

Java 開発者のためのブロックチェーン・チェーンコード

Hyperledger Fabric v0.6 のチェーンコードを Java 言語を使用して作成する方法

J Steven Perry

Principal Consultant

Makoto Consulting Group, Inc.

2017年 9月 14日

現在、ブロックチェーンが、インターネット上でビジネス・トランザクションを実行する方法を変えつつあります。このチュートリアルでは、ブロックチェーン・ネットワークをインストールして、Java 言語で作成されたチェーンコード (スマート・コントラクト) を実行する手順を説明します。

このビデオ, Java 開発者向けチェーンコード入門, をご覧になるには、オンライン記事を参照ください。

皆さんはおそらくブロックチェーンについて耳にしたことがあると思いますが、Java 開発者としての自分にブロックチェーンがどのように関わってくるのかについては、疑問を持っていることでしょう。このチュートリアルが、ブロックチェーンに関する曖昧な点をすべて解消します。このチュートリアルでは段階的なアプローチで、Hyperledger Fabric v0.6 を使用して Java で作成されたスマート・コントラクト、つまりチェーンコードをビルドし、実行する方法を説明します。手順に従って、ツールをインストールし、ローカル・ブロックチェーン・ネットワークを定義し、最後にチェーンコード (スマート・コントラクト) をビルドして実行する方法を学んでください。

ブロックチェーンの概要については、このリンク先の developerWorks のブログ「[What is blockchain? A Primer on Distributed Ledger Technology](#)」を参照してください。



選び抜かれた無料のツール、トレーニング、コミュニティ・リソースに関する情報を毎月まとめて入手して、ブロックチェーンの運用に役立ててください。

[最新号](#) | [購読](#)

前提条件

このチュートリアルでは、読者が以下の前提条件を満たしていることを前提とします。

- ブロックチェーンまたは Hyperledger Fabric フレームワークについて聞いたことがある。

- Java 言語と Java プラットフォームの両方について、ある程度のプログラミングの知識と経験があること。
- 以下のツールを十分理解しているか、(理想的には) 使い方を習熟していること。
 - Eclipse IDE
 - Docker および Docker Compose
 - Gradle
 - Linux コマンド・ライン
 - SoapUI または別の HTTP クライアント・ソフトウェア (Postman など)

また、最小限のガイダンスでコンピューターにソフトウェアをインストールできるようにでなければなりません。スペースの関係上、必要なすべてのソフトウェアのインストール手順を記載することはしません。インストール手順については、それぞれのソフトウェアの Web サイトを参照してください。

チュートリアルの本題に入る前に、ブロックチェーンについて簡単に説明しておきたいと思います。

ブロックチェーンの基本

ブロックチェーンにまつわる誇大宣伝が溢れていますが、それは当然のことです。このテクノロジーそのものが画期的であるだけでなく、ブロックチェーンには、インターネット上でのこれまでのビジネスのやり方を覆し、激変させる可能性があります。

ブロックチェーン開発者向けのお勧めコンテンツ

このリンク先の developerWorks Blockchain Developer Center で数々のチュートリアル、コース、ブログ、コミュニティ・サポートにアクセスしてスキルを磨いてください。

その理由を理解するには、ビジネス取引の成功に不可欠な 3 つの属性について考える必要があります。

- **信用:** 取引の契約をするとしても、自分が相手に対して、あるいは相手が自分に対して、その契約を順守することを完全に信用できなければ取引は成り立ちません。
- **透明性:** 透明性によって、「カーテンの陰」を覗くことができるようになります (信用を築けるだけでなく、信用を築く必要性も低くすることができます)。
- **説明責任:** 契約が順守されたことに参加者全員が合意するかどうかを決定する基準を定義します。

契約当事者間のビジネス関係の健全性は、以上の 3 つの属性のさまざまなレベルを意味しますが (例えば、信用が高ければ高いほど透明性は必要なくなり、信用が低ければ、それだけ透明性が必要になります)。レベルはどうであろうと、この 3 つすべての属性が 1 つでも欠けていると問題になります。

“ブロックチェーン・テクノロジーは、Java 開発者のソフトウェア開発プロジェクトにも急速に進出つつあります。そのための準備をしませんか？”

ブロックチェーンは、この3つの属性にどのように役立つのでしょうか？第一に、共通のフレームワークを使用することで、ビジネス・パートナーは前もって信用のネットワークを確立できます。第二に、すべてのビジネス・パートナーに可視となるレジャーを使用することで、ブロックチェーンは透明性を提供します。第三に、スマート・コントラクト (チェーンコード) という形で関係者全員のコンセンサスをとることで、説明責任が生まれます。

これは、Java 開発者にとっては、どのような意味を持つのでしょうか？

Hyperledger コミュニティーと Hyperledger Fabric が急速に成長しているということは、Java 開発者のソフトウェア開発プロジェクトにもブロックチェーン・テクノロジーが急速に進出しつつあることを意味します。そのための準備をしませんか？

ブロックチェーン・テクノロジーのランドスケープ

ときとして、ビジネス問題の解決へと至る道を開発テクノロジーが阻むことがあります。このチュートリアルで私が主な目標としているのは、皆さんに Java でチェーンコードを作成する方法を学んでもらうことなので、ここでは最もシンプルな開発スタックを使用することにしました。

最もシンプルな開発スタックとは言っても、そのコンポーネントには他の選択肢があります。このチュートリアルではネットワーク・コンテナ環境として [Docker](#) を使用しますが、[VirtualBox](#) と [Vagrant](#) を組み合わせて使用することもできます。Vagrant を使ったことがないとしたら、一度は試してみるべきです。

Docker はコンテナ環境ですが、Vagrant では仮想化を利用します。Vagrant の仮想化環境を VirtualBox と結合すると、コンピューティング環境を別のレベルで制御できるようになります (このことから、ファブリック開発者にとっては最適な選択肢となっています)。

コンテナ化と仮想化の違いについて詳しく学ぶには、このリンク先の developerWorks ブログ「[What is Docker? A Primer on the Benefits of Containers for Applications](#)」をご覧ください。

IBM Bluemix は、コンテナや仮想化、さらにはインフラストラクチャーについても懸念せずに、コードの作成だけに専念したいという開発者にとって最適な選択肢です。Bluemix では完全な IBM Blockchain ネットワークを管理できますが、Java 言語を使用したチェーンコードの開発をまだサポートしていません。この状況は間もなく変わると思うので、見守っててください。

雪崩から生まれた氷の彫刻？

(控えめに言ったとしても) Hyperledger Fabric は極めて流動的な状態です。たまたまプロジェクトの一部が機能しなくなることがあるとは思いますが、慌てないでください。生まれたばかりのテクノロジーにはよくあることです。

例えば、このチュートリアルに記載している資料のリンク先に飛ばないとしたら、コンテンツがなくなったわけではなく、別の場所に移動された可能性が考えられます。

進化途中のテクノロジーを扱う場合は、同じような落とし穴があるものです。テクノロジーを他に先駆けて採用するということは、その進化に適応しなければならないということなので、進化の流れに身を任せてください。

現時点では、ブロックチェーン・テクノロジーのランドスケープがかなり流動的だという印象を持っているとしたら、まさにその通りです。けれどもそれと同時に、これは、ブロックチェーン

とチェーンコードの進化に最初から関わる絶好のタイミングであることを意味します。ブロックチェーンが成熟するにつれ、初期段階からこのテクノロジーの習得に費やした投資が大きな利益をもたらすことになるでしょう。

ブロックチェーンは、一般的なビジネスのやり方を激変させる可能性のある革新的テクノロジーの 1 つです。B2B だけではなく。B2C、そして最終的には C2C のあり方も変える可能性があります。ブロックチェーンを導入するには、今が実にエキサイティングな時期なのです。

それは、手順を開始しましょう！

開発環境をセットアップする

チェーンコードを実行するには、何よりもまず、開発環境をセットアップする必要があります。

このセクションを完了すると、Hyperledger Java チェーンコードのサンプルを実行できる環境が整います。その環境で、サンプルのうちの 1 つをデプロイし、実際のチェーンコードに対してトランザクションを呼び出す方法を説明した後、(ほぼ) ゼロから新しいチェーンコード・プログラムを作成する方法を説明します。

このビデオ、開発環境をセットアップする動画、をご覧になるには、オンライン記事を参照ください。

このセクションの内容は次のとおりです。

1. ネットワーク環境をセットアップする (これによって、ローカル・ブロックチェーン・ネットワークを実行できるようになります)。
2. ビルド・ソフトウェアをインストールする (これによって、チェーンコードをビルドできるようになります)。
3. HTTP クライアントをインストールする (これによって、チェーンコードに対してトランザクションを呼び出せるようになります)。
4. ブロックチェーン・ネットワークを起動する。
5. Java shim クライアント JAR をビルドする。

正直に言って、チェーンコードを作成するには、かなりのセットアップ作業が必要になります。けれども、ここで説明する手順に短期集中型で取り組めば、その作業は報われるはずです。

ネットワーク環境をセットアップする

このチュートリアルでは、ローカル・ブロックチェーン・ネットワークを実行するために、Docker と、Docker Hub から提供されている事前ビルドされたブロックチェーン・ネットワーク・コンポーネント・イメージを使用します。お望みであればゼロからファブリックを作成することもできますが (結局のところ、これはオープンソースです)、この段階では、Docker Hub から入手できる事前ビルドされた Hyperledger Fabric イメージを使用するほうが事は容易に運びます。

前述のとおり、Docker 以外の選択肢としては (Hyperledger の資料にもあるように) Vagrant と VirtualBox を使用することもできます。Vagrant はファブリック開発者にはぴったりの選択肢ですが、チェーンコード開発者としては、ファブリックそのものに取り組むよりも、チェーンコードをビルド、実行、テストすることのほうが重要です。

Docker バージョン 1.12 以降をすでにインストールしている場合は、次のセクション（「[ビルド・ソフトウェアをインストールする](#)」）にスキップして構いません。このセクションで説明する手順では、Docker がまだインストールされていないことを前提としています（つまり、Docker を前のバージョンからアップグレードするものではありません）。Docker をインストールすると、Docker Compose もインストールされます。Docker Compose は、複数のコンテナが必要となるアプリケーションを定義して実行するためのツールです。このチュートリアルで実行するローカル Hyperledger ブロックチェーン・ネットワークも、そのようなアプリケーションに該当します。

Docker をインストールする

Mac、Windows、および Linux でのインストール手順については、以下のリンク先のページを参照してください。

[Docker を Mac、Windows、および Linux にインストールする手順](#)

Docker インストール済み環境を検証する

Docker インストール済み環境をテストするには、ターミナル・ウィンドウ (Windows の場合は、コマンド・プロンプト) を開いて、以下のコマンドを入力します。

```
docker -v
docker-compose -v
```

以下のような出力が表示されたら、Docker は正常にインストールされていることになります。

```
$ docker -v
Docker version 1.13.1, build 092cba3
$ docker-compose -v
docker-compose version 1.11.1, build 7c5d5e4
```

Docker の動作を確認するには、以下のようにして hello-world イメージを実行します。

```
$ docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
78445dd45222: Pull complete
Digest: sha256:c5515758d4c5e1e838e9cd307f6c6a0d620b5e07e6f927b07d05f6d12a1ac8d7
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
3. The Docker daemon created a new container from that image which runs the
   executable that produces the output you are currently reading.
4. The Docker daemon streamed that output to the Docker client, which sent it
   to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
$ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
https://cloud.docker.com/

For more examples and ideas, visit:
```

<https://docs.docker.com/engine/userguide/>

ビルド・ソフトウェアをインストールする

Hyperledger Fabric ではビルド・システムとして Gradle を使用するの、このチュートリアルでも Gradle を使用します。Gradle は、単純な構文を使用してビルド・コンポーネントを指定できる、使いやすいビルド・システムであるというだけでなく、Apache Ant と Apache Maven の最良の機能を備えた強力なビルド自動化システムです。かなり多くの開発者がプロジェクトを Gradle に切り替えているのも、不思議なことではありません。

