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

This Springer imprint is published by the registered company Springer Nature Singapore Pte Ltd.
The registered company address is: 152 Beach Road, #21-01/04 Gateway East, Singapore 189721, Singapore

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:

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

Initial state is σ, transformed state is σ', T means transaction, Y 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

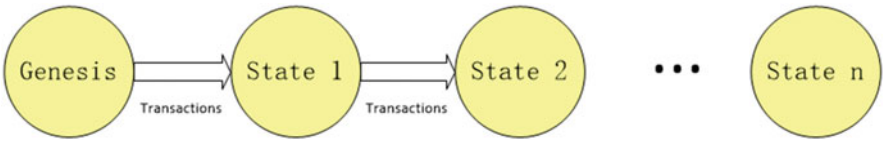

Fig. 1.1 Ethereum state machine

- 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 Y 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).

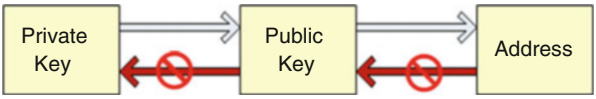

Fig. 1.2 Private key, public key, address relationship

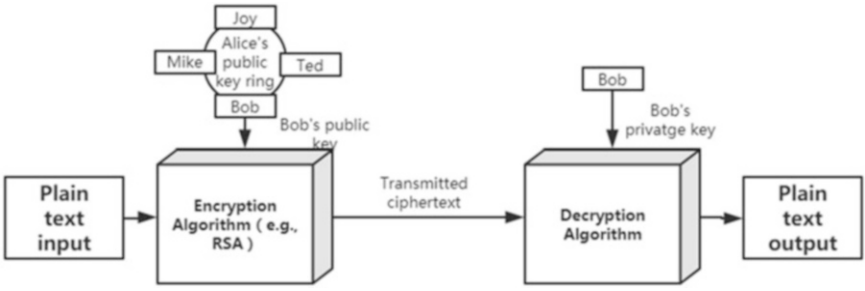

Fig. 1.3 Use priv/pub to encrypt and decrypt message

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.

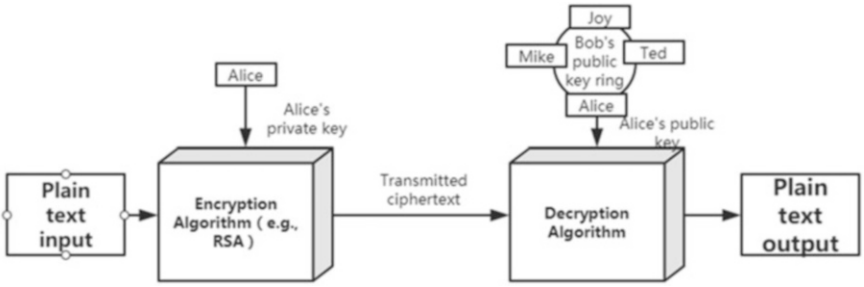

Fig. 1.4 Verify signature https://www.cnblogs.com/moonfans/p/3939335.html

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.

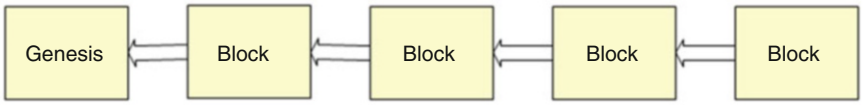

Fig. 1.5 Blockchain

- *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

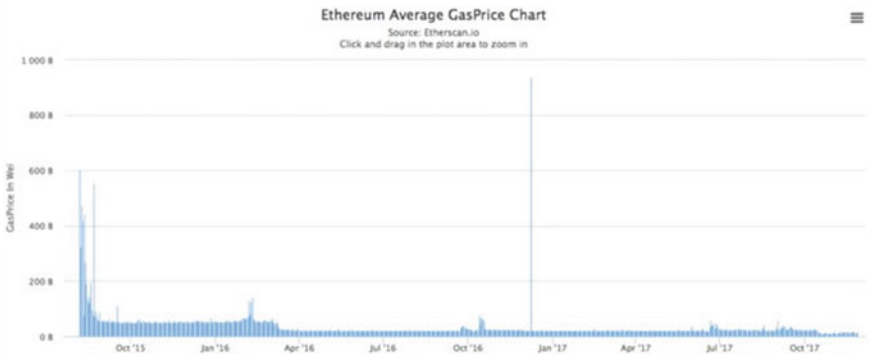

Image Credit: Etherscan

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;
 }
}
```


G. Zheng et al., *Ethereum Smart Contract Development in Solidity*,
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: