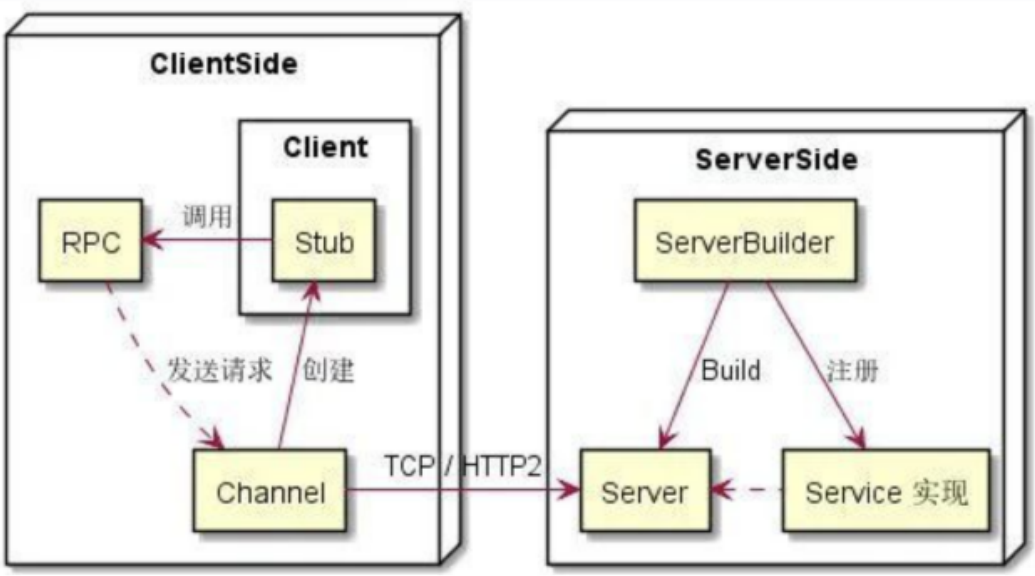
零基础入门gPRC（二）

五、gRPC详解

gRPC核心概念



上图中列出了 gRPC 基础概念及其关系图。其中包括：**Service(定义)、RPC、API、Client、Stub、Channel、Server、Service(实现)、ServiceBuilder** 等。

`.proto` 文件定义了**服务** RPCDateService和 **API** getDate：

Protobuf | 复制代码

```
// 服务接口.定义请求参数和相应结果
service RPCDateService {
    rpc getDate (RPCDateRequest) returns (RPCDateResponse) {
    }
}
```

`class` GRPCClient 是 **Client**，是对 **Stub** 封装；通过 **Stub** 可以真正的调用 RPC 请求。

Java | 复制代码

```java
//1,拿到一个通信channel
        ManagedChannel channel = ManagedChannelBuilder.forAddre
                usePlaintext()//无需加密或认证
                .build();
        try {
            //2.拿到stub对象
            RPCDateServiceGrpc.RPCDateServiceBlockingStub rpcDa
            RPCDateRequest rpcDateRequest = RPCDateRequest.newB
                    .setUserName("JACK")
                    .build();
            //3,请求
            RPCDateResponse rpcDateResponse = rpcDateService.ge
            //4,输出结果
            System.out.println(rpcDateResponse.getServerDate())
        } finally {
            // 5.关闭channel, 释放资源.
            channel.shutdown();
        }
```

Channel 提供一个与特定 gRPC server 的主机和端口建立的连接。
Stub 就是在 **Channel** 的基础上创建而成的。

Server 端需要实现对应的 RPC，所有的 RPC 组成了 **Service**：

Java | 复制代码

```java
// RPCDateServiceGrpc.RPCDateServiceImplBase 这个就是接口.
// RPCDateServiceImpl 我们需要继承他的,实现方法回调
public class RPCDateServiceImpl extends RPCDateServiceGrpc.RPCD
    @Override
    public void getDate(RPCDateRequest request, StreamObserver<
        //请求结果，我们定义的
        RPCDateResponse rpcDateResponse = null;
        //
        String userName = request.getUserName();
        String response = String.format("你好:%s,今天是%s.", use
        try {
            // 定义响应,是一个builder构造器.
            rpcDateResponse = RPCDateResponse.newBuilder()
                    .setServerDate(response)
                    .build();
            //int i = 10/0;
        } catch (Exception e) {
            responseObserver.onError(e);
        } finally {
            // 这种写法是observer,异步写法,
            responseObserver.onNext(rpcDateResponse);
        }

        responseObserver.onCompleted();

    }
}
```

Server 的创建需要一个 **Builder**，添加上监听的地址和端口，**注册**上该端口上绑定的服务，最后构建出 Server 并启动：

Java | 复制代码

```java
public class GRPCServer {
    private static final int port = 9999;

    public static void main(String[] args) throws IOException,
        //设置service端口
        Server server = ServerBuilder.forPort(port)
                .addService(new RPCDateServiceImpl())
                .build().start();
        System.out.println(String.format("GRpc服务端启动成功, 端

        server.awaitTermination();


    }
}

```

RPC 和 API 的区别：RPC (Remote Procedure Call) 是一次远程过程调用的整个动作，而 API (Application Programming Interface) 是不同语言在实现 RPC 中的具体接口。一个 RPC 可能对应多种 API，比如同步的、异步的、回调的。一次 RPC 是对某个 API 的一次调用。

使用 http2 协议

为什么会选用 http2 作为 gRPC 的传输协议?

除了速度之外，最大的原因就是最大程度的服务兼容性。因为 gRPC 基于 http2 协议，加之市面上主流的代理工具也都支持 http2 协议，所以自然就支持 gRPC 了。

HTTP/1.x 的缺陷

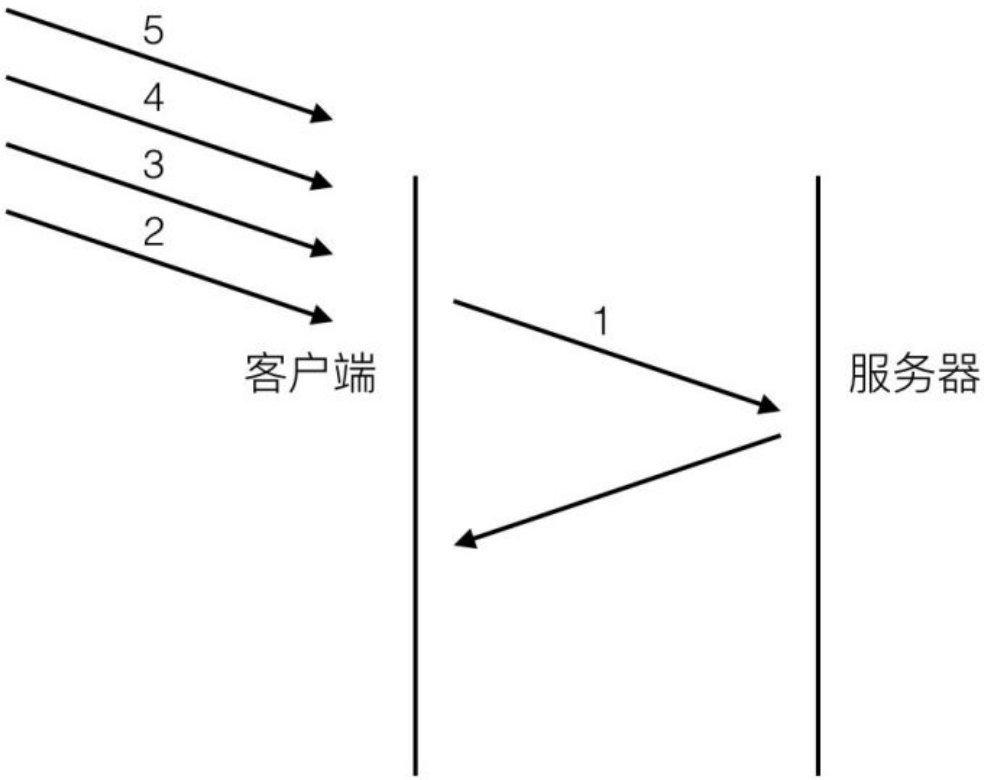
连接无法复用

连接无法复用会导致每次请求都经历三次握手和慢启动。三次握手在高延迟的场景下影响较明显，慢启动则对大量小文件请求影响较大（没有达到最大窗口请求就被终止）。

- HTTP/1.0 传输数据时，每次都需要重新建立连接，增加延迟。
- HTTP/1.1 虽然加入 keep-alive 可以复用一部分连接，但域名分片等情况下仍然需要建立多个 connection，耗费资源，给服务器带来性能压力。

