**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。这样就能够极大的加快业务开发速度，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. 阐述交互式应用需求，数据规模量级，服务的稳定性与容灾
2. 对微服务架构架构设计原则进行探讨，并对应业务需求，组合服务端相关技术，包括Restful框架、数据库技术、缓存技术，异步消息队列技术。
3. 基于微服务设计服务端功能模块,制定微服务模块设计方案,其中微服务模块主要包括有:基础服务，提交服务，用户服务。并在模块的具体实现中，对满足交互式应用性能要求下，对微服务模块的数据持久化存储，缓存设计，异步处理等关键点进行设计。
4. 微服务模块相关功能实现,并对功能及性能进行测试，绘制数据分析图表。

基于微服务的架构设计能够让开发人员精确的拆分通用的功能模块，模块之间的开发高内聚低耦合，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。本文在基于微服务架构的基础上，设计与实现支持高并发的交互式应用的服务端功能模块。

**关键词：**微服务；服务端架构；Restful交互式应用

The "Microservices" architecture is a very popular concept in recent 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

2) Discuss the design principles of Microservices architecture architecture, and study related technologies, including Restful API, database, cache, and asynchronous message queue.

3) 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.

4) 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.

**目 录**

[第一章 绪论 4](#_Toc500714915)

[1.1 研究背景 4](#_Toc500714916)

[1.2 研究内容与研究意义 5](#_Toc500714917)

[1.3 国内外研究现状 5](#_Toc500714918)

[1.4 各章安排 5](#_Toc500714919)

[第二章 项目关键技术分析 7](#_Toc500714920)

[2.1 微服务架构设计原则 7](#_Toc500714923)

[**2.1.1** **单一功能原则** 7](#_Toc500714926)

[**2.1.2** **轻量级通信** 7](#_Toc500714927)

[**2.1.3** **分散的数据管理** 7](#_Toc500714928)

[**2.1.4** **独立部署** 7](#_Toc500714929)

[2.2 web服务端关键技术分析 7](#_Toc500714930)

[**2.2.1** **Restful** 7](#_Toc500714932)

[**2.2.2** **RPC Thrift** 8](#_Toc500714933)

[**2.2.3** **python web 框架** 9](#_Toc500714934)

[**2.2.4** **数据持久化** 13](#_Toc500714938)

[**2.2.5** **缓存技术** 15](#_Toc500714940)

[**2.2.6** **消息队列** 16](#_Toc500714941)

[第三章 微服务架构及模块总体设计 18](#_Toc500714942)

[3.1 服务端架构设计需求分析 18](#_Toc500714943)

[**3.1.1** **微服务拆分设计** 18](#_Toc500714946)

[3.2 整体架构分析 19](#_Toc500714947)

[**3.2.1** **交互式应用微服务架构设计** 19](#_Toc500714949)

[**3.2.2** **存储设计** 20](#_Toc500714950)

[**3.2.3** **消息队列** 22](#_Toc500714953)

[3.3 微服务模块设计 23](#_Toc500714954)

[**3.3.1** **查询服务总体设计** 23](#_Toc500714956)

[**3.3.2** **提交服务总体设计** 24](#_Toc500714957)

[**3.3.3** **邀请服务的详细设计** 25](#_Toc500714958)

[3.4 模块对外提供接口 26](#_Toc500714959)

[**3.4.1** **json 服务返回数据** 26](#_Toc500714961)

[**3.4.2** **photobuf idl 对外接口提供** 26](#_Toc500714962)

[第四章 重要模块及接口详细设计 28](#_Toc500714963)

[4.1 提交服务具体设计 28](#_Toc500714964)

[**4.1.1** **提交类事件定义设计** 28](#_Toc500714967)

[**4.1.2** **提交服务接口设计** 29](#_Toc500714968)

[**4.1.3** **存储设计** 33](#_Toc500714969)

[**4.1.4** **缓存设计** 35](#_Toc500714970)

[4.2 查询服务 36](#_Toc500714971)

[**4.2.1** **查询服务接口设计** 37](#_Toc500714973)

[**4.2.2** **服务go化** 39](#_Toc500714974)

[4.3 邀请服务 40](#_Toc500714975)

[**4.3.1** **邀请服务接口设计** 40](#_Toc500714977)

[**4.3.2** **存储设计** 41](#_Toc500714978)

[第五章 测试 47](#_Toc500714979)

[5.1 测试环境 47](#_Toc500714980)

[5.2 测试方法 47](#_Toc500714981)

[第六章 总结 51](#_Toc500714982)

[参考文献 53](#_Toc500714983)

[致谢 55](#_Toc500714984)

1. 绪论
   1. 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（例如 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。此外，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，SaaS及 PaaS）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。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>)。



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。

* 1. 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. 围绕业务能力组织服务
3. 产品而非项目模式
4. “去中心化”治理及数据管理
5. 演进式的设计等

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设计方案，满足交互式应用的高并发请求，海量数据存储，良好的扩展性等业务及性能要求。

