マイクロサービスによる Java 移行（gRPC,OpenSSL,Docker,Envoy）

目次

マイクロサービスによる Java 移行（gRPC,OpenSSL,Docker,Envoy）

【改変履歴】
2024/2/1 項番 5～9 を追加
2024/3/1 項番 10～11 を追加及び誤字修正等

マイクロサービスによる Java 移行（gRPC,OpenSSL,Docker,Envoy）

はじめに

「プログラミング言語ランキング」でネット検索をすると各種の統計資料が見つかります。J2EE が発表されてから企業向けは Java というイメージがありましたが、近年は順位が下がっています（調査目的・範囲で順位は大分異なります）。Java に関しては Oracle 社が知的財産権を取得してすぐに起こした著作権訴訟に始まりパッケージの名前に Javax（JavaEE 向けクラス）の使用制限やライセンスの一部有償化等、自社製品にバンドルしていて大慌てしたメーカも多かったのではないでしょうか。

その他にもバージョン毎のサポート期間の問題があります。サポート期間が終わった後に見つかったセキュリティホールは対策される保証がなく、リスクがあればバージョンアップが必要です。ところが、コンパイラのバージョンが変わると古いソースは使っているメソッドが非推奨になったり、言語規約の変更等でエラーになる場合があります。特に Java9 ではパッケージの再編（rt.jar/tools.jar の廃止）が行われ、これを使っている旧バージョンのソースは動作しなくなりました。オープンソースで多数の人間が関わるためバージョンの上位互換性よりも新しい機能の実現に向かうのは致し方なく、互換性の問題は Java に限らず Web 開発に使われている言語では常に悩みの種になります。

新しい機能を使いたい場合やセキュリティ対策が必要になったとき等、システム全体をバージョンアップするのは費用も時間もかかる上に使っていた外部ライブラリが使えなくなることもあります。必要な機能や対処に限定して独立した環境を最適な言語で作り接続する方式・手法があります。

1. サービス指向アーキテクチャ(SOA)とマイクロサービス

一つのアプリケーションを分割しネットワーク上で一体化する方法に SOA やマイクロサービスがあります。どちらも実装の分割という点は共通していますが、SOA の方はかなり難しいものになります。まず、アーキテクチャ（方式）なので正しく規格化されたサービスに分割する必要があり、設計者の理解や教育が必要になります。でないとデータベースの更新やコミットのタイミングを始めとする機能分担を間違えたり、大きなサービスが一つだけできたり、改変の都度類似機能があちこちに実装されて収拾がつかなくなります。SOA で構築したことで有名な銀行システムは外資メーカが銀行用に作ったミドルウェアの上に国内メーカを主力として開発を行ったそうですが、保守の担当者に至るまで“方式”を文化として引き継ぐことはできたのでしょうか。

余談ですが、この外資メーカと国内メーカがそれぞれメインフレームで過去に作った銀行システムは業務の機能としては同様ですが、作り方（方式）は全く異なっていました。銀行業務には科目を跨ぐ連動という処理形態があり、例えば、定期預金口座から普通預金口座に移す処理では定期側の出金処理が終わった後に普通預金の入金処理が連動します。外資メーカの方式では定期口座側の処理が終わったら、後はミドルウェアにお任せで別の（かもしれない）CPU で稼働している普通預金の呼出しを行うという分散処理になっていました。国内メーカの方は全ての処理を同一の主記憶上で連続して実行するために、同時配置が必要なモジュールの組合せとサイズを人間が調べて主記憶上の配置を決めたロードモジュールを作り 1 CPU（論理的な）で処理を完結させます。

もう一つ忘れられない出来事に、大手通信会社のお客様と夜間の DB 環境切替作業を見守っていた際に「F 社の DB は凄く性能が出るんだけど、その性能が出るバンドが凄く狭いんだよな～」とこぼされ居心地の悪さと同時に、うまいこと言うなぁ...と感じたことがあります。

国内メーカは現有設備かつ現データで最高性能が出るように、外資メーカは性能が足りなければ設備を増設・増強し DB の状態が日々変動してもサービスレベルが維持できる設計にしているわけです。

マイクロサービスによる Java 移行（gRPC,OpenSSL,Docker,Envoy）

何が言いたいかというと、SOA は機能分散の全体最適化が難しく、凝縮させるのが好きな国民性（個人的な感想です）には向いていないのではないかということです。ただし、現状最適でガチガチに組んだシステムは変化への対応が難しくなります。今後は分散指向や、現状適合度が多少下がっても将来の変化に対応するための遊びを用意する感覚が必要だと思います。

一方、マイクロサービスは単独モジュール（Java の場合はメソッド）の別環境への切り出しです。例えば現システムで使っているバージョンで動作しないソフトを使いたいときに、マイクロサービスとして別環境で稼働させることができます。勿論 こちらも乱用すると収拾が付かなくなるし、環境上のリスクも伴いますが、漸進的な他言語への移行や緊急時の対策に使える可能性があります。

2. マイクロサービスの機能

マイクロサービスに関連するソフトウェアには各種あり、そもそもマイクロ―サービスの定義も微妙に異なるのですが、主要な分野としては以下のものがあります。

・ハブ／ゲートウェイ／キュー機能 … メッセージの通信経路で振分や停止、負荷分散等
・監視 … コンテナ化したサービスの動態観察や制御
・セキュリティ … 認証やアクセス制御
・API … サービス呼び出しのインタフェース

この資料ではセキュリティへの配慮を含めた API の使い方を中心に説明します。

3. マイクロサービスのリスク

マイクロサービスはアプリケーションの一部をネットワーク越しに切りだすので、これに伴い一般的に以下のリスクが発生します。

・セキュリティ … 更新系や非公開の情報を扱うサービスではクライアントの認証が必要です
・遅延 …ディスク IO 以上の遅延が発生する場合があります
・サービス停止 … 稼働する筐体やコンテナ、サービス自体の生死や負荷の監視が必要です

4. 実装例

殆どのマイクロサービスは Web サービスで JSON 等のテキストデータをやり取りする形態ですが、独自の通信形態を持つものに gRPC（Google が開発してオープンソース化…Apache License 2.0）があります。インタフェースにバイナリデータを使うことで高速な上にスマートフォン向けにも開発が進んでおり主要な Web 開発言語に対応しているので多言語でのシステム構築が可能です。

4.1. gRPC の機能

gRPC はクライアント／サーバ間の RPC（リモート・プロシージャ・コール）接続の基盤で、認証やフロー制御も行います。データ交換に関する機能は Protocol Buffers という別プロジェクトで開発していて、インタフェース定義から各種開発言語毎のインタフェース（メッセージの構造体とそれを扱うためのクラス）が生成できます。

マイクロサービスによる Java 移行（gRPC,OpenSSL,Docker,Envoy）

4.2. gRPC を使った開発

開発は以下の順で行います。

① マイクロサービスのインタフェース定義ファイル（拡張子 .proto）を作る

② 開発言語毎の Protocol Buffer Compiler(protoc)でインタフェース定義をコンパイルする

※これにより通信メッセージのクラスとサーバ、クライアントのスタブクラスを生成する

③ サーバ／クライアントのスタブを呼び出すことでサーバ、クライアント間の通信を行う

以下、Java のクライアントから Python のマイクロサービスにリクエストを投げる例で説明します。

＜実装する内容＞

・RDB に伝票データを登録する既存の Web サービスにジャーナル出力機能を追加する

・ジャーナルは登録／更新リクエストの都度 RDB に登録する内容をそのまま出力する

・ジャーナル出力機能をマイクロサービス化する。インタフェースは伝票データと処理結果の真偽値

（1） インタフェース定義

マイクロサービスのインタフェース定義を作ります。この例では伝票テーブルの定義を基にして denpyo.proto ファイルを作ります。

〔伝票テーブル〕

```
postgres=# ¥d denpyo
                テーブル"public.denpyo"
    列     |          タイプ          | 照合順序 | Null 値を許容 | デフォルト
-----------+---------------------------+----------+---------------+------------
 err       | boolean                   |          |               |
 trdate    | character(8)              |          | not null      |
 denpyono  | character(5)              |          | not null      |
 tantou    | character(4)              |          |               |
 karikamokucd | character(6)[]         |          |               |
 karikingaku | character varying(13)[] |          |               |
 kasikamokucd | character(6)[]         |          |               |
 kasikingaku | character varying(13)[] |          |               |
 tekiyo    | character varying(50)     |          |               |
 sysdate   | character(8)              |          | not null      |
 errmessage | character varying(100)   |          |               |
インデックス:
    "pk_denpyo" PRIMARY KEY, btree (denpyono)
```


マイクロサービスによる Java 移行（gRPC,OpenSSL,Docker,Envoy）

インタフェース定義にはメッセージ定義とサービス定義を記述し、拡張子.proto のテキストファイルとして保存します。2023 年末の最新規約は proto3 で、記法の詳細は以下のサイトにあります。

https://protobuf.dev/programming-guides/proto3/

https://protobuf.dev/reference/protobuf/proto3-spec/

定義は開発言語間で共用できますが、言語により制約や付加情報が付いている場合があります。

例えば、Java の場合はパッケージ名の指定等ができ、また、2023 年末のバージョンでは map 型は入れ子に含められない等の制約があります。

https://protobuf.dev/reference/java/java-generated/

【インタフェース定義（denpyo.proto）】

```
syntax = "proto3";
option java_package = "jp.co.focs.grpc.denpyo";
option java_multiple_files = true; // デフォルトはfalse…1つの外部クラスでラップして作成

// リクエストメッセージ（伝票）
message Denpyo {
  message Meisai {
    string kamokucd = 1;
    string kingaku  = 2;
  }
  bool   err      = 1;
  string trdate   = 2;
  string denpyono = 3;
  string tantou   = 4;
  repeated Meisai kari = 5;
  repeated Meisai kasi = 6;
  string tekiyo     = 7;
  string sysdate    = 8;
  string errmessage = 9;
}

// サービス定義
service DenpyoJournaling {
  rpc Journaling (Denpyo) returns (Ret) {}
}

// サーバー処理結果
message Ret {
  bool success = 1;
}[EOF]
```

メッセージは入れ子にすることができ（生成されるクラスも入れ子になります）、今回の例では配列になっている科目コードと金額で Meisai というサブメッセージを作り、Denpyo メッセージの項目 kari と kasi から繰り返し項目（repeated）として参照しています。

マイクロサービスによる Java 移行（gRPC,OpenSSL,Docker,Envoy）

（３） クライアント（Java）

インタフェース定義から Java 用に生成したクラスを使って開発を行います。以下の例は単純な同期 RPC（他に相互または片方向のストリーミングがあります）で、次の手順によりリクエストを送ってメソッドを呼び出し戻りを待ち受けます。

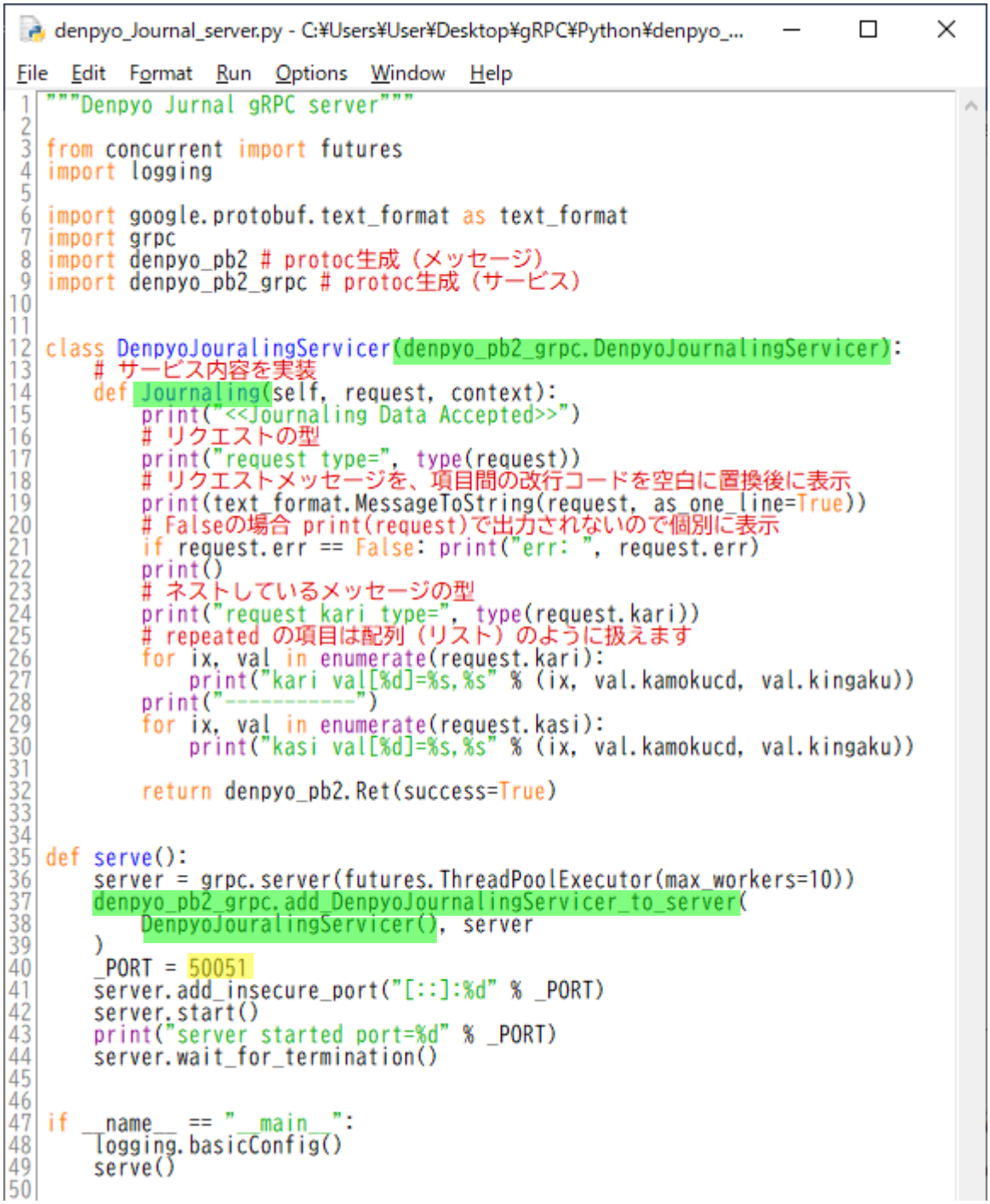
① io.grpc.ManagedChannel でサービス（ホスト IP とポート）への経路を作成する(channel)
② インタフェース定義から生成したサービスのスタブをインスタンス化する(client)
③ インタフェース定義から生成したメッセージのビルダーをインスタンス化(denpyoBuilder)し、セッターで値を設定(set*Xxxxx*)する（戻り値もビルダーなのでセッターのドット連結が可能）
※入れ子メッセージ（Denpyo.Meisai）も同様にビルダーを生成して値を設定し、親に add*Xxxx*
④ サービスのインスタンス(client)を通して RPC のメソッド(journaling)を呼出す

```
//---------- gRPCでジャーナル出力 --------------
// cahcnnel と client は複数のサーバ呼び出しで使い回しが可能
ManagedChannel channel;
DenpyoJournalingGrpc.DenpyoJournalingBlockingStub client;
// 実行環境に依存ライブラリが不足していると発生するRuntimeExceptionをThrowableでcatch
try {
  ①channel = ManagedChannelBuilder.forAddress("localhost", 50051).usePlaintext().build();
  ②client = DenpyoJournalingGrpc.newBlockingStub(channel);
} catch (Throwable   e){
    logger.log(Level.SEVERE, e.getMessage(), e.getStackTrace());
    throw e;
}

// リクエスト送信データ（最上位メッセージ）を作成
Denpyo.Builder denpyoBuilder = Denpyo.newBuilder()
  ③    .setDenpyono(denNo)
       .setErr(false)
       .setErrmessage(checkMsg==null? "":checkMsg) //nullだと例外が発生する
       .setTantou(tantouCd)
       .setTekiyo(tekiyou)
       .setSysdate( DateTimeFormatter.ofPattern("yyyyMMdd").format( LocalDateTime.now() ) );
// 子階層メッセージの借方明細の作成
for(Map<String, ?> kariGyo : kariMeisai) {
    denpyoBuilder.addKari(Denpyo.Meisai.newBuilder()
        .setKamokucd((String) kariGyo.get("kariKamokuCd"))
        .setKingaku((String) kariGyo.get("kariKingaku"))
    );
}
// 子階層メッセージの貸方明細の作成
for(Map<String, ?> kasiGyo : kasiMeisai) {
    denpyoBuilder.addKasi(Denpyo.Meisai.newBuilder()
        .setKamokucd((String) kasiGyo.get("kasiKamokuCd"))
        .setKingaku((String) kasiGyo.get("kasiKingaku"))
    );
}
Denpyo request = denpyoBuilder.build();
logger.log(Level.INFO, "> DenpyoJournalingGrpc request=" + request);

// 送信
Ret result;
try {
  ④ result = client.journaling(request);
} catch (StatusRuntimeException  e){
    logger.log(Level.SEVERE, e.getMessage(), e.getStackTrace());
    throw e;
} catch (Throwable   e){
    logger.log(Level.SEVERE, "gRCP sent error", e);
    throw e;
}
logger.log(Level.INFO, "> DenpyoJournalingGrpc return=" + result);
//---------- ジャーナル出力 終了 --------------
```


