

## 目录

- 预安装环境
  - 1、CentOS环境
    - 1.1、安装Docker
    - 1.2、安装Docker-compose
    - 1.3、安装Go
    - 1.4、安装Jq
  - 2、Ubuntu环境
  - 2.1、安装环境
- 基本部署
  - 1、架构概述
  - 2、身份验证
  - 3、使用OpenSSL构建CA证书
    - 3.1、CA自签名证书
    - 3.2、签发用户证书
    - 3.3、代理数据下载
  - 4、MSP成员服务提供器
  - 5、启动最小的Fabric网络
    - 5.1、启动排序节点
    - 5.2、启动对等节点
    - 5.3、查看创世块文件
  - 6、通道初始化与节点加入
  - 7、链表开发
- 共识机制之Raft算法
- 1、Raft算法介绍
  - 1.1、Raft选举流程
  - 1.2、数据如何分布式的去更新
  - 1.3、总结
    - 1.3.1、主节点选举（领导选举）
    - 1.3.2、日志复制（数据更新）
- 2、Raft算法的简单实现
  - 2.1、Raft对象实现代码
  - 2.2、基本常量实现代码
  - 2.3、主节点信息对象实现代码
  - 2.4、选举算法实现代码

- 2.5、初始化函数实现代码
  - 2.6、选举核心算法实现代码
  - 2.7、心跳机制实现代码
  - 2.8、主函数实现代码
  - 2.9、运行结果
  - 2.10、代码下载
  - 安全机制之国密
- 

## 预安装环境

---

**Fabric** 官方推荐的开发环境是基于 **Docker** 搭建的,需要安装以下开发环境: **Docker**、**Docker-compose**、**Go**、**Python**、**Git**、**Jq**。在连接服务器推荐使**MobaXterm**工具。

### 1、CentOS环境

安装git只需要执行`yum install -y git`,执行完命令之后设置两个额外的参数:

```
git config --global core.autocrlf false
git config --global core.longpaths true
```

#### 1.1、安装Docker

初始化需要安装**yum-utils**,执行命令: `sudo yum install -y yum-utils`

```
sudo yum-config-manager --add-repo
https://download.docker.com/linux/centos/docker-ce.repo
sudo yum install docker-ce docker-ce-cli containerd.io
sudo systemctl start docker
```

如果想要给非root用户使用,但是不用**sudo**命令来执行**Docker**,可以执行:

```
# 将username换成自己用户
sudo usermod -aG docker username
sudo systemctl restart docker

# 将用户加入该group组中，一定要执行改命令，不然用户无法使用sudo权限
sudo gpasswd -a username docker

# 切换到docker组
sudo newgrp - docker
```

为了加速代理Docker Images: `sudo curl -L`

```
https://oss.jtyoui.com/docker/daemon.json -o /etc/docker/daemon.json && sudo
systemctl daemon-reload && sudo systemctl restart docker
```

## 1.2、安装Docker-compose

安装compose的前提是先安装好docker，并且能正常启动。

```
# 如果这条语句执行失败，可以执行下面一条等效语句
sudo curl -L
"https://github.com/docker/compose/releases/download/1.29.0/docker-
compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose

# 失败执行
sudo curl -L "https://oss.jtyoui.com/docker/docker-compose-Linux-
x86_64" -o /usr/local/bin/docker-compose

sudo chmod +x /usr/local/bin/docker-compose
sudo ln -s /usr/local/bin/docker-compose /usr/bin/docker-compose
```

## 1.3、安装Go

安装Go编程语言，默认安装的位置在/usr/bin/go

```
sudo curl https://oss.jtyoui.com/software/go1.16.3.linux-
amd64.tar.gz -o /home/go-1.16.3.tar.gz
sudo tar -zxvf /home/go-1.16.3.tar.gz -C /usr/bin && sudo rm
/home/go-1.16.3.tar.gz
```

配置环境变量

```
echo export GOPATH=/usr/bin/go >> ~/.bashrc
echo export PATH=\$PATH:\$GOPATH/bin >> ~/.bashrc
source ~/.bashrc

# 代理GO环境
go env -w GO111MODULE=on
go env -w GOPROXY=https://goproxy.cn,direct

# 跳过私有库
go env -w GOPRIVATE=*.gitlab.com,*.gitee.com
```

输入go出现信息说明安装成功。

如果在运行go mod vendor时，提示Get <https://sum.golang.org/lookup/xxxxxx>: dial tcp 216.58.200.49:443: i/o timeout，则是因为Go 1.13设置了默认的GOSUMDB=sum.golang.org，这个网站是被墙了的，用于验证包的有效性，可以通过如下命令关闭：`go env -w GOSUMDB=off`。可以设置下面是专门为国内提供的sum 验证服务。

```
go env -w GOSUMDB="sum.golang.google.cn"
go env -w GOSUMDB="sum.golang.org"
```

## 1.4、安装Jq

jq 是一种轻巧灵活的命令行 JSON 处理器

```
sudo curl https://oss.jtyoui.com/software/jq-linux64 -o
/usr/local/bin/jq
sudo chmod +x /usr/local/bin/jq
```

## 2、Ubuntu环境

### 2.1、安装环境

```
# 安装docker go jq
apt install docker.io go jq

# 安装docker-compose
sudo curl -L
"https://github.com/docker/compose/releases/download/1.29.0/docker-
compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
sudo chmod +x /usr/local/bin/docker-compose
sudo ln -s /usr/local/bin/docker-compose /usr/bin/docker-compose
```

## 基本部署

基本介绍：HyperLedger Fabric（后文简称Fabric）是其中发展最好的一个**企业级区块链平台**，最初由Digital Asset和IBM 贡献，目前已经应用于沃尔玛的食物溯源链和马士基的物流跟踪链中，代表了当下企业级区块链应用的最高水平。可以认为Fabric是一种联盟链平台，它适合构建跨越多个企业边界的去中心化应用。由于Fabric项目的目标是应用于相对可信的企业联盟环境，因此其设计思路与比特币、以太坊 等公链平台有明显的差异。Fabric借鉴了区块链的数据结构，但引入了相当多的身份验证与 权限控制机制，以及数据隐私保护机制，以适应企业级应用的要求。同时由于企业联盟环境 要比完全开放的公链环境可控，因此Fabric没有强调其共识体系对拜占庭容错的实现，允许使用 非拜占庭容错算法建立共识，从而达到相当实用的交易吞吐量。

Fabric定位于企业级的分布式账本技术，它的主要目的是为跨越多个企业边界的活动提供不可篡改的分布式记账平台。例如在食物溯源应用中，为了让消费者可以了解到所购买食物是否安全，就必须将从农场到加工商、分销商、零售商乃至监管机构等各个环节的检验与放行信息记录到区块链上，以保证溯源信息的透明与可信。

因此Fabric是一种联盟链（Consortium BlockChain），它适合在多个企业间实现分布式记账，这一定位使Fabric的实现与以太坊这样的公链有了明显的差异：

1. **分布式账本 vs. 区块链**：分布式账本是比区块链更加宽泛的概念，可以认为区块链只是分布式账本的一种实现技术，其他的分布式账本实现还包括哈希图等。
2. **去中心化 vs. 分布式**：Fabric淡化了去中心化，而以分布式代替，这一思路带来了系统设计 与实现上的巨大影响。例如，在Fabric中，采用中心化的CA机制来发放证书，只有持有有效证书的节点和用户才可以访问区块链 上的账本数据。因此

