手写RPC框架

什么是RPC框架，有什么优点

RPC 是一种用于实现分布式系统中 跨网络进行通信 的技术，是一种计算机通信协议。RPC 框架基于 RPC 协议实现，允许一个程序(称为服务消费者)像调用自己程序的方法一样调用另一个程序(称为服务提供者)的接口，而不需要了解数据的传输处理过程、底层网络通信的细节等。这些都会由 RPC 框架帮你完成，使得开发者可以轻松调用远程服务，快速开发分布式系统。
此外，RPC 框架还有其他优点，比如:
1.提高性能:RPC 框架通常采用高效的网络通信协议和序列化/反序列化机制。
2.额外功能:如负载均衡、服务发现、容错机制等，能提高系统的可靠性、可用性和稳定性。
支持动态扩展:开发者可以动态扩展 RPC 的功能，比如自定义负载均衡器、自定义序列化3协议等。
在我的 RPC 框架项目中，就是利用自定义实现的 SPI机制实现了许多模块的扩展。

RPC和HTTP有什么区别，RPC是计算机网络的哪一层协议

RPC(远程过程调用)和 HTTP(超文本传输协议)是两种不同的通信协议，虽然都是应用层协议，但它们在设计目的、实现方式和使用场景等方面有着明显的区别。
1)设计目的:
·RPC:用于在分布式系统中实现远程方法调用，使得应用程序能够像调用本地方法一样调用远程服务。
·HTTP:用于在客户端和服务器之间传输超文本文档，用于在 Web 应用程序之间进行通信。
2)实现方式:
·RPC:通常基于特定的通信协议和编码机制，比如基于 TCP 和自定义协议头实现消息传
输，可以提供更高的性能。·HTTP:基于 TCP 协议，通过请求-响应模型进行通信，采用文本格式的消息头和消息体。直截了当地说:HTTP 只是 RPC 框架网络传输的一种可选方式罢了，3)使用场景:RPC 更多地运用于服务端之间的通信:HTTP 可同时运用于服务器端、、后端和前端的通信。

你了解哪些RPC框架,你的项目和他们相比有什么不同?

常见的 RPC框架包括 gRPC(Google)、Apache Thrift(Facebook)、Dubbo(阿里巴巴)、Feign(Netflix 基于 HTTP)等。在我平时的项目中，对 Dubbo 和 Feign 的使用较多
首先必须承认，我的 RPC 项目没办法和知名的、完善的 RPC 项目相比，但是我在设计实现自己的 RPC 项目时，借鉴了 Dubbo 的设计理念我的 RPC 框架有如下特点:
1)自定义 RPC 协议:我的 RPC 项目是基于 Vert.x 设计实现 TCP 服务器和客户端、并且自主设计了消息头的格式，相比 Feign 这种 HTTP 的传输性能更高。
2)可扩展性:我的 RPC项日大量运用了SPI机制，扩展性更强，使用者可以通过 SPI实现富的自定义接口，如自定义负载均衡器、自定义容错机制、自定义注册中心的实现、自定义序列化接口等，更加灵活通用。
3)自实现注册中心:我基于 Etcd 分布式云原生存储中间件实现了注册中心，完成服务的注册发现。Etcd 作为一个用 Go 语言编写的开源、分布式键值存储系统，具有高性能、强一致性的特点，k8s 中也有使用;而且可以通过jetcd 客户端轻松操作 Etcd，这些是我选择它而不是Redis 或 Zookeeper 的理由。
4)开箱即用:项日中实现了基于注解和 Spring Boot Starter 的项目启动机制，使得我的 RPC框架更简单易用。

请介绍整个系统的核心架构设计，有哪些模块每个模块的作用，各模块之间的关系

整个系统的核心架构设计和模块如下:
1.消费者代理模块:基于代理模式实现，在服务消费者对服务提供者进行远程调用时，不需要考虑是如何实

现调用的，直接调用写好的服务接口即可。
2.注册中心:基于 Etcd 实现注册中心，服务提供者会将自己的地址、接口、分组等详细信息都上报到注册中心模块，并且当服务上线、下线的时候通知到注册中心;服务调用方就可以从注册中心动态获取调用信息，而不用在代码内硬编码调用地址。
3.本地服务注册器:服务提供端存储服务名称和实现类的映射关系，便于后续根据服务名获取到对应的实现类，从而通过反射完成调用。
4.路由模块(负载均衡):当有多个服务和服务提供者节点时，服务消费者需要确认请求哪一个服务和节点。因此我设计了路由模块，通过实现不同算法的负载均衡器，帮助服务消费者选择一个服务节点发起调用。
5.自定义 RPC 协议:自主实现了基于 Vert.x(TCP)和自定义请求头的网络传输协议，并自主实现了对字节数组的编码/解码器，可以完成性能更高的请求和响应。
6.请求处理器:服务提供者在接收到请求后，可以通过请求处理器解析请求，从本地服务注册器中找到服务实现类并调用方法。
7.重试和容错模块:在调用出错时，进行重试、降级、故障转移等操作，保障服务的可用性和稳定性。
8.启动器模块:基于注解驱动 + Spring Boot Starter，开发者可以快速引入 RPC 框架到项目中。

