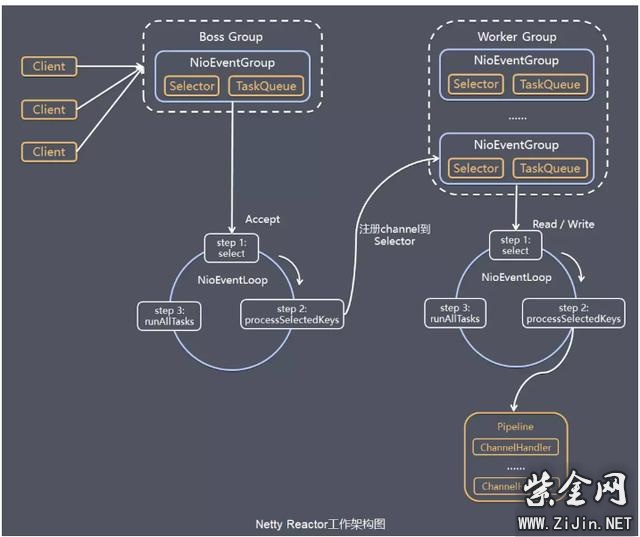
# Xnetter框架介绍

## 1 Netty介绍

Netty是由JBOSS提供的基于Java NIO的开源框架，Netty提供异步非阻塞、事件驱动、高性能、高可靠、高可定制性的网络应用程序和工具，可用于开发服务端和客户端。



### 1.1 概念解释

**Bootstrap、ServerBootstrap**

Bootstrap意思是引导，一个Netty应用通常由一个Bootstrap开始，主要作用是配置整个Netty程序，串联各个组件。Netty中Bootstrap类是用户端程序的启动引导类，ServerBootstrap是服务端启动引导类。

**Future、ChannelFuture**

正如前面详情，在Netty中所有的IO操作都是异步的，不能立刻得知消息能否被正确解决，但是可以过一会等它执行完成或者者直接注册一个监听，具体的实现就是通过Future和ChannelFutures，他们可以注册一个监听，当操作执行成功或者失败时监听会自动触发注册的监听事件。

**Channel**

Netty网络通信的组件，能够用于执行网络I/O操作。NioSocketChannel是异步的客户端 TCP Socket 连接。NioServerSocketChannel是异步的服务器端 TCP Socket 连接。NioDatagramChannel是异步的 UDP 连接。NioSctpChannel是异步的客户端 Sctp 连接。NioSctpServerChannel是异步的 Sctp 服务器端连接。

**Selector**

Netty基于Selector对象实现I/O多路复用，通过 Selector, 一个线程可以监听多个连接的Channel事件, 当向一个Selector中注册Channel 后，Selector 内部的机制即可以自动不断地查询(select) 这些注册的Channel能否有已就绪的I/O事件(例如可读, 可写, 网络连接完成等)，这样程序即可以很简单地使用一个线程高效地管理多个 Channel 。

**NioEventLoop**

NioEventLoop中维护了一个线程和任务队列，支持异步提交执行任务，线程启动时会调用NioEventLoop的run方法，执行I/O任务和非I/O任务：

I/O任务

即selectionKey中ready的事件，如accept、connect、read、write等，由processSelectedKeys方法触发。

非IO任务

增加到taskQueue中的任务，如register0、bind0等任务，由runAllTasks方法触发。

两种任务的执行时间比由变量ioRatio控制，默认为50，则表示允许非IO任务执行的时间与IO任务的执行时间相等。

**NioEventLoopGroup**

NioEventLoopGroup，主要管理eventLoop的生命周期，可以了解为一个线程池，内部维护了一组线程，每个线程(NioEventLoop)负责解决多个Channel上的事件，而一个Channel只对应于一个线程。

**ChannelHandler**

ChannelHandler是一个接口，解决I / O事件或者阻拦I / O操作，并将其转发到其ChannelPipeline(业务解决链)中的下一个解决程序。

ChannelHandler本身并没有提供很多方法，由于这个接口有许多的方法需要实现，方便使用期间，可以继承它的子类：

ChannelInboundHandler用于解决入站I / O事件

ChannelOutboundHandler用于解决出站I / O操作

或者者使用以下适配器类：

ChannelInboundHandlerAdapter用于解决入站I / O事件

ChannelOutboundHandlerAdapter用于解决出站I / O操作

ChannelDuplexHandler用于解决入站和出站事件

**ChannelHandlerContext**

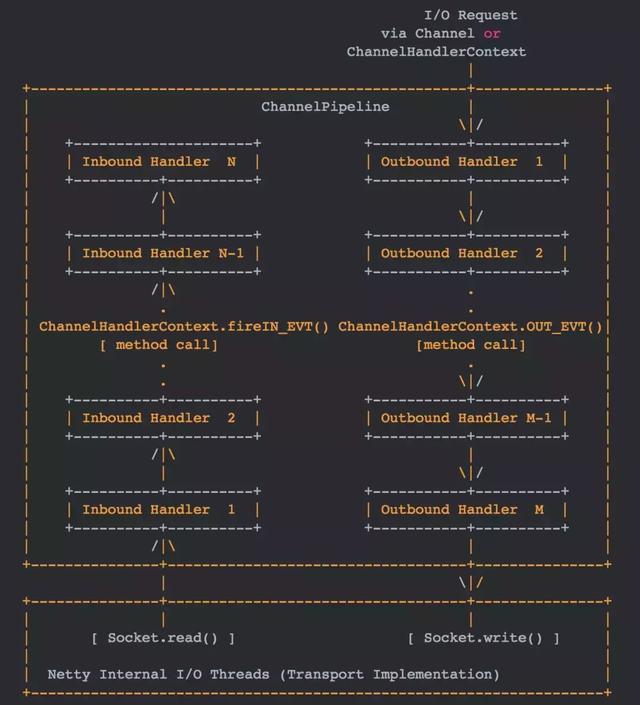
保存Channel相关的所有上下文信息，同时关联一个ChannelHandler对象。

