d****



**本 科 毕 业 设 计**

院 系 软件学院

专 业 软件工程

题 目 Kikbug系统架构与实现

年 级 2012级 学 号 121250214

学生姓名 张玄

指导老师 陈振宇 职 称 副教授

论文提交日期

**南京大学本科生毕业论文（设计、作品）中文摘要**

题目： Kikbug系统架构与实现

软件学院 院系 软件工程 专业 2012 级本科生姓名： 张玄

指导教师（姓名、职称）： 陈振宇 副教授

摘要：

众包，是如今互联网行业中兴新的热门概念之一。众包，作为共享经济的一种表现形式，正在受到越来越多的创业者的关注。众包是现今互联网发展的一个重要的、主流的方向，因为众包模式解决了中小型公司对于人力资源以及资金的缺乏，同时也让社会闲散人员有了额外的收入来源，可以说是一个双赢的模式。

移动应用测试是一个工作量巨大的且十分耗时的工作，而且由于手机市场的复杂性，要做到对所有机型的覆盖测试几乎是不可能的。这对于企业，特别是中小型创业公司而言是一个巨大的负担。于是，移动应用测试的众包模式应运而生，Kikbug正是一个移动应用众包测试平台。

Kikbug系统已上线运行有一段时间，但是由于代码质量不高，已经不能满足继续演进的需求。故需要对Kikbug系统进行一次重做，目的是要能满足新的需求，并且可以应对之后不断增长的用户量、访问压力以及新的需求。

Kikbug新系统将从单机系统演进为分布式系统，本文将对Kikbug新系统架构设计的各个部分的方案进行分析评估，选择出最适合现阶段的技术方案，同时，也要制定系统未来的演进方案。

关键词：Kikbug；分布式系统架构；SOA；NoSQL

**南京大学本科生毕业论文（设计、作品）英文摘要**

THESIS: Architecture and implementation of Kikbug

DEPARTMENT: Software Institute

SPECIALIZATION: Software Engineering

UNDERGRADUATE: Xuan Zhang

MENTOR: Zhenyu Chen

ABSTRACT：

Crowdsourcing is one of the trending new concepts of the Internet today. Crowdsourcing, as a form of sharing economy, is attracting more and more entrepreneurs’ attention. Crowdsourcing is an important and mainstream development of today’s Internet industry. Since crowdsourcing not only relieves the heavy burden of human resource and funding of many startup companies, but also enables individuals to earn extra money, it seems a win-win strategy.

Testing of mobile applications is a time-consuming work, and due to the complexity of the mobile market, it is impossible to test against all mobile models. This is a huge burden to many companies, especially the startup ones. And then comes the crowdsourcing mobile application tests. Kikbug is one of the platforms that promote crowdsourcing mobile application tests.

Kikbug has been in service for a period of time. However, because of the low quality codes, it is unable to fulfill new requirements. So we must re-design and rebuild the whole system in order to meet new requirements and be able to sustain under growing user count, access count, as well as new requirements.

Kikbug will evolve from a single-machine system into a distributed system. This paper will discuss and assess various plans of the new Kikbug system, and select a most proper plan under current circumstance. What’s more, a plan of evolvement in the future will be settled.

KEY WORDS: Kikbug, distributed system architecture, SOA, NoSQL

**目录**

[第一章 引言 1](#_Toc449982527)

[1.1 项目背景 1](#_Toc449982528)

[1.2 项目可行性分析 2](#_Toc449982529)

[1.3 论文的主要内容和组织结构 2](#_Toc449982530)

[第二章 架构设计与技术选型 3](#_Toc449982531)

[2.1 需求简述 3](#_Toc449982532)

[2.2 整体架构方案 3](#_Toc449982533)

[2.3 可行性分析以及技术选型 4](#_Toc449982534)

[第三章 数据层的设计 9](#_Toc449982535)

[3.1 RDBMS vs NoSQL 9](#_Toc449982536)

[3.2 读写分离 14](#_Toc449982537)

[3.3 分库分表 16](#_Toc449982538)

[3.4 缓存系统设计方案 18](#_Toc449982539)

[3.5 Kikbug数据层实现方案总结 21](#_Toc449982540)

[第四章 业务层的设计 22](#_Toc449982541)

[4.1 SOA架构 22](#_Toc449982542)

[4.2 RPC & RESTful 28](#_Toc449982543)

[4.3 微服务架构 29](#_Toc449982544)

[第五章 展现层的设计 32](#_Toc449982545)

[5.1 前后端分离 32](#_Toc449982546)

[5.2 MVC vs MVVM 33](#_Toc449982547)

[5.3 React vs Angular 35](#_Toc449982548)

[第六章 部署与运维的设计 36](#_Toc449982549)

[6.1 自动化构建工具 36](#_Toc449982550)

[6.2 虚拟化 41](#_Toc449982551)

[6.3 集群监控 46](#_Toc449982552)

[6.4 Web端部署 47](#_Toc449982553)

[6.5 服务器部署 47](#_Toc449982554)

[第七章 项目总结 48](#_Toc449982555)

[参考文献 49](#_Toc449982556)

[致谢 51](#_Toc449982557)

第一章 引言

1.1 项目背景

众包[[1]](#footnote-1)，是如今互联网行业中兴新的热门概念之一。众包，作为共享经济[[2]](#footnote-2)的一种表现形式，正在受到越来越多的创业者的关注。较知名且成功的众包模式有众包物流（达达配送[[3]](#footnote-3)、饿了么蜂鸟配送、京东到家[[4]](#footnote-4)）、众包设计（猪八戒[[5]](#footnote-5)）、众包任务（阿里众包[[6]](#footnote-6)、微差事[[7]](#footnote-7)）等。无疑，众包是现今互联网发展的一个重要的、主流的方向，因为众包模式解决了中小型公司对于人力资源以及资金的缺乏，同时也让社会闲散人员有了额外的收入来源，可以说是一个双赢的模式。

移动应用测试是一个工作量巨大的且十分耗时的工作，而且由于手机市场的复杂性，要做到对所有机型的覆盖测试几乎是不可能的。这对于企业，特别是中小型创业公司而言是一个巨大的负担。于是，移动应用测试的众包模式应运而生，旨在解决这一令众多企业头疼的问题。

众包移动应用测试的优势：

* 测试人员不需要具备专业知识，也不需要专业设备，仅需要拥有且会使用智能手机/平板。
* 成本低，按需利用社会人员，可随时、任意调整，按成果发放劳资。
* 测试机型覆盖广，降低企业购置测试设备的负担，且能够在一定程度上反映出真实用户的手机机型使用情况。
* 测试人员与应用的最终用户类似，可以检测出一些可用性方面的问题。

Kikbug移动应用测试众包平台在此基础上进一步整合了高校学生的资源为企业提供更加优质的测试人力资源，也为高校师生提供了一个优秀的教育学习平台。

1.2 项目可行性分析

国内有许多移动应用云测试平台，例如Testin、阿里云测移动质量中心、百度云移动App测试、腾讯bita云测试平台等。但是实行众包测试的平不多见。

Kikbug移动应用测试众包平台已经运行了一段时间，已经为南京大学软件学院的同学以及多个企业（包括阿里云测）提供了服务。

然而，由于Kikbug原有系统存在许多设计缺陷，代码难以维护，无法进行更多的新需求开发。为了将Kikbug推广至全国各个高校，需要对Kikbug系统进行一次彻底地重新设计与实现，以适应新的需求，提供更好的服务。

1.3 论文的主要内容和组织结构

本人在Kikbug项目中担任架构师，负责Kikbug新架构的设计。同时，本人负责Kikbug系统服务器的实现、参与了Kikbug网站的实现，同时也负责Kikbug系统的运维。所以本文主要内容在于阐明Kikbug系统新架构设计时的技术方案的对比分析、最终的技术方案选择以及原因、实现的部分技术细节，同时也会对Kikbug系统的部署以及运维方案进行一定的分析描述。

第一章 引言，阐述项目背景以及本文目的与结构。

第二章 系统需求简介以及系统基础架构的设计以及设计的过程、技术方案的舍弃。

第三章 Kikbug数据层的设计方案对比以及选择，主要涉及对传统的RDBMS与NoSQL的比较、分析读写分离与分库分表的技术方案设计以及最终的数据层的设计。

第四章 Kikbug业务层的设计方案对比以及选择，主要涉及SOA架构的应用分析、RESTful与传统RPC的对比以及服务治理、微服务架构的理解。

第五章 Kikbug展现层的设计方案对比以及选择，主要涉及前后端分离的理解以及传统MVC框架与现代MVVM框架的对比、React与Angular的选择。

第五章 Kikbug部署与运维的设计方案对比以及选择，主要涉及虚拟化的应用、日志分析监控系统的设计以及Kikbug系统的部署方案。

第六章 项目总结

第二章 架构设计与技术选型



2.1 需求简述

Kikbug原有系统是一个平台，用户包括学校师生、企业用户以及系统管理员。用户最主要的需求包括发布/接受/查看测试任务、上传/查看/编辑测试任务报告等。

Kikbug新系统最重要的需求是添加了新的移动应用客户端，即Android客户端以及iOS客户端。同时Kikbug的网站也会继续运行，保持与阿里云测的合作的同时还会与慕测以及自动化测试平台进行合作。

Kikbug上线后会面向多所高校提供服务，会有近2000名师生使用该系统，所以Kikbug系统必须能够承受这样的压力。

需求总结如下：

* 类似电商网站的平台系统
* 支持Web、Android以及iOS客户端
* 支持与外部系统的交互
* 至少能承受2000人的日常访问压力

2.2 整体架构方案

Kikbug原有系统采用JSP技术实现Web端，同时开发了一套API，以HTTP的方式向客户端和外部系统开放比较简单的接口。

Kikbug新系统的需求是如今互联网的最常见的需求，即三种客户端并存（Web、Android、iOS），且开发周期短。对于这样的需求，开发一个统一的API是关键。而Kikbug原有系统使用的传统网站技术（如Java JSP、.NET Asp等），由于开发周期长，没有对于客户端API的支持，已经无法满足这种需求，故我们需要使用的是现今最先进的互联网开发技术。

React Native是非常热门的一种开发一次即可生成Web、Android、iOS客户端的技术。这种技术被大量创业公司使用，以最少的人力和时间成本开发质量较高的产品。但是由于Kikbug客户端需要分别在Android以及iOS平台上深度定制，以实现应用截图、日志获取等功能，故无法使用这一技术进行开发。

退而求其次，为了提高开发效率，我们必须实现的思想是“前后端分离”，即把Web端视为一个和Android与iOS等同的一个客户端，通过API与服务器进行交互。这样做使得服务器端不再需要负责渲染Web页面，大大简化了服务器端的职责，统一了API的使用，且服务器端更加符合SRP这一设计原则。

Kikbug原有系统为单服务器设计，所有业务逻辑以及一切相关服务（数据库、静态文件资源等）完全部署在同一台服务器上。这是一个极其危险且不负责任的设计，一旦服务器发生任何故障（网络波动、进程意外终止、内存耗尽、磁盘写满等），都会导致服务不可用的情况出现。这对于一个互联网平台来说是最致命的问题。

新的架构必须能够解决这个问题，采用分布式的设计方案，保证服务的高可用性。同时由于在服务上线之后会在全国范围内进行推广，服务器将要面对不断增长的且不可预知的访问压力，需要保证服务的高可扩展性。对于服务的性能没有特殊要求。

架构方案总结：

* 大致符合传统电商网站的系统架构
* 使用前后端分离的网站架构，提供且仅提供一套API
* 使用分布式高可用、高可扩系统架构

2.3 可行性分析以及技术选型

原则：

Kikbug技术选型的原则并不在于使用最先进最复杂的技术，而是综合考虑现有资源（开发时间、开发人力等），选择最合适而不是最优的方案。

由于Kikbug系统的开发时间短、开发人员少、且现阶段的部署资源也少、用户量不大，所以系统的技术选型偏向于成熟可靠的技术风格。但是由于Kikbug新系统上线后会进行推广，故需要考虑方案的可扩展性，需要有一套完整的系统演进方案。

1. 客户端与服务器端的通讯协议

采用前后端分离的思想，Web端与服务器端通过HTTP协议通信是最自然、最有效率的方案，因为浏览器本身支持且仅支持HTTP/HTTPS协议。所以必须提供HTTP协议的API以支持Web端，即Web端与服务器交互只可使用HTTP协议，没有选择余地。现在讨论移动客户端与服务器的交互协议。

移动客户端与服务器通信协议可选方案有：

* RESTful[[8]](#footnote-8)
* Thrift[[9]](#footnote-9)
* gRPC[[10]](#footnote-10)

RESTful是建立在HTTP协议上的一种API规范，而Thrift与gRPC都是一种重量级的RPC框架。

|  |  |  |  |
| --- | --- | --- | --- |
|  | RESTful | Thrift | gRPC |
| 性能 | 一般 | 高 | 高 |
| 流量 | 一般 | 小 | 小 |
| 安全性 | 低 | 高 | 高 |
| 开发成本 | 低 | 高 | 高 |

表格 1 客户端与服务器端通信协议对比

Thrift和gRPC通过自有的序列化系统对传输的数据进行序列化与反序列化，传输协议是专门为RPC而设计的，故性能很高、安全性很高且流量占用很小（流量对于移动应用也很关键）。但是也正是由于使用了序列化与反序列化系统，所有的交互数据必须用IDL[[11]](#footnote-11)进行定义，且客户端必须使用对应的第三方类库进行开发，开发成本很大。而RESTful使用的是HTTP协议，HTTP协议使用文本通讯，且在没有HTTPS的支持下，通讯使用明文，安全性不高且性能一般，流量开销也一般。但是由于各个平台都原生支持HTTP，且Kikbug的Web端必须使用HTTP协议来与服务器通讯，故使用RESTful进行开发可以大大提高开发效率。

RESTful目前应用广泛，是一种非常成熟且优雅的方案。使用该方案不仅在客户端不需要额外依赖第三方类库，服务器端也有很多成熟的框架可以选择。Kikbug系统对于客户端流量的消耗主要发生在文件上传以，接口通讯时产生的流量可忽略不计。Kikbug尚不涉及任何金钱交易，故安全性并不是一个重要的关注点。所以选择RESTful作为客户端与服务器端的通讯协议是最合适的方案。

1. 前后端分离

前后端分离已经不是一个新奇的概念了，随着React、Angular、Vue等框架的崛起，Web前端的变化翻天覆地、日新月异。目前甚至连jQuery这样的老牌统治级的框架也已经被废弃，前后端分离的实现已经非常成熟，不存在任何技术难点。前后端分离的优点在上一节已经讨论，故Kikbug可以放心大胆地使用该方案。

1. 开发语言

服务器开发语言在近年也有了井喷式的发展。

|  |  |
| --- | --- |
| 语言 | 简介 |
| Java | 老牌语言，可靠性高  极丰富的类库  不够敏捷，缺乏现代语言特性 |
| PHP | “世界上最好的语言” |
| Python | 擅于数据处理  语法简单  拥有Django、Flask等成熟的MVC框架  多线程支持不佳 |
| Ruby | 硅谷创业必备  一切都是对象  开发人员友好型语言  敏捷型开发语言 |
| Node.js | 支持函数式编程  原生不支持阻塞型IO，单机可支持百万级高并发  拥有Express、Koa等成熟的MVC框架  不擅于处理CPU密集型应用 |
| Golang | Google出品  高性能编译型语言  原生支持协程（Goroutine） |

表格 2 Web开发语言对比

鉴于开发人员的知识储备，综合考虑部署难度以及后期维护难度，服务器将继续使用Java作为开发语言。

1. 分布式架构

由于采用Java语言开发且使用HTTP协议作为通讯协议，故有Tomcat、Jetty、Spring Boot等多种开源Java Web应用容器可以选择。

为了消除单点，达到高可用的目标，服务器必须以集群的方式部署。故需要引入一个LB[[12]](#footnote-12)组件，可选组件有Nginx、HAProxy等。

由于使用了集群，不可避免的问题是如何保持客户端会话的状态。现讨论如下两种方案：

1. LB转发请求时根据用户ID转发到指定服务器
2. 服务器设计为无状态，使用集中式的缓存服务保存状态

若使用方案1，不可避免的是要对LB进行一定的改造，这会引入一些BUG，但是该方案较方案二优点在于不需要引入一个新的外部组件，同时，它的效率也显然会比方案二要高。

若使用方案二，则需要引入Redis或者Memcached之类的缓存系统。但是该方案的优点在于服务器为无状态，可以任意启停，很好地满足了高可扩的需求。而方案一则无法做到这一点，且每次添加/删除机器后都有可能导致客户端会话状态的丢失。因此，方案二为最合适的方案。

最终，Kikbug基础架构的设计如下图：



图 1 Kikbug分布式基础架构设计

第三章 数据层的设计

本章主要对Kikbug系统的数据层设计方案进行讨论研究。由于如今NoSQL社区的蓬勃发展，传统的关系型数据库（RDBMS）虽然在互联网项目中的地位仍不可撼动，但是NoSQL也占据了半壁江山。本章会对近年社区活跃的NoSQL数据库以及传统的关系型数据库进行讨论分析，以找出最适合Kikbug系统的数据层解决方案的数据库。



3.1 RDBMS vs NoSQL

1. RDBMS的特性

RDBMS，即传统的关系型数据库。典型的代表有MySQL、Oracle、SQL Server以及PostgreSQL等。

关系型数据库一般满足事务型数据库要求ACID，即：

**Atomicity（原子性）：**事务结束后要么成功，要么失败，不会停留在中间状态。即可以把事务看作数据库变更的最小单位。如果事务执行过程中出现问题，则需要回滚至事务开始前的状态。

**Consistency（一致性）：**事务的执行必须保持数据库状态的合法性。即必须保证类型约束、外键约束、触发器等所有状态处于合法状态。

**Isolation（隔离性）：**事务之间不会相互干扰。

**Durability（持久性）：**一旦事务被提交，那么该事务对数据的变更是永久的。

除此之外，RDBMS的表设计还有如下几大设计范式：

**1NF：**数据库表的每一列都是不可分割的基本数据项，同一列中不能有多个值。

**2NF：**数据表的每一行数据都必须能被唯一标识。

**3NF：**数据表的列的值不依赖于除主键列之外的列。

**BCNF：**对于数据表中的每个函数依赖X→Y，若Y不属于X，则X必含有候选码。

这些原则与范式强调的无非一个问题，那就是数据必须在任何时刻都是完整的、可靠的、一致的。这种模型是很直观的、符合人们对于数据预期的一种模型，也是对于安全性来说最佳的数据模型。这种模型适用的场景是涉及交易的场景，因为这种场景对于数据一致性的要求极高，谁都不希望钱的数量出现差错。所以银行、金融企业、电子商务企业大都会在产品中大量应用关系型数据库，用来进行交易数据的实时处理。比如各大银行、金融系统使用IBM的DB2作为数据存储解决方案，又比如中国最大电商企业阿里巴巴集团拥有自己的MySQL代码分支[[13]](#footnote-13)。同时，由于关系型数据库十分成熟，社区活跃，文档齐全，拥有许多经验丰富的工程师，许多初创公司也会毫不犹豫地选择像MySQL这一类的开源关系型数据库作为数据层的解决方案。

1. 现代互联网需求的特点

然而，现代互联网行业并不是只有电商的行业。可以说今年的互联网发展是围绕着O2O、大数据等概念的。互联网产品大都是面向不具有专业知识的用户的，且同行竞争激烈，这对于现代互联网产品的用户体验要求很高。而影响用户体验的一大因素就是系统的响应时间。现代互联网产品必须能够承受住高并发量的考验，然而，处理高并发并不是传统关系型数据库的长处，由于关系型数据库存在锁（如MySQL存储引擎MyISAM的表锁、InnoDB的行锁），在高并发量下，锁竞争会异常激烈，导致响应时间下降。

同时，一个成功的互联网产品必定会遇到的问题是数据量的爆发式增长。然而，处理大量数据也不是关系型数据库的长处。在数据量达到百万级之后，对于表的查询如果在没有索引的情况下会十分缓慢。根据范式的要求，在查询时或多或少会使用到JOIN操作，然后这个操作对于百万行级别的表来说是十分可怕的，不仅内存会被大量消耗，响应时间也异常缓慢。

现代互联网产品会遇到的另一个问题是产品的需求会频繁地变更。这就导致了在表结构会在之后被改变。传统的关系型数据库在改变表结构时的代价很大，因为在改变时需要锁表，必然会导致大量请求被阻塞。当表的数据量大时，阻塞时间会更长，严重的会导致系统拒绝响应。所以在设计表的初期，很多工程师会使用一个字段用来存储一段JSON，来应付之后的需求变更。这样做虽然可以解决问题，但是只能算是一个hack，且在JSON中的字段无法建立索引。

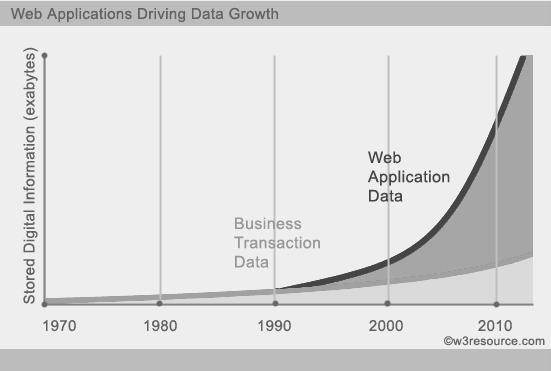


图 2 Web应用数据量

1. NoSQL的特性

NoSQL（Not only SQL），是为了满足现代互联网产品的需求而诞生的一类数据库。称它为一类数据库，是因为所有不是关系型数据库的数据库都可以被称为NoSQL数据库。主要分为以下几类：

