

記事

[Toshihiko Minamoto](#) · 2022年7月12日 21m read

gRPC と Hello World

はじめに

この記事では、gRPC とは何か、そして IRIS 組み込み Python を使って公式の Hello World を実行する例を紹介します。

ここで紹介するすべてのコードは、こちらの[プロジェクトリポジトリ](#)にあります。

gRPC

[gRPC \(gRPC リモートプロシージャールール\)](#) は、RPC プロトコルに基づく API アーキテクチャ式です。このプロジェクトは、2015 年に Google によって作成され、Apache 2.0 の下にライセンス供与されています。現在、プロジェクトは、[Cloud Native Computing Foundation \(CNCF\)](#) によってサポートされています。

マイクロサービス式のアーキテクチャのサービスなど、バックエンド間でサービスを接続することで使用することができます。

プロトコルバッファ

ほとんどの RPC ベースのプロトコルは、IDL（インターフェース記述言語）を使用して、サーバーとクライアント間の通信コントラクトを定義します。

gRPC は、プロトコルバッファという、シリアライザーメカニズムフォーマットを使用します。

このフォーマットの目的は、メソッドとデータ構造を定義できるという点で、WSDL に似ています。ただし、XML を使って定義される WSDL とは異なり、プロトコルバッファは非常に一般的な言語を混ぜた言語（プロトコルバッファ言語）を使用します。

たとえば、情報を補間するメッセージを定義する場合、以下のプロトコルバッファ定義を使用できます。

```
message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;
}
```

また、メッセージのサービスメソッドコントラクトも定義できます。例:

```
// greeter ????????
service Greeter {
  // ???????
  rpc SayHello (HelloRequest) returns (HelloReply) {}
}
```

```
// ??????????????????????  
message HelloRequest {  
    string name = 1;  
}
```

```
// ??????????????????????  
message HelloReply {  
    string message = 1;  
}
```

gRPC でプロトコルバッファを使用すると、リソースベースの設計原則ではなく、REST が使用する関数ベースの原則に従うことになります。

gRPC tools

プロトコルバッファ言語で定義するものは、直接使用できません。プロトコルバッファ言語を、gRPC がサポートする別の言語にトランスパイルする必要があります。

そのようなトランスパイルは、gRPC tools というパッケージで行われます。現在、gRPC プラットフォームでは、Java、C++、Dart、Python、Objective-C、C#、Ruby、JavaScript、および Go といった言語がサポートされています。

この記事では、Python サポートを使って、IRIS の組み込み Python 機能で gRPC を使用することにします。

たとえば、gRPC tools の以下のコマンドを使うと、Greeter サービス、HelloRequest、および HelloReply メッセージのプロトコルバッファ定義を Python にトランスパイルできます。

```
python3 -m grpc_tools.protoc -I ../.. /protos --python_out=. --grpc_python_out=. ../.. /protos/helloworld.proto
```

上記のコマンドによって、以下の Python ファイルが生成されます。

```
-rwxrwxrwx 1 irisowner irisowner 2161 Mar 13 20:01 helloworld_pb2.py*  
-rwxrwxrwx 1 irisowner irisowner 3802 Mar 13 20:01 helloworld_pb2_grpc.py*
```

これらのファイルはそれぞれ、proto ファイルから生成されたメッセージとサービスメソッドの Python ソースコードです。サーバーがこのサービスメソッドを実装すると、クライアント（スタブとも呼ばれる）がそのメソッドを呼び出します。

したがって、組み込み Python を使用して、Greeter サービスを通じて Hello メッセージを送受信できます。

gRPC に使用できる便利なツールには、curl に相当する [grpcurl ユーティリティ](#)もあります。Hello World の例を紹介した後に、このツールの使用方法をおおまかに説明します。

サービスメソッドタイプ

クライアントは、サービスメソッドからメッセージをどのように送受信するかに応じて異なります。メッセージは、呼び出しごとに 1 つずつ、またはストリームで受信できます。gRPC には、組み合わせごとに、次のサービスメソッドタイプがあります。

