一、自我介绍：

面试官你好，我叫黄万军，是电子科技大学计算机专业的一名研二学生。我这次应聘的意向岗位是java软件研发工程师，

我在学校期间学习成绩比较优秀，本科期间每年获得人民奖学金，研究生期间分别以院第2、3名得成绩获得两次一等奖学金，其中一次是入学奖学金。

除此之外，课余也参加了一些竞赛，比如联发科编程挑战赛，全国研究生数学建模大赛，IEEE-UESTC编程邀请赛等竞赛活动。同时，也积极参加研究生会，参与组织了各种学生活动。

在科研工作方面，我是我们教研室《智能抹灰机器人》项目视觉小组得核心人员。我主要负责这个项目部分底层通信库的开发，视频实时传输与显示功能的开发。以及部分的视觉算法的设计和开发。

其次我利用平时时间系统自学了Java相关技术，熟悉Java后端开发。我的技术站主要是java系列偏分布式方面，

为了实践自己学习的知 识，自己独立完成了一个完整的手机秒杀实战系统以及一个聊天APP，基于视觉的智能游戏脚本辅助，并开源在github上面。 此外，我平时偶尔也会在csdn上写写博客，写一些小工具代码放在CSDN上供别人下载。

平时除了学习科研以外，课余活动是打羽毛球和健身。

因为家庭经济条件不是很好，我也会利用闲暇时间在网上授课，主要是计算机方面的课程，凭借这个能力， 我在研究生期间实现了包括学费在内的一切花费的自给自足，主要就是给家庭减轻了经济负担，同时也 促使自己养成独立自主的性格以及培养了沟通能力和人际交往能力，以及责任心和积极向上的心态。

二、抹灰机器人

能不能说一下这个抹灰机器人项目？

这个项目是上海荷福集团、南通第三建筑集团与我校签署的多个合作项目中的一项。其目标是研发两款智能建筑抹灰机器人，一款是主体抹灰机器人：负责主体墙面抹灰；另一款辅助抹灰机器人：负责阴阳角、接缝、窗框、门框抹灰以及修补抹灰等。目前刚刚完成交付，还在提供一些技术支持以及优化。

在这个项目中，我的工作主要是负责，

1、使用C/C++语言实现自定义的通信协议（主要是Modbus的各种命令模式下多种数据的序列化封包、拆包以及错误信息反馈（串口通信，一开始申请一个缓冲区，如果打包或者拆包成功返回缓冲区使用长度，否则返回错误码），当然还有其他通信协议，因为我们这个项目里面很多个系统之间通信都是靠TCP通信的，所以这个协议也要自己实现。其实就是提供一些接口，大家要通信的时候就调用我的接口打包（帧开始，消息头，功能码，序列号，类型，对应类型长度的数据，CRC校验，帧尾（特定的一个十六进制数据））（一个约定好的格式的结构体，接收方拿到数据后直接调用拆包的API，便能得到他需要的数据类型，其实就是所有人把需要发送的数据放到一个统一的结构体中，根据调用者传参不同分别填写不同的数据段）

常用的TCP函数：send()和recv();返回大于0 ，表示发送和接受的字节长度，小于o,表示网络错误，可以重发，等于0，表示断开连接。

亮点：便是编码规范，设计按照成熟的协议来实现。这个算是亮点吧。

问题就是我刚开始开发的时候总是校验失败：因为我忽略了一个东西，数据存储的大小端。网络字节序是大端，但是主机存储（cpu架构）可能大端也可能是小端。所以我刚开始写的时候错了，然后我又写了一个转换，在填充的时候，把字节序换过来，但是这样明显不号。不通用。后来发现，windows平台提供了htons()之类的函数，自动帮我们根据大小端转换。但是这个过程也算是解决了一个问题，并且学习到了很多东西。

难点、问题和亮点：（有问题才有亮点）

延迟和帧率

刚开始的时候，采用YUYV的视频获取格式，不需要解码器，因为怕采用需要编解码的格式的话，会有大量的计算占用CPU，影响效率。但是这样带来一个问题就是因为我们是分开部署的，传输的数据量太大（2048X1536，178广角摄像头，视野很大），延迟的和帧率都很低，后来发现对每帧进行采样进行传输，大概是2：1或者4：1的情况对每帧进行采样，刚开始的时候我们怕采样会不会有延迟。最后发现反而更快，又因为本来软件界面显示区域就小，所以不需要太高的分辨率。这样就解决了延迟和帧率。经过多次测试，延迟控制在100ms以内，流畅度能到25帧左右。超额完成了任务（300-400ms）。所以这是一个亮点。