本论文针对交互式应用，设计了微服务架构加，基础服务模块的设计，满足交互式应用的基本交互功能，包括查询，提交，交互等基本功能。

综上所述，本课题设计了一种针对微服务架构下，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的实现，提供了灵活数据存储方案，以便更好地使用微服务架构本身的优势。

1.2.3， 国内外研究现状

* 1. 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，并对国内外相关研究进行了介绍。

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，例如应用到的语言，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。其次分析了模块实际应用上的QOS测试了组件的高可用性。

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，基于微服务架构的web服务端关键技术：包括web服务端的语言，对内/对外提供服务的框架，服务端常用存储工具，异步消息队列，服务部署等。

2. 1. 微服务架构设计原则

微服务架构通常有以下的通用特性。

1. 1. 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，把组件定于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。

* + 1. **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，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。

* + 1. **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。微服务倾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。

* + 1. **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。

* 1. web服务端关键技术分析
  2. 1. **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。

Restful架构上的特性有：

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，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； 等等状态码。

在基于restful api的设计风格下，我们可以设计出类似于下面的接口。

<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

* + 1. **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：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，而远程调用还会跑出 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。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，rest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(thrift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，比如即将开工的大权限系统)。现有的Rpc框架有很多，本文中涉及的rpc框架是thrift。Thrift 是facebook提出的开源的跨平台的RPC框架。

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。

* + 1. **Python Web 框架**

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，有很多成熟的模板技术。

常见的python web框架有flask，django等。

* + 4. **Django**

Django是一个高级的Python Web框架，可以快速开发安全和可维护的网站。使得可以专注于编写应用程序，而无需重新开发。它是免费和开源的，拥有繁荣和积极的社区，伟大的文档和许多免费和付费支持的选项。

Django最初由2003年至2005年间由负责创建和维护报纸网站的网络团队开发。在创建了许多网站后，团队开始考虑并重用很多常见的代码和设计模式。这个共同的代码演变成一个通用的Web开发框架，2005年7月被开源为“Django”项目。 从2008年9月的第一个里程碑版本（1.0）到最近发布的1.11（2017）版本，Django一直在不断发展壮大。每个版本都添加了新的功能和错误修复，从支持新类型的数据库，模板引擎和缓存，到添加“通用”视图函数和类（这减少了开发人员必须编写的代码量）。

Django现在是一个蓬勃发展的合作开源项目，拥有数千个用户和贡献者。虽然它仍然有一些反映其起源的功能，但Django已经演变成一个能够开发任何类型的网站的多功能框架。

在传统的数据驱动网站中，Web应用程序等待来自Web浏览器（或其他客户端）的HTTP请求。当接收到请求时，应用程序根据URL和可能的POST数据或GET数据信息，制定出所需要的内容。根据需要，可以从数据库读取或写入信息，或执行满足请求所需的其他任务。然后，应用程序将返回对Web浏览器的响应，通常通过将检索到的数据插入HTML模板中的占位符来动态创建浏览器显示的HTML页面。

Django Web应用程序通常将处理这些步骤的代码分组到单独的文件中：



URL： 虽然可以通过单个功能处理来自每个URL的请求，但是编写单独的视图功能来处理每个资源是更加可维护的。URL映射器用于根据请求URL将HTTP请求重定向到相应的视图。URL映射器还可以匹配出现在URL中的字符串或数字的特定模式，并将其作为数据传递给视图函数。

视图：  视图是一个请求处理函数，它接收HTTP请求并返回HTTP响应。视图访问通过模型满足请求所需的数据，并将响应的格式委托给  模板。

模型：  模型是定义应用程序数据结构的Python对象，并提供了管理（添加，修改，删除）和查询数据库中记录的机制。

模板：  模板是定义文件（例如HTML页面）的结构或布局的文本文件，用于表示实际内容的占位符。一个视图可以使用HTML模板，从数据填充它动态地创建一个HTML页面模型。可以使用模板来定义任何类型的文件的结构; 它不一定是HTML！

在开发层面，Django遵循“包含电池”的理念，并提供开发人员几乎可以“开箱即用”的所有功能。因为您需要的一切都是一个“产品”的一部分，所有这些都可以无缝结合在一起，遵循一致的设计原则，并且具有广泛而且[最新的文档](https://docs.djangoproject.com/en/1.10/)。

在适用性方面，Django可以（并已被）用于构建几乎任何类型的网站 - 从内容管理系统和维基，到社交网络和新闻网站。它可以与任何客户端框架一起工作，并且可以以几乎任何格式（包括HTML，RSS提要，JSON，XML等）提供内容。您正在阅读的网站是基于Django！在内部，虽然它为几乎所有可能需要的功能（例如几个流行的数据库，模板引擎等）提供了选择，但是如果需要，它也可以扩展到使用其他组件。

在安全方面，Django通过提供一个被设计为“做正确的事情”来自动保护网站的框架来帮助开发人员避免许多常见的安全错误。例如，Django提供了一种安全的方式来管理用户帐户和密码，避免了常见的错误，比如将会话信息放在易受攻击的Cookie中（而Cookie只包含密钥，实际数据存储在数据库中）或直接存储密码而不是密码哈希。密码散列是通过[密码散列函数](https://en.wikipedia.org/wiki/Cryptographic_hash_function)发送密码创建的固定长度值。Django可以通过哈希函数运行并将输出与存储的哈希值进行比较来检查输入的密码是否正确。但是由于功能的“单向”性质，即使存储的哈希值受到威胁，攻击者也难以解决原始密码。

