合约开发指南 2.20

> 注意，本文档是内部文档，如需要对外输出请联系baas平台。

> 如果您发现任何使用疑问或bug，请反馈至: https://yuque.antfin-inc.com/ivrug1/topics

更新记录

2.19 => 2.20 变更内容：

- 新增时间类型 `time64_t` 和相应接口
- 4.7.3 章节新增合约调用栈信息的介绍

0.10.2.17.1 => 2.19 变更内容：

- 4.5.2.2 MyBuffer 的 map/map_iterable 类型版本升级到 v2，所有相关用例均已修改。
- 新增API `BellmanSnarkVerify`
- 新增章节 4.1.1 介绍合约版本号

0.10.2.17 => 0.10.2.17.1 变更内容：

- 新增API `GetConfidentialDepositData`
- 新增API `GetConfidentialDepositFlag`

0.10.2.15 => 0.10.2.16 变更内容：

- 5.2 章节新增 AOT 合约的编译方式
- 5.3 章节新增 wasm 静态库的构建方式

0.10.2.14==>0.10.2.15的变更内容:

- **6. 合约调试**章节全面更新
- 新增章节 4.8.5：时间格式化

0.10.2.13==>0.10.2.14的变更内容:

- 4.8.1章节 修改区块链交互API `Log`的返回值描述，增加topics size的限制
- 新增第6章节来描述如何通过mock的方式来调试C++ WASM合约
- 新增第10章节来描述如何收集合约代码覆盖率
- 4.7.1. 异常错误码。更新了10622错误码的描述，更加详细了一些
- 4.8. 合约API，`CallContract`描述中增加关于10622 错误码的说明
- 4.8. 合约API，增加警告不要擅自使用更加底层的API

0.10.2.12.5==>0.10.2.13的变更内容:

- 8.4.2章节 新增说明"(默认为16M，如有更改需求请联系管理员)"

0.10.2.12==>0.10.2.12.5的变更内容:

- 修复4.8.1. 区块链交互API `GetBlockNumber`的返回值描述
- 修复4.8.1. 区块链交互API `GetBlockTimeStamp`的返回值描述
- 4.7.1. 章节删除`VM_OUT_OF_MEMORY`。
- 4.7.1. 章节更新对`VM_UNKNOWN_EXPORT`的描述
- 4.7.1. 章节更新`VM_INTERGER_OVERFLOW`的描述
- 4.7.1. 章节新增`VM_UNDEFINED_TABLE_INDEX`
- 4.7.1. 章节新增`VM_UNINITIALIZED_TABLE_ELEMENT`
- 8.2.1 新增"4. 不支持随机数"

0.10.2.11==>0.10.2.12的变更内容:

- `my++` 支持编译多个源文件
- MyBuffer map/map_iterable的get_element()方法增加了可选参数revert_on_failure。
- 订正了MyBuffer部分方法的返回值类型。
- 4.2.1章节 "浮点数。禁止用户使用浮点数。"改为 "浮点数。仅支持32位和64位的标量浮点数。参见 https://webassembly.org/"
- 修复SubPedersenCommit中对PC_left和PC_right的描述

0.10.2.10==>0.10.2.11的变更内容:

- API CallContract返回结构中的code字段，新增了错误码的种类
- API GetBalance、GetCode、GetCodeHash、GetRecoverKey的失败原因描述进行了修复，改为账户不存在或账户被冻结。
- 4.2.1章节 "浮点数。合约语言不保证浮点数运算的精度符合IEEE-754要求，不推荐用户使用浮点数。"改为 "浮点数。禁止用户使用浮点数。"

0.10.2.9.1==>0.10.2.10的变更内容:

- 对VM_UNKNOWN_EXPORT,VM_INTEGER_OVERFLOW,VM_OUT_OF_MEMORY三种错误描述进行了修改

0.10.2.9==>0.10.2.9.1的变更内容:

- 新增API `LiftedElgamalContractHomomorphicAdd`
- 新增API `LiftedElgamalContractHomomorphicSub`
- 新增API `LiftedElgamalScalarMutiply`
- 新增API `LiftedElgamalContractZeroCheckVerify`
- 新增API `LiftedElgamalContractRangeVerify`

0.10.2.8==>0.10.2.9的变更内容:

- 新增API `GetRelatedTransactionListSize`
- 新增API `GetRelatedTransactionList`
- 新增API `GetTransactionSender`
- 新增API `GetTransactionReceiver`
- 新增API `GetTransactionTimestamp`
- 新增API `GetTransactionData`
- 新增API `GetTransactionBlockIndex`
- 新增API `GetTransactionDepositFlag`
- 新增API `CreateContract`
- 去掉了8.1章节中“不推荐使用枚举和引用”的内容
- 第5章节【合约编译】新增了ABI定义的链接

0.10.2.7.1==>0.10.2.8的变更内容:

- 新增API `VerifyMessageSM2`
- 新增API `VerifyMessageECCK1`
- 新增API `VerifyMessageECCR1`
- 新增API `RangeProofVerify`
- 新增API `AddPedersenCommit`
- 新增API `SubPedersenCommit`
- 新增API `CalculatePedersenCommit`
- 新增API `PedersenCommitEqualityVerify`
- 新增第三方库，定点数库bcmath

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(mychain.mycdt)将合约代码编译成wasm字节码，合约平台区块链节点对wasm字节码进行解释执行。

2. 准备合约编译环境

合约编译有2种不同的方法，一是下载mychain.mycdt的安装包在本机编译合约，二是使用 Cloud IDE 在线编译合约。您可以根据自己的需要自由选择任意一种编译方式。

2.1. 安装mychain.mycdt在本机编译合约

`mychain.mycdt` 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/mycdt/2.20/

操作系统	安装包
Linux	MYCDT-2.20-Linux-x86_64.tar.gz
macOS	MYCDT-2.20-Darwin-x86_64.tar.gz

以 Linux 为例，下载安装包至 `$HOME` 目录并解压：

```
$ cd $HOME
$ tar xf MYCDT-2.20-Linux-x86_64.tar.gz
$ export PATH="$HOME/MYCDT-2.20-Linux-x86_64/bin:$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，MYCDT 的头文件目录为`$HOME/MYCDT-2.20-Darwin-x86_64/wasm-sysroot/include`。

2.2. 使用Cloud IDE编译合约

您可以选择不安装 mychain.mycdt，使用 Cloud IDE 在线编译 `C++` 合约。

> 本版本的Cloud IDE正在建设中，完成后会通知大家。

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 <mychainlib/contract.h>
```

`mychainlib/contract.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 <mychainlib/contract.h>
using namespace mychain;

class hello:public Contract {
public:
```

```
    INTERFACE void hi(const std::string& user ) {
        Log("hi"s, {"ok", user});
    }
};
INTERFACE_EXPORT(hello, (hi))
```

在完成合约编写后，使用如下命令编译合约：

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

您将在`/tmp/bob`目录下看到`hello.wasm、hello.abi、hello.wasc`文件。

`hello.wasm` 是 wasm 字节码文件，`hello.abi` 是合约的 ABI 定义文件。前两个文件通过合约开发平台定义的编码规则组合成智能合约文件 `hello.wasc`。

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)，其必须继承于`mychain::Contract`。编写智能合约所需的数据结构和API均位于`mychain`命名空间中，方便起见推荐使用`using namespace mychain;`。
例:

```
#include <mychainlib/contract.h>
using namespace mychain;

CONTRACT_VERSION(0);
class hello: public Contract {
    //...
};
```

4.1.1. 合约版本号

每个合约都有一个版本号，以32位无符号整数表示。在合约代码中使用宏 `CONTRACT_VERSION` 来定义这个版本号。如果合约代码中没有显式地定义版本号，那么默认版本号为0。

合约升级时，新合约的版本号不能低于旧合约的版本号（可以相等），否则更新合约的交易会失败，错误码为 10206。这样是为了防止用户不慎用旧的合约替换链上的新合约。

某些特殊情况下，比如合约回滚时，用户可能希望强行用旧合约替换新合约。此时可以在交易信息中指定扩展字段 `EXTENSION_DOWNGRADE_CONTRACT` 来关闭合约升级时的版本号检查。交易扩展字段的编号是 5，值可以是任意内容，比如空字符串。交易扩展字段的具体设置方法请参考相应的客户端 SDK 文档。

4.2. 合约中的数据类型

4.2.1. 基本类型支持

支持`C++`标准中的所有基本类型，例如：

```
int8_t
uint8_t
int16_t
uint16_t
int32_t
uint32_t
int64_t
uint64_t
bool
```

对于这些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/

4.2.2. 合约平台内置数据类型

合约平台定义了一些内置的数据类型，下面是编写合约过程比较常见的数据类型：

1. `ACCOUNT_STATUS`

```
namespace mychain {
//账户状态
enum ACCOUNT_STATUS : uint32_t {
    NORMAL = 0, //正常状态
    FREEZE,     //冻结状态
    RECOVERING  //恢复状态
};
```

2. `Identity` 是合约或账户的唯一标识，内容为32个字节，其定义如下:

```
class Identity {
    public:
    Identity(const std::string& id) ;//从字符串构造Identity，16进制或非16进制字符串均可

    Identity(const Identity& rl) ;  //拷贝构造函数

    Identity(Identity&& rl);    //移动构造函数

    Identity& operator=(const Identity& rl) ;   //赋值运算符

    Identity& operator=(Identity&& rl); //赋值运算符

    inline bool operator<(const Identity& x) ;  //小于运算

    inline bool operator==(const Identity& x) ; //等于运算

    inline bool operator!=(const Identity& x);  //不等于运算

    inline const std::string& get_data() ;      //获取其内容

    inline std::string to_hex(bool upppercase = false); //其16进制字符串形式
}
```

使用示例:

```
//16进制字符串构造Identity对象
Identity id("2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824");      //不带前缀，合法
Identity id2 = "0x2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824";  //带0x前缀，也合法

//判断Identity对象地址是否相等
if(id == "2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824") {
     //xxxxx
}

if(id == id2 ) {
    //xxx
}

//非法的id会造成合约异常终止
Identity id3 = "abcdefg";

//获取其data
if(id.get_data() == Hex2Bin("2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824")) {
    //xxxxx
}
//获取其16进制格式字符串
if(id.to_hex() == "2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824") {
  //xxx
}
```

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，例如3.快速开始中示例的`hi`方法。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. 使用`INTERFACE`宏修饰
3. `INTERFACE`宏导出的方法，其返回值类型只支持可序列化数据类型
4. 在合约外使用`INTERFACE_EXPORT`导出

4.3.2. `INTERFACE` 宏

`INTERFACE`的使用形式:

```
INTERFACE 返回值类型 函数名(参数列表){
    //函数体
}
```

例:

```
INTERFACE int64_t add(int64_t x, int64_t y) {
  int64_t z = x + y;
  return z;
}
```

4.3.3. `INTERFACE_EXPORT` 宏

INTERFACE_EXPORT的使用方式:

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

例:

```
/*hello.cpp*/
```

```
#include <mychainlib/contract.h>
using namespace mychain;

class hello:public Contract {
public:
    INTERFACE int64_t add(int64_t x, int64_t y) {
        return x + y;
    }

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

INTERFACE_EXPORT( hello, (add)(sub))
```

`INTERFACE_EXPORT`导出的方法，方法的参数只支持下面的类型:

- int8_t
- uint8_t
- int16_t
- uint16_t
- int32_t
- uint32_t
- int64_t
- uint64_t
- bool
- std::string
- Identity
- `std::vector<T>` //T仅限于上面的类型
  注意，`INTERFACE_EXPORT`导出的方法，不允许重载。

4.4. 合约初始化

用户如果想在部署合约时执行特定的初始化操作，可以在合约中专门实现一个方法来实现初始化逻辑，并且在部署合约时显式地调用该合约方法。智能合约平台本身在部署合约时不会执行默认初始化。例:

```
#include<mychainlib/contract.h>
using namespace mychain;
class DemoContract : public Contract {
public:
  INTERFACE void DemoInit(int32_t a, std::string b) {
    //do something...
  }
};
INTERFACE_EXPORT(DemoContract,(DemoInit))
```

在部署该合约时，可以通过SDK指定`DemoInit`方法作为初始化方法并指定参数a和b的值。`DemoInit`方法也是普通的合约接口，可以被用户调用。

- 如果部署时`DemoInit`执行出现了异常，那么交易回执的内容是异常信息。
- 如果部署时`DemoInit`执行成功，那么交易回执的内容是该合约字节码。所以请不要定义DemoInit的返回值，因为你无法获取到返回值。

注意：用户每次发起交易调用合约方法时都会构造一个新的合约实例，进而调用该合约的构造函数。因此不要把部署时的初始化操作实现在合约构造函数中，否则初始化操作会被重复执行。

4.5. 数据序列化和持久化

智能合约平台分别提供了基础数据类型的序列化和用户自定义数据类型的持久化方法。

4.5.1 基础数据序列化

可序列化的数据可以使用`pack()`函数序列化为字节串(即std::string)，并且可以使用`unpack()`函数将对应的字节串反序列化为原来的值。例:

```
  std::string buff = pack("hello"s);
  std::string str = unpack<std::string>(buff); // str的值将会是"hell
o"

  std::string int_buff = pack(1234);
  int n = unpack<int>(int_buff); // n 的值会是1234
```

可序列化数据类型

可序列化类型有2大类

- 平台自身支持的可序列化数据类型:
  - int8_t
  - uint8_t
  - int16_t
  - uint16_t
  - int32_t
  - uint32_t
  - int64_t
  - uint64_t
  - bool
  - std::string
  - Identity
  - `std::vector<T>` // T 仅限于上面的类型

请注意 `pack` 函数不支持 C 风格字符串作为参数，因为 C 风格字符串末尾有一个隐含的终止符 '\0'，在序列化和反序列化时可能会导致歧义。当用户需要表达字符串常量时，可以使用 C++17 中的 string_literals 操作符。

在下面的例子中，错误地把 C 风格字符串输入给 `pack` 函数会导致合约编译不通过：

