# 构建一个你自己的微信系统

## 可扩展通信系统实践

Max

2016.2

# 前言

正如你们所知的那样，微信是一个非常成功的在线服务系统，由几万台服务器组成的系统为几亿人提供着稳定的业务服务。可惜作为一个普通的工程师基本上不可能有整体设计这样一个系统的机会，即使加入 xx 也基本是个螺丝钉，只见树木不见树林。看着 inforQ 上面高大上的架构设计，高来高去，个人资质驽钝，看过以后，所得有限，仍然大脑空空，没有太多收获。

2016 年春节前后，我个人做了一个可扩展的系统实践，试图构建一个可扩展的类微信系统，本文记录了构建一个设计，实践可扩展通信系统的过程。

本文只是从外部反推后的一个设计，实践，模拟，实现的过程，和真实环境也不存在任何可比性，本人也和 tx 公司没有任何交集。文中必然充满了臆想，猜测以及错误，不能对本文的任何正确性做任何保证，也不对可能造成的损失负责。如果你愿意给出任何建议意见欢迎 email ppmsn2005#gmail.com. 本次设计开发，实验所使用的所有代码和文档都可以在 http://www.github.com/xiaojiaqi/fakewechat 里找到。

# 1. 目标及局限性

## 1.1 核心目标:

这个系统可扩展，可抗压，稳定并且正确服务。

1. 可扩展，scale out。就是加一些服务器 就可以服务更多的用户
2. 可抗压，系统在负载压力大的情况下，应该能自我保护，并且保证服务质量，不能因此崩溃掉。
3. 稳定并且正确服务，这是一个通信系统，在任何时候都应该保证 2 个用户之间的通信是稳定并且正确的。举例来说　我和一个好友通信，无论对象处于什么样的环境，网络，在任何情况下，我们之间的通信消息都将按照原始顺序发送和接受，而不会出现重复，乱序，掉消息。

注意 因为服务器负载和网络的原因，消息延迟到达是允许的。

## 1.2 本文及实验的局限性

局限性... 太多了，初始数据是随机的，数据集合大小是随机的，系统的架构是理想状况的，所以里面所有的结论都可以认为是不可信。

# 2. 技术要求

Linux 基本操作
熟悉高性能程序开发更佳
TCP/IP 以及编程 至少得熟悉
系统设计 对计算机体系有一定了解
Linux 常用工具熟练使用
C/CPP 或者 java
一门脚本语言 python
常用的 Linux server
大型分布式系统管理 监控 调优能力
数据库使用和简单调优
海量服务器管理维护经验

# 3. 环境准备

## 3.1 开发语言选择

我选择的语言是 Go, 版本 1.4/1.5 而不是 CPP。主要的原因 在前文 c1000k practice guide 一文中，我认为 go 是很好的后台开发语言。

在此次测试中，我也期待使用 go 进行一次实践。go 开发速度快，编译更是便捷，事实也证明的确很好用。

但是 GO 可能存在以下的问题

1）内存占用高，GO 使用的内存可控性不如 CPP。不可控

2）CPU 占用高。 我的 GO 语言代码水平偏低，学习时间大概也就 1-2 个月

3）GC 的不确定性. 这是我最担心的，Go 作为使用 GC 的一种语言，是否能在高负载的情况下，依然很好的工作，是否会出现 FullGC 这样的灾难，还不得而知。虽然已经有小米利用 go 做了很多高性能的负载工作，但我认为最好的办法是做更多定量的测试，设定系统的阀值来避免。

## 3.2 系统优化设置

我觉得以下几篇文章很好 可以参照进行设置，过程略

淘宝千石 高性能服务器架构设计和调优
锋寒 Linux TCP 队列相关参数的总结
前文 c1000k practice guide

## 3.3 脚本语言

python

## 3.4 数据库比较及选型

数据库可以分为关系型的 mysql/PG 和非关系型的 Redis /memcached

关系型的数据库 Mysql 和 pg 都能满足业务要求，mysql 我更熟悉，所以选 Mysql
非关系型数据库 Redis/ memcached

1. 设计需要支持原子操作，也就是 CAS 操作
   Redis 天生单线程，不存在这个问题，而 memcached 也满足这个要求。
2. 内存情况
   Redis 存在 fork 问题，对内存使用有要求
   memcached 的缺点数据不能持久化，数据存在被淘汰的问题。
   所以选 Redis
3. Mysql 和 Redis 之间进行对比
   1. CAS
      mysql 支持
      Redis 支持
   2. 吞吐量
      mysql 强
      Redis 很强
   3. 数据管理
      mysql 可以有完整的数据类型，可以严格保证数据安全，分表机制比较复杂，对硬件要求高
      Redis 没有完整的数据类型约束，完全靠客户程序自己保证，风险很大，对硬件要求低
   4. 数据估算
      以最后 1000 万用户在线考虑 以每人有 50 条相互好友关系，每个用户 id 为 4 字节 1000*10*1000*50*4/1024/1024/1024 = 1.8G，数据量并不大。同样情况下，Redis 操作更简单，可控性很强
      所以 最终选 Redis

## 3.5 缓存系统

暂无

# 4. 系统分析以及设计

## 4.1. 账号

首先考虑 一个用户的信息

简单的看 微信的界面，头像 昵称 微信号。没有类似 QQ 这样的数字 ID，而且微信号可以不设置。我个人觉得从系统设计的角度看，微信自己也需要一个唯一的可以标识用户的标识。这个唯一的标记应该是什么？没有资料描述，我将它设计为一个整数了，我在系统里会用一个唯一的整数表述一个用户。

