RPC远程调用

重点内容

1. 单机要求

- 需要事先约定调用的语义(接口语法)
- 需要网络传输
- 需要约定网络传输中的内容格式

2. 分布式要求

- 注册发现服务
- 负载均衡

3. gRPC实践

- 同步调用
- 异步调用

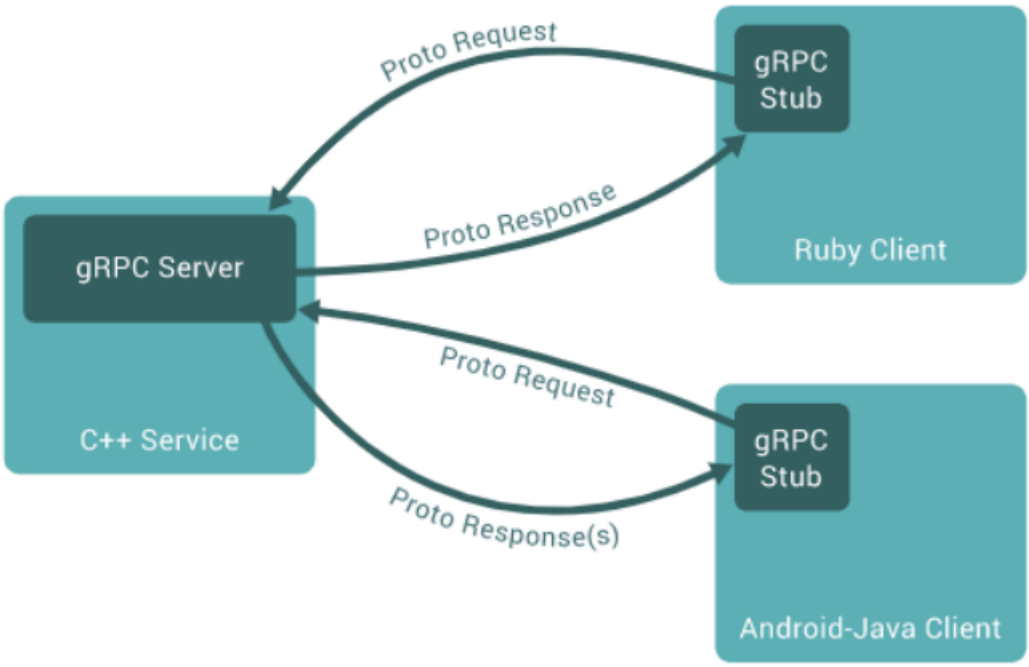
1. gRPC原理

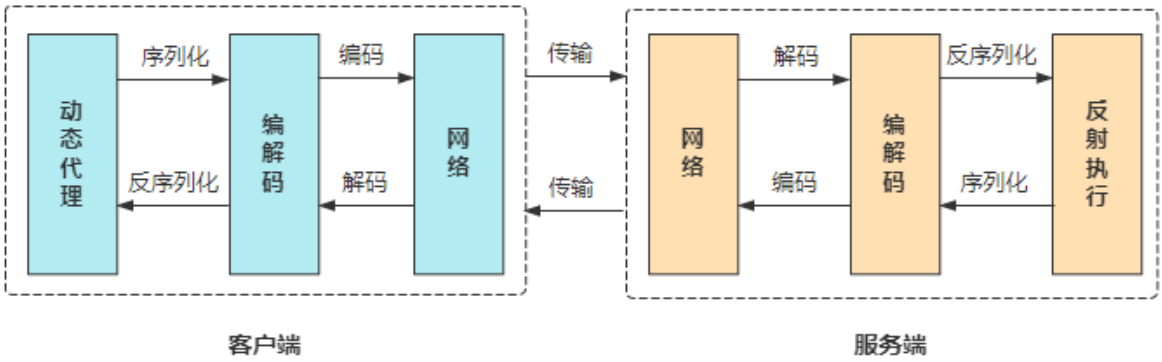
grpc: https://grpc.io/docs/what-is-grpc/faq/

