Engineer

gRPC

фреймворк для виклику віддалених процедур через HTTP/2

Що це таке

gRPC — це фреймворк для віддаленого виклику процедур. Програма викликає метод на сервері, що працює на іншій машині, так, ніби сервер був локальним об'єктом, а gRPC переносить виклик мережею у власному протоколі поверх HTTP/2 і доставляє відповідь назад як значення, яке повертає метод.

Усе починається з визначення сервісу: методів, які можна викликати ззовні, з типом кожного запиту й кожної відповіді. За замовчуванням його описують за допомогою Protocol Buffers у файлі, назва якого закінчується на `.proto`. Protocol Buffers — це формат Google з відкритим кодом для серіалізації даних, незалежний від мови та платформи. Можна підключити й інші формати, але без окремих вказівок gRPC використовує саме protocol buffers. Компілятор `protoc` разом із плагіном gRPC читає цей файл і генерує код для обох половин. Клієнт отримує stub — локальний об'єкт із тими самими методами, що й сервіс (у деяких мовах його називають просто клієнтом), а сервер отримує інтерфейс, який треба реалізувати. Оскільки обидві половини походять з одного файлу, сервер, написаний на Java, може відповідати клієнтам, написаним на Go, Python чи Ruby.

Метод має одну з чотирьох форм. Унарний метод приймає один запит і повертає одну відповідь, як звичайний виклик функції. Метод із серверним потоком приймає один запит і надсилає назад потік відповідей, метод із клієнтським потоком робить навпаки, а двонаправлений потоковий метод дає обом сторонам надсилати свої потоки незалежно одна від одної. У межах одного виклику gRPC гарантує, що повідомлення надходять у тому порядку, в якому їх надіслали.

На рівні мережі gRPC працює поверх HTTP/2. Кожен виклик — це `POST` на шлях, що складається з повного імені сервісу та назви методу, з типом вмісту `application/grpc`. Кожне повідомлення йде за п'ятибайтовим префіксом: один байт каже, чи стиснуте повідомлення, а чотири задають його довжину. Результат виклику — це не HTTP-статус. Відповідь, яку формує сам gRPC, має статус 200, навіть коли виклик завершився помилкою, а справжній результат приходить у трейлерах, тобто полях заголовків, які надсилають після тіла: `grpc-status` із числом і, за потреби, `grpc-message` з описом помилки.

Через це gRPC — зовсім інша річ, ніж REST. REST API називає ресурси й працює з ними власними методами та кодами статусу HTTP, зазвичай із тілами в JSON, які може прочитати будь-хто. gRPC API — це перелік методів із типізованими двійковими повідомленнями, а HTTP/2 — лише транспорт під ними. Документація Protocol Buffers сама підсумовує свій формат так: «як JSON, тільки менший і швидший».

Застосуйте поняття до рішення

Застосуйте «Що таке gRPC?» до конкретного рішення. Ми можемо пов'язати поняття із системою, яку ви створюєте або купуєте. Та частина, якої користувач не бачить: де зберігаються дані, які діють правила і як інші системи можуть із нею взаємодіяти.

Переглянути наші послуги Зв'язатися з нами →

Перевірте припущення за терміном

Використайте пояснення та пов'язані роботи, щоб сформулювати запитання для проєкту. Ми можемо перевірити припущення разом до вибору інструмента. Спроєктувала й підтримувала 6 пайплайнів ETL/ELT — 6 пайплайнів ETL на трьох рушіях баз даних

Спроєктувала й підтримувала 6 пайплайнів ETL/ELT Зв'язатися з нами →

Чому це важливо

Спершу контракт, а код випливає з нього. У багатьох HTTP API опис і код, що обслуговує API, — дві окремі речі, які можуть розійтися, а клієнта часто пишуть вручну за документом. У gRPC файл `.proto` — єдине місце, де визначають метод чи поле, і з нього генерують усіх клієнтів і всі сервери. Зміна повідомлення доходить до кожної мови через той самий генератор, а не через те, що хтось перечитав документ.

Кодування компактне, бо кожне поле передається як його номер і значення, а не як назва. Ціна в тому, що повідомлення неможливо повністю розтлумачити без його файлу `.proto`, адже protocol buffers не описують самі себе, і що формат розрахований на те, що ціле повідомлення вміщується в пам'ять за один раз. Власна документація формату радить інше рішення для даних, більших за кілька мегабайтів.

