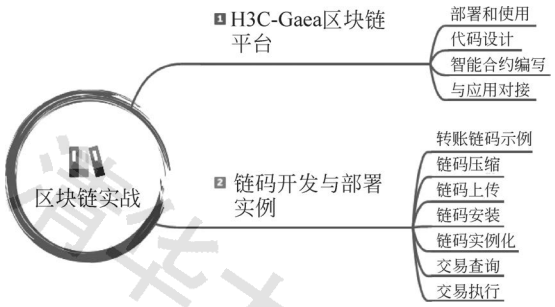
第8章 区块链实战

本章思维导图



前7章分别介绍了区块链的基础知识、进化过程、应用领域，本章着重介绍区块链的开发。

从开发角度讲，只需要关注两方面：区块链底层平台（公链中的以太坊或联盟链）和区块链应用开发（这里的应用开发就是指所谓的智能合约）。

本章只关注联盟链，其中新华三集团推出的H3C Gaea区块链平台是较有代表性的一个BaaS平台。

Gaea区块链平台除了具有BaaS的基本按需部署、快速构建区块链即服务平台、管理区块链资源等功能外，还开发了一系列便利的功能。目前最新发布的Gaea区块链平台1.0版本具有如下特性：

- 支持单机或者K8S集群环境运行；
- 支持网络规模、组织规模、通道规模的可设置、可动态调整；
- 支持状态数据库类型LevelDB和CouchDB；
- 支持Go、Java、NodeJS多种语言链码；
- 支持平台与区块链节点间通信SSL加密；
- 支持RAFT、solo共识；
- 支持各个组织的user-dashboard隔离；
- 支持链码模拟执行和链码升级；
- 支持基于CA的用户管理。

总体而言，Gaea区块链平台以保证数据安全为基础，以简化流程、提高多方协同效率为目标，将节省运营成本作为其关键价值。目前Gaea区块链平台已将底层开源，不仅能够使众多开发者享有更低的开发门槛与更公平的规则，同时也体现了接受所有开发者审阅的强大自信。

8.1 H3CGaea 区块链平台

8.8.1 部署和使用

1. 系统部署

1）硬件要求

Gaea 平台使用的服务器为 Ubuntu 系统 16.04 版本。所需服务器的数量与具体需要部署的区块链规模相关，建议的最低配置如表 8-1 所示。

表 8-1 服务器硬件要求

设备类型	规　格
Ubuntu 服务器	基本配置不低于：Intel XeonE54 核 CPU，32GB DDR4 内存，1TB 高速硬盘，1＋1 冗余电源

Gaea 平台为商业软件，需要 License 授权后才可用。

如果部署 K8S 网络，一般为一个 Master 主机，多个 Node 主机、主机数目与需要部署的区块链规模相关。如果部署单机网络，则只需要一台服务器。

部署 Gaea 时建议部署在一台单独的主机上，如果设备有限，也可以与 License 或者其他网络主机部署在一起。

区块链的主要特性为数据存储分布式，生产环境一般不采用单机网络（SOLO）模式，仅在实验或者学习区块链网络时采用。生产环境一般采用 K8S（Kubernetes）网络模式。

2）软件安装

Gaea 服务器是指最后部署、运行 Gaea 节点并存储账本的服务器。当采用 K8S 网络模式时，需要先部署好 K8S 环境，如表 8-2 所示。

表 8-2 K8S 环境部署要求

服务器	基础软件	建议版本	
宿主服务器	Docker-compose	1.21.0	
	Docker	18.03.1-ce	
	make	GNUMake 4.1	
Gaea 服务器	nfs-common	apt-getinstallnfs-common 即可	用于创建单机模式区块链网络
	K8S 相关软件	1.15（含）以上	用于创建 K8S 集群模式区块链网络
	keepalived 配置	最新版本即可	

3）部署容器

Gaea 系统采用容器方式部署，易部署，易升级，相互之间互不干扰。需要部署的容器镜像如表 8-3 所示（以当前最新版本为例）。

表 8-3　需部署的容器镜像及版本

服务器	镜像	版本
宿主服务器	h3c/Gaea-operator-dashboard	最新
	h3c/Gaea-user-dashboard	最新
	h3c/oauth2-server-sso	0.0.4
	mongo	3.4.10
	nginx	1.0.0
	h3c/blockchaindata	1.0
	itsthenetwork/nfs-server-alpine	9
Gaea 服务器	hyperledger/Fabric-CA	1.4.2
	hyperledger/Fabric-Orderer	1.4.2
	hyperledger/Fabric-Peer	1.4.2
	hyperledger/Fabric-ccenv	最新
	hyperledger/Fabric-couchdb	2.1.1
	hyperledger/Fabric-baseos	amd64-0.4.14
	bobrik/socat(单机模式下需要)	最新

4）一键启动

将获取到的 Gaea 区块链平台安装部署压缩包，解压到宿主服务器中。进入解压后的目录，可以看到 master、images、gaea-deploy、sso 和 tools 几个子目录。

进入 gaea-deploy 目录，执行 bash offline_deploy.sh 命令。offline_deploy.sh 脚本将会在宿主机服务器中离线安装 make、gcc、docker、docker-compose 等工具，并生成平台所需的公私钥证书文件(需要根据窗口提示按回车键)。

此时在 gaea-deploy 目录中执行 make start 命令，即可启动 Gaea 区块链部署平台。

5）预制操作

(1) 如果创建单机模式的区块链网络，需要执行以下操作。

首先，需要在 Gaea 网络所在的 Docker 主机上启动 2375 监听端口，配置的服务地址为：主机 IP：2375。启动 2375 端口的命令如下。

当 Docker 与 Gaea 部署于同一主机时，执行：

```
Dockerrun - d - v/var/run/Docker.sock:/var/run/Docker.sock - p
0.0.0.0:2375:2375bobrik/socatTCP - LISTEN:2375,forkUNIX - CONNECT:/var/run /Docker.sock
```

当 Docker 与 Gaea 部署于不同主机时，执行：

```
Dockerd - Htcp://0.0.0.0:2375 - Hunix:///var/run/Docker.sock -- api - cors - header = ' * ' --
default - ulimit = nofile = 8192:16384 -- default - ulimit = nproc = 8192:16384 - D&
```

然后，到 Worker 目录下执行 setup.sh，在此脚本中挂载 NFS 目录。

(2) 将区块链镜像文件部署到 Gaea 服务器中。

首先，将步骤 4)“一键启动”中解压后的区块链安装部署文件中的 images 目录复制到 Gaea 服务器中(Gaea 服务器用于运行区块链网络节点)。如果区块链网络节点与 Gaea 平台部署在同一台主机中就不需要复制。

然后，进入 images 目录，执行 bash loadall.sh，loadall.sh 脚本将会把当前目录中的镜

像文件加载到执行该命令的服务器中，创建区块链网络时需要用到这些镜像文件。

（3）如果创建 K8S 集群模式的区块链网络，需要执行以下操作。

首先，确保 Gaea 平台与 K8S 集群各主机时间同步，然后直接进入平台操作。

其次，安装好 K8S 集群环境。

最后，安装 Keepalive。

2. 平台操作

新华三 Gaea 区块链平台有两个后台服务，分别是区块链网络管理平台（监听 8071 端口）和区块链业务管理平台（监听 8081 端口）。

读者们在购买本书后会得到 Gaea 系统的 48 小时免费试用权限，学习完本章后可以验证实验并得到区块链平台链码部署的机会，具体说明见前言。

1）区块链网络部署

进入 Gaea 区块链平台【区块链网络管理】页面，如图 8-1 所示。

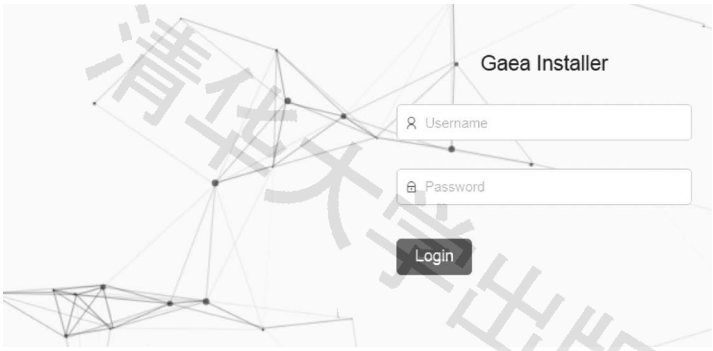


图 8-1　网络管理平台登录

（1）添加 host 主机。

在 Gaea 网络管理平台的【主机管理】页面中，可以进行添加主机的操作，如图 8-2 所示。

图 8-2　主机管理-添加

① 如果创建单机模式的区块链网络，这里选择 DOCKER，如图 8-3 所示。

图 8-3　创建主机-DOCKER

主机创建成功后，如图 8-4 虚线中所示，为 Active 状态。

图 8-4　主机状态

② 如果创建 K8S 集群模式的区块链网络，“主机类型”选择 KUBERNETES。

如图 8-5 所示，可以看到此时需要填写 K8S 环境的证书、私钥等相关内容。服务地址填写 Master 的 IP 及 kube-apiserver 服务监听的端口号（一般默认是 6443）。虚拟 IP 地址是指在 K8S 环境中配置 keepalived 服务时配置的虚 IP 地址。

选择凭证类型为 cert _ key 时，粘贴 kube-apiserver 任务中的证书内容和 CertificateKey。

选择凭证类型为 config 时，粘贴 root/. kube/config 中的配置内容。

如果使用 SSL 验证，则通过 kube-apiserver 任务中的 ca. cert 获取 SSL CA 证书。

（2）创建 Orderer 组织和 Peer 组织。

在 Gaea 网络管理平台的【组织管理】页面中，单击【添加】按钮，如图 8-6 所示。

“选择类型”这里可以选择 peer 或 orderer。

如果创建 peer 类型的组织，除了填写组织名和 domain（域名），还需要填写 peer 数（平台将根据这个 peer 数创建相同数量的容器）。同时在“主机”下拉列表框中选择该组织节点将要部署在哪个主机上，如图 8-7 所示。

