

# 山东大学

# 毕业论文(设计)

论文（设计）题目：

移动应用后端服务框架的设计和实现

姓 名	鞠强
学 号	200900301084
学 院	软件学院
专 业	软件工程
年 级	2009 级
指导教师	闫中敏

2013 年 5 月 27 日

山东大学本科毕业论文（设计）成绩评定表(指导教师用)

学院	软件学院			年级、专业	软件工程	
学生姓名	鞠强			学号	200900301084	
毕业论文（设计）题目： 移动应用后端服务框架的设计和实现						
指导教师 评语及成 绩评定	成绩 评定 标准	评分项目			分值	得分
		选题质量 20%	1	符合专业培养目标	6	
			2	注重与学科发展或 社会要求结合	6	
			3	拟解决的关键问题 明确	8	
		能力水平 50%	5	查阅文献资料能力	10	
			6	研究方案的设计能 力	10	
			7	研究方法和手段的 运用能力	10	
			8	外文运用能力	10	
			9	分析问题和综合运 用知识的能力	10	
		成果质量 30%	10	写作水平	10	
			11	写作规范与学术道 德	5	
			12	篇幅	5	
			13	成果的理论或实际 价值	10	
	指导教师评语：					
	成绩： 指导教师签名：年 月 日					

山东大学本科毕业论文（设计）成绩评定表（答辩用表）

学院	软件学院			年级、专业	软件工程					
学生姓名	鞠强			学号	200900301084					
毕业论文（设计）题目： <div>移动应用后端服务框架的设计和实现</div>										
答辩小组评语及成绩评定	成绩评定标准	评分项目			最高分	评分				
		1	论文（设计）质量水平	论文（设计）结构严谨，逻辑性强，有一定的学术价值或实用价值，有创新点；文字表达准确流畅，论文格式规范；图表、图纸规范，符合规定要求	50	>46	40-45	35-39	30-34	<30
		2	论文（设计）质量报告水平	思路清晰；概念清楚；语言表达准确；报告在规定时间内完成	20	>18	16-18	14-15	12-13	<12
		3	答辩情况	回答问题有理有据，基本概念清楚。主要问题回答准确、有深度	30	>27	24-26	21-23	18-20	<18
	答辩小组评语：									
	答辩成绩：									
	答辩组长、成员（签名）： <div>年 月 日</div>									
论文（设计）总成绩	论文（设计）总成绩： 论文（设计）成绩等级：									

注：1、论文（设计）总成绩=指导教师评定成绩（50%）+答辩成绩（50%）

2、论文（设计）成绩等级：优秀（90分以上），良好（80-89分），中等（70-79），及格（60-69），不及格（60分以下）

# 目 录

目 录.....	- 4 -
摘要 .....	- 1 -
Abstract.....	- 2 -
第 1 章 绪论.....	- 3 -
1.1 课题背景.....	- 3 -
1.2 移动应用开发技术的发展.....	- 3 -
1.3 开发者面临的问题.....	- 3 -
1.4 “后端即服务”所提供的服务和解决方案.....	- 4 -
1.5 本文的组织结构和创新点.....	- 4 -
第 2 章 技术背景及相关知识简介.....	- 5 -
2.1 开发环境.....	- 5 -
2.1.1 Ruby 语言简介.....	- 5 -
2.1.2 Mongodb 简介.....	- 6 -
2.1.3 Android SDK.....	- 6 -
2.1.4 Cocoa SDK.....	- 6 -
第 3 章 移动开发后端即服务行业现状.....	- 8 -
3.1 移动开发分工细化催生了后端即服务.....	- 8 -
3.2 当前后端即服务的服务内容概述.....	- 8 -
3.3 分析几个典型的移动后端服务厂商.....	- 8 -
3.3.1 StackMob.....	- 8 -
3.3.2 Parse.....	- 11 -
3.3.3 国内移动后端服务现状.....	- 12 -
3.4 行业现状小结.....	- 14 -
第 4 章 设计和实现一个移动应用后端服务框架.....	- 16 -
4.1 本章概述.....	- 16 -
4.2 系统设计原则.....	- 16 -
4.3 总体结构介绍.....	- 16 -
4.3.1 数据存储服务总体结构.....	- 17 -
4.3.2 统计服务.....	- 18 -
4.3.3 通用逻辑处理.....	- 19 -
4.4 关键技术讨论.....	- 22 -
4.4.1 搭建分布式数据库集群.....	- 22 -
4.4.2 构建 RESTFUL 的 Mongodb 接口.....	- 30 -
4.4.3 使用 HMAC 方式提供高效率的认证.....	- 34 -
4.4.4 服务器的选择和使用反向代理的方式部署应用服务器.....	- 35 -
4.4.5 基于 Countly-Server 构建统计服务器.....	- 38 -
4.4.6 Mobile First 的移动端 SDK 设计.....	- 42 -
第 5 章 通用性和可用性说明.....	- 50 -
5.1 通用性.....	- 50 -
5.2 可用性.....	- 50 -
第 6 章 结论.....	- 52 -
6.1 系统设计和实现的总结.....	- 52 -

6.2 移动应用后端服务发展展望.....	- 52 -
致 谢.....	- 53 -
参考文献.....	- 54 -
附录 .....	- 55 -

# 移动应用后端服务框架的设计和实现

## 摘要

随着移动互联网技术的崛起,继 SaaS(软件即服务:Software as a Service)、IaaS(基础设施即服务:Infrastructure as a Service)和 PaaS(平台即服务:Platform as a Service)之后,BaaS(后端即服务:Backend as a Service)生态系统正从一个小众垂直领域迅速成为非常重要的行业环节。本文从理论和实践两个角度出发,研究 native 的移动开发(以 IOS、Android 为例)过程中存在的主要问题,学习后端即服务的产业现状和实现技术,并且设计和实现一个简单的移动应用后端服务框架。

本文首先较为详细地介绍了在本系统中运用到的知识背景。接着对 Android 和 IOS 应用开发中数据存储传输方面存在的共同问题进行探讨,对其解决之道“后端即服务生态系统”的现状进行研究。而后对设计和实现一个简单的移动应用后端服务框架的可行性进行了分析,同时简要介绍了系统中所用到的开发语言 Ruby 和数据库 Mongodb 的特点以及使用方法。对系统的设计思想、设计目标与系统的整体结构进行了明确的规划。最后对框架的设计与实现作了较为详细的讲解。

系统涉及的主要技术有 Ruby on Rails 和相关的部署技术、Mongodb 和相关的分布式数据库技术、Java 和 Android SDK、Objective-c 和 Cocoa SDK。其主要功能有:移动应用数据远程存储和本地缓存服务,一站式的移动应用后台搭建,简单的统计服务。

论文在撰写过程中,力求将理论与实践应用相结合,提高系统的可用性和通用性,希望通过本文的论述能够更充分地体现出后端即服务的设计思想以及上述理论和技术在本系统中的应用。

**关键字:** 移动、框架、后端即服务

## Abstract

As the rapid development of mobile Internet, following the SaaS (Software as a Service), IaaS (Infrastructure as a Service) and PaaS (Platform as a Service) BaaS (Backend as a service) ecosystem from a vertical niche areas quickly become a very important industry links. From both theoretical and practical perspectives, we will study the main problems in native mobile development (IOS, Android, for example) process, learning the backend that service the industry status quo and implementation techniques, and design and implement a simple back-end services framework for mobile application.

We will firstly introduce the background knowledge in more detail used in this article. Then we will discuss common problems to Android and IOS application development in data storage transmission to study the status of the "(Backend as a service" of its solution. After that, we'd like to analyze the feasibility of the design and implementation of a simple mobile application framework for back-end services, while brief introduction to the system used in the development of language Ruby and databaseMongodb characteristics and usage. Clear planning system design thinking, design goals and overall system structure. Finally, the design of the framework and made more detailed explanation.

The system involves technologies are Ruby on Rails, Mongodb, Java + AndroidSDK, and Objective-c + CocoaSDK. Its main function: remote data storage and local caching services of mobile applications, building a one-stop mobile application background, message push service, simple statistical services.

We strive to make the system be the combination of theory and practical and hashigh usability and versatility. I hope to be able to more fully reflect the the backend that service design thinking and the theory and technology through the discussion of this paper in the system application

**Keywords:**mobile,framework,backend as a service.

## 第1章 绪论

### 1.1 课题背景

据《中国互联网络发展状况统计报告》显示,到2012年12月底,中国互联网用户规模达到5.64亿,其中,移动互联网用户达到4.2亿。截止2012年底,中国移动互联网用户数量为4.2亿,年增长率为18.1%,增幅超过整体互联网用户增长幅度。在所有互联网用户里,使用移动设备接入网络的用户比例由69.3%提升到74.5%。移动终端无疑已成为互联网入口的第一大终端。

随着移动互联网技术的发展和移动应用开发技术的成熟,移动应用开发过程中分工细化的趋势越来越明显,移动应用后端即服务生态系统正从一个小众垂直领域迅速成为非常重要的行业环节。

### 1.2 移动应用开发技术的发展

为了顺应移动互联网的快速发展,移动开发领域也经历了一代又一代的尝试。这些尝试包括最初个人英雄主义的本地应用开发、Flash、html5、widget、移动中间件等。

### 1.3 开发者面临的问题

经过了上述的实验和考证后,人们发现移动开发难度远比PC开发的难度高很多,《神庙逃亡》、《愤怒的小鸟》等神话毕竟是少数,小团队和独立开发者越来越难在移动应用生态系统中获得一席之地。从WAP、原生开发等各式各样的移动开发方式和平台在移动领域无攻而返时,人们开始思考到底什么样的开发方式更能高效、快捷适应移动产业的需要。

当Html5、Widget横空出世火速增援的时候,我们似乎看到了移动蓝海的救星,可是好景不长,Facebook放弃Html5另觅他路,让Html5的弱点暴露无疑,具备一定开发能力的厂商也发现Html5并非为移动开发而生,好多在移动中实现问题还有待解决,Widget也远没有如传说中的那样趟平整个移动领域,移动中间件则经过多年的锤炼一直拥有良好的口碑和解决问题的能力。可是需要学习第三方



的语言和开发原理才能应用自如，学习曲线和难度之大使得开发者敬而远之，并且市场上主流的移动中间件产品在技术上采取全面封锁措施，对开发者采取封闭状态，同时更多的软件厂商也不愿把关系产品命脉的赌注，全部押给并不开放的移动中间件的手里，这也是目前移动中间件所面临的四面高歌却孤立冷清的局面。

#### 1.4 “后端即服务”所提供的服务和解决方案

后端即服务的移动解决方案，让繁琐的移动开发过程更简单快捷，也使得移动开发领域分工更加明确，让android/IOS等移动开发者无需掌握任何一门服务器开发语言(JAVA、PHP)，就可以按照提供的android/IOS/WP三个平台的SDK内置的预定义功能接口快速开发移动应用。解决当前原生开发难以解决、耗时耗力的数据存储、文件存储、消息PUSH等工作。

#### 1.5 本文的组织结构和创新点

本文共分为六章，具体内容结构安排如下：

第一章绪论部分。介绍论文的研究背景和研究意义，阐述了论文的大体结构。

第二章简单介绍了毕业设计的技术背景及相关知识。

第三章介绍当前后端即服务的发展情况。

第四章设计和实现了一个简单的移动应用后端服务框架

第五章讨论了设计框架的通用性和可用性

第六章总结。

## 第 2 章 技术背景及相关知识简介

### 2.1 开发环境

- 服务端使用 Ruby 和 Node.js
- Android 端使用 Java 和 Android SDK
- IOS 端使用 Objective-c 和 Cocoa SDK
- 使用 MongoDB 进行数据存储
- 使用 NetBeans、Xcode、Eclipse 作为集成开发环境
- Windows 和 Mac 下进行开发，服务端部署在 Ubuntu 的 Linux 上

#### 2.1.1 Ruby 语言简介

Ruby，一种为简单快捷的面向对象编程（面向对象程序设计）而创的脚本语言，在 20 世纪 90 年代由日本人松本行弘（まつもとゆきひろ/Yukihiro Matsumoto）开发，遵守 GPL 协议和 Ruby License。它的灵感与特性来自于 Perl、Smalltalk、Eiffel、Ada 以及 Lisp 语言。由 Ruby 语言本身还发展出了 JRuby（Java 平台）、IronRuby（.NET 平台）等其他平台的 Ruby 语言替代品。Ruby 的作者于 1993 年 2 月 24 日开始编写 Ruby，直至 1995 年 12 月才正式公开发布于 fj（新闻组）。因为 Perl 发音与 6 月诞生石 pearl（珍珠）相同，因此 Ruby 以 7 月诞生石 ruby（红宝石）命名。

在开发层面，Ruby 具有以下优点：

- 语法简单
- 普通的面向对象功能(类,方法调用等)
- 特殊的面向对象功能(Mixin,特殊方法等)
- 操作符重载
- 错误处理功能
- 迭代器和闭包
- 垃圾回收
- 动态载入(取决于系统架构)

➤ 可移植性高.不仅可以运行在多数 UNIX 上,还可以运行在 DOS,Windows,Mac,BeOS 等平台上

➤ 适合于快速开发,一般开发效率是 JAVA 的 5 倍

### 2.1.2 MongoDB 简介

MongoDB 是一个基于分布式文件存储的数据库。由 C++语言编写。旨在为 WEB 应用提供可扩展的高性能数据存储解决方案。

MongoDB 是一个介于关系数据库和非关系数据库之间的产品,是非关系数据库当中功能最丰富,最像关系数据库的。他支持的数据结构非常松散,是类似 json 的 bson 格式,因此可以存储比较复杂的数据类型。Mongo 最大的特点是他支持的查询语言非常强大,其语法有点类似于面向对象的查询语言,几乎可以实现类似关系数据库单表查询的绝大部分功能,而且还支持对数据建立索引

MongoDB具有以下特点:

- 面向集合存储,易存储对象类型的数据。
- 模式自由。
- 支持动态查询。
- 支持完全索引,包含内部对象。
- 支持查询。
- 支持复制和故障恢复。
- 使用高效的二进制数据存储,包括大型对象(如视频等)。
- 自动处理碎片,以支持云计算层次的扩展性。
- 支持 RUBY, PYTHON, JAVA, C++, PHP,C#等多种语言。
- 文件存储格式为 BSON(一种 JSON 的扩展)。
- 可通过网络访问。

### 2.1.3 Android SDK

Android 开发工具包,为 Android 手机开发人员提供手机开发的 Java 接口,从 2009 年的 1.5 版本至今,Android SDK 已经发布了十数个版本。

### 2.1.4 Cocoa SDK

Cocoa 是苹果公司为 Mac OS X 所创建的原生面向对象的编程环境,是 Mac OS X 上五大 API 之一(其它四个是 Carbon、POSIX、X11 和 Java)。

苹果的面向对象开发框架，用来生成 Mac OS X 的应用程序。主要的开发语言为 Objective-c，一个 c 的超集。Cocoa 开始于 1989 年 9 月上市的 NeXTSTEP 1.0，当时没有 Foundation 框架，只有动态运行库，称为 kit，最重要的是 AppKit。1993 年 NeXTSTEP 3.1 被移植到了 Intel, Sparc, HP 的平台上，Foundation 首次被加入，同时 Sun 和 NeXT 合作开发 OpenStep 也可以运行在 Windows 系统上。

Cocoa 应用程序一般在苹果公司的开发工具 Xcode(前身为 Project Builder) 和 Interface Builder 上用 Objective-C 写成。

## 第 3 章 移动开发后端即服务行业现状

### 3.1 移动开发分工细化催生了后端即服务

开发一个需要联网的移动客户端应用是复杂的。开发者不仅需要掌握一门客户端语言 (java、objective-c 等), 了解相应开发工具包 (Android SDK、Cocoa SDK 等) 还需考虑服务器编程和数据库技术; 除了需要关注用户交互、产品逻辑, 还需分神于数据存储、网络交互、推送服务、数据统计和用户行为分析。随着移动开发分工的细化, 以方便移动客户端开发部署为目的的移动应用后端即服务概念应运而生。

### 3.2 当前后端即服务的服务内容概述

当前已知已有二十多家国外公司以及几家国内公司提供移动应用后端服务, 综合看来, 提供的服务主要包括以下方面:

- 数据存储
- 推送服务
- 地理位置
- 社交整合
- 统计和用户行为分析

### 3.3 分析几个典型的移动后端服务厂商

#### 3.3.1 StackMob

StackMob 是最早进入移动应用后端服务领域的厂商, 成立于 2010 年。月访问量超过 300 万, 为 IOS、Android 等应用提供开发 SDK 和部署支持。同时提供自定义代码接口, 以方便开发者进行后端逻辑的扩展。

StackMob 提供了较为全面的功能性服务。包括: 数据存储、推送服务、地理信息、较色管理等。

以从零开始依靠 StackMob 开发一个 Android 应用的角度看去, 开发一个 Android 应用包括以下步骤:

### (1) 安装安卓开发环境

后端服务不同于html5或者原理类似的应用生成工具，依然需要传统的本地开发环境，从Eclipse官网下载Eclipse，从Android开发者中心下载适应版本的Android SDK。完成后建立本地Android工程。

### (2) 注册 StackMob 账号和添加其 SDK

在StackMob网站注册账号，获得自己的密钥字符串。此密钥字符串作为服务器本地应用的标识。

从StackMob网站下载jar包，拖入libs文件夹中，android ADT将其自动引入项目。

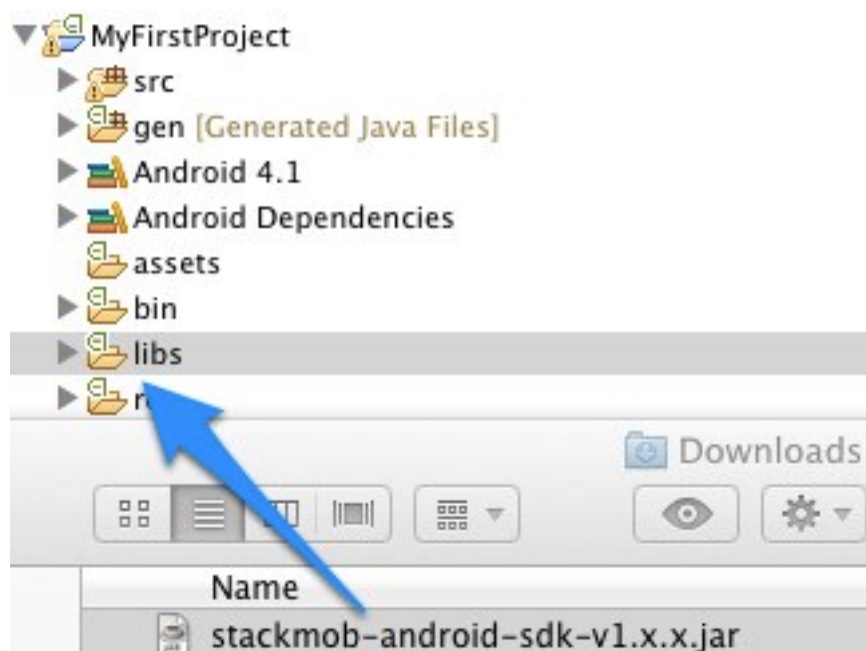


图 3.1 导入 stackMob

### (3) 简单的配置

在AndroidManifest.xml配置文件中设置StackMob必须的网络权限，并在程序启动处加入初始化代码指定应用密钥

### (4) 扩展数据模型基类构建自己的数据模型

传递给StackMob后端的数据只有被标示为StackMobSDK所定义的数据基类(StackmobModel)的子类才能被接收和识别，通过扩展StackMobModel构建符合

