## Netty

* 一个提供了易于使用的API的客户端/服务器框架
* 并发高 – NIO（非阻塞IO）
* 传输快 – 零拷贝（java读取IO流会先读取到缓存区，再读取到堆中，经历了2次拷贝；而netty读取数据是直接开辟新内存，IO流数据直接读取到该内存中。）

基本概念

阻塞与非阻塞

线程访问资源，该资源是否准备就绪的一种处理方式。

同步和异步

访问数据的一种机制

同步是请求数据后等待通知；异步是请求数据后处理其他的任务，数据返回后进行异步的通知。

* IO
* 同步阻塞IO/BIO （Block IO）进行IO读写时线程阻塞，无法执行其他操作。（比如上厕所，一直等待，并观察，直到某一个坑位空闲）
* 同步非阻塞IO/NIO，New IO（Non-Block IO）（比如，上厕所坑位满时，转去做一些其他事，同时时不时地主动去观察有没有坑位释放。）
* 异步非阻塞IO / AIO （厕所坑位满时干其他事情，有用户结束使用时来通知存在空闲坑位。）非常高效。
* 异步阻塞IO （厕所坑位满时，在厕所等待（很傻），让结束使用的用户来通知坑位空闲），这种方式基本不用。
* BIO：发起请求，一直阻塞，处理完成
* NIO：selector主动轮训channel，处理请求，处理完成。
* AIO：发起请求，通知回调。
* 常见面试
* 三者区别
* 什么是BIO，NIO，AIO

Netty

Reactor线程模型（netty的几种线程模型）：

* 单线程模型：所有IO操作都是用同一个NIO线程处理
* 多线程模型：由一组NIO线程处理IO操作（一个单线程仅负责进行请求监听，而读写请求都交给线程池来做
* 主从线程模型：一组线程池接受请求，一组线程池处理IO

Hello Netty服务器（经过实践，建立了一个简单的HelloNetty服务器，没有匹配url，返回一个字符串）

* 构建一对主从线程组
* 定义服务器启动类
* 为服务器设置channel（某一个客户端与服务器建立连接后，二者之间的双向通道就是channel）
* 设置处理从线程池的助手类初始化器（channel有对应的助手类对其进行处理，比如编解码，读写数据操作，都是归类在一个助手类中的，可以看做该channel的拦截器）（一个channel对应一个piplline，而pipline里包含多个助手类handler）
* 监听启动和关闭服务器。

Channel生命周期

实时通信三种方式 <https://blog.csdn.net/xiazeqiang2018/article/details/81189565>

1. Ajax轮询

原理是使用ajax方式让浏览器异步的每隔一段时间请求后端，询问有没有新消息。是一种死循环请求。

举例：本质上是定时发送ajax请求，注意async为True表示异步请求，不会锁死浏览器；setInterval 是定时执行函数。

1 <script>

2 setInterval("test()",500);

3 function test() {

4 $.ajax({

5 url: '/new\_window\_url/',

6 async:true,

7 type: 'get',

8 success: function (data) {

9 var new\_url = $('#new\_iframe').attr('src');

10 if (new\_url !== data){

11 $('#new\_iframe').attr('src', data);

12 }

13 }

14 })

15 }

16 </script>

1. Long pull（长轮询）

也是采用循环方式，只是采用阻塞的模式，当客户端发起请求后，如果服务器没有响应就一直等待，直到返回响应，客户端才会建立请求。

1. Websocket

以上两种方式比较耗资源。是一种协议，http不支持长连接，http1.1才支持keep-alive。

Websocket是一种持久化协议,http是非持久化的，http的生命周期由request来界定，有多少个request就对应多少个response。Websocket只需建立一次http请求，就源源不断的从服务器段接受数据，向客户端主动推送，就像打游戏，登录后就一直接受消息。

### Websocket API

Websocket协议，详见： <https://www.runoob.com/html/html5-websocket.html>

通过客户端和服务端有连接

Var socket = new WebSocket(“ws://[ip]:[port]”)

注意：ws是协议，这个路径相当于http://127.0.0.1:8000

生命周期：

onopen(), 客户端和服务端建立连接时触发

Onmessage() 收到消息时会触发，比如服务端主动向客户端发送了一些消息

Onerror() 后端出现异常时，前端的onerror里会接受到错误事件

Onclose() 连接关闭之后触发的函数。

主动的方法：

Socket.send() 在前端可以抓取用户输入的内容，放到send里发送到后端，后端netty就会接收到

Socket.close() 用户不想做任何的操作了，可以关闭客户端和服务端的连接。

### MUI H5+ Hbuilder

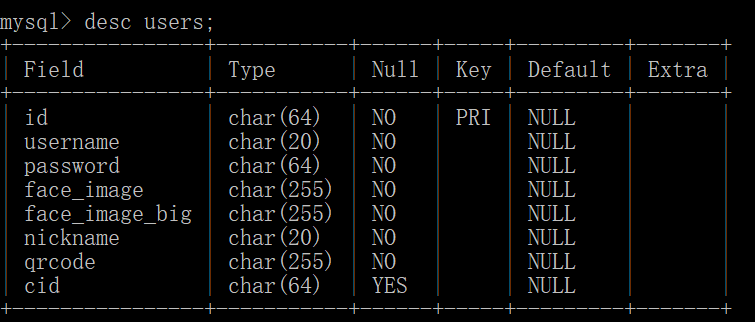
1. MUI <http://www.dcloud.io> 文档 <https://dev.dcloud.net.cn/mui/ui/> css相关
2. H5 PLUS (H5+) <http://www.html5plus.org/doc> js相关
3. Hbuilder 开发工具（ide）
4. [www.iconfont.cn](http://www.iconfont.cn) 阿里提供的图标库