マイクロサービスによる Java 移行（gRPC,OpenSSL,Docker,Envoy）

（4） サーバ（Python）

インタフェース定義から Python 用に生成したクラスの、サービスに関する機能を使って開発を行います。リクエストを待ち受けるポート番号以外はインタフェース定義に依存した名前が自動的に決まります（下記のソース、緑色部分）。下記の例はリクエストメッセージを標準出力に書き出すだけでレスポンスの真偽値に真を設定して戻しています。

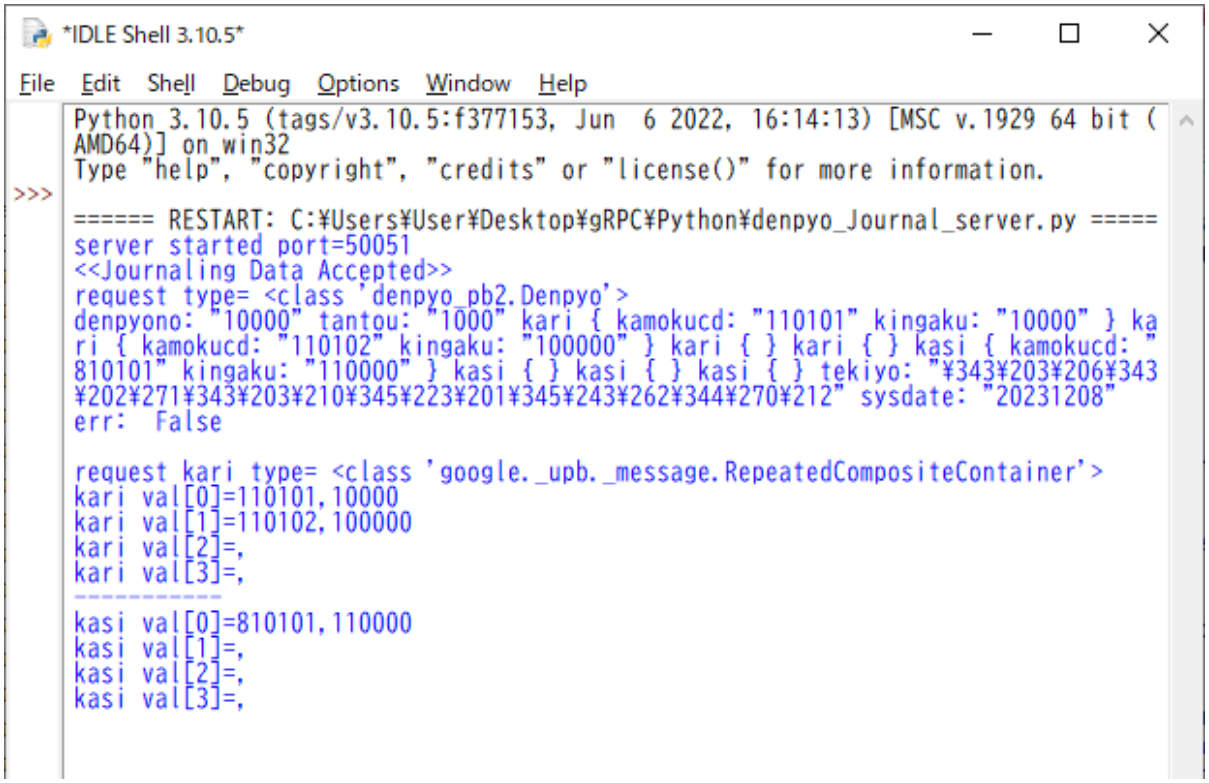


マイクロサービスによる Java 移行（gRPC,OpenSSL,Docker,Envoy）

（5）操作

入力内容

サーバ側の表示内容（Python の標準出力）



マイクロサービスによる Java 移行（gRPC,OpenSSL,Docker,Envoy）

5. 実装手順

インタフェース定義をエディターで作成した後、以下の手順で作業します（Windows を使った例）。

5.1. protoc による Java ソース生成

ソース生成は Protocol Buffer プロジェクトで開発している protoc（proto コンパイラ）を使います。コンパイラは以下のサイトに説明や入手先のリンクがあります。

https://grpc.io/docs/protoc-installation/#install-pre-compiled-binaries-any-os

protoc は開発言語共通で、指定するプラグインによって出力される言語が変わります。

（1） Java プラグインと依存ライブライ

Java 用のプラグインは以下のサイトから最新版が入手できます。

https://github.com/grpc/grpc-java/tree/master/compiler#grpc-java-codegen-plugin-for-protobuf-compiler

もう少し簡便に、protoc とプラグインのバージョンを合わせるには以下のようにします。

① Java 用のサンプルを取得

以下のサイトにあるリンクから zip 化したサンプルコードをダウンロードし解凍します。永続的に保存する必要はないのでデスクトップに解凍したことにして以降の説明を行います。

https://grpc.io/docs/languages/java/quickstart/#get-the-example-code

② サンプルのコンパイル

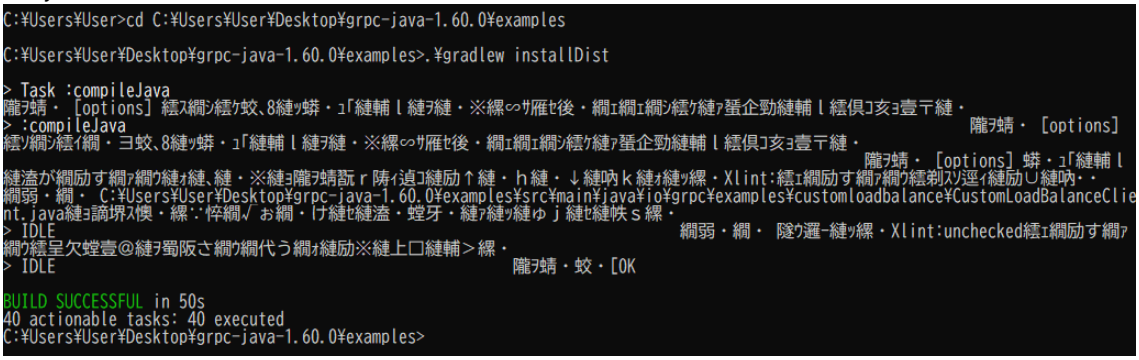
以下リンクの[1. Compile the client and server]で説明している内容を実行します。

https://grpc.io/docs/languages/java/quickstart/#run-the-example

具体的には、grpc-java-*x.xx.x* の **examples** ディレクトリで .¥gradlew installDist を実行します。

※サイトの説明は Linux 用になっているので、ファイルパスは “./”⇒”.¥”と置き換えます

※JDK21 を使い、Windows コマンドプロンプトで実行すると文字化けしたメッセージが出ます



JDK11 でも注意メッセージが出ますがどちらにしろ”BUILD SUCCESSFUL”が表示されれば OK

マイクロサービスによる Java 移行（gRPC,OpenSSL,Docker,Envoy）

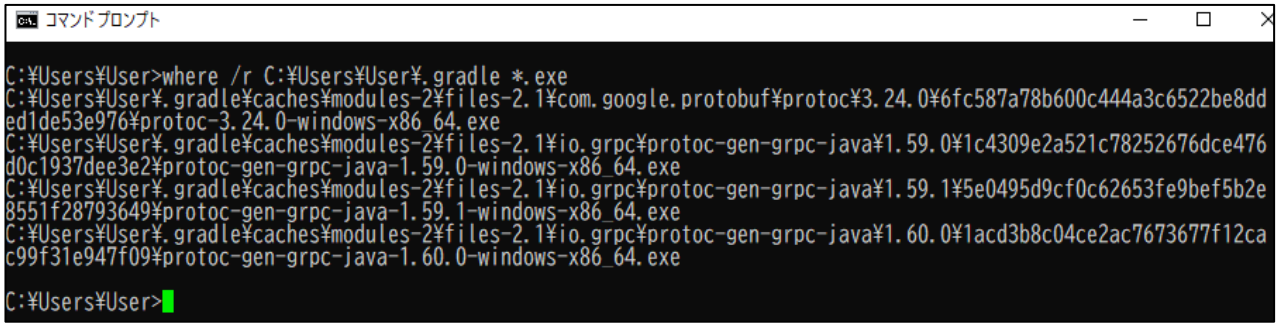
③ ptotoc とプラグイン及び依存ライブラリの保存

②で gradle が build.gradle に書かれている資材（protoc やプラグイン、gRPC が使うライブラリ）を集めるので、これを開発に使えるように保存しておきます。

・protoc、プラグインがダウンロードされる場所

ホームディレクトリの.gradle の下（ユーザ ID が”User”だったら C:¥Users¥User¥.gradle）

Windows のコマンド where /r C:¥Users¥User¥.gradle *.exe で見つけることができます



protoc-*x.xx.x*-windows~.exe と protoc-gen-grpc-java-*x.xx.x*-windows~.exe のバージョン(*x.xx.x*)が一番大きい（新しい）ものを保存しておきます。

・gRPC 依存ライブラリが集められている場所

②の gradlew を実行したディレクトリ配下、インストール用の lib ディレクトリの中に gRPC が必要とする jar ファイルが集められています

grpc-java-*x.xx.x*¥examples¥build¥install¥examples¥lib

（２） ptotoc の実行

開発対象のインタフェース定義を入力に、③で保存したプラグインを使って protoc コマンドを実行します。(コマンドを複数行で入力する場合は下の例のように、行末に“^”を付けて改行します)

```
protoc-x.xx.x-windows-x86_64^
    --plugin=protoc-gen-grpc-java=<③保存パス>¥protoc-gen-grpc-java-x.xx.x-windows-x86_64.exe^
    --java_out=<保存先パス>¥java-generated¥java^
    --grpc-java_out=<保存先パス>¥java-generated¥grpc^
     <インタフェース定義パス>¥denpyo.proto
```

