**密级：保密期限：**



**硕士学位论文**



**题目：企业移动协同云办公系统**

**——工作圈的设计与实现**

**学 号： 2011127136**

**姓 名： 赵炜**

**专 业： 软件工程**

**导 师： 吴国仕**

**学 院： 软件学院**

**2016年 6月10日**

目 录

[第一章 引言 1](#_Toc455229027)

[1.1 背景 1](#_Toc455229028)

[1.2 课题任务 2](#_Toc455229029)

[1.2.1 课题内容 2](#_Toc455229030)

[1.2.2 本人承担任务 3](#_Toc455229031)

[1.3 论文结构 3](#_Toc455229032)

[第二章 关键技术介绍 4](#_Toc455229033)

[2.1 分布式系统 4](#_Toc455229034)

[2.1.1 分布式系统的定义 4](#_Toc455229035)

[2.1.2 CAP定律 5](#_Toc455229036)

[2.1.3 现代分布式系统的特点 5](#_Toc455229037)

[2.2 开源技术 7](#_Toc455229038)

[2.2.1 Thrift — RPC Framework 7](#_Toc455229039)

[2.2.2 分布式服务框架 Zookeeper 7](#_Toc455229040)

[2.2.3 服务器缓存服务 Redis 8](#_Toc455229041)

[2.2.4 NoSQL数据库 MongoDB 9](#_Toc455229042)

[2.2.5 高性能HTTP服务器Nginx 9](#_Toc455229043)

[2.3 本章小结 10](#_Toc455229044)

[第三章 工作圈的需求分析 11](#_Toc455229045)

[3.1 系统用户角色分析 12](#_Toc455229046)

[3.2 系统功能需求分析 13](#_Toc455229047)

[3.2.1 用户账户管理与企业管理 13](#_Toc455229048)

[3.2.2 圈子与帖子 14](#_Toc455229049)

[3.2.3 评论 15](#_Toc455229050)

[3.2.4 赞 15](#_Toc455229051)

[3.2.5 工作功能 16](#_Toc455229052)

[3.2.6 工作流引擎 17](#_Toc455229053)

[3.2.7 即时通信（IM） 17](#_Toc455229054)

[3.3 系统的非功能需求分析 17](#_Toc455229055)

[3.3.1 数据存储持久性 17](#_Toc455229056)

[3.3.2 数据可迁移性 18](#_Toc455229057)

[3.3.3 数据私密性 18](#_Toc455229058)

[3.3.4 服务可用性 18](#_Toc455229059)

[3.3.5 故障恢复能力 19](#_Toc455229060)

[3.4 本章小结 19](#_Toc455229061)

[第四章 工作圈的总体设计 20](#_Toc455229062)

[4.1 工作圈的设计目标 20](#_Toc455229063)

[4.2 工作圈功能模块设计 20](#_Toc455229064)

[4.2.1 功能总体划分 20](#_Toc455229065)

[4.2.2 用户账户模块 23](#_Toc455229066)

[4.2.3 圈子与帖子 23](#_Toc455229067)

[4.2.4 评论 24](#_Toc455229068)

[4.2.5 赞 24](#_Toc455229069)

[4.2.6 工作应用 24](#_Toc455229070)

[4.2.7 工作流引擎 25](#_Toc455229071)

[4.3 数据库设计 25](#_Toc455229072)

[4.3.1 用户账户模块数据设计 26](#_Toc455229073)

[4.3.2 圈子帖子模块数据设计 26](#_Toc455229074)

[4.3.3 评论模块数据设计 27](#_Toc455229075)

[4.3.4 赞模块数据设计 28](#_Toc455229076)

[4.3.5 工作应用模块数据设计 28](#_Toc455229077)

[4.3.6 工作流引擎 29](#_Toc455229078)

[4.4 工作圈技术框架设计 30](#_Toc455229079)

[4.4.1 工作圈整体技术架构图 30](#_Toc455229080)

[4.4.2 工作圈Rest设计 30](#_Toc455229081)

