Article.eth：智能合约的编写

作为整个 Article.eth DApp 项目的基础，智能合约是整个项目中最至关重要的部分。本文详细阐述了整个智能合约的架构，以及编写和测试的过程。

开始之前，搭建环境

不造不必要的轮子相信已经成为项目开发时各位程序员秉承的信念。那么一个 DApp 有哪些可以让我们减少工作量的轮子可供使用呢？Truffle 或许是你不错的选择，它不仅会在你前期的合约编写中提供助力，也会让你能够非常快速高效地构建一个完整的，包含前后端的 DApp。下面我们简述如何为电脑配置好 truffle 环境，本文基于带有 HomeBrew 命令行工具的 macOS 构建，不同的操作系统和环境可能在具体命令上有所不同。

```
brew update && brew install node && npm install -g truffle
```

上面的命令一共做了三件事：

1. 通过 HomeBrew 更新当前安装的所有包
2. 通过 HomeBrew 安装 Nodejs 运行环境
3. 通过 Nodejs 的包管理器 npm 安装 truffle 框架的工具包

在上面的命令执行完成（国内可能需要科学上网，主要是 npm 需要进行特别的配置）之后，我们就已经配置好了 truffle 使用所需要的全部环境。接下来，我们需要创建一个工作目录来继续我们的工作：

```
cd ~
mkdir ethereum && cd ethereum
mkdir truffle && cd truffle
```

通过上面的命令，我们创建了一个路径为 `~/ethereum/truffle` 的文件夹并进入其中，其中 `~` 是用户文件夹。之后我们的开发任务都将在这个目录下进行。

创建 Truffle 项目及项目结构分析

然后，我们就可以在当前目录下新建一个空的 truffle 项目了（你也可以通过 unbox 命令来以模板新建项目，本次作业暂时用不到，我们就直接新建了空项目）：

```
truffle init
```

然后你会发现一个如下的文件目录结构：

```
|- contracts/
|-- Migrations.sol
|- migrations/
|-- 1_initial_migration.js
|- test/
|- truffle-config.js
|- truffle.js
```

他们的作用如下：

- contracts：存放所有的合约文件的文件夹
  - Migrations.sol：确保 Truffle 的 Migrate 功能正常使用的合约
- migrations：配置 Migrate 功能的配置文件存放地
  - 1_initial_migration.js：用于初始化 Migrate 的配置文件
- tests：项目测试文件的存放地
- truffle-config.js：项目配置文件

- truffle.js：项目的 Nodejs 入口

其中，Migrate 是 Truffle 带给我们的最重要的特性之一，他可以很方便地让我们在链上部署合约，具体来讲，当我们需要新部署一系列合约时，我们只需要新建一个文件，命名格式为 序号 + 名字 + .js，然后按如下格式编写即可：

```
var contractName = artifacts.require("./contractName.sol");

module.exports = function(deployer) {
  deployer.deploy(contractName);
};
```

