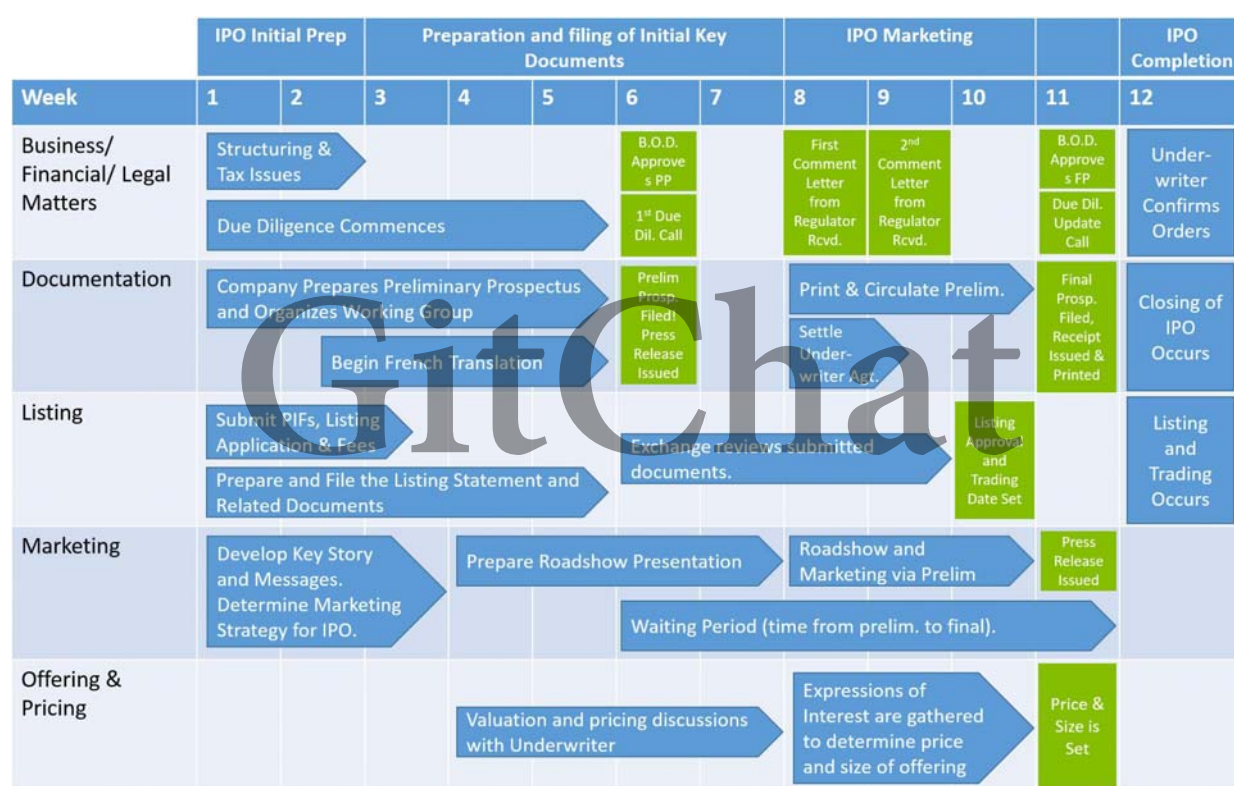


韭菜种植与收割：发布你自己的 ICO

讲在前面

你可曾想梭哈全部存款，参与 ICO，一夜身价暴涨千倍，获得财富自由，从此走上人生巅峰？

ICO 是借用 IPO 生造出来的一种概念，同样具有非常相似的募资机制，但 IPO 有着严格的上市流程、政策监管，如下图所示。



即便如此，参与 IPO 仍然有着相当大的风险，且为股市带来了相当大的不稳定因素。而与 ICO 比起来简直就是小巫见大巫了。一家公司想要进行 IPO 起码要达到能够上市的标准，而想发布 ICO 你只要有一个好听的 idea 就足够了。并且严重缺乏监管，虽然各国政府都在不断发出声明，但截至本分享写作前，也没有正式出台比较明朗的有关规定。这也导致无数的空气项目披着虚拟货币和区块链的高科技壳，到处招摇撞骗割韭菜，有过之无不及的还搞什么 AI+区块链，IOT+区块链，技术名词堆积越多的项目，死得往往越快。甚至一些有头有脸的大公司，也忍不住打打擦边球，收割一波，炒作炒作，股价就能翻几个涨停。

可就像马老爷子说的：

如果有10%的利润，它就保证到处被使用；有20%的利润，它就活跃起来；有50%的利润，它就铤而走险；为了100%的利润，它就敢践踏一切人间法律；有300%的利润，它就敢犯任何罪行，甚至绞首的危险。

即便如此，仍然有很多人跃跃欲试不信邪。这一场 Chat 就手把手教你为 ICO 做好所有技术面上的准备。在和大家一起点亮新技能的同时，也揭一揭所谓 ICO 的老底。

使用到的技术栈

目前市场上99%的项目 ICO 都是基于以太坊（Ethereum）智能合约（Smart Contracts）技术发布的 token（ERC20 Token）。本次分享也是基于这一套技术栈，介绍内容包括以下几个方面。

- 本地开发环境构建
- 以太坊智能合约开发
 - ERC20 Token 合约开发
 - ICO Crowdsale 合约开发
 - 补充说明与权限控制
 - 合约的发布及调试
 - 本地开发环境发布
 - 线上测试网络发布
 - 主网络发布
- Dapp 开发
 - web3.js 的使用
 - Metamask 简介
 - truffle-contract 的使用
 - ICO 前端应用开发
- Dapp 部署
 - IPFS 简介
 - 发布应用
 - 域名解析
 - IPNS
 - Nginx 反向代理

使用到的技术栈包括：

- [Truffle](#)
- [Ganache](#)
- [Metamask](#)
- [Solidity](#)
- [openzeppelin](#)
- [Infura](#)

- [web3.js](#)
- [truffle-contract](#)
- [ipfs](#)

对读者的基本要求有：

- 了解编程
- 会 JavaScript

本地开发环境构建

以太坊官方提供的 [Mist](#) 和 [Ethereum-Wallet](#)（其中 Mist 是一个可以用来访问 Dapp 的浏览器，Ethereum-Wallet 是 Mist 的一个独立发布版本，也算是浏览器，但只能用来访问以太坊钱包这个应用）在网络同步过程中或多或少都会遇到问题，而且目前网络拥堵，完整节点过大，同步完成相当困难。但事实上我们进行以太坊开发时并不需要同步完整的节点，也可以选择使用相应的模拟开发环境。

Truffle

Truffle 框架为你提供本地进行智能合约开发的所有依赖支持，使你可以在本地进行智能合约及 Dapp 的开发、编译、发布。安装非常简单，只需要：

```
npm install -g truffle
```

Ganache

Ganache 也是 Truffle 框架中提供的一个应用，可以在你的本地开启模拟一个以太坊节点，让你能够将开发好的智能合约发布至本地测试节点中运行调试。安装也非常简单，官网下载即可，双击打开运行。

