Engineer

gRPC

HTTP/2で遠隔の手続きを呼び出すフレームワーク

どのようなものか

gRPCは、遠隔手続き呼び出しのフレームワークです。プログラムは、別のマシンで動くサーバーを手元のオブジェクトのように扱い、メソッドを呼びます。gRPCが、HTTP/2上の独自プロトコルで呼び出しをネットワーク越しに運び、応答を戻り値として返します。

出発点はサービス定義です。遠隔で実行する操作と、それぞれのリクエスト、応答の型を並べます。既定では、Googleのオープンソースのデータ直列化形式Protocol Buffersを使い、末尾が `.proto` のファイルへ書きます。言語やプラットフォームに依存しない形式です。ほかの形式も組み込めますが、指定しなければprotocol buffersを使います。コンパイラー `protoc` とgRPCプラグインがファイルを読み、両側のコードを生成します。クライアントには、サービスと同じメソッドを持つ手元のオブジェクト、スタブを作ります。単にクライアントと呼ぶ言語もあります。サーバー側には、実装すべきインターフェースを作ります。両側が一ファイルから生まれるため、Java製のサーバーが、Go、Python、Ruby製のクライアントへ答えられます。

メソッドの形は四つあります。単項メソッドは、普通の関数のように、一つのリクエストから一つの応答を返します。サーバーストリーミングは、一つのリクエストへ応答のストリームを返し、クライアントストリーミングは、その逆です。双方向ストリーミングでは、両側が独立してストリームを送れます。一つの呼び出しの中では、送った順でメッセージが届くことをgRPCが保証します。

通信にはHTTP/2を使います。各呼び出しは、完全修飾したサービス名とメソッド名からなるパスへの `POST` で、コンテンツ型は `application/grpc` です。各メッセージの前には五バイトの接頭部があり、一バイトが圧縮の有無、四バイトが長さを表します。呼び出しの結果は、HTTPの状態ではありません。gRPCが作る応

答は、呼び出しが失敗しても状態200で、実際の結果は、本文の後に送るヘッダーフィールド、トレーラーに入ります。`grpc-status`が数値を持ち、必要に応じて`grpc-message`がエラーの説明を持ちます。

このためRESTとは性格が違います。REST APIは、リソースに名前を付け、HTTP自体のメソッドと状態コードで操作し、普通は誰でも読めるJSONの本文を使います。gRPC APIは、型付きのバイナリーメッセージを使うメソッドの一覧で、HTTP/2は下で運ぶ役目だけです。Protocol Buffersの文書もJSONと比較し、メッセージの小ささと処理の速さを強調します。

概念を判断に生かす

「gRPCとは？」を実際の判断に生かしましょう。その概念を、開発または購入するシステムに結び付けて考えるお手伝いをします。 データの保存先、処理のルール、他のシステムとの接続方法など、利用者の目に見えない部分をつくります。

サービスを見る お問い合わせ →

用語の背後にある前提を検証する

解説とリンク先の実績を参考に、プロジェクトについての質問を整理してください。ツールを選ぶ前に、前提を一緒に検討できます。 ETL/ELTパイプライン6本を設計・運用 — 6 3種類のDBエンジンにまたがるETLパイプライン

ETL/ELTパイプライン6本を設計・運用 お問い合わせ →

なぜ重要なのか

まず契約があり、そこからコードが生まれます。多くのHTTP APIでは、APIの説明と、それを実装するコードが別にあり、ずれる場合があります。クライアントも、文書を読んで手で書くことがよくあります。gRPCでは、メソッドやフィールドの定義場所は`.proto`だけで、そこからすべてのクライアントとサーバーを生成します。メッセージを変えると、誰かが文書を読み直すのではなく、同じ生成ツールを通して、すべての言語へ伝わります。

コンパクトに符号化できます。各フィールドを、名前ではなく、番号と値で送るためです。代償として、protocol buffersは自己記述形式ではないため、`.proto`なしには、完全には解釈できません。また、メッセージ全体が一度にメモリーへ収まることを前提とします。公式文書は、数メガバイトを超えるデータには、別の方法を勧めます。

ストリーミングは追加機能ではなく、設計の一部です。HTTP/2接続上のストリームとして呼び出すため、サーバーストリーミングのメソッドは、一度だけ答える呼び出しと同じ仕組みで、結果を送り続けられます。

時間制限も呼び出しとともに伝わります。クライアントが期限を設定すると、gRPCが残り時間を `grpc-timeout` ヘッダーで送り、時間が尽きると、クライアントは `DEADLINE_EXCEEDED` を受け取り、サーバー側の呼び出しはキャンセルされます。サーバーがさらに別のサーバーを呼ぶとき、一部の実装は期限を引き継ぎます。JavaとGoは既定で行い、C++は有効化が必要です。期限を時刻ではなく残り時間として各区間へ渡すため、時計がずれるマシン同士でも、いつ諦めるかは一致します。

