

本 科 生 毕 业 设 计



**题目 区块链运维和管理平台的设计与实现**

**姓名 金鹏**

**学号 3140102430**

**指导教师 杨小虎**

**年级与专业 2014级 软件工程**

**学院 计算机科学与技术学院**

**提交日期 2018年5月19日**

A Thesis Submitted to Zhejiang University

for the Degree of Bachelor of Engineering



TITLE Design and Implementation of

Block Chain Operation and

Management Platform

Author Jin Peng

Student ID 3140102430

Supervisor Yang Xiaohu

Major Software Engineering

College College of Computer Science and Technology

Submitted Date May 19, 2018

浙江大学本科生毕业论文（设计）独创性声明

本人声明所呈交的毕业论文（设计）是本人在导师指导下进行的研究工作及取得的研究成果。除了文中特别加以标注和致谢的地方外，文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得 **浙江大学** 或其他教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在文中作了明确的说明并表示谢意。

作者签名： 日期： 年 月 日

毕业论文（设计）版权使用授权书

本文作者完全了解 **浙江大学** 有权保留并向国家有关部门或机构送交本文的复印件和磁盘，允许本文被查阅和借阅。本人授权 **浙江大学** 可以将毕业论文（设计）的全部或部分内容编入有关数据库进行检索和传播，可以采用影印、缩印或扫描等复制手段保存、汇编毕业论文（设计）。

（保密的毕业论文（设计）在解密后适用本授权书）

作者签名： 导师签名：

日期： 年 月 日 日期： 年 月 日

摘要

随着区块链技术的发展，国内外很多金融或非金融机构都在尝试将业务运行在区块链上。节点的运行状况会影响上层业务，因此这些机构在使用区块链时需要保证节点的正常运行，同时需要获取区块链上的交易量变化情况等指标。此外，作为区块链与上层应用之间的媒介，智能合约的管理也是这些机构所需要的。

因此，本文设计并实现了区块链的运维和管理平台。本文首先调研了使用区块链平台用户的需求，确定了该平台所要实现的功能，进而设计了整体技术架构并进行了实现。在实现过程中，本文选用了高可用的前后端框架，并设计了合适的技术方案，保证系统展示的数据真实可靠。并且本文最后通过功能性测试证明系统的可用性和有效性。系统已经在生产环境中投入使用，用户反馈良好，也进一步证明了该系统的价值。

关键词 区块链，监控，智能合约，报警

Abstract

Nowadays the technology of block chain is rapidly developing, on which many financial and non-financial institutions at home and abroad are trying to run the business. However, the operating status of the nodes will affect the upper-layer services. Thus it is indispensable to ensure the stability of the nodes while using the block chain and understand the indicators includes changes in transaction volume on the block chain. In addition, as the medium between the block chain and the upper application, the management of smart contracts is also needed by these institutions.

Therefore, this paper designs and implements the operation and management of the block chain. This paper firstly investigates the needs of the users using the block chain platform, and determines the functions of the platform. Then it designs and implements the overall technical architecture. In the implementation process, this paper selects a highly available front-end framework and designs a suitable technical solution to ensure that the data displayed by the system is reliable. Finally, this paper proves the availability and effectiveness of the system through functional tests. This system has been put into use in the production environment, users offered favourable feedback which proves the value of the system.

**Keywords** **block chain, monitoring, smart contract, alarm**

目录

[摘要 I](#_Toc514446646)

[Abstract II](#_Toc514446647)

[第1章 绪论 1](#_Toc514446648)

[1.1 课题背景 1](#_Toc514446649)

[1.2 本文研究目标和内容 1](#_Toc514446650)

[1.3 本文结构安排 2](#_Toc514446651)

[第2章 相关理论与技术 4](#_Toc514446652)

[2.1 区块链 4](#_Toc514446653)

[2.2 智能合约 4](#_Toc514446654)

[2.3 LevelDB 5](#_Toc514446655)

[2.4 Beego 5](#_Toc514446656)

[2.5 AngularJS 6](#_Toc514446657)

[第3章 系统分析与设计 8](#_Toc514446658)

[3.1 需求分析 8](#_Toc514446659)

[3.1.1 功能性需求 8](#_Toc514446660)

[3.1.2 非功能性需求 9](#_Toc514446661)

[3.2 可行性分析 9](#_Toc514446662)

[3.2.1 节点数据动态展示可行性 9](#_Toc514446663)

[3.2.2 节点异常报警可行性 9](#_Toc514446664)

[3.2.3 可视化智能合约管理可行性 10](#_Toc514446665)

[3.2.4 结论 10](#_Toc514446666)

[3.3 核心功能设计 10](#_Toc514446667)

[3.3.1 节点管理 10](#_Toc514446668)

[3.3.2 区块链监控 12](#_Toc514446669)

[3.3.3 监控报警 12](#_Toc514446670)

[3.3.4 智能合约管理 13](#_Toc514446671)

[3.4 数据库设计 14](#_Toc514446672)

[3.4.1 数据库选择 14](#_Toc514446673)

[3.4.2 存储结构设计 15](#_Toc514446674)

[第4章 系统实现 16](#_Toc514446675)

[4.1 系统架构 16](#_Toc514446676)

[4.1.1 整体技术架构 16](#_Toc514446677)

[4.1.2 部署架构 17](#_Toc514446678)

[4.1.3 API服务器架构 18](#_Toc514446679)

[4.2 模块实现 19](#_Toc514446680)

[4.2.1 区块链监控数据汇总 19](#_Toc514446681)

[4.2.2 报警 22](#_Toc514446682)

[4.2.3 智能合约管理 26](#_Toc514446683)

[4.3 系统关键算法实现 30](#_Toc514446684)

[4.3.1 监控数据采集算法 30](#_Toc514446685)

[4.3.2 交易数据快速同步算法 33](#_Toc514446686)

[第5章 系统测试与成果展示 35](#_Toc514446687)

[5.1 功能测试 35](#_Toc514446688)

[5.1.1 用户模块 35](#_Toc514446689)

[5.1.2 节点管理模块 35](#_Toc514446690)

[5.1.3 区块链监控模块 36](#_Toc514446691)

[5.1.4 智能合约模块 36](#_Toc514446692)

[5.2 系统成果展示 37](#_Toc514446693)

[5.2.1 用户管理 37](#_Toc514446694)

[5.2.2 节点管理 38](#_Toc514446695)

[5.2.3 监控管理 41](#_Toc514446696)

[5.2.4 智能合约管理模块 45](#_Toc514446697)

[第6章 总结与展望 52](#_Toc514446698)

[6.1 论文主要工作总结 52](#_Toc514446699)

[6.2 展望 53](#_Toc514446700)

[参考文献 54](#_Toc514446701)

[致谢 56](#_Toc514446702)

绪论

课题背景

作为近些年来最热门的技术之一，区块链技术发展迅速，各种区块链创业公司纷纷涌现，各大互联网巨头也纷纷入局。区块链从本质上讲是一个去中心化的分布式账本，其特有的共识机制让它有别于传统互联网的数据库技术。而其去中心化、不可篡改、可追溯等特性，使得其在金融、存证、供应链等领域都有很好的应用场景。

随着区块链的应用越来越广泛，无论是拥有自己开发的区块链的公司，还是使用其他区块链的公司，都需要对自己的节点进行监控和管理，尤其是在联盟链（Consortium Block Chain）场景中，其节点数量远不如共有链（Public Block Chain），因而每一个节点都非常重要，任何一个节点发生异常都应该迅速被运维人员知晓。此外，在以太坊（Ethereum）将智能合约（Smart Contract）的概念结合到区块链之后，智能合约成为了很多区块链应用的媒介，因此智能合约的管理对于企业的开发运维人员来说也很重要。

因此，基于上述原因，本文提出了区块链运维和管理平台，旨在解决目前区块链数据无法可视化监控，智能合约部署调用不方便等问题，保证区块链的正常运行。

本文研究目标和内容

本项目将依托于杭州趣链科技有限公司自主研发的区块链平台Hyperchain，以及平台拥有的Go语言的SDK——Hyperkit，后台实时获取区块链数据，并将数据封装成RestFul风格的API，前端实现对区块链的监控及管理。本项目希望通过该种方式，使得区块链的运维人员能够更直观地看到节点的运行情况以及区块链的使用状况，同时方便用户对智能合约进行部署和调用，降低区块链应用的开发时间成本，从而降低区块链的使用门槛，推动区块链技术的发展。

本文将设计与实现区块链运维和管理平台，该系统主要包括以下模块：

1) 节点管理

节点管理是其他功能模块的基础，可以通过配置和注册将正在运行的区块链节点添加到监控平台，并可以随时切换接入点，实现对不同区块链的监控和数据浏览以及智能合约的部署位置切换。

2) 数据浏览

在确定接入点后，系统通过周期性地向接入点查询交易数据，并对数据进行汇总，可视化地展现在用户面前。浏览的数据包括交易吞吐量、交易成功率、交易响应率等监控指标。

3) 监控报警

在创建分区时可以配置相关监控数据的报警阈值，在区块链运行过程中，若某个数据发生异常，超出阈值范围内，则向用户发送提醒消息。报警指标包括交易总量、交易变化率、交易吞吐量、交易成功率、平均交易响应时间、交易失败笔数、交易响应率等。

4) 智能合约管理

用户能够通过该模块快速编译、部署和调用智能合约，并且在完成部署后可以查看合约变量和合约函数的调用历史记录。

5) 用户管理

为保证系统的安全性和隐私性，用户在使用该系统前需要先行进行登录。但由于系统的用户类型比较单一，故当前系统仅提供一个系统账户供用户使用。

本文结构安排

第一章为绪论，阐述了课题产生的背景，描述了当前研究存在的不足和问题，并提出了本文研究的内容和研究方法。

第二章围绕区块链运维和管理平台所用到的技术，介绍了区块链、智能合约、LevelDB数据库、Beego后端开发框架和AngularJS前端框架。

第三章为系统的分析与设计，描述了系统的需求以及其可行性，并对其核心功能进行了较为详细的设计。

第四章讲解了系统的具体实现方案，描述了系统的整体架构以及三大功能模块的实现方法，并对项目中所使用的关键技术与算法进行了详细描述。

第五章说明了系统进行功能性测试的结果，得出系统可用的结论，并对系统界面和操作流程进行了展示。

第六章总结了本文工作并提出了展望。

相关理论与技术

区块链

区块链技术起源于加密货币。2008年，一个化名中本聪的人发表了《比特币：一种点对点电子现金系统》，掀开了区块链技术的序幕。区块链实质上是一个去中心化的分布式账本数据库，通过特定的共识机制，让区块链中的每一个节点都参与到交易的验证中，保证存储在区块链上的信息不可伪造和篡改。

当前应用比较广泛的数据库主要都还是中心化的，无论是银行、电商还是网络游戏，系统属于谁，数据库就由谁来维护，无法解决去信任和去中心化两大难题。而在区块链技术下，各个节点共同维护同一个数据库，采取特定的算法，如POW（Proof of Work，工作量证明），POS（Proof of Stake，权益证明），DPOS（Delegate Proof of Stake，股份授权证明机制）、PBFT（Practical Byzantine Fault Tolerance 实用拜占庭容错算法）等，则能够很好地解决这两个问题。

根据当前的技术结构以及应用场景，区块链大致被分为了三种类别，分别是公有链（Public Blockchain）、联盟链（Consortium Blockchain）和私有链。联盟链和私有链相对于公有链来说是部分去中心化的，其节点数量一般也远少于公有链。

智能合约

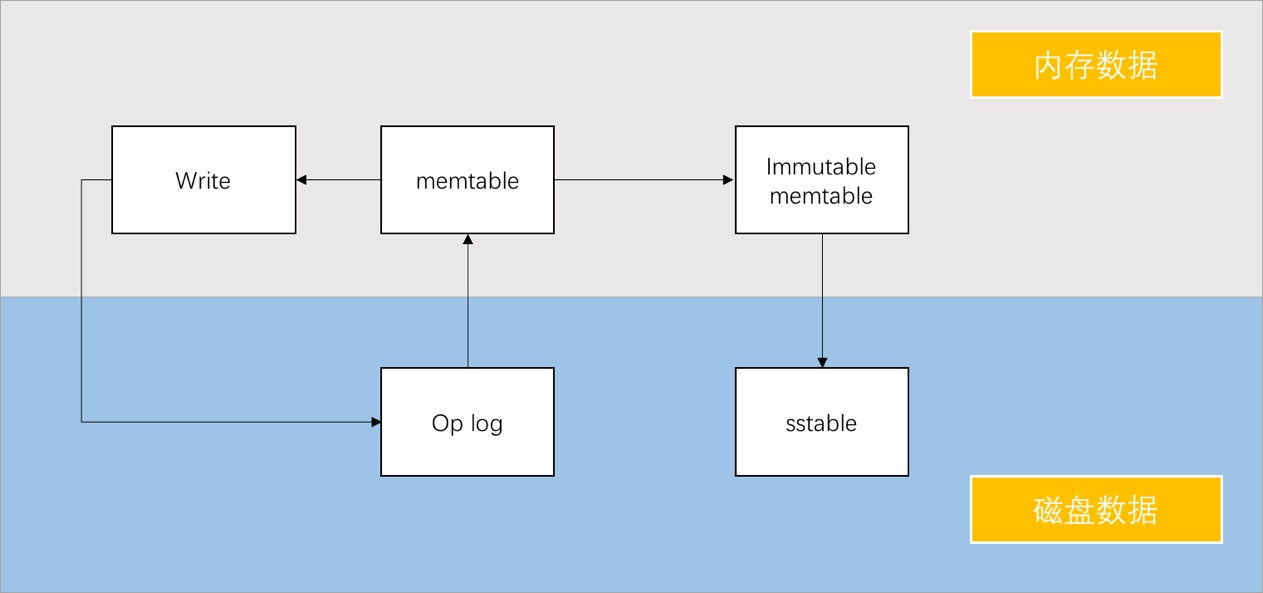
智能合约可以认为是区块链与具体应用之间的桥梁，如今各种基于区块链的应用都是在智能合约的基础上完成的，如代币系统、储蓄钱包、存证系统等。智能合约的存在给了区块链的应用无限的想象空间。智能合约的概念早在1994年就被提出，当时只是指一种计算机协议，用于保证交易在没有第三方的情况下是可信的。而以太坊率先将智能合约与区块链结合起来，为区块链在应用层上的使用提供了便捷。它开创性地纳入了一种图灵完备的语言——solidity，使得区块链变得可编程。在区块链不可篡改的特性下，智能合约可以定义一些高保险的操作。而且智能合约的执行是原子性的，与传统数据库的原子操作一致，因此在实际应用中的优势将更加明显。

LevelDB

LevelDB是Google实现的一个高效的Key-Value型数据库，适用于“写多读少”的存储场景。LevelDB的原理基于LSM (Log Structured Merge)文件树，本质上是一个文件系统，其特点如下：

1. 默认按key的字典序存储。
2. 数据库操作简单，基本操作有：Put()、Delete()、Get()、Batch()等。
3. 可以通过迭代器遍历数据。
4. 无内置C/S架构，无需安装即可直接使用。

如图2.1所示LevelDB的整体架构可以分为内存和磁盘两部分。如图所示，当要向数据库写入一条数据时，首先要写入log文件中，再写入memtable中，当memtable到达一定大小时，将转变为immutable memtable，并压缩数据存储到磁盘sstable中，同时系统也会重新创建新的memtable用于接受新的写入数据。



**图2.1 LevelDB整体架构**

Beego

系统后端服务器的实现使用Golang编程语言的Beego框架，以Restful的形式提供接口。

如图2.2所示为Beego的执行逻辑：接受到客户端请求后，对路由进行转发，并将过滤过滤，在controller层进行处理并将结果重新返回给用户。其中Beego在控制层中耦合了各个模块，如ORM连接数据库能够快速根据model创建出数据库表，并通过ORM函数对数据库表进行操作，能够大大减小开发人员用在数据库处理的时间。

**图2.2 Beego运行架构**

此外，Beego也是一个比较典型的MVC框架，controller层并不会进行业务的具体处理，而是交由service层来处理具体逻辑，在model层进行数据库的交互。

AngularJS

AngularJS是一个JavaScript的前端框架，在开发单页应用中具有很大的优势。其有以下特征：

1. 数据绑定：在传统JavaScript中，开发人员需要做大量的工作用于查询和监听DOM中的元素变化，DOM中的元素变化会影响Model层，同时Model层的变化也会使得DOM中的元素发生变化，如果手动来处理这个过程会非常繁琐。而AngularJS的数据双向绑定特性能够帮助开发人员处理数据流向，从而节省开发时间。
2. 模板：在AngularJS中，一个模板就是一个组件，模块化的特性使得组件能够重用，减少重复代码，同时也能更方便DOM操作。
3. 指令：AngularJS将指令引入了JavaScript当中，通过以ng作为前缀的指令能够更方便地修改元素或操作DOM属性。

系统分析与设计

需求分析

当前系统的直接用户为使用趣链科技Hyperchain平台的用户，他们在区块链的基础上运行着相关业务的应用，但无法实时地监测到区块链的运行情况，无法直观地看到当前区块链的交易量变化情况等。此外，任何一个节点的异常都有可能使得整条链的运行效率降低，进而影响其上层应用，因此对于区块链节点的异常报警也是用户需要的。此外，由于区块链应用的实现与智能合约密不可分，一个智能合约的管理模块也必不可少。

### 功能性需求

1. 动态展示区块链数据

系统首先应该能让用户选择要监控的区块链节点，并且图形化地向用户展示当前最新的区块链数据，如最新区块号、总交易数、吞吐量、延迟、各阶段交易数、智能合约数量、节点运行状况等。

1. 区块链异常报警

用户根据自己的需求设置区块链报警的阈值，当监控的节点发生异常或者监控的某些值超过设定的阈值，系统会向用户发送报警信息，报警信息的格式和发送方法由用户自己定制。

1. 智能合约管理

系统提供用户对智能合约进行管理，一个项目对应一份智能合约代码。用户可以创建智能合约项目，并对合约进行编译、部署和调用，并可以查看合约的调用历史，以及合约中的变量情况。此外，系统还提供用户对智能合约进行升级和冻结。