低延迟的意义：除了自动以外，复杂场景下有一个人工接管的模式，因为毕竟不可能太智能，还是会受环境影响。这个延迟的意义就是为了及时控制供灰管的出灰量，使得抹板上面的灰量均匀，所以时间很重要，并不是单纯的显示一下。

采用的手段：

采样 （减少数据量）+ TCP传输。如果是UDP的话。因为不可靠，所以不能保证时间，还有就是开发难度大，比如如果出现丢包之类的异常情况，都要自己处理，很可能需要丢弃，比如一帧图片数据量很大的，丢了一个包就要全部丢弃，这个代价太大，因为是做工业项目，所以工业上要求的是稳定可控，再加上我这个了刚好是点对点通信，综合商量选择了TCP协议，利用TCP协议的可靠特性来保证。

精确定位图像显示的话。是用的一个单生产者消费者模式，在缓冲区中一次只放一个结构体。精定位线程计算完成后需要在图像上画一些标记，然后再打包发给我显示。所以原本计划放一个双向链表来做缓冲区，类似于消息队列，只不过存放的是图片。后来发现并不需要，实际情况下生产者比消费者慢很多。而且因为实时性，生产者每次只去读最新的数据，以前的数据就算有也不要了。都需要删除，所以直接只要一个。使用事件实现数据同步。

具体的话，项目初期，我学习了一些视频编解码方面的知识，主要是YUY2和RGB的转化，因为YUY2是无压缩图像格式的视频，系统资源占用少（因为不用解码），不需要解码器，缺点是帧率稍慢（受限于USB分配的带宽）。因为我们除了显示外还需要进行计算。所以选择这种，后来又直接使用opencv提取图像，直接使用mat。

它将亮度信息（Y）与色彩信息（UV）分离，没有UV信息一样可以显示完整的图像，只不过是黑白的，这样的设计很好地解决了彩色电视机与黑白电视的兼容问题。

我们先设想一张图片，它具有长和高两个变量，首先明确一点，在我们对图片进行处理转换时，这两个变量width和height是不会变得，也就是说一张RGB图像和一张YUV图像它们的长和高都是一样的，这一点对我们后续的处理很重要。同时它的长和高都是以像素点为基本单位，所以总的像素就是这张图的面积width*height，既然width和height都不变，那么一张图片的总像素点肯定就不会变了，所以图片的一行所占用的内存就为: width*(一个像素点占用的字节数)，对这些有个明确的概念后，我们处理起来就容易多了。一张RGB24图像不考虑alpha通道，那么它一个像素占用三个字节，考虑alpha通道RGB32，那么一个像素就占用四个字节。

对于YUV格式的图像来说，降低的只是它的色度采样率，也就是它的UV分量，而亮度分量Y是没有降低的，这里就不再做具体的说明。例如，4:4:4的采样方式是一个Y分量对应一个U和一个V分量，并没有降低；4:2:2是两个Y共用一个UV分量；4:2:0是四个Y共用一个UV分量。所以它的像素点数仍然没有变得。如,YUYV格式16位一个像素，一张图片占用的总内存为: width*height*2，其他格式内存占用情况可以看这里。

下面对YUYV码流进行提取单帧处理，现在我们应该很清楚的知道了，YUYV格式单帧图片所占用的内存应该就为width*height*2，yuv格式和rgb之间的转换都是有相应的公式的，网上一找很多的讲解，这里也不再介绍了。

视觉供灰算法的设计和模块开发：

这个算法主要是从摄像头（视频服务器拿图片，这里是使用的opencv直接拿Mat），然后直接在ROI（目标区域），进行图像的预处理。主要就是矫正摄像头的畸变（主要有桶形畸变和枕型畸变，通过棋盘格标定板找角点，根据公式计算畸变矩阵，然后以后每张图都进行一次转换），Canny边缘检测（高斯滤波+中值（模糊），sobel算子算梯度，非极大值抑制，最后通过双阈值确定边缘），形态学变换（主要就是开关运算，把上一步找到的边缘粗化，使之连通），找外轮廓，存在一个vector中，然后把小于设定阈值的轮廓去掉。保留大的，然后从上下两个方向进行分区间寻找标记点。找不到的话就根据整体情况进行补偿。比如设初始经验值值或者均值或者众数。然后把得到的结果打包发送给主控。

