# 中移链

# 智能合约

# 快速开发指导

# 目录

# 基础

## 区块链(blockchain)

区块链是一种新型的基础设施软件。从开发人员的角度来看，您可以在区块链上创建不可变的事务历史。一旦这些事务存储在区块链上，就不能被不留痕迹的删除、修改或伪造。智能合约是一段代码，可以在区块链上执行，并将合约执行状态作为该区块链上不可变历史的一部分。因此，开发人员可以把区块链作作为可信的计算环境，在此区块链上，合约的输入/执行/结果都是独立的，不受外部影响。这些特性为开发人员提供了实现某种新的应用的可能性。

### EOSIO

EOSIO 是一种引入区块链体系结构的软件，旨在支持分布式应用程序（“EOSIO 软件”），可用于建立私有和公共区块链网络。这是通过一 个类似于操作系统的结构来实现的，应用程序可以在这个结构上构建。该软件提供帐户、身份验证、数据库、异步通信和跨多个 CPU 内核和/或集群的应用程序调度。这种区块链架构具有扩展到每秒数百万事务的潜力，消除了用户的费用，并允许快速和容易地部署分布式应用程序。

## 技术特点

- C ++ / WASM 虚拟机

EOSIO 使用 C ++作为其智能合约编程语言。 C++是世界各地开发人员的流行编程语言。因此，任何熟悉 C ++的开发人员都不需要学习新的编程语言，并且已经准备好学习 EOSIO 的 API，这将在本次入门系列中介绍。一旦熟悉 EOSIO 的 API，开发人员就能够使用 C ++编写 EOSIO 智能合约。

底层 EOSIO 是一个 WebAssembly（WASM）虚拟机，用来执行智能合约代码。WASM 还被谷歌，微软，苹果等公司开发的其他重要互联网基础设施软件使用。使用 WASM 的设计选择，使 EOSIO 能够利用经过广泛地社区维护、改进、优化和实战考验的编译器和工具链。此外，采用 WASM 标准还使编译器开发人员更容易将其他编程语言移植到 EOSIO 上。

- **高吞吐量和可扩展性**

EOSIO 旨在实现高的事务吞吐量。使用委托证明（DPOS）的共识机制，EOSIO 区块链网络不需要等待所有节点完成事务。与其他共识机制相比，这可以实现更高的事务吞吐量。

- **更快的确认和更低的延迟**

为了提供良好的用户体验，EOSIO 旨在实现事务确认的低延迟，以便开发人员构建的应用程序可以与其他非区块链的、中心式的方案竞争。

- **无费且成本可预测的区块链**

在 EOSIO 上构建的应用程序可以采用免费模式，用户无需支付基础设施成本和交易费用。 EOSIO 区块链的独特之处在于基础设施资源受到抵押（stake）机制的约束控制。

- **综合权限模式**

EOSIO 具有权限系统，可以为各种应用场景创建自定义权限模式。例如，您可以创建自定义权限并使用它来保护智能合约的一个特定功　能。您还可以拆分权限，用多个有不同权重的权限的帐户调用智能合约功能。这种权限系统允许开发人员在灵活的基础架构之上构建应用程序，而无需重新发明轮子。

- **可升级**

部署在基于 EOSIO 的区块链上的应用程序是可升级的。这意味着只要提供了足够的权限，开发人员就可以部署修正的代码，添加功能和/或更改应用程序逻辑。作为开发人员，您可以迭代您的应用程序。

- **减少能源消耗**

使用 DPOS 作为共识机制，与其他一致性算法相比，EOSIO 消耗更少的能量来验证交易并确保区块链的安全性。

- **可编程经济学与治理**

任何基于 EOSIO 的区块链的资源分配和治理机制都是可编程的。治理和资源分配通过智能合同进行编程。开发人员只需修改系统智能合约即可更改 EOSIO 区块链的资源分配和治理规则。使用系统智能合约时，链上治理变得更加简单，因为不需要修改基础层代码来对区块链进行更改。

### EOSIO 智能合约

EOS 智能合约是运行在采用 eosio 协议的 eos 区块链上运行的程序。智能合约（Smart Contact），里面有两个概念：action（动作），transaction（交易）的概念。action 其实它也是对一个智能合约中的某个函数的调用。transaction 是由一个或者多个 action 组合而成的关系，就是在一个 transaction 里，可以包含多个 action，这样你可以在一个 transaction 里签一次名，就可以调多个函数，做一组操作。

