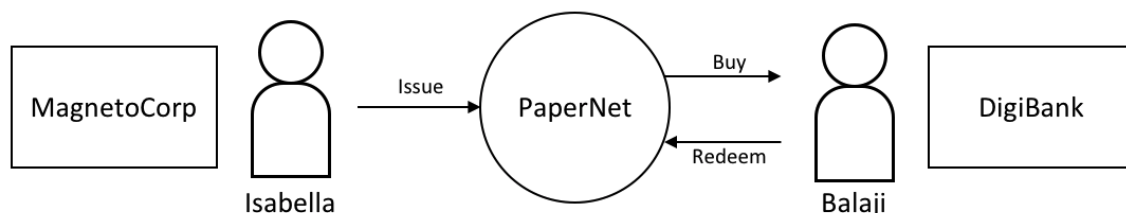


商业票据教程

受众: 架构师, 应用和智能合约开发者, 管理员

本教程将向您展示如何安装和使用商业票据样例应用程序和智能合约。该主题是以任务为导向的, 因此它更侧重的是流程而不是概念。如果您想更深入地了解这些概念, 可以阅读[开发应用程序](#)主题。



在本教程中, *MagnetoCorp* 和 *DigiBank* 这两个组织使用 *Hyperledger Fabric* 区块链网络 *PaperNet* 相互交易商业票据。

一旦建立了一个基本的网络, 您就将扮演 *MagnetoCorp* 的员工 *Isabella*, 她将代表公司发行商业票据。然后, 您将转换角色, 担任 *DigiBank* 的员工 *Balaji*, 他将购买此商业票据, 持有一段时间, 然后向 *MagnetoCorp* 赎回该商业票据, 以获取小额利润。

您将扮演开发人员, 最终用户和管理员, 这些角色位于不同组织中, 都将执行以下步骤, 这些步骤旨在帮助您了解作为两个不同组织独立工作, 但要根据 *Hyperledger Fabric* 网络中双方共同商定的规则来进行协作是什么感觉。

- [环境配置](#) 和 [下载示例](#)
- [创建网络](#)
- [检查商业票据智能合约](#)
- 通过以 *MagnetoCorp* 和 *MagnetoCorp* 组织的身份同意链码定义, [将智能合约部署到通道](#)。
- 理解一个 *MagnetoCorp* [应用程序](#) 的结构, 包括它的 [依赖](#)
- 配置和使用 [钱包和身份](#)
- 运行一个 *MagnetoCorp* 应用程序来 [发行商业票据](#)
- 理解 *DigiBank* 是如何在 [应用程序](#) 中使用智能合约的
- 以 *Digibank* 组织的身份运行应用程序来 [购买](#) 和 [赎回](#) 商业票据

本教程已经在 *MacOS* 和 *Ubuntu* 上进行了测试, 应该可以在其他 *Linux* 发行版上运行。*Windows* 版本的教程正在开发中。

准备阶段

在开始之前, 您必须安装本教程所需的一些必备工具。我们将必备工具控制在最低限度, 以便您能快速开始。

您必须安装以下软件：

- **Node**， Node.js SDK的README文档包含准备阶段中 对应最新版本的清单。

您会发现安装以下软件很有帮助：

- 源码编辑器，如 **Visual Studio Code** 版本 1.28， 或更高版本。VS Code 将会帮助您开发和测试您的应用程序和智能合约。安装 VS Code 看[这里](#)。

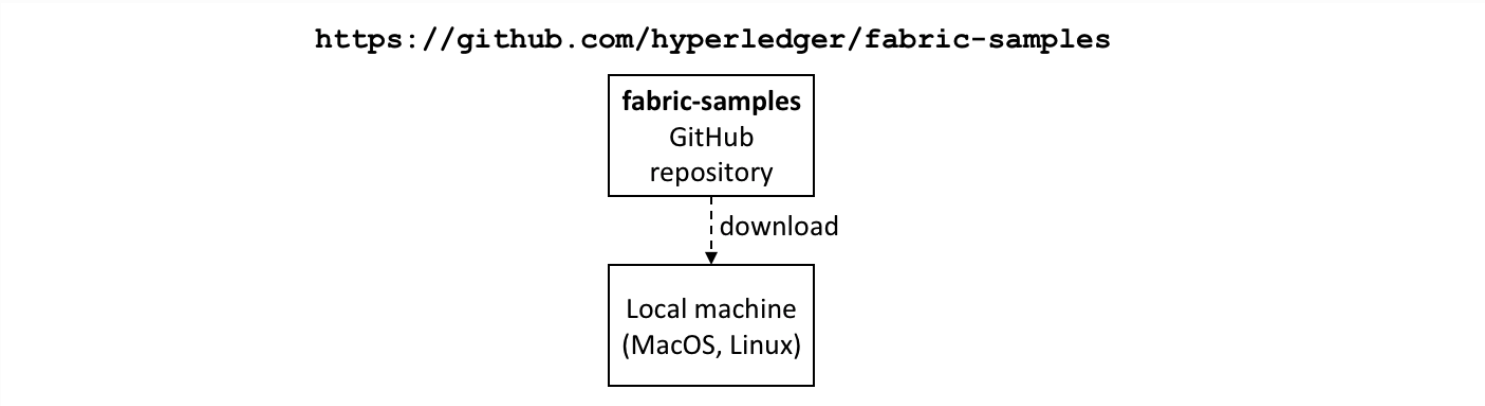
许多优秀的代码编辑器都可以使用， 包括 **Atom**, **Sublime Text** 和 **Brackets**。

您可能会发现，随着您在应用程序和智能合约开发方面的经验越来越丰富， 安装以下软件会很有帮助。首次运行教程时无需安装这些：

- **Node Version Manager**。NVM 帮助您轻松切换不同版本的 node——如果您同时处理多个项目的话， 那将非常有用。安装 NVM 看[这里](#)。

下载示例

商业票据教程是 `fabric-samples` 仓库中的示例之一。 在开始本教程之前， 确保您已经按照说明安装了 Fabric [准备阶段必备工具](#) 并[下载示例、二进制文件和Docker映像](#)。 当您完成后， 您便克隆了包含教程脚本、 智能合约和应用程序文件的 `fabric-samples` 仓库。



下载 `fabric-samples` GitHub 仓库到您的本地机器。

下载后， 可随意查看 `fabric-samples` 的目录结构：

```
$ cd fabric-samples
$ ls

CODEOWNERS          SECURITY.md           first-network
CODE_OF_CONDUCT.md  chaincode            high-throughput
CONTRIBUTING.md     chaincode-docker-devmode  interest_rate_swaps
LICENSE              ci                    off_chain_data
MAINTAINERS.md       commercial-paper       test-network
README.md            fabcar
```

注意 `commercial-paper` 目录， 我们的示例就在这里！

现在您已经完成了教程的第一个阶段！随着您继续操作，您将为不同用户和组件打开多个命令窗口。例如：

- 为了显示来自您的网络的peer节点、排序服务节点和CA日志输出。
- 作为Magnetocorp组织和DigiBank组织的管理员， 审批同意链码。
- 代表Isabella和Balaji运行应用程序， 他们将使用智能合约彼此交易商业票据。

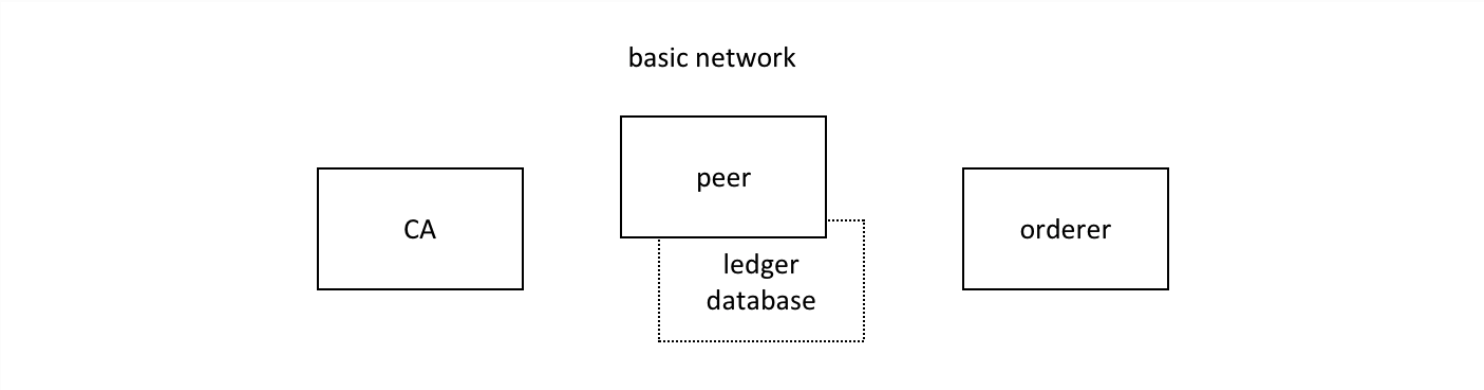
当您应该从特定命令窗口运行一项命令时，我们将详细说明这一点。例如：

```
(isabella)$ ls
```

