一. 入门

在我们开始之前，如果您尚未这样做，则不妨检查一下是否已在要开发区块链应用程序和/或运行 Hyperledger Fabric 的平台上安装了所有必备软件。

安装必备组件后，就可以下载和安装 HyperLedger Fabric。在我们为 Fabric 二进制文件开发真正的安装程序的同时，我们提供了一个脚本，该脚本会将 Samples，Binaries 和 Docker 映像安装到您的系统。该脚本还将把 Docker 镜像下载到您的本地注册表中。

1. Hyperledger Fabric 智能合约（链码）SDK

Hyperledger Fabric 提供了许多 SDK，以支持以各种编程语言开发智能合约（链码）。智能合约 SDK 可用于 Go，Node.js 和 Java：

- 转到 SDK 文档。
- Node.js SDK 和 Node.js SDK 文档。
- Java SDK 和 Java SDK 文档。

当前，Node.js 和 Java 支持 Hyperledger Fabric v1.4 中提供的新智能合约编程模型。计划在以后的版本中提供对 Go 的支持。

2. Hyperledger Fabric 应用程序 SDK

Hyperledger Fabric 提供了许多 SDK，以支持使用各种编程语言开发应用程序。SDK 可用于 Node.js 和 Java：

- Node.js SDK 和 Node.js SDK 文档。
- Java SDK 和 Java SDK 文档。

此外，还有两个尚未正式发布的应用程序 SDK（针对 Python 和 Go），但仍可供下载和测试：

- Python SDK。
- 转到 SDK。

当前，Node.js 和 Java 支持 Hyperledger Fabric v1.4 中提供的新应用程序编程模型。计划在以后的版本中提供对 Go 的支持。

3. Hyperledger Fabric CA

Hyperledger Fabric 提供了可选的 证书颁发机构服务 ，您可以选择使用该服务来生成证书和密钥材料，以配置和管理区块链网络中的身份。但是，可以使用任何可以生成 ECDSA 证书的 CA。

4. 先决条件

在我们开始之前，如果您尚未这样做，则不妨检查一下是否在要开发区块链应用程序和/或运行 Hyperledger Fabric 的平台上安装了以下所有先决条件。

5. 安装 Git

如果尚未安装最新版本的 git，或者如果您在运行 curl 命令时遇到问题，请下载它。

6. 安装 cURL

如果尚未安装 cURL 工具的最新版本，或者从文档中运行 curl 命令时遇到错误，请下载它。

注意

如果您使用的是 Windows，请参见下面有关 Windows Extras 的特定说明。

7. Docker 和 Docker Compose

您将需要在运行或在 Hyperledger Fabric 上（或为其开发）的平台上安装以下组件：

- MacOSX，* nix 或 Windows 10：Docker 需要 Docker 17.06.2-ce 或更高版本。
- Windows 的旧版本：Docker Toolbox-同样，需要 Docker 版本 Docker 17.06.2-ce 或更高。

您可以在终端提示符下使用以下命令检查已安装的 Docker 版本：

```
docker --version
```

注意

以下内容适用于运行 systemd 的 linux 系统。

确保 Docker 守护程序正在运行。

```
sudo systemctl start docker
```

可选：如果要在系统启动时启动 docker 守护程序，请使用以下命令：

```
sudo systemctl enable docker
```

将您的用户添加到 docker 组。

```
sudo usermod -a -G docker <username>
```

注意

安装适用于 Mac 或 Windows 的 Docker 或 Docker Toolbox 还将安装 Docker Compose。如果已经安装了 Docker，则应检查是否已安装 Docker Compose 1.14.0 或更高版本。如果没有，我们建议您安装较新版本的 Docker。

您可以从终端提示符使用以下命令检查已安装的 Docker Compose 的版本：

```
docker-compose --version
```

8. GO 编程语言

Hyperledger Fabric 将 Go 编程语言用于其许多组件。

- GO 版本 1.12.x 是必需的。

假设我们将用 Go 编写链式代码程序，则需要正确设置两个环境变量。您可以通过将这些设置放置在适当的启动文件（例如，`~/.bashrc` 如果您 `bash` 在 Linux 下使用 Shell）中的个人文件中来使它们永久化。

首先，您必须将环境变量设置 `GOPATH` 为指向包含下载的 Fabric 代码库的 Go 工作区，例如：

```
export GOPATH=$HOME/go
```

注意

您**必须**设置 GOPATH 变量

即使在 Linux 中，Go 的 `GOPATH` 变量可以是用冒号分隔的目录列表，并且将使用默认值（`$HOME/go` 如果未设置），则当前的 Fabric 构建框架仍然需要您设置和导出该变量，并且该变量**只能**包含 Go 工作空间的单个目录名称。（此限制可能会在将来的版本中删除。）

其次，您应该（再次在适当的启动文件中）扩展命令搜索路径以包括 Go `bin` 目录，例如 `bash` Linux 下的以下示例：

```
export PATH=$PATH:$GOPATH/bin
```

虽然此目录可能在新的 Go 工作区安装中不存在，但是稍后由 Fabric 构建系统填充，并由构建系统的其他部分使用少量的 Go 可执行文件。因此，即使您当前还没有这样的目录，也可以像上面那样扩展 shell 搜索路径。

9. Node.js 运行时和 NPM

如果要利用适用于 Node.js 的 Hyperledger Fabric SDK 开发 Hyperledger Fabric 应用程序，则 8.9.4 和更高版本支持版本 8。从 10.15.3 及更高版本开始支持 Node.js 版本 10。

- Node.js 下载

注意

安装 Node.js 还将安装 NPM，但是建议您确认已安装的 NPM 版本。您可以 `npm` 使用以下命令升级该工具：

```
npm install npm@5.6.0 -g
```

4.1. PYTHON

注意
以下内容仅适用于 Ubuntu 16.04 用户。
默认情况下，Ubuntu 16.04 附带安装了 Python 3.5.1 作为 `python3` 二进制文件。Fabric Node.js SDK 需要 Python 2.7 的迭代才能 成功完成操作。使用以下命令检索 2.7 版本：`npm install`

```
sudo apt-get install python
```

检查您的版本：

```
python --version
```

Windows Extras

如果您在 Windows 7 上进行开发，则需要在使用 Git Bash 的 Docker Quickstart Terminal 中工作，并为内置 Windows shell 提供更好的替代方法。
但是，经验表明这是一个功能有限的不良开发环境。它适合运行基于 Docker 的场景，例如 Getting Started，但是您可能难以处理涉及 `make` and `docker` 命令的操作。
在 Windows 10 上，您应该使用本机 Docker 发行版，并且可以使用 Windows PowerShell。但是，`binaries` 要使命令成功执行，您仍然需要使 `uname` 命令可用。您可以将其作为 Git 的一部分获得，但请注意，仅支持 64 位版本。
在运行任何命令之前，请运行以下命令：`git clone`

```
git config --global core.autocrlf false
git config --global core.longpaths true
```

您可以使用以下命令检查这些参数的设置：

```
git config --get core.autocrlf
git config --get core.longpaths
```

这些分别是 `false` 和 `true`。

`curl` Git 和 Docker Toolbox 随附的命令较旧，无法正确处理“ 入门”中使用的重定向 。确保从 cURL 下载页面安装并使用较新版本

对于 Node.js，您还需要必要的 Visual Studio C ++构建工具，这些工具可以免费获得，并且可以使用以下命令进行安装：

```
npm install --global windows-build-tools
```

有关更多详细信息，请参见 NPM windows-build-tools 页面。
完成此操作后，还应该使用以下命令安装 NPM GRPC 模块：

```
npm install --global grpc
```

现在，您的环境应该已经准备就绪，可以 开始阅读“ 入门”示例和教程。
注意
如果您有本文档未解决的问题，或在任何教程中遇到问题，请访问“ 还有问题？”。 页面上有关在哪里可以找到其他帮助的一些提示。

二. 安装样本，二进制文件和 Docker 映像

当我们为 Hyperledger Fabric 二进制文件开发实际的安装程序时，我们提供了一个脚本，该脚本会将示例和二进制文件下载并安装到您的系统中。我们认为您会发现安装的示例应用程序对于了解有关 Hyperledger Fabric 的功能和

操作的更多信息很有用。

注意

如果您在 **Windows** 上运行，则将要使用 Docker Quickstart Terminal 作为即将到来的终端命令。如果您以前没有安装先决条件，请访问它。

如果您在 Windows 7 或 macOS 上使用 Docker Toolbox，则在安装和运行示例时将需要使用 `C:\Users`（Windows 7）或 `/Users`（macOS）下的位置。

如果您使用的码头工人的 Mac，你需要在使用位置 `/Users`， `/Volumes`， `/private`，或 `/tmp`。要使用其他位置，请查阅 Docker 文档以获取 文件共享。

如果您使用的是 Docker for Windows，请查阅 Docker 文档中的共享驱动器 并使用其中一个共享驱动器下的位置。

确定机器上要放置 *Fabric-samples* 存储库的位置，然后在终端窗口中输入该目录。后面的命令将执行以下步骤：

1. 如果需要，克隆 hyperledger / fabric-samples 存储库
2. 签出适当的版本标签
3. 将 Hyperledger Fabric 平台特定的二进制文件和配置文件安装为指定到 Fabric-samples 的/ bin 和/ config 目录中的版本
4. 下载指定版本的 Hyperledger Fabric docker 映像

准备就绪后，在要安装 Fabric Samples 和二进制文件的目录中，继续执行命令以下拉二进制文件和映像。

如果要使用最新的生产版本，请省略所有版本标识符。

```
curl -sSL http://bit.ly/2ysbOFE | bash -s
```

如果要特定版本，请为 Fabric，Fabric-ca 和第三方 Docker 映像传递版本标识符。下面的命令演示了如何下载 **Fabric v2.0.0 Alpha 版本 v2.0.0-alpha**

```
curl -sSL http://bit.ly/2ysbOFE | bash -s -- <fabric_version> <fabric-ca_version> <thirdparty_version>
curl -sSL http://bit.ly/2ysbOFE | bash -s -- 2.0.0-alpha 2.0.0-alpha 0.4.15
```

如果在运行上述 curl 命令时遇到错误，则可能是因为 curl 版本太旧而无法处理重定向或不支持的环境。

请访问“ 先决条件”页面，以获取有关在何处找到最新版本的 curl 并获得正确环境的更多信息。或者，您可以替换未缩短的 URL：https : //raw.githubusercontent.com/hyperledger/fabric/master/scripts/bootstrap.sh

上面的命令下载并执行一个 bash 脚本，该脚本将下载并提取设置网络所需的所有特定于平台的二进制文件，并将其放入上面创建的克隆存储库中。它检索以下特定于平台的二进制文件：

- `configtxgen`，
- `configtxlator`，
- `cryptogen`，
- `discover`，
- `idemixgen`
- `orderer`，
- `peer`，
- `fabric-ca-client`

并将它们放置 `bin` 在当前工作目录的子目录中。

您可能希望将其添加到 PATH 环境变量中，以便无需完全限定每个二进制文件的路径就可以选择它们。例如：

```
export PATH=<path to download location>/bin:$PATH
```

最后，该脚本会将 Hyperledger Fabric Docker 映像从 Docker Hub 下载到本地 Docker 注册表中，并将其标记为“最新”。

该脚本列出了最终安装的 Docker 映像。

查看每个图像的名称；这些是最终构成我们的 Hyperledger Fabric 网络的组件。您还将注意到，您有两个具有相

同映像 ID 的实例-一个实例标记为“amd64-1.xx”，另一个实例标记为“最新”。在 1.2.0 之前，要下载的映像由“x86_64-1.xx” 确定并显示为“x86_64-1.xx”。`uname -m`

在不同的体系结构上，x86_64 / amd64 将替换为标识您的体系结构的字符串。

如果您有本文档未解决的问题，或在任何教程中遇到问题，请访问“还有问题？”。 页面上有关在哪里可以找到其他帮助的一些提示。

三.开发应用程序

本主题涵盖如何使用 Hyperledger Fabric 开发客户端应用程序和智能合约以解决业务问题。在涉及多个组织的真实**商业票据**场景中，您将了解实现此目标所需的所有概念和任务。我们假设区块链网络已经可用。

该主题专为多个受众设计：

- 解决方案和应用架构师
- 客户端应用程序开发人员
- 智能合约开发商
- 商务专业

您可以选择按顺序阅读主题，也可以根据需要选择各个部分。各个主题部分均根据读者的相关性进行了标记，因此，无论您是要寻找业务信息还是技术信息，当一个主题适合您时都会很清楚。

该主题遵循典型的软件开发生命周期。它从业务需求开始，然后涵盖开发应用程序和智能合约以满足这些需求所需的所有主要技术活动。

如果您愿意，可以按照简短的说明通过运行商业票据教程立即尝试商业票据方案。当需要对本教程中介绍的概念进行更全面的说明时，可以返回此主题。

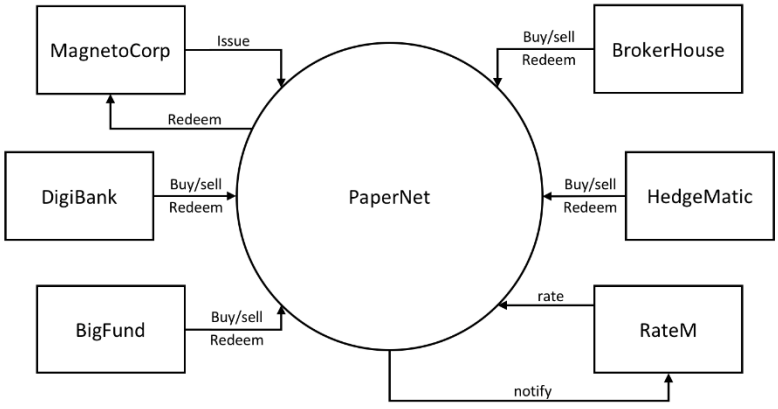
1. 场景

受众：建筑师，应用程序和智能合约开发人员，业务专业人员

在本主题中，我们将描述一个涉及六个组织的业务场景，这些组织使用 PaperNet（基于 Hyperledger Fabric 构建的商业票据网络）发行，购买和赎回商业票据。我们将使用该场景来概述参与组织使用的商业票据应用程序和智能合约的开发要求。

2. PaperNet 网络

PaperNet 是一个商业票据网络，允许经过适当授权的参与者发行，交易，赎回和评估商业票据。



PaperNet 商业票据网络。目前有六个组织使用 PaperNet 网络来发行，购买，出售，赎回和评价商业票据。MagentoCorp 发行和赎回商业票据。DigiBank，BigFund，BrokerHouse 和 HedgeMatic 都互相买卖商业票据。RateM 提供了各种商业票据风险度量。

让我们看看 MagnetoCorp 如何使用 PaperNet 和商业票据来帮助其业务。

3. 演示各方介绍

MagnetoCorp 是一家备受推崇的公司，生产自动驾驶电动汽车。2020 年 4 月上旬，MagnetoCorp 赢得了一项大

订单，为个人运输市场的新进入者戴因特里（Daintree）生产 10,000 辆 D 型汽车。尽管该订单对 MagnetoCorp 而言是一笔巨大的胜利，但在 9 月 1 日开始交付车辆之前，Daintree 不必为这些车辆付款。

要制造这些车辆，MagnetoCorp 将需要雇用 1000 名工人至少 6 个月。这给公司的财务带来了短期压力-每月将需要 500 万美元来支付这些新员工。**商业票据**旨在帮助 MagnetoCorp 克服其短期融资需求-满足每月的工资要求，因为人们期望当 Daintree 开始为新的 Model D 汽车付款时它将拥有大量现金。

在 5 月底，MagnetoCorp 需要 5 百万美元来支付 5 月 1 日雇用的额外工人的工资。为此，它发行了面值为 500 万美元的商业票据，到期日为 6 个月–当它希望看到 Daintree 的现金流。DigiBank 认为 MagnetoCorp 信誉良好，因此不需要高于央行 2％基本利率的溢价，后者在 6 个月内价值 495 万美元，今天为 500 万美元。因此，它以 494 万美元的价格购买了 MagnetoCorp 的 6 个月商业票据，与价值 495 万美元的价格相比略有折扣。DigiBank 完全希望它能够在 6 个月内从 MagnetoCorp 赎回 500 万美元，因承担与此商业票据相关的风险增加而获利 1 万美元。这额外的 10K 表示它收到 2。

6 月底，当 MagnetoCorp 发行 500 万美元的新商业票据以支付 6 月份的工资时，BigFund 以 494 万美元的价格购买了该票据。这是因为 6 月份的商业状况与 5 月份的大致相同，导致 BigFund 对 MagnetoCorp 商业票据的估值与 DigiBank 5 月份的价格相同。

随后的每个月，MagnetoCorp 都可以发行新的商业票据以履行其薪金义务，并且可以由 DigiBank 或 PaperNet 商业票据网络的任何其他参与者（BigFund，HedgeMatic 或 BrokerHouse）购买。这些组织可能会根据以下两个因素为商业票据支付或多或少的费用-中央银行基准利率和与 MagnetoCorp 相关的风险。后一个数字取决于多种因素，例如 D 型汽车的生产以及评级机构 RateM 评估的 MagnetoCorp 的信誉度。

PaperNet 中的组织扮演着不同的角色，MagnetoCorp 发行文件，DigiBank，BigFund，HedgeMatic 和 BrokerHouse 交易文件以及 RateM 利率文件。DigiBank，Bigfund，HedgeMatic 和 BrokerHouse 等具有相同角色的组织是竞争对手。角色不同的组织不一定是竞争对手，但可能仍会有相反的商业利益，例如，MagentoCorp 希望其论文的评级很高，以便以高价出售，而 DigiBank 将从低评级中受益，以便可以购买他们以低廉的价格。可以看出，即使是看似简单的网络（如 PaperNet）也可以具有复杂的信任关系。区块链可以帮助在竞争者或具有相反商业利益的组织之间建立信任，这可能会导致纠纷。

让我们暂停一下 MagnetoCorp 的故事，并开发 PaperNet 用来发行，购买，出售和赎回商业票据以及捕获组织之间的信任关系的客户端应用程序和智能合约。稍后，我们将回到评估机构 RateM 的角色。

四.分析

受众：建筑师，应用程序和智能合约开发人员，业务专业人员

让我们更详细地分析商业票据。诸如 MagnetoCorp 和 DigiBank 之类的 PaperNet 参与者使用商业票据交易来实现其业务目标–让我们研究一下商业票据的结构以及随着时间的推移会影响商业票据的交易。我们还将基于网络中各组织之间的信任关系，考虑 PaperNet 中的哪些组织需要在事务上签名。稍后，我们将集中讨论买卖双方之间的资金流向。现在，让我们集中讨论 MagnetoCorp 发表的第一篇论文。

1. 商业票据的生命周期

MagnetoCorp 在 5 月 31 日发布了论文 00001。花了一些时间看一下本文的第一**状态**，其不同的性质和值：

```
Issuer = MagnetoCorp
Paper = 00001
Owner = MagnetoCorp
Issue date = 31 May 2020
Maturity = 30 November 2020
Face value = 5M USD
Current state = issued
```