填写好后单击【提交】按钮即可。在创建 peer 组织的同时，平台会通过 Fabric-CA 自动

图 8-5　创建主机-KUBERNETES

图 8-6　组织管理-添加

为该组织注册并创建默认名称为 Admin@"组织名称"."组织 domain"的用户信息。如图 8-8 所示，在【用户管理】页面可以看到所有用户的信息。在登录"区块链业务平台"的时候需要用该用户名登录(默认密码为 666666)。

如果选择 orderer 类型，除了填写组织名和 domain(域名)，还需要添加 orderer 主机名称。可以填写多个主机名，以回车键隔开，每个 orderer 主机对应一个 orderer 容器)，同时需要在"主机"下拉列表框中选择主机，如图 8-9 所示。

图 8-7　创建 peer 组织

图 8-8　用户管理

图 8-9　创建 orderer 组织

如图 8-10 所示，可以在【组织管理】页面中看到所有已创建完成的 peer 组织和 orderer 组织。

图 8-10　组织管理列表

（3）新建网络。

在 Gaea 网络管理平台的【网络管理】页面中，单击【新建网络】按钮，如图 8-11 所示。

图 8-11　【网络管理】页面

此时进入【新建网络】页面，如图 8-12 所示。

Fabric 网络支持多种共识算法（Solo、Kafka、etcdraft、PBFT），其中 solo 一般为单机模式部署时使用，K8S 模式部署时建议选择 etcdraft 或 PBFT。选择 Kafka 模式会另外启动 Zoo 和 Kafka 的容器来执行共识算法，有需要时可以选择使用。

数据库类型可以选择 leveldb 或 couchdb。当选择 couchdb 类型时，创建网络时会另外启动 couchdb 的容器，只有 couchdb 支持富查询。

在图 8-12 所示的页面填写网络名称、选择 Fabric 版本后，选择之前已创建的组织和主机，单击【提交】按钮即可。稍等几秒就会在该组织对应的主机上看到运行的 Fabric 节点的容器。

如图 8-13 所示为创建成功后的页面，显示 mynet 网络处于 runing 状态。

单击【详情】可以看到该网络的详细信息，如图 8-14 所示。

图 8-12　网络管理-新建网络

图 8-13　网络管理-列表

图 8-14　网络管理-详情

2）区块链业务部署

登录 Gaea 区块链平台的【区块链业务管理】页面。

可在 Gaea 网络管理平台的【用户管理】页面中查看用户名（如图 8-8 所示），选择以 Admin 开头的组织用户名，初始密码为 666666。这里必须选择以 Admin 开头的组织管理员用户。

图 8-15 为 Gaea 业务管理平台登录页面。

图 8-15 区块链业务管理登录页面

（1）创建通道并添加节点。

在 Gaea 业务管理平台的【通道管理】页面中，单击【创建通道】按钮，如图 8-16 所示。

图 8-16 通道管理

单击【创建通道】按钮后，需要填写通道名称，并选择排序组织和节点组织，如图 8-17 所示。在这里可以选择刚创建的 orderer 组织和 peer 组织。

创建通道后，该通道内的节点数为 0（如图 8-18 所示），即该通道内并未加入任何组织的节点。接着需要将节点添加到通道内。

图 8-17　通道管理-创建通道

图 8-18　通道管理-列表

如图 8-19 所示，单击【添加节点】。

图 8-19　通道管理-添加节点

进入如图 8-20 所示的页面，在这里选择安装链码的节点，勾选节点名称前面的复选框即为选中。在“角色”下拉列表框中可选择该节点充当的角色。chaincodeQuery 表示节点可以调用链码查询，endorsingPeer 表示节点可以作为背书节点，ledgerQuery 表示允许查询

该节点账本。

图 8-20　通道管理-添加节点

创建完毕后，就会看到我们创建的通道 mychannel，它属于网络 mynet。通道 mychannel 中包含了两个组织，而当前的登录用户所属的组织中有两个 peer 节点（如图 8-21 所示）。通道中如果有其他组织，则需要登录其他组织的管理员账户，将其节点加入通道中。

图 8-21　通道管理-列表

（2）上传链码。

在 Gaea 业务管理平台的【链码管理】页面中，单击【上传链码】按钮，如图 8-22 所示。

进入【上传链码】页面（如图 8-23 所示）后可以看到，【语言选择】默认为 golang，也可以选择 Node 或 Java。根据选择的语言不同，实例化链码时需要的镜像容器也不同。

如果选择 golang 语言，需要 ccenv 的 images 文件；如果选择 Java 语言，则需要 Javaenv 的 images 文件；如果选择 Node 语言，则 peer 容器会根据 Node 链码中的 package.json 文件在线安装 node_modules 包（要求 peer 容器能从 npm 源在线下载依赖包）。

【MD5 值】是指该页面中选择上传的链码文件的 MD5 值，可以用 MD5 工具生成，或者通过一些网站在线获取文件的 MD5 值。

上传的链码要求是将文件夹压缩为.zip 格式的文件，文件夹内直接放链码文件，不要再

图 8-22　链码管理

图 8-23　链码管理-上传链码

嵌套文件夹，否则系统解析处理链码文件时可能会出现找不到链码文件的错误。

上传链码成功后，界面如图 8-24 所示。

（3）安装链码。

在 Gaea 业务管理平台的【链码管理】页面中，单击【安装】按钮，如图 8-25 所示。

选择想要安装链码的节点（这里只可以选择本组织内的节点），选择后单击【提交】按钮即可，如图 8-26 所示。

在安装完成后，即可单击【详情】按钮（在【安装】按钮左侧），查看链码已安装的节点信息，如图 8-27 所示。

图 8-24　链码管理

图 8-25　链码管理-安装入口

图 8-26　链码管理-安装

图 8-27　链码管理-详情

(4) 实例化链码。

在 Gaea 业务管理平台的【链码管理】页面中，单击【实例化链码】按钮，如图 8-28 所示。

图 8-28　链码管理-实例化链码入口

参数为该链码的 init 函数中需要传入的参数，函数名为选填，背书策略可以自定义，也可以选择默认的“或”或者“与”，如图 8-29 所示。

实例化成功后，单击【详情】，即可看到链码详细信息，包括该链码实例化的结果，如图 8-30 所示。

图 8-29　链码管理-实例化链码

图 8-30　链码管理-详情

（5）链码的升级。

将编写好的新版本链码打包为.zip 文件，先使该链码执行完步骤（2）“上传链码”和步骤（3）“安装链码”的操作。注意：上传该升级链码时其名称需要与待升级链码的名称一致，版本则应该不一致以进行区分。

然后单击原已实例化链码的【升级】，执行升级操作，如图 8-31 所示。

此时目标链码为需要升级的链码的名称，已自动关联。链码版本为上传链码名称时指定的新版本号，已自动关联，只可以进行选择（升级链码时可以修改原链码实例化时指定的背书策略），如图 8-32 所示。

图 8-31　通道管理-升级入口

图 8-32　链码升级

如图 8-33 所示，升级链码成功后显示的链码，版本已经发生了变化。此时区块链平台中，该通道对应账本仍然保留之前的账本数据，而执行时的链码已经变成了升级后的链码。通过此方法，可以在业务流程更改或者需要新增、删除接口时，通过升级链码的方式替换掉原本的链码，使修改后的链码生效。

（6）查询交易信息。

在 Gaea 业务管理平台的【系统概况】页面中，如图 8-34 所示，可以查询交易量、交易信息、块信息、通道概况及链码概况。可以通过平台实时查看链码中的交易情况和数据。

图 8-35 为查看到的块中交易的详情，包括背书的组织、读写集、链码名称等交易详细信息。

图 8-33　链码状态-版本

图 8-34　系统概况-块信息

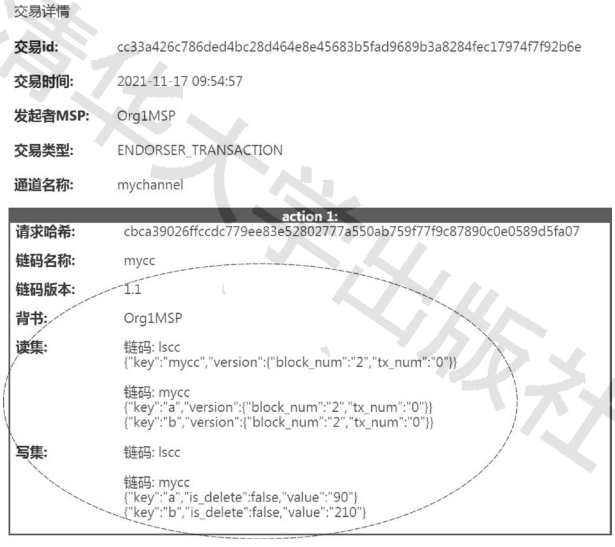


图 8-35　系统概况-块信息查询结果

3）网络、组织扩容

在网络通道已创建好、链码已经实例化、进行了多笔交易之后，如果有其他组织想要参与进来，该如何实现呢？

首先，需要参照第 2）步【创建组织】，创建好 peer 类型的组织信息（目前不支持扩容 orderer 类型的组织）。

然后，如图 8-36 所示，在区块链网络管理的【网络管理】页中，单击【添加组织】，可以完成网络中的组织扩容。

图 8-36　网络管理-添加组织入口

如图 8-37 所示，选择要添加的 peer 组织，在这里可以选择想要扩容到该网络中的组织。选择后单击【提交】即可。

图 8-37　网络管理-添加 peer 组织

提交后，即可在【网络管理】的【网络详情】页面中查看该扩容组织的状态信息，平台将会在 Fabric 服务器上启动该扩容组织对应的容器，容器启动成功后，组织信息会变成正常状态，如图 8-38 所示。但此时该组织尚不属于任何通道，还无法进行记账。

图 8-38　网络详情-新扩容组织状态

在【区块链网络管理】平台中完成对网络的组织扩容后，登录【区块链业务管理】平台，进入【通道管理】页面，如图 8-39 所示，单击【组织扩容】，就会进入通道的组织扩容页面。