这表示您应该在 Isabella 的窗口中执行 `ls` 命令。

创建网络

本教程将使用Fabric测试网络部署智能合约。测试网络由两个peer组织和一个排序服务组织组成。两个peer组织各自操作一个peer节点， 而排序服务组织操作单个Raft排序服务节点。我们还将使用测试网络创建一个名为 `mychannel` 的单一通道， 两个peer组织都将是该通道的成员。



The Hyperledger Fabric 基础网络的组成部分包括一个节点及该节点的账本数据库， 一个排序服务和一个证书授权中心。以上每个组件都在一个 Docker 容器中运行。

每个组织运行自己的证书颁发机构。两个peer节点、**状态数据库**、排序服务节点 和每个组织CA都在各自的Docker容器中运行。 在生产环境中， 组织通常使用与其他系统共享的现有CA; 它们不是专用于Fabric网络的。

测试网络的两个组织允许我们以两个独立的组织形式， 操作不同的peer节点与一个区块链帐本交互。 在本教程中， 我们将作为DigiBank操作测试网络中的Org1 以及作为Magnetocorp操作Org2。

您可以启动测试网络， 并使用商业票据目录中提供的脚本创建通道。 进入到 `fabric-samples` 中的 `commercial-paper` 目录：

```
cd fabric-samples/commercial-paper
```

然后使用脚本启动测试网络：

```
./network-starter.sh
```

当脚本运行时，您将看到部署测试网络的日志。当脚本完成后，您可以使用 `docker ps` 命令查看Fabric节点在您的本地机器上的运行情况：

```
$ docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED
a86f50ca1907	hyperledger/fabric-peer:latest	"peer node start"	About a minute a
77d0fcaee61b	hyperledger/fabric-peer:latest	"peer node start"	About a minute a
7eb5f64bfe5f	hyperledger/fabric-couchdb	"tini -- /docker-ent..."	About a minute a
2438df719f57	hyperledger/fabric-couchdb	"tini -- /docker-ent..."	About a minute a
03373d116c5a	hyperledger/fabric-orderer:latest	"orderer"	About a minute a
6b4d87f65909	hyperledger/fabric-ca:latest	"sh -c 'fabric-ca-se..."	About a minute a
7b01f5454832	hyperledger/fabric-ca:latest	"sh -c 'fabric-ca-se..."	About a minute a
87aef6062f23	hyperledger/fabric-ca:latest	"sh -c 'fabric-ca-se..."	About a minute a

看看您是否可以将这些容器映射到基本网络上(可能需要横向移动才能找到信息)：

- Org1的peer, `peer0.org1.example.com` , 在容器 `a86f50ca1907` 上运行
- Org2的peer, `peer0.org2.example.com` , 在容器 `77d0fcaee61b` 上运行
- Org1的peer对应的CouchDB数据库, `couchdb0` , 在容器 `7eb5f64bfe5f` 上运行
- Org2的peer对应的CouchDB数据库, `couchdb1` , 在容器 `2438df719f57` 上运行
- 排序服务节点, `orderer.example.com` , 在容器 `03373d116c5a` 上运行
- Org1的CA, `ca_org1` , 在容器 `87aef6062f23` 上运行
- Org2的CA, `ca_org2` , 在容器 `6b4d87f65909` 上运行
- 排序服务Org的CA, `ca_orderer` , 在容器 `7b01f5454832` 上运行

所有这些容器构成了被称作 `net_test` 的 `docker` 网络。您可以使用 `docker network` 命令查看该网络：

```
$ docker network inspect net_test
```

```
[
  {
    "Name": "net_test",
    "Id": "f4c9712139311004b8f7acc14e9f90170c5dcfd8cdd06303c7b074624b44dc9f",
    "Created": "2020-04-28T22:45:38.525016Z",
    "Containers": {
      "03373d116c5abf2ca94f6f00df98bb74f89037f511d6490de4a217ed8b6fbcd0": {
        "Name": "orderer.example.com",
        "EndpointID": "0eed871a2aaf9a5dbcf7896aa3c0f53cc61f57b3417d36c56747033fd9f81972",
        "MacAddress": "02:42:c0:a8:70:05",
        "IPv4Address": "192.168.112.5/20",
        "IPv6Address": ""
      },
      "2438df719f57a597de592cfc76db30013adfdcfa0cec5b375f6b7259f67baff8": {
        "Name": "couchdb1",
        "EndpointID": "52527fb450a7c80ea509cb571d18e2196a95c630d0f41913de8ed5abbd68993d",
        "MacAddress": "02:42:c0:a8:70:06",
        "IPv4Address": "192.168.112.6/20",
        "IPv6Address": ""
      },
      "6b4d87f65909afd335d7acfe6d79308d6e4b27441b25a829379516e4c7335b88": {
        "Name": "ca_org2",
        "EndpointID": "1cc322a995880d76e1dd1f37ddf9c43f86997156124d4ecbb0eba9f833218407",
        "MacAddress": "02:42:c0:a8:70:04",
        "IPv4Address": "192.168.112.4/20",
        "IPv6Address": ""
      },
      "77d0fcaee61b8fff43d3331073ab9ce36561a90370b9ef3f77c663c8434e642": {
        "Name": "peer0.org1.example.com",
```

```

    "EndpointID": "05d0d34569eee412e28313ba7ee06875a68408257dc47e64c0f4f5ef4a9dc491",
    "MacAddress": "02:42:c0:a8:70:08",
    "IPv4Address": "192.168.112.8/20",
    "IPv6Address": ""
  },
  "7b01f5454832984fcd9650f05b4affce97319f661710705e6381dfb76cd99fdb": {
    "Name": "ca_orderer",
    "EndpointID": "057390288a424f49d6e9d6f788049b1e18aa28bccd56d860b2be8ceb8173ef74",
    "MacAddress": "02:42:c0:a8:70:02",
    "IPv4Address": "192.168.112.2/20",
    "IPv6Address": ""
  },
  "7eb5f64bfe5f20701aae8a6660815c4e3a81c3834b71f9e59a62fb99bed1afc7": {
    "Name": "couchdb0",
    "EndpointID": "bfe740be15ec9dab7baf3806964e6b1f0b67032ce1b7ae26ac7844a1b422ddc4",
    "MacAddress": "02:42:c0:a8:70:07",
    "IPv4Address": "192.168.112.7/20",
    "IPv6Address": ""
  },
  "87aef6062f2324889074cda80fec8fe014d844e10085827f380a91eea4ccdd74": {
    "Name": "ca_org1",
    "EndpointID": "a740090d33ca94dd7c6aaf14a79e1cb35109b549ee291c80195beccc901b16b7",
    "MacAddress": "02:42:c0:a8:70:03",
    "IPv4Address": "192.168.112.3/20",
    "IPv6Address": ""
  },
  "a86f50ca19079f59552e8674932edd02f7f9af93ded14db3b4c404fd6b1abe9c": {
    "Name": "peer0.org2.example.com",
    "EndpointID": "6e56772b4783b1879a06f86901786fed1c307966b72475ce4631405ba8bca79a",
    "MacAddress": "02:42:c0:a8:70:09",
    "IPv4Address": "192.168.112.9/20",
    "IPv6Address": ""
  }
},
"Options": {},
"Labels": {}
}
]

```

看看这八个容器如何在作为单个 Docker 网络一部分的同时使用不同的 IP 地址。（为了清晰起见，我们对输出进行了缩写。）

由于我们是以DigiBank和MagnetoCorp的身份来操作测试网络的， `peer0.org1.example.com` 将属于DigiBank组织，而 `peer0.org2.example.com` 将由MagnetoCorp操作。现在测试网络已经启动并运行，从现在开始我们可以将我们的网络称为PaperNet。

回顾一下: 您已经从 GitHub 下载了 Hyperledger Fabric 示例仓库，并且已经在本地机器上运行了基本的网络。现在让我们开始扮演 MagnetoCorp 的角色来交易商业票据。

以 MagnetoCorp 的身份管理网络

商业票据教程允许您通过为DigiBank和MagnetoCorp 提供两个单独的文件夹来充当两个组织。这两个文件夹包含每个组织的智能合约和应用程序文件。由于这两个组织在商业票据交易中有不同的角色，所以每个组织的应用程序文件也不同。在 `fabric-samples` 仓库下打开一个新窗口，并使用以下命令切换到MagnetoCorp目录：

