.contracts["simple.sol"]["simple"].abi;
    storageJson.bytecode = result.contracts["simple.sol"]
    ["simple"].evm.bytecode.object;
    //输出文件的路径
    const compilePath = path.resolve(__dirname, "../ethereum", "simple.json");
    //将abi以及bytecode数据输出到文件或者将整个result输出到文件
    fs.writeFile(compilePath, JSON.stringify(storageJson), function(err){
        if(err){
            console.error(err);
        }else{
            console.log("contract file compiled sucessfully.");
        }
    });
});

```

注意合约文件和输出文件的路径，我这里用的是我自己的路径

注意：示例给出的合约代码文件中，开头限制了solidity的版本，我装的是最新版，不在这个版本限制的范围内，如果你也是这样，那么最简单的办法就是删掉这个版本限制的代码

```
pragma solidity >=0.4.22 <0.6.0;
```

```
contract simple {
```

编写好脚本后，在脚本所在的路径下打开控制台，输入命令 `sudo node compile.js` 来使用node执行 compile.js

```
zhy@zhy-virtual-machine: ~/下载/ethereum
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
zhy@zhy-virtual-machine:~/下载/ethereum$ sudo node compile.js
[sudo] zhy 的密码:

{
  component: 'general',
  errorCode: '1878',
  formattedMessage: 'Warning: SPDX license identifier not provided in source file. Before publishing, consider adding a comment containing "SPDX-License-Identifier: <SPDX-License>" to each source file. Use "SPDX-License-Identifier: UNLICENSED" for non-open-source code. Please see https://spdx.org for more information\n' +
    '--> simple.sol\n' +
    '\n',
  message: 'SPDX license identifier not provided in source file. Before publishing, consider adding a comment containing "SPDX-License-Identifier: <SPDX-License>" to each source file. Use "SPDX-License-Identifier: UNLICENSED" for non-open source code. Please see https://spdx.org for more information.',
  severity: 'warning',
  sourceLocation: { end: -1, file: 'simple.sol', start: -1 },
  type: 'Warning'
}

Contract file compiled successfully.
```

```
simple.json
~/下载/ethereum
{"abi":[{"inputs":[{"internalType":"string","name":"acc_id","type":"string"}, {"internalType":"int256","name":"amount","type":"int256"}], "name":"open", "outputs": [], "stateMutability":"nonpayable", "type":"function"}, {"inputs": [{"internalType":"string","name":"acc_id","type":"string"}], "name":"query", "outputs": [{"internalType":"int256","name":"amount","type":"int256"}], "stateMutability":"view", "type":"function"}, {"inputs": [{"internalType":"string","name":"acc_from","type":"string"}, {"internalType":"string","name":"acc_to","type":"string"}, {"internalType":"int256","name":"amount","type":"int256"}], "name":"transfer", "outputs": [{"internalType":"string","name":"acc_id","type":"string"}], "type":"function"}, {"stateMutability":"nonpayable", "type":"function"}], "bytecode":"608060405234801561001057600080fd5b50610"}
保存(S)
```

不要忘了将该合约定义文件与网络配置文件绑定，需要在网络配置文件中给出 `simple.json` 的路径。

3.2 使用solcjs编译

使用以下指令安装solc

