合约开发指南 1.0.0

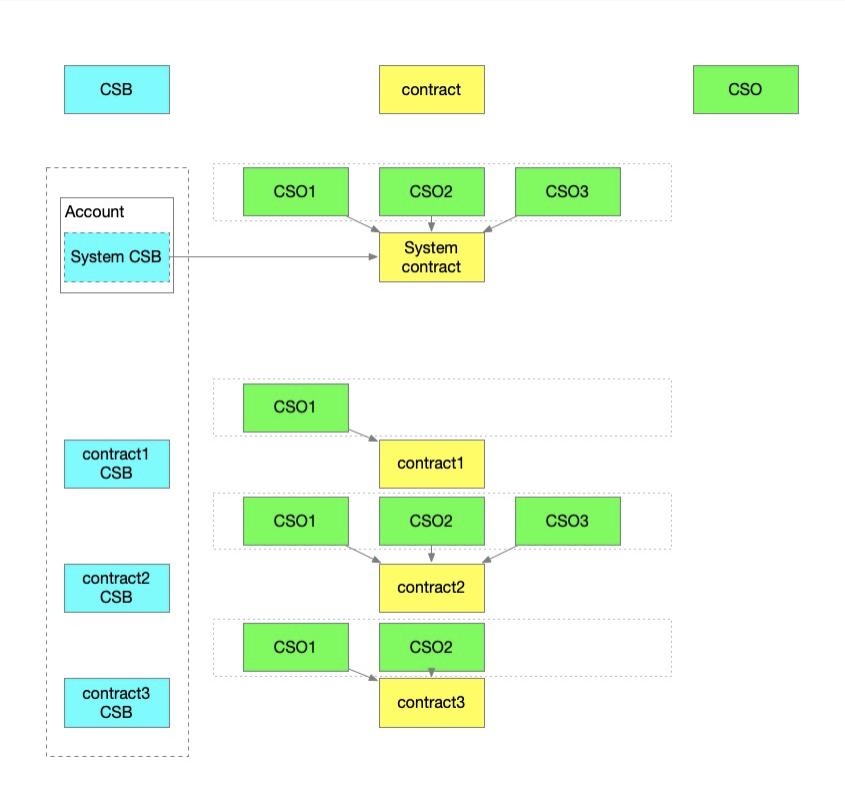
1. 简介

蚂蚁区块链智能合约平台基于 WebAssembly 开发（以下简称 wasm），合约语言是基于`C++17`标准的`C++`语言的一个子集。合约开发者通过编译工具(AldabaCDT)将合约代码编译成wasm字节码中间形式，并将其使用AOT编译技术编译成机器码格式，最终发送到合约平台区块链节点执行。

1.1. 背景知识

传统的区块链存储模型和执行方式(如以太坊等)，合约状态数据只跟合约地址相关，并且无法提前分析读写了哪些状态。aldaba的状态数据分为两部分,即`CSB`和`CSO`，分别跟账户地址和合约地址相关，并且通过定义良好的读写接口，可以提前分析出交易将会读写哪些对象，从而为交易的“并行执行”打下基础。

1.1.1 CSB(Contract State Block)和CSO(Contract Shared Object)



如上图所示，总共有两类状态存储空间，分别称之为`CSB`(合约状态块)和`CSO`(合约共享对象),上图中所有的实线框都是一个独立的读写对象。除了Account(包含系统合约的CSB)是一个单独的读写对象，其他CSB和CSO每个均为一个独立的读写对象。不同交易访问的读写对象之间的关系跟传统的读写锁模型类似（即读读不互斥，读写，写写互斥）。基于这个合约的读写锁模型以及提前分析出来的读写范围，交易的执行得以实现并行。

1.1.2 读写对象的生命周期

`系统合约CSB`: 在账户创建的时候就会产生

`服务合约CSB`: 在账户调用对应服务合约的“ 修改CSB”(调用对应的写CSB接口，即下文4.4中set_private) 接口时候，自动产生

`服务合约CSO`: 合约部署的时候依据对应的初始化代码

2. 合约编译环境

AldabaCDT是蚂蚁区块链平台中将合约代码编译成wasm字节码的工具。下载地址：
ftp://mychainftp.inc.alipay.net/aldaba-cdt/ （仅蚂蚁内部员工可见）