- 単純な RPC または単項: クライアントは単純なメッセージを送信し、サーバーから単純なレスポンスを受

信します。標準的な関数呼び出しです。

- レスポンスストリーミングまたはサーバーストリーミング:
クライアントは単純なメッセージを送信し、サーバーからメッセージのストリームを受信します。
- リクエストストリーミングまたはクライアントストリーミング:
クライアントはメッセージのストリームを送信し、サーバーから単純なメッセージを受信します。
- 双方向ストリーミング: クライアントはメッセージのストリームを送信し、サーバーから別のメッセージストリームを受信します。

このようなメソッドによる通信は、クライアントのニーズに応じて、非同期（デフォルト）の場合と同期の場合があります。

[gRPC の基本概念ページ](#)には、これらのタイプが gRPC ライフサイクルの機能として定義されています。他の機能もハイライトされていますが、それらはこの記事で説明する内容ではないため、詳しくは以下のリンクをご覧ください。

- [期限/タイムアウト](#)
- [RPC の終了](#)
- [RPC の取り消し](#)
- [メタデータ](#)
- [チャンネル](#)

メリットとデメリット

他の記事で見つけたメリットとデメリットを以下に示します。

メリット:

- メッセージがより軽量であること。
プロトコルバッファはバイナリ形式であるため、特殊文字によって生じる JSON オーバーヘッドが回避されます。
- シリアル化/逆シリアル化が高速であること。やはりバイナリ形式であるため、プロトコルバッファは、インタープリタを使用せずに固有の言語を使ってクライアントスタブにシリアル化/逆シリアル化することができます。
- クライアント（スタブ）が組み込まれていること。OpenAPI
やクライアントジェネレーターといったサードパーティツールに依存する JSON とは異なり、プロトコルバッファには、最も多く使用されている言語のジェネレーターが組み込まれています。
- 並列リクエスト。HTTP/1 では、最大 6 つの同時接続が許可されており、全 6 つの接続が終了するまで他のリクエストをブロックしてしまいます。これは HOL（ヘッドオブライン）ブロッキングとして知られる問題で、HTTP/2 ではその制約が修正されています。
- コントラクト優先型の API 設計である。REST API は、Open API などのサードパーティツールを通じてコントラクトを公開できますが、gRPC では、コントラクトはプロトコルバッファによって明示的に宣言されます。
- ネイティブのストリーミングである。HTTP/2 のストリーミング機能により、gRPC では、ネイティブの双方向ストリーミングモデルを実現できます。HTTP/1 を介してその動作を真似る必要のある REST とは異なります。

デメリット:

- プロトコルバッファには柔軟性がない（サーバーの結合度が緩い）。
- プロトコルバッファを人が読み取れない。
- REST/JSON
の専門家、リソース、およびツール/プロジェクトの数は、gRPC/プロトコルバッファよりはるかに多い。

ただし、上記のメリットとデメリットは、総意というわけではありません。
アプリケーションのニーズによって大きく異なります。

たとえば、REST/JSON にデメリットがあるとすれば、それは OpenAPI といったサードパーティツールが必要であるということです。こういったツールは世界中のさまざまな開発コミュニティや企業で重々にサポート、保守、使用されているため、まったく問題ではないでしょう。

一方で、プロジェクトで、gRPC が REST よりもうまく対処できる複雑な問題に対応する必要がある場合は、gRPC を選択したためにスキルを備えた開発者チームを構成する必要があるなど、さらに悩みが増える場合であっても、gRPC を選択することをお勧めします。

いつどこで gRPC を使用するのか

以下に、gRPC が必要となるユースケースをいくつか示します。

- マイクロサービス通信
- HTTP/2 が利用できるとした場合に、制約のあるハードウェアやネットワークでクライアントが実行しているクライアント/サーバーアプリケーション
- 強力なコントラクト API 設計で提供される相互運用性

gRPC が使用されている製品

ビッグテック企業数社では、特定の課題に対応するために gRPC を活用しています。

- [Salesforce](#): 同社のプラットフォームでは gRPC を採用し、プロトコルバッファが提供する強力なコントラクト設計によって、相互運用性の堅牢性を向上させています。
- [Netflix](#): gRPC を使用して、マイクロサービス環境を改善しています。
- [Spotify](#): Netflix と同様に、gRPC を使用して、マイクロサービスの課題に取り組み、多数の API に対処しています。

