智能合约处理

受众：架构师、应用程序和智能合约开发人员

区块链网络的核心是智能合约。在 PaperNet 中，商业票据智能合约中的代码定义了商业票据的有效状态，以及将票据从一种状态状态转变为另一种状态的交易逻辑。在本主题中，我们将向您展示如何实现一个真实世界的智能合约，该合约管理发行、购买和兑换商业票据的过程。

我们将会介绍:

- 什么是智能合约以及智能合约为什么重要
- 如何定义智能合约
- 如何定义交易
- 如何实现一笔交易
- 如何在智能合约中表示业务对象
- 如何在账本中存储和检索对象

如果您愿意，可以下载示例，甚至可以在本地运行。它是用 JavaScript 和 Java 编写的，但逻辑与语言无关，因此您可以轻松地查看正在发生的事情！（该示例也可用于 Go。）

智能合约

智能合约定义业务对象的不同状态，并管理对象在不同状态之间变化的过程。智能合约很重要，因为它们允许架构师和智能合约开发人员定义在区块链网络中协作的不同组织之间共享的关键业务流程和数据。

在 PaperNet 网络中，智能合约由不同的网络参与者共享，例如 MagnetoCorp 和 DigiBank。 连接到网络的所有应用程序必须使用相同版本的智能合约，以便它们共同实现相同的共享业务流程和数据。

实现语言

支持两种运行时，Java 虚拟机和 Node.js。支持使用 JavaScript、TypeScript、Java 或其他可以运行在支持的运行时上其中一种语言。

在 Java 和 TypeScript 中，标注或者装饰器用来为智能合约和它的结构提供信息。这就更加丰富了开发体验——比如，作者信息或者强调返回类型。使用 JavaScript 的话就必须遵守一些规范，同时，对于什么可以自动执行也有一些限制。

这里给出的示例包括 JavaScript 和 Java 两种语言。

合约类

PaperNet 商业票据智能合约的副本包含在单个文件中。如果您已下载，请使用浏览器或在您喜欢的编辑器中打开它。

- `papercontract.js` - JavaScript 版本
- `CommercialPaperContract.java` - Java 版本

您可能会从文件路径中注意到这是 MagnetoCorp 的智能合约副本。 MagnetoCorp 和 DigiBank 必须同意他们将要使用的智能合约版本。现在，你看哪个组织的合约副本无关紧要，它们都是一样的。

花一些时间看一下智能合约的整体结构; 注意，它很短！在文件的顶部，您将看到商业票据智能合约的定义：

▼ JavaScript

```
class CommercialPaperContract extends Contract {...}
```

▶ Java

`CommercialPaperContract` 类中包含商业票据中交易的定义——**发行**，**购买**和**兑换**。这些交易带给了商业票据创建和在它们的生命周期中流动的能力。我们马上会查看这些交易，但是现在我们需要关注一下 JavaScript， `CommericalPaperContract` 扩展的 Hyperledger Fabric `Contract` 类。

在 Java 中，类必须使用 `@Contract(...)` 标注进行包装。它支持额外的智能合约信息，比如许可和作者。 `@Default()` 标注表明该智能合约是默认合约类。在智能合约中标记默认合约类在一些有多个合约类的智能合约中会很有用。

如果你使用 TypeScript 实现，也有类似 `@Contract(...)` 的标注，和 Java 中功能相似。

关于可用的标注的更多信息，请查看 API 文档：

- Java 智能合约 API 文档
- Node.js 智能合约 API 文档

我们先导入这些类、标注和 `Context` 类：

▼ JavaScript

```
const { Contract, Context } = require('fabric-contract-api');
```

▶ Java

我们的商业票据合约将使用这些类的内置功能，例如自动方法调用，每个交易上下文，交易处理器，和类共享状态。

还要注意 JavaScript 类构造函数如何使用其超类通过一个命名空间来初始化自身：

```
constructor() {
    super('org.papernet.commercialpaper');
}
```