TCP需要分包，socket 自动分包组包功能。

C/C++相关知识：

同步，锁，mutex互斥锁。资源互斥访问。

三、秒杀系统

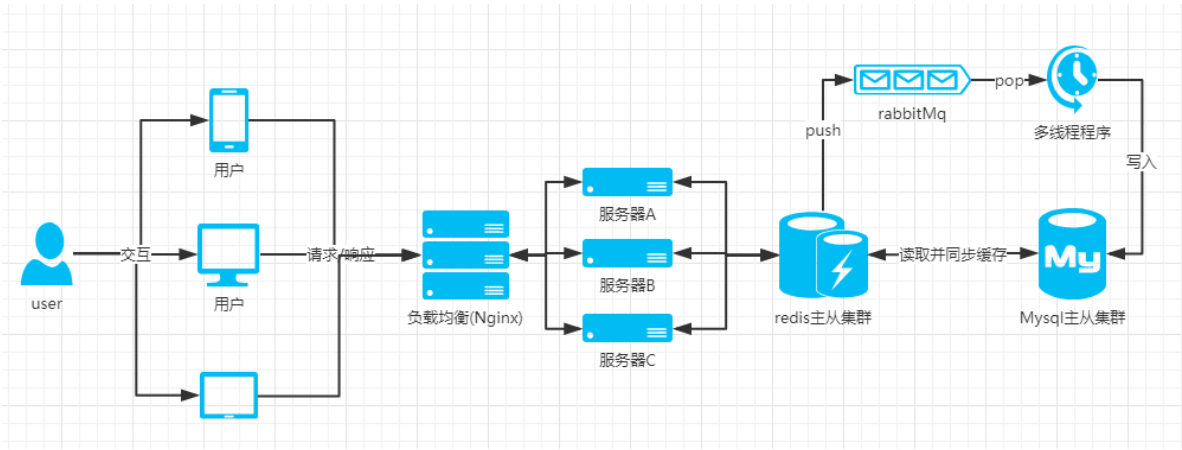
1、你是怎么想到要做这个系统的?

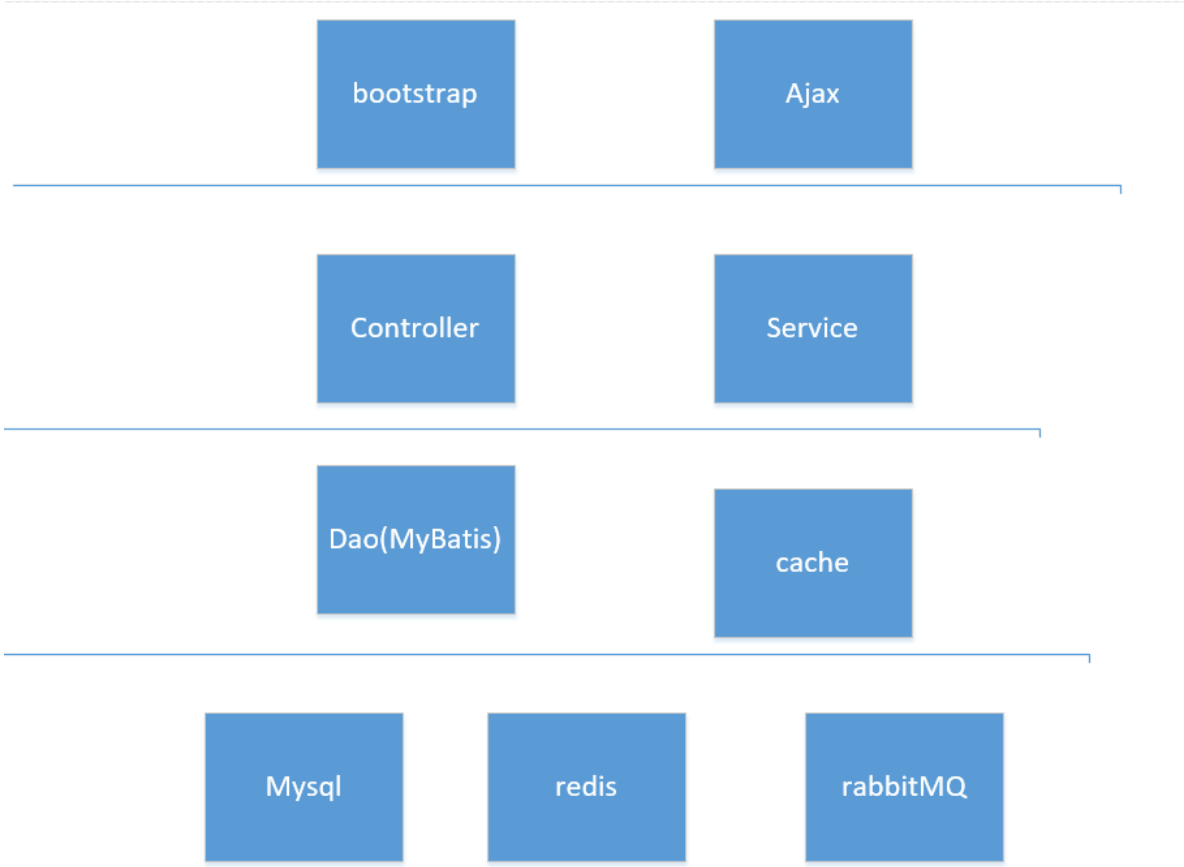
主要是我想系统地学习新的技术。因为我以前没有用过这种技术，而且也为了找工作，需要把技术都实战一下。