自己应用需求的数据模型，这个过程是十分自由的，构建过程如同构建通常的 java model 类，无需额外配置。

StackMob 支持最多三层嵌套的数据模型。

## (5) 存储和使用云端数据

至此 StackMob 的简单配置工作已经完成。分别调用 model 的 save 与 getDataStore 方法即可方便地从远端读取数据。这个过程使得服务器端数据库的建表、指定数据类型、建立索引等操作对移动开发者透明。

## (6) 文件存储

文件存储方面，StackMob 封装了亚马逊 S3 文件存储服务，开发者可以在 StackMob 后台配置自己的 S3 账号，然后即可在应用中使用简单的代码实现文件的云端存储。存储一张图片仅需要编写很少的代码：

```

1 | byte[] image = // ... get the bytes of whatever you're uploading
2 | Task myTask = new Task("Upload some files", new Date());
3 | myTask.setPhoto(new StackMobFile("image/jpeg", "mypicture.jpg", image));
4 | test.save(new StackMobModelCallback() {
5 |     @Override
6 |     public void success() {
7 |         System.out.println(test.getPhoto().getS3Url());
8 |     }
9 |
10 |     @Override
11 |     public void failure(StackMobException e) {
12 |     }
13 | });

```

图 3.2 StackMob 编码示例

## (7) 消息推送

类似使用 S3 实现文件存储功能，StackMob 直接使用了 GCM (Google Cloud Message) 实现自身的推送功能。这一方面提高了推送服务的覆盖面和可靠性，另一方面却增加了开发者的关注点和配置的工作量。

## (8) 地理位置服务

StackMob 的地理位置服务是对手机位置信息服务的封装，提供了简单的位置模型 StackMobGeoPoint，并在服务端封装和位置相关的算法，寻找十米之内的设备的方法在此框架中可以很容易得通过少量代码写成：

```
StackMobGeoPoint sf = new StackMobGeoPoint(122.68, 37.73);
StackMobQuery q = new StackMobQuery().fieldIsNearWithinMi("location", sf, 10);
Task.query(Task.class, q, new StackMobQueryCallback<Task>() {
    @Override
    public void success(List<Task> result) {
        for(Task task : result) {
            System.out.println(task.getLocation().getQueryDistanceRadians());
        }
    }

    @Override
    public void failure(StackMobException e) {
    }
});
```

图 3.3 StackMob 对 LBS 的支持

### 3.3.2 Parse

Parse 成立于 2011 年，是国外又一大移动应用后端服务提供商。有 2 万名开发者使用其平台，40 万款应用使用其服务。支持的 API 有 iOS、Android、JavaScript、Windows 8，以及提供了 REST API 和 Cloud Code JS API，作为后来者，Parse 提供了与 StackMob 类似的后端服务，下面从几个方面简单介绍 Parse 的独特之处。

#### (1) 数据存储

Parse 的数据存储模式不如 StackMob 灵活，所有的存储类型都必须是 ParseObject 所包含的键值对，并且通过 ParseObject 中的 save、get 等方法处理。

这种处理办法可以提高解析效率，也降低了服务端的复杂程度，在开发关于简单数据结构模型的应用时更加方便快捷。但提高了 Model 和 Controller 的耦合性，在开发复杂模型相关的应用时，存在着模型管理困难和移植成本高的风险。如此的情境，可能会增加开发对移动应用后端服务提供商的依赖性。

#### (2) 自定义后台

与 StackMob 一样，Parse 提供了对用户自定义后台逻辑的支持。所不同的是，StackMob 提供了 Java 的接口，而 Parse 用户以 JS 的形式编写自定义后台函数。例如定义：

```
1 | Parse.Cloud.define("hello", function(request, response) {
2 |     response.success("Hello world!");
3 | });
```

图 3.4 Parse 自定义后台



则客户端即可调用名为hello的服务端方法，打印“Hello World!”字符串。

### (3) 文件存储和权限管理

Parse的文件存储并未借用第三方服务，而是使用自己的文件服务器，这从一定程度上增强了基于Parse开发过程的流畅性。开发者可以直接使用统一的后端厂商和数据接口进行开发。

此外访问控制方面Parse相对StackMob也更为灵活，提供了细化到对象的访问控制管理。

### 3.3.3 国内移动后端服务现状

国内目前尚无StackMob或Parse一样提供如此功能覆盖全面、支持终端众多的移动后端服务平台，但是也有一些功能相对简单或更注重领域性的移动后端服务提供商正在发展。

#### (1) Bomb

Bomb 是广州鹏锐公司的产品，产品诞生于 2012 年，是国内第一个 Baas 平台，目前仅支持 Android 手机应用的后端服务。

Bomb 提供了与 Parse 非常类似的数据存储服务 and SDK 接口，同样采用名值对的形式将数据包装成 BombObject，继承了 Parse 简单易用优点的同时，也同样存在耦合性高、不易于管理复杂对象和迁移成本大的缺点。

同时作为一款本土的移动后端服务平台，Bomb 在 SDK 中集成了支付功能，目前支持十几种游戏点卡、充值卡的在线支付功能。

#### (2) 友盟

友盟是一个以应用统计起步的移动后端服务提供商，在2010年推出了第一个友盟统计产品。目前在Android、IOS、Windows三个移动平台提供数据统计、社交网络集成、移动广告集成、用户行为分析等服务。

在上述服务中，友盟最以数据统计服务闻名。仅需在应用工程中添加少量的代码，即可实现SDK自带的崩溃日志捕获和使用时长统计的功能。同时友盟SDK支持用户自定义事件，分别在后台界面设置事件类型、名称即可在移动端发送相应统计。

除去专业的数据统计分析，友盟的社交网络集成服务也很有特色。目前国内各大主流社交网络都提供了一定程度上的开放接口和SDK供开发者调用，众多的SDK与不同风格的接口代码，使在应用中简单整合社交网络成了一项复杂的工作。应用中很简单的连接一个社交网站，考虑界面、表情包等因素，动辄需要几百行代码。将不同社交网站以统一风格整合在APP中更是一个不小的工作。

友盟所提供的社会化组件很好得解决了这个问题，开发者只需将在各社交网络开放平台取得的API KEY填入友盟的统一后台，即可在应用中很方便地链接对应网站，也无需考虑界面风格不统一、代码风格混乱或API过期的问题。

```
controller.doOauthVerify(mContext, SHARE_MEDIA.SINA,
    new OauthCallbackListener() {
        @Override
        public void onError(SocializeException e, SHARE_MEDIA platform) {

        }

        @Override
        public void onComplete(Bundle value, SHARE_MEDIA platform) {
            if (value != null && !TextUtils.isEmpty(value.getString("uid"))) {
                Toast.makeText(mContext, "授权成功.", Toast.LENGTH_SHORT)
                    .show();
            } else {
                Toast.makeText(mContext, "授权失败", Toast.LENGTH_SHORT).show();
            }
        }
    });
```

图 3.5 友盟 SDK 示例

## (2) 百度开发者中心

依托自身技术积累，百度为移动开发者提供了较为个性的后端服务，包括：百度个人云存储服务、百度地图服务、百度移动统计服务、社会化网络整合服务等。

上述服务中，在国内最有特色的应属个人云存储服务。百度云为移动端个人云存储提供了易用的 REST 接口，以上传单个文件为例，客户端通过 post 方式发送传送请求：

表 3.1 百度 SDK 接口

参数名称	类型	是否必需	描述
method	string	是	固定值，upload。

access_token	string	是	开发者准入标识，HTTPS 调用时必须使用。
path	string	是	上传文件路径（含上传的文件名称）。
file	char[]	是	上传文件的内容。
ondup	string	否	<ul style="list-style-type: none"> <li>▪ <b>overwrite</b>: 表示覆盖同名文件；</li> <li>▪ <b>newcopy</b>: 表示生成文件副本并进行重命名，命名规则为“文件名_日期.后缀”。</li> </ul>

服务器则以 JSON 形式返回相应结果：

表 3.2 百度 SDK 返回结果

参数名称	类型	UrlEncode	描述
path	string	是	该文件的绝对路径。
size	uint64	否	文件字节大小。
ctime	uint64	否	文件创建时间。
mtime	uint64	否	文件修改时间。
md5	string	否	文件的 md5 签名。
fs_id	uint64	否	文件在 PCS 的临时唯一标识 ID。

### 3.4 行业现状小结

综上所述，当前移动应用后端服务行业在国外出现了像 StackMob、Parse 这一类提供多平台、综合服务的专业服务提供商，而国内 Baas 服务的实践者多是创业型公司，无论是提供服务的平台普及率或是功能覆盖面都还有较大差距。而国内技术实力较强的大公司，在移动应用后端服务领域多注重自己的专业方面。

综合看来，当前移动应用后端服务所提供的服务，功能上，主要包括数据存储、推送服务、地理位置、社交整合、统计和用户行为分析等；结构上，普遍由一个服务平台以及多个客户端平台的开发工具包构成。更少的代码、更便利的后

端服务，使开发者关注于前端业务逻辑和交互，而非后端技术，是各大后端服务产品统一的追求。

## 第4章 设计和实现一个移动应用后端服务框架

### 4.1 本章概述

本章将在上述研究和总结的基础上设计和实现一个简单的移动应用后端服务框架，框架以ruby、Mongodb搭建服务端，且分别提供对Android、IOS两个平台的支持。使开发这在此框架功能下，可以方便地进行移动客户端与后台的数据交互、日志传送、统计分析、消息推送等功能。

本章从：系统设计原则、总体结构介绍、关键技术讨论、系统使用和部署流程四个方面展开叙述。

### 4.2 系统设计原则

本系统纵向程度较大，为使系统的设计和实现过程是可以被把握的，在设计之初确定以下三个简单的设计原则，这三个简单的原则将贯彻体现在下文中：

- 1、 少做重复工作。积极复用久经考验的框架、技术和协议来保证开发效率和系统稳定，积极使用最新技术提供的新特性来提高系统的运行效率和性能。
- 2、 用合适的工具和方法。用合适的工具和方法做特定的事，综合利用各种技术解决问题。避免出现“如果你有的只是锤子，你会把所有问题看成钉子”这样的情况出现。
- 3、 保持干净和简单。避免过度设计，复杂过程透明，使系统对开发人员友好，便于部署、易于使用。

### 4.3 总体结构介绍

系统面向移动应用开发者，提供数据存储、统计分析、通用逻辑处理（用户、位置信息、文件IO）和推送服务

### 4.3.1 数据存储服务总体结构

数据存储服务依托文档型数据库 Mongodb，总体为数据库-服务器-客户端三层结构。服务器是一个 RESTFUL 的 Ruby on Rails 模块，客户端和服务端通过 REST 协议进行通信。

为做到安全和效率的平衡，REST 协议通过 HMAC 方法进行身份认证，HMAC 的方法对 REST 进行认证是很多通过 REST 接口向外提供服务的后端厂商的选择，例如 Amazon 的 S3 云存储服务，就使用了这种方法。

为提高存储效率和提升数据安全稳定性，Mongodb 使用数据库分片技术进行分布式部署，使用主从备份的方式进行安全保证。分片集群的规模和备份策略的选取被封装成简单和可配置的。

在客户端，定义了对象序列化反序列化接口、http 方法、对 REST API 封装的接口，然后分别根据不同客户端平台特征进行了各自实现，这三部分构成了客户端 SDK 数据存储方面的底层。

面向开发者，客户端 SDK 提供便于扩展的 Model 基类和通用的 Data Access Object (DAO) 及其相关方法，DAO 提供 Remote 和 native 的两套实现，使得开发者可以在本地缓存和远端存储之间进行选择。

数据存储服务的总体结构示意图如下：

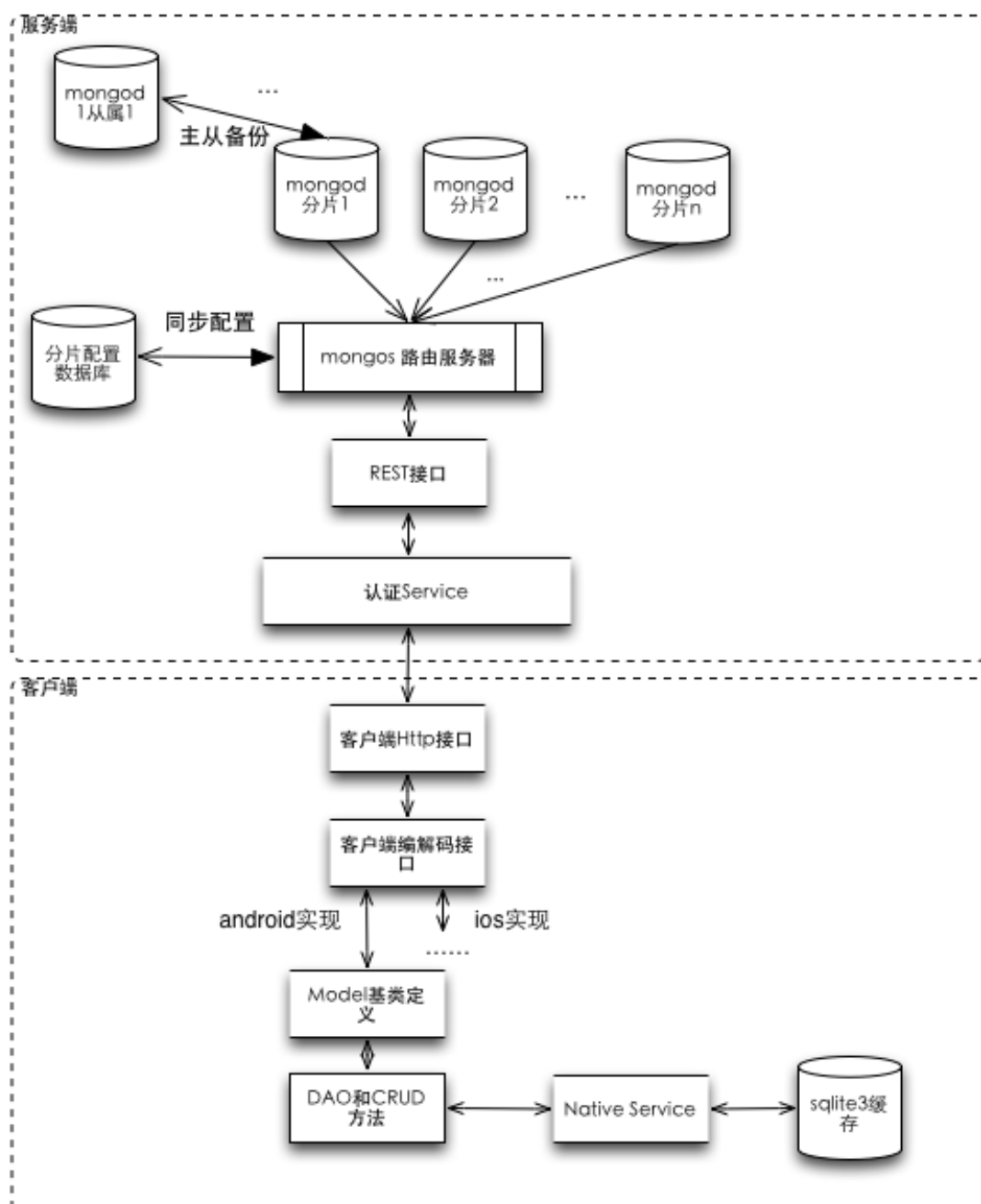


图 4.1 数据存储服务总体结构

#### 4.3.2 统计服务

统计服务围绕Countly进行整合和开发,Countly是目前最强大的开源移动分析应用。统计服务在Countly-server的基础上构建统计服务的监听器和前端,使用统一的数据库进行数据存储。

在客户端,根据安卓和IOS的不同特性,编写各自的统计SDK,对移动应用的生命周期、使用时长、日活率、版本分布、用户分布等通用信息统计项进行默认

支持，同时，为用户提供自定义事件统计的相关支持。

在服务端，围绕Countly统计数据，在Countly API的基础上进行开发，构建可视化的配置界面和Web前端。

统计服务的系统结构示意图如下：

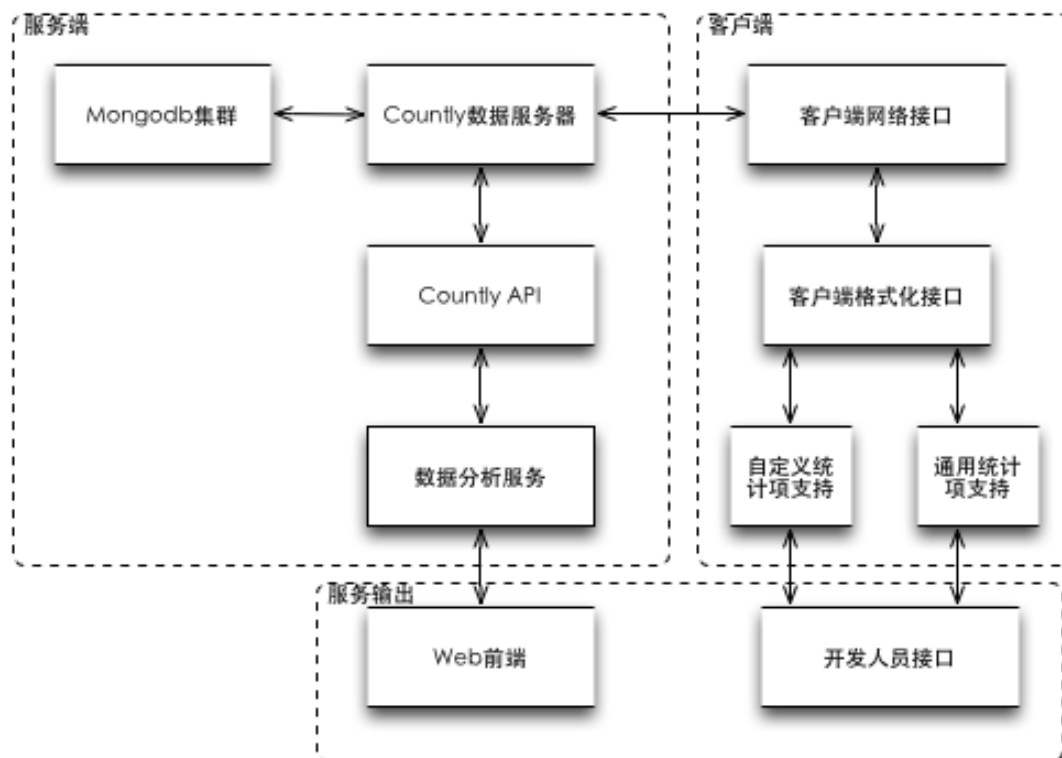


图 4.2 统计服务结构

### 4.3.3 通用逻辑处理

这一层服务主要针对移动应用开发过程中的常见功能和常用处理逻辑，使得简单应用开发不必进行重复劳动。

值得注意的是，成熟的后端服务提供商往往提供更为复杂的处理服务（例如根据地理信息推荐周围商家），或者提供支持开发者自定义服务端逻辑的中间语言接口（如Parse提供的JS接口），而笔者不具备精力和能力把握这些，故这一层功能支持相较较弱，只提供了简单而基本的服务，扩展服务端逻辑也须在服务器代码的基础上进行Ruby on Rails开发。



