PREDA-Toolchain (PREDA Language Preview Toolchain) is a local toolkit developed for PREDA. It provides the ability for programmers to compile, deploy, and conduct performance tests for their smart contracts.

Quick start

Welcome to PREDA-Toolchain. Before learning how to use it, please refer to the Installation Guide document to install PREDA-Toolchain.

After installing the PREDA-Toolchain, it is important to become familiar with it, we have provided some sample contracts and scripts in the examples directory of the installation directory.

Check out the sample contract

```
cd /opt/PREDA/examples
view Token.prd
```

```
contract Token {
    @address bigint balance;

    @address function bool transfer(address to, bigint amount) export
    {
        if(balance >= amount)
        {
            balance -= amount;
            relay@to (^amount)
            {
                                __debug.print("deposit: ", amount);
                balance += amount;
            }

            return true;
        }

        return false;
    }

    @address function mint(bigint value) {
        __debug.assert(value >= 0ib);
        balance += value;
    }

    struct payment
    {
        address     to;
        bigint      amount;
    }

    @address function bool transfer_n(array<payment> recipients) export
        {
        bigint total = 0ib;
        for (uint32 i = 0u; i<recipients.length(); i++)
        {
            __debug.assert(recipients[i].amount >= 0ib);
            total += recipients[i].amount;
        }
        if(total <= balance)
        {
                        __debug.print("transfer_n*", recipients.length(), ", total=", total);
                        balance -= total;
            for (uint32 i = 0u32; i<recipients.length(); i++)
            {
                if(recipients[i].amount>0ib)
                {
                    relay@recipients[i].to (bigint amount = recipients[i].amount){
                                                __debug.print("deposit_n: ", amount);
                        balance += amount;
                    }
                }
            }

            return true;
        }

        return false;
    }
}
~
~
~
"Token.prd" [readonly][dos] 60L, 1518B
```

The Token.prd is a simple token example written in the PREDA language, it provides mint, transfer and transfer_n functions.

Compile the smart contract

In the bin directory there is our executable program, we can compile the contract through it.

```
cd /opt/PREDA/bin
./chsimu ./../examples/Token.prd -stdout
```

```
[PRD]: Chain initialized with 4 shard(s), in sync-sharding mode
Source code for contract `Token` is loaded from ./../examples/Token.prd
Compiling 1 contract(s), target=PREDA_NATIVE
[
   {
      "contract": "Token",
      "engine": "PREDA_NATIVE",
      "hash": "czhs3km360dm6ytdwwd85c4q5d917xg8c71v4q3m61knpchwzh40",
      "finalized": false,
      "ImplmentedInterfaces": [],
      "StateVariables": [{"name": "balance", "scope": "address", "dataType": "bigint"}],
      "Scopes": {"address": "HasState|HasFunction"},
      "ScatteredMaps": {},
      "Structs":
      [
         {
            "scope": "payment",
            "layout":
            [
               {"identifier": "to", "dataType": "address"},
               {"identifier": "amount", "dataType": "bigint"}
            ]
         }
      ],
      "Enumerables": [],
      "Interfaces": {},
      "Functions":
      [
         {
            "name": "transfer",
            "flag": "InvokeByNormalTransaction|EmitRelayInAddressScope",
            "scope": "address",
            "opcode": 0
         },
         {
            "name": "transfer_n",
            "flag": "InvokeByNormalTransaction|EmitRelayInAddressScope",
            "scope": "address",
            "opcode": 1
         },
         {
            "name": "__relaylambda_2_transfer",
            "flag": "InvokeByRelayTransaction",
            "scope": "address",
            "opcode": 2
         },
         {
            "name": "__relaylambda_3_transfer_n",
            "flag": "InvokeByRelayTransaction",
            "scope": "address",
            "opcode": 3
         }
      ]
   }
]
[PRD]: Compile succeeded
```

Check out the test script

```
cd /opt/PREDA/examples
view Token.prdts
```

```
random.reseed
allocate.address 1024
chain.gaslimit 256

chain.deploy @1 Token.prd

log.highlight Token test
log Perparing test transactions

state.set address.Token @all { balance:"10000000000000" }

txn1[] = Token.transfer*$~count$ @random { to:"$@random$", amount:"$random(1000, 2000)*100$" }

chain.info

stopwatch.restart
chain.run
stopwatch.report

chain.info

viz.profiling
~
```

PREDA-toolchain provides a scripting language for testing smart contracts easily, it mainly includes the following functions:

- deploy smart contract
- set on-chain states
- call a smart contract function
- smart contract performance testing
- chain info visualization(need the VsCode)

For more syntax details, please refer to PREDA test script syntax Chapter.