# 技能要求

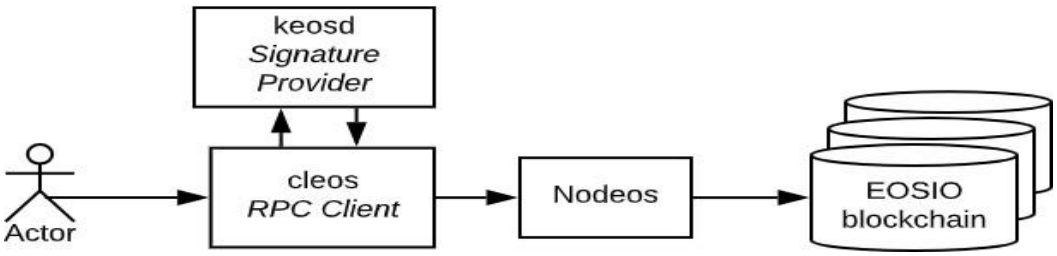
智能合约的开发人员需要具备 C++的语法知识，最好还能有一点数据库的简单经验。目前的 CDT（合约开发工具）是一个 c++编译器，使用 C++11 标准的语法，相比过去的 C++有很大的扩展，很多的语法前所未见的，但这没有关系，eos 智能合约本身类似于数据库的脚本，通常不会有复杂的逻辑，可以通过借鉴 https://github.com/eosio/eosio.contracts 使用的程序结构和语法以及使用的类库，很快就可以写出自己的程序。

# 资源网站

- EOS 官方网站：https://eos.io
- 博客：https://medium.com/eosio

- 开发人员文档：https://developers.eos.io/
- 问题解答：https://eosio.stackexchange.com/
- 开发人员 telegram 组：https://t.me/joinchat/EaEnSUPktgfoI-XPfMYtcQ
- 社区 telegram 组：https://t.me/EOSProject
- 白皮书：https://github.com/EOSIO/Documentation/blob/master/TechnicalWhitePaper.md
- 产品线路图：https://github.com/EOSIO/Documentation/blob/master/Roadmap.md
- CDT: https://github.com/EOSIO/eosio.cdt

# 技术栈




图：与 eosio 区块链的交互

## 智能合约（Smart Contract）与合约开发工具（CDT）

EOSIO.CDT 是 WebAssembly (WASM)的工具链和一组工具，用于构建 EOSIO 平台的合约。除了作为一个通用的 WebAssembly 工具链之外，EOSIO 特定的优化还可用来支持构建 EOSIO 智能合约。这个新的工具链是围绕 Clang 7 构建的，这意味着 EOSIO.CDT 拥有来自 LLVM 的最新优化和分析。然而，由于 WASM 目标仍然被认为是实验性的，一些优化是不可用的或不完整的。

## Nodeos

Nodeos 是核心 EOSIO 节点守护程序。 插件可用于配置 nodeos 的各种功能。Nodeos 处理所有对等网络，合同代码调度和区块链数据持久层。 对于开发环境，nodeos 还可以建立单节点区块链网络。

## cleos

cleos 是命令行工具，它使开发人员能够与区块链交互，部署/测试 EOSIO 智能合约，查询区块链状态等。它是通过 nodeos 和 keosd 的 RPC API 与钱包和区块链交互。