本系统中，通用逻辑处理提供三种系统内置对象的相关服务：用户、位置和文件。结构上，所有的服务端存储工作依旧基于如4.3.1中所描述的分布式Mongodb数据库群，在此基础上使用Ruby on Rails + MongoMapper 实现了运算逻辑并将计算和资源以RESTFUL的形式暴露。客户端分别根据Android、IOS提供的LBS、IO、网络接口对客户端运算进行了封装，根据前文所述通用REST接口与服务器进行通讯。

通用服务的系统结构示意图如下：

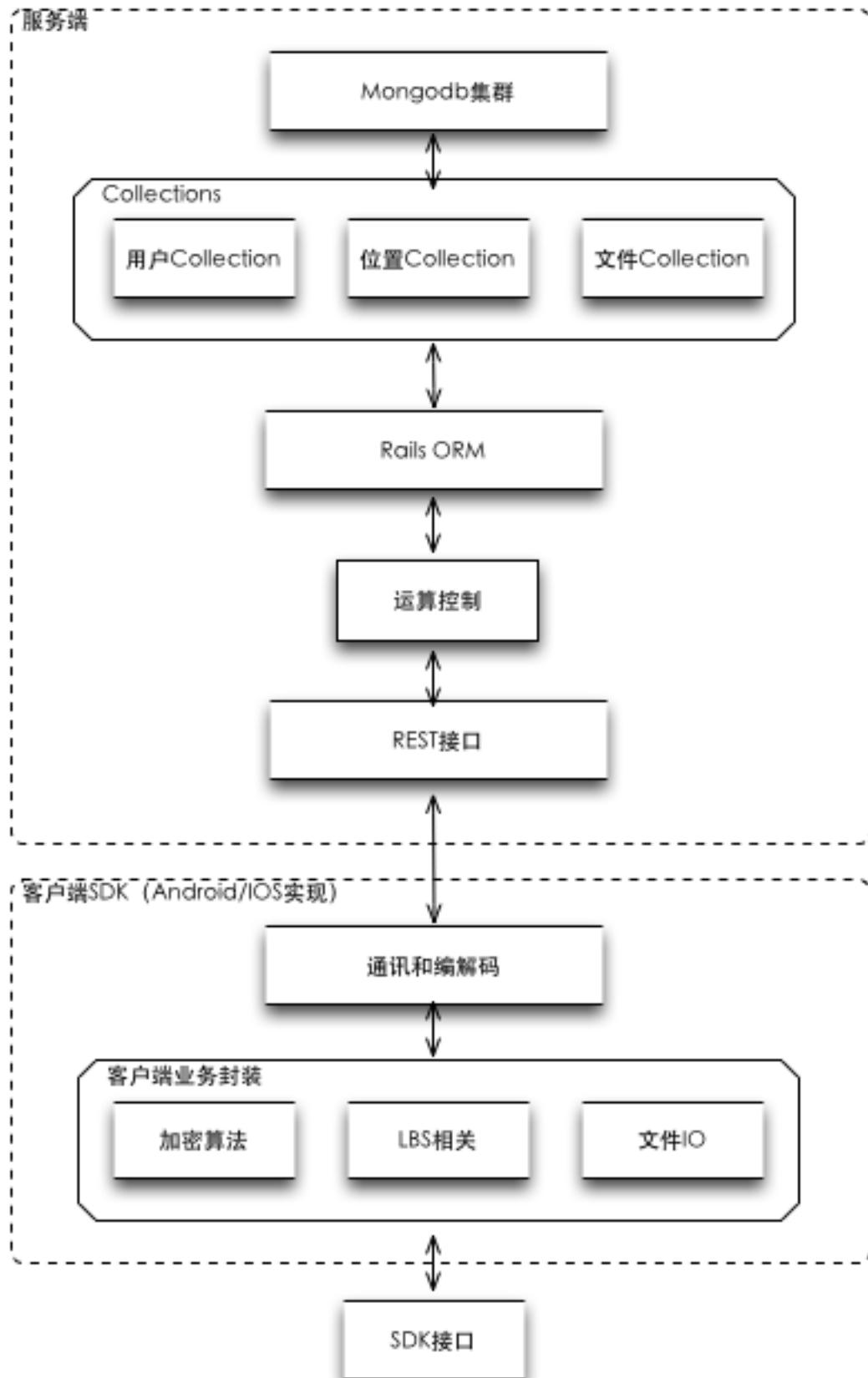


图 4.3 通用逻辑处理结构

## 4.4 关键技术讨论

### 4.4.1 搭建分布式数据库集群

使用分布式数据库集群的结构实现本系统的存储主要有性能和成本两方面的考虑。

一方面，分布式系统更经济，可用性、可靠性好。分布式系统比集中式系统具有更高的可靠性和更好的可用性。如由于数据分布在多个场地并有许多复制数据，在个别场地或个别通信链路发生故障时，不致于导致整个系统的崩溃，而且系统的局部故障不会引起全局失控。另一方面，分布式系统可扩展性好，易于集成现有系统，也易于扩充。对于一个企业或组织，可以采用分布式数据库技术在已建立的若干数据库的基础上开发全局应用，对原有的局部数据库系统作某些改动，形成一个分布式系统。这比重建一个大型数据库系统要简单，既省时间，又省财力、物力。也可以通过增加场地数的办法，迅速扩充已有的分布式数据库系统。

Mongodb 便捷的部署方式和天生具有的自动分片机制，决定了十分符合本系统对数据库层的需要。

一个完整的 Mongodb 数据库集群包括一个 mongos 路由器，若干 mongod 分片存储服务器，以及若干配置服务器。结构如图：

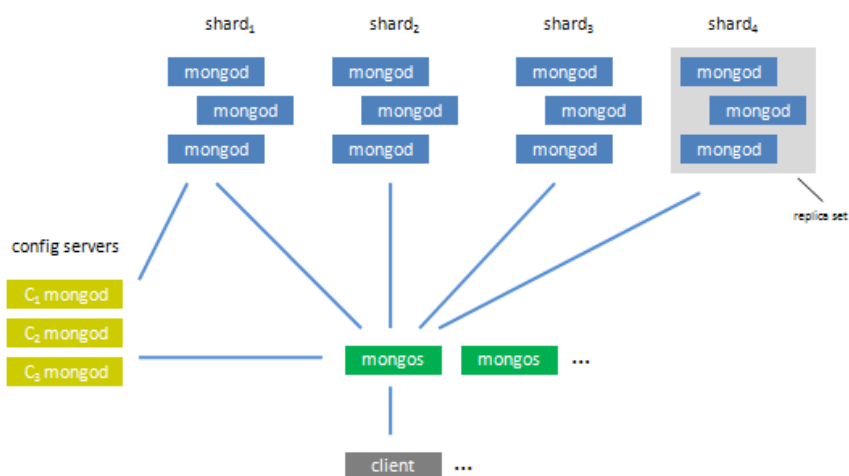


图 4.4 数据库集群示意图

## Shard:

图中 Shard 为一组 mongod，存储一组分片数据。一组 Mongod 之间实现主从备份。在 mongodb 中，主从备份的配置是十分便捷的，其数据结构如图：

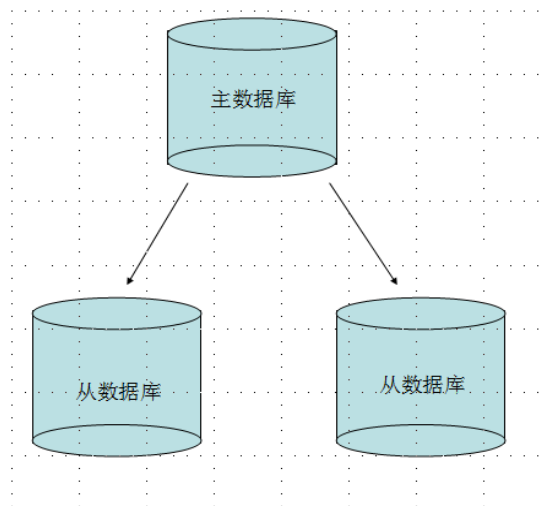


图 4.5 主从备份结构图

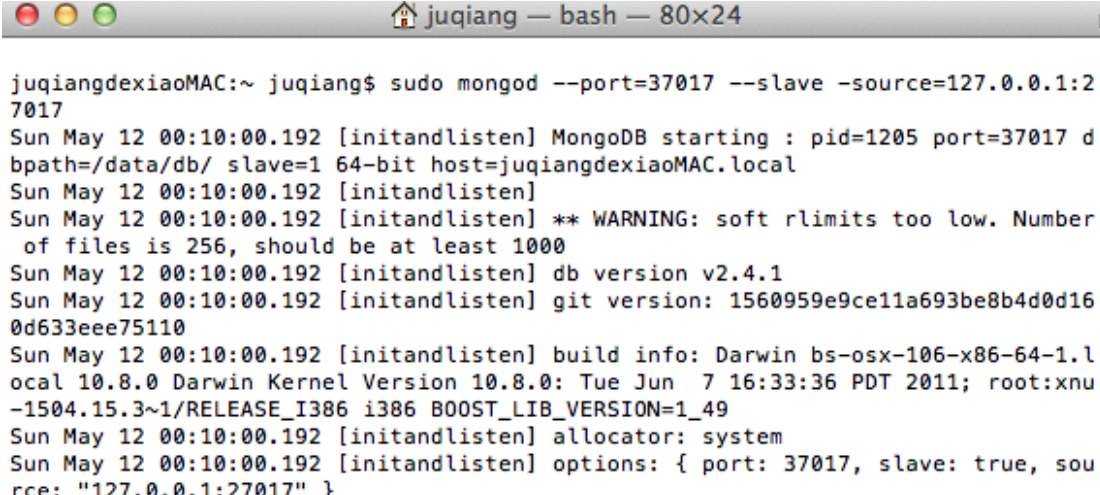
首先使—master 参数指定 shard 中的主数据库如图：

```

juqiang — mongod — 82x24
juqiangdexiaoMAC:~ juqiang$ sudo mongod --master
Password:
Sat May 11 23:52:52.768 [initandlisten] MongoDB starting : pid=1127 port=27017 dbp
ath=/data/db/ master=1 64-bit host=juqiangdexiaoMAC.local
Sat May 11 23:52:52.768 [initandlisten]
Sat May 11 23:52:52.769 [initandlisten] ** WARNING: soft rlimits too low. Number o
f files is 256, should be at least 1000
Sat May 11 23:52:52.769 [initandlisten] db version v2.4.1
Sat May 11 23:52:52.769 [initandlisten] git version: 1560959e9ce11a693be8b4d0d160d
633eee75110
Sat May 11 23:52:52.769 [initandlisten] build info: Darwin bs-osx-106-x86-64-1.loc
al 10.8.0 Darwin Kernel Version 10.8.0: Tue Jun 7 16:33:36 PDT 2011; root:xnu-150
4.15.3~1/RELEASE_I386 i386 BOOST_LIB_VERSION=1_49
Sat May 11 23:52:52.769 [initandlisten] allocator: system
Sat May 11 23:52:52.769 [initandlisten] options: { master: true }
Sat May 11 23:52:52.770 [initandlisten] journal dir=/data/db/journal
  
```

图 4.6 指定主数据库

添加一个从属服务器，此处使用本地其他分区 mongodb 模拟多台机器，开辟新的端口并指定主服务器为 mongod master，mongodb 默认使用 27017 端口，此处从属服务器使用 37017 端口，如图：



```

juqiangdexiaoMAC:~ juqiang$ sudo mongod --port=37017 --slave -source=127.0.0.1:27017
Sun May 12 00:10:00.192 [initandlisten] MongoDB starting : pid=1205 port=37017 dbpath=/data/db/ slave=1 64-bit host=juqiangdexiaoMAC.local
Sun May 12 00:10:00.192 [initandlisten]
Sun May 12 00:10:00.192 [initandlisten] ** WARNING: soft rlimits too low. Number of files is 256, should be at least 1000
Sun May 12 00:10:00.192 [initandlisten] db version v2.4.1
Sun May 12 00:10:00.192 [initandlisten] git version: 1560959e9ce11a693be8b4d0d160d633eeee75110
Sun May 12 00:10:00.192 [initandlisten] build info: Darwin bs-osx-106-x86-64-1.1 local 10.8.0 Darwin Kernel Version 10.8.0: Tue Jun 7 16:33:36 PDT 2011; root:xnu-1504.15.3~1/RELEASE_I386 i386 BOOST_LIB_VERSION=1_49
Sun May 12 00:10:00.192 [initandlisten] allocator: system
Sun May 12 00:10:00.192 [initandlisten] options: { port: 37017, slave: true, source: "127.0.0.1:27017" }
    
```

图 4.7 添加从属数据库

mongodb 从属数据库默认 10s 从主数据库同步数据，同步数据通过同步主数据库的 Oplog 实现。

对于更高要求的数据安全以及系统故障自动恢复要求，也可以使用副本集的结构代替以上的主从备份方式，副本集的特点是没有如上结构所说特定的 master 数据库，在当前数据库故障的情况下，副本集会使用另外的备份节点代替故障数据库，从而使系统具备了一定的自动恢复能力。

这里依然采用在一台机器上通过多个数据库进程模拟多台数据库服务器的方式。首先指定副本集中第一个副本的端口和副本集名称（`sudo mongod --port 2001 -replSet omgset/127.0.0.1:2002`），建立我们的副本集 omgset，如图

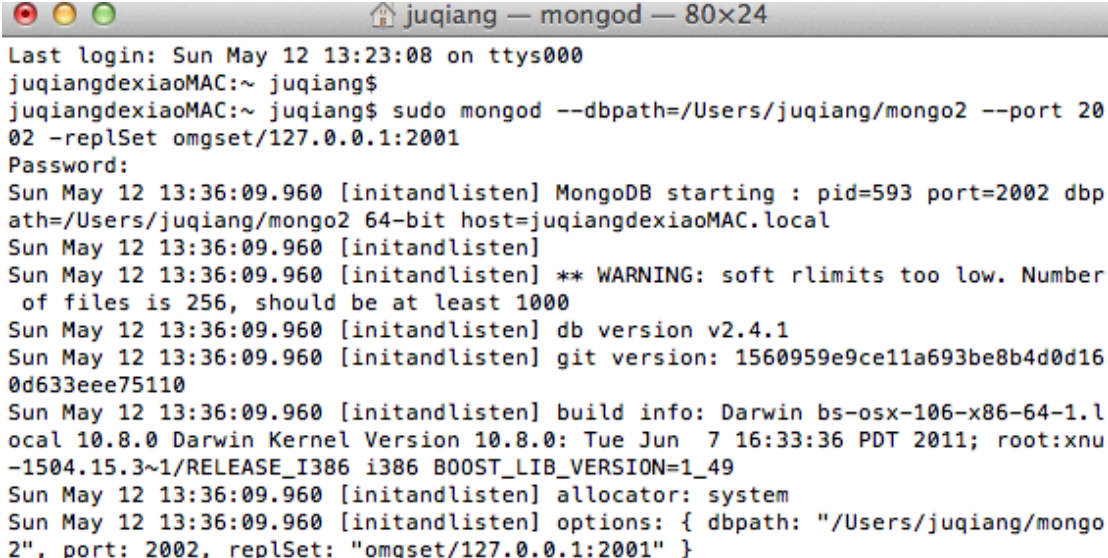


```

juqiang — mongod — 84x24
juqiangdexiaoMAC:~ juqiang$ sudo mongod --port 2001 --replSet omgset/127.0.0.1:2002
Password:
Sun May 12 13:28:03.372 [initandlisten] MongoDB starting : pid=537 port=2001 dbpath=
/data/db/ 64-bit host=juqiangdexiaoMAC.local
Sun May 12 13:28:03.372 [initandlisten]
Sun May 12 13:28:03.372 [initandlisten] ** WARNING: soft rlimits too low. Number of
files is 256, should be at least 1000
Sun May 12 13:28:03.372 [initandlisten] db version v2.4.1
Sun May 12 13:28:03.372 [initandlisten] git version: 1560959e9ce11a693be8b4d0d160d63
3eee75110
Sun May 12 13:28:03.372 [initandlisten] build info: Darwin bs-osx-106-x86-64-1.local
10.8.0 Darwin Kernel Version 10.8.0: Tue Jun 7 16:33:36 PDT 2011; root:xnu-1504.15
.3~1/RELEASE_I386 i386 BOOST_LIB_VERSION=1_49
Sun May 12 13:28:03.372 [initandlisten] allocator: system
Sun May 12 13:28:03.372 [initandlisten] options: { port: 2001, replSet: "omgset/127.
0.0.1:2002" }
    
```

图 4.8 建立副本集

随后，如上命令所说我们在 2002 端口初始化副本集的第二个副本，并指向 omgset(sudo mongod --dbpath=/Users/juqiang/mongo2 --port 2002 --replSet omgset/127.0.0.1:2001)，如图，其中—dbpath 所指的，为我们模拟另一个数据库服务器的 mongodb 的存储路径

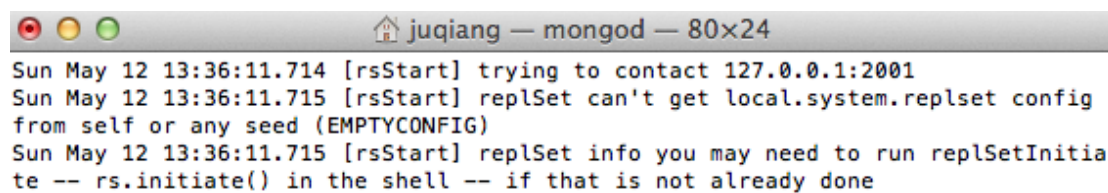


```

juqiang — mongod — 80x24
Last login: Sun May 12 13:23:08 on ttys000
juqiangdexiaoMAC:~ juqiang$
juqiangdexiaoMAC:~ juqiang$ sudo mongod --dbpath=/Users/juqiang/mongo2 --port 20
02 --replSet omgset/127.0.0.1:2001
Password:
Sun May 12 13:36:09.960 [initandlisten] MongoDB starting : pid=593 port=2002 dbp
ath=/Users/juqiang/mongo2 64-bit host=juqiangdexiaoMAC.local
Sun May 12 13:36:09.960 [initandlisten]
Sun May 12 13:36:09.960 [initandlisten] ** WARNING: soft rlimits too low. Number
of files is 256, should be at least 1000
Sun May 12 13:36:09.960 [initandlisten] db version v2.4.1
Sun May 12 13:36:09.960 [initandlisten] git version: 1560959e9ce11a693be8b4d0d16
0d633eee75110
Sun May 12 13:36:09.960 [initandlisten] build info: Darwin bs-osx-106-x86-64-1.l
ocal 10.8.0 Darwin Kernel Version 10.8.0: Tue Jun 7 16:33:36 PDT 2011; root:xnu
-1504.15.3~1/RELEASE_I386 i386 BOOST_LIB_VERSION=1_49
Sun May 12 13:36:09.960 [initandlisten] allocator: system
Sun May 12 13:36:09.960 [initandlisten] options: { dbpath: "/Users/juqiang/mongo
2", port: 2002, replSet: "omgset/127.0.0.1:2001" }
    
```

图 4.9 模拟另一个数据库服务器

此刻，副本集的连接工作基本完成，若需要在副本集中继续添加副本服务器，重复上述步骤即可，观察此刻 mongodb 系统日志如图



```

Sun May 12 13:36:11.714 [rsStart] trying to contact 127.0.0.1:2001
Sun May 12 13:36:11.715 [rsStart] replSet can't get local.system.replset config
from self or any seed (EMPTYCONFIG)
Sun May 12 13:36:11.715 [rsStart] replSet info you may need to run replSetInitia
te -- rs.initiate() in the shell -- if that is not already done
    
```

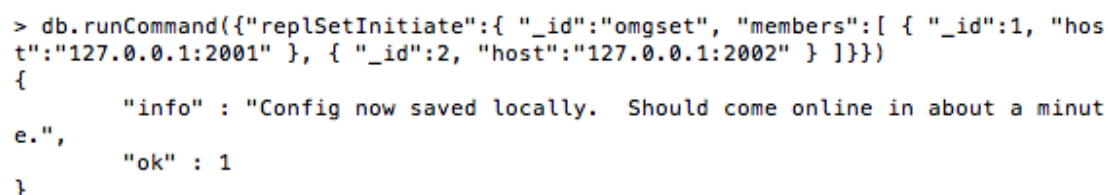
图 4.10 Mongodb 系统日志

表明副本集并未开始工作，此时需要进入任意一个 mongod 副本进行副本集的初始化。例如，进入 2001 端口的副本（mongo 127.0.0.1:2001/admin），在此处我们初始化副本集信息：

```

1. > db.runCommand({"replSetInitiate":{
2. ...   "_id":"omgset",
3. ...   "members":[
4. ...     {
5. ...       "_id":1,
6. ...       "host":"127.0.0.1:2001"
7. ...     },
8. ...     {
9. ...       "_id":2,
10. ...      "host":"127.0.0.1:2002"
11. ...     }
12. ...   ]})
    
```

如图：



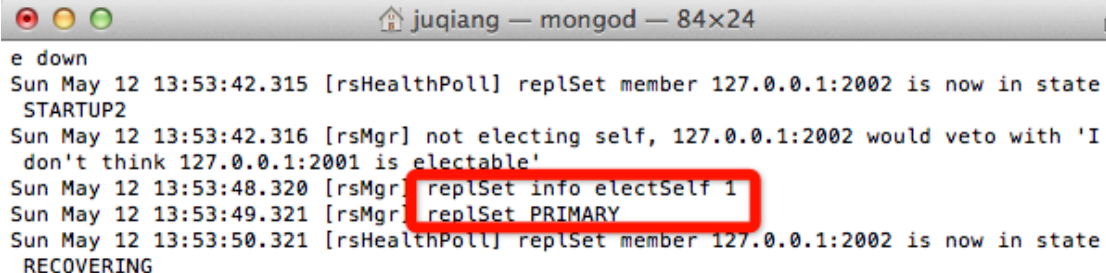
```

> db.runCommand({"replSetInitiate":{"_id":"omgset", "members":[{"_id":1, "host":
t":"127.0.0.1:2001" }, { "_id":2, "host":"127.0.0.1:2002" } ]}})
{
  "info" : "Config now saved locally.  Should come online in about a minut
e.",
  "ok" : 1
}
    
```

图 4.11 初始化副本集信息

此时我们进入了 omgset 副本集的控制区，可以看到 2001 端口的副本已成为主数据库：



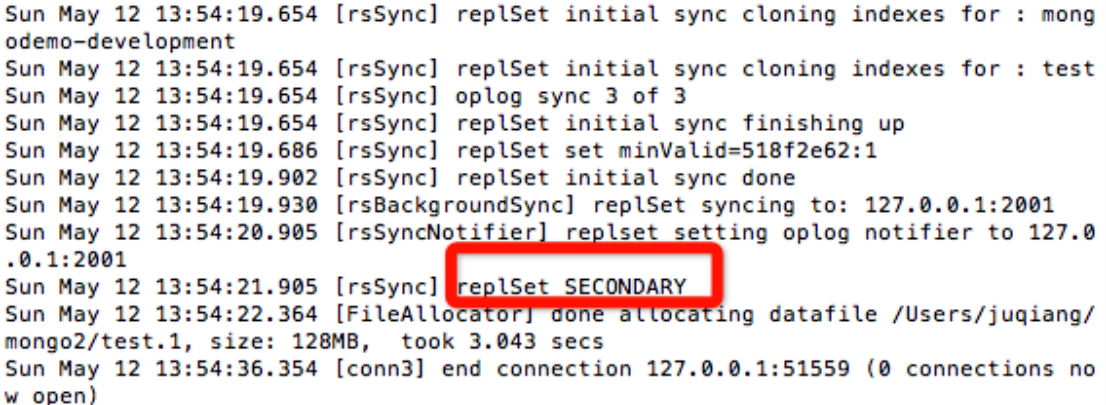


```

e down
Sun May 12 13:53:42.315 [rsHealthPoll] replSet member 127.0.0.1:2002 is now in state
STARTUP2
Sun May 12 13:53:42.316 [rsMgr] not electing self, 127.0.0.1:2002 would veto with 'I
don't think 127.0.0.1:2001 is electable'
Sun May 12 13:53:48.320 [rsMgr] replSet info electSelf 1
Sun May 12 13:53:49.321 [rsMgr] replSet PRIMARY
Sun May 12 13:53:50.321 [rsHealthPoll] replSet member 127.0.0.1:2002 is now in state
RECOVERING
    
```

图 4. 12 2001 端口日志

而 2002 端口的副本，已成为从数据库：

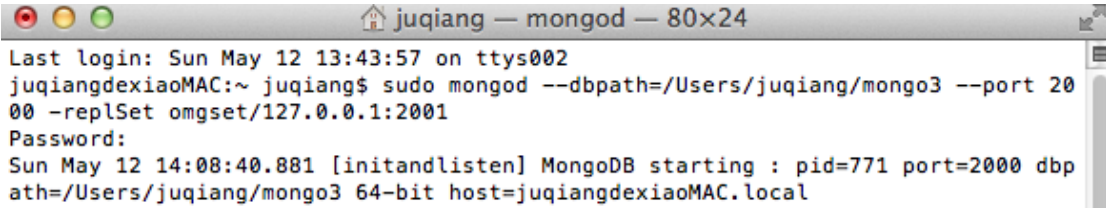


```

Sun May 12 13:54:19.654 [rsSync] replSet initial sync cloning indexes for : mong
odemo-development
Sun May 12 13:54:19.654 [rsSync] replSet initial sync cloning indexes for : test
Sun May 12 13:54:19.654 [rsSync] oplog sync 3 of 3
Sun May 12 13:54:19.654 [rsSync] replSet initial sync finishing up
Sun May 12 13:54:19.686 [rsSync] replSet set minValid=518f2e62:1
Sun May 12 13:54:19.902 [rsSync] replSet initial sync done
Sun May 12 13:54:19.930 [rsBackgroundSync] replSet syncing to: 127.0.0.1:2001
Sun May 12 13:54:20.905 [rsSyncNotifier] replSet setting oplog notifier to 127.0
.0.1:2001
Sun May 12 13:54:21.905 [rsSync] replSet SECONDARY
Sun May 12 13:54:22.364 [FileAllocator] done allocating datafile /Users/juqiang/
mongo2/test.1, size: 128MB, took 3.043 secs
Sun May 12 13:54:36.354 [conn3] end connection 127.0.0.1:51559 (0 connections no
w open)
    
```

图 4. 13 2002 端口日志

此外，一个完成的副本集还应包括仲裁服务器，仲裁服务器只参与投票不参与数据存储查询。使用在 2000 端口的 mongod 作为仲裁服务器,可任意指定它连接到 omgset 副本集中的任意服务器，如在此连接到 2001 端口的服务器，如图

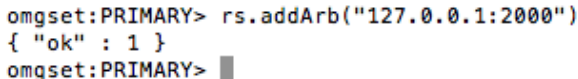


```

Last login: Sun May 12 13:43:57 on ttys002
juqiangdexiaoMAC:~ juqiang$ sudo mongod --dbpath=/Users/juqiang/mongo3 --port 20
00 -replSet omgset/127.0.0.1:2001
Password:
Sun May 12 14:08:40.881 [initandlisten] MongoDB starting : pid=771 port=2000 dbp
ath=/Users/juqiang/mongo3 64-bit host=juqiangdexiaoMAC.local
    
```

图 4. 14 连接到 2001 服务器

然后在 2001 服务器上，可以添加 2000 端口的 mongo3 为仲裁服务器。



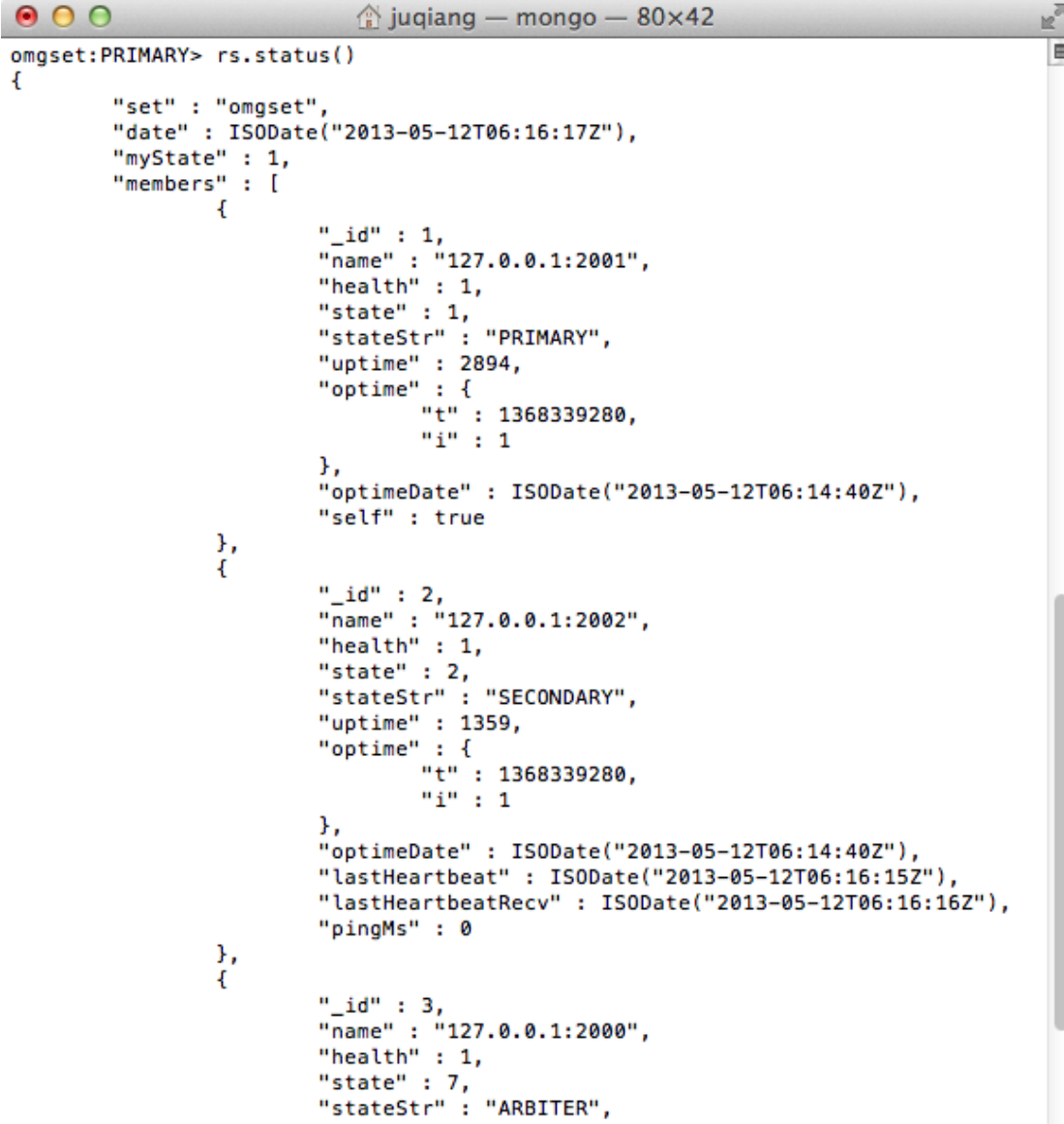
```

omgset:PRIMARY> rs.addArb("127.0.0.1:2000")
{ "ok" : 1 }
omgset:PRIMARY>
    
```

图 4. 15 添加仲裁服务器



此时，使用 `rs.status()` 命令查看 `omgset` 副本集状态，我们可以看到，PRIMARY（主）、SECONDARY（从）、ARBITER（仲裁）三个服务器已经连接就绪。



```

omgset:PRIMARY> rs.status()
{
  "set" : "omgset",
  "date" : ISODate("2013-05-12T06:16:17Z"),
  "myState" : 1,
  "members" : [
    {
      "_id" : 1,
      "name" : "127.0.0.1:2001",
      "health" : 1,
      "state" : 1,
      "stateStr" : "PRIMARY",
      "uptime" : 2894,
      "optime" : {
        "t" : 1368339280,
        "i" : 1
      },
      "optimeDate" : ISODate("2013-05-12T06:14:40Z"),
      "self" : true
    },
    {
      "_id" : 2,
      "name" : "127.0.0.1:2002",
      "health" : 1,
      "state" : 2,
      "stateStr" : "SECONDARY",
      "uptime" : 1359,
      "optime" : {
        "t" : 1368339280,
        "i" : 1
      },
      "optimeDate" : ISODate("2013-05-12T06:14:40Z"),
      "lastHeartbeat" : ISODate("2013-05-12T06:16:15Z"),
      "lastHeartbeatRecv" : ISODate("2013-05-12T06:16:16Z"),
      "pingMs" : 0
    },
    {
      "_id" : 3,
      "name" : "127.0.0.1:2000",
      "health" : 1,
      "state" : 7,
      "stateStr" : "ARBITER",

```

图 4.16 就绪日志

这个副本集已经具备了初步的备份和自我恢复能力。例如，我们把 2001 数据库进程关闭，

```

omgset:PRIMARY> rs.status()
{
  "set" : "omgset",
  "date" : ISODate("2013-05-12T06:23:22Z"),
  "myState" : 1,
  "members" : [
    {
      "_id" : 1,
      "name" : "127.0.0.1:2001",
      "health" : 0,
      "state" : 8,
      "stateStr" : "(not reachable/healthy)",
      "uptime" : 0,
      "optime" : {
        "t" : 1368339280,
        "i" : 1
      },
      "optimeDate" : ISODate("2013-05-12T06:14:40Z"),
      "lastHeartbeat" : ISODate("2013-05-12T06:23:21Z"),
      "lastHeartbeatRecv" : ISODate("1970-01-01T00:00:00Z"),
      "pingMs" : 0
    },
    {
      "_id" : 2,
      "name" : "127.0.0.1:2002",
      "health" : 1,
      "stateStr" : "PRIMARY",
      "uptime" : 2833,
      "optime" : {
        "t" : 1368339280,
        "i" : 1
      },
      "optimeDate" : ISODate("2013-05-12T06:14:40Z"),
      "self" : true
    }
  ]
}
    
```

图 4.17 自动修复能力示意图

可以清楚地看到，当 2001 主数据库不可用（health==0）的时候，副本集自动推举 2002 副本为主数据库（PRIMARY）。

### Mongos:

Mongos 是数据库集群的路由器和控制中心，mongos 处理客户端发来的请求并路由到 Shards 中，整理 Shards 返回的数据或结果传递给客户端。以 Mongos 为界，数据库的分布式结构对客户端是透明的。这使得整个分布式数据库系统十分容易在不影响线上产品的前提下进行维护和扩展。

### Config Server:

Config Server 是分片配置服务器，它是一个 MongoDB 数据库或如上所述的一个 Mongo 数据库副本集，Config Server 保存 Shards 集群的分片信息，若为副

本集，则每个副本中保存相同的数据，确保分片信息完全一致。

#### 4.4.2 构建 RESTFUL 的 Mongodb 接口

REST 描述了一个架构样式的互联系统。REST 约束条件作为一个整体应用时，将生成一个简单、可扩展、有效、安全、可靠的架构。系统中使用 REST 的方式构建数据访问的服务层，正是基于它简便、轻量级以及通过 HTTP 直接传输数据特性的考虑。

传统上，位于移动应用与数据库服务器之间的中间层，往往包含两个两个功能，一个是向下实现对数据库服务器的抽象，将数据库中的数据转换为结构化的数据模型，一个是向上对客户端层提供数据传输和数据操作的接口。在这样的结构中，数据模型的改变一般至少需要进行：修改数据库表结构、修改中间层数据模型、修改应用程序数据模型等步骤才能完成。这样复杂的步骤无疑增加了开发成本，也对后端框架的通用性造成了很大的局限。

Mongodb 作为非关系型文档数据库，无表结构的特点非常适合解决上述问题。而 Ruby 作为动态的服务端脚本语言，其编码的灵活性使其成为构建 Mongodb RESTFUL 接口的良好选择。

我们修改 Ruby 访问 Mongodb 的工作模式为非严格模式，以此获得更灵活的数据访问功能。如图：

```
def connect_to_db(db)
  @dbs ||= {}
  return @dbs[db] if @dbs[db]
  c = connect
  if c.database_names.include?(db.to_s)
    c.db(db, :strict => false)
  else
    return false
  end
end
```

图 4.18 Ruby 访问 Mongodb

在集合不存在的情况下自动建立集合和文档格式，从而在通用性的基础上最大程度的降低了前端移动开发人员对后端的关注度。

基于 HTTP 的 GET、POST、PUT、DELETE 四种方法，分别实现对数据的访问和操作：

## POST 方法

Post 方法是最基本的方法，通过 post /db/collection 的形式，将 json 格式的文档数据存储至服务器，若成功则返回 201 以及插入的文档。

```
desc "Add a new item to the collection"
post do
  data = extract_data_from_params
  id = @db.collection(params[:collection]).insert(data)
  @db.collection(params[:collection]).find_one(id)
end
```

图 4. 19 Post 方法

## GET 方法

通过 HTTP 的 GET 方法实现对文档资源的展现，使用 GET /db/colleciton 的形式，讲查询条件和选项参数传递给服务器，返回相关查询结果

```
desc "Retrieve given items in the collection"
get do
  selector = extract_selector_from_params
  #sort = extract_sort_from_params
  opt=extract_opt_from_params
  puts selector
  puts opt
  @db.collection(params[:collection]).find(selector,opt).to_a
end
```