下面我以一次完整的调用过程为例，讲解模块之间的关系。
服务提供者启动服务:启动基于 Vert.x的 TCP 服务器，并将服务信息注册到本地服务注册器和注册中心。
2.服务消费者调用服务:消费者代理模块从注册中心获取到服务提供者的调用信息列表，通过负载均衡算法确定一个调用地址，并构造基于 RPC 协议的请求。然后发起请求，消息经过编码器处理。
服务提供者处理请求:服务提供者通过解码器还原消息，根据消息内容，从本地服务注册器找到服务实现类，通过反射进行调用。
服务提供者响应请求:服务提供者将调用结果进行封装，通过编码器处理响应消息;服务4消费者收到消息后，通过解码器还原消息，从而完成调用。
服务消费端重试和容错:如果调用失败，服务消费端会根据预设好的策略进行重试、或者进行故障转移、降级等容错操作。

Java的反射机制，项目中如何应用反射机制

Java 反射机制让程序能在运行时访问某个类的信息或者操作某个类。比如在程序运行时检查类、获取类的属性和方法，以及动态创建对象、调用方法、修改属性的值等。使用反射可以增强代码的动态性和灵活性
我在项目中使用 Java 反射机制，实现了以下功能:
1.完成服务调用:服务提供端在从本地服务注册器中找到服务实现类后，通过反射创建实例，并通过 method.invoke 调用实例的指定方法。
2.实现动态代理:服务消费端使用了基于JDK 的动态代理，通过反射获取到调用的服务名称、方法、参数列表等，从而构造请求。
3.实现 SPI 机制:扫描自定义 SPI 的配置文件后，通过反射机制获取自定义 SPI 类的 Class类型和创建实例。

项目中如何实现消费方式调用的， 为什么选用JDK代理和工厂模式

RPC 框架的优势在于消费方可以像调用本地方法一样调用服务提供者的服务，为了实现这点我使用了动态代理技术。
先介绍整个调用流程:
1.消费方通过代理工厂，根据指定的服务类型获取代理对象
2.通过实现 InvocationHandler 接口实现 JDK动态代理，在 invoke 方法中根据 method等参数构造请求对象
3.通过注册中心进行服务发现，以获取提供该服务的所有服务提供者的信息
4.使用负载均衡选择一个服务提供者
5.发起 RPC 请求并得到响应结果

为什么使用 JDK 动态代理?
JDK 动态代理是 Java 提供的一种原生代理机制，允许开发者在运行时动态地创建代理类，这种机制主要用于接口的代理，符合我项目的需求。
选用它的原因:
1.JDK 动态代理作为 Java 标准库的一部分，无需引入任何第三方依赖即可使用2.相比于 CGLIB 等动态生成字节码的动态代理实现，JDK 动态代理的性能更高。3.JDK 动态代理能够在不同的 Java 平台上运行，具有良好的跨平台性。
为什么选用工厂模式?
1.更灵活:通过工厂模式提供的"根据类型获取代理类"的方法，我能够根据需要动态地选择合适的代理对象类型进行创建.
2. 解耦合:将创建代理对象的过程进行封装，调用方不需要了解具体的对象创建细节，使代码更利于维护。
此外，我还可以在工厂类中结合双检锁单例模式，实现代理对象的延迟初始化。

为什么使用concurrentHashMap实现本地服务注册器，其优势是什么?

首先，本地服务注册器需要存储的信息是服务唯一标识(比如名称)和对应实现类的映射，适用于使用 Map 结构来存储。
为什么使用 ConcurrentHashMap 而不是 Map 呢?主要是因为 ConcurrentHashMap 具有线程安全性和高性能。
具体优势如下:
1.线程安全性:ConcurrentHashMap 是线程安全的哈希表实现，可以在多线程环境下安全地进行操作，无需额外的同步措施。这意味着即使有多个线程同时访问注册器，也不会出现数据不一致或者并发访问异常的情况。
2.高效性能:ConcurrentHashMap 在并发环境下具有良好的性能表现。它采用了分段锁机制来实现并发访问，不同的线程可以同时访问不同的分段，从而减小了锁的粒度，提高了并发性能。
3.可伸缩性:ConcurrentHashMap 的设计考虑了高并发情况下的性能问题，并且能够根据实际的并发情况动态调整内部的数据结构，以保证高并发环境下的性能表现，可以适应不同规模的并发访问。
4.操作简便:ConcurrentHashMap 提供了丰富的操作方法，包括添加、删除、查找等常用操作，而且这些操作都是原子性的，无需额外的同步措施。这使得使用它实现本地服务注册器变得简单方便，同时也减少了出错的可能性。

