RPC 协议之争和选型要点

作者：李林锋

阅读数：2193 2019 年 1 月 16 日 话题：架构 网络协议 最佳实践

> 《Netty 进阶之路》、《分布式服务框架原理与实践》作者李林锋深入剖析 RPC 协议之争和选型问题。李林锋此后还将在 InfoQ 上开设 Netty 专题持续出稿，感兴趣的同学可以持续关注。

1. 协议之争背景

1.1 RPC 调用的协议选择

RPC 调用的协议选择包含两部分：

1．协议栈：广义上协议栈可以分为公有协议和私有协议，例如 HTTP、SMPP、WebService 等都是公有协议；如果是某个公司或者组织内部自定义、自己使用的协议，没有被国际标准化组织接纳和认可的，往往划为私有协议，例如 Thrift 协议。

2．序列化方式：同一种协议也可以承载多种序列化方式，以 HTTP 协议为例，它可以承载文本类序列化方式，例如：XML、JSON 等，也可以承载二进制序列化方式，例如谷歌的 Protobuf。

不同的协议选择，对 RPC 调用的性能、开发难度和问题定位效率都有影响，因此，选择哪种协议，对 RPC 框架而言至关重要。由于各个协议都有自己的优缺点，因此很多框架在技术选型时都非常纠结。各种观点存在激烈交锋，有的看重性能和时延、有的更看重跨语言和可维护性。协议的选择是 RPC 框架构建的一个技术难点。

1.2 私有协议流行的原因

私有协议本质上是厂商内部发展和采用的标准，除非授权，其他厂商一般无权使用该协议。私有协议也称非标准协议，就是未经国际或国家标准化组织采纳或批准，由某个企业自己制订，协议实现细节不愿公开，只在企业自己生产的设备之间使用的协议。私有协议具有封闭性、垄断性、排他性等特点。如果网上大量存在

尽管私有协议具有垄断性的特征，但并非所有的私有协议设计者的初衷就是为了垄断。由于现代软件系统的复杂性，一个大型软件系统往往会被人为地拆分成多个模块，另外随着移动互联网的兴起，网站的规模也越来越大，业务的功能越来越多，为了能够支撑业务的发展，往往需要集群和分布式部署，这样，各个模块之间就要进行跨节点通信。

在传统的 Java 应用中，通常使用以下 4 种方式进行跨节点通信。

1．通过 RMI 进行远程服务调用。

2．通过 Java 的 Socket+Java 序列化的方式进行跨节点调用。

3．利用一些开源的 RPC 框架进行远程服务调用，例如 Facebook 的 Thrift，Google 的 gRPC 等。

4．利用标准的公有协议进行跨节点服务调用，例如 HTTP+XML、Restful+JSON 或者 WebService。

跨节点的远程服务调用，除了链路层的物理连接外，还需要对请求和响应消息进行编解码。在请求和应答消息本身以外，也需要携带一些其他控制和管理类指令，例如链路建立的握手请求和响应消息、链路检测的心跳消息等。当这些功能组合到一起之后，就会形成私有协议。

私有协议的优点：灵活性高，可以按照业务的使用场景来设计和优化，在某个公司或者组织内部使用时也可以按需定制和演进，所以大部分 RPC 框架都支持私有二进制协议，例如阿里的 Dubbo、华为的 ServiceComb、Apache 的 Thrift 等。

1.3 序列化方式

当进行远程跨进程服务调用时，需要把被传输的数据结构 / 对象序列化为字节数组或者 ByteBuffer。而当远程服务读取到 ByteBuffer 对象或者字节数组时，需要将其反序列化为原始的数据结构 / 对象。利用序列化框架可以实现上述转换工作。

Java 序列化从 JDK 1.1 版本就已经提供，它不需要添加额外的类库，只需实现 java.io.Serializable 并生成序列 ID 即可，因此，它从诞生之初就得到了广泛的应用。

但是在远程服务调用（RPC）时，很少直接使用 Java 序列化进行消息的编解码和传输，这又是什么原因呢？下面通过分析 Java 序列化的缺点来找出答案:

缺点 1：无法跨语言，是 Java 序列化最致命的问题。对于跨进程的服务调用，服务提供者可能会使用 C++ 或者其他语言开发，当我们需要和异构语言进程交互时，Java 序列化就难以胜任。由于 Java 序列化技术是 Java 语言内部的私有协议，其它语言并不支持，对于用户来说它完全是黑盒。对于 Java 序列化后的字节数组，别的语言无法进行反序列化，这就严重阻碍了它的应用。事实上，目前几乎所有流行的 Java RCP 通信框架，都没有使用 Java 序列化作为编解码框架，原因就在于它无法跨语言，而这些 RPC 框架往往需要支持跨语言调用。

缺点 2：相比于业界的一些序列化框架，Java 默认的序列化效能较低，主要体现在：序列化之后的字节数组体积较大，性能较低。在同等情况下，编码后的字节数组越大，存储的时候就越占空间，存储的硬件成本就越高，并且在网络传输时更占带宽，导致系统的吞吐量降低。Java 序列化后的码流偏大也一直被业界所诟病，导致它的应用范围受到了很大限制。

当前比较流行的序列化方式可以分为两大类:

2．私有的二进制类序列化方式：比较流行的有 Thrift 序列化框架、MessagePack 和谷歌的 Protobuf 框架。它的优点是性能高，缺点就是可读性差，支撑的工具链不健全。

2. RPC 协议选型要点

2.1 协议栈的选择

2.1.1 公有协议

尽管公有协议种类繁多，例如之前非常流行的 WebService、WADL 等，但目前来看，如果选择公有协议，HTTP 协议还是首选，具有 Rest 风格的 Restful + JSON 接口是当前最流行的方式。

RPC 协议如果选择 Restful + JSON，会带来如下几个优点：

1．践行 API First 理念：通过使用 Swagger YAML 定义 API，服务端和客户端都基于 API 定义，通过 Swagger 代码生成工具生成不同语言的接口和模型定义类库，客户端不需要从服务端导入接口定义类库，也不需要配置 Maven 依赖，这样就实现了双方依赖的解耦：

图 1 基于 Swagger 代码工具生成服务端和客户端代码

除了代码生成，利用 swagger editor 和 swagger ui 工具，可以在线定义和维护 API 接口的契约，实现接口 API 的在线化管理，更好的管控 API 变更，示例如下：

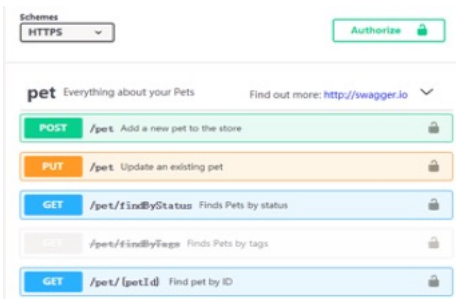

图 2 基于 Swagger UI 在线管理 Restful API

对于 RPC 接口的测试，由于是 Restful 风格的，利用一些开源的契约测试框架，可以方便的进行契约化测试：

语言非常重要。不同的业务场景，适合不同的语言，例如后端复杂业务逻辑使用 Java 开发效率更高，对于 API 网关或者边缘服务，适合 GO 语言。对于一些序列化框架，由于使用了一些特定语言的特性，例如 Exception、泛型等，这对于跨语言演进是个灾难，像 protostuff 就绑定了 Java 语言。

3．内部和外部 API 接口的统一：RPC 服务通常只开放给内部的客户端做调用，如果需要开放给外部的端侧、其它渠道调用，往往需要前置一个 API 网关或者 Edge Service，用来做安全接入、权限管理、统一流控、灰度发布等。组网示例如下所示：

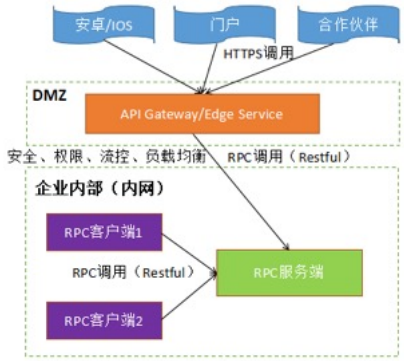

图 4 对 RPC 服务对外开放组网图

如果 RPC 调用选择 Restful API，则对外开放时，API 网关 /Edge Service 只要做安全等相关功能即可，消息可以透传转发给后端 RPC 服务。如果后端 RPC 服务选择私有协议，将私有协议直接开放给合作伙伴显然是不合适的，这就需要在 API 网关上为后端 RPC 服务定义 API 接口，同时做消息映射和转换，最终形成对外开放一套 API 接口，内部使用又一套 API 接口，但是功能却相同或者类似，这增加了接口的开发和维护工作量。