```
const char* s1 = "hello";
std::string b1 = pack(s1); // wrong

const char s1[] = "hello";
std::string b2 = pack(s2); // wrong

std::string b3 = pack("hello"); // wrong

string s4 = "hello";
std::string b4 = pack(s4); // correct

string s5 = "hello"s;
std::string b5 = pack(s5); // correct

```

```
15 std::string b6 = pack("hello"s); // correct
```

pack函数按照下面的规则进行序列化：

编码表（字节序均为小端）

类型	编码	size	编码示例(编码后数据以16进制展示)	C++类型	java sdk/js sdk 类型
bool	原码	1	`false`=>`00` `true`=>`01`	bool	Bool
uint8	原码	1	`123`=>`7B`	uint8_t	Uint8
uint16	原码	2	`12345`=>`39,30`	uint16_t	Uint16
uint32	原码	4	`1234567890`=>`D2,02,96,49`	uint32_t	Uint32
uint64	原码	8	`1234567890123ull`=>`CB,04,FB,71,1F,01,00,00`	uint64_t	Uint64
int8	补码	1	`-123`=>`85`	int8_t	Int8
int16	补码	2	`-12345`=>`C7,CF`	int16_t	Int16
int32	补码	4	`-1234567890`=>`2E,FD,69,B6`	int32_t	Int32
int64	补码	8	`-1234567890123`=>`35,FB,04,8E,E0,FE,FF,FF`	int64_t	Int64
任何指针		编译时报错		T *	
字符串，每个元素为字符类型	元素字符用utf-8编码	先入以LEB128编码的uint32表达	`"hello世界"`=>`0B,68`	std::string	Utf8String

		元素个数，然后遍历放入元素	`,65,6C,6C,6F,E4,B8,96,E7,95,8C`		
数组，元素必为上述内置整型或string类型		先入以LEB128编码的uint32表达元素个数，然后遍历放入元素	int32数组: `{10, 20, 30}`=>`03,0A,00,00,00,14,00,00,00,1E,00,00,00` string数组 `{"hello"s, "smart"s, "world"s}`=>`03,05,68,65,6C,6C,6F,05,73,6D,61,72,74,05,77,6F,72,6C,64`	std::vector	DynamicArray
字节流，操作接口和字符串一致		先入以LEB128编码的uint32表达元素个数，然后遍历放入元素	std::string: `{10, 20, 30}`=>`03,0A,14,1E`	std::string	std::string

- 用户自定义可序列化的类型
  用户可以使用`SERIALIZE`宏将自定义数据类型(struct/class)序列化。
  使用方法:

```
SERIALIZE(结构体名, (成员变量1)(成员变量2)...(成员变量n))
```

例:

```
struct Foo {
  int32_t x;
```

```
  std::string y;
  bool z;
  int32_t tmp_value; //如果SERIALIZE中不写tmp_value, 则tmp_value不参与序列化
  SERIALIZE(Foo, (y)(x)(z))
};

Foo f;
f.x = -1234567890;
f.y = "hello世界";
f.z = true;
```

基于编码表中对基本类型的编码,Foo类型的f变量的编码相当于按照`SERIALIZE`指定的顺序依次编码各成员,未声明的成员不参与编码，如上例中，即按 y -> x -> z 进行顺次编码，首尾相接，结果为`0B,68,65,6C,6C,6F,E4,B8,96,E7,95,8C,2E,FD,69,B6,01`

4.5.2 使用Schema持久化数据

4.5.2.1. 简介

开发智能合约时，一个常见的需求就是将某些数据持久化存储，或者从持久化存储中读取数据内容，为了方便开发者以比较友好的方式实现数据持久化，以及支持复杂的数据结构（比如map类型嵌套），引入基于Schema的存储系统，开发者在开发智能合约时，首先通过Schema描述存储对象数据结构以及各数据结构间的逻辑关系，然后在智能合约中使用根据Schema生成的API来操作数据对象，智能合约在运行时会自动将数据对象的修改持久化存储。

4.5.2.2. Schema语法

Schema的语法定义跟C语言非常类似，如果有使用protobuf或其它IDL经验的话，会十分容易理解Schema语法，下面首先给出一个Schema的示例：

```
// example IDL file

namespace MyContract.Sample;

//table中如果需要定义 map/map_iterable，需提前声明
```

```
attribute "map";
attribute "map_iterable";

table Transfer {
  count:int = 1;
  accounts:[Account](map:"v2");
  iterable_accounts:[Account](map_iterable:"v2");
}

table Account {
  age:short = 18; //定义字段后，设置默认值为18
  gender:short = 1;
  create: int(deprecated); //deprecated 表示该字段已废弃，相当于删除
  orders:[Order](map:"v2"); //带 'map'属性，该字段为 map
  banalce:Balance; //成员为Balance对象
  friends:[Friend]; //数组
}

table Order {
  id:string;
  sender:string;
  receive:string;
  amount:int = 0;
}

table Balance {
  rmb:int = 0;
  usd:int = 0;
}

table Friend {
  name:string;
  age:int = 0;
}

//通过root_type指定指定root类型table，其它所有table都直接或间接属于该table
root_type Transfer;
```

下面简要分析Schema语法:

4.5.2.2.1 table

Schema中的table对应`C++`中的Class，开发者可在table中定义任意数目不同类型字段，每个字段包含类型、名字、默认值（可选，如果未设置，默认为0/NULL），
同时字段还有附加属性，也就是字段后面括号中的内容，如`accounts:[Account](map:"v2")`中的map，表示该字段为map，v2表示该类型的版本号。
table中的字段类型可为基础类型、数组、map、string及其它table类型。关于Schema支持的详细类型可参考数据类型小节。
为了更加灵活，Schema语法支持在已有的table中添加新字段，废弃已有字段，在此过程中，为使得Schema仍然能够保持前向、后向兼容数据，在修改Schema时必须遵循：

1. 只能在table的尾部追加新字段；
2. 不能在Schema中直接删除字段，如果不再使用某字段，只需在该字段的属性中标明 `deprecated`即可，

满足以上条件后，Schema中对存储数据的访问可实现前向、后向数据兼容。

4.5.2.2.2 数据类型

Schema中table字段类型支持基础类型、数组、map、string等，其中基础数据类型如下：

```
1 1 byte:  bool
2 2 bytes: short (int16), ushort (uint16)
3 4 bytes: int (int32), uint (uint32)
4 8 bytes: long (int64), ulong (uint64)
```

数组: 通过 `[type]`定义，其元素成员`type`必须为table类型，比如：

```
1 friends:[Friend];
```

map/map_iterable: 定义形式与数组类似，仍然通过`[type]`定义，但在属性中标明`map`或者`map_iterable`以表示其类型。map/map_iterable的值为`type`类型，且`type`必须为table类型，key的类型是`string`。以下是两个例子，例子当中的“v2”表示 map/map_iterable 类型的版本号。

```
accounts:[Account](map:"v2");
iterable_accounts:[Account](map_iterable:"v2");
```

> 从0.10.2.18开始，用户推荐使用 map/map_iterable 的 v2 版本。新的版本更为安全和灵活，原有版本在某些重名或者嵌套使用的场景下，会触发编译器错误。原有的版本(见下例)会继续支持，但是后续新功能的增加将以 v2 为主。

```
accounts:[Account](map);
iterable_accounts:[Account](map_iterable);
```

> 关于新旧两个版本的使用，需要注意几点：
> 1. v2 提供的用户接口和旧版完全一致。对于用户来说，只需要在字段声明的时候添加一个"v2"的版本标识。
> 2. 在编写全新的 schema 时，或者添加新的字段时，推荐使用 v2，以获得更好的安全性，灵活性，以及持续的功能升级。
> 3. 旧版本的字段不能升级到 v2。

注意：数组与map的`type`必须为table类型，主要是为了良好的扩展性，在接下来的版本中会考虑支持数组与map的`type`为基本类型。

4.5.2.2.3 命名空间

Schema中可通过namespace来声明命名空间，这样在生成的所有`C++`代码会包裹在该命名空间中。

```
namespace MyContract.Sample;
```

如果在Schema中定义了以上`namespace`，为书写代码方便，可在合约cpp代码使用`using namespace MyContract::Sample;`，或者在使用到的变量及函数前添加命名空间前缀。

4.5.2.2.4 Root类型

Schema中最后通过root_type来声明根类型table，

```
root_type Transfer;
```

根类型的table将会作为合约中访问存储数据的入口，合约必须也只能从根table对象开始访问遍历各数据对象信息，也就是说其它table类型最终都是依附于根table而存在。
另外，开发者需要特别注意，访问存储数据过程中，根类对象生命周期必须要贯穿于整个使用范围，由于table对象间可能存在嵌套关系，在访问子对象成员时，同样要确保父对象仍存活。

4.5.2.3. Schema生成的C++代码

Schema中定义的每一个table都对应于`C++`中的一个类，`C++`中生成的类名与Schema中的table名字基本一致，但带`M`后缀。比如示例中的table Account，生成的C++类名为`AccountM`。同时为了使用方便，对生成的每个`C++`类都会生成带`MPtr`后缀的类型，该类型是相关类的智能指针缩写，即`AccountMPtr = std::shared_ptr<AccountM>`，由该智能指针负责管理对象的生命周期。

4.5.2.3.1 Schema生成C++规则

在根据Schema生成`C++`代码的过程中，table中的不同类型字段都会在`C++`类中生成相应API来操作该字段。出于对性能的考虑，我们会对不同类型的字段，生成不同形式的API接口，基本生成原则是：

- 若字段X为基本类型，如int，提供
  - set_x(_x)： 设置x的值
  - get_x()： 读取x的值
- 若字段X为string，提供
  - set_x(_x)： 设置x的值
  - get_x()： 读取x的值
- 若字段X为table对象，提供
  - get_x() ： 返回该对象指针
- 若字段X为数组类型且元素为table对象，提供
  - get_x()： 返回该数组容器指针
- 若字段X为map/map_iterable类型且val为table对象，提供
  - get_x()： 返回该map/map_iterable指针

对于数组容器，我们提供以下的方法调用：

- size()： 返回数组大小
- get_element(int index)： 获得某下标位置处的对象指针
- append_element()： 在尾部追加一个元素，返回对象指针

- insert_element(int index)：在某一下标索引处插入对象，返回对象指针
- delete_element(int index)：删除某一下标位置处的对象元素
- clear()：清空数组对象

对于map/map_iterable容器，我们提供以下的方法调用：

- add_element(string key)：添加一个键值对，返回值对象指针；若键存在,直接覆盖原来键值对
- get_element(string key, bool revert_on_failure=true)：通过键查找值，若存在，返回值对象指针；
  否则，在revert_on_failure的值为true时将抛出异常（Revert），为false时返回nullptr。
- has_element(string key)：判断某一键是否存在，若存在返回true，否则返回false
- delete_element(string key)：删除某一键值

对于数组字段，限制数组长度最大为1024，超过该长度后继续追加元素会报错；
对于string字段，限制最大长度为2K，即2048个字节,string字段超过该阈值后会导致赋值失败。
上面主要介绍了不同字段类型生成的API形式，未标明返回值与参数，如需了解完整的API信息，可参考Schema生成的C++API小节。

再次提醒：对于table X对象来说，所有通过get_x()系列接口返回的均为对象智能指针，即`std::shared_ptr<XM>`，这也就意味着：
当该智能指针维护的变量生命周期结束时，智能指针负责销毁并析构对象，如果对象被修改过，对象析构时将自动序列化并持久存储。

另外，对于Schema中定义的map/map_iterable类型来说，如有需要值为基本类型，考虑到性能与扩展性，建议将该基本类型封装在table中，因为table支持任意添加字段，具备良好的兼容性。以`table A`中需要定义字段`usd [int](map:"v2")`举例，`usd`为map类型，代表某一账户的金钱余额，值为`int`。经过封装改造，我们可在Schema中定义如下table:

```
table Balance {
  usd:int = 0;
}
```

然后在`table A`中定义如下字段:

```
balance [Balance](map:"v2");
```

在合约代码中可通过 `BalanceM` 提供的API来读取/修改`usd`字段值，随着合约不断迭代，可往 `table Balance` 中添加更多字段来代表不同余额，比如人民币、欧元等等。也就是说，**在Schema中一切皆为table。**

4.5.2.3.2 Schema生成的C++API

前一小节中简要分析了Schema中table定义的各类型字段生成API接口以及相关容器API接口的规则，下面表格列出了Schema中table各类型字段生成的完整API信息。当字段类型为table/vector/map时，只提供get_x()接口，返回table/vector/map类型的指针，之后可通过指针进一步操作其字段或者元素。

字段类型（T）	提供的接口	作用	参数	返回值
基础类型 bool short ushort int uint long ulong	T get_x()	获得成员x的值	无	返回成员x的值
	bool set_x(T _x)	设置成员x的值	参数_x，代表需要设置的值	返回类型为bool,true代表成功;false代表失败
string	string get_x()	获得成员x的值	无	返回成员x的值
	bool set_x(string _x)	设置成员x的值	参数_x，代表需要设置的值	返回类型为bool,true代表成功;false代表失败
Table T	TMPtr get_x()	获得指向该table的指针	无	返回指向该table的指针: TMPtr
[T]	TMVectorPtr get_x()	获得数组容器的指针	无	返回数组容器的指针: TMVectorPtr
[T](map:"v2")	TMMapPtr get_x()	获取该map容器指针	无	返回类型为map容器的指针:TMMapPtr
[T](map_iterable:"v2")	TMMapIterablePtr get_x()	获取该map_iterable容器指针	无	返回类型为map_iterable容器的指针:TMMapIterablePtr

下面的表格列出了数组容器的完整API信息。其中get_element()/append_element()/insert_element()接口返回成员类型指针TMPtr，可通过返回的指针调用字段接口为各字段赋值或读取字段内容。数组最大长度为1024，当数组元素超过1024时，调用append_element()/insert_element()接口会抛出异常（Revert）,代表失败。

提供的接口	作用	参数	返回值
TMPtr get_element(int i)	获得数组下标i处的元素	参数i,代表数组下标	返回数组成员类型的指针: TMPtr
TMPtr append_element()	数组尾部追加一个元素	无	返回数组成员类型的指针:TMPtr
TMPtr insert_element(int i)	数组下标i处插入一个元素	参数i，代表数组下标	返回数组成员类型的指针:TMPtr
int delete_element(int i)	删除数组下标i处的元素	参数i,代表数组下标	返回类型为int,0代表成功;其他代表失败
uint32_t size()	获得数组长度	无	返回数组长度
int clear()	清空数组	无	返回类型为int,0代表成功;其他代表失败

下面的表格列出了map容器的完整API信息。
其中add_element()/get_element()接口返回值类型的指针TMPtr，接下来通过指针调用各字段接口为字段赋值或读取字段内容。

提供的接口	作用	参数	返回值
TMPtr add_element(string key)	map中添加一对key/val	参数key，代表插入map中的key	返回类型为map中val类型的指针:TMPtr
TMPtr get_element(string key, bool revert_on_failure=true)	根据键查找值	参数为key,代表map中的键	返回类型为map中值类型的指针:TMPtr
bool has_element(string key)	确认键是否存在	参数key,代表map中的键	返回类型为bool,true代表存在;false代表不存在
int delete_element(strin	删除该键	参数key,代表map中的键	返回类型为int,0代表成功;其他代表失败

g key)			

下面的表格列出了map_iterable容器的完整API信息。

提供的接口	作用	参数	返回值
TMPtr add_element(string key)	map_iterable中添加一对key/val	参数key，代表插入map_iterable中的key	返回类型为map_iterable中val类型的指针:TMPtr
TMPtr get_element(string key, bool revert_on_failure=true)	根据键查找值	参数为key,代表map_iterable中的键	返回类型为map_iterable中值类型的指针:TMPtr
bool has_element(string key)	确认键是否存在	参数key,代表map_iterable中的键	返回类型为bool,true代表存在;false代表不存在
int delete_element(string key)	删除该键	参数key,代表map_iterable中的键	返回类型为int,0代表成功;其他代表失败
uint32_t size()	获得map_iterable长度	无	返回map_iterable长度
TMMapKeyIterator get_map_key_iterator()	获取map_iterable关键字迭代器	无	返回类型是map_iterable的关键字迭代器

下面的表格列出了MapKeyIterator的完整API信息。

提供的接口	作用	参数	返回值
bool valid()	判断当前的迭代器是否有效	无	返回类型为bool，true代表有效，false代表无效
std::string operator*()	获取当前迭代器指向的关键字	无	返回类型为字符串
TMMapKeyIterator& operator++()	移动迭代器指向下一个关键字	无	返回类型为迭代器本身的引用

经过学习上面表格中的`API`说明之后，当我们在Schema中定义table字段的时候，基本已经确定在合约中如何访问该字段信息。

4.5.2.3.3 Schema持久化存储

在智能合约里面使用 schema 生成的 API 之前，必须对持久化存储环境进行初始化。用户可以自定义一个合约方法 `Init` 来调用 schema 自动生成的初始化函数 `InitRoot`，然后在合约部署时显式调用 `Init` 进行初始化。**`InitRoot`不能多次调用，否则会抛出异常（Revert）**。调用方式如下：