图 8-39　通道管理-组织扩容入口

如图 8-40 所示，单击【发起邀请】按钮。

此时平台会提示“请选择组织”，邀请的组织必须属于该网络且不在该通道中。如图 8-41 所示，可以邀请刚刚加入该网络的组织 org11。

发起邀请后，并不会马上成功，因为区块链平台是联盟链，账本数据由多家组织共同维护、使用，邀请其他组织时需要大部分组织的同意，此时需要进入组织扩容页面（如图 8-42 所示），在此页面单击方框内的【同意】按钮（当有组织发起邀请后，通道内的所有组织都会收到消息，包括邀请发起者）。

当平台收集到 50％以上的组织同意的签名后，会执行扩容操作，此时通道扩容成功。

图 8-40　通道管理-组织扩容

图 8-41　通道管理-组织扩容发起邀请

图 8-42　通道管理-同意组织扩容

扩容成功后，新加入的组织的 Peer 节点中就会拉取同样的账本数据，并且能够发起交易，跟其他组织具备同样的权限。

8.1.2 代码设计

Gaea 系统分为两个平台，分别是区块链网络管理平台（监听 8071 端口）和区块链业务管理平台（监听 8081 端口）。

区块链网络管理平台的后台框架是 Flask，区块链业务管理平台的后台框架是 Egg，前端框架都是 Ant-design。这里主要介绍后台框架的架构设计。

1. Gaea 网络管理平台

Gaea 网络管理平台的后台框架是 Flask。Flask 是一个使用 Python 编写的轻量级 Web 应用框架，其 WSGI 工具采用 Werkzeug，模板引擎则使用 Jinja2。

Flask 的基本模式为：在程序里将一个视图函数分配给一个 URL，当用户访问这个 URL 时，系统就会执行给该 URL 分配的视图函数，获取函数的返回值并将其显示到浏览器上，其工作过程见图 8-43。

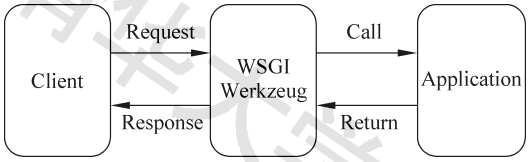


图 8-43 Flask 工作过程

之所以选择 Flask 来开发，原因如下：

- Flask 因为其灵活、轻便且高效的特点被业界认可，同时拥有 Werkzeug、Jinja2 等开源库，拥有内置服务器和单元测试，适配 RESTful API，支持安全的 cookies，而且官方文档完整，便于学习掌握。
- Flask 拥有灵活的 Jinja2 模板引擎，提高了前端代码的复用率，这样可以提高开发效率，有利于后期开发与维护。在现有标准中，Flask 算是微小型框架。
- 对于数据库访问、验证 Web 表单和用户身份认证等一系列功能，Flask 框架是不支持的。这些功能都是以扩展组件的方式实现，然后再与 Flask 框架集成。开发者可以根据项目的需求进行相应的扩展，或者自行开发，有很大的灵活性。这与大型框架恰恰相反，大型框架本身已做出了大部分决定，难以灵活改变方案。

网络管理平台主要分为如下几个功能模块：

- 路由管理；
- 数据库；
- 业务逻辑处理；
- 一般通用功能。

接下来看看这几个模块完成了哪些功能。

1）路由管理

使用普通的路由设置是不够的，由于程序的复杂度较高，需要对程序进行模块化的处理。新华三的网络管理平台采用了 Flask 内置的一个用于模块化处理的类，即 Blueprint。

Flask 使用 Blueprint 让应用实现模块化。Blueprint 具有如下属性：

- 一个应用可以有多个 Blueprint，可以将一个 Blueprint 注册到任何一个未使用的 URL 下，如“/”、“/sample”或者子域名；
- 在一个应用中，一个模块可以注册多次；
- Blueprint 可以具有自己独立的模板、静态文件或者其他通用操作方法，它并不必须实现应用的视图和函数；
- 在一个应用初始化时，就应该注册需要使用的 Blueprint。

以下为部分示例代码。< src/dashboard. py >的代码如图 8-44 所示。

```
app.register_blueprint(bp_index)
app.register_blueprint(bp_host_view)
app.register_blueprint(bp_host_api)
```

图 8-44　src/dashboard. py 代码示例

< src/resources/index. py >的代码如图 8-45 所示。

< src/resources/host_api. py >的代码如图 8-46 所示。

```
bp_index = Blueprint('bp_index', __name__)

@bp_index.route('/', methods=['GET'])
@bp_index.route('/index', methods=['GET'])
@login_required
def show():
    request_debug(r, logger)
    hosts = list(host_handler.list(filter_data={}))
    hosts.sort(key=lambda x: x["name"], reverse=False)
```

图 8-45　src/resources/index. py 代码示例

```
bp_host_api = Blueprint('bp_host_api', __name__,
                        url_prefix='/{}'.format("api"))

@bp_host_api.route('/hosts', methods=['GET'])
def hosts_list():
    logger.info("/hosts_list method=" + r.method)
    request_debug(r, logger)
    col_filter = dict((key, r.args.get(key)) for key in r.args)
```

图 8-46　src/resources/host_api. py 代码示例

通过上面的代码可以看到，dashboard. py 文件中，在 Flask 的实例化对象 app 中注册了两个 Blueprint 实例，分别为 bp_index 和 bp_host_view。在 index. py 文件及 host_api. py 文件中可以再定义各自模块的相对独立的路由。

项目中不仅是这两个路由对象，相对独立的资源都可以注册一个 Blueprint。

2）数据库

网络管理平台有许多数据需要存储，数据库模块选择了 MongoDB 数据库。

MongoDB 是一种基于分布式文件存储的，可以应用于各种规模的企业、各行业以及各类应用程序的开源数据库。作为适用于敏捷开发的数据库，MongoDB 的数据模式可以随着应用程序的发展而灵活地更新。与此同时，它也为开发人员提供了传统数据库的功能：二级索引、完整的查询系统以及严格一致性等。MongoDB 能够使企业具有敏捷性和可扩展性，各种规模的企业都可以通过使用 MongoDB 来创建新的应用，提高与客户之间的工作效率。

mongoengine 是一个对象文档映射器，使用 pIPinstallmongoengine 安装后，即可在 Flask 框架中使用。

以下为部分示例代码。<…/models/modelv2. py >的代码如图 8-47 所示。

该文件中定义了 BlockchainNetwork 和 BlockchainNetwork Schema 两个类，具体存储网络的数据内容。但是为什么定义两个类，一个继承于 Document，一个继承于 Schema 呢？这其实是由于需要对这个数据块进行序列化和反序列化操作。Schema 是 marshmallow 模块中的一个类。

BlockchainNetwork 类基于引用自 mongoengine 类的 Document，定义了 network 这样

```
from mongoengine import Document, StringField, \
    BooleanField, DateTimeField, IntField, \
    ReferenceField, DictField, ListField, CASCADE, DENY

class BlockchainNetwork(Document):

    id = StringField(required=True, primary_key=True)
    name = StringField(default="")
    description = StringField(default="")
    fabric_version = StringField(default="v1.1")
    orderer_orgs = ListField(StringField(), required=True)
    peer_orgs = ListField(StringField(), required=True)
    healthy = BooleanField(default=False)
    create_ts = DateTimeField(default=datetime.datetime.utcnow())
    status = StringField(choices=BLOCKCHAIN_NETWORK_STATUS)

    host = ReferenceField(HostModel, reverse_delete_rule=DENY)
    consensus_type = StringField(default="kafka")
    db_type = StringField()
    gm = BooleanField(default=False)
```

```
class BlockchainNetworkSchema(Schema):
    id = fields.String()
    name = fields.String()
    description = fields.String()
    fabric_version = fields.String()
    orderer_orgs = fields.List(fields.String())
    peer_orgs = fields.List(fields.String())
    healthy = fields.Boolean()
    create_ts = fields.DateTime()
    host_id = fields.Method("get_host_id")
    consensus_type = fields.String()
    status = fields.String()
    db_type = fields.String()
    gm = fields.Boolean()

    def get_host_id(self, network):
        return str(network.host.id)
```

图 8-47　…/models/modelv2.py 代码示例

一个数据结构，并定义了每个数据项的数据类型及初始数据。通过如图 8-48 所示的方法，将数据存入 MongoDB 数据库中。

```
network = modelv2.BlockchainNetwork(id=id,
                                    name=name,
                                    description=description,
                                    fabric_version=fabric_version,
                                    orderer_orgs=orderer_orgs,
                                    peer_orgs=peer_orgs,
                                    host=host,
                                    consensus_type=consensus_type,
                                    db_type=db_type,
                                    create_ts=create_ts,
                                    gm=gm,
                                    status="creating")
network.save()
```

图 8-48　将数据存入 MongoDB

使用时，执行“network＝modelv2.BlockchainNetwork.objects.get(id＝net_id)”，即可通过 id 查询到 BlockchainNetwork 的数据对象，并对该数据对象进行进一步的修改，删除等操作。也可以使用如下方法对已存储在 MongoDB 数据库中的数据项进行更新：

```
org_obj = modelv2.BlockchainNetwork.objects.get(id = org_id)
org_obj.update(set__name = ins)
```

如果需要删除某一个数据项，可以通过如下方法实现：

```
network = modelv2.BlockchainNetwork.objects.get(id = blockchain_network_id)
network.delete()
```

而 BlockchainNetworkSchema 类用于数据的反序列化，将数据项通过 json 格式返回给前端用户查看，如图 8-49 所示。

```
def _schema(self, doc, many=False):
    network_schema = modelv2.BlockchainNetworkSchema(many=many)
    return network_schema.dump(doc).data
```

图 8-49　数据的反序列化

当然，除了 BlockchainNetwork 还有其他数据结构，例如 ServiceEndpoint 用于存储 Peer 及 Orderer 的类型、域名等各数据项的详细信息，Organization 用于存储组织名、类型及与网络的关联信息，OperatorLog 用于存储操作日志信息，User 存储用户数据，Host 存储主机数据。限于篇幅，这里不再详细展示数据定义项。