[4.4.3 分布式服务与自动化配置中心 32](#_Toc455229082)

[4.4.4 数据分片(Data Sharding)与数据索引中心 35](#_Toc455229083)

[4.4.5 工作圈部署架构 37](#_Toc455229084)

[4.5 本章小节 37](#_Toc455229085)

[第五章 工作圈功能模块的设计与实现 39](#_Toc455229086)

[5.1 客户端界面设计 39](#_Toc455229087)

[5.1.1 用户账户界面设计 39](#_Toc455229088)

[5.1.2 圈子与帖子界面设计 40](#_Toc455229089)

[5.2 服务详细功能设计 41](#_Toc455229090)

[5.2.1 用户账户模块详细功能设计 41](#_Toc455229091)

[5.2.2 圈子帖子模块详细功能设计 43](#_Toc455229092)

[5.2.3 评论模块详细功能设计 44](#_Toc455229093)

[5.2.4 赞系统模块概述 44](#_Toc455229094)

[5.2.5 工作流引擎模块概述 44](#_Toc455229095)

[5.3 本章小结 45](#_Toc455229096)

[第六章 结束语 46](#_Toc455229097)

[6.1 论文工作总结 46](#_Toc455229098)

[6.2 问题和展望 46](#_Toc455229099)

# 第二章 关键技术介绍

本章将针对在该论文中涉及的分布式系统的思想与运用到的核心技术进行介绍。运用到的核心技术主要包括：Apache的两个顶级开源项目Zookeeper与Thrift，还有Redis服务与Mongo数据库。这些技术都是在最近十年的移动互联网中，应对上亿用户的高并发而开发的组件。

## 分布式系统

分布式系统并不是什么新鲜词，20世纪80年代之前就已经有出现了很多种不同类型的分布式系统。分布式系统正真普及与出彩的却是在互联网时代。谷歌、亚马逊、Facebook等行业领先的互联网公司将分布式系统推到了一个新的高度。谷歌的GFS文件系统、MapReduce工具都是构建于分布式的理念上，现在的大数据分析正是建立在这两个软件的思想之上。Apache旗下的Zookeeper、Spark、Kafka等分布式系统，简化了构建分布式计算的门槛，让更多企业客户体会到了分布式系统的好处。

### 分布式系统的定义

分布式系统是一个其硬件或软件组件分布在连网的计算机上，组件之间通过传递消息进行通信和动作协调的系统。这个简单的定义覆盖了所有可有效部署连网计算机的系统。[6]

由一个网络连接的计算机可能在空间上的距离不等。它们可能分布在地球上不同的洲，也可能在同一栋楼或同一个房间里。现代定义的分布式系统有如下显著特征：

并发：现代的软件编写中，多线程处理执行并是是很常见的方式。现在的CPU核心处越来越多，为了能充分利用CPU的运算能力，多线程并发是一种很好的方式。在计算机网络中也是同样的，不同用户的请求分配到不同的服务器，不同的CPU同时处理，系统的处理能力就会随着服务器数据增加而提高。

全局时钟：分布式服务之间的协作是通过消息来过时行的。服务与服务之间没有一个准备而有效的全局时钟来保证服务器之间的同步状态。

故障独立性：服务器都会有一定的概率发生故障，系统在设计的时候需要考虑到各种各格的故障情况以及应对策略。网络上的分布式系统是以多个服务器节点构成的，当一个节点故障时，并不代表其它的节点同时也发生了故障，同时正在运行的节点也不会很快的了解到故障发生了。

看一下现代分布式系统的几个例子，包括Web搜索、多人在线游戏和金融交易系统，也考察今天推动分布式系统发展的关键趋势：现代网络的泛在特性，移动和无处不在计算的出现，分布式多媒体系统不断增加的重要性，以及把分布式系统看成一种实用系统的趋势。接着本章强调资源共享是构造分布式系统的主要动机。资源可以被服务器管理，由客户访问，或者它们被封装成对象，由其他客户对象访问。

