向通道添加组织

注解

确保你已经下载了 安装示例、二进制和 Docker 镜像 和 :doc:`prereqs`中所列出的镜像和二进制，并确定其版本与本文的版本(v2.2，版本号可在左边目录底部找到)保持一致。

本教程通过向应用通道中添加一个新的组织——Org3来扩展Fabric测试网络。

虽然我们在这里将只关注将新组织添加到通道中，但执行其他通道配置更新（如更新修改策略，调整块大小）也可以采取类似的方式。要了解更多的通道配置更新的相关过程，请查看`:doc:*config_update*。值得注意的是，像本文演示的这些通道配置更新通常是组织管理者（而非链码或者应用开发者）的职责。

环境构建

我们将从克隆到本地的 `fabric-samples` 的子目录 ``testt-network``进行操作。现在， 进入那个目录。

```
cd fabric-samples/test-network
```

首先，使用 `network.sh` 脚本清理环境。这个命令会清除所有活动状态或终止状态的容器，并且移除之前生成的构件。关闭Fabric网络并非执行通道配置升级的**必要**步骤。但是为了本教程，我们希望从一个已知的初始状态开始，因此让我们运行以下命令来清理之前的环境：

```
./network.sh down
```

现在可以执行脚本，运行带有一个命名为``mychannel``的通道的测试网络:

```
./network.sh up createChannel
```

如果上面的脚本成功执行，你能看到日志中打印出如下信息：

```
========= Channel successfully joined ===========
```

现在你的机器上运行着一个干净的测试网络版本，我们可以开始向我们创建的通道添加一个新的组织。首先，我们将使用一个脚本将Org3添加到通道中，以确认流程正常工作。然后，我们将通过更新通道配置逐步完成添加Org3的过程。

使用脚本将 Org3 加入通道

你应该在 `test-network` 目录下，简单地执行以下命令来使用脚本：

```
./eyfn.sh up
```

此处的输出值得一读。你可以看到添加了 Org3的加密材料，创建了Org3的组织定义，创建了配置更新和签名，然后提交到通道中。

如果一切顺利，你会看到以下信息：

```
========= Finished adding Org3 to your test network! =========
```

现在我们已经确认了我们可以将Org3添加到通道中，我们可以执行以下步骤来更新通道配置，以了解脚本幕后完成的工作。

手动将 Org3 加入通道

如果你刚执行了 `addOrg3.sh` 脚本，你需要先将网络关掉。下面的命令将关掉所有的组件，并移出所有组织的加密材料：

```
./addOrg3.sh down
```

网络关闭后，将其再次启动：

```
cd ..
./network.sh up createChannel
```

这会使网络恢复到执行``addOrg3.sh``脚本前的状态。

现在我们准备将Org3手动将加入到通道中。第一步，我们需要生成Org3的加密材料。

生成 Org3 加密材料

在另一个终端，切换到 `test-network` 的子目录 `addOrg3` 中。

```
cd addOrg3
```

首先，我们将为Org3的peer节点以及一个应用程序和管理员用户创建证书和密钥。因为我们在更新一个示例通道，所以我们将使用``cryptogen``工具代替CA。下面的命令使用``cryptogen``读取``org3-crypto.yaml``文件并在``org3.example.com``文件夹中生成Org3的加密材料。

```
../../bin/cryptogen generate --config=org3-crypto.yaml --output="../organizations"
```

在``test-network/organizations/peerOrganizations``目录中，你能在Org1和Org2证书和秘钥旁边找到已生成的Org3加密材料。

一旦我们生成了Org3的加密材料，我们就能使用``configtxgen``工具打印出Org3的组织定义。我们将在执行命令前告诉这个工具在当前目录去获取``configtx.yaml`` 文件。

```
export FABRIC_CFG_PATH=$PWD
../../bin/configtxgen -printOrg Org3MSP > ../organizations/peerOrganizations/org3.example.com/org3
```

上面的命令会创建一个 JSON 文件 – `org3.json` – 并将其写入到``test-network/organizations/peerOrganizations/org3.example.com``文件夹下。这个组织定义文件包含了Org3 的策略定义，还有三个 base 64 格式的重要的证书：

