二

28 实战总结：RPC 实战总结与进阶延伸

经过前面几节的实战课，我们已经初步完成了一个 RPC 框架原型，其中串联了 RPC 框架所涉及的大部分核心知识点。纸上得来终觉浅，绝知此事要躬行，编码是每个程序员的基本功，一定要亲自动手做一遍，不要停留在纸上谈兵。虽然 RPC 框架原型已经可以运行起来了，但是离生产级使用还差得很远，例如性能、高可用等。本节课我会做一个有关知识点的总结回顾，并结合业界成熟的 RPC 框架再做一些知识补充，希望对你提升系统设计能力所有帮助。

实战知识点总结

Netty 服务端启动

Netty 提供了 ServerBootstrap 引导类作为程序启动入口，ServerBootstrap 将 Netty 核心组件像搭积木一样组装在一起，服务端启动过程我们需要完成以下三个基本步骤：

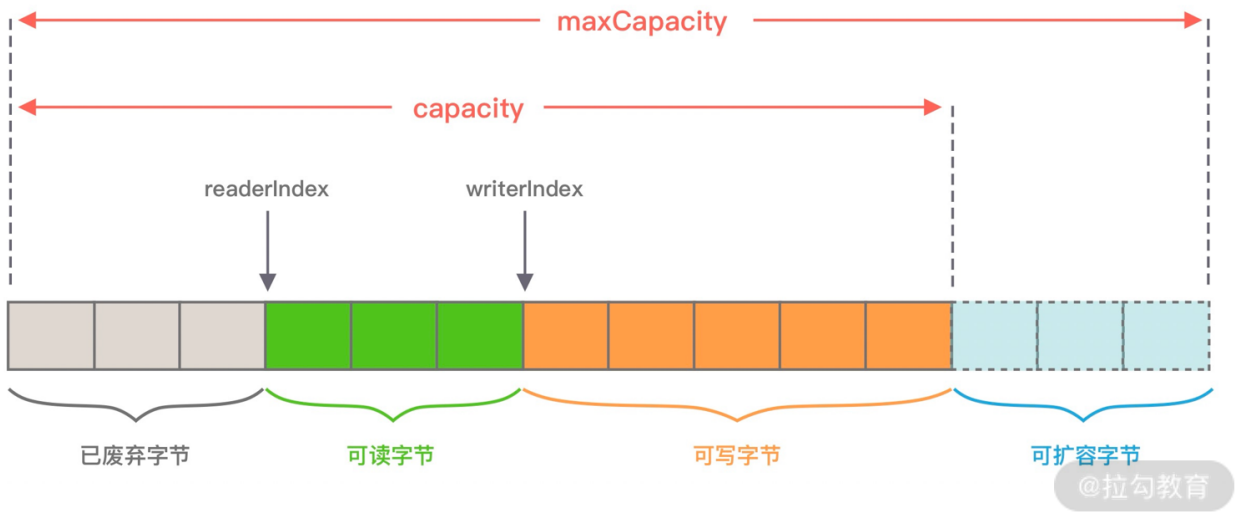
- 配置线程池。Netty 是采用 Reactor 模型进行开发的，在大多数场景下，我们采用的都是主从多线程 Reactor 模型。
- Channel 初始化。设置 Channel 类型，并向 ChannelPipeline 中注册 ChannelHandler，此外可以按需设置 Socket 参数以及用户自定义属性。
- 端口绑定。调用 bind() 方法会真正触发启动，sync() 方法则会阻塞，直至整个启动过程完成。

自定义通信协议

一个完备的网络协议需要具备的基本要素：魔数、协议版本号、序列化算法、报文类型、长度域字段、请求数据、保留字段。在实现协议编解码时经常用到两个重要的抽象类：**MessageToByteEncoder 编码器**和**ByteToMessageDecoder 解码器**。Netty 也提供了很多开箱即用的拆包器，推荐最广泛使用的 LengthFieldBasedFrameDecoder，它可以满足实际项目中的大部分场景。如果对 LengthFieldBasedFrameDecoder 的参数不够熟悉，实际直接使用 ByteBuf 反而更加直观，根据个人喜好按需选择。

ByteBuf

ByteBuf 是必须要掌握的核心工具类，并且能够理解 ByteBuf 的内部构造。ByteBuf 包含三个指针：**读指针 readerIndex**、**写指针 writeIndex**、**最大容量 maxCapacity**，根据指针的位置又可以将 ByteBuf 内部结构可以分为四个部分：废弃字节、可读字节、可写字节和可扩容字节。如下图所示。



