**密级： 保密期限：**

xm 拷贝

**硕士学位论文**



**题目： 基于微服务的交互式应用**

**服务端设计与实现**

**学 号： 2015140369**

**姓 名： 宋海云**

**专 业： 电子与通信工程**

**导 师： 张洪欣**

**学 院： 电子工程学院**

**2018年 1月 8日**

**密级： 保密期限：**

xm 拷贝

**硕士学位论文**



**题目：THE BACKEND DEVELOPMENT OF SNS APPLICATION BASED ON MICROSERVICES**

**学 号： 2015140369**

**姓 名： Haiyun Song**

**专 业：Communication Engineering**

**导 师： Hongxin Zhang**

**学 院：School of Electronic Engineering**

**2018年 1月 8日**

独创性（或创新性）声明

本人声明所呈交的论文是本人在导师指导下进行的研究工作及取得的研究成果。尽我所知，除了文中特别加以标注和致谢中所罗列的内容以外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得北京邮电大学或其他教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示了谢意。

申请学位论文与资料若有不实之处，本人承担一切相关责任。

本人签名： 日期：

独创性（或创新性）声明

本人声明所呈交的论文是本人在导师指导下进行的研究工作及取得的研究成果。尽我所知，除了文中特别加以标注和致谢中所罗列的内容以外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得北京邮电大学或其他教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示了谢意。

申请学位论文与资料若有不实之处，本人承担一切相关责任。

本人签名： 日期：

基于微服务的交互式应用服务端设计与实现

摘 要

“微服务”架构是近期软件应用领域非常热门的概念。从技术层面看，互联网及云计算中大量开源轻量级技术不停涌现，并日渐成熟，这催生了新的架构设计风格：微服务架构。微服务架构下的开发相互独立，模块与模块之间高内聚低耦合，具有很好的扩展性。这样就能够极大地加快业务开发速度，尽快交付市场。

本文在微服务的架构基础之上，分析了交互式应用的业务需求，对服务端的微服务模块设计展开研究，对相应的微服务模块进行功能拆分，以高性能高稳定性为目的，对其中的部分关键逻辑进行了设计与实现。主要研究内容如下:

1. 对微服务架构设计原则进行探讨，分析业务需求，拆分出相应的微服务模块。介绍服务端相关技术，包括数据库技术、缓存技术，异步消息队列技术，根据业务需求组合相关技术。
2. 基于微服务框架设计服务端功能模块,制定模块设计方案，其中微服务模块主要包括有:基础服务，提交服务，邀请服务。在实现交互式应用支持高并发的要求下，对微服务模块的数据持久化存储，缓存，异步处理等关键点进行了设计。
3. 实现了微服务模块相关功能,并对功能及性能进行测试，绘制数据分析图表，验证服务在高并发下的性能：例如查询服务在万级QPS调用下，服务延时平均在2-3毫秒左右，使得用户在浏览内容时有良好的用户体验，满足业务需求。

本文在基于微服务架构的基础上，设计与实现了支持高并发的交互式应用的服务端功能模块。基于微服务的架构设计能够让开发人员精确地拆分通用的功能模块，模块之间高内聚低耦合，每个模块只关注自己独一功能，模块数量上的扩展也非常方便，是更加灵活的架构模式。

**关键词：**微服务 服务端开发 数据库

T**HE BACKEND DEVELOPMENT OF SNS APPLICATION BASED ON MICROSERVICES**

**Abstract**

The "Microservices" architecture is the recently popular concept in software applications. From a technical perspective, a large number of open source lightweight technologies of the Internet and cloud computing continue to emerge and mature gradually, which has given rise to a new architecture design style: micro-service architecture. the development under the framework of Microservices is independent, both high cohesion and less coupling between modules, while with good scalability. This will greatly speed up the business development, delivery to the market as soon as possible.

Based on the architecture of Microservices, this paper analyzes the requirements of interactive applications developments; design micro-services modules on the server side, including separates the corresponding micro-services modules; with the purpose of high performance and stability, design and implement the Key part of service logic. The main research contents are as follows:

1)Describe interactive application developing requirements, considering mass data, service’s stability and disaster recovery. Discuss the design principles of Microservices architecture architecture, and study related technologies, including Restful API, database, cache, and asynchronous message queue.

2)Based on Microservices Architecture design backend micro-service modules, and implement some Key result. The micro-service modules can be separate as follow: query service, submit service, user service. In the specific implementation of the module, the Key points of design are data persistence storage, cache design and asynchronous message queue.

3)Implement micro-services module related functions, test services function and performance, draw the data charts and do some analysis.

Microservices architecture allows developers to split the common functional modules accurately, while the development of modules are high cohesion less coupling, and each module only focus on their own unique features. It supports expansion, which can be considered as a more flexible architecture. Based on Microservices architecture, this paper designs and implements backend macro-service modules which support high-concurrency performance on interactive applications.

**Key words：**microservices backend development mysql

**目 录**

[第一章 绪论 1](#_Toc503101672)

[1.1 研究背景 1](#_Toc503101673)

[1.2 研究内容与研究意义 2](#_Toc503101674)

[1.3 互联网应用架构发展现状 2](#_Toc503101675)

[1.4 各章安排 3](#_Toc503101676)

[1.5 小结 3](#_Toc503101677)

[第二章 项目关键技术分析 4](#_Toc503101678)

[2.1 微服务架构设计原则 4](#_Toc503101679)

[2.2 Web服务端关键技术分析 4](#_Toc503101680)

[2.2.1 Restful 4](#_Toc503101681)

[2.2.2 RPC Thrift 5](#_Toc503101682)

[2.2.3 Python语言及框架 7](#_Toc503101683)

[2.2.4 数据持久化 8](#_Toc503101684)

[2.2.5 缓存技术 10](#_Toc503101685)

[2.2.6 消息队列 10](#_Toc503101686)

[2.3 小结 11](#_Toc503101687)

[第三章 微服务架构及模块总体设计 12](#_Toc503101688)

[3.1 服务端架构设计需求分析 12](#_Toc503101689)

[3.1.1 微服务拆分设计 12](#_Toc503101690)

[3.2 整体架构分析 13](#_Toc503101691)

[3.2.1 交互式应用微服务架构设计 13](#_Toc503101692)

[3.2.2 存储设计 14](#_Toc503101693)

[3.2.3 消息队列 16](#_Toc503101694)

[3.3 微服务模块设计 17](#_Toc503101695)

[3.3.1 查询服务总体设计 17](#_Toc503101696)

[3.3.2 提交服务总体设计 18](#_Toc503101697)

[3.3.3 邀请服务的总体设计 20](#_Toc503101698)

[3.4 模块对外提供接口 21](#_Toc503101699)

[3.4.1 Json 服务返回数据 21](#_Toc503101700)

[3.4.2 Protobuf 对外接口提供 21](#_Toc503101701)

[3.5 小结 21](#_Toc503101702)

[第四章 重要模块及接口详细设计 22](#_Toc503101703)

[4.1 提交服务具体设计 22](#_Toc503101704)

[4.1.1 提交类事件定义设计 22](#_Toc503101705)

[4.1.2 提交服务接口设计 23](#_Toc503101706)

[4.1.3 存储设计与实现 26](#_Toc503101707)

[4.1.4 缓存设计与实现 29](#_Toc503101708)

[4.2 查询服务 31](#_Toc503101709)

[4.2.1 查询服务接口设计 31](#_Toc503101710)

[4.3 邀请服务 33](#_Toc503101711)

[4.3.1 邀请服务接口设计 33](#_Toc503101712)

[4.3.2 存储设计 35](#_Toc503101713)

[4.4 小结 39](#_Toc503101714)

[第五章 测试 40](#_Toc503101715)

[5.1 测试环境 40](#_Toc503101716)

[5.2 测试方法 40](#_Toc503101717)

[5.3 小结 47](#_Toc503101718)

[第六章 总结 48](#_Toc503101719)

[参考文献 49](#_Toc503101720)

[致谢 52](#_Toc503101721)

1. 绪论

## 研究背景

“微服务”架构是近期软件应用领域非常热门的概念。

传统的整体式架构(Monolithic Architecture)应用开发系统（如CRM、ERP等大型应用），随着新需求的增加，业务更新以及维护修复现有大型整体应用变得越来越困难。随着移动互联网的发展，很多企业将自己的应用迁移到适用于现代化UI界面架构上面，以便能兼容移动设备，新的应用架构要求企业实现功能的快速上线。传统的SOA架构模式可以通过标准化服务接口实现能力的重用，但对于快速变化的需求，受到整体式应用的限制。

随着应用云化的日益普及，基于云端的应用具有与传统IT不同的技术基因和开发运维模式。从技术方面看，云计算及互联网公司大量开源轻量级技术不停涌现并日渐成熟： 比如轻量级协议（Restful API接口），简化的基础设施发展（操作系统虚拟化，容器技术等），新的可替代的数据持久化模型（非关系型数据库）， 新的技术、方法、工具（node. js，DevOps，CI），新的设计理念（云服务架构设计理念 IaaS，PaaS及 SaaS）等等。这一切都催生了新的架构设计风格–微服务架构的出现。

微服务是一种架构风格，各个微服务之间是高内聚低耦合的。每个微服务之间有清晰的功能边界，仅关注于完成一件任务。一个大型应用系统可以由一个或多个微服务组成[1]，系统中的各个微服务可被独立部署。

微服务的概念源于2014年3月Martin Fowler所写的一篇文章“Microservices” (<http://martinfowler.com/articles/microservices.html)> [2]，整体式架构与微服务架构如图1-1对比所示：



图1-1 整体式架构和微服务架构

其中，对应用组件封装的方式是整体架构与微服务架构的主要差异，微服务架构将相关联的业务逻辑及数据放在一起形成独立的边界，其目的是能在不影响其他应用组件(微服务)的情况下更快地交付并推出市场。

本文在微服务的架构体系下，对交互式应用的服务端设计进行研究，主要内容包括应用基本功能拆分成不同的微服务模块，并对微服务模块进行具体设计。

## 研究内容与研究意义

本文主要研究，基于微服务架构下，交互式应用的服务端设计与实现。

交互型应用系统是指以与用户交互为主要目的而开发的应用系统，例如微信、微博、各种用户应用等等。在微服务的架构下，服务端的设计偏向于：

1）通过服务实现应用的组件化。

2）选取合适的工具实现服务（例如不同的开发语言）。

3）进行分散式数据管理，支持多样性持久化方案。

本论文针对使用微服务架构进行交互式应用开发，设计了基于微服务架构的服务端方案，满足交互式应用的高并发请求，具有良好的扩展性等业务及性能要求。

论文具体内容为：针对交互式应用的业务需求，拆分出基本的微服务模块，并对基础微服务模块进行设计与实现，以满足应用的查询，提交，交互等基本功能。在此基础上，对微服务模块选取了合适的服务端技术，包括不同语言上的选取，数据持久化的方案选取等。以满足交互式应用高并发请求、业务良好扩展性为目的，对数据存储及服务功能逻辑进行详细设计与实现。

综上所述，本论文设计了一种基于微服务架构下，满足交互式应用的高并发高可用的服务端架构及相应微服务模块，提供了灵活数据存储方案，充分地使用微服务架构本身的优势。

## 互联网应用架构发展现状

在应用开发的早期极端比较流行的是整体式架构，这种架构的主要特点是开发团队所有成员在同一个代码环境中开发，功能实现的代码集中在一个发布包中进行集中部署和发布。这种应用架构的弊端最早在大型企业和互联网行业中呈现，这些公司都遭遇到了复杂应用的开发维护成本变高，代码重复率增大，团队协作效率变差，系统可靠性变低，系统水平扩展困难，新功能上线周期变长等问题。随着应用不断的深化和业务数据几何级增长，业务用户对应用系统提出了越来越高的要求，传统的应用架构已无法满足开发需求。很多问题出露倪端，其中包括[2]：

