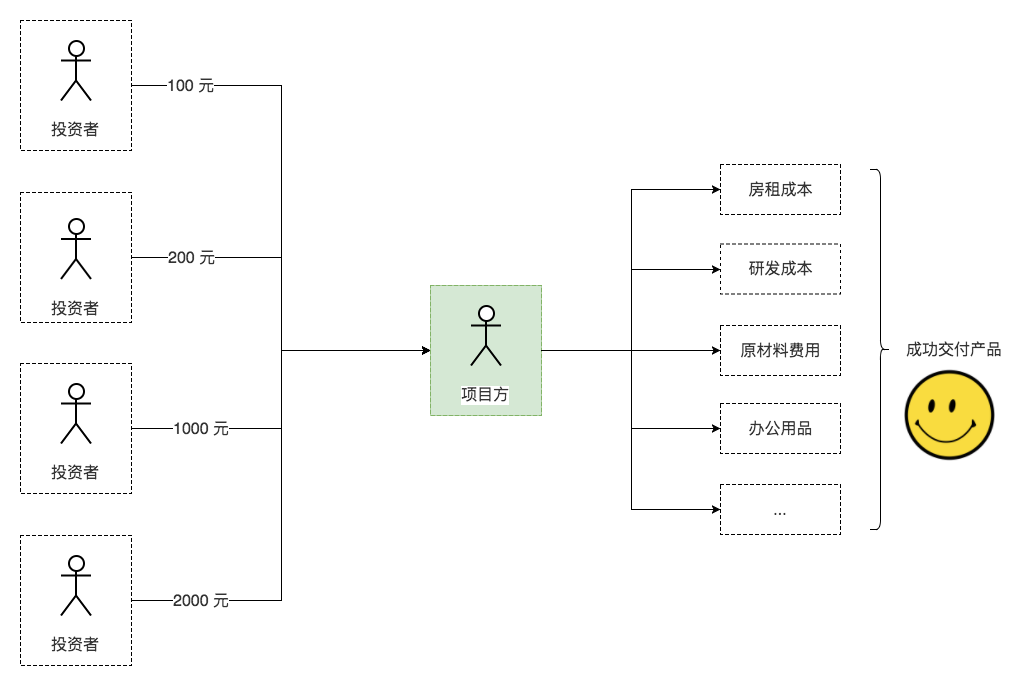
# 基于以太坊的众筹 (ICO) DApp

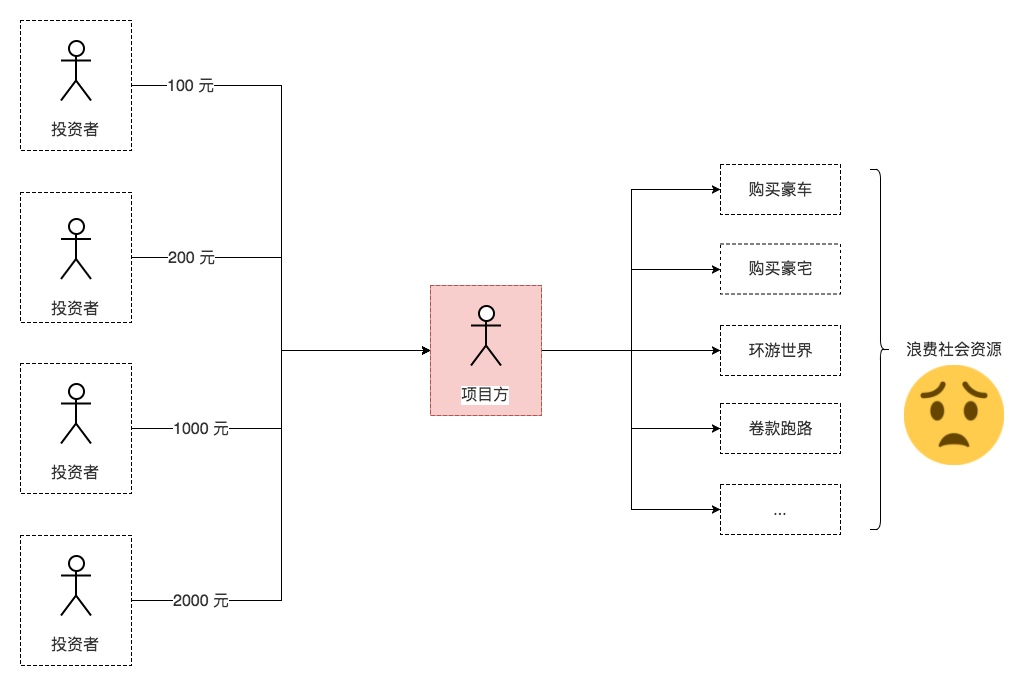
## 引言：ICO DApp 要解决什么问题？

随着科技的发展，创业团队募集资金的方式也发生了比较大的变化，比如kickstarter、众筹网等已经成熟的众筹模式，以及 2017 年盛行的 ICO 模式，都是项目发起者直接向公众募资，众筹和 ICO 都极大的提高了资金募集的效率和速度，从某种程度上讲，这种募资方式极大的提高了整个社会资金流转和利用的效率。

理想情况下，项目方或者创业团队从公众手里募集到资金之后，会踏实做事，在做事的过程中自然需要花钱，比如房租支出、人力成本、原材料等等费用，因为有了资金支持才能为成功交付产品提供基本的保障，理想情况下众筹或者 ICO 的过程可以用下图示意（其中忽略了众筹平台的存在，因为众筹平台大部分情况下只是起到信息发布的作用）：

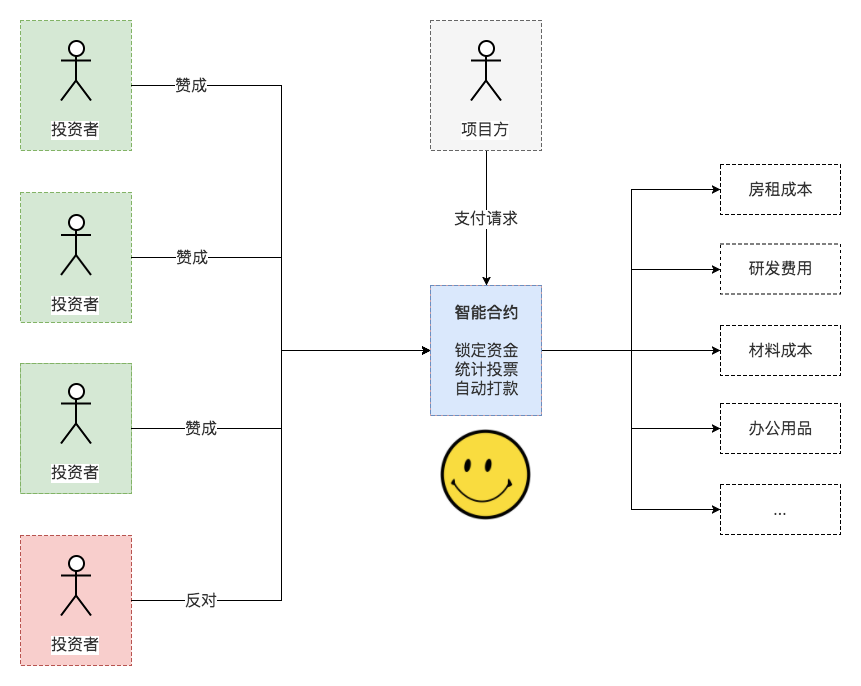


然而，现实世界通常跟我们的预期会有差距，恰恰因为募资方式的便利，众筹和 ICO 两种募资方式都存在严重的问题。尤其是基于以太坊智能合约可以方便的发行 ERC20 代币（全称 Initial Coin Offering，简称 ICO），极低的融资成本让这种模式被很多人给滥用了，和众筹类似，但比众筹更严重，ICO 过程是不受监管、完全中心化的，项目方对资金、募资方式有绝对的话语权，对募集来的资金（大多是 ETH、BTC）可以随意支配，而投资者拿到 ERC20 代币之后对于项目运作和资金使用没有任何约束力，因为资金使用的方式是不透明的，只能靠项目方团队的自我约束、及时披露。在团队不作为或纯走诈骗路线的情形下，整个事情很可能演变成下面这样：



众筹或者 ICO 跑路的案例并不少见，去谷歌或者百度上搜索 "ICO 诈骗"、"ICO 跑路"、"众筹诈骗"、"众筹跑路" 等关键词，看到的数据可谓触目惊心，投资者感觉被骗，资金也没有被用在正途上，久而久之，面对需要募集资金的项目，投资者可能都选择捂紧钱袋子，因为两种群体间的信任被打破了。

有什么办法能够缓解这种问题呢？我们或许可以借助区块链技术来重新思考这个事情，因为区块链本身是公开的、透明的、不可篡改的数据库技术，2017 年底以太坊创始人 Vatalik Buterin 提出了 DAICO 的理念，该理念可以用下图来示意：



即项目方募集到资金会被智能合约锁定在账户里面，智能合约本身是通过代码控制的，不能被任何人操纵，项目方在需要支出资金时先提出支出请求，然后让投资者投票，智能合约自动统计投票数量，如果超过半数的投资人同意该笔支出，智能合约会自动把资金转到对应的账户，这在很大程度上保障了投资者的利益，同时也鞭策项目方踏实做事。

我们接下来实战项目的灵感即来源于此，我们会先设计一个能够实现资金募集、资金锁定、资金支出请求管理、投资者投票统计、自动打款功能的智能合约，然后基于该智能合约开发一个最小可用的 DApp 应用，该应用具备众筹网站的大部分核心功能：

* 项目方能够在 DApp 中发起众筹；
* 项目方能够在 DApp 中管理众筹的支出请求；
* 投资者能够在 DApp 中参与众筹；
* 投资者能够对参与众筹的项目进行资金支出的投票；
* 众筹过程、资金支出的过程都是公开透明的；

需要大家注意的是，我们这里实现的只是简化的产品原型，虽然提到了 ICO，但我们即将要实现的智能合约和 DApp 中是不包含代币发行，也不会考虑投资者投资数量不同而导致的投票权重问题的。对实现完整发币、资金管理、支出管理智能合约和 DApp 感兴趣的同学，可以自行研究实现。

## ICO 智能合约的数据结构和接口设计

如果需要搞清楚任何一个信息系统，核心关注点只有两个，系统的数据结构和状态流转。数据结构通常可以理解为数据库表结构、系统中关键实体的属性，而状态流转常常和业务流程有关，即如何操作数据、数据如何变化，对应到实际的开发中就是系统对外暴露的接口，接口内部的业务规则。

具体到我们的众筹智能合约，该怎么设计呢？从两个角度分别来看。

### 1.1 数据结构

智能合约管理的基本数据单元是项目，指资金募集方可以创建、管理、投资者可以参与的项目，而项目需要具备基本属性，也需要存储项目生命周期中的各种数据。实战中会使用 Project 来命名项目智能合约。

具体来说，项目需要包含的基本属性：

* 所有者，即发起项目的人，在智能合约层面指调用项目创建合约的账户，数据类型是智能合约中独有的 address 类型，后续会详细介绍；
* 项目名称，项目的简单介绍，方便投资人投资前做检查，实际上如果要完整展示项目信息，需要的字段远不止这个，并且如果数据量很大，存储在以太坊上也不合适，我们在项目中做了简化处理；
* 资金余额，表示项目当前状态下的资金余额，如果项目没有任何支出，应该等于所有投资者投入的资金总和，数据类型为数值型，至于资金收支历史默认就存储在以太坊区块链上，不用我们关心；
* 最小投资金额，投资者投资项目的最小金额，比如项目需要的资金多，可以适当提高门槛，反之可以降低门槛；
* 最大投资金额，投资者投资项目的最大金额，比如项目方从分散风险的角度，避免单个投资人投入金额过大；
* 融资上限，即设定项目的融资目标，当投资者投资的累计金额达到这个目标，就不再接受新的投资；
* 投资人列表，指调用智能合约投资接口转账，参与投资的账户的集合，集合元素也是 address 类型，存储该集合的目的是为了在资金支出投票时做权限检查，也方便其他投资者了解项目基本状况；
* 资金支出列表，项目下所有的资金支出明细都存储在这里，列表中的每个条目会是个复杂的结构，会使用 Solidity 里面的 struct 来规范。

因为资金支出条目本身所包含的属性比较多，需要单独设计其中包含的字段：

* 资金用途，说明该笔资金支出的目的，字符串类型；
* 支出金额，标明资金支出的金额，数值类型；
* 收款方，该笔资金要转给谁，之所以要记录，是不想让该资金经项目方的手流转到收款方，减少操作空间；
* 状态，标明该笔支出是否已经完成；
* 投票记录，记录所有投资人在该笔支出上的投票记录，所有投票过的投资人都会被记录下来。

### 1.2 状态流转

状态流转会包含业务流程中数据变化的各种操作和业务规则。具体到我们的众筹智能合约，包含如下几个关键的业务流程：

* 创建项目，创建项目时需要指定项目名称，基本投资规则，自动记录项目的所有者；
* 参与众筹，参与的含义是投资人选定某个项目，并向智能合约转账，智能合约会把投资人记录在投资人列表中，并更新项目的资金余额；
* 创建资金支出条目，项目所有者有权限在项目上发起资金支出请求，需要提供资金用途、支出金额、收款方，默认为未完成状态，创建资金支出条目前需要检查资金余额是否充足；
* 给资金支出条目投票，投资人看到新的资金支出请求之后会选择投赞成票还是反对票，投票过程需要被如实记录，为了简化，我们只记录赞成票；
* 完成资金支出，项目所有者在资金支出请求达到超过半数投资人投赞成票的条件时才有权进行此操作，操作结果是直接把对应的金额转给收款方，转账前也要进行余额检查。