Run the test script

```
cd /opt/PREDA/bin
./chsimu ./../examples/Token.prdts -count:10 -stdout
```

```
Physical CPU core: 4, Logic CPU core: 8
Execution Engine Initialized, <PREDA Native Build> v0.0.1, PREDA DevTeam
Repository: /home/lei/.preda/chsimu_repo/native, Module: ./preda_engine (PREDA_NATIVE)

Execution Engine Initialized, <PREDA WASM Build (CWASM)> v0.0.1, PREDA DevTeam
Repository: /home/lei/.preda/chsimu_repo/wasm, Module: ./preda_engine (PREDA_WASM)

Failed to initialized PREDA engine, db path: /home/lei/.preda/chsimu_repo/evm
Failed to mount execution engine: ./preda_engine (SOLIDITY_EVM) with repository in /home/lei/.preda/chsimu_repo/evm

[PRD]: Chain initialized with 4 shard(s), in sync-sharding mode
1024 addresses added and evenly distributed in shards
Source code for contract `Token` is loaded from /opt/PREDA/examples/Token.prd
Compiling 1 contract(s), target=PREDA_NATIVE
[PRD]: Linking 1 contract(s), target=PREDA_NATIVE
[HIGHLIGHT] Token test
Perparing test transactions
Shd#g:  h:1 txn:0/0/1 addr:0
Shd#0:  h:1 txn:1/0/0 addr:256
Shd#1:  h:1 txn:4/0/0 addr:256
Shd#2:  h:1 txn:2/0/0 addr:256
Shd#3:  h:1 txn:3/0/0 addr:256
Total Txn:10/1
Stopwatch restarted
[PRD]: [qy45yz2s5tf6wqzq2ed6mn58er8t3yh3fw91se7dnq1dm91d141g7gzpp0:ed25519] #2-h1 => #1[d0f8342k6qz6zjk4k8ywb8csc0drfabsvv344abvzrq0c3jz9hn37vxgmw:ed25519]-h1: Token, line 11: deposit: "146500"
[PRD]: [j687491qtkck5s42ger8jy86931s9qeza7r107hbd0rh0wchkrw96zqe8g:ed25519] #3-h1 => #1[6e80jj468ggh8ysnv9w44tv4xn89znzeeq397ajh5v0x1mpn34417jzetc:ed25519]-h1: Token, line 11: deposit: "169600"
[PRD]: [69034c42v0f3wga6m7awfxcx0xxnp6fgy5xybt0bptg2gyg76pym72ddrw:ed25519] #3-h1 => #1[67rp1qxstksaa3jjj7acvcmbrycg9ra1xhvem583pef956f39q587h364m:ed25519]-h1: Token, line 11: deposit: "182400"
[PRD]: [eg2nj1a1gkeg9krvrkng6vz2ct9rnn8fdpfe3p0zev4jpcwf0w1178ds0m:ed25519] #0-h1 => #1[ayjt58kmq660xaesgnqvyvaeyhwc2htfmhhsax3363c5rkjc3tp77rwjv8:ed25519]-h1: Token, line 11: deposit: "139800"
[PRD]: [d7z3qb80d0dy7r65qg7s0cgp1hcp754hpdqq89ssypd9czapdk5a7x1z74:ed25519] #1-h1 => #0[25s6601ze6b8615xczcnmj0d4r56rmbxzy90a1jwrxdw07wx5v8e6wmjvm:ed25519]-h2: Token, line 11: deposit: "167400"
[PRD]: [be8g81qta5x063124w2m80ge2td5c4fxqwpbbjbs9nyn86jrj33a686t78:ed25519] #3-h1 => #0[e70tq6dnfy76kccvswt66c3j9tsnp4ha7akngytf62xs17ggwnv56t91nw:ed25519]-h2: Token, line 11: deposit: "143100"
[PRD]: [sp1vghn27z2mbmtjbrepn99k9bfw7dwe3sew6j9cgf1q0mrzzbrz6p70bc:ed25519] #1-h1 => #1[nhvjvgtzemp093p80tjss818svagzm0xnnefrav992ks6h3g5bwj6gd9kc:ed25519]-h2: Token, line 11: deposit: "105000"
[PRD]: [9bnq2weay6mhffrb5rxhja2man5620q44cd8xecqtxa9wqdgv21v7jzx1r:ed25519] #1-h1 => #1[fjhjyvw40q99e9zysn3b65vrhpz9ymhjwbjsrk3zssfhbnw0sbmg6fwbr8:ed25519]-h2: Token, line 11: deposit: "116600"
[PRD]: [e82j1rd900hyrkvcsqmxwwjjwgfvf8657k9r60bkpntrx2fypat7zbv8w:ed25519] #1-h1 => #2[xq9mmssj34p6dne5tgqqp5xm4v32w54maf47gey8b25ad1d57k0e6bm1cg:ed25519]-h2: Token, line 11: deposit: "148100"
[PRD]: [baryybrp9qyk15jbv2k0se3pjj1e13155kkv4c39gd18ekg6jne36vhb8w:ed25519] #2-h1 => #2[s9s6e97zaqbt2dxs587cw0pdbczpexsnenz79w7bkttmwaretn6f7xk3am:ed25519]-h2: Token, line 11: deposit: "111200"
Stopwatch: 3 msec
Order: 2, TPS:3333, uTPS:6666
Shd#g:  h:3 txn:0/0/1 addr:0
Shd#0:  h:3 txn:0/0/3 addr:256
Shd#1:  h:3 txn:0/0/10 addr:256
Shd#2:  h:3 txn:0/0/4 addr:256
Shd#3:  h:3 txn:0/0/3 addr:256
Total Txn:0/21
```