組み込み Python を使った Hello World

gRPC とは何か、そしてそれが何を行うかについて簡単に説明したので、それを管理できるようになったと思います。では、実際に使用してみましょう。この記事のタイトルのとおり、これは、元の Hello World を IRIS 組み込み Python で使用できるようにしたサンプルです。

実際のところ、このサンプルは helloworld と helloworldstreamingworld という他の 2 つのインスタンスを変更したものです。これらは、gRPC サンプルリポジトリにあります。その助けを借りて、単純なメッセージを単一モードとストリームモードで送受信する方法をお見せしたいと思います。特に便利な機能が備わっているわけでもなく、単純なサンプルではありますが、gRPC アプリケーションの開発に関わる主な概念を理解する上で役立つでしょう。

Python で使用する gRPC のインストール

まず、gRPC を Python で使用するために必要なパッケージをインストールしましょう。私の [GitHub プロジェクト](#)

に定義されたコンテナからこのサンプルを実行している場合は、すでにこれらのパッケージがインストールされているでしょう。その場合はこのステップを飛ばしてください。

```
python3 -m pip install --upgrade pip
python3 -m pip install --upgrade --target /usr/irissys/mgr/python grpcio grpcio-tools
```

サービスコントラクトの定義

次に、サービスコントラクト、つまりプロトコルバッファファイル（または略して protobuf）と、メッセージのスキーマと利用可能なメソッドを確認しましょう。

```
syntax = "proto3";

package helloworld;

// ??????????
service MultiGreeter {
    // ??????
    rpc SayHello (HelloRequest) returns (HelloReply) {}
    // ??????????
    rpc SayHelloStream (HelloRequest) returns (stream HelloReply) {}
}

// ??????????????????????????????
// ???????
message HelloRequest {
    string name = 1;
    Int32 num_greetings = 2;
}

// ??? 1 ??????????????????
message HelloReply {
    string message = 1;
}
```

この protobuf ファイルは、MultiGreeter というサービスを SayHello() と SayHelloStream() という 2 つの RPC メソッドで定義しています。

SayHello() メソッドは HelloRequest メッセージを受信すると、HelloReply メッセージを送信します。同様に、SayHelloStream() は同じメッセージを受信して送信しますが、1 つのメッセージではなく、HelloRequest メッセージのストリームを送信します。

サービス定義の後には、HelloRequest と HelloReply というメッセージの定義があります。HelloRequest メッセージは、name という文字列と numgreetings という整数の 2 つのフィールドをカプセル化しているだけです。HelloReply メッセージには、message という文字列フィールドのみが 1 つ含まれています。

フィールドの後の数字は、フィールド番号です。これらの番号は識別子として機能するため、メッセージが使用されたら変更することはできません。

サービスコントラクトから Python コードを生成する

おそらく気づいたかもしれませんが、protobuf 定義にはコードを記述する必要はなく、インターフェースのみを記述する必要があります。さまざまなプログラミング言語に使用できるコードを実装するタスクは、protobuf コンパイラである protoc によって行われます。gRPC がサポートする言語ごとに、1 つの protoc コンパイラがあります。

Python の場合、コンパイラは `grpc_tools.protoc` というモジュールとしてデプロイされます。

protobuf 定義を Python コードにコンパイルするには、以下のコマンドを実行します（[私のプロジェクト](#)を使用している場合）。

```
cd /irisrun/repo/jrpereira/python/grpc-test
```

```
/usr/irissys/bin/irispython -m grpc_tools.protoc -I ./ --python_out ./ --grpc_python_out ./ helloworld.proto
```

上記のコマンドは、`grpc_tools.protoc` モジュールを呼び出します。これは Python 用の `protoc` コンパイラーで、以下のパラメーターを使用します。

- `helloworld.proto`: サービスコントラクト用のメインの `.proto` ファイル
- `-I`: コンパイラーが依存関係を探す `.proto` ファイルの場所
- `--python_out`: メッセージ用に生成された Python コードの場所
- `--grpcpython_out`: サービス定義の RPC メソッドに応じてサーバーとスタブ（クライアント）用に生成された Python コードの場所