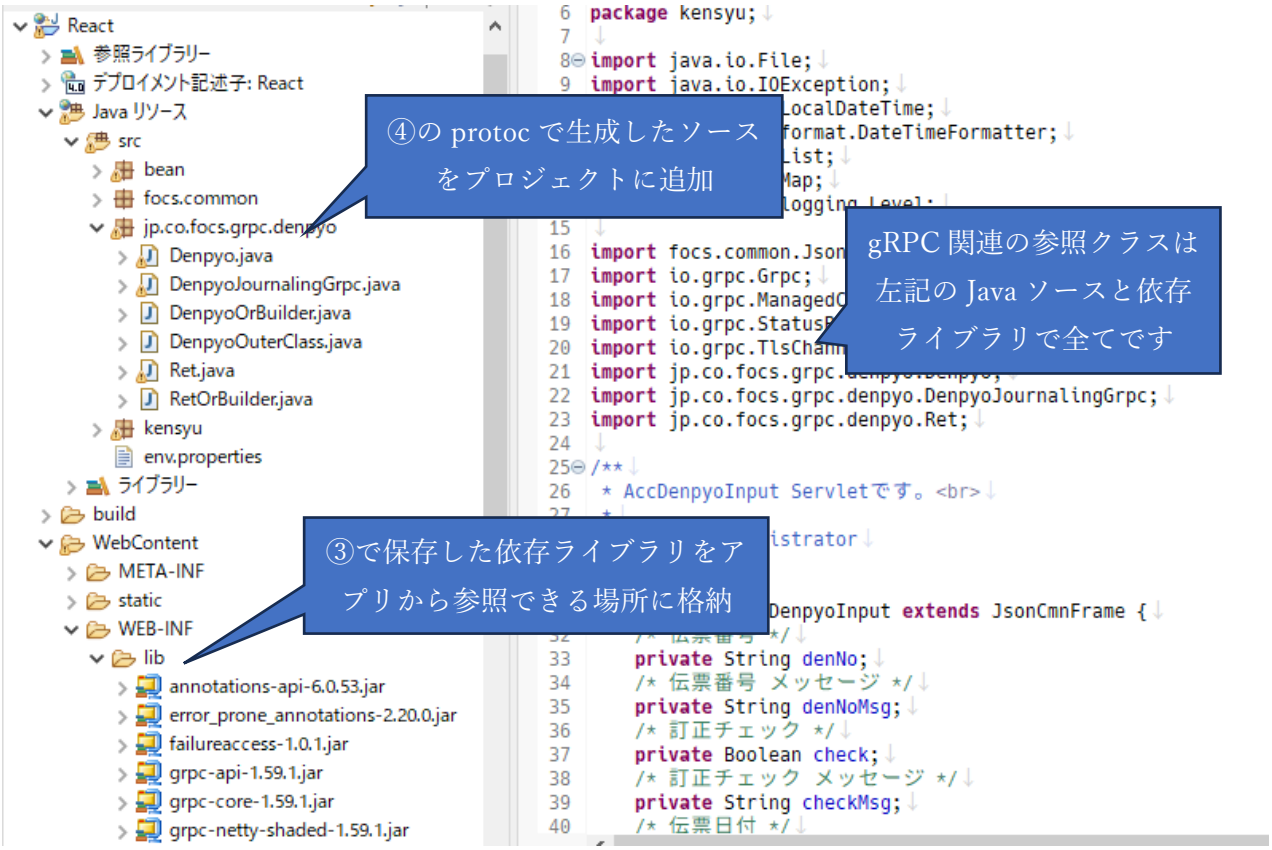
コマンドの実行により java_out にメッセージ編集用のクラス群、grpc-java_out に gRPC 通信用のクラス（スタブ）１つができます。

ディレクトリ構成はインタフェース定義の option java_package に従います。

マイクロサービスによる Java 移行（gRPC,OpenSSL,Docker,Envoy）

（3） 既存アプリケーションへの組み込み

この例では古いアプリケーションへの組み込みもできるように依存ライブラリを手動で設定していますが、Maven や Gradle を使っているのであればはそちらの依存定義に含めることができます。



5.2. Python のソース生成

以下のサイトに Linux 環境を前提とした gRPC の始め方が書かれています。

https://grpc.io/docs/languages/python/quickstart/

※ Windows で開発する場合は以降の手順で作業してください

（1） Python のインストールと設定

Python をこれからインストールする場合は、以下のサイトからインストーラがダウンロードできます。ホームページ上部に最新版のダウンロードボタンが表示されるので、他の目的がなければこれを使います。

https://www.python.org/downloads/

デフォルト設定でインストールすると py.exe（Python ランチャ）で python.exe を起動する環境になるので、その前提で以降 説明します。

マイクロサービスによる Java 移行（gRPC,OpenSSL,Docker,Envoy）

（2） gRPC 関連資材のインストール

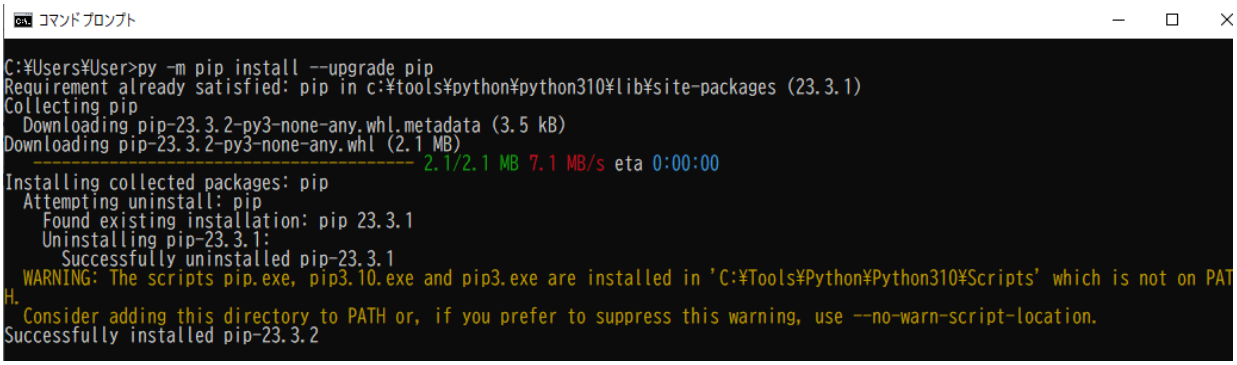
Python へは pip を使って関連のモジュール、grpcio と grpc-tools をインストールします。

Python はバージョン 3.7 以上が必要です。このバージョンには pip が同梱されているので、改めて pip をインストールする必要はなく、最新化から作業します。

・pip の最新化 … コマンドプロンプト等で以下のコマンドを実行します。

```
py -m pip install --upgrade pip
```

※ 警告がでますが、Successfully installed pip-*xx.xx.x* が表示されたら問題ありません。



・gRPC 関連資材のインストール

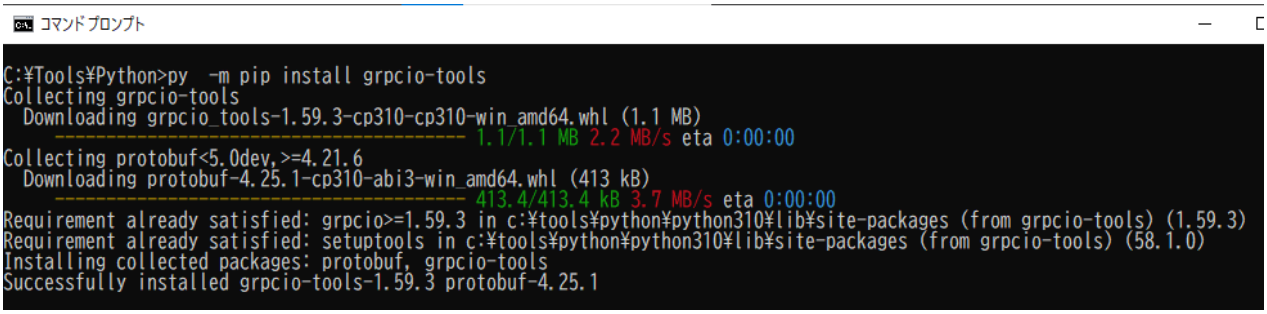
pip を使って以下の 2 つをインストールします。

grpcio　　：Protocol Buffers 以外の grpc

grpcio-tools ：Protocol Buffers とサービス定義ファイルからコードを自動生成するためのツール

```
> Py -m pip install grpcio
```


```
> Py -m pip install grpcio-tools
```



Python の開発環境は以上で完成です。

マイクロサービスによる Java 移行（gRPC,OpenSSL,Docker,Envoy）

（３） ptotoc の実行

開発対象のインタフェース定義を入力に、grpcio-tools の protoc コマンドを実行します。

＜コマンド実行時のファイル構成＞

```
C:¥USERS¥USER¥DESKTOP¥GRPC
 ├─denpyo.proto
 │
 └──Python-generated
     （空ディレクトリ）
```

【実行コマンド】

```
cd C:¥Users¥User¥Desktop¥gRPC¥Python-generated
py -m grpc_tools.protoc ^
 -I..¥ ^
 --python_out=. ^
 --pyi_out=. ^
 --grpc_python_out=. ^
 denpyo.proto
```

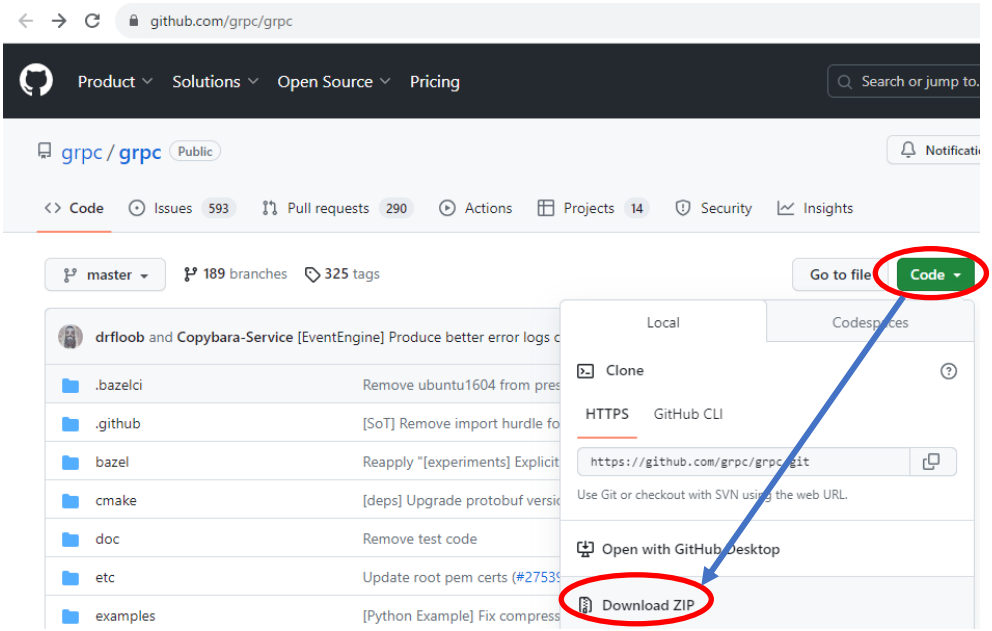
＜コマンド実行後のファイル構成＞

```
C:¥USERS¥USER¥DESKTOP¥GRPC
 │   denpyo.proto
 │
 └──Python-generated
        denpyo_pb2.py
        denpyo_pb2.pyi
        denpyo_pb2_grpc.py
```

※生成された denpyo_pb2.py と denpyo_pb2_grpc.py を開発資材から参照可能なディレクトリ（小規模であれば全て同一のディレクトリに集めても問題ありません）にコピーします

（４） Python のコードサンプル

examples ・・・ https://github.com/grpc/grpc



マイクロサービスによる Java 移行（gRPC,OpenSSL,Docker,Envoy）

6. 安全な接続のための構成（SSL/TLS）

プログラムの一部を切り離して API 経由で呼び出すようにすると不特定のプログラムから呼び出せるようになります。例えば、入力チェックとデータベース更新を行っていたプログラムの更新部分だけを切り離した場合、チェック処理を迂回して更新処理が直接呼び出される危険があります。

そこで、リクエストを受ける側（サーバ）がクライアントが正規の相手か否かを確認（認証）できる、クライアント／サーバ間の SSL/TLS（現在では SSL は使われないので、以降は TLS に限定）による相互認証を組込みます。システムの利用者の属性による認証／承認、例えば、正規の利用者か職務・役職がシステムを操作する権限を満たしているか否かを確認するのはマイクロサービスを呼出す Web アプリ等のクライアント側の役割分担です（更に厳重にしたいのであれば gRPC にはメタ情報の受け渡し機能によるセキュリティの強化も可能で、実装事例がネットで見つけれらます）。

6.1. 必要な証明書と鍵

TLS 接続は証明書で通信相手を認証します。証明書は IP アドレスやドメイン名と公開鍵が書かれたファイル（CSR:証明書署名要求）に認証局の秘密鍵（”Private Key”、“私有鍵”、“非公開鍵”と同）で電子署名することで作られます。

【証明書の内容】JPNIC https://www.nic.ad.jp/ja/newsletter/No23/080.html

図4：証明書の例

X.509 形式の証明書で使われる識別名（DN）の要素

要素	意味
C（Country）	国名
O（Organization）	組織名
OU（Organizational Unit）	部門名 ※2022 年 9 月 1 日以降に発行のサーバー証明書は OU の利用が禁止となりました
CN（Common Name）	一般名

マイクロサービスによる Java 移行（gRPC,OpenSSL,Docker,Envoy）

サーバとクライアントが提示した証明書の真正性は当該証明書に署名している認証局の公開鍵を使って検証します。署名している認証局の真正性は認証局の証明書に署名している上位の認証局の署名で検証し、最上位のルート認証局の自己証明書をルート証明書として OS の”証明書ストア”かアプリケーション固有の参照先に格納しておくことで成立します。

したがって、相互認証を行うためにはサーバの公開鍵を記載した認証局署名済の証明書と秘密鍵、クライアントの公開鍵を記載した認証局署名済の証明書と秘密鍵、認証局の公開鍵を記載した証明書と秘密鍵が必要になります。公式（社外に提示する）には行政組織やそれを代行する組織[1]が発行したルート証明書を使うべきですが、ここではテスト用のルート証明書を使う方法を説明します。

【注意】証明書や秘密鍵は OS からみれば単純なファイルなので、Windows の標準的なアクセス制御の環境では普通に複写できます。例えば、ブラウザが参照しているルート証明機関の証明書ストアにテスト用の証明書を格納すると攻撃者にとても都合の良い状態になります

6.2. OpenSSL による証明書作成

（1） 証明書作成ツールのダウンロード

証明書は OpenSSL（オープンソース Apache License v2）を使って作成することができます。OpenSSL の実行形式は、https://www.openssl.org/community/binaries.html からリンクされた wiki でダウンロードサイトを紹介しています。日本国内（.jp ドメイン）にも私的なダウンロードサイトがありますが、利用する場合はブラウザに表示される URL ではなく実際にリンクされているアドレスをページのソースから確認して信頼できるサイトであることを確認してください。

[1] e-gov 認証局のご案内 https://shinsei.e-gov.go.jp/contents/preparation/certificate/certification-authority.html