默认情况下，Django可以防范许多漏洞，包括SQL注入，跨站点脚本，跨站点请求伪造和点击劫持（有关此类攻击的详细信息，请参阅[网站安全性](https://developer.mozilla.org/en-US/docs/Learn/Server-side/First_steps/Website_security)）。

在可扩展性方面，Django使用基于组件的“无[共享](https://en.wikipedia.org/wiki/Shared_nothing_architecture) ”架构（架构的每个部分独立于其他架构，因此可以根据需要进行替换或更改）。在不同部分之间有明确的分隔意味着它可以通过在任何级别添加硬件来扩展流量：缓存服务器，数据库服务器或应用程序服务器。一些最繁忙的网站已经成功地缩放了Django以满足他们的需求（例如Instagram和Disqus，仅举两个）。

在可维护性方面，使用设计原则和模式编写Django代码，鼓励创建可维护和可重用的代码。特别地，它利用了不要重复自己（DRY）的原则，所以没有不必要的重复，减少了代码量。Django还将相关功能分组到可重用的“应用程序”中，并且在较低级别将相关代码分组为模块（沿[模型视图控制器（MVC）](https://developer.mozilla.org/en-US/Apps/Fundamentals/Modern_web_app_architecture/MVC_architecture)模式）。

在多平台方面，由于Django是用Python编写的，它在许多平台上运行。这意味着您不受任何特定的服务器平台的限制，并且可以在许多种类的Linux，Windows和Mac OS X上运行应用程序。

* + - 1. Flask

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在保持核心简洁并且容易扩展，它并不为使用用户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，一切都由开发和来掌握。

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 或其它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，其他的有开发者来选择扩展。

* + 1. **数据持久化**

多样性持久化考虑：使用合适的NoSQL技术存储持久化数据。



* + 1. 1. MySQL

Mysql是一款开源数据库，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。

优缺点：

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想过，减少表锁等待的时间，将大大的提高读取的性能。

* + - 1. Redis

Redis 是 key-value 开源的缓存数据库（基于内存），它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，它提供了强大的客户端 Lua、Php、Python、Ruby 等，Redis 分为五种数据类型：String，Hash，List，Set 及 Zset(sorted set)[3]。

Redis是各大公司通常采用的NoSQL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。[[1]](https://zh.wikipedia.org/wiki/Redis" \l "cite_note-1)在2013年5月之前，其开发由[VMware](https://zh.wikipedia.org/wiki/VMware)赞助。[[2]](https://zh.wikipedia.org/wiki/Redis" \l "cite_note-2)[[3]](https://zh.wikipedia.org/wiki/Redis#cite_note-3)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。[[4]](https://zh.wikipedia.org/wiki/Redis" \l "cite_note-4)

* + 1. **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 与 Memcache 的不同，它可以参与 DBA 数据结构构建，Redis 有自己的数据结构，支持自建数据结构，可以用默认冒号来做数据列。 Redis 是 key-value 开源的缓存数据库（基于内存）。

* + 1. **消息队列**

常用的消息队列有kafka，nsq等。其中kafka可以保持数据的时序性，nsq并不保持数据的时序性。在实际中应用中，nsq也可以作为kafka的备份。

kafka

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。

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patition，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

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。

1. 微服务架构及模块总体设计
   1. 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，比如说问题的名称，问题的描述，问题的图片，视频，问题的回答数，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的满足了这个需求，各个为服务之间高内聚低耦合，不同微服务模块开发交付互不影响，良好的支持了服务端的扩展性，是更加灵活的服务端架构。

1. 1. 1. **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的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，回答问题。运营邀请用户回答擅长领域的问题，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，运营同学邀请回答）

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

* 1. 整体架构分析
  2. 1. **交互式应用微服务架构设计**



图1 服务端架构设计

整个服务端架构设计如上，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。

* + 1. **存储设计**

存储设计主要分析在对应的业务需求场景下，整体的数据库架构设计模式，及选择的数据库类型。包括在使用关系型数据库存储时，MySQL的表结构的设计结构，字段的具体定义，数据量级的预计存储。并且对相应的查询场景进行数据库的优化，比如增加索引，增加Redis缓存技术。

* + 2. 1. 持久化存储设计

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的请求某些数据，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k-v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k-v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。

* + - 1. 支持高并发设计

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，数据规模海量，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k-v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redis缓存。Redis分为持久性和缓存型，可以以k-v结构缓存一层常用查询数据数据。

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，不能存储所有的数据，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。

****

* + 1. **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patition，为多个消费者消费，并且保证每个patition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，如下表所示：

|  |  |  |
| --- | --- | --- |
|  | NSQ | Kafka |
| 语言 | Go | Scala |
| 消息存储 | 内存，磁盘 | 磁盘，文件方式 |
| 顺序保证 | 无序 | 有序 |