### 1.2 ChannelPipline

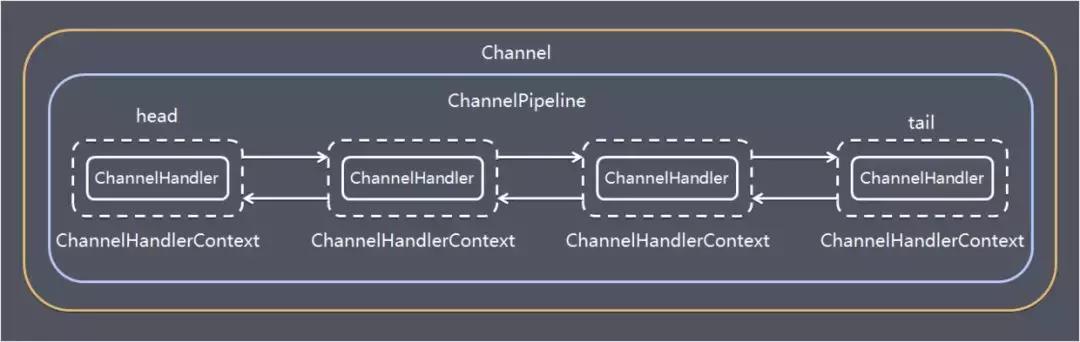
保存ChannelHandler的List，用于解决或者阻拦Channel的入站事件和出站操作。 ChannelPipeline实现了一种高级形式的阻拦过滤器模式，使客户可以完全控制事件的解决方式，以及Channel中各个的ChannelHandler如何相互交互。

入站事件由自下而上方向的入站解决程序解决，如图左侧所示。 入站Handler解决程序通常解决由图底部的I / O线程生成的入站数据。 通常通过实际输入操作（例如SocketChannel.read（ByteBuffer））从远程读取入站数据。

出站事件由上下方向解决，如图右侧所示。 出站Handler解决程序通常会生成或者转换出站传输，例如write请求。 I/O线程通常执行实际的输出操作，例如SocketChannel.write（ByteBuffer）。



在 Netty 中每个 Channel 都有且仅有一个 ChannelPipeline 与之对应, 它们的组成关系如下:

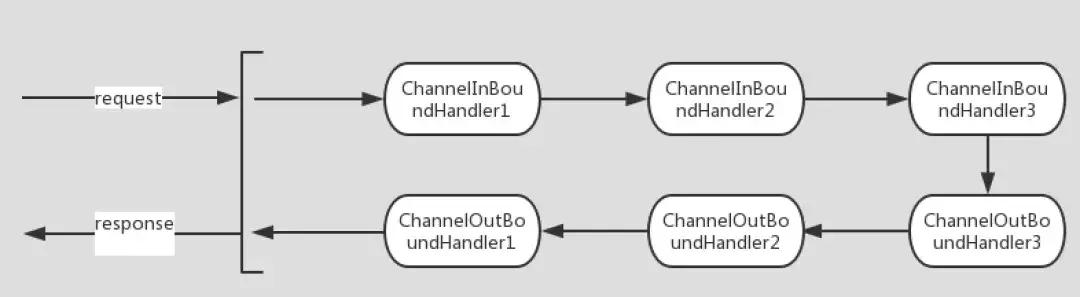


在Netty中关于ChannelHandler有两个重要的接口，ChannelInBoundHandler和ChannelOutBoundHandler。inbound可以理解为网络数据从外部流向系统内部，而outbound可以理解为网络数据从系统内部流向系统外部。

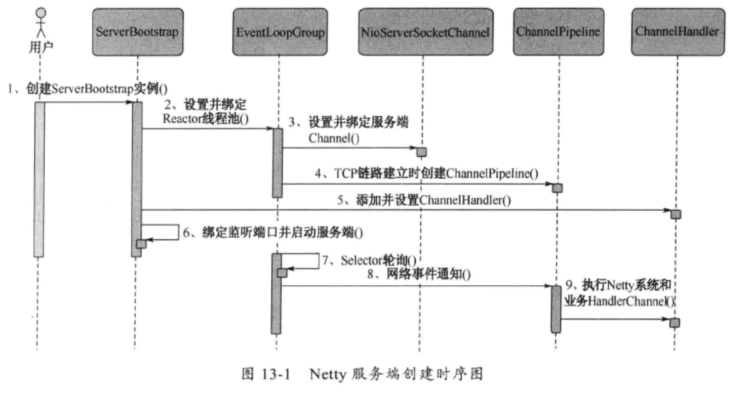
用户实现的ChannelHandler可以根据需要实现其中一个或多个接口，将其放入Pipeline中的链表队列中，ChannelPipeline会根据不同的IO事件类型来找到相应的Handler来处理。同时链表队列是责任链模式的一种变种，自上而下或自下而上所有满足事件关联的Handler都会对事件进行处理。

ChannelInBoundHandler对从客户端发往服务器的报文进行处理，一般用来执行半包/粘包，解码，读取数据，业务处理等；

ChannelOutBoundHandler对从服务器发往客户端的报文进行处理，一般用来进行编码，发送报文到客户端。



### 1.3 Netty 内部执行流程



1、创建ServerBootStrap实例

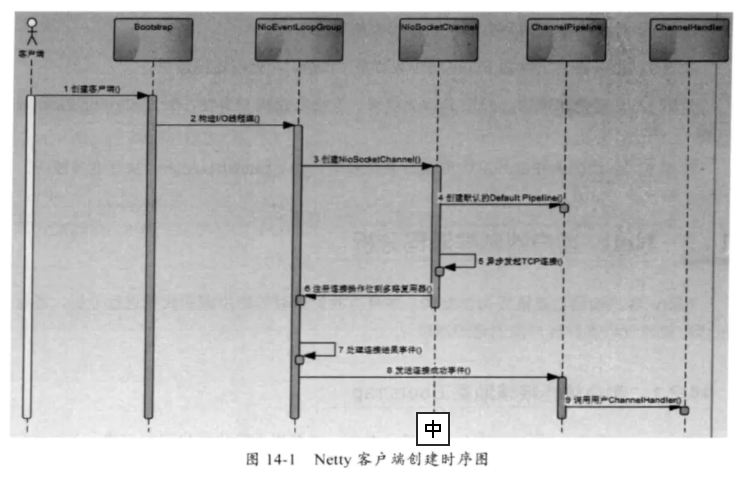
2、设置并绑定Reactor线程池：EventLoopGroup，EventLoop就是处理所有注册到本线程的Selector上面的Channel