该纸状态是**发行**交易的结果，它使 MagnetoCorp 的第一张商业纸成为现实！请注意，本文的面值为 500 万美元，要在今年晚些时候进行赎回。请参见发出纸张 00001 时的 `Issuer` 和 `Owner` 相同。请注意，本文可以唯一地标识为 `MagnetoCorp00001` – `Issuer` 和 `Paper` 属性的组合。最后，了解该属性如何 快速识别 MagnetoCorp 纸 00001 在其生命周期中的阶段。`Current state = issued`

发行后不久，该纸被 DigiBank 购买。花一些时间来看看同一笔商业票据由于此次**购买** 交易而发生了怎样的变

化：

```
Issuer = MagnetoCorp
Paper = 00001
Owner = DigiBank
Issue date = 31 May 2020
Maturity date = 30 November 2020
Face value = 5M USD
Current state = trading
```

最重大的变化是 `Owner`–看看最初由 `MagnetoCorp` 拥有的论文如何由拥有 `DigiBank`。我们可以想象该纸张随后如何出售给 BrokerHouse 或 HedgeMatic，并进行相应的更改 `Owner`。请注意如何使我们能够轻松地识别出该文件现在是。`Current statetrading`

6 个月后，如果 DigiBank 仍然持有该商业票据，则可以使用 MagnetoCorp 赎回它：

```
Issuer = MagnetoCorp
Paper = 00001
Owner = MagnetoCorp
Issue date = 31 May 2020
Maturity date = 30 November 2020
Face value = 5M USD
Current state = redeemed
```

最后的**赎回**交易已经结束了商业票据的生命周期-可以认为它已经完成。保留赎回的商业票据的记录通常是强制性的，并且 `redeemed` 状态允许我们快速识别这些票据。通过 `Owner` 将纸的价值与交易创建者的身份进行比较，可以将其价值用于对**赎回**交易执行访问控制 `Owner`。Fabric 通过 `getCreator()` chaincode API 支持此 功能。如果将 golang 用作链码语言， 则可以使用客户端身份链码库来检索事务创建者的其他属性。

2. 交易次数

我们已经看到，纸张 00001 的生命周期相对简单-它在之间移动 `issued`， `trading` 并且 `redeemed` 是由于 **issue**， **buy** 或**赎回**交易的结果。

这三个事务是由 MagnetoCorp 和 DigiBank（两次）发起的，它们驱动了纸张 00001 的状态变化。让我们更详细地了解影响本文的事务：

4.1. 问题

检查由 MagnetoCorp 发起的第一笔交易：

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

了解**发行**交易如何具有带有属性和值的结构。这项交易的结构是不同的，但密切匹配，纸 00001 的结构是因为他们是不同的东西-纸 00001 反映 PaperNet 的状态是，结果**问题**的交易。拥有这些属性并创建此文件的**问题**事务（我们看不到）背后的逻辑。因为交易**产生**了文件，所以这意味着这些结构之间存在非常密切的关系。

涉及**发行**事务的唯一组织是 MagnetoCorp。自然，MagnetoCorp 需要签署该交易。通常，论文发行人需要在发行新论文的交易上签字。

4.2. 购买

接下来，检查将纸张 00001 的所有权从 MagnetoCorp 转移到 DigiBank 的**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
```

看看**购买**交易如何具有较少的属性，这最终在本文中得出。这是因为此事务仅**修改**了本文。只是由于这次交易而改变；其他一切都一样。没关系–关于**购买**交易，最重要的是所有权的变更，实际上，在此交易中，已经确认了文件的当前所有者 MagnetoCorp。`New owner = DigiBank`

您可能会问为什么在纸张 00001 中没有捕获和属性？这又回到了交易与票据之间的区别。494 万美元的价格标签实际上是交易的财产，而不是本文的财产。花一点时间思考这种差异；它并不像看起来那样明显。稍后我们将看到分类帐将记录这两条信息-影响本文的所有交易的历史记录及其最新状态。明确区分信息非常重要。`Purchase timePrice`

还值得记住的是，纸张 00001 可能会被多次买卖。尽管在我们的方案中略有提前，但让我们研究一下如果纸张 00001 更改所有权时我们**可能会**看到哪些事务。

如果我们有 BigFund 购买：

```
Txn = buy
Issuer = MagnetoCorp
Paper = 00001
Current owner = DigiBank
New owner = BigFund
Purchase time = 2 June 2020 12:20:00 EST
Price = 4.93M USD
```

随后由 HedgeMatic 购买：

```
Txn = buy
Issuer = MagnetoCorp
Paper = 00001
Current owner = BigFund
New owner = HedgeMatic
Purchase time = 3 June 2020 15:59:00 EST
Price = 4.90M USD
```

查看纸张所有者如何变化，以及在我们的示例中价格如何变化。您能想到 MagnetoCorp 商业用纸价格可能下跌的原因吗？

直觉上，**购买**交易要求卖方和购买组织都必须签署此类交易，以便有证据证明作为交易一部分的双方之间的相互协议。

4.3. 赎回

纸张 00001 的**兑换**事务表示其生命周期的结束。在我们相对简单的示例中，HedgeMatic 发起了交易，该交易将商业票据转移回了 MagnetoCorp：

```
Txn = redeem
Issuer = MagnetoCorp
Paper = 00001
Current owner = HedgeMatic
Redeem time = 30 Nov 2020 12:00:00 EST
```

同样，请注意**兑换**交易的属性很少。赎回交易逻辑可以对纸张 00001 进行所有更改以计算数据：`Issuer` 将会成为新所有者，而将会变为 。该属性是在我们的示例中指定的，因此可以对照当前纸张持有者检查该属性。`Current stateredeemedCurrent owner`

从信任的角度来看，**购买**交易的相同理由也适用于**赎回**指令：交易中涉及的两个组织都必须在该交易上签字。

3. 帐本

在本主题中，我们已经看到了交易和结果票据状态是 PaperNet 中两个最重要的概念。的确，我们将在任何 Hyperledger Fabric 分布式分类帐中看到这两个基本元素 -一个世界状态，其中包含所有对象的当前值；一个区块链，记录导致当前世界状态的所有交易的历史记录。

交易所需的签字通过规则强制执行，在将交易追加到分类帐之前对其进行评估。仅当存在必需的签名时，Fabric 才会接受有效的交易。

您现在处在一个绝佳的位置，可以将这些想法转化为明智的合同。如果您的程序设计有些生锈，请不要担心，我们将提供提示和指针来理解程序代码。掌握商业票据智能合约是设计您自己的应用程序的第一步。或者，如果您是对一些编程感到满意的业务分析师，请不要害怕继续深入研究！

五.流程和数据设计

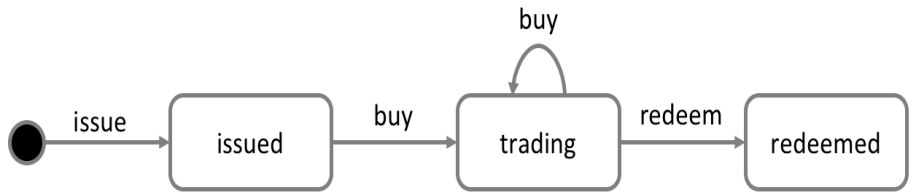
受众：建筑师，应用程序和智能合约开发人员，业务专业人员

本主题向您展示如何在 PaperNet 中设计商业票据流程及其相关的数据结构。我们的分析强调，使用状态和事务对 PaperNet 进行建模可以提供一种精确的方式来了解正在发生的事情。现在，我们将详细说明这两个紧密相关的概念，以帮助我们随后设计 PaperNet 的智能合约和应用程序。

1. 生命周期

如我们所见，在处理商业票据时，有两个重要的概念与我们有关。**状态**和**交易**。确实，对于*所有*区块链用例都是如此 。有一些以状态为模型的价值概念对象，其生命周期过渡由事务描述。对状态和事务进行有效分析是成功实施的重要起点。

我们可以使用状态转换图来表示商业票据的生命周期：



*商业票据的状态转换图。商业票据通过**发行**，**购买**和**赎回**交易在**发行**， **交易**和**赎回**状态之间转换。*

查看状态图如何描述商业票据如何随时间变化，以及特定交易如何控制生命周期过渡。在 Hyperledger Fabric 中，智能合约实现了交易逻辑，可以在不同状态之间转换商业票据。商业票据状态实际上是在分类帐世界状态中保存的；因此，让我们仔细看看它们。

2. 分类帐状态

回忆一下商业票据的结构：

```
Issuer: MagnetoCorp
Paper: 00001
Owner: DigiBank
Issue date: 31 May 2020
Maturity date: 30 Nov 2020
Face value: 5M USD
Current state: trading
```

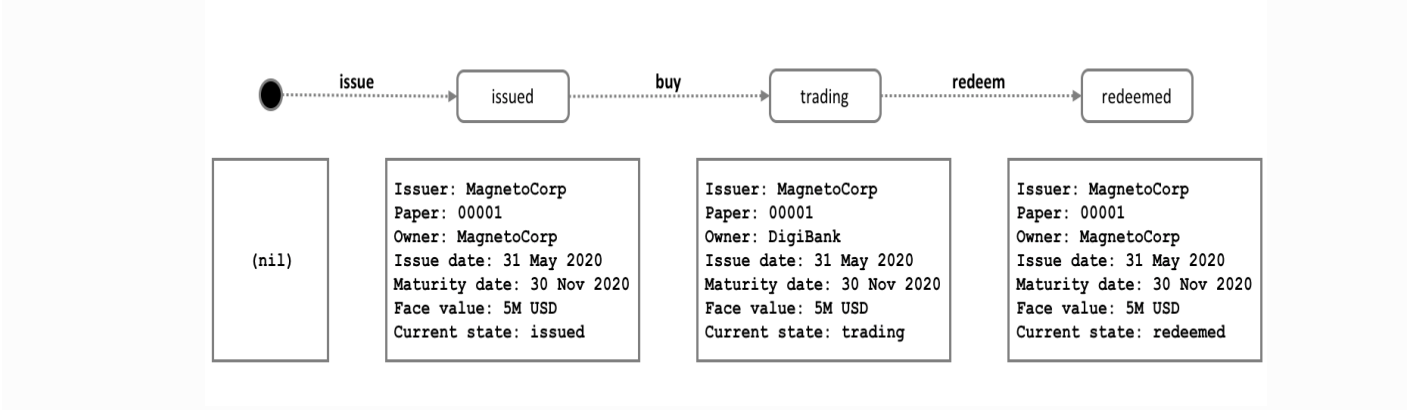
商业票据可以表示为一组属性，每个属性都有一个值。通常，这些属性的某种组合将为每张纸提供唯一的密钥。

了解商业票据 `Paper` 财产如何具有价值 `00001`，以及该财产具有价值。最重要的是 属性指示商业票据是否，或 。结合起来，全套属性构成了商业票据的**状态**。而且，这些单独的商业票据状态的整个集合构成了分类帐 世界状态。 `Face value5M USDCurrent stateissuedtradingredeemed`

所有分类帐州都共享此表格；每个都有一组属性，每个属性都有不同的值。状态的这种*多属性*方面是一个强大

的功能–它使我们可以将 Fabric 状态视为向量而不是简单的标量。然后，我们将有关整个对象的事实表示为单独的状态，随后这些状态将经历由事务逻辑控制的转换。Fabric 状态被实现为键/值对，其中，值以捕获对象的多个属性（通常为 JSON）的格式对对象属性进行编码。该台账数据库可以支持针对这些特性，它是复杂的对象检索非常有帮助的先进的查询操作。

了解如何将 MagnetoCorp 的论文 `00001` 表示为根据不同交易刺激而转变的状态向量：



商业票据状态由于不同的交易而存在并转变。Hyperledger Fabric 状态具有多个属性，使其成为向量而不是标量。

请注意，每张纸都是如何从空状态开始的，从技术上来说，这是 `nil` 纸的状态，因为它不存在！了解**发行**交易如何使纸张 `00001` 存在，以及随后的**购买**和**赎回**交易如何对其进行更新 。

注意每个状态如何自我描述；每个属性都有一个名称和一个值。尽管目前我们所有的商业用纸都具有相同的属性，但并非总是如此，因为 Hyperledger Fabric 支持具有不同属性的不同状态。这允许同一分类帐世界状态包含同一资产的不同形式以及不同类型的资产。这也使得更新状态的结构成为可能。想象一个需要附加数据字段的新法规。灵活的状态属性支持随时间推移数据演变的基本要求。

3. 状态键

在大多数实际应用中，状态将具有在给定上下文中唯一标识状态的属性组合-这是**关键**。PaperNet 商业票据的密钥是由 `Issuer` 和 `paper` 属性的串联形成的。因此对于 MagnetoCorp 的第一篇论文来说就是 `MagnetoCorp00001`。

状态键使我们能够唯一地识别论文。它是根据**发行**交易创建的，随后通过**购买**和**赎回**进行更新。Hyperledger Fabric 要求分类账中的每个状态都有唯一的密钥。

如果在可用属性集中没有唯一键，则将应用程序确定的唯一键指定为创建状态的事务的输入。此唯一密钥通常带有某种形式的 UUID，尽管可读性较低，但这是一种标准做法。重要的是，分类帐中的每个状态对象都必须具有唯一的密钥。

注意：应避免在键中使用 U + 0000（无字节）。

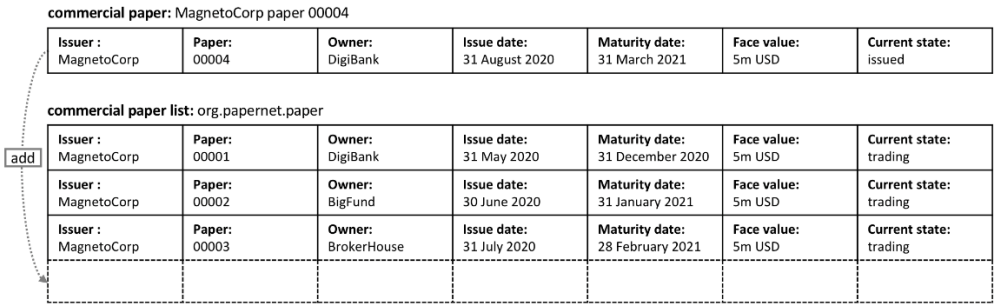
4. 多种状态

如我们所见，PaperNet 中的商业票据作为状态向量存储在分类帐中。能够从分类账查询不同的商业票据是合理的要求；例如：查找由 MagnetoCorp 发布的所有论文，或：查找该 `redeemed` 州由 MagnetoCorp 发布的所有论文。

为了使这些搜索任务成为可能，将所有相关论文归为一个逻辑列表将很有帮助。PaperNet 的设计结合了商业票据清单的概念–一个逻辑容器，每当发行商业票据或进行其他更改时，逻辑容器都会更新。

4.1. 逻辑表示

将所有 PaperNet 商业论文都放在一个商业论文列表中会很有帮助：

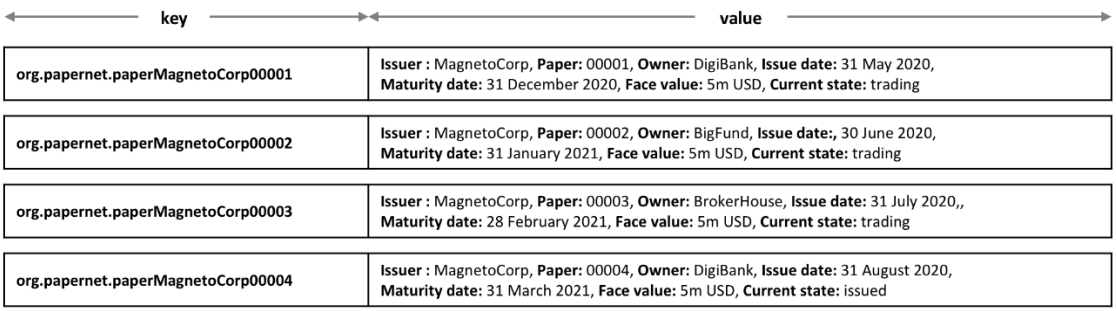


MagnetoCorp 新创建的商业用纸 00004 已添加到现有商业用纸列表中。

发行事务可以将新论文添加到列表中，并且可以使用**购买**或**赎回** 事务来更新列表中已有的论文。在列表中看到如何有一个描述性的名称：`org.papernet.papers`；使用这种 DNS 名称确实是个好主意，因为精心选择的名称将使您的区块链设计对其他人来说很直观。这个想法同样适用于智能合约名称。

4.2. 物理表示

虽然在 PaperNet 中只考虑一个文件 `org.papernet.papers` 列表是正确的- 最好将列表实现为一组单独的 Fabric 状态，其组合键将状态与其列表相关联。这样，每个状态的组合键都是唯一的，并支持有效的列表查询。



将 PaperNet 商业论文的列表表示为一组不同的 Hyperledger Fabric 状态

请注意，在列表中的每个纸张以向量表示的状态，用独特的**复合材料**通过串接形成键 `org.papernet.paper`， `Issuer` 和 `Paper` 特性。此结构很有用，其原因有两个：

- 它允许我们检查分类账中的任何状态向量，以确定其在哪个列表中，而无需参考单独的列表。这类似于看一组体育迷，并根据所穿衬衫的颜色来确定他们支持的球队。体育迷们自称忠诚。我们不需要粉丝列表。
- Hyperlegder Fabric 在内部使用并发控制机制 来更新分类帐，这样将纸张保持在单独的状态向量中就大大减少了共享状态冲突的机会。这样的冲突需要重新提交事务，使应用程序设计复杂化，并降低性能。

第二点实际上是 Hyperledger Fabric 的关键要素。状态向量的物理设计对于优化性能和行为**非常重要**。将您的州分开！

5. 信任关系

我们已经讨论了网络中的不同角色（例如发行人，交易者或评估机构）以及不同的商业利益如何决定谁需要签署交易。在 Fabric 中，这些规则由所谓的背书策略捕获 。可以在链码粒度以及各个状态键上设置规则。

这意味着在 PaperNet 中，我们可以为整个命名空间设置一个规则，该规则确定哪些组织可以发布新论文。之后，可以为各个论文设置和更新规则，以捕获购买和赎回交易的信任关系。

在下一个主题中，我们将向您展示如何结合这些设计概念来实现 PaperNet 商业用纸智能合约，然后在其中进行

应用！

六. 智能合约处理

受众：建筑师，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，以及将纸张从一种状态转换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施一个现实世界的智能合约，该合约管理着发行，购买和赎回商业票据的过程。

我们将介绍：

- 什么是智能合约及其重要性
- 如何定义智能合约
- 如何定义交易
- 如何进行交易
- 如何在智能合约中表示业务对象
- 如何在分类帐中存储和检索对象

如果愿意，您可以下载示例，甚至可以在本地运行。它是用 JavaScript 和 Java 编写的，但是逻辑是完全独立于语言的，因此您可以轻松地看到正在发生的事情！（该示例也将适用于 Go。）

1. 智能合约

智能合约定义业务对象的不同状态，并管理在这些不同状态之间移动对象的流程。智能合约之所以重要，是因为它们使架构师和智能合约开发人员能够定义关键业务流程和数据，这些关键业务流程和数据在区块链网络中进行协作的不同组织之间共享。

在 PaperNet 网络中，智能合约由 MagnetoCorp 和 DigiBank 等不同的网络参与者共享。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。

2. 实现语言

支持两个运行时，即 Java 虚拟机和 Node.js。这使您有机会使用 JavaScript，TypeScript，Java 或可以在这些受支持的运行时之一中运行的任何其他语言之一。

在 Java 和 TypeScript 中，注释或修饰符用于提供有关智能合约及其结构的信息。这样可以提供更丰富的开发经验-例如，可以强制执行作者信息或返回类型。在 JavaScript 中，必须遵循约定，因此，围绕自动确定的内容存在限制。

JavaScript 和 Java 都给出了示例。

3. 合同类别

PaperNet 商业用纸智能合约的副本包含在一个文件中。使用浏览器查看它，或者如果已下载，则在您喜欢的编辑器中将其打开。

- `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 必须就他们将要使用的智能合约的版本达成协议。现在，使用哪个组织的副本都无所谓，它们都是相同的。

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

其中 JavaScript

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

JAVA

本 `CommercialPaperContract` 类包含商业票据交易的定义- **问题**，**购买** 和**赎回**。正是这些交易使商业票据得以存在并在其生命周期中移动。我们将很快检查这些 事务，但是现在对于 JavaScript 来说，`CommericalPaperContract` 扩展了 Hyperledger Fabric `Contract` 类。

对于 Java，该类必须用 `@Contract(...)` 注释修饰。这提供了提供有关合同的其他信息的机会，例如许可证和作

者。该`@Default()`注释表明，该合同类是默认的合同类。能够将合同类别标记为默认合同类别在某些具有多个合同类别的智能合约中很有用。

如果您使用的是 TypeScript 实现，那么会有类似的`@Contract(...)`注释实现与 Java 中相同的目的。

有关可用注释的更多信息，请查阅可用的 API 文档：

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

这些类，注释和`Context`类早已纳入范围：

JavaScript

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

爪哇

我们的商业票据合同将使用这些类的内置功能，例如自动方法调用，每个事务上下文，事务处理程序和类共享状态。

还请注意，JavaScript 类构造函数是如何使用其超类使用 显式协定名称进行初始化的：

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

对于 Java 类，构造函数为空白，因为可以在`@Contract()`注释中指定显式协定名称。如果不存在，则使用该类的名称。

最重要的是，`org.papernet.commercialpaper`它具有非常强的描述性-该智能合约是所有 PaperNet 组织对商业票据的公认定义。

通常，每个文件只有一个智能合约–合约往往具有不同的生命周期，因此将它们分开是明智的。然而，在某些情况下，多个智能合同可能为应用程序提供语法的帮助，例如`EuroBond`，`DollarBond`，`YenBond`，但本质上提供同样的功能。在这种情况下，可以消除智能合约和交易的歧义。

4. 交易定义

在该类中，找到 **issue** 方法。

的 JavaScript

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

爪哇

Java 批注`@Transaction`用于将该方法标记为事务定义。TypeScript 具有等效的注释。

只要将此合同称为`issue`商业票据，就可以控制此功能。回想一下如何通过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风格的变量名，但是看到了这些属性如何几乎直接映射到`issue`方法变量。

`issue`每当应用程序请求发布商业票据时，该方法就会由合同自动授予控制权。交易属性值通过相应的变量可供方法使用。请参阅应用程序主题中的示例应用程序，了解应用程序如何使用 Hyperledger Fabric SDK 提交事务。

您可能已经注意到**问题**定义中有一个额外的变量– `ctx`。这称为事务上下文，并且始终是第一位。默认情况下，它维护与交易逻辑相关的按合同和按交易的信息。例如，它将包含 MagnetoCorp 的指定交易标识符，MagnetoCorp 颁发用户的数字证书以及对分类帐 API 的访问。

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：

的 JavaScript

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

爪哇

此扩展上下文将自定义属性添加`paperList`到默认值：

的 JavaScript

