Dubbo 在跨语言和协议穿透性方向的探索：支持 HTTP/2 gRPC

原创 刘军 阿里巴巴中间件 2019-11-21

本文整理自刘军在 Dubbo meetup 成都站分享的《Dubbo 在多语言和协议穿透性方向上的探索》。

本文总体上可分为基础产品简介、Dubbo 对 gRPC (HTTP/2) 和 Protobuf 的支持及示例演示三部分，在简介部分介绍了 Dubbo、HTTP/2、gRPC、Protobuf 的基本概念和特点；第二部分介绍了 Dubbo 为何要支持 gRPC (HTTP/2) 和 Protobuf，以及这种支持为 gRPC 和 Dubbo 开发带来的好处与不同；第三部分通过两个实例分别演示了 Dubbo gRPC 和 Dubbo Protobuf 的使用方式。

基本介绍

Dubbo 协议

从协议层面展开，以下是当前 2.7 版本支持的 Dubbo 协议：

Dubbo Protocol

Offsets Octet		0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Magic High								Magic Low								Req/Res	2 Way	Event	Serialization ID					Status							
4	32	RPC Request ID																															
8	64																																
12	96	Data Length																															
16 ...	128 ...	Variable length part, in turn, is: dubbo version, service name, service version, method name, parameter types, arguments, attachments																															

众所周知，Dubbo 协议是直接定义在 TCP 传输层协议之上，由于 TCP 高可靠全双工的特点，为 Dubbo 协议的定义提供了最大的灵活性，但同时也正是因为这样的灵活性，RPC 协议普遍都是定制化的私有协议，Dubbo 同样也面临这个问题。在这里我们着重讲一下 Dubbo 在协议通

用性方面值得改进的地方，关于协议详细解析请参见官网博客。

- Dubbo 协议体 Body 中有一个可扩展的 attachments 部分，这给 RPC 方法之外额外传递附加属性提供了可能，是一个很好的设计。但是类似的 Header 部分，却缺少类似的可扩展 attachments，这点可参考 HTTP 定义的 Ascii Header 设计，将 Body Attachments 和 Header Attachments 做职责划分。
- Body 协议体中的一些 RPC 请求定位符如 Service Name、Method Name、Version 等，可以提到 Header 中，和具体的序列化协议解耦，以更好的被网络基础设施识别或用于流量管控。
- 扩展性不够好，欠缺协议升级方面的设计，如 Header 头中没有预留的状态标识位，或者像 HTTP 有专为协议升级或协商设计的特殊 packet。
- 在 Java 版本的代码实现上，不够精简和通用。如在链路传输中，存在一些语言绑定的内容；消息体中存在冗余内容，如 Service Name 在 Body 和 Attachments 中都存在。

HTTP/1

相比于直接构建与 TPC 传输层的私有 RPC 协议，构建于 HTTP 之上的远程调用解决方案会有更好的通用性，如WebServices 或 REST 架构，使用 HTTP + JSON 可以说是一个事实标准的解决方案。