### 非功能性需求

1. 性能需求

由于该系统的应用场景较小，仅提供一个账号使用，因此并发量较小，故要求系统的一般响应时间不大于2秒。

1. 安全性

采用权限控制、用户密码加密的形式提高系统的安全性。同时保证在系统中的操作不会对区块链本身造成影响，不会因为系统中的操作造成区块链数据异常或区块链服务器宕机。

1. 易用性

提高系统的易用性，保证系统中界面简洁清晰，各指标监控显示通俗易懂，用户操作提示浅显易懂，提高用户体验。

1. 兼容性

兼容IE、Chrome、火狐、Safari等各大主流浏览器，MacOS、Windows等主流操作系统，保证系统在不同分辨率、不同网络情况下均能正常使用。

可行性分析

节点数据动态展示可行性

本项目采用趣链科技的Hyperchain底层平台作为监控对象，其本身具有一套完整的RPC接口。本项目可以通过调用该底层平台的SDK来获取当前节点的区块信息，并可以通过遍历所有区块来获取历史数据。前端定时进行轮询即可实现实时展示节点数据信息。且当前主流的比特币、以太坊均有相关的系统监控当前的交易量，也说明了监控区块链数据的可行性。

节点异常报警可行性

当前向用户报警的方式有很多，比如页面消息推送，短信提醒，邮件提醒等，短信提醒和邮件提醒可以使用如网易云信等提供的服务，不过是付费的。而消息推送的方式，也已经有一些比较成熟的库可以使用，因此在消息通知方面具有可行性。而在数据监控上，用户在添加节点时可以设定报警参数的阈值，本项目会在后台一直运行一个进程，当监测到区块链某个值超出设定的阈值时，调用消息通知模块实现报警。

可视化智能合约管理可行性

智能合约在区块链中已经是应用很广的一种技术，当前对于智能合约的操作主要有编译、部署和调用三个步骤，第一步编译的输入为合约代码，输出为合约的abi和bin，第二部部署的输入为abi或bin，输出为一个部署成功的合约地址，第三部调用的输入为合约地址以及要调用的函数，输出为调用的结果。因此对于智能合约的操作都是能够通过简单的输入输出来实现，而合约函数的调用利用可视化界面可以实现快捷的操作。而通过合约的调用也可以实现对合约变量的实时监控，因此，可视化的智能合约管理同样具有可行性。

结论

综合以上三点，说明项目最主要的三个模块在技术上是可行的，而其他一些模块通过前后端框架的特性也能够快速实现，而通过作者在项目开发中的工作，该系统的实现是可行的。

核心功能设计

### 节点管理

节点管理是其他功能模块的基础，可以通过配置和注册将区块链添加到监控平台，同时通过对区块链分区的切换实现对不同区块链的监控和数据浏览，并且制定智能合约的部署位置。

#### 分区管理

一条区块链是一组节点的集合，因此在添加节点之前首先要创建一个分区，用于放置该区块链的所有节点。分区在创建时需要由用户配置一些参数，参数包括：分区名称、交易总量上限、交易变化率上限、交易吞吐量上限、交易成功率下限、平均交易响应时间上限、交易失败笔数上限、交易响应率下限、文件系统空间占用率上限、文件系统文件数量上限以及各个指标的监控周期。

这些指标中，控制上下限的指标主要用于报警阈值，当监控到的数据超过某个指标的上限，或者低于某个指标的下限，则触发报警机制。监控周期则是控制监控后台向区块链获取数据的频率。

对分区的管理包含如下操作：

* 添加分区：添加一个分区，进行上述指标的配置，即可生成一个分区，此时的分区还没有添加节点，故不能进行监控和报警。
* 删除分区：从平台中删除一个分区，无法再对该分区进行监控。
* 查看配置：可以查看分区创建时的监控配置，并可以进行修改。
* 节点列表：可以查看在该分区下添加的区块链节点。
* 启动/停止监控：对运行正常的分区可以开启监控，此时正在监控的其他分区会停止监控。
* 启动/停止报警：对运行正常的分区可以开启报警，此时正在监控的其他分区会停止报警。

#### 接入点管理

一个分区代表一条区块链，而在节点管理中添加的则是该链上的节点。在同一分区中不允许存在位于两条链上的接入点。

节点管理包含如下操作：

* 添加节点：输入该分区区块链中一个节点的地址及rpc端口，并取一个别名，若节点确实存在于该区块链中，则添加成功。
* 删除节点：仅在平台中移除该接入点，对区块链不会产生影响。
* 设置为接入点：用于设置该平台的区块链数据从哪一个节点中获取，以及智能合约部署在哪个节点上。
* 查看系统消息：查看该节点的运行日志。

### 区块链监控

监控是该系统的核心功能，平台将不仅监控区块链的数据并进行处理汇总，还能够监控节点的文件占用情况，同时能够展示区块列表。

#### 区块链数据概览

当平台接入到区块链节点后，即可周期性地从接入点采集区块信息并进行处理。区块链节点只提供最原始的区块和交易数据查询，而需要展示的一些交易走势图标、交易吞吐、交易成功率等丰富的监控指标则需要后台进行额外处理。

#### 系统资源监控

该部分的指标主要有系统文件数量、磁盘空间占用率、系统日志等，为非区块链提供，因此需要在节点服务器中运行一个系统监控辅助程序，才能将这些信息返回给后台。

#### 区块历史

平台还提供了区块历史的浏览和查询，后台不保存另一套账本，而是直接从节点信息中获取，确保每一次的查询均与区块链中的账本结果相同。

### 监控报警

在分区创建的时候已经设置了报警的阈值，当后台监控到的数据超过阈值，则能够触发报警。报警方式目前支持socket和web-service，当前只提供接口，还需根据企业自身情况进行配置。另外电子邮箱与短信的方式也能够实现，不过也需要付费定制，因此当前系统暂不支持。

### 智能合约管理

智能合约模块是提供给业务人员进行合约的编译、部署和调用，并且提供用户查看合约调用历史、合约变量浏览等。

#### 合约项目管理

每一份合约对应着平台中的一个项目，项目在创建时需要输入合约代码或载入合约文件，在通过编译后即可创建成功。另外也可以删除项目，当然删除项目也仅为在平台数据库中删除，不会对已经部署在区块链上的合约有任何影响。

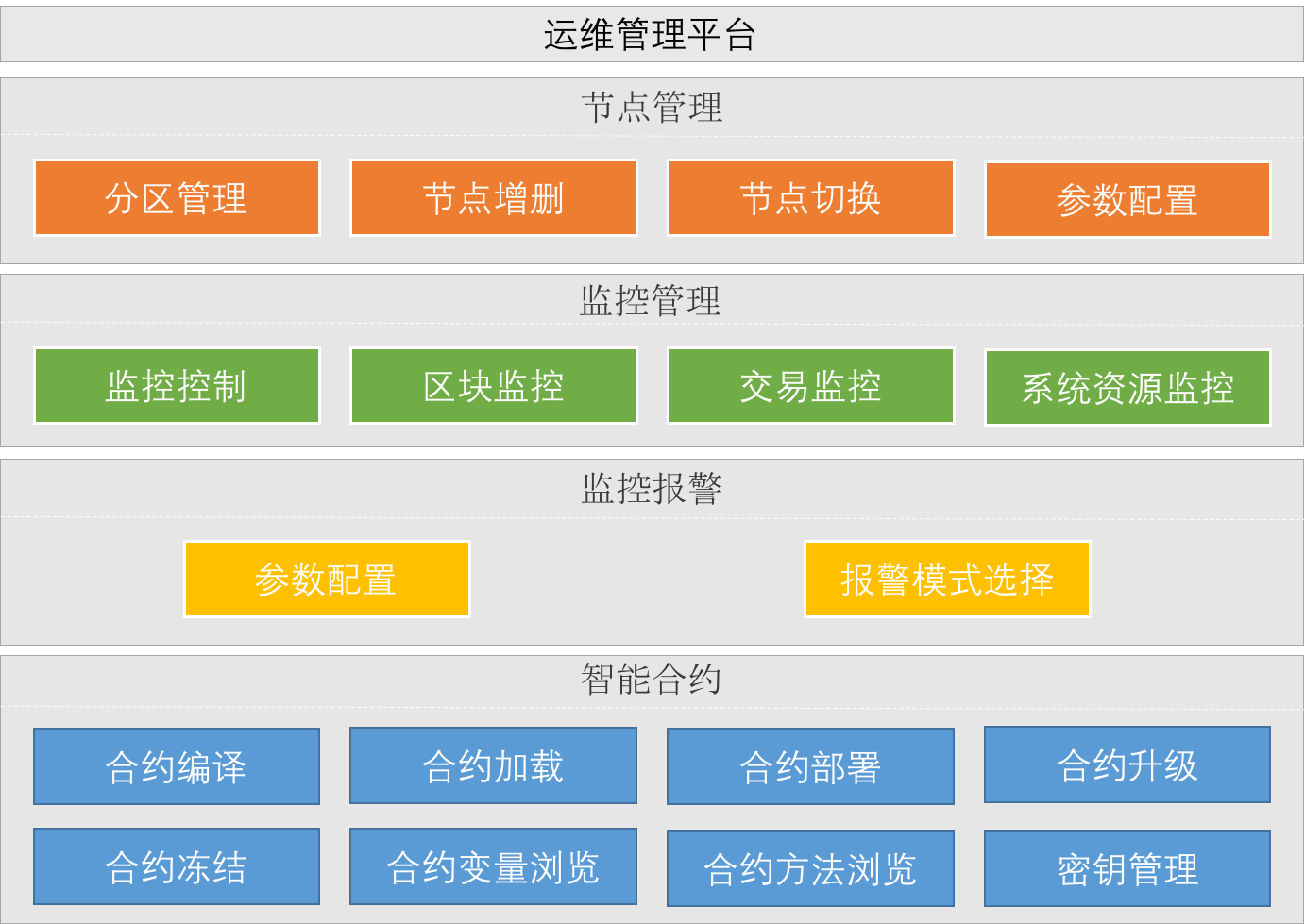
#### 密钥生成模块

合约在部署和调用时均需要密钥，系统将提供一个便捷的密钥生成模块，用于一键生成私钥、公钥和地址，并提供下载功能。

#### 合约操作模块

对已部署的智能合约可以进行的操作包括：

* 升级：将升级后的合约编译后的abi和bin值传入，即可完成动态升级。
* 冻结：合约被冻结后无法升级、调用和浏览。
* 调用：合约在部署后，后台会解析合约的abi，将合约函数的方法、参数分离，用户可以方便地输入入参，实现调用。
* 浏览：浏览的数据包括交易、变量和方法，可以输入要浏览的区块号区间，选择要查看的变量和方法来进行浏览。



**图3.1 系统功能模块**

数据库设计

### 数据库选择

该系统中需要存入数据库的数据结构并不复杂，区块链中的相关信息大多直接向节点请求，而非存储在后端数据库中，在后端数据库中一般只存储用户信息、节点管理信息（如添加了的节点ip，报警阈值等）、以及智能合约的代码与bin等。因此选择Key-Value型数据库足以支持该系统的读写操作，并非需要使用较为复杂的关系型数据库。

此外，使用MongoDB、Mysql等关系型数据库时均需要在服务器中安装特定的客户端。而在实际环境中，用户不一定能给系统权限以安装一个客户端（特别是银行用户），且安装过程要考虑系统环境，较为繁杂。

因此，该项目采用了Key-Value型数据库LevelDB，在使用时无需安装客户端，能够更好得适应生产环境。

### 存储结构设计

由于LevelDB是一个Key-Value型数据库，且存储顺序默认是按照Key的字典序排列的，而需要存入数据库的数据结构是比较单一的，结构体一般只有用户user，区块链分区chain，接入点endpoint，监控数据txstatistics，智能合约项目project等，因此为了提高区块链的查找数据，每一种数据类型的数据在存储时，都在其Key值前加入前缀，一方面可以更清晰地了解存储的数据是什么结构，一方面可以提升查找效率，不必每一次的查询都从头开始。而且LevelDB的迭代器也提供slice功能，可以加快查询效率。

系统实现

系统架构

整体技术架构

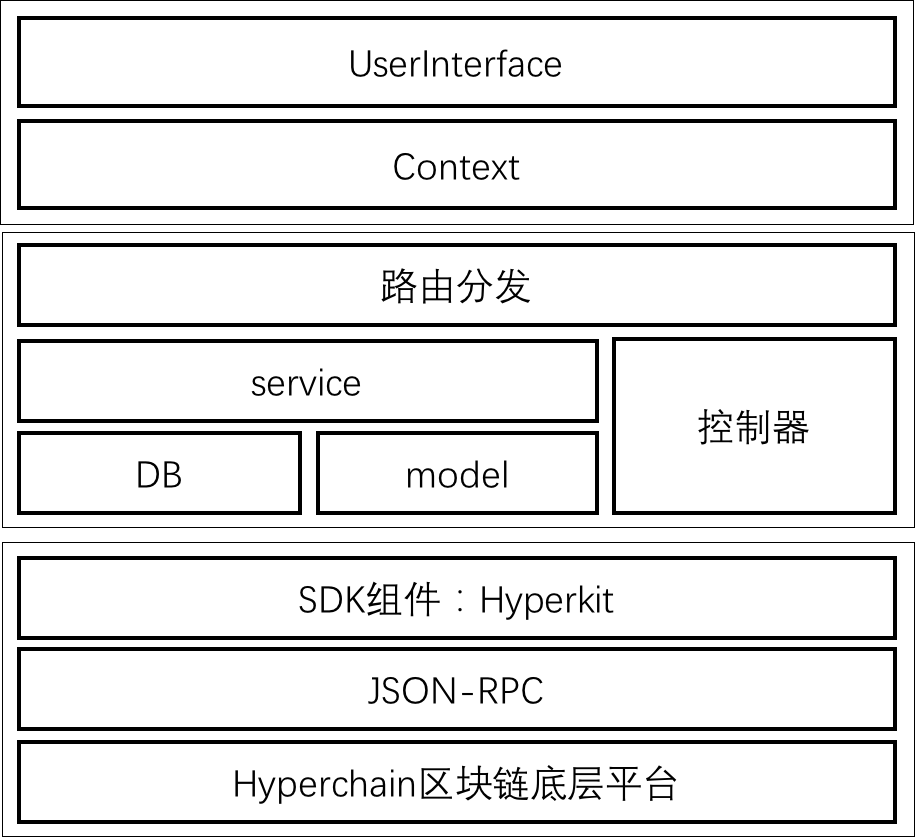
区块链管理和运维平台的后台实现基于GO语言开发，前端页面采用AngularJS框架开发，前后端通讯的HTTP接口均为RestFul风格。

如图所示为该项目的技术架构。处于顶层的前端模块UserInterface，使用的是AngularJS框架，页面布局和样式通过HTML和CSS实现，逻辑部分则是通过JavaScript代码进行实现。前端通过ajax发送请求，并通过双向绑定的模式将请求结果即时反应在页面中。

位于中间层的后端模块，使用的是基于GO语言的开源Web框架Beego。该框架的设计遵循MVC设计模式，收到请求后通过router进行路由转发，controller处理请求数据，service层处理逻辑，model层负责与底层数据库和Hyperchain进行交互。同时，后端还有一个消息推送的模块，拥有webService与socket两种可选模式，能够及时将后端的报警信息推送到指定地址。

最底层的是LevelDB数据库与Hyperchain。其中LevelDB主要存储区块链运维管理系统所产生的相关数据，如用户账号、添加的节点信息、在平台上创建的智能合约等；而后端与Hyperchain的交互则通过Go语言的SDK——Hyperkit来实现，保证后端实时获取区块链最新数据并传给前端进行展示。

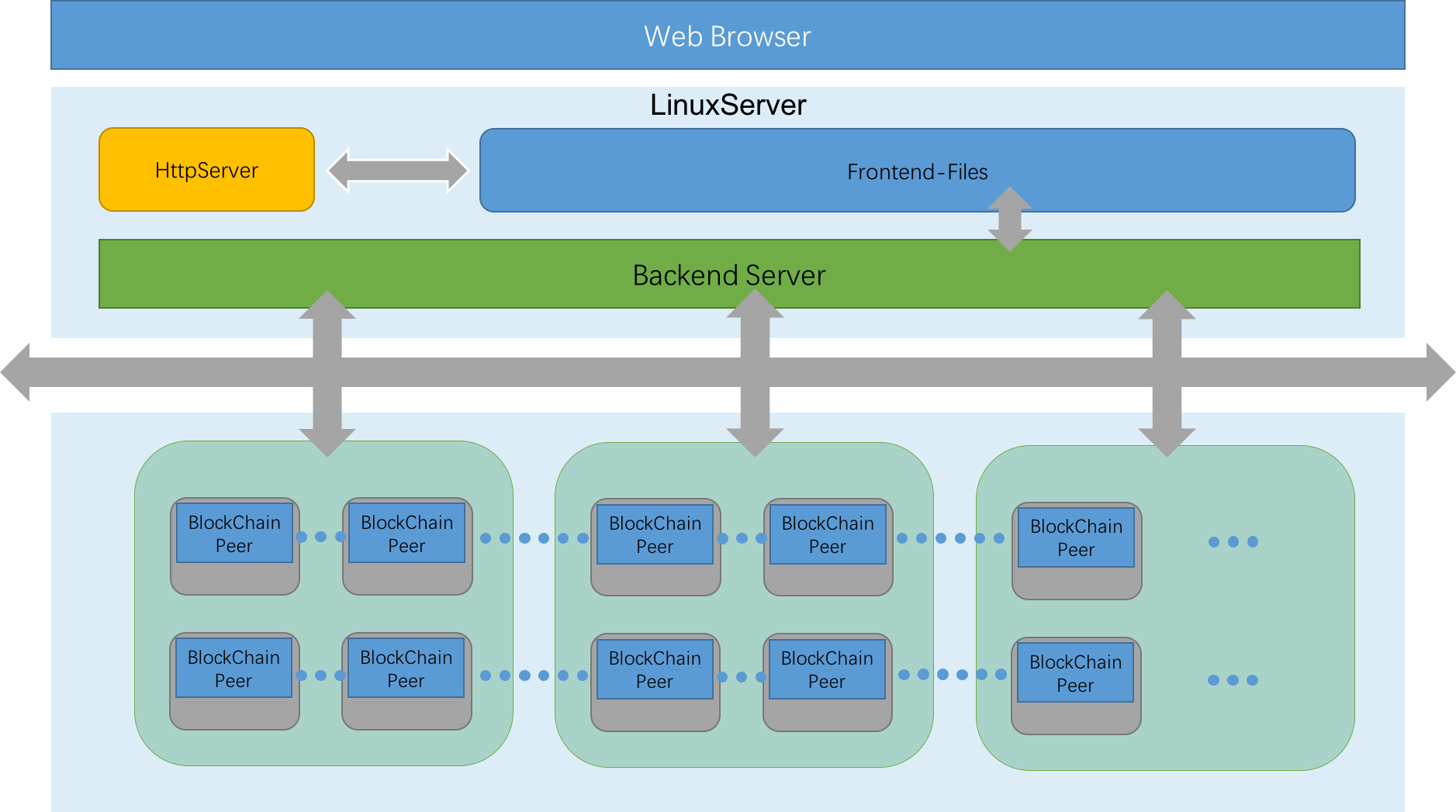
后端与Hyperchain的数据交互通过channel实时获取，保证后端拿到的数据与区块链始终一致，而前端获取某些区块链概览信息则通过定时轮询实现。