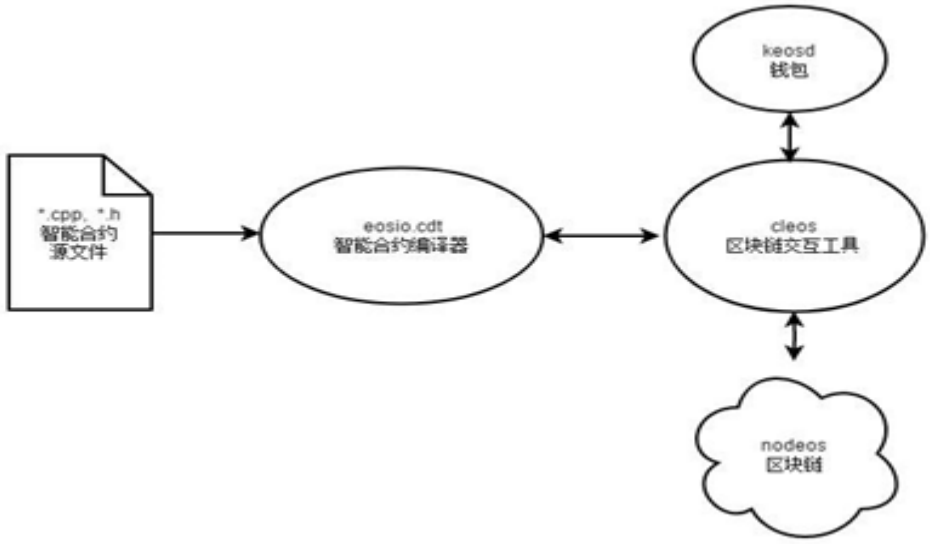
## keosd

keosd 是 EOSIO 自带的，用于管理 EOSIO 账户的密钥。

# 开发环境

- 区块链：开发好的智能合约将会部署和运行在区块链上，基本的由 nodeos，keosd，cleos 三个程序组成。你也可以利用已有的测试网络，或者在自己的环境钟建立一个自己的区块链。
- 源码编辑器：任何自己喜欢的文本编辑器都可以编辑代码，选择带有 c++语法功能的编辑器感觉会好一点。常用的编辑器包括：
  - eclipse c++
  - vscode
  - EOS studio
  - notepad
- 编译工具：eosio.cdt（合约开发工具）

合约开发的流程图如下：




# 搭建开发环境

## 1. 安装 EOSIO 二进制文件

Ubuntu 20.04 软件包安装:

```
wget https://github.com/eosio/eos/releases/download/v2.1.0/eosio_2.1.0-1-ubuntu-20.04_amd64.deb
sudo apt install ./eosio_2.1.0-1-ubuntu-20.04_amd64.deb
```

Ubuntu 18.04 软件包安装

```
wget https://github.com/eosio/eos/releases/download/v2.1.0/eosio_2.1.0-1-ubuntu-18.04_amd64.deb
sudo apt install ./eosio_2.1.0-1-ubuntu-18.04_amd64.deb
```

CentOS 7 RPM 包安装

```
wget https://github.com/eosio/eos/releases/download/v2.1.0/eosio-2.1.0-1.el7.x86_64.rpm
sudo yum install ./eosio-2.1.0-1.el7.x86_64.rpm
```

CentOS 8 RPM 软件包安装

```
wget https://github.com/eosio/eos/releases/download/v2.1.0/eosio-2.1.0-1.el8.x86_64.rpm
sudo yum install ./eosio-2.1.0-1.el8.x86_64.rpm
```

## 2. 安装 EOSIO.CDT

CDT(contract development tools)合约开发工具，用来编译合约源代码，生成 wams 格式文件和 abi 文件，然后将生成的文件发布到区块链上。

### Ubuntu (Debian)

安装

```
wget    https://github.com/eosio/eosio.cdt/releases/download/v1.8.0/eosio.cdt_1.8.0-1-ubuntu-18.04_amd64.deb
sudo apt install ./eosio.cdt_1.8.0-1-ubuntu-18.04_amd64.deb
```

卸载

```
sudo apt remove eosio.cdt
```

### CentOS/Redhat (RPM)

**安装**

```
wget            https://github.com/eosio/eosio.cdt/releases/download/v1.8.0/eosio.cdt-1.8.0-1.el7.x86_64.rpm
sudo yum install ./eosio.cdt-1.8.0-1.el7.x86_64.rpm
```

**卸载**

```
sudo yum remove eosio.cdt
```

## 3. 创建开发钱包

私钥存储在本地 Keosd 中。私钥是非对称密码学使用的公私钥对的一半。相应的公钥存储在区块链上并与帐户相关联。正是这些密钥用于保护帐户和签署交易。

**创建钱包**

```
cleos wallet create --to-console
// 返回钱包的钱包名称及钱包密码，需要手动保存密码，防止密码丢失
Creating wallet: default
Save password to use in the future to unlock this wallet.
Without password imported keys will not be retrievable.
```

```
"PW5Kewn9L76X8Fpd.....t42S9XCw2"
# 打开钱包
cleos wallet open
# 解锁钱包
cleos wallet unlock
// 这里系统将提示输入密码，粘贴上方的创建钱包时提供的密码
# 导入开发密钥
cleos wallet import  -private-key “5KQwrPbwdL6PhXujxW37FSSQZ1JiwsST4cqQzDeyXtP79zkvFD3”
// 这时系统会将私钥对应的公钥显示在终端上

// 使用 cleos 创建公私钥对
cleos create key --to-console
注：当需要公私钥对时可以使用此命令创建公私钥对并输出到终端上
```