目前主流的 API Gateway 都支持直接导入 Swagger 定义的 API，自动生成并发布 API 接口，以 AWS 的 API Gateway 为例，如下所示：

图 5 AWS 基于 Swagger 接口定义生成 API（来自 AWS 官网示例教程）

当 RPC 服务开放给内部和外部是同一套 API 接口时，接口的开发和维护工作量都会减少很多。

4．问题定位更简单：HTTP 协议 + JSON 文本序列化方式，更容易调试，抓包的码流解读也更容易，相反如果是二进制私有协议，码流需要人工解读，难度较高。

2.1.2 私有协议

绝大多数的私有协议传输层都基于 TCP/IP，对于 Java 语言，可以利用 Netty 的 NIO TCP 协议栈进行私有协议的定制和开发。

私有协议的格式往往存在较大差异，但是对于大多数 RPC 框架的私有协议，往往会包含如下几个字段：

1．消息头：消息头中通常会包含校验码、消息长度、消息类型、消息 / 会话 ID、需要调用的 RPC 接口名、方法名，以及扩展的消息头，通常是个类似 Map 的结构，用于隐式传参。

下面以基于 Netty 开发的 Netty 协议栈为例进行说明，它的协议栈模型如下所示：

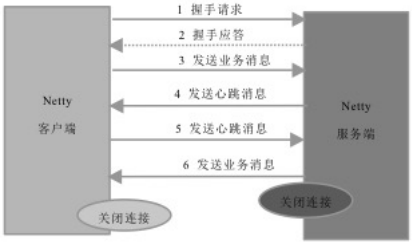

图 6 Netty 协议栈通信交互图

Netty 协议栈承载了业务内部各模块之间的消息交互和 RPC 调用，它的主要功能包括：

1．基于 Netty 的 NIO 通信框架，提供高性能的异步通信能力。

2．提供消息的编解码框架，可以实现 POJO 的序列化和反序列化。

3．提供基于 IP 地址的白名单接入认证机制。

4．链路的有效性校验机制。

5．链路的断连重连机制。

RPC 协议栈交互的流程说明如下：

1．Netty 协议栈客户端发送握手请求消息，携带节点 ID 等有效身份认证信息。

2．Netty 协议栈服务端对握手请求消息进行合法性校验，包括节点 ID 有效性校验、节点重复登录校验和 IP 地址合法性校验，校验通过后，返回登录成功的握手应答消息。

3．链路建立成功之后，客户端发送业务消息。

4．链路成功之后，服务端发送心跳消息。

5．链路建立成功之后，客户端发送心跳消息。

6．链路建立成功之后，服务端发送业务消息。

7．服务端退出时，服务端关闭连接，客户端感知对方关闭连接后，被动关闭客户端连接。

Netty 协议栈的消息定义如下：

消息包含消息头和消息体：

名 称	类 型	长度	描 述
header	Header	变长	消息头定义
body	Object	变长	对于请求消息，它是方法的参数（作为示例，只支持携带一个参数）；对于响应消息，它是返回值

名 称	类 型	长 度	描 述
crcCode	整型 int	32	Netty 消息的校验码，它由三部分组成： 1）0xABEF：固定值，表明该消息是 Netty 协议消息，2 个字节； 2）主版本号：1～255，1 个字节； 3）次版本号：1～255，1 个字节。 crcCode = 0xABEF + 主版本号 + 次版本号

续表

名 称	类 型	长 度	描 述
length	整型 int	32	消息长度，整个消息，包括消息头和消息体
sessionID	长整型 long	64	集群节点内全局唯一，由会话 ID 生成器生成
type	Byte	8	0：业务请求消息； 1：业务响应消息； 2：业务 ONE WAY 消息（既是请求又是响应消息）； 3：握手请求消息； 4：握手应答消息； 5：心跳请求消息； 6：心跳应答消息。
priority	Byte	8	消息优先级：0～255
interfaceName	String	变长	
methodName	String	变长	
attachment	Map<String, Object>	变长	可选字段，用于扩展消息头