```
npm install solc
```

在合约源代码目录下输入以下指令

```
solcjs --abi ./simple.sol
solcjs --bin ./simple.sol
```

Caliper对abi.json文件内容有要求，合约定义文件中要求有以下四个关键字

1. Name：随意，我这里取Candy
2. ABI：上面-abi指令生成的内容
3. Bytecode：上面-bin指令生成的内容
4. Gas：一定要填写，可以在Remix上Deploy试一下看看花多少，或者用Simulation工具估算下，这个影响不大，不要太小就可以

然后就可以新建一个json文件取名simple.json，把内容拼接起来。注意bytecode是-bin生成的一串数字，前面要加0x前缀

4. 进行测试

4.1 本地测试

4.1.1 工作负载

在本地测试阶段，我选择了购买贡献度的合约作为测试合约。

在开始编写配置文件并开始测试之前，需要仔细阅读合约代码，确定函数之间的调用关系，确定需要测试的函数及其参数。

在和同学交流后发现，caliper目前是不支持测试有构造方法的合约的测试的（除非手动进行部署），严格意义上说，是不支持具有“有参数的构造方法”的合约，因为这个构造方法是在EVM实例化合约的时候自动运行的，而所有工作负载中的测试内容都要在该合约实例化之后运行，所以如果此时构造方法中需要参数，我们是没办法通过caliper向构造方法传递参数的。

所以，在测试时可以选择没有构造方法的合约，或者通过修改构造方法，将其需要的参数直接写在代码里，来规避上述问题。

```
1  // SPDX-License-Identifier: GPL-3.0
2
3  pragma solidity >=0.4.0 <0.9.0;
4
5  import "./dataType.sol";
6
7  contract Trade{
8      mapping(uint256 => project) projects;
9      uint256 public projectID =0;
10
11      constructor(){
12          createProject(0,0,0,0,0,0,10000,100,0,0,0);
13          projects[0].name="test project";
14      }
15
16      //项目初始化
17      function createProject ( uint256 voteInvolvedRate, uint256 voteAdoptedRate,
18          uint256 applyDuration, uint256 modifyDuration, uint256 codeReviewDuration,
19          uint256 linesCommitPerContri, uint256 weiPerContri, uint256 linesBuyPerContri,
20          uint256 contriThreshold, uint256 entryThreshold , uint256 bounsRate) public returns(uint256) {
```

例如图中的合约，就符合条件。其实这也是通过将需要的参数直接写在代码里来避免实例化Trade合约的时候需要给构造方法传参的问题。

```
function buyContribution(uint256 projectId) public payable{
  project storage pro = projects[projectId];
  uint256 contriToBuy = msg.value / pro.weiPerContri;//可以购买的贡献度
  if(!projects[projectId].contributors[msg.sender].isIn){//如果不在这个项目里面
    require(msg.value >= projects[projectId].entryThreshold);
    projects[projectId].contributors[msg.sender].isIn = true;
    projects[projectId].contributors[msg.sender].joinTime = block.timestamp;
    projects[projectId].contributors[msg.sender].credit = 100;
    projects[projectId].contributors[msg.sender].addr = msg.sender;
    projects[projectId].allContributors.push(msg.sender);
  }
  projects[projectId].contributors[msg.sender].contribution += contriToBuy;
  projects[projectId].contributors[msg.sender].balance += contriToBuy;
  projects[projectId].totalContri += contriToBuy;
  projects[projectId].profitBalance += msg.value;
  // profitDistribute(projectId,msg.value);
}
```

通过观察我们可以发现，我们需要测试的购买贡献度的函数 `buyContribution`，只需要一个参数：`projectId`。而在构造方法里，该合约初始化了一个id为0的“test project”，所以如果我们要测试 `buyContribution` 这个函数，只需要向其传递参数 `0` 即可。

现在按照2.3.2中描述的规则，创建一个工作负载文件 `Myworkload.js` 编写 `submitTransaction()` 函数

```
async submitTransaction() {
  let requests = {
    contract: "Trade",
    verb: "buyContribution",
    args: [0],
    readOnly: false
  };
  await this.sutAdapter.sendRequests(requests);
}
}
```

工作负载的完整内容如下：

```
'use strict';

const { workloadModuleBase } = require('@hyperledger/caliper-core');

class MyWorkload extends workloadModuleBase {
  constructor(){
    super();
  }

  async submitTransaction() {
    let requests = {
      contract: "Trade",
      verb: "buyContribution",
      args: [0],
      readOnly: false
    };
  }
}
```



