# **一、U**buntu**安装以太坊私有链**

## 1．安装go

### 1.1 获取安装包

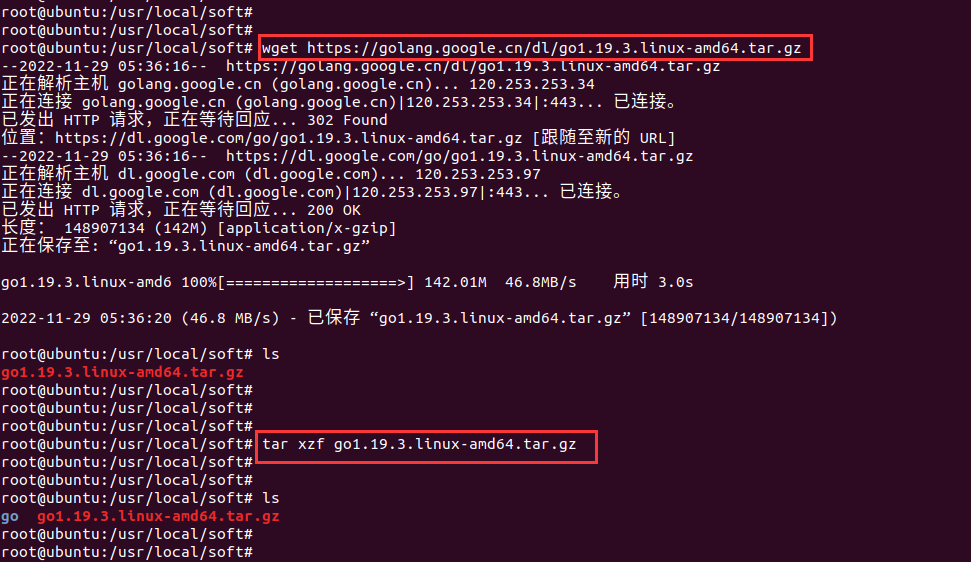
会直接将安装包下载到当前目录下，版本不低于1.7

Go官网：<https://go.dev/doc/install>

国内访问：<https://golang.google.cn/dl/> 下载哪个就在地址后加文件名，

如下载1.19版: wget <https://golang.google.cn/dl/go1.19.3.linux-amd64.tar.gz>

解压到当前文件： tar xfz go1.13.4.linux-amd64.tar.gz



**🡪🡪🡪🡪🡪🡪 之前写的解压如下（1.1）,与上边写的大体一样：**

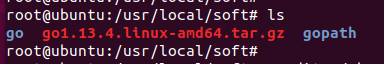
wget https://studygolang.com/dl/golang/go1.13.4.linux-amd64.tar.gz

**【1.1】**解压文件，我先创建了soft文件夹，并在soft下创建gopath文件夹

解压指定路径到soft下，会自动在/usr/local/soft下产生成go文件夹

tar xfz go1.13.4.linux-amd64.tar.gz -C /usr/local/soft

目录结构：



### 1.2 配置全局变量，

更改配置文件profile。先在解压go的同级目录中创建新文件夹 gopath,用来存放项目 ：

root@ubuntu:/usr/local/soft# **vim /etc/profile**

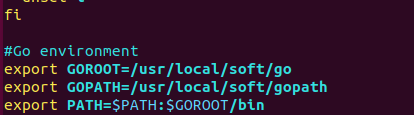
**// 添加的内容如下：**

**#Go enviroment**

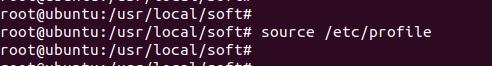
**export GOROOT=/usr/local/soft/go**

**export GOPATH=/usr/local/soft/gopath**

**export PATH=$PATH:$GOROOT/bin**



使配置文件生效：



**🡪🡪🡪🡪🡪🡪 之前的写法如下，只是修改、保存的配置文件不一样：**

修改 ~/.bashrc 文件

gedit ~/.bashrc

在文件最底部添加：

#Go environment

export GOROOT=/usr/local/soft/go

export GOPATH=/usr/local/soft/gopath

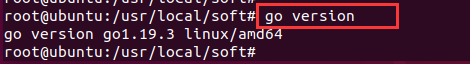
export PATH=$PATH:$GOROOT/bin

保存配置文件

source ~/.bashrc

### 1.3 验证

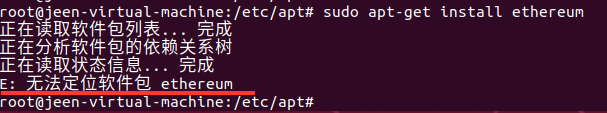
go version



## 2．安装以太坊

### 2.1 查看以太坊支持的ubutu版本

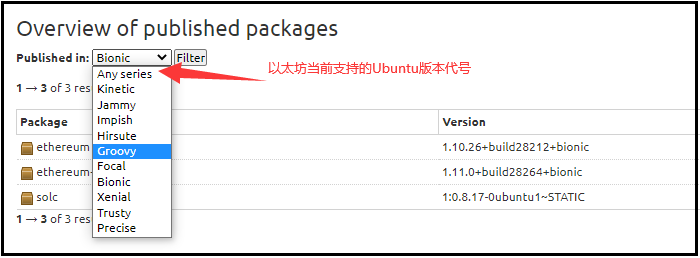
当前以太坊当前主要支持版本是ubuntu **14到18** ， 版本不持支时，容易出现以下问题：



查询支持的最新Ubuntu版本情况网址：

https://launchpad.net/~ethereum/+archive/ubuntu/ethereum?field.series\_filter=bionic

其中“Overview of published packages， Published in:”就是支持的ubuntu版本代号



### 2.2 安装以太坊环境geth和Ethereum

**（为减少不必要的问题，建议按以下命令执行）**

sudo apt-get install software-properties-common

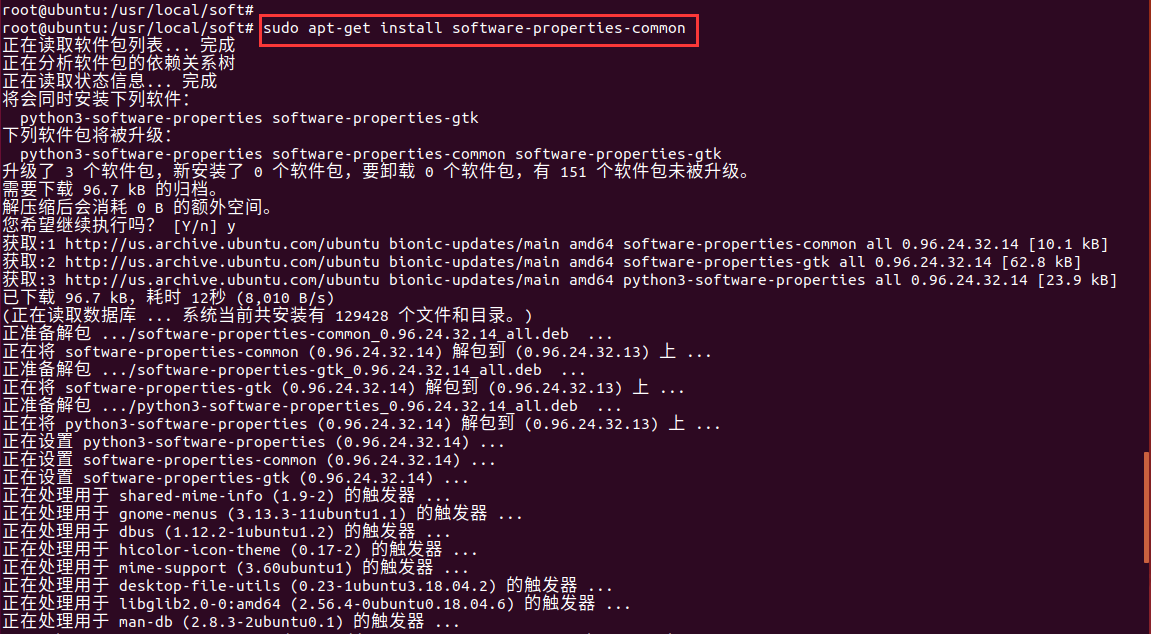
sudo -E add-apt-repository -y ppa:ethereum/ethereum

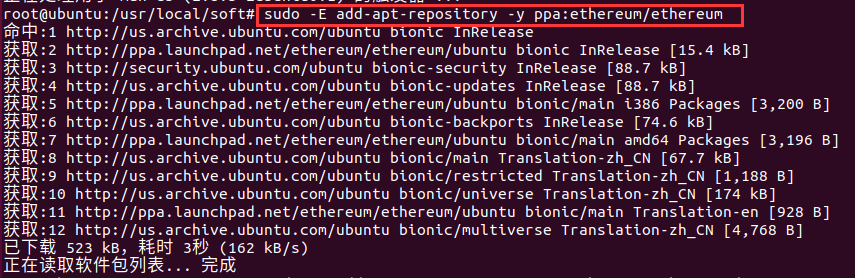
sudo -E add-apt-repository -y ppa:ethereum/ethereum-dev