以上是我们从概念和需求角度对众筹智能合约数据结构和接口初步设计，里面关于数据类型、子结构的阐述还不够详细，后面章节会结合代码做更详细的介绍，而实际开发过程中，也可能因为如下原因对最初的设计做小幅调整：

* Solidity 语言本身的限制；
* 以太坊智能合约的限制；
* 性能、安全、部署等方面的考量；
* DApp 端展示需求，比如需要获取信息；

调整包括给项目增加属性、接口等，具体的变动及原因在遇到时会做详细的说明。

## ICO 智能合约开发

ICO合约初版

利用之前的工作流目录，在contracts下新建一个 Project.sol 文件：

pragma solidity ^0.4.17;

contract Project {

struct Payment {

string description;

uint amount;

address receiver;

bool completed;

address[] voters;

}

address public owner;

string public description;

uint public minInvest;

uint public maxInvest;

uint public goal;

address[] public investors;

Payment[] public payments;

constructor(string \_description, uint \_minInvest, uint \_maxInvest, uint \_goal) public {

owner = msg.sender;

description = \_description;

minInvest = \_minInvest;

maxInvest = \_maxInvest;

goal = \_goal;

}

function contribute() public payable {

require(msg.value >= minInvest);

require(msg.value <= maxInvest);

require(address(this).balance + msg.value <= goal);

investors.push(msg.sender);

}

function createPayment(string \_description, uint \_amount, address \_receiver) public {

Payment memory newPayment = Payment({

description: \_description,

amount: \_amount,

receiver: \_receiver,

completed: false,

voters: new address[](0)

});

payments.push(newPayment);

}

function approvePayment(uint index) public {

Payment storage payment = payments[index];

// must be investor to vote

bool isInvestor = false;

for (uint i = 0; i < investors.length; i++) {

isInvestor = (investors[i] == msg.sender);

if (isInvestor) {

break;

}

}

require(isInvestor);

// can not vote twice

bool hasVoted = false;

for (uint j = 0; j < payment.voters.length; j++) {

hasVoted = (payment.voters[j] == msg.sender);

if (hasVoted) {

break;

}

}

require(!hasVoted);

payment.voters.push(msg.sender);

}

function doPayment(uint index) public {

Payment storage payment = payments[index];

require(!payment.completed);

require(payment.voters.length > (investors.length / 2));

payment.receiver.transfer(payment.amount);

payment.completed = true;

}

}

合约中声明的属性包括：

* owner，项目所有者；
* description，项目介绍；
* minInvest，最小投资金额；
* maxInvest，最大投资金额；
* goal，融资上限；
* investors，投资人列表；
* payments，资金支出列表；

合约中声明的函数包括：

* constructor，合约构造函数，要求传入所有合约的基本属性；
* contribute，参与项目投资的接口，投资人调用该接口时要求发送满足条件的资金，并且要求没有达到募资上线，这是所有合约接口中标记为 payable 的接口，即接受用户在交易中发送 ETH；
* createPayment，发起资金支出请求，要求传入资金支出的细节信息；
* approvePayment，投票赞成某个资金支出请求，需要指定是哪条请求，要求投票的人是投资人，并且没有重复投票；
* doPayment，完成资金支出，需要指定是哪笔支出，即调用该接口给资金接收方转账，不能重复转账，并且赞成票数超过投资人数量的 50%

为了快速测试合约代码的正确性，我们可以先在remix中进行一下功能和流程测试。

## ICO合约重构：安全、费用、性能等的最佳实践

以太坊上智能合约的安全性一直是大家非常关注的话题，合约漏洞引发的安全事件我们并不陌生，从历史上著名的the DAO遭受黑客攻击，到今年的美图币爆出溢出漏洞，市值瞬间几乎归零；智能合约中的 BUG 如果在部署前没有被发现，产生的影响通常是灾难性的。

另一方面，因为以太坊上智能合约的绝大部分操作都是要支付gas的，而支付gas的多少跟计算量有关，所以能不能开发用户喜欢用的 DApp，需要在智能合约接口中做比较细节的优化，确保用户花最少的gas即可完成所有功能；另外，合约中的所有接口都是公开的，怎么确保接口不被恶意调用，也是部署合约前需要考虑的问题。

接下来，我们就从安全、性能、费用等方面对初版众筹智能合约进行改进。

### 3.1 安全改进

#### 3.1.1 防止数学运算溢出

计算机[数学运算溢出是很多 BUG 的根源](https://link.juejin.im/?target=https%3A%2F%2Fethereumdev.io%2Fsafemath-protect-overflows%2F)，在智能合约中，我们可以引入 SafeMath 机制来确保数学运算的安全，SafeMath 机制就是通过简单的检查确保常见的数学运算不出现预期之外的结果：

/\*\*

\* @title SafeMath

\* @dev Math operations with safety checks that throw on error

\*/

library SafeMath {

function mul(uint a, uint b) internal pure returns (uint) {

uint c = a \* b;

assert(a == 0 || c / a == b);

return c;

}

function div(uint a, uint b) internal pure returns (uint) {

// assert(b > 0); // Solidity automatically throws when dividing by 0

uint c = a / b;

// assert(a == b \* c + a % b); // There is no case in which this doesn't hold

return c;

}

function sub(uint a, uint b) internal pure returns (uint) {

assert(b <= a);

return a - b;

}

function add(uint a, uint b) internal pure returns (uint) {

uint c = a + b;

assert(c >= a);

return c;

}

}

众筹合约中检查是否已经达到募资上限时使用了加法运算，要使 SafeMath 机制生效还需要做如下改动：

contract Project {

**using SafeMath for uint;**

struct Payment {

string description;

uint amount;

...

}

...

function contribute() public payable {

require(msg.value >= minInvest);

require(msg.value <= maxInvest);

~~require(address(this).balance + msg.value <= goal);~~

**uint newBalance = 0;**

**newBalance = address(this).balance.add(msg.value);**

**require(newBalance <= goal);**

investors.push(msg.sender);

}

...

这里我们在数字上显式的调用了 uint 类型变量的 add 方法。

#### 3.1.2 操作权限检查

在众筹场景下，资金支出请求的创建、资金划转操作都应该限定仅项目所有者有权进行，而不是所有人都可以进行，要完成这样的功能，我们可以使用 Solidity 里面的 require 函数来做断言，不满足条件的时候交易直接回滚，改动如下：

function createPayment(string \_description, uint \_amount, address \_receiver) public {

**require(msg.sender == owner);**

...

}

function doPayment(uint index) public {

**require(msg.sender == owner);**

...

}

我们还可以用 modifier 来复用检查权限的代码：

**modifier ownerOnly(){**

**require(msg.sender == owner);**

**\_;**

**}**

function createPayment(string \_description, uint \_amount, address \_receiver) public **ownerOnly** { ...

function doPayment(uint index) public **ownerOnly** { ...

#### 3.1.3 账户余额检查

在做资金划转时，检查账户余额是非常有必要的，我们应在每次做转出操作前检查账户余额是否充足。在众筹合约中，只有 doPayment 接口内部有转账逻辑，转账前我们需要账户余额大于等于当前需要支出的金额，代码改动如下：

function doPayment(uint index) public {

...

require(!payment.completed);

**require(address(this).balance >= payment.amount);**

require(payment.voters.length > (investors.length / 2));

...

}

账户余额检查也可以在 createPayment 时做，如果当前账户余额不足以支付本次需要支出的金额，就抛出错误，当然最严密的做法是在 createPayment 和 doPayment 两处都做余额检查，此外，在后续要开发的 DApp 也做一层检查，避免用户发起无效的交易。

### 3.2 性能和费用

以太坊智能合约里面的所有计算都是要花费gas的，如果进行简单的操作都要花很多钱，显然没有人愿意用这样的Dapp。我们在remix中进行测试时可以注意到，初版众筹智能合约编译时编译器报了很多 warning，这些warning主要都是针对性能方面的。

可以看到，编译器认为合约中的 contribute、createPayment、approvePayment、doPayment 的gas消耗都是很高的，极端情况下接近无限。通过简单的分析，我们不难发现，其中最严重的是 approvePayment，里面有两个循环，如果项目非常火爆，投资人数以万计，每次投票需要的计算量就非常大了。

有没有办法把线性的访问时间优化到到常数访问时间呢？熟悉数据结构的同学心里已经有了答案，我们可以使用哈希结构来存储这些大列表，而 Solidity 里面为我们提供了类哈希结构 mapping，我们在之前的项目中也用到过多次了，不过 Solidity 中的 mapping 类型有几个重要的特征需要再次强调一下，因为乍看起来 mapping 和 Javascript 中的对象很像，但是区别挺大。

* 一个mappings 要求所有的 key 和 value 都必须是完全相同的类型，Javascript 却没有这种要求；
* mappings 里面并没有存储所有的 key，因此无法获取所有的 key 列表，Javascript 中可使用 Object.keys 取所有 key；
* mappings 里面的值也是无法被遍历的，只能通过 key 逐个去取，Javascript 中可用 Object.values 取所有值。

接下来我们对代码的数据结构进行改进，用 mapping 来存储 investors 和 voters，而 payments 还保持是数组类型：

contract Project {

struct Payment {

...

bool completed;

~~address[] voters;~~

**mapping(address => bool) voters;**

**uint voterCount;**

}

...

~~address[] public investors;~~

**mapping(address => uint) public investors;**

**uint public investorCount;**

Payment[] public payments;

...

function contribute() public payable {

...

require(newBalance <= goal);

~~investors.push(msg.sender);~~

**investors[msg.sender] = msg.value;**

**investorCount += 1;**

}

function createPayment(string \_description, uint \_amount, address \_receiver) public {

...

completed: false,

~~voters: new address[](0)~~

**voterCount: 0**

});

payments.push(newPayment);

}

function approvePayment(uint index) public {

...

// must be investor to vote

~~bool isInvestor = false;~~

~~for (uint i = 0; i < investors.length; i++) {~~

~~isInvestor = (investors[i] == msg.sender);~~

~~if (isInvestor) {~~

~~break;~~

~~}~~

~~}~~

~~require(isInvestor);~~

**require(investors[msg.sender] > 0);**

// can not vote twice

~~bool hasVoted = false;~~

~~for (uint j = 0; j < payment.voters.length; j++) {~~

~~hasVoted = (payment.voters[j] == msg.sender);~~

~~if (hasVoted) {~~

~~break;~~

~~}~~

~~}~~

~~require(!hasVoted);~~

**require(!payment.voters[msg.sender]);**

~~payment.voters.push(msg.sender);~~

**payment.voters[msg.sender] = true;**

**payment.voterCount += 1;**

}

function doPayment(uint index) public {

...

require(address(this).balance >= payment.amount);

~~require(payment.voters.length > (investors.length / 2));~~

**require(payment.voterCount > (investorCount / 2));**

...

}

}

改动之后，每次投票所需要的计算量大大减少，自然能为用户节省费用。需要额外说明的是，因为 mapping 类型没有存储里面键值的个数，我们需要在 Project 合约上新增 investorCount 来记录投资人数，在 Payment 结构体上新增 voterCount 来记录投票人数，最后资金划转时需要用这两个计数器来确定投赞成票的人数是否达到过半的要求，这些在代码中都有体现。

代码重构完成后，同样将代码放到remix中，校验正确性和流程的完整性。

### 3.3 部署方面的考量

如果在传统的 WEB 应用中设计众筹系统的数据库，所有的项目都会存在一张表中，而表中的每条记录都是一个实际的项目，在智能合约场景下，我们已经具备创建单个项目的能力，Project 合约部署完就能得到项目的实例，但实际的众筹应用中肯定不止一个项目，怎么存储所有的项目呢？因为在 DApp 我们必须具备列出所有项目的能力，能不能把所有的数据都存储在以太坊区块链上面？答案是肯定的。

接下来我们介绍一种比较常见的智能合约部署方法：用合约来控制合约。我们将会在代码中新建一个合约，用来管理所有的Project合约实例。新增合约 ProjectList.sol：

contract ProjectList {

using SafeMath for uint;

address[] public projects;

function createProject(string \_description, uint \_minInvest, uint \_maxInvest, uint \_goal) public {

address newProject = new Project(msg.sender, \_description, \_minInvest, \_maxInvest, \_goal);

projects.push(newProject);

}

function getProjects() public view returns(address[]) {

return projects;

}

}

更改 Project 的构造函数，传入项目的 owner：

~~constructor(string \_description, uint \_minInvest, uint \_maxInvest, uint \_goal) public {~~

**constructor(address \_owner, string \_description, uint \_minInvest, uint \_maxInvest, uint \_goal) public {**

~~owner = msg.sender;~~

**owner = \_owner;**

...