**图4.1 整体技术架构**

部署架构

该区块链运维管理平台的前后端分开部署，后端启动后将端口暴露给前端，前端启动后将端口暴露给用户。区块链的部署不属于该系统，当区块链部署完成之后，用户可以将节点添加到该系统。当前系统仅支持趣链科技的Hyperchain区块链，且该区块链启动的要求为四个节点及以上。系统与区块链的数据交互则通过go语言的SDK——Hyperkit来实现。



**图4.2 系统部署架构**

API服务器架构

后端API服务器通过Beego框架实现，主要有以下四个模块：

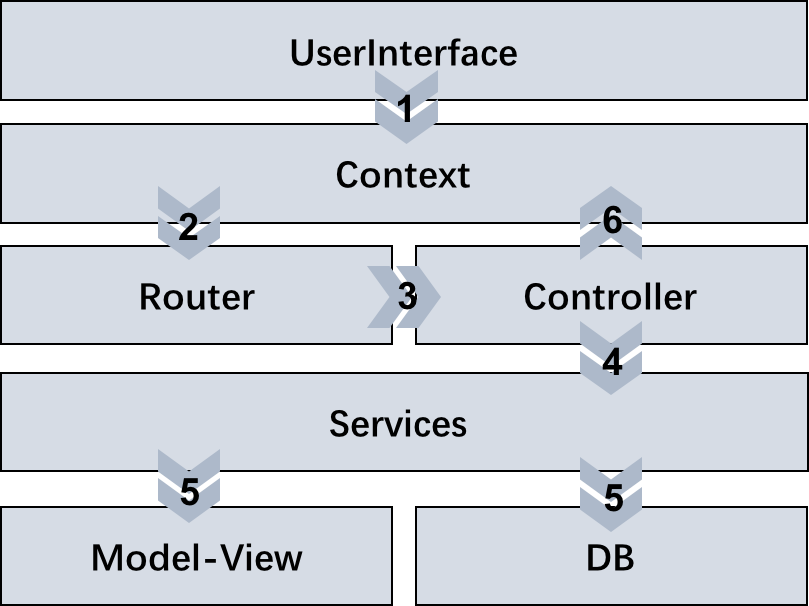
· Router：Router模块负责对前端发送的restful请求进行路由转发和预处理，预处理包括跨域请求预处理，认证信息预处理等，预处理完成后，Router模块会将请求分发到相应的Controller中进行处理。

· Controller：该模块一方面负责将请求参数进行解析处理，并且调用相应的Service函数对请求进行逻辑处理，并将Service处理的结果重新返回给前端。

· Service：该模块是后端真正处理逻辑的地方，该模块根据传入参数，进行逻辑判断，并调用model层函数对数据库进行增删改查，或者通过SDK对区块链进行交互，并将处理的结果包装成返回的数据格式并传回给Controller层。

· ModelView：视图模型，定义了前端传入的数据类型格式以及Service处理完后返回给用户界面所需要的数据模型，ModelView的调用者可以是Service也可以是Controller。

· DB：数据模型，定义了数据库中的数据对象与Go语言对象的映射，对数据库的读写操作也位于该模块执行，但有时一些较为简单的数据库操作也会直接在service层进行。



**图4.3 API服务器架构**

模块实现

### 区块链监控数据汇总

1. Router实现

指定路由“/v2/summary”的get请求，将其分发到Summary的Controller中的Get方法进行处理。

|  |
| --- |
| ns := beego.NewNamespace("/v2",  beego.NSRouter("/summary", &controllers.Summary{}), |

1. Controller实现

在该函数中，首先拿到传入请求体中的用户信息，并获取用户正在监控的节点所在的分区。进而调用service层中的endpointApi中的GetSpv()函数，来获取要返回的监控信息。调用GetDayHis()等函数获取区块链统计信息，同理通过GetContractCount()获取合约数量，GetAlarmDataFromTimer()获取报警指标。在获取完所有数据之后，将整个结构体以summaryResp的结构传到前端。

|  |
| --- |
| func (this \*Summary) Get() {  user := this.Ctx.Input.GetData("currentUser")  res := model\_view.Resp{}  summaryResp := model\_view.SummaryResp{}  dbApi := db.GetApi()  chain, err := dbApi.GetLinkedChainbyUser(user.(\*models.User).Id.Hex())  if err != nil {  res.SetMsg("GetLinkedChain by User error: " + err.Error())  this.Data["json"] = &res  this.ServeJSON()  return  }  service := services.GetOne()  endpointApi, err := service.ChainManager.Which(chain.Id\_.Hex())  if err != nil {  beego.Warn(err.Error())  } else {  spv := endpointApi.GetSpv()  summaryResp.ExeuteTime = spv.Latency  summaryResp.LatestBlk = spv.BlockHeight  summaryResp.Throughput = spv.Tps  summaryResp.TotalTxs = spv.TxCount  }  txStatistics := &model\_view.TxStatisticsResp{  Day: endpointApi.GetDayHis(),  Week: endpointApi.GetWeekHis(),  Month: endpointApi.GetMonthHis(),  }  summaryResp.TxStatistics = txStatistics  summaryResp.NodesInfo, err = endpointApi.NodesInfo()  if err != nil {  beego.Informational("GetSpv error: " + err.Error())  summaryResp.NodesInfo.Active = 0  summaryResp.NodesInfo.Active = 0  summaryResp.NodesInfo.Primary = "not set"  }  service = services.GetOne()  contracts, err := service.GetContractCount(chain.Name)  if err != nil {  res.SetMsg("GetContractCount error: " + err.Error())  this.Data["json"] = &res  this.ServeJSON()  return  }  summaryResp.SmartCntr = contracts  alarm := endpointApi.GetAlarmDataFromTimer()  alarmData := &model\_view.AlarmData{  Tps: alarm.Tps,  AvgResponseTime: alarm.AvgResponseTime,  }  summaryResp.AlarmData = alarmData  this.Data["json"] = &summaryResp  this.ServeJSON()  } |

1. Service实现

由于summary接口所需要返回的信息较多，一个Controller将调用多个Service，有的Service是从数据库中获取数据，而有的Service则是读取内存中保存的区块链实时数据。比如GetContractCount()函数，是直接统计数据库存各个项目所拥有的智能合约数目，并根据合约的部署状态做汇总。

|  |
| --- |
| func (this \*SummaryService) GetContractCount(chainName string) (\*model\_view.SummaryCountMap, error) {  contrcount := model\_view.SummaryCountMap{  "deployed": 0,  "undeployed": 0,  }  projects, err := this.dbApi.FindProjects()  if err != nil {  return nil, err  }  for \_, proj := range projects {  if proj.ChainName == chainName {  for \_, contr := range proj.Contracts {  if contr.Address == "" {  contrcount["undeployed"] = contrcount["undeployed"] + 1  } else {  contrcount["deployed"] = contrcount["deployed"] + 1  }  }  }  }  return &contrcount, nil  } |

而更多的监控信息是从服务器的内存中获取的，以GetSpv()函数为例，区块链的总交易数是直接调用SDK的rpc接口来获取，而其余数据则是通过resolver的interface，直接获取内存中spv存储的数据。而resolver保存区块链数据的方法在5.3.1部分会详细描述。

|  |
| --- |
| func (this \*BlockPool) GetSpv() \*Spv {  spv := this.resolver.GetSpv()  txCount, err := this.rpc.GetTransactionsCount()  if err != nil {  return spv  }  spv.TxCount = txCount.Count  return spv  } |

### 报警

由于报警模块是通过消息推送实现的，故后端并不会在Router中接收到相关请求信息，而是直接在Service层监控区块链数据，当发生异常时直接进行socket或webService消息推送。

在系统完成区块链添加并开启监控时，后台将开启一个定时器Timer，并且并发执行多个线程，分别用于检测各个指标是否超出阈值。

|  |
| --- |
| func (this \*TimerService) StartTimer() error {  r, err := lib.GetAvailableRpcByChainId(this.Chain.Id\_.Hex())  if err != nil {  beego.Warn(err.Error())  return fmt.Errorf("NO\_AVAILABLE\_ENDPOINT")  }  this.Rpc = r  if !this.TimerStarted {  beego.Info("Start alarm timer in chain ", this.Chain.Name)  this.initPreData()  go this.doTotalVolumnTimer()  go this.doChangeRateTimer()  go this.doThroughputTimer()  go this.doSuccessRateTimer()  go this.doAvgResponseTimer()  go this.doResponseRateTimer()  go this.doFailNumberTimer()  go this.doFileSystemFileCount()  go this.doFileSystemSpaceUtilization()  api := db.GetApi()  this.Chain.SpvStatus = "start"  api.UpdateChainById(this.Chain.Id\_, this.Chain)  this.TimerStarted = true  }  return nil  } |

以doFailNumberTimer()为例。后台将先根据存入的检测失败交易数频率值新建一个定时器Ticker，每隔一定时间查询失败交易数总量，当实际的失败交易数大于设定的失败交易数最高值时，后台将发送一条报警信息。

|  |
| --- |
| func (this \*TimerService) doFailNumberTimer() {  this.failNumberTicker = time.NewTicker(time.Second \* time.Duration(this.Chain.FailNumberInterval))  for t := range this.failNumberTicker.C {  intervalFail := this.getFailNumberCount(this.Chain.FailNumberInterval)  if intervalFail > this.Chain.FailNumberUpperLimit {  err := this.sendAlarm("FailNumber", fmt.Sprintf("%v", intervalFail), fmt.Sprintf("交易失败已达%v笔,超过阈值%v", intervalFail, this.Chain.FailNumberUpperLimit))  if err != nil {  beego.Info("Send fail number alarm error:", err)  }  }  beego.Info("Tick at", t, ", chain name:", this.Chain.Name, ", fail transaction count:", intervalFail)  this.dataGeneratedByTimer.FailTransactionCount = intervalFail  }  } |

这里根据选择的报警模式，通过socket或webService的方式将报警信息推送给设定的地址。

|  |
| --- |
| func (this \*TimerService) sendAlarm(parameter, instance, value string) error {  enableAlarm, \_ := beego.AppConfig.Bool("enableAlarm")  if enableAlarm && this.EnableSendAlarm {  beego.Info("Send alarm to service,", parameter, "is", instance)  opDetail := this.generateOpDetail(parameter, instance, value)  if this.alarmMode == "webservice" {  service := NewAlarmSwitchService("", false, nil)  request := &TransferAlarmRequest{}  request.SerSupplier = "OMS-IMP"  request.SerCaller = "HPC"  request.CallerPwd = "hyperchain"  request.CallTime = time.Now().Format("2006-01-02 15:04:05")  request.OpDetail = opDetail  response, err := service.TransferAlarm(request)  if err != nil {  return err  }  result := response.TransferAlarmReturn  splitStr := strings.Split(result, ";")  flagStr := strings.Split(splitStr[0], "=")[1]  flagBool, \_ := strconv.ParseBool(flagStr)  if !flagBool {  return errors.New(strings.Split(splitStr[1], "=")[1])  }  return nil  } else if this.alarmMode == "socket" {  err := Send(opDetail)  return err  }  }  return nil  } |

使用socket进行推送时，首先与目标地址建立一个tcp连接，通过Wirte方法发送信息，通过Read方法接收返回信息。

|  |
| --- |
| func Send(content string) error {  conn, err := net.DialTCP("tcp", nil, tcp)  if err != nil {  beego.Debug("Fatal error:", err.Error())  return err  }  beego.Info("Send alarm:", content)  \_, err = conn.Write([]byte(content))  if err != nil {  beego.Info("Send alarm error,", err.Error())  return err  }  buf := make([]byte, 1024)  n, err := conn.Read(buf)  beego.Info("Get alarm result", string(buf[:n]))  conn.Close()  return err  } |

而webService是基于http协议的，因此传输有格式要求，需要配置好header与body才能进行传输，接收时也需要对返回体进行xml解析。

|  |
| --- |
| func (s \*SOAPClient) Call(soapAction string, request, response interface{}) error {  envelope := SOAPEnvelope{}  if s.headers != nil && len(s.headers) > 0 {  soapHeader := &SOAPHeader{Items: make([]interface{}, len(s.headers))}  copy(soapHeader.Items, s.headers)  envelope.Header = soapHeader  }  envelope.Body.Content = request  buffer := new(bytes.Buffer)  encoder := xml.NewEncoder(buffer)  //encoder.Indent(" ", " ")  if err := encoder.Encode(envelope); err != nil {  return err  }  if err := encoder.Flush(); err != nil {  return err  }  log.Println(buffer.String())  req, err := http.NewRequest("POST", s.url, buffer)  if err != nil {  return err  }  if s.auth != nil {  req.SetBasicAuth(s.auth.Login, s.auth.Password)  }  req.Header.Add("Content-Type", "text/xml; charset=\"utf-8\"")  req.Header.Add("SOAPAction", soapAction)  req.Header.Set("User-Agent", "gowsdl/0.1")  req.Close = true  tr := &http.Transport{  TLSClientConfig: &tls.Config{  InsecureSkipVerify: s.tls,  },  Dial: dialTimeout,  }  client := &http.Client{Transport: tr}  res, err := client.Do(req)  if err != nil {  return err  }  defer res.Body.Close()  rawbody, err := ioutil.ReadAll(res.Body)  if err != nil {  return err  }  if len(rawbody) == 0 {  log.Println("empty response")  return nil  }  log.Println(string(rawbody))  respEnvelope := new(SOAPEnvelope)  respEnvelope.Body = SOAPBody{Content: response}  err = xml.Unmarshal(rawbody, respEnvelope)  if err != nil {  return err  }  fault := respEnvelope.Body.Fault  if fault != nil {  return fault  }  return nil  } |

### 智能合约管理

1. 智能合约项目管理

该系统中的一个智能合约项目对应于一份智能合约代码，项目的增删改查功能通过/project路由下的post,delete,put,get请求实现。Router将请求分发到Controller中，并调用相应的Service函数对数据库进行操作。以添加一个项目为例，Controller层中调用Service的CreatePorject函数。

|  |
| --- |
| func (this \*ProjectController) Post() {  defer this.ServeJSON()  service := services.GetOne()  user := this.Ctx.Input.GetData("currentUser")  dbApi := db.GetApi()  chain, err := dbApi.GetLinkedChainbyUser(user.(\*models.User).Id.Hex())  if err != nil {  this.Ctx.Output.Status = 400  this.Data["json"] = model\_view.Resp{Msg: err.Error()}  return  }  p, err := service.CreateProject(this.Ctx.Input.RequestBody, chain.Id\_.Hex())  if err != nil {  this.Ctx.Output.Status = 400  this.Data["json"] = model\_view.Resp{Msg: err.Error()}  return  }  this.Ctx.Output.Status = 200  this.Data["json"] = p  this.ServeJSON()  } |

在Service层中，后台首先将请求体解析，并校验该项目是否已经存在。若项目名称不存在数据库中，则继续解析ABI并保存。prepareProject函数的作用是存储项目中合约的状态。当校验全部完成且没有错误后，将项目添加到数据库中持久化存储。

|  |
| --- |
| func (this \*ProjectService) CreateProject(body []byte, chainId string) (\*models.Project, error) {  var p \*models.Project  err := json.Unmarshal(body, &p)  if err != nil {  return nil, fmt.Errorf("Umarshal: %v", err.Error())  }  proj, err := this.dbApi.FindProjectByName(p.Name)  if len(proj) > 0 {  return nil, fmt.Errorf("Proj\_Exist")  }  if err != nil {  return nil, fmt.Errorf("project %v db: %v", p.Name, err)  }  for \_, contract := range p.Contracts {  abiString, \_ := json.Marshal(contract.Abi)  for \_, method := range contract.Abi {  method.MethodID = util.GetMethodID(string(abiString), method.Name)  }  }  p.ChainId = chainId  if err := this.prepareProject(p); err != nil {  return nil, err  }  if err := this.dbApi.InsertProject(p); err != nil {  return nil, err  }  return p, nil  } |

1. 智能合约部署

部署智能合约的本质是将编译后得到的Bin存入区块链。在区块链上智能合约的部署需要三个输入，分别为合约Bin、合约Abi以及区块链账户地址，输出仅包含本次交易的Hash，通过该Hash可以查询到交易详情，其中包含合约地址。

路由从project/deploy分发到Controller层，并由Service层的DeployProject函数进行处理。在该函数中，首先将传入参数解析为结构体，再将传入的私钥转换为账户地址，调用rpc接口的Deploy函数在区块链上部署合约，若部署成功则更新该合约在数据库中存储的状态，包括将部署的合约地址存入数据库中，用于下次调用使用。整个过程完成后，Controller层将成功部署的信息重新传回前端。

|  |
| --- |
| func (this \*ProjectService) DeployProject(body []byte) (\*models.Project, error) {  var req struct {  Id string  PrivateKey string  ContractName string  Password string  Params []interface{}  }  if err := json.Unmarshal(body, &req); err != nil {  return nil, fmt.Errorf("deploy json: %v", err.Error())  }  if req.Password != "" {  decodedPK, err := decodePrivateKey(req.PrivateKey, req.Password)  if err != nil {  return nil, fmt.Errorf("Decode private key error: %v", err.Error())  }  req.PrivateKey = decodedPK  }  p, err := service.GetProject(req.Id)  if err != nil {  return nil, fmt.Errorf("get project %v error: %v", req.Id, err.Error())  }  ci := findContract(p.Contracts, req.ContractName)  if ci == -1 {  return nil, fmt.Errorf("project: no contract '%v' exists" + req.ContractName)  }  from, err := encrypt.PrivateToAddress(req.PrivateKey)  if err != nil {  return nil, fmt.Errorf("invalid private key")  }  abiString, err := json.Marshal(p.Contracts[ci].Abi)  if err != nil {  return nil, fmt.Errorf("unmarshal abi failed: %v", err.Error())  }  service := GetOne()  api, err := service.ChainManager.Which(p.ChainId)  if err != nil {  return nil, err  }  addr, err := api.Deploy(from, p.Contracts[ci].Code, p.VmType, req.PrivateKey, string(abiString), req.Params...)  if err != nil {  return nil, err  }  p.Contracts[ci].Address = addr  p.Contracts[ci].Status = "已部署"  p.Contracts[ci].Time = time.Now().Format("2006-01-02 15:04:05")  p.Status = checkDeployedOrLoaded(p.Contracts, true)  proj, err := this.dbApi.FindProjectById(p.Id\_.Hex())  if err != nil {  return nil, err  }  proj.Deployed = true  proj.Status = p.Status  proj.Contracts = p.Contracts  if err := this.dbApi.UpdateProject(req.Id, proj); err != nil {  return nil, fmt.Errorf("db update addr failed: %v", err)  }  return p, nil  } |