`-count:10` is the test script input parameter, we have two types of input parameters.

Built-in parameters

-order:n

The default value of order is 2, is means the blockchain will has 2^{order} shards, the max value of order is 16.

-sync/-async

The sharding mode describes the working mode between shards, when the sharding mode is sync, each shard will output blocks synchronously and the block height will be the same; while when the sharding mode is async, each shard will output blocks asynchronously and the block height may be different.

-perftest

By default, PREDA-toolchain will print logs when executing contract calls, which can consume intensive capability during performance testing. Under this circumstance, you can turn on the performance mode by this parameter.

Custom parameters

Users can use custom parameters in test scripts, such as:

```
Token.transfer*$~count$
```

The `$~count$` defines a parameter used to apply for the specified number of transfer transaction, then the user can set the value of this parameter after command

```
./chsimu ./../examples/Token.prdts -count:10 -stdout
```

PREDA test script syntax

Allocate address

Description:

Generate specific number of addresses, The actual number of addresses applied for conforms to the following formula:

$$\begin{cases} actual\_number = shard * n \\ shard * (n-1) <= specific\_number <= shard * n \end{cases}$$

- **shard:** the number of shards
- **n:** positive integer

Command:

```
allocate.address [address_number]
```

Parameter:

- **address_number:** the number of addresses to be generated

Example:

```
allocate.address 10
```

Output:

```
12 addresses added and evenly distributed in shards
```

Specify address

Description:

Use the Allocated address in the test script

Command:

```
@address_order
@all
@random
```

Parameter:

- **address_order:** address order n, random, all, represents the number n+1th address, random address, and all addresses respectively.

Example:

```
// address_0 initiate a vote
Ballot.init @0 { names: ["Spring", "Yarn", "Combat"] }
// all address vote
Ballot.vote @all { proposal_index: $random(0,2)$, case_num: 1 }
// a random address vote
Ballot.vote @random { proposal_index: $random(0,2)$, case_num: 1 }
```

Random

Description:

The PREDA-Toolchain provides some functions related to random numbers, for example, specify random address or specify a random input parameter.

First at all, we should specify a seed for random.

Command:

```
random.reseed [seed]
```

Parameters:

- **seed:** the default seed is timestamp, but you can set as any value manually

Example:

```
// set the random seed
random.reseed 88
// specify a random address when call a contract function
Ballot.init @random { names: ["Spring", "Yarn", "Combat"] }
// specify a random input parameter between 0 and 2
Ballot.vote @0 { proposal_index: $random(0,2)$, case_num: 1 }
```

Set gas limit

Description:

Set the gas limit which is the maximum amount of gas that transactions in a block can consume.

Command:

```
chain.gaslimit [limit]
```

Parameters:

- **limit:** the limit for all transaction's gaslimit in a block

Example:

```
chain.gaslimit 128
```

Deploy smart contracts

Description:

Deploy smart contracts, multiple contracts can be deployed.

Command:

```
chain.deploy @address_order [contract_file] [*contract_file]
```

Parameters:

- **contract_file:** the name of the contract file, which supports multiple names to be set at the same time, with space-separated.
- **address_order:** the order of the address that initiated the contract deployment.

Example:

```
chain.deploy @0 SimpleStorage.prd
```

Output:

```
Compiling 1 contract code(s) ...
contract `SimpleStorage`: 2 function(s) with states in address scope(s)
  0) SimpleStorage.increment: txn
  1) SimpleStorage.decrement: txn
Linking and deploying ...
[PRD]: Successfully deployed 1 contract(s)
```

Set contract states

Description:

Set the state for the blockchain, which is used to initialize the contract state. Users need to set all states in the contract.

