CloudMessage系统设计说明文档

1. 背景

在学校举办的各种程序设计大赛中，很多参赛项目都是跟消息发送接收相关的项目，比如即时通讯系统，RPC系统，分布式监控系统，这些系统都面临一个相同的问题，那就是消息的发送和接收。系统有多方参与，比如即时通讯有服务端和多个客户端，RPC系统有服务端和多个调用端，分布式系统有控制端和多个代理端，这些系统的服务端(控制端)都需要和众多的客户端(代理端)进行网络消息交互。而网络消息处理部分往往是一个系统设计最复杂的部分，如果每个参赛项目都自己从头编写这部分代码，那么势必会造成各个参赛组的参赛时间过于紧迫，难以将有限的时间集中在核心业务代码编写上，也会降低各个参赛项目的质量。鉴于此，开发了此项目，为各个项目提供统一的消息服务器，各个参赛项目统一接入CloudMessage，消息都通过该系统进行转发，每个项目只要在该系统进行注册，就可以发送接收消息，参赛者可以节约宝贵的参赛时间。

1. 项目接入举例

项目1：即时聊天系统

参赛者只需要编写客户端业务逻辑，每一个客户端现在CloudMessage进行注册，同时将需要消息来往的其它客户端加为好友，就可以进行消息收发；同时可以创建群组，对组内成员可以进行消息广播；

项目2：RPC系统

每一个调用者先在CloudMessage注册成为一个用户，调用者A需要调用项目B的某个功能，那么A将消息发送给CloudMessage，CloudMessage将该消息传递给项目B，B将结果返回CloudMessage，CloudMessage再把结果通知A。看起来更为复杂，但是在有用众多的项目之间的系统调用，比如有十个项目之间都有调用关系，那么不可能这是个项目之间直接调用，必须通过RPC服务器中转，就必须这样做。

项目3：分布式监控系统

分布式系统一般都是两部分，往往是一个控制端，比如机器A上面部署控制端的代码，下面多个Agent端，比如机器B和C上部署了Agent的代码，控制端需要管理所有的代理端，那么必然会有消息交互，比如每个Agent需要上报本机的各项监控信息给控制端，控制端也需要给某个Agent下发任务消息，必然需要一套消息周转中心，同样可以利用CloudMessage来做消息中转。

1. 项目部署方法：
   1. 下载代码，修改jdk1.8。构造数据库，按照src/main/resource下的mysql.sql脚本一条条命令执行，注意我的数据库用户名和密码是root 和ubuntu，第六条命令grant all privileges on \*.\* to 'root'@'\*' identified by 'ubuntu';要改成你自己的用户和密码。直接复制各条命令在mysql的命令行提示符上粘贴执行即可；
   2. 修改配置文件，主要是修改mysql相关的几项，其他的不用变。配置文件在core.util下的config.properties，修改前面跟jdbc有关的四项；
   3. 在eclipse中启动服务端服务，这里需要启动两个服务，一个是jetty，提供http服务，一个是socketserver，提供主要的核心服务，以deug模式或者run模式分别运行core.http.JettyServer.java和core.socket.Server.java，打开多个console，方便调试；
   4. 运行测试用例测试服务器功能：可以自己编写客户端和服务器进行交互，只要按照/wordy/src/main/java/core/service/MsgParseService.java里面的注释部分说明的通信协议来和服务器进行交互就可以，当然，本地也提供了一些测试用例，在/wordy/src/main/java/test/TestMain.java里，可以认为它就是客户端，在这里可以编写各种测试用例。
2. 项目架构图：