之所有选择构建在 HTTP 之上，我认为有两个最大的优势：

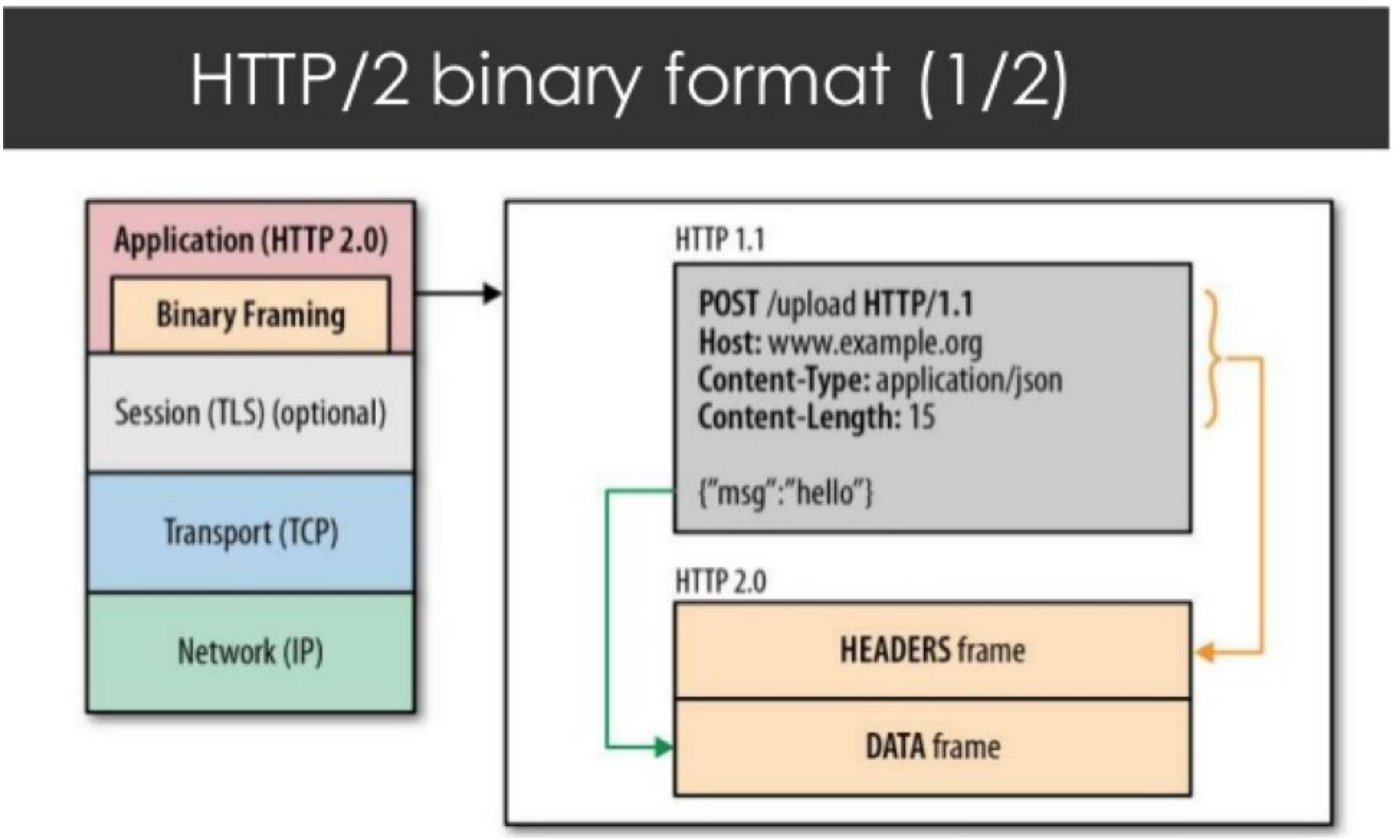
1. HTTP 的语义和可扩展性能很好的满足 RPC 调用需求。
2. 通用性，HTTP 协议几乎被网络上的所有设备所支持，具有很好的协议穿透性。

具体来说，HTTP/1 的优势和限制是：

- 典型的 Request – Response 模型，一个链路上一次只能有一个等待的 Request 请求
- HTTP/1 支持 Keep-Alive 链接，避免了链接重复创建开销
- Human Readable Headers，使用更通用、更易于人类阅读的头部传输格式
- 无直接 Server Push 支持，需要使用 Polling Long-Polling 等变通模式

HTTP/2

HTTP/2 保留了 HTTP/1 的所有语义，在保持兼容的同时，在通信模型和传输效率上做了很大的改进。



- 支持单条链路上的 Multiplexing，相比于 Request - Response 独占链路，基于 Frame 实现更高效利用链路
- Request - Stream 语义，原生支持 Server Push 和 Stream 数据传输
- Flow Control，单条 Stream 粒度的和整个链路粒度的流量控制
- 头部压缩 HPACK
- Binary Frame
- 原生 TLS 支持

gRPC

上面提到了在 HTTP 及 TCP 协议之上构建 RPC 协议各自的优缺点，相比于 Dubbo 构建于 TPC 传输层之上，Google 选择将 gRPC 直接定义在 HTTP/2 协议之上，关于 gRPC 的 基本介绍和