构造分布式系统的挑战是处理其组件的异构性、开放性（允许增加或替换组件）、安全性、可伸缩性（用户的负载或数量增加时能正常运行的能力）、故障处理、组件的并发性、透明性和提供服务质量的问题。

### CAP定律

在理论计算机科学中，CAP定理（CAP theorem），又被称作布鲁尔定理（Brewer's theorem），它指出对于一个分布式计算系统来说，不可能同时满足以下三点：

• 一致性（Consistence) (等同于所有节点访问同一份最新的数据副本）

• 可用性（Availability）（对数据更新具备高可用性）

• 容忍网络分区（Partition tolerance）（以实际效果而言，分区相当于对通信的时限要求。系统如果不能在时限内达成数据一致性，就意味着发生了分区的情况，必须就当前操作在C和A之间做出选择。）

根据定理，分布式系统只能满足三项中的两项而不可能满足全部三项。理解CAP理论的最简单方式是想象两个节点分处分区两侧。允许至少一个节点更新状态会导致数据不一致，即丧失了C性质。如果为了保证数据一致性，将分区一侧的节点设置为不可用，那么又丧失了A性质。除非两个节点可以互相通信，才能既保证C又保证A，这又会导致丧失P性质。[7]

### 现代分布式系统的特点

1.对服务器硬件要求低

• 因为分布式服务器的故障是不可预见的，所以由一个设计良好的软件架构的容错机制来对分布式系统的可靠性做保证。

• 允许使用低配置，低性能的服务器来部署分布式的系统。现代服务器的性能已经是非常高了，但是有限的网络带宽在制约者分布式系统的性能，再快的服务器也需要等网络IO。

2.强调横向可扩展性

通过增加节点数来提升服务性能的方式叫横向扩展（Scale Out），通过升级每个节点的性能来提升服务的性能的方式叫纵向扩展（Scale Up）。现代CPU的主频已进到达了4GHz的门槛，想要提升需要花费巨大的代价。服务器整体的吞吐性能更多的是受制于网络与磁盘。所以模向扩展对系统的整体性能提升相对要明显的多，而且花费的成本就要低不少。增加服务器的数量要比研发一个高频率的CPU要容易。

3.不允许单点失效（No Single Point Failure）

单点失效是指，因为一个服务器节点的宕机导致的整个服务不可用的情况。如果这个服务仅运行了一份实例在一台服务器上，很容易就出来单点失效的情况。

之前已提到过，分布式的服务器允许使用低配置，低性能的机器来部署。那么服务器的可用性也很难保证完全可靠。在设计分布式系统时，将所有的节点会故障的可能都要考虑进去，服务节点要有冗余，数据库服务要有热备份等等。而且服务节点的冗余和数据的冗余来保证它的可靠性。

一般的，服务器的运行的状态不应该是满负载的。过高或是长时间的高负载会增加服务器的故障率。为了平衡单机上的负载，可以通过增加大量的点节来实现。这些节点的性能不要求很高，一但节点的基数提升了，在总负载不变的情况下平均到每一台机器上的负载就变低了。

4.分布式系统尽可能减少节点间通讯开销

如前所述，分布式系统的整体性能瓶颈在于内部网络开销。所以尽量让服务调用本地的数据减少网络开销，能显著提高服务的整体性能。Hadoop的MapReduce就是一个很好的例子。

5.无状态的分布式系统服务

应用服务的状态是指运行时程序因为处理服务请求而存在内存的数据。在服务有状态的时，这具服务也就成为了一个不满足非点单失效这一项。因为有状态的服务一但故障了，存在服务中的状态数据就会丢失，依赖这个数据的业务就会失败，但且不能被其它的节点代替，这样的服务明显不是高可用型的分布式服务。将数据保存在客户端或是持久化的存储中，将服务无状态化。这样服务就不需要担心数据丢失的情况了，同时也让分布式中的所有节点的行为一致，不再有单点失效的问题。