- 一个CA 根证书t, 用于建立组织的根信任
- 一个TLS根证书, 用于在gossip协议中识别Org3的块传播和服务发现
- 管理员用户证书 (以后作为Org3的管理员会用到它)

我们将通过把这个组织定义附加到通道配置中来实现将Org3添加到通道中。

启动Org3组件

在创建了Org3证书材料之后，现在可以启动Org3 peer节点。在addOrg3目录中执行以下命令：

```
docker-compose -f docker/docker-compose-org3.yaml up -d
```

如果命令成功执行，你将看到Org3 peer节点的创建和一个命名为Org3CLI的Fabric tools容器：

```
Creating peer0.org3.example.com ... done
Creating Org3cli                ... done
```

这个Docker Compose文件以及被配置为桥接我们的处所网络，所以Org3的peer节点和Org3CLI可以被测试网络中的peer节点和ordering节点解析。我们将使用Org3CLI容器和网络通信，并执行把Org3添加到到通道中的peer命令。

准备CLI环境

配置更新的过程利用了配置翻译工具 – configtxlator。这个工具提供了一个独立于SDK的无状态REST API。此外它还提供了一个用于简化Fabric网络配置任务的的CLI工具。该工具允许在不同的等价数据表示/格式之间进行简单的转换(在本例中是在protobufs和JSON之间)。此外，该工具可以根据两个通道配置之间的差异计算配置更新交易。

使用以下命令进入Org3CLI容器：

```
docker exec -it Org3cli bash
```

这个容器已经被挂载在``organizations``文件夹中，让我们能够访问所有组织和Orderer Org的加密材料和TLS证书。我们可以使用环境变量来操作Org3CLI容器，以切换Org1、Org2或Org3的管理员角色。首先，我们需要为orderer TLS证书和通道名称设置环境变量:

```
export ORDERER_CA=/opt/gopath/src/github.com/hyperledger/fabric/peer/organizations/ordererOrganiza
export CHANNEL_NAME=mychannel
```

检查下以确保变量已经被正确设置:

```
echo $ORDERER_CA && echo $CHANNEL_NAME
```

注解

如果出于任何原因需要重启Org3CLI容器，你还需要重新设置两个环境变量 – `ORDERER_CA` and `CHANNEL_NAME`.

获取配置

现在我们有了一个设置了 `ORDERER_CA` 和 `CHANNEL_NAME` 环境变量的 CLI容器。让我们获取通道 – `mychannel` 的最新的配置区块。

我们必须拉取最新版本配置的原因是通道配置元素是版本化的。版本管理由于一些原因显得很重要。它可以防止通道配置更新被重复或者重放攻击（例如，回退到带有旧的 CRLs的通道配置将会产生安全风险）。同时它保证了并行性（例如，如果你想从你的通道中添加新的组织后，再删除一个组织，版本管理可以帮助你移除想移除的那个组织，并防止移除两个组织）。

因为Org3还不是通道的成员，所以我们需要作为另一个组织的管理员来操作以获取通道配置。因为Org1是通道的成员，所以Org1管理员有权从ordering服务中获取通道配置。作为Org1管理员进行操作，执行以下命令。

```
# you can issue all of these commands at once

export CORE_PEER_LOCALMSPID="Org1MSP"
export CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/organization
export CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/organizations/pe
export CORE_PEER_ADDRESS=peer0.org1.example.com:7051
```

我们现在执行命令获取最新的配置块:

```
peer channel fetch config config_block.pb -o orderer.example.com:7050 -c $CHANNEL_NAME --tls --caf
```

这个命令将通道配置区块以二进制protobuf形式保存在``config_block.pb``。注意文件的名字和扩展名可以任意指定。但是，推荐遵循标识要表示的对象类型及其编码(protobuf或JSON)的约定。

当你执行 `peer channel fetch` 命令后，下面的输出将出现在你的日志中:

```
2017-11-07 17:17:57.383 UTC [channelCmd] readBlock -> DEBU 011 Received block: 2
```