エラーにも共通の語彙があります。gRPCは、0の `OK` から16の `UNAUTHENTICATED` まで、17の状態コードをプロトコル自体に定義します。そのためGoのクライアントとJavaのサーバーでも、 `NOT_FOUND` や `UNAVAILABLE` の意味が一致します。

どれも、同じチームや企業が両端を管理し、 `.proto` を共有してビルドできる、サービス間の通信に適しています。ブラウザーには、それほど向きません。gRPCプロジェクトのプロトコル文書は、ブラウザーはスクリプトへHTTP/2のフレーミングを公開しておらず、今後もしないと記します。そのためブラウザーのコードは、gRPC-Webという別のプロトコルを使い、普通はプロキシーが通常のgRPCへ変換します。

実際の活用例

小さい請求サービスを示します。一つの単項メソッドが請求書を取得し、一つのサーバーストリーミングのメソッドが、顧客の請求書を一件ずつ送ります。

```
syntax = "proto3";

package billing.v1;

service InvoiceService {
  rpc GetInvoice(GetInvoiceRequest) returns (Invoice);
  rpc WatchInvoices(WatchInvoicesRequest) returns (stream Invoice);
}

message GetInvoiceRequest {
  string invoice_id = 1;
}

message WatchInvoicesRequest {
  string customer_id = 1;
}

message Invoice {
  reserved 4;
  reserved "discount_code";
```

```
  string invoice_id = 1;
  string customer_id = 2;
  int64 total_cents = 3;
  string currency = 5;
}
```

数値は既定値ではありません。フィールド番号で、通信中は名前の代わりに各フィールドを識別します。1から15は一バイトで符号化できるため、特によく設定するフィールドへ割り当てます。二つの `reserved` 行は、かつて存在した、番号4、名前 `discount_code` のフィールドを記録し、どちらも別のフィールドへ使い直せないようにします。

Pythonでは、一つのコマンドが、メッセージのクラスを持つ `billing_pb2.py` と、クライアントとサーバーのクラスを持つ `billing_pb2_grpc.py` を生成します。

```
python -m grpc_tools.protoc -Iprotos \
  --python_out=. --grpc_python_out=. protos/billing.proto
```

`GetInvoice` の一回の呼び出しは、HTTP/2のフレームでは次のようになります。

```
HEADERS (flags = END_HEADERS)
:method = POST
:scheme = https
:path = /billing.v1.InvoiceService/GetInvoice
:authority = billing.example.com
te = trailers
content-type = application/grpc+proto
grpc-timeout = 2S

DATA (flags = END_STREAM)
<Length-Prefixed Message>

HEADERS (flags = END_HEADERS)
:status = 200
content-type = application/grpc+proto

DATA
<Length-Prefixed Message>

HEADERS (flags = END_STREAM, END_HEADERS)
grpc-status = 0
```

`2S` は二秒で、 `te = trailers` は、gRPCと互換性のないプロキシーを検出するためです。請求書がなくても、応答の始まりは `:status = 200` で、最後のフレームに `grpc-status = 5` を入れ、 `NOT_FOUND` を表します。 `grpc-message` には、例えば `invoice not found` を添えます。すぐ失敗するサーバーは、メッセージをまったく付けず、状態を入れた一つの `HEADERS` フレームだけを返す場合もあります。

一つの契約から両側を生成するのは、gRPCだけではありません。このサイトのデータベースを起点にAPI契約を生成した事例は、OpenAPIで記述するAPIで行いました。流れはデータベースから外へ続きます。Goの型でAPIを定義し、HumaがOpenAPIの説明を生成し、そこから型付きのTypeScriptクライアントとモックハンドラーを作ります。フロントエンドの送るものとAPIの期待が一致しなくなれば、CIの契約検査がビルドを失敗させます。gRPCは、 `.proto` を唯一のソースにして、同じ規律をツールチェーンへ組み込みます。

問題が起きるところ

期限のない呼び出しが、いつまでも待ちます。 gRPCは既定で期限を設定しないため、応答が止まったサーバーを呼ぶクライアントは、事実上いつまでも応答を待ち続ける可能性があります。各呼び出しへ現実的な期限を付けてください。Pythonでは、秒単位の引数 `timeout` です。

```
import grpc

import billing_pb2
import billing_pb2_grpc

credentials = grpc.ssl_channel_credentials()
with grpc.secure_channel("billing.example.com:443", credentials) as channel:
    stub = billing_pb2_grpc.InvoiceServiceStub(channel)
    request = billing_pb2.GetInvoiceRequest(invoice_id="INV-1001")
    invoice = stub.GetInvoice(request, timeout=2.0)
```