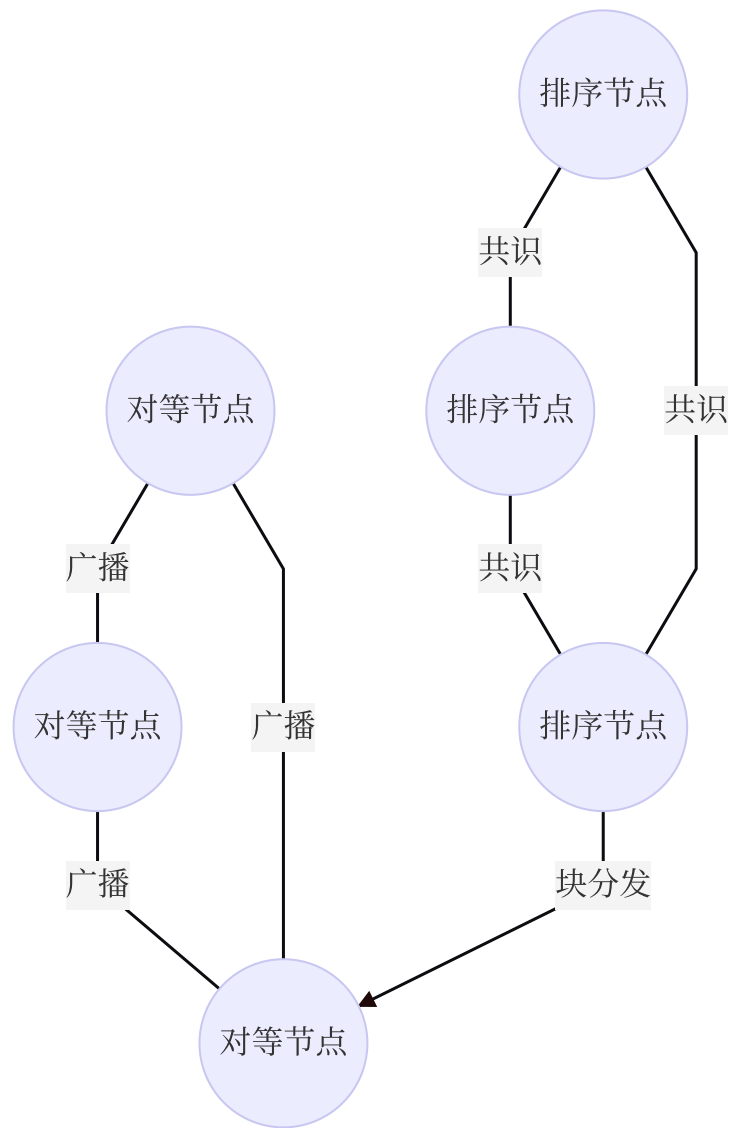
Fabric是许可制的区块链，这与不需要许可的以太坊这样的公链形成了鲜明的对比。

3. **拜占庭容错 vs. 崩溃容错**：由于采用许可机制，Fabric也淡化了对不可信环境下共识达成的依赖性，而假设联盟链中的企业有可能是值得信赖的，因此并不依赖于工作量证明这样的拜占庭容错算法—虽然Fabric模块化的设计可以支持引入不同的共识算法实现，但目前的产品化方案是Kafka共识，它显然是不能对抗拜占庭错误的—不过对不可信环境支持的淡化处理有利于提高交易的吞吐量，这对于企业级应用也是有益的。
4. **数据隐私保护**：Fabric强化了隐私保护能力。例如Fabric支持在同一套企业网络上建立多个不同的**通道**，每一个通道都有自己的区块链和访问控制，彼此互不影响，这有利于复用基础设施，例如不同企业间的销售部门可以建立一个通道来分享市场数据，而这些企业间的研发部门可以建立另一个通道来分享技术数据。

## 1、架构概述

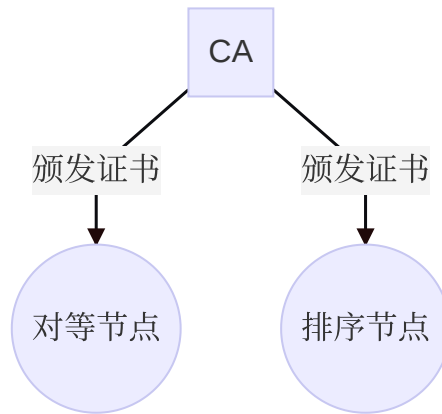
Fabric的架构中包含相当多的细节，为了便于理解，我们先观察最基本的两个组成部分：

1. 对等节点（Peer）：对等节点负责交易区块链的保存以及状态的更新
2. 排序节点（Orderer）排序器则负责确定应用所提交的交易的先后执行顺序，并将交易打包成块后分发给主导对等节点，再由主导对等节点通过gossip协议扩散至其他对等节点，对等节点进行验证后执行交易从而更新其状态。在1.4.4版本以后，Fabric提供了三种排序器实现：一个用于开发的单节点Solo排序器，一个基于Kafka和Zookeeper实现的崩溃容错排序器，以及一个基于Etcd的Raft共识排序器。



## 2、身份验证

和大多数区块链一样，Fabric也采用非对称加密技术来标识用户的身份，区别在于 Fabric 取消了用户自己生成身份标识的权利，而采用中心化的CA来统一颁发证书。



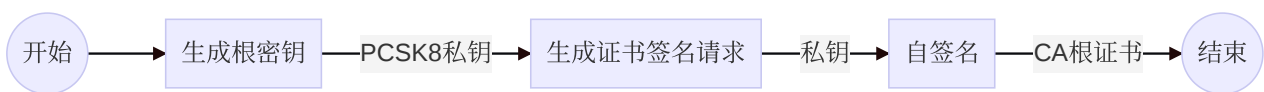
在一个Fabric网络中可以使用单一的CA，也可以存在多个不同的CA，例如每个参与机构都由自己的CA为自己组织内的节点和用户颁发证书，并单独为排序节点 建立一个CA。CA颁发的证书中包含了用户公钥和唯一标识名称等信息，同时也包含了CA的签名，因此第三方可以验证用户持有的证书是否由CA签发，从而验证用户的身份。

### 3、使用OpenSSL构建CA证书

我们可以使用OpenSSL来建立一个CA，并为Peer、Orderer和用户颁发证书。由于CA需要为自己签名，因此我们首先需要生成CA的自签名证书。

#### 3.1、CA自签名证书

生成CA自签名证书主要包括三个环节



1. **创建根密钥**：首先使用Ecpkparam子命令选择prime256v1椭圆曲线参数创建根密钥，这是Fabric中使用的非对称加密算法

```
openssl ecparam -genkey \                                # 生成椭圆曲线私钥
                                     -name prime256v1 \      # 使用prime256v1
椭圆曲线参数
                                     -out ca-key_.pem         # 生成的EC私钥存
入ca-key_.pem文件

# 执行命令
```



```

root@jtyoui-PC:/mnt# openssl ecparam -genkey -name
prime256v1 -out ca-key_.pem
root@jtyoui-PC:/mnt# ls
ca-key_.pem
root@jtyoui-PC:/mnt# cat ca-key_.pem
-----BEGIN EC PARAMETERS-----
BggqhkJOPQMBBw==
-----END EC PARAMETERS-----
-----BEGIN EC PRIVATE KEY-----
MHcCAQEEINvGtiRUFFmDF8mZITfXVcBKGCWGYpNkdJKKtvGvVaRWoAoGCCqG
SM49
AwEHoUQDQgAEp/5H83fZa8KJGND6315SgkWwjnhUzKBu5+C8wZRZ2Hc1K09F
jmTd
r/rmUtjv8WfwIVei4Zxu0IfHpoL8xZ1N2w==
-----END EC PRIVATE KEY-----

```

2. **转换为不加密的PKCS8私钥**：然后将上述输出文件转换为不加密的PKCS8格式私钥并保存在ca-key.pem文件中，以便后续在其他命令中利用私钥签名。

```

openssl pkcs8 -topk8 \                                # 将输入私钥转换为
PKCS8格式
                                -nocrypt \              # 不加密输出的
PKCS8私钥
                                -in ca-key_.pem \         # 输入私钥为ca-
key_.pem
                                -out ca-key.pem           # 输出存入ca-
key.pem

# 执行命令
root@jtyoui-PC:/mnt# openssl pkcs8 -topk8 -nocrypt -in ca-
key_.pem -out ca-key.pem
root@jtyoui-PC:/mnt# ls
ca-key.pem  ca-key_.pem
root@jtyoui-PC:/mnt# cat ca-key.pem
-----BEGIN PRIVATE KEY-----
MIGHAgEAMBMGBYqGSM49AgEGCCqGSM49AwEHBG0wawIBAQQg28a2JFQUWYMX
yZkh
N9dVwEoYJYbI+eR0koq28a9VpFahRANCAASn/kfzd9lrwokY13rfXlKCRbC0
eFTM
oG7n4LzBlFnYdzUrT0W0ZN2v+uZS20/xZ/AhV6LhnG44h8emgvzFnU3b
-----END PRIVATE KEY-----