Hbuilder做做调试时，电脑USB连接手机，手机打开打开USB调试。手机和电脑要在一个网络内，电脑连的无线，手机也要连上无线，否则无法请求到。

学习html css js ，或查询相关语法，就到这个网站：<https://www.runoob.com/js/js-json.html>

### 数据库

1. 库名 muxin
2. 表（共四张）
3. Uses表结构



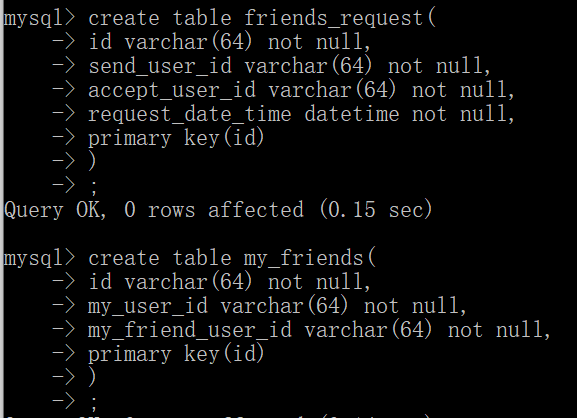
其中，face\_image一个是 正式头像，一个是缩略图

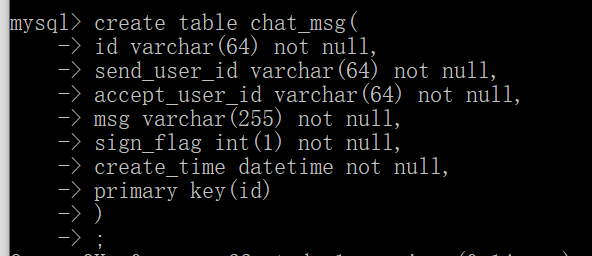
Nickname是昵称，开始设为username，可以由用户更改

Qrcode是为用户生成的个人二维码，扫描添加好友

cid是针对每一台手机设备，每一台手机设备都有其cid，这个cid可以通过plus获取，获取之后用于做用户推送。

1. 其他表





Sign\_flag是指签收状态，create\_time是消息发送时间

#### 使用mybatis做映射，映射数据库中的这些表。

项目使用mybatis逆向工具生成了许多xml文件并复制到了项目目录下。原理是什么？

答：项目使用了github上一个人的开发包，可以将数据库现有的表逆向生成xml文件和相应的类与接口。

**DataSource：数据源，也叫数据库连接池**

类似于线程池，里面存放了诸多与数据库建立好的连接，用户需要访问数据库时，直接从连接池获取连接并使用即可，使用后放回连接池，可以避免返回创建和撤销与数据库的连接。

连接池是由各数据库厂商实现的，另外阿里提供了一个水准较高的druid连接池。





### 文件上传

1. 普通文件上传

举例：一个app经过不断更新迭代可能会增加许多系统功能，每个系统功能又有各自对应的服务器，如果每个系统都把上传的文件都放在自己的服务器上，比如订单系统，用户系统，报表系统，那么用户在报表系统就无法获取到订单系统中上传的文件。

1. 使用文件服务器上传

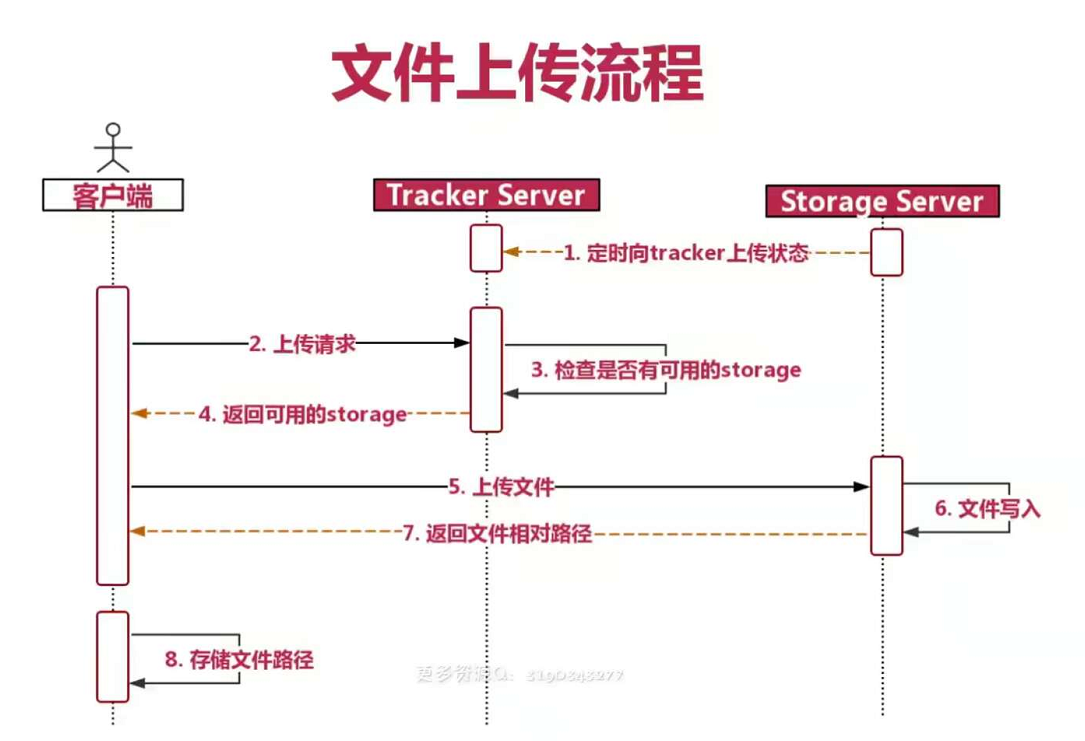
多个系统上传的文件都放到同一个文件服务器中。

文件服务器：

1. 第三方服务器，提供了一些接口，不需要人力维护。
2. 自己部署，需要对自己的文件进行管理和运维。

**FastDFS：**

C语言编写的开源分布式文件系统，充分考虑了冗余备份，负载均衡，线性扩容等机制，并注重高可用，高性能等指标。使用它很容易搭建一套高性能的文件服务器集群提供文件上传下载服务。



Tracker Server：相当于项目经理

Storage Server：相当于程序员

客户端拿到文件存储路径后可以保存到数据库里。

**FastDFS流程参考**：<https://www.cnblogs.com/scChen/p/11519648.html>

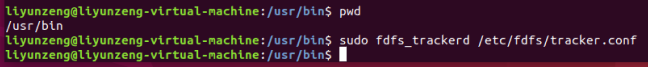
安装FastDFS文件服务器：

1. 准备一台linux
2. 文件准备：fastDFS： fastdfs-5.05.tar.gz，fastdfs-nginx-module\_v1.16.tar.gz, libfastcommon-1.0.7.tar.gz
3. nginx-1.12.0.tar.gz（没有这个版本，实际用的是1.12.2）
4. **我们需要使用FileZilla从我们的主机向服务器传送文件（需要服务器的ip，root，密码）。这个服务器可以是我们自己搭建的linux虚拟机，此时就填写虚拟机的ip和密码即可，也可以是云服务器，此时就填写云服务器的公网ip。**