关于 Migrate 更复杂的用法可以参照官网进行学习。

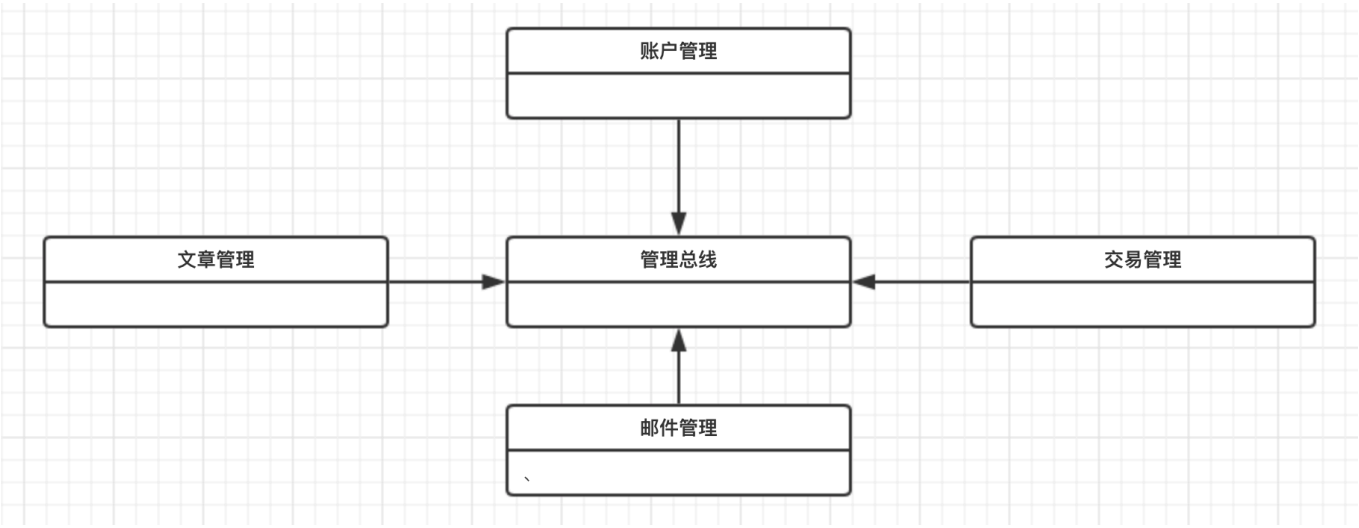
Article.eth 项目介绍

设计理念和需求功能

Article.eth 是基于区块链构建的作家权益保护及有偿创作交易平台。我们将其设计为作家和出版商可以自由、匿名地进行作品交易、信息交换、版权认证的平台。因而我们需要提供以下功能：

- 用户账户的管理
- 用户间的邮件发送和接受
- 用户间的版权交易的创建和完成
- 作者的文章撰写和记录

项目结构



项目的简化结构如上图，我们将系统拆分为四个子系统来分担不同的任务，最后通过总线进行系统间通信和综合。具体到文件来说：

- AccountSystem.sol：账户管理系统
- ArticleSystem.sol：文章管理系统
- OrderSystem.sol：订单管理系统
- MailSystem.sol：邮件管理系统
- ControlBus.sol：控制总线

下面我们将分各个系统来进行讲解。

账户管理系统

账户管理系统负责所有用户账号的管理。在这个合约下，我们把用户划分为两类：作者和出版商。通过 `mapping` ，我们使得用户可以通过自己的 address 来寻找作为作者或是出版商的用户信息（一个地址可以同时作为两种身份存在）。

在用户信息中，我们存储了一个邮件列表、一个订单列表、一个（拥有版权的）文章列表：

```
struct Account {
  address owner;
  uint articleCounter;
  uint orderCounter;
  uint mailCounter;

  mapping (uint => uint) mails;
  mapping (uint => uint) orders;
  mapping (uint => OwnedIndex) ownedArticles;
}
```

这些列表都是存储的对应实体的 index，我们将能够在控制总线中将这些 index 对应到相应的数据实体。

具体到写者和出版商，他们分别独占一个作品列表和一个创建的订单列表。实现和上面的列表大同小异：

```
struct Writer {
  Account account;
  uint writerId;
  uint articleCounter;
  mapping (uint => uint) workedArticles;
}

struct Publisher {
  Account account;
  uint publisherId;
  uint orderCounter;
  mapping (uint => uint) sentOrders;
}
```

最后，我们把用户信息组织为 mapping，方便之后的调用：

```
uint internal writerCounter = 1;
mapping (address => uint) internal writerIndex;
mapping (uint => Writer) internal writers;

uint internal publisherCounter = 1;
mapping (address => uint) internal publisherIndex;
mapping (uint => Publisher) internal publishers;
```

为了保证数据不会被非法修改，我们令所有的 State Variables 为 internal。

同时，为了使得用户和其他合约可以操作这些数据，我们实现了各个字段的 get 或 set 方法，并且根据数据的敏感程度设置为 internal 或是 public：

```
function createWriter() public {
  require(writerIndex[msg.sender] == 0);
  writers[writerCounter] = Writer(Account(msg.sender, 0, 0, 0), writerCounter, 0);
  writerCounter++;
}
// ...
function _getWriterAt(uint index) internal view
validWriter(index) returns(Writer) {
  return writers[index];
}
// ...
```

