RPC项目复习笔记

RPC基本概念

1. **RPC 框架基本概念是什么?**

   `RPC` 允许程序在不同的进程或机器之间进行通信，互相调用函数或方法，就像调用本地方法一样。

   `RPC` 框架的主要角色为：服务提供者、服务消费者与注册中心

   - 服务消费者是调用远程服务的应用程序
   - 服务提供者是提供远程服务的应用程序
   - 注册中心是负责管理服务地址和方法信息的中心化服务，它记录了所有可用的远程服务信息，并为客户端提供服务的地址发现功能。

   具体又可以分为以下几个核心组件：

   - `Client`
   - `Server`
   - `Client Stub`：将客户端请求的参数、服务名称、服务地址进行打包，统一发送给 `server` 方
   - `Server Stub`：服务端接收到 `Client` 发送的数据之后进行消息解包，调用本地方法

   RPC 简单调用流程为：

   1. `Client` 调用 `Client Stub`，传入方法信息
   2. `Client Stub` 对方法进行包装，并序列化，通过网络传输发送给 `Server Stub`
   3. `Server Stub` 进行反序列化，获取方法参数等信息，本地调用Server中提供的方法
   4. `Server` 执行方法后，将方法返回结果返回给 `Server Stub`
   5. `Server Stub` 将方法返回结果进行序列化，通过网络传输发送给 `Client Stub`
   6. `Client Stub` 进行反序列化，并传递给 `Client`，得到最终结果

2. **RPC 框架的代理层作用?**

   代理模式主要是为了屏蔽调用远程方法时，底层的建立网络连接、序列化、发送数据、获取返回结果等细节操作，使得远程调用看起来像是本地方法调用一样

   代理模式的主要优点有：

   - 代理模式在客户端与目标对象之间起到一个中介作用和保护目标对象的作用；
   - 代理对象可以扩展目标对象的功能；
   - 代理模式能将客户端与目标对象分离，在一定程度上降低了系统的耦合度，增加了程序的可扩展性。

3. **RPC 框架的路由层作用?**

   路由层解决的问题为：**当目标服务众多的时候，客户端需要如何确定最终请求的服务提供者**

   通过路由层，去匹配对应的 `Provider` 服务提供者

4. **RPC 框架的协议层作用?**

   - 使用 RPC 框架进行远程调用的时候，需要对数据信息进行统一的包装和组织，最终才能将其发送到目标机器并且被目标机器接收解析

因此对于数据的各种序列化、反序列化，协议的组装我们统一可以封装在协议层中进行实现

- 可以帮助开发者屏蔽网络传输和数据编解码等底层细节，使得开发者只需要关注业务逻辑的实现

5. **RPC 框架的注册中心层作用?**
   - 客户端在进行远程调用时，需要获取到远程服务的各种信息，这些信息一般存储在注册中心中
   - 服务发现：客户端需要订阅注册中心。在需要远程调用时，从注册中心中获取信息，然后进行方法调用
   - 服务注册：服务提供者将地址、接口、分组等信息存放在注册中心模块，当服务上线、下线均会通知注册中心
   - 服务管理：提供服务的上下线管理、服务配置管理、服务健康检查等功能，以保证服务的可靠性和稳定性
6. **RPC 框架的容错层作用?**
   - 超时控制：通过设置超时时间来避免因服务端响应慢或网络等原因导致客户端等待时间过长的问题
   - 重试机制：当 `RPC` 调用失败时，可以通过自动重试的方式进行容错处理，增加调用的成功率
   - 异常处理：对于 `RPC` 调用过程中出现的异常，进行捕捉处理