```

### 3. 利用PKCS8私钥生成证书签名请求：其中的配置文件 CA.conf中声明了CSR相关信息，例如唯一标识名称、启用的扩展等信息。

```
openssl req -new \                                # 生成证书签名请求
            -key ca-key.pem \                      # 使用ca-key.pem
中的私钥
            -out ca-csr.pem \                      # 输出的CSR存入
ca-csr.pem文件
            -config CA.conf                        # 使用ca.conf中
的信息
```

```
# CA.conf配置文件解读：来源地址
http://www.jinbuguo.com/linux/openssl\_install.html

##### 证书请求配置的"基本字段", 其它附属字段都以它为入口 #####
[req]

# 此处的设置相当于在命令行上使用 -sha256 选项(对应于"-digest")。
default_md = sha256

# 保存生成的密钥文件的默认文件名。对应 -keyout 命令行选项。
default_keyfile = ca-key.pem

# 定义输入用户信息选项的"特征名称"字段名, 该扩展字段定义了多项用户信息。
distinguished_name = req_distinguished_name

# 如果设为no, 那么 req 指令将直接从配置文件中读取证书字段的信息, 而不是提示用户输入。
prompt = no

# 生成自签名根证书时要使用的证书扩展项字段名, 该扩展字段定义了要加入到根证书中的一系列 X.509v3 扩展项。
# 对应 -extensions 命令行选项。
x509_extensions = v3_ca

# 证书请求扩展的字段名, 该扩展字段定义了要加入到证书请求中的一系列 X.509v3 扩展项。
# 对应 -reqexts 命令行选项。
req_extensions = v3_req

# 为一些字段设置默认的字符串类型, 比如证书请求中的城市和组织名称。可能的取值和解释如下:
```

```
# default: 包含了 PrintableString, T61String, BMPString 三种类
型
# pkix   : 包含了 PrintableString, BMPString 两种类型
# utf8only: 只使用 UTF8 字符串。推荐使用这个, 这样可以完美的包含任意
字符。
# nombstr : 包含了 PrintableString, T61String 两种类型
string_mask = utf8only

# 如果设为yes, 那么命令行与配置文件中的字符串都将按照UTF-8编码看待。默
认值no表示仅使用ASCII编码。
# 对应 -utf8 命令行选项。
utf8 = yes

##### "特征名称"字段包含了用户的标识信息, 对应 -subj 命令行选项
#####
[req_distinguished_name]
countryName=CN # 必须是两字母国家代码
stateOrProvinceName="北京" # 省份或直辖市
localityName="北京市" # 城市
organizationName="中国农业科学院" # 组织名或公司名
organizationalUnitName="区块链实验室" #部门名称
commonName="Root" # 全限定域名或个人姓名
emailAddress="Root@qq.com" # Email地址

##### 为签发的证书设置扩展项 #####
[user_cert]
basicConstraints=critical,CA:false,pathlen:1
keyUsage=critical,keyCertSign,cRLSign
subjectKeyIdentifier=hash

##### 要加入到证书请求中的一系列 X.509v3 扩展项, 对应 -addext 命令
行选项 #####
[v3_req]
# 基本约束(该证书是否为CA证书)。“CA:FALSE”表示非CA证书(不能签发其他
证书的“叶子证书”)。
basicConstraints=critical,CA:true,pathlen:1
# 密钥用法: 防否认(nonRepudiation)、数字签名(digitalSignature)、
密钥加密(keyEncipherment)。
# 密钥协商(keyAgreement)、数据加密(dataEncipherment)、仅加密
(encipherOnly)、仅解密(decipherOnly)
keyUsage=critical,keyCertSign,cRLSign
# 使用者密钥标识符(根据RFC3280规范自动生成)
subjectKeyIdentifier=hash
```

```
##### 生成自签名证书(RootCA)时使用的 X.509v3 证书扩展项, 对应 -
addext 命令行选项 #####
[v3_ca]
# 基本约束(该证书是否为CA证书)。"CA:TRUE"表示是CA证书(可签发其他证书)。
# "pathlen:N"后缀表示允许签发下级CA证书的深度("0"表示禁止签发下级CA证书), "critical"表示关键扩展。
# 例如"critical,CA:TRUE, pathlen:0"表示禁止签发下级CA证书(仅能签发"叶子证书")。
keyUsage=critical,keyCertSign,cRLSign
# 密钥用法: 证书撤销列表签名(cRLSign)、证书签发(keyCertSign)、数字签名(digitalSignature)
basicConstraints=critical,CA:true,pathlen:1
# 使用者密钥标识符(根据RFC3280规范自动生成)
subjectKeyIdentifier=hash
```

```
# 配置文件解读完毕
# 执行命令
root@jtyoui-PC:/mnt# openssl req -new -key ca-key.pem -out
ca-csr.pem -config ./CA.conf
root@jtyoui-PC:/mnt# ls
CA.conf  ca-csr.pem  ca-key.pem  ca-key_.pem
root@jtyoui-PC:/mnt# cat ca-csr.pem
-----BEGIN CERTIFICATE REQUEST-----
MIIBrzCCAVUCAQAwwZ4xCzAJBgNVBAYTAkNOMQ8wDQYDVQQIDABljJfkuqwx
EjAQ
BgNVBACMCeWml+S6rOW4gjEeMBwGA1UECgwV5Lit5Zu95Yac5Lia56eR5a2m
6Zmi
MRswGQYDVQQQLDBLljLrlnZfpk77lrp7pqozlrqQxDzANBgNVBAMMBuW8oOS8
nzEc
MBoGCSqGSIb3DQEJARYNanR5b3VpQHfXlmNvbTBZMBMGBYqGSM49AgEGCCqG
SM49
AwEHA0IABKf+R/N32WvCiRjXet9eUoJFsI54VMygbufgvMGUwdh3NStPRY5k
3a/6
5lLY7/Fn8CFXouGcbjiHx6aC/MWdTdugVDBSBgkqhkiG9w0BCQ4xRTBDMBIG
A1Ud
EwEB/wQIMAYBAf8CAQEwDgYDVROPAQH/BAQDAgEGMB0GA1UdDgQWBBR60Eoi
7YyL
fS7ukUdSgqJYRXnMxDAKBggqhkiG9w0BCQ4xRTBDMBIG
zZSz
```

```

cjKP5IZJE5Q8Xi47vqKKZwIhAK2PSJQs72nrfZuGWH0XuWq0tTVBe88w30+F
r8M2
rv9r
-----END CERTIFICATE REQUEST-----

```

4. 由于我们是要生成自签名的根证书，因此直接使用根密钥签发请求文件ca-csr.pem，就得到了自签名的根证书ca-cert.pem。

```

openssl x509 -req \                                # 生成x509证书
            -extensions v3_ca \                    # 在证书中添加
ca_cert段声明的扩展
            -extfile CA.conf \                      # 包含扩展声明的文
件
            -in ca-csr.pem \                        # 输入文件为ca-
csr.pem
            -out ca-cert.pem \                      # 输出文件为ca-
cert.pem
            -signkey ca-key.pem \                  # 用来自签名的私钥
            -CAcreateserial \                      # 创建证书序列号
            -days 3650                             # 证书有效期为
3650天，大约10年

# 执行命令
root@jtyoui-PC:/mnt# openssl x509 -req -extensions v3_ca -
extfile CA.conf -in ca-csr.pem -out ca-cert.pem -signkey ca-
key.pem -CAcreateserial -days 3650
Signature ok
subject=C = CN, ST = \E5\8C\97\E4\BA\AC, L =
\E5\8C\97\E4\BA\AC\E5\B8\82, O =
\E4\B8\AD\E5\9B\BD\E5\86\9C\E4\B8\9A\E7\A7\91\E5\AD\A6\E9\99
\A2, OU =
\E5\8C\BA\E5\9D\97\E9\93\BE\E5\AE\9E\E9\AA\8C\E5\AE\A4, CN =
\E5\BC\A0\E4\BC\9F, emailAddress = jtyoui@qq.com
Getting Private key
root@jtyoui-PC:/mnt# ls
ca-cert.pem  CA.conf  ca-csr.pem  ca-key.pem  ca-key_.pem
root@jtyoui-PC:/mnt# cat ca-cert.pem
-----BEGIN CERTIFICATE-----
MIICHTCCAiugAwIBAgIUUV9zbtDoeqtZ9SE57UgGH+wgV9B8wCgYIKoZIzj0E
AwIw
gZ4xCzAJBgNVBAYTAKNOMQ8wDQYDVQQIDAb1jJfkuqwxEjAQBgNVBAcMCeWM
l+S6