文章管理系统

文章管理系统负责处理文章数据结构及其管理。我们设计的文章数据结构如下：

```
struct Content {
  string headContent;
  bytes32 hashedContent;
```

```
  uint length;
}

struct Article {
  address writer;
  address owner;
  Content articleContent;
}
```

其中， `headContent` 是文章的开头的明文，用于向其他用户展示作者的历史作品； `hashedContent` 是文章全文经过两次 sha-3 哈希后的算法结果，用于校验一个字符串是否是文章； `length` 是文章的长度，用于辅助校验文章。在 `Article` 中， `writer` 记录了作者的账户地址，而 `owner` 则记录版权的所有者地址。

如此设计的目的在于，在不损失其公开可查看性的前提下提供重复检查的功能。作者既可以通过头部明文展现他的创作水平，又可以通过哈希结果来隐藏全文，确保版权方利益不受损，同时还可以进行文章校验。

订单管理系统

订单管理系统负责订单数据结构及其管理。订单的数据结构定义如下：

```
enum State {unfinished, finished}

struct Order {
  address seller;
  address buyer;
  uint price;
  State state;
}
```

这样的数据结构使得它可以记录订单的交易双方的地址，交易的金额，以及交易的状态。但是**这并不是订单的重点**，支付操作才是订单有意义的关键。在订单管理系统中，我们实现了方法 `_finishOrder`：

```
function _finishOrder(uint index) internal {
  require(index < orderCounter);
  require(orders[index].seller == msg.sender);
  require(orders[index].state == State.unfinished);
  msg.sender.transfer(orders[index].price);
  orders[index].state = State.finished;
}
```

他会完成一个订单，并且将订单内存放的金额支付给卖家。那么这个金额要如何得来呢？我们在 `_createOrder` 中并没有存放任何金额啊！这个问题就要留到控制总线来解答了。

邮件管理系统

邮件系统负责邮件的数据结构及其管理。通过这个系统，我们实现了任意两个地址用户之间的通信。邮件的数据结构非常简单：

```
struct Mail {
  address sender;
  address receiver;
  string content;
}
```

同时，我们也创建了一个 mapping 来管理这些邮件：

```
uint internal mailCounter = 0;
mapping (uint => Mail) internal mails;
```

最后，我们预留了 `_sendMail` 和 `getMailContentByIndex` 以供外部调用。

控制总线

控制总线沟通其余的四个子系统，完成所有需要跨系统进行的操作。因而从定义上来说，他是所有四个子系统的继承：

```
contract ControlBus is ArticleSystem, AccountSystem, OrderSystem, MailSystem {
  // ...
}
```

特别的，我们来讲讲创建订单和发送邮件这两个操作。前面讲到，订单的子系统并没有实现“存款”的操作。这个操作实际上就是在数据总线中实现的，因为这要涉及到付款账户的确认：

```
function createOrder(address writer) public payable {
  _createOrder(writer, msg.value);
}
```

在这里，我们调用订单子系统的方法进行创建订单，其中，我们令订单价格为 `msg.value` ，这是用户付给合约账户的钱（得益于 `payable` 的修饰符），这笔钱被临时锁在了合约中，在之后完成订单时，我们付款用到的钱也就是这部分钱。

另一个操作是发送邮件，我们实际上在这里遇到了一些困难：地址只有一个，怎样知道发给的是谁？我们解决这个问题的方法比较暴力，我们给同一地址的不同身份均发送这封邮件：

```
function writeMail(address receiver, string content) public {
  uint sendWriter = writerIndex[msg.sender];
  uint recvWriter = writerIndex[receiver];
  uint sendPublisher = publisherIndex[msg.sender];
  uint recvPublisher = publisherIndex[receiver];


  require(sendWriter != 0 && sendPublisher != 0);
  require(recvPublisher != 0 && recvWriter != 0);
  uint index = _sendMail(receiver, content);

  if (sendWriter != 0) {
    writers[sendWriter].account.mails[writers[sendWriter].account.mailCounter++]
      = index;
  }
  if (sendPublisher != 0) {
    publishers[sendPublisher].account.mails[publishers[sendPublisher].account.mailCounter++]
      = index;
  }

  if (recvWriter != 0) {
    writers[recvWriter].account.mails[writers[recvWriter].account.mailCounter++]
      = index;
  }
  if (recvPublisher != 0) {
    publishers[recvPublisher].account.mails[publishers[recvPublisher].account.mailCounter++]
      = index;
  }
}
```