時間が尽きると、 `grpc.RpcError` が発生し、その `code()` が `grpc.StatusCode.DEADLINE_EXCEEDED` を返します。サーバー側では、期限を過ぎるとgRPCが `CANCELLED` で呼び出しをキャンセルします。ただし、ハンドラーが始めた仕事を止めるのは、サービス自身のコードの責任です。長く動くハンドラーは、キャンセルを調べ、自分で停止する必要があります。

タイムアウトしても、何も起きなかったとは限りません。 成否はクライアントとサーバーが別々に決めるため、サーバーでは完了しても、クライアントでは `DEADLINE_EXCEEDED` で失敗する場合があります。キャンセルしても、済んだ仕事は

元へ戻りません。単に再試行すると、同じ請求書を二度作る可能性があります。冪等であると明示しない限り、プロトコルはそう仮定しません。`UNAVAILABLE`は、一時的で、待ち時間を置いて再試行する価値がある可能性の最も高いコードです。ただし状態コードの文書は、冪等でない操作の再試行は、常に安全とは限らないと警告します。どのコードで再試行するかも、呼び出すコードの判断に任せています。

フィールド番号を再利用して、データを壊します。 メッセージ型を使い始めた後は、番号を変えられません。フィールドを消し、後で番号を別のものへ与えると、古いクライアントや保存したメッセージが、別の意味だったバイト列を、新しいフィールドとしてデコードします。Protocol Buffersのガイドは、解析エラー、個人データの漏えい、データ破損を挙げます。番号は決して振り直さず、上の例のように、削除した番号と名前をすべて予約してください。

HTTPからは、どの失敗も成功に見えます。 gRPCは失敗してもHTTP状態200を返すため、アクセスログや状態コードで作るエラー率など、HTTP状態だけで判断するものは、失敗を成功に数えます。仕様が、`application/grpc`でないコンテンツ型のリクエストへ415を返させるのも、ほかのHTTP/2クライアントに、gRPCのエラーを成功と読ませないためです。`grpc-status`を数え、`:status`で判断しないようにします。

長いストリームが、一台へ負荷を固定します。 呼び出しは共有HTTP/2接続上のストリームで、一度始まると、別のサーバーへ移せません。何時間も続けば、その後いくらサーバーを増やしても、負荷は開始した場所に残ります。接続ごとに並行ストリーム数の上限もあり、達すると、終わるまで後の呼び出しがクライアントで待ちます。gRPCの性能ガイドは、性能や単純さの面で大きな利点がある場合だけストリーミングを使い、負荷の高い領域ごとに、別のチャネルを使うよう勧めます。

大きいデータが、一メッセージに収まりません。 gRPCは最大メッセージサイズを強制し、設定上限を超えると、送信側でも受信側でも`RESOURCE_EXHAUSTED`で失敗します。上限を上げても、問題が移るだけです。protocol buffersは全体がメモリーに収まることを前提とし、大きいデータでは、直列化したコピーがメモリー内に同時に複数存在する場合があります。小さいメッセージのストリームへ分けるか、gRPCの外で扱います。

参考資料

- 基本概念、構成、ライフサイクル — サービス定義、四つのメソッド、スタブ、期限、キャンセルです。

- HTTP2上のgRPC — 通信プロトコルで、パス、ヘッダー、メッセージ接頭部、トレーラーの状態を説明します。
- 状態コード — 全17コードの意味と、ライブラリー自身が決して作らないコードです。
- 期限 — 既定値がない理由、サーバーへの伝達方法、サービス間の引き継ぎです。
- 言語ガイド：proto 3 — フィールド番号、予約フィールド、メッセージ型を安全に変える規則です。

用語集

実際の制約に合わせて選ぶ

業務の流れ、既存のインターフェース、変更を必要とする利用者をお知らせください。検証可能な納品内容と、お客様のチームが保守を続けられる引き継ぎ内容を定められます。

料金を見る お問い合わせ →

用語を実務に結び付ける

「gRPCとは？」がお客様の組織にどう当てはまるかをご相談ください。現在のシステムの具体例が一つあれば、話を始められます。

サービスを見る

welcome@engineer.company +45 50 10 61 84 デンマーク、コペンハーゲン

このページを持ち歩く

連絡先カード

心を込めて丁寧に制作： D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

著作権 © 2025 – 現在 · Engineer ApS · すべての権利を留保します。

印刷元： https://engineer.company/ja/glossary/grpc/