2、聊一聊你的这个秒杀系统?

基于Spring Boot 2.x 的一个手机秒杀项目，本项目来源于我学习的多门实战课程内容，然后整合各种知识独立完成本系统各模块开发并进行一些改进（比如技术版本的升级， 支付功能的完善）。目前已经开源在github（https://github.com/haungwanjun/seckill）， 并且部署在阿里云服务器。

3、这个项目的架构图画一下。





根据架构图分析一下，

1、首先用户通过任意渠道访问我们的网站，然后根据一定的路由规则（比如对用户的ID进行hash）被分配到某个服务器接受服务。（我这里没有用Nginx做负载均衡，但实际应该要做的，不然所有用户都访问同一个服务器，负载 太大。）

2、当用户进行秒杀时，因为考虑到同一个时刻，并发量可能会特别大。所以不能让服务器直接访问DB，不然DB很容易挂掉，所以应该使用redis加缓存。用户在秒杀的时候在Redis中预减库存减少数据库的访问，同时使用内存标记减少redis的访问，（redis的处理能力也是有限的，负载太大也是会宕机的，所以这里也要进行Redis的保护，即加一个标记变量记录是否还有商品，如果商品已经没有了，那就置位，这样的话，后续的请求就不会去访问redis然后直接返回秒杀失败）。

3、RabbitMQ队列缓冲，异步下单。因为服务器处理下单涉及DB的读写，当并发量很大的时候，需要很多时间，从而用户体验会很不好，因为需要等待很久才知道结果。所以采用消息队列异步下单。即如果用户秒杀成功，那么创建的订单并不直接写入DB。而是给rabbitmq发送一条message.然后就直接返回给用户说下单成功。然后由监听消息队列的消费者根据接收到的消息，创建订单并写入DB.这里为了提高效率，可以使用一个线程池来解决并发及连接复用的的问题。

4、用户下单完成后，点击订单详情可以查看订单详情，然后选择立即支付。 可以使用支付宝支付，因为时测试，所以使用的是沙箱环境。想体验的朋友可以下载沙箱钱包来测试一下。支付完成后，可以回到商品列表继续秒杀。

难点：（比如session丢失什么的都是小问题）

高并发环境下的线程安全的处理：

比如，我想要做一个限流redis的流量，因为比如我库存已经没有了的话，我便希望接下来的请求不要再去访问redis,因为它的负载也是有限的。所以我便想到使用一个标志位来实现，但是我有多个秒杀商品，所以需要使用多个对象，那么就需要一个集合，字段是商品ID+标志位，比如true/flase;那就是一个map.首先想到是concurrentHashMap。因为要加锁，后来又直接使用 Volatile + HashMap;实现提