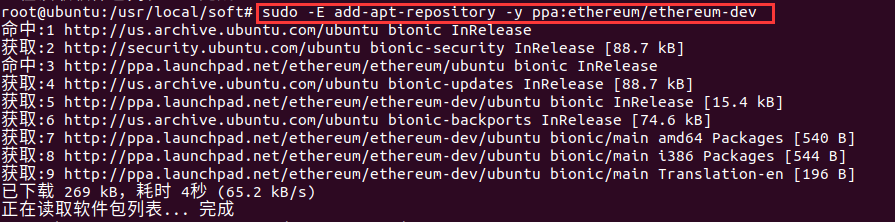
sudo apt-get update

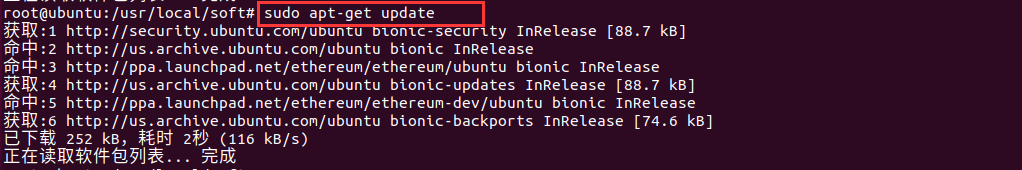
sudo apt-get install ethereum

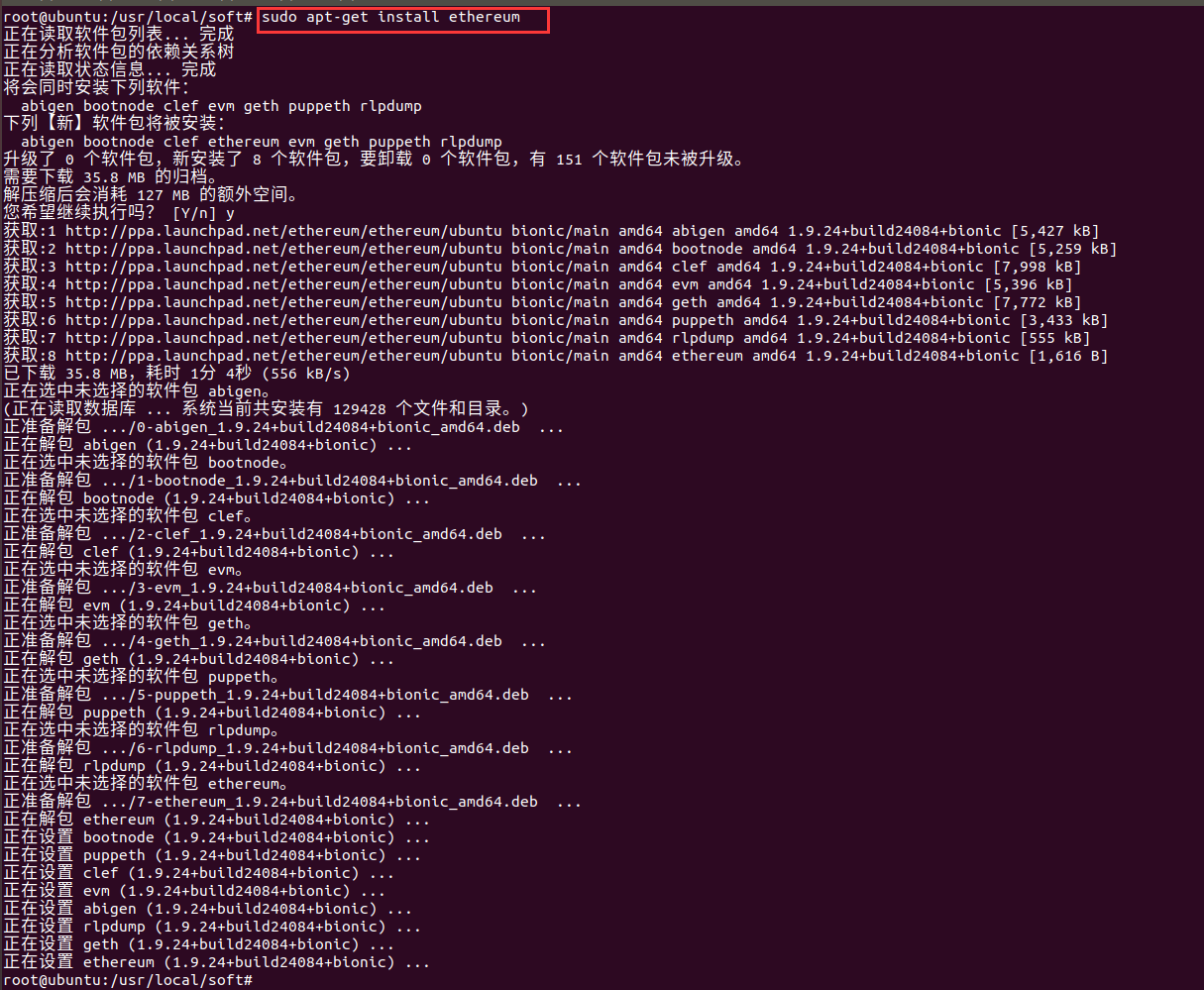
geth version //查看安装是否成功，查看安装版本信息

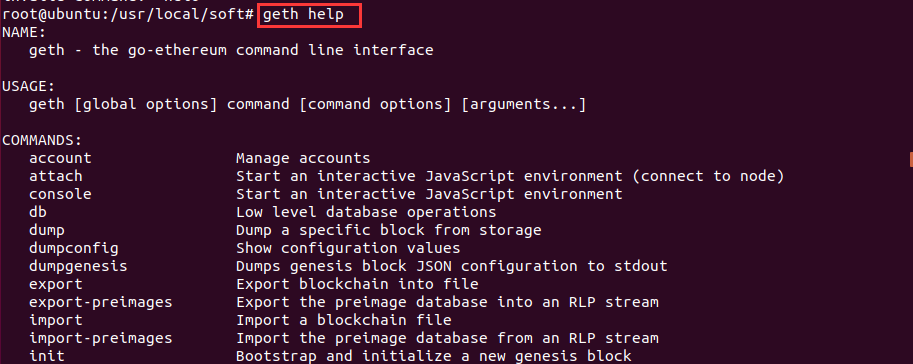


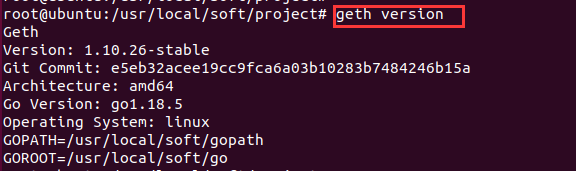












### 2.3 Geth基本命令

Geth命令是启动以太坊、**进入控制台之前**使用的命令，常用命令： <https://learnblockchain.cn/2017/11/29/geth_cmd_options/>

Web3命令是**进入控制台之后**使用的命令，使用文档： <https://learnblockchain.cn/docs/web3js-0.2x/>

*# geth version //查看geth版本信息*

*# geth account new //创建以太坊帐号，默认保存到*/root/.ethereum/keystore下

*# geth account new --datadir /usr/local/data //使用--datadir指定保存位置，在data下自动生成keystore文件，帐号保存在此文件夹下*

*# geth account list //查看以太坊帐号列表，查询到的数据默认来自*/root/.ethereum/keystore下

*# geth account list --datadir /usr/local/data //查看指定目录下的帐号列表，读取data下keystore中的数据*

*// 删除以太坊帐号：没有直接删除命令，但是可以到帐号目录keystore下手动删除文件，就可以删除相关账号*

*//进入javaScript控制台前指定输出日志位置，执行以下命令*

*# geth console 2>>/usr/local/geth.log*

*//进入javaScript控制台*

*# geth console*

## 3．运行以太坊

### 3.1 连接主网，同步区块数据。

方法一：将数据保存到当前目录下的data文件夹中，没有就自动创建

*# geth --datadir ./data*

方法二：“syncmode fast”以快速模式同步区块，该模式下只会下载每个区块头和区块体，但不执行验证交易，直到所有区块同步完毕再去获取一个系统当前的状态

*# geth --datadir .--syncmode fast*

### 3.2 连接测试网络Ropsten

*# geth --testnet --datadir .--syncmlde fast*

## 4．创世区块

### 4.1 配置创世区块json文件

*# mkdir /usr/local/soft/project //创建文件夹*

*# touch genesis.json //创建文件genesis.json*

创世区块文件是一个json格式的文件，一般命名为**genesis.json**，**一定要去官网拷贝该文件的内容**，因为geth是不停在升级的，genesis.json的内容也会有变化。 <https://github.com/ethereum/go-ethereum>

**chainId** : 为私链编号，区别于主网编号1、官方测试网编号2；

**homesteadBlock** : 以太坊推出的第2个主要的区块发行版本，Frontier 是第1个推出的区块发行版本（也是测试版本）。

建议使用homesteadB lock，值为0 表示有效。

**alloc** ：给指定帐号初始化一定数量的币，单位wei；

**difficulty** ：挖矿难度，该值越小，矿越容易；

**gaslimit** ：挖每个区块需要消耗资源的上限；

*{*

*"config": {*

*"chainId": 15,*

*"homesteadBlock": 0,*

*"eip155Block": 0,*

*"eip158Block": 0*

*},*

*"alloc": {*

*"e1d7b01dcdde4e0ff891ba2d5f4554b3f71d15a9": {"balance":"3000000000000000"}，*

*"7df9a875a174b3bc565e6424a0050ebc1b2d1d82": {"balance":"3000000000000000"},*

*"f41c74c9ae680c1aa78f42e5647a62f353b7bdde": {"balance":"4000000000000000"}*

*},*

*"coinbase" : "0x0000000000000000000000000000000000000330",*

*"difficulty" : "0x20000",*

*"extraData" : "",*

*"gasLimit" :"0xffffffff",*

*"nonce" : "0x0000000000000024",*

*"mixhash" : "0x0000000000000000000000000000000000000000000000000000000000000000",*

*"parentHash" : "0x0000000000000000000000000000000000000000000000000000000000000000",*

*"timestamp" : "0x00"*

*}*

### 4.2 创世块初始化

使用以下命令初始化创世区块文件geneses.json，指定私有链相关数据的存放目录myChain

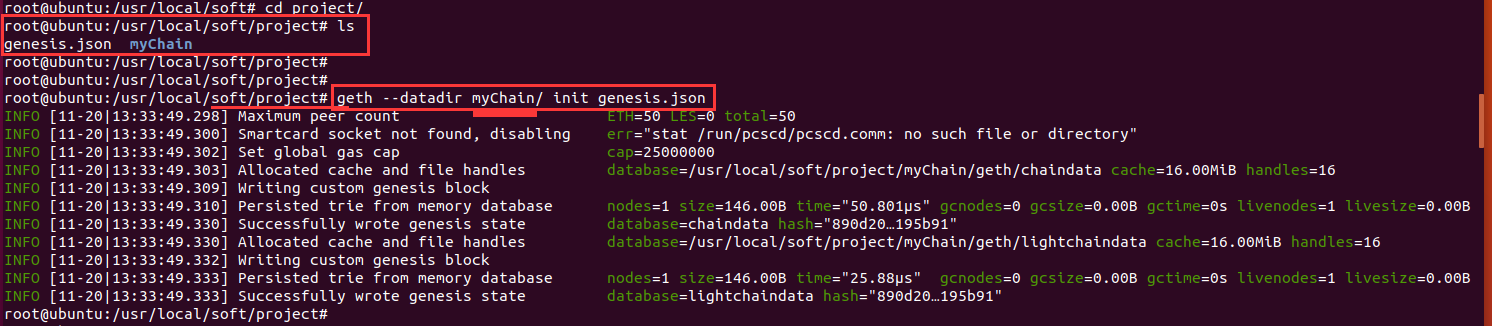
1. **--datadir：指定私链数据的存放目录**。当要在同一计算机中创建其他私链节点时，可创建另外的文件夹，例如myChain2。
2. 如果未设置--datadir，则会读取以太坊主网的数据， **默认位置**在/root/.ethereum下。

*geth init genesis.json --datadir myChain/*

初始化成功后，会在数据目录myChain中生成geth和keystore两个文件夹，此时目录结构如下：

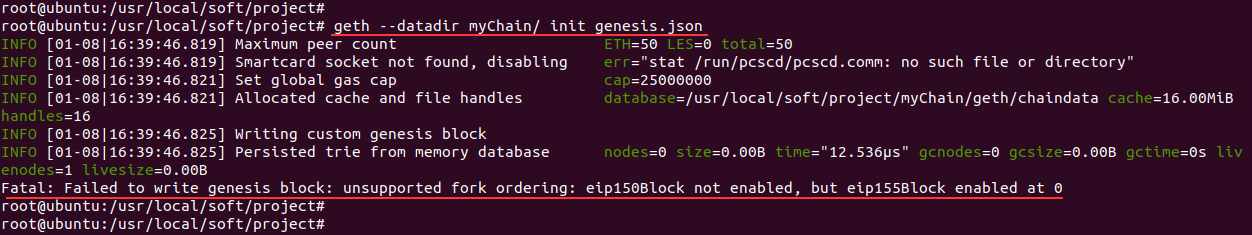
**geth** 中存放的是同步的区块链及相关数据。

**keystore**中存放的是账户数据。由于私有链刚创建，还没有创建账户，所以keystore中为空。



**报错1：**若出现以下失败，极有可能genesis.json文件有问题

**解决**：更换genesis.json文件，删除myChain/下所有文件，重新初始化。下载genesis.json网址 <https://github.com/ethereum/go-ethereum>

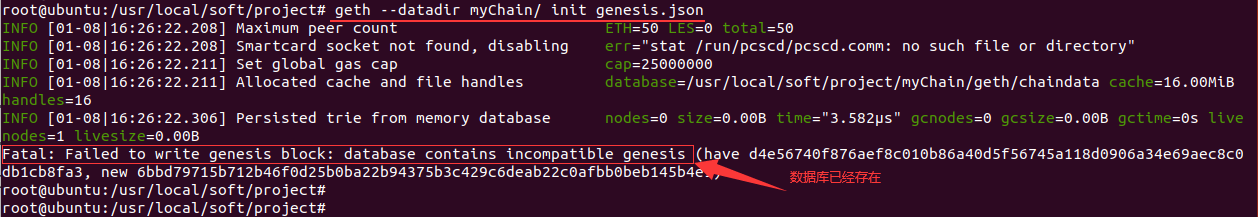


**报错2：**若出现以下失败，数据库已经存在：

**解决**：执行删除，再初始化：

方法1：删除myChain下所有文件，这些文件都是初始化后产生的

方法2：若删除myChain下文件不起作用，可以试试删除数据库： # geth removedb //删除数据库



### 4.3 启动创世块

**进入到myChain/下，执行以下命令**

**geth命令 参数详解：** <https://learnblockchain.cn/2017/11/29/geth_cmd_options/>

**方法一：**15是初始化时，genesis.json中定义的chainId：

# geth --datadir . --networkid 15 console

**方法二：（建议使用）**

// 输出log重定向，自动创建myChain/output.log文件，并将日志信息写入

# geth --datadir . --networkid 15 console 2>output.log

//单独开一个窗口，动态查看日志

# tail -f output.log

**方法三：配置更细化**

# geth --indentity “MyEthereum” --rpc --rpccorsdomain “\*” --datadir myChain/ --port “30303” --rpcapi “db,eth,net,web3,personal,miner” --networkid 15 cosole 2>1.txt