## 4.2 账户间的好友关系

微信不存在单向好友，所以好友链就是 2 个数字

在用户 100 的关系链表里存在 100,200 这样一个记录，表明 用户 100 和用户 200 互为好友。则在用户 200 的关系链里也应该存在 200,100 这样一个记录。

## 4.3. 多 IDC 以及账号区域问题

### 4.3.1 单 IDC

这个可以从腾讯大讲堂中看到一些端倪，QQ 也有这样的发展历程。最早所有的客户端是连到深圳一个 IDC 区域的，由单个点来完成。这样的好处是业务简单

但是单 IDC 也带来了很多问题

1. 单点问题，如果过于集中接入，一旦出现节点故障，可能导致大量的不可服务。这就是所谓的鸡蛋在一个篮子里。
2. 服务距离太远影响服务质量，设想一下，服务器在深圳，有一个用户，身在在东北，那么他每个消息都是要穿过整个中国大陆。要为他提供很好的服务的质量，代价会比为一位广州的用户大很多。距离的增加必然带来延迟上的增加。
3. 单节点是中心节点 压力太大 继续做扩展很难。你见过 10 吨的卡车， 100 吨的卡车，但是你见过 1000 吨的卡车吗？扩展有难度

### 4.3.2 多 IDC

虽然多年以前，我认为互联网是传统电信是不一样的东西，但是多年以后，我发现世界上很多东西都是类似的。电信的网络和 Internet 本身就是解决这个问题最好的例子。

你的电话是有区域的，也就是本地电话和国内长途和国际长途。你拨打的电话，绝大多数的电话是在本地的，有一些是国内的，更少一些是国际的。对绝大多数人来说本地的朋友

要比远方的朋友多一些额。如果你和你的朋友都在一个 IDC，那么肯定能更好的服务。（8/2 原则，至少我如此认为）

所以在用户的属性里应该有一项很重要的属性 Region, Region 代表了你这个用户属于哪个 IDC，类似于你电话区号，唯一的不同是，你的区号是可以变化的，只是这种变化不是很经常。平均每人每年的变化都不大的。并且这种改变不是由客户端发起，而是由服务器决定

多 IDC 的优势：

1. 数据规模减小，每个 IDC 里的大小都更容易控制
2. 系统更稳定，鸡蛋都放到了各个篮子。杭州被挖断光缆，不会让全中国都无法购物了
3. 即使某个 IDC 故障，其他 IDC 不受影响
4. IDC 相互支撑，当某些 IDC 出现故障，可以暂时把用户导流到其他 IDC，比如滨海 IDC 被大爆炸影响后，迁徙所有用户，就是这样的例子。

多 IDC 好，但是 IDC 带来一系列问题，需要仔细考虑。本次实践没有做这些部分，这是个大工程。实践过程中，将所有的用户按照不同的号段，分配到不同的 Region, 模拟出多 IDC 的情况。而更普遍的情况，则是每个 Region 内用户的号段是不连续，并且离散的。这需要一个专门的服务来解决，我会慢慢实现。

### 4.3.3 IDC 内部的用户如何分布到均匀地服务器上

假设有 IDC 内部有 1000 台服务器可以服务 1000 万用户，如何将用户均衡的分布到所有的用户上？

原始的办法有按照号段来区分，或者用户 id 取余来做，但是这样都存在数目不均衡的问题，一旦存在服务器宕机无法快速迁徙的问题。

可以利用一个一致性 hash 算法，将用户 id 映射成另一个 id 值，解决分布不集中的问题。然后做 sharding.如果用户数目足够多的话，应该非常均衡。（需要慢慢实践这一想法）

## 4.4. 多设备消息的支持

这个主要依靠后面的设计

# 5 核心服务器设计 以及通信保证

这部分是系统的核心，我是这样理解和设计它们的。

## 5.1 背景和需求

首先看前提， 这是一个大型的系统，里面任何一个服务器都是不稳定的，可以崩溃的。原因可以是软件故障，bug，操作系统，断电，操作失误。任何一个服务器的崩溃都是在预期之中的。（这又是云计算的一个典型特征）

然后看消息的传输， 这里的消息可以暂时看作用户之间的聊天消息

首先看丢失，一个消息在这样的系统里传输，是随时可能被丢弃的。虽然我们使用了 TCP 协议，但是需要途径多个服务器，而每个服务器的崩溃都是可以无法预测的，所以任何一个消息都可能丢弃。也就是传输不可靠

再看重传，因为整个系统的服务器是不稳定的，一条消息发出去以后，是否到达了目的服务器,这不可知，所以一定是存在重传机制的，那么对方在收到 2 个重复的消息，一定要能区别出来。一条消息重复收到 100 次 和处理 1 次没有区别。这也就是所谓的幂等性

最后是乱序，同样因为服务器是不稳定的，那么先传的消息，可能因为途经一个缓慢的服务器，会后到目的地，而后传的消息，也许会因为网络优化而先到目的地，这一点接受消息的服务器也需要能区分出来。