```

    };
    await this.sutAdapter.sendRequests(requests);
  }
}

function createWorkloadModule() {
  return new MyWorkload();
}

module.exports.createWorkloadModule = createWorkloadModule;

```

4.1.2 基准配置文件

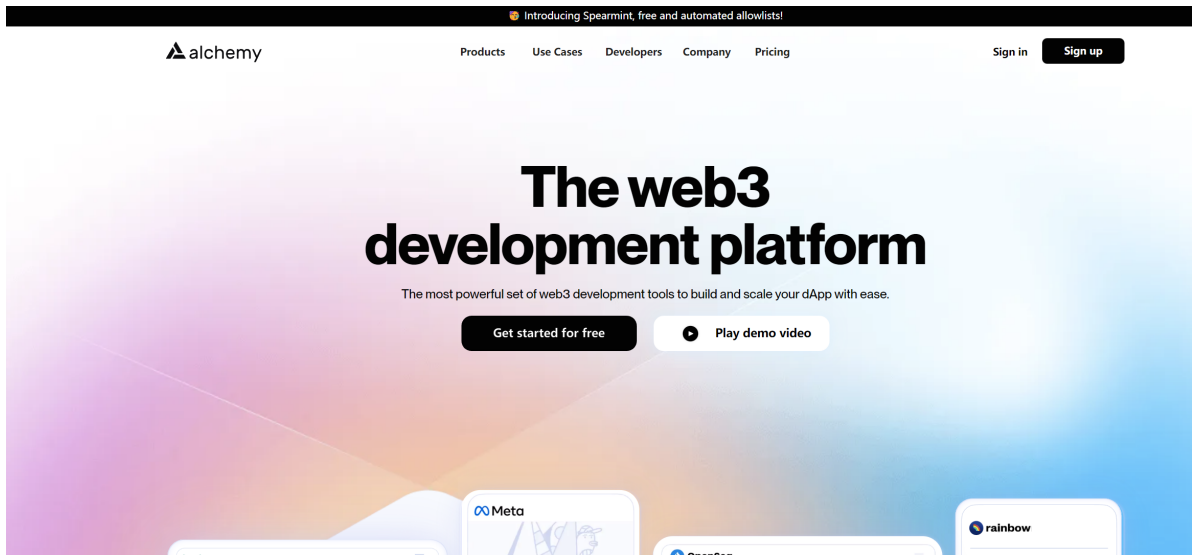
有了工作负载，现在就可以根据你的测试需要配置基准配置文件了