高性能。

比如

亮点就是使用了一些比较热门的技术，比如redis和rabbitMQ，并且在使用的过程中我做了很多优化来提高性能，

第二个优化：

缓存预热，为了避免缓存穿透，具体就是再Bean初始化阶段预加载。Controller 实现IniticalizingBean接口。并冲洗afterPropertySet().再bean初始化过程中加载进redis,从而避免这个问题。

第四个：

接口防刷

主要就是防止按键脚本之类的。疯狂点击秒杀按钮。这样疯狂发送请求的话，容易给服务器带来很大的压力。所以我们对每个用户进行了限定。比如每个用户1秒钟内或者3秒钟内只能点击5次按钮。超过规定的次数。就返回点击太频繁的异常提示。后台接口不处理业务，直接返回异常。这样的话可以很大程度上减少服务器的压力。

具体实现：使用redis缓存服务存储每个user在一段时间内的访问次数。设置一个key和过期时间。如果在过期时间内次数超过规定次数，那就返回点击频繁异常。不进行任何操作。

第三个：

LocalThread的使用：

用户信息：

支付的时候每一个线程根据自己的AppId访问concurrentHashMap获取到对应的ConfigureBean，直接构造对象。这样就省去了临时创建对象的过程，预加载参数的过程，如果出错可以提前知道，不用等到时机发生支付才知道。

第五个：

尚存的问题：

rabbitMQ并发的优化：我直接使用的是ampqTemplet 模板类。在我这个应用中是够了，spring中很多模板类使用了模板模式（就是实现一部分公用方法，其他的特有方法不实例化，留给子类实现）。在使用RabbitMQ官方的Client时，Connection对象创建的是TCP连接，TCP连接的创建和销毁本身就是很耗时。因此需要使用连接池技术预生成Connection，每次使用都从连接池中获取Connection。在使用SpringBoot时，RabbitTemplate比较死板，不能够满足项目中动态创建队列并发送的需求。

优化

因为高并发系统瓶颈在数据库：

根本解决方案加缓存！！！在保证数据一致性的前提下加缓存！（CAP理论：可用性，分区容错性，数据一致性，高可用，需要权衡）

页面优化技术：

1、页面缓存+URL缓存+对象缓存

因为内存redis里面比DB要快，所以最好加各种各样的缓存，尽量少访问DB。

对象级缓存，如果有更新一定要记得把数据库和缓存一起更新，这样的才能保证数据一致性。

2、页面静态化，前后端分离

最近比较火的，Restful 风格：前后端分离，在本项目中前端就是html+Ajax ，只传输动态参数，也就是VO对象。然后前端拿到数据后通过Ajax进行渲染。把页面等静态资源缓存在客户端，这样前端和后端之间的交互就只传输需要的参数就行。并不需要后端模板引擎吧页面渲染好然后把整个页面传到前端，极大的减少了服务器的压力和网络带宽的压力。

3、静态资源优化

js/css压缩，减小流量。

多个js/css组合。减少连接数。

4、CDN优化（未涉及）

接口优化技巧：

1、Redis预减库存减少数据库的访问。

2、内存标记减少Redis访问。（即：如果预减库存那一步已经把flag置位，表示没有商品了。那就不应该访问redis了。这样以后的请求就可以直接返回秒杀失败，从而减少redis的压力）。

3、RabbitMQ队列异步下单，增强用户体验。原理见：秒杀过程

安全优化：

1、秒杀接口地址隐藏：

为了防止别人提前得到接口，通过机器人来刷单。所以必须得等到秒杀开始时候才能点击按钮，此时再去请求真正的地址。然后再进行秒杀。

2、数学公式验证码：

数学验证码或者其他什么选字验证码之类的，主要是也是为了防止机器人刷单比如按键脚本什么的。但是对于用户来说的话不太友好。因为太麻烦了。所以这个看情况使用。

3、接口防刷

主要就是防止按键脚本之类的。疯狂点击秒杀按钮。这样疯狂发送请求的话，容易给服务器带来很大的压力。所以我们对每个用户进行了限定。比如每个用户1秒钟内或者3秒钟内只能点击5次按钮。超过规定的次数。就返回点击太频繁的异常提示。后台接口不处理业务，直接返回异常。这样的话可以很大程度上减少服务器的压力。

具体实现：使用redis缓存服务存储每个user在一段时间内的访问次数。设置一个key和过期时间。如果在过期时间内次数超过规定次数，那就返回点击频繁异常。不进行任何操作。

5、我看你用过redis。为什么要使用redis?有什么好处？你是怎么使用的?