この場合、これらすべての場所パラメーターは、現在のディレクトリにセットアップされます。

`protoc` コンパイラーが生成するコードは、理解できないわけではありませんが、可読性に問題があります。これは、`protoc` コンパイラーに渡されたディレクトリで確認できます。

それはさておき、これらのファイルはユーザー独自のコードにインポートされることを意図しているため、実際に使用して、サーバーとクライアントを実装してみましょう。

サービス用のサーバーを実装する

上記で定義したサービスのサーバーを実装するために、組み込み Python を使用しましょう。

まず、サーバーロジックが実装されている Python ファイルを使って、サーバーを定義しましょう。私の場合は、Python 並列ライブラリを使用する必要があるため、以下のように実装しました。

```
"""
GRPC helloworld.Greeter ????? Python ???
???:
- https://github.com/grpc/grpc/blob/master/examples/python/helloworld/async_greeter_server.py
- https://github.com/grpc/grpc/blob/master/examples/python/hellostreamingworld/async_greeter_server.py
- https://groups.google.com/g/grpc-io/c/6Yi_oIQsh3w
"""

from concurrent import futures
import logging
import signal
from typing import Any
from argparse import ArgumentParser, ArgumentDefaultsHelpFormatter

import grpc
from helloworld_pb2 import HelloRequest, HelloReply
from helloworld_pb2_grpc import MultiGreeterServicer, add_MultiGreeterServicer_to_server

import iris

NUMBER_OF_REPLY = 10

parser = ArgumentParser(formatter_class=ArgumentDefaultsHelpFormatter)
parser.add_argument("-p", "--port", default="50051", help="Server port")
args = vars(parser.parse_args())
```

```

class Greeter(MultiGreeterServicer):

    def SayHello(self, request: HelloRequest, context) -> HelloReply:
        logging.info("Serving SayHello request %s", request)
        obj = iris.cls("dc.jrpereira.gRPC.HelloWorldServer")._New()
        # ObjectScript ???????
        return obj.SayHelloObjectScript(request)

    def SayHelloStream(self, request: HelloRequest, context: grpc.aio.ServicerContext
) -> HelloReply:
        logging.info("Serving SayHelloStream request %s", request)
        obj = iris.cls("dc.jrpereira.gRPC.HelloWorldServer")._New()
        n = request.num_greetings
        if n == 0:
            n = NUMBER_OF_REPLY
        for i in range(n):
            # ObjectScript ???????
            yield obj.SayHelloObjectScript(request)

def get_server():
    port = args["port"]
    server = grpc.server(futures.ThreadPoolExecutor(max_workers=10))
    add_MultiGreeterServicer_to_server(Greeter(), server)
    listen_addr = f"[::]:{port}"
    server.add_insecure_port(f"[::]:{port}")
    logging.info("Starting server on %s", listen_addr)
    return server

def handle_sigterm(*_: Any) -> None :
    """?????"""
    done_event = server.stop(None)
    done_event.wait(None)
    print('Stop complete.')

logging.basicConfig(level=logging.INFO)

server = get_server()
server.start()

# https://groups.google.com/g/grpc-io/c/6Yi_oIQsh3w
signal.signal(signal.SIGTERM, handle_sigterm)

server.wait_for_termination()

```

ご覧のとおり上記では、protobuf の仕様に定義された SayHello() と SayHelloStream() メソッドが実装されています。

SayHello() メソッドは 1 つの値のみを送信するのに対し、SayHelloStream() メソッドは NUMBER_OF_REPLY に等しい数のメッセージを Python 演算子の yield を通じてクライアントに返します。

また、ObjectScript に定義されたロジックをインジェクトするフックを作成していることにも注意してください。そのため、dc.jrpereira.gRPC.HelloWorldServer クラスに SayHelloObjectScript というメソッドを定義しています。

```

Method SayHelloObjectScript(request)
{

```

```
Set sys = $system.Python.Import("sys")
Do sys.path.append("/usr/irissys/mgr/python/grpc-test/")

Set helloworldpb2 = $system.Python.Import("helloworld_pb2")

Set reply = helloworldpb2.HelloReply()
Set reply.message = "Hi "_request.name_"! :)"

Return reply
}
```