マイクロサービスによる Java 移行（gRPC,OpenSSL,Docker,Envoy）

（２） 証明書の作成

Windows で OpenSSL を使って証明書を作成するバッチファイルの例を以下に示します。

※ O=、CN=、subjectAltName= の値を環境に合わせて設定。DNS はワイルドカード「*」使用可

※ subjectAltName= に含めた localhost や IP:0.0.0.0 は本番のドメイン名(CN)以外で試験するため

【*TLS 証明書作成*.bat の例】

```
cd /d %~dp0

REM 1. 認証局（CA） 秘密（私有）鍵 と 自己署名証明書（ルート証明書）生成
openssl req -x509 -newkey rsa:4096 -days 365 -nodes -keyout ca.key -out ca.pem -subj "/C=JP/O=FOCS/CN=root.focs"

echo ルート証明書
openssl x509 -in ca.pem -noout -text

REM 2. サーバ秘密（私有）鍵 と 証明書署名要求（CSR）生成
openssl req -newkey rsa:4096 -nodes -keyout server.key -out server.csr -subj "/C=JP/O=FOCS/CN=*.server.focs"

REM CN と subjectAltName は一致させる。localhost を使う場合は IP:0.0.0.0 を 証明書の Subject Alternative Name（SAN）に追加する.
echo subjectAltName=DNS:*.server.focs,DNS:localhost,IP:0.0.0.0 > server-ext.cnf
REM Or you can use localhost DNS and grpc.ssl_target_name_override variable
REM echo subjectAltName=DNS:localhost > server-ext.cnf

REM 3. 認証局の秘密鍵で署名して サーバの CSR から証明書を取得
openssl x509 -req -in server.csr -days 365 -CA ca.pem -CAkey ca.key -CAcreateserial -out server.pem -extfile server-ext.cnf

echo 署名済サーバ証明書
openssl x509 -in server.pem -noout -text

REM 4. クライアント秘密（私有）鍵 と 証明書署名要求（CSR）生成
openssl req -newkey rsa:4096 -nodes -keyout client.key -out client.csr -subj "/C=JP/O=FOCS/CN=*.client.focs"

REM CN と subjectAltName は一致させる。localhost を使う場合は IP:0.0.0.0 を 証明書の Subject Alternative Name（SAN）に追加する.
echo subjectAltName=DNS:*.client.focs,IP:0.0.0.0 > client-ext.cnf

REM 5. 認証局の秘密鍵で署名して クライアントの CSR から証明書を取得
openssl x509 -req -in client.csr -days 365 -CA ca.pem -CAkey ca.key -CAcreateserial -out client.pem -extfile client-ext.cnf

echo 署名済クライアント証明書
openssl x509 -in client.pem -noout -text
```

〔*TLS 証明書作成*.bat 実行後のディレクトリ〕

```
C:¥USERS¥USER¥DESKTOP¥GRPC¥CERT
    ca.key
    ca.pem
    ca.srl
    client-ext.cnf
    client.csr
    client.key
    client.pem
    server-ext.cnf
    server.csr
    server.key
    server.pem
    TLS 証明書作成.bat
```

〔使うのは以下の６ファイル〕

- 認証局のルート証明書（ca.pem）、秘密鍵（ca.key）
- サーバ証明書（server.pem）、秘密鍵（server.key）
- クライアント証明書（client.pem）、秘密鍵（client.key）

6.3. アプリケーションへの組み込み

（1） Java クライアント

TLS 認証を使うための変更点は、TlsChannelCredentials.Builder のインスタンス（tlsBuilder）を作り、ManagedChannel（channel）を生成する際に資格情報を生成（build()）して渡すだけです。

```
//---------- gRPC でジャーナル出力 --------------
// cahcnnel と client は複数のサーバ呼び出しで使い回しが可能
// ※※ TLS 追加 ※※
String CLIENTCERTCHAINFILEPATH = "C:¥¥Users¥¥User¥¥Desktop¥¥gRPC¥¥cert¥¥client.pem";  /*クライアント証明書*/
String CLIENTPRIVATEKEYFILEPATH = "C:¥¥Users¥¥User¥¥Desktop¥¥gRPC¥¥cert¥¥client.key"; /*クライアント秘密鍵*/
String TRUSTCERTCOLLECTIONFILEPATH = "C:¥¥Users¥¥User¥¥Desktop¥¥gRPC¥¥cert¥¥ca.pem";  /*ルート証明書*/

TlsChannelCredentials.Builder tlsBuilder = TlsChannelCredentials.newBuilder();
try {
    tlsBuilder.keyManager(new File( CLIENTCERTCHAINFILEPATH ), new File( CLIENTPRIVATEKEYFILEPATH ));
    tlsBuilder.trustManager(new File( TRUSTCERTCOLLECTIONFILEPATH ));
} catch (IOException e) {
    e.printStackTrace();
}
// ※※ TLS 追加 end ※※

ManagedChannel channel;
DenpyoJournalingGrpc.DenpyoJournalingBlockingStub client;
// 実行環境に依存ライブラリが不足していると発生する RuntimeException を Throwable で catch
try {
    //※TLS 削※  channel = ManagedChannelBuilder.forAddress("localhost", 50051).usePlaintext().build();
    / ※TLS 追※/ channel = Grpc.newChannelBuilderForAddress("localhost", 50051, tlsBuilder.build()).build();
    client = DenpyoJournalingGrpc.newBlockingStub(channel);
} catch (Throwable   e){
    logger.log(Level.SEVERE, e.getMessage(), e.getStackTrace());
    throw e;
}

// リクエスト送信データ（最上位メッセージ）を作成
Denpyo.Builder denpyoBuilder = Denpyo.newBuilder()
        .setDenpyono(denNo)
        .setErr(false)
        .setErrmessage(checkMsg==null? "":checkMsg) //null だと例外が発生する
        .setTantou(tantouCd)
        .setTekiyo(tekiyou)
        .setSysdate( DateTimeFormatter.ofPattern("yyyyMMdd").format( LocalDateTime.now() ) );
// 子階層メッセージの借方明細の作成
for(Map<String, ?> kariGyo : kariMeisai) {
    denpyoBuilder.addKari(Denpyo.Meisai.newBuilder()
        .setKamokucd((String) kariGyo.get("kariKamokuCd"))
        .setKingaku((String) kariGyo.get("kariKingaku"))
    );
}
// 子階層メッセージの貸方明細の作成
for(Map<String, ?> kasiGyo : kasiMeisai) {
    denpyoBuilder.addKasi(Denpyo.Meisai.newBuilder()
        .setKamokucd((String) kasiGyo.get("kasiKamokuCd"))
        .setKingaku((String) kasiGyo.get("kasiKingaku"))
    );
}
Denpyo request = denpyoBuilder.build();
logger.log(Level.INFO, "> DenpyoJournalingGrpc request=" + request);

// 送信
Ret result;
try {
    result = client.journaling(request);
} catch (StatusRuntimeException  e){
    logger.log(Level.SEVERE, e.getMessage(), e.getStackTrace());
```



```
            throw e;
        } catch (Throwable   e){
            logger.log(Level.SEVERE, "gRCP sent error", e);
            throw e;
        }
        logger.log(Level.INFO, "> DenpyoJournalingGrpc return=" + result);
        //----------- ジャーナル出力 終了 --------------
```

（２） Python サーバ

TLS 相互認証を可能にするには、server()関数の中でポートを開く際に①add_secure_port を使いクライアント認証要（require_client_auth=True）にし、②サーバの証明書、秘密鍵、ルート証明書をサーバから参照可能な場所（プロジェクトと同一か配下のディレクトリ）に格納しておきます。

```
#!/usr/bin/env python
"""Denpyo Jurnal gRPC server"""

from concurrent import futures
import logging
import google.protobuf.text_format as text_format
import grpc
import denpyo_pb2 # protoc 生成（メッセージ）
import denpyo_pb2_grpc # protoc 生成（サービス）

class DenpyoJouralingServicer(denpyo_pb2_grpc.DenpyoJournalingServicer):
    # サービス内容を実装
    def Journaling(self, request, context):
        print()
        print("<<Journaling Servicer Context>>")
        print("[client]peer()=", context.peer())
        print("x509_subject  =", context.auth_context().get('x509_subject'))
        print()

        print("<<Journaling Data Accepted>>")
        # リクエストの型
        print("request type=", type(request))
        # リクエストメッセージを、項目間の改行コードを空白に置換後に表示
        print(text_format.MessageToString(request, as_one_line=True))
        # False の場合 print(request)で出力されないので個別に表示
        if request.err == False: print("err: ", request.err)
        print()

        # ネストしているメッセージの型
        print("request kari type=", type(request.kari))
        # repeated の項目は配列（リスト）のように扱えます
        for ix, val in enumerate(request.kari):
            print("kari val[%d]=%s,%s" % (ix, val.kamokucd, val.kingaku))
        print("-----------")
        for ix, val in enumerate(request.kasi):
            print("kasi val[%d]=%s,%s" % (ix, val.kamokucd, val.kingaku))

        ###
        # 業務処理を既述
        ###

        return denpyo_pb2.Ret(success=True)
```



```
def serve():
    server = grpc.server(futures.ThreadPoolExecutor(max_workers=10))
    denpyo_pb2_grpc.add_DenpyoJournalingServicer_to_server(
        DenpyoJouralingServicer(), server
    )
    _PORT = 50051
    ######## TLS(クライアント認証要-mTLS) 追加 start #######
    keyfile  = 'cert/server.key'
    certfile = 'cert/server.pem'
    cacert   = 'cert/ca.pem'
    private_key = open(keyfile, 'rb').read()
    certificate_chain = open(certfile, 'rb').read()
    cacert = open(cacert, 'rb').read()

    credentials = grpc.ssl_server_credentials(
        [(private_key, certificate_chain)]
         ,root_certificates=cacert
         ,require_client_auth=True  #クライアント認証要否
    )
    server.add_secure_port("[::]:%d" % _PORT, credentials)
    #TLS:削> server.add_insecure_port("[::]:%d" % _PORT)
    ######## TLS 追加 end #######
    server.start()
    print("server started port=%d" % _PORT)
    server.wait_for_termination()


if __name__ == "__main__":
    logging.basicConfig()
    serve()
```

6.4. 相互認証の確認

クライアントの TLS 認証が「必須」になっているか否かは、OpenSSL の診断ツール（s_client[2]）からサーバにアクセスすることで確認することができます。以下がコマンドの実行例です。

```
openssl s_client -state -verify 1^
 -connect localhost:50051 -servername s1.server.focs^
 -CAfile C:¥Users¥User¥Desktop¥gRPC¥cert¥ca.pem^
 -key C:¥Users¥User¥Desktop¥gRPC¥cert¥client.key^
 -cert C:¥Users¥User¥Desktop¥gRPC¥cert¥client.pem
```

※コマンドは TLS の接続手順の経過（gRPC の通信が始まる前）を表示するので、手続きの途中にクライアントの認証が含まれていることを確認します

[2] サーバ側の診断をする s_server もあり、そちらは Cafile,key,cert にサーバ側のファイルを指定し、起動すると connect で指定したポートでクライアントからのリクエストを待ち受けます

マイクロサービスによる Java 移行（gRPC,OpenSSL,Docker,Envoy）

相互認証（クライアント認証）が必須になっていると、以下のように...

“SSL_connect:SSLv3/TLS write client certificate” が表示されます。

```
コマンド プロンプト
C:¥Users¥User>openssl s_client -state -verify 1^
More?  -connect localhost:50051 -servername s1.server.focs^
More?  -CAfile C:¥Users¥User¥Desktop¥gRPC¥cert¥ca.pem^
More?  -key C:¥Users¥User¥Desktop¥gRPC¥cert¥client.key^
More?  -cert C:¥Users¥User¥Desktop¥gRPC¥cert¥client.pem
verify depth is 1
Connecting to ::1
CONNECTED(000001C4)
SSL_connect:before SSL initialization
SSL_connect:SSLv3/TLS write client hello
SSL_connect:SSLv3/TLS write client hello
SSL_connect:SSLv3/TLS read server hello
SSL_connect:SSLv3/TLS write change cipher spec
SSL_connect:SSLv3/TLS write client hello
SSL_connect:SSLv3/TLS write client hello
SSL_connect:SSLv3/TLS read server hello
SSL_connect:TLSv1.3 read encrypted extensions
SSL_connect:SSLv3/TLS read server certificate request
depth=1 C=JP, O=FOCS, CN=root.focs
verify return:1
depth=0 C=JP, O=FOCS, CN=*.server.focs
verify return:1
SSL_connect:SSLv3/TLS read server certificate
SSL_connect:TLSv1.3 read server certificate verify
SSL_connect:SSLv3/TLS read finished
SSL_connect:SSLv3/TLS write client certificate
SSL_connect:SSLv3/TLS write certificate verify
SSL_connect:SSLv3/TLS write finished
---
Certificate chain
 0 s:C=JP, O=FOCS, CN=*.server.focs
   i:C=JP, O=FOCS, CN=root.focs
   a:PKEY: rsaEncryption, 4096 (bit); sigalg: RSA-SHA256
   v:NotBefore: Dec 12 05:32:18 2023 GMT; NotAfter: Dec 11 05:32:18 2024 GMT
---
Server certificate
-----BEGIN CERTIFICATE-----
MIIFXzCCA0egAwIBAgIUGuwrol3dMdhNRmos5A7Sk6+9CqOwDQYJKoZIhvcNAQEL
```

クライアント証明書の要求がされて**いない**場合、サーバ証明書の検証後（下記 3 行目以降）は以下のようになります。

```
SSL_connect:SSLv3/TLS read server certificate
SSL_connect:TLSv1.3 read server certificate verify
SSL_connect:SSLv3/TLS read finished
SSL_connect:SSLv3/TLS write finished
---
Certificate chain
 0 s:C=JP, O=FOCS, CN=*.server.focs
```

クライアントが使った証明書が不正だった場合 以下の SSL_ERROR が発生し、Java のクライアントからの呼出しの場合は io.grpc.StatusRuntimeException が throw されます。

```
コマンド プロンプト
E1222 11:12:20.172481584      17 ssl_transport_security.cc:1511]      Handshake failed with fatal error
 SSL_ERROR_SSL: error:100000c0:SSL routines:OPENSSL_internal:PEER_DID_NOT_RETURN_A_CERTIFICATE.
```


マイクロサービスによる Java 移行（gRPC,OpenSSL,Docker,Envoy）

7. サーバの多重化と仮想化

マイクロサービスとして切り出す機能によってはリクエストが集中してボトルネックになったり、停止が許されない場合があります。負荷分散と障害耐久性を上げるには多重化が必要ですが、Docker 等のコンテナを使ってサーバを仮想化をすれば環境の複製や移植が簡単にできるようになります。

仮想化や多重化を実現するためのソフトウェアは規格化されているわけではなく明確な切り分けはできませんが、おおよそ、以下の分野に分けられます。

ⅰ. vm イメージ作成、コンテナの生成　… Docker、Containerd、CRI-O 他
ⅱ. コンテナのクラスタリング、管理　… Kubernetes、Docker Swarm、Apache Mesos 他
ⅲ. コンテナ（アプリ）間のネットワーク… Envoy、Nginx、Traefik、HAProxy 他