其他东西：

你的数据是怎么放入的？所有数据都放入么?

我是只放入频繁修改的数据，比如在这里就是库存，其他的信息不用。因为在普通访问过程中，这些数据都可以一次访问了之后缓存在本地，这个场景下只有库存是频繁修改的数据，所以缓存它，其他不缓存。不然的话。缓存数据太多。内存中根本存不下。

缓存是分级别的。

因为内存redis里面比DB要快，所以最好加各种各样的缓存，尽量少访问DB。

对象级缓存，如果有更新一定要记得把数据库和缓存一起更新，这样的才能保证数据一致性。

6、为什么要使用rabbitMQ？其他的MQ了解过吗？怎么保证高可用的?

我这里是

7、MyBatis核心原理是什么?

8、MySQL了解多少?索引是用的什么数据结构，为什么要用索引，怎么优化，多表查询?

9、数据库设计?怎么分库分表?怎么防止热点数据过于集中?怎么更方便的扩容，不需要数据迁移或者数据迁移比较少?

10、设计一个高可用的分布式系统?

11、你的系统能扛得住多大的压力?一般的数据库或者redis 的QPS多少?

我在虚拟机上面比较小。redis 的QPS大概4w左右。

我本机Mysql 的QPS大概一两千。然后电脑就卡的不行，会假死。

3、飞信APP

项目描述： 这是一个基于Netty框架的类似于微信一样的聊天App，我的目的是学习聊天服务器的开发。

已经实现了一般聊天软件都有的功能。比如说单人聊天，多人。

学习了一 般聊天服务器使用的是webSocket协议，他是一个长连接。一旦建立间接之后可以一直通信，不同于 Http的短链接。但是因为资源的原因，我们可以设置心跳机制。让它闲置的channel超时后自动断开。

每个用户连接是一个channel,在刚开始的时候便绑定，一个channel有一个selectKey用于workerEventLoop监听事件和处理，收到请求后，把写事件封装成一个个task放到taskQueue中，然后等读时间和accept事件(也是读事件)都处理完了，再去处理task。EventLoop是一个无限循环，netty几乎所有的事件和任务都是再EventLoop中执行得。而这些事件或任务大多会通过netty的ChannelPipeLine的Channelhandler链执行下去。每一个channel(链接)都有一个ChannelPipeLine，这条连接的读写操作都会通过这个管道里面的各种Channelhandler处理。 处理事件：读任务：通过pipeLine里面的ChannelinboundHandler处理读事件 写任务：通过ChanneloutboundHandler（都是一个handler链）处理写任务： 其中ChannelPipeLine有两个节点：分别是head和tail.读事件是head->ChannelinboundHandler链>tail; 写事件（task）： ChanneloutboundHandler链->head；因为head持有unsafe对象，可以直接操作底层缓冲区。与IO关系最密切。

具体细节：每一个user分配一个Channel.由workerGroup管理。bossGroup只管理接收请求，收到请求后，把这个请求封装成一个Channel，丢给workGroup进行管理，本质(每条线程)都是一个selector;线程组就有多个Selector,每个Selector只负责监控注册在它上面的channel.每个Selector的执行流程是，每一次循环，Select()轮询所有已经发生的事件selectKey， （每个selectKey 绑定了一条通道，每个通道一条连接）比如度或者写事件发生，所有实际操作都是UnSafe类是实现的，然后把这些事件包装成一个个Task(或者定时task),放到任务队列，然后下一个步骤就是去执行task，

双人（多人）通信：因为每人都有一个userID都绑定了一个channel,我们可以根据接收到的消息的接收方ID在GroupChannel 找到对应的channel,如果找不到说明不在线。以未签收状态入库，等用户上线后拉去，如果在线， 入库后直接推送，签收后修改状态即可。根据消息是否签收的状态，显示不同的状态，把每个人的历史记录都插到本地缓存的一个列表中。包括自己发送的和接受的，根据时间线来存储。 添加好友：根据ID准确搜索用户，找到便添加。发送一个好友请求消息。未签收状态入库，如果在线直接推送。不在线等上线后推送。当然可以使用第三方推送服务器。 通讯录，每次点击到这个页面，读取本地缓存。如果有添加好友等操作，成功后自动更新本地缓存，不用每次都去数据库里面读。 个人信息：缓存本地一份，每次更新后入库并更新缓存。二维码也是使用插件生成。保存账号信息。这 里没有 加密，实际上应该是要使用加密算法加密的。只有账号信息没有其他信息，所以也没什么关系。 头像上传到fastDFS文件服务器：