这是告诉我们最新的 `mychannel` 的配置区块实际上是区块 2， **并非**初始区块。
`peer channel fetch config``命令默认返回目标通道最新的配置区块，在这个例子里是第三个区块。这是因为测试网络脚本``network.sh``分别在两个通道更新交
和 ``Org2``定义了锚节点。最终，我们有如下的配置块序列:

- block 0: genesis block
- block 1: Org1 anchor peer update
- block 2: Org2 anchor peer update

将配置转换到 JSON 格式并裁剪

现在我们用 `configtxlator` 工具将这个通道配置解码为JSON格式（以便被友好地阅读和修改）。我们也必须裁剪所有的头部、元数据、创建者签名等以及其他和我们将要做的修改无关的内容。我们通过``jq``这个工具来完成裁剪:

```
configtxlator proto_decode --input config_block.pb --type common.Block | jq .data.data[0].payload.
```

这个命令使我们得到一个裁剪后的JSON对象 – `config.json`，这个文件将作为我们配置更新的基准。

花一些时间用你的文本编辑器（或者你的浏览器）打开这个文件。即使你已经完成了这个教程， 也值得研究下它，因为它揭示了底层配置结构，和能做的其它类型的通道更新升级。我们将在 :doc:`config_update`更详细地讨论。

添加Org3加密材料

注解

目前到这里你做的步骤和其他任何类型的配置升级所需步骤几乎是一致的。我们之

所以选择在教程中添加一个组织，是因为这是能做的配置升级里最复杂的一个。

我们将再次使用 `jq` 工具去追加 Org3 的配置定义 – `org3.json` –到通道的应用组字段，同时定义输出文件是 – `modified_config.json` 。

```
jq -s '.[0] * {"channel_group":{"groups":{"Application":{"groups": {"Org3MSP":.[1]}}}}}' config.js
```

现在，我们在Org3CLI 容器有两个重要的 JSON 文件 – `config.json` 和``modified_config.json`` 。初始的文件包含 Org1 和 Org2 的材料，而”modified”文件包含了总共3个组织。现在只需要将这 2 个 JSON文件重新编码并计算出差异部分。

首先，将 `config.json` 文件倒回到 protobuf 格式，命名为 `config.pb`：

```
configtxlator proto_encode --input config.json --type common.Config --output config.pb
```

下一步，将 `modified_config.json` 编码成 `modified_config.pb`：

```
configtxlator proto_encode --input modified_config.json --type common.Config --output modified_con
```

现在使用 `configtxlator` 去计算两个protobuf配置的差异。这条命令会输出一个新的 protobuf 二进制文件，命名为``org3_update.pb``：

```
configtxlator compute_update --channel_id $CHANNEL_NAME --original config.pb --updated modified_co
```

这个新的 proto 文件 – `org3_update.pb` – 包含了 Org3 的定义和指向Org1 和 Org2 材料的更高级别的指针。我们可以抛弃 Org1和Org2相关的MSP材料和修改策略信息，因为这些数据已经存在于通道的初始区块。因此，我们只需要两个配置的差异部分。

在我们提交通道更新前，我们执行最后做几个步骤。首先，我们将这个对象解码成可编辑的JSON 格式，并命名为 `org3_update.json`：

```
configtxlator proto_decode --input org3_update.pb --type common.ConfigUpdate | jq . > org3_update.
```

现在，我们有了一个解码后的更新文件 – `org3_update.json` –我们需要用信封消息来包装它。这个步骤要把之前裁剪掉的头部信息还原回来。我们将这个文件命名为``org3_update_in_envelope.json`` 。

```
echo '{"payload":{"header":{"channel_header":{"channel_id":"'$CHANNEL_NAME'", "type":2}},"data":{"
```

使用我们格式化好的 JSON – `org3_update_in_envelope.json` –我们最后一次使用 `configtxlator` 工具将他转换为 Fabric需要的完全成熟的protobuf``格式。我们将最后的更新对象命名为 ``org3_update_in_envelope.pb`` 。

```
configtxlator proto_encode --input org3_update_in_envelope.json --type common.Envelope --output or
```