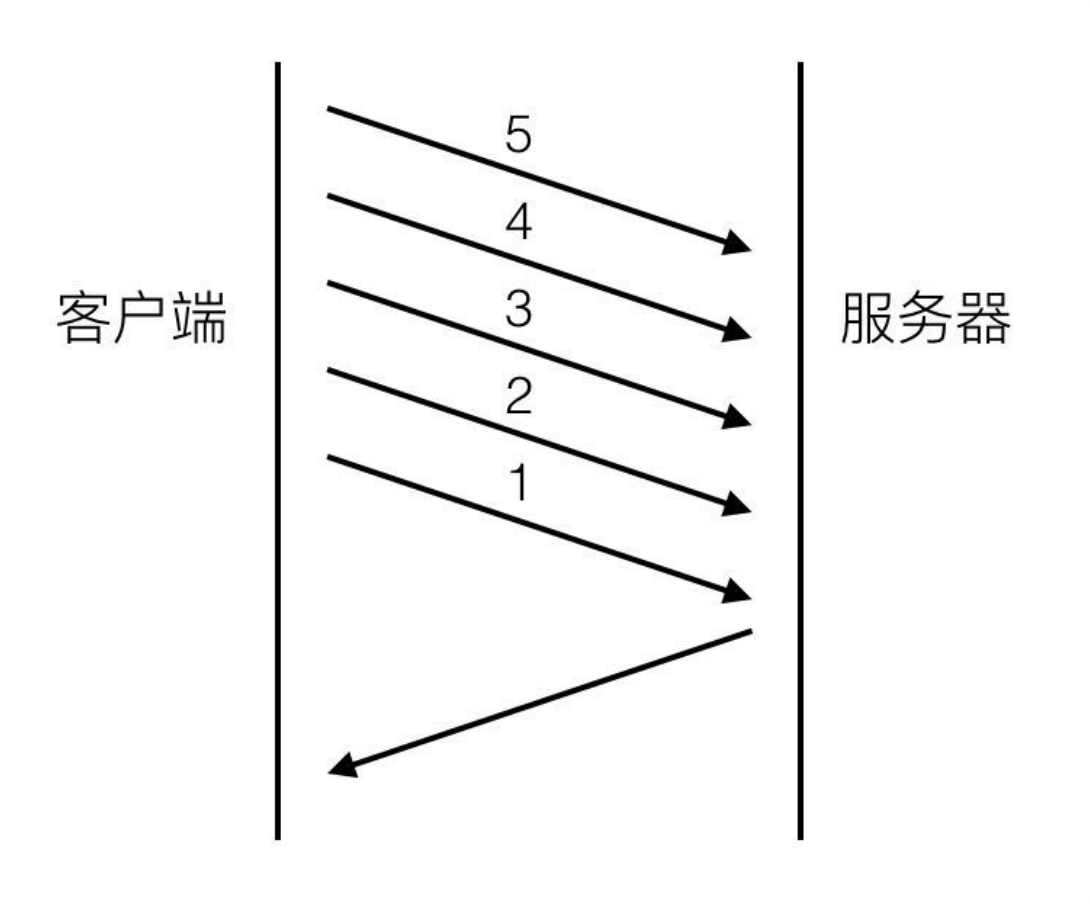
head-of-line blocking（线头阻塞）

会导致带宽无法被充分利用，以及后续健康请求被阻塞。假设有5个请求同时发出，如下图



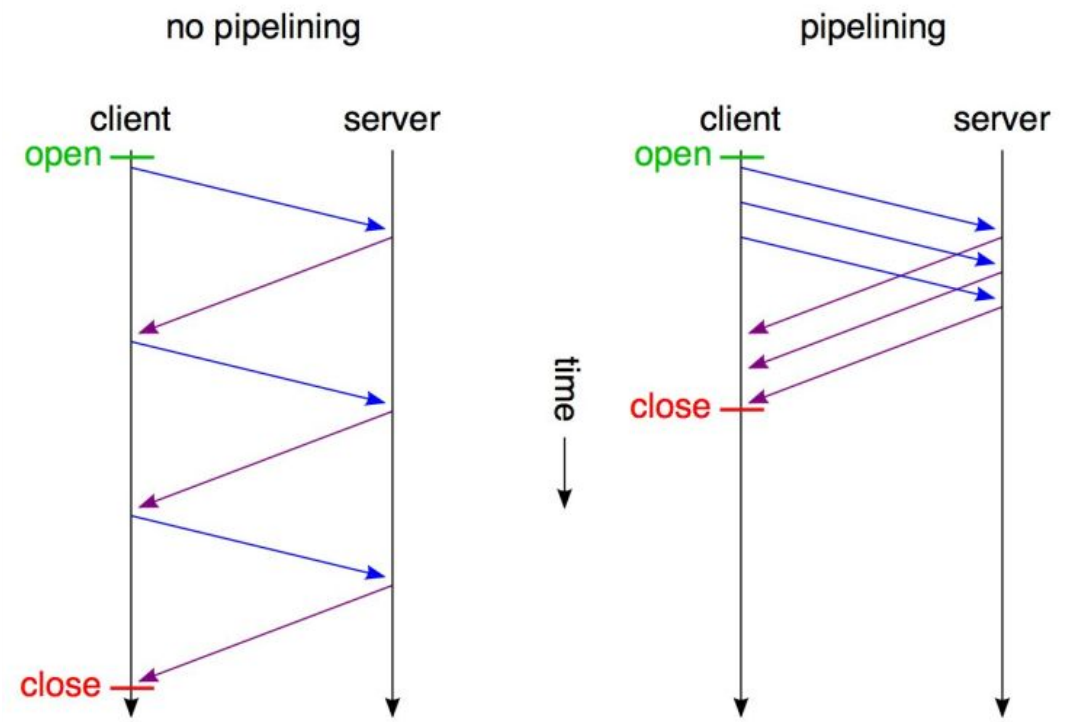
对于http1.0的实现，在第一个请求没有收到回复之前，后续从应用层发出的请求只能排队，请求2，3，4，5只能等请求1的response回来之后才能逐个发出。

在http1.1中，引入了pipeline来解决head of line blocking，如下图

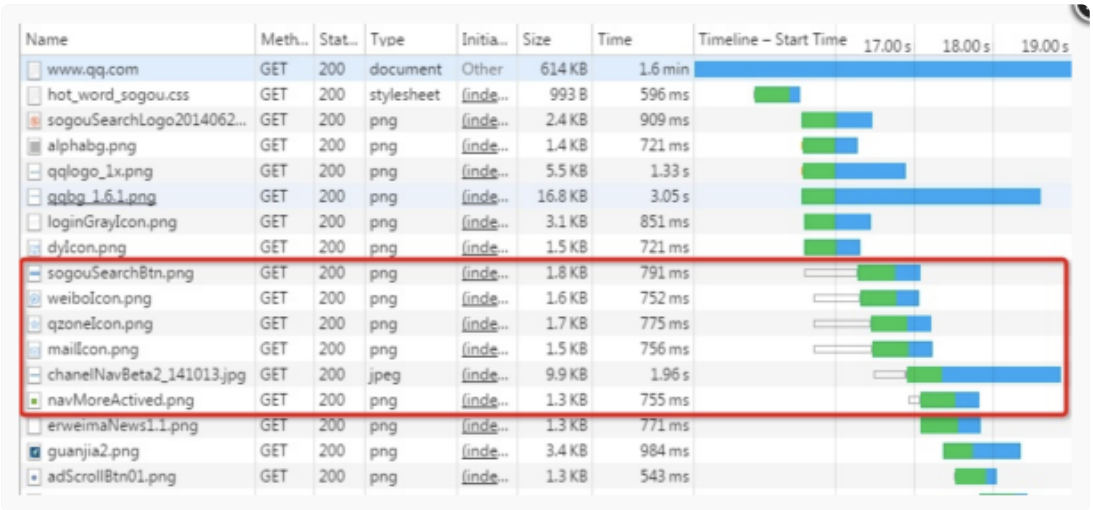


请求2，3，4，5不用等请求1的response返回之后才发出，而是几乎在同一时间把request发向了服务器。

下图可以看到pipeline机制对延迟的改变效果：



pipeline提高了性能，但head of line blocking并没有完全得到解决，server的response还是要求依次返回，遵循FIFO(first in first out)原则。也就是说如果请求1的response没有回来，2，3，4，5的response也不会被送回来。



如上图所示，红色圈出来的请求就因域名链接数已超过限制，而被挂起等待了一段时间。

协议开销大

HTTP1.x 在使用时，header 里携带的内容过大，在一定程度上增加了传输的成本，并且每次请求 header 基本不怎么变化，尤其在移动端增加用户流量。

安全因素

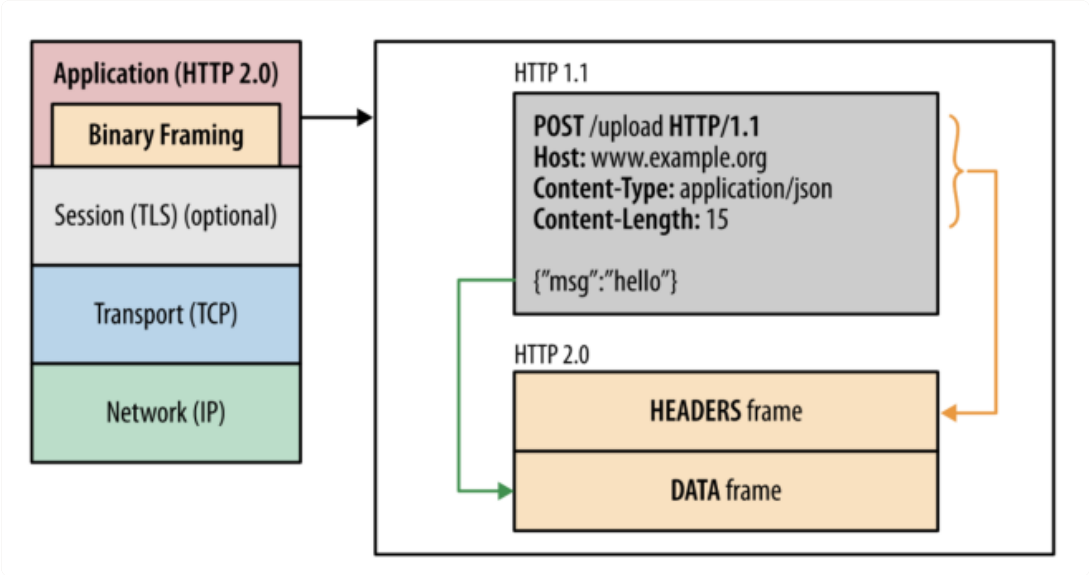
HTTP1.x 在传输数据时，所有传输的内容都是明文，客户端和服务器端都无法验证对方的身份，这在一定程度上无法保证数据的安全性

HTTP/2 新特性

HTTP/2 <https://hpbn.co/http2/> 是由google的SPDY协议衍生而来的。

1. 二进制传输

HTTP/2 采用二进制格式传输数据，而非 HTTP/1.x 的文本格式。
每个数据流都以消息的形式发送，而消息又由一个或多个帧组成。多个帧之间可以乱序发送，根据帧首部的流标识可以重新组装。
HTTP/2 仍是对之前 HTTP 标准的扩展，而非替代。 HTTP 的应用语义不变，提供的功能不变，HTTP 方法、状态代码、URI 和标头字段等这些核心概念也不变。