# 4. 启动 nodeos 和 keosd

### 启动 nodeos:

```
nodeos -e -p eosio \
--plugin eosio::producer_plugin \
--plugin eosio::producer_api_plugin \
--plugin eosio::chain_api_plugin \
--plugin eosio::http_plugin \
--plugin eosio::history_plugin \
--plugin eosio::history_api_plugin \
--filter-on="*" \
--access-control-allow-origin='*' \
--contracts-console \
--http-validate-host=false \
--verbose-http-errors >> nodeos.log 2>&1 &
```

### 启动 keosd:

```
keosd &
// 此时终端会有如下输出:
info  2018-11-26T06:54:24.789 thread-0  wallet_plugin.cpp:42          plugin_initialize    ]
initializing wallet plugin
info  2018-11-26T06:54:24.795 thread-0  http_plugin.cpp:554          add_handler          ]
add api url: /v1/keosd/stop
info  2018-11-26T06:54:24.796 thread-0  wallet_api_plugin.cpp:73     plugin_startup       ]
starting wallet_api_plugin
```

```
info  2018-11-26T06:54:24.796 thread-0  http_plugin.cpp:554           add_handler          ]
add api url: /v1/wallet/create
info  2018-11-26T06:54:24.796 thread-0  http_plugin.cpp:554           add_handler          ]
add api url: /v1/wallet/create_key
info  2018-11-26T06:54:24.796 thread-0  http_plugin.cpp:554           add_handler          ]
add api url: /v1/wallet/get_public_keys
```

**当需要调用 EOS 钱包接口时需要启动 keosd，keosd 会提供对应 EOS 钱包相关的接口功能。**

# 开发智能合约

下面我们将一步步实现一个名为 hello 的智能合约，来看看如何开发智能合约。

## 编写测试合约

```
# 创建一个合约工作目录
mkdir /contracts/hello -p
cd /contracts/hello
# 新建一个源文件 hello.cpp
vi hello.cpp
```

```
# include <eosio/eosio.hpp> //eosio.hpp 包含了写智能合约必须要用到的 class

using namespace eosio; //使用 namespace eosio;, eosio::print("foo") 就可以写成 print("foo"), 让我们能写出更简洁的代码

class [[eosio::contract]] hello : public contract { //必须继承 contract
public:
using contract::contract; //利用 using 能使代码更简洁

[[eosio::action]]
void hi( name user ) {
print( "Hello, ", user);
}
};
```

源文件参数说明:

- eosio.hpp 头文件：包含了写智能合约必须要用到的 class

- namespace eosio 语句： eosio::print("foo") 就可以写成 print("foo")，让我们能写出更简洁的代码
- 定义一个 c++11 类 hello，它继承了 eosio::contract 类，包含在 eosio.hpp 头文件里
- 利用 using 能让我们把代码写的更简洁
- 合约需要做一些事情，这里写了一个 action，名字叫 hi，还有一个 name 类型的参数。hi 做的事情是输出传入的参数。EOSIO 有很多的自定义的内建类型，name 类型属于内建的类型
- 在 eosio.cdt 中，如果没有一个属性，ABI 生成器就无法知道 hi() action 的情况。我们加一个 c++11 风格的属性[[eosio::action]] 语句到 hi() action 上方，这样 CDT 编译器就可以自动产生更多可信赖的信息到 abi 文件中

## 编译

准备好源文件，执行下面命令对源文件进行编译。编译后生成 wasm 格式的文件和 abi 文件，用来部署到区块链上。wasm 是程序的可執行文件，abi 是文本文件，描述 wasm 程序的接口和数据结构。

```
eosio-cpp hello.cpp -o hello.wasm -abigen
```

## 部署

所谓部署合约，就是将合约部署到一个账户上。

使用 cleos 命令来创建一个账户，其中 YOUR_PUBLIC_KEY 是 给账户指定的公钥。

```
cleos create account eosio hello YOUR_PUBLIC_KEY -p eosio@active
```

然后，通过 cleos set contract 命令部署编译好的 wasm 文件到区块链上，其中-p CONTRACTS_DIR/hello 指定存放 wasm 合约的目录。

```
cleos set contract hello CONTRACTS_DIR/hello -p hello@active
```