监

控

系

统

过期消息定时

清理线程

失效客户端连接定时清理线程

File

Driver

Redis Driver

消息持久化层

客户端连接MAP

客户端处理线程

客户端处理线程

消息QoS

消息QoS

服务器处理客户端线程池

HTTP

Socket

SDK

1. 项目代码设计

5.1 项目设计的第一件事就是数据库构造，目前只需要两张表，一张用户表，一张群表，字段信息详细见mysql.sql脚本里面，都很简单，可以看懂；

5.2 第二件事就是设计通信协议，这是最重要的部分。任何一个C/S架构的项目都必须要有自己的通信协议。这里的通信协议和TCP/UDP要区别开来，任何协议到了传输层，都是被封装在TCP或者UDP包里面的，大部分逃不出这两种协议，只要你的计算机操作系统是运行TCP/IP协议族的。目前大部分计算机操作系统都是运行TCP/IP的。但是对于各个项目它们需要自己的通信协议，比如服务器接收到一段消息，如何解析这段消息的规则就是通信协议；客户端接收到服务器的回复，解析这段回复内容的依据就是通信协议。

上层应用自定义的通信协议一般有两种方式，大多数都是包类型+包长度+消息体这种类型，举例如下：

----------------------------------------

| 包类型 |消息总长度|

----------------------------------------

| 消息体 |

-----------------------------------------

包类型是1字节，表示这个包的请求类型，假设放到我们这里，这个包类型表示客户端跟服务器交互的消息还是客户端和客户端发送消息，可以用一个整数表示1或者2；消息总长度表示后面的消息体的长度，消息总长度这个字段本身用2个字节表示；如果不添加这个字段，那么服务器接收到一堆消息，无从下手解析，因为根本不知道后面的一堆字符串到底哪里为本包的界限，服务器接收到一长串的字符，必须根绝协议把这堆长字符串一个包一个包拆分出来，然后进行处理。那么加入消息长度是120，接下来就接收120个字节的长度，多的不接受，因为多的肯定是属于下一个包的内容了。消息体里面还可以进一步细分字段，比如消息体里面第一个字节表示发送者的QQ号，第二个字节表示接收者的号码。这种通信协议实际上和操作系统是一样的，你看网络层的IP报文格式，传输层TCP包格式，数据链路层的帧格式，都是这样定义的。

服务端和客户端通过socket接受消息是这样接收的，先接收3个字节，拿到包类型和后面的消息总长度，比如第一个字节是1，第二个字节是330，然后再接收330字节，这样一个包就算接收完成了。接下来重复这个过程。当然，如果接收到的一个包无法解析，那说明传送过来的包本身就有问题，很多软件都会这样，就会表现为消息丢失，消息重复等，这就要考操作系统的TCP/IP协议完成流控重传可靠性保证等；

另外闲扯一句：二层数据链路层一般叫帧，网络层的数据叫报文，传输层的数据叫包，所以常见的叫法是MAC帧，IP报文，TCP包；上层应用产生的数据包通过TCP协议封装到TCP包里面，TCP包又被封装在IP报文里面，IP报文又被封装到MAC帧里面，MAC帧直接在网络电缆里面传输，底层的包装上层；这个数据包在电缆上传输，经过路由器，MAC帧被解封出来，路由器是三层设备，处理这个IP包，根绝IP包的报头将这个IP包转发出去，转发出去之前肯定还是要封装成MAC帧的，也就是说路由器要先拆MAC帧作分析处理，然后又封装成MAC帧转发出去，每一种设备都是如此。到达对方电脑之后，网卡接收到数据，扔给操作系统处理，OS先将MAC帧解析出来，得到IP报文，扔给内核的IP报文处理模块，然后扔给TCP报文处理模块，然后解析得到最上层的用户报文，传递给应用程序，应用程序在根据自己定义的通信协议格式进行内容解析处理。

另外一种通信协议定义就是我们这样的，直接采用特殊符号作为间隔，每一个间隔符号之间的内容事先双方约定好，比如接收到消息2+123456+234567表示123456这个用户需要添加234567这个用户为好友。那么双方都遵守这个协议，就不会错。这种通信协议的问题就是间隔符本身转义的问题，很不好处理。在简单的应用并且网络环境非常可靠比如局域网内部可以直接使用这种方式进行通信。