在设计后台服务的时候，用户的登录状态可以持久化到数据库中，也可以保存到缓存中去，不要保存到分布式服务器的内存中。这样就不会因为分布式服务节点的故障导致用户登录状态丢失的情况。

## 开源技术

随着开源社区的兴起，开源成为了计算行业中的一股中间力量。越来越多的优秀技术诞生于开源社区，成长于开源社区，开源社区给了程序以更多的选择。本章选择介绍Apache社区的两个顶级项目Thrift和Zookeeper，还有优秀的Redis与MongoDB。

### Thrift — RPC Framework

Apache Thrift 是 Facebook 实现的一种高效的、支持多种编程语言的远程服务调用的框架和二进制通讯协议。主要用来定义和创建分布式的远程服务。Thrift有优先有：相比类似于SOAP的协议，基于二进行格式的数据传输更加高效、快速；引入库代码少，没有编码框架，没有配置文件；Thrift的IDL对数据定义与各语言之间的翻译很自然，映射的很好；应用层与序列化层的通讯格式分离，独立修改；支持协议的向下兼容，定义良好的协议是可以向之前的版本兼容的。Thrift它采用接口描述语言定义并创建服务，支持可扩展的跨语言服务开发，所包含的代码生成引擎可以在多种语言中，如 C++，Java，Python，PHP，Ruby，Erlang，Perl，Haskell，C#，Cocoa，Smalltalk 等创建高效的、无缝的服务。Thrift还可以使用Java的NIO模式，极大的提高了服务整体的性能和吞吐能力。 [2]

### 分布式服务框架 Zookeeper

Zookeeper 分布式服务框架是 Apache Hadoop 的一个子项目，主要解决分布式系统中服务命名统一、服务状态同表以及服务配置项管理等问题。

Zookeeper 会维护一个具有树状层次关系的数据结构，它非常类似于一个标准的文件系统，如图2.2.1所示：

图 2.2.1 Zookeeper 数据结构[3]

Zookeeper服务的节点是基于路径标识的，每个节点可以有子节点在同一目录下的名称唯一。Zookeeper提供了对节点变化的监听器，使应用可以随时得到点节点变化的通知。工作圈的环境配置变量就是基于这个监听器来实现的。

### 服务器缓存服务 Redis

Redis 是一个开源（BSD许可）的，内存中的数据结构存储系统，它可以用作数据库、缓存和消息中间件。它支持多种类型的数据结构，如字符串（strings），散列（hashes），列表（lists），集合（sets），有序集合（sorted sets）与范围查询，bitmaps，hyperloglogs和地理空间（geospatial）索引半径查询。Redis 内置了复制（replication），LUA脚本（Lua scripting），LRU驱动事件（LRU eviction），事务（transactions）和不同级别的磁盘持久化（persistence），并通过Redis哨兵（Sentinel）和自动分区（Cluster）提供高可用性（high availability）。

可以对这些类型执行原子操作，例如：字符串（strings）的append 命令；散列（hashes）的hincrby命令；列表（lists）的lpush命令；集合（sets）计算交集sinter命令，计算并集union命令和计算差集sdiff命令；或者在有序集合（sorted sets）里面获取成员的最高排名zrangebyscore命令。

为了实现其卓越的性能，Redis 采用运行在内存中的数据集工作方式。根据实际的使用情况，系统可以每隔一定时间将数据集导出到磁盘，或者追加到命令日志中。同时也可以关闭持久化功能，将Redis作为一个高效的网络的缓存数据功能使用。Redis 同样支持主从复制（能自动重连和网络断开时自动重新同步），并且第一次同步是快速的非阻塞试的同步。[5]

### NoSQL数据库 MongoDB

MongoDB是一个开源文档数据库，提供高性能、高可用性和弹性扩展。

MongoDB的文档结构是若干由个key和value组成键值对构成的。MongoDB文档类似于JSON对象。字段的值可能包括其他文档、数组和数组的文档。在绝大多数的编程语言中，MongoDB中的文档(即对象)可以对应到本地数据类型。 文档和数组形式的数据组织方式减少非常耗损性能的表的外连接。

