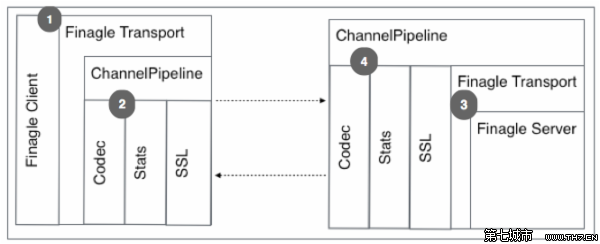
Twitter Finagle

Finagle是twitter内部的核心工具库，大量应用于Twitter内部系统，其是一个异步的网络框架，用于实现异步的RPC客户端程序和基于JVM语言的服务器，包括Java/Scala/

Jython等。Finage是基于Netty开发的支持容错、协议无关的RPC框架，其工作原理如下：



Finagle强调模块化的概念，将独立的组件组合在一起，每个组件根据配置进行替换，例如所有的跟踪器tracer都实现了相同的接口，这样创建tracer就可以追踪到存储到本地的文件，然后暴露给读取端点或者将其写入到网络之中：

在Finagle栈的底部是Transport，代表对象的流，可以异步读取和写入，在Transport中实现了为Netty的ChannelHandler，并插入到ChannelPipeline的最后。当Finagle识别到服务已经准备好读取数据时，Netty会重Socket中读取数据并使其通过ChannelPipeline，这些数据会被Codec解析，然后发送到Finagle的Transport。

对于客户端的连接，Finagle维持了Transport池，根据提供的连接池语义，向Netty请求一个新的连接，也可以重用空闲的连接，当请求新的连接时，会基于客户端的codec创建Netty的ChannelPipeline，额外的ChannelHandler添加到ChannelPipeline中，以完成统计、日志等功能：

1. Finagle客户端位于Finagle Transport之上，这个Transport为用户抽象了Netty
2. Netty ChannelPipeline包含了所有ChannelHandler实现，以完成实际的工作
3. 对于每一个连接都会创建Finagle服务器，并且会为其提供Transport进行读取和写入
4. ChannelHandler实现了协议的编码/解码逻辑、连接级别的统计以及SSL处理

在服务端，Netty提供ChannelPipelineFactory来管理codec、统计、超时以及日志等功能，在服务端ChannelPipeline中，最后一个ChannleHandler是Finagle桥。这个桥会等待新进入的连接并为每个连接创建新的Transport，这个Transport在传递给服务实现之前会包装一个新的Channel。

# Finagle概念

Finagle的三个核心概念：Future、Service、Filter

1）Future

Finagle使用的Future是com.twitter.util.Future，对异步操作的抽象，可以将Future理解为容器，其包含一个异步操作，其结果：

* 异步操作还没有完成
* 操作已经完成并包含成功的结果
* 操作失败并包含了异常结果

Future常用的用法是可以注册成功或失败的回调函数，如下面的代码：

*responseFuture.onSuccess(*

*func(response -> {*

*System.out.println(String.format("response status: %s, response string: %s",*

*response.status().toString(), response.contentString()));*

*return BoxedUnit.UNIT;*

*}));*

在responseFuture中注册成功的回调函数和失败的回调函数。

1. Service

Service是Finagle中的核心概念，可以理解为接收一个Request参数，返回一个Future对象的函数，如果定义为Java的抽象类，原型如下：

*public abstract class Service<Request, Response> {*

*public abstract Future<Response> apply(Request r);*

*}*

Finagle的Service类似于Controller的方法，可以用来处理客户端请求。Service的返回值是Future，代表操作可以异步完成。

1. Filter

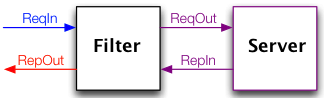
Finagle Filter类似于Servlet Filter，对Service的请求和响应进行过滤。在Finagle Filter使用类型参数明确定义输入输出的参数类型，定义为Java的抽象类，原型如下：

*public abstract class Filter<ReqIn, RepOut, ReqOut, RepIn> {*

*public abstract Future<RepOut> apply(ReqIn request, Service<ReqOut, RepIn> service);*