5.3 代码的api层现在只有两种，http和socket，客户端一般通过socket连接服务器，http层API的作用有限，只能针对一来一回的请求作出处理。因为jetty本身就和tomcat一样，是个web服务器，请求过来了，从线程池拿出一个连接处理该请求，处理完了这个连接就回收到线程池里面去了。同一个客户端再发出请求，服务器也不会知道是上次那个。当然，现在大部分浏览器都支持http header里面添加keep-alive属性了，避免频繁为这个客户端创建删除线程，keep-alive可以让这个连接保持一段时间而不销毁。

而且我们的架构图画的是有问题的，因为只有socket这部分有线程池，代码在/wordy/src/main/java/core/socket/Server.java里面，http形式的api在/wordy/src/main/java/core/http/JettyServer.java里面，是没有线程池的，当然，jetty本身带肯定会有这个功能，这里可以自己去研究下。

现在以socket这条线走一遍代码。Socket服务器启动代码在core/socket/Server.java的main函数里面，非常简单，启动的同时就开启两个定时任务线程：

new Thread( new CleanInactivateClientThread()).start();

new Thread(new CleanOverdueMessageThread()).start();

一个是清理不活动的客户端连接线程，一个是清理过期消息的线程，这两个线程后面再说。

接下来是服务器启动socket监听，如果没有客户端连接上来会一直阻塞在这里，知道有客户端连接，则立即将这个连接扔给线程池，从线程池启动一个线程去处理这个客户端的请求。

cachedThreadPool.execute(new ClientHandler(clientSocket));

newCachedThreadPool如何使用详见代码里面的注释或者自己去查。

启动客户端处理线程后，就到了/wordy/src/main/java/core/socket/ClientHandler.java里面来了，这里面首先把客户端发过来的消息取出来，然后放进一个阻塞队列里面，进行流控QoS，然后重新开一个线程处理这个队列里面的消息。当然，这里的阻塞队列里面永远是空的，因为来一个消息就开一个处理线程处理，根本不可能积压，这里只是演示QoS，如果多个接收线程和多个处理线程同时运行，那么阻塞队列里面的消息数量就可能动态变化了。

5.4 消息解析和处理这一部分设计的不太好。在一开始想到http和socket两种形式的请求都会面临消息解析处理这个问题，所以把消息解析设计为一个接口，里面有一个方法：

public interface MsgParseService {

public RequestDetail parseMessage(String message);

}

http和socket都可以用。然后parseMessage这个方法根据协议的第一部分判断这个消息的类型，利用case语句扔到各个请求的处理函数中去，各个处理函数都会返回一个庞大的类，就是/wordy/src/main/java/core/model/RequestDetail.java里面的RequestDetail，这个类包含了各个请求的结构体，这是为了procMessage返回的结果都是统一的一个对象，好传递给消息处理类进行处理。

消息解析完成后返回的是一个RequestDetail类，这个类里面有很多子类，每一个子类都是一种请求的封装，所以每一个服务器接受到的消息经过消息解析后返回的RequestDetail类，里面只有一个子类是有内容的，其它类都是没用的，因为每个接收到的消息只可能被解析为某一种请求。

接下来就是消息处理过程。这个过程http和socket是不一样的，因为消息处理完了后，http是放在jetty的response对象返回给客户端，而socket是直接往客户端的socket上写数据，这样客户端那边就会接收到。所以这里分开了两个包，一个http，一个socket。

各自自己处理RequestDetail。Socket这边处理完直接往client socket上回复，http那边

往jetty 的response对象上写数据回复。

重点说一下socket的几个处理函数。

第一个是/wordy/src/main/java/core/service/socket/MsgProcSocketServiceImpl.java里面的Login函数，这个除了简单的校验用户名密码之外还会涉及到两部分内容，第一是将客户端的socket对象保存在服务器的内存中，也就是一个全局变量/wordy/src/main/java/core/service/SocketMap.java里面的userNumSocketMap内，因为其它用户如果要向这个用户发送消息，必须要拿到这个socket才可以发出去。如果客户端由于意外中断连接，那么服务器是不会知道的，userNumSocketMap里面还保存着这个客户端的连接，那么如果这个时候往这个socket发送消息，就会失败，所以必须清除掉去。这就是服务器端开一个客户端失活连接清理线程的意义。第二个就是接受持久化消息，因为刚登陆上来，也许有其它人或者群发过来的消息，必须接收掉，然后调用持久化层的接口删除掉这些消息。删除消息的动作就交给持久化去完成，它会判断这个消息是否被删除。