MongoDB提供高性能的数据持久性。对于嵌入式数据模型提供支持，减少了数据库系统的I / O活动。MongoDB索引支持从文档和可以包含密钥和数组更快的执行查询，支持丰富的查询语言，支持读写操作以及：数据聚合、文本搜索和地理空间查询。

MongoDB的复制集功能，称为副本集,提供：自动故障转移和数据冗余。

水平可伸缩性。MongoDB提供水平可伸缩性的核心功能:分片跨一组机器分布数据、标记清楚分片允许数据引导到特定的分片,如考虑地理分布的碎片、支持多种存储引擎。MongoDB支持多个存储引擎，如:WiredTiger存储引擎和MMAPv1存储引擎。此外,MongoDB提供可插拔存储引擎API，允许第三方制定MongoDB的存储引擎。

### 高性能HTTP服务器Nginx

Nginx是一款开源的高性能HTTP服务器，特点是：占有内存少；稳定性高；模块丰富，扩展性好；采用了Linux平台的epoll事件模型，可支撑大于2万以上链接的并发，并且有优异的吞吐能力。本文采用Nignx来做为工作圈云服务的代理网关做负载均衡，接受从移动端或是WEB端提交的请求进行分发。

## 本章小结

分布式系统是互联网时代下高并发高性能平台构建的主要构建方向。横向可扩展性（Scale Out）使得其构建系统异常灵活，能应对千万级别的用户并发使用，降低了系统对硬件的需求，同时满足了互联网应用的高可用特点。随着开源社区的蓬勃发展，开源的框架也越来越多。在开源的世界里有一句名言：不要重复发明轮子。要充分利用开源的力量来提高我们的工作效率与工作质量。本章所述的组件都是开源社区的优秀成果，本文论述的分布式框架设计，其思想与实现都是与这些开源的组件紧密结合的。

# 第四章 工作圈的总体设计

结合前两章的关键技术介绍与需求分析，本章将开始进行工作圈的总体设计阶段。这一阶段的主要目标是在满足功能性需求与非功能性需求的基础上完成云服务的整体架构设计。本章的主要内容，首先完成对企业协同办公云服务的业务设计，接下来针对服务的性能需求进行分布式框架的设计。

## 工作圈技术框架设计

### 工作圈整体技术架构图

图4.4.1工作圈整体技术架构图

### 工作圈Rest设计

工作圈的用户终端有三种：Android、iOS和PC Web。这三个端可以简单的区分成为PC端和移动终端。这样区分是因为这两种端对应的场景和对后台的数据交互要求都是不一样的。PC端的特点：运行在浏览器中（B/S结构），网络环境信号较强，数据传输几乎没有成本，终端的尺寸较大可以展示的内容丰富，终端更换的频率较高；移动端特点：原生的APP程序运行在移动智能操作系统中（C/S结构），容易处在弱网络环境中，数据传输有成本，终端的尺寸有限，终端不易更换。从对比来看，移动端需要轻量级的增量更新接口，这样意味着移动端的数据在从服务端下行之后需要自行缓存在终端。这样可以避免在弱网环境下程序请求不到数据，同时节省的数据重复传输的成本。PC端应用因为是B/S结构不便于本地缓存数据，所以在查询数据的时候接口应该设计成为全量分页性的接口，根据页码来下行查询数据。然而站在具体的功能业务上，同一个功能的业务逻辑却又是相同的，只是对于不同类型的终端提供的数据交互接口不一样。后台的服务就不能只是一个server的设计了，首先应该把业务逻辑封装成为一层，本文中先简称为server层。然后在server层之上封装一个rest层，给不同的终端提供不同的服务。

图4.4.2 终端交互示意图

1. **Nginx负载均衡**

Rest server是整个工作圈服务的表现层，是和工作圈移动端或是WEB端通信的一层。工作圈终端借由Rest Server这一层提供的Rest api，来和工作圈服务进行数据的交换。同样，在面对大规模用户高并发的访问时一台服务器的能力是有限的，云服务就需要多台服务器来分担压力。站在最前面的负载服务器就尤其重要，这里选择了前面介绍过的Nginx服务器来做为HTTP请求的代理网关与负载。

