

# 目次

<b>第 1 章</b>	<b>gRPC Web Internals</b>	<b>3</b>
1.1	gRPC	3
1.2	gRPC Web とは	4
1.3	gRPC と gRPC Web の違い	5
1.3.1	HTTP のプロトコルバージョン	5
1.3.2	ヘッダ、トレイラ	5
1.4	Unary RPC と基本的な構造	5
1.4.1	環境のセットアップ	6
1.4.2	リクエストペイロードの取得	7
1.4.3	gRPC リクエストの定義	9
1.4.4	Codec	9
1.4.5	Protocol Buffers	10
1.4.6	レスポンスボディの取得	12
1.5	Server streaming RPC	13
1.6	Client streaming RPC	14
1.6.1	WebSocket	15
1.7	Bidirectional streaming RPC	16
1.8	HTTP/1.1 or HTTP/2	17
1.9	WHATWG Fetch/Streams	17
1.10	おわりに	17

# 第 1 章

## gRPC Web Internals

### 1.1 gRPC

gRPC は Google が開発した RPC フレームワークです。gRPC という名前の由来は、gRPC: Remote Procedure Calls<sup>\*1</sup> であり、再帰的頭字語になっています。また、各バージョンのコードネームとして g から始まる 単語<sup>\*2</sup> が割り当てられています。

gRPC の大きな特徴として、HTTP/2 上のプロトコルであり、ストリームを扱うための方法が用意されている点や、Protocol Buffers などによってコンテンツをシリアル化してサーバ・クライアント間のやり取りをする点が挙げられます。シリアル化には任意のライブラリを使用することができ、例えば JSON や FlatBuffers 等を利用することもできます。

ほとんどの Protocol Buffers の各言語向けのライブラリには、gRPC のクライアント・サーバ間のインターフェースコードを生成するプラグインが用意されているため、開発者は同じようなコードが何度も現れる、クライアント・サーバ間のインターフェースを書く必要がなくなり、アプリケーションロジックにのみ注力することができるようになります。また、JSON 等とは異なり、Protocol Buffers は静的型付けであるというのも強みの一つです。

この章では、Protocol Buffers がデファクトスタンダードとなっているため、以降で使用するシリアル化ライブラリは Protocol Buffers を前提とすることにします。

gRPC では、各 API メソッドを定義する際に、いくつかの種類から RPC の通信方式を選択することができます。

Unary RPC は 1 つのリクエストに対し、常に 1 つのレスポンスが返ってくる方式です。Server streaming RPC は 1 つのリクエストに対し、0 個以上の複数のレスポンスが

---

<sup>\*1</sup> What does gRPC stand for? <https://grpc.io/faq/>

<sup>\*2</sup> [https://github.com/grpc/grpc/blob/master/doc/g\\_stands\\_for.md](https://github.com/grpc/grpc/blob/master/doc/g_stands_for.md)

返ってくる方式です。Client streaming RPC は 0 個以上のリクエストを送信後に、1 つのレスポンスを返ってくるような方式です。Bidirectional streaming RPC はその名の通り、双方向にストリーム通信を行う方式で、クライアントかサーバのどちらかがストリームを終了させない限りずっとコネクションが保持されます。

Protocol Buffers で各 streaming RPC を定義するには、以下のように対応する箇所に `stream` を付与します。それ以外で Unary RPC との差異はありません。

```
service API {  
  rpc Unary (SimpleRequest) returns (SimpleResponse) {}  
  rpc ServerStreaming (SimpleRequest) returns (stream SimpleResponse) {}  
  rpc ClientStreaming (stream SimpleRequest) returns (SimpleResponse) {}  
  rpc BidiStreaming (stream SimpleRequest) returns (stream SimpleResponse) {}  
}
```

この他にもいくつか gRPC 特有の概念がありますが、あまり仕様は多くありません。各言語向けに公式の チュートリアル<sup>\*3</sup> が用意されているため、それを参照するとすぐに gRPC アプリケーションを開発できるようになるでしょう。

## 1.2 gRPC Web とは

gRPC Web は gRPC を Web ブラウザでも使えるようにするためのプロトコルです。通常、gRPC は Android や iOS のモバイルアプリケーションとサーバ間や、マイクロサービス間などのプロトコルとして利用される場合が多いです。同様に、Web アプリケーションとサーバ間でも利用したいという需要は多くありますが、そのままの gRPC プロトコルを利用できないという問題があります。