}

新增合约的名称叫做 ProjectList，顾名思义就是项目列表，其中仅包含 createProject 和 getProjects 两个接口，方便我们创建新的项目、拉取所有的项目列表，其中 createProject 因为是从外部进行的，需要给 Project 合约的构造函数新增 \_owner 参数。

把最新的代码放到 Remix 上，部署 ProjectList，并且调用它的 createProject 接口，然后加载创建后的 Project 合约实例进行测试。

## 众筹智能合约的编译、部署和自动化测试

现在我们有了两个智能合约源文件，在不使用 truffle 这类框架的前提下，怎样进行编译、部署和测试的管理呢？回忆一下我们之前做过的自动化编译部署脚本，现在可以重新利用起来。不过之前我们的脚本只是针对一个合约源文件的，现在需要稍作改进。

### 4.1 改进编译脚本

对workflow 项目下的 scripts/compile.js 做如下改进，使得我们可以编译 contracts 目录下的所有合约源文件：

compile.js

const fs = require('fs-extra');

const path = require('path');

const solc = require('solc');

// 1. cleanup

const compiledDir = path.resolve(\_\_dirname, '../compiled');

fs.removeSync(compiledDir);

fs.ensureDirSync(compiledDir);

~~// compile~~

~~const contractPath = path.resolve(\_\_dirname, '../contracts', 'Car.sol');~~

~~const contractSource = fs.readFileSync(contractPath, 'utf8');~~

~~const result = solc.compile(contractSource, 1);~~

**// 2. search all contracts**

**const contractFiles = fs.readdirSync(path.resolve(\_\_dirname, '../contracts'));**

**contractFiles.forEach(contractFile => {**

// 2.1 compile

const contractPath = path.resolve(\_\_dirname, '../contracts', contractFile);

const contractSource = fs.readFileSync(contractPath, 'utf8');

const result = solc.compile(contractSource, 1);

console.log(`file compiled: ${contractFile}`);

// 2.2 check errors

if (Array.isArray(result.errors) && result.errors.length) {

throw new Error(result.errors[0]);

}

// 2.3 save to disk

Object.keys(result.contracts).forEach(name => {

const contractName = name.replace(/^:/, '');

const filePath = path.resolve(compiledDir, `${contractName}.json`);

fs.outputJsonSync(filePath, result.contracts[name]);

console.log(` > contract ${contractName} saved to ${filePath}`);

});

**});**

上面的改动只是用到了 fs-extra 模块中的 readdirSync 方法，把 contracts 目录下所有合约文件读出来逐个编译，还对日志打印也做了改进。在项目根目录下执行 npm run compile，可以看到 Project.sol 文件编译出来3个 .json 文件，其中 ProjectList.json 和 Project.json 是可以单独部署的。

### 4.2 改进部署脚本

我们还需要改进下部署脚本，使其默认部署 ProjectList 合约，并且能把合约地址记录下来，方便后续使用，记录合约部署地址的方式可以是写到文件中，具体改动如下：

deploy.js

**const fs = require(‘fs-extra’);**

...

// 1. get bytecode

~~const contractPath = path.resolve(\_\_dirname, '../compiled/Car.json');~~

const contractPath = path.resolve(\_\_dirname, '../compiled/ProjectList.json');

...

(async() =>{

...

// 3. get contract instance and deploy

...

const result = await new web3.eth.Contract(JSON.parse(interface))

~~.deploy({ data: bytecode, arguments: ['AUDI'] })~~

**.deploy({ data: bytecode })**

...

**const constractAddress = result.options.address;**

**// 4. write contract address to file**

**const addressFile = path.resolve(\_\_dirname, '../address.json');**

**fs.writeFileSync(addressFile, JSON.stringify(contractAddress));**

**console.log('地址写入成功:', addressFile);**

**process.exit();**

})();

### 4.3 增加自动化测试

合约编译完之后，我们就可以基于bytecode、interface 来添加单元测试了，同样还是利用了mocha 框架，代码如下：

const assert = require('assert');

const path = require('path');

const ganache = require('ganache-cli');

const Web3 = require('web3');

const web3 = new Web3(ganache.provider());

const ProjectList = require(path.resolve(\_\_dirname, '../compiled/ProjectList.json'));

const Project = require(path.resolve(\_\_dirname, '../compiled/Project.json'));

let accounts;

let projectList;

let project;

describe('Project Contract', () => {

// 1. 每次跑单测时需要部署全新的合约实例，起到隔离的作用

beforeEach(async () => {

// 1.1 拿到 ganache 本地测试网络的账户

accounts = await web3.eth.getAccounts();

// 1.2 部署 ProjectList 合约

projectList = await new web3.eth.Contract(JSON.parse(ProjectList.interface))

.deploy({ data: ProjectList.bytecode })

.send({ from: accounts[0], gas: '5000000' });

// 1.3 调用 ProjectList 的 createProject 方法

await projectList.methods.createProject('Ethereum DApp Tutorial', 100, 10000, 1000000).send({

from: accounts[0],

gas: '1000000',

});

// 1.4 获取刚创建的 Project 实例的地址

const [address] = await projectList.methods.getProjects().call();

// 1.5 生成可用的 Project 合约对象

project = await new web3.eth.Contract(JSON.parse(Project.interface), address);

});

it('should deploy ProjectList and Project', async () => {

assert.ok(projectList.options.address);

assert.ok(project.options.address);

});

it('should save correct project properties', async () => {

const owner = await project.methods.owner().call();

const description = await project.methods.description().call();

const minInvest = await project.methods.minInvest().call();

const maxInvest = await project.methods.maxInvest().call();

const goal = await project.methods.goal().call();

assert.equal(owner, accounts[0]);

assert.equal(description, 'Ethereum DApp Tutorial');

assert.equal(minInvest, 100);

assert.equal(maxInvest, 10000);

assert.equal(goal, 1000000);

});

});

同样，我们每个测试运行都是在单独的合约实例下，确保环境隔离，环境准备过程的 1.3、1.4、1.5 三个步骤，我们调用 ProjectList 的方法直接创建 Project 合约实例，后续的测试基本都是针对 Project 合约实例编写，因为主要的业务逻辑都在 Project 合约中。

具备了基本测试骨架之后，可以逐步提高测试用例的覆盖度，比如先测试 Project 合约实例的 contribute 方法：

it('should allow investor to contribute', async () => {

const investor = accounts[1];

await project.methods.contribute().send({

from: investor,

value: '200',

});

const amount = await project.methods.investors(investor).call();

assert.ok(amount == '200');

});

又如，测试 contribute 接口的边界检查，即最小投资金额、最大投资金额，以及投资上限也是可以测试的，感兴趣的可以自己添加：

it('should require minInvest', async () => {

try {

const investor = accounts[1];

await project.methods.contribute().send({

from: investor,

value: '10',

});

assert.ok(false);

} catch (err) {

assert.ok(err);

}

});

这里的接口边界测试时使用了比较 trick 的方法，因为 contribute 接口在边界检查失败时会抛出错误，这样接口调用就会返回异常，所以我们使用了 try catch 把代码包起来，断言在超出边界时肯定会抛出错误。

## 众筹DApp的功能设计和原型图

在开始开发众筹 DApp 之前，我们有必要仔细思考下它的主要用户，要实现的关键功能，以及初步的界面设计。

### 5.1功能设计

任何面向用户的应用所有的功能设计都应该围绕着用户去做，对众筹 DApp 来说，用户都有谁？用户都会在这个应用里面做什么？搞清楚这些问题的答案，我们就能列出众筹 DApp 需要实现的核心功能。

* 众筹 DApp 的用户都有谁？最核心的用户就是项目方和投资人；
* 项目方会在众筹 DApp 上做什么？发起项目、请求支出资金、操作资金划转；
* 投资人会在众筹 DApp 上做什么？浏览项目、投资项目、参与项目资金支出的投票

首先，我们可以按照开发经典 WEB 应用的思路来想，如果系统中有用户，是不是需要用户系统？答案是不需要，因为我们架构在以太坊之上，直接让用户使用以太坊账户就行了。这样就要求使用我们系统的用户在浏览器中安装了 Metamask 插件，Metamask 中保存了他们的账户。

经典 WEB 应用的信息架构通常是：列表 --> 详情 --> 交互的模式，这种模式在众筹 DApp 也有很直观的体现，众筹 DApp 中存在两处这种模式的实例：

* 项目列表 --> 项目详情 --> 参与项目投资或直接返回，表现层就对应两个页面；
* 项目详情 --> 资金支出列表 --> 资金支出详情 --> 投票、资金划转，表现层可以把资金支出列表和资金支出详情以表格的形式放在项目详情中，方便用户预览信息。

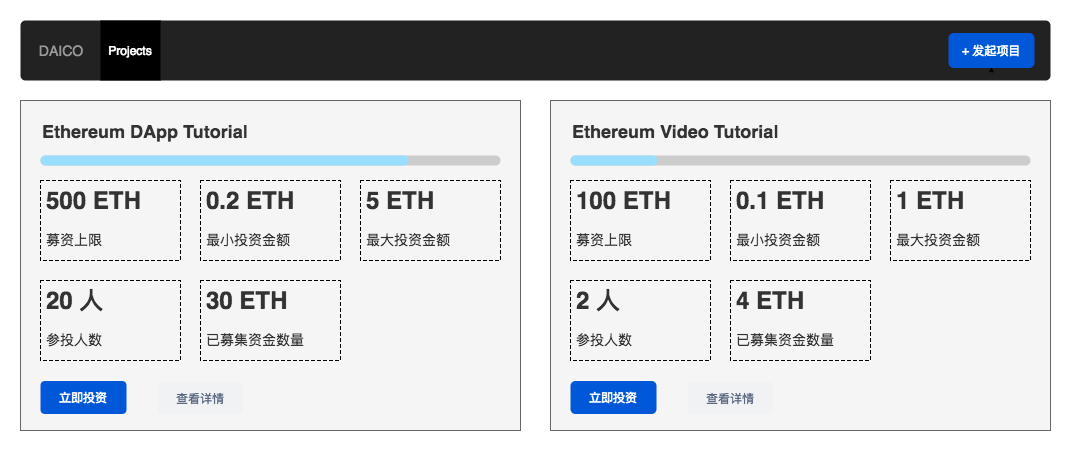
至于 DApp 中所呈现信息的来源，我们需要 3 个表单方便项目方和投资人填写和提交：

* 项目创建表单，项目方使用，创建新的众筹项目实例，新实例会追加到项目列表中；
* 资金支出请求创建表单，项目方使用，用于往项目中追加资金支出请求；
* 项目投资表单，投资人使用，用于往项目合约账户转账。

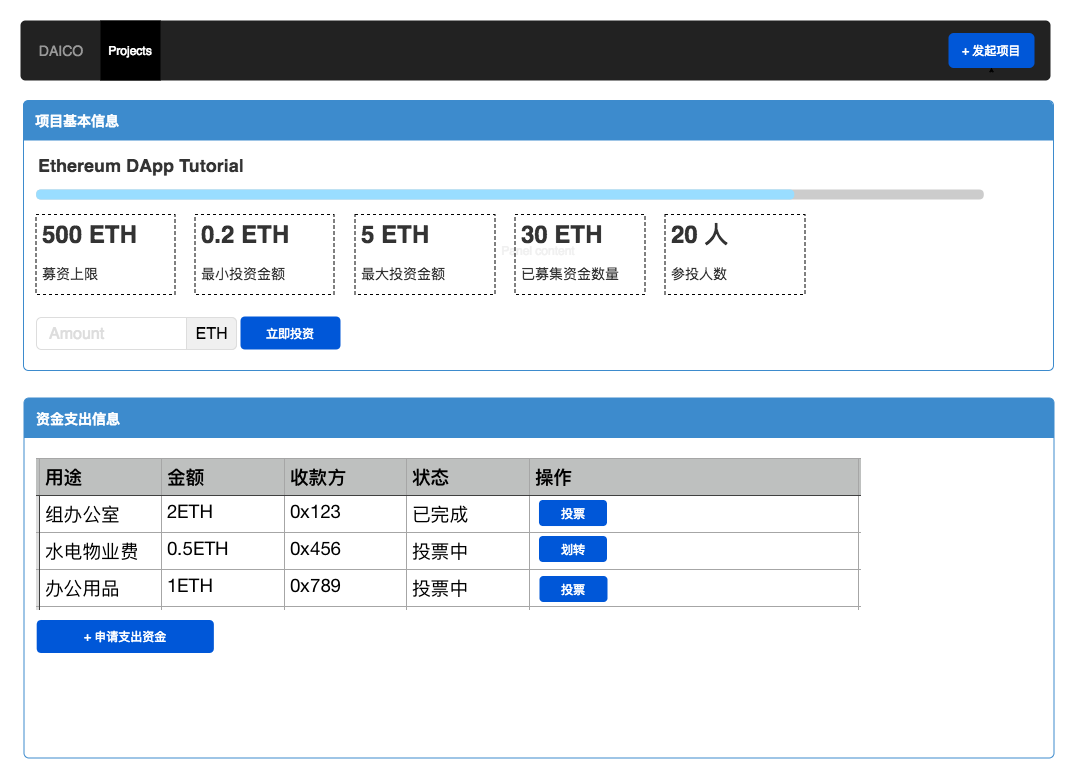
### 5.2 原型图

搞清楚核心用户、关键功能、信息架构之后，我们不难画出众筹 DApp 的原型图。原型图只是表征 DApp 开发之后大概的模样，实际开发出来的效果跟原型图可能会有出入。

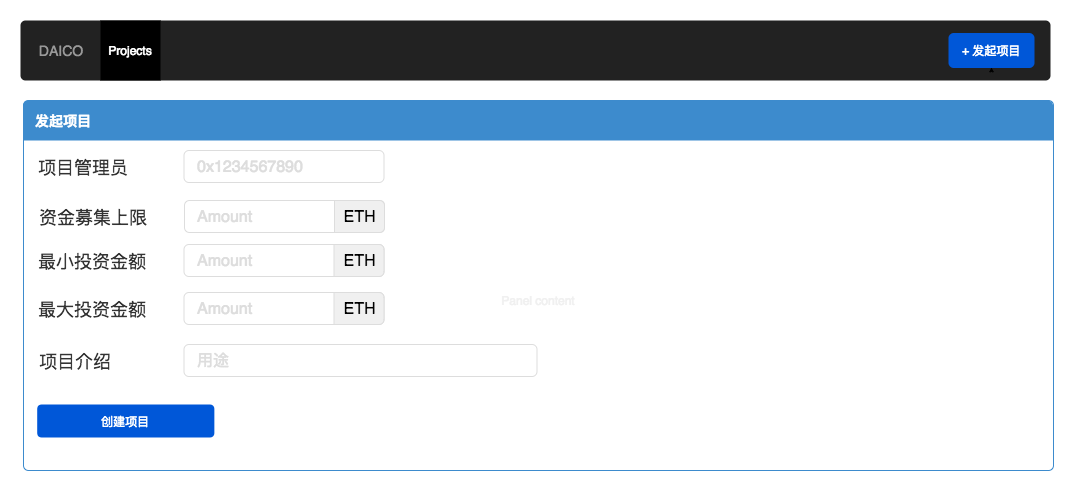
首先是 DApp 首页，以卡片的形式展示所有项目的列表，为了简化，列表页暂不考虑分页：



