OSPP 2022

项目申请书

项目编号：220640404

Seata RPC 扩展

gRPC 协议支持

申请人：曹凯勋

Github ID：106umao

邮箱：578961953@qq.com

目录

1.介绍 3
A.个人简介 3
B.关于 grpc 3
C.关于 seata 3
D.关于 grpc 在 seata 中的实现 3
E.申请书内容介绍 3
2.系统分析 4
A.rpc 模块包组织结构分析 4
B.protocol 包类功能分析，Netty 耦合程度分析 6
C.rpc 模块内部和外部对 rpc 通信接口的依赖分析 9
3.系统设计 13
A.系统重构 13
B.依赖替换 14
C.grpc 实现 16
4.时间规划 17
A.第一阶段：重构与依赖替换（7.1 – 8.1）144 小时 17
B.第二阶段：grpc 实现（8.1 – 9.1）144 小时 18
C.第三阶段：集成测试与用例编写（8.1 – 8.23）108 小时 19
5.展望未来（9.31 – …） 20

1.介绍

A.个人简介

就读于湖南工学院大三年级，seata 项目组下 seata-go 成员之一。目前正在着手 seata-go 中 TCC 模式 RM 模块的资源注册，分支事务注册等操作的设计工作。热爱生活，热爱开源，积极参与各种社区交流，参与贡献过 apache/dubbo，apache/dubbogo，seata 和 botania 等项目。

B.关于 grpc

grpc 是谷歌开源的一款语言无关性的 rpc 框架，其已然是云原生的事实标准，就目前而言，诸多优秀的开源项目都纷纷针对 grpc 进行了支持，包括但不限于 Dubbo，DolphinScheduler 等优秀的项目都对其进行了靠拢，其根本原因在于 grpc 所基于的 HTTP/2.0 协议的 tcp 多路复用技术与 protocol buffer 的高效编解码传输带来的传输效率提升。因此，在未来云原生的潮流趋势下，Seata 对 grpc 的协议扩展是不可避免的。

C.关于 seata

seata 是一组分布式事务解决方案，因此它的概念和内容涉及面是非常庞大和复杂的，但即使项目功能再庞大，其都离不开 seata 的三板斧 RM, TM, TC 三个模块，这三个模块是 seata 的基石，而 seata-core 模块的 rpc 包又是 RM, TM, TC 三个模块的通信的核心所在，所以针对 rpc 方式进行扩展，可以提升 seata 整体的性能，并且可以借助 grpc 语言无关性的特性，使 seata 能够参与语言异构的分布式事务当中，充分发挥自身的优势。

D.关于 grpc 在 seata 中的实现

在通过对 seata 的 rpc 模块的调研与分析后，发现 seata 原有的接口设计不太规范，在设计之初并没有考虑到未来有扩展协议的需求，因此我将 grpc 扩展的实现分为三个阶段进行设计:

- 重构原有接口，消除设计不合理的地方，方便进行扩展
- 对旧接口被依赖的地方进行依赖修改，使它们依赖新的抽象
- 针对新的抽象进行 grpc 的实现

E.申请书内容介绍

- 在本文第 2 部分将对 Seata RPC 模块进行分析，找出设计需要重构的地方和系统内外对 RPC 模块的依赖情况，方便对一些不符合依赖倒置原则的地方进行重构。
- 在本文第 3 部分将对重构方案，grpc 实现方案进行介绍。
- 在本文的第 4 部分进行对整个设计开发周期的时间进行详细规划
- 在本文第 5 部分结合本人对项目和社区的理解谈一谈对未来的展望

2.系统分析

目前的 rpc 包的组织结构应该是基于张乘辉重构后的状态，可以参考在 Seata 官网博客中的文章，rpc 模块通过 RemotingClient 和 RemotingServer 两个接口来对 Seata 中各个提供服务，这两个接口的实现类通过调用 Netty 模块进行通信细节操作。在分析 Seata 整个系统的依赖情况后发现除了 AbstractResourceManager 的 registerResource() 方法是通过调用 RmNettyRemotingClient 私有方法 registerResource()来间接调用 RemotingClient 的接口，其他的地方都是直接调用的 RemotingCilent 和 RemotingServer 接口，在接口调用的方面 seata-server 端的调用都是通过 RemotingServer 接口进行的，因此扩展会比较容易，而其他模块对 RemotingClient 的调用都是通过依赖具体的 RmNettyRemotingClient 和 TmNettyRemotingClient 来实现的，这些地方想要进行扩展还需要对原有 rpc 结构进行一些重构的操作。还有一个设计不利于扩展的地方就是 RmNettyRemotingClient 和 TmNettyRemotingClient 都是通过 getInstance 来获取单例对象的，这里我的设想是通过 SPI 来统一获取和管理单例，将 SingleHodler 类存放在更高层的抽象当中。但是我发现在 Netty 实现的模块内部也是通过 getInstance 的方式来获取实例的，这个地方不利于开展 SPI 的扩展

我分别从 rpc 包的 netty 包，processor 包还有根包中的接口说说我发现的一些存在的设计问题，并对每个问题简单说一下的思路。

基于分析我得到了两种重构方案，一种是再定义自己的接口，然后对原有实现进行适配操作，另一种方案是重构原有接口，然后直接在重构后的接口上进行新的实现。由于项目比较复杂，且考虑开源之夏结项时间周期，故以下分析基于第二种方案讨论，而第一种方案将在结项后再与社区一同讨论如何进行详细设计。

A.rpc 模块包组织结构分析

rpc 根包

从 rpc 包的结构可以看出来未来想要支持其他 rpc 方式扩展，但是该包还是存在一些与 Netty 耦合的接口，需要进行迁移和重构。

- RemotingClient 和 RemotingServer 接口：进行重构

  a) RemotingClient 和 RemotingServer 接口作为外部模块使用 rpc 的入口，其定义的 registerProcessor()方法将 processor 注册到 processTable 中作为 ChanlHandler 使用的回调函数，该方法职责与接口的职责关系不大，不符合单一职责原则，这个方法属于 Netty 特有的实现，应该迁移至 AbstractNettyRemoting 抽象类中
  b) RemotingClient 中其定义的 onRegisterMsgSuccess 和 onRegisterMsgFail 方法的职责用于 NettyPoolableFactory.makeObject()方法中注册新 Channel 成功或失败时将新 Channel 存入 ChannelManager 或抛出异常。这两个方法职责接口不太相关，应将其迁移到 AbstractNettyRemogting 中。