1）传统的企业应用代码庞杂并且业务组件之间耦合程度非常高，早晨业务应用维护难度大，升级成本高

2）业务模块之间的循环依赖，不合理调用，冗长负责的业务流程等问题对新功能上线造成极大影响

3）系统功能组件出现无法恢复的故障是，整个节点处于不可用状态

4）扩展性方面，由于传统应用大多是单一部署，在这种模式下一个应用某些部分偏I/O密集型，某些部分却偏CPU密集型，但应用只部署在一台机器上，很难用单一硬件来满足应用各部分对硬件资源的不同要求，造成无法高效的应对多用户高并发场景等问题[3]。

接着出现的SOA（Service-oriented Architecture）架构和SOAP技术的发展在一定程度上解决了代码之间过度耦合的问题[2]。SOA架构提倡将复杂的应用程序按照不同的、可重用的功能划分为不同的服务，服务是一个粗粒度并且可被发现的软件实体。并且随着Java编程语言、J2EE应用服务器市场的发展，以及WebService、SOAP等技术的出现，SOA得到了进一步的发展[4]。总的来说，SOA面向服务体系结构重点在于设计一系列松耦合、粗粒度的、可被发现的服务，并提供服务的高可复用性、扩展性和可用性[5]

近两年随着Docker容器[6]技术的产生和发展，微服务架构模式逐渐在软件开发领域变得人起来。微服务的概念和SOA有相似之处，Martin Fowler在他的一篇文章中对微服务概念进行了一段阐述[7]：微服务架构在边界上，提出业务垂直解耦，服务之间互不干涉，产品非项目，谁开发谁维护。在组件层面，提出强壮服务框架，进程内有基础组件支撑，强化终端，弱化通道。在服务治理上，提出基础设施自动化，去中心化治理，分散的数据管理。

云计算技术的蓬勃发展，不仅促进了计算机硬件体系结构的发展，也引发了软件使用方式上的变革。资源服务化的思想日益普及，呈现出一切皆为服务的趋势，服务成为云计算的本质和核心概念[7]，以IasS，PaaS和SaaS为代表的服务模型已经得到了广泛使用和实践，以Docker技术为代表的容器化微服务技术逐渐渗透到云计算的各个层面，系统从开发，部署到运维整个过程都可以服务化。云计算架构设计理念的发展，以及docker容器相关技术的发展，使得开发者可以充分利用微服务架构提出的独立部署，快速交付，动态扩容等优势来来部署构建高扩展高可性能高可用的应用服务。

对比传统的整体式架构和微服务架构：整体式架构中多个功能模块集中运行与同一进程，并一起打包部署和水平拓展，以支持高性能，而在微服务架构中，单一的微服务是各自运行于自身独立的进程，应用程序可以自由扩展使用独立的微服务来组合完成某一具体的功能[3]。

对比微服务架构和SOA架构可以发现有一些相似之处，比如都提倡采用划分功能独立的服务。但是SOA在服务里粒度，实现方式，服务部署等方面和微服务有不同之处。

SOA架构偏向于静态的，企业级大型应用；服务包含的功能多，力度大；通信机制采用SOAP，ESB等服务总线，重量级；服务部署在统一平台，实现技术有J2EE/EJB等[2]。

微服务架构适用于快速迭代，快速交付互联网应用，服务功能单一，粒度细，通信机制有Restful、RPC等轻量级通信，部署方式为独立部署，相关技术有Docker技术等。

综上，SOA架构偏向静态，企业级大型应用，微服务架构则适用于快速迭代交付的互联网应用。

为了解决本小节初提到的应用开发中存在的问题，本文在采用了微服务框架，在基于微服务框架基础上，将开发中会高度耦合的功能分解到各个离散的微服务中，以实现对应用系统的解耦，并对服务的具体实现进行设计，以保障高并发情况的下的良好处理性能。

## 各章安排

第一章 绪论

从微服务架构的发展与互联网应用架构演变的过程介绍了本文的研究背景，继而提出论文的主要研究内容及研究意义，并对互联网应用架构和技术现状进行了介绍。

第二章 Web服务端关键技术研究

本章首先探讨了微服务架构的设计原则，继而对服务端架构开发所需要的具体技术，例如开发语言，开发框架，数据存储，缓存，消息队列等基本技术进行介绍

第三章 高可用的交互式应用服务端框架设计

该章在基于微服务架构的基础上，对整体的服务端架构进行了设计与分析，并分析探讨了交互式应用的基本功能，并针对功能的具体实现进行了详细设计，包括数据的存储设计，缓存设计等。

第四章 重要模块及接口详细设计

该章描述了查询服务，提交服务，邀请服务这三个微服务模块的功能详细设计及实现过程，结合上一章的架构及存储设计方案，进行了具体实现上的探讨，包括事件类型的定义，数据库表的实现，存储的实现，接口说明等。

第五章 测试数据与实验结论

该章对我们实现的交互式应用中的关键模块功能模块进行了测试和分析。

## 小结

微服务体系架构，描述了一种将软件应用程序设计成可独立部署的服务套件的架构方式，这种架构风格在围绕业务能力，自动化部署，以及对语言、数据的分散控制等方面，具有很好的灵活性和延展性。本文在微服务架构的基础上，将交互式应用的基础功能拆分成微服务模块，并进行具体设计与实现。

1. 项目关键技术分析

本章主要介绍微服务架构的设计原则，以及基于微服务架构的Web服务端关键技术：包括Restful API接口设计，开发语言的介绍，数据持久化技术，异步消息队列等。

## 微服务架构设计原则

微服务架构通常有以下的通用特性[7]。

1)单一功能原则

以服务的方式实现组件化 (Componentizationvia Services)：微服务架构中，把组件定义为可以独立替换以及升级的单元，在架构设计中，通常将通用的模块以服务的方式抽离出来，以微服务模块的形式独立部署升级。

2）轻量级通信

智能端点与管道扁平化(Smartendpoints and dumb pipes)：微服务架构主张将组件之间通信的逻辑放在组件端点处，而非存放于组件中，并且通信机制应该尽量简单及松耦合。常用的通信协议有Restful HTTP协议、RPC Thrift协议等，也可以通过仅提供消息路由功能的轻量级异步机制来实现微服务架构的通信。

3）分散的数据管理

分散的数据管理(DecentralizedData Management)：微服务架构倡导采用多样性持久化(关系型数据库、非关系型数据库)的方法。微服务的设计思想允许服务设定自己的数据库模式，无论是同一个数据库的不同实例，还是不同的数据库系统，允许不同微服务采取不同的持久化技术。

4）独立部署

基础设施自动化(InfrastructureAutomation)：云和AWS的的发展，docker技术的出现，降低了微服务构建部署的复杂性。许多产品的研发团队采用微服务架构去进行产品开发，集成CI进行持续交付。

## Web服务端关键技术分析

### **Restful**

REST（Representational State Transfer）是网络系统架构模式，同类可对比于 SOAP（ Simple Object Access Protocol，简单对象访问协议）。 REST倾向于用更加简单轻量的方法设计和实现对URL的处理及 Payload的编码。REST并没有一个明确的标准，而更像是一种设计的风格[4]。

Restful架构上的特性有[8]：

1. 实现上多为组件上的交互，用户感知好，网络效率高；
2. 支持大量组件之间的交互，交互之间无状态，具有标准的方法和格式以便于语义明确及交互信息，交互的信息是可缓存的；
3. 接口统一；
4. 组件的可替换性；
5. 服务端组件的可视化交流；
6. 可移植，便于涵盖数据的代码迁移 ；
7. 可靠，可抵御系统级别的错误：包括来自组件的错误，链接错误，数据错误。

Restful原则的具体实现可体现在Restful API风格的设计上。

我们将文本、图片、服务都作为资源[9]，以URL作为资源的地址提供给获取方，而资源的展现形式则可以是HTML，Json等方式。 当访问一个网站时，代表着请求端与服务端资源互动的过程，数据及状态都储存在服务端。 HTTP/HTTPS是无状态协议，在协议中为请求端提供了基本操作资源及状态的动词：比如GET用来获取资源，POST用来创建资源，DELETE用来删除资源，PUT用来更新资源。 在请求端操作动词调动资源后，服务端会返回相应的状态码来表示结果，例如200 OK - [GET]：服务器成功返回用户请求的数据；400 INVALID REQUEST - [POST/PUT/PATCH]：用户发出的请求有错误，服务器没有进行新建或修改数据的操作。

基于Restful API的设计风格下，我们可以设计出类似下面的接口[10]。

<https://api.example.com/v1/zoos>；

<https://api.example.com/v1/animals>；

<https://api.example.com/v1/employees>。

对应的http动词操作：

GET /zoos/ID/animals：列出某个指定动物园的所有动物；

POST /zoos：新建一个动物园；

PUT /zoos/ID：更新某个指定动物园的信息（提供该动物园的全部信息）；

DELETE /zoos/ID/animals/ID：删除某个指定动物园的指定动物。

### **RPC Thrift**

RPC是指Remote Procedure Call Protocol。

我们常说的远程方法调用，就是像调用本地方法一样调用远程方法，通信协议大多采用二进制方式。比如服务A想要调用服务B上的某个方法/函数，使用方可以忽略底层的传输层的细节，专注于方法的使用。就像调用一个本地函数，使用十分便捷，不需要关心接口的URL，校验规则，返回值解析等过程。

本地和远程调用差异在于[11]：

1）本地调用一定会执行，而远程调用则不一定，调用消息可能因为网络原因并未发送到服务方。

2）本地调用只会抛出接口声明的异常，而远程调用还会抛出RPC框架运行时的其他异常。

3）本地调用和远程调用的性能可能差距很大，这取决于RPC固有消耗所占的比重。本地调用的固有消耗是纳秒级，而RPC的固有消耗是在毫秒级。那么对于过于轻量的计算任务就并不合适导出远程接口由独立的进程提供服务，只有花在计算任务上时间远远高于RPC的固有消耗才值得导出为远程接口提供服务。

服务化的一个好处就是，不限定服务的提供方使用什么技术选型，能够实现大公司跨团队的技术解耦。 如果没有统一的服务框架，RPC框架，各个团队的服务提供方就需要各自实现一套序列化、反序列化、网络框架、连接池、收发线程、超时处理、状态机等“业务之外”的重复技术劳动，造成整体的低效[12]。所以，统一RPC框架把上述“业务之外”的技术劳动统一处理，是服务化首要解决的问题。

与此同时，在微服务架构中，各个服务之间可能千差万别，Restful接口更加灵活，如果使用RPC则会有很多约束。

针对实际的使用场景，Restful更多用于对外提供的服务，更加规范、通用、易扩展、已维护，具有较高的安全性(HTTPS)。RPC更多用于对内提供的服务，尤其适用于需要进行大量数据交互的服务(PRC中的Thrift框架提供高效的压缩协议，交互更加简洁，吞吐量更大)、高频率交互的服务(可以考虑建立TCP长连接)。现有的RPC框架有很多，本文中涉及的RPC框架是Thrift框架。Thrift是Facebook提出的开源的跨平台的RPC框架[13]。

在微服务的框架性下，接口调用服务，以及微服务之间的互相调用都是通过RPC调用，而对外的接口调用，比如用户读取某个URL则是HTTP/HTTPS调用。

### **Python语言及框架**

Python是一种面向对象，直译式的计算机语言程序，具有近二十年的发展历史。它包含了一组功能完备的标准库，能够轻松完成很多常见任务，语法简单，使用缩进来定义语句块[14]。

与Scheme、Ruby、Perl、Tcl等动态语言一样，Python具备垃圾回收功能，能够自动管理内存。它被当做脚本语言用于处理系统管理任务和网络程序编写，而且它也很适合完成各种高级任务。Python虚拟机本身几乎可以在所以作业系统中运行，一些工具比如Pyinstaller之类的工具也可以将Python源代码转换成可以脱离Python解释器运行的程序。

Python是一个高层次的结合了解释性、编译性、互动性和面向对象的脚本语言。Python的设计具有很强的可读性。由于Python是一种解释型的脚本语言，开发效率高，所以非常适合用来做Web开发。Python有多种Web开发框架，例如Django，Flask等，同时也有很多成熟的模板技术。