Pipeline & ChannelHandler

ChannelPipeline 和 ChannelHandler 也是我们在平时应用开发的过程中打交道最多的组件，这两个组件为用户提供了 I/O 事件的全部控制权。ChannelPipeline 是双向链表结构，包含 ChannelInboundHandler 和 ChannelOutboundHandler 两种处理器。Inbound 事件和 Outbound 事件的传播方向相反，Inbound 事件的传播方向为 Head -> Tail，而 Outbound 事件传播方向是 Tail -> Head。在设计之初一定要梳理清楚 Inbound 和 Outbound 处理的传递顺序，以及数据模型之间是如何转换的。

注册中心

注册中心是 RPC 框架中一个非常重要的组件，主要用于实现服务的注册和发现。目前主流的注册中心有 ZooKeeper、Eureka、Etcd、Consul、Nacos 等，到底选择 CP 还是 AP 类型的注册中心呢？没有最好的选择，需要根据实际的业务场景进行技术选型。对于 RPC 框架而言，应当弱依赖于注册中心，即使注册中心出现问题，也不应该影响服务正常使用。所以建议使用 AP 类型的注册中心，在实现服务发现的场景下相比 CP 类型的注册中心有性能优势，整个集群是不存在 Leader、Flower 概念的，如果其中一个节点挂了，请求会立刻转移到其他节点上，通过牺牲强一致性来保证高可用性。

当服务节点下线时，注册中心需要及时通知服务消费者该节点已经下线了，否则可能会造成部分服务调用出现问题。实现服务优雅下线比较好的方式是采用主动通知 + 心跳检测的方

案，心跳检测可以由节点或者注册中心负责，例如注册中心可以向服务节点每 60s 发送一次心跳包，如果 3 次心跳包都没有收到请求结果，可以认为该服务节点已经下线。心跳检测通常也是客户端和服务端之间通知对方存活状态的一种机制，下文我会给你展示心跳检测的基本实现方式。

动态代理和反射调用

如果想做到 RPC 底层细节对服务消费者无感知，就无法绕开动态代理。动态代理提供了一种能够在运行时动态构建代理类以及动态调用目标方法的机制，我们必须创建一个接口代理对象，在代理对象中实现编码、请求调用、解码等操作。

常用的动态代理实现有 JDK 动态代理和 Cglib 动态代理，选择哪种动态代理技术需要根据场景有的放矢，需要做好性能压测。JDK 动态代理所代理的对象必须实现一个或者多个接口，生成的代理类也是接口的实现类，然后通过 JDK 动态代理是通过反射调用的方式代理类中的方法，不能代理接口中不存在的方法。Cglib 动态代理相比 JDK 动态代理更加灵活，Cglib 是通过字节码技术对指定类生成一个子类，并重写其中的方法，所以代理类的类型是不受限制的。

服务提供者在接收到 RPC 请求后，需要通过反射机制执行真实的方法调用。为了加速服务接口调用的性能，可以采用 Cglib 提供的 FastClass 机制直接调用方法，相比于反射性能更高。FastClass 机制并没有采用反射的方式调用被代理的方法，而是运行时动态生成一个新的 FastClass 子类，向子类中写入直接调用目标方法的逻辑。同时该子类会为代理类分配一个 int 类型的 index 索引，FastClass 即可通过 index 索引定位到需要调用的方法。生成 FastClass 子类是比较耗时的，可以使用缓存 FastClass 的方式进一步优化 RPC 框架的性能。

性能优化篇

RPC 框架的性能取决于很多因素，我们通常会关注几个方面：I/O 模型、网络参数、序列化方法、内存管理等。接下来我们主要以知识点的形式逐一介绍 RPC 框架中常用的优化方法。

I/O 模型

Netty 提供了高效的主从 Reactor 多线程模型，主 Reactor 线程负责新的网络连接 Channel 创建，然后把 Channel 注册到从 Reactor，由从 Reactor 线程负责处理后续的 I/O 操作。主从 Reactor 多线程模型很好地解决了高并发场景下单个 NIO 线程无法承载海量客户端连接建立以及 I/O 操作的性能瓶颈。

通常我们使用如下的方式配置主从 Reactor 线程模型：

```
EventLoopGroup bossGroup = new NioEventLoopGroup();

EventLoopGroup workerGroup = new NioEventLoopGroup();

ServerBootstrap b = new ServerBootstrap();

b.group(bossGroup, workerGroup)
```