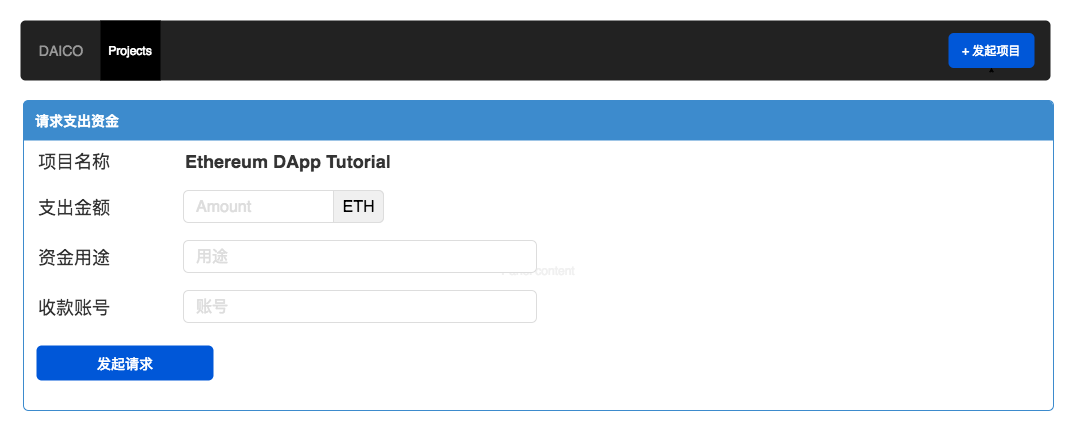
然后是项目详情页，展示项目详情、投资表单、项目资金支出历史：



接下来是项目创建表单：



最后是资金支出请求创建表单：



不同页面之间的跳转关系完全按照以上信息架构，以及原型图上各种按钮的定义。

## 众筹DApp框架搭建

### 6.1 技术选型

本着尽可能简单的原则，根据开发传统 WEB 应用的思路去选择 DApp 的技术栈，拟定的技术栈为：

* React，一个构建用户界面的JavaScript库，主要用于构建UI，负责视图层和简单的状态管理；
* Next.js，一个轻量级的 React 服务端渲染应用框架。负责后端请求的处理，把 React 生态里面的各种工具帮开发者拼起来，极大的降低了 React SSR 的上手门槛，同时使用 next-routes 来实现用户友好的 URL；
* Material UI，负责提供开箱即用的样式组件，相比 Semantic UI、Elemental、Element React 等更新更活跃，接触和熟悉的开发者群体更大

### 6.2 路由规划

对于 WEB 应用来说，可以认为路由即功能，因为设计良好的路由能够把应用的信息结构直观的展示出来，具体到众筹 DApp，需要支持的路由如下表：

|  |  |
| --- | --- |
| URL模式 | 路由功能 |
| / | 首页，即项目列表页 |
| /project/create | 项目创建页 |
| /project/0x123456 | 项目详情页 |
| /project/0x123456/payments/create | 创建资金支出请求的页面 |

其中的 0x123456 指每个项目的合约地址，因为众筹 DApp 中的每个项目都会有自己的 Project 合约实例，而所有项目的合约地址存储在 ProjectList 合约的 projects 属性中。

### 6.3 安装Next.js和React

npm install --save next react react-dom

修改package.json，添加用于启动项目和构建项目的npm script：

{

"scripts": {

...

"dev": "next",

"build": "next build",

"start": "next start"

},

...

}

### 6.4 创建项目首页

在项目根目录下创建 pages 目录，并添加 pages/index.js，在其中输入如下内容：

import React from 'react';

export default class Index extends React.Component {

render() {

return <div>Welcome to Ethereum ICO DApp!</div>;

}

}

在控制台中，切换到项目根目录下，执行如下命令启动服务：

npm run dev

使用浏览器打开 <http://localhost:3000/>，我们应该能看到正常的页面渲染。

### 6.5 集成next-routes

在next.js 默认路由机制中，项目详情页的路由会是这样：

/projects?address=0x123456

这也是我们已经熟悉的URI路径+查询参数的写法。显然这种表达可读性并不太好，我们可以将它改进为如下路由：

/projects/0x123456

使用 next-routes 能够很方便的实现用户友好的路由，步骤如下：

1. 首先安装依赖

npm install --save next-routes

1. 根目录下增加 routes.js，内容如下：

const routes = require('next-routes')();

routes

.add('/projects/create', 'projects/create')

.add('/projects/:address', 'projects/detail')

.add('/projects/:address/payments/create', 'projects/payments/create');

module.exports = routes;

1. 根目录下增加 server.js，内容如下：

const next = require('next');

const http = require('http');

const routes = require('./routes');

const app = next({ dev: process.env.NODE\_ENV !== 'production' });

const handler = routes.getRequestHandler(app);

app.prepare().then(() => {

http.createServer(handler).listen(3000, () => {

console.log('server started on port: 3000');

});

};

1. 修改 package.json 的服务启动命令，因为我们自定义了 server 文件，不再使用 next.js 内置的 server：

{

"scripts": {

...

~~"dev": "next",~~

"dev": "node server.js",

"build": "next build",

~~"start": "next start",~~

"start": "NODE\_ENV=production node server.js"

},

...

}

1. 重启 next.js 服务。结束之前的服务器进程，重新运行 npm run dev，刷新浏览器，确保能正常打开页面。

众筹 DApp 开发的脚手架已经就绪，接下来我们就将进入实际的功能实现阶段。

## 7. 众筹DApp基本页面布局

页面布局按照前面给出的 DApp 原型图去做就可以了，实现页面布局的先决条件是集成组件库 Material UI，然后基于这个组件库去创建全站通用的 Layout、Header 组件。

### 7.1 集成Material UI

1. 安装依赖

npm install @material-ui/core

2. 导入支持服务端渲染的必要文件

官方示例项目中有两个支持服务端渲染的必要文件：src/withRoot.js、src/getPageContext.js 需要复制到我们的项目中。在项目跟目录下新建 libs 目录，然后将这两个文件原封不动复制过来即可，两个文件的作用描述如下：

* withRoot.js 实质上是 React 组件，并且是高阶组件，作用在于确保服务端渲染时能正确的生成样式；
* getPageContext.js 为整个应用的前端、后端渲染提供 theme 配置，以及必要的工具函数，比如如何生成类名等

3. 创建自定义的页面结构

参照示例项目中的 pages/\_document.js，是基于 Next.js 里面的 [Custom Document](https://link.juejin.im/?target=https%3A%2F%2Fgithub.com%2Fzeit%2Fnext.js%2F%23custom-document) 机制实现的，创建自定义页面结构的动机在于我们可以全局性的在页面中引入需要的样式和字体文件。

4. 使用Material UI中的组件

接下来检验 Material UI 的集成是否成功，改动pages/index.js中的内容，其中调用了组件库中的 Button 组件：

import React from 'react';

import { Button } from '@material-ui/core';

import withRoot from '../libs/withRoot';

class Index extends React.Component {

render() {

return <Button variant="raised" color="primary">Welcome to Ethereum ICO DApp!</Button>;

}

}

export default withRoot(Index);

5. 重新启动服务

执行 npm run dev，刷新页面，应该看到添加了Button组件的首页。

### 7.2 页面布局组件

Material UI 集成之后，接下来就可以直接使用其中的各种组件搭建自己的界面，比如Layout、Header，因为这些代码是共享的，都存放在根目录下新建的 components 目录中。

#### 7.2.1 Layout组件

1. 新建文件 components/Layout.js

Layout 组件的目的主要是规范页面的宽度，源代码如下：

import React from 'react';

import { withStyles } from '@material-ui/core/styles';

const styles = {

container: {

display: 'flex',

flexDirection: 'column',

width: '100%',

minHeight: '100vh',

},

wrapper: {

margin: '0 auto',

width: '80%',

maxWidth: '1200px',

marginTop: '1em',

},

};

class Layout extends React.Component {

render() {

const { classes } = this.props;

return (

<div className={classes.container}>

<div className={classes.wrapper}>{this.props.children}</div>

</div>

);

}

}

export default withStyles(styles)(Layout);

这里Layout 组件中使用了 Material UI 里面的样式注入机制。

2. 修改 pages/index.js 应用新的 Layout

import React from 'react';

import { Button } from '@material-ui/core';

import withRoot from '../libs/withRoot';

**import Layout from '../components/Layout';**

class Index extends React.Component {

render() {

return (

**<Layout>**

<Button variant="raised" color="primary">

Welcome to Ethereum ICO DApp!

</Button>

**</Layout>**

}

}

export default withRoot(Index);

#### 7.2.2 Header组件

1.创建Header组件

在 components 目录下新建 Header.js，输入如下代码：

import React from 'react';

import { AppBar, Toolbar, Typography, Button } from '@material-ui/core';

import { withStyles } from '@material-ui/core/styles';

const styles = {

wrapper: {

margin: '0 auto',

width: '80%',

maxWidth: '1200px',

display: 'flex',

},

brand: {

borderRight: '2px solid #CCCCCC',

paddingRight: '1em',

marginRight: '1em',

},

toolbar: {

padding: 0,

flex: 1,

},

flexContainer: {

flex: 1,

},

anchor: {

textDecoration: 'none',

},

};

class Header extends React.Component {

render() {

const { classes } = this.props;

return (

<AppBar position="static" color="default">

<div className={classes.wrapper}>

<Toolbar className={classes.toolbar}>

<Typography variant="title" color="inherit" className={classes.brand}>

众筹 DApp

</Typography>

<p className={classes.flexContainer}>

<a href="/" className={classes.anchor}>

<Typography variant="title" color="inherit">

项目列表

</Typography>

</a>

</p>

<Button variant="raised" color="primary">

发起项目

</Button>

</Toolbar>

</div>

</AppBar>

);

}

}

export default withStyles(styles)(Header);

Header 中使用到了 Material UI 中的 AppBar、ToolBar、Button 组件，其中 AppBar 是和浏览器窗口等宽的，而 Toobar 和 Layout 里面的内容宽度相同。

2. 将 Header 组件集成到 Layout

因为 Header 组件也是全局通用的，在 Layout 中引入即可，修改 components/Layout.js 如下：

import React from 'react';

import { withStyles } from '@material-ui/core/styles';

**import Header from './Header';**

...

class Layout extends React.Component {

render() {

const { classes } = this.props;

return (

<div className={classes.container}>

**<Header />**

<div className={classes.wrapper}>{this.props.children}</div>

</div>

);

}

}

...

截至目前，我们的项目仓库代码结构如下：

.

|── components

|   |── Header.js

|   └── Layout.js

|── libs

|   |── getPageContext.js

|   └── withRoot.js

|── package.json

|── pages

|   |── \_document.js

|   └── index.js

|── routes.js

|── server.js

## 8. 构建前后端通用的Web3实例

### 8.1 在DApp中实例化Web3

Metamask 会在浏览器的 Javascript 中注入全局的 Provider，提供给Web3对象使用，所以在安装了 Metamask 的情况下，我们在浏览器环境下可以直接这样初始化 Web3 实例：

import Web3 from 'web3';

const web3 = new Web3(window.web3.currentProvider);

const accounts = web3.eth.getAccounts();

### 8.2 在DApp中调用链上数据渲染页面

为了测试 Metamask 提供的 Web3 Provider，我们可以直接修改 index.js，并在其中渲染当前Metamask钱包里面的账户地址：

import React from 'react';

**import Web3 from 'web3';**

import { Button } from '@material-ui/core';

import withRoot from '../libs/withRoot';

import Layout from '../components/Layout';

class Index extends React.Component {

constructor(props){

super(props);

this.state = {

accounts: []

};

}

async componentDidMount(){

const web3 = new Web3(window.web3.currentProvider);

const accounts = await web3.eth.getAccounts();

console.log(accounts);

this.setState({ accounts });

}

render() {

**const { accounts } = this.state;**

return (

<Layout>

~~<Button variant="raised" color="primary">~~

~~Welcome to Ethereum ICO DApp!~~

~~</Button>~~

<ul>{accounts.map(x => <li key={x}>{x}</li>)}</ul>

</Layout>

}

}

export default withRoot(Index);

我们在Index组件上新增了accounts状态变量，然后在componentDidMount 函数中实例化Web3实例，获取到所有的账户，然后更新state。

数组的map() 方法返回一个新数组，其中的元素为原始数组元素调用函数处理后的值。

如果想渲染所有钱包账户的余额，则可以利用web3.eth.getBalance改动代码：

class Index extends React.Component {

...

async componentDidMount(){

const web3 = new Web3(window.web3.currentProvider);

const accounts = await web3.eth.getAccounts();

**const balances = await Promise.all(accounts.map(x => web3.eth.getBalance(x)));**

~~console.log(accounts);~~

~~this.setState({ accounts });~~

**console.log(accounts, balances);**

**this.setState({ accounts: accounts.map((x, i) => ({ account: x, balance: balances[i] })) });**

}

render() {

const { accounts } = this.state;

return (

<Layout>

~~<ul>{accounts.map(x => <li key={x}>{x}</li>)}</ul>~~

**<ul>**

**{accounts.map(x => (**

**<li key={x.account}>**

**{x.account} => {x.balance}**

**</li>**

**))}**

**</ul>**

</Layout>

}

}

export default withRoot(Index);

代码里的Promise.all可以将多个Promise实例包装成一个新的Promise实例，成功的时候返回的是一个结果数组，而失败的时候则返回最先被reject失败状态的值。

而map() 方法会按照原始数组元素顺序依次处理元素，所以这样就实现了将一组异步调用的结果按原有顺序排列。

因为账户余额的单位是 wei，在 DApp 中展示为 ether 会更易读，我们可以使用 web3.utils.fromWei 工具函数做转换，尝试修改代码如下：

~~{x.account} => {x.balance}~~

**{x.account} => {web3.utils.fromWei(x.balance, 'ether')}**

然而我们却发现浏览器会报错：不能读取undefined变量的“fromWei”属性。查看代码自然能够想到，我们定义的Web3 实例是 componentDidMount 里面的局部变量，在 render 里面是无法访问的，那我们可以尝试把 Web3 的实例化挪到 constructor 里面：

class Index extends React.Component {

constructor(props){

...