```
INTERFACE void Init() {
    InitRoot();
}
```

另外，每次使用schema编译生成的接口访问持久化存储的时候，都必须首先获得Schema中定义的根类型table，参见下面的getTransferM()调用。可以考虑将这个处理放在合约类的构造函数里，这样每次调用合约的时候，都会自动获得存储的根节点，不需要在每个合约函数里反复写这样的处理。

4.5.2.4. 本地合约开发

前面主要介绍了引入Schema机制的目的，Schema语法，以及根据Schema生成`C++`API的规则。
本小节将会介绍如何在实际合约开发中使用Schema。
引入基于Schema的存储系统后，合约开发与平常的开发并无二样，仍然遵守合约开发指南即可。
目前来说，合约开发有两种情况：一是用户下载mycdt工具安装包，在本地开发智能合约，然后手动编译；二是通过在线IDE环境开发合约。
本小节先介绍本地开发的方法，在线IDE将会在后面介绍。

用户下载mtcdt的安装包后，即可在本地开发合约cpp。本地开发合约时，我们提供了以下两种方法可供选择：

1. 用户命令行手动编译，合约cpp文件和schema定义在不同的文件中
2. 使用`my++.sh`脚本自动化编译，合约cpp文件和schema文件可以定义在一个文件内。

下面分别介绍这两种方法。

4.5.2.4.1手动编译合约

如果为了灵活性，用户可选择手动编译合约。手动编译时需要遵循以下步骤，

1. 首先为合约存储定义Schema，并将Schema内容保存为以`.fbs`结尾的Schema文件。例如，将以下Schema内容保存为`transfer.fbs`。

```
//Schema Demo
namespace MyContract.Sample;

attribute "map";
attribute "map_iterable";

table Transfer {
  count:int = 1;
  accounts:[Account](map:"v2");
  iterable_accounts:[Account](map_iterable:"v2");
}

table Account {
  age:short = 18;
  gender:short = 1;
  create:int(deprecated);
  orders:[Order](map:"v2");
  balance:Balance;
  friends:[Friend];
}

table Order {
  id:string;
  sender:string;
  receive:string;
  amount:int = 0;
}

table Balance {
  rmb:int = 0;
  usd:int = 0;
}

table Friend {
```

```
  name:string;
  age:int = 0;
}

root_type Transfer;
```

2. 使用mycdt中的`myflatc.sh`工具将Schema生成C++API

```
myflatc.sh ./ transfer.fbs
```

第一个参数./代表生成的c++文件存储目录，
第二个参数**transfer.fbs**代表schema文件，

3. 在合约cpp文件中通过`#include "transfer_ant_generated.h"`将头文件引入即可，并将合约cpp保存为`transfer_contract.cpp`

下面是合约cpp的文件示例：

```
//简单转账合约示例，仅供演示
//合约源文件
#include <mychainlib/contract.h>

//只需包含上步中生成的xxx_ant_generated.h即可
#include "transfer_ant_generated.h"

using namespace mychain;
using namespace MyContract::Sample;//如果schema中没有定义namespace可省略

class demo:public Contract {
public:
    //使用成员变量保存指向存储root的指针
    TransferMPtr m_ptransfer;

    demo() {
        //在构造函数里获得Schema中定义的根类型table:Transfer, 之后可以直接使用了
```

```
        m_ptransfer = GetTransferM();
    }
    INTERFACE void Init() {
        InitRoot();
    }
    INTERFACE void writedemo() {
        //若空指针，代表异常出错，合约异常退出
        if(!m_ptransfer) {
            Revert("error");
        }
        //设置transfer的count值
        m_ptransfer->set_count(88);

        //map中新增一个账户，key为"btc"，返回对象指针
        AccountMMapPtr paccount_map = m_ptransfer->get_accounts(
);
        AccountMPtr paccount = paccount_map->add_element("btc");
        //如果对象指针为NULL，返回值异常，合约退出
        if(!paccount) {
            Revert("error");
        }
        paccount->set_age(19);
        paccount->set_gender(1);
        //获得Balance对象
        BalanceMPtr pbalance = paccount->get_balance();
        //如果对象指针为NULL，返回值异常，合约退出
        if(!pbalance) {
            Revert("error");
        }
        pbalance->set_rmb(199);
        pbalance->set_usd(1000);

        //账户中orders字段为map，新增加一个kv，key为“order1”，返回对象指
针，当该智能指针维护的
        //order对象生命周期结束后，该kv内容自动序列化并持久存储
        OrderMMapPtr porder_map = paccount->get_orders();
        OrderMPtr porder = porder_map->add_element("order1");
        porder->set_sender("user1");
        porder->set_receive("user2");
        porder->set_amount(8899);
```

```
56
57          //friends为数组，数组尾部追加一个对象元素，返回对象指针
58          FriendMVectorPtr pfriend_vec = paccount->get_friends();
59          FriendMPtr pfriend = pfriend_vec->append_element();
60          pfriend->set_name("myant");
61          pfriend->set_age(88);
62
63          //调用智能指针的reset()后，paccount管理的Account对象`btc`生命周
    期结束，若该对象内容修改
64          //过，则对象析构时自动序列化存储；
65          paccount.reset();
66
67          //账户“btc”中添加一个order，key为 `order2`，返回对象指针，porde
    r中维护的旧对象(order1)
68          //生命周期结束，析构时自动序列化存储
69          porder = porder_map->add_element("order2");
70          porder->set_sender("user3");
71          porder->set_receive("user4");
72          porder->set_amount(88888);
73
74          //合约中新增加一个账户"eth"，返回对象指针，paccount维护的旧对象
    ("btc")生命周期结束,析构时
75          //自动序列化存储
76          paccount = paccount_map->add_element("eth");
77          paccount->set_age(22);
78          paccount->set_gender(0);
79          pbalance = paccount->get_balance();
80          pbalance->set_rmb(299);
81          pbalance->set_usd(4000);
82
83          //账户"eth"中增加一个order，key为"order1"，返回对象指针，porder
    维护的旧对像生命周期结
84          //束，析构时自动序列化存储
85          porder_map = paccount->get_orders();
86          porder = porder_map->add_element("order1");
87          porder->set_sender("user1");
88          porder->set_receive("user2");
89          porder->set_amount(88999);
90
91          //账户"eth"中增加一个order，key为"order2"，返回对象指针，porder
```

```
维护的旧对像生命周期结
92          //束，析构时自动序列化存储
93          porder = porder_map->add_element("order2");
94          porder->set_sender("user5");
95          porder->set_receive("user6");
96          porder->set_amount(8899988);
97
98          //friends数组追加一个元素，返回对象指针，pfriend维护的旧对象生命周
期结束，析构时自动序列化
99          //存储
100         pfriend = pfriend_vec->append_element();
101         pfriend->set_name("lingling");
102         pfriend->set_age(87);
103
104         //iterable_accounts是map_iterable类型，增加两个元素“btc”和“e
th”
105         AccountMMapIterablePtr paccount_mi = m_ptransfer->get_it
erable_accounts();
106         AccountMPtr paccount1 = paccount_mi->add_element("btc");
107         AccountMPtr paccount2 = paccount_mi->add_element("eth");
108     }
109
110     INTERFACE void readdemo() {
111         //若空指针，代表异常出错，合约异常退出
112         if(!m_ptransfer) {
113             Revert("error");
114         }
115         //获取Root对象中的count字段值
116         int count = m_ptransfer->get_count();
117         //通过"btc"获取账户内容，若存在，返回对象指针；否则将抛出异常（Reve
rt）
118         AccountMMapPtr paccount_map = m_ptransfer->get_accounts(
);
119         AccountMPtr paccount = paccount_map->get_element("btc");
120         if(!paccount) {
121             Revert("error");
122         }
123         //访问账户内的各字段值
124         int age = paccount->get_age();
125         int gender = paccount->get_gender();
```

```
126
127         //test statement: will output to log file
128         println("age=%d", age);
129
130         //通过"order1"获得账单信息，若存在，返回对象指针；否则将抛出异常（Revert）
131         OrderMMapPtr porder_map = paccount->get_orders();
132         OrderMPtr porder = porder_map->get_element("order1");
133         if(!porder) {
134             Revert("error");
135         }
136         //获得order对象的各字段值
137         std::string sender = porder->get_sender();
138         std::string receive = porder->get_receive();
139         //通过下标0获得friends数组中的第一个元素，返回对象指针，若下标越界，将抛出异常（Revert）
140         FriendMVectorPtr pfriend_vec = paccount->get_friends();
141         FriendMPtr pfriend = pfriend_vec->get_element(0);
142         if(!pfriend) {
143             Revert("error");
144         }
145
146         auto name = pfriend->get_name();
147         age= pfriend->get_age();
148         //获得账户中balance对象，若存在，返回对象指针；若不存在，返回的指针指向空内容
149         BalanceMPtr pbalance = paccount->get_balance();
150         if(!pbalance) {
151             Revert("error");
152         }
153         int rmb= pbalance->get_rmb();
154         int usd= pbalance->get_usd();
155
156         //获得iterable_accounts的对象，并利用它的迭代器遍历所有元素：“btc”和“eth”
157         AccountMMapIterablePtr paccount_mi = m_ptransfer->get_iterable_accounts();
158         AccountMMapKeyIterator iaccount = paccount_mi->get_map_key_iterator();
159         for (; iaccount.valid(); ++iaccount) {
```

```
            std::string key = *iaccount;
            //在这里使用key
        }

        /*
         * 其它操作
         */
    }
};
INTERFACE_EXPORT(demo, (Init) (writedemo) (readdemo))
```

将上述代码保存为`transfer_contract.cpp`文件。

4. 调用`my++`工具将合约cpp编译成wasm字节码

```
my++ transfer_contract.cpp -o transfer.wasm
```

至此，本地开发合约并手动编译的过程全部结束。

4.5.2.4.2 使用脚本编译合约

本地开发合约时，为了用户使用方便，我们提供了`my++.sh`脚本，将整个编译流程封装起来,只需要该脚本就能编译合约代码。
为使用该脚本，用户需要将合约Schema内容与合约cpp文件放在一起。我们提供了`STORAGE_SCHEMA_BEGIN`与`STORAGE_SCHEMA_END`两个宏标记，用这两个宏分别作为开头与结尾将Schema内容包裹起来并嵌在合约cpp文件开始处即可。

注意，`STORAGE_SCHEMA_BEGIN`与`STORAGE_SCHEMA_END`两个宏包裹的Schema内容必须位于合约cpp的文件开始！

下面列举一个将Schema内容完整嵌入合约cpp的示例:

```
//简单转账合约示例，仅供演示
//合约源文件

#include <mychainlib/contract.h>
```

```
using namespace mychain;

STORAGE_SCHEMA_BEGIN
namespace MyContract.Sample;

attribute "map";
attribute "map_iterable";

table Transfer {
  count:int = 1;
  accounts:[Account](map:"v2");
  iterable_accounts:[Account](map_iterable:"v2");
}
table Account {
  age:short = 18;
  gender:short = 1;

  create:int(deprecated);

  orders:[Order](map:"v2");
  balance:Balance;
  friends:[Friend];
}
table Order {
  id:string;
  sender:string;
  receive:string;
  amount:int = 0;
}
table Balance {
  rmb:int = 0;
  usd:int = 0;
}
table Friend {
  name:string;
  age:int = 0;
}
root_type Transfer;

STORAGE_SCHEMA_END
```

```
45
46 using namespace MyContract::Sample;//如果schema中没有定义namespace
   可省略
47 class demo:public Contract {
48 public:
49     TransferMPtr ptransfer;
50
51     demo () {
52         //首先获得Schema中定义的根类型table : Transfer
53         ptransfer = GetTransferM();
54     }
55     INTERFACE void Init() {
56         InitRoot(); //初始化schema持久化存储环境
57     }
58     INTERFACE void writedemo() {
59         //若返回空指针，代表异常出错，合约异常退出
60         if(!ptransfer) {
61               Revert("error");
62         }
63         //设置transfer的count值
64         ptransfer->set_count(88);
65
66         //map中新增一个账户，key为"btc"，返回对象指针
67         AccountMMapPtr paccount_map = m_ptransfer->get_accounts(
   );
68         AccountMPtr paccount = paccount_map->add_element("btc");
69         //如果对象指针为NULL，返回值异常，合约退出
70         if(!paccount) {
71               Revert("error");
72         }
73         paccount->set_age(18);
74         paccount->set_gender(1);
75         //获得Balance对象
76         BalanceMPtr pbalance = paccount->get_balance();
77         //如果对象指针为NULL，返回值异常，合约退出
78         if(!pbalance) {
79               Revert("error");
80         }
81         pbalance->set_rmb(199);
82         pbalance->set_usd(1000);
```

