Gavin Zheng
Longxiang Gao
Liqun Huang
Jian Guan

Ethereum Smart Contract Development in Solidity

Springer

Ethereum Smart Contract Development in Solidity

Gavin Zheng • Longxiang Gao • Liqun Huang •
Jian Guan

Ethereum Smart Contract Development in Solidity

Springer

Gavin Zheng
Nenglian Technology Co. Ltd.
Beijing, China

Longxiang Gao
School of Information Technology
Deakin University
Burwood, VIC, Australia

Liqun Huang
Huazhong University of Science and Technology
Wuhan, Hubei, China

Jian Guan
Muhua Technology Co. Ltd.
Beijing, China

ISBN 978-981-15-6217-4 ISBN 978-981-15-6218-1 (eBook)
https://doi.org/10.1007/978-981-15-6218-1

Preface

Smart contract is one of the cornerstones of blockchain technology. Among all the smart contract programming languages in market (such as Viper, Bamboo, etc.), Solidity running on Ethereum Virtual Machine (EVM) is the most popular one in terms of number of users, developer community, scope of use, number of contracts in use, and the public recognition. This book introduces the Solidity programming language from scratch and explains the core features of Solidity in detail.

About the Book Structure

The book is organized in an orderly way as below:

Part I: Preliminary
Chapter 1: Concepts and terms of Ethereum
Chapter 2: Configuration and installation of Solidity development environment
Part II: Solidity Basics
Chapter 3: Basics of Solidity: keywords, statements, modifiers, etc.
Chapter 4: Popular Ethereum Request for Comments (ERC) protocols
Part III: Solidity Advanced Topics
Chapter 5: ABI specification and coding
Chapter 6: Advanced topics of Solidity programming: design pattern, GAS, assembly
Chapter 7: Upgradeable smart contract design and implementation
Chapter 8: Security of Solidity programming and best practice
Part IV: Application
Chapter 9: Decentralized Application (DApp) programming technique
Chapter 10: Testing and debugging
Part V: Prospect
Chapter 11: Primer of Web Assembly programming which is believed to be the future

In Part I, Chap. 1 introduces all the basic concepts and terms of Ethereum used in the book for understanding of subsequent content; you can skip it if you already know them well. And Chap. 2 tells you how to set up Solidity development and testing environment. After all the preparation work being done, Part II guides you through Solidity language details and coding simple contracts. To develop complex contracts, Part III collects all the information you need: interface, design pattern, GAS, assembly, upgrade, security, and best practices. In Part IV, the authors shed some light on developing DApp which showcases how to interact with Solidity contract. At last, the book also explores WASM a little bit, which is thought to be a future star in smart contract programming.

Beijing, China Gavin Zheng
Burwood, VIC, Australia Longxiang Gao
Wuhan, Hubei, China Liqun Huang
Beijing, China Jian Guan
April, 2020

Intended Audience

The book is intended for readers with prior experience of using at least one object-oriented programming language (e.g., C++, Java, etc.). If you have a solid understanding of object-oriented concepts, such as inheritance, polymorphism, etc., and you desire to jump into the blockchain industry and ramp up on smart contract development on Ethereum platform, this book is the best fit for you. Stepping from this book, the readers should dive deep into Ethereum Virtual Machine (EVM) and Web Assembly (WASM) for continuing study.

Acknowledgement

I would like to thank the co-authors: Dr. Jian Guan—CTO of xuetangx.com, Dr. Longxiang Gao at Deakin University, Geelong, VIC, Australia, and Associate Professor Liqun Huang at Huazhong University of Science and Technology (HUST), Wuhan, PRC, for their valuable efforts and contributions in writing this book. Besides, I would like to thank the responsible editor of this book—Dr. Wei Zhu who organizes to turn the initiative into reality.

Contents

Part I
Preliminary

Chapter 1
Basic Concepts

Ethereum is a public, open-source platform based on blockchain technology. You can view it as a world computer built on top of peer-to-peer (P2P) network. Trusted and decentralized application can be run on top of Ethereum with no threat of centralized management and single-point-of-failure (SPOF) problem. And as there is only one such machine in the world, the computation resource (such as CPU, memory, etc.) is limited and strained under the pressure of huge scale of user base and DApps. So, it is understandable that using Ethereum world machine will cost money in the form of crypto currency.

1.1 Ethereum

Here is official definition of Ethereum from Ethereum.org:

> Ethereum is a decentralized platform that runs smart contracts: applications that run exactly as programmed without any possibility of downtime, censorship, fraud or third-party interference. These apps run on a custom built blockchain, an enormously powerful shared global infrastructure that can move value around and represent the ownership of property.

Bitcoin is the first well-known application of blockchain technology. However, it is still a currency. Comparatively, Ethereum is an open-source and distributed platform on the foundation of blockchain technology and it brings full potential of blockchain technology to our attention.

Generally speaking, Ethereum is a state machine based on transaction. The state transformation formula is like below:

$$\sigma' = \Upsilon(\sigma, T)$$

Initial state is σ, transformed state is σ', T means transaction, Υ is transformation function. Let us have a look at the following example:

G. Zheng et al., *Ethereum Smart Contract Development in Solidity*,
https://doi.org/10.1007/978-981-15-6218-1_1

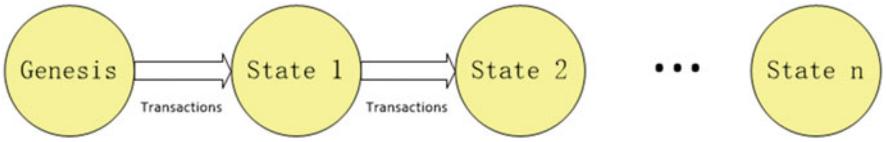


Fig. 1.1 Ethereum state machine

- Initial state is that account balance of Alice is 100 US$ and account balance of Bob is 0.
- Assume that the transaction is that Alice pay 10 US$ to Bob.
- Transformation Function Υ is to transfer 10 US$ from Alice's account to Bob's account
- Transformed state is σ': Account balance of Alice is 90 US$ and Bob has 10 US$ (Fig. 1.1).

Ethereum holds all the features of a public blockchain: Public/private key encryption, cryptography hash function, Merkle tree, and hard/soft fork, etc. Following are technologies and terminologies which we will use in the later chapter.

1.1.1 Asynchronized Cryptography

Async cryptography is the greatest invention of the history of cryptography. Synchronized encryption needs to share key beforehand. However, async cryptography does not need to do that. Like "asynchronized" suggests, encryption key and decryption key are different and they are named public key and private key. Normally, public key is open and available to public, while private key is usually kept private by person. Since public key and private key are separated, it means that public/private key can be used on unsafe channel. The downside is: the whole process is slow because encryption/decryption takes time; and encryption is not as robust as synchronized encryption.

The security of asynchronized encryption usually is supported by mathematics. At the moment, there are several main approaches: Large number prime factorization, discrete algorithm, and elliptic curve. Mainstream algorithms include RSA, ElGamal, and elliptic curve cryto-systems (ECC). Generally, they are used in the scenario like digital signature or key exchange, which is not fit for large-scale data encryption/decryption. As of now, RSA algorithm is considered unsafe to some extent. So, ECC is recommended.

1.1.1.1 Diffie–Hellman Algorithm

Here, we are going to introduce a key exchange algorithm—Diffie–Hellman. First, we will introduce mod operator. It is not difficult. Assuming that we have a modulo A, for any number of B, B mod A is the remainder of B/A

For example: we use modulo A which equals to 11

```
15 mod 11 = 4
(3 + 8) mod 11 = 0
(3^4) mod 11 = 4
```

Now we can start to build key exchange algorithm. Let us think about the scenario which has three persons: A, B, C

1. A and B select their own private number

   For the sake of convenience, we assume that A and B both select a small number: A selects 8, and B selects 9. And in the real world, you should select a big number for security.
2. A and B select two public numbers

   Both A and B share two numbers: one is modulo and the other is radix. In this example, A and B share 11 (modulo) and 2 (radix).
3. A and B build their own public-private number(PPN)

   Now, A and B will use two shared numbers in Step 2 with their own private number to calculate their public-private number (called PPN):

```
PPN = radix ^ private number mod modulo
```

then:

```
A's PPN = 2^8 mod 11 = 3
B's PPN = 2^9 mod 11 = 6
```

As can be seen, the calculation process is irreversible due to modulo operation.

4. A and B mingle the other party's PPN with his own private number.

   The method of mingle is very similar as above, just replace the radix with PPN:

```
A share key = B's PPN^8 mod 11 = 6^8 mod 11 = 4
B share key = A's PPN^9 mod 11 = 3^9 mod 11 = 4
```

The reason why we get the same result is because the power operation satisfies the commutative law.

```
(a^b)^c = (a^c)^b = a^(bc)
```

Now A and B get the final share key (which is 4 in the example above) through mingle process! And A and B can use this share key to encrypt/decrypt. Eavesdropper—C, since C cannot get A or B's private number, even though they know A or B's PPN, C still cannot get the final share key through mingle process.

1.1.1.2 Private/Public Key

Alice holds public key and private key. She can utilize private key to generate a digital signature. Since Alice's public key is available to public, Bob use it to verify that a digital signature is from Alice. When you create an Ethereum address or Bitcoin address, the long hexadecimal string (for example: 0xef. . .59) is a public key while private key may be stored somewhere else—could be on cloud server, or could be on personal device such as mobile phone or personal computer. If you lose the private key of your wallet, that means that you lose all the fund in that wallet permanently. So, it is always a good habit to backup your public key and private key (Figs. 1.2 and 1.3).

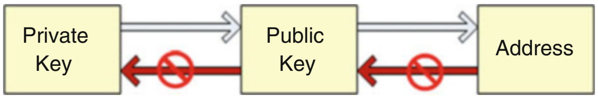


Fig. 1.2 Private key, public key, address relationship

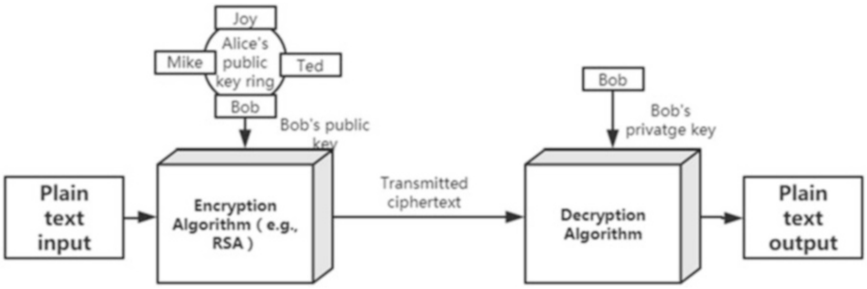


Fig. 1.3 Use priv/pub to encrypt and decrypt message

1.1.1.3 Encryption

The most important application of public/private key is encryption. The picture above depicts the usage:

- Alice has a key ring which contains public key of Bob, Joy, Mike
- Using Bob's public key, Alice encrypts the message sent to Bob
- Alice send encrypted text
- Bob receives encrypted text and uses his own private key to decrypt the text

1.1.1.4 Verifying Signature

Another important application of public/private key system is digital signature. And basic flow is shown as the figure above (Fig. 1.4)

- Using her own private key, Alice encrypts the message (clear text)
- Bob holds Alice's public key
- Once Bob receives encrypted message, using Alice's public key, he can verify whether the message is from Alice or not.

1.1.2 Cryptographic Hash Function

Cryptographical hash is a hash function: it takes message as input and returns a fixed-size string. The result string is called hash value, message digest, digital signature, digital fingerprint, digest or checksum.

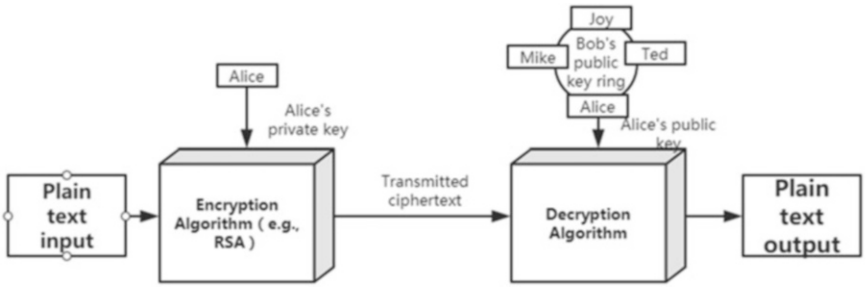


Fig. 1.4 Verify signature https://www.cnblogs.com/moonfans/p/3939335.html

Hash function must have three main attributes as below:

1. Easy to calculate
   Using hash function to calculate the hash value for any input should be easy and fast. The whole calculation process should not take too long. It is unacceptable if hash value calculation takes 1 day or longer.
2. Irreversible
   If hash result is already known, it is very hard to get source text through reverse calculation. For example: you have a movie ticket, but it is very difficult to fake one.
3. Collision Resistant
   Different message must have different hash value. The analog is: A and B both pay 10US$ to buy movie tickets and seat number should not be the same.

1.1.3 Peer-to-Peer Network

Just like BitTorrent, all Ethereum nodes are peers on a distributed network. There is no centralized server. In the future, there might be all kinds of quasi-centralized service for user and developer's convenience. P2P network mainly have three types of topology: Distributed hash table (DHT) tree and network.P2P technology has covered almost all network application areas, for example: distributed scientific computation, file sharing, stream play, voice communication, and online gaming platform. Some famous P2P algorithms are Kademlia, Chord, Gnutella, etc.

1.1.4 Blockchain

Blockchain can be viewed as a global ledger or a simple database which includes all transactions. All information is public available on network and verifiable. Blockchain is a distributed ledger technology, and is the foundation of Bitcoin and Ethereum crypto currency. It provides methods to record and transmit information in a transparent, safe and traceable way. The blockchain technology makes organization transparent, democratic, distributed, highly efficient, safe, and reliable and it provides a method to record and transmit data in a transparent, safe, and traceable way. Blockchain technology might disrupt existing industries in next 5 to 10 years.

- *Decentralized*
  Service or application is deployed to a network and no centralized server has absolute control on data and code. And, one or several server crash will not influence the application or service.
- *Distributed*
  Each server or node in the network connects each other through P2P protocol.
- *Database*
  Database has multiple copies on each node so that user can access at any time in time.

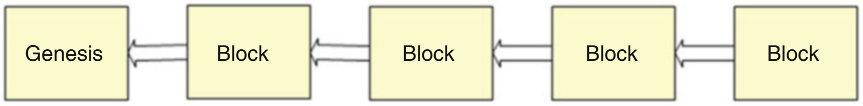


Fig. 1.5 Blockchain

- *Ledger*

  Each node holds an accounting system based on the same public ledger, recording all transaction information in the network. And the ledger is untampered and append-only (Fig. 1.5).

1.1.5 Ethereum Virtual Machine (EVM)

On top of Ethereum Virtual Machine (EVM), user can develop myriad apps. Compared to Bitcoin's script language, EVM provide more powerful and Turing-complete programming language. For every opcode executed on EVM, it will be run on each node in Ethereum network. Turing-completeness mean that computer can solve any mathematics formula under the assumption that algorithm is correct if we have enough time and memory.

1.1.6 Node

Running a node means that user can access on chain data through the node. A full node downloads the whole blockchain while a light node does not download the all data and connect to a full node to download desired data from it. Full node can be a computer running all necessary software which include full distributed ledger and P2P routing software.

1.1.7 Miner

Miner runs a node mining for network which processes the blocks on blockchain. Usually, miner maintains a full node running professional mining software. But not all full nodes are mining nodes. If there is a code upgrade, mining node needs to upgrade its mining software with up-to-date code. This can be done through a soft fork—a backward compatibility implementation. Although it can be done through hard fork, hard forked code is not compatible with old code. You can find miner list of Ethereum at stats.ethdev.com.

1.1.8 Proof of Work (PoW)

Miners compete with each other to challenge a mathematics problem. The first miner who solves the problem will get rewarded and will have the right to pack transactions into newly generated block. Each node will sync with new block automatically and because of the block reward, each miner will eager to win the right to generate next new block. This leads to a consensus network-widely. Please note, Ethereum is scheduled to migrate to Proof of Stake (PoS) consensus.

1.1.9 Decentralized App (DApp)

In Ethereum community, the applications which use smart contract are called decentralized APP. Using DApp, you can add UI and some extra functions to your smart contract, such as, IPFS. DApp can also be run on a centralized server if that server can interact with Ethereum nodes. DApp can connect to Ethereum node to submit transaction and retrieve interested data. Besides a typical user login system, user has a wallet address and data are saved locally, which is different with current web application.

1.1.10 Solidity

Solidity is an advance programming language based on smart contract. It has similar syntax as JavaScript. It supports static types, integration, library, and composite user-defined type. It can be compiled into EVM assembly and therefore can be executed on all Ethereum nodes. There are other smart contract programming languages: Serpent, Vyper, and LLL. Undoubtably, Solidity is the hottest, most popular programming language for smart contract. EVM is a runtime sandbox. So all smart contracts rested on Ethereum is segregated from surrounded environment. As a result, smart contract on EVM cannot access network, file system, or other processes on Ethereum.

Solidity is a static-type checking language. Its compiler could check:

- All functions should be defined
- Object could not be null
- Invalid operator

1.2 Smart Contract

As its name suggests, smart contract is automatic contract. In fact, smart contract is a computer program which runs automatically when specific condition is satisfied. Smart contact is a set of instructions which is developed by Solidity (could be other programming language, but this book only use Solidity). Solidity programming language is based on IFTTT (which is IF-THIS-THEN-THAT logic: execute code if some condition is met). Since smart contract is run on EVM, it cannot access network, file system, or other processes running on EVM. Smart contract can access external data through Oracle if necessary.

Generally, smart contract can be implemented based on two types of system:

1. Virtual Machine (VM): Ethereum
2. Docker: Fabric

All the following content including discussions, code, and diagrams are based on smart contract framework on Ethereum.

1.3 GAS

GAS is used to measure how many steps a transaction will need on EVM. It is straightforward: your transaction is complex, which means it needs more computation resources (such as CPU time and memory), you will need to pay more GAS. All opcodes on EVM will cost GAS and it is unlikely to change in the future. The smallest metric of GAS is wei, and 1 eth = 10^18 wei = 10^9 gwei.

1. Gas Price: Gas Unit Price
2. Gas Limit: Upper bound of Gas which user is willing to pay

Gas Limit × Gas Price = the maximum fee that user is willing to pay

In one transaction, if the maximum fee you specified is not used up, then the fund left will be returned to your account. Once Gas is exhausted, the transaction will stop at wherever it is and an out-of-gas exception will be thrown. All state changes made by the transaction will be reverted.

But, once the fund is used, even though the transaction fails, user will not get refunded since the fund is rewarded to miners. User needs to pay fee for using computation resource and fee consisted of three components:

1. Computation fee
2. Transaction (contract creation or message call) fee
3. Storage (memory, account/contract data storage) fee

If your contract save data to database, all the nodes in Ethereum will do the same, which is very costly. That is why Ethereum will charge for storage and it encourages less storage usage. If the op is to clear a storage item, Ethereum will do this for free, or even get refunded.

Here is a chart about average GAS price—https://etherscan.io/chart/gasprice

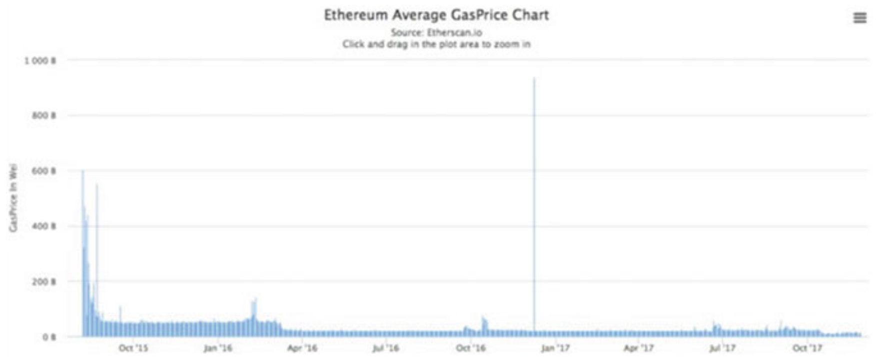


Image Credit: Etherscan

1.3.1 Why GAS?

Why we need GAS? Generally speaking, there are three main reasons: finance, theory, and computation.

From finance point of view, the purpose is to incentivize miners to execute transaction and smart contract by using their own time and resource. Many complex operations need more computation resource; that is to say, they need to pay more GAS. If a user wishes to have their transaction prioritized, he can submit transaction with higher GAS price. In this way, transaction could be processed sooner by miner incentivized by higher transaction fee. As compensation for computation resource which miner invests in, GAS becomes more crucial after consensus migrates to Proof of Stake (POS). In POS era, miner no longer get rewarded by mining blocks and packing transactions, it is more important for miner to process transaction and get paid for expending resources on the blockchain.

The theoretical purpose is to align the incentives of participants on the network. Much of blockchain theory discusses how to mitigate harmful or malicious actors in a trustless environment. GAS partially addresses this issue: Miners are incentivized to work on the network and users are de-incentivized from acting poorly or writing malicious code as they are putting their own ether (in the form of gas) at risk.

From computational point of view, the computational reason behind GAS goes back to an old, foundational aspect of computing theory—the Halting Problem. The Halting Problem is the issue of determining whether an arbitrary program will stop running or if it will run forever just from looking at the description and the input values. In 1936, Alan Turing determined that it is impossible for any machine to solve the Halting Problem. In the EVM, this means a miner is never able to begin processing a transaction and know 100% that the transaction will not go on forever. With GAS—specifically, GAS limit—a finite amount of gas is always attached to a transaction. Even if a miner began processing a transaction that was coded to continue indefinitely—either from a bug or an attack on the network—the gas would eventually run out, the transaction would end, and the miner would still be compensated.

1.3.2 Components of GAS

GAS is divided into three components: GAS cost, GAS price, and GAS limit. In Ethereum, calculation formula of transaction fee is very simple:

```
Transaction Fee(Tx Fee) = Actual GAS Cost * GAS Price
```

For example, if a transaction will need 50 GAS to complete, under the assumption that GAS Price which is set by user is 2 Gwei, then the total transaction fee is 50 * 2 = 100 Gwei.

1.3.2.1 GAS Cost

GAS Cost indicates that each opcode needs how much GAS, which is pre-defined in Ethereum yellow paper. For example, a "addition" opcode need three GAS, regardless of the fluctuation of ether price. The GAS for each opcode will not be changed. That is why GAS is used to estimate the cost of a transaction instead of ether. If ether is used for GAS cost, the price might vary sharply.

1.3.2.2 GAS Price

GAS Price indicates that a unit of GAS equals to how much ether. Commonly, Gwei is used as calculation unit. One Gwei equals to 1 billion Wei, and Gwei is 10 ^ −9 ether to be exact. That is to say, 1 Gwei = 0.000000001 ETH. 1 Wei is the smallest metric unit for ether and is indivisible. If you set GAS price to 20 Gwei, that means you will pay 0.00000002 ether for each step. Apparently, the higher the GAS Price, the more you will pay. There are several web sites like ethgasstation.info which provide the average price of GAS. Sometimes, user may be willing to pay higher price to make their transaction gain priority in getting picked and executed by miners. This is because: GAS specified in transaction will be sent to miner and miner will sort all transaction in their local pool by GAS price. The transaction with higher GAS price will have more chance than those with lower GAS price.

1.3.2.3 Gas Limit

GAS Limit is the upper limit of the GAS usage for a specific transaction. That is the maximum steps required to execute your transaction. GAS limit will be more than what is actually used. Since transaction complexity varies, the GAS actually used is only known after the transaction has been completed. So before you submit the transaction, you need to set an upper limit of GAS usage. If the GAS limit is set too low, a miner will try to complete the transaction until GAS is exhausted. Miner will get rewarded when GAS is exhausted since miner have spent time and power for users' transaction. And on blockchain, the transaction will be set to false. The GAS mechanism is set to protect user and miner: they will not lose fund or power due to the buggy code and malicious attack.

1.4 Ether (ETH)

Ether is the token issued on Ethereum. ETH is the short symbol of ether. And ETH is tradeable crypto currency. In Ethereum, Ether is mainly used to pay transaction fee. Transaction fee equals to GAS cost multiplied by GAS price, and is paid in ETH. User can set GAS price, but bear in mind that if GAS price is too low, it is possible that no miner is willing to pack the transaction.

1.5 Account

Each account has its own address. In the address space, there are two types of account: one is external owned account (EOA) controlled by public/private key. Normally, only people can hold such account which is used to save ETH; the other one is contract account controlled by code. These two types have some difference, but the most important is that only EOA can initiate transactions.

Both types of accounts can *send* and *receive* ethers. Therefore, both have a balance field to keep track of them. Contract accounts, however, can also *store* data. Therefore, they have a storage field, and a code field that contains machine instructions on how to manipulate the stored data. In other words, contract accounts are *smart contracts that live on the blockchain*.

contract account = balance + code + storage

external account = balance + *empty* + *empty*

1.6 Transaction

A transaction is a message sent from an account to another account. We could transfer ETH between accounts through sending transaction to an EOA. If the target account is a contract account, sending transaction will trigger its code execution. Since each transaction will be executed on all nodes in Ethereum, code execution or transaction will be recorded by blockchain.

In this chapter, we introduce all the concepts and terms needed as the foundation for understanding Solidity programming language thereafter. We will start to explore Solidity details in following chapters.

Chapter 2
Preparation

2.1 A Simple Example

Let us have a look at simple smart contract example as below and see how a Solidity contract looks like:

```
pragma solidity 0.4.20;

contract BasicToken {
      uint256 totalSupply_;
      mapping(address => uint256) balances;
      constructor(uint256 _initialSupply) public {
                totalSupply_ = _initialSupply;
                balances[msg.sender] = _initialSupply;
      }
      function totalSupply() public view returns (uint256) {
               return totalSupply_;
      }
      function balanceOf(address _owner) public view returns
(uint256) {
               return balances[_owner];
      }
      function transfer(address _to, uint256 _value) public
returns (bool)
      {
             require(_to != address(0));
             require(_value <= balances[msg.sender]);
             balances[msg.sender] = balances[msg.sender] - _value;
             balances[_to] = balances[_to] + _value;
             return true;
 }
}
```

https://doi.org/10.1007/978-981-15-6218-1_2

This is a typical token contract. Let us not to dive deep into details at the moment and just pay attention to the program structure itself. If you have some object-oriented programming language background, you can find that there is no big difference between Solidity and traditional programming languages such as C++ and Java:

1. There is a pragma keyword, which is a compiler directive that specifies the compiler version to be used for current Solidity file.
2. There is a class and its name is BasicToken; and the "Contract" can be seen as class in C++ language
3. There is a variable declaration: uint256, mapping
4. There is a constructor
5. There is a function
6. There is a parameter and return value
7. There is a "if..else.." statement, which is almost the same as Java or C++

2.2 Tool Preparation

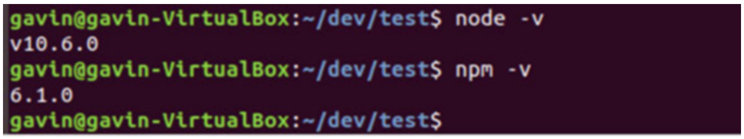
Before we start Ethereum Solidity programming, we need to prepare dev env and tools. Let us install some necessary software and get acquaint with them:

- Programming Language
  Ethereum has multiple versions implemented by different programming languages: such as C++, Python, and Go. While developers have their own preference, in this book, we mainly use Go, Nodejs and Solidity. We also use pyethereum, in order to interactively examine some function of Ethereum.
- Development Tools
  In developing Ethereum application, we may use some debug tools, wallet, and all kinds of plug-ins.
- Block Explorer
  Since data on blockchain is transparent, user can validate their transaction through blockchain explorer. Knowing how to check transaction on chain is a must-have skill.

2.2.1 Development Environment

2.2.1.1 Node Setup

There are multiple ways to install Nodejs: you can either download a package, or download a source code and compile. The easier way is as below:



```
gavin@gavin-VirtualBox:~/dev/test$ node -v
v10.6.0
gavin@gavin-VirtualBox:~/dev/test$ npm -v
6.1.0
gavin@gavin-VirtualBox:~/dev/test$
```

Fig. 2.1 Nodejs and Npm

```
Sudo apt-get install curl
curl -sL https://deb.nodesource.com/setup_10.x | sudo -E bash -
```

You can use following command to install Node.js 10.x and npm.

```
sudo apt-get install -y Nodejs
sudo apt-get install  npm
```

You may also need to install some dev tools for local add-ons:

```
sudo apt-get install gcc g++ make
```

Install Yarn package.

```
curl -sL https://dl.yarnpkg.com/debian/pubkey.gpg | sudo apt-key
add -
echo "deb https://dl.yarnpkg.com/debian/ stable main" | sudo tee /
etc/apt/sources.list.d/yarn.list
sudo apt-get update && sudo apt-get install yarn
```

Confirm the env has been installed properly.

```
node -v
npm -v
```

As shown in the screenshot below, we have installed Nodejs version 10.6.0 and npm package with manager version 6.1.0 (Fig. 2.1).

2.2.1.2 Web3 Installation

Web3 library is the most broadly used Ethereum dev package. We can use npm to install. ^0.20.0 means to install web3 package version 0.20.0.

```
npm install web3@^0.20.0
```

2.2.1.3 Ganache

Ganache is a local Ethereum node for testing purpose. Ganache was called testprc before. We can use the command below to install and start testrpc:

```
npm install -g ethereumjs-testrpc
testrpc
```

Or we can install Ganache directly. From this point on, we will use Ganache as the development and debug tool in this book.

```
sudo npm install -g ganache-cli
```

Let us have a test-run of Ganache:

Figure 2.2 shows that Ganache creates ten accounts with their associated private keys after launch.

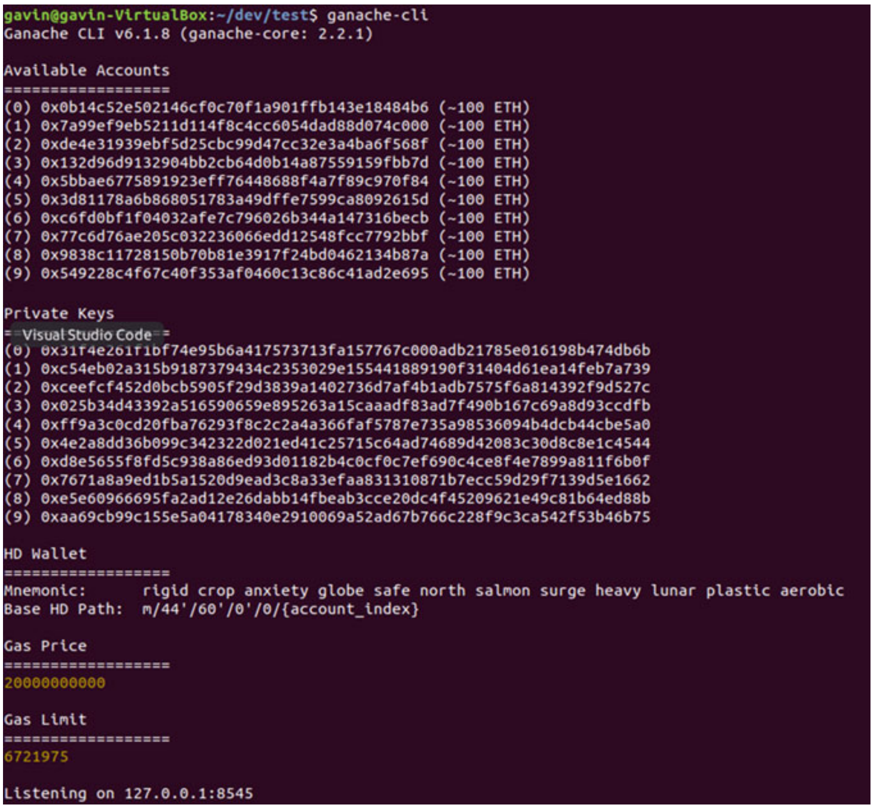
2.2.1.4 Truffle Installation (Fig. 2.3)

Truffle is an outstanding dev environment, testing framework, and Ethereum resource management channel. It makes development on Ethereum much easier. Truffle has following functions:

- Built-in smart contract compiler, linker, deployment, and binary file management.
- Automatic contract testing under the model of agile development.
- Scriptable and extensible deployment and publish framework.
- Network management which can be used to deploy on different network env.
- Package management based on Eth PM&NPM conformed to ERC190 standard.
- Console interacted directly with smart contract.
- Configurable construction workflow and supporting close integration.
- Executing external script in Truffle env.

Truffle project can be found at https://github.com/trufflesuite/truffle. To install Truffle on Ubuntu, please input the following command:

```
sudo npm install –g truffle
```



```
gavin@gavin-VirtualBox:~/dev/test$ ganache-cli
Ganache CLI v6.1.8 (ganache-core: 2.2.1)

Available Accounts
==================
(0) 0x0b14c52e502146cf0c70f1a901ffb143e18484b6 (~100 ETH)
(1) 0x7a99ef9eb5211d114f8c4cc6054dad88d074c000 (~100 ETH)
(2) 0xde4e31939ebf5d25cbc99d47cc32e3a4ba6f568f (~100 ETH)
(3) 0x132d96d9132904bb2cb64d0b14a87559159fbb7d (~100 ETH)
(4) 0x5bbae6775891923eff76448688f4a7f89c970f84 (~100 ETH)
(5) 0x3d81178a6b868051783a49dffe7599ca8092615d (~100 ETH)
(6) 0xc6fd0bf1f04032afe7c796026b344a147316becb (~100 ETH)
(7) 0x77c6d76ae205c032236066edd12548fcc7792bbf (~100 ETH)
(8) 0x9838c11728150b70b81e3917f24bd0462134b87a (~100 ETH)
(9) 0x549228c4f67c40f353af0460c13c86c41ad2e695 (~100 ETH)

Private Keys
= Visual Studio Code =
(0) 0x31f4e261f1bf74e95b6a417573713fa157767c000adb21785e016198b474db6b
(1) 0xc54eb02a315b9187379434c2353029e155441889190f31404d61ea14feb7a739
(2) 0xceefcf452d0bcb5905f29d3839a1402736d7af4b1adb7575f6a814392f9d527c
(3) 0x025b34d43392a516590659e895263a15caaadf83ad7f490b167c69a8d93ccdfb
(4) 0xff9a3c0cd20fba76293f8c2c2a4a366faf5787e735a98536094b4dcb44cbe5a0
(5) 0x4e2a8dd36b099c342322d021ed41c25715c64ad74689d42083c30d8c8e1c4544
(6) 0xd8e5655f8fd5c938a86ed93d01182b4c0cf0c7ef690c4ce8f4e7899a811f6b0f
(7) 0x7671a8a9ed1b5a1520d9ead3c8a33efaa831310871b7ecc59d29f7139d5e1662
(8) 0xe5e60966695fa2ad12e26dabb14fbeab3cce20dc4f45209621e49c81b64ed88b
(9) 0xaa69cb99c155e5a04178340e2910069a52ad67b766c228f9c3ca542f53b46b75

HD Wallet
==================
Mnemonic:      rigid crop anxiety globe safe north salmon surge heavy lunar plastic aerobic
Base HD Path:  m/44'/60'/0'/0/{account_index}

Gas Price
==================
20000000000

Gas Limit
==================
6721975

Listening on 127.0.0.1:8545
```

Fig. 2.2 Ganache startup screen

Fig. 2.3 Truffle logo (https://www.trufflesuite.com/)

Let us check the status of truffle and its version info.

```
truffle v
```

The output is as below (Fig. 2.4).

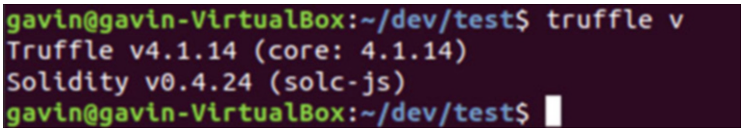
In order to create a new project by using truffle. First, we create a directory named test, and then type the following command:

```
truffle init
```

We check whether the project directory is created or not (Fig. 2.5):

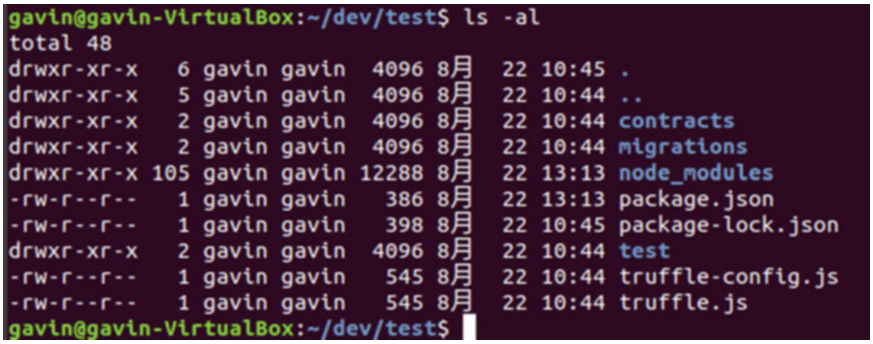
The project's directory is:

```
truffle-experiment/
├── contracts/
       └──Migrations.sol
├── migrations/
       └── 1_initial_migration.js
├── test/
├── truffle.js
└── truffle-config.js
```



```
gavin@gavin-VirtualBox:~/dev/test$ truffle v
Truffle v4.1.14 (core: 4.1.14)
Solidity v0.4.24 (solc-js)
gavin@gavin-VirtualBox:~/dev/test$
```

Fig. 2.4 Truffle version



```
gavin@gavin-VirtualBox:~/dev/test$ ls -al
total 48
drwxr-xr-x   6 gavin gavin  4096 8月  22 10:45 .
drwxr-xr-x   5 gavin gavin  4096 8月  22 10:44 ..
drwxr-xr-x   2 gavin gavin  4096 8月  22 10:44 contracts
drwxr-xr-x   2 gavin gavin  4096 8月  22 10:44 migrations
drwxr-xr-x 105 gavin gavin 12288 8月  22 13:13 node_modules
-rw-r--r--   1 gavin gavin   386 8月  22 13:13 package.json
-rw-r--r--   1 gavin gavin   398 8月  22 10:45 package-lock.json
drwxr-xr-x   2 gavin gavin  4096 8月  22 10:44 test
-rw-r--r--   1 gavin gavin   545 8月  22 10:44 truffle-config.js
-rw-r--r--   1 gavin gavin   545 8月  22 10:44 truffle.js
gavin@gavin-VirtualBox:~/dev/test$
```

Fig. 2.5 Truffle project overview

Smart contract source file is put under contracts directory. JavaScript files used to deploy smart contract are put under Migrations directory. You may also have seen a Migrations contract in the first folder, this is where the history of our migrations is going to be stored on-chain. Test directory is used to save all test files. There are truffle.js and truffle-config.js files under project root directory. These two files contain some parameters required to deploy smart contract. Please check Truffle's official documents for details. Here is a brief summary:

- contracts/: Solidity file directory
- migrations/: deployable script file directory
- test/: testing files reside in this directory
- truffle.js: Truffle configuration file

Truffle uses Mocha as testing framework and Chai for Assertion. Official document can be accessed through: https://mochajs.org/

```
const MyToken = artifacts.require('MyToken')
contract('MyToken', accounts => {
     it('has a total supply and a creator', async function () {
          const owner = accounts[0]
          const myToken = await MyToken.new({ from: owner })
          const creator = await myToken.creator()
          const totalSupply = await myToken.totalSupply()
          assert(creator === owner)
          assert(totalSupply.eq(10000))
 })
})
```

Here are some Truffle commands:

```
truffle migrate
truffle compile
truffle test
```

Truffle let developer start workflow of writing code—compiling—deployment—testing—packing DApp quickly.

2.2.2 *Development Tools*

2.2.2.1 Introduction of Remix

We use two sample contracts to introduce Remix: Callee and Caller. Callee is the contract being called and Caller is the calling contract.

```
pragma solidity ^0.4.24
contract Callee {
    uint[] public values;
    function getValue(uint initial) returns(uint) {
        return initial + 150;
    }
    function storeValue(uint value) {
     values.push(value);
    }
    function getValues() returns(uint) {
       return values.length;
    }
}
```

Source code for Caller contract:

```
pragma solidity ^0.4.24;
contract Caller {
    function someAction(address addr) returns(uint) {
       Callee c = Callee(addr);
       return c.getValue(100);
    }
    function storeAction(address addr) returns(uint) {
       Callee c = Callee(addr);
       c.storeValue(100);
       return c.getValues();
    }
    function someUnsafeAction(address addr) {
       addr.call(bytes4(keccak256("storeValue(uint256)")), 100);
    }
}
contract Callee {
    function getValue(uint initialValue) returns(uint);
    function storeValue(uint value);
    function getValues() returns(uint);
}
```

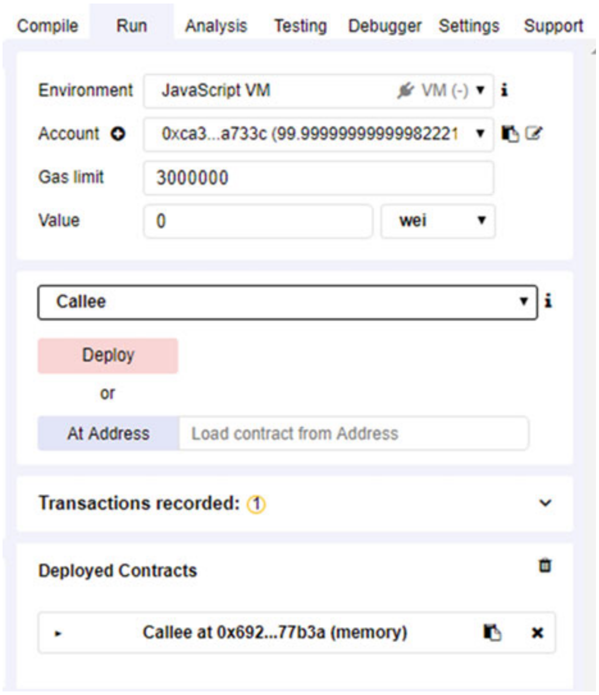
Let us open a browser and navigate it to https://remix.ethereum.org/. We copy and paste the above code to remix. And click "Deploy" on the "Run" tab to deploy Callee contract (Fig. 2.6):

We can use "Copy address" button on the bottom-right to copy contract address to clipboard. In our case, the address is 0x692a70d2e424a56d2c6c27aa97d1a86395877b3a.

Then, we go to create Caller contract: copy Caller source code to remix and click "Deploy" button to deploy Caller contract (Fig. 2.7).

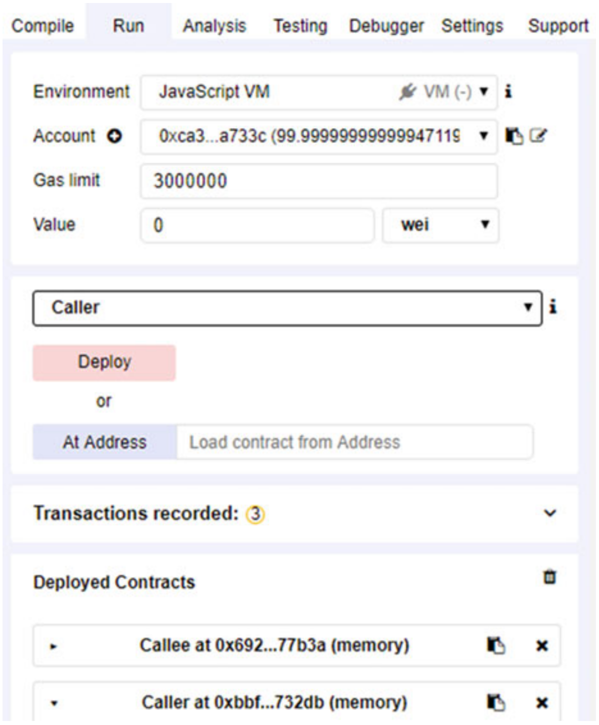
On the right panel, we pass the address of Callee contract as the parameter to someAction method (Fig. 2.8):

From the output pane, we see that the function returns the value 250 as we expect (Fig. 2.9).



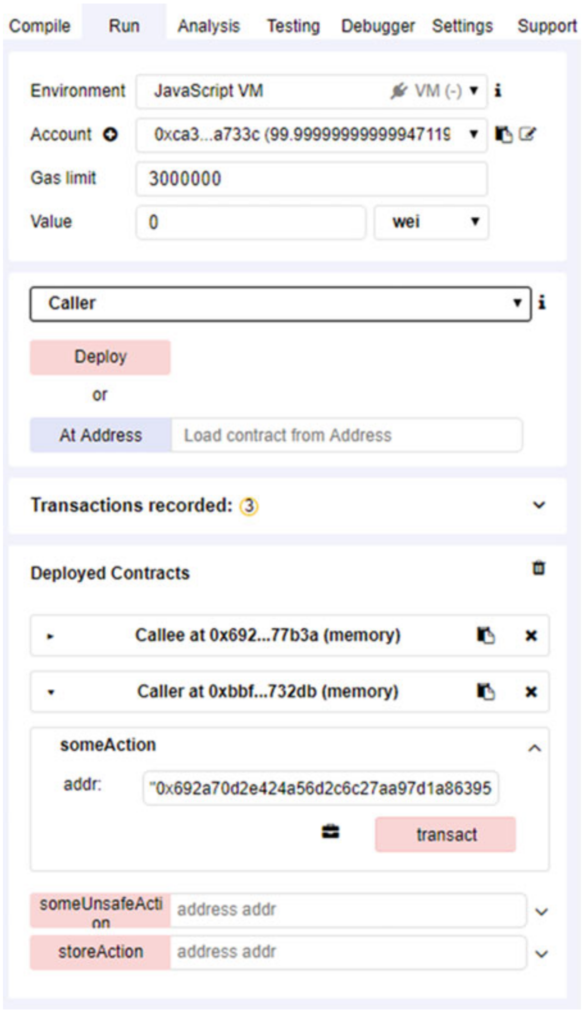
Environment JavaScript VM VM (-)
Account 0xca3...a733c (99.9999999999982221
Gas limit 3000000
Value 0 wei
Callee
Transactions recorded: 1
Callee at 0x692...77b3a (memory)

Fig. 2.6 Deploy Callee contract



Environment JavaScript VM VM (-)
Account 0xca3...a733c (99.999999999994711<
Gas limit 3000000
Value 0 wei
Caller
Transactions recorded: 3
Callee at 0x692...77b3a (memory)
Caller at 0xbbf...732db (memory)

Fig. 2.7 Deploy Caller contract


Compile | Run | Analysis | Testing | Debugger | Settings | Support

Environment: JavaScript VM — VM (-)
Account: 0xca3...a733c (99.999999999994711
Gas limit: 3000000
Value: 0 wei

Caller
Deploy
or
At Address — Load contract from Address

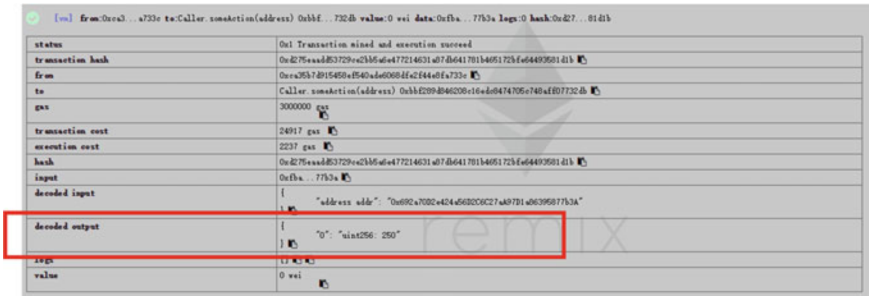
Transactions recorded: 3

Deployed Contracts
Callee at 0x692...77b3a (memory)
Caller at 0xbbf...732db (memory)

someAction
addr: "0x692a70d2e424a56d2c6c27aa97d1a86395
transact

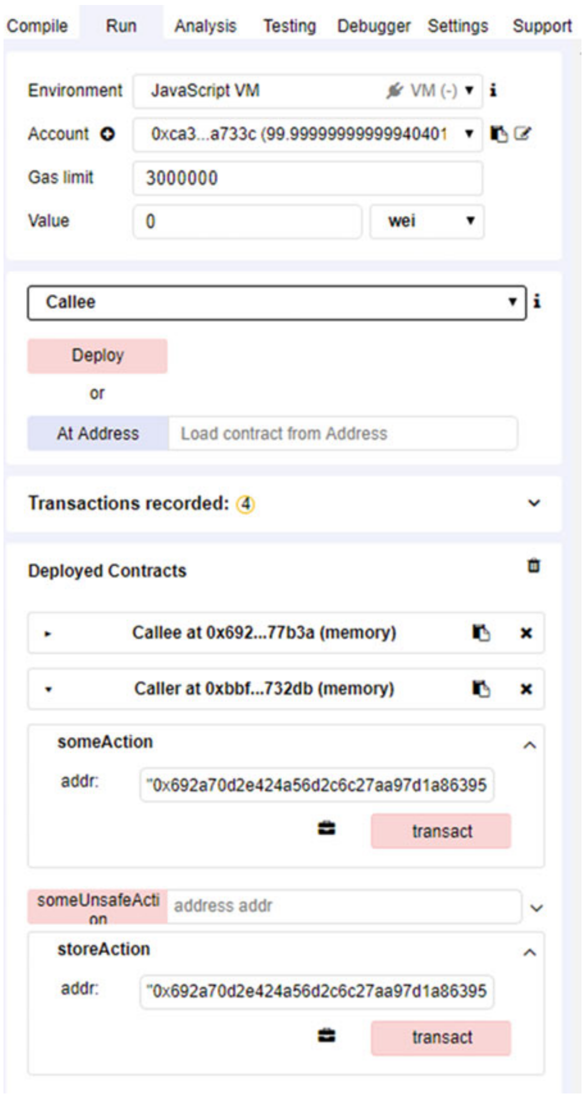
someUnsafeAction — address addr
storeAction — address addr

Fig. 2.8 Pass Callee's address to someAction()


```
[vm] from:0xca3...a733c to:Caller.someAction(address) 0xbbf...732db value:0 wei data:0xfba...77b3a logs:0 hash:0xd27...81d1b
```

status	0x1 Transaction mined and execution succeed
transaction hash	0xd275eaadd53729ce2bb5a6e477214631a07db641781b465172bfe64493581d1b
from	0xca35b7d915458ef540ade6068dfe2f44e8fa733c
to	Caller.someAction(address) 0xbbf289d846208c16edc8474705c748aff07732db
gas	3000000 gas
transaction cost	24917 gas
execution cost	2237 gas
hash	0xd275eaadd53729ce2bb5a6e477214631a07db641781b465172bfe64493581d1b
input	0xfba...77b3a
decoded input	{ "address addr": "0x692a70D2e424a56D2C6C27aA97D1a86395877b3A" }
decoded output	{ "0": "uint256: 250" }
logs	[]
value	0 wei

Fig. 2.9 Console output of someAction()



Environment: JavaScript VM — VM (-)
Account: 0xca3...a733c (99.9999999999994040l
Gas limit: 3000000
Value: 0 wei

Callee

Transactions recorded: 4

Deployed Contracts

Callee at 0x692...77b3a (memory)

Caller at 0xbbf...732db (memory)

someAction
addr: "0x692a70d2e424a56d2c6c27aa97d1a86395

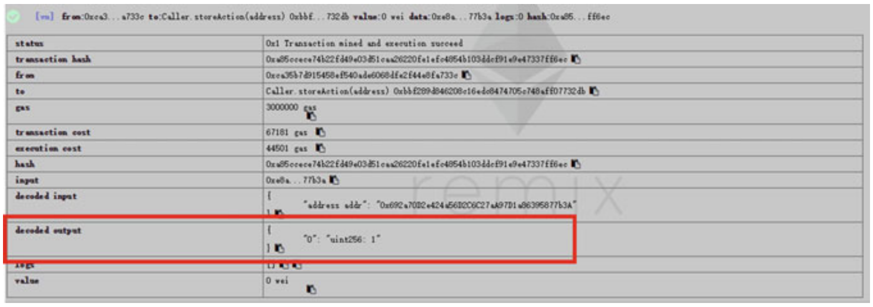
storeAction
addr: "0x692a70d2e424a56d2c6c27aa97d1a86395

Fig. 2.10 Test someUnsafeaction and storeAction()

In the same way, we can play with someUnsafeAction and storeAction function (Fig. 2.10):

storeAction function will return the current value held in Callee contract (in this example, it returns 1) (Fig. 2.11).

Please select appropriate compiler version to run the example, otherwise there will be some compilation errors. Here is just a simple introduction to remix. You can go to Sect. 10.4 for more details (Fig. 2.12).



```
[vm] from:0xca3...a733c to:Caller.storeAction(address) 0xbbf...732db value:0 wei data:0xe8a...77b3a logs:0 hash:0xa85...ff6ec
```

status	0x1 Transaction mined and execution succeed
transaction hash	0xa85ccece74b22fd49e03d51caa26220fe1efc4854b103ddcf91e9e47337ff6ec
from	0xca35b7d915458ef540ade6068dfe2f44e8fa733c
to	Caller.storeAction(address) 0xbbf289d846208c16edc8474705c748aff07732db
gas	3000000 gas
transaction cost	67181 gas
execution cost	44501 gas
hash	0xa85ccece74b22fd49e03d51caa26220fe1efc4854b103ddcf91e9e47337ff6ec
input	0xe8a...77b3a
decoded input	{ "address addr": "0x692a70D2e424a56D2C6C27aA97D1a86395877b3A" }
decoded output	{ "0": "uint256: 1" }
logs	[]
value	0 wei

Fig. 2.11 Console output of someUnsafeaction and storeAction()

2.2.2.2 Introduction of Infura

Ganache can only be used to test locally. If we want to deploy or test smart contract in production env or other public testing envs, we must connect to a full node. There are two options to do so:

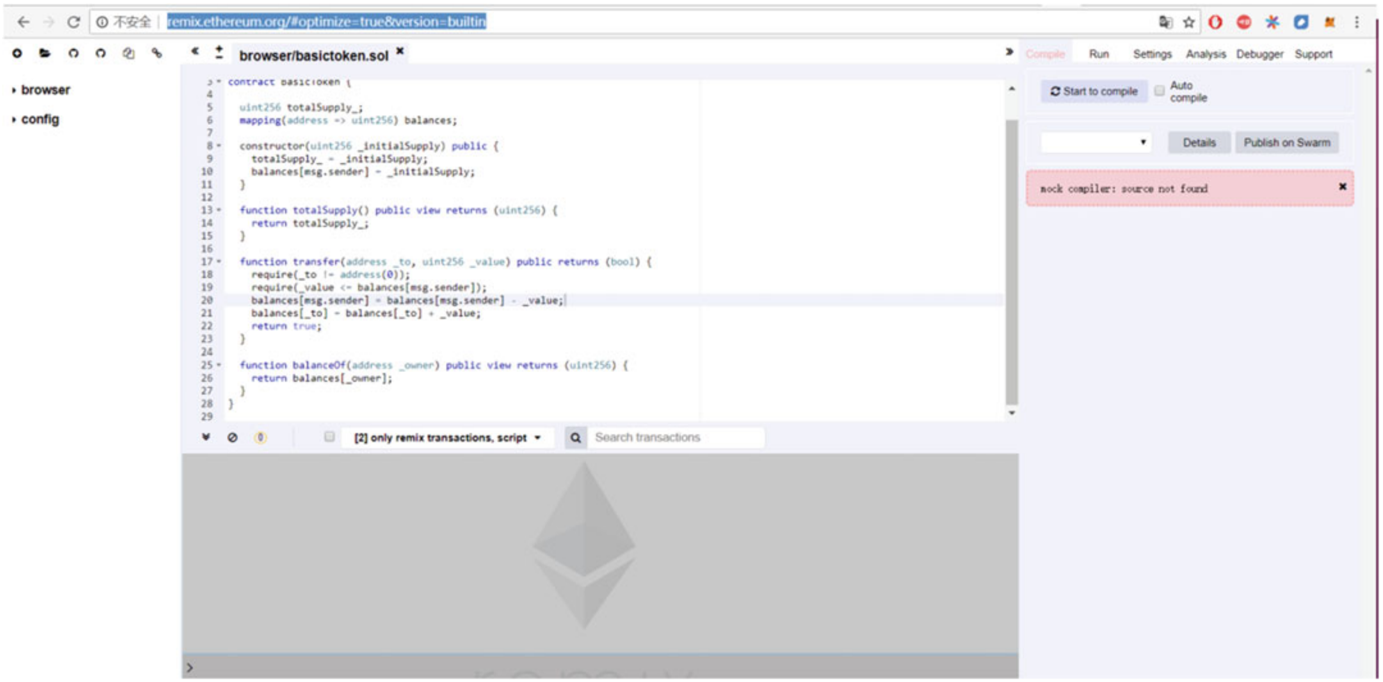
1. Running a full node on local
   This solution requires that developers need to configure, compile, and launch an Ethereum full node, where developers need to sync all the blocks. It is apparently complicated. To date, Ethereum block sync is much more difficult than before: First, the block data size is huge, which is close to TB level; Second, in order to full sync, it is better to use SSD due to the constraints of bandwidth and hard disc. Therefore, this solution will require a lot of work.
2. Public hosted Ethereum node
   Smart contract developer can also choose some public hosted nodes which are Ethereum full node themselves. Developer can connect to those nodes directly through programming interface with no need to setup up their own Ethereum node or wallet, e.g., Infura (http://infura.io) is a popular public hosted node. The drawback of the public hosted node is that developer must trust it, where there might be some security concerns. For development and testing, however, it is fast and efficient.

In order to use Infura in truffle framework, we need to install the following package:

```
npm install truffle-hdwallet-provider
```

2.2.2.3 Introduction of Metamask

MetaMask is a browser plug-in, and it plays a role of both Ethereum explorer and wallet. Through Metamask, you can interact with DApps and smart contract without



```
contract BasicToken {

  uint256 totalSupply_;
  mapping(address => uint256) balances;

  constructor(uint256 _initialSupply) public {
    totalSupply_ = _initialSupply;
    balances[msg.sender] = _initialSupply;
  }

  function totalSupply() public view returns (uint256) {
    return totalSupply_;
  }

  function transfer(address _to, uint256 _value) public returns (bool) {
    require(_to != address(0));
    require(_value <= balances[msg.sender]);
    balances[msg.sender] = balances[msg.sender] - _value;
    balances[_to] = balances[_to] + _value;
    return true;
  }

  function balanceOf(address _owner) public view returns (uint256) {
    return balances[_owner];
  }
}
```

```
mock compiler: source not found
```

Fig. 2.12 Pay attention to Compiler version

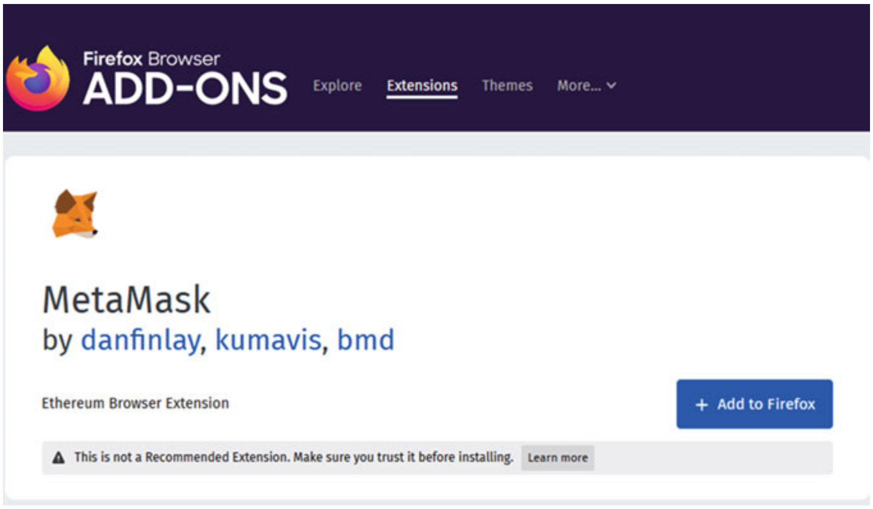
downloading software or blockchain. Instead, you only need to install plug-in and create wallet before you can start to transact, send, and receive ether. The biggest concern of Metamask is just like other online wallet: you must trust Metamask since all your information is stored on it, while it is possible that sensitive information is hacked or leaked.

2.2.2.3.1 Install Metamask

Official address of Metamask plug-in is https://metamask.io/. Metamask supports most of mainstream browsers, such as Firefox and Chrome (Fig. 2.13).

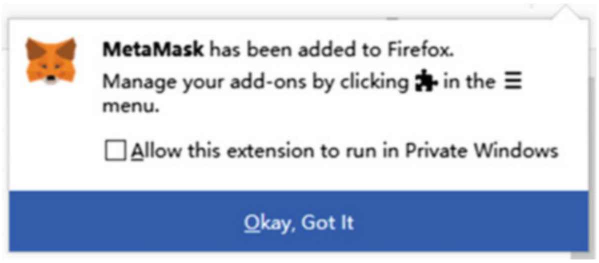
Click "Add to CHROME" to start installation. If succeed, it will show the following dialog (Fig. 2.14):

And then, Metamask will ask you to create new password (Fig. 2.15):



MetaMask
by danfinlay, kumavis, bmd
Ethereum Browser Extension
This is not a Recommended Extension. Make sure you trust it before installing.

Fig. 2.13 Metamask add-on on Firefox



MetaMask has been added to Firefox.
Manage your add-ons by clicking in the ☰ menu.
☐ Allow this extension to run in Private Windows

Fig. 2.14 Metamask added to Firefox

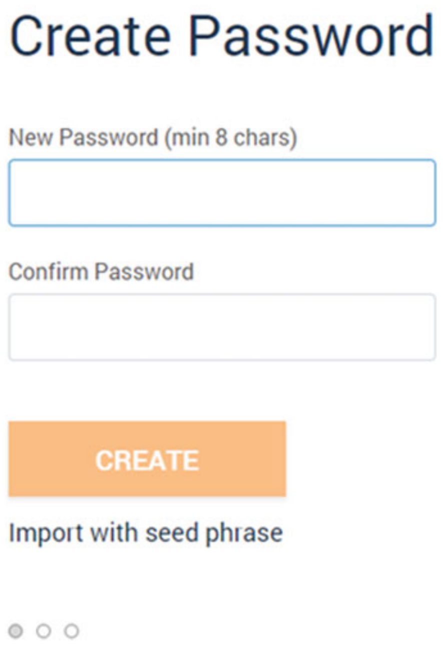


Fig. 2.15 Metamask added to Chrome browser

2.2.2.3.2 MetaMask

MetaMask will create 12 mnemonic phrases for each user. Please keep these mnemonic phrases in a safe place. You may need these words when you import the account into other wallets (Fig. 2.16).

Click the red key to copy it to clipboard. User must save it to a safe place and do not tell anyone else. Let us enter wallet page, as below (Fig. 2.17):

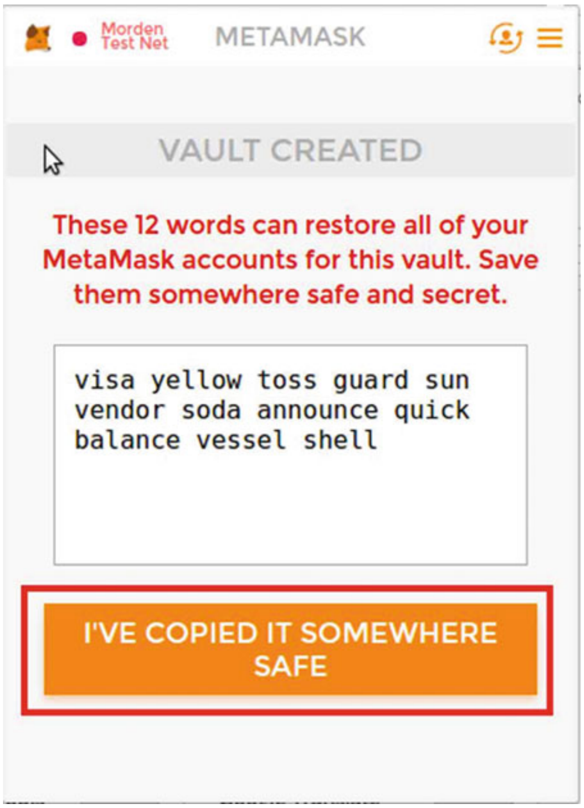
MetaMask already creates a wallet address for user automatically. If you cannot see the full address, clicking the "three dots" on the top-right and it will open a pop down menu related to wallet address as below (Fig. 2.18):

As shown above, there are two items in the pop down menu: the first item is the wallet QR-code. In "Account Details," you can export private key. The second item is to check all the transfer information related to the wallet on Etherscan (Fig. 2.19).

Here, you need to input the password which is set when wallet is created. Then, click "confirm."

Clicking the "Main Ethereum Network" on the top is to select the network which the wallet connects to (Fig. 2.20).

MetaMask prompts user that its default connection is to test net. But from the screenshot above, it now switched to mainnet. Clicking the setting icon on the top-right will show configuration menu as below (Fig. 2.21):



These 12 words can restore all of your MetaMask accounts for this vault. Save them somewhere safe and secret.

```
visa yellow toss guard sun
vendor soda announce quick
balance vessel shell
```

Fig. 2.16 Metamask mnemonic phrase

In the pop down menu, there are three main functions: create account, import account, and connect hardware wallet. The last menu item is "Setting," in which we can set up metrics of the crypto currency in wallet (Fig. 2.22).

Selecting the dropdown menu can select metrics including Fiat and crypto currency. User can choose their favorite currency for display.

2.2.2.3.3 Function of Metamask

The main function of Metamask is to deposit and transfer crypto currency (Fig. 2.23).

2.2.2.4 Introduction of Mist

Mist is an application based on electron, which means it is a desktop application with web interface. Mist includes a Geth node on the background. At the time Mist is

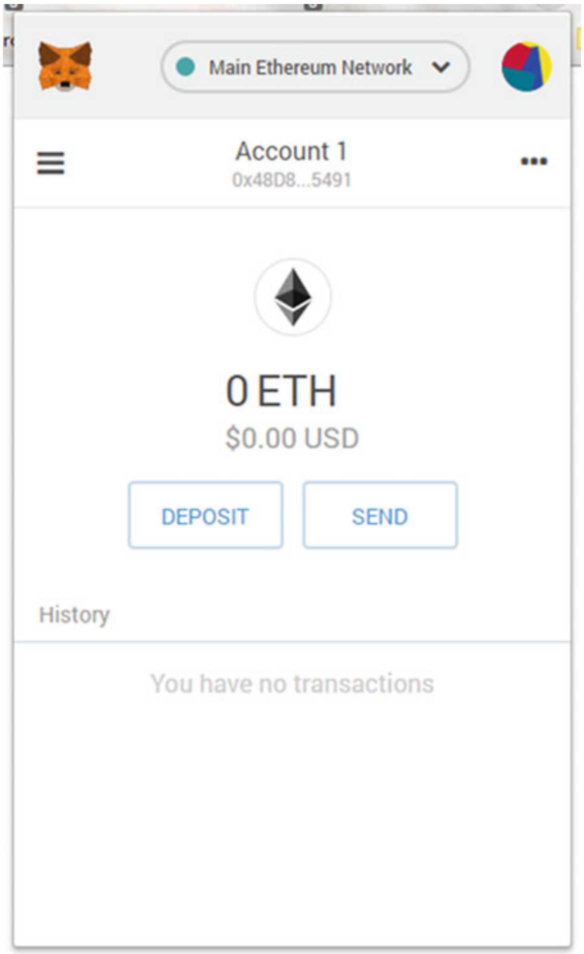


Fig. 2.17 Metamask account balance

started up, its connection to Ethereum blockchain is setup as well. Its project address is https://github.com/ethereum/mist. In Ubuntu, we can use the following command to install:

```
sudo dpkg -i ./Downloads/Ethereum-Wallet-linux64-0-11-1.deb
```

Download address is at: https://github.com/ethereum/mist/releases

We can start Mist with no parameter and it will start up an internal node. On the other side, if we already have a local Geth node, we can start Mist up with

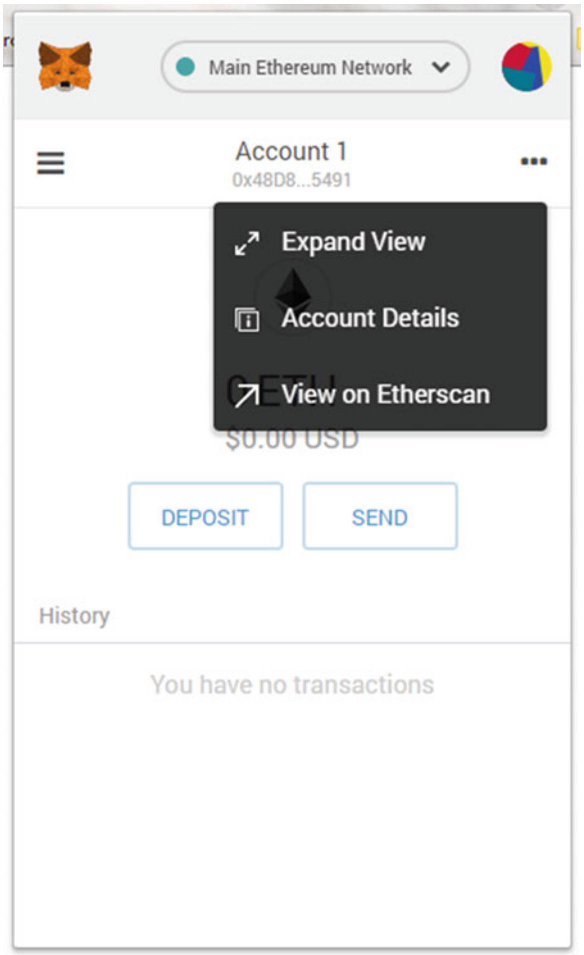


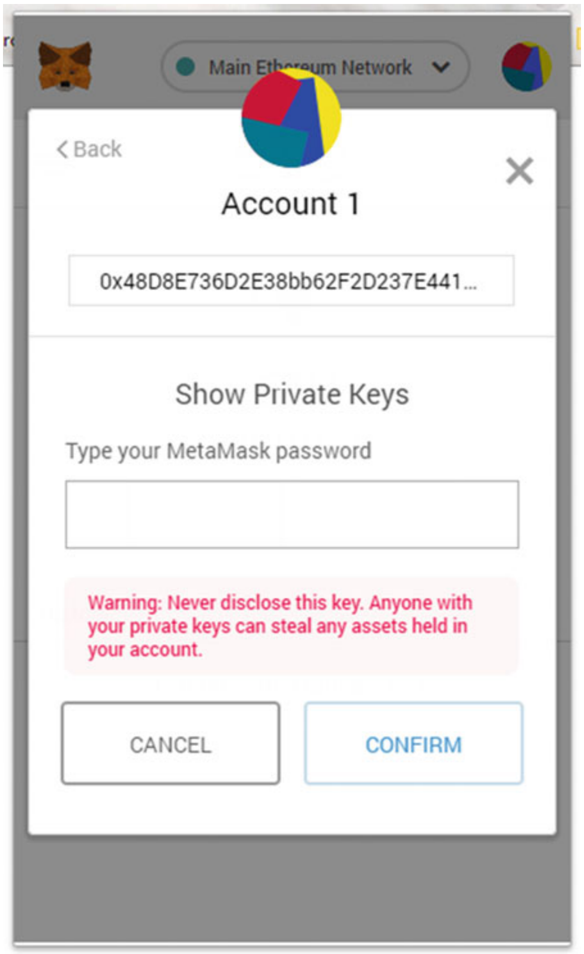
Fig. 2.18 Metamask dropdown menu

inter-process communication (IPC) path. In this situation, Mist connects to our local Geth node.

```
mist --rpc <test-chain-directory>/geth.ipc
```

Once Mist is started, please confirm that Mist is connected to a Geth node (Figs. 2.24 and 2.25).