如何实现项目中的网络通信，为什么选择了vert.x框架

在项目初期我是通过 Vert.x 的 HTTP 服务器的功能来实现网络通信，因为官方文档首先演示了这个 Demo，比较方便。但后面我了解到 RPC 框架需要更高的性能，所以我又基于 Vert.x 实现了 TCP 服务器，并且自主设计了协议消息结构，从而提高了 RPC 框架的网络通信效率。我的项目选用 Vert.x 实现 TCP 服务器，而不是 Netty 或直接用 Socket，主要有几个方面原因:

1.Vert.x 提供了高层次的抽象，相比 Netty 框架更简单易用。它不仅仅是一个网络框架，还是一个完整的异步编程框架，提供了包括 HTTP 服务器/客户端、WebSocket、事件总线、数据库访问、定时器等在内的多种构建高性能应用程序的工具和各种网络编程的API。
2.高性能。Vert.x基于事件循环机制，采用单线程或少量线程处理大量并发连接，可以有效减少线程上下文切换的开销，提高服务器的吞吐量和响应速度。更适合 RPC 框架这种需要处理大量 TCP 连接的应用场景。
3.对于 TCP 半包粘包的情况。Vert.x提供了 RecordParser 类，能更方便地解决，不用自己去设计算法来处理。

介绍Hutool工具库

Hutool是一个小而全的 Java 工具类库，它通过静态方法的形式，提供了Java 开发中常用的工具类，简化了 Java 开发中的很多复杂操作。我在项目中使用 Hutool:
1.我在读取 SP| 扩展配置文件时，使用 Hutool 工具库中的 ResourceUti.getResources 方法来获取指定目录下某个类名对应的所有资源文件的 URL 列表，然后通过将字节流转成字符流读取每个资源文件的内容进行SPI的加载。
2.在设计对请求的唯一标识时，使用 Hutool 工具类的 ldUtil.getSnowflakeNextld 方法来生成全局唯一的请求 ID 进行标识。
在实现注册中心时，我通过 Hutool提供的 ConcurrentHashSet 实现了一个线程安全的2Set 集合来存储正在监听的 Key。
4.我在设计注册中心的心跳机制时，通过 Hutool 提供的 CronUtil.schedule方法实现定时任务，定期发送心跳包实现节点续签。
5.解析 RPC 全局配置文件时，使用 Hutool的 Props 类简化了配置处理的复杂度，并且通过toBean 方法自动映射配置到对象属性，大大减少了手动解析配置文件和设置对象属性的复杂度。

项目中使用了哪些设计模式，举例说说如何应用的?

1)代理模式:
我采用 JDK 动态代理模式实现了消费者对服务提供者的"无感知"调用。消费者通过动态代理实现了对服务提供者的调用，当调用这些代理对象的方法时，实际上是通过网络向远程服务器发送请求，大大简化了远程服务的使用。
2)工厂模式:在设计 RPC 框架时，我通过工厂模式实现了序列化器工厂、注册中心工厂、服务代理工厂、负载均衡器工厂、重试策略工厂、容错策略工厂等。通过调用工厂类的方法来获取对象，而不是直接通过 new 操作符创建对象，这种方式降低了调用方和具体类之间的耦合度。
3)双检锁单例模式:
对于加载指定类型的实例、RPC 配置初始化等操作，我通过对实例进行两次非空的检查和一次对类的 Class 对象进行 synchronized 上锁，实现了双检锁单例模式。这样做的好处是在多线程环境下确保只有一个实例被创建，也可以避免在程序启动时就创建实例(按需加载)，从而节省了资源并提高了性能。
4)装饰者模式:
为了解决粘包半包的问题，我使用了 Vert.x 内置 RecordParser。但由于消息体的长度是不固定的，所以我要通过调整 RecordParser 的固定长度(变长)来解决，将读取完整的消息拆分为 2 次，先获取请求体的长度，再根据长度完整读取请求体。由于代码较为复杂，我运用了装饰者模式，新写一个 Wrapper 类来实现上述逻辑。使用时只需要用 Wrapper 封装原有 Vert.x的 TCP 处理器，就能解决粘包半包。不用修改原始处理器使得系统能够更灵活地扩展，代码更利于维护。

什么是Mock，项目中如何实现服务Mock的功能?