モバイルアプリケーションと違い、Web アプリケーションは Web ブラウザ上で動作するため、特有の問題が常に付きまといます。1 つ目は CORS や XSS、CSRF などのセキュリティに関する問題です。2 つ目は TLS 対応をしていない Web サイトや、レガシーな Web ブラウザ上で HTTP/2 を使用することができない問題です。gRPC には `insecure` オプションがあり、開発時などでは TLS の対応をせずに通信ができますが、Web ブラウザでは HTTP/2 を使用する際には HTTPS が必須であるため、`insecure` オプションを使うことができません。

gRPC Web は、そういった Web ブラウザ特有の問題を解決できるように定義された gRPC ベースのプロトコルです。

---

<sup>\*3</sup> <https://grpc.io/docs/tutorials/>

## 1.3 gRPC と gRPC Web の違い

この節では、具体的なプロトコルの仕様の違いを見ていきます。gRPC<sup>[\*4]</sup>・gRPC Web<sup>[\*5]</sup>の仕様はどちらも gRPC のリポジトリ内に置かれています。

### 1.3.1 HTTP のプロトコルバージョン

前節でも挙げられていたように、Web ブラウザでは HTTP/1 と HTTP/2 の 2 つのバージョンのいずれかが使用されます。それに対し、gRPC は HTTP/2 のみのサポートであるため、gRPC Web では条件に応じてプロトコルバージョンを切り替えるといったことが必要になります。

また、HTTP/2 はバイナリベースのプロトコルであるのに対し、HTTP/1 はテキストベースのプロトコルです。そのため、Internet Explorer 10 などのレガシー Web ブラウザをサポートするためにはバイナリベースではなくテキストベースで通信をしなければいけません。gRPC Web では HTTP/1 を使う際には、コンテンツをテキストエンコードしなければいけません。デフォルトのエンコーディング方式として Base64 を使うことが定められています。

### 1.3.2 ヘッダ、トレイラ

ヘッダやトレイラにもいくつかの些細な違いがあります。例えば、Content-Type に `application/grpc` ではなく、`application/grpc-web` を使う点などがあります。基本的には、リクエストが何のプロトコルを使っていて、シリアライザやエンコーダには何を使っているかを記すためのものなので、当たり前の変更点と言えます。

## 1.4 Unary RPC と基本的な構造

Unary RPC は最も基本的な RPC 方式なので、当然 gRPC Web でもサポートされています。どのようなリクエストが送信され、どのようなレスポンスが返ってきているのかを実際に見てみましょう。ここでは gRPC Web プロトコルの実装として `improbable-eng/grpc-web`<sup>[\*6]</sup> を利用します。

<sup>\*4</sup> <https://github.com/grpc/grpc/blob/master/doc/PROTOCOL-HTTP2.md>

<sup>\*5</sup> <https://github.com/grpc/grpc/blob/master/doc/PROTOCOL-WEB.md>

<sup>\*6</sup> <https://github.com/improbable-eng/grpc-web>

### 1.4.1 環境のセットアップ

この章のために用意した `ktr0731/techbookfest-5-grpc-web`<sup>\*7</sup> を利用します。Git でリポジトリをクローンし、定義された Protocol Buffers ファイルを元に JavaScript のインターフェースを自動生成します。

なお、Node.js の v10.8.0、Go の v1.10.0 以上が必要です。

```
$ git clone https://github.com/ktr0731/techbookfest-5-grpc-web
$ cd techbookfest-5-grpc-web
$ yarn run proto # or npm run proto
```

すると `api` ディレクトリ以下に複数のファイルが生成されています。

```
$ ls api
api.proto          api_pb.js          api_pb_service.js
api_pb.d.ts        api_pb_service.d.ts
```