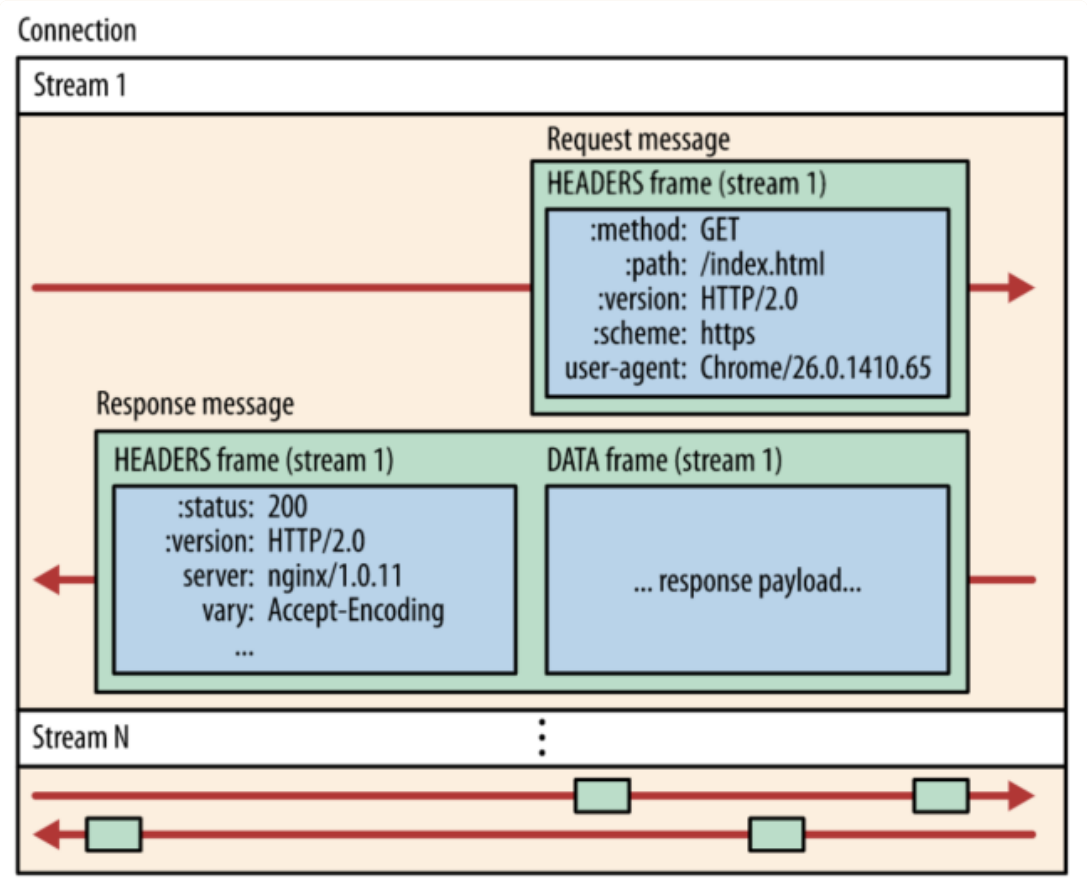


这里所谓的“层”，指的是位于套接字接口与应用可见的高级 HTTP API 之间一个经过优化的新编码机制：HTTP 的语义（包括各种动词、方法、标头）都不受影响，不同的是传输期间对它们的编码方式变了。 HTTP/1.x 协议以换行符作为纯文本的分隔符，而 HTTP/2 将所有传输的信息分割为更小的消息和帧，并采用二进制格式对它们编码。
新的二进制分帧机制改变了客户端与服务器之间交换数据的方式。 为了说明这个过程，我们需要了解 HTTP/2 的三个概念：

- **数据流 (stream)**：已建立的连接内的双向字节流，可以承载一条或多条消息。
- **消息 (message)**：与逻辑请求或响应消息对应的完整的一系列帧。
- **帧 (frame)**：HTTP/2 通信的最小单位，每个帧都包含帧头，至少也会标识出当前帧所属的数据流。

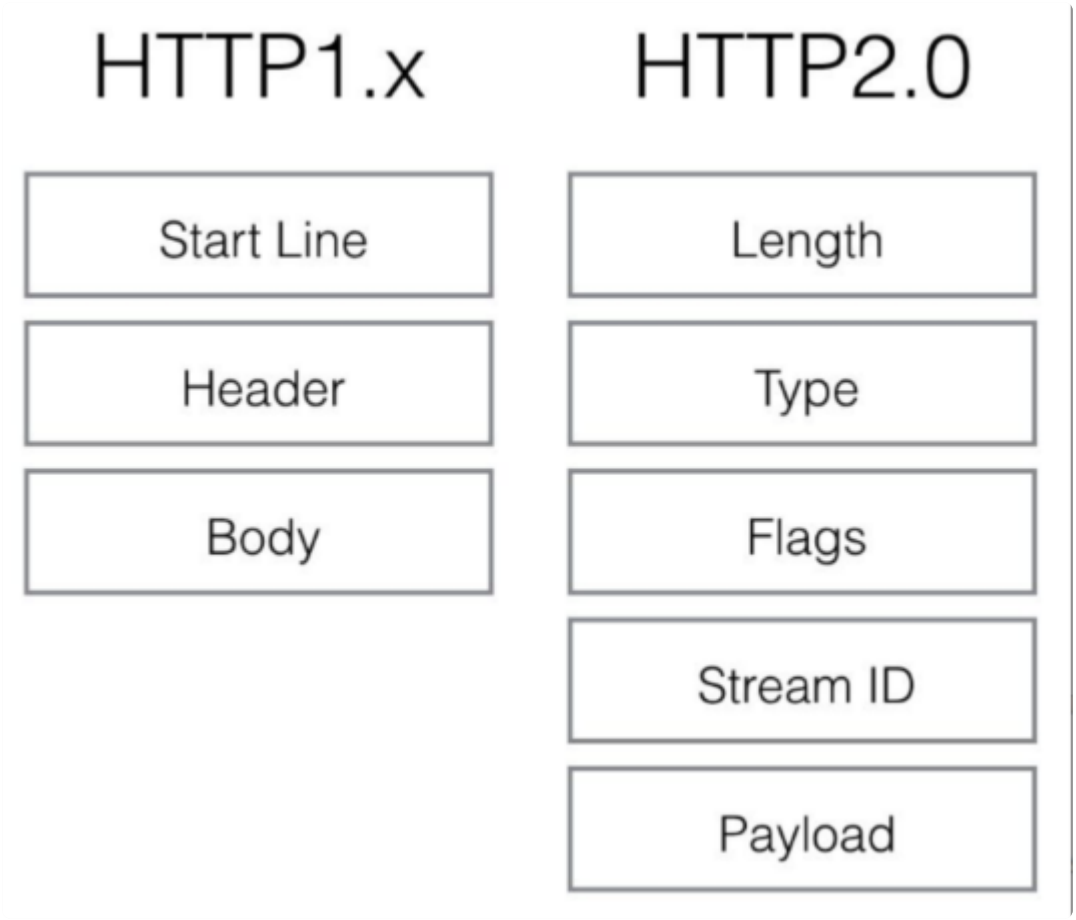
这些概念的关系总结如下：

- 所有通信都在一个 TCP 连接上完成，此连接可以承载任意数量的双向数据流。
- 每条stream都有一个唯一的标识符和可选的优先级信息，用于承载双向消息。
- 每个message都是一条逻辑 HTTP 消息（例如请求或响应），包含一个或多个帧。
- frame是最小的通信单位，承载着特定类型的数据，例如 HTTP 标头、消息负载等等。 来自不同数据流的帧可以交错发送，然后再根据每个帧头的数据流标识符重新组装。



简言之，HTTP/2 将 HTTP 协议通信分解为二进制编码帧的交换，这些帧对应着特定数据流中的消息。所有这些都在一个 TCP 连接内复用。

http2.0的frame格式与http1.x的消息对比：



- length定义了整个frame的开始到结束，
- type定义frame的类型（一共10种）；
- flags用bit位定义一些重要的参数；
- stream id用作流控制，一个request对应一个stream并分配一个id，这样一个连接上可以有多个stream，每个stream的frame可以随机的混杂在一起，接收方可以根据stream id将frame再归属到各自不同的request里面；
- payload就是request的正文了；

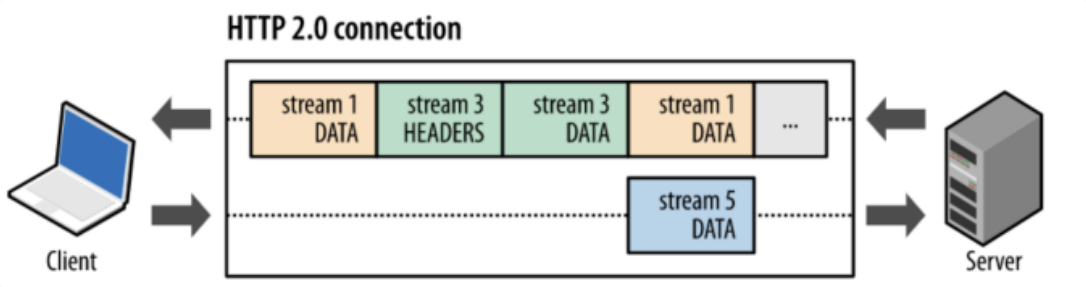
虽然看上去协议的格式和http1.x完全不同了，实际上http2.0并没有改变http1.x的语义，只是把原来http1.x的header和body部分用frame重新封装了一层而已。调试的时候浏览器甚至会把http2.0的frame自动还原成http1.x的格式。

对于http1.x来说，是通过设置tcp segment里的reset flag来通知对端关闭连接的。这种方式会直接断开连接，下次再发请求就必须重新建立连接。http2.0引入**RST_STREAM**类型的frame，可以在不断开连接的前提下取消某个request的stream，表现更好。

