

一、项目架构介绍

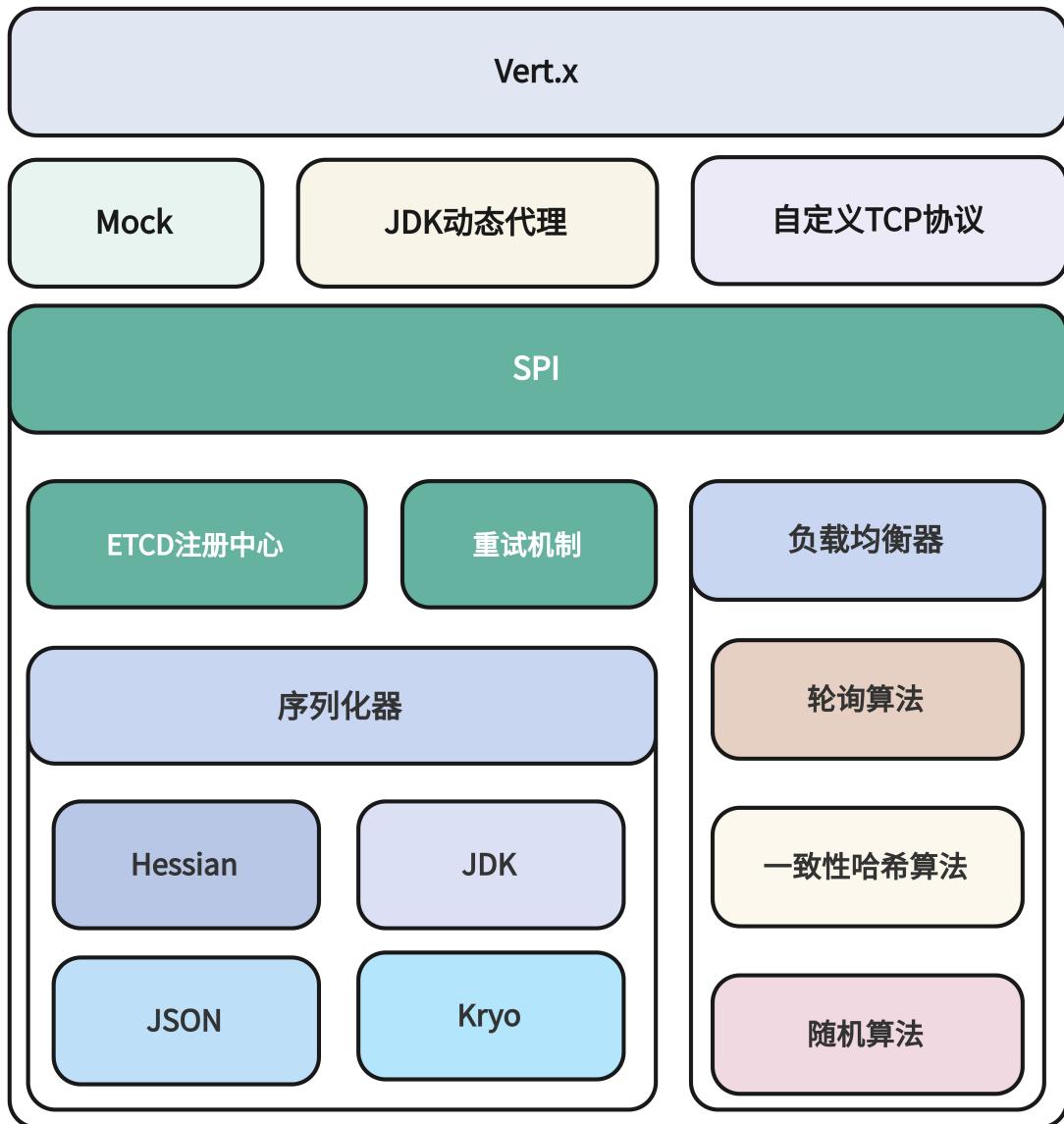
本项目基于Java + Etcd + Vert.x的高性能RPC框架。

本RPC框架采用模块化、可扩展理念构建。通过单例、工厂、装饰者模式实现分层解耦，支持灵活扩展。

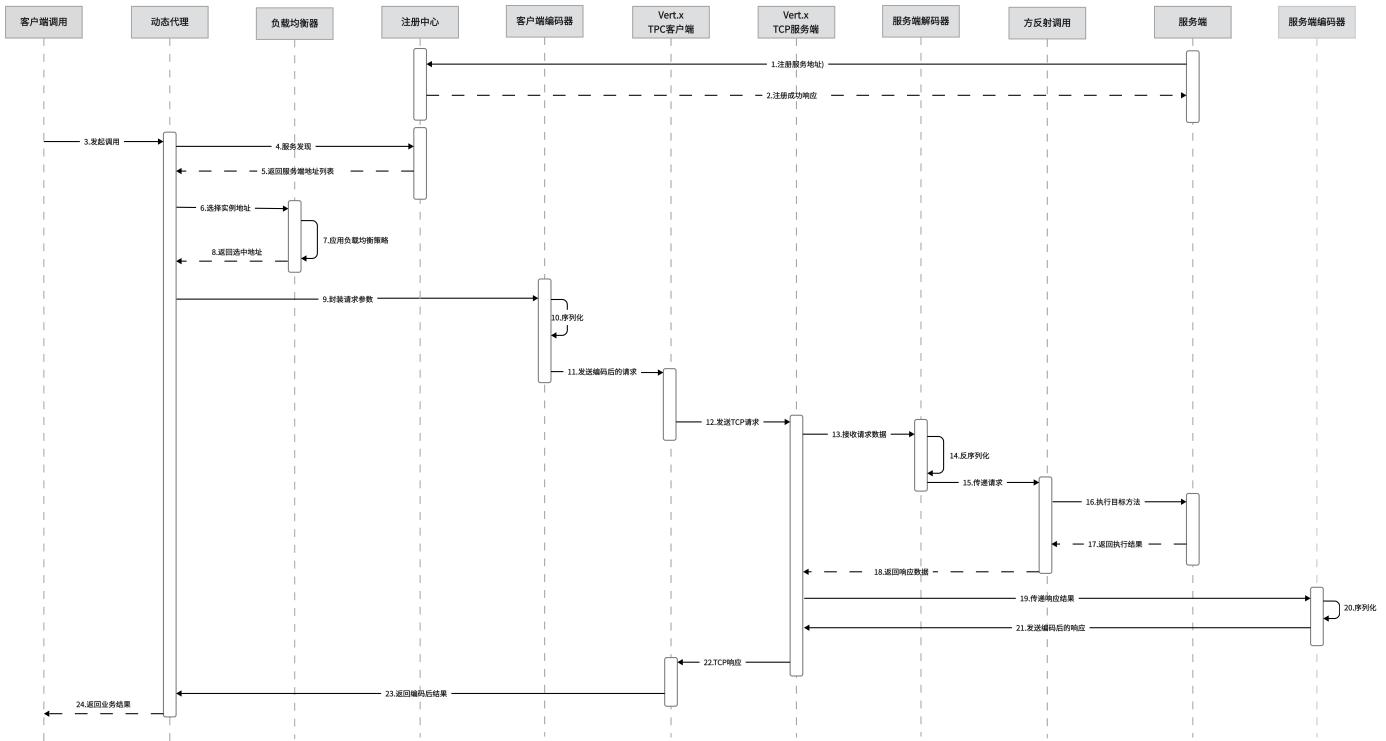
框架核心特征

- 框架内采用 Vert.x 作为异步网络传输层框架，支撑高并发网络交互。
- 实现 Mock 功能，便于客户端在服务端不可用时，通过 Mock 模拟服务响应进行测试。
- 底层采用 JDK 动态代理技术，实现 RPC 透明化调用（远程调用像本地方法调用一样简单）。
- 为提升传输效率，摒弃 HTTP 协议，采用自定义 TCP 协议（通过精简协议头、二进制编码等方式压缩数据包体积）。
- 集成 ETCD，基于其键值存储能力实现注册中心，支持服务注册与发现。
- 利用 `guava-retrying` 实现重试机制，增强调用容错性。
- 支持 Hessian、JDK、JSON 等多种序列化器；若不满足需求，可通过 SPI 机制自定义扩展。
- 支持轮询、随机、一致性哈希等多种负载均衡算法，可根据业务需求灵活切换。
- 为实现开箱即用，与 Spring Boot Starter 集成，通过 `@RpcService` `@RpcReference` 等注解式配置快速接入。

系统架构图如下：



二、项目时序图



三、TCP消息结构

本 RPC 框架的 请求协议头固定为 18 字节，各字段按如下结构设计：

字段名	位长	作用说明
魔数	16bit (2字节)	固定值，用于 校验请求合法性 ，防止非法数据包入侵。
版本号	8bit (1字节)	标识协议版本，支持 多版本兼容 （后续协议升级时，服务端可识别旧版本请求）。
序列化器	8bit (1字节)	标识序列化。
状态	8bit (1字节)	标记请求状态，用于服务端快速判断请求合法性。
请求ID	64bit (8字节)	全局唯一 ID（如 UUID 生成），用于 关联请求与响应 （异步调用时，客户端通过 ID 匹配结果）。
请求类型	8bit (1字节)	区分请求类型，用来区分请求或是响应
数据长度	32bit (4字节)	记录请求体（内容 Data）的字节长度，用于 解决 TCP 粘包 / 半包问题 。

示例图：

请求头	魔数 16bit	版本号 8bit	序列化器 8bit	状态 8bit	请求id 64bit	请求类型 8bit	数据长度 32bit
请求体	内容Data						

四、项目目录结构

```
ruott-rpc/
├── .idea/
├── example-common/          # 公共示例模块（可能放通用 DTO、接口）
├── example-customer/        # 客户端示例模块（RPC 调用方示例）
├── example-provide/         # 服务端示例模块（RPC 提供方示例）
├── example-springboot-consumer/ # Spring Boot 客户端示例
├── example-springboot-provide/ # Spring Boot 服务端示例
└── ruott-rpc-core/          # RPC 核心模块（框架核心代码）
    ├── src/                  # 源码目录
    │   ├── main/              # 主代码
    │   │   ├── java/           # Java 源码
    │   │   │   └── com/          # 包根路径
    │   │   │       └── ruott/ # 项目名
    │   │   │           └── rpc/ # RPC 框架根包
    │   │   │               ├── bootstarter/ # 服务提供者/服务消费者初始化
    │   │   │               ├── config/      # 配置类
    │   │   │               ├── exception/ # 异常定义
    │   │   │               ├── loadbalancer/ # 负载均衡策略
    │   │   │               ├── model/       # 数据模型（请求、响应等）
    │   │   │               ├── protocol/    # 协议封装（编解码、协议定义）
    │   │   │               ├── proxy/       # 动态代理
    │   │   │               ├── registry/   # 注册中心
    │   │   │               ├── retry/      # 重试机制
    │   │   │               ├── serializer/ # 序列化工具
    │   │   │               ├── server/     # 服务端核心逻辑（请求处理）
    │   │   │               ├── spi/        # SPI 扩展
    │   │   │               ├── utils/     # 工具类
    │   │   │               └── RpcApplication.java # 框架配置初始化
    │   │   └── resources/        # 配置文件（如 application.yml 等）
    │   └── test/                # 测试代码（单元测试、集成测试）
    └── target/                # 编译输出目录（class、jar 等）
    └── .gitignore/            # Git 忽略规则
    └── pom.xml/               # Maven 配置文件（依赖、打包等）
└── ruott-rpc-spring-starter/ # Spring Boot Starter 模块（简化框架集成）
```

五、框架使用说明

5.1 非SpringBoot应用

参照 `example-provider` 与 `example-customer` 示例

- 导入依赖

```
<dependency>
    <groupId>org.ruott</groupId>
    <artifactId>ruott-rpc-core</artifactId>
    <version>1.0-SNAPSHOT</version>
</dependency>
```

- 服务端使用

- 配置yaml依赖或properties，下面拿yaml举例。

```
ruott:
  rpc:
    serverName: order-server          #注册服务名称
    host: 127.0.0.1                   #注册服务IP
    port: 8081                         #注册服务端口
    serializer: jdk                   #指定序列化器
    registry-config:
      registrar: etcd                #固定etcd
      address: http://127.0.0.1:2379 #注册中心地址
      timeout: 15000                 #超时连接时间
```

- 项目入口调用初始化方法：

```
public static void main(String[] args) {
    List<ServiceRegistryInfo<?>> registryInfos = new ArrayList<>() {{
        add(new ServiceRegistryInfo<UserService>(UserService.class.getName(),
UserServiceImpl.class));
        add(new ServiceRegistryInfo<OrderService>
(OrderService.class.getName(), OrderServiceImpl.class));
    }};
    ProvideBootStarter.init(registryInfos);
}
```

- 客户端使用

- yaml配置文件

```
ruott:
  rpc:
```

```

serverName: user-server
host: 127.0.0.1
port: 28000
mock: false
serializer: jdk
#负载均衡算法,支持: {随机: random 轮询: roundRobin 一致性哈希: consistentHash}
loadBalance: consistentHash
#重试策略
retryConfig:
  #固定时间重试
  strategy: 'fixedTimeRetry'
  fixedTimeRetry:
    #重试次数
    retryNumber: 3
    #固定时间 单位秒
    retryTime: 3

```

- 进行初始化使用

```

public static void main(String[] args) {
    //初始化
    ConsumerBootStrap.init();

    //调用动态代理
    OrderService proxy = ServiceProxyFactory.getProxy(OrderService.class);
    System.out.println(proxy.getOrderNumber(1));
}

```

5.2 SpringBoot应用

可以参照项目中的: `example-springboot-provide` 与 `example-springboot-consumer`

导入依赖: 导入starter依赖

```

<dependency>
  <groupId>org.ruott</groupId>
  <artifactId>ruott-rpc-spring-starter</artifactId>
  <version>0.0.1-SNAPSHOT</version>
</dependency>

```

配置yaml文件: 与上方描述相同

应用注解:

- `@EnableRuottRpc(needServer = true)`

- 需放在 **Spring Boot** 主应用类（标注 `@SpringBootApplication` 的类）上，开启RPC功能
- 参数说明优化 `needServer` 参数用于控制是否启动服务端功能：
 - `true`：作为 **服务提供者**（启动服务端监听端口）。
 - `false`：作为 **服务消费者**（仅消费服务，不启动服务端）。
 - 默认值为 `false`，适用于纯客户端场景。
- `@RpcService`
 - 需放在 **服务实现类** 上，将该类注册为 RPC 服务。
 - **Note**

需注意：该注解本身未直接集成 Spring IOC 相关 Bean 注入能力（如 `@Component` `@Service` 等作用），因此被标注的实现类需额外通过 Spring 标准注解（`@Component` / `@Service` 等）或其他方式（如配置类 `@Bean` 声明）注入到 Spring IOC 容器中，才能被框架正常扫描和管理，保障 RPC 服务暴露流程完整。
- `@RpcReference`
 - 需放在 **消费者项目** 的成员变量上，用于注入远程服务代理。

5.3 SPI扩展

SPI (Service Provider Interface) 是一种插件化扩展机制，允许框架在运行时动态加载和替换核心组件，无需修改源码。

使用说明：项目中支持四种SPI

- 重试机制
- 注册中心
- 负载均衡
- 序列化器

1. 第一步

需要在 `resources` 目录下创建文件夹：`META-INF/ruott-rpc/custom`

2. 第二步

在 `META-INF/ruott-rpc/custom` 目录下，创建文件，文件名如下：

重试机制： `com.ruott.rpc.retry.RetryStrategy`

注册中心： `com.ruott.rpc.registry.Registry`

负载均衡： `com.ruott.rpc.loadbalancer.LoadBalancer`

序列化器： `com.ruott.rpc.serializer.Serializer`

3. 第三步

在自己的项目里创建对应的类，并实现第二步中的相关接口。

例如：如果需要实现序列化器就实现 `Serializer` 接口

4. 第四步

文件内容填写：

<名称=实现类全限定名称> 例如： `random=com.ruott.rpc.loadbalancer.RandomLoadBalancer`

5. 第五步

在 `application` 配置文件中指定相关配置。

例如，需要自定义一个序列化器。文件内容为： `test=com.xxx.xxx.xxx.A`

则在配置文件中 `ruott.rpc.serializer=test`

5.4 注意事项

注意事项一：

本框架基于 **ETCD** 实现服务注册与发现，因此在启动项目或集成至自有项目时，需提前安装并启动 **ETCD** 服务，同时将连接地址配置到配置文件中。

1. ETCD 已安装并运行

2. 配置注册中心地址

在项目配置文件（如 `application.yml`）中指定 ETCD 地址：

```
ruott:
  rpc:
    registry-config:
      #指定注册中心为etcd
      registrar: 'etcd'
      #指定etcd地址
      address: http://127.0.0.1:2379
```

注意事项二：

如果不使用 `ruott-rpc-spring-starter`

而是使用 `ruott-rpc-core`，则在配置文件中请用驼峰命名法，不可使用下划线命名，否则无法映射。