1.1 什么是RPC

RPC 即远程过程调用协议（Remote Procedure Call Protocol），可以让我们像调用本地对象一样发起远程调用。RPC 凭借其强大的治理功能，成为解决分布式系统通信问题的一大利器。

gRPC是一个现代的、高性能、开源的和语言无关的通用 RPC 框架，基于 HTTP2 协议设计，序列化使用 PB(Protocol Buffer)，PB 是一种语言无关的高性能序列化框架，基于 HTTP2+PB 保证了的高性能。





1.2 gRPC的一些特性

1. gRPC基于服务的思想：定义一个服务，描述这个服务的方法以及入参出参，服务器端有这个服务的具体实现，客户端保有一个存根，提供与服务端相同的服务
2. gRPC默认采用protocol buffer作为IDL(Interface Description Lanage)接口描述语言,服务之间通信的数据序列化和反序列化也是基于protocol buffer的，因为protocol buffer的特殊性，所以gRPC框架是跨语言的通信框架(与编程语言无关性)，也就是说用Java开发的基于gRPC的服务，可以用GoLang编程语言调用
3. gRPC同时支持同步调用和异步调用，同步RPC调用时会一直阻塞直到服务端处理完成返回结果，异步RPC是客户端调用服务端时不等待服务段处理完成返回，而是服务端处理完成后主动回调客户端告诉客户端处理完成
4. gRPC是基于http2协议实现的，http2协议提供了很多新的特性，并且在性能上也比http1提搞了许多，所以gRPC的性能是非常好的
5. gRPC并没有直接实现负载均衡和服务发现的功能，但是已经提供了自己的设计思路。已经为命名解析和负载均衡提供了接口
6. 基于http2协议的特性：gRPC允许定义如下四类服务方法
    1. 一元RPC：客户端发送一次请求，等待服务端响应结构，会话结束，就像一次普通的函数调用这样简单
    2. 服务端流式RPC：客户端发起一起请求，服务端会返回一个流，客户端会从流中读取一系列消息，直到没有结果为止
    3. 客户端流式RPC：客户端提供一个数据流并写入消息发给服务端，一旦客户端发送完毕，就等待服务器读取这些消息并返回应答

4. 双向流式RPC：客户端和服务端都一个数据流，都可以通过各自的流进行读写数据，这两个流是相互独立的，客户端和服务端都可以按其希望的任意顺序独写

1.3 gRPC支持的编程语言

C ++，Java(包括对Android的支持)，Objective-C(对于iOS)，Python，Ruby，Go，C＃，Node.js都在GA中，并遵循语义版本控制。

1.4 gRPC的使用场景

低延迟，高度可扩展的分布式系统

开发与云服务器通信的客户端

设计一个准确，高效，且与语言无关的新协议时

分层设计，以实现扩展，例如。身份验证，负载平衡，日志记录和监控等

1.5 谁在使用gRPC

谷歌长期以来一直在gRPC中使用很多基础技术和概念。目前正在谷歌的几个云产品和谷歌面向外部的API中使用。Square，Netflix，CoreOS，Docker，CockroachDB，Cisco，Juniper Networks以及许多其他组织和个人也在使用它。

1.6 gRPC设计之初的动机和原则

1. **自由，开放**：让所有人，所有平台都能使用，其实就是开源，跨平台，跨语言
2. **协议可插拔**：不同的服务可能需要使用不同的消息通信类型和编码机制，例如，JSON、XML和Thirft,所以协议应允许可插拔机制，还有负载均衡，服务发现，日志，监控等都支持可插拔机制
3. **阻塞和非阻塞**：支持客户端和服务器交换的消息序列的异步和同步处理。这对于在某些平台上扩展和处理至关重要
4. **取消和超时**：一次RPC操作可能是持久并且昂贵的，应该允许客户端设置取消RPC通信和对这次通信加上一个超时时间
5. **拒绝**：必须允许服务器通过在继续处理请求的同时拒绝新请求的到来并优雅地关闭。
6. **流处理**：存储系统依靠流和流控制来表达大型数据集，其他服务，如语音到文本或股票行情，依赖于流来表示与时间相关的消息序列
7. **流控制**：计算能力和网络容量在客户端和服务器之间通常是不平衡的。流控制允许更好的缓冲区管理，以及过度活跃的对等体提供对DOS的保护。
8. **元数据交换**：认证或跟踪等常见的跨领域问题依赖于不属于服务声明接口的数据交换。依赖于他们将这些特性演进到服务，暴露API来提供能力。
9. **标准化状态码**：客户端通常以有限的方式响应API调用返回的错误。应约束状态码名称空间，以使这些错误处理决策更加清晰。如果需要更丰富的特定领域的状态，则可以使用元数据交换机制来提供该状态。
10. **互通性**：报文协议(Wire Protocol)必须遵循普通互联网基础框架