```
class CommercialPaperContext extends Context {

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

爪哇

我们将很快看到如何 `ctx.paperList` 在随后用于帮助存储和检索所有 PaperNet 商业论文的方法。

为了巩固您对智能合约交易结构的理解，找到**购买**和**赎回**交易定义，并查看是否可以看到它们如何映射到其相应的商业票据交易。

该**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
```

的 JavaScript

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

爪哇

在**赎回**交易：

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

的 JavaScript

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

爪哇

在这两种情况下，请观察商业票据交易与智能合约方法定义之间的 1：1 对应关系。

所有 JavaScript 函数都使用 `async` 和 `await` 关键字，从而可以将 JavaScript 函数视为同步函数调用。

5. 交易逻辑

现在，您已经了解了合同的结构和交易的定义，让我们集中讨论智能合同中的逻辑。

回顾第一**期**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
```

结果是**发出**方法被控制了：

的 JavaScript

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

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

  // 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();
}
```

爪哇

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

查看如何 `paperList` 从交易上下文中检索如何提供对商业票据列表的访问。`issue()`，`buy()` 并 `redeem()` 不断进行重新访问 `ctx.paperList` 以使商业用纸的列表保持最新。

购买交易的逻辑更加复杂：

的 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 = '
+paper.getCurrentState());
  }

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

爪哇

了解如何交易的检查 `currentOwner`，并且 `paper` 是 `TRADING` 改变与业主之前 `paper.setOwner(newOwner)`。但是基本流程很简单–检查一些前提条件，设置新所有者，更新分类帐上的商业票据，并将更新后的商业票据（序列化为缓冲区）作为交易响应返回。

您为什么不明白是否可以理解**兑换** 交易的逻辑？

6. 代表一个对象

我们已经看到了如何 使用和类定义和实施**问题**，**购买**和**兑换**交易。通过查看这些类如何工作来结束本主题。`CommercialPaperPaperList`

找到 `CommercialPaper` 课程：

的 JavaScript 在 paper.js 文件中：

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

爪哇

此类包含商业票据状态的内存表示形式。查看该 `createInstance` 方法如何使用提供的参数初始化新的商业票据：

的 JavaScript

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

爪哇

回顾**发行**事务如何使用此类：

的 JavaScript

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

爪哇

了解每次调用发行交易的方式，都会创建一个包含交易数据的新的商业票据内存实例。

需要注意的几个要点：

- 这是内存中的表示形式；稍后我们将看到它如何显示在分类帐中。
- 本 `CommercialPaper` 类扩展 `State` 类。`State` 是一个应用程序定义的类，它为状态创建通用抽象。所有状态都有一个它们表示的业务对象类，一个复合键，可以序列化和反序列化等等。`State` 当我们在分类账上存储多个业务对象类型时，有助于使代码更清晰易读。检查文件中的 `State` 类。`state.js`
- 纸张在创建时会计算自己的密钥-访问分类帐时将使用此密钥。密钥是由 `issuer` 和组成的 `paperNumber`。

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

- 纸张是 `ISSUED` 通过事务而不是纸张类别转移到状态的。这是因为，智能合约支配着论文的生命周期状态。例如，一项 `import` 交易可能会在该 `TRADING` 州立即创建一组新文件。

`CommercialPaper` 该类的其余部分包含简单的帮助程序方法：

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

回忆一下智能合约如何使用这种方法在商业票据的生命周期中移动。例如，在**兑换** 交易中，我们看到：

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

7. 存取分类帐

现在 `PaperList`，在 `paperlist.js` 文件中找到该类：

```
class PaperList extends StateList {
```

该实用程序类用于管理 Hyperledger Fabric 状态数据库中的所有 PaperNet 商业用纸。PaperList 数据结构在体系结构主题中有更详细的描述。

与 `CommercialPaper` 该类一样，该类扩展了应用程序定义的 `StateList` 类，该类为状态列表创建通用抽象-在这种情况下，是 PaperNet 中的所有商业论文。

该 `addPaper()` 方法是该方法的简单饰面 `StateList.addState()`：

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

您可以在 `StateList.js` 文件中看到 `StateList` 该类 如何使用 Fabric API `putState()` 将商业票据作为状态数据写入分类帐中：

```
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);
}
```

分类帐中的每个状态数据都需要以下两个基本元素：

- **密钥**：`key` 由 `createCompositeKey()` 固定名称和的密钥组成 `state`。名称是在 `PaperList` 构造对象时分配的，并 `state.getSplitKey()` 确定每个状态的唯一键。

- **数据**：`data` 只是使用 `State.serialize()` 实用程序方法创建的商业票据状态的序列化形式。在 `State` 类和序列化使用 JSON 反序列化数据，以及该国的业务对象类的要求，在我们的例子 `CommercialPaper` 中，当再次设定 `PaperList` 对象构建。

请注意，a 如何 `StateList` 不存储有关单个状态或状态总列表的任何内容，而是将所有这些都委派给 Fabric 状态数据库。这是一种重要的设计模式–减少了 Hyperledger Fabric 中分类账 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);
}
```

看看他们如何使用面料的 API `putState()`， `getState()` 并 `createCompositeKey()` 访问总帐。稍后，我们将扩展此智能合约以在 paperNet 中列出所有商业票据-实施此分类账检索的方法将是什么样？

在本主题中，您已经了解了如何为 PaperNet 实施智能合约。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。

七.应用

受众：建筑师，应用程序和智能合约开发人员

应用程序可以通过向分类账提交交易或查询分类账内容来与区块链网络进行交互。本主题涵盖了应用程序如何执行此操作的机制。在我们的方案中，组织使用调用商业票据智能合约中定义的**发行**，**购买**和**赎回**交易的应用程序访问 PaperNet 。尽管 MagnetoCorp 发行商业票据的申请是基本的，但它涵盖了所有主要的理解要点。

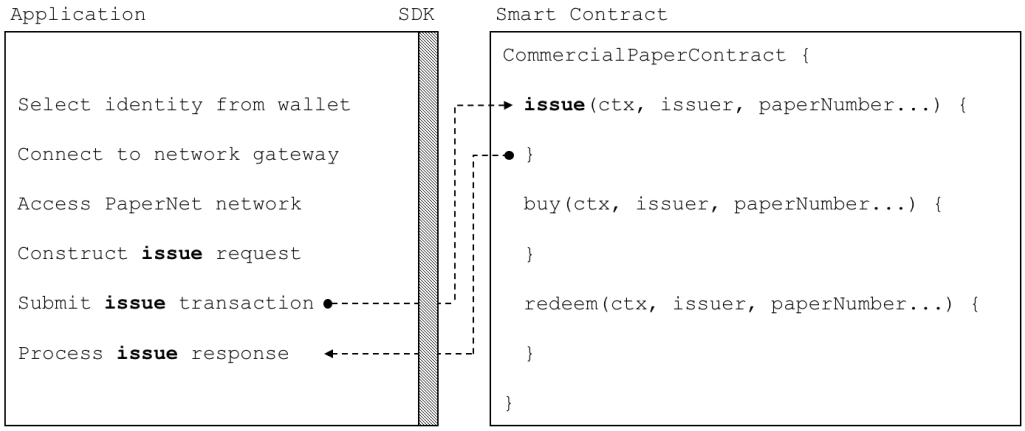
在本主题中，我们将介绍：

- 调用智能合约的应用程序流程
- 应用程序如何使用钱包和身份
- 应用程序如何使用网关进行连接
- 如何访问特定的网络
- 如何构造交易请求
- 如何提交交易
- 如何处理交易响应

为了帮助您理解，我们将参考 Hyperledger Fabric 随附的商业纸样应用程序。您可以下载它并在本地运行它。它是用 JavaScript 和 Java 编写的，但是逻辑是完全独立于语言的，因此您可以轻松地看到正在发生的事情！（该示例也将适用于 Go。）

1. 基本流程

应用程序使用 Fabric SDK 与区块链网络进行交互。这是应用程序如何调用商业票据智能合约的简化图：



PaperNet 应用程序调用商业票据智能合约以提交发行交易请求。

申请必须遵循六个基本步骤才能提交交易：

- 从钱包中选择一个身份
- 连接到网关
- 访问所需的网络
- 构建智能合约的交易请求
- 将交易提交到网络
- 处理回应

您将看到典型的应用程序如何使用 Fabric SDK 执行这六个步骤。您将在 `issue.js` 文件中找到应用程序代码。在浏览器中查看它，如果已下载，则在您喜欢的编辑器中将其打开。花一些时间查看应用程序的整体结构；即使有注释和空格，也只有 100 行代码！

2. 钱包

在的顶部 `issue.js`，您将看到两个 Fabric 类进入了作用域：

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

您可以 `fabric-network` 在 node SDK 文档中阅读有关这些类的信息，但是现在，让我们看看如何使用它们将 MagnetoCorp 的应用程序连接到 PaperNet。该应用程序使用 Fabric **Wallet** 类，如下所示：

```
const wallet = new FileSystemWallet('../identity/user/isabella/wallet');
```

查看如何在本地文件系统中 `wallet` 找到钱包。从钱包中检索到的身份显然适用于正在使用该 `issue` 应用程序的名为 Isabella 的用户。钱包包含一组身份（X.509 数字证书），可用于访问 PaperNet 或任何其他 Fabric 网络。如果您运行本教程，并查看此目录，则将看到 Isabella 的身份凭证。

想象一下一个钱包，里面装有政府身份证，驾驶执照或 ATM 卡的数字等效物。其中的 X.509 数字证书将使持有者与组织相关联，从而使他们具有网络通道中的权利。例如，`Isabella` 可能是 MagnetoCorp 的管理员，这可能给她比 `Balaji` 来自 DigiBank 的其他用户更多的特权。此外，智能合约可以使用交易上下文在智能合约处理期间检索此身份。

还要注意，钱包不持有任何形式的现金或代币-它们持有身份。

3. 网关

第二个关键类是结构**网关**。最重要的是， 网关标识一个或多个提供对网络访问权限的对等点-在我们的示例中为 PaperNet。查看如何 `issue.js` 连接到其网关：

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

`gateway.connect()` 有两个重要参数：

- **connectionProfile**：连接配置文件的文件系统位置，该 连接配置文件将一组对等方标识为 PaperNet 的网关
- **connectionOptions**：一组用于控制 `issue.js` 与 PaperNet 交互方式的选项

了解客户端应用程序如何使用网关将自身与网络拓扑隔离，这可能会发生变化。网关负责使用连接配置文件和连接选项将交易建议发送到网络中正确的对等节点 。

花一些时间检查连接 配置文件 `./gateway/connectionProfile.yaml`。它使用 YAML，使其易于阅读。

它已加载并转换为 JSON 对象：

```
let connectionProfile = yaml.safeLoad(file.readFileSync('./gateway/connectionProfile.yaml', 'utf8'));
```

目前，我们仅对个人资料的 `channels:` 和 `peers:` 部分感兴趣：（我们对细节进行了一些修改，以更好地解释发生了什么。）

```
  channels:
  papernet:
    peers:
      peer1.magnetocorp.com:
        endorsingPeer: true
        eventSource: true

      peer2.digibank.com:
        endorsingPeer: true
        eventSource: true

peers:
  peer1.magnetocorp.com:
    url: grpcs://localhost:7051
    grpcOptions:
      ssl-target-name-override: peer1.magnetocorp.com
      request-timeout: 120
    tlsCACerts:
      path: certificates/magnetocorp/magnetocorp.com-cert.pem

  peer2.digibank.com:
    url: grpcs://localhost:8051
    grpcOptions:
      ssl-target-name-override: peer1.digibank.com
    tlsCACerts:
         path: certificates/digibank/digibank.com-cert.pem
```

了解如何 `channel:` 识别 `PaperNet:` 网络通道及其两个对等端。MagnetoCorp `peer1.magenetocorp.com` 和 DigiBank 具有 `peer2.digibank.com`，两者都具有对同伴的支持作用。通过 `peers:` 密钥链接到这些对等设备，该密钥包含有关如何连接到它们的详细信息，包括它们各自的网络地址。

连接配置文件包含很多信息-不仅包括对等信息-而且还包含网络通道，网络订购者，组织和 CA，因此如果您不了解所有信息，请不要担心！

现在让我们将注意力转向该 connectionOptions 对象：

```
  let connectionOptions = {
  identity: userName,
  wallet: wallet
  }
```

了解如何指定标识 `userName` 和钱包 `wallet` 用于连接网关。这些是在代码的前面分配的值。

应用程序还可以使用其他连接选项来指示 SDK 代表其智能操作。例如：

```
  let connectionOptions = {
  identity: userName,
  wallet: wallet,
  eventHandlerOptions: {
```

```
    commitTimeout: 100,
    strategy: EventStrategies.MSPID_SCOPE_ANYFORTX
  },
    }
```

在这里，`commitTimeout`告诉 SDK 等待 100 秒以了解事务是否已提交。并指定 SDK 可以在单个 MagnetoCorp 对等方确认交易后通知应用程序，而相反，这要求 MagnetoCorp 和 DigiBank 的所有对等方确认交易。

`strategy: EventStrategies.MSPID_SCOPE_ANYFORTXstrategy: EventStrategies.NETWORK_SCOPE_ALLFORTX`

如果您愿意，请阅读更多有关连接选项如何允许应用程序指定面向目标的行为的信息，而不必担心如何实现。

1. 网络渠道

在网关上定义的同龄人`connectionProfile.yaml`提供`issue.js`与接入 PaperNet。因为这些对等方可以加入多个网络通道，所以网关实际上为应用程序提供了对多个网络通道的访问权限！

查看应用程序如何选择特定频道：

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

从那时起，`network`将提供对 PaperNet 的访问。此外，如果应用程序要同时访问另一个网络`BondNet`，则很容易：

```
const network2 = await gateway.getNetwork('BondNet');
```

现在我们的应用程序可以访问到第二网络，`BondNet`同时有`PaperNet`！

我们在这里可以看到 Hyperledger Fabric 的强大功能-应用程序可以通过连接到多个网关对等点来参与**网络网络**，每个网关对等点都连接到多个网络通道。根据中提供的钱包身份，应用程序将在不同渠道中拥有不同的权利`gateway.connect()`。

2. 构造要求

该应用程序现在准备**发布**商业票据。为此，它将`CommercialPaperContract`再次使用，访问此智能合约相当简单：

```
const contract = await network.getContract('papercontract', 'org.papernet.commercialpaper');
```

注意应用程序如何提供名称`papercontract`–和明确的合同名称：`org.papernet.commercialpaper`！我们将看到合同名称如何从`papercontract.js`包含许多合同的链码文件中挑选出一个合同。在 PaperNet 中，`papercontract.js`已使用名称安装并实例化了名称`papercontract`，如果您有兴趣，请阅读如何安装和实例化包含多个智能合约的链码。

如果我们的应用程序同时需要访问 PaperNet 或 BondNet 中的另一个合同，这将很容易：

```
const euroContract = await network.getContract('EuroCommercialPaperContract');

const bondContract = await network2.getContract('BondContract');
```

在这些示例中，请注意我们如何不使用合格合同名称-每个文件只有一个智能合同，并且 `getContract()` 将使用找到的第一个合同。

回顾一下 MagnetoCorp 发行第一张商业票据所使用的交易：

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

现在让我们将此交易提交给 PaperNet！

3. 提交交易

提交交易是对 SDK 的单一方法调用：

```
const issueResponse = await contract.submitTransaction('issue', 'MagnetoCorp', '00001', '2020-05-31', '2020-11-30', '5000000');
```