3、设置并绑定服务端的channel

4、5、创建处理网络事件的ChannelPipeline和handler，网络时间以流的形式在其中流转，handler完成多数的功能定制：比如编解码 SSl安全认证

6、绑定并启动监听端口

7、当轮训到准备就绪的channel后，由Reactor线程：NioEventLoop执行pipline中的方法，最终调度并执行channelHandler。



## 2 HTTP

HttpServer启动HTTP服务器，通过HttpConf完成参数的配置，HttpRouter是http请求的路由器，将http请求分发给Action去响应。

HttpServer接收到客户端的连接请求，会启动一个新的HttpHandler。当HttpHandler接收到客户端的数据请求时，会根据请求的URL找HttpRouter查询相应的Action去处理。

### 2.1 HttpHandler

httpHandler继承自SimpleChannelInboundHandler，如果请求的是静态资源，则直接将资源返回给客户端。可以通过HttpConf设置displayDirs或downloadDirs，其中displayDirs下的资源是返回给客户端，让客户端直接显示的；downloadDirs是返回给客户端，让客户端下载的。

否则当普通的HTTP请求处理。目前HttpMethod支持GET和POST，ContentType支持application/json、application/x-www-form-urlencoded和multipart/form-data。会自动将所有的参数（包括URL路径中的值、URL中的参数、Form提交的数据）全部解析出来，然后根据需要转换为Action对象的函数参数，并调用该函数进行处理，最后将结果写会给客户端。



### 2.2 HttpRouter

Http路由器负责将Action和Http请求关联起来。这样Http请求才能用相应的Action来处理。初始化时，http路由器会自动注册路径下面的所有实现了注解的Action，并自动扫描Action里面所有的Request注解的方法，用来处理Http的Request请求。



### 2.3 参数解析

参数解析包括两部分:从HTTP请求中解析参数、将解析到的参数映射到Action函数参数。

由于Http请求携带的参数可能有多个地方、多种方式，所以用解码链的方式去完成。目前负责解码的有下面几类：



UrlDecoder将URL请求路径中解析参数，比如user/login?account=mjk&pwrd=test，这时UrlDecoder解析到的是Map，其中两组key和value分别是account=mjk和pwrd=test。如果Content-Type = application/json ，则用JsonDecoder是解析json参数。如果是Content-Type = x-www-form-urlencoded或multipart/form-data，则用FormDecoder解析from表单数据或上传文件。

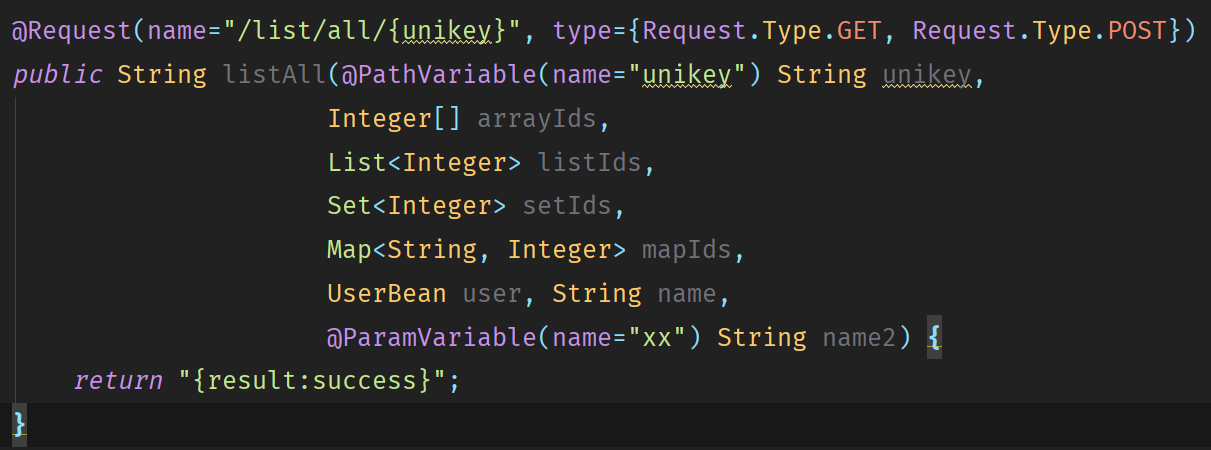
Encoder负责将Decoder解析到的参数转换为Action函数调用参数，然后再调用函数。Encoder能转换的参数类型有：基本类型、数组、List、Map、Set、Bean、FileUpload。其中基本数据类型有Boolean 、String、Byte、Char、Integer、Short、Long、Float、Double、BigInteger、BigDecimal。Bean是对象传递，不过需要提供set、get函数。FileUpload是文件上传时，用来接收文件的（接收文件的保存路径，可以通过HttpConf的setUploadDir来修改。

### 2.4 Action

Action是真正响应Http请求的地方，用户需要自行编写。Action的类需要添加Action注解，响应函数需要添加Request注解。类似下面的情况：



这样就可以响应http的post请求了：/user/login/finger或/user/login/account。这里需要说明的是参数type是路径变量，通过PathVariable注解声明，表示这个变量是从路径{type:finger|account}来指定的，只有这两个值才会用这个函数来响应。



这里listAll函数同时响应GET和POST的请求，路径变量unikey没有指定值，所以可以是非空的任意值。这个函数模拟了如何接收数组、List、Set、Map、Bean和普通类型，如果响应的变量不想跟http请求的变量名一样，可以通过ParamVariable来声明参数变量名。

这里需要说明的是文件上传和下载。文件上传，通过参数变量类型FileUpload来接收；文件下载，通过返回File类型来自动将文件二进制流回写给客户端（这里不用考虑零拷贝和分片处理，因为框架已经实现了）。