## 5.2 预设条件

系统里所有的服务器都是不稳定，可以崩溃的。唯一能保证安全的是被持久化的数据。简单的说，就是数据库 Redis 里的数据，我认为即使重启，在重启前后不会发生丢失和错乱，而其他所有的服务器都可以崩溃，崩溃以后数据就丢失了。（真实的业务情况，持久化的服务器也是不可信的，因为硬盘会损坏。即使使用了多备份的机制，CAP 又会成为一个绕不过去的坑，总之这是一个挑战自我极限，而没有最终解决办法的坑）

## 5.3 详细设计

这样背景情况好像在哪里见过？很多年前，第一次看 TCP/IP 卷一，我们不就是带着这样的疑惑吗？

我理解这个的背景其实就是 Internet 本身，问题则是 TCP 解决的问题。所有的路由器都是不稳定的，消息可能丢失，可能丢失，重传，乱序。所以回顾我开始的观点，世界上很多东西都是类似的。

下面是真正消息系统设计

### 5.3.1 简单图解

发起客户端(client) ====> 网关(gw)==> 源头服务器(localposter) =====> [中转服务器组(poster)] ======> 目标服务器 (localposter) ==>网关(gw) =====> 接受客户端(client)

发起客户端，接受客户端(client): 是同样的软件，也就是一个网页，或者 app

源头服务器,目标服务器(localposter) : 是同样的软件，它们负责消息的处理和存储到 Redis 工作

网关(gw): 接入客户端的请求，并做入口控制，防止压力过大，压垮后面的服务器

中转服务器组(poster)： 负责不同 RG 间消息传递。

## 5.4 用户数据部分

完全参照了 TCP 协议的设计 用户包含了几个计数器和 2 个队列

1. **用户的全局计数器**
2. **用户的好友计数器**
3. **队列**

**用户的全局计数器包括**

| 名称 | 类型 | 作用 |
|---|---|---|
| sendid | 整数 | **记录用户所有发过的消息 ID** |
| sendackid | 整数 | **记录用户所有发出的消息，被对方服务器接受的响应的 ID** |
| Recvid | 整数 | **记录用户接受的消息 ID** |
| clientrecvid | 整数 | **记录客户端已经获取的消息 ID** |