表 NSQ vs Kafka

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，我们采用kafka。

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。

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作为消息队列来实现异步处理。

异步事件简要介绍

本文主要探讨的消息队列为kafka，作为

于此同时，我们加入一层redis来容错消息队列，如果发送kafka事件失败，我们会向redis里面添加记录，并且轮询redis，尝试重新发送kafka事件。

对于kafka消息的异步处理，针对不同业务，用不同的worker去消费kafka信息，进行请求数据处理，读写数据库等操作，实现消息的异步处理。

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，收集到kafka，针对不同下游所需要的数据，实现不同的consumer，对这一个topic开不同的consumer group。

* 1. 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，每个服务只关注与一个业务功能，独立部署，可以用不同的语言编写，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。

* 1. 1. **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，需要服务端为前端打包数据

查询服务主要用户，提供文章的查看问题，查看回答，查看草稿等基础功能。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



查询服务的请求量级是非常大的，我们在存储设计中，增加一层缓存，缓存问题信息，回答信息，降低对db的请求量。

* + 1. **提交服务总体设计**

提交服务主要用户，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

整体的架构设计如下：

****

我们加入一层redis来容错消息队列，如果发送kafka事件失败，我们会向redis里面添加记录，并且轮询redis，尝试重新发送kafka事件。

* + 1. **邀请服务的总体设计**

邀请服务主要服务于用户，提供用户对某一个问题的邀请功能，用户可以对同一问题邀请多人来回答，可以收到别人对自己的邀请。邀请服务促进了用户与应用的交流，促进潜在用户成为产出型用户。

整体架构如下：

****

邀请也将应用提交事件系统，设定邀请的专门时间，在用户邀请时，根据问题的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kafka，并且由邀请的相关worker来从kafka中接受事件来继续完成邀请。

* 1. 模块对外提供接口
  2. 1. **json 服务返回数据**

Json是Jacascrpit的一个自己，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被广泛的应用于Web应用的开发，除此之外也应用于一些菲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解打包来获取数据。

* + 1. **photobuf idl 对外接口提供**

GoogleProtocolBuffers(下文简称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.,对限定修饰符的介绍如表2-3所示。

表2-3Protobuf限定修饰符表

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。

1. 重要模块及接口详细设计
   1. 提交服务具体设计
2. 1. 1. **提交类事件定义设计**

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，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comsumer worker。

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。

为了解决上一条中提出的问题，把问题和答案的所有属性字段分为两类，第一类为状态属性，问题的状态属性为status（标识问题是否被删除）；答案的状态属性为status（标识答案是否被删除）display\_status（表示答案的展示状态）；第二类为普通属性，其他大部分属性皆为普通属性。修改状态属性会导致状态机的状态迁移。普通属性的改变对应的Update事件（UpdateAnswer, UpdateQuestion）。

具体每个核心事件对应的数据库修改内容如下：

| 事件名称 | 对应的数据库变动 |
| --- | --- |
| CreateAnswer | answer表中插入了新数据 |
| CreateQuestion | 问题的display\_status状态变为放出状态（0），show\_level变为两端都可见或者仅app可见 |
| DeleteAnswer | 删除答案，答案的status变为0 |
| DeleteQuestion | 删除问题，问题的status变为0 |
| UpdateAnswer | 对应答案的content、abstarct字段的变更 |
| UpdateQuestion | 对应问题title, content, op\_status, concern\_tags字段的变更 |
| UserCreateQuestion | question表中插入了新数据 |
| UserEditAnswer | answer\_version表中插入了新数据 |
| UserEditQuestion | question\_version表中插入了新数据 |

提交事件具体类型定义如下：

|  |  |  |
| --- | --- | --- |
| 事件编号 | 事件名称 | 事件描述 |
| 1 | CreateQuestion | 创建问题 |
| 2 | DeleteQuestion | 删除问题 |
| 3 | RecoverQuestion | 恢复问题(取消删除问题) |
| 4 | CreateAnswer | 创建答案 |
| 5 | DeleteAnswer | 删除答案 |
| 6 | RecoverAnswer | 恢复答案(取消删除答案) |
| 7 | UpdateAnswer | 更新答案 |
| 8 | UpdateQuestion | 更新问题 |

* + 1. **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

具体接口描述如下：

1. SubmitQuestion 提交问题

|  |  |
| --- | --- |
| 字段 | 字段描述 |
| Title | 问题的标题 |
| Content | 问题的描述 |
| UserID | 用户登录id |
| Uname | 用户名 |

struct SubmitQuestionRequest {  
1: required string Title; // 问题的标题  
2: optional string Content; // 问题的补充内容，web直接存储 客户端需要转 对content的处理都收敛到服务端  
3: optional i64 UserID; // 登录用户的用户id，调用方需要保证id的合法性。core user, creator id  
4: optional string Uname; // 登录用户的用户名  
5: optional i64 TagID; // 问题所属分类的分类id  
6: optional i64 GroupID; // 绑定文章的group\_id  
7: optional i64 ItemID; // 绑定文章的item\_id