**注意：公网ip是从任何位置都可以访问的，内网ip只能在局域网内互相ping通。**

**其实云服务器就相当于我们项目中搭建的初始的linux虚拟机，我们同样需要上传一些安装文件（比如fastDFS，nginx的安装包等，以及我们的java代码）到服务器上进行安装。我们操作云服务器就相当于操作自己的虚拟机一样，只不过是通过ssh工具（比如xshell）进行远程连接（需要云服务器ip和账户密码）。当然，远程连接的前提是服务器开启ssh服务（命令行输入service sshd start）（补充知识：ssh的默认端口是22）**

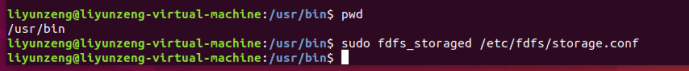
启动tracker服务器：



如果配置发生了变化，可以重新启动：

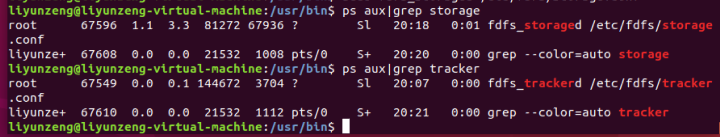


启动storage服务器：

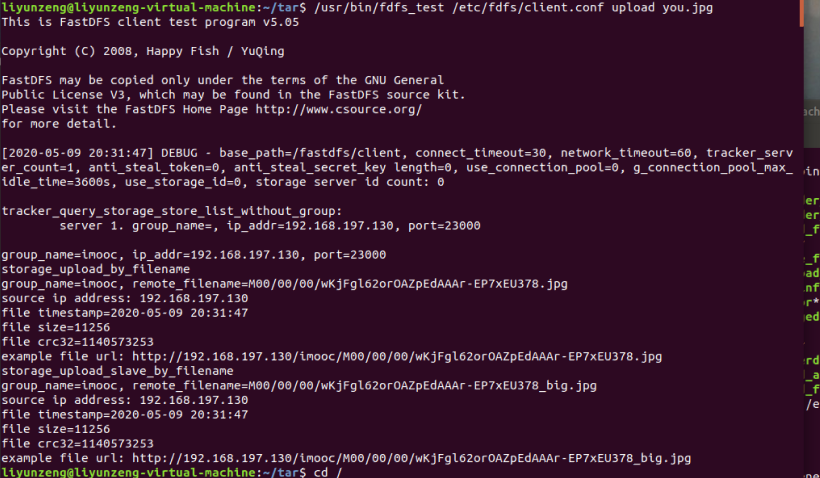


以上两个服务器的conf文件是可以自定义更改的，比如更改文件存储路径等信息。

查看二者是否已经启动：



使用自带的fdfs\_test对上传文件功能进行测试：



可以看到：最下面一行就是返回的文件存储地址，由<http://ip/group_name/remote_silename>组成。

到目前为止，我们使用fastDFS提供的一个客户端上传了一个文件，但是现在都是基于文件系统，并不能通过http去进行访问，所以借助nginx与fastDFS结合，这样就可以通过http在网页上访问相应的图片了。

 fastdfs-nginx-module 模块说明

　　FastDFS 通过 Tracker 服务器，将文件放在 Storage 服务器存储， 但是同组存储服务器之间需要进行文件复制， 有同步延迟的问题。

　　假设 Tracker 服务器将文件上传到了 192.168.51.128，上传成功后文件 ID已经返回给客户端。

此时 FastDFS 存储集群机制会将这个文件同步到同组存储 192.168.51.129，在文件还没有复制完成的情况下，客户端如果用这个文件 ID 在 192.168.51.129 上取文件,就会出现文件无法访问的错误。

Nginx+fastDfs已经可以通过http访问存储的文件，但是：

　　 fastdfs-nginx-module **可以重定向文件链接到源服务器取文件，避免客户端由于复制延迟导致的文件无法访问错误。**

**注意：以下内容中出现的192.168.197.130是linux虚拟机在nat模式下的ip地址，事实上，设置为nat模式的话，同一局域网内并不能直接使用ip地址访问虚拟机，尽管可以通过宿主机的端口映射，但是相当麻烦，因此作为一个服务器，应当设置为桥接模式。**

1. 解压fast-nginx-module，进入src，修改名为 config的文件，里面有三个路径，去掉其中的/local部分
2. 安装build-essential



再安装g++

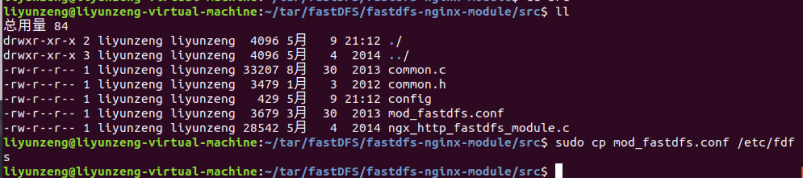


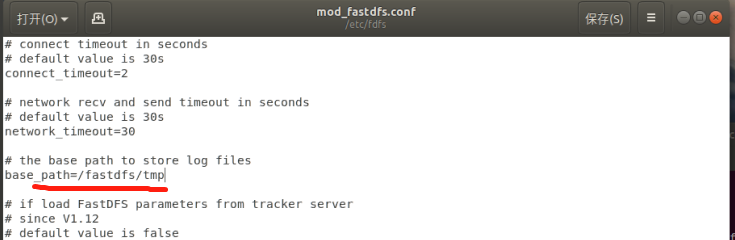
1. 安装nginx之前需要解决依赖包的安装。

Ubuntu下安装nginx时依赖库zlib pcre openssl安装方法：<https://blog.csdn.net/z920954494/article/details/52132125>

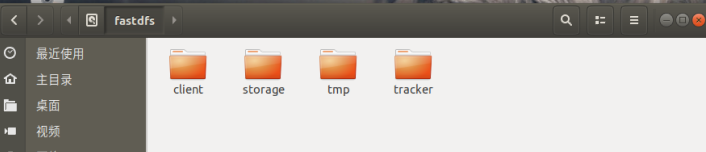
1. 配置nginx，解压nginx.1.12.2.tar.gz,进入解压文件夹（名为nginx-1.12.2）下，在命令行拷贝输入以下内容并回车：