As Mist integrates browser, wallet, testing node, and remix functions, through Mist, we can send transactions and check account balance, etc.



Warning: Never disclose this key. Anyone with your private keys can steal any assets held in your account.

Fig. 2.19 Metamask show private key

2.2.2.5 Other Tools

In order to explain the mechanism of Solidity smart contract programming, we may need to install solc—a command-line Solidity compiler. If Nodejs is already installed, we can use following commands to install solc. You may need to install node-gyp beforehand.

```
npm install -g solc
npm install -g solc-cli
```

If your operating system (OS) is Ubuntu,

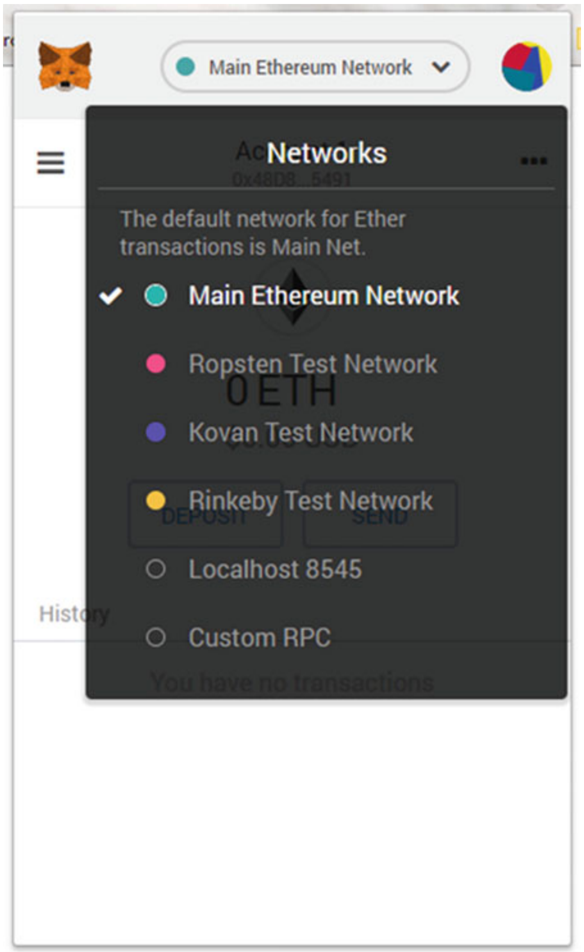


Fig. 2.20 Metamask connecting network

```
sudo add-apt-repository ppa:ethereum/ethereum
sudo apt-get update
sudo apt-get install solc
```

If your OS is Windows, you can go to https://github.com/ethereum/solidity/releases to download installation package.

To gain deep insight of smart contract, we need to install pyethereum and its related libraries to analyze the source code. Pyethereum is the Python implementation of Ethereum. We use Python 3.6 in this book.

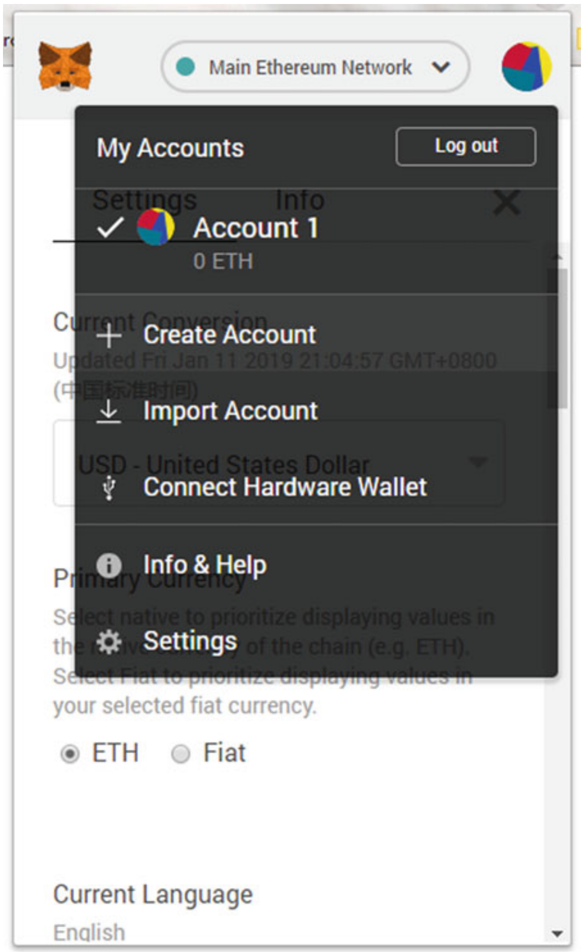
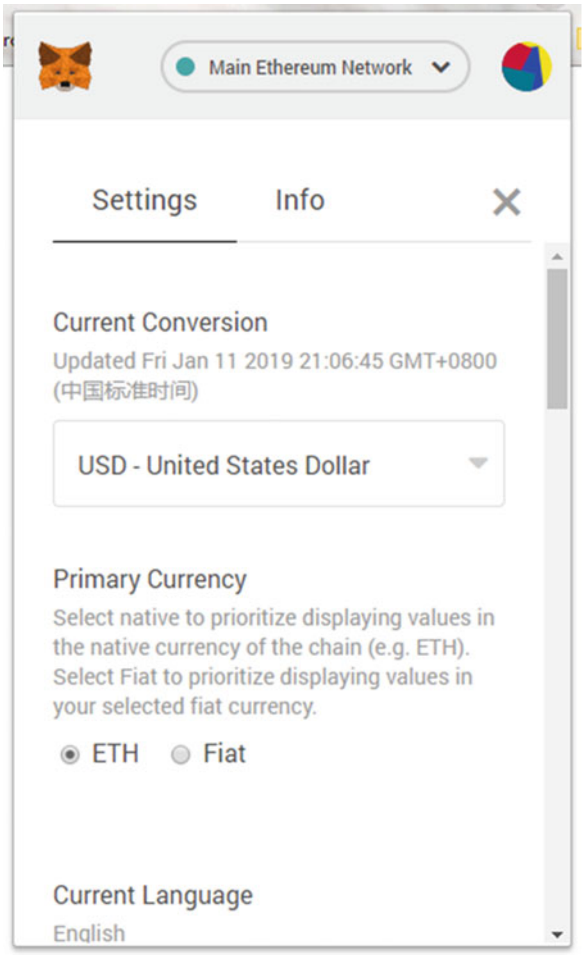

Main Ethereum Network
My Accounts
Log out
Account 1
0 ETH
Create Account
Import Account
Connect Hardware Wallet
Info & Help
Settings
Settings
Info
Current Conversion
Updated Fri Jan 11 2019 21:04:57 GMT+0800
(中国标准时间)
USD - United States Dollar
Primary Currency
Select native to prioritize displaying values in the native currency of the chain (e.g. ETH). Select Fiat to prioritize displaying values in your selected fiat currency.
ETH
Fiat
Current Language
English

Fig. 2.21 Metamask configuration menu

```
sudo apt-get install libssl-dev build-essential automake
pkg-config libtool
libffi-dev libgmp-dev libyaml-cpp-dev
git clone https://github.com/ethereum/pyethereum/
cd pyethereum
python setup.py install
```

To explain crypto functions and address calculation formula used in Ethereum, we need to install SHA3 library.

Fig. 2.22 Metamask configuration detail



Current Conversion
Updated Fri Jan 11 2019 21:06:45 GMT+0800 (中国标准时间)

USD - United States Dollar

Primary Currency
Select native to prioritize displaying values in the native currency of the chain (e.g. ETH). Select Fiat to prioritize displaying values in your selected fiat currency.

[✓ ETH] Fiat

Current Language
English

```
pip3 install pysha3
```

2.2.3 *Blockchain Explorer*

All the data on chain are public and transparent. That is to say, all blockchain data such as transaction and contract can be retrieved and verified through programming or block explorer. There are several popular browsers. We are using etherscan browser as an example here.

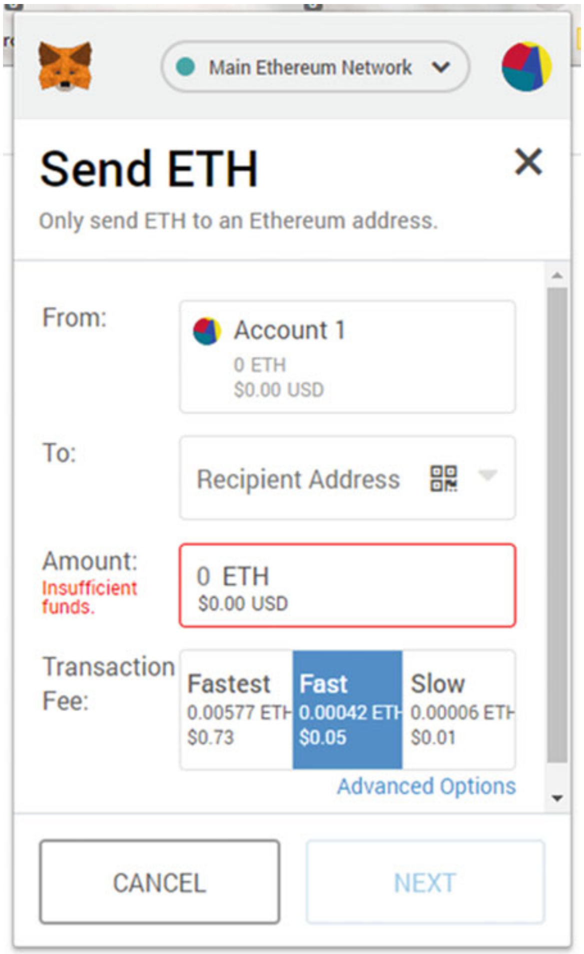


Fig. 2.23 Metamask sending ETH

Address of etherscan browser is at: https://etherscan.io

- *Query through contract address*:
  https://etherscan.io/token/0x6ebeaf8e8e946f0716e6533a6f2cefc83f60e8ab#readContract
- *Query through address*:
  https://etherscan.io/address/0x73d5c5f6a8925c817c7e5518592fe0b0a7cdf0af
- *Query through transaction ID*:
  https://etherscan.io/tx/0x837bf52f3a7eaa115fee9dde783b617976c907d59e2bd79014339f54c2b8decd

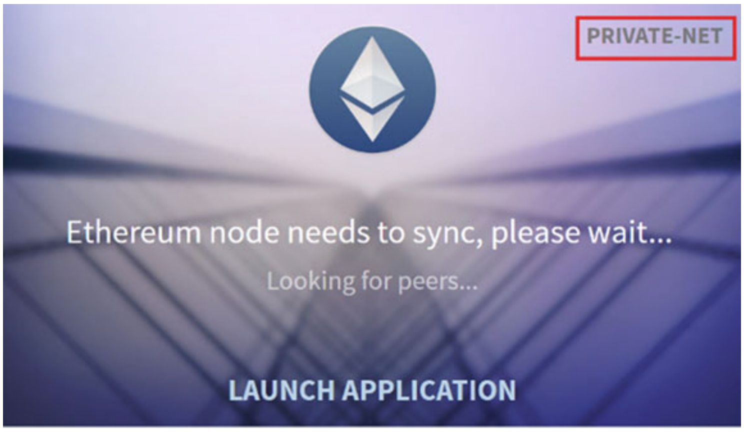


Fig. 2.24 Mist startup screen

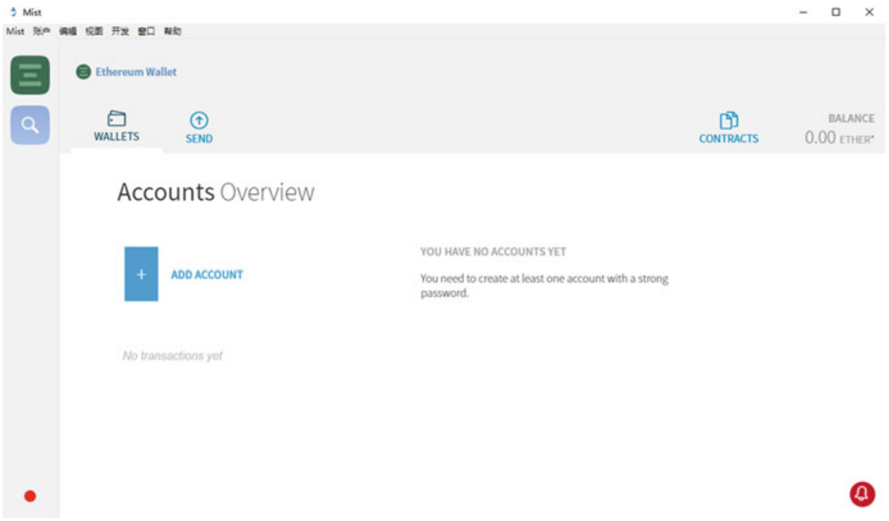


Fig. 2.25 Mist account overview

2.3 Testing Environment

	Ethereum	Explorer	Faucet
Rinkeby	Geth	https://rinkeby.etherscan.io/	https://www.rinkeby.io/#faucet
Ropsten	Geth	https://ropsten.etherscan.io/	http://faucet.ropsten.be:3001/
Kovan	Parity	https://kovan.etherscan.io/	https://app.chronologic.network/faucet

Besides mainnet, Ehtereum community also provides test net for developers to debug. Since running DApp costs certain amount of gas, debugging and testing in test net can save developers money. The first test net of Ethereum is Morden which started from July 2015 and completed on November 2016. Morden network was discarded because of consensus problem. The second test net is Ropsten which is started at the same time when Morden went offline. It ran until February 2017. Because of the lack of computing power of the test network itself, malicious attacker transmitted huge block data in the network, resulting in choking the whole network. The testing network is again unavailable.

2.3.1 *Metamask Switching Between Testing Environments*

Metamask plug-in can switch between different environments (Fig. 2.26).

2.3.2 *Obtain Testing Coins*

Since Solidity programming practice costs gas, for testing purpose, you will need to get some testing tokens for free. In addition, each testing net has its own blockchain explorer. We are picking Rinkeby as an example and introducing how to get free testing tokens. Anyone having a Twitter, Google+, or Facebook accounts may request funds within the permitted limits.

1. We need to publish a post on public web site (For example: Facebook, Twitter, Weibo, etc.) The post must contain your wallet address. For example, author (Gavin@gondola731) tweeted a tweet as following (Fig. 2.27):
2. Copy the URL of the post.
3. Go to https://www.rinkeby.io/#faucet, paste the URL of your post, select how much ETH you want, and click "Give me ether" button. You should receive your testing ETH shortly (Fig. 2.28).

2.3.3 *Connect to Testing Environment*

We use truffle to showcase how to connect to Rinkeby network. In order to connect Rinkeby, we need to modify truffle-config.js file:

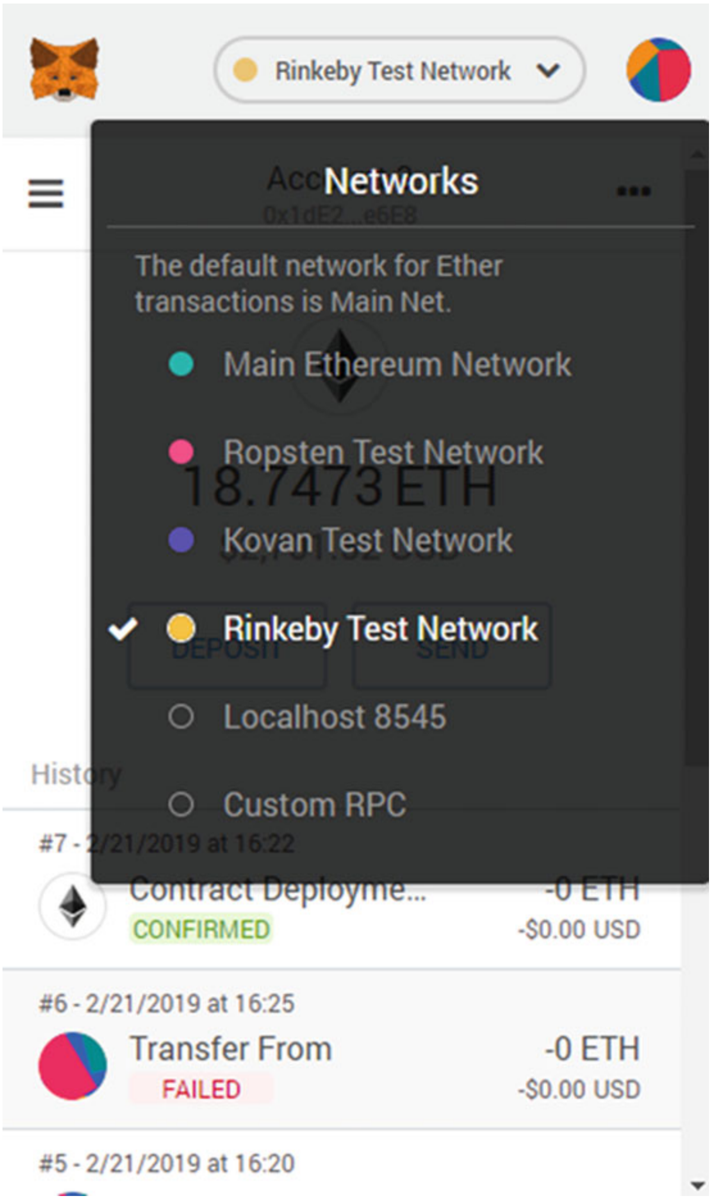
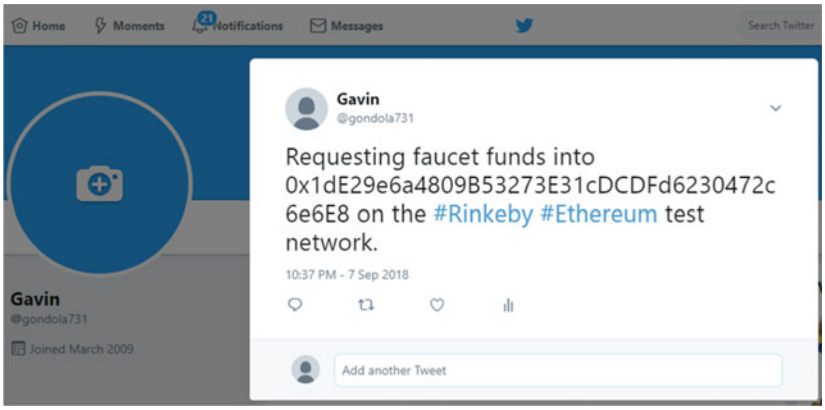


Fig. 2.26 Metamask Rinkeby network



Gavin
@gondola731
Joined March 2009

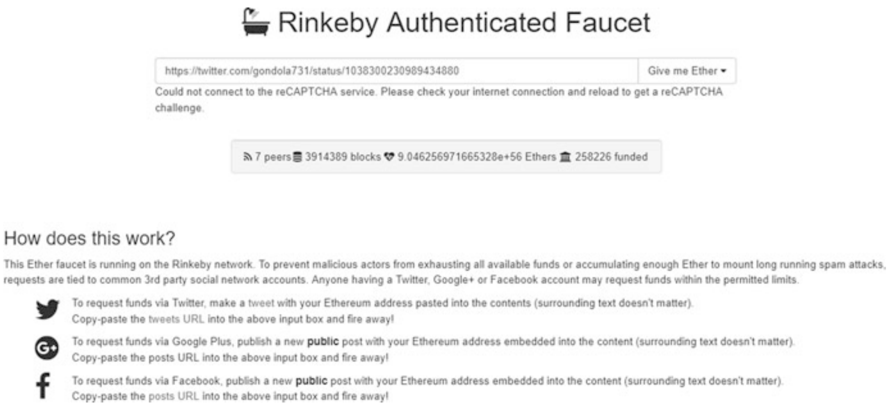
Gavin
@gondola731

Requesting faucet funds into 0x1dE29e6a4809B53273E31cDCDFd6230472c6e6E8 on the #Rinkeby #Ethereum test network.

10:37 PM - 7 Sep 2018

Add another Tweet

Fig. 2.27 Publish a message on Twitter



Rinkeby Authenticated Faucet

https://twitter.com/gondola731/status/1038300230989434880 | Give me Ether ▾

Could not connect to the reCAPTCHA service. Please check your internet connection and reload to get a reCAPTCHA challenge.

7 peers 3914389 blocks 9.046256971665328e+56 Ethers 258226 funded

How does this work?

This Ether faucet is running on the Rinkeby network. To prevent malicious actors from exhausting all available funds or accumulating enough Ether to mount long running spam attacks, requests are tied to common 3rd party social network accounts. Anyone having a Twitter, Google+ or Facebook account may request funds within the permitted limits.

To request funds via Twitter, make a tweet with your Ethereum address pasted into the contents (surrounding text doesn't matter).
Copy-paste the tweets URL into the above input box and fire away!

To request funds via Google Plus, publish a new **public** post with your Ethereum address embedded into the content (surrounding text doesn't matter).
Copy-paste the posts URL into the above input box and fire away!

To request funds via Facebook, publish a new **public** post with your Ethereum address embedded into the content (surrounding text doesn't matter).
Copy-paste the posts URL into the above input box and fire away!

Fig. 2.28 Rinkeby faucet

```
module.exports = {
  networks: {
     development: {
        host: "localhost",
        port: 8545,
        network_id: "*" // Match any network id
     },
     rinkeby: {
        host: "localhost",
        port: 8545,
        network_id: "4", // Rinkeby ID 4
        from: "0xf6e9dc98fe5d2951744967133b3c31765be657c1" //
Deploy Account
     }
     }
};
```

2.4 Ethereum Source Code Compilation

In this section, we show how to compile Ethereum Source code on Ubuntu. You can refer to https://github.com/ethereum/go-ethereum/wiki/Installing-Go#ubuntu-1404 for detailed instructions. It also contains information about how to compile on other platforms (such as Windows and MacOS).

First, we need to download Go language package and unzip it locally. Ethereum requires Go version above 1.8. Here we use Go version 1.9.4. For better speed, you can also use other mirror sites instead of storage.googleapis.com.

```
wget     https://storage.googleapis.com/golang/go1.9.4.linux-
amd64.tar.gz
sudo tar zxvf go1.9.4.linux-amd64.tar.gz
```

After the unzip, let us move Go package to /usr/local.

```
mkdir dev
sudo mv go /usr/local
```

And then, modify environment file and make Go package be accessible under current user environment.

```
sudo vi .bashrc
```

We need to append the following settings to .bashrc environment file:

```
export GOROOT=/usr/local/go
export GOPATH=/opt/goworkspace
export PATH=$PATH:$GOROOT/bin:$GOPATH/bin
```

To make effective of this modification, we use “source” command and check go version.

```
source .bashrc
go version
```

Next, we need to install some dependencies.

```
sudo apt install -y build-essential
```

And, we need to create source directory (Git needs to be installed).

```
mkdir geth
cd geth
```

Git clones the source code.

```
git init
git remote add origin https://github.com/ethereum/go-ethereum.
git
git pull origin master
```

The final step is to compile geth.

```
make geth
```

If there is no exception, you can now start to run geth.

Congratulations to all readers who get here, after you setup all necessary tools, dev and testing environment, you are fully prepared for Solidity programming.

Part II
Solidity Basics

Chapter 3
Solidity Basics

3.1 Sol File Structure

3.1.1 Pragma

```
pragma solidity ^0.4.0;
```

Pragma indicates that compiler version above 0.4.0 should be used to compile the Solidity source code. ^ means that compilation does not work if compiler version is earlier than 0.4.0. We can also specify the compiler version range as below:

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

3.1.2 Import

Global Import *

```
import "filename";
```

Import from self-defined namespace *

G. Zheng et al., *Ethereum Smart Contract Development in Solidity*,
https://doi.org/10.1007/978-981-15-6218-1_3

```
import * as symbolName from "filename"
```

Import separately

```
import {symbol1 as alias, symbol2} from "filename"
```

3.1.3 *Comment*

3.1.3.1 Code Comment

Solidity has two types of comment: single line (//) and multiple line (/*. . .*/)

```
// Single line comment
/*
This is a Multi-line comment
*/
```

3.1.3.2 Document Comment

The main function of document comment is to generate documentation automatically. Solidity uses /// or /** ... */, which follow Doxygen[1] syntax, to support document generation, parameter annotation and verification popped out when the function is called.

```
pragma solidity ^0.4.0;
/** @title Area Calculator*/
contract shapeCalculator{
    /**
    *@dev Calculate area and perimeter of rectangle
    *@param w Width of rectangle
    *@param h Height of rectangle
```

(continued)

[1]https://en.wikipedia.org/wiki/Doxygen.

```
    *@return s Area of rectangle
    *@return p Perimeter of rectangle
    */
    // Calculate area and perimeter of rectangle
    function rectangles(uint w, uint h) returns (uint s, uint p){
       s = w * h;
       p = 2 * ( w + h) ;
    }
}
```

3.1.4 Contract

```
pragma solidity ^0.4.11;
contract Contractexample {
  uint256 obj;
  function setObj(uint256 _para) {
     obj = _para;
  }
  function getObj() returns(uint256) {
     return obj;
  }
}
```

For those who are familiar with object-oriented programming language, it is easy to understand that contract can be thought as class. Contract also has constructor, member function, and member variables.

3.1.5 Library

The main difference between Library and Contract is that library cannot use fallback function, payable keyword, or define storage variable. However, library can modify storage variable in the contract which invokes it. For example, if we call a function in library from our contract and pass the storage variable defined in the contract to library function as parameter, the variable can be modified in library function and the change will be reflected in calling contract. You can think this as passing a C pointer to a C function.

Besides, library does not have Event log, but event can be emitted by library. Event triggered in library function is recorded in the Event log of calling contract. Here is a simple example:

```
library EventEmitterLib {
   function emit(string s) {
      Emit(s);
   }
   event Emit(string s);
}
contract EventEmitterContract {
   using EventEmitterLib for string;
   function emit(string s) {
      s.emit();
   }
   event Emit(string s);
}
```

When we use web3 library to develop DApp, we can receive event triggered by library functions when listening on EventEmitterContract.Emit event, while it cannot happen if we listen on EventEmitterLib's emit event.

3.1.6 Interface

```
pragma solidity ^0.4.11;
interface XXX {
  function cal(uint256 para) external view returns (uint256);
}
```

As Java, C++, Solidity interface just define prototype, instead of implementation.

3.2 Structure of Contract

A general contract has following parts:

1. State variables
2. Structure definitions
3. Modifier definitions
4. Event declarations
5. Enumeration definitions
6. Function definitions

We will explain them in more details.

3.3 Variable

Like traditional programming language, Solidity has basic data types, such as int, uint, struct, array, and mapping. Variable type can be categorized into two categories, namely Static and Dynamic. For state variable, the most important property of it is that the variable is stored into Ledger on blockchain by miners. Variables declared in contract and not defined in functions are state variables. Memory allocated to state variable is static and cannot be changed.

3.3.1 Value Type

3.3.1.1 Boolean

The value of bool is either true or false. Not like C programming language, 1 (number one) does not equal to true. Boolean type cannot be implicitly casted to integer. While, the explicit cast is allowable.

3.3.1.2 Integer

Signed integer and unsigned integer have different sizes, e.g., uint8, uint16, to uint256 or int8, int16 to int256. Unit and int also have their own alias, namely unit256 and int256

Integer literal consists of a sequence of numbers ranging from 0 to 9, which is interpreted as decimal by default. As octal is not supported in Solidity, leading 0 is ignored by default, such as 0300, which is considered as 300.

A decimal consists of .(dot) and must contain at least one number on its left or right. For example, 1.1 and 1.3 are all valid decimal numbers.

3.3.1.3 Address

The length of Ethereum address is 20 bytes (160 bit). So address is the same as uint160.

```
address addr= 0x692a70d2e424a56d2c6c27aa97d1a86395877b3a;
```

For all the properties and methods related to address, please refer to Sect. 3.10.1.3.

3.3.1.4 Fixed Byte Arrays

ByteN indicates fixed-size byte array and the range of N is from 1 to 32, such as bytes1, bytes2, bytes3 ... bytes32. Byte is the same as byte1. We can use hexadecimal literal or digital literal to set bytesi:

```
bytes1 aa = 0x75;
bytes1 bb = 10;
bytes1 ee = -100;
bytes2 cc = 256;
```

We also could use char to set bytesi:

```
bytes1 dd = 'a';
```

For bytes type, we can use bit operation: AND, OR, XOR, NOT, and Bit shift.

3.3.1.5 Rational and Integer Literals, String Literals

Literal itself is value. Solidity supports the following literal types:

- Integer Literal
  For example, 1, 10, -1, and -100.
- String Literal
  For example, “test.” String literals can use single quotation or double quotation.
- Address Literal
  For example, 0xfa35b7d915458ef540ade6078dfe2f44e8fa733c.
- Hexadecimal literal using 0x as prefix
  For example, “0x9A5C2F.”
- Digital Literal
  For example, 7.5 and 0.18.

3.3.1.6 Enum

Enum is a user-defined type, which can be converted to integer explicitly. However, the implicit conversion is not allowed.

```
pragma solidity ^0.4.0;
contract test {
   enum Direction {East, South, West, North}
   Direction choice;
...
}
```

3.3.1.7 Function Types

Function type is the type of function.

- Could assign a function to a variable—a function type variable.
- Could pass a function as a function parameter
- Could return a function in a method call

The complete function definition is as following:

```
function XXX(<parameter types>) {internal(default)|external}
[constant] [payable] [returns (<return types>)]
```

If the type is not specified, the default function type is internal. If the function does not have a return value, returns keyword must be omitted.

3.3.1.7.1 Internal Function

Internal function can only be used in the contract where it is defined. For instance, current code block, internal library function, or inherited function. It cannot be executed outside of current contract context. The default function type is internal.

3.3.1.7.2 External Function

External function consists of two parts: address and function signature. It can be used as parameter of external function call, or return by external function call.

3.3.2 Reference Type

Not like the value type described above, complex type occupies a larger space, which is more than 256 bytes. And it uses more space while copying variables. From the standpoint of saving space, solidity use reference to pass complex type. Here are some common reference type:

1. Dynamic byte array(bytes)
2. String
3. Array
4. Struts

3.3.2.1 Variable-Length Byte Array (Bytes)

Variable-length byte array—bytes, is a dynamic array and it can be any size. But it is different with byte []. byte [] array allocates 32 bytes for each element. In contrast, bytes pack all bytes together. Bytes can be used to declare a specified-length state variable as following code show:

```
bytes localBytes = new bytes(0) ;
```

Bytes variable can also be assigned as below:

```
localBytes = "Ritesh Modi";
```

Element can be pushed into byte array:

```
localBytes.push(byte(10));
```

Bytes variable also provide length property:

```
return localBytes.length; //reading the length property
```

3.3.2.2 String

In C programming language, string is ended with “\0”. Not like C, string in Solidity does not contain terminator. Therefore, the actual space allocated to string “test” is 4 bytes.

3.3.2.3 Array

We can use the keyword “new” to create a memory type array. The major difference between memory and storage array is: updating the length property cannot adjust the array size through. Let us have a look at the following example:

```
pragma solidity ^0.4.0;

contract example {
   function f() {
      // Create a memory array
      uint[] memory a = new uint[](7);
      // We can not update the length property of memory array
      // Erro info: Expression must be LValue
      // a.length = 100;
   }
   //storage Array
   uint[] b;
   function g(){
      b = new uint[](7);
      // We can adjust array length through setting length property
      b.length = 10;
      b[9] = 100;
   }
}
```

In the code above, f() function tries to adjust the length of memory array and compiler gives error information: “Error: Expression has to be an lvalue.” However, in g() function, for storage array, we can change the size of array through updating length property.

bytes: Dynamic byte array. It is not a value type

string:Dynamic UTF-8 encoded string. It is not a value type.

3.3.2.4 Struts

Struct is used to define user’s own data structure, which is a composite data type. It can include different types of member data. There is no code in struct and struct is composed of variables only.

```
  struct Funder {
     address addr;
     uint amount;
  }
```

3.3.3 *Mapping*

Mapping is a kind of key/value pair. The syntax is like mapping (KeyType => _KeyValue). Key can be any type except mapping, such as array, contract, enum, struct, etc. For example:

```
mapping(address => uint) public balances;
```

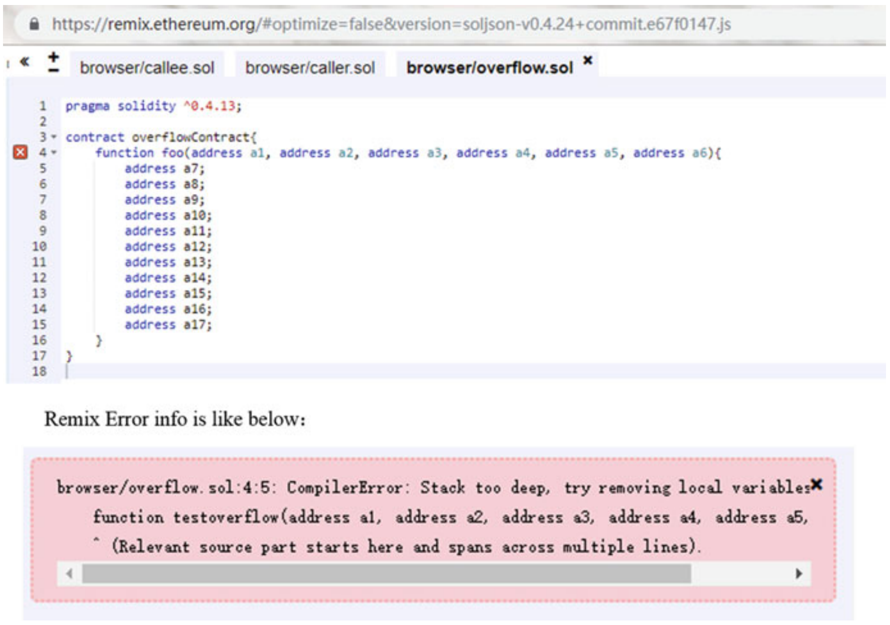
3.3.4 *Special Case*

In runtime, variable is allocated on stack. Due to the constraint of stack depth (Maximum 16), if too many local variables are defined, stack overflow error will occur. Here is the demo:

```
pragma solidity ^0.4.13;
contract overflowContract{
  function testoverflow(address a1, address a2, address a3, address
a4, address a5, address a6){
    address a7;
    address a8;
    address a9;
    address a10;
    address a11;
    address a12;
    address a13;
    address a14;
    address a15;
    address a16;
    address a17;
  }
}
```

We will see following errors:

```
browser/overflow.sol:4:5: CompilerError: Stack too deep, try
removing local
variables.
  function testoverflow(address a1, address a2, address a3,
address a4, address a5, address a6){
  ^
Spanning multiple lines.
```



https://remix.ethereum.org/#optimize=false&version=soljson-v0.4.24+commit.e67f0147.js

```
pragma solidity ^0.4.13;

contract overflowContract{
    function foo(address a1, address a2, address a3, address a4, address a5, address a6){
        address a7;
        address a8;
        address a9;
        address a10;
        address a11;
        address a12;
        address a13;
        address a14;
        address a15;
        address a16;
        address a17;
    }
}
```

Remix Error info is like below:

```
browser/overflow.sol:4:5: CompilerError: Stack too deep, try removing local variables
    function testoverflow(address a1, address a2, address a3, address a4, address a5,
    ^ (Relevant source part starts here and spans across multiple lines).
```

Fig. 3.1 Stack overflow

The error is caused by stack depth exceeding 16. The official recommendation is to use less variable in Solidity programming and make the function body as small as possible. There are other alternatives: wrap variables into struct, or array, or use keyword—memory (Fig. 3.1).

3.4 Operators

Here are the Solidity operators:

No.	Variables	Descriptions
1	Postfix Increment and Decrement	++, --
2	Function Call	<func>(<args...>)
3	Array Address	<array>[<index>]
4	Member Access	<object>.<member>
5	Bracket	(<statement>)
6	Prefix increment and decrement	++, --
7	Unitary add/substract	+, -
8	Unitary delete	delete
9	Logical Not	!
10	Bitwise Not	~

(continued)

No.	Variables	Descriptions
11	Exponent	**
12	Multiply, Divide, Modulo	*, /, %
13	Add/Subtract	+, -
14	Bit Shift	<<, >>
15	Bit And	&
16	Bit XOR	^
17	Bit OR	|
18	Larger than, smaller than, larger than or equal to, smaller than or equal to	<, >, <=, >=
19	Equal or not equal	==, !=
20	Logic AND	&&
21	Logic OR	||
22	Ternary Operator	<conditional> ? <if-true> :<if-false>
23	Assignment	=, |=, ^=, &=, <<=, >>=, +=, -=, *=, /=, %=
24	Comma	,

3.5 Statement

Solidity does not support switch and goto. The bracket in if condition statement is not omittable. But in the single line statement, curly bracket can be omitted.

3.5.1 Conditional Statement

In Solidity, if/else is just like other programming languages. But if if/else code block only contains just one statement, curly brace {} could be omitted. Attention: In Solidity, if(1){...} is invalid. If you really want to use this statement, you must force a type cast to convert 1 to Boolean value true.

```
if (totalPoints > bet.line)
     balances[bet.over] += bet.amount * 2;
else if (totalPoints < bet.line)
     balances[bet.under] += bet.amount * 2;
else { // refunds for ties
     balances[bet.under] += bet.amount;
     balances[bet.over] += bet.amount;
}
```

3.5.2 *Loop*

Example of While statement:

```
// While loops
uint insertIndex = stack.length;
while (insertIndex > 0 &&
     bid.limit <= stack[insertIndex-1].limit) {
     insertIndex--;
}
```

Example of for statement:

```
address[] public addressIndices;
// start adding address in array
addressIndices.push(newAddress);
...
// We know the length of the array
uint arrayLength = addressIndices.length;
for (uint i=0; i<arrayLength; i++) {
   // do something
}
```

3.5.3 *Miscellaneous*

Break: Jump out of current loop

Continue: Exit current round of loop and jump to next round of loop

Return: Return from function/Method

?: Ternary operator example: a>b?a:b return a if a>b, or return b

3.6 Data Location

Each variable declaration in the contract has its own data location. EVM provides the following four types of data locations:

1. Storage

   Storage variable is a global variable accessible to all function. And it is persistent, which means that storage variables will be saved to each full node in the public chain network.

2. Memory
   The lifetime of local memory variable in the contract is ephemeral: memory variable is destroyed when the function is finished.
3. Calldata
   All function calls use calldata, which includes function parameters. Calldata is read-only memory area
4. Stack
   EVM maintains a stack to load variable and EVM opcodes. The stack is the working environment of EVM. It has a depth of 1024. If it stores more than 1024 levels, an exception will be thrown.

Default function parameters including return values are all memory. Default local variables are storage. And default state variables (public variables declared by contract) are allocated in storage.

Parameters of external function (return parameter exclusive) are forced to calldata.

Data location is very important since the assignment between variables in different data location will have different effects. Assignment between a memory and a storage variable or between storage variables will create an extra irrelevant data copy.

Assigning a storage state variable to a storage local variable will be a reference assignment. So modifying local variable will be visible to its related state variable. Assigning a memory reference type to another memory reference type will not create a data copy.

We should consider the data location seriously: memory (non-persistent) or storage (persistent). Complex types, such as array and struct, have an extra attribute—data location in Solidity. The attribute value can be memory or storage.

Memory data location is just the same as memory in the traditional programming language. It is accessible after allocated and it becomes inaccessible if program steps out of its scope. Comparatively, on the blockchain, since its infrastructure is based on Turing-complete machine, there are so many states which need to be stored permanently. For example, the participant information of crowdfunding. In this situation, we need to use the storage type. Based on program context, in most situations, Solidity default data location is storage and we can specify storage or memory to modify default data location.

The third type of data location is called calldata. It stores function parameters and it is read-only and non-persistent.

```
pragma solidity ^0.4.0;
contract DataLocation{
  uint valueType;
  mapping(uint => uint) public refrenceType;
  function changeMemory(){
```

(continued)

```
        var tmp = valueType;
        tmp = 100;
    }
    function changeStorage(){
        var tmp = refrenceType;
        tmp[1] = 100;
    }
    function getAll() returns (uint, uint){
        return (valueType, refrenceType[1]);
    }
}
```

Summary
Forced data location

- Parameters of external function (not including return value) are forced to be calldata
- State variables are forced to storage

Default data location

- Function parameters (including return value): memory
- All other local variables: storage

Different storage location cost different amount of gas

1. Storage variable will be saved permanently and therefore it is costly.
2. Memory variable only saves value temporarily and memory variable will be freed after the function call. So it costs much less.
3. Stack keeps small local variables and it is free. But it has its own constraints: the maximum count of variables is 16.
4. Calldata contains the message body and its calculation will add Gas cost of n*68

3.7 Modifier

Function declaration uses the following template. Besides function name and parameters, we can add modifiers. In this section, we mainly introduce modifiers—their usage and difference.

```
function XXX(<parameter types>) {internal(default)|external}
[constant]
[payable] [returns (<return types>)]
```

3.7.1 Standard Modifier

Function like state variable has visibility modifier. Possible visibility modifiers are:

```
External, Internal (default), Public(default), Private
```

Function also has other modifiers which specify the accessibility/mutability of state variable:

```
Constant, View,  Pure, Payable
```

3.7.1.1 Internal Modifier

Functions and state variables defined with internal can only be accessed from internal: current contract or contract inheriting from it. Please note: do not use this keyword here since this keyword indicates that it is external access. If there is no modifier, internal is the default modifier. Internal is similar to protected keyword in Java or C++. The internal function is not accessible from external, and it is not a part of the contract interface.

3.7.1.2 External Modifier

We can call external functions from an external contract or through transaction since external functions are a part of the contract interface. For an external function f, it cannot be called from internal (For example, f() is invalid, but this.f() is ok). External functions are more effective in receiving a big size array.

3.7.1.3 Public Modifier

Public functions are a part of the contract interface. And it can be invoked through internal or external message call. For public state variable, the compiler will create getter and setter function automatically.

3.7.1.4 Private Modifier

Private functions and state variables are only visible to the current contract. They are not accessible from the inherited contract. And private functions are not a part of the contract interface.

3.7.1.5 Constant Modifier

Constant functions cannot modify state on chain. But they can read the value of state variables and return it to caller. Constant function body is not able to change variables, create another contract, trigger event, and invoke other functions which may change the state on chain.

Constant is included in function's JSON ABI file, and it is used by web3.js library to determine a function call is initiated by a transaction or a direct call.

```
contract HelloVisibility {
   function hello() constant returns (string) {
      return "hello";
   }
   function helloLazy() constant returns (string) {
      return hello();
   }
   function helloAgain() constant returns (string) {
      return helloQuiet();
   }
   function helloQuiet() constant private returns (string) {
      return "hello";
   }
}
```

Hello functions can be called by any other contract including itself (check helloLazy function). Other function in the contract can call helloQuiet function directly and helloAgain function shows how to do it. But if helloQuiet is called from an external contract, an error will be raised.

In the example above, all function is declared with constant since they will not change the world state.

3.7.1.6 View Modifier

Function declared with View cannot modify state variable. It equals to constant function.

3.7.1.7 Pure Modifier

Pure specifies more constraints for functions. Pure cannot read/write state variable. Function modified by pure cannot access current state and transaction variables.

3.7.1.8 Payable Modifier

Payable functions can accept ether from callers. If sender does not provide ether, call may fail. Payable functions can only accept ether.

3.7.1.9 Difference Between Modifiers

3.7.1.9.1 External and Public

From the visibility point of view, external is almost the same as public. After public/external functions in contract are deployed to blockchain, functions can be called by other contracts: through direct call or transaction.

The main difference between external and public is how they are implemented and how parameters are processed. In contract, a call to a public function is implemented by JUMP opcode, just like private and internal functions, while external functions are implemented through CALL opcode. Besides, the external function does not copy parameters from calldata to memory or stack.

Following example shows how external and public functions behave differently:

```
pragma solidity ^0.4.12;
contract Test {
  /*
  Can be called by internal or external function
  Since internal function read parameters from memory, Solidity
needs
  to copy array data into memory and it costs extra gas
  */
  function test(uint[20] a) public returns (uint) {
    return a[10] * 2;
  }
  /*
  Cannot be called from internal. And it read parameters directly
from
  CALLDATA. And there is no need to copy parameter into memory
  */
  function test(uint[20] a) external returns (uint) {
    return a[10] * 2;
```

(continued)

```
    }
    /*
    Code is invoked by JUMP opcode.
    Array type parameter is passed through pointer to memory.
    Function body access parameter through memory pointer
    */
    function test(uint[20] a) internal returns (uint) {
      return a[10] * 2;
    }
}
```

If we call test and test2 function in this contract, it will cost 496 gas for public function test(), while the external function just cost 261 GAS. (Please note: with the upgrade of Solidity compiler and EVM, Gas amount expended may vary) The reason is: in public function, Solidity copies array into memory while external function reads parameters from calldata. Memory allocation is expensive. So reading from calldata is comparatively cheap.

Why Solidity copy all parameters of public function to memory? The main reason is: it is possible to call a public function from an internal function. Comparatively, external call to a public function happens in a totally different context. The internal call is implemented by Jump opcode and array parameter is passed in the form of a memory pointer. So, when Solidity compiler generates opcode for internal function, function accesses parameter through memory pointer. Solidity compiler does not allow calling an internal function in an external function, so it allows to read parameters directly from calldata and skip the step of copying parameters to memory.

Now, let us summarize:

- The internal call is always the cheapest since it is implemented through JUMP opcode and parameter is passed through memory pointer.
- Since the public function does not know the caller is external or internal, the public function always copies parameters to memory like what internal function does. And this operation is pretty expensive.
- If you are absolutely sure that the function will be called by an external contract only, please use the external modifier.
- In most situations, this.f() is meaningless since it will use an expensive CALL opcode.

3.7.1.9.2 Internal vs External

Following example demos the method to call internal and external function. Inline comments clearly explain the difference:

```
pragma solidity ^0.4.5;
contract FuntionTest{
   function internalFunc() internal{}
   function externalFunc() external{}
   function callFunc(){
     // Call internal function directly
     internalFunc();

     // Cannot call an external func from an internal func.
     // Compilation error: not declared identifier
     //externalFunc();

     // Cannot call an `internal` function in an `external` way
     // Error: Cannot find function "internalFunc" or not visible
     //this.internalFunc();

     // Using `this` to call an external function in an `external` way
     this.externalFunc();
   }
}
contract FunctionTest1{
   function externalCall(FuntionTest ft){
     // Call external function in another contract
     ft.externalFunc();

     // Cannot call internal function of another contract
     // Error:Cannot find member function "internalFunc" or
     // invisible in FuntionTest
     //ft.internalFunc();
   }
}
```

3.7.2 *Self-defined Modifier*

A modifier can be used to change function behavior. For example: modifier could be used to check the condition before the execution of a function. A modifier can be inherited and can be overridden by derived contract.

```
pragma solidity >0.4.24;
contract owned {
    constructor() public { owner = msg.sender; }
    address payable owner;
```

(continued)

```
    // We define a modifier and use it in derived contract.
    // And the body of function will be add after the special
    // character `_` in modifier
    // This means that function body will be run here if contract owner
    // call this function, or raise exception
    modifier onlyOwner {
        require(
            msg.sender == owner,
            "Only owner can call this function."
        );
        _;
    }
}
contract mortal is owned {
    // This contract inherits `onlyOwner` modifier from owned
contract
    // Its function is to allow only owner to call `close`
    function close() public onlyOwner {
        selfdestruct(owner);
    }
}
contract priced {
    // Modifier can accept parameter as input
    modifier costs(uint price) {
        if (msg.value >= price) {
            _; // _ indicate function body
        }
    }
}
contract Register is priced, owned {
    mapping (address => bool) registeredAddresses;
    uint price;
    constructor(uint initialPrice) public { price = initialPrice; }
    // Please note: `payable` is indispensable, or function will
    // reject all ether sent to it
    function register() public payable costs(price) {
        registeredAddresses[msg.sender] = true;
    }
    function changePrice(uint _price) public onlyOwner {
        price = _price;
    }
}
// Mutex contract below set a mutex lock on contract
// Mutex contract protect itself by using modifier to avoid
re-entrancy
// attack
contract Mutex {
    bool locked;
    modifier noReentrancy() {
        require(
```

(continued)

```
            !locked,
            "Reentrant call."
        );
        locked = true;
        _; // modified function boday
        locked = false;
    }
    /// This function is protected by Mutex lock. This means that
    /// `msg.sender.call` cannot call f. This prevent re-entrancy
attack
    /// `return 7` statement return 7 and then execute `locked =
false`
    /// in modifier
    function f() public noReentrancy returns (uint) {
        (bool success,) = msg.sender.call("");
        require(success);
        return 7;
    }
}
```

3.8 Event

Event is a convenient interface provided by EVM infrastructure, which is used to capture the event.

```
pragma solidity ^0.4.0;
contract SimpleAuction {
    event aNewHigherBid(address bidder, uint amount);
    function bid(uint bidValue) external {
        aNewHigherBid(msg.sender, msg.value);
    }
}
```

Maximum three parameters can be defined as indexed. Once the parameter is set to be indexed, it means that the parameter can be used to query log, even can be filtered. If array (including string and bytes) is defined with indexed, then its Keccak-256 hash value will be used as topic. Except for anonymous event, all event signatures (For example: Deposit (address, hash256, uint256)) is one topic. This also means that an anonymous event cannot be filtered by name. Through function log0, log1, log2, log3, log4, developers can access low-level log component. Logi indicates that it has i + 1 parameters (i is the number of parameters and is counted from 0). The first parameter is used as the data of log events, and the others are used as topics.

Event and log have three main usages:

- Smart contract can use event to return value to the user interface
- Async triggers with data
- A cheaper storage

3.8.1 Return Value to UI

A simple use case of an event is to return value to App frontend. See following example:

```
contract ExampleContract {
   //some state variables ...
   function foo(int256 _value) returns (int256) {
      // manipulate state ...
      return _value;
   }
}
```

Assume that exampleContract is an instance of ExampleContract, and App frontend uses web3.js, you can get the return value if you run it under JavaScript VM.

```
var returnValue = exampleContract.foo.call(2);
console.log(returnValue) // 2
```

But once the contract call is submitted, it takes some time for miner to pack contract calls into the block as formal transactions. That is why we cannot get the return value of contract call instantly.

```
var returnValue = exampleContract.foo.sendTransaction(2, {from:
   web3.eth.coinbase});
console.log(returnValue) // transaction hash
```

sendTransaction function always returns the submitted transaction's hash. The reason why function does not return value to the frontend is that the transaction is probably not packed and is not on-chain yet.

If a return value is needed, the recommendation will be an event, which is event intend to do.

```
contract ExampleContract {
  event ReturnValue(address indexed _from, int256 _value);
  function foo(int256 _value) returns (int256) {
    ReturnValue(msg.sender, _value);
    return _value;
  }
}
```

Frontend can return values:

```
var exampleEvent = exampleContract.ReturnValue({_from: web3.
eth.coinbase});
exampleEvent.watch(function(err, result) {
  if (err) {
    console.log(err)
    return;
  }
  console.log(result.args._value)
  // check that result.args._from is web3.eth.coinbase then
  // display result.args._value in the UI and call
  // exampleEvent.stopWatching()
})
exampleContract.foo.sendTransaction(2, {from: web3.eth.
coinbase})
```

Once the transaction which invokes foo() is mined, the callback function in watch will be triggered. And it will allow frontend to get return value from foo().

3.8.2 Async Trigger with Data

Event can be seen as an asynchronized trigger with data. If a contract needs to interact with frontend, the contract will send an event; If frontend is listening on event, or a contract needs to trigger frontend, the contract will send an event. If frontend is listening on event, frontend will capture the event from the contract and take some action, for example: show some message, etc.

3.8.3 Cheap Storage

Event can also be used as a cheaper storage. In the yellow paper of Ethereum, event is considered as a log which has LOG opcodes. Data can be saved into LOG.

Whenever an event is triggered, its log is written into the blockchain. Events and logs may cause confusion.

Logs are designed as some type of storage and it costs much less gas than contract storage does. In Logs, each byte will cost 8 gas, while contract storage variable will cost 20,000 gas every 32 bytes. Although logs can save gas, logs are not accessible from other contracts.

Besides acting as a trigger to frontend, a log can also be used as a cheaper storage. An example is rendering historical data from logs.

A crypto currency exchange may need to show the charging and withdrawal history of a user. A solution is to save all charging records to logs which is comparatively cheaper than save them in the contract. Exchange platform only needs to know account balance which can be saved in contract, all the history info can be saved somewhere else, for example, log.

```
contract CryptoExchange {
  event Deposit(uint256 indexed _market, address indexed _sender,
uint256 _amount, uint256 _time);
  function deposit(uint256 _amount, uint256 _market) returns
(int256) {
     Deposit(_market, msg.sender, _amount, now);
  }
}
```

Assume that we need to update UI when the user is in the process of charging, we show how an event can be used as a trigger with data. Assume cryptoExContract is an instance of a CryptoExchange.

```
var    depositEvent    =    cryptoExContract.Deposit({_sender:
userAddress});
depositEvent.watch(function(err, result) {
  if (err) {
   console.log(err)
   return;
  }
})
```

_sender is defined as indexed to increase the efficiency of retrieving event.

```
event Deposit(uint256 indexed _market, address indexed _sender,
uint256
_amount, uint256 _time)
```

Event listening will be started after event is instantiated. When UI is loaded, Deposit is not effective yet. So we need to collect all information starting from block 0, and this can be implemented by adding “fromblock” parameter to event.

```
var   depositEventAll   =   cryptoExContract.Deposit({_sender:
userAddress},
{fromBlock: 0, toBlock: 'latest'});
depositEventAll.watch(function(err, result) {
  if (err) {
   console.log(err)
   return;
  }
  // append details of result.args to UI
})
```

If we need to render UI, we need to invoke depositEventAll.stopWatching().

3.8.4 Indexed Parameter in Event

An event can have maximum to three indexed parameters. For example, an event is defined as below:

```
event  Transfer(address  indexed  _from,  address  indexed  _to,
uint256 _value) .
```

This means that a frontend application can monitor token transfer:

- Send from an address: tokenContract.Transfer({_from: senderAddress})
- Receive from an address: tokenContract.Transfer({_to: receiverAddress})
- Transfer from an address to another address: address tokenContract.Transfer ({_from: senderAddress, _to: receiverAddress})

3.9 Inheritance

3.9.1 Single Inheritance

Solidity supports multiple inheritances through copy mechanism. All functions are virtual, which means that furthest derivation will be called unless the contract is specified explicitly. Derived contract should provide all parameters required by all parent contracts. Therefore, two methods can be employed, as shown below:

```
pragma solidity ^0.4.0;
contract Base {
   uint x;
   function Base(uint _x) { x = _x; }
}
contract Derived is Base(7) {
   function Derived(uint _y) Base(_y * _y) {
   }
}
```

- Use “is Base(7)” to inherit directly
  Convenient for the situation that constructor is constant.
- As a part of the constructor of derived contract Base(_y * _y)
  Fit for constructor parameter specified by derived contract.

If both methods above are used, the second one will have higher priority (Usually, we just use one of them).

3.9.2 Multi-Inheritance

In multi-inheritance, the order of base contracts is very important. When a contract inherits from multiple contracts, there will be only one contract created. Solidity copies code from all inherited contracts into inheriting contract. If inheritance leads to the name conflicts of modifier or function in one contract, an error will occur. By the same token, if there are name conflicts between event and modifier, or between event and function in contract, an error will be triggered.

Solidity supports multiple inheritance, meaning that one contract can inherit several contracts. Multiple inheritance introduces ambiguity called Diamond Problem: if two or more base contracts define the same function, which one should be called in the child contract? Solidity deals with this ambiguity by using reverse **C3 Linearization** (an algorithm used primarily to obtain the order in which methods should be inherited in the presence of multiple inheritance), which sets a priority between base contracts. That way, base contracts have different priorities, so the order of inheritance matters. Neglecting inheritance order can lead to unexpected behavior. When inheriting multiple contracts, especially if they have identical functions, a developer should carefully specify inheritance in the correct order. The rule of thumb is to inherit contracts from more/general/to more/specific/.

A programming language supporting multiple inheritance should solve several problems. One of them is diamond inheritance problem (Fig. 3.2).

Solidity solution for Diamond Problem is based on Python, which use C3_linearization mechanism to force converting base contracts to a directed acyclic graph (DAG).

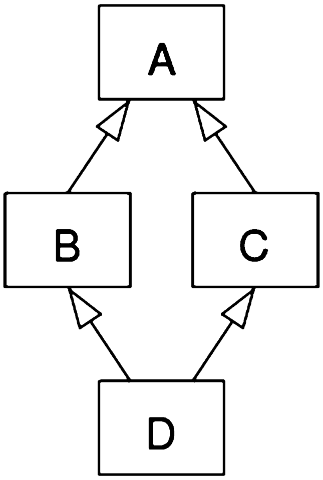


Fig. 3.2 Diamond inheritance problem

```
pragma solidity ^0.4.0;
contract base {}
contract A is base {}
contract C is A, base {}
```

The code snippet above will throw an error since contract C requests contract base to rewrite contract A (since the inheritance order is A and then base), while contract A rewrites contract base. And this will cause an insolvable conflict to Solidity compiler.

3.10 Miscellaneous

3.10.1 Built-in Variable

3.10.1.1 Special Variable

No	变量	说明
1	block.blockhash (uint blockNumber) returns (bytes32)	Block hash for the specified block number. Only support latest 256 blocks (excluding current block)
2	block.coinbase (address)	Current block miner address
3	block.difficulty (uint)	Current block difficulty
4	block.gaslimit (uint)	Current block gaslimit
5	block.number (uint)	Current block number
6	block.timestamp (uint)	Current block timestamp
7	msg.data (bytes)	calldata

(continued)

No	变量	说明
8	msg.gas (uint)	Current gas remainder
9	msg.sender (address)	Current call initiator
10	msg.sig (bytes4)	First 4 bytes of calldata which is function signature
11	msg.value (uint)	Ether amount used for this call. Unit is wei
12	now (uint)	Current block timestamp. Equals to block.timestamp
13	tx.gasprice (uint)	Transaction gas price
14	tx.origin (address)	Transaction initiator

3.10.1.2 Mathematical and Cryptographic Functions

No	Variable	Description
1	keccak256(...) returns (bytes32)	Using Keccak-256 to calculate hash
2	sha3(...) returns (bytes32)	Equals to keccak256()
3	sha256(...) returns (bytes32)	Using SHA-256 to calculate hash value
4	ripemd160(...) returns (bytes20)	Using RIPEMD-160 to calculate hash value
5	ecrecover (bytes32 hash, uint8 v, bytes32 r, bytes32 s) returns (address)	Recover public key from signature. Return 0 if error
6	revert()	Transaction rollback

Please bear in mind that Keccak and sha-3 are not the same. In 2007, U.S. National Institute of Standard and Technology (NIST) initiated a competition about SHA-3. In 2012, Keccak team won the competition. From then on, developers implemented lots of "sha3" solution based on Keccak. However, in 2014, NIST modified Keccak solution and released FIPS 202, and this updated proposal becomes official SHA-3 standard on Aug 2015. Many "old" program still use Keccak, and do not upgrade to official SHA-3 standard. We explain the difference between SHA-3 and Keccak below and hope it can be of help to programmers (Fig. 3.3).

Please note that "old" code based on Keccak does not generate the same hash value as SHA-3 does. So, if using a "sha3" library, you should be crystal clear that the library is based on Keccak or based on standard SHA-3. A simple solution is doing a test for empty input:

Fig. 3.3 SHA2/KECCAK/SHA3

SHA-3 standard output is:

```
a7ffc6f8bf1ed76651c14756a061d662f580ff4de43b49fa82d80a4
b80f8434a
```

Many old Keccak-256 outputs are:

```
c5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad
8045d85a470
```

So we must use terminology correctly and precisely: SHA-3 and Keccak. By the same token, you should validate the library is a real SHA-3 before using it. You may make mistakes based on unsolid assumption. For example, NPM's sha3 library is not conformed to standard SHA-3.

To Ethereum community, Ethereum uses algorithms in SHA-3 family and therefore enjoys the benefit brought by them. But be cautious that there is a little difference between Ethereum protocol and standard FIPS 202 implementations. In Ethereum yellow paper, it says Ethereum use "Keccak-256 hash function (the winner of SHA-3 competition)."

3.10.1.3 Address Related

No	变量	说明
1	<address>.balance (uint256)	Account balance of Address, unit is wei
2	<address>.transfer (uint256 amount)	Send specified amount of ether. Unit is wei and throw exception if error occurs
3	<address>.send (uint256 amount) returns (bool)	Send specified amount of ether. Unit is wei and return false if error occurs
4	<address>.call(...) returns (bool)	Initiate low-level call. Return false if failed
5	<address>.callcode(...) returns (bool)	Initiate low-level callcode. Return false if failed
6	<address>.delegatecall(...) returns (bool)	Delegatecall call, return false if fail

3.10.1.4 Contract Related

Every contract has the following three global functions:

- this: Current contract object, and it can be converted to Address type explicitly.
- selfdestruct: Self-destruction function of contract, it will send ether in contract to the specified address.
- suicide: Equals to selfdestruct.

3.10.2 Special Unit

Solidity uses some specific currency unit and time unit. The following contains the details.

3.10.2.1 Currency Unit

When a digital literal is combined with wei, finney, szabo, or ether, we can convert the number between them. The relationship between different currency units is like below:

1 ether == 10^3 finney == 1000 finney

1 ether == 10^6 szabo

1 ether == 10^18 wei

A little story: Ethereum currency unit is actually the name of cryptographers. Entrepreneurs of Ethereum pay respect to those cryptographers for their contributions in the digital currency area. They are:

wei: Wei Dai, Cryptographer, released B-money.

finney: Hal Finney, Cryptographer, the proposer of Proof of Work mechanism.

szabo: Nick Szabo, Cryptographer, the proposer of smart contract.

Besides the basic Unit—Wei, Ethereum also uses other currency units for user convenience. They are listed as below:

No.	Variables	Descriptions
wei	1 wei	1
Kwei (babbage)	1e3 wei	1,000
Mwei (lovelace)	1e6 wei	1,000,000
Gwei (shannon)	1e9 wei	1,000,000,000
microether (szabo)	1e12 wei	1,000,000,000,000
milliether (finney)	1e15 wei	1,000,000,000,000,000
ether	1e18 wei	1,000,000,000,000,000,000

3.10.2.2 Time Unit

Seconds, minutes, hours, days, weeks, years all can be used as a suffix, and can be converted between each other. Default time unit is seconds. The default rules are as follow:

1 = 1 second

1 minutes = 60 seconds

1 hours = 60 minutes

1 days = 24 hours

1 weeks = 7 days

1 years = 365 days

Please be careful if you want to use these units for calculation purpose since leap year, leap month, and leap day exist.

3.10.3 Type Cast and Inference

```
uint24 x = 0x123;
var y = x;
```

Function parameters including return values cannot use var which does not have an explicit type. Please be aware that type inference is based on the first variable. Following loop “for (var i = 0; i < 2000; i++) {}” will be infinite loop because type of i is inferenced to uint8 and the value i will never cross 2000.

```
pragma solidity ^0.4.4;
contract deadloop{
   function a() returns (uint){
    uint count = 0;
      for (var i = 0; i < 2000; i++) {
       count++;
       if(count >= 2100){
          break;
       }
      }
      return count;
   }
}
```

3.10.3.1 Implicit Conversion

If the operator supports different types on both sides, the compiler will try an implicit conversion. And the assignment is alike. Generally, the implicit conversion needs to ensure that conversion will not lose data and syntactically valid. For example, uint8 can be converted to uint16, uint256. However, int8 cannot be converted to uint256 since −1 is not in range of uint256.

For any unsigned integer, small type values can be converted to larger type value. For example, uint160 can be converted to address.

3.10.3.2 Explicit Conversion

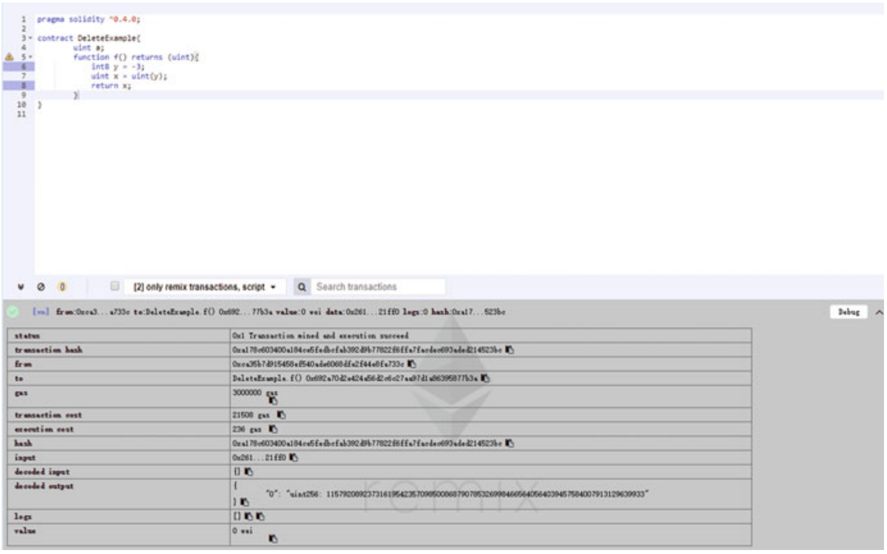
If automatic type conversion is not allowed, but you are absolutely sure that typecast is valid, you can force typecast. Please be cautious that invalid type cast will cause an error. So you must test the conversion comprehensively.

```
pragma solidity ^0.4.0;
contract DeleteExample{
    uint a;
    function f() returns (uint){
            int8 y = -3;
            uint x = uint(y);
            return x;
    }
}
```

From the screenshot below, it can be seen that x become a huge uint256 number after running f(): “uint256: 115792089237316195423570985008687907853269984665640564039457584007913129639933” (Fig. 3.4).

If it is converted to a smaller size type, the upper bit will be trimmed.

```
uint32 a = 0x12345678;
uint16 b = uint16(a); // b is 0x5678
```



```
pragma solidity ^0.4.0;

contract DeleteExample{
        uint a;
        function f() returns (uint){
            int8 y = -3;
            uint x = uint(y);
            return x;
        }
}
```

```
[vm] from:0xca3...a733c to:DeleteExample.f() 0x692...77b3a value:0 wei data:0x261...21ff0 logs:0 hash:0xa17...523bc
```

status	0x1 Transaction mined and execution succeed
transaction hash	0xa178c603400a184ce5fedbcfab392d9b77822f6ffa7facdec693aded214523bc
from	0xca35b7d915458ef540ade6068dfe2f44e8fa733c
to	DeleteExample.f() 0x692a70d2e424a56d2c6c27aa97d1a86395877b3a
gas	3000000 gas
transaction cost	21508 gas
execution cost	236 gas
hash	0xa178c603400a184ce5fedbcfab392d9b77822f6ffa7facdec693aded214523bc
input	0x261...21ff0
decoded input	{}
decoded output	{ "0": "uint256: 115792089237316195423570985008687907853269984665640564039457584007913129639933" }
logs	[]
value	0 wei

Fig. 3.4 Int value explicitly converted to unsigned int

3.10.4 Exception

Raising an exception results in the termination of current execution and rollback of the transaction (Any changed value and account balance change will be reverted). An exception can also be bubbled up through the Solidity call stack. User can trigger an exception through the following methods:

- Use throw.
- Use require, but parameter set to false.

In one word, assert is used to determine if the condition is satisfied and require is used to validate input.

3.10.5 Assembly

In this section, we will introduce inline assembly in detail. Please refer to this gist post[2] for all the details of opcodes. Here are some most-used opcodes:

Function style opcode:mul(1, add(2, 3)) equals to: push1 3 push1 2 add push1 1 mul

Inline local variable:let x := add(2, 3) let y := mload(0x40) x := add(x, y)

Accessible external variable:function f(uint x) { assembly { x := sub(x, 1) } }

Label:let x := 10 repeat: x := sub(x, 1) jumpi(repeat, eq(x, 0))

Loop:for { let i := 0 } lt(i, x) { i := add(i, 1) } { y := mul(2, y) }

switch:switch x case 0 { y := mul(x, 2) } default { y := 0 }

Function call:function f(x) -> y { switch x case 0 { y := 1 } default { y := mul(x, f(sub(x, 1))) } }

Please note that inline assembly is a low-level method to access EVM. And it skips over multiple security mechanism built-in EVM.