**说明：**

--indentity “MyEthereum” // 自定义链名

--rpc --rpccorsdomain “\*” // 允许所有rpc连接

--datadir myChain/ // 目录地址

--port “30303” // 端口

--rpcapi “db,eth,net,web3,personal,miner” // 允许的api方式

--networkid 15 // 网络标识符，链id（genesis.json中的chainId）

--networkid 6500 cosole 2>1.txt // 控制台输出位置

**方法四：**允许远程连接：--rpc，默认端口8545，（可以使用MetaMask插件的localhost进行连接）

# geth --datadir . --networkid 15 --rpc console 2>output.log

**方法五：**后台启动

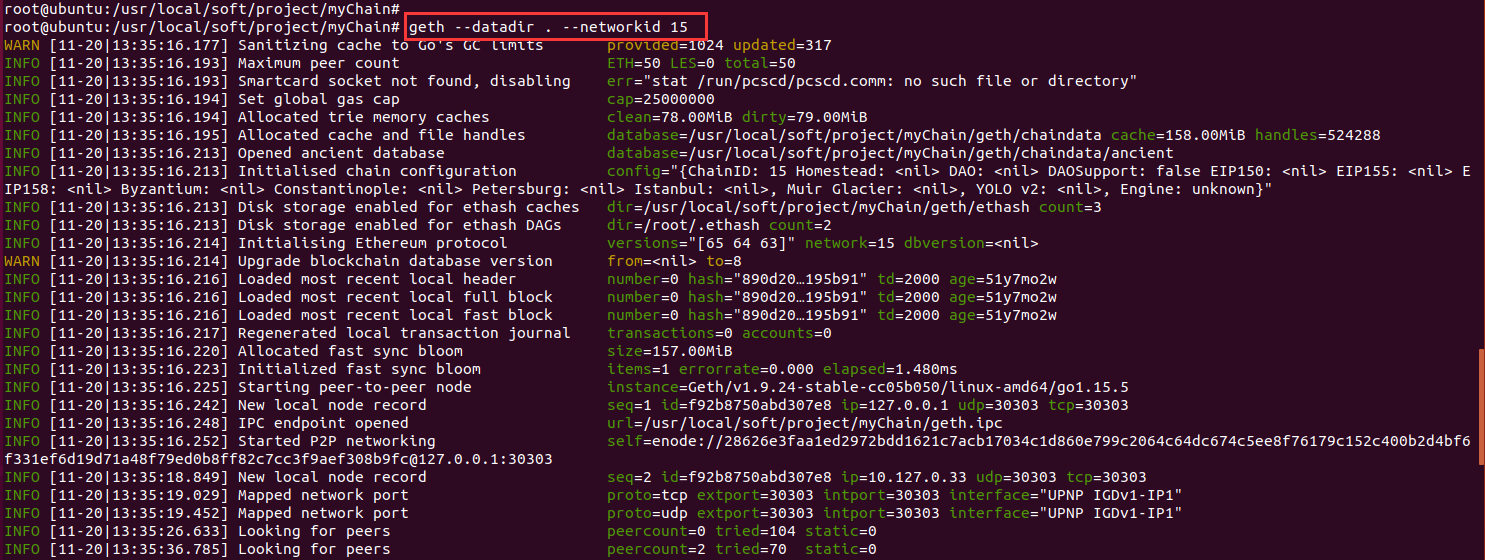
# nohup geth --datadir . --networkid 15 --rpc console 2>output.log &

**方法六：开发者模式**，此模式下，可以自由测试各种命令

# geth --datadir myChain/ --dev console 2>output.log

//删除原来的链 当然你可以用新的目录

# geth removedb --datadir ./



### 4.4 直接挖矿、停止挖矿

**进入控制台后，使用Web3命令，使用文档：** <https://learnblockchain.cn/docs/web3js-0.2x/>

首先绑定矿工和帐号信息，miner为矿工对象，绑定的账号为genesis.json中配置的帐号

> miner.**setEtherbase**(*”e1d7b01dcdde4e0ff891ba2d5f4554b3f71d15a9*”)

> miner.start() //开始挖矿, 参数表示挖矿使用的线程数，可自定义，默认值是CPU 内核的数量，

> mimer.stop() //停止挖矿

> miner. getBalance(“*e1d7b01dcdde4e0ff891ba2d5f4554b3f71d15a9*”) //查看挖矿后的账户余额

### 4.4 创建账户、挖矿、停止挖矿

也可以创建帐号再挖矿

> personal.newAccount(“123456”) //创建账户，密码123456，命令执行后返回账号id

> miner.start() //开始挖矿, 参数表示挖矿使用的线程，可自定义，默认值是CPU 内核的数量，

> mimer.stop() //停止挖矿

> miner.**setEtherbase**(eth.accounts[1]) //设置、切换矿工

### 4.5 控制挖矿数量

有时需要控制挖矿数量，即挖到指定数量的区块后自动终止挖矿

> miner.start() //开始挖矿, 参数表示挖矿使用的线程，可自定义，默认值是CPU 内核的数量，

> miner.sleepBlock(10) //要挖的数量，挖到10个后停止，执行miner.stop()操作

> mimer.stop() //停止挖矿

### 4.6 转账操作

// 1、查看指定用户余额

> eth.getBalabce(eth.accounts[0]) //查看第一个用户的余额

// 2、先解锁用户,输入账号、密码

> personal.unlockAccount(eth.account[0],’123456’)

// 3、后转账，从第一个用户转给“账号2”，将转账数额单位转换成Wei，返回值为交易的Hash值

> eth.sendTransaction({from:eth.account[0],to:”账号2”,value:web3.toWei(3,’ether’)})

// 4、查看转账后的余额：

> eth.getBalance(eth.accounts[0])

### 4.5 Web3相关命令

> personal.newAccount() //创建帐号

> personal.unlockAccount() //解锁账号

> eth.accounts //列出系统中的所有账号

> eth.getBalance() //查看账户余额，单位wei,可查看的具体账号

> eth.sendTransaction({from:eth.accounts[0],to:””, value:web3.toWei(10,’ether’)})

> eth.getTrnsaction(“交易Hash值”) //根据交易Hash查看交易详情

> eth.blockNumber //区块高度

> eth.getBlock(区块ID) //根据区块的id获取该区块的详细信息

> miner.start(1) //开始挖矿

> miner.stop() //停止挖矿

> web3.fromWei() //Wei换算成以太币

> web3.toWei() //以太币换算成Wei

# 二、Ubuntu安装truffle

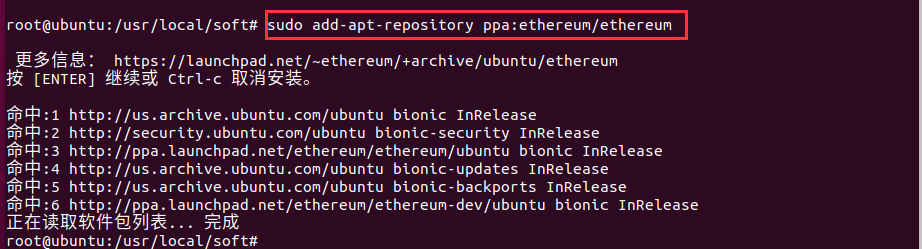
## 1．安装solc（solidity编译器）

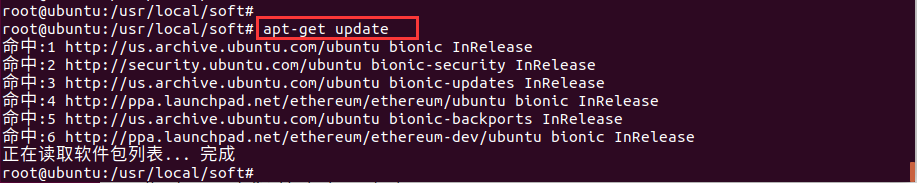
solidity是以太坊智能合约的开发语言。想要测试智能合约，开发DAPP的需要安装solc

