# Netty+Spring Boot仿微信全栈开发高性能后台及客户端

# Netty

<https://blog.csdn.net/qq_26975307/article/details/85004424>

是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

简单来讲，Netty是一个提供了易于使用的API的客户端/服务端框架。Netty并发非常高，一个非阻塞的IO，Netty传输速度也非常快，因为他是0拷贝，什么是零拷贝？NIO中的特性之一就是零拷贝，在Java中，内存分为堆和栈以及字符串常量值等等，如果有一些数据从IO中读取并且放到堆里面，中间会经过一些缓冲区。

具体来讲，如果要从IO中读取数据，分为两个步骤：

(1）从IO流中读取出来放到缓冲区，程序从缓冲区中读取，再放到堆中，此时数据就会被拷贝两次才能到达堆或者堆内存中。如果数据量很大，那么就会造成资源的浪费

(2）Netty其实就是利用NIO中的零拷贝特性，当Netty需要接收数据或者传输数据的时候，就会新开辟一块堆内存，然后数据就直接通过IO读取到了新开辟的堆内存中，这样也就加快了数据传输的速度。

<https://netty.io/wiki/user-guide-for-4.x.html>

## 阻塞和非阻塞

线程访问资源，该资源是否准备就绪的一种处理方式

线程在访问某一个资源的时候，该资源是否准备就绪的一种处理方式，如果说该资源当前没准备就绪，这个时候就会有两种处理方式：阻塞与非阻塞

（1）阻塞：这个线程会一直持续等待这个资源就绪并处理完毕，直到响应返回一个结果，这个时候线程是一直阻塞状态，不可以去做任何事情

（2）非阻塞：这个线程直接返回结果，不会持续等待这个资源处理完毕才响应，它会去请求别的资源。



## 同步和异步

是访问数据的一种机制

同步：主动请求，等待io操作完了会通知

异步：主动请求可以发起其他请求，异步通知

这里的 “同步与异步” 指的是访问数据的一种机制，类似于Ajax。  
    （1）同步：主动请求，并且会等待IO操作完成之后，IO会有一个通知  
    （2）异步：当一个线程主动请求数据之后，可以继续处理其他任务，发起其他请求，多个请求完成之后再逐一的通过异步形式通知



## BIO

（1）同步式阻塞IO：Block IO，IO在进行读取的时候，这个线程是会被阻塞的，无法去做其他操作，传统简单，通信方式也是如此。并发处理能力非常低，线程之间访问资源通信的时候，耗时也非常久，所以会比较依赖于网速与带宽，JDK1.4之前均是如此。

（2）服务器会有一个专门的线程称之为Acceptor，专门用于负责监听来自客户端之间的请求，只要客户端与服务端有建立请求，此时客户端与服务端之间都会创建一个新的线程进行处理，这是一种典型的 一应一答的模式。随着客户端逐渐增多，两者之间会频繁的创建和频繁销毁线程，此时服务器端会有很大的压力，甚至宕机。

（3）改进之后则通过线程池来处理，这种方式也可以称之为伪异步IO。

同步阻塞IO Block Io

Io在进行读写，这个线程是被阻塞，无法进行其他的操作 并发能力很低 资源间通信慢 依赖贷款网速



一应一答 客户端逐渐增多，频繁创建线程，销毁线程 很差劲 改良之后，就是线程池 伪异步io 这个bio就像是上厕所找不到位置一直等着

## NIO

（1）同步非阻塞IO：New IO（Non-Block IO）：JDK1.4之后。selector（其实就是一个线程，并且会主动轮询），也称之为多路复用器Buffer，它是一个缓冲区；Chanel，是一个双向通道。

（2）客户端在与服务端进行通信的时候就会使用到selector，两者要建立链接的时候，客户端到selector进行一个注册，注册完毕之后就会有一个chanel01，每一个客户端和selector建立连接之后都会有一个chanel，chanel是一个双向通道，可以进行一些相应数据的读写，这些数据的读写都会到Buffer缓冲区中

（3）通过selector注册并建立chanel通道就可以实现两者之间的通信，chanel的数据是一种非阻塞的读取，如果没有数据会直接跳过，不会同步等待数据。selector其实一个单线程，整体来讲，线程开销会非常小，光是一个selector就可以处理成千上万个客户端，客户端的增多不会影响它的性能，这也是与BIO的区别所在。