Following example show how to access a contract through a library function and save return value into a bytes variable. Some functionality cannot be realized by normal Solidity programming. Inline assembly can be of help in such situations.

```
pragma solidity ^0.4.0;

library GetCode {
   function at(address _addr) returns (bytes o_code) {
      assembly {
         // Get the code size
         let size := extcodesize(_addr)
```

(continued)

[2] https://gist.github.com/hayeah/bd37a123c02fecffbe629bf98a8391df.

```
            // Allocate byte array for output
            // by using o_code = new bytes(size)
            o_code := mload(0x40)
            // new "memory end" including padding
             mstore(0x40, add(o_code, and(add(add(size, 0x20), 0x1f),
not(0x1f))))
            // Load size bytes of data on stack to o_code
            mstore(o_code, size)
            // Retrieve code and only assembly can do this job.
            extcodecopy(_addr, add(o_code, 0x20), 0, size)
        }
    }
}
```

Inline assembly is useful when Solidity code cannot implement in an efficient way. But, you must be cautious that inline assembly programming is difficult and dangerous since the compiler will skip boundary check for inline assembly. So only use it if you are tackling a complex problem and you know what you are doing.

```
pragma solidity ^0.4.0;
library VectorSum {
   // sumSolidity()'s efficiency is pretty low since optimizer can
not
   //skip boundary check for array access
   function sumSolidity(uint[] _data) returns (uint o_sum) {
      for (uint i = 0; i < _data.length; ++i)
         o_sum += _data[i];
   }
// If we are absolutely confident that cross boundary is impossible,
we
// can skip boundary check to increase efficiency
// We need to add 0x20 since the storage 0 will be used to store array
// length
function sumAsm(uint[] _data) returns (uint o_sum) {
      for (uint i = 0; i < _data.length; ++i) {
         assembly {
            o_sum := mload(add(add(_data, 0x20), mul(i, 0x20)))
         }
      }
   }
}
```

In this chapter, we learned all basics of Solidity: keyword, statement, modifier, data location, event, etc. We should be able to develop some small and simple contract in Solidity. Next, we will switch to introduce some popular protocols in use in real world.

Chapter 4
Solidity Advanced Topics

In this chapter, we will discuss some advanced topics about Solidity programming, which includes:

- Some specialities of Ethereum Solidity programming
- Some well-known EIP Protocol: ERC20, ERC721
- Call between smart contract

4.1 Keyword "This"

Just like C++ has keyword "this," and Python has keyword "self." Solidity also has keyword—this

```
pragma solidity ^0.4.21;
contract Impossible {
  function Impossible() public { // constructor
     this.test();
  }
  function test() public pure returns(uint256) {
     return 2;
  }
}
```

This is a keyword. But using it in the constructor can cause a problem: this object is not created yet in the constructor. So, in the example above, this.test() will throw an exception.

G. Zheng et al., *Ethereum Smart Contract Development in Solidity*,
https://doi.org/10.1007/978-981-15-6218-1_4

4.2 ERC20 Interface

ERC20 token is much more well-known now. In this section, we are going to describe the details.

```
pragma solidity ^0.4.0;
contract MyToken {
  address public creator; // Contract creator
  uint256 public totalSupply; // Token total supply
  mapping (address => uint256) public balances; // Account balance
function MyToken() public {
    creator = msg.sender;
    totalSupply = 10000;
    balances[creator] = totalSupply;
}
function balanceOf(address owner) public constant returns
(uint256){
    return balances[ownesr];
 }
function sendTokens(address receiver, uint256 amount) public
returns(bool){
    address owner = msg.sender;
    require(amount > 0);
    require(balances[owner] >= amount);
    balances[owner] -= amount;
    balances[receiver] += amount;
    return true;
  }
}
```

- creator is an address which saves the owner of contract.
- totalSupply is a 256-bit unsigned integer which saves the total amount of token.
- balance is a dictionary mapping from an address to unsigned number, which saves the amount of token in a specific address.

Next, there is a constructor which has the same name as the contract. The constructor is called once only during contract deployment. In the constructor, we set the owner of the contract. It is common that function call needs to know whether the contract owner is the same as the function caller—msg.sender. And at last, the contract defines a total supply of 10,000 tokens and stores it to the creator's account.

Next function balanceOf() returns the balance of a specific address. The constant keyword is used here. Solidity has two types of functions: constants and non-constant.

The non-constant function is used to change state, while the constant function is read-only which means constant functions cannot perform any state-change operation. In actual use, there are two types of constant functions:

- The view keyword means that function cannot perform any state-change operation (equals to constant).

- The pure keyword is stricter than view and constant. It means that pure function cannot read state and cannot perform any state-change operation.

The last function allows us to transfer tokens between different addresses. It is non-constant since the function will change the balance of account. It takes receiver address and number of tokens as input parameters and returns a Boolean value which indicates the status of transaction.

Here are two conditions that must be true before execution of the function.

```
...
require(amount > 0);
require(balances[owner] >= amount);
...
```

Require is a method used for validation. It evaluates a condition and will revert if the condition is not satisfied. So the transfer amount must be larger than 0 and the sender must have enough balance in his account.

At last, we need to subtract the token amount from the owner's balance and add it to the receiver's balance.

```
...
 balances[owner] -= amount;
balances[receiver] += amount;
return true;
}
```

You can find the official definition of ERC20.[1]

4.2.1 Methods

Please note: the caller must process the situation if a call returns false. The caller cannot assume that all functions return successfully.

4.2.1.1 totalSupply

```
function totalSupply() constant returns (uint256 totalSupply)
```

Get the total supply of token.

[1]https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md

4.2.1.2 balanceOf

```
function balanceOf(address _owner) constant returns (uint256
balance)
```

Get account balance of the _owner address.

4.2.1.3 transfer

```
function transfer(address _to, uint256 _value) returns (bool
success)
```

Send _value amount of token to address _to.

4.2.1.4 transferFrom

```
function transferFrom(address _from, address _to, uint256 _value)
returns (bool success)
```

Send _value amount of token from address _from to address _to.

transferFrom method is used to withdraw fund from a contract, allowing sending token at will, such as deposit ether to a contract address or charge in form of tokens. The function will fail unless msg.send gets approval from _from account somehow. We suggest to use it after getting approval.

4.2.1.5 approve

```
function approve(address _spender, uint256 _value) returns (bool
success)
```

This function allows _spender to withdraw _value amount of token from your account. If the function is used multiple times, the latter call will reset the quota of the previous call.

4.2.1.6 allowance

```
function allowance(address _owner, address _spender) constant
returns
(uint256 remaining)
```

Return amount of token which _spender can send from _owner's account.

4.2.2 Events

Transfer

```
event Transfer(address indexed _from, address indexed _to,
uint256 _value)
```

Triggered when token is transferred.
Approval

```
event Approval(address indexed _owner, address indexed _spender,
uint256
_value)
```

Triggered when approve (address _spender, uint256 _value) is called.

4.2.3 OpenZeppline

OpenZeppline is an open-source framework providing re-usable smart contract templates to develop Dapp, protocol, and DAO. Using standard, fully tested code under intensive community review will lower the risk caused by vulnerabilities.

```
$ npm install zeppelin-solidity
```

Now, let us have a look at how to implement ERC20 token with OpenZeppelin library.

```
import 'zeppelin-solidity/contracts/token/BasicToken.sol';
import 'zeppelin-solidity/contracts/ownership/Ownable.sol';
contract MyToken is BasicToken, Ownable {
   uint256 public constant INITIAL_SUPPLY = 10000;
   function MyToken() {
    totalSupply = INITIAL_SUPPLY;
    balances[msg.sender] = INITIAL_SUPPLY;
   }
}
```

As can be seen, some core implementations are removed. Actually, they are not. We only let OpenZeppeline handle these core functionalities. It is meaningful to reuse secure design since risk being attacked will be largely reduced.

Our contract inherits from two OpenZeppeline pre-defined contracts: Ownable and BasicToken. Solidity supports multi-inheritance, and please note that the inheritance order of base contract is very important.

Our MyToken is derived from Ownable. Let us have a look of the contract:

```
OpenZeppelin Ownable contract
```

Ownable provides three functions:

- It defines a special address type variable—"owner."
- It allows us to switch the ownership of a contract.
- It provides a useful onlyOwner modifier, and this modifier ensures that a function can be only called by owner.

Besides, Mytoken inherits from BasicToken contract.

```
OpenZeppelin BasicToken contract
```

The implementation is very similar to MyToken with only a few differences: it uses sendTokens instead of transfer, and it implements all the functionalities of transfer except Transfer event.

Another important point is SafeMath. OpenZeppelin suggests using SafeMath library for mathematics operations with security check. SafeMath is most frequently used since it ensures that math operations will not cause overflow/underflow.

4.3 ERC721 Interface

In 2017, Cryptokitties game showcases how non-fungible assets are created and how to trade them on Ethereum. In the Game, player feeds and trades cat. But the most important is: all cats are on blockchain and only their owner can feed and trade themselves. In traditional game, data are stored on a centralized server and can be modified if necessary by game owner. Contrarily, in Cryptokitties game, the player is playing in a trustless decentralized network. Their ownership of cat can be proved undoubtedly: if you own a cat, blockchain can prove that the cat is yours, not others.

4.3.1 ERC721 Protocol

ERC721 standard includes four interfaces: one is ERC721, another is the standard receiving ERC721 token, the other two are optional extensions. Each ERC-721 standard contract must implement ERC721[2] and ERC165 interface.[3]

Generally, ERC721 tokens standard must solve the following problems:

- Ownership—How to process the ownership of token contract?
- Creation—How to create token contract?
- Transfer & Authorization—How to transfer token and how we approve other addresses (contract or EOA) to be able to transfer our tokens?
- Destroy—How to destroy token?

The following examples all follow OpenZeppelin's ERC721 token standard. Please refer to ERC721token.sol[4] for details

[2]https://github.com/ethereum/EIPs/blob/master/EIPS/eip-721.md

[3]https://github.com/ethereum/EIPs/blob/master/EIPS/eip-165.md

[4]https://github.com/OpenZeppelin/zeppelin-solidity/blob/master/contracts/token/ERC721/ERC721Token.sol

4.3.1.1 Token Ownership

ERC20 now becomes a standard and it is easy to understand. From the ownership point of view, ERC20 has a mapping variable which maps token balance to an address:

```
mapping(address => uint256) balances
```

If a user buys ERC20 token, the ownership of token can be verified by contract since contract keeps a record about how many tokens an address holds. If a user wants to transfer ERC20 tokens, the contract will validate the transaction through checking balances mapping: a transaction is valid only when user account holds enough tokens. Since balance mapping is initialized to 0, if a user never interacts with the token contract, his account balance will be 0.

As stated above, ERC721 is a non-fungible token (NFT). This is to say, Tokens of the same class or contract have different IDs. So, not like ERC20 mapping balance to an address, ERC721 token has its own unique ID which is used to identify the ownership. For this reason, in the ERC721 standard, ownership is determined by an array of token indexes or IDs that are mapped to users' address. Since each token ID is unique, we have to look at each individual token created by the contract. The main contract keeps a list of all the ERC721 tokens created in that contract in an array, so each token has its respective index within the context of all ERC721 tokens available from that particular contract via the allTokens array.

```
uint256[] internal allTokens
```

In addition to the array of token indexes in the entire contract, each individual address has an array of token indexes or IDs that are mapped to their address so that developers know which tokens a user owns.

```
mapping (address => uint256[]) internal ownedTokens
```

This simple difference spurs many additional requirements of an ERC721 token. With an ERC20 token, we check against a balance, but now, we need to check ownership against a specific index of a token. And we need to rearrange array if necessary

In order to verify ownership of a certain token ID, we map each token index or ID to an owner. In this way, every time we can check the address which a token ID maps to.

```
mapping (uint256 => address) internal tokenOwner
```

Since we already have ownedTokens, why we need tokenOwner? The reason is the array in Solidity programming: when we delete an element in an array, the element is not deleted but is replaced with a zero. For example, assume that we have an array myarray = [2 7 12], which is of length 3. Then we call a function that says to delete myarray[myarray.length.sub(1)]. Although we may expect that myarray = [2 7], we actually have the following array myarray = [2 7 0], and it would still be of length 3. There are cases in which we would like to delete or remove a token from the ownership of an address. In addition to delete tokens from owner array, we also need to rearrang the array. We will see later on how this information comes into play when we transfer (removing ownership) and burntokens. The ownedTokensIndex maps each token ID to its respective index in its owner's array. As stated below, we also map the token ID to its index in the allTokens array as well.

```
// Mapping from token ID to index of the owner tokens list
mapping(uint256 => uint256) internal ownedTokensIndex;
//Mapping from token id to position in the allTokens array
mapping(uint256 => uint256) internal allTokensIndex;
```

We also need to keep track of *how many* ERC721 tokens that a user or address actually owns. ownedTokensCount is introduced for this purpose. This ownedTokensCount is updated as we purchase, transfer, or potentially burn tokens accordingly.

```
mapping (address => uint256) internal ownedTokensCount
```

4.3.1.2 Token Creation

In the ERC20 design, we have a value that maintains total available token supply, totalSupply_. Also, ERC20 contract has a mapping variable which keeps a balance of tokens. In below, MyToken is used to set the value of the totalSupply_ of tokens. ERC20 standard was expanded to include a mint function in which a desired number of tokens are added to the totalSupply_ and balances are mapped accordingly. It can be seen that balances variable is updated in mint function.

```
   uint256 totalSupply_
   //Example of setting token supply via constructor
contract MyToken {
   function MyToken(uint _setSupply)
       { totalSupply_ = _setSupply_ }
...
   //Example of maintaining a variable token supply via minting
function mint(address _to, uint256 _amount)
onlyOwner
canMint
public
returns (bool)
{
   totalSupply_ = totalSupply_.add(_amount);
   balances[_to] = balances[_to].add(_amount);
   Mint(_to, _amount);
   Transfer(address(0), _to, _amount);
   return true;
}
```

As to ERC721's, we have learned that since each individual token is unique, we must create each individual token. Because we maintain an array of tokens in an ERC721 standard, we need to add each token to that array separately.

Here addTokenTo() uses super.addTokenTo() to first call the addTokenTo() function in basic ERC721 contract. Essentially, over the course of these two functions, all global ownership variables are updated. The functions take two parameters: _to or the address to which the token will be owned and _tokenId or the unique ID of the token. And caller is limited to the owner of the contract. In this case, the user can choose any unique number ID. First, in the ERC721BasicToken contract, we check that the token ID is not already owned. Then we set the token owner of the requested token ID, and add one to the number of owned tokens of that individual account. Going back to the full implementation contract, we also update the array of the new owner's (_to) tokens by adding this new token to the end of their ownedTokens array and saving the index of that new token.

```
// call this function in ERC721Token.sol
 function addTokenTo(address _to, uint256 _tokenId) internal {
   super.addTokenTo(_to, _tokenId);
   uint256 length = ownedTokens[_to].length;
   ownedTokens[_to].push(_tokenId);
   ownedTokensIndex[_tokenId] = length;
 }
// Call this function in ERC721BasicToken.sol
 function addTokenTo(address _to, uint256 _tokenId) internal {
   require(tokenOwner[_tokenId] == address(0));
   tokenOwner[_tokenId] = _to;
   ownedTokensCount[_to] = ownedTokensCount[_to].add(1);
 }
```

From the above, we can see that addTokenTo() updates an address to an individual. It can be seen below that when we call _mint() from our full implementation ERC721 contract, we jump up to our basic implementation, which ensures that we are not minting to an address of zero and calls addTokenTo(), which will actually call back to our full implementation contract to kick off the addTokenTo() calls. After the _mint() function in the basic contract is completed, back in our full implementation, we add the _tokenId to our allTokensIndex's mapping as well as our allTokens array.

```
function _mint(address _to, uint256 _tokenId) internal {
    super._mint(_to, _tokenId);
    allTokensIndex[_tokenId] = allTokens.length;
    allTokens.push(_tokenId);
 }
function _mint(address _to, uint256 _tokenId) internal {
    require(_to != address(0));
    addTokenTo(_to, _tokenId);
    Transfer(address(0), _to, _tokenId);
 }
```

We have created tokens and token IDs, but they are not holding any data yet. Example below shows how to give a string to URI data by looking up a mapping from a token ID to a string.

```
// Optional mapping for token URIs
mapping(uint256 => string) internal tokenURIs;
```

The following _setTokenURI() uses the token ID created via _mint() and desired URI information, to set the data that is mapped to a token ID in tokenURIs. Please note that we must be sure that a token ID exists prior to assigning data.

```
function _setTokenURI(uint256 _tokenId, string _uri) internal {
    require(exists(_tokenId));
    tokenURIs[_tokenId] = _uri;
}
function exists(uint256 _tokenId) public view returns (bool) {
    address owner = tokenOwner[_tokenId];
    return owner != address(0);
}
```

4.3.1.3 Transfer & Allowance

We could use transfer() to transfer ERC20 token. In transfer(), we specify a target address and amount of token. And then adjust the account balance in the contract.

```
function transfer(address _to, uint256 _value) public returns
(bool)
{
   require(_to != address(0));
   require(_value <= balances[msg.sender]);
   balances[msg.sender] = balances[msg.sender].sub(_value);
   balances[_to] = balances[_to].add(_value);
   Transfer(msg.sender, _to, _value);
   return true;
}
```

Allowance means that we can grant approval to another contract or address to be able to transfer our ERC20 tokens. And this requirement is common in distributed applications, such as escrows, games, auctions, etc. Hence, we need a way to approve other address to spend our tokens. Also, another version of the transfer function requires the contract to check the allowance.

In the ERC20 standard, we have a global variable "*allowed*" in which we keep the mapping from an owners address to an approved spender's address and then to the amount of tokens. Calling approve() function can add an approval to its desired _spender and _value. The amount of token is not checked here and it will be checked in transfer().

```
//Global variable
mapping (address => mapping (address => uint256)) internal allowed
//Allowance of another address to spend your tokens
function approve(address _spender, uint256 _value)
public
returns (bool)
{
   allowed[msg.sender][_spender] = _value;
   Approval(msg.sender, _spender, _value);
   return true;
}
```

Once the approval is granted, approved spender can use transferFrom() to transfer tokens. _from is the owner address and _to is receiver's address and _value is the required number of tokens to be sent. First, we check if the owner actually possess the required number of tokens.

```
require(_value ≤ balances[_from])
```

And then we check if msg.sender gets approval: Check global variable—allowed, and then update balances and allowed amount accordingly. There are two functions increaseApproval() and decreaseApproval() to increase/decrease the quota.

```
function transferFrom(address _from, address _to, uint256 _value)
public
  returns (bool) {
    require(_to != address(0));
    require(_value <= balances[_from]);
    require(_value <= allowed[_from][msg.sender]);
    balances[_from] = balances[_from].sub(_value);
    balances[_to] = balances[_to].add(_value);
    allowed[_from][msg.sender] = allowed[_from][msg.sender].
sub(_value);
    Transfer(_from, _to, _value);
    return true;
}
```

Since each ERC721 token is unique, we need to approve an address for transferring by token ID and to transfer token ID. tokenApprovals is a global variable which maps a token ID or index to an address. If approved, the address is able to transfer a specific token by its ID.

```
mapping (uint256 => address) internal tokenApprovals;
function getApproved(uint256 _tokenId) public view returns
(address) {
   return tokenApprovals[_tokenId];
 }
```

operatorApprovals is another global variable which maps the owner's address to an approved spender address and then maps to a Boolean variable. If the Boolean variable is set to true, this means that the owner allows this address to process all ERC721 tokens owned. We could use setApprovalForAll() to authorize an address to handle all tokens.

```
mapping  (address  =>  mapping  (address  =>  bool))  internal
operatorApprovals;
function isApprovedForAll(address _owner, address _operator)
public view returns (bool) {
   return operatorApprovals[_owner][_operator];
  }
function setApprovalForAll(address _to, bool _approved) public {
   require(_to != msg.sender);
   operatorApprovals[msg.sender][_to] = _approved;
   ApprovalForAll(msg.sender, _to, _approved);
  }
```

In approve() function, we check the ownership or check if msg.sender is approved to handle all tokens. Then we call getApproved() to check if tokens' owner and spender are not address(0). As the last step, we set address _to to tokenapprovals [_tokenId], which means that address _to gets approval to process ERC721 token with _tokenId.

```
function approve(address _to, uint256 _tokenId) public {
   address owner = ownerOf(_tokenId);
   require(_to != owner);
   require(msg.sender == owner || isApprovedForAll(owner, msg.
sender));
   if (getApproved(_tokenId) != address(0) || _to != address(0)) {
    tokenApprovals[_tokenId] = _to;
    Approval(owner, _to, _tokenId);
   }
 }
```

Now, we are going to describe how to actually transfer ERC721. The full implementation of transferFrom() doing the following:

1. The sender and receiver addresses are specified along with the _tokenId to transfer.
2. Using canTransfer() modifier to ensure that the msg.sender is approved to transfer the token or owns it.
3. The clearApproval() function is used to remove the approval of the transfer from the original owner of the token, so that a previously approved spender may not continue to transfer the token.
4. removeTokenFrom() is called in the ERC721 full implementation contract. The token is removed from the ownedTokensCount mapping, the tokenOwner mapping.
5. One more twist is that we move the last token in the owner's ownedTokens array to the index of the token that is being transferred and shorten the array by one.
6. Lastly, we use the addTokenTo() function to add this token index to its new owner.

```
modifier canTransfer(uint256 _tokenId) {
   require(isApprovedOrOwner(msg.sender, _tokenId));
   _;
}
function isApprovedOrOwner(address _spender, uint256 _tokenId)
internal view returns (bool) {
   address owner = ownerOf(_tokenId);
    return _spender == owner || getApproved(_tokenId) == _spender ||
isApprovedForAll(owner, _spender);
}
function transferFrom(address _from, address _to, uint256
```

(continued)

```
_tokenId) public canTransfer(_tokenId) {
   require(_from != address(0));
   require(_to != address(0));
   clearApproval(_from, _tokenId);
   removeTokenFrom(_from, _tokenId);
   addTokenTo(_to, _tokenId);
   Transfer(_from, _to, _tokenId);
}
function clearApproval(address _owner, uint256 _tokenId)
internal {
   require(ownerOf(_tokenId) == _owner);
   if (tokenApprovals[_tokenId] != address(0)) {
    tokenApprovals[_tokenId] = address(0);
    Approval(_owner, address(0), _tokenId);
   }
}
//Full ERC721 implementation
function removeTokenFrom(address _from, uint256 _tokenId)
internal {
   super.removeTokenFrom(_from, _tokenId);
   uint256 tokenIndex = ownedTokensIndex[_tokenId];
   uint256 lastTokenIndex = ownedTokens[_from].length.sub(1);
   uint256 lastToken = ownedTokens[_from][lastTokenIndex];
   ownedTokens[_from][tokenIndex] = lastToken;
   ownedTokens[_from][lastTokenIndex] = 0;
   ownedTokens[_from].length--;
   ownedTokensIndex[_tokenId] = 0;
   ownedTokensIndex[lastToken] = tokenIndex;
}
//Basic ERC721 implementation
function removeTokenFrom(address _from, uint256 _tokenId)
internal {
   require(ownerOf(_tokenId) == _from);
   ownedTokensCount[_from] = ownedTokensCount[_from].sub(1);
   tokenOwner[_tokenId] = address(0);
}
```

How do we ensure that we are sending our ERC721 to a contract that can handle additional transfers? We know an externally owned account (EOA) can use our ERC721 full implementation contract to trade tokens if desired; however, if we send our token to a contract that does not have the appropriate functions to trade and transfer the token via our original ERC721 contract, then the token will be lost. ERC223 proposal, which is a proposed modification to ERC20 aims to prevent such erroneous transfers.

In order to avoid issues, the ERC721 full implementation standard introduces the safeTransferFrom() function. Let us first have a look at ERC721Holder.sol contract. The ERC721Holder.sol contract is to be part of the wallet, auction, or broker contract to hold an ERC721 token. The reason this has been standardized goes back to EIP165 in which the goal is to create "a standard method to publish and detect what interfaces a smart contract implements."

How do we detect an interface? We will see a "magic value," ERC721_RECEIVED, which is the function signature of the onERC721Received () function. It is computed by bytes4(keccak256("onERC721Received(address, uint256, bytes)")). When a contract is called, the EVM uses a series of switch cases to find the function signature that matches the call and executes code accordingly. Consequently, in our ERCHolder contract, we see that the onERCReceived() function signature will match the ERC721_RECEIVED variable in our ERC721Receiver interface.

```
contract ERC721Receiver {
  /**
  * @dev if receive NFT, then return magic value which equals to
  *`bytes4(keccak256("onERC721Received(address,uint256,
bytes)"))`,
  * or Get it from `ERC721Receiver(0).onERC721Received.selector`
  */
  bytes4 constant ERC721_RECEIVED = 0xf0b9e5ba;

  /**
  * @notice
  * @dev ERC721 contract call this function after `safetransfer`
  * This function may raise exception which leads to rollback and
  * transfer rejected.
  * This function uses about 50,000 gas.
  * Must revert if magic value is not returned
  * Note: Contract address is msg.sender
  * @param _from from address
  * @param _tokenId NFT ID transferred
  * @param _data extra data with no specified format
  * @return
`bytes4(keccak256("onERC721Received(address,uint256,bytes)"))`
  */
  function onERC721Received(address _from, uint256 _tokenId,
bytes _data) public returns(bytes4);
}
contract ERC721Holder is ERC721Receiver {
  function onERC721Received(address, uint256, bytes) public
returns(bytes4) {
     return ERC721_RECEIVED;
  }
}
```

ERC721Holder contract above is not a complete contract for handling ERC721 tokens. This template is meant to provide developers with a standardized interface to verify that the ERC721Receiver standard interface is used. Developers will need to extend or inherit the ERC721Holder contract to include functions in their wallet or auction contract to handle ERC721's. Even to hold tokens in escrow developers would need to add functionality so that this holder contract could make a call to transfer tokens out of the contract as needed.

```
//Option 1
function safeTransferFrom(address _from, address _to, uint256
_tokenId) public canTransfer(_tokenId) {
   safeTransferFrom(_from, _to, _tokenId, ""); }

//Option 2
function safeTransferFrom(address _from, address _to, uint256
_tokenId, bytes _data) public canTransfer(_tokenId) {
   transferFrom(_from, _to, _tokenId);
   require(checkAndCallSafeTransfer(_from, _to, _tokenId,
_data)); }

function checkAndCallSafeTransfer(address _from, address _to,
uint256 _tokenId, bytes _data) internal returns (bool) {
   if (!_to.isContract()) {
    return true;
   }
   bytes4 retval = ERC721Receiver(_to).onERC721Received(_from,
_tokenId, _data);
   return (retval == ERC721_RECEIVED); }
}

// check if an address is a contract
function isContract(address addr) internal view returns (bool) {
   uint256 size;
   assembly { size := extcodesize(addr) }
   return size > 0;
  }
```

There are two choices to check how safeTransferFrom() works:

1. Option 1

   Call safeTransferFrom() function with no additional data

2. Option 2

   Call safeTransferFrom() function with data in the form of bytes_data.

   We use the function isContract() above to see if an address is an actual contract

   As before the transferFrom() function is used to remove token ownership from the _from address and add token ownership to the _to address. However, we have an additional requirement that we run the checkAndCallSafeTransfer() function. First, we check that the _to address is an actual contract through the use of the AddressUtils.sol library. After verifying that _to is a contract address, we check that calling the onERC721Received() function returns the same function signature that we are expecting from our standard interface. If the correct value is not returned, then the transferFrom() function is rolled back since the _to does not implement the expected interface.

4.3.1.4 Burning

ERC20 token contract only manages a single mapped balance, so burning tokens against a specific address only need to adjust respective account balance and totalSupply_ of tokens as well. This is exactly what burn() does below: _value is the number of tokens to burn and msg.sender is the owner of the address.

```
function burn(uint256 _value) public {
   require(_value <= balances[msg.sender]);
   address burner = msg.sender;
   balances[burner] = balances[burner].sub(_value);
   totalSupply_ = totalSupply_.sub(_value);
   Burn(burner, _value);
   Transfer(burner, address(0), _value);
}
```

For ERC721 tokens, we need to ensure that the specific token ID or index is eliminated. _burn()function uses super to call a function in basic ERC721 implementation.

1. Call ClearApproval().
2. Remove the token from ownership via removeTokenFrom().
3. Emit Transfer event to alert this change on the frontend.
4. Eliminate the metadata associated with that token by deleting what is mapped to that particular token index.
5. Rearrange allTokens array so that the _tokenId index is replaced with the last token in the array.

```
function _burn(address _owner, uint256 _tokenId) internal {
   super._burn(_owner, _tokenId);
   // 清除metadata (if any)
   if (bytes(tokenURIs[_tokenId]).length != 0) {
    delete tokenURIs[_tokenId];
   }
   // re-org all tokens array
   uint256 tokenIndex = allTokensIndex[_tokenId];
   uint256 lastTokenIndex = allTokens.length.sub(1);
   uint256 lastToken = allTokens[lastTokenIndex];
   allTokens[tokenIndex] = lastToken;
   allTokens[lastTokenIndex] = 0;
   allTokens.length--;
```

(continued)

```
      allTokensIndex[_tokenId] = 0;
      allTokensIndex[lastToken] = tokenIndex;
    }
  }
  function _burn(address _owner, uint256 _tokenId) internal {
      clearApproval(_owner, _tokenId);
      removeTokenFrom(_owner, _tokenId);
      Transfer(_owner, address(0), _tokenId);
  }
```

There are many ERC721 examples now, such as: Cryptokitties, Cryptogs, Cryptocelebrities, Decentraland. There are also a lot of examples of digital asset and collectibles on OpenSea.

4.3.1.5 Wallet Interface

Wallet app should implement the wallet interface[5]. A valid ERC721TokenReceiver needs to implement the function below:

```
function onERC721Received(address _operator, address _from,
uint256
_tokenId, bytes _data) external returns(bytes4);
```

And return:

```
bytes4(keccak256("onERC721Received(address,address,uint256,
bytes)"))
```

An invalid receiver does not implement the function above, neither return other content:

```
contract ValidReceiver is ERC721TokenReceiver {
   function onERC721Received(address _operator, address _from,
uint256 _tokenId, bytes _data) external returns(bytes4){
       return
bytes4(keccak256("onERC721Received(address,address,uint256,
bytes)"));
   }
}
```

[5]https://github.com/ethereum/EIPs/blob/master/EIPS/eip-721.md

and

```
contract InvalidReceiver is ERC721TokenReceiver {
   function onERC721Received(address _operator, address _from,
uint256 _tokenId, bytes _data) external returns(bytes4){
      return bytes4(keccak256("some invalid return data"));
   }
}
```

4.3.2 Metadata

Metadata extension gives the token contract a name and symbol, and took some extra data to make token unique. The enumerable extension enables to sort token much easier by attributes than by tokenID. Metadata extension is optional. Metadata interface allows the contract to retrieve metadata of non-fungible token (NFT): such as name and other detail information. Please refer to ERC721 standard for details of ERC721Metadata. Here is the "ERC721 Metadata JSON Schema":

```
{
 "title": "Asset Metadata",
 "type": "object",
 "properties": {
   "name": {
      "type": "string",
      "description": "Identifies the asset to which this NFT
represents",
   },
   "description": {
      "type": "string",
      "description": "Describes the asset to which this NFT
represents",
   },
   "image": {
      "type": "string",
       "description": "A URI pointing to a resource with mime type
image/* representing the asset to which this NFT represents.
Consider making any images at a width between 320 and 1080 pixels and
aspect ratio between 1.91:1 and 4:5 inclusive.",
   }
  }
}
```

The contract above inherits from TokenERC721.sol and ERC721Metadata extension.

```
contract TokenERC721Metadata is TokenERC721, ERC721Metadata {
```

Metadata extension has the following three functions:

```
function name() external view returns (string _name);
function symbol() external view returns (string _symbol);
function tokenURI(uint256 _tokenId) external view returns
(string);
```

Constructor:

```
constructor(uint _initialSupply, string _name, string _symbol,
string
_uriBase)
public TokenERC721(_initialSupply){
   __name = _name;
   __symbol = _symbol;
   __uriBase = bytes(_uriBase);

   //Add to ERC165 Interface Check
   supportedInterfaces[
      this.name.selector ^
      this.symbol.selector ^
      this.tokenURI.selector
   ] = true;
}
function name() external view returns (string _name){
   _name = __name;
}

function symbol() external view returns (string _symbol){
   _symbol = __symbol;
}
function tokenURI(uint256 _tokenId) external view returns
(string){
   require(isValidToken(_tokenId));
   uint maxLength = 78;
   bytes memory reversed = new bytes(maxLength);
   uint i = 0;
   // loop and add byte into array
   while (_tokenId != 0) {
      uint remainder = _tokenId % 10;
      _tokenId /= 10;
      reversed[i++] = byte(48 + remainder);
   }
   // Allocate memory for array
   bytes memory s = new bytes(__uriBase.length + i);
   uint j;
   // Append basic part to array
```

(continued)

```
    for (j = 0; j < __uriBase.length; j++) {
      s[j] = __uriBase[j];
    }
    // Append tokenId to array
    for (j = 0; j < i; j++) {
      s[j + __uriBase.length] = reversed[i - 1 - j];
    }
    return string(s);
}
```

4.3.3 *Enumerable Extension*

4.3.3.1 Interface

Enumerable extension is optional. It makes smart contract can publish all NFT list and make your NFT discoverable from outside. The extension provides three functions in which the latter two functions make us retrieve token through index. ERC721 optional enumerable extension standard is defined in https://github.com/ethereum/EIPs/blob/master/EIPS/eip-721.md.

4.3.3.2 Instance

Our TokenERC721Enumerable contract is inherited from TokenERC721.sol and implemented ERC721Enumerable extension interface.

```
contract TokenERC721Enumerable is TokenERC721, ERC721Enumerable {
uint[] internal tokenIndexes;
```

If the contract contains a self-destruct program, we must add a mapping to record the mapping from index to token. So, we add the following code:

```
mapping(uint => uint) internal indexTokens;
```

If the contract does not contain burn functionality, please ignore indexTokens below.

Another list is: given a specific address, we need a list recording all tokens related to this address. tokenOfOwnerByIndex function will use this list. There should be at least one token for each address which is implemented by two mapping variables:

```
mapping(address => uint[]) internal ownerTokenIndexes;
mapping(uint => uint) internal tokenTokenIndexes;
```

The first Mapping maps address to an array which records all token ID owned by an array element. The second Mapping maps each TokenID to its position of ownerTokenIndexes.

```
ownerTokenIndexes[ownerAddress][tokenIndex] = tokenId;
tokenTokenIndexes[tokenId] = tokenIndex;
```

Next is to create a constructor. We need to add interface into supportedInterfaces and need to pass initialSupply to *TokenERC721*'s constructor.

```
constructor(uint    _initialSupply)    public    TokenERC721
(_initialSupply){
   for(uint i = 0; i < _initialSupply; i++){
      tokenTokenIndexes[i+1] = i;
      ownerTokenIndexes[creator].push(i+1);
      tokenIndexes.push(i+1);
      indexTokens[i + 1] = i;
   }
   //Add to ERC165 Interface Check
   supportedInterfaces[
      this.totalSupply.selector ^
      this.tokenByIndex.selector ^
      this.tokenOfOwnerByIndex.selector
   ] = true;
}
```

4.3.3.3 totalSupply

Function totalSupply is very simple. It simply returns the length of tokenIndexes. Since all token is saved in array—tokenIndexes, the length of the array is the number of tokens.

```
function totalSupply() external view returns (uint256){
   return tokenIndexes.length;
}
```

4.3.3.4 tokenByIndex

tokenByIndex is simple too. Index keeps the position of tokens in an array. We need to check the range of tokenByindex before using it.

```
function tokenByIndex(uint256 _index) external view returns
(uint256){
   require(_index < tokenIndexes.length);
   return tokenIndexes[_index];
}
```

4.3.3.5 tokenOfOwnerByIndex

tokenOfOwnerByIndex is similar as tokenByIndex. It checks the index value and returns the position of tokens in ownerTokenIndexes array.

```
function tokenOfOwnerByIndex(address _owner, uint256 _index)
external view
returns (uint256){
   require(_index < balances[_owner]);
   return ownerTokenIndexes[_owner][_index];
}
```

4.3.3.6 transferFrom

```
function transferFrom(address _from, address _to, uint256
_tokenId) public {
   address owner = ownerOf(_tokenId);
   require ( owner == msg.sender
      || allowance[_tokenId] == msg.sender
      || authorised[owner][msg.sender]
   );
   require(owner == _from);
   require(_to != 0x0);
   emit Transfer(_from, _to, _tokenId);
   owners[_tokenId] = _to;
   balances[_from]--;
   balances[_to]++;
   if(allowance[_tokenId] != 0x0){
   delete allowance[_tokenId];
   }
   //=== Enumerable Additions ===
```

(continued)

```
    uint oldIndex = tokenTokenIndexes[_tokenId];
    //If the token isn't the last one in the owner's index
    if(oldIndex != ownerTokenIndexes[_from].length - 1){
    //Move the old one in the index list
      ownerTokenIndexes[_from][oldIndex] =
ownerTokenIndexes[_from][ownerTokenIndexes[_from].length - 1];
    //Update the token's reference to its place in the index list
      tokenTokenIndexes[ownerTokenIndexes[_from][oldIndex]] =
oldIndex;
    }
    ownerTokenIndexes[_from].length--;
    tokenTokenIndexes[_tokenId] = ownerTokenIndexes[_to].length;
    ownerTokenIndexes[_to].push(_tokenId);
}
```

Please note: before adding token into _to mapping variable, the developers must reduce ownerTokenIndexes[_from].length. If developer does not do so, it is very likely that tokenTokenIndexes[_tokenId] is set to an incorrect value.

4.3.3.7 burnToken

burnToken's program is almost the same as that of transferFrom. The only difference is that we need to remove the token from tokenIndexes when we add it to ownerTokenIndexes[_to].

```
function burnToken(uint256 _tokenId) external{
    address owner = ownerOf(_tokenId);
    require ( owner == msg.sender
        || allowance[_tokenId] == msg.sender
        || authorised[owner][msg.sender]
    );
    burned[_tokenId] = true;
    balances[owner]--;
    emit Transfer(owner, 0x0, _tokenId);
     //=== Enumerable Additions ===
    uint oldIndex = tokenTokenIndexes[_tokenId];
     if(oldIndex != ownerTokenIndexes[owner].length - 1){
       // Move last token to old position
     ownerTokenIndexes[owner][oldIndex] =
ownerTokenIndexes[owner][ownerTokenIndexes[owner].length - 1];
       // change token's reference to new position
     tokenTokenIndexes[ownerTokenIndexes[owner][oldIndex]]
= oldIndex;
 }
     ownerTokenIndexes[owner].length--;
```

(continued)

```
    delete tokenTokenIndexes[_tokenId];
    //This part deals with tokenIndexes
    oldIndex = indexTokens[_tokenId];
    if(oldIndex != tokenIndexes.length - 1){
       // Move last token to old position
      tokenIndexes[oldIndex] = tokenIndexes[tokenIndexes.length -
1];
   }
   tokenIndexes.length--;
}
```

4.3.3.8 issueTokens

Now let us discuss issueTokens function. No matter what token issue logic does, the developer must follow the standard workflow.

```
uint newId = maxId.add(1); //Using SafeMath to prevent overflows.
    // Append index of new token to ownerTokenIndexes
tokenTokenIndexes[newId] = ownerTokenIndexes[msg.sender].
length;
    // Append tokenId to ownerTokenIndexes
ownerTokenIndexes[creator].push(newId);
    // Append token to tokenIndexes
indexTokens[thisId] = tokenIndexes.length;
tokenIndexes.push(thisId);
```

For newId = maxId + 1, please make sure assigning newId after all new coin creation process is over:

```
function issueTokens(uint256 _extraTokens) public{
    require(msg.sender == creator);
    balances[msg.sender] = balances[msg.sender].add
(_extraTokens);
    uint thisId; //We'll reuse this for each iteration below.
    for(uint i = 0; i < _extraTokens; i++){
        thisId = maxId.add(i).add(1); //SafeMath to be safe!
        // Append index of new token to ownerTokenIndexes
        tokenTokenIndexes[thisId] = ownerTokenIndexes[creator].
length;
        // Append tokenId to ownerTokenIndexes
        ownerTokenIndexes[creator].push(thisId);
        // Append token to tokenIndexes
```

(continued)

```
        indexTokens[thisId] = tokenIndexes.length;
        tokenIndexes.push(thisId);
        emit Transfer(0x0, creator, thisId);
    }
    maxId = maxId.add(_extraTokens);
}
```

4.3.4 ERC165 Protocol

ERC165 has only one function used to check if the contract signature meets the specified interface signature.

```
interface ERC165 {
  /// @notice Query whether a contract implements some interface or
not
  /// @param interfaceID Interface ID specified by ERC-165 standard
  /// @dev Interface ID is defined in ERC-165 standard.
  /// This function use GAS less than 30,000 gas.
  /// @return If contract implements the specified interface, then
return
/// `true`
  /// and `interfaceID` is not 0xffffffff, or return `false`
  function supportsInterface(bytes4 interfaceID) external view
returns (bool);
}
```

If specified interface ID (byte4 is supported), return True. In ERC165 standard, Interface ID is defined as "the XOR of all function selectors in the interface." Now, let us use balanceOf function as an example. It is defined as below:

```
function balanceOf(address _owner) external view returns
(uint256){
 //...
};
```

There are two ways to get function selectors, one is as below:

```
this.balanceOf.selector
```

And the other one is:

```
bytes4(keccak256("balanceOf(address)"))
```

Both return 0x70a08231. The first one seems neat, but we occasionally will use the second one to get the function selector when contract overloads the function. It can be clearly seen that function selector does not care about parameter name, modifier, mutability, returns, and the function body. It only relates to function name and parameter types.

Assume that interface consists of three functions: function1(), function2(), and function3(), then interfaceID is:

```
interfaceID = this.function1.selector ^ this.function2.selector ^
              this.function3.selector;
```

We must implement ERC165 if we want to implement ERC721. Here, we are going to demo how to implement ERC165. Since we will use Solidity, we need to minimize the GAS usage since unnecessary calculation will cost resource and ether. ERC165 standard requires supportsInterface using <30,000 gas. So, we do not need to calculate InterfaceID everytime when supportsInterface is called. We can save interfaceIDs supported into a mapping variable.

Then let us start to code our contract CheckERC165:

```
contract CheckERC165 is ERC165 {
    mapping (bytes4 => bool) internal supportedInterfaces;
```

supportsInterface function will return a value from mapping variable. Here is the code:

```
function supportsInterface(bytes4 interfaceID) external
  view returns (bool){
    return supportedInterfaces[interfaceID];
}
```

The following code shows that we save interfaceID into Mapping in constructor:

```
constructor() public {
 supportedInterfaces[this.supportsInterface.selector] = true;
}
```

Any call to supportsInterface with ERC165 standard interfaceID will return true.

4.4 Call Between Contracts

There are several methods to call between contracts. A deployed contract has an address and this address object provides three methods to call other contracts:

- call—Run code in other contracts
- delegatecall—Using storage of calling contract and run code in other contract
- callcode—(discarded)

delegatecall is the revision of callcode which is obsoleted and will be removed in the future.

4.4.1 Function Call

Calling method in other contracts is through message. Whenever a contract calls functions in another contract, a message call will be generated. Every call has a sender, a recipient, a payload, a value, and gas.

Developer can provide gas and ether for call in the way as below:

```
someAddress.call.gas(1000000).value(1    ether)("register",
"MyName");
```

gas is the gas paid for the call, address is the address of callee, value is the ether to be sent and the unit is wei, data is the payload of the call. Please note that value and gas are optional.

Under the situation of out-of-gas (OOG) exception, at least one 64th of GAS will be reserved for sender to handle OOG exception and to stop the execution. This mechanism is trying to avoid bubbling up the exception. Let us have a look at the following Caller contract:

```
contract Implementation {
  event ImplementationLog(uint256 gas);
    function() public payable {
    emit ImplementationLog(gasleft());
    assert(false);
  }
}
contract Caller {
  event CallerLog(uint256 gas);
  Implementation public implementation;
  function Caller() public {
```

(continued)

```
    implementation = new Implementation();
  }
  function () public payable { // Fallback Function
    emit CallerLog(gasleft());
    implementation.call.gas(gasleft()).value(msg.value)(msg.
data);
    emit CallerLog(gasleft());
  }
}
```

Caller contract only has a fallback function which redirects all the request to an Implementation instance. And this instance just runs assert (false) statement for each request. This will lead to out of GAS. The design is to record the GAS before Implementation and after Implementation. Let us open a truffle console to see what happens:

```
truffle(develop) > compile
truffle(develop) > Caller.new().then(i => caller = i)
truffle(develop) > opts = { gas: 4600000 }
truffle(develop) > caller.sendTransaction(opts).then(r =>
result = r)
truffle(develop) > logs = result.receipt.logs
truffle(develop) > parseInt(logs[0].data) //4578955
truffle(develop) > parseInt(logs[1].data) //71495
```

As can be seen from above, 71495 is about one 64th of 4578955. And the example above demonstrates that developer can handle the OOG exception for inner calls.

Solidity also provides the following opcode, which allows us to use inline assembly to manage call:

```
call(g, a, v, in, insize, out, outsize)
```

g is the GAS provided, *a* is callee's address, *v* is ether to be sent in the unit of wei, *insize* bytes of memory starting from *in* is used to save calldata, *out* and *outsize* memory is used to save return data. The only difference is that the opcode allows developers to process return data. On contrary, call function only returns 1 or 0.

Please note that using delegatecall may cause security risk since callee can access and manipulate caller's storage. Due to the constraint of EVM, call and delegatecall do not have a return value.

4.4.2 Dependency Injection

Another method to call methods in other contract is dependency injection: caller needs to instantiate a callee contract instance first and get callee's function signature. Using this method, developer can get the return value of the function call. The following is an example using the source code in section 2.2.2.1: caller and callee.

- *Callee contract*
  Callee contract has an integer array and provides getter and setter to access the array. It also defines getValues with one input and one output. Function getValues is used to demonstrate how to pass parameter and how to return value.
- *Caller contract*
  At the bottom of the caller contract, you can find the Callee interface, which has the same signature with Callee. This interface can also be defined in a separated sol file and be imported. In this way, interface and implementation are separated. The contract provides three functions which all accept the address as parameter. The address parameter is the address of the deployed Callee contract. It is possible to initialize caller contract with an address and change it afterwards. A real scenario is to upgrade deployed contract.

4.4.3 Message Calls

EVM supports a special message call—delegatecall. Solidity also provides an assembly version of the built-in method. The difference between those two is: compared to low-level opcode, callee's code is run in the context of caller, which means that msg.sender and msg.value do not change when delegatgecall is used.

Let us analyze the following Greeter contract to understand the working mechanism of delegatecall.

```
contract Greeter {
   event Hello(address sender, uint256 value);
   function hello() public payable {
     emit Hello (msg.sender, msg.value);
   }
}
```

Greeter contract only defines a hello function. The only functionality of the function is to trigger Hello event, and pass msg.value and msg.sender to event as data. We call this method through Truffle console:

```
truffle(develop) > compile
truffle(develop) > someone = web3.eth.accounts[0]
truffle(develop) > ETH_2 = new web3.BigNumber('2e18')
truffle(develop) > Greeter.new().then(i => greeter = i)
truffle(develop) > opts = { from: someone, value: ETH_2 }
truffle(develop) > greeter.thanks(opts).then(tx => log =
tx.logs[0])
truffle(develop) > log.event                   //Hello
truffle(develop) > log.args.sender === someone  //true
truffle(develop) > log.args.value.eq(ETH_2)    //true
```

Now, we have a look at Wallet contract: this contract defines a fallback function. The fallback function call Hello in Greeter contract through delegatecall.

```
contract Wallet {
  Greeter internal greeter;
  function Wallet() public {
    greeter = new Greeter();
  }
  function () public payable { // Fallback function
    bytes4 methodId = Greeter(0).thanks.selector;
    require(greeter.delegatecall(methodId));
  }
}
```

Let us verify the logic in truffle:

```
truffle(develop) > Wallet.new().then(i => wallet = i)
truffle(develop) > wallet.sendTransaction(opts).then(r => tx = r)
truffle(develop) > logs = tx.receipt.logs
truffle(develop) > SolidityEvent = require('web3/lib/web3/event.
js')
truffle(develop) > Thanks = Object.values(Greeter.events)[0]
truffle(develop) > event = new SolidityEvent(null, Thanks, 0)
truffle(develop) > log = event.decode(logs[0])
truffle(develop) > log.event                  // Hello
truffle(develop) > log.args.sender === someone // true
truffle(develop) > log.args.value.eq(ETH_2)   // true
```

As can be seen from the example above, delegatecall function keeps the msg. value and msg.sender of calling contract. This means that a contract can dynamically load code of another contract. Callee contract's code works in the context of calling contract, including storage, address, balance, etc. We have a look at following Calculator contract to see how to use delegatecall to access storage variables.

```
// Base storage contract
contract ResultStorage {
        uint256 public result;
}
// Calculator contract derived from ResultStorage
contract Calculator is ResultStorage {
  Product internal product; // Product contract
  Addition internal addition; // Addition contract
  function Calculator() public {
    product = new Product();
    addition = new Addition();
  }
  // Addition proxy to Addition contract
  function add(uint256 x) public {
    bytes4 methodId = Addition(0).calculate.selector;
    require(addition.delegatecall(methodId, x));
  }
  // Multiple proxy to Product contract
  function mul(uint256 x) public {
    bytes4 methodId = Product(0).calculate.selector;
    require(product.delegatecall(methodId, x));
  }
}
// Addition contract derived from ResultStorage
contract Addition is ResultStorage {
  function calculate(uint256 x) public returns (uint256) {
    uint256 temp = result + x;
    assert(temp >= result);
    result = temp;
    return result;
  }
}
// Multiply contract derived from ResultStorage
contract Product is ResultStorage {
  function calculate(uint256 x) public returns (uint256) {
    if (x == 0) result = 0;
    else {
      uint256 temp = result * x;
      assert(temp / result == x);
      result = temp;
    }
    return result;
  }
}
```

Calculator contract has two functions: add and product. Calculator contract does not know how to do addition or multiplication; It is only a proxy redirecting addition or multiplication to addition and product contract. But all contracts share the same storage variables which keep the calculation result. Let us verify through Truffle:

```
truffle(develop) > Calculator.new().then(i => calculator = i)
truffle(develop) > calculator.addition().then(a =>
additionAddress=a)
truffle(develop) > addition = Addition.at(additionAddress)
truffle(develop) > calculator.product().then(a => productAddress
= a)
truffle(develop) > product = Product.at(productAddress)
truffle(develop) > calculator.add(5)
truffle(develop) > calculator.result().then(r => r.toString()) //
5
truffle(develop) > addition.result().then(r => r.toString())  // 0
truffle(develop) > product.result().then(r => r.toString())  // 0
truffle(develop) > calculator.mul(2)
truffle(develop) > calculator.result().then(r => r.toString()) //
10
truffle(develop) > addition.result().then(r => r.toString())  // 0
truffle(develop) > product.result().then(r => r.toString())  // 0
```

Above testing results show that through delegatecall, we are actually working on the storage of Calculator contract while code is stored in Addition and Product contract.

We also have delegatecall opcode in assembly:

```
contract Implementation {
  event ImplementationLog(uint256 gas);
  function() public payable {
    emit ImplementationLog(gasleft());
    assert(false);
  }
}
// Proxy contract
contract Delegator {
  event DelegatorLog(uint256 gas);
  Implementation public implementation;

  // Contract keeping Implementation object
  function Delegator() public {
    implementation = new Implementation();
  }
  function () public payable {
    emit DelegatorLog(gasleft());
    address _impl = implementation;
  assembly {
    let ptr := mload(0x40) // Get free memory pointer
    calldatacopy(ptr, 0, calldatasize) // Copy calldata to free
memory pointer
     // Call contract at address _impl through delegatecall
```

(continued)

```
      let result := delegatecall(gas, _impl, ptr, calldatasize, 0, 0)
    }

    emit DelegatorLog(gasleft());
  }
}
```

Let us emphasize that through delegatecall, callee's code can read/write the storage of Caller contract.

4.4.4 Return Value of Contract Call

In this section, we are going to introduce how to get the return value of the call between different contract methods. Assume that we already deploy a simple contract—"Deployed," and it has only one function allowing users to set/update the member variable of the "Deployed" contract.

```
pragma solidity ^0.4.18;
contract Deployed {
    uint public a = 1;

    function setA(uint _a) public returns (uint) {
        a = _a;
        return a;
    }
}
```

Then, we deploy another contract—"Existing" which plans to update the variable "a" in "Deployed" contract.

```
pragma solidity ^0.4.18;
contract Deployed {
      function setA(uint) public returns (uint) {}
      function a() public pure returns (uint) {}
}
contract Existing {
    Deployed dc;
    function Existing(address _t) public {
        dc = Deployed(_t);
    }
    function getA() public view returns (uint result) {
       return dc.a();
```

(continued)

```
    }
    function setA(uint _val) public returns (uint result) {
      dc.setA(_val);
      return _val;
    }
}
```

Here we only need the function signature which is required by ABI standard. During the initialization of “Existing” contract, “Existing” contract saves the address of “Deployed” contract and it contains two functions to call contract “Deployed”; SetA and GetA. However, if we do not have the ABI of deployed contract, do we still can invoke setA function in “Existing” contract? The answer is yes.

```
pragma solidity ^0.4.18;
contract ExistingWithoutABI {
     address dc;
   function ExistingWithoutABI(address _t) public {
      dc = _t;
   }
   function setA_Signature(uint _val) public returns(bool
success){
      require(dc.call(bytes4(keccak256("setA(uint256)")),
_val));
      return true;
   }
}
```

Function signature is 4 bytes long and the formula to calculating it is:

```
bytes4(keccak256("setA(uint256)”))
```

When we use the call method in setA method, we can pass a value since Call (and delegatecall) method can only pass value and cannot get a return value. So, we are not sure if setA is working properly unless we go to check the “Deployed” contract. As an alternative, we can use the Solidity inline assembly to get the return value of setA.

```
pragma solidity ^0.4.18;
contract ExistingWithoutABI {
    address dc;
    function ExistingWithoutABI(address _t) public {
       dc = _t;
    }
```

(continued)

```
    function setA_ASM(uint _val) public returns (uint answer) {
        // Create function signature
        bytes4 sig = bytes4(keccak256("setA(uint256)"));
        assembly {
          // Get free memory pointer
          let ptr := mload(0x40)
          // Put function signature to free memory pointer
          mstore(ptr,sig)
          // Append parameters to function signature
          mstore(add(ptr,0x04), _val)
          // Invoke call opcode
          let result := call(
            15000, // gas limit
            sload(dc_slot), // Target contract address
            0, //  do not send ether
            ptr, // input at ptr
            0x24, // the size of input is 36 bytes = 4 byte function
signature + 32 bytes parameters
            ptr,  // Return value at ptr
            0x20) // size of return value
          if eq(result, 0) {
            revert(0, 0)
          }
          answer := mload(ptr) // put the value at free memory pointer
into answer
          mstore(0x40,add(ptr,0x24)) // modify free memory pointer
        }
    }
}
```

"assembly" keyword indicates that the code enclosed in curly bracket is Solidity inline assembly. The code gets the free memory pointer from address 0×40. Then, we pass the function signature and parameters to free memory pointer. Since function signature is 4 bytes long and the parameter is aligned to 32 bytes, there are 36 bytes in totally. As the next step, we use "call" opcode and return value will be set at the place of free memory pointer. In the example above, return value is Boolean, and could be 0 or 1. If the return value is 0, then revert.

4.5 Basic Algorithms

In smart contract programming, we may use some traditional algorithms, such as sorting (bubble sorting, binary search, etc.), tree (Merkle tree), graph, etc. Since operations in contract cost GAS, we need to consider about GAS usage in smart contract programming, which is different from implementation in Java, C++, or Python.

First, let us summarize operations which cost GAS and do not cost GAS:

Operations which do not cost GAS:

```
- Read state variables
- Read <address>.balance or this.balance
- Read block variables tx, or sg
- Call any pure function
- Use "read" opcodes in inline assembly
```

Operations which cost GAS:

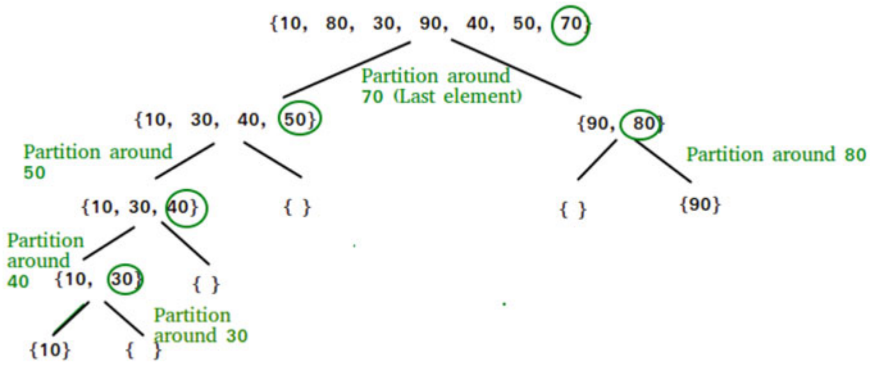
```
- send ethers
- Create contracts
- change state variables
- Send Event
- Call any non-pure or non-view function
- selfdestruct
- low-level calls
- Use "write" opcodes in inline assembly
```

We are not going to introduce all basic algorithms such as list, tree, and graph. We only use quicksort algorithm as an example to showcase different consideration in Solidity programming. First, quicksort is a famous sorting algorithm and it accepts an array as input. It selects a pivot based on some rule (random or medium). All elements smaller than the pivot are swapped to its left and all elements bigger than the pivot are swapped to its right. Then, recursively call function to the left array and right array until the array contains only one or zero element. Following pseudo-code shows the logic[6]:

```
/* low  --> Start of Array, high -->End of Array */
quickSort(arr[], low, high)
{
   if (low < high)
   {
     /* pi is pivot, and arr[pi] is treated as right array */
     pi = partition(arr, low, high);

     quickSort(arr, low, pi - 1); // recursive sort for elements
before pi
     quickSort(arr, pi + 1, high); // recursive sort for elements
after pi
   }
}
```

[6]https://www.geeksforgeeks.org/quick-sort

Quick-sort example:



Now, we have the keyword—swap. In the traditional programming language, it is easy to swap between two elements through declaration of a temporary variable (for example: temp variable below). Sample code is like below:

```
if (arr[j] <= pivot) {
  i++;
  // swap between arr[i] and arr[j]
  int temp = arr[i];
  arr[i] = arr[j];
  arr[j] = temp;
}
```

However, since creating temp variables in Solidity still cost GAS, we need to implement element swap in minimum cost. Here, we just introduce a simple solution: XOR swap. The pseudo logic for element swap between arr[i] and arr [j] becomes:

```
// Solidity (XOR swap)
arr[uint(i)] ^= arr[uint(j)];
arr[uint(j)] ^= arr[uint(i)];
arr[uint(i)] ^= arr[uint(j)];
```

And we can continue to improve the code above. Since method call in Solidity still cost GAS, we can put the body of Partition function directly into Sort function to save one method call to Partition(). And there are still a lot to improve. For example: remove force type cast, save extra if condition, etc.

Following is a neat version of Quick-sort.

```
pragma solidity ^0.4.18;
contract QuickSort {
   function sort(uint[] data) public constant returns(uint[]) {
     quickSort(data, int(0), int(data.length - 1));// Recursive
call
     return data;
   }
   function quickSort(uint[] memory arr, int left, int right)
internal{
     int i = left;
     int j = right;
     if(i==j) return;
     uint pivot = arr[uint(left + (right - left) / 2)];// pick Pivot点
     while (i <= j) {
        while (arr[uint(i)] < pivot) i++;
        while (pivot < arr[uint(j)]) j--;
        if (i <= j) {
           (arr[uint(i)], arr[uint(j)]) = (arr[uint(j)], arr[uint
(i)]);
           i++;
           j--;
        }
     }
     if (left < j)
        quickSort(arr, left, j); // Recursive call of left part
     if (i < right)
        quickSort(arr, i, right); // Recursive call of right part
   }
}
```

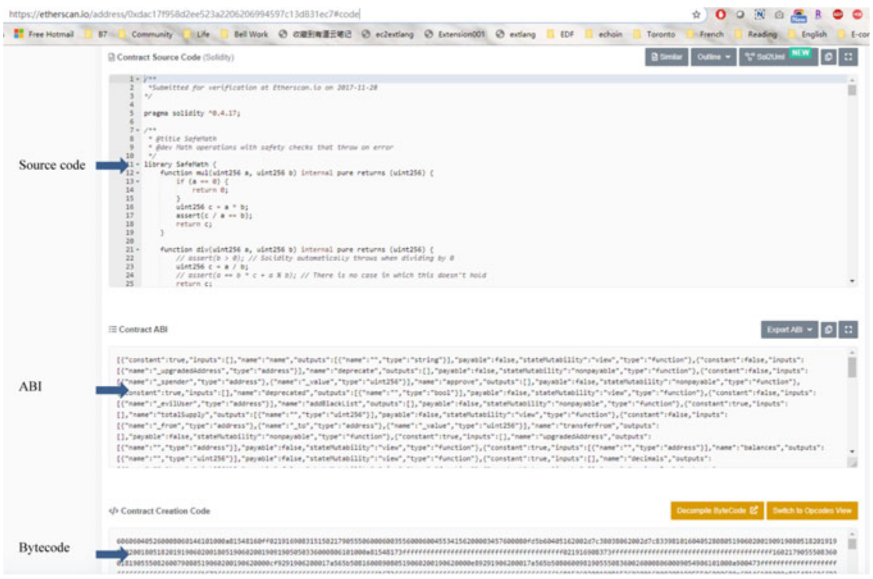
4.6 Using Golang to Interact with Contract

In Chap. 2, we already showed how to deploy and debug smart contact by using Remix, JavaScript, or Truffle framework. The benefit of using existing tools or framework is fast and efficient. The downside is that we may ignore many details. Most programmers prefer to go through the whole procedure manually and it helps programmers to dig out more technical details and understand basic principles. In this section, we are going to deploy a contract based on solc + GO[7].

[7]https://hackernoon.com/a-step-by-step-guide-to-testing-and-deploying-ethereum-smart-contracts-in-go-9fc34b178d78

4.6.1 Contract Source

USDT is considered as stable coin in crypto currency world. Traditional USDT is running on the omni layer on top of Bitcoin network. Now, USDT has an ERC20 version at https://etherscan.io/address/0xdac17f958d2ee523a2206206994597c13d831ec7. Most of crypto exchanges support ERC20 version of USDT. USDT ERC20 contract source code, ABI, and bytecode can be downloaded at https://etherscan.io/address/0xdac17f958d2ee523a2206206994597c13d831ec7#code as below:



```
/**
 *Submitted for verification at Etherscan.io on 2017-11-28
*/

pragma solidity ^0.4.17;

/**
 * @title SafeMath
 * @dev Math operations with safety checks that throw on error
 */
library SafeMath {
    function mul(uint256 a, uint256 b) internal pure returns (uint256) {
        if (a == 0) {
            return 0;
        }
        uint256 c = a * b;
        assert(c / a == b);
        return c;
    }

    function div(uint256 a, uint256 b) internal pure returns (uint256) {
        // assert(b > 0); // Solidity automatically throws when dividing by 0
        uint256 c = a / b;
        // assert(a == b * c + a % b); // There is no case in which this doesn't hold
        return c;
```

[{"constant":true,"inputs":[],"name":"name","outputs":[{"name":"","type":"string"}],"payable":false,"stateMutability":"view","type":"function"},{"constant":false,"inputs":
[{"name":"_upgradedAddress","type":"address"}],"name":"deprecate","outputs":[],"payable":false,"stateMutability":"nonpayable","type":"function"},{"constant":false,"inputs":
[{"name":"_spender","type":"address"},{"name":"_value","type":"uint256"}],"name":"approve","outputs":[],"payable":false,"stateMutability":"nonpayable","type":"function"},
{"constant":true,"inputs":[],"name":"deprecated","outputs":[{"name":"","type":"bool"}],"payable":false,"stateMutability":"view","type":"function"},{"constant":false,"inputs":
[{"name":"_evilUser","type":"address"}],"name":"addBlackList","outputs":[],"payable":false,"stateMutability":"nonpayable","type":"function"},{"constant":true,"inputs":
[],"name":"totalSupply","outputs":[{"name":"","type":"uint256"}],"payable":false,"stateMutability":"view","type":"function"},{"constant":false,"inputs":
[{"name":"_from","type":"address"},{"name":"_to","type":"address"},{"name":"_value","type":"uint256"}],"name":"transferFrom","outputs":
[],"payable":false,"stateMutability":"nonpayable","type":"function"},{"constant":true,"inputs":[],"name":"upgradedAddress","outputs":
[{"name":"","type":"address"}],"payable":false,"stateMutability":"view","type":"function"},{"constant":true,"inputs":[{"name":"","type":"address"}],"name":"balances","outputs":
[{"name":"","type":"uint256"}],"payable":false,"stateMutability":"view","type":"function"},{"constant":true,"inputs":[],"name":"decimals","outputs":

[illegible]

We copy and paste the source code into a local file—usdterc20.sol. And we add a new member variable—message and a method setMessage() to main contract—TetherToken:

```
contract TetherToken is Pausable, StandardToken, BlackList {
    string public name;
    string public symbol;
    uint public decimals;
    address public upgradedAddress;
    bool public deprecated;
    string public message = "This is a test";

    ...
```

(continued)

```
    // setter function
    function setMessage(string newMessage) public {
      message = newMessage;   }
    ...
}
```

In the following sections, we are going to show how to use Golang to deploy the contract above and how to interact with the deployed contract.

4.6.2 Project Creation

We are going to develop a simple USDT similar contract. First, let us to create the directory structure:

```
# Navigate to your Go src directory. Mine looks like:
# $GOPATH/src/demousdt
$ cd $GOPATH/src/demousdt
$ mkdir -p demousdt/ERC20
$ touch main.go
$ touch ERC20/connector.go
$ tree demousdt/
demousdt/
├── ERC20
        │├── connector.go
        │├── usdterc20_test.go
│        └── usdterc20.sol
├── update.go
└── main.go
```

Project root directory is demousdt. We create an ERC20 directory under the root directory, which contains all Solidity contract related source code. We have connector.go which is a wrapper of all interaction with contract. There is also a main.go to interact with the contract.

4.6.3 Using Golang to Interact with Contract

Next, we need to access and deploy the contract to Ethereum in our GO program which can also interact with the contract. Geth provides a simple code generation tool which can transfer a Solidity contract into a type-safe go library. And we can import the go library and use it directly. The tool is abigen. Let us go to inbox/contracts directory and run:

```
$
$abigen --abi usdterc20.abi --pkg ERC20 --type usdterc20 --out
usdterc20.go --bin usdterc20.bin

$ tree demousdt
demousdt
└── ERC20
    ├── usdterc20.sol
    ├── usdterc20.abi
    ├── usdterc20.bin   └── usdterc20.go
```

The command above generates usdterc20.go for usdterc20.sol automatically. usdterc20.go includes the link to usdterc20 smart contract. And we can start to test from now on.

4.6.4 Local Test

Before deploying the contract, we need to confirm it works fine on our local. Geth provides a tool to simulate blockchain. We will show how to use this tool below.

```
package ERC20
import (
   "testing"
   "github.com/ethereum/go-ethereum/accounts/abi/bind/
backends"
   "github.com/ethereum/go-ethereum/accounts/abi/bind"
   "github.com/ethereum/go-ethereum/crypto"
   "github.com/ethereum/go-ethereum/core"
   "math/big"
)

// Test USDTERC20 contract gets deployed correctly
func TestDeployUSDTERC20(t *testing.T) {
   //Setup simulated block chain
   key, _ := crypto.GenerateKey()
   auth := bind.NewKeyedTransactor(key)
   alloc := make(core.GenesisAlloc)
   alloc[auth.From] = core.GenesisAccount{Balance: big.NewInt
(1000000000)}
   blockchain := backends.NewSimulatedBackend
(alloc,100000000000)
```

(continued)

```
    //Deploy contract
    var total, _ = new(big.Int).SetString
("1000000000000000000000000000", 10)
    address, _, _, err := DeployUSDTERC20(auth, blockchain, total,
"usdt erc20", "usdterc201", big.NewInt(18))

    // commit all pending transactions
    blockchain.Commit()

    if err != nil {
      t.Fatalf("Failed to deploy the USDT ERC20 contract: %v", err)
    }
    if len(address.Bytes()) == 0 {
      t.Error("Expected a valid deployment address. Received empty
address byte array instead")
    }
}
```

TestDeployUSDTERC20 function first calls crypto.GenerateKey to generate a private key which will be used to develop transaction signature function used to simulate authorization transaction on blockchain. We create the function signature through bind.NewKeyedTransactor. And then we use this address to create genesis block which includes some account with some initial amount. This is implemented by calling make (core.GenesisAlloc and core.GenesisAccount). And at last, we start mining, showing all transactions waiting for process and verifying inbox contract has been deployed to a valid address.

Enter demousdt\ERC20 directory, and run go test command to check if our deployment works properly.

```
$ go test -v
=== RUN   TestDeployUSDTERC20
--- PASS: TestDeployUSDTERC20 (0.01s)
PASS
ok    demousdt/ERC20 0.335s
```

Next, test if our deployed contract contains correct initialization info.

```
package contracts
import (
        "testing"
        "github.com/ethereum/go-ethereum/accounts/abi/bind/
backends"
        "github.com/ethereum/go-ethereum/accounts/abi/bind"
```

(continued)

```
        "github.com/ethereum/go-ethereum/crypto"
        "github.com/ethereum/go-ethereum/core"
        "math/big"
)
//Test initial message gets set up correctly
func TestGetMessage(t *testing.T) {
  //Setup simulated block chain
  key, _ := crypto.GenerateKey()
  auth := bind.NewKeyedTransactor(key)
  alloc := make(core.GenesisAlloc)
  alloc[auth.From] = core.GenesisAccount{Balance: big.NewInt
(1000000000)}
  blockchain := backends.NewSimulatedBackend
(alloc,100000000000)

  //Deploy contract
  var total, _ = new(big.Int).SetString
("1000000000000000000000000000", 10)
  _, _, contract, _ :=DeployUSDTERC20(auth, blockchain, total,
"usdt erc20", "usdterc201", big.NewInt(18))
  // commit all pending transactions
  blockchain.Commit()
  if got, _ := contract.Message(nil); got != "This is a test" {
    t.Errorf("Expected message to be: Hello World. Go: %s", got)
  }
}
```

Similar to TestDeployUSDTERC20 above, TestGetMessage function first sets up the environment and then calls DeployUSDTERC20which is generated automatically. DeployUSDTERC20function returns a pointer to a deployed inbox instance. We can use this pointer to interact with the deployed USDTERC20 contract. In our test cases, we will query and validate the message in contract instance.