2. 多路复用

直白的说就是所有的请求都是通过一个 TCP 连接并发完成。HTTP/1.x 虽然通过 pipeline 也能并发请求，但是多个请求之间的响应会被阻塞的，所以 pipeline 至今也没有被普及应用，而 HTTP/2 做到了真正的并发请求。同时，流还支持优先级和流量控制。当流并发时，就会涉及到流的优先级和依赖。优先级高的流会被优先发送。

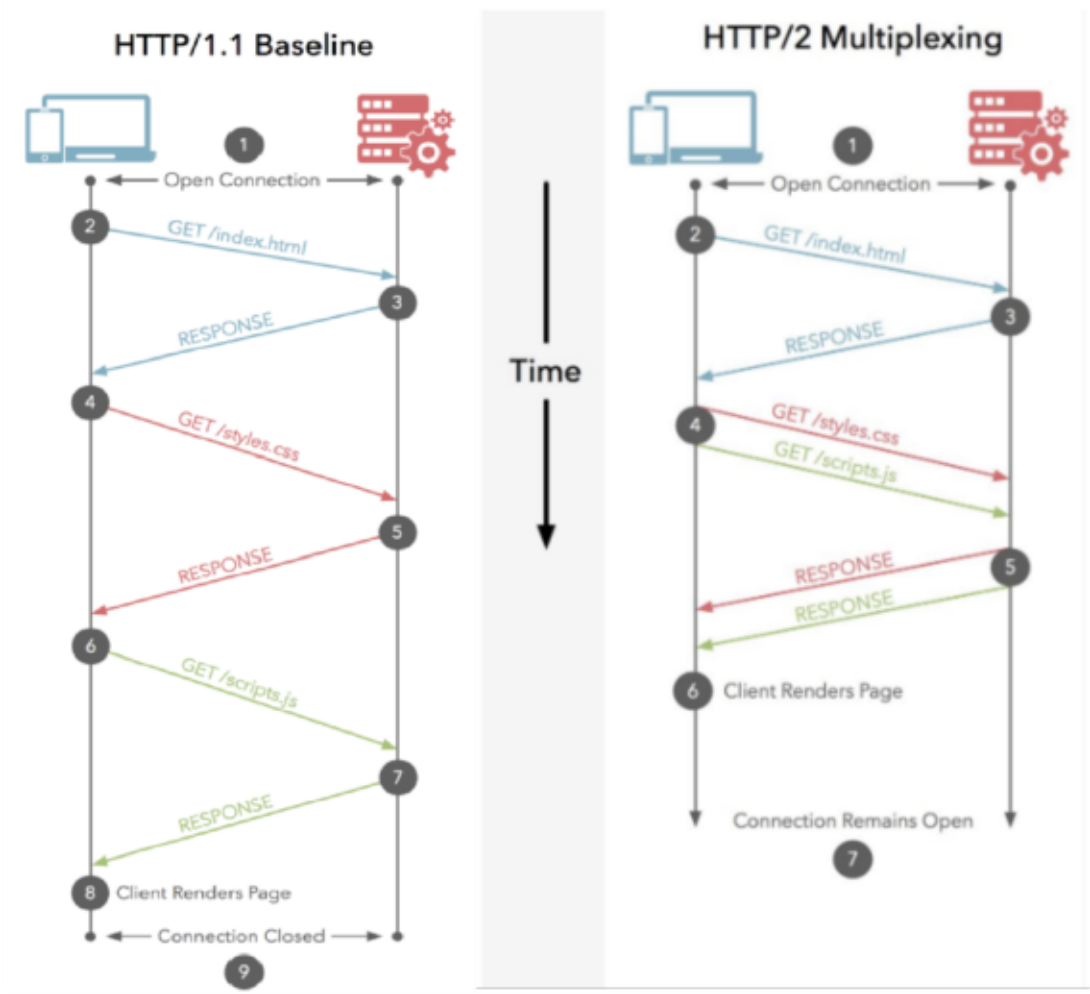
下图快照捕捉了同一个连接内并行的多个数据流。 客户端正在向服务器传输一个 `DATA` 帧（数据流 5），与此同时，服务器正向客户端交错发送数据流 1 和数据流 3 的一系列帧。因此，一个连接上同时有三个并行数据流。



将 HTTP 消息分解为独立的帧，交错发送，然后在另一端重新组装是 HTTP 2 最重要的一项增强：

- 并行交错地发送多个请求，请求之间互不影响。
- 并行交错地发送多个响应，响应之间互不干扰。
- 使用一个连接并行发送多个请求和响应。
- 不必再为绕过 HTTP/1.x 限制而做很多工作（例如级联文件、image sprites和域名分片）
- 消除不必要的延迟和提高现有网络容量的利用率，从而减少页面加载时间。

HTTP/2 中的新二进制分帧层解决了 HTTP/1.x 中存在的队首阻塞问题，也消除了并行处理和发送请求及响应时对多个连接的依赖。



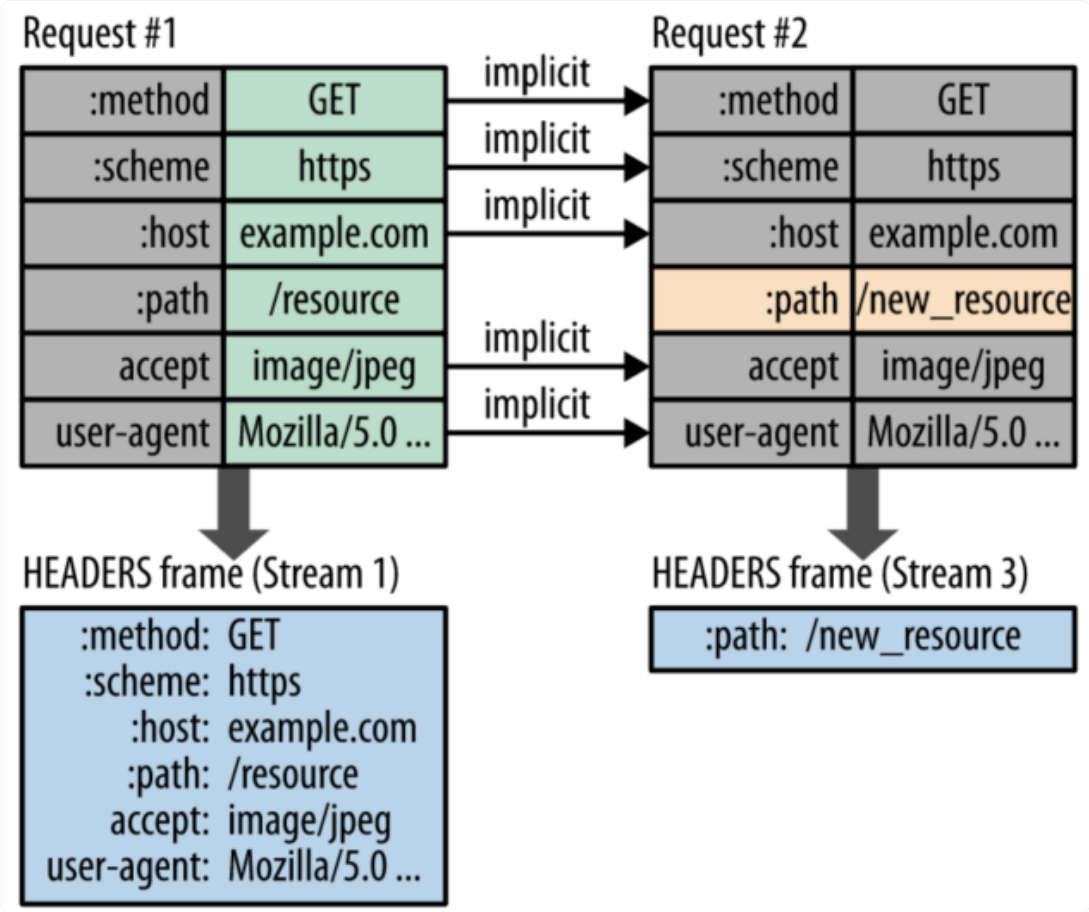
3. Header 压缩

在 HTTP/1 中，我们使用文本的形式传输 header，在 header 携带 cookie 的情况下，可能每次都需要重复传输几百到几千的字节。

为了减少这块的资源消耗并提升性能， HTTP/2 对这些首部采取了压缩策略：

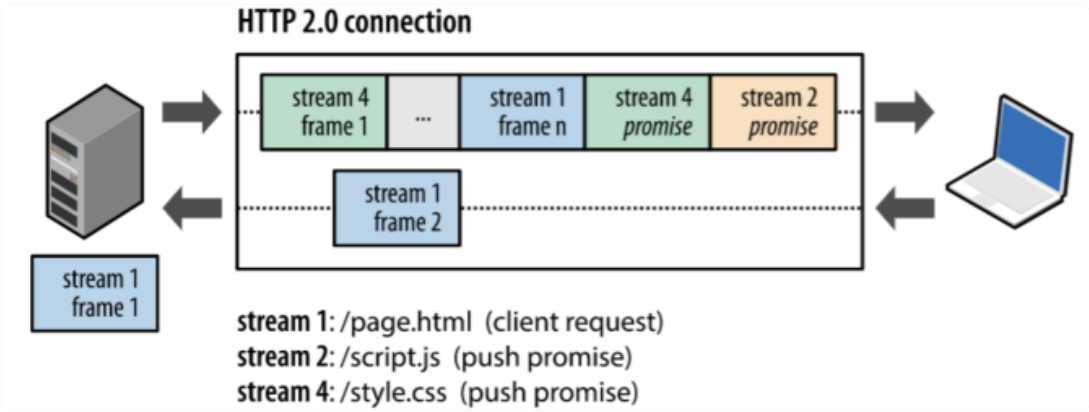
- HTTP/2 在客户端和服务器端使用“首部表”来跟踪和存储之前发送的键 - 值对，对于相同的数据，不再通过每次请求和响应发送；
- 首部表在 HTTP/2 的连接存续期内始终存在，由客户端和服务器共同渐进地更新;
- 每个新的首部键 - 值对要么被追加到当前表的末尾，要么替换表中之前的值

