

Liang Cai · Qilei Li
Xiubo Liang

Advanced Blockchain Technology

Frameworks and Enterprise-Level
Practices

Translated by
Nan Yu



人民邮电出版社
POSTS & TELECOM PRESS



Springer

Advanced Blockchain Technology

Liang Cai • Qilei Li • Xiubo Liang

Advanced Blockchain Technology

Frameworks and Enterprise-Level Practices



Liang Cai
School of Software Technology
Zhejiang University
Hangzhou, Zhejiang, China

Qilei Li
School of Software Technology
Zhejiang University
Hangzhou, Zhejiang, China

Xiubo Liang
School of Software Technology
Zhejiang University
Ningbo, Zhejiang, China

Translated by
Nan Yu

Translation proofreading
Xiufang Niu

ISBN 978-981-19-3595-4 ISBN 978-981-19-3596-1 (eBook)
<https://doi.org/10.1007/978-981-19-3596-1>

Jointly published with Posts & Telecom Press
The print edition is not for sale in China (Mainland). Customers from China (Mainland) please
order the print book from: Posts & Telecom Press

© Posts & Telecom Press 2022

This work is subject to copyright. All rights are solely and exclusively licensed by the Publisher, whether the whole or part of the material is concerned, specifically the rights of reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

The publishers, the authors, and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publishers nor the authors or the editors give a warranty, express or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publishers remain neutral with regard to jurisdictional claims in published maps and institutional affiliations.

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

Foreword 1

Building an Independent and Controllable Alliance Blockchain, Opening Up a New Horizon for the Development of Fintech

In recent years, “digital cryptocurrencies,” represented by Bitcoin, have become popular worldwide, bringing new opportunities and challenges. As the underlying support technology for the “digital cryptocurrency” system, blockchain has shown great potential application value and will trigger a wave of technological innovation in many fields such as finance, trade, logistics, credit collection, Internet of Things, and sharing economy. Developed countries such as the United States, the United Kingdom, Russia, Australia, and Japan have already carried out research and application of blockchain from a strategic level. The Chinese government also attaches great importance to the development of blockchain technology. The State Council has included blockchain in the National Informatization Plan of the 13th 5-Year Plan, and the Ministry of Industry and Information Technology has issued white papers on the development of China’s blockchain industry for many years. Generally speaking, the current blockchain technology is still in its infancy, and there is not a big gap between domestic and foreign development. In order to avoid the situation that the underlying key technologies such as operating system and database are monopolized by foreign giants in this important emerging technology field, it is of great significance to develop a fully dependent and controllable blockchain underlying platform.

Blockchain technology is essentially a distributed ledger database that applies a blockchain data structure to validate and store data, a distributed consensus-based algorithm to generate and update data, and a cryptographic approach to secure data transmission and access. On a functional level, blockchain, whose records are not easily tampered with, does not require a third party trusted intermediary, making it naturally suitable for multiple organizations to monitor and reconcile accounts in

real-time in a blockchain network, greatly enhancing the automation of economic activities and contracts through smart contracts.

According to their organizational forms, blockchain platforms consist of public chains, federated chains, and private chains. Public chains are non-permitted entirely decentralized but suffer from extremely low consensus efficiency, and lack of authority control and privacy protection. It is complicated to be used in other fields than “digital cryptocurrencies.” Alliance chains are permission chains, which require specific permission to join the network. Private chains are also permission chains, where the permission is in the hands of every single organization.

In terms of time, blockchain technology has undergone three significant technological evolutions since its birth. The first took place in 2009, represented by the platform “Bitcoin,” which for the first time demonstrated the feasibility of a “digital currency” without a central authority but with a deficient transaction performance of a few transactions per second. The second occurred in 2013, represented by Ethereum, which added programmability to the blockchain platform for the first time, thus greatly expanding its scope of application. However, its transaction performance was still low, at a few dozen transactions per second. The third occurred in 2015, represented by the platforms Hyperledger Fabric and Hyperchain. For the first time, permission control and privacy protection were added to blockchain platforms, and transaction performance increased to several thousand or tens of thousands of transactions per second.

Alliance blockchain has features such as efficient consensus, smart contract, multi-level encryption, authority control, privacy protection, and others, supported by visual monitoring and dynamic configuration. It is designed to mainly orient to enterprise-level application scenarios. It is the latest form of blockchain development and has extremely wide application value in China. Its core advantages mainly include three points: (1) From the perspective of supervision, alliance blockchain can facilitate regulation through CA authentication access and the formulation of regulatory rules contracts; (2) As commercial institutions and users have privacy protection requirements for account and part of transaction information, alliance blockchain can realize privacy protection through encryption, partition, and other ways; (3) From the perspective of commercial applications, transaction throughput and latency are the most important transaction performance indicators that concern enterprises most. The federated blockchain has greatly improved transaction efficiency through the innovation of consensus algorithms. However, some scenarios with business requirements put higher technical requirements on the alliance blockchain, for example, (1) High performance, how to reach consensus among multiple nodes efficiently and how to improve the efficiency of smart contract execution effectively; (2) High availability, which should allow for new nodes to be added without crashing and for rapid recovery after a node reboot; (3) Security and privacy, how to design permission control mechanisms that are compliant with national standards and can effectively protect private data; (4) Programmability, which should provide a Turing-complete and secure smart contract engine that can support multilingual and complex smart contracts. However, the current mainstream

open-source blockchain platforms (e.g. Etherum, Hyperledger Fabric) have yet to meet the above technical requirements.

Led by faculty members including Yang Xiaohu and Cai Liang, Zhejiang University's Research Centre for Exascale Information Systems has been conducting multi-level research work on the core technologies of the alliance blockchain since 2015. Zhejiang University established the Blockchain Research Centre of Zhejiang University in June 2018, focusing on cutting-edge research on blockchain theory, technology, and applications. The Hyperchain Alliance blockchain platform developed by Hangzhou Hyperchain Technology Co., Ltd. has made breakthroughs in high performance, high availability, security and privacy, and programmable technologies, supporting the implementation and stable operation of the first blockchain project in China to access the core system of banks. At present, Hyperchain platform has been applied in many fields such as finance.

This book provides a detailed analysis of the technical features and kernel code of Ethereum, Hyperledger Fabric, and Hyperchain, and an introduction to the application development techniques of each platform. It is believed that this book will be a good reference for enthusiasts of blockchain technology and practitioners in the blockchain industry.

Chinese Academy of Engineering,
Beijing, China

Chun Chen

School of Computer Science
and Technology,
Zhejiang University,
Hang Zhou, China

Foreword 2

Building a New Multi-party Business Collaboration Platform Using Blockchains

It is my great honor to be invited to write the foreword for this book. I am also glad to see the publication of such a comprehensive, systematic, and integrated introduction to blockchain technology.

With a series of “virtual currencies” represented by Bitcoin becoming a big hit, blockchain, the underlying support technology, has surfaced and received widespread attention. It is becoming clear to practitioners that this technology can be used not only to generate a new type of “digital currency” but also to solve the problem of exchanging products connected to everything, changing the mode of commodity trading, and thus influencing and changing the future shape of the economy and finance. Like TCP/IP, it will even have an inflammatory effect on the development of the Internet. In the future interconnected world, it will significantly impact human society and completely change the mode of social production and people’s way of life. The first generation of the Internet, based on TCP/IP, only solved the problem of efficiency in information transmission, but not the issue of trust. In business scenarios involving multi-party collaboration building information systems and establishing intermediaries to complete information transmission and exchange, a series of additional measures were required to resolve the trust issue between parties, which significantly increased the cost of communication and collaboration between parties. Blockchain technology enables stakeholders to share a network and a set of books from a technical level by establishing a distributed ledger with a multi-node consensus that is difficult to tamper with, thus significantly reducing the communication and workforce costs in the process of business collaboration.

As far as the new multi-party business collaboration system is concerned, the value of blockchain is mainly reflected in four aspects: (1) Reduce the complexity of system interfacing, cross-system data interaction, and routing sink to the blockchain layer, reduce the difficulty and cost of application development, and improve the development efficiency; (2) Improve the liquidity of digital assets and liquidity and

realize value transmission through assets on the chain; (3) Realize whole process monitoring, smart contracts record the flow of information and status, and data is difficult to be tampered with, realizing the entire process of data to be monitored; (4) Credible cooperation between various parties makes it possible to have credible collaboration in the industrial chain, which meets the requirements of the asset-light operation.

The China Foreign Exchange Trade Center has been committed to researching cutting-edge technologies in financial technology (FinTech) for many years and has explored blockchain, artificial intelligence, distributed architecture, microservices, software formalization, etc. The research on blockchain and distributed ledger technology has formed a blockchain architecture plan that is in line with the Center's technical planning and technical development route and has made a breakthrough in key technologies such as national secret algorithm, consensus protocol, and smart contract, and has clarified the improvement plan for general blockchain technology with the fruitful research results. The center has cooperated with the research team led by Teachers Xiaohu Yang and Liang Cai from Zhejiang University for many years in FinTech. In terms of blockchain, it began to cooperate with Hangzhou Hyperchain Technology Co., Ltd. in 2016 and has formed a certain technical reserve. The Hyperchain platform developed by Hangzhou Hyperchain Technology Co., Ltd. has excellent performance, safety, reliability, monitoring visualization, seamless contract upgrade support, horizontal expansion of data storage, etc. It is a competitive, independent, and controllable enterprise-level blockchain underlying platform in China. It is ideal for building a new generation of value transmission and business collaboration platforms based on blockchain.

In addition to the financial sector, blockchain technology is also gaining more and more attention and application in trade, logistics, credit collection, the Internet of Things, social welfare, etc. More and more professionals are engaged in the research and application development of blockchain platforms. Still, there are few books introducing blockchain development technology in the market, and there are even fewer books that can guide readers to practice. Based on years of blockchain technology research and development experience of Zhejiang University and Hangzhou Hyperchain Technology Co., LTD., this book makes an in-depth analysis of well-known open-source blockchain platforms Ethereum and Hyperledger Fabric self-developed Hyperchain platform. The fundamental algorithms and core principles of blockchain are interspersed with explanations of platform functions development guides and project examples for each platform, providing an in-depth introduction to the current mainstream blockchain development technologies. It is believed that this book will help readers to gain a deeper understanding of blockchain technology principles and effectively enhance their blockchain technology development skills.

China Foreign Exchange Trading
Centre, Shang Hai, China

Zaiyue Xu

Preface

Blockchain technology is a major technological innovation in financial technology and IT in general. It is essentially a distributed ledger technology that relies on essential data encryption, timestamps, and distributed consensus algorithms to achieve advanced functions such as chain storage, smart contracts, and privacy protection. Through mutual verification, supervision, and data backup between the nodes of the blockchain network, the technology ensures that the data stored in the chain ledger is challenging to be tampered with maliciously and is particularly suitable for solving the problem of high costs for credit maintenance in multi-party business collaboration scenarios.

Blockchain technology has its roots in “Bitcoin,” which underpins “digital cryptocurrencies.” Since its inception in 2009, the Bitcoin system has been operating steadily for 10 years without a central maintenance organization. With the popularity of Bitcoin, hundreds of “digital cryptocurrencies” have rapidly emerged.

In recent years, it has been found that the blockchain technology behind “digital cryptocurrencies” may have the potential to be of even greater value and could be used in a wide range of future business scenarios. Many experts believe that blockchain technology can address the problem of decentralized value exchange for a new generation of the Internet, i.e., credit for network transmission.

The blockchain technology used for “digital cryptocurrencies” can only implement basic functions such as transactions and transfers and is considered Version 1.0 of blockchain technology. For blockchain to be used in a wide range of scenarios beyond “digital cryptocurrencies,” the technology will have to be refined.

Ethereum was born in 2014. The platform transfers the design and control of business logic into the hands of platform users by supporting smart contracts, allowing Chaincode to be written to meet the needs of a variety of complex business scenarios. The platform is widely acclaimed and sought after as a prime example of blockchain Version 2.0.

However, Ethereum is not a perfect blockchain platform and suffers from low consensus efficiency, lack of privacy protection, difficulties in mass storage and information supervision, so it cannot be applied to large-scale enterprise-level information systems. To solve these problems, some enterprise-level blockchain

platforms have come into being, among which the typical representatives are Hyperledger Fabric supported by IBM and Hyperchain technology. With technologies such as efficient consensus, multi-level encryption, permission control, visual monitoring and dynamic configuration, enterprise-level blockchain platforms have opened up a wider application space for blockchain technology.

Unlike the majority of blockchain books on the market today, this book, with quite strong practicality, does not imagine any application scenarios that cannot be implemented in the short term, but focuses on the technical know-how and development experience.

Readers will be able to get started with the most popular blockchain platforms and quickly develop their first blockchain application by referring to the examples in this book.

Hangzhou, Zhejiang, China

Liang Cai

Hangzhou, Zhejiang, China

Qilei Li

Ningbo, Zhejiang, China

Xiubo Liang

September 2020

Book Structure

This book consists of four parts.

The first part introduces the basics of blockchain and gives readers a quick overview of blockchain technology.

This part contains one chapter, i.e., Chap. 1, which comprehensively analyzes the development of blockchain technology, introduces its concept, history, technical schools, key technologies, and typical application scenarios, makes a comparative analysis of mainstream platforms, and describes the current blockchain industry ecological map.

The second part provides a detailed explanation of the well-known open-source blockchain platforms Ethereum and Hyperledger and introduces how to develop blockchain applications based on these two platforms. This part has four chapters.

Chapter 2 introduces the basics of Ethereum, including its history, basic concepts, clients, account management, and the Ethereum network. It introduces the core principles of critical modules of Ethereum, such as consensus mechanism, virtual machines, data storage, and encryption algorithms. It also provides a detailed introduction to the writing, deployment, testing, and execution of Ethereum smart contracts and concludes with an overview of the major events and problems in the development of Ethereum. Finally, the significant events in the development of Ethereum and the critical issues that exist today are analyzed and discussed.

Chapter 3 first introduces how to set up an Ethereum development environment, including the Go language environment, the configuration of Node.js and npm, installation of the Solc compiler, and construction of a private chain using the Ethereum geth client. Secondly, it introduces the integrated development environment for Ethereum smart contract development, including Mix IDE and online browser compiler. Third, it describes two Ethereum programming interfaces, JSON RPC and JavaScript API, which make it possible to interact with the underlying Ethereum and call contract methods. This is followed by a description of the current mainstream Ethereum development frameworks and processes, including Meteor, Truffle, and a layered and scalable development process for commercial development. At the end of this chapter, the first complete example of Ethereum application development is given.

Chapter 4 provides an in-depth interpretation of Hyperledger Fabric to help readers understand the underlying implementation principles of Fabric. First, it illustrates the development status and management model of Hyperledger and its subprojects, focusing on Hyperledger Fabric. Secondly, it deeply analyzes the Hyperledger Fabric architecture, discussing the components and features of the Hyperledger Fabric from three aspects: member services, blockchain services, and contract code services giving the fabric architecture design and module components. It then presents the structure, invocation method, and execution flow of Chaincode; finally, it analyzes the transaction endorsement process in detail.

Chapter 5 mainly demonstrates how to develop blockchain applications on the Hyperledger Fabric platform. First, it describes the construction process of Hyperledger Fabric development and operating environment and then shows the development and deployment process of Chaincode. Finally, it introduces CLI application interface and SDK interface and illustrates how to develop Hyperledger Fabric blockchain application based on these two interfaces.

The third part explains the core technology of an enterprise-class blockchain platform, using the autonomous controlled federated blockchain Hyperchain as an example. It introduces the technology for developing enterprise-class blockchain applications based on Hyperchain. This part contains two chapters.

Chapter 6 introduces the core components of an enterprise-class blockchain platform, Hyperchain, which is taken as an example. Unlike public and private chains, enterprise-class blockchains directly address the needs of enterprise-class applications and have more stringent requirements on the security, flexibility, and performance of blockchain systems. The Hyperchain enterprise blockchain platform has been designed to implement a flexible, efficient, and stable consensus algorithm RBFT based on the optimization of traditional PBFT. In support of smart contracts, the Solidity language supporting the active open-source arena has been selected to optimize its execution of virtual machines at the system level and strengthen the security level of the enterprise blockchain through the transaction, transaction link, application development package, and other multi-level encryption processing. Hyperchain has also designed and implemented an enterprise blockchain control platform that supports system monitoring, contract writing, contract compilation, and other functions.

Chapter 7 focuses on the development of applications on the Hyperchain blockchain. Firstly, it shows the primary interfaces provided by the Hyperchain platform in terms of transaction invocation, contract management, and block query. Secondly, it presents how to build a working enterprise blockchain system, Hyperchain, in terms of configuration, deployment, and operation of Hyperchain clusters; finally, it introduces how to develop smart contract applications on the Hyperchain platform, using a mock bank as an example.

The fourth part includes several case studies of practical blockchain applications, each of which has project introduction, system function analysis, overall system design, smart contract design, system implementation and deployment, etc. with

detailed analysis on the development process and key code, and the full source code. This part contains three chapters.

Chapter 8 introduces two real-life projects based on Ethereum, namely the General Points System and the e-Coupon System. Based on the Ethereum basics and development technologies learned in the previous chapters, readers can compare the contents of this chapter and practice step by step to better understand the relevant concepts and technologies in the process of practical exercises, laying a solid foundation for building their blockchain application projects established on Ethereum.

Chapter 9 includes two actual project cases based on Hyperledger Fabric, namely social heritage management platform and high-end food security system. Based on the basic knowledge and development techniques of Hyperledger Fabric learned in the previous chapters, readers can put the content of this chapter into practice while learning, so as to accumulate practical experience for Hyperledger Fabric blockchain application development.

Chapter 10 introduces two enterprise blockchain application projects based on Hyperchain, namely accounts receivable management system and taxi-hailing platform. It can be seen that Hyperchain can be applied to build blockchain applications with complete functions, leading technologies, and requirements at enterprise level. Readers can learn and practice blockchain application development in depth by referring to the content in this chapter and using the comprehensive development interface provided by Hyperchain.

Example Code and Errata

All the sample code in Part IV has been uploaded to the Turing community homepage at ituring.cn/book/2805, where the Ethereum-Score-Hella project is an update of the Ethereum-Score project under Truffle 4.1.11.

Acknowledgments

As blockchain technicians, we feel honored to write a technical and practical book on blockchain. We would like to express our sincere gratitude to all those who have provided us with guidance, support, and encouragement.

We are grateful to the School of Computer Science and Technology and the School of Software of Zhejiang University for providing us with excellent conditions, various facilities, academician Chun Chen, and researcher Xiaohu Yang for their continuous care and support, which made it possible to complete this book successfully.

Thanks to all the Hangzhou Hyperchain Technology Co., LTD. staff, especially Dr. Wei Li; Postdoctoral Dr. Weiwei Qiu; and Dr. Keting Yin, for their tremendous support in bringing this book to fruition.

Thanks to Xiaoyi Wang, Fanglei Huang, Jialei Rong, Yufeng Chen, Faxiang Wu, Chen Wu, Wei Hu, Jiajin Song, Wei Guo, Chao Li, Bo Lin, Jinming Ou, Xin Zhou, Fei Chen, Lina Fei, Xiaoyang Chen, Ligang Sun, Qian Zhao, Jiafei Gu, Gangzhe Shi, Qisheng Yin, Qianwei Zhang, Zhenhao He, et al. for their contributions to the compilation of the manuscript. We are grateful to Yaojing Liu, Maifang Hu, Haizhen Zhuo, Weiwei Zhong, Qi Sun, Ke Zhao, and Zhisheng Huang for their time and sweat proofreading the manuscript.

Thanks to Zhoudong Ji, Deputy General Manager of the Advanced Technology Research Centre of Wanda Network Technology Group, and Menghang Zhang, a senior researcher in blockchain, for their helpful contributions to Chaps. 4 and 5 of this book.

Contents

Part I Blockchain Basics

1	Introduction to Blockchain Basics	3
1.1	Blockchain Basics	3
1.1.1	From Bitcoin to Blockchain	4
1.1.2	Definition of Blockchain	5
1.1.3	Blockchain-Related Concepts	6
1.1.4	Classification of Blockchain	8
1.2	Blockchain Development History	11
1.2.1	Technology Origins	11
1.2.2	Blockchain 1.0: “Digital Currency”	12
1.2.3	Blockchain 2.0: Smart Contracts	13
1.2.4	Blockchain 3.0: Beyond “Currency,” Economy, and Market	14
1.3	Key Technologies of Blockchain	14
1.3.1	Basic Model	14
1.3.2	Data Layer	15
1.3.3	Network Layer	21
1.3.4	Consensus Layer	23
1.3.5	Incentive Layer	26
1.3.6	Contract Layer	27
1.4	Status Quo of Blockchain Industry	29
1.4.1	Blockchain Development Trends	29
1.4.2	Government-Driven Blockchain	30
1.4.3	Ecological Mapping of Blockchain	33
1.5	Blockchain Application Scenarios	35
1.5.1	Digital Bills	35
1.5.2	Supply Chain Finance	36
1.5.3	Accounts Receivable	36
1.5.4	Data Trading	37
1.5.5	Bond Trading	37

1.5.6	Block Trading	38
1.5.7	Cross-Border Payment	38
1.5.8	Other Scenarios	39
1.6	Mainstream Blockchain Platforms	40
1.7	Summary	43

Part II Open Source Blockchain Platforms

2	In-Depth Interpretation of Ethereum	47
2.1	Basics Introduction to Ethereum	48
2.1.1	History of Ethereum Development	48
2.1.2	Basic Ethereum Concepts	49
2.1.3	The Ethereum Client	52
2.1.4	Ethereum Account Management	55
2.1.5	Ethereum Network	57
2.2	The Core Principle of Ethereum	58
2.2.1	Ethereum Consensus Mechanism	59
2.2.2	Ethereum Virtual Machine	62
2.2.3	Ethereum Data Storage	64
2.2.4	Ethereum Encryption Algorithm	65
2.3	Ethereum Smart Contracts	67
2.3.1	Introduction to Smart Contracts and Solidity	67
2.3.2	Smart Contract Programming and Deployment	70
2.3.3	Smart Contract Testing and Execution	83
2.3.4	Smart Contract Example Analysis	89
2.4	History, Problems, and Future Development of Ethereum	92
2.4.1	Historical Events	93
2.4.2	Current Problems of Ethereum	95
2.4.3	Ethereum 2.0	97
2.5	Summary	99
3	Fundamentals of Ethereum: Application Development	101
3.1	Building an Ethereum Development Environment	101
3.1.1	Configuring the Ethereum Environment	101
3.1.2	Building an Ethereum Private Chain	104
3.2	Ethereum Remix IDE	110
3.2.1	Compiling Smart Contracts	110
3.2.2	Obtaining Bytecode and ABI Files	112
3.2.3	Testing Contract Methods	115
3.3	Ethereum Programming Interface	116
3.3.1	jSON-RPC	116
3.3.2	JavaScript API	122
3.4	DApp Development Framework and Process	125
3.4.1	Meteor	126
3.4.2	Truffle	129

3.4.3	Layered Extensible Development Process	134
3.5	The First Ethereum Application	137
3.5.1	Optimizing the MetaCoin Application	137
3.5.2	MetaCoin Code in Detail	139
3.6	Deploying to the Ethereum Public Chain (Mainnet)	144
3.6.1	Infura	145
3.6.2	Project Configuration	147
3.6.3	Deploying MetaCoin	147
3.7	Summary	148
4	In-Depth Interpretation of Hyperledger Fabric	149
4.1	Project Introduction	149
4.1.1	Project Background	150
4.1.2	Project Introduction	151
4.2	Introduction to Fabric	154
4.3	Core Concepts	155
4.3.1	Anchor Nodes	155
4.3.2	Access Control List	155
4.3.3	Blocks	155
4.3.4	Blockchain	156
4.3.5	Smart Contracts	156
4.3.6	Channels	156
4.3.7	Submit	156
4.3.8	Concurrent Control Version Checking	156
4.3.9	Configuration Blocks	157
4.3.10	Consensus	157
4.3.11	Consent Setting	157
4.3.12	Consortium	157
4.3.13	World State	158
4.3.14	Dynamic Membership Management	158
4.3.15	Genesis Blocks	158
4.3.16	Gossip Protocol	158
4.3.17	The Ledger	158
4.3.18	Followers	159
4.3.19	Leaders	159
4.3.20	Principal Peer Node	159
4.3.21	Logging	159
4.3.22	Membership Service Provider	160
4.3.23	Membership Management Service	160
4.3.24	Sorting Service or Ordering Service	160
4.3.25	Organization	160
4.3.26	Node	160
4.3.27	Policy	161
4.3.28	Private Data	161
4.3.29	Private Data Set	161

4.3.30	Raft	161
4.4	Architecture Details	161
4.4.1	Architecture Interpretation	162
4.4.2	Membership Services	164
4.4.3	Blockchain Services	169
4.4.4	Chaincode Services	173
4.5	Chaincode Analysis	174
4.5.1	Chaincode Overview	174
4.5.2	Chaincode Structure	175
4.5.3	CLI Command Line Calls	178
4.5.4	Chaincode Execution Swimlane Diagram	179
4.6	Transaction Flow	180
4.6.1	General Process	181
4.6.2	Process Details	183
4.6.3	Endorsement Strategies	187
4.6.4	Verifying the Ledger and PeerLedger Checkpoints	188
4.7	Summary	189
5	Application Development: Fundamentals of Hyperledger Fabric	191
5.1	Environment Deployment	191
5.1.1	Software Download and Installation	191
5.1.2	Setting Up Development Environment	194
5.1.3	Go and Docker	195
5.2	Chaincode Development Guide	200
5.2.1	Interface Introduction	200
5.2.2	Case Study	201
5.2.3	Introduction to Private Data	205
5.3	CLI Application Examples	207
5.3.1	Preparation	208
5.3.2	Writing the Code	209
5.3.3	Initiate Network and Chaincode Calls	215
5.3.4	Switching on the Network Manually	216
5.4	SDK Application Examples	220
5.4.1	Introduction to the SDK	220
5.4.2	SDK Application Development	221
5.5	Summary	226
Part III The Enterprise-Level Blockchain Platform Hyperchain		
6	Anatomy of the Core: Principles of Enterprise-Level Blockchain Platform	229
6.1	Hyperchain Overall Architecture	230
6.2	Basic Components	233
6.2.1	Consensual Algorithm	233
6.2.2	Network Communication	240

6.2.3	Smart Contracts	242
6.2.4	Ledger Data Storage Mechanism	247
6.3	Expand Components	253
6.3.1	Privacy Protection	254
6.3.2	Encryption Mechanism	256
6.3.3	Member Management	260
6.3.4	Blockchain Governance	264
6.3.5	Message Subscription	266
6.3.6	Data Management	268
6.3.7	Verification Based on Hardware Acceleration	270
6.4	Summary	271
7	Hyperchain Application Development Fundamentals	273
7.1	Platform Functionality	273
7.1.1	Platform Interaction	274
7.1.2	Transaction Calls	275
7.1.3	Contract Management	278
7.1.4	Block Queries	282
7.2	Platform Deployment	285
7.2.1	Hyperchain Configuration	286
7.2.2	Hyperchain Deployment	287
7.2.3	Hyperchain Operation	289
7.3	The First Hyperchain Application	290
7.3.1	Writing Smart Contracts	290
7.3.2	Deployment and Contract Invocation	291
7.4	Summary	292
Part IV Blockchain Applications and Use Cases		
8	Ethereum Application and Case Studies	295
8.1	Case Study of Ethereum-Based Generic Points System	295
8.1.1	Project Introduction	296
8.1.2	System Function Analysis	297
8.1.3	General System Design	297
8.1.4	Smart Contract Design	300
8.1.5	System Implementation	309
8.1.6	System Deployment	317
8.2	Case Study of an Ethereum-Based E-Coupon System	321
8.2.1	Project Introduction	322
8.2.2	System Functional Analysis	323
8.2.3	General System Design	323
8.2.4	Smart Contract Design	326
8.2.5	System Implementation and Deployment	336
8.3	Summary	341

9	Hyperledger Fabric Application Case Studies in Detail	343
9.1	Case Study of a Fabric-Based Social Culture Heritage Management Platform	343
9.1.1	Project's Background Analysis	344
9.1.2	System Functional Analysis	345
9.1.3	General System Design	346
9.1.4	General Design of Smart Contracts	347
9.1.5	Core Functional Contract Design	348
9.1.6	Tool Contract Design	351
9.1.7	Deployment Implementation	352
9.2	Case Study of a High-End Fabric-Based Food Safety System	354
9.2.1	Background Analysis	355
9.2.2	Solution Proposal	355
9.2.3	System Functional Analysis	356
9.2.4	General System Design	357
9.2.5	API Design	358
9.2.6	Smart Contract Design	360
9.2.7	Leveraging the Node.js SDK	375
9.2.8	Deployment Implementation	378
9.3	Summary	382
10	Use Cases and Detailed Explanation of Enterprise-Level Blockchain	383
10.1	Case Study of Hyperchain-Based Accounts Receivable Management System	383
10.1.1	Project Introduction	384
10.1.2	System Functional Analysis	385
10.1.3	General System Design	387
10.1.4	Smart Contract Design	392
10.1.5	System Security Design	395
10.2	Case Study of Hyperchain-Based Ride-Hailing Platform	396
10.2.1	Project Introduction	396
10.2.2	System Functional Analysis	397
10.2.3	General System Design	401
10.2.4	Smart Contract Design	403
10.2.5	System Implementation and Deployment	420
10.3	Summary	422

Part I

Blockchain Basics

Chapter 1

Introduction to Blockchain Basics



The blockchain, invented by a person (or group of people) using the name Satoshi Nakamoto in 2008, serves as the public transaction ledger of bitcoin. Its decentralization, openness, and information that cannot be easily altered or tampered with are likely to have a disruptive impact on a range of industries, including finance and services. In January 2016, the PBOC (PBOC) held a seminar on digital currencies in Beijing to discuss the feasibility of issuing virtual digital currencies using blockchain technology. Although the pilot digital currency did not fully embrace blockchain technology, the “blockchain,” a somewhat mysterious term, suddenly became a hot topic, followed by the rapid rise of blockchain technology in China, and more and more startups and related research groups were established one after another. It has driven the rapid development of blockchain technology, making it one of the most revolutionary emerging technologies in recent years, and was even considered as the fifth disruptive computing paradigm after mainframe, personal computer, the Internet, and mobile/social networking. It has also been hailed as the fourth credit milestone in the history of human credit evolution, after blood relative credit, precious metal credit, and paper money credit.

This chapter will provide a comprehensive analysis of blockchain technology, from the basics, history, key technologies, industry status quo, scenario models, and mainstream platforms, so that readers can have a holistic and intuitive understanding of blockchain technology, laying a good foundation for the advancement and practice of blockchain technology.

1.1 Blockchain Basics

Learning a new technology must start with understanding its basic concepts. This section begins with a discussion of bitcoin, which leads into the concept of blockchain technology, and then introduces the basics necessary to get started with

blockchain technology, such as the definition, related basic concepts, and classification of blockchain.

1.1.1 *From Bitcoin to Blockchain*

When talking about blockchain technology, people tend to associate it with Bitcoin first, as it originally emerged as the underlying framework technology for Bitcoin. Therefore, before we explore blockchain technology, let us take a brief look at the origin of blockchain—Bitcoin.

As early as the 1980s, the exploration of “digital currency” was already underway. But it was not until the emergence of Bitcoin that the idea of a “digital cryptocurrency” became a reality, and “digital currency” and its derivatives began to develop rapidly. Bitcoin is the first application of blockchain technology, and it is by far the largest and most widely used blockchain application. In November 2008, in a paper entitled *Bitcoin: A Peer-to-Peer Electronic Cash System*, an author under the pseudonym Satoshi Nakamoto described a method for building a new, decentralized system of peer-to-peer transactions, and put the ideas he presented in his paper into practice by embarking on the development of Bitcoin-related features. Bitcoin was first launched on January 3, 2009, with the creation of the genesis block. On January 12, 2009, Satoshi Nakamoto sent 10 bitcoins to cryptographer Hal Finney. It was the first-ever Bitcoin transaction completed since the Bitcoin system went live. Despite the controversy, Bitcoin is a remarkable innovation in the history of “digital money” from a technical standpoint.

Compared to traditional currencies and “digital currencies” that preceded Bitcoin, Bitcoin’s biggest difference is that it does not rely upon any centralized institution but on mathematical principles such as encryption and consensus algorithms. The fact that people no longer need to consume additional resources due to trust issues has led to a lot of interest in Bitcoin and blockchain technology.

Bitcoin, as a “virtual digital currency” based on blockchain technology, is designed to solve the following problems with previous “digital currencies”:

- The monetary authority manages and controls the issue of the currency and related policies, which can decide everything;
- No previous “digital currency” can be traded anonymously;
- The value of the “virtual currency” itself cannot be fully guaranteed;
- The currency held is not completely secure for holders.

The current banking system, as a third-party institution for currency, can indeed solve the above problems at a cost. But if the scope of transactions is extended to a global scale, what bank can ensure that it can be trusted globally? Thus, it is asked whether it is possible to design a distributed database system that is globally accessible and completely neutral, impartial, and secure. Many researchers tried to explore and come up with some solutions, but they failed to be really accepted by the

society for various reasons. Yet blockchain technology has done so by becoming a form of distributed ledger technology.

Since 2014, it has been discovered that blockchain, the underlying support technology of Bitcoin, has great potential for broader application, and therefore has officially triggered a wave of innovation in Distributed Ledger technology. As explorers continue to innovate, blockchain technology has been born out of Bitcoin and is gradually emerging in many fields such as finance, trade, logistics, credit collection, Internet of Things, and sharing economy.

1.1.2 Definition of Blockchain

Blockchain technology, essentially defined as a decentralized database, is the core technology and infrastructure of Bitcoin. It is a new application model of computer technology such as distributed data storage, point-to-point transmission, consensus mechanism, and cryptographic algorithms. Blockchain, in a narrow sense, is a chained data structure that combines data blocks in a sequential manner and a chronological order. It also can be viewed as a tamper-proof and unforgeable distributed ledger guaranteed by cryptography. In a broad sense, the blockchain technology is a new distributed infrastructure and computing paradigm that uses the blockchain data structure to verify and store data, distributed node consensus algorithms to generate and update data, cryptography to ensure the security of data transmission and access, and smart contracts formed by automated scripts to implement programming and operate data.

By storing data across its peer-to-peer network, blockchain can effectively transfer value between nodes that lack trust. Compared with existing database technologies, blockchain has the following technical features.

Blockchain Data Structure

Blockchain utilizes a blockchain data structure to verify and store data. It is clear from the previous introduction to the basic concepts that each block is packaged to record the transactions that occurred over a period of time, is a consensus on the current ledger, and is associated by recording the hash of the previous block, resulting in a blockchain style data structure.

Distributed Consensus Algorithm

Blockchain system, using distributed consensus algorithms to generate and update data, technically eliminates the possibility of illegal data tampering thus replacing the third-party intermediaries that guarantee trust and transaction security in

traditional applications, and reducing the time cost, human cost, and resource consumption caused by credit maintenance.

Cryptography

Blockchain system ensures the security of data transmission and access by cryptography. The transaction information stored on the blockchain is public, but the account's identity is highly encrypted. The blockchain system integrates the advantages of symmetric encryption, asymmetric encryption, and hashing algorithms, and adopts digital signature technology to secure transactions.

The technical features above determine that blockchain applications have the following functional characteristics.

1. Multicenter

Unlike the centralized data management of traditional applications, there are multiple institutions in the blockchain network to monitor each other and reconcile in real time thus reducing the risk of single bookkeeping fraud and enhancing data security.

2. Automation

A smart contract is a self-enforcing agreement embedded in the computer code managed by a blockchain. The code contains a set of rules under which the parties of that smart contract agree to interact with each other. It greatly enhances the automation of economic activities and contracts.

3. Trustworthy

The transaction records and other data stored on the blockchain are tamper-resistant and traceable, and it can be a good solution to the problem of distrust without the need for a third-party trusted intermediary.

4. Openness

Typically, in a blockchain, each node has a ledger shared across a network, with many developed as open-source blockchain projects. In addition to the industry-related private data which will be encrypted, the data information of the blockchain is completely open and transparent.

1.1.3 *Blockchain-Related Concepts*

Blockchain maintains a distributed ledger that cannot be tampered with or falsified in a cryptographic way and addresses the issue of consistency in decentralized accounting system by consensus-based specifications and protocols (consensus mechanism). There are three main concepts as follows:

- Transaction: each operation on the blockchain that causes a change in the state of the block is called a transaction, and each transaction corresponds to a unique transaction hash, which is then packaged over a period.

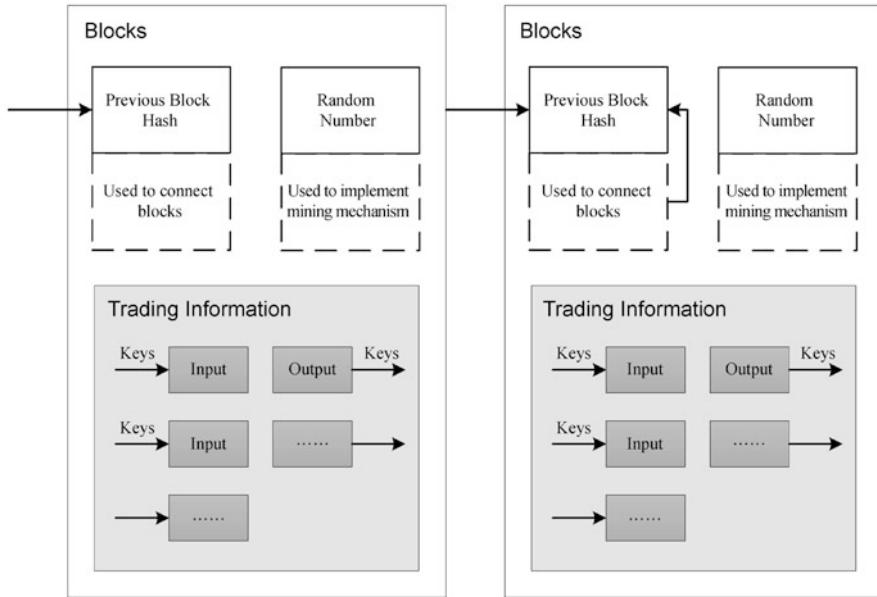


Fig. 1.1 Main structure of blockchain

- Block: packing records of transactions and states that occur over a period of time are a consensus on the current ledger. Each block is added to the chain at a relatively smooth interval, and the consensus time can be dynamically set in an enterprise-class blockchain-based application.
- Chain: the blocks are concatenated in chronological order. It is a log record of the entire state change by each block recording the association of the hash value of the previous block.

The main structure of the blockchain shown in Fig. 1.1 can help you understand these concepts.

In the blockchain technology system, the trustworthiness and security of transactions are not guaranteed by some authoritative centralized institution, but by encryption and distributed consensus mechanisms. There are four major technological innovations of blockchain:

Distributed Ledger

Transactions are recorded jointly by multiple nodes in a distributed system. Each node records a complete record of transactions, so they can all participate in overseeing the legitimacy of transactions and verifying the validity of transactions. Unlike traditional centralized technology solutions, no single node in the blockchain has the authority to record transactions individually, which avoids the possibility of

forgery due to a single bookkeeper or node being controlled. In addition, since the nodes of the entire network participate in the recording, in theory, unless all nodes are destroyed, otherwise the transaction record will not be lost thus ensuring the security of the data.

Cryptography and Authorization Technology

Blockchain technology integrates well many advantages of current symmetric encryption, asymmetric encryption and hashing algorithms, and secure transactions by using digital signature techniques. The most representative of these is the use of elliptic curve encryption algorithms to generate users' public and private key pairs and that of elliptic curve digital signature algorithms to secure transactions. The transaction information packaged on the block is public to all nodes participating in the consensus, but the account identity information is strictly encrypted.

Consensus Mechanism

Consensus mechanism is the strategy and method that each node reaches agreement on in the blockchain. The consensus mechanism of blockchain replaces the third-party central organization that guarantees the trust and transaction security in the traditional application, and can reduce the third-party credit cost, time cost, and capital consumption caused by the parties' distrust. Commonly used consensus mechanisms include PoW, PoS, DPoS, Paxos, Raft, and PBFT, a way to write data and a means to prevent tampering.

Smart Contract

A smart contract is a computer program which is intended to automatically execute relevant events and actions according to the terms of a contract or an agreement and is a system participant in its own right. It enables the storage, delivery, control, and management of value, providing an innovative solution for blockchain-based applications.

1.1.4 Classification of Blockchain

Blockchain networks can be classified into public, consortium, and private blockchains according to the way of node participation. Blockchain technology can be divided into permissioned blockchain and permissionless blockchain according to different permissions. Among the three major types of blockchain

technologies mentioned above, consortium chain and private chain belong to permissioned chain, and public chain belongs to permissionless chain.

Public Chain

A public chain, as its name implies, is a public blockchain in which everyone can act as a node in the network without the need for anyone to give permission or authorization. In the public chain, each node is free to join or leave the network, and can participate in reading and writing data on the chain, executing transactions, and also participate in the process of reaching consensus in the network, i.e., deciding which block can be added to the main chain and recording the current network status. The public chain is a fully decentralized blockchain, which ensures the security of on-chain transactions with the help of encryption algorithms in cryptography. When consensus algorithms are adopted to reach consensus, public chains mainly adopt consensus algorithms such as proof-of-work (PoW) mechanism, proof of stake (PoS) mechanism, and delegated proof of stake (DPoS) mechanism, which combine economic rewards and cryptographic digital verification to achieve the purpose of decentralization and network-wide consensus. In the process of forming such algorithmic consensus, each node can contribute to the consensus process, commonly known as “mining,” in return for a financial reward proportional to the contribution, namely “digital tokens” issued in the system.

Public chains, also known as public blockchains, are unlicensed chains that do not require permission to join or exit freely. The most typical applications include Bitcoin and Ethereum. Because of its completely decentralized and public-oriented nature, public chains are often used for “virtual cryptocurrencies” and some public-oriented financial services and e-commerce.

Consortium Chain

Consortium Blockchain is not completely decentralized, but a polycentric or partially decentralized blockchain. When a blockchain system is running, its consensus process may be controlled by designated nodes. For example, in a blockchain system with 15 financial institutions, each institution acts as a node on the chain, and for every transaction confirmed, at least 10 nodes need to be confirmed (2/3) before the transaction or block can be recognized. The data on the ledger of the consortium blockchain is different from that of the public blockchain. Only the consortium member nodes can access it, and the operations on the chain, such as read and write privileges and participation in bookkeeping rules, also need to be decided by the consortium member nodes together. Since participants in the consortium chain scenario form an alliance, the number of nodes involved in the consensus is much less than that of the public chain, and generally targeted at a certain business scenario, the consensus protocol generally does not adopt the mining mechanism similar to the PoW, and does not necessarily adopt tokens as an incentive

mechanism, but uses consensus algorithms, such as PBFT and Raft, suitable for polycentric and highly efficient. At the same time, the consortium chain has a great difference from the public blockchain in terms of transaction time, status, and transaction number per second, so it has higher security and performance requirements than the public blockchain.

Unlike public blockchains where anyone is free to join, it is not a public platform rather a permissioned platform that requires a certain permission license to join as a new node. Typical representatives now include the Hyperledger project supported by the Linux Foundation, Corda developed by the R3 Blockchain Alliance, and the Hyperchain platform launched by Chinese blockchain firm FunChain Technology.

Private Chain

In a private blockchain, permission to write data is controlled by one organization, while read permissions can be open or restricted according to circumstances. Therefore, the application scenario of a private chain is generally the management of branch offices by the head office within the enterprise, such as database management and auditing. Compared with public and federated chains, the value of a private chain is mainly reflected in that it can provide a secure, traceable, and not easily tampered platform, and can prevent security attacks from both internal and external sources. There is some controversy around private chains now. Some people think that private chain is of little significance, because it has to rely on third-party blockchain platform institutions, and all permissions are controlled by a node, which has violated the original intention of blockchain technology. It is not blockchain technology, but an existing distributed ledger technology. However, others believe that private chain has great potential value because it can provide a good solution to many existing problems, such as the compliance of internal rules and regulations of enterprises, anti-money laundering behavior of financial institutions, budget, and execution of government departments, and so on.

Like federated chains, private chains are also a type of licensing chain, although the licensing rights are held in a single node. In some scenarios, private chains are also called proprietary chains. There are not many applications of private chains today, and pioneers are still trying to explore them. The main applications that exist today are the multichain platform launched by UK-based Coin Sciences Ltd., which aims to help enterprises rapidly deploy private chain environments that provide good privacy protection and permission control.

Since its birth, blockchain technology has undergone three major technological evolutions. The typical representative platforms are Bitcoin in 2009, Ethereum in 2013, and Fabric and Hyperchain in 2015. Its organizational form has changed from a public blockchain with serious resource consumption, low transaction performance, and lack of flexible control mechanism to a consortium blockchain with efficient consensus, intelligent programmability, and privacy protection. Currently, the TPS (transaction processing per second) of Hyperchain platform has reached thousands or even tens of thousands, which can meet the needs of most business

scenarios. In the future, with the further development of technology, blockchain business applications based on federated chains will become the main form of blockchain applications.

1.2 Blockchain Development History

The distributed system based on zero-trust foundation and truly decentralized by Bitcoin is actually a solution to the Byzantine Generals' problem first theorized by the mathematicians Leslie Lamport and others more than three decades ago. The development history of blockchain technology since its birth can be roughly divided into four stages: technology origin, blockchain 1.0, blockchain 2.0, and blockchain 3.0, as shown in Fig. 1.2.

1.2.1 Technology Origins

Blockchain technology originated from Bitcoin created by Satoshi Nakamoto. Bitcoin was designed by Satoshi Nakamoto, standing on the shoulders of giants, based on various related technologies and algorithms of his predecessors, combined with his own unique creative thinking. The following is a brief history of blockchain-related basic technologies development.

Blockchain adopted PoW, a consensus mechanism to update and share transactions, to solve the Byzantine Generals problem proposed by Leslie Lamport and others in 1982. It is a well-known fault-tolerant problem in distributed computing, i.e., to ensure consensus of normal nodes in a distributed system with faulty nodes and misinformation, and thus to guarantee consistency of message transmission. In 1990, Leslie Lamport proposed the Paxos algorithm, which achieved highly fault-tolerant network-wide consistency in distributed systems but it did not consider the Byzantine General problem. Later, Barbara Liskov's Practical Byzantine-fault-tolerance (PBFT) algorithm in 1999 improved the Paxos algorithm to deal with the Byzantine Generals problem.

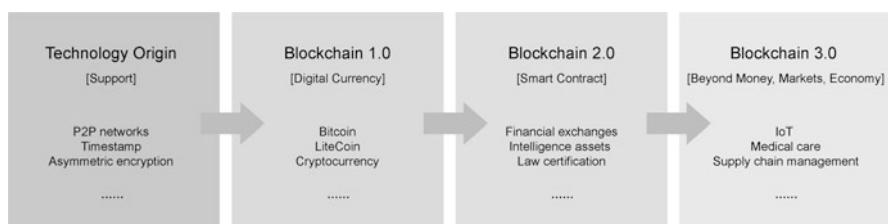


Fig. 1.2 Blockchain development history

The PoW mechanism originated from the proof-of-work idea first developed by Cynthia Dwork in 1993 and was initially widely used for filtering spam. In 1997, Adam Back invented Hashcash (a proof-of-work algorithm), which took advantage of the irreversibility of the cost function and was difficult to crack but easy to verify. Its algorithm design concept has become one of the core technologies for bitcoin blockchain nodes to reach consensus after being improved by Satoshi Nakamoto, and it is the cornerstone of Bitcoin that can prevent forged transactions. In 1998, developer Wei Dai proposed an “anonymous and distributed electronic money system” called B-money by PoW mechanism, the first decentralized “electronic cryptocurrency.” Many of the ideas of the Bitcoin blockchain were borrowed from B-money. In 1999, Markus Jakobsson and Ari Juels formally published the concept of PoW, and in 2005, Hal Finney invented the reusable proofs of work (RPoW) mechanism, combining Wei Dai’s B-money system with the Hashcash algorithm invented by Adam Back to create a “digital cryptocurrency.” In 2008, Nakamoto published a paper on Bitcoin in a secrecy cryptography forum group, proposing a decentralized, anonymous payment method using PoW and timestamp mechanisms to create chained transaction blocks. The timestamp mechanism, first introduced by Stuart Haber and W. Scott Stornetta to secure electronic documents, was adopted by Satoshi Nakamoto in Bitcoin to trace transactions in the ledger back to their origin.

Elliptic Curve Digital Signature Algorithm or ECDSA, proposed by Scott Vanstone et al. in 1992, is a cryptographic algorithm used by blockchain technology to ensure the security of transactions in blockchain. In 1985, Neal Koblitz and Victor Miller independently codiscovered Elliptic Curve Cryptography (ECC), the first use of elliptic curves in cryptography. ECDSA was proposed on the basis of ECC. ECC, an alternative to RSA, is a public key encryption algorithm. But ECC can achieve the same security strength as RSA with a short and fast key, and is more difficult to break. ECC is becoming the preferred policy for network security and privacy. In terms of security and privacy, much of Bitcoin’s design and innovation draws on the cryptographic anonymous cash system eCash, pioneered by David Chaum in 1990 based on the untraceable cryptographic network payment system he proposed in 1982. While eCash is not a decentralized system, it is also an important milestone in the history of digital currencies.

From the history of technology development mentioned above, it is clear that blockchain technology does not happen overnight, but is an inevitable product of a certain background and technological development. The core technology of blockchain will be systematically introduced in detail in the following chapters.

1.2.2 *Blockchain 1.0: “Digital Currency”*

In the blockchain 1.0 stage, the application scope of blockchain technology mainly focused on the field of “digital currency.” As Bitcoin blockchain solves the Double-spending problem and the Byzantine General problem, it really clears the main obstacle to the circulation of “digital currency,” and many “copycat digital

currencies” begin to emerge in large numbers. The technical architecture of these “digital currencies” can be generally divided into three layers: blockchain layer, protocol layer, and “currency layer.” As the underlying technology of these “digital currency” systems, blockchain layer is regarded as the most basic part of the system, through which core functions like the consensus process and message passing are achieved. Protocol layer primarily provides some software services and makes rules for the system. Currency layer is primarily used as a value representation to pass value between users.

In the blockchain 1.0 stage, many decentralized digital payment systems were built based on blockchain technology, exerting a certain impact on the traditional financial system.

1.2.3 Blockchain 2.0: Smart Contracts

After the drawbacks of Bitcoin and other “fake coins” such as serious resource consumption and inability to handle complex logic were gradually exposed, the industry gradually shifted its focus to the blockchain, the underlying support technology of Bitcoin, resulting in modular, reusable, and automatically executable scripts running on the blockchain or smart contracts. This greatly expanded the scope of blockchain applications, and blockchain thus entered phase 2.0. The industry was slowly recognizing the immense value of blockchain technology. Blockchain technology began to break away from the innovation in the field of “digital currency,” and its application scope extended to financial transactions, securities clearing and settlement, identity authentication, and other business fields. Many new application scenarios have emerged, such as financial transactions, smart assets, file registration, and judicial authentication.

Ethereum, a representative platform of this phase, is a blockchain-based development platform that provides a Turing-complete smart contract system. Users can write their own smart contracts and build decentralized DAPPs with Ethereum. Based on the Turing-complete nature of Ethereum smart contracts, developers can program any decentralized application, such as voting, domain names, financial transactions, crowdfunding, intellectual property, and smart property. There are many decentralized applications running on the Ethereum platform, and according to its “white paper,” there are three main ones. The first is financial applications, including applications involving financial transactions and value transfer, such as “digital currency,” financial derivatives, hedging contracts, savings wallets, and wills. The second is semi-financial applications, which involve monetary participation but have a large nonmonetary aspect. The third is nonfinancial applications, such as online voting and decentralized autonomous organizations that do not involve money.

In the blockchain 2.0 phase, led by smart contracts, more and more financial institutions, startups, and research groups have joined the queue to explore blockchain technology, driving its rapid development.

1.2.4 Blockchain 3.0: Beyond “Currency,” Economy, and Market

With the continuous development of blockchain technology, the value of blockchain technology such as low-cost credit creation, distributed structure, and openness and transparency is gradually attracting the attention of the whole society, and new applications are emerging in various industries such as Internet of Things, medical care, supply chain management, and social welfare. The development of blockchain technology has entered the blockchain 3.0 stage. In this stage, the potential role of blockchain is not only reflected in the monetary, economic, and market aspects, but also extends to the political, humanitarian, social and scientific fields, and blockchain technology has already facilitated special groups to deal with real-world problems. As blockchain technology continues to develop, it can be boldly imagined that in the future it will extensively and profoundly change people's way of life, reconstruct the whole society, and recast credit value. Perhaps in the future, when blockchain technology develops to a certain extent, the whole society will enter the blockchain era, and every individual can serve as a node in the blockchain network. Using decentralized technology for the distribution of social resources, blockchain may become an ideal framework for advancing social and economic development.

1.3 Key Technologies of Blockchain

As the basics of blockchain and its development history were introduced earlier, it is believed that readers have already had a more intuitive understanding of blockchain. This section will further introduce the system architecture and key technologies of blockchain in depth.

1.3.1 Basic Model

The basic architecture of blockchain is charted in Fig. 1.3, drawn with reference to the blockchain architecture diagram in the *Status and Prospects of Blockchain Technology Development*. The basic blockchain architecture consists of five layers, namely, data layer, network layer, consensus layer, incentive layer, contract layer, and application layer.

- **Data layer** encapsulates the core blockchain technologies such as the chain structure, block data, and asymmetric encryption of the blockchain.
- **Network layer** provides point-to-point data communication and verification mechanism.

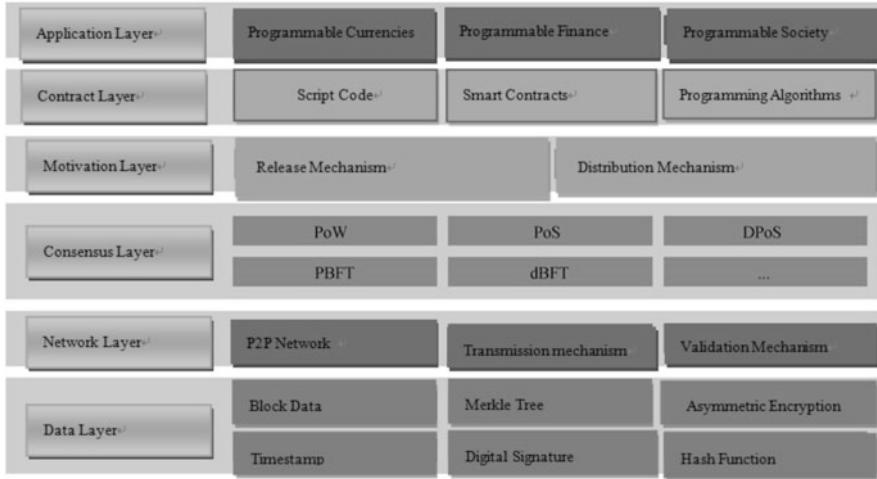


Fig. 1.3 Blockchain basic architecture

- **Consensus layer** mainly consists of various consensus algorithms for reaching consensus among network nodes.
- **Incentive layer** introduces economic factors into the blockchain technology system, mainly including the issuance mechanism and distribution mechanism of economic factors.
- **Contract layer** shows the programmability of the blockchain system and encapsulates various scripts, smart contracts, and algorithms.
- **Application layer** encapsulates the application scenarios and cases of blockchain technology.

In this architecture, the timestamp-based chain structure, distributed consensus mechanism among nodes, and programmable smart contracts are the most representative innovation points of blockchain technology. Smart contracts can generally be written or scripted at the contract layer to build blockchain-based decentralized applications. The technologies involved in each layer of this architecture are described in detail below.

1.3.2 Data Layer

The data layer is the core part of the blockchain, which is essentially a database technology and distributed shared ledger, a data structure consisting of blocks containing transaction information connected in an orderly manner from back to front. The technologies involved in this layer mainly include block structure, Merkle tree, asymmetric encryption, timestamp, digital signature, and hash function.

Timestamps and hash functions are relatively simple, and therefore will be the focus of the rest here.

Block Structure

Each block is generally composed of two parts: the block header and the block body. As shown in Fig. 1.4, the block header contains the parent block hash, timestamp, Merkle root, etc., while the block body contains all the transaction information in the block. In addition, each block corresponds to two values to identify the block: the block header hash and the block height.

Each block will have a block header hash, a 32-byte digital fingerprint obtained by a secondary hash calculation of the block header through the SHA256 algorithm. For example, the first block header hash of Bitcoin is 00000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f. The block header hash uniquely identifies a block on the blockchain and can be obtained by any node by performing a simple hash calculation on the block header. The block header hash is also included in the overall data structure of the block, but the data of the block header and of the block body are not necessarily stored together; for the sake of retrieval efficiency, they can be stored separately in the implementation.

In addition to identifying blocks by header hash values, blocks can also be identified by block height. For example, the block indexed by height 0 and preceding

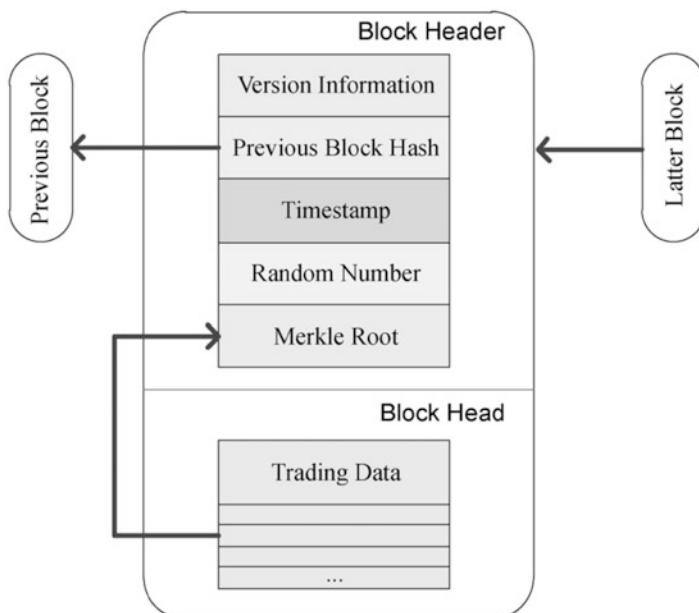


Fig. 1.4 Block structure

Table 1.1 Detailed structure of block header

Field	Size (bytes)	Description
Version	4	Version number for tracking software/protocol updates
Hash value of the previous block	32	Referencing the hash of the previous block in the blockchain
Merkle root	32	Hash of the Merkle tree root for transactions in this block
Timestamp	4	Approximate time to generate a Bitcoin block(UNIX timestamp accurate to the second)
Random number	4	Counters for PoW algorithms

000000000019 d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f is the first block. Unlike the header hash however the block height does not uniquely identify a block. Due to forks in the blockchain, there may be situations when two or more blocks have the same block height.

After the introduction of header hash values and block heights, the construction of the block header will be explained! Taking Bitcoin as an example, the block header has 80 bytes, and its detailed structure is shown in Table 1.1.

The block header consists of three groups of metadata. The first group is hash data that references the parent block used to connect to the previous block. The second group includes difficulty values, timestamps, and random numbers, all of which are relevant to mining competition. The third group is the Merkle root, the root node of the Merkle tree in the block body.

Merkle Tree

Now that the block header hash, block height, and the structure of the block header have been illustrated, the block body will be described. The block body stores transaction information, and in blocks they are stored as a Merkle tree, a data structure used to efficiently summarize all transactions in the block. A Merkle tree is a hash binary tree where each leaf node is a hash of a transaction. Similarly, taking Bitcoin as an example, in the Bitcoin network, Merkle trees are used to summarize all transactions in a block, while generating the digital fingerprint of the whole transaction set, i.e., Merkle roots, and providing an efficient way to verify the existence of a transaction in a block. A Merkle tree is constructed by recursively hashing pairs of nodes until there is only one hash, which is the Merkle root of all transactions in the block, stored in the block header structure described above.

The Merkle tree is further introduced with an example. Figure 1.5 is A Merkle tree with only four transactions, namely transactions A, B, C, and D.

The first step is to use SHA256 algorithm twice to hash the data of each transaction to get the hash value of each transaction. Here the four hashes of H_A , H_B , H_C , and H_D can be received, all of which are the leaves of the Merkle tree. For example,

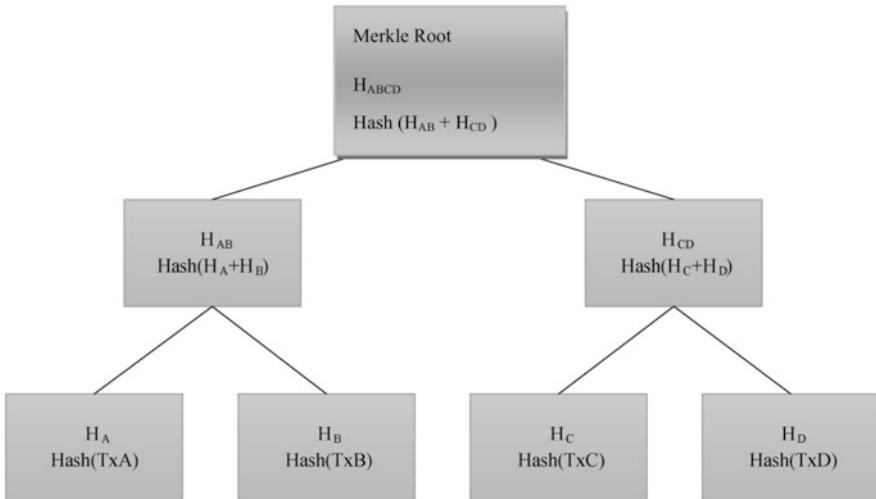


Fig. 1.5 Merkle tree

$$H_A = \text{SHA256}(\text{SHA256}(\text{transaction A}))$$

In the second step, a new hash value H_{AB} will be obtained by using SHA256 twice for the hash values of the two leaf nodes H_A and H_B , and another hash value H_{CD} gained by performing the same operation on H_C and H_D . For example,

$$H_{AB} = \text{SHA256}(\text{SHA256}(H_A + H_B))$$

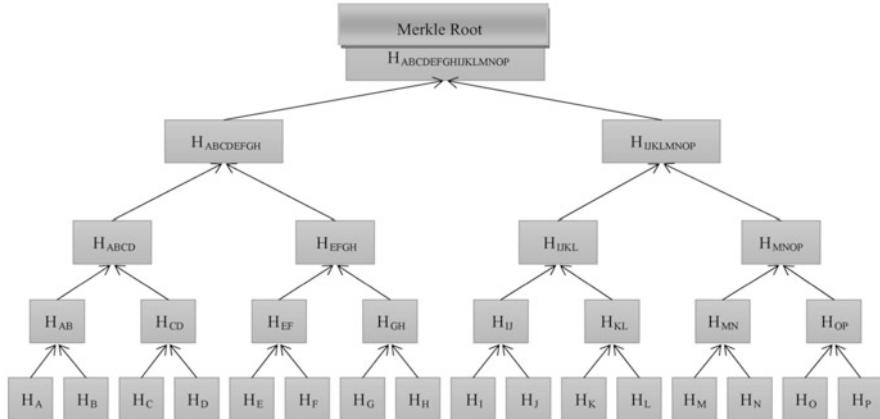
In the third step, the two existing hash values H_{AB} and H_{CD} will be combined in the second step, and finally a new hash value H_{ABCD} will be obtained.

$$H_{ABCD} = \text{SHA256}(\text{SHA256}(H_{AB} + H_{CD}))$$

At this point we have no other nodes of the same height, so the final H_{ABCD} is the Merkle root of this Merkle tree. The 32-byte hash value of this node is then written to the Merkle root field in the block header. The whole Merkle tree formation process is over.

Merkle trees are binary and therefore require an even number of leaf nodes, namely, even number of transactions. However, in many cases, there will be an odd number of transactions in a given block. For this case, the solution of the Merkle tree is to copy the last transaction and construct it as an even number of leaf nodes. Such a binary tree with an even number of leaf nodes is also called a balanced tree.

Figure 1.6 shows a larger Merkle tree consisting of 16 transactions. The illustration displays that whether there is one transaction or 100,000 transactions in a block, it can eventually be reduced into a 32-byte hash as the root node of the Merkle tree.

**Fig. 1.6** Merkle tree with multiple nodes**Table 1.2** Efficiency of the Merkle tree

Number of transactions (stroke)	Approximate size of block (kilobytes)	Path size (number of hashes)	Path size (bytes)
16	4	4	128
512	128	9	288
2048	512	11	352
65,535	16,384	16	512

When it is necessary to prove a particular transaction still exists in the transaction list, a node only needs to compute $\log_2 N$ 32-byte hashes to form a path from the root of the Merkle tree to a particular transaction, and the efficiency of the Merkle tree is shown in Table 1.2.

Asymmetric Encryption and Digital Signatures

Asymmetric encryption is a cryptographic technique used in blockchain technology for security requirements and ownership authentication. Common asymmetric encryption algorithms include RSA, Elgamal, Backpack algorithm, Rabin, D-H, ECC (Elliptic Curve Cryptography) and ECDSA (Elliptic Curve Digital Signature Algorithm), and so on. Unlike symmetric encryption algorithms, asymmetric encryption algorithms require two keys: a public key and a private key. The asymmetric encryption algorithm allows the two communicating parties to exchange information over insecure media and securely agree on the message. The public key is public and open to anyone in the system. The private key however is private, and it is not feasible to deduce the value of the private key from the public key. Each public key has a corresponding private key. If a message is encrypted with a public key, then the corresponding private key to decrypt the encrypted message is a must. And

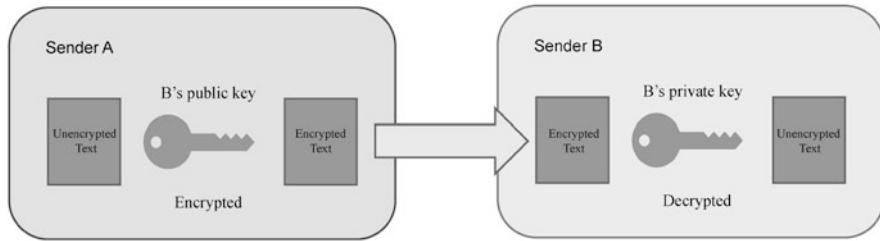


Fig. 1.7 Information encryption

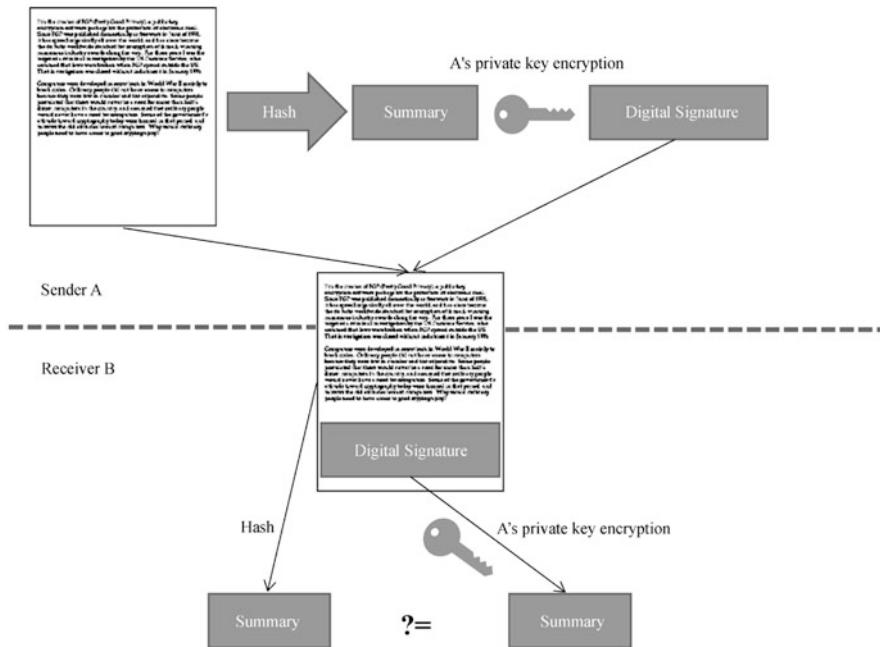


Fig. 1.8 Digital signature

if the information is encrypted with a private key, only the corresponding public key can be decrypted. In blockchain, asymmetric encryption is mainly used in scenarios such as message encryption and digital signature.

In the information encryption scenario (as shown in Fig. 1.7), the information sender A needs to send a message to the information receiver B, and the public key of B needs to encrypt the message first. After B receives it, it can decrypt this message with his private key, while others have no way to decrypt this encrypted message without the private key.

In the digital signature scenario (as shown in Fig. 1.8), sender A first generates a digest using a hash function on the original text, encrypts the digest with the private key to generate a digital signature, and then sends the digital signature together with

the original text to receiver B. After receiving the information, B uses A's public key to decrypt the digital signature to obtain A summary, and thus ensuring that the message is sent by A. Then the original received message uses the hash function to generate the digest and compares it with the digest obtained by decryption; if the same, it indicates that the received information has not been modified during transmission.

1.3.3 Network Layer

The network layer is the foundation of information transmission in the blockchain platform. Each node in the blockchain network can participate in consensus and bookkeeping equally in the way of P2P networking method, specific information dissemination protocol, and data verification mechanism. The P2P network architecture, information transmission mechanism, and data verification mechanism in the network layer of blockchain platform are introduced in detail below.

P2P Network Architecture

Blockchain network architecture typically adopts Internet-based P2P (peer-to-peer) architecture, in which each node of each computer is peer-to-peer, and they jointly provide services to the whole network. Moreover, without any centralized server, each host can either respond to requests as a server or use services provided by other nodes as a client. P2P communication does not require address verification from other entities or CAs thus effectively eliminating the possibility of tampering and third-party spoofing. Therefore, P2P networks are decentralized and open, which is right in line with the concept of blockchain technology.

In a blockchain network, all nodes are equal in status and are connected and interact with each other in a flat topology, and each node needs to undertake functions such as network routing, verifying block data, and propagating block data. In the Bitcoin network, two classes of nodes exist; light notes and full notes. A full node keeps a copy of every transaction on the network and needs to participate in verifying and recording the blockchain data in real time to update the blockchain master chain. A light node does not store the whole blockchain, but only a tiny part of it. Lightweight nodes request data from other neighboring nodes to verify transactions using a method called simplified payment verification (SPV).

Transmission Mechanism

After the new block data is generated, the node that generated the data will broadcast it to other nodes across the network for verification. The current blockchain underlying platforms generally redesign or improve the new transmission mechanism

based on the Bitcoin transmission mechanism according to their actual application requirements. For example, the Ethereum blockchain integrates the so-called “ghost protocol” to address the high block rejection rate and the ensuing security risks due to the fast block data confirmation. The steps of the transmission protocol of the Bitcoin system designed by Satoshi Nakamoto are listed here as an example.

1. The node in charge of the transaction will broadcast the new transaction data to all nodes in the whole network, and the node will store the data in a block after receiving it;
2. Each node finds a PoW that meets the difficulty requirements in the block based on its own computing power, and then broadcasts this block to all nodes in the whole network;
3. If all the transactions in the block are valid and have not occurred, then other nodes will agree and receive the data block. The node makes a new block at the end of the block to extend the chain, treating the random hash of the received block as the preorder block hash of the new block.

If the node associated with a transaction is a new node that is not connected to another node, the Bitcoin system will typically recommend a set of long-running “seed nodes” to the new node to establish a connection, or recommend at least one node to connect to the new node. In addition, broadcast transaction data does not need to be received by all nodes, as long as enough nodes respond, the transaction data can be integrated into the blockchain ledger. Nodes that do not receive the complete transaction data can request the missing transaction data from neighboring nodes.

Authentication Mechanism

In a blockchain network, all nodes are always listening to the transaction data and newly generated blocks broadcasted in the network. Upon receiving data from a neighboring node, the validity of the data is first verified. If the data is valid, a storage pool is established for the new data in the receiving order to store the data in a temporary area and continue to forward them to neighboring nodes. If the data is invalid, it is immediately discarded thus ensuring that the invalid data will not be propagated in the blockchain network. The validation method includes data structure, syntax normality, input and output, and digital signature based on predefined criteria. Similarly, after a node generates a new block, other nodes verify the PoW and time stamp of the new block according to predefined standards. If it is confirmed to be valid, the block will be linked to the main blockchain and start to fight for the right to accounting for the next block.

1.3.4 Consensus Layer

Leslie Lamport proposed the famous Byzantine General problem in 1982, which has triggered countless researchers to explore solutions. How to reach consensus efficiently in distributed systems is an important research topic in the field of distributed computing. The consensus layer of blockchain plays a role in enabling each node to reach consensus efficiently in the decentralized system with highly decentralized decision-making power by using different consensus algorithms in different application scenarios.

Initially, the Bitcoin blockchain adopted a PoW consensus mechanism that relied on node computing power to ensure the consistency of distributed bookkeeping in the Bitcoin network. As the blockchain technology improved, researchers successively proposed algorithms that could achieve the network-wide unanimity without relying too much on computing power, such as PoS consensus mechanism, DPoS consensus mechanism, Practical Byzantine Fault-Tolerant Algorithm, Raft consensus algorithm, and so on. The consensus algorithms will be briefly introduced as follows.

PoW (Proof-of-Work) Mechanism

The PoW mechanism was born out of the Hashcash system announced by Adam Back in 1997 and was originally created to prevent spam from flooding email systems. In 2009, Satoshi Nakamoto applied the PoW mechanism to the Bitcoin blockchain network as a consensus mechanism to achieve network-wide consistency. Strictly speaking, the use of a reusable Hashcash working proof in Bitcoin allows the generation of working proofs to be a probabilistic and random process. Each node on the network in this mechanism is computing a constantly changing hash of the block header using the SHA256 hash algorithm. The consensus says that the value must be equal to or less than a given value. In a distributed network, all participants need to use different random numbers to continuously compute the hash value until the goal is reached. When a node arrives at the exact value, all others must confirm with each other that the value is correct. After that, the transactions in the new block will be verified to prevent fraud. The set of transaction information used in the calculation is then recognized as the authentication result, represented by a new block in the blockchain. In Bitcoin, nodes that compute hash values are called “miners,” and the process of PoW is called “mining.” Since the calculation is a time-consuming process, incentives have been proposed (such as awarding miners a small portion of bitcoin). In a nutshell, PoW is the proof of the amount of work done, each block added to the chain must be verified by the consent of the network participants and the miner has done the corresponding amount of work on it. The advantages of PoW are its complete decentralization and distributed ledger. The disadvantages are obvious, namely resource consumption: mining causes a lot of waste of resources, and PoW takes a long time to reach consensus. The Bitcoin

network automatically adjusts the target value to ensure that the block generation process takes about 10 min, so it is not suitable for commercial use.

PoS (Proof of Stake) Mechanism

PoS was conceived by Nick Szabo as an energy-efficient alternative to PoW. Instead of requiring users to find a random number in an unrestricted space, it asked people to prove ownership of the amount of “currency” because it is believed that those with more “currency” are less likely to attack the network. Since selection based on the account balance is very unfair and a single richest person is bound to dominate the network, several solutions have since been proposed that combine equity to determine who will create the next block. Among them, Blackcoin uses random selection to predict the next creator, while Peercoin tends to choose based on coin age. Peercoin pioneered the real proof of ownership for the first time. It adopted the PoW mechanism to issue new coins and adopted the proof of ownership mechanism to maintain network security, also a pioneering initiative in the history of “virtual currency.” Unlike the Bitcoin network that requires proofs to perform a certain amount of work, this mechanism only requires proofs to provide ownership of a certain amount of “digital currency.” In the PoS mechanism, whenever a block is created, miners set up a transaction called a “coin right” that sends a certain percentage of coins to miners in advance. The PoS mechanism then algorithmically reduces the mining difficulty of the nodes in equal proportion to the proportion of tokens held by each node and over time, in order to speed up the nodes’ search for random numbers and shorten the time needed to reach consensus. Compared with PoW, PoS can save more energy and is more efficient. However, since the cost of mining is close to zero, it may be subject to attacks. PoS still essentially asks nodes in the network for performing mining operations, so it is equally difficult to be applied to the commercial sector.

DPoS (Proof of Share Authority) Mechanism

DPoS was invented by the Bitshares project team. Similar to the board of directors system in modern enterprises, DPoS refers to the token holders as shareholders, and the shareholders of blocks (who own shares) vote their representatives through democratic elections, who are responsible for the generation and data verification of new blocks. In order to become a representative and obtain the corresponding rights, the holder must first register with his or her blockchain public key to obtain a unique ID identifier of 32 bits in length, and shareholders will vote on the holder’s identifier in the form of a transaction. The top vote-getters will be selected as delegates. The representatives take turns producing blocks and share the proceeds of the exchange (i.e., fees). If there are representatives who do evil on the blockchain for their own interests, according to the nature of the distributed ledger, they will be easily found out by other shares and representatives and removed from the

blockchain system, resulting in vacancy that will be replaced by the coin holders with the next highest number of votes. DPoS dramatically reduces the number of nodes involved in block generation verification and bookkeeping, eliminating the need for nodes to expend additional workload and saving a significant amount of computing resources and consensus acceptance time. DPoS achieves a decentralized and centralized design that dramatically speeds up transactions on the blockchain.

PBFT (Practical Byzantine-Fault-Tolerance) Algorithm

Miguel and Barbara Liskov of Massachusetts Institute of Technology (MIT) originally published this algorithm in an academic paper. It was designed for a low-latency storage system, reducing the complexity of the algorithm, which can be applied to digital asset platforms with small throughput but a large number of events. It allows each node to publish a public key and any message passing through the node is signed by the node to verify its format. The verification process is divided into three stages: incubation, preparation, and implementation. A service operation will be valid if approval has been received from more than one-third of the different nodes. With PBFT, the blockchain network can contain f Byzantine malicious nodes out of N nodes, where $f = (N-1)/3$. In other words, PBFT ensures that at least $2f + 1$ nodes reach consensus before adding information to the distributed shared ledger. Many blockchain consortia at current stage such as Hyperledger Alliance and ChinaLedger Alliance are studying and verifying the practical deployment and application of this algorithm.

Raft (Channel Trustworthy) Consensus Algorithm

Raft is a more understandable consistency algorithm proposed by Stanford to reach a consistent consensus of all nodes under the premise that there are no evil nodes, but the network nodes may crash. In the Raft algorithm, each node owns one of the three identities: Follower, Candidate, and Leader. All nodes initially start out as a Follower, the node that wants to be the Leader will become the Candidate node and send request messages to other followers of the system for a vote. As long as more than half of the votes are collected, it regards themselves as Leader nodes, and Leader nodes rely on timing to send heartbeat data to all followers to maintain their position, otherwise the rest can compete for the Leader nodes again. When writing data, the Leader first writes the data temporarily to the local log and sends a request to the Follower to add data. When more than half of the add success messages returned, the Leader can write locally and send back the success result to the client, otherwise the write fails.

1.3.5 *Incentive Layer*

As a layer of introducing economic factors into blockchain technology, the necessity of the incentive layer depends on the specific application requirements established in blockchain technology.

This section takes the Bitcoin system as an example to introduce the incentive layer. In the Bitcoin system, a large number of node computing power resources are aggregated through the consensus process, so as to achieve data verification and bookkeeping of blockchain ledger. Therefore, it is essentially a task crowdsourcing process among consensus nodes. In a decentralized system, consensus nodes themselves are self-interested, and their fundamental purpose of participating in data verification and bookkeeping is to maximize their own benefits. Therefore, a reasonable incentive mechanism must be designed to make the individual behavior of consensus nodes to maximize their own benefits fit with the security and effectiveness of the blockchain system so that the large-scale nodes can form a stable consensus on the blockchain history.

Bitcoin adopts PoW consensus mechanism, where the economic incentive is composed of two parts: one is newly issued bitcoin; the other is the handling fee in the process of transaction circulation. The two are combined and rewarded to the nodes that successfully compute the required random number and generate a new block during the PoW consensus process. Therefore, bitcoin as a reward will have value only if nodes agree to work together to construct and maintain the blockchain history and the validity of their system.

Issuance Mechanism

In the Bitcoin system, the number of new blocks created and issued declines on a ladder over time. Starting with Genesis block, each new block will issue 50 bitcoins as a reward to the bookkeeper of that block, and every 4 years thereafter (210,000 blocks), the number of bitcoins issued per new block is reduced by half, and so on, until the number of bitcoins stabilizes at the maximum of 21 million. As mentioned earlier, the other part of the reward given to bookkeepers is the fee incurred during bitcoin transactions, currently 1/10,000 of a bitcoin by default. Both parts of the fee are encapsulated in the first transaction of the new block (called a Coinbase transaction). While the total fee for each new block is now much smaller compared to the number of new bitcoins issued, in the future fewer and fewer bitcoins will be issued, or even stopped, at which point the fee will become the primary driver for consensus node bookkeeping. The fee also serves as a security measure against “dust attacks” on the bitcoin system from a large number of microtransactions.

Distribution Mechanisms

As the Bitcoin mining ecosystem has matured, “mining pools” have come to the forefront of people’s minds. By joining a mining pool, many small-counterpower nodes join together and cooperate to increase the probability of gaining bookkeeping rights and share the newly issued bitcoins and transaction fees rewarded for generating new blocks. There are already 13 different allocation mechanisms, according to [Bitcoinminning.com](https://www.bitcoinminning.com). Nowadays, the mainstream mineral pools usually adopt PPLNS (Pay per last N Shares), PPS (Pay per Share), and PROP (PRO Portionately) mechanisms. In a mining pool, shares are divided proportionally according to the arithmetic power contributed by each node. The PPLNS mechanism distributes rewards to each cooperating node based on the actual share proportion it contributed within the last N shares after a new block is generated. PPS, on the other hand, estimates and pays a fixed theoretical return for each node directly based on the share proportion, and mining pools using this approach will charge a moderate fee to compensate for the risk of revenue uncertainty they bear for each node. The PROP mechanism distributes rewards proportionally based on the shares contributed by the nodes.

1.3.6 *Contract Layer*

The contract layer serves as the implementation of programming functions in a blockchain system, encapsulating various smart contracts, scripting algorithms, etc. As a global ledger blockchain system, Bitcoin itself has simple scripting programming capabilities. As a representative of Blockchain 2.0, the Ethereum platform has greatly enhanced programming languages that can theoretically be used to implement any functional application (i.e., DApp) using a specific scripting language. These applications allow Ethereum to be considered a “global computer” where anyone can use the consensus and consistency of the blockchain system to upload and execute arbitrary scripts and ensure that the programs run efficiently and securely without data leakage or malicious tampering. If the three layers of blockchain data, network interconnection, and consensus mechanism represent the “virtual machine” of the underlying blockchain structure (they represent data representation, data distribution, and data validation, respectively), then the contract layer is the basic component on which the blockchain can apply a series of advanced applications such as complex business logic and complex algorithms into action. While most “digital cryptocurrencies,” including Bitcoin, use simple script code that is not Turing-complete, this simple prototype that allows users to simply control the transaction process is the basis for blockchain systems to implement data control and Turing-complete programming. As technology evolves, the emergence of a series of Turing-complete platform systems such as Ethereum marks that blockchain systems

engage with social issues and financial regulation in the way of more complete and complex scripting languages.

Smart contracts were first proposed in 1995 by Nick Szabo, who coined the term, using it to refer to “a set of promises, specified in digital form, including protocols within which the parties perform on these promises.” This definition is regarded as the prototype of smart contracts. Szabo hopes to embed the smart contract into the physical machine entity to manage and create secure and controllable smart assets, and generate corresponding social value for the emerging physical machine. The concept of smart contracts did not receive much attention from practitioners due to the limitations of the time and the backwardness of the state of the art.

After 2008, the rise of Bitcoin and the maturity of blockchain technology newly defined smart contracts. As a part of blockchain technology, smart contract technology will undertake the work of running modular and intelligent script files in the blockchain system, allowing the systems to have a range of functions such as data set management, executing business logic, and solving basic financial problems. Contracts are compiled into a series of opcodes by a virtual machine after being deployed on the blockchain and then stored in a specific address. A smart contract is triggered when a transaction on the blockchain meets the conditions specified in the contract, and nodes across the network execute the opcode of the smart contract and write the final execution result into a new block. Developers can deploy smart contracts as embedded programmatic protocols into any blockchain data, transaction or asset to form a compliable control system, market monitoring system or digital asset. Smart contracts will not only provide blockchain solutions for the financial industry, but will also play an increasingly important role in managing information, assets, contracts, and regulatory operations in social systems.

Smart contracts can be applied to a large number of data-driven business work. By applying smart contracts, blockchain reduces operational costs, improves work efficiency, and can avoid the interference of some malicious behaviors, improving the security of blockchain applications. At the end of 2013, Vitalik Buterin, the founder of Ethereum, released the Ethereum white paper, titled *Ethernet: The Next Generation of Smart Contracts and Decentralized Application Platform* and launched the Ethereum project. Buterin first saw the tremendous advances that could be made by combining blockchain technology with smart contracts, creating a public blockchain system with an embedded Turing-complete programming language that enables any developer who subscribes to the Ethereum philosophy to create contracts and decentralized applications.

The combination of smart contracts and blockchain greatly enriches the value of blockchain with the following three characteristics.

- It achieves a fair exchange between untrusted parties through the ability to express rich contract rules in the program logic, avoiding the possibility of malicious parties interrupting the agreement, etc.
- It minimizes the interaction between the counterparties and avoids the possibility of unplanned monitoring and tracking.

- It enriches the interaction of transactions with external states, such as stock information from trusted data sources and weather forecasts.

1.4 Status Quo of Blockchain Industry

The development of new technologies cannot be separated from marketing and industrial promotion. It is not enough to only understand its technical principles, but to know the current trend of relevant industries in terms of blockchain technology. This section will analyze the development status of blockchain industry from three dimensions: the development trend of blockchain technology, the government's development plan for blockchain technology, and the ecological map of blockchain industry.

1.4.1 *Blockchain Development Trends*

Blockchain was first applied to “digital currency,” and digital asset crowdfunding experienced the process from germination to explosive growth, but it was followed by disorderly asset issuance, with various projects on the market of different quality, and most functional passes had no practical use value. After the year of 2018, the digital asset market saw the emergence of compliant asset issuance models. From 2017 to 2018, the digital currency craze gradually faded into a diversified technology system. The fervor for technical solutions such as on-chain capacity expansion, off-chain capacity expansion, DAG, cross-chain, and privacy coins continued to climb. Starting from 2019, the industrial development of blockchain will shift from theoretical thinking to physical applications, and from following the trend of issuing “virtual coins” to steadily benefiting the real industry in a longer period of time.

Here is a look at some more information related to blockchain technology as displayed by big data platforms. According to Google Trends, India ranks first in terms of search popularity for blockchain technology by region, followed by Australia, Indonesia, Canada, the UK, and the US. This ranking is related to the population size of the country, but also has a lot to do with the country’s interest in the technology. Google Trends does not have latest data for China, so it is unclear for the time being how the search fervor for blockchain technology compares between China and other countries. However, in order to explore the heat of blockchain in China and the trend situation, the Baidu Index platform, similar to Google Trends, was analyzed, and it was found that the heat of blockchain in China had been on an upward trend since August 2015, which may be related to the convening and publicity of the first Global Blockchain Summit in October 2015, after which more people contacted and paid attention to the new technology. By January 2016, the PBOC held a seminar to discuss the possibility of adopting blockchain technology to issue digital currencies, driving the Baidu Index of blockchain to keep

rising significantly. It was not until June 2016 that the value and security of blockchain was questioned by the public due to the hacking of DAO, the world-famous and biggest crowdfunding project ever at the time, and the forced adoption of measures to address the losses caused by this incident through hard forking, resulting in a significant decline in the corresponding Baidu Index. And by August 2016, the MIIT released a white paper on blockchain development, affirming the value of blockchain technology, and the Index started to rebound again and steadily increased. In 2017, with the holding of large-scale blockchain events such as the Global Blockchain Finance (Hangzhou) Summit Forum and first China Blockchain Development Contest held by the MIIT, the blockchain fever continued to rise. According to Deloitte's 2018 Global Blockchain Survey, nearly 40% of respondents reported that their organization will invest \$5 million or more in blockchain technology in the coming year. About 74% of respondents believe blockchain technology will bring many benefits to their companies. In December 2018, the China Academy of Information and Communication Research released the "Top 10 Trends in Information and Communication Industry (ICT) 2019–2021," stating that blockchain will promote its large-scale commercial exploration applications in healthcare, justice, industry, media, and other fields through exploring the construction of a distributed trust system, deep integration with cloud computing, Internet of Things and other technologies, and innovative breakthroughs.

It can be found through above series of data analysis that, blockchain, an emerging technology, has developed in such a fast-faced and fierce momentum in just a few years. This cannot help but remind people of the Internet a few years ago, the Internet achieved information dissemination and sharing, and the blockchain technology announced the evolution of the Internet from information Internet to value Internet.

1.4.2 ***Government-Driven Blockchain***

With the continuous development of blockchain technology, the awareness of blockchain has gradually increased around the world, and relevant departments have paid attention to, discussed and promoted blockchain technology, and launched corresponding development plans, as shown in Fig. 1.9.

First, take a look at the attention of international organizations on blockchain technology. In early 2016, the United Nations Division of Inclusive Social Development released a report entitled *The Role of Cryptocurrencies and Blockchain Technology in Building a Stable Financial System*, which proposed the idea of applying blockchain technology to build a stable financial system and recognized its value and development potential in the financial sector. Later, the International Monetary Fund (IMF) also released an analytical report entitled *Exploring “Cryptocurrencies,”* which provides a detailed analysis of the future of “digital cryptocurrencies” using blockchain technology. In August 2017, the United Nations Office of Information and Communications Technology (OICT) held a seminar on

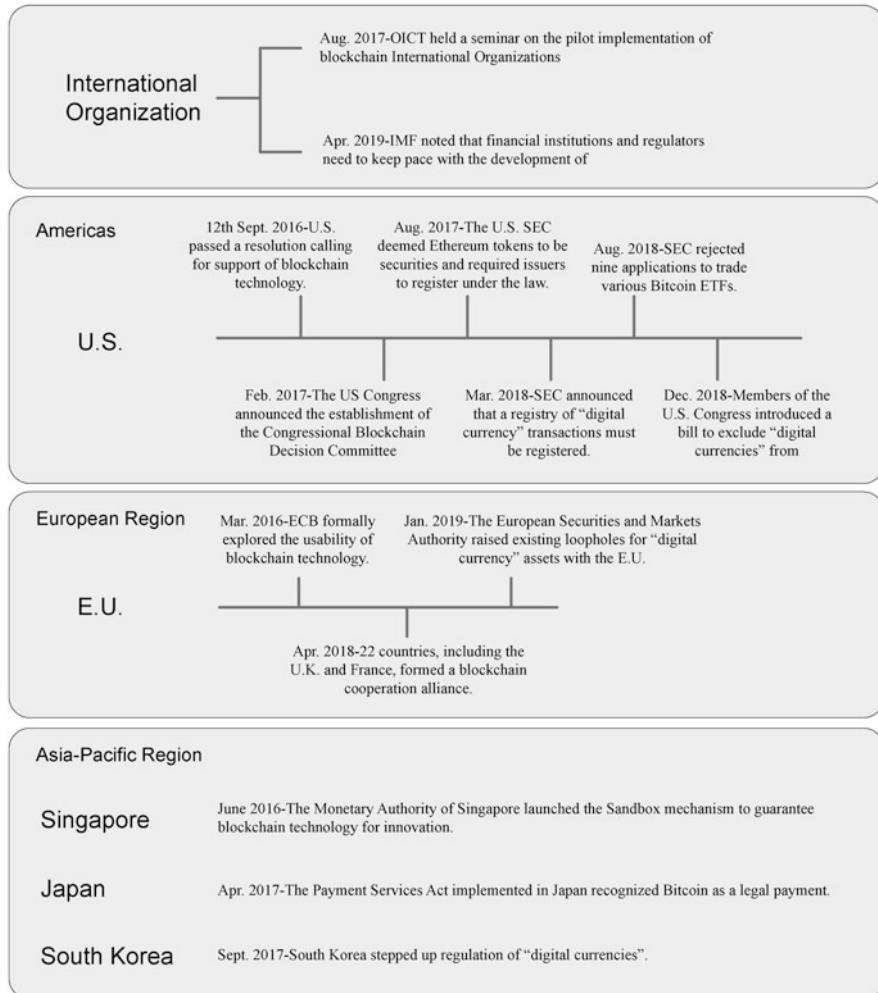


Fig. 1.9 Blockchain development strategies and plans by region

blockchain technology at the UN Headquarters in New York, focusing on the pilot implementation of blockchain projects in order to raise awareness of blockchain technology among UN member states. In April 2019, the IMF and the World Bank launched a “cryptocurrency” called “learning coin,” built on the private chain of the IMF and the World Bank, without any real value, but only to better understand the technical principles underlying “cryptocurrency.” In addition, the IMF noted that central banks, financial institutions and regulators globally need to keep pace with the development of crypto assets and distributed technologies.

In the Americas, several governments have shown support for the application and innovation of blockchain technology, but the regulation of ICOs has become

increasingly strict. For example, on November 10, 2015, the US Department of Justice held a “Digital Currency” summit to explore the possibilities of blockchain technology in “digital currency” applications. Subsequently, the US Securities and Exchange Commission agreed to approve the trading of company stock on the blockchain, and the US Commodity Futures Trading Commission (CFTC) recognized blockchain technology by treating bitcoin as a commodity for regulatory oversight while focusing on blockchain technology. On July 29, 2016, in a total of 22 US Senators sent a letter to the Federal Reserve requesting guidance on the development of blockchain technology, and on September 12, 2016, the US House of Representatives passed a nonbinding resolution calling for support of blockchain technology. On September 28, 2016, Fed Chair Janet Yellen revealed to the public that the Fed was working on an in-depth study to explore blockchain technology. In February 2017, the US Congress announced the establishment of the Congressional Blockchain Decision Committee to explore its application in the public sector. In August 2017, the US Securities and Exchange Commission (SEC) deemed Ethereum tokens to be securities and required issuers to register under the law. In March 2018, the SEC announced that a registry of “digital currency” transactions must be registered. In August 2018, the SEC has rejected a total of nine applications to trade various Bitcoin ETFs. In December 2018, members of the US Congress introduced a bill to exclude “digital currencies” from securities.

In the EU, countries have generally taken a more positive and loosely regulated approach to blockchain technology. In August 2013, Germany was the first country to announce that it would recognize the legal status of Bitcoin and bring it into the regulatory system. The German Government also said that Bitcoin could be used as a “private currency” and a “unit of currency.” In November 2014, UK Treasury officials issued a statement saying that “digital currencies” and their transactions were not subject to national regulation, but in March 2015, the UK Treasury released a report on “digital currencies” proposing to discuss its regulatory model and develop the best regulatory framework. In January 2016, the UK Government released a research report entitled *Distributed Ledger Technology: Beyond Blockchain*, which for the first time discussed, analyzed, and suggested the future and development of blockchain technology at a national level. In April 2018, the UK financial regulator began authorizing supervision of “digital currencies” issued by ICOs. The Central Bank of Russian Federation said in a research plan released in the first half of 2016 that it would explore the application of blockchain technology in the financial field, quite different from its attitude towards Bitcoin. In 2017, messages sent by Bank of Russia indicated that they had established a working group dedicated to the study of cutting-edge and innovative technologies in financial markets to investigate and study distributed ledgers, blockchain technology, and new achievements in a variety of Fintech fields. The European Central Bank (ECB) has also started to explore how blockchain technology can be applied to securities and payment settlement systems and released a report, titled *Eurosystem’s Vision for the Future of Europe’s Financial Market Infrastructure* in March 2016 to formally explore the usability of blockchain technology. In April 2018, in total of 22 countries, including the UK and France, formed a blockchain cooperation alliance, which was

joined a few months later by four European countries, including Denmark. In January 2019, the European Securities and Markets Authority raised existing loopholes for “digital currency” assets with the EU.

In the Asia-Pacific region, the Reserve Bank of Australia (RBA) has been very active in supporting banks in their exploration of distributed ledger technology, proposing the issue of digital version of the Australian dollar, and taking advantage of blockchain technology to reform the traditional financial system. The Prime Minister of Singapore Lee Hsien Loong has called on banks and regulators to keep a close eye on the development of new technologies such as blockchain. In June 2016, the Monetary Authority of Singapore launched the Sandbox mechanism to guarantee blockchain technology for innovation in the financial sector within a controlled scope. On November 16, 2015, Japan’s Ministry of Economy, Trade and Industry held a FinTech conference to discuss the development and application of FinTech, in which the development and future of blockchain technology was specifically analyzed and discussed. In April 2017, the Payment Services Act implemented in Japan recognized Bitcoin as a legal payment, but set clear requirements for the regulation of exchanges. On February 3, 2016, the Bank of Korea actively researched and discussed the status of blockchain technology and “digital currency” as well as distributed ledger technology in an analytical report entitled *Status and Implications of Distributed Ledger Technology and “Digital Currency.”* In September 2017, South Korea stepped up regulation of “digital currencies.”

1.4.3 Ecological Mapping of Blockchain

Blockchain technology is a universal underlying technology framework that can bring profound changes to various fields such as finance, economy, technology, and even politics. In the early stage of development, i.e., Blockchain 1.0, it was mainly adopted as the technical support for the “digital currency” (Bitcoin) system, and only realized a single payment function, so at this stage, blockchain applications and the underlying platform were closely coupled. However, as a new generation of blockchain platforms emerges represented by Ethereum, blockchain has entered the stage of 2.0, in which its applications and basic platforms start to be decoupled. Taking Ethereum as an example. It provides a more complete blockchain-based protocol and a Turing-complete smart contract, enabling users to develop a variety of decentralized applications on its platform. It can even be compared to a new Internet TCP/IP, relying on this protocol and the various APIs it provides to help developers develop decentralized applications or transplant some of the original Internet applications into a decentralized network. Thus, the whole blockchain industry chain began to derive various ecological levels.

The participants of blockchain industry chain can be divided into four layers: application layer, intermediate service layer, basic platform layer, and auxiliary platform layer. Among them, the application layer mainly serves the end users (individuals, enterprises, and governments). Developers develop different

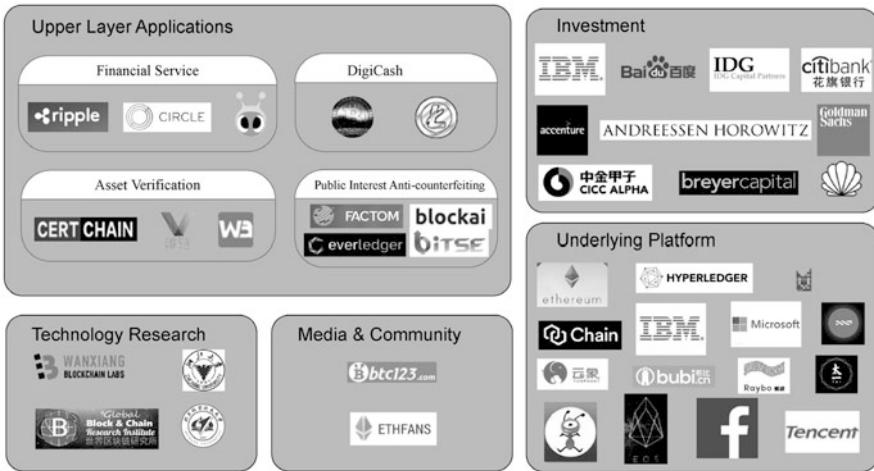


Fig. 1.10 Blockchain industry ecological mapping

decentralized applications based on different user needs to serve different industries. The intermediate service layer generally facilitates customers to carry out secondary development of various applications based on the underlying blockchain technology, and provides convenient tools and protocols for them to transform business processes using blockchain technology. The foundation platform layer mainly focuses on the basic protocols and underlying architecture of blockchain and offers technical support for the development of blockchain ecology in the whole society. The auxiliary platform layer is not a major player in the blockchain industry chain, but a very important external auxiliary force for the development of the blockchain industry, including funds, media and communities.