这样合约就被部署到了区块链上。之后就可以调用区块链上 hello 合约的 action 接口。

# 执行 action

通过 cleos push action 命令可以调用 hello 合约的 action hi，正如期望的，在 console 中打印出“hello，bob”。

```
$cleos push action hello hi '["bob"]' -p bob@active
executed transaction: 4c10c1426c16b1656e802f3302677594731b380b18a44851d38e8b5275072857 244 bytes 1000 cycles
# hello.code <= hello.code::hi {"user":"bob"}
>> Hello, bob
```

这里，alice 是是一个授权者， ‘bob’是参数。下面我们修改一下合约代码，增加 require_auth 语句。

```
void hi( name user ) {
require_auth( user );
print( "Hello, ", name{user} );
}
```

此时，在 hi()中将会检查执行授权者账户是否与参数里指定的账户一样，一样则执行后面的，不一样则发生异常。

重新编译

```
# eosio-cpp -abigen -o hello.wasm hello.cpp
```

更新合约

```
# cleos set contract hello CONTRACTS_DIR/hello -p hello@active
```

执行下面 action，如预期的那样，由于授权者账户 alice 和参数指定的账户 bob 不一样，require_auth 停止了事务并抛出了一个错误。

```
# cleos push action hello hi '["bob"]' -p alice@active
Error 3090004: Missing required authority
Ensure that you have the related authority inside your transaction!;
If you are currently using 'cleos push action' command, try to add the relevant authority using -p option.
```

再试下面的命令，提供同样的账户名，这次成功了。

```
# cleos push action hello hi '["alice"]' -p alice@active
executed transaction: 235bd766c2097f4a698cfb948eb2e709532df8d18458b92c9c6aae74ed8e4518 244 bytes 1000 cycles
# hello <= hello::hi {"user":"alice"}
>> Hello, alice
```

# 智能合约的命名规则

## Action 名

- 长度要小于等于 12
- 字符范围 12345abcdefghijklmnopqrstuvwxyz

## Table 名

- 最多只能包含 12 个字母字符
- 字符范围 12345abcdefghijklmnopqrstuvwxyz.

## 表 Stuct 体名与字段名

结构体名长度都要小于等于 13

下面的字段名字都是合法的。

```
TABLE CERTIFICATE
{
uint64_t id;
uint64_t validTime;
string cert;
string node_name;
};
```

## 类名

最多只能包含 12 个字母字符，小写，类名要和文件名一样

## Scope 名

- <=13 个字符
- 可用字符：[a-z1-5.]
- '.'不能在最后

## Action 参数的数据类型

action 处理函数（handler）支持下列数据类型:

```
unsigned __int128 uint128_t
unsigned long long uint64_t
unsigned long uint32_t
unsigned short uint16_t
unsigned char uint8_t
long long int64_t
long int32_t
short int16_t
char int8_t
std::vector<type>
std::string
name
```

数组可以用 std::vector<type>替代，bool 类型可以用 int8_t 替代

# 多索引表

EOSIO Multi-Index API 为 EOSIO 数据库提供 C ++接口。表对象的创建和修改都需要花费 RAM，删除则会返回 RAM 到账户中。

详细的类的说明参见:

https://eosio.github.io/eosio.cdt/latest/classeosio_1_1multi__index

## 主键

创建一个 EOSIO Multi-Index 表必需要一个 uint64_t 主键。为了使表能够检索主键，存储在表中的对象需要具有名为 primary_key() 的 const 成员函数，该函数返回 uint64_t 。

