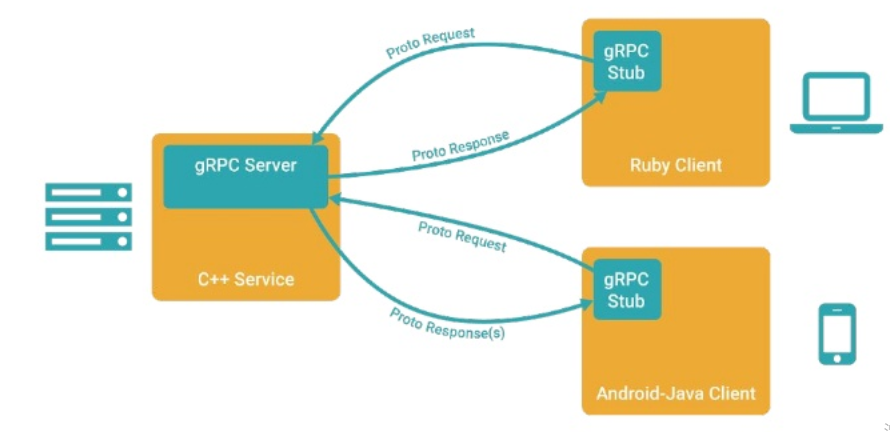
gRPC

gRPC是Google的一个高性能、开源和通用的RPC框架，面向移动和HTTP/2设计，目前提供C、Java和Go语言版本，分别是grpc、grpc-java和grpc-go。其基于HTTP/2标准设计，带来诸如双向流、流控、头部压缩、单TPC连接上的多复用请求等特性。序列化默认使用Protocol Buffer序列化机制。gRPC具有以下理念：

1. 定义一个服务，指定其能够远程调用的方法（包含参数和返回类型）
2. 在服务端实现这个接口，并运行一个gRPC服务器来处理客户端调用
3. 在客户端拥有一个Stub，访问服务端方法

gRPC客户端和服务端可以在多种环境中运行和交互——从Google内部的服务器到自己的PC，并且可以用任何gRPC支持的语言来编写。可以很容易的用JAVA创建gRPC服务端。gRPC的运行示意图如下所示：



使用gRPC服务的步骤如下：

1. 使用Protocol Buffers IDL以.proto文件的形式来描述Service
2. 使用Protocol buffer 编辑器生成server和client stub的代码
3. 扩展server class来提供service服务
4. 客户端通过stubs调用服务

# 1.gRPC使用示例

## 1.1. 添加Maven依赖

添加Maven依赖及对应的protobuf maven插件

*<dependency>*

*<groupId>io.grpc</groupId>*

*<artifactId>grpc-netty</artifactId>*

*<version>1.9.0</version>*

*</dependency>*

*<dependency>*

*<groupId>io.grpc</groupId>*

*<artifactId>grpc-protobuf</artifactId>*

*<version>1.9.0</version>*

*</dependency>*

*<dependency>*

*<groupId>io.grpc</groupId>*

*<artifactId>grpc-stub</artifactId>*

*<version>1.9.0</version>*

*</dependency>*

编译插件：

*<build>*

*<extensions>*

*<extension>*

*<groupId>kr.motd.maven</groupId>*

*<artifactId>os-maven-plugin</artifactId>*

*<version>1.5.0.Final</version>*

*</extension>*

*</extensions>*

*<plugins>*

*<plugin>*

*<groupId>org.xolstice.maven.plugins</groupId>*

*<artifactId>protobuf-maven-plugin</artifactId>*

*<version>0.5.0</version>*

*<configuration>*

*<protocArtifact>com.google.protobuf:protoc:3.0.0:exe:${os.detected.classifier}</protocArtifact>*

*<pluginId>grpc-java</pluginId> <pluginArtifact>io.grpc:protoc-gen-grpc-java:1.0.0:exe:${os.detected.classifier}</pluginArtifact>*