sudo apt-get update

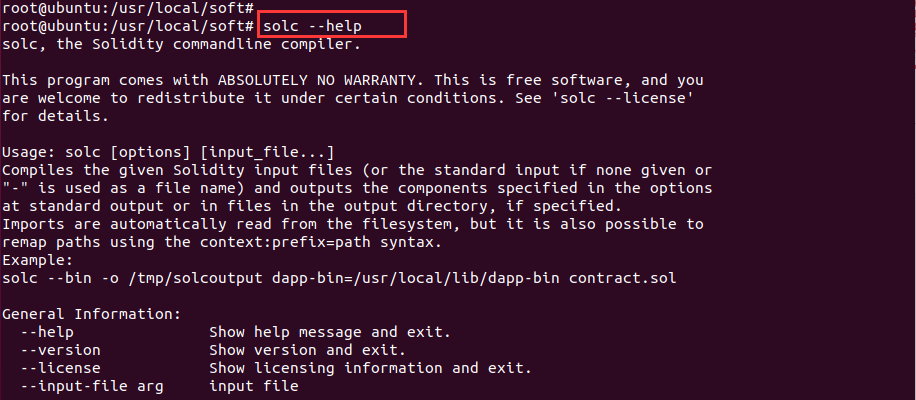
sudo apt-get install solc

solc --help //验证是否安装成功









## 2．安装nodejs

这个是truffle框架依赖

### 2.1 tar.xz源码包方式安装



下载最新稳定版本的nodejs，下载地址。先在go文件夹的同级目录中创建新的文件夹nodejs

<https://nodejs.org/zh-cn/download/>

//可以将文件移动到文件夹nodejs(记住安装位置，下一步用)，然后再解压

# mv /home/XXX/下载 **/usr/local/soft/nodejs**

# tar -xvf node-v18.12.1-linux-x64.tar.xz

在/usr/bin目录下创建软连接

//切换目录

# cd /usr/bin

//创建node安装目录的软链接

# sudo ln -s **/usr/local/soft/nodejs/bin/node** node

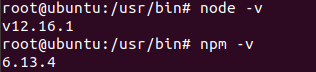
//创建npm安装目录的软链接

# sudo ln -s **/usr/local/soft/nodejs/bin/npm** npm

查看安装成功与否

# node -v

# npm -v



### 2.2 tar.gz源码包方式安装(未测试)

<https://nodejs.org/dist/v18.12.1/node-v18.12.1.tar.gz>

# wget [https://nodejs.org/dist/v18.12.1/node-v18.12.1.tar.gz](https://nodejs.org/dist/v18.12.1-linux-x64.tar.xz)

# tar xvf node-v18.12.1.tar.gz

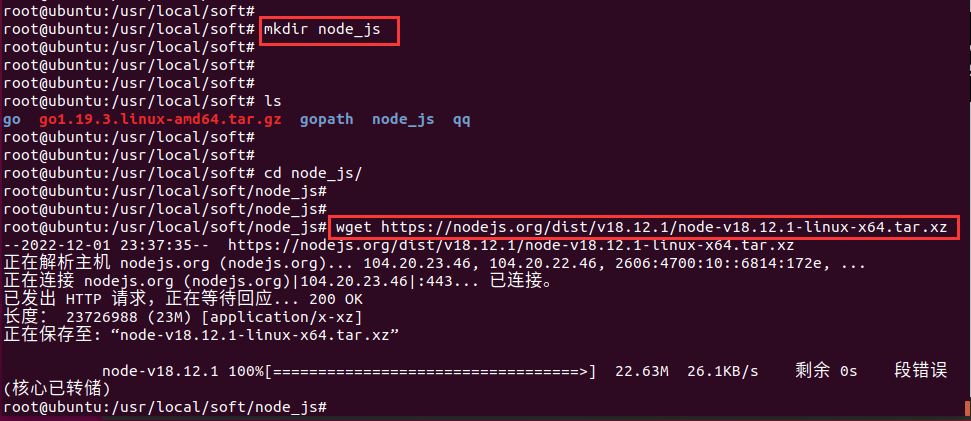
# cd node-v18.12.1 //进入到解压文件中

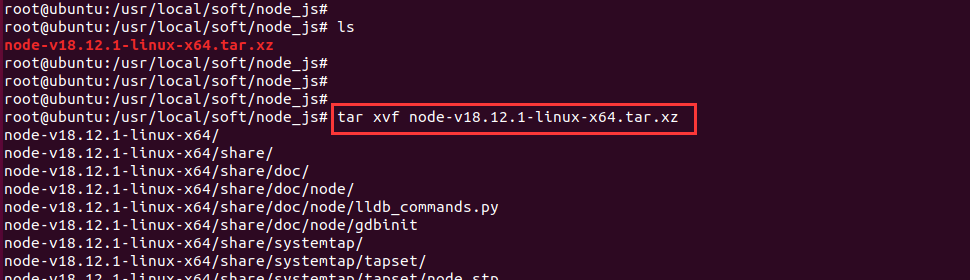
# ./configure

# make

# make install

# cp /usr/local/bin/node /usr/sbin/





### 2.3 镜像源方式安装

来源：<https://www.cnblogs.com/lidonghao/p/8495895.html>

先配置源

sudo apt-get update

sudo apt-get install -y python-software-properties software-properties-common

sudo add-apt-repository ppa:chris-lea/node.js

sudo apt-get update

安装nodejs

sudo apt-get install nodejs

sudo apt install nodejs-legacy

sudo apt install npm

更新npm的包镜像源，方便快速下载

sudo npm config set registry https://registry.npm.taobao.org

sudo npm config list

**全局安装n管理器(用于管理nodejs版本)**

sudo npm install n -g

安装最新的nodejs（stable版本

sudo n stable

sudo node -v

## 3．安装web3.js

Web3.js提供了用于和geth通信的JavaScript API。内部使用JSON RPC协议与geth通信

1. //新建一个文件夹**web3js**，进入文件夹，然后初始化

# npm init -f //此处的-f会自动给你配好配置文件

1. //安装web3,默认为最新版本

# sudo npm install web3 --save

如果安装失败 执行低版本：sudo npm install web3@^0.20.0

1. 更新npm：

# sudo npm i npm to update

验证缓存数据的有效性和完整性，清理垃圾数据:：

sudo npm cache verify

## 4．安装truffle和testrpc

truffle和testrpc是配套的以太坊开发框架。通过truffle可以快速的编译和部署合约并进行测试，同时还有web前端交互界面。  
testrpc可以理解为快速生成以太坊测试账号。 不过我自己后面还是习惯用geth搭建私有链测试部署。

sudo npm config set registry https://registry.npm.taobao.org

sudo npm install -g truffle

sudo npm install -g ethereumjs-testrpc

执行

testrpc

看是否安装成功

## 5．Solc操作

打包项目ABI接口：进入项目文件夹

solc --abi 项目名(如:Coin.sol)

打包bin文件：进入项目文件夹

solc --bin 项目名(如:Coin.sol)

## 6．web3操作

### 6.1 初始化web3

//进入node

# node

//退出 node

# .exit

//web3初始化，之后就可以使用web3的相关命令

# var Web = require(‘web3’)

# var web3 = new Web(new Web.providers.HttpProvider('http://localhost:8545'))

为保证我们的metaMask设置好的provider不被覆盖掉，在引入web3之前一般要做当前环境检查(以**v0.20.1为例**)

*if(typeof web3 != ‘undefined’){*

*web3 = new Web3(web3.currentProvider);*

*}else{*

*web3 = new Web3(new Web3.privoders.HttpProvider(“http://localhost:8545”));*

*}*

### 6.2 异步回调(callback)

web3.js API设计的最初目的，主要是为了和本地RPC节点共同使用，所以默认情况下发送的时同步HTTP请求。若要返送异步请求，可以在函数的最后一个参数的位置上，传入一个回调函数（可选）。

*//回调风格是所谓的“错误优先”,****以v0.20.0为例***

*we3.eth.getBlock(48，function(err, res){*

*if(err)*

*console.log(JSON.stringify(res));*

*else*

*console.log(err);*

*})*

### 6.3 回调Promise事件(v1.0.0)

* 为帮助web3集成到不同标准的所有类型的项目中，**v1.0.0** 提供了多种方式处理异步函数。大多数的web3对象允许将一个回调函数作为最后一个函数参数传入，同时会返回一个promise用于链式函数调用。
* 以太坊作为一个区块链系统，一次请求具有不同的结束阶段，为了满足这样的要求，**v1.0.0** 版本将这类函数调用的返回值宝成一个“承诺事件”(promiEvent)，这样一个promise和EventEmitter的结合体
* PromiEvent的用法就像promise一样，另外还加入了.on() .once() 和 .off() 方法:

*web3.eth.sendTransaction({from:’0x123…’,data:’0x456…’})*

*.once(‘transactionHash’,function(hash){…})*

*.once(‘receipt’,function(receipt){…})*

*.on(‘confirmation’,function(confNumber,receipt){…})*

*.on(‘error’,function(error){…})*

*.then(function(receipt){//will be fired once the receipt is mined});*

### 6.4 应用二进制接口(ABI)

* web3.js通过以太坊智能合约的json接口（Application Binary Interface，ABI）创建一个JavaScript对象，用来在js代码中描述
* 函数（functions）

type：函数类型，默认“function”，也可能是“constructor”

constant, payable, stateMutability：函数的状态可变性

inputs, outputs：函数输入、输出参数描述列表

* 事件（events）

type：类型，总是“event”

inputs：输入对象列表，包括 name，type，indexed

### 6.5 批处理请求（batch requests）

* 注意：批处理请求不会更快。实际上，在某些情况下，一次性的发出许多请求会更快，以为请求时异步处理。
* 批处理请求住哟啊用于确保请求的顺序，并串行处理。

*var batch = web3.createBtch();*

*batch.add(web3.eth.getBalance.request(‘0x0000000000000000000000000000’,’latest’,callback));*

*batch.add(web3.eth.contract(abi).at(address).balance.request(address,callback2));*

*batch.execute();*

### 6.6 大数处理（big numbers）

* JavaScript中默认的数字精度较小，所以web3.js会自动添加一个依赖库BigNumber,专门处理大数
* 对于数值，我们应该习惯把他转换成BigBumber对象来处理

*var balance = new BigNumber(‘1231231231231231231231231231231’);*

*//or var balance = web3.eth.getBalance(someAddress);*

*balance.plus(21).toString(10);*

* BigNumber.toString(10)对小数只保留20位浮点精度。所以推荐的做法是，内部总是用wei来表示余额（大整数），只有在需要显示给用户看的时候才转换成ether或其他单位。

### 6.7 基本信息查询

* 查看web3版本

# web3.version.api //**v0.2x.x**

# web3.version //**v1.0.0**

* 查看web3来年街道的结点版本（clientVersion）

# web3.version.node //同步v0.2x.x

# web3.version.getNode((error,result)=>{console.log(result)}) //异步：v0.2x.x

# web3.eth.getNodeInfo().then(console.log) //异步：v1.0.0

* 获取network id

同步**v0.2x.x** ：

# web3.version.network

异步：

# web3.version.getNetwork((err,res)=>console.log(res)) **//v0.2x.x**

# web3.eth.net.getId().then(console.log) **//v1.0.0**

* 获取节点的以太坊协议版本

同步**v0.2x.x** ：

# web3.version.etherum

异步：

# we3.version.getEthereum((err,res)=>console.log(res)) **//v0.2x.x**

# we3.eth.getProtocolVersion().then(console.log) **//v1.0.0**

### 6.8 网络状态查询

* 是否右节点连接/监听，返回true/false

同步 **v0.2x.x** :

# web3.isConnect() 或者 web3.net.listening

异步：

# web3.net.getListenint((err,res)=>conslole.log(res)) **//v0.2x.x**

# web3.eth.net.isListening().then(console.log) **//v1.0.0**

* 查看当前连接的peer节点

同步 **v0.2x.x** :

# web3.net.peerCount

异步：

# web3.net.getPeerCount((err,res)=>console.log(res)) **//v0.2x.x**

# web3.eth.net.getPeerCount().then(console.log) **//v1.0.0**

### 6.9 查看、设置Provider

//查看当前设置好的web3 Provider

# web3.currentProvider

//查看浏览器环境设置的web3 Provider**(v1.0.0版以后)**

# web3.givenProvider

//设置Provider,主要是改ip、端口

# web3.setProvider(new web3.providers.HttpProvider(‘http://localhost:8545’))

### 6.10 通用工具方法

//以太单位转换

# web3.fromWei() 例：web3.fromWei(7678739432342342,’ether’) //由wei转成以太

# web3.toWei() 例: web3.toWei(1,’ether’) //由以太转成wei

//数据类型转换

# web3.toString()

# web3.toDecimal()

# web3.toBigNumber() 例：*web3.toBigNumber(‘8564651335613565685653’)*

//字符编码转换

# web3.toHex()

# web3.toAscii()

# web3.toUtf8()

# web3.fromUtf8()

//地址相关

# web3.isAddress() 例：*web3.isAddress(‘0x6FD5A9099034839483CB’)* //地址长度20个字节即160位，转成16进制是40个字符

# web3.toChecksumAddress()

### 6.11 账户相关

* coinbase查询

同步 **v0.2x.x** :

# web3.eth.coinbase

异步：

# web3.eth.getCoinbase((err,res)=>console.log(res)) **//v0.2x.x**

# web3.eth.getCoinbase().then(console.log) **//v1.0.0**

* 账户查询

同步 **v0.2x.x** :

# web3.eth.accounts

异步：

# web3.eth.getAccounts((err,res)=>console.log(res)) **//v0.2x.x**

# web3.eth.getAccounts().then(console.log) **//v1.0.0**

### 6.12 区块相关

* 区块查询，callback为回调函数，(err,res)=>console.log(res)