- RemotingClient 和 RemotingServer 接口作为外部模块使用 rpc 的接口，依赖了 RpcMessage 的具体实现和 Netty 的 Channel 类，而 RpcMessage 实现严重耦合了私有协议内容，不符合依赖倒置原则。经过调研分析后，我发现接口带 Channel 类的方法基本上都是再 rpc 模块内部使用，例如各种 processor 和 NettyPoolableFactory 中，而在他们之中使用 Channel 的目的主要是进行参数传递，基于此，我们可以通过优化参数传递方式来进行 SeataChannel 的抽象：

  i. 定义 SeataChannel 类型，该类型包含 Socket 连接必要字段，例如 IP, Port 等，还需要包含如 Resource ID,TransactionServiceGroup 等。在传之前把 Channel 塞进 ThreadLocal 或者其他的容器里，然后接收方通过 SeataChannel 里的 ip,port 等等作为 key 将 Channel 给拿出来用，这样一来 grpc 似乎也可以这样操作。
  ii. 对于 RpcMessage 中与 Netty 实现耦合比较大的部分为 codec 参数，故可以忽略之。

- RemotingBootstrap 接口：新的 rpc 也有服务器的构建过程，该接口可复用
- Disposable 接口：可复用
- RegisterCheckAuthHandler 接口：Message 可复用，所以该接口也可复用，该接口也在 processor 中被调用。
- RpcContext 类：其中的 channel 字段可以替换成 SeataChannel ，采用相同的思路，从 ChannelManager 获取 Channel 所以此处构建更高层抽象会比较合适！
- ShutdownHook 类：可复用，用于销毁 Cilent 和 Server
- TransportProtocolType 类：与 Netty 实现耦合，提供协议类型枚举，应迁移至 netty 包
- TransportServerType 类：同样应迁移至 netty 包
- TransactionMessageHandler 接口：必须复用，seata 的其他模式都实现了该接口。该接口与 Netty 耦合不大，仅被 processor 调用，处理事务消息的 InBound 与 OutBound。在 grpc 实现时设计一组与 processor 相同的类来调用该接口。

rpc.hook 包

包在生命周期中挂载钩子 ，可复用，采用 SPI 机制

rpc.netty 包

这个包从狭义上来说就是我们所说的 rpc 的 netty 实现了，虽然不可复用，但还是需要进行一些重构才能方便扩展新的实现。

rpc.netty.v1 包：

这个包中为私有协议序列化反序列化类，不能复用

- RmNettyRemotingClient 和 TmNettyRemotingClient 的继承链，这种只要有异化就通过继承的方案对后期扩展不是很友好，如果能用组合代替扩展的话会更好。但是已有代码能够正常工作，不需要大改。其他方面需要：

    - RmNettyRemotingClient 将 registerResouce()的实现进行重构，无论是迁移至上层还是修改实现还是将注册逻辑迁移至调用端，因为绝对不能够允许将底部类的私有方法暴露至外部模块调用！
    - RmNettyRemotingClient，TmNettyRemotingClient 的单例模式问题，想要使用 SPI 机制进行扩展，就不允许在子类中存在 SingleHodler 的设计。
- NettyServerConfig，NettyClientConfig，NettyBaseConfig：不可复用，不过后期对于新的 rpc 实现可以参考这两个类进行 config 类的设计
- NettyServerBootstrap，NettyClientBootstrap：不可复用，grpc 实现可以参考该 bootstrap 进行设计。
- NettyRemotingServer：与上述 RmNettyRemotingClient 和 TmNettyRemotingClient 类似，好在不存在单例模式问题的困扰（因为将单例问题交给 DefaultCoordinator 去考虑了）。且 seata-server 端除了 DefaultCoordinator 同时以来了 RemotingServer 和 NettyRemotingServer，其他的 AbstractCore 和他的子类都是依赖的 RemotingServer。因此扩展起来会比较方便。
- NettyPoolableFactory：用于构建 Netty 的 Channal，与 Netty 强耦合，不可复用
- NettyPoolKey：该类结合 NettyPoolableFactory 与 NettyClientChannelManager 等一起管理 Channel，不可复用。
- NettyClientChannelManager：用于客户端，rpc 连接建立后客户端会将 Channel 缓存起来，不可复用。但是应该借鉴
- ChannelUtil：工具类，用于获取 Channel 中的地址等信息。不可复用
- ChannelManager：用于服务器端，rpc 连接建立后客户端会将 Channel 缓存起来，不可复用。可借鉴。这里面的 RM_CHANNELS，TM_CHANNELS 可以在 grpc 实现时借鉴。

rpc.processor 包

这个包和 Netty 就是定义一组具体的模板方法被 Handler 中调用。processor 的接口设计耦合了 Netty 内容，所以 processor 包应整体迁移 netty 包下。不过具体 processor 的处理逻辑与整个 seata 息息相关，故在进行新的 grpc 实现时也要实现一组与该包功能完全一致的处理过程。

- RemotingProcessor 接口：定义了 process 方法。不可复用，可借鉴
- Pair 类：存放 processor 与线程池的映射关系。不需要迁移，到时候直接 copy 个一样的，可以复用
- rpc.processor.server 包和 rpc.processor.client 包：定义了 RemotingProcessor 的实现类。很重要，但是不可复用，必须借鉴。

B.protocol 包类功能分析，Netty 耦合程度分析

rpc.protocol 包

这个包的大部分内容都被系统多个模块依赖了，所以需要对该包中与 Netty 强耦合的类进行抽象或者迁移。因为与 rpc 的实现方式耦合高，所以这部分有很多内容需要在 grpc 实现过程中进行，例如各种 Request, Response 这些都可以通过定义 IDL 的 proto 文件时同时设计。