RPC 框架的核心功能是调用其他远程服务。但是在实际开发和测试过程中，有时可能无法直接访问真实的远程服务，或者访问真实的远程服务可能会产生不可控的影响，例如网络延迟、服务不稳定等。在这种情况下，就需要使用 mock 服务来模拟远程服务的行为，以便进行接口的测试、开发和调试。
mock 是指模拟对象，通常用于测试代码中，特别是在单元测试中，便于我们跑通业务流程。实现步骤:
1.首先，我给 RPC 框架全局配置增加了 boolean 类型的 mock 属性，用于控制是否开启Mock.
2.服务消费者通过代理工厂获取代理对象时，如果发现全局配置中的 mock 值为 true，那么将返回 Mock 的服务代理(而不是发送请求的服务代理)。
3.在 Mock 服务代理中，会根据方法的返回值类型，来生成特定的默认值对象。如 boolean类型返回 ture 或 false，int 则会返回0等。还可以通过 Faker 等类库来随机生成模拟值。

项目中有哪些配置信息

配置信息有很多，大致可以分为 4 类:
1)基本通用配置:RPC 项目名称、版本号、序列化器
服务消费方配置:模拟调用、负载均衡器、重试策略、容错策略2)3)服务提供方配置:服务器主机名和端口号
4)注册中心配置:注册中心类型、地址、用户名密码、超时时间等由于注册中心配置较多，我单独写了一个 RegistryConfig 封装配置信息。我是如何读取和管理配置信息的?(见项目“全局配置加载” 章节)
由于配置信息会在项目中多次使用，所以我维护了一个全局配置对象，便于框架快速获取到致的配置。
而且 RPC 框架是需要被其他项目作为服务提供者或者服务消费者引入的，应当允许引入框架的项目通过编写配置文件来 自定义配置。所以我设计了一套全局配置加载功能。能够让 RPC 框架轻松地从配置文件中读取配置。
1)如何管理配置信息:我实现了 RpcApplication 全局应用类，相当于 holder，存放了项目全局用到的配置对象，通过双检锁单例模式保证只会创建一个配置对象实例。
2)如何读取配置信息:写了一个单独的 ConfigUtils 工具类。通过 Hutool的 Setting 模块的props.toBean方法读取 application.properties 文件内容，并转换成 RpcConfig 对象。还通过在配置文件名后拼接环境的方式实现了多环境配置文件的读取。

什么是序列化和反序列化，项目中如何处理序列化和反序列化的

什么是序列化和反序列化?
序列化:将 Java 对象转为可传输的字节数组。.
反序列化:将字节数组转换为 Java 对象。.
序列化和反序列化通常用于在不同系统之间或在不同时间点之间传输和保存对象状态，比如数据持久化、远程调用等场景。
在 RPC 框架中，无论请求或响应，都会涉及参数的传输。而Java 对象是存活在 IM 虚拟机中的，如果想在其他位置存储并访问、或者在网络中进行传输，就需要进行序列化和反序列化。有很多种不同的序列化方式，比如Java 原生序列化、JSON、Hessian、Kryo、protobuf等在我的项目中，我设计了一套可配置、可扩展的序列化器实现机制:
1.首先定义了序列化器接口，提供了序列化、反序列化方法
2.我实现了 JDK 原生序列化、Kryo、Hessian、JSON 等多种序列化器
3.可配置化:我通过工厂模式 + 单例模式提供了根据配置文件动态获取序列化器实例的方法。
4.可扩展:通过白定义的 SPI机制，实现了序列化器的自主扩展、或者覆盖默认的实现。
比如，只需要一行代码就能获取到指定的序列化器实例:
Serializer serializer =SerializerFactory.getInstance("json");

你熟悉哪些序列化协议或类库，它们各有哪些优缺点

我比较熟悉的序列化有 JDK序列化、Kryo序列化、Hessian 序列化、JSON 序列化(比如fastjson、jackson、gson)等，我也在项目中实现了这些序列化器。JDK 序列化的优点是不需要我引入任何额外的依赖就能用。只要一个对象实现了 Serializable接口，就可以实现序列化。但它也有一些缺点，比如序列化后的数据体积比较大、性能不是特别高、可读性差。
下面再说说我接触到的其他序列化协议的优缺点:
1)JSON
优点:
易读性好，可读性强，便于人类理解和调试。.
·跨语言支持广泛，几乎所有编程语言都有 JSON 的解析和生成库。
缺点:
序列化后的数据量相对较大，因为 JSON 使用文本格式存储数据，需要额外的字符表示键、值和数据结

构。
不能很好地处理复杂的数据结构和循环引用，可能导致性能下降或者序列化失败。
2)Hessian
优点:
二进制序列化，序列化后的数据量较小，网络传输效率高。
·支持跨语言，适用于分布式系统中的服务调用。
缺点:
性能较 JSON 略低，因为需要将对象转换为二进制格式,
对象必须实现 Serializable 接口，限制了可序列化的对象范围。
3)Kryo
优点:
高性能，序列化和反序列化速度快。
·支持循环引用和自定义序列化器，适用于复杂的对象结构。
·无需实现 Serializable 接口，可以序列化任意对象。