```

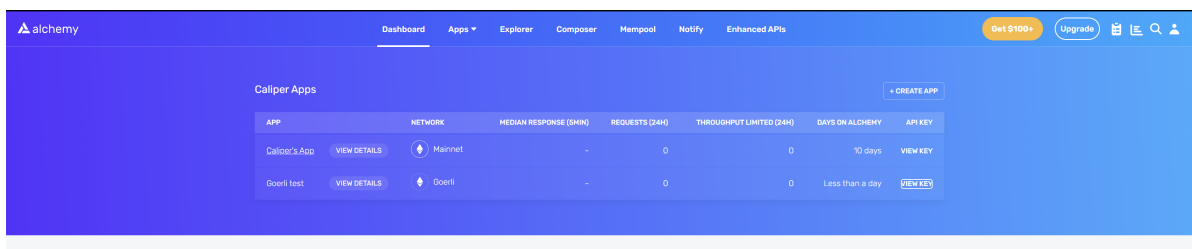
test:
  name: Trade
  workers:
    number: 2
  rounds:
    - label: buyContribution1
      txNumber: 1000
      rateControl:
        type: fixed-rate
        opts:
          tps: 100
      workload:
        module: TradeContract/MyWorkload.js
    - label: buyContribution2
      txNumber: 1000
      rateControl:
        type: fixed-rate
        opts:
          tps: 200
      workload:
        module: TradeContract/MyWorkload.js
    - label: buyContribution3
      txNumber: 1000
      rateControl:
        type: fixed-rate
        opts:
          tps: 300
      workload:
        module: TradeContract/MyWorkload.js
    - label: buyContribution4
      txNumber: 1000
      rateControl:
        type: fixed-rate
        opts:
          tps: 400
      workload:
        module: TradeContract/MyWorkload.js

```

大体内容如上，其中workload中的路径就是刚才我们编写的工作负载文件



访问主页后注册账号，然后进入如下界面：



在Caliper Apps中点击右侧的 VIEW KEY 获取WebSocket和API KEY

API KEY

VshvczcinTJAX5BS0pT...

Copy

HTTPS

https://eth-goerli.g.alchemy.com/v2/VshvczcinTJAX5BS0pT...

Copy

WEBSOCKETS

wss://eth-goerli.g.alchemy.com/v2/VshvczcinTJAX5BS0pT...

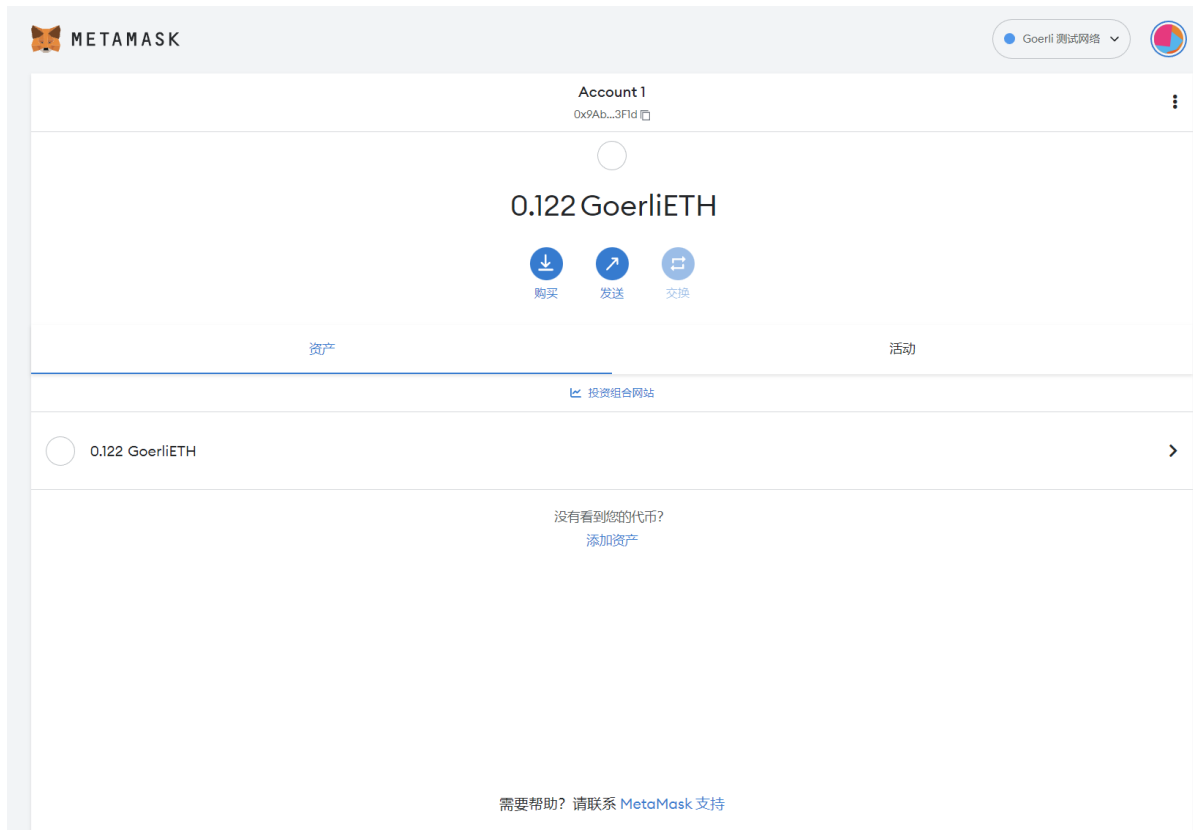
Copy

然后将WEBSOCKETS与API KEY组成network config新的url