同步:传参hash或区块高度，[true/false]

# web3.eth.getBlock(hashStringOrBlockNumber[,returnTransactionObjects])

例：*web3.eth.getBlock(25)*

异步：传参hash或区块高度

# web3.eth.getBlock(hashStringOrBlockNumber,callback)

例：*web3.eth.getBlock(25,(err,res)=>console.log(res))* //25为区块高度

* 块中交易数量查询

同步: 传参hash或区块高度

# web3.eth.getBlockTransactionCount(hashStringOrBlockNumber)

异步：传参hash或区块高度

# web3.eth.getBlockTransactionCount(hashStringOrBlockNumber,callback)

* 区块高度查询

同步: 传参hash或区块高度

# web3.eth.blockNumber

异步：传参hash或区块高度

# web3.eth.getBlockNumber(callback)

* gasPrice查询

同步:

# web3.eth.gasPrice

异步：

# web3.eth.getGasPrice(callback)

* 余额查询，defaultBlock默认latest

同步:

# web3.eth.getBalance(addressHesString [,defaultBlock])

异步：

# web3.eth.getBalance(addressHexString [,defaultBlock] [,callback])

* 交易查询

同步:

# web3.eth.getTransaction(transactionHash)

异步：

# web3.eth.getTransaction(transactionHash [,callback])

### 6.13 交易执行相关

* 交易收据查询(已进块)

同步:

# web3.eth.getTransactionReceipt(hashString)

异步：

# web3.eth.getTransactionReceipt(hashString [,callback])

* 估计gas消耗量

同步:参数为交易对象

# web3.eth.estimateGas(callObject)

例： *web3.eth.estimateGas({feom:web3.eth.accounts[0],to:web3.eth.accousnts[1],value:20000})*

异步：

# web3.eth.estimateGas(callObject [,callback])

### 6.14 发送交易

# web3.eth.sendTransaction(transactionObject [,callback])

注意：操作之前需要解锁，例：personal.unlockAccount(eth.account[0])

* 交易对象：
* from：发送地址
* to：接收地址，如果是创建合约交易，**可不填**
* value：交易金额，以wei为单位，可选
* gas：交易消耗gas上限，可选
* gasPrice：交易gas单价，可选
* data：交易携带的字串数据，可选
* nonce：正数nonce值，可选

### 6.15 消息调用

# web3.eth.call(callback [,defaultBlock] [,callback])

* 参数：
* 调用对象：与交易对象相同，只是from也是可选项
* 默认区块：默认“latest“，可以传入指定的区块高度
* 回调函数，如果没有则为同步调用：

*var result = web3.eth.call({*

*to:”0x4abd0339eb8d57087278718986382264244252f”,*

*data:”0xc6888fa100000000000000000000000000000000000003”*

*});*

*console.log(result);*

### 6.16 日志过滤

# web3.eth.filter(fileterOptions [,callback])

注意：

*//filterString可以是‘latest’(监听最新挖的块hash)or‘pending’(监听最新的交易的hash)*

***步骤1>*** *var filter = web3.eth.filter(filterString);*

*//或者可以填入一个日志过滤 options（参数from,to,address等）*

*var filter = web3.eth.filter(options);*

*//监听日志变化*

***步骤2>*** *filter.watch(function(error,result){if(!error)console.log(result);});*

*filter.stopWatching() //关闭监听*

*//还可以用传入回调函数的方法，立即开始监听日志*

*web3.eth.filter(options,function(error,result){*

*if(!error) console.log(result)*

*})*

### 6.17 合约相关

# web3.eth.contract

//**1>** 调用abi创建js中的合约实例，此时还没有和区块链中的合约真正管理起来

*var MyContract = web3.eth.contract(abiArray);*

*//****2>*** *通过地址初始化合约实例*

*var contractInsatance = MyContract.at(address);*

*//****或2>*** *部署一个新合约, 【constructorParam:构造函数中的参数；{}里为交易对象；data字节码，即通过 solc --bin Coin.sol(合约名)得到，此时得到的十六进制字节码，没有“0x”开头，需要手动拼接，byteCode = ‘0x’+ byteBin。之后拼接交易对象，例：TxObject = {data:bytecode, …}】*

*var contractInstance = MyContract.new([constructorParam1] [,constructorParam2],*

*{data:’0x12345…’, from:myAccount, gas:1000000})*

### 6.18 调用合约函数

* 通过已创建的合约实例，直接调用合约函数。

**myContractIntance**即合约实例

**myMethod**即自己在合约中当以的变量、函数

*//直接调用，自动按函数类型决定用sendTransaction还是call*

***myContractInstance****.myMethod(param1 [,param2, …] [,transactionObject][,defaultBlock][,callback]);*

*//显示以消息调用形式call该函数****（不需要发送交易情况 调用）***

***myContractInstance****.myMethod.call(param1 [,param2, …] [,transactionObject][,defaultBlock][,callback]);*

*//显示以发送交易形式调用该函数****（有状态的更改，比如转币时 调用）***

***myContractIntance****.myMethod.sendTransaction(param1 [,param2, …] [,transactionObject] [,callback]);*

### 6.19 监听合约事件

* 合约的event类似于filter，可以设置过滤选项来监听

myContractInstance 为合约实例

MyEvent() 为合约中定义的事件

*var event = myContractInstance.MyEvent({valueA:23} [,additionalFilterObject])*

*//监听事件*

*event.watch(function(error,result){if(!error) console.log(result);});*

*//还可以用传入回调函数的方法，立刻开始监听事件*

*var event = myContractInstance.MyEvent([{valueA:23}]*

*[,additionalFilterObject], function(error,result){*

*if(!error) console.log(result);*

*});*

如果在合约目录下/usr/local/soft/web3\_test/contract执行监听命令：

*//启动事件监听器。也可以监听“latest”*

*# myContractInstance.Sent(“pending”,(err,res)=>console.log(res))*

*myContractInstance 为合约实例*

*sent()为合约中的事件*

# 部署合约

此处为在自己私链上部署合约

1. **打开窗口1**，进入私链目录：/usr/local/soft/project/myChain，执行命令启动geth，创世纪块，与本地metaMask关联

*# geth --datadir . --networkid 15 --rpc console 2>output.log*

1. 查看是否存在账户，不存在就创建账户

*# eth.accounts //查看账户列表*

*# personal.newAccount(“123456”) //创建账户，密码为123456，返回用户id。此时账户是锁死的，不能指向交易*

1. 选择用户矿工，进行挖矿，多个用户时可切换矿工

*# miner.start(1) //开始挖矿*

*# mimer.stop() //停止挖矿*

*# miner.setEtherbase(eth.accounts[1]) //切换矿工*

1. **另打开新窗口2**，进入目录/usr/local/soft/project/myChain，实时查看output.log日志

*# tail -f output.log*

1. **另开新窗口3，**进入目录/usr/local/soft/web3\_test/contract,此目录存放智能合约.sol文件

# solc --bin --abi Coin.sol //获取Coin.sol智能合约的字节码bin内容、abi的Json串

1. **另打开新窗口4**，进入目录/usr/local/soft/web3\_test，此web3是v0.20.0版本，启动web3

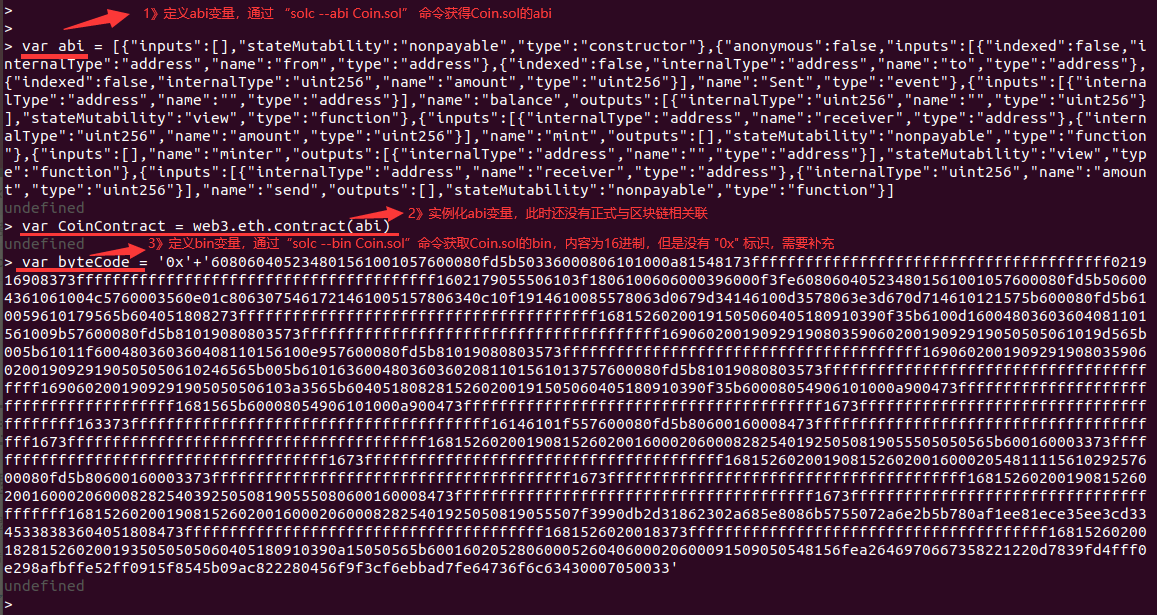
//首先进入node，初始化web3

# node

# var Web = require(‘web3’)

# var web3 = new Web(new Web.providers.HttpProvider('http://localhost:8545'))

1. **部署智能合约到区块链**，步骤如下：**第5步执行后返回 合约地址**





# Web3执行以太坊脚本

## 实现直接转币

1. 编写js文件，如下最简单的“转币”功能

*var Web = require(‘web3’);*

*var web3 = new Web(new Web.providers.HttpProvider(“http://localhost:8545”));*

*var \_from = web3.eth.accounts[0];*

*var \_to = “0xC8c895d3bCC6604a37C6dD22359D2DD268D53459”;*

*var \_value = 5000000000;*

*web3.eth.sendTransaction({from: \_from, to: \_to, value: \_value},(err,res)=>{*

*if(err)*

*console.log("Error : ”,err);*

*else*

*console.log(“Transaction Hash : ”,res);*

*});*

1. 为了方便，直接在目录/usr/local/soft/web3\_test/contract建js文件，将以上内容复制进去

*//<1> 创建文件*

*# vi transfer\_script.js*

*//<2> copy内容，保存文件*

*//<3> 运行文件，运行前账号解锁。运行后返回交易Hash*

*# node transfer\_script.js*

1. 查看转账情况

*# eth.getBalance(eth.accounts[0])*

1. **可以将脚本中的固定值，改成输入的形式，操作更灵活**

*var Web = require(‘web3’)*

*var web3 = new Web(new Web.providers.HttpProvider(“http://localhost:8545”))*

*var arguments = process.argv.splice(2); //作用是使用node执行脚本时，从控制台输入的第3个参数开始读取。*

*.argv是一个数组，控制台输入的内容 # node 脚本名称 参数1 参数2 会写入到数组中，参数从下标2开始截取*

*if(!arguments || arguments.length != 2){*

*console.log(“Parameter length must be 2”);*

*return;*

*}*

*var \_from = web3.eth.accounts[0];*

*var \_to = arguments[0]; //截取的第一个参数，即控制台输入的第3个值*

*var \_value = arguments[1]; //截取的第一个参数，即控制台输入的第4个值*

*web3.eth.sendTransaction({from: \_from, to: \_to, value: \_value},(err,res)=>{*

*if(err)*

*console.log("Error : ”,err);*

*else*

*console.log(“Transaction Hash : ”,res);*

*});*

运行脚本时，接两个参数即可：

*# node transfer\_script.js 0xC8c895d3bCC6604a37C6dD22359D2DD268D53459 5000000000*