*</configuration>*

*<executions>*

*<execution>*

*<goals>*

*<goal>compile</goal>*

*<goal>compile-custom</goal>*

*</goals>*

*</execution>*

*</executions>*

*</plugin>*

*</build>*

Maven插件protobuf-maven-plugin能够通过插件运行命令自动扫描src/main目录下的.proto文件

* mvn protobuf:compile，compile用来编译.proto文件中的message数据结构，生成对应的Java类
* mvn protobuf:compile-custom，compile-custom用来编译.proto文件中的service定义，生成对应的service基类。

## 1.2 定义service proto文件

在src/main/proto目录下编写proto文件，sayHello.proto

*syntax = "proto3"; //指定protobuf3语法*

*option java\_multiple\_files = true;*

*option java\_package = "com.fys.grpc.proto";*

*option java\_outer\_classname = "HelloProto";*

*option objc\_class\_prefix = "HLW";*

*package services; //与java\_package组成完成的包路径*

*service HelloService { //定义服务接口*

*rpc SayHello (HelloRequest) returns (HelloResponse) {}*

*}*

*message HelloRequest { //定义接口方法参数对象*

*string name = 1;*

*}*

*message HelloResponse { //定义接口方法返回对象*

*string message = 1;*

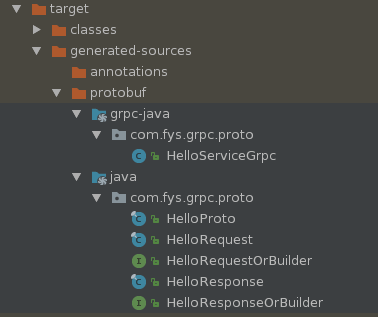
*}*

在pom.xml目录下，运行maven protobuf插件

*mvn protobuf:compile*

*mvn protobuf:compile-custom*

将.proto文件编译生成对应的java类文件，如下图：



## 1.3 编写服务发布类和Server

*public class HelloServiceImpl extends HelloServiceGrpc.HelloServiceImplBase {*

*public void sayHello(HelloRequest req,*

*StreamObserver<HelloResponse> responseObserver) {*

*HelloResponse reply = HelloResponse.newBuilder()*

*.setMessage(("Hello, "+ req.getName()))*

*.build();*

*System.out.println("Message from client:" + req.getName());*

*responseObserver.onNext(reply);*

*responseObserver.onCompleted();*

*}*

*}*

定义Server，

*public class HelloServer {*

*private int port = 50051;*

*private Server server;*

*private void start() throws IOException {*

*server = ServerBuilder.forPort(port)*

*.addService(new HelloServiceImpl())*

*.build()*

*.start();*

*Runtime.getRuntime().addShutdownHook(new Thread() {*

*@Override*

*public void run(){*

*HelloServer.this.stop();*

*}*

*});*

*}*

*private void stop() {*

*if (server != null) {*

*server.shutdown();*

*}*

*}*

*private void blockUntilShutdown() throws InterruptedException {*

*if (server != null) {*

*server.awaitTermination();*

*}*

*}*

*public static void main(String[] args) throws IOException ,InterruptedException {*