In general, blockchain applications fall into two categories:

The first category is the applications developed based on the characteristics of blockchain distributed bookkeeping, including identity verification, equity proof, asset authentication, etc.

The second category is various decentralized applications developed by using the decentralized system of blockchain. From the perspective of technical feasibility, all industries involved in value transfer can be reconstructed at the bottom by blockchain technology at present.

Figure 1.10 is the ecological mapping of the current blockchain industry, demonstrating the application of blockchain technology by some companies and organizations in their respective fields. In general, the entire blockchain industry includes the underlying platform, the upper application, technology research, media and community, investment, computing and security, and other ecological areas. In the field of blockchain underlying platform, the development platforms represented by Ethereum, Fabric and Hyperchain innovate the underlying blockchain technology and deliver underlying technical support for decentralized blockchain-based

applications. In the upper tier, developers are exploring application scenarios in various industries, such as financial services represented by Ripple and Circle, notarization and anti-counterfeiting by Factom, supply chain by Skuchain, and so on. In addition, blockchain alliances or blockchain labs have been established around the world to study blockchain-related technologies, and major financial companies have also started to participate in blockchain project investment, while some online media and communities have reported and discussed information related to blockchain technology.

Blockchain technology is considered to be a new technology with wide application prospect, but in recent years, due to the difficulties facing information supervision in the field of “digital currency,” the “digital currency” platform once became a channel for malicious speculators to make quick and illegal profits. In September 2017, the state explicitly banned ICO financing and “virtual currency” trading, which effectively curbed the illegal speculation and ensured the healthy development of blockchain industry.

1.5 Blockchain Application Scenarios

Currently, blockchain technology has shown application prospects in many fields, and many institutions and organizations are interested in blockchain technology and are taking active role in exploring for its implementation in this field, and this section will analyze and introduce the current main application scenarios of blockchain.

1.5.1 *Digital Bills*

Traditional paper-based bills are subject to risks such as easy loss, forgery and tampering. By introducing blockchain technology, the information and status of bills can be recorded on the blockchain platform. Once a transaction is generated, each node on the blockchain first verifies the transaction, and when the nodes reach a “consensus,” the transaction is recorded on the blockchain, and it is difficult to be tampered with. The presence of multiple copies in the system that increases the cost of malicious tampering of the content therefore provides higher security compared to traditional bills. In addition, in the traditional bills industry, reconciliation and clearing between various institutions is relatively complicated, while blockchain technology can effectively improve the efficiency of fund clearing by means of joint bookkeeping and mutual verification by various nodes. At the same time, each institution maintains relatively independent business autonomy thus achieving a perfect balance of efficiency and flexibility. The problem of mutual trust among participants makes the traditional bill circulation cumbersome and difficult to monetize, so as to achieve mutual benefit. By registering the bill information on the blockchain platform and taking advantage of the low cost of blockchain expansion

and simplified transaction steps, the bill can be transformed into a digital asset with a certain degree of standardization that can be held, circulated, split, and monetized by customers.

1.5.2 Supply Chain Finance

The traditional supply chain finance platform is generally dominated by a single financial institution, which makes it difficult to achieve inter-industry expansion and promotion. Blockchain technology allows the participants to focus only on business system that connects to the blockchain platform, which can achieve rapid industry-wide coverage. The trade information and credit financing information among enterprises in the supply chain, as well as the storage and logistics information involved in the trade process, are registered on the blockchain, and the information is not easily tampered with, which ensures the authenticity and validity of the assets and reduces the cost of enterprise financing and bank credit cost. Cross-institutional information is kept synchronized through the consensus mechanism of blockchain and distributed ledger, and complete transaction data can be obtained by accessing any node, breaking information silos. Institutions can obtain complete transaction data by accessing internal blockchain nodes to enhance inter-enterprise credit collaboration. By recording asset certificates such as accounts receivable, promissory notes, and warehouse receipts on the blockchain and supporting related operations including transfer and pledge, assets are digitized and a digital credit system that can transfer value peer-to-peer is constructed through the blockchain to realize its value transmission. This trusted value transmission system improves both the financing ability of the demand side and the supervisory ability of the supply side, providing a fundamental guarantee for a sound and stable financial system. Controlling the supply chain process through smart contracts reduces human intervention and improves industrial efficiency. Without the central platform's audit confirmation, real storage and logistics information is detected through sensors and trusted data is sent to blockchain verification nodes using wireless communication networks to ensure that relevant operations are automatically triggered when contract conditions are met, reducing operational errors.

1.5.3 Accounts Receivable

Traditional accounts receivable is completed through offline transaction confirmation, and the existence of risks such as forged transactions and tampered accounts receivable information reduces the trust of transaction participants. The whole process of accounts receivable operation is carried out through the blockchain platform, which realizes the authentication of signatures and non-repudiation of accounts receivable transactions. The smart contracts that enable authority and status

control make accounts receivable more secure and controllable and build a highly trusted trading platform. In the process of accounts receivable transaction, there are many parties involved and the business is complex. In the face of traditional accounts receivable financing applications, financial institutions have to run a lot of trade background checks. The blockchain platform records the entire lifecycle of accounts receivable through timestamps thus enabling all market participants visible to the flow of funds and information, eliminating the possibility of invoice fraud. Traditional accounts receivable are difficult to circulate in the trading market due to mutual trust issues. Accounts receivable are stored and traded as digital assets, which are not easily lost and cannot be tampered with, allowing the new business model to be rapidly promoted. It reduces the cost of use while improving the efficiency of customer fund management, and forms a mutual trust mechanism among different enterprises, enabling multiple financial ecosystems to interoperate and benefit from each other through the blockchain platform, which has good business value and broad space for development.

1.5.4 Data Trading

Data as a special commodity has unique characteristics, and there is a risk of being copied and transferred to storage. The data intermediary platform established according to the intermediary model of commodity circulation constitutes a potential threat to the rights and interests of both parties to data transactions and turns into an obstacle to data transactions. Only by establishing an information platform that conforms to the characteristics of data and safeguarding the security and rights of data through technical mechanisms instead of mere promises, can the smooth flow of data be accelerated by truly reassuring both parties to data transactions. By confirming the rights of data through blockchain technology, it can effectively protect the rights and interests of all parties of data, eliminate the risk of data being copied and resold many times, turn data into protected virtual assets, and confirm and record every transaction and data. Taking use of traceability and non-tampering characteristics of blockchain, it can ensure the compliance and effectiveness of data transactions, stimulate the enthusiasm of data transactions, and contribute to the scale growth of the data market.

1.5.5 Bond Trading

Bond business is a business that requires the joint participation of multiple institutions. In its issuance and trading processes, information needs to be synchronized and confirmed between institutions by traditional mail or message forwarding. If the bond issuance and transaction are realized through a centralized system, there may be risks of human errors or malicious tampering. After using blockchain technology,

the system can be guaranteed by the blockchain substrate to keep the synchronization and consistency of data, reduce the time, labor, and capital cost of interfacing between systems of different institutions, and upgrade from relying on inefficient collaboration based on business flow to a low-cost, high-efficiency, and high-trust collaboration system that does not rely on any intermediary but is guaranteed by the platform for basic business processes. In addition, the traditional centralized system has a lot of information closed inside the organization, which cannot timely and effectively supervise the external system, and there will be blind spots in supervision. Using blockchain technology, regulators join the blockchain in the form of nodes to monitor the transactions on the blockchain in real time. Meanwhile, smart contracts make bonds restrictive and controllable throughout their life cycle, which can also effectively improve regulatory effectiveness. Due to the data integrity and non-tamperability property of blockchain, any value exchange history can be tracked and queried, and the flow process of bonds can be clearly viewed and controlled, and thereby ensuring the safety, validity, and authenticity of bond transactions and effectively preventing market risks. Blockchain-based technology can also avoid the reconciliation and clearing by third-party institutions thus effectively enhancing the clearing efficiency of bond transactions.

1.5.6 Block Trading

Blockchain technology-based bulk trading platform can realize real-time clearing of bulk transactions among clearing banks, improve the efficiency of bulk transactions, and facilitate business development. Smart contracts control the bulk transaction process, reducing human interaction and improving processing efficiency. No central platform review and confirmation are required to ensure that relevant operations are automatically triggered when quotes meet the aggregation conditions, reducing operational errors. Exchanges and clearinghouses can back each other up and are responsible for order broadcast of all transaction data and initiating consensus. Fault tolerance can play a role in a disaster recovery strategy. And the primary node can be switched in seconds in case of major failures. When the access node fails, the built-in algorithm quickly recovers historical data to avoid transaction data loss. Members and bank access side independently process queries, and data is synchronized in real time to reduce the pressure on the master node.

1.5.7 Cross-Border Payment

Traditional cross-border payment relies heavily on third-party institutions due to currency and exchange rate problems. This has two main problems: first, the process is cumbersome and long-period, and second the fees are high. Traditional cross-border transactions are nonreal-time and usually take a day. Costs would naturally be

high due to manual reconciliation. Currently, the cross-border payment process of some third-party payment companies such as UnionPay, Tenpay, and Alipay, is roughly as follows.

1. Domestic users can use common domestic payment methods (such as net banking, quick payment, and code sweeping) when purchasing goods on cross-border e-commerce platforms.
2. The payment company goes to the partner bank to purchase foreign currency (after a successful purchase, the foreign currency enters the foreign currency reserve account of the payment company).
3. The merchant maintains the information of the designated overseas payee, and the payment company pays overseas foreign exchange (from the foreign currency reserve of the payment company to the account of the overseas payee).

The introduction of blockchain technology in cross-border payment scenarios, with its technical features such as distributed ledger, data not easily tampered with, full traceability, real-time supervision, and verification by multiple parties, can efficiently achieve credible sharing of fund and information flows thus establishing a new decentralized value transfer model and path, breaking the original trust-building model based on third-party institutions and reestablishing a new technology-based model that is objective, fair, credible, and efficient.

1.5.8 Other Scenarios

Blockchain, a protocol that can transfer value, can be applied to all other scenarios related to value transfer, such as digital copyright, notarization, identity authentication, and social welfare in addition to the above scenarios.

In the field of consumer finance, Sunshine Insurance Company uses blockchain technology as the underlying technical structure to launch “Sunshine Shell” credits, which enables users to transfer credits to friends in the form of “red packets” and exchange them with credits issued by other companies on top of the ordinary credits function.

In the area of digital rights, in view of intellectual property infringement, blockchain technology can confirm the rights of works through time stamps and hash algorithm, prove the existence, authenticity, and uniqueness of intellectual property rights, and trace the whole life cycle of works, which greatly reduces the cost of safeguarding rights.

In the healthcare sector, patients' personal data has been leaked from time to time. In April 2015, Factom announced a partnership with HealthNautica, a provider of medical records and services solutions, to study the use of blockchain technology to protect medical records as well as track accounts and provide tamper-proof data management for medical records companies.

In the education sector, the incomplete student credit system and the absence of a chain of historical data has resulted in the government and employers not having

access to complete and valid information. Storing students' academic data using blockchain technology can solve the problem of opaque and easily tampered information, which is conducive to building a benign student credit system.

In the field of social welfare, charities must have credibility if they want to build public trust, and information accessibility is necessary. Companies such as Ant Financial have launched a pilot using blockchain technology to track charitable donations, which provides a possibility to accelerate the transparency of public welfare. Blockchain technology can also be used in the field of government information disclosure to facilitate government departments to implement public governance and service innovation and improve the efficiency and effectiveness of the departments.

Blockchain applications go far beyond cryptocurrency and bitcoin, and it unlocks infinite possibilities for the future. This requires more excellent companies, businesses, and top talents to join for exploring blockchain technology so that it can develop faster and better. There is every reason to expect that under the paradigm of blockchain technology, the advent of another "Grand Voyage Era" will bring a reconstruction to all walks of life and society.

1.6 Mainstream Blockchain Platforms

This section will give a brief introduction and comparative analysis of the current mainstream blockchain platforms.

Bitcoin is the first blockchain application that uses a PoW mechanism to reach a consensus among network nodes. Since anyone is free to join and participate in the bitcoin network, it is a public chain that does not support smart contracts but does support simple operational programming with Turing's incomplete scripts. Its public network TPS is far below seven.

Ethereum is an open-source, Turing-complete blockchain with smart contract functionality, also known as the "second generation blockchain platform." The platform builds a vibrant ecosystem of apps around smart contracts and offers multi-language clients like Go-Ethereum, PyethApp, and Parity to make development easier and more efficient. Users can build their own DApp applications and publish them on the Ethereum platform by using Ethereum and Solidity-based contracts released by the platform. The current public TPS of Ethereum is about 100.

Hyperledger Fabric, an incubation project launched by the Hyperledger Consortium founded by the Linux Foundation, is building a standardized digital ledger. It aims to help reduce the expense of work and increase efficiency among startups, governments, and corporate alliances by blockchain technology. As such, it is not public-facing, but serves allied groups such as companies, businesses, and organizations, and is part of a federated chain. The platform adopted the Go language, and the consensus algorithm uses the PBFT algorithm. Similarly, it also supports smart contracts that are called Chaincode in Hyperledger Fabric. Chaincode is executed

only on authentication nodes and runs in an isolated sandbox, taking Docker as the container for executing Chaincode. The TPS of the fabric is about 3000.

EOS (Enterprise Operation System) is a new blockchain smart contracts platform developed by Block one. As its name indicates, the goal of building EOS is to bring a perfect and convenient “operating system” for commercial-grade smart contracts and applications, and the graphene technology it adopts solves the problems of low transaction time and throughput of Ethereum. As a result, the EOS architecture is capable of running commercial-grade program scheduling and parallel computing, and regular users of EOS can run smart contracts by pledging tokens rather than consuming them. The TPS of the EOS public network is about 3600.

BitShares, introduced as a Peer-to-Peer Polymorphic Digital Asset Exchange, is regarded as the originator of the DPoS consensus mechanism. It provides the next generation of entrepreneurs, investors, and developers with the technology to leverage global decentralized consensus and decision-making to look for free-market solutions. BitShares also hopes to apply blockchain to all Internet-based industries (e.g., banking, stock exchanges, lotteries, voting, music, auctions, etc.) to build distributed autonomous corporations (DACs) through a digital public ledger at a lower cost than traditional ones. The BitStock blockchain, as a public chain, is developed using the C++ language for its core technology framework and has a public network TPS greater than 500.

Factom applies blockchain technology to data management and records in the business community and government sector, assisting in the development of various applications, including audit systems, medical information records, supply chain management, voting systems, property deeds, legal applications, financial systems, and more. Factom makes policies and reward mechanisms based on user interests. In its mechanism, only when a user’s interest is submitted to the system can the corresponding voting rights be obtained, while transferable Factoid interests do not have voting rights, which effectively avoids the problems of “Stake Grinding” and “Noting at Stake” of the PoS mechanism. The core technology framework of Factom is developed in Go language, and the TPS is about 27.

Ripple, the world’s first open payment network that implements features such as blockchain-based payments, introduces a consensus mechanism, the RPCA. This consensus mechanism divides the entire network into many sub-networks and then performs consensus operations on the sub-networks. Since the trust cost of sub-networks is low, their transactions are faster and more efficient, while sub-networks must remain connected to 20% of the nodes in the entire network to ensure consistent data. The core technical framework of ripple is developed in C++ language, and the public network TPS is less than 1000.

Nextcoin (NXT), as the second generation of decentralized “virtual currency,” is the first “electronic currency” that adopts 100% equity proof PoS, eliminating the disadvantages of the first generation of currency, such as large resource consumption and vulnerability to attack. This is inseparable from the transparent forging proposed by Futurecoin, which can realize the automatic division of labor of nodes in the whole network, and significantly improve the transaction speed under the condition that miners are identified and there is no need to find bookkeeping nodes.

Table 1.3 Blockchain platform comparison

Platform	Consensus mechanism	Type	Language	Smart contracts	TPS
Bitcoin	PoW	Public chain	C++	N/A	<7
Ethereum	PoW and PoS	Public chain	Go	Yes	About 100
Hyperledger fabric	PBFT	Consortium chain	Go	Yes	About 3000
Eos	DPoS	Public chain	C++	Yes	About 3600
BitShares	DPoS	Public chain	C++	N/A	>500
Factom	PoS	Public chain	Go	N/A	About 27
Ripple	RPCA	Public chain	C++	N/A	<1000
NXT	PoS	Public chain	Java	N/A	<1000
Hyperchain	RBFT	Consortium chain	Go	Yes	>10,000

Meanwhile, forging point clearing for inactive nodes can greatly reduce the risk of network forking. Futurecoin adopts Java as programming language for its core technical framework with TPS less than 1000.

Hyperchain is a federated blockchain technology infrastructure platform developed by Hangzhou Funchain Technology to meet the needs of the industry. By integrating and improving the cutting-edge technologies in the blockchain open-source community and research areas, Hyperchain integrates the high-performance reliable consensus algorithm RBFT, compatible with the smart contract development language and execution environment of the open-source community. It also enhances key features such as bookkeeping authorization mechanism and transaction data encryption, and provides a powerful visual Web management console for efficient management of blockchain nodes, ledgers, transactions, and smart contracts. Like Fabric, Hyperchain which adopts the modular design concept is divided into eight core modules: consensus algorithm, permission management, multilevel encryption, intelligent contract engine, node management, block pool, ledger storage, and data storage. It is designed to serve digital assets and financial asset business applications such as notes, depository receipts, equity, bonds, registration, and supply chain management, and its system is able to process tens of thousands of transactions per second, second to none among current blockchain platforms.

Table 1.3 lists the consensus mechanism used by each platform, the type of blockchain it belongs to, the language used for platform development, whether it supports smart contracts, and the transaction processing per second (TPS) performance metrics for readers to make more intuitive statistics and comparisons.

It can be seen from the above platform introduction and comparison that the current blockchain platforms use different consensus algorithms. The corresponding consensus mechanisms have their own advantages and shortcomings in different application scenarios. There are basically two main categories of platforms: public chains and consortium chains, and private chains are less used. The programming

languages used for platform design are mainly Go and C++, because the environment in which the blockchain network is located is a distributed network, which requires high concurrency and efficient operation. Whether or not smart contracts are supported depends on the scenarios that each platform faces and the services it provides. For example, Ether, Hyperledger Fabric, EOS, Hyperchain, etc., as the underlying platforms, generally need to provide smart contract functions, while for other application platforms, smart contracts are not really needed. The performance of blockchain platforms, on the other hand, is constantly improving with the development of blockchain technology and has basically met the requirements of commercial applications in certain applications, among which the Hyperchain platform has reached a TPS of 10,000, with significant advantages in blockchain performance.

1.7 Summary

This chapter makes a panoramic analysis of blockchain technology, introduces the basics and development process of blockchain, provides a detailed explanation of its key technologies and features, analyzes the current state of the blockchain industry in the context of the times, selects some typical application scenarios for elaboration, and finally introduces and compares the current mainstream blockchain platforms so that readers can have a preliminary understanding and knowledge of blockchain technology and lay the foundation for the subsequent advancement and practice.

Part II

Open Source Blockchain Platforms

Chapter 2

In-Depth Interpretation of Ethereum



Ethereum is a Turing-Complete open-source platform that has attracted a large number of decentralized application developers. This chapter will give a shallow-to-deep explanation of Ethereum, mainly including related concepts, such as nodes, mining, accounts, gas, and messages; Ethereum core principles, including consensus mechanism, EVM, data storage, and cryptographic algorithms; Ethereum smart contracts, including the syntax structure of smart contracts, compilation deployment, and testing; major events dissection, the DAO event and the main problems of current Ethereum, such as consensus efficiency issues and privacy protection issues, and the upcoming Ethereum 2.0.

Ethereum, known as Blockchain 2.0, is a blockchain-based application platform. It allows anyone to build and use decentralized application DApps based on blockchain technology through smart contracts on the platform. Its core concept is a blockchain with a built-in Turing-complete programming language, which indicates that any computable problem can be solved by computation. The foundation for building the Turing-complete is the EVM. It is similar to the Java virtual machine (JVM), compiled from source code and run on bytecode, but can be implemented in a high-level language for development and automatically converted to bytecode by the compiler.

Ethereum-based applications need to provide predefined business logic to meet the normalized operation when transacting and running, a mechanism known as a smart contract in the blockchain. A contract acts as an automated agent, and when conditions are met, the smart contract automatically runs a specific piece of code that completes the specified logic. Smart contracts will be further explained in Sect. 2.3.

2.1 Basics Introduction to Ethereum

This section will introduce the development history, basic concepts, types of clients, account management methods, and Ethereum network, in order to facilitate readers to understand Ethereum intuitively and comprehensively, and lay the foundation for the development of Ethereum-based projects.

2.1.1 *History of Ethereum Development*

At the end of 2013, Vitalik Buterin, the founder of Ethereum, released a “white paper” on Ethereum and gathered a group of developers in the global cryptocurrency community who agreed with the idea of Ethereum and launched the related project. According to the vision of Vitalik Buterin, Ethereum’s development was planned with four major stages: Frontier, Homestead, Metropolis, and Serenity.

In early March 2014, the third proof-of-concept release (POC3) of Ethereum was released; in April, Gavin Wood published Ethereum “Yellow Book,” which standardized important technologies such as the EVM. In June, the team released the fourth proof-of-concept (POC4) and quickly moved towards the fifth proof-of-concept (POC5). During this time, the team also decided to operate Ethereum as a nonprofit organization.

In July 2015, Ethereum launched the Frontier version, which features Ethereum mining the most. With the mining feature, developers can test decentralized applications in the Ethereum blockchain. The Frontier version of the Ethereum client has multiple language implementations, but only has a command-line interface and does not provide a graphical interface, so the users at that stage are mainly developers and researchers.

In March 2016, Ethereum released its first production-ready version Homestead. At the time of writing, Ethereum was in the Homestead version. This indicates that Ethereum is running smoothly. At this stage, Ethereum provides a graphical interface to the Ethereum wallet, which can be easily operated by nontechnical people other than developers.

In 2017, Ethereum launched the Metropolis version, but given the cost and difficulty of development, the upgrade to the Metropolis phase would include two hard forks: Byzantium and Constantinople. On October 16, 2017, after several delays, Byzantium successfully underwent a hard fork. However, there is no exact official time for the Constantinople hard fork. The so-called hard fork is a change to the underlying protocol of Ethereum, creating new rules to improve the entire system.

In the Metropolis release, Ethereum will have many new features impacting its future development. For example, it will implement zk-Snarks (noninteractive zero-knowledge proofs), early implementation of PoS, more flexible and stable smart contracts, etc.

The release date of the Serenity Phase four version has not yet been set. In the Serenity release, Ethereum switches its consensus mechanism from PoW to PoS. PoW is a serious waste of computing power and power resources, while PoS will dramatically improve block generation efficiency. With the switch to PoS, the mining required in the first three phases will be terminated, and new Ethereum coins will be issued at a much lower rate, even to the point of no new coins being issued, and hence the term “Serenity” phase.

2.1.2 Basic Ethereum Concepts

Ethereum consists of many nodes, accounts corresponding to them, and a “transaction” between two by sending a message. The information carried in the transaction, and the code that implements a particular function is called a smart contract, and the environment in which the smart contract runs is the EVM. The EVM runs on each node, and the transaction requires the participation of nodes that generate the workload by repeating hash operations. These nodes are called miners, and the process of computation is called mining. The transactions are calculated at a cost, called gas. gas is generated by converting Ethereum coins, the medium used on Ethereum to pay for transaction fees and computing services. Consumed gas is used to reward miners. Those based on the above smart contract code and the Ethereum platform are called decentralized applications. The basic concepts of Ethereum include the following.

1. Nodes: Nodes allow the reading and writing of blockchain data. Many of the applications on Ethereum are now based on the public chain, so each node has the same status and rights; there is no central server, and each node can join the network and read and write data inside. Consensus mechanisms used between nodes ensure the reliability and correctness of data interactions. A single node can create a private chain, and several mutually trusted nodes can build a federated chain.
2. Miners: A miner is a network node that generates workload by constantly repeating hash operations. Its task is to solve mathematical puzzles and place the results into new blocks. Miners compete with each other and the node to calculate the consequence first will broadcast it to the whole network; when the result is confirmed, the reward contained in the newly generated block will be given to that node and deposited in the Ethereum address. The Ethereum coins held by that node can be used as an asset for the next initiated transaction.
3. Mining: In Ethereum, the only way to issue Ethereum coins is by mining. The mining process also ensures the verification and reliability of transactions in the blockchain. Mining is a task that requires vast computing power and time and is limited to a specific period, while dynamically adjusting the difficulty of the mining. In simple terms, the miner aims to take the current block's header, and add a random number for SHA256 to calculate the hash value. If the calculated

hash value meets certain conditions, such as the first 60 bits being 0 or less than or equal to some predetermined random number (Nonce), the miner wins the right to create the block.

4. Accounts: Two types of accounts are included in Ethereum: external accounts and contract accounts. External accounts are controlled by public-private key pairs. Contract accounts, on the other hand, uniquely identify a particular smart contract on the blockchain. Both types contain an Ethereum balance that can send transactions on Ethereum. Each account has an address length of 20 bytes and a persistent memory area called storage, which takes the form of a key-value pair, with both the key and the value being 32 bytes long. Notably, the address of the external account is determined by the public key. In contrast, the address of the contract account is chosen when the contract is deployed. When the contract account receives a legitimate transaction, it executes the contract code contained within. So the big difference between the two types is that contract accounts store the code and external accounts do not.
5. gas: Every transaction on Ethereum involves a miner and is subject to a fee, which is called gas in Ethereum. The purpose of gas is to limit the work required to execute the transaction while paying for the execution. While the EVM executes the transaction, the gas is gradually depleted according to specific rules. The price of gas is set by the creator of the transaction and the transaction fee = gas price * gas amount. If any gas is left at the end, this gas will be returned to the sender's account, while the consumed gas is treated as a reward and issued to the miner's account.
6. EVM: The EVM provides a running environment for smart contracts and is a sandbox environment wholly isolated from the outside. Smart contract code cannot perform network operations, file I/O, or other processes while running inside the EVM. Only limited calls can be made between smart contracts, which ensures the independence of contract operation and maximizes runtime security.
7. Smart Contracts: A contract is a collection of code and data at a specified address on the Ethereum blockchain. Contract methods support rollback operations, if an exception occurs while executing a method, such as running out of gas, all operations already executed by that method will be rolled back. But there is no way to tamper with an erroneous transaction once it has been performed. Details of the development of smart contracts will be further discussed in Sect. 2.3.
8. DApp: DApp, standing for decentralized application, is an Internet application which differs most from the traditional applications because DApp is deployed in a decentralized network, also known as the blockchain network (or Ethereum). Being decentralized, DApp has no central control and no single point of failure. A complete DApp is a fully open-source and autonomous application. DApps are self-tolerant and are not interfered with by centralized organizations and individuals. DApps cannot delete, modify or even shut down certain data. All data in a DApp must be encrypted and stored in a decentralized blockchain application platform. User and organization data are held encrypted, so DApps have extremely high data security and do not have the problem of mass leakage of

user data. Also, DApp is difficult to be modified and can only be upgraded and maintained with the consent of the majority of users.

9. Transactions: In Ethereum, transactions are marked by state transfers, consisting of objects called “accounts” and transfers of value and information state transitions between two accounts. Ethereum accounts are divided into externally owned accounts controlled by public and private keys and contract accounts controlled by contract code. Externally owned accounts have no associated code; users send messages from an external account by creating and signing a transaction. Every time the contract account receives a message, its internal code is activated to read and write to internal storage, send messages, or call a method.

Once the account has been determined, the Ethereum transaction begins. A “transaction” is a signed packet that stores messages sent from an external account, and the more critical part of the transaction process is the messaging mechanism. Ethereum’s messaging mechanism ensures that contract and external accounts have equal rights, including sending messages and creating other contracts. It allows contracts to be engaged by different actors simultaneously, cosigning to provide services without caring what type of account each party to the contract is.

The execution process for each transaction is as follows.

1. Building the initial object for the transaction. The user sets the address required for the transaction, the number of gas, and the total amount of Ethereum sent.
2. Signing the transaction. The user uses the private key to sign the original transaction object. It is used to prove that the user indeed made this transaction.
3. Ethereum node verification. The signed transaction object will be submitted to the Ethereum node for verification, which ensures that the transaction has been signed by the transaction account, guaranteeing the security of the transaction.
4. Broadcast transaction. After the authentication is passed, the node will post the transaction information to the other peer nodes. The peer nodes will continue to post until the transaction is broadcasted into the network and output the transaction ID.
5. The miner node receives the transactions and packages them on top of the block. The main responsibility of the miner is to maintain the block, the transactions are first added to the transaction pool and the miner evaluates the transactions for packaging based on the price of the transaction’s gasprice.
6. The miner packs the transactions, adds them to the block, and broadcasts them. After consuming a certain amount of computing power, the miner “mines” a valid block, packs the transactions from the transaction pool into the block, and broadcasts it across the network.
7. Synchronized block update. Eventually, the whole network accepts this new block and updates the entire blockchain.

2.1.3 *The Ethereum Client*

After its release, Ethereum has multiple language versions of the client and supports multiple platforms and languages such as go-ethereum and cpp-ethereum. After Ethereum entered the Homestead phase, the Go language client dominated. Meanwhile, the officially developed Ethereum wallets are gradually becoming popular. This section will introduce the mainstream Ethereum clients.

Go-ethereum

Go-ethereum is one of the three original implementations (along with C++ and Python) of the Ethereum protocol. It is written in Go, fully open source. Go-ethereum, also known as geth for short, is the most popular Ethereum client, a full command-line interface and an Ethereum node. By installing and running geth, common functions such as private chain building, mining, account management, smart contract deployment, and Ethereum interface invocation can be implemented. Go-ethereum is now hosted on GitHub.

Go-ethereum can connect to a network of other users' clients (also known as Ethereum nodes) and update and synchronize block data, while clients can continuously pass information to other nodes to keep their local copies of data up to date. At the same time, Go-ethereum can mine blocks, add transactions to the blockchain, and verify and execute transactions within blocks.

Here are the standard command lines for geth.

```
geth +
account      Manage accounts
attach        Starts the interactive JavaScript environment (connects
to the node)
bug          Report a bug on geth
console       Start an interactive JavaScript environment
copydb        Create a local chain in the target chain data folder
dumpconfig    Display configuration values
export        Export block information to file
import        Import a block information file
init          Initialize and create a new Genesis block
js            Execute a specific JavaScript file
license       Display protocol information
makecache     Generate ethash verification cache for testing
makedag      Generate ethash mining for testing
monitor       Monitor and display node indicator information
removed      Removal of blockchain and state database data
version       Display the version number
wallet        Manage Ethereum pre-sale wallet
```

```

INFO [07-01|12:14:53.630] Initialising Ethereum protocol
INFO [07-01|12:14:53.654] Loaded most recent local header
78 ns=3y10ms4w
INFO [07-01|12:14:53.659] Loaded most recent local full block
ns=50y2ms2w
INFO [07-01|12:14:53.663] Loaded most recent local fast block
73 ns=3y10ms4w
INFO [07-01|12:14:53.669] Upgrading chain index
INFO [07-01|12:14:53.669] Loaded local transaction journal
INFO [07-01|12:14:53.678] Regenerated local transaction journal
INFO [07-01|12:14:53.886] New local node record
=30303
INFO [07-01|12:14:53.891] Started P2P networking
self=enode://lab19c801c83c4839714357d10f45810ad2d43e3
0957e27e0d858d2b08d732ed0db2aa1a2237801918de701e366c497e4da
761eb68746109a423cc58eee52@127.0.0.1:30303
INFO [07-01|12:14:53.892] IPC endpoint opened
url='\\\\.\\pipe\\geth.ipc
INFO [07-01|12:14:58.498] Finished upgrading chain index
type=bloombits

```

Fig. 2.1 The geth client interface

Once you enter the geth console, the following common commands are available.

```

> eth.accounts: query account list
> personal.listAccounts: query account list
> personal.newAccount(): create a new account
> personal.deleteAccount(adrr,passwd) : delete the account
> personal.unlockAccount(adrr,passwd,time): unlock the account for
trading operations
> eth.sendTransaction({}): sends a transaction

```

Open the geth client, as shown in Fig. 2.1.

Browser-Solidity (Remix)

Browser-solidity, also known as Remix, is an online browser compiler. The client does not require installation and can be developed, debugged, and compiled directly in the browser. It works the same way across operating systems and is suitable for beginners to get started with compiling and deploying contracts and testing them. In smart contract development, browser-solidity is one of the most commonly used tools.

The main browser-solidity interface is shown in Fig. 2.2.

Parity

Parity is a very active open-source project for Ethereum that many developers currently love. Parity can be accessed with a browser, and the Ethereum wallet and DApp development environment built into Parity facilities developers develop smart contracts and deploy distributed applications quickly and easily. Using Parity to interact with the Ethereum blockchain is the fastest and most secure way to do so. Parity can also provide technical support for most of the infrastructure in public Ethereum, making it easy for developers to develop and research.

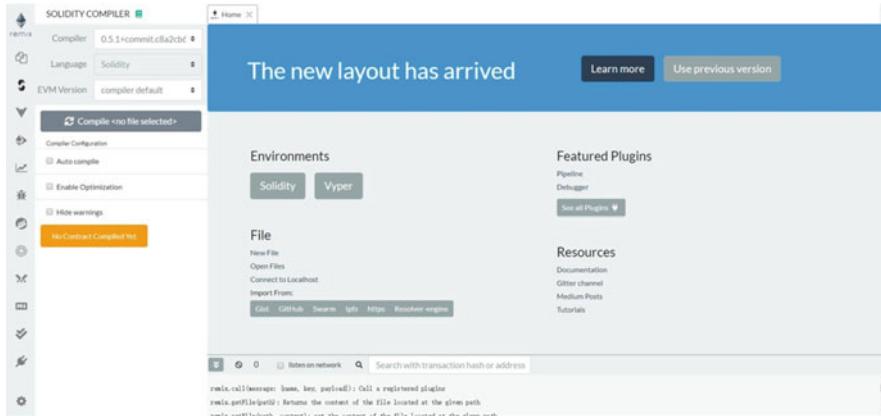


Fig. 2.2 Browser-solidity main interface

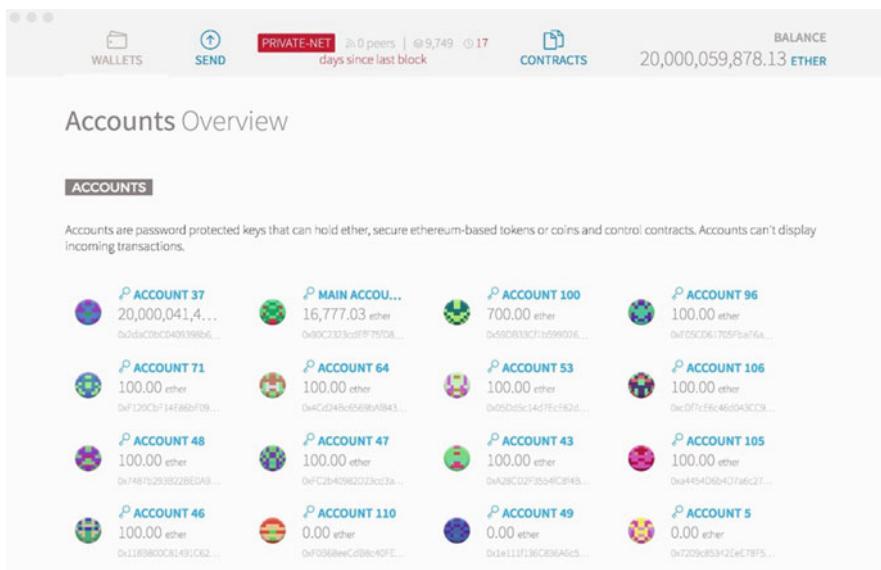


Fig. 2.3 Ethereum wallet interface

Ethereum Wallets

Ethereum Wallet is a standalone application for Ethereum account management. It is open source and allows anyone to submit code to improve the project and manage accounts offline, including account creation, backups, imports, updates, etc. Its most important feature is the ability to perform transactions. Figure 2.3 shows the main interface of the Ethereum Wallet. After the installation is completed, Ethereum Wallet will synchronize the information of the whole network, and after the

synchronization is finished, you can create accounts, set passwords, transfer money, and other operations. However, the project was officially suspended by Ethereum in March 2019 due to serious security vulnerabilities in the software itself. Readers can understand and learn, but it is not recommended to use.

2.1.4 Ethereum Account Management

Accounts are a key to operating Ethereum. External accounts can transact with each other, and each account is defined by a public and private key, indexed by an address. The address is derived from the last 20 bytes of the public key and encodes the key pair in JSON as a private key file. The private key of the account is used to encrypt the transactions sent. All created account information is stored under the keystore in the Ethereum installation directory. Keystore files can be backed up, and users can only perform transactions if they have both the key file and the password. Note that when the private key is lost, it also means that the account for this node is lost. The following describes the management of the account.

Using the geth Command Line

First check the current three accounts.

```
$ geth account list
Account #0: {1301639dc5dcade8fe4672faf1acda67bc14a057}
keystore://Users/zhongweiwei/Library/Ethereum/keystore/UTC--
2017-06-09T07-29- 12.862968835Z--
1301639dc5dcade8fe4672faf1acda67bc14a057
Account #1: {e3e711a6a3bcd13eb108ec2955c41f9c8995fb59}
keystore://Users/zhongweiwei/Library/Ethereum/keystore/UTC--
2017-06-09T07-30- 12.515649743Z--
e3e711a6a3bcd13eb108ec2955c41f9c8995fb59
```

Follow the prompts to create an account by entering your password.

```
$ geth account new
Your new account is locked with a password. Please give a password. Do
not forget this password.
Passphrase: Repeat passphrase:
Address: {d7e01b6aa71d3481f64856a5c013ab2df33c1c86}
```

Checking the account list again, you can see that the account has been successfully created: d7e01b6aa71d3481f64856a5c013ab2df33c1c86.

```
$ geth account list
Account #0: {1301639dc5dcade8fe4672faf1acda67bc14a057}
```

```

 UTC--2017-06-09T07-29-12.862968835Z--1301639dc5dcade8fe4672faf1acda67bc14a057
 UTC--2017-06-09T07-30-12.515649743Z--e3e711a6a3bcd13eb108ec2955c41f9c8995fb59
 UTC--2017-06-09T07-33-36.585818367Z--d7e01b6aa71d3481f64856a5c013ab2df33c1c86

```

Fig. 2.4 Viewing the public and private key files under the keystore

```

keystore:///Users/zhongweiwei/Library/Ethereum/keystore/UTC--
2017-06-09T07-29-12.862968835Z--
1301639dc5dcade8fe4672faf1acda67bc14a057
    Account #1: {e3e711a6a3bcd13eb108ec2955c41f9c8995fb59}
keystore:///Users/zhongweiwei/Library/Ethereum/keystore/UTC--
2017-06-09T07-30-12.515649743Z--
e3e711a6a3bcd13eb108ec2955c41f9c8995fb59
    Account #2: {d7e01b6aa71d3481f64856a5c013ab2df33c1c86}
keystore:///Users/zhongweiwei/Library/Ethereum/keysto

```

After the above operation is complete, you can go to the keystore folder, as shown in Fig. 2.4, which holds the public and private key files for the account. If you create multiple accounts, you will have multiple addresses.

Using the geth Console

Accessing the geth console also allows you to perform the same actions as the geth command line. First, go to the geth console.

```

$ geth console 2>> log_file_output Welcome to the Geth JavaScript
console!

Instance: geth/v1.6.0-stable-facc47cb/10arwin-amd64/go1.8.1
coinbase: 0x1301639dc5dcade8fe4672faf1acda67bc14a057
at block: 0 (Thu, 01 Jan 1970 08:00:00 CST)
datadir: /Users/zhongweiwei/Library/Ethereum
modules: admin:1.0 debug:1.0 eth:1.0 miner:1.0 net:1.0
personal:1.0 rpc:1.0 txpool:1.0 web3:1.0

```

Follow the prompts to create an account by entering your password:

```

$ personal.newAccount()
Passphrase:
Repeat passphrase: "0xfbcc605066317fd4f6fdf33d062d85513e3532c6"

```

A look at all accounts shows that they have been successfully created.

```
$ eth.accounts
["0x1301639dc5dcade8fe4672faf1acda67bc14a057",
 "0xe3e711a6a3bcd13eb108ec2955c41f9c8995fb59",
 "0xd7e01b6aa71d3481f64856a5c013ab2df33c1c86",
 "0xfbcc605066317fd4f6bdf33d062d85513e3532c6"]
```

2.1.5 Ethereum Network

The network protocol of blockchain, most notably the P2P protocol, is a peer-to-peer protocol. In a P2P network, there is no central server, no central router. Each node is peer-to-peer. Each node acts as a server, provides services to other nodes, and also enjoys the services provided by other nodes; that is, it has the functions of information consumer, information provider, and information communication at the same time. The P2P network system is a decentralized, de-trusted, and collective collaboration network system. The Ethereum blockchain has the following characteristics.

1. De-trusting. Because there are no central nodes in Ethereum and trust relationships are established between them through sophisticated cryptography to guarantee the reliability of transactions, the data exchange between all nodes involved in the whole system is trustless.
2. Decentralization. P2P networks are distributed, and the trust relationship between the nodes of the parties to a transaction is entirely free of third-party nodes, solving the strong dependence on central nodes.
3. Data reliability. In the whole Ethereum, each node has a backup copy of the blockchain data, so the modification of data by a single node is invalid, and the erroneous data of this node will be corrected jointly by others, unless it can control 51% of the nodes at the same time to modify the data maliciously, so the more nodes participate in the system and the more computing power, the higher the data security of this system.
4. Collective collaboration. Nodes jointly maintain all blocks in the system with maintenance functions throughout the system, and in public chains these functions are available to anyone. This collective maintenance generally requires incentives to promote network-wide participation, and the incentive mechanism used by Ethereum is mining.
5. Mutual agreement between nodes. In Ethereum, when a new block is added to the chain, the blocks interact with information based on specific commands. Unlike traditional blockchain, blocks in Ethereum will shake hands with each other in symmetric encryption methods to provide some brief. The information of blocks in the chain is synchronized, one of the essential features of Ethereum.

2.2 The Core Principle of Ethereum

As a more mature blockchain platform, Ethereum is trusted by many developers and companies for its security, reliability, and ease of use. The overall architecture is shown in Fig. 2.5.

P2P is generally referred to as peer-to-peer networking or peer-to-peer computing in academic circles. In P2P network, each node (peer) plays three important roles: information consumer, information producer, and communication medium. Unlike the traditional C/S model, the P2P structure is decentralized, robust, scalable, low cost of load, and protects node privacy. The bottom layer of Ethereum mainly includes P2P protocol, which is a protocol without a central server and direct network communication between two nodes. Only based on P2P blockchain can provide decentralized services.

Consensus algorithm, as a core part of the blockchain platform, is the algorithm and strategy for different nodes to reach consensus. Currently, the two most important consensus algorithms in Ethereum are PoW and PoS, which will be introduced in Sect. 2.2.1.

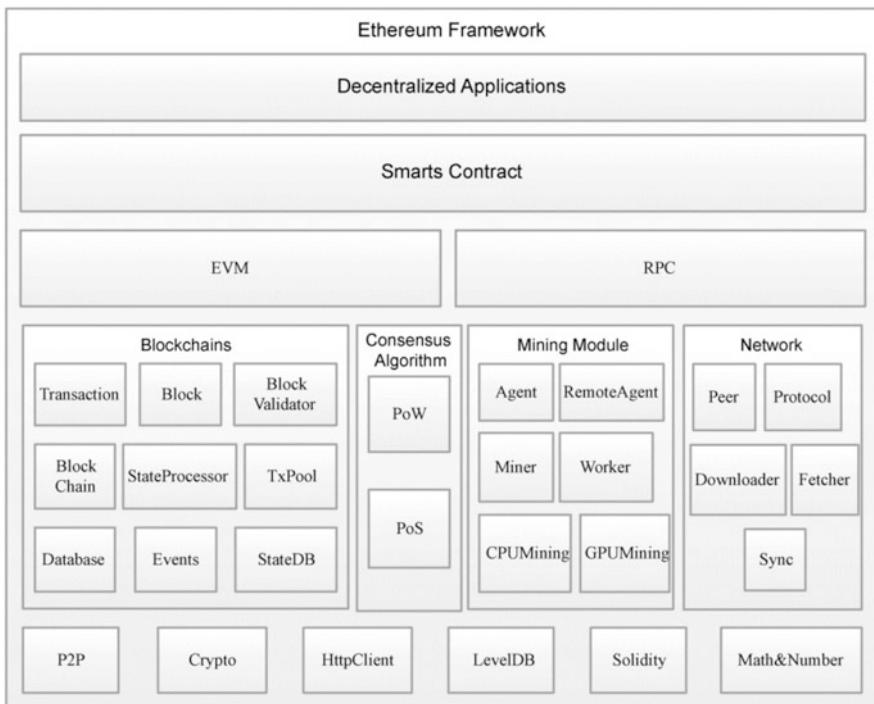


Fig. 2.5 Ethereum overall architecture

EVM, or Ethereum virtual machine, is a container for decentralized applications. Smart contracts can run in EVM after being compiled into bytecode. The detailed execution process of EVM will be summarized in Sect. 2.2.2.

LevelDB is the underlying database of Ethereum, a very efficient key-value database implemented by Google. LevelDB is also used at the base of many enterprise-level blockchain platforms. The data storage of Ethereum will be introduced in Sect. 2.2.3.

A variety of asymmetric encryption and hash algorithms ensure account security and transaction information security on Ethereum platform from the perspective of cryptography, and digital signature and verification signature mechanisms ensure that data is not easy to tamper with. The encryption algorithms involved in Ethereum will be introduced in Sect. 2.2.4.

Solidity is a high-level language for implementing smart contracts, with a syntax similar to JavaScript. Designed to generate Ethereum virtual machine code in a compiled manner, it is Ethereum's recommended flagship language and one of the most popular smart contract languages. Solidity and smart contracts will be explained in depth in Sect. 2.3.

RPC remote procedure call is an interface provided by Ethereum for external access. Upper-layer applications can interact with Ethereum in JSON-RPC mode to call contracts or send Ethereum. All business logic is realized through smart contracts. The Ethereum programming interface can be referred to in Sect. 3.3.

2.2.1 *Ethereum Consensus Mechanism*

The consensus mechanism is a mathematical algorithm for reaching agreement among multiple nodes. In blockchain, the role of the consensus mechanism is particularly important. Since each node in the blockchain is independent of each other and has a full backup of the distributed ledger, how to verify the consistency of these ledger data is the problem that the consensus mechanism needs to consider. In other words, the consensus mechanism is a mathematical algorithm that builds trust between different nodes to gain equity. It allows the associated machines to be connected to work and to function properly despite the failure of some members.

The work abstract model of the consensus algorithm is shown in Fig. 2.6, which is divided into four main phases. The first stage is the contention, the core of the consensus algorithm. How to select the nodes that have bookkeeping rights from all the nodes is also known as “miner mining.” The second stage is block building, in which miners with bookkeeping rights can pack transaction information into blocks according to a specific strategy. The third stage is verification, where the packaged blocks are broadcast to the entire network for verification to ensure the security of the transaction. The fourth stage is uploading where the verified blocks will be appended to the chain.

Common consensus mechanisms include proof-of-workload algorithm, proof-of-stake, proof-of-share authorization and Byzantine fault tolerance, etc. Based on

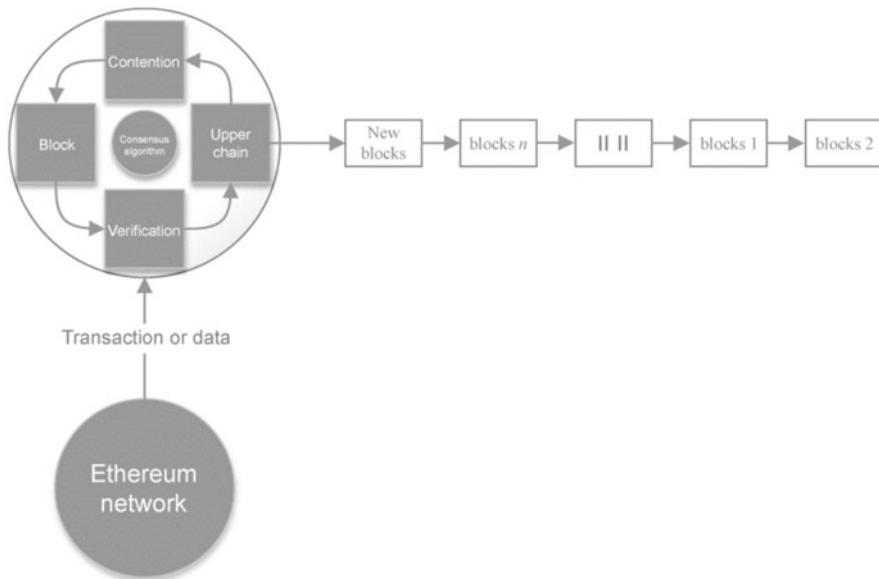


Fig. 2.6 Working model of consensus algorithm

different characteristics like application scenarios and consensus mechanisms, the following dimensions can evaluate the advantages and disadvantages of consensus mechanisms.

1. Compliance regulation: whether it supports super authority nodes to regulate the nodes and data of the whole network.
2. Performance efficiency: the efficiency of reaching consensus and confirming transactions.
3. Fault tolerance: the ability to prevent attacks and fraud.
4. Resource consumption: resource consumption during the consensus process, such as CPU, network IO, storage, and other computer resources.

Six common consensus mechanisms are described below.

1. **PoW:** Proof-of-Work. The right to record transactions is obtained through the speed of mathematical operations of machines. The consensus process requires the entire network nodes to participate in the operations, resulting in high resource consumption, inefficient performance, and weak supervision. In terms of fault tolerance, 50% of the nodes are allowed to make mistakes. This consensus mechanism is currently used in Ethereum.
2. **PoS:** Proof-of-Stake. It is an upgraded version of PoW. According to the proportion of tokens occupied by each node and time, the calculation difficulty is reduced in equal proportion thus speeding up the search for random numbers. PoS shortens the time to reach consensus to a certain extent, but still needs to consume time, essentially does not solve the pain points of commercial

applications, and is similar to PoW in terms of fault tolerance. Ethereum will switch to PoS algorithm afterward.

3. **Dpos:** Proof-of-Share Authority. The difference with PoS is that a certain number of nodes are voted to perform proxy verification and bookkeeping. DpoS drastically reduces the number of nodes involved in verification and bookkeeping and can achieve second-level consensus verification. However, the whole consensus mechanism still relies on tokens, and many commercial applications do not need the tokens.
4. **Paxos:** A consensus mechanism based on the election of a leader. The leader node has absolute authority and allows vital supervisory nodes to participate. High performance and low resource consumption. The election process does not allow malicious nodes and is not fault-tolerant.
5. **PBFT:** Practical Byzantine Fault Tolerance. It is also a consensus mechanism based on the election of leaders. The leader is elected on a permissive, majority-majority basis. It is characterized by allowing Byzantium fault tolerance (33% of nodes are permitted to do evil, and the fault tolerance is 33%), enabling vital supervision nodes to participate, and grading work authority.
6. **Casper FFG:** a hybrid PoW/PoS mechanism, details of which will be described in Sect. 2.4.3.

PoW and PoS algorithms are related to Ethereum. The Ethereum project is divided into four phases: Frontier, Homestead, Metropolis, and Serenity. In the first two phases, the Ethereum consensus algorithm adopts PoW. In the third phase, Ethereum will transition from PoW to PoS, and in the fourth phase will move to PoS.

PoW is that if a node pays enough computing power, then the blockchain will consider the blocks found by this node to be valid. Bitcoin was the first blockchain project to adopt PoW bookkeeping, where miners compete for the right to keep track of bitcoins through computer computing power. Whenever a bitcoin block is generated, the system rewards the first miner to successfully keep track of it with a certain amount of money. The purpose of proof-of-work is to create the next block complex thus discouraging attackers from maliciously generating blockchains. PoW is a SHA256 hash of each block, treating the resulting hash as an unpredictable value of 256 bits in length that is guaranteed to be less than a target value constantly and dynamically adjusted. If the current target value is 2^{192} , it takes an average of 2^{64} attempts to generate a valid block. In general, the Bitcoin network resets the target value every 2016 block, ensuring that a block is developed every 10 min on average. Ethereum generates blocks in about 14 s.

PoS, a consensus algorithm primarily used in a public blockchain, replaces proof-of-work. It can be described as a type of virtual mining that relies on the blockchain's tokens. In PoS, users are rewarded by purchasing tokens of equal value as a deposit to be placed in the PoS mechanism for the chance to generate new blocks. The general process can be described as a collection of token-holding users who put their tokens into the PoS mechanism, and these users become verifiers. Suppose in the latest block of the blockchain, the PoS algorithm randomly selects among these

verifiers based on the weight, which is based on the number of tokens invested by the verifier, e.g., if one user deposits 1000 tokens and another puts in 100 tokens, the former user is 10 times more likely to be selected than the latter one. After the user is selected, they are given the right to generate the next block. If this designated verifier does not create the block within a specific time, a second is selected to develop the block instead.

The PoW consensus mechanism is already very mature, but its implementation consumes a lot of power costs. PoS, although still in the development stage, has many advantages in terms of efficiency and cost, and no longer needs to consume a lot of power for safe block generation. PoS will rapidly develop in the coming period.

2.2.2 *Ethereum Virtual Machine*

An Ethereum Virtual Machine (EVM) is an environment that runs smart contracts on each node, acting as a separate sandbox with tightly controlled access; contract code cannot touch the network, files, or other processes while running in EVM. EVM module is mainly divided into three modules: contract compilation module, Ledger module, and EVM execution module.

The compile contracts module is primarily a wrapper around the underlying Solc compiler, providing an RPC interface to external services to compile smart contracts written in Solidity. After compilation, it will return the binary code and the corresponding contract ABI, which can be interpreted as the manual of the agreement. You can know the method name, parameters, return value, and other information about the contract.

The Ledger module is mainly for modifying and updating the blockchain account system. There are two types of accounts, ordinary accounts and smart contract accounts, and the caller can call the contract if he knows the contract account address. Every modification of the account will be persisted to the blockchain.

The EVM execution module, as the core module, is mainly responsible for parsing and executing the smart contract code in the transaction, generally divided into two parts: creating contract and calling contract. The EVM execution module supports ordinary bytecode execution and JIT mode instruction execution to improve efficiency, which mainly executes instructions directly on the compiled binary code. In contrast, the JIT mode optimizes instructions during the performance, such as packing consecutive push instructions into a slice to facilitate the efficient execution of the program. The flow is shown in Fig. 2.7.

1. EVM receives Transaction information and determines whether the Transaction type is deployment contract or execution contract. If the contract is deployed, execute the instruction sets to store the contract address and compiled code; if the contract is executed or called, EVM is used to perform the input instruction sets.

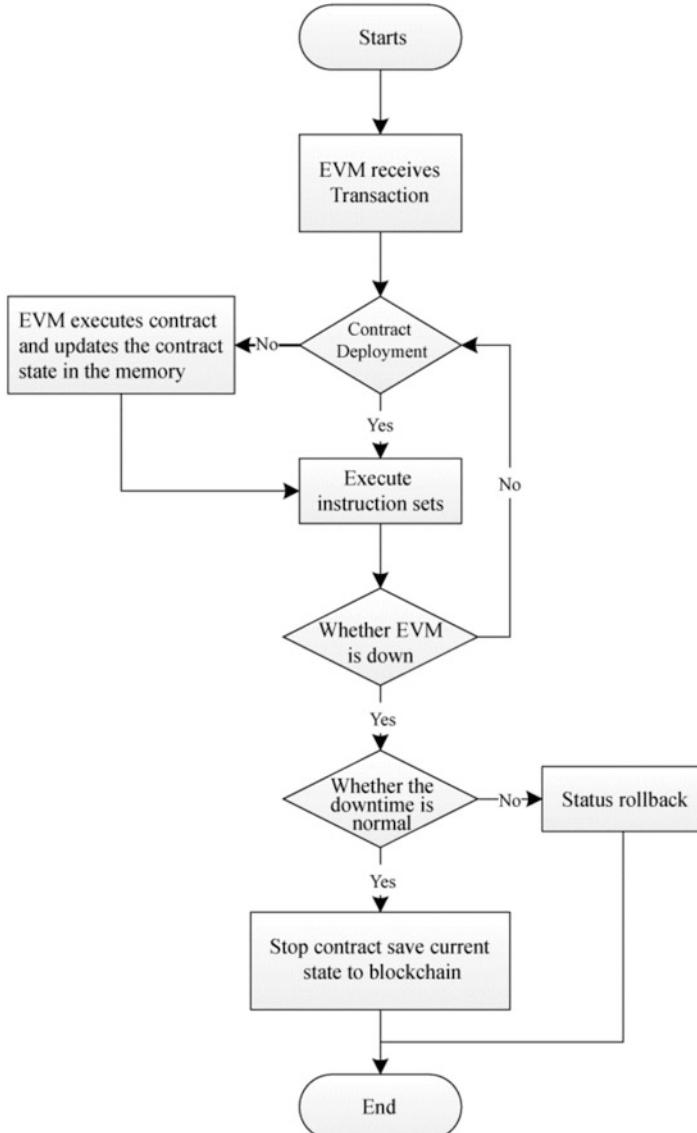


Fig. 2.7 EVM module execution flow

2. After the execution of the last instruction set, judge whether EVM is shut down. If it is shut down, consider whether it is shut down normally. If yes, update the contract state to the blockchain; otherwise, roll back the contract state. If the machine does not stop, go back to the previous step (1) for judgment.

3. The completed contract will return an execution result, which will be stored in the Receipt by EVM. The caller can query the result through the hash of the Transaction.

2.2.3 Ethereum Data Storage

The structure for storing data in Ethereum is the Merkle Patricia Tree. As the data structure, Merkle Patricia Tree, proposed by computer scientist Ralph Merkle, is used in the Bitcoin network for data correctness verification. Ethereum combines and optimizes Merkle Tree and Patricia Tree.

Merkle trees summarize all the transactions in a block in the Bitcoin network while generating a digital fingerprint of the entire set of transactions. The Merkle tree is constructed bottom-up. In Fig. 2.8, the four cells of data, L1 ~ L4, are first hashed, and then the hashes are stored into the corresponding leaf nodes, namely, Hash 0-0, Hash 0-1, Hash 1-0, and Hash 1-1.

Combine the hashes of two adjacent nodes into one string, then calculate the hash of this string, and what you get is the hash of the parent of the two nodes. (If the total number of hashes at the bottom is singular, then a single hash must appear at the end, in which case the hash operation is performed directly on it, and the hash value obtained is the hash value of its parent node.)

The above process is repeated to obtain the hash value of the last node, and the hash value of this node is used as the Merkle root of the whole tree. If two trees have

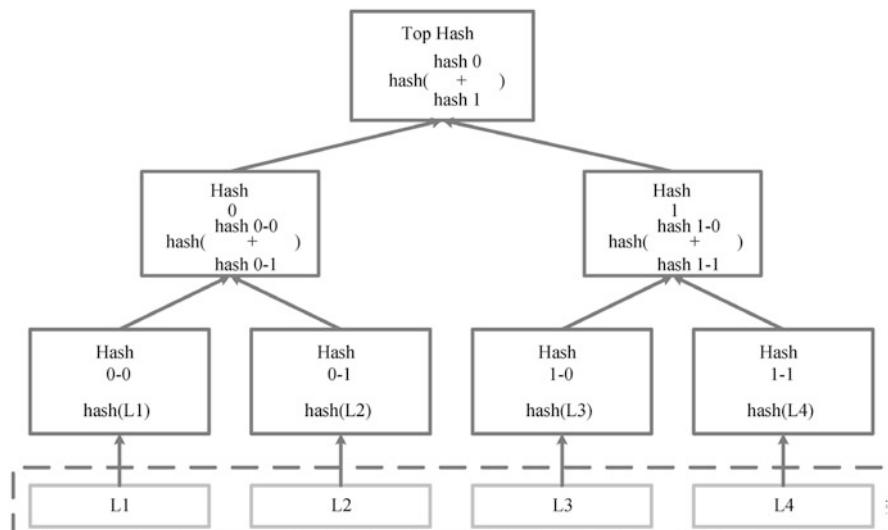


Fig. 2.8 Merkle tree structure

the same Merkle root, the structure of the two and the content of each node must be the same.

An obvious advantage of Merkle trees is that the verification process can quickly locate the location of “incorrect” data. Since a slight difference in the original data can cause the computed hash to be incorrect, comparing the two Merkle trees makes it easy to find this path and thus locate the location of the “incorrect” node.

Because of a single application in Bitcoin, Merkle tree cannot perform proofs involving the current state (such as the holding of digital assets, name registration, and the form of financial contracts), though it can prove contained transactions. The Merkle tree in Ethereum mainly plays the role of building the tree, while it is very inconvenient for procedures such as changing and inserting the contents of tree nodes. Ethereum has modified the Merkle tree and incorporated the Patricia tree to extend these operations.

The Patricia tree stores the status of each account. The tree is built by starting at each node, dividing the nodes into as many as 16 groups, then hashing each group, and continuing to hash the results until there is a final “root hash” for the whole tree. Patricia trees are easy to update, add or remove tree nodes, and generate new root hashes. An example of an update using the Patricia tree is shown in Fig. 2.9.

1. Pass in the root node as the current processing node and the Key of the target node as the path.
2. Pass in the new Value. If the Value is null, find the node and delete it; conversely, establish a new node to replace the old one.

2.2.4 Ethereum Encryption Algorithm

Ethereum uses a variety of encryption algorithms, most notably hashing and asymmetric encryption.

The hash algorithm is used to verify the identity of a user in Ethereum quickly, and its principle is to transform a piece of information or text into a string (digest) with a fixed length. The characteristic of the hashing algorithm is that if two pieces of information are identical, the final encrypted string will be identical; if two pieces are not identical, even if only one character is different, the final string will be very messy and random, and there is no correlation between the two. The current mainstream hash algorithm is Secure Hash Algorithm (SHA), a family of cryptographic hash functions designed by the National Security Agency (NSA)¹ and published by the National Institute of Standards and Technology (NIST)² as a series

¹The National Security Agency (NSA) is a national-level intelligence agency of the United States Department of Defense, under the authority of the Director of National Intelligence.

²The National Institute of Standards and Technology (NIST) is a physical sciences laboratory and nonregulatory agency of the United States Department of Commerce. Its mission is to promote American innovation and industrial competitiveness.

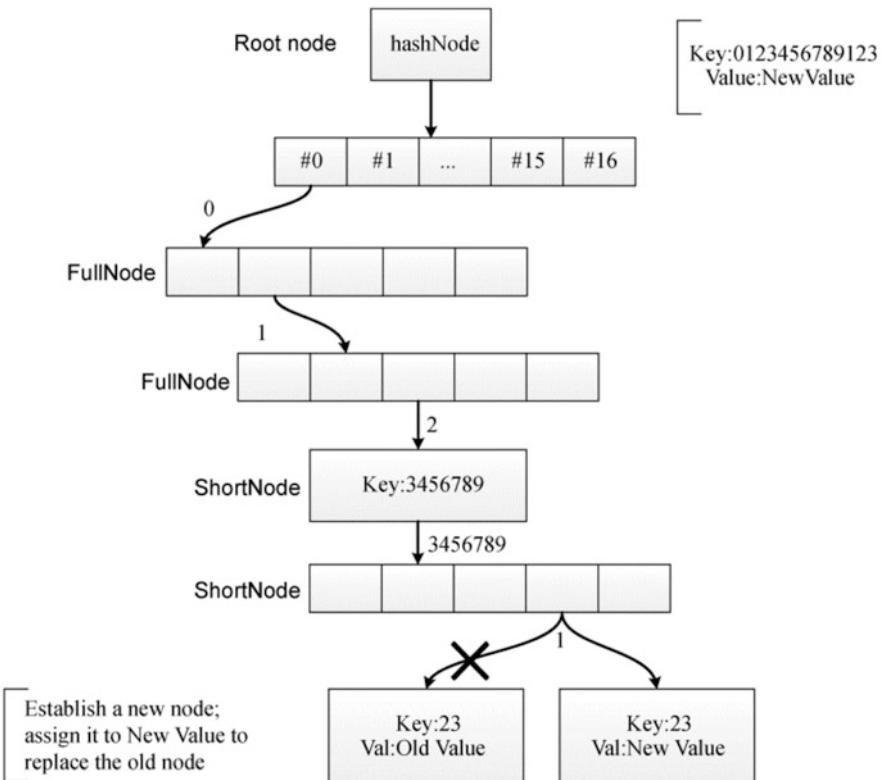


Fig. 2.9 Example of Patricia tree update

Table 2.1 Comparison of different hashing algorithms

Algorithm category	Security	Output size (bit)	Operation speed
SHA1	Middle	160	Middle
SHA256	High	256	Slightly lower than SHA1
SM3	High	256	Slightly lower than SHA1
MD5	Low	128	Fast

of cryptographic hash algorithms, a security algorithm validated by the US Federal Information Processing Standards.³ SHA includes a number of variants, such as SHA-1, SHA-224, SHA-256, and SHA-384. Currently, the Ethereum blockchain mainly uses SHA256 for mining operations. The characteristics of different hash algorithms are shown in Table 2.1.

³The Federal Information Processing Standards (FIPS) are a set of US Government security requirements for data and its encryption.

Table 2.2 Comparison of different asymmetric encryption algorithms

Encryption algorithm	Security	Maturity	Operation speed	Resource consumption	Key length	
					Level	Length (bit)
RSA	Low	High	Slow	High	80	1024
					112	2048
ECC	High	High	Middle	Middle	80	160
					112	224
SM2	High	High	Middle	Middle	80	160
					112	224

Asymmetric encryption is an encryption method that consists of a set of unique key pairs, including public and private keys. Only the user's public key is required to encrypt user information and implement secure information exchange, but the user's private key must decrypt the data. Only those who know the unique users' private key corresponding to the user's public key can access the information, and no unauthorized user (including the sender) can decrypt the data. In Ethereum, every transaction a user sends is signed using a private key, and the blockchain then uses a public key to verify the signature. When the verification is passed, the transaction is legitimate, and then it can be persisted to the blockchain. The public-private key pairs of users in Ethereum are generated using the ECC elliptic curve encryption algorithm. The characteristics of asymmetric encryption algorithms are shown in Table 2.2.

2.3 Ethereum Smart Contracts

This section will discuss how to write a smart contract using Solidity in detail. It will start with a brief introduction to Solidity and then detail the basic syntax of Solidity such as data types, state variables, and function calls. Also, Solidity, being an object-oriented language, will have the corresponding object-oriented nature. After writing a smart contract, the most important thing is the testing of the smart contract. This section will test the contract with different testing frameworks as a way to verify that the contract meets expectations. Finally, an example analysis of a more complex smart contract is given to help readers better understand smart contracts.

2.3.1 Introduction to Smart Contracts and Solidity

What is a smart contract?

Nick Szabo first proposed the concept of smart contracts in the 1990s. In the article *The Idea of Smart Contracts*, Szabo argues that vending machines are the

most primitive application of smart contracts. Although the concept was born almost simultaneously with the Internet, they were not used in natural industrial production on a large scale for a long time due to the lack of a trustworthy execution environment. It was not until Bitcoin emerged that people began to realize that Bitcoin's underlying technology, blockchain, and smart contracts were a match made in heaven. Ethereum is the most typical example. To put it simply, Ethereum can be understood as blockchain + smart contracts.

A contract collects code (logical description) and data (state representation) stored at a specific address on the Ethereum blockchain. Contract accounts are able to pass information between each other, perform Turing-complete operations, compile into EVM byte code (an Ethereum-specific binary format), and run on the blockchain. In other words, smart contracts are modular, reusable, and automatically executable scripts that run on the blockchain. Contracts are deployed by storing the bytecode obtained by compiler compilation on the blockchain, corresponding to a storage address that will be given. When a predetermined condition occurs, a transaction is sent to that contract address and the nodes across the network execute the opcode generated by the compilation of the contract script, and finally the execution result is written to the blockchain. Therefore, a smart contract can be understood as all the business logic code that performs operations on the blockchain.

An important feature of smart contracts is Turing-completeness. A Turing-complete system means a computational system that can compute every Turing-computable function, and Turing-completeness gives scripting systems the ability to solve all computable problems. Smart contracts are Turing-complete, i.e., they can do all the things that a Turing machine can do. In layman's terms, all the logical operations that can be done in a general programming language can be implemented in a smart contract.

Another important feature of smart contracts is sandbox isolation. Restrictions are placed on I/O, network operations, access to other processes, etc., effectively wholly isolated. Therefore, the currently implemented smart contracts cannot read and write files, enable access to network resources or provide network services directly. Smart contracts can only use the interface provided by the blockchain platform for accessing contract data, i.e., accessing data and methods in the smart contract, after it has been deployed to the blockchain platform. This feature improves the security of smart contract execution.

Smart contracts can be written in Solidity, Serpent, LLL, Mutan, but Solidity is the most widely used and popular language.

Solidity is a high-level programming language designed to write smart contracts, contract-oriented. Solidity's syntax is similar to a high-level object-oriented language like JavaScript, which is also a statically typed language and runs on top of the EVM. Solidity supports inheritance, libraries, and complex custom types. The file extension is .sol, a genuinely decentralized contract that runs on the web. Solidity currently has an online live compiler for developer convenience. There is also support for a wide range of standard library functions with the following features.

- The underlying Ethereum is account-based, so there is a particular Address type for locating users, locating contracts, and locating the contracts code (the contract itself is also an account).
- The framework embedded in the language is payment-capable, so it is possible to use some keywords, such as Playable, to make direct transaction payments at the language level.
- By using blockchain storage on the network, every state of the data can be stored permanently, so it needs to be determined whether the variables use memory or the blockchain.
- Once an exception occurs, all executions will be rolled back, mainly to ensure the atomicity of contract execution to avoid data inconsistency in intermediate states.
- Because Solidity runs on a decentralized blockchain network, function calls during runtime translate into the operation of nodes in the network; there is an emphasis on how functions are executed and how contracts are called.

Contracts for applications such as voting, crowdfunding, closed auctions, multi-signature wallets, and more can be easily created using Solidity today. Here are some of the official development tools introduced by Solidity.

1. Browser-Based Compiler

It is Solidity's official and highly recommended browser-based online IDE, also known as Remix. Browser-Based Compiler already integrates with the compiler and Solidity runtime environment, and does not require any server-side components.

2. Visual Studio Extension

A Solidity plugin for Visual Studio that provides a Solidity compiler.

3. Package for Sublime Text-Solidity language syntax

A Solidity syntax highlighting tool for the Sublime Text editor.

4. Etheratom

Solidity's recommended plugins on Atom.

5. Emacs Solidity

Plugins for Emacs editor for syntax highlighting, compilation, and error reporting.

6. Vim Solidity

Plugins for Vim editor and provides syntax highlighting.

7. Mix IDE

A Qt-based contract integrated development environment for designing, debugging, and testing smart contracts written by Solidity is described in detail in Chap. 3 of this book. However, it is no longer updated for maintenance and is less used in development.

8. IntelliJ IDEA plugin

Solidity plugin available for all other JetBrains IDEs.

Developers can use one or more of the above tools for Solidity-based smart contract development, depending on project requirements and their preferences.

2.3.2 Smart Contract Programming and Deployment

This section will start with a simple Hello World smart contract program to learn the essential components of a smart contract such as data types, method calls, events, and how to deploy the finished smart contract to the Ethereum environment. It automatically executes the logic in that contract. Here is this simple Solidity-Hello World example.

Solidity-Hello World

Solidity operates within the Ethereum environment and has no prominent “outputting” strings, so here we use logging events to put strings into the blockchain. Hello World’s contract implementation is as follows. Each time a contract is created, the constructor is called and the contract creates a log entry on the blockchain, printing the Hello World parameter.

```
contract Hello {
    function World() public pure returns(string memory) {
        return "Hello World!";
    }
}
```

The contract is successfully compiled by running the code interface in the online development tool Remix, as shown in Fig. 2.10. Figure 2.11 shows the execution structure, which successfully receives “Hello World!” from the event.

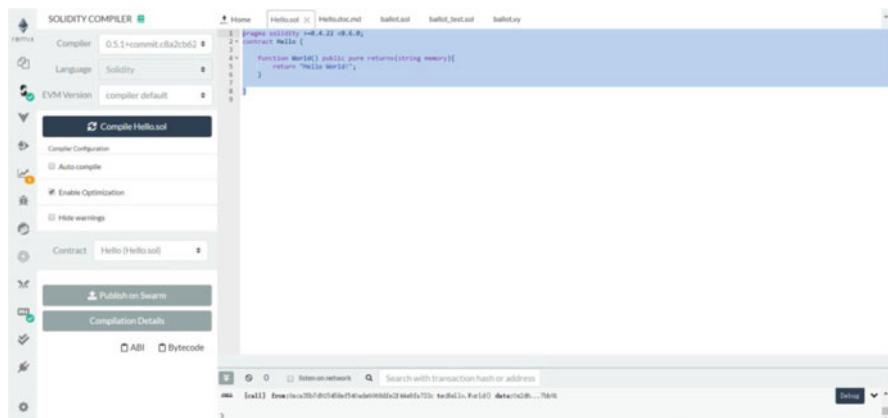


Fig. 2.10 Remix compiles Hello World

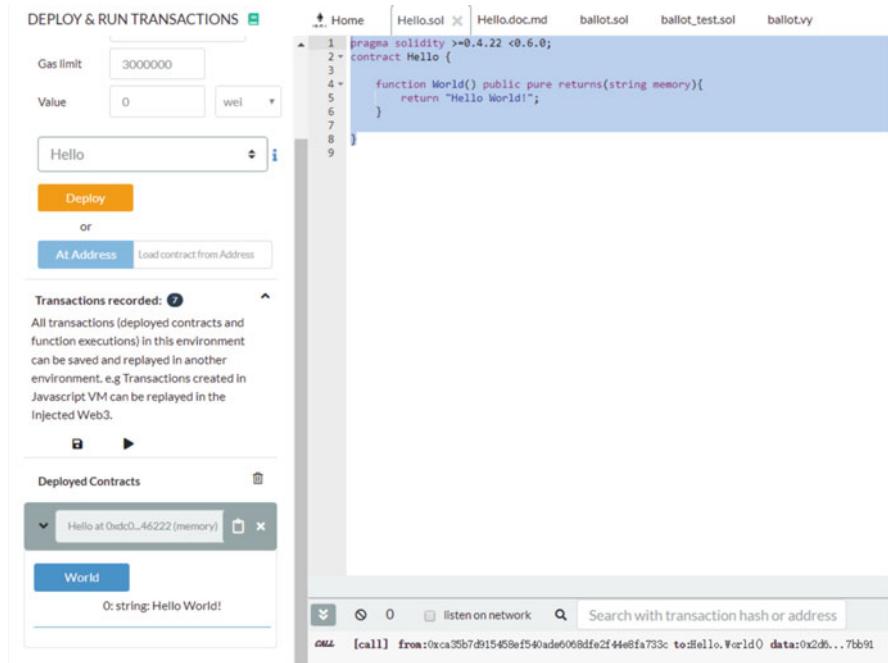


Fig. 2.11 Remix executes Hello World

Document Layout and Contract Structure

The Solidity contract is similar to the definition of classes in an object-oriented language. Each contract covers state variables, functions, function modifiers, events, struct types, enumeration types, and annotations. In addition, contracts can inherit from other contracts. The roles of the different components of a contract are as follows.

1. Compiled version declaration

Generally, the online compiler will ask for the Solidity compiler version number to be specified; otherwise, it will warn. Currently, it primarily has to be higher than version 0.4 to compile with the following declaration statement.

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

2. Status variables

They are used in contracts to permanently store the value of a variable by data types such as uint, int, bytes32, string, and address to modify a state variable.

3. Functions

A function is an executable block of code in a contract, modified with the function keyword, and can have zero or more input parameters and zero or more return values.

4. Function modifiers

It is possible to supplement the semantics of a function when declaring it such as using constant, internal, external, public, and private to modify a function. Functions with different modifiers have different access rights.

5. Events

Events are the interface to EVM logs, declared using events, and have one or more return values. It can be called within a function and captured in the JavaScript API.

6. Structures.

A structure is a custom set of composite data types containing multiple state variables of different data types. Structs are used for the description of an object.

7. Enumeration

An enumeration is a data type used to create a set of named values, declared using enum, and Solidity's underlying treatment of enums is the uint type.

Data Types and State Variables

Solidity is a statically typed language where every variable needs to be defined at compile time (including global or local variables). Solidity also provides many base data types that can be composed into complex types. A value type variable that permanently assigns a value, either as a function parameter or a copy of a new value in an assignment. Value types include booleans, integers, addresses, byte arrays, rational numbers and integers, hexadecimal literals, enumerated types, and functions.

Type of Boolean

Boolean types are only available as true or false values. Operators include: ! (logical not), && (logical with), || (logical or), == (equal), != (unequal). The regular short-circuiting rules also apply to the || and && operators: for the expression $f(x) \parallel g(y)$, if $f(x)$ is true, then $g(y)$ need not be computed because the result must be true; for the expression $f(x) \&\& g(y)$, if $f(x)$ is false, then $g(y)$ need not be computed because the result must be false. Boolean types are used essentially as in the other high-level programming languages.

Integer Type

int and uint are signed and unsigned integers, respectively, including uint8 to uint256 with steps of 8 (unsigned integers from 8 to 256 bits). uint and int are aliases for uint256 and int256, respectively.

Operators that use integers are the comparison operators <=, <, ==, !=, >=, >; the bitwise operators & (with by bit), | (or by bit), ^ (different or by bit), ~ (inverse by bit); and the arithmetic operators +, unity, unity+, *, /, % (take remainder). The use of integers is shown below.

```
pragma solidity >= 0.4.22 < 0.6.0; contract IntExample{
    uint value;
    // Two uint values are added together
    function add(uint x, uint y) public returns (uint) {
        value = x + y;
        return value;
    }
    // Division of two uint values, same rules as in C
    function divide() public returns (uint) {
        uint x = 3;
        uint y = 2;
        value = x / y;
        return value;
    }
}
```

Address

An Ethereum address has 20 bytes, and the address variable is modified with the address. You can use address(0) to represent an empty address, i.e., 0×0. address can modify a contract address or an account address. Address can also be manipulated with the compare operator. By default, the address type variable has two critical methods: account balance () and send (). When using the send () method, the execution has some caveats, such as the recursion depth of the call cannot exceed 1024; if there is not enough gas, the execution will fail, so after using the send() method you need to determine if it was successful based on the return value.

Balance is a quick index of member variables of type address, as shown in the following code to query the balance.

```
pragma solidity >= 0.4.22 < 0.6.0;
contract AddressExample{
    bool isSuccess;
    // Access to account balances
    function getBalance(address account) public returns (uint) {
        return account.balance;
    }
}
```

Account is a contract address, and when the account.balance function is executed, the contract code triggers execution. We can either check the account balance, or simply use this.balance to check the balance.

Byte Arrays

bytes1, bytes2, bytes3, ..., bytes32 (bytes1 is equivalent to byte) are all.

Fixed-length byte arrays. Byte arrays can also use the comparison and bit operators. By default, the length of this byte array is represented by the read-only member variable.length.

Bytes and string are two dynamic length byte arrays. Bytes is primarily used to represent byte data of arbitrary length, and string represents UTF-8 encoded string data of arbitrary length. If the length can be determined, try to use a fixed length, such as bytes1 to bytes32, as this takes up less space.

String

A string literal is a string caused by single or double quotes. Solidity strings, unlike C language, do not contain an end, and the size of the string “foo” contains only three bytes. The string itself is an array of bytes of indefinite length, which can be implicitly converted to bytes1, ..., bytes32. String literals support escape characters, such as “\n”.

```
pragma solidity >= 0.4.22 < 0.6.0; contract StringExample{
    string name;
    function stringTest() public{
        name = "Jack";
    }
}
```

Arrays

The size of an array can be determined at the compile stage (static arrays) or can be of indefinite length (dynamic arrays). For memory arrays, the member type is arbitrary (it can also be other arrays, mappings, or structures). When using a memory array inside a function, if the function is visible to the outside, the array elements cannot be of mapping types and can only be of types that support ABI. A fixed-length array with length k and element type T can be represented as T[k], and another variable-length array as T[].

The types bytes and string are essentially unique arrays, with bytes being similar to byte[]; string is more identical to bytes but does not provide access by length or by the index. Bytes is relatively more space-efficient.

The array has the following member functions.

`length()`: length field, indicating the number of elements held. Dynamic arrays can be resized if they are of the memory type. It can be done by adjusting the corresponding value via. `length`.

`push()`: both dynamically typed arrays and bytes have a `push()` function that adds new elements and returns the result as the length of the new array or bytes.

```
pragma solidity >= 0.4.22 < 0.6.0; contract ArrayExample {
    uint[] uintArray;
    bytes b;
    function arrayPush() public returns (uint) {
        uint[3] memory a = [uint(1), 2, 3];
        uintArray = a;
        // Add to the last position of the array
        return uintArray.push(4);
    }
    function bytesPush() public returns (uint) {
        b = new bytes(3);
        return b.length;
    }
}
```

Mapping/Dictionary

A mapping is defined as `_KeyType=>_KeyValue`. Keys cannot be of type mapping, struct, array, etc., while values are of type unlimited. A mapping can be considered as a hash table obtained by initializing all keys with default values corresponding to some type. However, unlike a hash table, a key stores only its Keccak-256 hash, used to find its associated value. As a result, mapping lookups are fast, and there is no notion of length or order.

Mapped types can only be used to define state variables, or as references to memory types in internal functions. Mapped value types can certainly also be mapped and provide key-value recursive access. The following example uses `msg.sender` contract caller as a key type and `amount` as a value type for storage.

```
pragma solidity >= 0.4.22 < 0.6.0;
contract MappingExample{
    mapping(address => uint) public balances;
    uint balance;
    function updateBalance(uint amount) public returns (uint) {
        // Set the value in mapping
        balances[msg.sender] = amount;
        // Fetch the value in mapping
        balance = balances[msg.sender];
        return balance;
    }
}
```

An important part of the contract is the state variables, whose values are permanently stored in the storage space of the contract, i.e., in these state variables. Different types of state variables can be declared using the data types above to store different types of data, as shown below.

```
pragma solidity >= 0.4.22 < 0.6.0;
contract TypeExample{
    uint uintValue;
    bool boolValue;
    address addressValue;
    bytes32 bytes32Value;
    uint [] arrayValue;
    string stringValue;
    enum Direction{Left, Right}
    mapping(address => uint) mappingValue;
}
```

A contract's variables (state variables or local variables) or function parameters have an essential concept—data storage location. In general contracts, the default storage location for variables is set as follows: global state variables and local variables exist in storage; function arguments and returned parameters are stored in memory. The keywords `storage` and `memory` can be used to override the default setting.

Variable assignments produce different results depending on where the data is stored. Assigning values in memory, storage, and state variables always creates a completely unrelated backup. Assigning a value to a local storage variable is given a reference, and even if that value changes, it still points to the corresponding state variable.

Function Calls

Functions are the most important part of a programming language, enabling modular programming and significantly improving the efficiency of project development and code readability and reusability. Functions in Solidity are similar to functions in JavaScript.

Internal Calls

In the same contract, as the current one, functions can be called directly internally or recursively. These functions are translated in the EVM as simple JUMP instructions.

```
pragma solidity >= 0.4.22 < 0.6.0;
contract FunctionExample {
    function g(uint a) public returns (uint) {
        // Calling functions in the same contract
```

```

        return f(a);
    }
    function f(uint a) public returns (uint) {
        return a;
}

```

External Function Calls

An external call must be used for functions with different contracts instead of a direct call via JUMP. For an external call, all parameters of the function must be copied to memory.

If the called contract does not exist, if the called contract generates an exception, or if there is insufficient gas, all function calls will have an exception and the impact on the ledger during the execution will be rolled back.

```

pragma solidity >= 0.4.22 < 0.6.0;
contract User {
    function age() external returns (uint) {
        return 25;
    }
}
contract FunctionExample {
    User user;
    function exfunction() public{
        // Get the contract address
        address addr = address(new User());
        // Force type conversion to get contract objects
        user = User(addr);
    }
    function callUser() public{
        // Calling methods using contract objects
        user.age();
    }
}

```

Named Parameter Calls

Parameter to function calls can be called by specifying names, passing in arguments in any order, including {}. However, the type and number of parameters should be consistent with the definition.

```

pragma solidity >= 0.4.22 < 0.6.0;
contract FunctionExample {
    function add(uint first, uint second) public returns (uint) {
        return first + second;
    }
    function callAdd() public returns (uint){

```

```
// The order in which parameters are passed in can be inconsistent
return add({second: 2, first: 1});
}
```

Exceptions are encountered in some functions, and an exception needs to be thrown manually using the throw instruction. The effect of the exception is that the currently executing call is stopped and the state is rolled back (i.e., all state and balance changes do not occur). In Solidity, exceptions cannot be caught externally.

Function Visibility

Functions can be marked as external, public, internal, or private, where the default is public.

- **external:** External functions are part of the contract interface to initiate calls from other contracts via transactions. An external function, f(), cannot be called directly by way of f(), but rather by way of this.f() external access. External functions are more efficient when receiving large arrays of data.
- **public:** public functions are part of the contract interface and can be called internally or via messages. Public type is the modifier with the most significant access rights.
- **internal:** internal functions can only be called via internal access (e.g., in the current contract) or within an inherited contract. Note that the prefix this cannot be added.
- **private:** private functions are only accessible within the current contract, not within the inherited contract. Private is the modifier with minor access rights.

It is important to note that everything within a contract is visible to external observers. Marking something as private only prevents other contracts from accessing and modifying it but does not prevent others from seeing the information in question. The visibility identifier is typically defined in the middle of the argument list and the return keyword.

```
pragma solidity >= 0.4.22 < 0.6.0;
contract FunctionExample {
    function func(uint a) private returns (uint b) { return a + 1; }
    function setData(uint a) internal { data = a; }
    uint public data;
}
```

Pure/View Functions

Functions can also be declared as pure/view functions, and such functions need to be guaranteed not to change any values, i.e., not to cause a change in the state of the

blockchain. Such functions do not consume gas and can be modified with pure/view. The underlying call() function is actively called when this class of functions is invoked. The purpose of the pure/view declaration is to tell the compiler that the function does not change or read state variables and does not require miner acknowledgement at runtime. The pure statement has more restrictions than view, such as no access to member variables and balance.

```
pragma solidity >= 0.4.22 < 0.6.0;
contract FunctionExample {
    uint a;
    // Generally declare the get method of a state variable as view
    function getA() view public returns(uint) {
        return a;
    }
}
```

Events

Events make it easy to use the built-in functionality of EVM logging by calling the JavaScript callbacks of the DApps that have listened to the corresponding events. When calling an event, it triggers parameters stored in the transaction's log, associated with the contract address and merged into the blockchain. They remain for as long as the block is accessible. Logs and events are not directly accessible within the contract, not even the contract that created the log. In developing DApps, events may need to be used frequently.

Events have up to three parameters set to indexed, allowing the corresponding parameter values to be retrieved in the Ethereum VM log. The corresponding value is found by entering the corresponding index when the user calls the interface. If an array (both string and bytes) type is marked as an indexed item, the value looked up is a Keccak-256 hash of that indexed value. All parameters that are not indexed will be saved as part of the log. Example event contract code is as follows.

```
pragma solidity >= 0.4.22 < 0.6.0;
contract EventExample {
    // Declare events, which can have multiple parameters
    event SendBalance(
        address indexed _from,
        bytes32 indexed _id,
        uint _value
    );
    // Implementing functions
    function sendBalance(bytes32 _id) public payable{
        // Any call to this function will trigger the SendBalance event and
        // be detected by the JavaScript API
        emit SendBalance(msg.sender, _id, msg.value);
    }
}
```

Using the JavaScript API to get the logs.

```
var abi = /* compile and get abi*/;
var EventExample = web3.eth.contract(abi);
var eventExample = EventExample.at(0x123 /* address */);
var event = eventExample.sendBalance();
// Listens for the SendBalance event to be called
event.watch(function(error, result){
    // The result will contain multiple parameters
    if (!error)
        console.log(result);
});
// Another way to call an event, start listening directly after the call
var event = eventExample.sendBalance(function(error, result){
    if (!error)
        console.log(result);
});
```

Special Variables

There are a number of variables and functions in Solidity that can be used globally, including Ethereum “currency units,” time units, block and transaction properties, math and cryptographic functions, and more.

Ethereum “Currency Unit”

A number can be suffixed to indicate an Ethereum “currency unit,” such as wei, finney or ether, and the units can be converted between them. If no suffix is added to the number of Ethereum, the default unit is wei. “Currency unit” can also perform logical operations, e.g., `2 ether == 2000 finney`. The result of this expression is true.

```
pragma solidity >= 0.4.22 < 0.6.0;
contract EtherExample{
    function isEqual() public returns (bool) {
        if (2 ether == 2000 finney) {
            return true;
        }
        return false;
    }
}
```

Time Units

Numbers can be suffixed with seconds, minutes, hours, days, weeks, and years and converted to each other by default in seconds. These suffixes cannot be used to

declare variables and need to be converted to constants if they are to be passed in. now is the current timestamp.

```
pragma solidity >= 0.4.22 < 0.6.0;
contract TimeExample{
    function f(uint start, uint daysAfter) public{
        if (now >= start + daysAfter * 1 days) {
        }
    }
}
```

Block and Transaction Attributes

The global scope has block and transaction attributes that provide current information about the blockchain.

```
block.blockhash(bytes32) : the hash value of the specified block. block.
coinbase(address) : the address of the current miner. block.difficulty
(uint) : the difficulty of the current block.
block.gaslimit(uint) : the current block's gas value.
block.number(uint) : the block number of the current block. block.
timestamp(uint) : timestamp of the current block.
msg.data(bytes) : the complete call data (calldata).
msg.gas(uint) : the current remaining gas value.
msg.sender(address) : the address of the originator of the call.
msg.sig(bytes4) : the first 4 bytes of calldata (i.e. the function
identifier).
msg.value(uint) : the number of Ethereum coins in the message sent.
now(uint) : the current block timestamp.
tx.gasprice(uint) : the gasoline price of the transaction.
tx.origin(address) : the sender of the transaction.
```

Contract Deployment

In natural project development, there are many ways to implement contract deployment. Such as deploying using the Console command line by calling the native RPC interface, deploying using the Console command line by calling the JavaScript API (web3.js), or automating deployment using the Truffle framework.

Here is a compiled contract deployment using a JavaScript script, calling the same interface as web3.js. Using this approach requires a bit of JavaScript programming basics. The Multiply7 contract is used here to test it. The contract code is as follows.

```
pragma solidity >= 0.4.22 < 0.6.0;
contract Multiply7 {
    event Print(uint);
    function multiply(uint input) public returns (uint) {
```

```

        emit Print(input * 7);
        return input * 7;
    }
}

```

Save the contract as a file with the sol suffix, such as Multiply7.sol, into the directory where the script file is located, and turn on the Ethereum private chain for mining. Write the deployment script AutoDeploy.js as follows.

```

var Web3 = require('web3');
var fs = require('fs');
// web3 initialization
var web3;
if (typeof web3 !== 'undefined') {
    web3 = new Web3(web3.currentProvider);
}
else {
    web3 = new Web3(new
        Web3.providers.HttpProvider("http://localhost:8545"));
}

// Read Multiply7 contract from file
// Note that the contract file is stored in the current directory
fs.readFile("./Multiply7.sol", function (error, result) {
    // Print the read contract
    console.log(result.toString());
    // Print the result of the contract compilation
    console.log("Contract compiled result: " + web3.eth.compile.
solidity(result.toString()));
    // Compile and get abi
    var abiString = web3.eth.compile.solidity(result.toString()).
Multiply7.info.abiDefinition;
    // Get the bytecode after compilation
    var code = web3.eth.compile.solidity(result.toString()).
Multiply7.code;
    // The new method will be executed twice, the first time to get the
transaction hash and the second time to get the contract address
    // Using abi and bytecode you can deploy a contract to Ethereum
    web3.eth.contract(abiString).new({
        data: code,
        from: web3.eth.accounts[0],
        gas: 1000000
    }, function (error, myContract) {
        if (!myContract.address) {
            // Obtain transaction hash
            console.log(myContract.transactionHash);
        } else {
            // Get the contract address
        }
    });
}

```

```
// myContract is the contract instance with which the contract  
methods can be called (myContract  
    console.log(myContract.address);  
}  
});  
});
```

Once the automated deployment script is complete, executing the node AutoDeploy.js command in the terminal will achieve the script file and automatically deploy the Multiply7.sol contract to Ethereum. This deployment method is more straightforward and more efficient compared to the Console.

2.3.3 *Smart Contract Testing and Execution*

The completion of the code writing does not end the development; the code testing is also an essential part. Smart contracts, as a special kind of computer program, need to be well tested. This section will test the smart contract in Truffle using the already integrated Mocha framework, either Ganache or geth, as the testing environment.

Truffle Suite Ganache

Chapter 3 will use the Ethereum client to build a private chain; all contracts can deploy using the private chain and then call the contract method to complete a DApp. But in the development, it is found that calling the contract method is very slow because the Ethereum private chain requires mining confirmation for each transaction executed, and mining is the most time-consuming. Another faster way to create a test network is to use Ganache. As the original TestRPC, Ganache provides a refined, interactive graphical interface and a more user-friendly command line, which is currently recommended. Ganache is developed as part of the Truffle framework suite, based on Node.js, and the entire blockchain's data resides in memory. Ganache can simulate the complete behavior of a geth client, including all RPC APIs. Transactions sent to Ganache are processed immediately without waiting for mining time, making it easier to test development work. Ganache can help you create a bunch of test accounts with funds at startup, making it more suitable for development and testing. TestRPC can be used during the initial stages of contract development and then deployed to a formal environment as the contract improves. Ganache provides both Ganache GUI and Ganache CLI clients. With the Ganache GUI, information such as transaction details can be queried in a graphical format. However, in terms of development efficiency, the command line is relatively better and more convenient. This book details the use of Ganache CLI, the current version of Ganache CLI is “Ganache CLI v6.5.0 (ganache-core: 2.6.0).”

Ganache CLI installation use the command line.

```
$ npm install -g ganache-cli
```

Once the installation is complete, type ganache-cli at the command line for start.

```
$ ganache-cli
```

The terminal screen after a successful Ganache CLI start is shown below.

```
→ ganache-cli
Ganache CLI v6.5.0 (ganache-core: 2.6.0)

Available Accounts
=====
(0) 0x800a4bbaf5f7051d56246a77b7d94759b25e8fe0 (~100 ETH)
(1) 0x288fd53e882a429cd60c3c603a2b03a99417dc79 (~100 ETH)
(2) 0x547bb254f94a8b8e38359c321ad63295ffe60c66 (~100 ETH)
(3) 0x541dbd9ddcce8d858e10e23534d79e5fb7712696 (~100 ETH)
(4) 0xff4067594feace0673bb06d5d4c75bd68358fe2b (~100 ETH)
(5) 0x79e6fc30fd00121e5951a1a0b71b1dbe4cf36da4 (~100 ETH)
(6) 0xdebc722d4170f43f86c1fd9d270f77a2689f7890 (~100 ETH)
(7) 0xdef6aac43557366218e7c1ba96a94025fd38bb88 (~100 ETH)
(8) 0x5853e4c75aa2a756d2801c0de96bc1885147d65c (~100 ETH)
(9) 0xb117611af8de9b44330b46422ba3da9d201f65d5 (~100 ETH)

Private Keys
=====
(0) 0xe52ce14a29a3386fe7b09aa886fba2868c3b1c8f2b90
2d58716180a45e22f593
(1) 0xfd78ef9c79ae5b9f446a2fa8584e5bf8ec2a5ce29c992
a8779755c14f3dafdc5 (2) 0xa42277f6d06da6725326ddb7c0e60b688f06
f100fd80c767a8bbb8a284fcdcc
(3) 0x3098d956910d80fc67dfb293df50617cab7998dc8cff24276360ce
3a5c27b2b (4) 0x9a9d617f2d84abfa2fe66b5b71abe6dda620ecb9e9367846
5ce66079fd7986fa
(5) 0x67a595a6ee6419446464fd7c2b6dc2c81d0a205c1794d9d6d79
67f33c363c68f
(6) 0x60750ef93ed80b06f5471b99bd85c8a9703ba72718bcf67c
8c79f62d25866c1
(7) 0x13c5014d6e44816dc1d0f7938e0a14b5c21dcc1a011f01b6fea
361f1b31d2287
(8) 0xc626ffcecbe0abd989e673b3e908a48be737996864ed62e8
dc433a4fb9c0e8be
(9) 0x92330f58eec3eaffc619618a777d047ceee7445ef4b76fed8df01b0
f07d11df6

HD Wallet
=====
Mnemonic: crystal mind team equal scorpion segment addict slot entire
slim scissors urban Base HD Path: m/44'/60'/0'/0/{account_index}
```

```
Gas Price
=====
20000000000

Gas Limit
=====
6721975
Listening on 127.0.0.1:8545
```

As you can see, Ganache CLI sets up 10 accounts by default and displays the public and private keys of the accounts, and the default port number is 8545, the same as the private chain created by geth. Also, Ganache CLI can specify parameter options at startup, at which point the startup command is as follows.

```
$ ganache-cli <options>
```

Options parameters are as follows.

1. -a or --accounts: specifies the number of accounts that need to be generated at startup.
2. -b or --blocktime: specify blockTime (in seconds) for automatic mining.
3. -d or --deterministic: generates deterministic account numbers based on predefined helper characters.
4. -n or --secure: lock the account by default (more useful for third-party transaction signatures).
5. -m or --mnemonic: use the specified wallet helper character to generate the initialization address.
6. -p or --port: the listening port number is 8545 by default.
7. -h or --hostname: the hostname of the listener, which defaults to Node's server. listen().
8. -s or --seed: use random data to generate wallet helpers.
9. -g or --gasPrice: the price of gas to be used, default is 20,000,000,000.
10. -l or --gasLimit: the gas limit to be used, default 0x6691b7.
11. f or --fork: fork from an Ethereum client with the specified block number already running, the input should be an HTTP address with the port number of another client, e.g., http://localhost:8545, or you can specify the block number to be forked using the @ symbol, e.g., http://localhost:8545@12684.
12. debug: output the VM debug opcode.