Gradle の詳細 (および注目を集めている Gradle ユーザー) については、[このリンク先の Gradle メイン・ページを参照してください](#)。

Gradle をインストールする

Gradle をインストールするには、以下のリンク先のページで説明している手順に従ってください。

[Gradle を Mac、Windows、および Linux にインストールする手順](#)

Gradle インストール済み環境を検証する

Gradle インストール済み環境を検証するには、ターミナル・ウィンドウを開いて、以下のコマンドを実行します。

```
gradle -v
```

以下のような出力が表示されたら、Gradle は正常にインストールされていることになります。

```
$ gradle -v

-----
Gradle 3.3
-----

Build time:   2017-01-03 15:31:04 UTC
Revision:     075893a3d0798c0c1f322899b41ceca82e4e134b

Groovy:       2.4.7
Ant:          Apache Ant(TM) version 1.9.6 compiled on June 29 2015
JVM:          1.8.0_102 (Oracle Corporation 25.102-b14)
OS:           Mac OS X 10.12.3 x86_64
```

HTTP クライアントをインストールする

次は、チェーンコードが Hyperledger ブロックチェーン・ファブリックの REST インターフェースと通信できるようにするために、HTTP クライアント・ソフトウェアをインストールします。ブラウザで HTTP GET を発行することはできますが、ファブリックとやりとりするためには、POST メッセージを送信できなければなりません。つまり、HTTP クライアントが必要になるということです。

このチュートリアルで私が HTTP クライアントとして選んだのは、SoapUI です。SoapUI には、強力で使いやすく、しかも多数の機能が含まれる無料のコミュニティー・エディションが用意されています。

SoapUI をインストールする

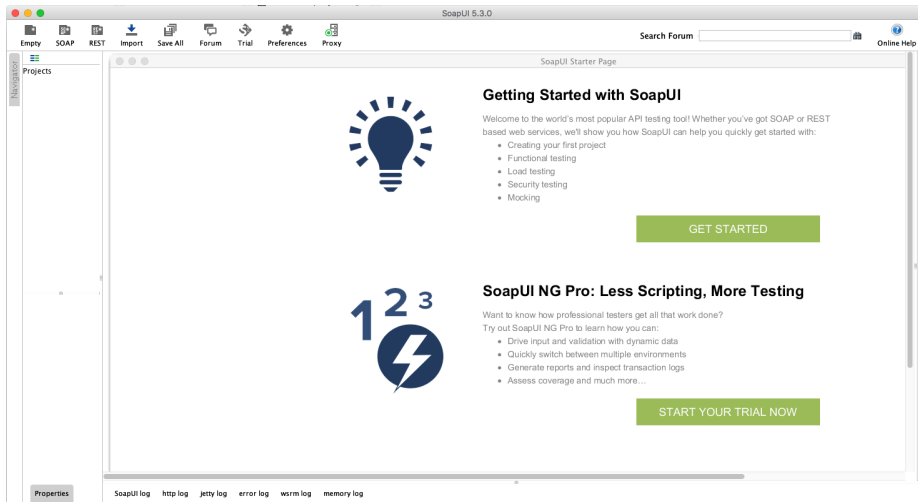
SoapUI をインストールするには、以下のリンク先のページで説明している手順に従ってください。

[SoapUI を Mac、Windows、および Linux にインストールする手順](#)

SoapUI インストール済み環境を検証する

SoapUI がインストールされたことを確認するには、このアプリケーションをコンピューター上で起動します。Mac OS 上では、SoapUI を起動すると「SoapUI Starter Page (SoapUI スターター・ページ)」が表示されます (図 1 を参照)。

図 1. Mac OS X 上の SoapUI



ブロックチェーン・ネットワークを起動する

GOPATH

Hyperledger Fabric は Go で作成されており、Hyperledger の資料では度々 GOPATH という言葉を目にするはずです。そのため、Hyperledger を使用してチェーンコードの開発に力を入れることを予定している場合は、GOPATH について十分に理解しておく必要があります。

GOPATH は、Go 環境のルートです。ソース・コード、バイナリー、およびその他の Golang パッケージはすべて、このパスを基準に参照されます。

チェーンコードを開発してテストするために必要なソフトウェアをインストールしたので、次は、ローカル・ブロックチェーン・ネットワークを起動します。そのための最初のステップは、ネットワークの構成を定義することです。

まず、チェーンコードの開発で使用するすべてのソース・コードのルートとしての役割を果たすディレクトリを作成します。このチュートリアルでは、~/home/mychaincode (Windows の場合は C:\home\chaincode) をルートとして使用します。

ルート・ディレクトリを作成したら、GOPATH 環境変数をそのディレクトリ・パスに設定します。このチュートリアルでは Go コードをコンパイルすることも、Golang パッケージやその他のバイナリーをビルドすることもしませんが、Hyperledger には Golang 用語が織り込まれているため、Go 言語と GOPATH の観点から考えることに慣れておくのが得策です。

Linux 上では、以下のコマンドを実行します。

```
export GOPATH=~/home/mychaincode
```

Windows 上では、以下のようなコマンドを使用することになります。

```
SET GOPATH=C:\home\mychaincode
```

次に、Docker Compose に対して、ブロックチェーン・ピア・ネットワークを構成および実行する方法を指示する必要があります。ネットワークの定義は [YAML](#) 形式なので、その設定ファイルには docker-compose.yml という名前を付けてください。別のファイル名にすることもできますが、その場合は、Docker Compose を起動するときに `-f` フラグを指定しなければなりません。したがって、デフォルトのファイル名、つまり docker-compose.yml に従うことをお勧めします。

GOPATH のルート内で docker-compose.yml ファイルを作成し、このファイルの内容として以下のコードを貼り付けます。

```
membersvc:  
  image: hyperledger/fabric-membersvc  
  ports:  
    - "7054:7054"  
  command: membersvc  
vp0:  
  image: hyperledger/fabric-peer  
  ports:  
    - "7050:7050"  
    - "7051:7051"  
    - "7053:7053"
```

```
environment:
  - CORE_PEER_ADDRESSAUTODETECT=true
  - CORE_VM_ENDPOINT=unix:///var/run/docker.sock
  - CORE_LOGGING_LEVEL=DEBUG
  - CORE_PEER_ID=vp0
  - CORE_PEER_PKI_ECA_PADDR=membersrv:7054
  - CORE_PEER_PKI_TCA_PADDR=membersrv:7054
  - CORE_PEER_PKI_TLSCA_PADDR=membersrv:7054
  - CORE_SECURITY_ENABLED=false
  - CORE_SECURITY_ENROLLID=test_vp0
  - CORE_SECURITY_ENROLLSECRET=MwYpmSRjupbT
links:
  - membersrv
command: sh -c "sleep 5; peer node start --peer-chaincodedev"
```

このファイルの内容のほとんどはチュートリアル範囲外ですが、いくつかの点を指摘しておきたいと思います。

- このファイルは Docker Compose に対し、以下の 2 つのサービスを定義するように指示します。
 - `membersrv`: メンバーシップ・サービスを提供するメンバー・サービス・ノードです。具体的には、このノードが認証局 (CA) の役割を果たし、暗号化ロジスティクス (証明書の発行や失効など) のすべてを処理します。メンバー・サービス・ノードには、`hyperledger/fabric-membersrv` という名前のビルド済み Docker イメージを使用します。
 - `vp0`: ネットワーク内の孤立した検証ピア・ノードです。開発を目的とする場合は、大規模な検証ピア・ネットワークは必要ありません。ピアが 1 つあれば十分です。このノードには、`hyperledger/fabric-peer` という名前のビルド済み Docker イメージを使用します。
 - `vp0` ピアによって多数の環境関数が設定されます。これらの変数のうち、`CORE_LOGGING_LEVEL` 変数が `DEBUG` に設定されていることに注意してください。時として、この設定によって生成される大量の出力が役に立つことはありますが、出力の量を抑えたい場合は `INFO` レベルに変更してください。ロギング・レベルについて詳しくは、このリンク先の Hyperledger セットアップ・マニュアルに記載されている「[Logging Control](#)」を参照してください。
- Docker Compose の YML ファイル定義について詳しくは、[このリンク先の Docker Web サイトを参照してください](#)。

上記のコードで、`CORE_SECURITY_ENABLED` の値が `false` に設定されていることに注意してください。この設定は、何らかのエンド・ユーザー証明書をファブリックに送信する必要がないことを意味します。

最後にもう 1 つ、注意事項があります。上記のデフォルト値のいずれか (特にポートの値) を変更すると、このチュートリアルで使用するサンプル・コードが機能しなくなる可能性があります。ブロックチェーン・ネットワークは分散された複数のソフトウェア・コンポーネントからなり、これらのコンポーネント間の通信は正確に調整されなければなりません。ファブリックを構成するすべてのコンポーネントがどのように相互作用するのかを理解するまでは、ポートの値はデフォルトのままにすることを強く推奨します。

これで、ブロックチェーン・ネットワークが定義されて、ローカル・ブロックチェーン・ネットワークを起動する準備が整いました。ローカル・ブロックチェーン・ネットワークを起動するには、Docker Compose を実行します。カレント・ディレクトリーを \$GOPATH に変更して、以下のコマンドを実行します。

```
docker-compose up
```

ターミナル・ウィンドウに、以下のような出力が表示されるはずです。

```
$ docker-compose up
.
.
Pulling membersvc (hyperledger/fabric-membersvc:latest)...
latest: Pulling from hyperledger/fabric-membersvc
.
.
Status: Downloaded newer image for hyperledger/fabric-membersvc:latest
Pulling vp0 (hyperledger/fabric-peer:latest)...
latest: Pulling from hyperledger/fabric-peer
.
.
Status: Downloaded newer image for hyperledger/fabric-peer:latest
Creating mychaincode_membersvc_1
Creating mychaincode_vp0_1
Attaching to mychaincode_membersvc_1, mychaincode_vp0_1
vp0_1          | 19:30:03.773 [logging] LoggingInit -> DEBU 001 Setting default logging level
to DEBUG for command 'node'
vp0_1          | 19:30:03.773 [nodeCmd] serve -> INFO 002 Running in chaincode development
mode
.
.
.
vp0_1          | 19:30:04.146 [peer] chatWithSomePeers -> DEBU 07c Starting up the first peer
of a new network
vp0_1          | 19:30:04.146 [consensus/statetransfer] verifyAndRecoverBlockchain -> DEBU 07d
Validating existing blockchain, highest validated block is 0, valid through 0
vp0_1          | 19:30:04.146 [consensus/statetransfer] blockThread -> INFO 07e Validated
blockchain to the genesis block
vp0_1          | 19:30:04.146 [consensus/handler] 1 -> DEBU 07f Starting up message thread for
consenter
vp0_1          | 19:30:04.146 [nodeCmd] serve -> INFO 080 Starting peer with ID=name:"vp0" ,
network ID=dev, address=172.17.0.3:7051, rootnodes=, validator=true
vp0_1          | 19:30:04.146 [rest] StartOpenchainRESTServer -> INFO 081 Initializing the
REST service on 0.0.0.0:7050, TLS is disabled.
vp0_1          | 19:30:04.147 [peer] ensureConnected -> DEBU 082 Starting Peer reconnect
service (touch service), with period = 6s
.
.
```

この出力は、ネットワークが稼働中になり、チェーンコード登録リクエストを受け入れられる状態になったことを意味します。

注: ブロックチェーン・ネットワークを初めて実行する際に、Docker は Docker Hub からイメージをダウンロードしなければならないため、以下の出力内で強調表示されている行は、ブロックチェーン・ネットワークの初回実行時にだけ表示されます。イメージがコンピューターにダウンロードされた後、Docker が Docker Hub からイメージをプルするのは、Docker Hub 上のイメージがコンピューター上のイメージよりも新しくなっている場合のみです。