本文提到的交互式应用选取Flask作为Python Web开发选用的框架。

相对比Django，Flask的ORM入侵并不严重。它更倾向于一个Mirco Web 框架。Flask虽然是个轻量级框架，但并不意味着要将整个应用在一个Python文件中实现（虽然是可以实现的）。Flask微小，但是很功能健全。框架中的“微”意味着Flask保持核心简洁并且容易扩展，它并不为用户在使用上做太多决策上的先入为主的判断，比如应该使用哪一种数据库等等，与此同时Flask本身的模板引擎也容易替换，一切都由开发来掌握[15]。

Flask在默认情况下不包括数据库层抽象，表单验证等这些已有库可以完成的功能，同时它支持用扩展应用来添加功能，Flask支持众多扩展，包括数据库集成，开放认证，表单认证等功能。Flask在微小的同时，有扩展功能的支持下完全可以完成需求繁复的生产需求。

随着代码库的增大，使用开发者依然对项目有着整体的设计决策权。Flask不包括数据层，所以开发者可以在SQLAlchemy 或其他数据库工具中实现更高级的模式，酌情引入非关系型数据持久化。Werkzeug是Python的WSGI规范的实用函数库，Jinja2是基于Python的模板引擎，在这两者的扩展支持下，Flask可以完成Python Web服务端的开发工作，除此之外还有一些通用的标准库包，比如说Logging。Flask的设计思想是为应用建立良好的基础，其他的由开发者来选择扩展。

我们选取Flask作为交互式应用的Web开发框架，Flask框架轻便，功能健全，我们引入SQLALchemy数据库工具实现数据的持久化，引入Logging标准库来做日志记录。

### **数据持久化**

关系型数据库在一些数据敏感的应用中性能表现不佳，例如高流量网站的网页服务，为巨量文档创建索引等。关系型数据库主要被调整用于执行规模小但读写频繁，或者大量极少写访问的事务。常见的关系型数据库有MySQL等[16]。

Nosql最普遍的解释是“非关联型的”，强调Key-Value存储和文档数据库的优点，而不是单纯的反对RDBMS（关系型数据库管理系统）。

常见的Nosql数据库有以下类型。

表2-1 NoSQL数据库类型图

|  |  |  |
| --- | --- | --- |
| 数据库类型 | 例子 | 使用场景 |
| 键值数据库 | Redis，Memcache等 | 存储交互式数据 |
| 文档数据库 | MongoDB等 | 事件日志，内容管理、分析；  不适用于复杂查询 |
| 列式数据库 | HBase，Hypertable等 | 事件日志，计数器等 |
| 图形数据库 | [Neo4j](https://zh.wikipedia.org/w/index.php?title=Neo4j&action=edit&redlink=1)，[DEX](https://zh.wikipedia.org/w/index.php?title=DEX_(%E5%9B%BE%E6%95%B0%E6%8D%AE%E5%BA%93)&action=edit&redlink=1)等 | 社交网络，基于位置的服务；  不适合某些类型的批量更新 |

**2.2.4.1 MySQL**

数据库是数据的集合，可以是存储器上的文件集合以及内存数据的集合。

MySQL数据库是数据库管理系统，它负责存储数据，并且提供查询和更新数据，提供查询、更新数据库中数据的功能；是一款关系型数据库，提供SQL查询，子查询，存储，触发器，视图，索引，事务，锁，外键约束等功能[17]。

MySQL作为一款开源数据库，广泛的应用于各大公司，是非常流行的关系型数据库。

MySQL提供索引来帮助加快查询。索引的类型如下：

1. 聚集索引：

表数据按照索引的顺序来存储的。聚集索引是⼀种稀疏索引，数据⻚上⼀级的索引⻚存储的是⻚指针，⽽不是⾏指针。

1. ⾮聚集索引：表数据存储顺序与索引顺序⽆关。

非聚集索引则是密集索引，在数据⻚的上⼀级索引⻚它为每⼀个数据⾏存储⼀条索引记录。

1. 优点：

A）索引可以避免全表扫描，变为少量的索引页扫描的数据页扫描；

B）对于⾮聚集索引，有些查询只需查询索引，无需访问数据页；

C）由于索引记录仅包含索引字段值以及指针，索引实体⽐真实的数据⾏要⼩许多，索引⻚相较数据⻚来说要密集许多。⼀个索引⻚可以存储数量更多的索引记录，这意味着在索引中查找时在I/O上占很⼤的优势。

1. 缺点：

索引占据更多的磁盘空间，很多时候索引甚至比数据本身还要大。其次，当建立索引的字段发生更新时，会引发索引本身的更新，也将产生不少的计算量，会导致更新数据的性能下降。

MySQL的引擎是Innodb。Innodb支持事务，即Rollback或者Commit操作。Innodb实现事务的方式是当有事务提交时，预先写入到日志的缓冲区，然后当事务日志写入磁盘后，才更新实际的数据和索引，在这里Innodb选择事务日志何时写入磁盘决定了事务的类型。

除了事务之外，Innodb还支持行锁。查询的开销主要分为两部分，一部分是查询本身的计算时间，另外一部分是查询等待的时间。查询本身的计算时间和索引的建立息息相关，查询等待的时候则和锁相关，减少表锁等待的时间，将大大地提高读取的性能。

**2.2.4.2 NoSQL**

NoSQL是一种区别与关系型数据库（RDBMS）的新型数据库。NoSQL是非关系型数据存储的广义定义。

1. 易扩展，NoSQL数据库种类繁多，但是一个共同的特点都是去掉关系数据库的关系型特性。数据之间无关系，这样就非常容易扩展，也无形之间，在架构的层面上带来了可扩展的能力。

2. 高性能，尤其在大数据量下，同样表现优秀。这得益于它的无关系性，数据库的结构简单。

3. 灵活的数据模型，NoSQL无需事先为要存储的数据建立字段，随时可以存储自定义的数据格式。

Redis是各大公司通常采用的Nosql Key-Value类型存储技术。

Redis使用[ANSI C](https://zh.wikipedia.org/wiki/ANSI_C)编写的[开源](https://zh.wikipedia.org/wiki/%E5%BC%80%E6%BA%90)、支持[网络](https://zh.wikipedia.org/wiki/%E7%94%B5%E8%84%91%E7%BD%91%E7%BB%9C)、基于[内存](https://zh.wikipedia.org/wiki/%E5%86%85%E5%AD%98)、可选[持久性](https://zh.wikipedia.org/w/index.php?title=%E6%8C%81%E4%B9%85%E6%80%A7_(%E6%95%B0%E6%8D%AE%E5%BA%93)&action=edit&redlink=1)的[键值对存储数据库](https://zh.wikipedia.org/w/index.php?title=%E9%94%AE%E5%80%BC-%E5%80%BC%E6%95%B0%E6%8D%AE%E5%BA%93&action=edit&redlink=1)。从2015年6月开始，Redis的开发由[Redis Labs](https://zh.wikipedia.org/w/index.php?title=Redis_Labs&action=edit&redlink=1)赞助，而2013年5月至2015年6月期间，其开发由[Pivotal](https://zh.wikipedia.org/wiki/Pivotal)赞助。在2013年5月之前，其开发由[VMware](https://zh.wikipedia.org/wiki/VMware)赞助。根据月度排行网站DB-Engines.com的数据显示，Redis是最流行的键值对存储数据库。

Redis是一项介于内存式缓存技术与关系型数据库中间的技术补充，具有丰富的键值数据类型：String，Hash，List，Set 及 Zset(Sorted Set)。

### **缓存技术**

在Web开发中，服务端有不同层次的缓存技术。

1) 页面缓存：缓存页面及HTTP返回结果，减少Render的开销。静态网页或变化很小的网页或文件常用用来替代静态化 。但随着CDN的流行，已经不常用了。

2）模版和脚本解析缓存，减少Parser模板和脚本的开销，减少CPU开销。

3）数据结果缓存：对实时性有一定影响，节省大部分CPU开销。

4）Innodb Buffer 缓存，缓存DB数据结构的访问，加快数据库访问。

5）DB Item 缓存，加快表的查询，节省CPU及IO。

6）DB外层缓存 以防高并发请求将数据库打挂等等。

本文着重探讨了在数据库外层，增加一层Redis作为缓存，以防止高并发查询落在数据库上，将数据库卡死。

Redis作为缓存时可以设置数据缓存时间[18]。Redis作为Key-Value结构存储，有自己灵活的数据结构，键值类型除了常见的String，还有Hash，List，Set，Sorted set。键值的数据类型决定了该键值支持的操作。Redis支持诸如列表、集合或有序集合的交集、并集、查集等高级原子操作；同时，如果键值的类型是普通数字，Redis则提供自增等原子操作。

对比常做缓存的Memcache，Redis支持数据的备份，支持数据的持久化，可以将内存中的数据保持在磁盘中，重启的时候可以再次加载进行使用。同时Redis也支持事务，Redis和Memcache最大的区别是并不是所有的数据都一直存储在内存中的，这种特性使得Redis可以保持超过其机器本身内存大小的数据。除此之外，Redis将数据存储在内存中，这是Redis读写速度快的重要原因之一。

### **消息队列**

对于业务来说，有时并不需要立即处理消息。消息队列提供了异步处理机制，允许把一个消息放到消息队队列中，但是并不即刻处理他。常用的消息队列有Kafka，Nsq等。

Kafka是由LinkedIn开发的分布式消息系统，由于其分布式及高吞吐率的特点，现在被多家不同类型公司作为多种类型数据管道和消息系统使用[19]。

Kafka集群包含一个或多个服务器，这种服务器被称为Broker。每一条发布到Kafka的信息都自带一个Topic，一个Topic的消息可以保存在一个或者多个服务器（Broker）上，用户无需关心消息的存储，只要指定Topic即可发布消息。物理上把Topic分成一个或多个Partition，每个Partition在物理上对应一个文件夹，下面存储这个Partition的所有消息和索引文件。Kafka保证一个Partition内消息的有序性。

Nsq是一个基于GO语言的分布式实时消息系统，可用于大规模系统中的实时消息服务，设计目标是为在分布式环境下运行去中心化服务提供强大的基础架构。Nsq具有分布式，取中心化的拓扑结构，该结构具有无单点故障，故障容错，高可能用以及保证消息的可靠传递的特征。

Nsq消息主要保存在内存，它在处理消息上更快，同时也支持可持久化到硬盘。Nsq保证消息至少传递一次，以确保消息可以成功送达，收到的消息是无序的，实现了松散订购。

## 小结

本章主要介绍微服务架构的设计原则，包括单一，轻量级，分散的数据管理和独立部署。介绍基于微服务架构的Web服务端关键技术：包括Restful API框架，RPC远程调用技术，开发语言，数据持久化技术，异步消息队列等。并针对于业务特性，对不同的服务端开发技术的相关功能点进行介绍。

1. 服务端架构及模块总体设计

## 服务端架构设计需求分析

交互式应用的需求分析如下：

交互式应用的交互部分功能设计，可以首要分析用户的应用场景。

用户在浏览一个问答的内容时，需要浏览问题，浏览问题相关答案，浏览问题的提问用户等等。

当看到感兴趣的话题的时候，用户可以编写自己的问题，回答去提交。

用户的查询场景，查询问题相关的信息，比如问题的名称，问题的描述，问题的图片，视频，问题的回答数目，回答的内容等。

用户的提交场景，提交一个问题，提交一个答案。

除上述功能性阐述之外，交互式应用对服务架构性能也有一定要求。

1. 支持高并发数据请求

交互式应用的数据请求量并发量高，需要服务端有高承载力，这对数据存储设计，对业务逻辑的处理都有一定要求。

2. 业务需求多，架构支持可扩展性

交互式应用对服务架构的可扩展性有一定要求，微服务的架构模式很好地满足了这个需求，各个微服务模块之间高内聚低耦合，不同微服务模块开发交付互不影响。微服务架构良好地支持了服务端开发的扩展性，是更加灵活的服务端架构。

### **微服务拆分设计**

微服务模块设计有以下几个考虑方面：

1.松耦合和高内聚

如果做到了服务之间的松耦合，那么修改一个服务就不需要修改另一个服务。这也符合微服务最重要的一点：能够独立修改及部署单个服务而不需要修改系统的其他部分。一个松耦合的服务应该尽可能少地知道与之协作的那些服务的信息。高内聚则是要求把相关的行为聚集在一起，把不相关的行为放在别处。