例如下图中的两个请求， 请求一发送了所有的头部字段，第二个请求则只需要发送差异数据，这样可以减少冗余数据，降低开销



4. Server Push

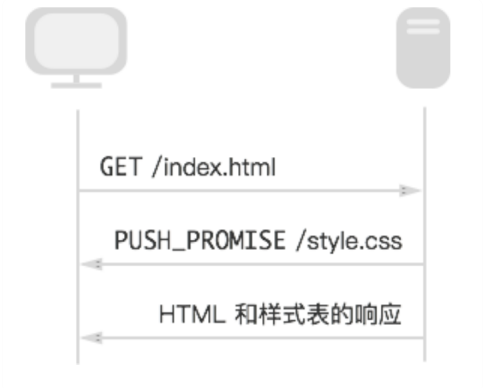
服务端能够更快的把资源推送给客户端。例如服务端可以主动把 JS 和 CSS 文件推送给客户端，而不需要客户端解析 HTML 再发送这些请求。当客户端需要的时候，它已经在客户端了。



HTTP/2 打破了严格的请求-响应语义，支持一对多和服务器发起的推送工作流。

所有服务器推送数据流都由 `PUSH_PROMISE` 帧发起，表明了服务器向客户端推送所述资源的意图，并且需要先于请求推送资源的响应数据传输。 这种传输顺序非常重要：客户端需要了解服务器打算推送哪些资源，以免为这些资源创建重复请求。 满足此要求的最简单策略是先于父响应（即， `DATA` 帧）发送所有 `PUSH_PROMISE` 帧，其中包含所承诺资源的 HTTP 标头。

假设服务端接收到客户端对 HTML 文件的请求，决定用 server push 推送一个css文件。那么，服务端会构造一个请求，包括请求方法和请求头，填充到一个 PUSH_PROMISE 帧里发送给客户端，来告知客户端它已经代劳发了这个请求。当客户端收到这个 PUSH_PROMISE 帧的时候，它就知道服务端将要推送一个CSS文件回来。如果此时客户端需要请求这个文件，即便服务端还没推完，它也不会往服务端发送对CSS文件的请求。在这个例子中，必须先发送 PUSH_PROMISE，再发送 HTML 的内容。这是因为 HTML 中存在对CSS的引用，一旦客户端发现了这个引用却还没收到 PUSH_PROMISE，它就会发起获取CSS文件请求。



在客户端接收到 `PUSH_PROMISE` 帧后，它可以根据自身情况选择拒绝数据流（通过 `RST_STREAM` 帧）。 （例如，如果资源已经位于缓存中，便可能会发生这种情况。） 这是一个相对于 HTTP/1.x 的重要提升。 相比之下，使用资源内联（一种受欢迎的 HTTP/1.x“优化”）等同于“强制推送”：客户端无法选择拒绝、取消或单独处理内联的资源。

使用 HTTP/2，客户端仍然完全掌控服务器推送的使用方式。 客户端可以限制并行推送的数据流数量；调整初始的流控制窗口以控制在数据流首次打开时推送的数据量；或完全停用服务器推送。 这些优先级在 HTTP/2 连接开始时通过 `SETTINGS` 帧传输，可能随时更新。

推送的每个资源都是一个数据流，与内嵌资源不同，客户端可以对推送的资源逐一复用、设定优先级和处理。 浏览器强制执行的唯一安全限制是，推送的资源必须符合原点相同这一政策：服务器对所提供内容必须具有权威性。

六、4种gRPC请求的和响应类型

当数据量大或者需要不断传输数据时候，我们应该使用流式RPC，它允许我们边处理边传输数据

gRPC主要有4种请求和响应模式，分别是`简单模式(Simple RPC)`、`服务端流式（Server-side streaming RPC）`、`客户端流式（Client-side streaming RPC）`、和`双向流式（Bidirectional streaming RPC）`。

- `简单模式(Simple RPC)`：客户端发起请求并等待服务端响应。
- `服务端流式（Server-side streaming RPC）`：客户端发送请求到服务器，拿到一个流去读取返回的消息序列。 客户端读取返回的流，直到里面没有任何消息。
- `客户端流式（Client-side streaming RPC）`：与服务端数据流模式相反，这次是客户端源源不断的向服务端发送数据流，而在发送结束后，由服务端返回一个响应。
- `双向流式（Bidirectional streaming RPC）`：双方使用读写流去发送一个消息序列，两个流独立操作，双方可以同时发送和同时接收。

以服务端流为例：
客户端发送请求到服务器，拿到一个流去读取返回的消息序列。 客户端读取返回的流，直到里面没有任何消息。
新建.proto文件

Protobuf 复制代码

```protobuf
syntax = "proto3";

option java_multiple_files = true;
// 生成java代码的package
option java_package = "com.chenj.grpc.api";
// 类名
option java_outer_classname = "MallProto";

// gRPC服务，这是个在线商城的订单查询服务
service OrderQuery {
    // 服务端流式：订单列表接口，入参是买家信息，返回订单列表(用stre
    rpc ListOrders (Buyer) returns (stream Order) {}
}

// 买家ID
message Buyer {
    int32 buyerId = 1;
}

// 返回结果的数据结构
message Order {
    // 订单ID
    int32 orderId = 1;
    // 商品ID
    int32 productId = 2;
    // 交易时间
    int64 orderTime = 3;
    // 买家备注
    string buyerRemark = 4;
}
```

根据.proto文件，生成Java代码

编写接口实现类

Java | 复制代码

```java
package com.chenj;

import com.chenj.grpc.api.Buyer;
import com.chenj.grpc.api.Order;
import com.chenj.grpc.api.OrderQueryGrpc;
import io.grpc.stub.StreamObserver;

import java.util.ArrayList;
import java.util.List;

public class OrderQueryServiceImpl extends OrderQueryGrpc.Order
    /**
     * mock一批数据
     * @return
     */
    private static List<Order> mockOrders(){
        List<Order> list = new ArrayList<>();
        Order.Builder builder = Order.newBuilder();

        for (int i = 0; i < 10; i++) {
            list.add(builder
                    .setOrderId(i)
                    .setProductId(1000+i)
                    .setOrderTime(System.currentTimeMillis()/10
                    .setBuyerRemark(("remark-" + i))
                    .build());
        }

        return list;
    }

    @Override
    public void listOrders(Buyer request, StreamObserver<Order>
        // 持续输出到client
        for (Order order : mockOrders()) {
            responseObserver.onNext(order);
```

responseObserver.onNext方法被多次调用，用以向客户端持续输出数据，最后通过responseObserver.onCompleted结束输出

定义服务端

Java 复制代码

```java
package com.chenj;

import io.grpc.Server;
import io.grpc.ServerBuilder;

import java.io.IOException;

public class GRPCServer {
    private static final int port = 9999;

    public static void main(String[] args) throws IOException,
        //设置service端口
        ////创建server对象,监听端口,注册服务并启动
        Server server = ServerBuilder.forPort(port)//监听50051端
                .addService(new OrderQueryServiceImpl())//注册服
                .build()//创建Server对象
                .start();//启动
        System.out.println(String.format("GRpc服务端启动成功, 端

        //我们将调用 awaittermination() 以保持服务器在后台保持运行
        server.awaitTermination();


    }
}
```

定义客户端

Java | 复制代码

```java
package com.chenj;

import com.chenj.grpc.api.*;
import io.grpc.ManagedChannel;
import io.grpc.ManagedChannelBuilder;
import io.grpc.StatusRuntimeException;

import java.time.LocalDateTime;
import java.time.ZoneOffset;
import java.time.format.DateTimeFormatter;
import java.util.ArrayList;
import java.util.Iterator;

public class GRPCClient {
    private static final String host = "localhost";
    private static final int serverPort = 9999;
    public static void main(String[] args) {
        //1,拿到一个通信channel
        ManagedChannel channel = ManagedChannelBuilder.forAdd
                usePlaintext()////这里将使用纯文本，无需任何加密
                .build();

        //

        try {

            //2.拿到stub对象
            OrderQueryGrpc.OrderQueryBlockingStub orderQueryB
            // 我们可以使用newBuilder 来设置 RPCDateRequest 对
            // gRPC的请求参数
            Buyer buyer = Buyer.newBuilder().setBuyerId(101).

            // 通过stub发起远程gRPC请求

            // gRPC的响应
            Iterator<Order> orderIterator;
            orderIterator = orderQueryBlockingStub.listOrders

            DateTimeFormatter dtf = DateTimeFormatter.ofPatte
            while (orderIterator.hasNext()) {
                Order order = orderIterator.next();
                System.out.println("订单ID: "+order.getOrderI
                        // 使用DateTimeFormatter将时间戳转为字
                        dtf.format(LocalDateTime.ofEpochSecon

```

```
46                 }
47             } finally {
48                 // 5.关闭channel, 释放资源.
49                 channel.shutdown();
50             }
51
52         }
53     }
```

对于服务端流类型的接口，客户端这边通过stub调用会得到Iterator类型的返回值，接下来要做的就是遍历Iterator

七、springcloud、grpc、dubbo 区别

gRPC由谷歌开发的一个高性能开源RPC框架，基于HTTp/2协议标准开发。利用ProtoBuf作为序列化工具和接口定义语言。

Dubbo 是一个分布式服务框架，致力于提供高性能和透明化的 **RPC 远程服务调用方案**，以及 **SOA 服务治理方案**。简单的说，Dubbo 就是个服务框架，说白了就是个**远程服务调用的分布式框架。**
Dubbo 未来的定位并不是要成为一个微服务的全面解决方案，而是专注在 RPC 领域，成为微服务生态体系中的一个重要组件

Spring Cloud 基于 Spring Boot，为微服务体系开发中的架构问题，提供了**一整套的解决方案**——服务注册与发现，服务消费，服务保护与熔断，网关，分布式调用追踪，分布式配置管理等。
SpringCloud 中，服务间的调用是通过 http 通信的，其实就相当于在调用 RESTFul 接口。