Under demousdt\ERC20, we run go test and confirm that all test cases pass.

```
$ go test -v
=== RUN  TestDeployUSDTERC20
--- PASS: TestDeployUSDTERC20 (0.01s)
=== RUN  TestGetMessage
--- PASS: TestGetMessage (0.01s)
PASS
ok   demousdt/ERC20 0.329s
```

Last, we are going to test and update message property in the deployed contract.

```
package contracts
import (
       "testing"
       "github.com/ethereum/go-ethereum/accounts/abi/bind/
backends"
       "github.com/ethereum/go-ethereum/accounts/abi/bind"
       "github.com/ethereum/go-ethereum/crypto"
       "github.com/ethereum/go-ethereum/core"
       "math/big"
)
// Test message gets updated correctly
func TestSetMessage(t *testing.T) {
  //Setup simulated blockchain
  key, _ := crypto.GenerateKey()
  auth := bind.NewKeyedTransactor(key)
  alloc := make(core.GenesisAlloc)
  alloc[auth.From] = core.GenesisAccount{Balance: big.NewInt
(1000000000)}
  blockchain := backends.NewSimulatedBackend
(alloc,100000000000)

  //Deploy contract
  var total, _ = new(big.Int).SetString
("1000000000000000000000000000", 10)
  _, _, contract, _ :=DeployUSDTERC20(auth, blockchain, total,
"usdt erc20", "usdterc201", big.NewInt(18))

  // commit all pending transactions
  blockchain.Commit()
  contract.SetMessage(&bind.TransactOpts{
    From:auth.From,
    Signer:auth.Signer,
    Value: nil,
  }, "Hello World")
  blockchain.Commit()
  if got, _ := contract.Message(nil); got != "Hello World" {
    t.Errorf("Expected message to be: Hello World. Go: %s", got)
  }
}
```

TestSetMessage function first sets up the environment and then calls DeployUSDTERC20 function to get deployed instance of Inbox Contract. We use this pointer to call SetMessage function to update the property. And this actually creates a new transaction. At last, we pass a TransactOpts struct pointer which contains transaction authorization.

We run “go test” to make sure that all test cases pass.

```
$ go test -v
=== RUN  TestDeployUSDTERC20
--- PASS: TestDeployUSDTERC20 (0.01s)
=== RUN  TestGetMessage
--- PASS: TestGetMessage (0.01s)
=== RUN  TestSetMessage
--- PASS: TestSetMessage (0.01s)
PASS
ok   demousdt/ERC20 0.353s
```

After our contract passed local test, we are ready to deploy it to Ethereum blockchain. We need to use Metamask to create a new account and deposit some money into this account.

4.6.5 Connect to an Ethereum Node

We may need to run geth on our local, which is resource-costly and time-intensive. A better solution is to connect to a third-party node service platform, such as infura. We can get a free infura account. Once the registration is completed, infura will connect to different nodes in a different environment, such as mainnet, rinkeby, ropsten, etc. In connector.go, we have code demonstrating how to connect to Ethereum Node through infura.

```
func NewConnecter() *Connecter {
  // Dial support multiple ways of connection such as ws、http、ipc
  conn, err :=
ethclient.Dial("https://ropsten.infura.io/v3/
fdda941e249941418aeb27b0323d03c6")
  if err != nil {
    panic(err)
  }
  coin, err := NewUSDTERC20(CoinAddr, conn)
  if err != nil {
    panic(err)
  }
  return &Connecter{
    ctx: context.Background(),
    conn: conn,
    coin: coin,
  }
}
```

Please note that your infura URL will be different from the infura URL above.

4.6.6 Create JSON Key for Our Account

In order to interact with contracts on Ethereum network, such as Rinkeby testnet, we need JSON key file created for Metamask account. We use Metamask account to deploy and interact with Inbox contract. In Metamask, we click "export private key" to export private key to a file and then load it into geth.

```
geth account import path/to/private/key/file
```

The command above creates an encrypted JSON key and we use it to interact with contract.

4.6.7 Validation

Now, let us to deploy the contract in main.go:

```
package main
import (
  "demousdt/ERC20"
  "github.com/ethereum/go-ethereum/common"
  "log"
  "math/big"
)

var (
  srcAddress  = common.HexToAddress
("0xA4e71aEeAdaA01FD0f15455f67D7d2CAD32EbFcA") // Keystore

)

func main() {
  keyin := "{\"address\":
\"a4e71aeeadaa01fd0f15455f67d7d2cad32ebfca\",\"crypto\":
{\"cipher\":\"aes-128-ctr\",\"ciphertext\":
\"7573755cb636a8712c595b63a69e69c968ff7be194e8757b1
122584a63ad2457\",\"cipherparams\":{\"iv\":
\"2afb66d1c584804b8f3265fff3b220cb\"},\"kdf\":\"scrypt\",
\"kdfparams\":{\"dklen\":32,\"n\":262144,\"p\":1,\"r\":8,
\"salt\":\"bad691204ff97a29742329338a56c91df858ee383733d
1357e95f7b4736ea5a6\"},\"mac\":
\"061dae9ec6409181dea894afd52c5
acf779e320a970159a91ab334a54a55c4d1\"},\"id\":\"d4a01c73-
```

(continued)

```
baeb-44b8-9489-2c6a7df5aa45\",\"version\":3}"
  srcTransactOpt := ERC20.AuthAccount(keyin, "geth123")

  s:=ERC20.NewConnecterWithDeploy(srcTransactOpt)

  // s.WatchTransferAndMint()
  log.Printf("Contract Name = %s, BlockNo= %d, BalanceofETH=%d,
BalanceofCoin=%d\n", s.ContractName(), s.BlockNumber(), s.
BalanceOfEth(srcAddress), s.BalanceOfCoin(srcAddress))
}
```

In ERC20/connector.go, we plan to deploy the contract with initial supply of 1000000 coins with decimal 18.

```
// NewConnecterWithDeploy deploy contract and create connecter
func NewConnecterWithDeploy(ownerAuth *bind.TransactOpts)
*Connecter {
   conn, err :=
ethclient.Dial("https://ropsten.infura.io/v3/
fdda941e249941418aeb27b0323d03c6")
  if err != nil {
    panic(err)
  }

   var total, _ = new(big.Int).SetString
("1000000000000000000000000", 10)
   _, tx, coin, err := DeployUSDTERC20(ownerAuth, conn, total,
"usdt erc20", "usdterc201", big.NewInt(18))
  if err != nil {
    panic(err)
  }
  ctx := context.Background()
  CoinAddr, err = bind.WaitDeployed(ctx, conn, tx)
  if err != nil {
    panic(err)
  }
  CoinHash = CoinAddr.Hash()
  return &Connecter{
    ctx: ctx,
    conn: conn,
    coin: coin,
  }
}
```

We use ethclient.Dial to connect Ropsten testnet through infura. And in main.go, we use ERC20.AuthAccount(keyin, "geth123") method to create an approved trader from key JSON. We could use “geth account list” command to find the location of

JSON key file. We generated password derived from the output of command “geth account import.”

```
$ go run main.go
```

It can take some time for contract being mined into block. We can check the status through blockchain explorer at: https://ropsten.etherscan.io/address/[contract address]. Our contract is deployed at https://ropsten.etherscan.io/address/0xc4dbf323c3c2ae821e4e02afe5a0542c78423e8b.

Once transactions are mined into block, we can interact with the deployed contract through contract address. As can be seen from main.go, we use member functions in connector.go to interact with the contract.

```
package main
import (
   "demousdt/ERC20"
   "github.com/ethereum/go-ethereum/common"
   "log"
   "math/big"
)
func main(){

   ...
   s:=ERC20.NewConnecterWithDeploy(srcTransactOpt)

   log.Printf("Contract Name = %s, BlockNo= %d, BalanceofETH=%d,
BalanceofCoin=%d\n", s.ContractName(), s.BlockNumber(), s.
BalanceOfEth(srcAddress), s.BalanceOfCoin(srcAddress))
 ...
}
```

As can be seen from the code above, we use ethclient.Dial to connect Ropsten testnet through infura URL and return connector struct containing the usdterc20 instance which is bound to the deployed contract address. Last, we access properties of contract (contract name, balance of ETH, and balance of USDTERC20) and print them. Here is the output:

```
2019/12/12 15:12:13 Contract Name = usdt erc20, BlockNo= 6954046,
BalanceofETH=2887430890000000000,
BalanceofCoin=100000000000000000000000000
```

Next, remember that we add a new member variable—**message** and a new setter function—**setMessage** and message variable is initialized to value “this is a test.” In update.go, we change the value of message in USDTERC20 contract.

```
package main
import (
   "demousdt/ERC20"
   "github.com/ethereum/go-ethereum/common"
   "github.com/ethereum/go-ethereum/ethclient"
   "log"
   "time"
)

var (
  CoinAddr = common.HexToAddress
("0xc4dbf323c3c2ae821e4e02afe5a0542c78423e8b") // USDT

  // CoinHash superCoin 合约地址Hash
  CoinHash = CoinAddr.Hash()
)

func main() {
  keyin := "{\"address\":
\"a4e71aeeadaa01fd0f15455f67d7d2cad32ebfca\",\"crypto\":
{\"cipher\":\"aes-128-ctr\",\"ciphertext\":
\"7573755cb636a8712c595b63a69e69c968ff7be194
e8757b1122584a63ad2457\",\"cipherparams\":{\"iv\":
\"2afb66d1c584804b8f3265fff3b220cb\"},\"kdf\":\"scrypt\",
\"kdfparams\":{\"dklen\":32,\"n\":262144,\"p\":1,\"r\":8,
\"salt\":\"bad691204ff97a29742329338a56c91df858ee383
733d1357e95f7b4736ea5a6\"},\"mac\":
\"061dae9ec6409181dea894afd52c5acf779
e320a970159a91ab334a54a54b4d2\"},\"id\":\"d4a01c73-baeb-44b8-
9489-2c6a7df5aa45\",\"version\":3}"
  srcTransactOpt := ERC20.AuthAccount(keyin, "geth123")

  conn, err :=
ethclient.Dial("https://ropsten.infura.io/v3/
fdda941e249941418aeb27b0323d03c6")
  if err != nil {
    panic(err)
  }
  coin, err := ERC20.NewUSDTERC20(CoinAddr, conn)
  if err != nil {
    panic(err)
  }

  str,_ := coin.Message(nil)
  log.Printf("Before : message = %s,\n", str )
  coin.SetMessage(srcTransactOpt,"USDT ERC20")
}
```

The code above shows that we use SetMessage to modify contract property by passing a TransactOpts struct which contains authorization data. Once the transaction is mined into some blocks, we can use contract.Message to get modified message property.

Now, we know the key protocols behind ICO(ERC20) and Cryptokitty (ERC721). Also, we learned 3 methods for cross contract calls and their pros and cons. And to dig out more technical details, we studied how to interact with smart contract by using Golang. In the following sections, we will dive deep into smart contract programming in Solidity.

Part III
Solidity Advanced Features

Chapter 5
Application Binary Interface (ABI)

When we finish the Solidity coding and compiling successfully, the next step is to let EVM execute the bytecode generated. From EVM point of view, transaction calldata is just a byte array and we need to understand how EVM interprets the byte array. If all the programming language based on EVM use the same standard to interpret input data, then it will be easier for them to interact with each other. This is why we need Application Binary Interface (ABI). Just as Google Protocol Buffer, Application Binary Interface (ABI) is a general-purposed data exchange format specified by Ethereum. ABI is applied to the interaction between external call and contract.

5.1 Memory Structure

When a contract starts to run, the EVM memory structure is shown below:

```
0x00 - 0x3f: Space reserved for Hash
0x40 - 0x5f: Current Memory Size (eg Free Memory Pointer)
0x60 - 0x7f: Storage slot with initialized value 0
```

5.2 Function Selector

The first 4 bytes of calldata are function selector. Parameter starts from 5th byte. Here is the formula to generate function selector:

G. Zheng et al., *Ethereum Smart Contract Development in Solidity*,
https://doi.org/10.1007/978-981-15-6218-1_5

```
bytes4(keccak256("functionname (parameter)"))
```

For example, for the function like below:

```
function add(uint256 _a, uint256 _b) public view returns (uint256
result)
```

The calculation of function selector is

```
bytes4(keccak256("add(uint256,uint256)"))
```

5.3 Type Definition

The following is coming from the official documentation of Solidity. Solidity defines the following types.[1]

Name	Description	Example
uint<M>	M bits unsigned int, 0 < M <= 256, M % 8 == 0	uint32, uint8, uint256
int<M>	Mbits signed integer(complementary coded) 0 < M <= 256, M % 8 == 0	int8, int256
Address	Equals to uint160	
uint, int	Equals to uint256, int256	
bool	Equals to uint8, possible values are O or 1	
fixed<M>x<N>	Mbits signed fix point 8 <= M <= 256, M % 8 ==0, 0 < N <= 80, v as v / (10 ** N)	fixed128x18
ufixed<M>x<N>	Variant of unsigned fixed<M>x<N>	ufixed128x18
fixed, ufixed	Equals to fixed128x18, ufixed128x18	
bytes<M>	M bytes binary, 0 < M <= 32	
function	Address + function selector(4 bytes), encode equals to bytes24	
<type>[M] Fixed-length array	Fixed-length array with M elements of specified type M >= 0	
bytes	Dynamic byte array	
string	Dynamic UTF-8 coded Unicode string	
<type>[]	Dynamic array of specified type	
(T1,T2,...,Tn)	Tuple of type T1, . . ., Tn, n >= 0 Tuple can be nested, it is possible to have tuple array or 0 tuple	

[1] https://solidity.readthedocs.io/en/develop/abi-spec.html#abi-json.

Non-fixed types include the following types:

- bytes
- string
- T[] T can be any type
- T[k] Any dynamic type T and any k >= 0
- (T1,...,Tk) Ti is dynamic type and 1 <= i <= k

Any other types are called fixed-size type.

5.4 Data Presentation in EVM

First, we need to understand how EVM presents data types in Solidity and how EVM stores data and its assembly implementation. On top of it, we can better understand the working mechanism of Solidity and how to estimate GAS. For example:

- sstore costs about 20000 gas, which is about 5000 times of that of basic arithmetic opcode
- sload costs about 200 gas, which is about 100 times of that of basic arithmetic opcode

Each contract has 2^{256} storage slots. And 2^{256} is an unfathomably large number. It is approximately 10^{77} in decimal. For comparison, the visible universe is estimated to contain 10e80 atoms.

5.4.1 Presentation of Fixed-Length Data Type

In this section, we introduce fixed-length types, such as basic types, struct, fixed-length array, etc.

5.4.1.1 Zero Value

Let us have a look at a small example below:

```
pragma solidity ^0.4.24;
contract vartest {
    uint256 a;
    uint256 b;
    uint256 c;
```

(continued)

```
    function C() {
      c = 0x519bef;
    }
}
```

A declaration of a storage variable does not incur any cost since there is no need for initialization. Solidity only keeps the space for the storage variable, and you only need to pay when you store values into it. So, for the example above, storage variables a and b do not cost any GAS because a and b are not used yet. You only need to pay for storage slot 0x2 which is the address of variable c.

5.4.1.2 Presentation of Struct

The following is our first complex data type—a struct with three member fields:

```
pragma solidity ^0.4.24;
contract C {
    struct objstruct{
      uint256 a;
      uint256 b;
      uint256 c;
    }
    objstruct t;
    function C() {
      t.c = 0x519bef;
    }
}
```

Just as the last Sect. 5.4.1.1, t.a is saved at storage slot 0x0. Since t.a is not used yet, we do not need to pay for t.a. And we need to pay for t.c since we assign some value to t.c in function C().

5.4.1.3 Fixed-Length Array

Here is a small example of fixed-length array:

```
pragma solidity ^0.4.24;
contract arraytest{
    uint256[3] numbers;
    function C() {
      numbers[2] = 0x519bef;
    }
}
```

The rule for fixed-length array is the same as what is stated in the previous sections. In the example above, although a three-elements array variable is declared, we only need to pay for the assignment to the third element.

5.4.2 Presentation of Dynamic Data Type

Solidity also provides some dynamic types which can be expanded with data growth. There are mainly three dynamic types:

- Mappings: mapping(bytes32 => uint256), mapping(address => string) etc.
- Arrays: []uint256, []byte etc.
- Byte arrays: string, bytes

5.4.2.1 Mapping

The following is a simple example of Mapping:

```
pragma solidity ^0.4.24;
contract mappingtest {
    mapping(uint256 => uint256) mapitems;
    constructor() public {
      mapitems[0x5f0f9b3e] = 0x3f;
    }
}
```

Let us use Solmap to disassemble the contract above and we get the following code. Please note: we can also use remix to disassemble.

```
 20 {0x60} [c30] PUSH1 0x3f (dec 63)
 21 {0x7f} [c32] PUSH32
0xc9d1063eb535bdb1d90a915b0e1c2b6e558bf2e7fd754
dd7e8fe6a4c4d94e3ce (dec 9.128419670285364e+76)
 22 {0x55} [c65] SSTORE
```

The SSTORE opcode above is to save 0x3f into slot 0xc9d1063eb535bdb1d90a915b0e1c2b6e558bf2e7fd754dd7e8fe6a4c4d94e3ce. But what does 0xc9d1063eb535bdb1d90a915b0e1c2b6e558bf2e7fd754dd7e8fe6a4c4d94e3ce mean?

The calculation formula for Map address is like below:

```
keccak256(bytes32(key) + bytes32(position))
```

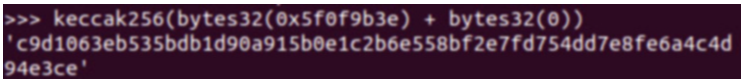

```
>>> keccak256(bytes32(0x5f0f9b3e) + bytes32(0))
'c9d1063eb535bdb1d90a915b0e1c2b6e558bf2e7fd754dd7e8fe6a4c4d
94e3ce'
```

Fig. 5.1 map address calculation output

In order to verify the formula above in Pyethereum, we need to input two helper functions first.

```
>>> def bytes32(i):
...   return binascii.unhexlify('%064x' % i)
...
>>> def keccak256(x):
...   return sha3.keccak_256(x).hexdigest()
```

Then, we type the formula with appropriate parameters (Fig. 5.1).

As can be seen, the result is the address for Map item with key equal to 0x5f0f9b3e. 0x5f0f9b3e is the key and 0 is the number of the storage slot. Let us see what the code is like if the map is big:

```
pragma solidity ^0.4.24;
contract mappingbig {
    mapping(uint256 => Tuple) maptuples;
    struct Tuple {
      uint256 a;
      uint256 b;
      uint256 c;
    }
    constructor () public {
      maptuples[0x1].a = 0x1A;
      maptuples[0x1].b = 0x1B;
      maptuples[0x1].c = 0x1C;
    }
}
```

Using Solmap, we get the assembly code:

```
  20 {0x60} [c27] PUSH1 0x1a (dec 26)
  21 {0x7f} [c29] PUSH32
0xada5013122d395ba3c54772283fb069b10426056ef8ca54750cb9bb552a
59e7d (dec 7.854166079704491e+76)
  22 {0x55} [c62] SSTORE
  23 {0x60} [c63] PUSH1 0x1b (dec 27)
```

(continued)

```
  24 {0x7f} [c65] PUSH32
0xada5013122d395ba3c54772283fb069b10426056ef8ca54750cb9bb552a
59e7e (dec 7.854166079704491e+76)
  25 {0x55} [c98] SSTORE
  26 {0x60} [c99] PUSH1 0x1c (dec 28)
  27 {0x7f} [c101] PUSH32
0xada5013122d395ba3c54772283fb069b10426056ef8ca54750cb9bb552a
59e7f (dec 7.854166079704491e+76)
  28 {0x55} [c134] SSTORE
```

As can be seen, for the <key,value> pair in mapping, if the value is larger than 32 bytes, storage address will increase in turn by 32 bytes. And the compiler will not optimize the code by packing.

5.4.2.2 Dynamic Array

In Solidity, array is the upgraded version of mapping. And array costs more GAS than mapping. All the elements in array will be saved in order in storage slots:

```
0x3ddf...e001
0x3ddf...e002
0x3ddf...e003
0x3ddf...e004
```

Access to slots is exactly the same as querying key-value pair in the database. There are not many differences between accessing an array element and accessing a mapping item. The main difference is that array access is much more complex and much stricter:

- Length – the length of array
- Array boundary check
- Much more complex storage packing behavior compared to mapping
- Do not release unused slots when array shrinks
- Optimization for bytes and string

Let us have a look at the example below:

```
pragma solidity ^0.4.24;
contract arraytest {
   uint256[] array1;
   constructor () public {
     array1.push(0xAA);
```

(continued)

```
        array1.push(0xBB);
        array1.push(0xCC);
    }
}
```

We run it in Remix and check the variable structure in storage:

```
key:
0x0000000000000000000000000000000000000000000000000000000000
000000
value: 0x00000000000000000000000000000000000000000000000000
000000000003 key: 0x290decd9548b62a8d60345a988386fc84ba6bc954
84008f6362f93160ef3e563 value:
0x0000000000000000000000000000000000000000000000000000000000
0000aa key:
0x290decd9548b62a8d60345a988386fc84ba6bc95484008f6362f93160ef
3e564 value: 0x00000000000000000000000000000000000000000000
00000000000000bb key:
0x290decd9548b62a8d60345a988386fc84ba6bc95484008f6362f93160ef
3e565 value: 0x00000000000000000000000000000000000000000000
00000000000000cc
```

The length of array which is three in the case above is saved into Slot 0. The address for specific element follows the same calculation formula as that of mapping. Each array item is aligned to the 32-byte boundary. If the size of array item is less than 32 bytes, Solidity compiler will pack array automatically. In this way, it minimizes the count of using SSTORE opcode and therefore lower GAS fee.

To byte array, there are two cases:

- If the length of byte array is less than 31
  Array occupies only 1 slot and the other rules are the same with string.
- If the length of byte array is larger than or equal to 31

Byte array is equal to []byte. Address calculation formula is the same as string and mapping

5.5 Encode

Calldata is a byte array. How to interpret this byte array? That is why we need ABI since all calldata follow ABI. In this section, we will describe the fixed-length data type and dynamic data type in details.

5.5.1 A Simple Example

The following is a simple example which includes a getter and a setter for variable obj:

```
pragma solidity ^0.4.24;
contract example {
    uint256 obj;
    function setObj (uint256 para) public{
        obj = para;
    }
    function getObj () public returns (uint256)  {
        return obj;
    }
}
```

This contract is deployed to rinkeby testnet. We can use etherscan and search address 0x6f7cf32ceeb1285b3de93c21f5d229a9680145b9. We created a transaction to call setObj (100). And it can be checked at address 0xd5d0f93b3d4d127ea423c66f72d09c693823e5bab20411588123ca13851e08dc

The transaction output is

```
0x5f0f9b3e000000000000000000000000000000000
0000000000000000000000000000064
```

To EVM, it is 36 bytes metadata and EVM will not process metadata and pass it directly to smart contract as calldata. If contract is a Solidity program, then EVM will interpret these metadata as function call and execute appropriate assembly code for setObj (100).

Calldata can be divided into two parts:

Function selector(4byte)

```
0x5f0f9b3e
```

First parameter(32byte)

```
000000000000000000000000000000000000000000000000000000000064
```

First 4 bytes of calldata is function selector and the rest of calldata is the function parameters padded to 32 bytes. In the example above, there is only one parameter which is 0x1. Function selector is the kecccak256 hash value for function signature.

In the example, the function signature is setObj (uint256), which is function name and function parameter type. So totally, 36 bytes raw data is passed to contract as calldata.

5.5.2 An Example of External Call

External data is passed into the contract as calldata. Calldata is just a series of bytes. And EVM does not provide the parse method for calldata. We use the sample contract in Sect. 5.5.1 for analysis purpose. First, we start to compile the contract source code:

```
solc --bin --asm --optimize example.sol
```

The assembly code for contract body is under label sub_0:

```
sub_0: assembly {
... */ /* "example.sol":26:196 contract example {
    mstore(0x40, 0x80)
    jumpi(tag_1, lt(calldatasize, 0x4)) // revert if calldatasize
less than 4 bytes
    and(div(calldataload(0x0),
0x100000000000000000000000000000000000000000000000000000000),
0xffffffff)
    0x5f0f9b3e
    dup2
    eq
    tag_2
    jumpi
    dup1
    0xeb9ee256
    eq
    tag_3
    jumpi
   tag_1:
    0x0
    dup1
    revert
  ...
   auxdata:
0xa165627a7a72305820f7b5c295ee9b5e1280498
c5294ed8dda9ab726ad61de70eac0149177ceb805210029
```

Here is a brief description:

- mstore(0x40, 0x60), the first 64 bytes of memory is reserved for SHA3 hash calculation by EVM.
- auxdata is used to validate if the published code is the same as the deployed bytecode

Assembly code can be divided into two parts:

1. Invoke methods based on function selector
2. Pass function parameter, execute method and return

First, the following code is to match function selector

```
  // Load first 4 bytes function selector
  and(div(calldataload(0x0),
0x100000000000000000000000000000000000000000000000000000000),
0xffffffff)
  // if function selector is `0x5f0f9b3e`, then jump to setObj()
0x5f0f9b3e
dup2
eq  // if function selector equals to 0x5f0f9b3e(setObj), then jump
to Tag_2, or jump to  Tag_1
tag_2
jumpi
    // Does not find matched function, Fail & revert.
tag_1:
  0x0
  dup1
  revert
    // setObj body
tag_2:
  ...
```

The code above is pretty straightforward—extract first 4 bytes as the function selector. To better understand the code, we rewrite the logic using pseudo-code:

```
methodSelector = calldata[0:4]
if methodSelector == "0x5f0f9b3e":
      goto tag_2 // goto setObj
else:
      // if do not find function, then fail and revert.
      revert
```

The following code is the function body:

```
// setObj
tag_2:
  // Where to goto after method call
tag_3
      // Load first argument (the value 0x1).
      calldataload(0x4)
     // Execute method.
     jump(tag_4)
tag_4:
       0x0
       dup2
       swap1
       sstore
tag_5:
       pop
       jump
tag_3:
   // end of program
   stop
```

Before entering the function body, EVM needs to complete the following two tasks:

1. Save the return address for function
2. Put the parameters in calldata on the top of stack

In order for better understanding, we use the following pseudo-code to implement the same functionality above:

```
@returnTo = tag_3
tag_2: // setObj
  // Loads the arguments from call data onto the stack.
  @arg1 = calldata[4:4+32]
tag_4: // obj = para
  sstore(0x0, @arg1)
tag_5 // return
  jump(@returnTo)
tag_3:
  stop
```

Now, the following is the pseudo-code of the logic combining two parts above.

```
methodSelector = calldata[0:4]
if methodSelector == "0x5f0f9b3e":
  goto tag_2 // goto setObj
```

(continued)

```
else:
  // does not find matched method, revert
  revert
@returnTo = tag_3
tag_2: // setObj(uint256 para)
  @arg1 = calldata[4:36]
tag_4: // obj = para
  sstore(0x0, @arg1)
tag_5 // return
  jump(@returnTo)
tag_3:
  stop
```

5.5.3 *ABI Encode for External Method Call*

We use Pyethereum as the research tool. Please refer to Sect. 2.2.2.5 for how to install Pyethereum. We use Pyethereum for demo and validation purpose. We first need to launch Python3, and then import Pyethereum library under python console:

```
gavin@gavin-VirtualBox:~/pyethereum$ python3
Python 3.6.7 (default, Oct 22 2018, 11:32:17)
[GCC 8.2.0] on linux
Type "help", "copyright", "credits" or "license" for more
information.
>>> from ethereum.utils import sha3;
>>> sha3("setObj(uint256)").hex()
'5f0f9b3e66505667ec37d5a76816853fd768a083
935a34a68c6bf88a744457d9'
```

5.5.3.1 Fixed-Size Data Type

For fixed-length data types, such as uint256:

```
>>> from ethereum.abi import encode_abi;
>>> encode_abi(["uint256", "uint256", "uint256"],[1, 2, 3]).hex()
'0000000000000000000000000000000000000000
0000000000000000000000010000000000000000000000000
00000000000000000000000000000000000000020
0000000000000000000000000000000000000000000
00000000000000000003'
```

Above three data types are all uint256. Let us try the fixed data types with different size

```
>>> encode_abi(["int8", "uint32", "uint64"],[1, 2, 3]).hex()
'000000000000000000000000000000000000000000000000000000000
0000001000000000000000000000000000000000000000000000000
000000000000000200000000000000000000000000000000000
0000000000000000000000000003'
```

Pay attention that above data type like int8, uint32 whose length is less than 256 will be automatically aligned to 32 bytes (256 bit). For the fixed-size array:

```
>>> encode_abi(
...  ["int8[3]", "int256[3]"],
...  [[1, 2, 3], [4, 5, 6]]
... ).hex()
'000000000000000000000000000000000000000000000000000
00000000000010000000000000000000000000000000000000000000
00000000000000000000200000000000000000000000000000000000
00000000000000000000000000003000000000000000000000000000000
000000000000000000000000000000000400000000000000000000000
00000000000000000000000000000000000000005000000000000000
000000000000000000000000000000000000000000000000006'
```

5.5.3.2 Dynamic Data Type

For dynamic data type, Solidity ABI uses head-tail scheme: dynamic part will be put at the tail of calldata, while the parameter is put at the head of calldata, which points to the start of the dynamic part.

```
>>> encode_abi(
...  ["uint256[]", "uint256[]", "uint256[]"],
...  [[0xd1, 0xd2, 0xd3], [0xe1, 0xe2, 0xe3], [0xf1, 0xf2, 0xf3]]
... ).hex()
'000000000000000000000000000000000000000000
0000000000000000000060000000000000000000000000000000
00000000000000000000000000000000e0000000000000000000000000
0000000000000000000000000000000000000160000000000000000000
000000000000000000000000000000000000000000000300000000000
000000000000000000000000000000000000000000000000000d10000
0000000000000000000000000000000000000000000000000000000000
d200000000000000000000000000000000000000000000000000000000
```

(continued)

```
0000000d3000000000000000000000000000000000000000000000000
000000000000000300000000000000000000000000000000000000000
000000000000000000000e10000000000000000000000000000000000
0000000000000000000000000000e2000000000000000000000000000
00000000000000000000000000000000000e30000000000000000000
00000000000000000000000000000000000000000003000000000000
0000000000000000000000000000000000000000000000000f1000000
00000000000000000000000000000000000000000000000000000000f
2000000000000000000000000000000000000000000000000000000000
00000f3'
```

We re-format the output above since binary format above is a little clumsy.

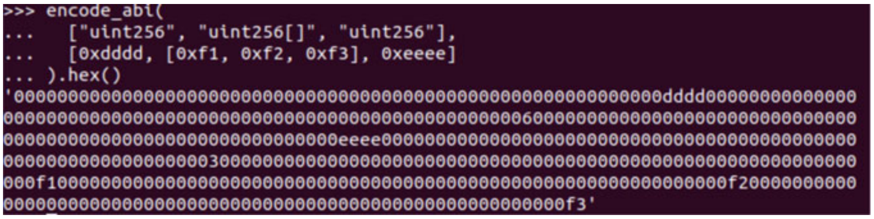
```
/************* HEAD (32*3 bytes) *************/
// arg1: Array data at 0x60
0000000000000000000000000000000000000000000000000000000000
000060
// arg2: Array data at 0xe0
000000000000000000000000000000000000000000000000000000
00000000e0
// arg3: Array data at 0x160
00000000000000000000000000000000000000000000000000000
00000000160
/************* TAIL (128**3 bytes) *************/
// Arg1 is saved at 0x60 including elements and array length
000000000000000000000000000000000000000000000000000000
0000000003
00000000000000000000000000000000000000000000000000000
000000000d1
00000000000000000000000000000000000000000000000000000
000000000d2
000000000000000000000000000000000000000000000000000000
00000000d3
// Arg2 is saved at 0xe0
00000000000000000000000000000000000000000000000000000
00000000003
00000000000000000000000000000000000000000000000000000
000000000e1
00000000000000000000000000000000000000000000000000000
000000000e2
00000000000000000000000000000000000000000000000000000
000000000e3
// Arg3 is saved at 0x160
00000000000000000000000000000000000000000000000000000
00000000003
00000000000000000000000000000000000000000000000000000
000000000f1
00000000000000000000000000000000000000000000000000000
```

(continued)

```
00000000000000000000000000000000000000000000000000000
000000000f2
00000000000000000000000000000000000000000000000000000
000000000f3
```

Let us see what will happen if we use fixed size and dynamic data type together: {static, dynamic, static} (Fig. 5.2).

```
/************* HEAD (32*3 bytes) *************/
// arg1: 0xdddd
000000000000000000000000000000000000000000000000000000
000000dddd
// arg2: look at position 0x60 for array data
00000000000000000000000000000000000000000000000000000
00000000060
// arg3: 0xeeee
00000000000000000000000000000000000000000000000000000
0000000eeee
/************* TAIL (128 bytes) *************/
// Address is 0x60. Data is at arg2 of 0x60
00000000000000000000000000000000000000000000000000000
00000000003
00000000000000000000000000000000000000000000000000000
000000000f1
00000000000000000000000000000000000000000000000000000
000000000f2
00000000000000000000000000000000000000000000000000000
000000000f3
```



```
>>> encode_abi(
...     ["uint256", "uint256[]", "uint256"],
...     [0xdddd, [0xf1, 0xf2, 0xf3], 0xeeee]
... ).hex()
'000000000000000000000000000000000000000000000000000000000000dddd00000000000000
000000000000000000000000000000000000000000000000600000000000000000000000000000
0000000000000000000000000000000eeee00000000000000000000000000000000000000000000
000000000000000000030000000000000000000000000000000000000000000000000000000000000
000f100000000000000000000000000000000000000000000000000000000000000f20000000000
0000000000000000000000000000000000000000000000000000f3'
```

Fig. 5.2 {static, dynamic, static} data presentation

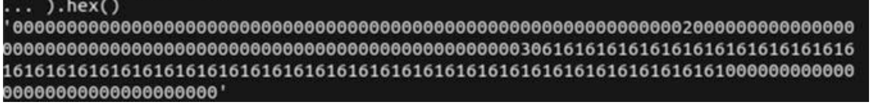
For string and byte array, Solidity also uses head-tail coding scheme:

```
// arg1: look at position 0x60 for string data
000000000000000000000000000000000000000000000000000000
0000000060
// arg2: look at position 0xa0 for string data
00000000000000000000000000000000000000000000000000000
000000000a0
// arg3: look at position 0xe0 for string data
0000000000000000000000000000000000000000000000000000
0000000000e0
// 0x60 (96). Data for arg1
000000000000000000000000000000000000000000000000000000
0000000004
616161610000000000000000000000000000000000000000000000
0000000000
// 0xa0 (160). Data for arg2
000000000000000000000000000000000000000000000000000000
0000000004
626262620000000000000000000000000000000000000000000000
0000000000
// 0xe0 (224). Data for arg3
0000000000000000000000000000000000000000000000000000000
000000004
636363630000000000000000000000000000000000000000000000
0000000000
```

If length of the string is larger than 32 bytes (Fig. 5.3).

```
// Arg1 indicates that array data start at 0x20
000000000000000000000000000000000000000000000000000000
0000000020
// Length of String is 0x30 (48)
0000000000000000000000000000000000000000000000000000
000000000030
```

(continued)



```
>>> encode_abi(
...   ["string"],
...   ["a" * (32+16)]
... ).hex()
'00000000000000000000000000000000000000000000000000000000000000200000000000000
000000000000000000000000000000000000000000000000306161616161616161616161616161616
16161616161616161616161616161616161616161616161616161616161616161616161000000000000
00000000000000000000'
```

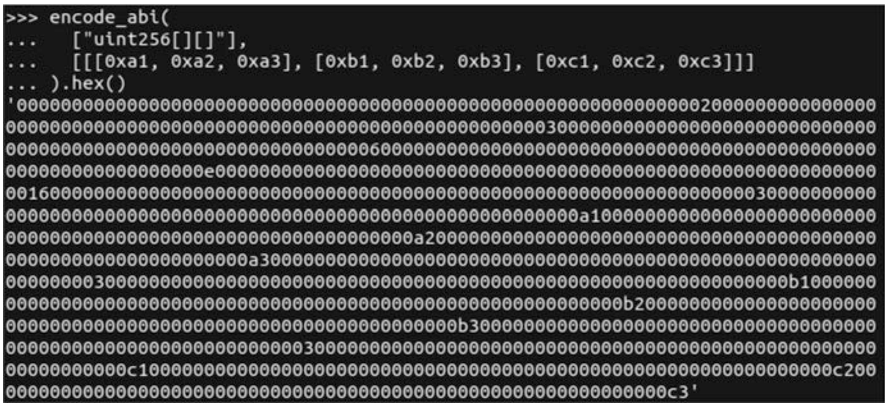
Fig. 5.3 Data presentation if string is larger than 32 bytes

```
61616161616161616161616161616161616161616161616161616
16161616161
616161616161616161616161616161610000000000000000000000
0000000000
```

The code of the nested data:

```
// arg1: The outter array is at position 0x20.
0000000000000000000000000000000000000000000000000000
000000000020
// 0x20. Each element is the position of an inner array.
0000000000000000000000000000000000000000000000000000
000000000003
0000000000000000000000000000000000000000000000000000
000000000060
0000000000000000000000000000000000000000000000000000
0000000000e0
0000000000000000000000000000000000000000000000000000
000000000160
// array[0] at 0x60
00000000000000000000000000000000000000000000000000000
00000000003
00000000000000000000000000000000000000000000000000000
000000000a1
00000000000000000000000000000000000000000000000000000
000000000a2
00000000000000000000000000000000000000000000000000000
000000000a3
// array[1] at 0xe0
00000000000000000000000000000000000000000000000000000
00000000003
00000000000000000000000000000000000000000000000000000
000000000b1
00000000000000000000000000000000000000000000000000000
000000000b2
00000000000000000000000000000000000000000000000000000
000000000b3
// array[2] at 0x160
0000000000000000000000000000000000000000000000000000
000000000003
0000000000000000000000000000000000000000000000000000
0000000000c1
0000000000000000000000000000000000000000000000000000
0000000000c2
0000000000000000000000000000000000000000000000000000
0000000000c3
```

EVM uses complement coding scheme to present negative numbers. So the presentation of -1 in EVM is (Fig. 5.4).



```
>>> encode_abi(
...     ["uint256[][]"],
...     [[[0xa1, 0xa2, 0xa3], [0xb1, 0xb2, 0xb3], [0xc1, 0xc2, 0xc3]]]
... ).hex()
'000000000000000000000000000000000000000000000000000000000000002000000000000000
0000000000000000000000000000000000000000000000000300000000000000000000000000000
0000000000000000000000000000000006000000000000000000000000000000000000000000000
000000000000000000e000000000000000000000000000000000000000000000000000000000000
0016000000000000000000000000000000000000000000000000000000000000000030000000000
0000000000000000000000000000000000000000000000000000a10000000000000000000000000
0000000000000000000000000000000000000a200000000000000000000000000000000000000000
0000000000000000000000a300000000000000000000000000000000000000000000000000000000
0000000030000000000000000000000000000000000000000000000000000000000000000b1000000
00000000000000000000000000000000000000000000000000000000b2000000000000000000000
00000000000000000000000000000000000000000b3000000000000000000000000000000000000
0000000000000000000000000003000000000000000000000000000000000000000000000000000
00000000000c100000000000000000000000000000000000000000000000000000000000000c200
000000000000000000000000000000000000000000000000000000000000c3'
```

Fig. 5.4 Data presentation for nested data

```
0xfffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffff
```

5.6 ABI Programming

We open any editor, create a file named MyToken.sol, and copy and paste the contract source code described in Sect. 4.3.1.2 into MyToken.sol. And then, open a command prompt, run the following command:

```
solcjs MyToken.sol --bin
```

After successful compilation, a MyToken_sol_MyToken.bin file will be generated. This file contains contract's bytecode. Next, we need to use solc to create ABI (Application Binary Interface). The generated file is a contract template used for interface containing available methods in the contract. Using the following command to generate ABI file:

```
solcjs MyToken.sol --abi
```

We should see a new json format file—MyToken_sol_MyToken.abi in which contract interface is defined. Last, we need to deploy it. Assume that we have testrpc/Ganache running on local and listening on port 8545:

```
//instance web3
Web3 = require('web3')
provider = new Web3.providers.HttpProvider("http://
localhost:8545")
web3 = new Web3(provider)
```

We use Web3 API to parse contract ABI and interact with the contract. Next, we need to deploy contract bytecode to testrpc:

```
// read file
myTokenABIFile = fs.readFileSync('MyToken_sol_MyToken.abi')
myTokenABI  = JSON.parse(myTokenABIFile.toString())
myTokenBINFile = fs.readFileSync('MyToken_sol_MyToken.bin')
myTokenByteCode = myTokenBINFile.toString()
// Deploy
account = web3.eth.accounts[0]
MyTokenContract = web3.eth.contract(myTokenABI)
contractData = { data: myTokenByteCode, from: account, gas: 999999
}
deployedContract = MyTokenContract.new(contractData)
```

At last, we call deployedContract to validate our deployed contract and save contract address because we will use it later. Solidity automatically maps state variable to memory slot. The mapping method is simple:

- Static size variable (all types except mapping and dynamic array) is mapped to memory slot starting from 0.
- For a dynamic array, memory slot saves the length of array and its data is saved to (keccak256(p)) slot.
- For mapping, the memory slot will not be used. For key K, its mapping value will be saved to slot at keccak256(k,p). Be careful: keccak256 (k,p) will be always aligned to 32 bytes.

ABI is the specification about how EVM interprets and executes contract bytecode. In this chapter, we know how EVM encodes and decodes calldata of different data type. We will switch focus to design complex smart contract in Solidity in next chapter.

Chapter 6
Operation Principles of Smart Contract

In this chapter, we will discuss some advanced topics of smart contract programming using Solidity, including:

- Design patterns
- How to save GAS costs
- Solidity inline assembly
- Operation principles of Solidity smart contract

Only by deeply understanding these topics can we write Solidity smart contracts on the Ethereum blockchain quickly, efficiently as well as cost effectively. Meanwhile, we will have a deeper understanding of the vulnerabilities and security programming in Solidity smart contracts. When we master the Solidity assembly language, we can also implement some functions that are not built-in Solidity.

6.1 Design Pattern

When we are writing a large-scale program system, design patterns are introduced to ensure the readability and maintainability. There are many patterns that software engineers are familiar with:

- Creational Patterns (e.g. Singleton, Factory, AbstractFactory, etc.)
- Behavioral Pattern (e.g. Observer, Visitor, Mediator, etc.)
- Structural Pattern (e.g. Bridge, Composite, Facade, etc.)

With the popularity of decentralized applications, some design patterns are also recognized widely in the Solidity community. In this sub-section, we will discuss some of the more mature patterns.

G. Zheng et al., *Ethereum Smart Contract Development in Solidity*,
https://doi.org/10.1007/978-981-15-6218-1_6

6.1.1 Contract Self-Destruction

Contract self-destruction is used to terminate a contract and remove the contract from the blockchain forever, which exists in the real world as well. One example is a loan contract. When the loan is fully paid off, the contract must be destroyed. Another example is bidding. After the bidding period ends, the contract must also be destroyed. When a contract is destroyed, two things will happen:

1. Transaction related to the contract will fail
2. The funds sent to the destroyed contract will be lost

Therefore, all references to the destroyed contract must be deleted. Moreover, when sending funds, call **Get ()** to confirm that the contract exists before sending.

The code snippet below is an example, in which **destroyContract ()** is used to destroy the contract itself. We also use the **ownerRestricted** modifier to ensure that only the owner of the contract can destroy it.

```
contract SelfDesctructionContract {
   public address owner;
   public string someValue;

    modifier ownerRestricted {
       require(owner == msg.sender);
     _;
   }
   // constructor
   function SelfDesctructionContract() {
      owner = msg.sender;
   }
   // setter
   function setSomeValue(string value){
      someValue = value;
   }
   // only the owner of the contract can destroy it
   function destroyContract() ownerRestricted {
     suicide(owner);
   }
}
```

6.1.2 Factory Contract

The Factory pattern is used to create and deploy sub-contracts. We can see these sub-contracts as assets, such as a house or a bicycle. Factory pattern stores all sub-contract's addresses for easy retrieval when they are needed. A common use case is buying and selling assets and tracking ownership changes. When selling an asset, you must add a **payable** modifier to the function.

```
contract AutoShop {
   address[] autoAssets;
   function createChildContract(string brand, string model)
public   payable {
     // check if the sent ether is enough to cover the car asset ...
     address newAutoAsset = new AutoAsset(brand, model, msg.
sender);
    autoAssets.push(newAutoAsset);
  }
function getDeployedChildContracts() public view returns
(address[]) {
    return autoAssets;
   }
}
contract AutoAsset {
   string public brand;
   string public model;
   address public owner;
   function AutoAsset(string _brand, string _model, address
_owner) public
   {
     brand = _brand;
     model = _model;
     owner = _owner;
   }
}
```

Address newCarAsset = new CarAsset (..) will trigger a transaction, deploys the sub-contract to the blockchain, and returns the contract address. Then, the contract addresses are stored in the array address [] CarAssets in contract AutoShop.

6.1.3 Name Registry

Imagine that you are writing a DApp for Ebay. You have written ClothesFactoryContract, GamesFactoryContract, BooksFactoryContract, etc., and recorded the address of these factory contracts in the DApp. If these contracts change frequently, you must track them. In this case, you can use the Name Registration Pattern to store the mappings between contract names and contract addresses, so that you can find the contract address according to the contract name. You can also track different versions.

```
contract NameRegistry {
    struct ContractDetails {
      address owner;
      address contractAddress;
      uint16 version;
    }
    mapping(string => ContractDetails) registry;
    function registerName(string name, address addr, uint16 ver)
returns (bool) {
    // versions should start from 1
    require(ver >= 1);
    ContractDetails memory info = registry[name];
    require(info.owner == msg.sender);
    // if the registry doesn’t exist, create it
    if (info.contractAddress == address(0)) {
      info = ContractDetails({
        owner: msg.sender,
        contractAddress: addr,
        version: ver
      });
    } else {
      info.version = ver;
      info.contractAddress = addr;
    }
    // modify the registry
    registry[name] = info;
    return true;
    }
    function getContractDetails(string name) constant returns
(address, uint16) {
    return (registry[name].contractAddress, registry[name].
version);
  }
}
```

With the example above, we can get the contract address and the specified version of the contract using getContractDetails (name).

6.1.4 Mapping Iterator

Mapping in Solidity does not support traversal, so Mapping Iterator is useful if you need to traverse it. However, because both storage and traversal on Ethereum cost Gas, and the cost increases when the size of the mapping grows, we shall avoid using the traversal function.

```
contract MappingIterator {
     mapping(string => address) elements;
     string[] keys;
     function put(string key, address addr) returns (bool) {
     bool exists = elements[key] != address(0)
     if (!exists) {
        keys.push(key);
     }
     elements[key] = addr;
     return true;
  }
  function getKeyCount() constant returns (uint) {
     return keys.length;
  }
  function getElementAtIndex(uint index) returns (address) {
     return elements[keys[index]];
  }
  function getElement(string name) returns (address) {
     return elements[name];
  }
}
```

6.1.5 *Withdrawal Pattern*

Imagine you are selling goods. It is possible that the goods are unqualified and returned by a buyer. In this case, you will have to refund the buyer. You can write a refund function to traverse all buyers, find those buyers who want to get a refund, and send the money to the addresses of those buyers using buyerAddress.transfer () or buyerAddress.send (). The difference between them is that transfer function will throw an exception when an error occurs and send () will not throw an exception, instead it set the return value to false. This feature of Send () is important, because most buyers are external accounts, but some buyers may be contract accounts. If the programmer of the contract account make a mistake in writing the fallback function and throw an exception, the traversal will be terminated. Then the transaction is completely rolled back, and no buyer will get the refund. In other words, they are blocked.

It is better and safer to use send () than to expose the withdrawFunds () function to the outside world. Therefore, the wrong contract account will not prevent other buyers from getting a refund.

```
contract WithdrawalContract {
   mapping(address => uint) buyers;
   function buy() payable {
      require(msg.value > 0);
      buyers[msg.sender] = msg.value;
   }
   function withdraw() {
   uint amount = buyers[msg.sender];
   require(amount > 0);
   buyers[msg.sender] = 0;
   require(msg.sender.send(amount));
  }
}
```

6.2 Save Gas Costs

Ethereum is known to some as a "World Computer." Any operation using its computing resources must pay a fee called GAS costs. The following web sites can be used to calculate GAS as costs:

- EthGasStation—Web site for estimating transaction costs and times.
  https://ethgasstation.info/
- Meter Petrometer—Calculates the daily gas used by a particular account.
  https://github.com/makerdao/petrometer
- CryptoProf—Smart contract gas consumption measurement tool.
  https://github.com/doc-ai/cryptoprof

GAS of all the opcodes is defined in Ethereum Yellow Book.[1] Web3 library package has a function named web3.eth.estimateGas() to estimate GAS needed for specific transaction.

```
web3.eth.estimateGas(callObject [, callback])
```

It performs a message call or transaction on the VM of local node, but does not broadcast the transaction as a block, and returns the amount of GAS being required. The parameter is of type web3.eth.sendTransaction, while other parameters are optional. The return value is the GAS consumed by the simulated call/exchange. Below is an example of a call:

[1]https://docs.google.com/spreadsheets/d/1n6mRqkBz3iWcOlRem_mO09GtSKEKrAsfO7Frgx18pNU/edit#gid=0.

```
  var result = web3.eth.estimateGas({
    to: "0xc4abd0339eb8d57087278718986382264244252f",
    data:
"0xc6888fa10000000000000000000000000000000000000000
0000000000000003"
  });
  console.log(result); //
"0x000000000000000000000000000000000000000000000000000
00000015"
```

Some of best practices to save Gas are explained in the following sections.

6.2.1 Mind the Data Types

In Ethereum smart contract development, it is preferred to use 256-bit variables, such as uint256 and bytes32. This may be counterintuitive at first, but considering the characteristics of Ethereum VM, it makes sense, because each memory slot has 256 bits. Therefore, if you store only one uint8 type variable, the EVM must pad it with 0 to align, which costs GAS. In addition, the EVM computing is based on uint256. If you use a non-uint256 variable, there will be one extra step to convert it to uint256. Please note in order to fill the entire storage slot, the size of your variables must be designed carefully. More details will be covered in Sect. 6.2.3.

6.2.2 Store Values in Byte-Encoded Form

A relatively cheaper way to store and read information is to include them directly in the smart contract as byte encoding. The disadvantage is that once the contract is deployed, the values cannot be changed. However, the GAS cost spent on initializing and storing data is greatly reduced. There are two possible ways to do this: (1) use constant when declaring variables and (2) hardcode variables.

```
uint256 public v1;
uint256 public constant v2;
function calculate() returns (uint256 result) {
   return v1 * v2 * 10000
}
```

In the example above, the variable **v1** is in the contract storage block and the variables **v2** and the constant **10000** are in the byte encoding of the contract. Reading the variable **v1** is implemented by SLOAD operation, which will consume 200 Gas.

6.2.3 Compressing Variables with the SOLC Compiler

When we want to store data permanently on the blockchain, the SSTORE opcode runs in the background. This is almost the most expensive command and will probably consume 20,000 GAS. Therefore, avoid to use it whenever possible. In structs, the amount of GAS consumed by the SSTORE operation can be reduced by rearranging the variables.

```
struct Data {
   uint64 a;
   uint64 b;
   uint128 c;
   uint256 d;
}
Data public data;
constructor(uint64 _a, uint64 _b, uint128 _c, uint256 _d) public {
   Data.a = _a;
   Data.b = _b;
   Data.c = _c;
   Data.d = _d;
}
```

All the variables in a struct can be sorted into 256-bit slots so that the EVM compiler can compress them later. In this example, the SSTORE operation is performed only twice, once to store a, b, and c, and the other one to store d. The same principle applies to variables other than struct. Keep in mind that putting multiple variables in the same slot is more economical than zero-padding. Remember to turn on the optimization switch for SOLC.

6.2.4 Compressing Variables Using Assembly Code

In general, we can compress the variables so that fewer SSTORE operations are performed. Such compressions can be done manually. The following code demonstrates how to compress 4 uint64 variables into a 256-bit memory slot:

```
function encode(uint64 _a, uint64 _b, uint64 _c, uint64 _d)
internal pure
  returns (bytes32 x) {
     assembly {
        let y := 0
        mstore(0x20, _d)
```

(continued)

```
            mstore(0x18, _c)
            mstore(0x10, _b)
            mstore(0x8, _a)
            x := mload(0x20)
        }
    }
```

When reading the contents of a variable, the compressed variable must be decoded. The following code snippet demonstrates the decoding logic.

```
function decode(bytes32 x) internal pure returns (uint64 a, uint64 b,
uint64 c, uint64 d) {
        assembly {
            d := x
            mstore(0x18, x)
            a := mload(0)
            mstore(0x10, x)
            b := mload(0)
            mstore(0x8, x)
            c := mload(0)
        }
    }
```

Comparing this method to the one earlier, you will find that this method is not only cheaper but also has two advantages:

1. Precision: With this method, we are actually doing a bitwise compression.
2. Read once: Because variables are stored in a storage slot, we only need to perform an import operation to get the values of all variables.

But why do we use the prior method? In the code, you will find that we use assembly to encode and decode variables, giving up the readability of the program. This could make the program very error-prone. In addition, because we must include encoding and decoding functions wherever needed, which greatly increases the cost of deployment. However, if you really want to save GAS, we can use the latter method. The more variables can be compressed, the more benefits and GAS savings you can get.

6.2.5 Concatenating Function Parameters

As we use encoding and decoding functions to optimize the process of reading and storing data, we can use the same mechanism to concatenate the parameters in a function call. This will reduce the load on the calling data. It might slightly increase the cost of execution, but with a reduction in the base cost, we can still save overall GAS costs.

This article[2] compares two function calls, one using parameter concatenation and the other not, which explains what happened behind the function execution. In the code example below, i*ts corresponding assembly code shows that the oldExam function incurs 4 "CALLDATALOAD" operations once it is called, and each "CALLDATALOAD" operation triggers one memory allocation operation in Ethereum, while the newExam function incurs only 1 instead.*

```
contract bitCompaction {
    function oldExam(uint64 a, uint64 b, uint64 c, uint64 d) public {
    }
    function newExam(uint256 packed) public {
    }
}
```

6.2.6 Using Merkle Proofs to Reduce Storage Load

The idea of Merkle proof is to prove the validity of a large amount of data using a small amount of data, which has its unique benefits. Let us look at an example. Suppose we want to store the transaction information of a car purchase, which contains 32 configurations. Creating 32 variables, each representing a configuration, is very expensive. In this situation, we can use the Merkel tree in the following manners.

Firstly, we divide the 32 configurations into groups; say, we find 4 groups, each containing 8 configurations. Secondly, we create a hash of the data in each group, and then divide the 4 hashes into, say 2 groups. We will repeat the process until there is only one hash, the Merkle root. Only the Merkle root will be uploaded to the chain, usually in the form of a 256-bit variable (keccak256) (Fig. 6.1).

We group all the information based on whether we need to use it, because all elements of each branch in the Merkle tree need to be verified automatically. This means that we only need one verification process. In the example above, assuming the car manufacturer gave you a wrong color, we can still easily identify this error (Fig. 6.2).

[2] https://medium.com/coinmonks/techniques-to-cut-gas-costs-for-your-dapps-7e8628c56fc9.

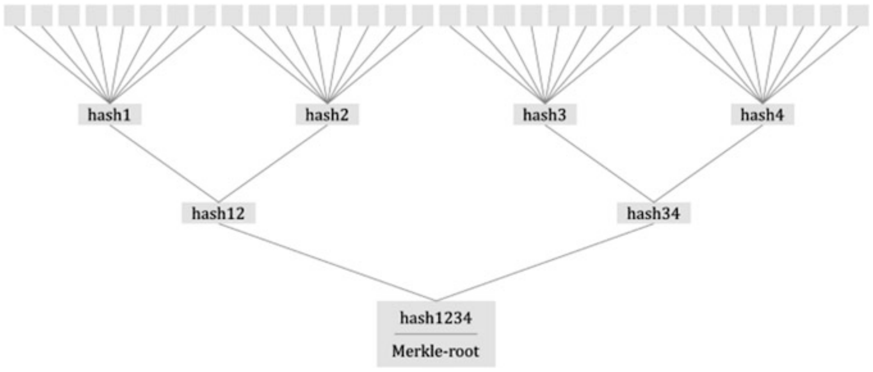


Fig. 6.1 Merkle tree

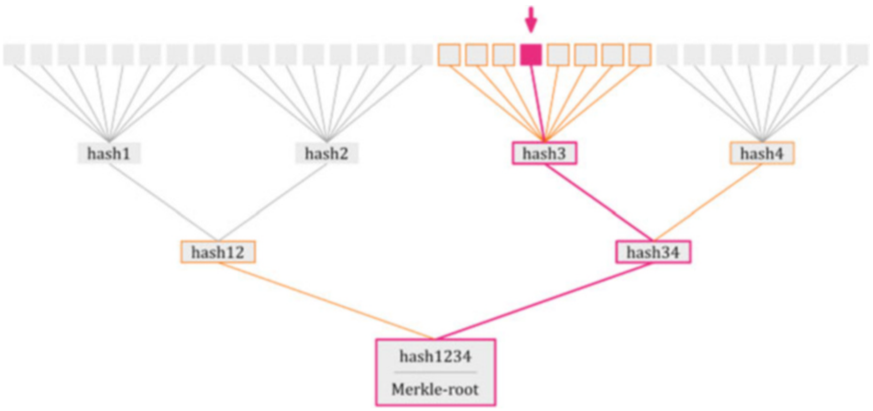


Fig. 6.2 Merkle tree verification

```
bytes32 public merkleRoot;

//Let a,...,h be the orange base blocks
function check
(
    bytes32 hash4,
    bytes32 hash12,
    uint256 a,
    uint32 b,
    bytes32 c,
    string d,
    string e,
```

(continued)

```
    bool f,
    uint256 g,
    uint256 h
)
    public view returns (bool success)
{
    bytes32 hash3 = keccak256(abi.encodePacked(a, b, c, d, e, f, g,
h));
    bytes32 hash34 = keccak256(abi.encodePacked(hash3, hash4));
    require(keccak256(abi.encodePacked(hash12, hash34)) ==
merkleRoot, "Wrong Element");

    return true;
}
```

If a variable is accessed or modified from time to time, traditional method is a better solution to store it instead. At the same time, we must make sure that every branch is not too large, which might exceed the depth of the transaction stack.

6.2.7 Stateless Contracts

Transaction data and event calls are stored on the blockchain. Therefore, using stateless contracts instead of constantly modifying the contract state, we only need to send transactions and values we want to modify. Because SSTORE operations cost most of the transaction GAS, stateless contracts use much less GAS than stateful contracts. More details about stateless contract can be found here.[3]

Let us apply the stateless contract scheme to the car transaction example above. We can send 1 or 2 transactions, depending on whether we can concatenate the parameters of the function and thus pass 32 configuration parameters about the car. As long as we only need to verify external information, this scheme is more appropriate and may be cheaper than Merkle proof scheme. Furthermore, accessing this information from the contract is basically impossible.

6.2.8 Storing Data on IPFS

The IPFS network is a decentralized data store, where each file is addressed by a hash of the content instead of a URL. Using its hash value to address a file not only prevent tampering the content, but also bring a huge advantage. We can upload data

[3]https://medium.com/@childsmaidment/stateless-smart-contracts-21830b0cd1b6.

on the IPFS network and store the corresponding hash value in the contract. A detailed explanation can be found here.[4]

As in stateless contract, this method is not using the data in the contract (possibly using Oracles). If you want to store a large amount of data, such as videos, it is best to store it on IPFS. (or Swarm, another decentralized storage system, which is an alternative to IPFS).

Because Sects. 6.2.6, 6.2.7, and 6.2.8 above are somehow similar, here is a comparison and summary:

- Merkel trees: small to medium data
  - Data can be used directly in the contract.
  - Modifying data is complicated
- Stateless contract: small to medium data
  - Data CANNOT be used directly into the contract
  - Data can be modified
- IPFS: large data
  - Using data in contracts is complicated
  - Modifying data is complicated

6.2.9 *Bit-Compaction*

Bit-compaction format for external function parameters will help cut gas costs because it reduces the size of the data sent to the Ethereum blockchain. At the same time, this solution also incurs some excess GAS costs to unpack the bits. Overall, the savings are greater than the excess costs. Take a look at the following code snippet:

```
pragma solidity ^0.4.21;
contract bitCompaction {
  function oldExam(uint64 a, uint64 b, uint64 c, uint64 d) public {
  }
  function newExam(uint256 packed) public {
  }
}
```

Their corresponding assembly code shows that the oldExam function has 4 “CALLDATALOAD” operations and each “CALLDATALOAD” operation will

[4] https://medium.com/@didil/off-chain-data-storage-ethereum-ipfs-570e030432cf.

trigger an Ethereum memory allocation operation; in comparison, the newExam function have only 1.

```
oldExam call data: ([]uint8) (len=132 cap=132) {
00000000  3e f2 62 fd 00 00 00 00  00 00 00 00 00 00 00 00
00000010  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00000020  00 00 00 01 00 00 00 00  00 00 00 00 00 00 00 00
00000030  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00000040  00 00 00 01 00 00 00 00  00 00 00 00 00 00 00 00
00000050  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00000060  00 00 00 01 00 00 00 00  00 00 00 00 00 00 00 00
00000070  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00000080  00 00 00 01
}
newExam call data: ([]uint8) (len=36 cap=36) {
00000000  83 ba 6e 5a 00 00 00 00  00 00 00 01 00 00 00 00
00000010  00 00 00 01 00 00 00 00  00 00 00 01 00 00 00 00
00000020  00 00 00 01
}
```

It can be seen that those uint64 parameters in the oldExam function are first converted to uint256. The GAS costs of the above two functions are 22235 and 21816, respectively, which means that the newExam function saves 419 gas costs. The more parameters, the more gas costs will be saved by using bit-compaction.

Please note that the full assembly code is too large to present here, so we only showed part of it above. But it can easily be generated using:

```
`solc—asm—optimize—optimize-runs 200 bitCompaction.sol`.
```

6.2.10 Batching

Batching can reduce gas cost because it can reduce common data processing. Consider the following code snippet:

```
Old: func once(uint256 header, uint256 val...) x N
New: func batch(uint256 header, uint256[] val... x N)
```

Executing the old function *N* times, we will process the common "header" field *N* times, as well as calling the function *N* times. However, using batching, the function is called only *once* and the common "header" field is processed only *once*. Therefore, it saves gas costs by reducing "CALLDATALOAD," memory allocation, and function call operations. The larger the *N*, the larger the savings.

6.2.11 *Reading and Writing Separation on Storage Struct*

Separating reading and writing on storage struct can reduce GAS costs in many ways. Let us take a look at the following code snippet:

```
pragma solidity ^0.4.21;
contract structWrite {
   struct Object {
     uint64 v1;
     uint64 v2;
     uint64 v3;
     uint64 v4;
   }
   Object obj;
   function StructWrite() public {
     obj.v1 = 1;
     obj.v2 = 1;
     obj.v3 = 1;
     obj.v4 = 1;
   }
   function oldExam(uint64 a, uint64 b) public {
     uint a0; uint a1; uint a2; uint a3; uint a4; uint a5; uint a6;
     uint b0; uint b1; uint b2; uint b3; uint b4; uint b5;
     obj.v1 = a + b;
     obj.v2 = a - b;
     obj.v3 = a * b;
     obj.v4 = a / (b + 1);
   }
   function setObject(uint64 v1, uint64 v2, uint64 v3, uint64 v4)
private {
     obj.v1 = v1;
     obj.v2 = v2;
     obj.v3 = v3;
     obj.v4 = v4;
   }
   function newExam(uint64 a, uint64 b) public {
     uint a0; uint a1; uint a2; uint a3; uint a4;
     uint b0; uint b1; uint b2; uint b3;
     setObject(a + b, a - b, a * b, a / (b + 1));
   }
}
```

When the compiler optimization switch is turned on, SOLC compiles the write operation of the storage struct variable for the example above in the following manner: the oldExam function will have 1 EVM “SSTORE” operation for each struct field, where “SSTORE” is the most expensive operation in terms of GAS costs. Because current SOLC implementation does not optimize the “SSTORE” operation if there is no enough stack space (note: stack space can hold 16 local variables only). If we use the “reading/writing separation” paradigm, there will be stack space available for the compiler to optimize the code, then only 1 "SSTORE" operation is generated in this case. This can be verified by looking at the compiled assembly code. This is only suitable for the current SOLC version and may change later. In the above example, gas costs for oldExam and newExam are 58140 and 27318, respectively, which means newExam costs about 30822 less. While using "reading/writing separation" strategy, if there is no enough stack space, the more fields in the struct, the more we can save on GAS costs.

6.2.12 uint256 and Direct Memory Storage

Calculation in SOLC is using uint256. Therefore, other types, such as uint8, must be converted when they are referenced, which would require extra GAS. In addition, direct memory access saves more than direct storage access, which also costs less GAS than accessing a struct pointer. So, here are some best practices:

```
uint8 data;                   => uint256 data;
uint256 val = storageData;  => uint256 memoryData = storageData;
(N Times)                        uint256 val = memoryData;
uint64 val = obj.v1;          => uint64 val = val1;
```

6.2.13 Assembly Optimization

While compiling SOLC code, make sure you turn on the compiler switch for GAS costs: SOLC "optimize-runs" to generate the best assembly code that can run on the EVM.

6.3 Assembly

We summarize the commonly used assembly operations and corresponding gas costs in the following table.

Operation	GAS	Illustration
ADD/SUB	3	Arithmetic operation
MUL/DIV	5	Arithmetic operation
ADDMOD/MULMOD	8	Arithmetic operation
AND/OR/XOR	3	Bitwise operation
LT/GT/SLT/SGT/EQ	3	Comparison operation

(continued)

Operation	GAS	Illustration
POP	2	Stack operation
PUSH/DUP/SWAP	3	Stack operation
MLOAD/MSTORE	3	Memory operation
JUMP	8	Unconditional Jump
JUMPI	10	Conditional Jump
SLOAD	200	Storage operation
SSTORE	5000/20000	Storage operation
BALANCE	400	Get account balance
CREATE	32000	Create new account
CALL	25000	Call operation

6.3.1 Stack

Unlike traditional x86 architecture which is based on registers instead of virtual stack, EVM is a stack-based machine. The maximum stack depth of EVM is 1024, where each item in the stack is 256 bits, i.e. the EVM itself operates on 256-bit words. The word size here is 32 bytes, and this design facilitates Keccak256 hash scheme and elliptic curve calculations. In order to access the stack directly, the EVM provides the following opcodes:

- POP remove item from the top of the stack
- PUSHn (n ranges from 1 to 32, e.g. PUSH2) put the next n-byte item into the stack
- DUPn (n ranges from 1 to 32, e.g. DUP32) duplicates the nth item in the stack
- SWAPn (n ranges from 1 to 32, e.g. SWAP4) swaps the 1st and nth items in the stack

6.3.2 Calldata

The Calldata is a read-only byte-addressable space where the data parameter of a transaction or call is held. Unlike the Stack, to use this data you have to specify an exact byte offset and number of bytes you want to read. The opcodes related to Calldata provided by EVM are

- CALLDATASIZE returns the size of transaction data
- CALLDATALOAD imports 32 bytes of transaction data onto the stack
- CALLDATACOPY copies transaction data of a certain number of bytes to memory

Solidity provides inline assembly corresponding to the above: calldatasize, calldataload and calldatacopy, respectively. For example, calldatacopy has three parameters (t, f, s), which will copy the *s* bytes of Calldata at position *f* to the memory at position *t*. At the same time, Solidity allows access to Calldata through msg.data. Let us take a look at the inline assembly statements in delegatecall:

```
assembly {
        let ptr := mload(0x40)
        calldatacopy(ptr, 0, calldatasize)
        let result := delegatecall(gas, _impl, ptr, calldatasize,
0, 0)
}
```

In order to delegate a call to the _impl address, msg.data must be sent over. We know that the delegatecall opcode operates on memory data, and we need to copy the Calldata to the memory. That is why we use calldatacopy to copy all Calldata to a memory position addressed by pointer ptr.

Let us take a look at the following code:

```
contract Calldata {
  function add(uint256 _a, uint256 _b) public view
  returns (uint256 result)
  {
  assembly {
    let a := mload(0x40)
    let b := add(a, 32)
    calldatacopy(a, 4, 32)
    calldatacopy(b, add(4, 32), 32)
    result := add(mload(a), mload(b))
  }
 }
}
```

The purpose of this function is to return the sum of 2 numbers _a and _b. It can be seen that we store the memory pointer of address 0x40 to the variable a, and the variable b is the 32 bytes after a. Then we use calldatacopy to store the first argument to a. Note that we are copying from the place where the calldata is shifted by 4. This is because the first 4 bytes are the function selector, which is used by EVM to determine which function to call. After that, we copy the second parameter to the memory space addressed by variable b. Finally, we just calculate the sum by loading a and b in the memory.