- MessageTypeAware 接口：定义一个方法，代表可获得消息类型。
- AbstractMessage 抽象类：定义抽象消息，实现 HeartbeatMessage，该类被多个类继承，依赖了 ChannelHandlerContext，与 Netty 强耦合。
- AbstractIdentifyRequest，AbstractIdentifyResponse 抽象类：继承 AbstractMessage，定义具有标识信息的请求和响应信息。无对 Netty 的直接依赖
- AbstractResultMessage 抽象类：继承 AbstractMessage，定义结果信息。无对 Netty 的直接依赖
- BatchResultMessage 类：批处理结果消息。继承 AbstractMessage。无对 Netty 的直接依赖
- HeartbeatMessage：实现 MessageTypeAware。心跳检测信息。
- IncompatibleVersionException 类：异常类，定义不兼容版本异常，被 Version 类依赖。
- MergeMessage 接口：未定义任何方法，作用仅为标记对象，表示该对象包含一组消息。
- MergedWarpMessage 类：实现 MergeMessage，继承了 AbstractMessage，作为 MergeMessage 的实现，该类的中包含了一组<AbstractMessage>消息。
- MergeResultMessage 类：实现 MergeMessage，继承了 AbstractMessage，作为 MergeMessage 的实现，该类的中包含了一组 AbstractResultMessage 消息
- MessageFuture 类：异步消息类，CompletableFuture 的包装类，setResultMessage(Object obj)用于执行消息，get 方法用于获取消息结果。其以来了私有协议相关的 RpcMessage，但是在类中并没有发现用 RpcMessage 做了什么。
- MessageType 接口：定义一组字段，为消息类型枚举，prosessorTable 就是通过 MessageType 的字段与 prosessor 建立映射关系。
- ProtocolConstants 接口：定义一组字段，表示私有协议需要用到的一些字段。
- RegisterRMRequest，RegisterRMResponse，RegisterTMRequest，RegisterTMResponse 类：注册 RM 和 TM 时用到的消息类，分别继承了 AbstractIdentifyRequest 和 AbstractIdentifyResponse 类，对 Netty 无直接依赖（所以还是得分析一下 AbstractMessage 中的 ChannelHandlerContext 在回事）
- ResultCode 类：结果状态码枚举。在 proto 文件中定义
- RpcMessage 类：该类与私有协议耦合度过高，需要进一步抽象或者进行适配设计才能方便扩展。可以忽略 codec 字段以达到通用的效果
- Version 类：用于标识 Channel 的版本号，与 Netty 耦合度过高。

rpc.protocal.transaction 包

定义了一组事务相关的消息类型，有部分类型依赖了 RpcContext 类是，其是 seata 事务特性的支撑，不可能进行修改，但是需要对其依赖的 Channel 进行处理。还有部分类依赖了 AbstractMessage，这里也需要注意。下列为依赖了 RpcContext 的类型，关于这部分的内容应该在 grpc 实现时在 proto 文件中以 message 的形式定义

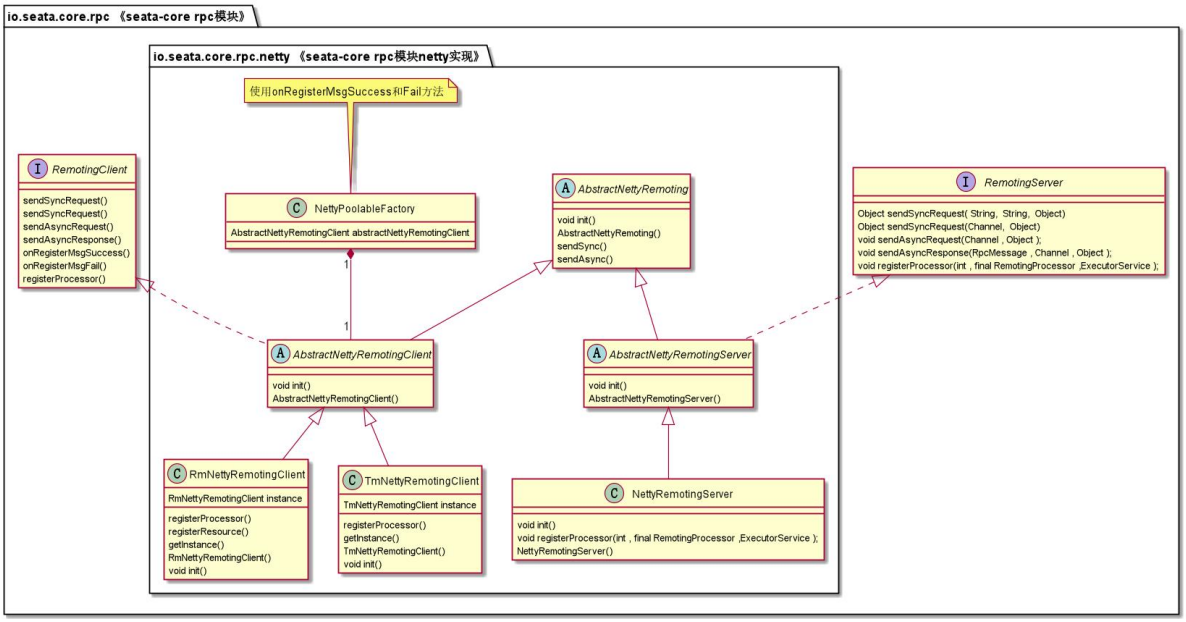
- AbstractTransactionRequest
- BranchCommitRequest
- BranchRegisterRequest
- BranchReportRequest
- BranchRollbackRequest
- GlobalBeginRequest
- GlobalCommitRequest

- GlobalLockQueryRequest
- GlobalReportRequest
- GlobalRollbackRequest
- GlobalStatusRequest
- UndoLogDeleteRequest