如果你没有指定 workerGroup 线程组初始化的线程数，那么 Netty 会默认创建 2 倍 CPU 核数作的线程，但这并不一定是一个最佳数量，可以根据实际的压测情况进行适当调整。一般来说，只要服务性能能够满足要求，workerGroup 初始化的线程数应该越少越好，这样可以有效地减少线程上下文切换。

Netty 提供了一个参数 ioRatio，可以调整 I/O 事件处理和任务处理的时间比例，默认值为 50。对于高并发的 RPC 调用场景，ioRatio 可以适当调大，控制 Netty 有更多的时间比例在执行 I/O 任务。

Netty 网络参数配置

Netty 提供了 ChannelOption 以便于我们优化 TCP 参数配置，为了提高网络通信的吞吐量，一些可选的网络参数我们有必要掌握。

- TCP_NODELAY，是否开启 Nagle 算法。Nagle 算法通过缓存的方式将网络数据包累积到一定量才会发送，从而避免频繁发送小的数据包。Nagle 算法 在海量流量的场景下非常有效，但是会造成一定的数据延迟。如果对数据传输延迟敏感，那么应该禁用该参数。
- SO_BACKLOG，已完成三次握手的请求队列最大长度。同一时刻服务端可能会处理多个连接，在高并发海量连接的场景下，该参数应适当调大。但是 SO_BACKLOG 也不能太大，否则无法防止 SYN-Flood 攻击。
- SO_SNDBUF/SO_RCVBUF，TCP 发送缓冲区和接收缓冲区的大小。为了能够达到最大的网络吞吐量，SO_SNDBUF 不应当小于带宽和时延的乘积。SO_RCVBUF 一直会保存数据到应用进程读取为止，如果 SO_RCVBUF 满了，接收端会通知对端 TCP 协议中的窗口关闭，保证 SO_RCVBUF 不会溢出。
- SO_KEEPALIVE，连接保活。启用了 TCP SO_KEEPALIVE 属性，TCP 会主动探测连接状态，Linux 默认设置了 2 小时的心跳频率。TCP KEEPALIVE 机制主要用于回收死亡时间交长的连接，不适合实时性高的场景。

序列化方式

在网络通信过程中，必然涉及序列化和反序列化操作，即将对象编码成字节，再把字节解码

成对象的过程。序列化和反序列化属于高频且较笨重的操作，属于 RPC 框架中一个重要的性能优化点。在选择序列化方式时需要综合考虑各方面因素，如高性能、跨语言、可维护性、可扩展性等。

比较常用的序列化算法有 Kryo、Hessian、Protobuf 等，这些第三方序列化算法都比 Java 原生的序列化操作都更加高效。Kryo 序列化后占用字节数较少，网络传输效率更高，但是不支持跨语言。Hessian 是目前业界使用较为广泛的序列化协议，它的兼容性好，支持跨语言，API 方便使用，序列化后的字节数适中。Protobuf 是 gRPC 框架默认使用的序列化协议，属于 Google 出品的序列化框架。Protobuf 支持跨语言、跨平台，具有较好的扩展性，并且性能优于 Hessian。但是 Protobuf 使用时需要编写特定的 prpto 文件，然后进行静态编译成不同语言的程序后拷贝到项目工程中，一定程序增加了开发者的复杂度。综合各方面因素以及实际口碑，个人比较推荐使用 Hessian 和 Protobuf 序列化协议。

关于 RPC 框架序列化进一步的性能优化我们可以采用以下方法：

- 减少不必要的字段以及精简字段的长度，从而降低序列化后占用的字节数。
- 提供不同的序列化策略。可以将不同的字段拆分至不同的线程里进行反序列化，例如 Netty I/O 线程可以只负责 className 和 消息头 Header 的反序列化，然后根据 Header 分发到不同的业务线程池中，由业务线程负责反序列化消息内容 Content，这样可以有效地降低 I/O 线程的压力。

内存管理

Netty 会使用堆外内存 DirectBuffer 进行 Socket 读写，相比使用堆内存减少了一次内存拷贝。然而堆外内存的创建和销毁成本更高，所以通常会使用内存池来提高性能，你可以回顾下《轻量级对象回收站：Recycler 对象池技术解析》课程中所介绍的 Netty 池化技术。对于数据量较小的一些场景，可以考虑使用 HeapBuffer，由 JVM 负责内存的分配和回收可能效率更高。