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

}

async componentDidMount(){

~~const web3 = new Web3(window.web3.currentProvider);~~

const accounts = await **this.**web3.eth.getAccounts();

const balances = await Promise.all(accounts.map(x => **this.**web3.eth.getBalance(x)));

...

}

render() {

const { accounts } = this.state;

return (

...

{x.account} => {**this.**web3.utils.fromWei(x.balance, 'ether')}

...

}

}

然而刷新浏览器，依然会报错：window is not defined。这是为什么呢？因为架构上我们的 DApp 是支持服务端渲染的，当我们刷新页面时，渲染过程是在服务端执行的，这是一个Node.js环境，所以并没有window对象。

nextjs是react进行服务端渲染的一个工具，默认以根目录下的pages为渲染路由。nextjs是基于 react 的同构渲染方案，结合了webpack和自身提供的路由机制。

前后端同构是指在前后端维护同一份代码，它利用服务端渲染（SSR）出首屏，解除单页面应用（SPA）在首屏渲染上面临的窘境。同构渲染的目的，是将传统的纯服务端渲染的首屏优势和SPA的站内体验优势结合起来，以取得最优解的解决方案。

### 8.3 前后端通用的Web3实例

为了解决服务端渲染带来的Web3对象缺失问题，我们需要兼容前后端的去创建 Web3 实例，产生在浏览器环境和在 Node.js 环境都可以使用 Web3 实例，直接在 libs 目录下新建 web3.js，然后在其中输入如下代码：

import Web3 from 'web3';

let web3;

// if browser enviroment & Metamask exists

if (typeof window !== 'undefined' && typeof window.web3 !== 'undefined') {

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

} else {

web3 = new Web3(new Web3.providers.HttpProvider('http://localhost:8545'));

}

export default web3;

这段代码大家并不陌生，此前我们的项目中为了避免没有安装metamask的情况，就都用了这样的处理方式。

这段代码在初始化 Web3 示例时会检查当前的环境，如果是浏览器环境并且用户安装了Metamask，直接使用Metamask注入的Provider，否则使用HttpProvider通过本地的以太坊节点与区块链网络进行通信。

然后修改pages/index.js，使用这个前后端通用的 Web3 实例：

import React from 'react';

~~import Web3 from 'web3';~~

**import web3 from '../libs/web3';**

import { Button } from '@material-ui/core';

import withRoot from '../libs/withRoot';

import Layout from '../components/Layout';

class Index extends React.Component {

constructor(props){

super(props);

this.state = {

accounts: []

};

~~this.web3 = new Web3(window.web3.currentProvider);~~

}

async componentDidMount(){

**const accounts = await web3.eth.getAccounts();**

**const balances = await Promise.all(accounts.map(x => web3.eth.getBalance(x)));**

console.log(accounts);

this.setState({ accounts });

}

render() {

const { accounts } = this.state;

return (

<Layout>

<ul>

{accounts.map(x => (

<li key={x.account}>

**{x.account} => {web3.utils.fromWei(x.balance, 'ether')}**

</li>

))}

</ul>

</Layout>

}

}

export default withRoot(Index);

然后刷新浏览器，可以看到现在页面能正常显示账户、余额、并进行格式化。

## 9. 在DApp中渲染合约数据：项目列表页

### 9.1 封装通用的合约实例

我们之前采用的获取合约实例的代码，一般采用的是以下形式（注意这里是web3 v1.0.0版本的写法）：

new web3.eth.Contract(interface)

.deploy({data: byteCode})

.send({from: accounts[0], gas: 1000000})

而我们的代码中，ProjectList 智能合约在项目列表页和项目创建页都会使用，所以有必要将其封装成可复用的代码，在 libs 目录下新建 projectList.js，输入如下代码：

import web3 from './web3';

import ProjectList from '../compiled/ProjectList.json';

import address from '../address.json';

const contract = new web3.eth.Contract(JSON.parse(ProjectList.interface), address);

export default contract;

类似的，Project 合约会存在很多实例，需要有基于地址去生成合约实例的可复用函数，在 libs 目录下新建 project.js，输入如下代码：

import web3 from './web3';

import Project from '../compiled/Project.json';

const getContract = address =>

new web3.eth.Contract(JSON.parse(Project.interface), address);

export default getContract;

### 9.2 使用脚本添加测试数据

把合约数据渲染到 DApp 之前，还需要准备一些测试数据，我们通过Node.js脚本来实现。脚本文件是 scripts/sample.js，通过代码调用 ProjectList 合约的 createProject 接口添加测试数据。

const Web3 = require('web3');

const ProjectList = require('../compiled/ProjectList.json');

const address = require('../address.json');

const web3 = new Web3(new

Web3.providers.HttpProvider('http://localhost:8545'));

const contract = new web3.eth.Contract(JSON.parse(ProjectList.interface),

address);

(async () => {

const accounts = await web3.eth.getAccounts();

console.log(accounts);

const projects = [

{

description: 'Ethereum DApp Tutorial',

minInvest: web3.utils.toWei('0.01', 'ether'),

maxInvest: web3.utils.toWei('0.1', 'ether'),

goal: web3.utils.toWei('1', 'ether'),

},

{

description: 'Ethereum Video Tutorial',

minInvest: web3.utils.toWei('0.1', 'ether'),

maxInvest: web3.utils.toWei('1', 'ether'),

goal: web3.utils.toWei('5', 'ether')

},

];

console.log(projects);

const owner = accounts[0];

const results = await Promise.all(projects.map(x =>

contract

.methods

.createProject(x.description, x.minInvest, x.maxInvest, x.goal)

.send({ from: owner, gas: '1000000' })

)

);

console.log(results);

})();

### 9.3 获取并渲染项目列表

有了测试数据，接下来就需要在前端获取数据并在页面上渲染出来。可以修改pages/index.js:

import React from 'react';

import web3 from '../libs/web3';

~~import { Button } from '@material-ui/core';~~

**import { Grid, Button, Typography, Card, CardContent, CardActions } from '@material-ui/core';**

**import { Link } from '../routes';**

**import ProjectList from '../libs/projectList';**

import withRoot from '../libs/withRoot';

import Layout from '../components/Layout';

class Index extends React.Component {

~~constructor(props){~~

~~super(props);~~

~~this.state = {~~

~~accounts: []~~

~~};~~

~~}~~

~~async componentDidMount(){~~

~~const accounts = await web3.eth.getAccounts();~~

~~const balances = await Promise.all(accounts.map(x => web3.eth.getBalance(x)));~~

~~console.log(accounts);~~

~~this.setState({ accounts });~~

~~}~~

**static async getInitialProps({ req }) {**

**const projects = await ProjectList.methods.getProjects().call();**

**return { projects };**

**}**

render() {

~~const { accounts } = this.state;~~

**const { projects } = this.props;**

return (

<Layout>

~~<ul>~~

~~{accounts.map(x => (~~

~~<li key={x.account}>~~

~~{x.account} => {web3.utils.fromWei(x.balance, 'ether')}~~

~~</li>~~

~~))}~~

~~</ul>~~

**<Grid container spacing={16}>**

**{projects.map(this.renderProject)}**

**</Grid>**

</Layout>

}

**renderProject(project) {**

**return (**

**<Grid item md={6} key={project}>**

**<Card>**

**<CardContent>**

**<Typography gutterBottom variant="headline" component="h2">**

**{project}**

**</Typography>**

**<Typography component="p">{project}</Typography>**

**</CardContent>**

**<CardActions>**

**<Link route={`/projects/${project}`}>**

**<Button size="small" color="primary">**

**立即投资**

**</Button>**

**</Link>**

**<Link route={`/projects/${project}`}>**

**<Button size="small" color="secondary">**

**查看详情**

**</Button>**

**</Link>**

**</CardActions>**

**</Card>**

**</Grid>**

**);**

**}**

}

export default withRoot(Index);

改动里面有几点需要注意：

* 用 getInitialProps 获取项目地址列表，getInitialProps 是 Next.js 中服务端渲染时获取数据的基本方法；
* 用 Card 组件实现项目卡片，Card 是 Material UI 中的组件，用来展示项目列表非常合适；
* 用 next-routes 里面的 Link 组件实现路由跳转，目前详情页还没开发，点击跳转会 404。

由于ProjectList 合约里面只存储了所有 Project 合约实例的地址，所以目前我们在卡片上显示的信息很少。如果需要每个 Project 合约实例的基本信息，如标题，名称、投资数额限制等，可以在 Node.js 里循环所有的 Project 合约地址，创建合约实例，再调用他们的各个方法，但这将会产生大量的网络请求，会有比较严重的性能问题，因此，我们需要对 Project 合约稍作改进，增加一个获取项目基本信息的接口，代码如下：

Project.sol

function getSummary() public view returns (string, uint, uint, uint, uint, uint, uint, address) {

return (

description,

minInvest,

maxInvest,

goal,

address(this).balance,

investorCount,

payments.length,

owner

);

}

新增的 getSummary 接口会返回项目说明、投资金额限制、管理员地址、账户余额、资金支出条目、投资者数量等信息。

### 9.4 改进项目列表数据

接下来需要把 getSummary 接口返回的数据渲染在项目卡片中，继续修改 pages/index.js，调用 Project 合约的 getSummary 方法：

import React from 'react';

import web3 from '../libs/web3';

import { Grid, Button, Typography, Card, CardContent, CardActions } from '@material-ui/core';

import { Link } from '../routes';

**import Project from '../libs/project';**

import ProjectList from '../libs/projectList';

import withRoot from '../libs/withRoot';

import Layout from '../components/Layout';

class Index extends React.Component {

static async getInitialProps({ req }) {

~~const projects = await ProjectList.methods.getProjects().call();~~

**const addressList = await ProjectList.methods.getProjects().call();**

**const summaryList = await Promise.all(**

**addressList.map(address =>**

**Project(address)**

**.methods.getSummary()**

**.call();**

**)**

**);**

**console.log({ summaryList });**

**const projects = addressList.map((address, i) => {**

**const [description, minInvest, maxInvest, goal, balance, investorCount, paymentCount, owner] = Object.values(**

**summaryList[i]**

**);**

**return {**

**address,**

**description,**

**minInvest,**

**maxInvest,**

**goal,**

**balance,**

**investorCount,**

**paymentCount,**

**owner**

**};**

**});**

**console.log(projects);**

return { projects };

}

render() {

const { projects } = this.props;

return (

<Layout>

<Grid container spacing={16}>

{projects.map(this.renderProject)}

</Grid>

</Layout>

}

renderProject(project) {

return (

<Grid item md={6} key={project**.address**}>

<Card>

<CardContent>

<Typography gutterBottom variant="headline" component="h2">

{project**.description**}

</Typography>

<Typography component="p">{project**.address**}</Typography>

</CardContent>

<CardActions>

<Link route={`/projects/${project**.address**}`}>

<Button size="small" color="primary">

立即投资

</Button>

</Link>

<Link route={`/projects/${project**.address**}`}>

<Button size="small" color="secondary">

查看详情

</Button>

</Link>

</CardActions>

</Card>

</Grid>

);

}

}

export default withRoot(Index);

需要注意的是，getSummary 接口返回的数据格式是一个数组，可以看作一个Result对象，对象里面使用数字为键，对应的值是合约源代码中声明的值，所以需要使用 Object.values 把值取出来，再使用解构赋值。

在getInitialProps中完成数据提取部分之后，在renderProject渲染部分将对应的值写入。

到目前为止，我们的页面和原型图还差两点没实现：

* 募资进度条，可以直接使用 Material UI 中的 LinearProgress 组件来实现；
* 项目基本信息小方块，小方块可以抽成公用的组件，方便在项目详情页、列表页间复用；

继续对 pages/index.js 做如下修改，增加进度条展示，并引入一个叫做InfoBlock的子组件来展示项目中的每个参数：

...

import { Grid, Button, Typography, Card, CardContent, CardActions, **LinearProgress** } from '@material-ui/core';

...

**import InfoBlock from '../components/InfoBlock';**

class Index extends React.Component {

...

renderProject(project) {

**const progress = project.balance / project.goal \* 100;**

return (

...

<Typography gutterBottom variant="headline" component="h2">

{project.description}

</Typography>

**<LinearProgress style={{ margin: '10px 0' }} color="primary" variant="determinate" value={progress} />**

~~<Typography component="p">{project.address}</Typography>~~

**<Grid container spacing={16}>**

**<InfoBlock title={`${web3.utils.fromWei(project.goal, 'ether')} ETH`} description="募资上限" />**

**<InfoBlock title={`${web3.utils.fromWei(project.minInvest, 'ether')} ETH`} description="最小投资金额" />**

**<InfoBlock title={`${web3.utils.fromWei(project.maxInvest, 'ether')} ETH`} description="最大投资金额" />**

**<InfoBlock title={`${project.investorCount}人`} description="参投人数" />**

**<InfoBlock title={`${web3.utils.fromWei(project.balance, 'ether')} ETH`} description="已募资金额" />**

**</Grid>**

...