查看 `submitTransaction()` 参数如何与事务请求的参数匹配。这些值将传递给 `issue()` 智能合约中的方法，并用于创建新的商业票据。回顾其签名：

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

在应用程序发布后不久，智能合约可能会收到控制权 `submitTransaction()`，但事实并非如此。在幕后，SDK 使用 `connectionOptions` 和 `connectionProfile` 详细信息将交易建议发送到网络中正确的对等方，在这里它可以获取所需的认可。但是，应用程序无需担心所有这些问题 `submitTransaction`，它只需要发布就可以了，SDK 会处理一切！

请注意，`submitTransaction` API 包含用于侦听事务提交的过程。需要侦听提交，因为没有提交，您将不知道您的交易是否已成功排序，验证并提交到分类账。

现在让我们将注意力转移到应用程序如何处理响应上！

4. 流程响应

回想一下发行交易 `papercontract.js` 如何返回商业票据响应：

```
return paper.toBuffer();
```

您会注意到一个小怪癖–新的新数据 `paper` 需要在返回到应用程序之前转换为缓冲区。请注意，如何 `issue.js` 使用类方法 `CommercialPaper.fromBuffer()` 为响应缓冲纸补充水分，作为商业论文：

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

这允许 `paper` 在描述性完成消息中以自然的方式使用：

```
console.log(`${paper.issuer} commercial paper : ${paper.paperNumber} successfully issued for value ${paper.faceValue}`);
```

了解 `paper` 在应用程序合约和智能合约中如何使用相同的类–如果您像这样构造代码，它将真正帮助提高可读性和重用性。

与交易建议一样，智能合约完成后，应用程序似乎很快就会收到控制权，但事实并非如此。在幕后，SDK 管理整个共识过程，并根据 `strategy` connectionOption 通知应用程序完成。如果您对 SDK 的幕后操作感兴趣，请阅读详细的 交易流程。

在本主题中，您已经了解了如何通过检查 MagnetoCorp 的应用程序如何在 PaperNet 中发布新的商业论文来从示例应用程序中调用智能合约。现在，检查关键分类账和智能合约数据结构是根据其背后的体系结构主题设计的。

八.应用程序设计元素

本节详细介绍了 Hyperledger Fabric 中的客户端应用程序和智能合约开发的关键功能。对功能的深入了解将帮助您设计和实施高效的解决方案。

- 合约名称
- 链码名称空间
- 交易环境
- 交易处理程序
- 背书政策
- 连接配置文件
- 连接选项
- 钱包
- 网关

九.合约名称

受众：架构师，应用程序和智能合约开发人员，管理员

链码是用于将代码部署到 Hyperledger Fabric 区块链网络的通用容器。在一个链码中定义了一个或多个相关的智能合约。每个智能合约都有一个在链码中唯一标识它的名称。应用程序使用其合同名称在实例化的链码中访问特定的智能合同。

在本主题中，我们将介绍：

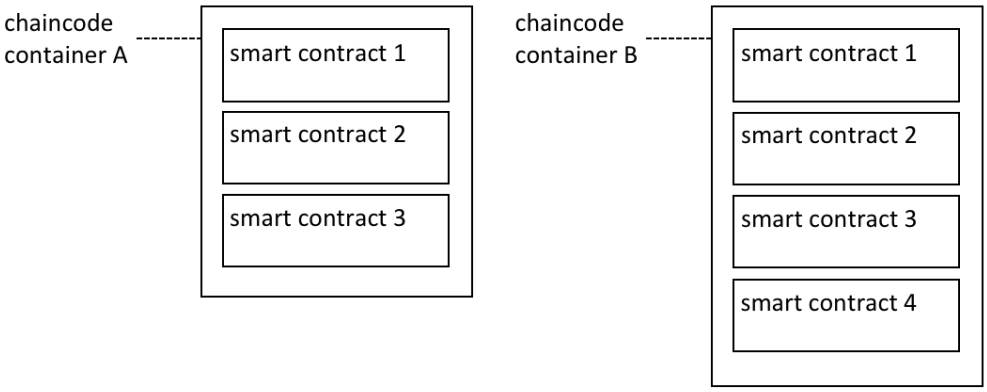
- 链码如何包含多个智能合约
- 如何分配智能合约名称
- 如何通过应用程序使用智能合约

- 默认智能合约

1. 链码

在“开发应用程序”主题中，我们可以看到 Fabric SDK 如何提供高级编程抽象，以帮助应用程序和智能合约开发人员专注于他们的业务问题，而不是如何与 Fabric 网络交互的底层细节。

智能合约是高级编程抽象的一个示例，可以在链码容器中定义智能合约。安装并实例化一个链码后，其中的所有智能合约都可用于相应的通道。



可以在一个链码中定义多个智能合约。每个代码均由其链码中的名称唯一标识。

在上图中，链码 A 中定义了三个智能合约，而链码 B 中包含四个智能合约。了解如何使用链码名称完全限定特定的智能合约。

分类帐结构由一组已部署的智能合约定义。这是因为分类账包含有关网络感兴趣的业务对象（例如 PaperNet 中的商业票据）的事实，并且这些业务对象通过智能合约中定义的交易功能在其生命周期（例如发行，购买，赎回）中移动。 。

在大多数情况下，一个链码中只会定义一个智能合约。但是，将相关智能合约保持在单个链码中是有意义的。例如，在不同的货币计价的可能有合同的商业票据 `EuroPaperContract`，`DollarPaperContract`， `YenPaperContract` 这可能需要保持在它们所部署的通道相互同步。

2. 名称

链码中的每个智能合约均由其合约名称唯一标识。当构造类时，智能合约可以显式分配该名称，或者让 `Contract` 该类隐式分配默认名称。

检查 `papercontract.js` chaincode 文件：

```
class CommercialPaperContract extends Contract {

    constructor() {
        // Unique name when multiple contracts per chaincode file
        super('org.papernet.commercialpaper');
        }
```

查看 `CommercialPaperContract` 构造函数如何将合同名称指定为 `org.papernet.commercialpaper`。结果是，在 `papercontract` 链码中，此智能合约现在与合约名称相关联 `org.papernet.commercialpaper`。

如果未指定显式合同名称，则将分配默认名称-类的名称。在我们的示例中，默认合同名称为 `CommercialPaperContract`。

仔细选择您的名字。不仅每个智能合约都必须具有唯一的名称，一个精心选择的名字很有启发性。具体来说，建议使用显式的 DNS 样式命名约定，以帮助组织清晰且有意义的名称。`org.papernet.commercialpaper` 表示 PaperNet 网络已经定义了标准的商业票据智能合约。

合同名称还有助于消除给定链码中具有相同名称的不同智能合同交易功能的歧义。当智能合约紧密相关时，就会发生这种情况。它们的交易名称将趋于相同。我们可以看到，交易是通过链代码和智能合约名称的组合在渠道内唯一定义的。

合同名称在链码文件中必须唯一。某些代码编辑器将在部署之前检测相同类名的多个定义。无论是否显式或隐式指定具有相同协定名称的多个类，链码都将返回错误。

3. 应用

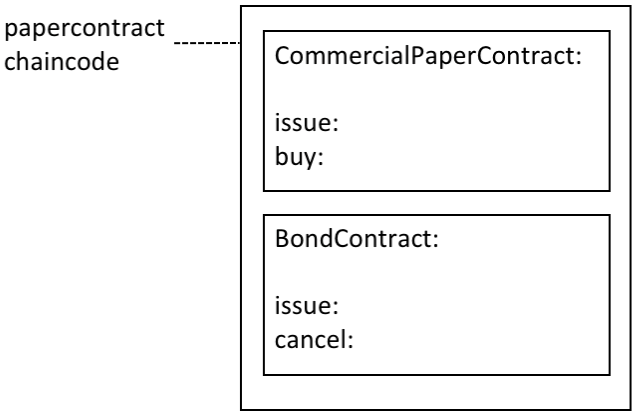
将链码安装到对等方并在通道上实例化后，应用程序即可访问其中的智能合约：

```
const network = await gateway.getNetwork(`papernet`);

const contract = await network.getContract('papercontract', 'org.papernet.commercialpaper');

const issueResponse = await contract.submitTransaction('issue', 'MagnetoCorp', '00001', '2020-05-31', '2020-11-30', '5000000');
```

查看应用程序如何通过该 `network.getContract()` 方法访问智能合约 。该 `papercontract` chaincode 名 `org.papernet.commercialpaper` 返回一个 `contract` 可用于提交交易与发行商业票据参考 `contract.submitTransaction()` API。

4. 违约合同

链码中定义的第一个智能合约称为*默认* 智能合约。默认值很有用，因为链码通常会在其中定义一个智能合约。默认值允许应用程序直接访问这些交易，而无需指定合同名称。



默认智能合约是链码中定义的第一个合约。

在此图中， `CommercialPaperContract` 是默认的智能合约。即使我们有两个智能合约，默认的智能合约也使我们 前面的示例更容易编写：

```
const network = await gateway.getNetwork(`papernet`);

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

const issueResponse = await contract.submitTransaction('issue', 'MagnetoCorp', '00001', '2020-05-31', '2020-11-30', '5000000');
```

之所以可行， `papercontract` 是因为默认的智能合约是， `CommercialPaperContract` 并且它具有 `issue` 交易记录。请注意，只能通过显式寻址来调用 `issue` 事务中的 事务 `BondContract` 。同样，即使 `cancel` 交易是独一无二的，因为 `BondContract` 是*不是*默认的智能合同，也必须明确处理。

在大多数情况下，链码将仅包含一个智能合约，因此，对链码进行仔细的命名可以减少开发人员将链码作为概念来考虑的需求。在上面的示例代码中，感觉就像 `papercontract` 是一个智能合约。

总之，合同名称是一种简单的机制，可以识别给定链码中的各个智能合约。合同名称使应用程序可以轻松找到特定的智能合同并使用它来访问分类账。

十.链码名称空间

受众：架构师，应用程序和智能合约开发人员，管理员

链码名称空间允许它保持其世界状态与其他链码分离。具体来说，具有相同链码的智能合约共享对同一世界状态的直接访问，而具有不同链码的智能合约不能直接访问彼此的世界状态。如果智能合约需要访问另一个链码世界状态，则可以通过执行链码到链码的调用来做到这一点。最后，区块链可以包含与不同世界状态相关的交易。

在本主题中，我们将介绍：

1. 动机

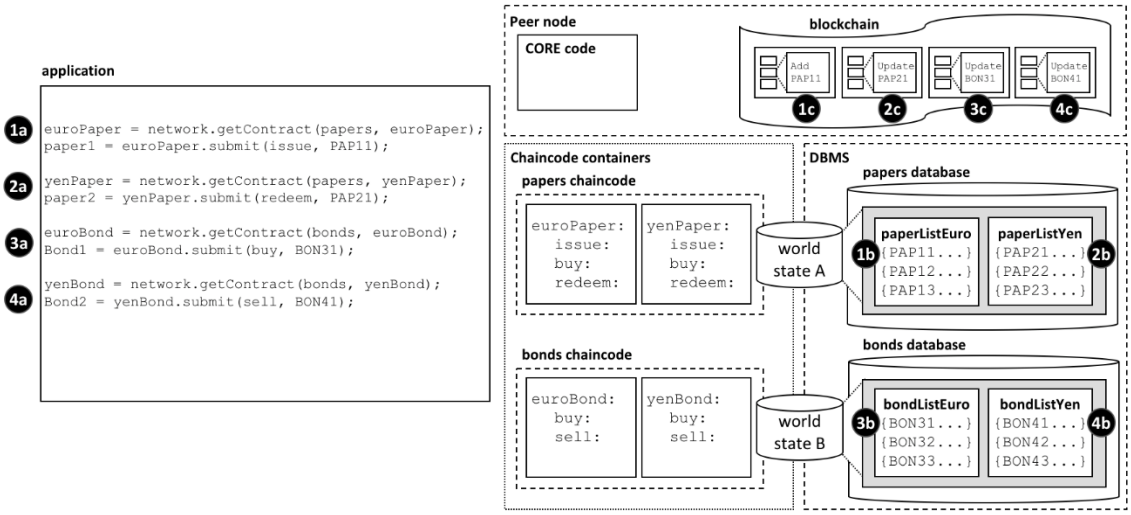
命名空间是一个常见的概念。据我们了解，*公园街，纽约*和 *公园街，西雅图*是即使它们具有相同的名称不同的街道。这个城市形成了公园街的**命名空间**，同时提供了自由和清晰。

在计算机系统中是相同的。命名空间允许不同的用户对共享系统的不同部分进行编程和操作，而不会互相干扰。许多编程语言都有名称空间，因此程序可以自由分配唯一的标识符，例如变量名，而不必担心其他程序会这样做。我们将看到 Hyperledger Fabric 使用名称空间来帮助智能合约将其分类帐世界状态与其他智能合约分开。

2. 情境

让我们使用下图检查分类帐世界状态如何组织有关业务对象的事实，这些事实对组织中的组织很重要。无论这些对象是商业票据，债券还是车辆登记证，以及它们在其生命周期中的任何位置，都将其维护为分类帐世界状态数据库中的状态。智能合约通过与分类帐（世界状态和区块链）进行交互来管理这些业务对象，在大多数情况下，这将涉及查询或更新分类帐世界状态。

了解分类账世界状态是根据访问智能账本的智能合约的链码进行分区的，这一点至关重要，而这种分区或*命名*间隔是架构师，管理员和程序员的重要设计考虑因素。



分类账世界状态根据访问它的链码分为不同的名称空间。在给定通道内，相同链代码中的智能合约共享相同的世界状态，而不同链代码中的智能合约无法直接访问彼此的世界状态。同样，区块链可以包含与不同链码世界状态相关的交易。

在我们的示例中，我们可以看到在两个不同的链码中定义了四个智能合约，每个合约都在自己的链码容器中。该 `euroPaper` 和 `yenPaper` 智能合同中规定 `papers` chaincode。 `euroBond` 和 `yenBond` 智能合约的情况相似-它们在 `bonds` 链码中定义。这种设计可帮助应用程序程序员了解他们使用的是欧元还是日元定价的商业票据或债券，并且由于每种金融产品的规则对于不同的货币并没有真正改变，因此有必要使用相同的链码来管理其部署。

该图还显示了此部署选择的后果。数据库管理系统（DBMS）为 `papers` 和和 `bonds` 链码以及其中包含的智能合约创建不同的世界状态数据库。并分别保存在不同的数据库中；数据彼此隔离，因此单个世界状态查询（例如）无法访问两个世界状态。据说世界状态根据其链码进行了*命名*。 `World state Aworld state B`

请参阅如何包含两个商业文件清单 和。状态和是分别由和合约管理的每份论文的实例。因为它们共享相同的链码名称空间，所以它们的键（）在链码名称空间内必须是唯一的，有点像街道名称在城镇中是唯一的。注意，有可能在链代码中编写智能合约，从而对所有商业票据（无论是欧元还是日元）进行汇总计算，因为它们共享相同的名称空间。债券的情况与此类似–它们被保存在 其中，并映射到一个单独的数据库，并且其键必须唯一。
`world state ApaperListEuropaperListYenPAP11PAP21euroPaperyenPaperPAPxyzpaperspapersworld state Bbonds`

同样重要的是，命名空间意味着 `euroPaper` 与 `yenPaper` 直接无法访问，并且和不能直接访问。这种隔离非常有用，因为商业票据和债券是非常不同的金融工具。它们具有不同的属性，并遵循不同的规则。这也意味着和 可能具有相同的键，因为它们位于不同的命名空间中。这很有帮助；它为命名提供了很大的自由度。使用这种自由来有意义地命名不同的业务对象。
`world state BeuroBondyenBondworld state Apapersbonds`

最重要的是，我们可以看到区块链与在特定通道中运行的对等实体相关联，并且它包含影响和的交易。那是因为区块链是对等体中最基本的数据结构。可以始终从此区块链重新创建世界状态集，因为它们是区块链交易的累积结果。世界国家有助于简化智能合约并提高其效率，因为它们通常仅需要一个国家的当前值。通过名称空间将世界状态分开可以帮助智能合约将其逻辑与其他智能合约隔离，而不必担心对应于不同世界状态的交易。例如，合同无需担心 `world state Aworld state Bbondspaper` 交易，因为它看不到它们产生的世界状态。

还值得注意的是，对等方，chaincode 容器和 DBMS 在逻辑上都是不同的进程。对等方及其所有 chaincode 容器始终在物理上独立的操作系统进程中，但是可以根据其类型将 DBMS 配置为嵌入或独立 。对于 LevelDB，DBMS 完全包含在同级中，但是对于 CouchDB，它是一个单独的操作系统进程。

重要的是要记住，在此示例中，名称空间的选择是业务要求的结果，该业务要求共享不同货币的商业票据，但将它们与债券分开。考虑如何修改名称空间结构以满足业务需求，以使每个金融资产类别保持独立，或共享所有商业票据和债券？

3. 频道

如果对等方加入了多个渠道，则会为每个渠道创建并管理一个新的区块链。而且，每次在新的通道中实例化一个链码时，都会为其创建一个新的世界状态数据库。这意味着通道还与世界状态的链码一起形成一种命名空间。

但是，相同的对等和链码容器过程可以同时加入多个渠道–与区块链和世界状态数据库不同，这些过程不会随着加入的渠道数量而增加。

例如，如果 `papers` 和 `bonds` 链码在新的通道上实例化，将创建一个完全独立的区块链，并创建两个新的世界状态数据库。但是，对等和链码容器不会增加；每个都将连接到多个通道。

4. 用法

让我们使用商业论文示例来说明应用程序如何使用带有名称空间的智能合约。值得注意的是，应用程序与对等方进行通信，并且对等方将请求路由到适当的链码容器，然后容器可以访问 DBMS。该路由由图中所示的对等**核心**组件完成。

这是使用商业票据和债券的应用程序代码，以欧元和日元定价。该代码是不言自明的：

```
    const euroPaper = network.getContract(papers, euroPaper);
paper1 = euroPaper.submit(issue, PAP11);

const yenPaper = network.getContract(papers, yenPaper);
paper2 = yenPaper.submit(redeem, PAP21);

const euroBond = network.getContract(bonds, euroBond);
bond1 = euroBond.submit(buy, BON31);

const yenBond = network.getContract(bonds, yenBond);
    bond2 = yenBond.submit(sell, BON41);
```

查看应用程序如何：

- 使用 指定链码的 API 访问 `euroPaper` 和 `yenPaper` 合同。参见交互点 **1a** 和 **2a**。 `getContract()papers`
- 使用指定链码的 API 访问 `euroBond` 和 `yenBond` 合同。参见交互点 **3a** 和 **4a**。 `getContract()bonds`
- 使用合同将 `issue` 交易提交到网络以获取商业票据。参见相互作用点 **1a**。结果产生了以国家为代表的商业票据; 相互作用点 **1b**。该操作在交互点 **1c** 被捕获为区块链中的事务。 `PAP11euroPaperPAP11world state A`
- 使用合同将 `redeem` 交易提交到网络以获取商业票据。参见交互点 **2a**。结果产生了以国家为代表的商业票据; 交互点 **2b**。该操作在交互点 **2c** 被捕获为区块链中的交易。 `PAP21yenPaperPAP21world state A`
- 使用 合同将 `buy` 交易提交到网络以进行担保。参见相互作用点 **3a**。这导致产生由状态表示的键的在; 相互作用点 **3b**。该操作在交互点 **3c** 被捕获为区块链中的交易。 `BON31euroBondBON31world state B`