```
{
  "caliper": {
    "blockchain": "ethereum",
    "command": {
      "start": "sleep 3"
    }
  },
  "ethereum": {
    "url": "wss://eth-goerli.g.alchemy.com/v2/VshvczcinTJAX5BS0pTTJqQ566pHzsLL/...",
    "contractDeployerAddress": "0x9Ab2F05bf3f82e44C6261031...",
    "contractDeployerAddressPassword": "0x025a68d46d78ffbaf9ba1d3cb5c14322e...",
    "fromAddress": "0x9Ab2F05bf3f82e44C6261031...",
    "fromAddressPassword": "0x025a68d46d78ffbaf9ba1d3cb5c14322e...",
    "transactionConfirmationBlocks": 12,
    "contracts": {
      "Trade": {
        "path": "./TradeContract/Trade.json",
        "estimateGas": true,
        "gas": {
          "buyContribution": 500000
        }
      }
    }
  }
}
```

4.2.2 创建个人钱包

这里我使用的是一个Chrome中的插件——MetaMask



创建好账户后，点击账户详情，获取钱包地址和个人私钥




```

rounds:
- label: buyContribution1
  txNumber: 1000
  rateControl:
    type: fixed-rate
    opts:
      tps: 100
  workload:
    module: TradeContract/MyWorkload.js
- label: buyContribution2
  txNumber: 1000
  rateControl:
    type: fixed-rate
    opts:
      tps: 200
  workload:
    module: TradeContract/MyWorkload.js
- label: buyContribution3
  txNumber: 1000
  rateControl:
    type: fixed-rate
    opts:
      tps: 300
  workload:
    module: TradeContract/MyWorkload.js
- label: buyContribution4
  txNumber: 1000
  rateControl:
    type: fixed-rate
    opts:
      tps: 400
  workload:
    module: TradeContract/MyWorkload.js
- label: buyContribution5
  txNumber: 1000
  rateControl:
    type: fixed-rate
    opts:
      tps: 500
  workload:
    module: TradeContract/MyWorkload.js
- label: buyContribution6
  txNumber: 1000
  rateControl:
    type: fixed-rate
    opts:
      tps: 600
  workload:
    module: TradeContract/MyWorkload.js
- label: buyContribution7