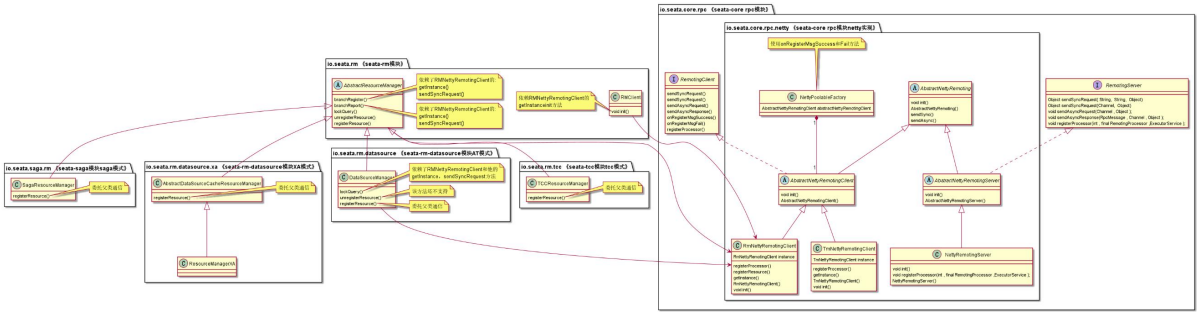
通过观察可以发现，除了 AbstractTransactionRequest 外，其他的 Request 类都是通过继承链重写了 AbstractTransactionRequest 的 handle 方法，因此依赖了 RpcContext。