```
cd commercial-paper/organization/magnetocorp
```

我们要做的第一件事就是以 MagnetoCorp 的角色监控 PaperNet 网络中的组件。管理员可以使用 `logspout` 工具。该工具可以将不同输出流采集到一个地方，从而在一个窗口中就可以轻松看到正在发

生的事情。比如，对于正在安装智能合约的管理员或者正在调用智能合约的开发人员来说，这个工具确实很有帮助。

在Magnetocorp目录下，运行下列命令以运行 `monitordocker.sh` 脚本，并为运行在 `net_test` 上的与PaperNet相关联的容器启动 `logspout` 工具：

```
(magnetocorp admin)$ ./configuration/cli/monitordocker.sh net_test
...
latest: Pulling from gliderlabs/logspout
4fe2ade4980c: Pull complete
decca452f519: Pull complete
(...)
Starting monitoring on all containers on the network net_test
b7f3586e5d0233de5a454df369b8eadab0613886fc9877529587345fc01a3582
```

注意，如果 `monitordocker.sh` 中的默认端口已经在使用，您可以传入一个端口号。

```
(magnetocorp admin)$ ./monitordocker.sh net_test <port_number>
```

现在，这个窗口将为本教程剩余部分显示Docker容器的日志输出，那么继续并打开另一个命令窗口。我们要做的下一件事是检查Magnetocorp将用于发行商业票据的智能合约。

检查商业票据智能合约

`issue`，`buy` 和 `redeem` 是 PaperNet 智能合约的三个核心功能。应用程序使用这些功能来提交交易，相应地，在账本上会发行、购买和赎回商业票据。我们接下来的任务就是检查这个智能合约。

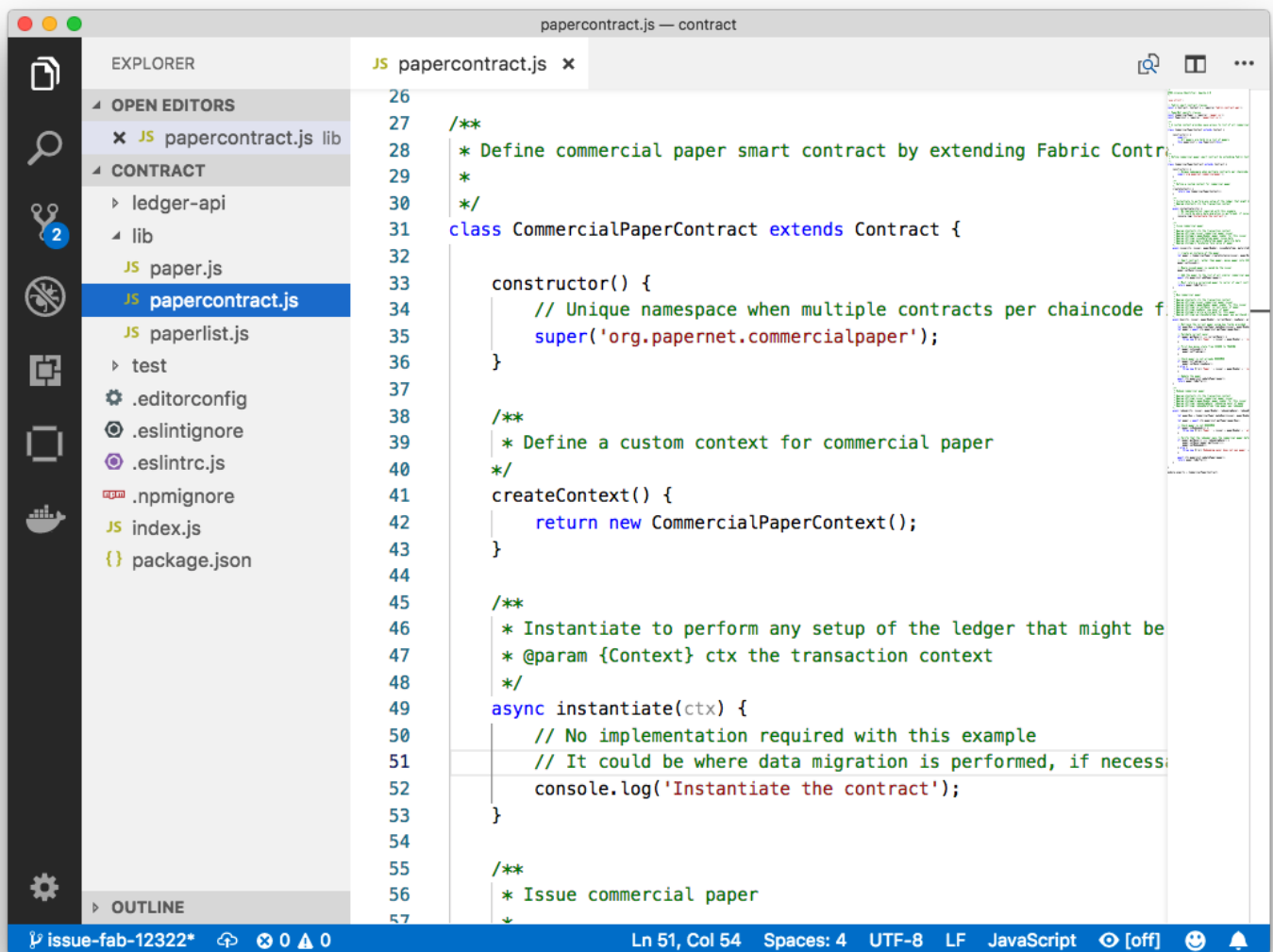
打开一个新的终端窗口来代表 Magnetocorp 开发人员。

```
cd commercial-paper/organization/magnetocorp
```

然后，您可以使用您选择的编辑器(本教程中的VS Code)在 `contract` 目录中查看智能合约：