对于私有协议栈的构建，需要考虑到如下几点，整体成本较高：

1．支持的数据结构类型，以及采用的序列化方式。

2．握手和接入认证。

3．心跳检测机制。

4．断连和重连机制。

5．并发连接数的控制。

6．异常、畸形码流的检测和保护。

7．流量限制和整形。

8．连接池。

9．网络闪断、宕机保护、消息缓存和重发机制等。

1. 灵活性、可定制性更好，可以针对业务特定场景做优化。

2. 可以实现更高的性能。

2.2 序列化框架

2.2.1 选型的原则

如果对性能要求不高，则建议选用 JSON，否则可以选择一些业界主流的序列化框架，需要注意的是，对于序列化框架的选择，一定要考虑跨语言性，如果绑定特定语言，会对未来 RPC 框架支持多语言带来极大的困难。

2.2.2 Google 的 Protobuf

Protobuf 全称 Google Protocol Buffers，它由谷歌开源而来，在谷歌内部久经考验。它将数据结构以.proto 文件进行描述，通过代码生成工具可以生成对应数据结构的 POJO 对象和 Protobuf 相关的方法和属性。

它的特点如下:

1. 结构化数据存储格式（XML，JSON 等）。

2. 高效的编解码性能。

3. 语言无关、平台无关、扩展性好。

4. 官方支持 Java、C++ 和 Python 三种语言（社区会支持更多中语言）。

Protobuf 的优点主要有两个:

1. IDL 契约：利用数据描述文件对数据结构进行说明，可以实现语言和平台无关，通过标识字段的顺序，可以实现协议的前向兼容，同时提供代码生成工具，可以生成各种语言的服务端和客户端代码。

2. 性能：相比于其它序列化框架，它的性能更优，数据对比如下所示:

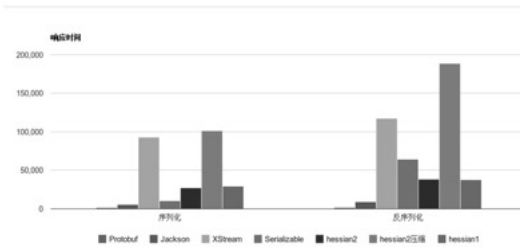

图 7 Protobuf 编解码和其他几种序列化框架的响应时间对比

2.2.3 Apache Thrift

Thrift 源于 Facebook，在 2007 年 Facebook 将 Thrift 作为一个开源项目提交给 Apache 基金会。对于当时的 Facebook 来说，创造 Thrift 是为了解决 Facebook 各系统间大数据量的传输通信以及系统之间语言环境不同需要跨平台的特性，因此 Thrift 可以支持多种程序语言，如 C++、Cocoa、Erlang、Haskell、Java、Ocami、Perl、PHP、Python、Ruby 和 Smalltalk。

在多种不同的语言之间通信，Thrift 可以作为高性能的通信中间件使用，它支持数据（对象）序列化和多种类型的 RPC 服务。Thrift 适用于静态的数据交换，需要先确定好它的数据结构，当数据结构发生变化时，必须重新编辑 IDL 文件，生成代码和编译，这一点跟其他 IDL 工具相比可以视为是 Thrift 的弱项。Thrift 适用

Thrift 主要由 5 部分组成:

1. 语言系统以及 IDL 编译器：负责由用户给定的 IDL 文件生成相应语言的接口代码；
2. TProtocol：RPC 的协议层，可以选择多种不同的对象序列化方式，如 JSON 和 Binary；
3. TTransport：RPC 的传输层，同样可以选择不同的传输层实现，如 socket、NIO、MemoryBuffer 等；
4. TProcessor：作为协议层和用户提供的服务实现之间的纽带，负责调用服务实现的接口；
5. TServer：聚合 TProtocol、TTransport 和 TProcessor 等对象。

需要重点关注的是编解码框架，与之对应的就是 TProtocol。由于 Thrift 的 RPC 服务调用和编解码框架绑定在一起，所以，通常我们使用 Thrift 的时候会采取 RPC 框架的方式。但是，它的 TProtocol 编解码框架还是可以以类库的方式独立使用的。

与 Protobuf 比较类似的是，Thrift 通过 IDL 描述接口和数据结构定义，它支持 8 种 Java 基本类型、Map、Set 和 List，支持可选和必选定义，功能非常强大。因为可以定义数据结构中字段的顺序，所以它也可以支持协议的前向兼容。