C.rpc 模块内部和外部对 rpc 通信接口的依赖分析

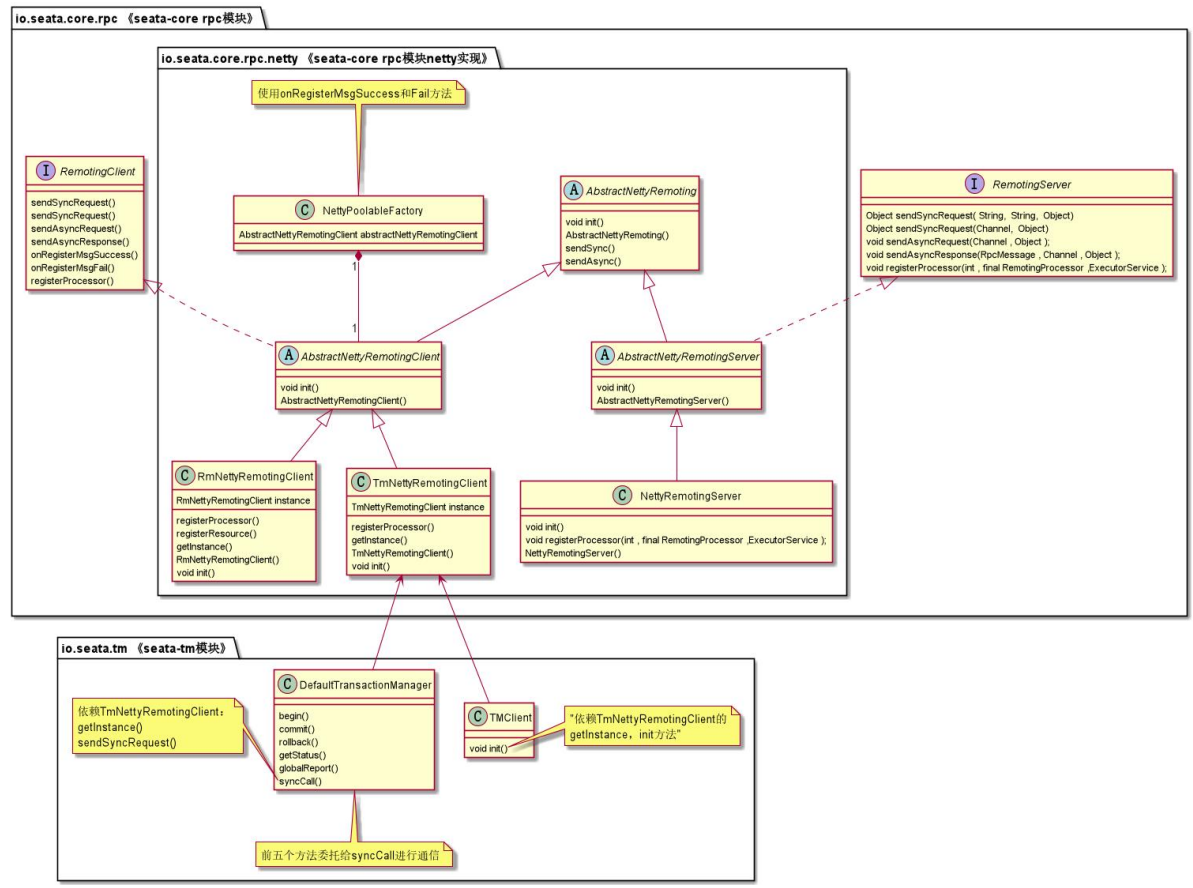
rpc 模块类图



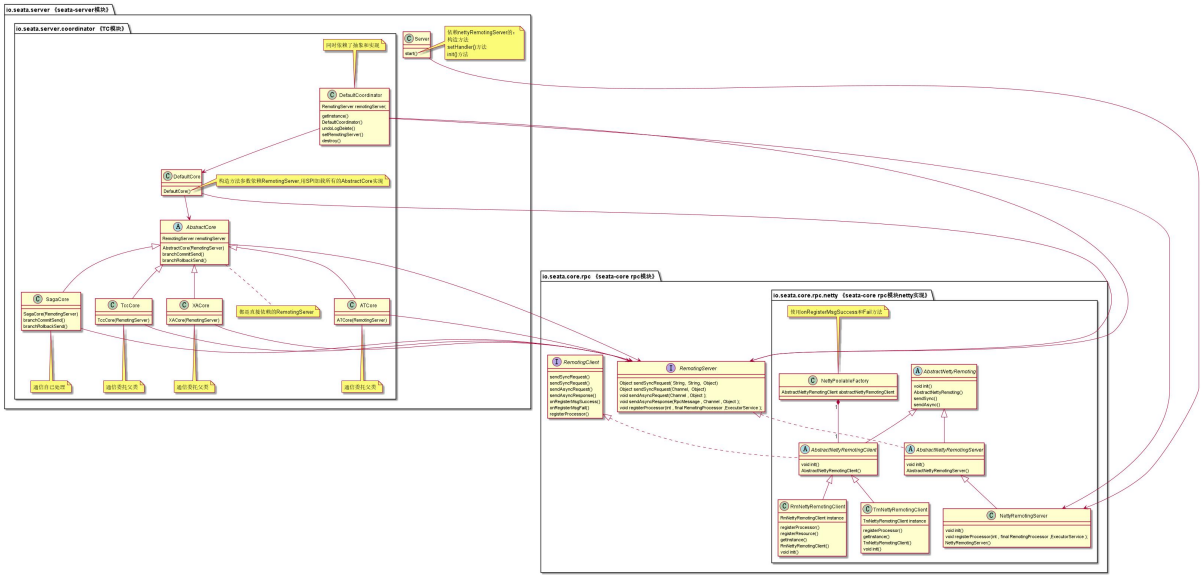
seata-rm 模块依赖类图-局部图



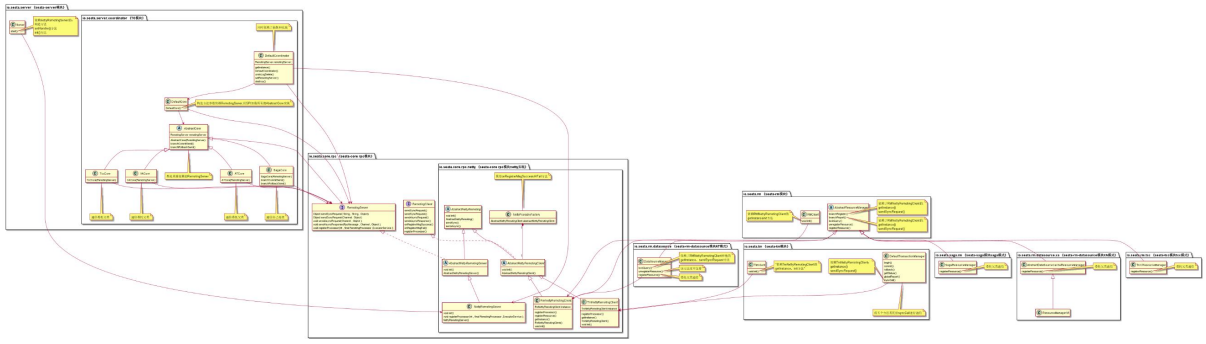
seata-tm 模块依赖类图-局部图



seata-server 模块依赖类图-局部图



全局依赖类图



部分关系依赖描述

- seata-rm 模块
  - io.seata.rm.RMClient.init()对 RmNettyRemotingClient 存在直接依赖
  - io.seata.rm.AbstractResourceManager 类对 RmNettyRemotingClient 直接依赖，不符合依赖倒置原则
    - registerResource(Resource) void 通过 RmNettyRemotingClient 依赖 registerResource->sendRegisterMessage-->RemotingClient.sendAsyncRequest。需要重构。
    - branchRegister(BranchType, String, String, String , String, String) Long 依赖 sendSyncRequest 方法，依赖了 RemotingClient
    - branchReport(BranchType , String, long, BranchStatus , String) void 依赖 sendSyncRequest 方法 依赖了 RemotingClient
- seata-tm 模块
  - seata-tm 模块中 io.seata.tm.DefaultTransactionManager.syncCall()方法对 TmNettyRemotingClient 进行的依赖，该方法中依赖了 RemotingClient 中的方法，不符合依赖倒置原则
  - io.seata.tm.TMClient.ini()对 TmNettyRemotingClient 的依赖
- TCC 模式
  - registerSource 通过 AbstractResourceManager 类 registerSource，没有直接对 rpc 模块进行依赖，无需修改
  - tcc 的 branchRegister 通过 TCCResourceManager 委托 AbstractResourceManager.branchRegister 方法进行处理，故不存在直接依赖
  - branchCommit 被 AbstractRMHandler 入站操作调动，而此处会被 core 模块 RmBranchCommitProcessor 调用
  - branchRollback 同上
- SAGA 模式
  - saga-tm 模块通过委托 seata-tm 模块进行通信，无需修改
  - saga-rm 模块通过委托 seata-rm 模块进行资源注册等操作无需修改，： seata branch commit,rollback 等操作暂时找不到 rpc 调用入口
- XA 模式，AT 模式，这两个模式被定义在 seata-rm-datasource 模块中
  - io.seata.rm.datasource.DataSourceManager.lockQuery()方法对 RmNettyRemotingClient 的依赖，该方法中依赖了 RemotingClient 中的方法，不符合依赖倒置原则

    - io.seata.rm.datasource.AbstractDataSourceCacheResourceManager.registerResource 依赖 io.seata.rm.AbstractResourceManager.registerResource 方法，对 rpc 模块属于间接依赖
    - io.seata.rm.datasource.ConnectionProxy#register 方法依赖 DefaultResourceManager.get().branchRegister()对 RPC 模块属于间接依赖
    - branchCommit 和 branchRollback 同样是 InBound 方法
    - io.seata.rm.datasource.ConnectionProxy#report 方法委托 DefaultResourceManager.get().DefaultResourceManager.get().branchRegister()方法对 RPC 模块调用
- seata-core 内部依赖
    - io.seata.core.rpc.processor.client.RmBranchCommitProcessor#handleBranchCommit 方法调用 SPI 加载的子类后 remotingClient.sendAsyncResponse(serverAddress, request, resultMessage);抽象依赖
    - io.seata.core.rpc.processor.client.RmBranchRollbackProcessor#handleBranchRollback 方法对 remotingClient.sendAsyncResponse(serverAddress, request, resultMessage);抽象依赖
    - AbstractNettyRemotingClient 对 RmNettyRemotingClient 的依赖
- seata-serve 模块
    - DefaultCoordinator 同时存在对 RemotingServer 和 NettyRemotingServer 的直接依赖
    - DefaultCore 对 RemotingServer 的依赖
    - AbstractCore 对 RemotingServer 的依赖
    - TxCore 对 RemotingServer 的依赖
    - TccCore 对 RemotingServer 的依赖
    - SagaCore 对 RemotingServer 的依赖
    - XaCore 对 RemotingServer 的依赖