（4）chanel相当于是一个读取的工具，每一个客户端都可以理解为一个单独的chanel，每一个服务端会有一个selector，Buffer的数据会进行读取，数据被读完之后还是会存在Buffer中，不会因为数据被读取之后会被消息，String中的数据读完之后就没有了。

同步非阻塞io

去上厕所，发现没坑了。就开始玩手机，时不时看看有没有坑，不是光等着。

Selector 选择器，轮询



## AIO

异步非阻塞io 相比下面不等着，在做自己的事情

异步阻塞io 几乎用不到 比如上厕所，没位置了。就等着。就让哪个人好了告诉我我再去。这种方式非常傻

（1）异步非阻塞IO：AIO其实也是NIO2.0，它是一种异步非阻塞的通信方式，在NIO没有的基础上引入了一个异步概念：在读写的时候所有返回的类型其实就是一个feature对象，这个对象模型其实就是异步的在这个过程中会有一些事件监听  
（2）异步阻塞IO：几乎用不到





## Netty

Java 的开源框架 网络服务器客户端框架

使用netty的优点 java开源框架 原始nio有bug存在。

## Reactor线程模型

（1）Reactor线程模型：

1）单线程模型：所有的IO操作都由同一个NIO线程处理，仅限于一些小型应用场景。但在高负载、高并发等情况下使用单线程肯定就不太合理，主要是因为NIO的一个线程同时要去处理成千上万的请求 的时候，在性能上会支撑不了，即便CPU负载100%，对于海量消息的处理，编码解码以及读取、发送消息等情况，依然满足不了。

2）当NIO的线程负载过重之后，整体服务性能处理就会变慢，结果就是导致客户端在向服务端发起请求、链接就会超时，由于客户端一般都会有一种超时机制，反复地向服务端再次发起请求，此时就相当于陷入了死循环，更加加重了服务器负载。

（2）多线程模型：由一组NIO线程处理IO操作

（3）主从线程模型：一组线程池接受请求，一组线程池处理IO



独立处理所有io的操作 完成所有的操作

适合场景：高并发、海量数据复杂的场景

多线程：一组nio线程处理io操作



重点概念：线程池 等待队列

主从线程模型：一组线程池接受请求，一组线程池，处理io



<https://blog.csdn.net/quxing10086/article/details/80296245>

# java仿微信全栈开发高性能后台+移动客户端

## hello netty步骤：

这些东西多看官方文档

一个最简单的Netty服务端包含了五个步骤：

（1）构建一对主从线程组  
（2）定义服务器启动类  
（3）为服务器设置Channel  
（4）设置处理从线程池的助手类初始化器  
（5）监听启动和关闭的服务器





### 先写helloserver



### 编写初始化器



### 子处理器





## 聊天小练习

### 实时通信

实时通信的分类：

（1）Ajax轮训

通过JS以Ajax异步地让浏览器每隔一段时间（10S）发送请求到后端，去询问服务端是否有新消息、新状态等，如果有则取出并通过前端再渲染。但这很容易造成无限循环，也就是前端Ajax会不停地循环后端的数据 （使用场景：浏览器不需要一直刷新，简单的后台管理系统中的数据更新等）

2）Long Pull

与Ajax轮训类似，也是使用异步请求，只不过它的轮训方式不太友好，阻塞式轮训：当客户端发起请求之后，服务端如果未响应，则Long Pull就不会有响应，直到服务端返回response。过程中不停地建立Http请求，等待服务器端进行处理，被动响应，缺点也是非常明显，也很耗费资源，性能低。

3）webSokect - 推荐

Http本身就不支持长连接，Http1.1支持长连接，WebSokect就是使用了Http1.1协议来完成一小部分的握手，简单来讲就是，客户端发起请求到服务端，服务端会去找一个副助理，找到之后服务器端会和客户端一直保持连接，为客户端进行服务，并且可以主动推送一些消息给客户端。

### Websocket

WebSokect有哪些协议，又有什么优点？

1）首先WebSokect相对于Http这种非持久化来讲，是一种持久化的协议，Http的生命周期可以说是通过一个request来进行判定，有一个request请求到后端，后端也会相应的返回一个response给客户端，或者有多个request对应到多个response，两者之间都是一一对应的，有多少个request请求就会有多少个response相应，不会有偏差。此时response其实也是被动的，它不能由服务器端主动发起相应，必须先有request请求。