操作系统	安装包
macOS	AldabaCDT-1.4.0-Darwin-x86_64.tar.gz
Linux	AldabaCDT-1.4.0-Linux-x86_64.tar.gz

以macOS 为例，下载安装包至 `$HOME` 目录并解压：

```
$ cd $HOME
$ tar xzvf AldabaCDT-1.4.0-Linux-x86_64.tar.gz
$ export PATH=$HOME/AldabaCDT-1.4.0-Darwin-x86_64/bin:$PATH
```

执行如下命令可以验证是否安装成功：

```
$ my++ -version
xxxxxxxx
```

此外，如果您需要使用IDE（VSCode、CLion等）编写合约并使用IDE的语法提示功能，请将相应的include 目录加入到IDE的头文件目录配置中。比如在 macOS 上，AldabaCDT 的头文件目录为`$HOME/AldabaCDT-1.4.0-Darwin-x86_64/wasm-sysroot/include`。

3. 快速开始

以`hello world`举例说明合约编写、编译过程。

创建一个临时目录存放编写的合约，例如`/tmp/bob`

```
mkdir /tmp/bob
cd /tmp/bob
```

创建一个文件`hello.cpp`，即合约代码

```
touch hello.cpp
```

编辑hello.cpp，引入需要的头文件

```
#include <aldaba/aldaba.h>
```

`aldaba/aldaba.h`中包含编写合约所需的数据结构和`C++`API。

使用如下方式编写您的第一个合约：

```
#include <aldaba/aldaba.h>
using namespace aldaba;

struct MyServiceContract {
  public:
    void hi(const std::string& user ) {
        set_private<0>(user + " hi");
    }

    // 创建1个cso，根据参数admin初始化其内容
    void on_contract_create(const std::string& admin) {
        set_shared<0>(admin);
    }

    // 创建1个csb，根据参数user初始化其内容
    void on_contract_sign(const std::string& user) {
        set_private<0>(user);
    }
};
EXPORT_CONTRACT(MyServiceContract, (hi)(on_contract_create)(on_co
ntract_sign))
```

在完成合约编写后，使用如下命令编译合约：

```
my++ hello.cpp -o hello.wasc
```

您将在`/tmp/bob`目录下看到`hello.wasc`文件，即智能合约字节码文件。

4. 合约开发

4.1. 合约的基本形式

一个智能合约是`C++`中的一个类(class)。编写智能合约所需的数据结构和API均位于aldaba命名空间中，方便起见推荐使用`using namespace aldaba;`。

```
#include <aldaba/aldaba.h>
using namespace aldaba;

class hello: public Contract {
    //...
};
```

4.2. 合约中的数据类型

4.2.1. 非合约接口参数类型

支持`C++`标准中的所有基本类型和标准库类型,没有限制。
对于基本类型的四则运算与逻辑运算(布尔运算、比较符等)，与`C++`标准一致。其不一致行为列举如下：

1. 除零错。如果除法中除数是0，则合约会异常终止。
2. 浮点数。仅支持32位和64位的标量浮点数。参见https://webassembly.org/

另外,提供了一个枚举,作为合约API的参数使用

```
enum class DigestType : std::uint16_t {
  UNKNOWN = 0,
  SHA256 = 1,
  SM3 = 2,
  KECCAK256 = 3,
```

```
};
```

4.2.2. 合约接口参数和返回值类型

共17种类型,包含15种基础数据类型和2种集合类型。

4.2.2.1. 基础类型

C++本身提供的10种:

```
bool
int8_t
int16_t
int32_t
int64_t
uint8_t
uint16_t
uint32_t
uint64_t
std::string
```

另外和平台相关内置的5种,描述和定义如下:

内置数据类型	描述
Bytes32	定长32字节
Bytes64	定长64字节
Account	账户地址（28字节）
Contract	服务合约或资产合约地址（28字节）
Asset	资产代码

```
using Bytes32 = std::array<uint8_t, 32>;
using Bytes64 = std::array<uint8_t, 64>;

class Address {
public:
    static constexpr size_t length = 28;
    Address(std::array<uint8_t, length>&& data) : data_(std::move
```

```
(data)) {}
    const std::array<uint8_t, length>& data() const {
        return data_;
    }
    std::string to_hex() const {
        return Bin2Hex(std::string(data_.begin(), data_.end()));
    }
    bool operator==(const Address& rhs) const {
        return data_ == rhs.data_;
    }
private:
    std::array<uint8_t, length> data_;
};

class Account : public Address {
public:
    using Address::Address;
};

class Contract : public Address {
public:
    using Address::Address;
};

class Asset : public std::string {
  using std::string::string;
public:
  Asset(const std::string &str) : std::string(str) {}
};
```