このようにすることで、Python から gRPC クライアントからのリクエストを受信し、Python と ObjectScript でコーディングされた混合ロジックを使用して処理することができます。

サービス用のクライアントを実装する

クライアントのコードは並列である必要がないため、ObjectScript クラスに直接 Python コードを使って実装しました。

```
ClassMethod ExecutePython() [ Language = python ]
{
    import sys
    sys.path.append('/usr/irissys/mgr/python/grpc-test/')

    import grpc
    from helloworld_pb2 import HelloRequest
    from helloworld_pb2_grpc import MultiGreeterStub

    channel = grpc.insecure_channel('localhost:50051')
    stub = MultiGreeterStub(channel)

    response = stub.SayHello(HelloRequest(name='you'))
    print("Greeter client received: " + response.message)

    for response in stub.SayHelloStream(HelloRequest(name="you")):
        print("Greeter client received from stream: " + response.message)
}
```

まず、生成される protobuf コードをインポートできるように、Python パスに grpc-test ディレクトリを追加します。

次に、localhost を ポート 50051 で接続し、スタブ（またはクライアント）を作成しています。

このようなクライアントを使用して、SayHello() と SayHelloStream() メソッドを通じて、localhost:50051 でリスンしているサーバーの情報をリクエストできます。

SayHello() メソッドは 1 つの値のみを返すため、リクエストを行ってそのレスポンスを使用するだけです。一方で、SayHelloStream() メソッドは、データのストリームをコレクションで返すため、イテレートしてすべてのデータを取得する必要があります。

コードをテストする

では、コードをテストしてみましょう。

全コードは、[hello world プロジェクト](#)にあります。 次のステップに従って、コードを実行してください。

```
git clone https://github.com/jrpereirajr/iris-grpc-example
cd iris-grpc-example
docker-compose up -d
```

次に、システムターミナルまたは Visual Studio Code を介して IRIS ターミナルを開きます。

```
docker exec -it iris-grpc-example_iris_1 bash
iris session iris
```

gRPC サーバーを起動します。

```
Set server = ##class(dc.jrpereira.gRPC.HelloWorldServer).%New()
Do server.Start()
```

では、このサーバーと対話する gRPC クライアントを作成しましょう。

```
Set client = ##class(dc.jrpereira.gRPC.HelloWorldClient).%New()
Do client.ExecutePython()
```

すべてが正常に実行されれば、ターミナルに多数の挨拶メッセージが表示されます。

最後に、サーバーを停止しましょう。

```
Do server.Stop()
```

Hello World 内で grpcurl ユーティリティを使用する

前に述べたように、[grpcurl ユーティリティ](#)は curl に相当するものですが、ここでは、（curl のように）HTTP クライアントとして動作する代わりに、grpcurl を gRPC クライアントとして使用して、実行中の gRPC サーバーからサービスをテストします。では、これを使用して、Hello World をもう少しだけ実行してみましょう。

まず、grpcurl ユーティリティをダウンロードしてインストールします。

```
cd /tmp
wget https://github.com/fullstorydev/grpcurl/releases/download/v1.8.6/grpcurl_1.8.6_linux_x86_64.tar.gz
tar -zxvf grpcurl_1.8.6_linux_x86_64.tar.gz
```

以下を入力して、インストールが正しく行われたかを確認します。

```
./grpcurl --help
```

正しく実行されれば、すべての grpcurl オプションが表示されます。

では、サーバーで利用できるサービスを問い合わせてみましょう。

```
./grpcurl \  
  -plaintext \  
  -import-path /irisrun/repo/jrpereira/python/grpc-test \  
  -proto helloworld.proto \  
  localhost:50051 \  
  list
```

以下のレスポンスが受信されます。

```
helloworld.MultiGreeter
```

ご覧のとおり、このユーティリティはすべての利用可能なサービスをリストする命令に対するレスポンスとして、proto ファイルに定義されたサービス (helloworld.MultiGreeter) を返しました。

上記のコマンドでは、パラメーターごとに改行しました。それぞれについて説明します。

-plaintext: TLS を使用せずに gRPC を使用できるようにします (安全でないモード)。ここでは、サーバーに安全な接続を実装していないため、この設定を使用しています。当然ながら、非本番環境でのみ使用してください。