图4.4.3 Rest集群示意图

1. **无状态的服务设计**

其实提出这点反而是为了解决用户的登录状态问题，假设用户登录的请求发到了Rest A服务上，对用户的数据请求又发送到了Rest B服务上。那Rest B怎么能知道用户的登录状态呢。在以前的单点式系统中，有一个Session的概念。用户登录后建立在服务器，请求的时候服务端会查询这个Session，登出后销毁。但是在分布式的Rest server上主机之前内存的数据不共享，无法得知用户的Session状态。

工作圈的解决方式是登录后返回一个加密的token给终端。不是在服务端保存用户的登录状态，而是在用户使用的终端上保存这个token。在Android iOS系统上是由工作圈App软件保存在手机的数据库中，PC WEB因为没有数据库，将这个token保存在Cookie中。在每次请求的时候将WEB端会自动的带上Cookie供服务器查询，手机端的App需要程序从手机上读取这个请求加到Http Header中来便于服务检查。

### 分布式服务与自动化配置中心

模块划分完成之后，这个框架还不能被称为分布式框架。因为它还没有满足前面（2.1.3 现代分布式系统的特点）中提到的特性。下面来尝试进一步的满足这些特性。

1. **分布式服务远程调用（RPC）**

服务层的模块规划完成了，接着要解决Rest层对服务层的业务服务调用。在第二章介绍的Apache Thrift就是一个优秀的RPC框架。Thrift使用自己的IDL语言来定义远程服务的接口，然后使用Thrift工具生成基于定义的服务端接口以及可以调用服务的客户端Client。

前面介绍了工作圈的两层结构：REST、Server。Client就是REST与SEVER之间的桥梁。工作圈的Server层是分布式部署的，所以Client层这个桥梁就需要支持分布式的RPC调用。Client首先要通过ZkHelper来读取服务注册的情况，简单说就是拿到已经注册在Zookeeper中心的可用服务器IP列表。然后随机的从这个列表中挑出一个服务，尝试与其建立连接。连接成功就调用服务，失败说明服务是有问题的需要把这个问题IP从Zookeeper中删掉然后取列表中的下一个IP。这样就形成了一完整的分布式服务的负载与监控。

Client层封装Server层通过Thrift定义的RPC方法，将其暴露给其它的服务。需要使用模块的服务的时，引用相应的Client即可。

以用户账户服务gongzuoquan-account举例，Thrift会为开发者生成一个接口定义程序gongzuoquan-account-core。gongzuoquan-account-server在这个包的基础上实现它定义的服务接口；gongzuoquan-account-client在这个包的基础上封装的客户端的调用方法。gongzuoquan-account-rest通过引用client实现了访问server的目的，结构图如下。

 图4.4.4 Server-Rest通信示意图

1. **横向可扩展性（Scale Out）**

横向可扩展性（Scale Out）。意味着一个业务服务模块是可以部署很多个副本，这样就带来一个问题，如何保证所有的请求都能平均的分发到所有的服务上去？ 前面完成了REST与SERVER层的剥离与远程调用，在已知server服务的IP前提下就可以进行调用。在有多台服务实例情况下，简单的办法就是维护一个IP数组，然后随机的从数组中选出一个IP来进行调用。

这时就需要一个自动化配置中心。在一个模块服务启动时，将启动的服务器在网络上的IP地址注册到自动化配置中心服务中，这样就完成了服务状态的发布。然后任一客户端在需要调用这个服务的时候，先在自动化配置中心中读取一下这个服务的服务状态，取得了可以提供这个服务的服务器地址列表，然后再以一个算法随机的地址发起请求，完成一次调用。这样就满足了服务的横向可扩展性（Scale Out）。