合约测试

测试环境

Truffle 提供了诸多可兼容的测试环境，我们在本项目中选用了 truffle develop，因为它直接被包含在 truffle 工具包中，使用方便且简单。

首先运行 `truffle develop` 来开启测试环境：

```
Emilia:ArticleChain siskon$ truffle develop
Truffle Develop started at http://127.0.0.1:9545/

Accounts:
(0) 0x627306090abab3a6e1400e9345bc60c78a8bef57
```

```
(1) 0xf17f52151ebef6c7334fad080c5704d77216b732
(2) 0xc5fdf4076b8f3a5357c5e395ab970b5b54098fef
(3) 0x821aea9a577a9b44299b9c15c88cf3087f3b5544
(4) 0x0d1d4e623d10f9fba5db95830f7d3839406c6af2
(5) 0x2932b7a2355d6fecc4b5c0b6bd44cc31df247a2e
(6) 0x2191ef87e392377ec08e7c08eb105ef5448eced5
(7) 0x0f4f2ac550a1b4e2280d04c21cea7ebd822934b5
(8) 0x6330a553fc93768f612722bb8c2ec78ac90b3bbc
(9) 0x5aeda56215b167893e80b4fe645ba6d5bab767de

Private Keys:
(0) c87509a1c067bbde78beb793e6fa76530b6382a4c0241e5e4a9ec0a0f44dc0d3
(1) ae6ae8e5ccbfb04590405997ee2d52d2b330726137b875053c36d94e974d162f
(2) 0dbbe8e4ae425a6d2687f1a7e3ba17bc98c673636790f1b8ad91193c05875ef1
(3) c88b703fb08cbea894b6aeff5a544fb92e78a18e19814cd85da83b71f772aa6c
(4) 388c684f0ba1ef5017716adb5d21a053ea8e90277d0868337519f97bede61418
(5) 659cbb0e2411a44db63778987b1e22153c086a95eb6b18bdf89de078917abc63
(6) 82d052c865f5763aad42add438569276c00d3d88a2d062d36b2bae914d58b8c8
(7) aa3680d5d48a8283413f7a108367c7299ca73f553735860a87b08f39395618b7
(8) 0f62d96d6675f32685bbdb8ac13cda7c23436f63efbb9d07700d8669ff12b7c4
(9) 8d5366123cb560bb606379f90a0bfd4769eecc0557f1b362dcae9012b548b1e5

Mnemonic: candy maple cake sugar pudding cream honey rich smooth crumble sweet treat

⚠️  Important ⚠️  : This mnemonic was created for you by Truffle. It is not secure.
Ensure you do not use it on production blockchains, or else you risk losing funds.

truffle(develop)>
```

通过这个文件，我们指定了 Migrate 将要进行的部署操作（也就是把我们的控制总线进行部署，由于其继承了所有的子系统，其他的子系统也会被一并部署）。

进行部署

在 `truffle develop` 的命令行下，直接输入 `migrate` 即可开始进行部署：

```
> migrate
Using network 'develop'.

Running migration: 1_initial_migration.js
  Replacing Migrations...
  ... 0x0a4bbadf4ec6bc785abd67a1d7ea3b78a79fcb44041af2ec1f805d3c9c976b11
  Migrations: 0x8cdaf0cd259887258bc13a92c0a6da92698644c0
Saving successful migration to network...
  ... 0xd7bc86d31bee32fa3988f1c1eabce403a1b5d570340a3a9cdba53a472ee8c956
Saving artifacts...
Running migration: 2_deploy_control_bus.js
  Replacing ControlBus...
  ... 0x5361792e402fefa206840aeca789e5d53df028e34752a608149d9684b115ede5
  ControlBus: 0x345ca3e014aaf5dca488057592ee47305d9b3e10
Saving successful migration to network...
  ... 0xf36163615f41ef7ed8f4a8f192149a0bf633fe1a2398ce001bf44c43dc7bdda0
Saving artifacts...
```