今回は TypeScript ではなく JavaScript でコードを書くため、`ts` と拡張子がついているファイルは不要です。

`client/index.js` はこの生成された JavaScript をインポートしています。`client/index.js` の始めのコードは以下のようになっています。

```
import { SimpleRequest } from '../api/api_pb';
import { SimpleServiceClient } from '../api/api_pb_service';

const client = new SimpleServiceClient('http://localhost:50051');

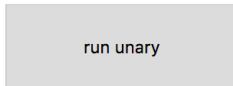
document.querySelector("#unary-button").addEventListener("click", e => {
  const req = new SimpleRequest();
  req.setName("kumiko oumae");

  client.unary(req, (err, res) => {
    if (err) {
      console.error(err);
      return
    }
    const e = document.querySelector("#unary-message");
    e.innerText = res.getMessage();
  });
});
```

---

<sup>\*7</sup> <https://github.com/ktr0731/techbookfest-5-grpc-web>

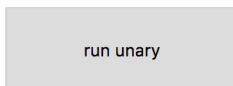
コマンドライン上で `yarn start` とコマンドを実行するとサーバが立ち上がります。Chrome で `http://localhost:1234` を見ると `run unary` というボタンが表示されています。



▲図 1.1 run unary ボタン

### 1.4.2 リクエストペイロードの取得

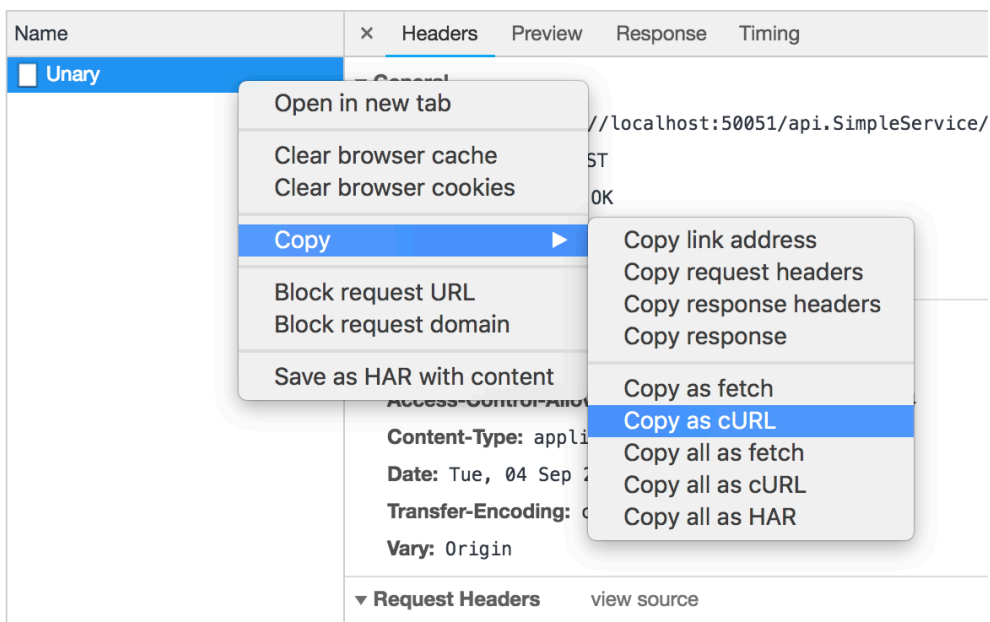
`run unary` ボタンをクリックすると、gRPC サーバの `unary` という名前の API へリクエストが飛びます。コードを見ればすぐわかるかと思いますが、そのレスポンスは `run unary` ボタン直下の `p` 要素へ反映されます。



Hello, kumiko oumae!

▲図 1.2 レスポンス

次に、Chrome の 検証 など Chrome DevTools を開き、先程のリクエストを cURL のコマンドとして取得します。



▲図 1.3 Copy as cURL

そのコピーしたコマンドの引数 `--data-binary` に送信されたリクエストのペイロードが入っています。

```
'\x00\x00\x00\x00\x0e\n\x0ckumiko oumae'
```

`xxd` を使い、16 進数へ変換してみると以下のようなバイト列になっていることがわかります。

```
00000000: 0000 0000 0e0a 0c6b 756d 696b 6f20 6f75  ....kumiko ou
00000010: 6d61 65                                mae
```

これを元にリクエストペイロードを読んでいきます。

### 1.4.3 gRPC リクエストの定義

gRPC<sup>\*8</sup> の仕様では、リクエストは以下のような ABNF 記法で定義されています。なお、紙面の都合上、一部省略している部分があるため正確な定義はリンク元を参照してください。

```
Request → Request-Headers *Length-Prefixed-Message EOS
```