*final HelloServer server = new HelloServer();*

*server.start();*

*server.blockUntilShutdown();*

*}*

*}*

## 1.4 定义客户端

*public class HelloClient {*

*private final ManagedChannel channel;*

*private final HelloServiceGrpc.HelloServiceBlockingStub blockingStub;*

*public HelloClient(String host, int port) {*

*channel = ManagedChannelBuilder.forAddress(host,port)*

*.usePlaintext(true)*

*.build();*

*blockingStub = HelloServiceGrpc.newBlockingStub(channel);*

*}*

*public void shutdown() throws InterruptedException {*

*channel.shutdown().awaitTermination(5, TimeUnit.SECONDS);*

*}*

*public String sayHello(String name) {*

*HelloRequest request = HelloRequest.newBuilder()*

*.setName(name)*

*.build();*

*HelloResponse response= blockingStub.sayHello(request);*

*return response.getMessage();*

*}*

*public static void main(String[] args) throws InterruptedException {*

*HelloClient client = new HelloClient("192.168.246.153", 50051);*

*String content = client.sayHello("fys");*

*System.out.println(content);*

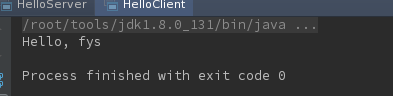
*client.shutdown();}}*

## 1.5 测试运行

Server执行，



Client执行：



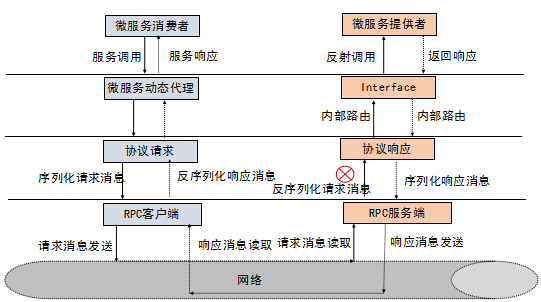
# 2.gRPC客户端调用流程

gRPC是在HTTP/2之上实现的RPC框架，HTTP/2是第7层应用层协议，运行在TCP(传输层)协议之上，相比于传统的REST/JSON机制有诸多缺点：

1. 基于HTTP/2之上的二进制协议，Protobuf序列化机制
2. 一个连接上可以多路复用，并发处理多个请求和响应
3. 多种语言的类库实现
4. 服务定义文件和自动代码生成（.proto文件和Protobuf编译工具）

gRPC还提供很多扩展点，用于对框架进行功能定制和扩展，例如，通过开发负载均衡接口可以无缝的与第三方组件进行集成对接（Zookeeper、域名解析服务和SLB服务等）。

完整的RPC调用流程示例如下：



## 2.1流程分析

以第一部分的Demo为例，客户端发起的RPC调用代码主要包括以下几个部分：

1. 根据hostname和port创建ManagedChannelImpl
2. 根据sayHello.proto文件生成的HelloServiceGrpc创建客户端Stub，用于发起RPC调用
3. 使用客户端Stub （HelloServiceBlockingStub）发起RPC调用，获取响应

代码如下所示：

*channel = ManagedChannelBuilder.forAddress(host,port)*

*.usePlaintext(true)*

*.build();*

*blockingStub = HelloServiceGrpc.newBlockingStub(channel);*

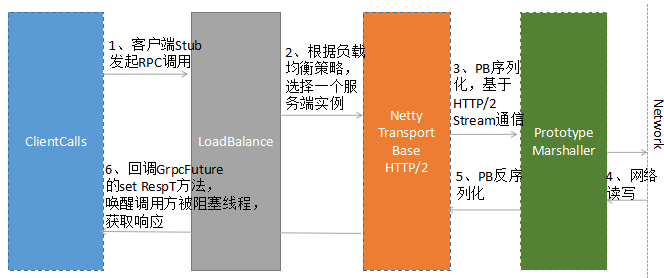
*HelloRequest request = HelloRequest.newBuilder()*

*.setName(name)*

*.build();*

*HelloResponse response= blockingStub.sayHello(request);*

gRPC的客户端调用主要基于Netty的HTTP/2客户端创建、客户端负载均衡、请求消息的发送和响应接收处理四个流程，总体流程如下图所示：



流程如下：

* 客户端Stub(HelloServiceBlockingStub)调用sayHello(request)，发起RPC调用
* 通过DnsNameResolver进行域名解析，获取服务器端的地址信息（列表），随后使用LoadBalancer策略，选择一个具体的gRPC服务端实例
* 如果与路由选用的服务端之间没有可用的连接，则创建NettyClientTransport和NettyClientHandler，发起HTTP/2连接
* 对请求消息使用PB做序列化，通过HTTP/2 Stream发送给gRPC服务端
* 服务端接收到消息后，使用Protobuf做反序列化
* 服务端处理后，通过set(Respoonse)将消息发送给客户端调用线程