信息来源 https://grpc.io/faq/

2 数据封装和数据传输问题

2.1 网络传输中的内容封装数据体积问题

早期的RPCJSON的方式，目前的RPC基本上都采用类似Protobuf的二进制序列化方式。

其差别在于：**json的设计是给人看的，protobuf则是利于机器。**

JSON

优点：在body中用JSON对内容进行编码，极易跨语言，不需要约定特定的复杂编码格式和Stub文件。在版本兼容性上非常友好，扩展也很容易。

缺点：JSON难以表达复杂的参数类型，如结构体等；数据冗余和低压缩率使得传输性能差。

Protobuf

gRPC对此的解决方案是丢弃json、xml这种传统策略，使用 Protocol Buffer，是Google开发的一种跨语言、跨平台、可扩展的用于序列化数据协议。

```
// XXXX.proto
service Test {
   rpc HowRpcDefine (Request) returns (Response) ; // 定义一个RPC方法
}
message Request {
   //类型 | 字段名字|  标号
   int64   user_id  = 1;
   string  name    = 2;
}
message Response {
   repeated int64 ids = 1; // repeated 表示数组
   Value info = 2;      // 可嵌套对象
   map<int, Value> values = 3;    // 可输出map映射
}
message Value {
   bool is_man = 1;
   int age = 2;
}
```

以上是一个使用样例，包含方法定义、入参、出参。可以看出有几个明确的特点：

- 有明确的类型，支持的类型有多种
- 每个field会有名字
- 每个field有一个**数字标号**，一般按顺序排列(下文编解码会用到这个点)
- 能表达数组、map映射等类型
- 通过嵌套message可以表达复杂的对象
- 方法、参数的定义落到一个.proto 文件中，**依赖双方需要同时持有这个文件，并依此进行编解码**

protobuf作为一个以跨语言为目标的序列化方案，protobuf能做到多种语言以同一份proto文件作为约定，不用A语言写一份，B语言写一份，各个依赖的服务将proto文件原样拷贝一份即可。

但.proto文件并不是代码，不能执行，要想直接跨语言是不行的，必须得有对应语言的中间代码才行，中间代码要有以下能力：

- 将message转成对象，例如C++里是class，golang里是struct，需要各自表达后，才能被理解
- 需要有进行编解码的代码，能解码内容为自己语言的对象、能将对象编码为对应的数据

更多protobuf的讲解请参考 课程《应用层协议设计protobuf》。

2.2 网络传输效率问题

grpc采用HTTP2.0，相对于HTTP1.0 在`更快的传输`和`更低的成本`两个目标上做了改进。有以下几个基本点：

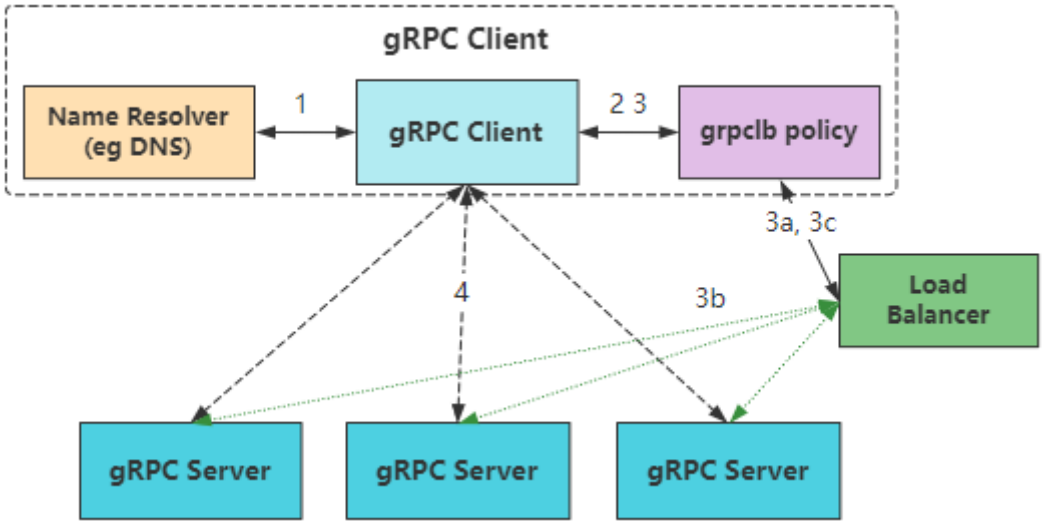
- HTTP2 未改变HTTP的语义(如GET/POST等)，只是在传输上做了优化
- 引入帧、流的概念，在TCP连接中，可以区分出多个request/response
- 一个域名只会有一个TCP连接，借助帧、流可以实现多路复用，降低资源消耗
- 引入二进制编码，降低header带来的空间占用