You can test it in the truffle console with the following command:

```
truffle(develop) > compilet
truffle(develop) > Calldata.new().then(i => calldata = i)
truffle(develop) > calldata.add(1, 6).then(r => r.toString())  // 7
```

6.3.3 Memory

Memory is a volatile, read-write, byte-addressable space. It is mainly used to store data during program execution, mostly to pass parameters to internal functions. The memory is cleared at the beginning of each message call. All address values are initialized to 0. Like Calldata, memory calculates addresses in bytes, but we can only read in the form of 32-byte words at a time. When we write a word into the memory that has not been used before, we say that the memory is "expanded." In this case, there are expansion costs in addition to the cost of the write operation itself. Expansion costs will increase linearly for the first 724 bytes, and then it will be quadratically. EVM provides 3 opcodes to operate memory:

- MLOAD loads a word from the memory to the stack
- MSTORE stores a word to the memory
- MSTORE8 stores a byte to the memory

Solidity also provides inline assemblies corresponding to these opcodes: mload, mstore, and mstore8.

Solidity always holds a free memory pointer at 0x40, which points to the address of the first unused memory word. That is why it is necessary to import the word in the inline assembly. The first 64 bytes of memory are reserved by the EVM. Using free memory pointers can ensure that our program does not overwrite the memory used by Solidity itself. For example, in the delegatecall example above, a free memory pointer is used to store calldata for later use, because delegatecall needs to import/export data from memory.

In addition, let us take a look at the bytes generated by the Solidity compiler. We will find that they all start with 0x6060604052 . . ., let us analyze the meaning of these byte codes:

```
PUSH1  : EVM opcode is 0x60
0x60   : The free memory pointer
PUSH1  : EVM opcode is 0x60
0x40   : Memory position for the free memory pointer
MSTORE : EVM opcode is 0x52
```

One must be very careful when using assembly to manipulate memory, otherwise it is very likely to overwrite the reserved space of the system.

6.3.4 Storage

Storage is a persistent read-write word-addressable space, where each contract stores its persistent information. It is a space of 2^{256} slots, where each slot is 32 bytes, using key-value mapping scheme. All positions are initialized to 0.

The GAS costs of saving data to storage is one of the highest of all EVM operations. Modifying a storage slot from 0 to a non-zero value incurs a gas cost of 20,000. Storing the same non-zero value or setting a non-zero value to zero costs 5,000. When a non-zero value is set to 0, 15,000 will be refunded.

EVM provides the following two opcodes to operate on storage:

- SLOAD loads a word from the storage into the stack.
- SSTORE stores a word to the storage.

Solidity's inline assembly also supports the two opcodes above.

Solidity automatically maps each defined state variable to a slot in the Storage. The strategy is simple. Static fixed-length variables (except mapping and dynamic arrays) are picked and placed on consecutive addresses starting from 0 in the storage. For dynamic arrays, its length will be stored in a slot (p), and its data will be stored in the slots addressed by a hash (keccak256 (p)). For mapping variables, the value corresponding to a key k is stored in keccak256 (k, p). Please note that the parameters in keccak256 (k, p) will be padded to 32 bytes.

Take a look at the storage contract below to examine how the Solidity storage strategy works:

```
contract Storage {
    uint256 public number;
    address public account;
    uint256[] private array;
    mapping(uint256 => uint256) private map;

    function Storage() public {
        number = 2;
        account = this;
        array.push(10);
        array.push(100);
        map[1] = 9;
        map[2] = 10;
    }
}
```

Let us open a truffle console to test its storage structure. First, we will compile and create a new contract instance:

```
truffle(develop) > compile
truffle(develop) > Storage.new().then(i => storage = i)
```

Then we can ensure that the address 0 holds a number 2 and the address 1 holds the address of the contract:

```
truffle(develop) > web3.eth.getStorageAt(storage.address, 0)
// 0x02
truffle(develop) > web3.eth.getStorageAt(storage.address, 1)
// 0x..
```

We can check that the storage position 2 holds the length of the array as follows:

```
truffle(develop) > web3.eth.getStorageAt(storage.address, 2)
// 0x02
```

Finally, we can check that the storage position 3 is unused and the mapping values are stored as we described above:

```
 truffle(develop) > web3.eth.getStorageAt(storage.address, 3)
 // 0x00
  truffle(develop) > mapIndex =
'00000000000000000000000000000000000000000000000000000000
0000003'
  truffle(develop) > firstKey = '00000000000000000000000000000000
00000000000000000000000000000001'
  truffle(develop) > firstPosition = web3.sha3(firstKey + mapIndex, {
encoding: 'hex' })
  truffle(develop) > web3.eth.getStorageAt(storage.address,
firstPosition)
// 0x09
   truffle(develop) > secondKey = '00000000000000000000000000000000
00000000000000000000000000000002'
   truffle(develop) > secondPosition = web3.sha3(secondKey +
mapIndex, { encoding: 'hex' })
   truffle(develop) > web3.eth.getStorageAt(storage.address,
secondPosition)
// 0x0A
```

6.4 Deconstruct Smart Contract

The following is a standard implementation of the ERC20 token contract. The program logic is simple. We use it as an example to deconstruct the implementation of smart contracts. Although this contract has an overflow vulnerability, we ignore it at the moment and focus on the implementation of contract, bytecode, and assembly.

```
pragma solidity ^0.4.24;
contract ERC20TokenContract {
  uint256 _totalSupply;
  mapping(address => uint256) balances;
  constructor(uint256 _initialSupply) public {
    _totalSupply = _initialSupply;
  balances[msg.sender] = _initialSupply;
 }
 function totalSupply() public view returns (uint256) {
  return _totalSupply;
 }
 function transfer(address _to, uint256 _value) public returns
(bool) {
   require(_to != address(0));
   require(_value <= balances[msg.sender]);
   balances[msg.sender] = balances[msg.sender] - _value;
   balances[_to] = balances[_to] + _value;
   return true;
 }
 function balanceOf(address _owner) public view returns (uint256)
{
   return balances[_owner];
 }
}
```

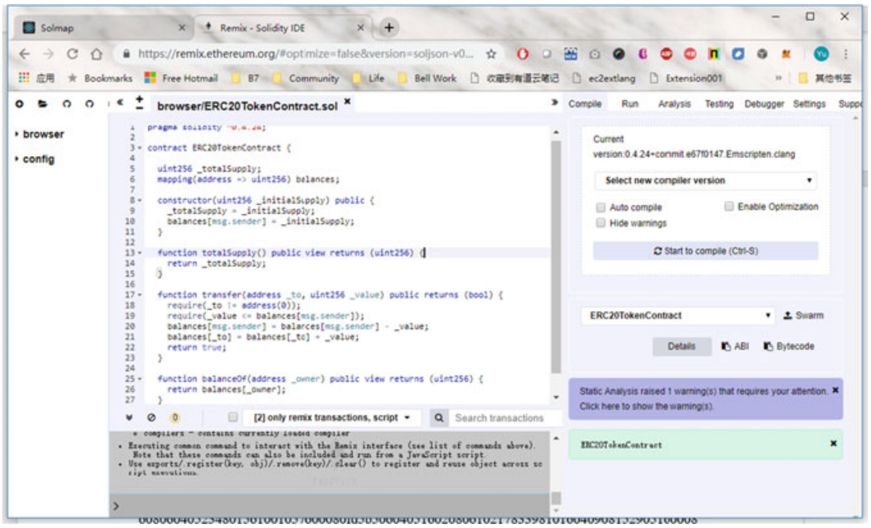
Let us copy the above program to Remix and compile. Note that compiler version 0.4.24 must be chosen (Fig. 6.3).

Let us click on the "Details" button to see information about the contract being generated (Fig. 6.4).

The Bytecode generated is as below:

```
608060405234801561001057600080fd5b5060405160208061 03ee8339810
18060405281019080805190602001909291905050508060008190555080 60
0160003373ffffffffffffffffffffffffffffffffffffffff1673fffffff
fffffffffffffffffffffffffffffffff1681526020019081526020016000
2081905550506103608061008e6000396000f300608060405260043610610
057576000357c0100000000000000000000000000000000000000000000000
000000000900463...
5600a165627a7a72305820 41e440ef41138511bd018bb2004da6344b9aeb2
49ba3cedf943e1e5d786bf67a0029
```

While the runtime bytecode is



```
pragma solidity ^0.4.24;

contract ERC20TokenContract {

  uint256 _totalSupply;
  mapping(address => uint256) balances;

  constructor(uint256 _initialSupply) public {
    _totalSupply = _initialSupply;
    balances[msg.sender] = _initialSupply;
  }

  function totalSupply() public view returns (uint256) {
    return _totalSupply;
  }

  function transfer(address _to, uint256 _value) public returns (bool) {
    require(_to != address(0));
    require(_value <= balances[msg.sender]);
    balances[msg.sender] = balances[msg.sender] - _value;
    balances[_to] = balances[_to] + _value;
    return true;
  }

  function balanceOf(address _owner) public view returns (uint256) {
    return balances[_owner];
  }
```

Current
version:0.4.24+commit.e67f0147.Emscripten.clang

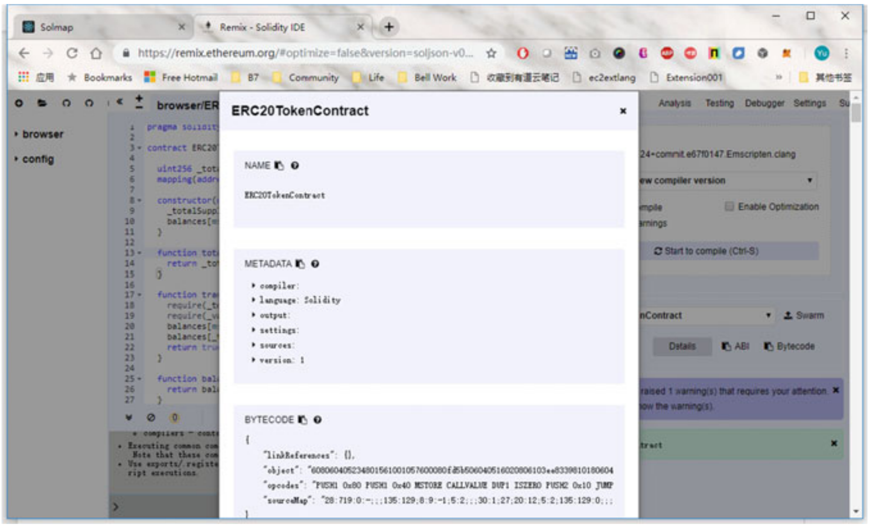
ERC20TokenContract

Static Analysis raised 1 warning(s) that requires your attention.
Click here to show the warning(s).

ERC20TokenContract

```
• compilers - contains currently loaded compiler
• Executing common command to interact with the Remix interface (see list of commands above).
  Note that these commands can also be included and run from a JavaScript script.
• Use exports/.register(key, obj)/.remove(key)/.clear() to register and reuse object across script executions.
```

Fig. 6.3 Create and deploy our example contract



ERC20TokenContract

```
▸ compiler:
▸ language: Solidity
▸ output:
▸ settings:
▸ sources:
▸ version: 1
```

```
{
    "linkReferences": {},
    "object": "608060405234801561001057600080fd5b506040516020806103ee8339810180604
    "opcodes": "PUSH1 0x80 PUSH1 0x40 MSTORE CALLVALUE DUP1 ISZERO PUSH2 0x10 JUMP
    "sourceMap": "28:719:0:-;;;135:129;8:9:-1;5:2;;;30:1;27;20:12;5:2;135:129:0;;;
}
```

Fig. 6.4 Deployed details our example contract

```
608060405260043610610057576000357c0100000000000000000000000000
0000000000000000000000000000900463ffffffff16806318160ddd14
61005c57806370a0823114610087578063a9059cbb146100de575b600080f
d5b34801561006857600080fd5b50610071610143565b...
fff1673ffffffffffffffffffffffffffffffffffffffff16815260200190
81526020016000208190555060019050929150505600a165627a7a7230582
041e440ef41138511bd018bb2004da6344b9aeb249ba3cedf943e1e5d786
bf67a0029
```

Comparing the 2 bytecodes above, we can see that the difference lies in the underlined part as highlighted. Actually, the following code is applicable to all contracts, which is imported by the contract itself.

```
608060405234801561001057600080fd5b506040516020806103ee8339810
1806040528101
9080805190602001909291905050508060008190555080600160003373fff
fffffffffffff
ffffffffffffffffffffffff1673fffffffffffffffffffffffffffffffff
fffffff168152
60200190815260200160002081905550506103608061008e6000396000
f300
```

The bytecodes above are completely unreadable, which is impossible for us to understand or analyze. Let us take a look at the readable assembly code (the description of the assembly instructions can be found here.[5]

```
.code
  PUSH 80               contract ERC20TokenContract {
...
  PUSH 40               contract ERC20TokenContract {
...
  MSTORE                contract ERC20TokenContract {
...
  CALLVALUE              constructor(uint256 _initialSu...
  DUP1                   olidity ^
  ISZERO                a
  PUSH [tag] 1          a
  JUMPI                 a
  PUSH 0                n
  DUP1                  \n
  REVERT                .24;
\n
```

(continued)

[5]https://github.com/ethereum/solidity/blob/develop/docs/assembly.rst#opcodes.

```
cont
tag 1                a
  JUMPDEST                    a
  POP                 constructor(uint256 _initialSu...
  PUSH 40                     constructor(uint256 _initialSu...
  MLOAD                       constructor(uint256 _initialSu...
  PUSH 20                     constructor(uint256 _initialSu...
  DUP1                        constructor(uint256 _initialSu...
  PUSHSIZE                    constructor(uint256 _initialSu...
  DUP4                        constructor(uint256 _initialSu...
 ...
```

The above assembly code is copied from Remix. You can also generate assembly code for smart contracts like this:

```
solc --asm --output-dir=build/binaries A.sol
```

A reference table of assembly and bytecode can be found in [40]. Here are some examples:

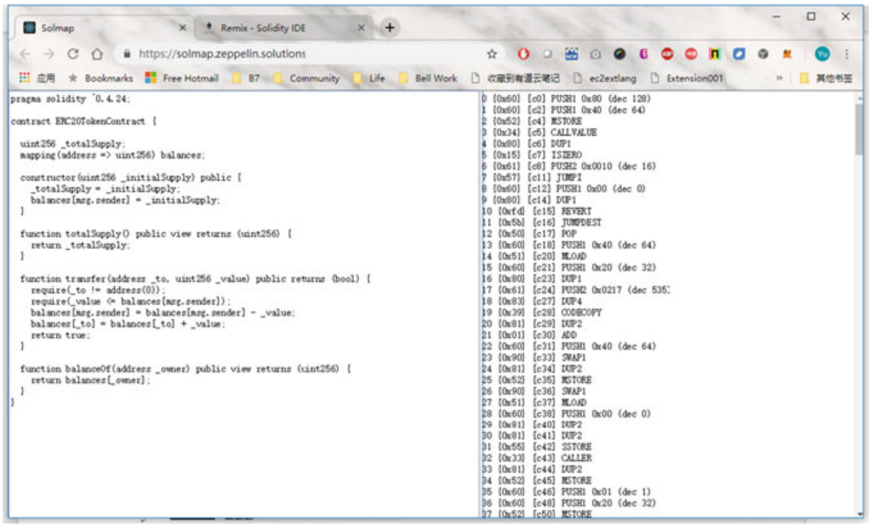
```
0x60 => PUSH
0x01 => ADD
0x02 => MUL
0x00 => STOP
...
```

In order to understand the code more directly, we copied the contract code to Solmap (https://solmap.zeppelin.solutions/). Please note that there is a slight difference between the code generated by Solmap and the code generated by Remix (Fig. 6.5).

For convenience of explanation, we use {...} to represent the stack, and {a, b} means that there are 2 elements in the stack, where the element on the top is a, and the bottom is b.

6.4.1 Contract Creation

Below is the generic binary code for contract creation/import. Every time a smart contract is imported, the compiler will inject the following code.



```
pragma solidity ^0.4.24;

contract ERC20TokenContract {

  uint256 _totalSupply;
  mapping(address => uint256) balances;

  constructor(uint256 _initialSupply) public {
    _totalSupply = _initialSupply;
    balances[msg.sender] = _initialSupply;
  }

  function totalSupply() public view returns (uint256) {
    return _totalSupply;
  }

  function transfer(address _to, uint256 _value) public returns (bool) {
    require(_to != address(0));
    require(_value <= balances[msg.sender]);
    balances[msg.sender] = balances[msg.sender] - _value;
    balances[_to] = balances[_to] + _value;
    return true;
  }

  function balanceOf(address _owner) public view returns (uint256) {
    return balances[_owner];
  }
}
```

```
0 {0x60} [c0] PUSH1 0x80 (dec 128)
1 {0x60} [c2] PUSH1 0x40 (dec 64)
2 {0x52} [c4] MSTORE
3 {0x34} [c5] CALLVALUE
4 {0x80} [c6] DUP1
5 {0x15} [c7] ISZERO
6 {0x61} [c8] PUSH2 0x0010 (dec 16)
7 {0x57} [c11] JUMPI
8 {0x60} [c12] PUSH1 0x00 (dec 0)
9 {0x80} [c14] DUP1
10 {0xfd} [c15] REVERT
11 {0x5b} [c16] JUMPDEST
12 {0x50} [c17] POP
13 {0x60} [c18] PUSH1 0x40 (dec 64)
14 {0x51} [c20] MLOAD
15 {0x60} [c21] PUSH1 0x20 (dec 32)
16 {0x80} [c23] DUP1
17 {0x61} [c24] PUSH2 0x0217 (dec 535;
18 {0x83} [c27] DUP4
19 {0x39} [c28] CODECOPY
20 {0x81} [c29] DUP2
21 {0x01} [c30] ADD
22 {0x60} [c31] PUSH1 0x40 (dec 64)
23 {0x90} [c33] SWAP1
24 {0x81} [c34] DUP2
25 {0x52} [c35] MSTORE
26 {0x90} [c36] SWAP1
27 {0x51} [c37] MLOAD
28 {0x60} [c38] PUSH1 0x00 (dec 0)
29 {0x81} [c40] DUP2
30 {0x81} [c41] DUP2
31 {0x55} [c42] SSTORE
32 {0x33} [c43] CALLER
33 {0x81} [c44] DUP2
34 {0x52} [c45] MSTORE
35 {0x60} [c46] PUSH1 0x01 (dec 1)
36 {0x60} [c48] PUSH1 0x20 (dec 32)
37 {0x52} [c50] MSTORE
```

Fig. 6.5 Solmap generated assembly code

```
608060405234801561001057600080fd5b5060405160208061 03ee8339810
1806040528101
90808051906020019092919050505080600081905550806001600033 73fff
fffffffffff
ffffffffffffffffffffff1673ffffffffffffffffffffffffffffffff
fffffff168152
6020019081526020016000208190555050610360806100 8e6000396000f
300
```

6.4.1.1 Get Free Memory Pointer

Take the first 5 bytes of the above byte code as an example: 6080604052. The bytecode for Push1 is 0x60, so:

```
6080 ==》 PUSH1 0x80
6040 ==》 PUSH1 0x40
52   ==》 MSTORE
```

After executing the second instruction, the stack status is like {0x40,0x80}. Then the third instruction can be interpreted as follows:

```
mstore(0x40, 0x80)
        |     |
        |    value
      address
```

The instruction above is to store value 0x80 to the address 0x40. As we explained above, 0x40 is the address of the free memory pointer (See Solidity's memory model in Sect. 5.1). So, the free memory starts at address 0x80 after this opcode. The code in the following sections is copied from Solmap.

6.4.1.2 Payable Check

The creation of a contract requires Ether/Wei. The following code is used to check whether this call comes with Wei/Ether.

```
3 {0x34} [c5] CALLVALUE
4 {0x80} [c6] DUP1
5 {0x15} [c7] ISZERO
6 {0x61} [c8] PUSH2 0x0010 (dec 16)
7 {0x57} [c11] JUMPI
8 {0x60} [c12] PUSH1 0x00 (dec 0)
9 {0x80} [c14] DUP1
10 {0xfd} [c15] REVERT
```

CALLVALUE pushes the Wei/Ether of this call to the stack, and DUP1 copies 1 byte of the current top element of the stack, then ISZERO determines whether Wei/Ether (that is, the current top element of the stack) is 0.

If Wei/Ether is not 0, ISZERO returns 0, and it executes sequentially, skip to [c16] to continue execution.

If wei/ether is 0, ISZERO returns 1, skip to [c12], set the return value to 0, and finally call REVERT to fall back.

The entire execution process can be executed step by step and verified in Remix. In short, the above assembly code implements the following statements:

```
if(msg.value != 0) revert();
```

6.4.1.3 Get Constructor Parameters

First, let us look at the JUMP instruction in assembly. The JUMP instruction fetches the value from the top of the stack and jumps to the address represented by that value. The destination must contain a JUMPDEST opcode, otherwise the JUMP operation will fail. The sole purpose of JUMPDEST is to mark an address as a legitimate jump destination. JUMPI is similar, but the second element on the top of the stack cannot be 0, otherwise the jump will not occur. So JUMPI is a conditional jump.

```
{0x5b} [c16] JUMPDEST
{0x50} [c17] POP
{0x60} [c18] PUSH1 0x40 (dec 64)
{0x51} [c20] MLOAD
{0x60} [c21] PUSH1 0x20 (dec 32)
{0x80} [c23] DUP1
{0x61} [c24] PUSH2 0x0217 (dec 535)
{0x83} [c27] DUP4
{0x39} [c28] CODECOPY
{0x81} [c29] DUP2
{0x01} [c30] ADD
{0x60} [c31] PUSH1 0x40 (dec 64)
{0x90} [c33] SWAP1
{0x81} [c34] DUP2
{0x52} [c35] MSTORE
{0x90} [c36] SWAP1
{0x51} [c37] MLOAD
```

C18-C20: Read the value at 0x40 and push it to the top of the stack. If you have not forgotten, 0x40 stores the free memory pointer {0x80,...}.

C23-C28: Before executing CODECOPY, the status of the stack is {0x80, 0x217, 0x20, 0x20, 0x80}, where 0x217 is the end of the contract body code. CODECOPY copies the 32 bytes at 0x217 to the free memory pointer (here 0x80). Because the constructor used to generate the contract is placed at the end of the code, in fact, the CODECOPY instruction copies the constructor parameters to memory 0x80. The constructor takes a uint256 parameter, which is exactly 32 bytes.

```
tag 1              a
  JUMPDEST         a
  POP              constructor(uint256 _initialSu...
  PUSH 40          constructor(uint256 _initialSu...
  MLOAD            constructor(uint256 _initialSu...
  PUSH 20          constructor(uint256 _initialSu...
  DUP1             constructor(uint256 _initialSu...
  PUSHSIZE         constructor(uint256 _initialSu...
  DUP4             constructor(uint256 _initialSu...
  CODECOPY         constructor(uint256 _initialSu...
```

In the code above, PUSHSIZE pushes the contract size onto the stack.

C29-C35: Modify the free memory pointer. MSTORE (0x40, 0xa0)

C36-C37: Push parameters onto the stack. This is a general pattern of Solidity EVM bytecode: push parameters onto the stack before calling a function.

6.4.1.4 Constructor Body

As can be seen in Solmap, the assembly code below represents the implementation of the following two Solidity statements.

```
_totalSupply = _initialSupply;
balances [msg.sender] = _initialSupply;
```

```
28 {0x60} [c38] PUSH1 0x00 (dec 0)
29 {0x81} [c40] DUP2
30 {0x81} [c41] DUP2
31 {0x55} [c42] SSTORE
```

C38-C42: Copy the value of the input parameter to the location of Storage 0, which is the location of _totalSupply. Before executing C42, the status of the stack {0x0, input parameter value of _initialSupply}

Below CALLER refers to msg.sender

```
32 {0x33} [c43] CALLER
33 {0x81} [c44] DUP2
34 {0x52} [c45] MSTORE
35 {0x60} [c46] PUSH1 0x01 (dec 1)
36 {0x60} [c48] PUSH1 0x20 (dec 32)
37 {0x52} [c50] MSTORE
38 {0x91} [c51] SWAP2
39 {0x90} [c52] SWAP1
40 {0x91} [c53] SWAP2
41 {0x20} [c54] SHA3
42 {0x55} [c55] SSTORE
```

6.4.1.5 Copy Contract Body Code to Memory

The following code pushes the size of the code (0x01d1) and the starting address of the code (0x0046) onto the stack, and finally calls CODECOPY to copy the code to address 0.

```
43 {0x61} [c56] PUSH2 0x01d1 (dec 465)
44 {0x80} [c59] DUP1
45 {0x61} [c60] PUSH2 0x0046 (dec 70)
46 {0x60} [c63] PUSH1 0x00 (dec 0)
47 {0x39} [c65] CODECOPY
```

6.4.1.6 Return Running Status

0x00 is the function status code. The following assembly language is to return the running status code.

```
48 {0x60} [c66] PUSH1 0x00 (dec 0)
49 {0xf3} [c68] RETURN
50 {0x00} [c69] STOP
```

6.4.2 General Part of Contract Body

6.4.2.1 Get Free Memory Pointer

The following code is familiar, which is to get the free memory pointer.

```
51 {0x60} [c70, r0] PUSH1 0x80 (dec 128)
52 {0x60} [c72, r2] PUSH1 0x40 (dec 64)
53 {0x52} [c74, r4] MSTORE
```

6.4.2.2 Calldata Check

```
54 {0x60} [c75, r5] PUSH1 0x04 (dec 4)
55 {0x36} [c77, r7] CALLDATASIZE
56 {0x10} [c78, r8] LT
57 {0x61} [c79, r9] PUSH2 0x0056 (dec 86)
58 {0x57} [c82, r12] JUMPI
```

C75-C78: Judges whether the size of Calldata is less than 4. If it is, jump to 0x56 using JUMPI (C82).

6.4.2.3 Function Selector

According to the source code above, this Contract has 3 member functions, except the constructor. The following assembly code is to determine which function package/wrapper to jump to according to the function selector.

```
   59 {0x63} [c83, r13] PUSH4 0xffffffff (dec 4294967295)
   60 {0x7c} [c88, r18] PUSH29
0x0100000000000000000000000000000000000000000000000000000000
(dec 2.695994666715064e+67)
   61 {0x60} [c118, r48] PUSH1 0x00 (dec 0)
   62 {0x35} [c120, r50] CALLDATALOAD
   63 {0x04} [c121, r51] DIV
   64 {0x16} [c122, r52] AND
```

First, push 4 bytes to the stack, then push 29 bytes to the stack, and then push 0x00 to the top of the stack. CALLDATALOAD then reads 32 bytes of data at Calldata address 0 to the top of the stack. Subsequent DIVs consume the top two elements of the stack and are used to obtain function selectors. Then the AND instruction consumes 2 more elements on the top of the stack to ensure that the 4-byte function selector is obtained. After execution, the top of the stack is a 4-byte function selector.

```
65 {0x63} [c123, r53] PUSH4 0x18160ddd (dec 404098525)
66 {0x81} [c128, r58] DUP2
67 {0x14} [c129, r59] EQ
68 {0x61} [c130, r60] PUSH2 0x005b (dec 91)
69 {0x57} [c133, r63] JUMPI
```

0x18160ddd is the function signature of totalSupply (). Compare the current function selector with 0x18160ddd. If they are equal, skip to 0x5b. Note r91, the function wrapper of totalSupply().

```
70 {0x80} [c134, r64] DUP1
71 {0x63} [c135, r65] PUSH4 0x70a08231 (dec 1889567281)
72 {0x14} [c140, r70] EQ
73 {0x61} [c141, r71] PUSH2 0x0082 (dec 130)
74 {0x57} [c144, r74] JUMPI
```

0x70a08231 is the function signature of balanceOf (). Compare the current function selector with 0x70a08231. If they are equal, skip to 0x82. Note r130, the function wrapper of balanceOf().

```
75 {0x80} [c145, r75] DUP1
76 {0x63} [c146, r76] PUSH4 0xa9059cbb (dec 2835717307)
77 {0x14} [c151, r81] EQ
78 {0x61} [c152, r82] PUSH2 0x00b0 (dec 176)
79 {0x57} [c155, r85] JUMPI
```

0xa9059cbb is the function signature of transfer (). Compare the current function selector with 0xa9059cbb. If they are equal, jump to 0xb0. Note r176, the function wrapper of transfer().

```
80 {0x5b} [c156, r86] JUMPDEST
81 {0x60} [c157, r87] PUSH1 0x00 (dec 0)
82 {0x80} [c159, r89] DUP1
83 {0xfd} [c160, r90] REVERT
```

If the current function selector cannot find a matching function, fall back to REVERT.

6.4.2.4 Function Wrapper

```
84 {0x5b} [c161, r91] JUMPDEST
85 {0x34} [c162, r92] CALLVALUE
86 {0x80} [c163, r93] DUP1
87 {0x15} [c164, r94] ISZERO
88 {0x61} [c165, r95] PUSH2 0x0067 (dec 103)
89 {0x57} [c168, r98] JUMPI
90 {0x60} [c169, r99] PUSH1 0x00 (dec 0)
91 {0x80} [c171, r101] DUP1
92 {0xfd} [c172, r102] REVERT
```

This is the Payable check. Because the totalSupply () function does not have a payable modifier, if there is ether, the function will REVERT, otherwise jump to 0x0067 (r103) to execute the function body. This is a general pattern of the Solidity compiler: check all functions without the Payable modifier for ether.

```
93 {0x5b} [c173, r103] JUMPDEST
94 {0x50} [c174, r104] POP
95 {0x61} [c175, r105] PUSH2 0x0070 (dec 112)
96 {0x61} [c178, r108] PUSH2 0x00f5 (dec 245)
97 {0x56} [c181, r111] JUMP
```

Here, decimal numbers 112 and 245 are pushed to the stack, and JUMP will jump directly to r245 to execute the function body (see Sect. 6.4.3.1). Meanwhile, the top of the stack is 112, which indicates that the execution of r112 is continued after the execution of the function body is completed.

```
98 {0x5b} [c182, r112] JUMPDEST
99 {0x60} [c183, r113] PUSH1 0x40 (dec 64)
100 {0x80} [c185, r115] DUP1
101 {0x51} [c186, r116] MLOAD
```

The free memory pointer is imported into the stack, and the status of the stack is {0x80, 0x40, totalSupply, func selector}

```
102 {0x91} [c187, r117] SWAP2
// after execution { totalSupply, 0x40, 0x80,func selector}
103 {0x82} [c188, r118] DUP3
// after execution { 0x80, totalSupply, 0x40, 0x80, func selector}
104 {0x52} [c189, r119] MSTORE // Store totalSupply at address 0x80
105 {0x51} [c190, r120] MLOAD
106 {0x90} [c191, r121] SWAP1
107 {0x81} [c192, r122] DUP2
108 {0x90} [c193, r123] SWAP1
109 {0x03} [c194, r124] SUB
110 {0x60} [c195, r125] PUSH1 0x20 (dec 32)
111 {0x01} [c197, r127] ADD
112 {0x90} [c198, r128] SWAP1
113 {0xf3} [c199, r129] RETURN
```

The totalSupply() function ends here. The Solmap disassembly of this paragraph is a little difficult to understand. Let us take a look at the corresponding Remix program:

```
117 DUP1
118 DUP3
119 DUP2
120 MSTORE
```

Store the value of initialSupply at the free memory pointer 0x80

```
121 PUSH1 20
123 ADD
124 SWAP2
125 POP
126 POP
127 PUSH1 40
129 MLOAD
130 DUP1
131 SWAP2
132 SUB
133 SWAP1
134 RETURN
```

Before executing the RETURN instruction, the stack status is {0x80,0x20,0x18160ddd}. Because initialSupply has been placed in memory address 0x80, the value returned is 0x80-initialsupply.

```
114 {0x5b} [c200, r130] JUMPDEST
115 {0x34} [c201, r131] CALLVALUE
116 {0x80} [c202, r132] DUP1
117 {0x15} [c203, r133] ISZERO
118 {0x61} [c204, r134] PUSH2 0x008e (dec 142)
119 {0x57} [c207, r137] JUMPI
120 {0x60} [c208, r138] PUSH1 0x00 (dec 0)
121 {0x80} [c210, r140] DUP1
122 {0xfd} [c211, r141] REVERT
123 {0x5b} [c212, r142] JUMPDEST
124 {0x50} [c213, r143] POP
125 {0x61} [c214, r144] PUSH2 0x0070 (dec 112)
126 {0x73} [c217, r147] PUSH20 0xffffffffffffffffffffffffffffff
ffffffffff (dec 1.461501637330903e+48)
127 {0x60} [c238, r168] PUSH1 0x04 (dec 4)
128 {0x35} [c240, r170] CALLDATALOAD
129 {0x16} [c241, r171] AND
130 {0x61} [c242, r172] PUSH2 0x00fb (dec 251)
131 {0x56} [c245, r175] JUMP
```

The balanceOf () function ends here.

```
132 {0x5b} [c246, r176] JUMPDEST
133 {0x34} [c247, r177] CALLVALUE
134 {0x80} [c248, r178] DUP1
135 {0x15} [c249, r179] ISZERO
```

(continued)

```
136 {0x61} [c250, r180] PUSH2 0x00bc (dec 188)
137 {0x57} [c253, r183] JUMPI
...
165 {0x03} [c309, r239] SUB
166 {0x60} [c310, r240] PUSH1 0x20 (dec 32)
167 {0x01} [c312, r242] ADD
168 {0x90} [c313, r243] SWAP1
169 {0xf3} [c314, r244] RETURN
```

The transfer() function ends here.

6.4.3 *Contract Bodies*

The following is the assembly code implemented by each function body. The following only deconstructs smart contracts, and does not discuss the assembly implementation of functions. But please pay attention to how the parameters are passed in and how the return value is passed back.

6.4.3.1 Body of the Function totalSupply()

```
170 {0x5b} [c315, r245] JUMPDEST
171 {0x60} [c316, r246] PUSH1 0x00 (dec 0)
172 {0x54} [c318, r248] SLOAD
173 {0x90} [c319, r249] SWAP1
174 {0x56} [c320, r250] JUMP
```

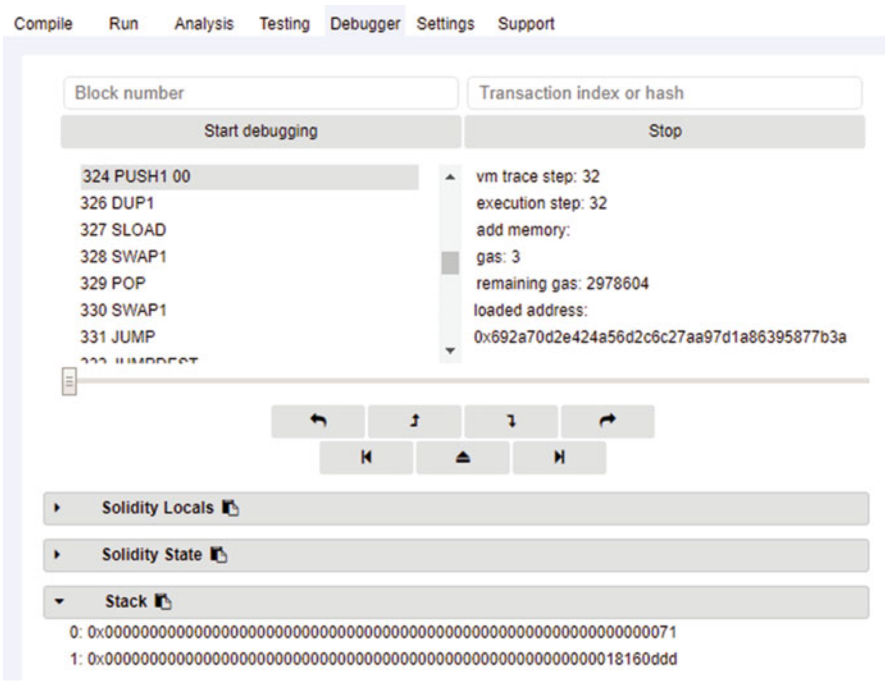
As previously analyzed, the value of _totalSupply is stored in Storage 0. SLOAD is to place the value of _totalSupply on the top of the stack {_totalSupply value, r112}. After SWAP1, the stack becomes {r112, _totalSupply value}. The last JUMP jumps back to r112 for execution (Fig. 6.6).

You can see it clearly from Remix:

In the figure above, 0x18160ddd is the function selector, and 0x71 is the return address 113, which indicates that jump back to r113.

6.4.3.2 Body of the Function balanceOf()

Below is an assembly implementation of the balanceOf () function. It should be easy to understand the logic through Remix.



```
324 PUSH1 00
326 DUP1
327 SLOAD
328 SWAP1
329 POP
330 SWAP1
331 JUMP
```

```
vm trace step: 32
execution step: 32
add memory:
gas: 3
remaining gas: 2978604
loaded address:
0x692a70d2e424a56d2c6c27aa97d1a86395877b3a
```

```
0: 0x0000000000000000000000000000000000000000000000000000000000000071
1: 0x0000000000000000000000000000000000000000000000000000000018160ddd
```

Fig. 6.6 JUMP opcode to go back

```
175 {0x5b} [c321, r251] JUMPDEST
176 {0x73} [c322, r252] PUSH20 0xffffffffffffffffffffffffffffff
fffffffff (dec 1.461501637330903e+48)
177 {0x16} [c343, r273] AND
178 {0x60} [c344, r274] PUSH1 0x00 (dec 0)
179 {0x90} [c346, r276] SWAP1
180 {0x81} [c347, r277] DUP2
181 {0x52} [c348, r278] MSTORE
182 {0x60} [c349, r279] PUSH1 0x01 (dec 1)
183 {0x60} [c351, r281] PUSH1 0x20 (dec 32)
184 {0x52} [c353, r283] MSTORE
185 {0x60} [c354, r284] PUSH1 0x40 (dec 64)
186 {0x90} [c356, r286] SWAP1
187 {0x20} [c357, r287] SHA3
188 {0x54} [c358, r288] SLOAD
189 {0x90} [c359, r289] SWAP1
190 {0x56} [c360, r290] JUMP
```

6.4.3.3 Body of the Function transfer()

Below is an assembly implementation of the transfer() function.

```
191 {0x5b} [c361, r291] JUMPDEST
192 {0x60} [c362, r292] PUSH1 0x00 (dec 0)
193 ...
263 {0x50} [c488, r418] POP
264 {0x50} [c489, r419] POP
265 {0x56} [c490, r420] JUMP
266 {0x00} [c491, r421] STOP
```

In order to design and implement complex smart contract, we discuss advanced topics: design patterns and GAS control. Also, we introduce assembly language just in case user needs to implement some functionality which is not built-in by Solidity. And to troubleshoot issues occurred in complex contract, we deconstruct a simple contract and let reader have a first-hand understanding of working mechanism of smart contract in Solidity. On top of it, we will present some best practices for upgradeable contract.

Chapter 7
Upgradable Contract

With the popularization of blockchain technology, most people know that data on blockchain is untampered and immutable. This is to say, contract deployed to blockchain is not updateable. However, in the real world, we may need to modify contract code due to various reasons: fixing bug, modifying business logic, upgrading functionality, etc. After years of discussion and practice, it becomes mainstream to implement contract upgrade by using Proxy/Delegator/Dispatcher pattern.

7.1 Solution

Generally speaking, the most popular design for an upgradeable contract can be classified into four categories:

- Proxy contracts
- Separate logic and data
- Separate logic and data through key-value pair
- Partially upgrade

7.1.1 Proxy Contracts

The main thought of proxy contract is to call the function in target contract through delegatecall opcode and target contract is upgradeable. Since delegatecall keeps working on the context of caller function, the target contract code is able to update the status of Proxy contract and keep the result in proxy contract. With the Byzantine hard fork, now we can get the return data and its size of a function call.

G. Zheng et al., *Ethereum Smart Contract Development in Solidity*,
https://doi.org/10.1007/978-981-15-6218-1_7

7.1.2 Separate Logic and Data Contracts

The main idea is to put all contract data (variable, struct, mapping, etc.) and its related getter and setter function into data contract; and put all business logic related code (possibly modify contract data) into a logic contract. So, even though business logic changes, data is still in the same location. This method allows the full upgrade of logic contract. Contract can ask users to use new logic contract (through ENS parser), and adjust data privilege to run getter and setter function.

7.1.3 Separate Logic and Data Through Key-Value

This method is similar to Sect. 7.1.2. The only difference is data access is abstracted and is through key-value pair via sha256 hash algorithm and standardized naming system.

7.1.4 Partially Upgradeable Strategies

Creating a fully upgradable contract could cause a severe trust problem: immutability of contract. So in many situations, the partial upgradeable design is also popular. Partial upgrade means that core functionality is not upgradeable, while other parts are upgradeable. The following are some famous examples which take partial upgrade strategy:

- Ethereum Name Service "ENS":

  ENS contract is a simple contract and un-changeable. Domain registration contract is a factory contract. Domain registration can be upgraded by administrators. Registration contract for ".eth" is a factory contract. When switching to a new domain manager, the contract is upgraded by linking to the old contract.
- 0x Project:

  Decentralized Exchange contract can be upgraded completely, while keeping proxy contract unchanged. 0x "proxy" contract contains funds and setting and this contract is in-upgradeable since the contract needs absolute trust.

7.1.5 Comparison

Here are simple comparisons of four upgrade strategies.

Strategies	Advantages	Disadvantages
Proxy contract	Upgradeable contracts do not need to be redesigned to work with this strategy	Contract code of the proxy will not reflect the state that it stores Cannot change the fields of the target contract but new fields can be added
Separate logic and data	Data can be read from the data contract normally The data structure that is used in the data contract can be updated and added to	Contracts need to be separated into data and logic contracts Complex data types can be changed but in a complex way
Separate logic and data contracts with data as a key-value pairs	Key-value pairs are more generic and simpler The data structure that is used in the data contract can be updated and added to	Contracts need to be separated into data and logic contracts Accessing data is very abstract as they are stored in key-value pairs
Partially upgradeable contract system	Leaves simple parts of the contract system as immutable to retain trust	Non-upgradeable contract code can never be upgraded

7.1.6 Simple Proxy Contract

A proxy contract is to dispatch function calls to target contract by using delegatecall opcode and all target contracts are upgradeable. The main idea of Proxy contract is a storage contract, a registry contract, and an implementation contract. Whenever we need to add new functionality or upgrade contract, we only need to develop a new implementation contract inherited from the old contract (Fig. 7.1).

Storage contract keeps all state variables:

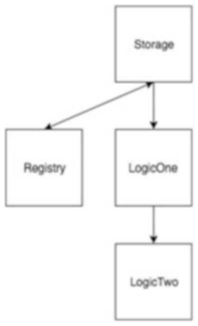


Fig. 7.1 Contract hierarchy

```
pragma solidity ^0.4.21;
contract Storage {
    uint public val;
}
```

Registry contract is a proxy to implementation contract.

```
pragma solidity ^0.4.21;
import './Ownable.sol';
import './Storage.sol';

contract Registry is Storage, Ownable {

    address public logic_contract;

    function setLogicContract(address _c) public onlyOwner
returns (bool success){
         logic_contract = _c;
         return true;
    }
    function () payable public {
         address target = logic_contract;
         assembly {
             let ptr := mload(0x40)
             calldatacopy(ptr, 0, calldatasize)
             let result := delegatecall(gas, target, ptr,
calldatasize, 0, 0)
             let size := returndatasize
             returndatacopy(ptr, 0, size)
             switch result
             case 0 { revert(ptr, size) }
             case 1 { return(ptr, size) }
         }
    }
}
```

We could use setLogicContract to let registry contract know proxying requests to which contract. Ownable.sol is used to ensure that only owner of the contact can call setLogicContract. We use assembly code in fallback function which allows an external contract to update its internal state. Please pay attention to the order of inheritance: we must initialize storage contract first and then initialize Ownable contract.

Then, let us have a look at the implementation contract.

```
pragma solidity ^0.4.21;
import './Storage.sol';

contract LogicOne is Storage {
    function setVal(uint _val) public returns (bool success) {
        val = 2 * _val;
        return true;
    }
}
```

LogicOne contract is to update "val" in storage contract. Steps are as below:

1. First, deploy Registry.sol and LogicOne.sol.
2. Register LogicOne address into registry kept in Registry.sol.

```
Registry.at(Registry.address).setLogicContract(LogicOne.
address)
```

3. Through LogicOne ABI, using contract address registered in Registry, we can update "Val" in Storage contract.

```
LogicOne.at(Registry.address).setVal(2)
```

4. When we want to upgrade contract: we first deploy LogicTwo contract and update Registry contract, making Registry address pointing to the newest implementation.

```
Registry.at(Registry.address).setLogicContract(LogicTwo.
address)
```

5. Now, we can use LogicTwo ABI to access storage.

```
LogicTwo.at(Registry.address).setVal(2)
```

7.2 Generic Proxy to Ethereum Call

We are going to introduce a more common and transparent proxy pattern based on EVM assembly. And proxy pattern is founded on library model:

1. First, instantiate an instance of Contract A—objA.
2. Create and instantiate a light contract—ALite to call contract A. ALite and A have the same function signature. But ALite uses callcode/delegatecall to call objA.

   This method costs some GAS:

1. Cost for using callcode/delegatecall (minimal).
2. Encode/decode parameter.

   When a method is called, all parameters in msg.data are encoded based on parameter types. So when contract A calls a method in contract B, contract B needs to decode msg.data.
3. Since contract B has the same structure as contract A, contract B could be very large caused by growth of contract A.

The following method based on assembly is more transparent: Proxy contract does not need to know the function signature of the target contract. The key point is fallback function in proxy contract: fallback function will redirect requests with msg. data to target contract through using callcode or delegatecall.

```
contract proxy{
         /*
           Member data must has the same structure as in target
contract.
           That is to say, "address add" must be at Slot 0
        */
        address add;
        function proxy(address a){
                  add = a;
        }
        function (){
                  assembly{
                             //gas must be uint
                             let g := and(gas,0xEFFFFFFF)
                             let o_code := mload(0x40) // Get Free
Memory Pointer
                             // 20bytes of Address
                             // sload(0) get Address at Slot 0
                             let addr := and(sload
(0),0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF) //Dest address
                             // Get calldata (function signature
and parameter)
                              calldatacopy(o_code, 0,
calldatasize)
```

(continued)

```
                                    // Could use callcode/delegatecall/
call
                                    let retval := call(g
                                                    , addr // Target contract
address
                                                    , 0 // value
                                                    , o_code //calldata
                                                    , calldatasize
//calldata size
                                                    , o_code //return
address
                                                    , 32) //32 byte return
data size

                                    // Check return value
                                    // 0 == if return 0 then jump to bad
destination (02)
                                    jumpi(0x02,iszero(retval))
                                    return(o_code,32)
                        }
            }
}
```

Steps to follow:

1. Instantiate contract A at <addressA>.
2. Instantiate contract B at <addressB>, and pass <addressA> to contract B's constructor.
3. Use contract A's ABI to call function at <addressB>.

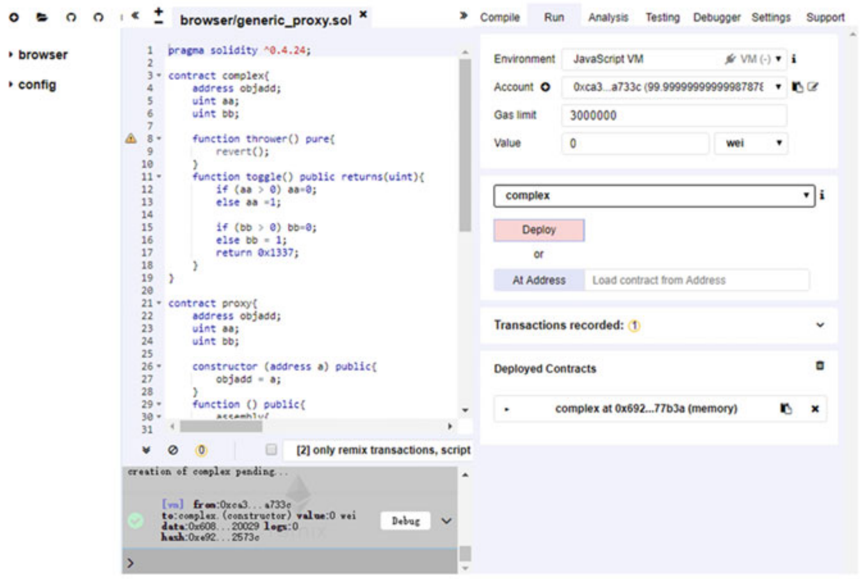
Let us check it in Remix (Fig. 7.2).

First let us deploy a "complex" contract. We get the deployed address at 0x692a70d2e424a56d2c6c27aa97d1a86395877b3a. And then we pass the address to proxy contract as the parameters of constructor. Then deploy proxy contract (Fig. 7.3).

And map complex contract to proxy contract's address 0xbbf289d846208c16edc8474705c748aff07732db (Fig. 7.4).

At this moment, EVM thinks the contract at 0xbbf289d846208c16edc8474705c748aff07732db is a complex contract, rather than a proxy contract. In this situation, calling toggle function in complex contract calls fallback function since it is a proxy contract and there is no such toggle function in proxy contract. And fallback function calls toggle function in complex contract through call/delegatecall (Fig. 7.5).

Calling toggle function generates a transaction on the bottom left part of screen capture above. Click "debug" button on the right side will jump to "debugger" tab. As can be seen, the code stops at entrance of toggle function. In this way, we can upgrade complex contract and we do not need to modify the code to call upgraded complex contract.



```
pragma solidity ^0.4.24;

contract complex{
    address objadd;
    uint aa;
    uint bb;

    function thrower() pure{
        revert();
    }
    function toggle() public returns(uint){
        if (aa > 0) aa=0;
        else aa =1;

        if (bb > 0) bb=0;
        else bb = 1;
        return 0x1337;
    }
}

contract proxy{
    address objadd;
    uint aa;
    uint bb;

    constructor (address a) public{
        objadd = a;
    }
    function () public{
        assembly{
```

[2] only remix transactions, script

```
creation of complex pending...

[vm] from:0xca3...a733c
to:complex.(constructor) value:0 wei
data:0x608...20029 logs:0
hash:0xe92...2573c
```

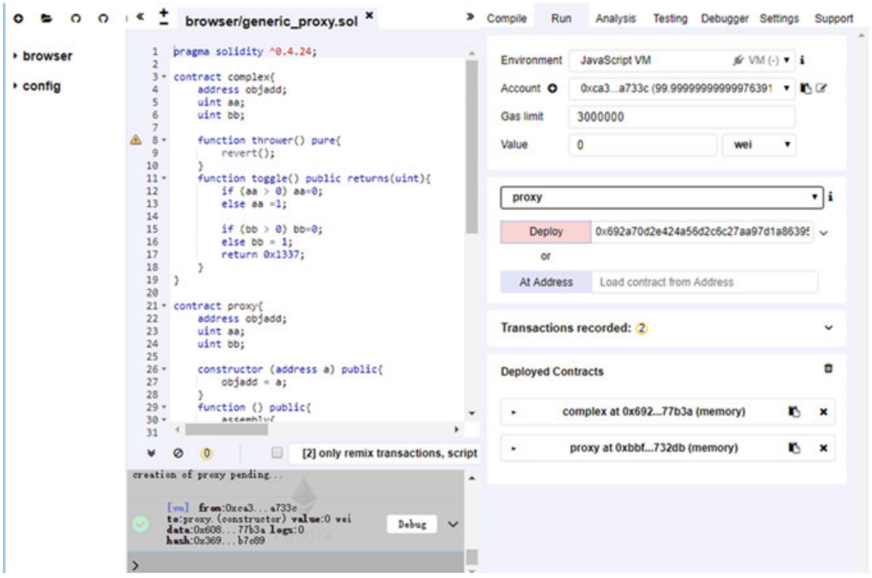
Environment: JavaScript VM
Account: 0xca3...a733c (99.9999999999998787
Gas limit: 3000000
Value: 0 wei

complex
Deploy
or
At Address: Load contract from Address

Transactions recorded: 1

Deployed Contracts
complex at 0x692...77b3a (memory)

Fig. 7.2 Deploy complex contract



```
pragma solidity ^0.4.24;

contract complex{
    address objadd;
    uint aa;
    uint bb;

    function thrower() pure{
        revert();
    }
    function toggle() public returns(uint){
        if (aa > 0) aa=0;
        else aa =1;

        if (bb > 0) bb=0;
        else bb = 1;
        return 0x1337;
    }
}

contract proxy{
    address objadd;
    uint aa;
    uint bb;

    constructor (address a) public{
        objadd = a;
    }
    function () public{
        assembly{
```

[2] only remix transactions, script

```
creation of proxy pending...

[vm] from:0xca3...a733c
to:proxy.(constructor) value:0 wei
data:0x608...77b3a logs:0
hash:0x369...b7e89
```

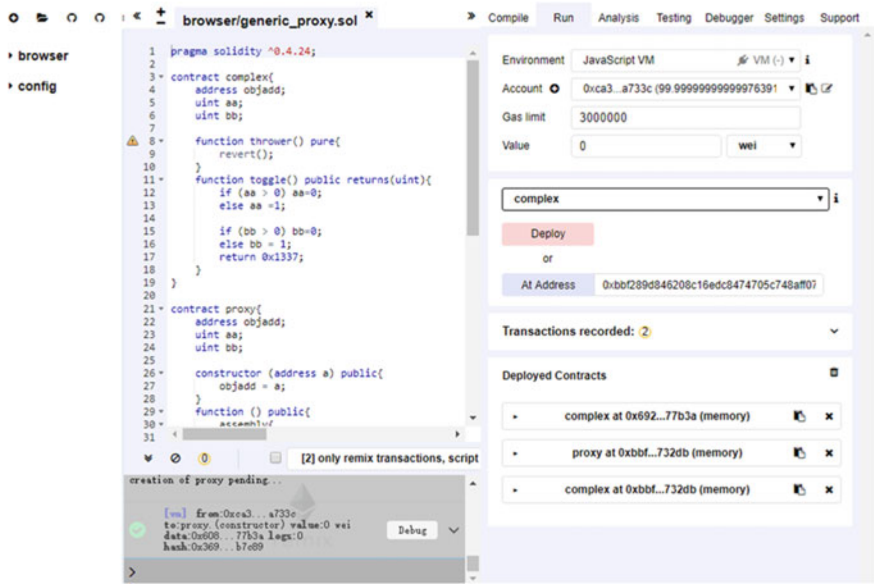
Environment: JavaScript VM
Account: 0xca3...a733c (99.99999999999976391
Gas limit: 3000000
Value: 0 wei

proxy
Deploy 0x692a70d2e424a56d2c6c27aa97d1a8639
or
At Address: Load contract from Address

Transactions recorded: 2

Deployed Contracts
complex at 0x692...77b3a (memory)
proxy at 0xbbf...732db (memory)

Fig. 7.3 Deploy proxy contract



```
pragma solidity ^0.4.24;

contract complex{
    address objadd;
    uint aa;
    uint bb;

    function thrower() pure{
        revert();
    }
    function toggle() public returns(uint){
        if (aa > 0) aa=0;
        else aa =1;

        if (bb > 0) bb=0;
        else bb = 1;
        return 0x1337;
    }
}

contract proxy{
    address objadd;
    uint aa;
    uint bb;

    constructor (address a) public{
        objadd = a;
    }
    function () public{
```

```
[2] only remix transactions, script
creation of proxy pending...

[vm] from:0xca3...a733c
to:proxy.(constructor) value:0 wei
data:0x608...77b3a logs:0
hash:0x369...b7c89
```

Environment: JavaScript VM — VM (-)
Account: 0xca3...a733c (99.99999999999976391
Gas limit: 3000000
Value: 0 wei

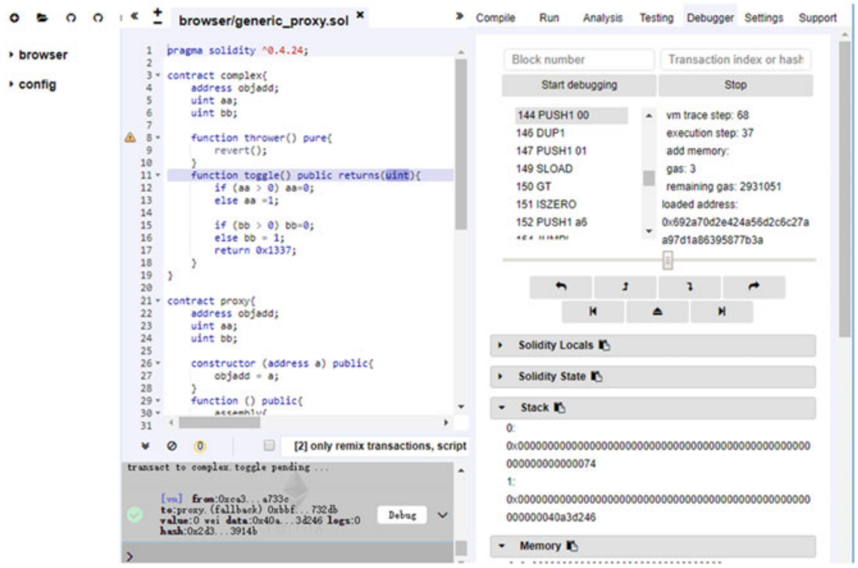
complex

At Address: 0xbbf289d846208c16edc8474705c748aff07

Transactions recorded: 2

Deployed Contracts
- complex at 0x692...77b3a (memory)
- proxy at 0xbbf...732db (memory)
- complex at 0xbbf...732db (memory)

Fig. 7.4 Map complex contract to proxy contract



```
pragma solidity ^0.4.24;

contract complex{
    address objadd;
    uint aa;
    uint bb;

    function thrower() pure{
        revert();
    }
    function toggle() public returns(uint){
        if (aa > 0) aa=0;
        else aa =1;

        if (bb > 0) bb=0;
        else bb = 1;
        return 0x1337;
    }
}

contract proxy{
    address objadd;
    uint aa;
    uint bb;

    constructor (address a) public{
        objadd = a;
    }
    function () public{
```

```
[2] only remix transactions, script
transact to complex.toggle pending ...

[vm] from:0xca3...a733c
to:proxy.(fallback) 0xbbf...732db
value:0 wei data:0x40a...3d246 logs:0
hash:0x2d3...3914b
```

```
144 PUSH1 00        vm trace step: 68
146 DUP1            execution step: 37
147 PUSH1 01        add memory:
149 SLOAD           gas: 3
150 GT              remaining gas: 2931051
151 ISZERO         loaded address:
152 PUSH1 a6       0x692a70d2e424a56d2c6c27a
                   a97d1a86395877b3a
```

```
Stack
0:
0x00000000000000000000000000000000000000000000000000
00000000000074
1:
0x0000000000000000000000000000000000000000000000000
000000040a3d246
```

Fig. 7.5 Fallback function call toggler function

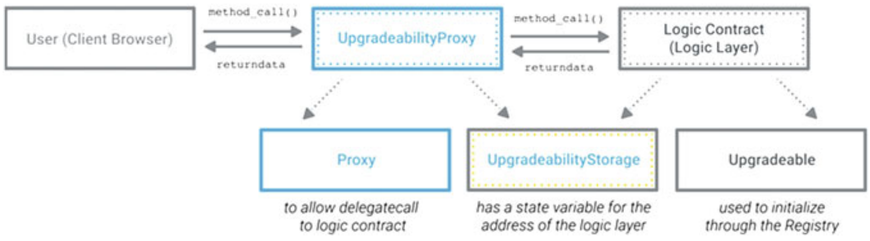

User (Client Browser)
method_call()
returndata
UpgradeabilityProxy
method_call()
returndata
Logic Contract (Logic Layer)
Proxy
to allow delegatecall to logic contract
UpgradeabilityStorage
has a state variable for the address of the logic layer
Upgradeable
used to initialize through the Registry

Fig. 7.6 Inherited storage (https://blog.zeppelinos.org/proxy-patterns/)

7.3 Storage

We talk about strategy to upgrade code in the previous section. But data upgrade is not discussed yet. It might be possible that the upgraded contract cannot access data in data contract or data in data contract is completely lost. Here the challenge is: how to make sure that the new implementation does not overwrite state variables used by proxy contract. There are mainly three kinds of Storage upgrade proposed by ZepplinOS:

- Inherited Storage
- Eternal Storage
- Unstructured Storage

7.3.1 *Inherited Storage*

This method is to make implementation contract and proxy contract follow the same storage structure. As the example stated in Sect. 7.1, raw contract—LogicOne is upgraded to LogicTwo. LogicTwo inherits from LogicOne. If the storage structure of Proxy and LogicOne does not change, upgrade can always be conducted smoothly (Fig. 7.6).

7.3.2 *Eternal Storage*

In Eternal Storage pattern, Storage pattern is defined in a contract which could be inherited by Proxy and logic contract. Storage pattern keeps all state variables. Since proxy contract inherits from storage contract, developer can define their own state variables in proxy contract without fear of being overwritten. And attention!!! It is not allowed to define other state variables in logic contract. This means that logic contract must use the initial data structure of contract.

Here is a diagram from ZepplineOS (Fig. 7.7).

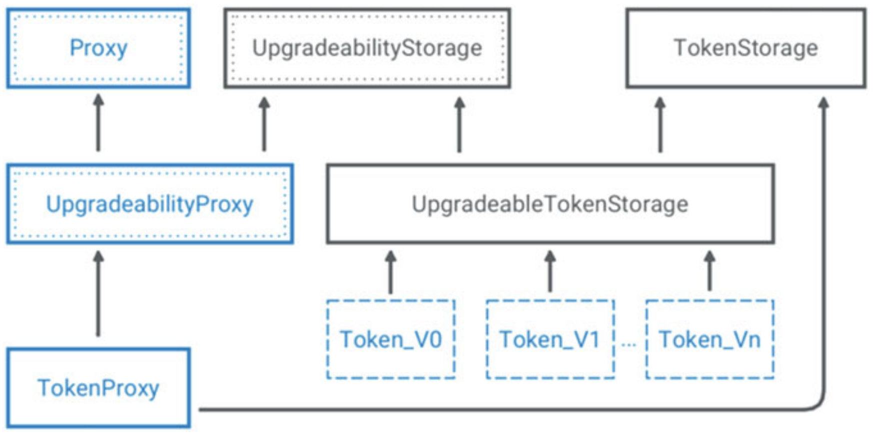


Fig. 7.7 Eternal storage (https://blog.zeppelinos.org/smart-contract-upgradeability-using-eternal-storage/)

Proxy Contract
Encapsulate a fallback function delegate calls to other contracts by using Delegatecall.

Upgradeability Storage Contract
Encapsulate all data to be upgraded.

Tokenstorage Contract
Encapsulate all data related to ERC20, such as totalsupply, balance.

UpgradeabilityProxy Contract
Inherit from UpgradeabilityStorage and Proxy.

UpgradeableTokenStorage
Inherit from UpgradeabilityStorage and Tokenstorage.

TokenProxy
Inherit from UpgradeabilityProxy and Tokenstorage.

Please note that TokenProxy and UpgradeableTokenStorage must have the same inheritance order and to keep the storage structure and order unchanged, it is not allowed to define new state variables in logic contract. That is what Eternal means.

7.3.3 Unstructured Storage

Unstructured Storage pattern is similar to Inherited Storage pattern. And the difference is: it requires logic contract inherits state variables to be upgraded. This pattern uses proxy contract to define and store data to be upgraded.

The main idea of this method is to define a constant variable in proxy contract. Since the variable is a unique SHA3 hash value. And using it as the key to the storage position and key position will not be overwritten because of enough randomness. The benefit of the strategy is: logic contract does not need to know the storage structure of proxy contract; and constant variable does not occupy space in storage.

```
bytes32 private constant implementationPosition =
            keccak256("org.zeppelinos.proxy.implementation");
```

Full source code can be found here.[1] In the following section, we will briefly introduce how Colony and Augur projects handle contract upgrade.

The concept of proxy patterns has been around for a while, but has not received wide adoption due to its complexity, fear of introducing security vulnerabilities, and the controversy of bypassing blockchain's immutability.

7.4 Augur

Augur is a decentralized predictive platform based on Ethereum technology. User can predict and bet by using crypto currency. Depending on crowd wisdom, it can lower counter-party risk and server centralization risk. User can use ETH to bet for prediction, such as stock price, weather, etc. And augur can be used to hedge disastrous risk. Its whitepaper can be found at http://www.augur.net/whitepaper.pdf and its official web is at https://www.augur.net/.

7.4.1 Contract Deployment

Augur project source code locates at https://github.com/AugurProject/augur-core/. In Augur, almost all contracts inherit from IControlled in which "I" indicates that it is an interface. Controlled keeps a reference to a Controller object.

```
contract Controlled is IControlled {
    IController internal controller;
}
```

[1] https://github.com/zeppelinos/labs/tree/master/upgradeability_using_unstructured_storage/.

Augur design of Controller is a contract registry allowing any contract to use a hash value to find its methods. And it allows contract to replace some methods dynamically through re-registration.

```
// Note that the code has been simplified
contract Controller {
      // This contract contains a function signature and contract
address
      struct ContractDetails {
           bytes32 name;
           address contractAddress;
      }
      // Registry which defined a mapping from function sig to
contract details
      mapping(bytes32 => ContractDetails) public registry;
      // Using function sig to find contract address
      function lookup(bytes32 _key) public view returns (address) {
           return registry[_key].contractAddress;
      }
      // Contract address registration
      function registerContract(bytes32 _key, address _address) {
           registry[_key] = ContractDetails(_key, _address);
      }
}
```

Steps are as below:

1. First deploy Controller contract.
2. Then deploy Augur contract and update with new Controller contract address.
3. Deploy the other contracts in turn with new Controller contract address and register in Controller contract.

Functionality upgrade can be implemented by re-registration with new Controller contract address.

7.4.2 Storage Deployment

Augur save some data in root contract in Singleton pattern. Other data are saved in contracts derived by Delegator factory. Delegator contract uses DELEGATECALL opcode delegating all function call to a contract.

```
contract Delegator is DelegationTarget {
        function Delegator(IController _controller, bytes32
    _controllerLookupName) public {
              controller = _controller;
              controllerLookupName = _controllerLookupName;
        }
        function() external payable {
             // Find contract address through name
             address _target = controller.lookup
(controllerLookupName);
             // assembly { DELEGATECALL }
        }
}
```

Some contracts are created dynamically, and each such contract has its own factory contract.

```
// Assume ExampleValueObject is created and registered under
// 'ExampleValueObject' Controller
contract ExampleFactory is Controlled {
  function createExample() external returns
(IExampleValueObject) {
     Delegator d = new Delegator(controller,
'ExampleValueObject');
     IExampleValueObject exampleValueObject =
IExampleValueObject(d);
     exampleValueObject.initialize();
     return exampleValueObject;
  }
}
contract IExampleValueObject {
   function initialize() external;
   function get() external view returns (uint);
}
contract ExampleValueObject is DelegationTarget,
IExampleValueObject {
   uint private value;
   function initialize() external {
     value = uint(42);
   }
   function get() external view returns (uint) {
     return value;
   }
}-
```

Singleton contracts are registered twice in the controller: the first being an instance of the contract and the second being an instance of a Delegator registered under the original name and pointing to the first instance. In this way, the Delegator instance is the place of storage for the contract, while the actual contract is registered separately and can be swapped out for different behaviors.

7.5 Colony

Colony is a platform for creating decentralized organizations. The code has been made public on Github under the colonyNetwork project. The project history began in early 2016. Colony consists of about 14 smart contracts. The smart contracts use the EtherRouter pattern to upgrade contracts. Conceptually, these contracts can be divided into those that concern the Colony Network as a whole, and those that concern an individual Colony. Concretely, all contracts inherit from either the ColonyNetworkStorage or the ColonyStorage contracts. The ColonyNetworkStorage and ColonyStorage contracts store all state variables for their respective descendants.

Each group of contracts has a Resolver instance in which they are all registered. The Resolver acts as a function registry. An EtherRouter is bound to a Resolver and looks up function addresses to what it delegates to. In this way, an EtherRouter will take on all the functions in a Resolver and the shape of the registered contracts.

```
contract Resolver is DSAuth {
  struct Pointer { address destination; uint outsize; }
  mapping (bytes4 => Pointer) public pointers;
  function register(string signature, address destination, uint
outsize) public auth {
    pointers[stringToSig(signature)] = Pointer(destination,
outsize);
  }
  function lookup(bytes4 sig) public view returns(address, uint) {
    return (destination(sig), outsize(sig));
  }
}
```

7.5.1 Contract Deployment

1. First, deploy Colony Network contract:
   - ColonyNetwork
   - ColonyNetworkStaking
   - EtherRouter
   - Resolver

2. Register ColonyNetwork and ColonyNetworkStaking in Resolver contract.
3. As the entry point to Colony Network, EtherRouter is pointed to Resolver.
4. Deploy Colony contract.
   - Colony
   - ColonyTask
   - ColonyFunding
   - ColonyTransactionReviewer
5. Create a new Resolver contract and register all contracts in step 4.
6. The new Resolver is added to Colony Network.

7.5.2 Storage

The first deployed EtherRouter becomes the entry point for the platform and will take on the storage shape and behavior of the contracts in the first Resolver. The first resolver contains the Colony Network contracts and will therefore have the same shape as ColonyNetworkStorage. The Colony Network creates new colonies by creating new EtherRouter instances pointing to the latest version of the Resolvers. Each Colony can upgrade their contract versions independently.

7.6 Summary

Let us summarize all we learned. Contract upgrade is complex and developers must take the following topics into consideration:

- Block GAS Limit
  Contract upgrade involves complex operations: deploying contract, moving data and reference etc., so it will cost a huge amount of GAS. Since the block has GAS upper-bound (Homestead's upper limit is 4,712,388), we should be very careful of upgrade failure derived by exceeding GAS limit.
- Inter-dependency between contract
  Usually, a complex application needs to deploy a series of contract. Replacing one or several contracts of them must take their inter-dependency into account.

Smart Contract Upgrade

- Separate smart contract into several related contracts
  Save related contract address into a contract registry and in this way, code upgrade becomes simple since developer only needs to replace old contract address with new contract address. This kind of upgrade strategy is common. The drawback is that the interface cannot change. So adding new function and deleting function are not allowed.
- Use Delegatecall to delegate the call to external contract

In this way, callee is working under caller's context, which means msg.sender and msg.value do not change. And developer must use assembly code to get the return value of such call.

– Using the method described in Sect. 7.5 can save return value into a Mapping.

  The downside is that return value is strongly bounded to function signature. That is, this method cannot handle dynamic return array or string. Furthermore, GAS used to access storage is expensive.
– To handle the above problem, developer can force a call to return 32 bytes only. And we do not need access storage anymore.
– Use new opcodes: reslultdatasize and resultdatacopy.