## 实现读取合约转币

1. 编写web3脚本,直接copy到/usr/local/soft/web3\_test/contract中的sendToken.js文件中,

*var Web = require(‘web3’);*

*var web3 = new Web(new Web.providers.HttpProvider(“http://localhost:8545”));*

*var \_from = web3.eth.accounts[0];*

*var \_to = web3.eth.accounts[1];*

*var amount = 500000;*

*//合约的abi,通过合约文件目录下执行 solc –abi Coin.sol(即合约名) 得到*

*var abi = [{"inputs":[],"stateMutability":"nonpayable","type":"constructor"},{"anonymous":false,"inputs":[{"indexed":false,"internalType":"address","name":"from","type":"address"},{"indexed":false,"internalType":"address","name":"to","type":"address"},{"indexed":false,"internalType":"uint256","name":"amount","type":"uint256"}],"name":"Sent","type":"event"},{"inputs":[{"internalType":"address","name":"","type":"address"}],"name":"balance","outputs":[{"internalType":"uint256","name":"","type":"uint256"}],"stateMutability":"view","type":"function"},{"inputs":[{"internalType":"address","name":"receiver","type":"address"},{"internalType":"uint256","name":"amount","type":"uint256"}],"name":"mint","outputs":[],"stateMutability":"nonpayable","type":"function"},{"inputs":[],"name":"minter","outputs":[{"internalType":"address","name":"","type":"address"}],"stateMutability":"view","type":"function"},{"inputs":[{"internalType":"address","name":"receiver","type":"address"},{"internalType":"uint256","name":"amount","type":"uint256"}],"name":"send","outputs":[],"stateMutability":"nonpayable","type":"function"}];*

*var CoinContract = web3.eth.contract(abi); //实例化abi，此时还没有和区块链产生实际联系*

*var cointractAddress = “0x814865b5d53562f6556f320056667d695c568a”; //合约地址，部署合约完成后会返回该地址*

*var contractInstance = CoinContract.at(contractAddress); //通过合约地址实例化合约对象，赋值给合约对象，****此处不是重新部署****，*

*//\_to:给谁发币；amount:发的自己代币的数量；*

*//交易本身：{from:发送者合约地址，to:即交易发给谁，接收者合约地址(此处可不写)，value:即给合约地址转的以太数(此处可不写)}*

*contractInstance.send(\_to, amount, {from: \_from},(err,res)=>{*

*if(err)*

*console.log("Error: ”, err);*

*else*

*console.log(“Result: ”, res);*

*})*

1. 创建文件，copy内容

*# vi* sendToken.js //copy脚本后保存

1. 起私链，新窗口：

*# ps -ef | grep geth //先查看是否有私链正在运行*

*# nohup geth --datadir . --networkid 15 --rpc 2>output.log & //后台启动私链，不需要有额外的窗口时刻打开着控制台*

*# ps -ef //查看启动状况*

*//若想重新进入控制台操作私链，attach命令进入后exit退出时不会关闭私链*

*# geth attach http://localhost:8545*

*// 若想 node执行前看合约对象：首先通过attach进入控制台*

*> var abi = [{“………”}]*

*> var CoinContract = web3.eth.contract(abi);*

*> var contractAddress = “0x8973478d998b0a7938899020”; //合约地址，部署合约后返回值*

*> var contractInstance = CoinContract.at(contractAddress);*

*> contractInstance* ***回车*** *//可查看合约对象*

*//查看合约相关的账户余额*

*> contractInstance.balance(eth.accounts[0]) //第一个账户*

*> contractInstance.balance(eth.accounts[1]) //第二个账户*

*//运行node前需要对交易的发起方\_from账户解锁。先查看当前账户，如果不是\_from账户，切换*

*# personal.unlockAccount(eth.accounts[0])*

*//解锁账户运行node后，如果账户值没有变化，说明没有挖矿，打开挖矿,有矿后关闭*

*# miner.start(1)*

*# miner.stop()*

1. 使用node调用执行脚本

*# node sendToken.js*

## 查询交易的脚本

查询的合约脚本，getAmountOf.js，被查询的账号有控制台输入，然后node执行

*var Web = require(‘web3’);*

*var web3 = new Web(new Web.providers.HttpProvider(“http://localhost:8545”));*

*var arguments = process.argv.slice(2);*

*if(!arguments || arguments!=2){*

*console.log(“Parameter length must be 1”);*

*}*

*var addr = arguments[0];*

*var abi = [{……}] //通过solc --abi Coin.sol(即合约名) 得到*

*var CoinContract = web3.eth.contract(abi);*

*var contractAddress = “0x………”; //合约地址：部署合约完成后返回*

*var contractInstance = CoinContract.at(contractAddtress);*

*//查询合约*

*contractInstance.balances(\_addr, (err,res)=>{*

*if(err)*

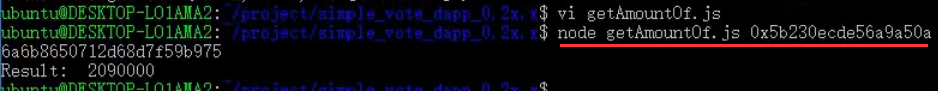
*console.log(“Error: ”,err);*

*else*

*console.log(“Result: “, res.toString());*

*})*

node执行脚本，传入账号参数



## 监听合约脚本

编写一个监听的脚本sentEvent\_listern.js文件

*var Web = require(‘web3’);*

*var web3 = new Web(new Web.providers.HttpProvider(“http://localhost:8545”));*

*var contractAddr = “0x83749859b68897d79ac768786” //合约的地址，发布合约后产生*

*var abi = [{…… ……}] //合约的abi，通过 solc --abi Coin.sol(即合约名) 得到*

*var CoinContract = web3.eth.contract(abi) //得到一个抽象的合约实例*

*var contractInstance = CoinContract.at(contractAddr); //初始化合约实例*

*contractInstance.Sent(“latest”, (err,res)=>{ //监听区块事件Sent()事件，每产生一个新的豆能监听到*

*if(err)*

*console.log("Error: ”,err);*

*else*

*console.log(“Sent Event occurs: ”,res);*

*});*

node执行脚本文件

*node sentEvent\_listener.js*

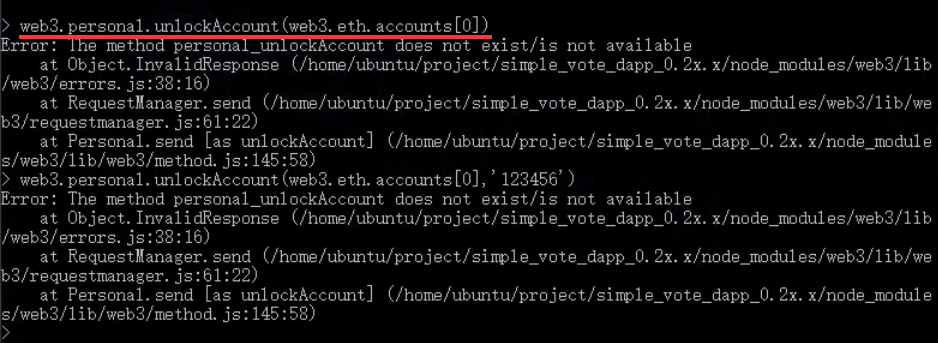
后台执行脚本，命令结尾加 & ，但可能看不到监听结果

*node sentEvent\_listener.js* ***&***

# 常见问题

1. **不能执行 web3.personal.unlockAccount()**

node进入使用web3模块命令时，**不能执行 web3.personal.unlockAccount()**命令，是因为eth结点没有授权给web3.personal来使用” 账户解锁”命令权限，涉及安全问题。若想使这个命令能执行，需要在启动geth时就打开相关权限。



**解决：<1>geth启动时赋权限**

***nohup*** *geth --datadir . --networkid 15 --rpc --rpcapi “eth,personal” 2>ouput.log* ***&***

*<1> nohup…& //用于后台启动*

*<2> --rpcapi “eth,personal…” //开启相关权限，多个权限逗号隔开*

**<2> web3使用命令行即可：**

*web3.personal.unlockAccount(web3.eth.accounts[0], ‘123456’) //需要输入密码*

该命令也可以写到脚本中，故进一步完善脚本：

*var Web = require(‘web3’);*

*var web3 = new Web(new Web.providers.HttpProvider(“http://localhost:8545”));*

*var arguments = process.argv.slice(2)*

*if(!arguments || arguments.length!=2){*

*console.log(“Parameter length must be 2”);*

*return;*

*}*

*var \_from = web3.eth.accounts[0];*

*var \_to = arguments[0];*

*var amount = arguments[1];*

*//合约的abi,通过合约文件目录下执行 solc –abi Coin.sol(即合约名) 得到*

*var abi = [{"inputs":[],"stateMutability":"nonpayable","type":"constructor"},{"anonymous":false,"inputs":[{"indexed":false,"internalType":"address","name":"from","type":"address"},{"indexed":false,"internalType":"address","name":"to","type":"address"},{"indexed":false,"internalType":"uint256","name":"amount","type":"uint256"}],"name":"Sent","type":"event"},{"inputs":[{"internalType":"address","name":"","type":"address"}],"name":"balance","outputs":[{"internalType":"uint256","name":"","type":"uint256"}],"stateMutability":"view","type":"function"},{"inputs":[{"internalType":"address","name":"receiver","type":"address"},{"internalType":"uint256","name":"amount","type":"uint256"}],"name":"mint","outputs":[],"stateMutability":"nonpayable","type":"function"},{"inputs":[],"name":"minter","outputs":[{"internalType":"address","name":"","type":"address"}],"stateMutability":"view","type":"function"},{"inputs":[{"internalType":"address","name":"receiver","type":"address"},{"internalType":"uint256","name":"amount","type":"uint256"}],"name":"send","outputs":[],"stateMutability":"nonpayable","type":"function"}];*

*var CoinContract = web3.eth.contract(abi); //实例化abi，此时还没有和区块链产生实际联系*

*var cointractAddress = “0x814865b5d53562f6556f320056667d695c568a”; //合约地址，部署合约完成后会返回该地址*

*var contractInstance = CoinContract.at(contractAddress); //通过合约地址实例化合约对象，赋值给合约对象，****此处不是重新部署****，*

*//授权后，可直接写到脚本中。但是密码是暴露的，有风险。接回调函数*

*web3.personal.unlockAccount(\_from, ‘123456’, (err,res)=>{*

*if(err)*

*console.log(“Error: ”, err);*

*else*

*//\_to:给谁发币；amount:发的自己代币的数量；*

*//交易本身：{from:发送者合约地址，to:即交易发给谁，接收者合约地址(此处可不写)，value:即给合约地址转的以太数(此处可不写)}*

*contractInstance.send(\_to, amount, {from: \_from},(err,res)=>{*

*if(err)*

*console.log("Error: ”, err);*

*else*

*console.log(“Result: ”, res);*

*})*

*})*

# DAPP实现

## 项目说明

**简单投票DAPP**，构建这个应用的主要步骤如下：

1. 首先安装一个叫做ganache的模拟区块链，能够让程序在开发环境中运行。
2. 写一个合约部署到ganache上。
3. 通过命令行、网页与ganache进行交互。