缺点:
不跨语言，只适用于 Java。
对象的序列化格式不够友好，不易读懂和调试。
4)Protobuf:
优点:
·高效的二进制序列化，序列化后的数据量极小。
跨语言支持，并且提供了多种语言的实现库。
·支持版本化和向前/向后兼容性。
缺点:
配置相对复杂，需要先定义数据结构的消息格式。
对象的序列化格式不易读懂，不便于调试。

什么是Java的SPI机制，如何利用SPI机制实现模块动态扩展的

SPI(Service Provider Interface)服务提供接口是 Java 的重要机制，主要用于实现模块化开发和插件化扩展。SPI机制允许服务提供者通过特定的配置文件将自己的实现注册到系统中，然后系统通过反射机制动态加载这些实现，而不需要修改原始框架的代码，从而实现了系统的解耦、提高了可扩展性。
-个典型的 SPI应用场景是JDBC(Java 数据库连接库)，不同的数据库驱动程序开发者可以使用 JDBC 库，然后定制自己的数据库驱动程序。此外，我们使用的主流 Java 开发框架中，几乎都使用到了 SPI机制，比如 Servlet 容器、日志框架、ORM 框架、Spring 框架。
虽然 Java 内置了 ServiceLoader 来实现 SPI，但是如果想定制多个不同的接口实现类，就没办法在框架中指定使用哪一个了，也就无法实现像"通过配置快速指定序列化器" 这样的需求。所以我自己定义了 SPI机制的实现，能够给每个自行扩展的类指定键名。

具体实现方
1)指定 SPI 的配置目录，并且将配置再分为系统内置 SPI和用户自定义 SPI，便于区分优先级和维护。
2)编写 SpiLoader 加载器,，实现读取配置、加载实现类的方法
1.用 Map 来存储已加载的配置信息 键名 =>实现类。
2.通过 Hutool 工具库提供的 ResourceUtil.getResources 扫描指定路径,，读取每个配置文件，获取到 键名 =>实现类 信息并存储在 Map 中。
3.定义获取实例方法，根据用户传入的接口和键名，从 Map 中找到对应的实现类，然后通过反射获取到实现类对象。可以维护一个对象实例缓存，创建过一次的对象从缓存中读取即可

3)重构序列化器工厂，改为从 SPI加载指定的序列化器对象。使用静态代码块调用 SPI 的加载方法，在工厂首次加载时，就会调用 SpilLoader 的 load 方法加载序列化器接口的所有实现类，之后就可以通过调用 getInstance 方法获取指定的实现类对象了

项目中的服务注册中心有什么作用，是基于什么技术实现的?

服务注册中心是 RPC 框架中不可或缺的重要模块:
1)服务提供者在启动时，会把自己能提供的服务信息(比如服务名称、地址、端口等信息)提交到注册中心。
2)服务消费者在调用服务时，会通过服务注册中心查询所需服务的地址和其他信息，从而完成调用。
3)服务注册中心还会定期检查服务提供者的健康状态，如果发现某个服务提供者不可用，它会自动将其从服务列表中剔除。这保证了服务消费者总是调用到正常的服务实例。常用的注册中心实现技术有 ZooKeeper、Redis、Eureka、Consul等。在我的项目中，基于高性能、强一致性、简单易用的 Etcd 实现了注册中心。

为什么使用ETCd实现注册中心，该技术有哪些优势和特性?

Etcd 是一个 Go 语言实现的、开源的、分布式 的键值存储系统，它主要用于分布式系统中的服务发现、配置管理和分布式锁等场景。
提到 Go 语言实现，有经验的同学应该就能想到，Etcd 的性能是很高的，而且它和云原生有着密切的关系，通常被作为云原生应用的基础设施，存储一些元信息。比如经典的容器管理平台k8s 就使用了 Etcd 来存储集群配置信息、状态信息、节点信息等,除了性能之外，Etcd 采用 Raft 一致性算法来保证数据的一致性和可靠性，具有高可用性、强一致性、分布式特性等特点。
而且 Etcd 还非常简单易用!提供了简单的 AP、数据的过期机制、数据的监听和通知机制等完美满足注册中心的实现诉求。
Etcd 在其数据模型和组织结构上更接近于 ZooKeeper，它使用层次化的键值对来存储数据支持类似于文件系统路径的层次结构，能够很灵活地单 key 查询、按前缀查询、按范围查询由于我之前学习过 ZooKeeper，所以 Etcd 的入门成本对我来说并不高,除了上面提到的优势和特性外，Etcd 应用较多的特性还有:
1.Lease(租约):用于对键值对进行 TTL 超时设置，即设置键值对的过期时间。当租约过期时，相关的键值对将被自动删除。
2.Watch(监听):可以监视特定键的变化，当键的值发生变化时，会触发相应的通知。

服务提供者节点主动下线或者宕机时，如何保证注册中心上服务信息的有效性?