图 4. 20 Get 方法

其中，最寻 Mongoddb 的 JSON 查询条件，Selector 可以构建类似关系型数据库的多种形式的查询，例如：

selector={"name":"juqiang"} //查询 name 是 juqiang 的文档记录

selector={"name":"juqiang","age":23} //查询 name 是 juqiang，且 age 为 23 的文档记录

`selector={"name":"juqiang","age":{"$exists":true}}` //查询 name 是鞠强, 且 age 存在 age 属性的文档记录

`selector={"age":{"$lt":30}}` //查询所有 age 属性小于 30 的文档记录

`selector={"age":{"$lt":30},"age":{"$gte":10}}` //查询所有 age 大于等于 10 小于等于 30 的文档记录

同时, 为更好得满足查询需求, 除 selector 之外, 一条 GET 请求还可单独附加选项参数, 以实现对查询行为的调控, 目前支持的参数有:

`fields(Array)`: 筛选文档的指定字段集合的查询结果返回

`skip(Integer)`: 跳过某几条记录查询其后的记录

`limit(Integer)`: 指定返回文档记录结果的最大数目

`sort(key-values)`: 指定返回结果的排列顺序, 且排列规则的优先级按 key-values 的先后顺序决定, 如`[["age","ASC"],["grade","DESC"]]`会返回先按年龄升序排列, 相同年龄按年级降序排列的文档队列

## PUT 方法

HTTP PUT 方法映射文档资源的 UPDATE 方法, 使用 PUT db/collection 的形式, 使用更新条件、目标内容和操作参数控制文档的更新行为

```
desc "update"
put do
  selector=extract_selector_from_params
  opt=extract_opt_from_params
  document=extract_document_from_params
  puts selector
  puts opt
  puts document
  @db.collection(params[:collection]).update(selector,{"$set" => document},opt);
```

图 4. 21 Put 方法

其中, selector, 如同上述 GET 方法中的查询参数, 支持=、lt、gt、lte、gte、exists 等参数及组合式的查询条件。

目标文档此处定义为需要更新的字段列表。或许在某些情况下，类似 `delete + insert` 操作的 `update` 操作是更合适的，但在移动客户端开发的过程中，我认为更新字段或字段列表才是更加经常的业务需求。

除了 `selector` 和 `document`，`update` 操作还支持两个可选参数

`upsert(Boolean)`:即 `update + insert`，默认情况下若 `update` 操作未发现符合 `selector` 条件的文档时，`update` 操作以失败结束，若开启 `upsert` 选项，则在查询无果的情况下，执行 `insert` 操作，将目标文档作为一条新的文档数据插入 `collection` 中

`multi(Boolean)`:默认情况下，`update` 操作更新 `collection` 中第一个符合 `selector` 的文档为目标文档，若开启 `multi` 选项，则 `update` 操作会对所有符合 `selector` 的文档进行更新

## DELETE 方法

HTTP DELETE 方法映射文档资源的 REMOVE 方法，使用 `DELETE db/collection` 的形式，进行文档删除操作

```
desc "Delete the item from the collection"
delete do
  selector=extract_selector_from_params
  if @db.collection(params[:collection]).remove(selector, {})
    {} # FIXME: we probably want to just return nil (i.e. null) here, but this is not parsable by JSON.parse()
  else
    error!("error on remove", 500)
  end
end
```

图 4. 22 Delete 方法

DELETE 方法与 UPDATE 方法极为类似，删除 `collection` 中符合 `selector` 描述形式的文档，同样的，`selector` 支持 `=`、`lt`、`lte`、`gt`、`gte`、`exists` 等操作。

通过上述讨论和介绍可以看出，本系统中 RESTFUL API 操作的最小粒度是 `Collection` 而非文档，这看似违反了 REST 规则中为所有资源确定唯一路径的要求。

这个问题可以从两个方面来看。一方面，本系统的 REST API 的底层实际上也包含了对文档资源的 REST 封装，如图是对指定 id 的文档的修改操作：

```
desc "Replaces the given attributes in the item with id :id in the collection"
patch('/:id' do
  id = BSON::ObjectId(params[:id])
  data = extract_data_from_params

  item = @db.collection(params[:collection]).find_one(id)

  unless item
    error!("Item #{params[:id].inspect} doesn't exist in #{params[:collection].inspect}!", 404)
  end

  data.delete 'id'
  data.delete '_id'

  #@db.collection(params[:collection]).save(item)
  @db.collection(params[:collection]).update({"_id" => id}, {"$set" => data})

  # FIXME: save ourselves the extra query and just return `item`?
  @db.collection(params[:collection]).find_one(id)
end
```

图 4.23 对指定 id 的文档的修改

这些接口暂时没有暴露，若开发扩展需要，可以很方便的修改至可用状态。另一方面，这么做的主要因素，不向移动端提供对文档的 REST 接口，是为了弱化关系型数据库中主键的概念，继而弱化表结构的概念，减少配置和改动带来的成本。

具体说来，Mongodb 文档的物理主键是文档插入时形成的 id 字符串，形如“4deeb1d9349c85523b000001”等，具有唯一性，这是 REST uri 的直接选择。但此 id 对于前端业务来说是没有意义的，加之 REST 是无状态的协议，使用此 id 进行对资源的操作必然增大了开发过程的复杂性。解决之道看似很简单，即使用逻辑主键，类似关系数据库中我们定义的主键。但是一方面文档型数据库 mongodb 不支持逻辑主键的定义，一方面定义逻辑主键再次把我们拉回了数据库定义-服务器模型-客户端模型的开发过程，违背了系统便捷轻巧的初衷。综合考虑我决定将这个功能对客户端层面舍去，并采用增强 GET 的方式作为补偿。

#### 4.4.3 使用 HMAC 方式提供高效率的认证

通过构建 RESTFUL 的 mongodb 接口，实现了对文档数据的 CRUD 操作，REST 是无状态的协议，若通过 Cookie 的方式进行身份验证，则破坏了无状态性，这一方面加大了服务器的压力，一方面增加了用户使用成本。

参考微软、谷歌、亚马逊等公司的做法，这个问题可以使用 HMAC 的解决办法解决。HMAC 全称 Hash-based Message Authentication Code. HMAC 保证消息的合法来源，而不对消息加密，使用轻量级的方法保证消息安全性。

加入 HMAC 的 REST API 工作过程大致如下：

- (1) 客户端与服务端事先约定用户名和密钥并各自存储
- (2) 客户端向服务端发送消息时，通过 HASH 算法以用户名和密钥+请求资源的唯一路径为输入，输出一个散列值
- (3) 客户端将用户名和散列值附加于 HTTP 请求的 HTTP 头中，通过 4.4.3 中所述 REST API 与服务器交流
- (4) 服务器通过用户名查询约定密钥，并通过与客户端相同的 HASH 算法对用户名、密钥、请求资源 URI 进行散列，输入出散列值。
- (5) 若服务端散列值匹配 HTTP 头中获取的散列值，则认证成功，执行相关请求，反之返回失败信号。

#### 4.4.4 服务器的选择和使用反向代理的方式部署应用服务器

在 Ruby 的服务器部署方面，系统选择使用 Nginx 和 Passenger 的组合。实际上，对于 Ruby 的应用服务器，我们可以有多种选择，可能是 FastCGI，或者是 Mongrel，Thin，Passenger 等。然后通过 Apache，Nginx，Lighttpd，HAProxy 之类的 Web 服务器访问我们的应用服务器，这个访问可以直接通过 HTTP 协议，也可以是 FastCGI，或者是自定义的其它协议，如此这般，我们便可以通过我们制定的访问协议访问我们的服务器程序了。

对于前端的 Web Server，Apache 是事实上的工业标准，在 Web 服务器市场，是占有率最高的，全球大量的网站采用 Apache 来部署他们的应用，Apache 是一款成熟稳定的 Web 服务器，功能非常强大，提供对几乎所有 Web 开发语言和框架的扩展支持，在对 Rails 框架的支持上，我们可以采用 mod\_fcgid 模块，通过 FastCGI 协议与 Rails 进程通讯。或者利用 mod\_proxy\_balancer 对后端的独立的 Rails 服务器如 Thin Cluster，Mongrel Cluster 或者 Apache/Nginx+Passenger 进行 HTTP 分发。但 Apache 作为一个通用的服务器，



在性能上和一些轻量型的 Web 服务器相差甚远。Apache 的 mod\_proxy\_balancer 模块的分发性能不高，比 Nginx 或者 HAproxy 都相差很多，另外，Apache 目前并不支持 Event（事件）模型，它仅支持 Prefork（进程）模式和 Worker（线程）模式，每处理一个链接，就需要创建一个进程或线程，而一些轻量级 Web 服务器如 Nginx 和 Lighttpd，则都很好地利用内核的事件机制提高性能，极大减少线程或进程数量，降低系统负载。

Nginx 是一个轻量级的高效快速的 Web 服务器，它作为 HTTP 服务器和反向代理服务器时都具有很高的性能。Nginx 可以在大多数 Unix like OS 上编译运行，并有 Windows 移植版。Nginx 选择了 Epoll 和 Kqueue 作为开发模型，它能够支持高达 50,000 个并发连接数的响应，可以在内部直接支持 Rails 和 PHP 程序对外进行服务。另外，Nginx 作为负载均衡服务器，也可以支持作为 HTTP 代理服务器对外进行服务，Nginx 不论是系统资源开销还是 CPU 使用效率都比 Apache 要好很多。Nginx 本身就是一个安装非常简单，配置文件非常简洁，甚至可以在配置文件中使 Perl 语法。

在处理静态文件上，Apache 和 Nginx 都可以胜任。但对于应用服务器或者是前端的负载均衡服务器，Nginx 是我们更好的选择。

对于应用服务器，Mongrel 一度是最流行的部署方式，它的 HTTP 协议的解析部分是用 C 语言编写的，效率上有所保证。Mongrel 使用了 Ruby 的用户线程机制来实现多线程并发，但是 Ruby 并不是本地线程，Rails 也不是线程安全的，因此 Mongrel 在执行 Rails 代码的过程中，完全是加锁的状态，那和单进程其实也没有太大差别。所有我们在使用 Mongrel 来部署 Rails 应用程序时，一般是在后台启动一个 mongrel\_cluster 来启动多个 Mongrel 进程。

Passenger 是类似于 mod\_php 的 Rails 运行环境，而不是 Mongrel 那样是独立的 Http 服务器。Passenger 对目前主流的 apache 和 Nginx 两大 Web 服务器都有支持。Passenger 使用起来极其方便，而且它具有较高的性能，从 Passenger 官方网站公布的测试结果来看，Passenger 的性能要优于 Mongrel 服务器，目前来说，Passenger 无疑是最好的选择。Passenger 继承了 Rails"不重复

自己"的惯例，通过 Passenger 部署应用程序，仅仅需要将 Rails 项目程序文件上传到目标服务器，甚至都不需要重启服务器，非常简单。

基于上述讨论，从性能和灵活性两方面综合考量，系统使用 Nginx 作为 HTTP 服务器，使用 Passenger 作为应用服务器。

由于 Nginx 是不支持加载插件的，使用 Passenger 需要重新编译 Nginx，使编译产出带有 Passenger 的 Nginx 服务器。使用如下命令进行上述过程：

```
gem install passenger
```

```
Passenger-install-nginx-module
```

如图

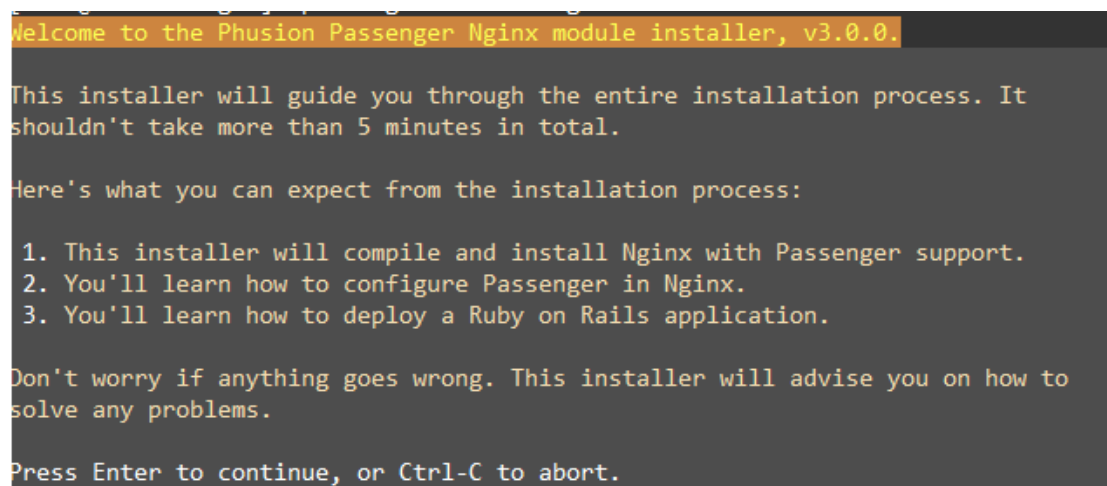


图 4.24 安装 Nginx

在 Nginx 配置文件中插入如下代码，连接 Nginx 和后端 Passenger 服务

```
http { ...
```

```
server { listen 80;
```

```
server_name www.omg.com;
```

```
root /var/www/omg/public;
```

```
passenger_enabled on;    }

... }
```

此处使用反向代理的好处是显而易见的，反向代理充分利用了 Nginx 处理静态数据的性能优势，可以将负载均衡和代理服务器的高速缓存技术结合在一起，将静态请求交付 Nginx 处理，将动态请求代理给 Passenger 处理，提高了系统性能。同时，反向代理为系统后端的扩展和负载均衡的优化提供了可能，具备额外的安全性，外部客户不能直接访问真实的服务器。将负载可以非常均衡的分给内部服务器，不会出现负载集中到某个服务器的偶然现象。

#### 4.4.5 基于 Countly-Server 构建统计服务器

不论从应用信息跟踪还是从用户行为分析的角度来说，一个完备的移动应用统计服务器都是十分有必要的。在上述基础架构实现之后，我们在 Countly-Server 的基础上构建统计服务器。一个完整的移动统计工具至少包括两本分：移动应用统计服务器和移动应用本地开发 SDK，本章主要讨论服务器部分，SDK 相关讨论将在 4.4.6 中进行。

Countly 是一个移动分析平台，是首个开源的移动分析的解决方案，任何人都可以将 Countly 客户端部署在自己的服务器并将开发工具包整合到他们的应用程序中，从而开始使用。

概括说来，一个统计过程就是一个客户端向服务端发送数据的过程。这种数据格式不便事先确定、数据量大、单条数据价值不高的数据特点，很适合使用我们前文所述的 MongoDB 或其服务器集群对其进行处理。

概括说来，一个分析过程就是一个将数据库存储数据以可视化形式从多个角度展示给用户的过程。分析过程有方便获取、访问快捷、形式直观的需求特点，这些特点使得分析服务很适合在我们前文所述的 Nginx 服务器上搭建 Web 应用进行实现。

统计系统的层次结构如图：

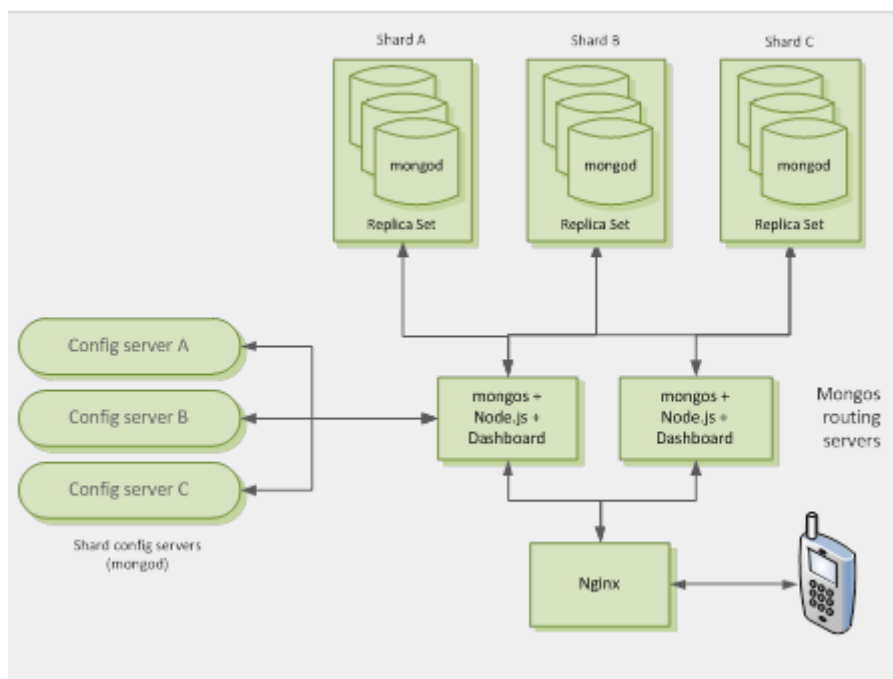


图 4.25 Countly 架构

在服务器实施过程中，为更好得管理服务器进程，保持系统的持续运行，我们使用 Supervisor 进行服务进程管理。通过 apt-get 工具安装 supervisor

```
sudo apt-get install supervisor
```

在系统的 /etc/init/ 目录建立 supervisor 启动配置文件 countly-supervisor.conf 关键代码如下：

```
description    "countly-supervisor"

start on runlevel [2345]
stop on runlevel [!2345]

respawn

exec /usr/bin/supervisord --nodaemon --configuration YOUR_PATH_TO_COUNTLY/countly/bin
```

图 4.26 配置 supervisor

在配置文件的指定位置建立服务器进程管理配置文件 supervisord.conf，其关键代码如下：

对 supervisord 的配置：

```
[supervisord]
logfile=/var/log/supervisord.log
logfile_maxbytes=50MB
logfile_backups=10
loglevel=warn
pidfile=/var/log/supervisord.pid
nodaemon=false
minfds=1024
minprocs=200
user=root
childlogdir=/var/log/
```

图 4.27 配置 supervisord

对统计服务控制前端的配置:

```
[program:countly-dashboard]
command=node %(here)s/../../frontend/express/app.js
directory=.
autorestart=true
redirect_stderr=true
stdout_logfile=%(here)s/../../log/countly-dashboard.log
stdout_logfile_maxbytes=500MB
stdout_logfile_backups=50
stdout_capture_maxbytes=1MB
stdout_events_enabled=false
loglevel=warn
```

图 4.28 配置统计服务器前端

对统计服务数据后端的配置:

```
[program:countly-api]
command=node %(here)s/../../api/api.js
directory=%(here)s
autorestart=true
redirect_stderr=true
stdout_logfile=%(here)s/../../log/countly-api.log
stdout_logfile_maxbytes=500MB
stdout_logfile_backups=50
stdout_capture_maxbytes=1MB
stdout_events_enabled=false
loglevel=warn
```