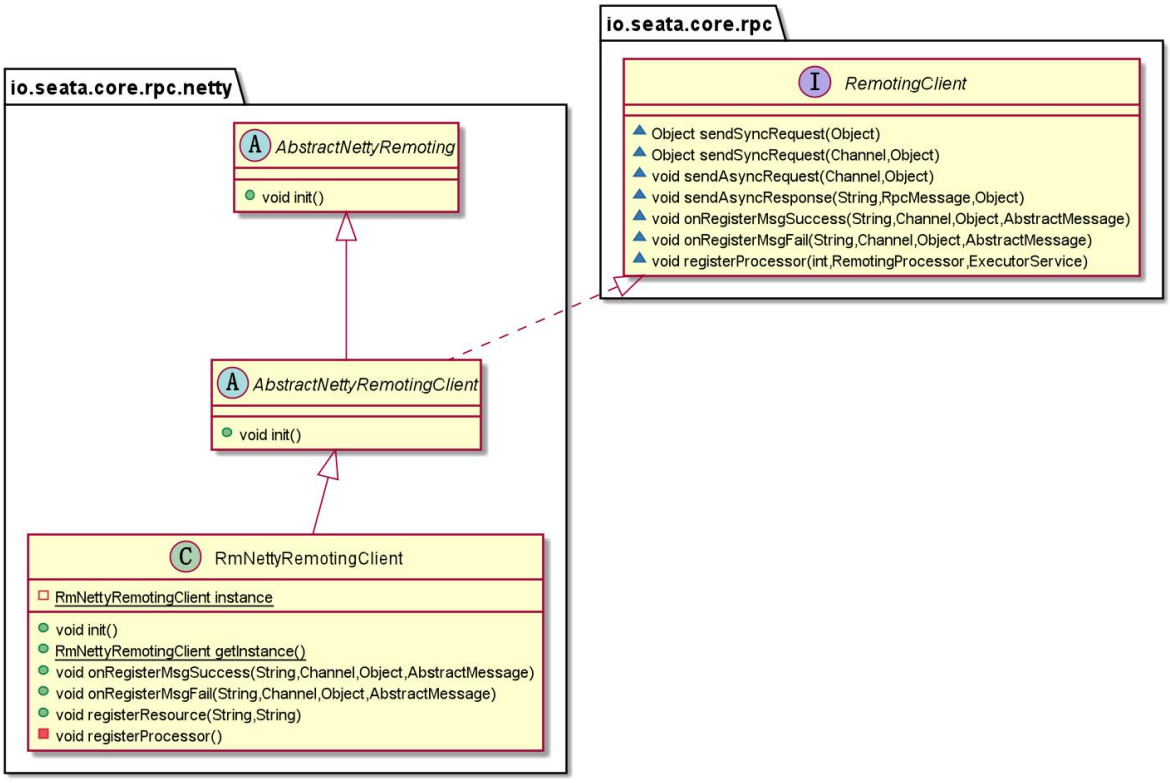
3.系统设计

基于上述的分析，我将系统设计主要分为了三个步骤，即系统重构，依赖修改和grpc 实现。

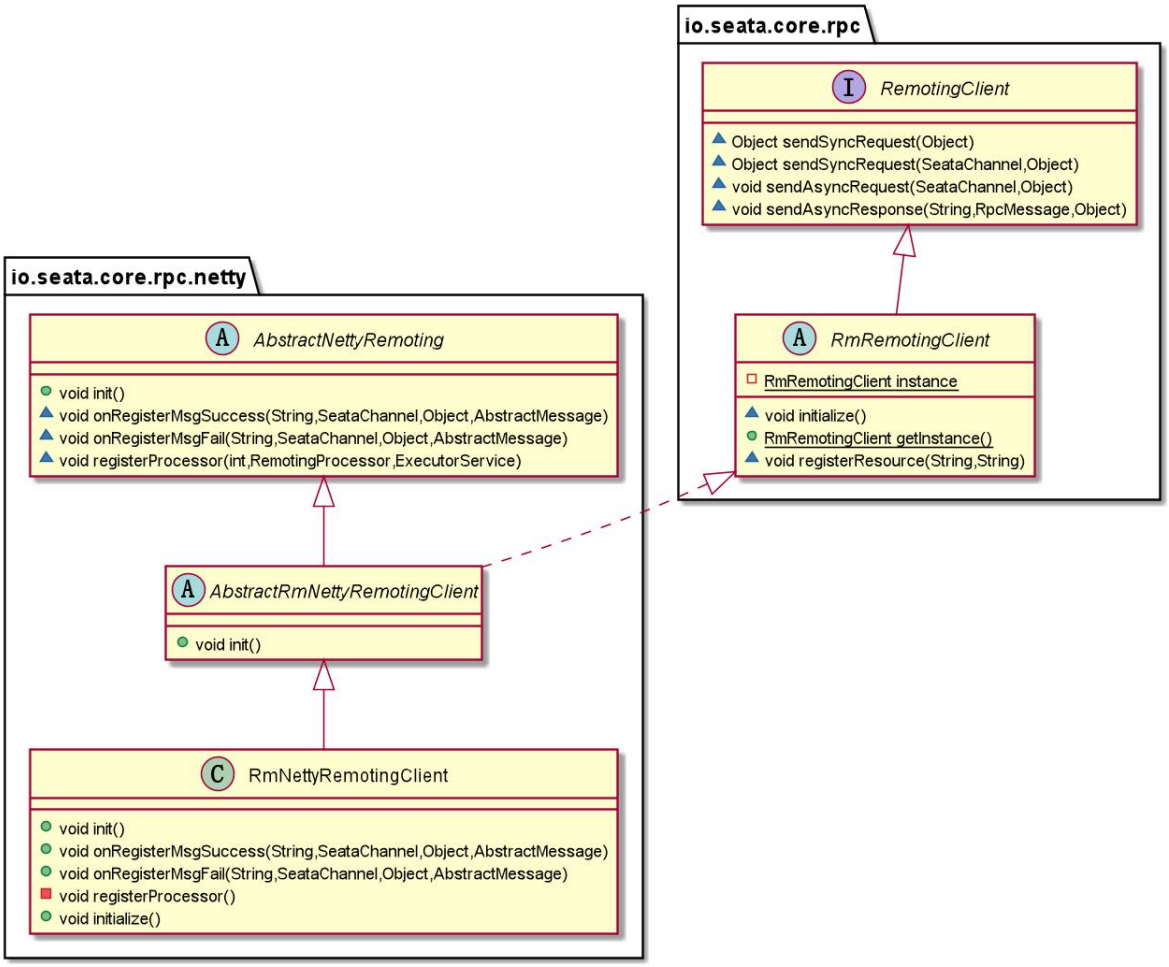
A.系统重构

通过从系统分析部分可知，我认为有两种方案可以进行系统重构。第一种是重新设计一套访问接口，然后对已有实现设计一组适配器，而第二种则是沿用原有的接口，针对接口中的内容做一些优化与重构。在于导师沟通且经过一番考虑后我决定采用第二种方案进行接口的重构。下面以 RmNettyRemotingClient 为例讨论重构前和重构后的区别。