The current Ganache implementation of RPC methods is declared as follows.

```
eth_accounts eth_blockNumber eth_call eth_coinbase eth_estimateGas
eth_gasPrice eth_getBalance eth_getBlockByNumber eth_getBlockByHash
eth_getBlockTransactionCountByHash
eth_getBlockTransactionCountByNumber eth_getCompilers
eth_getFilterChanges eth_getFilterLogs
eth_getLogs
// ...
```

Truffle Test Example Analysis

At present, blockchain data is still difficult to visualize all the data like a database. It cannot directly manipulate and modify the data in the blockchain, so it is not easy to test the contract. At present, the primary method is still by calling contract functions and testing based on the input and output results. Standard testing methods include: visualize the invocation method by the online compiler Remix, very convenient and efficient; or using the JSON-RPC and JavaScript API interfaces to execute the contract method through the interactive command line, respectively, and judging whether the invocation is successful based on the results of the output data. All of the above ways achieve the testing and execution of smart contracts.

The Truffle framework already integrates with the automated testing framework Mocha and asserts the use of Chai. It is straightforward to write automated test code for contracts by these two libraries. According to the normal specification development process, the test code should be done simultaneously writing the contract.

After creating a Truffle project, there is a test folder in the directory where the contract test code is stored, and there is already a test file by default after the project is created. The following section describes how to use the Truffle testing framework to implement contract testing. The example used is the MetaCoin project under truffle unbox <box-name>. The test file is named metacoin.js and the test script is implemented as follows.

```
var MetaCoin = artifacts.require("./MetaCoin.sol");
// Each time the contract() function is executed, the contract is
redeployed in the Ethereum client, and the results of the previous test
are not carried over to the next one
contract('MetaCoin', function(accounts) {
  it("Initialize 10,000 MetaCoin to Ethereum's first account", function()
  {
    // First deployment contract
    return MetaCoin.deployed().then(function(instance) {
      // Get the balance of the first account
      return instance.getBalance.call(accounts[0]);
    }).then(function(balance) {
      assert.equal(balance.valueOf(), 10000, "The first account balance
is not equal to 10000");
    });
  });
  it ("Call a function that depends on the link library", function() {
    var meta;
    var metaCoinBalance;
    var metaCoinEthBalance;
    return MetaCoin.deployed().then(function(instance) {
      meta = instance;
      return meta.getBalance.call(accounts[0]);
    }).then(function(outCoinBalance) {
      metaCoinBalance = outCoinBalance.toNumber();
      return meta.getBalanceInEth.call(accounts[0]);
    }).then(function(outCoinBalanceEth) {
      assert.equal(outCoinBalanceEth, 10000);
    });
  });
});
```

```

metaCoinEthBalance = outCoinBalanceEth.toNumber();
}).then(function() {
// Determine if the return value of the library function is
calculated correctly
assert.equal(metaCoinEthBalance, 2 * metaCoinBalance, "Library
function return value does not match, link may be broken";
});
});

It ("sendcoin", function() {
var meta;
var account_one = accounts[0];
var account_two = accounts[1];
var account_one_starting_balance;
var account_two_starting_balance;
var account_one_ending_balance;
var account_two_ending_balance;
var amount = 10;
return MetaCoin.deployed().then(function(instance) {
meta = instance;
// Get the initial balance of the first account
return meta.getBalance.call(account_one);
}).then(function(balance) {
account_one_starting_balance = balance.toNumber();
// Get the initial balance of the second account
return meta.getBalance.call(account_two);
}).then(function(balance) {
account_two_starting_balance = balance.toNumber();
// Send coin
return meta.sendCoin(account_two, amount, {from: account_one});
}).then(function() {
// Get the initial balance of the first account after sending
return meta.getBalance.call(account_one);
}).then(function(balance) {
account_one_ending_balance = balance.toNumber();
// Get the initial balance of the second account after sending(获取发
送后第二个账号初始余额)
return meta.getBalance.call(account_two);
}).then(function(balance) {
account_two_ending_balance = balance.toNumber();
// Compare account limits for matching
assert.equal(account_one_ending_balance,
account_one_starting_balance - amount, "Sending account balance does
not match");
assert.equal(account_two_ending_balance,
account_two_starting_balance + amount, "Receiving account balance
does not match");
});
});
});
```

Executing the test script enables the Ethereum private chain. Ganache CLI is recommended, and the following command can be used under the terminal to perform the automated deployment script.

```
// Execute all test scripts in the test folder
$ truffle test
// Execute a particular test script
$ truffle test metacoin.js
```

There are three `it()` code blocks in the above code, each representing three test cases. The execution process starts by compiling the deployment contract and then executing each `it()` block, while the execution time of each block and the execution time of the whole test process is displayed, and, finally, the test is indicated to pass, as follows.

```
→ MetaCoin truffle test Using network 'test'.

Compiling ./contracts/ConvertLib.sol... Compiling ./contracts/
MetaCoin.sol...
Compiling ./test/TestMetacoin.sol...
Compiling truffle/Assert.sol...
Compiling truffle/DeployedAddresses.sol...

TestMetacoin
✓ testInitialBalanceUsingDeployedContract (102ms)
✓ testInitialBalanceWithNewMetaCoin (104ms)

Contract: MetaCoin
✓ Initialize 10,000 MetaCoin to Ethereum first account
✓ Calling a function that depends on a link library (61ms)
✓ Send coin (135ms)

5 passing (1s)
```

Next, we change the `MetaCoin` value sent in the third test case, setting it to 1,000,000 (over the amount of `MetaCoin` the account has) and run a second test. It is found that the third test case does not pass, and you get the `AssertionError` message prompt, which shows the following.

```
→ MetaCoin truffle test Using network 'test'.

Compiling ./contracts/ConvertLib.sol... Compiling ./contracts/
MetaCoin.sol... Compiling ./test/TestMetacoin.sol...
Compiling truffle/Assert.sol...
Compiling truffle/DeployedAddresses.sol...

TestMetacoin
✓ testInitialBalanceUsingDeployedContract (115ms)
✓ testInitialBalanceWithNewMetaCoin (93ms)

Contract: MetaCoin
✓ Initialize 10,000 MetaCoin to Ether first account
✓ Calling a function that depends on a link library (72ms)
```

```

1) Send coin
  > No events were emitted
4 passing (1s)
1 failing

1) Contract: MetaCoin
  Send coin:
AssertionError: Sender account balance mismatch: expected 10000 to
equal -990000
  at test/metacoin.js:61:14
  at process._tickCallback (internal/process/next_tick.js:68:7)

```

The assertion fails because 1,000,000 MetaCoin is to be sent, and the resulting balance of the first account in the assertion is not equal to the initial balance minus the value of the MetaCoin sent. As seen in the error prompt, the event sent by the contract is received and can be easily viewed for more debugging information. Using events in those essential methods will help in subsequent debugging.

Truffle officially recommends automating tests under the Truffle Suite Ganache client, faster than the Ethereum private chain. Moreover, Truffle can leverage some features in Ganache to boost the speed at runtime. Truffle recommends Ganache during development and testing before running tests to geth or other official Ethereum clients as a more general workflow.

2.3.4 Smart Contract Example Analysis

Here we pick a contract example for detailed analysis. This contract is more complex but shows many features of Solidity. This example is a contract about bank credits. The bank can issue credits to customers who can transfer credits and use credits to exchange goods in the credits mall, and offer essential query functions. For the detailed implementation of the credits contract and the interaction with the interface, you can refer to Sect. 8.1 for a case study of a generic credits system based on Ethereum.

First, deploy the following contracts to the browser compiler or Ethereum private chain, then call the test.

```

contract Score{
    address owner; // The owner of the contract, bank
    uint issuedScoreAmount; // The total number of credits issued by the
bank
    uint settledScoreAmount; // Total number of credits cleared by the
bank
    struct Customer {
        address customerAddr; // Customer address
        bytes32 password; // Customer password
        uint scoreAmount; // Credits balance
        bytes32[] buyGoods; // An array of purchased items
    }
}
```

```

    }
    struct Good {
        bytes32 goodId; // commodities ID
        uint price; // Price
        address belong; // Which merchant does the item belong to address
    }
    mapping (address=>Customer) customer; // Find a customer based on the
client's address
    mapping (bytes32=>Good) good; // Find the item by item ID
    address[] customers; // Array of registered customers
    bytes32[] goods; // Array of items already online
    constructor() public {
        owner = msg.sender;
    }
    function newCustomer(address _customerAddr, string memory
_password) public {
        // Determine if it is already registered
        if (!isCustomerAlreadyRegister(_customerAddr)) {
            // Not yet registered
            customer[_customerAddr].customerAddr = _customerAddr;
            customer[_customerAddr].password = stringToBytes32
(_password);
            customers.push(_customerAddr);
            emit NewCustomer(msg.sender, true, _password);
            return;
        }
        else {
            emit NewCustomer(msg.sender, false, _password);
            return;
        }
    }

    // Determine whether a customer has registered
    function isCustomerAlreadyRegister(address _customerAddr)
internal view returns (bool) {
        for (uint i = 0; i < customers.length; i++) {
            if (customers[i] == _customerAddr) {
                return true;
            }
        }
        return false;
    }
    // Determine if a merchant has registered
    function isMerchantAlreadyRegister(address _merchantAddr) public
view returns (bool) {
        for (uint i = 0; i < merchants.length; i++) {
            if (merchants[i] == _merchantAddr) {
                return true;
            }
        }
        return false;
    }
    // Credits sent by the bank to the customer can only be called by the

```

```
bank and sent to the customer
    event SendScoreToCustomer(address sender, string message) ;

function sendScoreToCustomer(address _receiver,
    uint _amount) onlyOwner public {

    if (isCustomerAlreadyRegister(_receiver)) {
        // Registered
        issuedScoreAmount += _amount;
        customer[_receiver].scoreAmount += _amount;
        emit SendScoreToCustomer(msg.sender, "Issue credits success");
        return;
    }
    else {
        // Not yet registered
        emit SendScoreToCustomer(msg.sender, "The account is not
registered, and the credits failed to be issued");
        return;
    }
    // Two accounts to transfer credits, between any two accounts. Both
customers and merchants call this method
    event TransferScoreToAnother(address sender, string message) ;

function transferScoreToAnother(uint _senderType,
    address _sender,
    address _receiver,
    uint _amount) public {

    if (!isCustomerAlreadyRegister(_receiver)) {
        // The destination account does not exist
        emit TransferScoreToAnother(msg.sender, "The destination
account does not exist, please confirm before transfer!");
        return;
    }
    if (_senderType == 0) {
        // Customers transfer
        if (customer[_sender].scoreAmount >= _amount) {
            customer[_sender].scoreAmount -= _amount;

            if (isCustomerAlreadyRegister(_receiver)) {
                // The destination address is the customer
                customer[_receiver].scoreAmount += _amount;
            } else {
                merchant[_receiver].scoreAmount += _amount;
            }
            emit TransferScoreToAnother(msg.sender, "Credits transfer
successful!");
            return;
        } else {
            emit TransferScoreToAnother(msg.sender, "Your credit balance
is insufficient, transfer failed!");
        }
    }
}
```

```

        return;
    }
}
// The user buys an item with credits
event BuyGood(address sender, bool isSuccess, string message);

function buyGood(address _customerAddr, string memory _goodId)
public {
    // First determine if the input product Id exists
    bytes32 tempId = stringToBytes32(_goodId);
    if (isGoodAlreadyAdd(tempId)) {
        // This item has been added and can be purchased
        if (customer[_customerAddr].scoreAmount < good[tempId].price)
    {
        emit BuyGood(msg.sender, false, "Insufficient balance, failed
to purchase goods");
        return;
    }
    else {
        // Extract the method here
        customer[_customerAddr].scoreAmount -= good[tempId].price;
        merchant[good[tempId].belong].scoreAmount += good[tempId].
price;
        customer[_customerAddr].buyGoods.push(tempId);
        emit BuyGood(msg.sender, true, "buy goods successfully");
        return;
    }
}
else {
    // There is no item with this Id
    emit BuyGood(msg.sender, false, "Input product Id does not exist,
please confirm the purchase");
    return;
}
}
}

```

2.4 History, Problems, and Future Development of Ethereum

As Ethereum grew in popularity and governments and central banks looked into issuing their “digital currencies,” the number of projects hosted on Ethereum increased, with several significant events taking place. This section will analyze these essential events to deepen readers’ understanding of Ethereum.

2.4.1 Historical Events

The DAO Attack

The DAO is the largest crowdfunding project in Ethereum. A decentralized autonomous organization aims to establish a decentralized management structure by writing code for organizational rules and decision-making bodies, eliminating the need for written documentation and reducing the number of managers. The DAO started on April 30, 2016, and the crowdfunding window was open for 28 days. By the end of the project, \$150 million was raised, with over 11,000 members participating, making it the largest crowdfunding project in history.

Stephan Tual, one of the founders of the DAO, announced on June 12 that a problem with a recursive call bug in the software had been discovered, but it would have no impact on DAO's funding, so the issue was quickly squashed as DAO was in beta testing. While programmers were fixing this vulnerability and other issues, an unknown hacker began using this avenue to collect Ethereum coins from the DAO token sale. On June 18, the hacker exploited a vulnerability in a splitDAO function in the DAO smart contract to keep separating DAO assets from the project's asset pool for himself. The hackers managed to mine over 3.6 million Ethereum coins. At that time, the coins' price dropped directly below \$13 from over \$20. Many people tried to break away from the DAO to prevent Ethereum coin theft, but they failed to get the required number of votes in a short period. The DAO held almost 15% of the total Ethereum coins, so this attack had a significant negative impact. There are two main proposals for a solution to the attack on the DAO.

1. Soft fork proposal

The proposal was made in Vitalik Buterin's *CRITICAL UPDATE Re: DAO Vulnerability*. The paper proposes a soft fork, starting from block height 1,760,000, invalid all subsequent transactions, including the DAO and its sub-DAOs; in contrast, other transactions or blocks will be retained without a rollback to prevent attackers from withdrawing stolen Ethereum coins after 27 days. On June 19, after the proposal was published, the attackers announced in an anonymous interview that they would fight the soft fork and reward miners who did not support the soft fork by up to one million Ethereum. On June 22, white hat hackers also launched "Operation Robin Hood" to counter the attackers, aiming to transfer the DAO assets to more secure sub-DAOs.

2. Hard Fork Proposal

This proposal, made by Stephan Tual, called for miners to completely unsteal and return all the DAO's Ethereum coins, so that it can be automatically returned to the token holders thus ending the DAO project. Stephan Tual is against the soft fork, stating that if a soft fork is used for 27 days, the attacker would not be able to claim back the funds he put into the sub-DAO. A hard fork should be used to recover all the Ethereum coins, including the "extra balance" of the DAO and the stolen funds, back to the smart contract. This smart contract will contain a simple

function: withdraw(). This way, everyone may participate in the DAO to withdraw their funds.

Replay Attack

Technically speaking, replay attacks refer to identity fraud. However, replay attacks on Ethereum are not identity fraud, but rather, after a hard fork, transactions on one chain are legitimate on the other chain, and transactions done on one chain can be broadcast on the other chain. It is not an “attack” in nature, but it is still customary to call it replay.

The DAO incident also indirectly led to the replay attack on Ethereum. Ethereum officials carried out at the height of 1.92 million blocks to solve the DAO attack through a hard fork. After Ethereum was hard forked into ETH and ETC, as long as someone withdraws ETH coins from the exchange, it is possible to get the same amount of ETC coins as the data structure of the two chains is the same. Many users keep charging and withdrawing coins (ETH) from the exchange to get extra ETC.

The Birth of EEA

In early 2017, more than 20 top global financial institutions and technology companies, including JP Morgan Chase & Co,⁴ CME Group Inc.,⁵ Bank of New York Mellon,⁶ Microsoft, Intel, and others, formed the Enterprise Ethereum Alliance (EEA). The EEA was established to provide enterprise-class Ethereum blockchain solutions by collaborating among enterprises to develop technologies and standards.

As a nonprofit organization, the EEA Alliance is managed from the bottom-up. Many of the EEA’s proposals and route plans are made by members and voted on among them. The weight of the vote is determined by the members’ contributions to the Alliance in terms of technology or promotion.

Ethereum, one of the most widely used enterprise blockchain deployment and development technologies globally, continues to meet the demands of enterprises. Through the EEA project, Ethereum can improve the enterprise blockchain by standardizing privacy, managing enterprise privileges, and providing new consensus algorithms.

⁴JPMorgan Chase & Co. is an American multinational investment bank and financial services holding company headquartered in New York City. JPMorgan Chase is incorporated in Delaware.

⁵CME Group Inc. is an American global markets company. It is the world’s largest financial derivatives exchange, and trades in asset classes that include agricultural products, currencies, energy, interest rates, metals, stock indexes and cryptocurrencies futures.

⁶The Bank of New York Mellon Corporation, commonly known as BNY Mellon, is an American investment banking services holding company headquartered in New York City. BNY Mellon was formed from the merger of the Bank of New York and the Mellon Financial Corporation in 2007.

Parity Wallet Bug

As mentioned earlier in this chapter, smart contracts are stored on Ethereum's blockchain. It indicates that if a security breach occurs in a smart contract, the transaction information involved will also be affected. In July 2017, a bug in Parity multi-signed electronic wallet version 1.5 was discovered, and an attacker stole more than 150,000 ETH (about \$30 million) from three highly secure multi-signed contracts. In December of that year, a Parity Wallet developer made himself the owner of the library contract, and then called the contract suicide function, invalidating all the contract's functions.

But in fact, these incidents are security flaws in the Parity wallet itself, in the smart contract code, and have nothing to do with Ethereum blockchain security. It is worth noting that this incident is a wake-up call for developers of smart contracts. Streamlining code, trading off some of the Turing-completeness of smart contract code for security, and rigorously conducting a contract code review would significantly reduce the probability of such accidents.

2.4.2 Current Problems of Ethereum

As a more successful open-source project on the blockchain, Ethereum is still in rapid development and exploration. As more and more projects are built on Ethereum, many problems are coming to light, such as inefficient consensus, lack of privacy protection, difficulty in storing data on a large scale, and policing information. These problems will be the difficulties that we have to face after building large DApps.

Inefficient Consensus

At present, Ethereum is in the third stage of Metropolis; the first two stages of Frontier, Homestead, both adopt PoW consensus algorithm; in the third stage, Ethereum will slowly transition to PoS; in the fourth stage, Serenity will use the consensus algorithm of PoS.

PoW is a very effective consensus mechanism, and the Bitcoin network uses the PoW consensus mechanism. The amount of currency obtained by the PoW consensus mechanism depends on the effectiveness of the mining effort, and the better the computer's performance, the more currency is obtained by mining, distributed based on the workload. There are three main problems with this: first, Bitcoin has now attracted most of the world's computing power. It is difficult for blockchain applications using the PoW consensus mechanism to obtain the same amount of computing power to secure themselves, which may be risky; second, mining will cause a lot of wasted resources, especially electricity resources, making the cost of

maintaining this consensus mechanism too high; third, it takes a long period for consensus to be reached. On average, it takes 14 s to package a block, which is too long for commercial projects and cannot meet the requirements of commercial applications.

Meanwhile, PoW is also affected by a low-probability event, the blockchain fork. When two or more nodes on the network compete for bookkeeping power at the same time, two or more blockchain branches will be created in the network. At this point, the data recorded in which branch is valid will have to wait for the next bookkeeping cycle to be decided by the longest blockchain branch, so there is a large delay in transaction data. This situation may lead to data loss on the shorter blockchain.

Lack of Privacy Protection

Although the Ethereum public chain has a certain degree of anonymity, the ledger of transactions on the block is entirely public, and a complete copy of the ledger is available on each node. Since the blockchain needs to trace each account for calculating the balance and verifying the validity of transactions, etc., the transaction data are open and transparent. If you know someone's account, you can understand his transaction record and balance, and there is no privacy to speak of. So if financial applications use Ethereum public chain directly, it is obvious that there is a lack of privacy. Meanwhile, in the alliance chain, multiple enterprises should trust each other and share data, but in case of data leakage, it is difficult to track from which enterprise the leakage is made.

There are already several solutions for privacy protection: coin shuffling, homomorphic encryption, zero-knowledge proof, etc.

1. Coin shuffling. In a large number of transactions involving multiple people, separating the relationship between input and output addresses so that they cannot find a one-to-one correspondence is equivalent to making an obfuscation. The numerous uses of mixed coins can effectively increase privacy protection.
2. Homomorphic encryption. Commonly used in public and federated chains, it is a method that can perform calculations without decrypting encrypted data in advance. It allows public chains to have similar privacy effects as private chains without significant modifications to the blockchain.
3. Zero-knowledge proofs. Zero-knowledge proof is a cryptographic technique that can prove certain data operations without revealing the data. In the transaction records of both parties, establishing that certain data is actual without revealing other additional information reduces the risk of exposing more data.

Large-Scale Data Storage Difficulties

Whether it is the global Bitcoin network or Ethereum, large-scale data storage problems slowly emerge as the volume of transactions increases. Since each node

has a complete ledger, the blockchain system has very much redundant data storage. It sometimes has to trace back each transaction, so there are performance issues as time advances when the transaction data is oversized. If the first time you use it, you need to download all the transactions in the history to work properly, then the execution time of the first transaction will be longer. In order to verify that you have enough money, you need to go back to every transaction in history to calculate the balance, which is an obvious problem when the chain is too long.

Also, since decentralized applications currently rely on smart contracts, when an application contains too much data, the upgrade of smart contracts and data migration of applications can also encounter problems. How to ensure the reliability of large-scale data migration is also the focus of industry research.

Information Difficult to Regulate

Similar to the Bitcoin platform, Ethereum is a public chain where any user can participate in transactions with a string of meaningless numbers as a unique identifier. The transaction information recorded in the Ethereum block is anonymous, and the flow of funds is challenging to supervise, leading to many illegal transactions through “digital currency” for payment. This problem dramatically limits the development of commercial applications of the Ethereum blockchain platform, especially when it comes to the core business of government departments, financial institutions, large enterprises, etc. The current Ethereum platform cannot meet the demand. Only an enterprise-level federated blockchain platform (such as Hyperchain) with permission control can better solve the difficulties of information supervision.

2.4.3 Ethereum 2.0

With the success of the Byzantine fork and the Constantinople fork moving forward, Ethereum 2.0 is slowly surfacing. What is Ethereum 2.0? According to the official plan, Ethereum 2.0 will be an alternative, with many technical teams focusing on early development, and in the first quarter of 2019, the Ethereum 2.0 client has gone live for testing. Since its scope is very broad, this book will focus on the new features and technologies that are updated in Ethereum 2.0.

Beacon Chain

Beacon chain is a new type of blockchain, which is the core structure in Ethereum 2.0. One of the new features is that the original verifier will become the chain builder and can participate in the pledge system, and in this way replace the original miner role. Ethereum 2.0 is planned to use Casper FFG (Casper the friendly Finality gadget), a hybrid PoW/PoS mechanism, for proof of ownership. Users can bet

32 BETH (Beacon ETH, the new type of Ethereum) to register as a verifier in the form of an equity stake. The verifier is assigned one or two shards for validation. The verifier combines the information on his shard with the data on the beacon chain to produce new blocks, and the verifier has the right to approve or reject other verifiers' blocks.

Another feature is that the beacon chain stores an index of the shard state. Each time the state on a shard changes, a new hash value is established because of the changed state, and this hash value is then validated at checkpoints on the beacon chain. Because of this mechanism, shards can communicate with each other by tracking the state of each other through beacon chains.

The beacon chain does not store information traditionally stored in blockchains such as account information, DApp status information, and contract status; Instead, it stores the list of storage verifiers and sharding proof information. Therefore, like a GPS, beacon chain facilitates users to locate vast information and build a huge blockchain network together.

Sharding

Sharding is considered the best solution to Ethereum scalability. Sharding is when data on a blockchain is divided into many different pieces and stored on different nodes. By decentralizing the whole Ethereum into a distributed parallel system, the purpose of scaling is achieved at the cost of reducing the amount of data stored in each node while ensuring the integrity of Ethereum data.

The Beacon chain randomly selects the fragment verifier for each segment in each cycle to verify the content and status of the slice by crosslinking. However, if the verifier wants to update, all members of the verifier committee need to reach a consensus to ensure that the information is not easily tampered with.

eWASM

EVM are now less efficient than other mainstream virtual machines. This was mainly due to its early design goal of "flexibility over performance," and the overall overhead was very obvious in some scenarios. There are two main reasons: EVM involves a lot of state-changing operations, and the use of advanced mathematical and cryptographic algorithms within the EVM. eWASM can be considered the second generation of EVM and will continue to support smart contracts, accounts, and other abstractions. For the foreseeable future, familiar development tools such as Truffle, Ganache, and the Solc compiler will also be ported to eWASM. Compared to EVM, eWASM is faster, more scalable, supports more high-level languages, has a broader community, and is supported by more system platforms.

Casper FFG

It is well known that PoW works with a massive amount of power and computing power. The data shows that if you add the current annual electricity consumption of Bitcoin and Ethereum for mining, it ranks the 34th in the country's electricity consumption ranking, more than the annual electricity consumption of Pakistan. Moreover, the PoW mechanism is unsuitable for the subsequent sharding of Ethereum, which can cause serious security issues. Therefore, the Casper FFG consensus mechanism was born.

Here is a brief description of how Casper FFG works. Verification nodes are required to pay a certain amount of Ethereum in the blockchain network as a "margin" for access to verification. When a new block needs to be added to the blockchain, the verification phase will place a "bet" to exercise the verification rights. If the new block is successfully added to the chain, then the verifying block will receive a reward proportional to the size of the "bet." However, if the validation node does not comply with the rules, it will be sanctioned, losing all bets and part of the margin. Strict rewards and punishments prevent all participants from committing evil acts, and those who obey the rules will be richly rewarded. We can believe that under this mechanism, the behavior that benefits the network will maximize the interests of the nodes, reducing the network overhead and guarantee the network's security at the same time. In this way, it helps Ethereum to transition to PoS smoothly.

2.5 Summary

This chapter introduces the basics of the development history, basic concepts, clients, account management, and Ethereum network, explores the core principles of key Ethereum modules such as Ethereum consensus mechanism, virtual machine, data storage, and cryptographic algorithm, introduces in detail the writing, deployment, testing, and execution of Ethereum smart contracts, and finally analyzes and discusses the significant events in the development of Ethereum, the main problems that exist now, and the future development.

Chapter 3

Fundamentals of Ethereum: Application Development



This chapter explains how to develop an Ethereum DApp (decentralized application) from scratch. First, it describes how to configure the Ethereum environment and set up the Ethereum private chain, which can be used as the running environment of the DApp. Second, the integrated development environment Mix and the real-time browser Compiler are introduced. Most of the development of DApps (such as the writing of smart contracts) can be done in these IDE. Thirdly, two essential interfaces for Ethereum smart contract to interact with the front end are presented: JSON-RPC and JavaScript API, through which the smart contract in Ethereum can be called. We then introduce the mature frameworks for rapid DApp development (Meteor and Truffle) and give a layered and extensible project development process. Finally, an Ethereum application MetaCoin is implemented to facilitate readers to learn to use the framework to develop DApp, complete the writing, deployment, invocation of smart contracts, and test interaction with the front page.

3.1 Building an Ethereum Development Environment

This section describes the Ethereum environment configuration and private chain construction under Mac; the configuration of different operating systems is similar. It is recommended to set up the Ethereum private chain under Mac or Ubuntu.

3.1.1 Configuring the Ethereum Environment

Install Go Environment

Since Ethereum is developed in Go language, to install it on the local machine, you need to install Go environment first. You can download the corresponding Go

language installation package from the official website. For Mac, download go1.12.7.darwin-arm64.pkg and double click to install it. The default installation is in the /usr/local/go directory, and the environment variables are set automatically.

Also, you need to configure a GOPATH environment variable as a working directory for Go. Go to the terminal and edit the .bash_profile file.

```
vi ~/.bash_profile
```

Add the following environment variables:

```
#go
export GOROOT=/usr/local/go
export GOBIN=$GOROOT/bin
export PATH=$PATH:$GOBIN
```

For the configuration file to take effect immediately, execute the following command in the terminal.

```
source ~/.bash_profile
```

Execute the following command in the terminal to see if the installation was successful.

```
go version
```

If the following command line appears, the Go programming environment is successfully installed.

```
→ go version
go version go1.12.7 darwin/amd64
```

Installing Node.js and npm

npm is a package management tool under Node.js that makes it very easy to install JavaScript-based software and packages. Many development tools based on Ethereum are developed on JavaScript, which can be installed using npm. From the official Node.js website, you can download different versions of Node.js according to the operating system prompt, and install them after downloading. By default, both Node.js and npm will be installed, and you can check whether the installation is successful by executing the following commands in the terminal, respectively.

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

The installation is successful if the following command line appears.

```
→ npm -v  
6.9.0  
→ node -v  
10.16.0
```

Installing Brew

Brew, known as Homebrew, is a package manager that makes it easy to install and update applications and utilities. It has functions such as install, uninstall, and update, and it can manage packages with simple commands. Get the command line from Brew's official website and download and install it on the terminal.

```
→ /usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/  
Homebrew/install/master  
/install)"
```

Installing Ethereum

Go to the terminal and execute the following command to install it.

```
brew update  
brew upgrade  
brew tap ethereum/ethereum  
brew install ethereum
```

Then, execute the following command to see if the installation is successful.

```
geth version
```

The installation is successful if the following command line appears:

```
→ geth version  
Geth  
Version: 1.8.27-stable  
Architecture: amd64  
Protocol Versions: [63 62]  
Network Id: 1  
Go Version: go1.12.4  
Operating System: darwin  
GOPATH=  
GOROOT=/usr/local/go
```

Installing the solc Compiler

solc is the compiler for the smart contract Solidity and can be installed by executing the following command in the terminal.

```
npm install -g solc solc-cli --save-dev
To install solidity:
brew install solidity
link and overwrite.
brew link --overwrite solidity
Execute the following command to see if solc is installed successfully:
solc --version
If the following command line appears, solc is successfully installed:
→ solc --version
solc, the solidity compiler commandline interface
Version: 0.5.10+commit.5a6ea5b1.Darwin.appleclang
```

3.1.2 Building an Ethereum Private Chain

Create Accounts (Public Key)

You can create the Ethereum accounts by entering the following command three times in the terminal, and you need to enter the password of the account when making it (in the demo of this chapter, the password is 123456).

```
geth account new
```

The command line for successfully creating an account looks like the following.

```
→ geth account new
Your new account is locked with a password. Please give a password. Do
not forget this password.
Passphrase:
Repeat passphrase:
Address: {08aac788e0e6146586f61f57419b6e3b0868de22}
```

All accounts can be viewed in the `~/Library/Ethereum` directory under `keystore`.

Write the Original Block File

Create the test-genesis .Json file in the root directory (`~/.ethereum`). You can assign enough Ethereum to the account you just applied for by setting the balance of the account address in `alloc`.

Initialize the Original Block

Ethereum is installed in the “`~/Library/Ethereum`” directory by default. Use the following command to initialize the original block file you just set up.

```
qeth --datadir "~/Library/Ethereum" init ~/test-genesis.json
```

The command line result of a successful initialization is shown below:

```
→ geth --datadir "~/.Library/Ethereum" init ~/test-genesis.json
INFO [07-10|15:56:44.419] Maximum peer count          ETH=25 LES=0
total=25
INFO [07-10|15:56:44.432] Allocated cache and file handles
database=/Users/username/Library/Ethereum/geth/chaindata cache=16
handles=16
INFO [07-10|15:56:44.456] Writing custom genesis block
INFO [07-10|15:56:44.457] Persisted trie from memory database
nodes=4 size=600.00B time=150.02µs gcnodes=0 gcsize=0.00B gctime=0s
livenodes=1 livesize=0.00B
INFO [07-10|15:56:44.457] Successfully wrote genesis state
```

```

database=chaindata           hash=d03d49...e3ec3f
  INFO [07-10|15:56:44.457] Allocated cache and file handles
database=/Users/username/Library/Ethereum/geth/lightchaindata
cache=16 handles=16
  INFO [07-10|15:56:44.480] Writing custom genesis block
  INFO [07-10|15:56:44.480] Persisted trie from memory database
nodes=4 size=600.00B time=78.796µs gcnodes=0 gcsize=0.00B gctime=0s
livenodes=1 livesize=0.00B
  INFO [07-10|15:56:44.480] Successfully wrote genesis state
database=lightchaindata           hash=d03d49...e3ec3f

```

Configure the Script to Unlock the Accounts Automatically

Go to Ethereum installation directory “~/Library/Ethereum,” create a password file, and enter the password corresponding to each account in test-genesis.json in this file, one line for each password. Enter the password. As shown below:

```

123456
123456
123456

```

Write the Ethereum Startup Script

Create the startup script file private_blockchain.sh and configure the following in the file:

```

geth --rpc --rpcaddr "127.0.0.1" --rpcport "8545" --rpccorsdomain "*"
--unlock 0,1,2 --password ~/Library/Ethereum/password --nodiscover --
maxpeers 5 --datadir '~/Library/Ethereum' console

```

Whenever you start a geth node in the future, execute the following command in the terminal.

```
sh private_blockchain.sh
```

The result of successfully starting the Ethereum private chain is shown in the following command line.

```

→ sh private_blockchain.sh
INFO [07-10|16:42:14.066] Maximum peer count          ETH=0 LES=0
total=0
  INFO [07-10|16:42:14.078] Starting peer-to-peer node
instance=Geth/v1.8.27-stable/darwin-amd64/go1.12.4
  INFO [07-10|16:42:14.078] Allocated cache and file handles
database=/Users/username/Library/Ethereum/geth/chaindata cache=512
handles=5120

```

```

INFO [07-10|16:42:14.118] Initialised chain configuration
config="{ChainID: 1024 Homestead: 0 DAO: <nil> DAOSupport: false EIP150:
<nil> EIP155: 0 EIP158: 0 Byzantium: <nil> Constantinople: <nil>
ConstantinopleFix: <nil> Engine: unknown}"
INFO [07-10|16:42:14.118] Disk storage enabled for ethash caches
dir=/Users/username/Library/Ethereum/geth/ethash count=3
INFO [07-10|16:42:14.118] Disk storage enabled for ethash DAGs  dir=/
Users/username/.ethash
count=2
INFO [07-10|16:42:14.118] Initialising Ethereum protocol
versions="[63 62]" network=1
INFO [07-10|16:42:14.191] Loaded most recent local header      number=0
hash=d03d49...e3ec3f td=1024 age=50y2mo3w
INFO [07-10|16:42:14.191] Loaded most recent local full block
number=0 hash=d03d49...e3ec3f td=1024 age=50y2mo3w
INFO [07-10|16:42:14.191] Loaded most recent local fast block
number=0 hash=d03d49...e3ec3f td=1024 age=50y2mo3w
INFO [07-10|16:42:14.191] Loaded local transaction journal
transactions=0 dropped=0
INFO [07-10|16:42:14.191] Regenerated local transaction journal
transactions=0 accounts=0
INFO [07-10|16:42:14.229] New local node record                  seq=2
id=66ef7caeee07af93
ip=127.0.0.1 udp=30303 tcp=30303
INFO [07-10|16:42:14.229] Started P2P networking           self=enode://
bc9504a48e84e1e7c366
310f0c09a6a3331cacc5e558304d9ee9edc87d90df9f0329d95eb72d5340
5ec0c7ba7595cbd9a4287f7d158bb4c687d5dfcc9b1739ea@127.0.0.1:30303
INFO [07-10|16:42:14.231] IPC endpoint opened            url=/Users/
username/Library/Ethereum/geth.ipc
Welcome to the Geth JavaScript console!

instance: Geth/v1.8.27-stable/darwin-amd64/go1.12.4
INFO [07-10|16:42:14.295] Etherbase automatically configured
address=0x08aac788E0E6146586f61f57419b6E3B0868dE22
coinbase: 0x08aac788e0e6146586f61f57419b6e3b0868de22
at block: 0 (Thu, 01 Jan 1970 08:00:00 CST)
datadir: /Users/username/Library/Ethereum
modules: admin:1.0 debug:1.0 eth:1.0 ethash:1.0 miner:1.0 net:1.0
personal:1.0 rpc:1.0 txpool:1.0 web3:1.0

```

Initiating Mining

Every transaction executed on Ethereum requires mining by miners before it can be confirmed. The same mining operation can be performed on the private chain. After starting the private chain, the following command is executed to start mining.

```
miner.start()
```

Stop mining by executing the following command:

```
miner.stop()
```

Mining is shown in the following command line:

```
> miner.start()
INFO [07-10|16:43:47.456] Generating DAG in progress
epoch=0 percentage=99 elapsed=40.480s
INFO [07-10|16:43:47.458] Generated ethash verification
cache epoch=0 elapsed=40.482s
INFO [07-10|16:43:48.653] Successfully sealed new block
number=1 sealhash=81f936...f0ec4f hash=16b9b6...cf5e9d
elapsed=42.187s
INFO [07-10|16:43:48.654] ↳ mined potential block
number=1 hash=16b9b6...cf5e9d
INFO [07-10|16:43:48.654] Commit new mining work number=2
sealhash=bc00e4...b9f784 uncles=0 txs=0 gas=0 fees=0
elapsed=124.575µs
INFO [07-10|16:43:49.097] Successfully sealed new block
number=2 sealhash=bc00e4...b9f784 hash=fbbe83...321b7f
elapsed=443.516ms
INFO [07-10|16:43:49.097] ↳ mined potential block
number=2 hash=fbbe83...321b7f
INFO [07-10|16:43:49.118] Commit new mining work number=3
sealhash=357d41...c18556 uncles=0 txs=0 gas=0 fees=0
elapsed=19.682ms
INFO [07-10|16:43:49.523] Successfully sealed new block
number=3 sealhash=357d41...c18556 hash=10d5f6...3384f6
elapsed=425.227ms
INFO [07-10|16:43:49.545] ↳ mined potential block
number=3 hash=10d5f6...3384f6
INFO [07-10|16:43:49.565] Commit new mining work number=4
sealhash=f68f0c...7ebba6 uncles=0 txs=0 gas=0 fees=0
elapsed=20.053ms
INFO [07-10|16:43:49.609] Generating DAG in progress
epoch=1 percentage=0 elapsed=608.348ms
INFO [07-10|16:43:50.432] Generating DAG in progress
epoch=1 percentage=1 elapsed=1.431s
INFO [07-10|16:43:51.374] Generating DAG in progress
epoch=1 percentage=2 elapsed=2.373s
INFO [07-10|16:43:51.588] Successfully sealed new block
number=4 sealhash=f68f0c...7ebba6 hash=150a53...d1640f
elapsed=2.043s
INFO [07-10|16:43:51.588] ↳ mined potential block
number=4 hash=150a53...d1640f
```

```
INFO [07-10|16:43:51.589] Commit new mining work number=5
sealhash=4c8f1a...ebeaca uncles=0 txs=0 gas=0 fees=0
elapsed=179.548µs
> INFO [07-10|16:43:52.281] Generating DAG in progress
epoch=1 percentage=3 elapsed=3.279s
INFO [07-10|16:43:52.285] Successfully sealed new block
number=5 sealhash=4c8f1a...ebeaca hash=2dbbc4...a46d1a
elapsed=696.180ms
INFO [07-10|16:43:52.295] ✅ mined potential block
number=5 hash=2dbbc4...a46d1a
INFO [07-10|16:43:52.316] Commit new mining work number=6
sealhash=3c2b0a...a6cb17 uncles=0 txs=0 gas=0 fees=0
elapsed=183.722µs
> INFO [07-10|16:43:52.690] Successfully sealed new block
number=6 sealhash=3c2b0a...a6cb17 hash=62907a...9c2f1b
elapsed=374.531ms
INFO [07-10|16:43:52.690] ✅ mined potential block
number=6 hash=62907a...9c2f1b
INFO [07-10|16:43:52.698] Commit new mining work number=7
sealhash=01514a...fb9834 uncles=0 txs=0 gas=0 fees=0
elapsed=7.452ms
INFO [07-10|16:43:53.212] Generating DAG in progress
epoch=1 percentage=4 elapsed=4.211s
INFO [07-10|16:43:53.322] Successfully sealed new block
number=7 sealhash=01514a...fb9834 hash=808155...d2f019
elapsed=631.685ms
INFO [07-10|16:43:53.327] ✅ mined potential block
number=7 hash=808155...d2f019
INFO [07-10|16:43:53.349] Commit new mining work number=8
sealhash=528875...a41253 uncles=0 txs=0 gas=0 fees=0
elapsed=183.062µs
INFO [07-10|16:43:53.559] Successfully sealed new block
number=8 sealhash=528875...a41253 hash=0d2813...0d27d9
elapsed=210.476ms
INFO [07-10|16:43:53.560] ✅ block reached canonical chain
number=1 hash=16b9b6...cf5e9d
INFO [07-10|16:43:53.582] ✅ mined potential block
number=8 hash=0d2813...0d27d9
```

3.2 Ethereum Remix IDE

While common software projects rely on integrated development environments (IDE), Ethereum decentralized application development has a dedicated development environment: the Remix browser real-time compiler. Currently, Remix is the most widely used compiler in actual project development because it compiles quickly, with friendly error prompts, and easy to test.

The Remix online compiler makes it very easy to test the states and methods in smart contracts and get the bytecode files and ABI (Application Binary Interface) files for the contracts. If you want to use Remix offline, visit GitHub and download the ZIP file to decompress it.

3.2.1 Compiling Smart Contracts

Let us use the Sample contract for compilation tests in an online compiler.

```
pragma solidity >=0.4.22 <0.6.0;
contract Sample
{
    uint value;
    function setSample(uint v) public{
        value = v;
    }
    function set(uint v) public{
        value = v;
    }
    function get() public view returns (uint) {
        return value;
    }
}
```

The main interface is shown in Fig. 3.1. The left side of the online compiler is the code editing area, where the smart contract can be written directly; on the right side is the parameter input and output area and debugging site.

The online compiler can prompt in real time at the head of each line if there is a syntax error in the contract, as shown in Fig. 3.2. The error is that the entire expression lacks the right value.

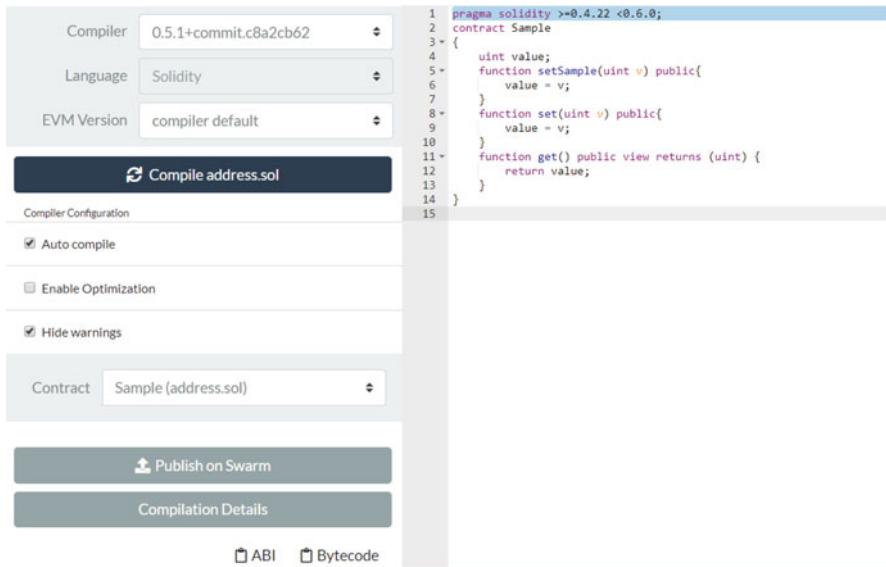


Fig. 3.1 Online real-time compiler compilation contract

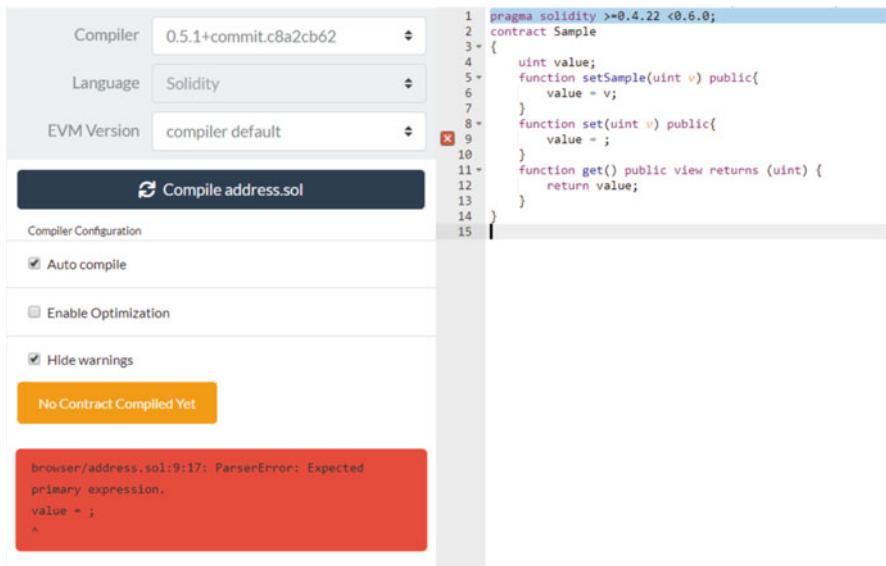


Fig. 3.2 Online compiler prompts for errors

3.2.2 Obtaining Bytecode and ABI Files

After the online compiler automatically compiles the smart contract, some necessary result data is generated on the right side. Bytecode is the bytecode file of the contract, that is, the file that EVM can run. The bytecode of the Sample contract is as follows:

```
245:0;;;;;"  
}
```

The string in Interface is the ABI of the contract, and the Interface is provided for external access. Every method in the contract (except the internal method) is described in the ABI: the constant field indicates whether the method is a continuous method, which does not change the state of the contract; inputs indicate the input parameters and types of the method. Name is the method name; Outputs represent the method's output parameters and types, that is, return values; Type is the type of the interface. Function stands for ordinary methods; constructor is a method of construction. The ABI for Sample is as follows:

```
[  
 {  
   "constant": false,  
   "inputs": [  
     {  
       "name": "v",  
       "type": "uint256"  
     }  
   ],  
   "name": "set",  
   "outputs": [],  
   "payable": false,  
   "stateMutability": "nonpayable",  
   "type": "function"  
 },  
 {  
   "constant": true,  
   "inputs": [],  
   "name": "get",  
   "outputs": [  
     {  
       "name": "",  
       "type": "uint256"  
     }  
   ],  
   "payable": false,  
   "stateMutability": "view",  
   "type": "function"  
 },  
 {  
   "constant": false,  
   "inputs": [  
     {  
       "name": "v",  
       "type": "uint256"  
     }  
   ],  
   "name": "setSample",  
   "outputs": [] ,
```

```

    "payable": false,
    "stateMutability": "nonpayable",
    "type": "function"
}
]
```

In web3 deploy is JavaScript code that uses the web3.js interface to create a contract instance. The incoming parameters include the bytecode and ABI file generated above. Return parameters include the contract instance, contract address, transaction hash, etc. The contract instance can be used to call the methods in the contract to achieve interaction with the contract.

```

Var sampleContract =
web3.eth.contract([{"constant":false,"inputs":
[{"name":"v","type":"uint256"}],"name":"set","outputs":
[],"payable":
false,"stateMutability": "nonpayable", "type": "function"}, {
"constant":true,"inputs":[],"name": "get", "outputs":
[{"name": "", "type": "uint256"}], "payable":
false,"stateMutability": "view", "type": "function"}, {"constant":
false,"inputs":
[{"name": "v", "type": "uint256"}], "name": "setSample", "outputs":
[], "payable":
false,"stateMutability": "nonpayable", "type": "function"}]);
var sample = sampleContract.new(
{
    from: web3.eth.accounts[0],
    data: '0x608060405234801561001057600080fd5
b50610140806100206000396000f3fe60806040526
0043610610051576000357
c01000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000009
004806360fe47b1146100565780636d4ce63c14610091
578063e221818b146100bc575b600080fd5b3480
1561006257600080fd5
b5061008f600480360360
2081101561007957600080fd5b81019
080803590602001909291905050506100f7565b00
5b34801561009d57600080fd5b506100a6610101565b6040518082815260
200191505060405180910390f35b3480156100c857600080fd5b506100f56
00480360360208110156100df57600080fd5b810190808035906020019092
919050505061010a565b005b8060008190555050565b600080549050905
65b806000819055505
056fea165627a7a723058208be72898628919db93a8bf00e200e6e311102
a923118a475612493c7da2401ca0029',
    gas: '4700000'
}, function (e, contract){
    console.log(e, contract);
    if (typeof contract.address !== 'undefined') {
        console.log('Contract mined! address: ' + contract.address + '
transactionHash: ' +
            contract.transactionHash);
    }
});
```

```

        }
    })
}
```

In uDApp is the JSON value returned after compilation, the JSON contains the contract name, bytecode, and ABI. If the Sample contract is compiled using the web3.js interface, the JSON value is what is returned.

```

{
  "timestamp": 1562724220093,
  "record": {
    "value": "0",
    "parameters": [],
    "abi": "0x99ff0d9125e1fc9531a11262e15aeb2c60509a078c4
cc4c64cefdfb06ff68647",
    "contractName": "Sample",
    "bytecode": "608060405234801561001057600080fd5b50610140806
100206000396000f3fe60806040
5260043610610051576000357c0100000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000
6360fe47b1146100565780636d4ce63c14610091578063e221818b146100
bc575b600080fd5b
34801561006257600080fd5b5061008f60048036036
02081101561007957600080fd5b81019080803590620019092919050505
06100f7565b005b34801561009d57600080fd5b506100a6610101565b604
0518082815260200191505060405180910390f35b3480156100c85760008
0fd5b506100f5600480360360208110156100df57600080fd5b81019080803
5906020019092919050505061010a565b005b8060008190555050565b60
008054905090565b806000819055505056fea165627a7a723058208be728
98628919db93a8bf00e200e6e31102a923118a475612493c7da2401ca0029",
    "linkReferences": {},
    "name": "",
    "inputs": "()",
    "type": "constructor",
    "from": "account{0}"
  },
}
```

3.2.3 Testing Contract Methods

As shown in Fig. 3.3, when calling the set() method, fill in the input box with the value, then click the “Set” button to set the value, and click the “get” method button to return the value. The online compiler makes it very easy to test the methods in a smart contract visually. The data is stored in memory, and there is no need to open the Ethereum private chain.

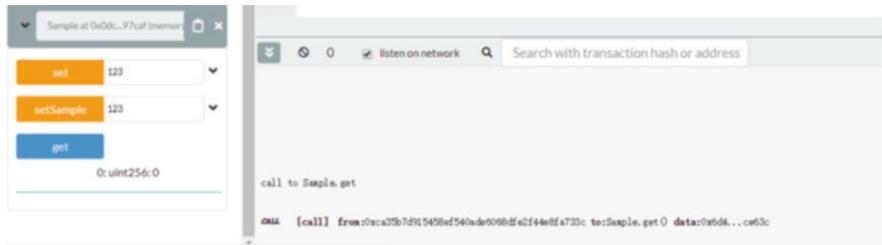


Fig. 3.3 Contract methods test

3.3 Ethereum Programming Interface

As an independent underlying platform, the Ethereum blockchain platform needs to interact with the outside, providing interfaces for the outside. At present, the RPC interface is natively supported by Ethereum, unlimited language, and cross-platform. JavaScript API (web3.js) encapsulates JavaScript for RPC, simpler and more convenient to use but limited to JavaScript language calls.

3.3.1 jSON-RPC

jSON is a lightweight data-interchange format that can represent numbers, strings, ordered sequences of values, and collections of name/value pairs. jSON-RPC is a stateless, lightweight remote procedure call protocol. It uses jSON (RFC 4627) as data format. jSON-RPC mainly defines several data structures and rules around their processing between different messaging environments (Sockets, HTTP).

The jSON-RPC interface is used in this case to deploy the call contract in the Ethereum private chain.

Smart Contracts

Let us take a straightforward contract, Multiply7, as an example. The contract has a multiply() method, which passes in a uint type data and returns the result after multiplying by seven.

```
pragma solidity >=0.4.22 <0.6.0;
contract Multiply7 {
    event Print(uint);
    function multiply(uint input) public returns (uint) {
        emit Print(input * 7);
        return input * 7;
    }
}
```

Compiling the Contract

First, you need to open the Ethereum private chain in one terminal (terminal A) (refer to Sect. 3.1.2), then open another terminal (terminal B) and enter commands to call the JSON-RPC interface in Ethereum.

Since go-ethereum dropped the `eth_compileSolidity` method in version 1.6, an attempt to call it with curl returns “The method `eth_compileSolidity` does not exist/is not available.” Therefore, the contract is compiled online using Remix as mentioned in Sect. 3.2 to obtain the most crucial bytecode and `abiDefinition` data.

Bytecode

"608060405234801561001057600080fd5b
5060fc8061001f6000396000f3fe6080604052600436106039576000357c01
000000000000000000000000000000000000000000000000000000000000000000000900
48063c6888fa114603e575b600080fd5b348015604957600080fd5b506073
60048036036020811015605e57600080fd5b810190808035906020019092
91905050506089565b6040518082815260200191505060405180910390f3
5b60007f24abdb5865df5079dcc5ac590ff6f01d5c16edb5fabb4e195d9febdb
1114503da600783026040518082815260200191505060405180910390a1
60078202905091905056fea165627a7a723058205cfb8ba182658158a76b8
3c6ae1695a36cd2853022dfc1d4ad9d63ba5e27cf370029"

abiDefinition

```

    {
      "indexed": false,
      "name": "",
      "type": "uint256"
    }
  ],
  "name": "Print",
  "type": "event"
}
]

```

Obtaining the Calling Account

First, you need to open the Ethereum private chain in one terminal (terminal A) (refer to Sect. 3.1.2); then open another terminal (terminal B) and enter commands to call the JSON-RPC interface in Ethereum.

There may be multiple accounts on the current Ethereum private chain. One account needs to be selected as the call initiator to deploy the contract and the call to the contract method. The account should contain enough Ethers. The mining base address here is used as the call initiator of the transaction, and that address is the first account in the current default account. Execute the following command in terminal B.

```
curl -H "Content-Type: application/json" -X POST --data
'{"jsonrpc": "2.0", "method": "eth_coinbase", "id": 1}' localhost:8545
```

The result is as follows, where “0x29f9ceac9ad80fd672ddb3f9ea4bedd89c3360f5” is the account to be used to initiate the transaction.

```
{"jsonrpc": "2.0", "id": 1, "result": "0x29f9ceac9ad80fd672ddb3f9ea4bedd89c3360f5"}
```

View the Ethers in the Current Account

Originators in Ethereum need to have enough Ethers for the transaction to confirm the transaction and then add it to the blockchain. Mining in the private chain sends Ethers to Coinbase, the first default account on the current Ethereum account. Coinbase can undoubtedly make changes as needed. Execute the following command on terminal B to see if there are enough Ethers.

```
curl -H "Content-Type: application/json" -X POST --data '{"jsonrpc": "2.0", "method": "eth_getBalance", "params": ["0x29f9ceac9ad80fd672ddb3f9ea4bedd89c3360f5", "latest"], "id": 2}' localhost: 8545
```

The result is the number of Ethers in the current account.

```
{"jsonrpc": "2.0", "id": 2, "result": "0x6765cb1e5eacbd41b340000"}
```

Deploying the Contract

Execute the following command in terminal B. The from in the params parameter is the account that initiated the transaction, and the data parameter is the bytecode of the contract.

```
curl -H "Content-Type: application/json" -X POST --data
'{"jsonrpc": "2.0", "method": "eth_sendTransaction",
"params": [{"from": "0x29f9ceac9ad80fd672ddb3f9ea4bedd89c3360f5",
"gas": "0x1d5b7",
"gasPrice": "0x4a817c800",
"data": "0x608060405234801561001057600080
fd5b5060fc8061001f6000396000f3fe6080604052600436
106039576000357c01000000000000000000000000000000
000000000000000000000000000000000000000000000000
e575b600080fd5b34801560495760080fd5b50607360048036036020811
015605e57600080fd5b81019080803590602001909291905050506089565
b6040518082815260200191505060405180910390f35b60007f24abdb586
5df5079dcc5ac590ff6f01d5c16edbc5fab4e195d9febcd114503da6007830
26040518082815260200191505060405180910390a160078202905091905
056fea165627a7a723058205cfb8ba182658158a76b83c6ae1695a36cd28
53022dfc1d4ad9d63ba5e27cf370029"
}], "id": 6}' localhost:8545
```

After executing the above command in terminal B, the transaction needs to be mined to be confirmed, so the following command is executed in terminal A to start mining.

```
miner.start()
```

Data returned from terminal A.:

```
Submitted contract creation
fullhash=0x21cbe81107f6ab833730416f144cb26ea5da61cc93
df0952f49f7d39bd6a3109
contract=0xc3A0E837c060445D8F27CE1541f3e58335B6fff43
```

Terminal B returns the result, which is the transaction hash of this transaction (deployment contract).

```
{"jsonrpc": "2.0", "id": 6, "result": "0x21cbe81107f6ab833730416f144cb26ea5da61cc93df0952f49f7d39bd6a3109"}}
```

Calling the Contract Method

The method to call the contract is as follows: the to parameter is the address of the contract, and data is the parameter to be passed to call the contract. Data parameter includes the method to call the contract and the specific parameter value, abbreviated as payload. It is confirmed that the transaction also needs to be mined in terminal A.

Calculate the bytes corresponding to the method selector in the payload in terminal A. Select the first four bytes of the Keccak hash table, and encode them in hexadecimal.

```
> web3.sha3 ("multiply(uint256)").substring(0, 10)  
"0xc6888fa1"
```

Suppose the value to be passed in is 6, a uint256 type, which will be encoded as

Combining method selectors and encoding parameters produces the data in the data above.

Terminal B returns the following result. Details of the transaction can be queried by returning the transaction hash in the result.

```
{"jsonrpc": "2.0", "id": 8, "result": "0xd39545b3e2217608f019816c09e04ff348f5e67c6d4dfaecc12831cc8310a0f3"}}
```

Find Transaction Details by Transaction Hash

After a transaction occurs, the actual return result is only the transaction hash. You can use this transaction hash to get the exact details of the transaction and the return values of some methods. Run the following command on terminal B to obtain the result and return value of the transaction.

```
curl -H "Content-Type : application/json" -X POST --data '{ "jsonrpc" : "2.0" , "method": "eth_getTransactionReceipt", "params": ["0xd39545b3e2217608f019816c09e04ff348f5e67c6d4dfaecc12831cc8310a0f3"] , "id":7}' localhost:8545B
```

The terminal returns the following result, where the logs.data field is the actual return value of the method. Converting hexadecimal 2a to decimal makes 42, as expected, and the contract method call passes.

```
        "transactionIndex": "0x0"  
    }  
}
```

3.3.2 *JavaScript API*

The process of calling the contract using JSON-RPC described above is cumbersome, involving complex encoding and decoding and base conversion. At present, Ethereum officially provides `web3.js` realized by JavaScript, which encapsulates RPC mode and provides a concise interface externally. You can view all the calling methods supported by `web3` by typing the `web3` command in the terminal with the Ethereum private chain turned on. The same `Multiply7` contract from Sect. 3.3.1 demonstrates how to compile and deploy contracts and methods for calling them using the `web3.js` interface.

Compiling the Contract

Since go-ethereum dropped the `eth_compileSolidity` method in version 1.6.0, you cannot use the `web3.eth.compile.solidity` method in the `web3.js` module; therefore, the method in Sect. 3.3.1 will still be used to compile the contract.

Deploying the Contract

First, assign the bytecode and ABI to two variables separately for subsequent use.

```
> var code = '0x608060405234801561001057
600080fd5b5060fc8061001f6000396000f3fe608060405260043
6106039576000357c0100000000
0000000000000000000000000000000000000000000000000090048063c6888
fa114603e575b600080fd5b348015604957600080fd5b5060736004803603
6020811015605e57600080fd5b8101908080359060200190929190505050
6089565b6040518082815260200191505060405180910390f35b60007f
24abdb5865df5079dcc5ac590ff6f01d5c16edbc5fab4e195d9febcd1114503
da600783026040518082815260200191505060405180910390a160078202
905091905056fea165627a7a723058205cfb8ba182658158a76b83c6ae16
95a36cd2853022dfc1d4ad9d63ba5e27cf370029';
undefined
```

```
> var abi = [{"constant":false,"inputs": [{"name":"input","type":"uint256"}],"name":"multiply","outputs":[{"name":"","type":"uint256"}]},{ "payable": false,"stateMutability": "nonpayable","type": "function"}, {"anonymous":false,"inputs": [{"indexed":
```

```
false, "name": "", "type": "uint256"}],  
"name": "Print", "type": "event"}];  
Undefined
```

Then, look at the code and ABI variables.

```
> code  
"0x608060405234801561001057600080fd5  
b5060fc8061001f6000396000f3fe6080604052600436106039576000357c  
0100000000000000000000000000000000000000000000000000000000000000000090  
048063c6888fa114603e575b600080fd5b348015604957600080fd5b50607  
360048036036020811015605e57600080fd5b81019080803590602001  
909291905050506089565b6040518082815260200191505060405180910  
390f35b60007f24abdb5865df5079dcc5ac590ff6f01d5c16edbc5fab4e195  
d9febdb1114503da6007830260405180828152602001915050604051809  
10390a160078202905091905056fea165627a7a723058205cfb8ba182658  
158a76b83c6ae1695a36cd2853022dfc1d4ad9d63ba5e27cf370029"  
> abi  
[  
  {  
    constant: false,  
    inputs: [  
      name: "input",  
      type: "uint256"  
    ],  
    name: "multiply",  
    outputs: [  
      name: "",  
      type: "uint256"  
    ],  
    payable: false,  
    stateMutability: "nonpayable",  
    type: "function"  
, {  
    anonymous: false,  
    inputs: [  
      indexed: false,  
      name: "",  
      type: "uint256"  
    ],  
    name: "Print",  
    type: "event"  
}]
```

Deploy the contract to Ethereum using the code and ABI generated above, and mining is required to confirm the transaction. You can see that the contract has been deployed and the contract address is 0x9305133e1d259a7bf4A61dcD4995b0DC29d2c3eB.

```
> var contractInstance = web3.eth.contract(abi).new  
({from: "0x29f9ceac9ad80fd672ddb3f9ea4bedd89  
c3360f5", "gas": "0x1d5b7", "gasPrice": "0x4a817c800", data:code})
```

After the contract is deployed, you can get the return value:

```
Submitted contract creation
fullhash=0x3e59133313cccd31ca069ed54bb79b8dc2d
f751970758201309e3d579c2114b
contract=0x9305133e1d259a7bf4A61dcD4995b0DC29d2c3eB
```

Then start mining and type contractInstance in the terminal to see its contents as follows.

```
{
  abi: [{{
    constant: false,
    inputs: [{...}],
    name: "multiply",
    outputs: [{...}],
    payable: false,
    stateMutability: "nonpayable",
    type: "function"
  }, {
    anonymous: false,
    inputs: [{...}],
    name: "Print",
    type: "event"
  }],
  address: "0x9305133e1d259a7bf4a61dc4995b0dc29d2c3eb",
  transactionHash: "0x58617ac3917f014b1623cf0d7fd8cb84756bc5
f781c144c64dd8858d1aaa4f4",
  Print: function(),
  allEvents: function(),
  multiply: function()
}
```

Calling Contract Methods

After the contract is successfully deployed, you can get the contract address, and use the ABI of the contract and the contract address to create a contract instance, and use the created contract instance to call the methods in the contract. The returned var multi is the contract instance:

```
> var multi =
web3.eth.contract(abi).at
("0x9305133e1d259a7bf4A61dcD4995b0DC29d2c3eB ")
```

The multiply() method in the contract is called using the contract instance multi, passing in argument 6. This step also requires mining to confirm:

```
> multi.multiply.sendTransaction(6, {from: "0x29f9ceac
9ad80fd672ddb3f9ea4bedd89c3360f5"}) ;
```

The return value is:

```
Submitted transaction
fullhash=0x357eb2b2519687ce8339a2a5a6db9df84da9eb0b1
2dc61f4829db7929aaed7d5 recipient=0x9305133e1d259a7bf4A6
1dcD495b0DC29d2c3eB
"0x357eb2b2519687ce8339
a2a5a6db9df84da9eb0b12dc61f4829db7929aaed7d5"
```

"0x357eb2b2519687ce8339a2a5a6db9df84da9eb0b12dc61f4829db7929aaed7-d5" is the transaction address.

Since calling the methods in the contract using web3 does not return the real calculated value, but only the transaction hash, the details of this transaction can be obtained through the Print event defined in the contract:

```
> multi.Print(function(err, data) {console.log(JSON.stringify
(data))})
```

The return transaction details are shown below. As expected, the 42 in args is the real value that needs to be returned after the calculation through the contract method.

```
{
  "address": "0x9305133e1d259
a7bf4a61dc4995b0dc29d2c3eb",
  "args": {
    "" : "42"
  },
  "blockHash": "0xa5c0c6a8ca0b
761e9775ff8f79d9f3a06d2578744869b8935728733dc728ad",
  "blockNumber": 118,
  "event": "Print",
  "logIndex": 0,
  "removed": false,
  "transactionHash": "0x2d88d4669
eb530826f33236ce90ffe92638dc8e2c221759258ec0a4229ad267",
  "transactionIndex": 0
}
```

3.4 DApp Development Framework and Process

Developing actual projects based on existing development frameworks can significantly accelerate the project progress. In the development of Ethereum decentralized applications, the more commonly used development frameworks are Meteor and

Truffle. This section introduces a layered and scalable development process. Developers can choose the project's appropriate development framework and process according to the type, scale, and difficulty.

3.4.1 Meteor

Meteor is a general-purpose WebApp front-end development framework that can very easily integrate with Ethereum's web3.js interface. Meteor is considered a full-stack framework implemented entirely in JavaScript and provides reloading, CSS injection, and support for pre-compilation (Less, CoffeeScript, etc.). Meteor can be very easy to build a single page application (single page App, SPA), write all the front-end code in index.html, and use a js file and css file added in the resource. Meteor supports responsive development, similar to AngularJS, can be very simple to build the interface.

There are many DApp applications developed based on the Meteor framework. The following describes how to install Meteor, load the Ethereum web3 module, call the web3.js interface, and deploy DApp applications.

Install Meteor

Meteor can be downloaded from the official website. Different operating systems have various download methods. Windows requires downloading the installation package, while macOS and Linux can be downloaded directly from the terminal using the following command line.

```
curl https://install.meteor.com/ | sh
```

The installation process is slow. After the installation, enter the following command in the terminal:

```
meteor -version
```

If the Meteor version number can be successfully displayed, then Meteor is successfully installed.

Loading the Ethereum web3 Module

Once Meteor is successfully installed, you can use it to develop DApps. First, create a Meteor project by executing the following command from the command line: meteor

```
meteor create project name
```

This process may be slow because it requires loading multiple initial modules. Then execute the following command to enter the project directory.

```
cd project name
```

Execute the following command to load web3:

```
meteor add ethereum:web3
```

If the above loading fails due to versioning, you can use the following command instead.

```
meteor npm install web3 -save
```

Because Ethereum's web3.js interface is easier to use and more mature, it is recommended to integrate web3.js interface in Meteor to develop DApps.

Calling the Ethereum web3.js Interface

After the Meteor project is created, there will be two file structures, client and server, by default, which correspond to the client-side program and server-side program, respectively. According to the actual needs, you can use web3.js on the client and server sides, respectively.

Create a new lib folder in the client folder, create a new init.js in the lib folder, and implement the following initialization web3 code in the init.js.

```
var Web3 = require('web3');
if (typeof web3 !== 'undefined') {
  web3 = new Web3(web3.currentProvider);
}
else {
  // Connect to an Ethereum client, such as an Ethereum private chain
  // running locally, default port 8545
  web3 = new Web3(new
    Web3.providers.HttpProvider("http://localhost:8545"));
}
console.log("client:" + web3.eth.accounts[0]);
```

After the client initializes the web3 instance, all the methods in the web3.js interface can be called, and a DApp framework can be set up. When the client is loaded in the browser, it will print out the first account of the current Ethereum in the Console panel.

You can implement the above code in server/main.js, so that you can call all the interfaces in web3.js on the server-side, and then realize the interaction with smart contracts.

Deploying DApp Applications

Before deploying a DApp application, you first need to open the Ethereum private chain in another terminal. Then you can automate the application deployment by executing the meteor command in the project directory. The command line for successful DApp deployment is shown below, where the printed server:0x90c2323cdeff75fd82e65ac496fc45eafadf4563 is the code implemented in server/main.js, and the printed address is the first account on the current Ethereum private chain.

```
→ meteor
[[[[ ~/Desktop/myapp ]]]]

=> Started proxy.
=> Started MongoDB.
I20170608-23:54:53.293(8)?
server:0x90c2323cdeff75fd82e65ac496fc45eafadf4563
=> Started your app.
=> App running at: http://localhost:3000/
```

The default port used by the DApp developed with Meteor is port 3000. You can access the DApp application by entering <http://localhost:3000> in the browser, as shown in Fig. 3.4.

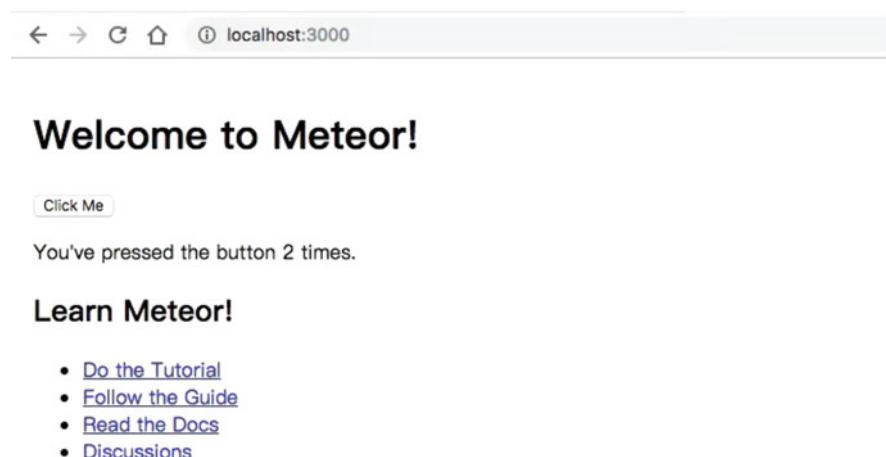
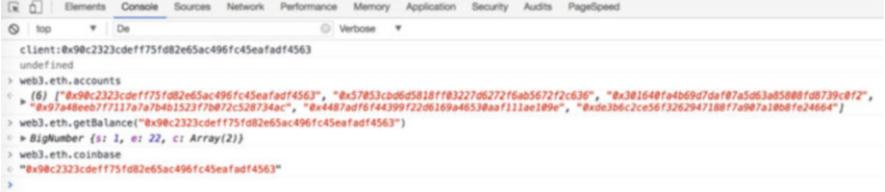


Fig. 3.4 Launching Meteor application



The screenshot shows the developer tools console in a browser. The tabs at the top are Elements, Console, Sources, Network, Performance, Memory, Application, Security, Audits, and PageSpeed. The Console tab is selected. The output in the console is:

```

client:@x9c2323cdeff75fd82e65ac496fc45eafadfadf4563
undefined
> web3.eth.accounts
< [Object]
> (6) ["0x9c2323cdeff75fd82e65ac496fc45eafadfadf4563", "0x57853cb0d5818ff03227d6272f6ab5672f2c636", "0xe381640fa4b069d7da07a5d3a858807db739c0f2", "0x957a48eeb77117a7a7b401523f70872c528734ac", "0x4487ad6f644399f22d6169a4653aaaf111ae189e", "0xde3bdc2ce56f326294718ff7a087a180dfe4664"]
> web3.eth.getBalance("0x9c2323cdeff75fd82e65ac496fc45eafadfadf4563")
> web3.eth.coinbase
< "0x9c2323cdeff75fd82e65ac496fc45eafadfadf4563"

```

Fig. 3.5 Using the web3.js interface in Meteor

Open your browser’s developer options, switch to the Console page, and you will see the printed message, the code implemented in client/lib/init.js, to obtain the first account in the current Ethereum, as shown in Fig. 3.5. It is possible to call web3.js interactively from the command line. This makes it possible to develop a simple DApp using Meteor.

3.4.2 Truffle

Truffle is a hugely popular development framework for Ethereum DApp development. Truffle allows you to use JavaScript for application development easily and use almost all the mechanisms in JavaScript, such as Promise and asynchronous calls. Truffle uses a JS Promise framework called Pudding on top of web3.js, so there is no need to load the web3.js library; you can effectively improve development efficiency. Truffle has a built-in smart contract compiler, which allows you to compile, link dynamic libraries, deploy, and test contracts using script commands, greatly simplifying the contract development lifecycle.

The following describes the installation of Truffle, project creation and client application operation and uses one of Truffle’s default contracts, MetaCoin, to demonstrate the compilation and deployment of smart contracts.

Truffle Installation

Truffle can be installed by executing the following command in the terminal, with the -g parameter indicating a global installation:

```
npm install -g truffle
```

After installation, type “truffle – version” in the terminal, and the Truffle version number will appear to indicate a successful installation. The version of Truffle used in this book is “Truffle v5.0.27 (core: 5.0.27)”.

```
→ truffle version
Truffle v5.0.27 (core: 5.0.27)
```

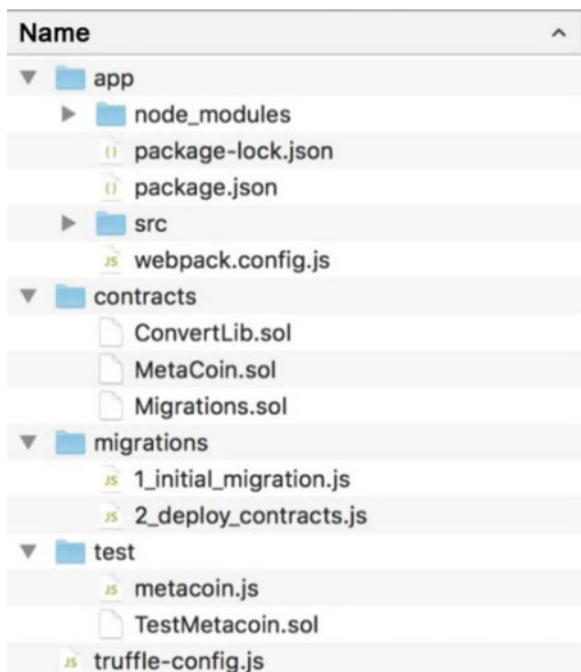
```
Solidity v0.5.0 (solc-js)
Node v10.16.0
Web3.js v1.0.0-beta.37
```

Truffle Project Creation

Enter an empty folder and use the truffle unbox webpack command to create a truffle project directory. Once created, the directory structure is shown in Fig. 3.6. Note that the command may fail to execute in different network environments. It is recommended to try it in different network environments.

- **app/** directory contains index.html front-end main page. The interaction design can be realized in index.html; javascripts includes index.js, all logical operations can be realized using JavaScript code; stylesheets are the CSS style files of the project.
- **contracts/** directory contains all the smart contracts; three contracts have been created by default.
- **migrations/** directory is the configuration file about contract deployment; if new contracts are added, you need to enter 2_deploy_contract for configuration.
- **the test/** directory is the contract test code, which enables unit testing of the contract.

Fig. 3.6 Truffle project directory structure



- **truffle.js** is the configuration file for the whole DApp project, including the host and port number configuration.
- **webpack.config.js** is used to configure front-end project related js files and Html files.

Smart Contract Compilation and Deployment

Truffle can automate the compilation and deployment of the contract, shielding the bytecode and ABI files generated during the contract compilation process. Before deploying the contract, you first need to change the port field in the develop attribute of the truffle-config.js file to 9545 and use the truffle develop command to enter the development mode console in the terminal. The following results will be returned.

```
→ truffle develop
Accounts :
(0) 0x7852c9d2dea82b942cef0837b6754b96e0e482c
(1) 0x7605d8b0017e25177a54db5fed4f5d5b42f150b9
(2) 0x8c3838e4fbe590314c2f6532945a2ebdeb6977f8
(3) 0x766d82ec6dacd7aeecb30dbfbe5644be59a3c7f6
(4) 0xbe6fd1e070af2d96f2cfede47a220f6868e807ca
(5) 0xa65a4d2dd90dab0f6204e7d1f5bff738d7bd1967
(6) 0x7e22b9fe37993d7675e94594cd7ed805cb3a8389
(7) 0x65f1f85a5141cd7aef362acd096c2bb094bcc9a6
(8) 0x381c2a1844de4a6520443b6774b69b53278a099d
(9) 0xffa6b400ac181fe3f1a83426233b9a5ee97e39ae

Private Keys :
(0) 98f457f5d635c36c80488eebf995a97f01e5ce52
f9697cb46174733c52608a57
(1) 72c8e654c5ec7583cb298d189829b16c00bdd05
7ec8bec22d1bf6517a25db465
(2) f50bb8eed05f00b65cf444f43a7ee5919dfc8c3d96
e3d6d501cc6cbe9614670
(3) 88735657137f47fb8c249417fa15fe5b64819c514d03
9702a8870fe6a8aac8d6
(4) c4143cb66a03c824867e3521ef3c6505421c5b462b5
d438bd76922fdbd4e04b4b
(5) 85d5dca87a41948de64615eabb1442dd181e697abed
db22e271212f75e25b6dd
(6) 471811b922b74140f62046470f7d3a9ec43155527d5906
38d34d728f7f176bb7
(7) 56919458d0544c6237890f0a2835ab60289ae1fe662ad85
822e922a664732bce
(8) 1c071643e22923ec39e147785dc32bad6fae8ba9fc614675
e065573e05ee308c
(9) 61d85f5c4d4eff8c631795238430772e5e2ed874fa7197735bf
6d5044b213331
```

Then enter the following command to implement compilation and deployment. Note that it is not necessary to prefix the command with truffle in this console.

```
truffle (develop) > compile
Compiling your contracts...
=====
> Compiling ./contracts/ConvertLib.sol
> Compiling ./contracts/MetaCoin.sol
> Compiling ./contracts/Migrations.sol
> Artifacts written to /Users/yqsh/WorkHistory/2019/July/
truffle_webpack/build/contracts
> Compiled successfully using:
- solc: 0.5.8+commit.23d335f2.Emscripten.clang

truffle (develop) > migrate
Starting migrations...
=====
> Network name: 'develop'
> Network id: 5777
> Block gas limit: 0x6691b7

1_initial_migration.js
=====

Deploying 'Migrations'
-----
> transaction hash: 0x108d10fc64d26097bdf8dbd2d
8ed1bd513706479819607dfd4bc8e7a1a280515
> Blocks: 0 Seconds: 0
> contract address: 0x724C128eF70320A43478601037D407C8acB4ae8F
> block number: 1
> block timestamp: 1563069054
> account: 0x7852c9D2DeA82B942cEfD0837B6754B96E0E482C
> balance: 99.99477214
> gas used: 261393
> gas price: 20 gwei
> value sent: 0 ETH
> total cost: 0.00522786 ETH

> Saving migration to chain.
> Saving artifacts
-----
> Total cost: 0.00522786 ETH

2_deploy_contracts.js
=====
Deploying 'ConvertLib'
-----
> transaction hash: 0x8e866cf1bbe2cd5377adb05fba
61d3f8d5eb7225beafcfc1bf7d408fc5c3d4f9d
> Blocks: 0 Seconds: 0
```

```

> contract address: 0xF348c11Dc62CD2D40f942213B963B0Ea25D13aC8
> block number: 3
> block timestamp: 1563069054
> account: 0x7852c9D2DeA82B942cEfD0837B6754B96E0E482C
> balance: 99.99185922
> gas used: 103623
> gas price: 20 gwei
> value sent: 0 ETH
> total cost: 0.00207246 ETH

Linking
-----
* Contract: MetaCoin <--> Library: ConvertLib (at address:
0xF348c11Dc62CD2D40f942213B963B0Ea25D13aC8)

Deploying 'MetaCoin'
-----
> transaction hash: 0xbc126160fc28e7dc1de4e275153d48d
0c6b8ddec9f53abfca33d5905b7c0cec8
> Blocks: 0 Seconds: 0
> contract address: 0xE5f425B77863e9Ea3A911e3e3B6cc5B0B41ff22E
> block number: 4
> block timestamp: 1563069054
> account: 0x7852c9D2DeA82B942cEfD0837B6754B96E0E482C
> balance: 99.98509224
> gas used: 338349
> gas price: 20 gwei
> value sent: 0 ETH
> total cost: 0.00676698 ETH

> Saving migration to chain.
> Saving artifacts
-----
> Total cost: 0.00883944 ETH

Summary
=====
> Total deployments: 3
> Final cost: 0.0140673 ETH

```

In the returned result, the contract address is the contract address of the three contracts Migrations, ConvertLib, MetaCoin. After starting the Truffle console each time, you need to enter migrate to deploy in order to run the project.

Client Application Running

After the contract is successfully deployed, you need to make a small change in app/scripts/index.js before running the service, changing the HttpProvider configuration port of Web3 from 9545 to 8545, then run the npm run dev command line to

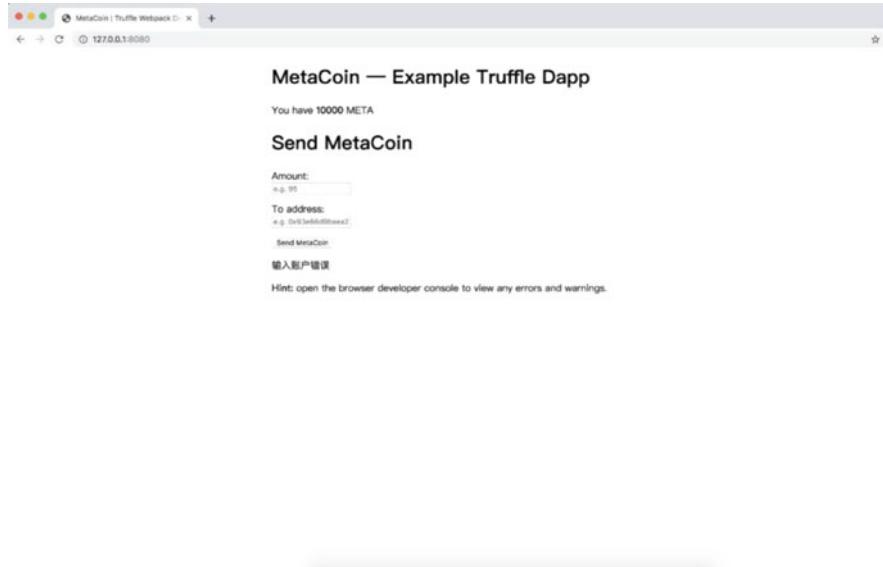


Fig. 3.7 Starting the Truffle application

turn on the service. In the process of starting the service, the webpack will have some warning, but it does not affect the regular operation of the program. You can ignore it.

```
→ cd app  
→ npm run dev
```

At this point, open a browser and type `http://localhost:8080` to see the Truffle application, as shown in Fig. 3.7.

MetaCoin, in the example, simulates a feature that can send tokens to other Ethereum accounts. We will cover MetaCoin applications and optimizations in Sect. 3.5. The universal credit system in Chap. 8 uses a decentralized application developed by the Truffle framework.

3.4.3 Layered Extensible Development Process

Using Meteor and Truffle frameworks to develop decentralized applications with web pages is very convenient, the architecture is relatively simple, and it can be deployed to servers quickly. However, the use of frameworks such as Truffle has significant limitations and one of the biggest problems is that the developed system is difficult to achieve cross-platform. If a client is required to run on browsers, mobile clients, or other terminals in an actual decentralized business project, Truffle

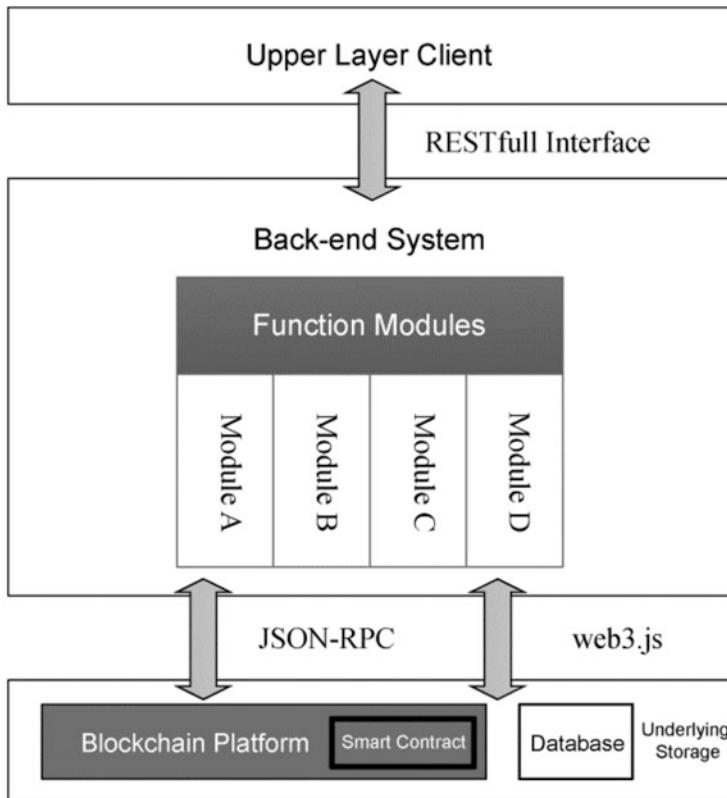


Fig. 3.8 Layered scalable architecture

cannot meet the demand. It is necessary to design a set of layered, scalable project development process, which has been widely used in the industry development process.

The general architecture of the scalable project development process is shown in Fig. 3.8.

The architecture is divided into three main layers, which we introduce separately below.

Underlying Storage

The underlying layer uses blockchain to ensure the security and tamper-proof of transactions. The platform can be the mainstream blockchain platform such as Ethereum, Hyperledger, and Hyperchain. The core business logic is all implemented by smart contracts, and the smart contracts are deployed to run on the blockchain. The database in the underlying storage is to make a complete backup of blockchain.

data and achieve disaster recovery. At the same time, because there is no good solution for block data visualization in the industry, adding a database can indirectly view the blockchain's data. In the actual development, dual writing of blockchain and database can be performed to achieve data synchronization.

Back-End Systems

The middle layer acts as a bridge between the upper application and the underlying blockchain. Back-end systems can interact with blockchain using various different interfaces, including JSON-RPC interface and web3.js interface. If you use Java like the back end, you can select JSON-RPC; If you use Node.js as the back end, you can select web3.js. For interface selection, please refer to Sect. 3.3. Since the key contract logic has been implemented in the smart contract, the primary function of the background implementation is data encoding, decoding, and forwarding, while providing RESTful interface invocation for the upper layer.

Upper Layer Client

The client provides direct users-oriented services. Clients can be broadly understood as browser pages, PC clients, mobile clients, etc. All of these clients can use the RESTful interface provided by the middle layer to interact with the blockchain. The middle layer services and the underlying blockchain are transparent to users and are a light client design. It does not require too many complex operations related to the blockchain to be implemented on the client-side, such as encryption and decryption, decoding and encoding. The client side generally only needs to handle the JSON data returned by RESTful.

Using this layered scalable project development process is in line with the idea of software engineering, where different layers allow different developers to implement designs. Importantly, it is also a highly cohesive and low-coupling approach. Cohesion is about how well elements within a module belong together and serve a common purpose, implying reuse and independence. Coupling is about how much one module depends on or interacts with other modules. The design principle of high cohesion and low coupling allows more independent modules to be reused and facilitates code optimization and extension. By a hierarchical and scalable model, different platforms only use interfaces for calling to reduce the degree of coupling to a minimum. The system can be developed in parallel by multiple people in the development, which is easy to connect and improves the development speed.

The e-coupon system in Chap. 8 and the accounts receivable management system in Chap. 10 is developed using the layered scalable development process.

3.5 The First Ethereum Application

This section takes the MetaCoin project generated by Truffle by default as the first Ethereum application. However, there are still imperfections in MetaCoin to be optimized, for example, verifying the validity of the entered account address and the feedback prompt when MetaCoin is insufficient.

3.5.1 Optimizing the MetaCoin Application

Verify the Validity of the Account Address

In the original MetaCoin project, it was possible to enter any character in the input address field and then send MetaCoin to these illegal addresses. These addresses are not the correct Ethereum accounts and do not match the actual need. When sending MetaCoin, the destination account needs to be verified to check if it is an account address in the current Ethereum client. Only after the verification is successful, the call contract method can be executed. Otherwise, a prompt message will be displayed.

The way to determine whether an account exists in the app/SRC/index.js file is isAccountCorrect().

```
const App = {
  ...
  isAccountCorrect: function(receiver) {
    const accounts = await web3.eth.getAccounts();
    for(let i = 0; i < accounts.length; i++) {
      if(receiver == accounts[i]) {
        return true;
      }
    }
    return false;
  },
  ...
}
```

Also, make the following judgments in the sendCoin() method:

```
const App = {
  ...
  sendCoin: async function() {
    const amount = parseInt(document.getElementById("amount").value);
    const receiver = document.getElementById("receiver").value;
    // Determine the validity of the account
    if(await this.isAccountCorrect(receiver)) {
      ...
    }
  }
}
```

```

        await sendCoin(receiver, amount).send({ from: this.account, gas:
10000000})
            this.refreshBalance();
        } else {
            this.setStatus("Error entering account");
        }
    },
...
}

```

Feedback Prompt When MetaCoin Is Insufficient

In the original application, if the MetaCoin balance was low and the sending operation continued, it would only prompt “Transaction complete!” There is no clear indication of insufficient balance, so it needs to be optimized.

In the contract, when the MetaCoin balance is insufficient, the message “MetaCoin insufficient, sending failed” is returned; otherwise, the letter “MetaCoin sent successfully” is returned. Use the Transfer() event in the sendCoin() method of the smart contract to return the message.

```

event Transfer(address indexed _from, string message);
function sendCoin(address receiver, uint amount) public returns(bool
sufficient) {
    if (balances[msg.sender] < amount) {
        // sending failed
        ...
        return false;
    } else {
        // sending failed
        ...
        emit Transfer(msg.sender, "MetaCoin sent successfully");
        return true;
    }
}

```

In the app/src/index.js file, when the contract instance calls the contract event Transfer(), it gets the data returned from the contract:

```

const App = {
    ...
    sendCoin: async function() {
        ...
        this.meta.events.Transfer()
            .on("data", function(event) {
                // The event contains all the data returned from the contract, which
can be obtained by event.returnValue
                // Send a success or failure message
                ...
            })
            .on("error", function(error) {
                ...
            })
    }
}

```

```
    ...
  });
  await sendCoin(receiver, amount).send({ from: this.account });
  ...
}

...
}
```

3.5.2 *MetaCoin Code in Detail*

The code structure and implementation of MetaCoin applications are further described below, including contract code, front-end HTML, JavaScript interface calls, and points to note in development.

Implementation of index.html

index.html is the main interface of the whole application. Since it is a single-page application, all the interface elements are implemented in index.html. The index.html needs to load the JavaScript code, the index.js in the application; therefore, it needs to be implemented in index.html as follows.

```
<!DOCTYPE html>
<html>
  <head>
    <title> the first Ethereum application MetaCoin</title>
  </head>
  <style>
    input {
      display: block;
      margin-bottom: 12px;
    }
  </style>
  <body>
    <div style="width:800px; margin: 0 auto;">
      <h1>MetaCoin – Example Truffle Dapp</h1>
      <p>You have <strong class="balance">loading...</strong> META</p>

      <h1>Send MetaCoin</h1>

      <label for="amount">Amount:</label>
      <input type="text" id="amount" placeholder="e.g. 95" />

      <label for="receiver">To address:</label>
      <input
        type="text"

```

```

        id="receiver"
placeholder="e.g. 0x93e66d9baea28c17d9fc393b53e3fbdd76899dae"
/>

<button onclick="App.sendCoin()">Send MetaCoin</button>

<p id="status"></p>
<p>
<strong>Hint:</strong> open the browser developer console to view
any
    errors and warnings.
</p>
</div>

<script src="index.js"></script>
</body>
</html>

```

The `<title>` tag defines the title of the page displayed in the browser, the `<style>` tag defines the CSS information, and the `<script>` tag loads the JavaScript script file, `index.js`.

Implementation of Contract Code

Since the MetaCoin contract is relatively simple, this is a preliminary introduction, and the complete implementation of the MetaCoin contract is as follows.

The mapping, similar to a hash, is a data structure of key-value pairs, where the left side of “`=>`” is the type of the input key and the right side is the type of the value to be found. The account of address type here is used to find the MetaCoin balance of uint type.

`constructor()` is the constructor method in which the initial balance of the transaction initiator is initialized to 10,000. Newer versions of the Solidity compiler recommend using `constructor()` to declare constructor methods instead of the traditional way of using functions.

`sendCoin()` is a method to send MetaCoin, which will fail to send when the balance is low. There are two types of methods in this contract: transaction methods and view methods. A transaction method that will change the state of Ethereum is a method that will modify the state variable. Executing this method is also called completing a transaction and requires gas. The `sendCoin()` method, in this case, is a transaction method. When using web3 to call a transaction method, it is impossible to get the method’s real return value. For example, the bool value returned by the `sendCoin()` method is not available in the web3 interface call, and the real value returned is an object that stores transaction information. However, the return value of the transaction method can be retrieved during the internal method call of the contract, so you can use the event to return data to web3, such as a “send failed” string.

The `getBalance()` method is a view method that only gets the value of Ethereum data and variables but does not change the Ethereum state. A method can be manually decorated as a view, and Ethereum will automatically recognize which methods are viewed if it does not have a view modifier. For the view method, the return value can be successfully received using the `web3` interface, such as the balance of `uint` type returned by the `getBalance()` method, which can be obtained in `index.js`.

```
contract MetaCoin {
    // Find your account balance by account address
    mapping (address => uint) balances;
    // Initial account balance is 10000
    constructor() public {
        balances[msg.sender] = 10000;
    }
    // event returns data to web3
    event Transfer(address indexed _from, string message);
    function sendCoin(address receiver, uint amount) public returns
(bool sufficient) {
        if (balances[msg.sender] < amount) {
            // Failed to send
            emit Transfer(msg.sender, " Insufficient MetaCoin, failed to
send");
            return false;
        } else {
            // Failed to send
            balances[msg.sender] -= amount;
            balances[receiver] += amount;
            emit Transfer(msg.sender, " MetaCoin sent successfully");
            return true;
        }
    }
    // Get the number of MetaCoin accounts multiplied by 2
    function getBalanceInEth(address addr) public view returns(uint) {
        return ConvertLib.convert(getBalance(addr),2);
    }
    // Get account MetaCoin
    function getBalance(address addr) public view returns(uint) {
        return balances[addr];
    }
}
```

Implementation of index.js

`index.js` handles all the business logic, and all the `web3` interface methods are called here. `index.js` is equivalent to an intermediate layer that receives the input from the front-end page and calls the contract methods to process it, then gets the processing results of the contract methods and displays them on the page.

When the page is loaded, JavaScript executes the window.onload() method, and since this is a private chain and Web3.providers.HttpProvider does not support subscribing to Transfer events, so the Web3.providers.WebsocketProvider service will be enabled.

```
const App = {
  web3: null,
  account: null,
  meta: null,
  start: function() {...},
  refreshBalance: function() {...},
  sendCoin: function() {...},
  setStatus: function() {...},
  isAccountCorrect: function() {...}
}
window.App = App;
window.addEventListener("load", function() {
  if (window.ethereum) {
    App.web3 = new Web3(window.ethereum);
    window.ethereum.enable();
  } else {
    App.web3 = new Web3(
      // new Web3.providers.HttpProvider("http://127.0.0.1:9545"),
      new Web3.providers.WebsocketProvider('ws://localhost:9545'),
    );
  }
  App.start();
});
```

The App object is responsible for the transaction logic in the front-end business, which has three properties (web3, account, and meta) and five methods. Among them, web3 is an instance object of Web3, the account is the current account, and meta is a contract instance.

The start() method is used to get the current account, contract instance, and balance. Get the ABI and deployment network address needed to build the contract from the compiled meta-json file, instantiate a contract object with web3.eth.contract(), and get all the accounts in the current Ethereum client (private chain) using the web3.eth.getAccounts() method. Set the account property as the first account in the account data, and the default account for subsequent initiation of contract methods.

```
App = {
  ...
  start: async function() {
    const { web3 } = this;
    try {
      // Get contract instance
      const networkId = await web3.eth.net.getId();
      const deployedNetwork = metaCoinArtifact.networks[networkId];
      this.meta = new web3.eth.Contract(
```

```

        metaCoinArtifact.abi,
        deployedNetwork.address,
    );
// Get the account
try{
    const accounts = await web3.eth.getAccounts();
    if(accounts.length == 0) {
        alert("Ethereum account is empty");
        return;
    }
    console.log("accounts", accounts);
    this.account = accounts[0];
    this.refreshBalance();
} catch(err) {
    console.error("Failed to obtain Ethereum account");
}
} catch (error) {
    console.error("No connection to contracts or blocks");
}
},
...
}
}

```

`refreshBalance()` is the method to refresh the balance value displayed on the screen. `getBalance` gets the balance data for a given account. Because you need to send account information to the contract, you use the `call()` method call. `this.account` is the account that initiates the transaction, which is encrypted with the account's private key. `Balance` is the received account balance data. An error message will be caught when a sending error occurs, and "Failed to get balance" will be displayed. The return value of the view method is received in the function return value.

```

const App = {
    ...
    refreshBalance: async function() {
        const { getBalance } = this.meta.methods;
        try{
            const balance = await getBalance(this.account).call();
            const balanceElement = document.getElementsByClassName
("balance") [0];
            balanceElement.innerHTML = balance;
        } catch(err) {
            console.log(err);
            this.setStatus("Failed to get balance");
        }
    },
    ...
}

```

The `sendCoin` function is used to send the MetaCoin. When calling the transaction method, the `from` parameter cannot be omitted and must be explicitly declared as originating from the account. A `gas` parameter is also required. Executing the

transaction method on Ethereum requires a certain amount of gas. When the gas parameter is not specified, a default gas value will be sent. The method generally executes successfully. However, if a contract method has too much code, an OOG (out of gas) error may occur, resulting in call failure. The solution is to explicitly send a large gas value. The EventEmitter object returned in meta.events.transfer () is used to obtain various data for this transaction. At the same time, the return value of event is displayed in the way of alert pop-up dialog box for easy observation.

```
const App = {
    ...
    sendCoin: async function() {
        const amount = parseInt(document.getElementById("amount").value);
        const receiver = document.getElementById("receiver").value;
        // Determine the account validity
        if(await this.isAccountCorrect(receiver)) {
            this.setStatus("Sending is in progress. Please wait.....");
            const { sendCoin } = this.meta.methods;
            let self = this;
            this.meta.events.Transfer()
                .on("data", function(event) {
                    // The event contains all the data returned from the transaction,
                    which can be obtained by event.returnValue
                    // Send a success or failure message
                    const _from = event.returnValues[0];
                    const message = event.returnValues[1];
                    alert("Transfer() Event return:" + JSON.stringify(event));
                    self.setStatus(message);
                })
                .on("error", function(error) {
                    console.log(error);
                    self.setStatus("Failed to send transaction");
                });
            await sendCoin(receiver, amount).send({ from: this.account, gas: 1000000});
            this.refreshBalance();
        } else {
            this.setStatus("Error entering account");
        }
    },
    ...
}
```

3.6 Deploying to the Ethereum Public Chain (Mainnet)

This section describes how to deploy the Sect. 3.5 application to the public chain. Ethereum has several test chains and the main chain, which can be checked out on the etherscan website. Ethereum Mainnet is the main chain. Before deploying, make sure you have enough Ethers in your account. After all, it would help if you

consumed Ether when deploying. Then you can test the deployment on the test chain (Ropsten, Rinkeby, Kovan, etc.) to see how many Ethers you have to consume prior to deploying to the main chain.