```
(magnetocorp developer)$ code contract
```

在这个文件夹的 `lib` 目录下，您将看到 `papercontract.js` 文件，其中包含了商业票据智能合约！



一个示例代码编辑器在 `papercontract.js` 文件中展示商业票据智能合约

`papercontract.js` 是一个在 Node.js 环境中运行的 JavaScript 程序。注意下面的关键程序行：

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

这个语句引入了两个关键的 Hyperledger Fabric 类：`Contract` 和 `Context`，它们被智能合约广泛使用。您可以在 `fabric-shim` [JSDocs](#) 中了解到这些类的更多信息。

- `class CommercialPaperContract extends Contract {`

这里基于内置的 Fabric `Contract` 类定义了智能合约类 `CommercialPaperContract`。实现了 `issue`，`buy` 和 `redeem` 商业票据关键交易的方法被定义在该类中。

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

这个方法为 PaperNet 定义了商业票据 `issue` 交易。传入的参数用于创建新的商业票据。

找到并检查智能合约内的 `buy` 和 `redeem` 交易。

- `let paper = CommercialPaper.createInstance(issuer, paperNumber, issueDateTime...);`

在 `issue` 交易内部，这个语句根据提供的交易输入使用 `CommercialPaper` 类在内存中创建了一个新的商业票据。检查 `buy` 和 `redeem` 交易看如何类似地使用该类。

- `await ctx.paperList.addPaper(paper);`

这个语句使用 `ctx.paperList` 在账本上添加了新的商业票据，其中 `ctx.paperList` 是 `PaperList` 类的一个实例，当智能合约场景 `CommercialPaperContext` 被初始化时，就会创建出一个 `ctx.paperList`。再次检查 `buy` 和 `redeem` 方法，以了解这些方法是如何使用这一类的。

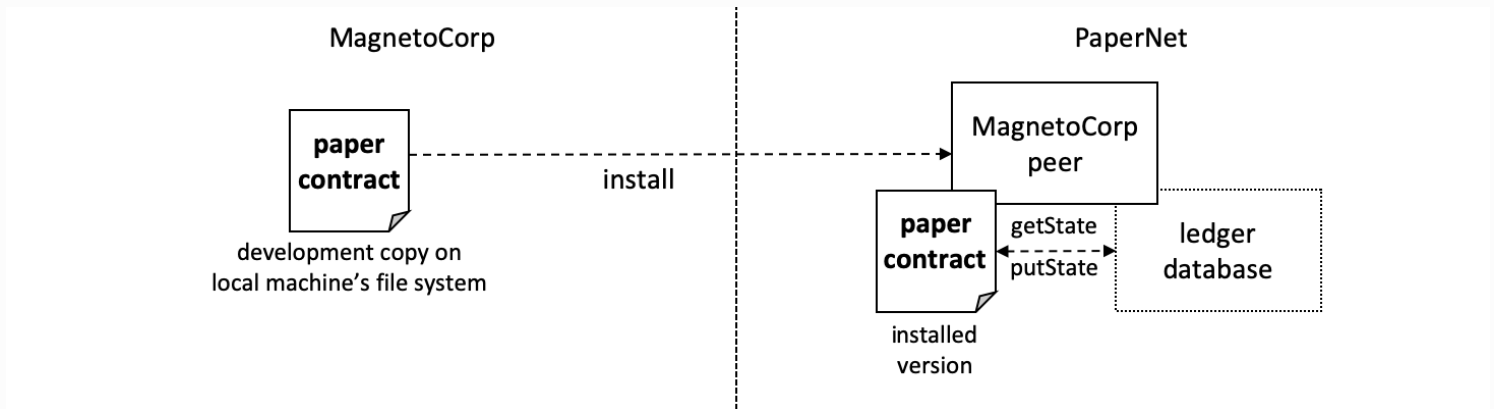
- `return paper;`

该语句返回一个二进制缓冲区，作为来自 `issue` 交易的响应，供智能合约的调用者处理。

欢迎检查 `contract` 目录下的其他文件来理解智能合约是如何工作的，请仔细阅读[智能合约处理](#)主题中 `papercontract.js` 是如何设计的。

将智能合约部署到通道

在应用程序调用 `papercontract` 之前，必须将它安装到测试网络中合适的peer节点上，然后在通道上使用 [Fabric链码生命周期](#) 定义它。Fabric链码生命周期允许多个组织在链码被部署到通道之前同意链码的参数。因此，我们需要以MagnetoCorp和DigiBank的管理员的身份来安装和审批同意链码。



MagnetoCorp 的管理员将 `papercontract` 的一个副本安装在 MagnetoCorp 的节点上。

智能合约是应用开发的重点，它被包含在一个名为[链码](#)的 Hyperledger Fabric 构件中。在一个链码中可以定义一个或多个智能合约，安装链码就使得 PaperNet 中的不同组织可以使用其中的智能合约。这意味着只有管理员需要关注链码；其他人都只需关注智能合约。

以 MagnetoCorp 的身份安装和批准智能合约

我们将首先以MagnetoCorp管理员的身份安装并同意智能合约。确保您正在 `magnetocorp` 文件夹里操作，或使用以下命令浏览至该文件夹：

```
cd commercial-paper/organization/magnetocorp
```

MagnetoCorp管理员可以通过使用 `peer` CLI与PaperNet交互。然而，管理员需要在其命令窗口中设置某些环境变量，以使用正确的 `peer` 二进制文件集，向MagnetoCorp的peer节点的地址发送命令和使用正确的加密资料对请求进行签名。

您可以使用示例提供的脚本在命令窗口中设置环境变量。在 `magnetocorp` 目录中执行如下命令：

```
source magnetocorp.sh
```


您将在窗口中看到被打印出来的环境变量的完整列表。我们现在可以使用这个命令窗口来以 MagnetoCorp 管理员的身份与 PaperNet 交互。

第一步是安装 `papercontract` 智能合约。可以使用 `peer lifecycle chaincode package` 命令 将智能合约打包成链码。在 MagnetoCorp 管理员的命令窗口中， 执行如下命令创建链码包：

```
(magnetocorp admin)$ peer lifecycle chaincode package cp.tar.gz --lang node --path ./contract --l
```

MagnetoCorp 管理员现在可以使用 `peer lifecycle chaincode install` 命令 在 MagnetoCorp 的 peer 节点上安装链码：

```
(magnetocorp admin)$ peer lifecycle chaincode install cp.tar.gz
```

在安装了链码包后， 您会在您的终端上看到类似如下的消息：

```
2020-01-30 18:32:33.762 EST [cli.lifecycle.chaincode] submitInstallProposal -> INFO 001 Installed
2020-01-30 18:32:33.762 EST [cli.lifecycle.chaincode] submitInstallProposal -> INFO 002 Chaincode
```

因为 MagnetoCorp 管理员已经设置了 `CORE_PEER_ADDRESS=localhost:9051` 来将 `peer0.org2.example.com` 作为指令的目标， 所以 `INFO 001 Installed remotely...` 表示 `papercontract` 已被成功安装在此 peer 节点上。

在安装智能合约之后，我们需要以 MagnetoCorp 的身份 同意 `papercontract` 的链码定义。 第一步是找到我们安装在我们的 peer 上的链码的 packageID。我们可以使用 `peer lifecycle chaincode queryinstalled` 命令查询 packageID：

```
peer lifecycle chaincode queryinstalled
```

该命令将返回与安装命令相同的包标识符。 您应该会看到类似如下的输出：

```
Installed chaincodes on peer:
Package ID: cp_0:ffda93e26b183e231b7e9d5051e1ee7ca47fbf24f00a8376ec54120b1a2a335c, Label: cp_0
```

在下一步中，我们将需要 package ID， 因此我们将其保存为一个环境变量。对于所有用户， package ID 可能不相同， 因此您需要使用从命令窗口返回的 package ID 来完成这个步骤。

```
export PACKAGE_ID=cp_0:ffda93e26b183e231b7e9d5051e1ee7ca47fbf24f00a8376ec54120b1a2a335c
```

管理员现在可以使用 `peer lifecycle chaincode approveformyorg` 命令 为 MagnetoCorp 同意链码定义：

```
(magnetocorp admin)$ peer lifecycle chaincode approveformyorg --orderer localhost:7050 --ordererT
```

为了使用链码定义，通道成员需要同意的最重要的链码参数之一 是链码背书策略。 背书策略描述了在确定交易有效之前必须背书(执行和签署)的组织集合。通过不指定 `--policy` 标志而同意 `papercontract` 链码， MagnetoCorp管理员将同意使用通道的默认 `Endorsement` 策略， 这在 `mychannel` 测试通道的实例下 要求通道上的大多数组织来背书交易。所有的交易，无论有效还是无效，都将被记录在 [区块链账本](#) 上， 但只有有效的交易才会更新[世界状态](#)。

以 DigiBank 的身份安装和批准智能合约

基于 `mychannel` 的 `LifecycleEndorsement` 策略， Fabric链码生命周期将要求通道上的大多数组织 在将链码提交到通道之前同意链码的定义。 这意味着我们需要以MagnetoCorp和DigiBank的身份同意 `papernet` 链码， 以达成所需的多数即2/2的要求。 在 `fabric-samples` 文件夹下打开一个新的终端窗口， 并浏览到包含DigiBank智能合约和应用程序文件的文件夹:

```
(digibank admin)$ cd commercial-paper/organization/digibank/
```

使用DigiBank文件夹中的脚本设置环境变量， 这将允许您作为DigiBank管理员操作:

```
source digibank.sh
```

我们现在可以以DigiBank的身份安装和同意 `papercontract` 。 执行如下命令打包链码:

```
(digibank admin)$ peer lifecycle chaincode package cp.tar.gz --lang node --path ./contract --labe
```

管理员现在可以在DigiBank的peer节点上安装链码:

```
(digibank admin)$ peer lifecycle chaincode install cp.tar.gz
```

然后我们需要查询并保存刚刚安装的 链码的packageID:

```
(digibank admin)$ peer lifecycle chaincode queryinstalled
```

将package ID保存为环境变量。 使用从控制台返回的package ID完成此步骤。

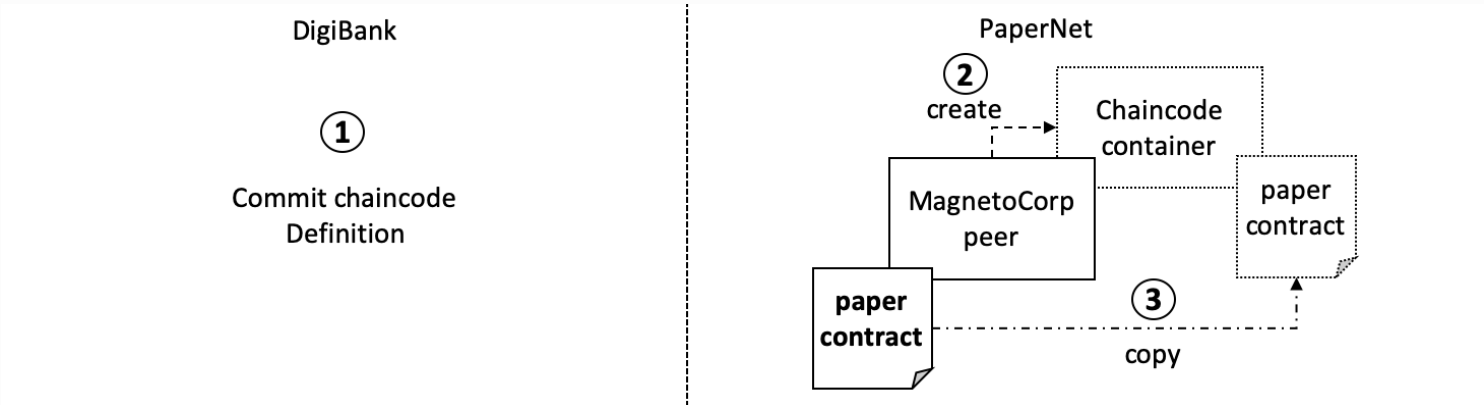
```
export PACKAGE_ID=cp_0:ffda93e26b183e231b7e9d5051e1ee7ca47fbf24f00a8376ec54120b1a2a335c
```

Digibank管理员现在可以同意 `papercontract` 的链码定义:

```
(digibank admin)$ peer lifecycle chaincode approveformyorg --orderer localhost:7050 --ordererTLSh
```

将链码定义提交到通道

现在，DigiBank和Magnetocorp都同意了 `papernet` 链码， 我们有了我们需要的大多数(2/2)组织的条件来提交链码定义到通道。一旦在通道上成功定义了链码， 通道上的客户端应用程序就可以调用 `papercontract` 链码中的 `CommercialPaper` 智能合约。由于任何一个机构都可以提交链码到通道， 我们将继续以DigiBank管理员的身份操作：



在DigiBank管理员将 `papercontract` 链码的定义提交到通道后， 将创建一个新的Docker链码容器， 以便在PaperNet的两个peer节点上运行 `papercontract`

DigiBank管理员使用 `peer lifecycle chaincode commit` 命令 将 `papercontract` 的链码定义提交到 `mychannel`：

```
(digibank admin)$ peer lifecycle chaincode commit -o localhost:7050 --ordererTLSHostnameOverride
```

链码容器将在链码定义提交到通道后启动。 您可以使用 `docker ps` 命令在两个peer节点上 看到 `papercontract` 容器的启动。

```
(digibank admin)$ docker ps
```

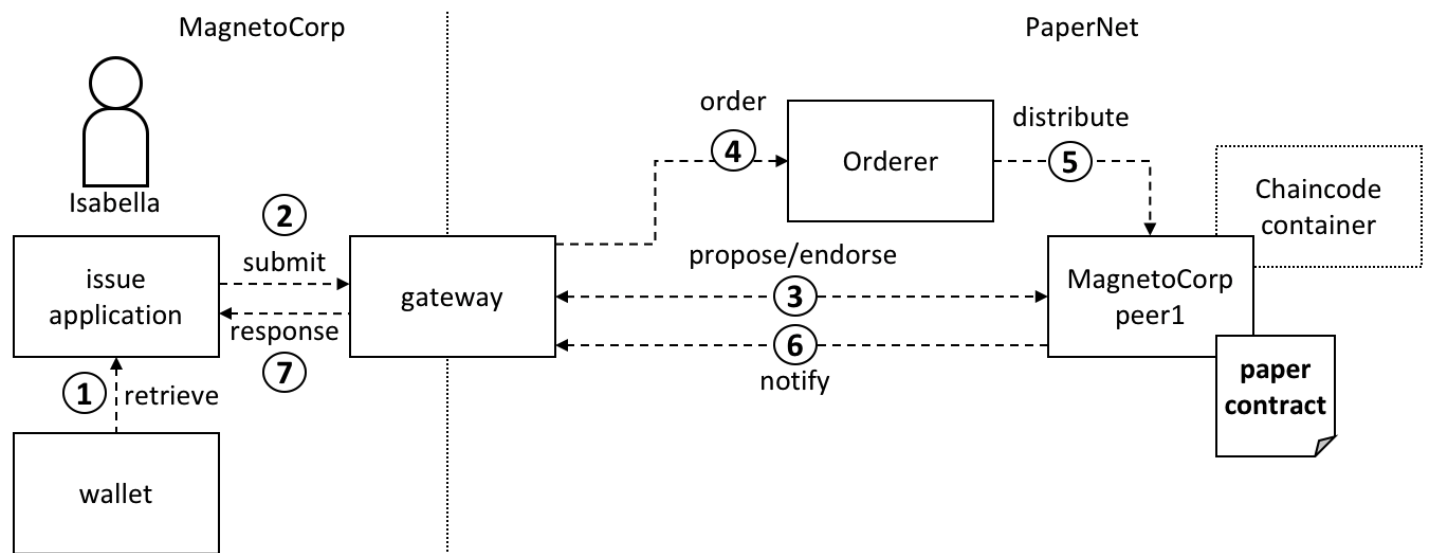
CONTAINER ID	IMAGE
d4ba9dc9c55f	dev-peer0.org1.example.com-cp_0-ebef35e7f1f25eea1dcc6fcad5019477cd7f434c6a5dc
a944c0f8b6d6	dev-peer0.org2.example.com-cp_0-1487670371e56d107b5e980ce7f66172c89251ab21d48

注意， 容器的命名指出了启动它的peer节点， 以及它正在运行 `papercontract` 版本 `0` 的实际情况。

现在我们已经将 `papercontract` 链代码部署到通道， 我们可以使用Magnetocorp应用程序来发行商业票据。 让我们花点时间检查一下应用程序的结构。

应用结构

包含在 `papercontract` 中的智能合约由 Magnetocorp 的应用程序 `issue.js` 调用。Isabella 使用该程序向发行商业票据 `00001` 的账本提交一项交易。让我么来快速检验一下 `issue` 应用是怎么工作的。



gateway允许应用程序专注于交易的生成、提交和响应。它协调不同网络组件之间的交易提案、排序和通知处理。

`issue` 应用程序代表 Isabella 提交交易，它通过从 Isabella 的 `钱包` 中索取其 X.509 证书来开始运行，此证书可能储存在本地文件系统中或一个硬件安全模块 `HSM` 里。随后，`issue` 应用程序就能够利用 gateway在通道上提交交易。Hyperledger Fabric的软件开发包（SDK）提供了一个 `gateway` 抽象，因此应用程序在将网络交互托管给网关时能够专注于应用逻辑。网关和钱包使得编写 Hyperledger Fabric 应用程序变得很简单。

让我们来检验一下 Isabella 将要使用的 `issue` 应用程序，为其打开另一个终端窗口，在 `fabric-samples` 中找到 MagnetoCorp 的 `/application` 文件夹：