1. 智能合约调用

智能合约调用时对前面部署的智能合约中所定义的函数进行调用，调用时需要输入以下参数：

·合约地址：部署成功时返回的Hash值即为合约地址，已存在数据库中。

·函数名称：在智能合约中定义的函数名。

·参数列表：一个字符串数组，长度必须与定义函数传入参数的个数保持一致。

同样，智能合约的调用时通过路由/project/invoke进入Controller层，并调用Service层的InvokeProj函数。函数中首先解析传入参数，在数据库中找到要调用的合约的地址，进而通过SDK的invoke函数来进行调用，并将获取的调用结果返回到Controller层的返回参数中，传回给前端。

|  |
| --- |
| func (this \*ProjectService) InvokeProj(body []byte) (interface{}, error) {  var req struct {  Id string  PrivateKey string  ContractName string  MethodName string  MethodParams []interface{}  Constant bool  Password string  }  if err := json.Unmarshal(body, &req); err != nil {  return nil, fmt.Errorf("invoke json: %v", err)  }  if req.Password != "" {  decodedPK, err := decodePrivateKey(req.PrivateKey, req.Password)  if err != nil {  return nil, fmt.Errorf("Decode private key error: %v", err.Error())  }  req.PrivateKey = decodedPK  }  p, err := this.dbApi.FindProjectById(req.Id)  if err != nil {  return nil, fmt.Errorf("project %v db: %v", req.Id, err)  }  ci := findContract(p.Contracts, req.ContractName)  if ci == -1 {  return nil, fmt.Errorf("project: no contract '%v' exists", req.ContractName)  }  from, err := encrypt.PrivateToAddress(req.PrivateKey)  if err != nil {  return nil, fmt.Errorf("invalid private key")  }  abiString, err := json.Marshal(p.Contracts[ci].Abi)  if err != nil {  return nil, fmt.Errorf("unmarshal abi failed: %v", err)  }  service = GetOne()  api, err := service.ChainManager.Which(p.ChainId)  if err != nil {  return nil, err  }  return api.Invoke(from, p.Contracts[ci].Address, p.VmType, req.PrivateKey, string(abiString), req.MethodName, req.Constant, req.MethodParams...)  } |

系统关键算法实现

监控数据采集算法

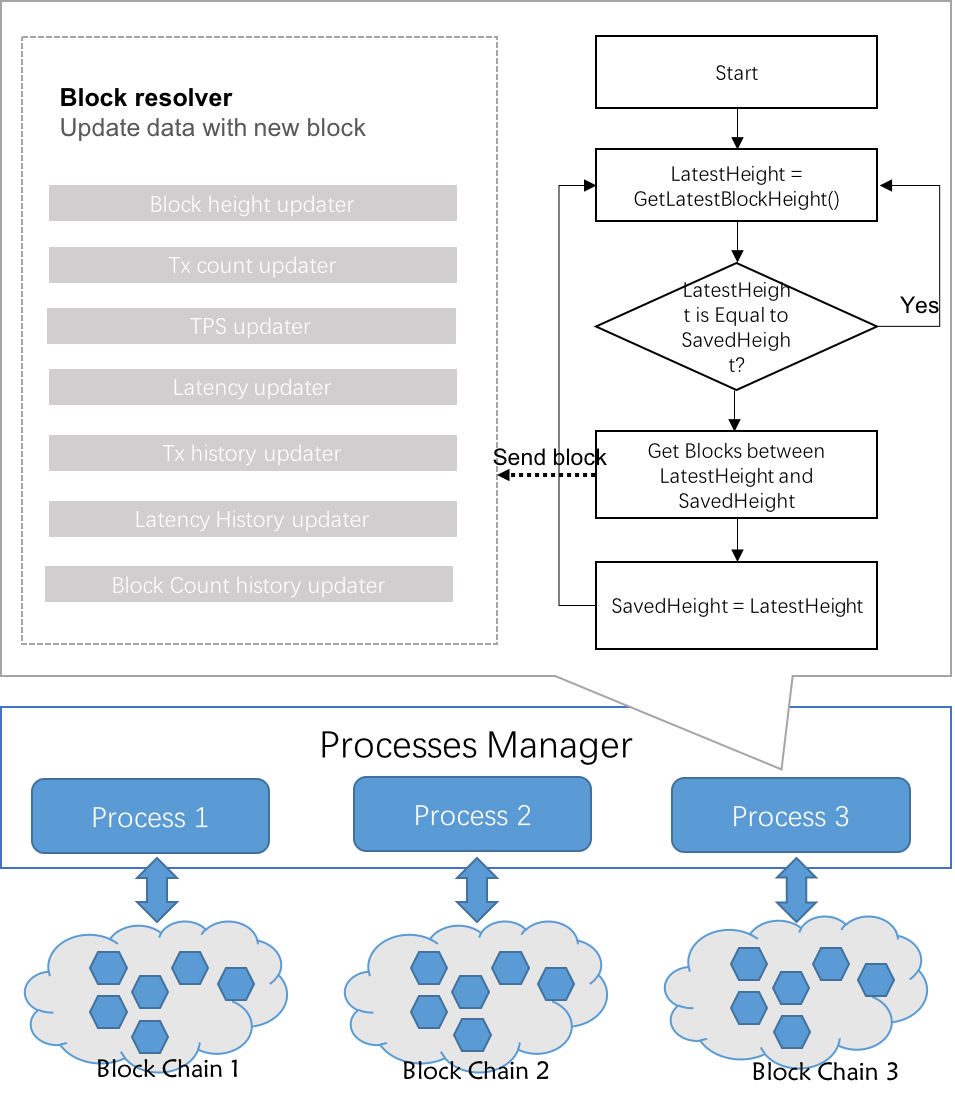
区块链数据采集是通过SDK调用底层rpc接口来实现的，如果按照传统API做法，前端调用summary接口，后端在Service层调用rpc接口来获取区块链数据，进行统计整理后传回给前端，会出现以下几个问题：

1. 接口响应缓慢：即使不考虑区块链服务器的网络情况，要调用多个rpc接口的响应速度也是有限的，而且在获取近一个月的交易数时，需要同步的数据量是非常大的。因此接口的响应速度将非常缓慢，甚至可能超时。
2. 无法保证与区块链一致：由于区块链数据同步缓慢，若同步未完成即把数据输出，会造成系统的区块信息与区块链上的真正数据不一致。
3. 资源浪费：每次数据统计都从头开始会占用大量的网络资源和带宽。

因此，如图4.4所示，本文设计了一种算法，在后端运行多个线程持续不断地获取最新区块信息并将其存在内存中，收到前端请求后直接讲内存中的数据返回，可以保证接口返回的数据与区块链一致，且响应速度很快。如图所示为该算法的实现逻辑。

当一条区块链加入到该平台中并开启监控功能，后台就会启动一个线程同步区块链数据。同步的过程如下：首先通过调用rpc接口获取区块链上的最新区块号，与当前存储的区块号进行对比，如果两个值相等，则说明这段时间内没有新的区块产生，内存中的区块数据即为真实区块数据，则继续获取区块链上的最新区块号，直到两个值不同；若内存中的区块号小于通过rpc请求获取的真实区块号，说明内存中的区块信息落后了，故调用rpc接口获取落后的这些区块的信息，并发送到resolver中进行数据处理和存储，并更新最新区块号；整个流程为一个死循环，直到系统收到关闭监控的信号。

而resolver模块的作用是将通过rpc接口获取的区块链数据整理成更直观的数据结构，并对某些值进行计算与统计。在resolver中更新的数据包括：区块高度、交易数量、交易吞吐率、交易处理时间、交易历史、处理时间历史、区块历史等。



**图4.4 监控数据采集算法流程图**

此外，在将区块链数据同步到后台的过程中，由于每次同步的间隔时间为一秒，有时候链上交易数很多，区块数也很多，一秒内无法将落后的区块全部同步，若同步未完成则会造成一些数据不统一。因此，在同步的过程中设置了GATESIZE，当前默认为10，即每次同步最多同步十个区块，可以保证在一秒内完成同步。而剩余落后的区块则会持续进行同步，直到平台的区块与区块链同步。

交易数据快速同步算法

平台中需要展示的大多数数据都可以直接通过rpc接口来获得，但是历史交易数据是需要通过统计一个月内产生的区块来获得的。如果区块链刚刚启动就开始监控，那么可以从第一个区块开始就进行实时同步，并直接更新数据。但是如果一个区块链已经运行了很久，此时将其接入到我们的平台，从第一个区块开始同步显然不现实，但区块链本身并没有提供按时间查找区块的接口。因此，如图4.5所示，本文设计了一种算法，用于快速同步近一个月的交易数据。

首先定义三个值：

·currentBlockHeight：指当前后台内存中存储的区块高度，监控启动时会置为当前区块链的区块高度。

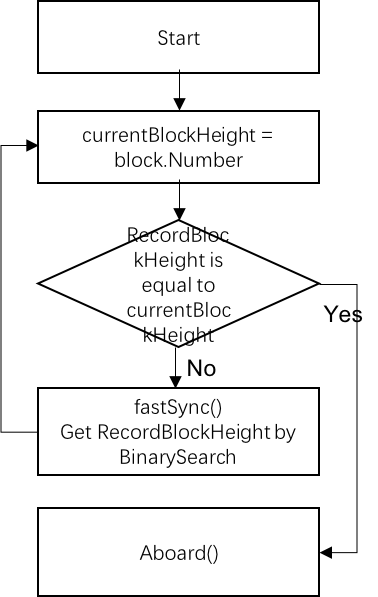
·recordBlockHeight：指当前后台数据库中存储一个月前产生的区块高度（若区块链运行时间未满一个月则置为第一个区块的区块号）。

·block.Number：当前区块链的区块高度。

当刚刚对区块链启动监控时，recordBlockHeight为1或者为上次对该区块链监控关闭前存储的区块高度，currentBlockHeight为当前区块链高度。后台将对比这两种个值判断这段时间是否有新的区块产生，如果两个值不相等，则调用快速同步函数fastSync()。

快速同步函数的作用是快速获得近一个月来的区块，并将其推入channel进行同步，而如何定位到前一个月产生的区块是该算法的关键。本文采用了二分查找的方法来快速找到一个月前的区块号。如图所示，通过一个循环来重复查找，首先判断(recordBlockHeight+currentBlockHeight)/2的区块的写入时间是否在一个月之前，从而判断一个月前的区块位于recordBlockHeight到(recordBlockHeight+currentBlockHeight)/2之间还是在(recordBlockHeight+currentBlockHeight)/2与currentBlockHeight之间，不断循环缩小范围，直到获得恰好在一个月前写入的区块。

另外，由于同步的时间可能会比较长，在这段时间内如果产生新的区块，则需要重新进行同步，直到同步后的最新区块等于区块链上的最新区块。



**图4.5 交易数据快速同步流程图**

系统测试与成果展示

功能测试

本节针对系统各功能点设计了测试用例，按照设计的测试用例对功能点进行测试并记录通过情况，共提出246个测试用例。测试共分为三轮，第一轮测试提出43个bug，修复后进行第二轮测试；第二轮测试提出13个bug，3个bug无解决方案转为优化项；回归测试通过，无明显缺陷。测试结果为系统是可用的。

用户模块

**表5.1 用户模块第一轮测试结果**

|  |  |  |  |
| --- | --- | --- | --- |
| 子模块 | 用例个数 | 通过个数 | 通过率 |
| 登录 | 6 | 6 | 100% |
| 修改密码 | 10 | 7 | 70% |
| 小计 | 16 | 13 | 81% |

用户模块中产生的缺陷主要由于在新密码设置时没有按照规范设定密码格式，明确需求后修改代码顺利通过第二轮测试。

节点管理模块

**表5.2 节点管理模块第一轮测试结果**

|  |  |  |  |
| --- | --- | --- | --- |
| 子模块 | 用例个数 | 通过个数 | 通过率 |
| 添加分区 | 9 | 8 | 89% |
| 分区列表 | 4 | 4 | 100% |
| 添加节点 | 9 | 8 | 89% |
| 节点列表 | 2 | 2 | 100% |
| 设置接入点 | 3 | 3 | 100% |
| 查看节点系统消息 | 1 | 0 | 0% |
| 配置详情 | 2 | 2 | 100% |
| 启动监控 | 4 | 3 | 75% |
| 停止监控 | 4 | 3 | 75% |
| 删除分区 | 7 | 7 | 100% |
| 小计 | 45 | 40 | 89% |

节点管理模块的缺陷较少，修复后主要功能点也均通过二轮测试。

区块链监控模块

**表5.3 区块链监控模块第一轮测试结果**

|  |  |  |  |
| --- | --- | --- | --- |
| 子模块 | 用例个数 | 通过个数 | 通过率 |
| 监控统计规则 | 18 | 14 | 78% |
| 发送交易模式 | 3 | 3 | 100% |
| 接入点 | 1 | 1 | 100% |
| 区块链监控区域 | 5 | 5 | 100% |
| 交易数区域 | 7 | 6 | 86% |
| 共识节点区域 | 3 | 3 | 100% |
| 智能合约区域 | 4 | 4 | 100% |
| 文件系统监控 | 18 | 8 | 44% |
| 小计 | 59 | 44 | 75% |

该模块的主要问题在于文件系统监控功能失效，由于该功能为额外功能，需要在节点服务器中另外启动一个进程，在系统实际使用过程中无法保证该功能正常使用，因此始终为一风险点。该缺陷的更好解决方案是在底层节点在SDK层进行封装，该项目无法进行优化。但主要功能均通过测试。

智能合约模块

**表5.4 智能合约模块第一轮测试结果**

|  |  |  |  |
| --- | --- | --- | --- |
| 子模块 | 用例个数 | 通过个数 | 通过率 |
| 创建项目 | 12 | 9 | 75% |
| 加载合约 | 8 | 6 | 75% |
| 项目详情 | 1 | 1 | 100% |
| 合约详情 | 1 | 1 | 100% |
| 部署 | 15 | 15 | 100% |
| 加载合约 | 7 | 6 | 86% |
| 调用 | 16 | 12 | 75% |
| 合约升级 | 20 | 15 | 75% |
| 冻结合约 | 8 | 8 | 100% |
| 解冻合约 | 8 | 8 | 100% |
| 历史调用信息 | 1 | 1 | 100% |
| 浏览-方法 | 11 | 8 | 73% |
| 浏览-交易 | 9 | 8 | 89% |
| 浏览-变量 | 9 | 8 | 89% |
| 小计 | 126 | 106 | 84% |

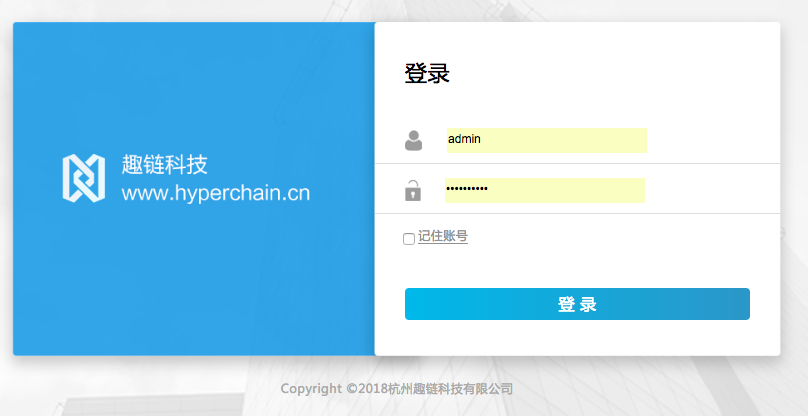
该模块也出现了较多缺陷，在修复完第一轮测试的问题后仍产生了新的问题，但经过二轮修复后，所有功能均通过回归测试。

系统成果展示

### 用户管理

登录

输入系统默认账号密码，登录成功。



**图5.1 用户登录**

修改密码

输入原密码和新密码，修改成功。



**图5.2 用户修改密码**

### 节点管理

创建分区

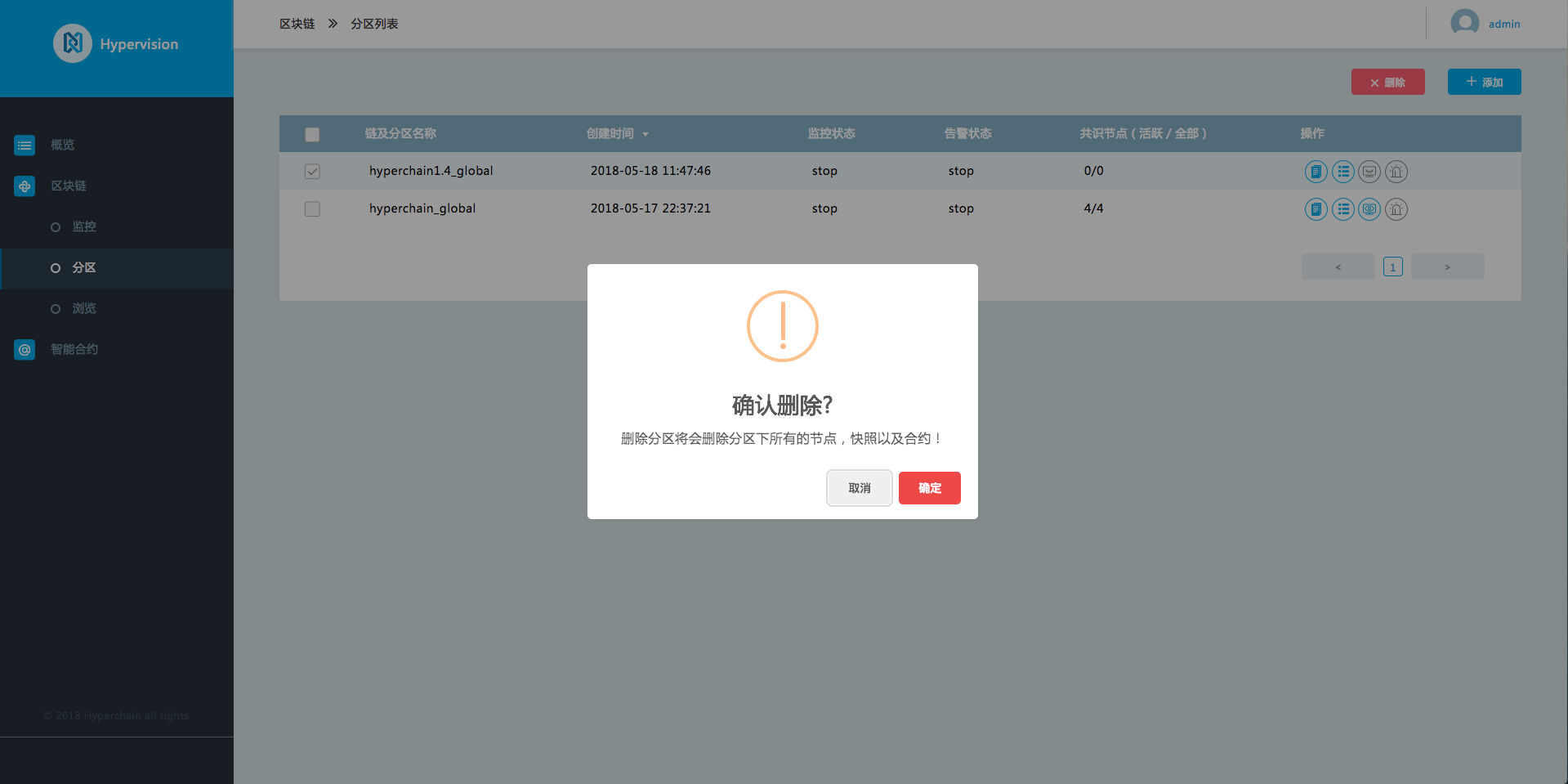
配置监控周期和监控阈值等参数进行分区创建。系统默认给出了一套配置参数，若不需要修改，可直接点击添加，完成区块链分区创建，创建后的区块链分区还不能正常进行监控，需要进入节点列表，添加相应的节点之后再进行后续监控相关操作。