设计愿景 请参考以上两篇文章，我这里仅摘取设计愿景中几个能反映 gRPC 设计目的特性来做简单说明。

gRPC 的基本介绍：
https://platformlab.stanford.edu/Seminar Talks/gRPC.pdf
设计愿景：
https://grpc.io/blog/principles/?spm=ata.13261165.0.0.2be55017XbUhs8

- Coverage & Simplicity，协议设计和框架实现要足够通用和简单，能运行在任何设备之上，甚至一些资源首先的如 IoT、Mobile 等设备。
- Interoperability & Reach，要构建在更通用的协议之上，协议本身要能被网络上几乎所有的基础设施所支持。
- General Purpose & Performant，要在场景和性能间做好平衡，首先协议本身要是适用于各种场景的，同时也要尽量有高的性能。
- Payload Agnostic，协议上传输的负载要保持语言和平台中立。
- Streaming，要支持 Request - Response、Request - Stream、Bi-Steam 等通信模型。
- Flow Control，协议自身具备流量感知和限制的能力。
- Metadata Exchange，在 RPC 服务定义之外，提供额外附加数据传输的能力。

总的来说，在这样的设计理念指导下，gRPC 最终被设计为一个跨语言、跨平台的、通用的、高性能的、基于 HTTP/2 的 RPC 协议和框架。

Protobuf

Protocol buffers (Protobuf) 是 Google 推出的一个跨平台、语言中立的结构化数据描述和序列化的产品，它定义了一套结构化数据定义的协议，同时也提供了相应的 Compiler 工具，用来将语言中立的描述转化为相应语言的具体描述。

Protocol buffers (Protobuf) 详情参考：
https://developers.google.com/protocol-buffers/docs/overview

Compiler 详情参考：

https://github.com/protocolbuffers/protobuf/releases/tag/v3.10.0

它的一些特性包括：

- 跨语言 跨平台，语言中立的数据描述格式，默认提供了生成多种语言的 Compiler 工具。
- 安全性，由于反序列化的范围和输出内容格式都是 Compiler 在编译时预生成的，因此绕过了类似 Java Deserialization Vulnarability 的问题。
- 二进制 高性能
- 强类型
- 字段变更向后兼容

```
message Person {
      required string name = 1;
      required int32 id = 2;
      optional string email = 3;

      enum PhoneType {
        MOBILE = 0;
        HOME = 1;
        WORK = 2;
      }

      message PhoneNumber {
        required string number = 1;
        optional PhoneType type = 2 [default = HOME];
      }

      repeated PhoneNumber phone = 4;
    }
```

除了结构化数据描述之外，Protobuf 还支持定义 RPC 服务，它允许我们定义一个 .proto 的服务描述文件，进而利用 Protobuf Compiler 工具生成特定语言和 RPC 框架的接口和 stub。后续将要具体讲到的 gRPC + Protobuf、Dubbo-gRPC + Protobuf 以及 Dubbo + Protobuf 都是通过定制 Compiler 类实现的。

```
service SearchService {
    rpc Search (SearchRequest) returns (SearchResponse);
}
```

Dubbo 所做的支持

跨语言的服务开发涉及到多个方面，从服务定义、RPC 协议到序列化协议都要做到语言中立，同时还针对每种语言有对应的 SDK 实现。虽然得益于社区的贡献，现在 Dubbo 在多语言 SDK 实现上逐步有了起色，已经提供了包括 Java, Go, PHP, C#, Python, NodeJs, C 等版本的客户端或全量实现版本，但在以上提到的跨语言友好型方面，以上三点还是有很多可改进之处。