重构前：



重构后:



上述重构操作主要有:

- 明确 RemotingClient 职责
- 增加 RmRemotingClient 抽象，将单例模式向上提升
- 更换 Channel 为 SeataChannel 依赖

B.依赖替换

这部分的操作主要是修改系统中原来对 rpc 模块依赖不合理的地方，由于需要替换的地方非常多，因此此处以 RmClient 的 init()方法为例。

替换前:

```
public static void init(String applicationId, String transactionServiceGroup) {
    RmNettyRemotingClient rmNettyRemotingClient = RmNettyRemotingClient.getInstance(applicationId,
transactionServiceGroup);
    rmNettyRemotingClient.setResourceManager(DefaultResourceManager.get());
    rmNettyRemotingClient.setTransactionMessageHandler(DefaultRMHandler.get());
    rmNettyRemotingClient.init();
}
```

替换后:

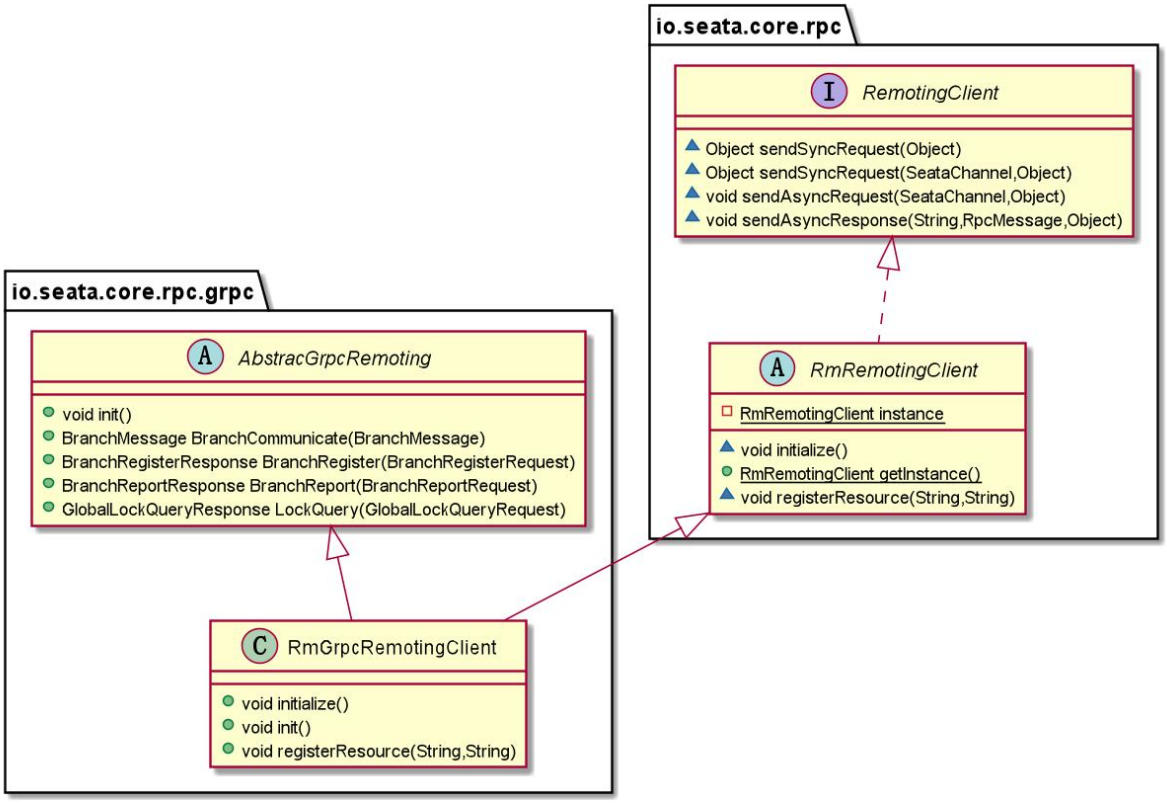
```
public static void init(String applicationId, String transactionServiceGroup) {
   RmRemotingClient.getInstance().initialize(applicationId,
transactionServiceGroup,DefaultResourceManager.get(),DefaultRMHandler.get())
}
```

可以很明显地发现，在进行重构后，未来只要扩展了新地 rpc 实现，都可以通过同样地 api 进行操作。而不是在调用方使用 type switch 硬编码。在重构后也方便利用 SPI 加载实现单例模式。例如下面地代码:

```
static private class SingletonHolder{
   private static  volatile RmRemotingClient singleton;
   private static RmRemotingClient getInstance(){
      if (singleton==null){
         synchronized (SingletonHolder.class) {
            if (singleton == null) {
               singleton = EnhancedServiceLoader.load(RmRemotingClient.class);
            }
         }
      }
      return singleton;
   }
}
```

C.grpc 实现

因为原有系统与 netty 耦合比较高，且本次重构时并没有非常彻底，故在进行 grpc 实现时还需要充分借鉴已有的 rpc 实现。下面以 rm 模块为例介绍 grpc 实现思路：