  Since Byzantine hard fork at Oct 17th 2017, developers now can use two new opcodes: resultdatasize and resultdatacopy to copy return value of call/delegatecall to memory. This means that developers now can get return value regardless of whatever the return value is

• Pay attention to GAS usage

  Test shows that GAS usage will increase about 1000–1500 in average.
• Perform comprehensive test before official upgrade.

Upgrade Storage

Storage upgrade can be: Inherited Storage, Eternal Storage and Unstructured Storage.

In next chapter, we will discuss about how to develop secure contract and recommend best practices.

Chapter 8
Develop Secure Contract

Nowadays, the security of smart contract is facing huge challenges:

- Smart contract deployed on Ethereum is publicly accessible and transparent.
  Developing smart contract needs a completely new thinking vector which is different from prior projects development.
- The disastrous consequence of vulnerability of smart contract.
  Smart contract usually manages a huge amount of fund. Due to its inherent feature of transparency and public accessibility, fixing the bug of smart contract is not as easy as traditional software patching.

In this section, we start from well-known vulnerability and attack, analyze the source code to identify the root cause and the recommended preventive strategy. We also recommend best practices in developing smart contract in using Solidity. And it must be realized that smart contract attack and defense are a dynamic, ever-changing process. There is no such myth that one solution can solve all problems.

In this chapter, we just discuss some well-known attacks and we do not promise to cover all the vulnerabilities. The new vulnerability is usually listed on the following sources. Official notice of Ethereum or Solidity is from Ethereum Blog. But it is often that the new vulnerability is disclosed and discussed in other places.[1]

- Ethereum Blog: The official Ethereum blog
  - Ethereum Blog—Security only: All relate blogs are tagged with *Security*
- Ethereum Gitter Chat room
  - Solidity
  - Go-Ethereum
  - CPP-Ethereum
  - Research

[1] https://github.com/sigp/solidity-security-blog#race-conditions.

G. Zheng et al., *Ethereum Smart Contract Development in Solidity*,
https://doi.org/10.1007/978-981-15-6218-1_8

- Reddit
- Network Stats

8.1 History

In May of 2016, Peter Vessenes analyzed a sample of smart contracts published online to evaluate their complexity and security. His conclusion: “Ethereum contracts are going to be candy for hackers.” This observation was prescient, as just 1 month later The DAO hack occurred, resulting in the loss of 3.6 million Ether.

In the coming months, more vulnerabilities were uncovered as the community became increasingly concerned with smart contract security. In Devcon2—which was held in September of 2016—smart contracts were still riddled with poor programming practices. That year, roughly 50% of projects holding significant amounts of funds were hacked.

In September of 2016, OpenZeppelin starts to help standardize best practices for smart contract development. Today, more than $1.5 billion of digital assets are powered by OpenZeppelin smart contracts.

While a community effort around smart contract audits and security is started, the Ethereum platform itself has also matured. We have seen the introduction of ERC20 tokens which have improved interoperability, along with security fixes like EIP150, which removed the possibility of a call stack attack. Solidity implemented new keywords such as require, assert, transfer, and revert, all of which reduced the difficulty of building secure smart contracts.

8.2 Attacking Vector

Attacking vectors are classified as below:

- Solidity related
- Platform: timestamp, random, Front running
- Reentrancy
- Denial of Service (DoS)

8.2.1 Solidity Related

8.2.1.1 Overflow/Underflow

When an operation needs to use a fixed-size variable to save a digital number, if the number is outside the range of variable type, it will lead to overflow/underflow.

Solidity could process 256-bit integer. So 2^{256}–1 (which is a big integer) plus 1 will be 0. This is called overflow.

```
0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
+ 0x000000000000000000000000000000000001
----------------------------------------
= 0x000000000000000000000000000000000000
```

If unsigned integer is used, then 0 minus 1 will get the largest integer in solidity. And this is called underflow.

```
0x000000000000000000000000000000000000
- 0x000000000000000000000000000000000001
----------------------------------------
= 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
```

Ethereum Virtual Machine (EVM) assigns fixed size to variables of integer type, which means that an integer variable has its range. If inadvertently, programmers do not check input or variable value underflow/overflow in the calculation process, then this vulnerability can be utilized to attack contract. These kinds of vulnerabilities allow attackers to misuse code and create unexpected logic flows. For example, consider the time locking contract below.

```
contract TimeLock {
    mapping(address => uint) public balances;
    mapping(address => uint) public lockTime;

    function deposit() public payable {
        balances[msg.sender] += msg.value;
        lockTime[msg.sender] = now + 1 weeks;
    }

    function increaseLockTime(uint _secondsToIncrease) public {
        lockTime[msg.sender] += _secondsToIncrease;
    }

    function withdraw() public {
        require(balances[msg.sender] > 0);
        require(now > lockTime[msg.sender]);
        msg.sender.transfer(balances[msg.sender]);
        balances[msg.sender] = 0;
    }
}
```

This contract allows user to deposit ether and user to expect the fund to be locked in contract for a certain period (1 week at least). And the user can extend the lock period. Does the above code achieve the functions the user expect? If attackers somehow have the private key of a user, they can determine the current lock time since userLockerTime is a public variable. Attackers can call increaseLockTime function with 2^{256} as a parameter and it will cause an overflow. lockTime[msg. sender] is set to zero and attackers get rewarded by calling withdraw function.

Let us have a look at the new example:

```
pragma solidity ^0.4.18;
contract Token {
  mapping(address => uint) balances;
  uint public totalSupply;
  function Token(uint _initialSupply) {
     balances[msg.sender] = totalSupply = _initialSupply;
  }
  function transfer(address _to, uint _value) public returns
 (bool) {
    require(balances[msg.sender] - _value >= 0);
    balances[msg.sender] -= _value;
    balances[_to] += _value;
    return true;
  }
  function balanceOf(address _owner) public constant returns
(uint balance) {
    return balances[_owner];
  }
}
```

Above contract is a simple token contract in which participants can employ transfer() function to move their token around. In transfer() function, require statement can be bypassed under the situation of underflow. For example, assume that a user has no balance; the user could call the transfer() function with any non-zero _value and pass the require statement. The reason is: balances[msg.sender] is zero, and subtracting any positive amount (excluding 2^{256}) will result in a positive number due to the underflow described above. Thus, in this example, the user has obtained free tokens caused by an underflow vulnerability.

The typical technique to guard against under/overflow vulnerabilities is to use or build mathematical libraries which replace the solidity standard math operators; addition, subtraction, and multiplication (division is excluded as it does not cause over/under flows and the EVM throws on division by 0). OpenZepplin provides a well-known and widely accepted implementation of Safe Math. We use OpenZepplin safe math library to rewrite the timelock contract as below:

```
library SafeMath {
  function mul(uint256 a, uint256 b) internal pure returns
(uint256) {
    if (a == 0) {
      return 0;
    }
    uint256 c = a * b;
    assert(c / a == b);
    return c;
  }
  function div(uint256 a, uint256 b) internal pure returns
(uint256) {
    // assert(b > 0); // Solidity automatically throws when dividing
by 0
    uint256 c = a / b;
    // assert(a == b * c + a % b); // There is no case in which this
doesn't hold
    return c;
  }
  function sub(uint256 a, uint256 b) internal pure returns
(uint256) {
    assert(b <= a);
    return a - b;
  }
  function add(uint256 a, uint256 b) internal pure returns
(uint256) {
    uint256 c = a + b;
    assert(c >= a);
    return c;
  }
}
contract TimeLock {
    using SafeMath for uint; // use the library for uint type
    mapping(address => uint256) public balances;
    mapping(address => uint256) public lockTime;

    function deposit() public payable {
        balances[msg.sender] = balances[msg.sender].add(msg.
value);
        lockTime[msg.sender] = now.add(1 weeks);
    }

    function increaseLockTime(uint256 _secondsToIncrease) public {
        lockTime[msg.sender] = lockTime[msg.sender].add
(_secondsToIncrease);
    }

    function withdraw() public {
        require(balances[msg.sender] > 0);
```

(continued)

```
        require(now > lockTime[msg.sender]);
        balances[msg.sender] = 0;
        msg.sender.transfer(balances[msg.sender]);
    }
}
```

Please note that all standard mathematics operators are replaced with functions in Safe Math library.

8.2.1.2 Solidity Modifier

Public function can be called from the contract itself, inherited contract, and other contracts.

External function cannot be called by the member functions of the contract itself.

Private function can be called only by the member functions of the contract itself.

Internal functions can be called by the functions of the contract itself and inherited contract.

Delegate Call is a message call. And callee contract code is running in the context of caller contract. That means callee contract code can work on storage, balance, and current address of caller contract. In the following example, attacker gets the control of caller contract (Delegation) through delegating call to public PWN function of Delegate under the context of caller contract.

```
pragma solidity ^0.4.11;
contract Delegate {
  address public owner;
  function Delegate(address _owner) {
    owner = _owner;
  }
  function pwn() {
    owner = msg.sender;
  }
}
contract Delegation {
   address public owner;
   Delegate delegate;
   function Delegation(address _delegateAddress) {
     delegate = Delegate(_delegateAddress);
     owner = msg.sender;
  }
  // an attacker can call Delegate.pwn() in the context of Delegation
  // this means that pwn() will modify the state of **Delegation**
and not Delegate
  // the result is that the attacker takes unauthorized ownership of
```

(continued)

```
the contract
  function() {
    if(delegate.delegatecall(msg.data)) {
      this;
    }
  }
}
```

The contract above opens a backdoor for hackers. Code flow is as below:

1. Attacker sends call—Delegation.PWN()
2. Code execution falls into the fallback function of Delegation since there is no PWN function in Delegation contract
3. In fallback function, delegatecall sends call to Delegate contract
4. Attacker manipulates msg.data which contains the function signature to be called. In this case, assume that attacker stuff pwn's signature into msg.data
   msg.data = first 8 bytes of sha3 (alias for keccak256)

```
web3.sha3("pwn()").slice(0, 10) --> 0xdd365b8b

// if pwn function has a parameter such as pwn(uint256 x):
web3.sha3("pwn(uint256)").slice(0,10) --> 0x35f4581b
```

5. PWN function in Delegate is called and owner of Delegation will be changed since PWN works in the context of Delegation.

The main problem here is a combination of insecure visibility modifiers and misuse of delegate call with arbitrary data.

8.2.1.3 Floating Points and Precision

Solidity does not support fix-point numbers and float point numbers until version 0.4.24. That is to say, we must use integers to present float numbers, which may lead to vulnerability if the contract is not implemented correctly. Here is an example (in order for easy understanding, please ignore the over/under flow vulnerability).

```
contract FunWithNumbers {
    uint constant public tokensPerEth = 10;
    uint constant public weiPerEth = 1e18;
    mapping(address => uint) public balances;
```

(continued)

```
    function buyTokens() public payable {
       uint tokens = msg.value/weiPerEth*tokensPerEth; // convert
wei to eth, then multiply by token rate
       balances[msg.sender] += tokens;
    }
    function sellTokens(uint tokens) public {
       require(balances[msg.sender] >= tokens);
       uint eth = tokens/tokensPerEth;
       balances[msg.sender] -= tokens;
       msg.sender.transfer(eth*weiPerEth); //
    }
}
```

This simple token buy/sell contract has some obvious problems. Although the calculation formula is correct, incorrect support of float point number leads to an unexpected result. For example: in buyTokens function, if the amount to buy is less than 1 ether, then the result of msg.value/weiPerEth will be 0. And the final result will also be 0. By the same token, in sellTokens function, if the amount to sell is less than 10, the final result will be 0. In fact, rounding here is always down, so selling 36 tokens will result in 3 ether. If high precision is needed in ERC20 contract, situation will become more complex.

To avoid such problem, the general principle is always to keep the right precision in your smart contracts, especially when dealing with ratios and rates which reflect economic decisions.

- ensure that any ratios or rates allow for large numerators in fractions.
- be mindful of the order of operations.
- convert variables into higher precision, perform all mathematical operations, then finally when needed, convert back down to the precision for output.

8.2.1.4 Tx.Origin Authentication

Solidity has a global variable—tx.origin which trace calling stack back to initiator's address. Using this variable to do user authentication may incur phishing attack which allure user to execute some operations requiring authorization. Let us have a look a simple contract below:

```
contract Phishable {
   address public owner;
   constructor (address _owner) {
      owner = _owner;
   }
   function () public payable {} // collect ether
```

(continued)

```
    function withdrawAll(address _recipient) public {
       require(tx.origin == owner);
       _recipient.transfer(this.balance);
    }
}
```

Notice that tx.origin is used in function withdrawAll(). Here is a contract created by the attacker.

```
import "Phishable.sol";
contract AttackContract {
   Phishable phishableContract;
   address attacker; // The attackers address to receive funds.
   constructor (Phishable _phishableContract, address
_attackerAddress) {
      phishableContract = _phishableContract;
      attacker = _attackerAddress;
   }
   function () {
      phishableContract.withdrawAll(attacker);
   }
}
```

Here we describe how to utilize this contract:

1. Attacker would deploy AttackContract

   Attacker convinces the owner of the Phishable contract to send AttackContract some amount of ether.
2. Disguise this contract as their own private address and convince engineer the victim to send some form of transaction to the address.

   The victim, unless being careful, may not notice that there is code at the attacker's address, or the attacker may pass it off as being a multi-signature wallet or some advanced storage wallet.
3. The victim sends a transaction (with enough gas) to the AttackContract address,
4. Invoke the fallback function,
5. Calls the withdrawAll() function of the Phishable contract, with the parameter attacker.

This will result in the withdrawal of all funds from the Phishablecontract to the attacker address. This is because the address that first initializes the call is the victim (i.e. the owner of the Phishable contract). Therefore, tx.origin will be equal to the owner and the require statement of the Phishable contract will pass.

Therefore, keep in mind, tx.origin should not be used for authorization in smart contracts.

8.2.1.5 Unchecked CALL Return Values

Usually, we need to call external contracts, such as using transfer(), or send(), or CALL opcode to send ether. The call() and send() functions return a boolean indicating if the call succeeds or fails. This will lead to a simple caveat: if the external call fails, solidity only returns a flag and will not rollback. So, the developer needs to revert manually in such situations, or it looks like a shark smelling blood. Let us have a look at the following example:

```
contract Lotto {
   bool public payedOut = false;
   address public winner;
   uint public winAmount;

   // ... extra functionality here
   function sendToWinner() public {
      require(!payedOut);
      winner.send(winAmount);
      payedOut = true;
   }

   function withdrawLeftOver() public {
      require(payedOut);
      msg.sender.send(this.balance);
   }
}
```

The contract is a lottery program and the winner can win a certain amount of ether. The problem of the contract is that the contract does not verify the return value of send(). If winner transaction fails (possibly caused by out of GAS, or call stack depth attack), payedOut will be set to true (regardless of whether ether was sent or not). In this case, the public can withdraw the winner's winnings via the withdrawLeftOver() function.

Whenever possible, using the transfer() function rather than send() as transfer() will revert if the external transaction reverts. If send() is required, always ensure to check the return value. An even more robust recommendation is to adopt a withdrawal pattern. The idea is to logically isolate the external send functionality from the rest of the code base and place the burden of potentially failed transaction to the end user who is calling the withdraw function. The real-world attack details can be found here.[2]

[2]https://github.com/etherpot.

8.2.1.6 Unexpected Ether

Each contract has a balance variable which can be used as [contract address].balance. Most of the cases, the variable can be updated by function with a payable modifier. There are two special cases that the developer can manipulate the ether in the contract.

Usually, we do invariant-checking in solidity programming. For example, in standard ERC20 contract, totalSupply is supposed to be invariant. From an unexperienced developer's point of view, there is one apparent invariant—the current balance the contract hold. Developers have a misconception that a contract can only accept or obtain ether via payable functions. The typical case of this misconception is the incorrect use of this.balance.

There are two cases that can send ether to a contract forcibly without using the payable function:

SelfDestruct/Suicide
Selfdestruct (address) function will delete all binary code in contract address and send ether to an address specified by the parameter. If selfdestruct() forcibly sends ether to a contract address, it will not go through any payable function (even fallback function) in target contract.

Pre-sent Ether
Another way to send ether bypassing payable function is to pre-load the contract address with ether. Contract addresses are deterministic, in fact, the address is calculated from the hash of the address creating the contract and the transaction nonce which creates the contract.

```
address = sha3(rlp.encode([account_address,transaction_nonce]))
```

This is to say, anyone can calculate what a contract address will be before it is created and thus send ether to that address. When the contract does get created, it will have a non-zero ether balance.

Let us have a look at the following contract and see how to utilize the vulnerability:

```
contract EtherGame {
   uint public payoutMileStone1 = 3 ether;
   uint public mileStone1Reward = 2 ether;
   uint public payoutMileStone2 = 5 ether;
   uint public mileStone2Reward = 3 ether;
   uint public finalMileStone = 10 ether;
   uint public finalReward = 5 ether;
   mapping(address => uint) redeemableEther;
   // users pay 0.5 ether. At specific milestones, credit their
accounts
```

(continued)

```
    function play() public payable {
        require(msg.value == 0.5 ether); // each play is 0.5 ether
        uint currentBalance = this.balance + msg.value;
        // ensure no players after the game as finished
         require(currentBalance <= finalMileStone);
        // if at a milestone credit the players account
        if (currentBalance == payoutMileStone1) {
            redeemableEther[msg.sender] += mileStone1Reward;
        }
        else if (currentBalance == payoutMileStone2) {
            redeemableEther[msg.sender] += mileStone2Reward;
        }
        else if (currentBalance == finalMileStone ) {
           redeemableEther[msg.sender] += finalReward;
        }
        return;
    }
    function claimReward() public {
        // ensure the game is complete
        require(this.balance == finalMileStone);
        // ensure there is a reward to give
        require(redeemableEther[msg.sender] > 0);
        redeemableEther[msg.sender] = 0;
        msg.sender.transfer(redeemableEther[msg.sender]);
    }
}
```

This contract is just a simple game whereby players send 0.5 ether quanta to the contract in hope to be the player that reaches one of three milestones (payoutMileStone1, payoutMileStone2, finalMileStone) first. Milestones are denominated in ether. The first to reach the milestone may claim a portion of the ether when the game has ended. The game ends when the final milestone (10 ether) is reached and users can claim their rewards.

```
 ...
uint currentBalance = this.balance + msg.value;
      // ensure no players after the game as finished
      require(currentBalance <= finalMileStone);
     ...
      require(this.balance == finalMileStone);
     ...
```

The issues with the EtherGame contract come from the poor use of this.balance. An attacker could:

1. Deploy a contract containing selfdetruct() function which forcibly sends the balance held by the contract to EtherGame contract.
2. Send 0.1ether to the contract. If succeed, the contract current hold 0.1ether.

3. Call selfdetruct() and it will send current balance—0.1ether to EtherGame. And it does not hit any payable functions in EtherGame.

As all legitimate players can only send 0.5 ether increments, once EtherGame receives 0.1 ether, this.balance would not be half integer numbers any more. This means that all milestones will not be reached.

```
...
// if at a milestone credit the players account
if (currentBalance == payoutMileStone1) {
   redeemableEther[msg.sender] += mileStone1Reward;
}
else if (currentBalance == payoutMileStone2) {
   redeemableEther[msg.sender] += mileStone2Reward;
}
else if (currentBalance == finalMileStone ) {
   redeemableEther[msg.sender] += finalReward;
}
  ...
```

A more aggressive case is: a vengeful attacker may forcibly send 10 ether (or an equivalent amount of ether that pushes the contract's balance above the finalMileStone) to EtherGame contract. And it will lock all rewards in the contract forever since claimReward() function will always revert caused by the require statement.

```
// ensure the game is complete
      require(this.balance == finalMileStone);
```

This vulnerability typically arises from the misuse of this.balance. The lesson learned is: Contract logic should avoid being dependent on exact values of the balance of the contract because it can be artificially manipulated. If you definitely need exact values of deposited ether, a self-defined variable should be used that gets incremented in payable functions, to safely track the deposited ether. This variable will not be influenced by the forced ether sent via a selfdestruct() call.

8.2.2 Platform Related Attack

Solidity code is compiled into EVM opcode (which is defined in Ethereum Yellow Paper). In this section, we are going to discuss vulnerabilities related to Ethereum Virtual Machine.

8.2.2.1 Platform Constraint

EVM has a lot of constraints due to the security design of the platform. Since constraints are platform-wise, unawareness of such constraints may put your contract in danger. Let us check an example of bonus calculation.

```
   // UNSAFE CODE, DO NOT USE!
contract BadArrayUse {
   address[] employees;
   function payBonus() {
     for (var i = 0; i < employees.length; i++) {
       address employee = employees[i];
       uint bonus = calculateBonus(employee);
       employee.send(bonus);
     }
   }
   function calculateBonus(address employee) returns (uint) {
     // some expensive computation ...
   }
}
```

Coding logic above is straightforward and seems safe and secure. But because of the platform-based special features, there are three timing bombs:

Infinite Loop
Because of the type inference rule of solidity, the type of i will be—uint8. Since the first use of i is to save 0 to i, and solidity will automatically pick the smallest appropriate type (which will be uint8) for i based on solidity type inference rule. So, if the array has more than 255 elements, the loop will never stop, which will lead to exhaustion of GAS in the end. Lesson learned here is: try to use explicit type instead of var. Let us see an improved version as below:

```
// UNSAFE CODE, DO NOT USE!
contract BadArrayUse {
  address[] employees;
  function payBonus() {
    for (uint i = 0; i < employees.length; i++) {
      address employee = employees[i];
      uint bonus = calculateBonus(employee);
      employee.send(bonus);
    }
  }
  function calculateBonus(address employee) returns (uint) {
    // some expensive computation ...
  }
}
```

GAS Limitation

Since Ethereum is a world computer, any state change function call will cost lots of resources. GAS is a mechanism to charge fees for resource use. If calculateBonus has some complex calculation, the function will cost lots of GAS which is likely to exceed the upper limit of GAS usage of transaction or block. If a transaction does exceed the GAS limit, all state changes will be reverted; but GAS spent will not be refunded. So, please pay attention to GAS usage if loop is used in contract programming. Let us optimize the code above: separate calculation from loop. Please note that the code below still has a potential risk of Exhaustion of GAS since array size might grow bigger and bigger:

```
// UNSAFE CODE, DO NOT USE!
contract BadArrayUse {
    address[] employees;
    mapping(address => uint) bonuses;
    function payBonus() {
      for (uint i = 0; i < employees.length; i++) {
        address employee = employees[i];
        uint bonus = bonuses[employee];
        employee.send(bonus);
      }
    }
    function calculateBonus(address employee) returns (uint) {
      uint bonus = 0;
      // some expensive computation modifying the bonus...
      bonuses[employee] = bonus;
    }
}
```

Stack Depth Constraint

EVM call stack has a hard cap—1024, which means that function will fail if nested call stack exceeds 1024. An attacker can recursively call our contract 1024 times and this will lead to a send fail due to the call stack depth constraint.

Here is a final version of the contract which fixes all problems described above:

```
import './PullPayment.sol';
contract GoodArrayUse is PullPayment {
  address[] employees;
  mapping(address => uint) bonuses;
  function payBonus() {
    for (uint i = 0; i < employees.length; i++) {
      address employee = employees[i];
      uint bonus = bonuses[employee];
      asyncSend(employee, bonus);
    }
  }
```

(continued)

```
    function calculateBonus(address employee) returns (uint) {
      uint bonus = 0;
      // some expensive computation...
      bonuses[employee] = bonus;
    }
}
```

8.2.2.2 Race Conditions/Front Running

The combination of external calls to other contracts and the multi-user nature of the underlying blockchain gives rise to a variety of potential Solidity pitfalls whereby users *race* code execution to obtain unexpected states. In this section, we will discuss generally different kinds of race conditions that can occur on the Ethereum blockchain.

Once a transaction is submitted, an Ethereum node will pool the transaction and pack it into block in an appropriate way. Only when one miner solves the consensus question, the miner will pick transactions in the pool and pack them into block. Only those transactions packed into block are considered valid. Generally, the packing order is dependent on gasPrice. The attacker can watch the pool of transaction which can be of interests of the attacker, such as modifying privilege or update states of contract, etc. Attacker can get the transaction data and create a new transaction with higher gasPrice. This will make new transaction included into block prior to the original one.

Here is a simple example showing how to conduct the attack:

```
contract FindThisHash {
       bytes32 constant public hash = 0xb5b5b97fafd9855eec9b41f7
4dfb6c38f5951141f9a3ecd7f44d5479b630ee0a;

       constructor() public payable {} // load with ether

       function solve(string solution) public {
           // If you can find the pre image of the hash, receive 1000
ether
           require(hash == sha3(solution));
           msg.sender.transfer(1000 ether);
       }
}
```

Assume this contract holds 10,000 ether. Any user can claim the balance of the contract (which is 10,000 ether) if the user finds the pre-image of 0xb5b5b97fafd9855eec9b41f74dfb6c38f5951141f9a3ecd7f44d5479b630ee0a and submit the solution. For example, if a user finds the solution is “resolved!” and they submit the solution through calling solve() with parameter—resolved!. If the attacker is watching the pool at this time, the attacker will see the solution, and validate it. If they believe the solution is correct, they can initiate an equivalent transaction with higher gasPrice than that of the original transaction. Miner will prefer to pack transaction with higher gasPrice into block. In this situation, the attacker will get 10,000 ether, while original solution provider gets nothing (since 10,000 ether has been withdrawn by attacker).

There are two types of user who can perform such front-running attack: (1) Users who modify the gasPrice and (2) miners themselves. A contract that is vulnerable to the first class (users) is significantly worse-off than one vulnerable to the second (miners) as miners can only perform the attack when they solve a block, which is unlikely for any individual miner targeting a specific block. Here are a few mitigation measures with relation to which type of attackers they may prevent.

One method to mitigate the situation is to set the upper bound of gasPrice in contract. This will prevent attackers from increasing gasPrice and making preferential transaction ordering. This method only mitigates the risk for the first type of user. As to miners, since they can order the transactions in new block regardless of gasPrice, miners still can perform front-running attack.

Another method is to use commit-reveal pattern: first, user sends a transaction with hidden data (usually a hashed value). After the transaction is packed into a block, user sends another transaction with solution details. In this way, miners and other users do not know solution details, they cannot create a new transaction with solution details to perform the front-running attack.

8.2.2.3 Uninitialized Storage Pointers

In EVM, the variable can be stored either as memory or as storage. It is recommended to learn implementation details of data location of the variable. Incorrect initialization of variables will lead to vulnerability.[3] Depending on the variable type, local variables are default to storage or memory. Un-initialized storage variable could be pointed to an unknown address, which will incur attack.

Let us have a look at the following name registry contract:

```
// A Locked Name Registrar
contract NameRegistrar {
    bool public unlocked = false;  // registrar locked, no name
updates
    struct NameRecord { // map hashes to addresses
       bytes32 name;
```

(continued)

[3]https://medium.com/cryptronics/storage-allocation-exploits-in-ethereum-smart-contracts-16c2aa312743, https://medium.com/cryptronics/ethereum-smart-contract-security-73b0ede73fa8, https://solidity.readthedocs.io/en/develop/miscellaneous.html#layout-of-state-variables-in-storage.

```
        address mappedAddress;
     }
     mapping(address => NameRecord) public registeredNameRecord;
// records who registered names
    mapping(bytes32 => address) public resolve; // resolves hashes
to addresses
    function register(bytes32 _name, address _mappedAddress)
public {
        // set up the new NameRecord
        NameRecord newRecord;
        newRecord.name = _name;
        newRecord.mappedAddress = _mappedAddress;
        resolve[_name] = _mappedAddress;
        registeredNameRecord[msg.sender] = newRecord;
        require(unlocked); // only allow registrations if contract
is unlocked
    }
}
```

This simple name registry contract has only one function. When the contract is unlocked, anyone is able to register a name (a bytes32 type of hash value) and map the name to an address. This contract is initialized to "locked" status when first deployed. *Require(unlocked)* statement prevents the user from adding new name when the contract is locked. However, there is a vulnerability which allows to add new names when the contract is locked.

To study this vulnerability, we must understand how storage variables are arranged in solidity. Please go over Chaps. 4 and 5 if you do not understand. State variables are stored sequentially in *slots* as they appear in the contract; so unlocked is at slot 0, registeredNameRecord is at slot 1, resolve is at slot 2 . . . Each slot is 32 bytes.

Slot	Variable	Type	Value
0	Unlocked	Boolean	0000000000000000000000000000000 00000000000000000000000000000000
1	registeredNameRecord	Struct	
2	Resolve	Mapping	

As the local variable of function, newRecord is default to Storage since it is a struct type. The vulnerability lies in that newRecord is not initialized. Because newRecord is storage type, it becomes a pointer to storage. And since it is not initialized, it points to slot 0 where is the place for unlocked. Next step, the function set _name to nameRecord.name, and set _mappedAddress to nameRecord. mappedAddress. These two assignments actually update slot 0 and slot 1, which is the slot for unlocked and registeredNameRecord. This means that unlocked can be updated in register() directly. So, if the last byte of _name is not zero, the last byte of Slot 0 is changed to a non-zero value which means the unlocked value is changed to true. We can try it in remix. If we pass the following value as parameter _name: 0x0000000000000000000000000000000000000000000000000000000000000001

Solidity compiler gives warnings for uninitialized variables. Developer needs to pay attention to these warnings. And it is always a good programming habit to add memory or storage modifier to variables explicitly.

8.2.2.4 Entropy Illusion

As described in previous chapters, all Ethereum transactions are deterministic state transfer: each transaction updates the global state of Ethereum system and state transfer is in the form of calculation. And theoretically, it can be formal verified. Since everything in Ethereum is deterministic, that is to say that there is not a random source in the whole blockchain. That is why there is no rand() function in solidity. Currently, there are many proposals about how to obtain randomness in blockchain. One of the most well-known proposals is RandDAO.

There are many gambling contracts/DApps built on the Ethereum platform. Generally speaking, gambling requires uncertainty coming from a source external to the blockchain. This is possible for bets among peers (for example, the commit-reveal scheme proposed by RandDAO), however, if you want to implement a contract to act as *the house* (like in blackjack or roulette), it is much more difficult. A common misuse is to use future block variables, such as hashes, timestamps, block number, or gas limit. The issue with these is that they are controlled by the miner who mines the block and as such are not truly random. For example, a gambling smart contract holds logic that returns black if the next block hash ends in an even number. A miner (or miner pool) could bet $1 M on black. If they solve the next block and find the hash ends in an odd number, they would prefer not to publish their block and mine another until they find a solution with the block hash being an even number. Using past or present variables can be even more devastating. Furthermore, using solely block variables means that the pseudo-random number will be the same for all transactions in a block, so an attacker can multiply their wins by doing many transactions within a block (should there be a maximum bet).

Since there is no source of randomness in blockchain, the source of randomness must be external to the blockchain. This can be done among peers with systems such as RandDAO (commit-reveal scheme), or via changing the trust model to a group of participants. Another popular solution is to introduce randomness source which is called Oracle (could be centralized or decentralized). Block variables should not be used as source entropy since they can be manipulated by miners.

8.2.2.5 Block Timestamp Manipulation

Block timestamp is used by many apps in implementing their business logic, such as randomness source, and lock period of the fund. And many state-change conditions are based on block timestamp. Since the miner is able to adjust block timestamp, improper use of block timestamp will endanger contracts on Ethereum platform.

Miners can change block timestamp if they have enough motive. Here is a simple example.

```
contract Roulette {
    uint public pastBlockTime; // Forces one bet per block

    constructor() public payable {} // initially fund contract

    // fallback function used to make a bet
    function () public payable {
        require(msg.value == 10 ether); // must send 10 ether to play
        require(now != pastBlockTime); // only 1 transaction per
block
        pastBlockTime = now;
        if(now % 15 == 0) { // winner
        msg.sender.transfer(this.balance);
        }
    }
}
```

This contract is a simple lottery contract. The logic is based on the assumption that block.timestamp is uniformly distributed about the last two digits. If the assumption is true, there would be a 1/15 chance of winning this lottery.

However, as we know, miners can adjust the timestamp if they need to. In this particular case, if enough ether pooled in the contract, in order to win the ether locked in the contract along with the block reward, a miner who solves a block is incentivized to choose a timestamp such that block.timestamp with module 15 as 0. As there is only one person allowed to bet per block, this is also vulnerable to front-running attacks.

In the real world, miners cannot choose arbitrary block timestamps since block timestamps are monotonically increasing. Because nodes will not validate blocks whose timestamps are in the future, miners are also limited to setting blocktimes not too far in the future as these blocks will likely be rejected by the network.

Block timestamps should not be used for entropy or generating random numbers—i.e. they should not be the deciding factor (either directly or through some derivation) for winning a game or changing an important state (if assumed to be random). Time-sensitive logic is sometimes required; i.e. unlocking contracts, completing an ICO after a few weeks or enforcing expiry dates. It is sometimes recommended to use block.number and an average block time to estimate times; i.e. 1 week with a 10 s block time, equates to approximately, 60,480 blocks. Thus, specifying a block number at which to change a contract state can be more secure as miners are unable to manipulate the block number as easily. The BAT ICO contract employed this strategy. This can be unnecessary if contracts are not particularly concerned with miner manipulations of the block timestamp, but it is something to be aware of when developing contracts.

8.2.2.6 External Contract Referencing

Ethereum is a world computer. We can re-use code and interact with deployed contracts. In fact, there are lots of contracts keeping the external reference to other contracts and using message to interact with them. These message calls may cause vulnerability if not properly used.

In Solidity, any address can be forced casted into contract regardless of the code running on the address. The following code showcases the risk:

```
//encryption contract
contract Rot13Encryption {
      event Result(string convertedString);
       //rot13 encrypt a string
      function rot13Encrypt (string text) public {
          uint256 length = bytes(text).length;
          for (var i = 0; i < length; i++) {
             byte char = bytes(text)[i];
             //inline assembly to modify the string
             assembly {
                 char := byte(0,char) // get the first byte
                 if and(gt(char,0x6D), lt(char,0x7B)) // if the
character is in [n,z], i.e. wrapping.
                 { char:= sub(0x60, sub(0x7A,char)) } // subtract from
the ascii number a by the difference char is from z.
                 if iszero(eq(char, 0x20)) // ignore spaces
                {mstore8(add(add(text,0x20), mul(i,1)), add
(char,13))} // add 13 to char.
             }
          }
          emit Result(text);
   }
   // rot13 decrypt a string
   function rot13Decrypt (string text) public {
       uint256 length = bytes(text).length;
       for (var i = 0; i < length; i++) {
           byte char = bytes(text)[i];
           assembly {
               char := byte(0,char)
               if and(gt(char,0x60), lt(char,0x6E))
               { char:= add(0x7B, sub(char,0x61)) }
               if iszero(eq(char, 0x20))
               {mstore8(add(add(text,0x20), mul(i,1)), sub(char,13))}
           }
       }
```

(continued)

```
      emit Result(text);
    }
  }
```

Code above accepts a string as input and (a-z) and encrypt it—shift each character 13 places to the right (wrapping around “z”). The contract below uses the encryption function above through an external reference.

```
import "Rot13Encryption.sol";
// encrypt your top secret info
contract EncryptionContract {
   // library for encryption
   Rot13Encryption encryptionLibrary;

   // constructor - initialise the library
   constructor(Rot13Encryption _encryptionLibrary) {
      encryptionLibrary = _encryptionLibrary;
   }
   function encryptPrivateData(string privateInfo) {
      // potentially do some operations here
      encryptionLibrary.rot13Encrypt(privateInfo);
   }
 }
```

encryptionLibrary address is neither public nor constant. So, we can pass the following contract address to encryptionLibrary in constructor while deploying. If a linked contract does not have rot13Encrypt function, the fallback function will be triggered. The following is an example:

```
contract Blank {
    event Print(string text);
    function () {
       emit Print("Here");
        //put malicious code here and it will run
    }
 }
```

encryptionLibrary.rot13Encrypt() triggers an Event which prints text “Here.” In principle, attackers can run any arbitrary code unknowingly.

There are multiple ways to prevent scenarios above. First, we can generate encryption library dynamically in the constructor as below:

```
constructor() {
        encryptionLibrary = new Rot13Encryption();
   }
```

The external reference is created while deploying and there is no way to replace the reference to Rot13Encryption. An alternative solution is to assign deployed Rot13Encryption address to encryptionLibrary in constructor. Real-world attack case can be found here.[4]

8.2.2.7 Short Address/Parameter Attack

Parameters are ABI-encoded while passed to the smart contract. If the parameter is an address which is 20 bytes long, it is possible that the developer just passes 19 bytes long as an address. And in this case, EVM will pad 0 to the end to align to 20 bytes. If third-party application does not validate the input parameter, this may incur a problem. Look at following transfer function of ERC20 standard.

```
function transfer(address to, uint tokens) public returns (bool
success);
```

Assume that an ERC20 contract holds billions of tokens and a user wants to withdraw 1000 tokens to his address—0xdeaddeaddeaddeaddeaddeaddeadd eaddeaddead. According to ABI specification, EVM will encode the parameters and result is like below: a9059cbb000000000000000000000000deaddeadde addeaddeaddeaddeaddeaddeaddead00000000000000000000000000000000000000 000000003635C9ADC5DEA00000. The first 4 bytes (a9059cbb) is the function selector of transfer(), and the next 32 bytes is an uint256 variable which is the withdrawal amount. 3635C9ADC5DEA00000 (which is 1000,000,000,000,000, 000,000 in decimal format) represents 1000 tokens (assume the ERC20 contract has 18 decimal).

Now, if attack pass an 19 bytes long address to transfer(), for example, attacker pass 0xdeaddeaddeaddeaddeaddeaddeaddeaddeadde as receiver address and 1000 token, and if function does not validate input, the call will be encoded as below a9059cbb000000000000000000000000deaddeaddeaddeaddeaddeaddeaddead-de00000000000000000000000000000000000000000000003635C9ADC5DE-A0000000. Please notice that 00 has been padded to the end of encoded result for

[4]https://www.reddit.com/r/ethdev/comments/7x5rwr/tricked_by_a_honeypot_contract_or_beaten_by/.

32-byte alignment. If the encoded result is sent to contract, the address becomes 0xdeaddeaddeaddeaddeaddeaddeaddeaddeadde00, and withdrawal amount becomes 3635C9ADC5DEA0000000. Now, the withdrawal amount becomes 256,000 tokens. In this situation, if contract balance allows, the user can withdraw 256,000 tokens. In the real world, if attacker generates address with trailing 0 (This is easy through brute-force), then they can steal tokens.

Validating all inputs before sending them to the blockchain will prevent these kinds of attacks. And parameter ordering plays an important role here.

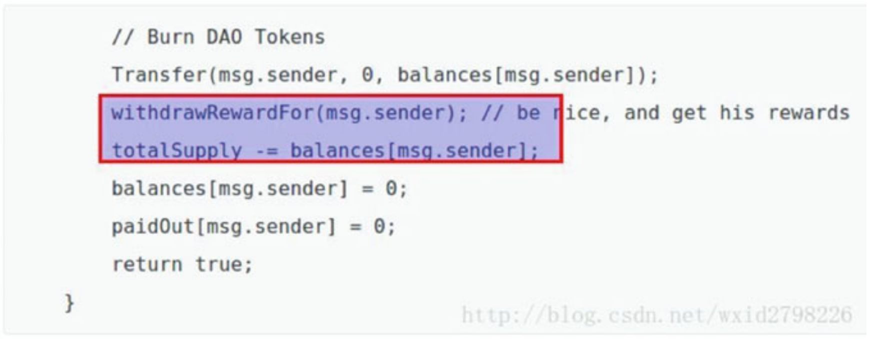
8.2.3 Reentrancy (THE DAO Hack)

Reentrancy is actually solidity programming related. The reason to put it into a separate category is one of its notorious attacks—THE DAO Hack: The DAO contract launched on 30th April, 2016, with a 28-day funding window. For whatever reason, The DAO was popular, raising over $100 m by 15th May, and by the end of the funding period, The DAO was the largest crowdfunding in history, having raised over $150 m from more than 11,000 enthusiastic members. In particular, Stephan Tual, one of The DAO's creators, announced on 12th June that a "recursive call bug" had been found in the software but that "no DAO funds [were] at risk" (Fig. 8.1).

Unfortunately, while programmers were working on fixing this and other problems, an unknown attacker began using this approach to start draining The DAO of ether collected from the sale of its tokens.

By Saturday, 18th June, the attacker managed to drain more than 3.6 m ether into a "child DAO" that has the same structure as The DAO. The price of ether dropped from over $20 to under $13.

As the bug fix, on 17th June, Vitalik Buterin of the Ethereum Foundation issued a critical update, saying that the DAO was under attack and that he had worked out a solution:



```
        // Burn DAO Tokens
        Transfer(msg.sender, 0, balances[msg.sender]);
        withdrawRewardFor(msg.sender); // be nice, and get his rewards
        totalSupply -= balances[msg.sender];
        balances[msg.sender] = 0;
        paidOut[msg.sender] = 0;
        return true;
    }
```

Fig. 8.1 The DAO attack

> A software fork has been proposed, (with NO ROLLBACK; no transactions or blocks will be "reversed") which will make any transactions that make any calls/callcodes/delegatecalls that reduce the balance of an account with code hash0x7278d050619a624f84f51987149dd b439cdaadfba5966f7cfaea7ad44340a4ba (i.e. the DAO and children) lead to the transaction (not just the call, the transaction) being invalid . . .

And the DAO attack finally leads to the hard fork of Ethereum into ETC and ETH.

Whenever <address>.transfer() is invoked, it provides an opportunity for control to flow to a corrupted contract, because the target address could be a contract. If the address is a contract, when called, the called contract's fallback function can contain code to call back the calling contract's function in a recursive manner that can be detrimental.

In the code snippet below, for withdrawFund() function to work, we have to make an external call to the recipient's address. Therefore, we utilize the withdrawal design pattern to separate the accounting logic from the transfer logic, making sure to finish all internal state changes first before invoking the external call. Usually, this means transfer() is the last step. Moreover, potential cross-function race conditions such as calling internal functions that call external functions are also designed away.

```
function withdrawFund(address movie, address recipient, uint
amount,
string expense)
    public
    stopInEmergency
    onlyFilmmaker(movie)
    chargeWithdrawFee(amount)
    returns (bool)
{
   require(recipient != address(0));
   require(amount > 0);
   require(bytes(expense).length > 0);
Movie(movie).withdrawFund(amount.add(withdrawFee.mul(amount).
div(100)));

   emit FundWithdrawn(now, movie, recipient, amount, expense);
   recipient.transfer(amount);
   return true;
}
```

Finishing all internal work before making external calls also protects against *denial of service* attacks. The called contract's fallback can always revert, which also halts the execution of the calling contract. If there are necessary operations further along, they will never get executed.

By utilizing the withdrawal design pattern to favor pull payments over push payments, we also avoid denial of service attacks that target the block gas limit. We could have looped through an array of addresses, pushing payments to the recipients automatically for their convenience. However, attackers could take advantage of the fact that we are looping through an array of unknown size, creating many corrupt recipient accounts to cause our function to hit the block gas limit whenever it executes. The effect, our contract becomes unusable.

There are a number of common techniques which help avoid potential reentrancy vulnerabilities in smart contracts. The first is to use the built-in transfer() function when sending ether to external contracts. The transfer function only sends 2300 gas which is not enough for the destination address/contract to call another contract (i.e. re-enter the sending contract).

The second technique is to ensure that all logic that changes state variables happen before ether is sent out of the contract (or any external call). It is good practice to place any code that performs external calls to unknown addresses as the last operation in a localized function or piece of code execution. This is known as the checks-effects-interactions pattern.

A third technique is to introduce a mutex. That is, to add a state variable which locks the contract during code execution, preventing reentrancy calls.

8.2.4 Denial of Service (DOS)

The definition of DDOS attack is making contract un-functional or irresponsive in a certain time of period or forever. The objective of DDOS attack is not to steal money from smart contracts, but to make smart contracts un-functional or unresponsive which block the users of contracts. There are multiple ways to achieve such goal. We just discuss a few of them.

Iterate Over a Big Mappings or Arrays in a Loop
The following code is to distribute tokens to all investors:

```
contract DistributeTokens {
    address public owner; // gets set somewhere
    address[] investors; // array of investors
    uint[] investorTokens; // the amount of tokens each investor
gets
    // ... extra functionality, including transfertoken()
    function invest() public payable {
       investors.push(msg.sender);
       investorTokens.push(msg.value * 5); // 5 times the wei sent
       }
    function distribute() public {
       require(msg.sender == owner); // only owner
       for(uint i = 0; i < investors.length; i++) {
          // here transferToken(to,amount) transfers "amount" of
tokens to the address "to"
          transferToken(investors[i],investorTokens[i]);
       }
    }
}
```

distribute () function use a for loop to iterate over an array. General logic is okay. But if array size becomes super huge, the iteration will lead to exhaustion of GAS. Thus, distribute() will always fail.

Owner Only Allowed Operations
The following is an ICO contract which need the owner of the contract to set finalize flag and tokens are allowed to be transferred only when the contract is finalized.

```
bool public isFinalized = false;
address public owner; // gets set somewhere
function finalize() public {
   require(msg.sender == owner);
   isFinalized == true;
}
function transfer(address _to, uint _value) returns (bool) {
   require(isFinalized);
   super.transfer(_to,_value)
}
...
```

As can be seen from the code above, the contract will be un-functional if the owner's private key is lost.

Progressing State Based on External Calls
Sometimes, we have such logic: we need to send some ether to external contract and move to the next step if the send succeeds. This may incur attack if the send above always fail due to some unexpected reasons, such as the external contract does not accept ether or is forbidden tentatively or permanently. It is impossible for contract execution to move to next step if sending ether always fails.

To avoid exhaustion of GAS(EOG), we propose:

- Do not iterate a huge data structure in the contract. If the data structure can be manipulated from outside, the situation becomes even worse. We recommend to use withdrawal pattern which requires participants to withdraw their own tokens by themselves.
- We should have a fail-safe pattern to avoid malfunction of the contract owner. The recommendations are
  - Create a multi-sig contract and contract owner is only one of the signers.
  - Use time lock which allows contract execution after a certain period.
  - Add some time out logic for progressive state judgment.
- Create another operation account to solve contract malfunction problem caused by DoS attack.

Block stuffing is a type of attack in blockchains where an attacker submits transactions that deliberately fill up the block's gas limit and stall other transactions. To ensure the inclusion of their transactions by miners, the attacker can choose to pay higher transaction fees. By controlling the amount of GAS spent by their transactions, the attacker can influence the number of transactions that get to be included in the block.

To control the amount of GAS spent by the transaction, the attacker utilizes a special contract. There is a function in the contract which takes as input the amount of GAS that the attacker wants to burn. The function runs meaningless instructions in a loop, and either returns or throws an error when the desired amount is burned.

For example, assume the average GAS price has been 5 Gwei in the last 10 blocks. In order to exert influence over the next block, the attacker needs to submit transactions with GAS prices higher than that, say 100 Gwei. The higher the GAS price, the higher the chance of inclusion by miners. The attacker can choose to divide the task of using 8,000,000 GAS—current gas limit for blocks—into as many transactions as they want. This could be 80 transactions with 100,000 GAS expenditure, or four transactions with 2,000,000 GAS expenditure.

8.3 Ethereum Smart Contract: Best Practice

Developing secure smart contract is costly. There are already some good summaries of security guides, such as best practice from Consensys, and security guides from solidity official site. In this section, we explain some strategies to improve the security of smart contract.

8.3.1 Fail Early and Fail Loud

A simple and strong best practice is to expose the problem as early and explicit as possible. Next, have a look at unsafe contract below:

```
// Unsafe code, DO NOT USE
contract BadFailEarly {
  uint constant DEFAULT_SALARY = 50000;
  mapping(string => uint) nameToSalary;
  function getSalary(string name) constant returns (uint) {
    if (bytes(name).length != 0 && nameToSalary[name] != 0) {
      return nameToSalary[name];
    } else {
      return DEFAULT_SALARY;
    }
  }
}
```

In getSalary function, we need to check the parameter before we return. In the example above, the problem is the function will return the default value if the condition is not satisfied. Someone thinks it is a compatibility design. But it will cover the parameter error which may spread and blow in the future. And it is very hard to trace back the root cause. Here is an example:

```
contract GoodFailEarly {
   mapping(string => uint) nameToSalary;
   function getSalary(string name) constant returns (uint) {
    if (bytes(name).length == 0) throw;
    if (nameToSalary[name] == 0) throw;
    return nameToSalary[name];
   }
}
```

We recommend a coding pattern to separate each condition's check and error process. And we can use the modifier for code reuse purpose.

8.3.2 *Use Pull Instead of Push*

Every time when we transfer ether, we need to take involving contract and potential code execution into account: A receiver contract may or may not implement a fallback function; fallback function may throw an exception. So we always need to handle the situation that sends may fail. A solution to this problem is to use pull instead of push. Following is a bid example:

```
// Unsafe code, DO NOT USE
contract BadPushPayments {
  address highestBidder;
  uint highestBid;
  function bid() {
    if (msg.value < highestBid) throw;
    if (highestBidder != 0) {
      // return bid to previous winner
      if (!highestBidder.send(highestBid)) {
         throw;
      }
    }
    highestBidder = msg.sender;
    highestBid = msg.value;
  }
}
```

Contract above uses send function and checks its return value, which appears reasonable. But bear in mind, send function might trigger code in other contracts. For example, some bidder address might throw whenever it receives some ether. Since send will always fail and exceptions will be bubbled up, state will remain un-changed as before calling send function. This means that no one is able to continue bid and contract is un-functional. A simple solution to this is to move payment to a separate function and let each individual user sends pull request for withdrawing fund.

```
contract GoodPullPayments {
  address highestBidder;
  uint highestBid;
  mapping(address => uint) refunds;
  function bid() external {
    if (msg.value < highestBid) throw;
    if (highestBidder != 0) {
      refunds[highestBidder] += highestBid;
    }
    highestBidder = msg.sender;
    highestBid = msg.value;
  }
  function withdrawBid() external {
   uint refund = refunds[msg.sender];
   refunds[msg.sender] = 0;
   if (!msg.sender.send(refund)) {
     refunds[msg.sender] = refund;
   }
  }
}
```

The contract above uses a mapping variable to store the information of all bidders and provides a withdraw function for refund. If one sends fails, it does not influence the other send functions. It is a simple pattern but useful.

8.3.3 Condition, Behavior, Interaction

As an extension of the principle of exposing problem as early as possible, a best practice is to follow the function structure: first, check condition; then update the state of contract; and at last, interact with other contracts. It will avoid most of the security vulnerabilities if you use the order of condition, behavior, and then interaction. Let us have a look at the example below:

```
function auctionEnd() {
    // 1. Conditions
    if (now <= auctionStart + biddingTime)
       throw; // auction did not yet end
    if (ended)
       throw; // this function has already been called
     // 2. Effects
     ended = true;
     AuctionEnded(highestBidder, highestBid);
     // 3. Interaction
     if (!beneficiary.send(highestBid))
        throw;
     }
}
```

Code above check state and throw an exception if the required condition is not satisfied. And it leaves the risky part—interaction with other contracts, to the end of the function.

8.3.4 Test Case

It costs a lot of time to write test cases. But it will be finally paid off in saving time spent on regression test. Regression test is designed for finding bugs of components (which is bug-safe before) caused by newly added functionalities.

8.3.5 Fault Tolerance and Bounty Program

If contract passes the code audit and security review process, we should always be ready for the worst case. We should have the method to recover securely if a contract vulnerability is found. Furthermore, we need to identify vulnerability as early as possible. Here is an example demonstrating how a bounty process works. The example is a faked token contract with a bug bounty program.

```
import './PullPayment.sol';
import './Token.sol';
contract Bounty is PullPayment {
  bool public claimed;
  mapping(address => address) public researchers;
  function() {
    if (claimed) throw;
  }
  function createTarget() returns(Token) {
    Token target = new Token(0);
    researchers[target] = msg.sender;
    return target;
  }
  function claim(Token target) {
    address researcher = researchers[target];
    if (researcher == 0) throw;
    // check Token contract invariants
    if (target.totalSupply() == target.balance) {
      throw;
    }
    asyncSend(researcher, this.balance);
    claimed = true;
  }
}
```

As to fault tolerance, we need to add an extra security check. A simple solution is to allow the supervisor of the contract to freeze contract as an emergency mechanism. We can inherit from the following contract:

```
contract Stoppable {
  address public curator;
  bool public stopped;
  modifier stopInEmergency { if (!stopped) _; }
  modifier onlyInEmergency { if (stopped) _; }
  function Stoppable(address _curator) {
    if (_curator == 0) throw;
    curator = _curator;
  }
  function emergencyStop() external {
    if (msg.sender != curator) throw;
    stopped = true;
  }
}
```

Stoppable accept a supervisor which can stop the whole contract. We can inherit from Stoppable and apply modifier stopInEmergency and onlyInEmergency to correspondent functions. Here is an example:

```
import './PullPayment.sol';
import './Stoppable.sol';
contract StoppableBid is Stoppable, PullPayment {
  address public highestBidder;
  uint public highestBid;
  function StoppableBid(address _curator)
    Stoppable(_curator)
    PullPayment() {}
  function bid() external stopInEmergency {
    if (msg.value <= highestBid) throw;
    if (highestBidder != 0) {
      asyncSend(highestBidder, highestBid);
    }
    highestBidder = msg.sender;
    highestBid = msg.value;
  }
  function withdraw() onlyInEmergency {
   suicide(curator);
  }
}
```

In the example above, the bid can be stopped by a supervisor which is specified in the constructor. In a normal situation, only the bid function can be used; and in an emergent situation, the supervisor can step in and activate emergency mode which makes the bid function no longer working and activate withdraw function. Emergency mode enables supervisor to destroy the contract, recover fund. And in the real world, recover logic might be more complex (for example: return fund to individual investor).

8.3.6 Limit the Funds That Can Be Deposited

Another method to protect our contract from attack is restriction. From the return of investment (ROI) standpoint, an attacker is usually searching high-value contract. Setting the upper limit of fund which a contract hold will lower the risk of being attacked. A simple way to do this is to set hard cap. Here is an example:

```
contract LimitFunds {
  uint LIMIT = 1000;
  function() { throw; }
  function deposit() {
    if (this.balance > LIMIT) throw;
  ...
  }
}
```

In the fallback function, the contract above will reject all payments. Deposit function first will check if the current balance is above the upper limit: if yes, an exception will be thrown. A more realistic way is to set a dynamic limit and its implementation is very easy.

8.3.7 Simple and Modular Code

Security is the distance from the implementation to what the code is expected to do. Security is very hard to prove, especially when the code set is huge and complex. And this means:

- Function should small
- File should be small
- Code inter-dependency should be as less as possible
- Put independent logic into a separate module
- Each module should implement just one logic
- Use explicit and easy-to-understand name convention

Please remember, try to write your contract as simple as you can with a modular and explicit name convention. And it will help others to understand and audit your code.

8.3.8 Do Not Start Coding from Scratch

As an idiom says: do not re-invent the wheel. And it is also fit for writing a smart contract. Solidity programmers usually handle money; contract source code and data are publicly accessible and smart contract platform is in fast growing. As of now, there are already lots of verified secure code and framework (such as Open Zeppelin) in market, and there are also some supporting tools such as better type systems, Serenity Abstractions, and the Rootstock platform.

8.4 Code Audit

As the last resort, if you are not sure about how secure your contract is, or do not know how to improve it, you can submit the contract for code audit. It is not free and it is most likely expensive. There are many companies providing such service in market. For example: New Alchemy founded by blockchain foundation ex-partner—Peter Venesse, Zeppeline. Besides, there are also several teams doing a great job in formal verification of smart contract. Formal verification is to use mathematics form to prove smart contract will do exactly what it is expected to do.

8.5 Summary

In summary, this chapter describes the security patterns including:

1. Do not code from scratch and use existing secure framework as much as possible.
2. Fail as early and loudly as possible.
3. Favor pull over push payments.
4. Follow the pattern: conditions, behavior, interactions.
5. Pay attention to platform constraints.
6. Test, test, and test.
7. Set fault tolerance and bug bounty program.
8. Control the amount of funds deposited.
9. Organize the code in a simple and modularized way.

And pay attention to:

- Reentrancy: Make the external call at the end of the function if necessary.
- Send can fail: Must handle the situation when send fails.
- Loops can cross gas limit: When you iterate through state variables, gas usage will grow with the growing size of state variables. Please note that contract might be terminated because of the exhaustion of GAS.
- Call stack depth limit: Do not use recursive since stack has constraints on depth and recursion might fail.
- Timestamp dependency: Do not use the timestamp of blocks since miners can change them.

We have learned everything to develop Solidity contract. Usually, contract contains core logic and is not user friendly, we will introduce decentralized APP which interact with contract on-chain and present input and output in a user friendly way.

Part IV
Application

Chapter 9
Decentralized Application (DApp)

9.1 Feature

Decentralized Application (DApp) can be considered as a web application and key components distributed to P2P network. In this way, DApp lowers the risk of single point of failure (SPOF) and ensures the user experience. Logic components of DApp are listed below (Fig. 9.1).

Web Application Infrastructure
DApp has the similar feature as traditional web application. That is to say, DApp should be able to process Http Request and return Response.

Distribute Data
Data are saved on blockchain.

Distributed Business Logic
Business logic is implemented in smart contracts. Since the smart contract is deployed on blockchain, it is transparent and open to everyone. The downside is that it may incur attack.

Client Side Encryption
Since the private key is usually saved on the client side, many transactions need to be transferred after encryption.

DApp has the following features:

Lower the Risk of Single Point of Failure (SPOF)
The infrastructure on which current web applications depends, such as servers, code, databases, etc., has the possibility of SPOF in nature. Even Amazon and Aliyun service sometimes crashed. DApp can largely lower the SPOF risk and increase availability.

G. Zheng et al., *Ethereum Smart Contract Development in Solidity*,
https://doi.org/10.1007/978-981-15-6218-1_9

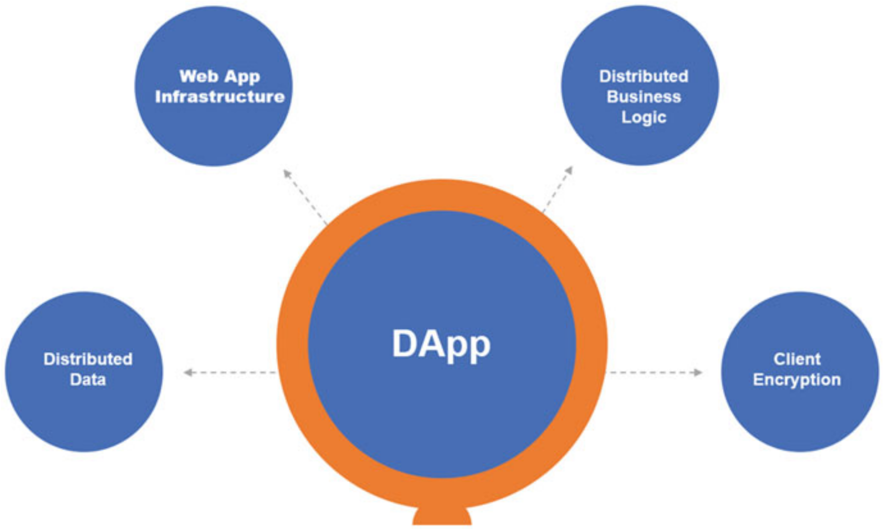


Fig. 9.1 Dapp logic structure

Lower the Dependency on Centralized Authority
Through smart contract, blockchain provides an untampered, immutable, and auditable environment. Any Dapp user can validate the business logic in smart contract including input, output, state, etc.

Improve Security
The introduction of client side encryption and DApp encryption largely improves security.

Network Effect
DApp uses public distributed ledger or distributed storage as the basis of trust to provide identity verification, authentication, authorization, and access control.

Distributed Data Storage
Distributed ledger, IPFS or Swarm. Data can be saved on multiple nodes.

Distributed Business Logic
Ethereum implements business logic in the smart contract, while Hyperledger uses similar technology—Chaincode. And the smart contract is run on distributed ledger.

Client Side Encryption
Blockchain wallet implements encryption on client side which makes DApp users can communicate with servers in a secure way or sign transaction data.

Here is a typical DApp overview picture (Fig. 9.2) from the user's point of view: user access web site through a browser which supports web3 library, web site read/write data from the database or directly from blockchain. Web site maintains an Ethereum full node or connect to an Ethereum full node.

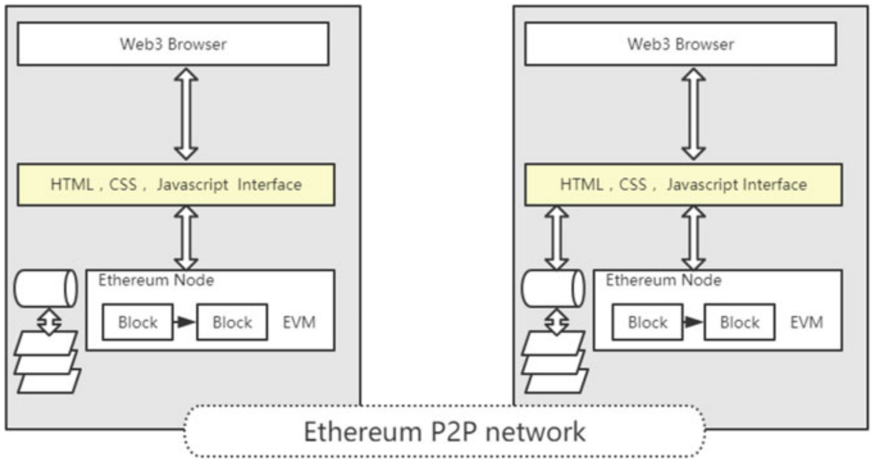


Fig. 9.2 Dapp user's view

Figure 9.3 contains more details which describe steps from the programmer's point of view. Contract creation (Steps 1–5) workflow is as below:

1. First, write the smart contract and save into a file with the extension .sol.
2. Compile contract, deploy to testrpc/ganache, and test it.
3. Return bytecode of contract if the test passes.
4. Deploy compiled contract bytecode to blockchain.
5. Return contract address and ABI.

The step to call contract function (Step 6):

6. Frontend call contract function with parameters: contract address, ABI, and nonce.

The major difference between DApp and the traditional program are:

- All DApp data is public and visible. Certainly, we can encrypt or hash DApp data to protect data. There is a terminology—code obfuscation for such purpose.
- Users need to pay some fee to run their transaction or the whole decentralized system will not work. And the payment amount depends on how many resources are consumed for contract execution. It is called gas cost.
- While considering paying fee for each transaction, user must understand that it will take some time to run the transaction. This is different with current centralized applications. Since the block size is fixed and transactions which each block can pack are not infinite, the higher the gas price, the higher the priority for transaction getting packed into block.

Just as building a normal web site or a mobile app, creating a DApp generally needs computation resources, file storage, external data, money, and payment system. Table 9.1 shows the progress of DApp tech stack in 2017.

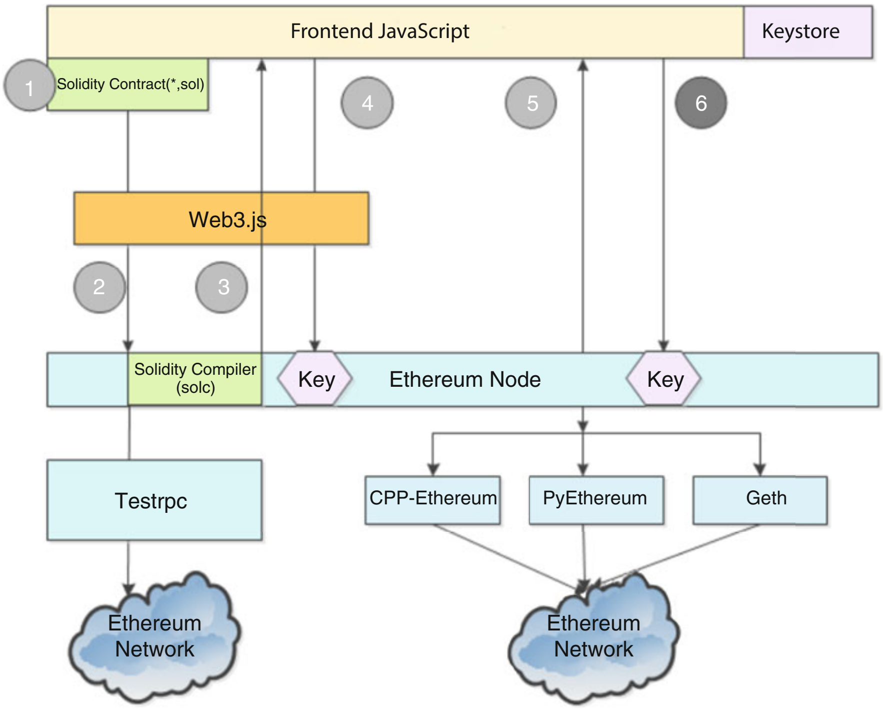


Fig. 9.3 Dapp programmer's view

Table 9.1 Dapp Tech stack status in 2017

	Web2.0	Web3.0	Status
Extensibility	Amazon EC2	Ethereum, Truebit	In progress
File storage	Amazon S3	IPFS/Filecoin, Storj	In progress
External Data	Third-party API	Oracle	In progress
Income	Advertisement, sales	Token model	Ready
Payment	Credit card, Paypal	Ethereum, Bitcoin, State Channel, 0x	Ready

9.2 DApp Architecture

What makes DApp different from traditional App?

- DApp's smart contract is not controlled by any centralized entity or organization, even not controlled by smart contract developer or sponsor.
- No backend.
- Transaction being broadcasted to blockchain and miners being responsible for executing the related smart contract.

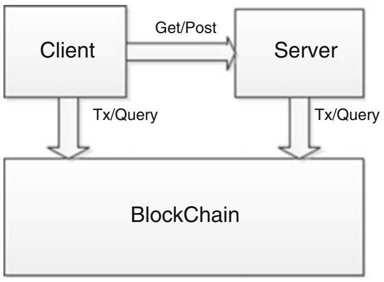


Fig. 9.4 Dapp program structure

- Smart contract forces some rules on data and money related operations. No one can change rules once the contract is deployed.
- Smart contract code must be transparent, auditable, and verifiable.
- Transaction must be transparent, auditable, and verifiable.
- Web site must use static files and depend on blockchain as the only database source.

We mainly discuss Serverless apps, browser plug-in, private node, off-line signature, and other related problems. Figure 9.4 depicts the structure of a DApp program.

9.2.1 Client Side

DApp Client can reside everywhere: static web pages, mobile phones, etc. These contents can be hosted by any cloud service providers, such as AWS, Aliyun, Google Cloud, GitHub, etc. If cloud service providers provide Swarm, Storj or IPFS, it is possible that all your contents (web page, resource, etc.) is distributed in a decentralized way.

How does client side software access blockchain? At the moment, most of DApp use web3 library. User can use web3 library to connect to blockchain, to query information, and to send transaction, etc. Alternative solution is to use official Mist software, or MetaMask browser plug-in.

The following code snippet demos how to check if web3 library is installed during page load.

```
window.addEventListener('load', function() {

  // Check if web3 object is injected into browser (Mist/MetaMask)
  if (typeof web3 !== 'undefined') {
    // if web3 object exists, then use Mist/MetaMask as provider
    window.web3 = new Web3(web3.currentProvider);
```

(continued)

```
   } else {
    // Plan B: if web3 object does not exist, then use local node +
  in-dapp id mgmt /)
    window.web3 = new Web3(new
  Web3.providers.HttpProvider("http://localhost:8545"));
   }

   // Start running your app
   startApp()
  })
```

9.2.2 Server Side

Why DApp still needs server side program? First, code on-chain cannot communicate with off-line service directly. If code on-chain need to interact with third-party services, such as external crypto exchange or send/receive email service, we will still need server side program.

Second, server side program can be used as buffer or index engine. It is common for client side program to provide search functionality or to validate on-chain data based on the server side service.

Last but not least, storage and calculation on Ethereum are expensive since they will cost gas. A typical solution to cut cost is to put storage and calculation offline, and to keep signature on-chain for validation purpose. Then, how server side program interact with blockchain?

9.2.2.1 Local Node

A simple method is to set up a local Ethereum node. All interactions with blockchain are through calling JSON RPC interface. If your program wants to update the state, you must keep an unlocked account at local. In this situation, you must ensure that JSON RPC interface held by a local node should not be used by other programs. If not, other people can access your node and steal your money. To be safe, it is better to keep as less money as possible in your unlocked account as shown in Fig. 9.5.

9.2.2.2 Off-line Signature

Another solution is to do off-line signature on transactions in client App and then send transactions to a public node. This solution chooses to believe public nodes blindly: public nodes cannot tamper transaction information and public nodes cannot

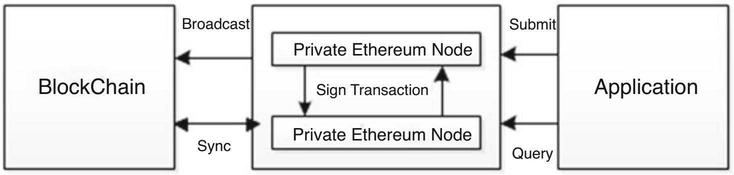


Fig. 9.5 Dapp node structure

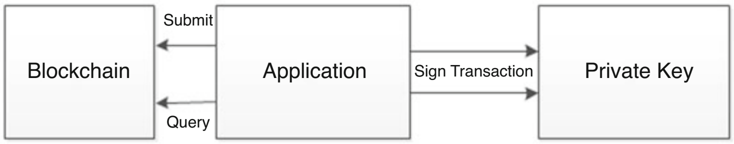


Fig. 9.6 Dapp transaction

selectively relay or not relay transaction or return false query information. In order to prevent public nodes from tampering, users can send transactions to multiple public nodes simultaneously. However, doing this way make programming much more complicated. Please refer to Fig. 9.6.

Such public nodes act a role as the gateway. Infura (infura.io) is one of those most popular nodes.

9.2.2.3 State-Change Transaction

Solutions above is good enough for query of blockchain. However, if client side operation will change the contract state on-chain, they are not flawless. As to send transaction, it is simple if there are mist or Metamask installed on client side: mist provides a gateway/account for local node used to sign transaction while Metamask signs transaction on client side and relay transactions to Metamask public nodes. If there is not Mist or Metamask on client side, client app needs to send transaction manually.