- 协议，上面我们已经分析过 Dubbo 协议既有的缺点，如果能在 HTTP/2 之上构建应用层协议，则无疑能避免这些弊端，同时最大可能的提高协议的穿透性，避免网关等协议转换组件的存在，更有利于链路上的流量管控。考虑到 gRPC 是构建在 HTTP/2 之上，并且已经是云原生领域推荐的通信协议，Dubbo 在第一阶段选择了直接支持 gRPC 协议作为当前的 HTTP/2 解决方案。我们也知道 gRPC 框架自身的弊端在于易用性不足以及服务治理能力欠缺（这也是目前绝大多数厂商不会直接裸用 gRPC 框架的原因），通过将其集成进 Dubbo 框架，用户可以方便的使用 Dubbo 编程模型 + Dubbo 服务治理 + gRPC 协议通信的组合。
- 服务定义，当前 Dubbo 的服务定义和具体的编程语言绑定，没有提供一种语言中立的服务描述格式，比如 Java 就是定义 Interface 接口，到了其他语言又得重新以另外的格式定义一遍。因此 Dubbo 通过支持 Protobuf 实现了语言中立的服务定义。
- 序列化，Dubbo 当前支持的序列化包括 Json、Hessian2、Kryo、FST、Java 等，而这其中支持跨语言的只有 Json、Hessian2，通用的 Json 有固有的性能问题，而 Hessian2 无论在效率还是多语言 SDK 方面都有所欠缺。为此，Dubbo 通过支持 Protobuf 序列化来提供更高效、易用的跨语言序列化方案。

示例

示例 1，使用 Dubbo 开发 gRPC 服务

gRPC 是 Google 开源的构建在 HTTP/2 之上的一个 PRC 通信协议。Dubbo 依赖其灵活的协议扩展机制，增加了对 gRPC (HTTP/2) 协议的支持。

目前的支持限定在 Dubbo Java 语言版本，后续 Go 语言或其他语言版本将会以类似方式提供支持。下面，通过一个简单的示例来演示如何在 Dubbo 中使用 gRPC 协议通信，详情参考：*https://github.com/apache/dubbo-samples/tree/master/dubbo-samples-grpc*

1. 定义服务 IDL

首先，通过标准的 Protobuf 协议定义服务如下：

```
syntax = "proto3";

   option java_multiple_files = true;
   option java_package = "io.grpc.examples.helloworld";
   option java_outer_classname = "HelloWorldProto";
   option objc_class_prefix = "HLW";

   package helloworld;

   // The greeting service definition.
   service Greeter {
     // 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;
   }
```

```

```

在此，我们定义了一个只有一个方法 sayHello 的 Greeter 服务，同时定义了方法的入参和出参，

2. Protobuf Compiler 生成 Stub

1. 定义 Maven Protobuf Compiler 插件工具。这里我们扩展了 Protobuf 的 Compiler 工具，

以用来生成 Dubbo 特有的 RPC stub，此当前以 Maven 插件的形式发布。

```
<plugin>
  <groupId>org.xolstice.maven.plugins</groupId>
  <artifactId>protobuf-maven-plugin</artifactId>
  <version>0.5.1</version>
  <configuration>
    <protocArtifact>com.google.protobuf:protoc:3.7.1:exe:${os.detecte
    </protocArtifact>
    <pluginId>dubbo-grpc-java</pluginId>
    <pluginArtifact>org.apache.dubbo:protoc-gen-dubbo-java:1.19.0-SNA
    <outputDirectory>build/generated/source/proto/main/java</outputDi
    <clearOutputDirectory>false</clearOutputDirectory>
    <pluginParameter>grpc</pluginParameter>
  </configuration>
  <executions>
    <execution>
      <goals>
        <goal>compile</goal>
        <goal>compile-custom</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

其中，pluginArtifact 指定了 Dubbo 定制版本的 Java Protobuf Compiler 插件，通过这个插件来在编译过程中生成 Dubbo 定制版本的 gRPC stub。

```
<pluginArtifact>org.apache.dubbo:protoc-gen-dubbo-java:1.19.0-SNAPSHOT
```

由于 protoc-gen-dubbo-java 支持 gRPC 和 Dubbo 两种协议，可生成的 stub 类型，默认值是 gRPC，关于 dubbo 协议的使用可参见 使用 Protobuf 开发 Dubbo 服务。

