RPC框架调用流程

- 接口定义
  - 把 RPC 框架对外提供的所有服务定义在一个接口 RpcAccessPoint
    - <T> T getRemoteService(URI uri, Class<T> serviceClass); ⊙ 等价于Dubbo 的 @Reference 注解，用于客户端获得远程服务的本地实例,即"桩"
    - <T> URI addServiceProvider(T service, Class<T> serviceClass); ⊙ 等价于Dubbo 的 @Service 注解，用于服务端注册服务的实现实例
    - Closeable startServer() throws Exception; ⊙ 服务端启动 RPC 框架，监听接口，开始提供远程服务。
  - 注册中心的接口 NameService
    - void registerService(String serviceName, URI uri) throws IOException;
    - URI lookupService(String serviceName) throws IOException;
- 流程
  - 服务提供者
    - URI uri = rpcAccessPoint.addServiceProvider(helloService, HelloService.class); ⊙ 向RPC框架注册服务，得到服务的地址uri ⊙
      注册服务具体流程：RPC框架服务端RpcRequestHandler维护一个map，Key 就是服务名，Value 就是服务提供方，也就是服务实现类的实例。注册服务的过程就是向map新增entry
    - nameService.registerService(serviceName, uri); ⊙ RPC 框架把这个服务的名称和地址发布到注册中心上 ⊙
      注册中心注册服务具体：把服务提供者保存到注册中心本地文件中。
      由于我们这个文件可能被多个进程读写，所以这里不能使用编程语言提供的那些锁。
      俄国是这些锁只能在进程内起作用，它依不住其他进程。我们这里直必须使用由操作系统提供的文件锁。
  - 服务调用者
    - URI uri = nameService.lookupService(serviceName); ⊙ 客户端step1：利用注册中心服务 NameService，读取注册中心本地文件，向注册中心查询serviceName的服务地址
    - HelloService helloService = rpcAccessPoint.getRemoteService(uri, HelloService.class); ⊙ 客户端step2：根据从注册中心获得的服务地址，利用RPC 框架提供的服务 RpcAccessPoint，获得远程服务的本地实例,即"桩"
    - String response = helloService.hello(name); ⊙ 客户端在调用 RPC 框架提供的服务时，实际调用的就是"桩"提供的方法，在桩的实现方法中，它会发请求的服务名和参数到服务端，服务端的 RPC 框架收到请求后，解析出服务名和参数后，调用在 RPC 框架中注册的"真正的服务提供者"，然后将结果返回给客户端
- 客户端调用过程
  - 桩
    - Dubbo 是在运行时动态生成的桩
    - gRPC 它是在编译 IDL 的时候就把桩生成好了，如何实现动态生成的？ ⊙
      - 需要动态生成的这个桩，它每个方法的逻辑都是一样的，都是把类名、方法名和方法的参数封装成请求，然后发给服务端
        ```
        public class DynamicStubFactory implements StubFactory{
         private final static String STUB_SOURCE_TEMPLATE =
           "package com.github.liyue2008.rpc.client.stubs;\n" +
           "import com.github.liyue2008.rpc.serialize.SerializeSupport;\n" +
             "\n" +
           "public class %s extends AbstractStub implements %s {\n" +
           "    @Override\n" +
           "    public String %s(String arg) {\n" +
           "        return SerializeSupport.parse(\n" +
           "                invokeRemote(\n" +
           "                        new RpcRequest(\n" +
           "                                \"%s\",\n" +
           "                                \"%s\",\n" +
           "                                SerializeSupport.serialize(arg)\n" +
           "                        )\n" +
           "                )\n" +
           "        \t\n" +
           "    }\n" +
           "";

         @Override
         @SuppressWarnings("unchecked")
        public <T> T createStub(Transport transport, Class<T> serviceClass) {
          try {
            // 填充模板
            String stubSimpleName = serviceClass.getSimpleName() + "Stub";
            String classFullName = serviceClass.getName();
            String stubFullName = "com.github.liyue2008.rpc.client.stubs." +
        stubSimpleName;
            String methodName = serviceClass.getMethods()[0].getName();

            String source = String.format(STUB_SOURCE_TEMPLATE,
        stubSimpleName, classFullName, methodName, classFullName,
        methodName);
            // 编译源代码
            JavaStringCompiler compiler = new JavaStringCompiler();
            Map<String, byte[]> results = compiler.compile(stubSimpleName +
        ".java", source);
            // 加载编译好的类
            Class<?> clazz = compiler.loadClass(stubFullName, results);
            // 把 Transport 赋值给桩
            ServiceStub stubInstance = (ServiceStub) clazz.newInstance();
            stubInstance.setTransport(transport);
            // 返回这个桩
            return (T) stubInstance;
          } catch (Throwable t) {
            throw new RuntimeException(t);
          }
        }
        ```
      - 生成桩需要两个重要的类 ⊙
        - 实现StubFactory 接口动态生成桩 ⊙
          ```
          public abstract class AbstractStub implements ServiceStub {

           protected Transport transport;

          protected byte [] invokeRemote(RpcRequest request) {
            Header header = new Header(ServiceTypes.TYPE_RPC_REQUEST, 1,
          RequestIdSupport.next());
            byte [] payload = SerializeSupport.serialize(request);
            Command requestCommand = new Command(header, payload);
            try {
            Command responseCommand = transport.send(requestCommand).get();
            ResponseHeader responseHeader = (ResponseHeader)
          responseCommand.getHeader();
            if(responseHeader.getCode() == Code.SUCCESS.getCode()) {
              return responseCommand.getPayload();
            } else {
            throw new Exception(responseHeader.getError());
            }
            } catch (ExecutionException e) {
            throw new RuntimeException(e.getCause());
            } catch (Throwable e) {
            throw new RuntimeException(e);
            }
           }