```
83
84          //账户中orders字段为map，新增加一个kv，key为“order1”，返回对象指针，当该智能指针维护的
85          //order对象生命周期结束后，该kv内容自动序列化并持久存储
86          OrderMMapPtr porder_map = paccount->get_orders();
87          OrderMPtr porder = porder_map->add_element("order1");
88          porder->set_sender("user1");
89          porder->set_receive("user2");
90          porder->set_amount(8899);
91
92          //friends为数组，数组尾部追加一个对象元素，返回对象指针
93          FriendMVectorPtr pfriend_vec = paccount->get_friends();
94          FriendMPtr pfriend = pfriend_vec->append_element();
95          pfriend->set_name("myant");
96          pfriend->set_age(88);
97
98          //调用智能指针的reset()后，paccount管理的Account对象`btc`生命周期结束，若该对象内容修改
99          //过，则对象析构时自动序列化存储；
100         paccount.reset();
101
102         //账户“btc”中添加一个order，key为 `order2`，返回对象指针，porder中维护的旧对象(order1)
103         //生命周期结束，析构时自动序列化存储
104         porder = porder_map->add_element("order2");
105         porder->set_sender("user3");
106         porder->set_receive("user4");
107         porder->set_amount(88888);
108
109         //合约中新增加一个账户"eth"，返回对象指针，paccount维护的旧对象("btc")生命周期结束,析构时
110         //自动序列化存储
111         paccount = paccount_map->add_element("eth");
112         paccount->set_age(22);
113         paccount->set_gender(0);
114         pbalance = paccount->get_balance();
115         pbalance->set_rmb(299);
116         pbalance->set_usd(4000);
117
118         //账户"eth"中增加一个order，key为"order1"，返回对象指针，porder
```

```
维护的旧对像生命周期结
119         //束，析构时自动序列化存储
120         porder_map = paccount->get_orders();
121         porder = porder_map->add_element("order1");
122         porder->set_sender("user1");
123         porder->set_receive("user2");
124         porder->set_amount(88999);
125 
126         //账户"eth"中增加一个order，key为"order2"，返回对象指针，porder
维护的旧对像生命周期结
127         //束，析构时自动序列化存储
128         porder = porder_map->add_element("order2");
129         porder->set_sender("user5");
130         porder->set_receive("user6");
131         porder->set_amount(8899988);
132 
133         //friends数组追加一个元素，返回对象指针，pfriend维护的旧对象生命周
期结束，析构时自动序列化
134         //存储
135         pfriend = pfriend_vec->append_element();
136         pfriend->set_name("lingling");
137         pfriend->set_age(87);
138 
139         //iterable_accounts是map_iterable类型，增加两个元素“btc”和“e
th”
140         AccountMMapIterablePtr paccount_mi = m_ptransfer->get_in
terable_accounts();
141         AccountMPtr paccount1 = paccount_mi->add_element("btc");
142         AccountMPtr paccount2 = paccount_mi->add_element("eth");
143 
144         //整体函数结束，最后顶层的Root类型ptransfer生命周期结束，析构时自
动序列化存储
145         //要保证Root对象的生命周期在整个函数中有效
146     }
147 
148     INTERFACE void readdemo() {
149         //获取Root对象中的count字段值
150         int count = ptransfer->get_count();
151         //通过"btc"获取账户内容，若存在，返回对象指针；否则将抛出异常（Reve
rt）
```

```
152         AccountMMapPtr paccount_map = m_ptransfer->get_accounts(
    );
153         AccountMPtr paccount = paccount_map->get_element("btc");
154         if(!paccount) {
155             Revert("error");
156         }
157         //访问账户内的各字段值
158         int age = paccount->get_age();
159         int gender = paccount->get_gender();
160 
161         //通过"order1"获得账单信息，若存在，返回对象指针；否则将抛出异常（R
    evert）
162         OrderMMapPtr porder_map = paccount->get_orders();
163         OrderMPtr porder = porder_map->get_element("order1");
164         if(!porder) {
165              Revert("error");
166         }
167         //获得order对象的各字段值
168         std::string sender = porder->get_sender();
169         std::string receive = porder->get_receive();
170         //通过下标0获得friends数组中的第一个元素，返回对象指针，若下标越
    界，将抛出异常（Revert）
171         FriendMVectorPtr pfriend_vec = paccount->get_friends();
172         FriendMPtr pfriend = pfriend_vec->get_element(0);
173         if(!pfriend) {
174              Revert("error");
175         }
176 
177         auto name = pfriend->get_name();
178         age= pfriend->get_age();
179         //获得账户中balance对象，若存在，返回对象指针；若不存在，返回的指针
    指向空内容
180         BalanceMPtr pbalance = paccount->get_balance();
181         if(!pbalance) {
182              Revert("error");
183         }
184         int rmb= pbalance->get_rmb();
185         int usd= pbalance->get_usd();
186 
187         //获得iterable_accounts的对象，并利用它的迭代器遍历所有元素：“bt
```

```
    c”和“eth”
188         AccountMMapIterablePtr paccount_mi = m_ptransfer->get_in
    terable_accounts();
189         AccountMMapKeyIterator iaccount = paccount_mi->get_map_k
    ey_iterator();
190         for (; iaccount.valid(); ++iaccount) {
191             std::string key = *iaccount;
192             //在这里使用key
193         }
194
195         /*
196          * 其它操作
197         */
198
199         //函数结束后，Root类型对象ptransfer生命周期结束，ptransfer的生命
    周期贯穿整个函数
200     }
201 };
202 INTERFACE_EXPORT(demo, (Init) (writedemo) (readdemo))
```

假如将以上合约代码保存为`transfer_contract.cpp`文件，接下来在命令行中调用如下命令可将合约编译成字节码：

```
1 my++.sh transfer_contract.cpp -o transfer.wasm
```

4.5.2.5. 在线IDE合约开发

使用在线IDE开发合约时，由于目前在线IDE只能在一个文件中编辑，所以需要开发者将合约Schema内容与合约代码放在一起。
开发者仍可以用`STORAGE_SCHEMA_BEGIN`与`STORAGE_SCHEMA_END`这两个宏分别作为开头与结尾将Schema内容包裹起来，放在合约cpp文件的开始处。
如何将合约Schema与合约代码放在一起的介绍可参考使用脚本编译合约小节。
在线编辑合约文件完成后，点击提交即可开始编译。

4.6. 合约间调用

4.6.1. 合约运行的上下文环境

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。 这些信息可以通过下的API获取到:

- Identity GetSender();

> 返回一个Identity类型的数据，该数据表示本次调用是谁发起的，消耗的gas是由发起者提供的。

- uint64_t GetValue();

> 返回一个int64_t类型的数据，该数据表示表示本次调用调用者给了本合约多少资产

- uint64_t GetGas();

> 返回一个uint64_t类型的数据，该数据表示表示当前合约还剩余多少可用gas

- Identity GetSelf();

> 返回一个Identity类型的数据，该数据表示表示返回本合约的唯一标识–Identity

- Identity GetOrigin();

> 返回交易发起者的id。这是一个不随着调用深度变化的值，例：账户alice调用了A合约，A合约调用了B合约，B合约调用了C合约...，无论调用层次多深，执行GetOrigin()得到的都是alice的账户Identity。

4.6.2. 普通调用

普通调用的概念为在发生 `CallContract` 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，其对存储的影响体现在B合约相关的成员变量存储中。当普通调用发生时，需要指定被调用合约的Identity，转移给被调用合约的资产数量，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. A–>B
   B执行过程中，GetSender()是A; B的代码操作的都是B合约本身的存储。GetSelf()得到的是B的合约

id。

2. A->B->C
   B执行过程中，GetSender()是A，GetSelf()是B; C执行过程中，GetSender()是B，GetSelf()是C。

普通调用使用`CallContract`，基本形式为:

```
auto ret = CallContract<TYPE>(contractId, methodName, value, gas,
args...);
```

其中TYPE是被调用合约方法的返回值类型。ret 是一个自动模板结构，其定义如下

```
template<class TYPE>
struct {
    int code;   //如果被调用合约没有异常终止，则该值为0，否则为1
    std::string msg;    //如果被调用合约异常终止，则该值就是异常的消息内容；
如果被调用合约没有出现异常，该值为空字符串
    TYPE result;    //如果合约没有出现异常，该值是被调用方法的返回值；若出现
了异常，该值无意义。若TYPE为void，则没有该字段
};
```

注意:

1. `CallContract` 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. 不要出现环形调用，即A->B->...->B，若出现环形调用，第一次调用B，也许修改了B中某些值，第二次调用B合约感知不到第一次调用造成的修改。这种情况很难精准预测B合约的行为，请尽量避免。

例:

```
/*A合约*/
#include <mychainlib/contract.h>
using namespace mychain;
class A: public Contract {
public:
    uint32_t last_value;
    /* A合约调用B合约，已知B合约的id为b */
    INTERFACE void CallB() {
```

```
        last_value = 100;
        Identity b = "00112233445566778899001122334455667788990011223344556677889900";
        auto ret = CallContract<uint32_t>(b/*contractID*/, "add"/*methodName*/, 0/*value*/,10000/*gas*/, 11/*x*/, 12/*y*/);
        // B合约中last_value的值为33，A合约中的last_value依然为100
        //ret.result的值为B合约的add方法的返回值，即1234
    }
};

INTERFACE_EXPORT(A,(CallB));
```

```
/*B合约*/
#include <mychainlib/contract.h>
using namespace mychain;
class B: public Contract {
    uint32_t last_value;
public:
    INTERFACE uint32_t add( uint32_t x, uint32_t y) {
        last_value = x + y;
        return 1234;
    }
};
INTERFACE_EXPORT(B, (add))
```

4.7. 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，即本次合约调用所涉及的存储变更和`TransferBalance`都不会生效。注意，如果合约执行过程中调用了`Log`方法产生通知消息，无论合约是否出现异常，这些消息都会被记录在交易回执中。

4.7.1. 异常错误码

异常产生的原因有以下几类:

1. 用户通过调用 `Revert()` 或 `Require()` 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

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
10200	无	gas耗尽
10201	METHOD_NOT_FOUND	找不到用户指定的合约方法
10201	PARAMS_NOT_MATCH	用户指定的参数与合约方法定义不匹配
10201	MYCHAINLIB_ERROR	mychainlib 内部异常，一般是错误的接口使用导致的
10201	RAPIDXML_ERROR	rapidxml 库解析 XML 时出错
10201	LIBCXX_ERROR	C++ 标准库内部异常，一般是使用了平台不支持的`C++`特性导致的
10201	ABORT_CALLED	第三库或者用户自己调用了`abort()` 并且未提供任何错误提示
10201	VM_MEMORY_OUT_OF_BOUNDS	读内存地址超出边界，检查合约代码是否有bug或者合约使用的内存超出了限制。
10201	VM_INTERGER_OVERFLOW	1. 浮点数到整数转换出现了溢出. 2. 用有符号数的最小值除以-1导致溢出。例:char最小值-128，-128/-1为128导致溢出。
10201	VM_DIVIDE_BY_ZERO	除0错误
10201	VM_COVERT_NAN_TO_INT	NAN浮点无法强制转换Int
10201	VM_INVALID_BYTE_CODE	请检查合约是否存在bug。
10201	VM_UNINITIALIZED_TABLE_E	检查合约有没有越界、非法指针

	LEMENT	等异常、合约内存使用超过可用内存等问题。
10201	VM_UNDEFINED_TABLE_INDEX	检查合约有没有越界、非法指针等异常、合约内存使用超过可用内存等问题。
10201	VM_OUT_OF_CALL_STACK	函数调用栈空间耗尽，可能函数调用深度太多，比如递归很深。
10201	VM_VALUE_STACK_EXHAUSTED	数据堆栈溢出，检查是不是递归太多了
10201	VM_HOST_RESULT_TYPE_MISMATCH	系统函数返回值类型不匹配
10201	VM_ARGUMENT_TYPE_MISMATCH	函数参数不匹配(参数个数，参数类型)
10201	VM_UNKNOWN_EXPORT	目标函数未定义 1.检查编译器是否被擅自修改过或恶意修改过 2.检查字节码是否被擅自修改或恶意修改过
10201	VM_EXPORT_KIND_MISMATCH	被导出的对象不支持该操作 1.检查编译器是否被擅自修改过或恶意修改过 2.检查字节码是否被擅自修改或恶意修改过 3. 不要用extern导出变量，函数
10201	VM_BUFFER_OVERFLOW	合约中出现了缓存溢出的情况。 可能的原因： 1:擅自或恶意修改了编译器中的文件。 2:生成的字节码被擅自修改。 3:若确定没有以上情况，请给我们提交一个bug。
10201	INVALID_INPUT_DATA	合约无法解析交易传入的方法名和参数列表，请确定一下传入的参数个数、类型是否正确。或者是合约调用合约过程中，传入的参数个数、类型是否正确

10500	无	内部异常，请联系管理员。
10622	无	合约在更新、部署或调用时发现合约字节码不是合法的wasm字节码，若在部署合约或更新合约出现，请检查使用的wasc文件是否正确。常见的情况是: 使用了高版本的mycdt编译合约字节码，然后尝试将该字节码部署在较低版本的mychain上。若mychain平台从低版本升级到高版本，注意必须发送一个交易激活升级（如何激活请联系管理员）。
10623	无	合约在部署或更新时无法识别合约格式，排查是不是正确使用了wasc文件。

4.7.2. 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不受影响。
- A->B，如果在调用B之后A合约出现异常，A合约造成的一切世界状态变化都不会生效。B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”指的是存储变更和`TransferBalance`函数造成的变化。而`Log(data,topics)`所产生的事件，无论合约是否出现异常，都会进入交易回执中。

4.7.3. 合约调用栈信息

合约异常终止时，虚拟机会在交易回执中新增一条主题为“backtrace”的 log，记录合约退出时的调用栈信息。

如果编译合约时保存了函数名（使用编译选项 `-dump-names`），那么调用栈信息会包含每个被调用的函数名，如:

```
#0: revert_internal()
#1: Contract::revert()
#2: std::__1::basic_string<char, std::__1::char_traits<char>, st
d::__1::allocator<char> > mychain::execute_interface<Contract, voi
d, Contract>(void (Contract::*)(), std::__1::basic_string<char, st
d::__1::char_traits<char>, std::__1::allocator<char> > const&)
#3: apply
```

如果没有保存函数名，那么调用栈会包含每个被调用函数的编号，如：