```

```

r0W4gjEeMBwGA1UECgwV5Lit5Zu95Yac5Lia56eR5a2m6ZmiMRswGQYDVQQL
DBLl
jLrlnZfPk77lrp7pqozlrqQxDzANBgNVBAMMBuW8o0S8nzEcMBoGCSqGSib3
DQEJ
ARYNanR5b3VpQHFXLmNvbTAeFw0yMTA0MjEwNjQ5MzFaFw0zMTA0MTkwNjQ5
MzFa
MIGeMQswCQYDVQQGEWJDTjEPMA0GA1UECAwG5YyX5LqsMRIwEAYDVQQHDAnl
jJfk
uqzluIIxHjAcBgNVBAoMFeS4reWbveW Gn0S4muenkewtpumZojEbMBkGA1UE
CwwS
5Yy65Z2X6Z0+5a6e6aqM5a6kMQ8wDQYDVQQDDAb1vKDkvJ8xHDAaBgkqhkiG
9w0B
CQEWDP0ew91aUBxcS5jb20wWTATBgqhkiOPOQIBBggqhkiOPOQMBBwNCAASn
/kfz
d9lrwokY13rfXlKCRbCOeFTMoG7n4LzBlFnYdzUrT0W0ZN2v+uZS20/xZ/Ah
V6Lh
nG44h8emgvzFnU3bo0UwQzA0BgNVHQ8BAf8EBAMCAQYwEgYDVR0TAQH/BAgw
BgEB
/wIBATAdBgNVHQ4EFgQUetBKiu2Mi30u7pFHUoKiWEV5zMqWcGyIKoZIZj0E
AwID
SAAwRQIgStoxjs8rYfYQN+FGjRv0yRBpt83ai8vVDcCmkUdhjHsCIQCHow7d
7mQx
vleKuu80PUjRMilJI3Lr7/GY6Q597FEV+g==
-----END CERTIFICATE-----

```

5. CA颁发的证书中包含了用户公钥和唯一标识名称等信息，同时也包含了CA的签名，因此第三方可以验证用户持有的证书是否由CA签发，从而验证用户的身份。

```

root@jtyoui-PC:/mnt# openssl x509 -in ca-cert.pem -text
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:

57:dc:db:b4:3a:1e:aa:d6:7d:48:4e:7b:52:01:87:fb:08:15:f4:1f
        Signature Algorithm: ecdsa-with-SHA256
        Issuer: C = CN, ST = \E5\8C\97\E4\BA\AC, L =
\E5\8C\97\E4\BA\AC\E5\B8\82, O =
\E4\B8\AD\E5\9B\BD\E5\86\9C\E4\B8\9A\E7\A7\91\E5\AD\A6\E9\99
\A2, OU =
\E5\8C\BA\E5\9D\97\E9\93\BE\E5\AE\9E\E9\AA\8C\E5\AE\A4, CN =
\E5\BC\A0\E4\BC\9F, emailAddress = jtyoui@qq.com
        Validity

```

Not Before: Apr 21 06:49:31 2021 GMT

Not After : Apr 19 06:49:31 2031 GMT

Subject: C = CN, ST = \E5\8C\97\E4\BA\AC, L =  
\E5\8C\97\E4\BA\AC\E5\B8\82, O =  
\E4\B8\AD\E5\9B\BD\E5\86\9C\E4\B8\9A\E7\A7\91\E5\AD\A6\E9\99  
\A2, OU =  
\E5\8C\BA\E5\9D\97\E9\93\BE\E5\AE\9E\E9\AA\8C\E5\AE\A4, CN =  
\E5\BC\A0\E4\BC\9F, emailAddress = jtyoui@qq.com

Subject Public Key Info:

Public Key Algorithm: id-ecPublicKey

Public-Key: (256 bit)

pub:

04:a7:fe:47:f3:77:d9:6b:c2:89:18:d7:7a:df:5e:

52:82:45:b0:8e:78:54:cc:a0:6e:e7:e0:bc:c1:94:

59:d8:77:35:2b:4f:45:8e:64:dd:af:fa:e6:52:d8:

ef:f1:67:f0:21:57:a2:e1:9c:6e:38:87:c7:a6:82:

fc:c5:9d:4d:db

ASN1 OID: prime256v1

NIST CURVE: P-256

X509v3 extensions:

X509v3 Key Usage: critical

Certificate Sign, CRL Sign

X509v3 Basic Constraints: critical

CA:TRUE, pathlen:1

X509v3 Subject Key Identifier:

7A:D0:4A:22:ED:8C:8B:7D:2E:EE:91:47:52:82:A2:58:45:79:CC:C4

Signature Algorithm: ecdsa-with-SHA256

30:45:02:20:4a:da:31:8d:2f:2b:61:f6:10:37:e1:46:8d:1b:

ce:c9:10:69:b7:cd:da:8b:cb:d5:0d:c0:a6:91:47:61:8c:7b:

02:21:00:87:a3:0e:dd:ee:64:31:be:57:8a:ba:ef:0e:3d:48:

d1:32:29:49:23:72:eb:ef:f1:98:e9:0e:7d:ec:51:15:fa

-----BEGIN CERTIFICATE-----

MIICHTCCAiuGAWIBAgIUV9zbtDoeqtZ9SE57UgGH+wgV9B8wCgYIKoZIZj0E  
AwIw

```
gZ4xCzAJBgNVBAYTAkNOMQ8wDQYDVQQIDAb1jJfkuqwxEjAQBgNVBAcMCeWM
l+S6
r0W4gjEeMBwGA1UECgwV5Lit5Zu95Yac5Lia56eR5a2m6ZmiMRswGQYDVQQL
DBLl
jLrlnZfpk77lrp7pqozlrqQxDzANBgNVBAMMBuW8o0S8nzEcMBoGCSqGSib3
DQEJ
ARYNanR5b3VpQHfXlmNvbTAeFw0yMTA0MjEwNjQ5MzFaFw0zMTA0MTkwNjQ5
MzFa
MIGeMQswCQYDVQQGEWJDTjEPMA0GA1UECAwG5YyX5LqsMRIwEAYDVQQHDAnl
jJfk
uqzluIIxHjAcBgNVBAoMFeS4reWbveW Gn0S4muenkewtpumZojEbmBkGA1UE
CwwS
5Yy65Z2X6Z0+5a6e6aqM5a6kMQ8wDQYDVQQDDAb1vKDkvJ8xHDAaBgkqhkiG
9w0B
CQEWDP0ew91aUBxcS5jb20wWTATBgqhkhj0PQIBBggqhkhj0PQMBBwNCAASn
/kfz
d9lrwokY13rfXlKCRbCOeFTMoG7n4LzBlFnYdzUrT0W0ZN2v+uZS20/xZ/Ah
V6Lh
nG44h8emgvzFnU3bo0UwQzA0BgNVHQ8BAf8EBAMCAQYwEgYDVR0TAQH/BAgw
BgEB
/wIBATAdBgNVHQ4EFgQUetBKiu2Mi30u7pFHUoKiWEV5zMqWcGyYIKoZIZj0E
AwID
SAAwRQIgStoxjs8rYfYQN+FGjRv0yRBpt83ai8vVDcCmkUdhjHsCIQCHow7d
7mQx
vleKuu80PUjRMilJI3Lr7/GY6Q597FEV+g==
-----END CERTIFICATE-----
```

- Issuer** 字段标识签发此证书的CA机构的唯一标识名
- Subject** 字段标识此证书持有者的唯一标识名
- 在唯一标识名中最常用的字段是常用名CN字段

由于证书中包含用户公钥以及CA签名这些二进制数据，为了便于在网络环境中 传输交换，通常会编码为仅包含ASCII码的 **PEM** 格式，即Privacy-Enhanced Mail - 隐私增强邮件。



Certificate:

Data:

Version: 3 (0x2)

Serial Number:

87:cd:7d:ed:ed:f4:a6:ee:14:30:c5:60:1b:9c:d7:29

Signature Algorithm: ecdsa-with-SHA256

Issuer: C=US, ST=California, L=San Francisco, O=org1.example.com, CN=ca.org1.example.com

Validity

Not Before: Mar 16 07:50:00 2019 GMT

Not After : Mar 13 07:50:00 2029 GMT

Subject: C=US, ST=California, L=San Francisco, CN=Admin@org1.example.com

Subject Public Key Info:

Public Key Algorithm: id-ecPublicKey

Public-Key: (256 bit)

pub:

04:bf:3b:93:05:3b:bd:60:d0:12:5f:9a:0c:8c:b5:

0a:68:5c:35:6a:53:36:6a:58:47:c5:d3:86:0f:59:

59:8f:1f:55:47:4e:92:ae:b7:c8:ce:fb:b4:7c:af:

4a:eb:44:47:81:a4:38:a0:5e:06:5e:1a:e6:13:6b:

52:f0:a8:4e:bc

ASN1 OID: prime256v1

NIST CURVE: P-256

X509v3 extensions:

X509v3 Key Usage: critical

Digital Signature

X509v3 Basic Constraints: critical

CA:FALSE

X509v3 Authority Key Identifier:

keyid:7A:78:F2:5A:27:EA:DC:12:20:C2:30:D9:8B:06:CF:E2:89:24:9D:4C:56:00:97:B8:C7:A7:6E:5E:3A:CC:4D:05

Signature Algorithm: ecdsa-with-SHA256

30:45:02:21:00:fb:ec:4f:a7:5c:5e:8b:1d:35:11:b7:22:4a:

6e:f5:4e:62:72:51:26:68:f1:e8:dc:95:fd:d9:97:4d:ef:d6:

26:02:20:03:cf:87:bd:c8:54:7d:47:a5:78:2b:5c:9b:a4:96:

f8:6a:80:b7:fe:ac:a2:b1:38:40:cf:67:0e:40:26:3c:06

签发本证书的CA名称

本证书持有人的名称

公钥

CA的签名

PEM头, BEGIN <label>

-----BEGIN CERTIFICATE-----

MIICGjCCAcCgAwIBAgIRAIfnFe3t9KbuFDDFYBuc1ykwCgYIKoZIzj0EAwIwczEL  
MAKGA1UEBhMCVVMxEzARBGNVBAgTCKNhbg1mb3JuaWExFjAUBGNVBACDVBhbiBG  
cmFuY2l2Y28xGTAXBgNVBAoTEG9yZzEuZXhhbXBsZS5jb20xHDAaBgNVBAMTE2Nh  
Lm9yZzEuZXhhbXBsZS5jb20wHhcNMTEwMzE2MDc1MDAwWhcNMjkwMzE2MDc1MDAw  
WjBMBQswCQYDVQGEWjVUzETMBEGA1UECBMKQ2FsaWZvcn5pYTEwMBQGA1UEBxMN  
U2FuIEZyYW5jaXNjbzEfMB0GA1UEAwwWQWRtaW5Ab3JnMS5leGFtcGxlLmNvbTBZ  
MBMGBYqGSM49AgEGCCqGSM49AwEHA0IABL87kwU7vWDQE1+aDIy1CmhcNwPTNmpY  
R8XThg9ZwY8fvUdOkq63yM77tHyvSutER4GkOKBeB14a5hNrUvCoTryjTTBLMA4G  
A1UdDwEB/wQEAwIHGDAMBGNVHRMBAf8EAJAAMCqGA1UdIwQkMCKAIIHp48lon6twS  
IMIw2YsGz+KJJJ1MVGcXUmenbl46zE0FMAoGCCqGSM49BAMCA0gAMEUCIQD77E+n  
XF6LHTURtyJKbvVOYnJRjmx6NyV/dmXTe/WJgIgA8+HvchUfUeLeCtcm6SW+GqA  
t/6sorE4QM9nDkAmPAY=

-----END CERTIFICATE-----

PEM体, base64编码

PEM脚, END <label>

利用CA的证书可以验证一个指定的证书是否由该CA签发, 例如

```
openssl verify -CAfile ca-cert.pem xx-cert.pem
```

## 3.2、签发用户证书

一旦有了根证书，我们就可以签发用于节点、排序器或应用用户的证书了。前面的步骤基本一样，都是生成EC私钥、转换为PKCS8格式、生成证书签名请求，只是分别由申请用户和CA完成。

```
# 设置配置文件USER.conf

[req]
default_md = sha256
default_keyfile = ca-key.pem
distinguished_name = req_distinguished_name
prompt = no
x509_extensions = v3_ca
req_extensions = v3_req
string_mask = utf8only
utf8 = yes

[req_distinguished_name]
countryName=CN
stateOrProvinceName="北京"
localityName="北京市"
organizationName="中国农业科学院"
organizationalUnitName="区块链实验室"
commonName="张伟"
emailAddress="jtyoui@qq.com"

[user_cert]
basicConstraints=critical,CA:false,pathlen:1
keyUsage=critical,keyCertSign,cRLSign
subjectKeyIdentifier=hash

[v3_req]
basicConstraints=critical,CA:false,pathlen:1
keyUsage=critical,keyCertSign,cRLSign
subjectKeyIdentifier=hash

[v3_ca]
keyUsage=critical,keyCertSign,cRLSign
basicConstraints=critical,CA:false,pathlen:1
subjectKeyIdentifier=hash
```

执行下面代码，用CA证书来签发用户证书

```
openssl ecparam -name prime256v1 -genkey -out user-key_.pem
openssl pkcs8 -topk8 -nocrypt -in user-key_.pem -out user-key.pem
openssl req -new -key user-key.pem -out user-csr.pem -config
USER.conf
openssl x509 -req -extensions user_cert -extfile CA.conf -in user-
csr.pem -out user-cert.pem -CA ca-cert.pem -CAkey ca-key.pem -
CAcreateserial -days 3650
```

验证是否签发成功

```
root@jtyoui-PC:/mnt# openssl verify -CAfile ca-cert.pem user-
cert.pem
user-cert.pem: OK
```

### 3.3、代理数据下载

我把这些数据打包到OSS服务上，只需要下载运行sh文件即可

```
wget http://oss.jtyoui.com/fabric/ca.tar.gz
tar -zxvf ca.tar.gz && cd artifact
sh ./run.sh
```

## 4、MSP成员服务提供者

在Fabric中，判定机制由成员服务提供者（MSP）来实现。每一个需要保护的资源都有自己的MSP，并利用MSP实现从身份向角色的转换。例如，对于Orderer和Peer而言，它们都需要一个本地的MSP配置目录来建立其MSP实例，并且按照约定的目录结构来组织各种文件

- admincerts：该目录中的所有PEM证书对应有管理员角色
- cacerts：该目录的每个PEM证书都是CA根证书
- keystore：该目录中包含节点私钥的单一PEM证书
- signcerts：该目录中包含节点的X509证书
- crls：可选，该目录中包含所有已回收的证书
- intermediatecerts：可选，该目录中的PEM证书对应中间CA证书
- config.yaml：可选
- tlscacerts：可选，该目录中包含用于TLS的CA根证书
- tlsintermediatecerts：可选，该目录中包含用于TLS的CA中间证书

我们可以将之前使用openssl创建的证书和user私钥整理为MSP配置目录

```

mkdir -p msp/admincerts && cp ./artifact/user-cert.pem
msp/admincerts # 赋予user管理员角色
mkdir -p msp/cacerts && cp ./artifact/ca-cert.pem msp/cacerts
# CA证书
mkdir -p msp/signcerts && cp ./artifact/user-cert.pem
msp/signcerts # 当前身份为user
mkdir -p msp/keystore && cp ./artifact/user-key.pem msp/keystore
# 当前身份的私钥

root@jtyoui-PC:/mnt/artifact# tree msp
msp/
├── admincerts
│   └── user-cert.pem
├── cacerts
│   └── ca-cert.pem
├── keystore
│   └── user-key.pem
└── signcerts
    └── user-cert.pem

4 directories, 4 files

```

## 5、启动最小的Fabric网络

```

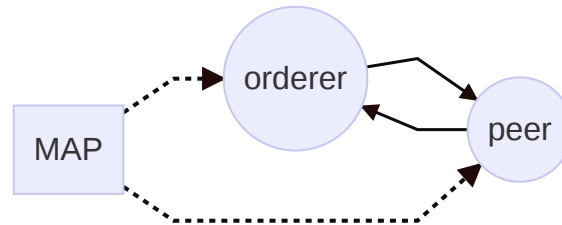
# 替换configtx.yaml文件
Profiles:
  SampleInsecureSolo:
    Orderer:
      OrdererType: solo
      Addresses:
        - 127.0.0.1:7050
    Consortiums:
      SampleConsortium:
        Organizations:
          - Name: SampleOrg
            ID: SampleOrg
            MSPDir: msp

  SampleSingleMSPChannel:
    consortium: SampleConsortium
    Application:
      Organizations:

```

```
- Name: SampleOrg
  ID: SampleOrg
  MSPDir: msp
```

有了MSP配置，我们就可以启动一个最简单的Fabric网络了



在这个网络中，只包含一个Orderer节点和一个Peer节点，而且两个节点使用同一个MPS配置。下载安装报，在官网下载：

```
# 官网下载
curl -L http://oss.jtyoui.com/fabric/hyperledger-fabric-linux-
amd64-1.4.11.tar.gz -o /mnt/fabric-1.4.11.tar.gz

# 解压
cd /mnt && tar -zxvf fabric-1.4.11.tar.gz
chmod +x bin/
cp -r bin/* /usr/local/bin/

root@jtyoui-PC:/mnt# configtxgen -version
configtxgen:
  Version: 1.4.11
  Commit SHA: 77facb430
  Go version: go1.14.12
  OS/Arch: linux/amd64
```

## 5.1、启动排序节点

启动排序节点之前，需要首先生成一个创世块，排序节点将利用这个创世块启动整个Fabric网络的系统通道。可以使用官方提供的configtxgen工具来生成Fabric网络系统通道的创世块。configtxgen依赖于配置文件configtx.yaml中定义的网络结构、访问策略等信息，并提供了不同的Profile以便应用于不同的场景，可以认为Profile是网络结构、访问策略等网络配置信息的组合套餐。

```
# 生成创世块并存入genesis.block
>> configtxgen -outputBlock genesis.block
# 该执行必须在官方config配置文件下
root@jtyoui-PC:/mnt/config# ls
configtx.yaml core.yaml genesis.block msp orderer.yaml
```

有了创世块，我们就可以启动排序节点了。排序节点程序依赖于配置文件 `orderer.yaml`。同样，Fabric提供了默认的配置文件，我们只需要做简单的个性化调整即可。

```
ORDERER_GENERAL_GENESISFILE=./genesis.block      # 创建块文件的路径
ORDERER_FILELEDGER_LOCATION=./data/orderer        # 系统通道账本（区块链）数据的路径

# 执行命令
>> ORDERER_GENERAL_GENESISPROFILE=SampleInsecureSolo
ORDERER_FILELEDGER_LOCATION=./data/orderer orderer
```

## 5.2、启动对等节点

类似的，对等节点依赖于配置文件 `core.yaml`，我们也可以在Farbic提供的 默认配置文件上进行自定义调整。

```
CORE_PEER_FILESYSTEMPATH=./peer                  # 通道账本（区块链）数据的路径
CORE_PEER_LISTENADDRESS=127.0.0.1:7051          # 节点监听地址
CORE_PEER_CHAINCODELISTENADDRESS=127.0.0.1:7052 # 节点链码服务监听地址
CORE_CHAINCODE_MODE=dev                          # 链码服务设置为开发模式

# 执行命令
>> CORE_PEER_FILESYSTEMPATH=./data/peer
CORE_PEER_LISTENADDRESS=127.0.0.1:7051
CORE_PEER_CHAINCODELISTENADDRESS=127.0.0.1:7052 peer node start --peer-chaincodedev
```

### 5.3、查看创世块文件

```
jtyoui@jtyoui-PC:/mnt/config$ tree data/
data/
├── orderer
│   ├── chains
│   │   └── test-system-channel-name
│   │       └── blockfile_000000
│   └── index
│       ├── 000001.log
│       ├── CURRENT
│       ├── LOCK
│       ├── LOG
│       └── MANIFEST-000000
└── peer
    ├── chaincodes
    ├── ledgersData
    │   ├── bookkeeper
    │   │   ├── 000001.log
    │   │   ├── CURRENT
    │   │   ├── LOCK
    │   │   ├── LOG
    │   │   └── MANIFEST-000000
    │   ├── chains
    │   │   └── index
    │   │       ├── 000001.log
    │   │       ├── CURRENT
    │   │       ├── LOCK
    │   │       ├── LOG
    │   │       └── MANIFEST-000000
    │   ├── configHistory
    │   │   ├── 000001.log
    │   │   ├── CURRENT
    │   │   ├── LOCK
    │   │   ├── LOG
    │   │   └── MANIFEST-000000
    │   ├── fileLock
    │   │   ├── 000001.log
    │   │   ├── CURRENT
    │   │   ├── LOCK
    │   │   ├── LOG
    │   │   └── MANIFEST-000000
    │   └── historyLeveldb
```

```
|   |─ 000001.log
|   |─ CURRENT
|   |─ LOCK
|   |─ LOG
|   └─ MANIFEST-000000
└─ ledgerProvider
|   |─ 000001.log
|   |─ CURRENT
|   |─ LOCK
|   |─ LOG
|   └─ MANIFEST-000000
└─ pvtdataStore
|   |─ 000001.log
|   |─ CURRENT
|   |─ LOCK
|   |─ LOG
|   └─ MANIFEST-000000
└─ stateLeveldb
    |─ 000001.log
    |─ CURRENT
    |─ LOCK
    |─ LOG
    └─ MANIFEST-000000
```

16 directories, 46 files

## 6、通道初始化与节点加入

当我们启动排序节点时，利用创世块也同时启动了保存整个网络配置的系统通道。

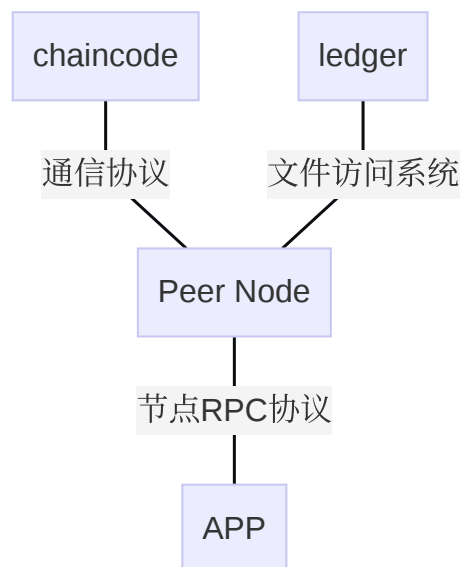


```
#!/bin/bash
configtxgen -outputCreateChannelTx data/ch1.tx -profile
SampleSingleMSPChannel -channelID ch1
peer channel create -f data/ch1.tx -c ch1 -o localhost:7050 --
outputBlock data/ch1.block
peer channel join -b data/ch1.block -o localhost:7050

## 查看加入的通道
root@jtyoui-PC:/mnt/config# peer channel list
2021-04-21 21:36:09.887 CST [channelCmd] InitCmdFactory -> INFO 001
Endorser and orderer connections initialized
Channels peers has joined:
ch1
```

## 7、链表开发

在Fabric中，智能合约被称为链码，链码负责定义区块链交易以及 状态，应用程序则通过链码来访问通道区块链。



与以太坊不同，Fabric并没有实现专门的智能合约虚拟机，链码实际上就是一个 操作系统进程，它通过链码通信协议与Peer节点通信，负责执行节点转发的调用请求，同时也利用节点提供的API访问通道账本。因此理论上可以使用任何图灵完备的语言开发Fabric链码，不过目前官方仅提供了NodeJS、Java和GO 的链码开发包。

# 共识机制之Raft算法

---

## 1、Raft算法介绍

---

我们现在有这样的需求：

1. 如何对大规模数据进行存储和计算？
  - a. 更好的机器
  - b. 更多的机器
2. 如何让跨网络的机器之间协调一致工作？
  - a. 状态的立刻一致性
  - b. 状态的最终一致性
3. 如何应对网络的不可靠以及节点的失效？
  - a. 可读写
  - b. 可读
  - c. 不可用

**诞生的原因：**过去，Paxos一直是分布式协议的标准，但是Paxos难于理解，更难以实现，Google的分布式锁系统Chubby作为Paxos实现曾经遭遇到很多坑。来自Stanford的新的分布式协议研究称为Raft，它是一个为真实世界应用建立的协议，主要注重协议的落地性和可理解性。