当服务提供者节点主动下线或宕机时，应该从注册中心移除掉已注册的节点，否则会影响消费端调用。所以我设计了一套服务节点下线机制。服务节点下线又分为:
主动下线:服务提供者项目正常退出时，利用JVM 的 ShutdownHook，主动从注册中心移除注册信息。
被动下线:服务提供者项目异常退出时，可以利用 Etcd 的 key 过期机制自动移除。
为了防止存活的服务提供者节点不会因为 key 过期被移除，我设计了心跳检测和续期机制:每个服务提供者节点维护自己注册的 key，并且通过定时任务每隔一段时间发送一次心跳包即重置该 key 在 Etcd 的过期时间。如果服务提供者节点已经宕机，则不会触发续期，过期后会自动被 Etcd 移除。

服务消费者每次都要从注册中心获取服务信息吗？有没有什么办法进行优化?

不需要。由于正常情况下，服务节点信息列表的更新频率是不高的，所以在服务消费者从注册中心获取到服务节点信息列表后，完全可以 缓存在本地，下次就不用再请求注册中心获取了能够提高性能。
因此，我设计了消费端服务缓存模块。服务消费者在本地维护一个服务缓存列表，每次向注册中心拉取服

务信息时会优先判断本地缓存列表是否已经存在，如果存在则优先获取缓存列表里面的服务信息，不存在则向注册中心拉取后存入本地缓存里面。

如何更新服务注册信息缓存，怎么保证缓存数据的一致性?

我利用 Etcd 的 watch 监听机制实现了缓存数据的更新和一致性。当第一次获取到某个服务的注册信息后，会对该服务下的所有 key 使用 Jetcd 的 watch 方法进行监听。当监听到 key 被移除的 DELETE 事件时，就表示对应的服务提供者节点已经下线，此时清空本地服务注册信息缓存即可。
额外补充:
1.还可以通过 Caffeine 等技术给服务注册信息缓存增加过期时间，定期刷新缓存。
2.在新的服务节点注册时，也可以通过某种通知机制，更新对应服务的缓存。

RPC框架采用了什么协议，为什么要自定义协议?

为了提高请求性能、减少数据传输，我自主设计了一套 RPC 协议，包括自定义消息结构和网络传输方式。我在设计自定义协议时，借鉴了 Dubbo 的协议。
1)自定义消息结构:设计的核心理念是"用最少的空间传递需要的信息"，所以我使用字节数组来存储拼接消息。消息内容分为消息头 +消息体，消息头里面的内容包括了魔数、版本号、序列化器、消息类型、状态、请求 id、消息体长度信息。
2)网络传输方式:我选择使用 TCP 协议完成网络传输，相比于 HTTP 协议的性能更高。因为 HTTP 头信息是比较大的，会影响传输性能。除了这点外，HTTP 本身属于无状态协议,这意味着每个 HTTP 请求都是独立的，每次请求/响应都要重新建立和关闭连接，也会影响性能。
为什么要自定义协议?理由如下:
1.更高的性能。自定义协议允许我根据应用的特定需求来设计数据格式和通信机制，从而减少不必要的数据传输，降低网络延迟，提高数据处理速度
2.灵活性和扩展性。通过自定义协议，我设计了包含版本信息的协议头，可以支持向后兼容和协议升级。
3.安全性。比如在消息头中引入了魔数进行安全校验。

项目中如何解决TCP的半包粘包问题

使用 TCP 协议网络通讯时，可能会出现半包和粘包问题。当客户端向服务端连续发送多条消息时，半包是指服务端单次收到的数据比客户端单次发送的数据少，粘包是指服务端收到的单次收到的数据比客户端单次发送的数据多。

如何在项目中解决半包、粘包问题?
在消息头中设置请求体的长度，服务端接收时，判断每次消息的长度是否符合预期，不完整就不读，留到下一次接收到消息时再读取。我在代码中使用了 Vert.x 框架中内置的 RecordParser 完美解决半包粘包，它的作用是:保证下次读取到特定长度的字符。因为消息体的长度是不固定的，所以我设计 RPC 时要通过调整 RecordParser的固定长度(变长)来解决。那我的思路是，将读取完整的消息拆分为2次:
1)先完整读取请求头信息，由于请求头信息长度是固定的，可以使用 RecordParser保证每次都完整读取。
2)再根据请求头长度信息更改 RecordParser的固定长度，保证完整获取到请求体。

为什么要使用负载均衡，有哪些负载均衡算法，项目中的负载均衡器模块如何实现?