与区块链通信的方式使用过RPC(Remote Procedure Call)。web3js是一个JavaScript库，他抽象出了所有的RPC调用，以便于你可以通过JavaScript与区块链进行交互。另一个好处是，web3js能够让你使用你最喜欢的JavaScript框架构建非常棒的web应用

## 环境准备

下面是基于Linux的安装指南，这要求我们预先安装nodejs和npm，再用npm安装ganache-cli、web3和solc，就可以继续项目的下一步了

*# mkdir simple\_voting\_dapp*

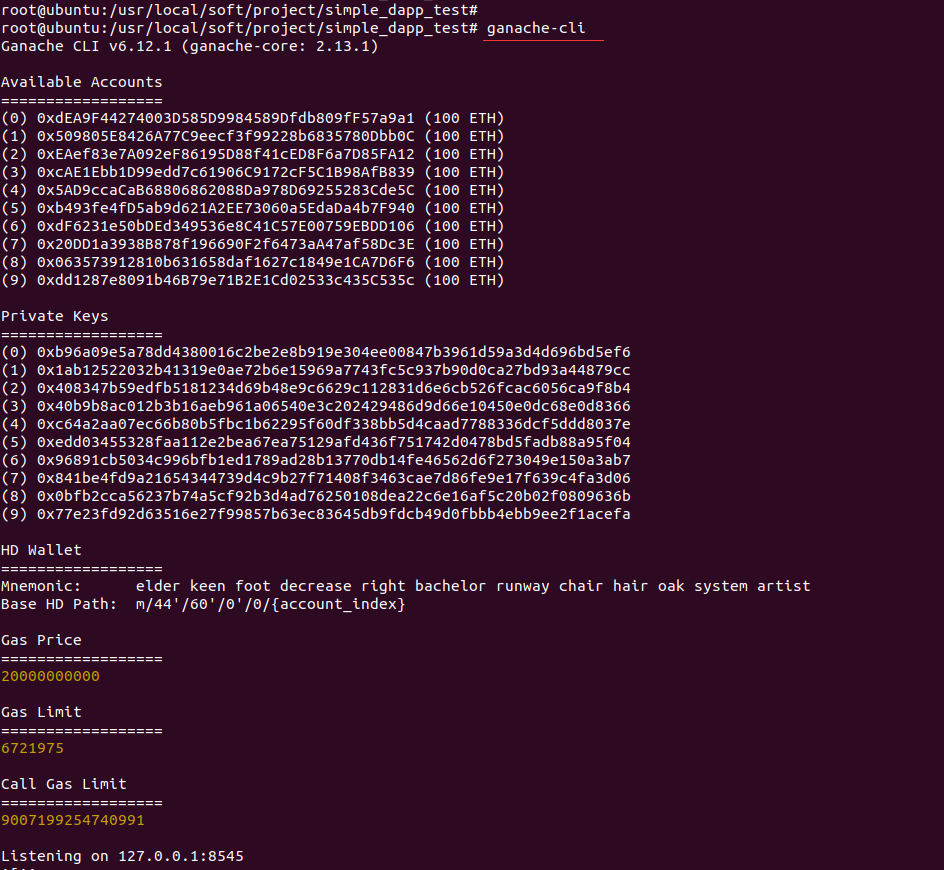
*# cd simple\_voting\_dapp*

*# npm init*

*# npm install ganache-cli web3@0.20.1 solc*

*# node\_modules/.bin/ganache-cli*

如果安装成功，运行命令node\_modules/.bin/ganache-cli会看到类似下图所示：



## Solidity合约

写一个叫做Voting的合约，内容包括

1. 一个构造函数，用来初始化一些候选者
2. 一个用来投票的方法（对投票数+1）
3. 一个返回候选者获得的总票数的方法

当把合约部署到区块链的时候，就会调用构造函数，并指调用一次。与web世界里每次部署代码都会覆盖旧代码不同，在区块链上部署的合约是不可改变的，也就是说，如果你更新合约并在此部署，旧的合约仍然会在区块链上存在，并且数据仍在。西部属的将会创建合约的一个新的实例。

### 3.1 编写合约代码

*pragma solidity ^0.4.22;*

*contract Voting{*

*mapping (bytes32 => uint8) public votesReceived; //每个候选人获取的投票数*

*bytes32[] public candidateList; //候选者列表*

*//构造函数*

*constructor(byte32[] candidateNames) public{*

*candidateList = candidateNames;*

*}*

*//判断候选人是否在名单中*

*function validCandidate(bytes32 candidateName) internal view returns(bool){*

*for(uint8 i = 0; i < candidateList.length; i++){*

*if(candidateName == candidateList[i])*

*return true;*

*}*

*return false;*

*}*

*//投票功能*

*function voteForCandidate(bytes32 candidateName) public{*

*require(validCandidate(candidateName));*

*votesReceived[candidateName] += 1;*

*}*

*//查询投票总数*

*function totalVotesFor(bytes32 candidateName) view public returns(uint8) {*

*require(validCandidate(candidateName));*

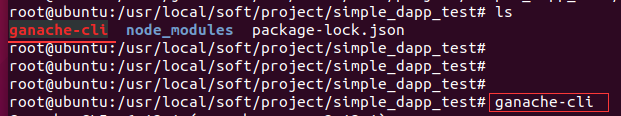
*return votesReceived[candidateName];*

*}*

*}*

### 3.2 编译、部署合约

* 在一个窗口启动ganache-cli



* 在另一个窗口，**进入sol合约文件所在文件夹**，进入node控制台，初始化web3

*# node*

*> var Web = require(‘web3’) //引入web3模块*

*> web3 = new Web(new Web.providers.HttpProvider(“http://localhost:8545”));*

* 测试连接ganachem，并向区块链查询获取所有账户。

*> web3.isConnected() //测试连接ganache-cli*

*> web3.eth.accounts (回车)*

*[‘0x364675348d7a98c90998b’,’0x67cb27635489d7586b6786a’…] //共10个*

* 启动solc，读取并编译合约源码

*> var fs = require(‘fs’) //引入f模块*

*> var solc = require(‘solc’) //引入solc模块*

*> sourceCode = fs.readFileSync(‘Voting.sol’).toString() //fs同步读取合约文件*

*> compiledCode = solc.compile(sourceCode) //编译sol源代码*

*> compiledCode //可查看编译后的文件内容*

* 部署合约文件

*> var abi = JSON.parse(compiledCode.contracts[‘:Voting’].interface) //读取abi并转JSON字符串*

*> abi //可查看转JSON串后的abi*

*> var byteCode = compiledCode.contracts[‘:Voting’].bytecode //读取字节码*

*> var VotingContract = web3.eth.contract(abi) //创建合约对象，还没有真正和区块链关联*

*> var deployTxObj = [data=byteCode, from=web3.eth.accounts[0], gas=3000000] //创建 交易对象,自己测试是“=”号*

*> var contractInstance = VotingContract.new([‘Alice,’Bob’,’Carry’],deployTxObj); //创建合约实例，提交合约，参数：构造函数参数 + 交易对象*

*> contractInstance.address //返回提交合约后，返回合约地址*

* 测试合约中的函数

*> contractInstance.voteForCandidate(“Alice”,{from: web3.eth.accounts[0]}) //执行投票函数*

*0x92739b987a897d0909c8b967a… … //执行后返回交易的hash码*

*> contractInstance.totalVotesFor(“Alice”).toString() //读取“Alice”获得的投票数*

## 网页交互

至此，大部分的工作都已完成，我们还需要做的事情就是创建一个简单的html，里面有候选者姓名并调用投票命令（我们已经在nodejs控制台里测试过）。我们可以在下面找到html代码和js代码，将他们放到小木目录，并在浏览器中打开index.html

**index.html**

<!DOCTYPE html>

<html>

<head>

<title>Voting DApp</title>

<link href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css" rel="stylesheet" type="text/css" >

</head>

<body class="container">

<h1>Simple Voting DApp</h1>

<div class="table-responsive">

<table class="table table-bordered">

<thead>

<tr>

<th>Candidate</th>

<th>Vote Count</th>

</tr>

</thead>

<tbody>

<tr>

<td>Alice</td>

<td id = “candidate-1”></td>

</tr>

<tr>

<td>Bob</td>

<td id = “candidate-2”></td>

</tr>

<tr>

<td>Cary</td>

<td id = “candidate-3”></td>

</tr>

</tbody>

</table>

<input type=”text” id="candidate"/>

<a href = “#” onclick="voteForCandidate" class="btn btn-primary">Vote</a>

</div>

</body>

<!-- 前端引入 web3 文件 -->

<script src=”https://cdn.jsdelivr.net/gh/ethereum/web3.js/dist/web3.min.js”></script>

<!-- 引入jquery -->

<script src="https://code.jquery.com/jquery-3.1.1.slim.min.js"></script>

<!-- 自定义的js文件 -->

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

</html>

**index.js**

//此处使用new 是因为html页面中引入jquery

var web3 = new web3(new web3.providers.HttpProvider("http://localhost:8545"));

**//sol文件的abi，替换自己的**

var abi = JSON.parse(‘…….’);

**//部署sol合约 后返回的合约地址**

var contractAddr = "";

var VotingContract = web3.eth.contract(abi);

var contractInstance = VotingContract.at(contractAddr);

var candidates = {"Alice":"candidate-1","Bob":"candidate-2","Cary":"candidate-3"};

**//投票**

function voteForCandidate(){

let candidateName = $(“#candidate”).val();

try{

contractInstance.vote(candidateName,{from:web3.eth.accounts[0]},(err,res)=>{

if(err)

console.log("Error: ”,err);

else{

let id = candidates[candidateName];

let count = contractInstance.totalVotesFor(candidateName).toString();

$(“#” + id).html(count);

}

})

}catch(err){}

}

$(document).ready({

var candidateList = Object.keys(candidates);

for(let i = 0; i < candidateList.length; i++){

let name = candidateList[i];

let count = contractInstance.totalVotesFor(name).toString();

$("#" + candidates[name]).html(count);

}

});