3）业务逻辑处理

Gaea 的区块链 BaaS 平台支持 Docker 类型和 K8S 类型的主机，对网络相关资源进行一般的增、删、改、查、更新等操作时，均需要针对不同的主机类型进行定制的处理。

如图 8-50 所示，在创建网络的过程中，首先获取主机 host 的数据对象，而创建组织对象时，由于 host.type 的不同，会调用不同的类方法。当 host.type 为 Docker 时，调用<…/Docker/blockchain_network.py>中的 create_Orderer_org 方法；而当 host.type 为 K8S 时，调用<…/K8S/blockchain_network.py>中的 create_Orderer_org 方法。这是由于在__init__中将 host_agents 初始化为一个字典结构对象，如图 8-51 所示。

```
for orderer_org in network_config['orderer_org_dicts']:
    host_id = orderer_org['host_id']
    host_handler.refresh_status(host_id)
    host = host_handler.get_active_host_by_id(host_id)
    host.update(add_to_set__clusters=[net_id])
    pbft_node_id_base = len(orderer_org['ordererHostnames'])*2
    self.host_agents[host.type].create_orderer_org(orderer_org, consensus_type, host, net_id,
                                                   net_name, fabric_version,  request_host_ports,
                                                   portid, gm, pbft_node_id_base, pbft_node_table)
```

图 8-50　业务逻辑处理器

当 host.type 为 Docker 时，需要通过页面的自定义配置，构造出符合规范的 Docker-compose.yaml 文件，然后使用引入的 compose.cli.command 模块中的方法，指定文件路径及服务器的 IP 和端口，将容器启动起来，如图 8-52 所示。

```
class BlockchainNetworkHandler(object):
    """ Main handler to operate the cluster in pool
    """
    def __init__(self):
        self.host_agents = {
            'docker': NetworkOnDocker(),
            'kubernetes': NetworkOnKubenetes()
        }
```

图 8-51　网络创建初始化

```
from compose.cli.command import get_project as compose_get_project

composefile_dict['services'].update(sevices_dict)
        deploy_dir = '{}/deploy/'.format(net_dir)
        if not os.path.exists(deploy_dir):
            os.makedirs(deploy_dir)
        composefile = '{}/docker-compose.yaml'.format(deploy_dir)

        with open(composefile, 'w') as f:
            yaml.dump(composefile_dict, f)

        project = compose_get_project(project_dir=deploy_dir,
                                      host = host.worker_api,
                                      project_name=net_id[:12])

        containers = project.up(detached=True, timeout=5)

        portid[0] = index

        return containers
```

图 8-52　host. type 为 Docker 时的自定义配置

当 host. type 为 K8S 时，所需的 K8S 相关部署资源需要配置，yaml 文件较多、较复杂不过与配置 Docker 主机类型时的总体思路基本一致，需要根据平台中用户的配置，构造所需的 yaml 文件后，然后调用 Flask 支持的、通用的 K8S 模块的相关接口，实现对 K8S 主机上 pod 容器的调度，如图 8-53 所示。

```
class K8sNetworkOperation():
    """
    Object to operate cluster on kubernetes
    """
    def __init__(self, kube_config):
        client.Configuration.set_default(kube_config)
        self.extendv1client = client.AppsV1Api()
        self.corev1client = client.CoreV1Api()
        self.appv1beta1client = client.AppsV1Api()
        self.support_namespace = ['Deployment', 'Service',
                                  'PersistentVolumeClaim', 'StatefulSet', 'ConfigMap']
        self.create_func_dict = {
            "Deployment": self._create_deployment,
            "Service": self._create_service,
            "PersistentVolume": self._create_persistent_volume,
            "PersistentVolumeClaim": self._create_persistent_volume_claim,
            "Namespace": self._create_namespace,
            "StatefulSet": self._create_statefulset,
            "ConfigMap": self._create_configmap
        }

.....
.....

def _create_deployment(self, namespace, data, **kwargs):
    try:
        resp = self.extendv1client.create_namespaced_deployment(namespace, data,
                                                                 **kwargs)
        logger.debug(resp)
    except ApiException as e:
        logger.error(e)
    except Exception as e:
        logger.error(e)
```

图 8-53　host. type 为 K8S 时的用户配置

4）一般的通用功能

在网络管理平台中还需要实现其他通用功能，如 yaml 文件的 dump 和 load 操作、日志功能、yaml 文件的更新操作、License 管理等。

2. Gaea 业务管理平台

Gaea 业务管理平台的后台框架是 Egg，使用 node.js 语言。Egg 是阿里巴巴公司研发的、基于 KOA 的企业级应用开发框架，按照约定进行开发，奉行约定优于配置的原则，团队协作成本低。Egg 框架有如下特性：

- 提供定制上层框架的能力；
- 提供高度可扩展的插件机制；
- 内置多进程管理。

Egg 框架的目录结构有约定的规范，如图 8-54 所示。

```
server(egg-project)
├── app
|  ├── router.js
|  ├── controller
|  |  └── home.js
|  ├── service (可选)
|  |  └── user.js
|  ├── middleware (可选)
|  |  └── accesslog.js
|  └── extend (可选)
|  ├── helper.js
|  ├── request.js
|  ├── response.js
|  ├── context.js
|  ├── application.js
|  └── agent.js
├── config
|  ├── plugin.js
|  ├── config.default.js(默认环境)
|  ├── config.local.js(本地开发环境)
|  ├── config.prod.js(线上环境)
|  ├── config.test.js (测试环境)
|  └── config.stage.js (预发环境)
├── logs
├── node_modules
├── jcloud
├── run
```

图 8-54　egg 框架的目录结构

app/controller/目录下的文件用于解析用户的输入，处理后返回相应的结果。app/service/目录下的文件用于编写业务逻辑层，可选，建议使用。app/middleware/目录下的文件用于编写中间件，可选。app/public/目录下的文件用于放置静态资源，可选。app/extend/目录下的文件用于框架的扩展，可选。config/config.{env}.js 用于编写配置文件（{fenv}指当前环境，具体为 default、local、prod、test、stage 中的某一个）。config/plugin.js 用于配置需要加载的插件。

业务管理平台主要分为如下几个模块：

- 路由管理；
- 数据库；
- 通道、链码、用户等业务功能模块；
- FabricSDK 操作。

1）路由管理

与网络管理平台相似，业务管理平台也包括路由管理模块，由于框架及语言的不同，路由的使用方法也有不同。代码示例如图 8-55 所示。

```
module.exports = app => {
  const { router, controller, passport, io } = app;
  .....
  省略若干.
  .....
  router.get('home', '/', controller.home.index);

  router.post('/login', passport.authenticate('local', { successRedirect: '/' }));
  router.get('/logout', controller.home.logout);
  require('./router/api')(app);
  io.of('/').route('join', io.controller.home.join);

  router.prefix('/');
```

图 8-55　路由使用方法

这是在项目启动时平台构造的初始路由，定义了根路径、/login、/logout 等框架基本路由，可以看到“require('./router/api')(app)”；这条语句，这是引入 app.js 文件，可以在该文件中定义框架需要用到的、各个业务功能相关的后台路由，如图 8-56 所示。

```
module.exports = app => {
    app.router.get('/api/currentUser', app.controller.user.currentUser);
    app.router.get('/api/chain', app.controller.chain.list);
    app.router.post('/api/chain', app.controller.chain.apply);
    app.router.get('/api/chain/:id', app.controller.chain.query);
    app.router.get('/v2/channels', app.controller.channel.getChannels);
    app.router.get('/v2/channels/:channel_id', app.controller.channel.getChannel);
    app.router.post('/v2/channels',app.controller.channel.create);
    app.router.get('/v2/peers',app.controller.channel.getPeers);
    app.router.put('/v2/peers/role', app.controller.channel.changePeerRole);
    app.router.post('/v2/channels/:channel_id/peerJoin',app.controller.channel.join);
.....
.....
```

图 8-56　定义后台路由

url 格式符合规范，app.router 后面的 get/put/post 定义了该路由的 http 方法，后面的 app.controller.即为该路由对应访问的方法函数。

2）数据库

业务管理平台与网络管理平台类似，也采用了 MongoDB 数据库。使用方法类似，需要先定义数据结构，如图 8-57 所示。

```
module.exports = app => {
    const mongoose = app.mongoose;
    const Schema = mongoose.Schema;

    const ChannelSchema = new Schema({
        name: { type: String },
        description: { type: String },
        orderer_url: { type: String },
        peer_orgsName: { type: Array },
        creator_id: { type: String },
        creator_name: { type: String },
        version: { type: String },
        blockchain_network_id: { type: String },
        peers_inChannel: { type: Array },
        date: { type: Date },
    });

    return mongoose.model('Channel', ChannelSchema);
};
```

图 8-57　定义数据结构

如图 8-57 所示，我们定义了一个 Channel 类型的数据结构。在使用时，即可针对该数据项进行基本的增、删、改、查、更新等操作。如图 8-58 所示是对 Channel、Chaincode 类型的数据结构的查询操作。

```
const channelInfo = await ctx.model.Channel.findOne({ _id: channelId });
const chainCode = await ctx.model.ChainCode.findOne({ _id: chaincodeId });
```

图 8-58　Channel 类型的数据结构

3）通道、链码、用户等业务功能模块

业务管理平台的核心功能是通过 Fabric 提供的 SDK 来执行 Fabric 支持的相关操作，包括但不限于：通道的创建，将 Peer 节点加入通道中，上传链码，将链码安装到 Peer 节点上，实例化链码，链码升级，链码的调用，以及网络扩容、组织扩容、通道扩容等功能。