此外，Netty 还提供了一些技巧来避免内存拷贝：

- CompositeByteBuf 是 Netty 中实现零拷贝机制非常重要的一个数据结构，它可以组合多个 Buffer 对象合并成一个逻辑上的对象，避免通过传统内存拷贝的方式将几个 Buffer 合并成一个大的 Buffer，我们经常使用 CompositeByteBuf 拼接协议数据的 头部信息 Header 和消息体数据 Body。
- 在失败重试的场景，我们想保留 ByteBuf 继续使用，你可以使用 copy() 方法拷贝原始 ByteBuf 的所有信息。但是深拷贝非常浪费性能的，你可以使用浅拷贝操作 oldBuffer.duplicate().retain() 复制出独立的读写索引，底层分配的内存、引用计数都是与原始 ByteBuf 共享的，其中 retain() 又会将 ByteBuffer 的引用计数加 1，从而避免了 ByteBuffer 被释放。

高可用篇

在整个 RPC 框架实践课中，我们并没有太多考虑 RPC 框架高可用相关的内容，但是高可用是分布式系统架构设计中一个重要的因素，下面我们便一起讨论如何提高 RPC 框架的可用性。

连接空闲检测+心跳检测

连接空闲检测是指每隔一段时间检测连接是否有数据读写，如果服务端一直能收到客户端连接发送过来的数据，说明连接处于活跃状态，对于假死的连接是收不到对端发送的数据的。如果一段时间内没收到客户端发送的数据，并不能说明连接一定处于假死状态，有可能客户端就是长时间没有数据需要发送，但是建立的连接还是健康状态，所以服务端还需要通过心跳检测的机制判断客户端是否存活。客户端可以定时向服务端发送一次心跳包，如果有 N 次没收到心跳数据，可以判断当前客户端已经下线或处于不健康状态。由此可见，连接空闲检测和心跳检测是应对连接假死的一种有效手段，通常空闲检测时间间隔要大于 2 个周期的心跳检测时间间隔，主要是为了排除网络抖动的造成心跳包未能成功收到。

Netty 中提供了开箱即用的 IdleStateHandler 实现连接空闲检测，如果我们想把一定时间间隔内没有读到数据的客户端连接进行关闭，可以采取如下的实现方式：

```
public class RpcIdleStateHandler extends IdleStateHandler {

    public RpcIdleStateHandler() {

        super(60, 0, 0, TimeUnit.SECONDS);

    }

    @Override

    protected void channelIdle(ChannelHandlerContext ctx, IdleStateEvent evt) {

        ctx.channel().close();

    }

}
```

IdleStateHandler 实现心跳检测本质是向任务队列中添加定时任务，判断 channelRead() 或 write() 方法是否发生空闲超时，IdleStateHandler 的构造函数支持设置读空闲时间、写空闲时间、读写空闲时间。super(60, 0, 0, TimeUnit.SECONDS) 表示我们只关注读空闲时间，如果服务端 60s 没未读到数据，就会回调 channelIdle() 方法，此时我们进行连接关闭，避免资源浪费。

心跳检测在 Netty 中并没有现成的实现，但是与空闲检测实现的原理是差不多的，客户端可以采用 EventLoop 提供的 schedule() 方法向任务队列中添加心跳数据上报的定时任务，如下所示：

```
public class RpcHeartBeatHandler extends ChannelInboundHandlerAdapter {

    @Override
    public void channelActive(ChannelHandlerContext ctx) throws Exception {
        super.channelActive(ctx);
        doHeartBeatTask(ctx);
    }
    private void doHeartBeatTask(ChannelHandlerContext ctx) {
        ctx.executor().schedule(() -> {
            if (ctx.channel().isActive()) {
                HeartBeatData heartBeatData = buildHeartBeatData();
                ctx.writeAndFlush(heartBeatData);
                doHeartBeatTask(ctx);
            }
        }, 10, TimeUnit.SECONDS);
    }
}
```

客户端向服务端定时发送心跳包，服务端收到后并不回复响应，因为如果同时与服务端建立的客户端连接规模较大，响应心跳数据需要消耗一定的资源。如果想要实现客户端和服务端互相感知存活状态，需要采用双向心跳机制。我们需要根据实际场景选择最合理的心跳检测方式。

线程池隔离

如果你的 RPC 服务是公司的基础服务，可能会有非常多的调用方，例如用户接口、订单接口等等。在我们实现的 RPC 框架中，业务线程池是共用的，所有的 RPC 请求都会有该线程池处理。如果有一天其中一个服务调用方的流量激增，导致线程池资源耗尽，那么其他服务调用方都会受到严重的影响。我们可以尝试将不同的服务调用方划分到不同等级的业务线程池中，通过分组的方式对服务调用方的流量进行隔离，从而避免其中一个调用方出现异常