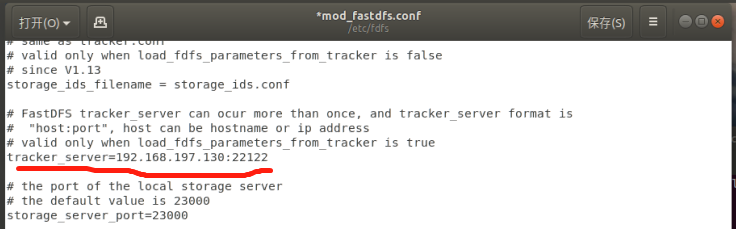
1. 输入make回车进行编译，再输入sudo make install ，回车，进行安装。
2. 安装完毕后在/usr/local下就会有个nginx文件夹。
3. 接下来：
4. 修改拷贝过来的这个文件：修改日志保存的路径

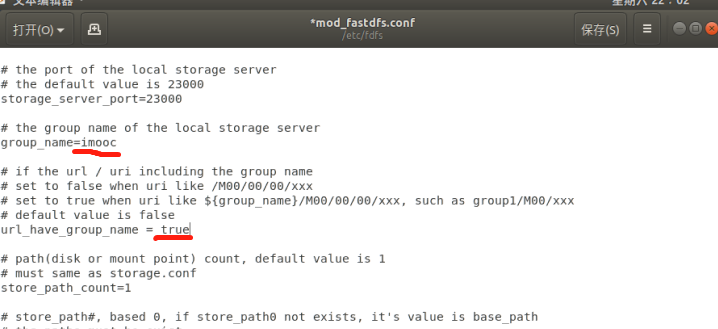


1. 在/fastdfs下创建tmp文件夹：

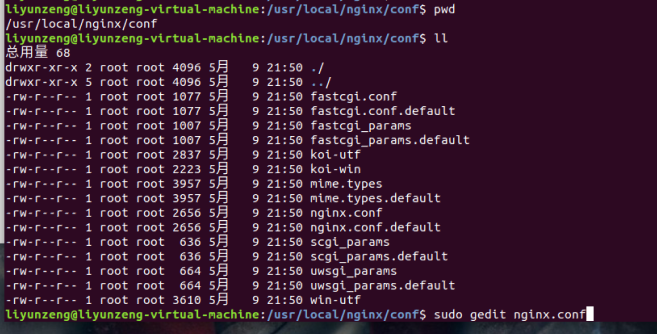


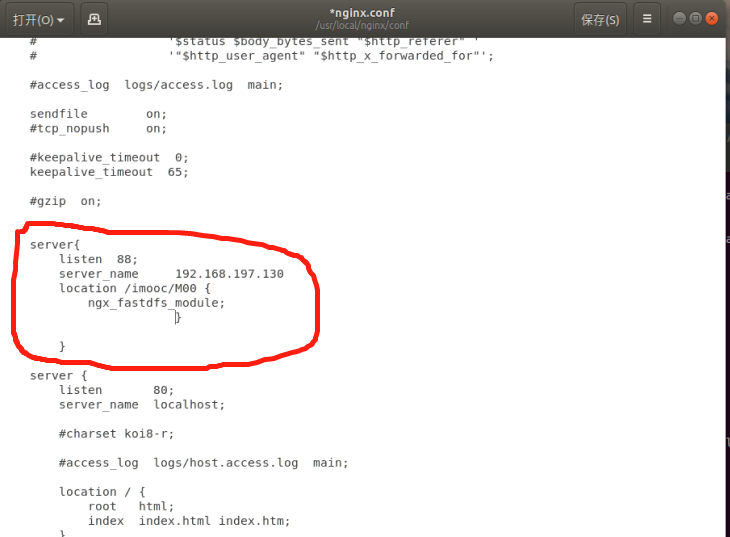
1. 继续修改mod\_fastdfs.conf文件：





1. 配置nginx：

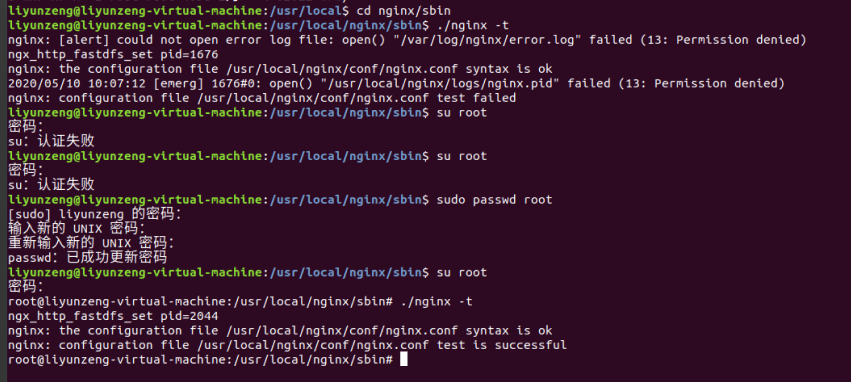




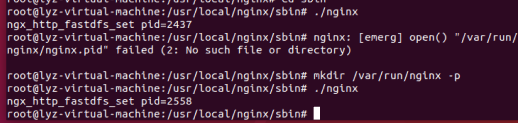
1. 上图的server\_name 192.168.197.130**最后应当加个分号！！！**
2. 进入/usr/local/nginx/sbin下运行 ./nginx -t (这是个测试运行的命令)
3. 如果nginx启动错误：