第二个是消息发送函数。

/wordy/src/main/java/core/service/socket/MsgProcSocketServiceImpl.java里面的public void sendMessage(RequestDetail requestDetail)函数。这个函数会从userNumSocketMap这个全局变量拿出要发送过去的用户的socket，然后往这个socket上写内容，如果全局变量没有找到这个用户的socket，就调用消息持久化层的API把消息保存在服务器上，这里有两种持久化形式，一种是用文件格式保存在本地，另外一种就是保存在redis里面去，redis本身有持久化功能，不会丢失消息，至于redis持久化功能怎么配置，需要自己研究，也不复杂，几个配置项的事。两种消息持久化的格式，也就是消息怎么保存，请参考代码里面的注释。在这个文件里面：

/wordy/src/main/java/core/persistence/api/PersistenceAPI.java

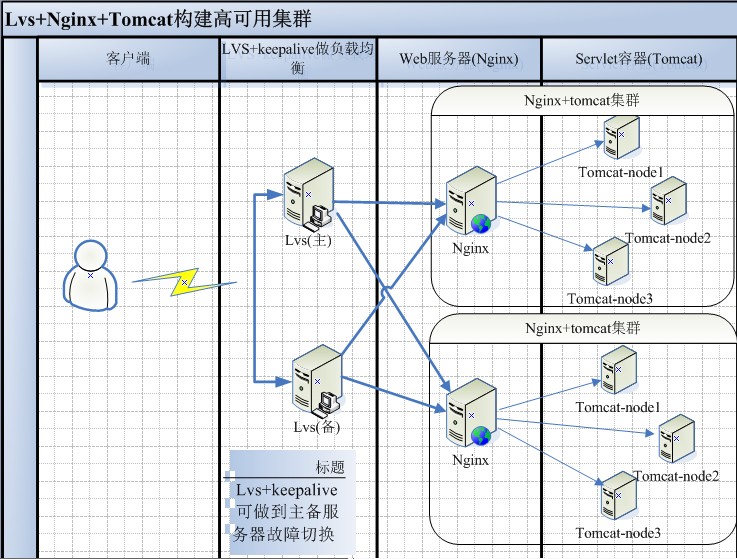
再说说两个定时任务，第一个是失活的客户端线程，他去遍历全局变量里面的每一个客户端，如果这个客户端对象ClientSocket里面的updateTime属性的值超过当前时间180s，那么就认为失活的，删除掉。这里用的是高并发的hash map concurrenthashmap，支持并发。因为userNumSocketMap这个变量使用的人真是太多了。所以客户端那边必须要有一个线程定时向服务器发送心跳，更新updateTime时间，这就是所谓的心跳。客户端的实现在/wordy/src/main/java/test/ReadWriteThread.java里面的UpdateHeartBeatThread类里面。第二个定时任务就是清理过期消息的，消息默认保存一段时间，超过时间后还没被接收就删除掉。这部分没有时间做，你需要明白原理，脑海中全部构思一遍，在大脑中实现了去。

5.5 接下来就是持久化这一部分，这部分内容也很简单，都没实现，框架都建好了。这一份最重要的就是消息的保存格式，我在代码注释里面定义的只是临时想了一下，基本是不可用的，需要想更好的存储方式。怎么存，怎么取，就是存进去要能取出来，并且高效。这部分可以自己再构思构思。