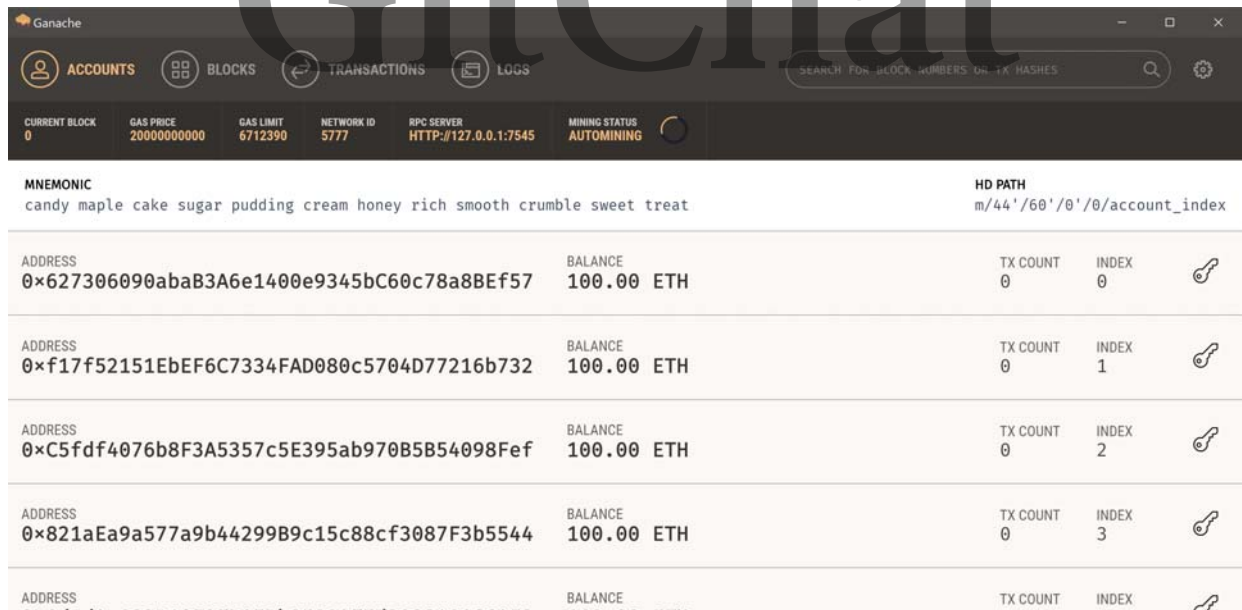
不过这里有一个隐藏的坑，如果你使用的是 Windows 系统的话，Ganache 提供的是后缀名为 `.appx` 的 Windows 应用商店版安装包。你需要打开 Windows 设置 -> 系统 -> 针对开发人员 -> 选择“旁加载应用”这个选项。

GitChat



确认之后就可以双击 Ganache.appx 进行安装了，假如系统仍然无法识别这一后缀名，你可以手动打开 powershell 输入如下命令进行安装。

```
Add-AppxPackage .\Ganache.appx
```



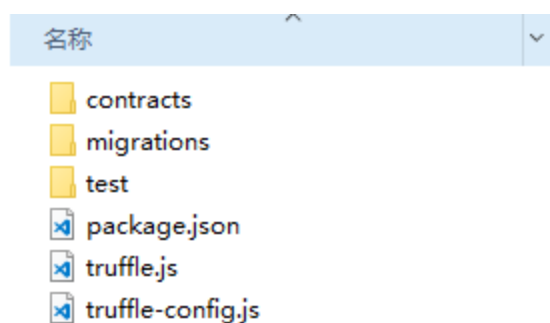
至此本地开发智能合约及 Dapp 的环境就算安装完成了，Truffle 官方提供了许多示例教程以及应用脚手架（truffle box），其中就包括教你开发以太坊宠物商店的教程等内容，在此不再赘述，感兴趣的同学自己动手可以试试。

以太坊智能合约开发

首先使用 Truffle 初始化我们的项目，命令如下。

```
mkdir my-ico
cd my-ico
npm init -y
truffle init
```

脚本运行完成之后 Truffle 会自动为我们的项目创建一系列文件夹和文件，如下图所示。这里又有一个隐藏的坑，如果你使用 Windows 命令行的话，需要删掉 `truffle.js` 文件，否则在项目目录执行 `truffle` 相关命令时，CMD 会混淆 `truffle` 与 `truffle.js` 文件，因此，你应该将配置写在 `truffle-config.js` 文件当中。



ERC20 Token 合约开发

现在我们的项目目录大概是这个样子：

- contracts/
 - Migrations.sol
- migrations/
 - 1_initial_migration.js
- test/
- package.json
- truffle-config.js 或 truffle.js

我们在编写智能合约时，需要在 `contracts` 目录下新建相应的智能合约文件。在以太坊开发智能合约的编程语言叫做 [Solidity](#)，它是一种在语法上非常类似 JavaScript 的语言，其后缀名为 `.sol`，例如在这里我们可以创建一个名为 `GitCoin.sol` 的文件，命令如下。

```
// *nix
touch GitCoin.sol
// win
copy NUL > GitCoin.sol
```

ERC20 ([Ethereum Request for Comments NO.20](#)) 是官方发行的 token 标准。如果你希望你发布的 token 能够在以太坊网络上流通、上市交易所、支持以太坊钱包，在开发 token 的合约时就必须遵从这一规范。

ERC20 规定了合约中的一系列变量、方法、事件，你可以参考官网教程 [Create your own CRYPTO-CURRENCY with Ethereum](#) 当中的示例代码：

```
pragma solidity ^0.4.16;

interface tokenRecipient { function receiveApproval(address
_from, uint256 _value, address _token, bytes _extraData) public;
}

contract TokenERC20 {
    // Public variables of the token
    string public name;
    string public symbol;
    uint8 public decimals = 18;
    // 18 decimals is the strongly suggested default, avoid
    changing it
    uint256 public totalSupply;

    // This creates an array with all balances
    mapping (address => uint256) public balanceOf;
    mapping (address => mapping (address => uint256)) public
    allowance;

    // This generates a public event on the blockchain that will
    notify clients
    event Transfer(address indexed from, address indexed to,
    uint256 value);

    // This notifies clients about the amount burnt
    event Burn(address indexed from, uint256 value);

    /**
     * Constrctor function
     *
     * Initializes contract with initial supply tokens to the
    creator of the contract
     */
    function TokenERC20(
        uint256 initialSupply,
        string tokenName,
        string tokenSymbol
    ) public {
        totalSupply = initialSupply * 10 ** uint256(decimals);
    // Update total supply with the decimal amount
        balanceOf[msg.sender] = totalSupply; //
    Give the creator all initial tokens
    }
```

```

        name = tokenName;                                //
Set the name for display purposes
        symbol = tokenSymbol;                            //
Set the symbol for display purposes
    }

/**
 * Internal transfer, only can be called by this contract
 */
function _transfer(address _from, address _to, uint _value)
internal {
    // Prevent transfer to 0x0 address. Use burn() instead
    require(_to != 0x0);
    // Check if the sender has enough
    require(balanceOf[_from] >= _value);
    // Check for overflows
    require(balanceOf[_to] + _value > balanceOf[_to]);
    // Save this for an assertion in the future
    uint previousBalances = balanceOf[_from] +
balanceOf[_to];
    // Subtract from the sender
    balanceOf[_from] -= _value;
    // Add the same to the recipient
    balanceOf[_to] += _value;
    Transfer(_from, _to, _value);
    // Asserts are used to use static analysis to find bugs
in your code. They should never fail
    assert(balanceOf[_from] + balanceOf[_to] ==
previousBalances);
}

/**
 * Transfer tokens
 *
 * Send `_value` tokens to `_to` from your account
 *
 * @param _to The address of the recipient
 * @param _value the amount to send
 */
function transfer(address _to, uint256 _value) public {
    _transfer(msg.sender, _to, _value);
}

/**
 * Transfer tokens from other address
 *
 * Send `_value` tokens to `_to` on behalf of `_from`
 *
 * @param _from The address of the sender
 * @param _to The address of the recipient
 * @param _value the amount to send
 */