然后在 components 目录下新建 InfoBlock.js，输入如下代码：

import React from 'react';

import { Grid, Typography } from '@material-ui/core';

import { withStyles } from '@material-ui/core/styles';

const styles = {

container: {

padding: '0.5em 1em',

border: '1px dotted #AAA'

},

title: {

color: '#333',

marginBottom: '10px',

fontWeight: 'bold'

},

description: {

margin: 0,

color: '#666'

}

};

class InfoBlock extends React.Component {

render() {

const { classes, title, description } = this.props;

return (

<Grid item md={4}>

<div className={classes.container}>

<Typography variant="title" color="inherit" className={classes.title}>

{title}

</Typography>

<Typography variant="paragraph" color="inherit" className={classes.description}>

{description}

</Typography>

</div>

</Grid>

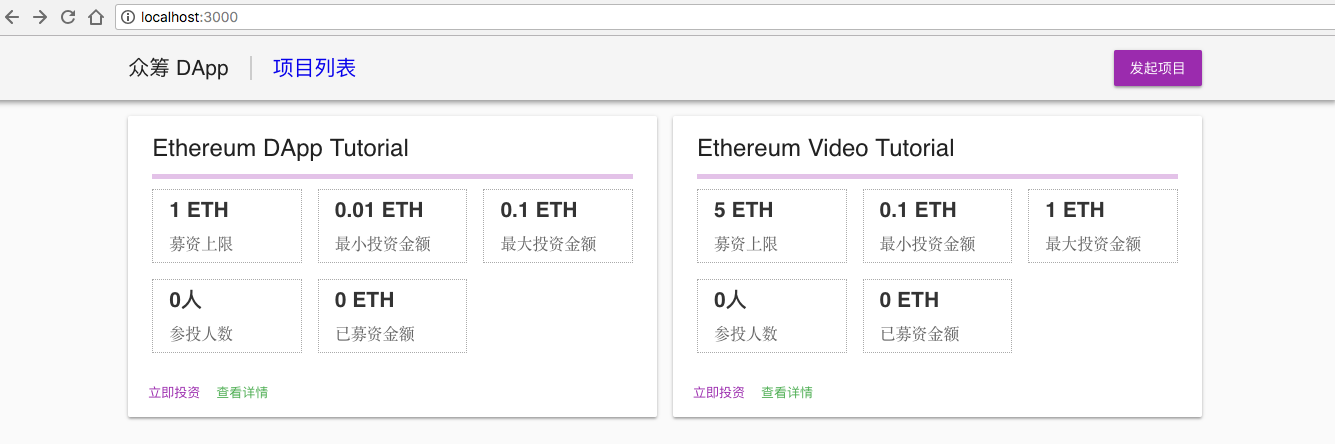
);

}

}

export default withStyles(styles)(InfoBlock);

改完代码之后，先npm run test保证合约代码正确，然后npm run deploy部署最新的合约，最后重启npm run dev，查看项目列表页应如下所示：



## 10. 从前端向合约提交数据：项目创建页

从智能合约获取数据并渲染在DApp页面中的流程我们已经很熟悉，接下来就是从 DApp 向智能合约提交数据，即项目创建页的实现。

### 10.1 项目创建页入口

之前在Header中已经有项目创建的按钮，但是没有加上链接跳转，修改 components/Header.js加上入口：

import React from 'react';

import { AppBar, Toolbar, Typography, Button } from '@material-ui/core';

import { withStyles } from '@material-ui/core/styles';

**import { Link } from '../routes';**

...

class Header extends React.Component {

render() {

const { classes } = this.props;

return (

...

**<Link route="/projects/create">**

<Button variant="raised" color="primary">

发起项目

</Button>

**</Link>**

...

### 10.2 项目创建表单

然后我们需要在 pages 目录下创建文件 pages/projects/create.js，用来展示创建项目的表单，内容如下：

import React from 'react';

import { Grid, Button, Typography, TextField, Paper } from '@material-ui/core';

import { Link } from '../../routes';

import web3 from '../../libs/web3';

import ProjectList from '../../libs/projectList';

import withRoot from '../../libs/withRoot';

import Layout from '../../components/Layout';

class ProjectCreate extends React.Component {

constructor(props) {

super(props);

this.state = {

description: '',

minInvest: 0,

maxInvest: 0,

goal: 0

};

}

getInputHandler(key) {

return e => {

console.log(e.target.value);

this.setState({ [key]: e.target.value });

};

}

render() {

return (

<Layout>

<Typography variant="title" color="inherit">

创建

</Typography>

<Paper style={{ width: '60%', padding: '15px', marginTop: '15px' }}>

<form noValidate autoComplete="off" style={{ marginBottom: '15px' }}>

<TextField

fullWidth

required

id="description"

label="项目名称"

value={this.state.description}

onChange={this.getInputHandler('description')}

margin="normal"

/>

<TextField

fullWidth

required

id="minInvest"

label="最小投资金额"

value={this.state.minInvest}

onChange={this.getInputHandler('minInvest')}

margin="normal"

InputProps={{ endAdornment: 'ETH' }}

/>

<TextField

fullWidth

required

id="maxInvest"

label="最大投资金额"

value={this.state.maxInvest}

onChange={this.getInputHandler('maxInvest')}

margin="normal"

InputProps={{ endAdornment: 'ETH' }}

/>

<TextField

fullWidth

required

id="goal"

label="募资上限"

value={this.state.goal}

onChange={this.getInputHandler('goal')}

margin="normal"

InputProps={{ endAdornment: 'ETH' }}

/>

</form>

<Button variant="raised" size="large" color="primary">

创建项目

</Button>

</Paper>

</Layout>

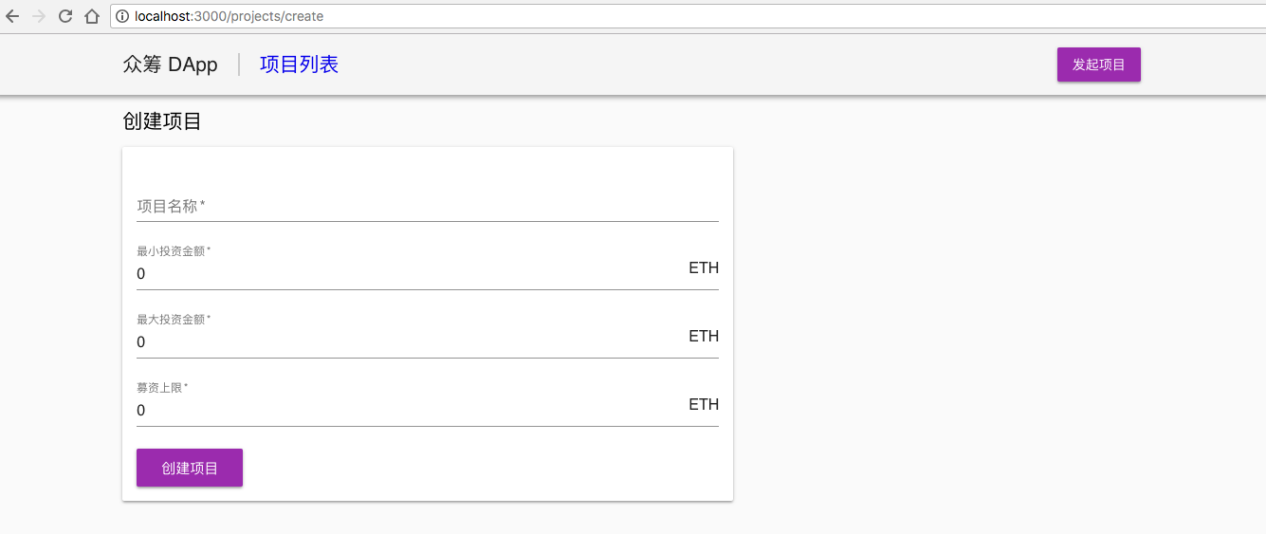
);

}

}

export default withRoot(ProjectCreate);

刷新浏览器，能看到项目创建表单如下：



接下来我们需要给表单提交添加对应的代码，具体代码改动如下：

create.js

import React from 'react';

import { Grid, Button, Typography, TextField, Paper**, CircularProgress** } from '@material-ui/core';

...

class ProjectCreate extends React.Component {

constructor(props) {

super(props);

this.state = {

description: '',

minInvest: 0,

maxInvest: 0,

goal: 0,

**errmsg: '',**

**loading: false**

};

**this.onSubmit = this.createProject.bind(this);**

}

...

**async createProject() {**

**const { description, minInvest, maxInvest, goal } = this.state;**

**console.log(this.state);**

**// filed validation check**

**if (!description) {**

**return this.setState({ errmsg: '项目名称不能为空' });**

**}**

**if (minInvest <= 0) {**

**return this.setState({ errmsg: '项目最小投资金额必须大于0' });**

**}**

**if (maxInvest <= 0) {**

**return this.setState({ errmsg: '项目最大投资金额必须大于0' });**

**}**

**if (maxInvest < minInvest) {**

**return this.setState({ errmsg: '项目最小投资金额必须小于最大投资金额' });**

**}**

**if (goal <= 0) {**

**return this.setState({ errmsg: '项目募资上限必须大于0' });**

**}**

**const minInvestInWei = web3.utils.toWei(minInvest, 'ether');**

**const maxInvestInWei = web3.utils.toWei(maxInvest, 'ether');**

**const goalInWei = web3.utils.toWei(goal, 'ether');**

**try {**

**this.setState({ loading: true });**

**// get accounts**

**const accounts = await web3.eth.getAccounts();**

**const owner = accounts[0];**

**// create project**

**const result = await ProjectList.methods**

**.createProject(description, minInvestInWei, maxInvestInWei, goalInWei)**

**.send({ from: owner, gas: '5000000' });**

**this.setState({ errmsg: '项目创建成功' });**

**console.log(result);**

**} catch (err) {**

**console.error(err);**

**this.setState({ errmsg: err.message || err.toString() });**

**} finally {**

**this.setState({ loading: false });**

**}**

**}**

render() {

return (

<Layout>

...

<Button variant="raised" size="large" color="primary" **onClick={this.onSubmit}** >

**{this.state.loading ? <CircularProgress color="secondary" size={24} /> : '创建项目'}**

</Button>

**{!!this.state.errmsg && (**

**<Typography component="p" style={{ color: 'red' }}>**

**{this.state.errmsg}**

**</Typography>**

**)}**

</Paper>

</Layout>

...

关于改动有几个关键点需要说明:

* 因为 createProject 是 Transaction 类型的接口，耗时通常比较长，DApp 在交互上需要给用户恰当的提示，可以借助 Material UI 中的 CircularProgress 来实现；
* 表单中填写的数值型字段的单位是 ether，但是调用 createProject 接口需要传入 wei，注意参数的单位转换；
* 在出错时需要给用户提示，所以设置了 errmsg 状态变量来显示错误提示。

另外注意JSX的语法表达，render中布尔值，NULL和Undefined都是合法的子元素，但它们都不会被渲染，所以可以用 *!!this.state.errmsg && < Typography >* 来表示：当errmsg不为空字符串时，渲染后面的Typography元素。

保存代码后，重新启动server，可以测试 DApp 的项目创建功能。当我们在浏览器中向智能合约发起交易时，会弹出 Metamask 的交易确认界面，我们需要在交易确认弹窗里面点击 Submit，交易才会被发到以太坊网络上。

## 11. 在DApp中同时展示和提交数据：项目详情页

按照原型图的设计，参与项目投资的功能内嵌在项目详情页中，先创建项目详情页的骨架，然后才能开发投资功能。

### 11.1 项目详情页框架

在 pages/projects 目录下新建 detail.js，代码如下：

import React from 'react';

import { Grid, Button, Typography, LinearProgress, Paper } from '@material-ui/core';

import { Link } from '../../routes';

import web3 from '../../libs/web3';

import Project from '../../libs/project';

import ProjectList from '../../libs/projectList';

import withRoot from '../../libs/withRoot';

import Layout from '../../components/Layout';

import InfoBlock from '../../components/InfoBlock';

class ProjectDetail extends React.Component {

static async getInitialProps({ query }) {

const contract = Project(query.address);

const summary = await contract.methods.getSummary().call();

const [description, minInvest, maxInvest, goal, balance, investorCount, paymentCount, owner] = Object.values( summary );

const project = {

address: query.address,

description,

minInvest,

maxInvest,

goal,

balance,

investorCount,

paymentCount,

owner

};

return { project };

}

render() {

const { project } = this.props;

return (

<Layout>

<Typography variant="title" color="inherit" style={{ margin: '15px 0' }}>

项目详情

</Typography>

{this.renderBasicInfo(project)}

</Layout>

);

}

renderBasicInfo(project) {

const progress = project.balance / project.goal \* 100;

return (

<Paper style={{ padding: '15px' }}>

<Typography gutterBottom variant="headline" component="h2">

{project.description}

</Typography>

<LinearProgress style={{ margin: '10px 0' }} color="primary" variant="determinate" value={progress} />

<Grid container spacing={16}>

<InfoBlock title={`${web3.utils.fromWei(project.goal, 'ether')} ETH`} description="募资上限" />

<InfoBlock title={`${web3.utils.fromWei(project.minInvest, 'ether')} ETH`} description="最小投资金额" />

<InfoBlock title={`${web3.utils.fromWei(project.maxInvest, 'ether')} ETH`} description="最大投资金额" />

<InfoBlock title={`${project.investorCount}人`} description="参投人数" />

<InfoBlock title={`${web3.utils.fromWei(project.balance, 'ether')} ETH`} description="已募资金额" />

</Grid>

</Paper>

);