*}*

对于ReqIn,ReqOut,ReqOut,RepIn这四个类型参数的定义，可以参考下图：



ReqIn和ReqOut分别是Filter的入参和出参，而RepIn和RepOut则是Service的入参和出参，实际用法：

*val baseService = new Service[HttpRequest, HttpResponse] {*

*def apply(request: HttpRequest) =     Future(new DefaultHttpResponse(HTTP\_1\_1, OK))*

*}*

*val authorize = new RequireAuthorization(…)*

*val handleExceptions = new HandleExceptions(...)*

*val decoratedService: Service[HttpRequest, HttpResponse] =*

*handleExceptions andThen authorize andThen baseService*

定义Service对象baseService，两个Filter对象authorize和handleExceptions，通过filter的addThen方法，可以将Filter和Service组装在一起。

# Finagle使用示例

构建一个基于HTTP协议的echo服务端和客户端

## Server端代码

*public class Server extends Service<Request, Response> {*

*@Override*

*public Future<Response> apply(Request request) {*

*System.out.println("Request: " + request.getContentString());*

*Response response = Response.apply(Version.Http11$.MODULE$, Status.Ok());*

*response.setContentString(request.getContentString());*

*return Future.value(response);*

*}*

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

*Server service = new Server();*

*try {*

*ListeningServer server = Http.server()*

*.withLabel("Echo Server")*

*.withTracer(ZipkinTracer.mk("192.168.239.144", 9410, DefaultStatsReceiver$.MODULE$, 1.0f))*

*.serve(new InetSocketAddress(8081), service);*

*Await.result(server);*

*} catch (Exception e) {*

*e.printStackTrace();*

*}*

*}*

*}*

1. 在Finagle中，实现RPC服务非常简单，只需要继承Service抽象类，实现其apply方法，其抽象类的两个类型参数：

* Request，表示请求对象
* Response，代表返回对象

这两个对象的具体类型与Service实现类使用的具体协议相关。

1. apply方法中，使用Response工厂方法构造对象，然后将Request中的请求内容原封不动的设置到Response中，然后设置到Future中返回。
2. 启动Service实例，通过ListeningServer.withLable设置服务名称，withTracker设置监控信息

## **客户端代码**

代码如下：

*public class Client {*

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

*Service<Request,Response> service = Http.client()*

*.withLabel("Echo-Client")*

*.withTracer(ZipkinTracer.mk("192.168.239.144", 9410, DefaultStatsReceiver$.MODULE$, 1.0f))*

*.newService("192.168.239.144:8081");*

*Reader data = Reader$.MODULE$.fromStream(new ByteArrayInputStream("Greetings!".getBytes(StandardCharsets.UTF\_8)));*

*Request request = Request.apply(Version.Http11$.MODULE$, Method.Post$.MODULE$, "/", data);*

*Future<Response> responseFuture = Await.ready(service.apply(request));*

*responseFuture.onSuccess(*

*func(response -> {*

*System.out.println(String.format("response status: %s, response string: %s",*

*response.status().toString(), response.contentString()));*

*return BoxedUnit.UNIT;*

*}));*

*responseFuture.onFailure(func(e -> {*

*System.out.println("error: " + e.toString());*

*return BoxedUnit.UNIT;*

*}));*

*responseFuture.ensure(func(() -> {*

*service.close();*

*//IDE may complain here, just ignore*

*return BoxedUnit.UNIT;*

*}));*

*}*

*}*

1. 和Server类很像，创建Service实例，并监听8081端口，通过newService创建Service的stub
2. 构造消息内容为Greetings的Http请求
3. service.apply，执行客户端到服务端的RPC调用，返回值为Future<Response>。Await.ready的作用是等待一个Future执行完成后再返回，是一个同步操作
4. Future上注册请求与失败的回调函数，请求成功则简单打印响应内容

## **程序的执行**

执行后Server端输出如下：

*Request: Greetings!*

Client端的输出如下：

*response status: Status(200), response string: Greetings!*

执行成功。

http://skaka.me/blog/2016/05/02/finagle3/

http://skaka.me/blog/2016/05/01/finagle2/

https://blog.csdn.net/pingyan158/article/details/52764106

https://twitter.github.io/finagle/