To run a specific contract function, we need users to send data with enough ether. We can get data of a function fairly easy through request method.

```
SimpleToken.at(tokenAddress).transfer.request
("0xbcfb5e3482a3edee72e101be938
7626d2a7e994a", 1e18).params[0].data

// Data return from transfer is:
'0xa9059cbb0000000000000000000000000000000000000000
000000000000000010000000000000000000000000000000
0000000000000000de0b6b3a7640000'
```

Please make sure that fallback function contains reasonable default fund processing in case that user lose money if they do something wrong inadvertently.

Another solution is to add a Proxy contract. Once receive some ether, the proxy contract will run a specific function in our main contract. For example, we can design a voting program for two options: in our main contract, there are a vote-yes and a vote-no function. We also deploy two supplementary contracts including the call to vote-yes or vote-no functions in the main contract only. This method fits for simple Dapp and largely reduces the coding complexity.

9.2.2.4 Implement Wallet

Another solution is to integrate wallet into applications. App creates a new account for user and send transaction by itself. In this way, user needs to deposit some ether beforehand into the new account and program will use these ethers to pay transaction fee. A more detailed explanation is: app holds the private key of the new account and sign transactions with this private key, and then send transaction to public nodes for execution.

This solution requires a lot of programming effort: create account and encryption, and import/export. Please refer to the implementation of Ethereum-wallet for details. All transactions must be manually created, signed, and sent to public nodes as raw transaction. And user also need to do more extra work: configure new account and save account profile. However, once configuration is completed, user enjoys the benefit of doing Ethereum transaction at zero cost with no need to install the third-party software or plug-in.

9.2.3 Workflow

Client and server side can access blockchain simultaneously. A typical solution is to send requests to blockchain and then listen on events. Here we can use observer pattern: Client side only listens on the event related to itself, while server side listen on all events. And we can distinguish event type through indexed parameter with the event. Figure 9.7 depicts the mechanism.

9.2.3.1 Contract Events

We can create event filter to monitor events selectively.

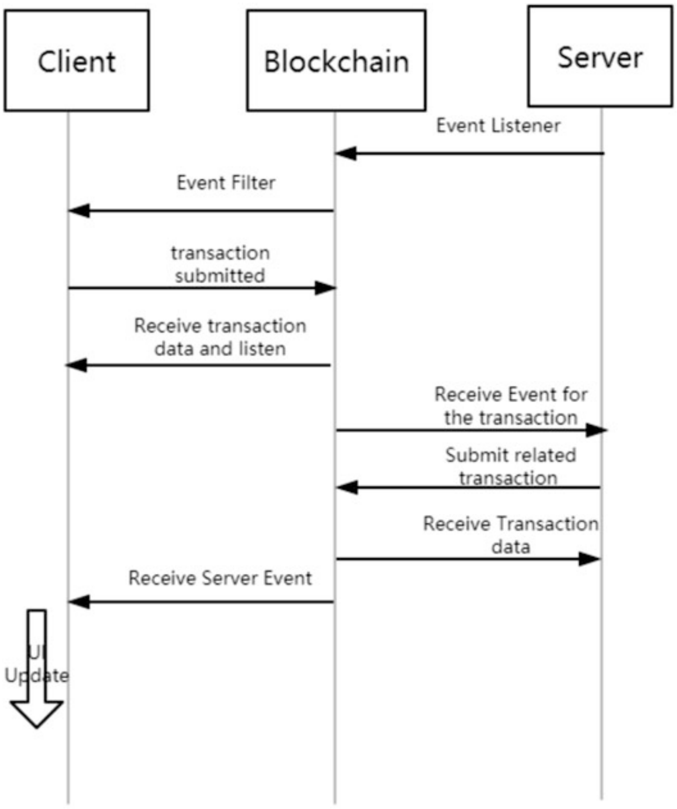


Fig. 9.7 Interaction between client/server/blockchain

```
var event = myContractInstance.myEvent({valueA: 23} [,
additionalFilterObject])

// Monitor change of event
event.watch(function(error, result){
 if (!error)
   console.log(result);
});

// Or pass a callback function to start monitoring
var event = myContractInstance.myEvent([{valueA: 23}] [,
additionalFilterObject] , function(error, result){
 if (!error)
   console.log(result);
});
```

1. Parameters

   Object—return value after filtering logs. For example: {“valueA”: 1, “valueB”: [myFirstAddress, mySecondAddress]}. Default filter value is set to null, which means to match all events related to the contract.

   Object—Filter option. Default filterObject has “address” field set to the contract address. And the first topic is event signature.

   Function—(Optional) if a callback function is passed as parameter, then event monitoring starts instantly with no need to call myEvent.watch(function(){}).
2. Callback return

   Object—Event object as below:
   address: String, 32 Bytes—Log source address
   args: Object—Event parameters
   blockHash: String, 32 Bytes—Block hash containing Log
   blockNumber: Number—block number containing Log
   logIndex: Number—Index of log in block
   event: String—Event name
   removed: bool—indicate the transaction has been removed from blockchain or not
   transactionIndex: Number– Transaction index for creating Log
   transactionHash: String, 32 Bytes—Transaction hash for creating Log
3. Example

   The following is an example about monitoring specific events.

```
// Using existing abi to create contract object
var MyContract = web3.eth.contract(abi);
// Map the contract object to a deployed contract address
var myContractInstance = MyContract.at
('0x78e97bcc5b5dd9ed228fed7a4887c0d7287344a9');

// Listen on event with {some: 'args'}
var myEvent = myContractInstance.myEvent({some: 'args'},
{fromBlock: 0, toBlock: 'latest'});
myEvent.watch(function(error, result){
 ...
});

// Get all the event log
var myResults = myEvent.get(function(error, logs){ ... });
...
// Stop listening and unload filter
myEvent.stopWatching();
```

9.2.3.2 allEvents

The following is an example about listening in all events:

```
// Get all contract related events
var events = myContractInstance.allEvents
([additionalFilterObject]);
// Monitor all changes
events.watch(function(error, event){
 if (!error)
   console.log(event);
});

// Or start monitoring through passing a callback function
var events = myContractInstance.allEvents
([additionalFilterObject], function(error, log){
 if (!error)
   console.log(log);
});
```

1. Parameters
   Object—Filter option. Default filterObject has address' field set to the contract address. And the first topic is event signature and does not support supplementary topics.
   Function—(Optional) If a callback function is passed as parameter, then start event listening instantly with no need to call myEvent.watch(function(){}).
2. Callback Return
   Object—Contract Events
3. Example

```
var MyContract = web3.eth.contract(abi);
var myContractInstance = MyContract.at
('0x78e97bcc5b5dd9ed228fed7a4887c0d7287344a9');

// Listen on event with {some: 'args'}
var events = myContractInstance.allEvents({fromBlock:
0, toBlock: 'latest'});
events.watch(function(error, result){
  ...
});

// Get all past log
events.get(function(error, logs){ ... });
...
// Stop listening and unload filter
events.stopWatching();
```

Client can query the status of transaction through transaction ID on-chain. In this case, please ensure that messages on client and server side are only notifications which could not be trusted blindly. Regardless of listening on specific event or all events, please ensure to start operation after enough confirmations have been reached. Even though a transaction is packed into a block, it is still possible that the transaction/block could be discarded because of blockchain re-Org. There are also several popular supporting services for EVM, such as: Filecoin or Storj as storage services, Truebit as off-line computation service, or Oraclize as oracle service, etc.

9.3 Ethereum DApp

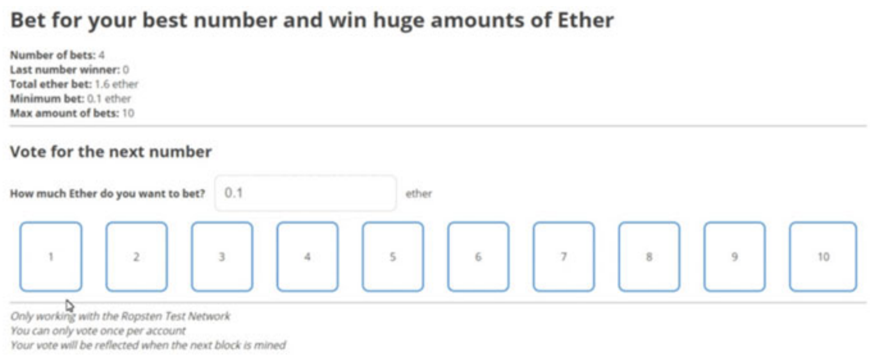
In this section, we are going to create a casino app. User can bet for a number between 1 and 10. After total number of bet reaches 100, winner can win a part of ether bet in the contract. The interface of the game is shown below (Fig. 9.8).

We demo some basic and complicate tricks for DApp development:

- How to deploy a smart contract from scratch
- How to deploy a smart contract to testnet
- How to create a frontend for a DApp
- How to connect to deployed smart contract on-chain from DApp
- How to deploy DApp to decentralized IPFS
- How to use customized IPFS domain name

Technologies we use for this example:

- *Database*: Ethereum testnet
- *Hosting*: Free hosting if using IPFS as a decentralized platform
- *Frontend*: React.js + webpack. Just for demo purpose. Any JavaScript framework can be chosen



Bet for your best number and win huge amounts of Ether

Number of bets: 4
Last number winner: 0
Total ether bet: 1.6 ether
Minimum bet: 0.1 ether
Max amount of bets: 10

Vote for the next number

How much Ether do you want to bet? 0.1 ether

Only working with the Ropsten Test Network
You can only vote once per account
Your vote will be reflected when the next block is mined

Fig. 9.8 Bid game interface (https://github.com/merlox/casino-ethereum)

- *Contract's programing language*: Solidity, most popular smart contract programming language up to date
- *Frontend contracts*: web3.js library used to access smart contract and its methods
- *Frameworks*: Truffle used to deploy, test and compile our smart contract
- *Development server*: Node.js used to develop APP locally, using testrpc/Ganache for testing
- *Metamask*: Browser plug-in for managing account

9.3.1 Environment Setup

First, we need to install necessary libraries: webpack, react, babel, and web3. We install version 0.20.0 of web3 (web3@0.20.0) since most up-to-date versions are 1.0 and it is still in Beta testing phase, which is unstable at the moment.

```
npm i -D webpack react react-dom babel-core babel-loader babel-
preset-
react babel-preset-env css-loader style-loader json-loader
web3@0.20.0
```

We also need to install a light-weighted web server so that we can access local web server: http://localhost:8080.

```
npm i -g http-server
```

9.3.2 Project

First, install truffle framework. In the following command, -D means dependency and -g means global.

```
npm i -D -g truffle
```

Next, we create Truffle project.

```
npm init -y
truffle init
```

Under project root directory, we create src directory and create js and css directory under it. Src directory is used to manage source code. Under js directory, we create index.js and index.css. Under Truffle root directory, we create dist directory, and create index.html in it.

File directory is like below:

```
contracts/
|-- Migrations.sol
|--migrations/
|--node_modules/
|--test/
|--src/
|-- css/index.css
|-- js/index.js
|--dist/
|-- index.html
|--package.json
|--truffle-config.js
|--truffle.js
|--webpack.config.js
```

The following is the home page of our DApp—index.html:

```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <meta name="viewport" content="width=device-width, initial-
scale=1.0">
   <link href='https://fonts.googleapis.com/css?family=Open
+Sans:400,700'
rel='stylesheet' type='text/css'>
   <title>Casino Ethereum Dapp</title>
</head>
<body>
   <div id="root"></div>
   <script src="build.js"></script>
</body>
</html>
```

<div id = “root” > </div> tag will contain all react codes injected automatically. <script src = “build.js” > </script> Tag contains the build file generated by webpack.

9.3.3 Solidity Programming

First, we create contracts/Casino.sol file which is the main source file of the solidity contract. All contracts start with the compiler version declaration. So at the beginning of Casino.sol, we specify the compiler version by pragma:

```
pragma solidity 0.4.20;
```

Next, we create the whole file:

```
pragma solidity 0.4.20;
contract Casino {
  address public owner;
  function Casino() public {  // Constructor
    owner = msg.sender; // Set contract Owner
  }
  function kill() public { // self destroy function. Only used by
owner
    if(msg.sender == owner) selfdestruct(owner);
  }
}
```

- Address type owner is a super-long string and the value is your Metamask account. Here we use 0xA4e71aEeAdaA01FD0f15455f67D7d2CAD32EbFcA.
- function Casino() is the constructor and we set the owner of the contract in it.
- function kill() is used to destroy the contract when necessary. Only the owner can destroy the contract. Once the contract is destroyed, remained ether in the contract will be returned to the owner's address. Only use kill() when the contract is hacked and owner cannot keep the fund in it.

We need to finish the following tasks:

- Record how much money a play bet on a number
- Minimum amount of bet required
- Calculate the total ether bet in the contract
- Variable to save how many people bet
- Determine when to stop bet and declare winner
- Develop a function to send bonus to winners' account

Main functionalities are:

- Bet a number between 1 and 10
- Create a random number and find winners based on it
- Send ether to winner

We use a struct type in which the mapping type variable is used to save the player address, bet and bet number. Struct type is like an object and mapping looks like an array.

Let us have a look at the code:

```
pragma solidity 0.4.20;
contract Casino {
   address public owner;
   uint256 public minimumBet; // minimum bet
   uint256 public totalBet; // Total bet
   uint256 public numberOfBets; // Number of bets
   uint256 public maxAmountOfBets = 100; // maximum number of bets
   address[] public players; // Player list
   struct Player { // definition of each player
      uint256 amountBet; // Amount of bet of each player
      uint256 numberSelected; // bet number of each player
   }
    // Mapping map player's address to his bet information
    mapping(address => Player) public playerInfo;
    function Casino() public {
       owner = msg.sender;
   }
   function kill() public {
      if(msg.sender == owner) selfdestruct(owner);
  }
}
```

Resource: https://github.com/merlox/casino-ethereum/blob/master/contracts/Casino.sol.

Player is a struct type which defines the amount of bet and bet number of each player. We use Player to save and track the player's bet amount and bet number. And then, we create a mapping named playerInfo.

```
 mapping(address => Player) public playerInfo;
```

We use address as a key to find its correspondent bid amount and bid number. And we use players (an array of players) to keep the information of each player so that we know how to distribute money to winners. Also, we modify constructor in which we define a minimum bet amount:

```
function Casino(uint256 _minimumBet){
owner = msg.sender;
   if(_minimumBet != 0 ) minimumBet = _minimumBet;
}
```

Now we create function Bet(): let the player bet a number between 1 and 10:

```
pragma solidity 0.4.20;
contract Casino {
   ...
   // To bet for a number between 1 and 10 both inclusive
   function bet(uint256 numberSelected) public payable {
      require(!checkPlayerExists(msg.sender));// Better must be a
player
      require(numberSelected >= 1 && numberSelected <= 10);//
number selected must between 1 and 10
      require(msg.value >= minimumBet);// Bet amount must larger
than minimumBet
      playerInfo[msg.sender].amountBet = msg.value; // save info.
In playerInfo
      playerInfo[msg.sender].numberSelected = numberSelected;
      numberOfBets++; // number of better increase by 1
      players.push(msg.sender); // Add player to players array
      totalBet += msg.value; // Add total bet amount
  }
}
```

Resource: https://github.com/merlox/casino-ethereum/blob/master/contracts/Casino.sol.

1. Keyword payable is a modifier indicating a function can accept ether.
2. require() is like if statement and it must return true or it will stop execution and return ether to initiator. We use the function require() to ensure that the player does not bet yet the player bet a number between 1 and 10 and bet amount is beyond the minimum Bet.
3. msg.sender and msg.value are info for the use which call bet(). Sender is the user address and value is amount bet.
4. We save information into mapping playerInfo[msg.sender].amountBet = msg.value;
5. msg.sender is the address of user which call bet().
6. We increase numberOfBets by 1. This variable is a counter recording the total bet count of the game. When bet count reaches 100, the game starts to calculate the winning number and distribute bonus accordingly.
7. At last, we add the bet amount to the total bet amount.

Now, look at the first require statement:

```
require(!checkPlayerExists(msg.sender));
```

We call checkPlayerExists() to check whether the player has bet since one player can only bet once. Let us start to write checkPlayerExists function as below:

```
pragma solidity 0.4.20;
contract Casino {
  ...
    // Go through player array. If existed, return true or return
false.
    // Only existed player can bet
   function checkPlayerExists(address player) public constant
returns(bool){
      for(uint256 i = 0; i < players.length; i++){
         if(players[i] == player) return true;
      }
      return false;
   }
   // To bet for a number between 1 and 10 both inclusive
    function bet(uint256 numberSelected) public payable {
       require(!checkPlayerExists(msg.sender));
       require(numberSelected >= 1 && numberSelected <= 10);
       require(msg.value >= minimumBet);
       playerInfo[msg.sender].amountBet = msg.value;
       playerInfo[msg.sender].numberSelected = numberSelected;
       numberOfBets++;
       players.push(msg.sender);
       totalBet += msg.value;
   }
}
```

Resource: https://github.com/merlox/casino-ethereum/blob/master/contracts/Casino.sol.

The constant keyword indicates that the function will not modify states and therefore will not cause gas consumption. This function is only to read a number.

As the next step, we are going to check whether the bet count exceeds the maximum number pre-defined. If bet count hit maximum count, then we start to generate the winning number.

```
if(numberOfBets >= maxAmountOfBets) generateNumberWinner();
```

generateNumberWinner() function is to generate a random number between 1 and 10:

```
pragma solidity 0.4.20;
contract Casino {
   ...
   // Geneate a random number between 1 and 10, and decide the winner
    function generateNumberWinner() public {
     uint256 numberGenerated = block.number % 10 + 1; // This random
```

(continued)

```
generator is not secure
    distributePrizes(numberGenerated);
  }
}
```

Resource: https://github.com/merlox/casino-ethereum/blob/master/contracts/Casino.sol.

The function above uses the last bit of current block number plus 1 as the winning number. For example, if current block number is 438542, then the winning number is 438542 % 10 = 2 and 2 + 1 = 3. Please note, this function is not secure since it is easy to guess winning number and miners can mine blocks selectively to benefit themselves.

Next, let us distribute bonus to winners. This is what distributePrices (numberGenerated) does:

```
pragma solidity 0.4.20;
contract Casino {
  ...
    // Return lottery to winners
    function distributePrizes(uint256 numberWinner) public {
       address[100] memory winners; // Create a memory winner array
       uint256 count = 0; // winner count
       for(uint256 i = 0; i < players.length; i++){
          address playerAddress = players[i];
           // If bid number is the same as winning number,
           // then save player info into winner array, count of
winner increase by 1
          if(playerInfo[playerAddress].numberSelected ==
numberWinner){
             winners[count] = playerAddress;
             count++;
          }
          delete playerInfo[playerAddress]; // Delete all players
       }
       players.length = 0; // Delete all player array
       uint256 winnerEtherAmount = totalBet / winners.length; //
calculate howmuch bonus each winner deserve
       for(uint256 j = 0; j < count; j++){
          if(winners[j] != address(0)) // Winner address should not
be 0
          winners[j].transfer(winnerEtherAmount);// call
transfer method to send bonus
       }
    }
}
```

Resource: https://github.com/merlox/casino-ethereum/blob/master/contracts/Casino.sol.

The function above mainly implement following:

1. Create winner array which is a memory array and will be released after jumping out of the function. Player will be added into winner array if player's numberSelected is the winning number.
2. Calculate the lottery amount based on the number of winners and bid amount for each winner.
3. Call winners[j].transfer to send ether to each winner.
4. Create an anonymous fallback function with the payable modifier. This function will return ether in contract to specified address:

```
// The objective of Fallback function is to ensure that fund in
contract will
// not be lost
   function() public payable {}
```

Fallback function allows us to save received ether. Please refer to this place[1] for full source code of Casino contract. Github has a higher version: it uses Oraclize service to generate a secure random number. We skip the design and implementation of Frontend.

9.3.4 Deployment

After installation of IPFS, we run the command below:

```
ipfs daemon
```

The command above will create an IPFS node. Next, let us type:

```
ipfs swarm peers
```

The command above allows your node's peers can share the content on your node.

```
ipfs add -r dist/
```

This command adds your dist directory into IPFS network. You should see long hash values generated for your directory. These hash values are unique ID of your directory files.

[1]https://github.com/merlox/casino-ethereum.

```
added   Qmc9HzLPur2ncuUArLjAaa4t2HrXFycgjUPb6122N6tzi2   dist/
build.js、
added QmZBaGYWsACJ5aCFhW459xHZ8hk4YazX1EQFiSenu3ANfR dist/
index.html
added QmfZoCnPcgmHYmFJqHBcyuFh3FEYrTZqGdGyioSMrAZzw2 dist
```

Copy hash value and run:

```
ipfs  name  publish  QmfZoCnPcgmHYmFJqHBcyuFh3FEYrTZqGdGyioSMr
AZzw2
```

You should see info like below:

```
Published to QmRDVed784YwKrYAgiiBbh2rFGfCUemXWk3NkD7nWdstER:
/ipfs/QmfZoCnPcgmHYmFJqHBcyuFh3FEYrTZqGdGyioSMrAZzw2
```

This says that your content has been saved on that URL. And you can check your content at gateway.ipfs.io/ipns/<your-hash-here>. For example:

```
gateway.ipfs.io/ipns/
QmRDVed784YwKrYAgiiBbh2rFGfCUemXWk3NkD7nWdstER
```

Since the current network is pretty large, it will take some time to see Dapp. Please remember to set Metamask to connect with Ropsten Test Network. If we modify some media file, we need to run webpack first, then run “ipfs add -r dist/”, and run “ipfs name publish <the-hash>” to publish the modification. Please note, hash should be the same as what is before the publishing.

9.4 IPFS DApp

In this section, we will introduce a simple implementation of Dapp based on IPFS.[2] This sample will demo how to save information to IPFS and how to interact with IPFS.

[2]https://medium.com/textileio/building-an-interplanetary-%C4%91app-from-scratch-51f9b8be5a74.

9.4.1 Environment Setup

First, we need to install browsersify and Babel library.

```
yarn add --dev browserify babelify babel-core babel-polyfill
babel-preset-env
```

We also need to install minify, lint, watch, simplify and node-ecstatic library.

```
yarn add --dev envify npm-run-all shx standard uglifyify watchify
ecstatic
```

To run our Dapp based on IPFS, we need to install a browser plug-in:

```
Google Chrome:
https://chrome.google.com/webstore/detail/ipfs-companion/
nibjojkomfdiaoajekhjakgkdhaomnch

Firefox:
https://addons.mozilla.org/en-US/firefox/addon/ipfs-companion/
```

Using the browser plug-in is optional. There will be more extra dev work if plug-in is not installed. Here, our purpose is to demo how to create an IPFS-based Dapp. So we focus on Dapp development and use IPFS plug-in to save some time.

We use window.ipfs-fallback to instantiate an IPFS node. And we use js-libp2p-crypto library to do encryption. IPFS browser plug-in will insert a window.ipfs object for each web page. In this way, any IPFS-based Dapp can detect the existence of window.ipfs. Window.ipfs-fallback library can detect Window.ipfs object automatically: if loaded, then use it; or automatically revert to the most up-to-date IPFS version downloaded from CDN.

```
yarn add window.ipfs-fallback js-libp2p-crypto
```

9.4.2 Project

First, create project directory:

```
mkdir encryptoid
cd encryptoid
```

We use NPM to initialize project. We need to answer some project-related questions.

```
ubuntu@VM-16-5-ubuntu:~/work/dapp/encryptoid$ npm init
```

Directory structure is like below:

```
Encryptoid
|____LICENSE
|____README.md
|____package.json
|____src
| |____index.html
| |____images
| | |____logo.png
| |____main.js
| |____style.css
|____dist
| |____...
```

Next, we are going to modify package.json to add some compilation commands.

```
...
"scripts": {
    "start": "ecstatic dist",
    "clean": "shx rm -rf dist",
    "build": "run-s build:*",
    "build:copy": "run-p build:copy:*",
    "build:copy:html": "shx mkdir -p dist && shx cp src/index.html
dist/index.html",
    "build:copy:css": "shx mkdir -p dist && shx cp src/style.css
dist/style.css",
    "build:js": "browserify src/main.js -o dist/bundle.js -g
uglifyify",
    "watch": "npm-run-all build:* --parallel watch:*",
    "watch:js": "watchify -t envify src/main.js -o dist/bundle.js
-v",
    "watch:serve": "ecstatic --cache=0 dist",
    "test": "standard"
  },
```

(continued)

```
   "browserify": {
     "transform": [
       ["babelify", {"presets": ["env"]}],
       ["envify"]
     ]
   },
...
```

Resource: https://github.com/textileio/encryptoid/blob/master/package.json.

In package.json, we provide all kind of start commands, such as: start, clean, build, test, etc.

Entry file: Index.html

```
<!doctype html>
<html>
<head>
  <meta charset="utf8">
  <title>Encryptoid App</title>
  <link rel="stylesheet" href="./style.css">
</head>
<body>
  <div id="main">
    <h1>Welcome to Encryptoid!</h1>
  </div>
  <script type="text/javascript" src="bundle.js"></script>
</body>
</html>
```

Resource: https://github.com/textileio/encryptoid/blob/master/src/index.html.

CSS style sheet: Style.css

```
html, body {
  font-family: "Lucida Sans Typewriter", "Lucida Console",
"Bitstream Vera Sans Mono", monospace;
  height: 100%;
}
#main {
  width: 50%;
  margin: 0 auto;
  padding-top: 2%;
}
```

Resource: https://github.com/textileio/encryptoid/blob/master/src/style.css.

Main.js.

We use Async/await model to get IPFS node information in asynchronization way.

```
// Import library we need
import 'babel-polyfill' // We need this for async/await polyfills

// Import 2 IPFS-based templates
import getIpfs from 'window.ipfs-fallback'
import crypto from 'libp2p-crypto'

let ipfs

// Create async setup function which run on page load
const setup = async () => {
  try {
    ipfs = await getIpfs() // Initialize an IPFS Peer node
    const id = await ipfs.id() // Get Peer node id
    console.log(`running ${id.agentVersion} with ID ${id.id}`)
  } catch(err) {
    console.log(err) // Just pass along the error
  }
}
setup()
```

Resource: https://github.com/textileio/encryptoid/blob/master/src/main.js.

Next, we add to input boxes to index.html: one is <textarea> and the other is <input>.

```
 <h1>Welcome to Encryptoid!</h1>
   <form id="secret">
     <label for="message">Message</label>
     <textarea id="message" required rows="5"></textarea>
     <label for="password">Password</label>
     <input id="password" type="text" required minlength="10"
maxlength="100">
   </form>
   <button id="button" type="button">Encrypt</button>
   <div id="output"></div>
```

Resource: https://github.com/textileio/encryptoid/blob/master/src/index.html.

We also add some logic to process two input boxes we add above.

```
// Import library we need
import 'babel-polyfill' // We need this for async/await polyfills
```

(continued)

```
// Import 2 IPFS-based template
import getIpfs from 'window.ipfs-fallback'
import crypto from 'libp2p-crypto'

let ipfs

// Setup a very simple async setup function to run on page load
const setup = async () => {
  try {
    ipfs = await getIpfs() // Init an IPFS Peer node
    const button = document.getElementById('button')
    // Listen on button click event
button.addEventListener("click", (e) => {
      e.preventDefault()

      // Get input message and password
      const message = document.getElementById('message')
      const password = document.getElementById('password')

      // Compute derived key for AES Encryption
      const key = crypto.pbkdf2(password.value, 'encryptoid',
5000, 24, 'sha2-256')

      // We just use iv once
      const iv = Buffer.from([...Array(16).keys()])
      // Create AES object
      crypto.aes.create(Buffer.from(key), iv, (err, cipher) => {
        if (!err) {
          // Call encryption function
          cipher.encrypt(Buffer.from(message.value), async (err,
encrypted) => {
            if (!err) {
          // if no error, then save encrypted content into hashed for
later display
             const hashed = (await ipfs.files.add(encrypted))[0]
             output.innerText = `/ipfs/${hashed.hash}`
            }
          })
        }
      })
    })
  } catch(err) {
   console.log(err) // Just pass along the error
  }
}
setup()
```

Resource: https://github.com/textileio/encryptoid/blob/master/src/main.js.

- We use PBKDF2 (Password-Based Key Derivation Function 2) to create derivation Key which is used for subsequent encryption. In this example, we use fixed salt and 5000 iterations. The size of key is 24 bytes.
  const key = crypto.pbkdf2 (password.value, ‘encryptoid’, 5000, 24, ‘sha2–256’)
- We create a 16 bytes fixed-length Initial Vector (IV). For the real use, we had better use a random IV.
  const iv = Buffer.from([...Array(16).keys()])
- We create AES object (Since we use length 32 bytes, we can use AES 256).
  crypto.aes.create(Buffer.from(key), iv, (err, cipher) =>
- We pass plain text buffer to encrypt method and encrypted text will be returned to our callback function.

```
if (!err) {
          const hashed = (await ipfs.files.add(encrypted)) [0]
          output.innerText = `/ipfs/${hashed.hash}`
}
```

- We put encrypted messages on IPFS and wait until CID hash returned.
- Our output is CID hash. We send CID hash to the receiver and they can decrypt the content.

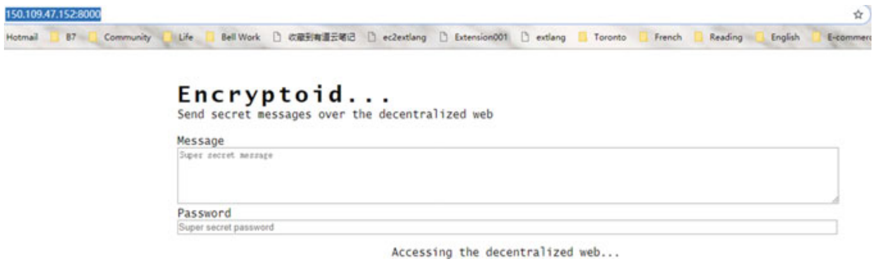
9.4.3 Compile

So, under the root directory of encryptoid project, we use the following command to compile:

```
Install dependencies: yarn install
Compile APP: yarn build
Start App: yarn start
```

Using any browser, we input http://server:8000/ you should see the interface like Fig. 9.9:

Now we have a simple DApp: it can encrypt information and propaganda through a distributed Web. Since we are running DApp in the browser and we do not pin them, DApp files will be garbage-collected after 24–48 h. But the most important is: encrypted message on IPFS P2P network is accessible and no centralized server can peek message. And this is the power of decentralization.



Encryptoid...

Send secret messages over the decentralized web

Message

Super secret message

Password

Super secret password

Accessing the decentralized web...

Fig. 9.9 IPFS Dapp interface (https://github.com/textileio/encryptoid)

Since you have reached here, you should know how to develop DApp with decentralized storage. Developing a large and complex DAPP is usually error-prone and we will introduce debug tools and mechanisms in the following chapter.

Chapter 10
Debug

10.1 Solidity Language

Like other programming languages such as Java, C++, Solidity has some built-in mechanisms for debugging purpose.

10.1.1 Event

A Solidity event can be defined as follow:

```
event Deposit(
   address indexed _from,
   bytes32 indexed _id,
   uint _value
);
```

- Maximum three indexed parameters.
- If an indexed parameter's is bigger than 32 bytes (eg: string and bytes), KECCAK256 abstract (Digest) will be saved instead of the actual data.

10.1.1.1 EVM Log Primitives

Ethereum Virtual Machine (EVM) has log0, log1, log2,log3, log4 opcode. EVM log primitives have the following parameters:

- "topics": maximum four topics with 32 bytes long each.
- "data": data is the payload of event and it can be any size of bytes

G. Zheng et al., *Ethereum Smart Contract Development in Solidity*,
https://doi.org/10.1007/978-981-15-6218-1_10

How a Solidity event is mapped to a log primitive?

- All non-indexed parameter will be saved as data.
- Any indexed parameter will be saved as a 32 byte topic

10.1.1.2 The log0 Primitive

Log0 primitive creates a log item with only data field and no topic field. Data Field can be any length of bytes.

Here is an example

```
pragma solidity ^0.4.18;
contract Logger {
  function Logger() public {
    log0(0xc0fefe);
  }
}
```

Now, let us check assembly code after compilation. 0x40 saves the value of free memory pointer. The first part is to load data into memory, while the second part is to prepare data on the top of stack.

```
memory: { 0x40 => 0x60 }

tag_1:
  // copy data into memory
  0xc0fefe
    [0xc0fefe]  // constant
  mload(0x40)
    [0x60 0xc0fefe]
  swap1
    [0xc0fefe 0x60]
  dup2
    [0x60 0xc0fefe 0x60]
  mstore
    [0x60]
    memory: {
      0x40 => 0x60k8    0x60 => 0xc0fefe
    }

// calculate data start position and size
  0x20
    [0x20 0x60]
  add
    [0x80]
```

(continued)

```
  mload(0x40)
    [0x60 0x80]
  dup1
    [0x60 0x60 0x80]
  swap2
    [0x60 0x80 0x60]
  sub
    [0x20 0x60]
  swap1
    [0x60 0x20]
log0
```

Before execution of log0, there are two parameters on stack: {0x60 0x20}.

- Start: 0x60 is a memory pointer for storing data.
- Size: 0x20 (or 32) specify data size.

10.1.1.3 Logging with Topics

The following example uses log2 primitive. The first parameter is data (can be any bytes), followed by two topics (32 bytes each):

```
// log-2.sol
pragma solidity ^0.4.18;

contract Logger {
  function Logger() public {
    log2(0xc0fefe, 0xaaaa1111, 0xbbbb2222);
  }
}
```

Assembly code is very similar. The only difference is that the two topics (0xbbbb2222, 0xaaaa1111) are pushed to stack:

```
tag_1:
      // push topics
      0xbbbb2222
      0xaaaa1111
      // copy data into memory
      0xc0fefe
      mload(0x40)
      swap1
      dup2
```

(continued)

```
      mstore
      0x20
      add
      mload(0x40)
      dup1
      swap2
      sub
      swap1

      // create log
      log2
```

Data is still 0xc0fefe and is copied to memory. Status before the execution of log2 is as below:

```
stack: [0x60 0x20 0xaaaa1111 0xbbbb2222]
memory: {
  0x60: 0xc0fefe
}
log2
```

The first two parameters specify the memory area of log data and the two new elements on top of stack is two topics, where each of them has 32 bytes.

10.1.1.4 All EVM Logging Primitives

EVM support five log primitives:

```
0xa0 LOG0
0xa1 LOG1
0xa2 LOG2
0xa3 LOG3
0xa4 LOG4
```

10.1.1.5 Logging Testnet Demo

```
pragma solidity ^0.4.18;

contract Logger {
  function Logger() public {
    log0(0x0);
    log1(0x1, 0xa);
    log2(0x2, 0xa, 0xb);
    log3(0x3, 0xa, 0xb, 0xc);
    log4(0x4, 0xa, 0xb, 0xc, 0xd);
  }
}
```

This contract is deployed to Rinkeby test net and URL is: https://rinkeby.etherscan.io/tx/0x0e88c5281bb38290ae2e9cd8588cd979bc92755605021e78550fbc4d130053d1

10.1.1.6 Solidity Events

Here is a log event with three non-indexed parameters (uint256 type):

```
pragma solidity ^0.4.18;
contract Logger {
  event Log(uint256 a, uint256 b, uint256 c);
  function log(uint256 a, uint256 b, uint256 c) public {
    Log(a, b, c);
  }
}
```

Generated log is located at:
https://rinkeby.etherscan.io/tx/0x9d3d394867330ae75d7153def724d062b474b0feb1f824fe1ff79e772393d395
Data is the parameter of event and is ABI encoded:

```
000000000000000000000000000000000000000000000000000000000000
0001
000000000000000000000000000000000000000000000000000000000000
0002
000000000000000000000000000000000000000000000000000000000000
0003
```

There is a topic with 32 byte hash:

```
0x00032a912636b05d31af43f00b91359ddcfddebcffa7c15470a13ba1992
e10f0
```

This is the SHA3 hash of Event type signature:

```
# Install pyethereum
# https://github.com/ethereum/pyethereum/#installation
> from ethereum.utils import sha3
> sha3("Log(uint256,uint256,uint256)").hex()
'00032a912636b05d31af43f00b91359ddcfddebcffa7c15470a13
ba1992e10f0'
```

Since one topic is used for event signature, there are only three topics left for indexed parameters.

10.1.1.7 Solidity Event with Indexed Arguments

Following is an event definition with indexed parameter of type uint256:

```
pragma solidity ^0.4.18;

contract Logger {
  event Log(uint256 a, uint256 indexed b, uint256 c);
  function log(uint256 a, uint256 b, uint256 c) public {
    Log(a, b, c);
  }
}
```

In above example, there are two topics:

```
0x00032a912636b05d31af43f00b91359ddcfddebcffa7c15470a13ba1992
e10f0
0x0000000000000000000000000000000000000000000000000000000000
0002
```

- The first topic is a function signature.
- The second topic is the value of indexed parameter

Except indexed parameter, all other data are ABI encoded:

```
0000000000000000000000000000000000000000000000000000000000
00001
0000000000000000000000000000000000000000000000000000000000
000003
```

10.1.1.8 String/Bytes Event Parameter

Following code set event's parameter to string:

```
pragma solidity ^0.4.18;

contract Logger {
  event Log(string a, string indexed b, string c);
  function log(string a, string b, string c) public {
    Log(a, b, c);
  }
}
```

The transaction is located at: https://rinkeby.etherscan.io/tx/0x21221c2924bbf1860db9e098ab98b3fd7a5de24dd68bab1ea9ce19ae9c303b56

There are two topics:

```
0xb857d3ea78d03217f929ae616bf22aea6a354b78e5027773679b7b4a6f
66e86b
0xb5553de315e0edf504d9150af82dafa5c4667fa618ed0a6f19c69b4116
6c5510
```

- The first topic is a function signature.
- The second topic is a SHA256 digest of string parameter.

In python console, type the following code to verify that the hash value of "b" is the same as the second topic value:

```
>>> sha3("b").hex()
'b5553de315e0edf504d9150af82dafa5c4667fa618ed0a6f19c69b411
66c5510'
```

Log data has two non-indexed string "a" and "c," which is ABI encoded:

```
0000000000000000000000000000000000000000000000000000000000
00040
0000000000000000000000000000000000000000000000000000000000
00080
000000000000000000000000000000000000000000000000000000000
000001
6100000000000000000000000000000000000000000000000000000000
00000
0000000000000000000000000000000000000000000000000000000000
00001
630000000000000000000000000000000000000000000000000000000
000000
```

Indexed string parameter is not saved and therefore DApp client cannot restore it.

If you really need them, the solution is to save them twice: indexed and non-indexed:

```
event Log(string a, string indexed indexedB, string b);
Log("a", "b", "b");
```

10.1.2 Assert/Require

Starting from version 0.4.10, Solidity introduced statements: assert (), require (), and revert (). We use if. . .throw pattern below prior to version 0.4.10:

```
contract HasAnOwner {
 address owner;

   function useSuperPowers(){
      if (msg.sender != owner) { throw; }
      // do something only the owner should be allowed to do
   }
}
```

If the caller is not the owner, function will throw an error—Invalid opcode. Keyword throw is obsoleted and we use assert, require, and revert instead.

```
if(msg.sender != owner) { throw; }
```

is equal to the following three statements:

```
if(msg.sender != owner) { revert(); }
assert(msg.sender == owner);
require(msg.sender == owner);
```

10.1.2.1 Difference Between Assert and Require

Assert is used to check a status totally unexpected and to ensure that contract can behave normally under unexpected situation. For example: divided by zero, over/underflow, etc. require will return unused GAS if failed.

require() is fit for situations below:

- Check user input
- Check response from external contract, For example “require (external.send (amount))”
- Check condition before state update
- Use require as much as possible
- Use require at the beginning of function

Use assert() in the following situations:

- Check overflow/underflow
- Check invariable value
- Check contract status after some modification
- Avoid impossible situations
- Use assert as less as possible
- Use assert at the end of function

In general, assert is used to avoid the worst scenario, and assert condition should not be false.

Use revert() in the following situations:

- Mostly like require, but can be used to process complex logic

Opcode

- assert() uses 0xfe opcode to raise an error
- require() uses 0xfd opcode to raise an error

10.1.2.2 Revert

10.1.2.2.1 Revert to Return Some Information

We can use revert to return some error-related information as the following:

```
revert('Something bad happened');
```

or

```
require(condition, 'Something bad happened');
```

10.1.2.2.2 Revert Return Unused GAS

Throw will exhaust all remained GAS and this becomes candid donation to miners. The downside is that this will lead to loss of money of user. Comparatively, REVERT will return unspent gas.

10.1.2.3 Choice Between Revert(), Assert(), and Require()

If revert() and require() return GAS with remaining value, why we still need assert() which will exhaust all gas? The difference is bytecode output: require function should be used to ensure correct condition, such as input, contract state variable or validate the return from external contract, etc. If require is used properly, analytic tool can evaluate your contract, identifying conditions and function calls which may cause assert failure. A correct function code will not raise an assert statement; but if it does happen, this means contract contains severe error. To be clear: if require statement fails, it is normal and healthy. If assert() fails, that mean some severe situation occur and developer must fix their program.

If contract follows the programming guide, static analysis and formal verification tool can check your contract, find or prove some specific cases which could fail the contract, or prove that your contract is run as expected. So, we use require() to check conditions and use assert() to avoid exceptions.

10.1.3 Test Case

Experienced programmers always have a good habit of writing unit test case. Now, we are going to write some test case by using Mocha framework + Nodejs to test ERC721 program in Sect. 4.3.1. Source code can be downloaded from: https://github.com/AnAllergyToAnalogy/ERC721

```
npm install --save mocha ganache-cli web3
```

Next, we need to edit package.json and modify it as below:

```
"test": "echo \"Error: no test specified\" && exit 1"
```

to:

```
"test": "mocha"
```

Under project root, we create two sub directories:

```
{your project directory}/test
```

and

```
{your project directory}/contracts
```

We put all compiled contract code in json format under contracts directory. It looks like: (refer to Sect. 4.3.1 for the content):

- TokenERC721.json
- ValidReceiver.json
- InvalidReceiver.json

And last, we create Token.test.js file under test directory. We declare some stuffs in it, which is pretty straightforward.

```
const assert = require('assert');
const ganache = require('ganache-cli');
const Web3 = require('web3');
const provider = ganache.provider({
 gasLimit: 10000000
});

const web3 = new Web3(provider);

const compiledToken = require('../contracts/TokenERC721.json');
const compiledValidReceiver = require('../contracts/
ValidReceiver.json');
const compiledInvalidReceiver = require('../contracts/
InvalidReceiver.json');
```

Using Mocha, we can use beforeEach to call some code before each test case. We plan to deploy a contract before running each test case. We declare accounts, token and initialTokens globally for subsequent test case access.

```
let accounts;
let token;
const initialTokens = 10;
beforeEach(async () => {
   accounts = await web3.eth.getAccounts();

   token = await new web3.eth.Contract(JSON.parse(compiledToken.
interface))
     .deploy({
        data: compiledToken.bytecode,
        arguments: [initialTokens]
    })
     .send({from: accounts[0], gas:'8000000'});
   token.setProvider(provider);
});
```

In our test file, each test case has test code in the form of *describe* … *it*…:

```
describe('Token Contract',() => {
   //All our test cases go here

   it('Example test case', () => {
      assert(true);
   });
});
```

If you want to run the test file, you need to go to the project root and run the following command:

```
npm run test
```

We use async and await pattern very often. These functions just make async calls look like “synchronized” a little. Let us have a look at the example below:

Testing creator’s Balance of creator == initial token supply.

```
it('Balance of creator == initial token supply', async () => {
   const balance = await token.methods.balanceOf(accounts[0]).
call();
   assert(balance == initialTokens);
});
```

Testing creator is able to issue tokens.

```
it('Creator can issue tokens', async () => {
   const toIssue = 2;
   const owner = accounts[0];
   await token.methods.issueTokens(toIssue).send({
      from: owner
   });
   const finalBalance = await token.methods.balanceOf(accounts
[0]).call();
   assert((initialTokens + toIssue) == finalBalance);
});
```

10.2 Testrpc/Ganache

In Sect. 2.2.1.3, we learned how to install Testrpc/Ganache. Ganache is a private blockchain running on local computer of Ethereum developer. Ganache provides a command-line interface which is fit for test automation and continuous integration. Ganache CLI can be configured to meet all your development needs. Ganache CLI can process transactions quickly due to no need to wait block confirmation.

We can use truffle to connect to local Ganache testnet. In order to do this, we need to modify truffle-config.js file:

```
module.exports = {
  networks: {
    development: {
     host: "localhost",
     port: 8545,
     network_id: "*" // Match any network id
    },
  }
};
```

Then, we need to deploy the contract to specified network. Here we use—network switch.

```
truffle migrate --network testenv
```

Description of the startup parameters of Ganache:

-account: specify private key and account balance to create initial testing accounts. And it can be set multiple times:

```
$ ganache-cli --account="<privatekey>,balance" [-
account="<privatekey>,balance"]
```

Please note: the length of private key is 64 and private key is a hex string starting with 0x. Account balance can be an integer and it also can be hex string starting with 0x. And the unit is wei. Ganache will not create HD wallet if it is started with–account.

-u or –unlock: Unlock specified account, or unlock account with specific series number. The flag can be set multiple times. And it will change the locking state of specified account when it is used with -secure.

-a: specify testing account number when startup.

-e: ether given to each testing account. Default value is 100 ether.

-b: Specify blockTime of automatic mining. And the unit is second. Default value is 0, which means automatic mining is turned off.

-d: Create testing accounts based on mnemonic.

-n: By default, lock all test account for third-party signature.

-m: Used to create mnemonic keyword for test account.

-p: Setup listening port, default value is 8545.

-h: Setup monitoring machine, default value is the same as server.listen() in Nodejs.

-s: Setup seed of Mnemonic words.

-g: Set gas price and default value is 20,000,000,000.

-l: Set upper bound of Gas and default value is 90,000.

-f: Fork chain from a running Ethereum client instance. The input value should be node's HTTP address and Port, For example: http://localhost:8545. @ can be used to specify block number. For example: http://localhost:8545@3645200.

-i or –networkId: specify network ID. Default value is current time, or forked chain's network ID.

-db: Set directory for storing chain data. If specified directory contains chain data, ganache-cli will use it to initialize chain and chain will not be built from scratch.

-debug: Output VM opcode for testing purpose.

-mem: Output statistics information for ganache-cli memory usage, this will replace standard output.

-noVMErrorsOnRPCResponse: do not send failed transactions as RCP error. Turning on this flag will make error report compatible with other clients, such as geth and Parity.

10.3 Truffle Debug

Truffle Debugger is a command-line tool in truffle framework. And it supports to debug smart contracts written in Solidity. When debugger is started up, command-line prompts a transaction or address list created, a transaction entry and available command list. Next, we will learn debug commands in details.

10.3.1 Debugging Interface

After the startup of Truffle debugger, you should see:

- Transaction list
- Possible command list
- Transaction entry

Return key is set to execute the latest command. After debugger starts, return key will execute to next logic unit. And you can hit return key to execute the whole transaction step by step. And at the same time, you can analyze transaction details. Command list is as below:

10.3.1.1 (o) Step Over

This command executes current line in Solidity source code, or relative line, or current expression. If you do not want to step into a function call or contract creation, or debug quickly, you can use this command.

10.3.1.2 (i) Step Into

This command steps into a function call or contract creation process. Using this command is to jump into function and debug quickly.

10.3.1.3 (u) Step Out

This command jumps out the current function. Using this command to return to caller function or terminate the execution of the current transaction.

10.3.1.4 (n) Step Next

This command debugs to the next statement or next expression. For example, the calculation of subexpression will be before calculation of EVM full expression. Using this command to analyze the calculation logic of EVM.

10.3.1.5 (;) Step Instruction

This command allows you to debug each opcode in EVM. It is useful if you have interests in compiled bytecode. If you use this command, debugger will print stack data related to this opcode.

10.3.1.6 (p) Print Instruction

This command prints current opcode and stack data but it does not jump to next opcode. This command print current opcode and stack data. But it does not jump to next opcode. Using this command if you want to check current opcode and stack data after the execution of some logic in the contract.

10.3.1.7 (h) Print This Help

Print available command.

10.3.1.8 (q) Quit

Exit Debugger.

10.3.1.9 (r) Reset

Reset debugger and get back to the beginning of contract.

10.3.1.10 (b) Set a Breakpoint

This command allows you to set breakpoint in contract source code at will. Breakpoint can be set through line number, relative line number, or line number in a specified file, or current location.

10.3.1.11 (B) Remove a Breakpoint

This command allows you to delete any existing breakpoint. The syntax is very like (b) command. Type “B all” to delete all breakpoints.

10.3.1.12 (c) Continue Until Breakpoint

This command will run until next breakpoint or the end of the program.

10.3.1.13 (+) Add Watch Expression

This command adds a watch expression and sytax is: +:<expression>.

10.3.1.14 (−) Remove Watch Expression

This command removes a watch expression. Syntax is like: -:<expression>.

10.3.1.15 (?) List Existing Watch Expressions

This command shows all expressions in watch.

10.3.1.16 (v) Display Variables

This command shows current variables and their value. Not all data types support this command.

10.3.2 Adding and Removing Breakpoints

Here are some examples to add/remove breakpoint.

```
11: event Generated(uint n);
12:
13: function generateMagicSquare(uint n)

 ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

(continued)

```
debug(develop:0x91c817a1...)> b 23
Breakpoint added at line 23.

debug(develop:0x91c817a1...)> B 23
Breakpoint removed at line 23.

debug(develop:0x91c817a1...)> b SquareLib:5
Breakpoint added at line 5 in SquareLib.sol.

debug(develop:0x91c817a1...)> b +10
Breakpoint added at line 23.

debug(develop:0x91c817a1...)> b
Breakpoint added at this point in line 13.

debug(develop:0x91c817a1...)> B all
Removed all breakpoints.
```

Please refer to https://truffleframework.com/docs/truffle/getting-started/debugging-your-contracts#debugging-interface for the details of all debugger commands.

10.3.3 How to Debug a Transaction

To use debugger, we need to collect transaction-related information, such as transaction hash. Then input following command

```
$ truffle debug <transaction hash>
```

We use transaction 0x8e5dadfb921dd... as an example.

```
$ truffle debug
0x8e5dadfb921ddddfa8f53af1f9bd8beeac6838d52d7e0c2fe5085b42a
4f3ca76
```

The command above will launch the debugger interface.

10.3.4 Debug a Foodcart Contract

We have talked a lot about debugger. In the following, we will use a real example of FoodCart to explain how to debug.[1] The following contract implements a simple food cart; anyone can select food sold online and add them to food cart. And at the same time, anyone can clear the food cart if ether payments succeed. Before we start, please check and confirm the environment first:

1. Truffle 4.0 or up
2. Solidity compiler 0.4.24 or up
3. Private blockchain (Ganache CLI v6.1.6 or above)

10.3.4.1 Step 1: Create Contract

First, we create project root directory by typing the following command:

```
mkdir FoodCart
```

Then, we enter the project directory and create a truffle project.

```
cd FoodCart
truffle init
```

Enter into contract directory and create a contract file named FoodCart.sol.

```
cd contracts
touch FoodCart.sol
```

Open any editor and type source code. Source code can be found here.[2] We briefly introduce FoodCart contract:

- State Variable
  State variables owner and skuCount are used to store owner and count of food items; mapping foodItems maps sku to food items.
- Enum Variable

[1]https://medium.com/ethereum-developers/the-ultimate-end-to-end-tutorial-to-create-and-deploy-a-fully-descentralized-dapp-in-ethereum-18f0cf6d7e0e

[2]https://gist.githubusercontent.com/mayorcoded/1fd0b4361e273f18e6d56e30b24cdb36/raw/cabb4283fee5d73719f284fd10780241b95c4aec/FoodCart.js

State enumeration variable is an user-defined data type which keeps the status of food items in cart. Enum type can be converted to integer explicitly. For example: ForSale = 0, Sold = 1.

- Event

  Event ForSale and Sold keep information on selling or sold food items. In javascript, these events can be called by callback function which improves the interactiveibility of Dapp.
- Struct Variable

  Struct variable FoodItem is an user-defined type, containing all info related to food. All its attributes can be accessed through struct name +.
- Modifier

  Modifier doesFoodItemExist, isFoodItemForSale, and hasBuyerPaidEnough are some function which will be executed before the function call.
- Function

  Function addFoodItem, buyFoodItem, and fetchFoodItem are to add food items to contract, buy food items and retrieve food-related information. Constructor initializes the owner state variable to the deployed contract address. Anonymous payable function is able to accept ether.

10.3.4.2 Step 2: Deploy Contract

Next, we deploy the contract to our private blockchain. Under the migrations directory, we create 2_foodcart_migration.js. Open this file in any editor and input the following code:

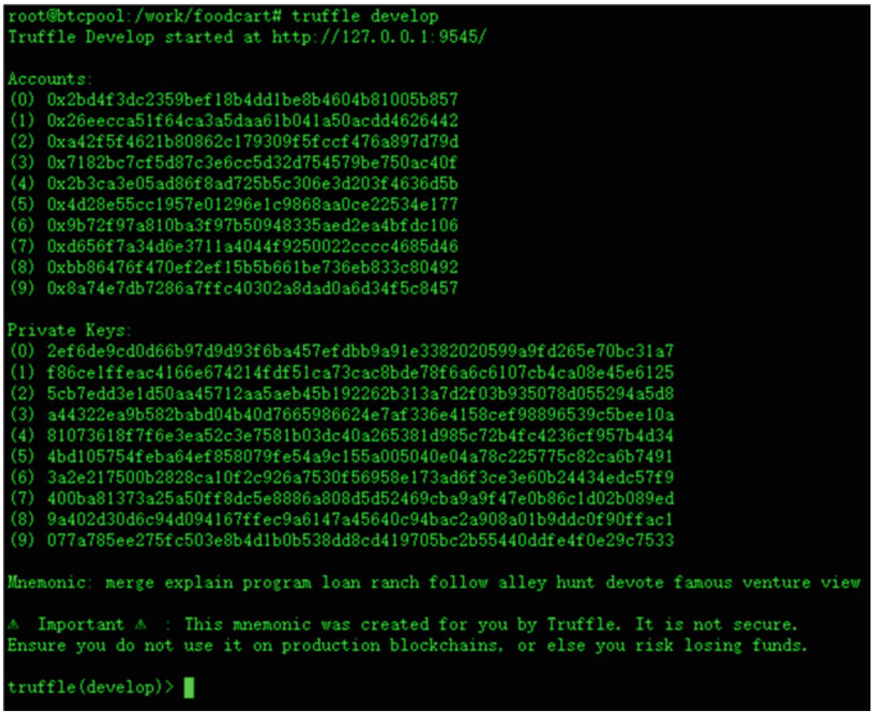
```
var FoodCart = artifacts.require("./FoodCart.sol");

module.exports = function(deployer) {
 deployer.deploy(FoodCart);
};
```

Resource: https://github.com/mayorcoded/FoodCartTutorial

Code snippet above makes truffle framework deploy FoodCart.sol contract to local private chain. Open command terminal and enter project directory, type the following command:

```
truffle develop
```

Command above starts to debug our contract and the output is like below (Fig. 10.1):



```
root@btcpool:/work/foodcart# truffle develop
Truffle Develop started at http://127.0.0.1:9545/

Accounts:
(0) 0x2bd4f3dc2359bef18b4dd1be8b4604b81005b857
(1) 0x26eecca51f64ca3a5daa61b041a50acdd4626442
(2) 0xa42f5f4621b80862c179309f5fccf476a897d79d
(3) 0x7182bc7cf5d87c3e6cc5d32d754579be750ac40f
(4) 0x2b3ca3e05ad86f8ad725b5c306e3d203f4636d5b
(5) 0x4d28e55cc1957e01296e1c9868aa0ce22534e177
(6) 0x9b72f97a810ba3f97b50948335aed2ea4bfdc106
(7) 0xd656f7a34d6e3711a4044f9250022cccc4685d46
(8) 0xbb86476f470ef2ef15b5b661be736eb833c80492
(9) 0x8a74e7db7286a7ffc40302a8dad0a6d34f5c8457

Private Keys:
(0) 2ef6de9cd0d66b97d9d93f6ba457efdbb9a91e3382020599a9fd265e70bc31a7
(1) f86ce1ffeac4166e674214fdf51ca73cac8bde78f6a6c6107cb4ca08e45e6125
(2) 5cb7edd3e1d50aa45712aa5aeb45b192262b313a7d2f03b935078d055294a5d8
(3) a44322ea9b582babd04b40d7665986624e7af336e4158cef98896539c5bee10a
(4) 81073618f7f6e3ea52c3e7581b03dc40a265381d985c72b4fc4236cf957b4d34
(5) 4bd105754feba64ef858079fe54a9c155a005040e04a78c225775c82ca6b7491
(6) 3a2e217500b2828ca10f2c926a7530f56958e173ad6f3ce3e60b24434edc57f9
(7) 400ba81373a25a50ff8dc5e8886a808d5d52469cba9a9f47e0b86c1d02b089ed
(8) 9a402d30d6c94d094167ffec9a6147a45640c94bac2a908a01b9ddc0f90ffac1
(9) 077a785ee275fc503e8b4d1b0b538dd8cd419705bc2b55440ddfe4f0e29c7533

Mnemonic: merge explain program loan ranch follow alley hunt devote famous venture view

⚠ Important ⚠ : This mnemonic was created for you by Truffle. It is not secure.
Ensure you do not use it on production blockchains, or else you risk losing funds.

truffle(develop)> █
```

Fig. 10.1 Truffle develop Console

The screen copy shows that blockchain is running and is listening on port 9545; at the same time, testing accounts are created and the command prompt becomes truffle (develop)>.

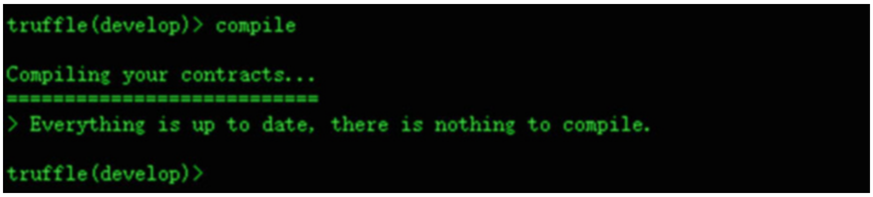
Under truffle(develop) > prompt, type compile command to compile contract. And compiled objects are stored under build subdirectory:

```
truffle(develop)> compile
```

Type “compile” command will generate the following output (Fig. 10.2):

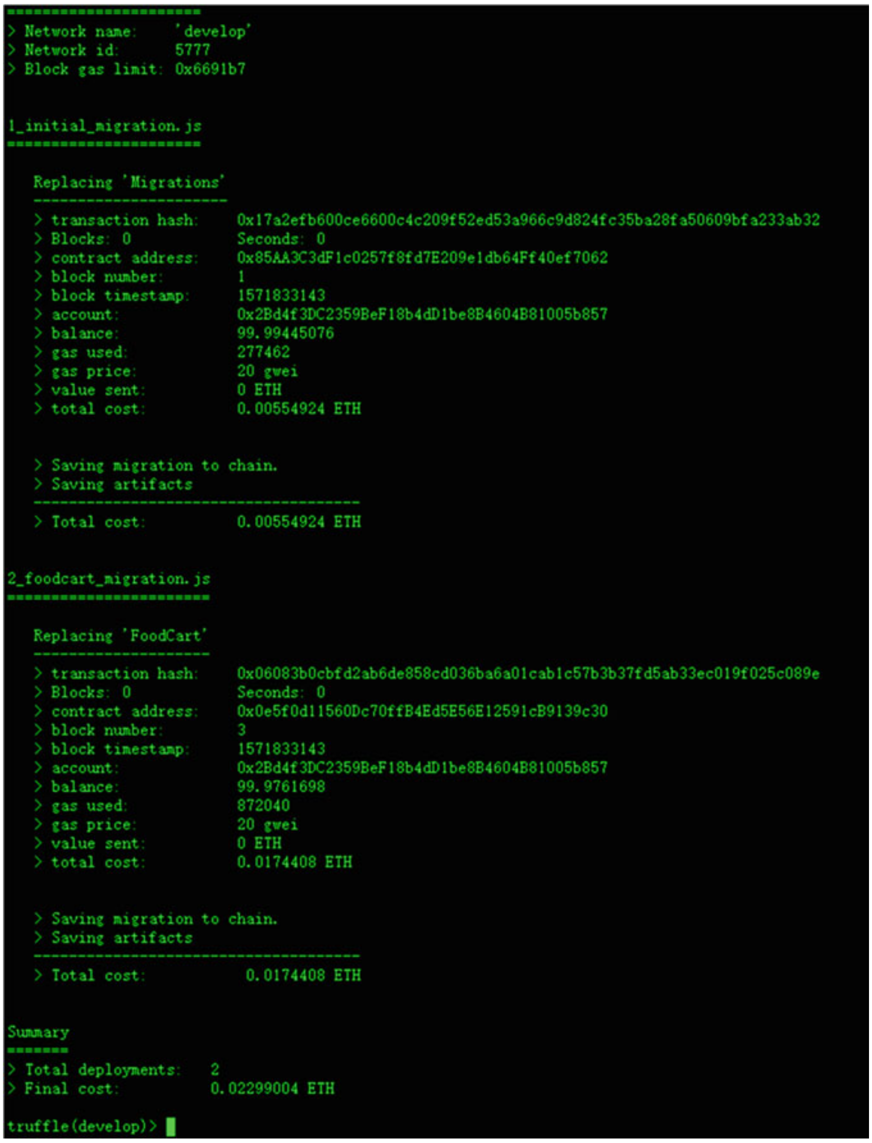
And the last step, we migrate compiled contract to blockchain. Type “migrate” command and output should be looked like (Fig. 10.3):

```
truffle(develop)> migrate
```



```
truffle(develop)> compile

Compiling your contracts...
===========================
> Everything is up to date, there is nothing to compile.

truffle(develop)>
```

Fig. 10.2 Truffle compile output



```
======================
> Network name:    'develop'
> Network id:      5777
> Block gas limit: 0x6691b7


1_initial_migration.js
======================

   Replacing 'Migrations'
   ----------------------
   > transaction hash:    0x17a2efb600ce6600c4c209f52ed53a966c9d824fc35ba28fa50609bfa233ab32
   > Blocks: 0            Seconds: 0
   > contract address:    0x85AA3C3dF1c0257f8fd7E209e1db64Ff40ef7062
   > block number:        1
   > block timestamp:     1571833143
   > account:             0x2Bd4f3DC2359BeF18b4dD1be8B4604B81005b857
   > balance:             99.99445076
   > gas used:            277462
   > gas price:           20 gwei
   > value sent:          0 ETH
   > total cost:          0.00554924 ETH


   > Saving migration to chain.
   > Saving artifacts
   -------------------------------------
   > Total cost:          0.00554924 ETH


2_foodcart_migration.js
=======================

   Replacing 'FoodCart'
   --------------------
   > transaction hash:    0x06083b0cbfd2ab6de858cd036ba6a01cab1c57b3b37fd5ab33ec019f025c089e
   > Blocks: 0            Seconds: 0
   > contract address:    0x0e5f0d11560Dc70ffB4Ed5E56E12591cB9139c30
   > block number:        3
   > block timestamp:     1571833143
   > account:             0x2Bd4f3DC2359BeF18b4dD1be8B4604B81005b857
   > balance:             99.9761698
   > gas used:            872040
   > gas price:           20 gwei
   > value sent:          0 ETH
   > total cost:          0.0174408 ETH


   > Saving migration to chain.
   > Saving artifacts
   -------------------------------------
   > Total cost:           0.0174408 ETH


Summary
=======
> Total deployments:   2
> Final cost:          0.02299004 ETH

truffle(develop)>
```

Fig. 10.3 Truffle migrate output

10.3.4.3 Step 3: Interact with Contract

Let us interact with the contract: add some fooditem to foodcart, check details of food item and pay ether to checkout the foodcart. In truffle(develop) > command prompt, create a foodCart variable and store the deployed contract instance to foodCart.

```
truffle(develop) > let foodCart;
truffle(develop) > FoodCart.deployed().then((instance) => {
foodCart = instance; });
```

We use web3.deployed method to deploy contract and it returns a promise. In Promise, we save contract instance to foodCart variable.

Next, we try to add some food to cart. First, we add a function which is responsible for adding fooditems to foodcart and print result on the console. addFoodItemToCart function accepts food name and price as parameters and add correspondent food into cart. This function is a wrapper of addFoodItem function in contract. Besides calling the contract function, this wrapper also formats output to log. We use the following commands to call addFoodItemToCart function:

```
truffle(develop) > addFoodItemToCart('Fried Rice', 10);
truffle(develop) > { name: 'Fried Rice',sku: 0, price: 10, state:
'ForSale', foodItemExist: true } //output

truffle(develop) > addFoodItemToCart('Chicken Pepper Soup', 10);
truffle(develop) > { name: 'Chicken Pepper Soup', sku: 1, price:
10, state: 'ForSale', foodItemExist: true } //output

truffle(develop) > addFoodItemToCart('Pepperoni Pizza', 50);
truffle(develop) > { name: 'Pepperoni Pizza', sku: 2, price:
50, state: 'ForSale', foodItemExist: true } //output
```

Now we have three food items in cart. Let us clear the cart. Before we pay, we need to know payment address from which we will get ether. When we start the local blockchain, truffle already creates ten testing accounts automatically and each account has 100 ether. We create a variable to save these address/account for subsequent use.

```
truffle(develop) > const buyerAddress = web3.eth.accounts[1];
truffle(develop) > buyerAddress
'0x26EeCca51f64cA3a5Daa61b041a50ACdD4626442'
```

buyFoodItemFromCart function accepts itemSku and food count as parameters, allowing us to buy some food. Ether for payment is from the account saved in buyerAddress variable. From our previous transaction, we know the sku of Fried Rice is 0 and its price is 10 wei. So, under truffle(develop) > command prompt, we use these values to call buyFoodItemFromCart:

```
let buyFoodItemFromCart = (itemSku, amount) => {foodCart.
buyFoodItem(itemSku,
{from: buyerAddress, value: amount}).then((trxn) => { const
details = trxn.logs[0].args;
details.sku = details.sku.toNumber();
details.price = details.price.toNumber();
details.state = trxn.logs[0].event;
console.log(details);
});
}
```

From the output, the status of Fried Rice is sol, and foodItemExist is false, which indicates that the food is not on sale any more.

10.3.4.4 Step 4: Debug Contract

Now, we already learn the mechanism about how the contract works. Next, we will introduce some error and then use debugger to debug errors and fix them. In order to debug a transaction, we need to know transaction hash. Under truffle(develop)>, type command debug [transaction hash], and use debug command until we found the root cause of errors.

We will try the following invalid transactions:

- Try to buy a food item not in the cart
- Try to buy a food item with the price lower than the sale price

To start debugging, we need to open a new terminal and run truffle develop logger: go to FoodCart project root directory and run truffle develop --log.