客户端同步阻塞RPC中的阻塞是调用方线程阻塞，底层Transport的I/O线程（Netty NioEventLoop）仍然是非阻塞的。

## 2.2ManagedChannelImpl的创建

ManagedChannel的创建代码如下：

*ManagedChannel channel = ManagedChannelBuilder.forAddress(host,port)*

*.usePlaintext(true)*

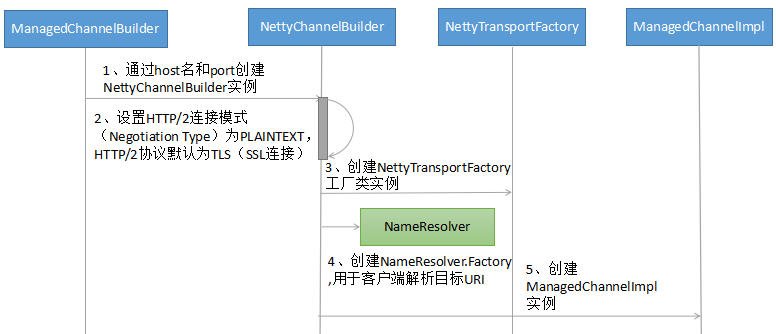
*.build();*

ManagedChannel是对Transport层SocketChannel的抽象，Transport层负责协议消息的序列化和反序列化，以及协议消息的发送和读取。ManagedChannel将处理后的请求和响应传递给与之关联的ClientCall进行上层处理。同时ManagedChannel提供了对Channel的生命周期管理（链路创建、空闲和关闭等）,类图如下所示：



ManagedChannel相关类图

ManagedChannel提供了接口式的切面ClientInterceptor，可以拦截RPC客户端调用，注入扩展点，以及功能定制，方便框架的使用者对gRPC进行功能扩展。ManagedChannel的主要实现类ManagedChannelImpl的创建流程如下：



* 使用builder模式创建ManagedChannelBuilder实现类NettyChannelBuilder，提供了buildTransportFactory工厂方法创建NettyTransportFactory，最终用于创建NettyClientTransport
* 初始化HTTP2连接方式，采用plaintext协商模式还是默认的TLS模式：h2(基于TLS之上构建的HTTP2)和h2c(直接在TCP之上构建的HTTP2)
* 创建NameResolver.Factory工厂类，用户服务类URI的解析，gRPC默认采用DNS域名解析方法

ManagedChannel实例构造完成之后，即可创建ClientCall，发起RPC调用。

## 2.3ClientCall创建流程

完成ManagedChannelImpl创建之后，由ManagedChannelImpl发起创建一个新的ClientCall实例。ClientCall的用途是业务应用层的消息调度和处理，典型用法如下所示：



关键流程如下：

* ClientCallImpl的主要构造函数是MethodDescriptor和CallOptions，其中MethodDescriptor存放需要调用RPC服务的接口名、方法名、服务调用方式如（UNARY类型）以及请求和响应的序列化和反序列化实现类。CallOptions则存放RPC调用的其他附加信息，例如超时时间、鉴权信息、消息长度限制和执行客户端调用的线程池的
* 设置压缩和解压缩的注册类（CompressorRegistry和DecompressorRegistry），以便可以按照指定的压缩算法对HTTP/2消息做压缩和解压缩

ClientCallImpl实例创建完成之后，就可以调用ClientTransport，创建HTTP/2 Client，向gRPC服务端发起远程服务调用。相关类图如下：



客户端调用流程的核心类是ClientCall，用于应用层的消息调度和处理，典型用法如下：

1. ClientCall的构建，代码如下：

*ClientCall<ReqT, RespT> call = channel.newCall(method, callOptions.withExecutor(executor));*

调用ReadChannel中的newCall方法：

*private class RealChannel extends Channel {*

*@Override*

*public <ReqT, RespT> ClientCall<ReqT, RespT> newCall(MethodDescriptor<ReqT, RespT> method,CallOptions callOptions) {*