4.2.2.2. 集合类型

集合类型	描述
template <typename T> std::vector<T>;	元素类型为基础数据类型之一
template <typename T> std::map<std::string, T>;	主键类型为string，值为基础数据类型之一

4.3. 合约接口

4.3.1. 合约接口

对一个已完成部署的合约进行调用，需要指定调用哪个方法，被调用的这个方法称为合约接口。并非所有的方法都可以成为合约接口，将一个方法定义为合约接口需要满足下列条件:

1. 将方法定义为public
2. 其参数和返回值类型只支持 17种类型
3. 在合约外使用EXPORT_CONTRACT导出

4.3.2. EXPORT_CONTRACT 宏

EXPORT_CONTRACT的使用方式:

```
EXPORT_CONTRACT(合约名,(方法1)(方法2)...(方法n))
```

例:

```
/*Hello.cpp*/
#include <aldaba/aldaba.h>
using namespace aldaba;

class Hello {
public:
    int64_t add(int64_t x, int64_t y) {
        return x + y;
    }

    int64_t sub(int64_t x, int64_t y) {
        return x - y;
    }
};

EXPORT_CONTRACT( Hello, (add)(sub))
```

4.4. 合约数据序列化和持久化

get_private

获取当前账户在当前合约中的第 i 号私有状态的内容，即CSB（目前 i 恒为 0）。

```
template <size_t I>
std::vector<uint8> get_private();

template <size_t I, typename T>
T get_private();
```

set_private

设置当前账户在当前合约中的第 i 号私有状态的内容，即CSB（目前 i 恒为 0）。

```
template <size_t I>
void set_private(std::vector<uint8_t>& bytes);

template <size_t I, typename T>
void set_private(T val);
```

get_shared

获取当前合约的第 i 号共享状态的内容（i 为 0-31），即CSO。

```
template <size_t I>
std::vector<uint8_t> get_shared();

template <size_t I, typename T>
T get_shared();
```

set_shared

设置当前合约的第 i 号共享状态的内容（i 为 0-31），即CSO。

```
template <size_t I>
void set_shared(const std::vector<uint8_t>& bytes);

template <size_t I, typename T>
void set_shared(T val);
```

在实际使用中,需要操作的状态并不一定是 std::vector<uint8_t> 类型,而是其它基础类型或者自定义类型,这时我们需要一套序列化和反序列化方法。

AldabaCDT里提供了对以下基础类型的序列化和反序列化支持:

- int8_t
- uint8_t
- int16_t
- uint16_t
- int32_t
- uint32_t
- int64_t
- uint64_t
- bool
- std::string
- `std::vector<T>` //T仅限于上面的类型

对于自定义类型,我们提供了SERIALIZE宏,使用方式如下

```
    struct Greeting {
        int32_t days;
        std::string name;
        SERIALIZE(Greeting, (days)(name));  // 序列化自定义结构体
    };
```

此时就可以使用此类型了

```
    int64_t balance = 1000;
```

```
    set_private<0>(balance);
    int64_t b = get_private<0,uint64_t>();

    Greeting greeting = {3, "jack"};
    set_shared<0>(greeting);
    Greeting g = get_shared<0,Greeting>();
```

4.5. 合约方法

4.5.1. 合约默认方法

本能力暂不开放

4.5.2. 合约特定方法

在特定时机下，该类方法会被调用。对于业务合约而言，目前支持三类特定方法（前两类在写业务合约时必须实现），即

```
// 可变参数由SDK contract_create的contract_create_args参数指定
void on_contract_create(...)

// 在后续可变参数由SDK contract_create的default_signup_args参数指定
void on_contract_sign(uint64_t role, ...)

// 在调用系统合约contract_destroy方法触发
void on_contract_destroy()
```