* 键值存储（Key-value store）
* 文档存储（Document store）
* 列存储（Wide column store）
* 图（Graph DBMS）
* 全文搜索引擎（Search engine）

主要代表有：

|  |  |  |
| --- | --- | --- |
| **名称** | **类型** | **特点** |
| MongoDB | Document store | 与JSON完美结合 |
| Cassandra | Wide column store | 分布式高可用数据库 |
| Redis | Key-value store | 高性能内存数据库  主要应用于缓存  取代Memcached |
| Elasticsearch | Search engine | 主要应用于实时关键词搜索 |
| HBase | Wide column store | 基于HDFS  高写入速度 |
| Neo4j | Graph DBMS | 主要应用于社交类网站的图关系存储 |

表格 3 NoSQL简介

NoSQL系统主要应对的是大型分布式的场景，所以NoSQL系统满足分布式系统的CAP定理，即：

**Consistency（一致性）：**所有节点上的数据一致。

**Availability（可用性）：**每个请求都有响应，无论成功或是失败。

**Partition tolerance（分区容忍性）：**即分布式系统，且某个节点出现问题不会导致整个系统出现问题。

有趣的是CAP三点是无法被同时满足的，而大部分NoSQL系统会满足P特性，所以是使用具备A特性的NoSQL系统还是使用具备C特性的NoSQL系统是要根据实际项目的需求而决定的。

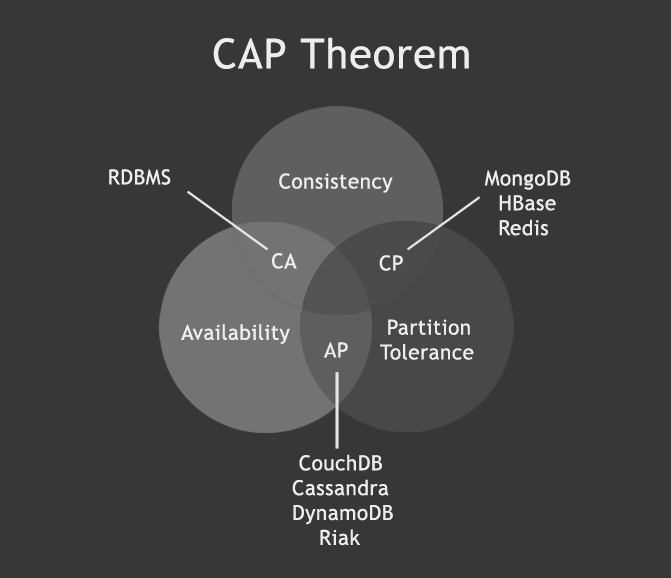


图 3 分布式系统的CAP理论

具备C特性的NoSQL系统，强调的是数据一致性，而它的可用性不会很强。如Redis是一个单线程内存数据库，所有的请求都在同一线程内序列化完成，并发的请求会进入队列等待处理。所以，显而易见，Redis具有很高的数据一致性，即C特性，但是因为使用了单线程模型，一旦某个请求发生阻塞，那么之后的所有请求都会阻塞相同的时间。这样就带来了可用性方面的问题。所以，具有CP特性的NoSQL系统，一般被用于作系统高速缓存。

具备A特性的NoSQL系统，强调的是系统可用性，而它的数据一致性不会很强。如Cassandra是一个分布式列存储数据库。它遵循BASE原则[[14]](#footnote-14)，即符合最终一致性模型。Cassandra在节点间使用Gossip[[15]](#footnote-15)协议进行数据传递，故在数据写入之后，并不是所有的节点都可以获取到写入的数据，即数据的一致性不强。所以，具有AP特性的NoSQL系统较多应用于社交网站（如Facebook，事实上Cassandra由Facebook开发）。

除了CAP的考虑，实际项目中也必须考虑到所使用NoSQL系统的社区活跃度、文档完整度、系统兼容性等额外的因素。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Rank** | | | **DBMS** | **Database Model** |
| **Apr 2016** | **Mar 2016** | **Apr 2015** |
| 1. | 1. | 1. | [Oracle](http://db-engines.com/en/system/Oracle) | [Relational DBMS](http://db-engines.com/en/article/RDBMS) |
| 2. | 2. | 2. | [MySQL](http://db-engines.com/en/system/MySQL) | [Relational DBMS](http://db-engines.com/en/article/RDBMS) |
| 3. | 3. | 3. | [Microsoft SQL Server](http://db-engines.com/en/system/Microsoft+SQL+Server) | [Relational DBMS](http://db-engines.com/en/article/RDBMS) |
| 4. | 4. | 4. | [MongoDB](http://db-engines.com/en/system/MongoDB) | [Document store](http://db-engines.com/en/article/Document+Stores) |
| 5. | 5. | 5. | [PostgreSQL](http://db-engines.com/en/system/PostgreSQL) | [Relational DBMS](http://db-engines.com/en/article/RDBMS) |
| 6. | 6. | 6. | [DB2](http://db-engines.com/en/system/DB2) | [Relational DBMS](http://db-engines.com/en/article/RDBMS) |
| 7. | 7. | 7. | [Microsoft Access](http://db-engines.com/en/system/Microsoft+Access) | [Relational DBMS](http://db-engines.com/en/article/RDBMS) |
| 8. | 8. | 8. | [Cassandra](http://db-engines.com/en/system/Cassandra) | [Wide column store](http://db-engines.com/en/article/Wide+Column+Stores) |
| 9. | 9. | 10. | [Redis](http://db-engines.com/en/system/Redis) | [Key-value store](http://db-engines.com/en/article/Key-value+Stores) |
| 10. | 10. | 9. | [SQLite](http://db-engines.com/en/system/SQLite) | [Relational DBMS](http://db-engines.com/en/article/RDBMS) |
| 11. | 11. | 14. | [Elasticsearch](http://db-engines.com/en/system/Elasticsearch) | [Search engine](http://db-engines.com/en/article/Search+Engines) |
| 12. | 12. | 11. | [SAP Adaptive Server](http://db-engines.com/en/system/SAP+Adaptive+Server) | [Relational DBMS](http://db-engines.com/en/article/RDBMS) |
| 13. | 13. | 13. | [Teradata](http://db-engines.com/en/system/Teradata) | [Relational DBMS](http://db-engines.com/en/article/RDBMS) |
| 14. | 14. | 12. | [Solr](http://db-engines.com/en/system/Solr) | [Search engine](http://db-engines.com/en/article/Search+Engines) |
| 15. | 15. | 15. | [HBase](http://db-engines.com/en/system/HBase) | [Wide column store](http://db-engines.com/en/article/Wide+Column+Stores) |
| 16. | 16. | 17. | [Hive](http://db-engines.com/en/system/Hive) | [Relational DBMS](http://db-engines.com/en/article/RDBMS) |
| 17. | 17. | 16. | [FileMaker](http://db-engines.com/en/system/FileMaker) | [Relational DBMS](http://db-engines.com/en/article/RDBMS) |
| 18. | 18. | 18. | [Splunk](http://db-engines.com/en/system/Splunk) | [Search engine](http://db-engines.com/en/article/Search+Engines) |
| 19. | 19. | 21. | [SAP HANA](http://db-engines.com/en/system/SAP+HANA) | [Relational DBMS](http://db-engines.com/en/article/RDBMS) |
| 20. | 20. | 22. | [Neo4j](http://db-engines.com/en/system/Neo4j) | [Graph DBMS](http://db-engines.com/en/article/Graph+DBMS) |

表格 4 2016年5月数据库排名[[16]](#footnote-16)

由上表可见，NoSQL与RDBMS表现相当。排名也从侧面反映出该数据库的社区支持情况。

1. 项目需求以及技术方案

Kikbug系统是一个平台，类似于电商网站，但是并没有电商所需要的交易功能，故Kikbug对于数据一致性的要求并不是很高。同时，由于Kikbug会服务于上千高校师生以及企业，系统可用性的要求较高。

但是由于系统的数据量并不大，预计数据量只会维持在GB级别，使用具备AP特性的NoSQL系统会大大增大部署难度以及运维成本，同时鉴于开发人员的知识储备，Kikbug系统仍选用开源RDBMS系统MySQL作为数据持久化存储解决方案。

由于Kikbug系统存在高峰期，在某个任务即将截止时可能会遇到大量的并发请求，故将引入NoSQL系统作为高速缓存，缓解MySQL的压力。鉴于社区活跃度以及系统兼容性，选用Redis作为Kikbug系统的缓存存储解决方案。

3.2 读写分离

在高并发量的情况下，如果没有任何保护措施，MySQL的压力会十分巨大。然而MySQL在Kikbug系统中属于一个单点，一旦出现宕机，会导致Kikbug系统的整体瘫痪，这是对于系统可用性的极大威胁。对于MySQL的频繁读取的请求是可以进行缓存的，这样做可以降低了MySQL的压力，但是却牺牲了一定的写入速度（要同时更新缓存）以及一致性（缓存更新存在延迟），所以必须使用在读多写少、一致性要求不是特别高的系统中。而Kikbug系统正符合这一特性。故在本节将讨论MySQL读写分离的方案，读写分离的目标是防止大量请求到达MySQL数据库。

1. MySQL主从读写分离

MySQL原生支持主从（Master-Slave）部署，数据通过Binlog的方式进行主从数据同步。使用主从部署的方式，可以做到读写分离，即使用从库进行数据查询（SELECT），而主库只负责数据更新（INSERT、UPDATE、DELETE）。数据写入主库后，主库会将自己的Binlog（主库必须开启Binlog记录功能）发送至所有从库，从库根据主库的Binlog复制相同的操作，生成相同的数据，故存在主从延迟的现象。

这种方案的优点是数据被复制了多份，存在不同的数据库中。所以，主从部署还支持当主库宕机时，从库切换为主库继续提供数据服务，提高系统的可用性。



图 4 MySQL主从同步读写分离方案

1. MySQL + Redis 读写分离

另一种解决方案则是利用Redis作为告诉系统缓存。API服务器在每次对MySQL进行查询前先在Redis中查询结果，若有结果则直接返回，若没有结果，则查询MySQL，然后将结果存储到Redis中。

由于Redis是一个高性能的内存数据库，所以这种方案可以支撑很高的读取并发量。然而，这种方案的问题在于对于Redis缓存的写入完全由API服务器负责，所以一旦数据发生更新，API服务器也要负责更新缓存。同时，由于API服务器存在多台，所以在写入缓存时存在一定的并发问题。但是由于写缓存是一个幂等[[17]](#footnote-17)的操作，所以多次写入只会造成部分的性能浪费。



图 5 MySQL + Redis读写分离方案

方案1是比较重量级的解决方案，比较成熟可靠，但是代价也比较大，因为需要部署多套MySQL系统。这不仅需要更多的服务器，也极大地增加了后期维护的成本，鉴于Kikbug系统维护人员的数量，该方案并不合适。