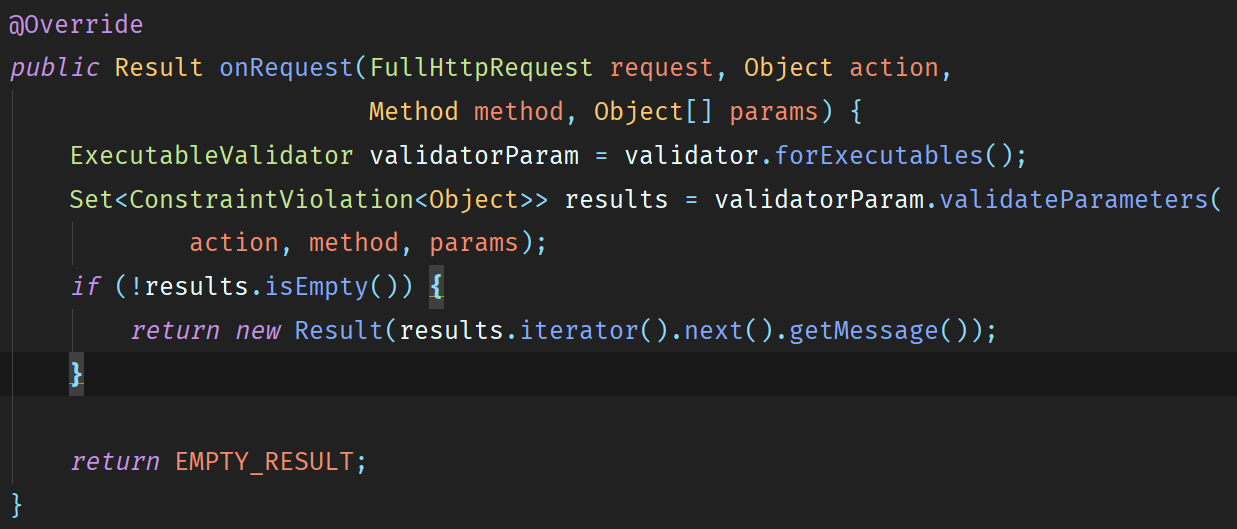
注意：返回的数据默认会以Json的形式回写给客户端。但通过Response注解，可以指定返回的类型。支持JSON、TEXT、XML和HTML四种类型。

### 2.5 过滤器

http还提供过滤器HttpFilter来过滤请求，返回的Result为空则继续后面的请求，否则根据Result直接返回给客户端。HttpFilter的使用可以是参数校验、权限验证等。其中参数校验提供了HttpValidFilter实现，大家可以参考来实现。这里是基于hibernate-validator实现的，大家可以替换成其他的。

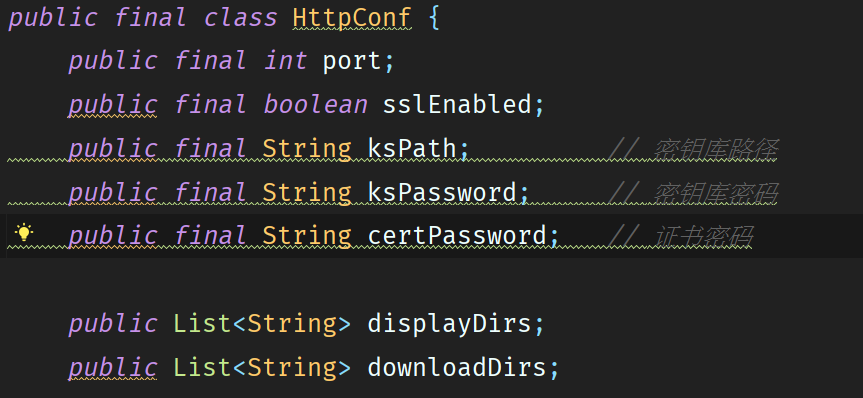
过滤器的注册：过滤器HttpFilter的注册是通过HttpServer的registFilter来完成的。所有的http请求都会执行所有的过滤，并没有区别对待。如果有特殊需求，在过滤器里面自行实现。

过滤器的接口：HttpFilter提供了两个接口，onDownload是下载文件前会执行，因为可能只有某些人才有现在权限；onRequest是http请求前会执行，如果返回非空的Result，则直接将结果回写给客户端。这两个接口如果返回EMPTY\_RESULT，则不执行任何操作。onResult的参数比较多，这里解释一下：action是负责响应的Action对象，Method是响应的Action函数，params是Action函数的所有请求参数。



### 2.6 SSL的支持

如果HttpConf配置了SSL证书参数，则需要用HTTPS来访问。SSL证书的生成可以利用Java工具Keytool来完成。HttpConf中涉及到SSL加密的参数有是否启动SSL加密sslEnabled、密钥库路径ksPath、密钥库密码ksPassword、证书密码certPassword。



HttpServer启动后，如果有HTTP连接请求后，会在Channel链的头部增加SslHandler来负责加解密。这里提供了两种SslFactory：AuthFactory（它生成的SSLEngine需要认证）和TrustFactory（它生成的SSLEngine不需要认证，信任所有的证书）。服务器必须用AuthFactory，客户端可以用AuthFactory或TrustFactory。



### 2.7 服务器的示例

1 编写Action

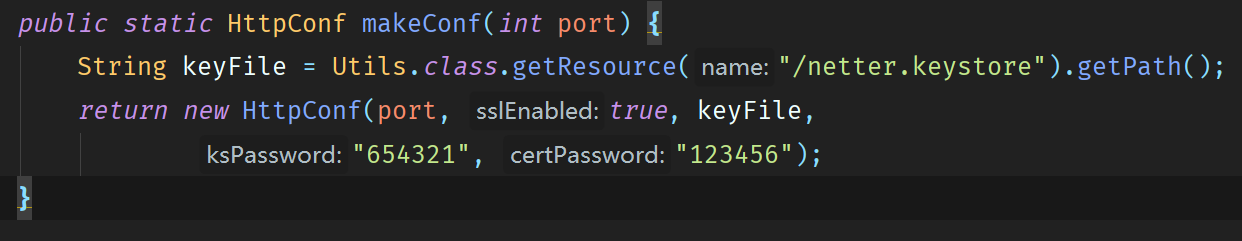
定义的类（必须UserAction）需要用注解Action来声明。不能存在相同的Action名，同一个Action下面不能存在相同的Request名。否则会抛出异常。