其中`on_contract_create`是合约开发者自定义的初始化CSO的函数，即初始化CSO的相关操作可以在该函数中实现，用户在**部署合约（SDK contract_create)**时隐式执行该函数。从合约生命周期而言，只会执行一次。
其中`on_contract_create`是合约开发者自定义的初始化CSB的函数，即初始化CSB的相关操作可以在该函数中实现，用户在**调用写CSB的接口时**隐式执行该函数。从账户生命周期而言，每个合约只会执行一次。

4.5.3. 合约普通方法

即除合约默认方法和特定方法外的方法，由合约编写者开发。

4.6. 合约初始化

在部署合约（调用SDK contract_create，但该接口只有链管理员有权限调用）时候，会调用业务合约开发者实现的`on_contract_create`方法，对合约的CSO进行初始化。

```
#include <aldaba/aldaba.h>
using namespace aldaba;

struct MyServiceContract {
  public:

    // 创建1个cso，在第0号CSO存入admin字符串
    void on_contract_create(const std::string& admin) {
        set_shared<0>(admin);
    }
}
```

- 其中on_contract_create的参数admin由调用contract_create时，通过参数contract_create_args传入

4.7. 合约间调用

4.7.1. 合约运行时的上下文环境

合约上下文指的是合约运行环境，例如，谁调用了这个合约（sender），本合约的ID是什么(self)，当前合约能使用的gas是多少，当前合约本次收到多少资产。 这些信息可以通过以下合约API获取到:

get_block_number

获取当前交易所属区块的区块号。

```
uint64_t get_block_number()
```

get_block_timestamp

获取当前交易所属区块的毫秒级时间戳。

```
uint64_t get_block_timestamp()
```

get_tx_hash

获取当前交易的哈希值。

```
1 Bytes32 get_tx_hash();
```

get_tx_sender

获取当前交易的发起账户。

```
1 Account get_tx_sender();
```

get_account

获取当前执行帧的账户。

```
1 Account get_account();
```

get_contract

获取当前执行帧的合约地址。

```
1 Contract get_contract();
```

get_method

获取当前执行帧的合约方法。

```
1 std::string get_method();
```

4.7.2. 普通调用

普通调用的概念为在发生 `co_call` 函数调用时，当前合约的上下文被保存，智能合约平台随后切换到被调用合约的上下文环境并执行相关代码。例如A合约调用B合约的某个方法f，意味着f执行过程中，上下文环境切换到B合约的环境。当普通调用发生时，需要指定被调用合约的account，转移给被调用合约地址，gas数量，被调用合约接口名称，被调用合约接口的参数列表。

1. Account1.Contract1.method(args1..)–> Account2.Contract2.method(args2...) –> Account3.Contract3.method(args3...)
   Contract1执行过程中，get_account()是Account1; 操作的存储对象是跟Contract1相关的CSO和CSB。Contract2的执行过程中，get_account()是Account2, get_tx_sender()是Account1（即交易发起者），
   Contract3的执行过程中，get_account()是Account3, get_tx_sender()是Account1（即交易发起者），

普通调用使用co_call，基本形式为:

```
//
auto ret = co_call<TYPE>(account, contractId, gas, methodName, args...);
```

其中TYPE是被调用合约方法的返回值类型。如果被调用合约内发生abort，那么直接退出，上一层无法感知
注意:

1. co_call 不支持指针类型作为参数或引用类型作为返回值，否则会导致合约编译错误。
2. 返回值不能是特殊的account, contract, asset类型.

例:

```
/*A合约*/
#include <aldaba/aldaba.h>
using namespace aldaba;

struct A {
    void call_contract(Account acct, Contract contract, std::string value) {
        auto res = co_call<void>(acct, contract, 10000000, "set_value", value);
    }
}

EXPORT_CONTRACT(A, (call_contract));
```

```
#include <aldaba/aldaba.h>
using namespace aldaba;

struct B {
    void set_value(std::string value) {
        set_private<0>(value);
```

```
    }
}

EXPORT_CONTRACT(B, (set_value));
```

4.8. 异常处理

合约在执行过程中遇到异常时，会立即停止执行并回滚该合约所造成的一切变更以确保世界状态不会受其影响，即本交易涉及到的存储变更和log输出都会被回滚。

4.8.1. 异常错误码

异常产生的原因有以下几类:

1. 用户通过调用 abort 主动抛出异常，此时异常信息会进入到交易回执的output字段中
2. 合约代码存在逻辑错误，导致mychainlib等第三方库中抛出异常
3. 合约代码存在逻辑错误，导致虚拟机执行字节码过程中抛出异常
4. 用户编译好合约后，不小心修改了字节码文件，导致合约是非法的字节码
5. 用户调用合约时，指定的合约接口名称或合约接口参数不正确
6. 用户调用合约时，调用hostapi遇到错误抛出异常

当合约调用执行出错时，用户可以在交易回执的 `output` 字段找到详细的错误信息。常见的错误见下表:

错误码	错误信息	错误原因
3001	host api异常	查看output中的Errormsg
3002	合约主动abort	查看output中的Errormsg
3003	不期望的异常	联系链管理员定位原因
10204	VM_METHOD_NOT_EXIST	合约方法不存在
10205	VM_PARAMETER_NOT_MATCH	调用合约方法参数不匹配

4.8.2. 合约互相调用过程中的异常处理

如果合约调用另一个合约过程中出现异常。处理的规则如下:

- A->B，B执行过程中出现异常，那么B造成的一切世界状态变化都不会生效，A造成的一切世界状态变化都不会生效。

上面所提到的“造成的一切世界状态变化”指的是存储变更。

4.9. 合约API

也许您可以看到这些API调用了一些更加基础的库函数，请不要擅自使用那些函数，使用不当可能造成您的合约行为不符合预期。请只使用本文档中列出的API。

4.9.1. 区块链交互API

合约代码包含头文件 `aldaba/aldaba.h` 后，可以调用合约工具链提供的合约运行时函数。所有合约运行时函数均声明在命名空间 `aldaba` 下。
下面介绍了各个合约运行时函数的接口和语义（4.7.1.中介绍的函数不再这里赘述）。

4.9.1.1. 控制流操作

abort

终止当前交易，撤销对世界状态的修改，并且调用 `log` 函数输出的信息会被撤回。

```
1 void abort(const std::string& msg);
```

co_call

对账户发起一个已存在合约的合约方法调用。

```
1 template <typename... Ts>
2 void co_call(const Account& a, const Contract c, uint64_t gas,
3              const std::string& fn, Ts... args);
```

4.9.1.2. 工具函数

log

输出一条结构化信息到receipt

```
1 void log(const std::string& topic, const std::string& message);
```

println

向虚拟机日志中输出一行。

```
void println(const char* format, ...);
```

4.9.1.3. 密码学函数

validate_mycrypto_pubkey

验证公钥合法性，目前支持SECP256-R1和RSA2048。

```
bool validate_mycrypto_pubkey(const std::string &pubkey);
```

verify_mycrypto_signature

验证签名合法性，目前支持SECP256-R1和RSA2048。

```
bool verify_mycrypto_signature(const std::string &pubkey, const std::string &msg,
                               const std::string &sig, DigestType dt);
```

base64_decode

base64解码。

```
bool base64_decode(const std::string &pem, std::string &der);
```

base64_encode

base64编码。

```
bool base64_encode(const std::string &msg, std::string &pem);
```

digest

sha256哈希函数。

```
bool digest(const std::string &msg, std::string &hash, DigestType dt);
```

4.9.2 第三方库

- jsoncpp

```
#include <aldaba/aldaba.h>
#include <third_party/jsoncpp/json.h>
```

```
3
4  using namespace aldaba;
5
6  class TestJsoncpp {
7  public:
8    void parse() {
9      std::string strValue = "{                        \
10           \"key\": \"value1\",  \
11           \"array\": [          \
12              {\"arraykey\": 1}, \
13              {\"arraykey\": 2}  \
14            ]                    \
15          }";
16
17     Json::Reader reader;
18     Json::Value root;
19     if (reader.parse(strValue, root)) {
20       if (!root["key"].isNull()) {
21         std::string strValue = root["key"].asString();
22         println("%s\n", strValue.c_str());
23       }
24
25       Json::Value arrayObj = root["array"];
26       for (int i = 0; i < arrayObj.size(); i++) {
27         int iarrayValue = arrayObj[i]["arraykey"].asInt();
28         println("%d\n", iarrayValue);
29       }
30     }
31   }
32 };
33
34 EXPORT_CONTRACT(TestJsoncpp, (parse))
```