Thrift 支持三种比较典型的编解码方式。

- 通用的二进制编解码
- 压缩二进制编解码
- 优化的可选字段压缩编解码

由于支持二进制压缩编解码，Thrift 的编解码性能表现也相当优异，远远超过 Java 序列化和 RMI 等。

2.2.4 MessagePack 序列化框架

MessagePack 是一个高效的二进制序列化框架，它像 JSON 一样支持不同语言间的数据交换，但是它的性能更快，序列化之后的码流也更小。MessagePack 提供了对多语言的支持，官方支持的语言如下：Java、Python、Ruby、Haskell、C#、OCaml、Lua、Go、C、C++ 等。

MessagePack 的 Java API 非常简单，如果使用 MessagePack 进行开发，只需要导入 MessagePack maven 依赖：

复制代码

```
<dependency>

    <groupId>org.msgpack</groupId>

    <artifactId>msgpack</artifactId>

    <version>${msgpack.version}</version>

</dependency>

```

复制代码

```
List<String> src = new ArrayList<String>();

src.add("msgpack");

src.add("kumofs");

src.add("viver");

MessagePack msgpack = new MessagePack();

byte[] raw = msgpack.write(src);

List<String> dst1 =

msgpack.read(raw, Templates.tList(Templates.TString));
```

2.2.5. 选型建议

上面列举的几种序列化框架各有优缺点，如果选用，则建议从如下几个维度做对比：

1. 支持数据类型的丰富度。

2. 性能对比测试，主要包括 序列化和反序列化的耗时、CPU 和内存占用，以及序列化之后的码流大小。

3. 尽管都支持跨语言，但是由于支持的语言丰富度不同，业务需要根据自己 RPC 框架未来可能支持的语言做选择。

3. RPC 协议栈实践

3.1 Restful API 的优化

使用 Restful API 可以带来很多收益：

- API 接口更加规范和标准，可以通过 Swagger API 规范来描述服务接口，并生成客户端和服务端代码。
- Restful API 可读性更好，也更容易维护。
- 服务提供者和消费者基于 API 契约，双方可以解耦，不需要在客户端引入 SDK 和类库的直接依赖，未来的独立升级也更方便。
- 内外可以使用同一套 API，非常容易开放给外部或者合作伙伴使用，而不是对内和对外维护两套不同协议的 API。

通常，对外开放的 API 使用 Restful 是通用的做法，但是在系统内部，例如商品中心和订单中心，RPC 调用使用 Restful 风格的 API 作为微服务的 API，却可能存在性能风险。

3.1.1 HTTP1.X 的性能问题

如果采用的 Restful API 底层使用的 HTTP 协议栈是同步阻塞 I/O，则服务端的处理性能将大打折扣：

1. 性能问题：一连接一线程模型导致服务端的并发接入数和系统吞吐量受到极大限制。

3．可维护性问题：I/O 线程数无法有效控制、资源无法有效共享（多线程并发问题），系统可维护性差。

如果 HTTP 协议栈采用了异步非阻塞 I/O 模型（例如 Netty、Servlet3.X 版本），则可以解决同步阻塞 I/O 的问题，带来如下收益：

1．同一个 I/O 线程可以并行处理多个客户端链接，有效降低了 I/O 线程数量，提升了资源调度利用率

2．读写操作都是非阻塞的，不会因为对端处理慢、网络时延大等导致的 I/O 线程被阻塞

相比于 TCP 类协议，例如 Thrift, 采用了非阻塞 I/O 的 HTTP/1.X 协议仍然存在性能问题，原因如下所示：

如果 HTTP 协议栈采用了异步非阻塞 I/O 模型（例如 Netty、Servlet3.X 版本），则可以解决同步阻塞 I/O 的问题，带来如下收益：

- 同一个 I/O 线程可以并行处理多个客户端链接，有效降低了 I/O 线程数量，提升了资源调度利用率。
- 读写操作都是非阻塞的，不会因为对端处理慢、网络时延大等导致的 I/O 线程被阻塞。

相比于 TCP 类协议，例如 Thrift, 采用了非阻塞 I/O 的 HTTP/1.X 协议仍然存在性能问题，原因如下所示：

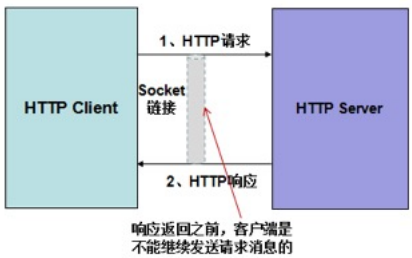

图 8 HTTP/1.X 请求 - 响应模式

由于 HTTP 协议是无状态的，客户端发送请求之后，必须等待接收到服务端响应之后，才能继续发送请求（非 websocket、pipeline 等模式）。在某一个时刻，链路上只存在单向的消息流，实际上把 TCP 的双工变成了单工模式。如果服务端响应耗时较大，则单个 HTTP 链路的通信性能严重下降，只能通过不断的新建连接来提升 I/O 性能。但这也会带来很多副作用，例如句柄数的增加、I/O 线程的负载加重等。显而易见，修 8 条单向车道的成本远远高于修一条双向 8 车道的成本。