HTTP1.0核心问题在于：在同一个TCP连接中，没办法区分response是属于哪个请求，一旦多个请求返回的文本内容混在一起，则没法区分数据归属于哪个请求，所以请求只能一个个串行排队发送。这直接导致了TCP资源的闲置。

HTTP2为了解决这个问题，提出了`流`的概念，每一次请求对应一个流，有一个唯一ID，用来区分不同的请求。基于流的概念，进一步提出了`帧`，一个请求的数据会被分成多个帧，方便进行数据分割传输，每个帧都唯一属于某一个流ID，将帧按照流ID进行分组，即可分离出不同的请求。这样同一个TCP连接中就可以同时并发多个请求，不同请求的帧数据可穿插在一起，根据流ID分组即可。**这样直接解决了HTTP1.0的核心痛点，通过这种复用TCP连接的方式，不用再同时建多个连接，提升了TCP的利用效率。**

3 RPC注册发现

gRPC开源组件官方并未直接提供服务注册与发现的功能实现，但其设计文档已提供实现的思路，并在不同语言的gRPC代码API中已提供了命名解析和负载均衡接口供扩展。



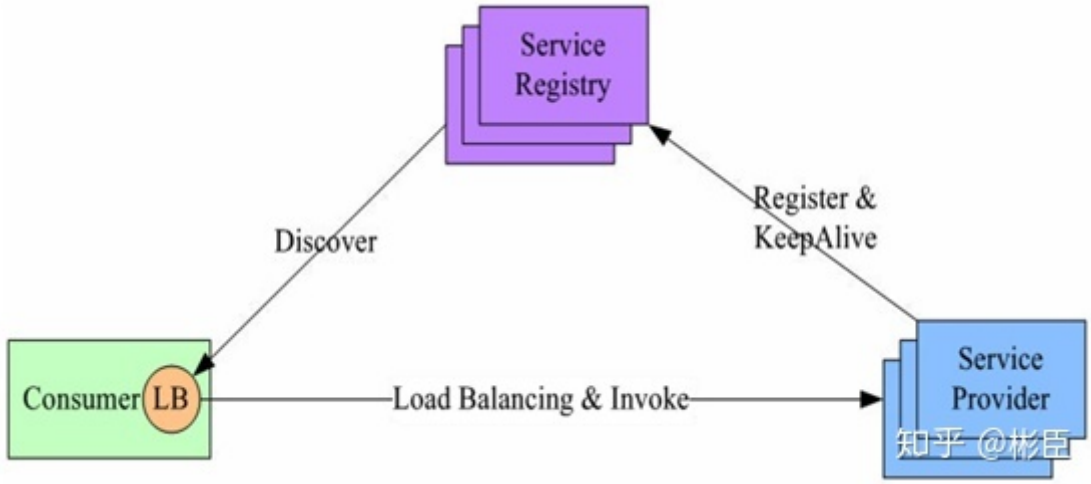
其实现基本原理：

1. 服务启动后gRPC客户端向命名服务器发出名称解析请求，名称将解析为一个或多个IP地址，每个IP地址标示它是**服务器地址**还是**负载均衡器地址**，以及标示要使用那个客户端负载均衡策略或服务配置。
2. 客户端实例化负载均衡策略，如果解析返回的地址是负载均衡器地址，则客户端将使用grpclb策略，否则客户端使用服务配置请求的负载均衡策略。
3. 负载均衡策略为每个服务器地址创建一个子通道（channel）。