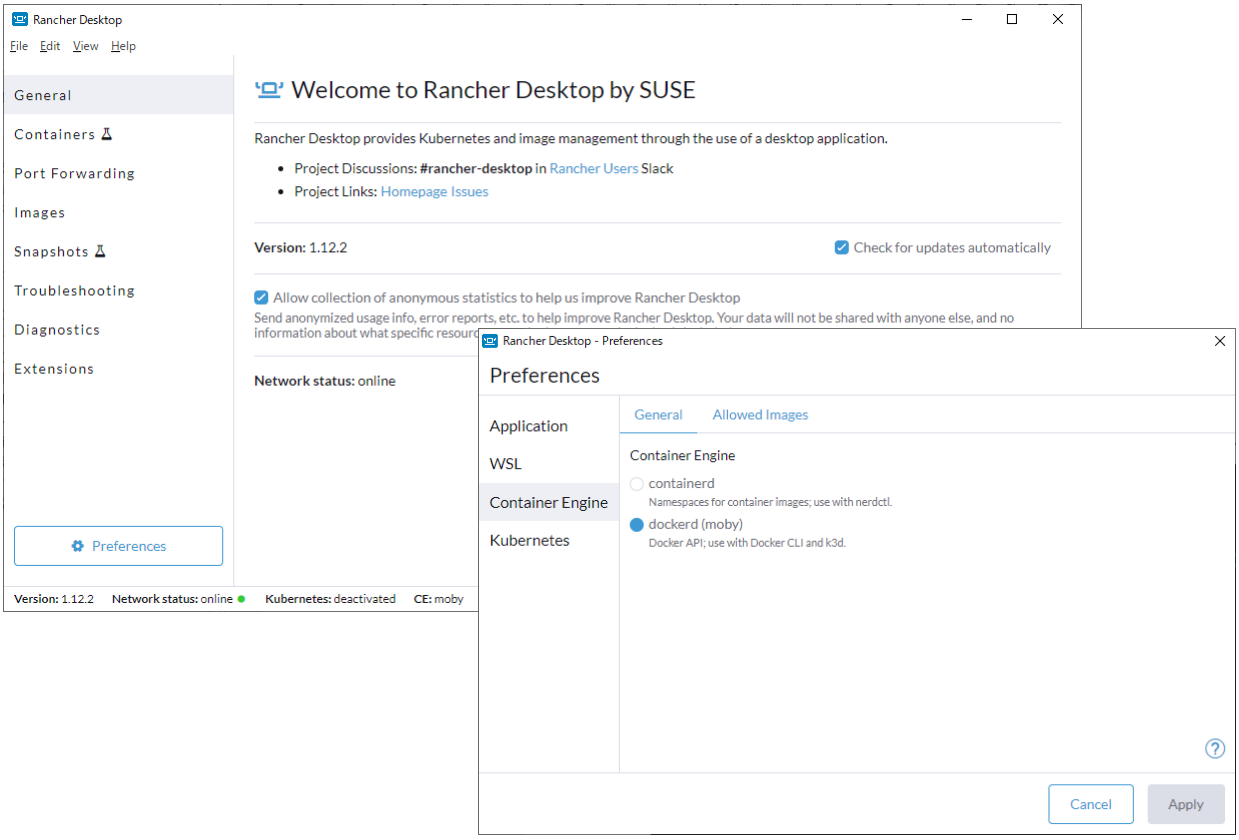
※ 上記の分野に明確な境界はなく、各ソフトは複数の分野に機能を拡張し続ける一方で相互利用もされています。また、開発が活発なので数年後には新たな主力ソフトが出現していたり逆に消えている可能性もあります。例えば、Kubernetes(略称 K8s)を軽量化した K3s や、K3s を更に簡単に使えるようにラップした K3d 等新たなオープンソースのコミュニティ参加者が増加しています

8. サーバの仮想化

サーバの仮想化は、必要なライブラリや資材を取り込んだ設定済のイメージを作っておき、仮想のネットワークや記憶装置（ボリューム）とイメージからコンテナを生成することで実現します。

8.1. マイクロサービスのイメージ作成

Rancher Desktop を使って作成する例を示します。バージョン等は以下のとおりです。



マイクロサービスによる Java 移行（gRPC,OpenSSL,Docker,Envoy）

（1）Docker イメージ

以下の要領で gRPC 実行用ライブラリと Python がインストールされているイメージを作ります。

① ディレクトリ構成

以下のディレクトリ構成を作成します（格納ファイルの内容は後述）。

```
C:¥USERS¥USER¥DESKTOP¥GRPC
 |
 ├──Docker-sv
 |       Dockerfile
 |
 └──Python
     |   denpyo_Journal_TLSserver.py
     |   denpyo_pb2.py
     |   denpyo_pb2_grpc.py
     |
     └──cert
           ca.pem
           server.key
           server.pem
```

② Docker ファイル

[Dockerfile]

```
# image 作成コマンド [docker build -t grpc-python:jp --rm=true .]
FROM python:3.11-slim-bookworm
# gRPC インストール
RUN pip install --upgrade pip ¥
    && pip install grpcio ¥
    && pip install protobuf
# 日本環境に設定
RUN ln -sf /usr/share/zoneinfo/Asia/Tokyo /etc/localtime
ENV LANG ja_JP.utf8

CMD cd /service;./denpyo_Journal_TLSserver.py
```

※上の例のイメージには日本語環境が無いため、実際には日本語や日本時間は表示されません

※最後の CMD はコンテナ起動時に実行され、マウントされている service ディレクトリに移動して denpyo_Journal_TLSserver.py を起動します

③ Python ディレクトリ

[denpyo_Journal_TLSserver.py]

ログをファイル出力するために次の行を追加・変更しています…7、17~22、74~82。

```
1   #!/usr/bin/env python
2   """Denpyo Jurnal gRPC server"""
3
4   from concurrent import futures
5   from socket     import gethostname
6   import logging
7   import sys
8
9   import google.protobuf.text_format as text_format
10  import grpc
11  import denpyo_pb2 # protoc 生成（メッセージ）
```



```
import denpyo_pb2_grpc # protoc 生成（サービス）

class DenpyoJouralingServicer(denpyo_pb2_grpc.DenpyoJournalingServicer):
    # サービス内容を実装
    def Journaling(self, request, context):
        logging.info(
"""<<Journaling Servicer Context>>
    client peer()=%s
    x509_subject =%s
""", context.peer(), context.auth_context().get('x509_subject')
        )
        print("<<Journaling Data Accepted>>")
        # リクエストの型
        print("request type=", type(request))
        # リクエストメッセージを、項目間の改行コードを空白に置換後に表示
        print(text_format.MessageToString(request, as_one_line=True))
        # False の場合 print(request)で出力されないので個別に表示
        if request.err == False: print("err: ", request.err)
        print()

        # ネストしているメッセージの型
        print("request kari type=", type(request.kari))
        # repeated の項目は配列（リスト）のように扱えます
        for ix, val in enumerate(request.kari):
            print("kari val[%d]=%s,%s" % (ix, val.kamokucd, val.kingaku))
        print("-----------")
        for ix, val in enumerate(request.kasi):
            print("kasi val[%d]=%s,%s" % (ix, val.kamokucd, val.kingaku))

        ###
        # 業務処理を既述
        ###

        return denpyo_pb2.Ret(success=True)

def serve():
    server = grpc.server(futures.ThreadPoolExecutor(max_workers=10))
    denpyo_pb2_grpc.add_DenpyoJournalingServicer_to_server(
        DenpyoJouralingServicer(), server
    )
    _PORT = 50051
    ######## TLS(クライアント認証要-mTLS) 追加 start #######
    keyfile  = 'cert/server.key'
    certfile = 'cert/server.pem'
    cacert   = 'cert/ca.pem'
    private_key = open(keyfile, 'rb').read()
    certificate_chain = open(certfile, 'rb').read()
    cacert = open(cacert, 'rb').read()

    credentials = grpc.ssl_server_credentials(
        [(private_key, certificate_chain)]
         ,root_certificates=cacert
         ,require_client_auth=True  #クライアント認証
    )
    server.add_secure_port("[::]:%d" % _PORT, credentials)
    ######## TLS 追加 end #######
    #<TLS:del> server.add_insecure_port("[::]:%d" % _PORT)
    server.start()
    print("server started port=%d" % _PORT)
    server.wait_for_termination()
```



```
72
73      if __name__ == "__main__":
74          hostname = gethostname()
75          logging.basicConfig(
76              handlers=[
77                      logging.FileHandler("service.log"),
78                      logging.StreamHandler(sys.stdout)
79              ]
80            , format="%(asctime)s {}: %(levelname)s %(message)s".format(hostname)
81            , level=logging.INFO
82          )
83          serve()
```

[denpyo_pb2.py, denpyo_pb2_grpc.py]

protoc（py -m grpc_tools.protoc）で生成した資材です。

④ cert ディレクトリ

OpenSSL で生成した以下の資材です。

・認証局のルート証明書（ca.pem）

・サーバの証明書（server.pem）と秘密鍵（server.key）

⑤ イメージ生成

Dockerfile が格納されているディレクトリに移動（cd C:¥Users¥User¥Desktop¥gRPC¥Docker-sv）し、以下のコマンドを実行します。

`docker build -t grpc-python:jp --rm=true .`

※"-t"でイメージに名前を付け、最後の"."でカレントディレクトリの Dockerfile を参照します

`docker images` コマンドで結果を確認

```
C:¥Users¥User¥Desktop¥gRPC¥Docker-sv>docker images
REPOSITORY                                  TAG       IMAGE ID       CREATED          SIZE
grpc-python                                 jp        1a7a1a1665a0   3 minutes ago    174MB
```


8.2. マイクロサービスのコンテナ生成

単独のコンテナを生成する場合は、以下のコマンドで行います。

```
docker run -d ^
 --name=grpc-python_s1 ^
 --hostname=s1 ^
 --mount "type=bind,src=C:¥Users¥User¥Desktop¥gRPC¥Python,dst=/service" ^
 --publish 50051:50051 ^
 --env PYTHONUNBUFFERED=1 ^
 grpc-python:jp
```

--name [任意]コンテナに名前を付けられます。指定しなかった場合、rm コマンド等でコンテナの特定をする場合は CONTAINER ID を使います

--hostname [任意]コンテナ内で hostname コマンドで表示される名前です。

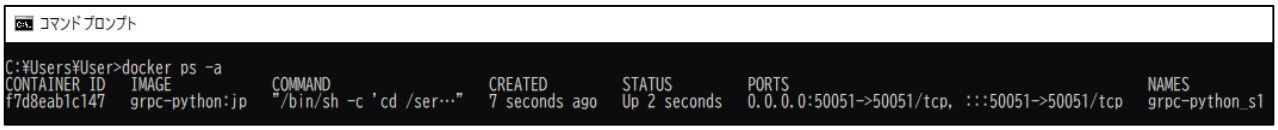
--mount コンテナにマウントするボリュームで -v オプションを使うこともできます（様式は別）

--publish コンテナのポート:ホスト OS のポート で公開するポートを指定します。-p と同

--env 環境変数の指定で、Python の場合は PYTHONUNBUFFERED=1 を指定することにより標準出力を表示することができます（指定しないとバッファに溜まり docker logs でも表示されません）

コンテナの状態は以下のコマンドで確認できます。

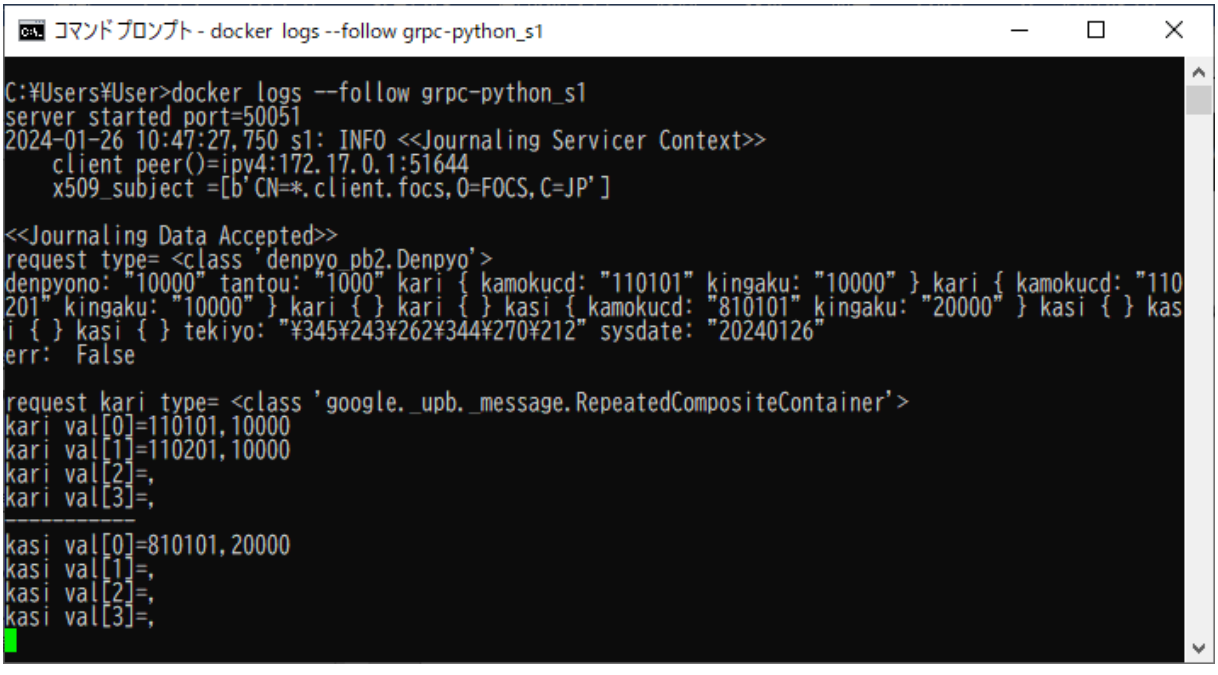
```
docker ps -a
```



8.3. サービスの標準出力やログの確認

コンテナの標準出力は以下のコマンドで見ることができます。

```
docker logs --follow コンテナ名
```



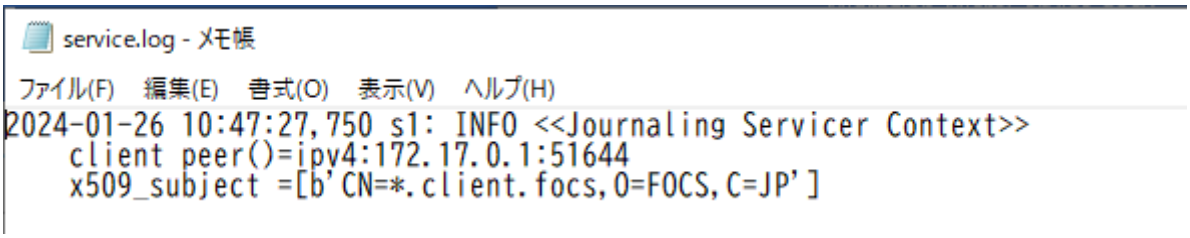
マイクロサービスによる Java 移行（gRPC,OpenSSL,Docker,Envoy）