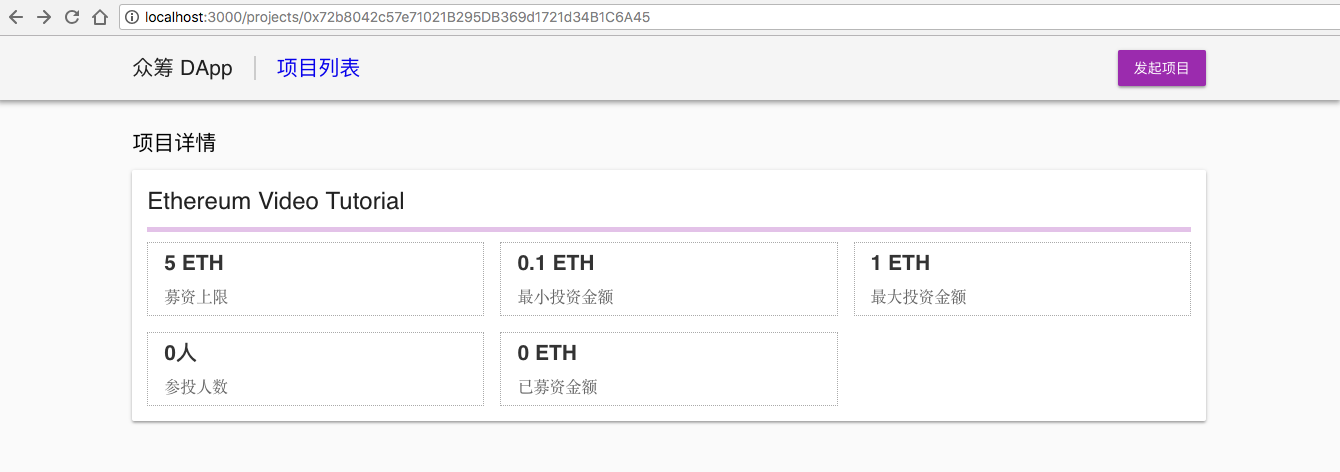
}

}

export default withRoot(ProjectDetail);

其中的getInitialProps方法，通过跳转路由中的query参数拿到合约地址，从而可以再次获取项目详细信息并渲染出来。

代码保存后，从项目列表页单击任意项目卡片底部的两个按钮都可以进入对应项目的详情页：



### 11.2 项目投资功能

接下来我们需要添加投资功能所需的界面元素：输入框 + 按钮，并且能在 React 组件状态中记录用户输入。代码改动如下：

import React from 'react';

import { Grid, Button, Typography, LinearProgress, Paper**, TextField** } from '@material-ui/core';

...

class ProjectDetail extends React.Component {

static async getInitialProps({ query }) {

...

return { project };

}

**constructor() {**

**super(props);**

**this.state = {**

**amount = 0;**

**};**

**}**

**getInputHandler(key) {**

**return e => {**

**console.log(e.target.value);**

**this.setState({ [key]: e.target.value });**

**};**

**}**

render() {

...

}

renderBasicInfo(project) {

const progress = project.balance / project.goal \* 100;

return (

<Paper style={{ padding: '15px' }}>

...

<Grid container spacing={16}>

...

</Grid>

**<Grid container spacing={16}>**

**<Grid item md={12}>**

**<TextField**

**required**

**id="amount"**

**label="投资金额"**

**style={{ marginRight: '15px' }}**

**value={this.state.amount}**

**onChange={this.getInputHandler('amount')}**

**margin="normal"**

**InputProps={{ endAdornment: 'ETH' }}**

**/>**

**<Button size="small" variant="raised" color="primary">**

**立即投资**

**</Button>**

**</Grid>**

**</Grid>**

</Paper>

);

}

}

export default withRoot(ProjectDetail);

然后给投资按钮添加行为，能够提交数据到 Project 合约实例，和创建项目类似，投资项目也是Transaction类型的接口调用，需要加上用户友好的状态提示：

import React from 'react';

import { Grid, Button, Typography, LinearProgress, **CircularProgress,** Paper, TextField } from '@material-ui/core';

...

class ProjectDetail extends React.Component {

...

constructor() {

super(props);

this.state = {

amount = 0;

**errmsg: '',**

**loading: false**

};

**this.onSubmit = this.contributeProject.bind(this);**

}

...

**async contributeProject() {**

**const { amount } = this.state;**

**const { minInvest, maxInvest } = this.props.project;**

**const minInvestInEther = web3.utils.fromWei(minInvest, 'ether');**

**const maxInvestInEther = web3.utils.fromWei(maxInvest, 'ether');**

**console.log({ amount, minInvestInEther, maxInvestInEther });**

**// filed validation check**

**if (amount <= 0) {**

**return this.setState({ errmsg: '投资金额必须大于0' });**

**}**

**if (amount < minInvestInEther) {**

**return this.setState({ errmsg: '投资金额必须大于最小投资金额' });**

**}**

**if (amount > maxInvestInEther) {**

**return this.setState({ errmsg: '投资金额必须小于最大投资金额' });**

**}**

**try {**

**this.setState({ loading: true, errmsg: '' });**

**// get accounts**

**const accounts = await web3.eth.getAccounts();**

**const owner = accounts[0];**

**// invest to project**

**const contract = Project(this.props.project.address);**

**const result = await contract.methods**

**.contribute()**

**.send({ from: owner, value: web3.utils.toWei(amount, 'ether'), gas: '5000000' });**

**this.setState({ errmsg: '投资成功', amount: 0 });**

**console.log(result);**

**setTimeout(() => {**

**location.reload();**

**}, 1000);**

**} catch (err) {**

**console.error(err);**

**this.setState({ errmsg: err.message || err.toString() });**

**} finally {**

**this.setState({ loading: false });**

**}**

**}**

render() {

...

}

renderBasicInfo(project) {

const progress = project.balance / project.goal \* 100;

return (

<Paper style={{ padding: '15px' }}>

...

<Grid container spacing={16}>

...

</Grid>

<Grid container spacing={16}>

<Grid item md={12}>

<TextField

required

id="amount"

label="投资金额"

style={{ marginRight: '15px' }}

value={this.state.amount}

onChange={this.getInputHandler('amount')}

margin="normal"

InputProps={{ endAdornment: 'ETH' }}

/>

<Button size="small" variant="raised" color="primary" **onClick={this.onSubmit}**>

**{this.state.loading ? <CircularProgress color="secondary" size={24} /> : '立即投资'}**

</Button>

**{!!this.state.errmsg && (**

**<Typography component="p" style={{ color: 'red' }}>**

**{this.state.errmsg}**

**</Typography>**

**)}**

</Grid>

</Grid>

</Paper>

);

}

}

export default withRoot(ProjectDetail);

代码中几个注意点：

* 校验投资金额是否满足要求时，使用 web3.utils.fromWei 转换成了 ether 进行对比，因为 BigNumber 的问题；
* 调用 contribute 接口时的投资金额是在 Transaction 配置里面，而不是作为参数传给 contribute；
* 投资成功之后，页面应该自动刷新（1秒钟后），就能反应出最新的投资人数量和募资金额。

和创建项目类似，当在浏览器中向智能合约发起交易时，会弹出 Metamask 的交易确认界面，我们需要在交易确认弹窗里面点击 Submit，交易才会被发到以太坊网络上。

## 12. 众筹DApp中的项目资金管理功能

项目的资金支出列表在项目详情页，创建资金支出请求的入口也在项目详情页。

### 12.1 创建资金支出请求的入口

添加入口很简单，直接修改 pages/projects/detail.js：

...

class ProjectDetail extends React.Component {

static async getInitialProps({ query }) {

...

render() {

const { project } = this.props;

return (

<Layout>

<Typography variant="title" color="inherit" style={{ margin: '15px 0' }}>

项目详情

</Typography>

{this.renderBasicInfo(project)}

**<Typography variant="title" color="inherit" style={{ margin: '30px 0 15px' }}>**

**资金支出请求**

**</Typography>**

**{this.renderPayments(project)}**

</Layout>

);

}

renderBasicInfo(project) {

...

}

**renderPayments(project) {**

**console.log(project);**

**return (**

**<Paper style={{ padding: '15px' }}>**

**<Link route={`/projects/${project.address}/payments/create`}>**

**<Button variant="raised" color="primary">**

**创建资金支出请求**

**</Button>**

**</Link>**

**</Paper>**

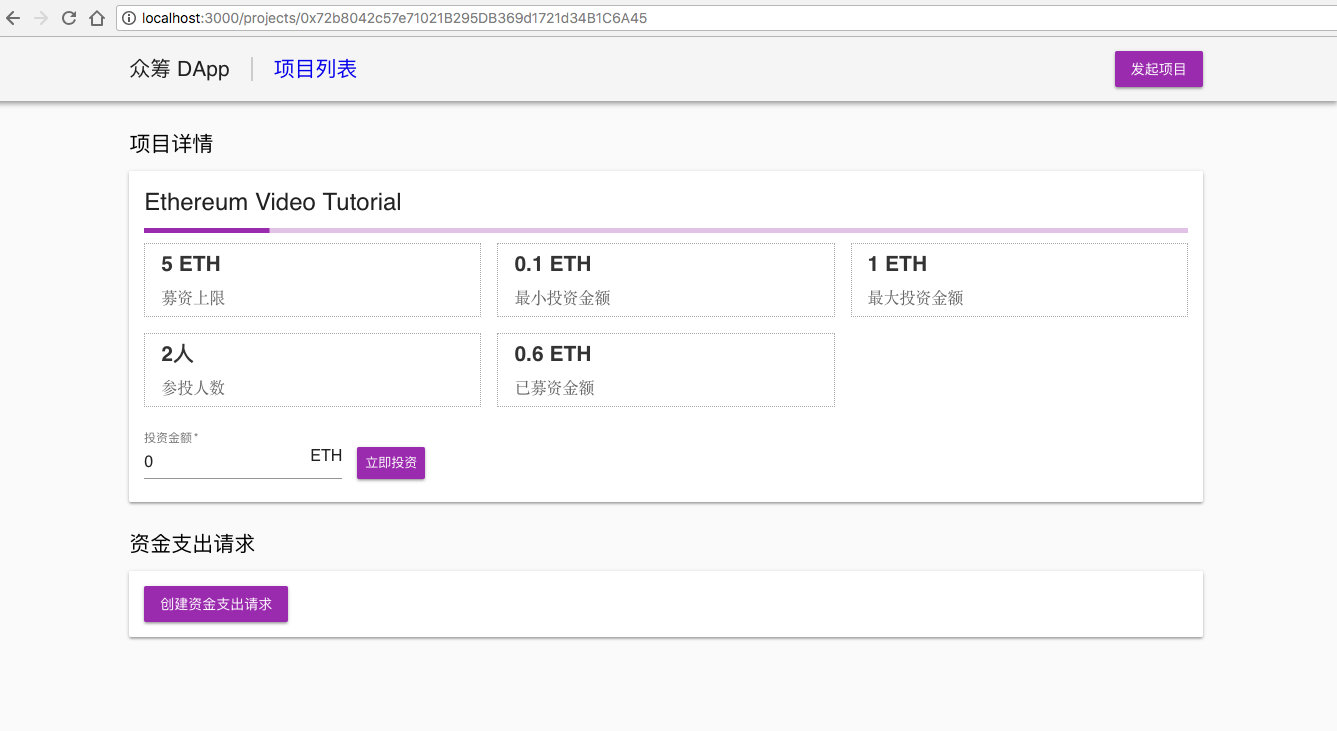
**);**

**}**

}

export default withRoot(ProjectDetail);

修改完的项目详情页如下：



### 12.2 创建资金支出请求

资金支出请求创建页面，可以完全参照项目创建页面去做，在 pages/projects 下面新建 payments 目录，然后在其中新建 create.js，输入代码：

import React from 'react';

import { Grid, Button, Typography, TextField, Paper, CircularProgress } from '@material-ui/core';

import { Router } from '../../../routes';

import web3 from '../../../libs/web3';

import Project from '../../../libs/project';

import withRoot from '../../../libs/withRoot';

import Layout from '../../../components/Layout';

class PaymentCreate extends React.Component {

static async getInitialProps({ query }) {

const contract = Project(query.address);

const summary = await contract.methods.getSummary().call();

const description = summary[0];

const owner = summary[7];

return { project: { address: query.address, description, owner } };

}

constructor(props) {

super(props);

this.state = {

description: '',

amount: 0,

receiver: 0,

errmsg: '',

loading: false

};

this.onSubmit = this.createPayment.bind(this);

}

getInputHandler(key) {

return e => {

console.log(e.target.value);

this.setState({ [key]: e.target.value });

};

}

async createPayment() {

const { description, amount, receiver } = this.state;

console.log(this.state);

// field validation check

if (!description) {

return this.setState({ errmsg: '支出理由不能为空' });

}

if (amount <= 0) {

return this.setState({ errmsg: '支出金额必须大于0' });

}

if (!web3.utils.isAddress(receiver)) {

return this.setState({ errmsg: '收款人账户地址不正确' });

}

const amountInWei = web3.utils.toWei(amount, 'ether');

try {

this.setState({ loading: true, errmsg: '' });

// get accounts

const accounts = await web3.eth.getAccounts();

const sender = accounts[0];

// check account

if(sender !== this.props.project.owner) {

return window.alert('只有管理员才能创建支出请求');

}

// create payment

const contract = Project(this.props.project.address);

const result = await contract.methods

.createPayment(description, amountInWei, receiver)

.send({ from: sender, gas: '5000000' });

this.setState({ errmsg: '资金支出请求创建成功' });

console.log(result);

setTimeout(() => {

Router.pushRoute(`/projects/${this.props.project.address}`);

}, 1000);

} catch (err) {

console.error(err);

this.setState({ errmsg: err.message || err.toString() });

} finally {

this.setState({ loading: false });