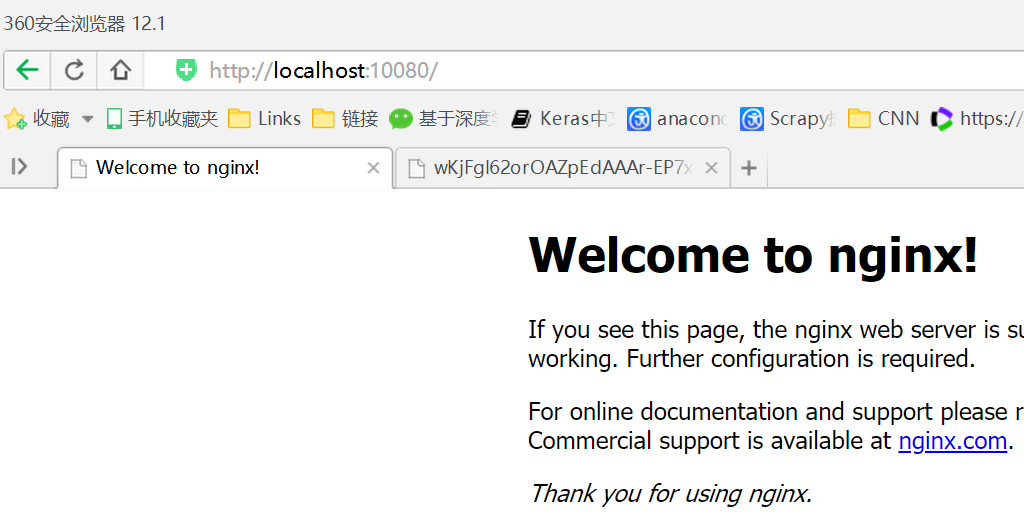
如果出现下图这样的错误就是权限不够，切换到root用户即可：（下图显示切换用户后已经测试成功）



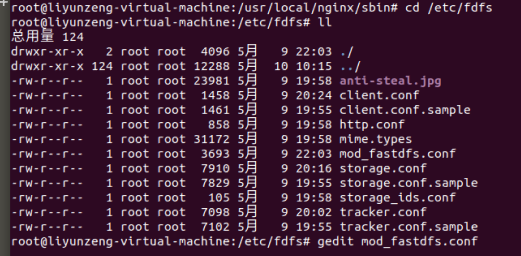
启动时可能出现没有某个目录，如下图，创建一个即可：



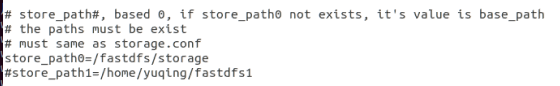
1. 在/usr/local/nginx/sbin/下输入 ./nginx 来启动，虚拟机的浏览器输入192.168.197.130（虚拟机ip）即可访问，默认端口是80 ./nginx –s reload 是重启
2. 外网（或者说宿主机）无法访问虚拟机的nginx启动页，宿主机上输入192.168.197.130不管用，可以先关闭**虚拟机的防火墙**，如果还是不管用，那就是因为虚拟机的网络使用的是nat模式，这是默认模式（共有桥接，host-only，nat三种模式），此时外网无法直接访问虚拟机，需要进行宿主机到虚拟机的端口映射，**端口映射设置方法及win10下防火墙设置**，详见<https://blog.csdn.net/Mrqiang9001/article/details/80820321>（注意不设置win10防火墙是无法使用端口映射的，很多博客都没提到这一点；端口映射就相当于把主机的端口和虚拟机的某个端口绑定，然后外网或主机访问主机的端口就等于访问了虚拟机的某个端口，绑定时需要提供一个主机空闲端口（随便设为10000以上即可），提供虚拟机ip（使用ifconfig在虚拟机上查询即可），以及虚拟机待绑定的端口）
3. 实际绑定的是主机端口10080与虚拟机的80，主机的10088与虚拟机的88.
4. 10080端口可以看到nginx的启动页：



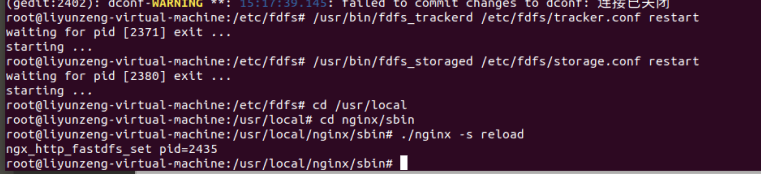
1. 修改mod\_fastdfs.conf：



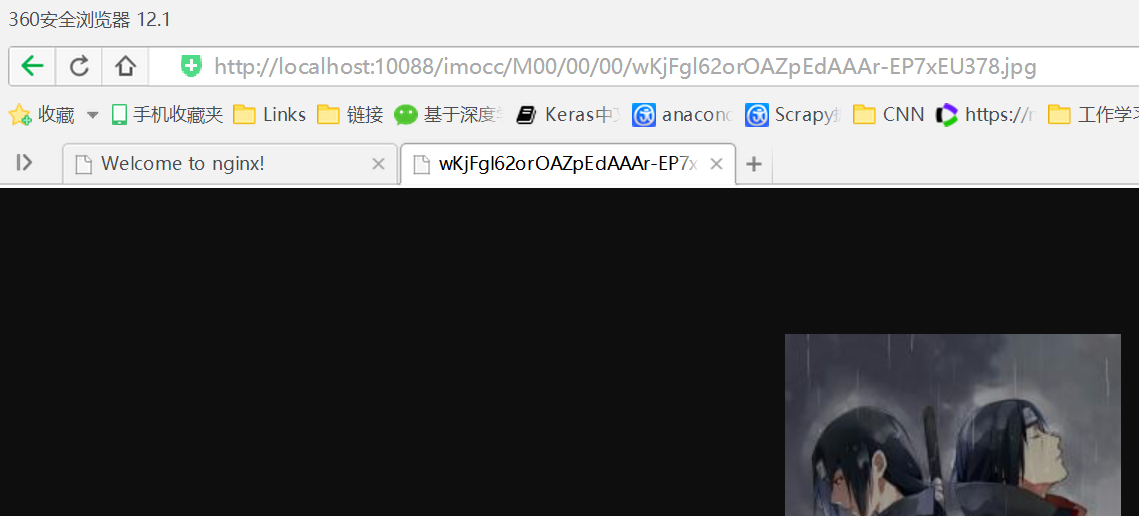
修改为下图：store\_path0设为和stotage.conf一样



修改后重启storage tracker nginx：



如图，可以在主机上访问到fastDFS服务器上的图片了（注意端口是10088，对应虚拟机的端口88.）



1. 到此为止，fastDFS服务器搭建完毕，接下来**整合进spring boot**。需要导入一个github上的一个jar包。

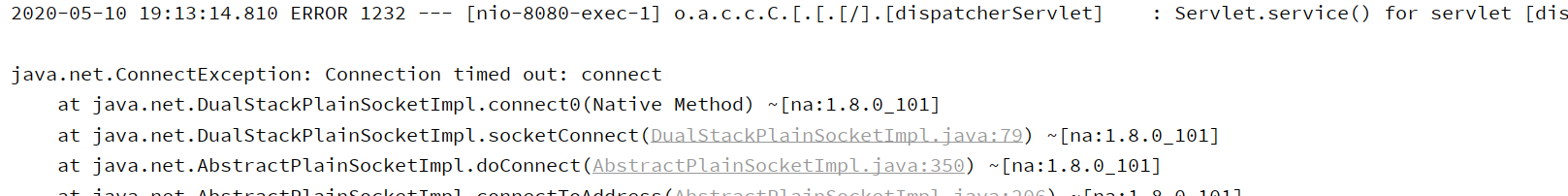
详见：<https://github.com/tobato/FastDFS_Client>

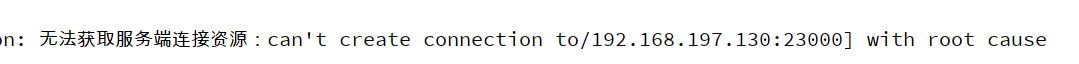
需要涉及到tracker服务器的端口，原端口是22122，映射为23001.