**用户的好友计数器**

2 个好友之间还维护一对 1:1 的 sendid, sendackid 来表明相互之间消息的顺序。

**队列**

sendqueue 发送队列，用户所有发送的消息都按照发送顺序排列在里面

recvqueue 接受队列，用户所有接受的消息都按照接受顺序排列在里面

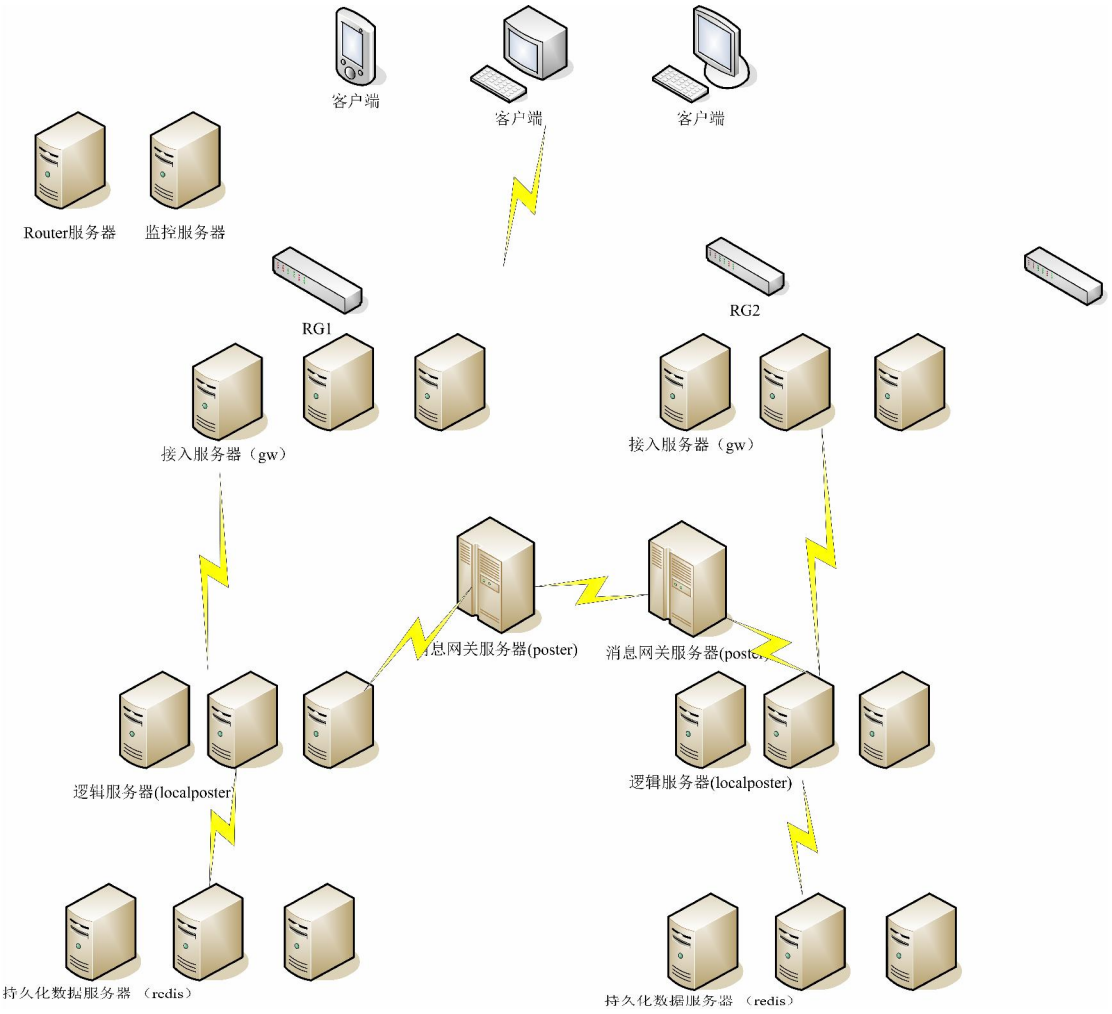
## 5.5 客户端发送消息的简单流程：

工作流程： 发起客户端查询自己的计数器 sendid, 假设服务器返回 100, 此时发起客户端发送一条消息，在消息上带上 sendid 101。此后重新查询自己的计数器 sendid, 如果数值变为 101 了，这说明，这条消息已经被持久化了，好友一定会得到它。如果没有就不断重试，直到数值变为 101，多次发送 101 号消息，对系统是没有负作用的。

具体的消息处理会有很多细节问题，可以学习一下 TCP 的工作方式。

# 6 架构图

## 6.1 架构图




系统特点：

**无状态 逻辑服务器 消息网关 接入服务器 都是无状态的，所以可以"无限" 增加机器来提高性能，而且效率的提高是线性的。 100 台服务器支持 100 万用户，加到 1000 台就能支持 1000 万**

**幂等的 操作都是幂等的，关掉运行中的机器没有影响，不会有任何影响。简单的等待重做就可以了，客户端无论如何重试也不会有问题。**

**支持多设备的 多个客户端不会出现,一台收到了消息，另一台收不到的情况，多设备完全不**

会有任何问题。

## 6.2 层次

### 6.2.1 引导层

进程名称：router

目标: 完成一个分布式服务框架。其功能主要包括：服务动态寻址与路由,依赖分析与降级等。

实际上只完成最简单的服务动态寻址服务，单节点，restful 接口。提供最基本的注册和查询功能

它的作用就是当一台服务器上线，注册服务，告诉大家，我现在在线了。其他服务器可以通过它来查询自己需要的服务。

接口：

注册新服务器

http://$ip:$port/regist/(cache|gw|poster|localposter|redis|memcached)/$rgid/.*

查询已经注册的服务器

http://$ip:$port/server

### 6.2.2 接入层

进程名称:gateway

目标：对客户提供标准的 http 接口，完成对外服务。所有的客户端，只能通过和接入层交互，完成所有的操作

接口：

获取用户信息

http://$ip:$port/api/user/$userid

结果返回一个 Json 对象，包含用户所有的信息

对好友发送消息

http://$ip:$port/api/c/$userid/$friendid/$sendid/?message=$message

结果返回一个 200 OK

### 6.2.3 业务层

6.2.3.1 进程名称: localposter

目标：完成所有消息业务处理

接口: golang RPC

func (t *LocalPosterAPI) PosterMessage(Req *GeneralMessage, id *uint64) error

6.2.3.2 进程名称: cacheserver

目标：完成用户信息查询

接口: golang RPC

func (t *CacheAPI) GetUserInfo(id *uint64, u *UserInfor) error

所有提供的接口，只是保证我拿到了你的请求，但不保证正常处理。处理的结果如何，都是靠异步来查询的

### 6.2.4 传输层

5.2.4.1 进程名称: poster

目标: 完成消息在不同 Region 之间的传输，类似路由器

接口: golang RPC

func (t *PosterAPI) PosterMessage(Req *GeneralMessage, id *uint64) error

同样也只是异步接口

### 6.2.5 持久层

直接使用 Redis

目标： 完成数据的存储和读取，利用了 watch，Exe 功能，保证读写是原子的。

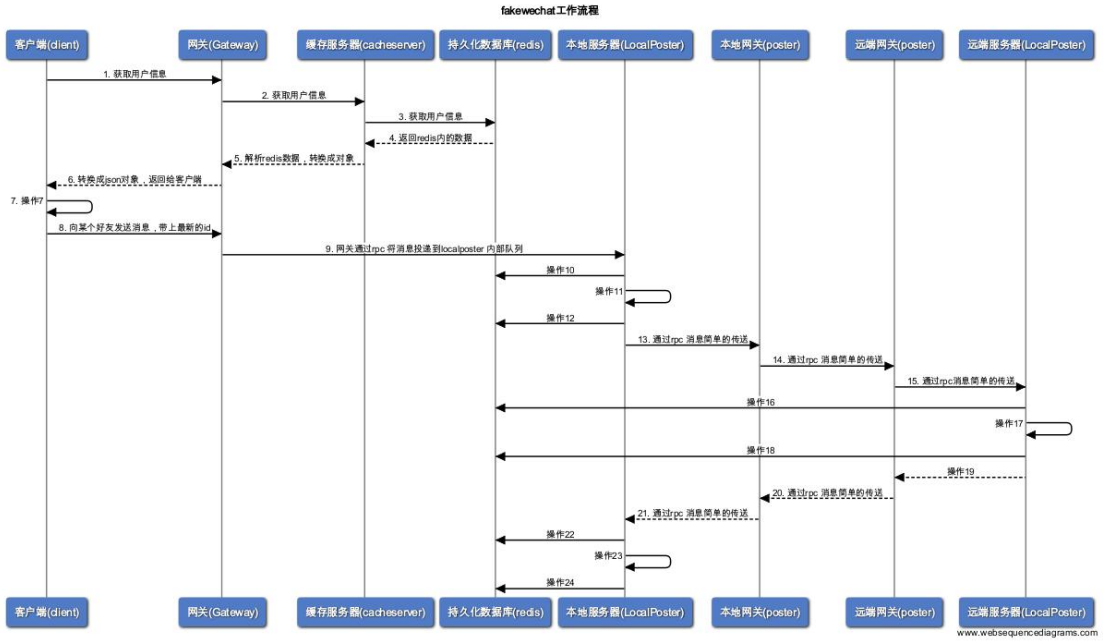
改进: 需要做一个服务器作为代理，蔽掉原始的 Redis 接口，保证可以迁移到别的持久化数据库