RPC实现方案

1. **RPC 框架代理层是如何设计的?**

   整体思路为：利用JDK动态代理，将调用对象的构造、网络传输、等待请求响应等细节放入 `JDK` 代理的增强逻辑中，客户端在调用时，底层实现逻辑被屏蔽。

   1. 设计 **RpcInvocation** 对象
      - args：调用方法参数
      - targetMethod：目标方法
      - targetService：目标服务
      - UUID：请求id，用于区分每次请求
   2. `JDK` 动态代理 `invoke` 方法执行逻辑
      1. 动态地获取请求方法与参数，创建 `RpcInvocation` 对象，并设置 `args`、`targetMethod`、`targetService`、`UUID` 等参数
      2. 将 `RpcInvocation` 对象放入到发送队列中，将 `UUID` 和 `Proxy` 组成的 `key-value` 放入响应 `map` 中，等待从响应 `map` 中获取响应
   3. 发送队列执行逻辑：为 `Client` 新创建的一个线程
      1. 阻塞式地从队列中获取 `JDK` 动态代理加入的 `RpcInvocation` 对象
      2. 将 `RpcInvocation` 对象序列化为 `JSON` 格式并转为 `byte` 数组，并根据协议封装为 `RpcProtocol` 对象，进行 `encode` 之后通过 `Netty` 发送到服务端
   4. 服务端执行逻辑
      1. 通过 `decode` 解码器获取到 `RpcProtocl` 对象，获取到 `byte` 数组并转为 `JSON` 字符串
      2. 将 `JSON` 字符串反序列化为`RpcInvocation` 对象
      3. 通过 `RpcInvocation` 中的服务名称，从 `map` 中找到目标服务，调用方法，得到响应

4. 将响应结果放入 `RpcInvocation` 对象中，将其序列化为 `JSON` 格式，并 `encode` 为 `RpcProtocl` 对象，通过 `Netty` 返回给客户端

5. 客户端执行逻辑
    1. 通过 `decode` 解码器获取到 `RpcProtoc` l对象，获取到byte数组并转为JSON字符串
    2. 将 `JSON` 字符串反序列化为 `RpcInvocation` 对象
    3. 获取到 `UUID` ，判断响应 `map` 中是否存在该 `UUID` ，不存在则抛出异常
    4. 将 `UUID` 和 `RpcInvocation组` 成的 `key-value` 放入响应 `map` 中，使得等待的 `JDK` 动态代理能够获取到响应并返回给客户端

2. **RPC 注册中心层是如何设计的?**

   采用 `Zookeeper` 作为注册中心

    1. `Zookeeper` 节点设计
        - 节点路径：
            - 针对 `Provider` ： `/ltyzzz/{serviceName}/provider/{host}:{port}`
            - 针对 `Consumer` ： `/ltyzzz/{serviceName}/consumer/{applicationName}:{host}`
        - 节点内容：
            - 针对 `Provider` ： `{applicationName};{serviceName};{host}:{port};{timeBytes}`
            - 针对 `Consumer` ： `{applicationName};{serviceName};{host};{timeBytes}`
    2. 客户端每次订阅服务端时，在 `Zookeeper` 创建一个节点，并将服务添加到自己的本地缓存中
    3. 订阅之后，客户端与每个订阅的服务端通过 `Netty` 建立连接时，同时需要去监听服务端节点的变化，以便于随时更新订阅的服务列表
    4. 服务端启动时，会在 `Zookeeper` 创建一个节点，相当于将服务信息注册到 `Zookeeper` 上
    5. 统一将节点的更新后的相关操作通过事件监听的机制来实现代码解偶

3. **RPC 引入注册中心之后的流程是什么样的?**
    1. `Server` 向 `Zookeeper` 注册服务，具体注册方式为依据定义的规则在 `Zookeeper` 中创建 `Provider` 节点
    2. `Server` 将注册的服务以自定义 `URL` 形式添加到本地缓存 `set` 中，包含服务应用名、`service` 名、服务地址
    3. `Client` 订阅指定服务，并将订阅的服务添加到本地缓存 `list` 中
    4. 底层通过 `Netty` 创建 `ChannelFuture` ，建立与远程 `service` 的连接，并添加到 `connect_map` 中
    5. 当 `Client` 通过 `JDK` 动态代理调用时，还是遵循之前的规则：发送到请求队列，等待响应
    6. 异步任务需要从请求队列中取出对象，然后根据服务名字，随机地从 `connect_map` 中取出连接，并发送请求