由于Kikbug系统中已经引入了Redis，故使用方案2的后期维护代价比较小。且方案2较灵活，可以在业务代码中缓存更多的中间结果而不一定是SQL查询的结果。同时Redis的查询性能也大大高于MySQL。唯一的缺点在于所有对MySQL的数据更新操作都必须删除对应的Redis缓存，否则会导致数据不一致问题。由于Redis原生支持Key TTL[[18]](#footnote-18)，且Kikbug系统对于数据一致性要求不高，故选择方案2作为读写分离方案。

3.3 分库分表

当数据量达到百万级时，对于数据表的一些查询会变得缓慢。例如范围查询可能会查询到大量的数据导致响应缓慢；数据插入可能会引起B树索引的分裂从而导致响应缓慢；JOIN操作对于两张百万级别的数据表来说性能也是相当差。所以我们需要降低数据库中的数据量，从而提升性能。

1. 分库

首先，分库是一个比较轻量级且易操作的解决方案。分库，也被称作对数据库的垂直分割。其主要目的是把数据库中的数据表按照业务分成若干个库，多个库之间相互不影响，从而把访问压力分散到了多个库中。

分库比较容易实现的原因在于根据业务逻辑分割数据库，由于被分割的数据表之间不存在关联，所以不会对业务产生较大影响。但是如果在业务中大量地使用了多个表的JOIN操作，在分库时可能会发现难以将多个不同的数据表拆开。所以在实现业务逻辑时需要注意除非必要情况，尽量不要使用JOIN操作，而使用反范式设计冗余存储，保持表的相对独立。

虽然分库比较容易实现，但是分库的效果往往不是那么突出。由于只是简单地将数据表拆开，单个数据表内的数据量并没有变化，所以只是减轻了数据库的压力，对于慢查询仍没有有效解决。

1. 分表

分表是比较重量级的解决方案，对于业务层会有较大影响，且比较难以实现，但是对于慢查询的效果是显著的。

分表的思想主要是将一个大数据表分割成若干个小数据表，故又被称为对于数据库的水平分割。并不是所有的数据表都可以被水平切割，因为数据表在业务上是内聚的，而数据表的行与行之间对于业务来说存在着较大的聚合关系。如当需要查询某一列的值在某个范围内的数据时，会查询出多行数据，这些行之间就对于业务存在着聚合关系。

分表一般按照如下几种方案进行：

* **直接根据主键分割** 根据行主键的值，将这行数据分割到对应的数据表中。一般根据 *主键值 mod 分表数量* 来找到实际数据表。这样的分割会出现上述的查询问题，即范围查询十分困难，需要在多个库中分别查询，之后再聚合成最终结果。这种分割方式比较适用于不需要进行范围查询的表，如用户表。缺点是增减分表数量时需要重新对所有数据进行分表。
* **按照日期分割** 这是一个现代互联网的经验结论，一般来说，当三个月前的数据几乎不会被访问。所以可以按照日期，根据实际的业务数据量以及业务的对于数据聚合的要求，将数据表按月份、星期、天等进行分割。这样做可以大大降低范围查询时需要多表查询的情况，但是仅适用于时间敏感型的数据表。
* **按照冷热数据分割** 在大多数情况下，数据库中会存在冷热数据。即部分数据的访问量大大高于其他数据，且热数据的数量往往远远低于冷数据。在这种情况下，如过所有的热数据都处于同一张表中，那么很有可能会导致性能问题。所以，需要将热数据尽可能地分割到不同的小数据表中，而冷数据可以大量保存在同一张数据表中。这种方案实现十分复杂，且在实际访问量发生之前，对于数据的冷热预判并不一定准确；数据过于分散会使数据的聚合分析更加难以进行。所以该方案只有在极端情况下才比较适合使用。

分表之后的数据需要有一个统一的数据访问层，实现数据的分表存储与读取逻辑，同时将数据存储与业务层进行解耦。

不论是分库还是分表，都需要部署多套MySQL实例。这样做不仅增加了成本也增大了维护难度。且在系统上线初期并不会有巨大到需要分割数据库的数据量，所以对于Kikbug系统来说，分库分表并不是一个在当前阶段必须实现的方案。但是随着后期Kikbug系统的推广，不能排除数据量爆发增长的可能，故在现阶段可以实现的是尽量不使用JOIN操作，即使用反范式，冗余存储字段。这样做不仅为以后实现分库铺平了道路，也不违反系统对于数据一致性的较低要求，更重要的是可以提升系统的响应时间。

3.4 缓存系统设计方案

缓存系统将使用Redis作为存储数据库，然而，面对大量数据，即使是Redis也有极限，且Redis单机部署造成了系统单点。需要将Redis以集群方式部署，那么我们就需要讨论Redis的集群设计方案。

Redis集群部署主要需要解决的问题是如何实现数据的分片（Sharding）存储。数据必须以某种策略存储在集群中的某一Redis实例上，在查询数据时必然也需要相同的策略找到对应的Redis实例查询数据。

现有如下几种主流方案：

1. 客户端分片

业务层在代码中按照统一的策略将数据（按照数据ID等）分片存储进某一Redis实例。Redis集群中的实例感知到不彼此的存在，相对独立。这样的策略优点在于实现简单，不需要部署任何额外系统，也不需要对Redis做任何配置，且可以对分片算法进行控制。缺点在于动态添加或删除Redis实例时，客户端分片的算法不一定支持，很有可能需要重启。即使客户端的算法实现了动态增减Redis实例的功能，要做到所有机器同时更新实例列表也是相当困难的。



图 6 Redis客户端分片

1. Twemproxy + Redis集群

Twemproxy 是由Twitter开发的一个快速的单线程代理程序，支持Memcached和Redis。Twemproxy的功能类似于Nginx，是一个处于API集群和Redis集群之间的系统。它的作用在于由它统一实现分片算法，Redis的客户端不需要知道究竟有多少Redis实例存在，也不需要关心如何对数据进行分片。另一个作用在于减少API服务器与Redis的连接数量。如采用方案1，那么每个API服务器都需要保持与所有Redis服务器的连接。而连接池中往往会保持有几百个Redis连接，所以如果API有100个实例，那么每一台Redis上的连接数可能会达到5000。由于Twemproxy实现了Redis的接口与通信协议，客户端是可以用Redis客户端与Twemproxy通信的，而不需要引入另一个第三方的客户端。

然而，Twemproxy也会成为系统的一个单点，故Twemproxy一般也以集群方式部署。这样就对运维提出了很大的挑战。



图 7 Twemproxy + Redis集群方案

1. Redis Cluster

Redis Cluster是Redis 3.0的一个新特性，即Redis原生支持的集群部署方案。Redis Cluster使用一致性哈希算法将数据分片到不同的节点上，故也支持动态地添加删除节点。Redis Cluster使用的一致性哈希算法为哈希槽，集群通过牺牲一定的数据一致性来提供更高的可用性，即使节点宕机也可以保证数据完整性。这点是以上两个方案不能做到的。

Redis Cluster无论是在服务器配置还是部署上都有一定的难度。且作为一个新特性，该功能的稳定性不得而知。不仅如此，Redis Cluster还需要使用新的通信协议，这意味着客户端不能单纯地使用现有的稳定的客户端，这进一步增加了不确定性。故不推荐在现阶段使用该方案。方案2是一个成熟可靠的方案，但是由于需要做大量的配置与部署，在现阶段也不推荐使用。而方案1由于部署简单，且大多Redis客户端原生支持分片，所以是现阶段使用的最佳方案。



图 Redis Cluster方案

3.5 Kikbug数据层实现方案总结

数据层解决方案主要需要应对不断增长的数据量，保证响应时间短，系统可用性高。在Kikbug系统上线之初，不用过于担心数据量过大问题，但是需要关心的是日后数据量增长的问题。数据层的解决方案不仅要在现阶段可以快速实现并部署，也要可以不断演进，达到项目后期可能面对的数据需求。

Kikbug数据层设计方案以及演进方案：

* 使用MySQL作为数据持久化存储数据库，现阶段使用单机，当访问压力上升之后使用MySQL主从读写分离。
* 使用Redis集群作为数据缓存存储数据库，现使用客户端的分片，之后可考虑使用Twemproxy进行集群部署。当Redis Cluster成熟之后可以考虑直接使用Redis Cluster。
* 业务层设计使用反范式，放弃使用JOIN。访问量上升之后对MySQL进行分库，后期数据量达到百万级时，对MySQL进行分表。
* 业务层尽量使用Redis缓存，减少MySQL访问量。

第四章 业务层的设计



4.1 SOA架构

SOA，即面向服务的架构（Service-Oriented Architecture），是现代大型企业级应用的标配。SOA架构的主要目的是为了提高系统的可用性、可扩展性以及可维护性。SOA是一种设计思想，他将一个完整的系统拆散成一个个独立的子系统，各个子系统独立运行提供一组功能上内聚的服务，然后在高层上对这些服务进行组装，完成最终的业务逻辑。这样做的好处是大大减少了业务代码的复杂度，减少了代码的重复即提高了代码的复用程度。同时，每个子系统可以独立于整个系统进行监控、部署、调整和优化。SOA架构的设计难点在于拆分系统的粒度：若子系统太多，那么运维的成本会太大；若子系统太少，子系统提供的功能不够内聚，就会出现系统之间循环依赖，代码复用率低等问题。



图 9 SOA架构

SOA架构一般是系统演进的最终形态，但是在系统设计的初期，我们就可以为之后的服务化打下基础。

SOA架构有三大组件构成：服务注册中心、服务提供者、服务消费者。

三者的关系如下图：

* 服务提供者向服务注册中心注册自己所能提供的服务。
* 服务消费者向服务注册中心查询自己所需要的服务，服务注册中心告知所请求的服务提供者的信息，并会在之后向服务消费者推送服务的最新变化。
* 服务消费者根据服务注册中心提供的信息调用服务。

服务注册中心可以视为是对服务消费者与服务提供者的解耦。服务的消费者与提供者互相不了解对方，而只知道服务注册中心的信息。这样一来，服务消费者与服务提供方都可以任意变化。服务消费者可以有任意的数量、调用任意的服务；服务提供者可以随意增减实例数量，甚至同一个服务可以有不同的实现。

故SOA架构的核心在于服务注册中心的设计与实现（传统的Web Service由于过于陈旧，不适合现代化的互联网开发需求，讨论见4.3节）。接来下讨论服务注册中心的实现方案。

服务注册中心设计的关注点：

* 服务查询策略
* 服务注册与查询效率
* 服务提供者状态监控
* 实时推送
* 接口

**服务查询策略** 服务查询策略不是简单地将所有名称符合的服务返回就结束的事。服务查询必须能够保证消费者查询到的服务是可用的。所以其中需要对服务提供者所提供的服务版本、服务调用协议以及服务提供者的状态等进行一定的过滤，如消费者所请求的服务版本必须与提供者所提供的版本兼容；消费者所使用的协议在提供者所提供的协议中；服务提供者如果处于不可用或者高负载状态则需要屏蔽该提供者。同时，服务查询要能够提供一定的负载均衡，使所有的服务提供者的负载平均。