- 使用 合同将 `sell` 交易提交到网络以进行担保。参见相互作用点 **4a**。这导致产生由状态表示的键的在; 相互作用点 **4b**。该操作在交互点 **4c** 被捕获为区块链中的交易。
`BON41yenBondBON41world state B`

了解智能合约如何与世界状态交互：

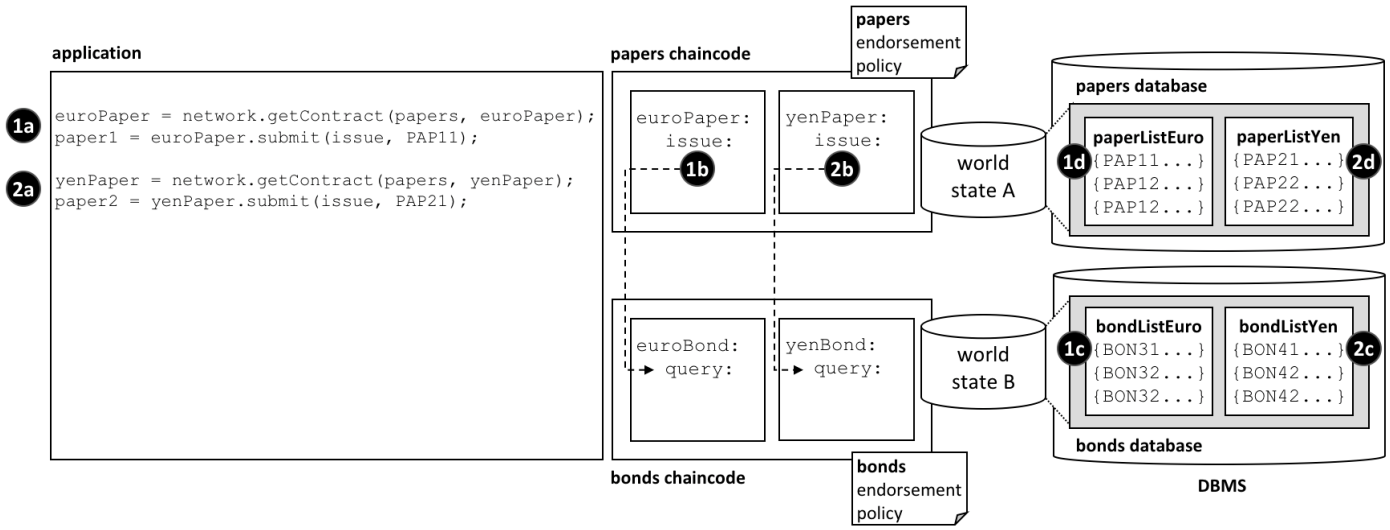
- `euroPaper` 和 `yenPaper` 合同可以直接访问，但不能直接访问。物理地存储在与链码相对应的数据库管理系统（DBMS）中的数据库中。`world state Aworld state BWorld state Apaperspapers`
- `euroBond` 和 `yenBond` 合同可以直接访问，但不能直接访问。物理地存储在与链码相对应的数据库管理系统（DBMS）中的数据库中。`world state Bworld state AWorld state Bbondsbonds`

了解区块链如何捕获世界所有州的交易：

- 互动 **1C** 和 **2C** 对应于交易创建和更新商业票据 `PAP11` 和 `PAP21` 分别。这些都包含在中。
`world state A`
- 互动 **3c** 和 **4c** 对应于更新债券 `BON31` 和的交易 `BON41`。这些都包含在中。`world state B`
- 如果或由于某种原因被销毁，则可以通过重播区块链中的所有交易来重新创建它们。
`world state Aworld state B`

5. 跨链码访问

正如我们在示例场景中看到的，`euroPaper` 并且 `yenPaper` 不能直接访问。这是因为我们已经设计了链代码和智能合约，以使这些链代码和世界状态彼此分开保存。但是，让我们假设需要访问。
`world state BeuroPaperworld state B`
为什么会发生这种情况？想象一下，当发行商业票据时，智能合约想要根据具有相似到期日的债券当前收益对票据定价。在这种情况下，`euroPaper` 合同必须能够查询中的债券价格。查看下图，以了解如何构建此交互。
`world state B`



链码和智能合约如何通过其链码间接访问另一个世界国家。

注意如何：

- 该应用程序 `issue` 在 `euroPaper` 智能合约中提交要发行的交易 `PAP11`。参见互动 **1a**。
- 该 `issue` 事务中的 `euroPaper` 智能合同要求的 `query` 事务中的 `euroBond` 智能合同。参见相互作用点 **1b**。
- 在 `query` 中 `euroBond` 能检索信息。参见相互作用点 **1c**。`world state B`
- 当控制权返回 `issue` 交易时，它可以使用响应中的信息对纸张定价并更新信息。参见相互作用点 **1d**。`world state A`
- 发行以日元计价的商业票据的控制流程相同。参见交互点 **2a**，**2b**，**2c** 和 **2d**。

使用 `invokeChaincode()` API 在链代码之间传递控制。该 API 将控制权从一个链码传递到另一个链码。

尽管在示例中我们仅讨论了查询事务，但是可以调用智能合约来更新被调用的链码的世界状态。请参阅以下注意事项。

6. 注意事项

- 通常，每个链码中都将包含一个智能合约。
- 如果多个智能合约关系密切，则仅应将它们部署在同一链码中。通常，仅当它们共享相同的世界状态时才需要这样做。
- 链码名称空间提供了不同世界状态之间的隔离。通常，将不相关的数据相互隔离是有意义的。请注意，您不能选择链码名称空间。它由 Hyperledger Fabric 分配，并直接映射到链码的名称。
- 为了使使用 `invokeChaincode()` API 的链码与链码交互，两个链码必须安装在同一对等体上。
  - 对于仅需要查询被调用链码的世界状态的交互，调用可以在与调用方链码不同的通道中进行。
  - 对于需要更新被调用链码的世界状态的交互，调用必须与调用者链码在同一通道中。

十一.交易环境

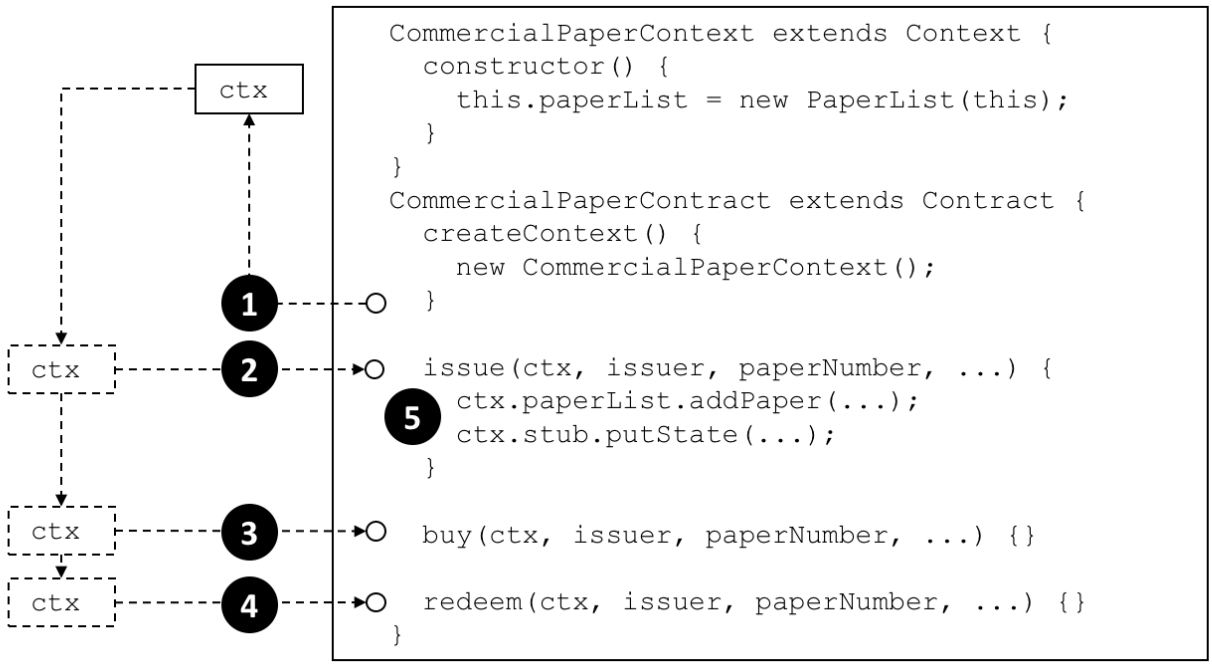
受众：建筑师，应用程序和智能合约开发人员

事务上下文执行两个功能。首先，它允许开发人员在智能合约中跨事务调用定义和维护用户变量。其次，它提供对广泛的 Fabric API 的访问，这些 API 允许智能合约开发人员执行与详细交易处理相关的操作。这些范围从查询或更新分类账（不可变区块链和可修改的世界状态）到检索提交交易的应用程序的数字身份。

实例化智能合约时会创建一个事务上下文，并且该上下文可用于以后的每个事务调用。事务上下文可帮助智能合约开发人员编写功能强大，高效且易于推理的程序。

1. 情境

在商业票据样本中， papercontract 最初定义了其负责的商业票据清单的名称。每个事务随后都引用此列表；发行交易为其添加了新票据，购买交易更改了其所有者，而赎回交易将其标记为完成。这是一种常见的模式。编写智能合约时，在顺序交易中初始化和调用特定变量通常会很有帮助。



智能合约交易上下文允许智能合约在交易调用之间定义和维护用户变量。请参阅文本以获取详细说明。

2. 程式设计

构建智能合约时，开发人员可以选择覆盖内置的 `Context` 类 `createContext` 方法以创建自定义上下文：

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

在我们的示例中， `CommercialPaperContext` 专门用于 `CommercialPaperContract` 。查看通过寻址的自定义上下文如何向其自身 `this` 添加特定变量 `PaperList` ：

```
CommercialPaperContext extends Context {
    constructor () {
        this.paperList = new PaperList(this);
    }
}
```

当在点的 createContext（）方法返回（**1**）在图中 上方，自定义上下文 `ctx` 已经被创建，其包含 `paperList` 作为其变量之一。

随后，每当调用诸如签发，购买或赎回之类的智能合约交易时，该上下文便会传递给它。请参阅如何在第（**2**），（**3**） 和（**4**）点使用 `ctx` 变量将相同的商业票据上下文传递到交易方法中。

查看在第（**5**）点如何使用上下文：

```
ctx.paperList.addPaper(...);
ctx.stub.putState(...);
```

请注意，`paperList`创建`CommercialPaperContext`事务对发行事务可用。了解**赎回**和**购买**交易如何`paperList`类似使用；使智能合约高效且易于推理。`ctx`

您还可以看到上下文中还有另一个元素`ctx.stub`–并没有明确添加`CommercialPaperContext`。这是因为`stub`和其他变量是内置上下文的一部分。现在，让我们检查一下内置上下文的结构，这些隐式变量以及如何使用它们。

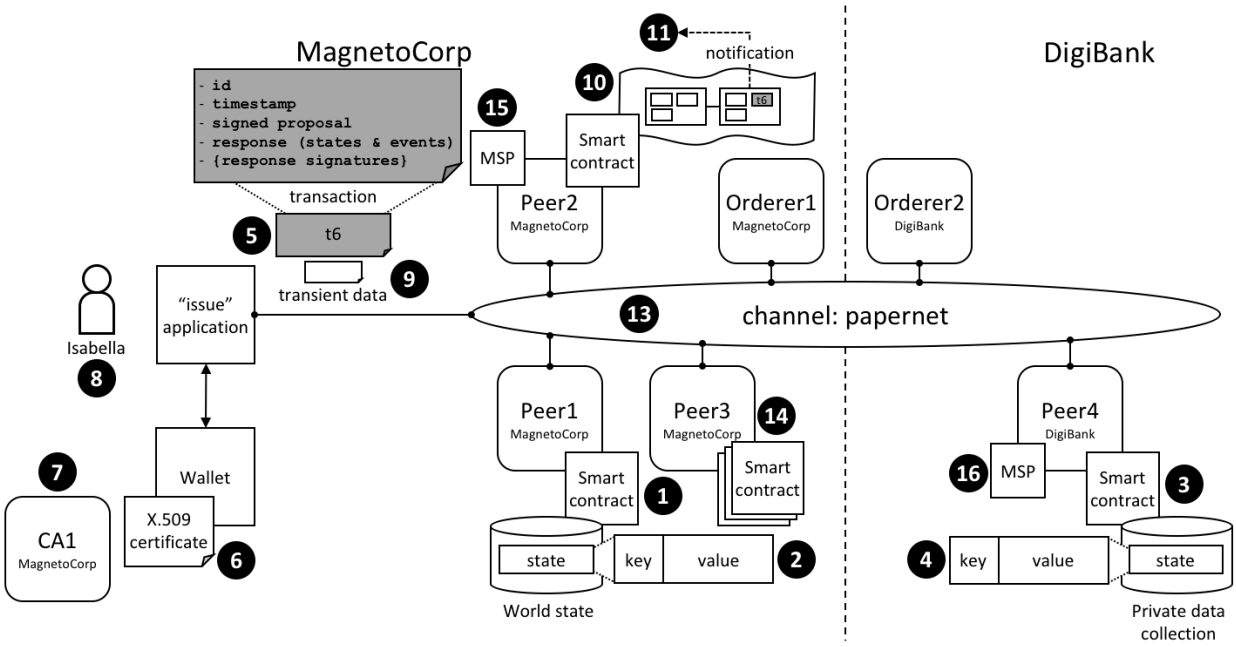
3. 结构体

从示例中可以看出，事务上下文可以包含任意数量的用户变量，例如`paperList`。

事务上下文还包含两个内置元素，它们提供对广泛的 Fabric 功能的访问，从提交事务的客户端应用程序到分类帐访问。

- `ctx.stub`用于访问 API，这些 API 从分类账提供各种交易处理操作`putState()`并`getState()`访问分类账，`getTxID()`以检索当前交易 ID。
- `ctx.clientIdentity`用于获取有关提交交易的用户身份的信息。

我们将使用下图向您展示使用`stub`和`clientIdentity`使用可用的 API 可以执行的智能合约：



智能合约可以通过事务上下文`stub`和访问智能合约中的一系列功能`clientIdentity`。请参阅文本以获取详细说明。

4. 存根

存根中的 API 分为以下几类：

- **世界状态数据 API**。请参阅交互点（**1**）。这些 API 使智能合约可以使用其密钥从世界状态获取，放置和删除与各个对象相对应的状态：

  - getState（）
  - putState（）
  - deleteState（）

这些基本 API 由查询 API 补充，查询 API 使合同能够检索一组状态而不是单个状态。请参阅交互点（**2**）。该集合可以通过使用完整或部分键的键值范围来定义，也可以根据基础世界状态数据库中的值进行查询 。对于大型查询，可以对结果集进行分页以减少存储需求：

  - getStateByRange（）
  - getStateByRangeWithPagination（）
  - getStateByPartialCompositeKey（）
  - getStateByPartialCompositeKeyWithPagination（）
  - getQueryResult（）
  - getQueryResultWithPagination（）

- **私有数据 API**。请参阅交互点（**3**）。这些 API 使智能合约能够与私有数据收集进行交互。它们类似于用于世界状态交互的 API，但用于私有数据。有一些 API 可以通过其键获取，放置和删除私有数据状态：

  - getPrivateData（）
  - putPrivateData（）
  - deletePrivateData（）

此集合由一组 API 补充，以查询私有数据（**4**）。这些 API 允许智能合约根据一系列键值（全部或部分键）从私有数据集合中检索一组状态，或者根据底层世界状态数据库中的值进行查询 。当前没有用于私有数据收集的分页 API。

  - getPrivateDataByRange（）
  - getPrivateDataByPartialCompositeKey（）
  - getPrivateDataQueryResult（）

- **交易 API**。请参阅交互点（**5**）。智能合约使用这些 API 来检索有关智能合约正在处理的当前交易建议的详细信息。这包括交易标识符和创建交易建议的时间。

  - getTxID（）返回当前交易建议（**5**）的标识符。
  - getTxTimestamp（）返回应用程序（**5**）创建当前交易建议时的时间戳。
  - getCreator（）返回交易建议创建者的原始身份（X.509 或其他）。如果这是 X.509 证书，则通常更适合使用 `ctx.ClientIdentity`。
  - getSignedProposal（）返回智能合约正在处理的当前交易提议的签名副本。

- getBinding（） 用于防止使用随机数恶意或意外重播事务。（出于实际目的，随机数是由客户端应用程序生成的并包含在加密哈希中的随机数。）例如，智能合约可以在（**1**）使用此 API 来检测交易的重播（**5**）。
- getTransient（） 允许智能合约访问应用程序传递给智能合约的瞬态数据。请参阅交互点（**9**）和（**10**）。临时数据是应用程序-智能合约交互的私有数据。它没有记录在分类帐中，通常与私人数据收集（**3**）结合使用。

- 智能合约使用**密钥 API** 来操纵世界状态或私有数据收集中的状态密钥。请参阅交互点 **2** 和 **4**。

这些 API 中最简单的 API 允许智能合约从其单独的组件中形成和拆分组合密钥。`ValidationParameter()` API 稍微先进一些，这些 API 可以获取并设置针对世界状态（**2**）和私有数据（**4**）的基于状态的认可策略。最后，`getHistoryForKey()` 通过返回存储值的集合来检索状态的历史记录，该存储值包括执行状态更新的交易标识符，从而允许从区块链读取交易（**10**）。

- createCompositeKey（）
- splitCompositeKey（）
- setStateValidationParameter（）
- getStateValidationParameter（）
- getPrivateDataValidationParameter（）
- setPrivateDataValidationParameter（）
- getHistoryForKey（）

- **事件 API** 用于管理智能合约中的事件处理。

- setEvent（）

智能合约使用此 API 将用户事件添加到交易响应中。请参阅交互点（**5**）。这些事件最终记录在区块链上，并在交互点（**11**）发送到侦听应用程序 。

- **实用程序 API** 是有用的 API 的集合，这些 API 并不容易包含在预定义的类别中，因此我们将它们分组在一起！它们包括检索当前通道名称，并将控制权传递给同一对等方上的不同链码。

- getChannelID（）

参见相互作用点（**13**）。在任何对等方上运行的智能合约都可以使用此 API 来确定应用程序在哪个通道上调用了智能合约。

- invokeChaincode（）

参见相互作用点（**14**）。MagnetoCorp 拥有的 Peer3 上安装了多个智能合约。这些智能合约可以使用此 API 相互调用。智能合约必须并置；无法在其他对等方上调用智能合约。