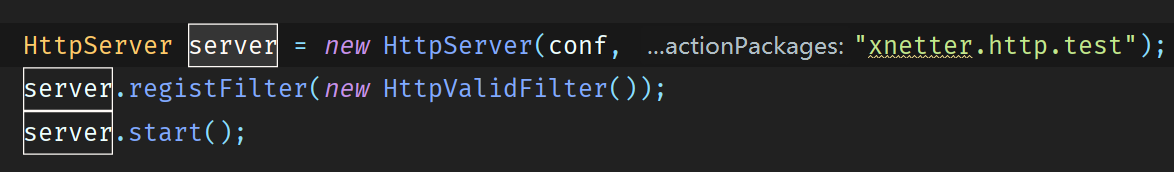
2 构建HttpConf



如果想支持SSL，需要构造不一样的HttpConf，比如用下面的方式：



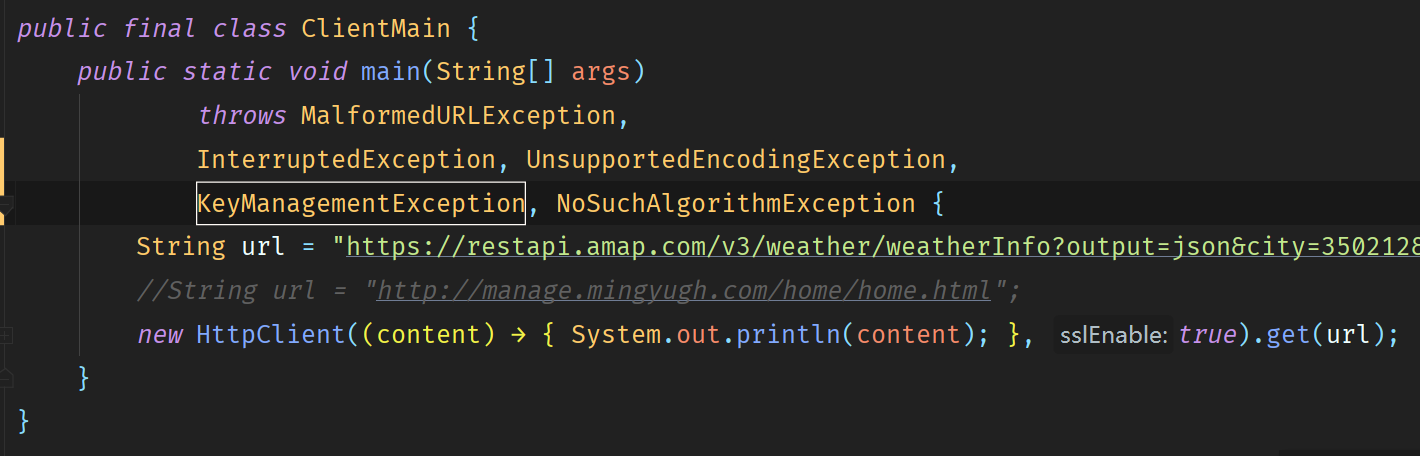
3 启动服务



### 2.8 客户端的示例

HttpClient的启动非常简单，但每一个HttpClient需要实现Handler来接收服务器的响应。HttpClient的优势是，如果多个地方调用相同的服务器请求，不会重连，会重用之前的连接。这样就省去了三次握手和四次挥手通信。

HttpClient提供了get、post和request方法。其中request是普通的http请求，可以满足所有的客户端http请求。



## 3 Websocket

### 3.1 建立连接

Websocket是一种特殊的http，建立连接时request的headers会包含Upgrade=websocket这样的键值对，表明浏览器希望与服务器进行websocket通信。这时服务器需要启动握手协议，成功后会根据URL继续调用Action（这里的Action需要实现接口WSockAction）的方法，这个方法可以认为是连接初始化的方法，可以完成用户的认证之类的操作。

连接建立后，后面的通信处理就交给WSockHandler了，不需要HttpHandler了。需要需要把WSockHandler动态添加到处理链里面。



### 3.1 WSockAction

WSockAction接口提供了下面方法，onConnect是建立连接的时候会调用，onClose是断开连接会调用，onRecv系列函数是收到消息时会调用，其中Close、Ping消息可以不用关心，用户主要处理Text和Binary消息就可以了。

*void* onConnect(WSockHandler handler);

*void* onClose(WSockHandler handler);

*void* onRecv(WSockHandler handler, CloseWebSocketFrame frame);

*void* onRecv(WSockHandler handler, PingWebSocketFrame frame);

*void* onRecv(WSockHandler handler, TextWebSocketFrame frame);

*void* onRecv(WSockHandler handler, BinaryWebSocketFrame frame);

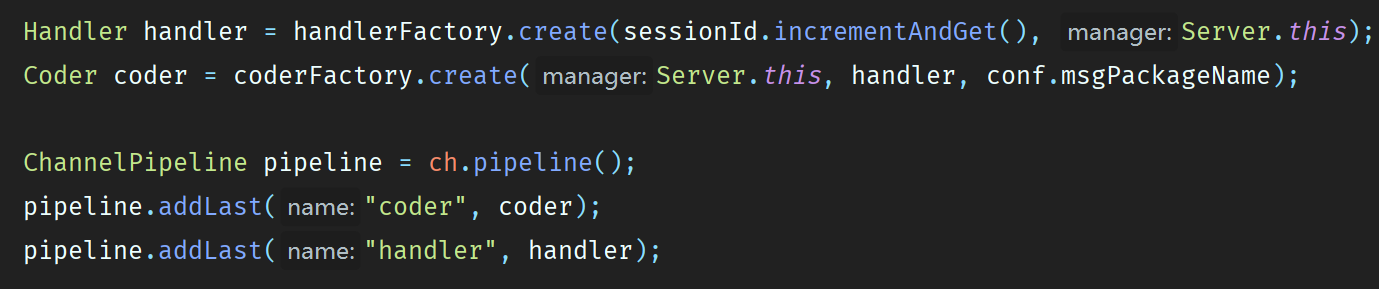
## 4 TCP

TCP通信有几个组件，分别是Coder负责加解密和编解码、Handler负责收发名明文数据（已经通过解码成通信对象）、Dispatcher将通信对象转给相应的Action来处理、Action用户需要实现的逻辑业务处理。