Application.properties文件：



上传文件报错：



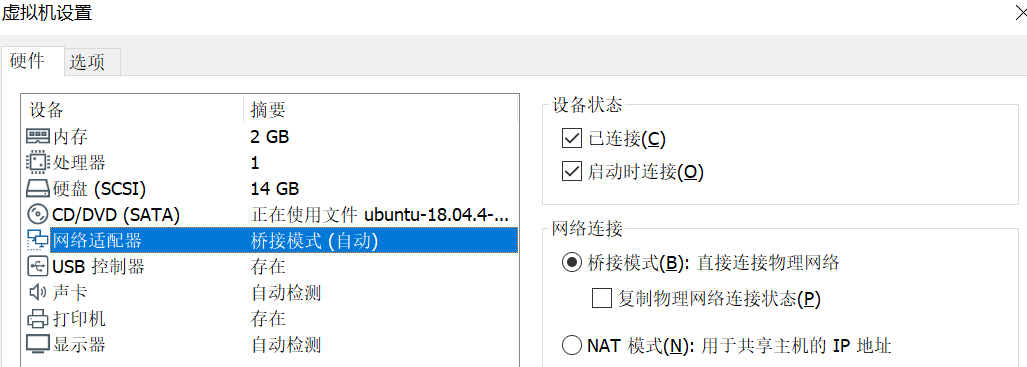


报错原因在于，上传文件的过程中其实调用了github上tobato提供的一个名为fastdfs-client的jar包，该包里面其实会自动抽取虚拟机ip和storage的端口号，即192.168.197.130：23000,并发起连接请求，但由于当前虚拟机是nat模式，无法直接ip连接，而是需要端口映射，因此会连接超时的问题，解决方法要么虚拟机的连接方式改为桥接，要么重写上传函数。

此时我重新建立了一个虚拟机，并设置为桥接模式，重新安装配置fastDFS和nginx。累死我了。。

。补充：如何改为桥接：

1. 在虚拟机的设置中设为设为桥接模式。



1. 虚拟网络编辑器中桥接模式的网卡设为主机主机使用的网卡，（本机使用的无线网卡 Wireless什么的）
2. 查看主机的ip，网关，DNS服务器，子网掩码等信息，（下图），修改ubuntu虚拟机的以上信息，ip自己设置（可以先用ifconfig查看虚拟机的ip然后设为那个值），其他信息与主机一致。然后重启虚拟机网卡（步骤在网络上搜到）

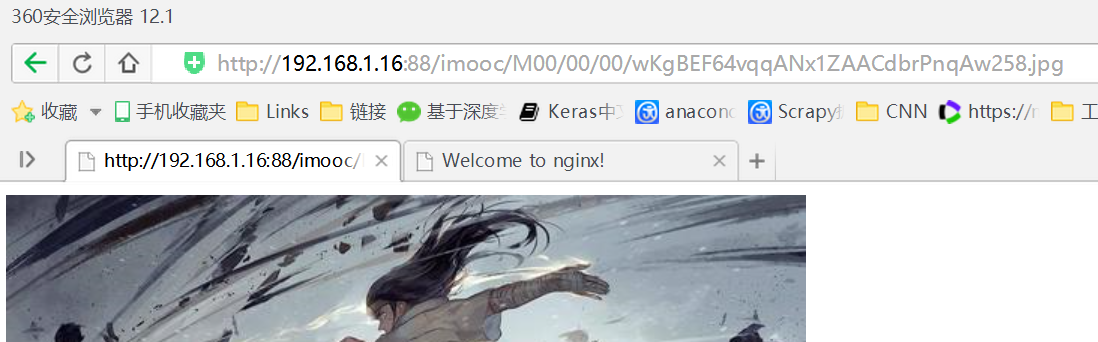


1. 注意重启虚拟网卡时，，发现没有eth0这个设备，是因为这个设备命名为了ens33，所以重启ens33即可如下图。然后就可以和主机互相ping通了。**互相ping通就说明可以在外网直接使用ip地址访问到虚拟机。**

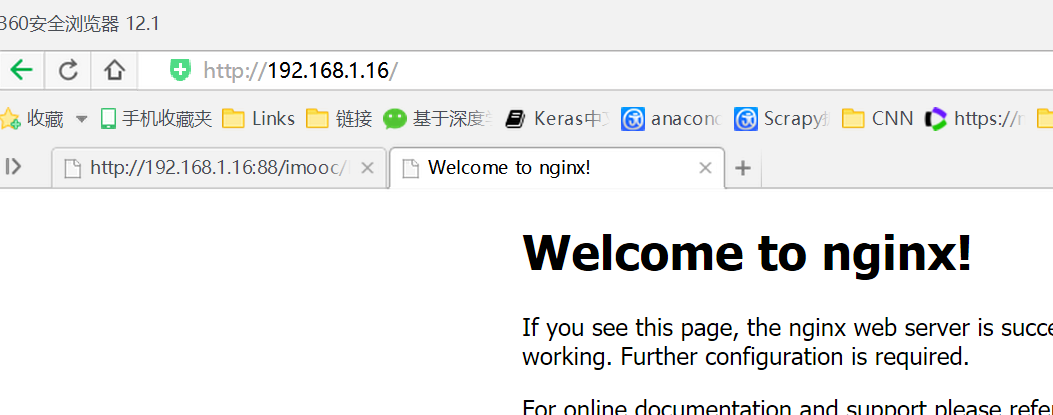


1. 虚拟机的ip为192.168.1.16，重新走一遍上面的nginx和fastDFS配置，配置完成且启动后，不需要端口映射，通过虚拟机的ip就可以从主机上访问服务器了。如图：