```

```

    function transferFrom(address _from, address _to, uint256
_value) public returns (bool success) {
    require( _value <= allowance[_from][msg.sender]);    //
Check allowance
    allowance[_from][msg.sender] -= _value;
    _transfer(_from, _to, _value);
    return true;
}

/**
 * Set allowance for other address
 *
 * Allows `_spender` to spend no more than `_value` tokens on
your behalf
 *
 * @param _spender The address authorized to spend
 * @param _value the max amount they can spend
 */
function approve(address _spender, uint256 _value) public
returns (bool success) {
    allowance[msg.sender][_spender] = _value;
    return true;
}

/**
 * Set allowance for other address and notify
 *
 * Allows `_spender` to spend no more than `_value` tokens on
your behalf, and then ping the contract about it
 *
 * @param _spender The address authorized to spend
 * @param _value the max amount they can spend
 * @param _extraData some extra information to send to the
approved contract
 */
function approveAndCall(address _spender, uint256 _value,
bytes _extraData)
    public
    returns (bool success) {
        tokenRecipient spender = tokenRecipient(_spender);
        if (approve(_spender, _value)) {
            spender.receiveApproval(msg.sender, _value, this,
_extraData);
            return true;
        }
    }

/**
 * Destroy tokens
 *
 * Remove `_value` tokens from the system irreversibly
 *

```



```

    * @param _value the amount of money to burn
    */
    function burn(uint256 _value) public returns (bool success) {
        require(balanceOf[msg.sender] >= _value); // Check if
the sender has enough
        balanceOf[msg.sender] -= _value;           // Subtract
from the sender
        totalSupply -= _value;                     // Updates
totalSupply
        Burn(msg.sender, _value);
        return true;
    }

    /**
    * Destroy tokens from other account
    *
    * Remove `_value` tokens from the system irreversibly on
behalf of `_from`.
    *
    * @param _from the address of the sender
    * @param _value the amount of money to burn
    */
    function burnFrom(address _from, uint256 _value) public
returns (bool success) {
        require(balanceOf[_from] >= _value); //
Check if the targeted balance is enough
        require(_value <= allowance[_from][msg.sender]); //
Check allowance
        balanceOf[_from] -= _value;           //
Subtract from the targeted balance
        allowance[_from][msg.sender] -= _value; //
Subtract from the sender's allowance
        totalSupply -= _value;               //
Update totalSupply
        Burn(_from, _value);
        return true;
    }
}

```

我只是想割韭菜而已，用得着写几百行代码吗？

当然不必，这时我们就需要使用到智能合约开发框架 [OpenZeppelin](#)，安装命令如下。

```
npm install zeppelin-solidity --save
```

GitCoin.sol

引入 OpenZeppelin，代码如下。

```
// 声明 solidity 编译版本
pragma solidity ^0.4.18;
// 引入框架为我们提供的编写好的 ERC20 Token 的代码
import "zeppelin-solidity/contracts/token/StandardToken.sol";
// 通过 is 关键字继承 StandardToken
contract GitToken is StandardToken {

    string public name = "GitToken"; // Token 名称
    string public symbol = "EGT"; // Token 标识 例如: ETH/EOS
    uint public decimals = 18; // 计量单位, 和 ETH 保持一样就设置为 18
    uint public INITIAL_SUPPLY = 10000 * (10 ** decimals); // 初始供应量

    // 与 contract 同名的函数为本 contract 的构造方法, 类似于 JavaScript 当中的 constructor
    function GitToken() {
        totalSupply = INITIAL_SUPPLY; // 设置初始供应量
        balances[msg.sender] = INITIAL_SUPPLY; // 将所有初始 token 都存入 contract 创建者的余额
    }
}
```

好了, 至此一个可以用来交易的符合 ERC20 标准的 token 就编写完毕了。就这么简单? 就这么简单! 当然智能合约的功能不止如此, token 中可以玩转设计的地方也不止这些, 不过我们要稍微放在后面一些来讨论, 接下来还是赶快着手 ICO 合约开发, 为我们的项目募集资金吧。

ICO Crowdsale 合约开发

同样, 以太坊官网文档在教程 [CROWDSALE Raising funds from friends without a third party](#) 中也为我们提供了用来 crowdsale 做 ICO 募资的示例代码:

```
pragma solidity ^0.4.18;

/**
 * interface 的概念和其他编程语言当中类似, 在这里相当于我们可以通过传参引用之前发布的 token 合约
 * 我们只需要使用其中的转账 transfer 方法, 所以就只声明 transfer
 */
interface token {
    function transfer(address receiver, uint amount);
}

contract Crowdsale {
    // 这里是发布合约时需要传入的参数
    address public beneficiary; // ICO 募资成功后的收款方
    uint public fundingGoal; // 筹多少钱
    uint public amountRaised; // 割到多少韭菜
}
```

```

uint public deadline; // 割到啥时候
/**
 * 卖多贵，即你的 token 与以太坊的汇率，你可以自己设定
 * 注意到，ICO 当中 token 的价格是由合约发布方自行设定而不是市场决定的
 * 也就是说你项目值多少钱你可以自己编
 */
uint public price;
token public tokenReward; // 你要卖的 token
mapping(address => uint256) public balanceOf;
bool fundingGoalReached = false; // 是否达标
bool crowdsaleClosed = false; // 售卖是否结束
/**
 * 事件可以用来记录信息，每次调用事件方法时都能将相关信息存入区块链中
 * 可以用作凭证，也可以在你的 Dapp 中查询使用这些数据
 */
event GoalReached(address recipient, uint totalAmountRaised);
event FundTransfer(address backer, uint amount, bool
isContribution);

/**
 * Constructor function
 *
 * Setup the owner
 */
function Crowdsale(
    address ifSuccessfulSendTo,
    uint fundingGoalInEthers,
    uint durationInMinutes,
    uint etherCostOfEachToken,
    address addressOfTokenUsedAsReward
) {
    beneficiary = ifSuccessfulSendTo;
    fundingGoal = fundingGoalInEthers * 1 ether;
    deadline = now + durationInMinutes * 1 minutes;
    price = etherCostOfEachToken * 1 ether;
    tokenReward = token(addressOfTokenUsedAsReward); // 传入已
发布的 token 合约的地址来创建实例
}

/**
 * Fallback function
 *
 * payable 用来指明向合约付款时调用的方法
 */
function () payable {
    require(!crowdsaleClosed);
    uint amount = msg.value;
    balanceOf[msg.sender] += amount;
    amountRaised += amount;
    tokenReward.transfer(msg.sender, amount / price);
    FundTransfer(msg.sender, amount, true);
}