サーバがコンテナ内で出力するログはサーバのスクリプトが保存されている/service ディレクトリに出力され、ホスト OS 上では service にマウントされている *C:¥Users¥User¥Desktop¥gRPC¥Python* に出力されます。

```
--mount "type=bind,src=C:¥Users¥User¥Desktop¥gRPC¥Python,dst=/service" ^
```

ログの内容は、denpyo_Journal_TLSserver.py 74～82 行目の中で指定している service.log に以下のように出力されます。

```
------------74～82 行目------------
    hostname = gethostname()
    logging.basicConfig(
        handlers=[
                logging.FileHandler("service.log"),
                logging.StreamHandler(sys.stdout)
        ]
      , format="%(asctime)s {}: %(levelname)s %(message)s".format(hostname) # {}にホスト名が埋まります
      , level=logging.INFO
    )
------------17～22 行目------------
        logging.info(
"""<<Journaling Servicer Context>>
    client peer()=%s
    x509_subject =%s
""", context.peer(), context.auth_context().get('x509_subject')
        )
```



マイクロサービスによる Java 移行（gRPC,OpenSSL,Docker,Envoy）

9. サーバの多重化

サーバの多重化は過去には耐障害性の向上と負荷分散を目的としてきましたが、コンテナを前提にしたシステム構成の場合は仮想化と多重化が表裏一体になっています。

9.1. 耐障害性

耐障害性はシステムを構成する CPU/主記憶、補助記憶装置、通信／ネットワーク機器を多重化（冗長化）して確保します。ざっくりした歴史的経緯は以下のようになります。

データセンターやクラウド技術が一般的になる前の多重化は、２台のサーバが仮想 IP アドレスを共用し、待機系からの定期的な生死確認（ハートビート）に応答がなくなると待機系が現用系として活性化するというものです。他に、補助記憶装置はサーバに内臓の HDD をミラーリングや RAID５で冗長化、通信／ネットワーク機器は二重化して２経路作りルータやスイッチを両系に繋いで切替え可能にしておきます。

もっと前にはメインフレームでも論理区画という単位で仮想マシンを構成し仮想マシン間で現用・待機構成を組む場合もありましたが、ここで扱う技術とは異なるものなので割愛します。

クラウド環境が一般的になったのは米国の EC サイト運営会社が自社データセンターの余剰資源を安価[3]に時間貸しをするビジネスを始めてからです。画期的だったのは CPU/主記憶、補助記憶を仮想化してプールとして扱ったことで、利用者が好きな OS、CPU（vCPU)、補助記憶（ディスク）を必要な時に必要な数や量 仮想ネットワークに接続した状態で使えるようになりました。これにより CPU/主記憶の故障や内部ネットワークの故障は考えなくてよくなりました。但し、データセンターの設置場所（国）や秘密情報を社外で保管することがセキュリティ上のリスクと認識されるようになってきているので、全てをクラウド化できるわけでもありません。

社内環境でも Docker や Kubernetes 等のソフトウェアを使えば上記クラウド環境と同様に仮想化による CPU/主記憶とネットワークの耐障害性を得られます。（但し、仮想化ソフトウェアを稼働させるホストやストレージシステム、仮想ネットワークの外のネットワークに関しては従前どおりの障害対策が必要になります…この資料では割愛）

9.2. 負荷分散

負荷分散はリクエストの量がサーバ単体の処理能力を超えている場合や、サービスを継続しながら一部のサーバを停止する場合に、サービスのコンテナ数を増加・減少させて負荷分散を行います。

サービスを行う複数のコンテナの集団をクラスタと呼びますが、クラスタの数や規模の増減に求められる柔軟性により サーバの多重化と仮想化 に必要なソフトウェアの組合せが異なります。

10. クラスタと逆プロキシの構成例

gRPC を基盤にしたマイクロサービスをクラスタで運用する場合、gRPC が扱えて複数のコンテナにリクエストの振り分け（負荷分散）ができる逆プロキシを設置します。これによりクライアントが個別のコンテナを意識する必要がなくなり、クラスタの柔軟な拡縮・縮退が可能になります。

---

[3] 現在の価格については EC2、S3 等で検索してみてください

マイクロサービスによる Java 移行（gRPC,OpenSSL,Docker,Envoy）

10.1. 逆プロキシ-Envoy

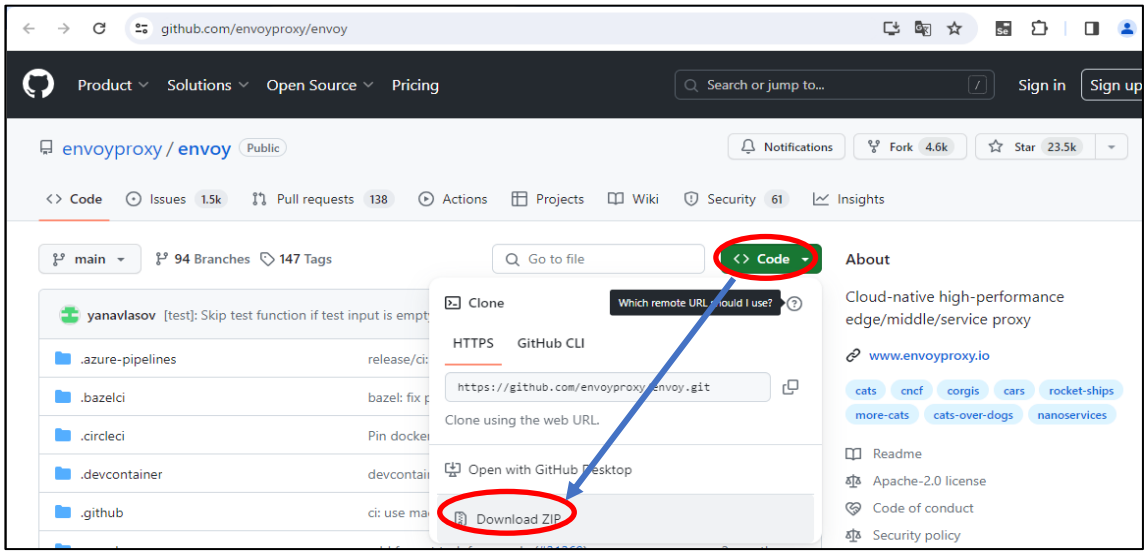
gRPC の負荷分散を行えるソフトウェアは幾つかあります[4]が、逆プロキシとして使えて比較的新しく、企業色が薄いのが Envoy[5]（著作権者: Envoy Project Authors.,ライセンス Apache-2.0）です。

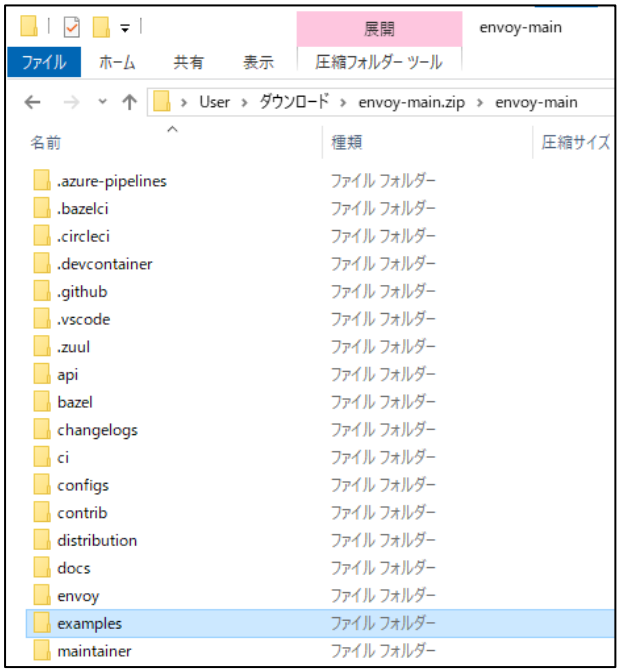
TLS の相互認証（mTLS: Mutual Transport Layer Security）を使った接続や docker compose を使った環境設定のサンプルが提供されています。

10.2. Envoy のセットアップ

Envoy は Docker のイメージとして公開されていて、GitHub に登録されている examples から目的に合った事例を選んで手を加え利用することができます。

Git でリポジトリのクローンを作るか、リポジトリの zip ファイルをダウンロードします。





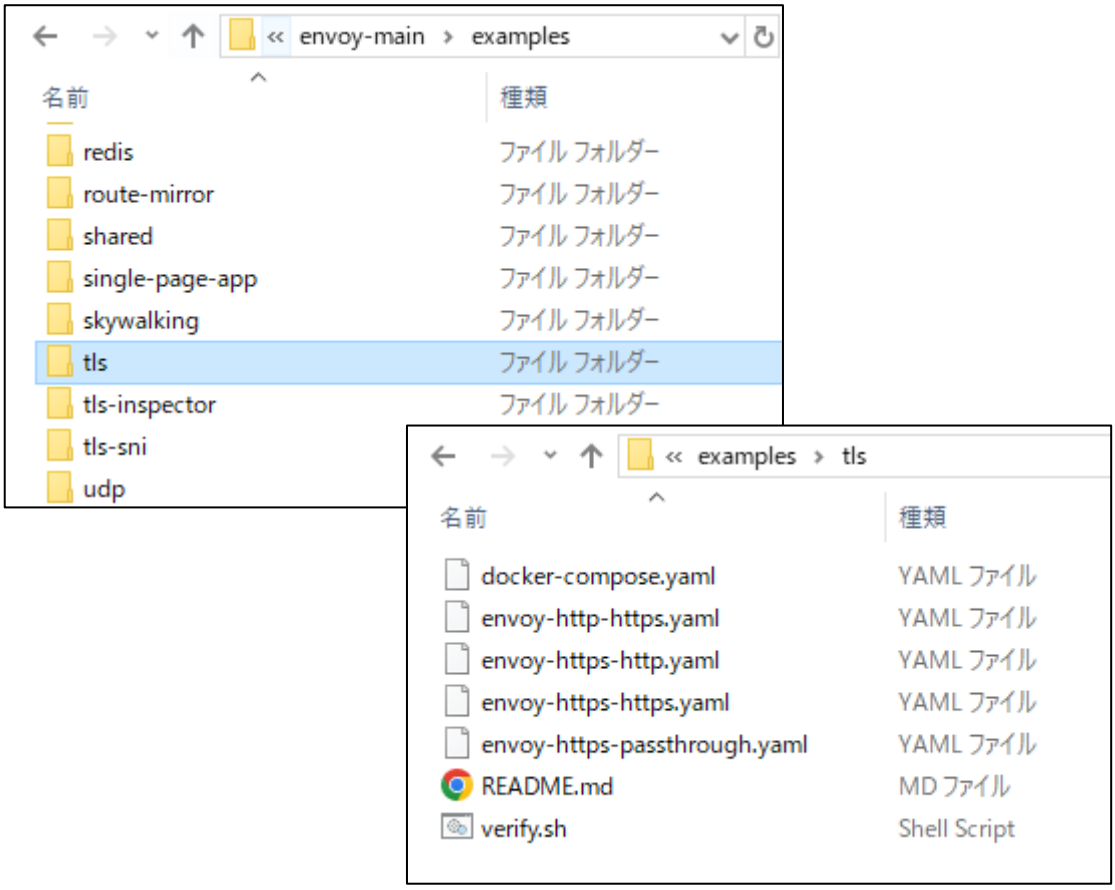
ダウンロードした zip ファイルの中身
⇒ examples を使います。

[4] gRPC ロード バランシング https://grpc.io/blog/grpc-load-balancing/#recommendations-and-best-practices

[5] Envoy GitHub リポジトリ https://github.com/envoyproxy/envoy

10.3. 逆プロキシの構成

TLS 相互認証（mTLS）を行う環境は exmples の中の tls にサンプルが入っています。また、その説明は README.md（md:マークダウン記法の文書…Chrome 等用の機能拡張があります）に書かれている URL を参照してください。以下では、サーバ／クライアントが TLS の認証を自前で行う https パススルーの構成を作ります。



マイクロサービスによる Java 移行（gRPC,OpenSSL,Docker,Envoy）

10.4. 構成方法

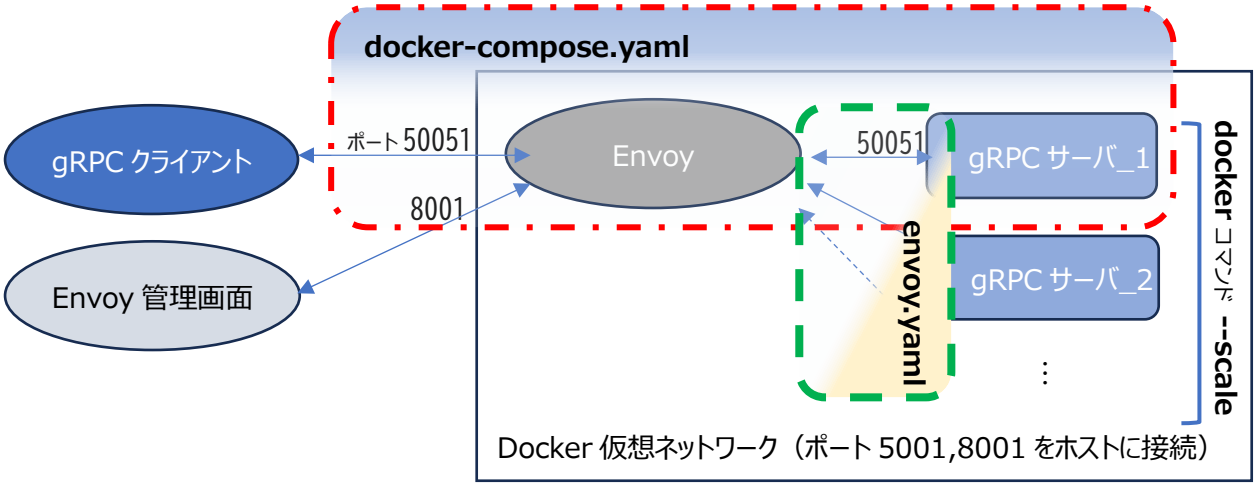
基本は Envoy の examples/tls ディレクトリに格納されている以下のファイルを修正して使います。