```
(isabella)$ cd commercial-paper/organization/magnetocorp/application/
(isabella)$ ls

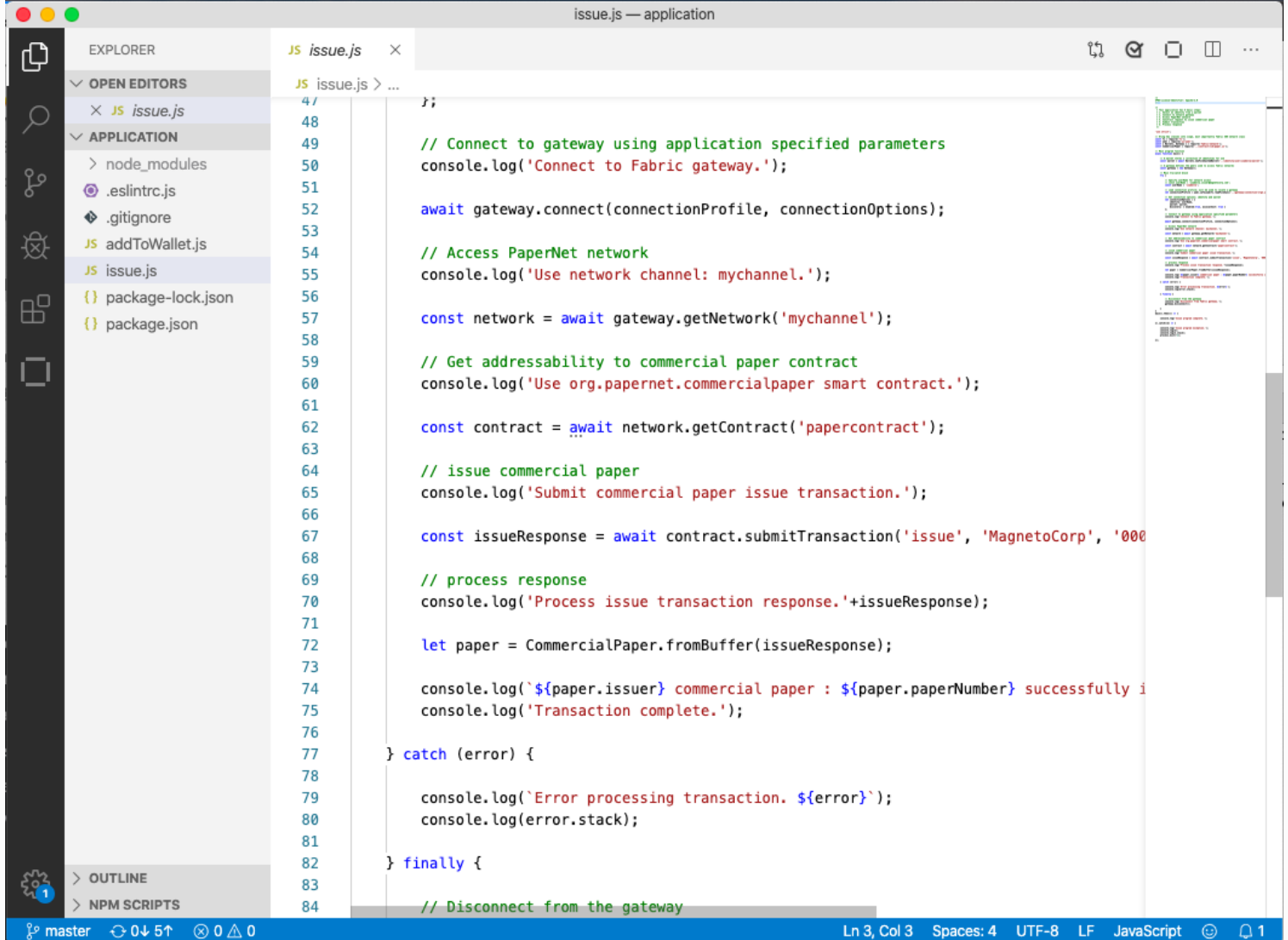
enrollUser.js      issue.js           package.json
```

`addToWallet.js` 是 Isabella 将用来把自己的身份装进钱包的程序，而 `issue.js` 将使用这一身份通过调用 `papercontract` 来代表 MagnetoCorp 生成商业票据 `00001`。

切换至包含 MagnetoCorp 的 `issue.js` 应用程序副本的目录，并且使用您的代码编辑器检查此目录：

```
(isabella)$ cd commercial-paper/organization/magnetocorp/application
(isabella)$ code issue.js
```

检查该目录；目录包含了 `issue` 应用程序和其所有依赖项。



一个展示了商业票据应用程序目录内容的代码编辑器。

注意以下在 `issue.js` 中的关键程序行：

- `const { Wallets, Gateway } = require('fabric-network');`

该语句把两个关键的 Hyperledger Fabric 软件开发包（SDK）类引入了 `Wallet` 和 `Gateway` 。

- `const wallet = await Wallets.newFileSystemWallet('../identity/user/isabella/wallet');`

该语句表明了应用程序在连接到区块链网络通道上将使用 `Isabella` 钱包。因为 Isabella 的 X.509 证书位于本地文件系统中，所以应用程序创建了一个 `FileSystemWallet`。应用程序会在 `isabella` 钱包中选择一个特定的身份。

- `await gateway.connect(connectionProfile, connectionOptions);`

此行代码使用 `connectionProfile` 识别的网关来连接到网络，使用 `ConnectionOptions` 当中引用的身份。

看看 `../gateway/networkConnection.yaml` 和 `User1@org1.example.com` 是如何分别被用于这些值的。

- `const network = await gateway.getNetwork('mychannel');`

该语句是将应用程序连接到网络通道 `mychannel` 上，`papercontract` 之前就已经在该通道上部署过了。

- `const contract = await network.getContract('papercontract');`

该语句是让应用程序可以访问由 `papercontract` 中的 `org.papernet.commercialpaper` 命名空间定义的智能合约。一旦应用程序请求了 `getContract`，那么它就能提交任意在其内实现的交易。

- `const issueResponse = await contract.submitTransaction('issue', 'MagnetoCorp', '00001', ...);`

该行代码是使用在智能合约中定义的 `issue` 交易来向网络提交一项交易。`MagnetoCorp`，`00001` ... 都是被 `issue` 交易用来生成一个新的商业票据的值。

- `let paper = CommercialPaper.fromBuffer(issueResponse);`

此语句是处理 `issue` 交易发来的响应。该响应需要从缓冲区被反序列化成 `paper`，这是一个能够被应用程序准确解释的 `CommercialPaper` 对象。

欢迎检查 `/application` 目录下的其他文档来了解 `issue.js` 是如何工作的，并仔细阅读应用程序 [主题](#) 中关于如何实现 `issue.js` 的内容。

应用程序依赖

`issue.js` 应用程序是用 JavaScript 编写的，旨在作为 PaperNet 网络的客户端来在 node.js 环境中运行。按照惯例，会在多个网络外部的节点包上建立 MagnetoCorp 的应用程序，以此来提升开发的质量和速度。考虑一下 `issue.js` 是如何纳入 `js-yaml` 包来处理 YAML 网关连接配置文件的，或者 `issue.js` 是如何纳入 `fabric-network` 包来访问 `Gateway` 和 `Wallet` 类的：

```
const yaml = require('js-yaml');
const { Wallets, Gateway } = require('fabric-network');
```

需要使用 `npm install` 命令来将这些包从 `npm` 下载到本地文件系统中。按照惯例，必须将包安装进一个相对于应用程序的 `/node_modules` 目录中，以供运行时使用。

检查 `package.json` 文件来看看 `issue.js` 是如何通过识别包装来下载自己的准确版本的：

`npm` 版本控制功能非常强大；点击[这里](#)可以了解更多相关信息。

让我们使用 `npm install` 命令来安装这些包，安装过程可能需要一分钟：

```
(isabella)$ cd commercial-paper/organization/magnetocorp/application/
(isabella)$ npm install

(
  (...)
  ) extract:lodash: sill extract ansi-styles@3.2.1
(...)
added 738 packages in 46.701s
```

看看这个命令是如何更新目录的：