-

其中一些实用程序 API 仅在使用低级链码而不是智能合约的情况下使用。这些 API 主要用于链码输入的详细操作。智能合约 `Contract` 类会自动为开发人员整理所有此参数。

  - getFunctionAndParameters（）
  - getStringArgs（）
  - getArgs（）

5. 客户身份

在大多数情况下，提交事务的应用程序将使用 X.509 证书。在该示例中，由（**7** ）颁发的 X.509 证书（**6**）被（**8**）在其应用程序中用来在事务（**5**）中签署提案。`CA1` `Isabella` `t6`

`ClientIdentity` 接收返回的信息 `getCreator()`，并将一组 X.509 实用程序 API 放在其顶部，以使其更易于用于此常见用例。

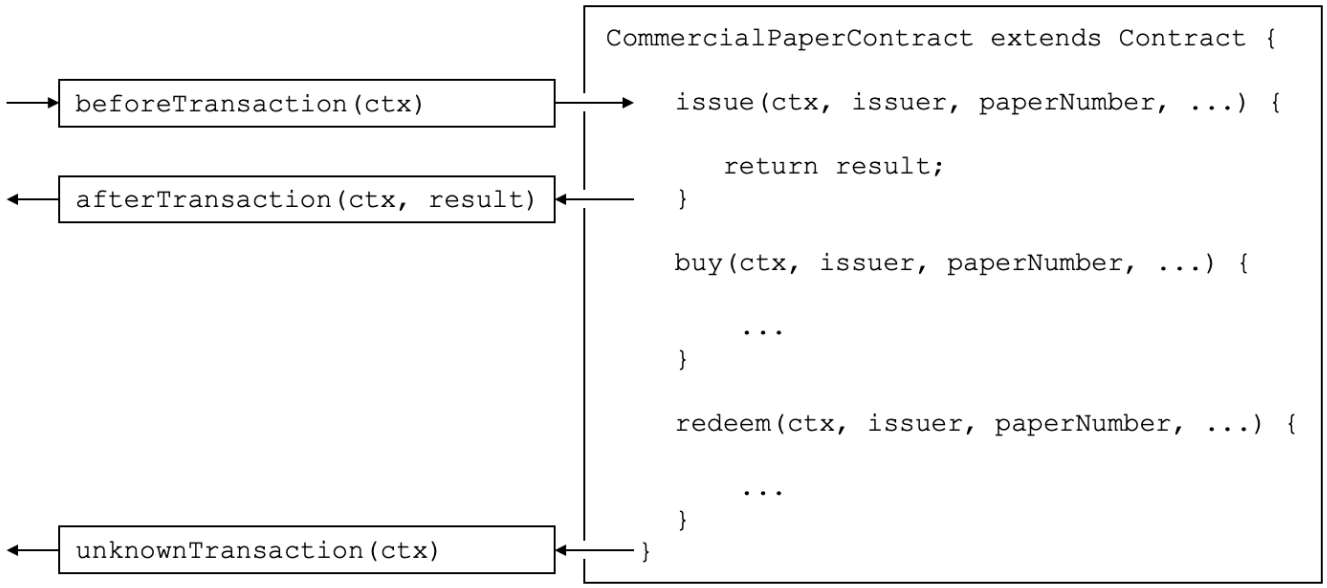
- getX509Certificate（） 返回事务提交者的完整 X.509 证书，包括其所有属性及其值。请参阅交互点（**6**）。
- getAttributeValue（） 返回特定 X.509 属性的值，例如，组织单位 `OU` 或专有名称 `DN`。请参阅交互点 （**6**）。
- assertAttributeValue（） 返回 `TRUE` 是否 X.509 属性的指定属性具有指定的值。请参阅交互点（**6**）。
- getID（） 根据事务提交者的专有名称和颁发方 CA 的专有名称返回事务提交者的唯一标识。格式为 。请参阅交互点（**6**）。`x509::{subject DN}::{issuer DN}`
- getMSPID（） 返回事务提交者的通道 MSP。这使得智能合约可以根据提交者的组织身份做出处理决策。请参阅交互点（**15**）或（**16**）。

十二.交易处理程序

受众：建筑师，应用程序和智能合约开发人员

事务处理程序允许智能合约开发人员在应用程序与智能合约之间的交互过程中的关键点定义通用处理。事务处理程序是可选的，但如果定义，它们将在调用智能合约中的每个事务之前或之后获得控制。还有一个特定的处理程序，在发出请求以调用未在智能合约中定义的交易时接收控制。

这是商业票据智能合约示例的交易处理程序示例：



之前，之后和未知事务处理程序。在此示例中， `beforeTransaction()` *在* ***issue，buy*** *和* ***赎回*** *事务之前被调用。* `afterTransaction()` *在* ***发行，购买*** *和* ***赎回*** *交易后称为。* `unknownTransaction()` *仅在请求调用未在智能合约中定义的交易的情况下调用。（该图通过不重复* `beforeTransaction` *并* `afterTransaction` *在每个事务中都用方框来简化。）*

6. 处理程序类型

三种类型的事务处理程序涵盖了应用程序和智能合约之间交互的不同方面：

- **在**每个智能合约交易被调用之前，调用 **handler**：之前。处理程序通常将修改事务上下文以供事务使用。处理程序可以访问所有的 Fabric API。例如，它可以发出 `getState()` 和 `putState()`。

- **在**每次智能合约交易被调用之后，调用 **handler**：之后。处理程序通常将执行所有事务通用的后处理，并且还具有对 Fabric API 的完全访问权限。

- 如果尝试调用智能合约中未定义的事务，则调用 **Unknown handler**：。通常，处理程序将记录故障，以供管理员进行后续处理。处理程序具有对 Fabric API 的完全访问权限。

定义事务处理程序是可选的；在没有定义处理程序的情况下，智能合约将正确执行。一个智能合约最多可以定义每种类型的一个处理程序。

7. 定义处理程序

事务处理程序将作为具有明确定义名称的方法添加到智能合约中。这是一个添加每种类型的处理程序的示例：

```
CommercialPaperContract extends Contract {

...

async beforeTransaction(ctx) {
    // Write the transaction ID as an informational to the console
```

```
    console.info(ctx.stub.getTxID());
};

async afterTransaction(ctx, result) {
    // This handler interacts with the ledger
    ctx.stub.cpList.putState(...);
};

async unknownTransaction(ctx) {
    // This handler throws an exception
    throw new Error('Unknown transaction function');
};

}
```

事务处理程序定义的形式对于所有处理程序类型都是相似的，但是请注意，事务处理程序定义也将如何接收事务返回的任何结果。该 API 文档 您展示了这些处理器的确切形式。`afterTransaction(ctx, result)`

8. 处理程序处理

将处理程序添加到智能合约后，将在交易处理期间将其调用。在处理期间，处理程序接收`ctx`， 事务上下文，执行一些处理，并在完成时返回控制。处理继续如下：

- **在处理程序之前**：如果处理程序成功完成，则使用更新的上下文调用事务。如果处理程序引发异常，则不会调用该事务，并且智能合约将失败，并显示异常错误消息。

- **处理程序之后**：如果处理程序成功完成，则智能合约将根据所调用的事务确定完成。如果处理程序引发异常，则事务将失败并显示异常错误消息。

- **未知处理程序**：处理程序应通过引发带有所需错误消息的异常来完成。如果未指定 **Unknown 处理程序**，或者未引发异常，则存在明智的默认处理；智能合约将失败，并显示**未知的交易**错误消息。

如果处理程序需要访问函数和参数，则很容易做到这一点：

```
async beforeTransaction(ctx) {
    // Retrieve details of the transaction
    let txnDetails = ctx.stub.getFunctionAndParameters();

    console.info(`Calling function: ${txnDetails.fcn} `);
    console.info(util.format(`Function arguments : %j ${stub.getArgs()} ``);
}
```

查看此处理程序如何 `getFunctionAndParameters` 通过 事务上下文使用实用程序 API 。

9. 多个处理程序

对于智能合约，最多只能为每种类型定义一个处理程序。如果智能合约需要在处理之前，之后或未知期间调用多个功能，则应在适当的功能内进行协调。

十三.背书政策

受众：建筑师，应用程序和智能合约开发人员

认可策略定义了认可交易以使其生效所需的最小组织集合。要进行认可，组织的认可同行需要运行与交易关联的智能合约并签署其结果。当订购服务将交易发送给提交对等方时，他们将分别检查交易中的背书是否满足背书策略。如果不是这种情况，则该交易将无效，并且对世界状态没有任何影响。

认可策略以两种不同的粒度起作用：可以为整个命名空间以及各个状态键设置它们。它们使用诸如 `AND` 和的基本逻辑表达式来表示 `OR`。例如，在 PaperNet 中，该方法可以按以下方式使用：从 MagnetoCorp 出售给 DigiBank 的论文的背书策略可以设置为，要求对本文进行任何更改都必须由 MagnetoCorp 和 DigiBank 背书。`AND(MagnetoCorp.peer, DigiBank.peer)`

十四.连接配置文件

受众：建筑师，应用程序和智能合约开发人员

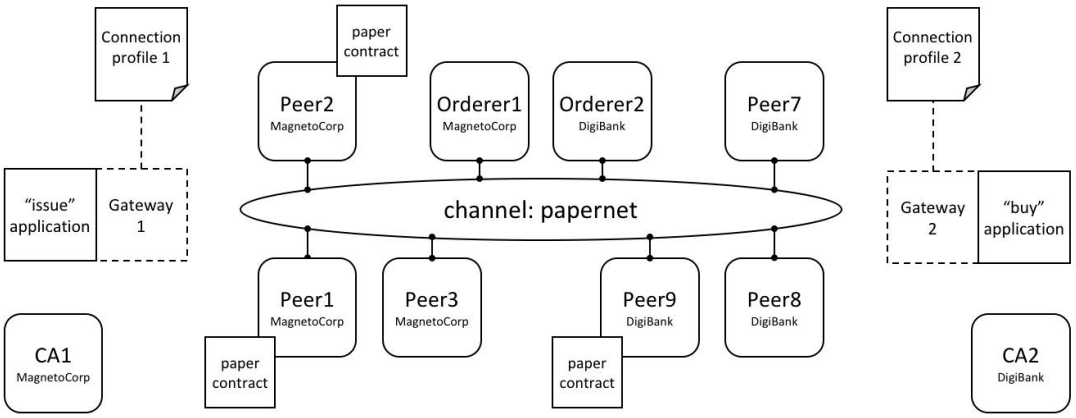
连接配置文件描述了一组组件，包括 Hyperledger Fabric 区块链网络中的对等方，订购者和证书颁发机构。它还包含与这些组件有关的渠道和组织信息。应用程序主要使用连接配置文件来配置处理所有网络交互的 网关，从而使其能够专注于业务逻辑。连接配置文件通常由了解网络拓扑的管理员创建。

在本主题中，我们将介绍：

- 为什么连接配置文件很重要
- 应用程序如何使用连接配置文件
- 如何定义连接配置文件

1. 情境

连接配置文件用于配置网关。网关之所以重要，有 很多原因，主要是简化应用程序与网络通道的交互。



发行和购买这两个应用程序使用配置有连接配置文件 1 &2 的网关 1 &2。每个配置文件都描述了 MagnetoCorp 和 DigiBank 网络组件的不同子集。每个连接配置文件都必须包含足够的信息，以使网关能够代表发行与网络交互并购买应用程序。请参阅文本以获取详细说明。

连接配置文件包含对网络视图的描述，以技术语法表示，可以是 JSON 或 YAML。在本主题中，我们使用 YAML 表示形式，因为它使您更容易阅读。静态网关比动态网关需要更多的信息，因为动态网关可以使用服务发现来动态扩展连接配置文件中的信息。

连接配置文件不应该是网络通道的详尽描述。它只需要包含足够的信息即可使用它的网关。在上面的网络中，连接配置文件 1 需要至少包含用于 `issue` 交易的认可组织和对等方，以及标识将在事务已提交到分类帐时通知网关的对等方。

将连接配置文件描述为描述网络*视图*最简单。它可能是一个全面的视图，但是由于以下几个原因，这是不现实的：

- 根据需要添加和删除对等方，订购者，证书颁发机构，渠道和组织。
- 组件可能会启动和停止，或者发生意外故障（例如断电）。
- 网关不需要整个网络的视图，例如仅需要成功处理事务提交或事件通知所需的视图。
- 服务发现可以扩充连接配置文件中的信息。具体来说，可以使用最少的 Fabric 拓扑信息来配置动态网关。其余的可以发现。

静态连接配置文件通常由详细了解网络拓扑的管理员创建。这是因为静态配置文件可以包含很多信息，并且管理员需要在相应的连接配置文件中捕获此信息。相反，动态配置文件将所需的定义量减到最少，因此对于想要快速上手的开发人员或想要创建响应速度更快的网关的管理员来说，是一个更好的选择。使用所选的编辑器以 YAML 或 JSON 格式创建连接配置文件。

2. 用法

稍后我们将介绍如何定义连接配置文件。让我们首先看看示例 MagnetoCorp `issue` 应用程序如何使用它：

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

const connectionProfile = yaml.safeLoad(fs.readFileSync('../gateway/paperNet.yaml', 'utf8'));

const gateway = new Gateway();

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

加载了一些必需的类之后，请查看如何 `paperNet.yaml` 从文件系统加载网关文件，如何使用 `yaml.safeLoad()` 方法将其转换为 JSON 对象 ，以及如何使用其 `connect()` 方法来配置网关。

通过使用此连接配置文件配置网关，发行应用程序将为网关提供应用于处理事务的相关网络拓扑。这是因为连接配置文件包含有关 PaperNet 渠道，组织，对等方，订购者和 CA 的足够信息，以确保可以成功处理交易。

对于任何给定的组织，连接概要文件都定义一个以上的对等体是一个好习惯-它可以防止单点故障。这种做法也适用于动态网关。为服务发现提供多个起点。

DigiBank `buy` 应用程序通常会将其网关配置为具有类似的连接配置文件，但有一些重要的区别。有些元素是相同的，例如通道；有些元素会重叠，例如认可同级。其他元素将完全不同，例如通知对等方或证书颁发机构。

该 `connectionOptions` 传递给网关配合连接配置文件。它们允许应用程序声明网关如何使用连接配置文件。SDK 会对它们进行解释，以控制与网络组件的交互模式，例如选择要与之连接的身份或用于事件通知的对等实体。阅读 有关可用连接选项的列表以及何时使用它们的信息。

3. 结构体

为了帮助您了解连接配置文件的结构，我们通过对显示的网络的例子要步以上。它的连接配置文件基于 PaperNet 商业论文样本，并 存储 在 GitHub 存储库中。为了方便起见，我们在下面复制了它。现在，您会发现将其显示在另一个浏览器窗口中很有帮助：

- 第 9 行： `name: "papernet.magnetocorp.profile.sample"`

这是连接配置文件的名称。尝试使用 DNS 样式名称；它们是传达意义的非常简单的方法。

- 第 16 行： `x-type: "hlfv1"`

用户可以添加自己的 `x-` “特定于应用程序” 的属性，就像使用 HTTP 标头一样。它们主要提供给将来使用。

- 第 20 行： `description: "Sample connection profile for documentation topic"`

连接配置文件的简短描述。尝试使此功能对可能是第一次看到此内容的读者有所帮助！

- 第 25 行： `version: "1.0"`

此连接配置文件的架构版本。当前仅支持 1.0 版，并且没有预见到该架构会经常更改。

- 第 32 行： `channels:`

这是第一条真正重要的线。 `channels:` 标识此连接配置文件描述的*所有*通道如下。但是，优良作法是将不同的通道保留在不同的连接配置文件中，尤其是当它们彼此独立使用时。

- 第 36 行： `papernet:`

有关 `papernet` 此连接配置文件中第一个频道的详细信息，将在后面介绍。

- 第 41 行： `orderers:`

所有订购者的详细信息，请 `papernet` 关注。您可以在第 45 行看到此频道的订购者为 `orderer1.magnetocorp.example.com`。这只是一个逻辑名称；稍后在连接配置文件（第 134-147 行）中，将详细介绍如何连接到此订购器。注意， `orderer2.digibank.example.com` 它不在此列表中。应用程序使用自己组织的订购者，而不是其他组织的订购者，这是有道理的。

- 第 49 行： `peers:`

所有同行的详细信息 `papernet` 如下。

你可以看到从 MagnetoCorp 列出了三个同行： `peer1.magnetocorp.example.com`， `peer2.magnetocorp.example.com` 和 `peer3.magnetocorp.example.com`。不必像这里所做的那样列出 MagnetoCorp 中的所有对等方。你可以看到从 DigiBank 上市仅一个对等体 `peer9.digibank.example.com`; 现在我们将确认，包括该对等点开始意味着背书策略要求 MagnetoCorp 和 DigiBank 背书交易。最好有多个同级，以避免单点故障。

下面每个对等，你可以看到四个非排他性的角色： **endorsingPeer**， **chaincodeQuery**，**ledgerQuery** 和 **EventSource 的**。了解他们如何 `peer1` 以及 如何 `peer2` 担任所有角色 `papercontract`。与相比 `peer3`，只能用于通知或访问分类账的区块链组件而不是世界状态的分类账查询，因此不需要安装智能合约。请注意 `peer9`，除了背书之外，不应将其用于其他任何用途，因为 MagnetoCorp 的同行可以更好地担当这些角色。

再次，看看如何根据其逻辑名称和角色来描述对等体。在配置文件的后面，我们将看到这些对等方的物理信息。

- 第 97 行： `organizations:`

所有组织的详细信息将在所有渠道中显示。请注意，尽管 `papernet` 目前是唯一列出的组织，但这些组织适用于所有渠道。这是因为组织可以位于多个渠道中，并且渠道可以具有多个组织。而且，某些应用程序操作与组织有关，而不是与渠道有关。例如，应用程序可以使用连接选项从其组织内的一个或所有对等方或网络内的所有组织请求通知。为此，需要一个组织到对等方的映射，本节将提供它。

- 第 101 行： `MagnetoCorp:`

被认为 MagnetoCorp 的一部分，所有的同行都列出： `peer1`， `peer2` 和 `peer3`。证书颁发机构也是如此。再次，注意逻辑名称的用法，与本 `channels:` 节相同；物理信息将在配置文件的后面。

- 第 121 行： `DigiBank:`

仅 `peer9` 列出为 DigiBank 的一部分，没有证书颁发机构。这是因为这些其他对等方和 DigiBank CA 与该连接配置文件的用户无关。

- 第 134 行： `orderers:`

现在列出了订购者的物理信息。由于此连接配置文件仅提及的一个订购者 `papernet`，因此您会看到 `orderer1.magnetocorp.example.com` 列出的详细信息。其中包括其 IP 地址和端口，以及 gRPC 选项，可以在需要时覆盖与订购者进行通信时使用的默认值。与一样 `peers:`，对于高可用性，指定多个订购者是一个好主意。