```

```

/**
 * modifier 可以理解为其他语言中的装饰器或中间件
 * 当通过其中定义的一些逻辑判断通过之后才会继续执行该方法
 * _ 表示继续执行之后的代码
 */
modifier afterDeadline() { if (now >= deadline) _; }

/**
 * Check if goal was reached
 *
 * Checks if the goal or time limit has been reached and ends
the campaign
 */
function checkGoalReached() afterDeadline {
    if (amountRaised >= fundingGoal){
        fundingGoalReached = true;
        GoalReached(beneficiary, amountRaised);
    }
    crowdsaleClosed = true;
}

/**
 * Withdraw the funds
 *
 * Checks to see if goal or time limit has been reached, and
if so, and the funding goal was reached,
 * sends the entire amount to the beneficiary. If goal was
not reached, each contributor can withdraw
 * the amount they contributed.
 */
function safeWithdrawal() afterDeadline {
    if (!fundingGoalReached) {
        uint amount = balanceOf[msg.sender];
        balanceOf[msg.sender] = 0;
        if (amount > 0) {
            if (msg.sender.send(amount)) {
                FundTransfer(msg.sender, amount, false);
            } else {
                balanceOf[msg.sender] = amount;
            }
        }
    }

    if (fundingGoalReached && beneficiary == msg.sender) {
        if (beneficiary.send(amountRaised)) {
            FundTransfer(beneficiary, amountRaised, false);
        } else {
            //If we fail to send the funds to beneficiary,
unlock funders balance
            fundingGoalReached = false;
        }
    }
}

```

```

    }
  }
}

```

至此我们的 ICO 合约也开发完毕了，基本上一行代码都没有写，只是改了几个参数，一个键盘上只有三个按键的程序员都能够完成这类智能合约的开发，没有比这更友好的编程体验了。虽然 solidity 是一种非图灵完备的编程语言，但我们仍然能够用它编写许多逻辑。

上述的 ICO 示例代码写得算比较客气的一种，在最后的提款方法中，如果筹资达标，ICO 发布方则可以取走所有筹款，而如果未达标，参与者则能够取回自己的投资，由合约来持有所有款项。但事实上，我们仍然可以随意修改其中的逻辑，看下面代码。

```

function () payable {
  require(!crowdsaleClosed);
  uint amount = msg.value;
  balanceOf[msg.sender] += amount;
  amountRaised += amount;
  tokenReward.transfer(msg.sender, amount / price);
  // 每次有人付款直接取走筹资
  beneficiary.send(amountRaised);
  amountRaised = 0;
  FundTransfer(msg.sender, amount, true);
}
// 删除剩余代码

```

补充说明与权限控制

既然咱是铁了心来割韭菜的，如此简单的代码怎么能够满足咱的贪欲呢？一定要学比特币固定供给量吗？我是来卖 token 的呀，万一有一天卖完了怎么办，万一有人手里筹码比我自己都多了控盘怎么办，万一发的数量太多卖的不好怎么办？

事实上解决这些问题的逻辑全部都可以写在智能合约里。

Ownable token

在我们的潜在观念里，区块链自有不可变属性，这种不可变属性在一些狂热信徒的演绎当中变成了平权属性，甚至带有了共产主义色彩，仿佛拥抱区块链技术就能够为未来的人类文明带来希望，把人民从集权的手中解救出来。然而事实上这种不可变性同样是两面的，它能够带来的也包括所有权的不可变性。

ERC20 标准只规定了我们的合约中应该包含哪些方法，而没有限制合约中不能出现哪些方法，因此在之前的基础上，我们还可以继续编写一些特殊的方法，赋予合约发布者一些管理员特权，请看下面代码：

```

contract Ownable {
    address public owner;

    function Ownable() public {
        owner = msg.sender;
    }
    // 通过 onlyOwner 我们可以限定一些方法只有所有者才能够调用
    modifier onlyOwner {
        require(msg.sender == owner);
        _;
    }

    function transferOwnership(address newOwner) onlyOwner public
    {
        owner = newOwner;
    }
}
// 合约可以同时有多个继承
contract GitToken is StandardToken, Ownable {
    ...
}

```

MintableToken

接下来我们来解决 token 不够卖的问题，万一我的 initial offer 卖断货了怎么办，万一我卖完一次还想卖怎么办，这时我们就需要把 token 编写成为 MintableToken，在我们想增发的时候就能增发，代码如下：

```

// 用 onlyOwner 限定只有 token 的所有者才能够进行增发操作
function mint(address _to, uint256 _amount) onlyOwner public
returns (bool) {
    totalSupply_ = totalSupply_.add(_amount);
    balances[_to] = balances[_to].add(_amount);
    Mint(_to, _amount);
    Transfer(address(0), _to, _amount);
    return true;
}

```

BurnableToken

万一我们的 token 不小心发了太多，卖的时间久了贬值怎么办？当然是销毁了，可参照下面代码：

```

/**
 * Destroy tokens
 *
 * Remove `_value` tokens from the system irreversibly
 */

```

```

* @param _value the amount of money to burn
*/
function burn(uint256 _value) public returns (bool success) {
    require(balanceOf[msg.sender] >= _value);    // Check if the
sender has enough
    balanceOf[msg.sender] -= _value;              // Subtract from
the sender
    totalSupply -= _value;                        // Updates
totalSupply
    Burn(msg.sender, _value);
    return true;
}

```

万一有人手里的筹码太多，或者 token 被竞争对手买走了怎么办？没关系，我们还可以指定销毁某一账户中的 token，请看下面代码：

```

/**
 * Destroy tokens from other account
 *
 * Remove `_value` tokens from the system irreversibly on behalf
of `_from`.
 *
 * @param _from the address of the sender
 * @param _value the amount of money to burn
 */
function burnFrom(address _from, uint256 _value) public returns
(bool success) {
    require(balanceOf[_from] >= _value);          // Check if
the targeted balance is enough
    require( _value <= allowance[_from][msg.sender]); // Check
allowance
    balanceOf[_from] -= _value;                  // Subtract
from the targeted balance
    allowance[_from][msg.sender] -= _value;      // Subtract
from the sender's allowance
    totalSupply -= _value;                       // Update
totalSupply
    Burn(_from, _value);
    return true;
}

```

只要上述的方法全部都出现在合约里，我们发布的 token 就能够具备上述所有属性。这样一来，不够的时候我们可以发钱，发多了可以销毁，我们成功创建了属于自己的一所中央银行，甚至看某人不爽还能够指定销毁其账户存款，这哪里是平权，简直是超级集权。

而事实上，在已发布的 ERC20 token 当中，例如排名第一的 [EOS 的合约](#)里也是存在类似方法的，如下所示。

```

function mint(uint128 wad) auth stoppable note {
    _balances[msg.sender] = add(_balances[msg.sender], wad);
    _supply = add(_supply, wad);
}
function burn(uint128 wad) auth stoppable note {
    _balances[msg.sender] = sub(_balances[msg.sender], wad);
    _supply = sub(_supply, wad);
}

```

当然在其官方网站和白皮书中是标明了会发布多少 token，创始团队持有多少，投资人分配多少，公开发布多少，如何销毁等内容的。但白皮书又不具备法律效力，token 的所有权也不在你手里，万一人家哪天想要跑路或者中途变卦岂是咱能拦得住的。换个角度讲，假如你现在手里有一家可以印钱的公司，印多少就有多少，你印还是不印？

通过这一部分内容的介绍，我只是想要证明，智能合约本身并不具备可无条件信任的特性，充其量就是一段没法改一直跑的程序而已，你也可以在逻辑中加入管理员权限，token 的发布方并不比央行可信多少，只要所有者愿意可以随时进行修改。以太坊官方宣传的所谓“trustless”这一概念根本不成立。

Kickstart a project with a trustless crowdsale

Do you already have ideas that you want to develop on Ethereum? Maybe you need help and some funds to bring them to life, but who would lend money to someone they don't trust?