```
#0: $func_55
#1: $func_60
#2: $func_58
#3: $func_56
```

假设用户部署合约时没有保存函数名，得到了由函数编号组成的调用栈信息，此时仍可以借助 MYCDT 把函数编号映射到原始函数名，方法如下：

使用 `my++` 重新编译原始的合约代码，并指定选项 `-dump-names`，除了生成 wasc 字节码文件外，还会生成一个后缀名为 names 的文件。names 文件中按照函数编号顺序列出了所有函数名，比如第一行对应第 0 个函数，第二行对应第 1 个函数，以此类推。

4.8. 合约API

**如果没有特殊说明，则该API在非TEE和TEE版本中都支持。
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。**

4.8.1. 区块链交互API

1. `bool CheckAccount(const Identity& id/*in*/);`
   检查当前状态下，id指定的账户是否存在
   请求参数

参数	类型	说明
id	Identity	要查询的账户id

1. 返回值

参数	类型	说明
result	bool	账户id存在则返回true，否则返回false

2. `uint64_t GetBlockNumber();`

   获取上一个区块的区块号。

   返回值

参数	类型	说明
result	uint64_t	最新已经形成的区块区块号

3. `int GetBlockHash(uint64_t block_number/*in*/, std::string& block_hash/*out*/);`

   获取指定区块的hash

   请求参数

参数	类型	说明
block_number	uint64_t	要查询的区块号
block_hash	std::string&	获取到的区块hash，原始字节流(注意不是16进制格式)；若查询的区块不存在，则block_hash的值维持原值

3. 返回值

参数	类型	说明
result	int	如果要查询的区块不存在，则返回1，否则返回0。

4. `uint64_t GetBlockTimeStamp();`

   获取上一个区块的时间戳。

   返回值

参数	类型	说明
result	uint64_t	最新已经形成的区块的时间戳,单位:ms

5. `Identity GetOrigin();`

   获取交易的发起者的id

   返回值

参数	类型	说明
result	Identity	出参，本次合约被调用的交易发

		起者id，参考合约运行的上下文环境。

6. `int GetAuthMap(const Identity& id/*in*/, std::map<std::string,uint32_t>& auth);`

   获取账户权限列表

   请求参数

参数	类型	说明
id	Identity	入参，要查询的账户id
auth	std::map<std::string, uint32_t>&	出参，获取到的<公钥,权重>map。如果返回非0，auth维持原值。注意公钥不是16进制格式。

6. 返回值

参数	类型	说明
result	int	若id指定的账户不存在或被冻结，返回1；否则返回0

7. `int GetBalance(const Identity& id/*in*/, uint64_t& value/*out*/);`

   获取指定账户的余额

   请求参数

参数	类型	说明
id	const Identity&	账户id
value	uint64_t&	该账户的余额。若返回值不为0，value维持原值。

7. 返回值

参数	类型	说明
result	int	若id对应的账户不存在或该账户被冻结则返回1，否则返回0

8. `int GetCode(const Identity& id/*in*/, std::string& code/*out*/);`

   获取指定合约的代码

   请求参数

参数	类型	说明
id	const Identity&	合约id
code	std::string&	获取到的合约的代码，若返回值不为0，code维持原值。若得到

		code值超过1M，则会抛出异常信息

8. 返回值

参数	类型	说明
result	int	若id对应的合约不存在或该合约被冻结则返回1，否则返回0

9. `int GetCodeHash(const Identity& id/*in*/, std::string& hash/*out*/);`

   获取指定合约的代码hash

   请求参数

参数	类型	说明
id	const Identity&	账户id
hash	std::string&	合约的代码hash，原始字节串，非16进制字符串。

9. 返回值

参数	类型	说明
result	int	若成功则返回0，否则返回1。

10. `int GetRecoverKey(const Identity& id/*in*/, std::string& recover_key/*out*/);`

    获取指定账户的恢复公钥

    请求参数

参数	类型	说明
id	const Identity&	账户id
recover_key	std::string&	账户的恢复公钥，若函数返回非0，参数recover_key维持原值

10. 返回值

参数	类型	说明
result	int	若成功则返回0，否则返回1(账户不存在或被冻结)

11. `int GetAccountStatus(const Identity& id/*in*/, uint32_t& status/*out*/);`

    获取指定账户的状态

    请求参数

参数	类型	说明
id	int	账户id

status	uint32_t	账户的状态代码，若函数返回非0，则参数status维持原值。status的定义参见中的`ACCOUNT_STATUS`

11. 返回值

参数	类型	说明
result	int	若id对应的账户不存在返回1，否则返回0

12. `std::string GetTxHash();`

    获取交易的hash（触发交易的hash）

    返回值

参数	类型	说明
result	std::string	本合约触发交易的hash，原始字节串，非16进制格式。

13. `int TransferBalance(const Identity& to/*in*/, unt64_t balance/*in*/);`

    向指定账户转移资产

    请求参数

参数	类型	说明
to	const Identity&	资产接收者的地址
balance	unt64_t	要转移的资产数量

13. 返回值

参数	类型	说明
result	int	转移成功返回0，否则返回1

14. `int Revert(const std::string& exception/*in*/);`

    立即终止该合约的运行，并给区块链发送信号，附带exception内容，表明该合约被异常终止。

    请求参数

参数	类型	说明
exception	const std::string&	报出的错误信息，该自动会被存放到交易回执的output字段中

14. 返回值

参数	类型	说明
result	int	恒为0

15. `int Require(bool condition/*in*/, const std::string& exception/*in*/);`

    若条件`condition`值为`false`，则调用`Revert`。相当于

```
if (!condition) {
    Revert(exception);
}
```

15. 请求参数

参数	类型	说明
condition	bool	需要判断的条件
exception	const std::string&	报出的错误信息，该自动会被存放到交易回执的output字段中

15. 返回值

参数	类型	说明
result	int	恒为0

16. `template <class T>int Log(const T& data/*in*/, const std::vector<std::string>& topics/*in*/);`

产生通知事件，data字段会做序列化，每个topic会以十六进制字符串的格式记录在交易回执中。

请求参数

参数	类型	说明
data	const T&	用户自定义日志内容，仅支持可序列化数据类型
topics	`std::vector<std::string>&`	事件主题,其序列化后的大小不能>1MB，否则会返回失败，事件不会记录在交易回执中。

16. 返回值

参数	类型	说明
result	int	失败返回1，成功返回0

使用示例:

```
std::string topic1 = "good";
std::string topic2 = "morning";
Log("hi"s, {topic1, topic2});
// 上链后对应字段为:
// "data":"026869"
// "topics":["676f6f64","6d6f726e696e67"],
// data字段会先做序列化, "hi"序列化后, 对应"026869"
```

```
// 针对topic字段, "good"对应"676f6f64", "morning"对应"6d6f726e696e67"
```

2. `template <typename T, typename... Args> inline decltype(auto) CallContract(const Identity& contract_id/*in*/, const std::string& method/*in*/, uint64_t value/*in*/, uint64_t gas/*in*/, Args... args);`
   调用另一个合约。**这是Contract类的成员方法，只有继承了Contract的合约才能调用该方法**
   请求参数

参数	类型	说明
contract_id	const Identity&	要调用的合约的Identity
method	const std::string&	被调用的合约方法名称
value	uint64_t	给被调用合约转移的资产数量
gas	uint64_t	给被调用合约的gas数量，填写0表示将本合约所有可用gas给被调用合约使用
args	任意个不定类型数参数	调用合约传入的参数列表，取决于被调用合约方法接收几个参数

2. 返回值

参数	类型	说明
返回值	`struct{int code; T result; std::string msg;}`	返回结构体
code	int	子合约正常: 0 子合约不存在或给子合约转账失败：1 子合约gas不足:10200 子合约执行过程中出现异常:10201 子合约的字节码非法(可能原因子合约不是wasm合约):10622
result	T	如果code为0， 该值为被调用方法的返回值；否则该值无意义。若T为void，则没有该字段
msg	std::string	如果code为0，该值无意义；若code为10201，该值表示被调用合约的抛出的异常信息。

3. `uint64_t GetGas();`

   获取本合约当前剩余的可用gas数量

   返回值

参数	类型	说明
result	uint64_t	本合约当前剩余的可用gas数量

4. `uint64_t GetValue();`

   获取本次合约调用本合约接收的资产数量。如果是交易调用合约，value的值就是交易中指定的这个字段；如果是合约调用合约，value的值就是CallContract()方法中指定的value值。

   返回值

参数	类型	说明
result	uint64_t	获取的value

5. `std::string GetData();`

   获取**触发交易**的data字段值（本次合约调用传入的参数）

   返回值

参数	类型	说明
result	std::string	交易调用本合约时，传递的inputdata内容，原始字节串，非16进制。

6. `Identity GetSender();`

   返回调用者的id（账户或合约），合约执行过程中，消耗sender的gas。

   返回值

参数	类型	说明
result	Identity	调用者id，参考合约运行的上下文环境。

7. `Identity GetSelf();`

   获取本合约的id

   返回值

参数	类型	说明
result	Identity	本合约的id

8. `uint32_t GetRelatedTransactionListSize(const Identity& recipient_id,uint64_t deposit_flag,uint64_t& count);`

   获取指定账户、指定flag的关联存证交易数量

   请求参数

参数	类型	说明
recipient_id	const Identity&	要查询的账户，即在关联存证交

		易中的to账户
deposit_flag	uint64_t	指定的关联存证的flag
count	uint64_t&	查到的结果数量

8. 返回值

参数	类型	说明
result	uint32_t	0： 成功 10350: 失败，交易不存在

9. `uint32_t GetRelatedTransactionList(const Identity& recipient_id,uint64_t deposit_flag,uint64_t start_indexuint32_t number,std::vector<std::string>& tx_list);`

   查询指定账户、指定flag的关联存证交易列表

   请求参数

参数	类型	说明
recipient_id	const Identity&	要查询的账户，即在关联存证交易中的to账户
deposit_flag	uint64_t	指定的关联存证的flag
start_index	uint64_t	指定开始索引
number	uint32_t	指定个数
tx_list	`std::vector<std::string>&`	获取到的结果

9. 返回值

参数	类型	说明
result	uint32_t	0：成功 10350: 失败，start_index或number错误

10. `uint32_t GetTransactionSender(const std::string& tx_hash, Identity& sender);`

    查询指定交易的发送者

    请求参数

参数	类型	说明
tx_hash	const std::string&	要查询的交易hash，长度32字节，只支持原始格式，不支持16进制格式

sender	Identity&	该交易的发送者

10. 返回值

参数	类型	说明
result	uint32_t	0 ： 成功 10315: 失败，交易所在区块超出了查询限制 10316: 失败，交易类型不是存证交易 10310: 失败，交易不存在

11. `uint32_t GetTransactionReceiver(const std::string& tx_hash Identity& receiver);`

    查询指定交易的接收者

    请求参数

参数	类型	说明
tx_hash	const std::string&	要查询的交易hash，长度32字节，只支持原始格式，不支持16进制格式
receiver	Identity&	该交易的接收者

11. 返回值

参数	类型	说明
result	uint32_t	0 ： 成功 10315: 失败，交易所在区块超出了查询限制 10316: 失败，交易类型不是存证交易 10310: 失败，交易不存在

12. `uint32_t GetTransactionTimestamp(const std::string& tx_hash, uint64_t& timestamp);`

    查询指定交易的上链时间

    请求参数

参数	类型	说明
tx_hash	const std::string&	要查询的交易hash，长度32字节，只支持原始格式，不支持16进制格式
timestamp	uint64_t&	该交易的上链时间

12. 返回值

参数	类型	说明
result	uint32_t	0 ： 成功 10315: 失败，交易所在区块超出了查询限制 10316: 失败，交易类型不是存证交易 10310: 失败，交易不存在

13. `uint32_t GetTransactionData(const std::string& tx_hash, std::string& data);`

查询指定交易存证数据

请求参数

参数	类型	说明
tx_hash	const std::string&	要查询的交易hash，长度32字节，只支持原始格式，不支持16进制格式
data	std::string&	该交易的存证数据，其内容是关联性标识和用户数据的组合内容，详情见TODO

13. 返回值

参数	类型	说明
result	uint32_t	0 ： 成功 10315: 失败，交易所在区块超出了查询限制 10316: 失败，交易类型不是存证交易 10310: 失败，交易不存在

14. `uint32_t GetTransactionBlockIndex(const std::string& tx_hash, uint64_t& block_number, uint32_t& index);`

查询指定交易的所在区块和在区块中的顺序

请求参数

参数	类型	说明
tx_hash	const std::string&	要查询的交易hash，长度32字节，只支持原始格式，不支持16进制格式
block_number	uint64_t&	该交易所在区块号

index	uint32_t&	该交易在区块中的顺序

14. 返回值

参数	类型	说明
result	uint32_t	0 ： 成功 10315: 失败，交易所在区块超出了查询限制 10310: 失败，交易不存在

15. `uint32_t GetTransactionDepositFlag(const std::string& tx_hash, uint64_t& deposit_flag);`

    查询指定交易的关联性标识

    请求参数

参数	类型	说明
tx_hash	const std::string&	要查询的交易hash，长度32字节，只支持原始格式，不支持16进制格式
deposit_flag	uint64_t&	该交易的关联性标识

15. 返回值

参数	类型	说明
result	uint32_t	0 ： 成功 10315: 失败，交易所在区块超出了查询限制 10316: 失败，交易类型不是存证交易 10310: 失败，交易不存在

16. `uint32_t GetConfidentialDepositData(const std::string& tx_hash, std::string& data);`

    查询指定交易机密存证数据

    请求参数

参数	类型	说明
tx_hash	const std::string&	要查询的交易hash，长度32字节，只支持原始格式，不支持16进制格式
data	std::string&	该交易的存证数据，其内容是关联性标识和用户数据的组合内容，详情见TODO

16. 返回值

参数	类型	说明
result	uint32_t	0 ： 成功 10315: 失败，交易所在区块超出了查询限制 10316: 失败，交易类型不是存证交易 10310: 失败，交易不存在

17. `uint32_t GetConfidentialDepositFlag(const std::string& tx_hash, uint64_t& deposit_flag);`

查询指定交易机密存证的关联性标识

请求参数

参数	类型	说明
tx_hash	const std::string&	要查询的交易hash，长度32字节，只支持原始格式，不支持16进制格式
deposit_flag	uint64_t&	该交易的关联性标识

17. 返回值

参数	类型	说明
result	uint32_t	0 ： 成功 10315: 失败，交易所在区块超出了查询限制 10316: 失败，交易类型不是存证交易 10310: 失败，交易不存在

18. `int CreateContract(const Identity& src, const Identity& new_id);;`

根据合约模版创建新合约

请求参数

参数	类型	说明
src	const Identity&	合约模版地址
new_id	const Identity&	所创建的新合约地址

18. 返回值

参数	类型	说明
result	int	创建成功返回0，否则返回1

		如果创建成功,则新合约: ID:传入的new_id Balance:0 AuthMap:和合约模版一致 RecoverKey:和合约模版一致 EncryptionKey:和合约模版一致 key,value对为空

4.8.2. 工具类API

1. `template <class T> std::string pack(T t);`
   将t序列化为std::string
   请求参数

参数	类型	说明
t	T	要序列化的数据，支持的数据类型见可序列化数据类型

1. 返回值

参数	类型	说明
result	std::string	序列化结果

2. `template <typename T> T unpack(const std::string& b);`
   将std::string反序列化为指定类型的数据
   请求参数

参数	类型	说明
b	std::string	要反序列化的数据内容

2. 返回值

参数	类型	说明
result	T	反序列化结果，支持的数据类型见可序列化数据类型

2. pack和unpack使用示例