           @Override
          public void setTransport(Transport transport) {
            this.transport = transport;
           }
          }
          ```
        - 定义一个 AbstractStub 的抽象类，在这个类中实现大部分通用的逻辑，让所有动态生成的桩都继承这个抽象类
      - 客户端生成桩，需要调用HelloService helloService = rpcAccessPoint.getRemoteService(uri, HelloService.class);
        类似于Dubbo 的 @Reference 注解
        方法内部调用stubFactory.createStub(transport, serviceClass);
  - 生成桩之后，如何调用到实际的server的 ⊙
    - 桩的基类AbstractStub内部有负责netty通信的Transport类 ⊙
      ```
      public class NettyTransport implements Transport {
       private final Channel channel;
       /**
       * NettyTransport类使用 inFlightRequests 维护在途的所有请求
      CompletableFuture
       */
      private final InFlightRequests inFlightRequests;
      ```
      ```
      public class InFlightRequests implements Closeable {

       /**
       * 用于维护所有的在途请求，key为请求id，value为返回值
       */
      private final Map<Integer, ResponseFuture> futureMap = new
      ConcurrentHashMap<>();

       @Override
       public void close() {
        scheduledFuture.cancel(true);
        scheduledExecutorService.shutdown();
       }
      }
      ```
      ```
        @Override
        public  CompletableFuture<Command> send(Command request) {
          // 构建返回值
          CompletableFuture<Command> completableFuture = new
        CompletableFuture<>();
          try {
            inFlightRequests.put(new
        ResponseFuture(request.getHeader().getRequestId(), completableFuture));

            channel.writeAndFlush(request).addListener((ChannelFutureListener)
        channelFuture -> {
              // 处理发送失败的情况
              if (!channelFuture.isSuccess()) {
               completableFuture.completeExceptionally(channelFuture.cause());
                channel.close();
              }
              });
            } catch (Throwable t) {
              // 处理发送异常
            inFlightRequests.remove(request.getHeader().getRequestId());
             completableFuture.completeExceptionally(t);
            }
            return completableFuture;
        }
      ```
    - AbstractStub的invokeRemote方法会调用transport.send方法 ⊙
    - RpcRequestHandler的handle方法 ⊙
      ```
      @Singleton
      public class RpcRequestHandler implements RequestHandler,
      ServiceProviderRegistry {
        private static final Logger logger =
      LoggerFactory.getLogger(RpcRequestHandler.class);
        /**
        * Key 就是服务名，Value 就是服务提供方，也就是服务实现类的实例
        */
        private Map<String/*service name*/, Object/*service provider*/>
        serviceProviders = new HashMap<>();

         @Override
        public Command handle(Command requestCommand) {
          Header header = requestCommand.getHeader();
            // 1.从payload中反序列化RpcRequest
            RpcRequest rpcRequest =
        SerializeSupport.parse(requestCommand.getPayload());
            try {
            // 2.根据 rpcRequest 中的服务名，去成员变量 serviceProviders 中查找已注
        册服务实现类的实例
            Object serviceProvider =
        serviceProviders.get(rpcRequest.getInterfaceName());
            if(serviceProvider != null) {
              // 3.找到服务提供者，利用Java反射机制调用服务的对应方法
              String arg =
        SerializeSupport.parse(rpcRequest.getSerializedArguments());
              Method method =
        serviceProvider.getClass().getMethod(rpcRequest.getMethodName(),
        String.class);
              String result = (String ) method.invoke(serviceProvider, arg);
              // 4.把结果封装成响应命令并返回
              return new Command(new ResponseHeader(type(),
        header.getVersion(), header.getRequestId()), SerializeSupport.serialize(result));
            }
            // 如果没找到，返回NO_PROVIDER错误响应。
            logger.warn("No service Provider of {}#{}(String)!",
        rpcRequest.getInterfaceName(), rpcRequest.getMethodName());
            return new Command(new ResponseHeader(type(), header.getVersion(),
        header.getRequestId(), Code.NO_PROVIDER.getCode(), "No provider!"), new
        byte[0]);
          } catch (Throwable t) {
            // 发生异常，返回UNKNOWN_ERROR错误响应。
            logger.warn("Exception: ", t);
            return new Command(new ResponseHeader(type(), header.getVersion(),
        header.getRequestId(), Code.UNKNOWN_ERROR.getCode(), t.getMessage()),
        new byte[0]);
          }
        }

        @Override
        public int type() {
          return ServiceTypes.TYPE_RPC_REQUEST;
        }

         /**
         * 注册 RPC 服务
         * @param serviceClass
         * @param serviceProvider
         * @param <T>
         */
         @Override
        public synchronized <T> void addServiceProvider(Class<? extends T>
        serviceClass, T serviceProvider) {
          serviceProviders.put(serviceClass.getCanonicalName(), serviceProvider);
          logger.info("Add service: {}, provider: {}.",
              serviceClass.getCanonicalName(),
              serviceProvider.getClass().getCanonicalName());
        }
        }
      ```