2.明确接口

随着业务需求的多样化，微服务模块应具备可拓展性，服务界限划分不清，功能考虑不完整，可能会导致不得不频繁地更改服务之间协作以及业务层面接口更改。接口的更改成本很高，所以在微服务最初设计时，预留出相应扩展字段。

所以花时间了解应用是做什么的，然后尝试识别出清晰的模块边界，这是划分服务之前所必须做的；了解模块应提供的的功能，确定合理的接口字段，是实现服务前必须做的。

在业务需求分析之前，先进行用户场景分析：

1.普通浏览用户：浏览问题，赞同或者关心此问题，去查看回答。

2.有生产内容习惯的潜在种子用户：浏览问题，赞同回答，或者关注此问题或者回答问题；继续浏览其他问题，回答其他问题。

3.专业用户：主动浏览问题后回答问题；被邀请其他用户邀请回答擅长领域的问题后回答问题。

从上述的用户场景中，可以提炼出通用需求为：查询相关（浏览问题，浏览回答）；提交相关（提交问题，提交答案）；用户相关（用户之间的相互邀请）。

由此，我们确立了以下微服务模块：

1. 基础服务：查询服务；

2. 提交服务：提交服务；

3. 用户服务：邀请服务。

## 整体架构分析

### **交互式应用微服务架构设计**

服务端总体架构如图3-1所示：



图3-1 服务端架构设计

服务端架构设计如图3-1所示，API层通常是业务层面调用接口方，比如Web站，比如APP端等；服务层中包含了交互式应用中的基础、通用服务模块，将通用功能拆成单个低耦合的服务模块，提供服务接口调用给API层，将业务需求简化为从服务中读取数据，提交数据，无需考虑后端存储，异步处理等；存储层包括数据库和缓存模块；消息队列接受服务层和API层的调用，由Kafka和Nsq两种消息队列，其中Kafka保持数据的有序性，Nsq不保证数据有序；消息队列接到消息后，由Worker异步接受处理。

微服务的架构模型中每个服务都比较简单，只关注于一个业务功能；与此同时微服务架构方式是松耦合的，可以提供更高的灵活性。

### **存储设计**

存储设计主要分析在对应的业务需求场景下，对数据库的架构及选型。包括何时使用关系型数据库，何时使用非关系型数据库；在使用关系型数据库存储时，对数据库表结构的设计，字段的具体定义；并且针对相应的查询场景进行数据库的优化，比如增加索引，增加缓存技术。

关系型数据库的事务使得存储数据具有强一致特性，有着不同的四种事务隔离级别，包括Serializable序列化，Repeatable可重复读，ReadCommitted读已提交和ReadUncommitted读未提交。级别越高，执行效率对应就越低，像是Serializable这种级别，在数据库中是以锁表的形式实现的，其他线程的调用只能在锁任务完成后进行。MySQL数据库支持上述四种隔离级别，它默认的隔离级别是Repeatable Read可重复读。

关系型数据库的数据强一致特性，满足大型的银行系统对数据的严格的要求。但在互联网应用软件特别是SNS交互应用的设计中，有时对数据的强一致性要求的并不是那样高。比如说，用户上一秒获取的阅读数和下一秒的阅读数的增减，并不影响用户对软件的使用。但通常交互式应用软件，需要支持高并发高流量请求的设计。交互式应用通常会高并发地请求某些数据，对于传统的关系型数据库，高并发查询就是相应的瓶颈，这种时候我们则会选择用非关系型数据库Nosql进行存储。常用的非关系型数据库有Redis，SpringDB等。

除此之外，某些场景考虑到：采用Key-Value存储方案主要考虑是数据量较大，采用MySQL存储对磁盘空间压力较大，且MySQL拆分扩展成本较高，故采用比较容易扩展的Key-Value存储系统，提高扩展性，以满足不同业务需求。

我们根据业务场景的不同选取两种存储设计，一种是关系型数据库MySQL的存储设计，一种是非关系型数据库Redis的存储设计。

在对于对并发读写能力要求极高，一致性要求不强的场景下，比如展示某个回答的阅读数（用户A看到的数目和用户B看到的数目不同完全不会影响用户体验），会选取非关系型数据库Redis来存储。

在对于一串相对有联系的数据，比如一条邀请数据（我们会记录邀请的问题ID，邀请人ID，被邀请人ID，邀请时间，邀请结果等有联系的信息），会选取关系型数据库MySQL来存储。

问题/答案等数据因为存在冷热，因此为了加快访问，加了层Redis的缓存。除此之外，对于业务上的一些需要维护的计数，时间戳等，有需要缓存，有需要持久化，因此使用Redis存储。

### **缓存设计**

传统的关系型数据库在支持高并发读写请求时，磁盘I/O是很大的瓶颈；作为交互式应用，并发读需求很高，对于关系型数据库来说在海量数据的表中查询，效率是非常低的。除此之外，MySQL拆分成本比较高，对数据库的系统升级和扩展相对麻烦，而采用Key-Value形式的非关系型数据存储，读取效率为O(1)，并且可提升数据扩展性。

考虑到以上问题，在除加入非关系型数据库外，我们在设计关系型数据库时，会在访问数据库之前，加入一层缓存。Redis是Key-Value内存型数据库，具有丰富的键值数据结构，可以使用Redis作为中间层缓存常用查询数据数据。

与此同时，也需要设计缓存的回源逻辑。缓存是源数据的子集及子集的聚合，未必存储所有的数据，当数据不存在的时候需要回源；缓存数据过期（超过过期时间或者源数据更新）时需要回源更新数据，回源逻辑如图3-2所示。



图3-2 缓存回源设计

### **消息队列**

对于业务来说，有时并不需要立即处理消息。消息队列提供了异步处理机制，允许把一个消息放到消息队队列中，但是并不即刻处理他。消息队列在业务处理过程中增加一层消息系统，允许独立扩展修改消息系统两边的处理方逻辑，解耦处理过程。因此，增大消息入队数量和提高处理频率是便于调节的，无需改动代码，只需增加处理过程。这使得业务有良好的灵活性和峰值处理能力，使得服务部署无需以峰值访问为标准来设定资源，消息队列帮助关键组件顶住突发的访问压力，不会因为突发的请求而导致服务崩溃。

不同的消息队列各具特色，本文以Kafka和Nsq这两种平台为例，进行讨论。

Kafka是一个分布式的流媒体平台。它通常应用于构建实时消息流程序，构建数据流之间的管道。它以Topic的方式来记录信息流，每个Topic有0-多个消费者。对于每个Topic，Kafka维护了一个分区日志，每个分区是都是有序的记录序列，每个序列都会分配一个序列ID号，称为Offset。Kafka做为消息系统，允许把数据划分成多个Partition，为多个消费者消费，并且保证每个Partition的序列都是有序的。

NSQ是基于GO语言开发的实时的分布式消息平台，具有分布式和去中心化拓扑结构，该结构具有无单点故障、故障容错、高可用性以及能够保证消息的可靠传递的特征。Nsq消息主要保存在内存，它在处理消息上更快，同时也支持可持久化到硬盘。Nsq保证消息至少传递一次，以确保消息可以成功送达，收到的消息是无序的，实现了松散订购。

对比Nsq和Kafka，如表3-1所示：

表3-1 NSQ vs Kafka

|  |  |  |
| --- | --- | --- |
| 特点 | NSQ | Kafka |
| 语言 | GO | Scala |
| 消息存储 | 内存，磁盘 | 磁盘，文件方式 |
| 顺序保证 | 不保证有序 | 有序 |

至于在实际应用中，两种消息队列如何选择，要根据业务需求和消息队列的特性做出合理选择。在我们的服务端设计中，NSQ和Kafka两种消息队列都在使用中：

在一些应用场景下，数据的处理顺序很重要，比如提交类的事件，就需要保证严格的时序性。比如用户先提交一个问题，随后又立即删除，业务方在处理信息是根据发送的消息做相关处理，如果删除问题的消息先发送，那业务方处理将出现错误。对于这种需要严格保证消息顺序的消息，我们选取Kafka来保证提交类消息的有序。

对一些时序性要求不强的事件，比如统计用户行为的消息，只需对消息进行记录，无关时序上的逻辑处理，我们就会选取Nsq作为消息队列，因为Nsq消息主要保存在内存上，在处理消息上更快。

综上，本文设计的微服务模块有查询服务，提交服务，邀请服务。其中涉及到异步处理的提交事件和邀请事件都对时序有严格要求，故我们选取Kafka作为消息队列保证消息的有序性。

于此同时，我们需要有个统一的收集所有提交类事件，并告知下游。我们把所有的消息都收集到Kafka，针对不同下游所需要的数据，实现不同的消费者及集群，实现消息的异步处理。

## 微服务模块设计

在3.1小节中，我们针对用户场景，提炼出来三个微服务模块，分别是：

1. 基础服务：提供查询服务；

2. 提交服务：提供提交服务；

3. 用户服务：提供邀请服务。

服务围绕业务功能构建，每个服务只关注于一个业务功能，服务独立部署，不同服务针对服务业务特性可以选取不同的语言开发，并使用不同的数据存储技术。下述小节将介绍每个微服务模块的总体设计。

### **查询服务总体设计**

用户在浏览应用内容时，直观看到问题及其相关描述信息，继而去关注问题的回答信息，以及提问者回答者的信息。能成功浏览这些信息，需要服务端为前端提供数据打包数据，我们把提供数据的功能抽象出来，作为单独的查询服务，在具体的业务需求中，通过查询服务来获取必要数据，再去根据业务需求打包数据，完成业务需求。

查询服务主要为用户提供查询问题数据，查询回答数据等基础功能。查询服务的大部分流量来自于以下两种查询场景：

1.根据问题Qid列表，查询问题信息；

2.根据回答Ansid列表，查询回答信息。

查询服务架构设计如图3-3所示：



图3-3 查询服务架构设计

查询服务提供接口给调用方，调用方传入问题ID等必要信息来查询问题详细信息，服务对数据进行校验后，去数据存储部分查询数据。查询服务是一个高流量服务，承载高并发高负荷的请求，为了加速请求的返回速度，我们在查询数据库之前，先去查询缓存层，缓存层中缓存了问题、答案的信息。如果请求没有命中数据库，则再去数据库中查询信息，最后返回结果给调用方。

### **提交服务总体设计**

用户在浏览应用内容时，引发自己想提问的兴趣，于是撰写问题提交；或者看到有意思的问题，想去回答共同交流。产品的设计引发用户有产出的需求，服务端需要提供相应的功能。提交功能是需要服务端对用户提交的信息，进行符合业务需求的逻辑处理，并进行持久化存储，以及提供给用户后续更改的条件。能成功提交这些信息，需要服务端从前端获取数据处理数据存储数据，我们把处理数据、存储数据的功能抽象出来，作为单独的提交服务。在具体的业务需求中，通过调用提交服务来提交问题、答案信息，来完成业务需求。

提交服务主要为用户提供文章的提交问题，提交回答，修改问题，修改回答等基础功能。提交服务的大部分流量来自于以下两种场景：

1.用户创建问题 & 用户创建答案；

2.用户编辑问题 & 用户编辑答案。

提交服务架构设计如图3-4所示：



图3-4 提交服务架构设计

提交服务为调用方提供接口，调用方传入必要的信息（比如问题名称、描述等信息），服务对信息进行校验及过滤后，写入相应的数据库进行信息的持久化存储。于此同时，由于产品设计和业务要求，一些提交类操作并不要求实时处理，于是我们加入消息队列来对解耦消息提交和消息处理。提交类的操作对时序性有着严格的要求，用户提出问题又删除问题，如果删除问题的消息先被业务方接受到并进行处理则会出错。我们选取可以保证时序性的Kafka来作为消息队列，在根据业务需求设置不同种类的Worker来接受信息，完成消息的异步处理。