今回重要な部分は `Length-Prefixed-Message` です。Message はリクエストのペイロードを指しています。

`Length-Prefixed-Message` は以下のような定義です。

```
Length-Prefixed-Message → Compressed-Flag Message-Length Message
Compressed-Flag → 0 / 1 # encoded as 1 byte unsigned integer
Message-Length → {length of Message} # encoded as 4 byte unsigned integer
Message → *[binary octet]
```

これを見ると、先頭 1 バイトは圧縮されているかどうかのフラグ、次の 4 バイトはメッセージのサイズ、そしてメッセージ本体の順番で現れるようです。unary のペイロードと照らし合わせながら見てみると、このペイロードは圧縮されておらず、メッセージのサイズは 14 バイト、メッセージの実態は 0a0c 6b75 6d69 6b6f 206f 756d 6165 ということを知ることができます。

ここまででペイロードの実態が 0a0c 6b75 6d69 6b6f 206f 756d 6165 ということがわかりました。ただ、このバイト列はどう解釈されているのでしょうか？ ここからは Codec の種類に依存します。

### 1.4.4 Codec

gRPC には Codec という概念があります。これは、メッセージ (= リクエストのペイロード) の marshaler / unmarshaler を包括したものです。Codec により、gRPC とメッセージの marshal / unmarshal は明確に分離されています。

デファクトスタンダードなものとして Protocol Buffers があります。

Go の場合、Codec は [google.golang.org/grpc/encoding](https://google.golang.org/grpc/encoding)<sup>\*9</sup> に定義されています。

<sup>\*8</sup> <https://github.com/grpc/grpc/blob/master/doc/PROTOCOL-HTTP2.md>

<sup>\*9</sup> <https://godoc.org/google.golang.org/grpc/encoding#Codec>



GoDoc にも記載されている通り、`encoding.RegisterCodec` を使って `encoding.Codec` インターフェースを満たす実装を登録をすることができ、`marshal / unmarshal` 時にはリクエストの `Content-Type` を元に最適な Codec が暗黙的に使用されます。

### 1.4.5 Protocol Buffers

Codec を理解したので、さっそくペイロードを Protocol Buffers Version 3 として解釈して読んでみましょう。Protocol Buffers の構造は、公式のページ<sup>\*10</sup> に詳しく記載されています。

#### 基本型のエンコーディング

まずは、Protocol Buffers の基本型のエンコーディングについて見ていきます。Protocol Buffers の各バイトには MSB (Most Significant Bit) があり、MSB が 0 の場合はバイト列の終端を表します。

`varint` (整数値、`bool`、`enum` で使われる型の総称) は、その数値を 2 進数へ変換した後、MSB を付与します。一点注意したいのが、Protocol Buffers はリトルエンディアンが使われていることです。そのため、最下位バイトが先頭に、最上位バイトが末尾に格納されます。

例として、150 を手動エンコードしてみます。まず 150 を 2 進数へ変換します。150 は 1 バイトで表せるため、MSB が 0 となることが分かります。

```
0111 1000
```

これを 16 進数に変換して完了です。

```
96
```

至極単純です。次に 1 ビットでは表せない 300 を考えてみます。

```
1 0010 1100
```

MSB を格納するために、バイト列の区切りを変えます。

---

<sup>\*10</sup> <https://developers.google.com/protocol-buffers/docs/encoding>

```
000 0010 010 1100
```

MSB を追加するまえに、バイト列をリトルエンディアンへ変換します。

```
010 1100 000 0010
```

2 バイトのため、最初のバイトの MSB は 1 になります。

```
1010 1100 0000 0010
```

16 進数へ変換します。

```
ac 02
```

string 型はもう少し簡単です。例えば `testing` という文字列をエンコードする場合、文字列の長さ + 文字列の UTF8 表現で表します。

```
07 74 65 73 74 69 6e 67
```

上記の例だと、07 が文字列の長さ、74 65 73 74 69 6e 67 は `testing` に該当するバイト列です。

他にもいくつか型の種類がありますが、省略します。詳しくはリンク先を参照してください。

## key と value

実際の Protocol Buffers のエンコーディングには、先程見たようなメッセージに含まれるフィールドの `value` と、その `value` がどのフィールドなのかと、型の `wire type` を識別するための `key` が必要になります。wire type は Protocol Buffers により各型と対応付けがなされています。