使用Nginx代理的：

难点：

数据库的设计： 比如好友：每一个好友请求需要添加两条记录。分别是userID和friendID.对调后在存储一次。因为我的好友是所有和我userID绑定的对应的ID;

不过这个可以使用分组查询优化：select friendID from my_friend group by userID Having userID = "0123131 ";

朋友圈的设计：每个人发送一条朋友圈。那便像所有的未屏蔽的朋友都推送一遍。然后再他的缓存里面根据时间线插入对应朋友的朋友圈消息队列。然后这条消息还应该有一个字段来存储那些人评论和点赞了。这也会和其他人进行推送。但是只会推送到都能看到这条 朋友圈并且是好友的人的channel.这样才能双方可见。可以使用inner join ： 既是这条消息的接收者也是自己的好友，就会显示。

具体而言，

1、介绍一下Netty

Netty是一个异步事件驱动的网络框架。快速开发高性能、高可靠性的网络服务器/客户端程序。

重点是NIO、快速、高性能。

2、Netty的线程模型

单线程模型

在 ServerBootstrap 调用方法 group 的时候，传递的参数是同一个线程组，且在构造线程 组的时候，构造参数为 1，

这种开发方式，就是一个单线程模型。

多线程模型

在 ServerBootstrap 调用方法 group 的时候，传递的参数是两个不同的线程组。负责监听 的 acceptor 线程组，线程 数为 1，也就是构造参数为 1。负责处理客户端任务的线程组，线 程数大于 1，也就是构造参数大于 1。这种开发方 式，就是多线程模型。

主从线程模型

在 ServerBootstrap 调用方法 group 的时候，传递的参数是两个不同的线程组。负责监听 的 acceptor 线程组，线程 数大于 1，也就是构造参数大于 1。负责处理客户端任务的线程组， 线程数大于 1，也就是构造参数大于 1。这种开 发方式，就是主从多线程模型。

3、你觉得那个组件最好，最喜欢?

1、EventLoop

每一个channel 分配一个EventLoop，一个EventLoop可以同时处理多个Channel.

每一个Channel分配一个thread。并且所有的工作都由一个Thread完成。

处理每一个channel 的工作都是调用对应的ChannelPipeLine来处理I/O工作。

读事件：直接处理

写事件：封装成task,放入队列。等select()完成再执行。

NioEventLoop 继承于 SingleThreadEventLoop, 而 SingleThreadEventLoop 又继承于 SingleThreadEventExecutor. SingleThreadEventExecutor 是 Netty 中对本地线程的抽象, 它内部有一个 Thread thread 属性, 存储了一个本地 Java 线程。从名字来看，NioEventLoop是一个单线程的，所以，一个 NioEventLoop 其实和一个特定的线程绑定, 并且在其生命周期内, 绑定的线程都不会再改变。

而NioEventLoop在顶层也实现了ExecutorService接口，表示可以向NioEventLoop提交task，由NioEventLoop进行调度执行。实际上，NioEventLoop维护了一个任务队列，可以执行一些定时任务和优先级任务。NioEventLoop 肩负着两种任务, 第一个是作为 IO 线程, 执行与 Channel 相关的 IO 操作, 包括 调用 select 等待就绪的 IO 事件、读写数据与数据的处理等; 而第二个任务是作为任务队列, 执行 taskQueue 中的任务, 例如用户调用 eventLoop.schedule 提交的定时任务也是这个线程执行的。并且，netty的write操作也是通过task来实现的，这就是NioEventLoop设计精巧之处。

NioEventLoop是一个单线程的无限循环，启动时首先会判断是否在循环中，如果在，就直接向NioEventLoop的taskQueue添加task；如果不在，则先启动线程，开始循环，然后向taskQueue添加task。

run方法中的代码看起来复杂，但其实主要的就只有三步：

select操作：轮询注册到reactor线程对应的selector上的所有channel的IO事件

processSelectedKeys操作：处理轮询到的IO事件

runAllTasks操作：处理任务队列的task

2、PipeLine(ChannelHandler/ChannelHandlerContext)

每一个channel都会在创建的时候创建一个channelPipeLine.并且绑定，就像搭积木，编程容易。

ChannelGroup.数据结构是一个Set里面可以存储所有的channel。每一个channel可以添加到多个channelGroup中。

使用的时候可以把所有创建的channel添加进一个全局的Channel中。每一个channel再创建的时候，可以和userId参数绑定。相互之间通信就可以根据userId在channelGroup中去获取这个对象。能获取到就主动发送，（双向的，全双工通信）。如果返回null.表示不在线。入库。

4、你是怎么用的?