3.6.1 Infura

Before deploying an Ethereum smart contract, you need to set up your Ethereum node, and due to a large number of blocks, the synchronization process to become a full node or light node is too cumbersome. It is recommended to adopt the nodes provided by Infura as the deployment target; in this way, there is no need to set up your Ethereum node. In addition, because Infura does not manage your private keys, there are no security issues associated with hosting. Meanwhile, Infura uses HDWalletProvider to sign transactions, and NPM has a dependency package for this.

Infura's registration screen is displayed in Fig. 3.9. Visit the official website, fill in the form to register an account, and then you can choose the necessary packages.

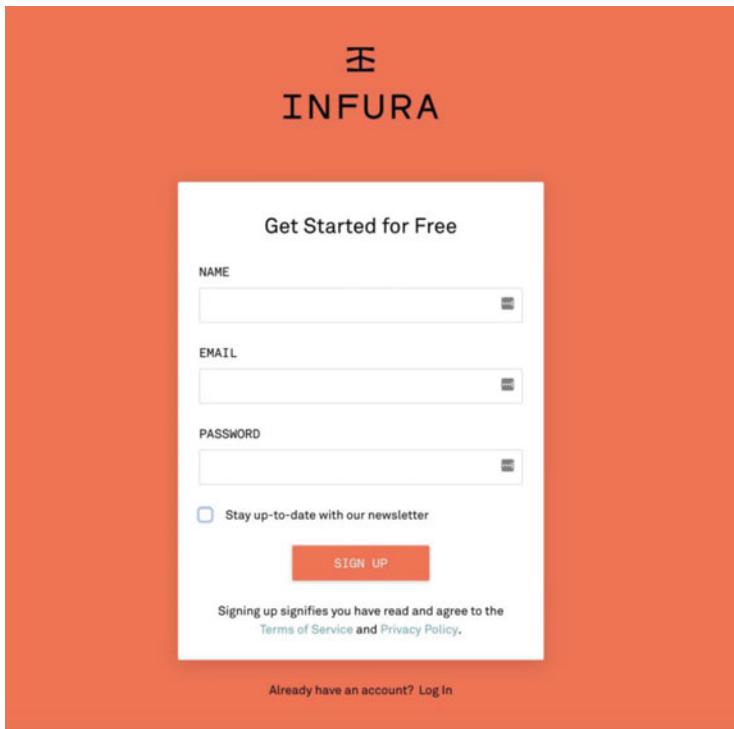


Fig. 3.9 Infura registration interface

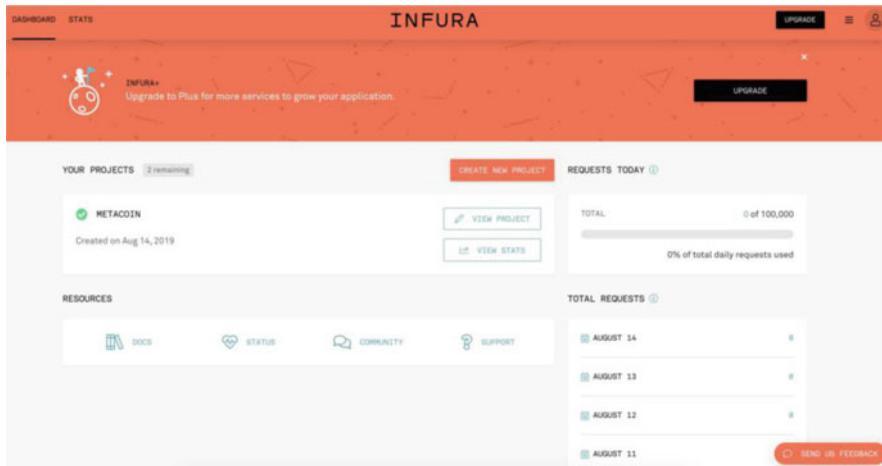


Fig. 3.10 Infura creates a project



Fig. 3.11 Project-related information

You can create your application by CREATE NEW PROJECT and click VIEW PROJECT to get the corresponding information, as shown in Fig. 3.10.

The relevant information to note on the Project page is PROJECT ID and PROJECT SECRET, as shown in Fig. 3.11, which will be applied in the truffle-config.js file.

3.6.2 Project Configuration

First, install the HDWalletProvider dependencies with the following command:

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

Then find truffle-config.js in the MetaCoin project and add the following information:

```
const HDWalletProvider = require('truffle-hdwallet-provider');
const fs = require('fs');
const mnemonic = fs.readFileSync(".secret").toString().trim();
const projectID = fs.readFileSync(".projectID").toString().trim();
```

The .secret file and .projectID file store the previous PROJECT SECRET and PROJECT ID data on Infura, respectively.

Finally, add the network configuration information about mainnet in networks:

```
module.exports = {
  networks: {
    ...
    mainnet: {
      provider: () => new HDWalletProvider(mnemonic,
        `https://mainnet.infura.io/v3/${projectID}`),
      from: "0xb6109Df2ACBE91d5660bdecf28D960f0A5F07463",
      gas: 8500000,
      gasPrice: 200000000000,
      confirmations: 2,
      network_id: 1,
      websockets: true
    }
    ...
  }
}
```

Note that you must make sure that the address in the from is available in Ether; otherwise, the deployment will fail.

3.6.3 Deploying MetaCoin

Open a terminal, go to the MetaCoin project, and enter the following command to deploy to the Ethereum public chain.

```
truffle migrate --network mainnet
```

At the end, the following summary will appear, indicating that the deployment is complete:

```
Summary
=====
> Total deployments:  3
> Final cost:        0.01486752 ETH
```

Finally, open another terminal, enter the App, enter “npm run dev” to start the project, enter “localhost:8080” in the browser; you will see the page display similar to Sect. 3.5, but the transaction data in Sect. 3.5 is on the private chain. Any transaction in the project will be written to Ethereum’s public chain; thus, it will take a few seconds to respond.

If you successfully perform some operations on the page, you have built a complete application on the Ethereum public chain.

3.7 Summary

This chapter firstly introduces how to set up the development environment of Ethereum, including Go language environment, the configuration of Node.js and npm, installation of solc compiler, and how to adopt Ethereum geth client to build private chain; it then explains the integrated development environment for Ethereum smart contract development, including Mix IDE and an online real-time compiler. Next, it describes two Ethereum programming interfaces, JSON-RPC and JavaScript API. The above can realize the interaction with Ethereum’s bottom layer, and acknowledge the contract method call. It introduces the mainstream Ethereum development frameworks and processes, including Metero, Truffle, and layered scalable development processes in commercial development. Finally, it has a relatively complete example of Ethereum application development built onto the public chain.

Chapter 4

In-Depth Interpretation of Hyperledger Fabric



Bitcoin is considered by many to be the powerful emerging platform of the 1.0 era of blockchain technology. With the birth of the Ethereum platform, primarily characterized by smart contracts, blockchain has entered the 2.0 era. In contrast, the Hyperledger Fabric blockchain technology, an open-source platform, marks the arrival of the 3.0 era of blockchain technology. The latest release of Fabric v1.4.1 (LTS) proposes many new design concepts, adds many new features, provides a highly modular and configurable architecture, supports common programming languages (such as Java, Go, and Node.js) for writing smart contracts, and supports unpluggable consensus protocols, making the development of enterprise-level applications based on this platform a reality, and the platform captures growing attention. This chapter will take readers into the world of Hyperledger Fabric and explore the basic operating principles to deepen their understanding of the platform and lay the foundation for subsequent study of Fabric-based application development technologies.

4.1 Project Introduction

The Hyperledger project is an open-source project dedicated to advancing blockchain digital technology and transaction validation to bring together members of the open-source community to collaborate on building open platforms that meet the needs of users in different industries and simplify business processes. The project enables the exchange of value in virtual and digital forms by establishing open standards for distributed ledgers.

4.1.1 Project Background

Backed by blockchain technology, “digital cryptocurrencies” such as Bitcoin have become widely accepted, with the number of active users and transactions growing faster than people expect. Many entrepreneurs, companies, and financial institutions realize the value of blockchain technology and widely believe that it can be used for more than just “digital cryptocurrency.”

For this reason, Vitalik established the Ethereum project with the hope of creating a Turing-complete smart contract programming platform that would make it better and more accessible for blockchain enthusiasts to build blockchain applications. Following that, many new blockchain applications emerged in the market, such as asset registration, prediction market, identity authentication, and others. However, blockchain technology then still had some insurmountable problems. For example, the transaction efficiency was low. The entire Bitcoin network could only support about seven transactions per second; secondly, the certainty of transactions could not be well guaranteed; finally, the mining mechanism to reach consensus would cause a tremendous waste of resources. These problems made blockchain technology unable to meet the needs of most commercial applications.

Therefore, the design and implementation of a blockchain platform to meet business needs played a key role in the development of blockchain. Amidst the strong calls from all walks of life, the Linux Foundation open-source organization launched an open-source project called Hyperledger in December 2015 to develop cross-industry blockchain by cooperating with all parties to jointly build an enterprise-level application platform for blockchain technology.

At its inception, Hyperledger attracted many famous companies, such as IBM, Cisco, Intel, and other tech Internet giants. Meanwhile, financial industry giants like Wells Fargo and JP Morgan Chase ranked among the first group. Its development community has now grown to over 270 organizations. It is worth mentioning that more than 1/4 of the project members are from China such as start-up blockchain companies of Funchain Technology, Antshares, and Bubi Tech, along with well-known companies, such as Wanda, Huawei, and China Merchants Bank involved. From the perspective of the membership lineup, the Hyperledger open-source project is compelling, bringing together many elites from various industries to collectively collaborate on solutions and advance the development of enterprise-class blockchain platforms.

The Hyperledger project, for the first time, proposes and implements complete member rights management, innovative consensus algorithms, and pluggable frameworks, exerting a profound impact on the advancement of blockchain-related technologies and industries. It already displays that blockchain technology is not simply an open-source technology anymore. It has been formally recognized by mainstream organizations and markets in other sectors.

4.1.2 *Project Introduction*

Hyperledger, as a large open-source project, expects to jointly promote the development of blockchain technology in commercial applications through cooperation with all parties. In terms of composition structure, it contains a large number of related specific subprojects. These subprojects can be independent projects or associated ones, such as building tools and blockchain browsers. Hyperledger does not give many constraints on subprojects; as long as there is a good concept related, you can send an application proposal to the Hyperledger committee.

The official address of the project is hosted on the Linux Foundation website, the code is hosted on Gerrit, and code images are available through GitHub. To better manage subprojects and develop the project, the Hyperledger project has established a body called the Technical Steering Committee (TSC), the highest authority that will make important decisions regarding the management of subprojects and the development of the entire ecosystem. The Hyperledger project adopts a lifecycle approach to manage its subprojects, giving each a lifecycle that facilitates the operation and management of the project. The lifecycle is divided into five phases: proposal, incubation, active, deprecated, and end of life. Each project will only have one step at one point in time during the development process. Projects do not necessarily follow the above sequence; they may remain in a particular phase or change between phases for specific reasons. All projects should focus on a modular design that includes technical scenarios for transactions, contracts, consistency, identity, storage, and code readability to ensure that new features and modules be easily added and extended. In addition, new projects need to be added and evolved to meet increasingly commercial needs and rich application scenarios.

At the time of writing, in total of 14 subprojects are running under the Hyperledger, details of which are shown in Table 4.1.

A few of the essential subprojects are described in more detail.

Fabric

Fabric is an implementation of blockchain technology and is a platform for distributed ledger solutions based on transaction calls and digital events. More than other blockchain technology implementations, it adopts a modular architectural design that supports the development and use of pluggable components. Multiple participating nodes jointly maintain the data on its ledger, and once recorded, the transaction information on the ledger can never be tampered with and supports traceability queries via timestamps. As for other public chains, Fabric introduces a member management service, so each participant must prove identity before accessing the Fabric system by offering a corresponding certificate and submitting the multichannel multi-ledger design to enhance the security and privacy of the system. Compared with Ethereum, Fabric prefers powerful Docker container technology to run the service and supports a more convenient and powerful smart contract service.

Table 4.1 Hyperledger subproject information table

Item name	Status	Dependency	Description
Hyperledger Aries	Incubation	Fabric	Blockchain infrastructure for peer-to-peer interaction
Hyperledger Burrow	Incubation		Licensed Ethereum smart contract blockchain
Hyperledger Caliper	Incubation		Blockchain benchmark framework
Hyperledger Cello	Incubation	Fabric (Sawtooth, Iroha)	Blockchain management/operations
Hyperledger Composer	Incubation	Fabric (Sawtooth, Iroha)	Development framework/tools for building blockchain business networks
Hyperledger Explorer	Incubation	Fabric (Sawtooth, Iroha)	Blockchain Web UI
Hyperledger Fabric	Active		Distributed ledger in Golang
Hyperledger Grid	Incubation	Fabric	Platform for building supply chain solutions with distributed ledger components
Hyperledger Indy	Active		Distributed ledger to decentralize identities
Hyperledger Iroha	Active		Distributed ledger in C++
Hyperledger Quilt	Incubation		Interoperability solutions for blockchain, DLT, and other types of ledgers
Hyperledger Sawtooth	Active		Distributed ledger with multi-language support
Hyperledger Transact Home	Incubation		Transaction execution platform
Hyperledger Ursa	Incubation		A shared cryptographic library

Ethereum can write contracts through the provided Solidity language only, while Fabric supports multi-language contract writing, such as Go, Java, and Node.js. In addition, Fabric offers a multilingual SDK development interface, allowing developers to freely and efficiently use the blockchain services it provides. The architecture and operation of Fabric will be analyzed in-depth later in this chapter.

Iroha

Iroha is a distributed ledger project inspired by Fabric architecture. Being approved by the Technical Steering Committee on October 13, 2016, it entered the incubation phase and moved out of the incubation area on May 18, 2017. It aims to provide C++ and mobile application developers with a development environment for the Hyperledger project. The project wants to implement reusable components of Fabric, Sawtooth Lake, and other potential blockchain projects in C++, and these

components can be called in Go. Iroha complements existing projects with the long-term goal of implementing a robust library of reusable components that will allow Hyperledger technology projects to freely select and use these reusable elements when running distributed ledgers.

Sawtooth Lake

Approved by the TSC on April 14, 2016, and moved out of the incubation area on May 18, 2017, Sawtooth Lake is an Intel-initiated modular distributed ledger platform experiment designed for versatility and scalability. Sawtooth Lake provides a modular platform for building, deploying, and running distributed ledgers, supporting licensed and unlicensed chain deployments. It includes a new consensus algorithm, PoET, which, like the proof-of-work algorithm used by Bitcoin, randomly selects a node according to specific rules to be the bookkeeper of a block. In contrast, other nodes verify the block and execute the result. The difference is that PoET does not consume arithmetic power and energy but requires CPU hardware to support the SGX (software guard extensions). Due to the hardware limitations of the PoET algorithm, it is at the current stage only suitable for the production environments for the time being.

Blockchain Explorer

The Blockchain Explorer project aims to create a user-friendly web application for Hyperledger to query information on the Hyperledger blockchain, including block information, transaction-related data information, network information, chaincode, and related information stored in the distributed ledger. The project was approved by the TSC on August 11, 2016, after which it was launched into the incubation phase.

Cello

The Cello project entered incubation after being approved by the TSC on January 5, 2017. Cello aims to provide a blockchain as a service (BaaS) to reduce the effort of manually manipulating (creating and destroying) blockchains. With Cello, operators can create and manage blockchains in the way of a dashboard, while users (chaincode developers) can immediately access blockchain information through a single request. In other words, it provides an easy and convenient blockchain operation platform for operators.

4.2 Introduction to Fabric

Hyperledger Fabric is a unique implementation of Distributed Ledger Technology (DLT) that provides enterprise-grade network security, scalability, confidentiality, and high performance on top of modular blockchain architecture. The latest version of Fabric is v1.4.1 (LTS). Compared with the previous v0.6, v1.4 has made many improvements in security, confidentiality, deployment, maintenance, and actual business scenario requirements, such as the functional separation of Peer nodes in architecture design and the privacy isolation of multiple channels and pluggable implementation of consensus. The available introduction of Raft crash tolerance ordering service improves maintainability and operability, adds private data support, etc., offering better Fabric support. Therefore, contents on Fabric later in this book will be described based on the v1.4 version.

Hyperledger Fabric v1.4 has the following features.

- **Identity Management.** Fabric blockchain is a network of license chains. Fabric thus renders a membership service that manages user IDs and authenticates all participants on the web. In a Hyperledger Fabric blockchain network, members can be identified by identity. Still, they do not know what each other is doing, which is the confidentiality and privacy that Fabric serves.
- **Privacy and confidentiality.** Hyperledger Fabric enables competing commercial organizations and other arbitrary parties with privacy and confidentiality requirements for transaction to coexist in the same license chain network. It limits the transmission path of messages through channels and provides privacy and confidentiality protection for network members. All data in the medium, including transactions, members, and channel information, are invisible and inaccessible to network entities that do not subscribe to the channel.
- **Efficient processing.** Hyperledger Fabric assigns network roles based on node types. For better network concurrency and parallelism, Fabric effectively separates transaction execution, transaction ordering, and transaction commit. By executing transactions before sorting, each Peer node can process multiple transactions at the same time. Such concurrent performance dramatically improves the processing efficiency of Peer nodes and speeds up the delivery process of transactions to ordering services.
- **Chaincode functionality.** The smart contract code is the coded logic of the transaction invocation in the channel, defining the parameters used to change the ownership of the asset, and ensuring that all transactions for digital asset ownership transfer are subject to the same rules and requirements.
- **Modular design.** The modular architecture implemented by Hyperledger Fabric can provide network designers with available options. For example, specific identification, consensus, and encryption algorithms can be plugged into Fabric networks as pluggable components. Any industry or public domain can adopt a typical blockchain architecture and ensure that its networks are interoperable across the market, regulatory, and geographic boundaries.

- **Serviceability and operations.** Improvements to logging and the addition of health check mechanisms and operations metrics make V1.4 a huge leap forward in maintainability and operability. The new RESTful Operations Service monitors and manages peer node and ordering service node operations by offering three services for production operators. The first uses the logging/logspec endpoint, enabling operators to dynamically obtain and set the logging level of peer nodes and ordering service nodes; the second adopts the health check/healthz endpoint, allowing operators and business process containers to check the activity and health of peer nodes and ordering service nodes. The third allows operators to use Prometheus to record utilization metrics from peer and ordering service nodes by the operational metrics/metrics endpoint.

4.3 Core Concepts

This section introduces the core concepts employed by Hyperledger Fabric.

4.3.1 *Anchor Nodes*

The Gossip protocol makes use of anchor nodes to ensure that peer nodes in different organizations interact with each other. When a configuration block containing an update to the anchor node is submitted, the peer node can detect and be informed of all peers known to the anchor node. Since organizations communicate via Gossip, at least one anchor node must be defined in the channel configuration. Providing a set of anchor nodes for each organization allows for high availability and reduced redundancy.

4.3.2 *Access Control List*

Access control lists (ACL) associate access to specific peer node resources, such as system chaincode APIs or transaction services, with policies that specify the number and type of organizations or roles required. As a part of channel configuration, ACLs can be updated through the standard configuration update mechanism.

4.3.3 *Blocks*

A block contains an ordered set of transactions established by a consensus system and verified by peer nodes. It is linked cryptographically in the channel, first to the

preorder blocks and then to the postorder blocks. The first block is called the Genesis block.

4.3.4 *Blockchain*

A blockchain is a transaction log-structured by hashing the transaction blocks together. After receiving a transaction block from the ordering service, the peer node marks the block as valid or invalid based on endorsement strategy and concurrency conflicts. It appends the block to the hash chain of the peer node's file system.

4.3.5 *Smart Contracts*

Smart contracts are codes invoked by clients external to the blockchain network and can be used to manage access and modification of key-value pairs in the state of the world, installed on peer nodes and instantiated on the channel. Smart contracts are also referred to as chaincodes.

4.3.6 *Channels*

Channels are private blockchains built on the Fabric network, defined by configuration blocks that guarantee data isolation and privacy. All peer nodes share a specific ledger in the channel, and the correctness of the channel must verify the interaction between the counterparty and the ledger.

4.3.7 *Submit*

Each peer in a channel verifies the orderliness of the transaction blocks and then commits (writes or attaches) the blocks to each copy of the ledger on that channel. Peer nodes also mark whether the transaction is valid or not.

4.3.8 *Concurrent Control Version Checking*

Concurrency Control Version Checking (CCVC) keeps the state of peer nodes in the channel synchronized. Peer nodes execute transactions in parallel, and before a

transaction is submitted to the ledger, the peer node checks whether the data read during the execution of the transaction has been modified. If it is modified, a CCVC conflict is raised and the transaction is marked as invalid in the ledger and its value is not updated to the state database.

4.3.9 Configuration Blocks

It contains configuration data for system chains (ordering services) or channel definition members and policies. Configuration changes to a channel or the network as a whole (e.g., members leaving or joining) will result in a new configuration block being generated and appended to the appropriate chain. This configuration block will contain the contents of the founding block plus the increment.

4.3.10 Consensus

Consensus is used to confirm the ordering of transactions and the correctness of the transaction set.

4.3.11 Consent Setting

In the Raft ordering service, the consent set is the ordering nodes on the channel vigorously attending the consensus mechanism. If other sorting nodes exist on the system channel but are not part of the channel, these sorting nodes are not part of the channel's sorting set.

4.3.12 Consortium

Consortia are collections of disordered organizations on a blockchain network. These collections that form and join channels have their peer nodes, and while blockchain networks can have multiple federations, most networks have only one federate. At the time of creation, all organizations joining the channel must be part of the consortium. Organizations that are not defined in the consortium may be added to an existing pipeline.

4.3.13 *World State*

The world state, also known as the ledger's current state data, represents the latest values for all keys ever included in the chain transaction log. The peer node updates the value corresponding to each recently processed transaction modified to the world state of the ledger. Since the world state provides direct access to the latest value of the key, rather than by traversing the entire transaction log, the chaincode must first know the world state of the key value and then execute a transaction proposal against this world state.

4.3.14 *Dynamic Membership Management*

The fabric supports the dynamic addition/removal of members, peer nodes, and ordering service nodes without affecting network-wide operability. Dynamic membership management is critical when business relationships are adjusted or when entities need to be added/removed for various reasons.

4.3.15 *Genesis Blocks*

The Genesis block is the configuration block that initializes the blockchain network or channel and is the first block on the blockchain.

4.3.16 *Gossip Protocol*

Gossip data transfer protocol contains three functions: managing peer nodes, discovering members on the channel, broadcast of ledger data between all peer nodes on the channel, and synchronization of ledger data between all peer nodes on the channel.

4.3.17 *The Ledger*

The ledger consists of a chain of blocks and the world state. The blockchain is immutable; it cannot be changed once a block is added to the chain. On the other hand, the world state is a database containing the current values of the set of keys that have been added, modified, or deleted by the set of validation and commit transactions in the blockchain. Each channel in the network has a logical ledger; in fact,

each peer node in the channel maintains its copy of the ledger, logically single through a consensus process with copies of other peers, but with many identical copies distributed among a set of network nodes (peers and ordering services). The term Distributed Ledger Technology (DLT) is commonly associated with such a ledger.

4.3.18 *Followers*

In leader-based consensus protocols such as Raft, followers copy nodes of log entries generated by the leader. In Raft, followers will receive “heartbeat” messages from the leader. If the leader stops sending these messages for a configurable amount of time, followers will initiate a leader election, and one of them will be chosen as the leader.

4.3.19 *Leaders*

In leader-based consensus protocols (such as Raft), the leader takes on extracting new log records, replicating them to follower consensus nodes, and managing when the documents are considered committed.

4.3.20 *Principal Peer Node*

Each organization can obtain multiple peer nodes on the channel they subscribe to, with at least one of them acting as the primary peer node to communicate with the network ordering service on behalf of the organization. The ordering service provides blocks to the primary peer node on the channel and then distributes them to other peer nodes within the same organization.

4.3.21 *Logging*

Logging is the primary unit of work in the Raft ordering service, distributed from the leader to the followers. The complete sequence of these records is called a “log.” A log is considered to be consistent if all members agree on the documents and their ordering.

4.3.22 *Membership Service Provider*

Membership Service Provider (MSP) stands for a system abstraction component that provides certificates to client and peer nodes. Client nodes adopt certificates to authenticate their transactions; peer nodes use certificates to authenticate their transactions (endorsements). The interface, closely associated with the transaction processing component of the system, is intended to enable the defined membership service component to be inserted smoothly in this way without modifying the core of the transaction processing component of the system.

4.3.23 *Membership Management Service*

The Membership Management Service authenticates, authorizes, and manages identities on the permission blockchain. It runs agents of the Membership Management Service in peer nodes and sorting service nodes.

4.3.24 *Sorting Service or Ordering Service*

A collection of nodes that order transactions into a block. The sorting service is independent of the peer node process and sorts transactions on a first-come, first-served basis for all channels on the network. The sorting service upholds pluggable implementations, currently Solo and Kafka by default.

4.3.25 *Organization*

Organizations, also known as “members,” are invited by blockchain service providers to join the blockchain network. An organization joins a network by adding its MSP to the network. An organization’s transaction endpoints are peer nodes, a group of organizations that form a consortium. While all organizations on the web are members, not every organization will become part of the alliance.

4.3.26 *Node*

A network entity maintains the ledger and runs the contract container to perform read and write operations on the ledger. Members owned and maintained the nodes.

4.3.27 *Policy*

Policies are expressions consisting of attributes of digital identities, such as Org.Peer and Org2.Peer, restrict access to resources on the blockchain network. Policies can be defined before bootstrapping an ordering service or creating a channel or can be specified when instantiating the chaincode on the channel.

4.3.28 *Private Data*

Private data is confidential, stored in a private state database at each authorized peer node, logically separate from the channel book data. Access to private data is limited to the organizations defined on the private data set. Unauthorized organizations can only have hashes of private data on the channel book as evidence of transactional data. In addition, to further protect privacy, the hashes of the private data are passed through the ordering service rather than the private data itself thus making the private data confidential to the ordering service nodes.

4.3.29 *Private Data Set*

A private dataset is applied to manage two or more organizations on a channel that wish to maintain confidentiality with other partners on that channel. It describes the subset of organizations on the channel that has the right to store private data. Only those organizations can transact with private data.

4.3.30 *Raft*

Raft, a new feature added in v1.4.1, is implemented based on the Crash Fault Tolerant (CFT) ordering service of the Raft protocol's etcd library. Compared to Kafka-based ordering services, it is easier to set up and manage than of Raft, which allow organizations to contribute nodes to a distributed ordering service.

4.4 *Architecture Details*

Hyperledger is now the most recognized alliance chain implementation in the industry. As its most crucial subproject, Fabric has drawn much attention. The architectural design of Fabric has evolved from incubation to development. We

have already introduced the essential content and functions of Fabric. Next, we will explore Fabric in-depth, analyze its latest overall architecture, and discuss the features and advantages of the new architecture of Fabric by comparing it with previous ones.

4.4.1 Architecture Interpretation

Fabric adopts the modular building concept in design. According to the overall logical architecture shown in Fig. 4.1, Fabric is mainly composed of three service sections: Membership Service, Blockchain service, and Chaincode service. The membership service provides membership registration, identity management, and authentication services, making platform access more secure and facilitating authority management. Blockchain service is in charge of consensus management among nodes, distributed calculation of the ledger, and implementation of P2P network protocol and ledger storage. As the core component of blockchain, it takes on underlying service support for the primary function of blockchain. The chaincode service offers a smart contract execution engine that has deployment and running environment for Fabric's chaincode (smart contract) program. In the logical architecture diagram, you can see the event stream running through the three service components with the technical support for asynchronous communication of each

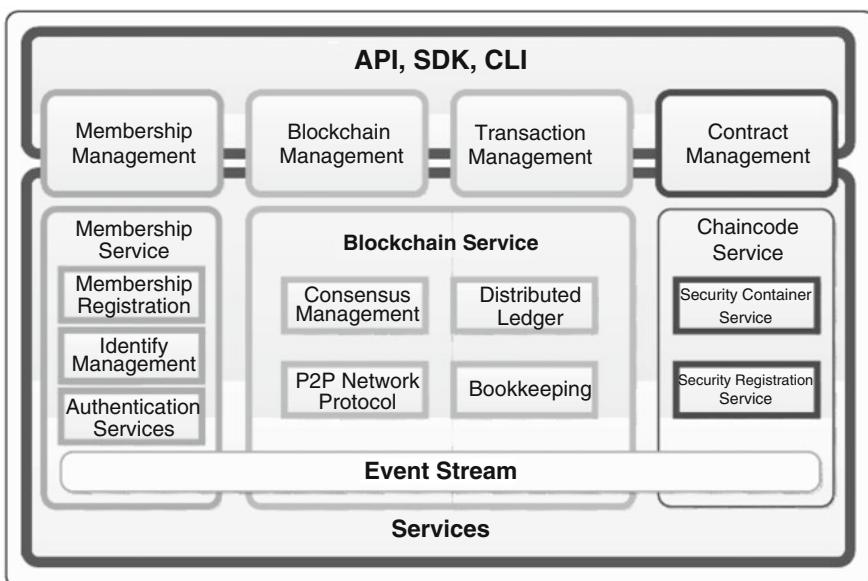


Fig. 4.1 Logical architecture diagram

Fig. 4.2 Runtime architecture (v0.6)

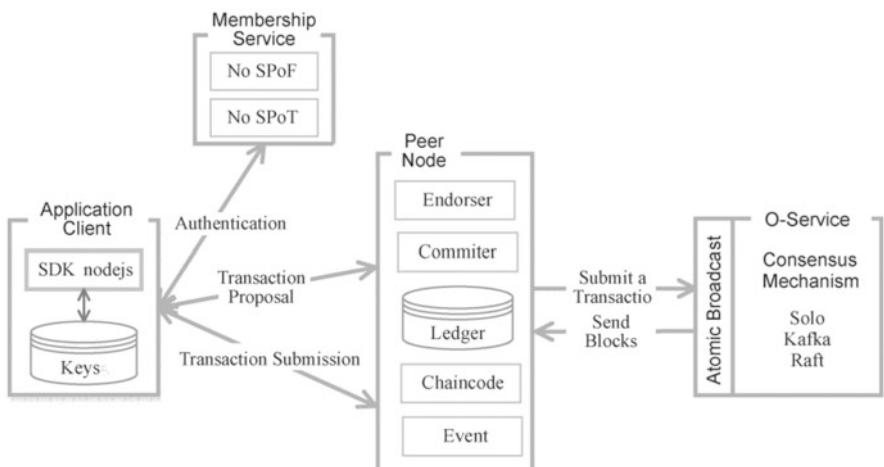
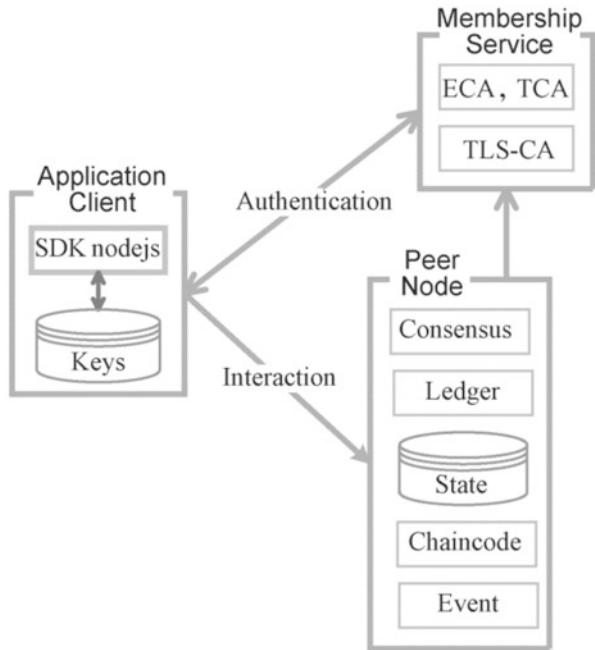


Fig. 4.3 Runtime architecture (v1.4)

element. In the interface part of Fabric, there are three types of interfaces: API, SDK, and CLI, which can be used to manage the operation of Fabric.

Figures 4.2 and 4.3 show the Fabric operating architecture, respectively. The straightforward structure of V0.6, namely, Application-Membership Management-Peer, presents a triangular relationship, and Peer nodes do all business functions of

the system. However, Peer nodes bear too many service functions and expose many scalabilities, maintainability, security, and service isolation problems. Therefore, in v1.4, the official architecture is improved and reconstructed. The ordering service part is completely separated from the Peer node, and a new node is formed independently to provide ordering service and broadcast service. In addition, by introducing the channel concept, the v1.4 realizes multichannel structure and multichain network and brings more flexible business adaptability. It underpins more robust configuration functions and policy management functions to enhance the system's flexibility further.

Compared with v0.6, the new architecture makes the system much better in many aspects, mainly the following advantages.

- **Chaincode Trust Flexibility.** The v1.4 architecturally separates the trust assumptions of contract code from the trust assumptions of ordering services. The new version of the ordering service can be provided by a single set of nodes (orderers), even allowing for some failed or malicious nodes. As for the chaincode, the program can specify different endorsement nodes, which significantly enhances its flexibility.
- **Scalability.** Under the new architecture, the endorsement node for the specified chaincode has an orthogonal relationship with the consensus node. Compared with v0.6 architecture, all business functions are performed on the Peer node, the scalability of v1.4 architecture has been dramatically improved. In particular, when the endorsing nodes specified by different chaincode do not exist intersection, the system can perform endorsement operations of multiple chaincode programs simultaneously, which is good to improve the system processing efficiency.
- **Confidentiality.** Multichannel is designed to make it easy to deploy chaincode that requires confidentiality for content and execution status updates. Support for private data has been added, and Zero-Knowledge Proof (ZKP) is being developed for future availability.
- **Consensus Modularity.** The v1.4 architecture separates the ordering service from the Peer node to be a consensus node alone. The ordering service is designed as a pluggable modular component, enabling various implementations of consensus algorithms to be applied to complex and diverse business scenarios.

4.4.2 *Membership Services*

Membership services can provide identity management, privacy, confidentiality, and authentication services for Fabric participants on the network. The following section attaches importance to the PKI system and the registration process of users.

PKI System

The goal of PKI (public key infrastructure) is to enable different members to communicate securely without interacting with each other, and the model adopted by Fabric is established on certificates issued by trusted third-party authorities, i.e., certification authorities (CA). CA will issue the certificate after confirming the applicant's identity and provide the latest revocation information of the certificate issued by CA online so that users can verify whether the certificate is still valid. A certificate is a file containing the public key, information about the applicant, and a digital signature. The digital signature ensures that the certificate's contents cannot be tampered with by an attacker, and the verification algorithm can detect any forged digital signature. In this way, the public key and identity are bound together and cannot be tampered with or developed, and membership management can be achieved.

Membership services combine a PKI architecture with a decentralized consensus protocol to transform an unlicensed blockchain into a licensed blockchain. In a permission-free blockchain, entities are not necessarily to be authorized, and all nodes in the network do not have different roles. They are all unified peer entities with equal rights to submit transactions and keep accounts. In a licensed blockchain, the entities obtain a long-term certificate of identity (such as a certificate of registration) that can be differentiated by entity type by registration. In terms of users, the Transaction Certificate Authority (TCA) issues an anonymous certificate to the user upon registration. On the other hand, the submitted transaction is authenticated by a transaction certificate, permanently stored on the blockchain for the authentication service to trace the transaction. The membership service is a certification center for providing users with certificate authentication and permission management functions to manage and authenticate nodes and transactions in the blockchain network.

In the Fabric system, membership services consist of several basic entities that collaborate to manage the identity and privacy of users on the network. Some of them verify the users' identity, some register their identity in the system, and others provide the user with the required certificate credentials when entering the network or invoking transactions. PKI is a framework for public key encryption, ensuring the security of data exchange on the web and confirming and managing the other party's identity. PKI is also employed to manage the generation, distribution, and revocation of keys and digital certificates in Fabric systems.

Typically, a PKI system consists of a certificate issuing authority (CA), a registration authority (RA), a certificate database, and a certificate store entity. Among other things, the RA is a trusted entity that authenticates users and validates data, certificates, or other materials used to support user requests and create registration credentials needed for registration. The CA issues digital certificates to specified users according to RA's suggestions. These certificates are authenticated directly or hierarchically by the root CA. The detailed entities of the membership services are shown in Fig. 4.4.

The following is a detailed description of the entities in the figure.

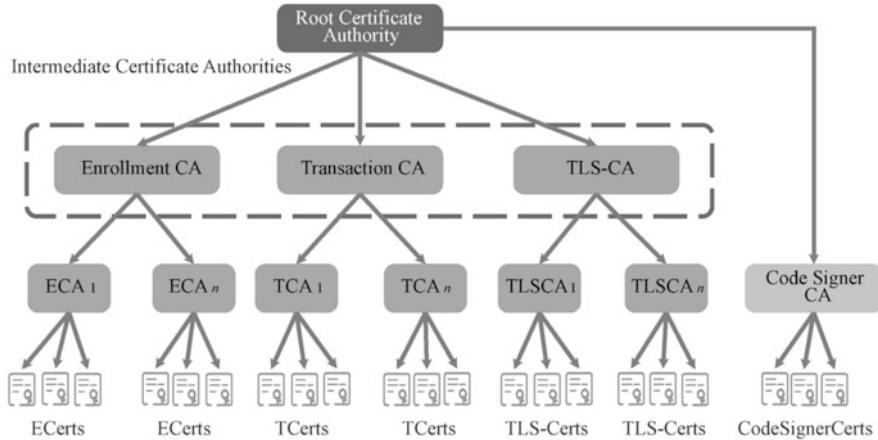
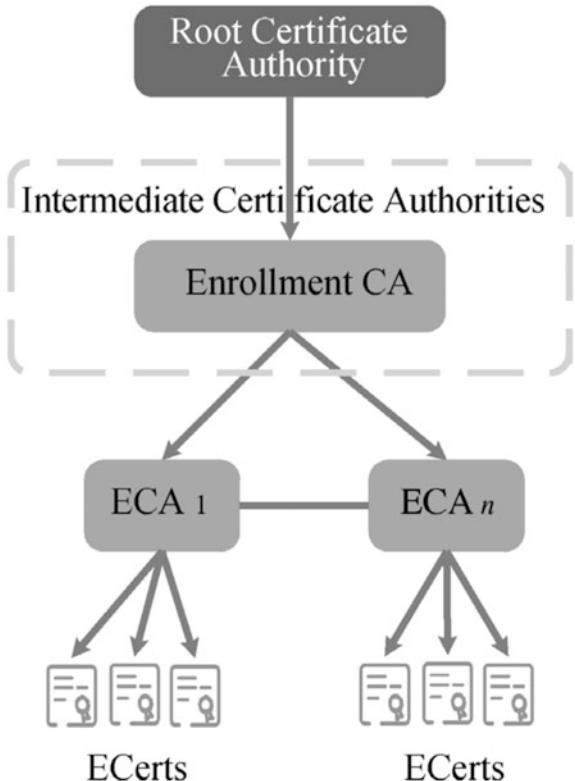


Fig. 4.4 Member service entity composition

- **Root Certificate Authority:** The Root CA represents the trusted entity in the PKI architecture and is the top-level CA in the PKI architecture.
- **Enrollment CA (ECA):** After verifying the Enrollment credentials provided by the user, the ECA issues Enrollment Certificates (ECerts).
- **Transaction CA (TCA):** After confirming the registration credentials the user provided, the TCA issues Transaction Certificates (TCerts).
- **TLS CA:** TLS CA issues Transport Layer Security (TLS) certificates and credentials to allow users to adopt the network.
- **ECerts (enrollment certificates):** ECerts are long-term certificates issued for all roles.
- **TCerts (transaction certificates):** TCerts are short-term certificates per transaction issued by the TCA based on authorized users' requests. They can configure TCerts to participate in the system anonymously without identifying the users and prevent transaction linkability..
- **TLS-Certs (TLS-Certificates):** It carries the owner's identity, acts as a communicator between systems and components, and maintains network-level security.
- **CodeSignerCerts (Code Signer Certificates):** It digitally signs software code, identifies the source of software and the real identity of the software developer, and ensures that the code cannot be tampered with after being signed.

The financial IC card system applied the PKI system with its architecture shown in Fig. 4.5. In contrast to Fabric's PKI system, there is no TCert, and ECert does every transaction, so transactions in this system are not anonymous.

Fig. 4.5 Financial IC card PKI architecture



User/Client Registration Process

The previous section introduced the entities of the PKI system for membership services and their essential functions, and the next part will briefly explore the specific user registration process. Figure 4.6 shows a high-level description of the user registration process, divided into two phases: the offline method and the online process.

Out-of-Band Process

1. Each user or Peer node must provide identification documents (proof of ID) to the RA registry, while this process must be transmitted via out-of-band data (out-of-band, OOB) to provide the evidence required by the RA to establish (and store) an account for the user.
2. The RA registrar returns the user's relevant username and password, as well as the trust anchor (containing the TLS CA Cert). If the user has access to a local client, the client can use the TLS CA Cert as a trust anchor.

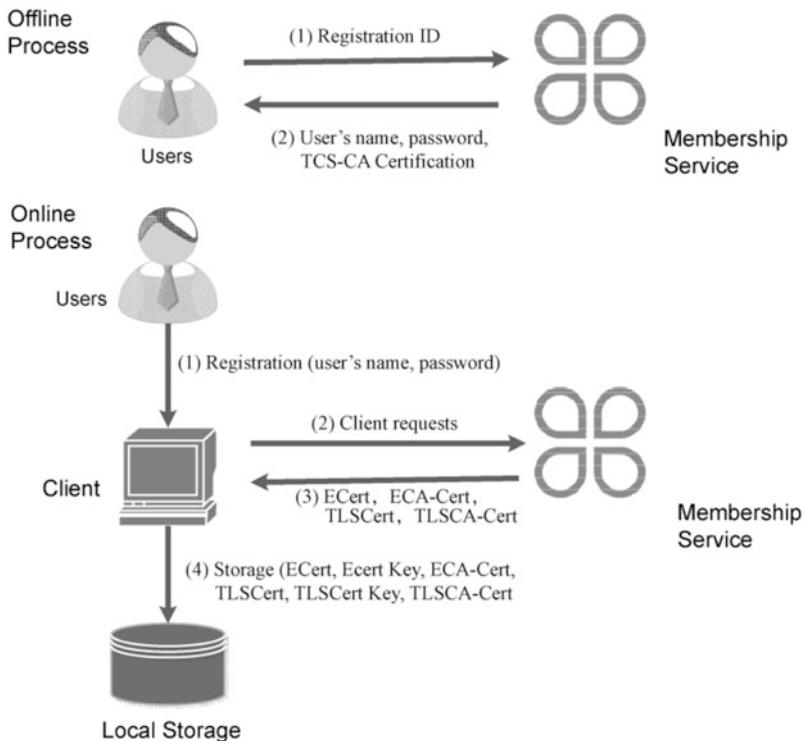


Fig. 4.6 User registration process

Online Process

1. The user connects to the client to request a login to the system, in which the user sends the user name and password to the client.
2. The client sends a request to the member service on behalf of the user, and the membership service accepts the request.
3. The membership service sends a package containing several credentials to the client.
4. Once the client verifies that all encrypted materials are correct and valid, it stores the certificate in the local database. It notifies the user, and thus the user registration is complete.

4.4.3 *Blockchain Services*

Blockchain services consist of four modules: consensus management, distributed ledger, ledger storage, and P2P network protocol. Consensus management is to make messages reach consensus in a distributed complex network of multiple nodes. Distributed ledgers and ledger stores are applied for all data storage in the blockchain system, such as transaction information, world state, and private data. P2P network protocol, the communication mode of nodes in the network, connects the communication and interaction between nodes in the Fabric.

The P2P Network

The distributed application architecture of a P2P network is a networking or network form formed by a peer-to-peer computing model at the application layer, allocating tasks and workloads among peer entities. In a P2P network environment, multiple computers connected are peers and have the same functions. A computer can serve as a server, setting up shared resources for other computers in the network and as a workstation to request services. In general, the entire network does not rely on dedicated centralized servers or dedicated workstations. However, in the distributed environment of blockchain, each node should be equal, naturally suitable for P2P network protocol.

In Fabric's network environment, nodes are the communication entities of the blockchain. There are three types of nodes, namely Client Node (Client), Peer Node (Peer), and Ordering Service Node (Ordering Service Node or Orderer).

The client node represents the end-user entity. It must be connected to a Peer node before it can communicate and interact with the blockchain. The client node can connect to any Peer node of its own choice to create transactions and invoke transactions simultaneously. In the actual system operating environment, the client should communicate with Peer nodes to submit virtual transaction invocations and communicate with ordering services to request broadcast transactions.

The Peer node communicates with the ordering service node to maintain and update the world state. They receive messages broadcast by the ordering service, sort transaction information in blocks, and edit and support their local world state and ledger. At the same time, the Peer node can play the role of endorsement node for endorsing transactions. The unique function of the endorsing node is set for a particular transaction and approved before it is submitted. Each contract code program can specify an endorsement policy that contains multiple sets of endorsement nodes. This policy will define the necessary and sufficient conditions for a valid transaction endorsement (usually a collection of endorsements node signatures). Note that there is a special case where the endorsement policy is specified by the endorsement policy of the system contract code, not by itself, in a deployment transaction in which a new contract code is installed.

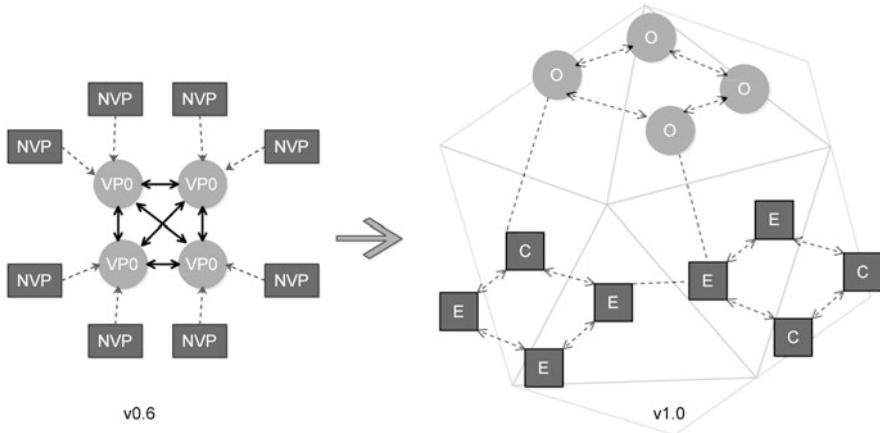


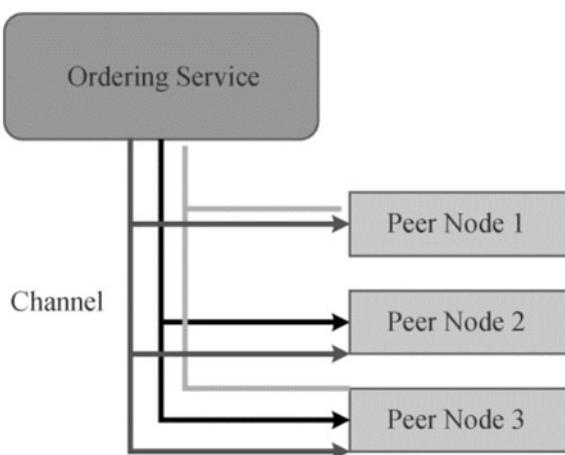
Fig. 4.7 Network topology

The Orderer is a component of the ordering service. The ordering service can be treated as a communication organization that offers delivery guarantees. By sorting the transactions, the ordering service node undertakes the responsibility of ensuring that all the final transactions output in the same sequence and provides broadcast communication service for the delivery assurance service. We will also describe more about the ordering service later.

In addition to the types of nodes, let us take a look at the network's topology. In v0.6, the entire network consists of VP (validating Peer) validating nodes and NVP non-validating nodes. As shown in Fig. 4.7, the network contains four validating nodes, each of which also connects to two non-validating nodes, while the whole network consists of four validating nodes. In v1.4, the network topology has changed significantly as the network node types change, in which the ordering service nodes together form the ordering service and then abstract the service. Meanwhile, the Peer nodes can be divided into endorsing nodes or submitting Peer nodes, and they can be grouped. Then the whole ordering service, together with the group formed by the Peer nodes, constitute the new complete network.

At the same time, Fabric introduced the concept of channels after v1.0. Messaging over ordering services supports multiple channels, allowing Peer nodes to subscribe to any number of channels on applying access control policies. A subset of Peer nodes can be specified by the application to set up a related channel. Peer nodes with the same channel can form a set and submit the transaction of the channel. Only these Peer nodes can receive the relevant transaction block and are completely isolated from other transactions. Fabric supports multichain and multichannel, that is, multiple channels and chains exist in a system, as displayed in Fig. 4.8. Depending on the business logic, the application sends each transaction to one or more designated channels without any connection among transactions on different channels.

Fig. 4.8 Multichannel structure



Starting with v1.2, Fabric can set up private data sets in ledgers, allowing subsets of organizations to acknowledge, submit, or query private data, enabling the ability for data from one group of organizations on a channel to be kept secret from another organization without building a separate channel. The actual private data is stored in a private state database (sometimes called a side database or SideDB) on the peer nodes of the authorized organization and can be accessed by chaincode on the authorized nodes through the Gossip protocol. Ordering services are not involved and cannot see private data. Since the Gossip protocol distributes private data peer-to-peer across authorized organizations, this requires the creation of anchor nodes in the channel and the configuration of CORE_PEER_GOSSIP_EXTERNALENDPOINT on each node in order to direct cross-organizational communication. The hashes of the private data can be recognized, sorted, and written to the ledger of each peer on the channel and can be used as evidence of transactions for status validation and auditing purposes.

Overall, Fabric's refactorings and new features in nodes and networks have resulted in its enhanced transaction processing capabilities and sound privacy isolation.

Ordering Service

The Orderer nodes in the network come together to form ordering service. It can be seen as a communication organization that provides delivery guarantees. The ordering service offers a shared communication channel for clients and Peer nodes, and the function of a broadcast service for messages containing transactions. As a client connects to the channel, it broadcasts messages to all Peer nodes through the ordering service. The ordering service can provide an atomic delivery guarantee for all messages, that is, the ordering service in Fabric ensures that message communication is serialized and reliable. In other words, the ordering service

outputs the same message to all Peer nodes connected to the channel, and the logical order of the output is the same.

Ordering service can be implemented in different ways. In v1.4, Fabric designs ordering services as pluggable modules configured with varying options for different application scenarios. Currently, Fabric offers three mode implementations: Solo, Kafka, and Raft.

Solo is a simple timing service deployed on a single node, mainly for development testing. It only supports single chains and single channels. Kafka is a cluster ordering service that supports multichannel partitioning and CFT (crash faults tolerance). It tolerates partial node downtime failure but not malicious nodes. It is basically implemented on the Zookeeper service. The total number of nodes and failed nodes must satisfy $n \geq 2f + 1$ in the distributed environment used. Raft follows the “leader and follower” model, where each channel elects a “leader,” and its decisions are replicated to the “followers,” supporting CFT. As long as the total number of nodes and the number of failed nodes satisfy $n \geq 2f + 1$, it allows some nodes, including the leader, to go down and fall. It should be easier to set up and manage the Raft than Kafka-based ordering services. Both are designed to allow organizations to contribute nodes to decentralized ordering services.

Distributed Ledger

Blockchain technology can be considered as a shared ledger technology when analyzed in terms of its underlying construction. The ledger is the core component of the blockchain, in which all historical transactions and state change records are stored. In Fabric, each channel corresponds to a shared ledger. Each Peer node connected to the shared ledger is qualified to be in the network and view the ledger information, i.e., it allows all nodes in the network to take part and view the ledger information. The information on the ledger is publicly shared, and a copy of the ledger is maintained on each Peer node. Figure 4.9 illustrates the structure of the Fabric ledger.

As you can see from the figure above, the shared ledger is stored locally as a file system. The shared ledger consists of the Chain part in the diagram and the State part on the right where the State data is stored. The Chain part stores all transaction information, which can be queried only but cannot be deleted or modified. The State section holds the latest values for all variables in the transaction log and is sometimes called the “world state” because it represents the latest values for all variable key-value pairs in the channel.

Chaincode calls execution transactions to change the current state data. The design stores the latest key-value pair data in the state database to facilitate the chaincode to interact efficiently. The default state database is Level DB but can be switched to Couch DB or other configurations.

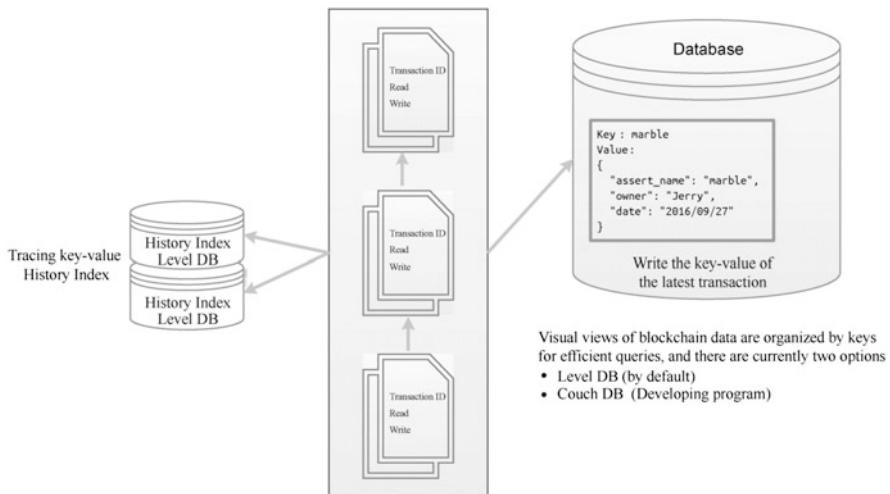


Fig. 4.9 Shared ledger structure

4.4.4 Chaincode Services

The chaincode service provides a secured and lightweight method to sandbox chaincode execution, secure container services, and secure chaincode registration services. The runtime environment that is “locked down” is a secure container, where the chaincode is first compiled into a standalone application that runs in an isolated Docker container. When the chaincode is deployed, a set of signed Docker base images of the smart contract is automatically generated. The chaincode interacts with the Peer node in the Docker container, as shown in Fig. 4.10.

The steps are as follows:

1. The Peer node receives a request for chaincode execution from the client, interacts with the chaincode through gRPC, and sends a message object to the corresponding chaincode.
2. The chaincode executes the GetState() operation and PutState() operation by calling the Invoke() method to get the ledger state database and send the ledger pre-submitted state number to the Peer node. The GetPrivateDate() and PutPrivateDate() methods will be used once it is necessary to read and write private data.
3. After the chaincode is executed successfully, the output result is sent to the Peer node. The endorsing node endorses and signs the input and output and answers to the client after completion.

We will introduce the specific analysis and writing of chaincode in detail in the next section.

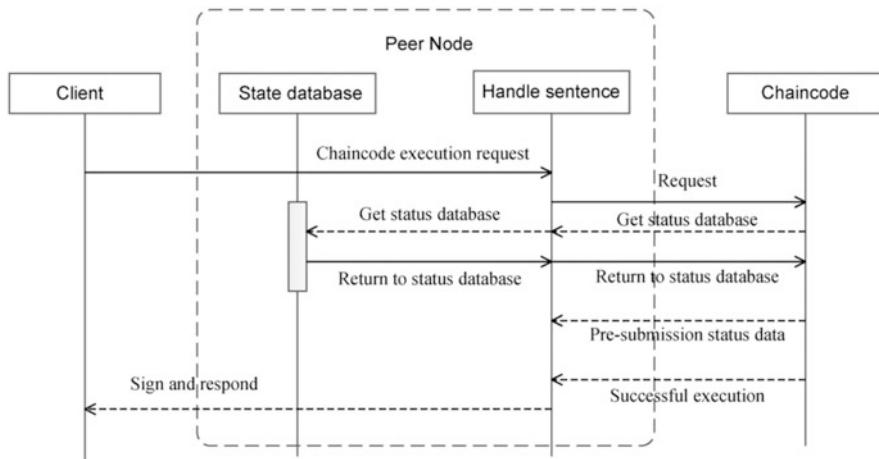


Fig. 4.10 Interaction between the contract code and peer node

4.5 Chaincode Analysis

Based on the above architectural interpretation, it can be seen that the chaincode service is a core part of the Fabric architecture. This section studies the chaincode running in the chaincode service and describes how to write, deploy, and invoke the specific code.

4.5.1 Chaincode Overview

Chaincode, a piece of code that runs on the blockchain, is how smart contracts are implemented in Fabric. In Fabric, the chaincode is the sole source of transaction generation. The shared ledger is a growing chain of hashes connected by blocks. The blocks contain all the transaction information represented in a Merkle tree data structure; therefore, transactions are the most basic physical unit on the blockchain. How are transactions generated? Transactions can only be generated through chaincode invocation operations, and, thus, the core component of Fabric and the only channel to interact with the shared ledger.

The Fabric now supports writing chaincode by implementing the Java, Go, and Node.js languages. According to Fabric's design, the Shim package located in the/core/chaincode directory is an SDK offering chaincode development. It can, in theory, be used independently but may not be well isolated at this time due to the requirement to call some other dependency modules.

The chaincode in the Fabric runs on the Peer node, and operations related to the chaincode such as deployment, installation, and invocation are performed on the Peer node. The chaincode is installed and initialized on Peer nodes of the Fabric

network using the SDK or CLI to enable the user to interact with the shared ledger of the Fabric network. At present, chaincode runs in two-node modes: general mode and development mode. The available model is the system default mode, and the chaincode runs in the Docker container. The Docker container runs the Fabric system, which provides an isolated environment for running the Fabric system and chaincode to improve the entire system's security. However, under the model, the development and the debugging process is very complicated and troublesome for developers because the Docker container calls for restarting after each code modification, significantly reducing program development efficiency. As a result, Fabric supplies an alternative mode of operation, development mode, for the sake of developer efficiency. Chaincode no longer runs in Docker container in the development phase but is directly deployed, operated, and debugged locally, greatly simplifying the development process.

4.5.2 *Chaincode Structure*

The chaincode is one of the essential parts of Fabric development. It enables the interaction and operation of entities such as ledgers and transactions and the implementation of various business logic. The chaincode at current stage supports programming in Go, Java, and Node.js languages by implementing the chaincode interface. The following introduces the Go language as an example.

The structure of the chaincode mainly includes the following three aspects.

Chaincode Interface

In Fabric v1.4, the chaincode interface contains two methods: Init() method and Invoke() method. Init() method will be called when the chaincode is deployed for the first time, which is somewhat similar to the constructor method in a class. The Invoke() method works when the chaincode method is called to perform some actual operation, and each call is treated as transaction execution. The detailed transaction flow is described in Sect. 4.6. The chaincode interface code in Go is shown below:

```
Type Chaincode interface {
    // Initialization work, normally called only once
    Init(stub ChaincodeStubInterface) pb.Response
    // Query or update world state, can be called multiple times
    Invoke(stub ChaincodeStubInterface) pb.Response
}
```

API Methods

When the Init or Invoke interface is called, Fabric passes the shim. ChaincodeStubInterface parameters and returns the pb.Response, which manipulates the ledger service, generates transaction information or calls other chaincode by invoking the API methods.

The API methods are defined in the shim package in the/core/chaincode directory and can be generated by the following command.

```
godoc github.com/hyperledger/fabric/core/chaincode/shim
```

The API methods can be mainly divided into six categories, namely State read/write operations, Args read/write operations, Transaction read/write operations, PrivateData read/write operations, Chaincode inter-calls, and Event settings. Table 4.2 shows these methods and their corresponding functions.

Chaincode Return Information

The chaincode data model is returned in the form of a protobuf, defined as shown below.

```
message Response {
    // Status Code
    int32 status = 1;
    // Response code information
    string message = 2;
    // Response content
    bytes payload = 3;
}
```

The chaincode also returns event messages, including Message events and Chaincode events, as defined below.

```
messageEvent {
    oneof Event {
        Register register = 1;
        Block block = 2;
        ChaincodeEvent chaincodeEvent = 3;
        Rejection rejection = 4;
        Unregister unregister = 5;
    }
}
messageChaincodeEvent {
    string chaincodeID = 1;
    string txID = 2;
    string eventName = 3;
```

Table 4.2 API methods and their corresponding functions

API methods	Functions
GetState(key string)	Get the latest state
PutState(key string, value []byte)	Add state
DelState(key string)	Delete state
GetStateByRange(startKey, endKey string)	Get state
GetStateByPartialCompositeKey(objectType string, keys []string)	Get state
GetStateValidationParameter(key string)	Get state validation parameters
GetStateByRangeWithPagination(startKey,endKey string, pageSize int32,bookmark string)	Paging to get state
GetStateByPartialCompositeKeyWithPagination(objectType string, keys []string,pageSize int32, bookmark string)	Paging to get state
GetQueryResult(query string)	Query results
GetHistoryForKey(key string)	Seach history
CreateCompositeKey(objectType string, attributes []string)	Create a composite key
SplitCompositeKey(compositeKey string)	Spliting composite keys
GetArgs()	Read parameters
GetStringArgs()	Read parameter string
GetFuncGonAndParameters()	Read functions and parameters
GetArgsSlice()	Read parameter slices
GetCreator()	Read creators
GetTransient()	Read transaction information
GetBinding()	Read bind
GetTxTimestamp()	Read timestamp
GetTxID()	Read transaction ID
InvokeChaincode(chaincodeName string, args [][]byte, channel string)	Invoke contract code
SetEvent(name string, payload []byte)	Set event flow
GetChannelID()	Get the channel ID
SetStateValidationParameter(key string, ep []byte)	Set status verification parameters
GetQueryResultWithPagination(query string, pageSize int32, Bookmark string)	Paging to get query results
GetPrivateData(collection, key string)	Get private data
PutPrivateData(collection string, key string, value []byte)	Add private data
DelPrivateData(collection, key string)	Delete private data
SetPrivateDataValidationParameter(collection, key string, ep []byte)	Set private data validation parameters
GetPrivateDataValidationParameter(collection, key string)	Get private data validation parameters
GetPrivateDataByRange(collection, startKey, endKey string)	Get private data
GetPrivateDataByPartialCompositeKey(collection, objectType string, keys []string)	Get private data

(continued)

Table 4.2 (continued)

API methods	Functions
GetPrivateDataQueryResult(collection, query string)	Private data query results
GetDecorations()	Obtain additional data
GetSignedProposal()	Get signature identity information
NewLogger(name string) *ChaincodeLogger	Create a new log
Start(cc Chaincode)	Start contract code
SetLoggingLevel(level LoggingLevel)	Set log level

```
    bytes payload = 4 ;
}
```

Once the development of the chaincode is complete, there are two ways to interact: through the SDK or the CLI command line. Interaction via the CLI command line is described in the next section, and interaction with the SDK can be found in the examples in Chap. 5.

4.5.3 *CLI Command Line Calls*

Once the chaincode has been written, it is important to understand how to deploy it and how to invoke it. To deploy the chaincode and other related operations, it is necessary to start the Fabric system. Fabric provides a CLI interface to support Peer node-related operations from the command line. Fabric supports Peer node start-stop operations, various operations associated with contract code, and channel-related operations through the CLI interface.

The CLI commands supported by Fabric are shown in Table 4.3.

Among them, logging getlevel, logging setlevel, and logging revertlevels are not recommended and will be removed in the subsequent version.

Meanwhile, you can check more information about peer commands by following commands.

```
# This command requires
cd /opt/gopath/src/github.com/hyperledger/fabric
build /bin/peer

# Or go to the cli container after entering the boot network
docker exec -it cli bash
# Enter the cli container and run the peer command
peer
```

After running the above command, you should see the information shown in Fig. 4.11.

Table 4.3 CLI commands

Command line parameters	Function	Command line parameters	Function
Version	View version information	Chaincode package	Contract code package
Node start	Start the node	Chaincode query	Contract code inquiry
Node status	View node status	Chaincode signpackage	Contract code signature
Logging getlevel	Get log levels	Chaincode upgrade	Contract code update
Logging setlevel	Get log levels	Chaincode list	Contract code list
Logging getlogspec	Get log specification	Channel create	Create a channel
Logging setlogspec	Set log specification	Channel fetch	Get channels
Logging revertlevels	Recovery level	Channel join	Join channels
Help	Help	Channel list	Channel list
Chaincode install	Contract code installation	Channel update	Channel update
Chaincode instantiate	Contract code instantiation	Channel signconfigtx	Sign the channel configuration
Chaincode invoke	Contract code call	Channel getinfo	Get channel information

```

Usage:
  peer [command]

Available Commands:
  chaincode  Operate a chaincode: install|instantiate|invoke|package|query|signpackage|upgrade|list.
  channel    Operate a channel: create|fetch|join|list|update|signconfigtx|getinfo.
  help       Help about any command.
  logging   Logging configuration: getlevel|setlevel|getlogspec|setlogspec|revertlevels.
  node      Operate a peer node: start|status.
  version   Print fabric peer version.

Flags:
  -h, --help   help for peer

Use "peer [command] --help" for more information about a command.

```

Fig. 4.11 Peer command line parameters

For more detailed command information, please operate the `peer [command] -help` command as shown in the figure above.

4.5.4 Chaincode Execution Swimlane Diagram

The chaincode execution process is described below in Fig. 4.12.

- The client (SDK/CLI) creates a transaction proposal containing chaincode functions and invocation parameters and sends it to the endorsing node in proto message format.
- The endorsing node invokes the methods of shim package to create chaincode to emulate the content of transaction execution.

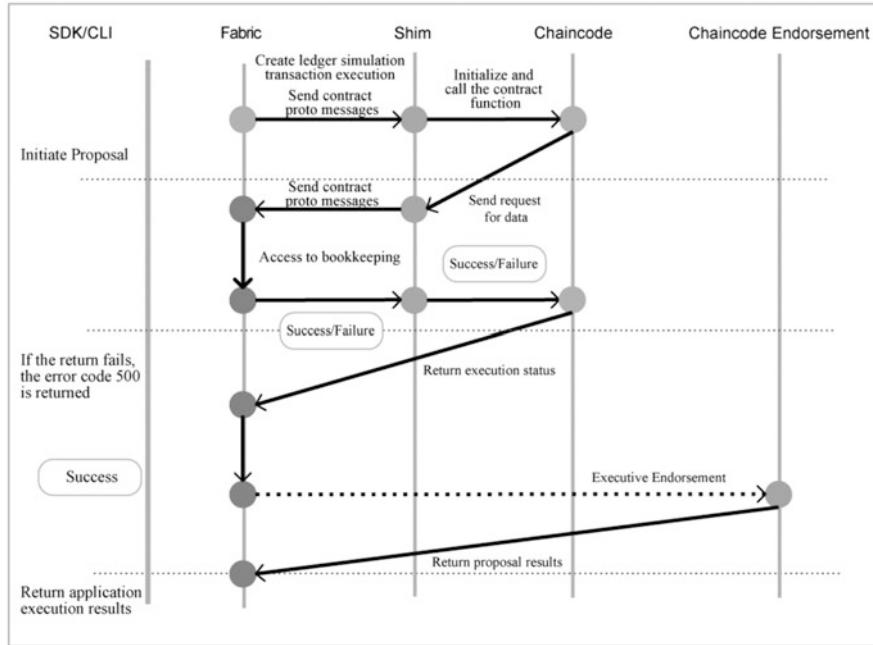


Fig. 4.12 Flow chart of contract code execution

- Endorsing nodes initialize the contract, invoke parameters, and produce read and write operation sets based on the read and write keys.
- Endorsement cluster nodes simulate proposal execution: execute read operation, send a request to the ledger to query the status database; simulate write operation, get the value version number of the key, and simulate updating the status data.
- For a successful return execution, the endorsement operation will be executed; if the return fails, the error code 500 will be sent.
- The endorsing node performs a signature on the transaction result and returns the proposal result to the client (SDK/CLI). The proposal result includes the execution return value, the transaction result, the signature of the endorsing node, and the endorsement result (agree or reject).

For more information on writing contract code, check out the sample contract code under project/examples/chaincode/to learn more.

4.6 Transaction Flow

This section mainly analyzes the transaction endorsement process in Fabric by translating the architecture into the official Hyperledger Fabric document. Firstly, it introduces the mechanism of the transaction endorsement process in Fabric, then

describes its general flow through a simple case. Afterward, it analyzes the endorsement process in detail and briefly displays Fabric’s endorsement strategy and its application in the verification ledger and PeerLedger checkpoint.

4.6.1 General Process

In a Fabric system, a transaction is to call a chaincode, which has one of two types below:

- **Deploy transaction:** It adopts a program as a parameter to build a new chaincode installed on the blockchain after successfully executing the deploy transaction.
- **Call transaction.** A call transaction executes the chaincode and the functions it provides in the context of a previously deployed transaction. When the call transaction is completed, the chaincode performs the specified operation, possibly modifying the state of the corresponding ledger, and returns the output.

Transactions executed in the blockchain are packaged into blocks linked together to form a hash chain in a shared ledger. This section describes the process of completing a transaction in the Fabric system. To better understand the transaction endorsement process for the Fabric system, this section begins with a simple graphical example to show a successful transaction execution.

First of all, in this illustrative case, you have to make some assumptions about the configuration that needs to do in actual development. Let us assume the following.

- **Node type:** E0, E1, E2, E3, E4, and E5 are all Peer nodes. E0, E1, and E2 are the endorsement nodes of this transaction, and the Ordering Service is an ordering service composed of orderers.
- **Channel configuration:** There are two channels in this case. E0, E1, E2, and E3 are connected to the same channel, and E4 and E5 are connected to another Channel2.
- **Endorsement policies:** E0 and E1 must be signed and endorsed. E2, E3, E4, and E5 are not part of the strategies.

After making assumptions, start the case flow, as shown in Fig. 4.13.

1. The client application sends a transaction proposal to the endorsement node E0 via the SDK, which is employed to receive requests from the relevant function in the smart contract and then updates the book data (i.e., the key/value of the asset). The client also packages this transaction proposal into a recognizable format (e.g., a protocol buffer on gRPC) before sending and signs the transaction proposal using the user’s cryptographic credentials.
2. After receiving the transaction proposal from the client, the endorsing node E0 will first verify whether the client’s signature is correct and then simulate the execution of the parameters of the transaction proposal as input. The execution operation will produce a transaction result with the execution return value, and

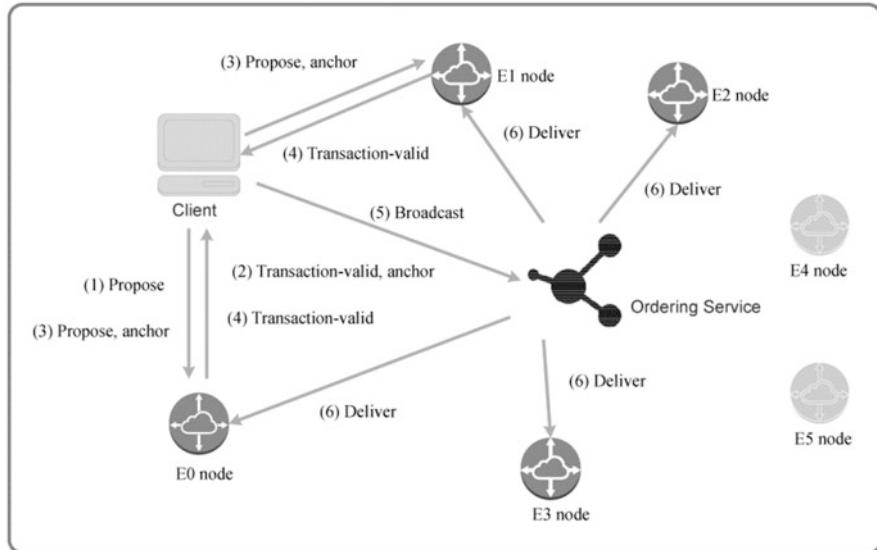


Fig. 4.13 General flow of transaction endorsement

the set of read and the write operations (the ledger will not be updated at this time). Then it will perform the endorsement operation on this transaction proposal, attach the anchor information, and send it back to the client. Suppose the data in this proposal has private data. In this case, the endorsing node E0 will first store the private data in the local temporary database and then propagate it to other authorized peer nodes through the Gossip protocol. Only when reaching a specific number can the endorsing node return the transaction result to the client, which does not carry private data, but only the hash value of the confidential data key-value pairs.

3. The client wants to be further endorsed by E1, so it will send a transaction offer to E1 and decide whether to attach the anchor information obtained from E0.
4. The endorsing node E1 validates the client signature in the same way as E0 did earlier, simulates the execution after the validation, and sends the validated transaction-valid information back to the client.
5. The client waits until enough endorsement information has been collected and then passes the transaction proposal and result to the ordering service in the form of a broadcast. The transaction includes the readset, the signature of the endorsing node, the channel ID, and the hash of the private data. The ordering service does not read the transaction details but sorts the received transaction information by channel classification and packages it to generate blocks, so the ordering service cannot see the private data.
6. The ordering service will package the agreed transaction into a block and send it to all nodes connected on the channel. E4 and E5 cannot receive any message because they are not associated with the medium of the current transaction.

7. Each node verifies the received block, whether it satisfies the endorsement strategy and whether the status value on the ledger changes to make a judgment of the validity of the transaction. The successful verification will facilitate the ledger and world state to be updated. The node notifies the client via an event mechanism whether the transaction has been added to the blockchain and whether the transaction is valid. If the block contains hash values for private data, the private data from the temporary database is stored in the private database after successful validation.

4.6.2 Process Details

In Fabric, a transaction refers to a chaincode call. The process of endorsing a transaction will be analyzed in detail below.

The Client Sends the Transaction Proposal to the Specified Endorsement Node

To invoke a transaction, the client sends a PROPOSE message to a set of endorsing nodes of its choice (these messages may not be sent simultaneously, as in the example in the previous section). To select the location of endorsement nodes, the client can use the set of endorsement nodes with a given chaincodeID through Peer, and in turn, obtain the set of endorsement nodes through the endorsement policy. For example, the transaction is sent by the client to all relevant endorsement nodes via chaincodeID. In addition, some endorsement nodes may be offline or opposed, so there are endorsement nodes that do not sign the transaction. The submitting client tries to meet the endorsement policy with valid endorsement nodes.

This section will first introduce the PROPOSE message format and then describe the possible interaction patterns between the submitting client and the endorsing node.

PROPOSE Message Format

A PROPOSE has the format $\text{PROPOSE} = \langle \text{PROPOSE}, \text{tx}, [\text{anchor}] \rangle$ and contains two parameters; the tx transaction message field is required, and the anchor is an optional parameter. Both are analyzed in detail below.

The tx parameter contains various information related to the transaction, with the following field format.

```
tx=<clientID, chaincodeID, txPayload, timestamp, clientSig>
```

- clientID: ID of the submitting client,
- chaincodeID: Call the contract codeID,
- txPayload: the carrier containing the transaction information,
- timestamp: timestamp,
- clientSig: client signature.

As for the txPayload field, the details of the invoke and deployment transactions will be somewhat different.

If the current transaction is a call transaction, txPayload contains only two fields.

```
txPayload = <operation, metadata>
```

- operation: refers to the function and parameters called by the contract code,
- metadata: refers to the other attributes associated with this call.

If the current transaction is a deployment transaction, txPayload will contain a policies field.

```
txPayload = <source, metadata, policies>
```

- source: refers to the source code of the contract code,
- metadata: the other attributes associated with this call,
- policies: the policies associated with the contract code, e.g., endorsement policies.

The anchor parameter includes the readset (a collection of versioning dependency key-value pairs read from the original ledger), the versioning dependency in the world state. If the PROPOSE message sent by the client carries the anchor parameter, the endorsing node also needs to verify whether the anchor parameter contains a local match.

Furthermore, all nodes will take the cryptographic hash tid of the tx field as an identifier for the transaction ($\text{tid}=\text{HASH(tx)}$), and the client will store it in memory until the endorsing node responds.

Message Mode

Since the client's message is sent to a set of endorsing nodes, the client can control the shipped order. For example, the client will first send a PROPOSE message without anchor parameters to a single endorsing node under normal circumstances. Upon receiving it, the endorsing node will process the message and return it to the client with the anchor parameters. Then the client will send the PROPOSE message with the anchor parameter to the rest of the endorsing nodes. In the other mode, the client sends the PROPOSE message without the anchor parameter directly to the set

of endorsing nodes and waits for their return. The client is free to choose the message mode to interact with the backing nodes.

Endorsement Node Simulates Transaction Execution and Generates Endorsement Signature

Since the endorsement node receives the < Withdraw, TX,[Anchor]> message from the client, it will verify the client's signature first and then simulate the transaction's execution after passing the verification. It should be noted that if the client specifies the anchor field, then the value of the corresponding key in the local KVS needs to be verified. Only when it is consistent with the Anchor parameter value will the endorsement node simulate the execution of the transaction.

Stimulation execution will experimentally execute the action in txPayload by calling the chaincode corresponding to the chaincodeID. It will hold a copy of the world state maintained locally by the endorsing node. After execution, the endorsing node updates the set of readset and writeset (which stores status updates) key-value pairs. This mechanism is also known as MVCC+ Postimage information in DB databases. The key-value pair operations are as follows.

Given the state S before the endorsing node executes the transaction, (k,s(k).version) each key k read from the transaction will be embedded to the readset. For each key k modified by the transaction, (k,v') will be embedded to the writeset, where v' is the updated new value. Alternatively, v' differs from the previous value (s(k).value).

After the simulation execution, the Peer node decides whether to endorse this transaction on endorsement logic. By default, the Peer node receives the tranproposal message and signs it. However, the endorsement logic can be set. For example, the Peer node takes tx as input and interacts with the legacy system to decide whether to endorse this transaction.

If it decides to endorse this transaction, it sends a <TRANSACTION-ENDORSED, tid, tran-proposal, epSig> message to the client.

```
tran-proposal := (epID,tid,chaincodeID,txContentBlob,readset,
writeset)
txContentBlob: Transaction Information txPayload
epSig: Signature of endorsing node
```

If it refuses to endorse the transaction, it sends a (TRANSACTION-INVALID, tid, REJECTED).

Note that the endorsing node simulates execution without changing any hash chain or world state information. It just simulates execution and stores the state changes caused by the operation in the writeset.

The Client Collects the Transaction Endorsement and Broadcasts It Through the Ordering Service

Within a certain time interval, if the client receives enough endorsement messages back from the endorsing nodes (TRANSACTION-ENDORSED, tid, *, *), then the endorsement policy is satisfied, and the transaction is considered to be endorsed successfully. It should be noted that it has not been committed at this point. Otherwise, if enough endorsement messages are not received within a specific time interval, the client discards the transaction or retries later.

For a valid endorsed successful transaction, the client will call the ordering service via the broadcast(blob) method, where the blob refers to the endorsement message. If the client cannot call the ordering service directly, it will choose some proxy Peer, yet this Peer node must be trusted; otherwise, this transaction may be considered an invalid endorsement.

The Ordering Service Transmits the Block to the Peer Node

After the ordering service sorts the transaction and reaches a block, the ordering service triggers the Deliver (Seqno, PrevHash, BLOb) event and then broadcasts this block to all Peer nodes linked in the Fabric on the same channel.

The Peer node performs two types of verification after receiving the block broadcast by the ordering service.

The first type verifies the validity of blob.tran-proposal.chaincodeId by the endorsement strategy in the contract code. As soon as the first type of validation completes, the second type will verify the blob.endorsement. tran-proposal. readset set.

Validation regarding readset collections can be done in different ways depending on consistency and isolation guarantees. Serializability is the default validation method if the corresponding endorsement policy is not specified in the contract code. Serializability asks for the version of the key in each readset corresponding to the performance in state, rejecting transactions that do not satisfy the criteria.

If the verification above passes, the transaction can be valid or submitted. After validation, the Peer node marks the transaction with a 1 in the bitmask corresponding to the peerLedger ledger and applies updates in writeset to the Fabric blockchain's state world state. If the authentication fails, the transaction is considered invalid, and the Peer node marks the transaction with 0 in the Bitmask of the Peerledger. The null transaction does not cause any change updates.

With the guarantee of the ordering service, the above process ensures that all normal Peer nodes process the same world state after executing a Deliver event. That is, all the correct nodes will receive an identical sequence of Deliver events. At this point, the transaction process is over.

4.6.3 Endorsement Strategies

The endorsement policy mechanism delivered by the Fabric is to specify the rules for transaction validation of the blockchain nodes. Whenever an endorsing node receives a transaction request, the system verifies the transaction's validity through VSCC (validation system Chaincode). In the transaction flow, a transaction may have one or more endorsements from the endorsing node. The VSCC mechanism will determine the validity of the transaction according to the following rules.

- Whether the number of endorsements meets the requirements;
- Whether the endorsement comes from the expected source;
- Whether all endorsements from the endorsing nodes are valid (i.e., whether they come from valid signatures of valid certificates on the expected messages).

An endorsement policy is to specify the above requirements for the number of endorsements and the expected set of endorsement sources. Each policy consists of two parts, the principle and the threshold gate. The principle P identifies the entities with expected signatures; the threshold gate T has two input parameters, t denotes the number of endorsements and n denotes the list of endorsement nodes, i.e., the condition that t is satisfied and the endorsement nodes belong to n . For example, $T(2, "A", "B", "C")$ means that more than two endorsements from “A,” “B,” and “C” are required. $T(1, "A") \ T(2, "B", "C")$ indicates the need to receive an endorsement from “A” or two endorsements from “B” and “C.”

The syntax for representing endorsement policies in CLI command line interactions is EXPR([E, E...]). EXPR has two options, AND or OR, where AND stands for “with”, indicating each is required, and OR means “or”. For example, AND(“Org1.member,” “Org2.member,” “Org3.member”) ask for the three groups’ signatures, while OR (“Org1.member,” “Org2.member”) asks for the signature of any one of the two groups. Furthermore, OR(“Org1.member,” AND(“Org2.member,” “Org3.member”)) indicates there are two options. The first requests the signature of organization 1 and the second asks for the signatures of organization 2 and organization 3.

When applying the CLI to interact with the blockchain, use the -P option after the command to specify the appropriate endorsement policy for the executed chaincode, for example, the following chaincode deployment command.

```
peer chaincode deploy -C testchainid -n mycc -p $ORDER_CA -c '{"Args": ["init", "a", "100", "b", "200"]}' -P "AND('Org1.member', 'Org2.member')"
```

The mycc that indicates the deployment chaincode requests the signatures of organization 1 and organization 2.

Fabric will further enhance and improve the endorsement strategy in the future. In addition to the current identification principle through the relationship with MSP, Fabric plans to add the form of OU (organization unit) to complete the certificate

function. It intends to improve the syntax of the endorsement strategy by using a more intuitive syntax.

4.6.4 Verifying the Ledger and PeerLedger Checkpoints

The verification ledger is a hash chain derived from a ledger that holds valid and committed things only. In addition to the world state and the ledger, peer nodes maintain verification ledgers.

A VLedger block (vBlock), a block filtered out of invalid transactions, has a dynamic size and can be empty. Since PeerLedger blocks may contain invalid things (i.e., things with weak endorsements or invalid version dependencies), these are filtered out by peer nodes before being added to the vBlock. Each peer node links vBlocks to a hash chain, and each vBlock has the hash of the previous vBlock, the vBlock number, an ordered list of all valid transactions committed by the last peer node that was estimated, the hash of the corresponding block derived from the current vBlock, and the hash of all this information.

The ledger contains invalid transactions which may not necessarily be recorded forever. However, once peer nodes connect to the corresponding vBlock, they cannot simply discard PeerLeger blocks to prune PeerLedger. To facilitate PeerLedger pruning, v1.4 of the Hyperledger Fabric provides a checkpointing mechanism: vBlocks that traverse the peer node network are used, allowing checkpointed vBlocks to replace discarded PeerLedger blocks. Since there is not much reason to store invalid things and establish the validity of individual transactions when replacing PeerLedger's reconstructed state, there is less storage space and workload to rebuild the state for new peer nodes joining the network. Only the state updates included in the verified ledger are likely replaced.

Peer nodes periodically perform checkpoints, where CHK is a configurable parameter. The peer node initiates a checkpoint with the message <CHECKPOINT, blocknohash,blockno,stateHash,peerSig> broadcast to other peer nodes to initiate a checkpoint, where blocknohash is the respective hash, blockno is the current block serial number, stateHash is the latest hash on blockno block verification, and peerSig is the signature of the peer. (CHECKPOINT,blocknohash,blockno,stateHash) refers to the verified ledger.

The peer node collects checkpoint messages until it gets correctly enough signed messages (blockno, blocknohash, and stateHash) to establish a valid checkpoint. If blockno>latestValidCheckpoint.blockno, then the peer node should mark latestValidCheckpoint=(blocknohash,blockno) and store the corresponding set of peer node signatures that constitute a valid checkpoint into the set latestValidCheckpointProof, and the state corresponding to stateHash is stored in the latestValidCheck-pointedState, pruning the PeerLedger larger than the block serial number blockno.

The checkpoint validity policy defines not only when a peer node can prune its Peerledger but also how many CHECKPOINT messages are considered “enough.” Here are two possible approaches, which can also be used in combination.

The first is the Local (peer-specific) Checkpoint Validity Policy (LCVP): the local policy for a given peer p specifies a set of peers trusted by p and whose CHECKPOINT messages are sufficient to establish a valid checkpoint.

The second is the Global Checkpoint Validity Policy (GCVP): the checkpoint validity policy is specified globally, similar to the local peer node policy, but established at the system (blockchain) granularity rather than the peer node granularity.

4.7 Summary

This chapter explains Hyperledger Fabric in-depth to facilitate readers to understand the underlying implementation principles of Fabric. Firstly, it introduces the development status and management model of Hyperledger and its subprojects, and then it focuses on Hyperledger Fabric. Secondly, it analyzes the Hyperledger Fabric architecture in detail, discusses the architectural composition and features of Hyperledger Fabric from three aspects: member service, blockchain service, and chaincode service, and reveals the Fabric architecture design and module components. Thirdly, it draws out the code structure, invocation method, and execution flow of the chaincode. Finally, it carries out a detailed analysis of the transaction endorsement process.

Chapter 5

Application Development: Fundamentals of Hyperledger Fabric



Chapter 4 provides an in-depth explanation of Hyperledger Fabric that allows readers to have a basic understanding of the Fabric's core principles. On top of that, this chapter will mainly explain the best practices of blockchain application development on Fabric, giving the deployment of Fabric environment, chaincode development guide and CLI and SDK application development examples from the perspective of application. It enables readers to develop better applications based on Fabric by combining theory and practice.

5.1 Environment Deployment

Environment deployment is the first step in the development practice. Only when the development environment is successfully set up can we continue the later practice. This section describes the detailed process of setting up the Fabric development environment to support the later practice.

5.1.1 Software Download and Installation

The required softwares to install are Oracle VM VirtualBox, Vagrant, and Git. Let us introduce them respectively below.

Oracle VM VirtualBox

Oracle VM VirtualBox is a virtual machine software developed by Oracle that allows users to employ virtual machines to install other operating systems under the daily operating system. It is an open-source and free virtual machine software,



Fig. 5.1 Select the appropriate virtualbox version for download

which can be well adapted to various cross-platform development needs as a good tool for developers. And it allows running multiple virtual operating systems (such as Windows, Linux, and Solaris) on a single computer. Users can efficiently perform development operations on different systems by simply switching between different windows when using it. It is something that installing a system on a new physical machine cannot bring us.

To employ Oracle VM VirtualBox, users must first go to the official website to download the installation package. VirtualBox supports multiple platforms and offers download and installation solutions for various operating systems. Take an example of the Windows 1064bit operating system, with the most extensive user base, here it selects the Windows hosts version to download and install, as shown in Fig. 5.1.

After the download is complete, you will get an installation package file named VirtualBox-6.1.12-139181-Win.exe; double click on it to install and follow the prompts to select the installation configuration; if there are no special requirements, select the default. After successful installation, enter the main interface of the software, as shown in Fig. 5.2.

Vagrant

Vagrant is a tool that creates lightweight, highly reusable, and easily portable development environments for establishing and deploying virtualized development environments. In current real-world development, the first step to developing a project is to configure the development environment. However, as technology continues to evolve and software projects expand in size, the configuration of development environments becomes more complex. For example, a project may require a database, cache server, reverse proxy server, search engine server, web server, and live push server. Therefore, many times it is not an easy task to complete

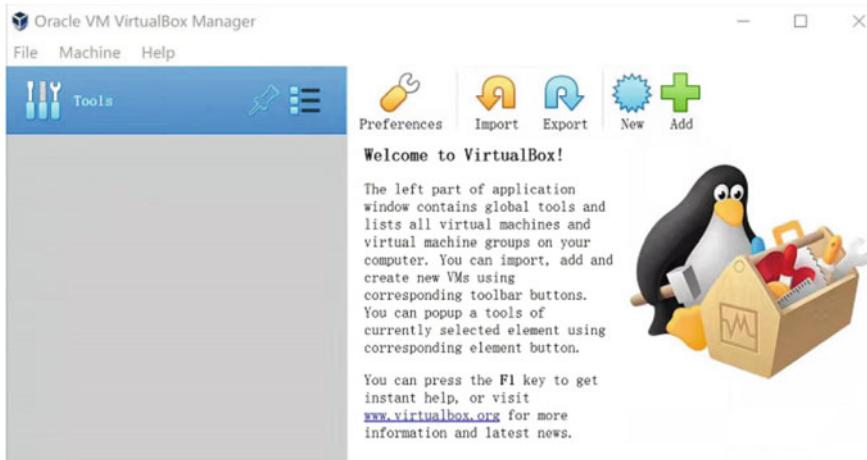


Fig. 5.2 VirtualBox main interface

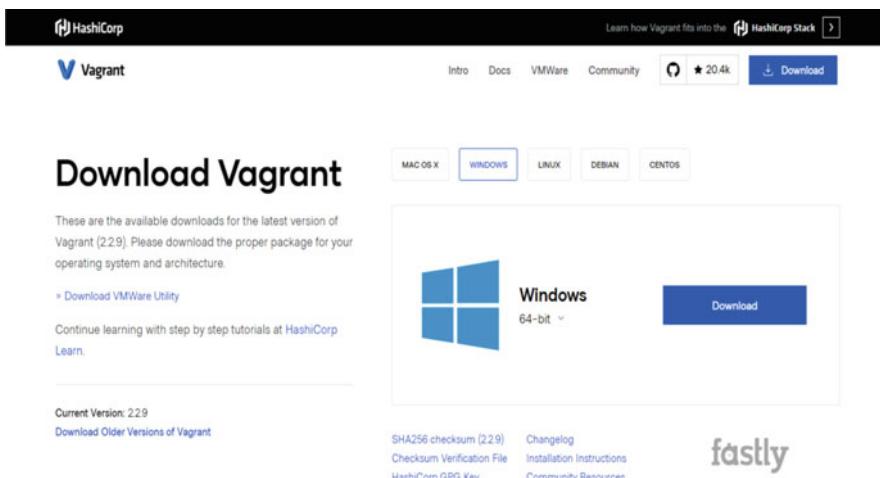


Fig. 5.3 Vagrant download interface

the first step. And Vagrant is a VirtualBox virtual machine with Linux installed; you can quickly realize the virtual development environment with some kits.

Please visit the official Vagrant website and go to the download screen as shown in Fig. 5.3. By selecting the Windows option, you will download the vagrant_2.2.9_x86_64.msi installation file and double click to install.

Git

Git is an open-source distributed version control software that can efficiently and quickly handle tiny to substantial projects. Developed by Linux founder Linus Torvalds to help manage Linux kernel development, it has evolved into the world's most popular version control software. Hyperledger Fabric, also established on Git for development management, hosts code on GitHub.

Please visit the Git website and select the Windows version to download and install, also available by default. Once installed, right-click on any folder to open the Git visualization interface via the GitGUIChere option or the Git console via the Git Bash Here option.

5.1.2 *Setting Up Development Environment*

After installing the required software, you can start Vagrant to create a virtual machine for the development environment. First, select a folder, copy the Hyperledger Fabric project locally by the following command and start Vagrant.

```
# Copy the project
git clone https://github.com/hyperledger/fabric.git
# Enter the vagrant virtual machine directory
cd fabric/devenv
# Start vagrant
vagrant up
# Connect to the VirtualBox virtual machine
vagrant ssh
```

The first execution of the vagrant up can take a long time because it includes many operations, such as downloading the required virtual machine image and performing various environment configurations according to the script. In the Windows system, you will need to use PuTTY or XShell to log in. The initial username and password will be in the Vagrantfile under the configuration folder of the corresponding box. Let us take a look at script operations during the execution of Vagrant up below.

When you execute the Vagrant up command for the first time, the system will first look for the existence of the required box image file. If not found, it will automatically download it and create a VirtualBox virtual machine. It will use the ssh protocol to connect to the virtual machine and execute the written shell script to configure the development environment. The environment configuration steps are as follows.

1. Update the system.

```
apt-get update
```

2. Install some basic tools and software.

```
apt-get install -y build-essential git make curl unzip g++ libtool
```

3. Install Docker and docker-compose.
4. Install Go language environment.
5. Install Node.js, a JavaScript runtime environment that makes it easy to build responsive and scalable web applications.

```
NODE_VER=8.9.4  
NODE_URL=https://nodejs.org/dist/v$NODE_VER/node-v$NODE_VER-  
linux-x64.tar.gz  
curl -sL$NODE_URL | (cd /usr/local && tar --strip-components 1 -xz )
```

6. Install Java environment, because Fabric supports Java contract code writing.

```
apt-get install -y openjdk-8-jdk maven  
wget https://services.gradle.org/distributions/gradle-5.5.1-  
bin.zip -P /tmp --quiet  
unzip -q /tmp/ gradle-5.5.1-bin.zip -d /opt && rm /tmp/  
gradle-5.5.1-bin.zip  
ln -s /opt/ gradle-5.5.1/bin/gradle /usr/bin
```

7. Make some various configurations, and the development environment configuration is complete.

Vagrant builds a development environment conveniently and comprehensively, which needs to establish on VirtualBox virtual machine. However, the final application often runs on a physical server. For the Fabric development environment on the physical machine, the two core items of Fabric are the Go language environment and the Docker environment. Fabric source code is written in Go language, while Fabric applications run in Docker containers. As for other tools and software, you can choose them according to needs at the development time.

5.1.3 Go and Docker

This section provides a detailed introduction to the Go language environment and the Docker environment.

Go Language Environment

Go, the second open-source programming language, is a statically typed, compiled programming language designed at Google in 2009. Programs compiled in Go have speed comparable to C or C++ code, being safer and supporting concurrent operations. Go, an excellent programming language with high expectations from Google, is designed to allow the software to take full advantage of simultaneous multiplexing of multicore processors and solve the troubles of object-oriented programming. It has a highly streamlined syntax with modern programming language features such as garbage collection mechanisms to help developers deal with trivial and troublesome memory management issues. Nowadays, most cloud platforms are developed on the Go language, such as the current Docker container. Blockchain technology is implemented in the Go language as the mainstream development language. For example, geth, the Go language client for Ethereum, and Hyperledger Fabric, which we have introduced, are also developed in Go language. The configuration of the Go language environment is described below.

1. First, download the Go language installer from the official website, as shown in Fig. 5.4.
2. Select the Linux version, copy the download link, and use the following command to download.

```
mkdir Download
cd Download
wget https://storage.googleapis.com/golang/go1.15.linux-amd64.tar.gz
```

The screenshot shows the Go language download interface. At the top, there is a navigation bar with links for 'Documents', 'Packages', 'The Project', 'Help', 'Blog', 'Play', a search bar, and a magnifying glass icon. Below the navigation bar, the title 'Downloads' is displayed. A sub-header below it reads: 'After downloading a binary release suitable for your system, please follow the installation instructions.' Below this, there is a note: 'If you are building from source, follow the source installation instructions.' Another note says: 'See the release history for more information about Go releases.' Further down, there is a section titled 'Featured downloads' with three cards:

- Microsoft Windows**
Windows 7 or later; Intel 64-bit processor
[go1.15.windows-amd64.msi](#) (115MB)
- Apple macOS**
macOS 10.12 or later; Intel 64-bit processor
[go1.15.darwin-amd64.pkg](#) (117MB)
- Linux**
Linux 2.6.23 or later; Intel 64-bit processor
[go1.15.linux-amd64.tar.gz](#) (116MB)

At the bottom left of the 'Featured downloads' section, there is a separate card for the 'Source' download:

- Source**
[go1.15.src.tar.gz](#) (22MB)

Fig. 5.4 Go language download interface

3. Unzip the zip file go1.15.linux-amd64.tar.gz to the /usr/local/ directory to install Go.

```
sudo tar -C /usr/local -xzf go1.15.linux-amd64.tar.gz
```

4. Configure Go environment variables.

```
mkdir $HOME/go  
sudo vim /etc/profile
```

Add the following Go environment variable below the judgment statement in the /etc./profile file.

The modified profile file looks like this.

```
if [ -d /etc/profile.d ]; then  
    for i in /etc/profile.d/*.*sh; do  
        if [ -r $i ]; then  
            . $i  
        fi  
        done  
        unset i  
    fi  
  
export PATH=$PATH:/usr/local/go/bin  
export GOPATH=$HOME/go
```

Run the source command to take effect, and then run the go env command to output the GO environment configuration information.

```
source /etc/profile  
go env
```

5. Test the Go language.

Enter the following command:

```
cd $GOPATH  
mkdir -p src/hello  
cd src/hello/  
vim hello.go
```

Input test code.

```
package main  
import "fmt"  
func main() {  
    fmt.Printf("hello, world\n")  
}
```

Save and exit; use the go build command to compile and execute; the following result indicates a successful Go environment configuration.

```
→ go build
→ ls
hello hello.go
→ ./hello
hello, world
```

Docker Environment

Docker, an open-source project, offers a software abstraction layer that becomes a container on top of the Linux operating system. Because of better portability, developers can package applications and dependency packages into a container, distribute them to any Linux machine, and virtualize it. Docker containers are closed to each other and do not have any interface.

The development of Hyperledger Fabric, described later in this chapter, adopts Docker containers to manage development. Hence, you need to first configure the Docker runtime environment on the Ubuntu operating system.

Docker offers a quick way to install Ubuntu in a virtual operating system with just a few command lines.