これで、Java 言語のチェーンコードが Hyperledger Fabric ファブリックと通信するために必要な Java shim クライアント JAR をビルドする準備ができました。

Java shim クライアント JAR をビルドする

サンプル・チェーンコードを実行するには、その前に、Hyperledger の GitHub リポジトリから最新のソース・コードを入手する必要があります。

まず、チェーンコードをビルドするために、Hyperledger Fabric をローカル・マシンに複製します (注: この方法は一時的な手段です。何らかの時点で、中央の Maven リポジトリから Java shim クライアント JAR にアクセスできるようにする必要があります)。

注: 前の作業で、GOPATH を Linux (または Mac) 上の ~/home/mychaincode あるいは Windows 上の C:\home\mychaincode に設定したことを思い出してください。

ファブリック・ビルド・スクリプトが期待するディレクトリー構造を作成するために、以下のコマンドを実行します。

```
mkdir -p $GOPATH/src/github.com/hyperledger
```

次に、カレント・ディレクトリーを、新しく作成したディレクトリー構造の最下部にあるディレクトリーに変更します。

```
cd $GOPATH/src/github.com/hyperledger
```

ここから、Hyperledger ソース・コードを取得して、Java shim クライアント JAR をビルドできるようにする必要があります。

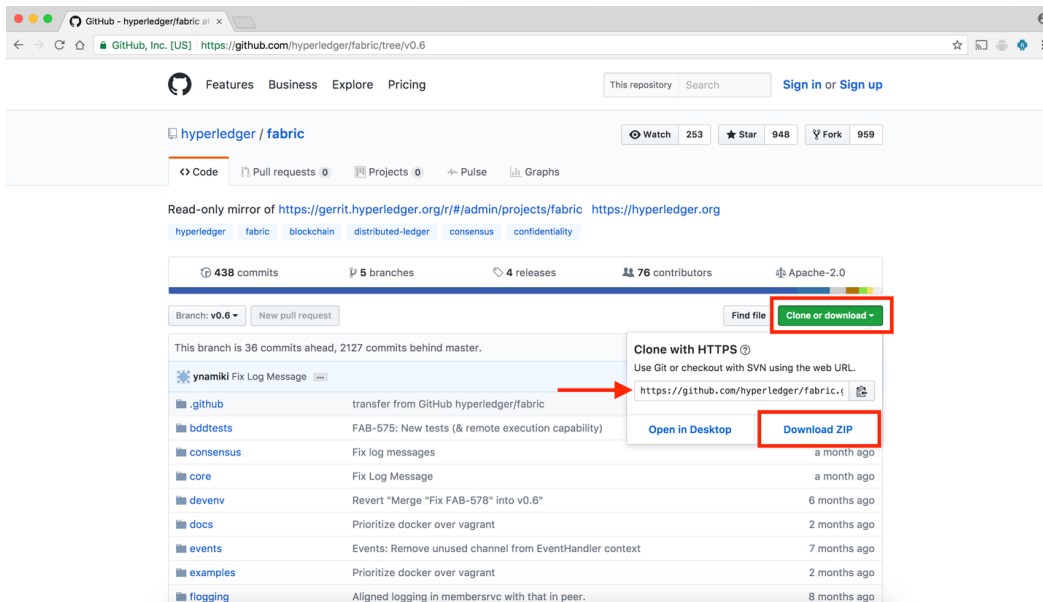
Hyperledger ソースにアクセスするには、2 つの方法があります。

- git を使用しない場合:
このリンク先の [Hyperledger GitHub のミラー](#) にアクセスし、「Clone or download (複製またはダウンロード)」ボタンをクリックして、表示されるポップアップ・ウィンドウで「Download ZIP (ZIP をダウンロード)」をクリックします (図 2 を参照)。fabric-master.zip という名前の ZIP ファイルがコンピューターにダウンロードされるので、この ZIP ファイルの中身を \$GOPATH/src/github.com/hyperledger に抽出します。注: ファイルを抽出する際は、必ずルート・ディレクトリーの名前を fabric-master から fabric に変更してください。
- git を使用する場合:
カレント・ディレクトリーを \$GOPATH/src/github.com/hyperledger に変更します。「Clone with HTTPS (HTTPS を使用して複製)」ボックス内のテキスト・フィールドから URL をコピーし (図 2 を参照)、コピーした URL を使用して以下のコマンドを実行します。

```
git clone https://github.com/hyperledger/fabric.git -b v0.6
```

git コマンドによるターミナル・ウィンドウの出力は、以下のような内容になります。

```
$ git clone https://github.com/hyperledger/fabric.git -b v0.6
Cloning into 'fabric'...
remote: Counting objects: 29272, done.
remote: Compressing objects: 100% (128/128), done.
remote: Total 29272 (delta 55), reused 0 (delta 0), pack-reused 29142
Receiving objects: 100% (29272/29272), 44.95 MiB | 5.67 MiB/s, done.
Resolving deltas: 100% (16671/16671), done.
```



.com/developerWorks/jp/

これで、Java チェーンコードの shim クライアント JAR をビルドできる状態になりました。カレント・ディレクトリーを `$GOPATH/src/github.com/hyperledger/fabric/core/chaincode/shim/java` に変更し、以下の 2 つのコマンドを実行します。

```
gradle -b build.gradle clean
gradle -b build.gradle build
```

Gradle ビルドからの出力は、以下のような内容になっているはずです。

```
$ cd $GOPATH/src/github.com/hyperledger/fabric/core/chaincode/shim/java
$ gradle -b build.gradle clean
Starting a Gradle Daemon (subsequent builds will be faster)
:core:chaincode:shim:java:clean

BUILD SUCCESSFUL

Total time: 5.422 secs
$ gradle -b build.gradle build
:core:chaincode:shim:java:copyProtos UP-TO-DATE
:core:chaincode:shim:java:extractIncludeProto
:core:chaincode:shim:java:extractProto UP-TO-DATE
:core:chaincode:shim:java:generateProto UP-TO-DATE
:core:chaincode:shim:java:compileJava
:core:chaincode:shim:java:processResources
:core:chaincode:shim:java:classes
:core:chaincode:shim:java:jar
:core:chaincode:shim:java:assemble
:core:chaincode:shim:java:extractIncludeTestProto
:core:chaincode:shim:java:extractTestProto UP-TO-DATE
:core:chaincode:shim:java:generateTestProto UP-TO-DATE
:core:chaincode:shim:java:compileTestJava UP-TO-DATE
:core:chaincode:shim:java:processTestResources UP-TO-DATE
:core:chaincode:shim:java:testClasses UP-TO-DATE
:core:chaincode:shim:java:test UP-TO-DATE
:core:chaincode:shim:java:check UP-TO-DATE
:core:chaincode:shim:java:build
:core:chaincode:shim:java:copyToLib
:core:chaincode:shim:java:generatePomFileForMavenJavaPublication
:core:chaincode:shim:java:publishMavenJavaPublicationToMavenLocal
:core:chaincode:shim:java:publishToMavenLocal

BUILD SUCCESSFUL

Total time: 4.521 secs
```

ビルド処理では最後に、shim クライアント JAR がローカル Maven リポジトリーに追加されます。この時点で、チェーンコードをビルドすることが可能になります。将来のある時点でファブリック・ソース・コードを更新する場合、あるいは何らかの理由で shim クライアント JAR を再ビルドしなければならない場合を除けば、今後、Java shim クライアント JAR のビルドを再度実行する必要はありません。

サンプル Java チェーンコードをデプロイして実行する

ローカル・ブロックチェーンを定義して起動し、Java shim クライアント JAR をビルドしてローカル Maven リポジトリーにインストールするところまで完了しました。次は、前にダウンロードした Hyperledger Fabric に同梱されているサンプル Java チェーンコードのうちの 1 つを使用して、Java チェーンコードをビルドする方法、登録する方法、そして Java チェーンコードに対してトランザクションを呼び出す方法を説明します。

このビデオ, チェーンコードをデプロイして実行する動画, をご覧になるには、オンライン記事を参照ください。

このセクションでは、以下の手順について説明します。

1. Gradle を使用してサンプルをビルドする。
2. Gradle ビルドによって作成されたスクリプトを実行して、サンプルを検証ピア・ネットワークに登録する。
3. SoapUI を使用して、サンプルをローカル・ブロックチェーン・ネットワークにデプロイする
4. SoapUI を使用して、サンプル・チェーンコードに対してトランザクションを呼び出す。

サンプルをビルドする

カレント・ディレクトリーを `$GOPATH/src/github.com/hyperledger/fabric/examples/chaincode/java/Example` ディレクトリーに変更します。

コマンド・ラインで、以下のコマンドを使用して Gradle ビルドを起動します。

```
gradle -b build.gradle build
```

上記のコマンドを実行すると、以下のような出力が表示されます。

```
$ cd GOPATH/src/github.com/hyperledger/fabric/examples/chaincode/java/Example
$ gradle -b build.gradle build
Starting a Gradle Daemon (subsequent builds will be faster)
:examples:chaincode:java:Example:compileJava
:examples:chaincode:java:Example:processResources UP-TO-DATE
:examples:chaincode:java:Example:classes
:examples:chaincode:java:Example:jar
:examples:chaincode:java:Example:startScripts
:examples:chaincode:java:Example:distTar
:examples:chaincode:java:Example:distZip
:examples:chaincode:java:Example:assemble
:examples:chaincode:java:Example:compileTestJava UP-TO-DATE
:examples:chaincode:java:Example:processTestResources UP-TO-DATE
:examples:chaincode:java:Example:testClasses UP-TO-DATE
:examples:chaincode:java:Example:test UP-TO-DATE
:examples:chaincode:java:Example:check UP-TO-DATE
:examples:chaincode:java:Example:build
:examples:chaincode:java:Example:copyToLib

BUILD SUCCESSFUL

Total time: 6.935 secs
```

ビルドによって、`build/distributions` ディレクトリー内にスタンドアロンのディストリビューションが TAR ファイルと ZIP ファイルという 2 つの形式で作成されます。このそれぞれのファイルに、チェーンコードを実行するために必要なすべてのものが格納されています。その中には、チェーンコードを駆動する `Example` というスクリプトも含まれています。

これで、`Example` チェーンコードをローカル・ブロックチェーン・ネットワークに登録する準備が整いました。

サンプルを登録する

ローカル・ブロックチェーン・ネットワークが稼働していることを確認してください。稼働していない場合は、「[ブロックチェーン・ネットワークを起動する](#)」セクションに戻って起動してください。

カレント・ディレクトリーが `$GOPATH/src/github.com/hyperledger/fabric/examples/chaincode/java/Example` になっていない場合は、このディレクトリーに変更します。