Using Ethereum, you can create a contract that will hold a contributor's money until any given date or goal is reached. Depending on the outcome, the funds will either be released to the project owners or safely returned back to the contributors. All of this is possible without requiring a centralized arbitrator, clearinghouse or having to trust anyone.

You can even use the token you created earlier to keep track of the distribution of rewards.

没有第三方担保，没有法律法规的维护，仅凭智能合约本身你的投资得不到任何保证。智能合约的不可变性反而给割韭菜的一方提供了巨大的便利，从前你看不惯某家公司还能够黑掉它的系统，获取管理员权限，如今所有程序都跑在区块链上，黑无可黑，集权永远都在合约发布者手里。讲到这里，希望你能理解这次分享的良苦用心，不要轻信任何 ICO 项目。

合约的发布及调试

本地开发环境发布

合约开发完成之后，我们需要编译并发布合约至区块链网络中，只需要进行以下两步操作。

首先在 migrations 文件夹下新建 2-deploy-contract.js 文件，配置部署脚本如下。

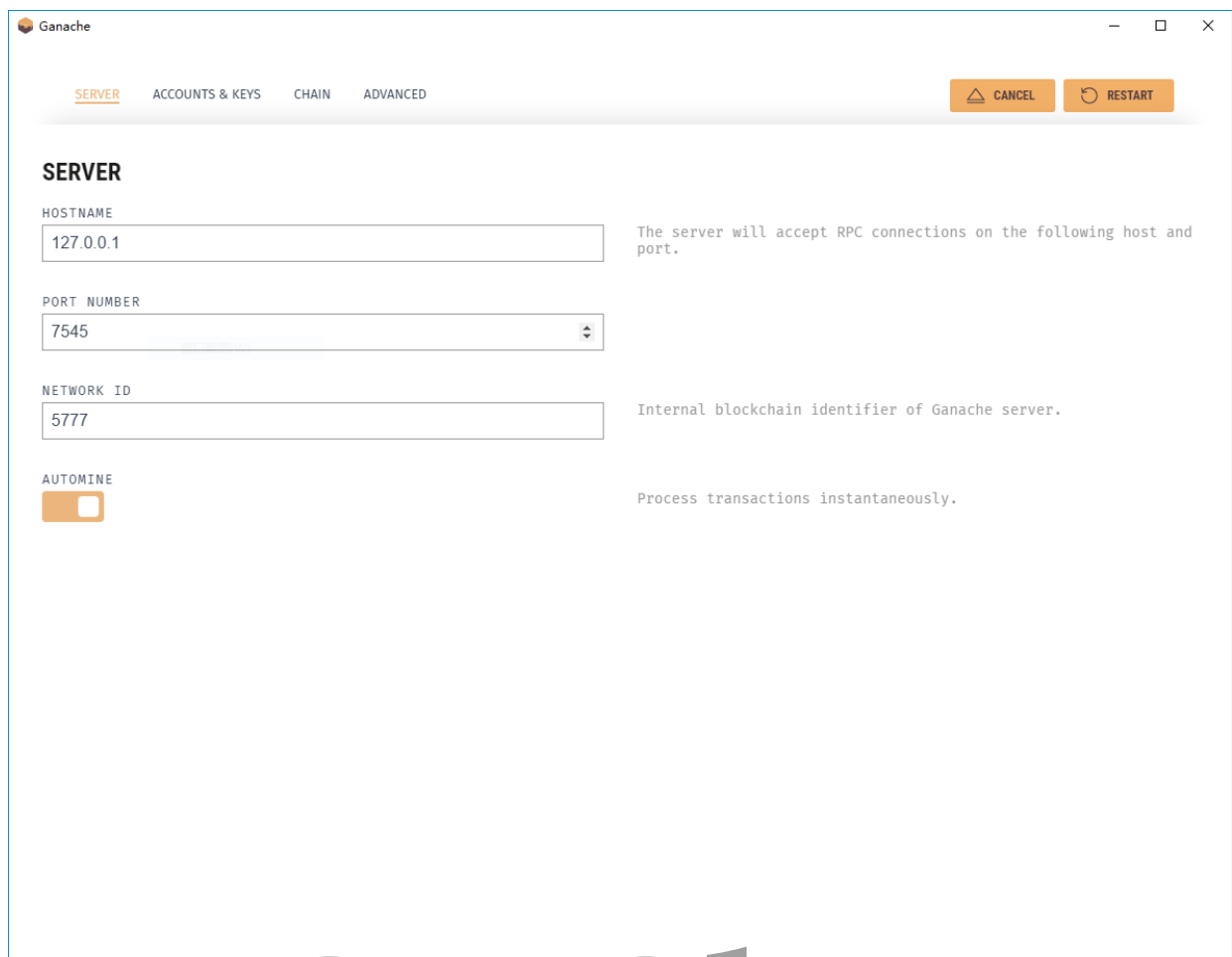

```
// 引入我们编写的合约
const GitCoin = artifacts.require("./GitCoin.sol")
const GitCoinCrowdsale =
artifacts.require("./GitCoinCrowdsale.sol")

module.exports = function(deployer, network, accounts) {
  // 设定参数，此处的参数即使传入合约构造方法的参数，与你自己编写的合约保持一致
  const ifSuccessfulSendTo = accounts[0] // 当前以太坊网络中的默认账户
  const fundingGoalInEthers = 1000
  const durationInMinutes = 36000000
  const etherCostOfEachToken = 0.01
  // 这里的 Promise 可以保证我们在发布完 token 合约之后再发布 ICO 合约，
  并将已发布 token 的地址作为参数传入
  deployer.deploy(GitCoin).then(function() {
    return deployer.deploy(GitCoinCrowdsale, ifSuccessfulSendTo,
    fundingGoalInEthers, durationInMinutes, etherCostOfEachToken,
    GitCoin.address);
  });
};
```

接着在 truffle-config.js 或 truffle.js 中设置发布网络，脚本如下。

```
module.exports = {
  networks: {
    development: {
      host: "127.0.0.1",
      port: 7545, // 与你本地的 ganache 设置保持一致
      network_id: "*" // Match any network id
    }
  }
};
```

现在只需要开启 Ganache:



然后在命令行中输入：

```
truffle compile
truffle migrate
```

你的合约就会顺利发布至测试网络中了。然后你可以输入：

```
truffle console
```

这样就能够进入本地的命令行调试了：

```
# 所有的合约方法都是 Promise 对象
truffle(development)> GitCoinCrowdsale.deployed().then(inst=>
{crowd=inst})
truffle(development)> GitCoin.deployed().then(inst=>{git=inst})
truffle(development)>
crowd.sendTransaction({from:web3.eth.accounts[0],value:web3.toWei
(1, "ether")})
truffle(development)>
git.mint(web3.eth.accounts[0],web3.toWei(100, "ether"))
```

线上测试网络发布

以太坊网络分为测试网和主网，在正式发布主网之前，我们可以先发送到测试网络进行调试。发布至以太坊网络也无需同步完整节点，我们可以使用 [Infura](#) 为我们提供的公共接口。

The image shows the Infura sign-up form. At the top, there is a large orange rectangle containing the Infura logo (a stylized 'I' made of three horizontal bars) and the word 'INFURA' in bold, black, uppercase letters. Below the logo, the text 'GET STARTED FOR FREE' is centered. The form itself is a white rectangle with a thin grey border. It contains the following elements: a label 'What is your name?' followed by two input fields for 'First' and 'Last' names; a label 'What is your email address? *' followed by a single input field; a section titled 'Mailing List' with a checkbox and the text 'Check to sign up for the INFURA™ News mailing list.'; a section titled 'Terms of Service *' with a checkbox and the text 'Check to acknowledge acceptance of our [Terms of Service](#)'; a reCAPTCHA widget with the text 'I'm not a robot' and a 'reCAPTCHA Privacy - Terms' link; and a 'Submit' button at the bottom. A large, semi-transparent 'GitChat' watermark is overlaid across the center of the form.