在 Java 类中，构造器是空的，合约名会通过 `@Contract()` 注解进行识别。如果不是就会使用类名。

最重要的是，`org.papernet.commercialpaper` 非常具有描述性——这份智能合约是所有 PaperNet 组织关于商业票据商定的定义。

通常每个文件只有一个智能合约（合约往往有不同的生命周期，这使得将它们分开是明智的）。但是，在某些情况下，多个智能合约可能会为应用程序提供语法帮助，例如 `EuroBond`、`DollarBond`、`YenBond` 但基本上提供相同的功能。在这种情况下，智能合约和交易可以消除歧义。

交易定义

在类中定位 **issue** 方法。

▼ JavaScript

```
async issue(ctx, issuer, paperNumber, issueDateTime, maturityDateTime, faceValue) {...}
```

▶ Java

Java 标注 `@Transaction` 用于标记该方法为交易定义；TypeScript 中也有等价的标注。

无论何时调用此合约来 `发行` 商业票据，都会调用该方法。回想一下如何使用以下交易创建商业票据 00001：

```
Txn = issue
Issuer = MagnetoCorp
Paper = 00001
Issue time = 31 May 2020 09:00:00 EST
Maturity date = 30 November 2020
Face value = 5M USD
```

我们已经更改了编程样式的变量名称，但是看看这些属性几乎直接映射到 `issue` 方法变量。

只要应用程序请求发行商业票据，合约就会自动调用 `issue` 方法。交易属性值通过相应的变量提供给方法。使用示例应用程序，了解应用程序如何使用应用程序主题中的 Hyperledger Fabric SDK 提交一笔交易。

您可能已经注意到 **issue** 方法中定义的一个额外变量 ctx。它被称为交易上下文，它始终是第一个参数。默认情况下，它维护与交易逻辑相关的每个合约和每个交易的信息。例如，它将包含 MagnetoCorp 指定的交易标识符，MagnetoCorp 可以发行用户的数字证书，也可以调用账本 API。

通过实现自己的 `createContext()` 方法而不是接受默认实现，了解智能合约如何扩展默认交易上下文：

▼ JavaScript

```
createContext() {
  return new CommercialPaperContext()
}
```

▶ Java

此扩展上下文将自定义属性 `paperList` 添加到默认值：

▼ JavaScript

```
class CommercialPaperContext extends Context {

  constructor() {
    super();
    // All papers are held in a list of papers
    this.paperList = new PaperList(this);
}
```

▶ Java

我们很快就会看到 `ctx.paperList` 如何随后用于帮助存储和检索所有 PaperNet 商业票据。

为了巩固您对智能合约交易结构的理解，找到**购买**和**兑换**交易定义，看看您是否可以理解它们如何映射到相应的商业票据交易。

购买交易：

```
Txn = buy
Issuer = MagnetoCorp
Paper = 00001
Current owner = MagnetoCorp
New owner = DigiBank
Purchase time = 31 May 2020 10:00:00 EST
Price = 4.94M USD
```

▼ JavaScript

```
async buy(ctx, issuer, paperNumber, currentOwner, newOwner, price, purchaseTime) {...}
```

▶ Java

兑换交易：

```
Txn = redeem
Issuer = MagnetoCorp
Paper = 00001
Redeemer = DigiBank
Redeem time = 31 Dec 2020 12:00:00 EST
```

▼ JavaScript

```
async redeem(ctx, issuer, paperNumber, redeemingOwner, redeemDateTime) {...}
```

▶ Java

在两个案例中，注意商业票据交易和智能合约方法调用之间 1：1 的关系。

所有 JavaScript 方法都使用 `async` 和 `await` 关键字。

交易逻辑

现在您已经了解了合约的结构和交易的定义，下面让我们关注智能合约中的逻辑。

回想一下第一个**发行**交易：

```
Txn = issue
Issuer = MagnetoCorp
Paper = 00001
Issue time = 31 May 2020 09:00:00 EST
Maturity date = 30 November 2020
Face value = 5M USD
```