- flatbuffer

```
1 namespace TestApp;
2 struct KV {
3  key: ulong;
```

```
 value: double;
}
table TestObj {
 id:ulong;
 name:string;
 flag:ubyte = 0;
 list:[ulong];
 kv:KV;
}
root_type TestObj;
```

```
#include <aldaba/aldaba.h>
#include <third_party/flatbuffers/flatbuffers.h>
#include "test_generated.h"
#include <vector>

using namespace aldaba;
using namespace std;
using namespace TestApp;

void Require(bool condition, std::string msg) {
  if (!condition) {
    abort(msg);
  }
}

class FlatbufferTest {
public:
  void test() {
    flatbuffers::FlatBufferBuilder builder;

    /////////// Serialize /////////
    // Create list
    std::vector<uint64_t> vec;
    for (size_t i = 0; i < 10; i++) {
      vec.push_back(i);
    }
    // Create flat buffer inner type
```

```
28      auto id = 123;
29      auto name = builder.CreateString("name");
30      auto list = builder.CreateVector(vec); // vector
31      auto flag = 1;
32      auto kv = KV(1, 1.0); // struct
33      // table
34      auto mloc = CreateTestObj(builder, id, name, flag, list, &kv)
    ;
35      builder.Finish(mloc);
36
37      char *ptr = (char *)builder.GetBufferPointer();
38      uint64_t size = builder.GetSize();
39      Require(size > 0, "size is wrong");
40    }
41 };
42
43 EXPORT_CONTRACT(FlatbufferTest, (test))
```

5. 合约编译

5.1 AOT wasm 合约

AldabaCDT 支持通过 AOT 编译技术，把 wasm 字节码预编译成机器码格式。使用 AOT 技术，合约在链上的执行效率可以接近原生 C++ 程序的效率，并且保证和 wasm 同样的安全性和确定性 。

```
1 $ my++ -o hello-aot.wasc hello.cpp
```

注意:
AOT 合约体积较大，可能会超出区块链平台 payload 默认 1MB 的大小限制，导致部署失败。可以通过修改区块链的创世块配置参数，或者发送交易调用系统合约来修改 payload 大小限制。具体操作方法请参见 ALDABA 的使用手册。

5.2 构建静态库

用户可以使用 AldabaCdt 提供的工具，将一组源文件编译打包成一个 wasm 的静态库，提供给合约使用。

比如我有两个 C++ 源文件 foo.cc 和 bar.cc，分别定义了一些工具函数，函数接口统一声明在头文件 foobar.h 中：

```
├── bar.cc
├── foo.cc
└── foobar.h
```

用下面的命令编译源代码（`my++` 可以一次编译单个或多个源文件），然后打包成一个静态库 foobar.a：

```
$ my++ -c foo.cc bar.cc
$ llvm-ar rcs foobar.a foo.o bar.o
```

要在合约内使用这个静态库，只需在合约代码中包含头文件 foobar.h，然后编译合约时指定静态库路径即可：

```
$ my++ main.cc /path/to/foobar.a -o contract.wasc
```

6. 合约调试

正在支持中

7. 合约部署和调用

请参照 Java SDK 文档

8. 合约升级

正在支持中

对于已经部署的合约，可以通过发送一个特殊的交易进行合约升级，即用一个新的合约替代一个旧的合约。合约升级时，仅仅会用新的合约代码替代原来的合约代码，并会执行新合约的on_contract_update方法。

9. 合约语言特性说明

9.1. 基础设施支持

与标准 `C++` 相比，从安全与审计角度考虑，不推荐以下的基础设施：

1. 指针。指针的越界行为是 `C++` 中最令人难以捉摸的行为，也因此需要在审计时格外小心。
2. 数组。数组的越界是 `C++` 中的常见错误且很难排查，从安全角度建议使用 `std::vector` 或 `std::array`。
3. 全局变量与静态成员变量。全局对象和静态成员对象的构造与析构是在合约开始和退出的时候执行的，并且执行顺序得不到保证，使用时很容易出错。