`build/distributions` ディレクトリー内にある `Example.zip` (または `Example.tar`) の中身を抽出します。

```
$ cd $GOPATH/src/github.com/hyperledger/fabric/examples/chaincode/java/Example
$ cd build/distributions/
$ unzip Example.zip
Archive:  Example.zip
  inflating: Example/lib/chaincode.jar
  inflating: Example/lib/grpc-all-0.13.2.jar
  inflating: Example/lib/commons-cli-1.3.1.jar
  inflating: Example/lib/shim-client-1.0.jar
  inflating: Example/lib/grpc-netty-0.13.2.jar
  inflating: Example/lib/grpc-auth-0.13.2.jar
  inflating: Example/lib/grpc-protobuf-nano-0.13.2.jar
  inflating: Example/lib/grpc-core-0.13.2.jar
  inflating: Example/lib/grpc-protobuf-0.13.2.jar
  inflating: Example/lib/grpc-okhttp-0.13.2.jar
  inflating: Example/lib/grpc-stub-0.13.2.jar
  inflating: Example/lib/protobuf-java-3.0.0.jar
  inflating: Example/lib/netty-tcnative-boringssl-static-1.1.33.Fork21-osx-x86_64.jar
  inflating: Example/lib/netty-codec-http2-4.1.0.CR3.jar
  inflating: Example/lib/google-auth-library-oauth2-http-0.3.0.jar
  inflating: Example/lib/guava-18.0.jar
  inflating: Example/lib/protobuf-javanano-3.0.0-alpha-5.jar
  inflating: Example/lib/jsr305-3.0.0.jar
  inflating: Example/lib/okio-1.6.0.jar
  inflating: Example/lib/okhttp-2.5.0.jar
  inflating: Example/lib/netty-codec-http-4.1.0.CR3.jar
  inflating: Example/lib/netty-handler-4.1.0.CR3.jar
  inflating: Example/lib/google-auth-library-credentials-0.3.0.jar
  inflating: Example/lib/google-http-client-1.19.0.jar
  inflating: Example/lib/google-http-client-jackson2-1.19.0.jar
  inflating: Example/lib/netty-codec-4.1.0.CR3.jar
  inflating: Example/lib/netty-buffer-4.1.0.CR3.jar
  inflating: Example/lib/netty-transport-4.1.0.CR3.jar
  inflating: Example/lib/httpclient-4.0.1.jar
  inflating: Example/lib/jackson-core-2.1.3.jar
  inflating: Example/lib/netty-common-4.1.0.CR3.jar
  inflating: Example/lib/netty-resolver-4.1.0.CR3.jar
  inflating: Example/lib/httpcore-4.0.1.jar
  inflating: Example/lib/commons-logging-1.1.1.jar
  inflating: Example/lib/commons-codec-1.3.jar
  inflating: Example/bin/Example
  inflating: Example/bin/Example.bat
```

「なぜ、これほどまで多くのファイルがあるのか」と不思議に思うかもしれません。このディストリビューションには、チェーンコードをスタンドアロンで (独自のプロセス内で) 実行するために必要なすべてのものが、すべての従属 JAR と一緒に含まれているからです。

サンプル・チェーンコードを登録するには、`build/distributions` フォルダー内にある以下のスクリプトを実行します。

```
./Example/bin/Example
```

このスクリプトが実行するスタンドアロン・プロセスにより、サンプル・チェーンコードがローカル・ブロックチェーン・ネットワークに登録されます。ターミナル・ウィンドウには、以下のような出力が表示されます。

```
$ ./Example/bin/Example
Hello world! starting [Ljava.lang.String;@7ef20235
Feb 22, 2017 10:05:08 AM example.Example main
INFO: starting
Feb 22, 2017 10:05:08 AM org.hyperledger.java.shim.ChaincodeBase newPeerClientConnection
INFO: Inside newPeerClientConnection
Feb 22, 2017 10:05:08 AM io.grpc.internal.TransportSet$1 call
INFO: Created transport io.grpc.netty.NettyClientTransport@3dd7b80b(/127.0.0.1:7051)
for /127.0.0.1:7051
Feb 22, 2017 10:05:14 AM io.grpc.internal.TransportSet$TransportListener transportReady
INFO: Transport io.grpc.netty.NettyClientTransport@3dd7b80b(/127.0.0.1:7051)
for /127.0.0.1:7051 is ready
```

コンソールでローカル・ブロックチェーン・ネットワークを確認すると、以下のような出力行が表示されます。

```
.
.
vp0_1 | 16:05:14.048 [chaincode] HandleChaincodeStream -> DEBU 06d Current context
deadline = 0001-01-01 00:00:00 +0000 UTC, ok = false
vp0_1 | 16:05:14.065 [chaincode] processStream -> DEBU 06e []Received message
REGISTER from shim
vp0_1 | 16:05:14.065 [chaincode] HandleMessage -> DEBU 06f []Handling
ChaincodeMessage of type: REGISTER in state created
vp0_1 | 16:05:14.065 [chaincode] beforeRegisterEvent -> DEBU 070 Received REGISTER in
state created
vp0_1 | 16:05:14.065 [chaincode] registerHandler -> DEBU 071 registered handler
complete for chaincode hello
vp0_1 | 16:05:14.065 [chaincode] beforeRegisterEvent -> DEBU 072 Got REGISTER for
chaincodeID = name:"hello" , sending back REGISTERED
.
.
```

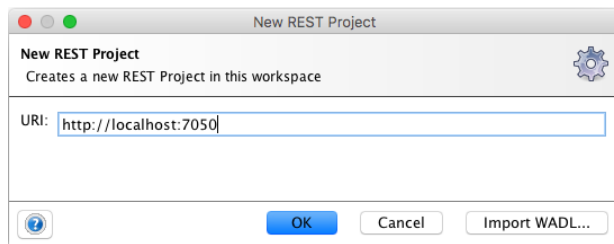
登録のログ出力に含まれる chaincodeID の name の値 (この例では hello。上記の出力の# 8 を参照) をメモしておいてください。この値は、後でファブリックの REST インターフェースを使って Example チェーンコードをデプロイするときに、JSON メッセージ用に必要になります。

上記の出力には、Example チェーンコードが実行されていること、そして実行中の Example チェーンコードがローカル・ブロックチェーンの検証ピア・ネットワークに登録されたことが示されています。これで、チェーンコードをデプロイする作業に移ることができます。

サンプルをデプロイする

Hyperledger Fabric には、このファブリックとやりとりするために使用する REST Web インターフェースが用意されています。ファブリックとの最初のやりとりでは、チェーンコードをデプロイします。ローカル・ブロックチェーン・ネットワークが稼働していることを確認してから、SoapUI を起動し、新しい REST プロジェクトを作成するために「REST」ボタンをクリックします。図 3 に示すようなダイアログ・ボックスが表示されるので、ここに、すべての REST リクエストでベース URL として使用する URL を入力します。

図 3. SoapUI の「New REST Project (新規 REST プロジェクト)」ダイアログ



URL として `http://localhost:7050` と入力し、「OK」をクリックします。ポート 7050 は、ファブリックがデフォルトで使用する REST ポートです。ブロックチェーン・ネットワークはローカル・コンピューター上で稼働していることから、`localhost` をホスト名として使用します。

SoapUI が表示されたら、ローカル・ブロックチェーン・ネットワークと通信できることを確認するために、簡単なスモーク・テストを行うことができます。新しく作成した REST リソースを展開していき、`Request 1` が見つかったら、このリクエストをエディター・ウィンドウ内で開きます。「Method (メソッド)」には「GET」を選択し、「Resource (リソース)」の下に `/chain` と入力します。必ず出力タブで「JSON」をクリックしてから、リクエストを実行してください (矢印アイコンをクリックします)。このリクエストは単純なもので、実行すると、エディター・ウィンドウの右側にある出力タブに、現在のブロックのハッシュが表示されるだけです (図 4 を参照)。

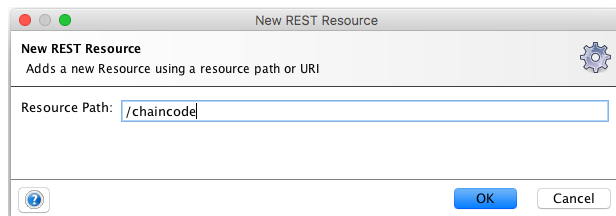


図 4. ブロックチェーンのスモーク・テスト

図 4 のような JSON メッセージが表示されたら (もちろん、皆さんのネットワークでは `currentBlockHash` の値が異なります)、サンプル・チェーンコードはデプロイできる状態になっています。

REST Project 1 (`http://localhost:7050`) の下に表示されているエンドポイントを右クリックして「New Resource (新規リソース)」を選択します。これによって「New REST Resource (新規 REST リソース)」ダイアログ・ボックスが開き、「Resource Path (リソース・パス)」フィールドが表示されます (図 5 を参照)。

図 5. SoapUI の「New REST Resource (新規 REST リソース)」ダイアログ



リソースのパスとして `/chaincode` と入力し、「OK」をクリックすると、SoapUI のプロジェクト・パネルに新しいリソースが表示されます。このリソースに対してリクエストを開き (デフォルトでは、Request 1 という名前)、メソッドを「POST」に変更して、リクエスト・エディター・ウィンドウの左下隅にあるリクエスト域に以下の JSON を貼り付けます。

```
{
  "jsonrpc": "2.0",
  "method": "deploy",
  "params": {
    "type": 1,
    "chaincodeID": {
      "name": "hello"
    },
    "CtorMsg": {
      "args": ["" ]
    }
  },
  "id": 1
}
```

以下の 3 つの点に注意してください。

- 行 3: `method` の値は `deploy` でなければなりません。
- 行 6 ~ 7: JSON メッセージに含まれる `chaincodeID.name` は、前のセクションでサンプル・チェーンコードを登録したときに表示された `chaincodeID` (このサンプル・チェーンコードの場合は `hello`) と一致している必要があります。
- 行 13: `id` の値は、複数のリクエストを調整するために使用されます。このチュートリアルではそれほど気にする必要はありませんが、レスポンス上では常にこの値が返されることに注意してください (次のリスティングを参照)。

このリクエストを送信すると、以下のような JSON 出力が表示されます。

```
{
  "jsonrpc": "2.0",
  "result": {
    "status": "OK",
    "message": "hello"
  },
  "id": 1
}
```

図 6 に、SoapUI に表示された出力例のスクリーンショットを記載します。JSON 出力メッセージは、リクエスト・エディターの右側にある出力タブに表示されます。

Resource Properties

Property	Value
Name	chaincode
Description	
Path	/chaincode

Properties

```
},  
  "ctorMsg": {  
    "args": [""]  
  }  
},  
  "id": 1  
}
```

...He...Attac...Repres...JMS...JMS P...Headers (6)Attachments (0)SSL InfoRepresentations (1)Schema (conflicts)JMS (0)

response time: 4ms (67 bytes)

1 : 55

SoapUI loghttp logjetty logerror logwsrm logmemory log

ターミナル・ウィンドウに表示されるネットワーク・ログ出力には、以下のような行が含まれているはずです。

```

      .
      .
request...  vp0_1      | 20:48:39.482 [rest] ProcessChaincode -> INFO 0c4 REST processing chaincode
chaincode...  vp0_1      | 20:48:39.482 [rest] processChaincodeDeploy -> INFO 0c5 REST deploying
(hello)      vp0_1      | 20:48:39.483 [devops] Deploy -> DEBU 0c6 Creating deployment transaction
to validator  vp0_1      | 20:48:39.483 [devops] Deploy -> DEBU 0c7 Sending deploy transaction (hello)
transaction  vp0_1      | 20:48:39.483 [peer] sendTransactionsToLocalEngine -> DEBU 0c8 Marshalling
CHAINCODE_DEPLOY to send to local engine
CHAIN_TRANSACTION with timestamp seconds:1487796519 nanos:483661510 to local engine
CHAIN_TRANSACTION  vp0_1      | 20:48:39.483 [consensus/noops] RecvMsg -> DEBU 0ca Handling Message of type:
CHAIN_TRANSACTION  vp0_1      | 20:48:39.483 [consensus/noops] broadcastConsensusMsg -> DEBU 0cb Broadcasting
CONSENSUS      vp0_1      | 20:48:39.483 [peer] Broadcast -> DEBU 0cc Broadcast took 1.135s
uuid: hello    vp0_1      | 20:48:39.483 [consensus/noops] RecvMsg -> DEBU 0cd Sending to channel tx
chainCode: hello  vp0_1      | 20:48:39.483 [rest] processChaincodeDeploy -> INFO 0ce Successfully deployed
chaincode: {"jsonrpc":"2.0","result":{"status":"OK","message":"hello"},"id":1}
      .
      .