图 4.29 配置统计服务器后端

对 Nginx 进行配置实现反向代理, 将 80 端口的相关请求代理到应用服务器监听的 3001 端口, 关键代码如下:

```

access_log off;

location = /i {
    proxy_pass http://127.0.0.1:3001;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header X-Real-IP $remote_addr;
}

location ^~ /i/ {
    proxy_pass http://127.0.0.1:3001;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header X-Real-IP $remote_addr;
}

location = /o {
    proxy_pass http://127.0.0.1:3001;
}

location ^~ /o/ {
    proxy_pass http://127.0.0.1:3001;
}

location / {
    proxy_pass http://127.0.0.1:6001;
    proxy_set_header Host $http_host;
}

```

图 4. 30 Nginx 反向代理

上述过程被封装成一个可以在 Ubuntu Linux Server 上运行的自动脚本, 极大简化了统计服务器部署和整合的难度。此时, 我们启动 Supervisor 和 Nginx 服务器。访问域名或本机 ip 即可配置统计服务器和添加移动应用:



图 4. 31 Countly 运行成功

#### 4.4.6 Mobile First 的移动端 SDK 设计

框架的最终目的是方便移动应用前端开发人员的使用，因此整个系统的设计和实现都优先考量移动应用的开发情境。移动端的 SDK 作为提供给前端开发人员的系统产品，在尽量减少开发人员的编码难度和提高编程体验方面做了很大努力。移动端 SDK 的主要功能是协助开发人员添加数据统计；快速转换数据库模型，方便进行数据的存储查询；封装常用功能，方便快速开发。关键技术如下：

提供 Android 端、IOS 端统计接口。统计的本质是客户端向服务器发送带有事件标示、移动设备唯一标示符和时间戳的一条消息。发送消息与获取时间戳的方式比较简单，但对于设备唯一标示符，参考 OPEN\_UDID 的设计，Android 与 IOS 有不同的实现。在 Android 系统中，设备唯一标示符是以系统的 Android\_ID 为基础产生的。关键代码如下：

```

/*
 * Generate a new OpenUDID
 */
private void generateOpenUDID() {
    if (LOG) Log.d(TAG, "Generating openUDID");
    //Try to get the ANDROID_ID
    OpenUDID = Secure.getString(mContext.getContentResolver(), Secure.ANDROID_ID);
    if (OpenUDID == null || OpenUDID.equals("9774d56d682e549c") || OpenUDID.length() < 15 ) {
        //if ANDROID_ID is null, or it's equals to the GalaxyTab generic ANDROID_ID or bad,
        final SecureRandom random = new SecureRandom();
        OpenUDID = new BigInteger(64, random).toString(16);
    }
}

```

图 4.32 确定 Android 设备标示符

同时，利用 Android 的 Service 机制，开启一个远程 Service，共享 UDID 服务，实现使用本统计服务的不同程序可共享相同的设备标示符产生办法。关键代码如下：

```

@Override
public void onServiceConnected(ComponentName className, IBinder service) {
    //Get the OpenUDID from the remote service
    try {
        //Send a random number to the service
        android.os.Parcel data = android.os.Parcel.obtain();
        data.writeInt(mRandom.nextInt());
        android.os.Parcel reply = android.os.Parcel.obtain();
        service.transact(1, android.os.Parcel.obtain(), reply, 0);
        if (data.readInt() == reply.readInt()) //Check if the service returns us this number
        {
            final String _openUDID = reply.readString();
            if (_openUDID != null) { //if valid OpenUDID, save it
                if (LOG) Log.d(TAG, "Received " + _openUDID);

                if (mReceivedOpenUDIDs.containsKey(_openUDID)) mReceivedOpenUDIDs.put(_openUDID,
                    else mReceivedOpenUDIDs.put(_openUDID, 1);
            }
        }
    } catch (RemoteException e) {if (LOG) Log.e(TAG, "RemoteException: " + e.getMessage());}
    mContext.unbindService(this);

    startService(); //Try the next one
}
    
```

图 4.33 开启 Android 端服务

在 IOS 系统中，通常获取设备唯一标示的方法是直接读取设备的 UDID 值，然而在 IOS5 版本的系统中，苹果标示为保证用户隐私安全将弃用 UDID。继续使用 UDID 作为唯一标示符将面临应用在未来 IOS 系统版本出现兼容性问题、应用 App Store 审核不通过和复审被下架的风险。在此我们使用苹果推荐的替代方式产生 UUID 作为替代：

```

if (_openUDID==nil) {
    CFUUIDRef uuid = CFUUIDCreate(kCFAllocatorDefault);
    CFStringRef cfstring = CFUUIDCreateString(kCFAllocatorDefault, uuid);
    const char *cStr = CFStringGetCStringPtr(cfstring, CFStringGetFastestEncoding(cfstring));
    unsigned char result[16];
    CC_MD5( cStr, strlen(cStr), result );
    CFRelease(cfstring);
    CFRelease(uuid);

    _openUDID = [NSString stringWithFormat:
        @"%02x%02x%02x%02x%02x%02x%02x%02x%02x%02x%02x%02x%02x%02x%02x%02x",
        result[0], result[1], result[2], result[3],
        result[4], result[5], result[6], result[7],
        result[8], result[9], result[10], result[11],
        result[12], result[13], result[14], result[15],
        (NSUInteger)(arc4random() % NSUIntegerMax)];
}
    
```

图 4.34 产生 IOS 设备唯一标示符

我们讲新生成的标示符存储，作为针对本应用的设备唯一标示。

封装公共接口和常用方法。除统一标示符接口外，客户端 SDK 还附带了一些开发者常用方法。例如 LBS 相关的通过 wifi、gps、基站三种方式定位用户位置的通用定位方法，关键代码如下（以 wifi 方式为例）：



```

137     private JSONObject doWifi() throws Exception {
138         JSONObject holder = new JSONObject();
139         holder.put("version", "1.1.0");
140         holder.put("host", "maps.google.com");
141         holder.put("address_language", "zh_CN");
142         holder.put("request_address", true);
143
144         WifiManager wifiManager = (WifiManager) context.getSystemService(Context.WIFI_SERVICE);
145
146         if(wifiManager.getConnectionInfo().getBSSID() == null) {
147             throw new RuntimeException("bssid is null");
148         }
149
150         JSONArray array = new JSONArray();
151         JSONObject data = new JSONObject();
152         data.put("mac_address", wifiManager.getConnectionInfo().getBSSID());
153         data.put("signal_strength", 8);
154         data.put("age", 0);
155         array.put(data);
156         holder.put("wifi_towers", array);
157
158         return holder;
159     }

```

图 4. 35 定位方法

bitmap 工具包封装了对图像文件的常用操作



图 4. 36 bitmap 工具包

无论 Android 还是 IOS, sqlite 作为优秀的嵌入式数据库都是其首选的本地存储工具。DBOMG 模块是一个针对 SQLITE3 的 ORM 框架, 它将本地存储的方式简化, 开发者不必书写 SQL, 即可实现对 sqlite3 的简单操作, 它向开发者提供了如下接口, 涵盖 sqlite3 的 CRUD 以及自定义查询操作:

```

8      public abstract long insert(T entity);
9
10     public abstract void delete(int id);
11
12     public abstract void delete(Integer... ids);
13
14     public abstract void update(T entity);
15
16     public abstract T get(int id);
17
18     public abstract List<T> rawQuery(String sql, String[] selectionArgs);
19
20     public abstract List<T> find();
21
22     public abstract List<T> find(String[] columns, String selection,
23                                String[] selectionArgs, String groupBy, String having,
24                                String orderBy, String limit);
25
26     public List<Map<String, String>> query2MapList(String sql,
27                                                    String[] selectionArgs);
28
29     public void execSql(String sql, Object[] selectionArgs);

```

图 4.37 sqlite 工具接口

封装通信方法。系统主体基于 REST API 构建，使得无论 Android 客户端还是 IOS 客户端，与服务器都是主要使用 HTTP 协议进行通讯。为此 Android 与 IOS 端 SDK 分别对通信协议进行了封装。Android 端使用 Apache Http Client 库作为通信的基础库，使用枚举以及 java 的多态性对 Http 方法进行封装：

```

public enum HttpRequestMethod {
    HttpGet {
        public HttpUriRequest createRequest(String url) {return new HttpGet(url);}
    },
    HttpPost {
        public HttpUriRequest createRequest(String url) {return new HttpPost(url);}
    },
    HttpPut {
        public HttpUriRequest createRequest(String url) {return new HttpPut(url);}
    },
    HttpDelete {
        public HttpUriRequest createRequest(String url) {return new HttpDelete(url);}
    },
    HttpPatch {
        public HttpUriRequest createRequest(String url) {return new HttpPatch(url);}
    };

    public HttpUriRequest createRequest(String url) {
        return null;
    }
}

```

图 4.38 对 HTTP 方法的封装

而后使用统一的请求方法，判断不同的请求类型，自动组装相应的请求体格式：

```

public static String sendRequest(HttpRequestMethod methodEnum, String url,
    Map<String, String> paramsMap, String stringEntity) {
    DefaultHttpClient httpClient = new DefaultHttpClient();
    httpClient.getParams().setParameter(HttpConnectionParams.SO_TIMEOUT,
        Integer.valueOf(5000));
    httpClient.getParams().setParameter(
        HttpConnectionParams.CONNECTION_TIMEOUT, Integer.valueOf(3000));
    String result = null;
    HttpUriRequest req = null;

    try {
        if (paramsMap != null) {
            if (methodEnum == HttpRequestMethod.HttpPost)
            {
                req = methodEnum.createRequest(url);
                ((HttpEntityEnclosingRequest) req)
                    .setEntity(new UrlEncodedFormEntity(
                        keyValueToValuePairList(paramsMap),
                        DEFAULT_REMOTE_ENCODE));
            }
            else {
                url+="?" + URLEncodedUtils.format(keyValueToValuePairList(paramsMap),
            }
        }
        if (req == null) {
            req = methodEnum.createRequest(url);
        }

        if (stringEntity != null) {
            ((HttpEntityEnclosingRequest) req)
                .setEntity(new StringEntity(stringEntity));
        }
    }
}

```

图 4.39 自动组装请求格式

IOS 端，使用比较成熟了的 ASI\_HTTP\_Request 实现 http 通信。其中设置 http 方法及代码如下：

```

617 - (void)setRequestMethod:(NSString *)newRequestMethod
618 {
619     [[self cancelledLock] lock];
620     if (requestMethod != newRequestMethod) {
621         [requestMethod release];
622         requestMethod = [newRequestMethod retain];
623         if ([requestMethod isEqualToString:@"POST"] || [requestMethod isEqualToString:@"PUT"] || [postBody length]
624             [self setShouldAttemptPersistentConnection:NO];
625     }
626 }
627 [[self cancelledLock] unlock];
628 }

```

图 4.40 ASI\_HTTP 方法

同时可以使用 appendPostData 的方式为 http 方法追加请求体：

```

- (void)appendPostData:(NSData *)data
{
    [self setupPostBody];
    if ([data length] == 0) {
        return;
    }
    if ([self shouldStreamPostDataFromDisk]) {
        [[self postBodyWriteStream] write:[data bytes] maxLength:[data length]];
    } else {
        [[self postBody] appendData:data];
    }
}
    
```

图 4.41 追加请求体

对象自动序列化反序列化。以 JSON 作为客户端与服务器交互的主要数据格式，决定了客户端 model 类的对象需要序列化为 JSON 向服务器传递，服务器返回数据需反序列化为对象，供客户端程序使用。正因如此，客户端 SDK 的对象自动序列化反序列化机制，成为了开发人员可以高效利用本框架的重要保证。具体看来：Android 端，使用 Java 的反射机制，对于继承了框架中 OmgModel 的类，通过 Reflect 相关方法获取其字段名和字段类型，形成 JSON 模板，获取特定对象的属性值，与字段名和字段属性形成 JSON 串，实现了 Model 对象的自动序列化支持。将 List 接口自动转换为 JSON 的 Array 形式，并将这个过程递归得向下进行，实现了对嵌套对象的序列化支持。整个过程对编码人员是完全透明的，编码人员享受这个过程而无需编写一行额外的代码，仅需按标准的方式继承 OmgModel 类并定义所需属性即可。反序列化是上述过程的逆过程，此时需传入目标 Model 的 Class 对象，通过反射方法创建 Model 对象，并将同名的键值对赋值给同名属性，返回组合好的 model 对象，实现反序列化。如图一个 Student Model：

```

public static class student extends OmgModel{
    private static final long serialVersionUID = -6778289744214333731L;

    private String name;
    private int age;
    private int grade;
    public String getName() {
        return name;
    }
}
    
```

图 4.42 Model 的示例

直接调用 student.save 方法即可讲 student 对象保存在远端。IOS 端，除提供与 Android 类似的通过反射实现的自动序列化反序列化方法外，还提供了可自定

义数据格式的序列化，一个通过字段配置解析 JSON 过程如下：

```
- (void)loadArticles
{
    RKObjectMapping* articleMapping = [RKObjectMapping mappingForClass:[Article class]];
    [articleMapping addAttributeMappingsFromDictionary:@{
        @"title": @"title",
        @"body": @"body",
        @"author": @"author",
        @"publication_date": @"publicationDate"
    }];

    RKResponseDescriptor *responseDescriptor = [RKResponseDescriptor responseDescriptorWithMapping:articleMapping pathPattern:nil];

    NSURL *URL = [NSURL URLWithString:@"http://restkit.org/articles"];
    NSURLRequest *request = [NSURLRequest requestWithURL:URL];
    RKObjectRequestOperation *objectRequestOperation = [[RKObjectRequestOperation alloc] initWithRequest:request responseDescriptor:responseDescriptor];
    [objectRequestOperation setCompletionBlockWithSuccess:^(RKObjectRequestOperation *operation, RKMappingResult *mappingResult) {
        RKLogInfo(@"Load collection of Articles: %@", mappingResult.array);
    } failure:^(RKObjectRequestOperation *operation, NSError *error) {
        RKLogError(@"Operation failed with error: %@", error);
    }];

    [objectRequestOperation start];
}
```

图 4.43 解析配置

使用操作对象封装 Selector 参数，使用选项对象封装 Options 参数。参考第三章所述目前已有移动应用后端服务系统，其前端数据查询功能或比较简单（只能通过 id 查询，实用性不大），或功能强大但编码不直观，需要进行对 SDK 的进一步学习。在本系统的数据查询功能中，通过 Data Access Tool 类 OmgDAT 实现了对数据的 CRUD 操作，对于操作的选择器，使用 OmgSelector 进行抽象，对于操作参数，使用 OmgOpt 进行抽象，使用户在编写查询器和添加选项参数时不必手写 JSON，既提高了内聚性减小了无效查询出现的概率又方便了用户编码。例如进行一个这样的查询：查询所有名字叫 moneky 的同学，选择他们其中年龄小于或者等于 15 的同学，将查询结果按年龄降序排序，如果年龄相同再按姓名降序排序，跳过前 8 条数据（例如我们在 app 的第一页已经使用过了这个数据），使用后来最多 8 条数据（例如我们的 app 需要分页，每页最多 8 条），把结果的 name 和 age 字段序列化成学生对象，代码仅需如下：

```
OmgSelector s=new OmgSelector();
s.addField(new OmgField("name").is("monkey"));
s.addField(new OmgField("age").lte(15));
OmgOption opt=new OmgOption();
opt.limit(8).sort("age", false).sort("name", false).skip(8).field(new String[]{"name", "age"});

ArrayList<ani> arr=OmgDAT.load(ani.class, s, opt);
```

图 4.44 查询代码示例

编写上述代码的过程是愉快的，当我们想在对年龄的条件中加一条件，例如年龄

小于等于 15，大于某个值，可直接追加点号和方法，编码过程大概是这样的：

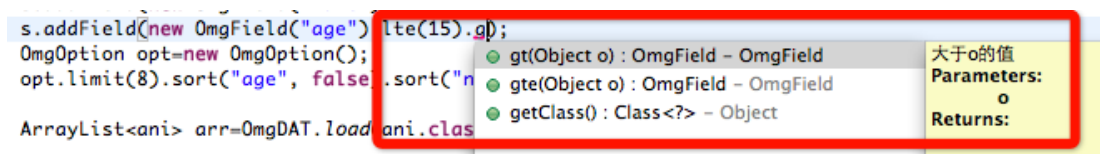


图 4.45 追加查询条件

## 第 5 章 通用性和可用性说明

### 5.1 通用性

系统针对移动应用开发中面临的共同问题而设计。数据存储和统计服务是移动应用开发过程中面临的两大共同问题，前端人员普遍只掌握特定客户端技术而不具备整体的服务端技术，系统以解决这个问题为目标设计，是尝试解决一个开发过程中广泛存在的问题，具有普遍性。

系统的抽象性保证系统在一定的移动应用开发领域里是通用的。系统在设计 and 实现的过程中，始终站在数据和传输的层面考虑，未涉及具体的业务和特定的逻辑，这种抽象性是系统通用性的保证。

系统的无表结构设置使系统的数据层面可适用于通用的环境。系统从Mongodb的无表结构的数据存储形式展开，从服务器到客户端的各个层面强调数据结构和数据模型，忽略关系型数据库中的表结构概念，对数据模型的存储在首次存储时动态创建集合和数据结构，使系统在数据层面具备了通用性。

### 5.2 可用性

系统分布式数据库结构增强了数据安全性，为扩展和数据库性能性能优化提供了可能。

系统的服务端使用Nginx+Passenger部署，使用Ruby编写，结合了Web服务器与应用服务器的优势，使用业界成熟的技术，服务端较为稳定。

系统的统计服务，使用先进的County-server相关技术，其前端图表工具可为开发者数据分析人员带来巨大便利：



图 4. 46 Countly 前端统计

系统的客户端SDK适用于Android与IOS平台，有效地屏蔽了数据传输、对象序列化反序列化操作，使操作过程对前端开发人员透明，提供了符合语言特性的SDK开发接口，使前端开发人员无需关心后台实现即可轻松使用相关接口完成开发。



## 第6章 结论

### 6.1 系统设计和实现的总结

本文通过对先有移动应用后端服务系统的研究,以及对移动应用前后端文献的学习,根据移动应用开发的特点,结合当前环境下移动应用开发者面临的一系列问题,设计和实现了一个以数据存储为主体的移动应用后端服务框架。该框架能够对移动应用前端开发者屏蔽数据处理的相关逻辑,使其忽略服务器后端逻辑而专注于前端交互和业务逻辑的开发,通过集成本框架,前端开发者可以轻松获得数据存储查询、移动数据统计分析、LBS工具、移动应用常用本地工具、sqlite3本地数据缓存等的服务。本系统在设计和实现的过程里使用了Nginx、Mongodb、Passenger、Rails、Supervisor等比较成熟的服务端技术,使系统无论从性能还是稳定性方面考量都具备了一定的实用性。

### 6.2 移动应用后端服务发展展望

从2010年,第一个移动应用后端服务提供商StackMob出现至今,短短3年时间,移动应用后端服务产业迅速发展,出现了大大小小的移动应用后端服务提供商和以移动应用后端服务为立足点的创业团队,随着产业的不断发展,移动应用后端服务已从小众的垂直领域迅速成长为移动行业中的一个重要产业环节。移动应用后端服务的发展正在促进移动开发领域的分工细化,移动应用前端开发者,正在从后端技术中解放,得以以更少的精力关注后端技术而专注前端实现。