（这是上传的测试用的图片，可以在主机上通过虚拟机ip访问到）



（这是nginx启动页，可以看到也是直接使用虚拟机ip访问的）：



1. application.properties文件可以直接使用虚拟机ip了，不再使用端口映射：



补充：linux查看ip的命令：ifconfigwin查看ip的命令：ipconfig

，**注意：在局域网内，主机和虚拟机的ip可能会变，因此需要更改之前的配置文件。所有需要更改的地方有：**

**虚拟机：/usr/local/nginx/conf/nginx.conf 中有一处**

**/etc/fdfs/下的storage.conf, client.conf, mod\_fastdfs.conf中都有需要改ip地址的地方。**

**后端代码：application.properties中tracker\_server =**

**前端代码：app.js开头设置的几个ip地址。**

#### 心跳

心跳就是客户端向服务器端发送的一种消息，让服务器端知道客户端还活着，若服务器端接收不到心跳包，就认为客户端不存在了，就切换该连接，以防资源浪费。

在正常的情况下，用户在手机上结束进程，后端的chathandler会在它的生命周期里面获取到相应的事件，会知道这是人为手动的关闭，就会关闭相应的channel连接。

如果手机开启了飞行模式，对于进程来说是一种被动的无法连接，对于后端的netty来说channel依然存在，因为没有主动的关闭channel，没有主动关闭websocket。如果多台手机都这种情况就会发生资源浪费。Netty这一端就需要接收来自手机的心跳包，接收不到就关闭channel。

#### 部署到云服务器

如果要将项目部署到云服务器，

1. 首先购买云服务器并启动，可以得到云服务器的内网ip和公网ip。
2. 使用公网ip远程登录（ssh工具）云服务器
3. 上传一些安装文件，比如jdk8，fastDFS相关，nginx，tomcat，并进行安装（注意：storage.conf mod\_fastdfs.conf这两个文件服务器的配置文件中，tracker服务器需要改为云服务器的内网ip；nginx.conf中，对于dastDFS的服务器地址要改为云服务器的公网ip；另外可以在nginx.conf中进行配置，为tomcat设置反向代理，使用到的是云服务器的内网ip）（nginx默认端口是80，tomcat默认端口是8080）
4. 我们本地的后端程序：application.properties配置文件中，设置tracker服务器地址为云服务器的内网ip
5. Pom中添加maven打包插件依赖，并且需要配置打包方式为war包（默认打包方式为jar包，jar包中自带tomcat容器，此时不需要，要使用云服务器上安装的tomcat）；除去spring-boot-web依赖中的tomcat依赖（也就是去掉自带的tomca容器）
6. 打包为war包，上传到云服务的某个路径（可能跟tomcat有关），启动tomcat，tomcat就会自动解压这个war包并生成一个目录，里面就有我们编写的程序。
7. 前端打包：manifest.json进行相关设置；项目名上右键》发行》自动打包》云端打包后自动下载apk安装包

设计：

前后端分离的项目，前端使用HBuilder创建的Web App项目，使用了内嵌的MUI框架，第一次接触，遇到不会的就查文档，js也是刚开始学，有java和python的基础，就学得很快，整理了一下基本语法和对dom的操作方式。后端使用springboot整合mybatis提供了一些http接口供前端调用（主要包括，注册登录，个人信息修改，查找朋友，增删好友，处理好友请求，获取联系人列表，获取未签收消息等），netty作为websocket服务器，整合进springboot中，通过websocket传递的信息主要有四种，一个是用户刚连接时发送一个初始化消息，携带用户id，这样后端使用一个map结构管理每个用户对应的channel，实现消息的精准推送。二种是用户之间的消息传递，第三种是用户对接收消息的签收，也就是改变数据库中消息的签收状态。签收是指这个消息已经被用户设备接收到，（因为当有用户a的消息时，用户a可能不在线，这时就把消息存入数据库设为未签收，当用户上线时，自动去数据库读取一次未签收消息，并将所有消息的状态改为已签收）第四种是用户定时向服务器发送心跳数据，保证服务端能够时刻注意用户的在线状态。在 TCP 保持长连接的过程中，可能会出现断网等网络异常出现，异常发生的时候， client 与 server 之间如果没有交互的话，它们是无法发现对方已经掉线的。为了解决这个问题, 我们就需要引入 **心跳机制** 。

心跳机制的工作原理是: 在 client 与 server 之间在一定时间内没有数据交互时， 客户端或服务器就会发送一个特殊的数据包给对方, 当接收方收到这个数据报文后, 也立即发送一个特殊的数据报文, 回应发送方, 此即一个 PING-PONG 交互。所以, 当某一端收到心跳消息后, 就知道了对方仍然在线, 这就确保 TCP 连接的有效性.

TCP 实际上自带的就有长连接选项，本身是也有心跳包机制，也就是 TCP 的选项：SO\_KEEPALIVE。但是，**TCP 协议层面的长连接灵活性不够。所以，一般情况下我们都是在应用层协议上实现自定义心跳机制的，也就是在 Netty 层面通过编码实现。通过 Netty 实现心跳机制的话，核心类是 IdleStateHandler**。

TCP拆包粘包：TCP是面向字节流的，保证发送方到接收方之间的字节流是有序且无丢失的传递，操作系统在发送TCP数据的时候，底层会有一个缓冲区，例如1024个字节大小，如果一次请求发送的数据量比较小，没达到缓冲区大小，TCP则会将多个请求合并为同一个请求进行发送，这就形成了粘包问题；如果一次请求发送的数据量比较大，超过了缓冲区大小，TCP就会将其拆分为多次发送，这就是拆包。

解决方式就是使用netty自带的解码器：

* + LineBasedFrameDecoder : 发送端发送数据包的时候，每个数据包之间以换行符作为分隔，LineBasedFrameDecoder 的工作原理是它依次遍历 ByteBuf 中的可读字节，判断是否有换行符，然后进行相应的截取。
  + DelimiterBasedFrameDecoder : 可以自定义分隔符解码器，LineBasedFrameDecoder 实际上是一种特殊的 DelimiterBasedFrameDecoder 解码器。
  + FixedLengthFrameDecoder: 固定长度解码器，它能够按照指定的长度对消息进行相应的拆包。
  + LengthFieldBasedFrameDecoder