}

1. OperateQuestion 修改问题

|  |  |
| --- | --- |
| 字段 | 字段描述 |
| QID | 问题id |
| Operation | 操作类型 |
| UserID | 操作的用户id |

struct OperateQuestionRequest {  
1: required i64 QID; // 问题id  
2: required OperateQuestionType Operation; // 操作类型  
3: optional i64 UserID; // 操作的用户id，调用方需要保证id的合法性。core user, creator id

}

1. SubmitAnswer 提交回答

|  |  |
| --- | --- |
| 字段 | 字段描述 |
| QID | 问题id |
| UserID | 提交答案的用户id |
| Content | 提交的答案的内容 |
|  |  |
|  |  |

struct SubmitAnswerRequest{  
1: required i64 QID; // 问题id  
2: optional i64 DeviceID; // device id  
3: required i64 UserID; // 提交答案的用户id，调用方需要保证id的合法性。  
4: required string Content; // 提交的答案的内容，格式待定  
  
}

1. OperateAnswer 修改答案

|  |  |
| --- | --- |
| 字段 | 字段描述 |
| QID | 问题id |
| AnsID | 答案id |
| UserID | 登录用户的uid |
| Operation | 操作类型 |

struct OperateAnswerRequest{  
1: required i64 QID; // 问题id  
2: required i64 AnsID; // 答案id  
3: required i64 UserID; // 登录用户的uid，调用方需要保证id的合法性  
4: required OperateAnswerType Operation;  
}

接口具体实现如下：

以提交答案为例，首先调用方按照接口约束，传入qid，userid，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）；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redis缓存，发送异步消息队列等）。

流程如下图所示：



其中我们根据userid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utf8编码。然后我们会检查参数的有无、合法性，检查检查question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，提交的方式是通过写入answer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发送答案，如果发送不成功，则向我们用作备用的一个redis添加记录，事后重复提交。其中时候提交的方式，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。

其中不同类型的事件由不同类型的worker来接收。Worker处理中，一个worker同时起多个进程。Python在语言实现上为我们提供多进程库函数，为multiprocessing.Process。

我们以多进程启动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patitions里面对应读消息 patition从0开始。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。

* + 1. **存储设计**

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：

存储文章用户手动发表的问题内容及属性信息，包括问题id，问题标题title，问题内容content，问题创建时间create\_time，问题文章扩展属性 extra 等字段。

2. 回答信息表answer：：

存储文章用户手动发表的回答内容及属性信息，包括回答id，回答状态status，回答内容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。

问题的相关字段设计如下：

|  |  |  |
| --- | --- | --- |
| 表字段名 | 数据类型 | 字段描述 |
| qid | Bigint | 问题id |
| title | varchar | 问题标题 |
| 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' |

qid 设计为bigint类型，可以容纳2^64个问题容量；title设计为varchar类型，字长最大为400个汉字/英文字符；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。

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：

question表。主要存储问题相关的属性

create table question(

    qid bigint(20) unsigned not null COMMENT '问题id，',

    title varchar(400) not null COMMENT '问题标题',

    content text not null COMMENT '问题描述',

    status tinyint(4) not null COMMENT '问题状态，0表示删除状态，1表示正常状态'，

    user\_id bigint(32) not null COMMENT '发问题的用户id',

    uname varchar(128) not null COMMENT '发问题的用户名',

    extra text NOT NULL COMMENT '扩展, 存储提交ip等,json格式',

    create\_time timestamp not null DEFAULT 0 COMMENT '问题创建时间',

    modify\_time timestamp not null ON UPDATE CURRENT\_TIMESTAMP DEFAULT 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'

答案的相关字段设计如下：

|  |  |  |
| --- | --- | --- |
| 表字段名 | 数据类型 | 字段描述 |
| ansid | bigint | 答案id |
| content | text | 回答的内容 |
| 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 |
|  |  |  |

ansid 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；abstract设计为text类型，最大存储65535字符；content 设计为text类型，最大存储65535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。

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(

     ansid bigint(20) unsigned not null COMMENT '答案id，64位id',

     content text not null COMMENT '回答的内容HTML',

     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',

     uname varchar(128) not null COMMENT '回答用户的用户名',

     create\_time timestamp not null DEFAULT 0 COMMENT '答案创建时间',

     modify\_time timestamp not null ON UPDATE CURRENT\_TIMESTAMP DEFAULT CURRENT\_TIMESTAMP COMMENT '修改时间',

    primary key (ansid)

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

* + 1. **缓存设计**

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redis缓存，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设计如下

|  |  |  |
| --- | --- | --- |
| 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。

在查询的时候，请求首先请求到cache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。

具体实现逻辑如下：

1. 读取cache数据，记录没有命中的qid/ansid

keys = makekey(qid)

Data= redis\_client.mget (keys)

Result = Json.dumps(Data)

1. 去数据库中查询未命中id

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

1. 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

1. 回写redis

for key, item in res:

keys = makekey(key)

value = json.loads(item)

redis\_client.setex(key, value, expireTime)

其中setex是 [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。Cache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。