```
std::string buff = pack("hello"s);
std::string str = unpack<std::string>(buff);
//str的值为hello

std::string buff = pack(1234);
int n = unpack<int>(buff);
```

```
//n的值为1234
```

3. `bool Digest(const std::string& data/*in*/, DigestType type/*in*/, std::string& output/*out*/);`

   返回data的散列值

   请求参数

参数	类型	说明
data	const std::string&	要hash的数据
type	DigestType	散列算法,支持的散列算法: SHA256,SM3
output	std::string&	对data进行散列的结果，原始字节串，非16进制格式

3. 返回值

参数	类型	说明
result	bool	如果输入不支持的type类型，则返回false，如果计算成功则返回true

4. `bool VerifyRsa(const std::string &pk/*in*/, const std::string &sig/*in*/, const std::string &msg/*in*/, DigestType hash_type=SHA256);`

   验证一个RSA签名是否有效

   请求参数

参数	类型	说明
pk	const std::string&	DER格式的公钥
sig	const std::string&	签名，必须是PKCS#1编码的格式
msg	const std::string&	源消息
hash_type	DigestType	散列算法。默认为SHA256，可选值: SHA256, SHA1

4. 返回值

参数	类型	说明
result	bool	如果验证通过返回true，否则返回false

5. `std::string Hex2Bin(const std::string& input/*in*/);`

   Hex字符串转字节串std::string

   请求参数

参数	类型	说明
input	const std::string&	16进制字符串

5. 返回值

参数	类型	说明
result	std::string	若输入的input是非法的16进制字节串，返回空std::string，否则返回转换的结果

6. `std::string Bin2Hex(const std::string& input/*in*/, bool uppercase=false/*in*/);`

   字节串转16进制字符串

   请求参数

参数	类型	说明
input	const std::string&	要转换的字节串
uppercase	bool	是否是大写,默认false

6. 返回值

参数	类型	说明
result	std::string	转换结果

7. `bool Base64Encode(const std::string& input/*in*/, std::string& output/*out*/);`

   base64编码

   请求参数

参数	类型	说明
input	const std::string&	进行base64编码的原数据，不允许为空，若为空则返回false。
output	std::string	base64编码结果。若返回值为false，output不会被修改。

7. 返回值

参数	类型	说明
result	bool	如果编码成功返回true，否则返回false。

8. `bool Base64Decode(const std::string& input/*in*/, std::string& output/*out*/);`

   base64解码

   请求参数

参数	类型	说明
input	const std::string&	base64编码的字符串
output	std::string&	对input进行base64解码的结果，若返回值为false，output不会被修改

8. 返回值

参数	类型	说明
result	bool	如果input是非法的base64字符串，则返回false，否则返回true

9. `bool Ecrecovery(const std::string& hash/*in*/, const std::string signature/*in*/, Identity& id/*out*/);`

   根据hash和签名恢复id

   请求参数

参数	类型	说明
hash	const std::string&	消息的hash
signature	const std::string&	消息的签名
id	Identity&	要恢复的id，如果返回值为false，则Identity不会被修改

9. 返回值

参数	类型	说明
result	bool	如果hash和signature非法或不匹配，则返回false，恢复失败。如果恢复成功则返回true。

10. `bool VerifyMessageSM2(const std::string& pk, const std::string& sig, const std::string& msg);`

    验证SM2国密签名

    请求参数

参数	类型	说明
pk	const std::string&	asn1 ECC 公钥
sig	const std::string&	(r, s, v) (65 bytes) 或 ASN.1 编码的格式 (<=72 字节)
msg	const std::string&	原始消息

10. 返回值

参数	类型	说明

result	bool	如果验证签名成功，返回true； 如果验证签名失败，返回false

11. `bool VerifyMessageECCK1(const std::string& pubkey, const std::string& sig, const std::string& msg, DigestType hash_type);`
    验证ECC椭圆曲线K1产生的签名
    请求参数

参数	类型	说明
pubkey	const std::string&	asn1 ECC pubkey格式 或 65字节长的压缩格式
sig	const std::string&	(r, s, v) (65 字节) 或 ASN.1 编码的格式(<=72 字节)
msg	const std::string&	原始消息
hash_type	DigestType	散列的算法，支持的值: DigestType::SHA256, DigestType::SHA1

11. 返回值

参数	类型	说明
result	bool	如果验证签名成功，返回true； 如果验证签名失败，返回false

12. `bool VerifyMessageECCR1(const std::string& pubkey, const std::string& sig, const std::string& msg, DigestType hash_type);`
    验证ECC椭圆曲线R1产生的签名
    请求参数

参数	类型	说明
pubkey	const std::string&	asn1 ECC pubkey格式 或 65字节长的压缩格式
sig	const std::string&	(r, s, v) (65 字节) 或 ASN.1 编码的格式(<=72 字节)
msg	const std::string&	原始消息
hash_type	DigestType	散列的算法，支持的值: DigestType::SHA256, DigestType::SHA1

12. 返回值

参数	类型	说明

result	bool	如果验证签名成功，返回true； 如果验证签名失败，返回false

13. `int BellmanSnarkVerify(const std::string& verification_key, const std::vector<std::string>& inputs, const std::string& proof);`
    验证zksnarks的proof是否合法
    请求参数

参数	类型	说明
verification_key	const std::string&	setup阶段产生的验证key
inputs	`const std::vector<std::string>&`	公开输入
proof	const std::string&	零知识证明的证据

13. 返回值

参数	类型	说明
result	int	如果验证proof通过，返回0；如果验证签名失败，返回对应的错误码

4.8.3 pedersen密码学API

1. `int32_t RangeProofVerify(const std::string& proof, const std::vector<PC>& pc_list);`
   pedersen范围验证。**仅在非TEE版本中支持**
   请求参数

参数	类型	说明
proof	const std::string&	证据
pc_list	`const std::vector<std::string>&`	pedersen commitment

1. 返回值

参数	类型	说明
result	int32_t	RET_RANGEPROOF_VERIFY_PASS 0X00000100 验证通过 RET_RANGEPROOF_VERIFY_FAIL 0X00000101 验证不通过

		RET_RANGEPROOF_ERROR_* 出错

2. `int AddPedersenCommit(PC& PC_result, const PC& PC_left, const PC& PC_right);`

   PC相加，PC即std::string。**仅在非TEE版本中支持**

   请求参数

参数	类型	说明
PC_result	std::string&	PC相加结果
PC_left	`const std::vector<std::string>&`	相加的左PC
PC_right	`const std::vector<std::string>&`	相加的右PC

2. 返回值

参数	类型	说明
result	int32_t	RET_RANGEPROOF_SUCCESS 0X00000000 成功 RET_RANGEPROOF_ERROR_* 出错

3. `int SubPedersenCommit(PC& PC_result, const PC& PC_left, const PC& PC_right);`

   PC相减，PC即std::string。**仅在非TEE版本中支持**

   请求参数

参数	类型	说明
PC_result	std::string&	PC相减结果
PC_left	`const std::vector<std::string>&`	左PC
PC_right	`const std::vector<std::string>&`	右PC

3. 返回值

参数	类型	说明

result	int32_t	RET_RANGEPROOF_SUCCESS 0X00000000 成功 RET_RANGEPROOF_ERROR_* 出错

4. `int32_t CalculatePedersenCommit(PC& dst_pc, const PC& src_pc, const std::vector<PC>& positive, const std::vector<PC>& negative);`

   PC计算。**仅在非TEE版本中支持**

   请求参数

参数	类型	说明
dst_pc	PC&	PC结果
src_pc	const PC&	PC初值
positive	`const std::vector<PC>&`	正PC列表
negative	`const std::vector<PC>&`	负PC列表

4. 返回值

参数	类型	说明
result	int32_t	RET_RANGEPROOF_SUCCESS 0X00000000 成功 RET_RANGEPROOF_ERROR_* 出错

5. `int PedersenCommitEqualityVerify(const std::vector<PC>& positive, const std::vector<PC>& negative);`

   PC等式验证。**仅在非TEE版本中支持**

   请求参数

参数	类型	说明
positive	`const std::vector<PC>&`	正PC列表
negative	`const std::vector<PC>&`	负PC列表

5. 返回值

参数	类型	说明
result	int32_t	RET_RANGEPROOF_VERIFY_PASS 0X00000100 验证通过 RET_RANGEPROOF_VERIFY_FAIL 0X00000101 验证不通过 RET_RANGEPROOF_ERROR_* 出错

ElGamal隐私保护API（仅在非TEE版本中支持）

1. `int LiftedElgamalContractHomomorphicAdd( const std::string& first, const std::string& second, std::string& res);`

   计算两个密文相加

   请求参数

参数	类型	说明
first	const std::string&	第一个密文原始字节串，长度要求为68字节，否则返回2000
second	const std::string&	第二个密文原始字节串，长度要求为68字节，否则返回2000
res	std::string&	两个密文相加结果，长度为68字节

1. 返回值

参数	类型	说明
result	int	1：执行成功 0: 执行失败，非法的密文 2000: 参数格式错误

2. `int LiftedElgamalContractHomomorphicSub(const std::string& first, const std::string& second, std::string& res);`

   计算两个密文相减

   请求参数

参数	类型	说明
first	const std::string&	第一个密文原始字节串，长度要求为68字节，否则返回2000
second	const std::string&	第二个密文原始字节串，长度要求为68字节，否则返回2000
res	std::string&	两个密文相减结果，长度为68字节

2. 返回值

参数	类型	说明
result	int	1：执行成功 0: 执行失败，非法的密文

		2000: 参数格式错误

3. `int LiftedElgamalScalarMutiply(const std::string& src, uint64_t scalar, std::string& res);`

   同态相乘

   请求参数

参数	类型	说明
src	const std::string&	乘法密文原始字节串，长度要求为68字节，否则返回2000
scalar	uint64_t	乘法标量
res	std::string&	乘法结果，长度为68字节

3. 返回值

参数	类型	说明
result	int	1：执行成功 0: 执行失败，非法的密文 2000: 参数格式错误

4. `int LiftedElgamalContractZeroCheckVerify(const std::string& ciphertext, const std::string& proof, int& verify_result);`

   零值检测

   请求参数

参数	类型	说明
ciphertext	const std::string&	单个Elgamal密文原始字节串，长度要求68，否则返回10001
proof	const std::string&	证据
verify_result	int&	检测结果 1: 检测通过 0: 检测不通过

4. 返回值

参数	类型	说明
返回值	int	若返回结果不为1，则verify_result的值没有意义。 1：函数执行成功 0: 函数执行失败 10001: 参数格式错误

5. `int LiftedElgamalContractRangeVerify(const std::vector<std::string>& ciphertext, const int per_value_bit_size, const std::string& public_key, const std::string& proof, int& verify_result);`

   范围检测

   请求参数

参数	类型	说明
ciphertext	`const std::vector<std::string>&`	密文列表，每个密文要求68字节，否则报错10001
per_value_bit_size	int	验证范围证明时，证明验证的范围
public_key	const std::string&	公钥，长度33字节
proof	const std::string&	范围证明的证据
verify_result	int&	检测的结果 1: 检验通过 0: 检验不通过

5. 返回值

参数	类型	说明
result	int	若返回结果不为1，则verify_result的值没有意义。 1：函数执行成功 0: 函数执行失败 10001: 参数格式错误

4.8.4. 调试类API

1. int print(const char* format, ...);

   打印输出函数

   注：此接口已弃用，请使用新的 `println` 接口。

   请求参数

参数	类型	说明
format	const char*	格式与C99中printf相仿，不支持%f %F %g %e %E %G。

1. 返回值

参数	类型	说明
result	int	恒为0

2. int println(const char* format, ...);

   与 `println` 的功能类似，但是对复杂格式化字符串的支持更加健壮。

   请求参数

参数	类型	说明
format	const char*	格式与 C99 中 printf 相仿

2. 返回值

参数	类型	说明
result	int	成功时返回 0，失败时返回 -1

4.8.5. 时间格式化

wasm 合约内无法直接获取系统时间信息，因为区块链不同节点的时间无法保证一致。用户可以通过 `GetBlockTimeStamp`，或者合约调用传参的的方式，在合约内获取一个有一致性保证的时间信息。

MYCDT 提供了标准的 C/C++ 时间格式化接口，可以把此类时间信息按照用户需求格式化成一个字符串。

接口定义

1. **localtime**

```
struct tm *localtime (const time_t * t);
struct tm *localtime64 (const time64_t * t);
```

1. 把以秒为单位的时间t分解后转换存入struct tm格式。该接口默认时区是CST北京时区。

返回值	参数
分解后的时间信息结构体tm，后续时间格式化函数strftime需要。返回的tm中会被隐含初始化为CST北京时区。	以秒为单位的时间，如果原始时间戳是以毫秒为单位，需要除以1000，转换成秒后再传入。

2. **strftime**

```
size_t strftime (char *__restrict, size_t, const char *__restrict,
  const struct tm *__restrict);
```

2. 根据指定的格式符，格式化tm格式的时间，并把结果放入指定的buffer

返回值	参数
恒为0	参数1：存放格式后结果字符串的buffer指针 参数2：buffer长度 参数3：格式化格式符组合 参数4：struct tm格式的时间信息

3. **gmtime**

```
struct tm *gmtime (const time_t *);
struct tm *gmtime64 (const time64_t *);
```

3. 把以秒为单位的时间t分解后转换存入struct tm格式。该接口默认时区是GMT。

返回值	参数
分解后的时间信息结构体tm，后续时间格式化函数strftime需要。返回的tm中会被隐含初始化为GMT时区。	以秒为单位的时间，如果原始时间戳是以毫秒为单位，需要除以1000，转换成秒后再传入。

注意：默认的 `time_t` 类型内部使用 32 位整数来表示时间，因此无法表示 2038 年以后的时间。使用 `time64_t` 类型和配套的接口可以避免这个问题。

示例一

```
#include <mychainlib/contract.h>
#include <locale_impl.h>
#include <time.h>
#include <locale.h>

using namespace mychain;

class hello:public Contract {
public:
    INTERFACE void hi(uint64_t x ) {

      //长度26，此处是示例值。请设置为实际格式化后字符串的长度。
      //此处指定长度可以大于实际长度
      //如果小于实际长度，则会发生截断
      char buffer[26];
```