对比：

1、协议：服务间通信协议不同，Dubbo是基于TCP协议的rpc，spring cloud基于http协议。

2、RPC避免了上面提到的原生RPC带来的问题。而且REST相比RPC更为灵活，SpringCloud不存在代码级别的强依赖
3、效率：由于协议的不同，调用的效率相比之下Dubbo比SpringCLoud效率高。
4、技术开放性：SpringCLoud基于http协议，由于http的协议更方便于多语言情况下的整合，提供服务方可以用任意语言开发服务。
5、spring cloud是微服务生态，包括完整的微服务相关的组件工具集，而RPC是远程调用的技术，仅仅是微服务的一部分，而dubbo框架正是RPC的实现框架。
6、Spring Cloud还提供了包括Netflix Eureka、hystrix、feign、Spring Boot Admin 、Sleuth、config、stream、security、sleuth等分布式服务解决方案，而Dubbo为了拥抱融入Spring Cloud生态，Dubbo也在积极规划和演进适配SpringCloud生态的新版本。

八、使用spring boot集成grpc

核心技术
为了用java发布gRPC服务，我使用的是开源库net.devh:grpc-server-spring-boot-starter
在调用其他gRPC服务时用的是net.devh:grpc-client-spring-boot-starter
感谢该开源库的作者Michael大神，您的智慧的简化了java程序员的gRPC开发工作，项目地址：https://github.com/yidongnan/grpc-spring-boot-starter <https://github.com/yidongnan/grpc-spring-boot-starter>
特性：

- 在 spring boot 应用中，通过 `@GrpcService` 自动配置并运行一个嵌入式的 gRPC 服务。
- 使用 `@GrpcClient` 自动创建和管理您的 gRPC Channels 和 stubs
- 支持Spring Cloud <https://spring.io/projects/spring-cloud> (向 Consul <https://github.com/spring-cloud/spring-cloud-consul> 或 Eureka <https://github.com/spring-cloud/spring-cloud-netflix> 或 Nacos <https://github.com/spring-cloud-incubator/spring-cloud-alibaba> 注册服务并获取 gRPC 服务端信息)
- 支持Spring Sleuth <https://github.com/spring-cloud/spring-cloud-sleuth> 作为分布式链路跟踪解决方案(如果brave-instrument-grpc <https://mvnrepository.com/artifact/io.zipkin.brave/brave-instrumentation-grpc> 存在)
- 支持全局和自定义的 gRPC 服务端/客户端拦截器
- 支持 Spring-Security <https://github.com/spring-projects/spring-security>

- 支持metric (基于micrometer <https://micrometer.io/> /actuator <https://github.com/spring-projects/spring-boot/tree/master/spring-boot-project/spring-boot-actuator>)
- 也适用于 (non-shaded) grpc-netty

版本：

2.x.x.RELEASE 支持 Spring Boot 2.1.x/2.2.x 和 Spring Cloud Greenwich / Hoxton。

最新版本： `2.12.0.RELEASE`

(`2.4.0.RELEASE` 用于 Spring Boot 2.0.x & Spring Cloud Finchy).

1.x.x.RELEASE 支持 Spring Boot 1 & Spring Cloud Edgware, Dalston, Camden.

最新版本： `1.4.2.RELEASE`

注意: 该项目也可以在没有 Spring-Boot 的情况下使用，但是您需要手动配置一些 bean。

创建maven父工程spring-boot-grpc

创建springboot项目，勾选springboot-web即可

XML | 复制代码

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi=
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.
    <modelVersion>4.0.0</modelVersion>
    <packaging>pom</packaging>
    <modules>
        <module>spring-boot-grpc-lib</module>
        <module>local-server</module>
        <module>local-client</module>
    </modules>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.5.6</version>
        <relativePath/> <!-- lookup parent from repository --
    </parent>
    <groupId>com.chenj</groupId>
    <artifactId>spring-boot-grpc</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>spring-boot-grpc</name>
    <description>Demo project for Spring Boot</description>
    <properties>
        <java.version>1.8</java.version>
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>

    </dependencies>
<!--    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifac
            </plugin>
        </plugins>
```

```
46        </build>-->
47
48    </project>
```

创建模块spring-boot-grpc-lib

在父工程spring-boot-grpc下新建maven模块，名为spring-boot-grpc-lib

XML | 复制代码

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.
    <parent>
        <artifactId>spring-boot-grpc</artifactId>
        <groupId>com.chenj</groupId>
        <version>0.0.1-SNAPSHOT</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>

    <artifactId>spring-boot-grpc-lib</artifactId>
    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.so
        <maven.compiler.source>1.8</maven.compiler.source>
        <maven.compiler.target>1.8</maven.compiler.target>
    </properties>


    <dependencies>

        <dependency>
            <groupId>io.grpc</groupId>
            <artifactId>grpc-netty-shaded</artifactId>
            <version>1.37.0</version>
        </dependency>
        <dependency>
            <groupId>io.grpc</groupId>
            <artifactId>grpc-protobuf</artifactId>
            <version>1.37.0</version>
        </dependency>
        <dependency>
            <groupId>io.grpc</groupId>
            <artifactId>grpc-stub</artifactId>
            <version>1.37.0</version>
        </dependency>
        <dependency> <!-- necessary for Java 9+ -->
            <groupId>org.apache.tomcat</groupId>
            <artifactId>annotations-api</artifactId>
            <version>6.0.53</version>
            <scope>provided</scope>
        </dependency>
    </dependencies>


```

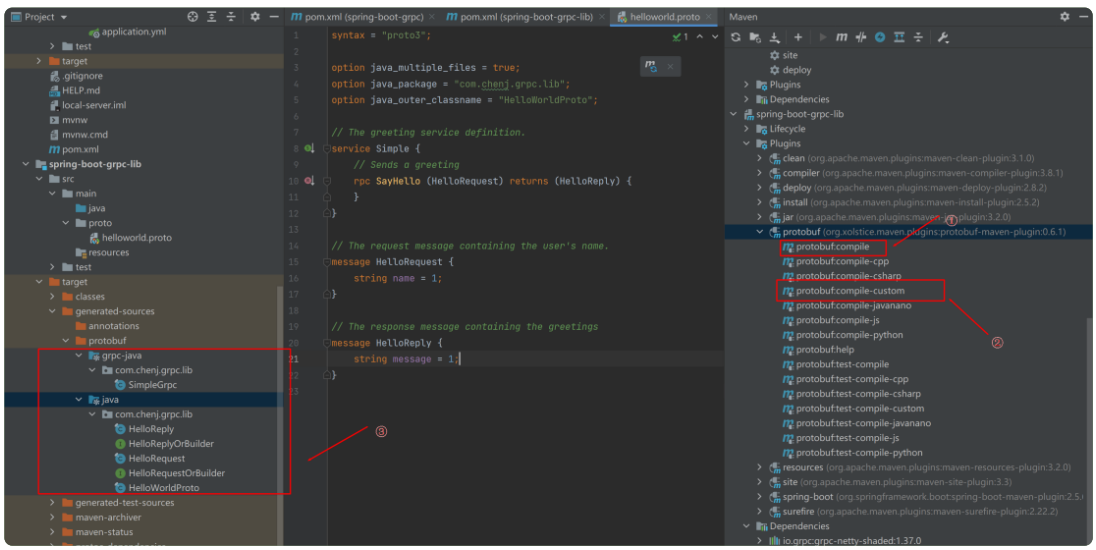
```
    <build>
        <extensions>
            <extension>
                <groupId>kr.motd.maven</groupId>
                <artifactId>os-maven-plugin</artifactId>
                <version>1.6.2</version>
            </extension>
        </extensions>
        <plugins>
            <plugin>
                <groupId>org.xolstice.maven.plugins</groupId>
                <artifactId>protobuf-maven-plugin</artifactId
                <version>0.6.1</version>
                <configuration>
                    <protocArtifact>com.google.protobuf:proto
                    <pluginId>grpc-java</pluginId>
                    <pluginArtifact>io.grpc:protoc-gen-grpc-j
                </configuration>
                <executions>
                    <execution>
                        <goals>
                            <goal>compile</goal>
                            <goal>compile-custom</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>


```

在spring-boot-grpc-lib模块的src/main/proto目录下新增名为helloworld.proto的文件，这里面定义了一个gRPC服务，里面含有一个接口，并且还有这个接口的入参和返回结果的定义：

Protobuf | 复制代码

```protobuf
syntax = "proto3";

option java_multiple_files = true;
option java_package = "com.chenj.grpc.lib";
option java_outer_classname = "HelloWorldProto";

// The greeting service definition.
service Simple {
    // Sends a greeting
    rpc SayHello (HelloRequest) returns (HelloReply) {
    }
}

// The request message containing the user's name.
message HelloRequest {
    string name = 1;
}

// The response message containing the greetings
message HelloReply {
    string message = 1;
}
```

proto文件已经做好，接下来要根据这个文件来生成Java代码



SimpleGrpc里面有抽象类SimpleImplBase，制作gRPC服务的时候需要继承该类，另外，如果您要远程调用gRPC的sayHello接口，就会用到SimpleGrpc类中的

SimpleStub类，其余的HelloReply、HelloRequest这些则是入参和返回的数据结构定义

创建模块local-server（gRPC服务端）

在父工程下面新建名为local-server的springboot模块，

gRPC 服务端使用一下命令添加 Maven 依赖项

XML 复制代码

```xml
<dependency>
  <groupId>net.devh</groupId>
  <artifactId>grpc-server-spring-boot-starter</artifactId>
  <version>2.12.0.RELEASE</version>
</dependency>
```

依赖上面的spring-boot-grpc-lib模块

XML 复制代码

```xml
<dependency>
      <groupId>com.chenj</groupId>
      <artifactId>spring-boot-grpc-lib</artifactId>
      <version>0.0.1-SNAPSHOT</version>
</dependency>
```