**什么是共识机制：**在了解Raft之前，我们先了解Consensus（共识）一致性这个概念，它是指多个服务器在状态达成一致，但是在一个分布式系统中，因为各种意外可能，有的服务器可能会崩溃或变得不可靠，它就不能和其他服务器达成一致状态。这样就需要一种Consensus协议，一致性协议是为了确保容错性，也就是即使系统中有一两个服务器宕机，也不会影响其处理过程。为了以容错方式达成一致，我们不可能要求所有服务器100%都达成一致状态，只要超过半数的大多数服务器达成一致就可以了，假设有N台服务器， $N/2 + 1$ 就超过半数，代表大多数了。Paxos和Raft都是为了实现Consensus一致性这个目标，这个过程如同选举一样，参选者需要说服大多数选民(服务器)投票给他，一旦选定后就跟随其操作。Paxos和Raft的区别在于选举的具体过程不同。

在Raft中，任何时候一个服务器可以扮演下面角色之一：

1. Leader: 处理所有客户端交互，日志复制等，一般一次只有一个Leader.
2. Follower: 类似选民，完全被动
3. Candidate: 类似Proposer律师，可以被选为一个新的领导人。

Raft阶段分为两个，首先是选举过程，然后在选举出来的领导人带领进行正常操作，比如日志复制等。下面用图示展示这个过程

## 1.1、Raft选举流程

任何一个服务器都可以成为一个候选者Candidate，它向其他服务器Follower发出要求选举自己的请求，一个分布式的节点至少需要3个，我们也三个节点为例子。（思考一下：为什么至少需要三个节点）

# 前面我们假设了有N台服务器， $N//2+1$ 台就可以实现选取，那么N至少需要多少？

1. 假设N=1台的时候，单节点

2. 假设N=2台的时候，全部同意才选举

3. 假设N=3台的时候，那么选取个数： $3//2+1=2$ ，选取成功。

至少需要三台。在思考一个问题，假如有X台服务器，宕机Y台，怎么才能保证选取成功？

$$X - (\lfloor \frac{X - Y}{2} \rfloor + 1) \geq 1$$

假如我们现在已经开启三台节点。如何开始选举，看如下流程图：

1. 每个节点都会随机生成150-300毫秒的超时机制，有三个问题：

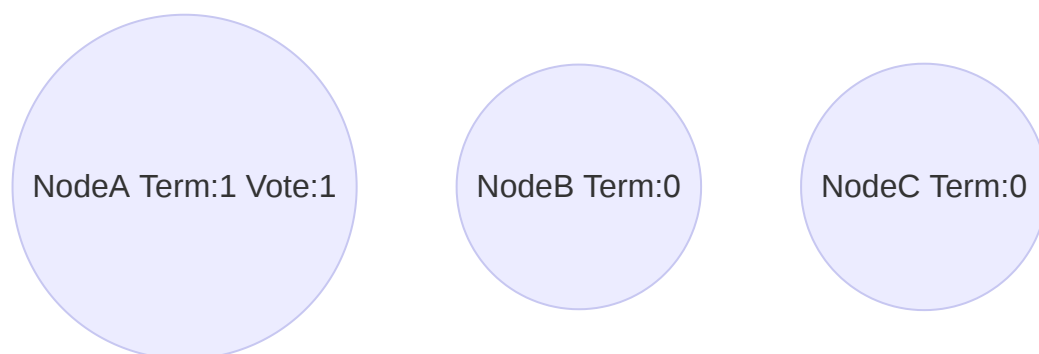
第一：为什么要设计超时机制？因为保证每个节点选举的性能。

第二：为什么要设计随机时间？如果不设计随机时间、每个节点都有可能同时请求选举，造成进程堵死现象。

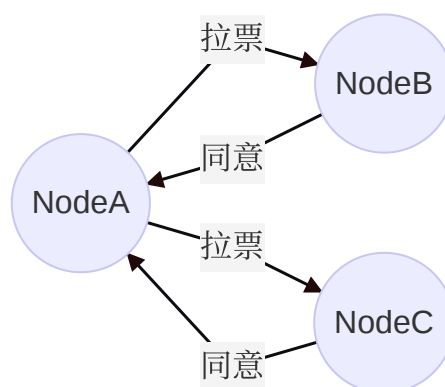
第三：为什么随机时间要设计到150-300毫秒之间？因为这个实验数据，这个段时间不快不慢最好。

2. 由于每一个节点的选举时间间隔不一样。必定有一个会被选举成候选人（candidate节点）。

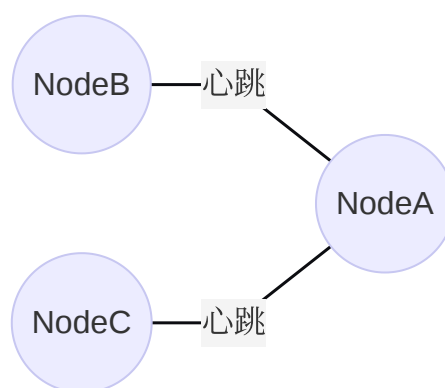
假如A节点被选中为候选人，那么它会给自己先投一票，即Vote==1。term是任期。



3. A节点向其它节点发送拉票。如果NodeB和NodeC都同意。并且回复。



4. 如果有 $N/2$ 节点都回复了A，那么所有节点都会变成跟随者，他们之间的连续用的心跳机制

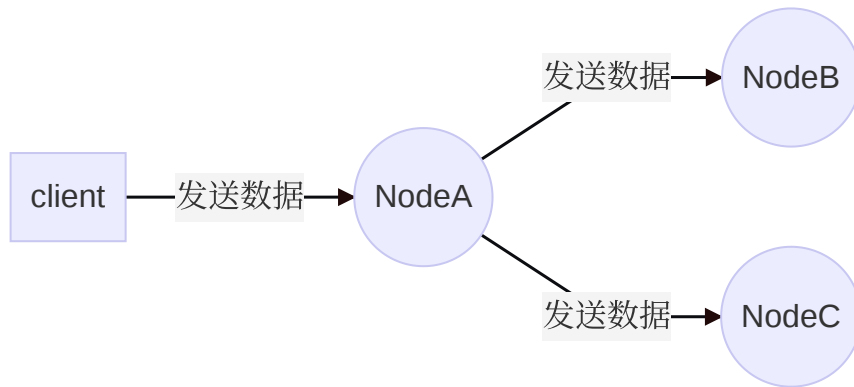


5. 如果当节点A死去，那么节点B和C就开始进行选举，又回到第一点内容。

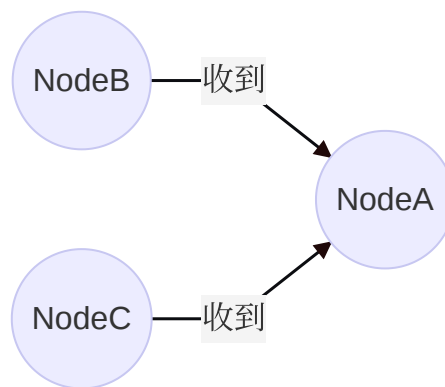
## 1.2、数据如何分布式的去更新

Raft数据传播主要是日志发送与更新。看下面的流程图我们分几步来讲述：

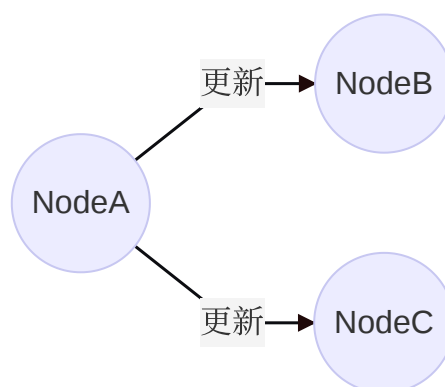
1. 当客户端发送数据给节点A的时候，节点A将数据同步给其它节点，同步日志方法是心跳机制，并且这些数据并没有更新。而是仅仅更新到日志中，节点A在等待其余节点的回复。



2. 当节点A发送数据给其它节点的时候，其它节点如果都收到信息，就会立即返回信息。



3. 其余节点都返回收到信号。当主节点收到全部信息之后，主节点A发出命令其余节点都更新数据。



注意：如果当出现网络断层现象。比如现在节点有5个，A、B、C、D、E节点，如果出现网络断层，A、B、C是一个局域网，D、E是一个局域网。我把A、B、C网络叫成组1，把D、E网络叫成组2。那么组1和组2出现网络延迟或者网络断掉的时候，那么组1就会在内部推荐出一个主节点，组2在内部推荐出一个主节点。当组1和组2网络恢复的时候，这个时候两个网络之间的主节点就会通讯。他们就会在次