### 6.2.6 监控

我写了一个非常简单的监控服务器，所有的服务器内部都内置了一些监控点和计数器，会将状态数据发往监控服务器。这样我们可以知道，在运行的过程中，到底情况如何。在一个分布式式系统里，监控非常,非常，非常重要！

## 6.3 详细工作流程

# 发送消息流程




操作 7:  根据结果，计算下一个消息的 id, 对比本地的消息 id,计算有多少新消息

操作 10: 读取消息发送者的用户信息，同时对消息发送者的用户的信息展开 watch, 保证操作是原子的

操作 11: 对消息进行处理，如果是 id 是新的则处理，消息发送者的发送消息 id 增加 1 否则抛弃

操作 12: 新消息则存入持久化数据库，同时更新用户的信息，如果操作失败意味非原子，需要重回第 10 步重做

操作 16: 读取消息接受者用户的信息，同时对消息接受者用户的信息展开 watch, 保证操作是原子的

操作 17: 对消息进行处理，如果是 id 是新的则处理，同时接收者的接受消息 id 增加 1， 其他情况抛弃

操作 18： 新消息则存入持久化数据库，同时更新消息接受用户的信息，如果操作失败意味非原子，需要重回第 15 步重做

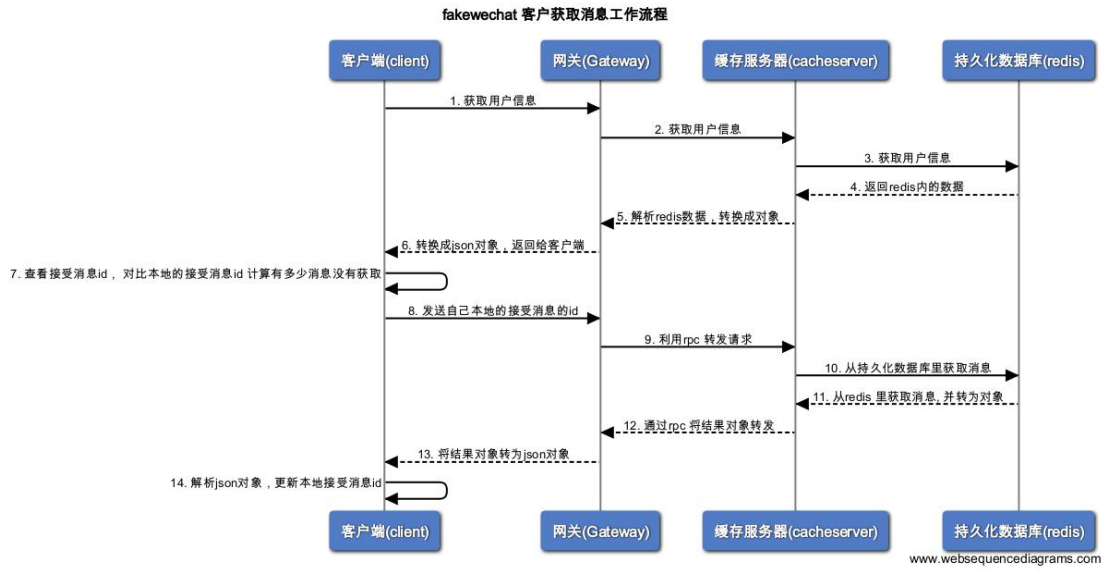
操作 19： 发回一个同样 id 号的 ack 消息,接收者是 原消息的发送者

操作 22: 读取发送用户的信息，同时对用户的信息展开 watch, 保证操作是原子的

操作 23: 确认 ack 是新的 id, 发送用户的 ackid 增加 1 如果已经过时就丢弃

操作 24: 更新发送用户信息，如果失败就重回 22 步

# 接受消息流程




新消息的到来，可以由服务器 push,但是工作原理是一样的。

# 6. 代码分析

我知道你们也不会看的，略

代码思想就是每个纤程无状态，每个纤程都有队列保护，不至于过载。超过负载就丢弃。

# 7. 实践以及测试过程

1. 系统估算

仅考虑个人对个人的情况。以为个人的好友关系为例，好友关系大概在 150 左右。我参考了自己日常的信息发生次数为 150 左右。但是每天交流的人数在 20 人以下。

所以我将模拟系统存在 1000 个用户，每位用户拥有 30 位好友，每位用户对自己的每一位好友发送 5 条消息。 一共是 1000*30*5 = 15 万条消息。 实践测试系统 将检测所有的消息都被稳定的发送和接收成功。 并且通过添加主机的办法，验证系统是可以水平扩展的。