下面以通道的创建为例，说明如何在 Egg 框架中执行业务逻辑操作。

首先在 controller/channel. js 文件中，通过 create()接口解析用户的输入，然后调用 service 层的 create 接口，如图 8-59 所示。

```
async create() {
    const { ctx } = this;
    if((typeof(ctx.req.body.channel.name)!=='string')||
        (typeof(ctx.req.body.channel.description)!=='string')||
        (typeof(ctx.req.body.channel.orderer_url)!=='string')||
        ((Array.isArray(ctx.req.body.channel.peer_orgs)) === false)) {
            ctx.log.error('some params type validate failed when create, please check');
            throw new Error("channel inputdates' type validate failed");
        }

        const channel = await ctx.service.channel.create();
        ctx.log.debug(JSON.stringify(channel));
        ctx.status = channel.success ? 200 : 400;
        ctx.body = {
          channel,
        };
      }
```

图 8-59　controller/channel. js 解析用户输入

而对于所有的业务逻辑处理，MongoDB 数据库的操作均在 service 层进行。

用户及其他基本数据项的解析如图 8-60 所示。

```
const orgName = userName.split('@')[1].split('.')[0];
const networkId = channel.blockchain_network_id;
const fabricFilePath = `${config.fabricDir}/${networkId}`;

const channelConfigPath = `${fabricFilePath}/${channel._id}/channel-artifacts`;
const channelName = channel.name;
const fabricVersion = channel.version;
```

图 8-60　用户及其他基本数据项的解析

Channel 的数据项的创建操作如图 8-61 所示。

```
var channel = await ctx.model.Channel.create({
  name,
  description,
  orderer_url,
  peer_orgsName,
  version: fabricVersion,
  creator_id: ctx.user.id,
  creator_name: ctx.user.username,
  blockchain_network_id: networkId,
  date,
});
```

图 8-61　Channel 数据项的创建

network 数据结构的构造如图 8-62 所示。

```
const channelsConfig = {};
channelsConfig[`${config.default.channelName}`] = channels;
network = Object.assign(network, {
  config: {
    version: '1.0',
    'x-type': 'hlfv1',
    name: `${chain.name}`,
    description: `${chain.name}`,
    orderers,
    certificateAuthorities,
    organizations,
    peers,
    channels: channelsConfig,
  },
});

return network;
```

图 8-62　network 数据结构的构造

利用所有处理后的数据项进行 FabricSDK 的接口调用，同步等待执行结果，如图 8-63 所示。

```
await ctx.createChannel(network, channelName, channelConfigPath, orgName, userName,
                        channel.version);
```

图 8-63　FabricSDK 接口调用

service 层会将获取的执行结果及返回的数据返回给 controller 层，controller 层再将数据及代码返回给用户。

4）FabricSDK 操作

通过对 FabricSDK 的调用，实现区块链 Fabric 容器中对应资源的使用，是业务平台的核心目的。目前 Fabric 支持 NodeJS 和 Python 两种版本的 SDK。由于 NodeJS 发展时间更早、更为成熟，本书采用了 NodeJS 版本的 SDK。

在 package.json 中指定 Fabric 的版本，目前最新版本的 Gaea 采用了 Fabric 1.4.4 版本，如图 8-64 所示。

```
{
  "name": "user-dashboard",
  "version": "1.0.0",
  "description": "Cello User Dashboard",
  "private": true,
  "dependencies": {
    "fabric-ca-client": "1.4.4",
    "fabric-client": "1.4.4",
    "fabric-network": "1.4.4"
  },
```

图 8-64　package.json 指定 Fabric 版本

createchannel 调用 service 层的 ctx.createChannel（见图 8-63）后，会执行文件<../src/app/lib/Fabric/v1_4.js>，在该文件中，Gaea 会调用 SDK 提供的接口，如图 8-65 所示。

getClientForOrgCA 接口中调用的 loadFromConfig 等几个接口都是在 FabricSDK 文件中定义的。具体接口定义见 SDK 文档 https://hyperledger.github.io/Fabric-SDK-Node/release-1.4/index.html。

```
const hfc = require('../../../packages/fabric-1.4/node_modules/fabric-client');

async function getClientForOrgCA(id, orgName, network, username) {
    const ctx = app.createAnonymousContext();
    ctx.req.request_id = id;
    const client = hfc.loadFromConfig(network.config);
    client.loadFromConfig(network[orgName]);

    await client.initCredentialStores();
```

图 8-65　Gaea 调用 SDK 接口

通过 getClientForOrgCA 接口可以获取验证了用户权限的 client 对象，如图 8-66 所示。

```
async function createChannel(id, network, channelName, channelConfigPath, username, orgName)
{
    try {
        const ctx = app.createAnonymousContext();
        ctx.req.request_id = id;
        const client = await getClientForOrgCA(id, orgName, network, username);
        ctx.log.debug('Successfully got the fabric client for the organization "%s"', orgName);
        // read in the envelope for the channel config raw bytes
        const envelope =
fs.readFileSync(path.join(`${channelConfigPath}/${channelName}.tx`));
        var channelConfig = client.extractChannelConfig(envelope);
        const signature = client.signChannelConfig(channelConfig);

        const request = {
          config: channelConfig,
          signatures: [signature],
          name: channelName,
          txId: client.newTransactionID(true), // get an admin based transactionID
        };
        const result = await client.createChannel(request)
        ctx.log.debug('create channel result: ', result);
        if (result) {
          if (result.status === 'SUCCESS') {
            ctx.log.debug('Successfully created the channel.');
            const response = {
              success: true,
              message: 'Channel \'' + channelName + '\' created Successfully',
```

图 8-66　通过 getClientForOrgCA 获取 client 对象

client.extractChannelConfig 从 configtxgen 工具生成的 ConfigEnvelope 对象中提取 protobuf ConfigUpdate 对象，然后可以使用 Client 类的 signChannelConfig()方法对返回的对象进行签名。收集完所有签名后，ConfigUpdate 对象和签名就可以用于 createChannel()方法。

client.createChannel(request)将创建通道的请求发送给了 Orderer，如果收到状态为 Success 的返回消息，则表明通道创建成功。

其他所有与 Fabric 相关的操作都与此类似，需要根据 Fabric 的流程来进行数据的处理与构造，具体可参考 SDK 文档。

8.1.3　智能合约编写

1. 一个链码的整体逻辑

链码(Chaincode)会对 Fabric 应用程序发送的交易做出响应，执行代码逻辑，与账本进

行交互。

链码与 Fabric 之间的逻辑关系如下：

- Hyperledger Fabric 中，Chaincode 默认运行在 Docker 容器中。
- Peer 通过调用 Docker API 来创建和启动 ChainCode 容器。
- Chaincode 容器启动后与 Peer 之间创建 gRPC 连接，双方通过发送 ChaincodeMessage 来进行交互通信。
- Chaincode 容器利用 core.chaincode.shim 包提供的接口来向 Peer 发起请求。

每个 Chaincode 程序都必须实现 Chaincode 接口，接口中的方法会在响应传来的交易时被调用。

```
type Chaincodeinterface{
    Init(stub ChaincodeStubInterface) pb.Response
    Invoke(stub ChaincodeStubInterface) pb.Response
}
```

Init(初始化)方法会在 Chaincode 接收到 Instantiate(实例化)或者 Upgrade(升级)交易时被调用，进而使得 Chaincode 顺利执行必要的初始化操作，包括初始化应用的状态。

Invoke(调用)方法会在响应 Invoke 交易时被调用以执行交易。

2. 基本的链码结构

(1) 引入必要的包。

```
import (
    "fmt"
    "github.com/hyperledger/Fabric/core/chaincode/shim"
    pb"github.com/hyperledger/Fabric/protos/Peer"
)
```

fmt 是 golang 系统提供的通用输入输出包，后面两个包 shim 和 Peer 是必需的。

第二个包 shim 是 Fabric 系统提供的上下文环境，包含了 Chaincode 和 Fabric 交互的接口，在 Chaincode 中，执行赋值、查询等操作都需要通过 shim。

(2) 声明一个结构体，即 Chaincode 的主结构体。

该结构体需要实现 Fabric 提供的接口 github.com/hyperledger/Fabric/protos/Peer，其中必须实现 Init()和 Invoke()两个方法。

```
type DemoChaincodestruct{}
```

(3) 实现 Init()和 Invoke()方法，其中利用 shim.ChaincodeStubInterface 结构实现与账本之间的交互逻辑。

Init()方法实现链码初始化或升级时的处理逻辑，编写时可以灵活使用 stub 中的 API。

```
func(t *DemoChaincode)Init(stub shim.ChaincodeStubInterface) pb.Response{
    return stub.Success(nil)
}
```

Invoke()方法实现链码运行中被调用或查询时的处理逻辑，编写时可以灵活使用 stub

中的 API。

```
func (t * DemoChaincode) Invoke(stub shim.ChaincodeStubInterface) pb.Response
    return stub.Success(nil)
}
```

(4) 主函数需要调用 shim.Start()方法,只有存在该方法,才会启动对应的链码容器,才会有环境来执行链码中的其他方法。

```
Func main(){
    err : = shim.Start(new(DemoChaincode))
    if err!= nil{
        fmt.Printf("Error starting DemoChaincode: % s",err)
    }
}
```

8.1.4　与应用对接

Gaea 区块链平台通过 Fabric-CA 进行用户的注册管理,并使用 Token 来验证用户身份及 API 操作权限。

Token 中包含了用户身份信息。对于该文档中的所有 API 接口,在调用时均需在请求的 Headers 中包含 Token 信息,如图 8-67 所示。

图 8-67　PostManHeaders 示例

1. 获取 Token

(1) URL: https://IP: port/v2/token。

(2) 方法: POST。

(3) 请求参数: body。格式如下:

```
{
    "username": username@org1.ex1.com,
    "password": "userpassword"
}
```