-import-path および -proto: .proto

ファイル (サービス定義) のパスと名前です。サーバーがリフレクションを実装しない場合に必要です。

上記のパラメーターの後にサーバーのホスト名とポートを指定し、この例では grpcurl コマンドの list を指定しています。

では、helloworld.MultiGreeter サービス内のすべてのメソッドを問い合わせてみましょう。

```
./grpcurl \  
  -plaintext \  
  -import-path /irisrun/repo/jrpereira/python/grpc-test \  
  -proto helloworld.proto \  
  localhost:50051 \  
  list helloworld.MultiGreeter
```

以下の内容が出力されます。

```
helloworld.MultiGreeter.SayHello  
helloworld.MultiGreeter.SayHelloStream
```

ご覧のとおり、これがサーバーのコードを生成するために使用される proto ファイルに定義されたメソッドです。

次に、SayHello() メソッドをテストしましょう。

```
./grpcurl \  
  -plaintext \  
  -d '{"name": "you"}' \  
  -import-path /irisrun/repo/jrpereira/python/grpc-test \  
  localhost:50051 \  
  sayhello
```

```
-proto helloworld.proto \  
localhost:50051 \  
helloworld.MultiGreeter.SayHello
```

以下の出力が期待されます（前にクライアントが実装したものと同じです）。

```
{  
  "message": "Hi you! :)"  
}
```

もう 1 つの SayHelloStream() メソッドもテストしましょう。

```
./grpcurl \  
-plaintext -d '{"name":"you"}' \  
-import-path /irisrun/repo/jrpereira/python/grpc-test \  
-proto helloworld.proto localhost:50051 \  
helloworld.MultiGreeter.SayHelloStream
```

10 個の挨拶メッセージが含まれるストリームが取得されます。

```
{  
  "message": "Hi you! :)"  
}  
{  
  "message": "Hi you! :)"  
}  
...  
{  
  "message": "Hi you! :)"  
}
```

最後に、protobuf メッセージのもう 1 つの numgreetings

プロパティを使用するように、このコマンドを少し変更しましょう。

これは、ストリーム内で送信されるメッセージ数を制御するために、サーバーが使用するプロパティです。

このコマンドはサーバーに対し、ストリーム内に含めるメッセージ数をデフォルトの 10 ではなく 2 にして返すように指示しています。

```
./grpcurl \  
-plaintext -d '{"name":"you", "num_greetings":2}' \  
-import-path /irisrun/repo/jrpereira/python/grpc-test \  
-proto helloworld.proto localhost:50051 \  
helloworld.MultiGreeter.SayHelloStream
```

すると、ターミナルには以下のように表示されます。

```
{  
  "message": "Hi you! :)"  
}
```

```
{  
  "message": "Hi you! :)"  
}
```

まとめ

この記事では、gRPC について、主に REST と比較したメリットとデメリットも合わせて説明しました。
また、公式の gRPC リポジトリにある Python 用サンプルを一部変更して、IRIS
での使用例もいくつかテストしました。

上記で述べたとおり、gRPC にはすでに採用済みのユースケースがいくつかあり、相互運用性は可能性の 1
つです。IRIS Interoperability アダプターを作成するのは、IRIS で gRPC
を実用的に使用する上で自然な考え方と言えます。

ただし、これには多くの作業が必要とされます。これについては、別の記事で取り上げることにします =)

この記事の内容が役に立ちますように！ それではまた！

参考情報

<https://grpc.io/docs/what-is-grpc/introduction/> <https://developers.google.com/protocol-buffers>
<https://en.wikipedia.org/wiki/GRPC> <https://www.imaginarycloud.com/blog/grpc-vs-rest/>
<https://www.vineethweb.com/post/grpc/> <https://www.capitalone.com/tech/software-engineering/grpc-framework-for-...> <https://www.altexsoft.com/blog/what-is-grpc/>

[#Embedded Python](#) [#InterSystems IRIS](#)

ソースURL:<https://jp.community.intersystems.com/post/grpc-%E3%81%A8-hello-world>