除了无状态导致的链路传输性能差之外，HTTP/1.X 还存在如下几个影响性能的问题：

- HTTP 客户端超时之后，由于协议是无状态的，客户端无法对请求和响应进行关联，只能关闭链路重连，反复的建链会增加成本开销和时延（如果客户端选择不关闭链路，继续发送新的请求，服务端可能会把上一条客户端认为超时的响应返回回去，也可能按照 HTTP 协议规范直接关闭链路，无路哪种处理，都会导致链路被关闭）。如果采用传统的 RPC 私有协议，请求和响应可以通过消息 ID 或者会话 ID 做关联，某条消息的超时并不需要关闭链路，只需要丢弃该消息重发即可。
- HTTP 本身包含文本类型的协议消息头，占用一些字节。另外，采用 JSON 类文本的序列化方式，报文相比于传统的私有 RPC 协议也大很多，降低了传输性能。
- 服务端无法主动推送响应。

如果业务对性能和资源成本要求非常苛刻，在选择使用基于 HTTP/1.X 的 Restful API 代替私有 RPC API（通常是基于 TCP 的二进制私有协议）时就要三思；反之，如果业务对性能要求较低，或者在硬件成本和开放

如果选择 Restful API 作为内部 RPC 或者微服务的接口协议，则建议使用 HTTP/2.0 协议来承载，它的优点如下：支持双向流、消息头压缩、单 TCP 的多路复用、服务端推送等特性, 某个 RPC 调用超时也不需要关闭 HTTP 连接，只需要关闭对应的 Stream 流即可，这样可以避免大量超时时频繁的 HTTP 连接重建，有效解决传统 HTTP/1.X 协议遇到的问题，效果与 RPC 的 TCP 私有协议接近，采用 HTTP/2 的 gRPC Steaming 通信模式示例如下：

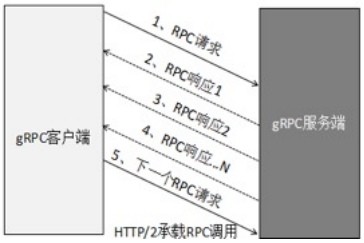

图 9 gRPC 基于 HTTP/2 的服务端 streaming 调用

3.2 Apache ServiceComb 的多协议实践

3.2.1 标准化的服务契约

ServiceComb 使用 yaml 文件格式定义服务契约，推荐使用 Swagger Editor 工具来编写契约，可检查语法格式及自动生成 API 文档。服务契约与具体协议无关系，无论使用 RPC 二进制协议通信还是使用标准的 Restful + JSON，都可以使用 Swagger API 来描述和定义微服务接口。

接口定义完成之后，将契约文件放到 "resources/microservices" 或者 "resources/application" 目录下即可，目录结构如下所示：

```
resources
  - microservices
    - serviceName #微服务名
      - schemaId.yaml #schema接口的契约
  - applications
    - appId #应用ID
      - serviceName #微服务名
        - schemaId.yaml #schema接口的契约
```

图 10 微服务接口契约定义

3.2.2 通信协议

ServiceComb 实现了两种网络通道，包括 Rest 和 Highway，均支持 TLS 加密传输。其中，REST 网络通道将服务以标准 Restful 形式发布，调用端兼容直接使用 Http client 使用标准 Restful 形式进行调用。

Rest 协议栈支持两种实现方式：

1．REST over Servlet 对应使用 web 容器部署运行，需要新建一个 servlet 工程将微服务包装起来，打成 war 包，加载到 web 容器中启动运行。

2．REST over Vertx 通信通道对应使用 standalone 部署运行模式，可直接通过 main 函数拉起。使用 REST over Vertx 网络通道需要在 maven pom 文件中添加如下依赖：

```
<dependency>
    <groupId>org.apache.servicecomb</groupId>
    <artifactId>transport-rest-vertx</artifactId>
</dependency>
```