1. 项目代码说明
   1. 项目还有很多没完成，但是不影响整个项目的理解，自己要是有空可以去写，比如群组消息发送接收，持久化层的底层实现，都没有实现，但是框架都写好了，原理也很简单，可以自己去做；
   2. 上面的架构图上，API层有三种，客户端可以使用SDK、Socket和HTTP三种方式接入服务器，但是实际上SDK没有写，HTTP层只有少数的几个请求可以走HTTP层，比如注册，添加好友这种只需要和服务器进行交互的请求可以通过http来完成，对于那种需要和别的客户端交互的请求，比如收发消息，登录这些都是不行的，根本原因就是HTTP服务是没有记忆的，来一个请求开启一个线程处理一个，处理完后不留下任何痕迹，这个线程是Jetty开启的。所以实际上只有Socket这种方式是完善的。
   3. 代码一定要仔细看，每一句都要读懂，多看几遍，虽然谈不上复杂的大项目，但是对于应届生来说足够了，里面刻意用了一些稍微复杂点技术，都有注释，注释要读懂，项目读懂之后消化掉，面试最禁忌自己写在简历上的东西被人问倒，这会造成不懂装懂或者偷别人项目的印象；这个项目也不复杂的，至于没实现的部分，你自己也知道怎么实现的，原理你肯定懂。
2. 项目问题集中

7.1 cloudmessage能否支持高可用和负载均衡？

所谓负载均衡，就是后端部署多个服务器，每个请求来了，通过前段的负载均衡分发器进行请求分发，后端的每个服务器各自承担一点请求，这样就可以线性扩大服务器规模。所谓高可用就是某些服务器出现宕机，服务还能正常提供，比如刚才说的负载均衡器挂了，那么服务就全部挂了，所以负载均衡器本身也要部署两台或者多台，保证高可用。一般的项目部署图如下所示：nginx是负载均衡器，负责转发请求到后面各个业务服务器上，一般部署两个，否则负载均衡器本身挂了就全部挂了，lvs是检测nginx的，一个nginx挂了就把请求全部转发到另外一个nginx上。



cloudmessage高可用当然可以提供，只要负载均衡器多部署几个就可以了。暂时是不能提供负载均衡的，根本原因在于服务不是无状态的。所谓状态，就是服务器保存了客户端的相关信息，这样的服务是不利于扩展的。比如一个电商网站，后台部署了两台tomcat服务器，采用haproxy或者nginx等负载均衡方式分担流量请求。某用户登录网站后，会在其中tomcat A服务内部记录下该用户的session，用户每点击一个页面，发送到后端的请求是经过负载均衡器转发的，可能下一个请求就被转发B，那么这台tomcat上面并没有用户的session，所以用户的状态就变成退出了，这是绝对不允许的。要解决这个问题就是要去服务状态化，将tomcat改造成一个没有状态的服务，也就是tomcat自己不能保存任何客户端的信息。那么方法一是session同步，tomcatA和tomcatB将两者的session全部同步，互相复制，这样每台机器都要保存所有的session，较为浪费内存，很不可取，另外一种方式就是将session保存在另外一台独立的memcache上去，tomcat每次去memcache获取用户的session，这就是session集中化，并且tomcat做到了无状态化，企业一般都是这么做的。

很显然，我们的cloudmessage不是无状态化的，因为服务保存了全局变量userNumSocketMap，它记录了客户端的连接，如果部署多台cloudmessage的话，就会出现上面tomcat的问题，所以必须将这个变量抽取出来，放到独立的存储上去，比如单独部署一个memcache或者redis服务器，专门用来保存userNumSocketMap信息。至于memcache

7.2 看起来userNumSocketMap像一个聊天软件，怎么说是云消息系统呢？

本质上这两者没啥区别，核心功能都是消息转发，并且保证不丢消息。对应的角色其实都可以变通引申。比如这里的用户相当于其它系统的一个项目，一个进程等。群相当于一组项目，一组进程。举例：阿里的蚂蚁理财，余额宝，支付宝这些业务肯定都是独立的，他们之间又需要消息通信，那么必须都连接到某个消息服务中心，消息中心就是cloudmessage，支付宝余额宝蚂蚁理财这几个业务就是用户，有人在手机支付宝上消费200元，那么支付宝就必须向cloudmessage发送消息，加入这个消息只跟余额宝有关，那么就点对点发送，如果跟蚂蚁理财以及其它业务也有关，那么就群发送，这里的群就是余额宝蚂蚁理财和其它相关业务。项目中代码很多地方用了QQ号码什么的进行注释，只是为了理解更加容易。