部署完成了，可是我们要怎样访问我们的合约呢？我们可以通过如下命令为我们的合约创造一个实例引用：

```
> ControlBus.deployed().then(instance => {controlBusInstance = instance})
undefined
```

这个实例引用被保存在 `controlBusInstance` 中，直接输入该变量名就可以将其打印出来……只是（由于我们的合约比较复杂）有些长~下面我们就可以进行测试了。

开始测试

创建用户

测试内容：用两个账户分别创建一个作者账户和一个出版商账户，并通过获取其对应的 Index 来进行验证。

测试命令：

```
> controlBusInstance.createWriter({from: web3.eth.accounts[1]})
> controlBusInstance.getWriterIndex(web3.eth.accounts[1])
> controlBusInstance.createPublisher({from: web3.eth.accounts[2]})
> controlBusInstance.getPublisherIndex(web3.eth.accounts[2])
```

输出结果：

```
truffle(develop)> controlBusInstance.createWriter({from: web3.eth.accounts[1]})
{ tx:
   '0x22221a6bc549449aff8f4509d61a2f86fd6470766fb71bf9a2e12e990b48b481',
  receipt:
   { transactionHash:
      '0x22221a6bc549449aff8f4509d61a2f86fd6470766fb71bf9a2e12e990b48b481',
     transactionIndex: 0,
     blockHash:
      '0x91ae23b2d4afa7ec224329ed5cccc0473e0d71508140496bfc9554d9481f07ea',
     blockNumber: 5,
     gasUsed: 108340,
     cumulativeGasUsed: 108340,
     contractAddress: null,
     logs: [],
     status: '0x01',
     logsBloom:
      '0x00000000000000000000000000000000000000000000000000000000000000000
  logs: [] }
truffle(develop)> controlBusInstance.getWriterIndex(web3.eth.accounts[1])
BigNumber { s: 1, e: 0, c: [ 1 ] }
truffle(develop)> controlBusInstance.createPublisher({from: web3.eth.accounts[2]})
{ tx:
   '0x27157f3901f5c07b78ee03c63fb25af8d833f8f26b86b45e8e04aa5b48fdeb0a',
  receipt:
   { transactionHash:
      '0x27157f3901f5c07b78ee03c63fb25af8d833f8f26b86b45e8e04aa5b48fdeb0a',
     transactionIndex: 0,
     blockHash:
      '0x28e3302f7b135f0e6e7234ecc0b58ca26a07ea4558a0e6a17e193d21ea2ed7f2',
     blockNumber: 6,
     gasUsed: 108626,
     cumulativeGasUsed: 108626,
     contractAddress: null,
     logs: [],
     status: '0x01',
     logsBloom:
      '0x00000000000000000000000000000000000000000000000000000000000000000
  logs: [] }
truffle(develop)> controlBusInstance.getPublisherIndex(web3.eth.accounts[2])
BigNumber { s: 1, e: 0, c: [ 1 ] }
```

可以看到，我们成功创建了这两个账户，并且能够通过地址获取到他们在 mapping 中的 Index。

撰写文章和查看文章

下面我们尝试通过作者账户撰写和查看文章。测试用的文章如下：

```
This is an text article. I hope no exception will be thrown!
```

其中，第一句话为明文首部。测试命令如下：

```
> var testArticle = "This is an text article. I hope no exception will be thrown!"
> var headContent = "This is an text article."
> controlBusInstance.writeArticle(headContent, web3.sha3(testArticle), testArticle.length, {from: web3.eth.
```

输出结果：

```
{ tx:
   '0x64ccd6edd9d5a8bc26ff92af742242b7733b3cd2a77353fd249848e390f9b7ea',
  receipt:
   { transactionHash:
      '0x64ccd6edd9d5a8bc26ff92af742242b7733b3cd2a77353fd249848e390f9b7ea',
     transactionIndex: 0,
     blockHash:
      '0xcde58712bc52e5e771715cf33e0e6838698876e2f065b1243ab1f48be3b64663',
     blockNumber: 7,
     gasUsed: 221879,
     cumulativeGasUsed: 221879,
     contractAddress: null,
     logs: [],
     status: '0x01',
     logsBloom:
      '0x00000000000000000000000000000000000000000000000000000000000000000
  logs: [] }
```