## 索引

EOSIO Multi-Index 表还支持最多 16 个二级索引.索引支持的格式如下：

- uint64_t
- uint128_t
- double
- long double
- eosio::checksum256

## 例子说明

通过一个例子很容易了解怎样写一个操作表的程序。下面是一个保存联系人信息的合约。通过注释可以容易了解到合约的结构和多索引表的用法。

- addressbook.hpp

```
#include <eosiolib/eosio.hpp>
using namespace eosio;
using std::string;

CONTRACT addressbook : public eosio::contract //定义一个合约类
{
public:
using contract::contract;

private:
TABLE addressbook_t //表记录的结构，使用 TABLE
{
uint64_t id;
string name;
```

```
uint8_t age;
uint64_t phonenumber;
string address;

uint64_t primary_key() const {return id;} //主键函数
uint64_t by_phonenumber() const {return phonenumber;} //索引函数
};


public:

//定义多索引表类型
typedef eosio::multi_index<
name("addressbooks"), //表名
addressbook_t, //前面定义的表记录的结构
//指定索引
indexed_by<
name("phonenumber"), //索引名
const_mem_fun<
addressbook_t, //表记录的结构
uint64_t, //字段的类型
&addressbook_t::by_phonenumber //索引函数
>
>
>addressbooks_t; //表类型

//几个 action 函数
ACTION add( name account, string name, uint8_t age, uint64_t
phonenumber, string address );
ACTION remove(name account, uint64_t id);
ACTION update(name account, uint64_t id, string name, uint8_t age,
uint64_t phonenumber, string address);
};
```

- addressbook.cpp

```
#include <eosiolib/asset.hpp>
#include "addressbook.hpp"
using namespace eosio;

//action 函数 add
ACTION addressbook::add( name account, string name, uint8_t age,
uint64_t phonenumber, string address ) {
require_auth( account );
addressbooks_t addressbooks(_self, account.value); //定义一个多索引表变量
addressbooks.emplace(account, //插入一个表记录
```

```
[&]( auto& row ){
row.id = addressbooks.available_primary_key();
row.name = name;
row.age = age;
row.phonenumber = phonenumber;
row.address = address;
}
);
}

ACTION addressbook::remove(name account, uint64_t id) {
require_auth(account);
addressbooks_t addressbooks(_self, account.value); //定义一个多索引表变量
auto itr = addressbooks.find(id); //根据主键查询记录
addressbooks.erase(itr); //删除记录
}

ACTION addressbook::update(name account, uint64_t id, string name,
uint8_t age, uint64_t phonenumber, string address) {
require_auth(account);
addressbooks_t addressbooks(_self, account.value);
auto itr = addressbooks.find(id); //根据主键查询记录
addressbooks.modify( itr, //修改指定的记录
account,
[&]( auto& row ){
row.name = name;
row.age = age;
row.phonenumber = phonenumber;
row.address = address;
}
);
}
```

## 测试

假设在区块链网络上部署合约给账号 ab，可以通过 cleos 命令行做个简单的测试

- 部署合约

```
cleos set contract ab ./build addressbook.wasm addressbook.abi
```

- 添加一个联系人

action 的参数以 json 的格式发给合约，json 的字段与 action 函数的参数意义对应

```
cleos push action ab add '
{
"account": "alice",
"name": "xiaoli",
"age": "22",
"phonenumber": 13911112222,
"address": "北京市朝阳区"
}' -p alice@active
```

- 查询

查询结果也是以 json 的形式返回客户端

cleos get table ab alice addressbooks

```
{
"rows": [{
"id": 0,
"account": "alice",
"name": "xiaoli",
"age": "22",
"phonenumber": 13911112222,
"address": "北京市朝阳区"
}
],
"more": false
}
```

# ABI 文件

EOS 区块链网络都部署了 eosio.token 基础的合约，下面以 eosio.token 合约为例，来窥探 ABI 文件。

可以使用 eosio.cdt 提供的 eosio-cpp 实用程序生成 ABI 文件，但是，有几种情况可能导致 ABI 的生成出现故障或完全失败，高级 C++模式可以将其提升，自定义类型有时会导致 ABI 生成的问题，因此，你必须了解 ABI 文件的工作原理，以便在必要时进行调试和修复。

# 什么是 ABI

应用程序二进制接口（ABI）是一个基于 JSON 的描述，介绍如何在 JSON 和二进制表示之间转换用户操作，ABI 还描述了如何将数据库状态(tables)转换为 JSON 或从 JSON 转换，通过 ABI 描述合约后，开发人员和用户将能够通过 JSON 无缝地与你的合约进行交互。简而言之，一旦我们用 ABI 文件对智能合约进行描述之后，开发者和用户们就能轻而易举的用 JSON 文件与智能合约交互了。

值得注意的是，ABI 文件只是一个交互说明，而不是强制执行，所以，可以向智能合约传递非严格按照 ABI 文件说明的数据。

**说明:**

执行交易时可以绕过 ABI，传递给合约的消息和操作不必符合 ABI，ABI 是一个指南，而不是看门人。

# 创建 ABI 文件

从空的 ABI 开始，将其命名为 eosio.token.abi

```
{
"version": "eosio::abi/1.0",
"types": [],
"structs": [],
"actions": [],
"tables": [],
"ricardian_clauses": [],
```

```
"abi_extensions": [],
"___comment" : ""
}
```