但是到目前为止的移动应用后端服务提供商,提供的服务还多是局限于数据统计和数据存储两方面,移动应用的后端依然存在着大量重复性工作。怎样才能 在坚固易用性和性能的基础上,进一步减少前端人员对后端业务的关注度,将后端云服务更好的和移动应用的开发过程结合起来,是一个值得深入研究的领域, 这些都可以作为我们以后研究和奋斗的方向。

## 致 谢

经过了大概三个月的时间，通过对移动应用后端服务相关技术的学习、参考相关书籍和网上信息、资料的查找，以及在老师和同学们的指导和帮助下，我得以顺利的完成了本科毕业设计。在此表示我由衷的谢意。

感谢山东大学我的指导老师闫中敏，感谢闫老师在本课题的开题、设计和撰写论文期间，给予我的宝贵意见和督促、指导，为我能够完成毕业设计提供了良好的实践环境。

感谢我的实习单位百度公司为我提供了良好的实践环境提供了优秀的开发设备，感谢我的公司实习导师和公司同事们，他们在百忙之中抽出时间对我在毕业设计中遇到的难题进行细致的分析与指导，并且在指导的过程中悉心负责，在毕业设计的开始阶段，他们为我的课题选择进行了严谨的分析，为我的毕业设计指明了方向，引导我的设计思路，关注设计的进度，改进程序的功能，使我无论从理论研究还是从实践动手两个方面都受益匪浅。他们渊博的专业知识、敬业的工作态度给予了我极大的鼓励。

感谢我的同学，我亲爱的舍友们，他们在毕设的过程中给予我很多的帮助和鼓舞，他们陪我度过了很美好的四年时光。

感谢我的父母，感谢二老对我的抚养和教育，感谢他们一直支持我的选择，感谢他们做我倾诉的朋友，动力的源泉，避风的港湾。在毕设的间隙时间，我常陷入一种回忆中，回忆父母恩情相伴的我的十几年的求学时光，而今儿子终于要毕业了。

感谢所有指导、帮助我完成毕业设计的专业课老师、教务员老师、辅导员老师，他们的帮助，使我的毕业设计得以顺利完成。

## 参考文献

- [1] Perrotta P. Ruby 元编程[M]. 1. 华中科技大学出版社, 2012.
- [2] 陶利军. 决战 Nginx 系统卷:高性能 Web 服务器详解与运维[M]. 1. 清华大学出版社, 2012.
- [3] 陶利军. 决战 Nginx 技术卷:高性能 Web 服务器部署与运维(基于 php、Java、ASP.NET 等)[M]. 1. 清华大学出版社, 2012.
- [4] Kyle Banker、丁雪丰. MongoDB 实战[M]. 1. 人民邮电出版社, 2012.
- [5] 费尔南德斯(Obie Fernandez). Rails 之道[M]. 1. 人们邮电出版社, 2012.
- [6] 马尔科·加尔根塔(Marko Gargenta)、李亚舟、任中龙、杜钢. Learning Android(中文版)[M]. 1. 电子工业出版社, 2012.
- [7] 邓凡平. 深入理解 Android(卷 1)[M]. 1. 机械工业出版社, 2011.
- [8] 钟冠贤(Carlo Chung)、刘威. Objective-C 编程之道:iOS 设计模式解析[M]. 1. 人民邮电出版社, 2011.
- [9] 艾伦 (Grant Allen)、欧文斯 (Mike Owens)、杨谦、刘义宣. SQLite 权威指南(第 2 版)[M]. 1. 电子工业出版社, 2012.
- [10] 阿拉马拉尤 (Subbu Allamaraju)、李锟、丁雪丰、常可. RESTful Web Services Cookbook(中文版) [M]. 1. 电子工业出版社, 2011.
- [11] 吴淼、倪力舜;一种针对 MongoDB 数据库的证据获取方法[J];中国司法鉴定;2011 年 03 期
- [12] 李莉莎;关于 NOSQL 的思考[J];中国传媒科技;2010 年 04 期
- [13] 杨磊;基于 NoSQL 数据库的结构化存储设计与应用[J];科技风;2011 年 18 期
- [14] 王晓玲、董逸生;面向 Web 的异构信息系统集成方案[A];第二十届全国数据库学术会议论文集(研究报告篇)[C];2003 年
- [15] 鲍海燕、朱学玲;ROR 在 Web 开发中综合运用的研究[J];硅谷;2009 年 23 期
- [16] 黄春芳、石浩波;ROR 网站敏捷开发[J];计算机时代;2007 年 10 期

## 附录

原文:

### Research on The Improvement of MongoDB Auto- Sharding in Cloud Environment

Abstract—With the rapid development of the Internet Web 2.0 technology,the demands of large-scale distributed service and storage in cloud computing have brought great challenges to traditional relational database.NoSQL database which breaks the shackles of RDBMS is becoming the focus of attention.In this paper, the principles and implementation mechanisms of Auto-Sharding in MongoDB database are firstly presented, then an improved algorithm based on the frequency of data operation is proposed in order to solve the problem of uneven distribution of data in auto-sharding.The improved balancing strategy can effectively balance the data among shards,and improve the cluster's concurrent reading and writing performance.

key words—NoSQL, MongoDB, Auto-Sharding, balance strategy

In recent years,with the rapid growth of the amount of data and the development of Internet web 2.0 technology, how to efficiently store, process and extract large amounts of data becomes an urgent problem, Cloud computing emerged in this context. Cloud computing is the delivery of computing as a service rather than a product, whereby shared resources, software, and information are provided to computers and other devices as a metered service over a network (typically the Internet)[1]. Many universities, vendors and government organisations are investing in research around the topic of cloud computing, for example: Amazon launched Simple Storage Service (S3) and Elastic Compute Cloud(EC2), Google proposed GFS, BigTable and MapReduce, which have all been successfully used in production

environment. Distributed File System can organize the huge amounts of data in cloud computing, and also be able to efficiently read the cloud data, but we still need specialized data management tools for better management of structured data.

Cloud Data Management is a new data management concept with the development of cloud computing, it must be able to efficiently manage of large data sets in the cloud, and quickly locate specific data in massive data sets, which makes the Cloud Data Management with the following common characteristics: (1) high concurrent read and write performance, (2) efficiently store and access huge amounts of data, and (3) high scalability and high availability requirements of the database. In the face of these demands, the traditional relational data management system(RDBMS) has encountered an insurmountable obstacle. Therefore, NoSQL database systems rose alongside major internet companies, such as Google, Amazon, Twitter, and Facebook which had significantly different challenges in dealing with data that the RDBMS solutions could not cope with. These companies realized that performance and real time nature was more important than consistency, which traditional relational databases were spending a high amount of processing time to achieve. As such, NoSQL databases are often highly optimized for retrieve and append operations and often offer little functionality beyond record storage. The reduced run time flexibility compared to RDBMS systems is compensated by significant gains in scalability and performance. Often, NoSQL databases are categorized according to the way they store the data and fall under categories such as key-value stores(e.g. Dynamo[2]), BigTableimplementations[3]anddocumentstore databases(e.g. MongoDB[4]). However, due to the immature technology of cloud data mangement, there are still many issues need to be addressed in actual production environment. This paper discusses the design principles and implementation mechanism of MongoDB database, and focuses on the principle of Auto-Sharding. The goal of Auto-Sharding is to split data up across machines and rebalance automatically making it possible to store more data and handle more load without requiring large or powerful machines. But the balancer algorithm isn't so intelligent that data isn't evenly distributed among servers. In order to solve this problem, an improved FODO (i.e. frequency of data

operation)algorithm is proposed. The FODO algorithm is based on the frequency of data operation and taking the load of server into consideration. The data balancing strategy based on FODO algorithm can effectively balance the data among servers, and improve the cluster's concurrent reading and writing performance.

#### A. Brief introduction of MongoDB

MongoDB (from "humongous") is an open source document-oriented NoSQL database system written in the C++ programming language. It manages collections of BSON documents. Development of MongoDB began in October 2007 by 10gen. MongoDB features[5]: Document-oriented storage, JSON-style documents with dynamic schemas offer simplicity and power; full index support; replication and high availability; Auto-Sharding, scale horizontally without compromising functionality; Map/Reduce, flexible aggregation and data processing; GridFS, store files of any size without complicating your stack.

At the heart of MongoDB is the concept of a document which is the basic unit of data for MongoDB, roughly equivalent to a row in a RDBMS. Similary, a collection can be thought of as the schema-free equivalent of a table. A single instance of MongoDB can host multiple independent databases, each of which can have its own collections and permissions.

#### B. Auto-Sharding architecture

Sharding refers to the process of splitting data up and storing different portions of the data on different machines. By splitting data up across machines, it becomes possible to store more data and handle more load without requiring large or powerful machines[6]. MongoDB supports Auto-Sharding, which eliminates some of the manual sharding, the cluster can split up data and rebalance automatically. MongoDB sharding provides: (1) automatic balancing for changes in load and data distribution, (2) easy addition of new machines without down time, (3) no single points of failure, and (4)Automatic failover. The basic concept behind MongoDB's sharding is to break up collections into smaller chunks. These chunks can be distributed across shards so that each shard is responsible for a subset of the total data set. To partition a collection, MongoDB specify a shard key pattern which names one

or more fields to define the key upon which we distribute data. Because of shard key, chunks can be described as a triple of collection, minkey and maxkey. Chunks grow to a maximum size, usually 200MB, once a chunk has reached that approximate size, the chunk splits into two new chunks.

The architecture of Auto-Sharding is displayed in Fig. 1.

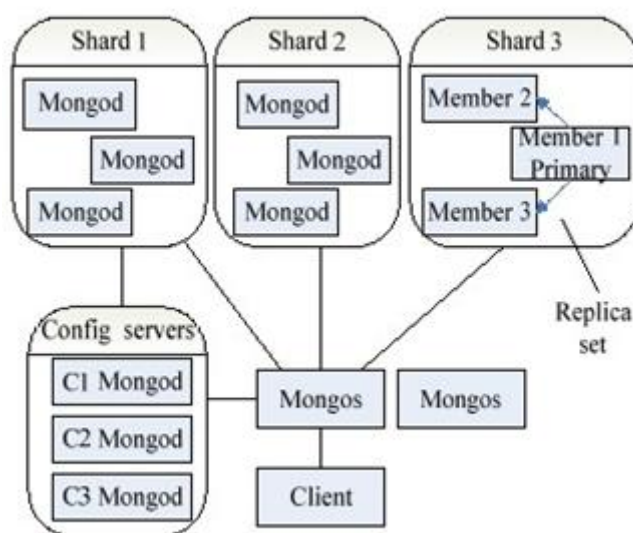


Figure. 1

A MongoDB shard cluster consists of two or more shards, one or more config servers, and any number of routing processes. Each of the components is described below.

1) **Shard:** Each shard consists of one or more servers and stores data using mongod processes. In production situation, each shard will consist of a replica set to ensure availability and automated failover.

2) **Config server:** It stores the cluster's metadata which includes basic information on each shard server and the chunks contained therein.

3) **Mongos (Routing Processes):** It can be thought of as a routing and coordination process. When receiving requests from client, the mongos routes the request to the appropriate server and merges any results to be sent back to the client.

### C. The balancing strategy of Auto-Sharding

MongoDB uses balancer to keep chunks evenly across all servers of the cluster. The unit of transfer is a chunk, and balancer waits for a threshold of uneven chunks counts to occur. In the field, having a difference of 8 chunks between the least and

most loaded shards showed to be a good heuristic. Once the threshold is reached, balancer will redistribute chunks, until that difference in chunks between any two shards is down to 2 chunks.

In order to reduce the amount of data transferred, each shard contains multiple ranges. When a new shard added into the cluster or some of the shards contain too much data to reach the threshold, the balancer will skim chunks off of the top of the most-populous shard and move these chunks to the least-populous shard, allowing the data evenly distributed in the cluster by moving the bare minimum.

The goal of the balancer is not only to keep the data evenly distributed but also to minimize the amount of data transferred. The balancer's algorithm isn't terribly intelligent. It moves chunks based on the overall size of the shard[7]. The migration chunks are just the ones located on the top of each shard, but the operation of data is not taking into consideration. The data transferred between shards may not often be used, it will makes the system load can not achieve an effective balance. It is necessary to improve the balancing strategy of Auto-Sharding in MongoDB.

#### A. Basic idea of FODO algorithm

In order to solve the problem of data's uneven distribution in auto-sharding, an improved algorithm (FODO algorithm) based on the frequency of data operation is proposed. In MongoDB, the unit of transfer is a chunk. There are n chunks in a shard, the ith chunk is represented as Ci and its frequency of data operation value is F\_DOi. In MongoDB database, the main operations of data are insert, find, update and delete. As the data in the cluster rarely be frequently deleted, the manipulations which affect system performance mainly concentrated in the first three operations. We use Ii, Fi and Ui to represent the number of these three kinds of operations to a chunk. So the value of F\_DOi in normalized form is (1)

$$F\_DO_i = \frac{I_i^{A,j}}{n^{A,j}} + \frac{F_i^{A,j}}{n^{A,j}} + \frac{U_i^{A,j}}{n^{A,j}}$$

$$\frac{\sum_{i=1}^{n^{A,j}} I_i^{A,j}}{\sum_{i=1}^{n^{A,j}} I_i^{A,j}} + \frac{\sum_{i=1}^{n^{A,j}} F_i^{A,j}}{\sum_{i=1}^{n^{A,j}} F_i^{A,j}} + \frac{\sum_{i=1}^{n^{A,j}} U_i^{A,j}}{\sum_{i=1}^{n^{A,j}} U_i^{A,j}}$$

However, the insert, find and update operations have different impacts on the cluster load and should not be treated equally. In Auto-Sharding environment, it's not



necessary to make a physical connection when query record from database because caches have already kept the fresh data. Moreover, each shard will consists of a replica set in production situation. The secondary nodes in a replica set can read the data written from the primary node, so it can lighten the querying load on the primary node to a certain extent. Contrarily, the insert and update operations result directly to database every time when they commit a transaction, which occupy a bigger part of the whole cluster workload. In particular, insert data will cause the number of data in each shard is different. Once the threshold is reached, balancer will balance data automatically, which will greatly consume system resources. So FODO algorithm adds a parameter inc (always >1) before inserting part in the equation, called insert coefficient, to put more weight on insert operation.

Here is the final FODO value definition in (2)

$$F\_DO_i = inc \cdot \frac{I_{i,j}}{n_{i,j}} + \frac{F_{i,j}}{n_{i,j}} + \frac{U_{i,j}}{n_{i,j}} \quad (2)$$

The F\_DO value of each shard is the sum of its containing chunk's F\_DOi value, that is (3)

$$s(F\_DO) = \sum_{i=1}^{n_{i,j}} c_i(F\_DO_i)_{i,j} \quad (3)$$

We need to modify data structures recording the chunk's information, adding I, F, U and F\_DOi four variables to store insert, find, update and F\_DOi values. Each operation to the data must be recorded in the four variable of the corresponding chunk.

## B. Balancing strategy of FODO algorithm

F\_DOi value indicates the frequency of data operations in each chunk. If a chunk's F\_DOi value is high, it means data in this chunk is frequently used. The threshold of data redistribution is still the difference in the number of chunks in each shard, but the migration chunks are selected based on its F\_DOi value. There are trade-offs between overall size of shard and operating frequency of data in this balancing strategy. The process of data balance has three steps.

- 1) The threshold of data migration: Calculate the number of chunks in each shard.

If the difference is greater than 8, then begin to balance the data until the difference is less than 2. Chunks will be migrated from the most-populous shard called from-shard to the least-populous shard called to-shard.

2) Choose the migration chunks: Calculate the difference of F\_DO value between from-shard and to-shard. If  $f(F\_DO) > t(F\_DO)$ , then choose the chunk with the maximum F\_DOi value in from-shard. Otherwise, choose the chunk with the minimum F\_DOi value in from-shard.

3) Migrate chunks: Migrate the selected chunk in step 2) to the to-shard, and recalculate the F\_DOi value of each chunks both in from-shard and to-shard. Repeat step 1).

The flowchart of data balance process is displayed in Fig. 2. Moreover, we should determine the value of inc. The role of inc is to put more weight on insert operation, so the value should be greater than 1. The specific value of inc relies on server-related parameters and system load, we can set the initial value and then adjust it according to the actual needs of business.

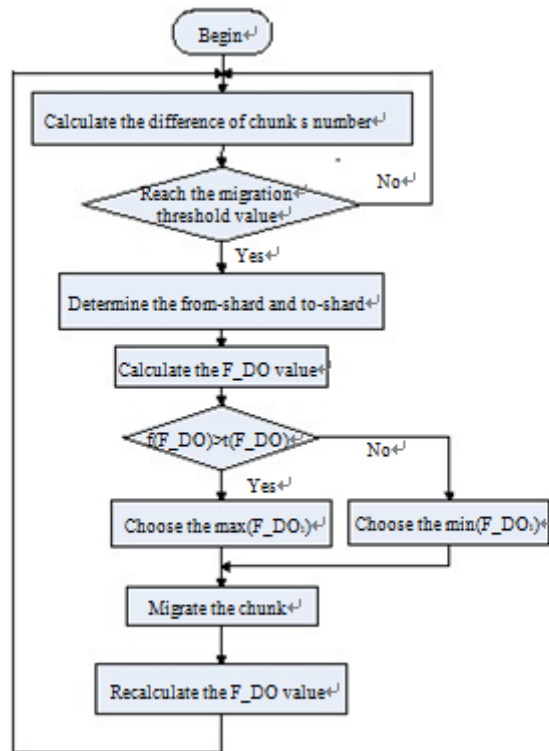


Figure. 2

The test environment based on the MongoDB Auto-Sharding cluster, it contains

10 virtual machines, each of which has the same hardware configuration: 8GB of memory and CPU speed is 2.4G HZ. Each virtual machine has a Linux RedHat operating system, and be connected by the 100Mbps LAN. The MongoDB version is 1.8.2, and the Auto-Sharding cluster has three shards, each of which consists of a replica set.

Each replica set contains three nodes: one primary node and two secondary nodes. We should run the `rs.slaveOk()` command on the secondary nodes in order to query data from them.

We realize the FODO algorithm in MongoDB, and compare the two algorithm by testing the concurrent read and write performance of the cluster. The data set used in test is a simple line data, including four fields of int, long, string and double. In addition, the value of inc parameter in FODO algorithm is 1.5. In order to see the effect of the FODO algorithm, we only add two shards, insert 1000000 records and use randomly generated id to do find and update operations. Then we add the third shard to the cluster. The purpose of this action is to ensure the cluster has data at the beginning of the test and the value of F\_DO is not zero.

Firstly, we test the concurrent writing performance of the cluster. We insert 10000000 records into the cluster and remain the overall amount of records be same under different number of concurrent. The concurrent writing performance of this two algorithm is shown in the Fig. 3.

Remain the concurrent number and records be unchanged, and test the concurrent reading performance of the cluster. The concurrent reading performance of this two algorithms is shown in the Fig. 4.