```
<pluginParameter>grpc</pluginParameter>
```

3. 生成 Java Bean 和 Dubbo-gRPC stub

```
# 运行以下 maven 命令
$ mvn clean compile
```

生成的 Stub 和消息类 如下：
重点关注 GreeterGrpc，包含了所有 gRPC 标准的 stub 类/方法，同时增加了 Dubbo 特定的接口，之后 Provider 端的服务暴露和 Consumer 端的服务调用都将依赖这个接口。

```
/**
* Code generated for Dubbo
*/
public interface IGreeter {
  default public io.grpc.examples.helloworld.HelloReply     sayHello(
    throw new UnsupportedOperationException("No need to override this
  }
  default public com.google.common.util.concurrent.ListenableFuture<i
    io.grpc.examples.helloworld.HelloRequest request) {
    throw new UnsupportedOperationException("No need to override this
  }
  public void sayHello(io.grpc.examples.helloworld.HelloRequest reque
                       io.grpc.stub.StreamObserver<io.grpc.examples.h
}
```

4.业务逻辑开发

继承 GreeterGrpc.GreeterImplBase （来自第 2 步），编写业务逻辑，这点和原生 gRPC 是一

致的。

```
package org.apache.dubbo.samples.basic.impl;

import io.grpc.examples.helloworld.GreeterGrpc;
import io.grpc.examples.helloworld.HelloReply;
import io.grpc.examples.helloworld.HelloRequest;
import io.grpc.stub.StreamObserver;

public class GrpcGreeterImpl extends GreeterGrpc.GreeterImplBase {
  @Override
  public void sayHello(HelloRequest request, StreamObserver<HelloRepl
    System.out.println("Received request from client.");
    System.out.println("Executing thread is " + Thread.currentThread(
    HelloReply reply = HelloReply.newBuilder()
      .setMessage("Hello " +    request.getName()).build();
    responseObserver.onNext(reply);
    responseObserver.onCompleted();
  }
}
```

5.Provider 端暴露 Dubbo 服务

以 Spring XML 为例:

```
<dubbo:application name="demo-provider"/>

<!-- 指定服务暴露协议为 gRPC -->
<dubbo:protocol id="grpc" name="grpc"/>

<dubbo:registry address="zookeeper://${zookeeper.address:127.0.0.1}:2

<bean id="greeter" class="org.apache.dubbo.samples.basic.impl.GrpcGre

<!-- 指定 protoc-gen-dubbo-java 生成的接口 -->
<dubbo:service interface="io.grpc.examples.helloworld.GreeterGrpc$IGr
```

```
public static void main(String[] args) throws Exception {
  ClassPathXmlApplicationContext context =
    new ClassPathXmlApplicationContext("spring/dubbo-demo-provider.xml
  context.start();

  System.out.println("dubbo service started");
  new CountDownLatch(1).await();
}
```

6.引用 Dubbo 服务

```
<dubbo:application name="demo-consumer"/>

<dubbo:registry address="zookeeper://${zookeeper.address:127.0.0.1}:21

<!-- 指定 protoc-gen-dubbo-java 生成的接口 -->
<dubbo:reference id="greeter" interface="io.grpc.examples.helloworld.G
```

```
public static void main(String[] args) throws IOException {
  ClassPathXmlApplicationContext context =
    new ClassPathXmlApplicationContext("spring/dubbo-demo-consumer.xm
  context.start();

  GreeterGrpc.IGreeter greeter = (GreeterGrpc.IGreeter) context.getBe

  HelloReply reply = greeter.sayHello(HelloRequest.newBuilder().setNa
  System.out.println("Result: " + reply.getMessage());