（估算一下考虑下微信的系统，每天的消息量在线 1.2 亿*150 可能在 300 亿以上，而且消息的发送是有峰值的，那么峰值在???? 以上）

2. 计算实践系统的计算能力

下面是一些测试硬件的参考数据。

Redis 的每秒读写次数 1.2w/s 参考普通 mysql 3000 左右的 qps，redis 性能非常优异。但是也存在着对象需要序列化的问题。

一个 pb 对象的反序列化时间 0.14ms 左右

一个 pb 对象的序列化时间 0.07ms 左右

这几项数据大概描述了这个硬件的处理能力，很弱。 ：（ 首次测试，单机完成这个测试的时间为 182 秒。单机每秒能完成的消息数目只有 740+. 所以如果系统能水平扩展的话，

可以将每秒处理能力提高 至少 1 倍，将处理时间缩短到 110 秒左右。

测试流程：

数据准备

1. 首先进入 bin 目录，编译所有需要的服务器

   运行./build.sh 即可

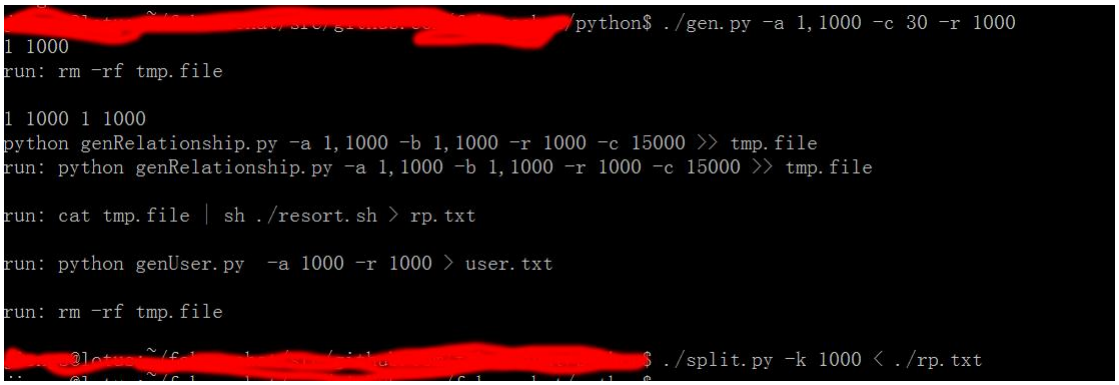

   这样就将所有需要的可执行文件生成了

2. 进入 python 目录，创建用户之间的关系链

   ./gen.py -a 1,1000 -r 1000 -c 30

   这表示我创建了 1 到 1000 用户的好友关系，分成了 1 个 Region,平均每个人有大概 30 个好友

输出大概是这样的




   将新产生的文件 1.txt 复制进 bin 目录

4. 进入 Redisconf 目录 启动 Redis

   Redis 是单线程程序，所以将它们绑定到单个核心上会更有利

   taskset -c 0 Redis-server ./Redis1.conf
   taskset -c 1 Redis-server ./Redis2.conf
   taskset -c 2 Redis-server ./Redis3.conf
   taskset -c 3 Redis-server ./Redis4.conf

5. 进入 bin 目录 启动所有的服务

   ./start.sh

   脚本将自动将数据载入 redis 脚本大概是这样的

```
redis-cli -p  1501 flushall
./savetoredis.py -m 10.29.101.3 -p 1501 < 1.txt
```

启动服务， 启动服务的操作，可以通过 listcmd.py 这个脚本自动生成，当然你需要根据你的实际情况进行一些修改。

运行的脚本大概是这样的。

```
nohup ./router  >/dev/null &
nohup ./monitorserver  >/dev/null &
nohup  ./cacheserver -hostid=cal -servertype=cache -listenaddress="10.29.101.3" -listenport=9601 -rgid=1 -process=ca  -routeserverurl="http://10.29.101.3:8089/server/"   -routeregisturl="http://10.29.101.3:8089/regist/"  -rgsize=1000  -monitorhost="10.29.101.3:8000"  >/dev/null &
nohup  ./gateway -hostid=gw2 -servertype=gw -listenaddress="10.29.101.3" -listenport=9501 -rgid=1 -process=gw  -routeserverurl="http://10.29.101.3:8089/server/"   -routeregisturl="http://10.29.101.3:8089/regist/"  -rgsize=1000  -monitorhost="10.29.101.3:8000"  >/dev/null &
nohup  ./poster -hostid=poster3 -servertype=poster -listenaddress="10.29.101.3" -listenport=9801 -rgid=1 -process=poster  -routeserverurl="http://10.29.101.3:8089/server/"   -routeregisturl="http://10.29.101.3:8089/regist/"  -rgsize=1000  -monitorhost="10.29.101.3:8000"  >/dev/null &
nohup  ./localposter -hostid=local4 -servertype=localposter -listenaddress="10.29.101.3" -listenport=9704 -rgid=1 -process=local  -routeserverurl="http://10.29.101.3:8089/server/"   -routeregisturl="http://10.29.101.3:8089/regist/"  -rgsize=1000  -monitorhost="10.29.101.3:8000"  >/dev/null &
```

简单的介绍一下
第一行 nohup ./router > /dev/null & 是简单的让进程在后端运行，将输出写入/dev/null 设备。
router 就是那个路由程序，它主要负责服务的注册和发布
第二行 monitorserver 就是监控所用的程序，它主要将系统内所有的运行状态进行记录
第三行 是 cacheserver 它主要负责 redis 内数据的读取
第四行 是网关 它提供标准的 web 接口，它是整个系统对外的接口
第五行 poster, 是消息内部转发的服务器
第六行 localposter 是真正业务处理的机制