4. 当有RPC请求时，负载均衡策略决定那个子通道即 gRPC 服务器将接收请求，当可用服务器为空时客户端的请求将被阻塞。

根据gRPC官方提供的设计思路，基于进程内LB方案（阿里开源的服务框架 Dubbo 也是采用类似机制），结合分布式一致的组件（如Zookeeper、Consul、Etcd），可找到gRPC服务发现和负载均衡的可行解决方案。

PS: 进程内LB（Balancing-aware Client）方案

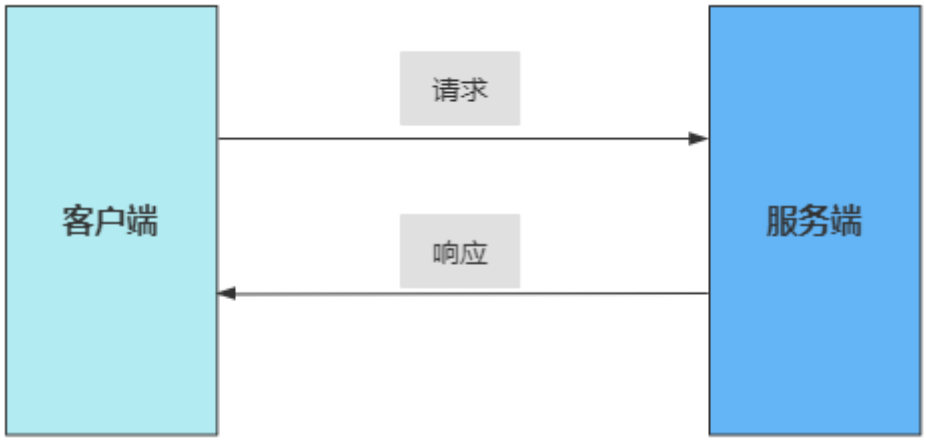


此方案将LB的功能集成到服务消费方进程里，也被称为软负载或者客户端负载方案。服务提供方启动时，首先将服务地址注册到服务注册表，同时定期报心跳到服务注册表以表明服务的存活状态，相当于健康检查，服务消费方要访问某个服务时，它通过内置的LB组件向服务注册表查询，同时缓存并定期刷新目标服务地址列表，然后以某种负载均衡策略 (Round Robin, Random, etc) 选择一个目标服务地址，最后向目标服务发起请求。LB和服务发现能力被分散到每一个服务消费者的进程内部，同时服务消费方和服务提供方之间是直接调用，没有额外开销，性能比较好。

4 GRPC 4种模式

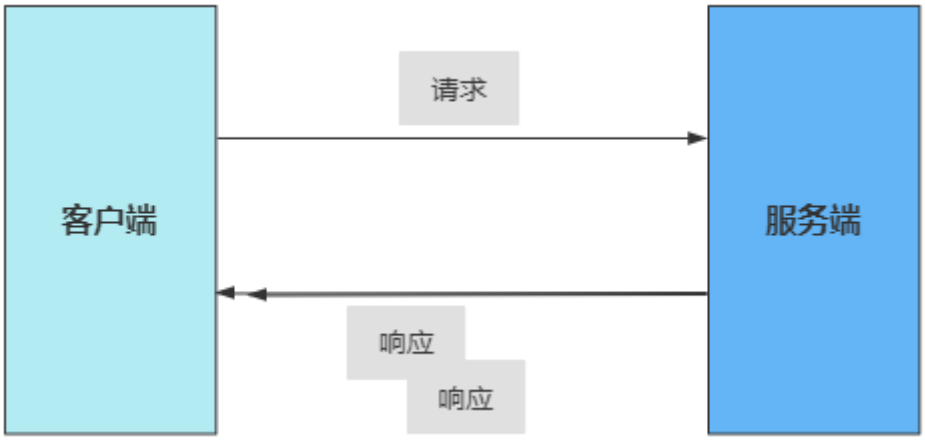
1. 一元RPC模式

一元 RPC 模式也被称为**简单 RPC 模式。**在该模式中，当客户端调用服务器端的远程方法时，客户端发送请求至服务器端并获得一个响应，与响应一起发送的还有状态细节以及 trailer 元数据。