1. Add the remote repository address.

```
sudo apt-get -y install \
    apt-transport-https \
    ca-certificates \
    curl
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo
apt-key add -
sudo add-apt-repository \
    "deb [arch=amd64] https://download.docker.com/linux/ubuntu \
    $(lsb_release -cs) \
    stable"
sudo apt-get update
```

2. Install docker-ce

```
sudo apt-get -y install docker-ce
```

3. Test Docker

```
sudo docker run hello-world
```

Docker is successfully installed if the following information is displayed

```
→ sudo docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
78445dd45222: Pull complete
Digest:sha256:c5515758d4c5ele838e9c
d307f6c6a0d620b5e07e6f927b07d05f6d12a1ac8d7
Status: Downloaded newer image for hello-world:latest
```

```
Hello from Docker!
This message shows that your installation appears to be working
correctly.
```

After Docker is installed, additional docker-compose tool needs to be completed; the container thus can be deployed and started through the configuration file. Docker-compose installation adopts the following command.

```
sudo curl -L "https://github.com/docker/compose/releases/download/
1.23.2/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/
docker-compose
sudo chmod +x /usr/local/bin/docker-compose
```

Check the Docker and docker-compose versions. If the following information is displayed, the Docker environment will be installed.

```
→ docker -v
Docker version 18.09.2, build 6247962
→ docker-compose -v
docker-compose version 1.23.2, build 1110ad01
```

If the official Docker site is not accessible you can visit some mirror sites, such as AliCloud and CloudDao. You can set CloudDao to configure the Docker accelerator and download Docker images quickly by the following commands.

```
curl -sSL https://get.daocloud.io/daotools/set_mirror.sh | sh
-s http://176a5be3.m.daocloud.io
sudo systemctl restart docker.service
```

5.2 Chaincode Development Guide

Chaincode that runs in a secure Docker container isolated from the backing node is the code that implements some specific interfaces. The application initiates transactions that require the consent of all network members, submits events to the chaincode, and initializes or manages the data in the ledger through the chaincode.

Chaincode, known as smart contracts, can be written in Go, Node.js, Java, etc. This section describes how to write chaincode using the Go language.

5.2.1 Interface Introduction

In Go language, the chaincode mainly adopts the functions in the shim package to complete the business logic of Fabric transactions. In chaincode writing, you need to import the shim package.

The shim package defines two critical structures, Chaincode and ChaincodeStubInterface. The former contains two functions that must be implemented by Fabric runtime. The latter provides a series of functions to query and update the ledger. We will describe each of them below.

The Chaincode structure defines the Init() function and the Invoke() function, which is used to initialize the data in the ledger and is called when the chaincode receives the install command in the cli operation in Sect. 5.3. The Invoke() function handles requests to query or update a ledger. It can take a specific request as an argument and call the corresponding function accordingly.

ChaincodeStubInterface structure includes all the functions to manipulate the blockchain ledger, and some standard functions are introduced below.

- **GetState(key string)([]byte, error)**: This function receives a query parameter key, queries the data in the ledger based on this key, and returns it as a stream of bytes.
- **PutState(key string, value []byte) error**: Corresponding to the GetState method, this function receives two parameters to store the value in the form of a byte stream with the first parameter key.
- **DelState(key string) error**: Receive one parameter key, and delete the data corresponding to the key in the ledger.
- **GetStateByRange(startKey, endKey string) (StateQueryIteratorInterface, error)**: Receive two parameters, a start Key value (nonunique) and an end Key value (unique), and return the queried data as an iterator.
- **CreateCompositeKey(objectType string, attributes []string) (string, error)**: The attributes in the attributes slice are combined into a new key. The two parameters received by this function must be valid utf8 strings and cannot contain U + 0000 and U + 10FFFF. The combined new key can be used as a PutState() function parameter.

- **SplitCompositeKey(compositeKey string) (string, []string, error)**: Split the constructed composite key into scattered attributes.
- **GetStateByPartialCompositeKey(objectType string, keys []string) (StateQueryIteratorInterface, error)**: Receive a composite key as an argument, query the data containing the composite key in the ledger, and return it as an iterator. If the keys in the argument are not part of the composite key, an empty iterator will be returned.

Fabric v1.4 introduces the concept of private data, allowing members to set some of their data as private. When executing query function, private data is only visible to Peer nodes in authorized members. For example, the private data of member Org1 cannot be read by Peer nodes in Org2, which further ensures the data security and control; thus, Fabric v1.4 adds some interfaces to operate on private data as follows.

- **GetPrivateData (collection, key string) ([]byte, error)**: Receive a determined private data collection name collection parameter and a key parameter for querying, and return the result of the query as a byte stream. An error is returned if the queried data does not have the corresponding read permission.
- **PutPrivateData(collection string, key string, value []byte) error**: Set the name of the seeming collection and write the key value and the corresponding data to the ledger.
- **DelPrivateData(collection, key string) error**: Delete the private data corresponding to the key value.

5.2.2 Case Study

This section will take the marbles02 case in fabric-samples/chaincode on GitHub as a reference and introduce Fabric's relevant data manipulation interface by simulating marble generation, query, transfer by color, and other operations.

initMarble Initialize Data

```

if len(args) != 4 {
    return shim.Error("Incorrect number of arguments. Expecting 4")
} // Check if the number of parameters entered is correct
fmt.Println("- start init marble")
// check if each argument is legal separately
if len(args[0]) <= 0 {
    return shim.Error("1st argument must be a non-empty string")
}
if len(args[1]) <= 0 {
    return shim.Error("2nd argument must be a non-empty string")
}
if len(args[2]) <= 0 {
    return shim.Error("3rd argument must be a non-empty string")
}

```

```
}

if len(args[3]) <= 0 {
    return shim.Error("4th argument must be a non-empty string")
}
marbleName := args[0]
color := strings.ToLower(args[1])
owner := strings.ToLower(args[3])
size, err := strconv.Atoi(args[2])
if err != nil {
    return shim.Error("3rd argument must be a numeric string")
}
// Get the state of the ledger first and check if the marble already exists
marbleAsBytes, err := stub.GetState(marbleName)
if err != nil {
    return shim.Error("Failed to get marble: " + err.Error())
} else if marbleAsBytes != nil {
    fmt.Println("This marble already exists: " + marbleName)
    return shim.Error("This marble already exists: " + marbleName)
}
// Create the marble object and convert the data to JSON format
objectType := "marble"
marble := &marble{objectType, marbleName, color, size, owner}
marbleJSONAsBytes, err := json.Marshal(marble)
if err != nil {
    return shim.Error(err.Error())
}
// Set key to marbleName and store data as key-value
err = stub.PutState(marbleName, marbleJSONAsBytes)
if err != nil {
    return shim.Error(err.Error())
}
// Create a key combination of color and name to store the data in the
book as a composite key
indexName := "color~name"
// Call the CreateCompositeKey interface to set the composite key
colorNameIndexKey, err := stub.CreateCompositeKey(indexName, []
string{marble.Color,
    marble.Name})
if err != nil {
    return shim.Error(err.Error())
}
value := []byte{0x00}
stub.PutState(colorNameIndexKey, value) // Store a null byte into the
ledger
// Initialize successfully; return the result
fmt.Println("- end init marble")
return shim.Success(nil)
```

readMarble Query Marbles Data

```

    var name, jsonResp string
var err error
// Check the number of arguments
if len(args) != 1 {
    return shim.Error("Incorrect number of arguments. Expecting name of
the marble to query")
}
name = args[0]
valAsbytes, err := stub.GetState(name) // Query data from the ledger
with name as the key
if err != nil {
    jsonResp = "{\"Error\":\"Failed to get state for " + name + "\"}"
    return shim.Error(jsonResp)
} else if valAsbytes == nil {
    jsonResp = "{\"Error\":\"Marble does not exist: " + name + "\"}"
    return shim.Error(jsonResp)
}
// Return the results as a binary stream
return shim.Success(valAsbytes)

```

transferMarble

```

if len(args) < 2 {
    return shim.Error("Incorrect number of arguments. Expecting 2")
}
// Set related properties
marbleName := args[0]
newOwner := strings.ToLower(args[1])
fmt.Println("- start transferMarble ", marbleName, newOwner)
// Search for the original marbles and their related information from
the ledger
marbleAsBytes, err := stub.GetState(marbleName)
if err != nil {
    return shim.Error("Failed to get marble:" + err.Error())
} else if marbleAsBytes == nil {
    return shim.Error("Marble does not exist")
}
marbleToTransfer := marble{}
err = json.Unmarshal(marbleAsBytes, &marbleToTransfer) // Data
format conversion
if err != nil {
    return shim.Error(err.Error())
}
marbleToTransfer.Owner = newOwner // Transaction
marbleJSONAsBytes, _ := json.Marshal(marbleToTransfer)
err = stub.PutState(marbleName, marbleJSONAsBytes) // Writing new
data to the ledger
if err != nil {
    return shim.Error(err.Error())
}

```

```

    }
    fmt.Println("- end transferMarble (success) ")
    return shim.Success(nil)
}

```

transferMarblesBasedOnColor

```

if len(args) < 2 {
    return shim.Error("Incorrect number of arguments. Expecting 2")
}
color := args[0]
newOwner := strings.ToLower(args[1])
fmt.Println("- start transferMarblesBasedOnColor ", color, newOwner)
// Query the ledger by the color-name composite key to get an iterator
coloredMarbleResultsIterator, err := stub.
GetStateByPartialCompositeKey("color~name",
    []string{color})
if err != nil {
    return shim.Error(err.Error())
}
defer coloredMarbleResultsIterator.Close()
// Iterate over all results, trading all marbles of this color
var i int
for i = 0; coloredMarbleResultsIterator.HasNext(); i++ {
    responseRange, err := coloredMarbleResultsIterator.Next()
    if err != nil {
        return shim.Error(err.Error())
    }
    // Decompose the key combination to get the color and name
    objectType, compositeKeyParts, err := stub.SplitCompositeKeSpltiy
(responseRange.Key)
    if err != nil {
        return shim.Error(err.Error())
    }
    returnedColor := compositeKeyParts[0]
    returnedMarbleName := compositeKeyParts[1]
    fmt.Printf("- found a marble from index:%s color:%s name:%s\n",
objectType, returnedColor,
    returnedMarbleName)
    // Call the previous transferMarble function to initiate the
transaction
    response := t.transferMarble(stub, []string{returnedMarbleName,
newOwner})
    // Check the transaction result for each element in the iterator
    if response.Status != shim.OK {
        return shim.Error("Transfer failed: " + response.Message)
    }
}
//Print the result of the transaction for each one of the marbles
responsePayload := fmt.Sprintf("Transferred %d %s marbles to %s", i,
color, newOwner)

```

```
fmt.Println("- end transferMarblesBasedOnColor: " + responsePayload)
return shim.Success([]byte(responsePayload))
```

5.2.3 Introduction to Private Data

Fabric v1.4 adds the ability to have private data. In the example marbles02_private, the structure marble and marblePrivateDetails can be defined in JSON format as follows:

```
type marble struct {
    ObjectType string      `json:"docType"`
    Name        string      `json:"name"`
    Color       string      `json:"color"`
    Size        int         `json:"size"`
    Owner       string      `json:"owner"`
}
type marblePrivateDetails struct {
    ObjectType string      `json:"docType"`
    Name        string      `json:"name"`
    Price       int         `json:"price"`
}
```

To use private data, you need first to construct a collection that defines access to the private data. Specify the authorized organization members, the number of peer nodes to which the data is assigned, the number of peer nodes required to disseminate the private data, and the amount of time the private data will remain in the private database.

```
[
  {
    "name": "collectionMarbles",
    "policy": "OR('Org1MSP.member', 'Org2MSP.member')",
    "requiredPeerCount": 0,
    "maxPeerCount": 3,
    "blockToLive": 1000000,
    "memberOnlyRead": true
  },
  {
    "name": "collectionMarblePrivateDetails",
    "policy": "OR('Org1MSP.member')",
    "requiredPeerCount": 0,
    "maxPeerCount": 3,
    "blockToLive": 3,
    "memberOnlyRead": true
  }
]
```

The example marbles02_private contains two private data collection definitions: collectionMarbles and collectionMarblePrivateDetails. The collectionMarbles allows all channel members (Org1 and Org2) to read and modify data, while the collectionMarblePrivateDetails allows only members of Org1 to manipulate private data. In this case, the name, color, size, and owner of the marble structure definition will be visible to all channel members. The price defined in the marblePrivateDetail structure will only be visible to authorized organization members (Org1) on the private dataset.

Writing Private Data

The use of private data is introduced in the chaincode of marbles02_private, which allows you to set private data to be accessed by authorized organization members.

```
// Create marble object and convert it to JSON format
marble := &marble{
    ObjectType: "marble",
    Name:      marbleInput.Name,
    Color:     marbleInput.Color,
    Size:      marbleInput.Size,
    Owner:     marbleInput.Owner,
}
marbleJSONasBytes, err := json.Marshal(marble)
if err != nil {
    return shim.Error(err.Error())
}
// Add the public data to the ledger first
err = stub.PutPrivateData("collectionMarbles", marbleInput.Name,
marbleJSONasBytes)
if err != nil {
    return shim.Error(err.Error())
}
// Create a private data object and convert it to JSON format
marblePrivateDetails := &marblePrivateDetails{
    ObjectType: "marblePrivateDetails",
    Name:      marbleInput.Name,
    Price:     marbleInput.Price,
}
marblePrivateDetailsBytes, err := json.Marshal(
(marblePrivateDetails)
if err != nil {
    return shim.Error(err.Error())
}
// Write the private data to the ledger
err = stub.PutPrivateData("collectionMarblePrivateDetails",
marbleInput.Name,
marblePrivateDetailsBytes)
if err != nil {
```

```
    return shim.Error(err.Error())
}
```

This function employs the PutPrivateData() interface to write data to the database. Since the data storage is divided into storing public data and private data, the interface needs to be called twice.

Read Private Data

```
var name, jsonResp string
var err error
if len(args) != 1 {
    return shim.Error("Incorrect number of arguments. Expecting name of
the marble to query")
}
name = args[0]
// Query Marbles private data
valAsbytes, err := stub.GetPrivateData(
    "collectionMarblePrivateDetails", name)
if err != nil {
    jsonResp = "{\"Error\":\"Failed to get private details for " + name +
": " + err.Error() + "\"}"
    return shim.Error(jsonResp)
} else if valAsbytes == nil {
    // The private data section does not exist
    jsonResp = "{\"Error\":\"Marble private details does not exist: " +
name + "\"}"
    return shim.Error(jsonResp)
}
return shim.Success(valAsbytes)
```

This function calls the GetPrivateData() interface to query the private data of marbles. If the private data does not exist in the obtained marbles object, an error will be returned signaling the data does not exist, and, finally, the queried private data will be returned in byte form.

5.3 CLI Application Examples

Fabric now supports two modes of application development: CLI and SDK interfaces. CLI is short for Command Line Interface. SDK stands for Software Development Kit. This section describes how to use the command line interface (CLI) to develop deployment and invoke chaincode.

The CLI command line provided by Fabric has been described and explained in Chap. 4. In this section, you will learn how to use the CLI command line interface through the examples in the /fabric-samples/first-network/directory.

5.3.1 Preparation

1. Copy the hyperledger/fabric-samples project locally

```
# Create fabric workspace
cd $GOPATH
mkdir -p src/github.com/hyperledger
# Copy the fabric-samples project
git clone https://github.com/hyperledger/fabric-samples.git
```

2. Install specific files and pull docker images

Upon preparing the fabric-samples project, you need to download the specified version of the Fabric platform binaries and the related configuration files and install them into the /bin directory and /config directory under the fabric-samples folder, respectively.

```
# Go to the project folder
cd src/github.com/hyperledger/fabric-samples
# Install the relevant files and pull the docker image
curl -sSL https://bit.ly/2ysbOFE | bash -s
# If there is an error with the above command, you can replace the link
in the command with the following link:
https://raw.githubusercontent.com/hyperledger/fabric/master/
scripts/bootstrap.sh
```

The above script includes downloading binaries and configuration files and pulling Docker images.

First, the binaries configtxgen, configtxlator, cryptogen, discover, fabric-ca-client, idemixgen, order, peer are generated in the bin folder of fabric-samples for the next Fabric network opening process, or you can add them to the system PATH environment with the following command.

```
export PATH = $GOPATH/src/github.com/hyperledger/fabric-samples/
bin:$PATH
```

Finally, the script will download the relevant Hyperledger Fabric docker images from the official Docker Hub to the local Docker registry, setting their markers to the latest.

A list of downloaded images is displayed after execution and the result is shown below.

```
====> List out Hyperledger docker images
hyperledger/fabric-ca           1.4.1   3a1799cda5d7   3 months ago
252MB
hyperledger/fabric-ca           latest   3a1799cda5d7   3 months ago
```

252MB	hyperledger/fabric-tools	1.4.1	432c24764fbb	3 months ago
1.55GB	hyperledger/fabric-tools	latest	432c24764fbb	3 months ago
1.55GB	hyperledger/fabric-ccenv	1.4.1	d7433c4b2a1c	3 months ago
1.43GB	hyperledger/fabric-ccenv	latest	d7433c4b2a1c	3 months ago
1.43GB	hyperledger/fabric-orderer	1.4.1	ec4ca236d3d4	3 months ago
173MB	hyperledger/fabric-orderer	latest	ec4ca236d3d4	3 months ago
173MB	hyperledger/fabric-peer	1.4.1	a1e3874f338b	3 months ago
178MB	hyperledger/fabric-peer	latest	a1e3874f338b	3 months ago
178MB	hyperledger/fabric-javaenv	1.4.1	b8c9d7ff6243	3 months ago
1.74GB	hyperledger/fabric-javaenv	latest	b8c9d7ff6243	3 months ago
1.74GB	hyperledger/fabric-zookeeper	0.4.15	20c6045930c8	4 months ago
1.43GB	hyperledger/fabric-zookeeper	latest	20c6045930c8	4 months ago
1.43GB	hyperledger/fabric-kafka	0.4.15	b4ab82bbaf2f	4 months ago
1.44GB	hyperledger/fabric-kafka	latest	b4ab82bbaf2f	4 months ago
1.44GB	hyperledger/fabric-couchdb	0.4.15	8de128a55539	4 months ago
1.5GB	hyperledger/fabric-couchdb	latest	de128a55539	4 months ago
1.5GB	hyperledger/fabric-baseimage	amd64-0.4.15	c4c532c23a50	
4 months ago	1.39GB			
	hyperledger/fabric-baseos	amd64-0.4.15	9d6ec11c60ff	4 months ago
				145MB

5.3.2 Writing the Code

In this example, three code files need to be written, containing two YAML configuration files and one Go language chaincode file. Among them, configtx.yaml is used to configure the initial block and channel rules, which defines a Fabric blockchain network containing one Orderer node and four Peer nodes. Docker-compose.yaml, on the other hand, is the configuration file for the docker-compose tool employed to configure various properties and operations between containers. They are described in detail below.

1. configtx.yaml

This configuration file defines the channel rules for the instance network and employs the Configtxgen tool below to generate the Genesis block and channel configuration files.

```
# Define the member service ordererOrg and the two groups Org1 and Org2
Organizations:
- &OrdererOrg
  Name: OrdererOrg
  ID: OrdererMSP
# Member service paths
  MSPDir: crypto-config/ordererOrganizations/example.com/
msp
# Define the encryption policy
Policies:
  Readers:
    Type: Signature
    Rule: "OR('OrdererMSP.member')"
  Writers:
    Type: Signature
    Rule: "OR('OrdererMSP.member')"
  Admins:
    Type: Signature
    Rule: "OR('OrdererMSP.admin')"
- &Org1
  Name: Org1MSP
  ID: Org1MSP
  MSPDir: crypto-config/peerOrganizations/org1.example.com/msp
  Policies:
    Readers:
      Type: Signature
      Rule: "OR('Org1MSP.admin', 'Org1MSP.peer', 'Org1MSP.
client')"
    Writers:
      Type: Signature
      Rule: "OR('Org1MSP.admin', 'Org1MSP.client')"
    Admins:
      Type: Signature
      Rule: "OR('Org1MSP.admin')"
# Define anchor nodes
AnchorPeers:
- Host: peer0.org1.example.com
  Port: 7051
- &Org2
...
# Define the information related to the ordering service
Orderer: &OrdererDefaults
# Ordering services are currently available as solo, kafka, raft
OrdererType: solo
Addresses:
- orderer.example.com:7050
BatchTimeout: 2s
BatchSize:
  MaxMessageCount: 10
```

```
AbsoluteMaxBytes: 99 MB
PreferredMaxBytes: 512 KB
Kafka:
  Brokers:
    - 127.0.0.1:9092
Organizations:
Policies:
  Readers:
    Type: ImplicitMeta
    Rule: "ANY Readers"
  Writers:
    Type: ImplicitMeta
    Rule: "ANY Writers"
  Admins:
    Type: ImplicitMeta
    Rule: "MAJORITY Admins"
BlockValidation:
  Type: ImplicitMeta
  Rule: "ANY Writers"
```

2. docker-compose-cli.yaml

```
version: '2'
volumes:
  orderer.example.com:
  peer0.org1.example.com:
  peer1.org1.example.com:
  peer0.org2.example.com:
  peer1.org2.example.com:
networks:
  byfn:
services:
  # The only consensus node in this network, orderer0, is used to
  # provide ordering service
  orderer.example.com:
    extends:
      file: base/docker-compose-base.yaml
      service: orderer.example.com
      container_name: orderer.example.com
    networks:
      - byfn
  # Configuration information for the 4 peer nodes
  peer0.org1.example.com:
    container_name: peer0.org1.example.com
    extends:
      file: base/docker-compose-base.yaml
      service: peer0.org1.example.com
    networks:
      - byfn
  peer1.org1.example.com:
    ...
  peer0.org2.example.com:
    ...
```

```

peer1.org2.example.com:
...
# The cli container is used to provide a command line operating
environment
cli:
  container_name: cli
  image: hyperledger/fabric-tools:$IMAGE_TAG
  tty: true
  stdin_open: true
# Environment variables are omitted here
environment:
  # [see documentation for detail]
# Working directory
working_dir: /opt/gopath/src/github.com/hyperledger/fabric/
peer
# Run the script
command: /bin/bash
# Disk Area
volumes:
  - /var/run/:/host/var/run/
  - ./chaincode:/opt/gopath/src/github.com/chaincode
  - ./crypto-config:/opt/gopath/src/github.com/hyperledger/
fabric/peer/crypto/
  - ./scripts:/opt/gopath/src/github.com/hyperledger/fabric/
peer/scripts/
  - ./channel-artifacts:/opt/gopath/src/github.com/
hyperledger/fabric/peer/channel-artifacts
# Dependencies
depends_on:
  - orderer.example.com
  - peer0.org1.example.com
  - peer1.org1.example.com
  - peer0.org2.example.com
  - peer1.org2.example.com
networks:
  - byfn

```

3. example_chaincode02.go

```

package main
// Required dependency packages
import (...)

// SimpleChaincode implement the contract code interface
type SimpleChaincode struct {}

// Init Interface implementation for initialization
func (t *SimpleChaincode) Init(stub shim.ChaincodeStubInterface)
pb.Response {
    fmt.Println("ex02 Init")
    _, args := stub.GetFunctionAndParameters()
    var A, B string // Entity A, B
    var Aval, Bval int // The value corresponding to the entity
    var err error // Error
    # Too many parameters
}

```

```
if len(args) != 4 {
    return shim.Error("Incorrect number of arguments. Expecting
4")
}
// Instantiate the contract code object, A assignment operation
A = args[0]
Aval, err = strconv.Atoi(args[1])
if err != nil {
    return shim.Error("Expecting integer value for asset holding")
}
// Instantiate the contract code object, B assignment operation
B = args[2]
Bval, err = strconv.Atoi(args[3])
if err != nil {
    return shim.Error("Expecting integer value for asset holding")
}
fmt.Printf("Aval = %d, Bval = %d\n", Aval, Bval)
// Write A to the state variable
err = stub.PutState(A, []byte(strconv.Itoa(Aval)))
if err != nil {
    return shim.Error(err.Error())
}
// Write B to the state variable
err = stub.PutState(B, []byte(strconv.Itoa(Bval)))
if err != nil {
    return shim.Error(err.Error())
}
return shim.Success(nil)
}
// Invoke implementation method
func (t *SimpleChaincode) Invoke(stub shim.
ChaincodeStubInterface) pb.Response {
fmt.Println("ex02 Invoke")
function, args := stub.GetFunctionAndParameters()
if function == "invoke" {
    // Calling the invoke method
    return t.invoke(stub, args)
} else if function == "delete" {
    // Calling the delete method
    return t.delete(stub, args)
} else if function == "query" {
    // Calling the query method
    return t.query(stub, args)
}
return shim.Error("Invalid invoke function name. Expecting
\"invoke\" \"delete\" \"query\"")
}
// invoke method
func (t *SimpleChaincode) invoke(stub shim.
ChaincodeStubInterface, args []string) pb.Response {
var A, B string // EntityA,B
var Aval, Bval int // The value corresponding to A, B
var X int        // The value to be transferred for the transaction
```

```

var err error // error
if len(args) != 3 {
    return shim.Error("Incorrect number of arguments. Expecting
3")
}
// Assignment A, B
A = args[0]
B = args[1]
// Get the state variable of A
Avalbytes, err := stub.GetState(A)
if err != nil { return shim.Error("Failed to get state") }
if Avalbytes == nil { return shim.Error("Entity not found") }
Aval, _ = strconv.Atoi(string(Avalbytes))
Bvalbytes, err := stub.GetState(B)
if err != nil { return shim.Error("Failed to get state") }
if Bvalbytes == nil { return shim.Error("Entity not found") }
Bval, _ = strconv.Atoi(string(Bvalbytes))
// Execute the call
X, err = strconv.Atoi(args[2])
if err != nil { return shim.Error("Invalid transaction amount,
expecting a integer value") }
// Value operation, A decreases, B increases
Aval = Aval - X
Bval = Bval + X
fmt.Printf("Aval = %d, Bval = %d\n", Aval, Bval)
// Write A to the state variable
err = stub.PutState(A, []byte(strconv.Itoa(Aval)))
if err != nil { return shim.Error(err.Error()) }
// Write B to the state variable
err = stub.PutState(B, []byte(strconv.Itoa(Bval)))
if err != nil { return shim.Error(err.Error()) }
return shim.Success(nil)
}
// Delete state variable method
func (t *SimpleChaincode) delete(stub shim.
ChaincodeStubInterface, args []string) pb.Response {
    if len(args) != 1 { return shim.Error("Incorrect number of
arguments. Expecting 1") }
    A := args[0]
    // Delete state variables based on key
    err := stub.DelState(A)
    if err != nil { return shim.Error("Failed to delete state") }
    return shim.Success(nil)
}
// Query contract code method based on key query value
func (t *SimpleChaincode) query(stub shim.
ChaincodeStubInterface, args []string) pb.Response {
    var A string // Entity A
    var err error
    if len(args) != 1 { return shim.Error("Incorrect number of
arguments. Expecting name of the person to query") }
    A = args[0]
    // Get the state variable of A
}

```

```

Avalbytes, err := stub.GetState(A)
if err != nil {
    jsonResp := "{\"Error\":\"Failed to get state for " + A + "\"}"
    return shim.Error(jsonResp)
}
if Avalbytes == nil {
    jsonResp := "{\"Error\":\"Nil amount for " + A + "\"}"
    return shim.Error(jsonResp)
}
// JSON format response
jsonResp := "{\"Name\":\"" + A + "\", \"Amount\":\"" + string
(Avalbytes) + "\"}"
fmt.Printf("Query Response:%s\n", jsonResp)
return shim.Success(Avalbytes)
}
// main function
func main() {
    err := shim.Start(new(SimpleChaincode))
    if err != nil {
        fmt.Printf("Error starting Simple chaincode: %s", err)
    }
}

```

5.3.3 *Initiate Network and Chaincode Calls*

This section will employ the first-network sample. In the first-network directory, a script is provided that includes operations such as creating channels and deploying chaincode. The script can be executed to complete the one-stop experience.

```

# Go to the first-network subdirectory
1. Cd first-network
1. Generate the Genesis block and channel

```

```

# Execute the byfn.sh script
./byfn.sh generate

```

Next, you will see a brief description of what is about to happen and the option to execute it or not, type y or press enter to execute it.

```

# Generating certs and genesis block for channel 'mychannel' with CLI
timeout of '10' seconds and CLI delay of '3' seconds
Continue? [Y/n]
Select y

```

The first step generates various certificates and keys for the network entities, genesis blocks for consensus services, and some needed to configure the channels.

2. The script automatically starts the network and performs related operations

```
# Run the network startup command
./byfn.sh up
```

Again the action to be performed will be displayed, enter y or press enter to confirm. (Node.js and Java are also supported, the commands are. /byfn.sh up -l node and. /byfn.sh up -l java.)

This command will start all Docker containers and launch the complete end-to-end application. After successfully starting the network, the following will be reported in the terminal.

```
===== All GOOD, BYFN execution completed =====
```



3. Shutting Down the Network.

Shutting down the network is done with the byfn.sh script. Shutting down the network will terminate all Docker containers, delete the four configuration artifacts generated earlier, remove the encrypted material, and delete the associated contract code image from the Docker registry.

```
# Execute the network shutdown command
./byfn.sh down
```

5.3.4 Switching on the Network Manually

After performing all the operations via scripts in the previous section, you need to perform the above operations manually, such as creating channels, joining channels, installing chaincode, and calling chaincode.

1. Manual certificate and channel generation.

Employing the cryptogen tool in the bin directory, manually generate the certificate/key (MSP) material established on the network configuration defined in the crypro-config.yaml file and export it to the crypto-config folder in the first-network directory.

```
..../bin/cryptogen generate --config=./crypto-config.yaml
```

Output the following in the terminal.

```
org1.example.com  
org2.example.com
```

Next you need to create four channel configuration artifacts by the configtxgen tool.

```
# # Create orderer genesis blocks  
..../bin/configtxgen -profile TwoOrgsOrdererGenesis -channelID  
byfn-sys-channel -outputBlock ./channel-artifacts/genesis.block  
# Set the channel name to mychannel  
export CHANNEL_NAME=mychannel  
# Create channel channel configuration channel.tx  
..../bin/configtxgen -profile TwoOrgsChannel  
-outputCreateChannelTx ./channel-artifacts/channel.tx  
-channelID mychannel  
# Define the anchor node of Org1  
..../bin/configtxgen -profile TwoOrgsChannel  
-outputAnchorPeersUpdate ./channel-artifacts/  
Org1MSPanchors.tx -channelID mychannel -asOrg Org1MSP  
# Define the anchor node of Org2  
..../bin/configtxgen -profile TwoOrgsChannel  
-outputAnchorPeersUpdate ./channel-artifacts/  
Org2MSPanchors.tx -channelID mychannel -asOrg Org2MSP
```

2. Start the network with the docker-compose command.

```
docker-compose -f docker-compose-cli.yaml up -d
```

3. Go to the console of the cli container.

```
docker exec -it cli bash
```

4. Create the channel.

```
peer channel create -o orderer.example.com:7050 -c mychannel -f ./  
channel-artifacts/channel.tx --tls --cafile /opt/gopath/src/  
github.com/hyperledger/fabric/peer/crypto/  
ordererOrganizations/  
example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.  
example.com-cert.pem
```

5. Set environment variables for each node.

Whenever you use the peer command, you need to specify the Peer node for the operation by configuring the global environment variables, such as the environment variables for peer0.org1 as.

```
CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/
hyperledger/fabric/peer/crypto/peerOrganizations/org1.example.
com/users/Admin@org1.example.com/msp
CORE_PEER_ADDRESS=peer0.org1.example.com:7051
CORE_PEER_LOCALMSPID="Org1MSP"
CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/
hyperledger/fabric/peer/crypto/peerOrganizations/org1.example.
com/peers/peer0.org1.example.com/tls/ca.crt
```

Similarly, we can set the environment variables for the remaining three nodes, e.g., peer0.org2 as.

```
CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/
hyperledger/fabric/peer/crypto/peerOrganizations/org2.example.
com/users/Admin@org2.example.com/msp
CORE_PEER_ADDRESS=peer0.org2.example.com:9051
CORE_PEER_LOCALMSPID="Org2MSP"
CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/
hyperledger/fabric/peer/crypto/peerOrganizations/org2.example.
com/peers/peer0.org2.example.com/tls/ca.crt
```

Before executing commands on each node, switch the environment to the node that needs to be executed; the above code will be omitted below.

Similarly, you can set the environment variable of the orderer node to replace the TLS certificate directory of the orderer node above.

```
ORDERER_TLS=/opt/gopath/src/github.com/hyperledger/fabric/
peer/crypto/ordererOrganizations/
example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.
example.com-cert.pem
```

6. Add the node to the channel.

```
# Switch to the environment variable of peer0.org1
# Join peer0.org1 to the channel
peer channel join -b mychannel.block
# Switch to the environment variable of peer0.org2
# Add peer0.org2 to the channel
peer channel join -b mychannel.block
# Switch to the environment variable of peer0.org1
# Update the channel to define the anchor node of Org1 as peer0.
org1.example.com
peer channel update -o orderer.example.com:7050 -c mychannel -f .
/channel-artifacts/Org1MSPanchors.tx --tls --cafile $ORDERER_TLS
# Switch to the peer0.org2 environment variable
```

```
# Update the channel to define the Org2 anchor node as peer0.org2.example.com
peer channel update -o orderer.example.com:7050 -c mychannel -f ./channel-artifacts/Org2MSPanchors.tx --tls --cafile $ORDERER_TLS
```

- After the Peer node joins the channel, you can install the chaincode on the Peer node; the command is peer chaincode install. At present, the chaincode of Fabric supports three languages: Go, Node.js, and Java. Go language's chaincode is employed here.

```
# Switch to peer0.org1 environment
# Install the contract code on Org1
peer chaincode install -n mycc -v 1.0 -p github.com/chaincode/chaincode_example02/go/
# Switch to peer0.org2 environment
# Install the contract code on Org2
peer chaincode install -n mycc -v 1.0 -p github.com/chaincode/chaincode_example02/go/
```

- After successfully installing the chaincode on the Peer node, you can instantiate the chaincode on the corresponding node and obtain the chaincode object. Here we instantiate the object mycc on peer0.Org2 node with parameters to initialize the values of a and b and specify the endorsement policy as "OR."

```
# Switch to the peer0.Org2 environment and instantiate the chaincode
peer chaincode instantiate -o orderer.example.com:7050 --tls --cafile $ORDERER_TLS -C mychannel -n mycc -v 1.0 -c '{"Args": ["init", "a", "100", "b", "200"]}' -P "OR ('Org1MSP.peer', 'Org2MSP.peer')"
```

- Querying the value of a on the peer0.Org1 node will give the result 100.

```
# Switch to peer0.Org1's environment, call query
peer chaincode query -C mychannel -n mycc -c '{"Args": ["query", "a"]}'
```

- Send a transaction on peer0.Org1, invoke method to transfer 10 from a to b.

```
# invoke to transfer assets
peer chaincode invoke -o orderer.example.com:7050 --tls true --cafile $ORDERER_TLS -C mychannel -n mycc --peerAddresses peer0.org1.example.com:9051 --tlsRootCertFiles/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/ca.crt --peerAddresses peer0.org2.example.com:9051 --tlsRootCertFiles/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org2.example.com/peers/
```

```
peer0.org2.example.com/tls/ca.crt -c '{"Args": ["invoke", "a", "b", "10"]}'
```

11. Install the chaincode on peer1.Org2 and query the value of a.

```
# Switch to peer1.org2 environment and install the chaincode
peer chaincode install -n mycc -v 1.0 -p github.com/chaincode/
chaincode_example02/go/
# Call query on peer1.Org2 to query the value of a
peer chaincode query -C mychannel -n mycc -c '{"Args": ["query", "a"]}'
# The result is 90. Correct!
```

Program execution is complete.

5.4 SDK Application Examples

The Hyperledger Fabric SDK provides a structured library environment for developers to write and test chaincode applications. The SDK provided by Fabric is fully configurable and extensible through a standard interface. The SDK API employs a gRPC-based protocol buffer to provide functions such as transaction processing, member service management, and node traversal. This section will focus on developing Fabric-based blockchain applications using the SDK API.

5.4.1 *Introduction to the SDK*

There are multiple implementations of the Hyperledger Fabric SDK client, including Go, Node.js, Java, and Python. This example is introduced using Node.js as an example. The Hyperledger Fabric SDK for Node.js is designed in an object-oriented programming style. Its modularity allows application developers to insert their implementations of core functions based on the API, such as encryption algorithms, persistent storage of states, and logging. The APIs provided by the Hyperledger Fabric Node.js SDK client can be divided into two categories: fabric-ca-client and fabric-client.

The fabric-ca-client interacts with the fabric-ca component to provide membership management services. The methods it offers are shown in Table 5.1.

The fabric-client interacts with the core components of the Hyperledger Fabric, such as Peer nodes, Orderer nodes, and event flows. Some of the interfaces it provides are shown in Table 5.2.

Table 5.1 Fabric-client interface

Command line parameters	Function
NewFabricCAClient()	Create a new Fabric-client
Enroll(request)	Enroll a registered user
Reenroll(currentUser, Optional)	Register a user who has already been registered
Register(request, registrar)	Register
Revoke(request, registrar)	Revoke a certificate
createSigningIdentity(user)	Create a signed identity

For a more detailed SDK interface implementation, please refer to hyperledger/fabric-sdk-node/tree/master/fabric-client/lib on GitHub.

5.4.2 *SDK Application Development*

Previously we covered how to use the CLI command line to initiate the network and manipulate the chaincode. This section will present an example of marble asset transfer. The repository address can be found at IBM-Blockchain/marbles on GitHub.

Write the Code

marbles.go is the smart contract implementation chaincode for this application example; the code is as follows.

```
package main
import (
    ...
)
// Chaincode implementation
type SimpleChaincode struct { }
// Entity object definition, Marblesand Owners
// ---- Marbles Object---- //
type Marble struct {
    ObjectType string `json:"docType"` // for couchdb
    Id string `json:"id"` // id
    Color string `json:"color"` // marble color
    Size int `json:"size"` // marble size
    Owner OwnerRelation `json:"owner"` // Owner
}
// ---- Owners object ---- //
type Owner struct {
    ObjectType string `json:"docType"` // for couchdb
    Id string `json:"id"` // id
    Username string `json:"username"` // username
}
```

Table 5.2 Fabric-client interface

Command line parameters	Function
newChain(name)	Create a new chain
getChain(name)	Get the chain by name
newPeer(url, opts)	Create a new peer node
newOrderer(url, opts)	Create a new Orderer node
newMSP(msp_def)	Create a new member service
createChannel(request)	Create a channel
updateChannel(request)	Update a channel
queryChainInfo(name, peers)	Query chain information
queryChannels(peer)	Query the channels joined by peer nodes
queryInstalledChaincodes(peer)	Query the chaincode installed on peer nodes
installChaincode(request)	Install a chaincode
setStateStore(keyValueStore)	Set the state of the key-value store
saveUserToStateStore()	Save users to state
setUserContext(user, skipPersistence)	Set the user context
getUserContext(name, checkPersistence)	Get users context
loadUserFromStateStore(name)	Load users from state
getStateStore()	Get a state
buildTransactionID(nonce, userContext)	Build a transaction
createUser(opts)	Create users
setLogger(logger)	Logging
setMSPManager(msp_manager)	Set up member service management
getMSPManager()	Get the current member service management
addPeer(peer)	Add a peer
removePeer(peer)	Remove peer
getPeers()	Get the peer node collection
addOrderer(orderer)	Add an Orderer
removeOrderer(orderer)	Remove an Orderer
getGenesisBlock(request)	Get the genesis block
joinChannel(request)	Add a channel
queryBlockByHash(blockHash)	Query blocks
queryBlock(blockNumber)	Query blocks
queryTransaction(transactionID)	Query a transaction
queryInstantiatedChaincodes()	Query instantiated chaincodes
sendTransactionProposal(request)	Send a transaction proposal
sendTransaction(request)	Send transactions

```

    Company string `json:"company"` // user company
}
//Mables and holder relationship table, for query
type OwnerRelation struct {
    Id string `json:"id"`      // id
    Username string `json:"username"` // username
    Company string `json:"company"` // company
}

```

```
// Main method
func main() {
    err := shim.Start(new(SimpleChaincode))
    if err != nil { fmt.Printf("Error starting Simple chaincode - %s",
err) }
}
// Init method
func (t *SimpleChaincode) Init(stub shim.ChaincodeStubInterface) pb.
Response {
    fmt.Println("Marbles Is Starting Up")
    // Get parameter
    _, args := stub.GetFunctionAndParameters()
    var Aval int
    var err error
    if len(args) != 1 { return shim.Error("Incorrect number of arguments.
Expecting 1") }
    // Convert numeric to integer
    Aval, err = strconv.Atoi(args[0])
    if err != nil { return shim.Error("Expecting a numeric string argument
to Init()") }
    // Write state marbles_ui
    err = stub.PutState("marbles_ui", []byte("3.5.0"))
    if err != nil { return shim.Error(err.Error()) }
    // Start a test
    err = stub.PutState("selftest", []byte(strconv.Itoa(Aval)))
    if err != nil {
        // Test fails
        return shim.Error(err.Error())
    }
    // Test passes
    fmt.Println(" - ready for action")
    return shim.Success(nil)
}
// Invoke method
func (t *SimpleChaincode) Invoke(stub shim.ChaincodeStubInterface)
pb.Response {
    function, args := stub.GetFunctionAndParameters()
    fmt.Println(" ")
    fmt.Println("starting invoke, for - " + function)
    // Handling different method invoke
    if function == "init" {           // Initialize chaincode state
        return t.Init(stub)
    } else if function == "read" {      // Form readset
        return read(stub, args)
    } else if function == "write" {     // Form writeset
        return write(stub, args)
    } else if function == "delete_marble" { // Delete marbles from state
        return delete_marble(stub, args)
    } else if function == "init_marble" { // Create a new_marble
        return init_marble(stub, args)
    } else if function == "set_owner" {   // Change the owner of the marble
        return set_owner(stub, args)
    } else if function == "init_owner" { // Create a new owner of the
```

```

marble
    return init_owner(stub, args)
} else if function == "read_everything" {    // Read (owners+marbles
+companies)
    return read_everything(stub)
} else if function == "getHistory" {          // Retrieve the history
information of the marbles
    return getHistory(stub, args)
} else if function == "getMarblesByRange" {   // Read the set of marbles
    return getMarblesByRange(stub, args)
}
// Error
fmt.Println("Received unknown invoke function name - " + function)
return shim.Error("Received unknown invoke function name - '" +
function + "'")
}

```

This following code shows how to use the HFC client to interact with the Hyperledger blockchain.

```

enrollment.enroll = function (options, cb) {
    var chain = {};
    var client = null;
    try {
// Step 1 Create HFC client)
client = new HFC();
    chain = client.newChain(options.channel_id);
    }
    catch (e) {
    }
    if (!options.uuid) { ... }
    ...
// Step 2 Set up ECert kvs (Key Value Store)
HFC.newDefaultKeyValueStore({
// Store eCert in the kvs directory
    path: path.join(os.homedir(), '.hfc-key-store/' + options.uuid)
//store eCert in the kvs directory
}).then(function (store) {
    client.setStateStore(store);
// Step 3
    return getSubmitter(client, options);
}).then(function (submitter) {
// Step 4
    chain.addOrderer(new Orderer(options.orderer_url, {
        pem: options.orderer_tls_opts.pem,
        'ssl-target-name-override': options.orderer_tls_opts.
common_name //can be null if cert matches hostname
    }));
// Step 5
    try {
        for (var i in options.peer_urls) {
            // Create a new peer node
            chain.addPeer(new Peer(options.peer_urls[i], {

```

```

        pem: options.peer_tls_opts.pem,
        'ssl-target-name-override': options.peer_tls_opts.
common_name
            })) ; logger.debug('added peer', options.peer_urls[i]);
        }
    }
    catch (e) { }
    ...
// Step 6
// Print Log
    logger.debug('[fcw] Successfully got enrollment ' + options.uuid);
    if (cb) cb(null, { chain: chain, submitter: submitter });
    return;
}).catch(
    function (err) { ... return; }
);
};
}
;

```

The SDK calling process is as follows.

- **Step 1:** Create an SDK instance.
- **Step 2:** Create a key-value store to store the registration certificate by newDefaultKeyValueStore.
- **Step 3:** Register users. Meanwhile, you have to use the registration ID and level key to get the authentication from CA. CA will issue the registration certificate that SDK will store in the key-value store. If you adopt the default key-value library, it will be stored in the local file system.
- **Step 4:** After successful registration, set the orderer URL. Orderer is not needed yet but will be needed when calling the contract code. ssl-target-name-override is only needed if you have signed the certificate. Set this field to be the same as the common name of the PEM file you created before.
- **Step 5:** Set up Peer's nodes. These are not needed for now, but you need to set up the chain object of the SDK.
- **Step 6:** The SDK is fully configured and ready to interact with the blockchain.

Application Operation

The application interaction flow of Marbles is shown in Fig. 5.5.

1. The browser communicates with the Node.js application via a Websocket service.
2. Interaction between Node.js application and Hyperledger blockchain network is via FabricNodeSDK (i.e., HFC).
3. The communication between HFC and CA institutions is established on HTTP protocol.
4. HFC acts as a client node in the Hyperledger blockchain network, and the nodes in the blockchain network communicate with each other on the gRPC protocol.

The steps for application are as follows

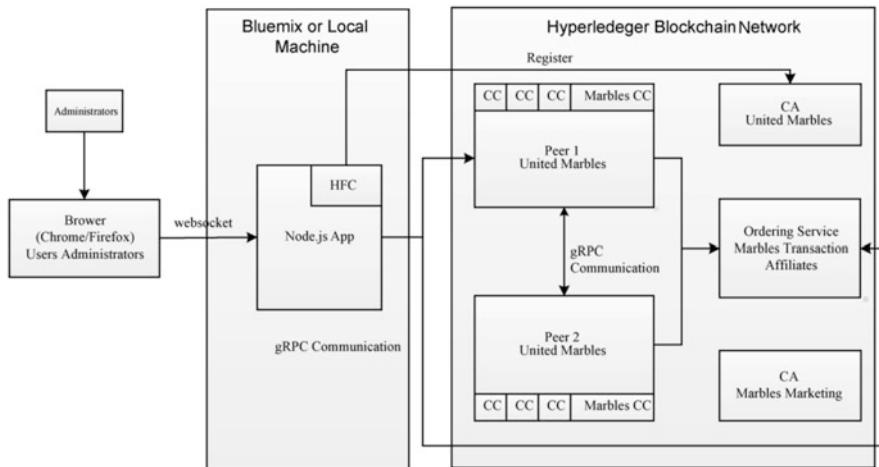


Fig. 5.5 Application interaction flow

- **Step 1:** Refer to the steps in Example 1 to configure the network and start the network.
- **Step 2:** Configure the JSON files: /config/marbles1.json and /config/blockchain_creds1.json.
- **Step 3:** Install and instantiate the chaincode with path /chaincode/src/marbles.
- **Step 4:** Start the application.

```
npm install gulp -g
npm install
gulp
# After success, you will see the following display result
----- Server Up - localhost:3000 -----
```

Once done, you can access it by typing localhost:3000 in your browser.

5.5 Summary

This chapter introduces how to develop blockchain applications on the Hyperledger Fabric platform. Firstly, it describes the process of setting up the Hyperledger Fabric development runtime environment; secondly, it explains the chaincode development and deployment process, and finally, it explains the CLI application interface and SDK interface and illustrates how to develop Hyperledger Fabric blockchain applications based on these two interfaces through examples.

Part III

The Enterprise-Level Blockchain Platform

Hyperchain

Chapter 6

Anatomy of the Core: Principles of Enterprise-Level Blockchain Platform



Enterprise blockchain, also known as federated blockchain, provides enterprise-level blockchain network solutions to address the blockchain technology requirements of large companies, government agencies, and industrial alliances. Each node of a federated chain usually corresponds to an organization of entities, and nodes must be authorized to join and withdraw. The various organizations form a coalition of stakeholders to maintain the blockchain network health.

Unlike private and public chains, enterprise-level blockchain focuses more on the practical implementation of blockchain technology. There are higher requirements in blockchain performance speed and security, member authentication management, and data privacy protection. In addition, the development of enterprise-level blockchain is often directly related to actual business scenarios, closer to the industry's pain points, and provides a complete all-in-one blockchain solution for enterprise alliances. Figure 6.1 illustrates the mutually reinforcing relationship between a federated chain platform and blockchain applications. On the one hand, the federated chain platform supplies the underlying technical support for the development and implementation of practical industry applications; on the other hand, the verification and implementation of industry applications and concepts push forward the continuous development and maturity of the alliance chain platform.

As a domestic enterprise-level blockchain service platform, Hyperchain processes enterprise-level blockchain network solutions for enterprises, government agencies, and industrial alliances with blockchain technology needs. This chapter will take Hyperchain as an example to illustrate the core principles of enterprise-level blockchain platform design.

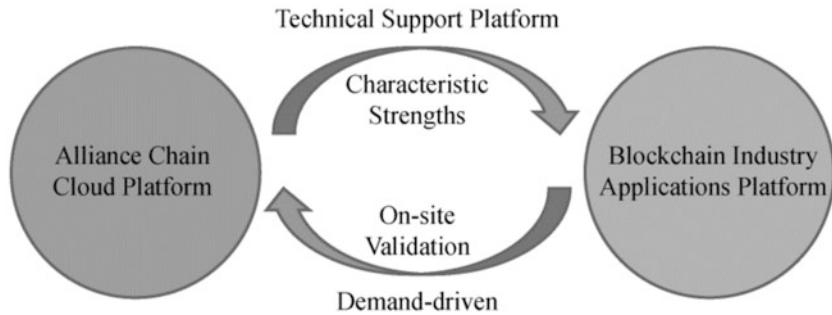


Fig. 6.1 Relationship between federated chain cloud platforms and industry applications

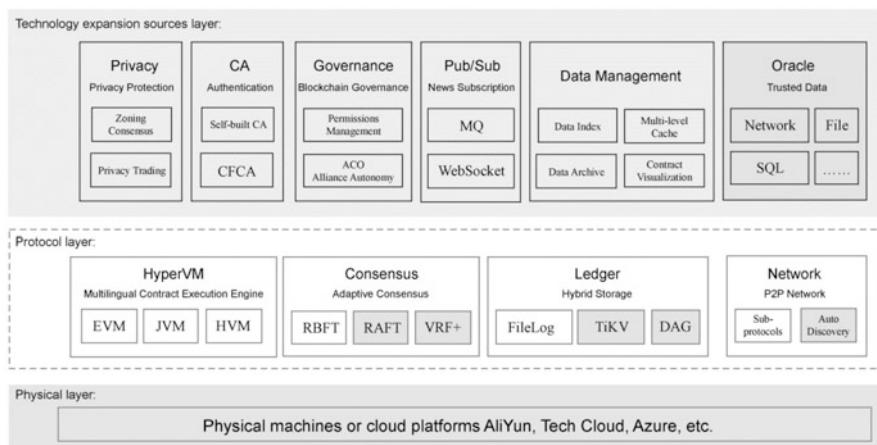


Fig. 6.2 Hyperchain system architecture diagram

6.1 Hyperchain Overall Architecture

Hyperchain supports enterprises to rapidly deploy, extend, configure, and manage blockchain networks established on existing cloud platforms, conducts real-time visual monitoring of the operation status of blockchain networks, and is the core system platform of domestic blockchain that conforms to ChinaLedger technical specifications. Hyperchain has core features such as authentication node authorization mechanism, multilevel encryption mechanism, consensus mechanism, and Turing-complete high-performance smart contract execution engine. It is a virtual technology platform of alliance chain with perfect functions and high performance. The overall system architecture of Hyperchain is displayed in Fig. 6.2.

The platform has the following features for the participant membership authentication permission mechanism:

- Membership authentication licensing mechanism for participants.
- Security and privacy of business transaction data.
- Higher transaction throughput and lower transaction latency.
- Secure and complete smart contract engine.
- High user experience interoperability.

For the participant membership authentication permission mechanism, the platform has the following features.

- **Coalition Autonomy ACO.** The platform permits the creation of autonomous member organizations of the coalition chain in the coalition chain network to submit and organize internal voting on state actions in the coalition, such as system upgrades, contract upgrades, and member management, through proposals. The approach supplies an effective model for blockchain coalition governance.
- **Member Management.** The platform realizes the access control of alliance members through CA system authentication, supports two modes (self-built CA and CFCA), and provides hierarchical authority management mechanisms for chain-level administrator, node administrator, and ordinary users, so as to implement different access controls for permissions.

For the security and privacy of transaction data, the platform has the following features.

- **Multilevel Encryption Mechanism.** Adopting pluggable encryption mechanism, different policies are encrypted for the data, users, and communication connections involved in the complete life cycle of the business, ensuring the security of the platform data through multilevel encryption, and fully supporting the national security algorithm.
- **Privacy Protection.** The platform provides both Namespace partition consensus and privacy transaction mechanisms to achieve privacy protection. The partitioning consensus isolates sensitive transaction data storage and execution space, allowing some blockchain nodes to create partitions of their own. Data transactions between partition members and the repository are not visible to nodes in other partitions. The privacy transaction, on the other hand, is done by specifying the related party of that transaction at the time of sending, and the transaction details are only stored in the related party. The private transaction hash is stored after the network-wide consensus, ensuring the effective isolation of the privacy data and verifying the authenticity of that privacy transaction.
- **Trusted Data Source.** The blockchain is a closed deterministic environment, and the chain cannot actively obtain data from the real world outside the chain. The platform introduces the Oracle prophecy machine mechanism, which supports writing outside information to complete data interoperability between the blockchain and the real world. The prophecy machine realizes trust endorsement through the signature of third-party trusted institutions to meet the requirement of provable honesty request.

For throughput as well as transaction latency, the platform has the following features.

- **Efficient Consensus Algorithm.** The platform adopts the RBFT (Robust Byzantine Fault-Tolerant) consensus algorithm, which improves the overall transaction throughput capacity and stability while ensuring the strong consistency of node data. At the same time, the platform can employ GPU-based signature verification acceleration to improve the overall performance further, fully meet the needs of blockchain business applications, and support dynamic node management and failure recovery mechanisms, enhancing the fault tolerance and availability of the consensus module. Other consensus algorithms (e.g., RAFT) will be integrated later to meet the needs of different business scenarios.
- **Data Separation.** Blockchain ledger data is mainly divided into two parts: block data and state data. Considering that block data will keep growing while state data will only be updated frequently, the platform introduces the Filelog storage engine to realize the separation of block data and state data to ensure that the read and write performance will not be affected when the system data volume keeps increasing.

For a secure and complete smart contract engine, the platform has the following features.

- The platform supports various smart contract engines such as EVM, JVM, and HVM;
- Support programming languages such as Solidity and Java.
- Provide sound contract lifecycle management.
- With programming-friendly, contract-safe, and efficient execution features, it can adapt to changing and complex business scenarios.

For high user experience interoperability, the platform has the following features.

- **Data Archiving.** To solve the infinite growth of blockchain storage data in the blockchain, we move part of the old online data archive to the offline dump by data archiving and provide Archive Reader for archived data browsing.
- **Data Visualization.** To facilitate real-time access to contract status data on the blockchain, the platform provides a data visualization component Radar, which can import contract status data from the blockchain into relational data while the blockchain is running normally library (e.g., MySQL), making the contract status visual and monitorable for business statistics and analysis of commercial applications.
- **Message Subscription.** The platform provides a unified message subscription interface so that external systems can capture and listen to changes in the state of the blockchain platform thus enabling the interoperability of on-chain and off-chain messages and supporting the subscription of events such as block events, contract events, transaction events, and system exception monitoring and control.

We will take Hyperchain as an example and explain the core technology modules that constitute an enterprise-level blockchain platform, generally analyzing the implementation principles of consensus algorithms, smart contracts, ledgers, security mechanisms, and data management.

6.2 Basic Components

This section details the fundamental components of an enterprise-level blockchain platform, including consensus algorithms, network communication, smart contracts, and ledger data storage mechanisms. We will describe each of them below.

6.2.1 *Consensual Algorithm*

The consensus algorithm is the key to ensure the consistency of the ledger data of each node in the blockchain platform. The common consistency algorithms of distributed systems include PoW, PoS, Paxos, Raft, PBFT, and so on. PoW relies on the computing power of machines to obtain the bookkeeping rights of the ledger, more resource-consuming and less supervisory, and the consensus of each transaction needs to be calculated by the whole network; thus, it is not suitable for the supervision and performance requirements of the federated chain. Paxos and Raft are consistent and mature solutions for traditional distributed systems, having high performance and low resource consumption but no Byzantine fault tolerance. The algorithm tolerates Byzantine errors and allows for the participation of strongly supervised nodes. The mainstream enterprise blockchain solutions, Fabric and Hyperchain, both offer PBFT implementations. However, the native PBFT algorithm is not perfect in terms of reliability and flexibility. The Hyperchain platform enhances the reliability and flexibility by designing and implementing an improved algorithm of PBFT, namely RBFT.

RBFT Overview

Hyperchain's consensus module adopts a pluggable modular design and can be configured with different consensus algorithms for different business scenarios. RBFT can control the latency of transactions within 300 ms and support up to tens of thousands of transactions per second, providing a stable and high-performance algorithm for blockchain commercial applications. The core algorithm of RBFT is described in detail below.

RBFT Routine Process

RBFT's regular process ensures that each node of the blockchain processes transactions from clients in the same order. RBFT has the same fault tolerance as PBFT and requires at least $3f + 1$ node to tolerate f Byzantine errors. The example in Fig. 6.3 shows the minimum number of cluster nodes with a value of f being 1. The Primary node in the figure is the dynamically elected primary node of the blockchain nodes, for sorting and packaging client messages, the Replica node is the backup node, and all Replica nodes perform transactions with the same logic as the Primary node. Replica nodes can participate in the election of new Primary nodes when the Primary node fails.

RBFT's consensus retains the original three-stage processing flow of PBFT (PrePrepare, Prepare, Commit), but interspersed with an important transaction validation component.

The regular consensus flow of the RBFT algorithm is shown below:

1. Client sends the transaction to any node in the blockchain.
2. Replica node receives the transaction and forwards it to the Primary node, and the Primary itself can receive the transaction message directly.
3. Primary packages the received transactions, generates a batch for validation, and rejects any illegal transactions.
4. Primary broadcasts the validated batch construct PrePrepare message to other nodes, here only the hash of the batch transaction is broadcast.
5. Replica receives the PrePrepare message from the Primary and constructs the Prepare message to send to other Replica sections, indicating that the node has received the PrePrepare message from the master node and acknowledged the batch sorting of the master node.

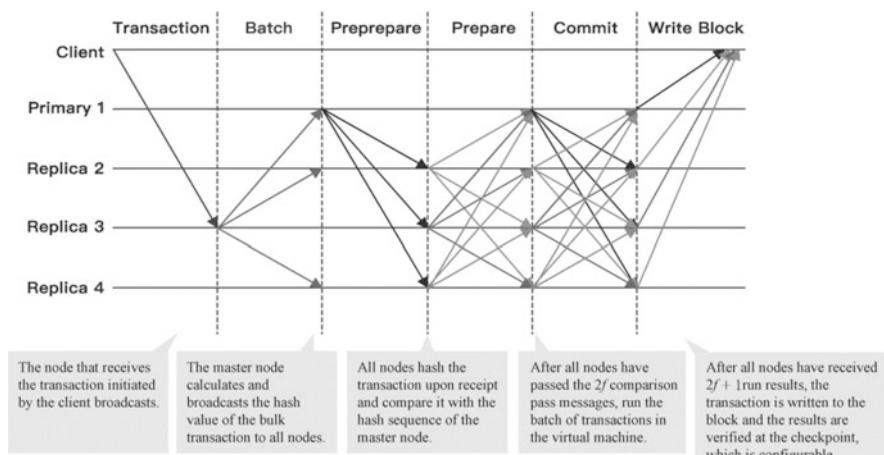


Fig. 6.3 RBFT regular consensus flow

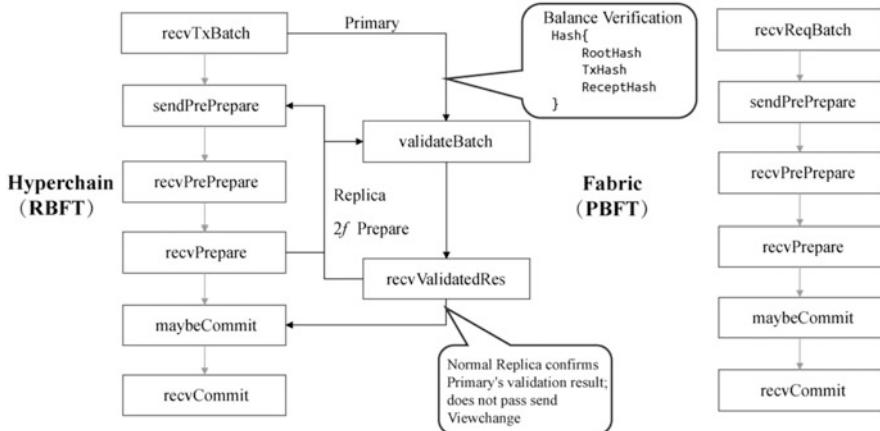


Fig. 6.4 RBFT process versus PBFT process

6. Replica receives the Prepare message from the $2f$ node and verifies the legitimacy of the batch message. After verification, it broadcasts a Commit message to the other nodes indicating agreement with the Primary node's verification result.
7. The Replica node receives $2f + 1$ Commits and then executes the transactions in the batch and verifies them with the execution result of the master node.

As can be seen from the above RBFT routine flow, RBFT intersperses the validation process of transactions throughout the consensus algorithm to achieve consensus on the results of writing blocks. First, Primary node first performs validation after receiving a transaction, which ensures that the platform's arithmetic power is not consumed by illegal transactions and enables the Replica node to efficiently handle Byzantine failures of the Primary node. Second, Replica node verifies the validation result of the Primary node after receiving $2f$ Prepare messages, and triggers a ViewChange message if the result fails, which again ensures the security of the system. Figure 6.4 shows the specific flow of the RBFT consensus process compared with the traditional PBFT algorithm verification.

RBFT View Replacement

In the PBFT algorithm, the nodes involved in the consensus can be divided into Primary and Replica nodes according to their roles. Replica will forward the transactions they receive to the master node, and its most important function is to pack all the received transactions into blocks according to a certain policy and let all nodes participate in consensus verification. Then, a natural question comes out: if the master node is down, has a system error or is captured (i.e., becomes a Byzantine node), how can the other Replica nodes discover the master node's anomaly in time

and elect a new master node to continue the consensus? This is a problem that must be solved to ensure the stability of BFT-like algorithms.

The concept of View is introduced in both PBFT and RBFT, and the ViewChange mechanism is the key to ensure the robustness of the entire consensus algorithm by switching the view every time a master node is replaced.

There are currently three scenarios of Byzantine behavior of primary nodes that can be detected: (1) the node stops working and does not send any more messages; (2) the node sends an incorrect message; the error could be incorrect message content, a message containing a malicious transaction, etc. It should be noted that the message type here could be batch or a functional message for view replacement; (3) the node disguises itself as a regular node and sends the right message.

For scenario (1), it can be guaranteed by the nullRequest mechanism that the primary node with the correct behavior will send a nullRequest to all slave nodes to indicate the truth of this situation when no transaction occurs. If the slave node does not receive a nullRequest from the primary node within the specified time, it will trigger the view replacement behavior to elect a new primary node.

For scenario (2), when the slave node receives a message from the primary node, it detects the content through the validation mechanism to determine the content accordingly. Suppose it finds that the primary node's transaction contains a transaction that does not conform to the corresponding format or a malicious transaction, i.e., the validation does not pass. In that case, it will initiate a view replacement and elect a new primary node.

For scenario (3), there is no need to consider that; an extreme case is, if a Byzantine node has been behaving like a normal node, it can be assumed not to be a Byzantine node. In this way, the correctness of the result is guaranteed by the whole system.

A slave node will broadcast a view replacement message to the entire network after detecting the above exception in the primary node or receiving a view replacement message from other $f + 1$ nodes. When the new primary node receives $N-f$ view replacement messages, it sends a NewView message. After receiving the NewView message, the Replica node verifies and compares the notes. After ascertaining that the view switching information is the same, it officially replaces the view and prints the FinishVC message thus completing the whole view replacement process, as shown in Fig. 6.5 (where ViewChange represents view replacement, Primary represents a primary node, and Replica represents a slave node).

RBFT Auto-Recovery

The blockchain network may cause some nodes to lag behind most nodes in execution or go down outright due to network jitter, sudden power outages, disk failures, etc., during operation. In such scenarios, the nodes need to automatically recover and synchronize the ledger to the latest ledger state to participate in subsequent transaction execution. To address this type of data recovery RBFT algorithm has a dynamic data auto-recovery mechanism.

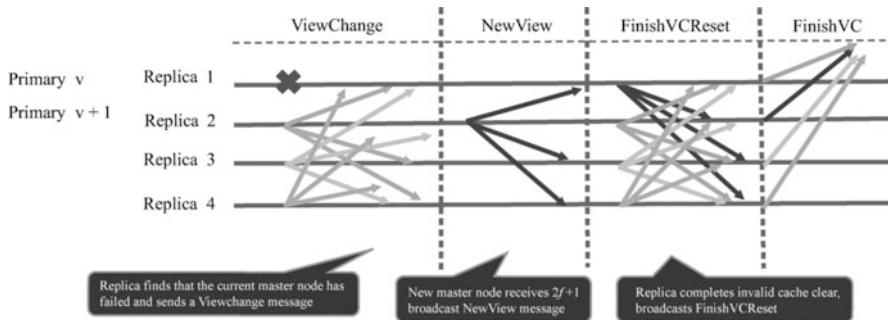


Fig. 6.5 Schematic diagram of RBFT view replacement



Fig. 6.6 Schematic diagram of RBFT checkpoints

RBFT's auto-recovery mechanism aligns the storage of its nodes with the latest storage state in the system as soon as possible by actively requesting information about blocks and blocks under consensus. The auto-recovery mechanism greatly enhances the availability of the entire blockchain system. RBFT sets checkpoints on the executed data for the convenience of recovery, and the checkpoints result from a network-wide consensus. It ensures that the data before the checkpoint on each node is consistent. In addition to the checkpoints, there is a portion of the data stored that is the current progress of the local execution not yet consensus. It requires during the recovery process, firstly the node's checkpoints are synchronized with the checkpoints of the other normally serving nodes. Secondly, it needs to recover some of the data outside the checkpoint. Figure 6.6 shows a schematic diagram of checkpointing, with the checkpointing portion on the left and the portion outside the currently executing checkpoint on the right. The basic processing flow of the automatic recovery mechanism is shown in Fig. 6.7.

RBFT Node Addition and Deletion

In an alliance chain scenario, due to its expansion or the withdrawal of some members, the chain supports the dynamic entry and exit of members from the service. In contrast, the traditional PBFT algorithm does not support dynamic addition and deletion. RBFT can dynamically add and delete nodes to PBFT while keeping the cluster nonstop to control the entry and exit more easily. As shown in Fig. 6.8, RBFT adds an algorithmic processing flow for new nodes.

First, the new node asks to get a certificate from the certificate authority and then sends a request to all the nodes in the alliance. Each confirms its consent and then broadcasts it network-wide to other nodes within, and when a node gets $2f+1$ replies

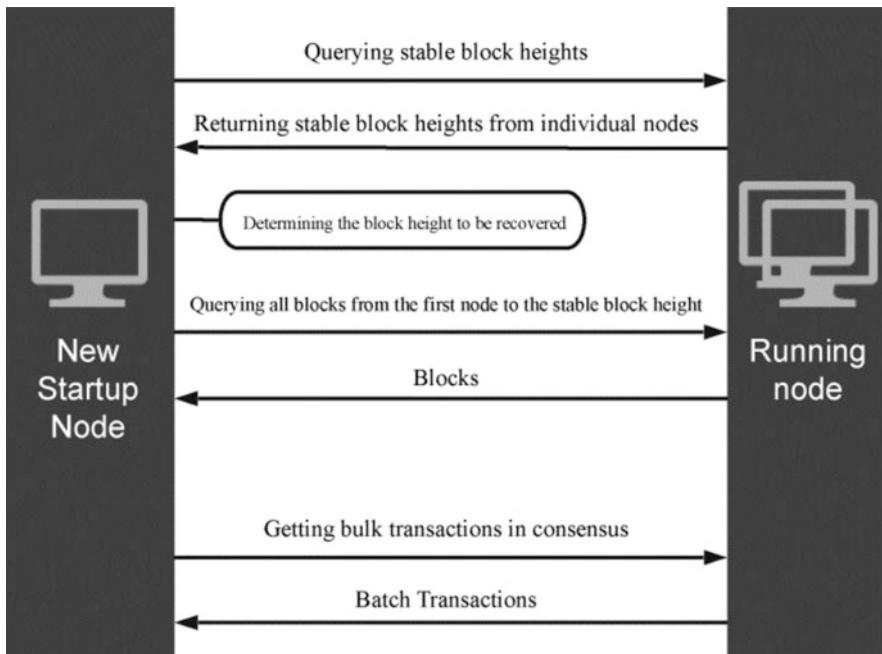


Fig. 6.7 RBFT auto-recovery process

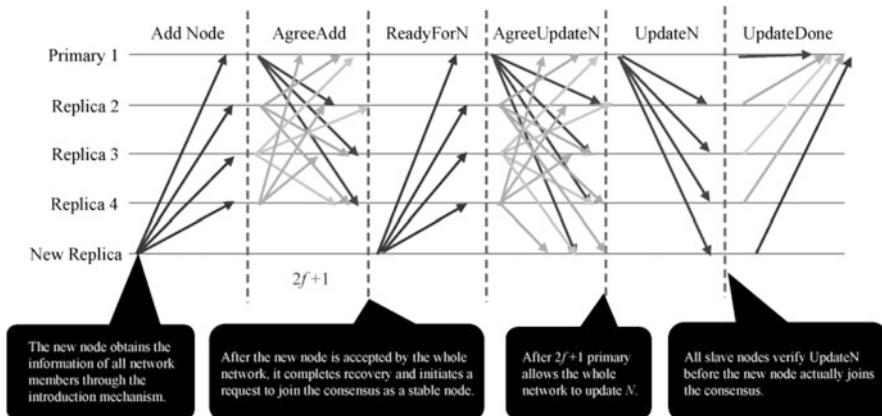


Fig. 6.8 RBFT node addition and deletion

agreeing to join, it establishes a connection with the new ones. Second, once the new node establishes a connection with $N-f$ (N is the total number of blockchain federation nodes), nodes can perform an active recovery algorithm to synchronize the latest members' status. Again, the new node then requests the primary node to join the normal consensus process. Finally, the primary node, after acknowledging the

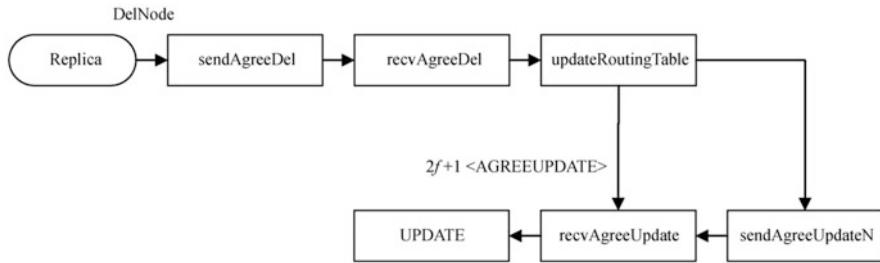


Fig. 6.9 Dynamic node exit function call

new node's request, defines at which block number the total number of nodes N needs to be changed for consensus (ensuring that the joining of a new node does not affect the original consensus since the union of a new one leads to a change in the network-wide consensus N , indicating that the f value may change).

The dynamic deletion of RBFT nodes is similar to the dynamic addition nodes process, and its main processing functions are shown in Fig. 6.9. The process is as follows.

1. The exiting node needs to get the hash value of this node by invoking an RPC request and then initiating an exit to all nodes in the network request.
2. The administrator of the node that received the deletion request confirms that it agrees to the node's withdrawal and then broadcasts a DelNode message to the entire network, indicating that it conforms to the node's request to withdraw from the whole blockchain consensus.
3. When an existing node receives $2f + 1$ DelNode messages, the node updates the connection information, disconnects from the node requesting exit, and broadcasts an AgreeUpdateN message to the entire network after disconnection, indicating a request for a system-wide suspension of the transaction processing behavior in preparation for updating the N and view of the entire system participating in the consensus.
4. When the node receives $2f + 1$ AgreeUpdateN messages, it updates the node system state.

At this point, the requesting exit node officially exits the blockchain system.

The above is the main algorithm flow of Hyperchain's improved consensus algorithm RBFT, which is more stable, flexible, and efficient than the traditional PBFT algorithm by adding verification steps to the regular consensus process, adding automatic node recovery mechanisms, and adding dynamic node addition and deletion functions, which can better meet the needs of production environments of enterprise-level federated chains.

6.2.2 Network Communication

Communication Principles

A P2P network is designed as a channel for nodes to reach consensus and transfer information and is the basis of Hyperchain. The network communication module mainly comprises node, peer, and encrypted transmission. The node point module is primarily employed to provide the gRPC call service and exists as the server-side. Peer submodules are used as clients when the local node requests other nodes. The encryption module uses the ECDH key negotiation algorithm to generate a key recognized only between two nodes and encrypts data transmitted between nodes based on enhanced AES symmetry to ensure data transmission security.

The whole communication process is shown in Fig. 6.10.

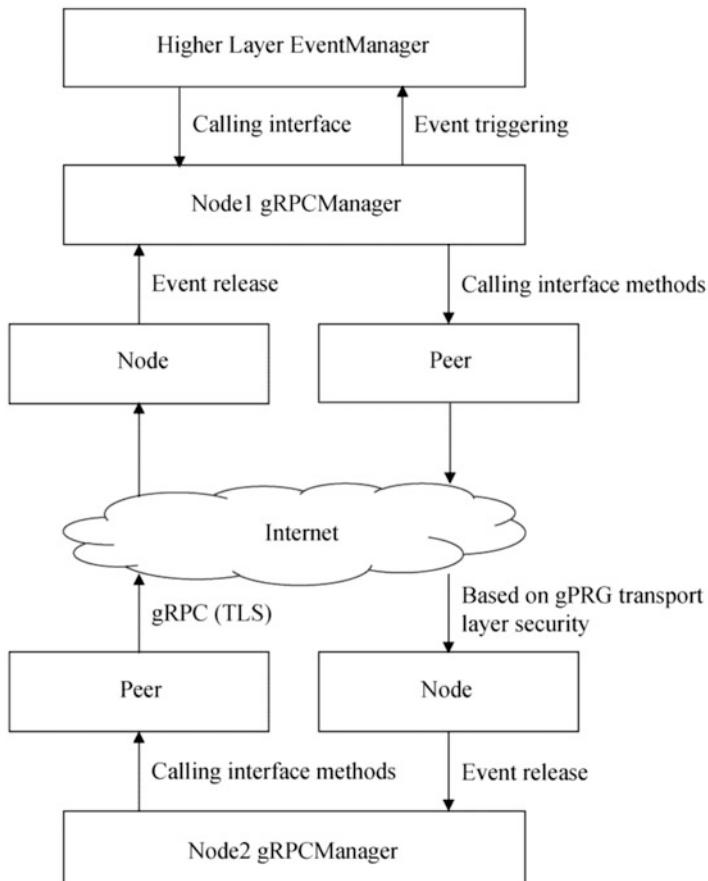


Fig. 6.10 Hyperchain communication flowchart

The leading architecture of Hyperchain is designed to separate Peer and Node, with Peer offering the message sending interface for the upper layer modules. Both Peer and Node are managed through the gRPCManager, which controls the communication and distribution of the layers, exposing the Peer's interface to the outside and holding each state of the node. The modules are also separated in the P2P module and controlled by a control layer, with submodules each doing their respective job.

Node Type

Hyperchain's nodes are divided into two categories: verified nodes (VPs) and non-verified nodes (NVPs).

- A validation node means a node in the blockchain network involved in consensus validation.
- Non-verifying nodes are not in consensus verification in the blockchain network, but only in bookkeeping.

NVPs are primarily designed for transaction forwarding and disaster recovery, and do not process transactions themselves or participate in consensus; it thus relies on connected VPs to ensure eventual consistency with the network-wide state. However, NVPs can receive and forward the received transactions to the connected VPs for processing.

VPs do not vigorously connect to NVPs, when VPs restart, all the NVPs connected to it will be disconnected and will not be reconnected automatically, and need to be connected manually. The NVPs have a perfect state recovery mechanism, which can synchronize in time after the state has just started or fallen behind due to other reasons.

VPs communicate via the gRPC remote call service to form a P2P network, where the gRPC service adopts protobuf3 to serialize and de-serialize line data to ensure data integrity and efficient and secure transmission.

Flow Control Mechanism

According to business needs, the underlying platform can artificially control the traffic allowed into the blockchain system. When the system traffic exceeds the upper limit, it will refuse to receive the exceeding part. This prevents the network communication process from delaying other transactions due to many useless transaction requests taking up the node processing time thus ensuring the system security while meeting the business requirements.

On-demand traffic can be configured for contract transactions and general transactions through configuration files. The configuration items and configuration information are shown in Table 6.1.

Table 6.1 Configuration file table

Parameters	Description
global.configs.ratelimit.enable	Traffic control on or off (generally on)
global.configs.ratelimit.txRatePeak	General transaction flow control
global.configs.ratelimit.txFillRate	Common transaction traffic threshold recovery interval
global.configs.ratelimit.contractRatePeak	Contract transaction flow control
global.configs.ratelimit.contractFillRate	Contract transaction flow threshold recovery interval

6.2.3 Smart Contracts

A smart contract is an automatically executable program deployed on the blockchain. In a broad sense, it includes programming languages, compilers, virtual machines, events, state machines, fault tolerance mechanisms, etc. The programming language and the execution engine of the smart contract, the virtual machine, significantly impact application development. The virtual machine is encapsulated as a sandbox, and the entire execution environment is entirely isolated. Smart contracts executing inside the virtual machine do not have access to system resources such as the network, file system, or other threads in the system. Only limited calls can be made between contracts.

There are three typical examples of smart contracts' implementations and runtime environments.

1. IBM's Hyperledger Fabric project employs Docker as the execution environment for smart contracts.
2. Smart contracts in the R3 Corda project employ the JVM as the underlying execution environment for the contracts.
3. The smart contracts in the Ethereum project are written in Solidity and use an embedded Solidity virtual machine for implementation.

Smart Contract Execution Engine

Because a smart contract is essentially an automatically executable script program, there is the potential for errors, even serious ones, to be triggered problems or chain reactions, and hence the security of the smart contract execution engine is critical to the security of the enterprise blockchain.

Solidity is a high-level programming language designed for writing smart contracts, with a syntax similar to JavaScript. It is straightforward to write, a Turing-complete language. More importantly, it can only be used to implement the logical functions of the contract and does not provide any interface to system resources (e.g., opening files, accessing underlying OS resources, etc.). This ensures at the language level that smart contracts written in Solidity can only run in an OS-independent

sandbox and cannot manipulate any system resources. On the other hand, Fabrics are based on a Docker form of the virtual machine with no particular restrictions on the language and therefore cannot fully guarantee security.

Compared to Docker and JVM, the Solidity language and its smart contract execution engine are smaller in program size and have the finer granularity of control over resources. The Solidity language maximizes the reusability of smart contracts by leveraging the open-source community's accumulated smart contract technology and experience. Therefore, the Hyperchain platform chose the Solidity language for its smart contract implementation and designed and developed HyperVM, an efficient smart contract execution engine that supports Solidity execution.

HyperVM is Hyperchain's pluggable smart contract engine generic framework, allowing access to different smart contract execution engines. It currently implements HyperEVM, which is compatible with the Solidity language, and HyperJVM and HVM, smart contract engines that support the Java language, and will continue to integrate with other virtual machines such as WVM and JSVM.

HyperEVM

HyperEVM is a deep refactoring of the EVM virtual machine to make maximum use of the open-source community's accumulation of smart contract technology and experience and to improve their reusability, fully compatible with smart contracts developed on EVM. HyperEVM optimizes the performance of the smart contract virtual machine while maintaining the compatibility of the Solidity development language. It maintains the sandbox security model of the EVM, makes full fault tolerance mechanisms, and performs system-level optimizations.

HyperJVM

HyperJVM provides a high-performance and secure execution sandbox for native Java smart contract execution through a microservices architectural design and multiple security checks. HyperJVM offers the following benefits:

- Support Java language for smart contract development, which significantly lowers the development threshold;
- Support complete smart contract lifecycle management, including contract deployment, upgrades, freezing, etc.;
- Support rich book operations, KV interfaces, batch processing, range queries, and columnar data operations;
- Support complex contract logic development and authorized cross-contract calls;
- Support contract custom event listening.

HVM

HVM (Hyperchain virtual machine) is a lightweight Java smart contract runtime integrated into Hyperchain. It provides a sandbox environment to execute smart contracts written in the Java language and secure them in various ways. On HVM, users can efficiently write simple and powerful smart contracts. HVM has the following advantages:

- Well-established contract lifecycle support;
- A more secure environment for executing smart contracts in Java;
- A more efficient mechanism for manipulating the state space;
- A more user-friendly programming interface solution.

HyperVM Design Principles

The design of HyperVM is displayed in Fig. 6.11. The main components include a compiler for contract compilation, an optimizer for code execution optimization, an interpreter for contract bytecode execution, a security module for security control of the contract execution engine, and a state management module for interaction between the virtual machine and the ledger.

HyperVM Execution Flow

Figure 6.12 shows a typical flowchart of HyperVM executing a transaction. After HyperVM executes a transaction, it returns a result, which the system saves in a variable called a transaction receipt, after which the platform client can query the transaction result on the hash of this transaction.

The specific execution process of HyperVM is as follows.

1. HyperVM receives the transaction passed from the upper layer and performs the initial validation.

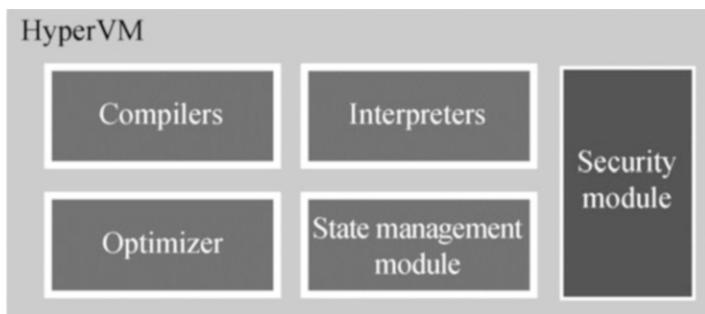
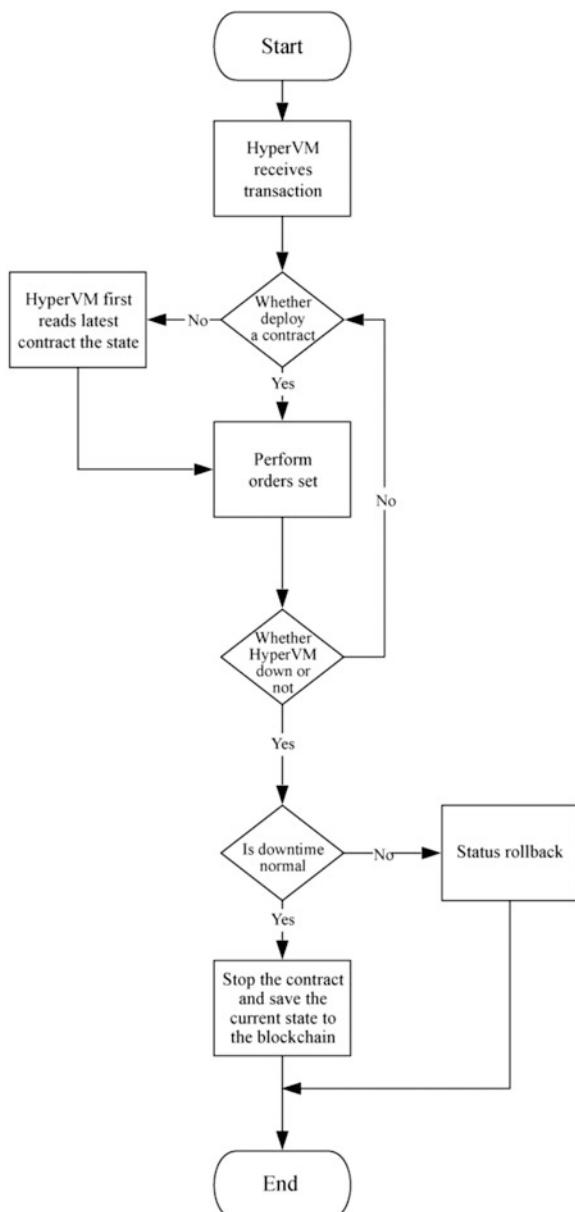


Fig. 6.11 HyperVM component diagram

Fig. 6.12 HyperVM execution flowchart



2. Determine the type of transaction; execute step (3) for a deployment contract or otherwise perform step (4).
3. HyperVM creates a new contract account to store the address and the code after the contract has been compiled.

4. HyperVM parses the transaction parameters and other information in the transaction and calls its execution engine to execute the corresponding smart contract byte code.
5. After the command is executed, HyperVM will determine whether it is down or not; skip to step (2) if it is not; otherwise run step (6).
6. Determine if the downtime status of HyperVM is normal, and end the execution if it is normal; otherwise, execute step (7).
7. Perform Undo operation; the status is rolled back to before the execution of the transaction, and the transaction ends.

The execution instruction set module in Fig. 6.12, which is the core of the HyperVM execution module, can be implemented in two ways: bytecode-based execution and the more complex and efficient just-in-time compilation (JIT).

Bytecode execution is relatively simple, and the HyperVM implementation of the VM will have an instruction execution unit. This unit will always try to execute the instruction set; when execution does not complete by a specified time, the VM will interrupt the computational logic and return a timeout error message thus preventing malicious code execution in the smart contract.

The JIT method of execution is relatively complex. JIT compilation, also known as real-time compilation, is both a form of dynamic compilation and is a way to improve the efficiency of running programs. There are two ways to run a program: static compilation and dynamic direct translation. The former means that the program is wholly translated into a machine code before execution, while the latter is translated and executed simultaneously. The just-in-time compiler mixes static and dynamic direct translation, compiling the source code one sentence at a time, but caching the translated code at the same time, which has the advantage of reducing performance loss. Just-in-time compiled code handles latency binding and enhances security compared to statically compiled code. JIT mode execution of smart contracts consists of the following main steps:

1. All the smart contract-related information is encapsulated in the contract object, and then the code hash is adopted to figure out if the contract object has been stored and compiled. There are four normal common states of the contract object, i.e., contract unknown, contract compiled, contract ready to execute via JIT, and contract error;
2. If the contract status is that the contract is ready to be executed via JIT, HyperVM will select the JIT executor to execute the contract. During execution, the VM will further compile the compiled smart contract into machine code and perform deep optimization of instructions such as push, and jump;
3. If the contract state is contract unknown, HyperVM first needs to check if the VM forces JIT execution, and, if so, compiles sequentially and executes via the JIT instructions. Otherwise, a separate thread is opened for compilation and the current program is still compiled via normal bytecode. When a contract with the same encoding is encountered again during the next virtual machine execution, the virtual machine directly selects the optimized contract. This allows the

contract to be executed and deployed more efficiently due to its optimized instruction set.

6.2.4 *Ledger Data Storage Mechanism*

Blockchain is essentially a distributed ledger system, so the design of the ledger system of the blockchain platform is crucial. Hyperchain's ledger design consists of three main parts. Firstly, customer transaction information is stored through a chain structure like blockchain to ensure customer transactions are not easily tampered with and are traceable. Secondly, the account system model is used to maintain the state of the blockchain system, i.e., the contract state part in Fig. 6.13. Finally, to quickly determine whether key information such as ledger information and transaction information exists, the ledger uses a modified version of Merkle tree to store relevant information.

This section follows with a detailed analysis of the design of these critical data structures related to ledgers.

Blockchain

The blockchain is the fundamental data structure in the blockchain ledger that stores core transaction information. It is a data structure consisting of blocks containing transaction information linked orderly from back to front. All blocks are linked in this chain in an orderly manner from back to front, with each block pointing to its parent block. The blockchain is often thought as a vertical stack, with the first block serving as the first block at the bottom, and each subsequent block being placed on top of the others. After visualizing the concept of sequential block linking with a stack, we can use terms such as “height” for the distance between the newest block and the first block, “top” or “top part” denotes the most recently added block.

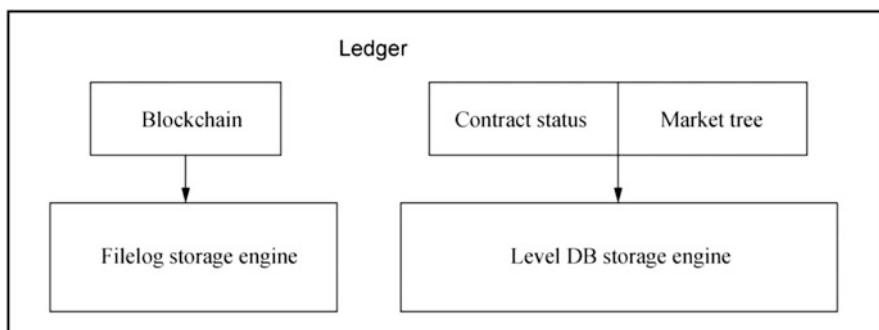


Fig. 6.13 Ledger storage structure

Fig. 6.14 Blockchain structure

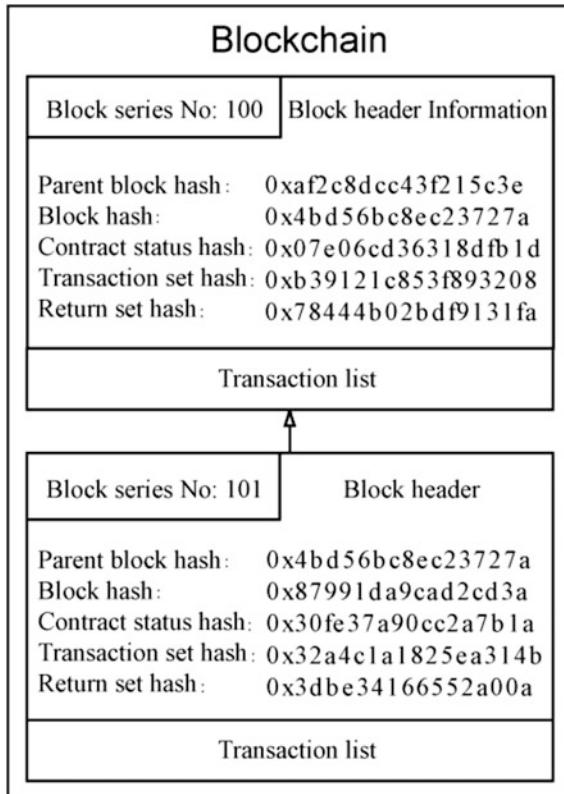


Table 6.2 Hyperchain block definitions

Field name	Description	Size
Block header	Block metadata set	203 bytes
Transaction list	Information on transactions recorded in the block	Variable

As shown in Fig. 6.14, there are two parts in the block structure: the block header and the transaction list. Some fixed-size block metadata information is recorded in the header, and all the transactions included are recorded in the transaction list. The specific definitions of the corresponding stored contents are shown in Tables 6.2, 6.3, and 6.4.

A SHA256 hash calculation of each block header generates a hash value that can be used as a digital fingerprint that uniquely identifies the block in the blockchain. Also, the hash value of the previously generated block is referenced in the block header information, i.e., in each block, the hash value of its parent block is included. In this way, all blocks are concatenated into a vertical chain structure that can eventually be traced back to the creation block (the first block) of the blockchain by continuously iterating through accessing the parent blocks.

Table 6.3 Hyperchain block header definitions

Field name	Description	Size (bytes)
Version information	Block structure definition version information	3
Parent block hash	Parent block hash	32
Block hash	Block hash identifier	32
Block number	Block height	8
Block timestamp	Block construction approximate time by primary node	8
Contract status hash	Hash identifier of all contract accounts status	32
Transaction set hash	Hash of the list of transactions included in the block	32
Return set hash	Hash identifier for the list of receipts generated by executing a transaction	32
Other	Block execution time stamp, block on-chain time stamp, etc.	24

Table 6.4 Definition of transaction structures

Field name	Description	Size
Version information	Transaction structure definition version information	3 bytes
Transaction hash value	Hash markers produced on the transaction	32 bytes
Transaction initiator address	A hexadecimal string of length 40 that identifies the initiator	20 bytes
Transaction receiver address	A hexadecimal string of length 40 that identifies the receiver	20 bytes
Contract call information	Call to contract function flags and call parameters after encoding	Indefinite
Transaction timestamp	Approximate time when a Hyperchain node receives a transaction	8 bytes
Random number	A randomly generated 64-bit integer	8 bytes
User signature	Signature information generated by the user to sign the transaction content	65 bytes

Due to this special chain design, any change in the parent block will change the hash of the parent block, forcing the “parent hash” field in the child block to change, resulting in a change in the hash of the resulting child block. If the latest locally maintained block hash matches the latest block hash maintained by the blockchain network, the locally held blockchain information is determined to be legitimate; otherwise, the local node will become a “Byzantine node.”

The block transaction list stores the collected transaction data. Each transaction contains the following fields, as shown in Table 6.4.

Contract Status

The Hyperchain system maintains information about the current state of the system in addition to the blockchain data. Unlike the UTXO model applied in the Bitcoin system, Hyperchain uses the account model to represent the system state.

When a Hyperchain node receives a “pending” transaction, it is first sent to the execution module for execution. After the transaction’s performance, the state of the relevant contract account is changed, e.g., a user A initiates a transaction to invoke deployed contract B, causing the value of variable b in contract B to change from 0 to 1 and persist in being stored in the contract state.

The execution of each transaction represents a transfer of the status of the contract account and a transfer of the status of the system book transfer. Thus, Hyperchain can also be considered as a state transfer system.

Hyperchain ledger records the status information of all contracts on the chain. The contract status metadata has the following paragraphs, as shown in Table 6.5.

In addition to the above metadata, the contract account has two other data fields: the executable code and the variable storage. Executable code is a set of instructions encoded in byte arrays, and each call to the contract is essentially a run of executable code. The variables defined in the contract are stored in the contract’s storage space, as shown in Fig. 6.15.

Table 6.5 Definition of contract account

Field name	Description	Size (bytes)
Contract address	A unique identifier used to identify the contract	20
Hash identifier of the contract storage space	Merkle tree is used to calculate the identity of the contract storage space	32
Contract code hash	Contract executable code hashes the generated identity	32
Founder	Create the account address for this contract	20
Create block height	Block height when deploying the contract	8
Contract status	Current contract accessibility status (normal or frozen)	1

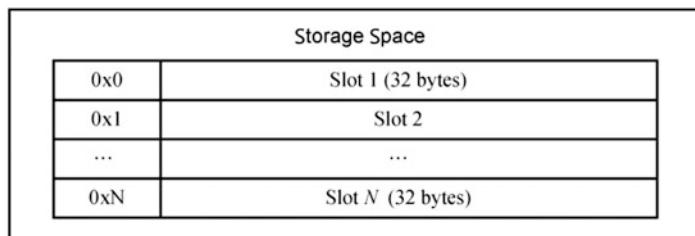


Fig. 6.15 Diagram of contract account storage space

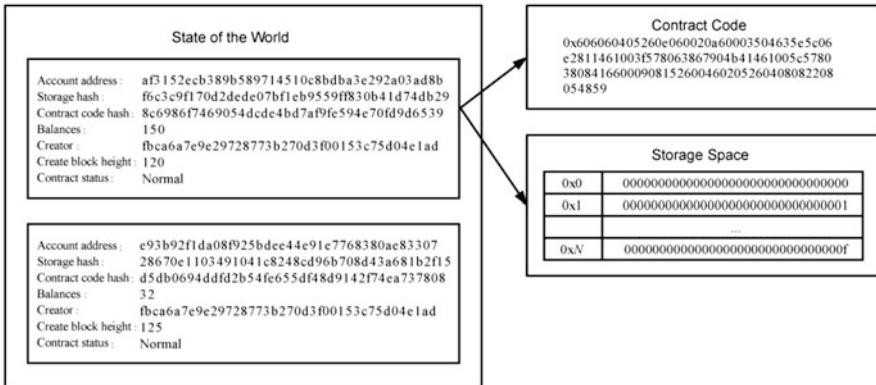


Fig. 6.16 Schematic diagram of contract status data

The storage space is similar to the standard storage structure and logically consists of a slice of address adjacent storage cells (empty storage cells not written to the disk to save disk storage space). Each memory cell is called a slot and is 32 bytes in size. Contract variables get their index addresses in the storage by compiling them in the contract phase, and the contents are stored in the corresponding slots.

A simple diagram of contract state data is shown in Fig. 6.16.

Merkle Tree

To quickly generate a hash that identifies the new state of all contract accounts, the Hyperchain system introduces the Merkle tree for hashing. The following is a brief introduction to the structure and role of the Merkle tree.

A Merkle tree is a hash binary tree, a data structure to summarize and verify the integrity of large-scale data quickly. This binary tree contains cryptographic hashes and is used in the Bitcoin network to summarize all transactions in a block while generating a digital fingerprint of the entire transaction set and providing an efficient way to verify the existence of a transaction in a block. However, the performance of traditional Merkle trees is poor, and the computation performance cannot meet the requirements of federated chains when facing high-frequency and massive data. Therefore, in Hyperchain, a HyperMerkle tree that combines the respective advantages of Merkle tree and hash table data structures is designed, which dramatically improves the rate of hash computation in the ledger.

A traditional Merkle tree is constructed from the bottom up, as shown in Fig. 6.17, starting from four data blocks, L1, L2, L3, and L4. The data of these four data blocks are first hashed, and then the hash values are stored to the corresponding leaf nodes. These leaf nodes are Hash 0-0, Hash 0-1, Hash 1-0, and Hash 1-1, respectively.

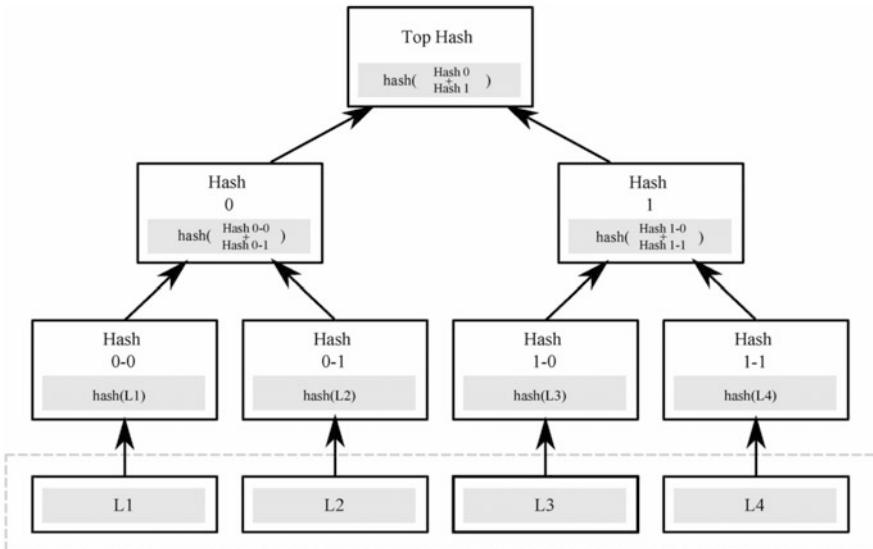


Fig. 6.17 Example of a traditional Merkle tree

After completing the bottom leaf node assignment, the value of the non-leaf nodes begins, which is calculated by concatenating the hashes of the neighboring leaf nodes and using them as input to calculate the hash. The result is the hash of the pair's parent node.