状态导致其他所有调用方都不可用，提高服务整体性能和可用率。

流量隔离技术是服务治理中非常重要的一个措施，在很多大规模流量的业务系统中都有所应用，例如秒杀系统，可以根据特殊的请求头识别出是否是秒杀请求，从而跟日常请求的流量隔离开来。那么对于 RPC 框架而言，如何对服务调用方进行合理的分组呢？一般来说，根据应用的重要等级作为分组依据是一个很好的衡量标准，一定要保障核心业务不受影响，例如下单、支付等接口都需要有自己独立的业务线程池，避免受到其他服务调用方的影响。

重试机制

重试机制你再熟悉不过了，在平时的项目开发中你一定经常用到。为了保障服务的稳定性和容错性，重试机制是一般可以帮助我们解决不少问题，例如网络抖动、请求超时等场景都需要重试机制。

关于 RPC 框架的重试机制有几点最佳实践和注意事项，有必要与你分享一下：

- 被调用的服务接口的业务逻辑需要保证幂等才可以考虑使用重试机制，例如数据插入、更新操作，无论重复请求多少次都不会产生任何影响。
- 重试机制虽然可以提升服务可用性，但是重试可能会导致服务提供方流量倍增，极端情况下甚至造成雪崩。服务调用方最好设置合理的服务调用超时时间以及失败后的重试次数，需要综合考虑接口依赖服务的平均耗时、TP99 响应时间、服务重要等级等因素作为参考依据。为了防止重试引发的流量风暴，服务提供方必须考虑熔断、限流、降级等保护措施。
- RPC 框架的重试机制一般会采取指数退避的策略，两次重试之间指数级增加间隔时间，例如 1s、2s、4s、8s，以此类推，同时必须限制最大延迟时间。指数退避会存在负载峰值的问题，例如服务提供方可能发生 FullGC 导致同一时间产生超时重试的请求增多。为了解决负载峰值问题，可以在重试间隔中增加随机值，将请求分摊在不同的时间点中。
- 在负载均衡选择服务节点时，应该剔除上次重试失败的节点，进一步提高重试的成功率。

集群容错

集群容错是指服务消费者调用服务提供者集群时发生异常时的处理方案。以 Dubbo 框架为例，提供了六种内置的集群容错措施。

- **Failover，失效转移策略。**Failover 是 Dubbo 默认的集群容错措施，当出现调用失败时，会重新尝试调用其他服务节点。对于幂等性操作我们可以选择 Failover 策略，但是重试的副作用在上文中我们已经提到过，如果服务提供者出现问题可能会产生大量的重

试请求。

- **Failfast，快速失败策略。**Failfast 非常适合非幂等性操作，服务消费者只发起一次调用，如果出现失败的情况则立刻报错，不进行任何重试。Failfast 的缺点就是需要服务消费者自己控制重试逻辑。
- **Failsafe，失效安全策略。**Failsafe 策略在出现异常时，直接忽略。Failsafe 策略适合执行非核心的操作，如监控日志记录。
- **Failback，失效自动恢复策略。**服务消费者调用失败后，Dubbo 会记录此次失败请求到队列中，然后定时重新发送该请求。Failback 策略适用于实时性不高的场景，如消息推送。
- **Forking，并行措施。**服务调用者并行调用多个服务提供者节点，只要有一个调用成功就返回结果。通常用于实时性要求较高的操作，而且可以降低 TP999 指标，但是需要牺牲一定的服务器资源。
- **Broadcast，广播措施。**Broadcast 策略会广播所有的服务提供者，逐个调用，任意一台失败则等待广播最后完成之后抛出，通常用于更新服务提供方的本地资源状态。

以上几种集群容错措施可以根据实际的业务场景进行配置选择，而且 Dubbo 给我们提供了 Cluster 扩展接口，我们可以自己定制集群的容错模式。

此外，实现 RPC 框架高可用的措施还有很多，如限流保护、动态扩容、平滑重启、服务治理等等，由于篇幅有限，我在这里就不一一展开了。实现一个 RPC 框架原型并不是什么难事，但是如何保证 RPC 框架的高性能、高可用、易扩展，是需要我们不断去学习和积累的技能。

总结

要想精通一门技术，自然离不开源码学习以及长期的实践经验。为了便于学习，本专栏完整地实现了 RPC 框架的基础功能，更有趣的是 RPC 框架还有更多高阶特性等待我们去挖掘，如服务治理、线程池隔离、集群容错、熔断限流等。你是否已经迫不及待地想去进一步深入研究 RPC 框架更多的知识了呢？一起动手把实战项目打磨得更加完善，一步步提升自己架构设计和编码的基本功！

上一页 下一页