(4) 返回参数。

成功时返回:

```
Code:200
Body: { " token ":" JWTeyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9. eyJkYXRhIjp7ImlkIjoiNWU1
Y2I3MGI4YWMzZTEwMDVkY2E3YTNiIiwidXNlcm5hbWUiOiJBZG1pbkBvcmcxLmV4Mi5jb20ifSwiZXhwIjoxNTg0Mz
QxOTUyLCJpYXQiOjE1ODM5MDk5NTJ9. tEeABkcJ3v - owK9 - oBj17kJseuBEJaSegJ5WMMTb4Xj9binsxx4o5u
UjxFk33OVqPQNBlDBYW _ U - K32OXYnSYk4UMLAIOdbaQPyeraKszStrtk2xE4LrDlgq6ChAu9yurqN4kmXFUZViY2C0x _
vK8aFqQDUhIdZ4BgZsYvDa78EFao0V0GjR64ScHeQBSQ6u4zy9epc2zxIteYilVXG1TUaV3OQWumY7SZzxP - u6V
- xFITxvlR5ID2xTIBODN8d _ tf7sHbdeefWq17UQouUpKZzfcDktzLF7pZ7 - Z _ eiJoc3R _ - zWa8 -
```

```
wD2TCOqa4YkC6bSBo9l_Ua1dO71lg1lfDg",
"success": true}
失败时返回:
Code:400
Body:
{
"message":"Pleasecheckiftheusernameandpasswordisright",
"success":false
}
```

（5）说明：该接口获取用户 Token 值，以备后续调用 Gaea 区块链平台 API 时使用。首次需要用组织 admin 的账户（即 Gaea 区块链平台创建区块链时的组织用户）登录，后续可用该 Token 创建的各用户名登录。建议该用户与 Gaea 区块链业务管理平台的用户一一绑定。调用 Gaea 平台 API 之前需要通过 Gaea 用户的 Token 来操作，成功时返回 token，失败时返回 400。username 的格式如前文所示，为：创建的 username@组织名.组织域名。

2. 智能合约

部署好区块链网络、安装好实例化链码之后，不仅可以通过 Gaea 区块链平台调用链码，也可以通过对接其他应用系统来执行链码，而应用调用链码的方式为 RestAPI 方式。

Gaea 平台对外提供的用于与区块链网络对接、实现数据上链和查询的调用方法如下方所示，API 的 URL 地址是固定的，用户只需要以实际的 IP 地址及 channelid 来替换即可调用。body 中的参数以实际需要调用的链码为准。

调用时请参考如下格式。

（1）URL：http://IP：port/v2/channels/：channel_id/chaincodeOperation

注：URL 中的 channel_id 需要用实际 channel 的 ID 来替换。

（2）方法：POST。

（3）body

```
{
    "chaincode_operation":{
        "operation":"",       //invoke 或 query,当链码仅仅执行查询操作时为 query,其他时候为 invoke
        "functionName":"",    //实际链码中定义的链码接口,如 3.2.1 节中定义的接口函数 invoke,query 和 delete
        "args":"",            //调用接口时需要的入参
        "chaincodeId":""      //上传的每个链码都有一个唯一的 id,即 chaincodeId
        }
}
```

说明：operation 指操作，分为 invoke 和 query；functionName 指函数名称，每个 API 对应一个；args 指函数的参数；chaincodeId 为将智能合约上传后自动生成的一个唯一标识。

注意：args 需要按照参数顺序来赋值，请严格按照顺序赋值。如有参数需要为空值，请增加空字符串来占位。

8.2 链码开发与部署实例

以一个实现 A、B 账户之间转账的链码为例，演示链码开发、上传、安装、实例化到最后实际使用的全流程。

在执行如下链码开发和部署的实例操作前，请确保已经根据 8.1.1 节中的“2.平台操作”创建了网络和通道，并已将节点组织增加到通道中。

8.2.1 转账链码示例

如下链码主要实现这样的功能：在链码实例化的时候，会初始化两个账户，并分别赋予账户指定的余额数；链码有三个功能函数 invoke、query 和 delete。

invoke 函数实现转账功能，可以实现 A 账户将部分余额 X 转账给 B 账户；query 函数实现查询余额的功能，可以从区块链中读取某个账户的余额；delete 函数则可以从区块链中删除某个指定的账户，但是请注意，这里的删除并不是真正意义上地完全清除干净该账户，由于区块链的特性，数据是以链的方式向后递增，因此这里并不会删除某个历史区块，而是删除了键值数据库(worldstate)中的数据，同时，delete 函数也是一笔交易，会以区块的形式递增地存储在区块链中。

完整的链码如下所示：

```
package main                    //包名.必须有一个包名为 main,否则会导致 Go 编译时无法生成 EXE
文件,实例化会失败
import (
/* 引入包,注意 shim 和 peer 是必须要引入的.shim 中定义了账本操作的接口,所有调用函数都需
要通过 shim 包来调用; 而 peer 是 Fabric 源码中的 protos 下的 peer 包,这里定义了很多数据结构和
函数,链码中用 Response 作为统一的返回值结构. */
    "fmt"
    "strconv"
    "github.com/hyperledger/fabric/core/chaincode/shim"
    pb "github.com/hyperledger/fabric/protos/peer"
)

//main()函数每个链码中都需要有一个 main()函数,在 main()函数中需要包含 shim.Start,这是链码
容器启动的关键函数.用户链码调用了该函数后,会向 Peer 发起消息、进行注册,开始链码与 Peer 之
间的交互
func main() {
    err := shim.Start(new(example02.SimpleChaincode))
    if err != nil {
      fmt.Printf("Error starting Simple chaincode: %s", err)
}
}

// 简单链码的简单实现
type SimpleChaincode struct {  //声明一个结构体
}

//为结构体添加 Init 方法.该方法会在链码被实例化的时候调用
func (t *SimpleChaincode) Init(stub shim.ChaincodeStubInterface) pb.Response {
    fmt.Println("ex02 Init")
//这里首先从实例化的调用方法中获取参数,并声明几个 string 和 int 类型的变量
    _, args := stub.GetFunctionAndParameters()
    var A, B string            // 记账对象
    var Aval, Bval int         // 持有资产
```

```
    var err error

    if len(args) != 4 {
        return shim.Error("Incorrect number of arguments. Expecting 4")
    }

    // 初始化链码
    A = args[0]
    Aval, err = strconv.Atoi(args[1])
    if err != nil {
        return shim.Error("Expecting integer value for asset holding")
    }
    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)

//可以看出以上,操作是将参数分别赋值给声明的变量,其中 A 和 B 为账户,
//Aval 和 Bval 分别是 A、B 账户的值

    // 将状态写入账本
    err = stub.PutState(A, []byte(strconv.Itoa(Aval)))
    if err != nil {
        return shim.Error(err.Error())
    }

    err = stub.PutState(B, []byte(strconv.Itoa(Bval)))
    if err != nil {
        return shim.Error(err.Error())
    }
//stub.PutState()就是将数据存入账本的接口函数.这里需要注意,PutState 接口的函数原型有两个参数,
//一个为 string 类型,作为 key,另一个为[]byte 类型
    return shim.Success(nil)
}

/* 为结构体添加 Invoke()方法,该方法在链码实例化成功后,在使用过程中被调用.这里以大写字母开头的 Invoke 相当于是一个链码调用的总入口,函数调用时会先进入这里,通过 stub.GetFunctionAndParameters()来获取具体调用的函数名,再通过 if-else 判断找到具体的函数内容来执行. */
func (t *SimpleChaincode) Invoke(stub shim.ChaincodeStubInterface) pb.Response {
    fmt.Println("ex02 Invoke")
    function, args := stub.GetFunctionAndParameters()
    if function == "invoke" {
        // 实验 A 给 B 转账
        return t.invoke(stub, args)
    } else if function == "delete" {
        //从状态中删除某个记账对象
        return t.delete(stub, args)
```

```
    } else if function == "query" {
        return t.query(stub, args)
    }

    return shim.Error("Invalid invoke function name. Expecting \"invoke\" \"delete\" \"query
\"")
}

//invoke 函数实现了将 A 账户中的金额 X 转账给 B 账户的功能
func (t *SimpleChaincode) invoke(stub shim.ChaincodeStubInterface, args []string) pb.
Response {
    var A, B string          // 记账对象
    var Aval, Bval int       // 持有资产
    var X int                // 交易资产额
    var err error

    if len(args) != 3 {
        return shim.Error("Incorrect number of arguments. Expecting 3")
    }

    A = args[0]
    B = args[1]

    // 从账本中获取当前资产状态
    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))

    // 执行转账计算操作
    X, err = strconv.Atoi(args[2])
    if err != nil {
        return shim.Error("Invalid transaction amount, expecting a integer value")
    }
    Aval = Aval - X
    Bval = Bval + X
    fmt.Printf("Aval = %d, Bval = %d\n", Aval, Bval)
//从 A 的余额里减去 X,并将其加到 B 账户中
```

```
    //将修改后的 A、B 账户存回账本中
    err = stub.PutState(A, []byte(strconv.Itoa(Aval)))
    if err != nil {
        return shim.Error(err.Error())
    }

    err = stub.PutState(B, []byte(strconv.Itoa(Bval)))
    if err != nil {
        return shim.Error(err.Error())
    }

    return shim.Success(nil)
}

//删除账户
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]

    err := stub.DelState(A)
    if err != nil {
        return shim.Error("Failed to delete state")
    }

    return shim.Success(nil)
}

//query 操作,查询某个账户的余额
func (t * SimpleChaincode) query(stub shim.ChaincodeStubInterface, args []string) pb.Response
{
    var A string                    // Entities
    var err error

    if len(args) != 1 {
        return shim.Error("Incorrect number of arguments. Expecting name of the person to
query")
    }

    A = args[0]

    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)
    }

    jsonResp := "{\"Name\":\"" + A + "\",\"Amount\":\"" + string(Avalbytes) + "\"}"
    fmt.Printf("Query Response: %s\n", jsonResp)
    return shim.Success(Avalbytes)
}
```

从上面这个链码示例中可以看到一个完整、可用的链码必须包含的基本结构。方法Invoke()中调用的方法名及方法内容都可以自定义，但是从链码的交互调度和可操作性来看，它必须要满足如下结构要求：首先，必须定义一个包名，即 import 要用的包，这也是 Go 语言的基础语法要求；其次，要有 main 函数，并需要在 main 函数中执行 shim.Start()接口，链码作为一个功能完备的 Go 程序，需要一个入口，main 函数不可或缺，而 shim.Start()是链码容器启动及与 Peer 交互的开始；然后，需要定义一个 struct 类型的结构体，链码中的 Init()和 Inooke()方法以及它们调用的链码具体功能实现，都是这个结构体的方法；最后，必须有 Init()方法和 Invoke()方法，这两个方法一个是实例化时调用的接口，一个是用户链码后续日常调度方法时的接口，链码其他具体的实现放在 Invoke 方法中即可。