填写表单提交后，Infura 会为你提供专用的接口地址，然后我们只需要将网络地址填入到配置文件中，如下所示。

```
var HDWalletProvider = require("truffle-hdwallet-provider"); //
```

在这里我们需要通过 js 调用以太坊钱包，通过 `npm install truffle-hdwallet-provider` 安装这个库

```
var infura_apikey = "ubQWERwasd"; // infura 为你提供的 apikey 请与你申请到的 key 保持一致，此处仅为示例
```

var mnemonic = "apple banana carray dog egg fault great"; // 你以太坊钱包的 mnemonic，可以从 Metamask 当中导出，mnemonic 可以获取你钱包的所有访问权限，请妥善保存，在开发中切勿提交到 git

```
module.exports = {
  networks: {
    development: {
      host: "127.0.0.1",
      port: 7545,
      network_id: "*"
    },
    ropsten: {
      provider: function() {
        return new HDWalletProvider(mnemonic,
"https://ropsten.infura.io/"+infura_apikey)
      },
      network_id: 3,
      gas: 3012388,
      gasPrice: 30000000000
    },
    main: {
      provider: function() {
        return new HDWalletProvider(mnemonic,
"https://mainnet.infura.io/"+infura_apikey)
      },
      network_id: 3,
      gas: 3012388,
      gasPrice: 10000000000
    }
  }
};
```

在以太坊网络中发布合约需要使用 ETH 支付矿工的 gas 费用，你可以在 [Ethereum Ropsten Faucet](#) 免费获取到用于 Ropsten 测试网络的 ETH。

Ethereum Ropsten Faucet

Enter your testnet account address

Send me 3 test ether!


This faucet drips 3 Ether every 7 seconds. You can register your account in our queue. Max queue size is currently 5 .

Example command line: `wget http://faucet.ropsten.be:3001/donate/<your ethereum address>`

Example REST API: `http://faucet.ropsten.be:3001/donate/<your ethereum address>` [API docs](#)

Faucet queue

The queue is empty

This component is proudly brought to you by  [A-Labs](#)

由于网络环境的变化，不同的拥堵状况可能造成燃料费用和消耗的不同，如果发布不成功，可以调整 `gas/gasPrice` 的数值，你可以通过 `web3.getBlock('latest').gasLimit` 这一数值判断当前网络的消耗。

在命令行输入如下命令：

```
truffle migrate --network ropsten
```

通过 `--network` 设置发布的目标网络。

主网络发布

同理，在发布至主网络时，只需要执行如下命令。

```
truffle migrate --network main
```

但由于当前的以太坊网络的现实状况，如果设置燃料费太低，可能要等待数天后合约才会被网络确认，注意到我们编写的发布脚本是需要合约地址回调的，介于这种状况，我们可以将 `token` 合约和 `crowdsale` 合约分开发布，只需要再新建 `3-deploy-crowdsale.js` 文件，脚本如下。

```
const LeekCoinCrowdsale =  
artifacts.require("./GitCoinCrowdsale.sol")
```

```

module.exports = function(deployer, network, accounts) {
  const ifSuccessfulSendTo = accounts[0]
  const fundingGoalInEthers = 1000
  const durationInMinutes = 36000
  const etherCostOfEachToken = 0.01
  const tokenAddress = '0x123456789ABCDFGHSDWDVC' // 先单独发布
token 合约，上线成功后将其合约地址填在此处

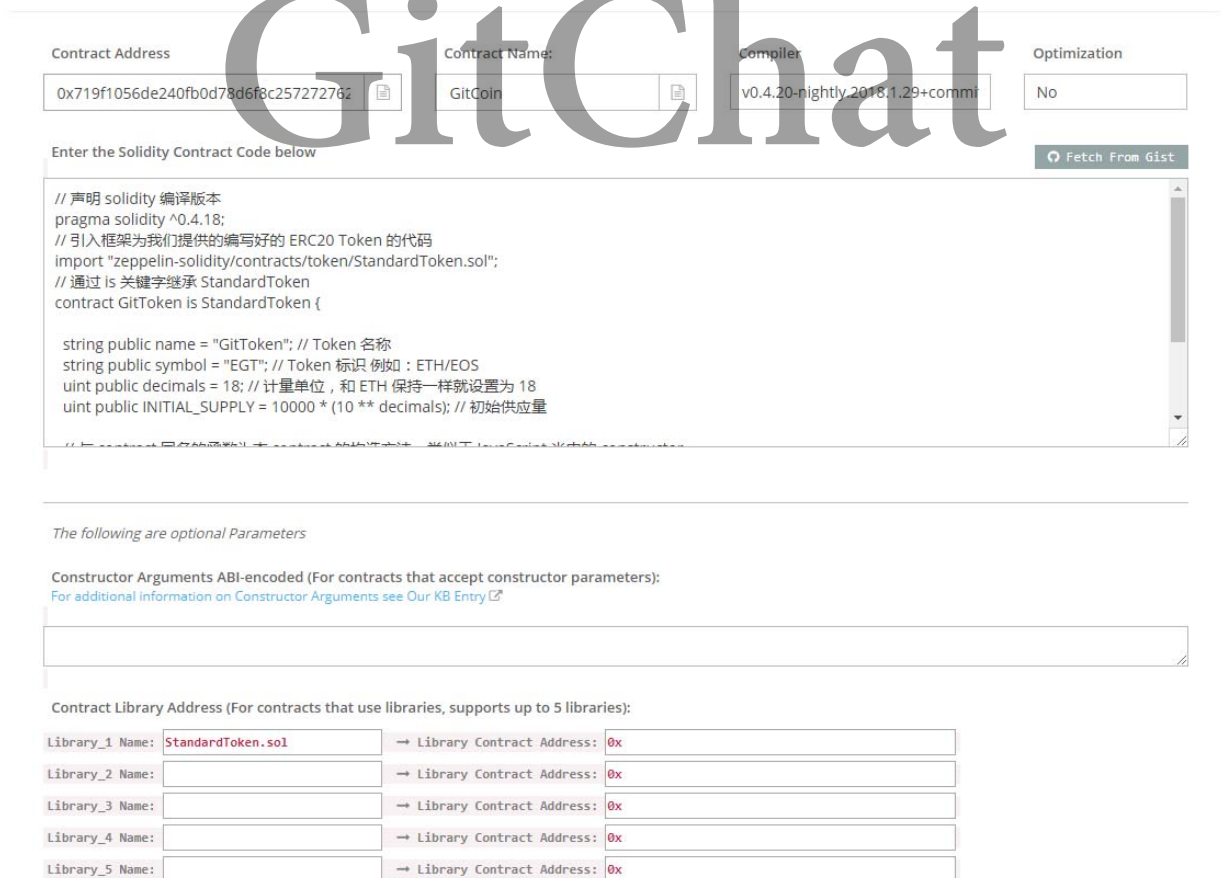
  deployer.deploy(GitCoinCrowdsale, ifSuccessfulSendTo,
fundingGoalInEthers, durationInMinutes, etherCostOfEachToken,
tokenAddress);
};