`key` は 1 バイトで表現され、 $(\text{field\_number} \ll 3) \mid \text{wire\_type}$  を計算することで算出されます。ただし、先頭 1 ビットは MSB として使われることに注意します。例えばフィールド番号が 1、型 が `string` の場合を考えます。`string` の wire type は 2 なので、

```
(1 << 3) | 2 = 0b1010 = 0xa
```

が key になります。

### ペイロードの解読

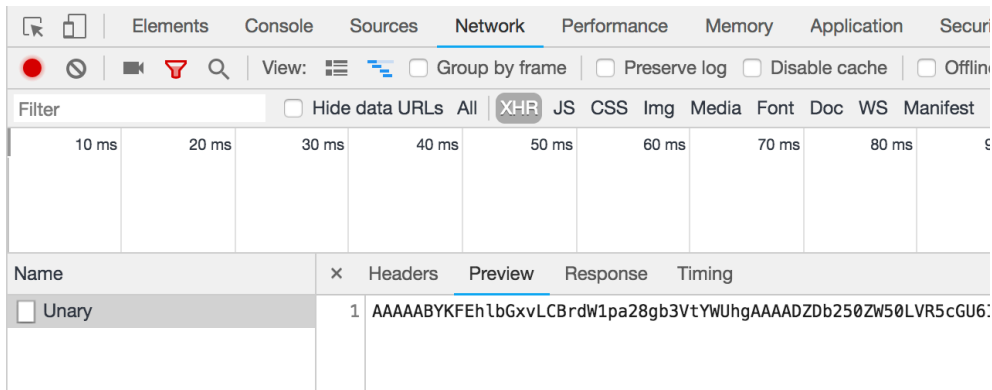
長くなってしまいましたが、今までの知識を元にリクエストペイロードを解読してみます。

```
0a0c 6b75 6d69 6b6f 206f 756d 6165
```

先頭の 0a が key を表していて、これは先程算出した値と同じです。つまり、フィールド番号 1、wire type が 2 であるフィールドが次に続くことがわかります。wire type 2 に対応する型は `string` などが該当するため、次のバイトにはその長さが来ます。0c、すなわち 12 文字です。6b75 6d69 6b6f 206f 756d 6165 を文字列に直すと `kumiko oumae` となります。無事復元できました！

### 1.4.6 レスポンスボディの取得

次にレスポンスの内容を見てみます。Chrome DevTools の Preview タブを見ると、謎の文字列が見えると思います。これをコピーします。



▲ 図 1.4 レスポンスボディ

これがレスポンスの内容です。長過ぎるため、一部 ... で省略しています

```
AAAAABYKFEh1bGxvLCBrdW1pa28gb3VtYWUhgAAAADZDb250Z...
```

gRPC Web<sup>\*11</sup> の仕様を見ると、

The default text encoding is base64

という記載があります。そのため、その文字列は Base64 エンコードされた文字列です。これをデコードします。

```
$ echo -n 'AAAAABYKFEh1bGxvLCBrdW1pa28gb3VtYWUhgAAAADZDb250Z...' | base64 -D | xxd
```

以下が 16 進数表示されたレスポンスです。

```
00000000: 0000 0000 160a 1448 656c 6c6f 2c20 6b75 .....Hello, ku
00000010: 6d69 6b6f 206f 756d 6165 2180 0000 0036 miko oumae!....6
00000020: 436f 6e74 656e 742d 5479 7065 3a20 6170 Content-Type: ap
00000030: 706c 6963 6174 696f 6e2f 6772 7063 2b70 plication/grpc+p
00000040: 726f 746f 0d0a 4772 7063 2d53 7461 7475 roto...Grpc-Statu
00000050: 733a 2030 0d0a                                     s: 0..
```

リクエストの時と同様、先頭から圧縮フラグ・メッセージ長・メッセージと続きます。メッセージ Hello, kumiko oumae! の直後には 80 0000 0036 ... と続きます。