4. **RPC 路由层是如何设计的?**

   同一个服务可能对应着多个服务提供者，因此当客户端请求服务时，需要通过负载均衡策略从中选择一个合适的服务提供者。

   常见的负载均衡策略有：随机选取、带权重的随机选取、轮询、Hash 算法

5. **RPC 序列化层是如何设计的?**

   引入多种序列化策略，由用户自行配置与选择对应的策略

   - FastJson
   - Hessian
   - Kryo
   - JDK自带的序列化

   具体为创建序列化工厂接口，定义接口方法：serialize与deserialize（均为范型方法）

   采用具体的序列化策略去实现该工厂类。

   - SerializeFactory
     - FastJsonSerializeFactory
     - HessianSerializeFactory
     - KryoSerializeFactory
     - JdkSerializeFactory
6. **介绍一下你用到的序列化技术?**
   - JDK 序列化
     - 通过 `ObjectOutputStream` 的 `writeObject` 和 `readObject` 方法实现，可通过重写指定其他序列化方式
     - `JDK` 默认序列化方式性能差，且只适用于Java
   - JSON、XML
     - 用于 `Web` 服务、分布式系统、日志记录等场景
     - 可读性较好，但是性能较差，由于其描述信息过多，消息很大
   - Protocol Buffer
     - 支持跨语言、跨平台，可扩展性强
     - 需要使用 `IDL` 来定义 `Schema` 描述文件，定义完描述文件后，可以直接使用 `protoc` 来直接生成序列化与反序列化代码
     - 性能低于 `Kyro`，但是高于大部分序列化协议，序列化后的 `size` 也较小
   - Kyro
     - 主要适用于 `Java`，不支持字段扩展
     - 使用简洁，直接使用 `Input`、`Output`对象
     - 高性能，序列化与反序列化时间开销都很低，序列化后的 `size` 也很小
   - Hessian
     - 支持跨语言、跨平台，可扩展性强
     - 易用：只需要实现 `Serializable` 接口即可
     - 序列化时间与大小都比较小
7. **你的责任链模式是如何设计的？作用是什么？还了解别的设计模式吗?**

   责任链模式的作用

1. 对客户端请求进行鉴权

   客户端请求的远程接口可能需要进行权限校验（比如与用户隐私相关的数据），服务端必须确认该请求合法才可放行

2. 分组管理服务

   同一个服务可能存在多个分支，有的分支为 `dev` 代表正在处于开发阶段，有的分支为 `test` 代表正在处于测试阶段。

   为了避免客户端调用到正在开发中的服务，在进行远程调用时，需要根据 `group` 进行过滤。

3. 基于 `ip` 直连方式访问服务端

   可能存在两个名字相同但代码逻辑不同的服务。为了避免出现不同的结果，需要根据服务提供方的 `ip` 进行过滤

4. 调用过程中记录日志信息

传统模式中，客户端需要在发送请求之前，逐个的调用过滤请求的方法；服务端在接受请求之前，也需要逐个调用过滤请求的方法

这种模式下，代码耦合度高，且扩展性差。

而采用责任链模式可以带来：

- 发送者与接收方的处理对象类之间解耦。
- 封装每个处理对象，处理类的最小封装原则。
- 可以任意添加处理对象，调整处理对象之间的顺序，提高了维护性和可拓展性，可以根据需求新增处理类，满足开闭原则。
- 增强了对象职责指派的灵活性，当流程发生变化的时候，可以动态地改变链内的调动次序可动态的新增或者删除。
- 责任链简化了对象之间的连接。每个对象只需保持一个指向其后继者的引用，不需保持其他所有处理者的引用，这避免了使用众多的 if 或者 if…else 语句。
- 责任分担。每个类只需要处理自己该处理的工作，不该处理的传递给下一个对象完成，明确各类的责任范围，符合类的单一职责原则。

责任链模式设计

1. 首先创建 `IFilter` 接口，然后分别创建服务器与客户端对应的接口，继承 `IFilter` 接口
2. 分别创建服务器与客户端过滤链，用于存放过滤器实现类，并遍历过滤器实现类集合，执行过滤方法
3. 依次实现过滤器实现类

8. **可插拔组件是什么？在项目中是如何设计的?**

   SPI

   使用 `Java SPI` 机制的优势是实现解耦，使得第三方服务模块的装配控制的逻辑与调用者的业务代码分离，而不是耦合在一起。应用程序可以根据实际业务情况启用框架扩展或替换框架组件。