**服务注册与查询效率** 服务注册并不是一个会被频繁使用，但是服务查询必定会被特别频繁地使用。故设计时必须考虑这种读多写少的情况。且服务注册中心对于数据一致性的要求较高，因为一旦返回了一个无效的提供者给消费者，那么必然会产生服务调用失败。同时，查询的效率必须高，才能保证消费者每次调用都能成功。

**服务提供者状态监控** 服务注册中心必须能够实时地检测服务提供者的状态。一旦发现服务提供者宕机，则立刻屏蔽该提供者，并通知消费者。同时，服务注册中心也需要能够对流量进行一定的监控以及导流，以保持服务提供者可以以最高效率提供服务。

**实时推送** 服务注册中心需要将服务的变化实时通知至服务消费者处，使消费者可以及时调整服务调用的目标，防止调用失败。故设计时必须考虑到推送的实现方式。

**接口** 服务注册中心本身也是一个对外提供服务的系统，故接口设计也必须考虑到简单易用以及兼容的问题。

下面讨论具体的实现方案：

1. 基于DNS[[19]](#footnote-19)实现

在内网搭建一个DNS服务器可以实现一个轻量级的服务注册中心。服务消费者通过域名调用服务，DNS返回服务提供者的真实IP，同时可以在DNS处实现负载均衡的逻辑。

这种方案的优点是十分容易实现，缺点是这种方案过于简陋，没有对于服务提供者的监控方法，也没有能够向服务调用方推送服务变更信息的有效方法。且DNS只能提供服务提供者的IP地址信息，不能提供如服务提供者的服务版本、服务通信协议等关键信息。更加致命的一个弱点是DNS系统一般都会有分布式的、分级的缓存。即使DNS信息已经更新，也需要较长一段时间才会被服务调用者感知。这使得变更服务的信息变得异常困难。

所以该方案适用比较简单的轻量级的服务化结构的系统中。



图 10 基于DNS的服务注册中心

1. 基于DUBBO框架

“DUBBO是一个分布式服务框架，致力于提供高性能和透明化的RPC远程服务调用方案，是阿里巴巴SOA服务化治理方案的核心框架，每天为2,000+个服务提供3,000,000,000+次访问量支持，并被广泛应用于阿里巴巴集团的各成员站点。”[[20]](#footnote-20)

但是实际上阿里巴巴内部已放弃DUBBO，停止了对其的维护（所以开源），转向了HSF（未开源）。DUBBO是一套非常重量级的框架，且只有Java语言版，所以一旦使用，所有的服务必须使用Java语言开发，所以该框架的应用实际上并没有想象中的那么广泛。

1. 基于Zookeeper实现

Zookeeper是Apache基金会的开源项目之一，它主要提供的功能是分布式系统的同步功能。Zookeeper可以做到数据变化的实时推送，且可以集群部署，可靠性很高。Zookeeper虽然不是为了SOA而设计的，但是却十分适合用来作为服务注册中心。事实上，Apache的Curator项目，作为官方的Zookeeper的Java客户端，自带服务注册与发现的功能。且可以通过该框架，将Zookeeper改造成一个以RESTful为接口的服务注册中心。

Zookeeper使用的是基于PAXOS[[21]](#footnote-21)的ZAB[[22]](#footnote-22)协议，是一种最终一致性模型，所以在数据一致性上稍逊色，但是由于Zookeeper完全基于内存，所以查询的效率非常高。且Zookeeper通过与客户端保持长连接来进行实时的变更通知，可以将服务提供者的信息缓存在消费者端，只有变更时才会重新查询，这样做进一步提高了查询的效率。Zookeeper集群可用性极高，是一种Leader-Follower集群，使用ZAB协议进行Leader选举，最多可以容忍集群中一半节点宕机。



图 11 Zookeeper方案

服务化是大型企业级应用的最终演化目标，Kikbug系统在现阶段还不需要达成服务化的目标，那样只会使得系统变得难以开发和维护。但是在设计业务层时，必须有意识地将业务划分为一组一组内聚的服务，且服务尽量不能存在依赖关系，最重要的是服务之间不能有循环依赖关系。对于需要多个服务配合的业务，应在更高层，如Controller中进行对于服务的组合。

Kikbug系统的服务设计：

|  |  |
| --- | --- |
| **服务名** | **服务用途** |
| AppService | 应用服务，测试应用的管理 |
| BugService | Bug服务，测试结果的Bug管理 |
| DistributionService | 分发服务，测试应用的分发管理 |
| GroupService | 群组服务，群组管理 |
| KikbugService | Kikbug对外服务 |
| MessageService | 消息服务，站内消息管理 |
| MoocService | 慕测服务，对慕测接口的封装 |
| OperatorService | 群主服务，群主管理 |
| ProviderService | 厂商服务，厂商管理 |
| ReportService | 报告服务，应用测试报告管理 |
| StatisService | 统计服务，业务信息统计 |
| TaskService | 任务服务，测试应用的任务管理 |
| TesterService | 测试人员服务，测试人员的管理 |
| UserService | 用户服务，用户的管理 |

表格 5 Kikbug系统服务设计



图 12 Kikbug Service以及Controller依赖关系

4.2 RPC & RESTful

在分布式系统中，各个系统间必然存在相互的方法调用，即RPC（Remote Procedure Call）。传统的RPC技术有Java RMI、SOAP、CORBA，较现代化的有Thrift、gRPC、RESTful。Thrift、gRPC、RESTful三者之间的比较在第一章中已经阐明。在客户端与服务器端通信中，使用RESTful是个好方案，但是在服务器端内部，情况就不太一样了。Thrift/gRPC与RESTful在服务器端应用的优缺点如下：

|  |  |  |
| --- | --- | --- |
|  | 优点 | 缺点 |
| Thrift/gRPC | * 长连接，处理速度快 * 配合序列化库，数据传输量小 | * 所有接口与传递的数据需要额外定义，实现困难且维护难度大 * 需要第三方库来完成服务调用 * 无法部署于Web应用容器中 * 通常无法通过直接拦截响应数据来查找问题 |
| RESTful | * 使用HTTP协议，复杂度低 * 实现简单 * 便于部署 * 便于拦截请求及响应查找问题 | * 短连接，效率低 * 数据传输量大 |

由于一般内网的带宽极大，且不收取费用，所以数据传输量可以忽略。且现阶段Kikbug系统的压力并不大，所以可以不用担心性能问题。故使用RESTful作为Kikbug系统内部调用的通信方式。

Kikbug RESTful设计原则：

|  |  |  |
| --- | --- | --- |
| HTTP Method | 用例 | 属性 |
| GET | 查询某实体 | 安全、幂等 |
| POST | 创建某实体 | 非安全、非幂等 |
| PUT | 更新某实体 | 非安全、幂等 |
| DELETE | 删除某实体 | 非安全、幂等 |

表格 6 Kikbug RESTful设计原则

Kikbug RESTful URL设计原则：

|  |  |  |  |
| --- | --- | --- | --- |
| URL格式 | HTTP方法 | CONTENT-TYPE | 作用 |
| /entity/{id} | GET |  | 获取某实体详细信息 |
| /entity?criteria | GET |  | 按条件查询实体 |
| /entity/ | POST | application/json | 创建一个新实体 |
| /entity/{id} | PUT | application/json | 更新某实体 |
| /entity/{id} | DELETE |  | 删除某实体 |

表格 7 Kikbug RESTful URL设计原则

Kikbug RESTful接口设计如下（部分）：

|  |  |  |
| --- | --- | --- |
| URL | HTTP方法 | 说明 |
| /app | GET | 搜索应用 |
| /app/ | POST | 创建应用 |
| /app/{appId} | GET | 获取应用信息 |
| /app/{appId} | PUT | 更新应用信息 |
| /app/{appId} | DELETE | 删除应用 |
| /app/{appId}/audit/ | POST | 审核应用 |
| /app/{appId}/distribution/ | POST | 推送应用 |
| /app/distribution | GET | 获取应用推送列表 |
| /app/distribution/{distributionId} | GET | 获取应用推送信息 |
| /app/distribution/{distributionId} | PUT | 上传推送的总结报告 |
| /app/categories | GET | 获取应用所有分类 |
| /bug | GET | 搜索BUG |
| /bug/ | POST | 创建BUG |
| /bug/{bugId} | GET | 获取BUG信息 |
| /bug/{bugId} | PUT | 更新BUG信息 |
| /bug/{bugId} | DELETE | 删除BUG |
| /bug/categories | GET | 获取BUG所有分类 |
| /report | GET | 搜索测试报告 |
| /report/ | POST | 创建测试报告 |
| /report/{reportId} | GET | 获取测试报告信息 |
| /report/{reportId} | PUT | 更新测试报告信息 |
| /report/{reportId} | DELETE | 删除测试报告 |
| /report/{reportId}/review/ | POST | 测试报告打分 |
| /task | GET | 搜索任务 |
| /task/ | POST | 创建任务 |
| /task/{taskId} | GET | 获取任务信息 |
| /task/taken | GET | 是否已接受任务 |

表格 8 Kikbug RESTful接口设计

4.3 微服务架构

SOA架构是一种设计思想，在过去，人们使用一种叫Web Service的技术/规范来实现这种思想。Web Service中的几大核心为UDDI（Universal Description, Discovery and Integration）作为服务注册中心、WSDL（Web Service Definition Language）作为服务定义、SOAP（Simple Object Access Protocol）作为通信协议、BPEL（Business Process Execution Language）作为高层服务定义。这是一套重量级的规范，对所有的细节都做了完备的定义。然而，这套规范已经逐渐被互联网企业抛弃。

如此完备的规范被抛弃，其中涉及的原因有很多。首先，这套规范定义过于详细，实现起来过于复杂。如SOAP协议，要做一个简单的方法调用需要做大量的与实现逻辑毫无关系的定义工作。这与互联网这样需求快速变化，十分强调敏捷开发的背景不符。其次，现代互联网企业对SOA架构的应用往往是在企业内部。即企业内部的某个服务对企业内部其他服务提供服务，故对于接口定义并不是那么的严格。