Потокова передача — частина задуму, а не надбудова. Виклики йдуть як потоки в з'єднаннях HTTP/2, тож метод із серверним потоком може й далі надсилати результати тим самим механізмом, що й виклик, який відповідає один раз.

Обмеження часу супроводжує виклик. Клієнт задає дедлайн, gRPC передає залишок часу в заголовку `grpc-timeout`, а коли час спливає, клієнт отримує `DEADLINE_EXCEEDED`, а виклик на боці сервера скасовується. Коли сервер сам викликає інший сервер, деякі реалізації передають дедлайн далі: Java і Go роблять це за замовчуванням, а в C++ це треба ввімкнути. Оскільки дедлайн перетинає кожну ланку як залишок часу, а не як позначка часу на годиннику, машини з розбіжними годинниками однаково припиняють чекати в той самий момент.

Помилки мають спільний словник. gRPC визначає 17 кодів статусу, від 0 `OK` до 16 `UNAUTHENTICATED`, і вони є частиною самого протоколу, тож клієнт на Go і сервер на Java однаково розуміють, що означає `NOT_FOUND` чи `UNAVAILABLE`.

Усе це добре підходить для трафіку між сервісами, які контролює одна команда чи одна компанія, де обидва кінці можуть ділитися файлами `.proto` і збиратися з них. Для браузера це підходить гірше. Власні нотатки проєкту gRPC про протокол кажуть, що браузери не дають скриптам доступу до фреймів HTTP/2 і ніколи не даватимуть, тому код у браузері розмовляє окремим протоколом, gRPC-Web, і зазвичай проксі перекладає його на нативний gRPC.

Як це виглядає на практиці

Ось невеликий сервіс виставлення рахунків. Унарний метод отримує один рахунок, а метод із серверним потоком надсилає рахунки замовника по одному:

```
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;
}
```

Числа тут — не значення за замовчуванням. Це номери полів, і в двійковому кодуванні вони ідентифікують кожне поле замість його назви. Номери від 1 до 15 займають один байт, тож їх варто віддавати полям, які заповнюють найчастіше. Два рядки `reserved` фіксують поле, яке колись існувало, номер 4 з назвою `discount_code`, щоб ні номер, ні назва не повернулися як інше поле.

Для Python одна команда генерує `billing_pb2.py` з класами повідомлень і `billing_pb2_grpc.py` з класами клієнта й сервера:

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

Один виклик `GetInvoice` у фреймах HTTP/2 тоді виглядає так:

```
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` означає дві секунди, а `te = trailers` потрібен, щоб виявляти проксі, несумісні з gRPC. Якщо рахунку не існує, відповідь однаково починається з `:status = 200`, а останній фрейм несе `grpc-status = 5`, тобто `NOT_FOUND`, разом із `grpc-message` на кшталт `invoice not found`. Сервер, який одразу завершує виклик помилкою, може навіть надіслати єдиний фрейм `HEADERS` зі статусом і без жодного повідомлення.

Генерувати обидві сторони з одного контракту — не винахід gRPC. В одному проєкті з портфоліо сайту контракт API згенерували з бази даних назовні для API, описаного в OpenAPI. Ланцюг починається з бази даних і тягнеться назовні: типи Go визначають API, Huma формує з них опис OpenAPI, з цього опису генеруються типізований клієнт на TypeScript і mock-обробники, а перевірка контракту в CI зупиняє збірку, коли те, що надсилає фронтенд, перестає збігатися з тим, чого чекає API. gRPC вбудовує ту саму дисципліну у свій інструментарій, де єдиним джерелом є файл `.proto`.

Де виникають проблеми

Виклик без дедлайну може чекати вічно. За замовчуванням gRPC не задає дедлайну, тож клієнт, який викликає сервер, що завис, може чекати відповіді фактично вічно. Задавайте кожному виклику реалістичний дедлайн. У Python це аргумент `timeout` у секундах:

```
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)
```

Коли час спливає, виклик кидає `grpc.RpcError`, а метод `code()` цього винятку повертає `grpc.StatusCode.DEADLINE_EXCEEDED`. На сервері gRPC скасовує виклик із `CANCELLED`, щойно дедлайн минув, але зупинити роботу, яку запустив обробник, має власний код сервісу, тож обробник, що працює довго, мусить сам перевіряти, чи виклик скасовано, і зупинятися.

Таймаут не означає, що нічого не сталося. Клієнт і сервер кожен сам вирішує, чи вдався виклик, тож виклик може успішно завершитися на сервері й водночас упасти на клієнті з `DEADLINE_EXCEEDED`. Зміни, зроблені до скасування, теж не відкочуються. Клієнт, який просто повторює спробу, може створити той самий рахунок двічі. Протокол не вважає виклик ідемпотентним, якщо його явно так не позначено. `UNAVAILABLE` — код, який найчастіше означає тимчасовий збій і за якого варто повторити спробу з backoff, але документація кодів статусу попереджає, що повторювати неідемпотентні операції не завжди безпечно. Той самий документ покладає на код, що робить виклики, рішення, за яких кодів повторювати спробу.

Повторно використаний номер поля псує дані. Номер поля не можна змінювати, коли тип повідомлення вже використовується. Якщо поле видалити, а його номер згодом віддати новому полю, старіші клієнти й збережені повідомлення декодують нове поле з байтів, які означали щось інше. Посібник з Protocol Buffers перелічує можливі наслідки: помилки розбору, витік персональних даних і пошкоджені дані. Ніколи не перенумеровуйте поле і резервуйте кожен видалений номер і кожну назву, як у прикладі вище.

Кожна помилка для HTTP виглядає як успіх. Оскільки gRPC відповідає HTTP-статусом 200, навіть коли виклик завершився помилкою, усе, що оцінює трафік лише за HTTP-статусом, наприклад журнал доступу чи частка помилок, побудована на кодах статусу, зараховує збої до успіхів. Специфікація саме з цієї причини каже серверам відповідати 415 на запити з типом вмісту, відмінним від `application/grpc`, щоб інші клієнти HTTP/2 не прочитали помилку gRPC як успіх. Рахуйте за `grpc-status`, а не за `:status`.

Довгі потоки прив’язують навантаження до одного сервера. Виклики йдуть як потоки в спільному з’єднанні HTTP/2, і щойно потік почався, його не можна перенести на інший сервер. Потік, що живе годинами, тримає своє навантаження там, де почався, хоч скільки серверів додадуть пізніше. Кожне з’єднання також має ліміт одночасних потоків, і коли його досягнуто, наступні виклики чекають на клієнті, доки якийсь потік не завершиться.

Посібник gRPC з продуктивності радить використовувати потоки лише там, де вони дають суттєвий виграш у продуктивності чи простоті, і виділяти окремий канал для кожної ділянки з високим навантаженням.

Великі обсяги даних не вміщуються в одне повідомлення. gRPC обмежує максимальний розмір повідомлення, і повідомлення, більше за налаштований ліміт, відхиляється з `RESOURCE_EXHAUSTED` на тій стороні, яка його надсилає чи отримує. Підвищення ліміту лише переносить проблему, бо protocol buffers очікують усе повідомлення в пам'яті, а з великими даними там одночасно може опинитися кілька серіалізованих копій. Розбивайте великі дані на потік менших повідомлень або виносьте їх за межі gRPC.

Що почитати далі

- Core concepts, architecture and lifecycle — визначення сервісів, чотири види методів, stubs, дедлайни та скасування.
- gRPC over HTTP2 — протокол на рівні мережі: шляхи, заголовки, префікс повідомлення і статус у трейлерах.
- Status Codes — усі 17 кодів, що означає кожен і які з них сама бібліотека ніколи не формує.
- Deadlines — чому немає значення за замовчуванням, як дедлайни доходять до сервера і як вони передаються між сервісами.
- Language Guide (proto 3) — номери полів, зарезервовані поля і правила безпечної зміни типу повідомлення.

Словник

Обирайте за фактичними умовами

Опишіть робочий процес, наявні інтерфейси та користувачів, яким потрібна зміна. Ми можемо визначити перевірюваний результат і спосіб передачі, який дасть вашій команді змогу підтримувати його.

Дивитися, скільки це коштує Зв'язатися з нами →

Зробіть термінологію практичною

Запитайте, як «Що таке gRPC?» стосується вашої організації. Конкретного прикладу з поточної системи достатньо для початку розмови.

Переглянути наші послуги

welcome@engineer.company +45 50 10 61 84 Копенгаген, Данія

Візьміть цю сторінку із собою

Візитівка

Дбайливо зроблено з любов'ю — D.

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

Авторське право © 2025 – дотепер · Engineer ApS · Усі права захищено.

Надруковано з https://engineer.company/uk/glossary/grpc/