```
FoodCart $ truffle develop --log
```

We should see the following output:

```
Connected to existing Truffle Develop session at http://
127.0.0.1:9545/
```

Logger connects to an existing port, monitoring transaction events and record transactions to logs including transaction hash, block ID, etc.

10.3.4.4.1 Illegal Transaction 1: Try to Buy a Food Item that Does Not Exist

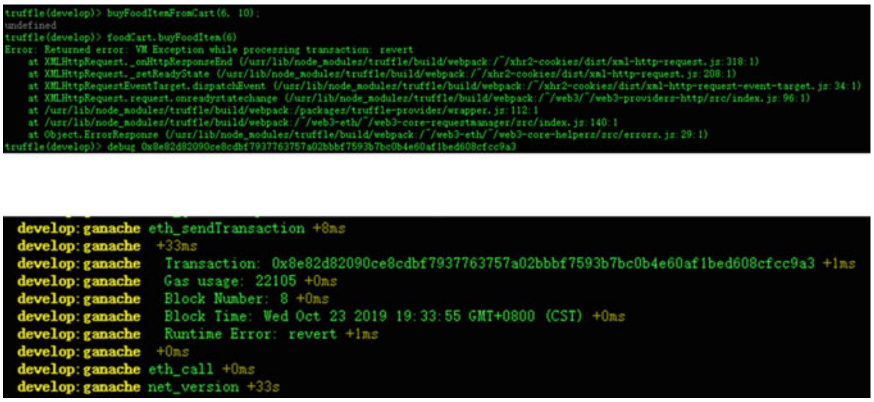
We already add some food items to cart and their skus are 0, 1, 2. We try to buy a non-existing item whose sku is 6. The try will generate information: UnhandledPromiseRejectionWarning: Error: VM Exception while processing transaction: revert . . ., and it does not tell us much clue for the root cause (Fig. 10.4).

In log, the most important content is transaction hash: 0x8e82d82090ce8cdbf7937763757a02bbbf7593b7bc0b4e60af1bed608cfcc9a3. Using this transaction hash, we can debug transaction and identify the root cause. So we copy transaction hash and under truffle(develop)>, we type "debug your-trasanction-hash" and Output is like below (Fig. 10.5):

From the screen capture above, we can find that debug command is run properly: debugger compile transaction, collect transaction data, get address influenced, and deployed contract address, and list a series of available commands. The most frequently used command is "step next" which track all opcodes of transaction, step by step. Typing enter or n will execute "step next."

Under debug(develop:0x8e82d820...) > prompt, keep hit enter key to follow the code flow to the place where transaction fails. In our example, we need to run eight steps (which means hit enter key nine times) to reach the place where problem occur. Let us have a look at the output after four steps (Fig. 10.6):

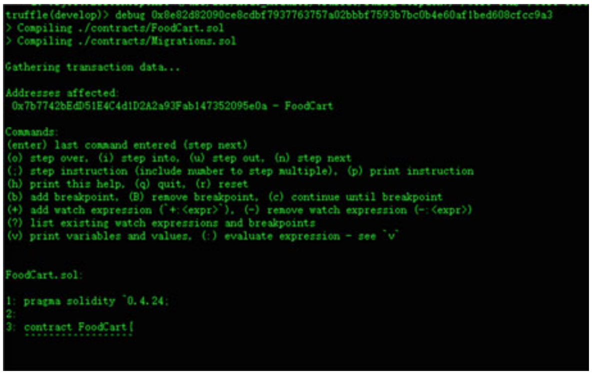
From the screen copy above, we find that buyFoodItem function is called on line 64. But before the execution of the function, the modifier doesFoodItemExist on line 64 is called to check if there are food items in food cart. On line 33, we see that require function on line 33 was used to ensure that the foodItemExist attribute of food with specified sku is true. Then, let us check what happens next (Fig. 10.7):



```
truffle(develop)> buyFoodItemFromCart(6, 10);
undefined
truffle(develop)> foodCart.buyFoodItem(6)
Error: Returned error: VM Exception while processing transaction: revert
    at XMLHttpRequest._onHttpResponseEnd (/usr/lib/node_modules/truffle/build/webpack:/~/xhr2-cookies/dist/xml-http-request.js:318:1)
    at XMLHttpRequest._setReadyState (/usr/lib/node_modules/truffle/build/webpack:/~/xhr2-cookies/dist/xml-http-request.js:208:1)
    at XMLHttpRequestEventTarget.dispatchEvent (/usr/lib/node_modules/truffle/build/webpack:/~/xhr2-cookies/dist/xml-http-request-event-target.js:34:1)
    at XMLHttpRequest.request.onreadystatechange (/usr/lib/node_modules/truffle/build/webpack:/~/web3/~/web3-providers-http/src/index.js:96:1)
    at /usr/lib/node_modules/truffle/build/webpack:/packages/truffle-provider/wrapper.js:112:1
    at /usr/lib/node_modules/truffle/build/webpack:/~/web3-eth/~/web3-core-requestmanager/src/index.js:140:1
    at Object.ErrorResponse (/usr/lib/node_modules/truffle/build/webpack:/~/web3-eth/~/web3-core-helpers/src/errors.js:29:1)
truffle(develop)> debug 0x8e82d82090ce8cdbf7937763757a02bbbf7593b7bc0b4e60af1bed608cfcc9a3
```

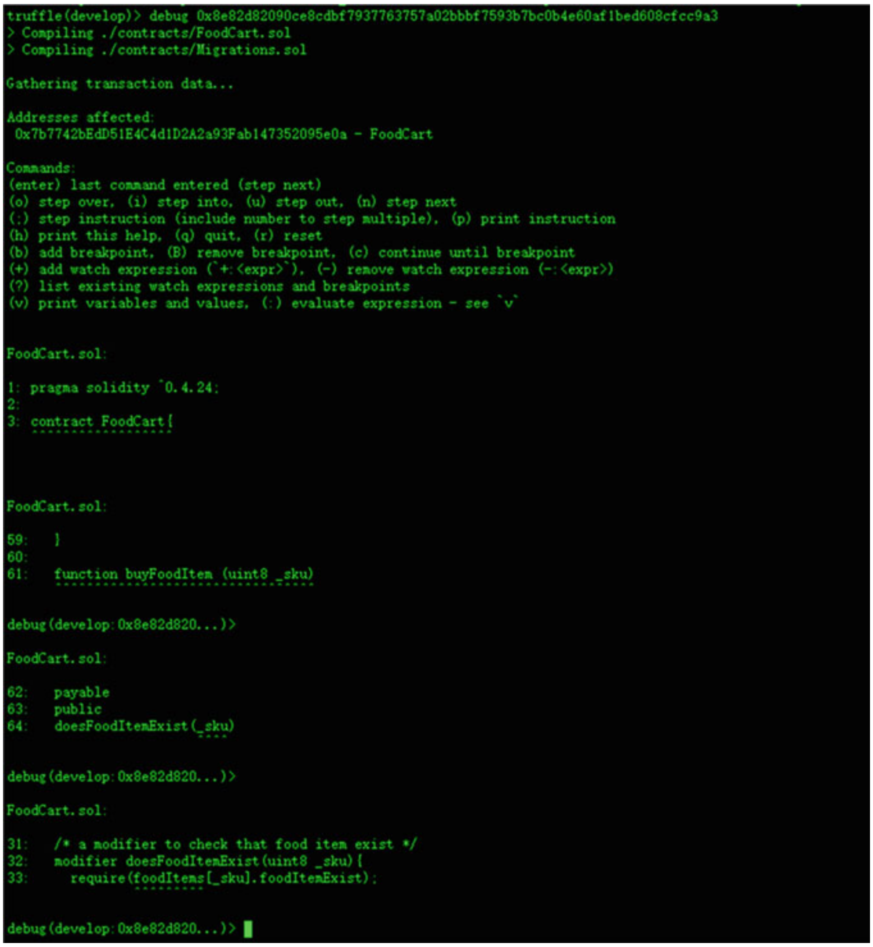
```
develop:ganache eth_sendTransaction +8ms
develop:ganache +33ms
develop:ganache   Transaction: 0x8e82d82090ce8cdbf7937763757a02bbbf7593b7bc0b4e60af1bed608cfcc9a3 +1ms
develop:ganache   Gas usage: 22105 +0ms
develop:ganache   Block Number: 8 +0ms
develop:ganache   Block Time: Wed Oct 23 2019 19:33:55 GMT+0800 (CST) +0ms
develop:ganache   Runtime Error: revert +1ms
develop:ganache +0ms
develop:ganache eth_call +0ms
develop:ganache net_version +33s
```

Fig. 10.4 Invalid transaction output



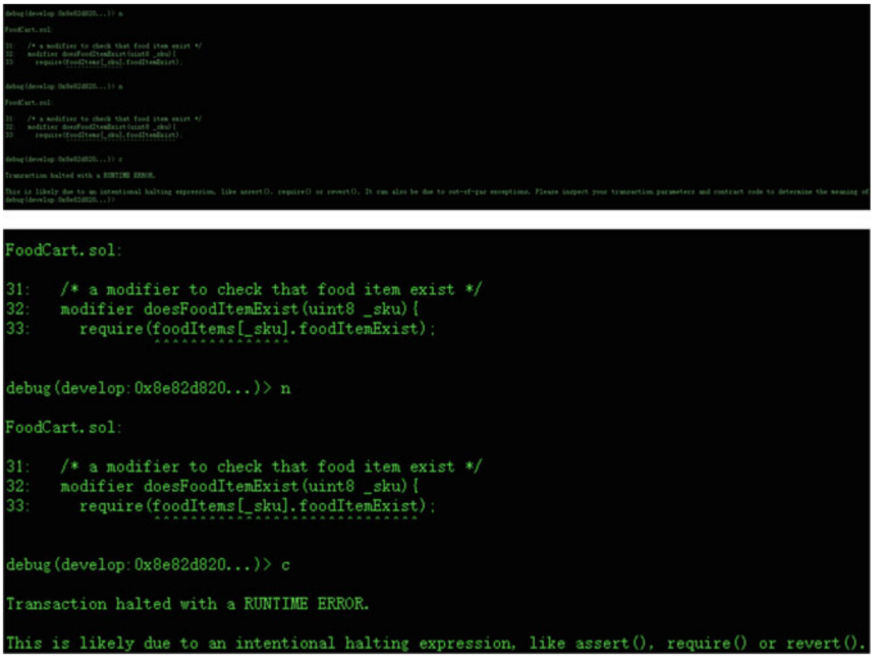
```
truffle(develop)> debug 0x8e82d82090ce8cdbf7937763757a02bbbf7593b7bc0b4e60af1bed608cfcc9a3
> Compiling ./contracts/FoodCart.sol
> Compiling ./contracts/Migrations.sol

Gathering transaction data...

Addresses affected:
 0x7b7742bEdD51E4C4d1D2A2a93Fab147352095e0a - FoodCart

Commands:
(enter) last command entered (step next)
(o) step over, (i) step into, (u) step out, (n) step next
(;) step instruction (include number to step multiple), (p) print instruction
(h) print this help, (q) quit, (r) reset
(b) add breakpoint, (B) remove breakpoint, (c) continue until breakpoint
(+) add watch expression (`+:<expr>`), (-) remove watch expression (-:<expr>)
(?) list existing watch expressions and breakpoints
(v) print variables and values, (:) evaluate expression - see `v`


FoodCart.sol:

1: pragma solidity ^0.4.24;
2:
3: contract FoodCart{
   ^^^^^^^^^^^^^^^^^
```

Fig. 10.5 Debug Transaction on Turffle Console



```
truffle(develop)> debug 0x8e82d82090ce8cdbf7937763757a02bbbf7593b7bc0b4e60af1bed608cfcc9a3
> Compiling ./contracts/FoodCart.sol
> Compiling ./contracts/Migrations.sol

Gathering transaction data...

Addresses affected:
 0x7b7742bEdD51E4C4d1D2A2a93Fab147352095e0a - FoodCart

Commands:
(enter) last command entered (step next)
(o) step over, (i) step into, (u) step out, (n) step next
(;) step instruction (include number to step multiple), (p) print instruction
(h) print this help, (q) quit, (r) reset
(b) add breakpoint, (B) remove breakpoint, (c) continue until breakpoint
(+) add watch expression (`+:<expr>`), (-) remove watch expression (-:<expr>)
(?) list existing watch expressions and breakpoints
(v) print variables and values, (:) evaluate expression - see `v`


FoodCart.sol:

1: pragma solidity ^0.4.24;
2:
3: contract FoodCart{
   ^^^^^^^^^^^^^^^^^



FoodCart.sol:

59:    }
60:
61:    function buyFoodItem (uint8 _sku)
       ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^


debug(develop:0x8e82d820...)>

FoodCart.sol:

62:    payable
63:    public
64:    doesFoodItemExist(_sku)
                         ^^^^


debug(develop:0x8e82d820...)>

FoodCart.sol:

31:    /* a modifier to check that food item exist */
32:    modifier doesFoodItemExist(uint8 _sku){
33:      require(foodItems[_sku].foodItemExist);
                 ^^^^^^^^^

debug(develop:0x8e82d820...)> █
```

Fig. 10.6 Execute four steps of Foodcart.sol on Truffle Debug Console



```
debug(develop:0x8e82d820...)> n

FoodCart.sol:

31:   /* a modifier to check that food item exist */
32:   modifier doesFoodItemExist(uint8 _sku){
33:     require(foodItems[_sku].foodItemExist);
                ^^^^^^^^^^^^^^^

debug(develop:0x8e82d820...)> n

FoodCart.sol:

31:   /* a modifier to check that food item exist */
32:   modifier doesFoodItemExist(uint8 _sku){
33:     require(foodItems[_sku].foodItemExist);
                ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

debug(develop:0x8e82d820...)> c

Transaction halted with a RUNTIME ERROR.

This is likely due to an intentional halting expression, like assert(), require() or revert(). It can also be due to out-of-gas exceptions. Please inspect your transaction parameters and contract code to determine the meaning of
debug(develop:0x8e82d820...)>
```

```
FoodCart.sol:

31:   /* a modifier to check that food item exist */
32:   modifier doesFoodItemExist(uint8 _sku){
33:     require(foodItems[_sku].foodItemExist);
                ^^^^^^^^^^^^^^^

debug(develop:0x8e82d820...)> n

FoodCart.sol:

31:   /* a modifier to check that food item exist */
32:   modifier doesFoodItemExist(uint8 _sku){
33:     require(foodItems[_sku].foodItemExist);
                ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

debug(develop:0x8e82d820...)> c

Transaction halted with a RUNTIME ERROR.

This is likely due to an intentional halting expression, like assert(), require() or revert().
```

Fig. 10.7 FoodCart.sol output after require on Truffle Debug Console

From the screen copy above, we find that transaction terminates on line 33 because require function fails. The food item we want to buy does not exist in food cart. Require function throw a state-reverting exception. And this exception rolls back all state change and return an error to caller. We can use Truffle Debugger to debug this contract.

10.3.4.4.2 Illegal Transaction 2: Try to Pay an Amount Less Than the Price of a Food Item

In this transaction, we try to buy food with a price lower than the sale price. In "step 3: Interacting with the smart contract," we add three food items. Let us try to buy "Chicken Pepper Soup" (sku = 1) and the price is 10 wei. Under truffle (develop) > command prompt, invoking buyFoodItem function to buy food with the price lower than the sale price.

```
truffle(develop)> buyFoodItemFromCart(1, 5);
truffle(develop)> (node:40269)
UnhandledPromiseRejectionWarning: Error: VM Exception while
processing transaction: revert
```

It can be expected that the transaction will fail with not many details about the error. In order to debug the transaction, we go to log console and copy transaction hash. On my local machine, transaction hash is 0x495ac8917b6b0a6fd5ac9965d74d116cd4072157fe466429a08ea6fdadf401e5. We use this transaction hash to start debugger.

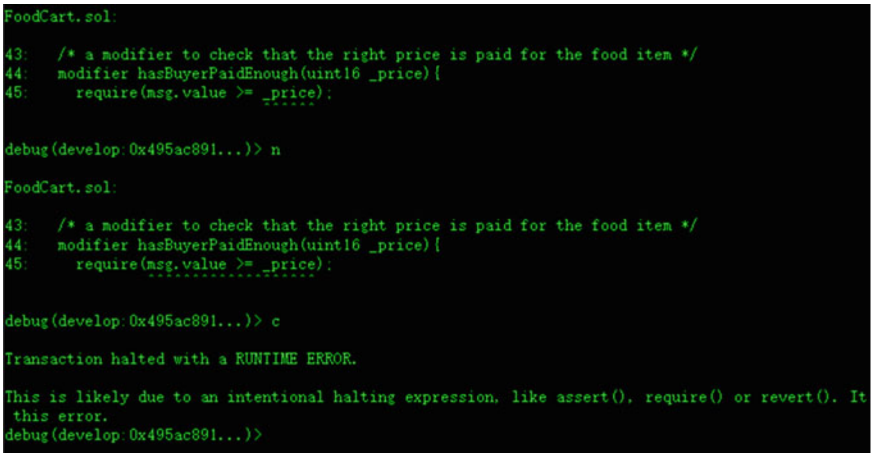
```
truffle(develop)> debug
0x495ac8917b6b0a6fd5ac9965d74d116cd4072157fe466429a08ea6fdad
f401e5
```

We first use step next command to track the transaction. By this way, it shows all opcodes executed to exception. Using enter key to execute opcode continuously as below (Fig. 10.8):

From above, we could see that on line 44, modifier hasBuyerPaidEnough is run before buyFoodItemFromCart function. This modifier requires the buyer's bidding price is more than the sale price of the food item. And our bidding price does not meet this condition. So we need to send enough ether to buy food.

10.4 Remix Debug

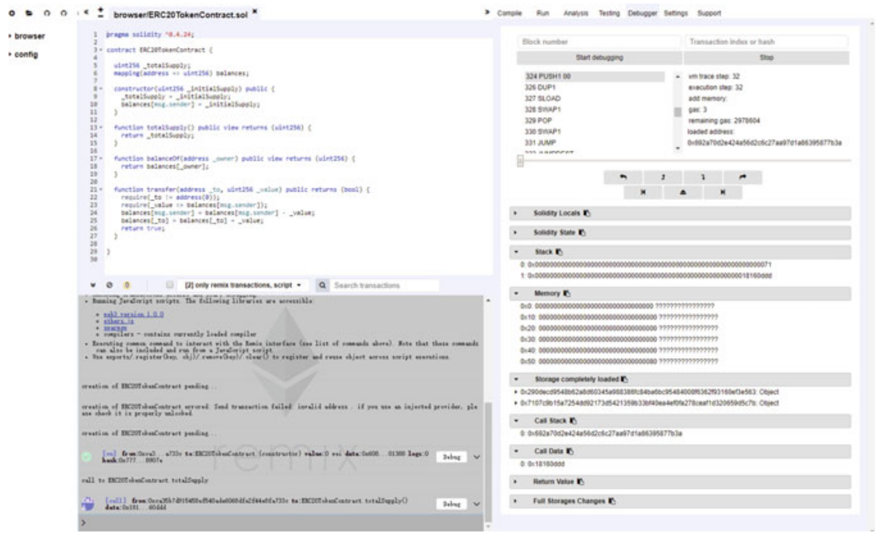
Remix official URL is at: https://remix.ethereum.org/. Here is the main panel of remix[3].



```
FoodCart.sol:

43:   /* a modifier to check that the right price is paid for the food item */
44:   modifier hasBuyerPaidEnough(uint16 _price) {
45:     require(msg.value >= _price);
                                  ------

debug(develop:0x495ac891...)> n

FoodCart.sol:

43:   /* a modifier to check that the right price is paid for the food item */
44:   modifier hasBuyerPaidEnough(uint16 _price) {
45:     require(msg.value >= _price);
                ---------------------

debug(develop:0x495ac891...)> c

Transaction halted with a RUNTIME ERROR.

This is likely due to an intentional halting expression, like assert(), require() or revert(). It
 this error.
debug(develop:0x495ac891...)>
```

Fig. 10.8 Debug transaction on Truffle debugger conbsole

[3]https://zhuanlan.zhihu.com/p/38102679



```
pragma solidity ^0.4.24;

contract ERC20TokenContract {

  uint256 _totalSupply;
  mapping(address => uint256) balances;

  constructor(uint256 _initialSupply) public {
    _totalSupply = _initialSupply;
    balances[msg.sender] = _initialSupply;
  }

  function totalSupply() public view returns (uint256) {
    return _totalSupply;
  }

  function balanceOf(address _owner) public view returns (uint256) {
    return balances[_owner];
  }

  function transfer(address _to, uint256 _value) public returns (bool) {
    require(_to != address(0));
    require(_value <= balances[msg.sender]);
    balances[msg.sender] = balances[msg.sender] - _value;
    balances[_to] = balances[_to] + _value;
    return true;
  }

}
```

```
- Running JavaScript scripts. The following libraries are accessible:
  - web3 version 1.0.0
  - ethers.js
  - swarmgw
  - compilers - contains currently loaded compiler
- Executing common command to interact with the Remix interface (see list of commands above). Note that these commands
  can also be included and run from a JavaScript script.
- Use exports/.register(key, obj)/.remove(key)/.clear() to register and reuse object across script executions.

creation of ERC20TokenContract pending...

creation of ERC20TokenContract errored: Send transaction failed: invalid address . if you use an injected provider, ple
ase check it is properly unlocked.

creation of ERC20TokenContract pending...

[vm] from:0xca3...a733c to:ERC20TokenContract.(constructor) value:0 wei data:0x608...01300 logs:0
hash:0x777...8807e

call to ERC20TokenContract.totalSupply

[call] from:0xca35b7d915458ef540ade6068dfe2f44e8fa733c to:ERC20TokenContract.totalSupply()
data:0x181...60ddd
```

```
324 PUSH1 00
326 DUP1
327 SLOAD
328 SWAP1
329 POP
330 SWAP1
331 JUMP

vm trace step: 32
execution step: 32
add memory:
gas: 3
remaining gas: 2978604
loaded address:
0x692a70d2e424a56d2c6c27aa97d1a86395877b3a

Stack
0: 0x0000000000000000000000000000000000000000000000000000000000000071
1: 0x0000000000000000000000000000000000000000000000000000000018160ddd

Memory
0x0: 00000000000000000000000000000000 ????????????????
0x10: 00000000000000000000000000000000 ????????????????
0x20: 00000000000000000000000000000000 ????????????????
0x30: 00000000000000000000000000000000 ????????????????
0x40: 00000000000000000000000000000000 ????????????????
0x50: 00000000000000000000000000000080 ????????????????

Storage completely loaded
0x290decd9548b62a8d60345a988386fc84ba6bc95484008f6362f93160ef3e563: Object
0x7107c9b15a7254dd92173d5421359b33bf40ea4ef0fa278ceaf1d320659d5c7b: Object

Call Stack
0: 0x692a70d2e424a56d2c6c27aa97d1a86395877b3a

Call Data
0: 0x18160ddd
```

Fig. 10.9 Remix: main interface

Using Remix to code and test the smart contract is easy and simple. Remix is an online editor maintained by Ethereum foundation (Fig. 10.9).

Using Remix makes it easy to develop and debug Solidity program. And remix also can auto-compile and audit code real-timely. If everything is ready, we can choose "Injected Web3" (through Metamask) to deploy our contract to a real network. And the most important is: remix can create a virtual network in the browser to test our contract quickly. If we select "JavaScript VM" under run tab, remix will create five external accounts for use and each account is pre-allocated 100 ethers (Fig. 10.10).

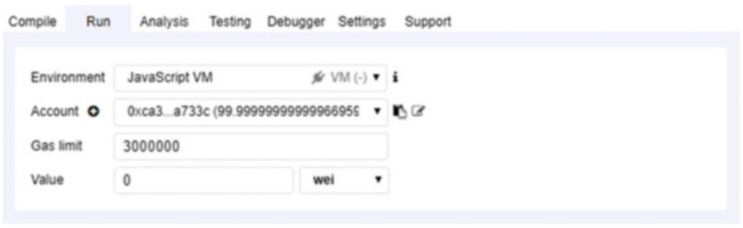
We provide necessary parameters to create contract and then hit "Deploy" button (Fig. 10.11).

After the deployment, an instance of our contract is created in our local network. Remix will create *input fields* for each function in our contract automatically (Fig. 10.12).

Remix provides a console at the bottom, showing all the interaction with the contract (Fig. 10.13).

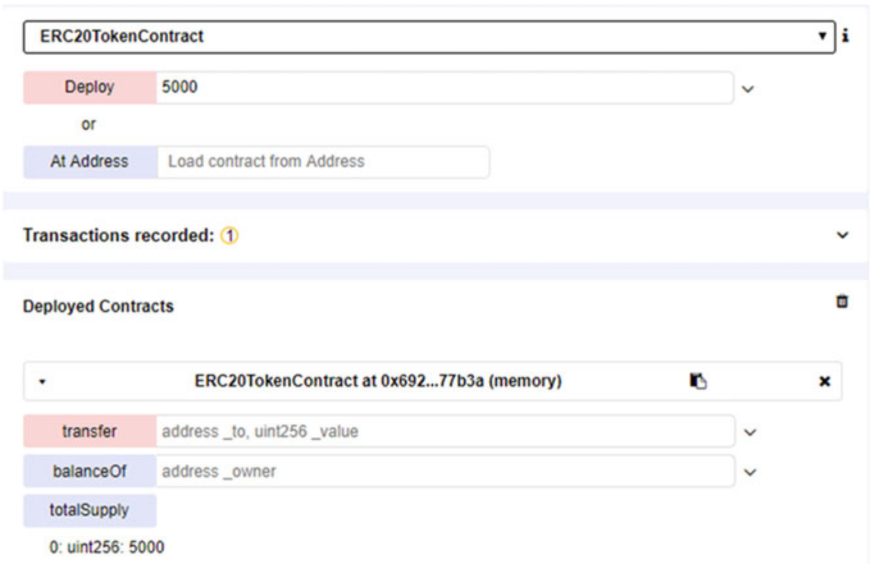
For each interaction with blockchain, there is a "Debug" button on the right. Clicking it will jump to "Debugger" tab as below (Fig. 10.14):

The interface is straightforward: we can Step in, Step over, Step Out, Step back, continue, etc. It is easy to learn by doing. And we can check the situation of Memory, Stack, Storage, Calldata while executing each step, which is really helpful in learning the logic of the program.



Environment JavaScript VM VM (-)
Account 0xca3...a733c (99.999999999999669599
Gas limit 3000000
Value 0 wei

Fig. 10.10 Remix: run tab

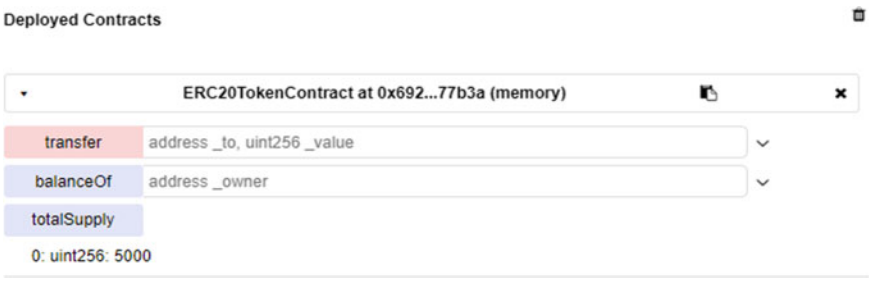


Deploy 5000

ERC20TokenContract at 0x692...77b3a (memory)

transfer address _to, uint256 _value
balanceOf address _owner
totalSupply
0: uint256: 5000

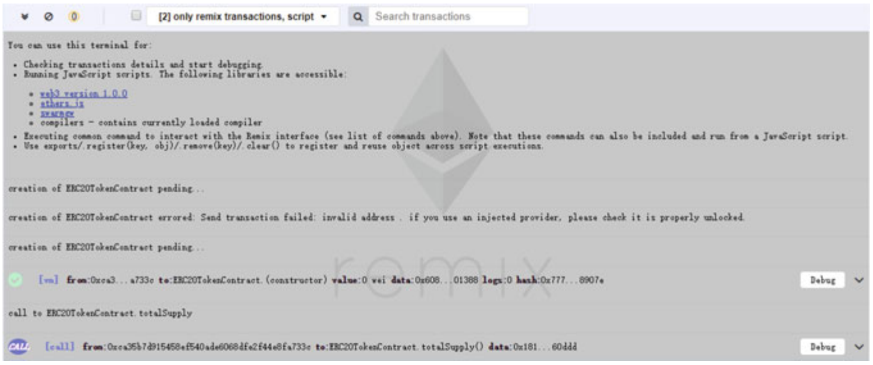
Fig. 10.11 Remix: contract deployment



ERC20TokenContract at 0x692...77b3a (memory)

transfer address _to, uint256 _value
balanceOf address _owner
totalSupply
0: uint256: 5000

Fig. 10.12 Remix: deployed contract interface



```
You can use this terminal for:

• Checking transactions details and start debugging.
• Running JavaScript scripts. The following libraries are accessible:

    • web3 version 1.0.0
    • ethers.js
    • swarmgw
    • compilers - contains currently loaded compiler

• Executing common command to interact with the Remix interface (see list of commands above). Note that these commands can also be included and run from a JavaScript script.
• Use exports/.register(key, obj)/.remove(key)/.clear() to register and reuse object across script executions.

creation of ERC20TokenContract pending...

creation of ERC20TokenContract errored: Send transaction failed: invalid address . if you use an injected provider, please check it is properly unlocked.

creation of ERC20TokenContract pending...

[vm] from:0xca3...a733c to:ERC20TokenContract.(constructor) value:0 wei data:0x608...01388 logs:0 hash:0x777...8907e

call to ERC20TokenContract.totalSupply

[call] from:0xca35b7d915458ef540ade6068dfe2f44e8fa733c to:ERC20TokenContract.totalSupply() data:0x181...60ddd
```

Fig. 10.13 Remix: console output

10.5 Other Debug Tools

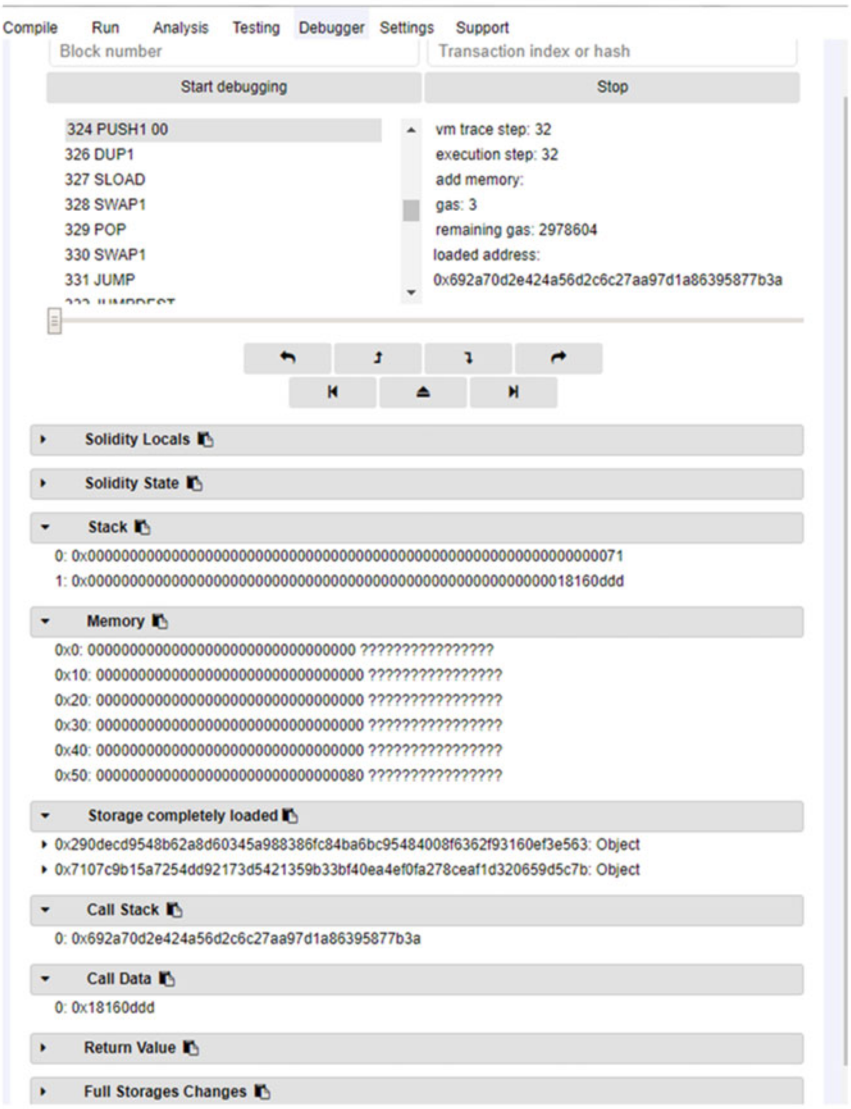
10.5.1 JEB

You can download the demo version from https://www.pnfsoftware.com/jeb/demoevm. The installation package is relatively big, more than 160 M. JEB's main function is as below:

- Given EVM code specified, JEB de-compiler can de-compile it into pseudo-Solidity source code.
- Through EVM code analysis, JEB can determine the contract's public and private methods, which includes decompiled implementation of public method.
- In the situation with no ABI, code analysis can determine method name, event name, and their prototype.
- De-compiler will also try to recover some advanced feature:
  - Some popular interface such as ERC20, ERC721, multi-sign wallet, etc.;
  - Storage variable and type;
  - Function modifier
  - Function payable attribute
  - Trigger including event name
  - address.send() or address.transfer()
  - Pre-compiled contract call

10.5.2 Porosity

Porosity source code is https://github.com/comaeio/porosity. You need to download the source code and compile it by using Visual Studio (Fig. 10.15).



```
324 PUSH1 00
326 DUP1
327 SLOAD
328 SWAP1
329 POP
330 SWAP1
331 JUMP
```

```
vm trace step: 32
execution step: 32
add memory:
gas: 3
remaining gas: 2978604
loaded address:
0x692a70d2e424a56d2c6c27aa97d1a86395877b3a
```

Stack

```
0: 0x0000000000000000000000000000000000000000000000000000000000000071
1: 0x0000000000000000000000000000000000000000000000000000000018160ddd
```

Memory

```
0x0: 00000000000000000000000000000000 ????????????????
0x10: 00000000000000000000000000000000 ????????????????
0x20: 00000000000000000000000000000000 ????????????????
0x30: 00000000000000000000000000000000 ????????????????
0x40: 00000000000000000000000000000000 ????????????????
0x50: 00000000000000000000000000000080 ????????????????
```

Storage completely loaded

```
0x290decd9548b62a8d60345a988386fc84ba6bc95484008f6362f93160ef3e563: Object
0x7107c9b15a7254dd92173d5421359b33bf40ea4ef0fa278ceaf1d320659d5c7b: Object
```

Call Stack

```
0: 0x692a70d2e424a56d2c6c27aa97d1a86395877b3a
```

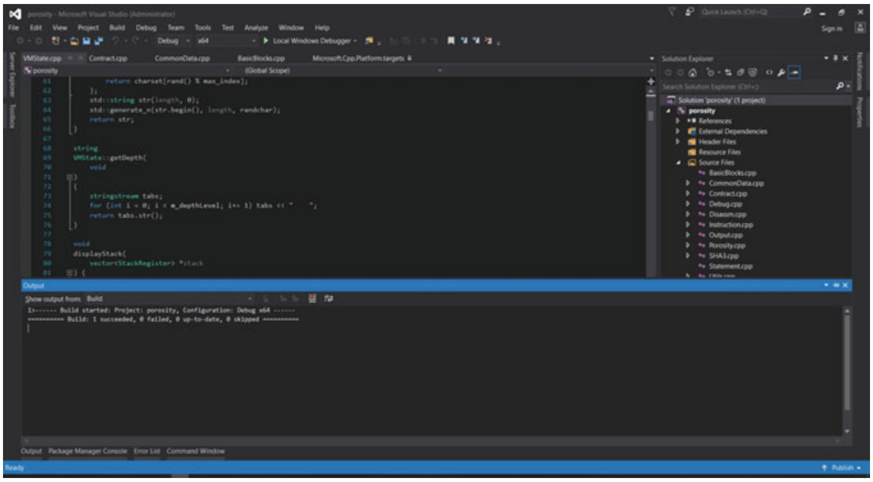
Call Data

```
0: 0x18160ddd
```

Fig. 10.14 Remix: debugger tab

```
    C:\dev\Coins\porosity-master\porosity\x64\Debug>porosity.exe
    parse: Please at least provide some byte code (--code) or run it
in debug mode
```

(continued)

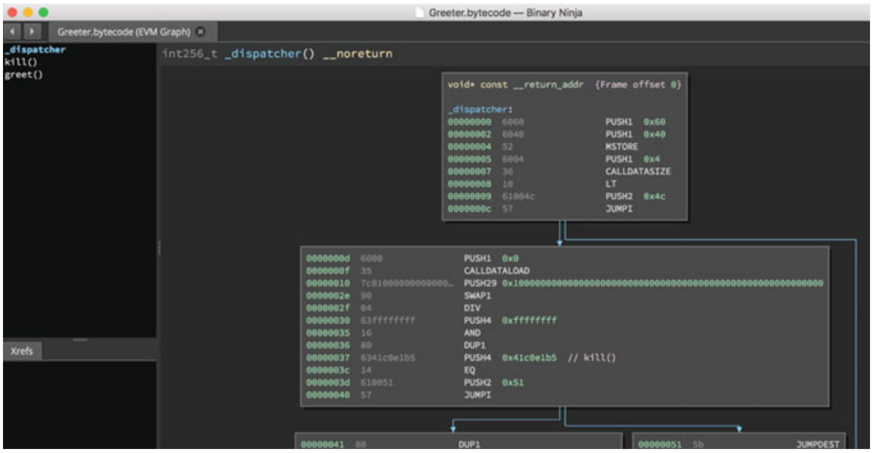


```
        return charset[rand() % max_index];
    };
    std::string str(length, 0);
    std::generate_n(str.begin(), length, randchar);
    return str;
}

string
VMState::getDepth(
    void
)
{
    stringstream tabs;
    for (int i = 0; i < m_depthLevel; i++ 1) tabs << "    ";
    return tabs.str();
}

void
displayStack(
    vector<StackRegister> *stack
) {
```

```
1>------ Build started: Project: porosity, Configuration: Debug x64 ------
========== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped ==========
```

Fig. 10.15 Porosity compilation result

```
(--debug) with pre-configured inputs.
    Porosity v0.1 (https://www.comae.io)
    Matt Suiche, Comae Technologies <support@comae.io>
    The Ethereum bytecode commandline decompiler.
    Decompiles the given Ethereum input bytecode and outputs the
Solidity code.
    Usage: porosity [options]
    Debug:
        --debug                    - Enable debug mode. (testing only - no
input parameter needed.)

    Input parameters:
        --code <bytecode>              - Ethereum bytecode. (mandatory)
        --code-file <filename>         - Read ethereum bytecode from file
        --arguments <arguments>           - Ethereum arguments to pass to
the function. (optional, default data set provided if not provided.)
        --abi <arguments>              - Ethereum Application Binary
Interface (ABI) in JSON format. (optional but recommended)
       --hash <hashmethod>           - Work on a specific function, can be
retrieved wit --list. (optional)

    Features:
        --list                     - List identified methods/functions.
        --disassm                  - Disassemble the bytecode.
        --single-step              - Execute the byte code through our VM.
        --cfg                     - Generate a the control flow graph in
Graphviz format.
        --cfg-full                 - Generate a the control flow graph in
Graphviz format (including instructions)
        --decompile                - Decompile a given function or all the
bytecode.
```



```
int256_t _dispatcher() __noreturn

void* const __return_addr  {Frame offset 0}

_dispatcher:
00000000  6060                  PUSH1  0x60
00000002  6040                  PUSH1  0x40
00000004  52                    MSTORE
00000005  6004                  PUSH1  0x4
00000007  36                    CALLDATASIZE
00000008  10                    LT
00000009  61004c                PUSH2  0x4c
0000000c  57                    JUMPI

0000000d  6000                  PUSH1  0x0
0000000f  35                    CALLDATALOAD
00000010  7c0100000000000000…   PUSH29 0x100000000000000000000000000000000000000000000000000000000
0000002e  90                    SWAP1
0000002f  04                    DIV
00000030  63ffffffff            PUSH4  0xffffffff
00000035  16                    AND
00000036  80                    DUP1
00000037  6341c0e1b5            PUSH4  0x41c0e1b5  // kill()
0000003c  14                    EQ
0000003d  610051                PUSH2  0x51
00000040  57                    JUMPI

00000041  80                    DUP1
00000051  5b                    JUMPDEST
```

Fig. 10.16 Binary Ninja interface

10.5.3 Binary Ninja

In order to de-compile the smart contract, we can use the Etherplay plug-in of Binary Ninja. Binary Ninja is a premium software, and its official url is https://arvanaghi.com/blog/reversing-ethereum-smart-contracts/ (Fig. 10.16).

In this section, we introduce all kinds of debug tool: built-in features, command-line tools, and IDEs. At last, we will explore a little bit about WebAssembly which is believed to be the future of smart contract programming.

Part V
Prospect

Chapter 11
WebAssembly(WASM)

WebAssembly (abbreviated Wasm) is a binary instruction format for a stack-based virtual machine. Wasm is designed as a portable target for compilation of high-level languages like C#/C++, enabling deployment on the web for client and server applications. WebAssembly defines an Abstract Syntax Tree (AST) which gets stored in a binary format.
(https://webassembly.org/)

WASM is not designed for some specific CPU directly, but instead is an intermediate representation that can be compiled to actual machine code by the

WebAssembly runtime environment. WASM can be thought as the Universal Virtual Machine where developer write ANY code once, and it runs everywhere.

"Regular" assembly is a set of primitive instructions that can be compiled into executable machine code for different physical hardware, such as x86 and arm opcodes. On contrary, WASM is a set of primitive instructions that are compiled into executable code running on a platform-agnostic and stack-based virtual machine.

The foremost goal of WebAssembly is to compile source code written in different programming languages (such as C++, Rust, Go, etc.) to "execute at native speed." That is to say, even though the WASM code might have been loaded by a browser, the code will execute nearly as fast as if it were a native binary.

11.1 Blame for EVM

Generally, a blockchain virtual machine (VM) should satisfy the following requirements:

- Secure

  Anyone should be able to write and deploy smart contract and no one can violate their permissions in using the smart contract platform.
- Deterministic

  Smart contract running on different nodes, platform, OS, or even architecture will produce the same results.
- Generic

  Using the primitives provided by VM, developers basically are able to implement any business logic.
- Efficient

  To end user, running smart contract should be affordable and economically feasible, since the execution costs multiple computation resources, such as CPU, memory, and hard disc. All these resources are provided by miners and they need to get paid for providing such service and resources.

Ethereum was the first practical implementation, and by far the most-used blockchain platform currently. Roughly speaking, EVM meets the requirement of being secure and deterministic. However, even though EVM is a Turing-complete machine, it is still not generic and efficient. An important method to improve efficiency is just-in-time(JIT) compilation technology which means compiling code to native code while running. The pre-requirement of applying JIT technology is that the language must be interpreted for complexity analysis, which is not easy and simple for EVM.

Over the past three years of smart contract development, the cryptocurrency community has reported smart contracts written in Solidity subverted by a variety of bugs and exploits, such as the DAO exploit and the Parity multi-sig wallet bug. It is common to point the finger to the preponderance of unsafe smart contracts on the

Solidity smart contract language and its many variants. However, as we described in Chapter 8, Solidity has some of the worst flaws:

- Lack of inspection and traceability
- Opacity and illegibility of code on-chain
- Expensive, slow, and dangerous calls to external smart contracts

All the deficiencies stem directly from foundational design decisions in the architecture of the EVM. Solidity still has a long way to go in terms of programmer productivity and language expressiveness. In this section, we examine some of the flaws in detail.

11.1.1 Lack of Modern VM Feature

A well-constructed VM seeks to protect language designers from dangerous mistakes and support efficient construction with central features like dispatch support, name resolution, and so forth. The EVM, however, fails to provide similar guarantees. Ethereum aims to be a global computation machine. It is possible to have a Turing-complete machine without subjecting developers to risks that modern VMs have eliminated for decades. Instead, the EVM disregards this wisdom and expects the programming languages that compile to it to address them.

The EVM is too complex to be safe in the austerity of its bytecode language and native functionality. And it does not contain enough of the security features of a proper VM by construction. If the EVM were to adopt a stricter, Turing-incomplete computational model, it could approach the safety guarantees of Bitcoin bytecode. On the other hand, if the EVM offered features that one would expect from a modern VM, then it would approach the low-level safety afforded by machines such as the Java Virtual Machine (JVM).

Hence, any programming language targeting the EVM must contend with its unsafe design choices and lack of modern VM features. The EVM leaves many features and critical components of its execution model unhandled, forcing language designers to manually implement them. For example, the EVM leaves the following features up to the smart contract language:

- True library support
- Richer data types
- Direct support and enforcement of interfaces/APIs

11.1.2 Human Readability

As shown in Chapter 6, currently, the EVM bytecode from compilation of Solidity program is hardly human-readable like below.

```
 608060405234801561001057600080fd5b5060405160208061 03ee833981
0180604052810190808051906020019092919050505080600081905550806
00160003373ffffffffffffffffffffffffffffffffffffffff1673ffffff
ffffffffffffffffffffffffffffffffff168152602001908152602001600
02081905550506103608061008e6000396000f30060806040526004361061
0057576000357c01000000000000000000000000000000000000000000000
00000000000900463...
5600a165627a7a7230582041e440ef41138511bd018bb2004da6344b9aeb2
49ba3cedf943e1e5d786bf67a0029
```

EVM treat bytecode as a target machine code and it is impossible to read Solidity contract on-chain if the source code is not uploaded. In doing so, EVM loses the critical safety feature of making contract readable on-chain. This introduces a lot of complexity in debugging and developing due to cognitive overhead.

In contrary to Ethereum, Bitcoin provides a safe, simple, and legible bytecode language intended to guarantee the consistency and correctness of programs running on the blockchain. It offers a minimal instruction set and uses bytecode primarily to keep payloads small. It was never intended to be a compile target for a general-purpose language. These deliberate, language-level restrictions minimize the cognitive overhead of understanding coding logic so that developers can reason more effectively about their own code as well as the code of others. Indeed, experienced Bitcoin developers can read and interpret Bitcoin opcodes directly.

As an alternative, Pact (Kadena project) takes a different approach to provide an interpreted language. As shown in history, if a programming language cannot provide inlining, caching, and "just-in-time" optimization, an interpreted language can out-perform its compiled competitor, while offering superior legibility.

11.1.3 Expensive and Slow

Contract running on the EVM utilize an opaque and "top-down" execution model: In order for EVM to find a particular function to run, EVM load entire content of the smart contract as a whole black box and run opcodes from beginning. It lacks the modern VM feature of "load and execute on demand." For example, JVM allows to load function individually and to call it directly with the support of namespace and modular design. EVM execution model is simply impractical to write many programs since the amount that a user would have to pay in order to use the software makes it economically infeasible.

More, EVM does not provide standard libraries which is a common feature of modern VMs. Ideally, if smart contracts had access to a standard library, these function calls should be able to redirect many common tasks to the standard library rather than implementing them in the smart contract directly. Executing every single instruction in a user smart contract will costs GAS. Every time forcing developers to pay significant extra gas just for setting up basic functionality makes running and deploying smart contracts on the EVM much more expensive.

In EVM, a solution for cost control is to use built-in contracts. For example, we may find a lot of ERC20 contract using the following safe Math library.

```
 library SafeMath {

   /**
   * @dev Multiplies two numbers, throws on overflow.
   */
     function mul(uint256 a, uint256 b) internal pure returns
(uint256 c) {
        // Gas optimization: this is cheaper than asserting 'a' not
being zero, but the
        // benefit is lost if 'b' is also tested.
        // See:
https://github.com/OpenZeppelin/openzeppelin-solidity/pull/522
        if (a == 0) {
           return 0;
        }

        c = a * b;
        assert(c / a == b);
        return c;
     }

   /**
   * @dev Integer division of two numbers, truncating the quotient.
   */
     function div(uint256 a, uint256 b) internal pure returns
(uint256) {
        // assert(b > 0); // Solidity automatically throws when
dividing by 0
        // uint256 c = a / b;
        // assert(a == b * c + a % b); // There is no case in which this
doesn't hold
        return a / b;
     }

   /**
   * @dev Subtracts two numbers, throws on overflow (i.e. if
subtrahend is greater than minuend).
   */
     function sub(uint256 a, uint256 b) internal pure returns
(uint256) {
        assert(b <= a);
        return a - b;
     }
```

(continued)

```
    /**
    * @dev Adds two numbers, throws on overflow.
    */
      function add(uint256 a, uint256 b) internal pure returns
(uint256 c) {
          c = a + b;
          assert(c >= a);
          return c;
      }
  }
```

Resource:https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/math/SafeMath.sol

Safe Math library is provided by openzeppline team and went through comprehensive code review by community. So, it is believed to be strong and safe and is recommended as a best practice. Using it as a built-in contract/library is to save gas and to avoid external calls (which is believed to lower risk). However, from a bigger viewpoint, is it reasonable to include exactly same code in many ERC20 contracts? To be frank, including safe Math library in smart contract looks clumsy: it needs more space and lead to slower execution of the function. In summary, built-in solution is an unsatisfying workaround that can mitigate gas cost but do nothing to improve the efficiency of EVM's execution model.

11.1.4 Dangerous

In addition to being slow and expensive, external calls (call and delegatecall opcodes) on the EVM are deeply unsafe. Because a calling contract is not able to determine what in-contract references are valid in some other contract, external call just call the code at reference address blindly. There is absolutely no protection at runtime from a reference referring to a malicious code. For example, in Parity wallet issue, wallets called a central core contract which was deleted subsequently. All the funds in those wallets are locked up due to the disappearance of the contract. Another example is the unsafe contract reference for "user" account (EOA) in DAO hack. Calling it from a malicious contract account triggered the default method which execute send method inexplicably to initiate the recursion exploit.

11.1.5 No Support of Multi-Sig and Upgradable Contract

In Ethereum, each contract or account has an address, which is a hash representation of public key. The consequence of this design approach disallows the native support of multi-sig contract and makes it impossible to upgrade a contract at a particular address (it is a design choice derived from the belief of “Code is law” which is advocated by Ethereum team). Since Ethereum does not have any name base resolution, it is also impossible to do code dispatch in Ethereum. Without code dispatch and name-based resolution, there would be no way to detect if code at an address is upgraded. To make situation much worse, the singular address format (which means one signature only for one address) forces all Ethereum contracts to be single signature. And in order to support multi-sig functionality, developer need to use more expensive solutions, such as built-in contract, or multiple transaction calls.

Due to the single signature approach of Ethereum, there are a lot of solutions in market which aims to fill the gap: proxy contract, multi-sig wallet, etc. And for sure, such workarounds are more expensive and add more complexity to contract development.

11.2 Web Assembly

From the webassembly’s official web site (webassembly.org): "WebAssembly or wasm is a new portable, size, and load-time-efficient format suitable for compilation to the web." So, WebAssembly is an excellent candidate for your technical architecture if creating computation-intensive and high-performance web apps.

11.2.1 Features

WebAssembly has the following superior features:

11.2.1.1 Performance—Near Native Speed

WebAssembly allows us to run code at near native speed which supports many other languages. It allows us to develop CPU-intensive calculations and executions on the *frontend*. With the support of WASM, it will allow us to create applications that were once only possible on the desktop, such as word processors, complex simulations, image editing software, and games. We can implement performance critical stuff in various modern languages, such as C++, Rust and Golang, and compile it into WASM. On the frontend, WASM file can be imported like a standard JavaScript module. By using different languages, we can take advantage of common hardware capabilities available on a wide range of platforms. And that is the true power of WASM.

In general, WebAssembly is faster than JavaScript because:

- Less time to fetch WebAssembly because it is more compact than JavaScript in text form, even when compressed. This means fast-delivery and low latency

- Less time to decode WebAssembly than to parse JavaScript.
- Less time to compile and optimize because WebAssembly is closer to machine code than JavaScript. And it already has gone through optimization on the server side.
- No need to reoptimize because WebAssembly has types and other information built in, so the JavaScript engine does not need to speculate when it optimizes the way it does with JavaScript.
- Less time to execute WebAssembly because there are fewer compiler tricks and gotchas that the developer needs to know to write consistently performant code. Moreover, WebAssembly's set of instructions are much more fit for machines.
- No need for garbage collection since the memory is managed manually.

11.2.1.2 No Plugins

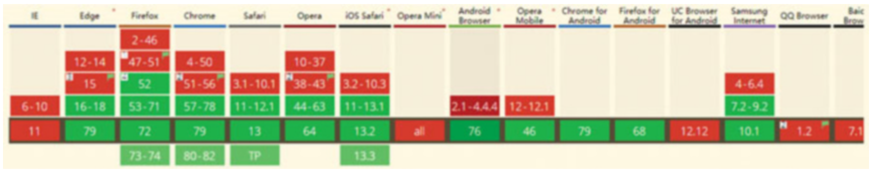
We do not need to install browser plug-in since most of mainstream browsers implemented WASM standard at the moment. Following chart shows the cutting-edge status quo of WASM-supported browsers (Fig. 11.1).

11.2.1.3 Work with JavaScript

WASM works perfectly with JavaScript in both directions. WASM can manipulate DOM object, events in a similar way to JavaScript. And JavaScript can trigger the execution of appropriate WASM code just like restful API and RPC call.

11.2.1.4 Portability

With the support of WASM, we will be able to easily port massive libraries of existing C and C++ apps to the web. As toolchains and the WASM spec evolve, apps built other languages like Java or Python will also be supported. To port existing apps or libraries, we need to go through the 3-step process: (1) develop app/library using whatever WASM-supported programming language appropriate; (2) compile the code into WASM format; (3) JavaScript engine of WASM-supported browser translate the WASM code and run it (Fig. 11.2).



IE	Edge	Firefox	Chrome	Safari	Opera	iOS Safari	Opera Mini	Android Browser	Opera Mobile	Chrome for Android	Firefox for Android	UC Browser for Android	Samsung Internet	QQ Browser	Baic Brow
		2-46													
	12-14	47-51	4-50		10-37										
	15	52	51-56	3.1-10.1	38-43	3.2-10.3							4-6.4		
6-10	16-18	53-71	57-78	11-12.1	44-63	11-13.1		2.1-4.4.4	12-12.1				7.2-9.2		
11	79	72	79	13	64	13.2	all	76	46	79	68	12.12	10.1	1.2	7.1
		73-74	80-82	TP		13.3									

Fig. 11.1 WASM-supported browsers. https://caniuse.com/#search=webassembly

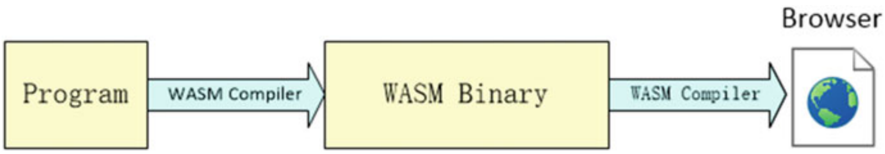


Fig. 11.2 WASM working mechanism

11.2.1.5 Readability

WebAssembly code defines an AST (Abstract Syntax Tree) represented in a binary format. You can author and debug in a text format so it is readable. A WASM file is looked like as below. And it is a simple function from the WASM specification tests:

```
(module
  (func (param i32) (param i32) (result i32)
    get_local 1
    (block (result i32)
       get_local 0
       get_local 0
       br_if 0
       unreachable
    )
    i32.add
  )
)
```

You should be able to interpret the WASM code if well-trained.

11.2.2 WebAssembly Runtime

In the world, there are several teams working on WebAssembly runtime environment, such as KWASM, Microwasm, and EWASM. Since EWASM is the primary candidate to replace the EVM as part of the Ethereum 2.0 "Serenity" roadmap and EWASM is also proposed for adoption on the Ethereum mainnet (EWASM, 2019), we are going to shed some light on EWASM.

The main goals of EWASM is to enhance performance and support smart contract development across an increasing number of languages including C++, Rust, etc. EWASM defines the EWASM Contract interface (ECI) and Ethereum Environment Interface (EEI). Any programming language which can implement ECI and EEI can be used to write smart contract.

EWASM proposes following integration path for existing languages (Fig. 11.3):

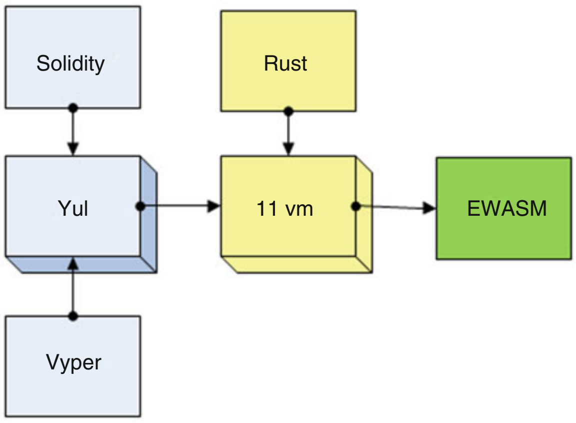


Fig. 11.3 EWASM working flow

- For existing smart contract programming languages, such as Solidity and Vyper, EWasm suggests to compile them into Yul, and then to coding backend (such as LLVM). And finally to deploy it on EWASM testnet
- For existing programming languages, such as Rust, Go, and C++, Ewasm suggests to compile them to coding backend (such as LLVM and emscripten) and then to deploy it on EWASM testnet

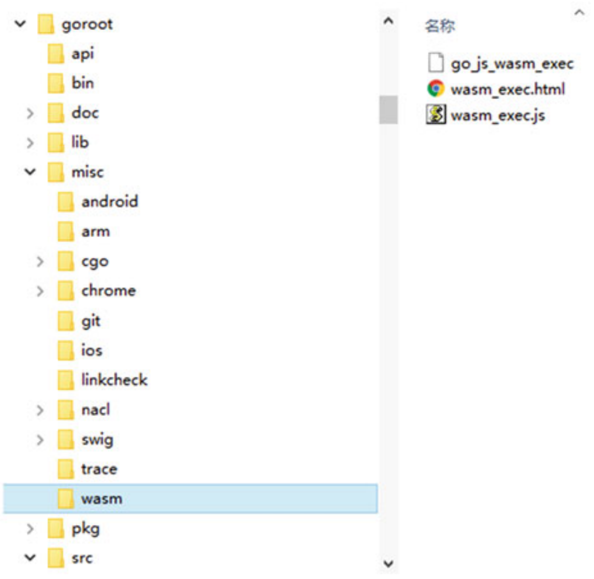
11.3 Golang + WASM

Starting from Go 1.11, WebAssembly is shipped with the syscall/js package. The syscall/js package allows our Go code to interact with JS references, mainly through the js.Value type with methods such as Get(), Set() or Invoke(). If installed correctly, syscall/js package can be found under $GOROOT/misc/wasm (Fig. 11.4).

11.3.1 Basic Usage

Under $GOROOT/misc/wasm, there are 2 files we want to talk about: (1) **Wasm_exec.js** is the glue provided by Go team and it encapsulates a global Go object. (2) **Wasm_exec.html** demonstrates how to interact with backend written in Go. Here is the content of wasm_exec.html:

Fig. 11.4 Wasm directory



```
<!doctype html>
<!--
Copyright 2018 The Go Authors. All rights reserved.
Use of this source code is governed by a BSD-style
license that can be found in the LICENSE file.
-->
<html>

<head>
        <meta charset="utf-8">
        <title>Go wasm</title>
</head>

<body>
        <!--
        Add the following polyfill for Microsoft Edge 17/18 support:
        <script src="https://cdn.jsdelivr.net/npm/text-
encoding@0.7.0/lib/encoding.min.js"></script>
        (see https://caniuse.com/#feat=textencoder)
        -->
        <script src="wasm_exec.js"></script>
        <script>
                if (!WebAssembly.instantiateStreaming) { // polyfill
                        WebAssembly.instantiateStreaming = async
```

(continued)

```
(resp, importObject) => {
                              const source = await (await resp).
arrayBuffer();
                              return await WebAssembly.instantiate
(source, importObject);
                      };
              }

              const go = new Go();
              let mod, inst;
              WebAssembly.instantiateStreaming(fetch("test.
wasm"), go.importObject).then((result) => {
                      mod = result.module;
                      inst = result.instance;
                      document.getElementById("runButton").
disabled = false;
              }).catch((err) => {
                      console.error(err);
              });

              async function run() {
                      console.clear();
                      await go.run(inst);
                      inst = await WebAssembly.instantiate(mod,
go.importObject); // reset instance
              }
       </script>

       <button onClick="run();" id="runButton" disabled>Run</
button>
</body>

</html>
```

As can be seen from above, to interact with backend in Golang, we need to:

- Include wasm_exec.js into <Head> tag
  <script src="wasm_exec.js"></script>
- Instantiate WebAssembly and should be polyfilled for unsupported browsers
  WebAssembly.instantiateStreaming
- Create Go object for later use
  const go = new Go();
- Fetch WASM file into browser
  WebAssembly.instantiateStreaming(fetch("test.wasm"), go.importObject)

  In the example, we fetch test.wasm. Actually, we can fetch whatever WASM file we want.

11.3.2 Hello WASM Example

In this section, we start WASM programming with a simple Hello world sample like below. The file name is main.go

```
package main

import "fmt"
 func main() {
    fmt.Println("Welcome to WASM!")
}
```

Then we need to use the following command to compile the go file above into WASM

Linux

```
GOOS=js GOARCH=wasm go build -o main.wasm
```

Windows

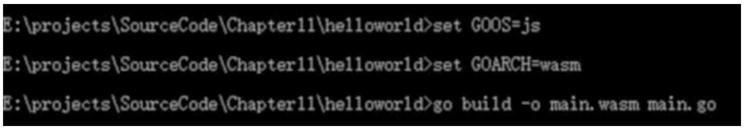
```
E:\projects\SourceCode\Chapter11\helloworld>set GOOS=js
E:\projects\SourceCode\Chapter11\helloworld>set GOARCH=wasm
E:\projects\SourceCode\Chapter11\helloworld>go build -o main.
wasm main.go
```

Here GOOS is the target "operating system" which in this instance is JavaScript, and GOARCH is the architecture which in this case is WebAssembly (WASM) (Fig. 11.5).

The output should be the same as below if everything runs smoothly. A new file—main.wasm is created (Fig. 11.6).

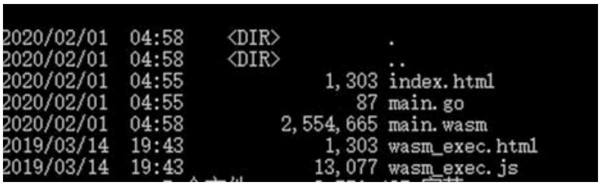
And then, we need to adjust index.html to fetch main.wasm (Fig. 11.7):

To verify, we need to start a simple http server and the most important step is to enable httpserver to accept content-type—"application/wasm." If you are using nginx for test purpose, you need to update mime.types file under nginx configuration directory. Here, we use SimpleHTTPServer written in Python and we need to tweak a little bit to respond wasm request correctly. The code (server.py) is like below:



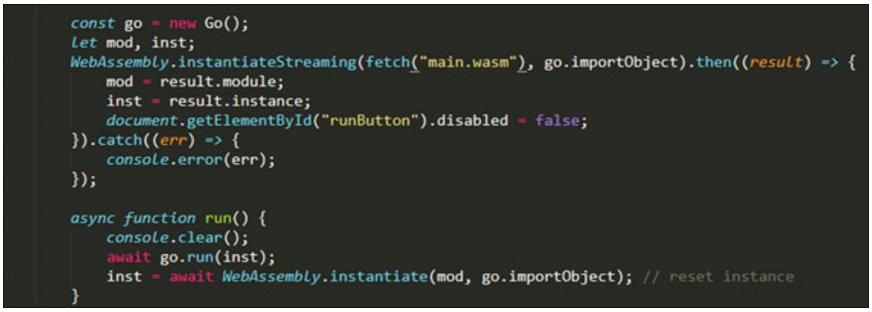
```
E:\projects\SourceCode\Chapter11\helloworld>set GOOS=js
E:\projects\SourceCode\Chapter11\helloworld>set GOARCH=wasm
E:\projects\SourceCode\Chapter11\helloworld>go build -o main.wasm main.go
```

Fig. 11.5 Set environmental variables



```
2020/02/01  04:58    <DIR>          .
2020/02/01  04:58    <DIR>          ..
2020/02/01  04:55             1,303 index.html
2020/02/01  04:55                87 main.go
2020/02/01  04:58         2,554,665 main.wasm
2019/03/14  19:43             1,303 wasm_exec.html
2019/03/14  19:43            13,077 wasm_exec.js
```

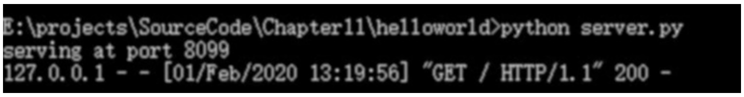
Fig. 11.6 Successful build result



```
const go = new Go();
let mod, inst;
WebAssembly.instantiateStreaming(fetch("main.wasm"), go.importObject).then((result) => {
    mod = result.module;
    inst = result.instance;
    document.getElementById("runButton").disabled = false;
}).catch((err) => {
    console.error(err);
});

async function run() {
    console.clear();
    await go.run(inst);
    inst = await WebAssembly.instantiate(mod, go.importObject); // reset instance
}
```

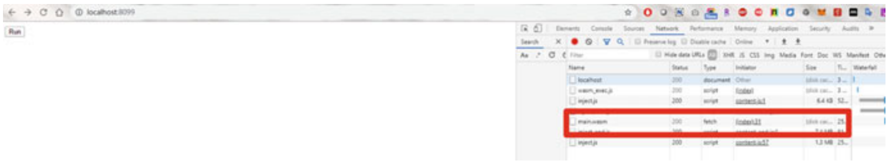
Fig. 11.7 Nodejs code to fetch main.wasm

```
import http.server
from http.server import HTTPServer, BaseHTTPRequestHandler
import socketserver

PORT = 8099

Handler = http.server.SimpleHTTPRequestHandler

Handler.extensions_map={
   '.wasm': 'application/wasm',
   '.manifest': 'text/cache-manifest',
   '.html': 'text/html',
    '.png': 'image/png',
   '.jpg': 'image/jpg',
   '.svg':  'image/svg+xml',
   '.css':  'text/css',
   '.js': 'application/x-javascript',
   '': 'application/octet-stream', # Default
    }

httpd = socketserver.TCPServer(("", PORT), Handler)

print("serving at port", PORT)
httpd.serve_forever()
```



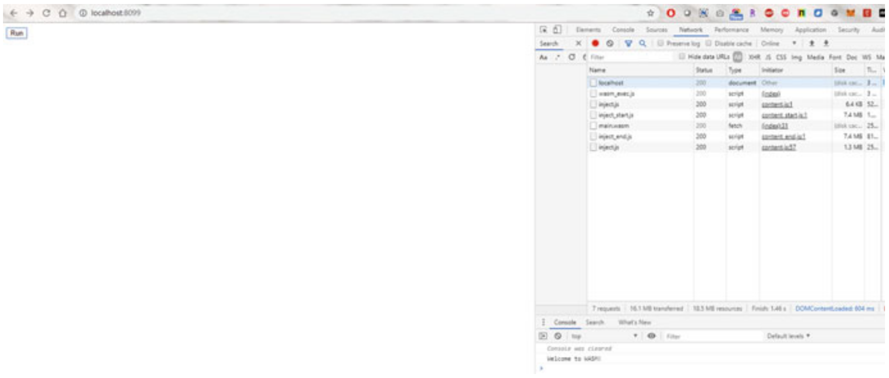
```
E:\projects\SourceCode\Chapter11\helloworld>python server.py
serving at port 8099
127.0.0.1 - - [01/Feb/2020 13:19:56] "GET / HTTP/1.1" 200 -
```

Fig. 11.8 Simple Python HTTP server supporting wasm content



Name	Status	Type	Initiator	Size
localhost	200	document	Other	(disk cac...
wasm_exec.js	200	script	(index)	(disk cac...
inject.js	200	script	content.js:1	6.4 KB
main.wasm	200	fetch	(index):31	(disk cac...
inject.js	200	script	content.js:57	1.3 MB

Fig. 11.9 index.html



Name	Status	Type	Initiator	Size
localhost	200	document	Other	(disk cac...
wasm_exec.js	200	script	(index)	(disk cac...
inject.js	200	script	content.js:1	6.4 KB
inject_start.js	200	script	content_start.js:1	7.4 MB
main.wasm	200	fetch	(index):31	(disk cac...
inject_end.js	200	script	content_end.js:1	7.4 MB
inject.js	200	script	content.js:57	1.3 MB

7 requests | 16.1 MB transferred | 18.3 MB resources | Finish: 1.46 s | DOMContentLoaded: 604 ms

```
Console was cleared
Welcome to WASM!
```

Fig. 11.10 Run button console outputl

Resource: https://gist.github.com/HaiyangXu/ec88cbdce3cdbac7b8d5

And in command prompt window, we type the command:

```
python server.py
```

It should look like below if everything works OK (Fig. 11.8).

Then we open any browser and type localhost:8099 to access the page—index.html (Fig. 11.9):

You should see a "Run" button and main.wasm is loaded correctly (check the panel on the right above). And click "Run" button, you should see our welcome message in console output (Fig. 11.10).

11.3.3 Interaction Between Golang and Frontend JavaScript

Usually, in a professional App, there will be a lot of interactions between backend and frontend. For example: manipulating DOM elements from Go module; invoking frontend methods and waiting for callback from backend. Here we will summarize how Go module interacts with front JavaScript.

11.3.3.1 From Go to JS

In syscall/js package, there is a new type js.Value available. Following table is member APIs of js.Value:

No	API	Description

1	js.Value.Get()
	js.Value.Set()
		Retrieve and set properties on an Object value

2	js.Value.Index()
	js.Value.SetIndex()
		Retrieve and set values in an Array value

3	js.Value.Call()	Calls a method on an Object value
4	js.Value.Invoke()	Invokes a function value
5	js.Value.New()	Invokes the new operator on a reference representing a JS type

6	js.Value.Int()
	js.Value.Bool()
		Retrieve a JavaScript value in its corresponding Go type

7	js.Undefined()	js.Value corresponding to JS's undefined
8	js.Null()	js.Value corresponding to JS's null
9	js.Global()	js.Value giving access to JS's global scope
10	js.ValueOf()	Returns the corresponding js.Value

Following code snippet shows how to use js package to manipulate DOM. It is easy to understand because of its similarity to JavaScript syntax.

```
func setup() {
        window = js.Global()
        doc = window.Get("document")
        body = doc.Get("body")

        windowSize.h = window.Get("innerHeight").Float()
        windowSize.w = window.Get("innerWidth").Float()

        canvas = doc.Call("createElement", "canvas")
        canvas.Set("height", windowSize.h)
        canvas.Set("width", windowSize.w)
        body.Call("appendChild", canvas)
```

(continued)

```
        // red laser dot canvas object
        laserCtx = canvas.Call("getContext", "2d")
        laserCtx.Set("fillStyle", "red")
}
```

11.3.3.2 From JS to Go

In Go 1.12, there is a new js.Func() type, which represents a Go function wrapped in order to be used as a JS callback. Its counterpart in Go 1.11 is js.Callback(). Here is a simple example showing how to implement callback and send control over to frontend.

```
package main

import (
    "fmt"
    "syscall/js"
)

var done = make(chan struct{})

func main() {
    callback := js.NewCallback(printMessage)
    defer callback.Release()

    setPrintMessage := js.Global().Get("setPrintMessage")
    setPrintMessage.Invoke(callback)
    <-done
}

func printMessage(args []js.Value) {
    message := args[0].String()
    fmt.Println(message)
    done <- struct{}{}
}
```

Several key points we need to pay attention to are:

- Channel

  Since the callback function is implemented in a dedicated goroutine, main goroutine needs to wait for the callback to be called. Channel ("done" in example above) is used to inform main goroutine of the end of execution of callback function

- Release the callback

  Once we create a new callback function as js.Callback or js.Func, it is always a best practice to release it before the wasm binary terminate.
- Need to re-instantiate each time

  On frontend, we need to instantiate WebAssembly everytime when we call backend methods. This may cause a performance issue. And certainly, we have solution to improve it.

11.3.4 Solidity to WASM

We have discussed a lot above about WASM. But the question is: how does WASM ameliorate smart contract programming in Solidity? The idea here is: if EVM support WASM, programmers can use whatever programming language they prefer, such as Go, Swift, Rust, Ruby, C++, and OCaml, to develop smart contract. The only requirement is that the programming language selected should be able to be compiled into WASM format which is supported by different platforms, such as different browsers (Chrome/Firefox…), OSs (windows/linux…), and CPUs (x86/ARM…). In this way, EVM can enjoy the benefit provided by mature platforms and their low-level infrastructures which are believed to be safer and more stable since they already run in production longer enough than Ethereum.

During the 2019 Ethereum Foundation Devcon5 in Osaka, Japan, Second State demonstrated the alpha release of its SOLL compiler project. It is the world's first LLVM-based toolchain that can compile Solidity smart contracts into WebAssembly bytecode and successfully deploy onto the official EWasm (Ethereum flavored WebAssembly) testnet. SOLL project is located at https://github.com/second-state/soll and it contains practical examples about how to compile and deploy an ERC20 contract into WASM.

Thanks for reading this far!!!