选举，组1的节点有3个，组2的节点有2个，那么组2就会服从组1，推荐组1的节点。数据将会去同步组1的数据。

## 1.3、总结

Raft无法两大步，第一是主节点选举，第二是日志复制（数据同步），我们重新总结一下核心步骤。

### 1.3.1、主节点选举（领导选举）

1. Raft使用一种心跳机制来触发领导人选举
2. 当服务器程序启动的时候，所有节点都是跟随者状态
3. 如果一个跟随者在超时机制下，没有一个节点向它发送心跳机制，他就认为没有可用的主节点，自己就会发送拉票请求
4. 要开始选举的时候，跟随者会给自己当前的term加1，并且讲状态转为候选人，然后向集群其它服务器发送RPCs请求来获取投票请求
5. 候选人的状态一直保持到发生以下状态的时候：
  - a. 他自己赢得这次选举
  - b. 其它服务器成为领导人
  - c. 一段时间之后没有任何人获胜

### 1.3.2、日志复制（数据更新）

1. 当选举主节点的时候。它就开始接收客户端的请求，每一个请求会带有一个指令，然后被放回状态机中
2. 主节点把指令追加成一个日志记录，然后通过RPC并行的将数据发送给其它服务器，主节点会把记录放回到状态机中，然后把结果返回给客户端
3. 当跟随者宕机或者网络比较慢或者运行速度比较慢，主节点都会不停的重复发送数据，直到所有的跟随者都记录到改数据
4. Raft的日志记录要保证：两个有两个日志记录有相同的index和term，那么他们会保存一份
5. 主节点在一个特定的term和index下，只会创建一个日志记录

## 2、Raft算法的简单实现

---

我们用Go语言实现一个主节点选举。实现一个Raft对象， 里面包括了基本核心算法。下面是一个单机版的Raft简单实现流程。

## 2.1、Raft对象实现代码

```
// raft raft对象
type raft struct {
    mu          sync.Mutex // 锁
    me          int        // 节点编号
    currentTerm int        // 当前任期
    voteFor     int        // 为哪个节点投票
    state       int        // 状态：0为跟随者，1为候选人，2为选举者
    timeout     int        // 超时设置
    currentLeader int       // 当前的节点的领导是谁
    lastMessageTime int64     // 发送的最后的时间
    message     chan bool  // 节点间发送信息的通道
    electChan   chan bool  // 选举通道
    heartBeat   chan bool  // 心跳机制的通道
    heartBeatReturn chan bool // 返回心跳机制的通道
}
```

## 2.2、基本常量实现代码

```
const (
    RaftCount      = 5           // 节点个数
    NotVote        = -1          // 不投票
    NotLeader      = -1          // 无领导者
    Follower       = 0           // 跟随者
    Candidate      = 1           // 候选者
    Leader         = 2           // 领导者
    MaxTimeout     = 3000        // 最大超时时间
    MinTimeout     = 1500        // 最小超时时间
    Timeout        = MaxTimeout - MinTimeout // 超时时间
    IsGtNodeNumber = RaftCount / 2 // 大于选举节点的数量
    NotTerm        = 0           // 无任期
)
```

需要两个全局变量

```
// 记录主节点的基本信息，包括主节点的ID个任期，还有投票数量
var LeaderNode = leader{id: NotLeader, term: NotTerm, vote: 0}
var isBirthLeader = false
```

## 2.3、主节点信息对象实现代码

```
// leader 主节点对象
type leader struct {
    term int // 任期
    id    int //编号
    vote  int //投票数
}
```

基本信息完毕之后，我们需要定义两个核心的函数，一个是选举函数一个是心跳机制函数

## 2.4、选举算法实现代码

```
// election 选举函数
func (rf *raft) election() {
    for {
        timeout := randRange()
        rf.lastMessageTime = millisecond()
        select {
            case <-time.After(time.Duration(timeout) *
time.Millisecond):
                fmt.Printf("当前%d节点状态为: %s\n", rf.me,
getStatus(rf.state))
        }
        for !isBirthLeader {
            fmt.Println("=====选举中.....=====")
            isBirthLeader = rf.electionLeader(&LeaderNode)
            fmt.Println("=====选举成功=====")
        }
    }
}

// sendLeaderHeartBeat 心跳机制
func (rf *raft) sendLeaderHeartBeat() {
    for {
        select {
            case <-rf.heartBeat:
```



```

        rf.sendAppendEntriesImpl()
    }
}
}

```

核心函数基本上定义完毕！我们需要初始化函数

## 2.5、初始化函数实现代码

```

// initialize 初始化函数
func initialize(me int) {
    rf := &raft{
        me:           me,
        state:        Follower,
        voteFor:      NotVote,
        timeout:      Timeout,
        currentLeader: NotLeader,
        message:      make(chan bool),
        electChan:    make(chan bool),
        heartBeat:    make(chan bool),
        heartBeatReturn: make(chan bool),
    }
    rf.setTerm(NotTerm)
    rand.Seed(time.Now().UnixNano())
    go rf.election()
    go rf.sendLeaderHeartBeat()
}

```

核心选举函数里面需要如何去选举？

## 2.6、选举核心算法实现代码

```

// 如何选举主节点
func (rf *raft) electionLeader(lead *leader) bool {
    rf.mu.Lock()
    rf.becomeCandidate()
    rf.mu.Unlock()
    fmt.Println("开始选举领导者")
    for i := 0; i < RaftCount; i++ {
        if i != rf.me && lead.id == NotLeader {

```

```

        go func() {
            rf.electChan <- true
        }()
    }
}
vote := 1
for i := 0; i < RaftCount; i++ {
    select {
    case ok := <-rf.electChan:
        if ok {
            vote++
            if vote > IsGtNodeNumber {
                rf.mu.Lock()
                rf.becomeLeader()
                lead.id = rf.me
                lead.term++
                lead.vote = vote
                rf.mu.Unlock()
                rf.heartBeat <- true
                fmt.Println(rf.me, "成为了主节点, 开始发送心跳信号")
                return true
            }
        }
    }
}
return false
}

```

// 如何变成候选人

```

func (rf *raft) becomeCandidate() {
    rf.state = Candidate
    rf.setTerm(rf.currentTerm + 1)
    rf.voteFor = rf.me
    rf.currentLeader = NotLeader
}

```

// 变成主节点需要改变的属性

```

func (rf *raft) becomeLeader() {
    rf.state = Leader
    rf.currentLeader = rf.me
}

```

```

func millisecond() int64 {

```

```

        return time.Now().UnixNano() / int64(time.Millisecond)
    }

    func (rf *raft) setTerm(term int) {
        rf.currentTerm = term
    }

    func getStatus(status int) (stu string) {
        switch status {
        case Follower:
            stu = "跟随者"
        case Candidate:
            stu = "候选者"
        case Leader:
            stu = "领导者"
        default:
            stu = ""
        }
        return
    }

    func randRange() int64 {
        return rand.Int63n(Timeout) + MinTimeout
    }

```

核心函数心跳机制实现流程

## 2.7、心跳机制实现代码

```

// 监听心跳
func (rf *raft) sendLeaderHeartBeat() {
    for {
        select {
        case <-rf.heartBeat:
            rf.sendAppendEntriesImpl()
        }
    }
}

// 心跳实现部分
func (rf *raft) sendAppendEntriesImpl() {
    if rf.currentLeader == rf.me {

```

```

        successCount := 0
        for i := 0; i < RaftCount; i++ {
            if i != rf.me {
                go func() {
                    rf.heartBeatReturn <- true
                }()
            }
        }

        for i := 0; i < RaftCount; i++ {
            select {
            case ok := <-rf.heartBeatReturn:
                if ok {
                    successCount++
                    if successCount > RaftCount/2 {
                        fmt.Println("投票检查成功! 心跳检查正常")
                    }
                }
            }
        }
    }
}

```

## 2.8、主函数实现代码

```

func main() {
    for i := 0; i < RaftCount; i++ {
        initialize(i)
    }
    for {
        time.Sleep(time.Hour)
    }
}

```

## 2.9、运行结果

=====选举中.....=====

开始选举领导者

3 成为了主节点，开始发送心跳信号

=====选举成功=====

投票检查成功！心跳检查正常

当前2节点状态为：跟随者

当前4节点状态为：跟随者

当前1节点状态为：跟随者

当前3节点状态为：领导者

当前1节点状态为：跟随者

当前2节点状态为：跟随者

当前0节点状态为：跟随者

## 2.10、代码下载

在file文件下：[点击下载](#)

# 安全机制之国密

---