**图5.3 创建分区参数配置**

删除分区

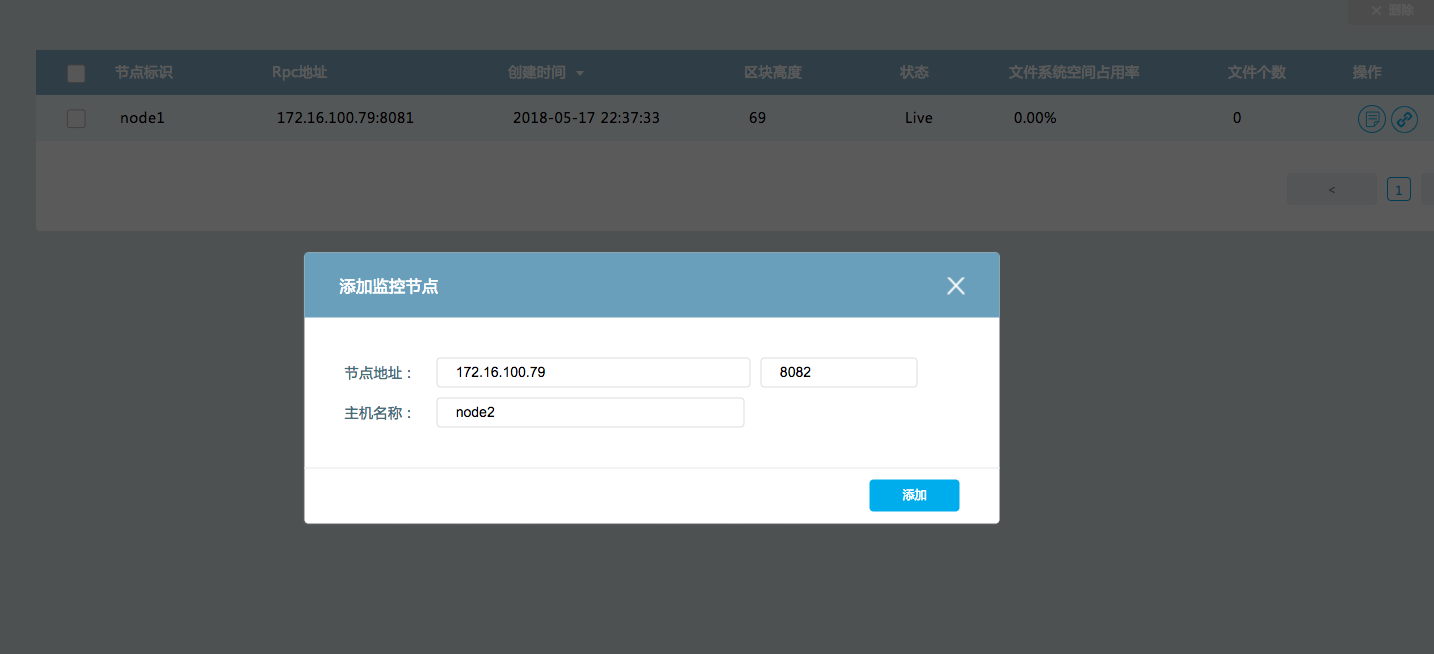
当想要删除区块链分区时，勾选对应链及分区名称的第一个勾选框，再点击接入点列表右上角的删除按钮，在弹出的确认框中点击确认，该区块链分区将会从监控平台中移除。



**图5.4 删除分区**

添加接入点

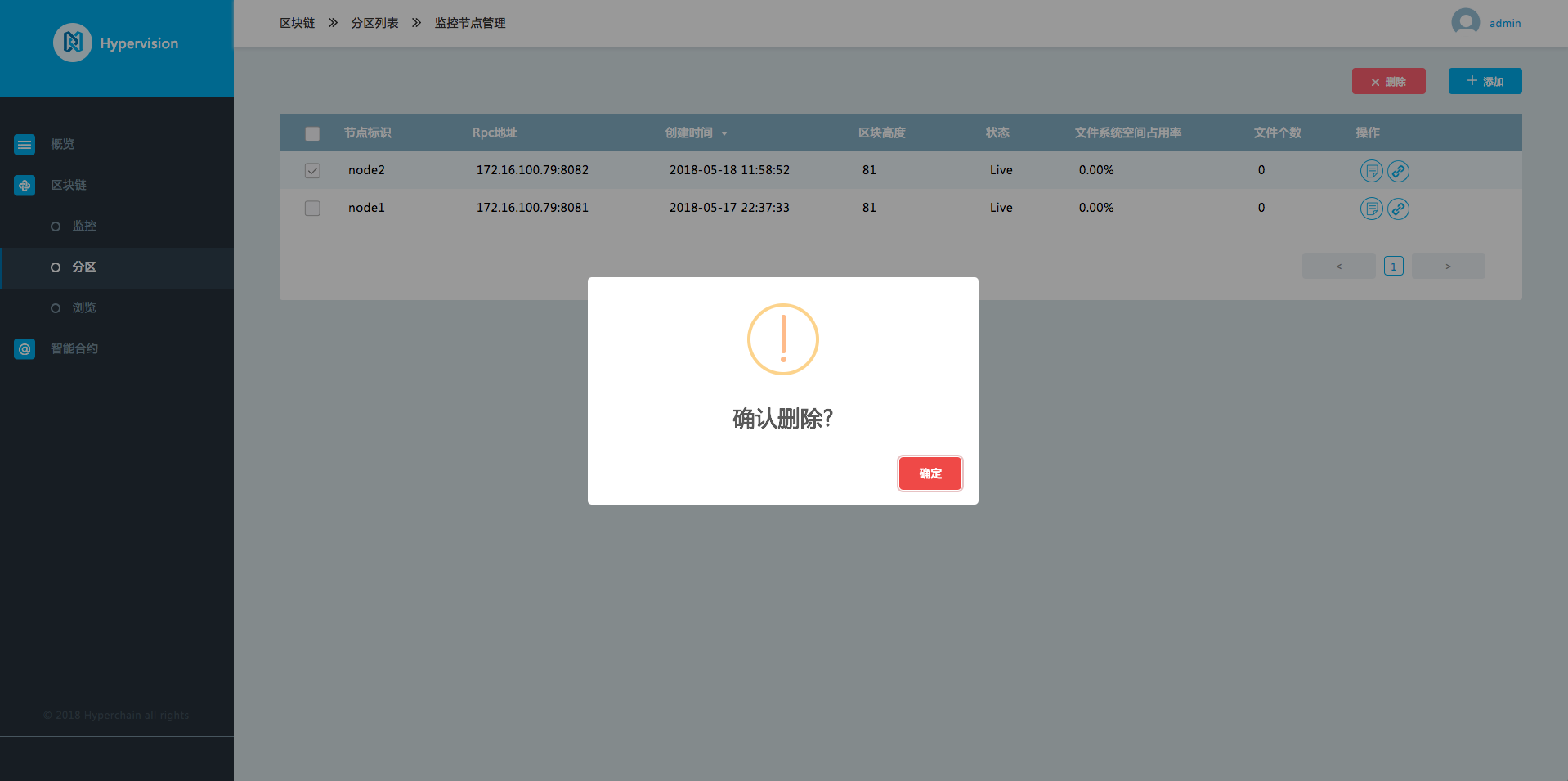
在弹出框中输入接入点的rpc地址和主机名称，主机名称用与当发生告警信息时，向监控中心通知当前信息是从哪个节点收集到的，对应统一监控系统接口中的hostname参数，若该节点已正确启动，点击部署将会在该链中新增一个节点。



**图5.5 添加区块链节点**

删除接入点

当想要删除节点时，勾选对应节点的第一个勾选框，再点击节点列表右上角的删除按钮，在弹出的确认框中点击确认，该节点将会从监控平台中移除，删除节点并不会影响实际的区块链节点的运行，仅表示不再对此节点进行监控。



**图5.5 删除区块链节点**

### 监控管理

监控设置

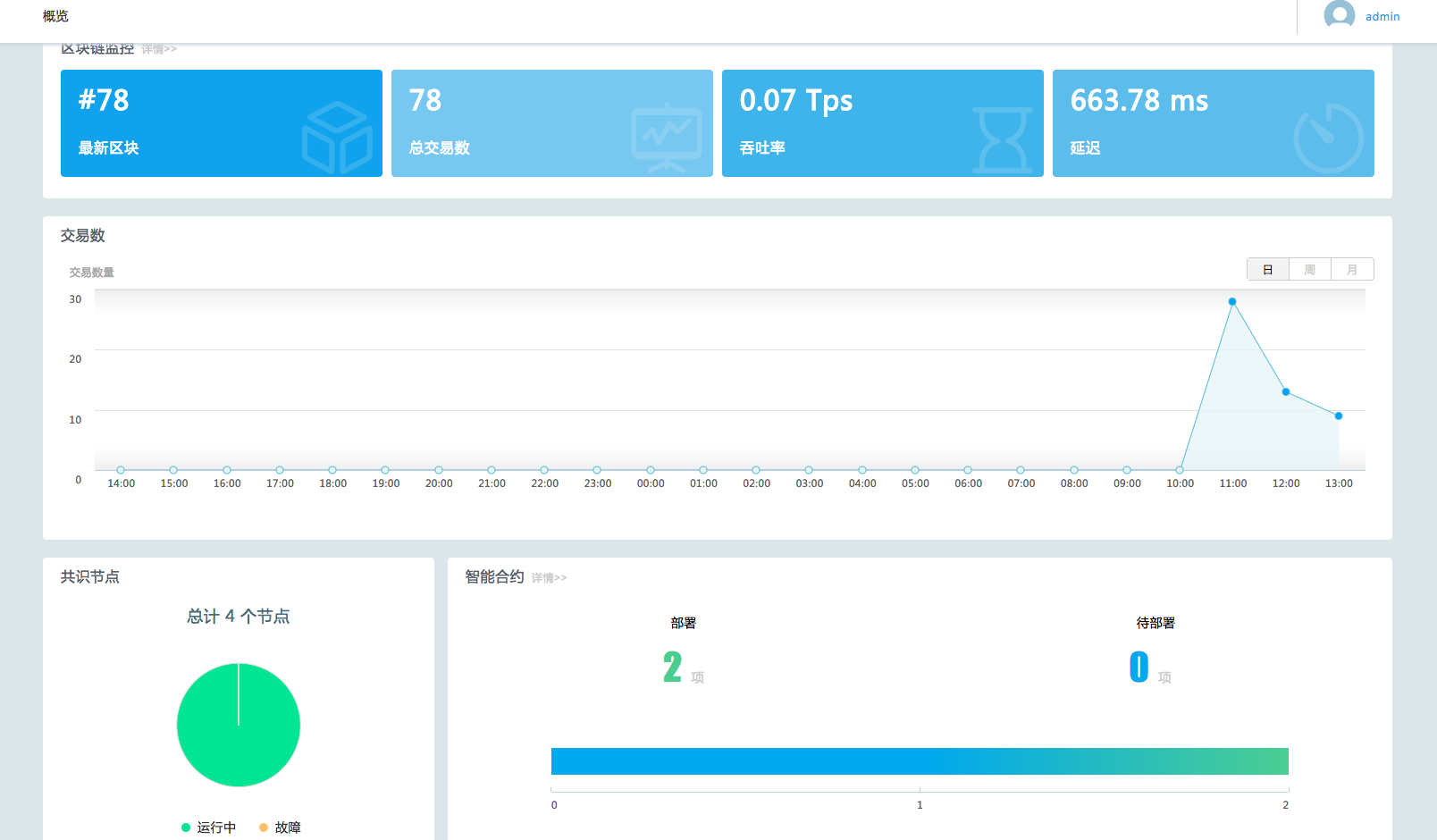
设置接入点并开启分区监控即开始进行监控。同时也可以对监控状态和报警状态进行暂停。当停止监控时，概览界面和区块链监控界面将不再有变化。当停止报警时，即使区块链相关指标超出阈值，也不会推送报警信息。



**图5.6 监控状态开启/关闭**

区块链数据概览

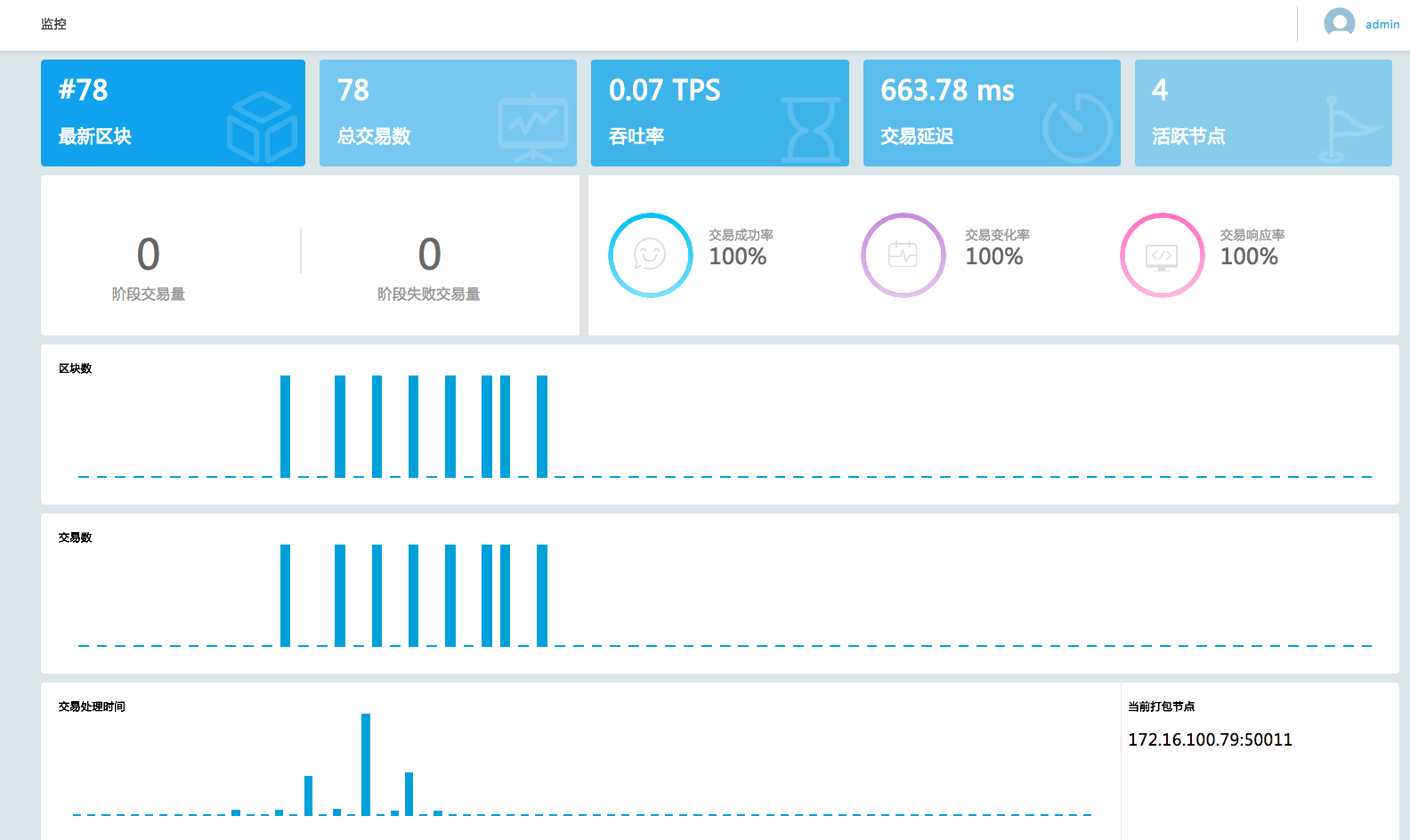
可以查看当前的最新区块、总交易数、吞吐率、平均延迟时间、日/周/月交易数变化情况、公示节点状态、智能合约统计等。