```

行 3 ~ 4 の出力は、ネットワークがデプロイ・メッセージを受信し、ファブリックがチェーンコードをデプロイ中であることを示しています。行 13 ~ 14 の出力は、チェーンコードが正常にデプロイされたことを示しています。

チェーンコードを実行中のターミナル・ウィンドウ内で、以下のような出力が表示されていることを確認してください。

```

$ ./build/distributions/Example/bin/Example
Hello world! starting [Ljava.lang.String;@7ef20235
Feb 22, 2017 2:44:43 PM example.Example main
INFO: starting
Feb 22, 2017 2:44:43 PM org.hyperledger.java.shim.ChaincodeBase newPeerClientConnection
INFO: Inside newPeerClientConnection
Feb 22, 2017 2:44:43 PM io.grpc.internal.TransportSet$1 call
INFO: Created transport io.grpc.netty.NettyClientTransport@46adccd3(/127.0.0.1:7051)
for /127.0.0.1:7051
Feb 22, 2017 2:44:48 PM io.grpc.internal.TransportSet$TransportListener transportReady
INFO: Transport io.grpc.netty.NettyClientTransport@46adccd3(/127.0.0.1:7051)
for /127.0.0.1:7051 is ready
Feb 22, 2017 2:48:40 PM example.Example run
INFO: In run, function:
Feb 22, 2017 2:48:40 PM example.Example run

```

コンテキストがわかるように出力のすべてを記載していますが、デプロイ・メッセージをブロックチェーン・ネットワークに送信した時点で表示されるのは、行 11 ~ 13 のような行です。

サンプルに対してトランザクションを呼び出す

最後に、hello メソッドを呼び出して、チェーンコードを実行中のターミナル・ウィンドウのログ・メッセージに表示される出力を確認します。

SoapUI 内で、チェーンコード・リソースに含まれる Method 1 を右クリックし、「Clone Method (メソッドの複製)」を選択します。複製するメソッドに「Invoke」という名前を付けて、「OK」をクリックします。新しく複製した Invoke メソッドに含まれる Request 1 を開き、そこに以下の JSON リクエストを貼り付けます。

```
{
  "jsonrpc": "2.0",
  "method": "invoke",
  "params": {
    "type": 1,
    "chaincodeID": {
      "name": "hello"
    },
    "CtorMsg": {
      "args": ["hello"]
    }
  },
  "id": 2
}
```

このリクエストを実行すると、以下の JSON レスポンスが表示されます。

```
{
  "jsonrpc": "2.0",
  "result": {
    "status": "OK",
    "message": "1c1811d0-a958-4c58-ab1d-e1df550c18a3"
  },
  "id": 2
}
```

図 7 に、SoapUI に表示された出力例のスクリーンショットを記載します。

Request Properties

Property	Value
Name	Request 1
Description	
Encoding	

Properties

Raw

```
method: invoke
params: {
  "type": 1,
  "chaincodeID": {
    "name": "hello"
  },
  "ctorMsg": {
    "args": ["hello"]
  }
},
"id": 2
}
```

1/

SoapUI loghttp logjetty logerror logwsrm logmemory log

ネットワーク・ログ出力には、以下のような行が含まれているはずです。

```

      .
      .
request...    vp0_1      | 21:44:35.143 [rest] ProcessChaincode -> INFO 555 REST processing chaincode
chaincode...  vp0_1      | 21:44:35.143 [rest] processChaincodeInvokeOrQuery -> INFO 556 REST invoke
              vp0_1      | 21:44:35.143 [devops] invokeOrQuery -> INFO 557 Transaction ID: 1c1811d0-
a958-4c58-ab1d-e1df550c18a3
              vp0_1      | 21:44:35.143 [devops] createExecTx -> DEBU 558 Creating invocation
transaction (1c1811d0-a958-4c58-ab1d-e1df550c18a3)
              vp0_1      | 21:44:35.143 [devops] invokeOrQuery -> DEBU 559 Sending invocation
transaction (1c1811d0-a958-4c58-ab1d-e1df550c18a3) to validator
              vp0_1      | 21:44:35.143 [peer] sendTransactionsToLocalEngine -> DEBU 55a Marshalling
transaction CHAINCODE_INVOKE to send to local engine
              vp0_1      | 21:44:35.143 [peer] sendTransactionsToLocalEngine -> DEBU 55b Sending message
CHAIN_TRANSACTION with timestamp seconds:1487799875 nanos:143438691 to local engine
              vp0_1      | 21:44:35.143 [consensus/noops] RecvMsg -> DEBU 55c Handling Message of type:
CHAIN_TRANSACTION
              vp0_1      | 21:44:35.143 [consensus/noops] broadcastConsensusMsg -> DEBU 55d Broadcasting
CONSENSUS
              vp0_1      | 21:44:35.143 [peer] Broadcast -> DEBU 55e Broadcast took 1.249s
              vp0_1      | 21:44:35.143 [consensus/noops] RecvMsg -> DEBU 55f Sending to channel tx
uuid: 1c1811d0-a958-4c58-ab1d-e1df550c18a3
              vp0_1      | 21:44:35.143 [rest] processChaincodeInvokeOrQuery -> INFO 560 Successfully
submitted invoke transaction with txid (1c1811d0-a958-4c58-ab1d-e1df550c18a3)
              vp0_1      | 21:44:35.143 [rest] ProcessChaincode -> INFO 561 REST successfully
submitted invoke transaction: {"jsonrpc":"2.0","result":{"status":"OK","message":"1c1811d0-a958-4c58-ab1d-
e1df550c18a3"},"id":2}
      .
      .

```

チェーンコード・ログ出力は、以下のような内容になっています。

```

$ ./build/distributions/Example/bin/Example
Hello world! starting [Ljava.lang.String;@7ef20235
Feb 22, 2017 3:26:57 PM example.Example main
INFO: starting
Feb 22, 2017 3:26:57 PM org.hyperledger.java.shim.ChaincodeBase newPeerClientConnection
INFO: Inside newPeerClientConnection
Feb 22, 2017 3:26:57 PM io.grpc.internal.TransportSet$1 call
INFO: Created transport io.grpc.netty.NettyClientTransport@765e4953(/127.0.0.1:7051)
for /127.0.0.1:7051
Feb 22, 2017 3:27:02 PM io.grpc.internal.TransportSet$TransportListener transportReady
INFO: Transport io.grpc.netty.NettyClientTransport@765e4953(/127.0.0.1:7051)
for /127.0.0.1:7051 is ready
Feb 22, 2017 3:27:24 PM example.Example run
INFO: In run, function:
Feb 22, 2017 3:27:24 PM example.Example run
SEVERE: No matching case for function:
Feb 22, 2017 3:30:55 PM example.Example run
INFO: In run, function:hello
hello invoked

```

ここでもチェーンコードの出力をすべて記載していますが、どこで hello 関数が呼び出されたのかはわかるはずです (行 16)。

Java チェーンコードをビルドして、ローカル・ブロックチェーン・ネットワークにデプロイし、実行する方法は以上のとおりです。次のセクションでは、Eclipse IDE を使用してチェーンコー

ド・プログラムを (ほぼ) ゼロから作成し、そのチェーンコード・プログラムを Gradle を使ってビルドしてから、SoapUI を使って実行する方法を説明します。

初めての Java チェーンコード・プログラムを作成する

前のセクションで、チェーンコードをビルド、実行、デプロイして呼び出す方法を理解してもらえたはずですが、実際に作成した Java コードはまだ 1 行もありません。

このセクションでは、Eclipse IDE と、Gradle を Eclipse と連動させるためのプラグイン、そして ChaincodeTutorial という名前のスケルトン Java チェーンコード・プロジェクトを使用して、皆さんが初の Java チェーンコード・プログラムを作成できるよう案内します。このセクションで使用するスケルトン・コードは、私がこのチュートリアルのために作成した GitHub リポジトリから入手できます。そのスケルトン・コードを Eclipse にインポートし、チェーンコードを要件に応じたスマート・コントラクトとして機能させるためのコードを追加します。そして最後に、Eclipse IDE 内で Gradle を使用してコードをビルドします。

このセクションでは、以下の手順について説明します。

1. Eclipse の Gradle 対応 Buildship プラグインをインストールする。
2. GitHub から ChaincodeTutorial プロジェクトを複製する。
3. プロジェクトを Eclipse にインポートする。
4. チェーンコード・スケルトン・プロジェクトの内容を調べる。
5. Java チェーンコードを作成する。
6. Java チェーンコードをビルドする。

このセクションを完了すると、ローカル・ブロックチェーン・ネットワーク上で実行できるチェーンコードが完成します。

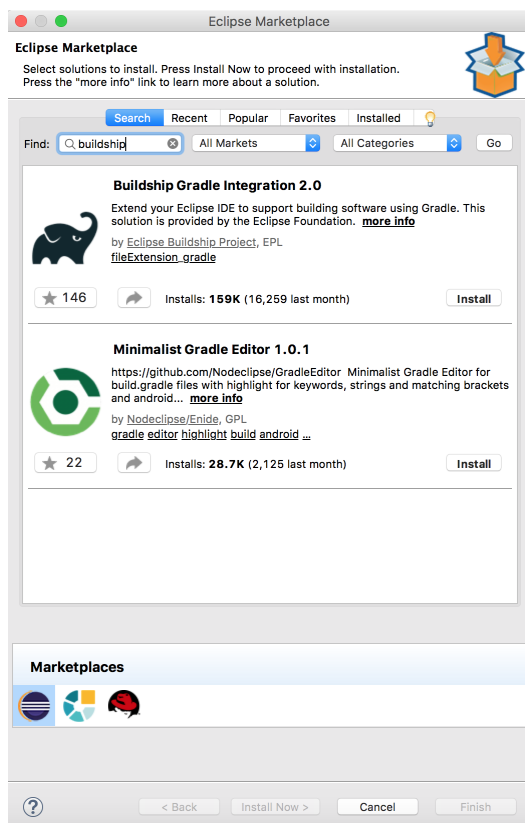
Eclipse の Gradle 対応 Buildship プラグインをインストールする

任意の IDE を使用できますが、このチュートリアルでは Eclipse を使用する場合の手順を説明します。注: Buildship Gradle プラグインは、Gradle を Eclipse に統合するためのものですが、それでも Gradle がコンピューター上にインストールされた状態にしておく必要があります。

チュートリアルのこれまでの手順に従っていれば、Gradle はコンピューター上にインストールされた状態になっています。インストールされていなければ、この時点で「ビルド・ソフトウェアをインストールする」セクションの手順に戻って Gradle をコンピューターにインストールしてください。

Eclipse の Buildship Gradle プラグインは、Gradle システムを Eclipse と連動させるようにするためのプラグインです。このプラグインをインストールするには、Eclipse 内で「Help (ヘルプ)」>「Eclipse Marketplace... (Eclipse マーケットプレイス...)」を表示します。「Eclipse Marketplace (Eclipse マーケットプレイス)」ダイアログの「Find (検索)」テキスト・フィールドに buildship と入力し、「Go (実行)」ボタンをクリックします。図 8 のようなページに、最初の検索結果として Buildship Gradle Integration 2.0 プラグインが表示されるはずです。

図 8. 「Eclipse Marketplace (Eclipse マーケットプレイス)」 : Buildship Gradle プラグイン



Buildship Gradle Integration の検索結果の右下にある「Install (インストール)」ボタンをクリックし、表示される画面の指示に従います。「Finish (完了)」をクリックすると、Eclipse の Buildship Gradle プラグインがインストールされて、Eclipse を再起動するよう促されます。

Eclipse が再起動した時点で、Gradle は Eclipse IDE に完全に統合された状態になります。これで、GitHub から ChaincodeTutorial リポジトリを複製する手順に進むことができます。

GitHub から ChaincodeTutorial プロジェクトを複製する

Eclipse IDE が Gradle と連動するように構成された後は、GitHub から ChaincodeTutorial プロジェクトを複製して、このプロジェクトのコードを Eclipse にインポートします。コマンド・プロンプトまたはターミナル・ウィンドウを開き、カレント・ディレクトリーを `$GOPATH` に変更してから、以下のコマンドを実行します。

```
git clone https://github.com/makotogo/ChaincodeTutorial.git
```