上記のバイナリから何となく分かるように、メッセージの次は Content-Type、Grpc-Status の順にトレイラが続きます。

## 1.5 Server streaming RPC

Server streaming RPC はクライアントは一つだけリクエストを送信し、サーバはストリームを通じて複数のレスポンスを返すような RPC です。

Server streaming RPC のリクエストは Unary RPC とまったく同様です。ではレスポンスはどうなっているのでしょうか？ さっそく実際の挙動を見てみましょう。

再びテスト用のサーバを起動し、Chrome を開きます。run unary ボタンの下にある run server streaming ボタンをクリックすると、いくつかのメッセージが順番に現れます。この一行と一つのレスポンスが対応しています。

<sup>\*11</sup> <https://github.com/grpc/grpc/blob/master/doc/PROTOCOL-WEB.md>

```
run server streaming
```

```
[1] Hello, kumiko oumae!  
[2] Hello, kumiko oumae!  
[3] Hello, kumiko oumae!  
finished
```

▲図 1.5 Server streaming RPC のレスポンス

Unary RPC の時と同様、Preview タブを開くと Base64 エンコードされたレスポンスボディがあります。Timing のタブを開いてみると Unary RPC の時より Content Download の時間が長くなっています。これはリクエストを送信しつつこのタブを監視することでもよく分かると思います。

Server streaming RPC は一つの HTTP リクエストですべてのレスポンスを Content-Type: application/octet-stream で返します。

```
00000000: 0000 0000 1a0a 185b 315d 2048 656c 6c6f .....[1] Hello  
00000010: 2c20 6b75 6d69 6b6f 206f 756d 6165 2100 , kumiko oumae!..  
00000020: 0000 001a 0a18 5b32 5d20 4865 6c6c 6f2c .....[2] Hello,  
00000030: 206b 756d 696b 6f20 6f75 6d61 6521 0000 kumiko oumae!..  
00000040: 0000 1a0a 185b 335d 2048 656c 6c6f 2c20 .....[3] Hello,  
00000050: 6b75 6d69 6b6f 206f 756d 6165 2180 0000 kumiko oumae!..  
00000060: 0036 436f 6e74 656e 742d 5479 7065 3a20 .6Content-Type:  
00000070: 6170 706c 6963 6174 696f 6e2f 6772 7063 application/grpc  
00000080: 2b70 726f 746f 0d0a 4772 7063 2d53 7461 +proto..Grpc-Sta  
00000090: 7475 733a 2030 0d0a tus: 0..
```

ASCII 表示された文字列を眺めてみると、3 つのレスポンスが順番に入っていて、Unary と同様にトレイラが最後に入っていることが確認できます。つまり、Protocol Buffers でエンコードされたレスポンスボディがサーバのレスポンスの数だけ順に格納されています。

## 1.6 Client streaming RPC

Client streaming RPC は Server streaming RPC とは反対にクライアントが複数回リクエストを送信し、最後にサーバがレスポンスを 1 つだけ返します。今までと同様に

Client streaming RPC の挙動を見てみます。run server streaming ボタンの下にある run client streaming ボタンをクリックすると、500 ms ごとに 1 つずつリクエストを送信します。この一行と一つのレスポンスが対応しています。

しかし、Unary や Server streaming とは異なり、以下のようなエラーが表示されてしまいます。

```
✖ ▶ Uncaught Error: No transport available for client-streaming (requestStream) method
    at Object.DefaultTransportFactory (Transport.js:11)
    at GrpcClient.createTransport (client.js:48)
    at new GrpcClient (client.js:27)
    at Object.client (client.js:11)
    at Object.client (index.js:16)
    at HTMLButtonElement.<anonymous> (index.js:42)
```

▲図 1.6 No transport available for client-streaming method

### 1.6.1 WebSocket

実は gRPC Web の仕様では、Client streaming RPC と Bidirectional streaming RPC は現在サポートされていません。これらは将来、WHATWG Streams がサポートされた後に実装される予定です。WHATWG Streams についてはあとの節で紹介します。

gRPC Web のロードマップには HTTP/2 over WebSocket を利用できる Web ブラウザが増えていくに連れ、WebSocket を使った実装が入るかもしれない、といった記述がありますが、今はまだ未実装です。