图 11 REST over Vertx 通信方式

使用 Highway 网络通道需要在 maven pom 文件中添加如下依赖：

```
<dependency>
    <groupId>org.apache.servicecomb</groupId>
    <artifactId>transport-highway</artifactId>
</dependency>
```

图 12 Highway 通信方式

ServiceComb 的通信协议有如下几个特点：

1．协议与服务接口契约定义没关系，使用 Swagger 定义的 API 接口，仍然可以使用二进制 Highway 私有协议进行 RPC 调用。

2．协议与业务代码无耦合关系，一套业务接口，可以选择发布成哪种协议。支持同一个服务接口发布成多种协议，客户端可以根据自己的需要选择哪种协议方式调用。

3.2.3 对 HTTP/2 协议的支持

ServiceComb 提供了对 HTTP/2 的支持，用于解决传统 HTTP/1.1 的性能问题，它支持 2 种 HTTP/2 通信方式：

1．h2(Http2 + TLS)：服务端在配置服务监听地址时，可以通过在地址后面追加?sslEnabled=true 开启 TLS 通信，示例如下：

```
servicecomb:
  rest:
    address: 0.0.0.0:8080?sslEnabled=true&protocol=http2
  highway:
    address: 0.0.0.0:7070?sslEnabled=true&protocol=http2
```

图 13 h2 通信方式配置

2．h2c(Http2 without TLS)：服务端在配置服务监听地址时，可以通过在地址后面追加?protocol=http2 启用 h2c 通信：

```
servicecomb:
  rest:
    address: 0.0.0.0:8080?protocol=http2
  highway:
    address: 0.0.0.0:7070?protocol=http2
```

图 14 h2c 通信方式配置

3.2.4. 性能对比数据

Restful 和 Highway（私有二进制协议）两种协议在不同模式的性能对比数据如下所示：

环境	transport	测试驱动	TPS	Latency(ms)	consumer CPU	producer CPU
两台华为云C3 VM 8C/16G 应答1K的字符串 VM 1部署： • consumer VM 2部署： • producer • ServiceCenter	highway	reactive 250并发	572349	0.435	551%	457%
		同步 200线程	362138	0.55	742%	586%
	RESTful	reactive 100并发	292787	0.339	767%	770%
		同步 100线程	181664	0.55	737%	755%

图 15 性能对比数据

通过上述性能测试数据对比可以发现，私有的二进制协议 Highway（TCP + Protobuf）比 Restful+JSON 协议整体性能高 2 倍 + 左右。如果 Restful+JSON 采用 HTTP/1.1 承载，无法实现链路的多路复用，一旦 RPC 调用超时就会频繁重建链路，可靠性相比于 TCP 私有协议会差很多。

无论是选择 Restful（HTTP 协议）还是私有二进制协议，总是存在各自的优缺点，对于比较复杂的业务场景，可能多种协议都需要支持，这就要求 RPC 框架必须实现接口定义与协议本身的解耦，即业务可以平滑的切换协议，上层应用不需要感知。

4.2 RPC 协议的定制和扩展

一个比较好的协议往往具备较强的扩展性，协议的扩展主要体现在两点：

1．协议栈本身的扩展，例如基于协议框架扩展出另一种协议。

2．协议栈采用的序列化框架的扩展。

对于 RPC 框架而言，更有价值的是序列化框架的扩展，可以在协议头中增加一个标识协议类型的字段，RPC 框架根据协议类型来调用对应的序列化框架，实现业务自定义的序列化和反序列化。

4.3 一些技术难点

不同的序列化框架，对数据类型的支持不同，对字段是否支持乱序也存在差异，所以，从某种程度看，协议和序列化方式完全与业务接口解耦也是很困难的，会有很多约束和限制，所以需要辨证的看待接口和协议的解耦，尽量做到对业务的影响最小、以及约束和限制的规范化。

5. 作者简介

李林锋，10 年 Java NIO、平台中间件设计和开发经验，精通 Netty、Mina、分布式服务框架、API Gateway、PaaS 等,《Netty 进阶之路》、《分布式服务框架原理与实践》作者。目前在华为终端应用市场负责业务微服务化、云化、全球化等相关设计和开发工作。

联系方式：新浪微博 Nettying 微信：Nettying

Email：neu_lilinfeng@sina.com

延伸阅读：

深入剖析通信层和 RPC 调用的异步化 (上)
深入剖析通信层和 RPC 调用的异步化 (下)

架构 网络协议 最佳实践