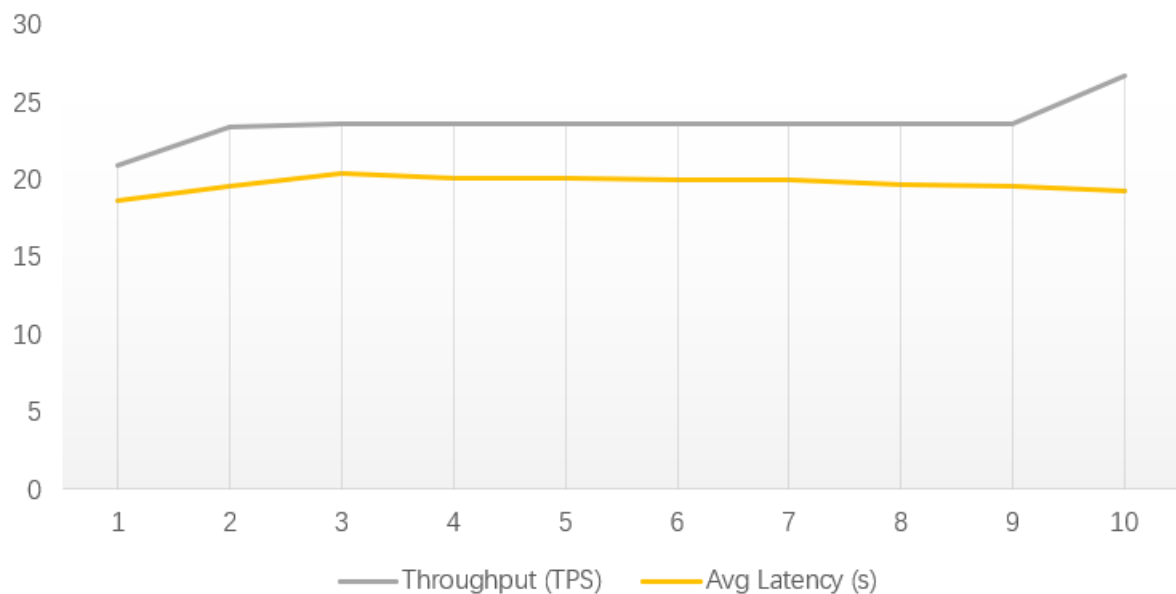
```

首先我对购买贡献度的一个合约中的购买方法 buyContribution 进行了测试，基准测试配置如上。一共十轮测试，每一轮都是一千笔交易，但是tps由100逐渐增大到1000，最终观测测试结果

Caliper report

Summary of performance metrics

Name	Succ	Fail	Send Rate (TPS)	Max Latency (s)	Min Latency (s)	Avg Latency (s)	Throughput (TPS)
buyContribution1	1000	0	100.1	37.91	2.11	18.67	20.9
buyContribution2	1000	0	200.2	38.05	2.23	19.59	23.4
buyContribution3	1000	0	300.6	39.32	2.09	20.44	23.6
buyContribution4	1000	0	376.8	40.01	2.25	20.11	23.6
buyContribution5	1000	0	500.8	40.44	2.39	20.12	23.6
buyContribution6	1000	0	601.3	40.76	2.39	20.04	23.6
buyContribution7	1000	0	701.3	41.02	2.42	19.99	23.6
buyContribution8	1000	0	696.4	40.92	2.33	19.68	23.6
buyContribution9	1000	0	755.9	41.09	2.41	19.54	23.6
buyContribution10	1000	0	721.5	36.20	2.34	19.27	26.7



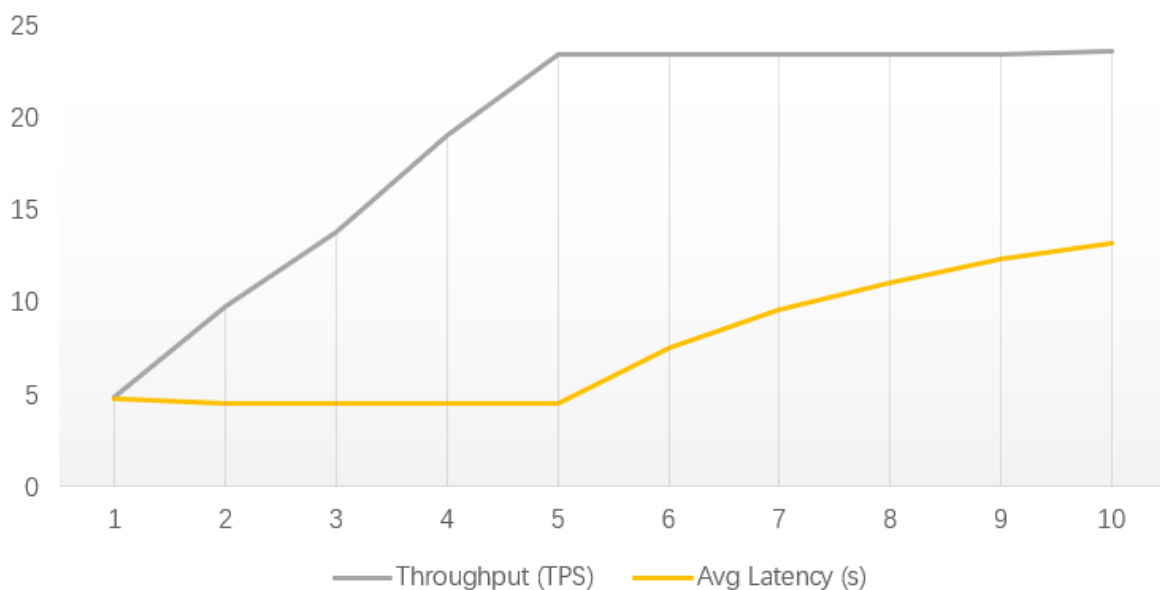
将数据制成图表可以看到，该合约的 `buyContribution` 函数吞吐量并没有随着发送速率的提升而提升，而是小幅提升后维持在了在一定水平；而平均延迟时间也一直很高，大多数情况都在20s左右。可以认为是由于该函数为写函数，对资源消耗较大，一开始就造成了性能饱和，所以后续几轮测试结果变化不大。

于是我降低了每轮发送交易的tps，并重新进行了测试，结果如下：

Caliper report

Summary of performance metrics

Name	Succ	Fail	Send Rate (TPS)	Max Latency (s)	Min Latency (s)	Avg Latency (s)	Throughput (TPS)
buyContribution1	1000	0	5.0	11.94	2.13	4.74	4.9
buyContribution2	1000	0	10.0	7.03	2.01	4.48	9.8
buyContribution3	1000	0	15.0	7.00	2.02	4.52	13.8