```
(isabella)$ ls

enrollUser.js      node_modules      package.json
issue.js           package-lock.json
```

检查 `node_modules` 目录，查看已经安装的包。能看到很多已经安装了的包，这是因为 `js-yaml` 和 `fabric-network` 本身都被搭建在其他 npm 包中！`package-lock.json` 文件 能准确识别已安装的版本，如果您想用于生产环境的话，那么这一点对您来说就很重要。例如，测试、排查问题或者分发已验证的应用。

钱包

Isabella 马上就能够运行 `issue.js` 来发行 MagnetoCorp 商业票票据 `00001` 了；现在还剩最后一步！因为 `issue.js` 代表 Isabella，所以也就代表 MagnetoCorp，`issue.js` 将会使用 Isabella 钱包中反应以上事实的身份。现在我们需要执行这个一次性的活动，向 Isabella 的钱包中添 X.509 证书。

运行在PaperNet上的MagnetoCorp证书颁发机构 `ca_org2`， 有一个在部署网络时便注册的应用程序用户。Isabella可以使用身份名和secret 为 `issue.js` 应用程序生成X.509加密材料。使用CA生成客户端加密资料的过程称为**enrollment**。在实际的应用场景中，网络运营商将向应用程序开发人员 提供使用CA所注册的客户端身份的名称和secret。然后，开发人员将使用证书凭据注册他们的应用程序并与网络交互。

`enrollUser.js` 程序使用 `fabric-ca-client` 类生成私有、公共密钥对, 然后发起一个**Certificate Signing Request**给CA。如果Isabella提交的身份名称和secret匹配CA注册的证书凭据, CA将发行并签名一个编码了公钥的证书, 证明Isabella属于Isabella签名。签名请求完成后，`enrollUser.js` 将私钥和签名证书存储在Isabella的钱包中。您可以查看 `enrollUser.js` 文件， 了解更多关于Node SDK如何使用 `fabric-ca-client` 类来完成这些任务的信息。

在 Isabella 的终端窗口中运行 `addToWallet.js` 程序来把身份信息添加到她的钱包中：

```
(isabella)$ node enrollUser.js

Wallet path: /Users/nikhilgupta/fabric-samples/commercial-paper/organization/magnetocorp/identity
Successfully enrolled client user "isabella" and imported it into the wallet
```

现在我们可以把焦点转向这个程序的结果—— 将用于提交交易到PaperNet的钱包内容：

```
(isabella)$ ls ../identity/user/isabella/wallet/

isabella.id
```

Isabella可以在她的钱包中存储多个身份， 但在我们的示例中，她只使用一个。`wallet` 文件夹里有一个 `isabella.id` 文件， 该文件提供Isabella连接到网络所需的信息。Isabella使用的其他身份都有自己对应的文件。您可以打开这个文件， 查看JSON文件中 `issue.js` 将使用的代表Isabella的身份标识信息。为清晰起见，输出已经进行了格式化。

```
(isabella)$ cat ../identity/user/isabella/wallet/*

{
  "credentials": {
    "certificate": "-----BEGIN CERTIFICATE-----\nMIICKTCCAdCgAwIBAgIQWKwvLG+sqeO3LwwQK6avZDAKBggq\n"privateKey": "-----BEGIN PRIVATE KEY-----\nMIGHAgEAMBMGBYqGSM49AgEGCCqGSM49AwEHBG0wawIBAQQgg\n  },
```



```
"mspId": "Org2MSP",
"type": "X.509",
"version": 1
}
```

在文件中您可以注意到以下内容:

- `"privateKey"`: 用来代表Isabella签名交易，但不能被分发到她的直接控制范围之外。
- `"certificate"`: 它包含Isabella的公钥和证书颁发机构在创建证书时添加的其他X.509属性。该证书被分发到网络中，以便不同的参与者可以在不同的时间以加密方式验证由Isabella的私钥加密过的信息。

点击[此处](#)获取更多关于证书信息。在实践中，证书文档还包含一些 Fabric 专门的元数据，例如 Isabella 的组织角色——在[钱包](#)主题阅读更多内容。

发行应用

Isabella 现在可以用 `issue.js` 来提交一项交易，该交易将发行 MagnetoCorp 商业票据 `00001`：

```
(isabella)$ node issue.js

Connect to Fabric gateway.
Use network channel: mychannel.
Use org.papernet.commercialpaper smart contract.
Submit commercial paper issue transaction.
Process issue transaction response.{"class":"org.papernet.commercialpaper","key":"\"MagnetoCorp\"
MagnetoCorp commercial paper : 00001 successfully issued for value 5000000
Transaction complete.
Disconnect from Fabric gateway.
Issue program complete.
```

`node` 命令初始化一个 node.js 环境，并运行 `issue.js`。从程序输出我们能看到，系统发行了一张 MagnetoCorp 商业票据 00001，面值为 500 万美元。

如您所见，为实现这一点，应用程序调用了 `papercontract.js` 中 `CommercialPaper` 智能合约里定义的 `issue` 交易。MagnetoCorp 管理员已经在网络上安装并实例化了 `CommercialPaper` 智能合约。在世界状态里作为一个矢量状态来代表新的商业票据的是通过 Fabric 应用程序编码端口(API)来与账本交互的智能合约，其中最显著的 API 是 `putState()` 和 `getState()`。我们即将看到该矢量状态在随后是如何被 `buy` 和 `redeem` 交易来操作的，这两项交易同样也是定义在那个智能合约中。

潜在的 Fabric 软件开发包（SDK）一直都在处理交易的背书、排序和通知流程，使得应用程序的逻辑变得简单明了；SDK 用[网关](#)提取出网络细节信息和[连接选项](#)，以此来声明更先进的流程策略，如交易重试。

现在让我们将重点转换到 DigiBank（将购买商业票据），以遵循 MagnetoCorp `00001` 的生命周期。

DigiBank应用

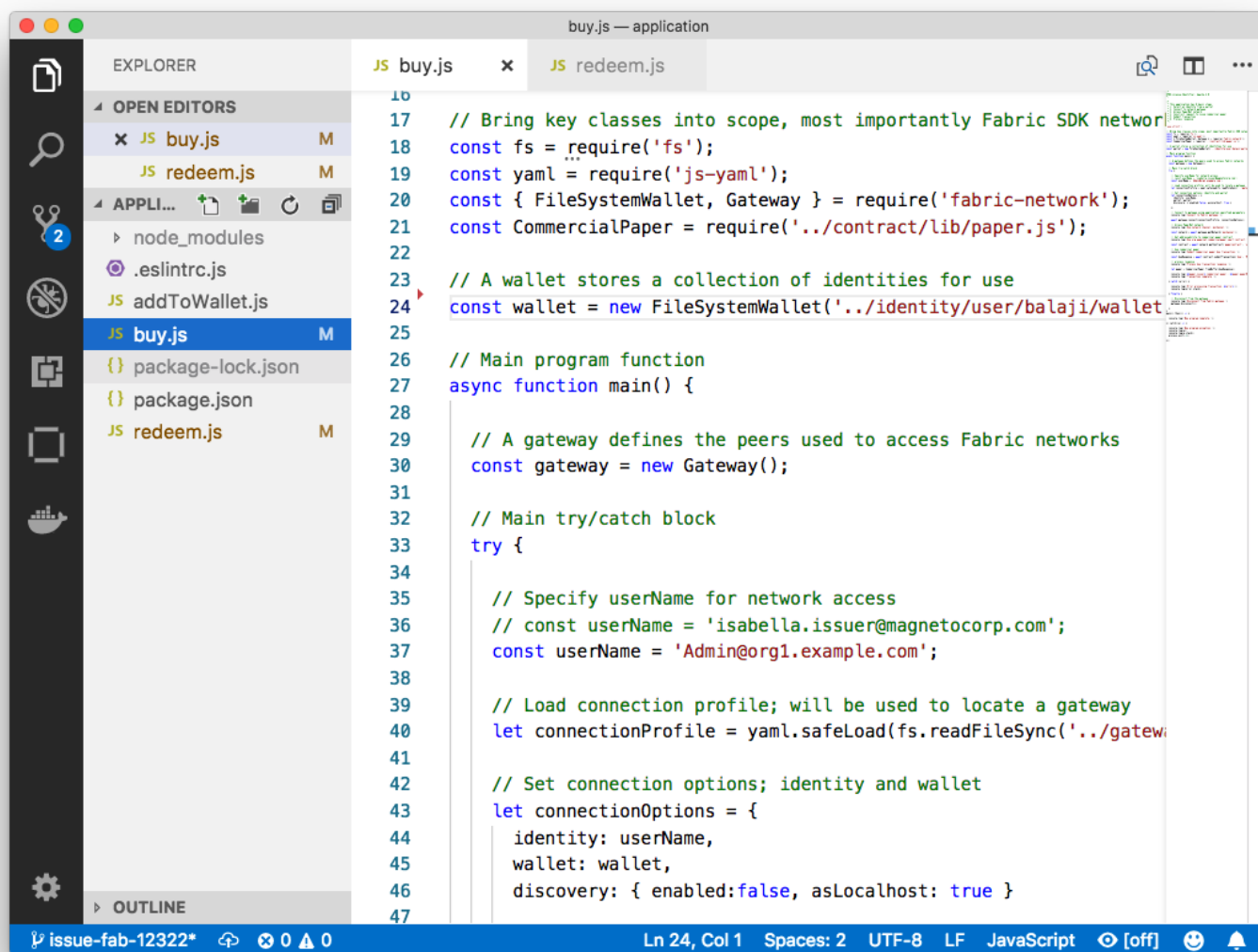
Balaji 使用 DigiBank 的 `buy` 应用程序来向账本提交一项交易，该账本将商业票据 `00001` 的所属权从 MagnetoCorp 转向 DigiBank。`CommercialPaper` 智能合约与 MagnetoCorp 应用程序使用的相同，但

是此次的交易不同，是 `buy` 交易而不是 `issue` 交易。让我们检查一下 DigiBank 的应用程序是怎样工作的。

为 Balaji 打开另一个终端窗口。在 `fabric-samples` 中，切换到包含 `buy.js` 应用程序的 DigiBank 应用程序目录，并用编辑器打开该目录：