*Executor executor = callOptions.getExecutor();*

*if (executor == null) {*

*executor = ManagedChannelImpl.this.executor;*

*}*

*return new ClientCallImpl<ReqT, RespT>(*

*method,*

*executor,*

*callOptions,*

*transportProvider,*

*terminated ? null : transportFactory.getScheduledExecutorService(),*

*channelTracer)*

*.setFullStreamDecompression(fullStreamDecompression)*

*.setDecompressorRegistry(decompressorRegistry)*

*.setCompressorRegistry(compressorRegistry);*

*}*

*}*

1. ClientCall的使用

*private static <ReqT, RespT> void asyncUnaryRequestCall(*

*ClientCall<ReqT, RespT> call, ReqT param,*

*ClientCall.Listener<RespT> responseListener,*

*boolean streamingResponse) {*

*startCall(call, responseListener, streamingResponse);*

*try {*

*call.sendMessage(param);*

*call.halfClose();*

*} ...*

*}*

1. ClientStream最终将ClientCall发送到Server端，如下所示：

*@Override*

*public void start(final Listener<RespT> observer, Metadata headers) {*

*......*

*final String compressorName = callOptions.getCompressor();*

*Compressor compressor = null;*

*if (compressorName != null) {*

*compressor = compressorRegistry.lookupCompressor(compressorName);*

*.......*

*prepareHeaders(headers, decompressorRegistry, compressor, fullStreamDecompression);*

*if (!deadlineExceeded) {*

*updateTimeoutHeaders(effectiveDeadline, callOptions.getDeadline(),*

*context.getDeadline(), headers);*

*if (retryEnabled()) {*

*stream = clientTransportProvider.newRetriableStream(method, callOptions, headers, context);*

*} else {*

*ClientTransport transport = clientTransportProvider.get(*

*new PickSubchannelArgsImpl(method, headers, callOptions));*

*Context origContext = context.attach();*

*try {*

*stream = transport.newStream(method, headers, callOptions);*

*} finally {*

*context.detach(origContext);*

*}*

*}*

*}*

*if (callOptions.getAuthority() != null) {*

*stream.setAuthority(callOptions.getAuthority());*

*}*

*if (callOptions.getMaxInboundMessageSize() != null) {*

*stream.setMaxInboundMessageSize(callOptions.getMaxInboundMessageSize());*

*}*

*if (callOptions.getMaxOutboundMessageSize() != null) {*

*stream.setMaxOutboundMessageSize(callOptions.getMaxOutboundMessageSize());*

*}*

*stream.setCompressor(compressor);*

*stream.setFullStreamDecompression(fullStreamDecompression);*

*stream.setDecompressorRegistry(decompressorRegistry);*

*channelTracer.reportCallStarted();*

*stream.start(new ClientStreamListenerImpl(observer));*

*context.addListener(cancellationListener, directExecutor());*

*.....*

*}*

1. ClientStream的初始化，ClientStream封装了NettyClientTransport，ClientTransport的启动如下所示：

*public Runnable start(Listener transportListener) {*

*lifecycleManager = new ClientTransportLifecycleManager(*

*Preconditions.checkNotNull(transportListener, "listener"));*

*EventLoop eventLoop = group.next();*

*if (keepAliveTimeNanos != KEEPALIVE\_TIME\_NANOS\_DISABLED) {*

*keepAliveManager = new KeepAliveManager(*

*new ClientKeepAlivePinger(this), eventLoop, keepAliveTimeNanos, keepAliveTimeoutNanos,keepAliveWithoutCalls);*

*}*

*handler = NettyClientHandler.newHandler(*

*lifecycleManager,*

*keepAliveManager,*

*flowControlWindow,*

*maxHeaderListSize,*

*GrpcUtil.STOPWATCH\_SUPPLIER,*

*tooManyPingsRunnable,*

*transportTracer);*

*NettyHandlerSettings.setAutoWindow(handler);*

*negotiationHandler = negotiator.newHandler(handler);*

*Bootstrap b = new Bootstrap();*

*b.group(eventLoop);*

*b.channel(channelType);*

*if (NioSocketChannel.class.isAssignableFrom(channelType)) {*

*b.option(SO\_KEEPALIVE, true);*

*}*

*for (Map.Entry<ChannelOption<?>, ?> entry : channelOptions.entrySet()) {*

*b.option((ChannelOption<Object>) entry.getKey(), entry.getValue());*

*}*

*b.handler(negotiationHandler);*

*ChannelFuture regFuture = b.register();*

*channel = regFuture.channel();*

*handler.startWriteQueue(channel);*

*channel.writeAndFlush(NettyClientHandler.NOOP\_MESSAGE).addListener(new ChannelFutureListener() {*