Continue similar operations until only the top node, the Merkle root, remains. The hash value of the root node represents the identification of the batch of data blocks.

This traditional Merkle tree is only suitable for scenarios like the hashing of bulk transaction data in the Bitcoin system, not for the rapid calculation of ledger hashing in the federation chain. Therefore, the HyperMerkle tree with hash table features is redesigned in Hyperchain.

HyperMerkle tree is a multi-fork tree built on a hash table. Each storage cell of the hash table is a leaf node of the HyperMerkle tree. All the leaf nodes are called n -layer nodes. Several adjacent leaf nodes are summarized into a parent node, and the parent node set generated is called $n-1$ layer node. Recurse until only the top node is left as the root of the HyperMerkle tree. Each parent node maintains a list of child node hash values. The HyperMerkle tree structure is shown in Fig. 6.18.

A calculation of the HyperMerkle tree is shown below:

1. Hash each element in the input data set to different positions according to the critical value and use the zipper method to deal with hash conflicts.
2. Hash recomputation is performed for each leaf node involved in the change, with the input being the content of the leaf node; the result is written to the child node hash list of the corresponding parent node when the computation is complete.
3. Hash recomputation is performed for each $n-1$ level node involved in the change. The input is a list of the node's child node hashes (the hashes of the child nodes

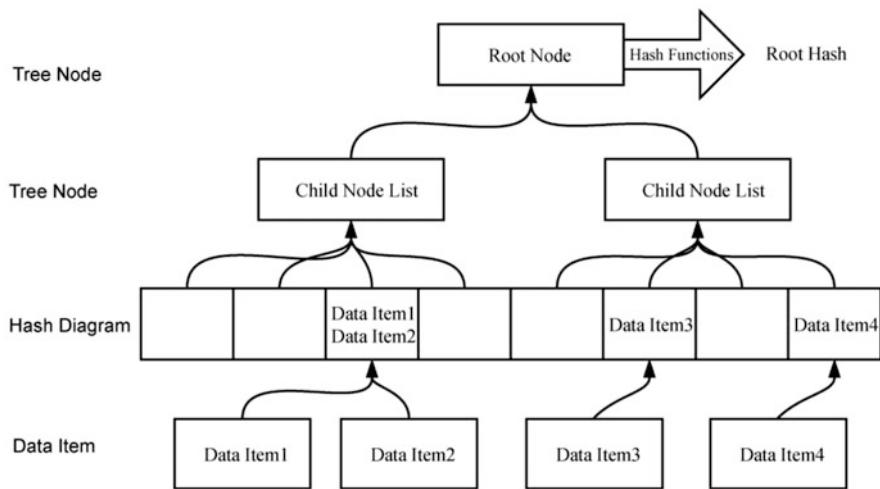


Fig. 6.18 Example of a HyperMerkle tree

not involved in this computation using the value of the last calculation); the count is completed, and the result is written to the child node hash list of the corresponding parent node.

4. Repeat step (3) until the computation reaches the Tier 1 node. The Tier 1 node is also called the root node, and the root hash represents the current hash of the ledger.
5. Persist the contents of all nodes for this recomputation.

A HyperMerkle tree maintains a batch of data and only hashes the modified portion after each change, significantly improving computational efficiency.

HyperMerkle tree performs hash calculation of two parts in Hyperchain: hash calculation of storage space of contract account; Hash calculation of the set of contract accounts.

For each contract account, the contents of the storage space are the inputs to the HyperMerkle tree, and the outputs are stored in the metadata of the contract account; for a set of contract accounts, the contents of each contract are inputs to the HyperMerkle tree, and the output is stored in the block as an indication of the current state of the contract account set.

6.3 Expand Components

Enterprise blockchain platform is also known as alliance chain. The term “alliance chain” has two meanings: first, it is a blockchain, and, second, it is a limited member alliance. Therefore, in the design of enterprise blockchain security mechanisms, it is necessary to consider both the trust between the members of the traditional

blockchain and the security management mechanism of the access of alliance members. To this end, the Hyperchain platform proposes a multilevel encryption mechanism based on cryptography, which applies secure encryption algorithms to fully encrypt user information in the transaction network, transaction parties, and transaction entities, and also proposes a ca-based permission control mechanism. In addition, to meet the requirements of high scalability and high availability of the enterprise blockchain platform, the platform has introduced data management, message subscription, and other functions. The platform features are described in detail in this section.

6.3.1 Privacy Protection

Partitioned Consensus

Hyperchain has designed the Namespace mechanism to achieve partitioned consensus for transactions within the blockchain network to improve data security and privacy protection and support flexible and independent business scenarios. Users can divide business transactions according to Namespace, and nodes in the same Hyperchain federated chain network form a subnetwork with Namespace as the granularity according to the business they are involved in, like a box to achieve physical separation between different businesses, so that transactions in each space do not interfere with each other.

A Hyperchain node can participate in one or more namespaces based on service requirements. As shown in Fig. 6.19, Node1, Node2, Node4, and Node5 form Namespace1, while Node2, Node3, Node5, and Node6 form Namespace2. Node1

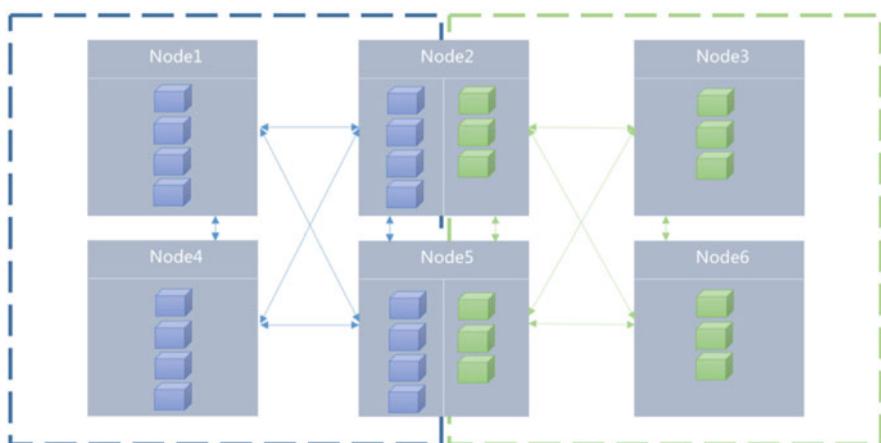


Fig. 6.19 Schematic diagram of partition consensus

only participates in Namespace1, while Node2 participates in two namespaces. The Namespace controls the dynamic join and exit of nodes through CA authentication.

The verification, consensus, storage, and transmission of transactions with specific Namespace information are only carried out between nodes participating in a particular Namespace. Transactions between different namespaces can be executed in parallel. For example, Node1 in Fig. 6.19 can only participate in transaction validation and ledger maintenance for Namespace1, while Node2 can participate in transaction execution and ledger maintenance for both Namespace1 and Namespace2. However, the ledger of Namespace1 and Namespace2 in Node2 are isolated from each other and not visible to each other.

Privacy Transaction

Hyperchain mainly implements the deposition of private transactions, the deployment, invocation, upgrade of privacy contracts, etc. It has a more granular privacy protection solution for federated chains.

Hyperchain can support privacy protection at transaction granularity, specifying the related party of the transaction when sending the transaction; the transaction details are only stored in the related party, and the hash of the private transaction is stored in the public ledger after network-wide consensus, which ensures effective isolation of the private data and verifies the authenticity of the private transaction.

Figure 6.20 illustrates the respective processes and differences between privacy.

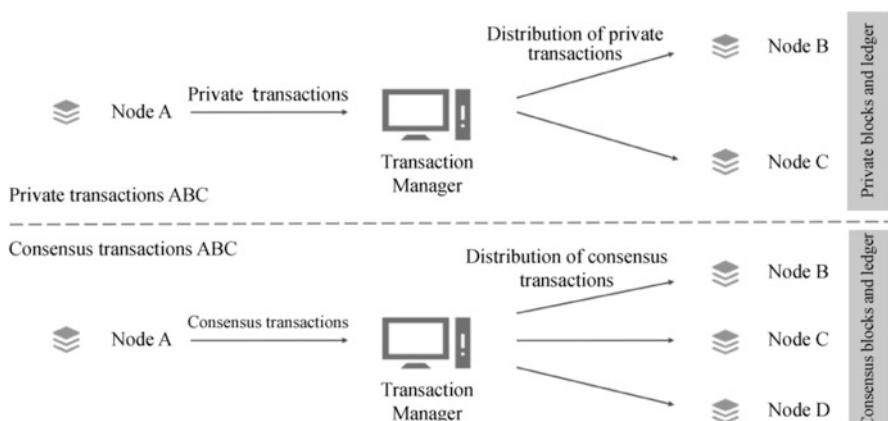


Fig. 6.20 Schematic diagram of a privacy transaction versus a consensus transaction

Encrypted On-Chain/Hashed On-Chain

For certain highly sensitive information that is not strongly correlated with transactions and ledgers, the data can be encrypted in plain text with symmetric encryption before being put on the chain to protect the private data, or the original data and files can be kept under the chain. Only their digital abstracts can be stored on the chain by hashing, solving data capacity and sensitivity issues.

Contract Access Control

Contract coders can restrict the roles and users who can access the data through smart contracts and access control policies, i.e., different access rights to contract functions can be customized in the contract for nodes, roles, and users. Contract coders can set access control for some high authority functions in the contract, so that the function can only be called by a fixed address caller thus achieving access control.

6.3.2 *Encryption Mechanism*

Hyperchain adopts a pluggable multilevel encryption mechanism, which encrypts data, users, and communication connections involved in the complete business lifecycle with different policies, making it convenient for enterprise users to choose encryption methods according to specific business scenarios while ensuring system security and efficiency.

Hash Algorithm

The hashing algorithm can transform an input of arbitrary length into an output of fixed size (a hash); the space of the hash is usually much smaller than the space of the input, and the hash function is irreversible so that the content of the original input cannot be inferred back from the hash.

Hash algorithms are widely used in the Hyperchain platform, such as transaction digests, contract addresses, and user addresses. Hyperchain provides pluggable options for different security levels of hashing algorithms. The security levels from low to high are SHA2-256, SHA2-256, SHA2-384, SHA2-384, etc. All these hashing algorithms can ensure the generation of digital fingerprints for messages with controllable volume and irreversible pushback to ensure the platform's data security.

ECDSA-Based Transaction Signatures

To prevent transactions from being tampered with, Hyperchain uses the proven elliptic curve digital signature algorithm (ECDSA) to sign transactions and secure the platform's identity. The signature process is shown in Fig. 6.21.

The security of elliptic curve cryptography is based on solving the elliptic curve discrete logarithm problem. Since there is no sub-exponential time solution to this problem, the unit bit intensity of elliptic curve cryptography (ECC) is much higher than that of a traditional discrete logarithm system; therefore, the calculation parameters are smaller, which leads to shorter keys and faster computation and thus shorter signatures.

Hyperchain adopts secP256K1 curve and R1 curve to implement the digital signature algorithm. Users can select a signature to verify transactions on the platform, ensuring the correctness and integrity of transactions. Meanwhile, the platform supports the signature verification of messages between nodes by using this algorithm to ensure the correctness and integrity of message communication between nodes. The Hyperchain employs the ELLIPtic curve encryption standard encapsulated in THE C language to achieve better performance in signature and verification.

ECDH-Based Key Negotiation

During network communication, session keys to encrypt transmitted messages prevent hackers from eavesdropping on confidential messages for fraud, etc. Hyperchain accomplishes the establishment of session keys and mutual authentication between users in the network by implementing the Elliptic Curve Diffie-Hellman (ECDH) key negotiation protocol. This ensures that both communicating

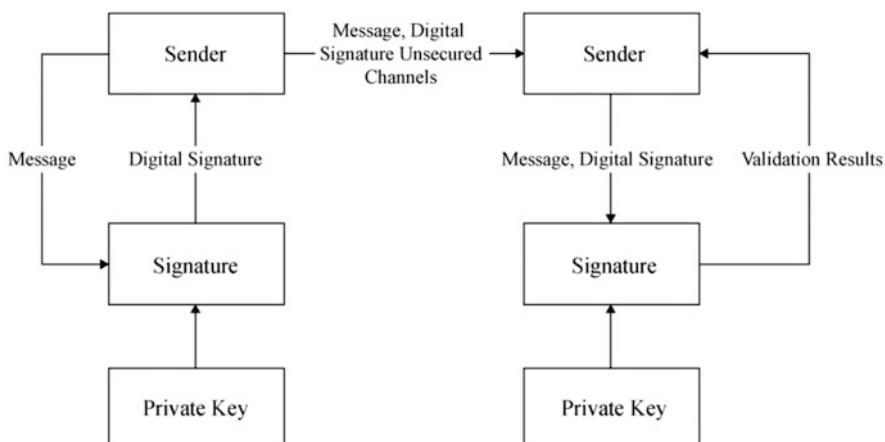


Fig. 6.21 Digital signature flowchart

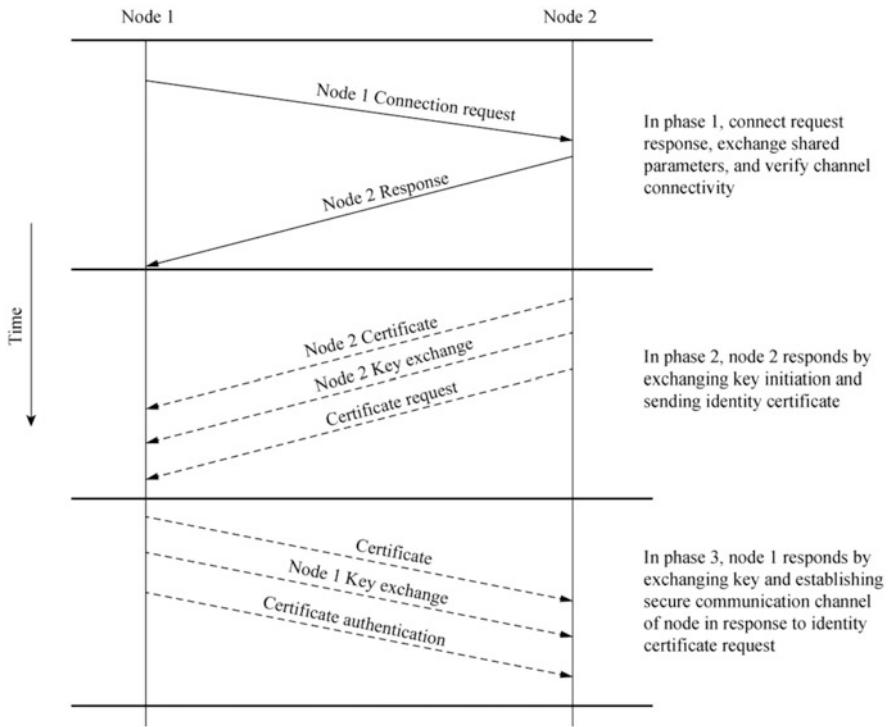


Fig. 6.22 Schematic diagram of the key negotiation exchange process

parties can create a shared confidential protocol over insecure public media to create a shared secret protocol without exchanging any private information beforehand.

In Hyperchain, the exchange of shared keys is first implemented using ECDH, and the exchange process is shown in Fig. 6.22. The ECDH algorithm establishes a secure channel for key negotiation with secure identity authentication in mind. Any organization intercepting the exchange can copy the public parameters and the public keys of both communicating parties but cannot generate a shared confidential agreement from the publicly shared values. After the shared public key is negotiated, symmetric encryption is significantly improved communication efficiency.

ECDH key negotiation plays a vital role in identity authentication and transaction security. The secure communication channel established by key negotiation can realize secure information exchange and ensure the communication security of the platform. The critical negotiation secure channel established with secure identity authentication can confirm the legitimate identity of the two communicating parties and then, through symmetric encryption, can significantly improve communication efficiency because there is no need to authenticate the uniqueness of each communication.

Ciphertext Transmission Based on Symmetric Encryption

After the Hyperchain communication parties negotiate a secret shared key, the symmetric encryption algorithm ensures ciphertext transmission between nodes, making it more difficult to decrypt the transmitted content and ensuring high message transmission security on the platform.

Symmetric encryption is also a conventional, private key or single-key encryption. A complete symmetric encryption scheme consists of five parts.

- Plaintext: The original message or data serves as the algorithm input.
- Encryption algorithm: Encryption algorithm performs various substitutions and transformations on the plaintext.
- Secret key: The algorithm input, the algorithm for substitution, and transformation all depend on the secret key.
- Ciphertext: A message that has been scrambled as the output of an encryption algorithm, depending on the plaintext and the secret key. For a given message, two different secret keys will yield different ciphertexts.
- Decryption algorithm: Essentially a reverse operation of an encryption algorithm using ciphertext and secret keys to generate the original plaintext. Hyperchain supports the Advanced Encryption Standard (AES) algorithm, an iterative symmetric key grouping based on permutations and permutations. It can encrypt and decrypt data in 128-bit (16-byte) groups using 128-bit, 192-bit, and 256-bit keys.

Transport Layer Security

In addition to the above key negotiation and ciphertext transmission, Hyperchain nodes adopt transport layer security (TLS) to ensure communication safety between them, including Google, Taobao, Baidu, WeChat, etc.

Transport layer security is a feature that Hyperchain enables by default, using the certificate issued by TLSCA for secure communication; that is, the security of the transport layer security protocol certificate needs to be verified during network transmission, and normal network communication can be carried out if the verification is passed; otherwise, network communication cannot be carried out. It is optional for configuration.

State-Secret Support

Compared to other blockchain platforms, Hyperchain has a significant advantage in cryptographic algorithms: it fully supports the integration of state secret algorithms. Hyperchain has integrated the state secret algorithms SM2, SM3, and SM4 and is compliant with the SSL VPN technical specification.

Among them, SSL VPN includes various network communication protocols, for replacing OpenSSL; SM2 is a public essential cryptographic algorithm standard

established on elliptic curve cipher, which has a digital signature, key exchange, and public key encryption, replacing international algorithms such as RSA, Diffie-Hellman, ECDSA, and ECDH; SM3, as a cryptographic hashing algorithm, replaces international hash algorithms such as MD5, SHA-1, and SHA-256; SM4 is a packet cipher algorithm used to replace global symmetric encryption algorithms AES, DES, and 3DES.

6.3.3 Member Management

CA System

Hyperchain authenticates primarily through a CA system by a certificate issuance streamlined system, as shown in Fig. 6.23.

Root.ca (Root Certificate Authority) represents the trust anchor in the PKI system. The Root CA, the uppermost CA in the PKI hierarchy, is designed for issuing certificate authorities as well as role certificate access certification authorities.

ECert (enrollment certificate) is an access certificate, and ECA (enrollment certificate authority) is an access certificate authority, which can issue node access certificates downwards. Nodes with ECert can only interact with Hyperchain services; otherwise, they cannot join the corresponding Namespace.

In addition, Hyperchain's ECert is designed to be implemented in two ways. An organization holding an ECert1 has the authority to interact with Hyperchain services and issue a TCert (transaction certificate) down the chain. The transaction certificate is used to implement pseudo-anonymous transactions, and the client

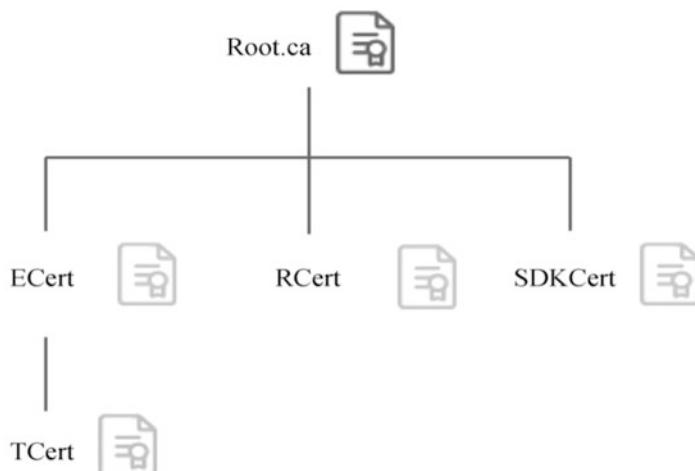


Fig. 6.23 CA certificate issuance system

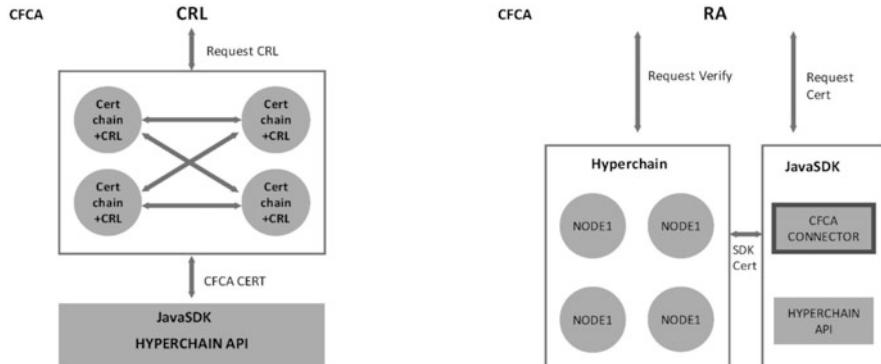


Fig. 6.24 Schematic diagram of CFCA certificate system

carries it out when initiating a transaction. The client encrypts the Transaction with a private key matching the TCert. TCert can be applied online and issued by each node. Each Transaction is signed with a new TCert thus achieving relative anonymity for each Transaction, but can be reviewed by the issuer.

Role Certificate Authority (RCA) is a role certificate authority that can issue Role Certificate (RCert). RCert is mainly used to distinguish authentication and non-authentication nodes in blockchain nodes. Only those with RCert are considered authentication nodes in blockchain and participate in the consensus among blockchain nodes. RCert and TCert can only be used as identity certificates and cannot be issued to other certifications.

Hyperchain's certificates are compliant with the ITU-T X.509 international standard, which contains only public key information and no private key information, and can be publicly distributed.

In addition, the Hyperchain platform integrates with the China Financial Certification Authority (CFCA) to manage digital certificates, meeting the requirements of banks and financial companies that have high requirements on certificate system security and authority. CFCA certificate system is shown in Fig. 6.24. It provides two service modes: CRL mode and RA mode. The CRL manages the RA service and uses the CFCA to host the RA service to issue and verify certificates. In RA mode, the privatized RA service is deployed locally to give and verify certificates in RA mode. Both modes are now integrated.

CFCA certificate system can be applied to Hyperchain nodes to verify the certificate and validity of the SDK. The purchased certificate is configured in the SDK, and Hyperchain nodes have to configure the verification certificate chain provided by CFCA; when the SDK sends the certificate to the node, Hyperchain will verify its validity. It has to verify that it is not blocked by the certificate through network request CRL.

In terms of JavaSDK, SDK selects the corresponding signature algorithm to sign the transmitted content according to the CFCA feature switch when sending

SDKCert. The CFCA function of the SDK and Hyperchain must be enabled or disabled simultaneously.

Certificate Management

Hyperchain provides the certificate management tool Certgen, which is used to generate and manage CA and digital certificates, including certificate issuing, public and private key generation, and certificate checking.

Certification Issuance

A pair of public and private keys must be generated before the node starts. During the node startup, each node generates a license certificate based on the public key and generates a root certificate.

When issuing a sub-certificate, a sub-certificate of the specified type (ECert, RCert, TCert, and SDKCert), or the user provides the public key to the issuer, which in turn issues one for them.

Node Access

The new node first issues a join request to each node and all VP nodes on the chain query their CA public vital certificates on the handshake information of the applied node, use the general key certificate for signature verification, and vote by ACO whether to allow the node with this CA certificate to join. If qualified, the ECert certificate of this node is issued to it, while the new node also gives ECert certificate to the applied node.

Certificate Checking

The certgen provides a certificate checking service, which includes whether the certificate is issued by a CA certificate, whether the signature is legal, and whether it is a CA certificate that can print a sub-certificate.

Certificate Revocation

Certificate revocation generally occurs when the user's personal identifying information is changed, the private key is lost, leaked, or suspected to be leaked. The general steps are: firstly, the certificate user submits the certificate revocation request to CA, and then CA puts this certificate into the publicly released certificate

revocation list, which contains all the digital certificates that are within the validity period but revoked.

In addition, a digital certificate can be revoked before its expiration date. For example, in some special cases, the certificate user arbitrarily uses the certificate for improper purposes and is discovered by the CA, or the authority such as the government agency requires the certificate to be revoked for some reason.

Key Management

Hyperchain provides two key management models for user public-private key pairs: either banks and third-party organizations host the private keys, or the users themselves keep them. Both management models are equipped with corresponding critical recovery solutions.

1. The user's private key is split into two separate parts for encryption at the time of issuance, one of which is held by the bank and the other by a trusted third party thus ensuring that no one institution can independently steal the user's private key for signature transactions.

If the user's private key is lost, the organization will authenticate the user offline and initiate the private key recovery process after the authentication is passed. The user obtains part of the private key from the two institutions hosting the private key, respectively, decrypts and pieces together, and finally obtains the complete private key. The private key is exactly the same as the user's original private key, and the original private key can still be used for signature transactions.

2. The user keeps the complete public and private key pair and does not back it up.

Once the user's private key is lost, it cannot be recovered. The bank will generate a new private key for the user and transfer the user's original assets to the new public key address by invoking a smart contract of the super administrator in the background. This scenario relies on the following operations:

- Design a separate super administrator smart contract based on the existing business contract (a respective administrator smart contract for each bank that can only make transfers to the assets of the Bank's customers);
- The public key address of the user's depository bank needs to be recorded in the contract associated with the user's assets.
- When the user key pair is generated, the corresponding public key address is stored in the bank database, forming a mapping relationship (user account/identity information ->public key address), where the user identity information can be authenticated offline.
- After the user's private key is lost, the bank initiates a private key reset application; the bank first authenticates the user's identity, and then initiates the asset transfer process, where the system generates a new public-private key pair, then obtains the user's last public key from the database, then invokes the super

administrator smart contract to transfer all existing assets corresponding to the user's previous public key address to the user's new public key address, and uses the bank's private key to sign the transaction. The transaction record of the asset transfer is also recorded in the blockchain and can be publicly queried.

- The user's new public key address is added to the database (without deleting the user's original public key address).

Once an asset is allocated to a user's new public key address, the user can use the new private key to transfer the original asset to conduct transactions. When the user queries the historical transaction, all the public key addresses of the user are obtained from the database. After the private fundamental change, the transaction is queried through the new public key address, and the transaction before the change of the private key can be traced back to the used public key address.

6.3.4 Blockchain Governance

Hyperchain Implements Blockchain Governance Through Confederation Autonomy and Node Permission Management

Blockchain has attracted widespread attention for its decentralized and non-tamper-prone properties. It is considered to be used to solve the next-generation Internet value exchange problem and the credit problem of network transmission. However, in the process of engineering practice, the characteristics such as polycentric and not easy to tamper, which give blockchain its trustworthy properties, often bring many limitations to use. One of the more prominent points is the upgrade of smart contracts. It is well known that no system is free from vulnerabilities and no system is designed to determine all requirements at the outset. There is an obvious contradiction and conflict between the non-tamperability of blockchain and the need for iterative engineering updates, and resolving the conflict requires strong decision-making, but existing blockchain systems lack good governance mechanisms to make rational and democratic decisions.

To resolve the contradiction between the characteristics of blockchain, such as polycentric and not easy to tamper with, and the actual engineering practice, Hyperchain proposes an effective governance mechanism ACO that can promote blockchain self-improvement. When the initial protocol cannot meet the real needs or the blockchain network has extraordinary contradictions that are difficult to reconcile during the operation, and the protocol needs to be upgraded, the ACO coalition autonomy can adequately resolve these contradictions.

The advantages of the ACO federation autonomy mechanism proposed by Hyperchain are reflected in three aspects.

1. Federation membership changes. In existing federated chain systems, membership change is often strongly tied to identity authentication, usually authorized and certified by a third-party CA, making it the only intense center in a

multicenter blockchain system. The ACO mechanism uses smart contracts as the negotiation platform for changes and the nodes' self-distributed data certificates as the negotiation result credentials (distributed CAs). The member change process remains polycentric, and the negotiation process is open and transparent.

2. Smart Contract upgrades. With the design concept of initial trust from offline governance and subsequent trust from online governance, the ACO mechanism provides a set of effective contract upgrade governance: the upgrade strategy is specified by the coalition members in advance and written into the smart contract, and when an upgrade is needed, a proposal is initiated and voted on by the coalition members, and the smart contract collects the votes and automatically executes the corresponding proposal, which solves the problem of upgrading blockchain contracts with the help of the permission-controlled contract upgrade command. The smart contract collects the votes and automatically executes the corresponding proposals.
3. Alliance chain system upgrades. There are two system upgrades: hard-forked noncompatible promotions for public chains and offline manual compatibility upgrades for alliance chains. Hyperchain proposes an effective online negotiation system upgrade mechanism that can realize efficient and automated synchronous system upgrades.

Permissions Management

To meet the needs of more affluent and complex business application scenarios, Hyperchain proposes a hierarchical permission management mechanism to safeguard business privacy and security further.

1. Chain-level administrator: involved in the permission management at the blockchain level, including the permission control of node management, system upgrade, and contract upgrade, is usually the super internal administrator designated by each alliance organization. Chain-level operation rights, such as node access, system upgrade, and contract upgrade, should be decided by the vote of the alliance agencies, not by a single entity. Specific voting rules are negotiated offline by the consortium and written into the Genesis block. If you want to change later, you must vote one round according to the rules agreed previously to complete the change. Chain-level authority management relies on the ACO mentioned above.
2. Node Administrator: participates in node-level rights management, including control of node access rights, and is often the designated O&M administrator for each federated organization. The node administrator issues access certificates (SDKCert) to each user to control the user's access to the SDK interface, and the node only accepts requests with node access certificates. The node administrator can issue the certificate through the client, configure the user permission table, and assign user permissions to access the SDK, such as access to call contracts and access to blocks. The chain-level administrator comes with a node access certificate SDKCert by default.

3. Users: ordinary users involved in on-chain business scenarios. Users can hold certificates issued by different nodes and initiate transactions to other nodes. The upper layer business system itself defines the permissions of specific users in the corresponding business scenarios. The platform can later abstract a series of standard permission management interfaces for the business layer to manage the permissions better.

At the business level, Hyperchain sets up contract access control, where contract coders can customize the access rights of contract functions in the contract, and set up permission control for some high authority functions so that the operation can only be called by callers with fixed addresses thus achieving access control.

6.3.5 *Message Subscription*

System Design

Hyperchain, as a shared-state blockchain implementation, operates through constant state changes. Each state change generates a corresponding series of events that serve as flags for this state change. Thus, to allow external users to better monitor Hyperchain's state changes, we provide a set of unified message subscription interfaces for external systems to capture and listen to Hyperchain's state changes as a message channel for smart contracts to communicate with the outside.

There are three types of events that can be easily monitored externally through message subscription systems: (1) Generating new blocks; (2) New events arising from the contract; (3) The system is abnormal. More event subscriptions, such as status changes for transactions, will be supported in the future. The message subscription system is encapsulated in the Hyperchain event routing module, and its architecture is shown in Fig. 6.25.

The system has a three-tier structure, which can be logically divided into upstream data collection, filtering and pushing, and downstream data exporting. The upstream data is obtained through the event router. The message subscription

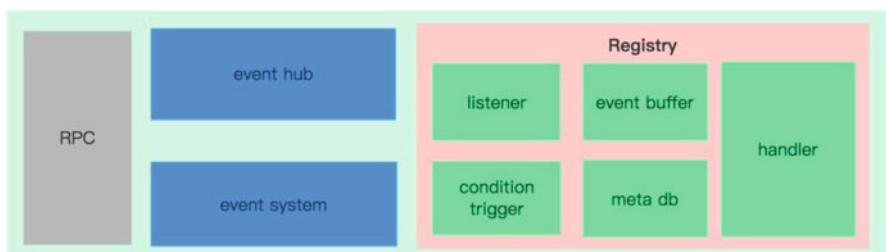


Fig. 6.25 Message subscription system architecture diagram

system does the data filtering and pushing, and the downstream data exporting is done through the RPC module.

Message subscriptions are easy to use, and the general process is as follows.

1. External users initiate subscription requests.
2. Hyperchain returns the subscription ID.
3. External users obtain their subscribed messages by either active polling or passive pushing.

Concrete Implementation

Hyperchain implements message subscription according to WebSocket and MQ (Message Queue), respectively.

WebSocket, a method provided by the system for network communication, contains both sides of the communication, i.e., Client and Server. Publishing a subscription requires maintaining a list of connected clients on the Server side, and a long and reliable connection between the Client and the Server needs to be established. The Socket on the Server side must create a Listener to listen for connections from the Client, and the Client can only exchange messages once it has connected to the Server.

MQ is a method of communication with the addition of a message holding container, where the server and client communicate by reading and writing notes to and from a queue without direct interconnection. The exception recovery mechanism provided by MQ solves the problem of lost messages due to disconnections in WebSocket messaging subscription systems. The platform MQ service requires the user to start a RabbitMQ server independent of the platform and to refer to the server on which it is hosted as a RabbitMQ-broker. The MQ service provided by the platform is equivalent to a producer client, dealing with pushing platform messages to the RabbitMQ-broker; the consumer of the messages acts as a consumer client that establishes a connection to the RabbitMQ-broker and waits for messages to be pushed from the RabbitMQ-broker.

The specific usage of the MQ service is as follows:

1. The user registers his message queue with the platform, specifying the set of RoutingKeys required for the queue; the platform sets up the queue and starts the service, returning the name of the queue and the switch name Exchanger to the user.
2. Users typically send transactions using the platform, and messages are automatically pushed to the RabbitMQ-broker by the MQ service when an event corresponding to a routing key is generated.
3. Users can get the information they subscribe to by active query or passive push.

6.3.6 Data Management

Data Visualization

To maintain the privacy of the contract data, the blockchain platform encodes all the underlying data of the smart contract deployed on the Hyperchain platform in a complex encoding way. The blockchain nodes cannot obtain the plaintext information of the contract data even if they have the total amount of blockchain data.

However, some organizations must analyze or audit the contract data stored on the blockchain; Hyperchain thus provides a source code parsing-based contract data parsing solution. In this way, organizations can use the data visualization service provided by Hyperchain to parse the data of the contract and extract the plaintext data of the contract for auditing and analysis, provided that they have obtained the contract source code, contract address, and contract data key set, while other organizations without the contract source code, contract address, or contract data key set cannot parse the plaintext data.

There are three necessary prerequisites in the process of parsing data at one time.

1. Possessing the source code of the contract;
2. Possessing the address of the contract;
3. Possessing the data key set for the contract.

The primary data in a smart contract is organized and managed using a basic data structure such as a map. The data in a map can be likened to a table in a traditional relational database, and parsing the data in a map requires having a key set of all key value pairs stored in the map. Thus, for a particular participating institution, the key set corresponding to its own generated data can be maintained by itself, and this data can only be parsed by itself without disclosure.

Finally, the data can be imported into a relational database (e.g., MySQL) for analysis or auditing, as shown in Fig. 6.26.

More importantly, Hyperchain sees Radar as an important tool to obtain high-quality data that can be analyzed. After obtaining these data, users can conduct data processing, data analysis, and other operations to mine the hidden value in the data.

ser	status	version	balance	creditRating	accountID	account_json	address	birthday	cardAccontNu...	id_number	isOverdue
1	HULL	0	0	1876811...	HULL	HULL	15024...	62230912358...	HULL	0	
1	HULL	0	0	1876811...	HULL	HULL	15024...	62230912358...	HULL	0	
1	HULL	0	0	1876811...	HULL	HULL	15024...	62230912358...	HULL	0	
1	HULL	0	0	1876811...	HULL	HULL	15024...	62230912358...	HULL	0	
1	HULL	0	0	1876811...	HULL	HULL	15024...	62230912358...	HULL	0	
1	HULL	0	0	1876811...	HULL	HULL	15024...	62230912358...	HULL	0	
1	HULL	0	0	1876811...	HULL	HULL	15024...	62230912358...	HULL	0	
1	HULL	0	0	1876811...	HULL	HULL	15024...	62230912358...	HULL	0	
1	HULL	0	0	1876811...	HULL	HULL	15024...	62230912358...	HULL	0	
1	HULL	9000000	5	1876811...	HULL	HULL	15024...	62230912358...	HULL	1	

Fig. 6.26 Contract plaintext data in MySQL

Data Archiving

With the growth of blockchain operation time, the storage capacity of blockchain will grow linearly, and this data growth will even outpace the growth of storage media capacity, so blockchain data storage will naturally become an important factor limiting the development of blockchain technology. Faced with this thorny and much-needed problem, Hyperchain proposes a blockchain data archiving approach that enables the entire blockchain system to perform dynamic data archiving without downtime.

The blockchain data archiving approach provided by Hyperchain is established on state backup. In simple terms, if a user wants to do data archiving for a blockchain node, he must make a blockchain state snapshot at a certain point in the past; when the user does data archiving, he can archive all the blockchain data (including block data, transaction data, return data, etc.) before the snapshot point and dump it, so as to achieve the reduction of storage pressure on the blockchain node.

Figures 6.27, 6.28, and 6.29 show the specific process of a data archive. Figure 6.27 shows the current state of a blockchain node: the node obtains the current state of the world (also known as the blockchain ledger state) through 94 state transitions based on the creation state. At this point, the user initiates a snapshot request, and the blockchain node stores the state of the world as a backup.

As shown in Fig. 6.28, after making the snapshot, the node has undergone three state changes, the world state has been updated three times, and the latest local block height is updated to 97.

At this point, the user initiates a data archiving request to dump and archive all blockchain data before the snapshot. This node dumps the block data with block number 0 ~ 93 and the corresponding transaction receipt data, updates the local content of the genesis state to the snapshot state obtained from the backup before, and updates the genesis block to the block with number 94, as shown in Fig. 6.29.

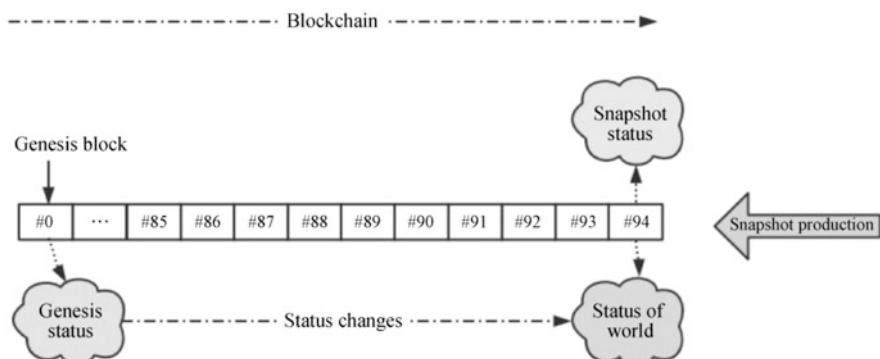


Fig. 6.27 User-initiated snapshot request

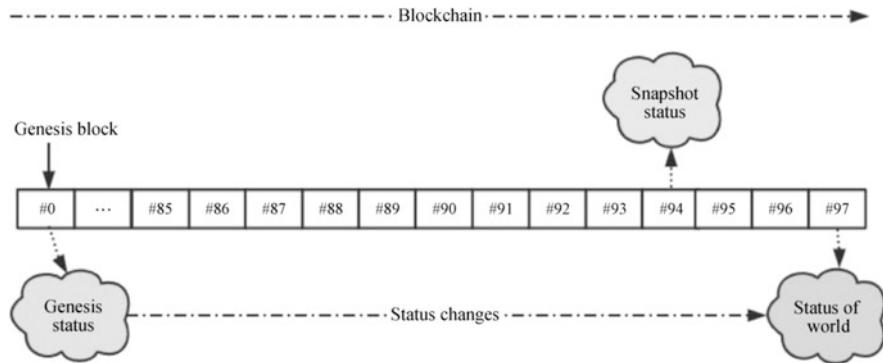


Fig. 6.28 Snapshot completed

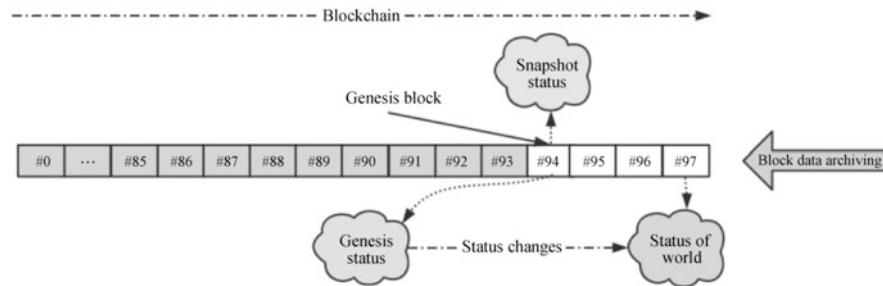


Fig. 6.29 User-initiated data archiving request

If the normal change of a blockchain state is a process in which the end point is constantly updated forward, then data archiving can be considered as a process in which the starting point of the state of a blockchain is updated towards the end point.

While the above data archiving is targeted at blockchain data, smart contracts deployed on the blockchain have a large storage requirement to record huge business data. Hyperchain provides another archiving mechanism for such data. Users only need to initiate a transaction marked with a special mark and invoke the customized archiving function in the smart contract to dump the contract data. Contract coders can implement arbitrary logical archiving functions in contracts to meet different business requirements. In addition, Hyperchain introduces an Archive Reader for easy access to archived data in the future.

6.3.7 Verification Based on Hardware Acceleration

The unit bit intensity of elliptic curve cryptosystem is much higher than that of traditional discrete logarithm system. Since all signature verification requests are stored in blocks of fixed size, each block has a fixed and large number of requests,

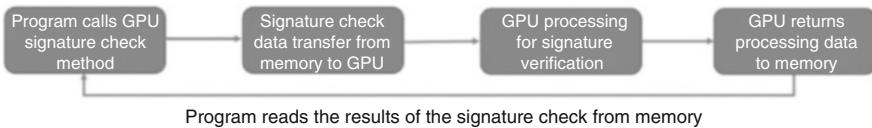


Fig. 6.30 GPU-based signature verification processing flow

Table 6.6 Influence of package check of different order quantity on system performance

Number of orders in the transaction (order)	Transaction TPS	Orders TPS	Time delay
1	22,630	22,630	22.14
10	21,600	21,600	23.08
20	20,271	405,420	24.66
30	18,466	553,980	27.1

Table 6.7 Impact of GPU acceleration on system throughput

Number of package inspection	GPU attestation TPS	CPU attestation TPS
1	46,421	9749
10	36,855	8654

and the program is required to complete the operation quickly and the response delay is high. Therefore, hardware acceleration of elliptic curve signature algorithm is needed to achieve faster signature operation.

Hyperchain implements a GPU-based signature verification algorithm based on existing theories. GPU has far more computing units than CPU and is good at large-scale concurrent computing. Therefore, the platform uses NVIDIA GPGPU and CUDA as the development environment to realize elliptic curve scalar multiplication by GPU in parallel. Figure 6.30 shows the processing flow of this algorithm.

Compared with CPU, using GPU improves the transaction and check by an order of magnitude, as shown in Tables 6.6 and 6.7. The comparison between GPU and CPU is shown in Fig. 6.31.

As can be seen from the above chart, Hyperchain uses hardware acceleration to significantly improve the signature and verification performance of transactions.

6.4 Summary

This chapter has introduced the implementation principles of the core components of an enterprise-level blockchain platform, taking Hyperchain as an example. Unlike public and private chains, enterprise-level blockchains directly address the needs of enterprise-level applications and impose more stringent requirements on the security, privacy, ease of use, flexibility, and performance of blockchain systems. The Hyperchain enterprise blockchain platform is built to meet the needs of enterprises from the following points of view.

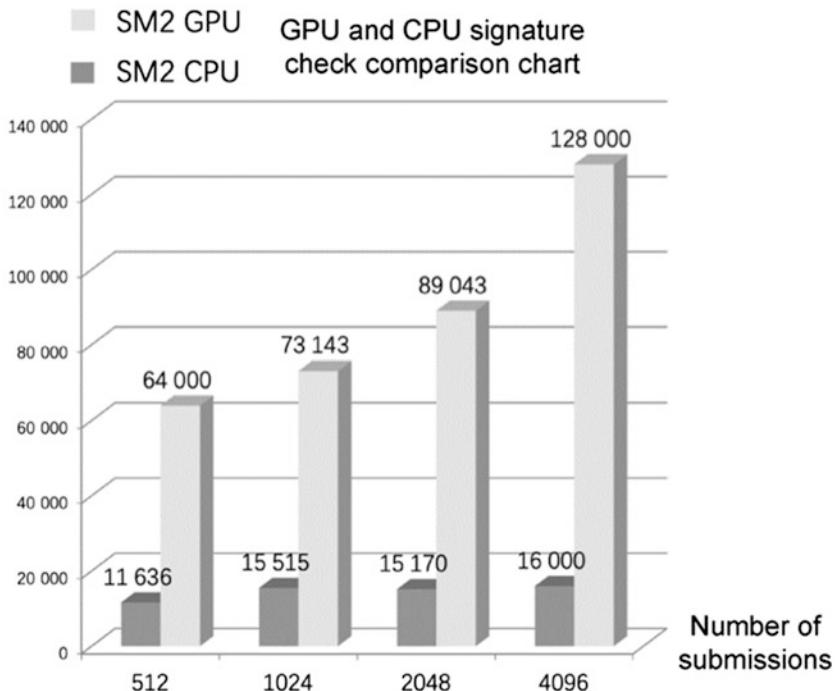


Fig. 6.31 Comparison of GPU and CPU checking of signatures

Firstly, the platform adopts ACO alliance autonomy and CA system authentication for member management, providing a comprehensive protection mechanism in terms of member access, identity authentication, and authority management. Secondly, the platform has designed and implemented a flexible, efficient, and stable consensus algorithm RBFT based on the optimization of traditional PBFT, which provides a solid algorithmic foundation for the enterprise-level blockchain platform and improves the throughput of transaction data. Moreover, the Solidity language, being active in the open-source field, is chosen to support smart contracts, and HVM, a lightweight sandbox smart contract engine, is independently developed to provide perfect contract lifecycle management and a contract environment with friendly programming, contract security, and efficient execution. Again, a different ledger system than Bitcoin is selected for the design of the blockchain ledger, and separate storage of block data and state data is adopted to improve data processing speed. In addition, the Hyperchain platform systematically strengthens the security level of the enterprise-level blockchain by partitioning consensus and privacy transactions for privacy protection at the business and transaction layers, and by encrypting transactions, transaction links, and application development packages at multiple levels. Finally, to improve the usability and interactivity of the enterprise-level blockchain, the platform provides functions such as message subscription, data visualization, and data archiving, which greatly enhance the utilization of transaction data.

Chapter 7

Hyperchain Application Development Fundamentals



Chapter 6 provides an introduction to the core system components of the Hyperchain platform, including detailed descriptions of features such as authentication licensing mechanisms for members, security and privacy of business transaction data, high transaction throughput and low transaction latency, a secure and complete smart contract engine, and interoperability for high user experience. As a federated blockchain technology infrastructure platform that satisfies the needs of industry applications, Hyperchain absorbs the most cutting-edge technologies from the blockchain open-source community and research fields, integrates high-performance and reliable consensus algorithms, compatible with a variety of smart contract development languages, and good execution environments from the open-source community; in the meantime, Hyperchain also enhances key features such as bookkeeping authorization mechanism and transaction data encryption, and offers a powerful visual web management console to efficiently manage blockchain nodes, ledgers, transactions, and smart contracts.

For such an enterprise-level blockchain core platform, how to develop applications on Hyperchain is the most concerning issue for developers. Therefore, this chapter will first focus on the basic features provided by the platform and its configuration and deployment and then specifically explain how to employ the Hyperchain platform for blockchain application development through a practical case.

7.1 Platform Functionality

Hyperchain, as a distributed and consistent ledger system, basically supplies services to the public in the form of transaction processing. There are two types of transactions in Hyperchain: normal and smart contract transactions. Normal types are relatively single-function, providing only relatively simple transfer functions. At the same time, smart contract-based transactions can implement user-defined complex

logic and encapsulate the transaction logic inside the smart contract, satisfying the flexibility and security of contract transactions. In addition, Hyperchain provides many functions related to blockchain data management, which is convenient for users to query blockchain data.

The Hyperchain platform has external software development services related to JSON-RPC, Java, and Go in terms of application interfaces. We will then take JSON-RPC as an example and explain in detail the primary services of Hyperchain, such as transaction invocation, chaincode management, and block management, respectively.

7.1.1 Platform Interaction

jJSON-RPC is a stateless and lightweight remote procedure call (RPC) protocol using JSON as the data transfer format, usually used as the operational portal for applications to access the blockchain platform. Users can easily make interface calls in any language according to their needs to complete interactive communication with the blockchain. The client sends a request object to the server via jJSON-RPC, each request representing an RPC call; generally, each RPC call request object needs to contain the following member fields.

- **jJSON-RPC:** a string specifying the jJSON-RPC protocol version, which must be written exactly as 2.0 if it is version 2.0 (Hyperchain adopts jJSON-RPC version 2.0).
- **method:** a string indicating the name of the method to be called. Method names that generally begin with RPC and are concatenated with an English period (U +002E or ASCII 46) are the method names and extensions reserved internally by RPC and cannot be used elsewhere.
- **params:** the values of the structured parameters needed to call the method, which can be omitted.
- **id:** the unique identification of the client, which must contain a string, numeric, or NULL value. If it does not contain one, the member is identified as a notification call. The value is typically not NULL, and if it is a value, it should be an integer. When initiating an RPC call, the server MUST reply with a JSON object as a response, containing the following members.
- **result:** This member field MUST be included when the RPC call succeeds and MUST not be included when the invoked method fails. The value of this member field is determined by the called method on the server-side.
- **error:** This field must be included when the RPC call fails and when no error occurs. If an error occurs, this member object will contain the code and message attributes.
- **id:** This member field must be included. Note that the member value here should be consistent with the value of the id member in the request object.

7.1.2 Transaction Calls

Transaction invocation is the leading service interface used by Hyperchain applications. The interfaces associated with transaction invocation mainly include two types: one is the query interface for querying transactions, which supports query operations such as getting transaction information, querying transaction details, and querying transaction processing time; the other is the invocation for initiating transactions, which is mainly completed by the `tx_sendTransaction` method. It can encapsulate the transaction, make it execute on the blockchain, and store the result of the transaction on each distributed Hyperchain node in a consensus manner. Table 7.1 shows all the transaction-related interfaces provided by Hyperchain, among which the three methods `tx_sendTransaction`, `tx_getTransactionReceipt`, and `tx_getTransactionByHash` are most commonly used. The following will take these three interfaces as examples to describe how to call the transaction-related interfaces in Hyperchain.

The method `tx_sendTransaction` that sends the transaction interface is described in detail in Table 7.2.

This interface only supports non-smart contract transactions, such as ordinary transactions between users such as transfer transactions. Before calling the `tx_sendTransaction`, you need to call the `tx_getSighHash` interface to get the hash used for the client-side signature; next, call the `tx_sendTransaction` interface method to send the transaction to the blockchain platform after the client signature generates the signature.

Example 1 below shows an example of a call, transferring 2441406250 from account `0xb60e8dd61c5d32be8058bb8eb970870f07233155` to another account `0xd46e8dd67c5d32be8058bb8eb970870f07244567`. If this call is successful, the system will return a hash of the transaction, which subsequent clients can use to query the details of the transaction.

Table 7.1 Overview of transaction service interfaces

RPC methods	Function
<code>tx_getTransactions</code>	Get all transactions
<code>tx_getDiscardTransactions</code>	Access to all discard transactions
<code>tx_getTransactionByHash</code>	Query transaction details based on transaction hash value
<code>tx_getTransactionByBlockHashAndIndex</code>	Query transaction details based on block hash
<code>tx_getTransactionByBlockNumberAndIndex</code>	Check transaction details by block number
<code>tx_sendTransaction</code>	Send transactions
<code>tx_getTransactionsCount</code>	Query all transaction volumes on the chain
<code>tx_getTxAvgTimeByBlockNumber</code>	Query the average transaction processing time in a given block
<code>tx_getTransactionReceipt</code>	Check the information of the designated transaction receipt
<code>tx_getBlockTransactionCountByHash</code>	Check the number of block transactions
<code>tx_getSighHash</code>	Get the hash value used for the signature algorithm

Table 7.2 Sending transaction interface

RPC methods	Parameters	Return value
tx_sendTransaction	{ from: <string> Originator Address to: <string> recipient address value: <number> Transaction volume value timestamp: <number> Transaction timestamp signature: <string> signature }	transactionHash: <string> Hash of the transaction, 32-byte hexadecimal string

Table 7.3 Interface table for querying transactions based on hash

RPC methods	Parameters	Return value
tx_getTransactionByHash	transactionHash <string> Hash of the transaction, a 32-byte hexadecimal string	< TransactionResult>TransactionResult object

Example 1 (Example of Sending a Transaction Interface Call)

```
// Request
curl -X POST --data '{
  "jsonrpc": "2.0",
  "method": "tx_sendTransaction",
  "params": [
    "from": "0xb60e8dd61c5d32be8058bb8eb970870f07233155",
    "to": "0xd46e8dd67c5d32be8058bb8eb970870f07244567",
    "value": 2441406250,
    "timestamp": 1477459062327000000,
    "signature": "your signature"
  ],
  "id": 71
}'
// Return results
{
  "id": 71,
  "jsonrpc": "2.0",
  "result": "0xe670ec64341771606e55d6b4ca35a1a6b75ee3d5145a99d05921026d1527331"
```

Table 7.3 shows the detailed description of the interface for querying transaction information according to the hash value. After the above-described sending interface successfully sends a transaction, it will return a hash representing the transaction. The specific information of the transaction can be queried through the hash value.

TransactionResult object.

```
{
  hash: <string> Hash of the transaction, 32-byte hexadecimal
  string
  blockNumber: <number> The height of the block the transaction is
  in
  blockHash: <string> Exchange in block hash
  txIndex: <number> The position of the transaction in the block's
  transaction list
  from: <string> Address of the sender of the transaction, a 20-byte
  hexadecimal string
```

Table 7.4 Interface for returning transaction acknowledgment information based on transaction hash

RPC methods	Parameters	Return value
tx_getTransactionReceipt	transactionHash: <string> transaction hash	<Receipt>

```

to: <string> Address of the recipient of the transaction, a
20-byte hexadecimal characters amount: <number> Transaction volume
timestamp: <number> Time when the transaction occurred (in ns)
executeTime: <string> Processing time of the transaction (in ms)
invalid: <boolean> whether the transaction is not legal
invalidMsg: <string> Invalid message for the transaction
}

```

The following Example 2 shows an example of querying transaction information by transaction hash. From the returned results, querying a transaction enables querying the hash of that transaction, the block number where the transaction is stored, and transaction-related details such as transaction specifics and execution time.

Example 2 (Example of Querying a Transaction Based on its Hash) // request

```

curl -X POST --data '{
  "jsonrpc": "2.0",
  "method": "tx_getTransactionByHash",
  "params": [
    "0x658406ea4edf92f4b9d1589c3ea84d75c07f4179
    908e899703eae0e3ea54caa2"
  ],
  "id": 71
}'
// Return result
{
  "jsonrpc": "2.0",
  "id": 71,
  "result": {
    "hash": "0x658406ea4edf92f4b9d1589c3ea84d75c07f4179908e89970
    3eae0e3ea54caa2",
    "blockNumber": "0x1",
    "blockHash": "0x9e330e8890df02d22a7ade73b5060db6651658b676dc
    9b30e54537853e39c81d",
    "txIndex": "0x9",
    "from": "0x000f1a7a08ccc48e5d30f80850cf1cf283aa3abd",
    "to": "0x0000000000000000000000000000000000000000000000000000000000000003",
    "amount": "0x1",
    "timestamp": 147745906232700000,
    "executeTime": "0x3ee",
    "invalid": false,
    "invalidMsg": "",
    "invalidMsg": ""
  }
}

```

The transaction result returned by the method that queries the transaction based on the transaction hash is a `TransactionResult` object, which contains only the transaction details, but not the execution result information. Whether the transaction succeeds or not and the corresponding returned result can be queried by the transaction return information interface method `tx_getTransactionReceipt`. Table 7.4 provides a detailed description of the `tx_getTransactionReceipt` call.

<Receipt> object.

```

{
  txHash: <string> transaction hash
  postState: <string> transaction state
  contractAddress: <string> contract address ret: <string> Result
of execution
}

```

Example 3 (Example of Querying Transaction Receipt Information Based on Transaction Hash) // Request

```

curl -X POST --data '{ "jsonrpc": "2.0",
  "method": "tx_getTransactionReceipt",
  "params": ["0xb60e8dd61c5d32be8058bb8eb970870f07233155"] ,
"id": 71
}'
// Return results
{
  "id":71,
  "jsonrpc": "2.0", "result": {
    "txHash": "0x9e330e8890df02d22a7ade73b5060db665165
8b676dc9b30e54537853e39c81d", "postState": "1".
    "contractAddress":
"0xe04d296d2460cfb8472af2c5fd05b5a214109c25688d3704aed5484f"
"ret": "0x606060405260e060020a60003504633ad14af3811460305780
63569c5f6d146056578063d09de08a14606d575b6002565b34600257600

0805460043563fffffffffff8216016024350163fffffff19909116179055
5b005b3460025760005463fffffffff166060908152602090f35b3460025

760546000805463fffffffffff19811663fffffffff9091166001011790555"
}
}

```

7.1.3 Contract Management

Smart contracts are a core component of blockchain systems, and through smart contracts, application developers can customize more complex and flexible on-chain asset processing logic. The Hyperchain platform provides a set of related APIs for chaincode management, including chaincode compilation, deployment, and invocation. The list of smart contract interfaces offered by Hyperchain is shown in Table 7.5.

The essential functions of the contract management interface are compiled, deployed, and invoked in detail. A detailed description of the invocation interface for smart contract compilation is shown in Table 7.6. Compiling the smart contract interface only requires uploading the source code of a locally written smart contract in parameters. As for how the contract is written, this is described in Sect. 7.3.1. compileCode is the result of the compilation of the interface, and the description

Table 7.5 Overview table of smart contract service interfaces

RPC methods	Function
contract_compileContract	Compile contracts
contract_deployContract	Deploy contracts
contract_invokeContract	Call contract
contract_getCode	Get the contract code
contract_getContractCountByAddr	Get the number of contracts
contract_encryptMessage	Get the homomorphic balance and the amount transferred
contract_checkHmValue	Obtain validation results for homomorphic transactions
contract_maintainContract	Maintain contracts (upgrades, freezes, unfreezes)
contract_getStatus	Check contract status
contract_getCreator	Query contract deployers
contract_getCreateTime	Check contract deployment time
contract_getDeployedList	Query the list of deployed contracts for a given account

Table 7.6 Compiling contract source code

RPC methods	Parameters	Return value
contract_compileContract	<string> Contract source code	< compileCode>

Note: The contract is compiled locally; make sure that the Solidity environment is installed locally; otherwise, it will not compile

shows that the contract compilation returns several fields: abi, bin, and types. Among them, abi is the specific interface and parameter information called in the contract; bin is the binary bytecode file after the contract is compiled, and if there are multiple contracts in the source code, bin is the bytecode of the top-level contract; types stores the names of the relevant contracts compiled.

<compileCode> object.

```
{
  abi: <Array> abi array corresponding to the source code of the
  contract bin: Bytecode compiled from <Array> contract
  types: <Array> Name of the corresponding contract
}
```

Example 4 // request

```
{
  "jsonrpc": "2.0", "
    method": "contract_compileContract", "
    params": ["contract Accumulator{ uint sum = 0; function increment
() { sum = sum + 1; } function getSum() returns(uint) { return sum; }}"], "
    id":1}

    // return result
    {
  "jsonrpc": "2.0", "id": 1, "
```

Table 7.7 Deployment contracts

RPC methods	Parameters	Return value
contract_deployContract	{ from: <string> Contract Deployer Address timestamp: <number> transaction timestamp (in ns) payload: <string> Contract encoding contract_complieContract method returns a bin signature: <string> transaction signature }	transactionHash: <string> Hash of the transaction, 32-byte hexadecimal string

Description: If the contract constructor requires passing a parameter, the payload is a string splice of the bin returned by the compiled contract with the constructor parameter encoding

```

    result": {
"abi": ["[{"constant":false,"inputs":[],"name":"getSum",
"\outputs": [
    [{"name": "", "type": "uint256"}], "payable":false,
"\type": "function"}, {"constant":false,"inputs":[],"\name":
"\increment", "\outputs": [], "payable":false, "\type": "function"}]]", "
bin": ["0x60606040526000600060005055604a806
0186000396000f3606060405260e060020a600035
0463569c5f6d81146026578063d09de08a14603757
5b6002565b346002576000546060908152602090f35b346002
576048600080546001019055565b00"], "
    "types": ["Accumulator"
]
}
}
```

After the contract is compiled, the next step is to deploy the contract to the chain. Deployment to the blockchain requires that the contract can be deployed to each node of the blockchain by consensus. Table 7.7 shows the detailed parameter information for the contract deployment interface method, where from refers to the contract deployer's account address on the blockchain. It is worth noting that a contract deployment is also a transaction, similar to the sendTransaction interface, which returns a hash of that transaction. Suppose you want to confirm the result of the contract deployment further. In that case, you need to query the transaction details through the hash, which contains information about the success of the contract deployment and the deployment address of the contract, used to uniquely locate and when the contract is invoked.

Example 5 (Deployment Contract Invocation Example) // request

```

curl -X POST --data '{ "
jsonrpc": "2.0", "
method": "contract_deployContract ", "params": [{ "

```

Table 7.8 Call contracts

RPC methods	Parameters	Return value
contract_invokeContract	{ from: <string> contract caller address to: <string> contract address payload: <string> method name and method parameters encoded as input bytecode signature: <string> transaction signature timestamp: <number> Transaction timestamp }	transactionHash: <string> The hash of the transaction hash of the transaction, a 32-byte hexadecimal string

Description: The to contract address needs to be obtained by calling the tx_getTransactionReceipt method after the contract is deployed

Table 7.8 shows the interface details of the contract call, with two critical parameters—to and payload. The to is the address of the contract after the contract is compiled, indicating which specific contract will be called for the transaction. The payload is the encoded information about the specific function in the contract and its parameter values. Like a typical transaction call, the contract transaction call will only return a hash of the transaction, and the result of the transaction will also need to be queried by that hash in the form of a query transaction return.

```
Example 6 (Example of a Contract Call)           // request
    curl -X POST --data '{
        jsonrpc": "2.0",
        method": "contract_invokeContract",
        "params": [
            {
                "from": "0xb60e8dd61c5d32be8058bb8eb970870f07233155",
                "to": "0xe670ec64341771606e55d6b4
ca35a1a6b75ee3d5145a99d05921026d1527331",
                "payload": "
```

7.1.4 Block Queries

Data is stored and chained according to blocks, designed as a critical measure for blockchain platforms to keep their data easily tampered with. Block query interfaces provide methods for querying block information on the blockchain according to different rules. Through these query interfaces, users can perform functions such as querying the latest block information, querying blocks by block height, and querying blocks by time range.

Table 7.9 offers a partial list of interfaces for the block service structure.

Next, we will detail the block service structure using `block_getBlockByNumber` and `block_getPlainBlocks` as examples. `block_getBlockByNumber` interface method can query the information stored inside the corresponding block according to the block number. Table 7.10 provides a detailed description of this interface. The

Table 7.9 Block service interface overview table

RPC methods	Function
block_latestBlock	Get the latest blocks
block_getBlocks	Query all blocks in the specified block interval
block_getPlainBlocks	Query all blocks in the specified block interval, but do not contain information about the transactions in the block
block_getBlockByHash	Return block information based on the block's hash value
block_getBlockByNumber	Return block information based on block number
block_getAvgGenerateTimeByBlockNumber	Calculate the average block generation time based on the block interval
block_getBlocksByTime	Query the number of blocks in the specified time interval

Table 7.10 Querying block details based on block numbers

RPC methods	Parameters	Return value
block_getBlockByNumber	blockNumber: <blockNumber> block number	<blockResult>

return value `blockResult` contains the primary information stored in the block. The essential information is transactions, the detailed information of all the transactions stored in the block.

The `blockNumber` can be a decimal integer or a hexadecimal string, or the latest string, indicating the latest block.

`<blockResult>` object.

```
{
    version: <string> Platform version number
    number: <string> Height of the block
    hash: <string> Hash of the block, 32-byte hexadecimal string
    parentHash: <string> parent block hash, 32-byte hexadecimal string
    writeTime: <number> Block generation time (in ns)
        avgTime: <number> Average processing time of transactions in the current block (in ms)
        txCounts: <number> The number of transactions packed in the current block
        merkleRoot: <string> root hash of the Merkle tree
        transactions: [<TransactionResult>] List of transactions in the block
}
```

Example 7 (Example of Querying Block Information Based on Block Number)

```
// request
curl -X POST --data '{
    "jsonrpc": "2.0",
    "method": "block_getBlockByNumber",
    "params": ["0x3"],
    "id": 1
}' // return result
{
    "jsonrpc": "2.0",
    "id": 1,
    "code": 0,
    "message": "SUCCESS",
    "result": {
        "version": "1.0",
        "number": "0x3",
        "hash": "0x00acc3e13d8124fe799d55d7d2af06223
148dc7bbc723718bb1a88fead34c914",
        "parentHash": "0xb709670922de0dda68926f96cffbe48c98
0c4325d416dab62b4be27fd73cee9",
        "writeTime": 1481778653997475900,
        "avgTime": "0x2",
        "txcounts": "0x1",
        "merkleRoot": "0xc6fb0054aa90f3bfc78fe79cc459f7c7f268af7
eef23bd4d8fc85204cb00ab6c",
        "transactions": [
            {
                "version": "1.0",
                "hash": "0xf57a6443d08cda4a3dfb8083804b6334d17d7
af51c94a5f98ed67179b59169ae",
                "blockNumber": "0x3",
                "blockHash": "0x00acc3e13d8124fe799d55d7d2af06223148dc
7bbc723718bb1a88fead34c914",
                "txIndex": "0x0",
                "from": "0x17d806c92fa941b4b7a8ffffc58fa2f297a3bffc",
                "to": "0xaecd2fd1118334402c5de1cb014a9c192c498df",
                "amount": "0x0"
            }
        ]
}
```

Table 7.11 Querying all block interfaces for a given block interval

RPC methods	Parameters	Return value
block_getPlainBlocks	{ from: <blockNumber> Starting block number to: <blockNumber> Terminating block number }	[<plainBlockResult>]

Table 7.11 shows the details of an interface that queries for information about a block within a specified block interval according to that interval. The return value of this interface is a list of block information. Still, unlike the above example, the block information in this return value is only simple information about the block and does not contain complete information. The details are shown in Example 8.

The blockNumber can be a decimal integer or a hexadecimal string, and can be the latest string, indicating the latest block. from must be less than or equal to to; otherwise, it will return error.

< plainBlockResult> object.

```
{  
    version: <string> Platform version number  
    number: <string> Height  
    of the block  
    hash: <string> Hash of the block, 32-byte hexadecimal string  
    parentHash: <string> parent block hash, 32-byte hexadecimal string  
    writeTime: <number> Block generation time (in ns)  
    avgTime: <number> The average processing time (in ms) of the  
    transaction in the current block  
    txCounts: <number> The number of  
    transactions packed in the current block
```

```
    merkleRoot: <string> root hash of the Merkle tree
}
```

Example 8 (Interval Block Query Interface Example) // request

```
curl -X POST --data '{
  "jsonrpc": "2.0", "method": "block_getPlainBlocks", "params": [{"from":2, "to":3}], "id": 1}

// return result

{ "jsonrpc": "2.0", "id": 1, "code": 0, "message": "SUCCESS", "result": [
  {
    "version": "1.0", "number": "0x3", "hash": "0x0acc3e13d8124fe799d55d7d2af06223148dc7b
bc723718bb1a88fead34c914", "parentHash": "0xb709670922de0dda68926f96cffbe48c980c4325
d416dab62b4be27fd73cee9", "writeTime": 1481778653997475900,
"avgTime": "0x2", "txcounts": "0x1", "merkleRoot": "0xc6fb0054aa90f3bfc78fe79cc459f7c7f268af7eef23b
d4d8fc85204cb00ab6c" }

  ,
  {
    "version": "1.0", "number": "0x2", "hash": "0x2b709670922de0dda68926f96cffbe48c980c4
325d416dab62b4be27fd73cee9", "parentHash": "0xe287c62aae77462aa772bd68da9f1a1ba21a0d0
44e2cc47f742409c20643e50c", "writeTime": 1481778642328872960,
"avgTime": "0x2", "txcounts": "0x1"
    , "merkleRoot": "0xc6fb0054aa90f3bfc78fe79cc459f7c7f268af7eef
23bd4d8fc85204cb00ab6c"
  }
]
}
```

7.2 Platform Deployment

The Hyperchain blockchain platform employs RBFT, an improved algorithm based on PBFT, as the platform's consensus algorithm. This algorithm requires a minimum of four machines for the Hyperchain cluster to start properly. The configuration requirements for the Hyperchain platform deployment machines are shown in Table 7.12.

Table 7.12 Hyperchain basic configuration requirements table

Hardware and network facilities	Configuration requirements
Processors	Intel Xeon CPU E5-26xx v3 2 GHz, 4 cores and above
RAM	8 GB and above
Hard drive	1 TB HDD
Bandwidths	1000 Mbit/s

7.2.1 *Hyperchain Configuration*

Before officially deploying the Hyperchain system, its relevant parameters must be configured. After unpacking the Hyperchain installation package hyperchain.tar.gz, modify its main configuration file.

peerconfig.json: Configure Node IP and Port Number

Set the IP addresses of the four nodes and the gRPC communication port (default 8001) to ensure that all ports are available.

pbft.yaml: Configure the Consensus Algorithm

Modify the corresponding number of nodes Nodes (default value is 4), modify the batch size (default value is 500), indicating that 500 transactions will be packed into one block; modify timeout: batch size (default value is 100 ms), i.e., when the number of concurrent transactions is small and the batch size limit is not reached, the nodes pack a block every 100 ms.

global.yaml: Global Configuration

Please modify each configuration item according to your needs after understanding its meaning. If there is no special need, you can just use the default configuration.

LICENSE: Hyperchain Certificate of Authority

Issued by Hyperchain for legally authorized use for a limited period, the default LICENSE file can be overwritten if a more advanced certificate of authorization is obtained.

Other specific configuration information can be configured for specific choices based on corporate requirements, hardware environment, and other information.

7.2.2 *Hyperchain Deployment*

First, install the Hyperchain node, then change the network configuration file.

LICENSE Document

Since the LINCESE file and the Hyperchain installation package are distributed separately, you need to check whether the LICENSE file has been updated to the correct version before starting the node. The LICENSE file is located in the root directory of the Hyperchain node and is named LICENSE (please check all nodes in turn; if you are not sure if it is the latest version, you can overwrite it again with the original LICENSE file).

```
# Decompression
cd ~
tar xvf LICENSE-20180701.tar.gz
# After unpacking, the name of the LICENSE folder may be License-
20180701 # Update the LICENSE of all nodes
# Modify License-20180701/LICENSE-abcdef and /opt/hyperchain as
appropriate # The target file name of the copy command, which must be
LICENSE
cp License-20180701/LICENSE-abcdef /opt/hyperchain/LICENSE
```

Network Configuration File host.toml

```
hosts = [
  "node1 127.0.0.1:50011",
  "node2 127.0.0.1:50012",
  "node3 127.0.0.1:50013",
  "node4 127.0.0.1:50014"]
```

The configuration rule is simple: hostname ip_address:port only needs to configure the node name and IP address port for all nodes (port is for inter-node communication).

To modify: replace 127.0.0.1 in each line with the IP address of each of the four servers, and replace port 5001x in each line with each Hyperchain node's grpc port.

Network Configuration File addr.toml

```
# The domain name of the domain where this node is located domain =
"domain1"
# Address of this node accessed by nodes in other domains when they
connect to this node addrs = [
  "domain1 127.0.0.1:50011",
```

```

"domain2 172.16.100.112:50011",
"domain3 10.21.14.2:50011",
"domain4 220.11.2.54:50011"]
# It is important to note here that the configuration is configured
with this node's IP address that other nodes use when accessing this
node. For example, if node 2 belongs to domain domain2, then node 2 needs
to access node 1 with the address declared by node 1 exposed to the public
in the domain domain2; in other words, the address used by node 2 to access
this node is 172.16.100.112:50011.
# It should be noted that the number of domains here can be less than
the number of hosts self = "node1"

```

Logical Network Configuration File peerconfig.toml

```

[peer]
n = 4
hostname = "node1" vp = true
caconf = "config/namespace.toml" new = false
[[nodes]]
hostname = "node1" score = 10
[[nodes]]
hostname = "node2" score = 10
[[nodes]]
hostname = "node3" score = 10

[[nodes]]
hostname = "node4" score = 10

```

Global Configuration File global.toml

Usually, when we get the default configuration file, we only need to change the port subsection, which reads as follows.

```

[port]
jsonrpc = 8081
restful = 9001
websocket = 10001
jvm = 50081
ledger = 50051 grpc = 50011 # p2p

```

Note that after making changes to the above LICENSE file and the four configuration files, the configuration files for all nodes are also checked to avoid omissions due to a large number of nodes.

7.2.3 Hyperchain Operation

The Hyperchain platform is relatively simple to run. For a first-time Hyperchain platform, enter the hyperchain directory of each of the four Hyperchain nodes and run ./start.sh to start the corresponding Hyperchain service, and the successful sign is shown in Fig. 7.1.

When Hyperchain starts, the logs may show certain exceptions, the following are some startup exceptions and their handling options.

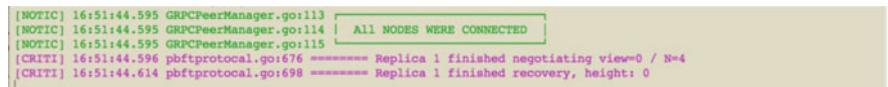
Communication Anomalies

When a single node has the error in Fig. 7.2, one of the nodes in the Hyperchain cluster is down, or there is a problem with network communication. If this happens, the node can be restarted using restart.sh script. As displayed in Fig. 7.2, the specific node with the problem can be located by its IP.

When multiple nodes have the above problem, you need to stop sending transactions: first shut down all nodes (stop.sh) and restart by running start.sh on all nodes.

ViewChange Exception

As shown in Fig. 7.3, when this message can be seen on all nodes and from the same node (e.g., node 2), this indicates an exception has occurred in node 2, triggering a ViewChange, at which point no additional processing is required. When node



```
[NOTIC] 16:51:44.595 GRPCPeerManager.go:113 [All NODES WERE CONNECTED]
[NOTIC] 16:51:44.595 GRPCPeerManager.go:114
[NOTIC] 16:51:44.595 GRPCPeerManager.go:115
[CRITI] 16:51:44.596 pbftprotocol.go:676 ====== Replica 1 finished negotiating view=0 / N=4
[CRITI] 16:51:44.614 pbftprotocol.go:698 ====== Replica 1 finished recovery, height: 0
```

Fig. 7.1 Hyperchain startup log



```
2016/12/20 19:51:56 grpc: addrConn.resetTransport failed to create client transport: connection error: desc = "transport: dial tcp 127.0.0.1:8002: getsockopt: connection refused"; Reconnecting to {"127.0.0.1:8002": <nil>}
[ERROR] 19:51:56.531 peer.go:68 cannot establish a connection
[ERROR] 19:51:56.531 gRPCPeerManager.go:137 Node: 127.0.0.1 : 8002 can not connect!
[ERROR] 19:51:56.531 gRPCPeerManager.go:107 Node: 127.0.0.1 : 8002 can not connect!
    rpc error: code = 14 desc = grpc: the connection is unavailable
```

Fig. 7.2 Hyperchain communication exceptions



```
Replica1 already has a viewchange message for view 3 from replica2
```

Fig. 7.3 Hyperchain ViewChange exception

2 sends 10 unsuccessful ViewChanges, recovery will be automatically triggered and finally be agreed upon.

When similar messages can be found in all nodes and keep coming, please shut down all nodes first (stop.sh) and start them one by one with start.sh.

Ignore Duplicate Exception

If the occasional message shown in Fig. 7.4 appears, it is normal and may be caused by excessive CPU load on a node or poor network communication.

If this situation keeps appearing, it is possible that a node is too slow to make the processing fall behind, and can be recovered by waiting for the node to discover it automatically (it takes a while to wait, and falling 50 blocks behind triggers a recovery), or by restarting the node falling behind.

7.3 The First Hyperchain Application

The previous section has introduced the services associated with the Hyperchain platform available to the public. This section will analyze how to build a blockchain application by the Hyperchain platform through a specific case of a simulated bank. The case requires the ability to simulate deposits and withdrawals within a bank and transfer operations between depositors. For this core financial business, all the operational processing of user asset-related data need to be operated and recorded on the blockchain. This case enables a smart contract to manage user assets, with deposit, withdrawal, and transfer logic implemented inside the smart contract.

7.3.1 Writing Smart Contracts

Hyperchain platform now supports Java language, Solidity language, and other multi-languages for smart contract development, and here Solidity language is used as an example for smart contract writing. Solidity is a smart contract writing language similar to JavaScript language. In Solidity, the contract consists of functions and data definitions. The abstract description of the user's assets is realized through the data definitions, and the rules for operating the user's digital assets are realized through the contract functions. Please refer to the Solidity syntax specification for details.

Replica2ignoringprepareforview=2/seqNo=10:notin-wv,inview1,lowwatermark20

Fig. 7.4 Hyperchain ignore duplicate exception

Each contract of a Solidity smart contract is modified with the contract keyword. This is similar to the concept of class in the Java language, where functions related to that contract's data and operator data can be defined inside the contract code. In this example, bank-related attributes are declared: bankName, bankNum, and the bank status field (isValid). In addition to this, the contract stores a contract creator address modified by an Address field, which records the creator's ability to implement permission control for contract calls. The chaincode is as follows.

```

contract SimulateBank{ address owner;
bytes32 bankName; uint bankNum;
bool isValid;
mapping(address => uint) public accounts;
function SimulateBank(bytes32 _bankName, uint _bankNum, bool
_isValid) { bankName = _bankName;
bankNum = _bankNum; isValid = _isValid;
owner = msg.sender;
}
function issue(address addr, uint number) return (bool) { if(msg.
sender == owner) {
accounts[addr] = accounts[addr] + number;
return true;
}
return false;
}
function transfer(address addr1, address addr2, uint amount)
return (bool) { if(accounts[addr1] >= amount) {
accounts[addr1] = accounts[addr1] - amount; accounts[addr2] =
accounts[addr2] + amount; return true;
}
return false;
}
function getAccountBalance(address addr) return(uint) { return
accounts[addr]; }
}

```

The above code employs a map structure to maintain the mapping relationship between users and their assets. The issue function implements the operation of user deposits, the transfer completes the operation of asset transfers between users, and the getAccountBalance function fulfills the function of querying user balances. This concludes the writing of the stimulated bank contract and then proceeds to describe how to compile, deploy, and invoke the contract.

7.3.2 Deployment and Contract Invocation

Smart contracts should be deployed and invoked through the consensus algorithm of the blockchain platform, enabling contracts to be deployed to the blockchain, and changes to the state of contract assets should be synchronized across multiple nodes. The Java SDK provided externally by Hyperchain enables easy smart contract

invocation. The following is an example of SimulateBank's issue method call using Hyperchain's Java SDK.

```

String contractSourceCode = "SimulateBank source code";
String ownerAddr =
"0x00acc3e13d8124fe799d55d7d2af06223148dc7bbc
723718bb1a88fead34c914"
// 1. Compile contracts
HyperchainAPI api = new HyperchainAPI(
    "http://localhost:8081", HttpProvider.INTERNET, false);
CompileReturn result = api.CompileContract
(contractSourceCode, 1);
// 2. Deploy contracts
SingleValueReturn deployRs = api.deployContract(
    ownerAddr, result.getBin().get(0), null, "password123", 1);
String contractAddr = api.getTransactionReceipt(
    deployRs.toString(), 1).getContractAddress();
// 3. Call contracts
FuncParamReal addr = new FuncParamReal(
    "address", "0x23acc3e13sd8124fe799d55d7d2af0
6223148dc7bbc7723718bb1a88fead34c914"); FuncParamReal number = new
FuncParamReal("uint", "10000");
String input = FuncParamReal.encodeFunction("issue", addr,
number); api.invokeContract(ownerAddr, contractAddr, input, null,
"password123", 1);

```

In the above code, we first instantiate an instance of HyperchainAPI, and then adopt that instance to compile, deploy, and invoke the contract. After the contract is compiled, it can only return a transaction hash that looks up the transaction result, such as the contract address in the transaction receipt obtained in the deployment contract in the above code. The contract address is a mandatory option for invoking the contract. In addition, the method to be called and its parameters need to be encoded before the method call is performed, and the specific contract call is eventually made through the invokeContract method.

7.4 Summary

This chapter introduces the content related to application development on the Hyperchain blockchain. First, it introduces the primary interfaces provided by the Hyperchain platform to the outside world in terms of transaction invocation, chaincode management, and block query; second, it explains how to build a runnable enterprise blockchain system Hyperchain with respect to the configuration, deployment, and operation of Hyperchain clusters; and finally, it tells how to use a mock bank as an example of Hyperchain platform for the development of smart contract applications.

Part IV

Blockchain Applications and Use Cases

Chapter 8

Ethereum Application and Case Studies



As a core technology that triggers a wave of disruptive revolution, the application of blockchain technology in the financial sector will potentially change the conventional transaction process and record-keeping methods, and thus significantly reducing costs and improving efficiency. Due to blockchain's security, transparency, and non-tampering characteristics, the trust model between financial systems will no longer rely on intermediaries, and many businesses will be "decentralized" to realize real-time digital transactions. Ethereum is the well-known Turing-complete open-source blockchain platform where all computable logic operations can be implemented based on Ethereum smart contracts. Numerous innovative blockchain applications and innovative products of startups are growing up on it. The previous chapters have reviewed the core principles and development practices of Ethereum in detail. This chapter will concentrate more on the actual procedure by introducing two Ethereum-based practical applications: generic points system and electronic coupon system. The generic points system develops DApp applications directly on the underlying platform of Ethereum and calls the methods of smart contracts through the web3.js interface provided by Ethereum to send transactions or read data and display them to users on web pages, while the electronic coupon system, the overall architecture of which is a Java Web Project with the SSM framework, introduces blockchain technology at the application layer, independently encapsulates the interface tools for interaction with the Ethereum platform, uses the platform and the database to cooperate and store data, and develops smart contracts to record the actual backend business logic as transactions on the blockchain.

8.1 Case Study of Ethereum-Based Generic Points System

To increase loyalty, reward points act as a marketing tool by banks, supermarkets, securities companies, etc. This traditional mechanism has disadvantages, such as many use restrictions, tedious redemption, and complex circulation, and therefore is

no longer suitable for consumption habits today. The system on blockchain technology aims to transfer points between different users and introduce offline merchants to offer abundant points for prizes and services. Many projects are based on Ethereum, one of the most prevalent underlying blockchain platforms. Combining the Ethereum platform with a bank points system is of practical significance.

8.1.1 Project Introduction

Blockchain, as a distributed database ledger technology that cannot be easily tampered with, stores data in every node of the network thus determining its security. Each user on the blockchain will have their private key, and each transaction will be signed by the private key, which can be stored only after being certified by the nodes on the whole network. Once reserved, it cannot be modified to ensure the circulation's security, which makes the design of points no longer impractical, significantly enhancing the user experience and user stickiness.

Its core business lies in the circulation of bank points. The straightforward process is described as follows: the bank can issue points to customers, and customers can transfer the points in their accounts to other customers or merchants, and in the meantime, they can purchase goods in the points mall. Merchants can issue commodities at the points mall and obtain corresponding points for each commodity sold. They can also initiate the point settlement to the bank and convert the points into currency. The system's overall flow chart is shown in Fig. 8.1.

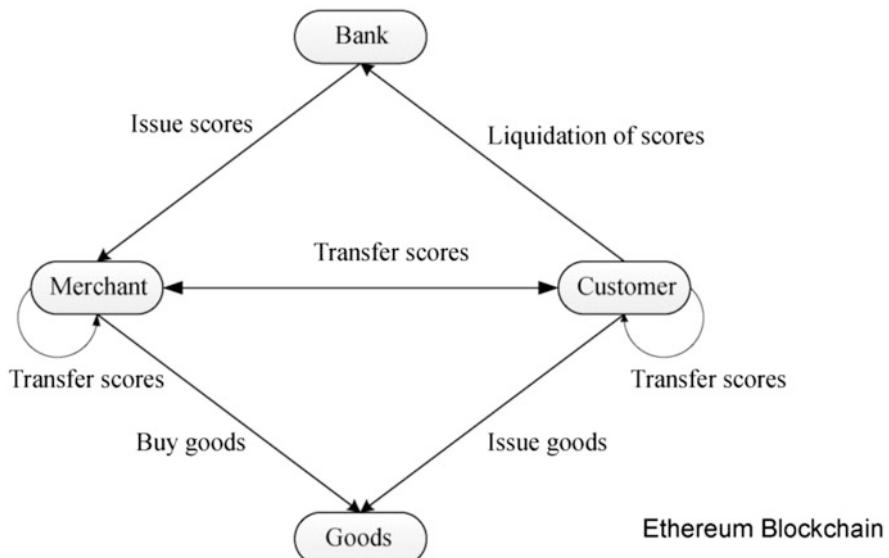


Fig. 8.1 The score system's overall flow chart

8.1.2 System Function Analysis

The system mainly involves three users: customers, merchants, and banks. Banks can interact with merchants directly and issue the points, and in return, merchants can initiate the settlement of the points to the bank. Points can also be now circulated between merchants and customers. Points can be transferred between customer-customer, merchant-merchant, and merchant-customer. At the same time, customers can purchase merchants' products, and as a result, the corresponding amount of scores will flow from the customer's account to the merchant's account. Different users can perform everyday inquiry operations. The specific functions for each user are displayed in Table 8.1.

8.1.3 General System Design

The overall design mainly includes solution selection and overall architecture design. The former includes the selection of the Ethereum client, the development

Table 8.1 Requirement points of the generic points system

Requirement points		Remarks
Customer	Register	Customer registration account
	Login	Customers log in to the points system
	Transfer scores	Transfer of points by customers to other users (customers or merchants)
	Redeem goods	Customers use points to redeem goods
	Query purchased items	Customers query the array of purchased items
	Query credits	Customers query score balance
Merchant	Register	Merchant registration account
	Login	MERCHANTS log in to the score system
	Transfer scores	Transfer of scores by merchants to other users (customers or merchants)
	Initiate liquidation	MERCHANTS settle points with banks
	Post a product	MERCHANTS post products
	Query posted products	MERCHANTS query on the array of posted products
Bank	Query scores	MERCHANTS query score balance
	Register	Bank registration administrator account
	Login	Banks log in to points system
	Issue scores	Banks issue points to customers
	Query issued scores	Banks query the total amount of scores issued
	Query settled scores	Banks query the total amount of scores settled with merchants

framework, and the Ethereum interface. The latter mainly refers to the design between the underlying blockchain platform and the upper layer business. An excellent overall system design guarantees the subsequent smart contract design and system implementation.

Solution Selection

Solution selection generally includes Ethereum client, development framework, and interface type.

Ethereum Client

There are two commonly applied types of Ethereum clients in the DApps now, Ganache and geth. This case can be run and deployed in Ganache (i.e., the original TestRPC) and geth simultaneously. However, Ganache is recommended for test development. Ganache is an Ethereum client on Node.js. The data of the whole blockchain resides in memory, and transactions sent to Ganache will be processed immediately without waiting for mining time. Ganache necessarily creates a test account with a pile of funds at startup, and it runs faster, making it more suitable for development and test. The command line interface of Ganache is as follows:

```
→ ~
ganache-cli
Ganache CLI v6.1.0 (ganache-core: 2.1.0)

Available Accounts
=====
(0) 0xfdबa43f72cc5a093e99db2cc7138f7308d4f19e8
(1) 0xa5d53b7b0a86638c3e267a2e4a7bc67bb6876f9b
(2) 0x0eae90fd6b146483e87e7d5a3576c77be7ddf4fb
(3) 0x266f2795b7dd6375b74198535bc86e7aeacd129
(4) 0x1c82f13f0ede4a870fdd06dde4591e643f226161
(5) 0xbbe8a9c67eb776421957b90c10ed59bea7705d21
(6) 0xcde95925e6843f694f5d3c2060b66d04060746c3
(7) 0x7826c380d0fd1bbd6db5e62e6cbd500f97b5d3dd
(8) 0x7d24e70c174992e3147a788961db20952f4ee8d6
(9) 0x37acb234a2df883b524b9c60241899af67abcd7a

Private Keys
=====
(0) a70bda70d68493eb4dea07c45dc213a170382916bc8c936
5f50937ce81d45afb
(1) a0285b66a67692ac5eddb2530de374ae5370d77a12
ba8b5295d90fcbb12dc53f8
(2) 1288b4166da62e7e9d470c2605ca02fb6a2be34c3a25492a2
ecd07d096099d9b
```

```
(3) cb436c744c12b5805ca6098e4ccb3302c45fc9fae5b37d100ce
178774cd621f4
(4) 0e0a5fe159e2982a9dbd737b898d906ef6cb478435
da599d31cadbf0457454b7
(5) 4aa7d0feaf9011a2bcae5dee8a24e206780a3aaa7ca
d1f2fa64a31674bb278af
(6) f40fdc6364a7a90c17a0d8add7e52061419caf5d063f45ca
60f49a67a9676317
(7) 4bd8f62190734395580c196ca0e6880a35863bdb4c0197
94c418aa89b36fa41
(8) f678ff9c401f8279be50a37da6499b9e1d379b190e8c9eabbc
1f7c22cecf5df3
(9) 3a0fbbed195feab0a2e992559e2625c2f0845ca56b1874b240
262221e667c1867

HD Wallet
=====
=====

Mnemonic: story random fox secret visit seek cook renew connect slice
isolate deal
Base HD Path: m/44'/60'/0'/0/{account_index}

Listening on localhost:8545
```

Development Framework

Truffle, applied in this case, is an Ethereum-based smart contract development tool that supports unit testing of contract code, ideal for test-driven development. It has a built-in smart contract compiler, which allows you to compile, deploy, and test contracts by simply using script commands, greatly simplifying the contract development lifecycle.

Ethereum Interface

Currently, Ethereum provides two interfaces, JSON-RPC and web3.js. Using the Truffle framework, we explore the web3.js interface by default. Truffle wraps a JavaScript Promise framework ether-pudding of web3.js, making it very convenient to tap JavaScript code to call methods in smart contracts asynchronously.

Overall Architecture

Figure 8.2 displays the system architecture of this case. The underlying leverage Ethereum blockchain, while Ganache is being applied locally to open Ethereum and deploy the smart contracts through the Truffle tool. The sources system calls

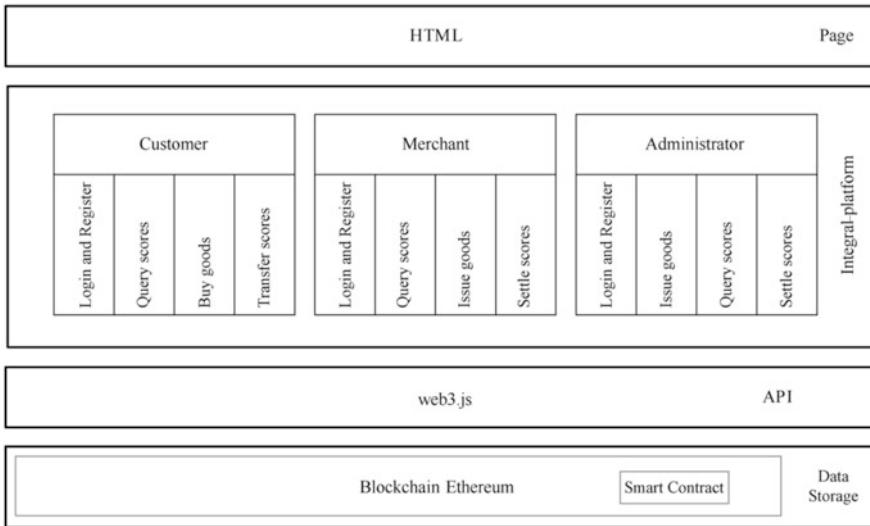


Fig. 8.2 The system architecture chart

methods in the smart contract by the web3.js interface. By efficiently making full use of the system's functions, users can tap the front-end page.

8.1.4 Smart Contract Design

Ethereum smart contracts can be written in several languages, such as Solidity, Serpent, and LLL. Solidity, at the current stage, is officially recommended. There are generally two design options for smart contracts. The first option is that one entity in the project corresponds to one contract, and therefore there may be more than one contract in the project; for example, three contracts are designed for customer entity, merchant entity, and bank entity, respectively, which is more in line with the object-oriented thinking. The other is to develop only one contract, in which different objects are stored utilizing structures and mappings. Relatively speaking, it is easier to understand the second option, simpler to test, and more convenient for subsequent expansion and maintenance; thus, the second option is applied in this case.

Tool Contract

The contract is constantly interacting with the front-end page in this case, which involves some data type conversion, and the front-end usually passes in the string type, yet the contract adopts Bytes32 more. Hence, the contract handles the conversion between string and Bytes32. Here we establish a tool class contract, to which

subsequent tool methods can be directly added, and the actual master contract inherits the tool class contract:

```
pragma solidity >=0.4.22 <0.6.0;
contract Utils {

    function stringToBytes32(string memory source) internal pure
returns (bytes32 result) {
        assembly {
            result := mload(add(source, 32))
        }
    }

    function bytes32ToString(bytes32 x) internal pure returns (string
memory result) {
        bytes memory bytesString = new bytes(32);
        uint charCount = 0;
        for (uint j = 0; j < 32; j++) {
            byte char = byte(bytes32(uint(x) * 2 ** (8 * j)));
            if (char != 0) {
                bytesString[charCount] = char;
                charCount++;
            }
        }
        bytes memory bytesStringTrimmed = new bytes(charCount);
        for (uint j = 0; j < charCount; j++) {
            bytesStringTrimmed[j] = bytesString[j];
        }
        return string(bytesStringTrimmed);
    }
}
```

Contract State Design

The objects in the contract involve customers, merchants, administrators, and products. Since there is only one master contract, we treat the administrator as the “owner” of the master contract and the state of the administrator as the public state.

```
address owner; //the owner of the contract, bank
uint issuedScoreAmount; // total amount of scores the bank issued
uint settledScoreAmount; // total amount of scores the bank settled
```

Customers, merchants, and goods use structs to encapsulate the properties of these objects. Customers own four attributes: account address, password, points balance, and an array of purchased goods. Merchants have four features: account address, password, points balance, and an array of issued goods. Goods have three attributes: ID, price, and merchant address to which they belong.

```

struct Customer {
    address customerAddr; // customer address
    bytes32 password; // customer password
    uint scoreAmount; // score balance
    bytes32[] buyGoods; //an array of purchased goods
}
struct Merchant {
    address merchantAddr; // merchant address
    bytes32 password; // merchant password
    uint scoreAmount; // score balance
    bytes32[] sellGoods; // an array of issued goods
}
struct Good {
    bytes32 goodId; // goods ID;
    uint price; // price;
    address belong; // which merchant address the goods belong to ;
}

```

Mapping should be established in the contract to allow the customers and merchants to be found by account address or items by ID. Solidity affords a way to find key-value pairs for this mapping.

```

mapping (address=>Customer) customer; // find a customer by their
address
mapping (address=>Merchant) merchant; // find a merchant by their
address
mapping (bytes32=>Good) good; // find an item by its product ID

```

Establish arrays of customers, merchants, and goods uniformly; store all registered or added objects.

```

address[] customers; // Array of registered customers
address[] merchants; // Array of registered merchants
bytes32[] goods; // Array of goods already online

```

Contract Method Design

The contract method is mainly designed for the methods provided externally in each function module, including the customer/merchant registration method, the approach to judge whether to register, the merchant/customer login method, the method to transfer scores, etc.