buyContribution4	1000	0	20.0	7.01	2.01	4.52	19.1
buyContribution5	1000	0	25.0	7.06	2.05	4.56	23.5
buyContribution6	1000	0	30.0	12.36	2.06	7.52	23.5
buyContribution7	1000	0	35.0	16.26	2.02	9.59	23.5
buyContribution8	1000	0	40.0	19.24	2.05	11.10	23.5
buyContribution9	1000	0	45.0	21.54	2.05	12.34	23.6
buyContribution10	1000	0	50.1	23.34	2.06	13.18	23.6



可以看到第五轮是转折点，也就是发送速率为25TPS时，如果以更高的速率发送交易，吞吐量将不再提升，而平均延迟开始逐渐变高。可以大致推断出，`buyContribution` 函数会在发送率在25TPS占满链路性能。

为了对比结果，我又测试购买贡献度合约中的一个只读函数 `getContributionInfo`，一共十轮测试，每一轮都是一千笔交易，但是tps由100逐渐增大到1000，结果如下：

Caliper report

Summary of performance metrics

Name	Succ	Fail	Send Rate (TPS)	Max Latency (s)	Min Latency (s)	Avg Latency (s)	Throughput (TPS)
getContributionsInfo	1000	0	100.1	0.01	0.00	0.00	100.1
getContributionsInfo2	1000	0	200.2	0.01	0.00	0.00	200.2
getContributionsInfo3	1000	0	300.4	0.01	0.00	0.00	300.3
getContributionsInfo4	1000	0	400.5	0.00	0.00	0.00	400.3
getContributionsInfo5	1000	0	500.8	0.00	0.00	0.00	500.3
getContributionsInfo6	1000	0	601.3	0.00	0.00	0.00	601.0
getContributionsInfo7	1000	0	678.9	0.01	0.00	0.00	678.4
getContributionsInfo8	1000	0	595.2	0.01	0.00	0.00	594.5
getContributionsInfo9	1000	0	627.4	0.01	0.00	0.00	627.0
getContributionsInfo10	1000	0	643.5	0.01	0.00	0.00	642.7

可以看到十轮测试，延迟时间几乎都为0，并且吞吐量和发送速率差距并不大，说明只读函数对资源消耗很小。