高性能NIO框架Netty入门篇 ☆

尹吉欢 · 2018-03-02 · 0条评论 · 6939人阅读

java (http://cxytiandi.com/article/search/java) netty (http://cxytiandi.com/article/search/netty)

Netty介绍

Netty是由JBOSS提供的一个java开源框架。Netty提供异步的、事件驱动的网络应用程序框架和工具，用以快速开发高性能、高可靠性的网络服务器和客户端程序。

也就是说，Netty 是一个基于NIO的客户、服务器端编程框架，使用Netty 可以确保你快速和简单的开发出一个网络应用，例如实现了某种协议的客户，服务端应用。Netty相当简化和流线化了网络应用的编程开发过程，例如，TCP和UDP的socket服务开发。

官网地址：http://netty.io/ (http://netty.io/)

使用场景

Netty之所以能成为主流的NIO框架，是因为它有下面的优点：

- NIO的类库和API使用难度较高，Netty进行了封装，容易上手
- 高性能，功能强大，支持多种编解码功能，支持多种主流协议
- 成熟，稳定，已经在多个大型框架中使用（dubbo，RocketMQ，Hadoop，mycat，Spring5）
- .....

在几年之前我上家公司用的是Mina来开发一个IM的系统，Mina也是一个很好的框架（http://mina.apache.org/）。(http://mina.apache.org/）。）
如今很多的框架都改成用Netty来做底层通讯了，我司现在还有一个代理框架用Mina写的，等把Netty玩遛了可以重构了。

不知道大家看完了上面的介绍是不是已经知道Netty能用在什么场景了，下面我结合一个我之前做过的事情来进行详细的说明，当然只是使用场景的一方面而已。

之前做抓取的时候，有一些小型的网站，页面结构比较复杂，还需要登录等操作，这种就不能用统一的抓取系统去抓取，只能通过写脚本的方式针对具体的网站做抓取，抓取必备的一个条件就是代理IP，为了方便抓取，特意封装了一个抓取的SDK,提供了抓取的方法，内置了切换代理。

我们有一个代理池服务，通过一个网址去获取能使用的代理IP，在刚开始用的Http请求去获取代理IP,由于抓取量比较大，通过Http请求去获取代理IP效率不行，后面用Netty改造了获取IP这部分，通过Netty来获取数据，解决了实时获取的性能问题。

通过长连接的方式，避免了Http请求每次都要建立连接带来的性能消耗问题，通过二进制的数据传输减少网络开销，性能更高。

简单入门

我们编写一个服务端和客户端，客户端往服务端发送一条消息，消息传输先用字符串进行传递，服务端收到客户端发送的消息，然后回复一条消息。

首先编写服务端代码：

```
1. public class ImServer {
2. 
3.     public void run(int port) {
4.         EventLoopGroup bossGroup = new NioEventLoopGroup();
5.         EventLoopGroup workerGroup = new NioEventLoopGroup();
6. 
7.         ServerBootstrap bootstrap = new ServerBootstrap();
8.         bootstrap.group(bossGroup, workerGroup)
9.                 .channel(NioServerSocketChannel.class)
10.                .childHandler(new ChannelInitializer<SocketChannel>() {
11.                    @Override
12.                    public void initChannel(SocketChannel ch) throws Exception {
13.                        ch.pipeline().addLast("decoder", new StringDecoder());
14.                        ch.pipeline().addLast("encoder", new StringEncoder());
15.                        ch.pipeline().addLast(new ServerStringHandler());
16.                    }
17.                })
18.                .option(ChannelOption.SO_BACKLOG, 128)
19.                .childOption(ChannelOption.SO_KEEPALIVE, true);
20. 
21.         try {
22.             ChannelFuture f = bootstrap.bind(port).sync();
23.              f.channel().closeFuture().sync();
24.         } catch (InterruptedException e) {
25.             e.printStackTrace();
26.         } finally {
27.             workerGroup.shutdownGracefully();
28.             bossGroup.shutdownGracefully();
29.         }
30.     }
31. 
32. }
```

- 通过ServerBootstrap 进行服务的配置，和socket的参数可以通过ServerBootstrap进行设置。

- 通过group方法关联了两个线程组，NioEventLoopGroup是用来处理I/O操作的线程池，第一个称为“boss”，用来accept客户端连接，第二个称为“worker”，处理客户端数据的读写操作。当然你也可以只用一个NioEventLoopGroup同时来处理连接和读写，bootstrap.group()方法支持一个参数。
- channel指定NIO方式
- childHandler用来配置具体的数据处理方式，可以指定编解码器，处理数据的Handler
- 绑定端口启动服务

消息处理：

```
/**
 * 消息处理
 * @author yinjihuan
 *
 */
public class ServerStringHandler extends ChannelInboundHandlerAdapter {

    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) {
        System.err.println("server:" + msg.toString());
        ctx.writeAndFlush(msg.toString() + "你好");
    }

    @Override
    public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) {
        cause.printStackTrace();
        ctx.close();
    }

}
```

启动服务,指定端口为2222：

```
public static void main(String[] args) {
    int port = 2222;
    new Thread(() -> {
        new ImServer().run(port);
    }).start();
}
```

编写客户端连接逻辑：

```java
public class ImConnection {

    private Channel channel;

    public Channel connect(String host, int port) {
        doConnect(host, port);
        return this.channel;
    }

    private void doConnect(String host, int port) {
        EventLoopGroup workerGroup = new NioEventLoopGroup();
        try {
            Bootstrap b = new Bootstrap();
            b.group(workerGroup);
            b.channel(NioSocketChannel.class);
            b.option(ChannelOption.SO_KEEPALIVE, true);
            b.handler(new ChannelInitializer<SocketChannel>() {
                @Override
                public void initChannel(SocketChannel ch) throws Exception {
                    ch.pipeline().addLast("decoder", new StringDecoder());
                    ch.pipeline().addLast("encoder", new StringEncoder());
                    ch.pipeline().addLast(new ClientStringHandler());
                }
            });

            ChannelFuture f = b.connect(host, port).sync();
            channel = f.channel();
        } catch(Exception e) {
            e.printStackTrace();
        }
    }

}
```

客户端消息处理：

```
/**
 * 当编解码器为字符串时用来接收数据
 * @author yinjihuan
 *
 */
public class ClientStringHandler extends ChannelInboundHandlerAdapter {

    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) {
        System.out.println("client:" + msg.toString());
    }

    @Override
    public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) {
        cause.printStackTrace();
        ctx.close();
    }

}
```

客户端启动入口，然后发送消息给服务端：

```
public static void main(String[] args) {
    String host = "127.0.0.1";
    int port = 2222;
    Channel channel = new ImConnection().connect(host, port);
    channel.writeAndFlush("yinjihuan");
}
```

测试步骤如下：

1. 首先启动服务端
2. 启动客户端，发送消息
3. 服务端收到消息，控制台有输出 `server:yinjihuan`
4. 客户端收到服务端回复的消息，控制台有输出 `client:yinjihuan你好`

源码参考：https://github.com/yinjihuan/netty-im (https://github.com/yinjihuan/netty-im)

欢迎加入我的知识星球，一起交流技术，免费学习猿天地的课程（http://cxytiandi.com/course） (http://cxytiandi.com/course）)

PS：目前星球中正在星主的带领下组队学习Sentinel，等你哦！