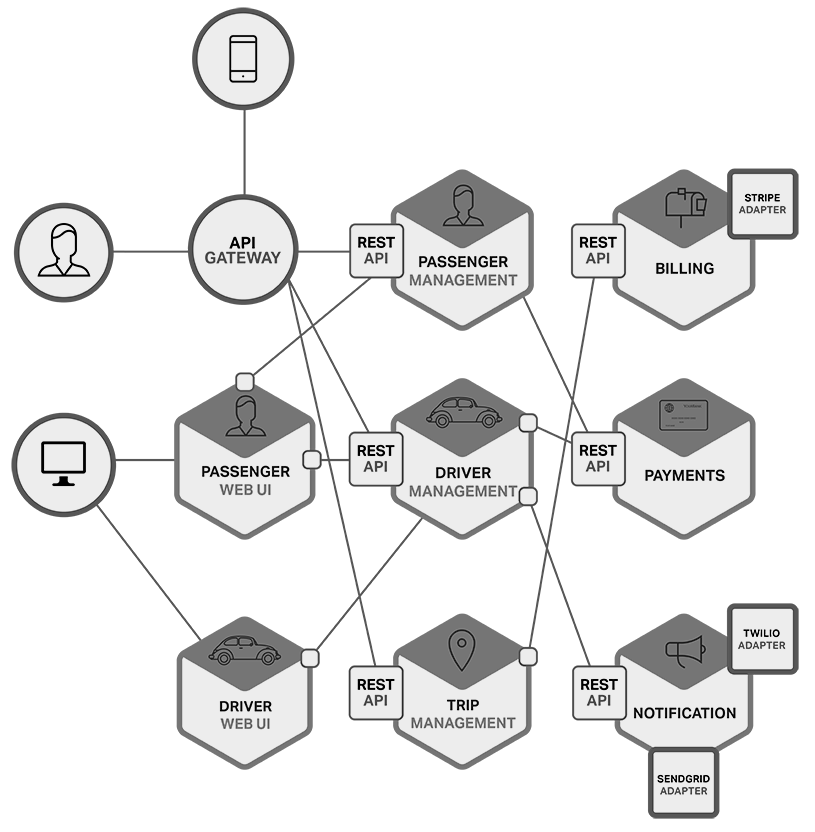


图 微服务架构设计样例[[23]](#footnote-23)

随着HTTP + JSON，即RESTful的崛起，SOAP逐渐被淘汰。即使是对外提供接口的服务型互联网产品（SaaS），也都会选择使用RESTful作为接口标准。同时，越来越多的敏捷开发服务化工具不断出现，如Spring Boot。这种使用RESTful的、注重敏捷开发的SOA架构就是微服务架构。

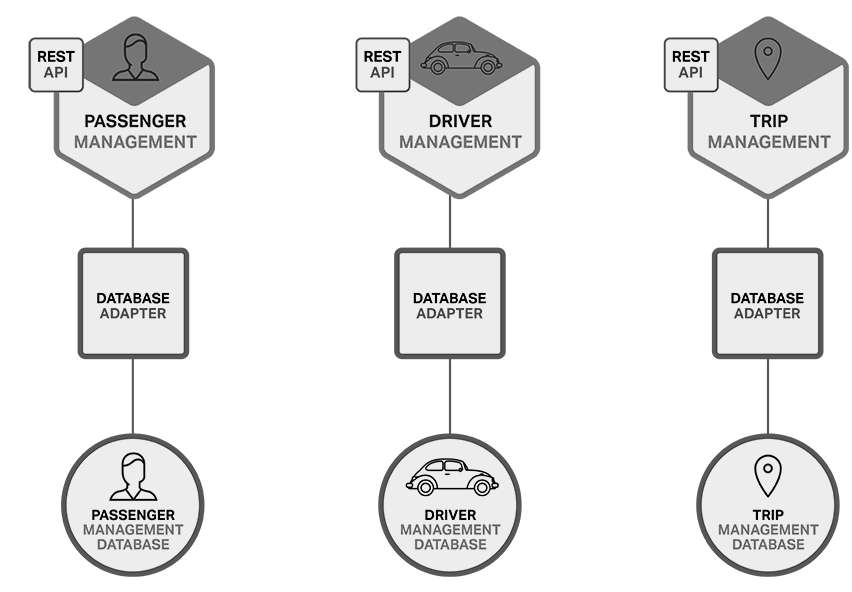


图 微服务架构设计样例

*“微服务并不是什么新鲜的理念，仅仅是一种面向服务交付（Service Oriented Delivery）的方式。”*[[24]](#footnote-24)

在Kikbug系统的演进中，如果引入了SOA架构，那么必然要选择微服务架构，而不是传统的WS架构。

第五章 展现层的设计



5.1 前后端分离

前后端分离是近年Web前端最热门的技术。优点已在第一章中简述。下面将详细分析对比传统的服务器端渲染技术（如JSP、ASP、PHP）等技术与前后端分离技术。

在传统的服务器端渲染技术中，浏览器所展现的Web页面（HTML页面）是由服务器生成的、拿来即用的页面。也就是说，浏览器在向服务器发出请求之后，服务器响应的是一个完整的HTML文档，可以直接被浏览器渲染成页面展示给用户。然而，有很多页面是需要动态变化的，故出现了jQuery一类的技术。这类技术可以让浏览器在渲染完静态的HTML页面之后通过JavaScript脚本的方式动态地根据用户操作，对整个页面进行修改。这种以服务器渲染为主、浏览器脚本为辅的网页设计模式已经统治了Web前端多年。



图 15 传统Web请求/响应方式

然而，随着前端页面的要求越来越高（用户体验的要求越来越高），这种模式变得越来越难以维护。且随着并发量的提升，服务器原本就已不堪重负，而客户端（即浏览器）也是拥有一定计算能力的，这部分计算能力没有被很好地利用起来。同时，谷歌Chrome发布了JavaScript V8引擎，使得浏览器JavaScript的运行速度有了质的飞跃，于是，前后端分离的思想应运而生。即服务器端只负责返回类似模板的一个（或多个）静态文件，以及相应的数据，由浏览器的JavaScript负责将HTML页面渲染出来，然后由浏览器最终将HTML页面渲染给用户。

静态文件是很少会变化的，且可以被浏览器缓存，也可以通过CDN加速。而动态的数据一般是以JSON方式返回的，数据量和一个完整HTML页面比起来小太多。所以对于服务器端来说，前后端分离大大减轻了访问压力。



图 16 前后端分离请求/响应方式

前后端分离后，服务器端分为静态文件服务器以及API服务器，这与传统的单一服务器的部署方式是不同的。正式由于这样的架构，使得前端可以独立出来，单独成为一个项目，独立进行版本控制与部署。Kikbug系统也将前端项目与API分离，成为两个项目独立开发。

5.2 MVC vs MVVM

MVC是一个经典的设计模式，即Model-View-Controller模式。

在MVC模式中:

* Controller负责接受用户输入，并操纵Model
* Model中是所有的业务逻辑，Model发生改变后会通知View
* View根据Model的变化进行相应的展现改变

有些人认为业务逻辑应该在Controller中，而另一些人认为应该在Model中。这涉及到了DDD（Domain Driven Design）中的贫血和充血模型。两种设计方法并没有太大的实际区别，都是可以接受的。

MVC的目的是把Model层和View层解耦，即把业务逻辑与展现解耦，使得这两部分可以独立地变化。MVC是一个比较容易实现的思想，使用传统的jQuery就可以很好地实现。



图 17 MVC设计模式

MVVM是一个微软提出的，已在.NET等微软其他产品中服役多年的设计模式，即Model-View-View Model模式。该模式比MVC模式更加进一步，将MVC中十分轻量级的Controller部分做成了重量级的View Model。View Model是视图逻辑模块，即所有对于View的控制都在View Model中进行，View只提供操纵View的接口。这样一来，Controller和View之间的耦合也解开了，View和View Model也可以独立变化。

MVVM的一大特点是数据的双向绑定。即用户对于界面的操作会实时改变View Model中的数据，View Model中数据的变化也会实时地改变界面。这种模式的优点是在页面复杂的情况下可以大大减少开发难度，当然由于它是对于MVC的进阶，它也拥有MVC的所有优点。



图 18 MVVM设计模式

总的来说，MVC适合界面简单、交互简单的应用，而MVVM适合开发大型的交互复杂的应用。

随着HTML5标准、V8引擎、前后端分离思想的出现，Web应用开始走向挑战桌面原生应用的道路。另一个热门的概念是SPA（Single Page Application），即单页应用。由于采用前后端分离，请求服务器只需要用过Ajax即可完成，故页面直接的跳转是完全没有必要的。由于没有了页面跳转，Web应用的响应速度就得到了大大地提升。但是随之而来的问题在于这样的应用由于所有的逻辑都在同一个页面中，所以十分复杂，MVVM正是处理这种问题的最佳方案。

5.3 React vs Angular

如今，MVVM框架已经成为Web前端的主流。典型的代表有React、Angular、Vue。下面主要对比讨论React以及Angular。

React是由Facebook推出的一个前端类库，小而精。它主要实现了前端的模块化开发。React的特点：

* 页面高度模块化
* 单向数据流
* VirtualDOM，页面高性能渲染
* JSX

Angular是由Google推出的一个前端框架，大而全。主要实现了MVVM思想，以及大量的实现单页应用的工具。

* MVVM数据双向绑定
* 完整的应用组件（HTTP客户端、页面路由等）

总的来说，React适用与开发需要高度定制的Web应用，而Angular适用于开发大部分普通的Web应用。如果使用React，那么免不了需要引入更多的类库或者框架来实现SPA。使用Angular虽然不用引入更多的类库，但是Angular的页面渲染速度远远比不上React。

Kikbug系统页面的复杂度不会很高，为了降低开发难度以及维护成本，故选择Angular作为开发框架。

第六章 部署与运维的设计



6.1 自动化构建工具

1. Maven

Kikbug的API服务器由于使用Java语言进行开发，故使用Maven作为构建工具。由于使用Spring MVC，所以有大量的Spring配置文件需要管理。这些Spring配置文件中的Bean定义是不需要随着外部的部署环境而变化的，但是类似与数据库地址这一类的配置是需要随着部署的环境变化的。故需要将这些部分单独抽取出来放置在properties文件中。

使用Spring的PropertyPlaceholderConfigurer可以做到从properties文件中读取配置，然后在装备Bean的时候将这些配置注入。只需要在Spring配置文件中定义Bean，然后指定properties文件的路径即可。

然而，我们的配置会有许多版本，如本地测试环境、线上测试环境、线上正式环境。故我们需要能够在不同的环境使用不同的配置文件。Spring无法支持这种功能（除非自己实现）。但是Maven提供的filter可以实现这样的需求。

将Spring配置文件中的properties文件地址也写为placeholder的形式，然后在Maven中创建多份Profile（dev、test、production）以及对应的变量，在使用Maven打包时，只需要指定使用哪一种Profile，Maven就会自动将Spring配置文件中的placeholder进行替换。然后Spring在加载Bean时，就会根据被替换之后的路径读取properties文件。

Maven配置：

|  |
| --- |
| <profiles>  <profile>  <id>dev</id>  <activation>  <activeByDefault>true</activeByDefault>  </activation>  <properties>  <env.id>dev</env.id>  <config.path>classpath:properties/dev</config.path>  </properties>  </profile>  <profile>  <id>test</id>  <properties>  <env.id>dev</env.id>  <config.path>classpath:properties/test</config.path>  </properties>  </profile>  <profile>  <id>production</id>  <properties>  <env.id>production</env.id>  <config.path>classpath:properties/production</config.path>  </properties>  </profile>  </profiles> |