```
16
17        struct tm *pt;
18
19        //此处假定1585018706922是调用GetBlockTimeStamp()获得的毫秒时间戳
20        //其长度是64位整型
21        //因此需要先除以1000转换为秒
22        time_t mytime = (time_t)(1585018706922 / 1000);
23
24        //CST: 20200324105826
25        pt = localtime(&mytime);
26        strftime(buffer, 26, "%Y%m%d%H%M%S", pt);
27        println("CST: %s\n", buffer);
28
29        //Full: Tue Mar 24 10:58:26 2020
30        strftime(buffer, 26, "%c", pt);
31        println("Full: %s\n", buffer);
32
33        //GMT: 20200324025826
34        pt = gmtime(&mytime);
35        strftime(buffer, 26, "%Y%m%d%H%M%S", pt);
36        println("GMT: %s\n", buffer);
37    }
38 };
39 INTERFACE_EXPORT(hello, (hi))
```

示例二

下面给出使用std::put_time接口来格式化时间戳的例子，请注意：由于该接口引入了stringstream，可能会导致合约体积显著增大。优点是：不用显式的管理格式化后的字符串buffer了。

```
1 #include <mychainlib/contract.h>
2 #include<iomanip>
3 #include<ctime>
4 #include<sstream>
5 #include<string>
6
7 using namespace mychain;
8
```

```
class hello:public Contract {
public:
    INTERFACE void hi(uint64_t x ) {

            //此处假定1585018706922是调用GetBlockTimeStamp()获得的毫秒时间戳
            //其长度是64位整型
            //因此需要先除以1000转换为秒
            std::time_t mytime = (std::time_t)(1585018706922 / 1000);

            //也可以调用gmtime接口格式化为0时区时间
            std::tm tm = *std::localtime(&mytime);

            std::stringstream ssTp;
            ssTp << std::put_time(&tm, "%Y%m%d%H%M%S");

            //ssTp.str()返回一个string对象
            println("PUT_TIME: %s\n", ssTp.str().c_str());
    }
};
INTERFACE_EXPORT(hello, (hi))
```

示例三

下面使用 `time64_t` 类型来表示 2038 年以后的时间。

```
#include <mychainlib/contract.h>
#include <ctime>
#include <iomanip>
#include <sstream>

class Hello {
public:
    INTERFACE void hi() {
        time64_t t = 2234491342; // 某个 2040 年的时间

```

```
        std::ostringstream oss;
        oss << std::put_time(localtime64(&t), "%c");
        mychain::println("%s", oss.str().c_str());
        // 输出: Mon Oct 22 12:02:22 2040

        oss.str("");
        oss << std::put_time(gmtime64(&t), "%c");
        mychain::println("%s", oss.str().c_str());
        // 输出: Mon Oct 22 04:02:22 2040
    }
};
INTERFACE_EXPORT(Hello, (hi))
```

4.8.6. 第三方库

- XML 库 版本v1.1.0，需要包含头文件

```
#include <third_party/rapidxml/rapidxml.hpp>
#include <third_party/rapidxml/rapidxml_print.hpp>
```

- 使用示例

```
#include <mychainlib/contract.h>
#include <third_party/rapidxml/rapidxml.hpp>
#include <third_party/rapidxml/rapidxml_print.hpp>

using namespace mychain;  // NOLINT

class TestXMLParse : public Contract {
public:
    INTERFACE void XMLParse() {
        std::string for_parse = R"(<?xml version="1.0" encoding
="UTF-8"?>
<EllipsoidParams>
  <Datum Name="BeiJing54" SemiMajorAxis="6378245.0" Flattening="2
98.3"/>
  <Datum Name="XiAn80" SemiMajorAxis="6378140.0" Flattening="298.
257"/>
  <Datum Name="CGCS2000" SemiMajorAxis="6378137.0" Flattening="29
```

```
8.257222101"/>
</EllipsoidParams>)";

        // need memory more than 64*1024, so the stack can't hold it, use "static" to
        // make it allocate in global
        static rapidxml::xml_document<> doc;
        doc.parse<0>(for_parse.data());

        const rapidxml::xml_node<>* ellipsoid = doc.first_node("EllipsoidParams");
        if (NULL != ellipsoid) {
            for (rapidxml::xml_node<>* datum = ellipsoid->first_node("Datum");
                 NULL != datum; datum = datum->next_sibling()) {
                std::string szTmp;
                for (rapidxml::xml_attribute<char>* attr =
                        datum->first_attribute("Name");
                    attr != NULL; attr = attr->next_attribute())
{
                    szTmp.append(attr->name());
                    szTmp.append(": ");
                    szTmp.append(attr->value());
                    szTmp.append(", ");
                }
                println("%s\n", szTmp.c_str());
            }
        }
    }
};

INTERFACE_EXPORT(TestXMLParse, (XMLParse))
```

- 智能合约平台目前支持 rapidjson 和 jsoncpp 两种 json 解析库，用户可以根据自己的需要选择使用。下面展示两种 json 解析库的简单用法，用户可以到官方网站查询更详细的使用方法。

rapidjson 使用示例

```
#include <mychainlib/contract.h>
```

```
#include <third_party/rapidjson/document.h>
#include <third_party/rapidjson/stringbuffer.h>
#include <third_party/rapidjson/writer.h>

using namespace mychain;  // NOLINT

class TestJsonParse : public Contract {
public:
    INTERFACE void JsonParse() {
        using namespace ::rapidjson;
        std::string stringFromStream = R"({
    "dictVersion": 1,
    "content":
    [
        {"key": "word1", "value": "单词1"} ,
        {"key": "word2", "value": "单词2"} ,
        {"key": "word3", "value": "单词3"} ,
        {"key": "word4", "value": "单词4"} ,
        {"key": "word5", "value": "单词5"}
    ]
    })";

        // ---------------------------- read json ---------------
-----
        // parse json from string.
        using rapidjson::Document;
        Document doc;
        doc.Parse<0>(stringFromStream.c_str());
        if (doc.HasParseError()) {
            rapidjson::ParseErrorCode code = doc.GetParseError();
            Require(code == kParseErrorNone, "parse from string")
;
        }

        // use values in parse result.
        using rapidjson::Value;
        Value& v = doc["dictVersion"];
        if (v.IsInt()) {
            println("%d\n", v.GetInt());
        }
```

```
40
41          Value& contents = doc["content"];
42          if (contents.IsArray()) {
43              for (size_t i = 0; i < contents.Size(); ++i) {
44                  Value& v = contents[i];
45                  Require(v.IsObject(), "parse error");
46                  if (v.HasMember("key") && v["key"].IsString()) {
47                      println("%s\n", v["key"].GetString());
48                  }
49                  if (v.HasMember("value") && v["value"].IsString()
) {
50                      println("%s\n", v["value"].GetString());
51                  }
52              }
53          }
54          // ---------------------------- write json --------------
------
55          println("add a value into array\n");
56
57          Value item(Type::kObjectType);
58          item.AddMember("key", "word5", doc.GetAllocator());
59          item.AddMember("value", "单词5", doc.GetAllocator());
60          contents.PushBack(item, doc.GetAllocator());
61
62          // convert dom to string.
63          StringBuffer buffer;                // in rapidjson/str
ingbuffer.h
64          Writer<StringBuffer> writer(buffer);  // in rapidjson/wri
ter.h
65          doc.Accept(writer);
66
67          println("%s\n", buffer.GetString());
68      }
69  };
70
71  INTERFACE_EXPORT(TestJsonParse, (JsonParse))
```

jsoncpp 使用示例

```cpp
#include <mychainlib/contract.h>
#include <third_party/jsoncpp/json.h>

using namespace mychain;

class TestJsoncpp : public Contract {
public:
    INTERFACE void parse()
    {
        std::string strValue =
        "{                      \
          \"key\": \"value1\",  \
          \"array\": [          \
             {\"arraykey\": 1}, \
             {\"arraykey\": 2}  \
           ]                    \
         }";

        Json::Reader reader;
        Json::Value root;
        if (reader.parse(strValue, root))
        {
            if (!root["key"].isNull())
            {
                std::string strValue= root["key"].asString();
                println("%s\n", strValue.c_str());
            }

            Json::Value arrayObj = root["array"];
            for (int i=0; i<arrayObj.size(); i++)
            {
                int iarrayValue = arrayObj[i]["arraykey"].asInt()
;
                println("%d\n", iarrayValue);
            }
        }
    }
};

```

```
39 INTERFACE_EXPORT(TestJsoncpp, (parse))
```

- 定点数库，需要包含头文件

```
1 #include <third_party/bcmath/bcmath_stl.h>
```

定点数库 使用示例

```
 1 #include <mychainlib/contract.h>
 2 #include <third_party/bcmath/bcmath_stl.h>
 3 using namespace mychain;
 4 using namespace bcmath;
 5
 6 void test_bcmath();
 7
 8 class BCMathTest: public Contract {
 9 public:
10     INTERFACE void run() {
11         test_bcmath();
12     }
13 };
14
15 INTERFACE_EXPORT(BCMathTest, (run))
16
17 void test_bcmath()
18 {
19     BCMath::bcscale(4); //Num Decimals
20     BCMath test("-5978");
21     std::string gold;
22
23     auto require = [](bool cond, const char* msg) {
24         Require(cond, "Test failed: "s + msg);
25     };
26
27     test^=30; //Pow, only integers. Not work decimals.
28     gold = "19800553066925374953329022278263479633645078658128486138177771480479590017172693860399739519392198484225658611302 4";
```

```
    require(test.toString() == gold, "BigDecimal 1");

    test-=1.23; //sub
    gold = "19800553066925374953329022278263479633645078658128486138177771480479590017172693860399739519392198484225658611302 2.7700";
    require(test.toString() == gold, "BigDecimal 2");

    test*=1.23; //mul
    gold = "24354680272318211192594697402264079949383446749498037949958658920989895721122413448291679608852404135597560091901 8.0071";
    require(test.toString() == gold, "BigDecimal 3");

    test*=-1.23; //mul
    gold = "-29956256734951399766891477804784818337741639501882586678449150472817571736980568541398765918888457086784998913039 2.1487";
    require(test.toString() == gold, "BigDecimal 4");

    BCMath::bcscale(70); //Num Decimals

    BCMath randNum("-5943534512345234545.89989283928392478443534 57");
    BCMath pi("3.14159265358979323846264338327950288419716939937 510582097494459230781640628 62");

    BCMath result1 = randNum + pi;
    BCMath result2 = randNum - pi;
    BCMath result3 = randNum * pi;
    BCMath result4 = randNum / pi;

    gold = "-5943534512345234542.758300185694131545972702316720497115802830600624894179025055407692 1835";
    require(result1.toString() == gold, "Super Precision 1");
    gold = "-5943534512345234549.041485492873718022897989083279502884197169399375105820974944592307 8164";
    require(result2.toString() == gold, "Super Precision 2");
    gold = "-18672164360341183116.911478389507334918090475396299279694387192096235243607911833888728 7186";
```

```
    require(result3.toString() == gold, "Super Precision 3");
    gold = "-1891885794154043400.2804849527556211973567525043250278948318788149660700494315139982452600";
    require(result4.toString() == gold, "Super Precision 4");


    //Other example
    BCMath::bcscale(4); //Num Decimals
    gold = "8023400.1075";
    require(BCMath::bcmul("1000000.0134", "8.0234") == gold, "Other 1");
    gold = "1000008.0368";
    require(BCMath::bcadd("1000000.0134", "8.0234") == gold, "Other 2");

    require(BCMath::bccomp("1", "2") == -1, "Compare 1");
    require(BCMath::bccomp("1.00001", "1", 3) == 0, "Compare 2");
    require(BCMath::bccomp("1.00001", "1", 5) == 1, "Compare 3");
    require(BCMath("1")< BCMath("2"), "Compare 4");
    require(BCMath("1")<=BCMath("2"), "Compare 5");
    require(!(BCMath("1")> BCMath("2")), "Compare 6");
    require(!(BCMath("1")>=BCMath("2")), "Compare 7");
    require(!(BCMath("2")< BCMath("2")), "Compare 8");
    require(BCMath("2")<=BCMath("2"), "Compare 9");
    require(!(BCMath("2")> BCMath("2")), "Compare 10");
    require(BCMath("2")>=BCMath("2"), "Compare 11");

    gold = "123.0125";
    require(BCMath::bcround("123.01254") == gold, "Round 1");
    gold = "-123.013";
    require(BCMath::bcround("-123.01254", 3) == gold, "Round 2");
    gold = "123.01";
    require(BCMath::bcround("123.01254", 2) == gold, "Round 3");
    pi.round(3);
    gold = "3.142";
    require(pi.toString() == gold, "Round 4");

```

```
    BCMath part1("-.123");
    BCMath part2(".123");
    BCMath part3("123");
    require(part1.getIntPart() == "-0", "Int part 1");
    require(part1.getDecPart() == "123", "Dec part 1");
    require(part2.getIntPart() == "0", "Int part 2");
    require(part2.getDecPart() == "123", "Dec part 2");
    require(part3.getIntPart() == "123", "Int part 3");
    require(part3.getDecPart() == "0", "Dec part 3");
}
```

5. 合约编译

5.1 普通 wasm 合约

假设合约文件名字是xxx.cpp。
执行下面的命令即可

```
my++ -o xxx.wasm xxx.cpp
```

如果执行成功会出现3个文件:

- xxx.wasm 是合约的字节码文件
- xxx.abi 是产生的abi定义文件
- xxx.wasc 是 wasm 智能合约的文件，在部署合约时交易中的code字段就是该文件内容
  如果使用-o选项指定的文件名不是以.wasm结尾，那么产生的abi文件就是-o指定的文件名后面直接加.abi。
  例

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

会产生名为`abc`的字节码文件和名为`abc.abi`的abi定义文件。
ABI文件的格式定义请参考ABI定义

`my++` 也可以同时编译多个源文件。比如用户可以在 `contract.cpp` 中定义合约，在 `utils.cpp` 中定义一些工具函数，那么编译命令为：

```
$ my++ contract.cpp utils.cpp -o contract.wasm
```

5.2 AOT wasm 合约

MYCDT 还支持通过 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 。

AOT 合约编译方式和普通 wasm 合约编译基本相同，只需要额外加上 `-aot` 选项：

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

上面的命令会生成合约文件 hello-aot.wasc，部署方式和普通 wasm 合约相同。区块链虚拟机能够自动识别和运行 AOT 格式的智能合约。

注意：

1. AOT 形式的合约执行方式目前还在试验阶段，稳定性可能不如解释执行的普通 wasm 合约。请谨慎使用。
2. 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 MYCHAIN 的使用手册。

5.3 构建静态库

用户可以使用 MYCDT 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(仅支持MAC)

目前 wasm 字节码尚不支持运行时调试功能。为了方便开发者调试代码，MYCDT 支持把合约代码编译成 x86 代码，可在本地离线执行和调试合约。这个过程大致可以这么理解，首先用专用工具myclang将合约编译为动态库文件，然后使用专用工具mydebug加载合约动态库文件并运行。
调试需要特殊版本的mycdt，下载地址: ftp://mychainftp.inc.alipay.net/mycdt/release-0.10.2.15_debug/MYCDT-release-0.10.2.15_debug-Darwin-x86_64.tar.gz

6.1 使用示例

1. 准备合约代码hello.cpp，内容如下:

```
#include <mychainlib/contract.h>
using namespace mychain;
class A : public Contract {
public:
  INTERFACE std::string hello(int n, Identity id, const std::string& msg) {
    std::string ret = std::to_string(n) + ":" + id.to_hex() + ":" + msg;
    return ret;
  }
```

```
 9 };
10 INTERFACE_EXPORT(A, (hello));
```

2. 将合约编译为动态库:

```
1 myclang -g -shared -fPIC -fvisibility=hidden /hello.cpp -o /tmp/he
  llo.so
```

2. 根据您的实际情况填写contract_path，执行完此命令之后，请不用移动、拷贝hello.so文件。注意上面的`-g -shared -fPIC -fvisibility=hidden`4个选项是必须填写的。
3. 部署合约

```
1 mydebug -d /tmp/hello.so -t myhello_1
```

3. `-d 或 --deploy`指定合约文件名；`-t 或 --to`指定部署的合约Identity
4. 调用合约

```
1 mydebug -t myhello_1 -i "hello(int,identity,string)[12345,bob,hi]"
```

4. `-i`或`--input` 指定要调用等方法名和参数值，支持多种写法，见下方**交易选项**
5. 调试合约

   5.1. 使用lldb启动mydebug

```
1 ➜  /tmp lldb mydebug
2 (lldb) target create "mydebug"
3 Current executable set to 'mydebug' (x86_64).
4 (lldb)
```

5. 5.2. 设置断点

```
1 (lldb) breakpoint set --file hello.cpp --line 6
2 Breakpoint 1: no locations (pending).
```

5. 5.3. 指定参数运行

   如果您已经部署过合约，可以不使用-d选项

```
1 (lldb) r -d /tmp/hello.so -i "hello(int32,identity,string)[12345,b
  ob,hi]" -t myhello_1
```

5. 看到以下界面

```
1 Process 36397 stopped
2 * thread #1, queue = 'com.apple.main-thread', stop reason = break
```

```
  point 1.1
3     frame #0: 0x0000000102f31532 hello.so`A::hello(this=0x00007ff
  eefbfa530, n=12345, id=(data_ = "\x81@7@@@@@cY@1\x13@\x17\r@@%@M
  \x1e\vL@\x9eŌ@@"), msg="hi") at hello.cpp:6
4   3    class A : public Contract {
5   4     public:
6   5      INTERFACE std::string hello(int n, Identity id, const st
  d::string& msg) {
7 -> 6        std::string ret = std::to_string(n) + ":" + id.to_hex
  () + ":" + msg;
8   7        return ret;
9   8      }
10  9    };
11 Target 0: (mydebug) stopped.
```

5. 现在可以用lldb工具进行单步调试了，关于lldb请参见 https://lldb.llvm.org/

6.2 mydebug命令选项介绍

6.2.1 基本选项

- `-h [ --help ]` 显示帮助介绍
- `--show` 显示交易选项配置文件示例
- `-c [ --config ] arg` 指定交易配置文件名。下方所有'交易选项'均可以通过一个json配置文件进行配置，配置文件中指定的交易选项优先级低，如果在json文件中和命令行中同时指定了某个交易选项，那么最终结果以命令行中的为准。json文件的格式见下方 **交易选项配置文件** 。

6.2.2 交易选项:

- `-v [ --value ] arg (=0)`
  要给合约转移的balance数量，对应交易value字段
- `-g [ --gas ] arg (=10000000000000000)`
  交易gas值，默认`10000000000000000`
- `-s [ --sender ] arg (=Tester001)`
  sender的identity值，支持短名称或16进制格式的identity，例:

  alice123

0xc60a9d48105950a0cca07a4c6320b98c303ad42d694a634529e8e1a0a16fcdb5

- `-t [ --to ] arg` 指定 调用/部署/升级 的合约identity，支持短名称或16进制全写，例:

  alice123

  c60a9d48105950a0cca07a4c6320b98c303ad42d694a634529e8e1a0a16fcdb5
- `-i [ --input ] arg`
  要调用合约的接口名和参数，对应交易的data字段，默认Init()接口。

  支持编码后的16进制格式，对于简单的接口，可以直接写人类可读的字符串:

  `hello(int32,string)[100,bob]`
  `04496e697400`
- `-d [ --deploy ] arg` 部署合约时用该选项，指定要部署的合约文件名
- `-u [ --update ] arg` 更新合约时用该选项，指定要更新的合约文件名
- `-n [ --number ] arg (=0)` 指定最新区块的高度，默认为0。合约内API:GetBlockNumber()获取到的值由此决定
- `--block_timestamp arg (=0)` 指定最新区块的timesamp，默认为0。合约内API:GetBlockTimeStamp()获取到的值由此决定
- `--database arg (=./mock_data)` 指定数据库目录，默认`./mock_data`，见下方**数据库目录**

6.2.3 交易选项配置文件

交易选项配置文件是一个json文件，您可以将所有的交易选项写进该文件中，该文件中json的键请写选项的全称。
该配置文件允许将调用的合约接口名和参数列表以json格式编写。args中的type表明了参数类型，参数类型的名称和ABI中的一致。示例:

```
{
  "to": "my_hello_world_contract",
  "sender": "Tester001",
  "input": {
    "method": "Run",
    "args": [{
        "type": "int16",
        "value": 123
      },{
```

```
        "type": "float32",
        "value": 0.123
      },{
        "type": "string",
        "value": "there is another json string: {\"age\":19}"
      },{
        "type": "int8[]",
        "value": "1234abcd"
      },{
        "type": "int16[]",
        "value": [-32768,-1,0,1,32767]
      },{
        "type": "identity[]",
        "value": [
          "b1931361f6f7b4c5168a2f17ddf67ace5d234fc8ebdd8e239cc555bb1b913666",
          "0xb1931361f6f7b4c5168a2f17ddf67ace5d234fc8ebdd8e239cc555bb1b913666",
          "Tester001"
        ]
      }
    ]
  }
}
```

6.2.4 数据库目录

数据库目录下有2个json文件，world_state.json和history.json。

- world_state.json里面存储的是当前世界状态信息，当您部署/调用/升级合约时，世界状态的改变都会存储在该文件中，该文件若不存在会自动生成。可以直接打开world_state.json查看合约的kv数据内容。示例:

```
{
    "accounts": "",
    "contracts": {
        "790420269d6bceeedc8325286e54decf15512dd3d9fca92b25a71e280365a541": {
```

```
            "id": "790420269d6bceeedc8325286e54decf15512dd3d9fca92b25a71e280365a541",
            "balance": "0",
            "recovery_key": "0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000",
            "recovery_time": "0",
            "status": "0",
            "code": "cffaedfe2f7072697661746 52f746d702f737461746 9635f73656e6465722e736f",
            "storage_root": "76be8b528d0075f7aae98d6fa57a6d3c83ae480a8469e668d7b0af968995ac71",
            "code_hash": "892f4238009170defb29fffea28e7d796c0a3ab96b4f760ded25ee58cfd0056c",
            "encryption_key": "",
            "version": "2",
            "auth_map": "",
            "storage": {
                "746869735f69735f6b65795f61": "7468697320697320 616c7565206f66206b65792061",
                "746869735f69735f6b65795f62": "7468697320697320 616c7565206f66206b65792062"
            }
        }
    }
}
```

- history.json里面存储的是历史交易，历史区块头等信息，运行合约调试过程中history.json不会被更改。如果您的合约中没有访问历史区块、历史交易等动作，则该文件不起作用。您可以根据需要改写history.json从而实现合约读历史区块/历史交易的模拟行为。

```
{
    "block_headers": {
        "1": {
            "hash": "0000000000000000000000000000000000000000000000000000000000000000a",
            "version": "281474976710658",
            "number": "0",
            "parent_hash": "0000000000000000000000000000000000000000000000000000000000000000",
```

```
            "transaction_root": "0000000000000000000000000000000000000000000000000000000000000000",
            "receipt_root": "0000000000000000000000000000000000000000000000000000000000000000",
            "state_root": "0000000000000000000000000000000000000000000000000000000000000000",
            "gas_used": "0",
            "timestamp": "0",
            "log_bloom": "00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000"
        }
    },
    "deposit_relation": {
        "0000000000000000000000000000000000000000000000000000000000000001": {
            "112233": [
                "0000000000000000000000000000000000000000000000000000000000000011",
                "0000000000000000000000000000000000000000000000000000000000000012"
            ]
        },
        "0000000000000000000000000000000000000000000000000000000000000002": {
            "445566": [
                "0000000000000000000000000000000000000000000000000000000000000021",
                "0000000000000000000000000000000000000000000000000000000000000022"
            ]
        }
    },
    "transactions": {
```

```
        "0000000000000000000000000000000000000000000000000000000000000008": {
            "hash": "0000000000000000000000000000000000000000000000000000000000000008",
            "type": "255",
            "timestamp": "0",
            "nonce": "0",
            "period": "0",
            "from": "b1931361f6f7b4c5168a2f17ddf67ace5d234fc8ebdd8e239cc555bb1b913666",
            "to": "0000000000000000000000000000000000000000000000000000000000000000",
            "value": "0",
            "gas": "0",
            "group_id": "00000000000000000000000000000000000000000",
            "version": "2",
            "data": "",
            "extensions": "",
            "signature": ""
        }
    }
}
```


7. 合约部署和调用

部署合约和调用合约请参照 Java/C++/JavaScript 的相应 SDK 文档

- java SDK
- C++ SDK
- javascript SDK

8. 合约升级

对于已经部署的合约，可以进行升级，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，不会执行合约中的任何方法。

通过发送一个特殊的交易进行合约升级，请参考SDK中关于"合约升级"的章节。

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

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

- 访问未初始化变量

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

整数运算

- 有符号整数溢出

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

- 非法位移运算

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

- 非法数学运算

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

指针操作

- 访问空指针

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

- 访问分配空间为 0 的指针

```
int* p = new int[0];
*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10201。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10201。

9.5.3 合约间调用深度限制

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1024，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VM_STACK_OVERFLOW即10002，请您在写合约时留意。

10 C++ WASM合约代码覆盖率收集

代码覆盖率信息是软件开发的一个重要技术指标。从0.10.2.14版本开始，MYCDT支持收集C++ WASM合约的代码覆盖率信息，开发者仅需要在编译合约时增加参数`--coverage`来指示编译器进行代码插桩即可，然后正常部署运行合约，代码覆盖率信息会通过交易的Log返回，对应的topic为`coverage`，信息存放于data字段。

请注意，此种模式下编译出来的合约字节码文件，不能用于生成环境。

下面以我们的老朋友hello world合约为例来演示如何收集合约代码覆盖率。

10.1 编译合约

合约源代码如下：

```cpp
#include <mychainlib/contract.h>

class Hello : public mychain::Contract {
public:
    INTERFACE void hi() {
        mychain::print("Hi");
    }
};

INTERFACE_EXPORT(Hello, (hi))
```

编译时使用参数`--coverage`：

```
$ my++ hello.cc -o hello.wasm --coverage
```

编译完成后，除了合约字节码 hello.wasm 外，在当前目录下还会产生一个名为 hello.gcno 的文件。这个文件在生成覆盖率报告时将会用到，请不要删除。

10.2 运行合约

合约在链上执行时，会把动态生成的覆盖率信息保存到log中，topic为`coverage`，因此开发者需要在SDK侧获取topic为`coverage`的log信息，每条log的内容通过LEB128解码可以得到两个字段：覆盖率文件路径和覆盖率信息。开发者需要根据文件路径信息来保存覆盖率信息。覆盖率信息文件以`.gcda`结尾，对应于上面一步MYCDT生成的`.gcno`文件。

不同SDK收集覆盖率的具体实现不同，以C++SDK为例：

```cpp
bool CallHi(const Identity& from) {
    // 调用合约
    auto params = std::make_shared<WASMParameter>();
    params->SetFunctionSelector("hi");
    auto req = std::make_shared<CallContractRequest>(from, contra
ct_id, VMType::WASM, params, 0);
    auto res = client_ptr->GetContractService()->CallContract(req
);
```

```
    if (res->GetReturnCode() != ErrorCode::SUCCESS || res->tx_receipt_.result_ != 0) {
        LOG_ERROR(env->logger_, "call contract failed, error code:%s, receipt result:%s",
                  StringForErrorCode(res->GetReturnCode()).c_str(),
                  StringForErrorCode(res->tx_receipt_.result_).c_str());
        return false;
    }

    // 保存覆盖率信息
    std::string topic = "636f766572616765"; // hex from "coverage"
    for (auto& log : res->tx_receipt_.logs_) {
        if (!log.MatchTopic(topic))
            continue;

        WASMOutput output;
        output.SetOutput(log.log_data_);
        auto path = output.GetString();
        auto data = output.GetString();

        std::ofstream fs(path, std::ios::binary);
        if (!fs.is_open())
            LOG_WARN(env->logger_, "failed to save file: %s", path.c_str());
        fs.write(data.data(), data.size());
    }
    return true;
}
```

10.3 生成覆盖率报告

使用MYCDT提供的llvm-gcov工具处理编译时生成的hello.gcno文件和运行时得到的hello.gcda文件，得到直观的覆盖率报告。

```
$ llvm-gcov hello.gcda
```

上述命令会隐式的寻找hello.gcno文件。最终会在当前目录生成一个名为hello.cc.gcov文件，用文本编辑器打开该文件可以看到代码覆盖率信息，如下所示：

```
        -:    0:Source:hello.cc
        -:    0:Graph:hello.gcno
        -:    0:Data:hello.gcda
        -:    0:Runs:1
        -:    0:Programs:1
        -:    1:#include <mychainlib/contract.h>
        -:    2:
        1:    3:class Hello : public mychain::Contract {
        -:    4:public:
        1:    5:    INTERFACE void hi() {
        1:    6:        mychain::print("Hi");
        1:    7:    }
        -:    8:};
        -:    9:
        2:   10:INTERFACE_EXPORT(Hello, (hi))
```

10.4 生成网页版覆盖率报告

如果要生成网页形式的覆盖率报告，需要使用1.13版本以上的第三方工具lcov，需要开发者自行安装，MYCDT中暂不单独提供。报告生成命令为：

```
$ lcov -c -d . -o coverage.info --gcov-tool llvm-gcov
$ genhtml coverage.info -o coverage-html
```

上述命令执行完成后会生成一个coverage-html目录，用浏览器打开该目录下的index.html文件即可看到网页版的覆盖率报告。

11. 参考文献