签名并提交配置更新

差不多大功告成了！

我们现在有一个 protobuf二进制文件 – `org3_update_in_envelope.pb` – 在我们的 Org3CLI容器内。但是，在配置写入到账本前，我们需要来自必要的Admin用户的签名。我们通道应用组的修改策略（mod_policy）设置为默认值"MAJORITY"，这意味着我们需要大多数已经存在的组织管理员去签名这个更新。因为我们只有两个组织 – Org1 和 Org2 – 所以两个的大多数也还是两个，我们需要它们都签名。没有这两个签名，排序服务会因为不满足策略而拒绝这个交易。

首先，让我们以 Org1 管理员来签名这个更新 proto。记住我们导出了必要的环境变量，以作为Org1管理员来操作Org3CLI容器。因此，下面的``peer channel signconfigtx``命令将更新签名为Org1。

```
peer channel signconfigtx -f org3_update_in_envelope.pb
```

最后一步，我们将容器的身份切换为 Org2管理员用户。我们通过导出和Org2 MSP相关的4个环境变量实现这步。

注解

切换不同的组织身份为配置交易签名（或者其他事情）不能反映真实世界里Fabric 的操作。一个单一容器不可能挂载了整个网络的加密材料。相反地，配置更新需要在网络外安全地递交给Org2管理员来审查和批准。

导出 Org2 的环境变量：

```
# you can issue all of these commands at once

export CORE_PEER_LOCALMSPID="Org2MSP"
export CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/organization
export CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/organizations/pe
export CORE_PEER_ADDRESS=peer0.org2.example.com:9051
```

最后，我们执行 `peer channel update` 命令。Org2管理员在这个命令中会附带签名，因 此就没有必要对 protobuf 进行两次签名:

注解

将要做的对排序服务的更新调用，会经历一系列的系统级签名和策略检查。你会发现通过检视排序节点的日志流会非常有用。在另外一个终端执行``docker logs -f orderer.example.com`` 命令就能展示它们了。

发起更新调用：

```
peer channel update -f org3_update_in_envelope.pb -c $CHANNEL_NAME -o orderer.example.com:7050 --t
```

如果你的更新提交成功，将会看到一个类似如下的信息：

```
2020-01-09 21:30:45.791 UTC [channelCmd] update -> INFO 002 Successfully submitted channel update
```

成功的通道更新调用会返回一个新的区块 – 区块3 – 给所有在这个通道上的peer节点。你是否还记得，区块 0-2是初始的通道配置，区块3就是带有Org3定义的最新的通道配置。

你可以通过进入到Org3CLI容器外的一个终端并用以下命令来检查查看 `peer0.org1.example.com` 的日志：

```
docker logs -f peer0.org1.example.com
```

将 Org3 加入通道

此时，通道的配置已经更新并包含了我们新的组织 – `Org3` – 意味者这个组织下的节点可以加入到 `mychannel`。

在Org3CLI容器中，导出一下的环境变量用来以Org3Admin的身份来进行操作：

```
# you can issue all of these commands at once

export CORE_PEER_LOCALMSPID="Org3MSP"
export CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/organization
export CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/organizations/pe
export CORE_PEER_ADDRESS=peer0.org3.example.com:11051
```

现在，让我们向排序服务发送一个调用，请求``mychannel``的创世块。由于成功地更新了通道，排序服务将验证Org3可以拉取创世块并加入该通道。如果没有成功地将Org3附加到通道配置中，排序服务将拒绝此请求。

使用 `peer channel fetch` 命令来获取这个区块：

```
peer channel fetch 0 mychannel.block -o orderer.example.com:7050 -c $CHANNEL_NAME --tls --cafile $
```

注意，我们传递了 ``0``去索引我们在这个通道账本上想要的区块（例如，创世块）。如果我们简单地执行``peer channel fetch config``命令，我们将会收到区块 3 – 那个带有Org3定义的更新后的配置。然而，我们的账本不能从一个下游的区块开始 – 我们必须从区块 0 开始。

如果成功，该命令将创世块返回到名为``mychannel.block``的文件。我们现在可以使用这个块来连接到通道的peer端。执行``peer channel join``命令并传入创世块，以将Org3的peer节点加入到通道中:

```
peer channel join -b mychannel.block
```