Zookeeper是这个自动化中心实现的最佳选择。服务以一个约定的路径：/gongzuoquan/account/cluster/在Zookeeper服务中注册一个节点，然后把服务自身的IP地址写入到这个节点下，如下图：

图4.4.5 模块服务注册中心数据节点示意图

1. **不允许单点失效（No Single Point Failure）**

不允许单点失效（No Single Point Failure）。只要对自动化配置中心的功能稍加改动，便能满足这一特性。只要能将服务状态列表中的故障节点检测到，并从中剔出除们，客户端便不会受到影响。只要这个服务状态列表中仍有一台服务器上的服务是正常的，就会不造成整个服务失效。当然，这里也要考虑到这个自动化配置中心服务本身会出问题，需要保证这组服务也是高可用的服务。

对故障列表的检测，有两种方式：1.在启动的服务与自动化配置中心之间建立一个心跳，心跳消失即失效；2.从客户端来检测，如果客户端调用服务失败后，将其从自动化配置中心中剔除然后访问下一个服务地址。

第一种方式，因为心跳会有延时。如果在延时中发生问题，那么在这一段时间内就会有一定机率的请求会因为请求到故障节点上。（如下图4.4.3.1）第二种方式，需要做到客户端实时监听自动化配置中心的服务器状态变化，这样虽然会避免请求故障节点，但是在客户端数量过大的时候会大量的占用自动化配置中心服务的链接。工作圈目前的设计，客户端服务还是可控的，所以选择了第二种实时性较高的处理方式。

图4.4.6服务-配置中心心跳示意图

在实际的开发中，也可以将数据库和缓存服务的地址与链接配置存到Zookeeper中相应的节点下。当服务器的数据库或是缓存进行迁移数所据的时候，监听节点的变化，重新建立指向新服务的链接池就可以了。

### 数据分片(Data Sharding)与数据索引中心

大数据时代，数据的持久化性能会影响服务的整体的性能。如果一个模块下的业务都存储在一个表中，那么随着数据量的上升性能必然会下降。解决的办法是对数据库中的数据进行水平扩展(分表，分库等)，控制住单表中的数据量不会超过一个阀值即可。这个阀值的界定不是固定的，会根据数据库的选择以及运行的环境不同而不同。比如系统需要数据库插入一条业务数据的响应时间要在1ms以内，需要在实际的运行环境中对数据库进行一个insert压力测试，得到在多少条数据的情况下单表的插入响应时间超过了1ms（这个时间需要根据业务场景来设定），那么这个阀值就是估算的一个单表数据量上限的指标。假设这个值是f，需求预估的整个系统容量是m，那个分表数t = m / f。如果数据库服务器不只一台，那么可以再将数据散列到不同的数据库上去，此时假设有n台数据库服务，那么t = m / (f \* n)。

对数据进行分表，分库的插入的过程称为数据分片(Data Sharding)。常见的分片策略有以下几种：

根据ID特征：例如对记录的ID取模，得到的结果是几，那么这条记录就放在编号为几的数据分区上。

根据时间分区：例如以年/月为单位，某一年/月中产生的数据放在一起。

基于索引表：根据ID先在一个复合索引中查到分区信息，然后再到分区中去查找具体的数据。

这些分片的策略没有好坏之分，实际应用中需要根据具体的业务需求或是数据特征来选择设计。需要注意的是：数据分片不是银弹，它对系统的性能和伸缩性（Scalability）带来一定好处的同时，也会对系统开发带来许多复杂度。例如，有两条记录分别处在不同的分区中，如果有另一条数据与这两条记录其中之一有关联，那么关联的信息就需要在这两个分区中各保存一次。在对数据完整性有要求的场合，也就是对数据操作需要事务的情况下，跨分区的事务会变成“性能杀手”。另一方面，数据分片对全局扫描的需求也没有带来提升，反而增加了扫描的复杂度。