以下のようなコマンド出力が表示されるはずです。

```
$ export GOPATH=/Users/sperry/home/mychaincode
$ cd $GOPATH
$ git clone https://github.com/makotogo/ChaincodeTutorial.git
Cloning into 'ChaincodeTutorial'...
remote: Counting objects: 133, done.
remote: Compressing objects: 100% (90/90), done.
remote: Total 133 (delta 16), reused 118 (delta 1), pack-reused 0
Receiving objects: 100% (133/133), 9.39 MiB | 1.95 MiB/s, done.
Resolving deltas: 100% (16/16), done.
$ cd ChaincodeTutorial
$ pwd
/Users/sperry/home/mychaincode/ChaincodeTutorial
```

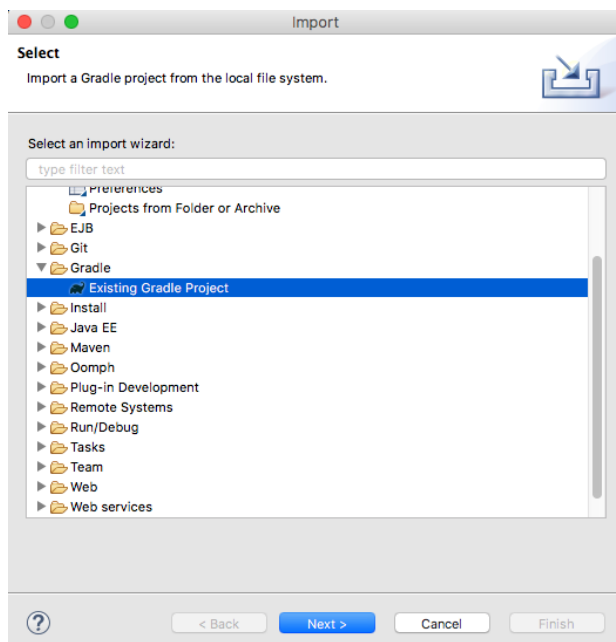
このコマンドによって、GitHub から \$GOPATH に Blockchain ChaincodeTutorial リポジトリが複製されます。このリポジトリ内に含まれるスケルトン Java チェーンコード・プロジェクトをビルドして、ローカル・ブロックチェーン・ネットワーク内で実行し、いろいろと実験することができます。

ただし、いずれの作業にしても、コードを Eclipse にインポートしてからでなければ実行できません。

プロジェクトを Eclipse にインポートする

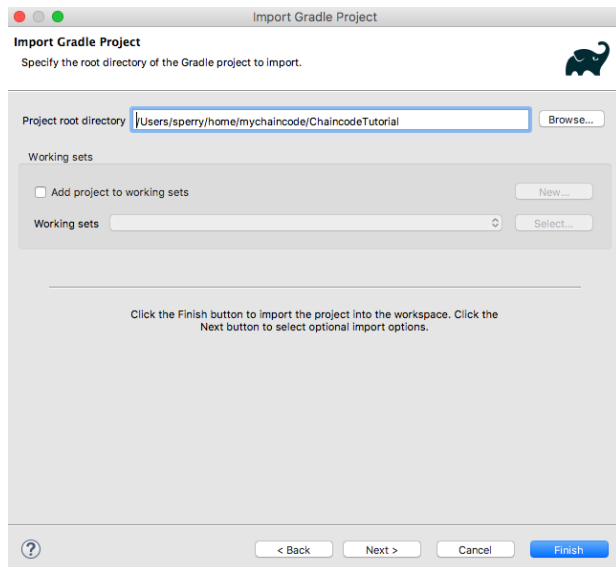
Eclipse メインメニューから、「File (ファイル)」 > 「Import... (インポート...)」 > 「Gradle」 > 「Existing Gradle project (既存の Gradle プロジェクト)」の順に選択して、ウィザード・ダイアログ・ボックスを開きます (図 9 を参照)。

図 9. Eclipse の「Import (インポート)」ウィザード: Gradle プロジェクト



「Next (次へ)」をクリックします。ウィザード内で次に表示されるダイアボックス (図 10 を参照) で、`$GOPATH/ChaincodeTutorial` を参照し、「Finish (完了)」をクリックしてプロジェクトをインポートします。

図 10. Eclipse の「Import (インポート)」ウィザード: Gradle プロジェクト (プロジェクトのルート・ディレクトリー)



プロジェクトのインポートが完了したら、必ず「Java Perspective (Java パースペクティブ)」を選択し、インポートした ChaincodeTutorial プロジェクトが「Project Explorer (プロジェクト・エクスプローラー)」ビューに表示されていることを確認してください。

これで、コードが Eclipse ワークスペースにインポートされたので、チェーンコードを作成する作業を開始できます。

チェーンコード・スケルトン・プロジェクトの内容を調べる

Java コードを作成する前に、このセクションでは、チェーンコードがどのように機能するよう想定されているのかを皆さんが理解できるよう、チェーンコード・プロジェクトの内容を探ります。

開発者である私たちはコードを作成するのが大好きなので、皆さんから Java コードを作成する機会を取り上げる気はありませんが、プロジェクトのセットアップは複雑になることがあります。その複雑さのせいで、このチュートリアルの要点を台無しにすることはしたくないので、必要となるコードの大部分をあらかじめ用意しておきました。

詳細に踏み込む前に、`com.makotojava.learn.blockchain.chaincode` 内にあ
る、`AbstractChaincode` という名前の基底クラスについて簡単に説明します。リスト 1 に記載す
る `AbstractChaincode` を見てください。

リスト 1. `AbstractChaincode` クラス

```
package com.makotojava.learn.blockchain.chaincode;

import java.util.Arrays;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.hyperledger.java.shim.ChaincodeBase;
import org.hyperledger.java.shim.ChaincodeStub;

public abstract class AbstractChaincode extends ChaincodeBase {

    private static final Log log = LogFactory.getLog(AbstractChaincode.class);

    public static final String FUNCTION_INIT = "init";
    public static final String FUNCTION_QUERY = "query";

    protected abstract String handleInit(ChaincodeStub stub, String[] args);
    protected abstract String handleQuery(ChaincodeStub stub, String[] args);
    protected abstract String handleOther(ChaincodeStub stub, String function, String[] args);

    @Override
    public String run(ChaincodeStub stub, String function, String[] args) {
        String ret;
        log.info("Greetings from run(): function -> " + function + " | args -> " +
Arrays.toString(args));
        switch (function) {
            case FUNCTION_INIT:
                ret = handleInit(stub, args);
                break;
            case FUNCTION_QUERY:
                ret = handleQuery(stub, args);
            default:
                ret = handleOther(stub, function, args);
                break;
        }
        return ret;
    }

    @Override
    public String query(ChaincodeStub stub, String function, String[] args) {
        return handleQuery(stub, args);
    }

}
```

まず始めに指摘したい点として、`AbstractChaincode` は、ファブリック shim クライアントに含ま
れる `ChaincodeBase` のサブクラスです (行 7、10)。

行 17 ~ 19 の各行には、`ChaincodeLog` クラス (`AbstractChaincode` のサブクラス) に実装する必要
がある、初期化を処理するメソッド、レジラーのクエリーを処理するメソッド、log 関数を処理す
るメソッドが示されています。

行 22 ~ 36 に示されているのは、(チェーンコード shim クライアントに含まれる) `ChaincodeBase`
クラスの `run()` メソッドです。このメソッドの内容を見ると、どの関数が呼び出されると、どの

ハンドラーにその呼び出しが委任されるのかがわかります。このクラスは、`init` と `query` 以外のどの関数 (例えば、`log` 関数) でも `handleOther()` (これも実装する必要があります) で処理できるという点で、拡張性に優れています。

次は、`com.makotojava.learn.blockchain.chaincode` パッケージに含まれる `ChaincodeLog` クラスを開いてください。

皆さんが具体化できるように、スケルトンだけを用意しておきました。つまり、コンパイルするのに十分なだけのコードなので、残りの部分は皆さんが作成する必要があります。JUnit テストを実行すると、テストが失敗して (まだ実装が作成されていないため)、なぜ失敗するのかを確認できます。別の言い方をすれば、コードを適切に実装するには、JUnit テストを案内役として利用できるということです。

自力で実装するには少々手に負えないかもしれないと感じたとしても、心配しないでください。お手上げ状態になった場合に備え (あるいは、実装のリファレンス・ガイドとして)、`com.makotojava.learn.blockchain.chaincode.solution` にソリューションを用意しておきました。

Java チェーンコードを作成する

まず、`ChaincodeLog` にチェーンコード・メソッドを実装するために知っておかなければならない予備知識について少々説明しておきます。`ChaincodeStub` クラスは、Java チェーンコードが Hyperledger Fabric フレームワークとやりとりするための手段です。ブロックチェーン・テクノロジーの透明性という側面の中核となるのはレジャーであることを思い出してください。スマート・コントラクト (責任説明) を有効にするのはレジャーの状態であり、チェーンコードは `ChaincodeStub` を使用してレジャーの状態にアクセスします。レジャーの状態にアクセスすることによって、スマート・コントラクト (別名チェーンコード) を実装することも可能になります。

`ChaincodeStub` には、レジャーの状態を保管、取得したり、レジャーの現在の状態から項目を削除したりするために使用できるメソッドはいくつもありますが、このチュートリアルで説明するメソッドは、レジャーの状態を保管、取得するメソッドの 2 つに絞ります。

`putState(String key, String value)` — 指定された状態値を、指定されたキーに従ってマッピングされたレジャーに保管します。

`getState()` — 指定されたキーに関連付けられた状態値を取得し、その値を文字列として返します。

このチュートリアルに従ってコードを作成する際は、レジャーに状態値を保管するには常に `putState()` 関数を、レジャーから状態値を取得するには常に `getState()` 関数を使用します。`ChaincodeLog` クラスはスマート・コントラクトを実装するためにレジャーの値を保管、取得するだけなので、この 2 つのメソッドを実装する方法を理解していれば、それで十分です。より複雑なチェーンコードになってくると、`ChaincodeStub` の他のメソッドを利用することになりますが、それらのメソッドについても、このチュートリアルの範囲外です。

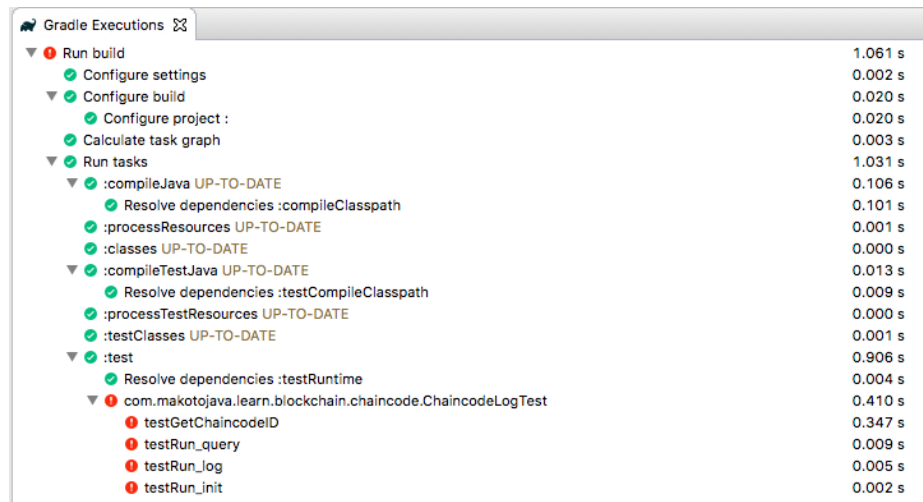
私はテスト駆動型開発 (TDD) の大ファンなので、TDD の方法に従って、最初に単体テストを作成しました。これらの単体テストを実行して、テストが失敗することを確認してください。その