  System.in.read();
}
```

示例1附：高级用法

1、异步调用

再来看一遍 protoc-gen-dubbo-java 生成的接口：

```
/**
* Code generated for Dubbo
*/
public interface IGreeter {
  default public HelloReply sayHello(HelloRequest request) {
    // ......
  }
  default public ListenableFuture<HelloReply> sayHelloAsync(HelloRequ
    // ......
  }
  public void sayHello(HelloRequest request, StreamObserver<HelloRepl
}
```

这里为 sayHello 方法生成了三种类型的重载方法，分别用于同步调用、异步调用和流式调用，如果消费端要进行异步调用，直接调用 sayHelloAsync() 即可：

```
public static void main(String[] args) throws IOException {
  // ...
  GreeterGrpc.IGreeter greeter = (GreeterGrpc.IGreeter) context.getBea
  ListenableFuture<HelloReply> future =
    greeter.sayHAsyncello(HelloRequest.newBuilder().setName("world!").
  // ...
}
```

2、高级配置

由于当前实现方式是直接集成了 gRPC-java SDK，因此很多配置还没有和 Dubbo 侧对齐，或者还没有以 Dubbo 的配置形式开放，因此，为了提供最大的灵活性，我们直接把 gRPC-java 的配置接口暴露了出来。

绝大多数场景下，你可能并不会用到以下扩展，因为它们更多的是对 gRPC 协议的拦截或者

HTTP/2 层面的配置。同时使用这些扩展点可能需要对 HTTP/2 或 gRPC 有基本的了解。

扩展点

目前支持的扩展点如下：

- org.apache.dubbo.rpc.protocol.grpc.interceptors.ClientInterceptor
- org.apache.dubbo.rpc.protocol.grpc.interceptors.GrpcConfigurator
- org.apache.dubbo.rpc.protocol.grpc.interceptors.ServerInterceptor
- org.apache.dubbo.rpc.protocol.grpc.interceptors.ServerTransportFilter

GrpcConfigurator 是最通用的扩展点，我们以此为例来说明一下，其基本定义如下：

```
public interface GrpcConfigurator {
  // 用来定制 gRPC NettyServerBuilder
  default NettyServerBuilder configureServerBuilder(NettyServerBuilde
    return builder;
  }
  // 用来定制 gRPC NettyChannelBuilder
  default NettyChannelBuilder configureChannelBuilder(NettyChannelBui
    return builder;
  }
  // 用来定制 gRPC CallOptions, 定义某个服务在每次请求间传递数据
  default CallOptions configureCallOptions(CallOptions options, URL u
    return options;
  }
}
```

以下是一个示例扩展实现：

```
public class MyGrpcConfigurator implements GrpcConfigurator {
  private final ExecutorService executor = Executors
    .newFixedThreadPool(200, new NamedThreadFactory("Customized-grpc"

  @Override
  public NettyServerBuilder configureServerBuilder(NettyServerBuilder
    return builder.executor(executor);
  }

  @Override
  public NettyChannelBuilder configureChannelBuilder(NettyChannelBuil
{
    return builder.flowControlWindow(10);
  }

  @Override
  public CallOptions configureCallOptions(CallOptions options, URL ur
    return options.withOption(CallOptions.Key.create("key"), "value")
  }
}
```

配置为 Dubbo SPI，`resources/META-INF/services 增加配置文件。

```
default=org.apache.dubbo.samples.basic.comtomize.MyGrpcConfigurator
```

1.指定 Provider 端线程池

默认用的是 Dubbo 的线程池，有 fixed (默认)、cached、direct 等类型。以下演示了切换为业务自定义线程池。

```
private final ExecutorService executor = Executors
              .newFixedThreadPool(200, new NamedThreadFactory("Customi

public NettyServerBuilder configureServerBuilder(NettyServerBuilder bu
{
return builder.executor(executor);
}
```

2. 指定 Consumer 端限流值

设置 Consumer 限流值为 10。

```
@Override
public NettyChannelBuilder configureChannelBuilder(NettyChannelBuilder
{
  return builder.flowControlWindow(10);
}
```

3.传递附加参数

DemoService 服务调用传递 key。