Constructors

Each contract will have a default constructor, which will be called if the contract is initialized. We can override the constructor to initialize the parameters. The case

takes the caller of the contract as the account address of the bank administrator and rewrites the constructor method as follows:

```
// Constructor
constructor() public {
    owner = msg.sender;
}
```

Customer/Merchant Registration

There are two types of smart contracts: transaction methods and constant methods. The transaction method modifies the state variables on the blockchain and generates an accurate transaction record on the block. The constant method is primarily intended to get a variable as an operation. It will not modify the variable or produce a transaction record on the block, and, naturally, the get method equals to the constant method. The register client method should be a transaction method and employ the event to return the value. By leveraging the web3.js interface, the transaction method cannot return the value directly by returns; instead, due to the transaction hash as the default value, we can only return the value by sending events using event. In contrast, since data in the constant method can be returned directly in the way of returns, the constant method usually does not write events. The customer/merchant registration is implemented as follows:

```
// Register a customer
event NewCustomer(address sender, bool isSuccess, string password);

function newCustomer(address _customerAddr, string memory
_password) public {
    // Determine if you are already registered
    if (!isCustomerAlreadyRegister(_customerAddr)) {
        // Not yet registered
        customer[_customerAddr].customerAddr = _customerAddr;
        customer[_customerAddr].password = stringToBytes32(_password);
        customers.push(_customerAddr);
        emit NewCustomer(msg.sender, true, _password);
        return;
    }
    else {
        emit NewCustomer(msg.sender, false, _password);
        return;
    }
}

// Register a merchant
event NewMerchant(address sender, bool isSuccess, string message);

function newMerchant(address _merchantAddr,
    string memory _password) public {
```

```

// Determine if you are already registered
if (!isMerchantAlreadyRegister(_merchantAddr)) {
    // Not yet registered
    merchant[_merchantAddr].merchantAddr = _merchantAddr;
    merchant[_merchantAddr].password = stringToBytes32(_password);
    merchants.push(_merchantAddr);
    emit NewMerchant(msg.sender, true, " Register successfully");
    return;
}
else {
    emit NewMerchant(msg.sender, false, " This account is
registered");
    return;
}
}

```

Judge Whether the Customer/Merchant Is Registered

Some methods that need to be called only inside the contract and are not visible to the external interface can be decorated with the internal keyword. Judgment shall be made before each registration to prevent duplicate registration of the same account of the customer/merchant, as follows.

```

// Determine if a customer are already registered
function isCustomerAlreadyRegister(address _customerAddr)
internal view returns (bool) {
    for (uint i = 0; i < customers.length; i++) {
        if (customers[i] == _customerAddr) {
            return true;
        }
    }
    return false;
}

// Determine if a merchant is already registered
function isMerchantAlreadyRegister(address _merchantAddr) public
view returns (bool) {
    for (uint i = 0; i < merchants.length; i++) {
        if (merchants[i] == _merchantAddr) {
            return true;
        }
    }
    return false;
}

```

Customer/Merchant Login

In this case, the logic to get the password of the login object using the smart contract method and determine whether the login is successful or not is performed in the JavaScript code. Solidity's methods can return multiple values directly by return. The method to obtain the password of the logged-in person in the contract is as follows:

```
// Query the user password
function getCustomerPassword(address _customerAddr) public view
returns (bool, bytes32) {
    // Check whether the user is registered
    if (isCustomerAlreadyRegister(_customerAddr)) {
        return (true, customer[_customerAddr].password);
    }
    else {
        return (false, "");
    }
}

// Query the merchant's password
function getMerchantPassword(address _merchantAddr) public view
returns (bool, bytes32) {
    // Check whether the merchant is registered
    if (isMerchantAlreadyRegister(_merchantAddr)) {
        return (true, merchant[_merchantAddr].password);
    }
    else {
        return (false, "");
    }
}
```

Issue Scores

In this case, the bank administrator can issue scores to any customer, and the amount of scores given is recorded in the issuedScoreAmount variable. The scores for customer increase by the corresponding amount. The method is implemented as follows:

```
// The bank sends scores to the customer, which can only be called by the
bank and sent to the customer
event SendScoreToCustomer(address sender, string message);

function sendScoreToCustomer(address _receiver,
    uint _amount) public onlyOwner {

    if (isCustomerAlreadyRegister(_receiver)) {
        // Registered
```

```

        issuedScoreAmount += _amount;
        customer[_receiver].scoreAmount += _amount;
        emit SendScoreToCustomer(msg.sender, "Release scores
successfully");
        return;
    }
    else {
        // Not yet registered
        emit SendScoreToCustomer(msg.sender, "The account is not
registered, failed to issue scores");
        return;
    }
}

```

Transfer Scores

Scores can be transferred between any two accounts which yet are completed through the same contract method. Since it is necessary to determine whether the caller is a customer or a merchant, the parameter `_senderType=0` indicates the points sender is a customer and `_senderType=1` means the score sender is a merchant. The method is implemented as follows:

```

// Transfer scores between two accounts, which can be done between any
two accounts; both the customer and merchant call this method
// _senderType means the caller type, 0 means customer, and 1 means
merchant
event TransferScoreToAnother(address sender, string message);

function transferScoreToAnother(uint _senderType,
address _sender,
address _receiver,
uint _amount) public {

    if (!isCustomerAlreadyRegister(_receiver) && !
isMerchantAlreadyRegister(_receiver)) {
        // Destination account does not exist
        emit TransferScoreToAnother(msg.sender, "Destination account
does not exist; please confirm before transferring!");
        return;
    }
    if (_senderType == 0) {
        // Customer transfer
        if (customer[_sender].scoreAmount >= _amount) {
            customer[_sender].scoreAmount -= _amount;

            if (isCustomerAlreadyRegister(_receiver)) {
                // destination address is the customer
                customer[_receiver].scoreAmount += _amount;
            } else {
                merchant[_receiver].scoreAmount += _amount;
            }
        }
    }
}

```

```

        }
        emit TransferScoreToAnother(msg.sender, " Scores transfer
succeeded!");
    return;
} else {
    emit TransferScoreToAnother(msg.sender, " Your scores balance
is insufficient; transfer failed!");
    return;
}
} else {
// Merchant transfer
if (merchant[_sender].scoreAmount >= _amount) {
    merchant[_sender].scoreAmount -= _amount;
    if (isCustomerAlreadyRegister(_receiver)) {
        // destination address is the customer
        customer[_receiver].scoreAmount += _amount;
    } else {
        merchant[_receiver].scoreAmount += _amount;
    }
    emit TransferScoreToAnother(msg.sender, " Scores transfer
succeeded!");
    return;
} else {
    emit TransferScoreToAnother(msg.sender, " Your scores balance
is insufficient; transfer failed!");
    return;
}
}
}

```

Post Goods

A merchant can add an item to the contract, each identified by an ID. The same ID cannot be added repeatedly. The added goods will add objects using mapping and add them to the sellGoods array of merchant attributes. The method is implemented as follows:

```
// Merchant adds an item
event AddGood(address sender, bool isSuccess, string message);

function addGood(address _merchantAddr, string memory _goodId, uint
_price) public {
    bytes32 tempId = stringToBytes32 (_goodId);

    // First, determine if the item Id already exists
    if (!isGoodAlreadyAdd(tempId)) {
        good[tempId].goodId = tempId;
        good[tempId].price = _price;
        good[tempId].belong = merchantAddr;
```

```

        goods.push(tempId);
        merchant[_merchantAddr].sellGoods.push(tempId);
        emit AddGood(msg.sender, true, "Add goods successfully");
        return;
    }
    else {
        emit AddGood(msg.sender, false, "This item has been added; please
confirm the operation");
        return;
    }
}

```

Purchase Goods

The customer can enter the product ID to buy an item. If the score amount owned is greater than or equal to the items required, the purchase is successful; otherwise, it fails. After the successful purchase, the item ID will be added to the customer's buyGoods array. The method is as follows:

```

// Users buy an item with scores
event BuyGood(address sender, bool isSuccess, string message);

function buyGood(address _customerAddr, string memory _goodId)
public {
    // First determine if the item Id entered exists
    bytes32 tempId = stringToBytes32(_goodId);
    if (isGoodAlreadyAdd(tempId)) {
        // This item has been added and is available for purchase
        if (customer[_customerAddr].scoreAmount < good[tempId].price) {
            emit BuyGood(msg.sender, false, "Insufficient balance, failed to
buy goods");
            return;
        }
        else {
            // For the methods extracted here
            customer[_customerAddr].scoreAmount -= good[tempId].price;
            merchant[good[tempId].belong].scoreAmount += good[tempId].
price;
            customer[_customerAddr].buyGoods.push(tempId);
            emit BuyGood(msg.sender, true, "buy the goods successfully");
            return;
        }
    }
    else {
        // There is no such Id
        emit BuyGood(msg.sender, false, "The goods Id entered does not
exist; please confirm and buy");
        return;
    }
}

```

8.1.5 System Implementation

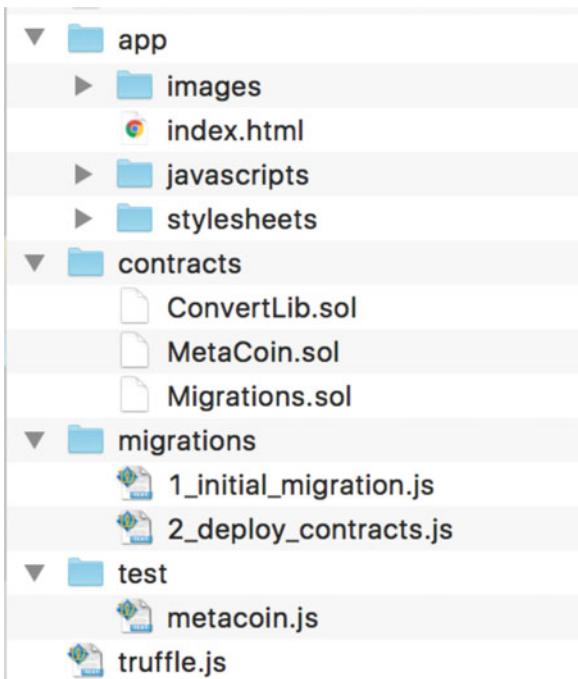
Above there is the general design and smart contract design we performed. The following is the system implementation. The project is mainly built with Truffle, and the detailed implementation is done after the contract method design in Sect. 8.1.4, using web3.js interface implementation to align with the contract methods, corresponding to the method interfaces in the plan.

Establish Projects

This case is built by the Truffle framework. First, it asks for a new folder, then uses the terminal command line to enter and execute the Truffle unbox Webpack command, when a truffle-based decentralized Ethereum application will be automatically built. The directory structure after the established project is shown in Fig. 8.3.

The app folder contains front-end pages and JavaScript code, and the main code we implemented will be in the javascripts folder; the contracts folder contains smart contracts. Truffle generates three contracts by default, and our implementation goes into this folder as well. The migrations folder concerns contract deployment configuration and the 2_deploy_schemes.js file needs to be modified before contracts can be deployed to Ethereum; the test folder is used to write the smart contract test code,

Fig. 8.3 Directory structure for project establishment



Truffle.js for configuring Ethereum networks, and Webpack.config.js for configuring JS files and HTML files related to front-end engineering.

Detailed Implementation

It calls for JavaScript code to be written, including connecting to Ethereum, customer/merchant registration interface, customer/merchant login interface, score-transferring interface, etc.

Connecting to Ethereum

The application requires an instance of the contract deployed on Ethereum at page launch before it can invoke the methods in the contract. Since Truffle has integrated web3.js interface by default, we can directly employ all the methods under web3.eth. First of all, we listen to the page load event and call the init method of window.App to get the available accounts and contract instances on Ethereum when the page loads. The implementation in App.js is as follows:

```
window.App = {
  // Get contract instances
  init: function () {
    // Set up web3 connection
    ScoreContract.setProvider(window.web3.currentProvider)
    // Get the initial account balance for display
    window.web3.eth.getAccounts(function (err, accs) {
      if (err != null) {
        window.App.setStatus('There was an error fetching your
accounts.')
        return
      }

      if (accs.length === 0) {
        window.App.setStatus('Couldn\'t get any accounts! Make sure your
Ethereum client
          is configured correctly.')
        return
      }
      accounts = accs
      account = accounts[0]
    })

    ScoreContract.deployed().then(function (instance) {
      ScoreInstance = instance
    }).catch(function (e) {
      console.log(e, null)
    })
  },
}
```

```

    ...
}

window.addEventListener('load', function () {
  // Set up a web3 connection to http://127.0.0.1:8545
  window.web3 = new Web3(new Web3.providers.HttpProvider('http://
127.0.0.1:9545'))
  window.App.init()
})

```

Customer/Merchant Registration

The registration page locates on the main page index.html, where customers and merchants can register an account by entering their legitimate Ethereum account address and password. The front-end HTML code for registration is implemented as follows.

```

<div class="block">
  <h3>Customer registration</h3>
  <label for="customerAddress">Customer account</label><input
  type="text" id="customerAddress"
  placeholder="e.g., 0x93e66d9baea28c17d9fc393b53e3fbdd76899dae"/>
  <br/><label for="customerPassword"> password </label><input
  type="text" id="customerPassword"
  placeholder="e.g., 123456"/>
  <button onclick="App.newCustomer () ">Customer registration</
  button>
</div>

```

The simple front-end page for customer registration is displayed in Fig. 8.4. The implementation of the logic code in the app.js file is as follows.

```

// Register a customer: to specify gas; the default gas value will
appear out of gas
newCustomer: function (ScoreInstance, account) {
  const address = document.getElementById('customerAddress').value
  const password = document.getElementById('customerPassword').value
  console.log(address + ' ' + password)

```

Customer address:

e.g., 0x93e66d9baea28c17d9fc393b53e3fbdd76899dae

Password:

e.g., *****

Customer register

Fig. 8.4 The front-end page for customer registration

```

ScoreInstance.newCustomer(address, password, { from: account, gas: 3000000 }).then(function () {
    // Invoke event
    ScoreInstance.NewCustomer(function (e, r) {
        if (!e) {
            console.log(r)
            console.log(r.args)
            if (r.args.isSuccess === true) {
                window.App.setStatus(' Register successfully')
            } else {
                window.App.setStatus(' Account already registered')
            }
        } else {
            console.log(e)
        }
    })
})
})
}

```

Customer/Merchant Login

Likewise, the login operation is performed on the index.js main page, where registered customers/merchants can successfully log in by entering the correct account password and jumping to the customer page or merchant page. The front-end HTML code for login is implemented as follows.

```

<div class="block">
    <h3> customer login</h3>
    <label for="customerLoginAddr"> customer address</label><input type="text" id="customerLoginAddr" placeholder="e.g., 0x93e66d9baea28c17d9fc393b53e3fbdd76899dae">
    <br><label for="customerLoginPwd"> password</label><input type="text" id="customerLoginPwd" placeholder="e.g., *****">
    <button id="customerLogin" onclick="App.customerLogin()"> customer login</button>
</div>

```

The brief front-end page for customer login is shown in Fig. 8.5.

Implement the logic code in app.js to discover the specified account's password on the blockchain according to the input account address and compare it with the input password to see if it matches. The code implementation is as follows.

```

customerLogin: function (ScoreInstance, account) {
    const address = document.getElementById('customerLoginAddr').value
    const password = document.getElementById('customerLoginPwd').value
    ScoreInstance.getCustomerPassword(address, { from: account, gas: 3000000 }).then(function ()

```

Customer address:

```
e.g., 0x93e66d9baea28c17d9fc393b53e3fbdd76899dae
```

Password:

```
e.g., *****
```

Customer login

Fig. 8.5 The front-end page for customer login

```
(result) {
if (result[0]) {
    // Check password successfully
    if (password.localeCompare(utils.hexCharCodeToStr(result[1])) === 0) {
        console.log(' Login suceeded')
        // Jump to user interface
        window.location.href = 'customer.html?account=' + address
    } else {
        console.log(' Wrong password, login failed')
        window.App.setStatus(' Wrong password, login failed')
    }
} else {
    // Password check failed
    console.log(' The user does not exist, please confirm your account and log in again!!!')
    window.App.setStatus(' The user does not exist, please confirm your account and log in again! ')
}
})
```

Issue Scores

In this case, the administrator can log in directly using the address of the contract caller. The first address displayed on the following command line interface is the account that invoked the current contract and is the default administrator account for this case. A different test account is created each time Ganache is started. You can use this account to log in to the administrator page.

```
→ ~
ganache-cli
Ganache CLI v6.1.0 (ganache-core: 2.1.0)
```

```
Available Accounts
=====
(0) 0x13f1cc81ce166ef8d14047e21ad347cdffdf5224
```

Customer address:

e.g., 0x93e66d9baea28c17d9fc393b53e3fbdd76899dae

Amount of scores:

e.g., *****

Issue scores

Fig. 8.6 The front-end page of the bank issuing scores

After administrator login, it will jump to the admin page bank.html. Afterward, every time you have a new html file, you have to modify the plugins in the webpack.config.js file as follows.

```
plugins: [
  new CopyWebpackPlugin([
    { from: './app/index.html', to: 'index.html' }
  ]),
  new CopyWebpackPlugin([
    { from: './app/customer.html', to: 'customer.html' }
  ]),
  new CopyWebpackPlugin([
    { from: './app/bank.html', to: 'bank.html' }
  ]),
  new CopyWebpackPlugin([
    { from: './app/merchant.html', to: 'merchant.html' }
  ])
]
```

The bank.html issuing score code is implemented as follows.

```
<div class="block">
  <h3> issue scores </h3>
  <label for="customerAddress">customer address </label><input
  type="text" id="customerAddress"
  placeholder="e.g., 0x93e66d9baea28c17d9fc393b53e3fbdd76899dae">
  <br><label for="scoreAmount"> amount of scores</label><input
  type="text" id="scoreAmount"
  placeholder="e.g., *****">
  <br>
  <button onclick="App.sendScoreToCustomer()"> issue scores </button>
</div>
```

A simple front-end page of the bank issuing scores is shown in Fig. 8.6. The method the bank.js issue scores is implemented as follows.

```

sendScoreToCustomer: function (ScoreInstance, account) {
  const address = document.getElementById('customerAddress').value
  const score = document.getElementById('scoreAmount').value
  ScoreInstance.sendScoreToCustomer(address, score, { from: account
})
  ScoreInstance.SendScoreToCustomer(function (e, r) {
    if (!e) {
      console.log(r.args.message)
      window.App.setStatus(r.args.message)
    }
  })
}

```

Transfer Scores

After successful login, customers or merchants can transfer scores within their credit limit. If the amount of scores to be transferred exceeds the existing limit, it will fail. This operation can be performed between two parties: customer-customer, customer-merchant, and merchant-merchant. Here is an example of the implementation in customer, with a new customer.js and customer.html.

```

<div class="block">
  <h3>Transfer scores </h3>
  <label for="anotherAddress"> Transfer address </label><input
type="text" id="anotherAddress"
  placeholder="e.g., 0x93e66d9baea28c17d9fc393b53e3fbdd76899dae">
  <br><label for="scoreAmount"> amount of scores </label><input
type="text" id="scoreAmount"
  placeholder="e.g., *****">
  <br>
  <button onclick="App.transferScoreToAnotherFromCustomer
(currentAccount)"> transfer scores </button>
</div>

```

The score-transfer page is basically the same as the page for bank scores issuing. The customer.js transfer scores are implemented as follows.

```

// customers can transfer scores at will
transferScoreToAnotherFromCustomer: function (currentAccount,
ScoreInstance, account) {
  const receivedAddr = document.getElementById('anotherAddress').value
  const amount = parseInt(document.getElementById('scoreAmount').value)
  ScoreInstance.transferScoreToAnother(0, currentAccount,
receivedAddr, amount, { from:
  account })
  ScoreInstance.TransferScoreToAnother(function (e, r) {
    if (!e) {

```

```
        console.log(r.args)
        window.App.setStatus(r.args.message)
    })
}
```

Post Goods

Logged-in merchants can post multiple goods with different IDs that cannot be duplicated. The goods will specify the amount of scores to be purchased. After successful posting, the ID will be added to the merchant's purchased items array. We will set the gas value in the parameters in some methods. When executing the transaction method, it will come with a default gas. Yet, if this method is more extensive with more code, resulting in a low default gas value, an out of gas error will occur, causing the transaction method to fail. In this example, some ways define the gas value explicitly. The following code is implemented in merchant.html and merchant.js, respectively.

merchant.html:

add goods0

goods ID

goods price

merchant.js:

```
// merchants add an item
addGood: function (currentAccount, ScoreInstance, account) {
  const goodId = document.getElementById('goodId').value
  const goodPrice = parseInt(document.getElementById('goodPrice').value)
  ScoreInstance.addGood(currentAccount, goodId, goodPrice, { from: account, gas:
    2000000 }).then(function () {
    ScoreInstance.AddGood(function (error, event) {
      if (!error) {
        console.log(event.args.message)
        window.App.setStatus(event.args.message)
      }
    })
  })
}
```

```
    })  
}
```

Purchase Items

Customers are capable of buying an item by inputting the product ID within their credit limit. If the scores required are greater than the customer's credit balance, it will fail. Once successful, the item will be added to the purchased items array. The following code is implemented in customer.html and customer.js, respectively.

Customer.html:

```
<div class="block">  
  <h3> buy goods</h3>  
  <label for="goodId"> buy goods Id</label><input type="text"  
  id="goodId">  
  <br>  
  <button id="buyGood" onclick="App.buyGood(currentAccount)"> buy  
  goods</button>  
</div>
```

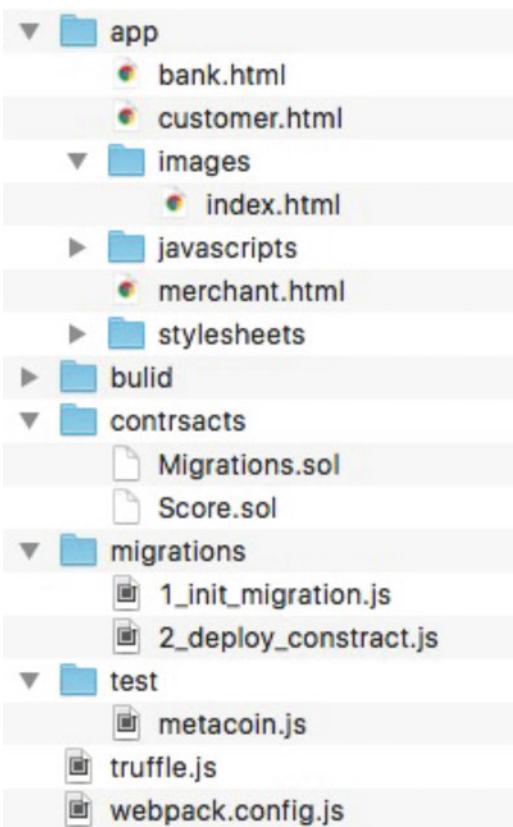
Customer.js:

```
// customers buy goods  
buyGood: function (currentAccount, ScoreInstance, account) {  
  const goodId = document.getElementById('goodId').value  
  ScoreInstance.buyGood(currentAccount, goodId, { from: account, gas:  
  1000000 }).then(function () {  
    ScoreInstance.BuyGood(function (error, event) {  
      if (!error) {  
        console.log(event.args.message)  
        window.App.setStatus(event.args.message)  
      }  
    })  
  })  
}
```

8.1.6 System Deployment

After we initialize the project by the truffle unbox webpack command earlier, some unneeded sample code that can be deleted manually will be generated automatically, such as the ConvertLib.sol and MetaCoin.sol files under the contracts/folder, but the Migrations.sol file will be kept. This file required during deployment corresponds to 1_initial_migration.js in the migrations/folder. We put all the html files directly into the app/folder, all the JavaScript files into the app/javascripts folder, and the written

Fig. 8.7 Project completion directory structure



smart contracts into the contracts folder with the sol suffix. The completed project directory after this case is shown in Fig. 8.7.

Project Configuration

A completed project calls for a simple deployment configuration, including the contract configuration and the overall configuration, where the latter means the design of HTML pages and JavaScript scripts.

Contract Configuration

Contracts must be compiled and deployed to an Ethereum client such as Ganache. migrations/2_deploy_contracts.js, viewed as the contract deployment file, needs to have our target contract configured into it (the code for the default contract has been removed). 2_deploy_contracts.js configuration information is as follows.

```

var Score = artifacts.require("./Score.sol");
module.exports = function(deployer) {
  deployer.deploy(Score);
};

```

Overall Configuration

In view of established multiple HTML files, we have to register them in webpack.config.js in the project root directory as follows:

```

plugins: [
  new CopyWebpackPlugin([
    { from: './app/index.html', to: 'index.html' }
  ]),
  new CopyWebpackPlugin([
    { from: './app/customer.html', to: 'customer.html' }
  ]),
  new CopyWebpackPlugin([
    { from: './app/bank.html', to: 'bank.html' }
  ]),
  new CopyWebpackPlugin([
    { from: './app/merchant.html', to: 'merchant.html' }
  ])
]

```

The Ganache client employs 8545 ports by default; however, they may correspond to different hosts and ports if we shift to other clients. Therefore, different Ethereum networks can be configured in truffle.js and the corresponding network can be selected with the -network parameter when running the deployment command. The truffle.js configuration is as follows.

```

require('babel-register')

module.exports = {
  networks: {
    truffle: {
      host: '127.0.0.1',
      port: 9545,
      network_id: '*'
    },
    develop: {
      host: '127.0.0.1',
      port: 8545,
      network_id: '*'
    }
  }
}

```

Project Deployment

Truffle project can be completed by a simple command line to compile the contract, deploy the contract to Ethereum and the whole project to the local server, and then make use of the front-end page to realize the interaction with the blockchain. Before deployment, you have to open Ganache service in one command line terminal, and then enter the project root directory in another terminal command line, executing the following commands.

- **truffle compile:** If there is a syntax error in compiling a smart contract, the compile will fail with an error message. The project has multiple contracts, but if you want to compile all of them at the same time after modifying only one, you can execute the trufflecompile --compile-all command.
- **truffle migrate:** Deploy the compiled smart contract to the Ethereum client Ganache.

If the command fails due to an exception, probably because it has been done before, you can execute the truffle migrate

- reset command, or if you have configured multiple networks in truffle.js, execute truffle migrate -network name
- reset for selective deployment, where name is the develop and truffle network configured in truffle.js.

- **npm run dev:** Open Truffle local server, employing port 8080 by default, and automate the deployment of the project to the Truffle server.

The command line interface of the deployment steps is implemented as follows.

```
→ Ethereum-Score-Hella git: (master) ✘ truffle compile --compile-all
Compiling ./contracts/Migrations.sol...
Compiling ./contracts/Score.sol...
Writing artifacts to ./build/contracts

→ Ethereum-Score-Hella git: (master) ✘ truffle migrate --network
develop --reset
Using network 'develop'.

Running migration: 1_initial_migration.js
Deploying Migrations...
... 0xc08c61fa40122887f66f7cc14ebdd73f03bed
e29c3308d8069cc7c31d37f2342
Migrations: 0x160d5c8a432fb2880cd8c66f8537f08d5d05868a
Saving successful migration to network...
... 0x81985c1bd8fff18f2231f2c220a5618b086f5528eb
0189768b2a406933c8d7d2
Saving artifacts...
Running migration: 2_deploy_contracts.js
Deploying Score...
... 0x34b2ca22fa2025188f0036b86093d35922160a38fbe9
fb8b8ae2c1dc9964fc1a
```

```
Score: 0xd88cd5030af17dcaed3d088d9cd160457e1da7e0
Saving successful migration to network...
... 0x07caa3f1eeb89cac93638872a34b0c4bfdeb5817
05ccd3707419732e1193d9e7
Saving artifacts...
→ Ethereum-Score-Hella git:(master) ✘ npm run dev

> truffle-init-webpack@0.0.2 dev /Users/username/truffleProjects/
Ethereum-Score-Hella
> webpack-dev-server

Project is running at http://localhost:8080/
```

Meanwhile, in another terminal with Ganache turned on, a blockchain log printed out will include information about the method called, the transaction hash, the block number, the value of gas spent, etc. The command line log printed by Ganache is as follows.

```
Listening on localhost:8545
net_version
eth_accounts
eth_accounts
net_version
net_version
eth_sendTransaction

Transaction: 0xc08c61fa40122887f66f7cc14ebdd73f03bede29
c3308d8069cc7c31d37f2342
Contract created: 0x160d5c8a432fb2880cd8c66f8537f08d5d05868a
Gas usage: 277462
Block Number: 1
Block Time: Mon Jun 18 2018 12:08:25 GMT+0800 (CST)
```

After completing the above steps, enter “<http://localhost:8080>” in your browser to access the points system.

8.2 Case Study of an Ethereum-Based E-Coupon System

This project applies blockchain technology to an electronic coupon system, aiming to simplify the coupon circulation process with the help of blockchain technology, eliminate third-party platforms, and realize merchants' true autonomy in issuing universal coupons. By employing the Ethereum platform to build a federation chain with banks and merchants as nodes, the system implements the business logic through the developed smart contracts. The system effectively utilizes the P2P communication, disintermediation, and non-tampering characteristics of blockchain

technology, giving a new solution for the financial consumption application of “coupons.”

8.2.1 Project Introduction

Electronic coupons are a widely used financial promotion tool in the market, playing a more prominent role in enhancing merchants' awareness and sales growth, as well as bringing convenience and benefits to consumers. However, electronic coupons issued by merchants are on the whole limited by expiration date and amounts, lacking flexible circulation capability. Applying blockchain technology to universal electronic coupons would stimulate the initiative of merchants to a greater extent, reducing the necessity to rely on a central platform and pay additional commissions or draws. Electronic coupons bring users a greater sense of control with coupons in their hands, and the value of coupons increases. In addition, the application in this field can liberate the data of merchants and users from the data monopoly of Internet giants and realize absolute data freedom and data transparency.

Blockchain-based electronic coupons are applied to identify the authenticity of transaction commitments. The consensus mechanism on blockchain technology, coupled with the issuance, circulation, and use of electronic coupons, does not require a central institution. Instead, it can prevent fraudulent events and provide merchants with more convenient, autonomous, efficient, and low-cost services.

In this case, blockchain technology is primarily employed for authorization and bookkeeping, and smart contracts record the system data. In contrast, for the management of other data in the system, the database still plays a role of a storage tool. The bank, both acting as the participant and the administrator, handles the deployment and maintenance of the database; thus, the system is not decentralized in the complete sense. However, the use of blockchain technology to record the business in the system as a transaction on the blockchain, and the data related to the core functions of the system in the database can be authenticated by blockchain technology traceability, which also effectively reflects the advantages of applying blockchain technology to this case.

The core business of this system is the circulation of settlement coupons and coupons, and the quick process is as follows: the bank approves the merchant's application for settlement coupons and issues the corresponding amount; the merchant issues coupons based on the balance of the settlement coupons and grants them to the consumers; consumers spend their coupons to pay the merchant when they make a purchase, and they can also transfer coupons to others. The flow chart of the system as a whole is shown in Fig. 8.8.

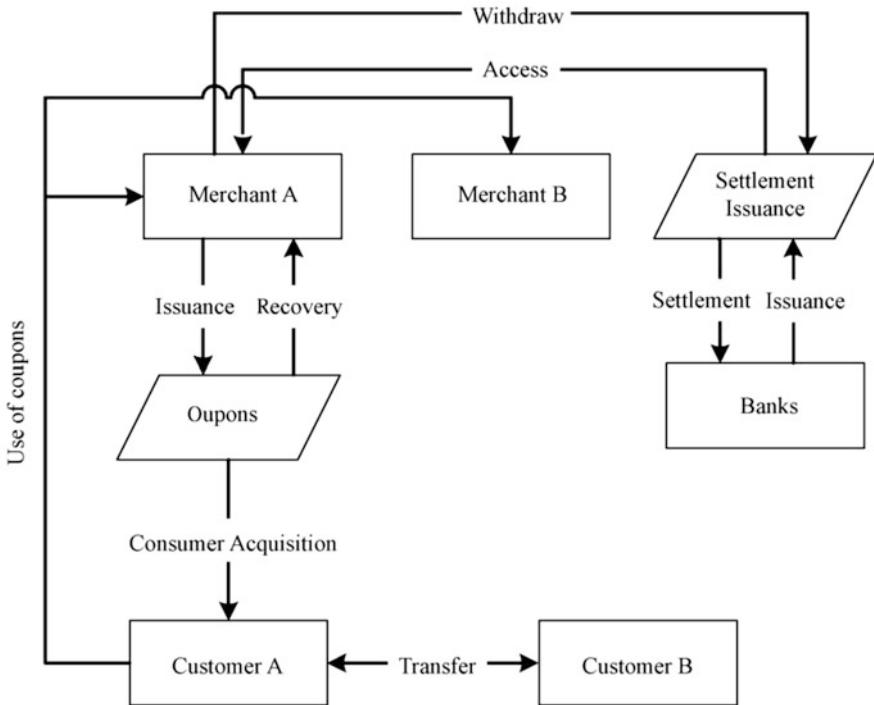


Fig. 8.8 Flow chart of universal coupon system

8.2.2 System Functional Analysis

The system involves three kinds of users, among which banks and merchants interact directly, employing settlement coupons as the interactive medium, while merchants and consumers interact directly, using coupons as a medium of interaction. In addition, the bank acts as an administrator to monitor and manage users in the system. The specific functions of each user are shown in Table 8.2.

8.2.3 General System Design

The overall design includes the designs in scheme, architecture, and underlying data storage. The scheme design specifies the hierarchical structure and the interaction design between the layers; the architectural design analyzes the functional architecture of the system; in addition, since this system employs database and blockchain together as the underlying data storage tool, the underlying data storage design clarifies the boundary of data storage, i.e., which data are stored in the blockchain

Table 8.2 Functional modules of universal electronic coupon system

Requirement points		Remarks
Customer	Register	Consumer registration account
	Log in	Consumers log in to the system
	Apply for coupons	After consumption, consumers apply for coupons from merchants
	Pay with coupons	Consumers apply to merchants to use the existing coupons
	Check coupon wallet	Consumers search their coupon wallets
	Transfer coupons	Consumers transfer coupons to each other
Merchant	Register	Merchant registration account
	Log in	Merchant login score system
	Apply for settlement coupons	Merchants apply for settlement coupons as the base amount for issuing coupons
	Withdraw settlement coupons	Merchants apply for settlement coupons as the base amount for issuing coupons
	Check out turnover	Merchants check the application and withdrawal turnover
Merchant	Issue coupons	Merchants set their coupon issuance rules and quantities
	Grant coupons	Merchants agree to the consumer's request for coupons and issue the coupons to consumers
	Approve consumer coupon payment application	Merchants approve the application for the coupon's use by consumers
	Terminate issuance	Merchants terminate the coupon issuance
	Check issue status	Merchants check the current coupon issuance status
Bank	Register	Bank registration administrator account
	Log in	Bank login management system
	Examine and review applications for settlement coupons	Approve or reject merchants' application for settlement coupons, and issue equal amount in case of unified application
	Approve merchant registration	Banks approve the merchant's application for joining the chain
	Consumer management	Banks can freeze or unfreeze consumer accounts according to their behavior
	Coupon query	Query all coupons in the system

and which in the database. The analysis of the above three contributes together to the basic framework of the system.

Scheme Design

This system that utilizes the JavaWeb system for business logic processing and integrates with SSM framework introduces blockchain technology in the application layer to confirm authority and realize the bookkeeping function. Using geth client to

access the Ethereum platform, the system builds web3.js-style interface to call Java background application layer through tool classes and interacts with Ethereum platform. The internal tool class employs the Jersey framework to communicate with the Ethereum client through JSON-RPC. This system builds the Ethereum private chain environment for development and test on the server. Private link data is stored in the specified file directory on the server (see Sect. 8.1.6).

System Architecture Design

The system applies the Ethereum platform in the bottom layer, and the nodes access the blockchain network through the geth client, design their smart contracts to deploy on the blockchain, and package their Web3 tool classes in the application layer to provide web3.js-style interfaces (that can be called directly in the application). By applying the JSON-RPC interface of the Ethereum client, it can interact with the geth client in the Web3 tool class.

Settlement coupons and coupons are the interactive media of a core business, and the related operations of the two are written into the blockchain as transactions to confirm rights. Therefore, the smart contract of the system should cover the application, issuance, and withdrawal of settlement coupons and the issuance, distribution, and use of the coupons. Due to the non-tamper-evident feature, the data that can reflect the current state should be read from the blockchain.

In addition to core business data, the database should store user registration information, identity credentials, and contract addresses of entity users or objects corresponding to contracts on the blockchain. Meanwhile, to facilitate the bank to monitor the system users and operation status as the administrator, the bank directly reads the historical record data in the system from the blockchain. All the information can be traced through blockchain technology, and authenticity can still be guaranteed. The overall architecture of the system is shown in Fig. 8.9.

System Data Storage Division

Data reading includes reading from both database and blockchain persistent storage. Among them, historical flow and bill type data (such as merchant's settlement coupon application and withdrawal flow and bank query of all coupons that have appeared in the system) will be displayed in the form of database; transaction-related status data and the system's status value data will be read from the blockchain (such as merchant's balance of settlement coupons, information associated with coupons currently issued, and coupons owned by consumers). The classification of both in this project is shown in Table 8.3.

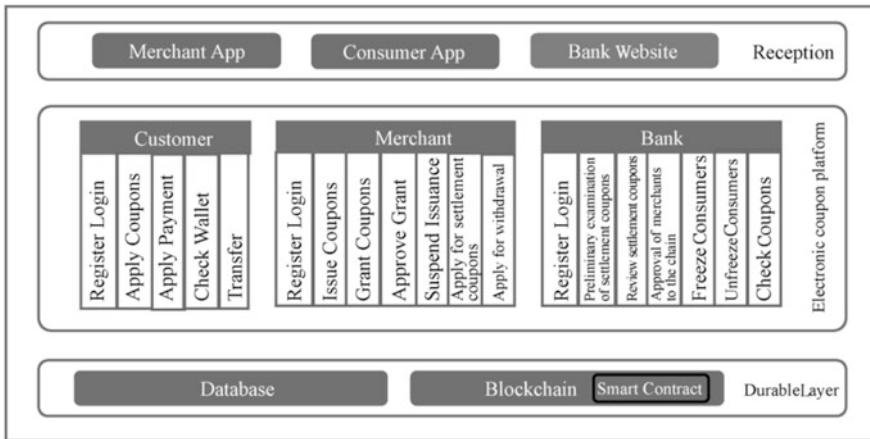


Fig. 8.9 Bank's architecture of the universal electronic coupon system

Table 8.3 Data storage classification of the electronic coupon system

Data from the database	Data from the blockchain
Merchants' settlement coupon application and withdrawal flow	Information on all currently issued coupons managed by merchants
List of merchants and consumers read by bank management systems	Merchants' coupon balance
All coupons read by bank management systems	Coupons in consumers' wallets
Authentication information of system users when logging into the system and details of registration	Details of coupons in the process of coupon issuance, distribution, and use
Public keys of banks, merchants, consumers, and coupons in the system	Details of the coupons being issued in the information system

8.2.4 Smart Contract Design

The smart contract design of this system consists of three parts: the outline design, the contract state design, and the contract method design. The outline design discusses the types of contracts and their main responsibilities; the contract state design analyzes which state values should be stored in the contract to ensure the functional integrity of the contract and provide the necessary data access; and the last explores which actual system operations should be booked by the blockchain and how to map these operations into contract methods.

Outline Design

This case is an electronic coupon system based on blockchain technology, considering the system users. The primary users include bank employees, merchants, and

Table 8.4 General electronic coupon system outline design table

Contract name	Description	Function
Bank	Bank contracts are to be manually deployed to the blockchain at the same time as the project deployment	Approval of merchants' settlement voucher-related operations
Merchant	Store the merchant's creator address, its own contract address, maintain its own array of unissued, issued, and used coupons, and an array of historical coupons	Merchants can use the contract to complete the issuance and termination of coupons; expose the interface to the bank side to complete the settlement coupon application and settlement coupon withdrawal; store their coupon balance for inquiry and maintain coupon arrays for traceability
Coupon	Created by the merchant, storing all relevant status values throughout its lifecycle	Generally passive status change, through the operation of merchants or consumers to passively change their own status
Consumer	Store the consumer's account address, current status, and coupons in their own wallet	Transfer coupons to other consumers' accounts

consumers. Since the users are the initiators of various business operations, these three users should be stored in the blockchain as contract objects.

In addition to system users, coupons are essential for participation in the system. Since the issuance, payment, and transfer of coupons all require the confirmation of rights by blockchain technology, coupons are taken as the object of the contract in this case. The contract passively changes the state to cooperate with the business logic operation of the system.

Apart from being the user, the bank plays the administrator's role, so the bank contract's deployment should be carried out before the system operation, and the bank contract address should be directly available after the project starts. To sum up, there are four contracts in this case system, and the specific design is shown in Table 8.4.

Contract Status Design

In this system, the transaction-related data should be read from the blockchain to play the role of the blockchain's authority. In the contract design, both business logic and data records should be considered comprehensively, and the specific state values are designed as follows.

Bank Contract

The Bank contract should maintain all the merchants it approves to the chain, and use the bank's public key as the “owner” of the contract, as follows:

```
Contract Bank{
    address owner; // Store the bank's public key
    address[] merchants; // Maintain all merchant contract addresses
    approved by yourself for inclusion in the chain
}
```

Merchant Contract

The merchant contract itself shall store its public key (owner field) to supply identity authentication externally and approve the bank contract address of its entry into the chain. The bank shall be entrusted to complete the approval of the relevant operations of the settlement voucher (detailed in the method design below).

The merchant should maintain data on both aspects as the applicant of settlement coupon-related operations and the approver of coupon-related operations. First, merchants should record their clearing coupon balances, issue and grant coupons, and process payment requests using coupons. Therefore, the merchant should maintain arrays of three types of coupons that have been issued but not granted, issued but not used, and used. In addition, to ensure that the entire lifecycle of coupons is traceable, an array of historical coupons should also be maintained.

```
contract Merchant{
    address owner;
    address banker;
    uint settlementBalance;
    address[] unusedCoupons; // unused coupons issued
    address[] usedCoupons; // used coupons
    address[] notGivenCoupons; // issued but not distributed coupons
    address[] historyCoupons; // historical coupons

    // The following two state values are designed for query data
    operations, which will be explained in detail in the method design later
    address[] curGrant;
    mapping(bytes32 => address[]) grantPair;
}
```

Consumer Contract

Consumers should store their public key (as owner), their current status in the system (whether they are frozen or not), and an array of coupons available in their wallets. Besides, since consumers are subject to freezing and unfreezing, they should store

the address of the bank contract that allows them to change the status and deliver the freeze and unfreeze operations to the bank for execution.

```
contract Consumer{
    address owner;
    address banker; // The contract address of the bank corresponding to
    the consumer can be frozen and unfrozen
    uint state;
    address[] coupons;
}
```

Coupon Contract

The data of coupon contracts is largely applied for query and passive status change. Therefore, its state value takes more into account the query of data and the state identification of each stage of the whole life cycle, and the specific fields and the corresponding meanings are shown below:

```
contract Coupon{
    address owner; // owner stores the owner's contract address, which is
    the merchant's contract address at the time of issuance
        // and the consumer's contract address after issuance. If a
    transfer occurs, the value of owner should be changed accordingly.
    address granter; // the issuer merchant's and contract address
    uint value; // Coupon value
    uint limit; // The "full" field in the issuance rules
    bytes32 startDate; // Valid start date as specified in the
    distribution rules
    bytes32 endDate; // Expiration date specified in the issuance rules
    uint obtainValue; // Amount of money spent by the consumer when the
    coupon was obtained
    bytes32 obtainDate; // The time when the consumer received the coupon
    bytes32 consumeDate; // The date the consumer paid with the coupon
    uint consumeValue; // The amount of money spent by the consumer when
    paying with the coupon
    uint state; // Status of the coupon (1 is issued, 2 is issued, and 3 is
    used)
}
```

Contract Method Design

In the following, we mainly introduce the contract method design from three aspects: construction method, functional method, and reading blockchain stored data.

Construct Method

The constructor method is in charge of contract initialization and deployment in the system. The constructor methods for each of the four contract objects are used as follows.

Bank Contract

Ahead of system's running, the transaction is directly deployed by the geth client (see "Project Deployment" section for details), and the contract address is obtained and used as a constant item in the system.

Merchant Contract

The constructor method is called by the bank contract. After the corresponding bank in the system agrees to the merchant's entry into the chain, the bank as the transaction initiator calls the constructor method of the Merchant contract to obtain the contract address. The specific code is as follows:

```
contract Bank{
    ...
    modifier OnlyOwner{
        if (msg.sender == owner) // Method modifier; the method is executed
        only when the method caller is the owner
    }
}

function createMerchant(address merchantAccount) public OnlyOwner
{
    merchants.push(new Merchant(merchantAccount)); // Transfer
    merchant public keys; create and deploy merchant contracts
}
contract Merchant{
    constructor(address merchantAccount) public {
        owner = merchantAccount;
        banker = msg.sender;
    }
}
```

Consumer Contract

The consumer in the system does not enter the chain as a node, but only has the public key and the contract address. It is necessary to specify the bank administrator contract account that can modify its status, constructed as follows.

```
contract Consumer{
    constructor(address bankAccount) public {
```

```

        owner = msg.sender;
        banker = bankAccount;
        state = 1; // Account is not frozen
    }
}

```

Coupon Contract

The owner field of the Coupon contract should store the users holding the Coupon in the system. When the merchant issues the Coupon, the contract is created and constructed as follows.

```

contract Coupon{
    constructor(uint _value, uint _limit, bytes32 _startDate, bytes32
_endDate) public {
        value = _value;
        limit = _limit;
        owner = msg.sender;
        startDate = _startDate;
        endDate = _endDate;
        consumeValue = 0;
        state = 1; // When the contract is created, the coupon must be in the
state of being issued but not granted
    }
}

```

Functional Methods

The functional methods in this system include settlement coupon correlation, coupon correlation, and consumer transfer.

Settlement Coupon Correlation

The main functional methods related to settlement coupons are how the bank approves the merchant's settlement coupon application, defines the settlementApprove() method in the merchant, and sets the method modifier OnlyBanker, i.e., it can only be called by the banker's address, and defines the approve() method in the bank to call the merchant's settlementApprove() method with the following code.

```

contract Bank{
    function approve(address merchantAddress, uint amount) public
OnlyOwner {
        Merchant m = Merchant(merchantAddress);
        m.settlementApprove(amount);
    }
}

```

```
contract Merchant{
    function settlementApprove(uint amount) public OnlyBanker {
        settlementBalance += amount;
    }
}
```

Coupon Correlation

Coupon Issuance

The merchant issues coupons, i.e., establishes the corresponding number of coupon contracts and stores the contract addresses in the `notGivenCoupons` array, with the following code.

```
contract Merchant{
    function issueCoupon(uint value, uint limit, uint quantity, bytes32
startDate, bytes32
endDate) public OnlyOwner {
        // The parameters passed in are: coupon face value, the "full" field of
        // the issuance rule, and the start and end date of the validity period
        if(settlementBalance >= (value*quantity)) {
            for(uint i = 0; i < quantity; i++) {
                notGivenCoupons.push(new Coupon(value, limit, startDate,
endDate));
                settlementBalance -= value; // Note that merchant settlement
voucher balance is changed accordingly at the time of issuance
            }
        }
    }
}
```

Coupon Termination

A merchant can terminate a coupon issue at any time after the coupon has been issued. Upon termination, all coupons in the `notGivenCoupons` array, `unusedCoupons` array, and `usedCoupons` array must be moved to the `historyCoupons` array, and the total amount of coupons not granted needs to be added back to their settlement coupon balance. If any coupons given are beyond the expiration date at the time of termination, they should be withdrawn.

```
contract Merchant{
    function terminateCoupon(bytes32 curDate) public OnlyOwner {
        // for coupons not granted, move them to historyCoupons array and add
        // the amount back to settlement coupon balance
        for(uint k=0;k < notGivenCoupons.length;k++) {
            historyCoupons.push(notGivenCoupons [k] );
            Coupon c = Coupon(notGivenCoupons [k] );
            settlementBalance += c.getValue();
        }
        delete notGivenCoupons;
    }
}
```

```

    // for coupons given but not used, if they are not expired, just move
    them to historyCoupons array
    // the amount is not added back to settlement coupon balance; if
    coupons are expired, add them back to settlement coupon array
    for(uint i=0;i<unusedCoupons.length;i++){
        Coupon c1 = Coupon(unusedCoupons[i]);
        if(c.getEndDate() < curDate){
            settlementBalance += c1.getValue();
        }
        historyCoupons.push(unusedCoupons[i]);
    }
    delete unusedCoupons;
    // for coupons already used, move them to historyCoupons array
    for(uint j=0;j<usedCoupons.length;j++) {
        historyCoupons.push(usedCoupons[j]);
    }
    delete usedCoupons;
}
}

```

Coupon Distribution

The merchant takes the corresponding number of addresses from the notGivenCoupons array from back to front according to the number of coupons passed in, sets the owner of the coupon contract on these addresses as the grantees (consumers), moves the contract address from the notGivenCoupons array to the unusedCoupons array; finally, uses the mapping data structure to facilitate the Java-side to read the data on the blockchain (described in detail in the “Reading Blockchain Stored Data” section below); the specific code is as follows.

```

Contract Merchant{
    function grant(address _consumer, uint quantity, bytes32 date,
bytes32 mark,
        uint obtainValue) public OnlyOwner {
        // The incoming parameters in order are the contract address of the
        grantees, the number of pieces given, the date for distribution, and the
        mark.
        // and the amount of consumption corresponding to this granting
        operation
        if(quantity<=notGivenCoupons.length) {
            Consumer consumer = Consumer(_consumer); // obtain the grantees
            for(uint i=notGivenCoupons.length-1;i>=
                notGivenCoupons.length-quantity;i--) {
                Coupon couponTemp = Coupon(notGivenCoupons[i]);
                // Set the corresponding information of the granted coupons
                couponTemp.setObtainDate(date);
                couponTemp.setState(2);
                couponTemp.setObtainValue(obtainValue);
                couponTemp.setGranter(couponTemp.getOwner());
                couponTemp.setOwner(_consumer);
                consumer.addCoupon(notGivenCoupons[i]);
            }
        }
    }
}

```

```

    // Move the granted coupons in the array
    unusedCoupons.push(notGivenCoupons[i]);
    curGrant.push(notGivenCoupons[i]);

    // Because the variable i is of type uint, jump out the loop by
determining i==0 when the last coupon is granted by the merchant
    if(i == 0){ break;}
}

// The coupon distribution operation is stored in mappingby the
parameter mark, which makes it easy to take the value
grantPair[mark] = curGrant;
delete curGrant;
notGivenCoupons.length = notGivenCoupons.length - quantity;
}
}
}

```

Coupon Payment

The merchant moves the specified coupon from the unusedCoupons array to the usedCoupons array (if the coupon has been terminated, the coupon should be moved directly to the historyCoupons array), and removes the coupon from the Consumer contract. At the same time, the face value of the coupon will be added to the merchant's account balance with the following code.

```

Contract Merchant{
    function confirmCouponPay(uint consumeValue, bytes32 consumeDate,
        address couponAddr, address _consumer) public OnlyOwner{
        Coupon coupon = Coupon(couponAddr);
        if(consumeValue>=coupon.getLimit()) { // If this purchase can use
the coupon

            // Set the status related to this coupon
            coupon.setConsumeValue(consumeValue);
            coupon.setConsumeDate(consumeDate);
            coupon.setState(3);

            // Execute the consumer's couponPay method to move the coupon out
of the consumer's coupon array
            Consumer consumer = Consumer(_consumer);
            consumer.couponPay(couponAddr);

            // Determine if the coupon is a coupon currently being issued by the
merchant, and if it is, move it to the UsedCoupons array.
            // Otherwise, whether issued by a current merchant or not, or
issued by a current merchant but terminated
            // it is moved directly to the historyCoupons array
            uint i = unusedCoupons.length;
            for(i=0;i<unusedCoupons.length;i++) {

```

```
        if(unusedCoupons[i] == couponAddr) {
            break;
        }
    }
    if(i!=unusedCoupons.length){
        for(uint j=i;j<unusedCoupons.length-1;j++){
            unusedCoupons[j] = unusedCoupons[j+1];
        }
        unusedCoupons.length -= 1;
        usedCoupons.push(couponAddr);
        settlementBalance += coupon.getValue();
    }else{
        settlementBalance += coupon.getValue();
        Merchant m = Merchant(coupon.getGranter());
        if(m.getOwner() != owner){
            m.addToUsedCoupons(couponAddr);
        }
    }
}
}
```

Consumer Transfer

Consumer Transfer Coupon forwarding is as simple as moving a specified coupon from the sender's coupon array to the recipient's coupon array and changing the owner field of the coupon as shown below.

```

Contract Consumer{
    function transfer(address newConsumer, address _coupon) public
OnlyOwner{
        Coupon coupon = Coupon(_coupon);
        coupon.setOwner(newConsumer);
        Consumer to = Consumer(newConsumer);
        to.addCoupon(_coupon);
        uint i = 0;
        for(;i<coupons.length;i++) {
            if(coupons[i] == _coupon) {
                break;
            }
        }
        for(uint j=i;j<coupons.length-1;j++){ // the array element that
follows the corresponding position of that coupon.
            // shift each one one place forward
            coupons[j] = coupons[j+1];
        }
        coupons.length -= 1;
    }
}

```

Reading Blockchain Stored Data

There are two primary methods to read the data stored in the blockchain as follows.

- Read the data directly through the getter method in the following format.

```
// where the constant field indicates that the method is simply
reading data and does not need to be called as a transaction.
function getValue() public view returns (dataType) {
    return value;
}
```

- Some tricks have to be applied to get the system's scenario-specific data. When the bank approves the merchant to the chain, the bankside sends a transaction to create a merchant contract and stores its address in the merchants' array maintained by itself. When reading the address of this newly created merchant contract, you cannot simply read the last field of the array but need to traverse the merchants' array by passing in the current merchant public key to find out the merchant whose public key matches and return it, as designed below.

```
Contract Bank{
    function getCorrespondingMerchant(address merchantAccount)
public view returns (address) {
    uint i = merchants.length;
    for(i=merchants.length-1;i>=0;i--) {
        Merchant m = Merchant(merchants[i]);
        if(merchantAccount == m.getOwner()) {
            break;
        }
    }
    return merchants[i];
}
}
```

Similarly, mapping data structure is used to conveniently obtain all coupons given this time to identify the array of coupons with the marked field passed in (see the “Coupon Correlation” section above for the code).

8.2.5 System Implementation and Deployment

This section first describes the overall system deployment, then describes the software and hardware environment required for the system operation, and finally introduces in detail how to build a blockchain environment for the system.

System Deployment Diagram

The system deployment diagram is shown in Fig. 8.10.

In this case, the blockchain bottom layer is established on the Ethereum platform; banks and merchants can access the Ethereum platform as nodes, while consumers access the Ethereum through the bank nodes. The head office can have its node among merchant users, branches can access it through the head office, and franchisees can access it themselves. If the Ethereum nodes adopt multiple servers, these servers should be in the same network segment and provide a unified interface to the outside, ensuring the unhindered communication between servers. In addition, each merchant node can only obtain its transaction information on the blockchain, although all the underlying transaction information is set on the Ethereum platform.

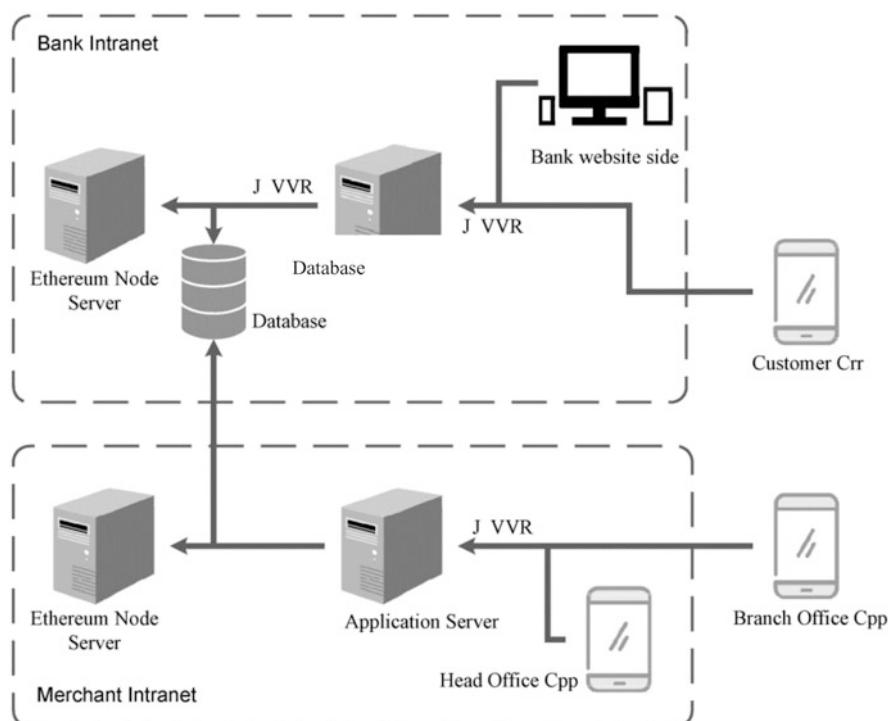


Fig. 8.10 The system deployment diagram

Hardware and Software Environment Deployed

Software Environment

Server side: Linux operating system (Ubuntu 14.04); Tomcat 8 server software; MySQL 5.7.6 database management system or above; JDK 1.7.

Client side: Chrome or Firefox browser installed.

Mobile side: Apple phone or iPad.

Hardware Environment

Ali cloud server, dual-core CPU, 4GB RAM, 500GB available storage space.

Blockchain Environment Construction

An initialized server can follow the steps below to build a blockchain environment.

Install Curl Command

```
apt-get update  
apt-get install git  
apt-get install curl
```

Install the Go Environment (Go Version 1.5.1 Is Installed Here)

```
curl -O https://storage.googleapis.com/golang/go1.5.1.linux-amd64.  
tar.gz  
Unpack it to the /usr/local (may require sudo permissions)  
tar -C /usr/local -xzf go1.5.1.linux-amd64.tar.gz
```

Configure Go's Environment Variables

```
mkdir -p ~/go; echo "export GOPATH=$HOME/go">> ~/.bashrc  
echo "export PATH=$PATH:$HOME/go/bin:/usr/local/go/bin">> ~/.bashrc  
source ~/.bashrc
```

Install Node.js.npm

```
curl -sL https://deb.nodesource.com/setup_4.x | sudo -E bash -  
apt-get install Node.js
```

Verify Node.js.npm

```
Node.js -v  
npm -v
```

Install Ethereum

```
bash <(curl -L https://install-ethereum.org)
```

If an error occurs, use the following command.

```
sudo apt-get install software-properties-common  
sudo add-apt-repository -y ppa:ethereum/ethereum  
sudo add-apt-repository -y ppa:ethereum/ethereum-dev  
sudo apt-get update  
sudo apt-get install Ethereum
```

Install solc

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

Create Accounts (Public Key)

Enter the following command three times in the console to have three accounts.

```
geth account new
```

Write the Founding Block File

Create the test-genesis.json file in the root directory (`~/`). Note that you can assign a large enough balance to the public key you just requested by setting the account address in alloc.

```
{  
  "config": {  
    "chainID": 1024,  
    "homesteadBlock": 0,  
    "eip155Block": 0,  
    "eip158Block": 0  
  },
```

Initialize the Founding Block

```
geth --datadir "~/.ethereum" init ./test-genesis.json
```

Configure the Script to Unlock the Accounts Automatically

Go to the `~/.ethereum` directory, create the password file and enter the password corresponding to each account you just set in that file, one line for each password, and only the password is required.

Write Ethereum Startup Script

Set the startup script file `private_blockchain.sh` and configure the following in the file.

```
geth --rpc --rpccaddir "127.0.0.1" --rpcport "8545" --rpccorsdomain "*" --unlock 0,1,2 --password ~/Library/Ethereum/password --maxpeers 5 --datadir '~/Library/Ethereum' console
```

Each time you start the GEth node, run the following command:

```
bash private_blockchain.sh
```

Deploy the Bank Contract and Get the abi File

Since go-ethereum abandoned the eth_compileSolidity method in its 1.6.0 version, the eth.compileSolidity method cannot now obtain the abi; in this way, we compile the contract online using the Remix mentioned in Sect. 3.2, and get the most critical data bytecode and abiDefinition data.

At this point, all blockchain-related deployments have been completed before JavaWeb project is run. Note that the Ethereum accounts for banks, merchants, and consumers in this project must be created by themselves using the method shown in step “Create Accounts (Public Key).” The established accounts are currently stored in the account.properties file in the project resource directory and any code that employs this account information will be read from the configuration file.

According to the basic knowledge and development techniques of Ethereum learned in previous chapters, readers can practice step by step to better understand relevant concepts and technologies in practice thus laying a solid foundation for their blockchain application projects based on Ethereum.

8.3 Summary

This chapter introduces two Ethereum-based cases, with each including project introduction, system function analysis, overall system design, smart contract design, system implementation, and deployment. By following the basic knowledge and development techniques of Ethereum learned in previous chapters, readers can practice step by step by referring to this chapter to better understand related concepts and technologies in the actual process, so as to lay a solid foundation for their blockchain application projects based on Ethereum.

Chapter 9

Hyperledger Fabric Application Case Studies in Detail



Hyperledger Fabric has a complete permission control and security mechanism, on which you can develop enterprise-level applications with features such as tamper-resistant and traceable. In this chapter, we will start with two real-world examples, dive into the world of fabrics, coupled with the knowledge introduced earlier in this book, to make clear the principles of fabric projects and lay a solid foundation for developing fabric projects on your own.

9.1 Case Study of a Fabric-Based Social Culture Heritage Management Platform

Cultural relics are humanity's valuable historical and cultural heritage with historical, artistic, and scientific values. Its collection, management, and scientific research are essential to understanding history. More and more people are showing interest in the collection of cultural relics. China has also introduced relevant policies regarding cultural relics protection and legal trade. However, many cultural relics, along with the increasing complexity of their management, have made it more challenging to identify and assess their value. Once in circulation, it is crucial to ensure that cultural objects' value and safe circulation follow the regulations. This is precisely the problem that must solve.

This section designs and implements a blockchain-based social heritage management platform for the existing problems and practical needs of the heritage collection. It aims primarily at offering readers with a simple and intuitive blockchain application development example to familiarize them with its process and operation. The explanation of this case will simplify the complexity of the business process and implementation, so as to offer a sort of experience landing practice. We have written smart contracts for the actual business and requirements and used CouchDB as the

underlying database in conjunction with the final business requirements for requesting values.

9.1.1 Project's Background Analysis

There are at the current stage many problems in the management of cultural heritage circulation, such as the difficulty of controlling the flow and the inadequacy of preventive mechanisms for assessing the risks, the cumbersome process of identification, the lack of an effective platform for registering and querying the results of heritage identification, the difficulty for the public to perceive its value, the lack of a platform for certifying its value, and the weak tracing in its flow. A series of pain points above have limited the progress of cultural relics collection.

Blockchain is a distributed database on encryption and consensus algorithms, which can complete peer-to-peer transactions among nodes without mutual trust through data encryption, timestamping, distributed consensus, and economic incentives. The decentralization, non-tamperability, and timestamp of blockchain technology can well solve the problems of circulation traceability, value recognition, and risk assessment in the process of circulation of cultural relics.

In terms of risk assessment and prevention of cultural relics, information on the flow of cultural relics is recorded in blocks. The characteristics of blockchain, such as not being accessible to tamper and timestamp can provide adequate technical support for managing social-cultural relics and improve the market vitality of cultural relics. In terms of cultural relics circulation management, the timestamp feature of blockchain provides the possibility of real-time recording of cultural relics flow. The information on the flow of cultural relics can be recorded in the blockchain, and the supervisory department can track its latest trend in real time to achieve one-stop management, which is conducive to strengthening the management of non-state museum collections and regulating the supervision of legitimate private collections and circulation. In terms of cultural relics value perception, by joining the blockchain in the form of the alliance chain, cultural relics management departments, private institutions, regulatory departments, and others establish the integrity system in private cultural relics collection, connect the database, break information silos, achieve information sharing, solve the problem of information asymmetry, protect legitimate rights and interests, and active the market circulation. After the artifact is identified in the Alliance Chain, it will receive a digital certificate with a unique blockchain code logo, and each node within provides credit endorsement, and the information in the chain is not easy to tamper with. Traditional cultural relics are hard to identify. The information on the chain can provide a reference for demanders, solve the cognition problem of the cultural relics' value, and serve as an authoritative reference for traders in the market, reducing the transactions risk. This will help standardize the identification of cultural relics, guide rational collection, and standardize the legal circulation of cultural relics.

9.1.2 System Functional Analysis

The blockchain-based social heritage management platform functionally consists of three main modules: government management module, external interface module, and general user module.

1. The primary function of the government management module is to audit the organizations and users who apply to join the chain, examine whether the heritage trading organization has authoritative and influential appraisers, and generate a unique ID certificate for organizations that pass the audit and save it to the blockchain. The value certificate of the cultural relics issued by the audited appraisal institution is informative only. The government management module has an early warning mechanism for transactions between users, and once illegal transactions are detected, it will alert the regulator for review. Illegal transactions will be rejected, only legal transactions will be recorded on the blockchain, and the auction and legal collection of cultural relics will be regulated.
2. The main function of the external interface module is to record the information of cultural relics, necessarily being uploaded to the chain and stored on the blockchain for query. Cultural heritage identification institutions supply identification services, which calls for collecting information and conducting chain audit, by collecting and storing physical information such as photos, high-definition scanning, and 3D data model of artworks into the database. In this way, it will generate a hash value of these informations, stored on the chain as the unique identification of artworks.
3. The general user module provides information management of cultural relics on the user platform, including personal information, information on cultural relics in the user's collection, while allowing for inquiries on cultural relics on the chain and complaints about illegal transactions.

The blockchain-based heritage management system is broken down into regulators, merchants, and authenticators in terms of user roles. The regulator vets the information of users who want to join the blockchain network; the trader refers to ordinary users who can legally trade the vetted artifacts through the system; the authenticator screens the authenticity and origin of the artifacts and adds the vetted artifacts to the blockchain network.

Meanwhile, the system module includes auction institutions, which in this section include: cultural museums, museums, galleries, auction houses, etc. The cultural objects held by the auction institutions can be traded on the chain after an appraisal agency has authenticated them.

9.1.3 General System Design

In the following, we explore how blockchain technology can be practically applied to the management of social-cultural relics. We realize the application in various scenarios by using the existing technical reserve and the mature and controllable blockchain underlying platform, coupled with the current situation and characteristics of the specific social-cultural relics industry. For example, the process of mutual exchange mechanism of cultural relics in collections is simplified, and the process supervision is strengthened so that cultural relics become “alive”; the social-cultural relics registration and filing system and the integrity system in the field of folk cultural relics collection are established on the blockchain underlying technology, guiding rational collection and standardizing the legal circulation of cultural relics; the social-cultural relics circulation management system on the blockchain is set up to enhance management services for the social-cultural relics circulation.

Overall Thought

1. Formation of authentic and credible information flow of cultural relics: making full use of the blockchain's characteristics such as distributed storage and not easily tampered with to solve the information asymmetry, establish digital certificates of cultural relics, record information of cultural relics circulation transactions, and realize effective risk assessment and prevention. The digital certificate and circulation records of cultural relics on the blockchain are shared to solve the problem of information asymmetry and to achieve cross-validation of cultural relics information and mutual credit endorsement among all parties involved.
2. Improving the management capacity of cultural relics circulation: introducing supervisory nodes to enhance social heritage transactions and circulation control capacity, constructing a reliable social heritage management network through the blockchain, and ensuring the traceability of the whole process of cultural relics transactions and circulation in the blockchain.
3. Providing an infrastructure for collaboration among participating social heritage institutions: Blockchain, as a distributed ledger, acts as a platform for equal collaboration and mutual supervision among all participants, so that multiple parties can collaborate in the identification and protection of cultural relics, which can significantly reduce the human and material resources consumed in the identification of cultural relics, standardize the management of social heritage, and enhance the data credibility.

Application Architecture

Figure 9.1 illustrates the system structure, dividing into two main blocks: Web application module and blockchain storage module. The blockchain storage module

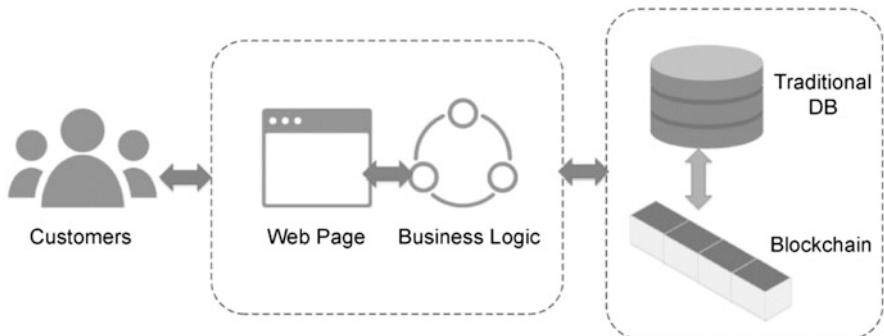


Fig. 9.1 System architecture diagram

basically provides reliable data storage, ensures the credibility of records, avoids data monopoly, and achieves the purpose of decentralization.

Following details the functional architecture of the system, which mainly includes five modules: essential services, middle office services, business services, interfacing methods, and participating organizations.

(a) **Basic services**

The underlying services have blockchain nodes, message queue services, third-party services, caching services, and database services.

(b) **Middle platform services**

The middle office services include a block data middle office, a smart contract middle office, and a service aggregation middle office.

(c) **Business services**

The business services include four modules: general services, social heritage management services, data visualization, and back-end management.

(d) **Interacting method**

The interacting method employs mobile clients, mobile browsers, and APIs.

(e) **Participating institutions**

Participating institutions include heritage bureaus, museums, and accreditation bodies.

9.1.4 General Design of Smart Contracts

The difference between smart contracts and traditional applications is that it is hard to modify once a smart contract is published in the blockchain network. Even if there is a problem or a need to change the business logic, the new smart contract cannot directly modify the original version thus the following objectives should be achieved when designing a smart contract: perfect business functions, compact logic code, excellent module design, clear contract structure, suitable security mechanism, and a complete upgrade scheme.

The smart contract initializes each parameter through the Init method and invokes the Invoke method whenever the node needs to query or modify the parameter value. The parameters in the Invoke method are the method names of the smart contract application to be invoked. The GetFunctionAndParameters method of the ChaincodeStubInterface interface is called to extract the method name and the parameters of the smart contract initiated transaction. The method name is validated, and the corresponding smart contract application method is called, and, finally, shim. Success or shim. Error is returned depending on the result of the execution.

9.1.5 Core Functional Contract Design

Objects in Smart Contracts

The main reason for employing JSON tags here is that the state values are reserved as character arrays when storing state. It would help if you serialize the business entities before storing them, and the best way is to use JSON, yet you can adopt messagepack and protobuf. In this case, we have JSON for convenience to better support rich queries by CouchDB.

```
type RelicOrder struct {
    // Transaction information
    OrderID      string `json:"orderID"`           // Transaction ID
    OrderValue   string `json:"orderValue"`         // Transaction value
    OrderDate    string `json:"orderDate"`          // Transaction time
    OrderStatus  string `json:"orderStatus"`        // Transaction
    status
    ProvideID   string `json:"provideID"`         // Transaction
    provider ID
    BuyerID      string `json:"buyerID"`           // Buyer ID
    SellerID     string `json:"sellerID"`          // Seller ID
    RelicID      string `json:"relicID"`            // Relics details
    GovNum       string `json:"govNum"`             // Relics ID
    GovNum       string `json:"govNum"`             // National certification
    number of the relics

    RelicName    string `json:"relicName"`          // Name of relics
    RelicDescribe string `json:"relicDescribe"`     // Description of
    relics
    RelicDataURL string `json:"relicDataURL"`       // Original data URL
    of relics
    ImageURL    string `json:"imageURL"`           // Image URL of relics
    InputDate    string `json:"inputDate"`           // Date of input
    JudgeName   string `json:"judgeName"`          // Name of judge
    JudgeNum    string `json:"judgeNum"`            // ID of judge
    JudgeOrgID  string `json:"judgeOrgID"`         // Organization ID of
    judge
    Evaluation   string `json:"evaluation"`         // Evaluation of
```

```

relics
EvaluationName string `json:"evaluationName"` // Name of evaluator
EvaluationNum  string `json:"evaluationNum"`   // ID of evaluator
newValue      string `json:"newValue"`        // Latest transaction
price
newValueDate  string `json:"newValueDate"`    // Time of latest
transaction price
OwnerID       string `json:"ownerID"`         // Owner ID of relics
RelicStatus   string `json:"relicStatus"`      // Relics status

```

Initialization Methods for Smart Contracts

The Init method is called when the smart contract is instantiated to initialize the data. In addition, this function is called to reset or migrate data when the smart contract is upgraded.

```

func (t *RelicChaincode) Init(stub shim.ChaincodeStubInterface)
peer.Response {
    return shim.Success([]byte("Success Init"))
}

```

New Transactions

New transactions are created through the AddNewOrder method, which stores transaction and relics information on the blockchain.

```

func AddNewOrder(stub shim.ChaincodeStubInterface, orderData []string) (string, error) {
    // Determining the correctness of parameters
    if len(orderData) != 24 {
        return "", fmt.Errorf("the number of args is %d, not 24", len(orderData))
    }
    // Store data
    relicOrder := new(RelicOrder)
    relicOrder.OrderID = orderData[0] relicOrder.OrderValue = orderData[1]
    relicOrder.OrderDate = orderData[2] relicOrder.OrderStatus =
    orderData[3] relicOrder.ProvideID = orderData[4] relicOrder.
    BuyerID = orderData[5] relicOrder.SellerID = orderData[6] relicOrder.
    RelicID = orderData[7] relicOrder.GovNum = orderData[8] relicOrder.
    RelicName = orderData[9] relicOrder.RelicDescribe = orderData[10]
    relicOrder.RelicDataURL = orderData[11] relicOrder.ImageURL =
    orderData[12] relicOrder.InputDate = orderData[13] relicOrder.
    JudgeName = orderData[14] relicOrder.JudgeNum = orderData[15]
    relicOrder.JudgeOrgID = orderData[16] relicOrder.Evaluation =
    orderData[17] relicOrder.EvaluationName = orderData[18]
    relicOrder.EvaluationNum = orderData[19]
    relicOrder.NewValue = orderData[20] relicOrder.NewValueDate =

```

```

orderData[21]
    relicOrder.OwnerID = orderData[22] relicOrder.RelicStatus =
orderData[23]

    data1, err := stub.GetState(relicOrder.OrderID) if data1 != nil &&
err == nil {
    return "", errors.New(fmt.Sprintf("relic order : %s has been existed
", relicOrder.OrderID))
}
// Serialized data
data, err := json.Marshal(relicOrder)
if err != nil {
    return "", errors.New("relic order marshal is failed for " + err.
Error())
}
// Store data
err = stub.PutState(relicOrder.OrderID, data)
// Check for successful deposit
if err != nil{
    return "", errors.New(fmt.Sprintf("put State for relic order %s
failed", relicOrder.OrderID))
}
return "add relic order success", nil
}

```

Read Transaction Log Information

The following chaincode function implements reading the record information from the blockchain data block so that basic query operations can be performed according to the transaction's ID.

```

func GetOrder(stub shim.ChaincodeStubInterface, orderID []string)
(string, error) {
    // Determining the correctness of parameters
    if len(orderID) != 1 {
        return "", fmt.Errorf("the number of args is %d, not 1", len
(orderID))
    }
    // Verify the existence of the data
    relicData, err := stub.GetState(orderID[0]) if err != nil{
        return "", fmt.Errorf("getting relic order %s error for %s",
orderID
[0], err.Error())
    }
    if relicData == nil {
        return "", fmt.Errorf("relic order %s is not exist", orderID[0])
    }
    return string(relicData[:]), nil
}

```

Search for Relics Transaction Records

This function has the caller with a record of the relics transaction and requires the relics ID for calling the function by means of the query statement.

```
{\"selector\":{\"relicID\":\"%s\"}.
func GetOrderbyRelicID(stub shim.ChaincodeStubInterface, relicID []string) (string, error)
{ if len(relicID) != 1{
    return "", fmt.Errorf("the number of args is %d, not 1", len(relicID))
}
queryString := fmt.Sprintf("{\"selector\":{\"relicID\":\"%s\"}}", relicID[0]) queryResults, err := getQueryResultForQueryString(stub, queryString)
if err != nil {
    return "", err
}
return string(queryResults), nil
}
```

9.1.6 Tool Contract Design

This function is a concrete implementation of a custom query.

```
func getQueryResultForQueryString(stub shim.ChaincodeStubInterface, queryString string)
([]byte, error) {
fmt.Printf("- getQueryResultForQueryString queryString:\n%s\n",
queryString) resultsIterator, err := stub.GetQueryResult(
queryString)
if err != nil {
    return nil, err
}
defer resultsIterator.Close()
buffer, err := constructQueryResponseFromIterator(
resultsIterator) if err != nil {
    return nil, err
}
fmt.Printf("- getQueryResultForQueryString queryResult:\n%s\n",
buffer.String())

return buffer.Bytes(), nil
}

func constructQueryResponseFromIterator(resultsIterator shim.StateQueryIteratorInterface) (*bytes.Buffer, error) {
// buffer is a JSON array containing the query result set
```

```

var buffer bytes.
buffer.WriteString("[")

bArrayMemberAlreadyWritten := false for resultsIterator.HasNext() {
    queryResponse, err := resultsIterator.Next() if err != nil {
        return nil, err
    }

    if bArrayMemberAlreadyWritten == true
        { buffer.WriteString(",")
    }
    WriteString("{\"Key\":\"") buffer.WriteString("\"") buffer.
WriteString(queryResponse.Key) buffer.WriteString("\"") buffer.
    WriteString(", \"Record\":") buffer.WriteString(string
(queryResponse.Value)) buffer.WriteString("}")
bArrayMemberAlreadyWritten = true
}
buffer.WriteString("]")

return &buffer, nil
}

```

9.1.7 Deployment Implementation

This project uses the fabric developer mode for testing the chaincode during the deployment and installation and aims to display a simple deployment process to the reader. The reader can configure the channels and nodes of the platform according to the project requirements.

The lifecycle of the contract code throughout the process is divided into installation, instantiation, upgrade, packaging, and signing processes.

Pre-Installation Preparation

Before starting developer mode, you are asked to download the fabric-samples file and the Docker image; see Chap. 5 for details. Place the folder containing the smart contract (relic) under.

fabric-samples/chaincode.

Start Developer Mode

In “fabric-sample” mode, the user builds and launches the chaincode. It is essential to remind developers to use three terminals to run the chaincode in “fabric-sample” mode. Terminal 1 manages the development mode network environment; terminal 2 manages the chaincode container; and terminal 3 deals with the cli container. All of the three must be in the chaincode- docker-devmode folder.

```
$ cd ~ /go/src/github.com/hyperledger/fabric-samples/chaincode-docker-devmode  
// Terminal 1 start the development network environment  
$ docker-compose -f docker-compose-simple.yaml up
```

Chaincode Operations

1. Terminal 2 accessing the chaincode container

```
$ docker exec -it chaincode bash # Enter the chaincode container
```

2. Compiling the project

```
$ cd relic  
$ go build
```

3. Run chaincode

```
$ CORE_PEER_ADDRESS=peer:7052 CORE_CHAINCODE_ID_NAME=relic:0 ./relic
```

4. Terminal 3 installation and instantiation

```
$ docker exec -it cli bash # Enter the cli container  
#Installation  
$ peer chaincode install -p chaincodedev/chaincode/relic -n relic -v 0  
$ peer chaincode instantiate -n relic -v 0 -C myc -c '{"Args": ["init"]}'  
#insert single transaction  
$ peer chaincode invoke -n relic -c '{"Args": ["addneworder", "10001", "456.5", "2019-3-6", "1", "20001", "888123201903068888", "888123201903067777", "30001", "zjureliccode", "rules", "the rules of zju", "www.zju.edu.cn", "www.relicimages.com//zjurules", "2019-3-6", "tomas", "888123201903066666", "40001", "456.5", "tomas", "888123201903066666", "456.5", "2019-3-6", "888123201903067777", "1"]}' -C myc  
$ peer chaincode invoke -n relic -c '{"Args": ["addneworder", "10002", "500.5", "2019-3-7", "1", "20001", "888123201903067777", "888123201903068888", "30001", "zjureliccode", "rules", "the rules of zju", "www.zju.edu.cn", "www.relicimages.com//zjurules", "2019-3-6", "tomas", "888123201903066666", "40001", "456.5", "tomas", "888123201903066666", "456.5", "2019-3-6"}'
```

```

6", "888123201903068888", "1"]}' -C myc
$ peer chaincode invoke -n relic -c '{"Args": ["addneworder",
"10003", "456.5", "2019-3-7", "1",
"20001", "888123201903068888", "888123201903067777", "30001",
"zjureliccode", "rules", "the
rules of
zju", "www.zju.edu.cn", "www.relicimages.com//zjurules", "2019-
3- 6", "tomas",
"888123201903066666", "40001", "456.5", "tomas",
888123201903066666", "456.5", "2019-3-
6", "888123201903067777", "1"]}' -C myc
#Inquire about a single transaction (查询单个交易)
$ peer chaincode query -n relic -c '{"Args": ["getorder", "10001"]}' -
-C myc
$ peer chaincode query -n relic -c '{"Args": ["getorder", "10002"]}' -
-C myc
$ peer chaincode query -n relic -c '{"Args": ["getorder", "10003"]}' -
-C myc

```

5. Packaging

Packaging and signing operations can be achieved by encapsulating the data associated with the chaincode.

```
$ peer chaincode package -n relic -p chaincodedev/chaincode/relic
-v 0 -s -S -i "AND('OrgA.admin')" relic.out
```

Some of the commands are explained below.

- s: Create a package of CC deployment specifications supported by the role, instead of the original CC deployment specifications.
- S: If creating a CC deployment specification scheme role support, also sign it using the local MSP.
- i: Specify the instantiation policy.

6. Signature

```
$ peer chaincode signpackage relic.out signedRelic.out
```

9.2 Case Study of a High-End Fabric-Based Food Safety System

Because of the fast social-economic progress, coupled with the rapid improvement of living standards, people pay more and more attention to food safety, especially high-end food products, whose safety level is of particular concern. Most high-end food products have traceability systems, but the systems are established on the

original centralized system architecture, and the background data can be easily tampered with. The information presented to the public is not credible. Blockchain is a new distributed data storage technology with multi-party collaborative book-keeping and information not easily tampered with, being particularly suitable for supporting the development of high-end food safety traceability systems.

In this section, we design and implement a blockchain-based high-end food safety system to address the existing problems and practical demands in the flow and transportation of high-end food products and draw a picture of a simple and intuitive blockchain application development example. As this case aims to provide readers with a simple case that is easy to practice, we will simplify the business process and implementation.

9.2.1 Background Analysis

For many people advocate a healthy diet, low-heat, nutritious food is gradually gaining popularity. High-end food often has higher standards and requirements than ordinary ones in processing, transportation, and trading, from the ingredients introduction and processing process to the final food distribution and trading, all of which require strict supervision.

The Internet's progress has made the food selection, processing, and sales process more transparent and consumers with more rights to know. In parallel, some food safety issues have come to light repeatedly, since the high-end food industry has many processing and food distribution hygiene issues. However, it is quite difficult to track the food process that has gone wrong, and make claims during the problem handling process.

Lack of adequate supervision on any of these issues will pose risks to consumers' health. We want to credibly record every process of food production and flow and present it to consumers instantly; then, we want to ensure that the recorded information is genuine, effective, and traceable, while not being tampered with. Blockchain-based applications will give a practical and feasible solution under such a business scenario.

9.2.2 Solution Proposal

1. Supply and production of high-end food products

The food delivery industry is combined with blockchain technology. In the process of identity audit, all stores in the food delivery platform must be audited by the relevant institutions on the chain, and are allowed to enter the chain only if they meet the requirements of national food safety and business audit. In the process, the blockchain platform ensures that the stores, store staff, and raw material suppliers all comply with the relevant national regulations (certificate

certification); in contrast, the certification information of the participating organizations and individuals in the food delivery production process is stored on the chain.

2. High-end food supply chain records

The information will be instantiated in the blockchain network after the consumer places an order through the food delivery platform. In receiving and producing the food delivery order, the merchant store needs to enter the raw material information required for the food in the order through the food delivery platform, and the data will be finally deposited in the blockchain network. From the order initiation to the receipt, the consumer can view the client's order information in the blockchain.

3. High-end food safety traceability

The source of ingredients, processing and production works, and the delivery process of food delivery is recorded and saved on the blockchain simultaneously. It is convenient for consumers to quickly and accurately trace back to the responsible party when they encounter food problems and safeguard their legitimate rights and interests. It also ensures that the information on the chain is genuine and valid, not easily tampered with, and accurately traced and defined. It allows consumers to have intuitive and accurate access to the production and operation of the food. It also uses blockchain encryption algorithms and authorized access mechanisms to ensure data security and privacy.

9.2.3 System Functional Analysis

Overall Functional Diagram

The system overall functionality is illustrated in Fig. 9.2.

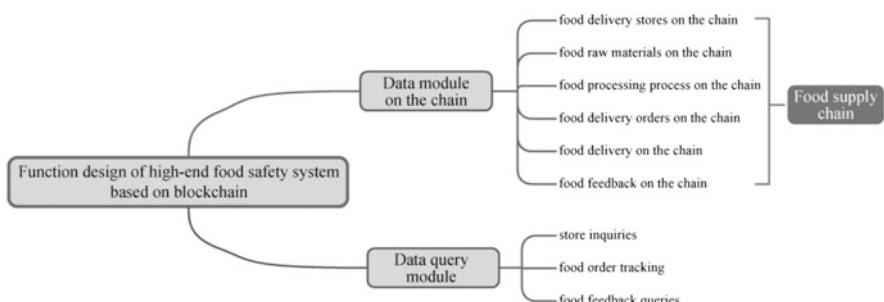


Fig. 9.2 System functions

System Function Description

1. Food delivery stores on the chain. Stores can upload their identity documents and pass the audit before being on the chain. The food supervision platform for certification can guarantee it.
2. Food raw materials on the chain. Stores need to upload the ingredients for the order involving products, including information on procurement, quality assurance, etc., to ensure the information transparency on ingredients.
3. Food processing process on the chain. Food processing information refers to those about making food, including the information of the person in charge of the production.
4. Food delivery orders on the chain. This function records food transactions on the one hand, and trace back to the store owner and store manager related to the responsibility when something goes wrong with the food on the other hand.
5. Food delivery on the chain. Both the delivery person and path need to be on the blockchain.
6. Food feedback on the chain. Consumers can report problems caused by the food delivered through feedback to hold those involved accountable.
7. Store inquiries. Consumers can check information about the store on the one hand, view feedback received by the store on the other, and information about the store's legal compliance.
8. Food order tracking is a way for merchants and consumers to check orders related to them and facilitate the tracing of food safety issues.
9. Food feedback queries. This function makes it convenient for customers to share the business status of the store.

9.2.4 General System Design

Overall System Architecture

As displayed in Fig. 9.3, the system is developed on blockchain, and the whole service differs from the centralized system. The system will provide essential services such as blockchain, message queue, third-party, and cache and database. The essential services give the basic support for upper layer applications, generally satisfying the most basic services, such as distributed data storage, data transmission, and queries.

On the basic services, the SDK service of the blockchain platform is offered by an ordinary business, i.e., the underlying service. The underlying service allows users to build their Turing-complete smart contracts and deploy them to the blockchain through a series of convenient interfaces while encapsulating generic blockchain services and providing a series of generic call query interfaces for generic business calls.

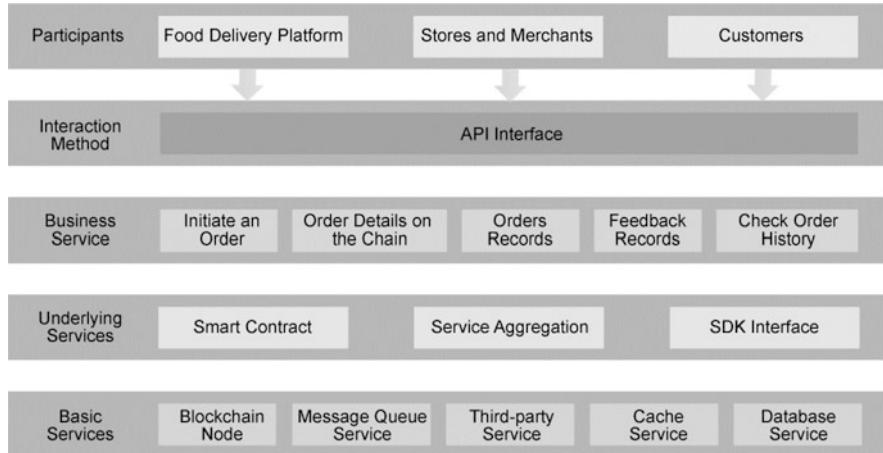


Fig. 9.3 Overall system architecture diagram

While serving, the API of each function is implemented for food distribution organizations to call, store, and query relevant information to achieve food safety traceability.

System Flow Chart

Figure 9.4 displays the system flowchart, depicting the whole process from merchant uplink to food supply chain uplink and then food evaluation uplink, recording the information of each order in detail and ensuring high transparency and traceability of high-end food products.

9.2.5 API Design

The API design comprises three parts of data encryption module, data upload module and data query module, as illustrated in Fig. 9.5.

The data encryption module has the function of symmetric encryption and asymmetric encryption of data. The data uploading module provides the function of uploading information related to food orders. The data query module supplies the function of querying order data.

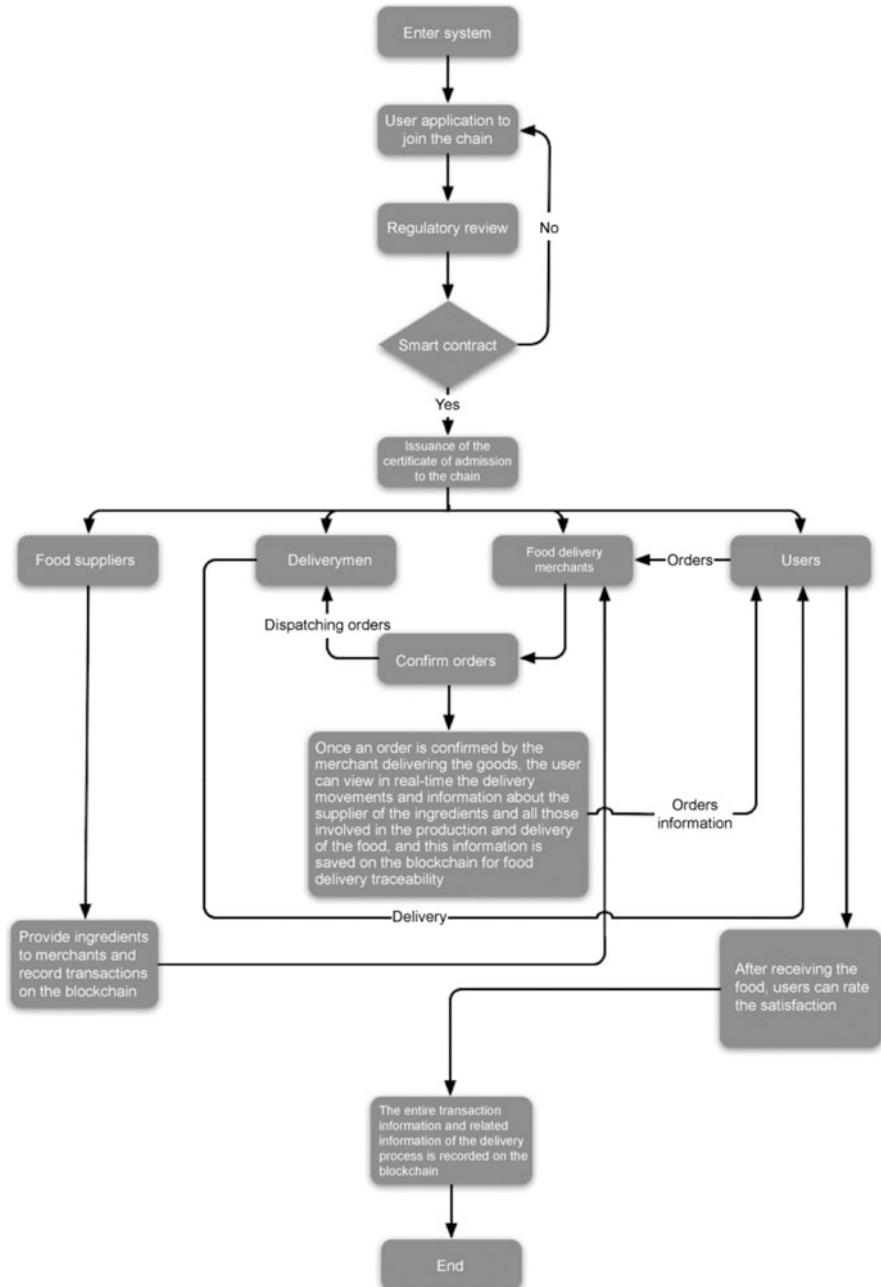


Fig. 9.4 Overall system flowchart

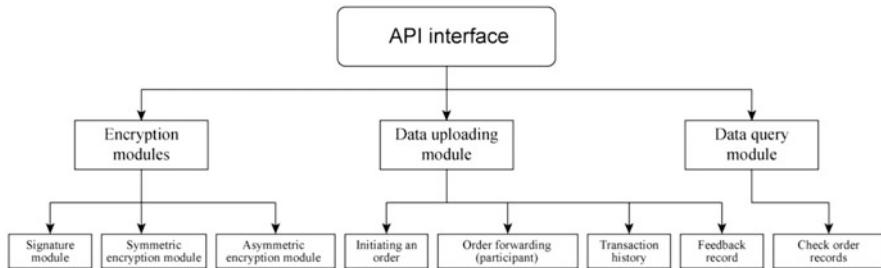


Fig. 9.5 API module diagram

9.2.6 Smart Contract Design

I believe that readers have already understood the design of smart contracts through the examples in the previous sections; therefore, here is a brief review: since smart contracts cannot be tampered with once they are published in the blockchain network, we need to design smart contracts with complete business functions, compact logic code, good module design, clear contract structure, good security checks, and complete upgrade solutions. It is necessary to design the smart contract with complete business functions, compact logic code, good module design, clear contract structure, reasonable security checks, and a complete upgrade scheme.