}

}

render() {

return (

<Layout>

<Typography variant="title" color="inherit" style={{ marginTop: '15px' }}>

创建资金支出请求：{this.props.project.description}

</Typography>

<Paper style={{ width: '60%', padding: '15px', marginTop: '15px' }}>

<form noValidate autoComplete="off" style={{ marginBottom: '15px' }}>

<TextField

fullWidth

required

id="description"

label="支出理由"

value={this.state.description}

onChange={this.getInputHandler('description')}

margin="normal"

/>

<TextField

fullWidth

required

id="amount"

label="支出金额"

value={this.state.amount}

onChange={this.getInputHandler('amount')}

margin="normal"

InputProps={{ endAdornment: 'ETH' }}

/>

<TextField

fullWidth

required

id="receiver"

label="收款方"

value={this.state.receiver}

onChange={this.getInputHandler('receiver')}

margin="normal"

/>

</form>

<Button variant="raised" size="large" color="primary" onClick={this.onSubmit} >

{this.state.loading ? <CircularProgress color="secondary" size={24} /> : '保存'}

</Button>

{!!this.state.errmsg && (

<Typography component="p" style={{ color: 'red' }}>

{this.state.errmsg}

</Typography>

)}

</Paper>

</Layout>

);

}

}

export default withRoot(PaymentCreate);

这里我们从项目的summary中提取出owner信息，并在发起创建支出请求的交易之前做了身份验证，只有项目的创建人才能提起请求。另外，创建成功后应该刷新页面，并且跳转到项目详情页。

保存后，我们就可以从项目详情页开始测试资金支出请求的创建功能了。

### 12.3 资金支出列表

资金支出请求的数据现在已经有了，我们需要把它读取出来并渲染到 DApp 中。继续修改 pages/projects/detail.js，读取 payments 列表，并渲染成表格：

import { Grid, Button, Typography, LinearProgress, CircularProgress, Paper, TextField, **Table, TableHead, TableBody, TableRow, TableCell** } from '@material-ui/core';

...

class ProjectDetail extends React.Component {

static async getInitialProps({ query }) {

const contract = Project(query.address);

const summary = await contract.methods.getSummary().call();

const [description, minInvest, maxInvest, goal, balance, investorCount, paymentCount, owner] = Object.values( summary );

**const task = [];**

**for (let i = 0; i < paymentCount; i++) {**

**tasks.push(contract.methods.payments(i).call());**

**}**

**const payments = await Promise.all(tasks);**

const project = {

address: query.address,

description,

minInvest,

maxInvest,

goal,

balance,

investorCount,

paymentCount,

owner,

**payments**

};

return { project };

}

...

render() {

...

}

renderBasicInfo(project) {

...

}

renderPayments(project) {

console.log(project);

return (

<Paper style={{ padding: '15px' }}>

**<Table style={{ marginBottom: '30px' }}>**

**<TableHead>**

**<TableRow>**

**<TableCell>支出理由</TableCell>**

**<TableCell numeric>支出金额</TableCell>**

**<TableCell>收款人</TableCell>**

**<TableCell>是否完成</TableCell>**

**<TableCell>投票状态</TableCell>**

**<TableCell>操作</TableCell>**

**</TableRow>**

**</TableHead>**

**<TableBody>**

**{project.payments.map(payment => {**

**return (**

**<TableRow key={payment.id}>**

**<TableCell>{payment.description}</TableCell>**

**<TableCell numeric>{web3.utils.fromWei(payment.amount, 'ether')} ETH</TableCell>**

**<TableCell>{payment.receiver}</TableCell>**

**<TableCell>{payment.completed ? '是' : '否'}</TableCell>**

**<TableCell>{payment.voterCount}/{project.investorCount}</TableCell>**

**<TableCell></TableCell>**

**</TableRow>**

**);**

**})}**

**</TableBody>**

**</Table>**

<Link route={`/projects/${project.address}/payments/create`}>

<Button variant="raised" color="primary">

创建资金支出请求

</Button>

</Link>

</Paper>

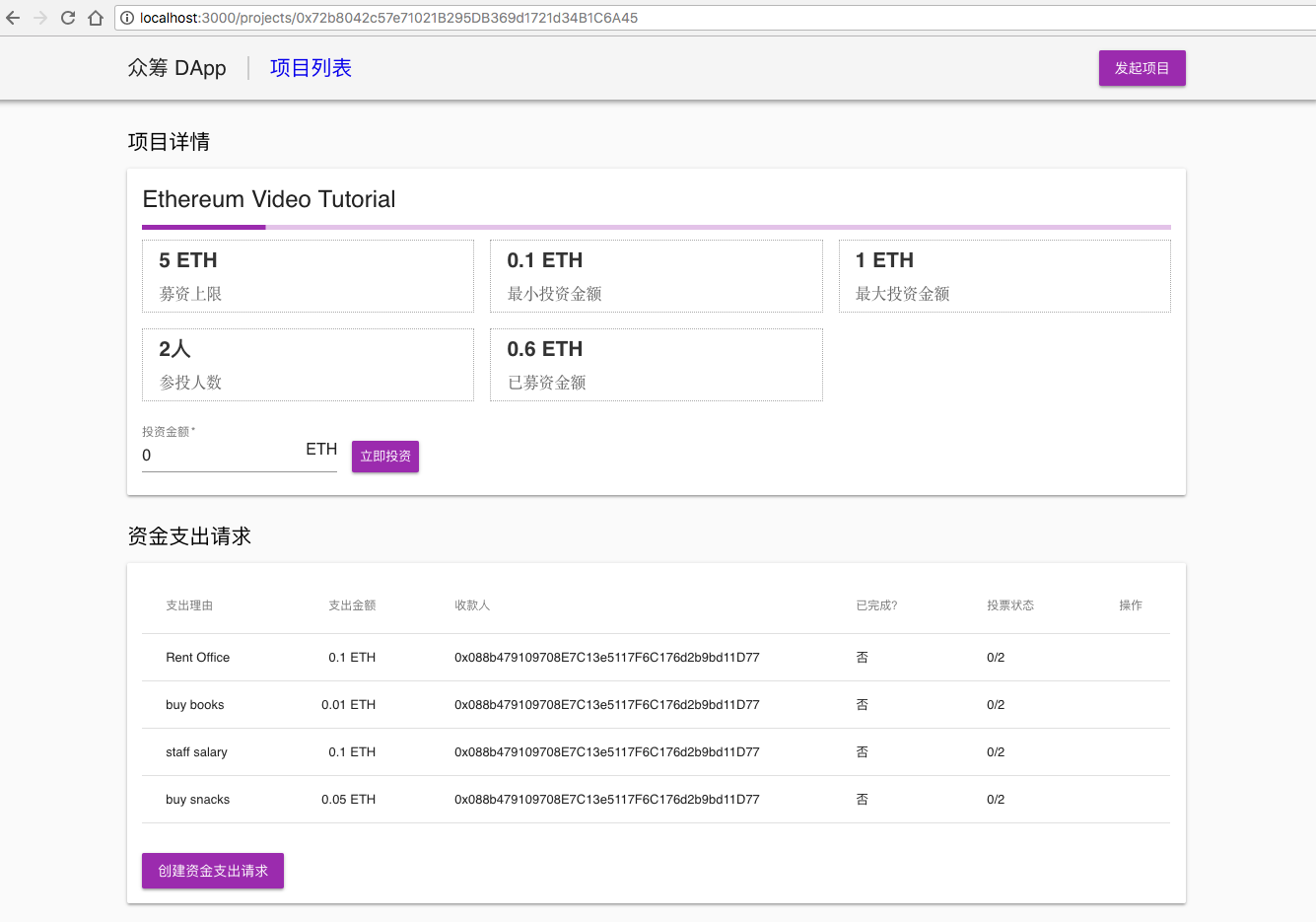
);

}

}

export default withRoot(ProjectDetail);

资金支持列表渲染后的详情页如下：



### 12.4 资金支出投票

接下来是资金支出请求的投票功能，继续修改 pages/projects/detail.js：

...

class ProjectDetail extends React.Component {

static async getInitialProps({ query }) {

...

}

constructor() {

super(props);

this.state = {

amount = 0;

errmsg: '',

loading: false,

**isApproving: false**

};

this.onSubmit = this.contributeProject.bind(this);

}

...

**async approvePayment(i) {**

**try {**

**this.setState({ isApproving: i });**

**// get accounts**

**const accounts = await web3.eth.getAccounts();**

**const sender = accounts[0];**

**// invest to project**

**const contract = Project(this.props.project.address);**

**const isInvestor = await contract.methods.investors(sender).call();**

**if (!isInvestor) {**

**return window.alert('只有投资人才有权投票');**

**}**

**const result = await contract.methods**

**.approvePayment(i)**

**.send({ from: sender, gas: '5000000' });**

**window.alert('投票成功');**

**setTimeout(() => {**

**location.reload();**

**}, 1000);**

**} catch (err) {**

**console.error(err);**

**window.alert(err.message || err.toString() );**

**} finally {**

**this.setState({ isApproving: false });**

**}**

**}**

render() {

...

}

renderBasicInfo(project) {

...

}

renderPayments(project) {

console.log(project);

return (

<Paper style={{ padding: '15px' }}>

<Table style={{ marginBottom: '30px' }}>

<TableHead>

...

</TableHead>

<TableBody>

~~{project.payments.map(payment => {~~

~~return (~~

~~<TableRow key={payment.id}>~~

~~<TableCell>{payment.description}</TableCell>~~

~~<TableCell numeric>{web3.utils.fromWei(payment.amount, 'ether')} ETH</TableCell>~~

~~<TableCell>{payment.receiver}</TableCell>~~

~~<TableCell>{payment.completed ? '是' : '否'}</TableCell>~~

~~<TableCell>{payment.voterCount}/{project.investorCount}</TableCell>~~

~~<TableCell></TableCell>~~

~~</TableRow>~~

~~);~~

~~})}~~

**{project.payments.map((payment, index) => this.renderPaymentRow(payment, index, project))}**

</TableBody>

</Table>

...

</Paper>

);

}

**renderPaymentRow(payment, index, project)** {

const canApprove = !payment.completed;

return (

<TableRow key={payment.id}>

<TableCell>{payment.description}</TableCell>

<TableCell numeric>{web3.utils.fromWei(payment.amount, 'ether')} ETH</TableCell>

<TableCell>{payment.receiver}</TableCell>

<TableCell>{payment.completed ? '是' : '否'}</TableCell>

<TableCell>{payment.voterCount}/{project.investorCount}</TableCell>

**<TableCell>**

**{canApprove && (**

**<Button size="small" color="primary" onClick={() => this.approvePayment(index)}>**

**{this.isApproving(index) ? <CircularProgress color="secondary" size={24} /> : '投赞成票'}**

**</Button>**

**)}**

**</TableCell>**

</TableRow>

);

}

**isApproving(i) {**

**return typeof this.state.isApproving === 'number' && this.state.isApproving === i;**

**}**

}

export default withRoot(ProjectDetail);

这里的改动需要说明几点：

* 把资金支出请求的渲染抽到了单独的函数里面，方便后续扩展；
* 投票按钮的出现条件是资金支出请求没有完成划转；
* 投票按钮的 loading 状态和参投的 loading 状态实现机制稍有不同，因为页面上存在多条资金支出请求，我们需要判断出在处理哪条，所以增加了状态isApproving 来记录当前处理的请求index。

### 12.5 资金划转

最后还应该实现资金划转功能，继续修改 pages/projects/detail.js：

...

class ProjectDetail extends React.Component {

static async getInitialProps({ query }) {

...

}

constructor() {

super(props);

this.state = {

amount = 0;

errmsg: '',

loading: false,

isApproving: false,

**isPaying: false**

};

this.onSubmit = this.contributeProject.bind(this);

}

...

**async doPayment(i) {**

**try {**

**this.setState({ isPaying: i });**

**// get accounts**

**const accounts = await web3.eth.getAccounts();**

**const sender = accounts[0];**

**// check account**

**if(sender !== this.props.project.owner) {**

**return window.alert('只有管理员才能划转资金');**

**}**

**// invest to project**

**const contract = Project(this.props.project.address);**

**const result = await contract.methods**

**.approvePayment(i)**

**.send({ from: sender, gas: '5000000' });**

**window.alert('资金划转成功');**

**setTimeout(() => {**

**location.reload();**

**}, 1000);**

**} catch (err) {**

**console.error(err);**

**window.alert(err.message || err.toString() );**

**} finally {**

**this.setState({ isPaying: false });**

**}**

**}**

render() {

...

}

renderBasicInfo(project) {

...

}

renderPayments(project) {

...

}

renderPaymentRow(payment, index, project) {

const canApprove = !payment.completed;

**const canDoPayment = !payment.completed && payment.voterCount / project.investorCount > 0.5;**

return (

...

<TableCell>

{canApprove && (

<Button size="small" color="primary" onClick={() => this.approvePayment(index)}>

{this.isApproving(index) ? <CircularProgress color="secondary" size={24} /> : '投赞成票'}

</Button>

)}

**{canDoPayment && (**

**<Button size="small" color="primary" onClick={() => this.doPayment(index)}>**

**{this.isPaying(index) ? <CircularProgress color="secondary" size={24} /> : '资金划转'}**

**</Button>**

**)}**

</TableCell>

</TableRow>

);

}

isApproving(i) {

return typeof this.state.isApproving === 'number' && this.state.isApproving === i;

}

**isPaying(i) {**

**return typeof this.state.isPaying === 'number' && this.state.isPaying === i;**

**}**

}

export default withRoot(ProjectDetail);

资金划转的实现套路和投赞成票的实现套路非常类似，除了按钮出现条件是要求投票人数超过投资人 50%。

我们可以重新启动服务器，完整地测试创建资金支出请求、投票、资金划转的整个流程。至此，我们已经实现了最初设计的众筹DApp的全部功能。