### 4.1 Coder

连接建立时，无论是服务器还是客户端，都会构建一个Coder对象放到处理链中。

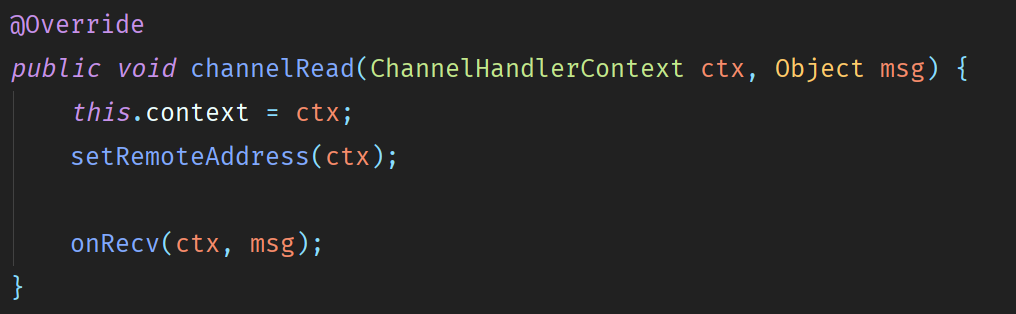


Coder会自动扫描包路径下面的所有通信对象，这样收到通信数据时，才能根据数据构造通信对象。还提供setInSecurity和setOutSecurity来提供加解密算法，这里只实现了RC4Security，其他算法后面再完善。



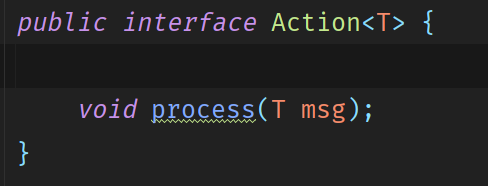
### 4.2 Handler

建立连接时，也会构建Handler放到处理链上。每个连接对应一个Handler实例。Handler继承自ChannelDuplexHandler，提供send方法来发送数据。如果收到数据，会调用抽象方法onRecv，只不过这里的参数msg已经是经过解码的通信对象了。

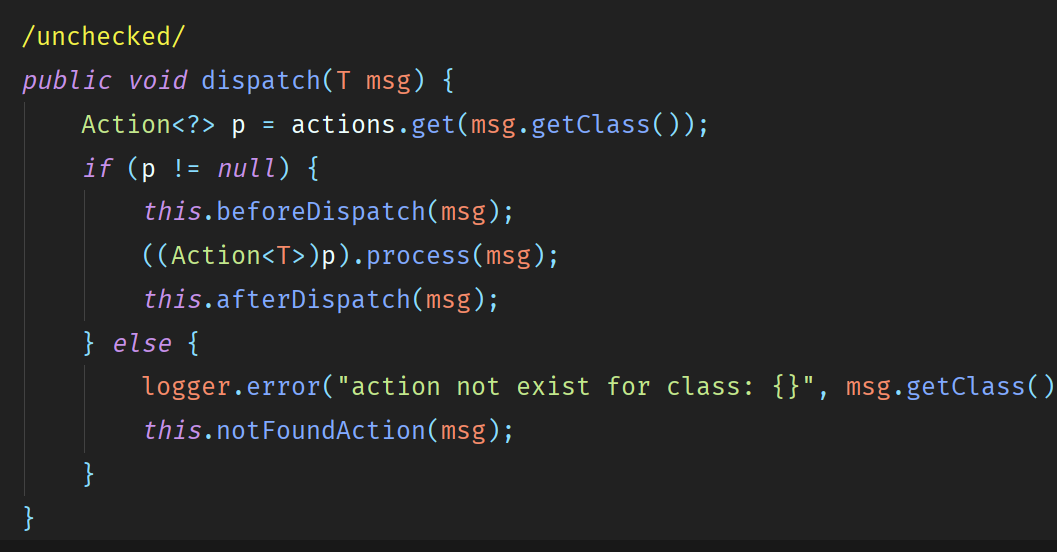


### 4.3 Dispatcher

Dispatcher是通信对象分发器，初始化时会注册通信对象和Action业务处理。



Dispatcher维护者通信对象和Action之间的关系，收到通信对象，就会调用Action去处理。但是为了用户扩展，在处理前后会调用beforeDispatch和afterDispatch函数。这是Dispatcher的具体实现类来实现的。



### 4.4 Manager

Manager需要提供配置文件Conf、Dispatcher、coderFactory、handlerFactory。建立连接的时候会利用coderFactory构建Coder对象，用handlerFactory构建Handler对象。



Conf配置参数如下，其中msgPackageName是通信消息的包路径，actionPackageName是Action的包路径。如果keepAlive为true，则服务器和客户端会保持心跳。心跳发送间隔为sendInterval，超过expireTime还没有心跳则断开连接。客户端需要配置是否需要断线重连，如果reconnect为true则断线重连。如果没有什么特别的，配置参数保持默认值就可以了。

*public final* String ip;  
 *public final int* port;  
 *// 定义客户端和服务器通信消息的路径  
 public final* String msgPackageName;  
 *// 定义Action的路径，Action是处理消息的业务逻辑  
 public final* String actionPackageName;  
  
 *// 普通加解密算法  
 public* Security inSecurity;  
 *public* Security outSecurity;  
  
 *// SSL加解密算法  
 public boolean* sslEnabled;  
 *public* String ksPath;  
 *public* String ksPassword;  
 *// 服务器需要设置，客户端保持为空  
 public* String certPassword;  
  
 *public boolean* tcpNoDelay = *false*;  
 *public int* backlog = 1000;  
 *public boolean* keepAlive = *true*;  
 *// 心跳包发送间隔  
 public int* sendInterval = 10; *// unit: second  
// 超过这个时间还没收到心跳包，则关闭连接  
 public int* expireTime = 300; *// unit: second  
  
 public int* socketSendBuff = 16 \* 1024;  
 *public int* socketRecvBuff = 16 \* 1024;  
 *public int* maxMsgSize = 64 \* 1024;  
  
 *// 客户端是否断线重连  
 public boolean* reconnect = *true*; *// for client  
 public int* reconnectInterval = 1; *// second (for client)*