後、単体テストが合格するようになるまで、仕様に従ったコードを作成します。単体テストの役割は、期待される動作を確保することです。したがって、単体テストを調べることで、必要なメソッドを実装するための情報を十分に入手できるはずですが、

(TDD または JUnit を使用するのが初めての場合に備え) 各メソッドの先頭に参考になりそうな javadoc コメントも書き込んでおきました。チュートリアルはこのセクションを完了すると、スケルトン `ChaincodeLog` 内の JUnit テストのコードと javadoc コマンドとの間が、チェーンコードを実装するために必要なすべてのコードで埋まることになります。

(Java パースペクティブの)「Project Explorer (プロジェクト・エクスプローラー)」で `ChaincodeLogTest` クラスにナビゲートし、このクラスをクリックして「Run As (実行)」>「Gradle Test (Gradle テスト)」を選択します。テストが実行されると、図 11 のように、実行された Gradle タスクのすべてがツリー形式で表示されます。正常に完了したタスクは、その横にチェック・マークが付けられます。

図 11. Eclipse: 「Gradle Executions (Gradle 実行)」 ビュー



Task	Execution Time
Run build	1.061 s
Configure settings	0.002 s
Configure build	0.020 s
Configure project :	0.020 s
Calculate task graph	0.003 s
Run tasks	1.031 s
compileJava UP-TO-DATE	0.106 s
Resolve dependencies :compileClasspath	0.101 s
:processResources UP-TO-DATE	0.001 s
:classes UP-TO-DATE	0.000 s
compileTestJava UP-TO-DATE	0.013 s
Resolve dependencies :testCompileClasspath	0.009 s
:processTestResources UP-TO-DATE	0.000 s
:testClasses UP-TO-DATE	0.001 s
test	0.906 s
Resolve dependencies :testRuntime	0.004 s
com.makotojava.learn.blockchain.chaincode.ChaincodeLogTest	0.410 s
testGetChaincodeID	0.347 s
testRun_query	0.009 s
testRun_log	0.005 s
testRun_init	0.002 s

「Gradle Executions (Gradle 実行)」ビューに示される感嘆符アイコンは、失敗した単体テストに対応する Gradle タスクを示します (予想どおり、4 つすべての単体テストが失敗しました)。

私はこの JUnit テスト・ケースを、それぞれのテスト・メソッドがこのチュートリアルの一環として実装する必要がある `ChaincodeLog` 内の 1 つのメソッドに対応するように作成しました。

getChaincodeID() を実装する

最初に、`getChaincodeID()` を実装する必要があります。このメソッドの要件は、チェーンコードの一意の ID を返すことです。`ChaincodeLog` クラスの先頭に `CHAINCODE_ID` という名前の定数を定義しておいたので、この定数を使用してください。定数の値はお好きなように変更して構いませんが、`getChaincodeID()` から返されるチェーンコード ID を変更する場合は、ネットワーク内で一意の ID にならなければなりません。また、JSON メッセージの `ChaincodeID.name` 属性も忘れずに変更してください。

```
/**
 * Returns the unique chaincode ID for this chaincode program.
 */
@Override
public String getChaincodeID() {
    return null; // ADD YOUR CODE HERE
}
```

演習: `getChaincodeID()` メソッドを完成させます。リファレンスが必要な場合は、`com.makotojava.learn.blockchain.chaincode.solution` パッケージを調べてください。

handleInit() を実装する

次は、`handleInit()` メソッドを実装します。このメソッドの要件は、チェーンコード・プログラムの初期化を処理することです。この例の場合、(呼び出し側が指定した) メッセージをレジャーに追加し、呼び出しが成功した場合はそのメッセージを呼び出し側に返すことを意味します。

```
/**
 * Handles initializing this chaincode program.
 *
 * Caller expects this method to:
 *
 * 1. Use args[0] as the key for logging.
 * 2. Use args[1] as the log message.
 * 3. Return the logged message.
 */
@Override
protected String handleInit(ChaincodeStub stub, String[] args) {
    return null; // ADD YOUR CODE HERE
}
```

演習: `handleInit()` メソッドを完成させます。リファレンスが必要な場合は、`com.makotojava.learn.blockchain.chaincode.solution` パッケージを調べてください。

handleQuery() を実装する。

次に実装するのは、`handleQuery()` メソッドです。このメソッドの要件は、レジャーのクエリーを実行することです。具体的には、指定されたキー (複数可) を取り、そのキーと一致する値 (複数可) をレジャー内で検索して、呼び出し側に値を返します。複数のキーが指定された場合、メソッドが返すそれぞれの値をコンマで区切る必要があります。

```

/**
 * Handles querying the ledger.
 *
 * Caller expects this method to:
 *
 * 1. Use args[0] as the key for ledger query.
 * 2. Return the ledger value matching the specified key
 *    (which should be the message that was logged using that key).
 */
@Override
protected String handleQuery(ChaincodeStub stub, String[] args) {
    return null; // ADD YOUR CODE HERE
}

```

クエリー呼び出しの結果をコンソールの出力内で確認できるよう、必ず、結果を出力するようにコードを作成してください (結果を出力する方法の一例としては、私が用意しておいたソリューションを見てください)。

演習: `handleQuery()` メソッドを完成させます。リファレンスが必要な場合は、`com.makotojava.learn.blockchain.chaincode.solution` パッケージを調べてください。

handleOther() を実装する

最後に、`handleOther()` メソッドを実装する必要があります。このメソッドの要件は、他のメッセージを処理することです (他のメッセージと言うと、かなり広範囲にわたりますが、それがクラスに拡張性をもたらす理由です)。このメソッドには、`log` 関数を実装します。この関数の要件は、呼び出し側が指定したメッセージをレジャーに追加し、呼び出しが成功した場合はそのメッセージを呼び出し側に返すことです。この処理は、`init` 関数での処理内容と似ているので、実装にはその内容を利用するという方法も考えられます。

```

/**
 * Handles other methods applied to the ledger.
 * Currently, that functionality is limited to these functions:
 * - log
 *
 * Caller expects this method to:
 * Use args[0] as the key for logging.
 * Use args[1] as the log message.
 * Return the logged message.
 */
@Override
protected String handleOther(ChaincodeStub stub, String function, String[] args) {
    // TODO Auto-generated method stub
    return null; // ADD YOUR CODE HERE
}

```

演習: `handleOther()` メソッドを完成させます。リファレンスが必要な場合は、`com.makotojava.learn.blockchain.chaincode.solution` パッケージを調べてください。

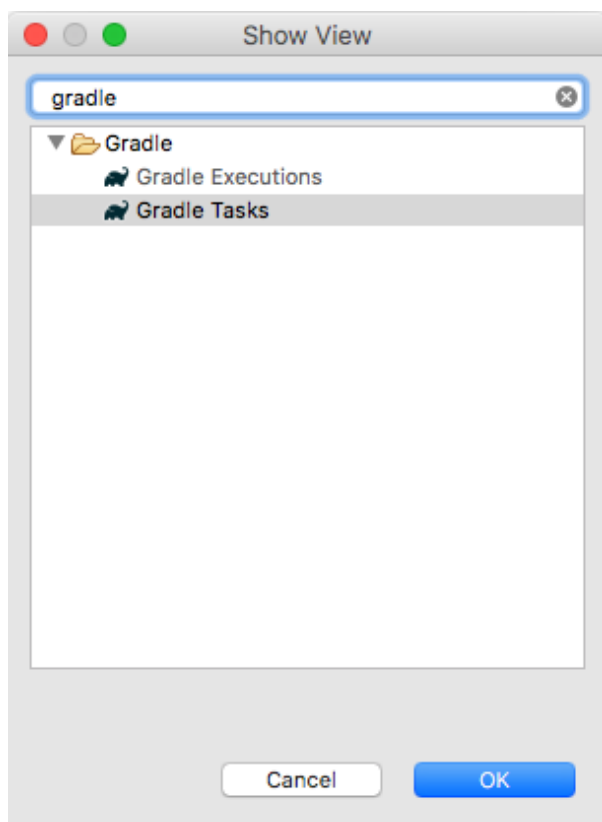
以上の演習のそれぞれで作成したコードが、このセクション (およびコードのコメント) で設定した要件を満たすとしたら、JUnit テストはすべて合格するはずです。したがって、チェーンコードをローカル・ブロックチェーン・ネットワークにデプロイして実行すると、チェーンコードが問題なく機能することになります。

皆さんがお手上げ状態になった場合に備え、ソリューションが用意されていることを忘れないでください (ただし、ソリューションをカンニングする前に、自力でメソッドの実装を試してみてください)。

Java チェーンコードをビルドする

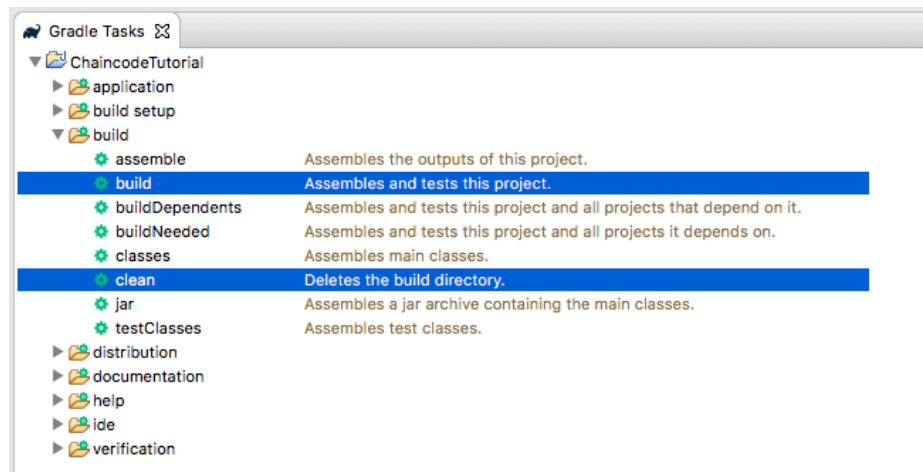
Java チェーンコードを作成して、すべての JUnit テストが合格するようになったので、次は、Eclipse と、Eclipse の Buildship Gradle プラグインを使用してチェーンコードをビルドする作業に取り掛かります。まず、「Gradle Tasks (Gradle タスク)」ビューを表示します。それには、Eclipse メインメニューで「Window (ウィンドウ)」 > 「Show View (ビューの表示)」 > 「Other... (その他...)」の順に選択します。次に、gradle を検索して、それによって表示される「Gradle Tasks (Gradle タスク)」を選択し、「OK」をクリックします (図 12 を参照)。

図 12. Eclipse: 「Show View (ビューの表示)」: 「Gradle Tasks (Gradle タスク)」



「Gradle Tasks (Gradle タスク)」ビューが開いたら、「ChaincodeTutorial」 > 「build」ノードを展開し、「build」と「clean」を選択します (図 13 を参照)。

図 13. Eclipse: 「Gradle Tasks (Gradle タスク)」 ビュー



「build」と「clean」を右クリックして、「Run Gradle Tasks (Gradle タスクの実行)」を選択します (Gradle はこの 2 つのタスクの正しい実行順序を把握します)。「Gradle Executions (Gradle 実行)」ビューに、クリーン・ビルドが表示されます (図 14 を参照)。各項目の横にはいずれもチェック・マークが付いているはずです。

✓ Resolve dependencies :testRuntime	0.003 s
▼ ✓ com.makotojava.learn.blockchain.chaincode.ChaincodeLogTest	0.427 s
✓ testGetChaincodeID	0.357 s
✓ testRun_query	0.018 s
✓ testRun_log	0.008 s
✓ testRun_init	0.002 s
✓ :check	0.001 s
✓ :build	0.000 s
✓ :copyToLib	0.093 s
✓ :clean	0.042 s