完整的pom.xml文件

XML | 复制代码

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi=
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <artifactId>spring-boot-grpc</artifactId>
    <groupId>com.chenj</groupId>
    <version>0.0.1-SNAPSHOT</version>
  </parent>
  <groupId>com.chenj</groupId>
  <artifactId>local-server</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>local-server</name>
  <description>Demo project for Spring Boot</description>
  <properties>
    <java.version>1.8</java.version>
  </properties>
  <dependencies>
    <dependency>
      <groupId>net.devh</groupId>
      <artifactId>grpc-server-spring-boot-starter</artifactId
      <version>2.12.0.RELEASE</version>
    </dependency>

    <dependency>
      <groupId>com.chenj</groupId>
      <artifactId>spring-boot-grpc-lib</artifactId>
      <version>0.0.1-SNAPSHOT</version>
    </dependency>


    <!--Lombok引入-->
    <dependency>
      <groupId>org.projectlombok</groupId>
      <artifactId>lombok</artifactId>
    </dependency>




  </dependencies>

  <!--<build>
    <plugins>
      <plugin>
```

```
46          <groupId>org.springframework.boot</groupId>
47          <artifactId>spring-boot-maven-plugin</artifactId>
48        </plugin>
49      </plugins>
50    </build>-->
51
52  </project>
```

springboot应用，配置文件内容如下：

YAML | 复制代码

```yaml

spring:
  application:
    name: spring-boot-grpc-server
# gRPC有关的配置，这里只需要配置服务端口号
grpc:
  server:
    port: 9898
server:
  port: 8080
```

新建拦截类LogGrpcInterceptor.java，每当gRPC请求到来后该类会先执行，这里是将方法名字在日志中打印出来，您可以对请求响应做更详细的处理：

Java 复制代码

```java
package com.chenj.springbootgrpcserver.interceptor;

import io.grpc.Metadata;
import io.grpc.ServerCall;
import io.grpc.ServerCallHandler;
import io.grpc.ServerInterceptor;
import lombok.extern.slf4j.Slf4j;


@Slf4j
public class LogGrpcInterceptor implements ServerInterceptor {
    @Override
    public <ReqT, RespT> ServerCall.Listener<ReqT> interceptCal

        log.info(serverCall.getMethodDescriptor().getFullMethod
        return serverCallHandler.startCall(serverCall, metadata
    }
}

```

为了让LogGrpcInterceptor可以在gRPC请求到来时被执行，需要做相应的配置，如下所示，在普通的bean的配置中添加注解即可：

Java 复制代码

```java
package com.chenj.springbootgrpcserver.config;

import com.chenj.springbootgrpcserver.interceptor.LogGrpcInterc
import io.grpc.ServerInterceptor;
import net.devh.boot.grpc.server.interceptor.GrpcGlobalServerIn
import org.springframework.context.annotation.Configuration;


@Configuration(proxyBeanMethods = false)
public class GlobalInterceptorConfiguration {
    @GrpcGlobalServerInterceptor
    ServerInterceptor logServerInterceptor() {
        return new LogGrpcInterceptor();
    }
}
```

应用启动类很简单：

Java 复制代码

```java
package com.chenj.springbootgrpcserver;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplica

@SpringBootApplication
public class SpringBootGrpcServerApplication {

    public static void main(String[] args) {
        SpringApplication.run(SpringBootGrpcServerApplication.c
    }

}
```

接下来是最重要的service类，gRPC服务在此处对外暴露出去，完整代码如下

Java 复制代码

```java
package com.chenj.springbootgrpcserver.service;


import com.chenj.grpc.lib.HelloReply;
import com.chenj.grpc.lib.HelloRequest;
import com.chenj.grpc.lib.SimpleGrpc;
import io.grpc.stub.StreamObserver;
import net.devh.boot.grpc.server.service.GrpcService;

import java.util.Date;


@GrpcService
public class GrpcServerService extends SimpleGrpc.SimpleImplBas

    @Override
    public void sayHello(HelloRequest request,
                         StreamObserver<HelloReply> responseObs
        HelloReply reply = HelloReply.newBuilder().setMessage("
        responseObserver.onNext(reply);
        responseObserver.onCompleted();
    }
}
```

上述GrpcServerService.java中有几处需要注意：
是使用@GrpcService注解，再继承SimpleImplBase，这样就可以借助grpc-server-spring-boot-starter库将sayHello暴露为gRPC服务；
SimpleImplBase是前文中根据proto自动生成的java代码，在spring-boot-grpc-lib模块中；
sayHello方法中处理完毕业务逻辑后，调用HelloReply.onNext方法填入返回内容；
调用HelloReply.onCompleted方法表示本次gRPC服务完成；
至此，gRPC服务端编码就完成了，咱们接着开始客户端开发

创建模块local-client（gRPC客户端）

在父工程grpc-turtorials下面新建名为local-client的模块

gRPC客户端使用一下命令添加 Maven 依赖项：

XML 复制代码

```xml
<dependency>
  <groupId>net.devh</groupId>
  <artifactId>grpc-client-spring-boot-starter</artifactId>
  <version>2.12.0.RELEASE</version>
</dependency>
```

依赖上面的spring-boot-grpc-lib模块

XML 复制代码

```xml
<dependency>
      <groupId>com.chenj</groupId>
      <artifactId>spring-boot-grpc-lib</artifactId>
      <version>0.0.1-SNAPSHOT</version>
</dependency>
```

完整的pom.xml文件

XML | 复制代码

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="h
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <artifactId>spring-boot-grpc</artifactId>
        <groupId>com.chenj</groupId>
        <version>0.0.1-SNAPSHOT</version>
    </parent>
    <groupId>com.chenj</groupId>
    <artifactId>local-client</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>local-client</name>
    <description>Demo project for Spring Boot</description>
    <properties>
        <java.version>1.8</java.version>
    </properties>
    <dependencies>
        <dependency>
            <groupId>net.devh</groupId>
            <artifactId>grpc-client-spring-boot-starter</artifa
            <version>2.12.0.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>com.chenj</groupId>
            <artifactId>spring-boot-grpc-lib</artifactId>
            <version>0.0.1-SNAPSHOT</version>
        </dependency>


    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactI
            </plugin>
        </plugins>
    </build>

</project>

```

应用配置文件src/main/resources/application.yml，注意address的值就是gRPC服务端的信息，我这里local-server和local-client在同一台电脑上运行，请您根据自己情况来设置：

YAML | 复制代码

```yaml
server:
  port: 8088
spring:
  application:
    name: local-client


grpc:
  client:
    # gRPC配置的名字，GrpcClient注解会用到
    local-grpc-server:
      # gRPC服务端地址
      address: 'static://127.0.0.1:9898'
      enableKeepAlive: true
      keepAliveWithoutCalls: true
      negotiationType: plaintext
```

新建拦截类LogGrpcInterceptor，与服务端的拦截类差不多，不过实现的接口不同：

Java 复制代码

```java
package com.chenj.springbootgrpcclient.interceptor;


import io.grpc.*;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class LogGrpcInterceptor implements ClientInterceptor {

    private static final Logger log = LoggerFactory.getLogger(L

    @Override
    public <ReqT, RespT> ClientCall<ReqT, RespT> interceptCall(

        log.info(method.getFullMethodName());
        return next.newCall(method, callOptions);
    }
}


```

为了让拦截类能够正常工作，即发起gRPC请求的时候被执行，需要新增一个配置类：

Java 复制代码

```java
package com.chenj.springbootgrpcclient.config;



import com.chenj.springbootgrpcclient.interceptor.LogGrpcInterc
import io.grpc.ClientInterceptor;
import net.devh.boot.grpc.client.interceptor.GrpcGlobalClientIn
import org.springframework.context.annotation.Configuration;
import org.springframework.core.Ordered;
import org.springframework.core.annotation.Order;

@Order(Ordered.LOWEST_PRECEDENCE)
@Configuration(proxyBeanMethods = false)
public class GlobalClientInterceptorConfiguration {

    @GrpcGlobalClientInterceptor
    ClientInterceptor logClientInterceptor() {
        return new LogGrpcInterceptor();
    }
}
```

启动类：

Java 复制代码

```java
package com.chenj.springbootgrpcclient;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplica

@SpringBootApplication
public class SpringBootGrpcClientApplication {

    public static void main(String[] args) {
        SpringApplication.run(SpringBootGrpcClientApplication.c
    }

}
```

接下来是最重要的服务类GrpcClientService，

Java 复制代码

```java
package com.chenj.springbootgrpcclient.service;

import com.chenj.grpc.lib.HelloReply;
import com.chenj.grpc.lib.HelloRequest;
import com.chenj.grpc.lib.SimpleGrpc;
import io.grpc.StatusRuntimeException;
import net.devh.boot.grpc.client.inject.GrpcClient;
import org.springframework.stereotype.Service;

@Service
public class GrpcClientService {

    @GrpcClient("local-grpc-server")
    private SimpleGrpc.SimpleBlockingStub simpleStub;

    public String sendMessage(final String name) {
        try {
            final HelloReply response = this.simpleStub.sayHell
            return response.getMessage();
        } catch (final StatusRuntimeException e) {
            return "FAILED with " + e.getStatus().getCode().nam
        }
    }
}


```

上述GrpcClientService类有几处要注意的地方：
用@Service将GrpcClientService注册为spring的普通bean实例；
用@GrpcClient修饰SimpleBlockingStub，这样就可以通过grpc-client-spring-boot-starter库发起gRPC调用，被调用的服务端信息来自名为local-grpc-server的配置；
SimpleBlockingStub来自前文中根据helloworld.proto生成的java代码；
SimpleBlockingStub.sayHello方法会远程调用local-server应用的gRPC服务；

为了验证gRPC服务调用能否成功，再新增个web接口，接口内部会调用GrpcClientService.sendMessage，这样咱们通过浏览器就能验证gRPC服务是否调用成功了：

Java 复制代码

```java
package com.chenj.springbootgrpcclient.controller;


import com.chenj.springbootgrpcclient.service.GrpcClientService
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class GrpcClientController {

    @Autowired
    private GrpcClientService grpcClientService;

    @RequestMapping("/")
    public String printMessage(@RequestParam(defaultValue = "wi
        return grpcClientService.sendMessage(name);
    }
}
```