在 grpc 的实现中，定义了四个接口进行 grpc 的通信，在此处，AbstractGrpcRemoting 接口定义属于伪代码，因为这里的接口应该是定义在 proto 文件中的，而接口中使用到的通信接口是 proto 文件针对 java 编译过后的代码。在与导师沟通过程中，明确了此次 grpc 的扩展意在适应异构语言的分布式事务场景，故借鉴了 opemtrx/seata-golang 中的 IDL 文件。其文件中定义的 service 描述如下：

```
service TransactionManagerService {
    rpc Begin(GlobalBeginRequest) returns (GlobalBeginResponse);
    rpc GetStatus(GlobalStatusRequest) returns (GlobalStatusResponse);
    rpc GlobalReport(GlobalReportRequest) returns (GlobalReportResponse);
    rpc Commit(GlobalCommitRequest) returns (GlobalCommitResponse);
    rpc Rollback(GlobalRollbackRequest) returns (GlobalRollbackResponse);
}

service ResourceManagerService {
    rpc BranchCommunicate(stream BranchMessage) returns (stream BranchMessage);
    rpc BranchRegister(BranchRegisterRequest) returns (BranchRegisterResponse);
```

```
  rpc BranchReport(BranchReportRequest) returns (BranchReportResponse);
  rpc LockQuery(GlobalLockQueryRequest) returns (GlobalLockQueryResponse);
}
```

在 seata-golang 中的 proto 文件还定义了一系列的 message 作为 service 所使用的 DTO,因为内容比较多，所以在此不再赘述。

因为在 seata 中分别存在四种事务模式，seata 中也定义了多种 MessageType 来区分不同模式发出的通信信息。所以在 grpc 实现中也需要考虑匹配不同 MessageType 来进行不同的事务调用操作。此处将采用 guaua 提供的 EvenBus 进行与 netty 实现中的 prosessorTable 中相同的功能。定义一组订阅者 Subscriber 在客户端与服务器启动的过程中注册在 EvenBus 中监听通信到来的事件，然后更具不同的 EventType 调用订阅者的 onEvent 方法。在 proto 中定义的 service 的实现类中激活 EventBus 中的事件。

4.时间规划

在报名开源之下之前，我找到了一份暑期实习工作，目前还没有入职，我认为开源经历更加重要，开源社区的同学，大佬比公司的人更加友好，在这里能结识到志同道合的朋友，提升自己的技能，因此如果有幸能够参与到 seata 的这个项目中来，我将会拒绝这份实习工作，全身心投入到项目开发中来，所以我会有充足的时间完成这个项目的任务。也许您认为我目前正在参与其他开源社区的工作，会无暇参加本次开源之夏的开发，但是其实这并不会与本次项目的任务产生冲突，其实在报名开源之夏之初，我就考虑过这个问题，因为我目前在 seata-go 中的工作与 rpc 通信关联性非常强，如果我能够在将 seata-go 中的项目做好，这同样有助于我在前期进行 grpc 扩展时理解业务流程逻辑，在后期进行针对不同事务模式适配 grpc 模块时也会比较畅通，反之，如果我能够完成此次开源之夏的任务，这也将对我在 seata-go 中的工作大有裨益。正是因为这样，我才选择了 seata 的 grpc 协议扩展这个项目。

我认为在与导师沟通这个方面是非常重要的，因为他决定了我是否能够顺利完成本次开源之夏的任务，但在时间选择上也是非常难以把控的，因此我打算每周与导师进行一次 30~60 分钟的视频通话，与导师汇报本周工作的完成情况，下周工作如何准备。

以下是我的一些简短的时间规划，在后期开发过程中或与导师沟通之后可能会有一些小的改动。

我将整个项目开发总共划分为 3 阶段，11 周，396 小时。

A.第一阶段：重构与依赖替换（7.1 – 8.1）144 小时

- 第一周 36 小时
  - 定义 SeataChannel，重构 RemotingServer, RemotingClient 接口 （35 小时）
  - 向导师汇报进度与安排（1 小时）

- 第二周 36 小时
  - 重构实现类单例模式 （5 小时）
  - 修改 Netty 实现对 Channel 的依赖改为对 SeataChannel（5 小时）
  - 设计单元测试（5 小时）
  - 修改 rm 模块对 rpc 模块的依赖（20 小时）
  - 向导师汇报进度与安排（1 小时）
- 第三周 36 小时
  - 设计单元测试 （5 小时）
  - 修改 tm 模块对 rpc 模块的依赖（20 小时）
  - 设计单元测试（5 小时）
  - 修改 server 模块对 rpc 模块的依赖（5 小时）
  - 向导师汇报进度与安排（1 小时）
- 第四周 36 小时
  - 修改 server 模块对 rpc 模块的依赖（10 小时）
  - 测试单元测试（5 小时）
  - Code Review （10 小时）
  - 弹性时间安排（10 小时）
  - 向导师汇报进度与安排（1 小时）

B.第二阶段：grpc 实现（8.1 – 9.1）144 小时

- 第一周 36 小时
  - 选择版本依赖，定义 grpc 接口内容（35 小时）
  - 向导师汇报进度与安排（1 小时）
- 第二周 36 小时
  - rm 实现 grpc 接口（20 小时）
  - 设计单元测试（5 小时）
  - tm 实现 grpc 接口（10 小时）
  - 向导师汇报进度与安排（1 小时）
- 第三周 36 小时

- tm 实现 grpc 接口（10 小时）
- 设计单元测试（5 小时）
- server 实现 grcp 接口（20 小时）
- 向导师汇报进度与安排（1 小时）

- 第四周 36 小时
  - 测试单元测试（5 小时）
  - Code Review （10 小时）
  - 完善设计细节（10 小时）
  - 弹性时间安排（10 小时）
  - 向导师汇报进度与安排（1 小时）

C.第三阶段：集成测试与用例编写（8.1 – 8.23）108 小时

- 第一周 36 小时
  - 测试 tcc 模式功能（20 小时）
  - 测试 xa 模式空能（15 小时）
  - 向导师汇报进度与安排（1 小时）
- 第二周 36 小时
  - 测试 xa 模式功能（5 小时）
  - 测试 saga 模式功能（20 小时）
  - 测试 at 模式功能（10 小时）
  - 向导师汇报进度与安排（1 小时）
- 第三周 36 小时
  - 测试 at 模式功能（10 小时）
  - 选用进行 sample 代码编写（10 小时）
  - 进行结项报告文档编写（15 小时）
  - 向导师汇报进度与安排（1 小时）

5.展望未来（9.31 – …）

本次开源之夏的项目难度非常大，能够完成可能也就属于刚刚及格的程度，因此在结项后还我还会投入更多时间参与 seata 社区的交流，完善自己的代码，推动社区向前发展。期望能够在以后的开源之夏活动中成为别人的导师。