使用负载均衡的好处:
1)我通过实现负载均衡将请求分发到多个服务器，即使我们的部分服务器出现故障，系统仍然可以继续提供服务，从而提高了系统的整体可用性和可靠性。
2)负载均衡能够合理分配客户端请求或网络流量到多个服务器，避免单个服务器因负载过重而成为瓶颈，从

而提升整体系统性能。
我熟悉的几种负载均衡算法:
1)轮询(Round Robin):按照循环的顺序将请求分配给每个服务器，适用于各服务器性能相近的情况。
2)随机(Random):随机选择一个服务器来处理请求，适用于服务器性能相近且负载均匀的情况。
3)加权轮询(Weighted Round Robin):根据服务器的性能或权重分配请求，性能更好的服务器会获得更多的请求，适用于服务器性能不均的情况。
4)加权随机 (Weighted Random):根据服务器的权重随机选择一个服务器处理请求，适用于服务器性能不均的情况。
5)最小连接数(Least Connections):选择当前连接数最少的服务器来处理请求，适用于长连接场景。
6)IP Hash:根据客户端 IP 地址的哈希值选择服务器处理请求，确保同一客户端的请求始终被分配到同一台服务器上，适用于需要保持会话一致性的场景还可以使用一致性 Hash 算法，解决了节点下线和倾斜问题。

我在项目中实现了可扩展的负载均衡器，支持多种负载均衡算法，比如轮询、随机、一致性Hash。
1)可扩展性设计:
1.定义通用的负载均衡器接口。
2.使用工厂模式，可以动态根据用户的配置创建不同的负载均衡器实例。
3.使用 SPI 机制，允许用户扩展自己的负载均衡器。
2)负载均衡器算法实现:
1.轮询:通过对自增轮询下表对服务器总数取模来实现轮询。
2.随机:通过指定提供者服务数量范围，使用 Random 取随机数。
3.-致性 Hash 负载均衡:使用 TreeMap 实现一致性 Hash 环，该数据结构提供了ceilingEntry 和 firstEntry 两个方法，便于获取符合算法要求的节点。

什么是一致性Hash算法，相比普通的轮询算法有什么优势?

致性哈希(Consistent Hashing)是一种经典的哈希算法，用于将请求分配到多个节点或服务器上，所以非常适用于负载均衡。
它的核心思想是将整个哈希值空间划分成一个环状结构，每个节点或服务器在环上占据一个位置，每个请求根据其哈希值映射到环上的一个点，然后顺时针寻找第一个大于或等于该哈希值的节点，将请求路由到该节点上。

与普通的轮询算法相比，一致性哈希还解决了 节点下线 和 倾斜问题。
1)节点下线:当某个节点下线时，其负载会被平均分摊到其他节点上，而不会影响到整个系统的稳定性，因为只有部分请求会受到影响。
2)倾斜问题:通过虚拟节点的引入，将每个物理节点映射到多个虚拟节点上，使得节点在哈希环上的 分布更加均匀，减少了节点间的负载差异。

为什么要使用重试机制，有哪些重试策略，项目中的重试机制如何实现?

如果没有重试机制，使用 RPC 框架的服务消费者调用接口失败，就会直接报错。调用接口失败可能有很多原因，有时可能是服务提供者返回了错误，但有时可能只是网络不稳定或服务提供者重启等临时性问题。这种情况下，我们可能更希望服务消费者拥有自动重试的能力，提高系统的可用性。
我了解到的几种重试策略:
1.固定重试间隔(Fixed Retry Interval):在每次重试之间使用固定的时间间隔。2.指数退避重试(Exponential Backoff Retry):在每次失败后，重试的时间间隔会以指数级增加，以避免请求过于密集。
3.随机延迟重试(Random Delay Retry):在每次重试之间使用随机的时间间隔，以避免请求的同时发生。

4.可变延迟重试(Variable Delay Retry):这种策略更“高级”了，根据先前重试的成功或失败情况，动态调整下一次重试的延迟时间。比如，根据前一次的响应时间调整下一次重试的等待时间。
值得一提的是，以上的策略是可以组合使用的，一定要根据具体情况和需求灵活调整。比如可以先使用指数退避重试策略，如果连续多次重试失败，则切换到固定重试间隔策略。
在项目中，我实现了可扩展的重试机制:
1)可扩展性设计:
1.定义通用的重试策略接口。
2.使用工厂模式，可以动态根据用户的配置创建不同的重试策略实例。
3.使用 SPI机制，允许用户扩展自己的重试策略。
2)重试策略算法实现:我开发了“固定时间间隔”和”不重试” 这两种策略。实现固定重试间隔的方法:
。通过使用 Guava-retrying 库实现了具有固定时间间隔的重试机制。
设置了固定的等待策略，即在每次重试之间等待固定的时间间隔，设置的等待时间为3.艉秒。
设置了停止策略，即在尝试了3次之后停止重试。这意味着最多会执行1次原始调用和 2.
次重试。
指定重试条件，碰见指定类型的 Exception 才进行重试。.
添加了一个重试监听器，这个监听器可以在每次重试发生时执行一些自定义的逻辑，比如这里记录了当前重试的次数。

为什么要使用容错机制，有哪些容错策略，项目中的容错机制如何实现?