- 第 152 行： `peers:`

现在列出了所有先前对等方的物理信息。此连接配置文件有三个同行的 MagnetoCorp： ， `peer1`， `peer2` 和 `peer3`; 对于 DigiBank，单个同级 `peer9` 列出了其信息。对于每个对等方，如订购者一样，将列出其 IP 地址和端口以及 gRPC 选项，这些选项可以覆盖与特定对等方进行通信时使用的默认值（如有必要）。

- 194 行： `certificateAuthorities:`

现在列出了证书颁发机构的物理信息。连接配置文件为 MagnetoCorp 列出了一个 CA `ca1-magnetocorp`，其物理信息如下。除了 IP 详细信息之外，注册商信息还允许将此 CA 用于证书签名请求（CSR）。这些用于请求本地生成的公用/专用密钥对的新证书。

现在，您已经了解了 MagnetoCorp 的连接配置文件，您可能希望查看 DigiBank 的 相应配置文件。找到该配置文件与 MagnetoCorp 的配置文件相同的地方，查看其相似之处，最后是不同的地方。考虑一下为什么这些差异对于 DigiBank 应用程序有意义。

这就是您需要了解的有关连接配置文件的所有信息。总之，连接配置文件为应用程序定义了足够的通道，组织，对等方，订购者和证书颁发机构，以配置网关。网关允许应用程序专注于业务逻辑，而不是网络拓扑的细节。

4. 样品

该文件是从 GitHub 商业文件样本中直接复制的 。

```
1: ---
2: #
3: # [Required]. A connection profile contains information about a set of network
4: # components. It is typically used to configure gateway, allowing applications
5: # interact with a network channel without worrying about the underlying
6: # topology. A connection profile is normally created by an administrator who
7: # understands this topology.
8: #
9: name: "papernet.magnetocorp.profile.sample"
10: #
11: # [Optional]. Analogous to HTTP, properties with an "x-" prefix are deemed
12: # "application-specific", and ignored by the gateway. For example, property
13: # "x-type" with value "hlfv1" was originally used to identify a connection
14: # profile for Fabric 1.x rather than 0.x.
15: #
16: x-type: "hlfv1"
17: #
18: # [Required]. A short description of the connection profile
19: #
20: description: "Sample connection profile for documentation topic"
21: #
22: # [Required]. Connection profile schema version. Used by the gateway to
23: # interpret these data.
24: #
25: version: "1.0"
26: #
27: # [Optional]. A logical description of each network channel; its peer and
28: # orderer names and their roles within the channel. The physical details of
29: # these components (e.g. peer IP addresses) will be specified later in the
30: # profile; we focus first on the logical, and then the physical.
31: #
32: channels:
33:   #
34:   # [Optional]. papernet is the only channel in this connection profile
35:   #
36:   papernet:
37:     #
38:     # [Optional]. Channel orderers for PaperNet. Details of how to connect to
39:     # them is specified later, under the physical "orderers:" section
40:     #
41:     orderers:
42:     #
43:     # [Required]. Orderer logical name
44:     #
45:       - orderer1.magnetocorp.example.com
46:     #
47:     # [Optional]. Peers and their roles
```

```
    #
    peers:
    #
    # [Required]. Peer logical name
    #
      peer1.magnetocorp.example.com:
        #
        # [Optional]. Is this an endorsing peer? (It must have chaincode
        # installed.) Default: true
        #
        endorsingPeer: true
        #
        # [Optional]. Is this peer used for query? (It must have chaincode
        # installed.) Default: true
        #
        chaincodeQuery: true
        #
        # [Optional]. Is this peer used for non-chaincode queries? All peers
        # support these types of queries, which include queryBlock(),
        # queryTransaction(), etc. Default: true
        #
        ledgerQuery: true
        #
        # [Optional]. Is this peer used as an event hub? All peers can produce
        # events. Default: true
        #
        eventSource: true
      #
      peer2.magnetocorp.example.com:
        endorsingPeer: true
        chaincodeQuery: true
        ledgerQuery: true
        eventSource: true
      #
      peer3.magnetocorp.example.com:
        endorsingPeer: false
        chaincodeQuery: false
        ledgerQuery: true
        eventSource: true
      #
      peer9.digibank.example.com:
        endorsingPeer: true
        chaincodeQuery: false
        ledgerQuery: false
        eventSource: false
#
# [Required]. List of organizations for all channels. At least one organization
# is required.
#
organizations:
   #
   # [Required]. Organizational information for MagnetoCorp
   #
   MagnetoCorp:
     #
     # [Required]. The MSPID used to identify MagnetoCorp
     #
     mspid: MagnetoCorpMSP
     #
     # [Required]. The MagnetoCorp peers
     #
     peers:
       - peer1.magnetocorp.example.com
       - peer2.magnetocorp.example.com
       - peer3.magnetocorp.example.com
     #
     # [Optional]. Fabric-CA Certificate Authorities.
     #
     certificateAuthorities:
       - ca-magnetocorp
   #
   # [Optional]. Organizational information for DigiBank
   #
```

```
  DigiBank:
    #
    # [Required]. The MSPID used to identify DigiBank
    #
    mspid: DigiBankMSP
    #
    # [Required]. The DigiBank peers
    #
    peers:
      - peer9.digibank.example.com
#
# [Optional]. Orderer physical information, by orderer name
#
orderers:
  #
  # [Required]. Name of MagnetoCorp orderer
  #
  orderer1.magnetocorp.example.com:
    #
    # [Required]. This orderer's IP address
    #
    url: grpc://localhost:7050
    #
    # [Optional]. gRPC connection properties used for communication
    #
    grpcOptions:
      ssl-target-name-override: orderer1.magnetocorp.example.com
#
# [Required]. Peer physical information, by peer name. At least one peer is
# required.
#
peers:
  #
  # [Required]. First MagetoCorp peer physical properties
  #
  peer1.magnetocorp.example.com:
    #
    # [Required]. Peer's IP address
    #
    url: grpc://localhost:7151
    #
    # [Optional]. gRPC connection properties used for communication
    #
    grpcOptions:
      ssl-target-name-override: peer1.magnetocorp.example.com
      request-timeout: 120001
  #
  # [Optional]. Other MagnetoCorp peers
  #
  peer2.magnetocorp.example.com:
    url: grpc://localhost:7251
    grpcOptions:
      ssl-target-name-override: peer2.magnetocorp.example.com
      request-timeout: 120001
  #
  peer3.magnetocorp.example.com:
    url: grpc://localhost:7351
    grpcOptions:
      ssl-target-name-override: peer3.magnetocorp.example.com
      request-timeout: 120001
  #
  # [Required]. Digibank peer physical properties
  #
  peer9.digibank.example.com:
    url: grpc://localhost:7951
    grpcOptions:
      ssl-target-name-override: peer9.digibank.example.com
      request-timeout: 120001
#
# [Optional]. Fabric-CA Certificate Authority physical information, by name.
# This information can be used to (e.g.) enroll new users. Communication is via
# REST, hence options relate to HTTP rather than gRPC.
#
```

```
194: certificateAuthorities:
195:   #
196:   # [Required]. MagnetoCorp CA
197:   #
198:   ca1-magnetocorp:
199:     #
200:     # [Required]. CA IP address
201:     #
202:     url: http://localhost:7054
203:     #
204:     # [Optioanl]. HTTP connection properties used for communication
205:     #
206:     httpOptions:
207:       verify: false
208:     #
209:     # [Optional]. Fabric-CA supports Certificate Signing Requests (CSRs). A
210:     # registrar is needed to enroll new users.
211:     #
212:     registrar:
213:       - enrollId: admin
214:         enrollSecret: adminpw
215:     #
216:     # [Optional]. The name of the CA.
217:     #
218:     caName: ca-magnetocorp
```

十五.连接选项

受众：架构师，管理员，应用程序和智能合约开发人员

连接选项与连接配置文件一起使用，以*精确*控制 网关与网络的交互方式。使用网关可以使应用程序专注于业务逻辑，而不是网络拓扑。

在本主题中，我们将介绍：

- 为什么连接选项很重要
- 应用程序如何使用连接选项
- 每个连接选项的作用
- 何时使用特定的连接选项

1. 情境

连接选项指定网关行为的特定方面。网关之所以重要，有很多原因，主要是允许应用程序专注于业务逻辑和智能合约，同时管理与网络许多组件的交互。

连接选项控制行为的不同交互点。这些选项在本文中进行了详细说明。

连接选项的一个示例可能是指定 `issue` 应用程序使用的网关应使用身份 `Isabella` 向 `papernet` 网络提交事务。另一个可能是网关应等待 MagnetoCorp 的所有三个节点来确认已提交事务以返回控制。连接选项允许应用程序指定网关与网络交互的精确行为。没有网关，应用程序需要做更多的工作。网关可以节省您的时间，使您的应用程序更具可读性，并且不易出错。

2. 用法

稍后我们将描述可用于应用程序的全套连接选项。让我们首先看看示例 MagnetoCorp `issue` 应用程序如何指定它们：

```
    const userName = 'User1@org1.example.com';
const wallet = new FileSystemWallet('../identity/user/isabella/wallet');

const connectionOptions = {
  identity: userName,
  wallet: wallet,
  eventHandlerOptions: {
    commitTimeout: 100,
    strategy: EventStrategies.MSPID_SCOPE_ANYFORTX
    }
  };

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

查看 `identity` 和 `wallet` 选项如何成为 `connectionOptions` 对象的简单属性 。它们 分别具有值 `userName` 和 `wallet`，这些值在代码的前面设置。将这些选项与 `eventHandlerOptions` 本身就是对象的选项进行对比。它具有两个属性：（以秒为单位）和。
`commitTimeout: 100strategy: EventStrategies.MSPID_SCOPE_ANYFORTX`

了解如何 `connectionOptions` 传递给网关以作为对它的补充 `connectionProfile`；网络由连接配置文件标识，选项精确指定网关应如何与之交互。现在让我们看看可用的选项。

3. 选件

这是可用选项及其作用的列表。

- `wallet` 标识网关将代表应用程序使用的钱包。见互动 **1**；钱包是由应用程序指定的，但实际上是从中检索身份的网关。

  必须指定一个钱包；最重要的决定是 要使用的钱包类型，是文件系统，内存，HSM 还是数据库。
- `identity` 是应用程序将使用的用户身份 `wallet`。见互动 **2a**；用户身份由应用程序指定，代表应用程序的用户 Isabella，**2b**。身份实际上是由网关检索的。

  在我们的示例中，不同 MSP（**2c**， **2d**）将使用 Isabella 的身份来标识她来自 MagnetoCorp，并在其中具有特定角色。这两个事实将相应地确定她对资源的许可，例如能够读取和写入分类帐。

  必须指定用户身份。如您所见，此身份对于 Hyperledger Fabric 是一个*允许的*网络这一观念至关重要-所有参与者都具有一个身份，包括应用程序，对等方和订购者，这些身份决定了他们对资源的控制。您可以在会员服务主题中阅读有关此想法的更多信息。
- `clientTlsIdentity` 是从钱包（**3a**）检索并用于网关和不同通道组件（例如对等设备和订购者）之间的安全通信（**3b**）的身份。

  请注意，此身份不同于用户身份。尽管 `clientTlsIdentity` 对于安全通信很重要，但它不如用户身份那么基础，因为它的范围不会超出安全网络通信的范围。

`clientTlsIdentity` 是可选的。建议您在生产环境中进行设置。您应该始终使用 `clientTlsIdentity`， `identity` 因为这些身份具有不同的含义和生命周期。例如，如果您 `clientTlsIdentity` 受到了损害，那么您的也会受到损害 `identity`。将它们分开可以更安全。

- `eventHandlerOptions.commitTimeout` 是可选的。它以秒为单位指定在将控制权返回给应用程序之前，网关应等待任何对等方（**4a**）提交事务的最长时间。用于通知的对等体集由该 `eventHandlerOptions.strategy` 选项确定 。如果未指定 commitTimeout，则网关将使用 300 秒的超时。

- `eventHandlerOptions.strategy` 是可选的。它标识网关应用于侦听事务已提交的通知的对等体的集合。例如，是侦听组织中的单个对等方还是所有对等方。它可以采用以下值之一：
  - `EventStrategies.MSPID_SCOPE_ANYFORTX` 侦听用户组织内的**任何**同级。在我们的示例中，请参见交互点 **4b**；MagnetoCorp 的对等体 1，对等体 2 或对等体 3 中的任何一个都可以通知网关。
  - `EventStrategies.MSPID_SCOPE_ALLFORTX` **这是默认值**。监听用户组织内的**所有**对等方。在我们的对等示例中，请参见交互点 **4b**。MagnetoCorp 的所有对等方都必须已通知网关。对等体 1，对等体 2 和对等体 3。仅当对等体是已知/发现且可用时，才对等体进行计数；不包括已停止或已失败的对等体。
  - `EventStrategies.NETWORK_SCOPE_ANYFORTX` 侦听整个网络通道中的**任何**对等方。在我们的示例中，请参见交互点 **4b** 和 **4c**；MagnetoCorp 的对等 1-3 或 DigiBank 的对等 7-9 中的任何一个都可以通知网关。
  - `EventStrategies.NETWORK_SCOPE_ALLFORTX` 侦听整个网络通道中的**所有**对等方。在我们的示例中，请参见交互点 **4b** 和 **4c**。MagnetoCorp 和 DigiBank 的所有同级都必须通知网关；对等 1-3 和对等 7-9。仅当已知/发现并可用时，才对等；不包括已停止或已失败的对等体。
  - < `PluginEventHandlerFunction` >用户定义的事件处理程序的名称。这允许用户定义自己的事件处理逻辑。了解如何 定义 插件事件处理程序，并检查样本处理程序。

  仅当您有非常特定的事件处理要求时，才需要用户定义的事件处理程序。通常，内置事件策略之一就足够了。用户定义的事件处理程序的一个示例可能是等待组织中超过一半的对等方来确认事务已提交。

  如果确实指定了用户定义的事件处理程序，则它不会影响您的应用程序逻辑。它与此完全不同。SDK 在处理过程中会调用处理程序；它决定何时调用它，并使用其结果选择要用于事件通知的同级。SDK 完成处理后，应用程序将收到控制权。

  如果未指定用户定义的事件处理程序， `EventStrategies` 则使用的默认值 。
- `discovery.enabled` 是可选的，并且可能具有 `true` 或值 `false`。默认值为 `true`。它确定网关是否使用服务发现来扩展连接配置文件中指定的网络拓扑。见相互作用点 **6**；网关使用的对等八卦信息。

  该值将被 `INITIALIIZE-WITH-DISCOVERY` 环境变量覆盖，可以将其设置为 `true` 或 `false`。
- `discovery.asLocalhost` 是可选的，并且可能具有 `true` 或值 `false`。默认值为 `true`。它确定是否将在服务发现期间找到的 IP 地址从 docker 网络转换为本地主机。

通常，开发人员会编写使用 docker 容器作为其网络组件（例如对等方，订购者和 CA）的应用程序，但这些应用程序本身不会在 docker 容器中运行。这就是 `true` 默认设置的原因；在生产环境中，应用程序可能会以与网络组件相同的方式在 docker 容器中运行，因此不需要地址转换。在这种情况下，应用程序应显式指定 `false` 或使用环境变量替代。

该值将被 `DISCOVERY-AS-LOCALHOST` 环境变量覆盖，可以将其设置为 `true` 或 `false`。

4. 注意事项

下面的注意事项列表在决定如何选择连接选项时会很有帮助。

- `eventHandlerOptions.commitTimeout` 和 `eventHandlerOptions.strategy` 一起工作。例如，并且意味着网关将等待长达 100 秒*的任何*同行，以确认交易已提交。相反，指定意味着网关将对*所有*组织中的*所有*对等方最多等待 100 秒。

`commitTimeout: 100strategy: EventStrategies.MSPID_SCOPE_ANYFORTXstrategy: EventStrategies.NETWORK_SCOPE_ALLFORTX`

- 默认值将等待应用程序组织中的所有对等方提交事务。这是一个很好的默认设置，因为应用程序可以确保所有对等方都拥有分类账的最新副本，从而最大程度地减少了并发问题 `eventHandlerOptions.strategy: EventStrategies.MSPID_SCOPE_ALLFORTX`

但是，随着组织中对等方数目的增加，等待所有对等方变得有点不必要，在这种情况下，使用可插入事件处理程序可以提供更有效的策略。例如，在共识将使所有分类帐保持同步的安全假设下，可以使用相同的一组对等点来提交交易和侦听通知。

- 需要 `clientTlsIdentity` 设置服务发现。这是因为与应用程序交换信息的对等方需要确信他们正在与自己信任的实体交换信息。如果 `clientTlsIdentity` 未设置，则 `discovery` 无论是否设置，都将不服从。

- 尽管应用程序可以在连接到网关时设置连接选项，但是管理员可能需要覆盖这些选项。这是因为选项与网络交互有关，该交互可能随时间而变化。例如，管理员试图了解使用服务发现对网络性能的影响。

一种好的方法是在配置文件中定义应用程序替代，当应用程序配置其与网关的连接时，该配置文件将由应用程序读取。

由于发现选项 `enabled` 和 `asLocalHost` 最经常需要由管理员覆盖，环境变量 `INITIALIIZE-WITH-DISCOVERY`，并 `DISCOVERY-AS-LOCALHOST` 提供了方便。管理员应在应用程序的生产运行时环境中设置这些设置，该环境很可能是 docker 容器。

十六.钱包

受众：建筑师，应用程序和智能合约开发人员

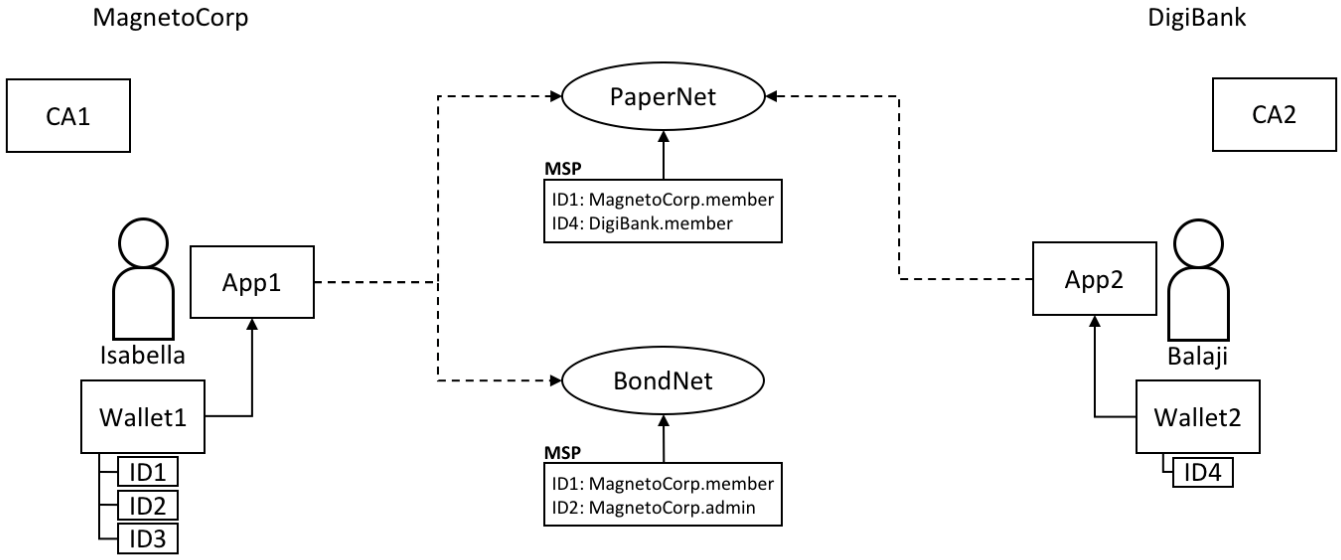
钱包包含一组用户身份。用户运行的应用程序在连接到通道时会选择这些身份之一。结合 MSP 使用此身份确定对诸如分类帐之类的信道资源的访问权限。

在本主题中，我们将介绍：

- 为什么钱包很重要
- 钱包的组织方式
- 不同类型的钱包
- 钱包操作

1. 情境

例如，当应用程序连接到诸如 PaperNet 的网络通道时，它会选择一个用户身份进行连接 `ID1`。通道 MSP `ID1` 与特定组织内的角色相关联，该角色最终将确定应用程序对通道资源的权利。例如，`ID1` 可能将某个用户标识为可以读写账本的 MagnetoCorp 组织成员，而 `ID2` 可能会在 MagnetoCorp 中标识可以向联盟添加新组织的管理员。



伊莎贝拉（Isabella）和巴拉吉（Balaji）这两个用户拥有包含不同身份的钱包，可用于连接到不同的网络渠道 PaperNet 和 BondNet。

考虑两个用户的例子；MagnetoCorp 的 Isabella 和 DigiBank 的 Balaji。Isabella 将使用 App 1 在 PaperNet 中调用一个智能合约，在 BondNet 中调用另一个智能合约。同样，Balaji 将使用 App 2 调用智能合约，但仅限于 PaperNet。（应用程序很 容易访问其中的多个网络和多个智能合约。）

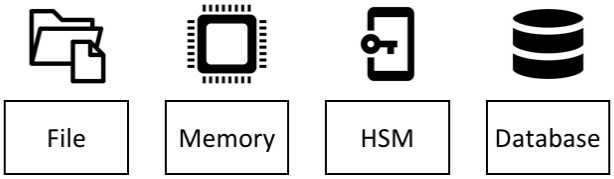
怎么看：

- MagnetoCorp 使用 CA1 颁发身份，而 DigiBank 使用 CA2 颁发身份。这些身份存储在用户钱包中。
- Balaji 的钱包拥有 `ID4` 由 CA2 发行的单一身份。伊莎贝拉的钱包有很多身份，`ID1`，`ID2` 和 `ID3`，由 CA1 颁发。钱包可以为一个用户保留多个身份，并且每个身份可以由不同的 CA 颁发。
- Isabella 和 Balaji 都连接到 PaperNet，并且其 MSP 确定 Isabella 是 MagnetoCorp 组织的成员，而 Balaji 是 DigiBank 组织的成员，因为各自的 CA 均发布了它们的身份。（这是 可能的组织使用多个 CA，且单个 CA 支持多个组织。）

- Isabella 可以 `ID1` 用来连接 PaperNet 和 BondNet。在这两种情况下，当 Isabella 使用此身份时，她都被视为 MangetoCorp 的成员。
- Isabella 可以用于 `ID2` 连接到 BondNet，在这种情况下，她被确定为 MagnetoCorp 的管理员。这给了伊莎贝拉两个不同的特权：`ID1` 将她标识为可以读写 BondNet 总帐的 MagnetoCorp 的简单成员，而将 `ID2` 她标识为可以向 BondNet 添加新组织的 MagnetoCorp 管理员。
- Balaji 无法使用连接到 BondNet `ID4`。如果他尝试连接，`ID4` 将不会被视为 DigiBank 的成员，因为 BondNet 的 MSP 不知道 CA2。

2. 种类

根据其存储身份的钱包有不同类型：



钱包的四种不同类型：文件系统，内存，硬件安全模块（HSM）和 CouchDB。

- **FileSystem**：这是最常见的存放钱包的地方；文件系统无处不在，易于理解，并且可以通过网络安装。对于钱包来说，它们是一个很好的默认选择。

  使用 `FileSystemWallet` 该类 来管理文件系统钱包。
- **内存中**：应用程序存储中的钱包。当您的应用程序在受限环境中运行而无法访问文件系统时，请使用这种钱包。通常是网络浏览器。值得记住的是，这种钱包是易变的。应用程序正常结束或崩溃后，身份将丢失。

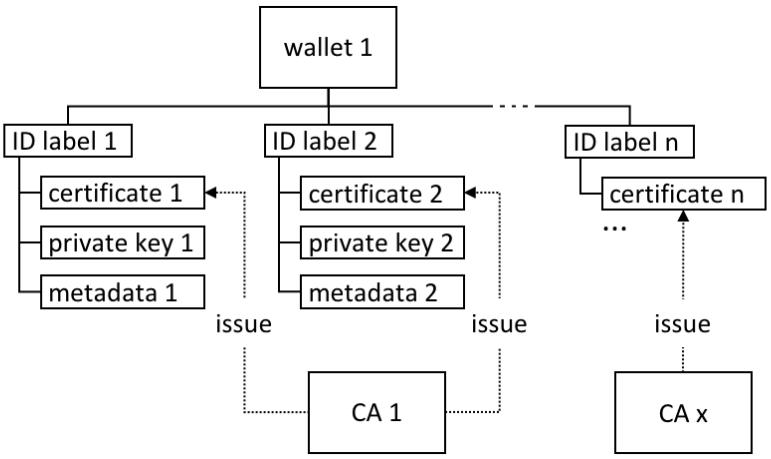
  使用 `InMemoryWallet` 该类 来管理内存中的钱包。
- **硬件安全模块**：存储在 HSM 中的钱包 。这种超安全，防篡改的设备可存储数字身份信息，尤其是私钥。HSM 可以本地连接到您的计算机或可通过网络访问。大多数 HSM 提供了使用私钥执行机载加密的功能，这样私钥就永远不会离开 HSM。

  当前，您应该 结合使用 `FileSystemWallet` 该类和 HSMWalletMixin 类来管理 HSM 钱包。
- **CouchDB**：存储在 Couch DB 中的钱包。这是钱包存储的最稀有形式，但是对于那些希望使用数据库备份和还原机制的用户，CouchDB 钱包可以提供简化灾难恢复的有用选项。

  使用 `CouchDBWallet` 该类 来管理 CouchDB 钱包。

3. 结构体

单个钱包可以保存多个身份，每个身份由特定的证书颁发机构颁发。每个身份具有一个标准结构，包括描述性标签，一个包含公共密钥，私有密钥和某些特定于 Fabric 的元数据的 X.509 证书。不同的钱包类型将此结构适当地映射到其存储机制。



Fabric 钱包可以使用由不同证书颁发机构颁发的证书来保存多个身份。身份包括证书，私钥和结构元数据。

有几种关键的类方法可以简化钱包和身份的管理：