然后，我们尝试访问和验证文章，两次测试代码如下：

```
// Access
> var articleIndex
> controlBusInstance.getWriterCreatedArticleAt(1,0).then((result)=>{articleIndex = result})
> articleIndex = articleIndex.c[0]
> controlBusInstance.getArticle(articleIndex)
// Verify
> controlBusInstance.verifyArticle(articleIndex, testArticle, testArticle.length)
> controlBusInstance.verifyArticle(articleIndex, "This is a fake article.", testArticle.length)
> controlBusInstance.verifyArticle(articleIndex, testArticle, 5)
```

输出结果：

```
'This is an text article.'
true
false
```

测试结果符合预期。

创建订单和查看订单

下面我们尝试用出版商账户创建一个给作者账户的订单并且查看订单信息。测试前，我们首先验证一下各个账户内的以太币数量：

```
truffle(develop)> web3.eth.getBalance(web3.eth.accounts[1])
BigNumber { s: 1, e: 19, c: [ 999669, 78100000000000 ] }
truffle(develop)> web3.eth.getBalance(web3.eth.accounts[2])
BigNumber { s: 1, e: 19, c: [ 999891, 37400000000000 ] }
```

我们这里让 Publisher 给 Writer 提交一个 5 以太币的交易吧！

测试代码如下：

```
> var price = web3.toWei(5, "ether")
> controlBusInstance.createOrder(web3.eth.accounts[1], {from: web3.eth.accounts[2], value: price})
```

然后我们检查一下付款账户的余额：

```
truffle(develop)> web3.eth.getBalance(web3.eth.accounts[2])
BigNumber { s: 1, e: 19, c: [ 949729, 8200000000000 ] }
```

5个以太币已经被顺利扣掉了。接下来我们尝试从作者方获取我们接到的订单：

```
> var orderIndex
> controlBusInstance.getWriterOrderAt(1,0).then((result)=>{orderIndex = result.c[0]})
> controlBusInstance.getOrder(orderIndex)
[ '0xf17f52151ebef6c7334fad080c5704d77216b732',
  '0xc5fdf4076b8f3a5357c5e395ab970b5b54098fef',
  BigNumber { s: 1, e: 18, c: [ 50000 ] },
  false ]
```

可以看到，订单正确地记录了双方的地址和交易金额，并且订单当前的状态为未完成，符合实验预期。

完成订单及收款情况

接下来，我们让读者完成这个订单然后检查他的订单和余额变化情况，另外我们也需要检查出版商是否拿到了我们的文章版权：

```
> controlBusInstance.finishOrder(0, orderIndex, {from:web3.eth.accounts[1]})
> controlBusInstance.getOrder(orderIndex)
> web3.eth.getBalance(web3.eth.accounts[1])
> controlBusInstance.getPublisherOwnedArticleCount(1)
> controlBusInstance.getPublisherOwnedArticleAt(1, 0)
```

输出结果：

```
[ '0xf17f52151ebef6c7334fad080c5704d77216b732',
  '0xc5fdf4076b8f3a5357c5e395ab970b5b54098fef',
  BigNumber { s: 1, e: 18, c: [ 50000 ] },
  true ]
BigNumber { s: 1, e: 20, c: [ 1049571, 5300000000000 ] }
BigNumber { s: 1, e: 0, c: [ 1 ] }
[ BigNumber { s: 1, e: 0, c: [ 0 ] }, true ]
```

订单状态被顺利更新，作者也成功拿到了钱，出版商也拿到了文章的版权。结果符合预期。

发送邮件和查看邮件

接下来我们测试邮件系统，这包含发送和查看邮件。那么就让我们的作者给出版商写一封感谢信吧：

```
> controlBusInstance.writeMail(web3.eth.accounts[2], "Thank you!", {from:web3.eth.accounts[1]})
> var mailIndex
> controlBusInstance.getPublisherMailAt(1, 0).then((result)=>{mailIndex = result.c[0]})
> controlBusInstance.getMailContentByIndex(mailIndex, {from: web3.eth.accounts[2]})
```

测试结果：

```
'Thank you!'
```

测试结果符合预期。所有测试均完成，整个系统运行正常。