它导致 **issue** 方法被传递调用：

▼ JavaScript

```
async issue(ctx, issuer, paperNumber, issueDateTime, maturityDateTime, faceValue) {

   // create an instance of the paper
  let paper = CommercialPaper.createInstance(issuer, paperNumber, issueDateTime, maturityDateTime

  // Smart contract, rather than paper, moves paper into ISSUED state
  paper.setIssued();

  // Newly issued paper is owned by the issuer
  paper.setOwner(issuer);

  // Add the paper to the list of all similar commercial papers in the ledger world state
  await ctx.paperList.addPaper(paper);

  // Must return a serialized paper to caller of smart contract
  return paper.toBuffer();
}
```

▶ Java

逻辑很简单：获取交易输入变量，创建新的商业票据 `paper`，使用 `paperList` 将其添加到所有商业票据的列表中，并将新的商业票据（序列化为buffer）作为交易响应返回。

了解如何从交易上下文中检索 `paperList` 以提供对商业票据列表的访问。`issue()`、`buy()` 和 `redeem()` 不断重新访问 `ctx.paperList` 以使商业票据列表保持最新。

购买交易的逻辑更详细描述：

▼ JavaScript

```
async buy(ctx, issuer, paperNumber, currentOwner, newOwner, price, purchaseDateTime) {

  // Retrieve the current paper using key fields provided
  let paperKey = CommercialPaper.makeKey([issuer, paperNumber]);
  let paper = await ctx.paperList.getPaper(paperKey);

  // Validate current owner
  if (paper.getOwner() !== currentOwner) {
      throw new Error('Paper ' + issuer + paperNumber + ' is not owned by ' + currentOwner);
  }

  // First buy moves state from ISSUED to TRADING
  if (paper.isIssued()) {
      paper.setTrading();
  }

  // Check paper is not already REDEEMED
  if (paper.isTrading()) {
      paper.setOwner(newOwner);
  } else {
      throw new Error('Paper ' + issuer + paperNumber + ' is not trading. Current state = ' +pape
  }

  // Update the paper
  await ctx.paperList.updatePaper(paper);
  return paper.toBuffer();
}
```

▶ Java

在使用 `paper.setOwner(newOwner)` 更改拥有者之前，理解交易如何检查 `currentOwner` 并检查该 `paper` 应该是 `TRADING` 状态的。基本流程很简单：检查一些前提条件，设置新拥有者，更新账本上的商业票据，并将更新的商业票据（序列化为 buffer ）作为交易响应返回。

为什么不看一下你否能理解**兑换**交易的逻辑?

对象的表示

我们已经了解了如何使用 `CommercialPaper` 和 `PaperList` 类定义和实现**发行**、**购买**和**兑换**交易。让我们通过查看这些类如何工作来结束这个主题。

定位到 `CommercialPaper` 类：

▼ JavaScript
In the paper.js file:

```
class CommercialPaper extends State {...}
```

▶ Java

该类包含商业票据状态的内存表示。了解 `createInstance` 方法如何使用提供的参数初始化一个新的商业票据：

▼ JavaScript

```
static createInstance(issuer, paperNumber, issueDateTime, maturityDateTime, faceValue) {
  return new CommercialPaper({ issuer, paperNumber, issueDateTime, maturityDateTime, faceValue })
}
```

▶ Java

回想一下**发行**交易如何使用这个类：

▼ JavaScript

```
let paper = CommercialPaper.createInstance(issuer, paperNumber, issueDateTime, maturityDateTime,
```