   相比使用提供接口 `jar` 包，供第三方服务模块实现接口的方式，`SPI` 的方式使得源框架，不必关心接口的实现类的路径，可以不用通过下面的方式获取接口实现类：

   - 代码硬编码 `import` 导入实现类

- 指定类全路径反射获取：例如在JDBC4.0之前，JDBC中获取数据库驱动类需要通过**Class.forName("com.mysql.jdbc.Driver")**，类似语句先动态加载数据库相关的驱动，然后再进行获取连接等的操作
- 第三方服务模块把接口实现类实例注册到指定地方，源框架从该处访问实例

通过 `SPI` 的方式，第三方服务模块实现接口后，在第三方的项目代码的 `META-INF/services` 目录下的配置文件指定实现类的全路径名，源码框架即可找到实现类

SPI 设计

设计一个SPI加载类，通过当前Class的类加载器去加载META-INF/irpc/目录底下存在的资源文件

在需要加载资源时（初始化序列化框架、初始化过滤链、初始化路由策略、初始化注册中心），使用SPI加载类去实现

从而避免了在代码中通过switch语句以硬编码的方式选择资源

9. **RPC 容错层是如何设计的?**
    1. 报错日志打印

        当客户端发送请求到指定的服务提供者后，其调用对应的方法，但此时方法出现异常Exception。

        若将异常只记录在服务端中，则客户端较难定位异常发生的时间、位置与原因，因为同一个服务可能有多个服务提供者。

        因此，服务端在处理异常时，需要将所有异常捕获，并写回到客户端。

    2. 超时重试机制

        反向代理在发送请求之后，会以异步或同步的方式等待结果返回。

        因此在反向代理等待请求返回的过程中，可以对请求超时与否进行判断，并根据可重发次数进行重新发送。

    3. 服务端流量控制
        - 整体流量控制

            限制服务端的总体连接数，超过指定连接数时，拒绝剩余的连接请求。

            通过为ServerBootstrap设置最大连接数处理器，及时地对连接进行释放。

        - 单服务限流

            采用 **Semaphore** 进行流量控制，在每一个服务进行注册时，便指定服务对应的最大连接数。

            在请求到达服务端之前，配置一层前置过滤器。

            - 当前连接数超过最大连接数时，根据Semaphore的tryAcquire原理，会直接返回False，据此判断流量超峰，抛出异常。
            - 当前请求结束之后，需要对资源进行释放，也就是对Semaphore持有资源数加1。通过请求后置过滤器实现

RPC 其他问题

1. **你认为 RPC 项目设计最亮眼的地方是什么?**

    除了常规 RPC 项目必备的代理层、路由层、注册中心层、容错层等

    - 使用了异步设计，对各个操作进行解耦
        - 对于服务端：

当请求抵达服务器时，将其直接丢入业务阻塞队列中，然后开辟一个新的线程，从阻塞队列中循环获取Handler请求任务。

将获取到的任务对象交付于业务线程池进行消费处理。

- 对于客户端：

在 `RpcReferenceWrapper` 中设置一个 `isAsync` 字段，用于判断是否为异步。

若该字段为 `True`，则在动态代理层中，不需要同步阻塞等待响应结果，直接返回 `null` 即可

- 仿照 Dubbo 设计方式，引入 `@IRpcReference` 与 `@IRpcService` 注解，并将项目接入 SpringBoot。

2. **是否有投入到实际生产环境中？是否有尝试做过压力测试？**

通过设置连续请求次数为 100 / 1000 / 10000，对框架进行压力测试，发现随着请求次数梯度上升，整体接口的响应速度和结果并没有发生变化。说明框架稳定

关于线程池线程数的选择策略如下：

> 根据线程池处理任务的类型进行选择
>
> - 如果是CPU密集型任务，如：加密、解密、压缩、计算，应该根据当前服务器CPU核心数进行选择，最好是CPU核心数的1~2倍
> - 如果是IO密集型任务，如：数据库、网络传输、文件读写，应该尽可能提升线程数
> - 公式为：线程数 = CPU 核心数 *（1+平均等待时间/平均工作时间）
>   - 平均等待时间越长，说明是IO密集型，需要增大线程数
>   - 平均工作时间越长，说明是CPU密集型，需要减少线程数