```

在发布至主网络时，可以分开两次进行，确保你设置的账户里有真实的 ETH 余额，注意设置好合理的 gas 数值，根据确认时间的长短，可能需要 0.08~1 ETH 不等，如果是壕请随意。

上线合约验证

无论是发布至以太坊的测试网络还是主网络，在发布完成之后都需要在 [Etherscan](#) 进行线上验证。在 Etherscan 上打开你刚刚发布的合约地址，你可以看到如下内容：



Contract Address: 0x719f1056de240fb0d78d6f8c257272762

Contract Name: GitCoin

Compiler: v0.4.20-nightly.2018.1.29+commit

Optimization: No

Enter the Solidity Contract Code below

```

// 声明 solidity 编译版本
pragma solidity ^0.4.18;
// 引入框架为我们提供的编写好的 ERC20 Token 的代码
import "zeppelin-solidity/contracts/token/StandardToken.sol";
// 通过 is 关键字继承 StandardToken
contract GitToken is StandardToken {

  string public name = "GitToken"; // Token 名称
  string public symbol = "EGT"; // Token 标识 例如：ETH/EOS
  uint public decimals = 18; // 计量单位，和 ETH 保持一致就设置为 18
  uint public INITIAL_SUPPLY = 10000 * (10 ** decimals); // 初始供应量
}

```

The following are optional Parameters

Constructor Arguments ABI-encoded (For contracts that accept constructor parameters):
[For additional information on Constructor Arguments see Our KB Entry](#)

Contract Library Address (For contracts that use libraries, supports up to 5 libraries):

Library_1 Name:	StandardToken.sol	→ Library Contract Address:	0x
Library_2 Name:		→ Library Contract Address:	0x
Library_3 Name:		→ Library Contract Address:	0x
Library_4 Name:		→ Library Contract Address:	0x
Library_5 Name:		→ Library Contract Address:	0x

点击 **Verify And Publish** 链接就可以进入验证页面：

[illegible]

1. Compiler 选择最新版本
2. Optimization 选择 No。

当然你也可以选择使用官方的 [Remix](#) 预先 concrete 你的合约文件，也可以安装 [solidity compiler](#) 在本地编译好再发布。ICO 和 token 的合约如此简单，根本不需要这些玩意儿，所以此处不再赘述，感兴趣的同学可以自行研究。

智能合约相当于我们的后端逻辑，以太坊的 EVM 就是我们的云服务器，Infura 为我们提供 API 接口，接下来我们就只需要给韭菜开发一个可以花钱消费的前端界面了。

1. 文字不要太多，页面要大片留白，简洁明了有现代感；
2. 配色一定要深，加上动态几何图形，设计要有未来感；
3. 开发团队全配齐，不是常春藤，没有硅谷背景的不要，一定要国际化；
4. 各种站台大佬，海量媒体报道，一线互联网公司合作全放上去。

言归正传，我们还是专注于技术。

web3.js 的使用

[web3.js](#) 为我们提供了一系列访问以太坊网络的 JavaScript 编程接口，完整的说明文档可以在 [web3.js Doc](#) 中参阅。我们一般通过如下脚本来初始化 web3 对象。

```
// 判断当前浏览器中有未注入 web3 对象
if (typeof web3 !== 'undefined') {
  App.web3Provider = web3.currentProvider;
  web3 = new Web3(web3.currentProvider);
} else {
  // 注意设置到你自己的 infura 地址
  App.web3Provider = new
Web3.providers.HttpProvider('https://ropsten.infura.io/ubQWERawsd
');
  web3 = new Web3(App.web3Provider);
}
```

Metamask 简介

[Metamask](#) 是一个浏览器插件，通过 Metamask 我们可以在浏览器中使用以太坊钱包，在访问 Dapp 应用时，也可以为其注入 web3 对象。具体配合应用开发的文档可以在 [MetaMask Compatibility Guide](#) 查阅，一般我们通过如下脚本来监测 Metamask 状态获取以太坊账户。

```
var account = web3.eth.accounts[0];
var accountInterval = setInterval(function() {
  if (web3.eth.accounts[0] !== account) {
    account = web3.eth.accounts[0];
    updateInterface();
  }
}, 100);
```

truffle-contract 的使用

web3.js 默认为我们提供的接口还是太底层，许多调用需要 hard code 设置参数，以太坊网络使用的 BigNumber 也需要我们手动转换，我们可以选择使用 [truffle-contract](#) 来调用更高一层的封装对象，并且在之前使用 truffle 开发构建的智能合约文件也能派上用场。

我们可以在 `build/contracts/` 下找到编译好的 `GitCoin.json` 和 `GitCoinCrowdsale.json` 文件，之后可以在我们的应用中通过如下脚本获取合约对象。


```

<script type="text/javascript" src="./dist/truffle-
contract.min.js"></script>
<script>
var GitCoin;
$.getJSON('contracts/GitCoin.json', function(data) {
  // 获取编译好的合约文件
  var GitCoinArtifact = data;
  // 通过 truffle-contract 获取合约对象
  GitCoin = TruffleContract(GitCoinArtifact);
  // 将合约绑定至当前 web3 对象
  GitCoin.setProvider(App.web3Provider);
});
</script>

```

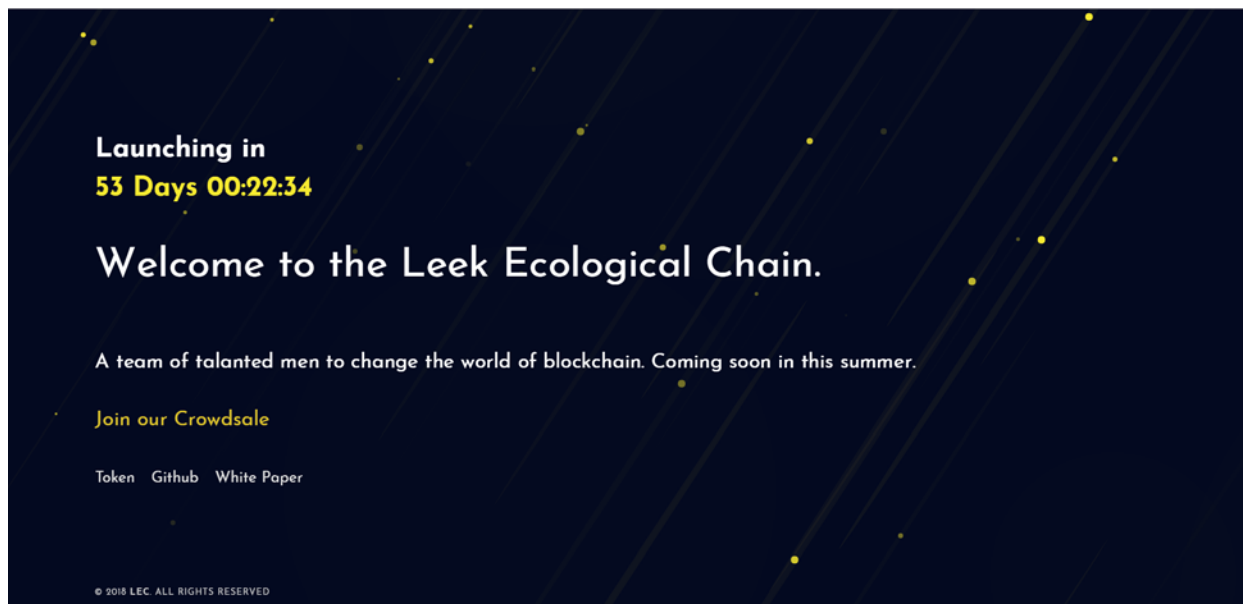
之后我们就可以像在 truffle console 当中一样，对合约对象进行各种操作啦。