Spring配置：

|  |
| --- |
| <bean id="configProperties" class="org.springframework.beans.factory.config.PropertiesFactoryBean">  <property name="locations">  <list>  <value>${config.path}/application.properties</value>  <value>${config.path}/jdbc.properties</value>  <value>${config.path}/redis.properties</value>  </list>  </property>  </bean>  <bean id="propertyConfigurer" class="org.springframework.beans.factory.config.PreferencesPlaceholderConfigurer">  <property name="properties" ref="configProperties"/>  </bean> |

Maven的另一大功能是依赖管理。Kikbug使用Maven下载所有项目依赖包，所以项目所使用的所有依赖包必须已经托管在Maven的中央库。为了免去每次部署都需要重新下载的时间，而且并不是所有的服务器都具有外网的访问权限，必须在本地缓存一份Kikbug所有依赖包的副本。后期如果项目需要，可以考虑在内网搭建Sonatype Nexus作为一个Maven私服。

1. Gulp

前端项目的构建工具采用了Gulp。Gulp的特点是以流的方式进行文件处理，故处理效率比Grunt高。前端构建的大致过程如下：

* 组装HTML（由公共部分以及各部分组件拼接）
* 编译SCSS并压缩
* 对JS进行Browserify[[25]](#footnote-25)并且压缩
* 复制其他资源（图片、JSON数据文件等）

Gulp构建脚本：

|  |
| --- |
| var srcPath = 'app';  var distPath = 'build';  var path = require('path');  var gulp = require('gulp');  var $ = require('gulp-load-plugins')({  pattern: ['gulp-\*', 'del', 'browserify', 'vinyl-source-stream', 'vinyl-buffer'],  rename: {  'gulp-html-extend': 'extender',  'gulp-minify-html': 'minifyHTML',  'gulp-ruby-sass': 'sass',  'gulp-ng-annotate': 'ngAnnotate',  'vinyl-source-stream': 'source',  'vinyl-buffer': 'buffer'  }  });  gulp.task('default', function() {  gulp.start('build');  });  // 清理  gulp.task('clean', function() {  $.del.sync([distPath + '/\*', '!' + distPath + '/.gitkeep'], { force: true });  });  // 组装HTML  gulp.task('html-main', function() {  var minifyOpts = {  conditionals: true,  spare: true  };  return gulp.src(srcPath + '/html/\*.html')  .pipe($.extender())  .pipe($.minifyHTML(minifyOpts))  .pipe(gulp.dest(distPath));  });  gulp.task('templates', function() {  var opts = {  conditionals: true,  spare: true  };  return gulp.src(srcPath + '/templates/\*\*/\*.html')  .pipe($.minifyHTML(opts))  .pipe(gulp.dest(distPath + '/templates'));  });  gulp.task('html', ['templates', 'html-main']);  // 图片资源  gulp.task('images', function() {  return gulp.src(srcPath + '/image/\*\*/\*')  .pipe(gulp.dest(distPath + '/image/'));  });  // JSON数据资源  gulp.task('json', function() {  return gulp.src(srcPath + '/json/\*\*/\*')  .pipe(gulp.dest(distPath + '/json/'));  });  gulp.task('resources', ['images', 'json']);  // 编译sass并压缩  gulp.task('styles', function() {  return $.sass(srcPath + '/scss/\*\*/\*.scss')  .on('error', $.sass.logError)  .pipe($.csso())  .pipe(gulp.dest(distPath + '/css/'));  });  // 检查JS文件语法  gulp.task('scripts', function() {  return gulp.src(srcPath + '/js/\*\*/\*.js')  .pipe($.jshint())  .pipe($.jshint.reporter('jshint-stylish'));  });  // broswerify  gulp.task('browserify', ['scripts', 'lib-css', 'lib-font'], function() {  var b = $.browserify();  b.require([  // 此处省略对外部库的引用  ], {  basedir: './bower\_components'  });  return b.add(srcPath + '/js/app.js')  .bundle()  .pipe($.source('dist.js'))  .pipe($.buffer())  .pipe($.uglify())  .pipe(gulp.dest(distPath + '/js/'));  });  gulp.task('\_build\_all', ['html', 'styles', 'resources', 'browserify']);  gulp.task('build', ['clean', '\_build\_all']);  module.exports = gulp; |

6.2 虚拟化

为了达到高可扩展性，部署时需要尽量与系统环境隔离。由于项目往往有大量的依赖，如果需要部署新系统，安装依赖是一个很耗时的过程。且由于操作系统的版本可能不同，安装依赖的过程有很大的概率会失败，需要根据不同的操作系统做出调整。这大大降低了系统的可扩展性。故我们需要一种技术手段，来使得应用可以快速地在一台新的机器上部署。

Kikbug系统依赖外部组件如下：

|  |  |  |
| --- | --- | --- |
| JDK 1.8 | Apache Tomcat 8 | Nginx |
| MySQL | Maven | Redis |
| Nodejs / NPM | Ruby / RubyGems | Bower |
| Gulp | Sass |  |

表格 9 Kikbug依赖

Docker是近年比较热门的技术，它是一种轻量级的虚拟机，一种应用容器。Docker是基于LXC的，启动一个虚拟机的速度极快。使用Docker可以很方便地在多个机器上进行部署，而不需要担心依赖的问题。

首先，我们需要用Docker创建一个镜像：

1. 使用Docker拉取一个操作系统的镜像。
2. 使用该镜像生成一个容器
3. 在容器中安装应用所需要的依赖
4. 将容器保存为一个镜像

至此，这个镜像就是我们需要的应用环境了。我们可以将这个镜像复制到任意机器上，然后通过Docker启动，只要在目标机器上可以运行Docker，就不用担心依赖的问题。

Docker的思想是把一个应用打包为一个容器，即一个容器就是一个完整的可运行的应用。同时，容器应当是无状态的，即所有的数据文件、日志文件都应该通过目录映射的方式映射到宿主机上，这样才可以保证数据的安全性，也保证了Docker容器可以在任意时刻启停。

根据Docker的设计原则，Kikbug系统的Docker容器设计如下：

Kikbug-DB Dockerfile：

|  |
| --- |
| FROM centos:centos6  MAINTAINER zhangxuan  RUN yum install -y wget tar  WORKDIR /tmp  # MySQL 5.7  RUN wget http://dev.mysql.com/get/mysql57-community-release-el6-7.noarch.rpm && \  yum localinstall -y mysql57-community-release-el6-7.noarch.rpm && \  yum install -y mysql-server && \  rm -f /etc/my.cnf  # PHP  RUN rpm -Uvh http://mirror.webtatic.com/yum/el6/latest.rpm && \  yum install -y php56w.x86\_64 php56w-fpm.x86\_64 php56w-mbstring.x86\_64 php56w-mysql.x86\_64  # phpMyAdmin  RUN wget https://files.phpmyadmin.net/phpMyAdmin/4.5.3.1/phpMyAdmin-4.5.3.1-all-languages.tar.gz && \  mkdir -p /var/www/html && \  cd /var/www/html && \  tar xzvf /tmp/phpMyAdmin-4.5.3.1-all-languages.tar.gz && \  mv phpMyAdmin-4.5.3.1-all-languages pma && \  cp pma/config.sample.inc.php pma/config.php  # Nginx  ADD nginx.repo /etc/yum.repos.d/nginx.repo  RUN yum install -y nginx && \  rm -f /etc/nginx/conf.d/default.conf  # ENV  ADD launch /bin/launch  RUN chmod u+x /bin/launch  VOLUME ["/var/db/mysql"]  EXPOSE 80  CMD launch && bash  # config MySQL  ADD my.cnf /etc/my.cnf  # config Nginx  ADD nginx\_http.conf /etc/nginx/conf.d/nginx\_http.conf |

Kikbug-Redis Docker File：

|  |
| --- |
| FROM centos:centos6  MAINTAINER zhangxuan  RUN yum install -y wget tar gcc  WORKDIR /tmp  # Redis  RUN wget http://download.redis.io/releases/redis-3.0.6.tar.gz && \  tar xzvf redis-3.0.6.tar.gz && \  mv redis-3.0.6 redis && \  cd redis && \  make && \  make install && \  rm -f redis.conf && \  mkdir -p /data/db/redis  # ENV  ADD launch /bin/launch  RUN chmod u+x /bin/launch  VOLUME ["/data/db/redis"]  CMD launch && bash  # config Redis  ADD redis.conf redis |

Kikbug-Tomcat Docker File：

|  |
| --- |
| FROM centos:centos6  MAINTAINER zhangxuan  RUN yum install -y wget tar  WORKDIR /tmp  # JDK 1.8.0  RUN yum install -y java-1.8.0-openjdk-devel.x86\_64  # Maven  RUN wget http://apache.opencas.org/maven/maven-3/3.3.9/binaries/apache-maven-3.3.9-bin.tar.gz && \  tar xzvf apache-maven-3.3.9-bin.tar.gz && \  mv apache-maven-3.3.9 maven  RUN mkdir -p /root/.m2/repository  # Tomcat  RUN wget http://mirror.bit.edu.cn/apache/tomcat/tomcat-8/v8.0.32/bin/apache-tomcat-8.0.32.tar.gz && \  tar xzvf apache-tomcat-8.0.32.tar.gz && \  mv apache-tomcat-8.0.32 tomcat && \  rm -f tomcat/conf/tomcat-users.xml tomcat/conf/server/xml  # ENV  ADD var.sh /etc/profile.d  ADD launch /bin/launch  RUN chmod u+x /bin/launch  RUN mkdir -p /data/app  VOLUME ["/root/.m2/repository", "/data/app", "/data/log"]  EXPOSE 8080  CMD launch && bash  # config Tomcat  ADD setenv.sh tomcat/bin  ADD tomcat-users.xml tomcat/conf  ADD server.xml tomcat/conf |

Kikbug-Web Docker File：

|  |
| --- |
| FROM centos:centos6  MAINTAINER zhangxuan  RUN yum install -y wget git  WORKDIR /tmp  # Nginx  ADD nginx.repo /etc/yum.repos.d/nginx.repo  RUN yum install -y nginx && \  rm -f /etc/nginx/conf.d/default.conf  # NodeJS  RUN curl --silent --location https://rpm.nodesource.com/setup\_4.x | bash - && \  yum install -y nodejs && \  npm install -g cnpm --registry=https://registry.npm.taobao.org && \  cnpm install -g bower && \  cnpm install -g gulp  # Ruby  RUN yum install -y ruby rubygems && \  gem sources --remove http://rubygems.org/ && \  gem sources -a https://ruby.taobao.org/ && \  gem install sass  # ENV  ADD launch /bin/launch  RUN chmod u+x /bin/launch  VOLUME ["/data/app", "/data/app/node\_modules", "/data/app/bower\_components", "/var/log/nginx"]  EXPOSE 80  CMD launch && bash  # config Nginx  ADD nginx\_http.conf /etc/nginx/conf.d/nginx\_http.conf |