## Types

内置类型

EOSIO 实现了许多自定义内置类型，不需要在 ABI 文件中描述内置类型，如果你想熟悉明确的内置类型，可以这样定义。

```
{
"new_type_name": "name",
"type": "name"
}
```

## structs

暴露于 ABI 的结构也需要描述，通过查看

eosio.token.hpp(|https://github.com/EOSIO/eosio.contracts/blob/master/contracts/eosio.token/include/eosio.token/eosio.token.hpp

可以快速确定公共操作使用了哪些结构，这对下一步尤为重要。

JSON 中的结构对象定义如下所示：

```
{
"name": "create", //The name
"base": "", //Inheritance, parent struct
"fields": [] //Array of field objects describing the struct's fields.
}
```

fields

```
{
"name":"", // The field's name
"type":"" // The field's type
```

```
}
```

在 eosio.token 合约中，有许多结构需要定义，请注意，并非所有结构都是显式定义的，有些结构对应于 action 的参数，以下是需要对 eosio.token 合约进行 ABI 描述的结构列表。

## 隐式 structs

以下结构是隐式的，因为结构从未在合约中显式定义，查看 create 操作，你将找到两个参数，类型为 name 的 issuer 和 asset 类型的 maximum_supply：

- create

```
{
"name": "create",
"base": "",
"fields": [
{
"name":"issuer",
"type":"name"
},
{
"name":"maximum_supply",
"type":"asset"
}
]
}
```

## 显式 structs

这些结构是显式定义的，因为它们是实例化多索引表的模式，定义方法与如上所示的隐式结构没有什么不同。例如和账户 token 资产有关的结构。

- account

```
{
"name": "account",
"base": "",
"fields": [
```

```
{
"name":"balance",
"type":"asset"
}
]
}
```

## Actions

action 的 JSON 对象定义类似如下所示：

```
{
"name": "transfer", //The name of the action as defined in the contract
"type": "transfer", //The name of the implicit struct as described in the ABI
"ricardian_contract": "" //An optional ricardian clause to associate to this action describing its intended functionality.
}
```

通过聚合 eosio.token 合约头文件中描述的所有公共函数来描述 eosio.token 合约的操作。然后根据之前描述的结构描述每个操作的类型，在大多数情况下，action 名称和结构名称将相等，但不必相等。

## Tables

描述表，这是表的 JSON 对象定义：

```
{
"name": "", //The name of the table, determined during instantiation.
"type": "", //The table's corresponding struct
"index_type": "", //The type of primary index of this table
"key_names" : [], //An array of key names, length must equal length of key_types member
"key_types" : [] //An array of key types that correspond to key names array member, length of array must equal length of key names array.
}
```

eosio.token 合约实例化两个表，accounts 和 stats。

accounts 表是一个 i64 索引，基于 account struct，有一个 uint64 作为它的主键。

以下是如何在 ABI 中描述 accounts 表。

```
{
```

```
"name": "accounts",
"type": "account", // Corresponds to previously defined struct
"index_type": "i64",
"key_names" : ["primary_key"],
"key_types" : ["uint64"]
}
```

stat 表是一个 i64 索引，基于 currenct_stats struct，有一个 uint64 作为它的主键。以下是如何在 ABI 中描述 stat 表。

```
{
"name": "stat",
"type": "currency_stats",
"index_type": "i64",
"key_names" : ["primary_key"],
"key_types" : ["uint64"]
}
```

你会注意到上面的表格具有相同的“key name”，将键命名为相似的名称是象征性的，因为它可能暗示一种主观关系，与此实现一样，这意味着可以使用任何给定的值来查询不同的表。

## ABI 示例

最后，来看一下完整的 ABI 是什么样子。执行下面命令可以返回一个主网的 eosio.token 合约的 ABI。

```
cleos -u https://nodes.get-scatter.com:443 get abi eosio.token
```