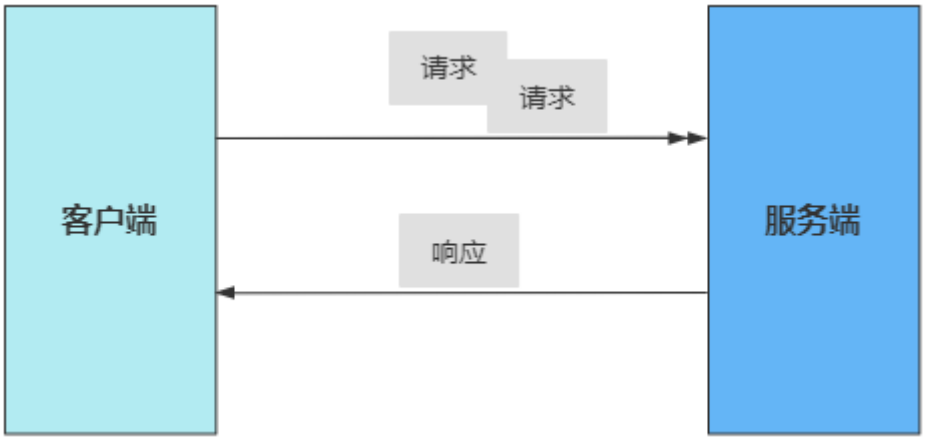
2. 服务器端流RPC模式

在一元 RPC 模式中，gRPC 服务器端和 gRPC 客户端在通信时始终只有一个请求和一个响应。在服务器端流 RPC 模式中，服务器端在接收到客户端的请求消息后，会发回一个响应的序列。这种多个响应所组成的序列也被称为“流”。在将所有的服务器端响应发送完毕之后，服务器端会以 trailer 元数据的形式将其状态发送给客户端，从而标记流的结束。



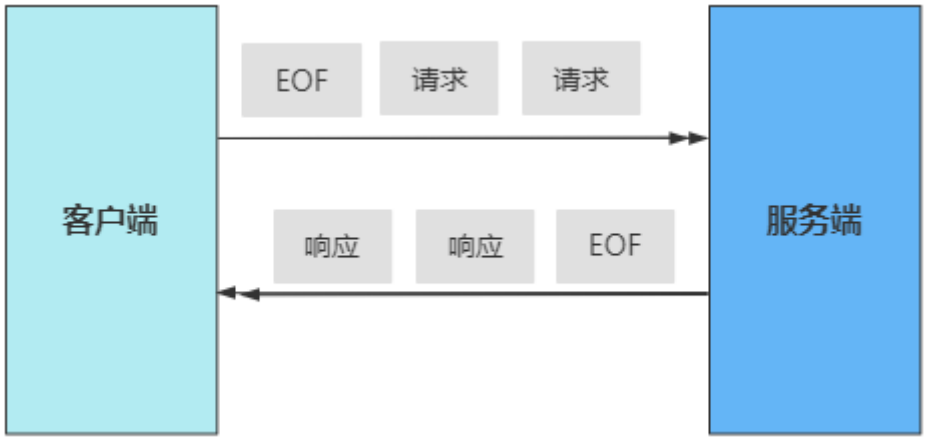
3. 客户端流RPC模式

在客户端流 RPC 模式中，客户端会发送多个请求给服务器端，而不再是单个请求。服务器端则会发送一个响应给客户端。但是，服务器端不一定要等到从客户端接收到所有消息后才发送响应。基于这样的逻辑，我们可以在接收到流中的一条消息或几条消息之后就发送响应，也可以在读取完流中的所有消息之后再发送响应。



4. 双向流RPC模式

在双向流 RPC 模式中，客户端以消息流的形式发送请求到服务器端，服务器端也以消息流的形式进行响应。调用必须由客户端发起，但在此之后，通信完全基于 gRPC 客户端和服务器端的应用程序逻辑。



5 GRPC异步同步

具体参考：examples/cpp/helloworld

代码分析见课堂讲解。

如果一步步把RPC集成到自己项目，见课堂讲解。