```
const identity = X509WalletMixin.createIdentity('Org1MSP', certificate, key);

await wallet.import(identityLabel, identity);
```

了解该 `X509WalletMixin.createIdentity()` 方法如何 创建 `identity` 具有元数据 `Org1MSP`，`certificate` 和的私有 `key`。了解如何 `wallet.import()` 将此身份添加到具有特定的钱包中 `identityLabel`。

在上面的示例中，`Gateway` 该类仅需要 `mspid` 为身份设置元数据 `Org1MSP`。*当前*，例如，当请求特定的通知策略时，它使用此值从连接配置文件中标识特定的对等方。在 DigiBank 网关文件中，查看通知将如何 与以下内容关联：`networkConnection.yamlOrg1MSPpeer0.org1.example.com`

```
organizations:
  Org1:
    mspid: Org1MSP

    peers:
      - peer0.org1.example.com
```

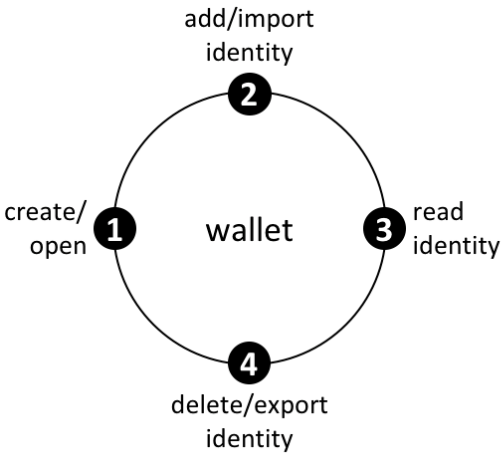
您确实不需要担心不同钱包类型的内部结构，但是如果您有兴趣，请导航到商业票据示例中的用户身份文件夹：

```
magnetocorp/identity/user/isabella/
                                wallet/
                                     User1@org1.example.com/
                                                          User1@org.example.com
                                                          c75bd6911aca8089...-priv
                                                          c75bd6911aca8089...-pub
```

您可以检查这些文件，但是正如所讨论的，使用 SDK 来操作这些数据更加容易。

4. 运作方式

不同的钱包类派生自一个通用的 电子钱包 基类，该基类提供一组标准的 API 来管理身份。这意味着可以使应用程序独立于底层钱包存储机制。例如，文件系统和 HSM 钱包的处理方式非常相似。



钱包遵循生命周期：可以创建或打开钱包，并且可以读取，添加，删除和导出身份。

应用程序可以根据简单的生命周期使用钱包。可以打开或创建钱包，然后可以添加，读取，更新，删除和导出身份。花一些时间在 JSDOC 中的不同 `Wallet` 方法上，看看它们如何工作。商业论文教程在以下方面提供了一个很好的示例： `addToWallet.js`

```
const wallet = new FileSystemWallet('../identity/user/isabella/wallet');

const cert = fs.readFileSync(path.join(credPath, '.../User1@org1.example.com-cert.pem')).toString();
const key = fs.readFileSync(path.join(credPath, '.../_sk')).toString();

const identityLabel = 'User1@org1.example.com';
const identity = X509WalletMixin.createIdentity('Org1MSP', cert, key);

await wallet.import(identityLabel, identity);
```

注意如何：

- 首次运行该程序时，会在本地文件系统上创建一个钱包 `.../isabella/wallet`。
- 从文件系统加载证书 `cert` 和私有证书 `key`。
- 一个新的身份与创建 `cert`， `key` 并 `Org1MSP` 使用 `X509WalletMixin.createIdentity()`。
- 新身份 `wallet.import()` 带有标签 导入钱包 `User1@org1.example.com`。

这就是您需要了解的所有钱包信息。您已经了解了它们如何保留应用程序代表用户使用的身份来访问 Fabric 网络资源。根据您的应用程序和安全需求，可以使用不同类型的钱包，并且有一组简单的 API 可以帮助应用程序管理钱包及其中的身份。

十七.网关

受众：建筑师，应用程序和智能合约开发人员

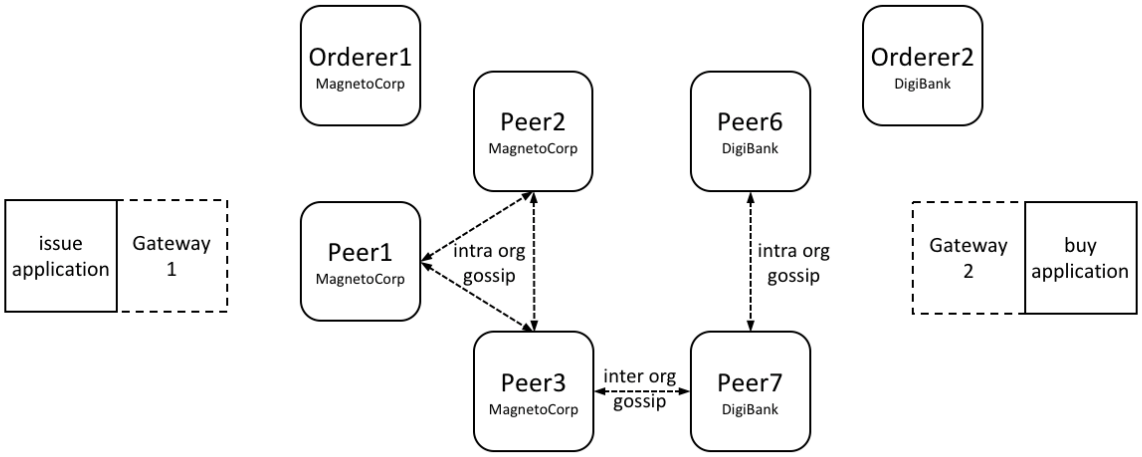
网关代表应用程序管理网络交互，使其专注于业务逻辑。应用程序连接到网关，然后使用该网关的配置管理所有后续交互。

在本主题中，我们将介绍：

- 为什么网关很重要
- 应用程序如何使用网关
- 如何定义静态网关
- 如何为服务发现定义动态网关
- 使用多个网关

1. 情境

Hyperledger Fabric 网络通道可以不断变化。由网络中不同组织贡献的对等，订购者和 CA 组件将会来来往往。造成这种情况的原因包括业务需求增加或减少，以及计划内和计划外的停机。网关减轻了应用程序的负担，使它可以专注于要解决的业务问题。



MagnetoCorp 和 DigiBank 应用程序（发行和购买）将各自的网络交互委托给其网关。每个网关都了解网络通道拓扑，其中包括两个组织 MagnetoCorp 和 DigiBank 的多个同级和订购者，使应用程序专注于业务逻辑。同行可以使用八卦协议在组织内部和组织之间相互交谈。

网关可以由应用程序以两种不同方式使用：

- **静态**：网关配置在连接配置文件中*完全*定义。应用程序可用的所有对等方，订购者和 CA 在用于配置网关的连接配置文件中静态定义。例如，对于对等方，这包括其作为支持对等方或事件通知中心的角色。您可以在连接配置文件主题中了解有关这些角色的更多信息。

  SDK 将结合网关连接选项使用此静态拓扑 来管理事务提交和通知过程。连接配置文件必须包含足够的网络拓扑，以允许网关代表应用程序与网络交互；这包括网络渠道，组织，订购者，对等方及其角色。

- **动态**：网关配置在连接配置文件中最少定义。通常，指定来自应用程序组织的一个或两个对等方，它们使用服务发现来发现可用的网络拓扑。这包括对等方，订购者，渠道，实例化的智能合约及其签注策略。（在生产环境中，网关配置应至少指定两个对等方以确保可用性。）

  SDK 将使用所有静态和发现的拓扑信息以及网关连接选项来管理事务提交和通知过程。作为其一部分，它还将智能地使用发现的拓扑。例如，它将 使用发现的智能合约背书策略*计算*所需的最少背书对等体。

您可能会问自己，静态或动态网关是否更好？需要在可预测性和响应性之间进行权衡。静态网络始终会以相同的方式运行，因为它们将网络视为不变的。从这种意义上讲，它们是可预测的–如果可用，它们将始终使用相同的对等方和订购方。当动态网络了解网络的变化时，它们的响应速度更快–他们可以使用新添加的对等点和订购者，从而带来额外的弹性和可扩展性，但可能会在可预测性方面付出一些代价。通常，使用动态网络很好，实际上这是网关的默认模式。

请注意，可以静态或动态使用*同*一连接配置文件。显然，如果要静态使用配置文件，则它必须是全面的，而动态使用只需要稀疏的人口即可。

两种样式的网关对应用程序都是透明的。无论使用静态网关还是动态网关，应用程序设计都不会改变。这也意味着某些应用程序可以使用服务发现，而其他应用程序则可以不使用。通常，使用动态发现意味着 SDK 的定义更少，情报更多。这是默认值。

2.连接

当应用程序连接到网关时，将提供两个选项。这些用于后续的 SDK 处理：

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

- **连接配置文件**：`connectionProfile` 是网关配置，它将用于 SDK 的静态或动态事务处理。尽管可以在传递给网关时将其转换为 JSON 对象，但是可以使用 YAML 或 JSON 指定它：

```
let connectionProfile = yaml.safeLoad(fs.readFileSync('../gateway/paperNet.yaml', 'utf8'));
```

  阅读有关连接配置文件以及如何配置它们的更多信息。

- **连接选项**：`connectionOptions` 允许应用程序声明而不是实现所需的事务处理行为。SDK 会解释连接选项，以控制与网络组件的交互模式，例如选择要连接的身份或用于事件通知的对等对象。这些选项可在不影响功能的情况下显着降低应用程序的复杂性。这是可能的，因为 SDK 已实现了许多应用程序否则需要的低层逻辑。连接选项控制此逻辑流。

  阅读有关可用连接选项的列表 以及何时使用它们的信息。

3. 静态的

静态网关定义网络的固定视图。在 MagnetoCorp 方案中，网关可能会标识 MagnetoCorp 的单个对等方，DigiBank 的单个对等方和 MagentoCorp 订购者。或者，网关可以定义 MagnetCorp 和 DigiBank 的*所有*对等方和订购方。在这两种情况下，网关都必须定义足以使商业票据交易获得认可和分发的网络视图。

通过在 API 上显式指定 connect 选项，应用程序可以静态使用网关。或者，环境变量设置 将始终覆盖应用程序选择。

```
discovery: { enabled:false }gateway.connect()FABRIC_SDK_DISCOVERY=false
```

检查 MagnetoCorp 问题应用程序使用的连接配置文件。查看如何在此文件中指定所有对等方，订购者甚至 CA，包括它们的角色。

值得牢记的是，静态网关代表某个*时刻*的网络视图。随着网络的变化，将其反映在网关文件的变化中可能很重要。当应用程序重新加载网关文件时，它们将自动获取这些更改。

4. 动态

动态网关为网络定义了一个小的固定*起点*。在 MagnetoCorp 方案中，动态网关可能只从 MagnetoCorp 识别单个对等方。一切都会被发现！（为了提供弹性，最好定义两个这样的引导对等点。）

如果应用程序选择了服务发现，则网关文件中定义的拓扑将使用此过程生成的拓扑进行扩充。服务发现从网关定义开始，并使用八卦协议在 MagnetoCorp 组织中找到所有连接的对等方和订购方。如果已为通道定义了锚点对等点，则服务发现将使用跨组织的八卦协议来发现所连接组织中的组件。此过程还将发现在对等方安装的智能合约及其在渠道级别定义的认可策略。与静态网关一样，发现的网络必须足以使商业票据交易得到认可和分发。

动态网关是 Fabric 应用程序的默认设置。可以使用 API 上的 connect 选项明确指定它们。或者，环境变量设置 将始终覆盖应用程序选择。

```
discovery: { enabled:true }gateway.connect()FABRIC_SDK_DISCOVERY=true
```

动态网关代表网络的最新视图。随着网络的变化，服务发现将确保网络视图正确反映了应用程序可见的拓扑。应用程序将自动获取这些更改；他们甚至不需要重新加载网关文件。

5. 多个网关

最后，对于应用程序而言，定义相同或不同网络的多个网关非常简单。此外，应用程序可以静态和动态使用名称网关。

拥有多个网关可能会有所帮助。原因如下：

- 代表不同用户处理请求。
- 同时连接到不同的网络。
- 通过同时比较其行为与现有配置来测试网络配置。