**图5.7 区块数据概览**

区块链监控详情

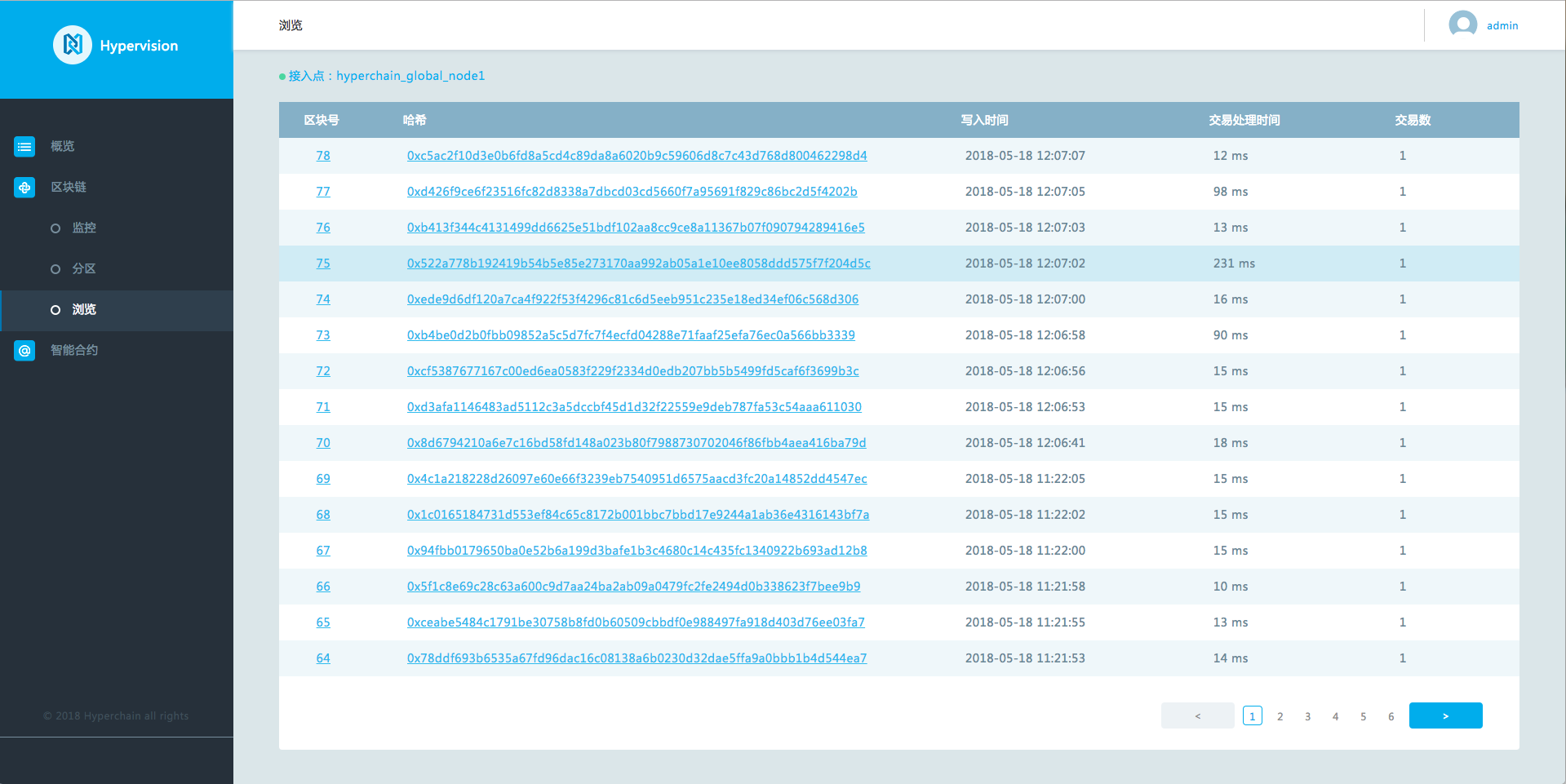
该页面除了显示与概览相同的信息，还能够动态显示最新的区块数、交易数、交易处理时间的变化情况。



**图5.8 区块数据监控**

区块浏览

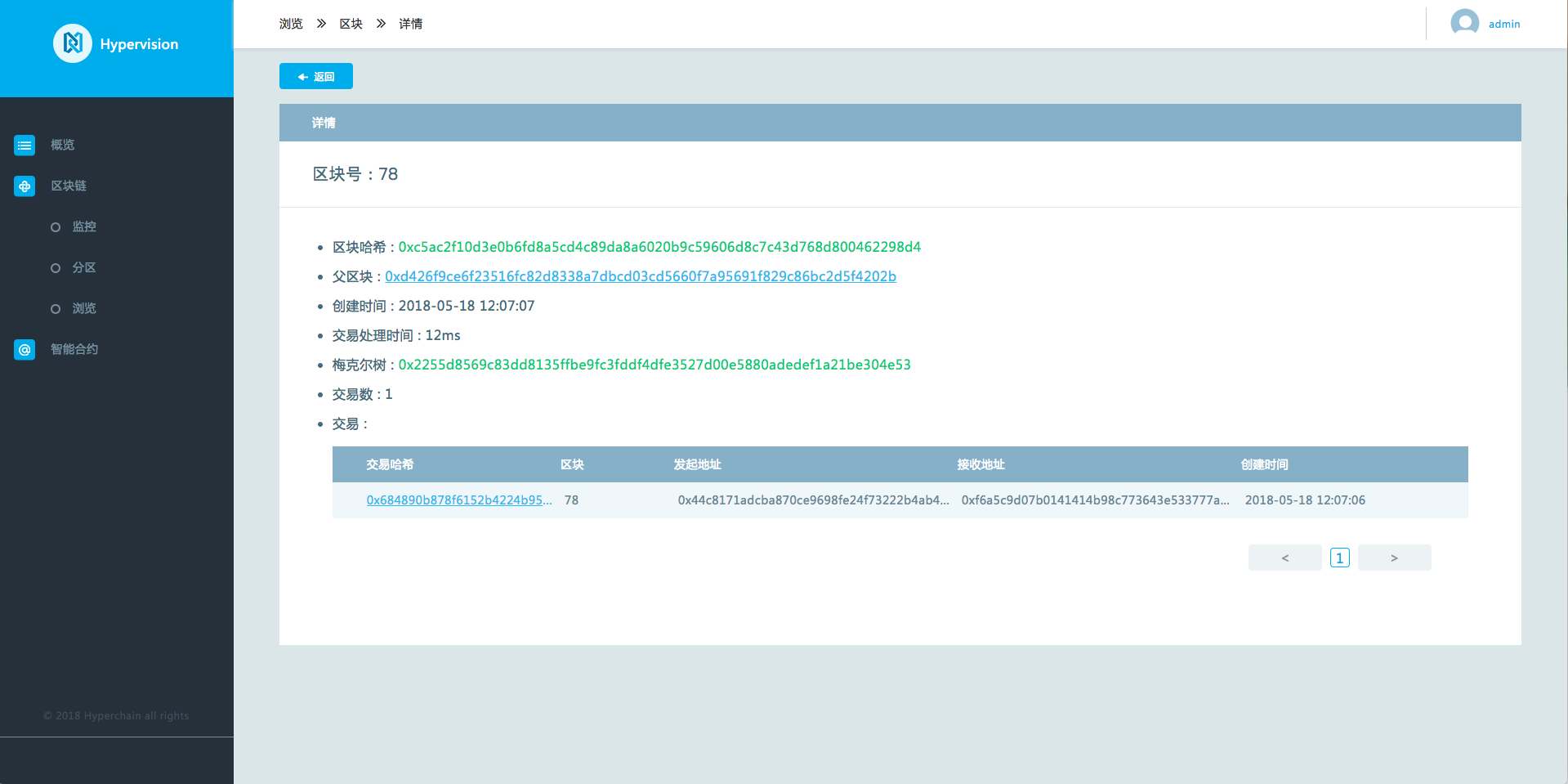
分页查看当前的区块信息。该页面展示了区块历史，包括区块编号、哈希、写入时间、交易处理时间和交易数等。



**图5.9 区块浏览**

区块详情

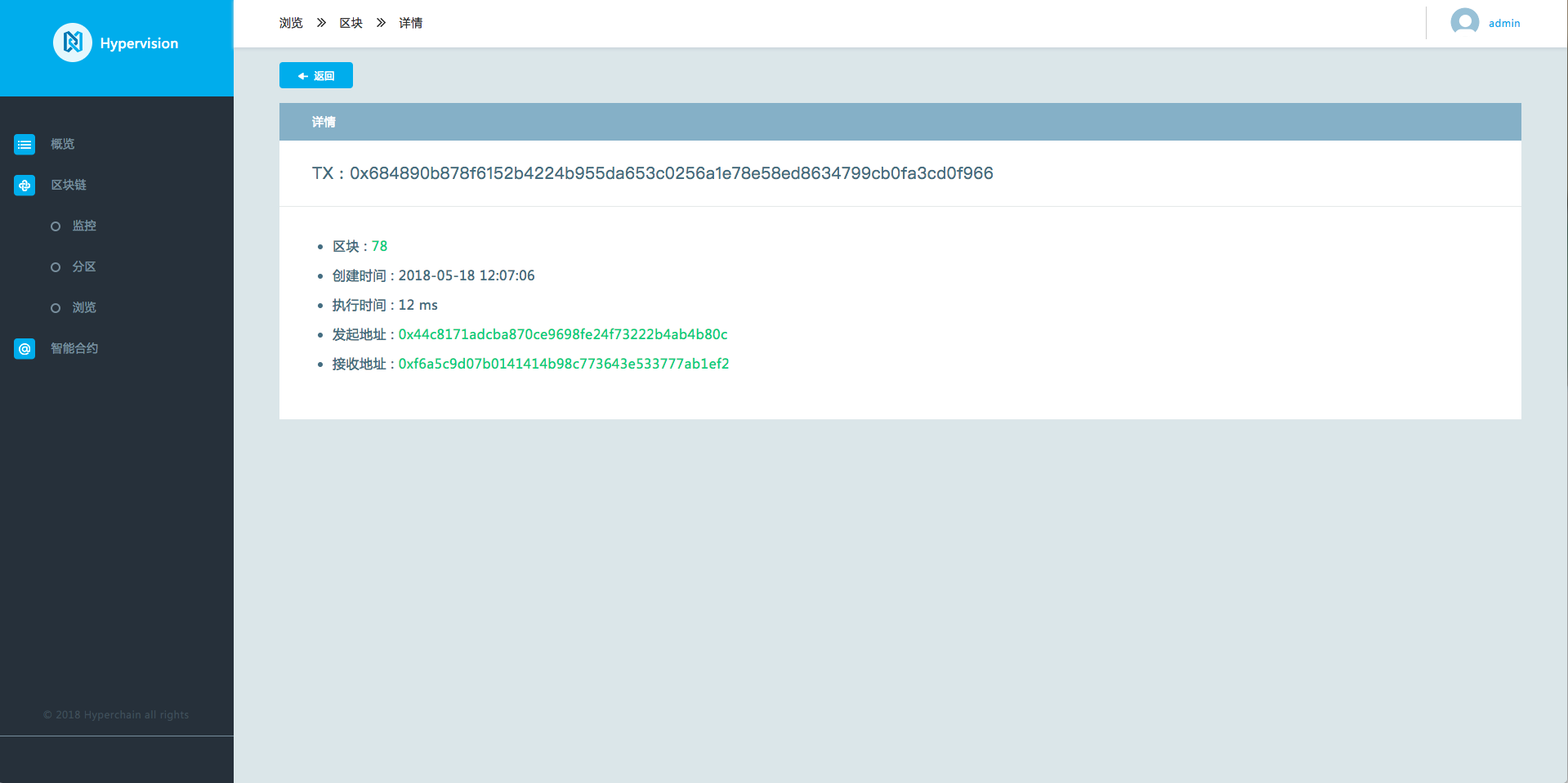
点击区块列表中的某一个区块，可以进入查看详情，详情页显示区块哈希、父区块哈希、区块创建时间、交易处理时间、梅克尔树hash值、总交易数以及产生的交易列表。



**图5.10 区块详情**

交易详情

进一步点击交易hash，可以查看该交易的创建时间以及发送方和接收方

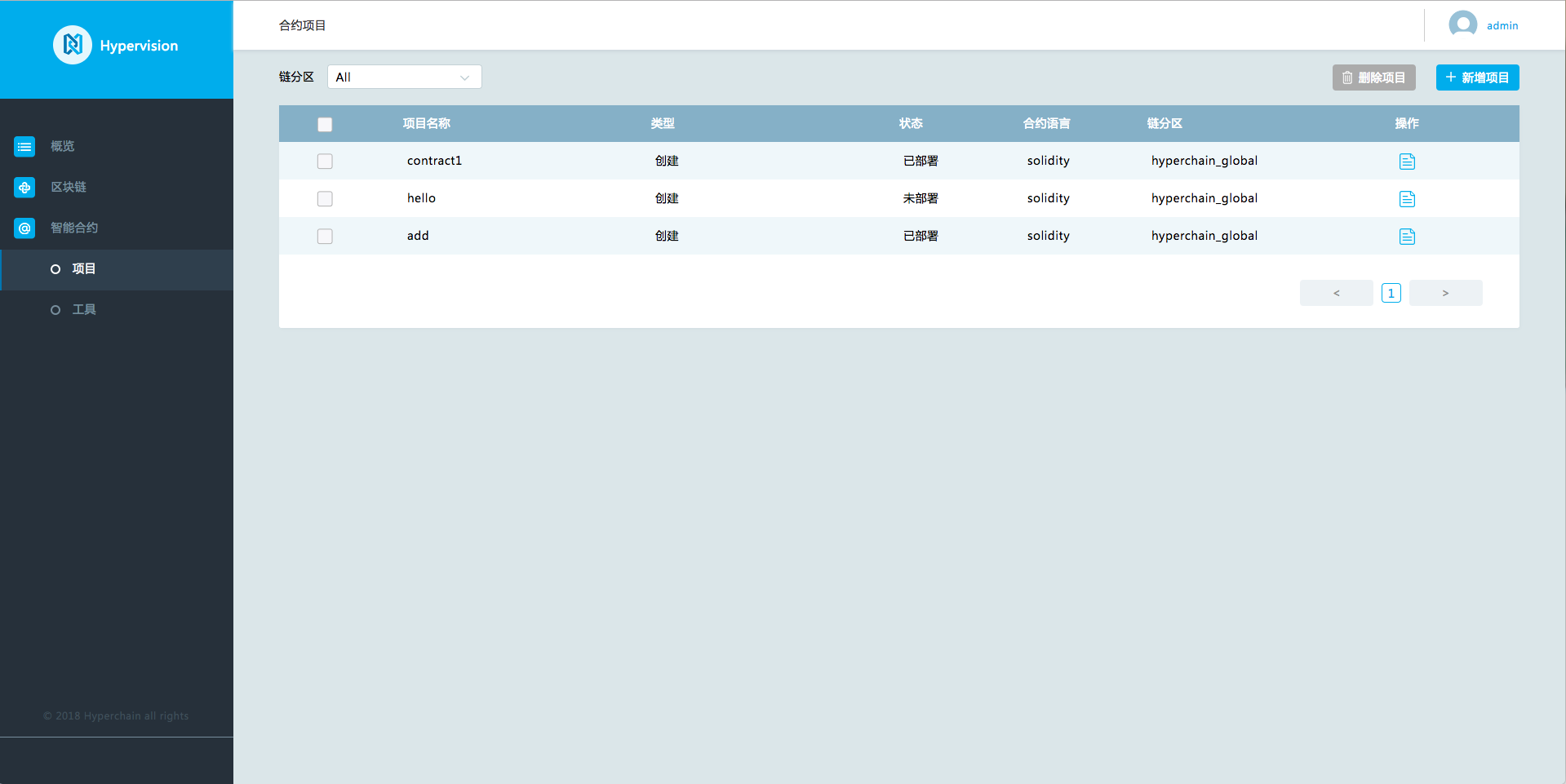


**图5.11 交易详情**

### 智能合约管理模块

合约列表

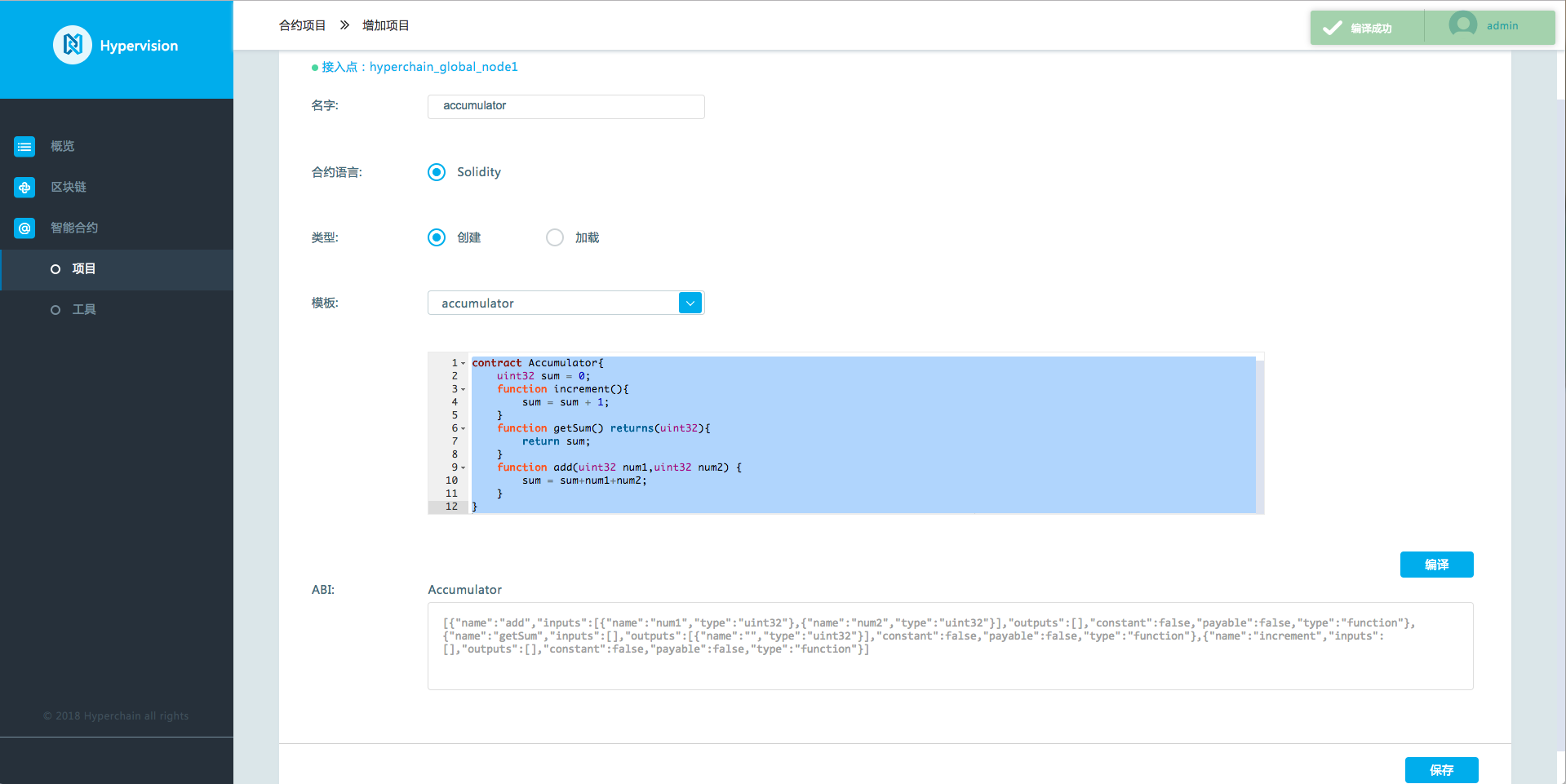
可查看已添加合约的合约名称，合约所在的项目名称，合约类型，合约状态，合约地址，以及可执行的相关操作。