① docker-compose.yaml

② envoy-https-passthrough.yaml（必要な部分を切り出して envoy.yaml を作成）

③ envoy-https-passthrough.yaml から参照している examples¥shared¥envoy¥Dockerfile

＜構成イメージ＞



（1） ファイル構成

```
C:¥Users¥User>tree /f C:¥Users¥User¥Desktop¥gRPC¥envoy
C:¥USERS¥USER¥DESKTOP¥GRPC¥ENVOY
|   docker-compose.yaml
|   Dockerfile
|   envoy.yaml
```

・ディレクトリ名がコンテナの名前の一部になります。以降の例は envoy というディレクトリを作り、配下に必要な全ての構成ファイルをコピーしています

・examples に入っている構成ファイルは複数のサンプルが混在しているので、envoy ディレクトリへコピー後に使わない部分を削除します。これに伴い Dockerfile の中で envoy.yaml にコピーしている envoy-https-passthrough.yaml を例中ではデフォルトの名前 envoy.yaml に変えています

（2） ファイルの用途や参照関係

docker-compose.yaml で Envoy の構成（target オプションで選択）と gRPC サーバのイメージ、環境変数、使用ポート等を定義します。

examples の Dockerfile は target オプションで事例に合わせた初期設定や処理を記述したイメージが作られるようにしていますが、必要な部分だけ残して docker-compose.yaml から参照します。

envoy.yaml は Envoy の起動用時に読み込む構成定義ファイルで、Dockerfile 内の Envoy を起動するコマンドのパラメータとして書いておきます。

マイクロサービスによる Java 移行（gRPC,OpenSSL,Docker,Envoy）

<docker-compose.yaml>

```
-----------------------------------------------
#すべての受信リクエストは、リバース プロキシ proxy-https-passthrough 経由でルーティング
#Port 8001 は、envoy の admin（管理サーバ）、50051 は gRPC サーバへそれぞれ転送します。

services:

  proxy-https-passthrough:
    build:
      context: .
      dockerfile: ./Dockerfile
    depends_on:
      - service1
    ports:
    - "${PORT_STATS:-8001}:8001"
    - "${PORT_GRPC:-50051}:50051"

  service1:
    image: grpc-python:jp
    environment:
      - PYTHONUNBUFFERED=1
    volumes:
      - C:¥Users¥User¥Desktop¥gRPC¥Python:/service
----------------ファイル終わり--------------
```

<Dockerfile>

※コンテナ生成時に実行される CMD [/usr/local/bin/envoy]のパラメータに--log-level を追加しておくと envoy.yaml の誤り等でプロキシが想定通りに動作しない場合の情報収集ができます

※examples にの Dockerfile には複数の FROM 命令が書かれ、as で名前が付けられています。名前は後続の FROM 命令や、イメージを作成する際に build target パラメータで参照します

```
-----------------------------------------------
ARG ENVOY_IMAGE="${ENVOY_IMAGE:-envoyproxy/envoy}"
ARG ENVOY_VARIANT="${ENVOY_VARIANT:-dev}"

FROM ${ENVOY_IMAGE}:${ENVOY_VARIANT} as envoy-base
# ARG コマンドで ENVOY_CONFIG が呼び元から与えられなかった場合の値を設定
ARG ENVOY_CONFIG=envoy.yaml
ENV ENVOY_CONFIG="$ENVOY_CONFIG"
ENV DEBIAN_FRONTEND=noninteractive
RUN --mount=type=cache,target=/var/cache/apt,sharing=locked ¥
    --mount=type=cache,target=/var/lib/apt/lists,sharing=locked ¥
    rm -f /etc/apt/apt.conf.d/docker-clean ¥
    && echo 'Binary::apt::APT::Keep-Downloaded-Packages "true";' | tee /etc/apt/apt.conf.d/keep-cache ¥
    && apt-get -qq update -y ¥
    && apt-get -qq install --no-install-recommends -y curl
COPY --chmod=777 "$ENVOY_CONFIG" /etc/envoy.yaml
#envoy 起動時に loglevel の指定追加
CMD ["/usr/local/bin/envoy", "-c", "/etc/envoy.yaml", "--log-level", "trace"]

FROM envoy-base
----------------ファイル終わり--------------
```


マイクロサービスによる Java 移行（gRPC,OpenSSL,Docker,Envoy）

<envoy.yaml>

```
-------------------------------------------------
#<envoy.yaml> Docker の image envoy-base の/etc/envoy.yaml にコピーし envoy が参照する。
#プロキシ形態 https パススルー：HTTPS を行うアップストリーム サービスに直接プロキシする。

#envoy-proxy-https-passthrough が起動しない場合は、以下を実行して envoy.yaml を検証する。
# docker run -it envoy-proxy-https-passthrough bash
# envoy -c /etc/envoy.yaml

static_resources:
  listeners:
  - address:
      socket_address:
        address: 0.0.0.0
        port_value: 50051
    filter_chains:
    - filters:
      - name: envoy.filters.network.tcp_proxy
        typed_config:
          "@type": type.googleapis.com/envoy.extensions.filters.network.tcp_proxy.v3.TcpProxy
          cluster: service1
          stat_prefix: https_passthrough

  clusters:
  - name: service1
    type: STRICT_DNS
    lb_policy: ROUND_ROBIN
    load_assignment:
      cluster_name: service1
      endpoints:
      - lb_endpoints:
        - endpoint:
            address:
              socket_address:
                address: service1
                port_value: 50051

admin:
  address:
    socket_address:
      address: 0.0.0.0
      port_value: 8001
layered_runtime:
  layers:
  - name: static_layer_0
    static_layer:
      envoy:
        resource_limits:
          listener:
            example_listener_name:
              connection_limit: 10000
----------------ファイル終わり--------------
```

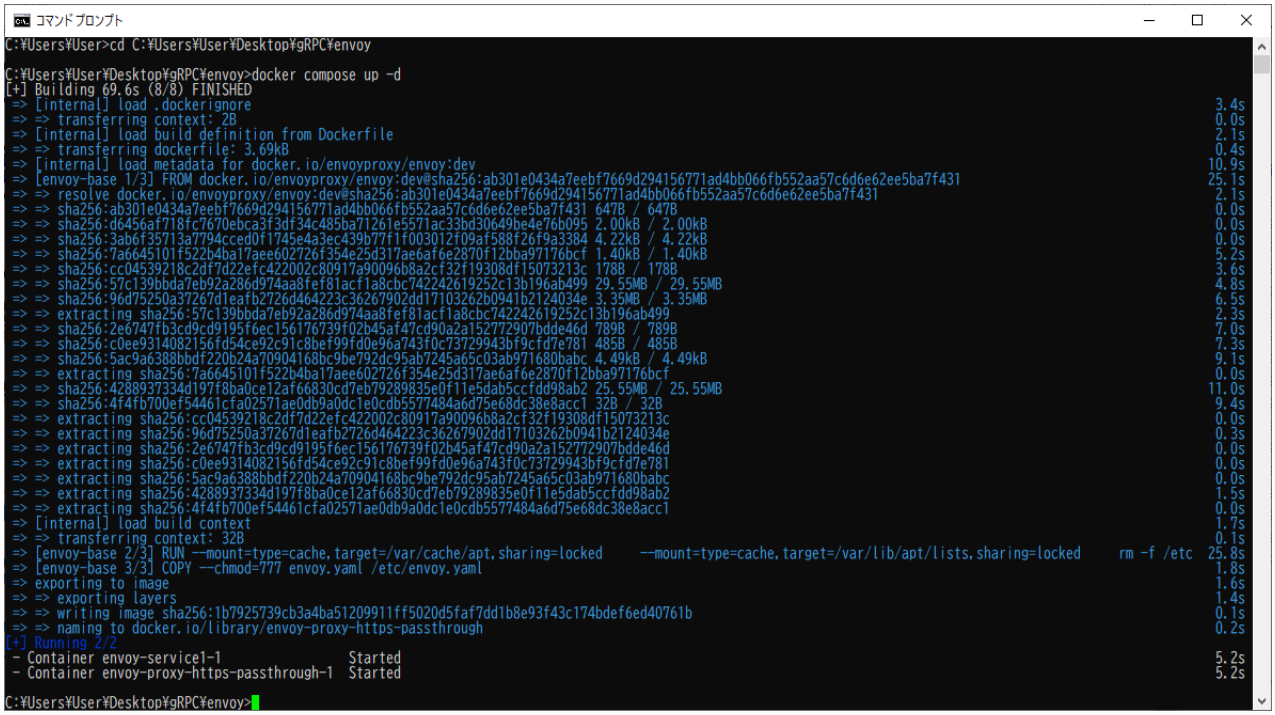

10.5. コンテナの生成 docker compose up

docker-compose.yaml ファイルが入っているディレクトリに移動し up コマンドを実行します。

＜実行例＞

cd C:¥Users¥User¥Desktop¥gRPC¥envoy

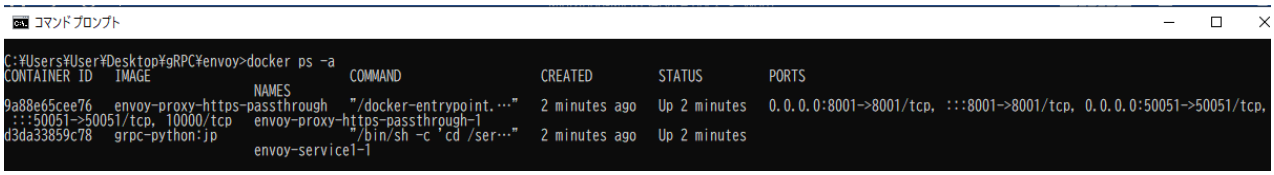
C:¥Users¥User¥Desktop¥gRPC¥envoy>docker compose up -d



（1）正常な起動（STATUS : Started）

全てのコンテナに Started が表示されたらイメージの構築とコンテナ化が完成で、エラーが無ければ docker ps で表示される STATUS 列が Up になります。

docker ps -a

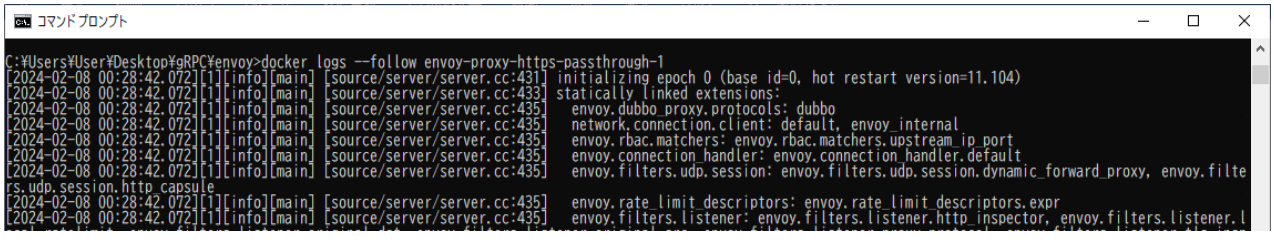


（2）Started にならない場合

Dockerfile に書いた CMD が終わってしまうとコンテナが停止し STATUS 列が Exited になります。コンテナ内で出力されているログを docker logs で確認してください。

docker logs --follow *envoy-proxy-https-passthrough-1*

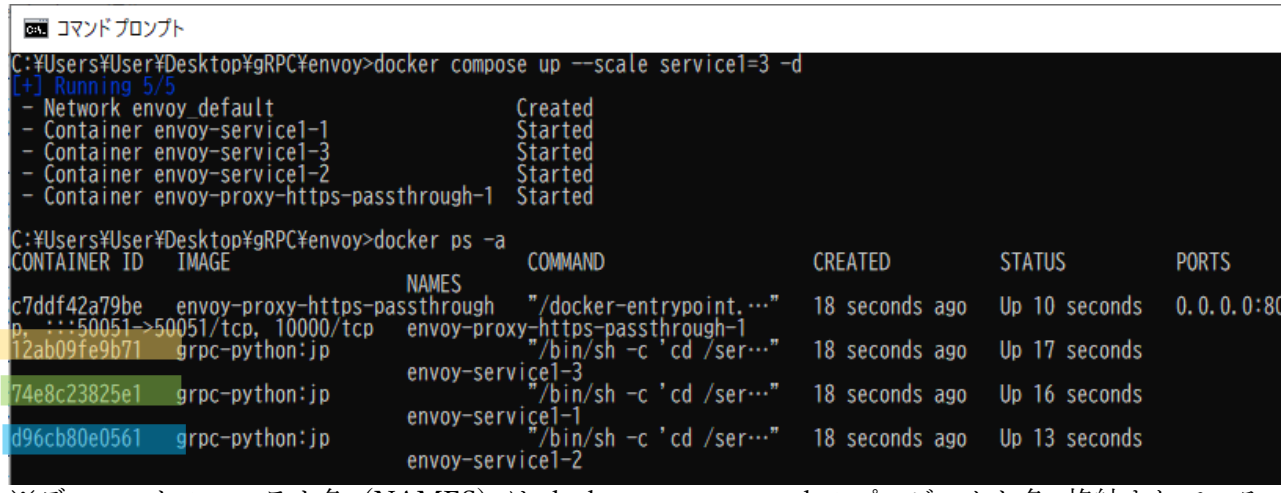
ctrl + C 等で打ち切るまでログの出力を待ち受けます。



10.6. 複数コンテナの実行（多重度の変更）

docker-compose.yaml に定義されているサービスを指定し、scale で多重度を変更することができます。service1 の多重度を 3 にする場合は以下のコマンドを実行します。