`C++` 中常见的基础设施还包括重载、模板与继承，合约语言对这些基础设施支持良好，且允许组合使用。

9.2. C++17 标准库支持

鉴于`C++`在内存管理、进程控制、文件使用等方面强大但不符合区块链行为定义的能力，合约语言对`C++`做了很多限制与改造。

9.2.1. 标准库支持与系统调用封装

智能合约平台对合约语言的标准库支持边界定义如下：

1. malloc/free、new/delete等内存管理类操作。已改写以保证安全性。
2. abort/exit等进程控制类操作。已改写以保证安全性，不应在合约中使用。
3. iostream/cstdio中所包含io操作，合约语言不允许进行类似操作。同时提供了与c++中printf行为相仿的print接口供合约开发者本地调试与输出使用，请参阅合约API中的print函数。
4. 不支持随机数

合约语言不支持任何系统调用，因此用户在使用标准库时也不能直接或间接依赖于底层系统调用。与`C++17`标准库相比，不支持：

分类	细分说明
流操作	fstream/iostream
文件系统	filesystem

位操作	bit
线程与异步	std::thread std::future

9.2.2. 不支持的C++特性

合约语言所基于的WebAssembly技术中，在当前阶段不支持C++异常、线程、浮点数-整数转换、RTTI等特性。因此，合约开发者在开发时无法使用如上的特性来开发合约。

编译工具使用下面编译参数限制了某些C++特性，编写合约时请注意不要使用被禁止的特性：

参数选项	说明
-fno-cfl-aa	禁用CFL别名分析
-fno-elide-constructors	禁用复制构造函数的复制消除行为
-fno-lto	禁用链接时优化
-fno-rtti	禁用rtti
-fno-exceptions	禁用异常
-fno-threadsafe-statics	禁用静态局部变量的线程安全特性

9.3. 未定义行为

C++ 语言标准为了给编译器提供更大的优化空间，把许多不符合规范的代码行为都归类为未定义行为。当用户代码中出现未定义行为时，编译器可能认为程序的正确性已经失去保证，从而实施一些十分激进的优化。未定义行为会导致程序运行时中出现难以理解的逻辑错误，用户在编写合约代码时一定要十分谨慎。

下面列出了 C++ 中一些常见的未定义行为：

基础操作

- 越界访问

```
1 int a[4];
2 int i = a[4];
```

- 访问未初始化变量

```
1 int i;
2 std::cout << i;
```

整数运算

- 有符号整数溢出

```
1 int i = INT_MAX + 1;
```

- 非法位移运算

```
1 int i = 1 << -1;
2 int j = 1 << 32;
```

- 非法数学运算

```
1 int i = 1 / 0;
```

指针操作

- 访问空指针

```
1 int* p = nullptr;
2 *p = 0;
```

- 访问分配空间为 0 的指针

```
1 int* p = new int[0];
2 *p = 0;
```

- 访问已被释放的指针

```
int* p = new int;
delete p;
*p = 0;
```

- 指针运算产生的结果越界

```
int a[4];
int* p = a;
int* q = p + 4;
```

- 非法指针类型转换

```
float f;
int* p = (int*)&f;
*p = 0;
```

- 使用 `memcpy` 拷贝有重叠的数据段

```
int a[4];
memcpy(a, a, sizeof(a));
```

9.4 推荐使用的编译选项

C++ 编译器提供了一些选项可以检查许多不正确的或危险的代码写法，我们建议用户在编译智能合约时把这些选项都打开。

选项	说明
-Wall	对可疑的代码写法提出告警
-Wextra	在 -Wall 的基础上提供一些额外的检查
-Werror	把所有的告警当做编译错误

9.5 运行时资源限制

9.5.1 栈空间

合约执行过程中，栈空间的限制为8K。如果您的合约执行过程中，用的栈空间超过这个值，那么合约最终会被强制异常终止，错误码为3001。所以请慎重使用深度递归和过大的栈上变量。

9.5.2 内存空间

合约运行过程中，最多可用的内存空间为16M(默认为16M，如有更改需求请联系管理员)，如果合约运行过程中所需内存超过这个限制，则合约会被强制异常终止，错误码为3001。

9.5.3 合约间调用深度限制 @大栩

一个合约A调用合约B，合约B又可以调用合约C...。这种合约之间的调用深度最多不能超过16，如果超过了这个限制，那么最后被调用的合约会产生异常，错误码为3001（EXECUTOR_RUNTIME_ERROR），错误信息"stack depth overflow"，请您在写合约时留意。

10. C++ WASM合约代码覆盖率收集

正在支持中

11. 参考文献