* 1. 查询服务
  2. 1. **查询服务接口设计**

提交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. mgetQuestionInfo查询问题

|  |  |
| --- | --- |
| 字段 | 字段描述 |
| QidList | 问题id列表 |

struct MgetQuestionInfoReq{  
1: list<i64> QidList, // 问题id列表  
}

|  |  |
| --- | --- |
| 字段 | 字段描述 |
| QID | 问题id |
| Title | 问题的标题 |
| UserID | 发表问题的登录用户id |
| Uname | 发表问题的登录用户名 |
| CreateTime | 发表问题的时间 |

表 返回问题对应字段

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

struct QuestionData{  
1: i64 QID, // 问题id  
2: string Title, // 问题的标题  
3: i64 UserID, // 发表问题的登录用户id  
4: string Uname, // 发表问题的登录用户名  
5: i64 CreateTime, // 发表问题的时间  
}

1. mgetAnswerInfo 查询答案

|  |  |
| --- | --- |
| 字段 | 字段描述 |
| AnsidList | 回答id列表 |

struct MgetAnswerInfoReq{  
1: list<i64> AnsidList, // 回答id列表  
}

|  |  |
| --- | --- |
| 字段 | 字段描述 |
| AnsID | 回答id |
| Content | 回答内容 |
| UserID | 回答的用户id |
| Uname | 回答的用户名称 |
| CreateTime | 回答的时间 |

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

struct AnswerData{  
1: i64 AnsID, // 回答id  
2: string Content, // 回答内容，html模板数据  
3: i64 UserID, // 回答的登录用户id  
4: string Uname, // 回答的登录用户名称

5: i64 CreateTime, // 回答的事件，unix时间戳  
}

接口具体设计如下：

以查询答案为例，首先调用方按照接口约束，传入ansid答案id list（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。

流程如下图所示：



* + 1. **服务go化**

后续随着对服务性能的要求提高，重新将查询服务进行了go话。核心go话逻辑如下：

首先在dal.go 定义了newdal mget等方法 ，dal的结构如下：

type Dal struct {

   ClientGetter cache.CacheGeter

   //Client       cache.CacheCaller

   CallName   string

   Prefix     string

   Format     string

   Version    string

   ExpireTime int32

   DbGet      DbFetchHandler

   Marshal    Marshal

   Unmarshal  Unmarshal

}

mget的函数实现逻辑如下：

先从cache里面拿key，其中key的设计如下：

func (d \*Dal) makeKey(id int64) string {

   return fmt.Sprintf(d.Format, d.Prefix, id, d.Version)

}

mcData, err := client.GetMultiValue(keys)

在用key从redis中拿到数据后，进行解json，使用的是go的Unmarshal函数。我们开发很多应用的时候，最后都是要输出JSON数据串，那么如何来处理呢？JSON包里面通过Marshal函数来处理，对应的Unmarshal就是解json。

综上，逻辑如下：

查cache

cache没有的话 查db

把查到是db数据回写到redis

返回result result = cache查询 + 漏网的db查询

写回redis srting类型的

items := make(map[string]interface{})

value, err := d.Marshal(data[id])

items[key] = value

存到redis里面，并且设置过期时间

err := client.SetValue(items, d.ExpireTime)

所以回写逻辑，是在goquery查数据中，mget中实现的。

* 1. 邀请服务
  2. 1. **邀请服务接口设计**

1. Invite 邀请接口

|  |  |
| --- | --- |
| 字段 | 字段描述 |
| QID | 邀请的问题 |
| UserIDList | 被邀请人 |
| FromUserID | 邀请人 |

struct InviteRequest{

1: i64 QID, //邀请的问题

2: optional list<i64> UserIDList, //邀请的人

3: optional i64 FromUserID, //谁邀请的

}

|  |  |
| --- | --- |
| 字段 | 字段描述 |
| InviteResult | 邀请是否成功 |
| InviteResultTips | 错误信息 |

struct InviteResponse{

1: map<i64, i32> InviteResult, //邀请是否成功

2: optional map<i64, string> InviteResultTips, // 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\_userid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）。

流程图如下：



* + 1. **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user\_id，邀请人from\_uid，邀请时间create\_time，邀请状态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，邀请时间create\_time，content，回答创建时间create\_time等。（qid ，user\_id）二元组唯一 记录单一问题单一用户的邀请结果

邀请表设计：

邀请的相关字段设计如下：

|  |  |  |
| --- | --- | --- |
| 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；abstract设计为text类型，最大存储65535字符；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 `invited\_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',

`op\_source` tinyint(4) NOT NULL DEFAULT '3' COMMENT '答案操作来源，1表示用户操作，2表示审核操作，3表示运营操作，默认值是3',

`invite\_status` tinyint(4) DEFAULT NULL COMMENT '0 表示问题审核待邀请， 1表示已邀请， 2表示问题未通过不发起邀请，3表示因为问题被合并而不发起邀请',