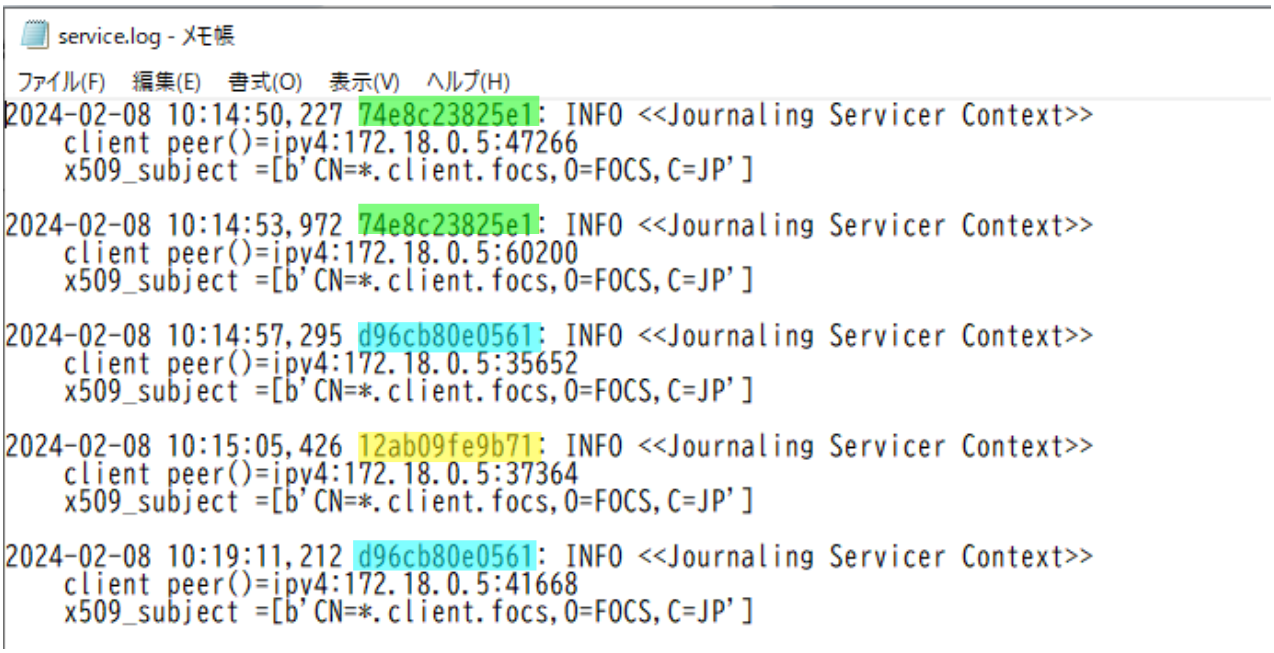
docker compose up --scale service1=3



※デフォルトのコンテナ名（NAMES）は docker-compose.yaml のプロジェクト名=格納されているディレクトリ名が付き、ディレクトリ名-サービス名-インデックス（1~n）になります

＜リクエストの振分＞

以下 envoy-service1 に対して 5 回リクエストしたログ。蛍光ペンを塗った部分がサーバのホスト名で、デフォルトは CONTAINER ID がホスト名になります

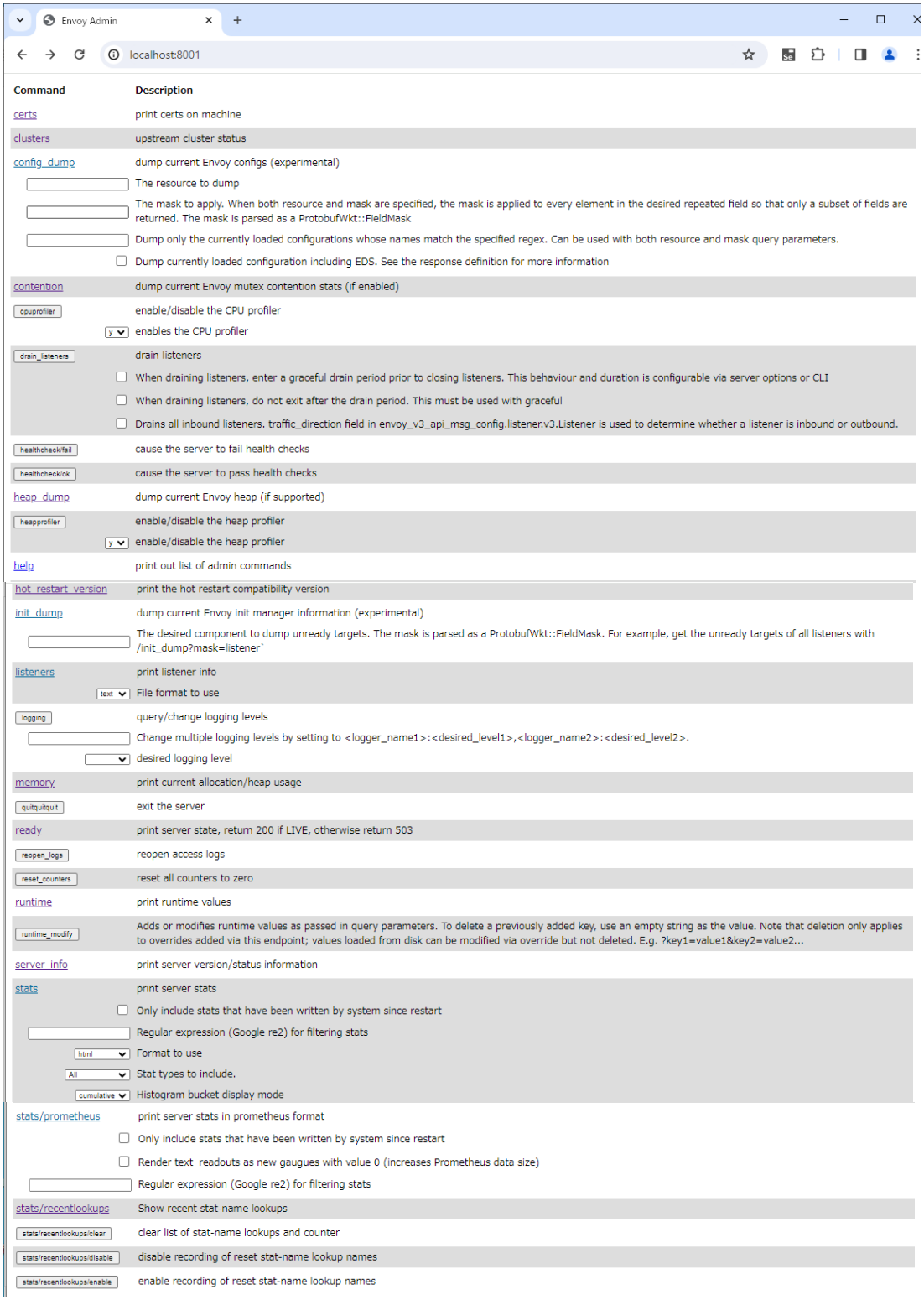


※上記実行例ではロードバランスの方式にラウンドロビン（重みづけ無し）を選んでいますが、Docker version 24.0.7-rd, build 72ffacf と Envoy の 2 月開発版（envoy:dev）で実行するとコンテナへのリクエストの振分はランダムになっているように見えます

マイクロサービスによる Java 移行（gRPC,OpenSSL,Docker,Envoy）

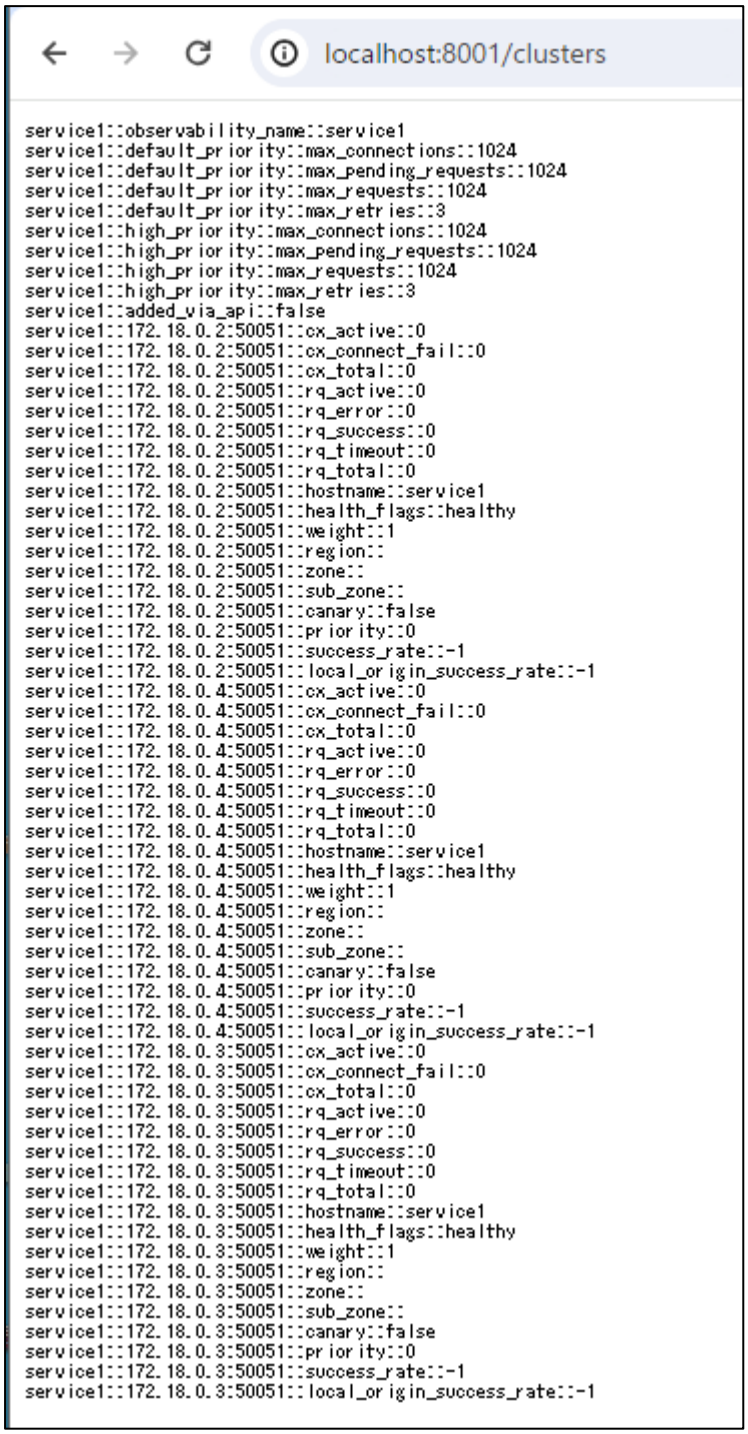
10.7. Envoy の管理画面

envoy.yaml の admin: > port_value: 8001 で指定したポート番号を通して Envoy の管理画面を表示することができます。主にクラスタの稼働監視用の機能ですが、Envoy を停止させる機能もあります。

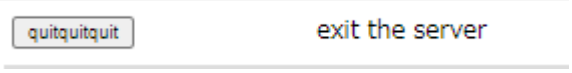


マイクロサービスによる Java 移行（gRPC,OpenSSL,Docker,Envoy）

clusters のリンク…以下の例では service1 が 172.18.0.2、172.18.0.3、172.18.0.4 でクラスタを構成し、ポート 50051 で待ち受けていることが分かります。



exit the server …プロキシが停止します。



＜quitquitquit ボタン押下後＞

docker ps -a コマンドにより逆プロキシ envoy-proxy-https-passthrough-1 のコンテナが停止〔STATUS : Exited (0) nn xxxxx ago〕していることが確認できます。

```
C:¥Users¥User>docker ps -a
CONTAINER ID   IMAGE                              COMMAND                  CREATED          STATUS                       PORTS     NAMES
e413b468cbc0   envoy-proxy-https-passthrough      "/docker-entrypoint.…"   56 minutes ago   Exited (0) 15 seconds ago              envoy-proxy-https-passthrough-1
2c0cf64d9e01   grpc-python:jp                     "/bin/sh -c 'cd /ser…"   56 minutes ago   Up 56 minutes                          envoy-service1-1
bf524096d21a   grpc-python:jp                     "/bin/sh -c 'cd /ser…"   56 minutes ago   Up 56 minutes                          envoy-service1-3
1f33f5bfa331   grpc-python:jp                     "/bin/sh -c 'cd /ser…"   56 minutes ago   Up 56 minutes                          envoy-service1-2
```

＜コンテナの再開＞

docker restart *コンテナ名*　コマンドでコンテナが再開できます。

```
C:¥Users¥User>docker restart envoy-proxy-https-passthrough-1
envoy-proxy-https-passthrough-1

C:¥Users¥User>docker ps -a
CONTAINER ID   IMAGE                              COMMAND                  CREATED          STATUS          PORTS
NAMES
e413b468cbc0   envoy-proxy-https-passthrough      "/docker-entrypoint.…"   57 minutes ago   Up 8 seconds    0.0.0.0:8001->8001/tcp, :::8001->8001/tcp, 0.0.0.0:50051->500
51/tcp, :::50051->50051/tcp, 10000/tcp   envoy-proxy-https-passthrough-1
2c0cf64d9e01   grpc-python:jp                     "/bin/sh -c 'cd /ser…"   57 minutes ago   Up 57 minutes
envoy-service1-1
bf524096d21a   grpc-python:jp                     "/bin/sh -c 'cd /ser…"   57 minutes ago   Up 57 minutes
envoy-service1-3
1f33f5bfa331   grpc-python:jp                     "/bin/sh -c 'cd /ser…"   57 minutes ago   Up 57 minutes
envoy-service1-2
```

11. 運用環境について

前項までの説明はアプリケーションの開発と試験を念頭にしたものです。運用環境を作る場合は、以下の点が課題になります。

・Docker の複数コンテナに Windows のディレクトリをマウントしてログ書き出しましたが、性能面とログファイルの競合が問題になります（IO 競合に関する検証はしていません）
　⇒ログサーバを別に用意するか、Volume イメージを使った方がよい

・複数マシン／ホスト OS を跨るコンテナによる大規模システムを構築したい場合は、Kubernetes（K3s 等の派生含む）や Swarm 等のコンテナ・クラスタを管理（コンテナオーケストレーション）できるソフトか Docker のマルチホスト・ネットワーク機能を使う必要があります

・多種のマイクロサービルを開発する場合はポート番号の割り当てに関する設計が必要になります。マイクロサービス毎にポート番号やホストを割り振るのは管理も運用も煩雑なので、サービス名による振分（ルーティング／サービスメッシュ）を逆プロキシで構成する方法があります

以上