配置领导节点选举

注解

引入这个章节作为通用参考，是为了理解在完成网络通道配置初始化之后，增加组织时，领导节点选举的设置。这个例子中，默认设置为动态领导选举，这是为网络中所有的节点设置的。

新加入的节点是根据初始区块启动的，初始区块是不包含通道配置更新中新加入的组织信息的。因此新的节点无法利用gossip协议，因为它们无法验证从自己组织里其他节点发送过来的区块，除非它们接收到将组织加入到通道的那个配置交易。新加入的节点必须有以下配置之一才能从排序服务接收区块：

1. 采用静态领导者模式，将peer节点配置为组织的领导者。

```
CORE_PEER_GOSSIP_USELEADERELECTION=false
CORE_PEER_GOSSIP_ORGLEADER=true
```

注解

这个配置对于新加入到通道中的所有节点必须一致。

2. 采用动态领导者选举，配置节点采用领导选举的方式：

```
CORE_PEER_GOSSIP_USELEADERELECTION=true
CORE_PEER_GOSSIP_ORGLEADER=false
```

注解

因为新加入组织的节点，无法生成成员关系视图，这个选项和静态配置类似，每

个节点启动时宣称自己是领导者。但是，一旦它们更新到了将组织加入到通道的配置交易，组织中将只会有一个激活状态的领导者。因此，如果你想最终组织的节点采用领导选举，建议你采用这个配置。

安装、定义和调用链码

我们可以通过在通道上安装和调用链码来确认Org3是``mychannel``的成员。如果现有的通道成员已经向该通道提交了链码定义，则新组织可以通过批准链码定义来开始使用该链码。

注解

这些链码生命周期指令是在v2.0 release版本中引入的。如果你想要使用先前的生命周期去安装和实例化链码，可参考v1.4版本的`Adding an org to a channel tutorial <https://hyperledger-

fabric.readthedocs.io/en/release-1.4/channel_update_tutorial.html>`__.

在我们以Org3来安装链码之前，我们可以使用 `./network.sh` 脚本在通道上部署Fabcar链码。在Org3CLI容器外打开一个新的终端，并进入 `test-network` 目录。然后你可以使用 `test-network` 脚本来部署 ``Fabcar``链码:

```
cd fabric-samples/test-network
./network.sh deployCC
```

该脚本将在Org1和Org2的peer节点上安装Fabcar链码，批准Org1和Org2的链码定义，然后将链码定义提交给通道。一旦将链码定义提交到通道，就会初始化Fabcar链码并调用它来将初始数据放到账本上。下面的命令假设我们仍在使用 `mychannel` 通道。

在部署了链码之后，我们可以使用以下步骤在Org3中调用Fabcar链代码。这些步骤可以在 `test-network` 目录中完成，而不必在Org3CLI容器中执行。在你的终端中复制和粘贴以下环境变量，以便以Org3管理员的身份与网络交互:

```
export PATH=${PWD}/../bin:$PATH
export FABRIC_CFG_PATH=$PWD/../config/
export CORE_PEER_TLS_ENABLED=true
export CORE_PEER_LOCALMSPID="Org3MSP"
export CORE_PEER_TLS_ROOTCERT_FILE=${PWD}/organizations/peerOrganizations/org3.example.com/peers/p
export CORE_PEER_MSPCONFIGPATH=${PWD}/organizations/peerOrganizations/org3.example.com/users/Admin
export CORE_PEER_ADDRESS=localhost:11051
```

第一步是打包Fabcar链码：

```
peer lifecycle chaincode package fabcar.tar.gz --path ../chaincode/fabcar/go/ --lang golang --labe
```

这个命令会创建一个链码包，命名为``fabcar.tar.gz``，用它来在我们的Org3的peer节点上安装链码。如果通道中运行的是java或者Node.js语言写的链码，需要根据实际情况修改这个命令。输入下面的命令在peer0.org3.example.com上安装链码：

```
peer lifecycle chaincode install fabcar.tar.gz
```