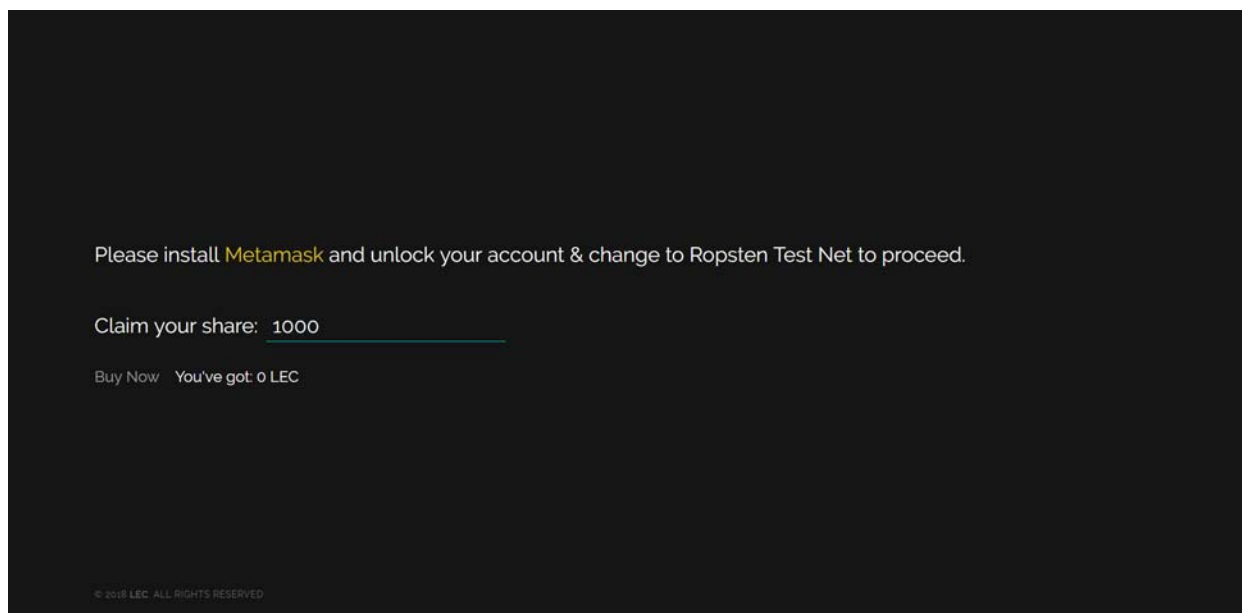
ICO 前端应用开发

我们的 ICO 应用只需要解决一个核心需求，那就是买币；只需要两个核心功能，一个是选择买多少，另一个就是付款，所以我们的界面自然是相当简单，如下图所示。



然后再稍微美化一下，如下面两张图所示。





一场成功的 ICO，自然需要精雕细琢，完整的代码示例可以在 [Leek Ecological Chain](#) 找到，同时此网站也是上述教程的一个完整示例，你可以切换到 Ropsten 网络在本网站上购买 [LEC 韭菜币](#)。

Dapp 部署

既然我们开发的是 Dapp 去中心化应用，怎么能够部署在中心化的服务器上呢？这不是自掉身价吗？Dapp 自然有其部署的解决方案。

IPFS 简介

[IPFS](#) 提供去中心化的点对点的 Web 服务，说简单点，你可以把它理解成为一个 p2p 的网盘，你网站的静态文件可以发布到 IPFS 上面托管，而且只要 IPFS 的节点不挂，你的网站就永远都不会挂，而不像部署到单独服务器上。同时 IPFS 上的一个文件也就对应着一个 hash 地址，普通用户可以通过公共的 http gateway 访问到你的页面，不像云服务器还要备案，正好也方便你割完韭菜跑路。

使用也非常简单，只需要在 [Install Go IPFS](#) 下载安装。

发布应用

只需要一行命令，把你 Dapp 的所有静态文件上传至 IPFS，命令如下。

```
ipfs add -r your-ico/  
# 返回 hash 地址，此处仅为示例  
added QWERabcd1234qwerABCD your-ico/
```

然后你就能够通过 <https://ipfs.io/ipfs/QWERabcd1234qwerABCD/> 访问你的网站。当然这样的域名十分不友好，为 IPFS 站点设置解析需要一些不常用的操作。

域名解析

IPNS

你的站点必然包含多个文件，每个文件对应着独立的 hash 地址，而且你也不能保证你的网站只需要发布一次，因此在网站发布后，我们需要使用 ipns 来获取到对应的唯一地址，之后的 DNS 解析也会对应到这一地址，同样只需要一行命令，如下所示。

```
# 站点发布后的 hash 地址，此处仅为示例
ipfs name publish QWERabcd1234qwerABCD
# 返回 ipns 地址
Published to ABCDqwer1234abcdQWER
```

之后你就能够通过 <https://ipfs.io/ipns/ABCDqwer1234abcdQWER> 访问你的站点了。在设置域名解析时，我们需要添加一条 **TXT** 类型的解析记录，解析值为：

```
dnslink=/ipns/ABCDqwer1234abcdQWER
```

这样我们就能够通过 <https://ipfs.io/ipns/yourico.com/> 访问你的 Dapp，这样是不是友好多了？

Nginx 反向代理

当然你也可能希望使用自己的独立域名，这时我们只需要使用 Nginx 设置反向代理即可。

```
server {
    listen 80;
    server_name yourico.com;

    location / {
        proxy_pass https://ipfs.io/ipns/yourico.com/;
    }
}
```

写在后面

以太坊官网，第一篇教程教你发 token，第二篇就教你卖 token，居心何在我也不好评判。

除了 ICO 还有 IMO/IFO，IMO 你只用买个路由器，IFO 只需要 fork 一份 Bitcoin 的代码，稍微调调参数，就不需要什么教程了。

程序员总是妄图通过技术手段解决社会问题，然而人性是不变的。以太坊希望建立一个 trustless 的网络，可惜被无数人滥用，巧立空气项目，搞空壳公司，逃避监管搞非法集资。

区块链和虚拟货币期望用点对点分布式的网络，脱离第三方，让世界上任何角落的两个人都能够低成本地进行交易，结果大量投机者涌入，导致网络堵塞，如今我们连一笔交易的矿工费都支付不起。

当然我信奉技术本身是无罪，就好像这篇教你割韭菜的文章一样，你是选择擦亮双眼，看清 ICO 的本质，从此势不两立；还是选择投机倒把，滥用以太坊技术，坠身同流合污？

Read at your own risk.

GitChat