2）WebSokect由此诞生，它使得资源不会像以前一样浪费，并且它也是非常的主动，只要链接一旦被建立完毕之后，那么服务端就可以不停的主动推送消息给客户端，客户端不需要主动请求服务端也可以达到一样的效果。 也就是说，只要建立一次Http请求就能达到信息的源源不断的传输。类似于在线Online小游戏，一开始建立连接，就可以一直保持在线了。



前端采用hbuilder来开发

### 主程序



### 初始化器



### 子处理器





### 前端

采用hbuilder编写 感觉这个ide不错



补充一个聊天室应用

<https://blog.csdn.net/xiaoping0915/article/details/81202851>

# 使用mui与h5+构建移动端

### 工具

Mui:

<https://www.dcloud.io/mui.html>

h5+

<http://www.html5plus.org/doc/>

hbuilder开发工具

阿里的图标库：

<https://www.iconfont.cn/>

加入购物车，然后添加到项目， 下载到本地 复制文件夹到项目里。然后打开index.html照着做

### 步骤

调试的时候需要打开手机的usb调试模式

新建工程的时候用mui项目 不要用默认的。

Js代码块一般放在body底部

主页面







开发过程要用mui代码块 很方便



这东西在浏览器上看不见效果。用到plus必须要用真机调试

页面被遮挡了 用一个浏览器调试器里的小箭头看高度就可以了

页面切换 事件管理 绑定事件

Tap手机触摸事件



# 使用springboot整合netty构建后台

### 数据库设计

数据库：可以下载phpstudy来替代mysql 也可以直接安装mysql

数据库密码：123456

新建数据库：feige-dev



User: 不采用自增长 分库分表



头像有大小之分 qrcode用来扫码 扫码用来添加好友 cid用于消息的推送

用户请求



朋友



聊天记录



这里时间根据当前时间戳来设置

### idea下springBoot+mybatis的逆向工程



MAVEN工程

Java <https://www.cnblogs.com/shirley-0021/p/8510051.html>

Maven <https://blog.csdn.net/qq_37904780/article/details/81216179>

Idea版

数据库生成实体类

工程目录



如果数据库发生了变化，那么需要把刚刚生成的文件都要删掉。因为逆向工程是追加的方式。

### 创建maven工程、添加依赖

注意检查 idea的配置 要用阿里的镜像 不然特别慢

先把工具类依赖全部注释掉。不然会报错



发现一个问题就是 之前逆向工程的时候就users生成有问题 但是改成users\_copy就好了

### Netty

先复制之前的netty的wssserver 进行一定的修改

作为一个工具类，在项目加载之后再去启动 使用额外的线程去启动

应该是一个单例



新建nettybooter 调用上面这个单例



将以前另外两个文件复制过来



# 用户注册登陆个人信息

## 登陆页面



先将这些页面实现一遍 main.css util.css要看看学一下怎么用别人的设计

这里推荐一个css学习网站：<https://www.w3school.com.cn/cssref/pr_padding-top.asp>



### 后端

后端准备工具类







还有一个对字符串进行md5加密 密码加密

Idworker是唯一id生成器

在sid文件上面加上@Component 里面有个nextshort我们需要用

修改启动类中的包扫描



Example的用法

<https://blog.csdn.net/qq_38719039/article/details/80488515>

查询需要事物注解    @Transactional(propagation = Propagation.SUPPORTS)

事物的资料

<https://blog.csdn.net/shi750989074/article/details/78855770>



Bo 从前端传到controller

vo 从controller返回前端

很多东西不能直接返回前端



先完成控制器



服务接口



完成服务接口





### 前端

## 文件上传



跨系统查看，是访问不到。文件太乱了。没有统一规划

推荐用下面这种方式来上传文件 文件服务器可以是单机也可以是集群



文件服务器 1 第三方 云 只要实现一些接口就好

2 让企业自己搭建服务器

## FastDFS



## 文件上传的流程

Ts相当于一个项目经理 ss相当于程序员要去汇报 定时汇报状态



这一部分他是在linux系统下配置的 我们可以在docker里面配置 就没事了

# 发现页面与通讯录相关功能开发

# 聊天页面开发

# 课程总结