下一步是以Org3的身份批准链码Fabcar定义。Org3需要批准与Org1和Org2同样的链码定义，然后提交到通道中。为了调用链码，Org3需要在链码定义中包含包标识符。你可以在你的peer中查到包标识：

```
peer lifecycle chaincode queryinstalled
```

你应该会看到类似下面的输出：

```
Get installed chaincodes on peer:
Package ID: fabcar_1:25f28c212da84a8eca44d14cf12549d8f7b674a0d8288245561246fa90f7ab03, Label: fabc
```

我们后面的命令中会需要这个包标识。所以让我们继续把它保存到环境变量。把``peer lifecycle chaincode queryinstalled``返回的包标识粘贴到下面的命令中。这个包标识每个用户可能都不一样，所以需要使用从你控制台返回的包标识完成下一步。

```
export CC_PACKAGE_ID=fabcar_1:25f28c212da84a8eca44d14cf12549d8f7b674a0d8288245561246fa90f7ab03
```

使用下面的命令来为Org3批准链码Fabcar定义:

```
# use the --package-id flag to provide the package identifier
# use the --init-required flag to request the ``Init`` function be invoked to initialize the chain
peer lifecycle chaincode approveformyorg -o localhost:7050 --ordererTLSHostnameOverride orderer.ex
```

你可以使用 `peer lifecycle chaincode querycommitted` 命令来检查你批准的链码定义是否已经提交到通道中。

```
# use the --name flag to select the chaincode whose definition you want to query
peer lifecycle chaincode querycommitted --channelID mychannel --name fabcar --cafile ${PWD}/organi
```

命令执行成功后会返回关于被提交的链码定义的信息:

```
ommitted chaincode definition for chaincode 'fabcar' on channel 'mychannel':
Version: 1, Sequence: 1, Endorsement Plugin: escc, Validation Plugin: vscc, Approvals: [Org1MSP: t
```

Org3在批准提交到通道的链码定义后，就可以使用Fabcar链码了。链码定义使用默认的背书策略，该策略要求通道上的大多数组织背书一个交易。这意味着，如果一个组织被添加到通道或从通道中删除，背书策略将自动更新。我们之前需要来自Org1和Org2的背书(2个中的2个)，现在我们需要来自Org1、Org2和Org3中的两个组织的背书(3个中的2个)。

你可以查询链码，以确保它已经在Org3的peer上启动。注意，你可能需要等待链码容器启动。

```
eer chaincode query -C mychannel -n fabcar -c '{"Args":["queryAllCars"]}'
```

你应该看到作为响应添加到账本中的汽车的初始列表。

现在，调用链码将一辆新车添加到账本中。在下面的命令中，我们以Org1和Org3中的peer为目标，以收集足够数量的背书。

```
peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride orderer.example.com --tls --c
```

我们再次查看下账本中的新车，发现"CAR11"已结在我们的账本中了：

```
peer chaincode query -C mychannel -n fabcar -c '{"Args":["queryCar","CAR11"]}'
```

总结

通道配置更新过程确实非常复杂，但是各个步骤都有一个逻辑方法。最后就是为了形成一个用protobuf二进制表示的差异化的交易对象，然后获取必要数量的管理员签名来使通道配置更新交易满足通道的修改策略。

`configtxlator` 和 `jq` 工具，和不断使用的 `peer channel` 命令，为我们提供了完成这个任务的基本功能。

更新通道配置包括Org3的锚节点（可选）

因为Org1和Org2在通道配置中已经定义了锚节点，所以Org3的节点可以与Org1和Org2的节点通过gossip协议进行连接。同样，像Org3这样新添加的组织也应该在通道配置中定义它们的锚节点，以便来自其他组织的任何新节点可以直接发现Org3节点。在本节中，我们将对通道配置进行更新，以定义Org3锚节点。这个过程将类似于之前的配置更新，因此这次我们会更快。

如果你没有进入到Org3CLI容器，执行:

```
docker exec -it Org3cli bash
```

如果尚未设置$ORDERER_CA和$CHANNEL_NAME变量，执行：

```
export ORDERER_CA=/opt/gopath/src/github.com/hyperledger/fabric/peer/organizations/ordererOrganiza
export CHANNEL_NAME=mychannel
```