**图5.12 智能合约列表**

新增项目

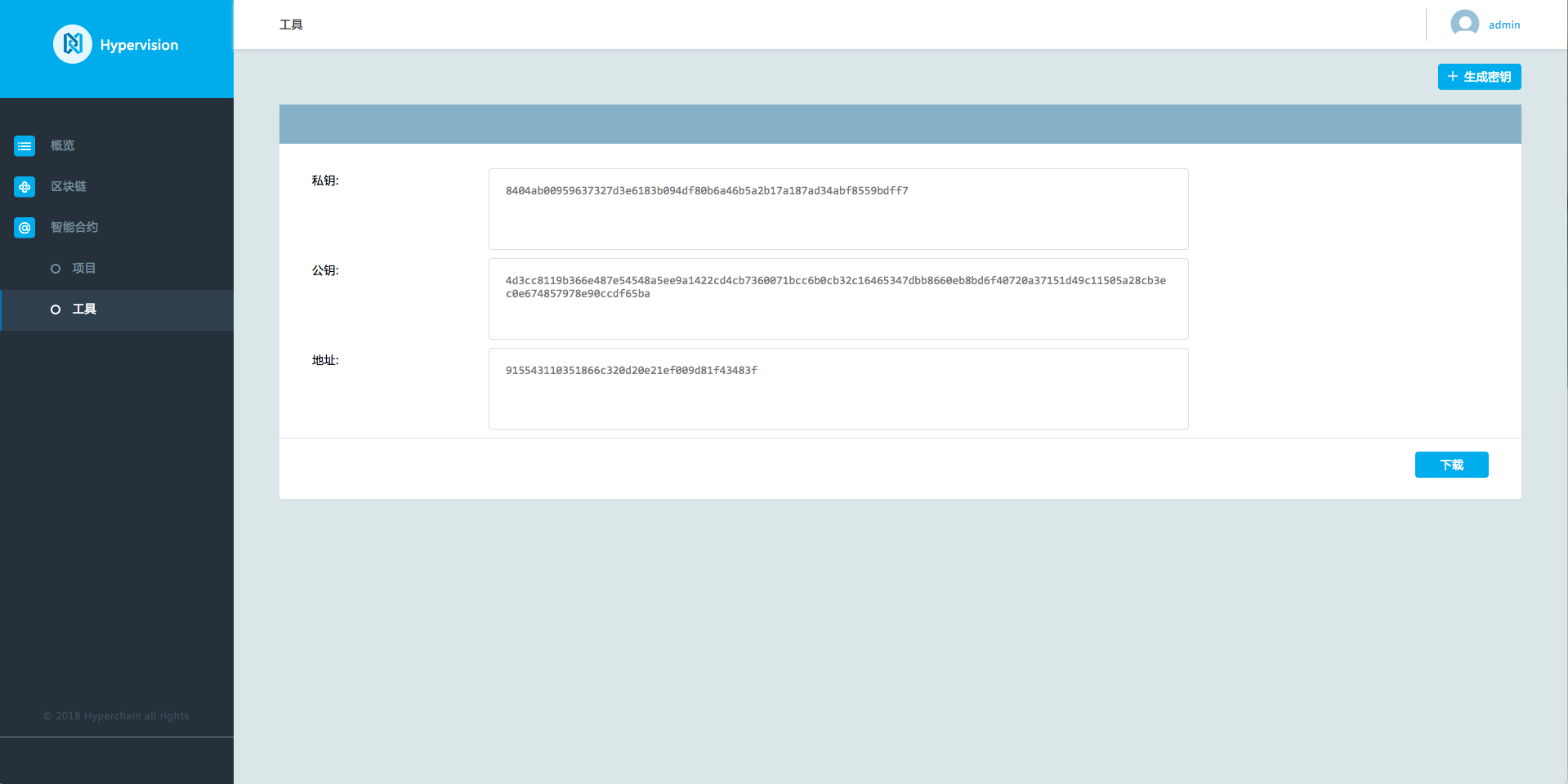
当用户想要新增新的合约项目时，用户可点击“新增项目”按钮，进入添加项目界面，输入项目名称，选择项目类型进行项目的新增。项目分为创建和加载两个类型。若为创建合约，则需要先进行编译，编译无误后点击保存即可创建成功。若为加载合约，则需要本地上传ABI文件，上传成功后会出现相应的ABI，点击保存创建成功。



**图5.13 添加智能合约**

密钥管理

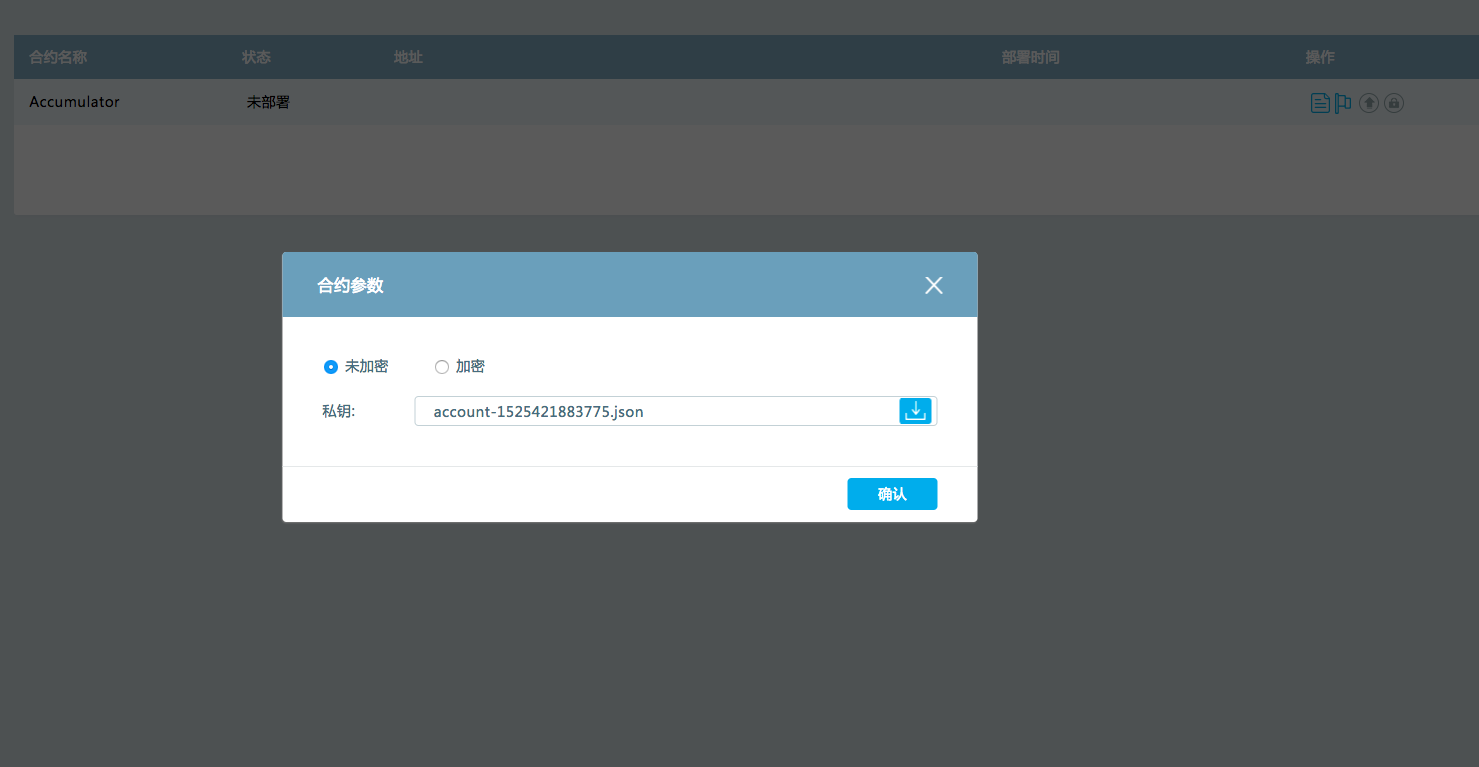
在智能合约的工具页面，点击右上角生成密钥按钮即可随机产生一个密钥，点击下载可以将密钥信息以json文件形式保存到本地。



**图5.14 密钥生成和下载**

合约部署

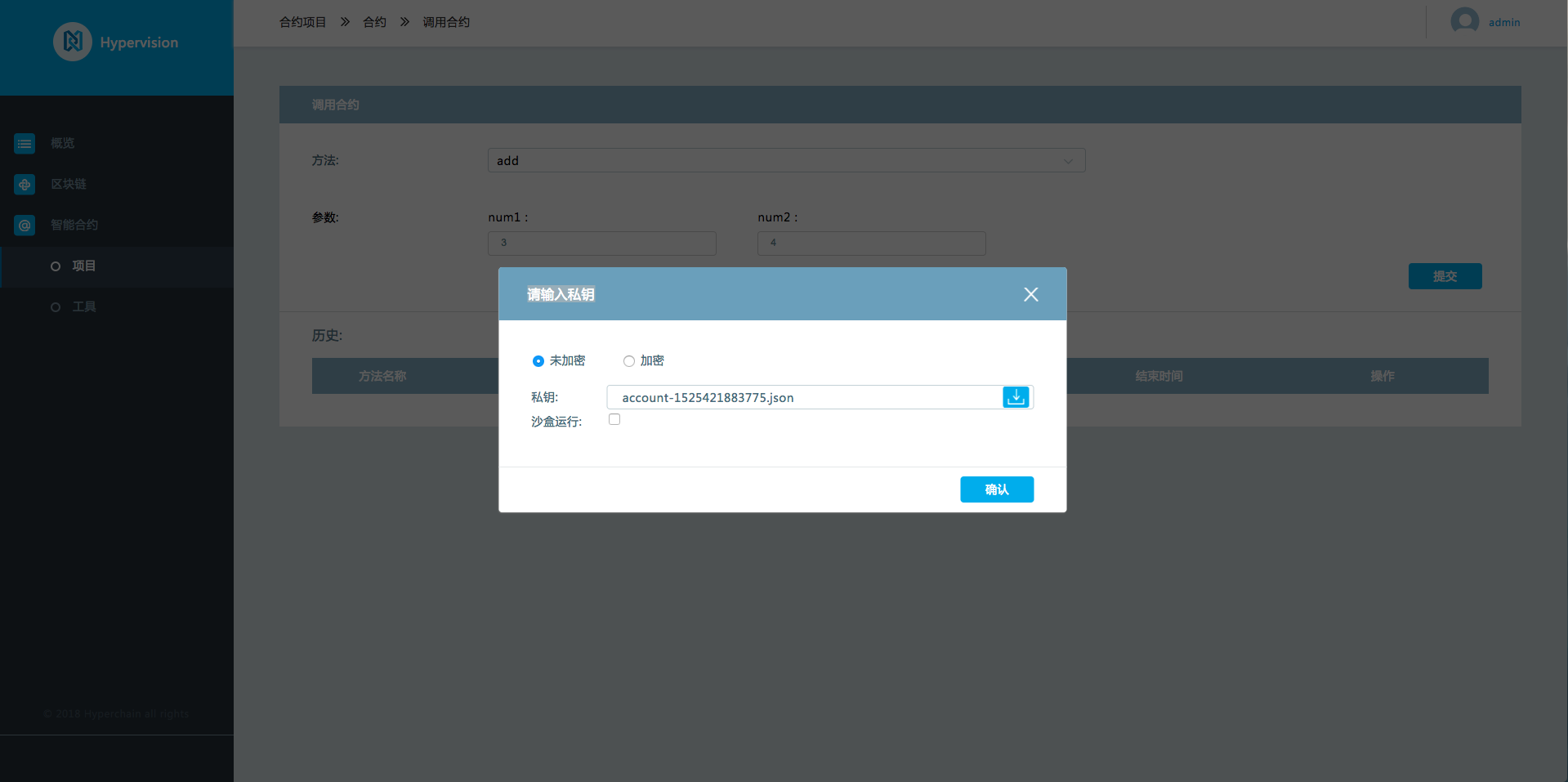
用户进入部署页面之后，根据合约的情况，用户需要输入指定的合约参数，以及私钥进行部署。输入正确之后，合约即可部署成功。



**图5.15 合约部署**

合约调用

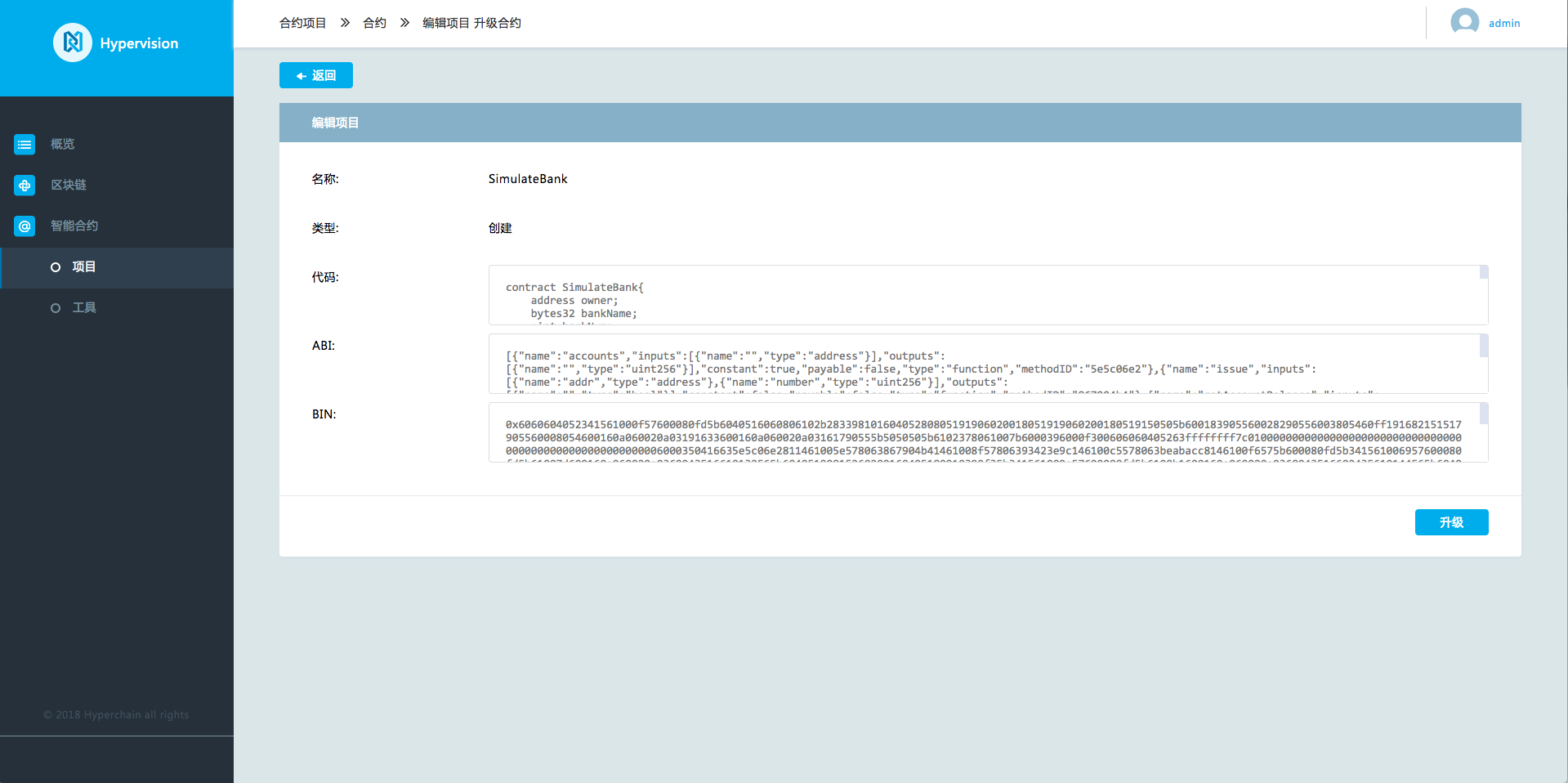
当部署完合约之后，该可约可进行调用。调用时可选择要调用的函数，并传入参数，上传私钥即可调用成功。为方便调用，系统会记住最近一次上传的私钥。这里用户还可以选择沙盒运行选项，勾选该选项来进行调用，调用将不会写入区块链。调用成功后可以在下方查看调用历史。