九、gRPC在Spring Cloud中的使用

eureka应用开发

新建名为cloud-eureka的模块,作为微服务注册中心

XML | 复制代码

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi=
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <artifactId>spring-boot-grpc</artifactId>
        <groupId>com.chenj</groupId>
        <version>0.0.1-SNAPSHOT</version>
    </parent>
    <groupId>com.chenj</groupId>
    <artifactId>cloud-eureka</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>cloud-eureka</name>
    <description>Demo project for Spring Boot</description>
    <properties>
        <java.version>1.8</java.version>
        <spring-cloud.version>2020.0.4</spring-cloud.version>
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-netflix-eureka-s
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-actuator</artifac
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-devtools</artifactId>
            <scope>runtime</scope>
            <optional>true</optional>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>
```

```
46      <dependencyManagement>
47          <dependencies>
48              <dependency>
49                  <groupId>org.springframework.cloud</groupId>
50                  <artifactId>spring-cloud-dependencies</artifa
51                  <version>${spring-cloud.version}</version>
52                  <type>pom</type>
53                  <scope>import</scope>
54              </dependency>
55          </dependencies>
56      </dependencyManagement>
57
58      <build>
59          <plugins>
60              <plugin>
61                  <groupId>org.springframework.boot</groupId>
62                  <artifactId>spring-boot-maven-plugin</artifac
63              </plugin>
64          </plugins>
65      </build>
66
67  </project>
```

配置文件application.yml，设置自己的web端口号和应用名

YAML 复制代码

```yaml
server:
  port: 8085

spring:
  application:
    name: cloud-eureka

eureka:
  instance:
    hostname: localhost
    prefer-ip-address: true
# info 就是我们自己配置在配置文件中以 info 开头的配置信息
    status-page-url-path: /actuator/info
# health 主要用来检查应用的运行状态，这是我们使用最高频的一个监控点。
    health-check-url-path: /actuator/health
    lease-expiration-duration-in-seconds: 30
    lease-renewal-interval-in-seconds: 30
  client:
    registerWithEureka: false
    fetchRegistry: false
    serviceUrl:
      defaultZone: http://localhost:8085/eureka/
  server:
    enable-self-preservation: false
#启用接口关闭 Spring Boot
endpoints:
  shutdown:
    enabled: true

```

启动类

Java 复制代码

```java
package com.chenj.cloudeureka;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplica
import org.springframework.cloud.netflix.eureka.server.EnableEu

@EnableEurekaServer
@SpringBootApplication
public class CloudEurekaApplication {

    public static void main(String[] args) {
        SpringApplication.run(CloudEurekaApplication.class, arg
    }

}
```

以上就是一个简单通用的eureka服务了；

gRPC服务端开发

新建名为cloud-server-side的模块

XML | 复制代码

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi=
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.5.6</version>
        <relativePath/> <!-- lookup parent from repository --
    </parent>
    <groupId>com.chenj</groupId>
    <artifactId>cloud-server-side</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>cloud-server-side</name>
    <description>Demo project for Spring Boot</description>
    <properties>
        <java.version>1.8</java.version>
        <spring-cloud.version>2020.0.4</spring-cloud.version>
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-actuator</artifac
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-netflix-eureka-c
        </dependency>

        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <optional>true</optional>
        </dependency>

        <dependency>
            <groupId>net.devh</groupId>
            <artifactId>grpc-server-spring-boot-starter</arti
            <version>2.12.0.RELEASE</version>
        </dependency>

```

```
        <dependency>
            <groupId>com.chenj</groupId>
            <artifactId>spring-boot-grpc-lib</artifactId>
            <version>0.0.1-SNAPSHOT</version>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>
    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-dependencies</artifa
                <version>${spring-cloud.version}</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifac
                <configuration>
                    <excludes>
                        <exclude>
                            <groupId>org.projectlombok</group
                            <artifactId>lombok</artifactId>
                        </exclude>
                    </excludes>
                </configuration>
            </plugin>
        </plugins>
    </build>

</project>
```

配置文件application.yml，设置自己的应用名，另外值得注意的是server.port和grpc.server.port这两个配置的值都是0，这样两个端口就会被自动分配未被占用的

值:

YAML 复制代码

```yaml
spring:
  application:
    name: cloud-server-side

server:
  port: 0
grpc:
  server:
    port: 0
eureka:
  instance:
    prefer-ip-address: true
    instanceId: ${spring.application.name}:${vcap.application.i
  client:
    register-with-eureka: true
    fetch-registry: true
    service-url:
      defaultZone: http://localhost:8085/eureka/

```

启动类

Java 复制代码

```java
package com.chenj.cloudserverside;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplica
import org.springframework.cloud.client.discovery.EnableDiscove
import org.springframework.cloud.netflix.eureka.EnableEurekaCli

@EnableEurekaClient
@EnableDiscoveryClient
@SpringBootApplication
public class CloudServerSideApplication {

    public static void main(String[] args) {
        SpringApplication.run(CloudServerSideApplication.class,
    }

}
```

提供gRPC服务的类GrpcServerService，和local-server模块中的一样：

Java | 复制代码

```java
package com.chenj.cloudserverside;


import com.chenj.grpc.lib.HelloReply;
import com.chenj.grpc.lib.HelloRequest;
import com.chenj.grpc.lib.SimpleGrpc;
import io.grpc.stub.StreamObserver;
import net.devh.boot.grpc.server.service.GrpcService;
import java.util.Date;

@GrpcService
public class GrpcServerService extends SimpleGrpc.SimpleImplBas

    @Override
    public void sayHello(HelloRequest request,
                         StreamObserver<HelloReply> responseObs
        HelloReply reply = HelloReply.newBuilder().setMessage("
        responseObserver.onNext(reply);
        responseObserver.onCompleted();
    }
}


```

以上就是服务端代码了，可见除了将gRPC端口设置为0，以及常规使用eureka的配置，其他部分和local-server模块是一样的；

gRPC客户端开发

新建名为cloud-client-side的模块

XML | 复制代码

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi=
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.5.6</version>
        <relativePath/> <!-- lookup parent from repository --
    </parent>
    <groupId>com.chenj</groupId>
    <artifactId>cloud-client-side</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>cloud-client-side</name>
    <description>Demo project for Spring Boot</description>
    <properties>
        <java.version>1.8</java.version>
        <spring-cloud.version>2020.0.4</spring-cloud.version>
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-actuator</artifac
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-netflix-eureka-c
        </dependency>

        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <optional>true</optional>
        </dependency>

        <dependency>
            <groupId>net.devh</groupId>
            <artifactId>grpc-client-spring-boot-starter</arti
            <version>2.12.0.RELEASE</version>
        </dependency>
        <dependency>
```

```
            <groupId>com.chenj</groupId>
            <artifactId>spring-boot-grpc-lib</artifactId>
            <version>0.0.1-SNAPSHOT</version>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>
    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-dependencies</artifa
                <version>${spring-cloud.version}</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifac
                <configuration>
                    <excludes>
                        <exclude>
                            <groupId>org.projectlombok</group
                            <artifactId>lombok</artifactId>
                        </exclude>
                    </excludes>
                </configuration>
            </plugin>
        </plugins>
    </build>

</project>
```

配置文件application.yml，设置自己的web端口号，另外值得注意的是gRPC配置项cloud-server-side的名字要等于gRPC服务端在eureka注册的名字，并且不需要address配置项：

YAML 复制代码

```yaml
server:
  port: 8086
spring:
  application:
    name: cloud-client-side
eureka:
  instance:
    prefer-ip-address: true
    status-page-url-path: /actuator/info
    health-check-url-path: /actuator/health
    instanceId: ${spring.application.name}:${vcap.application.i
  client:
    register-with-eureka: true
    fetch-registry: true
    service-url:
      defaultZone: http://localhost:8085/eureka/
grpc:
  client:
    # gRPC配置的名字，GrpcClient注解会用到
    cloud-server-side:
      enableKeepAlive: true
      keepAliveWithoutCalls: true
      negotiationType: plaintext

```

启动类

Java 复制代码

```java
package com.chenj.cloudclientside;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplica
import org.springframework.cloud.client.discovery.EnableDiscove
import org.springframework.cloud.netflix.eureka.EnableEurekaCli

@EnableEurekaClient
@EnableDiscoveryClient
@SpringBootApplication
public class CloudClientSideApplication {

    public static void main(String[] args) {
        SpringApplication.run(CloudClientSideApplication.class,
    }

}
```

封装gRPC调用的服务类GrpcServerService，和local-server模块中的一样，GrpcClient注解对应配置中的gRPC配置项：

Java 复制代码

```java
package com.chenj.cloudclientside;




import com.chenj.grpc.lib.HelloReply;
import com.chenj.grpc.lib.HelloRequest;
import com.chenj.grpc.lib.SimpleGrpc;
import io.grpc.StatusRuntimeException;
import net.devh.boot.grpc.client.inject.GrpcClient;
import org.springframework.stereotype.Service;

@Service
public class GrpcClientService {

    @GrpcClient("cloud-server-side")
    private SimpleGrpc.SimpleBlockingStub simpleStub;

    public String sendMessage(final String name) {
        try {
            final HelloReply response = this.simpleStub.sayHell
            return response.getMessage();
        } catch (final StatusRuntimeException e) {
            return "FAILED with " + e.getStatus().getCode().nam
        }
    }
}


```

再做一个web接口类，这样我们就能通过web调用验证gRPC服务了：

Java 复制代码

```java
package com.chenj.cloudclientside;


import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class GrpcClientController {

    @Autowired
    private GrpcClientService grpcClientService;

    @RequestMapping("/")
    public String printMessage(@RequestParam(defaultValue = "wi
        return grpcClientService.sendMessage(name);
    }
}
```

客户端开发完毕，接下来可以验证了；

1045255dc3a4.png&title=%E9%9B%B6%E5%9F%BA%E7%A1%80%E5%85%A5%