Core Functional Contract Design

Contract Objects in Smart Contracts

Similarly, we apply JSON tags here, mainly because when storing state, the state values are reserved as characters arrays, then there is a need to serialize the business entities before storing, and the best way is to resort to JSON, and in addition, JSON, messagepack and protobuf can be used as well. We are using JSON for convenience and to support rich queries when using CouchDB.

```

// User
type User struct {
    Name      string `json:"name"`
    Id       string `json:"id"`
    Ingredients []string `json:"ingredients"`
    Foods     []string `json:"foods"`
}
// Food
type Food struct {
    Name      string `json:"name"`
    Id       string `json:"id"`
    Metadata  string `json:"metadata"`
    Ingredients []string `json:"ingredients"`
}
  
```

```

}
// Ingredients
type Ingredient struct {
    Name      string `json:"name"`
    Id        string `json:"id"`
    Metadata  string `json:"metadata"`
}
// Foodstuffs Circulation
type IngredientHistory struct {
    IngredientId   string `json:"ingredient_id"`
    OriginOwnerId  string `json:"origin_owner_id"`
    CurrentOwnerId string `json:"current_owner_id"`
}
// Food distribution circulation
type FoodHistory struct {
    FoodId        string `json:"food_id"`
    OriginOwnerId string `json:"origin_owner_id"`
    CurrentOwnerId string `json:"current_owner_id"`
}

```

Initialization Methods for Smart Contracts

The Init method is called when the smart contract is instantiated, and its role is to initialize the data. Also, this function is called when the smart contract is upgraded to reset or migrate the data.

```

func (c *IngredientsExchangeCC) Init(stub shim.ChaincodeStubInterface) pb.Response { return shim.Success(nil)
}

```

Smart Contract Invoke Method

In the Invoke method, there is user registration, user deletion, ingredient registration, food list generation, ingredient registration, food circulation, ingredient circulation history change query, and food delivery history change query.

```

func (c *IngredientsExchangeCC) Invoke(stub shim.ChaincodeStubInterface) pb.Response {
    funcName, args := stub.GetFunctionAndParameters()

    switch funcName{
    case "userRegister":
        return c.userRegister(stub, args)
    case "userDestroy":
        return c.userDestroy(stub, args)
    case "ingredientEnroll":
        return c.ingredientEnroll(stub, args)
    case "foodEnroll":

```

```

        return c.foodEnroll(stub, args)
    case "ingredientExchange":
        return c.ingredientExchange(stub, args)
    case "foodExchange":
        return c.foodExchange(stub, args)
    case "ingredientExchangeFood":
        return c.ingredientExchangeFood(stub, args)
    case "queryUser":
        return c.queryUser(stub, args)
    case "queryIngredient":
        return c.queryIngredient(stub, args)
    case "queryFood":
        return c.queryFood(stub, args)
    case "queryIngredientHistory":
        return c.queryIngredientHistory(stub, args)
    case "queryFoodHistory":
        return c.queryFoodHistory(stub, args) default:
        return shim.Error(fmt.Sprintf("unsupported function: %s",
funcName))
    }
}
}

```

User Registration

When a user registers, we first check whether the number of parameters required to register the user is correct, then verify the correctness of the parameters, and whether the user has been registered. After satisfying all these conditions, the user information is written to the blockchain, and the result of successful execution is returned.

```

// User registration
func (c *IngredientsExchangeCC) userRegister(stub shim.ChaincodeStubInterface, args []string) pb.Response {
    // Check the number of parameters
    if len(args) != 2 {
        return shim.Error("not enough args")
    }
    // Verify the correctness of the parameters
    name := args[0]
    id := args[1]
    if name == "" || id == "" {
        return shim.Error("invalid args")
    }
    // Verify the existence of the data
    if userBytes, err := stub.GetState(constructUserKey(id)); err == nil
&& len(userBytes) != 0 {
        return shim.Error("user already existing")
    }
}

```

```

        }
        // Write Status
        user := &User{
            Name:name, Id:id,
            Ingredients: make([]string, 0),
            Foods:make([]string, 0),
        }
        // Serialized objects
        userBytes, err := json.Marshal(user)
        if err != nil {
            return shim.Error(fmt.Sprintf("marshal user error %s", err))
        }
        if err := stub.PutState(constructUserKey(id), userBytes); err != nil {
            return shim.Error(fmt.Sprintf("put user error %s", err))
        }
        // Return successfully
        return shim.Success(nil)
    }
}

```

Delete User

The following chaincode function implements the operation of deleting a user on a block. The first step is to verify the number and correctness of the parameters; the second is to query if the user to be deleted exists, and, if so, delete the user, then change the state, and finally delete the ingredients under the user name as well.

```

// Delete user
func (c *IngredientsExchangeCC) userDestroy(stub shim.ChaincodeStubInterface, args []string)

pb.Response {
    // Check the number of parameters
    if len(args) != 1 {
        return shim.Error("not enough args")
    }
    // Verify the correctness of the parameters
    id := args[0]
    if id == "" {
        return shim.Error("invalid args")
    }
    // Verify the existence of the data
    userBytes, err := stub.GetState(constructUserKey(id))
    if err != nil || len(userBytes) == 0 {
        return shim.Error("user not found")
    }
    // Write Status
    if err := stub.DelState(constructUserKey(id)); err != nil {
        return shim.Error(fmt.Sprintf("delete user error: %s", err))
    }
}

```

```

    }
    // Delete ingredients under username
    user := new(User)
    if err := json.Unmarshal(userBytes, user); err != nil {
        return shim.Error(fmt.Sprintf("unmarshal user error: %s", err))
    }
    for _, ingredientid := range user.Ingredients {
        if err := stub.DelState(constructIngredientKey(ingredientid)); err != nil {
            return shim.Error(fmt.Sprintf("delete ingredient error: %s", err))
        }
    }
    return shim.Success(nil)
}

```

Registration of Foodstuffs

The first step is to check the number of parameters, verify the parameters' correctness, and check whether the ingredients have been registered. The second step is to document the components under a user name and record ingredient changes to provide data for future ingredient history queries.

```

// Ingredients registration
func (c *IngredientsExchangeCC) ingredientEnroll(stub shim.ChaincodeStubInterface, args []string) pb.Response {
    // Check the number of parameters
    if len(args) != 4 {
        return shim.Error("not enough args")
    }
    // verify the correctness of the parameters
    ingredientName := args[0]
    ingredientId := args[1]
    metadata := args[2]
    ownerId := args[3]
    if ingredientName == "" || ingredientId == "" || ownerId == "" {
        return shim.Error("invalid args")
    }
    // Verify the existence of the data
    userBytes, err := stub.GetState(constructUserKey(ownerId))
    if err != nil || len(userBytes) == 0 {
        return shim.Error("user not found")
    }
    if ingredientBytes, err := stub.GetState(constructIngredientKey(ingredientId)); err == nil && len(ingredientBytes) != 0 {
        return shim.Error("ingredient already existing")
    }

    // Write Status
    ingredient := &Ingredient{
        Name: ingredientName,
        Id:   ingredientId,
        Metadata: metadata,
    }
    ingredientBytes, err := json.Marshal(ingredient)
    if err != nil {

```

```

        return shim.Error(fmt.Sprintf("marshal ingredient error: %s",
err))
    }
    if err := stub.PutState(constructIngredientKey(ingredientId),
ingredientBytes); err != nil {
        return shim.Error(fmt.Sprintf("save ingredient error: %s", err))
    }
    user := new(User)
    // Deserialize the user
    if err := json.Unmarshal(userBytes, user); err != nil {
        return shim.Error(fmt.Sprintf("unmarshal user error: %s", err))
    }
    user.Ingredients = append(user.Ingredients, ingredientId)
    // Serialized users
    userBytes, err = json.Marshal(user)
    if err != nil {
        return shim.Error(fmt.Sprintf("marshal user error: %s", err))
    }
    if err := stub.PutState(constructUserKey(user.Id), userBytes); err
!= nil {
        return shim.Error(fmt.Sprintf("update user error: %s", err))
    }
    / History of change of ingredients
    history := &IngredientHistory{
        IngredientId: ingredientId, OriginOwnerId: originOwner,
        CurrentOwnerId: ownerId,
    }
    historyBytes, err := json.Marshal(history) if err != nil {
        return shim.Error(fmt.Sprintf("marshal ingredient history error: %
s", err))
    }
    historyKey, err := stub.CreateCompositeKey("history", []string{
        ingredientId,
        originOwner, ownerId,
    })
    if err != nil {
        return shim.Error(fmt.Sprintf("create key error: %s", err))
    }
    if err := stub.PutState(historyKey, historyBytes); err != nil {
        return shim.Error(fmt.Sprintf("save ingredient history error: %s",
err))
    }
    return shim.Success(nil)
}

```

Food Establishment

Register the food products under the appropriate user name after being set up and establish a food change record.

```

// Food registration
func (c *IngredientsExchangeCC) foodEnroll(stub shim.ChaincodeStubInterface, args []string) pb.Response {
    // Check the number of parameters
    if len(args) != 4 {
        return shim.Error("not enough args")
    }
    // Verify the correctness of the parameters
    foodName := args[0]
    foodId := args[1] metadata := args[2] ownerId := args[3]
    if foodName == "" || foodId == "" || ownerId == "" {

        return shim.Error("invalid args")
    }
    // Verify the existence of the data
    userBytes, err := stub.GetState(constructUserKey(ownerId))
    if err != nil || len(userBytes) == 0
        { return shim.Error("user not found")
    }
    if foodBytes, err := stub.GetState(constructFOODKey(foodId)); err ==
    nil && len(foodBytes) != 0
        { return shim.Error("food already existing")
    }
    // Write Status
    food := &Food{
        Name: foodName,
        Id: foodId, Metadata: metadata,
    }
    foodBytes, err := json.Marshal(food)
    if err != nil {
        return shim.Error(fmt.Sprintf("marshal food error: %s", err))
    }
    if err := stub.PutState(constructFoodKey(foodId), foodBytes); err !=
    nil
        { return shim.Error(fmt.Sprintf("save food error: %s", err))
    }
    user := new(User)
    // Deserialize the user
    if err := json.Unmarshal(userBytes, user); err != nil {
        return shim.Error(fmt.Sprintf("unmarshal user error: %s", err))
    }
    user.Foods = append(user.Foods, foodId)
    // Serialized users
    userBytes, err = json.Marshal(user)
    if err != nil {
        return shim.Error(fmt.Sprintf("marshal user error: %s", err))
    }
    if err := stub.PutState(constructUserKey(user.Id), userBytes); err
    != nil
        { return shim.Error(fmt.Sprintf("update user error: %s", err))
    }
    // History of changes in food distribution
    history := &FoodHistory{

```

```

FoodId:foo dId, OriginOwnerId: originOwner, CurrentOwnerId:
ownerId,
}
historyBytes, err := json.Marshal(history) if err != nil {
    return shim.Error(fmt.Sprintf("marshal food history error: %s",
err))
}
historyKey, err := stub.CreateCompositeKey("history", []string{
foodId,
    originOwner, ownerId,
})
if err != nil {
    return shim.Error(fmt.Sprintf("create key error: %s", err))
}
if err := stub.PutState(historyKey, historyBytes); err != nil {
    return shim.Error(fmt.Sprintf("save food history error: %s", err))
}
return shim.Success(nil)
}

```

Distribution of Foodstuffs

This function enables the ingredients circulation between users, that is, between sellers of ingredients, and saves a record of the changes to the blockchain.

```

// Change in ingredients
func (c *IngredientsExchangeCC) ingredientExchange(stub shim.
ChaincodeStubInterface, args []string) pb.Response {
    // Check the number of parameters
    if len(args) != 3 {
        return shim.Error("not enough args")
    }
    // Verify the correctness of the parameters
    ownerId := args[0]
    ingredientId := args[1] currentOwnerId := args[2]
    if ownerId == "" || ingredientId == "" || currentOwnerId == ""
        { return shim.Error("invalid args") }
    }
    // Verify the existence of the data
    originOwnerBytes, err := stub.GetState(constructUserKey(ownerId))
    if err != nil || len(originOwnerBytes) == 0
        { return shim.Error("user not found") }
    }
    currentOwnerBytes, err := stub.GetState(constructUserKey(
(currentOwnerId)) if err != nil || len(currentOwnerBytes) == 0 {
        return shim.Error("user not found")
    }
    assetBytes, err := stub.GetState(constructIngredientKey(
(ingredientId)) if err != nil || len(assetBytes) == 0 {
        return shim.Error("asset not found")
    }

```

```

    // Verify that the original owner does have the ingredients for the
    current change
    originOwner := new(User)
    // Deserialize the user
    if err := json.Unmarshal(originOwnerBytes, originOwner); err != nil {
        return shim.Error(fmt.Sprintf("unmarshal user error: %s", err))
    }
    aidexist := false
    for _, aid := range originOwner.Ingredients
    { if aid == ingredientId { aidexist = true break
    }
    }
    if !aidexist {
        return shim.Error("ingredient owner not match")
    }
    // Write Status
    ingredientIds := make([]string, 0)
    for _, aid := range originOwner.Ingredients
    { if aid == ingredientId { continue
    }
    ingredientIds = append(ingredientIds, aid)
    }
    originOwner.Ingredients = ingredientIds
    originOwnerBytes, err = json.Marshal(originOwner)
    if err != nil {
        return shim.Error(fmt.Sprintf("marshal user error: %s", err))
    }
    if err := stub.PutState(constructUserKey(ownerId),
    originOwnerBytes); err != nil
    { return shim.Error(fmt.Sprintf("update user error: %s", err))
    }
    // The current owner inserts the ingredient id
    currentOwner := new(User)
    // Deserialize the user
    if err := json.Unmarshal(currentOwnerBytes, currentOwner); err != nil
    {
        return shim.Error(fmt.Sprintf("unmarshal user error: %s", err))
    }
    currentOwner.Ingredients = append(currentOwner.Ingredients,
    ingredientId)
    currentOwnerBytes, err = json.Marshal(currentOwner)
    if err != nil {
        return shim.Error(fmt.Sprintf("marshal user error: %s", err))
    }
    if err := stub.PutState(constructUserKey(currentOwnerId),
    currentOwnerBytes); err != nil
    { return shim.Error(fmt.Sprintf("update user error: %s", err))
    }
    // Insert ingredient change record
    history := &IngredientHistory{
        IngredientId: ingredientId, OriginOwnerId: ownerId,
        CurrentOwnerId: currentOwnerId,
    }
    historyBytes, err := json.Marshal(history)
    if err != nil {

```

```

        return shim.Error(fmt.Sprintf("marshal ingredient history error: %s",
                                         err))
    }
    historyKey, err := stub.CreateCompositeKey("history", []string{
        ingredientId,
        ownerId, currentOwnerId,
    })
    if err != nil {
        return shim.Error(fmt.Sprintf("create key error: %s", err))
    }
    if err := stub.PutState(historyKey, historyBytes); err != nil {
        return shim.Error(fmt.Sprintf("save ingredient history error: %s",
                                         err))
    }
    return shim.Success(nil)
}

```

Food Distribution

This function enables the flow of food, i.e., from the food store to the food dispatcher and from the food dispatcher to the user who ordered the food, and sets up and saves a record of the flow on the blockchain.

```

// Changes in food distribution
func (c *IngredientsExchangeCC) foodExchange(stub shim.ChaincodeStubInterface, args []string) pb.Response {
    // Check the number of parameters
    if len(args) != 3 {
        return shim.Error("not enough args")
    }
    // Verify the correctness of the parameters
    ownerId := args[0] foodId := args[1]
    currentOwnerId := args[2]
    if ownerId == "" || foodId == "" || currentOwnerId == "" {
        return shim.Error("invalid args")
    }
    // Verify the existence of the data
    originOwnerBytes, err := stub.GetState(constructUserKey(ownerId))
    if err != nil || len(originOwnerBytes) == 0 {
        return shim.Error("user not found")
    }
    currentOwnerBytes, err := stub.GetState(constructUserKey(
        currentOwnerId)) if err != nil || len(currentOwnerBytes) == 0 {
        return shim.Error("user not found")
    }
    foodBytes, err := stub.GetState(constructFoodKey(foodId)) if err != nil ||
        len(foodBytes) == 0 {
        return shim.Error("food not found")
    }
    // Verify that the original owner does own the current change in food
    distribution
}

```

```

originOwner := new(User)
// Deserialize the user
if err := json.Unmarshal(originOwnerBytes, originOwner); err != nil
{
    return shim.Error(fmt.Sprintf("unmarshal user error: %s", err))
}
aidexist := false
for _, aid := range originOwner.Foods
{
    if aid == foodId { aidexist = true break
    }
}
if !aidexist {
    return shim.Error("food owner not match")
}
// Write Status
foodIds := make([]string, 0)
for _, aid := range originOwner.Foods
{
    if aid == foodId { continue
    }
    foodIds = append(foodIds, aid)
}
originOwner.Foods = foodIds
originOwnerBytes, err = json.Marshal(originOwner) if err != nil {
    return shim.Error(fmt.Sprintf("marshal user error: %s", err))
}
if err := stub.PutState(constructUserKey(ownerId),
originOwnerBytes); err != nil
{
    return shim.Error(fmt.Sprintf("update user error: %s", err))
}
// The current owner inserts the food distribution id
currentOwner := new(User)
// Deserialize the user
if err := json.Unmarshal(currentOwnerBytes, currentOwner); err != nil
{
    return shim.Error(fmt.Sprintf("unmarshal user error: %s", err))
}
currentOwner.Foods = append(currentOwner.Foods, foodId)
currentOwnerBytes, err = json.Marshal(currentOwner)
if err != nil {
    return shim.Error(fmt.Sprintf("marshal user error: %s", err))
}
if err := stub.PutState(constructUserKey(currentOwnerId),
currentOwnerBytes); err != nil
{
    return shim.Error(fmt.Sprintf("update user error: %s", err))
}
// Insert food distribution change log
history := &FoodHistory{
    FoodId: foodId, OriginOwnerId: ownerId, CurrentOwnerId:
currentOwnerId,
}
historyBytes, err := json.Marshal(history) if err != nil {
    return shim.Error(fmt.Sprintf("marshal food history error: %s",

```

```

    err))
    }
    historyKey, err := stub.CreateCompositeKey("history", []string{
foodId,
    ownerId, currentOwnerId,
})
if err != nil {
    return shim.Error(fmt.Sprintf("create key error: %s", err))
}
if err := stub.PutState(historyKey, historyBytes); err != nil {
    return shim.Error(fmt.Sprintf("save food history error: %s", err))
}
return shim.Success(nil)
}

```

User Inquiry, Food Ingredient Inquiry, and Food Delivery Inquiry

Because of quite similar nature, we introduce these modules together here. User inquiry allows one to query the user's basic information as well as the ingredients and food items owned. Food ingredient queries can query the ingredients' basic knowledge. Food queries can query basic information about the food, including which ingredients it is made from.

```

// User queries
func (c *IngredientsExchangeCC) queryUser(stub shim.
ChaincodeStubInterface, args []string) pb.Response {
    // Check the number of parameters
    if len(args) != 1 {
        return shim.Error("not enough args")
    }
    // Verify the correctness of the parameters
    ownerId := args[0]
    if ownerId == "" {
        return shim.Error("invalid args")
    }
    // Verify the existence of the data
    userBytes, err := stub.GetState(constructUserKey(ownerId))
    if err != nil || len(userBytes) == 0
    { return shim.Error("user not found")
    }
    return shim.Success(userBytes)
}
// Ingredients Search
func (c *IngredientsExchangeCC) queryIngredient(stub shim.
ChaincodeStubInterface, args []string) pb.Response {
    // Check the number of parameters
    if len(args) != 1 {
        return shim.Error("not enough args")
    }
    // verify the correctness of the parameters ingredientId := args[0] if

```

```

ingredientId == "" {
    return shim.Error("invalid args")
}
// Verify the existence of the data
ingredientBytes, err := stub.GetState(constructIngredientKey
(ingredientId))
if err != nil || len(ingredientBytes) == 0
{ return shim.Error("ingredient not found")
}
return shim.Success(ingredientBytes)
}
// Food inquiries
func (c *IngredientsExchangeCC) queryFood(stub shim.
ChaincodeStubInterface, args []string) pb.Response {
    // Check the number of parameters
    if len(args) != 1 {
        return shim.Error("not enough args")
    }
    // Verify the correctness of the parameters
    foodId := args[0]
    if foodId == "" {
        return shim.Error("invalid args")
    }
    // Verify the existence of the data
    foodBytes, err := stub.GetState(constructFoodKey(foodId))
    if err != nil || len(foodBytes) == 0
    { return shim.Error("food not found")
    }
    return shim.Success(foodBytes)
}

```

Foodstuffs Circulation Records Search and Food Circulation Records Search

Food circulation records can be queried chronologically to determine the movement of ingredients between food suppliers and how they end up in what food. The food circulation record query searches the circulation record of food delivered to the subscriber from the beginning to the end.

```

// Food Change History Enquiry
func (c *IngredientsExchangeCC) queryIngredientHistory(stub shim.
ChaincodeStubInterface, args []string) pb.Response {
    // Check the number of parameters
    if len(args) != 2 && len(args) != 1 {
        return shim.Error("not enough args")
    }
    // verify the correctness of the parameters ingredientId := args[0] if
    ingredientId == "" {
        return shim.Error("invalid args")
    }
    queryType := "all" if len(args) == 2 {

```

```
    queryType = args[1]
}
if queryType != "all" && queryType != "enroll" && queryType != "exchange" { return shim.Error(fmt.Sprintf("queryType unknown %s",
queryType))
}
// Verify the existence of the data
ingredientBytes, err := stub.GetState(constructIngredientKey
(ingredientId))
if err != nil || len(ingredientBytes) == 0
{ return shim.Error("ingredient not found")
}
// Search for relevant data
keys := make([]string, 0)
keys = append(keys, ingredientId) switch queryType {
case "enroll":
    keys = append(keys, originOwner) case "exchange", "all":
default:
    return shim.Error(fmt.Sprintf("unsupport queryType: %s",
queryType))
}
result, err := stub.GetStateByPartialCompositeKey("history", keys)
if err != nil {
    return shim.Error(fmt.Sprintf("query history error: %s", err))
}
defer result.Close()
histories := make([]*IngredientHistory, 0) for result.HasNext() {
    historyVal, err := result.Next() if err != nil {
        return shim.Error(fmt.Sprintf("query error: %s", err))
    }
    history := new(IngredientHistory)
    if err := json.Unmarshal(historyVal.GetValue(), history); err !=
nil
        { return shim.Error(fmt.Sprintf("unmarshal error: %s", err))
    }
    // Filter out records that are not ingredient transfers
    if queryType == "exchange" && history.OriginOwnerId == originOwner {
        continue
    }
    histories = append(histories, history)
}
historiesBytes, err := json.Marshal(histories) if err != nil {
    return shim.Error(fmt.Sprintf("marshal error: %s", err))
}
return shim.Success(historiesBytes)
}
// Food Change History Enquiry
func (c *IngredientsExchangeCC) queryFoodHistory(stub shim.
ChaincodeStubInterface, args []string)
pb.Response {
// Check the number of parameters
if len(args) != 2 && len(args) != 1 {
    return shim.Error("not enough args")
```

```

    }
    // Verify the correctness of the parameters
    foodId := args[0]
    if foodId == "" {
        return shim.Error("invalid args")
    }
    queryType := "all" if len(args) == 2 {
        queryType = args[1]
    }
    if queryType != "all" && queryType != "enroll" && queryType != "exchange" { return shim.Error(fmt.Sprintf("queryType unknown %s",
    queryType))
    }
    // Verify the existence of the data
    foodBytes, err := stub.GetState(constructFoodKey(foodId))
    if err != nil || len(foodBytes) == 0
    { return shim.Error("food not found")
    }
    // Search for relevant data
    keys := make([]string, 0)
    keys = append(keys, foodId) switch queryType {
    case "enroll":
        keys = append(keys, originOwner) case "exchange", "all":
    default:
        return shim.Error(fmt.Sprintf("unsupport queryType: %s",
    queryType))
    }
    result, err := stub.GetStateByPartialCompositeKey("history", keys)
    if err != nil {
        return shim.Error(fmt.Sprintf("query history error: %s", err))
    }
    defer result.Close()
    histories := make([]*FoodHistory, 0) for result.HasNext() {
        historyVal, err := result.Next() if err != nil {
            return shim.Error(fmt.Sprintf("query error: %s", err))
        }
        history := new(FoodHistory)
        if err := json.Unmarshal(historyVal.GetValue(), history); err != nil
        { return shim.Error(fmt.Sprintf("unmarshal error: %s", err))
        }
        // Filter out records that are not ingredient transfers
        if queryType == "exchange" && history.OriginOwnerId == originOwner {
            continue
        }
        histories = append(histories, history)
    }
    historiesBytes, err := json.Marshal(histories) if err != nil {
        return shim.Error(fmt.Sprintf("marshal error: %s", err))
    }
    return shim.Success(historiesBytes)
}

```

9.2.7 Leveraging the Node.js SDK

The case implements back-end interaction with blockchain through Node.js SDK and front-end operation of data using Express framework. Some key functions in the SDK are shown below.

```
// Initialization
var express = require('express')
var bodyParser = require('body-parser') var app = express()
// Use the static files in the public folder
app.use(express.static('public'));
app.use('/', express.static( dirname + '/public')); app.use
(bodyParser.urlencoded({ extended: false })) app.use(bodyParser.json
()))
// Define the route
app.post('/users',function (req,res) {...}) // User registration
app.get('/users',function (req,res) {...}) // user query app.post('/
ingredients/enroll', function (req,res) {...}) // ingredient
registration app.get('/ingredients/get/:id',function (req,res)
{...}) // ingredient inquiry app.post('/ingredients/exchange',
function (req,res){...}) // Ingredients transfer transaction
app.get('/ingredients/exchange/history',function(req,res){...})
// Food transaction history query app.post('/foods/enroll',function
(req,res){...}) // food distribution registration app.post('/foods/
exchange',function(req,res){...}) // food distribution circulation
app.get('/foods/exchange/history',function(req,res){...}) // food
distribution history query app.delete('/deleteusers',function(req,
res){...}) // User cancellation
// Blockchain management component
var options = {
  user_id: 'Admin@org1.zjucst.com', // User's id
  user_msp_id: 'Org1MSP',
  channel_id: 'assetschannel', // channel name
  chaincode_id: 'assets',
  network_url: 'grpc://localhost:27051', // communication address of
the peer node
  peer_url: 'grpc://localhost:27051',
  orderer_url: 'grpc://localhost:7050', // communication address of
the orderer node
  privateKeyFolder: ,
  signedCert: , peer_tls_cacerts: , orderer_tls_cacerts: ,
  tls_cacerts: ,
  server_hostname: "peer0.org1.zjucst.com"
};
```

PrivateKeyFolder, signedCert, Peer_TLs_cacerts, OrderER_TLs_cacerts, and TLs_cacerts in the above code are the addresses of the relevant certificates.

Next comes the operation of connecting the blockchain.

```
console.log("Load privateKey and signedCert");
client = new hfc();
```

```

// Create a user object based on the address set above
var createUserOpt = {
  username: options.user_id,
  mspid: options.msp_id,
  cryptoContent: { privateKey: getKeyFilesInDir(options.
privateKeyFolder) [0] ,
    signedCert: options.signedCert }
}
const store = await sdkUtils.newKeyValueStore({
  path: "/tmp/fabric-client-stateStore/"
});
client.setStateStore(store);
let user = await client.createUser(createUserOpt);
channel = client.newChannel(options.channel_id);
let data = fs.readFileSync(options.tls_cacerts);
let peer = client.newPeer(options.network_url,
{
  pem: Buffer.from(data).toString(),
  'ssl-target-name-override': options.server_hostname
})
;
peer.setName("peer0");
channel.addPeer(peer); // add the peer node to the channel

// Generate transaction requests, submit to blockchain, get results
let transaction_id = await client.newTransactionID();
console.log("Assigning transaction_id: ", transaction_id.
transaction_id);
const request = {
  chaincodeId: options.chaincode_id,
  txId: transaction_id,
  fcn: fcn,
  args: args
};
let query_responses = await channel.queryByChaincode(request);

```

If a query request is submitted to the blockchain, the query_responses returned is the result of that query. If successful, the query information is returned to the front-end page of the routing function.

```

// query function, in the query function
if (!query_responses.length) {
  console.log("No payloads were returned from query");
} else {
  console.log("Query result count = ", query_responses.length)
}
if (query_responses[0] instanceof Error) {
  console.error("error from query = ", query_responses[0]); // error
handling
}
return query_responses[0].toString(); // return the result as a
string

```

If a request is submitted to the blockchain to write data, the query_result returned contains the legitimacy of this modification request. If legal, a new event object will be built to register the event with the blockchain, bearing the registration result.

```
let eh = await channel.newChannelEventHub('localhost:27051'); // set the event listening port
let data = fs.readFileSync(options.peer_tls_cacerts);
let grpcOpts = {
  pem: Buffer.from(data).toString(),
  'ssl-target-name-override': options.server_hostname
} // Set the connection properties
eh.connect();
// Set the connection time limit
let txPromise = new Promise((resolve, reject) => {
  let handle = setTimeout(() =>
    { eh.disconnect(); reject(); },
    30000);
  eh.registerTxEvent(transactionID, (tx, code) =>
    { clearTimeout(handle); eh.unregisterTxEvent(transactionID); eh.disconnect(); }

  if (code !== 'VALID')
    { console.error(
      'The transaction was invalid, code = ' + code); reject(); }
    else {
      console.log(
        'The transaction has been committed on peer ' + eh.getPeerAddr());
      resolve();
    }
  });
  eventPromises.push(txPromise);
var sendPromise = await channel.sendTransaction(requests);
return Promise.all([sendPromise].concat(eventPromises)).then
(result) =>
  { console.log(' event promise all complete');
    return result; // The contract code is executed and the result is
returned
  }).catch((err) => {
  console.error(
    'Failed to send transaction and get notifications within the timeout
period.')
  );
  return 'Failed to send transaction and get notifications within the
timeout period.';
});
```

After the modification is executed successfully, the function returns the result and the status in the routing function for front-end operations.

9.2.8 Deployment Implementation

This project employs a fabric multi-machine deployment for testing contract code during the deployment and installation process, with the aim of showing readers a simple deployment process. Readers can configure the channels and nodes of the platform according to the project requirements.

Network Configuration

The network architecture enabled by the project is shown in Table 9.1.

Encryption Configuration

Go to the my-network folder and first write the encryption configuration file crypto-config.yaml for the organization (refer to the project source code), and then execute the command in the terminal.

```
... /bin/cryptogen generate --config=. /crypto-config.yaml
```

The terminal returns two lines of the organization's domain name to indicate success, and a crypto-config folder is added to the my-network folder to store the encryption configuration file. If the ca mechanism is enabled, additional environment variables are required.

```
export BYFN_CA1_PRIVATE_KEY=$(cd crypto-config/peerOrganizations/org1.zjucst.com/ca && ls *_sk)
export BYFN_CA2_PRIVATE_KEY=$(cd crypto-config/peerOrganizations/org2.zjucst.com/ca && ls *_sk)
BYFN_CA*_PRIVATE_KEY variable name definition should match the one in docker-compose-ca.yaml .
```

Channel Configuration

(a) Generating genesis blocks

Table 9.1 Project enabled network architecture

Name	IP address
orderer.zjucst.com	IP address of host 1
peer0.org1.zjucst.com	IP address of host 1
peer1.org1.zjucst.com	IP address of host 1
peer0.org2.zjucst.com	IP address of host 2
peer1.org2.zjucst.com	IP address of host 2

Note: If you enable raft multi-srot node consensus, you need to add the corresponding order node

First, write the configuration file configtx.yaml for the channels (refer to the project source code) and create a channel-Artifacts folder, where the configuration files for the channels are stored. Next, configure the environment variable FABRIC_CFG_PATH=\$PWD on the terminal to tell the ConfigtxGen tool the current working directory. Finally, the ConfigtxGen tool can be invoked to create the creation block by executing commands on the terminal.

```
... /bin/configtxgen -profile TwoOrgsOrdererGenesis -channelID  
byfn-sys-channel -outputBlock ./channel-artifacts/genesis.block
```

The terminal returns the last two lines—Generating genesis block and Writing genesis block—to indicate success.

(b) Channel allocation transactions

Configure the environment variable export CHANNEL_NAME=mychannel in the terminal and call the configtxgen utility to create the channel configuration transaction at

```
... /bin/configtxgen -profile TwoOrgsChannel  
-outputCreateChannelTx ./channel-artifacts/channel.tx  
-channelID $CHANNEL_NAME
```

The terminal returns the last two lines—Generating new channel configtx and Writing new channel tx—to indicate success.

(c) Defining the anchor node

Call the configtxgen tool to define the anchor nodes for each organization.

```
#Organization 1  
... /bin/configtxgen -profile TwoOrgsChannel  
-outputAnchorPeersUpdate ./channel-artifacts/  
Org1MSPanchor.tx -channelID $CHANNEL_NAME -asOrg Org1MSP  
#Organization 2 (组织2)  
... /bin/configtxgen -profile TwoOrgsChannel  
-outputAnchorPeersUpdate ./channel-artifacts/  
Org2MSPanchor.tx -channelID $CHANNEL_NAME -asOrg Org2MSP
```

The terminal returns the last two lines—Generating anchor peer update and Writing anchor peer update—to indicate success.

Start All Containers

First configure the .env file in the my-network directory (refer to the project source code). Execute the command in the terminal.

```
docker-compose -f docker-compose-cli.yaml up -d 2>&1
```

If CA and RAFT awareness are enabled, add the additional parameter: -f [profile name]. Use the docker ps command to check the container startup status, at least four containers should be running: cli, [orderer.zjucst.com](#), [peer0.org1.zjucst.com](#), [peer1.org1.zjucst.com](#).

```
#Enter the CLI container
docker exec -it cli bash
```

(a) Configure environment variables for calling the peer0.org1.zjucst.com node

```
# Execute the command in the CLI container terminal:
CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/
hyperledger/fabric/peer/crypto/peerOrganizations/ org1.zjucst.
com /users/Admin@org1.zjucst.com/msp
    CORE_PEER_ADDRESS=peer0.org1.zjucst.com:7051
    CORE_PEER_LOCALMSPID="Org1MSP"
    CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/
hyperledger/fabric/peer/crypto/peerOrganizations/org1.zjucst.
com/peers/ peer0.org1.zjucst.com/tls/ca.crt
    export CHANNEL_NAME=mychannel
```

(b) Create channel

```
peer channel create -o orderer.zjucst.com:7050 -c $CHANNEL_NAME -f
./channel-artifacts/channel.tx --tls --cafile
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
ordererOrganizations/ zjucst.com/orderers/orderer.zjucst.com/
msp/tlscacerts/tlsca.zjucst.com-cert.pem
```

The terminal returns Received block: 0 for success.

```
#peer0.org1.zjucst.com node join channel
peer channel join -b mychannel.block
```

The terminal returns Successfully submitted proposal to join channel, indicating success.

```
#Install contract code to peer0.org1.zjucst.com node
peer chaincode install -n mycc -v 1.0 -p github.com/chaincode/go
```

The terminal returns Installed remotely response:<status:200 payload: "OK">, indicating success.

Backup Configuration File to Host 2

First, make sure that host 1 can log into host 2 without a key over SSH. Open a separate terminal in my-network directory, and then execute the following command in the terminal.

```

IP=IP address of host 2
NAME=Username of host 2
CATALOG=directory where the file is stored on host 2 scp -r . /channel-
artifacts ${NAME}@${IP}:${CATALOG}
scp -r . /crypto-config
${NAME}@${IP}:${CATALOG} CLI_ID=$(docker ps | grep cli | awk '{print
$1}')
docker cp $CLI_ID:/opt/gopath/src/github.com/hyperledger/fabric/
peer/mychannel.block./ scp . /mychannel.block ${NAME}@${IP}:$-
{CATALOG}

```

If the transfer is successful, you will see the channel-artifacts folder, the crypto-config folder, and.

the mychannel.block file under the my-network folder on host 2.

Operating Mainframe 2

First configure the .env file in the my-network directory (same contents as host #1). Execute the command in the terminal.

```
docker-compose -f docker-compose-cli.yaml up -d 2>&1#start container
```

Run the docker ps to check the container open status and you will see three containers running.

After that, run the following command to make the mychannel.block file into the CLI container.

```

CLI_ID=$(docker ps | grep cli | awk '{print $1}')
docker cp . /mychannel.block ${CLI_ID}:/opt/gopath/src/github.com/
hyperledger/fabric/peer/
#Go to the CLI container
docker exec -it cli bash

```

- (a) Configure the environment variables for the node calling peer0.org2.zjucst.com
Execute the command in the CLI container terminal:

```

CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/
hyperledger/fabric/peer/crypto/peerOrganizations/ org2.zjucst.
com/users/Admin@org2.zjucst.com/msp
CORE_PEER_ADDRESS=peer0.org2.zjucst.com:9051
CORE_PEER_LOCALMSPID="Org2MSP"
CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/
hyperledger/fabric/peer/crypto/peerOrganizations/org2.zjucst.
com/peers/ peer0.org2.zjucst.com/tls/ca.crt
export CHANNEL_NAME=mychannel
#peer0.org2.zjucst.com node join channel
peer channel join -b mychannel.block

```

The terminal returns Successfully submitted proposal to join channel, indicating success.

```
#Install chaincode to peer0.org2.zjucst.com node
peer chaincode install -n mycc -v 1.0 -p github.com/chaincode/go
```

The terminal returns Installed remotely response:<status:200 payload: “OK”>, indicating success.

Return to Host 1 to Instantiate the Chaincode

```
# Execute the command in the terminal of the CLI container
peer chaincode instantiate -o orderer.zjucst.com:7050 --tls --cafile
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
ordererOrganizations/zjucst.com/orderers/orderer.zjucst.com/msp/
tlscacerts/tlsca.zjucst.com-cert.pem -C $CHANNEL_NAME -n mycc -v 1.0
-c '{"Args": []}' -P "AND ('Org1MSP.peer', 'Org2MSP.peer')"
```

This completes the network configuration!

Test with the Console

Execute the command: docker exec cli scripts/script-test.sh from a terminal in the my-network directory on host 1. This script creates a new user, and if executed successfully, you will see the initialization information for user “00001”.

Execute the same command from the terminal in the my-network directory of host 2. If the “00001” user message is also output, the multi-node configuration is successful. The stop-net.sh in the directory is used to shut down the network.

9.3 Summary

This chapter presents two real-world cases, a blockchain-based social heritage management platform case and a high-end food safety system case. We demonstrate the application of the fabric in the heritage management scenario and the food safety management scenario, respectively, to help readers increase their experience in practice. At the same time, we use the fabric developer mode to test the contract code and the multi-machine deployment network so that readers can better understand the fabric and do hands-on work.

Chapter 10

Use Cases and Detailed Explanation of Enterprise-Level Blockchain



Ethereum, Hyperledger, and other open-source blockchain platforms are still in the process of further development and improvement. There are a large number of problems in consensus efficiency, privacy protection, large-scale storage, regulatory access, which hinder the implementation and commercialization of applications on open-source platform projects. Hyperchain is an alliance chain platform explicitly designed for enterprise-level applications with complete functions and leading technologies. Many financial institutions so far have developed blockchain projects as the Hyperchain platform as a basis, directly connecting with the banking system. Some have already been implemented and promoted commercially. The previous chapters have reviewed the core principles and development practices of Hyperchain in detail. This chapter will present two examples of Hyperchain-based enterprise-level blockchain application projects closer to reality: accounts receivable system and travel ride-sharing platform.

10.1 Case Study of Hyperchain-Based Accounts Receivable Management System

In this section, based on the business requirements of accounts receivable of several banks and factoring companies, a Hyperchain-based accounts receivable management platform is built comprehensively, combined with the blockchain technology of Hyperchain Technology. This platform is mainly designed to solve the problems of complex financing, expensive financing, and opaque supply chain information for enterprises. The platform functions include online accounts receivable issuance, transfer, financing, and refinancing. The accounts receivable system designed in this section can be an example of blockchain in the financial field and provide references for other blockchain-based applications.

10.1.1 Project Introduction

With the gradual improvement of regulatory policies, the winter of developing the C-side consumer finance industry has arrived. On the contrary, B-end supply chain finance is still a blue ocean. It has attracted many commercial banks, third-party payment institutions, e-commerce giants, logistics enterprises, and P2P companies to carry out layout by virtue of its high integration with the industry.

Generally speaking, the amount of accounts receivable financing or prepayment financing is 70–80% of the total amount of funds. The amount of inventory financing is 30–50% of the value of goods. Combined with the three supply chain business scenarios of listed companies' accounts receivable, accounts prepaid, and inventory, the scale of China's supply chain finance market will reach 15.86 trillion yuan (\$2.5 trillion) in 2020. As existing players and new entrants deeply penetrate the market, supply chain finance will usher in a period of rapid development and is expected to reach 19.19 trillion (\$3 trillion) by 2022 (data source: Wind). But the traditional offline accounts receivable model has some pain points as follows.

- There are many silos of independent information. The independent ERP system is widely used in the supply chain, and the communication between enterprises is fragmented, making it challenging to integrate the data between industrial chains.
- Credit transmission in the industry chain is complex. The credit of the core enterprise can only be transmitted to the first- and second-tier suppliers but not to the suppliers afterward, and these SMEs cannot rely on the credit of the core enterprise for financing.
- It is not easy to prove the trade authenticity. It is more difficult for SMEs to verify their repayment ability and confirm a reliable trade relationship after getting the receivables. Therefore, financing is generally difficult and expensive.
- Unable to effectively control the performance risk leads to frequent moral hazard in pursuit of their interests, the participants conceal negative information about the enterprise to obtain credit fraudulently, fabricate trade transactions to get loans and bury the actual flow of funds after loans.

To this end, Hyperchain Technology launched the Filoop supply chain platform, which offers multilevel receivables circulation and financing services based on blockchain and transferable digital receivables certificates for core enterprises, member enterprises, and multilevel suppliers. Suppliers can easily split, transfer, and finance their receivables by the receivable certificates based on the credit of the core enterprise and effectively benefit the harder-to-reach end suppliers other than the first-tier suppliers. The credit resources of core enterprises have achieved an effective multilevel transmission. By introducing external financial institutions, we provide low-cost financing rates for suppliers' receivables and ultimately build a bridge between high-quality assets and funds.

This accounts receivable platform involves four significant roles: suppliers, core companies, member companies, and factors/banks. The accounts receivable flow operation and life cycle among users are shown in Fig. 10.1.

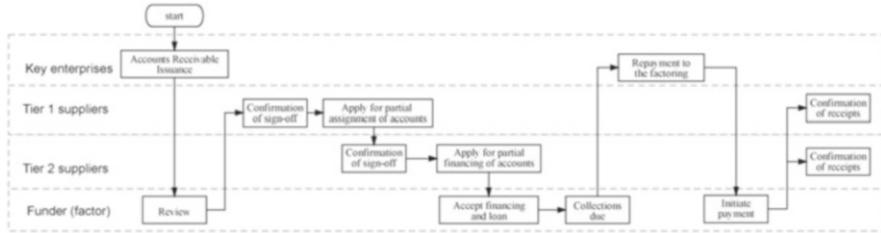


Fig. 10.1 Accounts receivable flow operations and life cycle

We explain each process in Fig. 10.1 in detail.

1. The core company issues digital vouchers for accounts receivable to the first-tier supplier on the accounts payable in the actual trade situation.
2. Factors review the billed receivables.
3. The Tier-1 supplier confirms the sign-off of the accounts receivable.
4. If the Tier-1 supplier has relatively strong negotiating power, it may transfer the account amount in full or in part to pay its upstream Tier-2 supplier when conducting the relevant trade.
5. The Tier-2 supplier confirms to sign the accounts receivable.
6. If the secondary supplier needs funds, it can apply for full or partial financing by factors and pay the corresponding amount to the financing party after acceptance.
7. When the accounts are due, the factor initiates collection from the issuer of the receivable certificate, i.e., the core company, which in return makes repayment.
8. For unfinanced accounts receivable, the factor shall pay the corresponding amount to the holders of receivables at all levels in turn when the accounts are due.

10.1.2 System Functional Analysis

The receivables platform built on Hyperchain contains the following functions, as shown in Fig. 10.2.

1. On-chain information: Users can query the information recorded on the blockchain, including transaction information, accounts receivable information, block information, etc.
2. Quota management: Factors can set the amount of receivables certificate issuance to different core enterprises according to their credit qualification and transaction records, etc., and set up the amount of receivables voucher for diverse suppliers.
3. Accounts receivable business: including the issuance, transfer, financing, refinancing, and cash collection of accounts receivable digital certificate business.

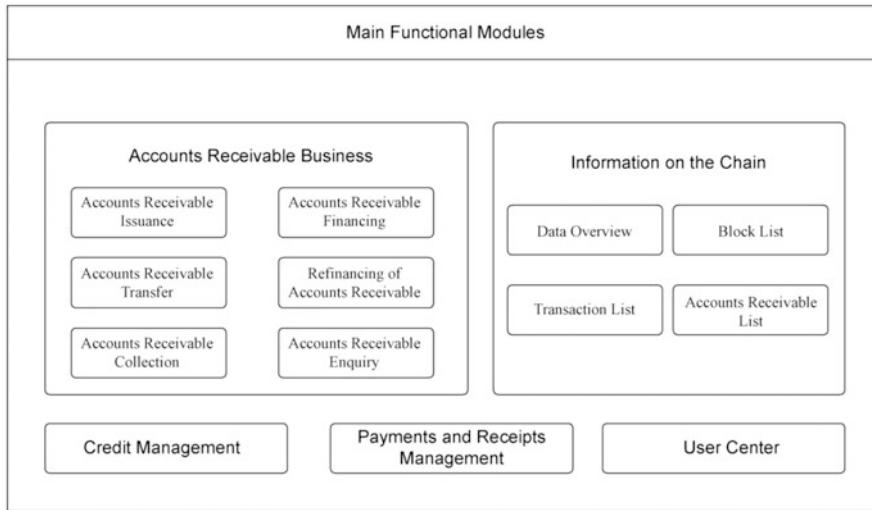


Fig. 10.2 Main functions of accounts receivable platform

Electronic Voucher Issuance for Accounts Receivable

Under this module, the core enterprise issues a blockchain receivables digital certificate to the supplier on its natural trade background. After the fund is approved, the certificate will be held by the supplier, and the transfer of the receivable will be completed at the legal level. Online issuance and electronic signature shorten the time of material delivery and signature and improve efficiency.

Accounts Receivable Transfer for Payment

The holder of an accounts receivable digital certificate may transfer the certificate to another person as a payment settlement instrument on an as-needed split basis, on a bona fide trade relationship, while the accounts receivable is active. The new holder can also split the voucher transfer to other business parties. Under this model, the credit of the core enterprise penetrates at multiple levels throughout the supply chain. Meanwhile, the holder, for its part, does not have to realize the traditionally hard-to-liquidate receivables before using them for payment, but can instead pay directly using the receivable vouchers, reducing the interest costs arising from liquidation in circulation.

Accounts Receivable Financing

Holders of accounts receivable certifications can apply for financing on the platform, and the factor on the platform will review the application online and release the funds quickly once it is approved. The online application, review, and signing of the agreement shortens the release time from the traditional 1 month to a few hours, significantly improving financing efficiency.

By way of blockchain, smart contracts, and other technologies, at the current stage, the Filoop accounts receivable platform realizes the main business processes of accounts receivable on-chain issuance, transfer, financing, refinancing, and redemption as described above. Likewise, it is committed to expanding more on-chain financial scenarios, such as prepayment financing, inventory financing, and international letters of credit. By continuously accumulating on-chain data, using historical data and current information to build measurement indicators has enhanced the science of contract design and established a more efficient and accurate risk control model. Based on legal compliance, Filoop supply chain will follow the policy, innovate business model through new technology, develop financial service business scenarios, and build a harmonious and co-prosperous on-chain supply chain financial ecosystem.

10.1.3 General System Design

General Idea

The multistage receivables transfer platform on blockchain adopts the credit spill-over of the core enterprise. It builds an industrial network developed between the multistage upstream suppliers of the core enterprise to support actual trade. By relying on the natural trade background of tier 1 suppliers and core enterprises, accounts receivable assets certificates can provide comprehensive financial services such as financing, payment and settlement, process optimization, and so on for suppliers at all levels. The platform provides user management, quota management, accounts receivable issuance, transfer payment, financing, and collection so that electronic certificates of accounts receivable can flow freely. After the electronic accounts are receivable, they can be split and flowed. Enterprises can partially finance the electronic vouchers of accounts receivable as needed and then collect the remaining electronic vouchers of accounts receivable when they expire to reduce their financing cost. By connecting with the core enterprise ERP system, the platform can directly obtain trade data and invoice information to ensure the authenticity and reliability of essential assets and reduce the possibility of financing based on false trade and invoices.

The platform consists of four layers: client layer, back-end API layer, back-end business layer, and underlying data storage layer.

- At the client level, the PC side provides a browser as the user entrance, through which users can carry out operations such as registration, limit management, receivables issuance, transfer, financing, and collection.
- On the one hand, the back-end API layer provides an interface for the client so that the client can obtain data for display. Meanwhile, it offers interfaces for other services so that the risk control system and post-loan management system can get trade data and repayment information from the receivables financing platform.
- The back-end business layer handles various business logic, calls to third-party services, data synchronization (databases and blockchains), and more.
- The underlying data storage layer involves traditional relational database and blockchain storage. Traditional databases store the total amount of information, and general data retrieval operations are carried out directly in the database, which makes up for the defect that blockchain is not good at retrieval. Blockchain stores the key data information, including accounts, receivables, invoice, relevant operation, etc., with the number, holder, status, amount, face value, and other information of receivables on the chain. By putting critical data on the chain, the authenticity and reliability of such information are guaranteed by using blockchain's decentralized and not easily tampered characteristics. Accurate trade data can provide reliable primary data for the risk control system. The enterprise score, calculated by the risk control system using the risk control model, turns out to be more accurate. The capital can provide credit to more honest enterprises, forming a virtuous circle.

The receivables financing platform realized in this section improves the credibility of receivables through the trust mechanism of the Hyperchain blockchain platform. It guides customers from traditional offline channels to online Internet channels in terms of information technology, which lifts the convenience of business accumulates and precipitates data.

Application Architecture

Figure 10.3 shows the system module diagram. This system includes four modules: display, core business, data storage, and associated system.

Display Module

Users can complete the receivable, electronic voucher flow operation through the PC browser and realize the operation functions of account management, amount management, receivable issuance, transfer, financing, and collection through the visualized page and querying receivables and payments and receipts information.

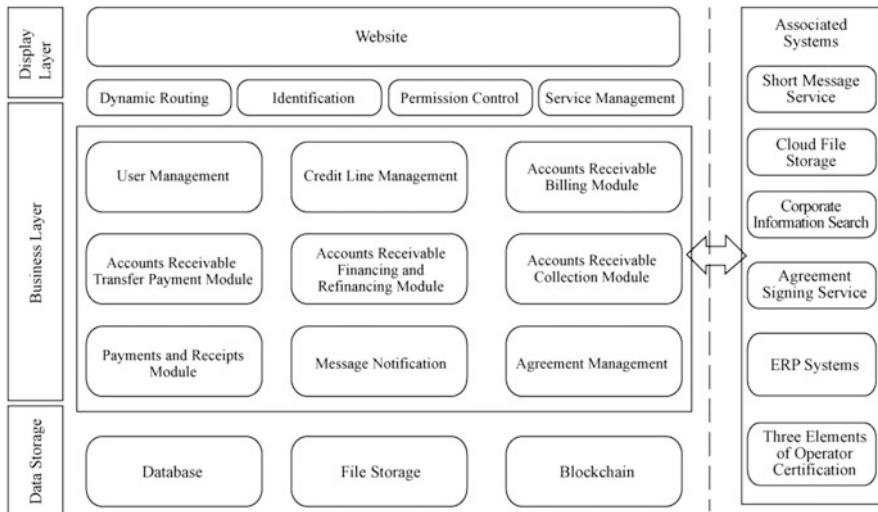


Fig. 10.3 System module diagram

Core Business Module

The receivables financing platform implemented on blockchain realizes user management, credit line management, accounts receivable issuance, transfer, financing, and collection. Connecting with ERP of core enterprises directly obtains trade data between core enterprises and first-tier suppliers, making it more authentic and reliable. The platform has realized the function of online electronic signature. Participants can sign the contract online instead of a paper contract agreement, reducing cost and increasing efficiency that facilitates financing.

Data Storage Module

The underlying Hyperchain-based platform uses the irreversible and non-tamperable characteristics of the data stored on the blockchain to ensure the authenticity and reliability of the electronic credentials of accounts receivable. In addition, writing smart contracts running on the blockchain controlled the circulation and transfer of receivables and confirmed the receivables.

Associated System Modules

The platform connects with many third-party services, such as file storage services, to achieve secure and reliable file storage; SMS service, realizing the sending of SMS verification code and important node messages; industrial and commercial

information inquiry service facilitates directly obtaining the related information of enterprises and simplifying the registration process; the three-factor authentication service of the cell phone number, name, and ID number verifies the authenticity of user information.

Main Function Design

In the system implemented in this section, the functional modules are user management, amount management, accounts receivable management, and payment collection management. Accounts receivable management includes accounts receivable issuance, financing, transfer payment, refinancing, collection, and inquiry. The following is an example of core enterprises' process to issue accounts receivables, as displayed in Fig. 10.4.

After filling in and uploading the receivables-related trade materials, the core enterprise initiates an application for receivables issuance, and the factor examines the application. After the factor uploads the relevant audit materials and approves the application, the receipt party (i.e., the supplier) can support the application and

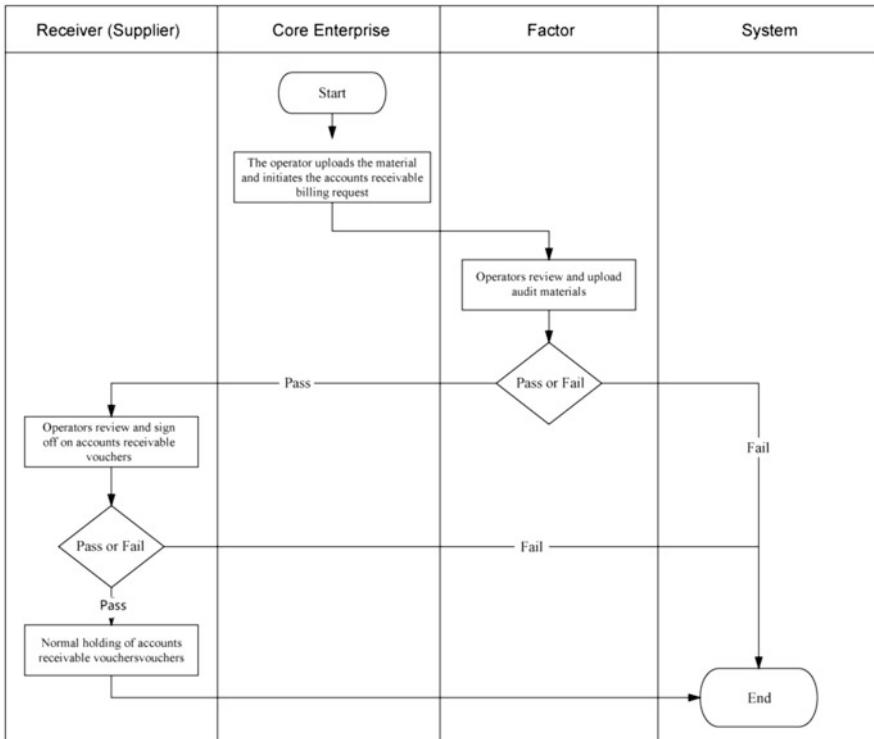


Fig. 10.4 Accounts receivable issuance process

typically hold the receivables certificate. If either the factor or the supplier chooses to reject the application during the whole process, the entire process ends.

Deployment Architecture

The system offers services for a wide range of suppliers, core enterprises, factors, banks, etc. The whole system network has two layers: intranet and extranet. The system network topology is shown in Fig. 10.5. The system application server is deployed in the extranet, and the Hyperchain blockchain platform and database server are deployed in the intranet.

When other supply chain enterprises join the alliance chain, they have to deploy blockchain nodes accordingly. In the initial collaboration, the newly added blockchain node server and the original federation must communicate to form a new partnership by consensus with the blockchain nodes.

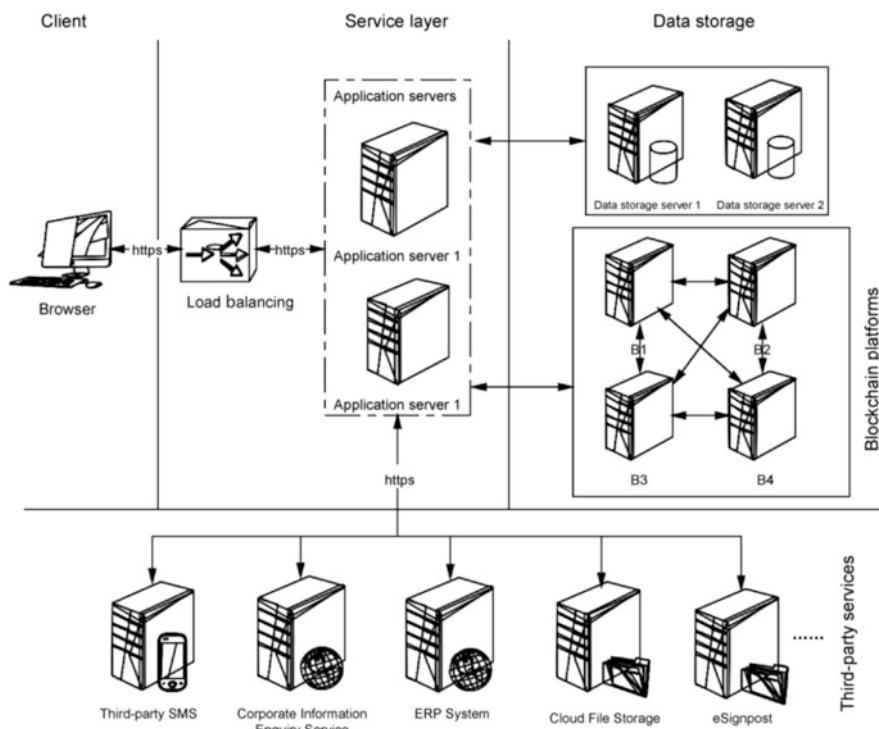


Fig. 10.5 System network topology diagram

10.1.4 Smart Contract Design

The main difference between blockchain applications and the traditional version is smart contracts implement the main business logic. Smart contracts are the main body of both program logic and data storage. This section focuses on smart contracts for the accounts receivable system.

Outline Design

Customers have registration and login functions, accounts receivable issuance, financing, transfer, refinancing, and collection. The smart contract designed in this section addresses the system's business requirements, including the structure body in the fields of business customers, accounts receivable, business operations, and some business operations methods core introduction to accounts receivable issuance, financing, and assignment. See Table 10.1 for details on the functions of the smart contract.

Contract Design

```
contract Filolink{
// Accounts receivable information
struct Bill {
    bytes32 billCode; // Account receivable code
    bytes32 billType; // Account receivable type
    bytes32 status; // Account receivable status
    uint billAmount; // Account receivable amount
    address holdCmyAddress; // Account receivable holder
}
```

Table 10.1 Function description of smart contract

Structure name	Description	Function
Enterprise customer structure	Maintain enterprise information of the system	The function includes recording the enterprise's on-chain address and basic information
Accounts receivable structure	The basic information of accounts receivable is maintained by the corresponding structure	Store basic details about accounts receivable
Business operation structure	The business information of each accounts receivable operation will be saved, and the structure maintains its corresponding basic information	Store accounts receivable operation information
Functional method	Functional methods for manipulating various structures	Record structure information through operation method

```

// Company information
struct Company {
    bytes32 cmyCode; // Company code
    bytes cmyName; // Company name
    address cmyAddress; // Company address
    bytes socialCreditCode; // Unified social credit code
    bytes licenseNumber; // Business license number
}

// Business records
struct Business {
    bytes32 businessNo; // Business number
    bytes32 businessType; // Business type
    bytes32 billCode; // Accounts receivable number
    uint businessDate; // Transaction time
    uint amount; // Amount involved
    address[] cmyAddress; // Transaction participant Address
}

// Accounts Receivable Number => Accounts Receivable
mapping(bytes32 => Bill) _Bill_Table;

// Company address => Company information
mapping(address => Company) _Company_Table;

// Business number => Business information
mapping(bytes32 => Business) _Business_Table;
}

```

Contract Method Design

Functional Methods

A common approach to business information on the chain.

```

function BusinessUpdate(bytes32 businessNo, bytes32 businessType ,
bytes32 billCode,
uint businessDate, uint amount, address[] cmyAddress) internal {
    Business newBusiness = _Business_Table[businessNo];
    newBusiness.businessNo = businessNo;
    newBusiness.businessType = businessType;
    newBusiness.billCode = billCode;
    newBusiness.businessDate = businessDate;
    newBusiness.amount = amount;
    newBusiness.cmyAddress = cmyAddress;
}

```

Company information registration: after companies register on the platform, their information will be stored on the chain.

```

function register(bytes32 cmyCode, bytes cmyName, bytes
socialCreditCode,
bytes licenseNumber, address cmyAddress) returns(uint) {
Company newCompany = _Company_Table[cmyAddress];
newCompany.cmyCode = cmyCode;
newCompany.cmyName = cmyName;
newCompany.cmyAddress = cmyAddress;
newCompany.socialCreditCode = socialCreditCode;
newCompany.licenseNumber = licenseNumber;
return 0;
}

```

Accounts receivable sign-off: the sign-off is invoked by the user of the supply chain enterprise. A new accounts receivable is generated after the core enterprise issues the receivable. After other related approvals in Feiloo's receivables system, the voucher is finally signed off by the supply chain enterprise, and the related information will be uploaded accordingly.

```

function issue(bytes32 billCode, bytes32 billType, bytes32 status,
uint billAmount, address holdCmyAddress, bytes32 businessNo, bytes32
businessType ,bytes32 billCode, uint businessDate, uint amount,
address[] cmyAddress) returns(uint) {
    // _Bill_Table is the bill structure that records the corresponding
bill number, bytes32Array[0] is the bill number
    _Bill_Table[billCode].billCode = billCode;
    _Bill_Table[billCode].billType = billType;
    _Bill_Table[billCode].status = status;
    _Bill_Table[billCode].billAmount = billAmount;
    _Bill_Table[billCode].holdCmyAddress = holdCmyAddress;
    // Save the sign-off operation and call the above generic method
    BusinessUpdate(businessNo,businessType,billCode,businessDate,
amount,cmyAddress);
    return 0;
}

```

Financing: Suppliers upload financing operations after confirming financing release from funders.

```

function fullAdvance(bytes32 billCode,bytes32 businessNo, bytes32
businessType ,bytes32
billCode, uint businessDate, uint amount, address[] cmyAddress)
returns(uint) {
    Bill bill = _Bill_Table[billCode];
    // Determine whether the note holding company is the current
operator
    if (bill.holdCmyAddress != msg.sender) {
        return 1;
    }
    // Update the status of the ticket to "financing completed" (Notes:
    _advanceSettled means financing completed)bill.status =
    _advanceSettled;
}

```

```

    // Business information for up-linking
    BusinessUpdate(businessNo,businessType,billCode,businessDate,
amount,cmyAddress) ;
    return 0;
}

```

Transfer sign-off completion: The note is transferred, and the transfer operation is uploaded when the transferee signs off on the transfer.

```

function fullTransfer(address to, bytes32 billCode, bytes32
businessNo, bytes32 businessType,
bytes32 billCode, uint businessDate, uint amount, address
[] cmyAddress) returns(uint) {
    Bill bill = _Bill_Table[billCode];
    if (bill.holdCmyAddress != msg.sender) {
        return 1;
    }
    // The business holding the receivable voucher is changed (Note: to is
    // the transferred party) bill.holdCmyAddress = to;
    // Business information for up-linking
    BusinessUpdate(businessNo,businessType,billCode,businessDate,
amount,cmyAddress) ;
    return 0;
}

```

The main difference between the contract method of refinancing, collection, and other functions and financing and transfer is the change of the enterprise holding the receivables certificate or the transformation of the receivables certificate state, which will not be described here.

10.1.5 System Security Design

In the receivables financing platform implemented in this section, in addition to applying mechanisms such as JWT authentication, graphical verification code, application layer permission control, encrypted password transmission, and HTTPS transmission in traditional applications to enhance system security, the blockchain platform has some mechanisms to ensure data security.

The multilevel encryption mechanism based on the blockchain platform can ensure that transactions sent to the blockchain platform are not tampered with, and information is transmitted with ciphertext between blockchain nodes. The consensus mechanism of the blockchain platform can ensure that each node stores all the transaction records, which guarantees the correctness and non-tamperability of the transactions. Once a blockchain transaction is completed, it cannot be changed. If changed, it must be approved by a majority of the nodes on the platform thus ensuring the security and credibility of the transaction. If a new node wants to join the alliance chain, it must get a certificate issued by the platform. This method limits

access to the federation to ensure data security and prevent malicious nodes from joining.

10.2 Case Study of Hyperchain-Based Ride-Hailing Platform

In this section, we design and implement a Hyperchain-based taxi-hailing platform. The platform generally has identity management, order aggregation, payment and other functions, and a decentralized online dating platform for passengers and drivers by invoking smart contracts deployed on the Hyperchain blockchain platform.

10.2.1 Project Introduction

In recent years, companies such as DiDi, transportation company, and Airbnb have become very popular, and they all involve a concept called “sharing economy.” However, the current travel market does not fully meet the sharing economy definition. Wikipedia defines the sharing economy as follows: Sharing economy refers to an organization or individual who owns idle resources giving the right to use them to others for a fee, and the sharer gets a reward for sharing the idle resources of others to create value.

The above definitions introduce that an essential feature of the sharing economy is to create values by idle resources. In today’s travel market, behind the policy of quietly increasing prices, freely regulating prices, and lowering subsidies, the original idle social vehicles and drivers are gradually decreasing in the condition of limited benefit. Besides, there is a certain degree of demand decline with the price increase on the demand-side. What ensued was that highly specialized groups exiting the traditional rental sector began to venture into the path of online taxi operations. Meanwhile, nowadays, many taxi services are no longer simply the integration of idle items, but a new industry transformation by the repurchase of vehicles and equipment by some professional teams.

Based on the sharing economy concept, we use blockchain technology to establish a distributed Internet travel platform—KVCOOGO. The platform has demand discovery, transaction aggregation, payment settlement, credit evaluation, and derivative services for multiple participants. With the help of blockchain, we will establish a low-cost and more equitable travel market and realize an actual sharing economy.

The interaction between passengers, drivers, and smart contracts is shown in Fig. 10.6.

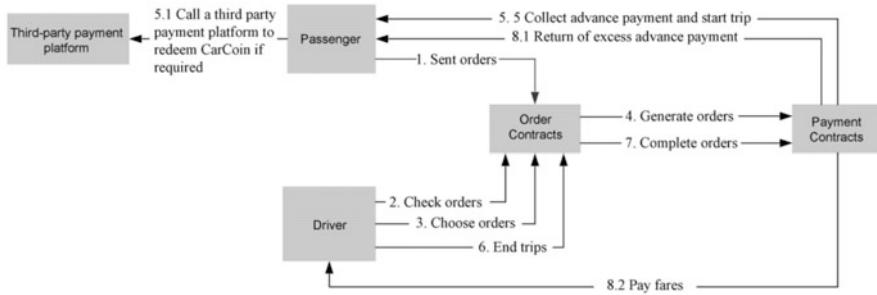


Fig. 10.6 The interaction between passengers, drivers, and smart contracts

10.2.2 System Functional Analysis

This system aims to establish a P2P ride-hailing platform for passengers and drivers. This section will analyze the two functions, respectively.

Passenger Function Analysis

The passenger function consists of five systems: identity system, positioning system, order system, payment system, and evaluation system.

Identity System

The identity system provides platform passengers with identity management-related functions, including registration, login, personal information management, etc. The user registers as a passenger through the App. When writing, the system generates a public and private key pair for the passengers as their identity on the blockchain. Additionally, the system has the passengers' blockchain addresses based on their public keys. It will ask the user to enter the mobile phone number as the user name during registration. Afterward, the user can log in to the system.

The personal information includes basic information like the passenger's name, mobile phone number, profile picture, and standard address. Passengers can complete and modify their personal information after logging in.

Positioning System

KVCOOGO has a positioning service for passengers through a GPS on cell phones, offering strong essential data support to the order system.

Order System

The ordering system, reviewed as the essential function system in KVCOOGO, has all the orders-related functions. On the whole, the order system has three parts of appointment service, order cancellation, and order inquiry.

Appointment service equips passengers with the benefit of car appointments. Passengers can make a car appointment after setting the departure place and destination. After the driver accepts the order, the user will receive a prepayment request. After the passenger pays, the driver will receive a system notification signaling that the passenger has paid. At this point, the driver will leave to pick up the passenger. Afterward, the order status is updated to trip in progress. After arriving at the destination, the driver chooses to end the order, at which the payment system will clear the order.

Cancellation of orders gives passengers a cancellation service. In car appointments, there are inevitably various circumstances that cause passengers to cancel the appointment. We classify cancellations into two periods: within 1 min of the passenger's payment and 1 min after the payment. Before payment, the driver will not pick up the passenger so that no dispute will arise, and the cancellation is blameless and will not result in liquidated damages. If the order is canceled after payment, when the driver is on the way to pick up the passenger, it will bring loss for the driver. Considering the passenger's rights, we give the passenger a grace period of 1 min. If the passenger cancels the order within 1 min, no penalty will incur. If made after 1 min, there will be a penalty. If the cancellation is unreasonable, the driver will also incur a penalty and compensate the passenger in return.

Orders inquiry provides passengers with the service of querying their chronological orders. Users can check all their historical orders, including completed and unfinished ones, each of which will be listed with detailed information, including departure, destination, time, price, driver, bill, etc.

Payment System

The payment system, the second essential function of KVCOOGO, controls all payment-related functions. The system can be divided into several modules, including exchange, cash withdrawal, prepayment, and order settlement.

The platform uses CarCoin, a “digital cash” issued for the KVCOOGO platform, for payment-related operations.

- The exchange module plays the role of exchanging CarCoin. Users can exchange RMB with CarCoin through the third-party payment platform.
- The cash withdrawal module provides the function of CarCoin withdrawal. Users can exchange Carcoins for RMB through third-party payment platforms.
- The prepayment module provides the payment function. At the early stage of the design, drivers' rights and interests will be considered, and payment in advance will be adopted; passengers will pay the order fee before the order starts. When a

passenger initiates a request, and a driver picks it up, the passenger receives a request to pay the payment in advance. The system will calculate the reasonable price according to the order information.

- The order settlement module provides a fee settlement function. Usually, passengers pay more in advance than the actual fare incurred. Therefore, a settlement is required at the end of the order. Specifically, the fare calculated is paid to the driver based on the actual trip, and then the excess fare will be returned to the passenger.

Evaluation System

The evaluation system is particularly critical because of the decentralization and no human intervention of KVCOOGO. It will dynamically adjust the price and distribution of orders according to the ratings of passengers and drivers.

Driver Function Analysis

In general, the functions of drivers are divided into five systems: identity, positioning, order, payment, and evaluation. Some functions overlap with the passenger end; therefore, they will not be described here.

Identity System

The driver's ID system is similar to that of the passenger. The identity system provides drivers with identity management-related functions, including registration, login, personal information management, etc. Users register to be a driver through the App. When drivers register, the system generates a public and private key pair for the driver, which acts as the driver's identity on the blockchain. In addition, the system generates the passenger's blockchain addresses based on the driver's public key. The system will allow the user to enter the mobile phone number as the user name during registration. Afterward, the user can log in to the system.

Personal information includes the driver's name, cell phone number, avatar, car model, and license plate number. Drivers can improve and modify their personal information after logging in.

Positioning System

KVCOOGO provides positioning service to users through GPS on cell phones, offering strong basic data support to the order system.

Order System

The driver's order system is divided into start order, stop order, start a trip, and end a trip.

- The start order allows drivers to “go to work.” As soon as the driver selects it, he will enter the service status. At this point, the order system will send the driver a selection of suitable orders.
- Once the driver picks one, the system will notify the passenger to pay in advance. The stop order allows drivers to “go offline.” After the driver picks it, it will go offline. The ordering system will not push orders to them again.
- The start trip allows the driver to “get to work.” When the driver picks up a passenger, he starts the journey when the order system records his location, tracking, time, and other trip information.
- Ending a trip allows the driver to “end the job.” When the trip is over, the driver chooses to end the trip. The ordering system will then perform the settlement process.

Payment System

The payment system on the driver's side includes fee settlement and cash withdrawal.

- Order Settlement provides the driver with a cost settlement function.
- The system will get the actual fare from the order system to pay the driver. Withdrawal offers drivers a withdrawal service. Drivers can withdraw the CarCoin they received as RMB in cash through the third-party payment platform.

Evaluation System

The driver-side rating system provides drivers with the function of rating passengers. The ratings given to passengers are equally critical in KVCOOGO. Drivers rate passengers reasonably according to their performance during the trip.

Function Analysis of Third-Party Payment Platform

The “currency” circulating on the KVCOOGO platform is CarCoin, but there is no exchange between RMB and CarCoin. To connect CarCoin with the real world, KVCOOGO needs to introduce a third-party payment platform.

The third-party payment platform offers a transaction platform for KVCOOGO and its users while users conduct transactions with KVCOOGO's official account on the third-party payment platform. The third-party payment has the following two functions:

- Exchange CarCoin. Users can purchase CarCoin by trading with KVCOOGO's official account through the account on the third-party payment platform.
- Withdrawal. Users can trade CarCoin with KVCOOGO's official account through the account on the third-party payment platform.

10.2.3 General System Design

The KVCOOGO system is a blockchain-based and decentralized online car-hailing platform designed to conform to the fundamental characteristic of decentralization. This section will introduce the KVCOOGO system in business logic design and system architecture design.

Business Logic Design

The business logic mainly consists of two parts: itinerary and payment.

Itinerary Business Logic

In this logic, there are four states of passengers: idle, scheduled, waiting for pickup, or on the trip; the driver can be in four states: offline, online, receiving, and on the trip. The itinerary business logic is designed as follows.

1. After entering the basic information, such as departure and destination, the passenger initiates a ride request. At this time, the passenger's status changes from idle to appointment.
2. According to the order aggregation logic, the system sends the order to the appropriate online driver. The driver selects an order.
3. The passenger receives the prepayment request and pays the fee in advance. After successful payment, the status changes from car appointment to waiting for pickup, and the status of the driver changes to picking up the passenger.
4. After picking up the passenger, the driver will confirm. The status of both the passenger and the driver changes to on the trip.
5. After arriving, the driver clicks to end the order. At this point, the passenger's status changes to the initial idle status and the driver's turns to the online status. A complete order is finished.

Payment Business Logic

The payment business logic is divided into payment in advance and billing settlement.

1. *Prepayment.* First, the passenger receives a payment request from the system. Once the passenger clicks on the payment, RMB will be paid to the platform account of KVCOOGO on the third-party payment platform. Once the platform account receives the RMB, it will top up the equivalent amount of CarCoin to the passenger's blockchain account. The passenger then uses CarCoin to pay for the prepayment. If the CarCoin balance is sufficient, the passenger can pay directly with CarCoin.
2. *Billing.* Billing is relatively simple. When the trip is over, the payment contract settles the fare on the actual cost of the order and the prepayment by paying the actual fare to the driver, and returning the excess prepayment to the passenger.

System Architecture Design

The figure shows that KVCOOGO's system architecture consists of four layers.

Business Layer

The business layer handles the business logic, being realized in App. In the business layer, passengers and drivers' identity systems, positioning systems, order systems, and payment systems are all implemented, and users can use these functions through the App.

Payment Layer

The payment layer with payment-related interfaces consists of two parts: CarCoin payment-related interfaces, including payments, etc.; the other is third-party payment-related interfaces, including CarCoin exchange, cash withdrawals, etc.

Interface Layer

The interface layer provides the interface to interact with the underlying blockchain by the JSON-RPC method.

Blockchain Layer

The blockchain layer has the blockchain platform, in which two smart contracts from the KVCOOGO project, order contract and payment contract, are deployed.

10.2.4 Smart Contract Design

KVCOOGO system adopts smart contracts, the main body of both program logic and data storage, to realize the main business logic. This section will introduce the design of smart contract in KVCOOGO travel system.

Conceptual Design

There are mainly two options for smart contract design. The first scheme adopts an object-oriented approach, where a smart contract is designed correspondingly for each object in the project. In this case, both the driver and passenger are contracts, and then a third contract (e.g., a taxi contract) is used to associate them together. Another option is to design a contract where the different objects are represented as structures and stored in a mapping. In this case, we use the second option, in which it is relatively easier to interact between objects, easier to understand the code, and simpler to test.

In general term, there is a need to design two contracts—a payment contract and a taxi contract. The former realizes the function of “electronic cash” CarCoin, while the latter recognizes the data storage function needed for taxi-hailing. Two are designed for they are less-coupled, and taxi contract may face frequent upgrades. In contrast, such design can reduce the maintenance overhead of extensions due to the relatively stable payment contract.

Payment Contract Design

Let us first introduce the contract state design.

The payment contract function is intended to enable “electronic currency” and requires a mapping to keep the user’s balance. Simultaneously, we need to retain the owner of the contract and the address of the taxi contract. The former works for identifying the issuer of the currency and controlling permissions for top-ups and withdrawals, equivalent to the role of an administrator. The latter controls access to transfers so that only the taxi contract can invoke methods involving transfers such as prepayment and checkout.

In reality, the smallest unit of currency is not the typical unit; for example, the smallest unit of RMB is “cent,” while the typical unit is “yuan.” To correspond to the RMB, we set the minimum unit of CarCoin to be also cents. Therefore, when using an int type to store it, 1 is 1 cent.

```
address owner; // contract owner, "currency" issuer  
address taxi; // taxi contract  
mapping (address => int) balances; // user balances
```

In addition, we hold the transfer record by a struct, with four attributes: payer, payee, value, and notes. The comments are used to indicate the transfer intent. There is a counter to keep track of the total number of these records, as our transfer records are stored in numbered order, which is done to save overhead. Finally, we need a mapping to a specific structure using the number.

```
struct record{
    address from; // payer
    address to; // payee
    int value; // value
    string comment; // intention
}
uint counter; // counter
mapping (uint => record) records; // Number mapping to structure
```

Next, we describe the contract method design in detail.

Constructors

Every contract has a default constructor, the same as in object-oriented programming (e.g., C++). It is called when the contract is deployed and is designed to complete some initialization work. We can override it. In our constructor, we want to initialize the owner field, which is used to record the contract owner, i.e., the currency issuer, set the total amount of currency issued afterward, and finally initialize our transfer record counter.

```
// constructor
function CarCoin() {
    owner = msg.sender;
    balances[owner] = 100000000000;
    counter = 1;
}
```

Permission Controllers

Solidity has a modifier identifier that can be declared and applied later in the method definition to reduce repetitive permission control statements in the code. The first one requires that the method caller must be the user address stored in the taxi field, and the second one requires that the method caller must be the owner of the contract stored in the owner field. We will see their specific usage later.

```
modifier onlyTaxi() {
    if (msg.sender != taxi) throw;
}
```

```
modifier onlyOwner() {
    if (msg.sender != owner) throw;
}
```

Registration Function

Our taxi field should change dynamically since the taxi contract may be redeployed to complete upgrades, maintenance, etc. This function requires the use of the identifier set for permission control so that only the contract owner can call it.

```
function exeOnce(address addr) onlyOwner{
    taxi = addr;
}
```

Record Function

According to the previous settings, any transfer action should have a corresponding record, and the following function will store the document.

```
// record function
function Transfer(address from, address to, int value, string comment)
private{
    records[counter].from = from;
    records[counter].to = to;
    records[counter].value = value;
    records[counter].comment = comment;
    counter++;
}
```

Transfer Function

Although we implement an “e-money,” we only allow it to circulate in the taxi process and do not allow users to transfer money freely. We subdivide the transfer functions into prepayment, checkout, default fee collection, and recharge/withdrawal according to the scenario. The first three are only allowed to be called by the taxi contract, and the last one by the contract owner. You can see that the identifier we defined earlier is employed in the function declaration for permission control. By the way, the same function works for recharge/withdrawal because we just set the amount to be negative to implement the withdrawal.

```
// Prepayment Function
function prepay(address client, int preFee) onlyTaxi
returns(bool success) {
```

```

balances[client] -= preFee;
balances[owner] += preFee;
Transfer(client, this, preFee, "prepay");
return true;
}
// Checkout function
function confirm(address client, address driver, int preFee,
    int finalFee) onlyTaxi returns(bool success){
    int remain = preFee - finalFee;
    balances[owner] -= preFee;
    balances[client] += remain;
    balances[driver] += finalFee;
    Transfer(this, client, remain, "remain fee");
    Transfer(this, driver, finalFee, "final fee");
    return true;
}
// Penalty function
function penalty(address from, address to, int amount)
    onlyTaxi returns(bool) {
    balances[from] -= amount;
    balances[to] += amount;
}
// Charge/withdrawal function
function recharge(address addr, int amount) onlyOwner
    returns(bool) {
    if (balances[owner] < amount) {
        return false;
    }
    balances[addr] += amount;
    balances[owner] -= amount;
    Transfer(this, addr, amount, "recharge");
    return true;
}

```

Query Function

We want to query the user's balance, and by the way, we can check who the contract owner really is.

```

// Query the balance
Function getBalance(address addr) returns(int) {
    return balances[addr];
}
// Query the contract owner
function getOwner() returns(address) {
    return owner;
}

```

Taxi Contract Design

The main objects of a taxi contract to have a complete taxi process include order, driver, passenger, and evaluation. The passenger is not abstracted separately, because, in the taxi scene, we do not require specific passenger information, but simply the passenger's contact and address, included in the order object, and the passenger's location is listed as a separate structure, which is just to make some logic more reasonable in the taxi scene, not necessary.

When it comes to data structure types, Solidity does not support floating-point calculations, so it has to adopt integers to simulate. This is the case with the CarCoin setup. Here we have to deal with the distance data, where the latitude and longitude are kept to six decimal places, i.e., 120° is stored in the primary form of 120,000,000, where 1 means 0.1 m based on the rough calculation. As for the time, we use the Linux timestamp, whose value is the number of seconds from January 1, 1970, to the current date. The rating uses the value [0, 5000], indicating a 5-point scale, i.e., it is possible to keep three decimal places.

We first list the order object, which contains the following attributes: order number, passenger address, driver address, origin and destination latitude and longitude, origin and destination place name, distance, time, cost, status, passenger information, and driver information. This information, as it literally means, is very well understood. A counter indicates the current number of orders and mapping to a specific order structure.

```
struct Order{
    uint id; // order number
    address passenger; // passenger's address
    address driver; // driver's address
    int s_x; // Longitude of starting point
    int s_y; // Latitude of starting point
    int d_x; // Longitude of the end point
    int d_y; // Latitude of the end point
    string sName; // name of starting point
    string dName; // name of end point
    int distance; // Straight line distance from starting point to the end
    point
    int preFee; // Prepayment amount
    int actFee; // Mileage fee
    int actFeeTime; // Hourly fee
    uint startTime; // Order submission time UNIX standard time
    uint pickTime; // Pickup time
    uint endTime; // End time
    int state; // Order Status 1To be assigned 2Order taken 3Order
    completed 4Order terminated
    string passInfo; // passenger information
    string drivInfo0; // driver information
    string drivInfo1;
    string drivInfo2;
}
```

```
uint counterOrderIndex; // Next empty order number
mapping (uint => Order) orders;
```

The driver structure holds information about the driver with the following attributes: latitude, longitude, whether to take orders, own address, data, order pool, last latitude, and longitude. It is important to note that the “last latitude and longitude” attribute, for real-time billing, is a secondary attribute. In addition, since orders do not necessarily have to correspond one-to-one to a particular driver or passenger (when a specific order ends, a new one begins), only the number is required as a mapping critical value. On the other hand, the driver structure corresponds one-by-one to a specific driver and still requires a number within the drivers, so we see a double mapping: the driver’s address is mapped to a number, which in turn is mapped to a specific structure.

```
mapping (address => uint) driverIndexes; // assign an internal serial
number to each driver
uint counterDriverIndex; // Next empty driver serial number (current
number of drivers + 1)
struct Driver{
    int cor_x; // longitude
    int cor_y; // latitude
    bool state; // true means taking orders; false means being in rest
    address name; // driver's address
    string info0;
    string info1;
    string info2;
    uint counterOrder; // the current number of orders available to the
driver
    uint [8] orderPool; // Driver's available order pool
    int last_x; // Last longitude
    int last_y; // Last latitude
}
mapping (uint => Driver) drivers; // Use the serial number to find the
driver's information
```

The evaluation structure is fairly simple; it has the following attributes: total number of evaluations, average evaluation, a single score, and single evaluation. Among them, both single scores and evaluations are stored in order, and there is also a mapping to map the driver addresses to the evaluation structure.

```
struct Judgement{
    int total; // Total number of evaluations
    int avgScore; // Average score
    mapping (int => int) score; // Single score
    mapping (int => string) comment; // Single evaluation
}
mapping (address => Judgement) driverJudgements;
```

The passenger position structure is simple and will not be repeated.

```

struct passengerPosition{
    int x;
    int y;
}
mapping (address => passengerPosition) passPos;

```

Finally, there are some separate structures. The first one is the payment contract structure, a way for contracts to call each other, much like classes in C++, where calling a contract method is just a matter of using “variable. method name.” We also have two mappings that map the passenger and driver to an order, representing the current order in which the passenger and driver are in. As mentioned earlier, it will be updated as the old order ends and the new one begins. Two mappings follow this to store the current state of the user, a vital identifier indicating which stage the user is in, as we will detail in the contract method design. The last mapping indicates nearby drivers.

```

CarCoin carcoin;

// Each passenger corresponds to a certain order
mapping (address => uint) passengerToOrder;
// Each driver corresponds to an order
mapping (address => uint) driverToOrder;
// Passenger states
mapping (address => uint) passengerStates;
// Driver states
mapping (address => uint) driverStates;
// Nearby drivers
mapping (address => uint [5]) passengerNearDrivers;

```

Let us take a closer look at contract method design.

Constructor

Unlike the default constructor, we make use of a parameterized constructor that takes the address of the payment contract. This is because the method of payment contract needs to be applied many times in the taxi-hailing process. Other than that, it initializes both the order counter and the driver counter.

```

function Taxi (address cc) {
    counterDriverIndex = 1;
    counterOrderIndex = 1;
    carcoin = CarCoin(cc);
}

```

Private Functions

As in C++, we define some private functions that can only be used internally. Here we have four private functions. The first is the rooting function. Solidity does not have a ready-made function for rooting integers; we define it ourselves. There are many discussions about integer rooting on the Internet, and they can obtain integer square roots with high accuracy. Secondly, the distance calculation function uses the two-point distance formula, which is deviated in this case because there is a set of formulas to calculate the actual distance on the latitude and longitude between two points, taking into account the radius of the earth as well as the latitude and longitude position, etc. We have simplified the spherical distance formula into a plane distance formula because Solidity is unsuitable for complex mathematical operations. In addition, a small range of distance error does not impact. That is why we apply this strategy. The third is the order assignment function, which employs the most straightforward scheme, i.e., assigning drivers within a specific range. The last is the calculation of the advance function, established by calculating the distance between the right-angled sides of two points and multiplying it by a factor.

```

function sqrt(int x) private returns (int) {
    if(x < 0)
        x = -x;
    int z = (x + 1) / 2;
    int y = x;
    while (z < y) {
        y = z;
        z = (x / z + z) / 2;
    }
    return y;
}

function calculateDistance(int x0, int x1, int y0, int y1)
    private returns(int) {
    int tempX = x0 - x1;
    int tempY = y0 - y1;
    return sqrt(tempX*tempX + tempY*tempY);
}

function driverSelction(int x, int y, uint orderIndex)
    private returns(bool) {
    uint i;
    uint j;
    int threshold = 50000; // Threshold, when the distance is less than the
    value after the order is dispatched, the value can be adjusted
    int temp;
    uint maxOrder = 8; // The maximum number of orders a driver can grab
    bool flag = false;
    for (i=1; i<counterDriverIndex; ++i) {
if (drivers[i].state && driverStates[drivers[i].name] == 0) {
    temp = calculateDistance(x, drivers[i].cor_x, y,
        drivers[i].cor_y);
}
}
}

```

```

        if (temp < threshold) {
            // Find an empty spot in the order pool
            for(j=0; j<maxOrder; ++j) {
                if(orders[drivers[i].orderPool[j]].state != 1) {
                    flag = true;
                    drivers[i].orderPool[j] = orderIndex;
                    break;
                }
            }
        }
    }
    return flag;
}

function calculatePreFee(int s_x, int s_y, int d_x, int d_y)
    private returns(int) {
    int tempX = s_x - d_x;
    int tempY = s_y - d_y;
    if (tempX < 0) {
        tempX = -tempX;
    }
    if (tempY < 0) {
        tempY = -tempY;
    }
    return ((tempX + tempY) * unitPrice) / 2 * 3 / 100;
}
}

```

Passenger Submits an Order

As it literally says, the passenger submits an order request through this function, and the contract assigns a new order number, writes data, and assigns it to the driver. If the passenger has an insufficient balance, if no drivers, or if incorrect status (for example, if the previous order is not finished, a new order will come in), the submission will fail. After submission, the order number will be returned, by which the passenger can quickly check the order specifics. If the request is successful, the corresponding status will be set up, and the order will enter the stage of waiting for the driver to grab the order.

```

function passengerSubmitOrder(int s_x, int s_y, int d_x, int
    d_y, uint time, string passInfo, string sName, string dName) returns
(uint) {

    // Passenger account balance must be positive
    if(carcoin.getBalance(msg.sender) < 0) {
        return 0;
    }
}

```

```

// Passengers must be on hold to grab an order
if (passengerStates[msg.sender] != 0) {
    return 0;
}
if (counterDriverIndex <= 1) { // No divers
    return 0;
}
// Create a new order
passengerToOrder[msg.sender] = counterOrderIndex;
orders[counterOrderIndex].id = counterOrderIndex;
orders[counterOrderIndex].passenger = msg.sender;
orders[counterOrderIndex].driver = 0x0;
orders[counterOrderIndex].s_x = s_x;
orders[counterOrderIndex].s_y = s_y;
orders[counterOrderIndex].d_x = d_x;
orders[counterOrderIndex].d_y = d_y;
orders[counterOrderIndex].distance = 0;
orders[counterOrderIndex].preFee = penaltyPrice + calculatePreFee
(s_x, s_y, d_x, d_y);
orders[counterOrderIndex].actFee = 0;
orders[counterOrderIndex].actFeeTime = 0;
orders[counterOrderIndex].startTime = time;
orders[counterOrderIndex].state = 1;
orders[counterOrderIndex].passInfo = passInfo;
orders[counterOrderIndex].sName = sName;
orders[counterOrderIndex].dName = dName;
counterOrderIndex++;
passengerStates[msg.sender] = 1; // Passenger orders are being
assigned

if (!driverSelction(s_x, s_y, counterOrderIndex-1)) {
    orders[counterOrderIndex-1].state = 4;
    passengerStates[msg.sender] = 0;
    return 0;
}

return counterOrderIndex-1;
}

```

Driver Order-Grabbing

We take the driver to grab order mode, by which this function can complete this operation; it will check the conditions of grabbing an order: if successful, it will set the corresponding status, and enter the stage of waiting for the passenger to pay in advance.

```

function driverCompetOrder(uint orderIndex) returns(bool) {
    if (driverIndexes[msg.sender] == 0) { // Driver not registered
        return false;
    }
}

```

```

if(driverStates[msg.sender] != 0) { // Driver is not available, pending
    status)
    return false;
}
if(orders[orderIndex].state != 1) { // Failed to grab the order
    return false;
}
orders[orderIndex].state = 2;
orders[orderIndex].driver = msg.sender;
orders[orderIndex].drivInfo0 =
    drivers[driverIndexes[msg.sender]].info0;
orders[orderIndex].drivInfo1 =
    drivers[driverIndexes[msg.sender]].info1;
orders[orderIndex].drivInfo2 =
    drivers[driverIndexes[msg.sender]].info2;
passengerStates[orders[orderIndex].passenger] = 2; // Passenger
pending payment
driverStates[msg.sender] = 1; // Driver has taken the order
driverToOrder[msg.sender] = orderIndex;
// Initialize the driver's last position
drivers[driverIndexes[msg.sender]].last_x =
    orders[orderIndex].s_x;
drivers[driverIndexes[msg.sender]].last_y =
    orders[orderIndex].s_y;
return true;
}

```

Prepayment from Passengers

We adopt the advance payment model, where the advance payment is to invoke the interface of the payment contract and then check the status. If successful, the corresponding status will be set, and the driver will wait for picking up passengers.

```

function passengerPrepayFee() returns (bool) {
    uint orderIndex = passengerToOrder[msg.sender];
    address driver = orders[orderIndex].driver;

    // Passengers not waiting for payment or orders not already taken
    if (passengerStates[msg.sender] != 2 || 
        orders[orderIndex].state != 2) {
        return false;
    }

    // Payment process, make sure the money is in the contract account
    if (carcoin.prepay(msg.sender, orders[orderIndex].preFee)) {
        passengerStates[msg.sender] = 3;
        driverStates[driver] = 2;
        return true;
    } else {
        // ...
    }
}

```

```

orders [orderIndex] .state = 4;
passengerStates [msg.sender] = 0;
driverStates [driver] = 0;
return false;
}
}
}

```

Driver Picks Up Passenger

When the driver picks up a passenger, the function is called to perform a status check and setup. Here we perform an anti-cheat check, i.e., it can only be called successfully when the current position of the driver and passenger are close enough to prevent the driver from committing fraud. Such automated judgments are required because this is an entirely unattended system and the stringent checks help reduce disputes. With a successful call, it goes to the state of the trip.

```

function driverPickUpPassenger(int x, int y, uint time)
returns(bool) {
uint orderIndex = driverToOrder [msg.sender];
address passenger = orders [orderIndex] .passenger;

// Status check
if (driverStates [msg.sender] != 2 || 
passengerStates [passenger] != 3 || 
orders [orderIndex] .state != 2) {
    return false;
}

int passX = passPos [passenger] .x;
int passY = passPos [passenger] .y;
int threshold = 20000;

if (calculateDistance(x, passX, y, passY) > threshold) {
    return false;
}

drivers [driverIndexes [msg.sender]] .last_x = x;
drivers [driverIndexes [msg.sender]] .last_y = y;
orders [orderIndex] .pickTime = time;

passengerStates [passenger] = 4;
driverStates [msg.sender] = 3;
return true;
}

```

Real-Time Billing

The method to realize real-time billing is that the driver continuously calls this function during the journey and passes in the current position to perform segmented billing. When segmented segments are fine enough, we can approximate the total length of the trip with the sum of the straight-line distances of each piece thus achieving the effect of real-time billing.

```
function driverCalculateActFee(int cur_x, int cur_y) returns(int) {
    uint orderIndex = driverToOrder[msg.sender];
    uint driverindex = driverIndexes[msg.sender];
    int distance;
    address passenger = orders[orderIndex].passenger;

    // Status check
    if (driverStates[msg.sender] != 3 || 
        passengerStates[passenger] != 4 || 
        orders[orderIndex].state != 2) {
        return 0;
    }

    distance = calculateDistance(cur_x,
        drivers[driverindex].last_x, cur_y,
        drivers[driverindex].last_y);
    orders[orderIndex].distance += distance;
    orders[orderIndex].actFee += distance * unitPrice / 100;
    drivers[driverindex].cor_x = cur_x;
    drivers[driverindex].cor_y = cur_y;
    drivers[driverindex].last_x = cur_x;
    drivers[driverindex].last_y = cur_y;
    return orders[orderIndex].actFee;
}
```

Complete the Order

The driver can use this function to complete the order after the passenger gets off the car. The system will settle the order based on the actual cost and prepayment amount, and call the payment contract interface to complete the payment.

```
function driverFinishOrder(uint time) returns(bool) {
    uint orderIndex = driverToOrder[msg.sender];
    address passenger = orders[orderIndex].passenger;
    // The driver not on a trip, the order not already taken
    if (driverStates[msg.sender] != 3 || 
        passengerStates[passenger] != 4 || 
        orders[orderIndex].state != 2) {
        return false;
    }
    if (time < orders[orderIndex].pickTime) {
```

```

        time = orders [orderIndex] .pickTime;
    }
    orders [orderIndex] .actFeeTime = (int) (time -
        orders [orderIndex] .pickTime) * unitPriceTime;
    int preFee = orders [orderIndex] .preFee;
    int finalFee = orders [orderIndex] .actFee +
        orders [orderIndex] .actFeeTime;
    if (finalFee > preFee) {
        finalFee = preFee;
    }
    orders [orderIndex] .actFee = finalFee -
        orders [orderIndex] .actFeeTime;
}
}

// Payment
if (carcoin.confirm(passenger, msg.sender, preFee,
    finalFee)) {
    orders [orderIndex] .state = 3;
    orders [orderIndex] .endTime = time;
    passengerStates [passenger] = 0;
    driverStates [msg.sender] = 0;
    return true;
} else {
    // ...
    passengerStates [passenger] = 0;
    driverStates [msg.sender] = 0;
    orders [orderIndex] .state = 4;
    return false;
}
}
}

```

Cancellation Function

Order cancellation often happens in practical situations. We provide a unified interface, and the system will automatically determine whether the current state can be canceled and will set accordingly. The passenger calls the cancellation function corresponding to the passenger, and the driver calls the cancellation function corresponding to the driver.

```

function passengerCancelOrder (bool isPenalty) returns (bool) {
    uint orderIndex = passengerToOrder [msg.sender];
    address driver = orders [orderIndex] .driver;

    // Passengers cancel orders before the driver takes them, no penalty
    if (passengerStates [msg.sender] == 1 &&
        orders [orderIndex] .state == 1) {
        passengerStates [msg.sender] = 0;
        orders [orderIndex] .state = 4;
        return true;
    }
}

```

```
// Passengers cancel orders after the driver takes them and before they
// prepay themselves, no penalty
if (passengerStates[msg.sender] == 2 &&
    driverStates[driver] == 1 && orders[orderIndex].state
    == 2) {
    passengerStates[msg.sender] = 0;
    driverStates[driver] = 0;
    orders[orderIndex].state = 4;
    return true;
}

// Cancellation of orders by passengers after prepayment and while
// waiting for the driver to pick up
if (passengerStates[msg.sender] == 3 &&
    driverStates[driver] == 2 && orders[orderIndex].state
    == 2) {
    // Refund of advance payment
    if (!carcoin.confirm(msg.sender, driver,
        orders[orderIndex].preFee, 0)) {
        return false;
    }
    // Penalty
    if (isPenalty) {
        carcoin.penalty(msg.sender, driver, penaltyPrice);
    }
    passengerStates[msg.sender] = 0;
    driverStates[driver] = 0;
    orders[orderIndex].state = 4;
    return true;
}

return false;
}

function driverCancelOrder() returns(bool) {
    uint orderIndex = driverToOrder[msg.sender];
    address passenger = orders[orderIndex].passenger;

    // Driver cancels before prepayment
    if (driverStates[msg.sender] == 1 &&
        passengerStates[passenger] == 2 &&
        orders[orderIndex].state == 2) {
        passengerStates[passenger] = 0;
        driverStates[msg.sender] = 0;
        orders[orderIndex].state = 4;
        return true;
    }
    // Driver cancels after the prepayment, with penalty for breach of
    // contract
    if (driverStates[msg.sender] == 2 &&
        passengerStates[passenger] == 3 &&
        orders[orderIndex].state == 2) {
        // Refunds of prepayment
```

```

if (!carcoin.confirm(passenger, msg.sender,
    orders[orderIndex].preFee, 0)) {
    return false;
}
carcoin.penalty(msg.sender, passenger, penaltyPrice);
passengerStates[passenger] = 0;
driverStates[msg.sender] = 0;
orders[orderIndex].state = 4;
return true;
}
return false;
}

```

Passenger Evaluation

Passengers can evaluate after completing an order. After the evaluation, the binding will be released and passengers will not be able to evaluate again.

```

function passengerJudge(int score, string comment)
returns(bool) {
uint orderIndex = passengerToOrder[msg.sender];
address driver = orders[orderIndex].driver;
int total = driverJudgements[driver].total;

if (orderIndex == 0) {
    return false;
}

passengerToOrder[msg.sender] = 0; // Unbinding
if (score > 5000)
    score = 5000;
if (score < 0)
    score = 0;
driverJudgements[driver].avgScore =
    (driverJudgements[driver].avgScore * total + score) / (total + 1);
driverJudgements[driver].total += 1;
total++;
driverJudgements[driver].score[total] = score;
driverJudgements[driver].comment[total] = comment;
return true;
}

```

Driver Registration

The driver has a number inside the contract, and a new driver is not directly assigned a driver structure, this function thus requires being called to “register.”

```

function newDriverRegister(string info0, string info1, string
info2) returns(uint) {
if (driverIndexs[msg.sender] > 0){// already registered
    return driverIndexs[msg.sender];
}
driverIndexs[msg.sender] = counterDriverIndex;
drivers[counterDriverIndex].state = false;
drivers[counterDriverIndex].name = msg.sender;
drivers[counterDriverIndex].cor_x = 0;
drivers[counterDriverIndex].cor_y = 0;
drivers[counterDriverIndex].info0 = info0;
drivers[counterDriverIndex].info1 = info1;
drivers[counterDriverIndex].info2 = info2;
drivers[counterDriverIndex].counterOrder = 0;
driverStates[msg.sender] = 0;
counterDriverIndex++;
return counterDriverIndex - 1;
}

```

Query Function

Due to the specific nature of smart contracts, Solidity does not support active notifications, all information needs to be actively queried by the client. A large number of query functions are required to keep the query. The structure of these functions is very similar in that they all return the concrete contents of some form. The following is only to query the order information as an example; as the details will not be described, readers can refer to the contract source code.

```

function getOrderInfo0(uint orderIndex) returns(uint id,
address passenger, int s_x, int s_y, int d_x, int d_y,
int distance, int preFee, uint startTime, string
passInfo) {
id = orders[orderIndex].id;
passenger = orders[orderIndex].passenger;
s_x = orders[orderIndex].s_x;
s_y = orders[orderIndex].s_y;
d_x = orders[orderIndex].d_x;
d_y = orders[orderIndex].d_y;
distance = orders[orderIndex].distance;
preFee = orders[orderIndex].preFee;
startTime = orders[orderIndex].startTime;
passInfo = orders[orderIndex].passInfo;
}

```

10.2.5 System Implementation and Deployment

The system has the front-end of the app and the back-end of the blockchain; the former can be directly downloaded and installed; thus, we will not go into details here. This section will introduce the deployment of the blockchain back-end of the KVCOOGO system.

System Deployment Diagram

The system deployment diagram is shown in Fig. 10.7.

This case employs Hyperchain as the underlying platform of blockchain, and KVCOOGO, a third-party payment platform, and other partners each provide a server to join the Hyperchain network as a blockchain node. Users (passengers and drivers) access the Hyperchain platform through the App. The payment server of the third-party payment platform connects to the Hyperchain as the system's payment gateway and the App connects to the payment gateway through the third-party payment API.

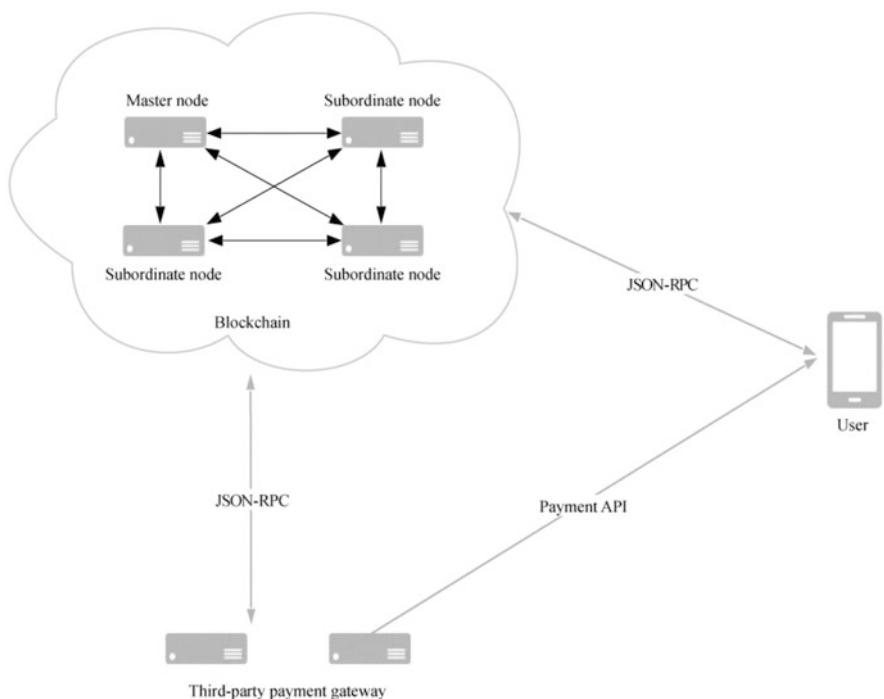


Fig. 10.7 The system deployment diagram

System Deployment Environment

Hardware Environment

Blockchain: Ali cloud server, dual-core CPU, 4GB RAM, 500GB available storage space.

App: iPhone.

Third-party payment gateway: Ali cloud server, dual-core CPU, 4GB RAM, 200GB available storage space.

Software Environment

Blockchain: Linux operating system (Ubuntu 14.04); Go language environment.

App: iOS 8.0 or above.

Third-party payment gateway: Linux operating system (Ubuntu 14.04); Go language environment.

Blockchain Environment Construction

For an empty server, you can follow the following steps to build the blockchain environment.

Install Git

```
apt-get update  
apt-get install git
```

Install Curl

```
apt-get install curl
```

Install and Configure Go Language Environment

```
curl -o https://storage.googleapis.com/golang/go1.8.1.linux-amd64.  
tar.gz  
tar -C /usr/local -xzf go1.8.1.linux-amd64.tar.gz  
export PATH=$PATH:/usr/local/go/bin
```

Get the Blockchain Executable Binary

```
git clone https://github.com/trakel-project/trakelchain.git
```

Start the Blockchain

```
cd trakelchain && ./start.sh
```

Note that the above environment refers to the building steps of the blockchain environment, excluding the third-party payment gateway and iOS App. trakelchain has already deployed the smart contracts required by KVCOOGO, which can be called directly.

10.3 Summary

This chapter introduces two enterprise blockchain application project cases based on Hyperchain. Each includes project introduction, system function analysis, overall system design, intelligent contract design, implementation, deployment, etc. It can be seen that Hyperchain can work for building blockchain applications with complete functions, leading technologies, and meeting enterprise-level requirements. Readers can compare this chapter through the comprehensive development interface provided by Hyperchain for in-depth study and practice of blockchain application development.