一方、今回のデモに使用している improbable-eng/grpc-web<sup>\*12</sup> では既に WebSocket を使った Client / Bidirectional streaming がサポートされています。WebSocket を使うには、クライアントとサーバそれぞれで明示的に指定しなければいけません。

client/index.js をエディタで開き、grpc.client メソッドの第 2 引数のオプションに transport を追加し、その値として、grpc.WebsocketTransportFactory を設定します。

```
const client = grpc.client(SimpleService.ClientStreaming, {
  host: host,
  transport: grpc.WebsocketTransportFactory,
});
```

次にサーバの修正を行います。server/main.go をエディタで開き、grpcweb.WrapS

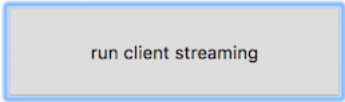
<sup>\*12</sup> <https://github.com/improbable-eng/grpc-web>

server を以下のように変更します。

```
ws := grpcweb.WrapServer(  
    s,  
    grpcweb.WithWebsockets(true),  
    grpcweb.WithWebsocketOriginFunc(func(req *http.Request) bool {  
        return true  
    }),  
)
```

WebSocket を使うためのオプションを有効にし、すべてのオリジンからの接続を許可します。

再度ボタンをクリックすると、今度は正常にリクエストが送信され、結果が表示されます。



run client streaming

[1]: name = oumae  
[2]: name = kousaka  
[3]: name = kato  
[4]: name = kawashima

Hello, oumae, kousaka, kato, kawashima!

## 1.7 Bidirectional streaming RPC

Bidirectional streaming RPC はクライアントが複数回リクエストを送り、サーバが一つのリクエストごとに複数回レスポンスを返すことのできる RPC です。

こちらも Client streaming RPC と同様に WebSocket を使った実装のみ存在しています。動作としては Server streaming RPC + Client streaming RPC なので、省略したいと思います。

## 1.8 HTTP/1.1 or HTTP/2

gRPC Web では HTTP/1.1 または HTTP/2 が使われます。主要な Web ブラウザでは、HTTP/2 over TLS のみをサポートしているため、TLS が使われていない場合、HTTP/1.1 が使用されます。TLS が有効になっている場合、HTTP/2 が使用されます。

仕様にも明記されている通り、gRPC Web は HTTP/2 と HTTP/1.1 をシームレスに使うために HTTP/2 固有の特徴をすべて排除しています。例えばストリームやフレームの ID は gRPC Web では一切使用されず、PING フレームも使用されません。

## 1.9 WHATWG Fetch/Streams

現在、gRPC Web の仕様では XHR (XMLHttpRequest) を使い、Unary RPC と Server streaming RPC のみをサポートしています。ただし、improbable-eng/grpc-web<sup>[\*13]</sup> では Fetch がサポートされているブラウザでは Fetch API が優先的に使用されます。また、前述の通り、WebSocket による Client / Bidirectional streaming RPC のサポートが行われています。

gRPC Web は、将来的に WHATWG Streams<sup>[\*14]</sup> と WHATWG Fetch<sup>[\*15]</sup> で置き換えられることが予定されています。これは、ロードマップの記述<sup>[\*16]</sup>によると、XHR や Fetch API ではストリームを扱うことができないため、Stream API を使ってクライアントのオーバーヘッドを低減するのが目的のようです。しかし、これらの API は、まだ一部の Web ブラウザでしかサポートされていないため当分先になりそうな予感がします。

## 1.10 おわりに

この章では gRPC Web とは何か？ gRPC との差異、特徴や課題点を紹介しました。しかし、まだまだ発展途上の技術であり、実際に遊んで見ることでより実感できると思います。gRPC は多くの良い点があり、その技術が Web ブラウザという大きなプラットフォーム上で動作する可能性があるというのは非常に魅力的です。

gRPC Web は Web ブラウザとともに進化していく技術であり、主要な Web ブラウザが成熟する頃にはきっと強力な技術となっていると思います。

---

<sup>\*13</sup> <https://github.com/improbable-eng/grpc-web>

<sup>\*14</sup> <https://github.com/whatwg/streams>

<sup>\*15</sup> <https://github.com/whatwg/fetch>

<sup>\*16</sup> <https://github.com/grpc/grpc-web/blob/master/ROADMAP.md#streaming-friendly-transport-implementation>



## Go and Kotlin Playground

---

v1.0.0

著 者 GCF members

発行所 GCF

---

(C) 2018 GCF members