8.2.2　链码压缩

链码编码完成后，将其放入一个目录中，并对该目录进行压缩（目录不要嵌套，Go 文件直接保存在该目录中，否则可能导致找不到链码文件）。

目前新华三 Gaea 区块链平台要求链码必须压缩为.zip 格式。

要求目录不要嵌套，并不是说目录里绝对不能有其他目录文件，而是说 Go 的入口执行文件需要保存在当前目录中，链码较复杂的情况下请参考 Go 语言的包管理规则来合理进行规划。

将 8.2.1 节中的转账链码保存到 chaincode_example02.go 文件中，将该 Go 文件保存到 chaincode_example02 文件夹中，并对该文件夹进行压缩，得到 chaincode_example02.zip 文件，如图 8-68 所示。

图 8-68　链码压缩示例

8.2.3　上传

将压缩后的链码文件上传到区块链平台中。这一步操作只需要执行一次，而不需要将每个组织都上传，同一网络内所有组织均可以看到该链码。

由于链码等同于企业间的合同，是联盟内组织一起商定的合约，为了保证安全，上传链码的时候需要同时提交一个链码的 MD5 值。这个 MD5 值有两个作用，一方面为了防止由于网络传输出错等原因而导致上传的文件发生错误，平台会将上传文件的 MD5 值与提交

的 MD5 值进行校验，不匹配则上传失败；另一方面，当各个组织安装链码时，也会将从平台获取的链码与该 MD5 值进行校验，避免平台中的链码被人为修改。

MD5 是一个通用的算法，用户可以在线获取某个文件的 MD5 值，也可以通过一些工具来获取。

将 8.2.2 节压缩得到的 chaincode_example02.zip 链码文件上传到平台。注意：两个文件只要有少许差别，其 MD5 值就会截然不同，所以 MD5 值需要根据自己压缩的文件自行生成。

如图 8-69 所示，链码名称及版本可以自定义，这里语言选择 golang。填写完成后，单击【提交】按钮即可。

图 8-69　上传链码示例

8.2.4　安装

将上述链码上传成功后，即可在链码管理页面中查看该链码。

此时需要参考图 8-25 开始进行链码的安装。

根据图 8-25 所示的入口，进入【安装链码】页面后，需要选择节点。此时该通道内的所有节点均可选择，如图 8-26 所示。单击【选择节点】文本框，将弹出本组织中尚未安装该链码的所有节点。可以多选，甚至可以选中所有节点，在这些节点上均安装链码。

选择哪些节点是自己决定的，但是只有安装了链码的组织节点能够参与背书（因为背书需要执行交易，以校验交易结果是否正确）。

操作如图 8-70 所示。

8.2.5　实例化

当链码上传成功后，需要对该链码执行实例化操作。实例化成功后，就会在主机上启动对应的链码容器，后续所有的链码交易都会在该链码容器中执行。

图 8-70　链码安装

继续将 8.2.4 节安装的链码 mycctest 进行实例化操作。此时需要参考图 8-28 进行链码的实例化操作。

在实例化页面中，选择之前创建的通道，输入参数“a,100,b,200”，因为实例化的时候链码容器会执行 Init 函数，如下所示，链码里会获取传入的 args，并且分别赋值给声明的变量 A、Aval、B、Bval。然后，Init 函数会分别将变量 A（值为 Aval）和变量 B（值为 Bval）存入账本。

```
func (t *SimpleChaincode) Init(stub shim.ChaincodeStubInterface) pb.Response {
    _, args := stub.GetFunctionAndParameters()
    var A, B string            // 记账对象
    var Aval, Bval int         //持有资产
    var err error

    if len(args) != 4 {
        return shim.Error("Incorrect number of arguments. Expecting 4")
    }
    //初始化链码
    A = args[0]
    Aval, err = strconv.Atoi(args[1])
    …
    B = args[2]
    Bval, err = strconv.Atoi(args[3])
    …
    err = stub.PutState(A, []byte(strconv.Itoa(Aval)))
    …
    err = stub.PutState(B, []byte(strconv.Itoa(Bval)))
    …
}
```

操作如图 8-71 所示。

如果本组织内没有任何节点安装链码，则在指定背书策略时，需要注意该组织不能被指定为必须背书的组织，否则会导致实例化失败。

8.2.6　交易查询

在 Gaea 业务管理平台的【通道管理】页面中，单击【通道详情】链接，如图 8-72 所示。

进入通道详情页面，选择【实例化链码列表】选项卡，如图 8-73 所示。

链码中的 query 接口如下，可以看到，首先只需要一个参数，并将该参数赋值给变量 A，

图 8-71　实例化链码示例

图 8-72　通道管理-通道详情

图 8-73　通道管理-实例化链码列表

再将 A 存在账本中的数据读取出来，并将数据返回。

```
func (t * SimpleChaincode) query(stub shim.ChaincodeStubInterface, args []string) pb.Response
{
var A string              //记账对象
var err error
```

```
if len(args) != 1 {
    return shim.Error("Incorrect number of arguments. Expecting name of the person to query")
}

A = args[0]

//将状态写入账本
Avalbytes, err := stub.GetState(A)
...
return shim.Success(Avalbytes)
}
```

在【实例化链码列表】页签中，先选中想要操作的链码，单击【选择】栏中链码对应的单选按钮框即可。【操作】选择 query，【函数名】输入 query，【参数值】输入 a，如图 8-74 所示。如果是 query 查询操作，结果会在下方显示。

因为实例化时的入参为“a，100，b，200”，所以执行 query 操作查询 a 的值时获取到数值 100，执行 query 操作查询 b 的时候获取到值 200。链码查询如图 8-74、图 8-75 所示。

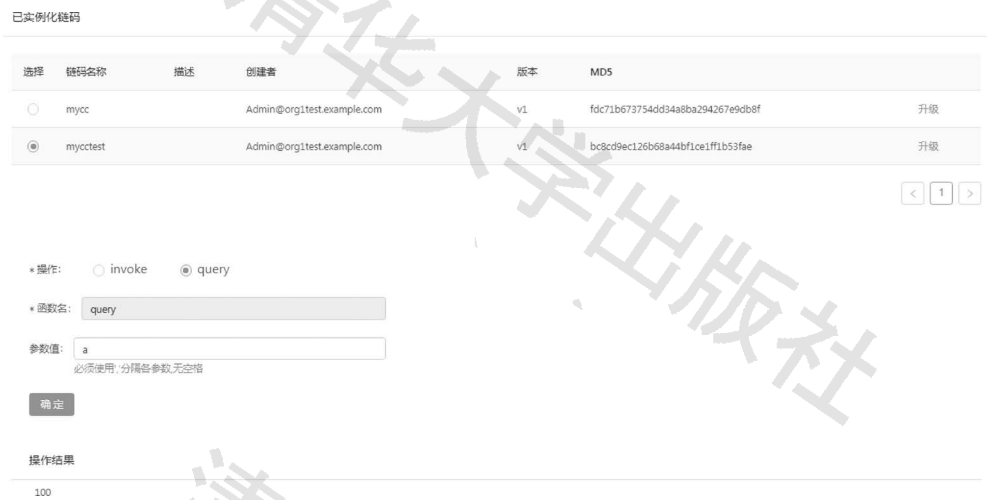


图 8-74　通道管理-链码查询(1)

图 8-75　通道管理-链码查询(2)

8.2.7 交易执行

查询到正确的值后，尝试进行 invoke 操作。

从转账示例中，分析得知 invoke 主要进行了以下操作：

（1）校验参数个数必须为 3，并将 3 个参数分别赋值给了提前声明的变量 A、B、X。

（2）分别对变量 A、B 执行查询账本的操作，将值取出后，赋值给变量 Aval 和 Bval。

（3）执行 Aval＝Aval － X 及 Bval＝Bval＋X 的操作。

（4）将修改后的 Aval、Bval 分别作为值存入 A、B 账户中。

```
func (t * SimpleChaincode) invoke(stub shim.ChaincodeStubInterface, args [ ]string) pb.
Response {
    var A, B string              //记账对象
    var Aval, Bval int           //持有资产
    var X int                    //交易资产额
    var err error
    if len(args) != 3 {
        return shim.Error("Incorrect number of arguments. Expecting 3")
    }
    A = args[0]
    B = args[1]
    Avalbytes, err : = stub.GetState(A)
    …
    Aval, _ = strconv.Atoi(string(Avalbytes))
    Bvalbytes, err : = stub.GetState(B)
    …
    Bval, _ = strconv.Atoi(string(Bvalbytes))
    X, err = strconv.Atoi(args[2])
    …
    Aval = Aval - X
    Bval = Bval + X
    err = stub.PutState(A, []byte(strconv.Itoa(Aval)))
    …
    err = stub.PutState(B, []byte(strconv.Itoa(Bval)))
    …
    return shim.Success(nil)
}
```

由此链码逻辑可知，invoke 主要实现的就是从账户 A 给账户 B 转账的操作。

【操作】选择 invoke，【函数名】输入“invoke”，【参数值】输入“a，b，9”，单击【确定】按钮后，会弹出“执行成功”或“执行失败”的提示信息，如图 8-76 所示。

注意：这里的操作选择 invoke 不是因为函数名是 invoke，而是因为该函数是一笔交易，修改了账本数据，要将其作为账本数据添加到区块链中；而 query 操作仅仅执行查询操作，不会生成交易 id，更不会增加区块到账本中。

执行完成后，再执行查询操作进行检查，看看是否执行成功，如图 8-77 所示。

查询结果显示，a 的余额变成了 91，而 b 的余额变成了 209，说明 a 给 b 转账 9 操作成功。

图 8-76　通道管理－链码操作

图 8-77　通道管理-链码查询(3)