Command:

- Set global state

```
state.set contract_name.global { state_name:state_value }
```

- Set shard state

```
state.set contract_name.shard @shard_order { state_name:state_value }
```

- Set address state

```
state.set contract_name.address @address_order { state_name:state_value }
```

Parameters:

- **contract_name:** the name of the contract
- **shard_order:** the serial number for shard
- **address_order:** the serial number for address
- **state_name:** the name of the state to be set
- **state_value:** the value of the state to be set

Example:

Set global state

```
state.set Ballot.global { controller:"$@0$", current_case:0, proposals:[],
last_result:{topVoted:"",case:0}, shardGatherRatio:0}
```

Set shard state

```
state.set Ballot.shard #all { votedWeights:[] }
```

Set address state

```
state.set Ballot.address @all { weight:$random(1, 20)$, voted_case:0 }
```

Update contract state

Description:

Update the state for the blockchain, which is used to initialize the contract state. Users can individually update the specified state in the contract.

Command:

- Update global state

```
state.update contract_name.global { state_name:state_value }
```

- Update shard state

```
state.update contract_name.shard @shard_order { state_name:state_value }
```

- Update address state

```
state.update contract_name.address @address_order { state_name:state_value }
```

Parameters:

- **contract_name:** the name of the contract
- **shard_order:** the serial number for shard
- **address_order:** the serial number for address
- **state_name:** the name of the state to be set
- **state_value:** the value of the state to be set

Example:

Update global state

```
state.update Ballot.global { controller:"$@0$", current_case:0, proposals:[],
last_result:{topVoted:"",case:0}, shardGatherRatio:0}
```

Update shard state

```
state.update Ballot.shard #all { votedWeights:[] }
```

Update address state

```
state.update Ballot.address @all { weight:$random(1, 20)$, voted_case:0 }
```

Call a contract function

Description:

Call a contract function and generate the transaction into mempool .

Command:

```
// call a global function
contract_name.contract_function[*call_number] contract_params
// call a shard function
contract_name.contract_function[*call_number] #shard_order contract_params
// call a address function
contract_name.contract_function[*call_number] @address_order contract_params
```

Parameters:

- **contract_name:** the name of the contract
- **contract_function:** the name of the contract function
- **call_number:** the number of call times, which is an optional parameter
- **shard_order:** the serial number for shard, users can also use `#all` to specify shard
- **address_order:** the serial number for address, users can also use `@random` and `@all` to specify address
- **contract_params:** contract input parameters

Example:

```
// call a global function
KittyBreeding.mint*3 { genes: "$bigint.random(32)$", gender: true, owner:
"$@all$" }
// call a shard function
KittyBreeding.registerNewBorns #all {}
// call a address function
KittyBreeding.breed*$~count$ @random { m: $random(1, ~count-1)$, s:
$random(~count+1, ~count*2-1)$, gender : false }
```

Set the permission to issue FCA (First-Class Asset)

Description:

Set the permission to issue FCA, only contracts with FCA issuance authority can mint token in the contract.

Command:

```
state.token mint token_name by contract_name
```

Parameters:

- **token_name:** the name of the first-class asset
- **contract_name:** the name of the contract which will mint first-class asset

Example:

```
state.token mint BTC by FCA
```

Call a contract function with FCA (First-Class Asset)

Description:

Carry the specified FCA with contract function call.

Command:

```
contract_name.contract_function @address_order {contract_params} <= (token_amount
token_name..)
```

Parameters:

- **contract_name:** the name of the contract
- **contract_function:** the name of the contract function
- **address_order:** the serial number for address, users can also use `@random` and `@all` to specify address
- **token_amount:** the number of tokens carried
- **token_name:** the name of the first-class asset

Example:

```
FCA.transfer @0 {to:"$@1$"} <= (100BTC)
```

Run the blockchain

Description:

Run the blockchain to execute transanctions in the mempool, then add them to block **until each shard is archived**.

Command:

```
chain.run
```

Example:

```
chain.run
```

Get chain info

Description:

Output the number of transactions and addresses of current shard in the blockchain.

Command:

```
chain.info
```

Example:

```
chain.info
```

Output:

```
Global: h:0 txn:0/0/0 addr:0
Shd#0:  h:0 txn:17/0/0 addr:25
Shd#1:  h:0 txn:31/0/0 addr:25
Shd#2:  h:0 txn:23/0/0 addr:25
Shd#3:  h:0 txn:29/0/0 addr:25
Total Txn:100/0
```

log

Description:

Print log

Command:

```
log text
```

Parameters:

- **text:** content of the log

Example:

```
log this is log
```

stopwatch

Description:

Test contract performance with stopwatch.

Command:

```
stopwatch.restart
stopwatch.report
```

Example:

```
stopwatch.restart
chain.run
stopwatch.report
```

Output:

```
Stopwatch: 5 msec
Order: 2, TPS:20000, uTPS:20000
```