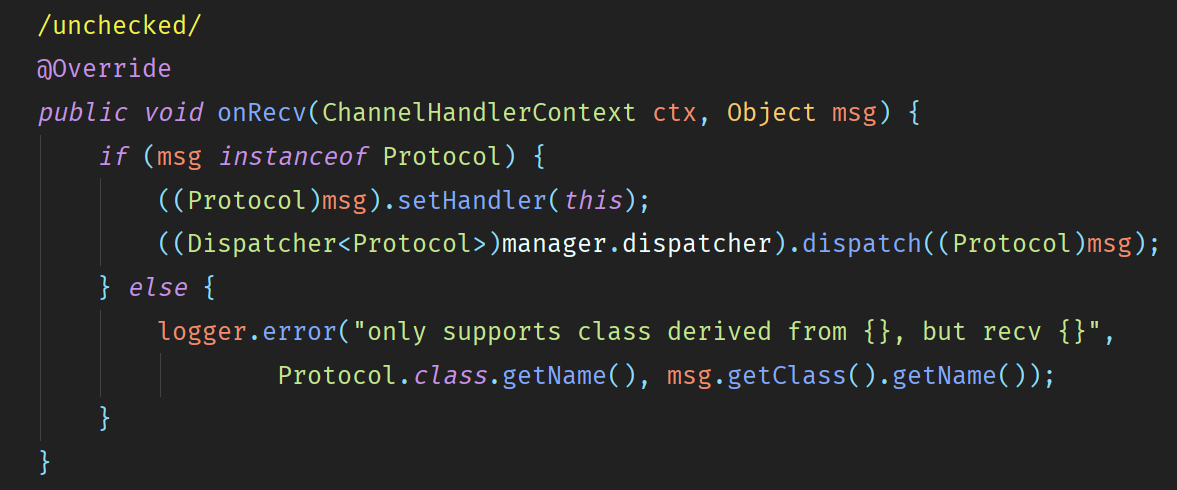
### 4.5 MultiClient

为了大家使用方便，提供了MultiClient类，它可以维护多个客户端，分别去连接不同的服务器。如果相应的服务器发生变化，可以通过方法updateClients去调整。但有一点需要记住的是：每一个客户端需要有一个remoteId来做标识，所以建立连接的时候，需要主动调用MultiClient的registClient方法去设置该Client的remoteId。



### 4.6 默认通信框架

为了大家的方便，提供了一套默认的通信框架的实现，为Protocol。大家也可以仿照Protocol来实现其他的通信数据框架。ProtocolCoder继承自Coder，维护所有的Protocol对象。ProtocolHandler继承自Handler，收到协议会调用其onRecv函数，最终调用ProtocolDispatcher来完成协议的转发和处理。



所以实现一套新的通信框架，需要定义通信对象（比如Protocol）、实现Coder（如ProtocolCoder）、实现Handler（如ProtocolHandler）和实现Dispatcher（如ProtocolDispatcher），当然还有业务处理的Action。



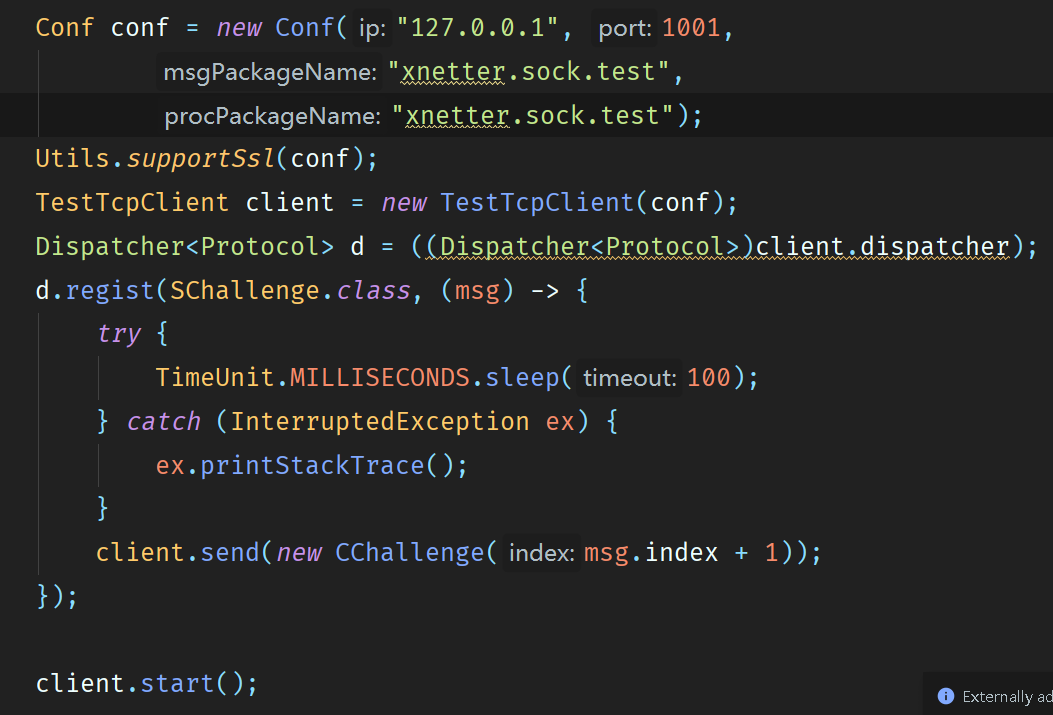
### 4.7 服务器的示例

TestTcpServer继承自Server，然后就可以启动了。太简单了。不过示例中没有使用自定义的加解密，而是使用了SSL加解密。PChallenge继承自Action，收到CChallenge后，返回SChallenge给客户端。对CChallenge处理的PChallenge是自动扫描注册的。



### 4.8 客户端的示例

TestTcpClient继承自Client，然后就可以启动了。这里没有实现Action来处理SChallenge，所以Dispatcher没有办法自动扫描和注册，只能通过接口regist来注册协议和处理函数。