工作圈的数据分片使用了ID取模和索引表两种的方式，但是对这个策略进行了改进。首先引入一个gongzuoquan-idcenter的服务，用来随机生成数据在工作圈服务内全局唯一的主键id。这样，id在进行取模分片的时候就可以较为均匀的分布在所有的分表中。同时，为了解决跨区数据查询的问题，服务需要一个数据索引中心（类拟于Hadoop的nameService的概念)。相一组有关系的数据id索引在一起保存起来，在查询的时候通过这个索引可以得到这些数据的id，然后再通过id取模来得到分库分表的数据，从而查询到需要的业务数据。工作圈中承担这部份工作的核心服务是gongzuoquan-idlist。

### 工作圈部署架构

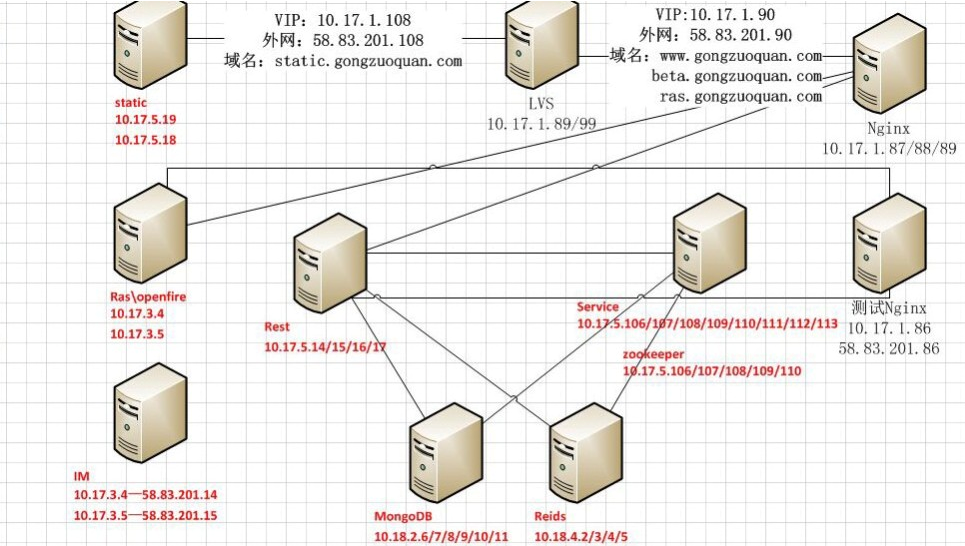


图4.4.7工作圈部署网络拓扑图

工作圈完整的部署架构如图4.4.7所示。工作圈主体的服务是由Rest和Server层来承担的。为了提高性能，Rest和Server层都有不止一台的服务器来提供服务，每一台Rest服务器中都部署着工作圈所有Rest模块，同样Server中的每一台服务器也都部署着工作圈所有的Server模块。这样部署，使得整个系统中只要有一个Rest服务器和Server服务器还在工作，都可以继续为用户提供服务。集群中任意一个节点的故障都不会影响到系统的完整功能，仅会影响整个集群的性能。

Nginx承担着网关与Rest层服务的负载，图中可以看到Rest集群一共层部署了4台主机，Server集群一共部署了8台主机。注册配置中心用到的Zookeeper集群服务与Server集群共享了5台主机。MongoDB集群使用了一个3×2的方案，集群使用了三组服务做分布式数据存储，每一组服务都是由Master/Slave/仲裁节点做的高可用配置。

## 本章小节

本章重点描述了工作圈的Rest server层、模块化的服务层的设计。以前一章的需求作为入口，对工作圈的服务进行了模块的划分，并且对公共功能的模块进行了封装设计，减少开发人员的工作量。结合第二章的分布式思想为Rest和Server层设计了负载均衡的解决方案。同时也对持久化的数据存储进行了数据分片的设计，以期能够在大量用户并发的场景下能有高性能的表现。

综上所述为本章的核心内容，本章虽然没有具体提及某个功能的具体设计与实现，但从宏观的角度出发以系统运转主线为基础简明的描述了整个系统的业务逻辑，在后面的章节中将选择这些逻辑或模块中的核心内容作为重点阐述。