```
@Override
public CallOptions configureCallOptions(CallOptions options, URL url)
  if (url.getServiceInterface().equals("xxx.DemoService")) {
    return options.withOption(CallOptions.Key.create("key"), "value");
  } else {
    return options;
  }
}
```

3、双向流式通信

代码中还提供了一个支持双向流式通信的示例，同时提供了拦截流式调用的 Interceptor 扩展示例实现。

* MyClientStreamInterceptor，工作在 client 端，拦截发出的请求流和接收的响应流。

* MyServerStreamInterceptor，工作在 server 端，拦截收到的请求流和发出的响应流。

双向流式通信的示例，详情点击：

https://github.com/apache/dubbo-samples/tree/master/dubbo-samples-grpc/src/main/java/org/apache/dubbo/samples/basic/impl/routeguide

4、TLS 配置

配置方式和 Dubbo 提供的[通用的 TLS 支持]()一致，具体请参见文档

示例 2， 使用 Protobuf 开发 Dubbo 服务

下面，我们以一个[具体的示例]()来看一下基于 Protobuf 的 Dubbo 服务开发流程。

1. 定义服务

通过标准 Protobuf 定义服务。

```
syntax = "proto3";

    option java_multiple_files = true;
    option java_package = "org.apache.dubbo.demo";
    option java_outer_classname = "DemoServiceProto";
    option objc_class_prefix = "DEMOSRV";

    package demoservice;

    // The demo service definition.
    service DemoService {
      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;
    }
```

这里定义了一个 DemoService 服务，服务只包含一个 sayHello 方法，同时定义了方法的入参和出参。

2. Compiler 编译服务

1. 引入 Protobuf Compiler Maven 插件，同时指定 protoc-gen-dubbo-java RPC 扩展。

```
<plugin>
  <groupId>org.xolstice.maven.plugins</groupId>
  <artifactId>protobuf-maven-plugin</artifactId>
  <version>0.5.1</version>
  <configuration>
  <protocArtifact>com.google.protobuf:protoc:3.7.1:exe:${os.detected.
  </protocArtifact>
  <pluginId>dubbo-grpc-java</pluginId>
  <pluginArtifact>org.apache.dubbo:protoc-gen-dubbo-java:1.19.0-SNAPS
  <outputDirectory>build/generated/source/proto/main/java</outputDire
  <clearOutputDirectory>false</clearOutputDirectory>
  <pluginParameter>dubbo</pluginParameter>
  </configuration>
  <executions>
  <execution>
    <goals>
    <goal>compile</goal>
    <goal>compile-custom</goal>
    </goals>
  </execution>
  </executions>
</plugin>
```

注意，这里与 Dubbo 对 gRPC 支持部分的区别在于：dubbo

2、生成 RPC stub

```
# 运行以下 maven 命令
$mvn clean compile
```

生成的 Java 类如下：

```
public final class DemoServiceDubbo {

```

```
private static final AtomicBoolean registered = new AtomicBoolean();

private static Class<?> init() {
 Class<?> clazz = null;
 try {
   clazz = Class.forName(DemoServiceDubbo.class.getName());
   if (registered.compareAndSet(false, true)) {
    org.apache.dubbo.common.serialize.protobuf.support.ProtobufUtils.
      org.apache.dubbo.demo.HelloRequest.getDefaultInstance());
    org.apache.dubbo.common.serialize.protobuf.support.ProtobufUtils.
      org.apache.dubbo.demo.HelloReply.getDefaultInstance());
   }
 } catch (ClassNotFoundException e) {
   // ignore
 }
 return clazz;
}

private DemoServiceDubbo() {}

public static final String SERVICE_NAME = "demoservice.DemoService";

/**
       * Code generated for Dubbo
       */
public interface IDemoService {

 static Class<?> clazz = init();
 org.apache.dubbo.demo.HelloReply sayHello(org.apache.dubbo.demo.Hell

 java.util.concurrent.CompletableFuture<org.apache.dubbo.demo.HelloRe
   org.apache.dubbo.demo.HelloRequest request);

}
```

```
38 }
```

最值得注意的是 IDemoService 接口，它会作为 Dubbo 服务定义基础接口。

3、开发业务逻辑

从这一步开始，所有开发流程就和直接定义 Java 接口一样了。实现接口定义业务逻辑。