®

ビルドが完了すると、\$GOPATH/ChaincodeTutorial ディレクトリー (GitHub のコードを複製した場所) の直下に、build/distributions という名前のサブディレクトリーが作成されているはずです。このサブディレクトリーに、チェーンコードが格納されています (このチュートリアルの前半で扱った hello サンプルから、見覚えがあるはずです)。

Java チェーンコードのビルドが完了しました。次は、このチェーンコードをローカル・ブロックチェーンにデプロイして実行し、このチェーンコードに対してトランザクションを呼び出す作業に移ります。

Java チェーンコードをデプロイして実行する

このセクションでは、チェーンコードを起動して登録する方法、チェーンコードをデプロイする方法、そしてこのチュートリアル hello サンプルで行ったように Hyperledger Fabric REST インターフェースを使用して、チェーンコードに対してトランザクションを呼び出す方法を説明します。ローカル・ブロックチェーンが稼働していることを確認します (そのための方法を再確認するには、「[ブロックチェーン・ネットワークを起動する](#)」セクションを参照)。

このセクションでは、以下の手順について説明します。

1. Java チェーンコードを登録する。
2. Java チェーンコードをデプロイする。
3. Java チェーンコードに対してトランザクションを呼び出す。

Java チェーンコードを登録する

build/distributions/ChaincodeTutorial.zip ファイルを圧縮解除して、そこから抽出されたチェーンコード・スクリプトを実行します。その手順は、このチュートリアルで hello サンプルを実行した際の手順とまったく同じです (「[サンプルを登録する](#)」セクションを参照してください)。

ChaincodeTutorial スクリプトを実行すると、以下のような出力が表示されるはずです。

```
$ ./ChaincodeTutorial/bin/ChaincodeTutorial
Feb 28, 2017 4:18:16 PM org.hyperledger.java.shim.ChaincodeBase newPeerClientConnection
INFO: Inside newPeerClientConnection
Feb 28, 2017 4:18:16 PM io.grpc.internal.TransportSet$1 call
INFO: Created transport io.grpc.netty.NettyClientTransport@10bf86d3(/127.0.0.1:7051)
for /127.0.0.1:7051
Feb 28, 2017 4:18:21 PM io.grpc.internal.TransportSet$TransportListener transportReady
INFO: Transport io.grpc.netty.NettyClientTransport@10bf86d3(/127.0.0.1:7051)
for /127.0.0.1:7051 is ready
```

Java チェーンコードがローカル・ブロックチェーン・ネットワークに登録されました。次は、チェーンコードをデプロイしてテストしましょう。

Java チェーンコードをデプロイする

hello サンプル・チェーンコードで作業したときと同じように、Java チェーンコードをデプロイして、そのチェーンコードに対してトランザクションを呼び出すには、ファブリックの REST インターフェースを使用します。

SoapUI を開きます。新しい REST プロジェクトを作成することに慣れている場合は、新規 REST プロジェクトと REST リクエストのすべてを自分で作成するのも構いません。そうでない場合

は、REST プロジェクトをインポートできるように、前に複製した GitHub プロジェクトに SoapUI REST プロジェクトを含めておきました。SoapUI プロジェクトは、`$GOPATH/ChaincodeTutorial` ディレクトリー内にあります。

チェーンコードをデプロイするには、`ChaincodeLog Deploy` リクエストにナビゲートして、このリクエストを送信します。



GitHub から複製した SoapUI プロジェクトを使用しているのではない場合 (または、別の HTTP クライアントを使用している場合)、送信する JSON リクエストは以下のような内容になっているはずです。

```
{
  "jsonrpc": "2.0",
  "method": "deploy",
  "params": {
    "type": 4,
    "chaincodeID": {
      "name": "ChaincodeLogSmartContract"
    },
    "ctorMsg": {
      "args": ["init", "KEY-1", "Chaincode Initialized"]
    }
  },
  "id": 1
}
```

リクエストを送信します。リクエストが成功すると、以下のような JSON レスポンスが返されます。

```
{
  "jsonrpc": "2.0",
  "result": {
    "status": "OK",
    "message": "ChaincodeLogSmartContract"
  },
  "id": 1
}
```

これで、チェーンコードがデプロイされた状態になったので、チェーンコードを実行します。

Java チェーンコードに対してトランザクションを呼び出す

次は、デプロイして初期化された Java チェーンコードに対し、トランザクションを呼び出します。このセクションでは、トランザクションとして `log` 関数と `query` 関数を呼び出します。

`log` 関数を呼び出すには、`ChaincodeLog` Log リクエストを開いて、このリクエストを送信します (図 16 を参照)。



GitHub から複製した SoapUI プロジェクトを使用しているのではない場合 (または、別の HTTP クライアントを使用している場合)、送信する JSON リクエストは以下のような内容になっているはずです。

```
{
  "jsonrpc": "2.0",
  "method": "invoke",
  "params": {
    "type": 1,
    "chaincodeID": {
      "name": "ChaincodeLogSmartContract"
    },
    "CtorMsg": {
      "args": ["log", "KEY-2", "This is a log message."]
    }
  },
  "id": 2
}
```

リクエストが成功すると、以下のような JSON レスポンスが返されます。

```
{
  "jsonrpc": "2.0",
  "result": {
    "status": "OK",
    "message": "a6f7a4fc-2980-4d95-9ec2-114dd9d0e4a5"
  },
  "id": 2
}
```

`query` 関数を呼び出すには、`ChaincodeLog Query` リクエストを開いて、このリクエストを送信します(図 17 を参照)。



GitHub から複製した SoapUI プロジェクトを使用しているのではない場合 (または、別の HTTP クライアントを使用している場合)、送信する JSON リクエストは以下のような内容になっているはずです。

```
{
  "jsonrpc": "2.0",
  "method": "invoke",
  "params": {
    "type": 1,
    "chaincodeID": {
      "name": "ChaincodeLogSmartContract"
    },
    "ctorMsg": {
      "args": ["query", "KEY-1", "KEY-2"]
    }
  },
  "id": 3
}
```

リクエストが成功すると、以下のような JSON レスポンスが返されます。

```
{
  "jsonrpc": "2.0",
  "result": {
    "status": "OK",
    "message": "84cbe0e2-a83e-4edf-9ce9-71ae7289d390"
  },
  "id": 3
}
```

ターミナル・ウィンドウに、以下のようなソリューション・コードの出力が表示されます。

```
$ ./ChaincodeTutorial/bin/ChaincodeTutorial
Feb 28, 2017 4:18:16 PM org.hyperledger.java.shim.ChaincodeBase newPeerClientConnection
INFO: Inside newPeerClientConnection
Feb 28, 2017 4:18:16 PM io.grpc.internal.TransportSet$1 call
INFO: Created transport io.grpc.netty.NettyClientTransport@10bf86d3(/127.0.0.1:7051)
for /127.0.0.1:7051
Feb 28, 2017 4:18:21 PM io.grpc.internal.TransportSet$TransportListener transportReady
INFO: Transport io.grpc.netty.NettyClientTransport@10bf86d3(/127.0.0.1:7051)
for /127.0.0.1:7051 is ready
Feb 28, 2017 4:34:52 PM com.makotojava.learn.blockchain.chaincode.AbstractChaincode run
INFO: Greetings from run(): function -> init | args -> [KEY-1, Chaincode Initialized]
Feb 28, 2017 4:34:52 PM com.makotojava.learn.blockchain.chaincode.solution.ChaincodeLog
handleLog
INFO: *** Storing log message (K,V) -> (ChaincodeLogSmartContract-CLSC-KEY-1,Chaincode
Initialized) ***
Feb 28, 2017 4:50:27 PM com.makotojava.learn.blockchain.chaincode.AbstractChaincode run
INFO: Greetings from run(): function -> log | args -> [KEY-2, This is a log message.]
Feb 28, 2017 4:50:27 PM com.makotojava.learn.blockchain.chaincode.solution.ChaincodeLog
handleLog
INFO: *** Storing log message (K,V) -> (ChaincodeLogSmartContract-CLSC-KEY-2,This is a log
message.) ***
Feb 28, 2017 5:02:13 PM com.makotojava.learn.blockchain.chaincode.AbstractChaincode run
INFO: Greetings from run(): function -> query | args -> [KEY-1, KEY-2]
Feb 28, 2017 5:02:13 PM com.makotojava.learn.blockchain.chaincode.solution.ChaincodeLog
handleQuery
INFO: *** Query: For key 'ChaincodeLogSmartContract-CLSC-KEY-1, value is 'Chaincode
Initialized' ***
Feb 28, 2017 5:02:13 PM com.makotojava.learn.blockchain.chaincode.solution.ChaincodeLog
handleQuery
```

```
INFO: *** Query: For key 'ChaincodeLogSmartContract-CLSC-KEY-2, value is 'This is a log message.' ***
```

おめでとうございます！皆さんはこれで、将来に向かって最初の一步を踏み出したこととなります。

メソッドを追加したり、実装を変更したりするなど、ChaincodeTutorial プロジェクトにいろいろと手を加えてみてください。そうすることで、チェーンコードを作成するのに慣れてくるはずです。実験を楽しんでください！

まとめ

このチュートリアルでは、ブロックチェーン・テクノロジーの概要、チェーンコード・プログラムとして実装されるスマート・コントラクトの概要、そしてブロックチェーンに関する現在のテクノロジーのランドスケープを簡単に説明しました。

チュートリアルの手順に従って、Java チェーンコード開発環境をセットアップし、その一環として必要なソフトウェアをインストールした上で、ローカル・ブロックチェーン・ネットワークを定義して実行する方法、GitHub の Hyperledger Fabric プロジェクトに含まれるサンプル Java チェーンコード・プログラムをデプロイし、そのチェーンコードに対してトランザクションを呼び出す方法を説明しました。

さらに、Eclipse、JUnit、Gradle を使用して皆さんにとって初めてとなる独自の Java チェーンコード・プログラムを作成してビルドしてから、そのチェーンコード・プログラムをデプロイし、チェーンコードに対して実際にトランザクションを呼び出しました。

ブロックチェーン・テクノロジーとスマート・コントラクトを調べて実際に操作した今、ブロックチェーン・テクノロジーが成熟して市場に普及するようになるにつれ、さらに複雑な Java チェーンコードを作成するための腕を磨くには、皆さんは絶好の出発点にいます。

この出発点から、どの方向へ向かいますか？

次のステップ

これまでに学んだ知識を広げるために、以下のことを提案します。

- [Hyperledger Fabric のアーキテクチャーをさらに詳しく調査する](#)
- [Hyperledger Fabric 自体を構築する](#)
- [Hyperledger メーリング・リストのいずれかに登録する](#)

関連トピック： [What is blockchain?](#) [What is Docker?](#) [チェーンコードについて学ぶ](#)
[Hyperledger Fabric ドキュメント](#) [Hyperledger コミュニティー](#) [Blockchain Developer Center](#)
[IBM ブロックチェーンの基礎: 開発者向けクイック・スタート・ガイド](#) [開発者向け IBM Blockchain コース \(無料\)](#) [developerWorks TV での IBM Blockchain ビデオ](#) [Bluemix での IBM Blockchain サービス](#)

著者について

J Steven Perry



Steve Perry is a software developer, architect, and general Java nut who has been developing software professionally since 1991. His professional interests range from the inner workings of the JVM to UML modeling and everything in-between. Steve has a passion for writing and mentoring; he is the author of *Java Management Extensions* (O'Reilly), *Log4j* (O'Reilly), and the IBM developerWorks articles "[Joda-Time](#)" and "[OpenID for Java Web applications](#)." In his spare time, he hangs out with his kids, plays ice hockey, and practices yoga.

© Copyright IBM Corporation 2017

(www.ibm.com/legal/copytrade.shtml)

商標

(www.ibm.com/developerworks/jp/ibm/trademarks/)