返回 json 格式的 ABI：

```
{
"version": "eosio::abi/1.0",
"types": [
{
"new_type_name": "name",
"type": "name"
}
],
"structs": [
{
```

```
"name": "create",
        "base": "",
        "fields": [
                {
                "name":"issuer",
                "type":"name"
                },
                {
                "name":"maximum_supply",
                "type":"asset"
                }
]
},
{
"name": "issue",
        "base": "",
        "fields": [
        {
        "name":"to",
        "type":"name"
        },
        {
        "name":"quantity",
        "type":"asset"
        },
        {
        "name":"memo",
        "type":"string"
        }
]
},
{
"name": "retire",
        "base": "",
        "fields": [
        {
        "name":"quantity",
        "type":"asset"
        },
        {
```

```
        "name":"memo",
        "type":"string"
        }
]
},
{
"name": "close",
        "base": "",
        "fields": [
        {
        "name":"owner",
        "type":"name"
        },
        {
        "name":"symbol",
        "type":"symbol"
        }
]
},
{
"name": "transfer",
        "base": "",
        "fields": [
        {
        "name":"from",
        "type":"name"
        },
        {
        "name":"to",
        "type":"name"
        },
        {
        "name":"quantity",
        "type":"asset"
        },
        {
        "name":"memo",
        "type":"string"
        }
]
```

```
},
{
"name": "account",
        "base": "",
        "fields": [
        {
        "name":"balance",
        "type":"asset"
        }
]
},
{
"name": "currency_stats",
        "base": "",
        "fields": [
        {
        "name":"supply",
        "type":"asset"
        },
        {
        "name":"max_supply",
        "type":"asset"
        },
        {
        "name":"issuer",
        "type":"name"
        }
]
}
],
"actions": [
{
"name": "transfer",
"type": "transfer",
"ricardian_contract": ""
},
{
"name": "issue",
"type": "issue",
"ricardian_contract": ""
```

```
},
{
"name": "retire",
"type": "retire",
"ricardian_contract": ""
},
{
"name": "create",
"type": "create",
"ricardian_contract": ""
},
{
"name": "close",
"type": "close",
"ricardian_contract": ""
}
],
"tables": [
{
"name": "accounts",
"type": "account",
"index_type": "i64",
"key_names" : ["currency"],
"key_types" : ["uint64"]
},
{
"name": "stat",
"type": "currency_stats",
"index_type": "i64",
"key_names" : ["currency"],
"key_types" : ["uint64"]
}
],
"ricardian_clauses": [],
"abi_extensions": []
}
```

## 自动生产 ABI

其实 ABI 不需要自己手动生成，在编译 cpp 的时候，可以指定--abigen 选项，编译器会根据 cpp，hpp 代码中的描述自动生成。例如，执行下面命令:

```
eosio-cpp hello.cpp -o hello.wasm -abigen
```

将生成两个文件：hello.wasm 和 hello.abi。 hello.abi 就是自动生成的 ABI 文件。 代码中的[[eosio::action]]就是告诉编译器，这个函数属于 action，可以放到 ABI 文件中。

## 维护

每次更改结构、添加表、添加操作或向操作添加参数、使用新类型时，你都需要记住更新 ABI 文件，在许多情况下，更新 ABI 文件失败不会产生任何错误。

## 故障排除

- 表不返回任何行

检查你的表是否在 ABI 文件中准确描述，例如，如果使用 cleos 在具有格式错误的 ABI 定义的合约上添加表，然后从该表中获取行，则将收到空结果。当合约未能在其 ABI 文件中正确描述其表时，cleos 在添加行或读取行时不会产生错误。

# 总结

到此，我们已经经历了一个完整合约开发的流程：搭建一个单节点的区块链平台，写一个 hello world 程序，编译程序成 wasm 格式文件，将 wasm 部署到了区块链上，最后通过 cleos 调用合约的 action。了解到了整个的开发流程，

源代码的一般结构，ABI 文件是怎么样描述一 个合约的。拥有了这些知识，我们就可以进一步的开发自己想要的合约。在这里我们学的仅仅是最基本的，要开发一个现实的合约需要用到更多的知识，比如持久化、权限验证，这些知识在官方的网站上有更加具体的讲述，作为一个合约开发者必须要充分利用官方网站提供的文档。随着区块链的发展，各种版本 EOS 系统，CDT 也是变化的，所以我们需要学会充分利用官方的文档。下面是有关的一些 eosio 官方资料。

- https://eos.io // EOS 官网
- https://developers.eos.io // EOS 官方开发人员的文档
- https://github.com/EOSIO/eosio.cdt //合约开发工具包
- https://github.com/EOSIO/eosio.contracts //官方提供的区块链用的系统智能合约
- https://github.com/EOSIO/eos/tree/v1.5.2/contracts //合约源代码学习