和以前一样，我们开始会获取最新的通道配置。在Org3的CLI容器中获取通道中最近的配置区块， 使用``peer channel fetch``命令。

```
peer channel fetch config config_block.pb -o orderer.example.com:7050 -c $CHANNEL_NAME --tls --caf
```

在获取到配置区块后，我们将要把它转换成JSON格式。为此我们会使用configtxlator工具，正如前面在通道中加入Org3一样。当转换时，我们需要删除所有更新Org3不需要的头部、元数据和签名，使用jq工具添加包含一个锚节点的Org3更新。这些信息会在更新通道配置前重新合并。

```
configtxlator proto_decode --input config_block.pb --type common.Block | jq .data.data[0].payload.
```

``config.json``就是现在修剪后的JSON文件，表示我们要更新的最新的通道配置。

再使用jq工具，我们将想要添加的Org3锚节点更新在JSON配置中。

```
jq '.channel_group.groups.Application.groups.Org3MSP.values += {"AnchorPeers":{"mod_policy": "Admi
```

现在我们有两个JSON文件了，一个是当前的通道配置``config.json``，另外一个是期望的通道配置``modified_anchor_config.json``。 接下来我们依次转换成protobuf格式，并计算他们之间的增量。

把``config.json``翻译回protobuf格式``config.pb``。

```
configtxlator proto_encode --input config.json --type common.Config --output config.pb
```

把``modified_anchor_config.json``翻译回protobuf格式``modified_anchor_config.pb``。

```
configtxlator proto_encode --input modified_anchor_config.json --type common.Config --output modif
```

计算这两个protobuf``格式配置的增量。

```
configtxlator compute_update --channel_id $CHANNEL_NAME --original config.pb --updated modified_an
```

现在我们已经有了期望的通道更新，下面必须把它包在一个信封消息里以便正确读取。要做到这一点，我们先把protobuf格式转换回JSON格式才能被包装。

我们再此使用configtxlator命令，把``anchor_update.pb``转换成``anchor_update.json``。

```
configtxlator proto_decode --input anchor_update.pb --type common.ConfigUpdate | jq . > anchor_upd
```

接下来我们来把更新包在信封消息里，恢复先前去掉的头，输出到``anchor_update_in_envelope.json``中。

```
echo '{"payload":{"header":{"channel_header":{"channel_id":"'$CHANNEL_NAME'", "type":2}},"data":{"
```

现在我们已经重新合并了信封，我们需要把它装换成protobuf格式以便正确签名并提交到orderer进行更新。

```
configtxlator proto_encode --input anchor_update_in_envelope.json --type common.Envelope --output
```

现在更新已经被正确格式化，是时候签名并提交了。因为这只是对Org3做更新，我们只需要Org3对更新签名。为了确保我们以Org3的管理员身份操作，运行以下命令：

```
# you can issue all of these commands at once

export CORE_PEER_LOCALMSPID="Org3MSP"
export CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/organization
export CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/organizations/pe
export CORE_PEER_ADDRESS=peer0.org3.example.com:11051
```

在将更新提交给order之前，现在我们以Org3 admin身份使用``peer channel update``命令进行签名。

```
peer channel update -f anchor_update_in_envelope.pb -c $CHANNEL_NAME -o orderer.example.com:7050 -
```

orderer接收到配置更新请求，用这个配置更新切分成区块。当节点接收到区块后，他们就会处理配置更新了。

检查其中一个peer节点的日志。当处理新区块带来的配置更新时，你会看到gossip使用新的锚节点与Org3重新建立连接。这就证明了配置更新已经成功应用。

```
docker logs -f peer0.org1.example.com
```

```
2019-06-12 17:08:57.924 UTC [gossip.gossip] learnAnchorPeers -> INFO 89a Learning about the config
2019-06-12 17:08:57.926 UTC [gossip.gossip] learnAnchorPeers -> INFO 89b Learning about the config
2019-06-12 17:08:57.926 UTC [gossip.gossip] learnAnchorPeers -> INFO 89c Learning about the config
```

恭喜，你已经成功做了两次配置更新 – 一个是向通道加入Org3，第二个是在Org3中定义锚节点。