*@Override*

*public void operationComplete(ChannelFuture future) throws Exception {*

*if (!future.isSuccess()) {*

*lifecycleManager.notifyTerminated(Utils.statusFromThrowable(future.cause()));*

*}*

*}*

*});*

*channel.connect(address);*

*if (keepAliveManager != null) {*

*keepAliveManager.onTransportStarted();*

*}*

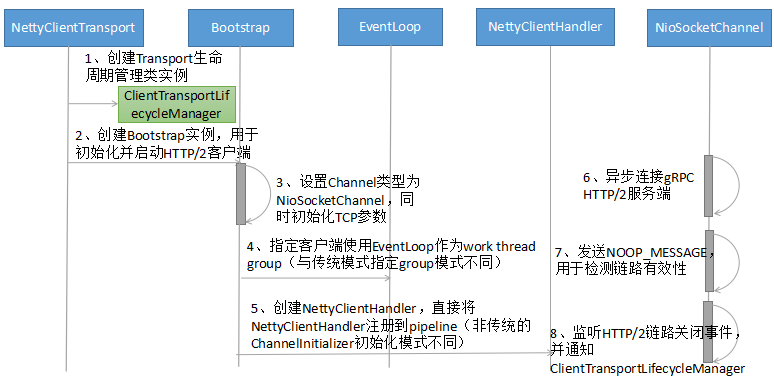
*return null;*

*}*

通过调用newStream创建ClientStream。

## 2.4 基于Netty的HTTP/2 Client创建流程

gRPC客户端底层基于Netty4的HTTP/2协议栈框架构建，以便可以使用HTTP/2协议来承载RPC消息，gRPC的关键是实现NettyClientTransport和NettyClientHandler，客户端初始化流程如下所示：



1. NettyClientHandler的创建，级联创建Netty的Http2FrameReader,Http2FrameWriter和Http2Connection，用于构建基于Netty的Grpc http2客户端协议栈
2. Http2 Client的启动，基于Netty的BootStrap来初始化并启动客户端

* NettyClientHandler，实际上被包装ProtocolNegotiator.Handler（HTTP2的握手协商），通过BootStrap的handler方法直接加入到pipeline中
* 客户端使用work线程是一个EventLoop，多路复用

1. WriteQueue的创建，Netty的NioSocketChannel初始化并向Selector注册后，由NettyClientHandler创建WriteQueue，用于接受并处理gRPC内部的各种命令，如链路关闭指令，发送Frame指令，发送ping指令