异步的提交机制很好的解耦了消息的提交和消息的处理，我们只需要增加异步事件的定义即可丰富提交事件的类型，对新业务的支持非常灵活；通过增加Worker即可增加处理量级而无需改动代码，架构本身良好的支持请求量级的大幅度增加。

### **邀请服务的总体设计**

用户在浏览应用内容时，看到感兴趣的问题，一可以自己产出回答，二可以邀请专业人士回答；同时自己提出一个问题后，为了尽快的获得回答，也可以邀请别人来回答自己的问题。用户既可以邀请别人，也会收到别人的邀请，邀请的这个功能促进了用户的产出，增加了用户之间的交流。能去实现成功邀请他人回答问题，需要服务端从前端获取数据处理数据，我们把处理邀请数据的功能抽象出来，作为单独的邀请服务，在具体的业务需求中，调用邀请服务来提交邀请信息，完成业务需求。

邀请服务主要为用户提供邀请其他用户回答问题的功能。邀请服务的大部分流量来自于以下两种场景：

1.用户创建问题时邀请别人回答问题；

2.用户浏览问题是邀请别人回答问题。

整体架构如图3-5所示：

****

图3-5 邀请服务架构设计

邀请也应用了提交事件系统。在用户邀请时，根据问题的审核通过与否状态来判断是即时邀请还是延后邀请。如果问题是正常状态则即时邀请，将邀请信息写入数据库中，如果问题是非正常状态，则将邀请信息发送异步事件给消息队列，并且由邀请相关的Worker接受事件来继续完成后续邀请。

## 模块对外提供接口

### **Json 服务返回数据**

Json是Jacascrpit的一个子集，采用了类似于C语言家族的一些习惯，是轻量级的数据交换语言。Json的数据格式与语言无关，但目前很多编程语言都支持Json数据格式的生成和解析。Json被广泛地应用于Web应用的开发，除此之外也应用于一些非关系型数据库作为其存储格式。

API接口调用服务时，服务以Json格式打包返回数据；在存储上，无论是缓存型的Redis还是持久性的Redis，我们多将数据用Json格式打包成字符串的形式，再进行存储，服务或者其他接口读取的时候再进行Json解打包来获取原数据。

### **Protobuf 对外接口提供**

GoogleProtocolBuffers（简称Protobuf）是由谷歌所主导的开源项目, Protobuf独立于语言和平台,能够实现分布式系统下的数据序列化和反序列化从而实现数据交换过程photobuf能够支持Java、C++和Python等多语言甚至跨语言的开发。

Protobuf—共有3种数据的限定修饰符，介绍如表3-2所示：

表3-2 Protobuf限定修饰符表

|  |  |
| --- | --- |
| 限定修饰符 | 描述 |
| required | 不可删除或者增加的字段，必须进行初始化 |
| optional | 可选字段，在发送时可以选择是否初始化 |
| repeated | 可重复的字段，其中可以包含多个元素，每个元素的基本特性和optional类型相同，可以看做是一个list |

微服务对外提供接口以Protobuf的格式展示。在第四章中的模块及接口设计中，可以看到每个服务对外的Protobuf格式的接口设计。

## 小结

本章对基于微服务的服务端整体架构进行介绍，根据微服务原则和业务分析，拆分出应有的功能服务模块，对查询服务，提交服务，邀请服务分别进行总体设计，选取相应的数据持久化方案及对应的缓存设计，以满足业务功能及性能需求。

1. 重要模块及接口详细设计

## 提交服务具体设计

### **提交类事件定义设计**

对于业务来说，有时并不需要立即处理消息。消息队列提供了异步处理机制，允许把一个消息放到消息队队列中，但是并不即刻处理它。考虑到提交类事件对时序的要求，我们选取Kafka作为消息队列实现异步处理，以保证消息的有序性。

我们需要统一收集提交类事件，并告知下游。我们把所有的消息，收集到消息队列中，针对不同下游所需要的数据，实现不同的Worker消费。

异步事件用于完成异步调用逻辑，异步事件分为两类：核心事件和非核心事件，核心事件代表了问题或者答案的相关字段在数据库中发生了改变，而非核心事件则包括了用户邀请等等各类业务事件。对于核心事件的定义原则，如下所示：

一个事件一般对应数据库中数据的一次变更（Insert、Update、Delete等）。事实上，消息消费者关心的并不是事件本身，而是某个字段或者某几个字段发生了怎样的改动。一次数据库状态的变更，一定可以用一个三元组来表示（Old\_Value, New\_Value, Extra\_Info），其中，Old\_Value表示状态变更前，答案或者问题在数据库中的各种状态属性的值，New\_Value表示状态变更后，答案或者问题在数据库中的各种状态属性的值，Extra\_Info则表示状态变更的其他额外信息，如事件发生的时间，事件操作的来源等等。

具体每个核心事件对应的数据库修改内容如表4-1所示：

表4-1 异步事件定义

|  |  |
| --- | --- |
| 事件名称 | 对应的数据库变动 |
| Create\_Answer | 回答表中插入了新数据 |
| Create\_Question | 问题的展示状态变为放出状态 |
| Delete\_Answer | 删除答案 |
| Delete\_Question | 删除问题 |
| Update\_Answer | 答案的内容、摘要字段变更 |
| Update\_Question | 问题名称、内容字段变更 |

我们为提交事件建立事件编号，提交事件具体类型定义及编号如表4-2所示：

表4-2 提交事件具体类型

|  |  |  |
| --- | --- | --- |
| 事件编号 | 事件名称 | 事件描述 |
| 1 | Create\_Question | 创建问题 |
| 2 | Delete\_Question | 删除问题 |
| 3 | Create\_Answer | 创建答案 |
| 4 | Delete\_Answer | 删除答案 |
| 5 | Update\_Answer | 更新答案 |
| 6 | Update\_Question | 更新问题 |

服务在发送异步消息的时候，参数中包含异步事件编号及其他消息内容，序列化后作为消息发送给消息队列Kafka，处理不同业务的Worker从Kafka中获取数据，根据异步事件类型来进行不同业务的逻辑处理。

### **提交服务接口设计**

提交服务主要有以下几种提交类型：

1.用户创建问题 & 用户创建答案；

2.用户编辑问题 & 用户编辑答案。

我们分别创建四个接口，用户创建问题接口为SubmitQuestion，用户修改问题接口为OperateQuestion，用户创建答案接口为SubmitAnswer，用户修改答案接口为OperateAnswer。

调用接口名为接口名+Request，返回接口名为接口名+Response。必要字段用required修饰，选填字段用optional修饰。

具体接口描述如下：

1. SubmitQuestion 提交问题

SubmitQuestion接口实现提交问题功能，接口字段设计如表4-3所示：

表4-3 提交问题接口字段

|  |  |
| --- | --- |
| 字段 | 字段描述 |
| Question\_Title | 问题的标题 |
| Question \_Content | 问题的描述 |
| User\_id | 用户登录id |
| User\_name | 用户名 |

具体接口代码如下，以protobuf形式展示：

struct SubmitQuestionRequest {  
1: required string Question Title; // 问题的标题  
2: optional string Question Content; // 问题的补充内容  
3: optional i64 User\_id; // 用户id，   
4: optional string User\_name; // 登录用户的用户名  
}

1. OperateQuestion 修改问题

OperateQuestion接口实现修改问题功能，接口字段设计如表4-4所示：

表4-4 修改问题接口字段

|  |  |
| --- | --- |
| 字段 | 字段描述 |
| Qid | 问题id |
| Operation\_Type | 操作类型 |
| User\_id | 操作的用户id |

具体接口代码如下，以protobuf形式展示：

struct OperateQuestionRequest {  
1: required i64 Qid; // 问题id  
2: required OperateType Operation\_Type; // 操作类型  
3: optional i64 User\_id; // 操作的用户id，  
}

1. SubmitAnswer 提交回答

SubmitAnswer接口实现提交回答功能，接口字段设计如表4-5所示：

表4-5 提交回答接口字段

|  |  |
| --- | --- |
| 字段 | 字段描述 |
| Qid | 问题id |
| User\_id | 提交答案的用户id |
| Answer\_Content | 提交的答案的内容 |

具体接口代码如下，以protobuf形式展示：

struct SubmitAnswerRequest{  
1: required i64 Qid; // 问题id  
3: required i64 User\_id; // 提交答案的用户id  
4: required string Answer\_Content; // 提交的答案的内容  
}

1. OperateAnswer 修改回答

OperateAnswer接口实现修改回答功能，接口字段设计如表4-6所示：

表4-6 修改回答接口字段

|  |  |
| --- | --- |
| 字段 | 字段描述 |
| Qid | 问题id |
| Ans\_id | 答案id |
| User\_id | 登录用户的uid |
| Operation\_Type | 操作类型 |

具体接口代码如下，以protobuf形式展示：

struct OperateAnswerRequest{  
1: required i64 Qid; // 问题id  
2: required i64 Ans\_id; // 答案id  
3: required i64 User\_id; // 用户的uid   
4: required OperateType Operation\_Type;  
}

接口具体实现以提交答案为例，首先调用方按照接口约束，传入Qid，User\_id，Answer\_Content字段信息；提交服务接受到调用信息后，从数据库中获取对应的信息（比如从问题的表中获取Qid对应的该问题相关信息），从其他服务中获取必要信息（RPC远程调用，比如从用户服务获取提问者信息）；对所获得的参数进入输入合法性检查；检查具体的业务策略（比如是否满足权限控制等）；提交数据，将数据写入数据库中；进行善后处理（例如删除该问题信息缓存，发送异步消息等）。

流程如图4-1所示：



图4-1 提交答案流程图

其中我们根据Useri\_d从用户服务中获取用户详情，并且将提交的内容进行转码，将内容转成UTF-8编码。然后我们会检查参数的有无、合法性，检查检查问题的状态：比如这个问题是否存在、是否是删除状态；检查提交回答的频率是否过快；检查内容是否符合规范：比如内容长度是否达标；检查这个问题用户之前是否提交过；检查用户的状态：比如用户是否有提问的权限，是否是一个封禁的用户；并且根据相应的产品策略对内容进行过滤，过滤不合法的词汇以及其他输入等等。

以上工作都处理完成后，我们将提交数据，提交数据是指写入回答表；除此之外，我们还将删除答案所对应的问题相关的缓存，并且向消息队列Kafka发送提交答案的消息，消息由worker接受做其他业务逻辑处理；如果消息发送不成功，则向我们用作备用的一个Redis添加记录，事后重复提交。其中事后重复提交的方式，则是不断的轮询Redis，尝试重新把信息插入Kafka。

在我们的异步事件的设计中，提交类的事件，后续全部由异步Worker来从Kafka的固定Topic中的Partition中读取信息来进行异步处理。不同类型的事件由不同类型的Worker来接收。在Worker处理中，一个Worker同时起多个进程处理异步事件。

我们以启动Worker 从Kafka里面的Submit这个Topic里面读消息， 通常一个Worker起10个进程 ，对应从消息队列Kafka的Submit 这个Topic下的10个Partitions里面对应读消息。除此之外，将Kafka是Offset Commit模式，每消费一个消息，Offset会自动下移动一位，在这种情况下，消息只被消费一次。

### **存储设计与实现**

查询服务主要依赖DB进行查询，主要涉及两张表：

1. 问题信息表Question：

主要存储问题相关的属性信息，包括问题qid，问题标题question\_title，问题内容question\_content，问题状态status，问题创建时间create\_time，问题修改时间modify\_time，问题文章扩展属性 extra 等字段。

2. 回答信息表Answer：：

存储回答相关的属性信息，包括回答ans\_id，回答对应问题qid，回答内容摘要answer\_abstract，回答状态status，回答内容answer\_content，回答创建时间create\_time，回答的用户user\_id, 回答的扩展属性extra 等字段。