因为 redis 也是系统的一部分，它不能自己注册服务，所以需要手动帮它注册，这样依赖它的服务器才能找到它。在通常情况下，可以用本地配置文件解决，但是这不太方便。大型系统都有自动的配置中心服务，我这个是一个最简陋的做法了。

运行下面的语句完成注册

```
curl -kvv "http://10.29.101.3:8089/regist/redis/1/?id=res1&host=10.29.101.3&port=1501&cellid=1"
```

开始测试

在一台主机上运行以下脚本即可

```
./client -host 10.29.101.3 -port  9501  -minid  1  -maxid  1000
```

最后的结果大概是这样

```
995 user finished 995
996 125 125 125
996 user finished 996
997 150 150 150
997 user finished 997
998 185 185 185
998 user finished 998
999 170 170 170
999 user finished 999
1000 170 170 170
1000 user finished 1000
spend  182
```

整个操作大概花了 182 秒，大约处理了 15 条消息的发送，处理和接受。总数约为 45 次

但是这样并不能获得我们想要的东西，可扩展性，如果加一台服务器结果会如何？如果不能以线性的扩展处理能力，那么这个系统就是失败的。所以我们简单的再加一台服务器。
用这样的命令 同时在 2 台服务器上

```
~/fakewechat/src/github.com/fakewechat/b./client -host 10.29.101.3 -port  9501  -minid  1  -maxid  500
```

```
./client -host 10.29.101.7 -port  9501  -minid  501  -maxid  1000
```

将负载简单的分到了 2 个服务器上，每台服务器各处理 500 个用户的业务

```
497 user finished 497
498 200 200 200
498 user finished 498
499 110 110 110
499 user finished 499
500 160 160 160
500 user finished 500
spend  117
```

```
995 200 200 200
495 user finished 995
996 125 125 125
496 user finished 996
997 150 150 150
497 user finished 997
998 185 185 185
498 user finished 998
999 170 170 170
499 user finished 999
1000 170 170 170
500 user finished 1000
spend  124
```

可以看到第二次 2 台服务器的处理时间缩短到了 117 秒和 124 秒，不到 50%. 但是也有明显的提高。

8. 监控运行数据分析

只是做这样的实践，我们能获得到什么呢？

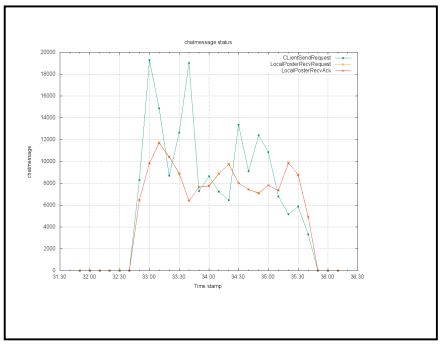
当然是监控数据，我用 gnuplot 编写了最简单的数据可视化代码，帮我了解系统的运作情况

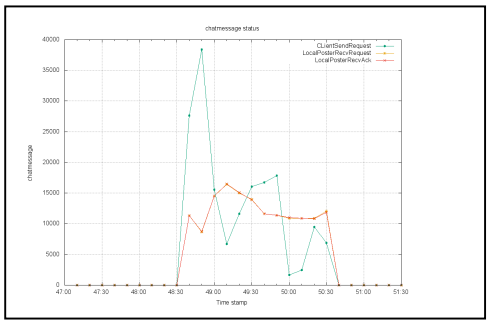
在 bin 目录下 运行 parselog -log log.txt，并运行 gunplot draw.plt