也可以自定义序列化编解码器，但没有了解过。

1. 零拷贝：是指计算机执行操作时，CPU 不需要先将数据从某处内存复制到另一个特定区域。这种技术通常用于通过网络传输文件时节省 CPU 周期和内存带宽。
2. Netty是一个NIO框架（什么是NIO，BIO，AIO），每个用户连接时都有自己的channel，netty使用一个线程组处理用户的连接，另一个线程组处理不同channel的消息事件。每个channel都有自己的pipeline，由若干个handler组成，可以为每个channel的pipeline绑定自定义的handler来实现对消息的处理。
3. 收获：收获就是了解了各种框架的基本使用，这些框架以前都没接触过，只是在这个项目中当时也算是为了将来多熟悉一些框架方便找工作，这些框架我都从0开始学，笔记都整理在了博客上，然后再用到项目里。
4. 困难：
5. 搭建FastDFS时是在本地虚拟机上搭建的，网络环境是自家的局域网。虚拟机的网络连接一开始使用默认的NAT模式，但这情况下同一局域网内的设备无法通过ip直接访问这个虚拟机，因此我一开始是设置了端口映射，将tracker服务器和strage服务器的端口都映射到了宿主机的端口。但是实际运行的时候发现，客户端是需要先访问tracker服务器，此时走端口映射是没问题的，tracker服务器会返回一个可用的trorage服务器地址，该地址是虚拟机的ip地址，客户端会直接访问这个地址，不会走端口映射，就导致上传失败。后来将虚拟机的网络连接改成了桥接模式，取消了端口映射，就成功了。

**桥接模式和NAT模式**

桥接模式下，虚拟操作系统就像局域网中的一台独立主机，能够访问网中任何一台机器。

NAT模式下，让虚拟系统借助NAT（网络地址转换功能）通过宿主机所在的网络来访问公网，也无法和局域网其他主机进行通讯。

1. 用户的管理：把连接到服务器的用户放入一个HashMap中管理，key为用户id，value为对应的channel，这样就可以把消息精准地推送到某个具体用户。但是在分布式服务器场景中可能不适用。
2. 如何在netty的handler中使用service？service无法直接使用Autowired注入netty的handler中，如果要调用service的方法操作数据库，就需要导入service。方法是：

在启动类中，用@Bean注解标注一个方法，该方法返回一个Util工具类实例作为Bean交给Spring管理，这样在netty的handler中就可以使用该Bean（Util工具类实例）。该工具类实现ApplicationContextAware接口，重写setApplicationContext方法获取应用上下文对象，应用上下文对象有一个通过名字获取Bean的方法getBean（String name）。这样该工具类实例就可以通过一个Bean的名字获取Bean了（service也是一个Bean），而且该工具类实例作为Bean交给了spring，可以在handler中直接调用。这样就避免了无法使用@Autowired导入service到netty的handler。

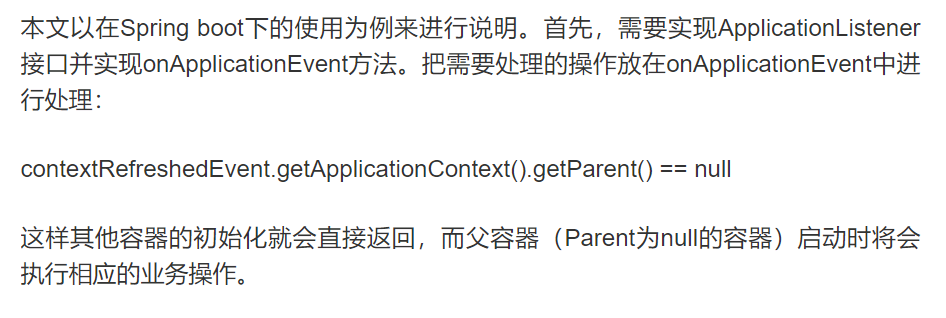
1. Netty是如何整合springboot的？

参考：<https://baijiahao.baidu.com/s?id=1652408605441442858>

使某个方法在spring启动时只执行一次。



定义一个启动器类，并注解component交给spring管理，实现ApplicationListener接口，监听应用上下文事件，当父容器启动时，执行操作。此处的的操作就是运行Netty服务器（WSServer）的start方法，WSServer已经改造成了单例模式。



改进：

1. 改进：用户与所有好友的聊天记录的存储是使用的js列表，调取时需要遍历来比对好友id，速度太慢，js有没有hash结构的？有Map集合可以使用，将朋友的id作为键，聊天记录作为值，存为Map，就可以快速获取。
2. 改进：当前退出登录时就清除用户的全局对象信息，这样每次切换用户登录都会重新拉取信息，因此可以用一个Map对象保存用户全局信息，用户全局信息中除了聊天记录，个人信息等内容，额外设置一个是否登录的标志位，打开时只需判断对应用户的登陆标志位即可判断用户是否登陆，即使退出登陆也不会清空用户数据。这个用户数据可以额外添加一个清除缓存的功能按键提供给用户。
3. 改进：好友分组怎么实现？群聊怎么实现？

群聊实现需要在数据库中添加三个表，一个是群聊信息表，一个是群与成员映射关系表，一个是群聊的聊天信息表。后端对群聊的管理：使用一个Map结构存放群聊，key为群聊id，值为成员id集合。用户信道注册后，查找数据库判断所属的群聊，如果该群聊没有加入到Map结构中就先加入。当用户向群聊发消息时，接收方信息改为群聊id，查找对应的群聊成员id，将消息逐个推送到群聊成员信道上，如果成员离线，就将消息存放到该成员的群聊消息数据表（包含群聊id，发送成员id，接收方id，信息内容），当成员上线时主动获取未读取的信息。

用户分组如何实现？在好友关系映射表中多设置一个字段作为用户对好友的分组标签，拉取好友列表时将标签一并读取出来，然后将所有好友按标签组合展示即可。

1. **其他缺陷**：只把netty当做实时消息传递的工具了，实际上还可以扩展它的传递内容，比如添加好友请求是通过http协议实现的，服务器接收到这个请求后无法及时告知接收方有一条新请求，目前需要用户定时的查询好友请求记录。但是如果将这个请求通过websocket传递，那么netty服务器就可以实时将这个请求告知给接收方，不必再通过定时查询来获取好友请求了。另外还有好友请求通过的消息传递。