问题表的相关字段设计如表4-7所示：

表4-7 问题表字段

|  |  |  |
| --- | --- | --- |
| 表字段名 | 数据类型 | 字段描述 |
| Qid | bigint | 问题id |
| question\_title | varchar | 问题标题 |
| question\_content | text | 问题描述 |
| Status | tinyint | 问题删除/展示状态 |
| user\_id | bigint | 发问题的用户id' |
| create\_time | timestamp | 问题创建时间 |
| modify\_time | timestamp | 修改时间' |
| Extra | text | 扩展属性 |

qid设计为bigint类型，可以容纳2^64个问题容量；question\_title设计为varchar变长类型，字长最大为400个汉字/英文字符；question\_content设计为text类型，最大存储65535字符；status设计为tinyint类型，存储问题的删除展示状态，0表示删除状态，1表示正常状态；user\_id设计为bigint，可以容纳2^64个用户容量；create\_time和modify\_time记录了问题的创建时间和修改时间，用timestamp数据类型存储，extra设计为text类型，用于存储问题的扩展字段，通常存储内容为Json格式。

主键和索引设计如下：

Innodb采用聚集索引（Clustered Index，又叫聚簇索引）来存储数据。聚集索引是一种索引组织形式，索引的键值逻辑顺序决定了表数据行的物理存储顺序。而非聚集索引则就是普通索引了，仅仅只是对数据列创建相应的索引，不影响整个表的物理存储顺序。简单理解就是聚集索引里面存储了数据。Innodb 中，有主键索引（Primary Key）的话，主键就是聚集索引。主键索引的叶子节点上存储了整行的数据，而辅助索引（Secondary Key）只存储了数据的指针。

PRIMARY KEY 约束唯一标识数据库表中的每条记录。主键必须包含唯一的值，不能包含NULL 值。每个表都应该有一个主键，并且每个表只能有一个主键。我们把qid问题id作为表中的主键，唯一地标识表中的某一条记录。

考虑到以下查询场景：

1. 统计近某段时间的时间的创建新问题数量；
2. 查询修改提问的问题；
3. 查询修改提问的用户和问题。

我们增加以下索引。

1. 创建时间create\_time；
2. 修改时间modify\_time；
3. 用户id和创建时间的联合索引 (user\_id, create\_time)。

综上，建表语句如下：

CREATE TABLE `question` (

`qid` bigint(20) unsigned NOT NULL COMMENT '问题id，64位id',

`question\_title` varchar(400) NOT NULL COMMENT '问题标题',

`question\_content` text NOT NULL COMMENT '问题详细内容',

`status` tinyint(4) NOT NULL COMMENT '问题状态',

`user\_id` bigint(32) NOT NULL COMMENT '用户id',

`extra` text NOT NULL COMMENT '扩展',

`create\_time` timestamp NOT NULL DEFAULT '0000-00-00 00:00:00' COMMENT '问题创建时间',

`modify\_time` timestamp NOT NULL DEFAULT CURRENT\_TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP COMMENT '修改时间',

PRIMARY KEY (`qid`),

KEY `idx\_mtime` (`modify\_time`),

KEY `idx\_user` (`user\_id`,`create\_time`),

KEY `idx\_ctime` (`create\_time`),

) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COMMENT='问题数据表';

回答表的相关字段设计如表4-8所示：

表4-8 回答表字段

|  |  |  |
| --- | --- | --- |
| 表字段名 | 数据类型 | 字段描述 |
| ans\_id | bigint | 答案id |
| answer\_content | text | 回答的内容 |
| answer\_abstract | text | 回答内容的摘要 |
| Qid | bigint | 回答的问题id |
| user\_id | bigint | 回答用户的id |
| create\_time | timestamp | 答案创建时间 |
| modify\_time | timestamp | 答案修改时间 |
| Extra | text | 答案扩展字段 |

ans\_id设计为bigint类型，可以容纳2^64个问题容量；qid设计为bigint类型，可以容纳2^64个问题容量；answer\_abstract设计为mediumtext类型，最大存储65535字符；answer\_content设计为mediumtext类型，最大存储16777215(2^24-1)个字符；user\_id设计为bigint，可以容纳2^64个用户容量；create\_time和modify\_time记录了问题的创建时间和修改时间，用timestamp数据类型存储，extra设计为text类型，用于存储答案的扩展字段，通常存储内容为Json格式。

我们把ansid回答id作为表中的主键，唯一地标识表中的某一条记录。建表语句如下:

CREATE TABLE `answer` (

`ans\_id` bigint(20) unsigned NOT NULL COMMENT '答案id，64位id',

`answer\_content` mediumtext NOT NULL COMMENT '回答的内容HTML',

`answer\_abstract` text COMMENT '回答内容的摘要',

`qid` bigint(20) unsigned NOT NULL COMMENT '回答的问题id',

`user\_id` bigint(20) unsigned NOT NULL COMMENT '回答用户的did',

`extra` text NOT NULL COMMENT '扩展',

`create\_time` timestamp NOT NULL DEFAULT '0000-00-00 00:00:00' COMMENT '答案创建时间',

`modify\_time` timestamp NOT NULL DEFAULT CURRENT\_TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP COMMENT '修改时间',

PRIMARY KEY (`ansid`)

) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COMMENT='答案数据表';

### **缓存设计与实现**

问题表和回答表的查询非常频繁，我们在数据库上增加一层缓存，缓存用Redis来实现，让请求首先请求到Redis上，以O(1)的时间复杂度取得问题、答案相关数据。

Redis的Key设计如表4-9所示:

表4-9 Redis Key字段

|  |  |  |
| --- | --- | --- |
| 名称 | 唯一id | 版本号 |
| Question | qid | 1.0 |
| Answer | ansid | 1.0 |

转换成代码表达，如下

Question\_Key = 'Question:%s:1.0' %qid  
 Answer\_Key = 'Answer:%s:1.0' %ansid

Redis有多种数据结构，包括Hash，Zset，String等，我们选取String类型存储。Json是常见的序列化工具，可以把List，Dict等数据格式转化成字符串形式。我们将问题和答案的属性信息序列化成Json格式，再存储到Redis上，其中Key为上述表所示，Value为对应的问答答案打包信息。

在查询的时候，请求首先请求到缓存上，从Redis中获取问题/答案数据，如果没有命中缓存，则去对应的从库读取数据，并回写到缓存Redis中。

过期时间我们设置为：1200s。

缓存流程图如图4-2所示：



图4-2 缓存流程图

具体实现逻辑如下：

1.读取缓存数据，记录没有命中的qid/ansid

Keys = makeKey(qid)

Data= Redis\_client.mget (Keys)

Result = Json.dumps(Data)

2.去数据库中查询未命中id

with session\_scope() as session:  
 res =session.query(Question).filter(Question.qid == qid).all()

3.返回数据，其中数据内容 = 缓存查询数据 + 未命中后db查询数据

for Key, item in res:

Result.(Key) = item

return result

4.回写Redis

for Key, item in res:

Keys = makeKey(Key)

Value = Json.loads(item)

Redis\_client.setex(Key, Value, expireTime)

其中 [SETEX](http://redisdoc.com/string/setex.html#setex) 是一个原子性(atomic)操作，关联值和设置生存时间两个动作会在同一时间内完成，该命令在 Redis 用作缓存时，非常实用。缓存除了自动过期以外，还会在用户修改问题/答案内容的时候，同时删除问题、答案的缓存。

## 查询服务

在提交服务中，完成了数据的存储设计，查询服务基于之前的数据存储来实现查询。

### **查询服务接口设计**

查询服务主要有以下几种查询类型：

1. 查询问题相关信息；
2. 查询答案相关信息。

具体接口描述如下：

1.MgetQuestion查询问题

MgetQuestion接口实现查询问题功能，接口字段设计如表4-10所示：

表4-10 查询问题接口字段

|  |  |
| --- | --- |
| 字段 | 字段描述 |
| Qid\_List | 问题id列表 |

具体接口代码如下，以protobuf形式展示：

struct MgetQuestionReq{  
1: required list<i64> Qid\_List, // 问题id列表  
}

MgetQuestion接口返回字段如表4-11所示：

表4-11 查询问题接口返回字段

|  |  |
| --- | --- |
| 字段 | 字段描述 |
| Qid | 问题id |
| Question\_Title | 问题的标题 |
| User\_id | 发表问题的登录用户id |
| Uname | 发表问题的登录用户名 |
| Create\_Time | 发表问题的时间 |

struct MgetQuestionInfoResponse{  
1: required list<QuestionData> QuestionList, // 返回的问题数据列表  
2: required i32 ListNum, // 返回的数据的数量  
}

struct QuestionData{  
1: required i64 Qid, // 问题id  
2: required string Question\_Title, // 问题的标题  
3: required i64 User\_id, // 发表问题的登录用户id  
4: required string Uname, // 发表问题的登录用户名  
5: required i64 Create\_Time, // 发表问题的时间  
}

2.MgetAnswer 查询答案

MgetAnswe接口实现查询答案功能，接口字段如表4-11所示：

表4-11 查询答案接口字段

|  |  |
| --- | --- |
| 字段 | 字段描述 |
| Ansid\_List | 回答id列表 |

具体接口代码如下，以protobuf形式展示：

struct MgetAnswerReq{  
1: required list<i64> Ansid\_List, // 回答id列表  
}

MgetAnswer接口返回字段如表4-12所示：

表4-12 查询答案接口返回字段

|  |  |
| --- | --- |
| 字段 | 字段描述 |
| Ans\_id | 回答id |
| Answer\_Content | 回答内容 |
| User\_id | 回答的用户id |
| Uname | 回答的用户名称 |
| Create\_Time | 回答的时间 |

具体接口代码如下，以protobuf形式展示：

struct MgetAnswerResponse{  
1: required list<AnswerData> AnsList, // 返回回答id列表  
2: required i32 ListNum, // 返回的数据的数量  
}

struct AnswerData{  
1: required i64 Ans\_id, // 回答id  
2: required string Answer\_Content, // 回答内容，html模板数据  
3: required i64 User\_id, // 回答的登录用户id  
4: required string Uname, // 回答的登录用户名称  
5: required i64 Create\_Time, // 回答的事件，unix时间戳  
}

接口具体实现以查询答案为例，首先调用方按照接口约束，传入Ans\_id答案ID（可以同时传入多个ID）；查询服务接受到调用信息后，从缓存中去相应的回答数据，如果没有命中缓存，则从数据库以中获取对应的信息，并且将数据库中的信息写到缓存中；返回数据。

流程如图4-3所示：



图4-3 查询答案流程图

Python 的解释性语言特性以及其落后的多进程服务模型受到了巨大的挑战。GO 语言性能高，编译快，开发效率也不低，原生支持并发，协程模型是非常优秀的服务端模型，同时也适合网络调用。

在第五章的测试中可以看到在高并发请求下，服务的相应情况。

## 邀请服务

### **邀请服务接口设计**

1. Invite 邀请接口

Invite接口实现了邀请功能，接口字段设计如表4-13所示：

表4-13 邀请接口字段

|  |  |
| --- | --- |
| 字段 | 字段描述 |
| QID | 邀请的问题 |
| User\_id\_List | 被邀请人 |
| From\_User\_ID | 邀请人 |

具体接口代码如下，以protobuf形式展示：

struct InviteRequest{

1: required i64 QID, //邀请的问题

2: optional list<i64> User\_id\_List, //邀请的人

3: optional i64 From\_User\_ID, //谁邀请的

}

Invite接口返回字段设计如表4-14所示：

表4-14 邀请接口返回字段

|  |  |
| --- | --- |
| 字段 | 字段描述 |
| Invite\_Result | 邀请是否成功 |
| Invite\_Result\_Tips | 错误信息 |

具体接口代码如下，以protobuf形式展示：

struct InviteResponse{

1: required map<i64, i32> Invite\_Result, //邀请是否成功

2: optional map<i64, string> Invite\_Result\_Tips, // 邀请的一些错误信息

}

接口逻辑具体实现：

首先调用方按照接口约束，传入QID，User\_id，From\_User\_ID字段信息；提交服务接受到调用信息后，从数据库中获取对应的信息（比如从用户信息表中获取用户相关信息）；对所获得的参数进入输入合法性检查；检查具体的业务策略（比如是否满足权限控制：用户是否有邀请权限等）；提交数据，将数据写入数据库中；返回邀请结果，进行善后处理（例如发送异步消息队列等）。