6.3 集群监控

集群监控有许多工具，Kikbug选用Zabbix作为集群的机器状态监控工具。现讨论业务监控的实现方案。

业务监控离不开对于业务日志的分析。由于处于分布式的环境下，首先需要从各个业务服务器上收集这些业务日志。接下来讨论业务日志的收集方案。

1. Flume + MySQL

Flume是Apache基金会的一个开源项目，用于在分布式的环境下收集日志。部署时只需要在目标机器上启动一个Flume Agent，在收集的机器上配置存储数据用的Sink在数据量不大的情况下，使用MySQL存储由于不需要引入新的组件，所以可以接受

1. Flume + Kafka + Hive + HBase

后期当访问量增加，数据量变大之后，使用Flume收集日志会遇到问题，即数据收集的速度大于数据写入DB的速度。那么我们就需要有一个消息队列来对数据进行一个缓冲。

Kafka是由Apache基金会维护的一款消息队列，它的特点是

* 单broker可以每秒处理百万次读写，每一个broker可以在性能没有影响的情况下处理TB级消息
* 数据是以分区的格式存储在整个集群中，消息持久化在磁盘上，且有多个备份，不会造成数据丢失

当数据量增大后，RDBMS不能胜任存储与查询的工作，需要由NoSQL完成。HBase是Apache基金会维护的建立在HDFS上的一款NoSQL数据库。适用与写多读少的场景，正符合日志分析。

Hive同样是由Apache基金会维护的一套基于Hadoop的查询工具。Hive本身不存储和计算数据，它依赖于HDFS和MapReduce。用户可以通过类似SQL的HQL语句进行查询，HQL经过编译转为MapReduce后使用Hadoop运行。

Hive适用于大量日志的离线分析，而HBase更适用于实时数据查询。故二者配合可以很好地完成日志分析的任务。但是使用该方案需要额外部署Hadoop集群。

由于方案2依赖大量额外的系统组件（Kafka、HDFS），且这些组件都需要集群部署，现阶段没有足够的资源支撑这套系统。故Kikbug前期将使用方案1作为业务监控。随着访问量的上升，逐渐迁移至方案2。

6.4 Web端部署

Web端使用Docker部署一套Web端的构建环境，同时将部署Nginx。代码目录通过目录映射，映射至宿主机。宿主机上使用git更新代码。同时，为了构建的速度，node\_modules以及bower\_components将映射到宿主机，使得宿主机上的多个虚拟机可以共享。后期如需要在多台物理机上部署时，可以考虑将这两个目录映射为网络目录，在多个宿主机之间共享。同时，所有的日志文件将被映射至宿主机。

部署流程：

* 1. 宿主机拉取最新代码
  2. 进入Docker容器，使用gulp build命令构建
  3. 将构建完毕的文件复制到Ngnix网站根目录中

6.5 服务器部署

服务器端部署与Web端类似。首先会使用Docker部署一套拥有JDK1.8、Maven、Tomcat8的环境。代码文件映射至宿主机，Maven的本地repository也同样映射至宿主机。

部署流程：

* 1. 宿主机拉取最新代码
  2. 进入Docker容器，使用mvn clean package -P production命令构建
  3. 使用mvn tomcat7:undeploy tomcat7:deploy-only命令发布至tomcat

第七章 项目总结

Kikbug系统经过本次重做，从单机演变为了分布式的架构，各个部分的设计有了完整的演进方案，大大提高了系统的可用性、可扩展性。这也符合互联网应用的演进规律，即从单机系统走向大型分布式系统。

在设计Kikbug演进方案的过程中，时时刻刻体现了软件工程中“没有银弹”。任何技术目标都会有多种解决方案，而这多种解决方案不能用“好”和“不好”来形容，只能用“合适”以及“不合适”来描述。随着时间的推移，项目资源与需求的变化，“合适”与“不合适”也发生着转变，所以一个优秀的项目应该是能够随着系统的增长，而不断演进，适应新的变化的系统。

在互联网项目方案设计过程中，最重要的一点，也是最难的一点就是如何设计出一个不仅适合项目初级阶段敏捷开发需求的系统，也要能够在未来可以在不重做系统的前提下继续演进的系统。这要求架构师有丰富的经验。

本次对于Kikbug项目的重做，不仅提高了Kikbug系统的用户体验，也为Kikbug系统未来的发展铺平了道路。

参考文献

1. Howe J. The rise of crowdsourcing[J]. Wired magazine, 2006, 14(6): 1-4.
2. Brabham D C. Crowdsourcing as a model for problem solving an introduction and cases[J]. Convergence: the international journal of research into new media technologies, 2008, 14(1): 75-90.
3. Padhy R P, Patra M R, Satapathy S C. RDBMS to NoSQL: Reviewing some next-generation non-relational databases[J]. International Journal of Advanced Engineering Science and Technologies, 2011, 11(1): 15-30.
4. Nance C, Losser T, Iype R, et al. Nosql vs rdbms-why there is room for both[C]//Proceedings of the Southern Association for Information Systems Conference. 2013: 111-116.
5. Macedo T, Oliveira F. Redis cookbook[M]. " O'Reilly Media, Inc.", 2011.
6. Erl T. Soa: principles of service design[M]. Upper Saddle River: Prentice Hall, 2008.
7. Krafzig D, Banke K, Slama D. Enterprise SOA: service-oriented architecture best practices[M]. Prentice Hall Professional, 2005.
8. Sneps-Sneppe M, Namiot D. Micro-service Architecture for Emerging Telecom Applications[J]. International Journal of Open Information Technologies, 2014, 2(11): 34-38.
9. Lamport L. Paxos made simple[J]. ACM Sigact News, 2001, 32(4): 18-25.
10. Rao J, Shekita E J, Tata S. Using paxos to build a scalable, consistent, and highly available datastore[J]. Proceedings of the VLDB Endowment, 2011, 4(4): 243-254.
11. Richardson L, Ruby S. RESTful web services[M]. " O'Reilly Media, Inc.", 2008.
12. Christensen J H. Using RESTful web-services and cloud computing to create next generation mobile applications[C]//Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications. ACM, 2009: 627-634.
13. Pautasso C, Zimmermann O, Leymann F. Restful web services vs. big'web services: making the right architectural decision[C]//Proceedings of the 17th international conference on World Wide Web. ACM, 2008: 805-814.
14. Hamad H, Saad M, Abed R. Performance Evaluation of RESTful Web Services for Mobile Devices[J]. Int. Arab J. e-Technol., 2010, 1(3): 72-78.
15. Li L. Analysis and Application of MVVM Pattern[J]. Microcomputer Applications, 2012, 12: 019.
16. Nitze A, Schmietendorf A. Modularity of javascript libraries and frameworks in modern web applications[C]//Selected Topics to the User Conference on Software Quality, Test and Innovation (ASQT 2014), OCG, Klagenfurt, AT. 2014.
17. Hunt P, Konar M, Junqueira F P, et al. ZooKeeper: Wait-free Coordination for Internet-scale Systems[C]//USENIX Annual Technical Conference. 2010, 8: 9.
18. Junqueira F, Reed B. ZooKeeper: Distributed Process Coordination[M]. " O'Reilly Media, Inc.", 2013.
19. Merkel D. Docker: lightweight linux containers for consistent development and deployment[J]. Linux Journal, 2014, 2014(239): 2.
20. Sneps-Sneppe M, Namiot D. Micro-service Architecture for Emerging Telecom Applications[J]. International Journal of Open Information Technologies, 2014, 2(11): 34-38.

致谢

最后，感谢陈振宇老师为我提供的这次为Kikbug系统设计架构的机会。同时感谢陈丹丹同学在Kikbug系统Web端与服务器端实现工作中的贡献。也感谢开发Kikbug系统的前辈，是他们为Kikbug系统做了完整的需求分析，为我们的开发工作铺平了道路。

感谢Apache基金会、GitHub对于开源社区的大力支持，没有他们，互联网的今天也不会如此繁荣。同时也感谢各大开源社区无私贡献的工程师们、Docker的所有Contributor、Gulp的所有Contributor为我们提供了如此丰富的类库、强大的工具，从而大大提高了开发效率，缩短了开发周期，使得Kikbug项目能在短时间内完成。

1. 众包，指的是一个公司或机构把过去由员工执行的工作任务，以自由自愿的形式外包给非特定的（而且通常是大型的）大众网络的做法。 [↑](#footnote-ref-1)
2. 共享经济，一般是指以获得一定报酬为主要目的，基于陌生人且存在物品使用权暂时转移的一种新的经济模式。 [↑](#footnote-ref-2)
3. http://www.imdada.cn/ [↑](#footnote-ref-3)
4. https://daojia.jd.com/ [↑](#footnote-ref-4)
5. http://www.zbj.com/ [↑](#footnote-ref-5)
6. http://zhongbao.alibaba.com/ [↑](#footnote-ref-6)
7. http://www.weichaishi.com/ [↑](#footnote-ref-7)
8. RESTful，Representational State Transfer，一种接口设计风格。 [↑](#footnote-ref-8)
9. Thrift是Facebook推出的一套RPC框架，现已加入Apache基金会。 [↑](#footnote-ref-9)
10. gRPC是Google推出的一套RPC框架，较年轻。 [↑](#footnote-ref-10)
11. IDL，Interface Definition Language，一种中立的方式来描述接口，使得在不同平台上运行的对象和用不同语言编写的程序可以相互通信交流。 [↑](#footnote-ref-11)
12. LB，Load Balance，负载均衡。 [↑](#footnote-ref-12)
13. 即AliMySQL [↑](#footnote-ref-13)
14. BASE，Basically Available, Soft-state, Eventually Consistent。 由 Eric Brewer 定义。 [↑](#footnote-ref-14)
15. Gossip是一种分布式数据同步算法，符合最终一致性。 [↑](#footnote-ref-15)
16. http://db-engines.com/en/ranking [↑](#footnote-ref-16)
17. 幂等，如某个操作f幂等，则f(x) = f(f(x)) [↑](#footnote-ref-17)
18. 即Redis的EXPIRE操作，可在指定时间自动删除键值。 [↑](#footnote-ref-18)
19. DNS，域名服务器。 [↑](#footnote-ref-19)
20. 参见http://dubbo.io [↑](#footnote-ref-20)
21. Paxos算法是莱斯利·兰伯特于1990年提出的一种基于消息传递的一致性算法。 [↑](#footnote-ref-21)
22. ZAB，Zookeeper Atomic Broadcast [↑](#footnote-ref-22)
23. 参见http://nginx.com/blog/introduction-to-microservices/ [↑](#footnote-ref-23)
24. http://www.infoq.com/cn/news/2014/06/microservices-soa/ [↑](#footnote-ref-24)
25. Browserify是将符合Nodejs标准的JavaScript脚本转化成浏览器可用的版本的工具 [↑](#footnote-ref-25)