## 5 UDP

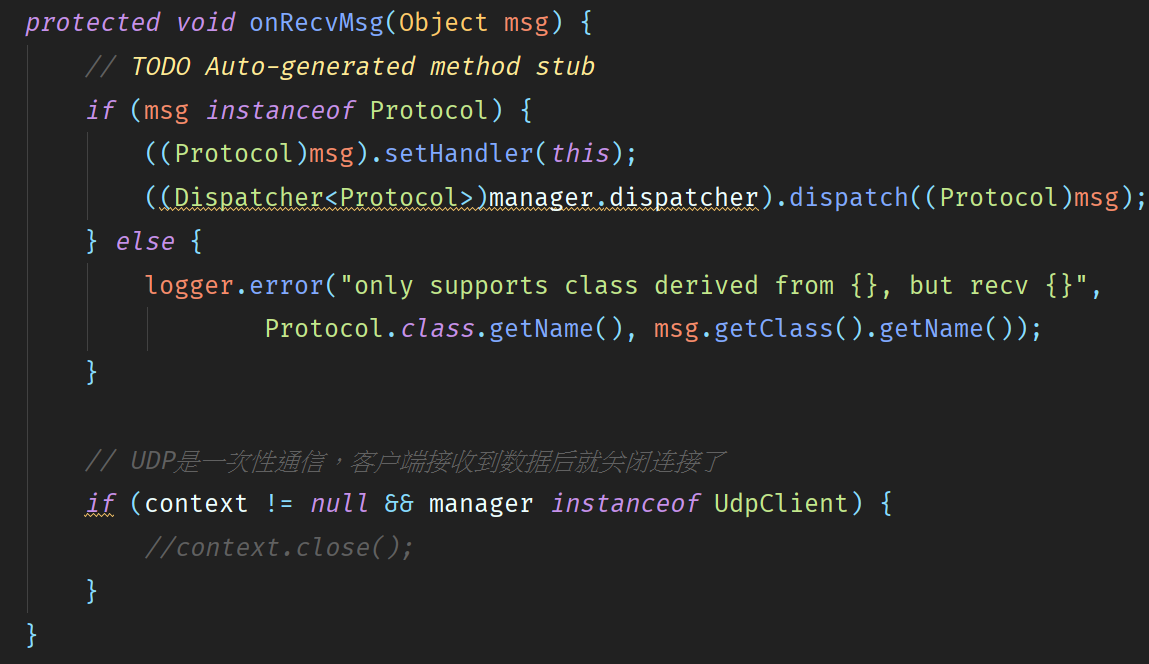
upd的实现流程跟tcp很类似。只不过服务器需要继承自UdpServer，客户端需要继承自UdpClient。udp是无连接的，客户端发送数据总是向指定的IP和端口发送，服务器启动单线程监听端口并接收数据。

### 5.1 Handler

客户端利用UdpHandler来收发数据，而为了提高服务器处理性能，用RecvHandler来接收数据，并从对象池handlers里面获取一个UdpHandler来处理数据。udp的处理器需要继承自UdpHandler，可以仿照ProtocolUdpHandler来写。注意：udp是无连接的，所以服务器发送数据只能原路返回。

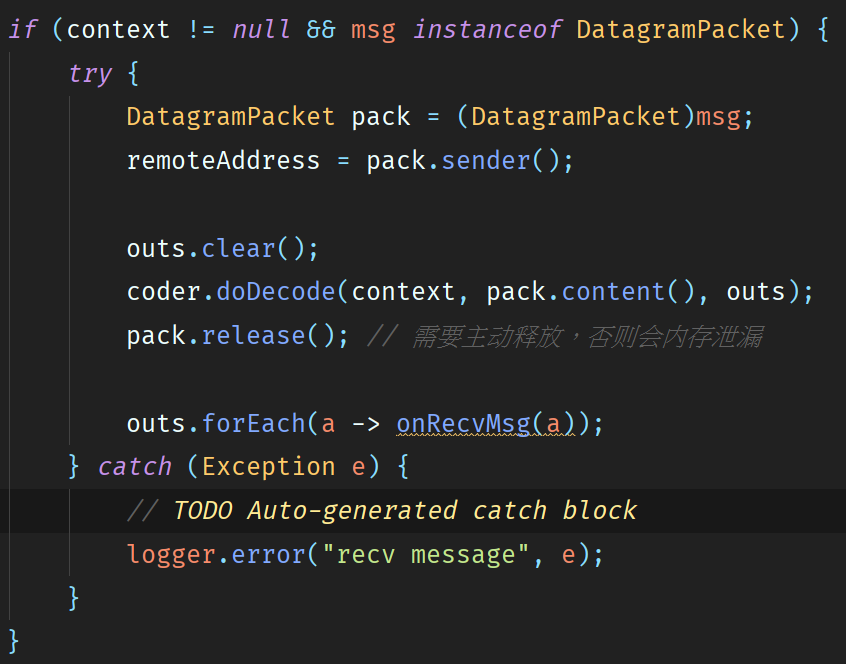


ProtocolUdpHandler实现onRecvMsg函数，主要就是调用dispatcher将协议转发给Action去处理。

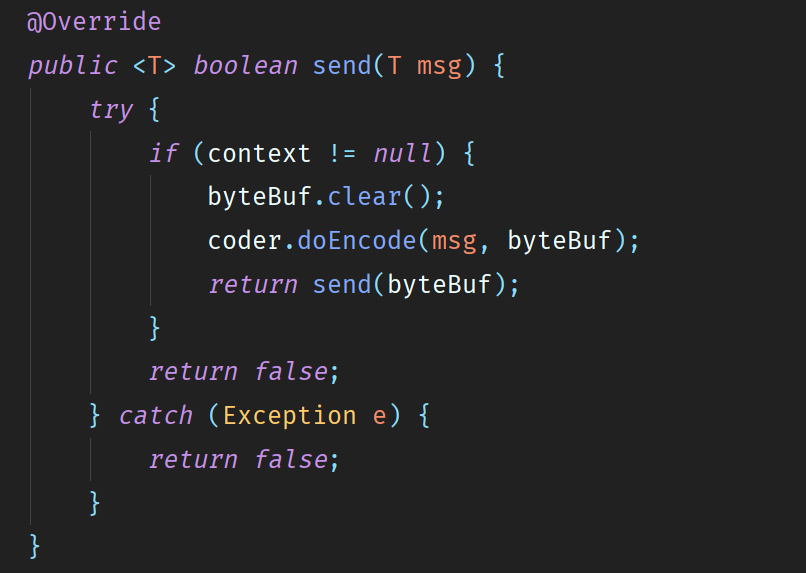


### 5.2 UDP加密

因为UDP收到的数据是DatagramPacket，包含了通信数据和对方的地址。所以要在UdpHandler里面收到数据后，再进行解密和解码。



发送数据的时候，需要先编码和加密后再封装成DatagramPacket发送对对方。注意UDP不支持SSL，所以只能使用自己实现的加解密算法。





### 5.3 服务器的示例

TestUdpServer继承自UdpServer，然后就可以启动了!



### 5.4 客户端的示例

TestUdpClient继承自UdpClient，然后就可以启动了!