流程图如图4-4所示：



图4-4 邀请流程图

### **存储设计**

邀请服务主要涉及两张表：

1. 邀请信息表：

存储邀请状态，包括问题qid，被邀请人to\_uid，邀请人from\_uid，邀请时间create\_time，邀请状态invite\_status等。（qid，user\_id，from\_uid）三元组唯一，表示同一问题，同一用户，用户邀请只能邀请一次

2. 邀请结果表：

存储邀请结果，包括问题qid，被邀请人user\_id，接受邀请时间receive\_time，是否回答过该问题is\_answered，问题展示状态question\_status，最新邀请时间last\_receive\_time。（qid，user\_id）二元组唯一，记录单一问题单一用户的邀请结果

邀请信息表设计如表4-15所示：

表4-15 邀请信息表字段

|  |  |  |
| --- | --- | --- |
| 表字段名 | 数据类型 | 字段描述 |
| Id | bigint | 自增id |
| Qid | bigint | 问题id |
| from\_uid | bigint | 邀请来源用户id |
| to\_uid | bigint | 被邀请用户id |
| invite\_status | tinyint | 邀请状态 已邀请or待邀请 |
| create\_time | timestamp | 邀请创建的时间 |

id设计为bigint类型，可以容纳2^64个自增id；qid 设计为bigint类型，可以容纳2^64个问题容量；from\_uid设计为bigint类型，可以容纳2^64个用户id；to\_uid设计为bigint类型，可以容纳2^64个用户id；invite\_status tinyint类型，记录邀请状态是已邀请、未邀请；create\_time记录了问题的创建时间和修改时间，用timestamp数据类型存储。

因为一个问题可能被邀请多次；一个用户可能邀请别人多次，所以这两者都不能作为主键id，所以我们用一个自增id来作为主键，唯一的标识表中的唯一记录。

我们用(`qid`, `from\_uid`, `to\_uid`)这个三元组确定：一个问题，只能被一个人邀请另一个人一次，避免一个人对其他用户的重复邀请，避免打扰用户。

除此之外，考虑到以下查询场景：

1. 查询某一个用户邀请某一问题的情况，来做频率控制，比如同一问题下，一个人只能邀请5个人等；

2. 查询某人最近的邀请；

3. 查询一个人被一个问题邀请的次数。

我们增加以下索引。

1. 邀请用户uid和问题qid的联合索引 (`from\_uid`, `qid`)；

2. 邀请时间和被邀请人的联合索引(`create\_time`, `to\_uid`)；

3. 邀请人和邀请问题的联合索引(`from\_uid`, `qid`)。

建表语句如下:

CREATE TABLE `invit\_information` (

`id` bigint(20) unsigned NOT NULL AUTO\_INCREMENT COMMENT '自增id',

`qid` bigint(20) unsigned NOT NULL COMMENT '问题id',

`from\_uid` bigint(20) unsigned NOT NULL COMMENT '邀请来源用户id',

`to\_uid` bigint(20) unsigned NOT NULL COMMENT '被邀请用户id',

`invite\_status` tinyint(4) DEFAULT NULL COMMENT '0',

`create\_time` timestamp NOT NULL DEFAULT CURRENT\_TIMESTAMP COMMENT '邀请创建的时间',

PRIMARY KEY (`id`),

UNIQUE KEY `uniq\_Question` (`qid`,`from\_uid`,`to\_uid`),

KEY `idx\_from\_user` (`from\_uid`,`qid`),

KEY `idx\_to\_user` (`to\_uid`,`qid`),

KEY `idx\_create\_time` (`create\_time`,`to\_uid`)

) ENGINE=Innodb AUTO\_INCREMENT=73298772 DEFAULT CHARSET=utf8 COMMENT='邀请信息表';

邀请结果表设计：

邀请的相关字段设计如图4-16所示：

表4-16 邀请结果表字段

|  |  |  |
| --- | --- | --- |
| 表字段名 | 数据类型 | 字段描述 |
| Id | bigint | 自增id |
| Qid | bigint | 问题id |
| user\_id | bigint | 邀请用户id |
| receive\_time | timestamp | 接受邀请的时间 |
| is\_answered | tinyint | 是否回答过该问题 |
| question\_status | tinyint | 问题展示状态 |
| last\_recv\_time | timestamp | 最近一次被邀请时间 |

id设计为bigint类型，可以容纳2^64个自增id；qid 设计为bigint类型，可以容纳2^64个问题容量；uid设计为bigint类型，可以容纳2^64个用户id；is\_answered为tinyint类型，记录邀请问题是回答过、未回答过；question\_status tinyint类型，记录邀请问题展示状态；receive\_time和last\_receive\_time记录了用户接受邀请时间和最新一次被邀请时间，用timestamp数据类型存储。

因为一个问题可能被邀请多次；一个用户可能邀请别人多次，所以这两者都不能作为主键id，所以我们用一个自增id来作为主键，唯一的标识表中的唯一记录。

考虑到以下查询场景：

1. 查询某问题的邀请情况；

2. 查询某用户邀请别人的情况；

3. 查询某个时间点的邀请/查询近期的邀请。

于是我们增加以下索引：

1. 问题id qid；

2. 邀请时间 recv\_time；

3. 用户id uid。

建表语句如下:

CREATE TABLE `Invited\_result` (

`id` bigint(20) unsigned NOT NULL AUTO\_INCREMENT COMMENT '自增id',

`qid` bigint(20) unsigned NOT NULL COMMENT '问题id',

`uid` bigint(20) unsigned NOT NULL COMMENT '邀请用户id',

`receive\_time` timestamp NOT NULL DEFAULT CURRENT\_TIMESTAMP COMMENT '接受邀请的时间',

`is\_answered` tinyint(4) DEFAULT '0',

`question\_status` tinyint(4) NOT NULL DEFAULT '1' COMMENT '问题状态',

`last\_receive\_time` timestamp NOT NULL DEFAULT CURRENT\_TIMESTAMP COMMENT '最近一次被邀请时间',

PRIMARY KEY (`id`),

KEY `idx\_Question` (`qid`),

KEY `idx\_user` (`uid`),

KEY `idx\_recv\_time` (`recv\_time`)

) ENGINE=Innodb AUTO\_INCREMENT=71708365 DEFAULT CHARSET=utf8 COMMENT='邀请关系表';

邀请拒绝表设计如表4-17所示：

表4-17 邀请拒绝表字段

|  |  |  |
| --- | --- | --- |
| 表字段名 | 数据类型 | 字段描述 |
| Id | bigint | 自增的id |
| Qid | bigint | 问题id |
| from\_uid | bigint | 邀请来源用户id |
| to\_uid | bigint | 被邀请用户id |
| invite\_result | tinyint | 拒绝邀请的理由 |
| invite\_time | timestamp | 邀请处理时间 |

id设计为bigint类型，可以容纳2^64个自增id；qid 设计为bigint类型，可以容纳2^64个问题容量；uid设计为bigint类型，可以容纳2^64个用户id；from\_uid设计为bigint类型，可以容纳2^64个用户id；invite\_result tinyint类型，记录拒绝邀请的理由；invite\_time记录了邀请的时间，用timestamp数据类型存储。

考虑到以下查询场景：

1. 查询某问题的邀请情况；

2. 查询某用户邀请别人的情况；

3. 查询某个用户被邀请的情况；

4. 查询用户近期邀请的情况。

于是我们增加以下索引，方便查询：

1. 邀请用户uid和问题qid的联合索引 (`from\_uid`,`qid`)；

2. 被邀请用户uid和问题qid的联合索引 (`to\_uid`,`qid`)；

3．被邀请用户和邀请时间的联合索引 (`Invite\_time`,`to\_uid`)；

4. 问题id qid。

建表语句如下：

CREATE TABLE `reject\_Invite` (

`id` bigint(20) unsigned NOT NULL AUTO\_INCREMENT COMMENT '自增的消息id',

`qid` bigint(20) unsigned NOT NULL COMMENT '问题id',

`from\_uid` bigint(20) unsigned NOT NULL COMMENT '邀请来源用户id',

`to\_uid` bigint(20) unsigned NOT NULL COMMENT '被邀请用户id',

`invite\_result` tinyint(4) unsigned NOT NULL DEFAULT '0' COMMENT '邀请结果',

`invite\_time` timestamp NOT NULL DEFAULT CURRENT\_TIMESTAMP COMMENT '邀请处理时间',

PRIMARY KEY (`id`),

KEY `idx\_Question` (`qid`),

KEY `idx\_from\_user` (`from\_uid`,`qid`),

KEY `idx\_to\_user` (`to\_uid`,`qid`),

KEY `idx\_Invite\_time` (`Invite\_time`,`to\_uid`)

) ENGINE=Innodb AUTO\_INCREMENT=9054470 DEFAULT CHARSET=utf8 COMMENT='邀请拒绝信息表';

## 小结

本章对拆分出的三个微服务模块进行详细设计，包括功能梳理，数据持久化存储实现等详细设计，针对不同服务特性采用了不同的语言实现，每个服务有自己独立的数据管理及异步处理。在数据存储实现中，详细描述了数据库表的设计，包括字段定义和索引建立，以及相应表的缓存设计，以支持高并发高负荷的查询。

1. 测试

## 测试环境

本次测试均在项目所部署的物理机中进行。环境基础配置如下：

系统版本：Debian GNU/Linux 8.9 (jessie)。

内存：4G。

CPU：Intel(R) Xeon(R) CPU E5-2630 v4 @ 2.20GHz。

## 测试方法

一台Web服务起在单位时间内能处理的请求越多越好，这也成为了Web服务器的能力高低所在，体现了我们常说的“服务器并发处理能力”。本章所说的服务器指的是部署微服务，提供服务的服务器，不涉及操作系统和内核的内容。

常见的衡量Web服务器并发处理能力指标之一是吞吐率。我们一般使用单位时间内服务器处理的请求数来描述其并发处理能力，习惯撑其为吞吐率Troughtoutput。吞吐率这个概念有时候还用于描述其他指标，比如单位时间内通信数据量等，但本章中，吞吐率指单位时间内服务器处理的请求数。

常见的衡量指标之二是用户等待时间。事实上我们所关心的时间有以下两种：

（1） 用户平均请求等待时间

（2） 服务器平均请求处理时间

如果并发用户数为1，只有一个用户在向服务器源源不断地发送请求，那么每个请求的等待时间也就是它的处理时间，等于总时间除以总请求数，这是用户平均请求等待时间和服务器平均请求处理时间是相同的。

假设并发用户数为100，那么便会有100个用户同时向服务器发送请求。这是Web服务器一半会采用多进程或者多线程的并发模型，通过多个执行流来同时处理多个并发用户请求，而多执行流的的设计原则就是轮流交错使用CPU时间片，所以每个执行流花费的时间都被拉长。对每个而用户而言，每个请求的平均等待时间必然增加；对于服务器而言，如果并发策略得当，每个请求的平均处理时间可能减少。

本文选取成功调用服务延时和吞吐率作为服务性能衡量指标。

其中吞吐率为单位时间内服务器处理的请求数，用来来描述其并发处理能力，由于业务性质，查询场景偏多，提交和邀请场景相对没有特别高的并发请求。在查询服务中可以明显看到高吞吐率的性能表现。

延时我们采取两种统计方式，一种是平均延时，一种percent 99的方式。

平均延时为，我们先对每台机器的单位时间请求延时做平均，在对所有机器在该时间点的平均时延做取和平均，这样得到的也就是所有请求的平均耗时。

Percent 99的延时为，我们将每台机器的单位请求延时时间，从小到大排序，选取第99的数据结果点，表示99%的结果都小于这个数，再对所有机器加和求平均，这样能反馈出实际最长的请求时延。

* + 1. 邀请服务性能分析

邀请服务性能分析如下：

由于用户邀请是随机进行的，所以选取了在随机1小时内，服务的的性能情况。Invite接口成功率延时如图5-1所示：成功调用接口的平均时延在400毫秒左右，最高达到800毫秒。



图5-1 邀请服务邀请接口延迟

我们将一台机器的所有请求进行排序，取99%的分位点，再将所有机器求平均，得到percent 99的数据。如图5-2所示。可见对99%的调用请求，可以99%的服务调用时延平均在800毫秒，该小时段内的延时峰值为1800毫秒。