```
1  public class DemoServiceImpl implements DemoServiceDubbo.IDemoService
2    private static final Logger logger = LoggerFactory.getLogger(DemoSe
3
4    @Override
5    public HelloReply sayHello(HelloRequest request) {
6      logger.info("Hello " + request.getName() + ", request from consum
7      return HelloReply.newBuilder()
8        .setMessage("Hello " + request.getName() + ", response from pro
9                    + RpcContext.getContext().getLocalAddress())
10       .build();
11   }
12
13   @Override
14   public CompletableFuture<HelloReply> sayHelloAsync(HelloRequest req
15     return CompletableFuture.completedFuture(sayHello(request));
16   }
17 }
```

4、配置 Provider

暴露 Dubbo 服务。

```
<dubbo:application name="demo-provider"/>

<dubbo:registry address="zookeeper://127.0.0.1:2181"/>

<dubbo:protocol name="dubbo"/>

<bean id="demoService" class="org.apache.dubbo.demo.provider.DemoServi

<dubbo:service interface="org.apache.dubbo.demo.DemoServiceDubbo$IDemo
```

```
public static void main(String[] args) throws Exception {
  ClassPathXmlApplicationContext context =
    new ClassPathXmlApplicationContext("spring/dubbo-provider.xml");
  context.start();
  System.in.read();
}
```

5、配置 Consumer

引用 Dubbo 服务。

```
<dubbo:application name="demo-consumer"/>

<dubbo:registry address="zookeeper://127.0.0.1:2181"/>

<dubbo:reference id="demoService" check="false" interface="org.apache.
```

```
public static void main(String[] args) throws Exception {
  ClassPathXmlApplicationContext context =
    new ClassPathXmlApplicationContext("spring/dubbo-consumer.xml");
  context.start();
  IDemoService demoService = context.getBean("demoService", IDemoServ
  HelloRequest request = HelloRequest.newBuilder().setName("Hello").b
  HelloReply reply = demoService.sayHello(request);
  System.out.println("result: " + reply.getMessage());
  System.in.read();
}
```

展望

RPC 协议是实现微服务体系互通的核心组件，通常采用不同的微服务通信框架则意味着绑定某一个特定的协议，如 Spring Cloud 基于 HTTP、gRPC 提供 gRPC over HTTP/2、Thrift Hessian 等都是自定义私有协议。

Dubbo 自身同样提供了私有的 Dubbo 协议，这样你也能基于 Dubbo 协议构建微服务。但除了单一协议之外，和以上所有框架不同的，Dubbo 最大的优势在于它能同时支持多协议的暴露和消费，再配合 Dubbo 多注册订阅的模型，可以让 Dubbo 成为桥接多种不同协议的微服务体系的开发框架，轻松的实现不同微服务体系的互调互通或技术栈迁移。

这篇文章详细讲解了 Dubbo 对 gRPC 协议的支持，再加上 Dubbo 之前已具备的对 REST、Hessian、Thrift 等的支持，使 Dubbo 在协议互调上具备了基础。我们只需要在服务发现模型上也能实现和这些不同体系的打通，就能解决不同技术栈互调和迁移的问题。关于这部分的具体应用场景以及工作模式，我们将在接下来的文章中来具体分析。

作者信息：

刘军，Github账号Chickenlj，Apache Dubbo PMC，项目核心维护者，见证了Dubbo从重启开源到Apache毕业的整个流程。现任职阿里巴巴中间件团队，参与服务框架、微服务相关工作，目前主要在

推动Dubbo开源的云原生化。

惊喜放送：云原生应用平台正在招聘中间件工程师/专家，可以到公众号主页的菜单栏“招聘”页面了解详细信息，有意向的同学发送邮件至 ken.lj@alibaba-inc.com。

本文缩略图：icon by 胡说Y道

Tips：

点下“在看”❤️

然后，公众号对话框内发送“**比心**”，试试手气？😆

本期奖品是来自淘宝心选的蒸汽眼罩。