容错是指系统在出现异常情况时，可以通过一定的策略保证系统仍然稳定运行，从而提高系统的可靠性和健壮性。
在分布式系统中，容错机制尤为重要，因为分布式系统中的各个组件都可能存在网络故障、节点故障等各种异常情况。要顾全大局，尽可能消除偶发/单点故障对系统带来的整体影响。打个比方，将分布式系统类比为一家公司，如果公司某个优秀员工请假了，需要“触发容错”，让另一个普通员工顶上，这本质上是容错机制的一种 降级 策略,容错策略有很多种，常用的容错策略主要是以下几个:
1.Fail-Over 故障转移:一次调用失败后，切换一个其他节点再次进行调用，也算是一种重试。
2.Fail-Back 失败自动恢复:系统的某个功能出现调用失败或错误时，通过其他的方法，恢复该功能的正常。可以理解为降级，比如重试、调用其他服务等!
3.Fail-Safe 静默处理:系统出现部分非重要功能的异常时，直接忽略掉，不做任何处理，就像错误没有发生过一样。
4.Fail-Fast 快速失败:系统出现调用错误时，立刻报错，交给外层调用方处理
在项目中，我实现了可扩展的容错机制:
1)可扩展性设计:
1.定义通用的容错策略接口。
2.使用工厂模式，可以动态根据用户的配置创建不同的容错策略实例。
3.使用 SPI机制，允许用户扩展自己的容错策略.
2)多种容错策略实现:
1.Fail-Over 故障转移:当调用服务提供者异常时，先在本地缓存中移除该服务，然后通过注册中心获取到剩下的服务节点，通过负载均衡从剩下节点中选择一个节点再次进行调用。2.Fail-Back 失败自动恢复:通过 SPI 机制使用用户的本地 Mock 服务，当服务发生失败将自动调用 Mock 服务获取兜底数据。
3.Fail-Safe 静默处理:不进行异常抛出，重新 new 一个响应对象进行返回.
4.Fail-Fast 快速失败:遇到异常后，将异常再次抛出，交给外层处理。

如何简化开发者使用RPC框架的成本，怎么通过注解驱动框架的启动?

根据我之前使用其他框架的经验，一定要尽可能地让开发者写更少的代码，就能使用框架。所以我实现了两种方案:
1.我把所有启动代码封装成一个专门的启动类或方法，然后由服务提供者/服务消费者调用即可。
2.在 Spring 项目中，支持 Bean 的注解注入，我觉得可以利用这点进一步简化开发者的使用。于是我实现了注解驱动，并且将 RPC框架封装成 Spring Boot Starter，供使用者直接引入。
如何通过注解驱动框架启动?
1)先定义注解参考 Dubbo 注解功能，遵循最小可用化原则定义了三个注解
@EnableRpc:用于全局标识项目需要引入 RPC框架、执行初始化方法。@RpcService:服务提供者注解，在需要注册和提供的服务类上使用。
@RpcReference:服务消费者注解，在需要注入服务代理对象的属性上使用，类似Spring 中的 @Resource 注解。
2)在框架初始化时，获取 @EnableRpc 注解属性，通过实现 Spring 的ImportBeanDefinitionRegistrar 接口，并且在 registerBeanDefinitions 方法中，获取到项目的注解和注解属性，选择是否启动服务器(为了区别服务提供者和消费者)。
3)编写服务提供者启动类 RpcProviderBootstrap获取到所有包含 @RpcService 注解的类并且通过注解的属性和反射机制，获取到要注册的服务信息，并且完成服务注册。
4)编写服务提供者启动类 RpcConsumerBootstrap 在 Bean 初始化后，通过反射获取到Bean 的所有属性，如果属性包含 @RpcReference 注解，那么就为该属性动态生成代理对象并赋值。
5)注册已编写的启动类。可以通过给 @EnableRpc 增加 @Import 注解，来注册我们自定义的启动类，实现灵活的可选加载。

开发过程中，遇到的较为复杂的技术问题或挑战，怎么解决的

1)在设计 RPC 模块、 以及制定 RPC 协议时，我进行充分的调研和思考。我先了解市面上的多个 RPC 类型如 GRPC、Dubbo、Apache Thrift 等，最终选择了我相对更熟悉的 Dubbo 作为我 RPC 项目的参考，我以前写过的一个 API开放平台正好使用过 Dubbo 作为 RPC 的调用。确定好参考目标后，我先参考了他的架构模块，后面参考了其协议的制定、注解功能等方面，逐步将 RPC 框架的各个功能拨茧抽丝，先整体后局部地完成了整个框架
2)在选型方面如 Etcd、Vert.x的使用，我先是参考了市面上的主流 RPC 后面去 Github 上查看别人写的 RPC 样例，结合着 GPT 以及各种文章，我觉得选用 Etcd 作为注册中心 Vert.x作为通讯服务器，因为经过我的多方考量 Etcd 跟 Vert.x 使用 API功能丰富且强大、易于上手能够把精力更多的放在具体的架构业务上面。