**图5.16 合约调用**

升级合约

当部署或者加载完合约之后，该合约可进行升级。点击升级按钮，跳转到升级合约页面，填写代码，ABI，BIN ，点击升级按钮，输入私钥，即可进行升级。



**图5.17 合约升级**

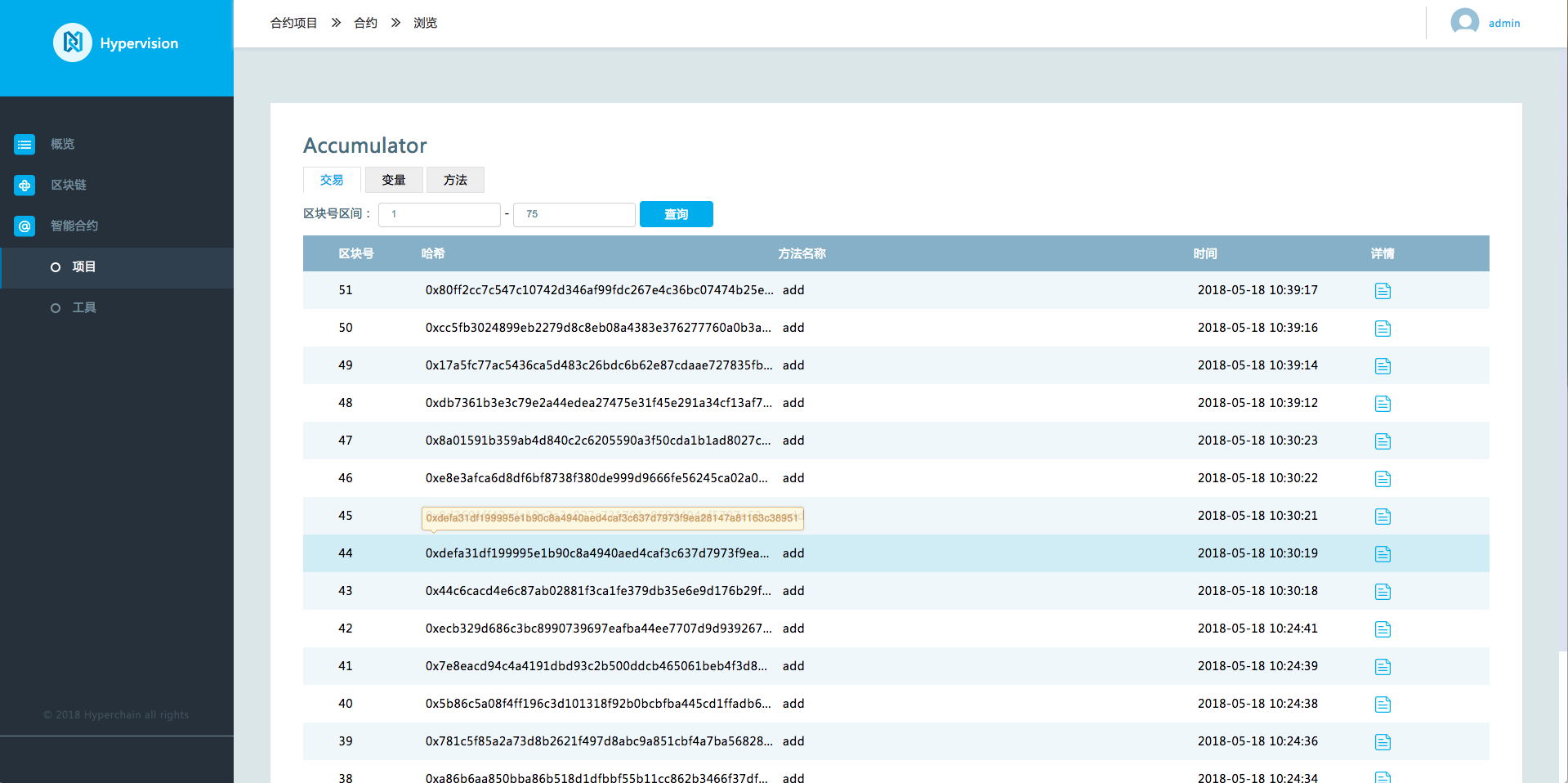
冻结/解冻合约

当部署或者加载完合约之后，该合约可进行冻结解冻。冻结合约之后，合约就不能进行升级，调用，浏览。解冻之后，可以恢复操作。

合约数据浏览

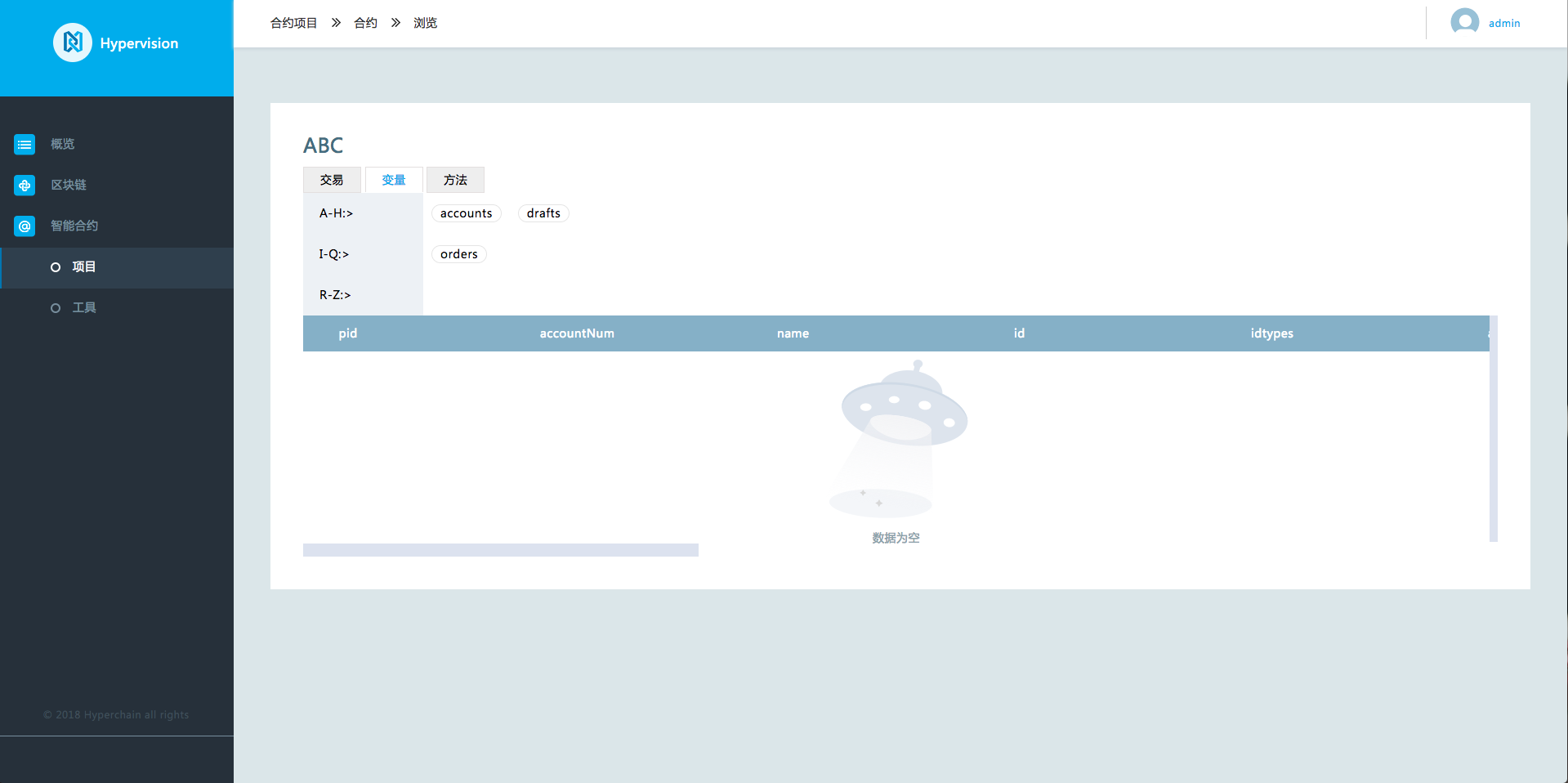
当部署或者加载完合约之后，该合约可进行浏览。浏览页面有3个选项卡，交易是显示该合约的在区块号区间下的所有交易。

交易选项卡：



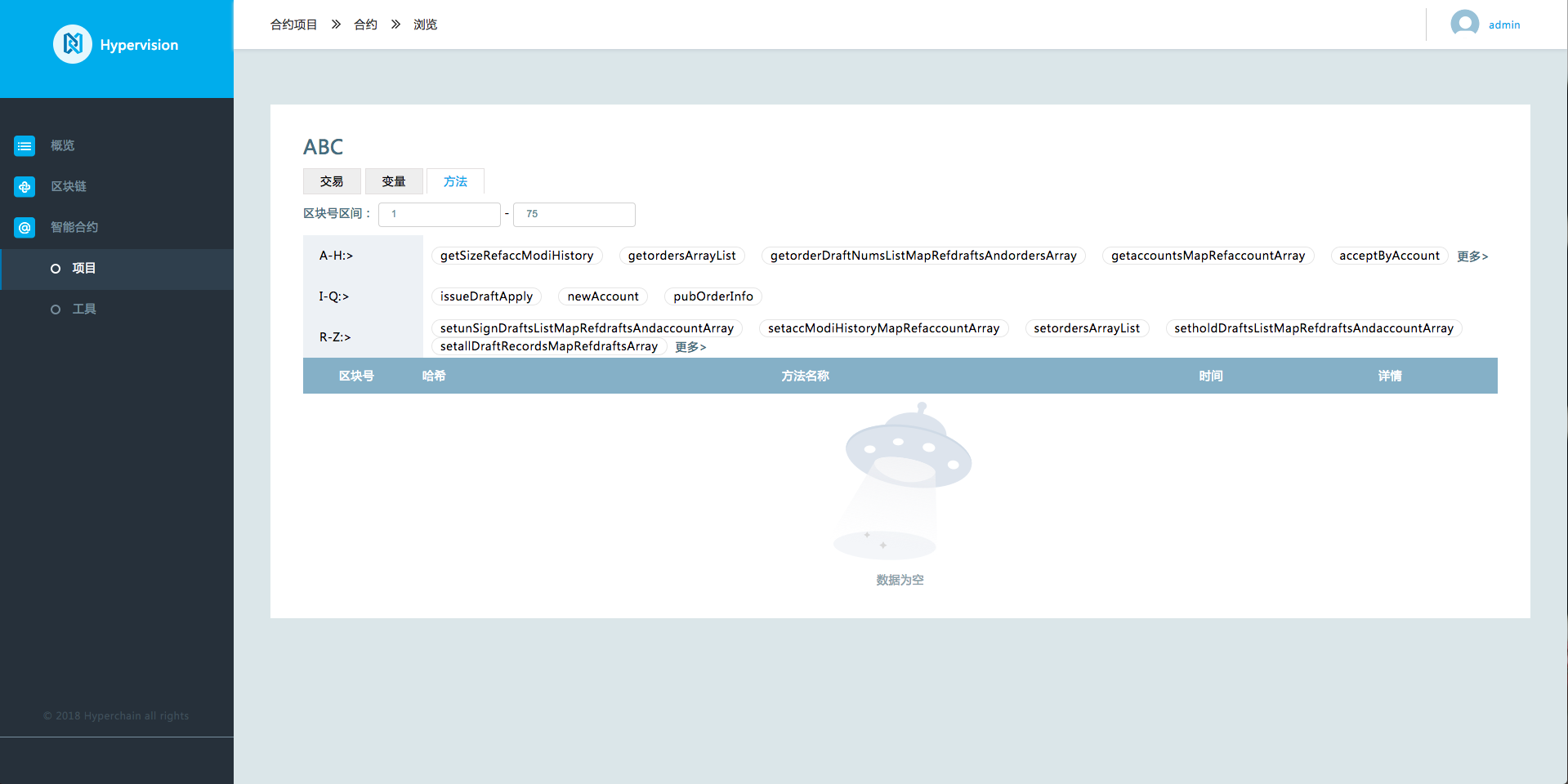
**图5.18 合约交易数据浏览**

变量选项卡：



**图5.19 合约变量数据浏览**

方法选项卡



**图5.20 合约方法数据浏览**

总结与展望

论文主要工作总结

基于当前区块链的实际需求和现状，本文提出了这样一个区块链运维和管理平台，一方面为了让区块链的使用者更直观地了解区块链的运行情况，另一方面也提供用户更便捷的方式来使用智能合约。

本文从需求和可行性进行了分析，了解真实用户对于该运维管理平台功能点的需求，从而从功能模块的角度设计了该平台。本文在技术层面采用了AngularJS和Beego两个高性能高可用的框架作为前后端架构，实现整个平台清晰的数据流。本文所有的设计都是基于趣链科技的底层区块链平台Hyperchain，因此当前系统也仅支持对Hyperchain区块链平台的运维和管理。得益于趣链科技已开发完成的SDK，该平台能够更方便地直接调用SDK来实现区块链数据采集，而不用重新封装rpc请求来与区块链交互。为了更准确和实时地获取区块链数据，本文还设计了一套后台实时获取数据的算法，同时为了满足用户报警的需求，本文也设计了两种模式的报警方案。

在目前版本的区块链运维和管理平台中，已经实现的功能如下：

1. 增删区块链分区，配置区块链节点，实现区块链节点监控切换。
2. 区块链数据实时监控并统计。
3. 区块链节点异常或数据异常报警。
4. 智能合约管理。

最后对系统的进行了功能性测试，说明系统的可用的，在实际生产环境的使用中也会证明该平台的价值。

展望

在未来的工作中，我们计划进一步改进区块链运维管理平台，可以提高的地方有以下几点：

1. 实现更多形式的报警，如支持短信报警、邮件报警等。
2. 支持多版本Hyperchain的监控，如同时支持Hyperchain1.4与1.2版本。
3. 支持其他类型平台的监控，如支持以太坊、fabric等的节点监控。
4. 加入CA体系，使平台调用区块链接口时需要安装证书，以提高平台安全性。
5. 优化UI界面，提升平台的用户体验。

此外，区块链的具体监控指标也需要进行完善，这项工作的实际生产环节中将会进行验证。

总的来说，该项目还有较大提升空间，作业也会在之后的工作中继续优化该项目。

参考文献

[1] NAKAMOTO S. Bitcoin: a peer-to-peer electronic cash system. https:/ /bitcoin.org/bitcoin.pdf. 2009

[2] 胡凯，白晓敏，高灵超，董爱强. 智能合约的形式化验证方法. 信息安全研究. 2016(12)

[3] BUTERIN V.Ethereum:A next generation smart contract and decentralized application platform. https://github.com/ethereum/wiki/wiki/White-Paper. 2013

[4] Massimo Bartoletti and Livio Pompianu. An empirical analysis of smart contracts: platforms, applications, and design patterns. 1st Workshop on Trusted Smart Contracts (WTSC17). 2017

[5] 谢辉. 区块链技术及其应用研究. 第31次全国计算机安全学术交流会论文集. 2016

[6] 王飞. 基于Docker的研发部署管理平台的设计与实现. 硕士学位论文. 2015

[7] 张玲，张翠肖. WebSocket服务器推送技术的研究. 河北省科学院学报. 2014,31(02):49-53

[8] 陈伟颖，李艳平，翟玥. 基于goroutine的web并发编程的研究与应用[J]. 电脑知识与技术. 2015(33)

[9] 金婧，陈家铨，王靖天，谢俊杰. 基于Go语言及Beego等开源技术的公务车辆管理系统的设计与实现[J]. 信息通信. 2015(02)

[10] Wu X,Xu Y,Shao Z, et al. LSM-trie:An LSM-tree-based Ultra-Large Key-Value Store for Small Data Items. Proceedings of 2015 USENIX Annual Technical Conference (ATC’’15) . 2015

[11] 于春娜,王晨升,杨光,郭世龙,刘丰. Web前端MVC框架的意义研究[J]. 产业与科技论坛. 2014(01)

[12] 宋朝辉. RESTful Web API模式下基于角色的权限管理设计与实现[J]. 科技创新导报. 2017(12)

[13] Kosba A,Miller A,Shi E,et al. Hawk:The Blockchain Model of Cryptography and Privacy-Preserving Smart Contracts. Security and Privacy . 2016

[14] 陈纯. 自主可控联盟链的技术难点和案例. 中国信息化周报. 2017(07)

[15] 饶毓琳. 基于LSM-Tree的持久化缓存机制的优化研究. 硕士学位论文. 2016

[15] Zhou Liqun, Li Zhihua. Application of block chain in supply chain finance[J]. China CIO News, 2016(7): 49-51.

[16] Künnapas . From Bitcoin to Smart Contracts: Legal Revolution or Evolution from the Perspective of de lege ferenda?. In The Future of Law and eTechnologies. 2016:111-131

[17] 杨晓晨,张明. 比特币:运行原理、典型特征与前景展望[J]. 金融评论.2014(01):43~53

[18] Mitchell ArnettAhmed KosbaAndrew MillerElaine Shi. Step by Step Towards Creating a Safe Smart Contract: Lessons and Insights from a Cryptocurrency Lab. Financial Cryptography and Data Security.2016:79-94

[19] María Teresa González-Aparicio, Adewole Ogunyadeka, Muhammad Younas, Javier Tuya, Rubén Casado. Transaction processing in consistency-aware user’s applications deployed on NoSQL databases. Human-centric Computing and Information Sciences. 2017(07):1-18

[20] 朱兴雄, 何清素, 郭善琪. 区块链技术在供应链金融中的应用. 中国流通经济. 2018(03)

致谢

值此论文完成之际，我要感谢我的老师，同学和朋友，在我的学习和生活中为我提供了源源不断的帮助和鼓励，在我论文的完成阶段对我的悉心指导。

在这里，我必须要感谢我的导师杨小虎老师，是他的引导让我能够接触到区块链这种新兴技术，让我打开了眼界；同样，也是他的推荐，让我得以进入杭州趣链科技有限公司实习，在那里真正接触到了区块链技术的开发和实践，也是我整个毕业设计的源泉。同样也感谢杨小虎老师在我论文完成阶段对我的指导和督促，我才能顺利完成毕业设计。

此外我还要感谢我的校外导师钟蔚蔚学长和邱伟伟博士，是他们在项目进行过程中为我提供了源源不断的指导。在他们的帮助下我才能攻克一个又一个的难题，才能顺利将想法变成了现实。

感谢为区块链运维和管理平台付出过的所有人，是我们共同的努力才能有如此的效果。感谢我的家人在生活上和思想上对我的激励和支持。

最后，再次感谢我的老师、同学和朋友在这一年里对我的帮助和支持，也感谢审核论文的各位老师，在百忙之中抽出时间给予我指导。

金鹏

2018年5月

**本科生毕业论文（设计）任务书**

**一、题目 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**二、指导教师对毕业论文（设计）的进度安排及任务要求**

**此部分，原则上打印。**

**起讫日期 年 月 日 至 年 月 日**

**指导教师**（**签名） 职称**

**三、学院审核意见**

**正式上交的那本论文（正本），请导师手写签名。**

**此处留空，由学院统一处理。**

**负责人**（**签名）**

**年 月 日**

**毕 业 论 文（设计） 考 核**

**一、指导教师对毕业论文（设计）的评语**

**导师评语部分，可手写，可打印。**

**正式上交的那本论文（正本），导师务必手写签名，并写上日期。**

**指导教师(签名）**

**年 月 日**

**二、答辩小组对毕业论文（设计）的答辩评语及总评成绩：**

**答辩评语部分，答辩日当场手写。**

**答辩得分，直接填写0-60分之间的一个整数。**

**这三项请从毕设信息系统中查阅，和前期上交的开题报告保持一致，分数请手写。**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **成绩比例** | **开题报告**  **占（20%）** | **外文翻译**  **占（10%）** | **文献综述**  **占（10%）** | **毕业论文（设计）**  **质量及答辩占（60%）** | **总成绩** |
| **分值** |  |  |  |  |  |

**答辩小组负责人（签名）**

**对于毕业设计，请将“文献综述”替换成“中期检查”。**

**年 月 日**

**正式上交的那本论文（正本），答辩小组组长务必手写签名，并写上答辩当日的日期。**