▶ Java

查看每次调用发行交易时，如何创建包含交易数据的商业票据的新内存实例。

需要注意的几个要点：

- 这是一个内存中的表示; 我们稍后会看到它如何在帐本上显示。
- `CommercialPaper` 类扩展了 `State` 类。 `State` 是一个应用程序定义的类，它为状态创建一个公共抽象。所有状态都有一个它们代表的业务对象类、一个复合键，可以被序列化和反序列化，等等。当我们在帐本上存储多个业务对象类型时， `State` 可以帮助我们的代码更清晰。检查 `state.js` 文件中的 `State` 类。
- 票据在创建时会计算自己的密钥，在访问帐本时将使用此密钥。密钥由 `issuer` 和 `paperNumber` 的组合形成。

  ```
  constructor(obj) {
    super(CommercialPaper.getClass(), [obj.issuer, obj.paperNumber]);
    Object.assign(this, obj);
  }
  ```

- 票据通过交易而不是票据类变更到 `ISSUED` 状态。那是因为智能合约控制票据的状态生命周期。例如， `import` 交易可能会立即创建一组新的 `TRADING` 状态的票据。

`CommercialPaper` 类的其余部分包含简单的辅助方法：

```
getOwner() {
    return this.owner;
}
```

回想一下智能合约如何使用这样的方法来维护商业票据的整个生命周期。例如，在**兑换**交易中，我们看到：

```
if (paper.getOwner() === redeemingOwner) {
  paper.setOwner(paper.getIssuer());
  paper.setRedeemed();
}
```

访问账本

现在在 `paperlist.js` 文件中找到 `PaperList` 类：

```
class PaperList extends StateList {
```

此工具类用于管理 Hyperledger Fabric 状态数据库中的所有 PaperNet 商业票据。PaperList 数据结构在架构主题中有更详细的描述。

与 `CommercialPaper` 类一样，此类扩展了应用程序定义的 `StateList` 类，该类为一系列状态创建了一个通用抽象——在本例中是 PaperNet 中的所有商业票据。

`addPaper()` 方法是对 `StateList.addState()` 方法的简单封装：

```
async addPaper(paper) {
  return this.addState(paper);
}
```

您可以在 `StateList.js` 文件中看到 `StateList` 类如何使用 Fabric API `putState()` 将商业票据作为状态数据写在帐本中：

```
async addState(state) {
  let key = this.ctx.stub.createCompositeKey(this.name, state.getSplitKey());
  let data = State.serialize(state);
  await this.ctx.stub.putState(key, data);
}
```

帐本中的每个状态数据都需要以下两个基本要素：

- **键（Key）**：`键` 由 `createCompositeKey()` 使用固定名称和 `state` 密钥形成。在构造 `PaperList` 对象时分配了名称，`state.getSplitKey()` 确定每个状态的唯一键。
- **数据（Data）**：`数据` 只是商业票据状态的序列化形式，使用 `State.serialize()` 方法创建。`State` 类使用 JSON 对数据进行序列化和反序列化，并根据需要使用 State 的业务对象类，在我们的例子中为 `CommercialPaper`，在构造 `PaperList` 对象时再次设置。

注意 `StateList` 不存储有关单个状态或状态总列表的任何内容——它将所有这些状态委托给 Fabric 状态数据库。这是一个重要的设计模式 – 它减少了 Hyperledger Fabric 中账本 MVCC 冲突的机会。

StateList `getState()` 和 `updateState()` 方法以类似的方式工作：

```
async getState(key) {
  let ledgerKey = this.ctx.stub.createCompositeKey(this.name, State.splitKey(key));
  let data = await this.ctx.stub.getState(ledgerKey);
  let state = State.deserialize(data, this.supportedClasses);
  return state;
}
```

```
async updateState(state) {
  let key = this.ctx.stub.createCompositeKey(this.name, state.getSplitKey());
  let data = State.serialize(state);
  await this.ctx.stub.putState(key, data);
}
```

了解他们如何使用 Fabric APIs `putState()` 、 `getState()` 和 `createCompositeKey()` 来存取账本。我们稍后将扩展这份智能合约，以列出 paperNet 中的所有商业票据。实现账本检索的方法可能是什么样的?

是的！在本主题中，您已了解如何为 PaperNet 实现智能合约。您可以转到下一个子主题，以查看应用程序如何使用 Fabric SDK 调用智能合约。