将在目录里生成一些图片，这些图片可以展示系统的运行情况

对前后 2 次的处理能力进行分析

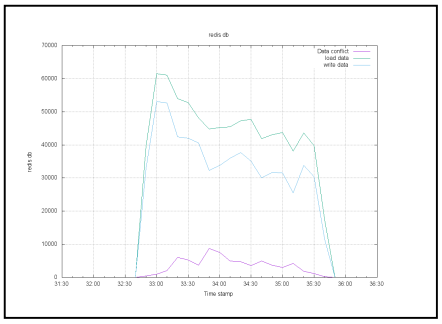
逻辑服务器处理的各类消息变化情况

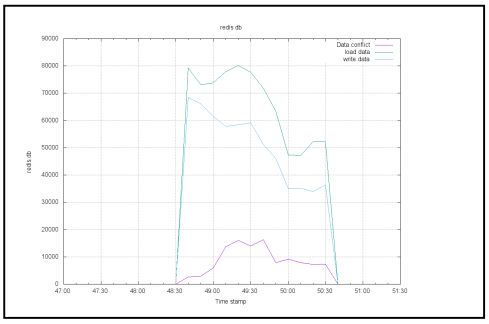



这里可以看到 2 台服务器的峰值能到 4 万，而单机只有 1.8 万。同时也可以看出在削峰这方面，系统还有很大的欠缺

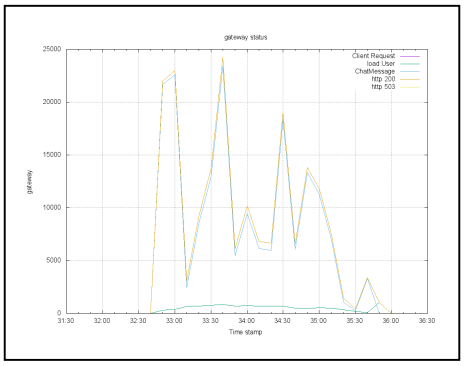
redis 服务器的负载情况

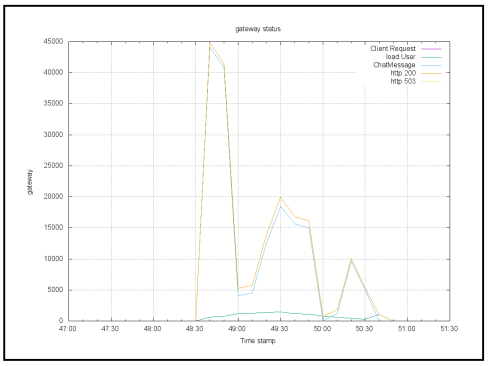



这是 redis 的操作曲线，从最初的测试可以得知 redis 的负载能力在 12k qps 左右（单线程）pipeline 在这里并无实际意义。可以看到单机的频率大概是 600, 而双机是 800, 同时可以看到双机的情况下，数据冲突的情况也更加明显，这也是为什么不能达到线性下降的原因之一。

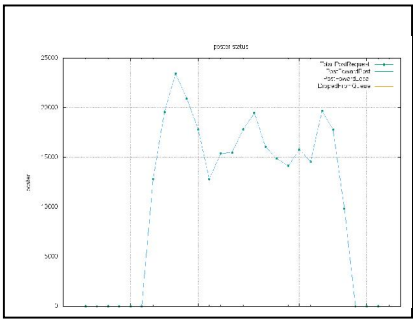
网关的负载情况

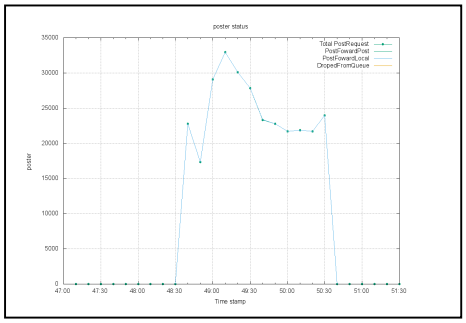



网关的处理能力从 2.5 万 升到 4.5 万，有提高

poster 服务器的处理情况




有一些提高，转发能力从 2 万到了 3.5 万。

从日志里还可以挖掘出很多，有用的信息，指导做系统优化，当然用 zabbix 来实时监控数据也没有问题，脚本已经准备好了。

日志文件看上去是这样的


具体的细节挖取。

name:"GetRequest" timestamp:1458402690 servername:"local4" servertype:"localposter" absolutevalue:253321
name:"GetRequest_time" timestamp:1458402690 servername:"local4" servertype:"localposter" absolutevalue:71715733476

将一条消息从系统里获取出来需要 0.28 毫秒。这 0.28 毫秒包含 redis 的读取和反序列化的时间

name:"StoreRequest" timestamp:1458402690 servername:"local4" servertype:"localposter" absolutevalue:329376
name:"StoreRequest_time" timestamp:1458402690 servername:"local4" servertype:"localposter" absolutevalue:105441499384

将一条消息存储进系统的时间为 0.32 毫秒，包含序列对象和 redis 存储的时间

name:"rpc_send_time" timestamp:1458402690 servername:"local4" servertype:"localposter" absolutevalue:102540007263
name:"rpc_send" timestamp:1458402690 servername:"local4" servertype:"localposter" absolutevalue:200038
一次 golang 的 rpc 需要 0.52 毫秒，god 的序列化对象的速度并不快

name:"CLientSendRequest_time" timestamp:1458402690 servername:"local4" servertype:"localposter" absolutevalue:366212620457
name:"CLientSendRequest" timestamp:1458402690 servername:"local4" servertype:"localposter" absolutevalue:67787
一个普通的消息请求 需要 0.5 毫秒才能完全处理完成，非常的慢。解决的办法是加大线程

并发能力，改进流程，因为流程中有多次数据的读取和写入非常影响了速度，同时也要避免数据冲突。当然最简单的办法 就是换一颗更强力的 CPU！这 2 台服务器都是 7 年以前的产品了性能可能比较低下。

# 8. 回顾以及目标

本文主要是实践了构建一个可扩展，通信系统的各个最基本的层次。用代码实现了自己的一些构想。作为一个玩具级别的项目，我觉得它达到了预想的目标，但是还存在很多问题

还有下面的问题要解决：

1) 性能不高，一个消息从发生到最终处理大概需要做 6 次序列号和反序列化
   用户数据需要做 3-4 次序列化和反序列化，甚至更多
2) 代码没有考虑大多数异常和安全问题, 缺乏足够的单元测试。
3) 缺乏真正的多副本 高可用的机制
4) 缺乏 sharding 机制
5) 缺乏负载迁移的机制
6) 缺乏大规模分布式系统的跟踪系统. 类似阿里的分布式调用监控系统 鹰眼，如果要真正的上线运作，这也是必须的。可以监控整个业务链的
7) 灰度上线，回滚 监控
8) 代码优化
9) 系统动态调度

.....

目标: 一个真正高可用的，高性能，可扩展的通信系统。路还很长，加油!!

欢迎在 http://www.github.com/xiaojiaqi/fakewechat 提出你的意见和代码！ 一起努力把项目做好。 谢谢

Max

2016.2