8.3　习　　题

1. 在 Fabric 网络的部署过程中，需要执行哪些操作？

2. 假设有 Org1、Org2、Org3、Org4 四家公司，这四家公司均加入了同一个网络、同一个 channel。经几家公司商定，背书策略为：需要 Org1、Org2 同时签名，或者 Org3、Org4 同时签名，或者 Org1、Org3 同时签名。请问如何配置背书策略？

3. 执行链码实例化、链码 invoke 调用、链码升级这三个操作，分别会执行链码中的哪个方法？链码升级成功后，账本中的内容会发生什么变化？

4. 请思考并回答：编写链码时，链码的哪些部分是固定不可缺少的？这几个部分的作用分别是什么？

5. 一家物流公司的快递信息想要上链，简单的上链数据定义如下：

```
{
    快递单号            string
    收件人姓名          string
    收件人电话          string
    收件人地址          string
    寄件人姓名          string
    寄件人电话          string
    …
}
```

请思考：如果最终实现的链码只需要通过快递单号来查询订单信息，在部署 Fabric 网络时应该使用什么数据库？如果最终实现的链码不仅需要通过快递单号来查询订单信息，还需要通过收件人电话、寄件人电话等信息来查询历史记录，那么需要使用什么数据库？为什么？

6. 请详细描述从 client 端发起交易到最终账本记账成功的整个交易执行的流转过程。

附录 A

区块链常用英文短语

附录 A　区块链常用英文短语

英文缩写	英 文 全 称	中 文 含 义
DApp	Decentralized Application	去中心化应用
EVM	Ethereum Virtual Machine	以太坊虚拟机
DLT	Distributed Ledger Technologies	分布式账本
EOS	Enterprise Operation System	企业操作系统
P2P	peer-to-peer	点对点
	Consensus	共识
PoW	Proof of Work	工作证明
PoS	Proof of Stake	权益证明机制
DPoS	Delegated Proof of Stake	委任权益证明
PBFT	Practical Byzantine Fault Tolerance	实用拜占庭容错算法
RAFT	Replicated And Fault Tolerant	管理复制日志的一致性算法
CFT	Crash Fault Tolerance	基于故障容错，非拜占庭容错
DAG	Directed Acyclic Graph	有向无环图
IoT	Internet of Things	物联网
UTXO	Unspent Transaction Output	未花费的交易输出，比特币交易生成及验证的一个核心概念
SPV	Simplified Payment Verification	简单支付验证
RLP	Recursive Length Prefix Encoding	递归长度前缀编码
	double-spends	双重支付
	smart contract	智能合约
	smart contract code	智能合约代码
	smart legal contracts	智能法律合约
MVCC	Multiversion Concurrency Control	多版本并发控制
PKI	Public Key Infrastructure	公钥基础设施
MSP	Membership Service Provider	联盟链成员的证书管理
DPDK	Data Plane Development Kit	数据平面开发套件
TPS	Transaction Processing Systems	事务处理系统
KISS	Keep It Simple & Stupid	简单就是美（懒人原则）
RPC	Remote Procedure Call	远程过程调用
QPS	Queries-per-second	每秒查询率
CAP	Consistency、Availability、Partition tolerance	一致性、可用性、分区容错性
ECC	Elliptic Curves Cryptography	椭圆曲线加密算法
CRC32	Cyclic Redundancy Check 32	循环冗余校验

续表

英文缩写	英 文 全 称	中 文 含 义
IaaS	Infrastructure as a Service	基础设施即服务
PaaS	Platform as a Service	平台即服务
SaaS	Software as a Service	软件即服务
BaaS	Backend as a Service	后端即服务
BaaS	Blockchain as a Service	区块链即服务
OS	Operation System	操作系统
MPT	Merkle Patricia Tree	经过改良的、融合了默克尔树和前缀树两种树结构优点的数据结构
P2SH	Pay-to-Script Hash	支付到脚本哈希
P2PKH	Pay-to-Public-Key-Hash	支付到公钥哈希
P2PK	Pay-to-Public-Key	支付到公钥
DCEP	Digital Currency Electronic Payment	数字货币和电子支付
SWIFT	Society for Worldwide Interbank Financial Telecommunications	环球同业银行金融电信协会
ICO	Initial Crypto-token Offering	首次加密代币发行
	double-spend attack	双花攻击

附录 B

比特币相关术语

比特币地址：如同物理地址或者电子邮件地址，是比特币支付时唯一需要提供的信息。

账户：在总账中的记录，由其地址索引，总账包含有关该账户的状态的完整数据。

bit：bit 是比特币的一个常用单位，1 000 000 bits 等于 1 个比特币。

比特币：首字母大写的 Bitcoin 用来表示比特币的概念或整个比特币网络本身，而首字母小写的 bitcoin 则表示一个记账单位。

块链：块链是一个按时间顺序排列的比特币交易公共记录，由所有比特币用户共享。

区块：一个区块是块链中的一条记录，包含并确认待处理的交易。平均每约 10 分钟就有一个包含交易的新区块通过挖矿的方式添加到块链中。

BTC：用于标示一个比特币的常用单位。

交易：一个交易是一个文档，授权与区块链相关的一些特定的动作。

交易确认：一笔交易已经被网络处理且不太可能被撤销。当交易被包含到一个区块时会收到一个确认，后续的每个区块都会增加一个确认。

密码学：在比特币中，用来保证任何人都不可能使用他人钱包里的资金，或破坏块链。

双重消费（双重花费）：将比特币同时支付给两个不同的收款人。

哈希率：衡量比特币网络处理能力的测量单位。

工作证明：在区块中的散列值必须比某个目标值小，散列值是伪随机的。在分布式系统中任何人都可以产生区块，为了防止网络中区块泛滥，需要进行大量试验和验证，使得产生一个区块非常艰难。

随机数：为了满足工作证明的条件来进行调整。

比特币挖矿：利用计算机硬件为比特币网络做数学计算、进行交易确认和提高安全性的过程。作为对他们服务的奖励，矿工可以得到他们所确认的交易中包含的手续费，以及新创建的比特币。

对等式网络：允许单个节点与其他节点直接交互，从而实现整个系统像有组织的集体一样运作。

私钥：一个证明用户有权从一个特定的钱包消费比特币的保密数据块，是通过一个密码学签名来实现的。

密码学签名：一个让用户可以证明自身所有权的数学机制。对于比特币来说，一个比特币钱包和它的私钥通过一些“数学魔法”关联到一起。

比特币钱包：实体钱包在比特币网络中的等价物。包含了用户的私钥，可以让用户消费块链中分配给钱包的比特币。

附录 C

以太坊相关术语

计算上不可行：一个处理被称为是计算上不可行，如果有人想有兴趣完成一个处理但是需要采取一种不切实际的长的时间来做到这一点的（如几十亿年）。通常，2 的 80 次方的计算步骤被认为是计算上不可行的下限。

散列：一个散列函数（或散列算法）是一个处理，依靠这个处理，一个文档（比如一个数据块或文件）被加工成看起来完全是随机的小片数据（通常为 32 个字节），从中没有意义的数据可以被复原为文档，并且最重要的性能是散列一个特定的文档的结果总是一样的。

加密：与被称为钥匙的短字符串的数据相结合，对文档（明文）所进行的处理。加密会产生一个输出（密文），这个密文可以被其他掌握这个钥匙的人“解密”回原来的明文，但是对于没有掌握钥匙的人来说是解密是费解的且计算上不可行。

序列化：将一个数据结构转换成一个字节序列的过程。

幽灵（Ghost）：幽灵是一个协议，通过这个协议，区块可以包含不只是他们父块的散列值，也散列父块的父块的其他子块（被称为叔块）的陈腐区块。

叔块：是父区块的父区块的子区块，但不是自个的父区块，或更一般的说是祖先的子区块，但不是自己的祖先。

账户随机数：每个账号的交易计数，防止重放攻击。

EVM 代码：以太坊虚拟机代码，以太坊的区块链可以包含的编程语言的代码。

消息：一种由 EVM 代码从一个账户发送到另一个账户的“虚拟交易”。

合约：一个包含并且受 EVM 的代码控制的账户。合约不能通过私钥直接进行控制，除非被编译成 EVM 代码，一旦合约被发行就没有所有者。

以太（Ether）：以太坊网络的内部基础的加密代币。以太是用来支付交易和以太坊交易的计算费用。

附录 D

开源区块链及其地址

附录 D　开源区块链及其地址

开 源 体 系	开 源 地 址
比特币(BitCoin)	https://github.com/bitcoin/bitcoin
Ethereum	https://github.com/ethereum/go-ethereum
EOS	https://github.com/EOSIO/eos
Hyperledger Fabric	https://github.com/hyperledger/fabric
Corda	https://github.com/corda/corda
Quorum	https://github.com/jpmorganchase/quorum
Bitshares	https://github.com/bitshares
瑞波 ripple	https://github.com/ripple/rippled
Factom	https://github.com/FactomProject/FactomCode
百度超级链	https://github.com/xuperchain/xuperchain
京东智臻链	https://github.com/blockchain-jd-com/jdchain

参 考 文 献

[1] 百度搜索公司，百度区块链实验室，百度营销研究院. 百度区块链白皮书 V1.0[EB/OL]. (2018-09-26).

[2] 京东数科智臻链. 京东区块链技术实践白皮书 2019[EB/OL]. 2019 年 4 月.

[3] 腾讯区块链. 腾讯_TrustSQL_developer_installation_guide[EB/OL]. 2018 年 2 月.

[4] 区块链服务网络发展联盟. 区块链服务网络技术白皮书 v1.0.0. 2020 年 4 月 25 日.

[5] 安德烈亚斯 M. 安东诺普洛斯. 精通比特币[M]. 南京：东南大学出版社，2018.

[6] 廖雪峰. Git 教程[EB/OL]. [2020-07-10]. https://www.liaoxuefeng.com/wiki/896043488029600.

[7] 蔚 1. 一起学：以太坊智能合约开发[EB/OL]. (2018-07-03)[2020-06-15]. https://blog.csdn.net/valada/article/details/80892582.