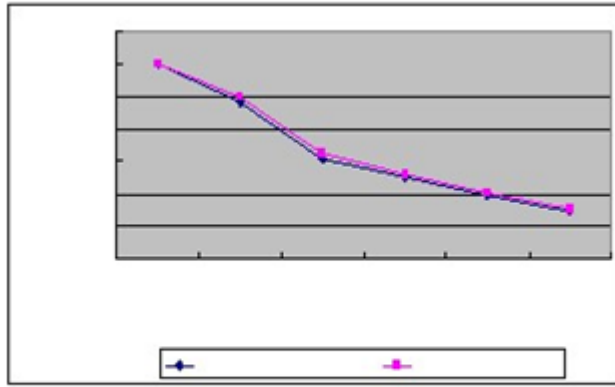


Figure 3. Concurrent writing performance<sup>+,J</sup>

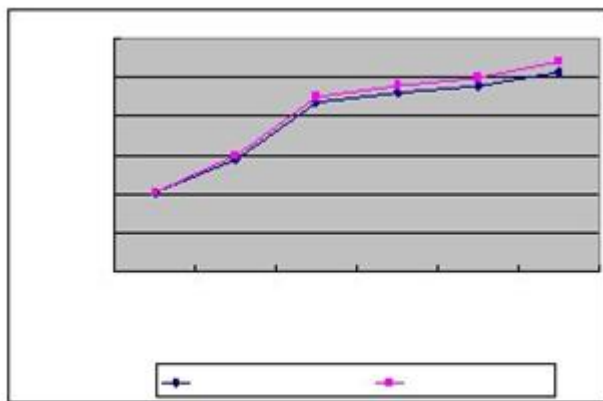


Figure 4. Concurrent writing performance<sup>+,J</sup>

This paper analyses the principle of the MongoDB Auto-Sharding. For the problem of uneven distribution of data among shards, we introduce an improved balancing algorithm-algorithm based on the frequency of data operation(FODM). A data balancing strategy based on FODO algorithm is proposed and its effectiveness is verified by experiments. The concurrent writing and reading performance of the Auto-Sharding cluster is significantly improved. And more aspects of the FODO algorithm could be explored, such as determination of inc value.

[1] Peter Mell, Tim Grance. "The NIST Definition of Cloud Computing".

National Institute of Science and Technology. Retrieved 24 July 2011.

[2] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, et al.

Dynamo:Amazon's Highly Available Key-value Store. SOSP'07.

Stevenson, washington, USA:2007.

[3] Fay Chang, Jeffery Dean, Sanjay Ghemawat, et al. Bigtable: A

Distributed Storage System for Structured Data. 7th Symposium on

Operating System Design and Implementation. Seattle, WA, USA: 2006.

[4] 10gen. MongoDB. <http://www.mongodb.org>, 2011-07-15.

[5] MongoDB. features.<http://www.mongodb.org/>.

[6] Kristina Chodorow, Michael Dirolf. “MongoDB: The Definitive Guide”.

O Reilly Media, September 2010. p135

[7] Kristina Chodorow. “scaling MongoDB”. O Reilly Media, January 2011. p13.

译文:

# 对云环境中改进 MongoDB 自动分片技术的研究

## 摘要

随着网际网路 2.0 技术的快速发展,大规模分布式服务的需求,给存储在云计算带来的巨大的挑战,动摇了传统的的关系型数据库。NoSQL 数据库从 RDBMS 的桎梏中解放出来成为人们关注的焦点。在这篇文章中,首先本着自动的原则和实施机制,提出了 MongoDB 数据库分片技术,而后为了解决分布不均的问题提出一个在数据操作的频率的基础上的改进的数据自动分片算法。改进的均衡策略能有效平衡之间的数据碎片,提高集群的并发读写性能。

关键词 NoSQL, MongoDB, 自动分片, 平衡策略

## 一、绪论

近年来,随着互联网数据量的快速增长和互联网的 Web 2.0 技术的发展,如何有效地存储,处理和提取的大量数据,成为一个迫切需要解决的问题,云计算在这种情境下应运而生。云计算通过网络(通常是因特网)[1]以计算而不是一个产品的形式作为服务的交付,从而共享资源,软件和信息提供给计算机和其他作为计量的服务设备。许多大学,供应商和政府组织各地投资于发展云计算的项目,例如:亚马逊推出简单存储服务(S3)和弹性计算云(EC2),谷歌推动的 BigTable 和 MapReduce, GFS, 已经全部成功地应用在生产环境中。分布式文件

系统可以组织大量的数据在云端,并且也能够有效地读取云数据,但我们为更好的结构化数据的管理,仍然需要专门的数据管理工具。

云数据管理是一个随云计算的发展产生的新的数据管理概念,它能够在云中的海量数据集有效地管理大型数据集,这使得快速找到特定的数据,以下常见的云数据管理的特点:

- (1) 高并发读写性能,
- (2) 有效地存储和访问大量的数据
- (3) 高可扩展性和高可用性要求

面对这些需求,传统的关系数据管理系统(RDBMS)遇到了难以逾越的障碍。因此, NoSQL 数据库系统使用量大幅上涨。主要的互联网公司,如谷歌,亚马逊, Twitter 和 Facebook 上有显著不同的挑战,在处理数据的 RDBMS 解决方案无法应对。这些公司意识到,性能和实时性比一致性更重要,而传统的关系数据库花费高额的处理时间来实现一致性。因此, NoSQL 数据库通常高度优化检索和追加操作,而且往往以小功能超出记录存储。减少运行时的灵活性相比, RDBMS 系统的显著收益补偿可扩展性和性能。通常情况下, NoSQL 数据库根据他们的数据存储和下降的方式分类根据类别,如键-值存储(如迪纳摩[2]), BigTableimplementations[3] and documentstore 数据库(例如 MongoDB[4])。然而,由于未成熟的技术云数据 MANGEMENT,在实际生产环境中还是有很多需要解决的问题。

本文讨论 MongoDB 数据库的设计原则和,实现机制专注于自动分片的原则。自动分片的目标就是分裂整个机器的数据和实现重新平衡,使得能够存储更多的数据和处理而无需大或强大的机器。但平衡算法没有那么智能,数据是不是均匀均衡地分布在服务器之间。为了解决这个问题,提出了一个改善 FODO(即频率的数据操作)算法。FODO 算法是在数据操作的频率的基础上的,考虑服务器负载实现的。基于 FODO 算法的数据均衡策略能够有效地平衡服务器之间的数据,提高集群的并发读写性能。

## 二、 MongoDB 中的自动分片机制

A 对 MongoDB 的简单介绍:

MongoDB 的（从“堆积如山”一词而来）是一个开放源码，以 C++ 编程语言编写的面向文档的 NoSQL 数据库系统。管理集合 BSON 文档。MongoDB 的开发始于 2007 年 10 月通过 10gen 的介绍，MongoDB 的特点[5]：面向文档存储，JSON 风格的文档与动态架构提供简单和完全索引支持，复制和高可用性，自动分片，水平缩放而不影响功能，Map / Reduce 的，灵活的聚合和数据加工；GridFS，无需复杂的堆栈存储任意大小的文件。

MongoDB 中文档是核心的概念，这是 MongoDB 的数据的基本单位，大致上，相当于在 RDBMS 中的一列。同样的，一个集合可以认为是相当于无架构的一个表。一个单一的 MongoDB 的实例可以承载多个独立的数据库，每一个都可以有自己的集合和权限。

## B 自动分片机制

分片是指分割数据的过程中和存储的数据的不同部分在不同的机器上。由于分裂数据跨机器，能够存储更多的数据和处理更多的负载，而无需大型或强大的机器[6]。MongoDB 支持自动分片，从而消除了一些手工分片，集群拆分数据和自动平衡。MongoDB 的分片提供：（1）在负载和数据更改自动平衡分配（2）方便地添加新的机器无需停机（3）无单点故障（4）自动故障转移。

MongoDB 的分片背后的基本概念就是要打破成较小的块的集合。这些块可以是分布的碎片，使每个分片负责总的数据集的子集。要集合，MongoDB 的分区指定碎片关键的图案命名一个或多个 fields 的定义后，我们的关键数据分发。因为碎片键块可以被描述为一个三元收集，minkey 的和 maxkey。块增长的最大尺寸通常是 200MB，一旦一大块已经达到近似的大小，块分裂成两个新的块。自动分片的体系结构如图 1 所示：

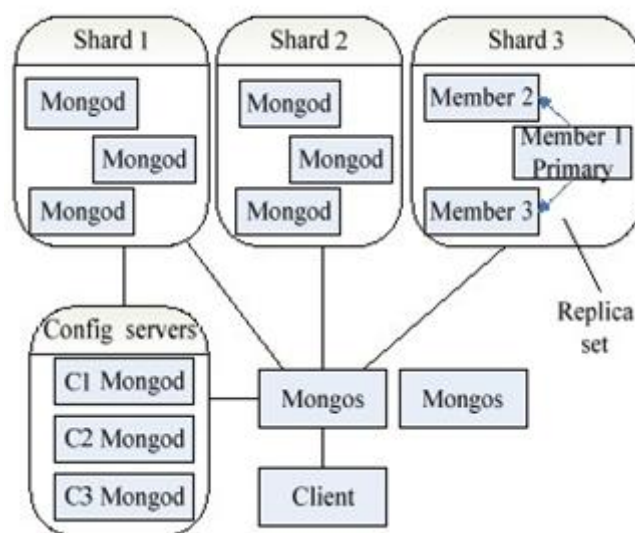


图. 1

MongoDB 的分片集群由两个或两个以上的分片，一个或多个配置服务器，以及任意数目的路由组成。每一个组成部分说明如下。

**shards:** 一个 shard 为一组 mongod，通常一组为两台，主从或互为主从，这一组 mongod 中的数据是相同的。数据分割按有序分割方式，每个分片上的数据为某一范围的数据块，故可支持指定分片的范围查询，这同 google 的 BigTable 类似。数据块有指定的最大容量，一旦某个数据块的容量增长到最大容量时，这个数据块会切分成为两块；当分片的数据过多时，数据块将被迁移到系统的其他分片中。另外，新的分片加入时，数据块也会迁移。

**config server:** 存储集群的信息，包括分片和块数据信息。主要存储块数据信息，每个 config server 上都有一份所有块数据信息的拷贝，以保证每台 config server 上的数据的一致性。

**mongos:** 可以有多个，相当于一个控制中心，负责路由和协调操作，使得集群像一个整体

的系统。mongos 可以运行在任何一台服务器上，有些选择放在 shards 服务器上，也有放在 client 服务器上的。mongos 启动时需要从 config servers 上获取基本信息，然后接受 client 端的请求，路由到 shards 服务器上，然后整理返回的结果发回给 client 服务器。

### C 自动分片的负载均衡

MongoDB 使用平衡器来保持所有的块均匀地分布在集群服务器。传送单位是一大块，平衡器等待不均匀块计数的阈值发生。在该字段中，具有 8 组块之间的差异，至少装的碎片表明是一个很好的启发。一旦达到阈值时，平衡器将重新分配块，直到差异块 between 的任意两个碎片下降到 2 个组块。

为了减少传输的数据量，每个碎片包含多个范围。当一个新的碎片加入集群或一些碎片含有太多的数据到达阈值时，均衡器将压缩块的顶部数据量最多的碎片，并将这些块的最低集合众多的碎片，使数据均匀地分布在集群移动最低限度下。

平衡器的目的不仅是为了均匀地保持数据分布而且也可以最大限度地减少传输的数据量。平衡器的算法是极端的智能。它的动作基于组块的整体尺寸的碎片[7]。迁移块还仅仅是位于顶部的每个碎片，但数据的操作还没有考虑到。碎片之间的数据传输可能不经常被使用，这将使得系统的负荷，不能达到有效的平衡。在 MongoDB 中提高自动分片的均衡策略是十分必要的。

## 三、基于数据操作频率的算法

### A FODO 算法的基本思想

为了解决自动分配数据的分布不均的问题，提出了一种基于数据操作的频率的改进算法（FODO 算法）

在 MongoDB 中，转让的单位是一大块。有 n 个大块碎片中，表示为第 i 块  $C_i$  和其频率运行数据值  $F\_DO_i$ 。在 MongoDB 数据库中，主营业务数据的插入，查找，更新和删除。由于集群中的数据很少会频繁删除，则该操作影响系统的性能主要集中前三操作。我们使用二，网络连接和  $U_i$  的数字代表这三种操作一大块。因此，在归一化的值  $F\_DO_i$  形式为：

$$F\_DO_i = \frac{I_i^{k,j}}{\sum_{i=1}^{n^{k,j}} I_i^{k,j}} + \frac{F_i^{k,j}}{\sum_{i=1}^{n^{k,j}} F_i^{k,j}} + \frac{U_i^{k,j}}{\sum_{i=1}^{n^{k,j}} U_i^{k,j}}$$

然而，插入，查找和更新操作上的集群中进行负载，并不应该被视为的产生不同的影响而一视同仁。在自动分片环境，这是没有必要的物理连接时，从数据库中查询记录因为兑现已经保持新鲜的数据。此外，每个碎片将包括的副本集生产形势。次要节点副本集可以读取数据写入从节点，因此它可以在一定程度上减轻主节点的查询负载。反之，插入和更新操作每次当他们发生了操作直接导致数据库，这占据了整个的集群工作的更大的一部分负载。特别是，插入数据，将导致在每一个片的数据的数量是不同的。一旦的阈值是达到平衡器平衡数据自动分片机制，这将极大地消耗系统资源。所以 FODO 算法在插入在方程中的一部分之前添加参数 INC（始终为 > 1），被称为插入系数，把更多的权重赋予插入操作。这里是 FODO 的最后的值的定义：

$$F\_DO_i = inc \cdot \frac{I_i^{k,j}}{\sum_{i=1}^{n^{k,j}} I_i^{k,j}} + \frac{F_i^{k,j}}{\sum_{i=1}^{n^{k,j}} F_i^{k,j}} + \frac{U_i^{k,j}}{\sum_{i=1}^{n^{k,j}} U_i^{k,j}}$$

每个分片的  $F\_DO$  值是所有它的副本集的  $F\_DO$  值得和：

$$s(F\_DO) = \sum_{i=1}^{n^{k,j}} c_i(F\_DO_i)^{k,j}$$



我们需要修改记录块的数据结构信息，添加 I, F, U 和 F\_DOi 的四个变量来存储插入，查找，更新和 F\_DOi 值。每一个操作的数据必须记录在相应的四个变量中。

#### B FODO 算法的负载均衡

F\_DOi 值表示每个块的数据操作的频率。如果一大块 F\_DOi 值高，这意味着数据在经常使用这个块。每个碎片数据的阈值再分配仍然是块的数目的差异，但迁移块是根据它的选择 F\_DOi 值。有取舍之间的总体规模碎片和操作在此均衡策略的数据的频率。该数据平衡的过程中有三个步骤。

- 1、数据迁移的门槛：计算数字在每个碎片块。如果该差值大于 8，则开始，以平衡数据，直到差值小于 2。从数据最多的碎片称为块将被迁移从碎片最少数据最多的碎片，叫做分片。
- 2、选择迁移大块：计算差异的 F\_DO 值之间从碎片和碎片。如果  $F(F\_DO) > T(F\_DO)$ ，然后从碎片的 F\_DOi 值选择最大块。否则，从碎片块选择最小 F\_DOi 的值。
- 3、迁移数据块：迁移在步骤 2 中所选择的组块到碎片，并重新计算每个块的 F\_DOi 值在从碎片和到 shard.Repeat 的步骤 1。

此外，我们应该确定业务的价值和权重。若业务是插入操作，所以把更多的重量，该值应该大于 1。业务的具体价值依赖于服务器相关的参数和系统负载，我们可以使用在设定初始值，然后根据实际业务需求调整。

#### 四、性能提升

MongoDB 的自动测试环境的基础上分片群集，它包含 10 个虚拟机，其中每一个具有相同的硬件配置：8GB 的内存和 CPU 速度是 2.4G HZ。每个虚拟机都有一个 Linux RedHat 的操作系统，并通过 100Mbps 的连接局域网中。MongoDB 的版本是 1.8.2，自动分片集群有三个碎片，每一个都包含一个副本集。每个副本集包含三个节点：一个主节点和两个次要节点。我们应该运行的 rs.slaveOk() 为辅助节点的命令来查询数据。

我们在 MongoDB 中实现 FODO 的算法，比较这两个算法通过测试并发读取和写集群的性能。测试中使用的数据集是一个简单的线数据，包括四个字段如 int, long, 字符串。此外，增量值在 FODO 参数算法是 1.5。

为了看到 FODO 算法的效果，我们只添加两个碎片，插入 1000000 条记录，并使用随机生成的 ID 做查找和更新操作。然后，我们添加第三个碎片到集群。此操作的目的是确保群集具有在测试开始的值的数据 F\_DO 不为零。

首先，我们测试集群的并发写入性能。1000000 条记录插入到集群中，并保持记录的总量不同的并发数字下是相同的。这两个并发写入性能图中所示的算法。保持并发的数量和记录保持不变，测试并发读取性能的集群。该这两种算法的并发读取性能图所示。

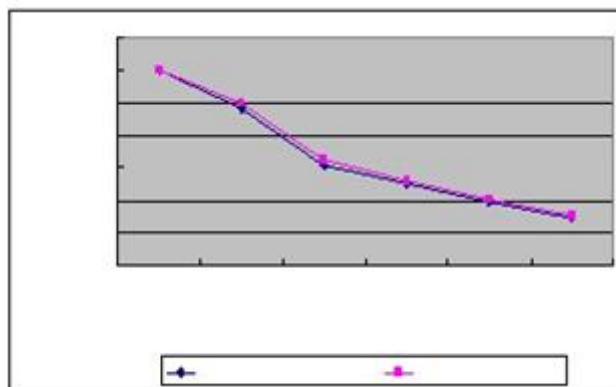


图. 2 写入性能 1

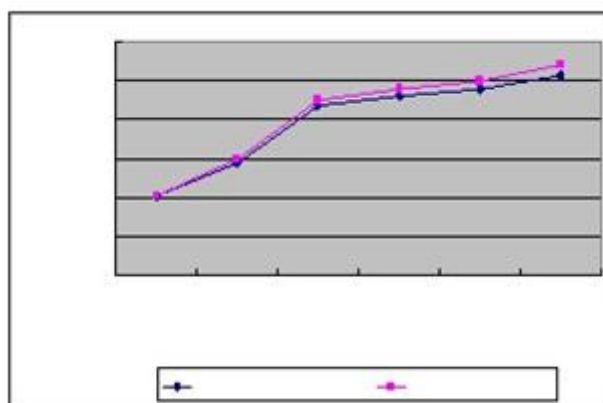


图. 3 写入性能 2

## 五、结论

本文分析 MongoDB 的自动分片原理。对于数据碎片分布不均的问题，我们引入了一种改进的均衡算法基于数据操作频率（FODM）。基于 FODO 的算法，提出数据的均衡策略是并通过实验验证了其有效性。该并发自动写入和读取性能分片群集显著改善。在多个方面的 FODO 算法值得探索和研究，如测定的 INC 价值。