ClientCallImpl发送Request的源码如下：

*public void sendMessage(ReqT message) {*

*try {*

*if (stream instanceof RetriableStream) {*

*@SuppressWarnings("unchecked")*

*RetriableStream<ReqT> retriableStream = ((RetriableStream<ReqT>) stream);*

*retriableStream.sendMessage(message);*

*} else {*

*InputStream messageIs = method.streamRequest(message);*

*stream.writeMessage(messageIs);*

*}*

*} .......*

*if (!unaryRequest) {*

*stream.flush();*

*}*

*}*

# 3.gRPC服务端运行

## 3.1 gRPC server的初始化

定义了Service处理逻辑后，将Service作为Server的参数传入ServerBuilder, 初始化代码如下：

*server = ServerBuilder.forPort(port)*

*.addService(new HelloServiceImpl())*

*.build()*

*.start();*

HelloServiceImpl继承类*HelloServiceImplBase*：

*public final T addService(ServerServiceDefinition service) {*

*registryBuilder.addService(service)；*

*return thisT();}*

默认使用NettyServerBuilder，启动的NettyServer。

## 3.2处理客户端请求

NettyServer.start过程的核心代码如下：

*public void start(ServerListener serverListener) throws IOException {*

*listener = checkNotNull(serverListener, "serverListener");*

*allocateSharedGroups();*

*ServerBootstrap b = new ServerBootstrap();*

*b.group(bossGroup, workerGroup);*

*b.channel(channelType);*

*b.childHandler(new ChannelInitializer<Channel>() {*

*@Override*

*public void initChannel(Channel ch) throws Exception {*

*ChannelPromise channelDone = ch.newPromise();*

*NettyServerTransport transport =*

*new NettyServerTransport(*

*ch,*

*channelDone,*

*protocolNegotiator,*

*streamTracerFactories,*

*transportTracerFactory.create(),*

*maxStreamsPerConnection,*

*flowControlWindow,*

*maxMessageSize,*

*maxHeaderListSize,*

*keepAliveTimeInNanos,*

*keepAliveTimeoutInNanos,*

*maxConnectionIdleInNanos,*

*maxConnectionAgeInNanos,*

*maxConnectionAgeGraceInNanos,*

*permitKeepAliveWithoutCalls,*

*permitKeepAliveTimeInNanos);*

*ServerTransportListener transportListener;*

*synchronized (NettyServer.this) {*

*eventLoopReferenceCounter.retain();*

*transportListener = listener.transportCreated(transport);*

*}*

*transport.start(transportListener);*

*ChannelFutureListener loopReleaser = new LoopReleaser();*

*channelDone.addListener(loopReleaser);*

*ch.closeFuture().addListener(loopReleaser);*

*}*

*});*

*ChannelFuture future = b.bind(address);*

*channel = future.channel();*

*}*

处理客户端传过来的ClientCall，为*transportListener*，核心类为ServerTransportListenerImpl，主要方法为streamCreated:

*@Override*

*public void streamCreated(*

*......*

*final class StreamCreated extends ContextRunnable {*

*@Override*

*public void runInContext() {*

*ServerStreamListener listener = NOOP\_LISTENER;*

*try {*

*//根据methodName名，获取Service定义，在前面初始化时注册*

*ServerMethodDefinition<?, ?> method = registry.lookupMethod(methodName);*

*if (method == null) {*

*method = fallbackRegistry.lookupMethod(methodName, stream.getAuthority());*

*}*

*listener = startCall(stream, methodName, method, headers, context, statsTraceCtx);*

*}*

*}*

*wrappedExecutor.execute(new StreamCreated());*

*}*

startCall如下：

*private <ReqT, RespT> ServerStreamListener startCall(ServerStream stream, String fullMethodName,*

*ServerMethodDefinition<ReqT, RespT> methodDef, Metadata headers,*

*Context.CancellableContext context, StatsTraceContext statsTraceCtx) {*

*ServerCallImpl<ReqT, RespT> call = new ServerCallImpl<ReqT, RespT>(*

*stream, methodDef.getMethodDescriptor(), headers, context,*

*decompressorRegistry, compressorRegistry);*

*ServerCallHandler<ReqT, RespT> callHandler = methodDef.getServerCallHandler();*

*statsTraceCtx.serverCallStarted(call);*

*for (ServerInterceptor interceptor : interceptors) {*

*callHandler = InternalServerInterceptors.interceptCallHandler(interceptor, callHandler);*

*}*

*ServerCall.Listener<ReqT> listener = callHandler.startCall(call, headers);*

*return call.newServerStreamListener(listener);*

*}*

*}*

在ServerImpl中根据调用的method，获取注册的service定义，之所以是从method： Map<String,ServiceMethodDefinition>中获取是因为service添加后，会自动的添加到method中，如下所示：

*InternalHandlerRegistry build() {*

*Map<String, ServerMethodDefinition<?, ?>> map =*

*new HashMap<String, ServerMethodDefinition<?, ?>>();*

*for (ServerServiceDefinition service : services.values()) {*

*for (ServerMethodDefinition<?, ?> method : service.getMethods()) {*

*map.put(method.getMethodDescriptor().getFullMethodName(), method);*

*}*

*}*

*return new InternalHandlerRegistry(*

*Collections.unmodifiableList(new ArrayList<ServerServiceDefinition>(services.values())),*

*Collections.unmodifiableMap(map));*

*}*

*}*

至此客户端和server端执行流程大致分析结果，一些高级特性如安全，负载均衡不再分析

https://grpc.io/

https://github.com/grpc/grpc-java

https://github.com/grpc/grpc-java/blob/master/README.md

http://www.infoq.com/cn/articles/grpc-server-creation-and-invocation-principle