```
(balaji)$ cd commercial-paper/organization/digibank/application/  
(balaji)$ code buy.js
```

如您所见，该目录同时包含了 Balaji 将使用的 `buy` 和 `redeem` 应用程序。



DigiBank 的商业票据目录包含 `buy.js` 和 `redeem.js` 应用程序。

DigiBank 的 `buy.js` 应用程序在结构上与 MagnetoCorp 的 `issue.js` 十分相似，但存在两个重要的差异：

- 身份：用户是 DigiBank 的用户 `Balaji` 而不是 MagnetoCorp 的 `Isabella`

```
const wallet = await Wallets.newFileSystemWallet('../identity/user/balaji/wallet');
```

看看应用程序在连接到 PaperNet 网络上时是如何使用 `balaji` 钱包的。`buy.js` 在 `balaji` 钱包里选择一个特定的身份。

- **交易**：被调用的交易是 `buy` 而不是 `issue`

```
const buyResponse = await contract.submitTransaction('buy', 'MagnetoCorp', '00001', ...);
```

提交一项 `buy` 交易，其值为 `MagnetoCorp`，`00001` ...，`CommercialPaper` 智能合约类使用这些值来将商业票据 `00001` 的所属权转换成 DigiBank。

欢迎检查 `application` 目录下的其他文档来理解应用程序的工作原理，并仔细阅读应用程序[主题](#)中关于如何实现 `buy.js` 的内容。

以 DigiBank 的身份运行

负责购买和赎回商业票据的 DigiBank 应用程序的结构和 MagnetoCorp 的发行交易十分相似。所以，我们来安装这些应用程序 的依赖项，并搭建 Balaji 的钱包，这样一来，Balaji 就能使用这些应用程序购买和赎回商业票据。

和 MagnetoCorp 一样，Digibank 必须使用 `npm install` 命令来安装规定的应用包，同时，安装时间也很短。

在 DigiBank 管理员窗口安装应用程序依赖项：

```
(digibank admin)$ cd commercial-paper/organization/digibank/application/
(digibank admin)$ npm install

(
  ...
) extract:lodash: sill extract ansi-styles@3.2.1
(...)
added 738 packages in 46.701s
```

在 Balaji 的终端窗口运行 `addToWallet.js` 程序，把身份信息添加到他的钱包中：

`addToWallet.js` 程序为 `balaji` 将其身份信息添加到他的钱包中，`buy.js` 和 `redeem.js` 将使用这些身份信息来向 `PaperNet` 提交交易。

```
(balaji)$ node enrollUser.js

Wallet path: /Users/nikhilgupta/fabric-samples/commercial-paper/organization/digibank/identity/us
Successfully enrolled client user "balaji" and imported it into the wallet
```

`addToWallet.js` 程序已经将 `balaji` 的身份信息添加到他的钱包中，该钱包将被 `buy.js` 和 `redeem.js` 用于向 `PaperNet` 提交交易。

与 Isabella 一样，Balaji 可以在他的钱包中存储多个身份，但在我们的示例中，他只使用一个。他在 `digibank/identity/user/balaji/wallet/balaji.id` 对应的 id 文件 和 Isabella 的非常相似——请随意查看。

购买应用

Balaji 现在可以使用 `buy.js` 来提交一项交易，该交易将会把 MagnetoCorp 商业票据 `00001` 的所属权转换成 DigiBank。

在 Balaji 的窗口运行 `buy` 应用程序：

```
(balaji)$ node buy.js

Connect to Fabric gateway.
Use network channel: mychannel.
Use org.papernet.commercialpaper smart contract.
Submit commercial paper buy transaction.
Process buy transaction response.
MagnetoCorp commercial paper : 00001 successfully purchased by DigiBank
Transaction complete.
Disconnect from Fabric gateway.
Buy program complete.
```

您可看到程序输出为：Balaji 已经代表 DigiBank 成功购买了 MagnetoCorp 商业票据 `00001`。`buy.js` 调用了 `CommercialPaper` 智能合约中定义的 `buy` 交易，该智能合约使用 Fabric 应用程序编程接口 (API) `putState()` 和 `getState()` 在世界状态中更新了商业票据 `00001`。如您所见，就智能合约的逻辑来说，购买和发行商业票据的应用程序逻辑彼此十分相似。

赎回应用

商业票据 `00001` 生命周期的最后一步交易是 DigiBank 从 MagnetoCorp 那里赎回商业票据。Balaji 使用 `redeem.js` 提交一项交易来执行智能合约中的赎回逻辑。

在 Balaji 的窗口运行 `redeem` 交易：

```
(balaji)$ node redeem.js

Connect to Fabric gateway.
Use network channel: mychannel.
Use org.papernet.commercialpaper smart contract.
Submit commercial paper redeem transaction.
Process redeem transaction response.
MagnetoCorp commercial paper : 00001 successfully redeemed with MagnetoCorp
Transaction complete.
Disconnect from Fabric gateway.
Redeem program complete.
```

同样地，看看当 `redeem.js` 调用了 `CommercialPaper` 中定义的 `redeem` 交易时，商业票据 `00001` 是如何被成功赎回的。`redeem` 交易在世界状态中更新了商业票据 `00001`，以此来反映商业票据的所属权又回归其发行方 MagnetoCorp。

清理

在您完成商业票据教程后，您可以使用一个脚本来清理您的环境。 打开一个命令窗口浏览回商业票据示例的根目录：

```
cd fabric-samples/commercial-paper
```

然后您可以使用以下命令关闭网络:

```
./network-clean.sh
```

除了logspout工具之外，这个命令还将关闭peer节点、CouchDB容器和网络中的排序节点。它还会移除我们为Isabella和Balaji创造的身份标识。请注意，账本上的所有数据都将丢失。如果你想再次学习教程，你将从一个干净的初始状态开始。

下一步

要想更深入地理解以上教程中所介绍的应用程序和智能合约的工作原理，可以参照 [开发应用程序](#)。该主题将为您详细介绍商业票据场景、[PaperNet](#) 商业网络，网络操作者以及它们所使用的应用程序和智能合约的工作原理。

欢迎使用该样本来开始创造您自己的应用程序和智能合约！