# node index.js

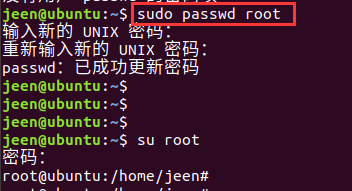
# 自动编译脚本

编译事对合约进行部署、测试的前置步骤，编译步骤的目标是把源码转换成ABI和bytecode，并且能够处理编译时抛出的错误，确保不会在包含才哦呜的源代码上编译

# 其他问题汇总

## 设置root密码

# sudo passwd root

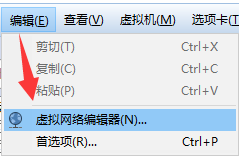


## 设置网络连接

虚拟机中ubuntu系统不能联网，可如下设置：

**2.1**关闭虚拟机

**2.2** 编辑-🡪虚拟网络编辑器，然后进行设置即可





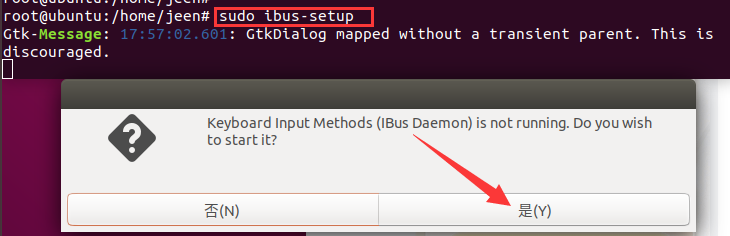
## 设置拼音输入法

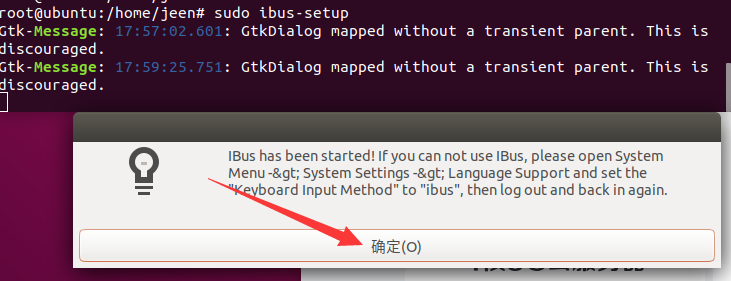
# sudo apt-get install ibus ibus-pinyin

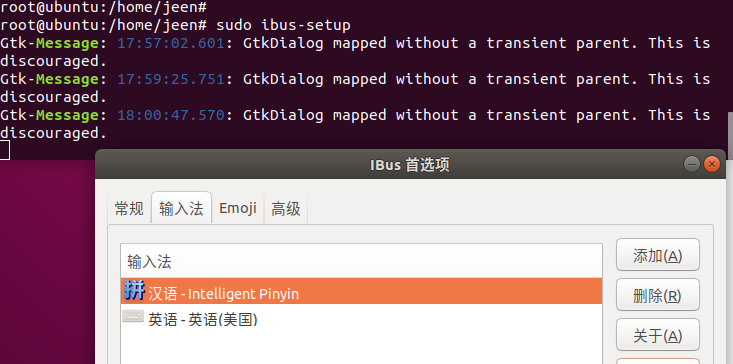
# sudo ibus-setuo //

# reboot //重启系统

然后进行“区域和语言”设置，选择需要的输入法，过程如下：







重启后设置“区域和语言”，选择输入法







## 安装 vim

# sudo apt-get install vim

## 多个版本gcc/g++切换

gcc g++系统默认版本使7.0+，安装多个版本后，若要更改其他版本为默认版本，可以更改优先级来实现

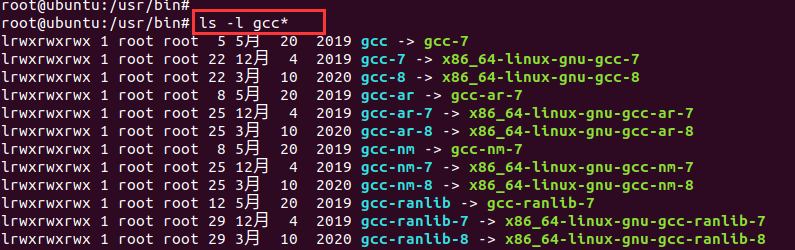
# ls -l gcc\* //查看所有gcc\g++版本

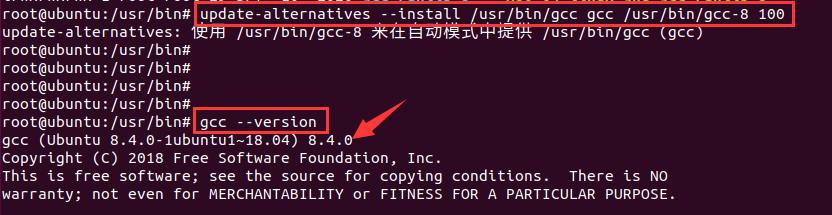
# update-alternatives --install /usr/bin/gcc gcc /ust/bin/**gcc-8 100** //设置gcc-8版本为最高优先级，0~100，数越大级别越高

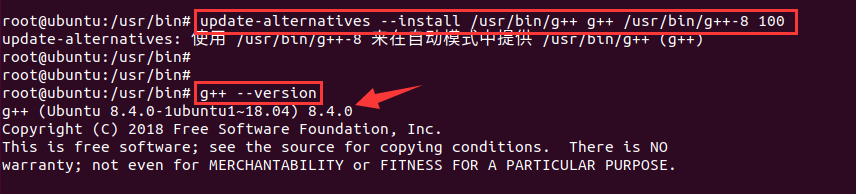
# update-alternatives --install /usr/bin/g++ g++ /usr/bin/**g++-8 100** //设置g++-8版本为最高优先级，0~100，数越大级别越高

# gcc --version //查看gcc版本号

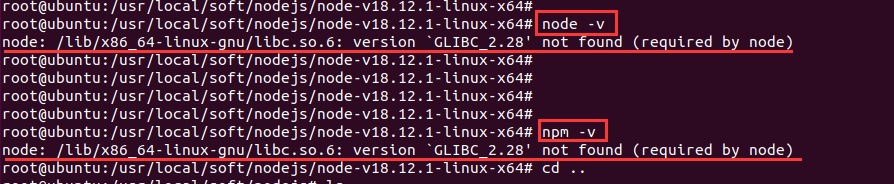
# g++ --version //查看g++版本号





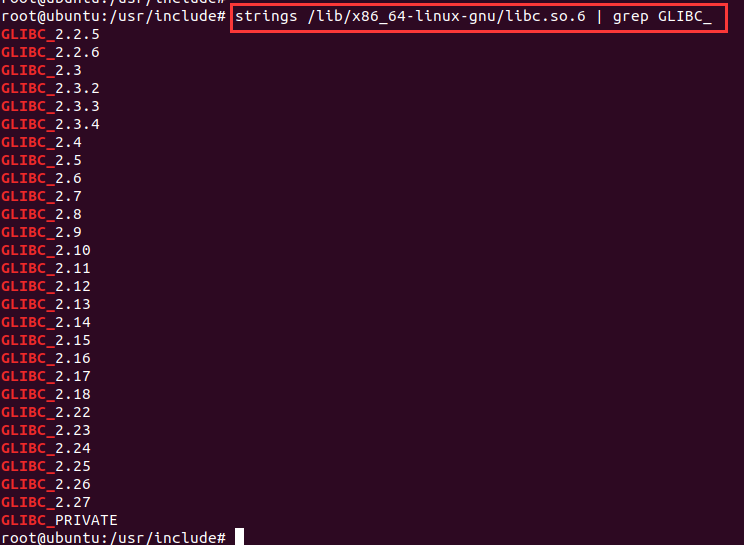


## 报错 ‘GLIBC\_2.28’ not found



### 2.1执行以下代码查看已安装的版本

# strings /lib/x86\_64-linux-gnu/libc.so.6 | grep GLIBC\_



### 2.1去官网下载文件

下载地址：<http://ftp.gnu.org/gnu/glibc/>

### 2.3 解压安装配置

# tar -zxvf glibc-2.28.tar.gz //解压

# cd glibc-2.28 //进入解压后的文件

# mkdir build //进入后，创建文件夹 build

# cd build

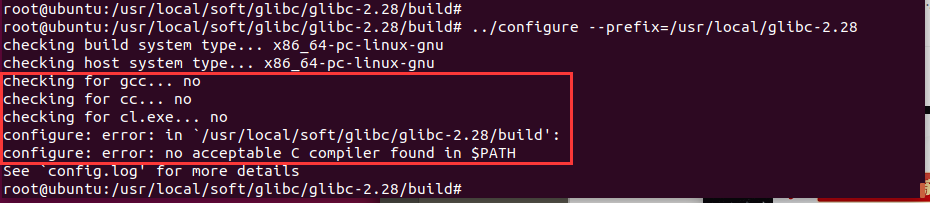
// 进行配置，

# ../configure --prefix=/usr --disable-profile --enable-add-ons --with-headers=/usr/include --with-binutils=/usr/bin

// 或者进行简单配置 ../configure --prefix=/usr/local/glibc-2.28

# make -j8 && make install //编译、执行, make -j8(使用8个核心编译目标) 或者简单执行 make && make install

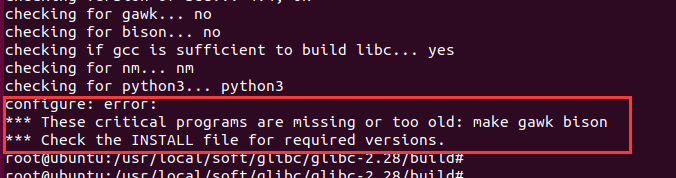
## 报错 配置gilbc-2.28软连接时报错



缺少C编译器，安装即可：

# apt install gcc

## 报错 配置glibc-2.28软链接时报错



缺少 make、gawk、bison，分别进行安装即可，命令如上安装gcc

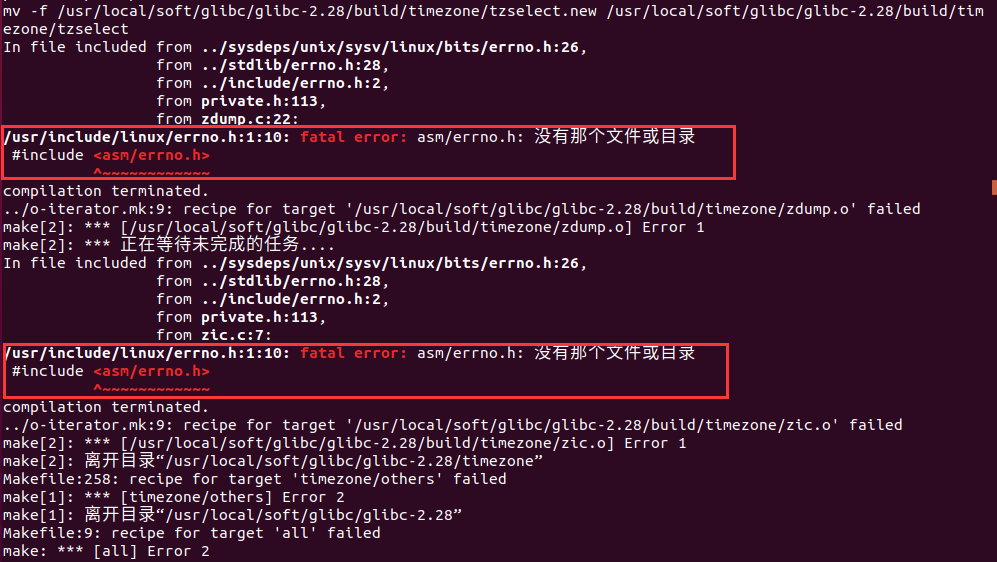
## 报错 These critical programs are missing or too old : gawk bison

// 安装gawk bison 即可

# sudo apt-get install gawk bison

## 报错 fatal error : asm/error.h

编译报错/usr/include/linux/errno.h:1:10: fatal error: asm/errno.h



// 查找你系统里asm文件位置

# find / -name asm

// 创建软连接

# sudo ln -s 找到的路径 /usr/include/asm

#类似 sudo ln -s /usr/i686-linux-gnu/include/asm /usr/include/asm

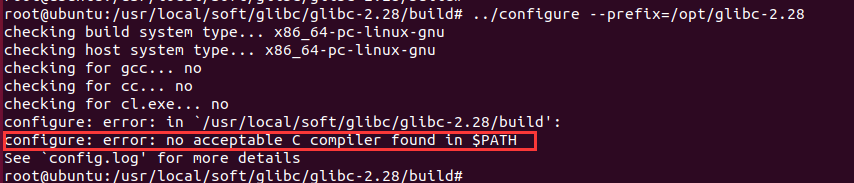
或 sudo ln -s /usr/**include/x86\_64-linux-gnu/asm** /usr/include/asm

找到的路径略有差异，看具体情况而定

## 报错 no acceptable C compiler found in $PATH

新安装的 Ubuntu 系统在执行 ./configure 编译软件时会提示如下错误消息：

configure: error: no acceptable C compiler found in $PATH



这是没有安装 C 编译器，在 Ubuntu 系统下可以执行以下命令来安装 C 编译环境：

# sudo apt-get install -y build-essential

完成安装后，再次执行就 OK 了。