4.2 拆包粘包问题解决 netty 使用 tcp/ip 协议传输数据。而 tcp/ip 协议是类似水流一样的数据传输方式。多次 访问的时候有可能出现数据粘包的问题，解决这种问题的方式如下： 4.2.1 定长数据流 客户端和服务器，提前协调好，每个消息长度固定。（如：长度 10）。如果客户端或服 务器写出的数据不足 10，则使用空白字符补足（如：使用空格）。

4.2.2 特殊结束符 客户端和服务器，协商定义一个特殊的分隔符号，分隔符号长度自定义。
如：‘#’、‘$_$’、 ‘AA@’。在通讯的时候，只要没有发送分隔符号，则代表一条数据没有结束。

4.2.3 协议 相对最成熟的数据传递方式。有服务器的开发者提供一个固定格式的协议标准。客户端 和服务器发送数据和接受数据的时候，都依据协议制定和解析消息。

4.3 序列化对象 JBoss Marshalling 序列化 Java 是面向对象的开发语言。传递的数据如果是 Java 对象，应该是最方便且可靠。

4.4 定时断线重连 客户端断线重连机制。 客户端数量多，且需要传递的数据量级较大。可以周期性的发送数据的时候，使用。要 求对数据的即时性不高的时候，才可使用。 优点： 可以使用数据缓存。不是每条数据进行一次数据交互。可以定时回收资源，对 资源利用率高。相对来说，即时性可以通过其他方式保证。如： 120 秒自动断线。数据变 化 1000 次请求服务器一次。300 秒中自动发送不足 1000 次的变化数据。

4.5 心跳监测 使用定时发送消息的方式，实现硬件检测，达到心态检测的目的。 心跳监测是用于检测电脑硬件和软件信息的一种技术。如：CPU 使用率，磁盘使用率， 内存使用率，进程情况，线程情况等。 4.5.1 sigar 需要下载一个 zip 压缩包。内部包含若干 sigar 需要的操作系统文件。sigar 插件是通过 JVM 访问操作系统，读取计算机硬件的一个插件库。读取计算机硬件过程中，必须由操作系 客户端 WriteTimeoutHandler(3) 服务器 1 connect 2 write 1s 3 write 3s 6 write 10s 4 close 6s 5 connect 10s

统提供硬件信息。硬件信息是通过操作系统提供的。zip 压缩包中是 sigar 编写的操作系统文 件，如：windows 中的动态链接库文件。 解压需要的操作系统文件，将操作系统文件赋值到${Java_home}/bin 目录中。 4.6 HTTP 协议处理 使用 Netty 服务开发。实现 HTTP 协议处理逻辑。 5 流数据的传输处理 在基于流的传输里比如 TCP/IP，接收到的数据会先被存储到一个 socket 接收缓冲里。不 幸的是，基于流的传输并不是一个数据包队列，而是一个字节队列。即使你发送了 2 个独立 的数据包，操作系统也不会作为 2 个消息处理而仅仅是作为一连串的字节而言。因此这是不 能保证你远程写入的数据就会准确地读取。所以一个接收方不管他是客户端还是服务端，都 应该把接收到的数据整理成一个或者多个更有意思并且能够让程序的业务逻辑更好理解的 数据。 在处理流数据粘包拆包时，可以使用下述处理方式： 使用定长数据处理，如：每个完整请求数据长度为 8 字节等。

（FixedLengthFrameDecoder） 使用特殊分隔符的方式处理，如：每个完整请求数据末尾使用'\0'作为数据结束标记。 （DelimiterBasedFrameDecoder） 使用自定义协议方式处理，如：http 协议格式等。 使用 POJO 来替代传递的流数据，如：每个完整的请求数据都是一个 RequestMessage 对象，在 Java 语言中，使用 POJO 更符合语种特性，推荐使用。