图5-2 邀请服务邀请接口Pct99延迟

该时间段内的请求量如图5-3所示，平均QPS为每秒0.5的请求量。



图5-3 邀请服务邀请接口QPS

* + 1. 提交服务性能分析

提交服务性能分析如下：

用户提交类行为也相对随机，所以选取了在随机1小时内服务的的性能情况。修改答案接口成功率延时如图5-4所示：成功调用接口的平均时延在200毫秒左右，峰值达到300毫秒。



图5-4 提交服务修改答案接口延迟

同时，我们选取数据的99%分位点，获得更覆盖绝大部分请求的延时结果，如图5-5所示。修改答案接口的99%请求延时平均在800毫秒，峰值在1500毫秒。



图5-5 提交服务修改答案接口Pct99延迟

该时间段内的请求量如图5-6所示，平均QPS为每秒8的请求量。



图5-6 提交服务修改答案接口QPS

提交答案接口成功率延时如图5-7所示：成功调用接口的平均延时在150毫秒左右，最高平均延时达到300毫秒。



图5-7 提交服务提交答案接口延迟

同时，我们选取数据的99%分位点，获得更覆盖绝大部分请求的延时结果。如图5-8所示：99%的请求延时平均在800毫秒，峰值在1700毫秒。



图5-8 提交服务提交答案接口Pct99延迟

该时间段内的请求量如图5-9所示，平均QPS为每秒3.6的请求量。



图5-9 提交服务提交答案接口QPS

* + 1. 查询服务性能分析

查询服务性能分析如下：

对于高并发的查询请求调用，不同时间段请求量差距较大，我们选取24小时时间段，来体现服务的的性能情况。查询答案接口成功率延时如图5-10所示：成功调用接口的调用延迟平均在2.5毫秒左右，最高达到4.5毫秒。



图5-10 查询服务查询答案接口延时

同时，我们选取数据的99%分位点，获得更覆盖绝大部分请求的延时结果。如图5-11所示：99%的请求延时在10毫秒以下，峰值在45毫秒。



图5-11 查询服务查询答案接口Pct99延时

该时间段内的请求量如图5-12所示，平均QPS为每秒8000的请求量，峰值请求量为12000。



图5-12 查询服务查询答案接口QPS

查询问题接口成功率延时如图5-13所示：成功调用接口的平均延迟在1.2毫秒左右，最高达到2.7毫秒。如图5-1所示。



图5-13 查询服务查询问题接口延时

同时，我们选取数据的99%分位点，获得更覆盖绝大部分请求的延时结果。如图5-14所示：99%的请求延时在5毫秒以下，峰值在25毫秒。



图5-14 查询服务查询问题接口Pct99延时

该时间段内的请求量如图5-15所示，平均QPS为每秒3000的请求量，峰值请求量为5000。



图5-15 查询服务查询答案接口QPS

除此之外，微服务部署特点为服务直接独立开发，独立部署，便于持续交付。服务部署在Docker容器中，在服务发现的机制下，请求高峰期扩容时无需改动改动代码，只需增加部署机器的数量，可以良好的支持高并发情况下的服务性能。

## 小结

本章介绍了服务测试环境，以及测试的指标设计，对前文中所设计的微服务模块进行测试，获取实时的数据绘图，验证了服务的性能，验证了服务在高并发请求下的高性能表现：例如查询服务在万级QPS下延时为2-3毫秒，使得用户浏览内容时有良好的用户体验，需求满足业务需求。

1. 总结

本文的主要工作如下：

对微服务架构设计原则进行探讨，分析业务需求，拆分出三个基础功能模块，分别是查询服务，提交服务，邀请服务。介绍服务端开发技术，包括数据库技术、缓存技术、异步消息队列技术，并根据业务需求组合相关技术。

在微服务的架构下，实现了查询服务，提交服务，邀请服务的服务端设计，并针对业务情况，对每个服务进行了独自的存储设计，包括数据持久化和缓存设计，针对具体的数据设计，设计了字段设计，并优化查询，添加索引。对查询服务，因为Python的解释性语言特性以及其落后的多进程服务模型，我们选择GO语言对查询服务进行重构。GO语言原生支持并发，协程模型是非常有些的服务端模型，适合网络调用。

在实现模块后对模块进行了性能测试，模块体现了良好的支持高可用高并发的需求性能，查询服务可以支持万级的高QPS访问。

在本次实践中，对微服务架构也有了深刻的认识，并总结出微服务的优势：微服务模块每个模块专注于自己的功能，模块与模块之间是高内聚低耦合的。由于微服务的独立部署和开发特性，对于不同的微服务模块可以用不同的编程语言及编程工具进行开发，利用语言的特性更有针对性的解决问题。除此之外，由于微服务模块之间相互对立，开发团队也可相互独立，互不影响，这样平均人力，可以加快开发交付市场。微服务架构也很好的支持了持续开发持续交付，在频繁推出新的微服务模块时保证其他微服务不受影响，保证整个系统其他部分的可用性和稳定性。

于此同时也了解到微服务架构下的一些开发时应注意的点：微服务架构部署运行多个独立的服务，可能需要支持多种语言环境；开发人员通常开发运维一体化，也需要掌握必要的数据库存储技术，除此之外开发人员也应了解分布式系统在实际开发运维中会遇到的问题。

参考文献

[1] <http://blog.csdn.net/qq_31655965/article/details/71258191>

[2] 李苏璇.基于微服务架构的SaaS应用构建方法研究.硕士学位论文[D].广东:华南理工大学.2016

[3] 王磊 微服务架构与实践. 北京：电子工业出版社,2015.

[基于微服务架构的统一应用开发平台]

[4] 叶钰，应时，李伟斋等.面向服务体系结构及其系统构建研究[J]. 计算机应用研究,2005,22(2):

[5] Nils Joachim, Daniel Beimborn, Tim Weitzel et al. The influence of SOA governance mechanisms on IT flexibility and service reuse[J]. The journal of strategic information systems, 2013,22(1):86-101

[6] Docker[EB/OL.].https://www.docker.com/what-docker.2016

[7]Microservice[EB/OL.].http://martinfowler.com/article/microservices.html.2014

[8] 马威，韩臻，成阳.可信云计算中的多级管理机制研究[j].信息网络安全，2015，(7):20- 25

[9] <https://en.wikipedia.org/wiki/Representational_state_transfer>

[10] <http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>

[11] <http://www.ruanyifeng.com/blog/2011/09/restful.html>

[12] <http://www.ruanyifeng.com/blog/2014/05/restful_api.html>

[13] <http://blog.csdn.net/mindfloating/article/details/39474123/>

[14]<https://www.w3cschool.cn/architectroad/architectroad-rpc-framework.html>

[15] <https://github.com/apache/thrift>

[16] <https://zh.wikipedia.org/zh-cn/Python>

[17] <http://docs.jinkan.org/docs/flask/>

[18] <https://zh.wikipedia.org/zh-cn/NoSQL>

[19] <http://www.cnblogs.com/lina1006/archive/2011/04/29/2032894.html>

[20] <http://blog.nosqlfan.com/html/1087.html>

[21]<http://www.jasongj.com/2015/01/02/Kafka%E6%B7%B1%E5%BA%A6%E8%A7%A3%E6%9E%90/>

[22] 闫璐.3D云游戏平台的设计与实现.硕士学位论文[D].北京:北京交通大学.2014

[23] 廖伟桥.分布式网页游戏服务端架构技术研究与实现.硕士学位论文[D].四川:西南交通大学.2017

[24] 李俊.服务QoS多指标预测系统的设计与实现.硕士学位论文[D].北京:北京邮电大学.2014

[25] 杨金彬.个人云同步系统服务端的分析与实现.硕士学位论文[D].北京:北京邮电大学.2012

[26] 郭路易.基于服务端反馈的服务质量评价与推荐技术研究.硕士学位论文[D].上海:上海交通大学.2008

[27] 林征.基于面向服务架构的系统集成技术研究.硕士学位论文[D].上海:上海交通大学.2008

[28] 刘晓光.基于MySQL的分布式SQL数据库的设计与实现.硕士学位论文[D].北京:中国科学院大学.2016

[29] 刘晓光.基于MySQL的分布式SQL数据库的设计与实现.硕士学位论文[D].北京:中国科学院大学.2016

[30] 朱红.基于MySQL集群实现的高性能数据库架构设计.硕士学位论文[D].北京:中国科学院大学.2016

[31] 曾泉匀.基于Redis的分布式消息服务的设计与实现.硕士学位论文[D].北京:北京邮电大学.2014

[32] 刘兴邦. 基于Restful和Androi\_省略\_的途家网房源管理系统的设计与实现.硕士学位论文[D].北京:北京交通大学.2015

[33] 庄鹏.基于ZooKeeper的分布式服务中间件设计与实现.硕士学位论文[D].广东:深圳大学.2015

[34] 宁维宇.面向服务的未来互联网服务迁移策略的研究.硕士学位论文[D].北京:北京邮电大学.2013

[35] 项凯.面向海量高并发数据库中间件的研究与应用.硕士学位论文[D].上海:上海交通大学.2015

[36] 开金宇.面向可靠性的微服务系统自适应调整技术研究.博士学位论文[D].上海:上海大学.2016

[37] 王啸.面向资源架构的业务逻辑执行容器的设计与实现.硕士学位论文[D].北京:北京邮电大学.2013

[38] 吴兆定.棋牌游戏平台服务端的设计与实现.硕士学位论文[D].上海:上海交通大学.2007

[39] 王玉奎.通用高性能网络棋牌游戏服务端的研究和设计.硕士学位论文[D].湖北:武汉理工大学.2011

[40] 郁大威.网络游戏服务器架构技术研究与优化.硕士学位论文[D].上海:上海交通大学.2014

[41] 李亚明.烟草企业营销渠道微服务管理系统的设计实现.硕士学位论文[D].山东:山东大学.2015

[42] 梁志君.针对Web应用的服务端准入控制技术的研究.硕士学位论文[D].北京:北京邮电大学.2015

[43] 王冲.支持QoS的语义Web服务发现的研究与实现.硕士学位论文[D].上海:上海交通大学.2009

[44] 高尚.Ajax\_REST架构的研究与开发框架的实现.硕士学位论文[D].北京:北京邮电大学.2008

[45] 程桂花.MVVM前后端数据交互中安全机制的研究与实现.硕士学位论文[D].浙江:浙江理工大学.2016

[46] 林瑶. Web实时数据同步研究.硕士学位论文[D].北京:北京邮电大学.2015

[47] <http://martinfowler.com/articles/microservices.html>

[48] <https://zh.wikipedia.org/wiki/Python>

[49] https://zh.wikipedia.org/wiki/Nosql

致谢

在论文完成之际，堇向我尊敬的导师张洪欣老师致以由衷的感谢！感谢张老师在读研期间对我的谆谆教导和指点，在张老师的带领下，实验室的学术氛围浓郁，良好的科研条件让我的研究生生涯过的非常充实。感谢张老师为我的课题付出的诸多心血，在我科研遇到瓶颈时的不遗余力的指导。感谢张老师在研究生期间对我的包容和引导，张老师正直，高尚的人格时刻的影响着我，让我以端正积极的心态面对学习和生活。同时要感谢吕老师,感谢张金铃老师在学习和生活的各个方面给我的指导。

同时，我也要感谢实验室的同学和我的朋友，在他们的帮助和陪伴下，使得我的研究生学习生活过的满足和充实。感谢振友的在我科研中的指导，感谢伟斌的谆谆教导，感谢伟鹏细致到点的指导，感谢原嘉强有力的支持，感谢刘杰同学不厌其烦的帮助和开导。特别感谢夏翼老师，亦师亦友，在专业知识上和职业生涯上都给予了我很大的指导和帮助，夏翼老师严谨的做事风格，极大的影响了我，使我受益终身。

最后我要感谢我的家人，一直以来对我的学业上的支持和生活上的关照，有他们的陪伴才使我能够以单纯的心态去学习成长，去探寻生活的深意。