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

`invite\_reason` text COMMENT '邀请理由',

`extra` text COMMENT '邀请需要维护的额外信息，服务负责直接set k-v',

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下：

|  |  |  |
| --- | --- | --- |
| 表字段名 | 数据类型 | 字段描述 |
| id | bigint | 自增id |
| qid | bigint | 问题id |
| uid | bigint | 邀请用户id |
| recv\_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 |
| newest\_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 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；newest\_recv\_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。

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',

`media\_id` bigint(20) unsigned NOT NULL COMMENT 'pgc的media\_id',

`recv\_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，0表示删除状态，1表示正常状态',

`newest\_recv\_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：

相关字段定义如下：

|  |  |  |
| --- | --- | --- |
| 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 '17' COMMENT '17 表示邀请结果为none， 具体邀请数字信息参见wiki',

,

`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';

1. 测试
   1. 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。



* 1. 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，习惯撑起为吞吐率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。

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。

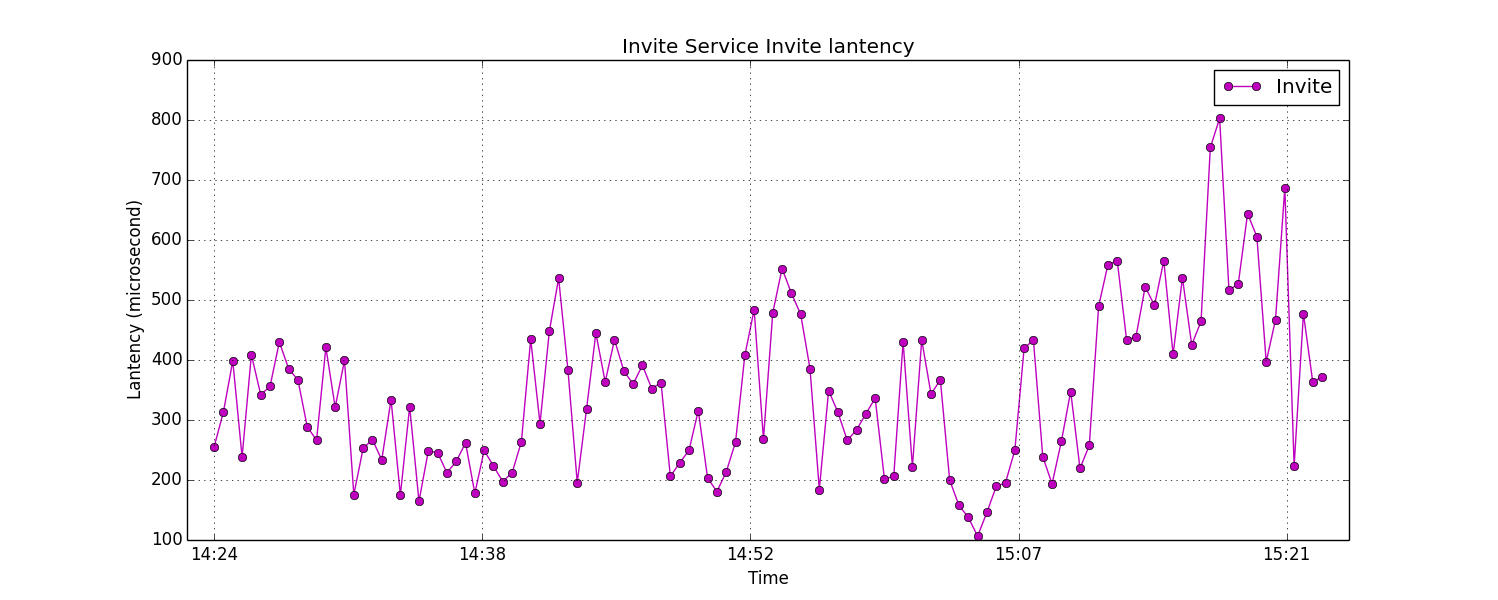
延时我们采取两种统计方式，一种是平均延时，一种percent 99的方式。

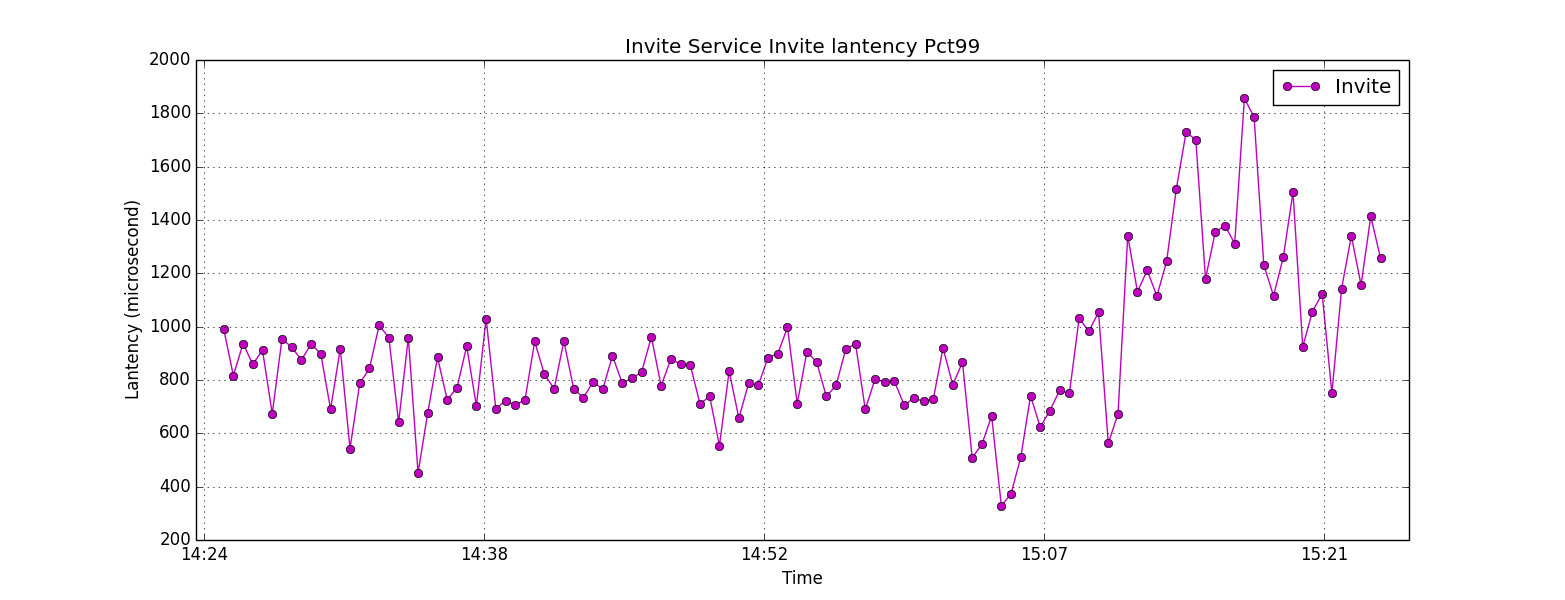
平均延时为，我们先对每台机器的单位时间请求延时做平均，在对所有机器在该时间点的平均时延做取和平均，这样得到的也就是所有请求的平均耗时。

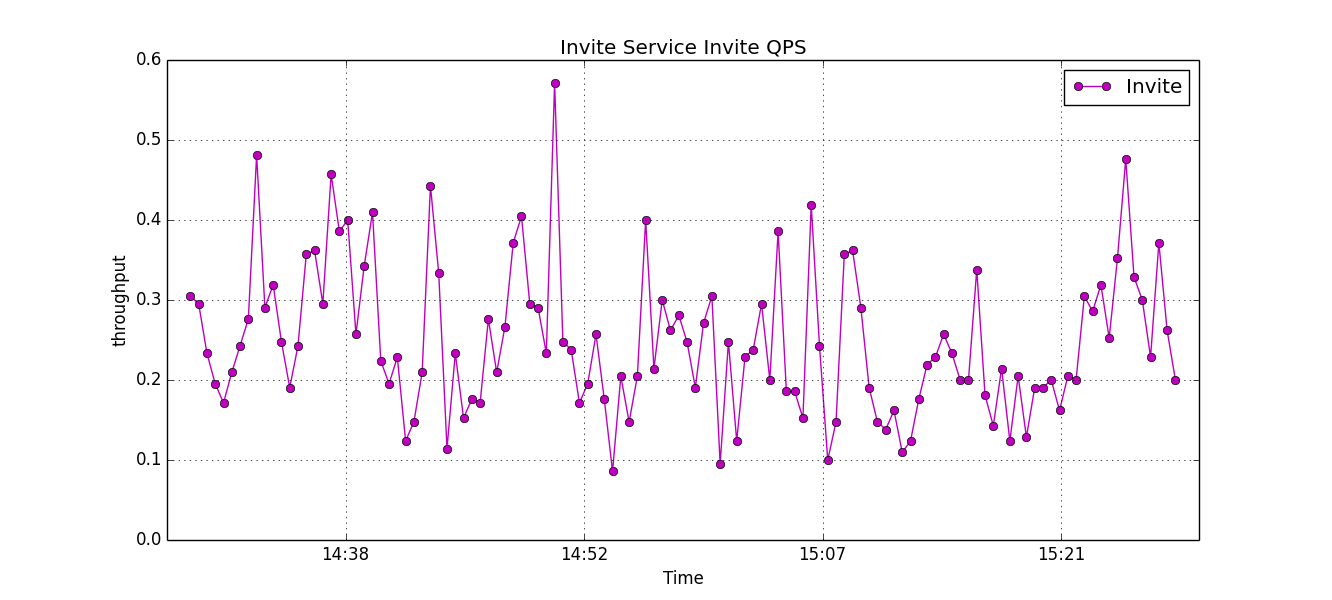
Percent 99的延时为，我们将每台机器的单位请求延时时间，从小到大排序，选取第99的数据结果点，表示99%的结果都小于这个数，再对所有机器加和求平均，这样能反馈出实际最长的请求时延。

邀请服务：

Invite接口成功率延时如下：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。如下图可见，邀请服务的调用平均时延在600毫秒左右，最高达到800毫秒。

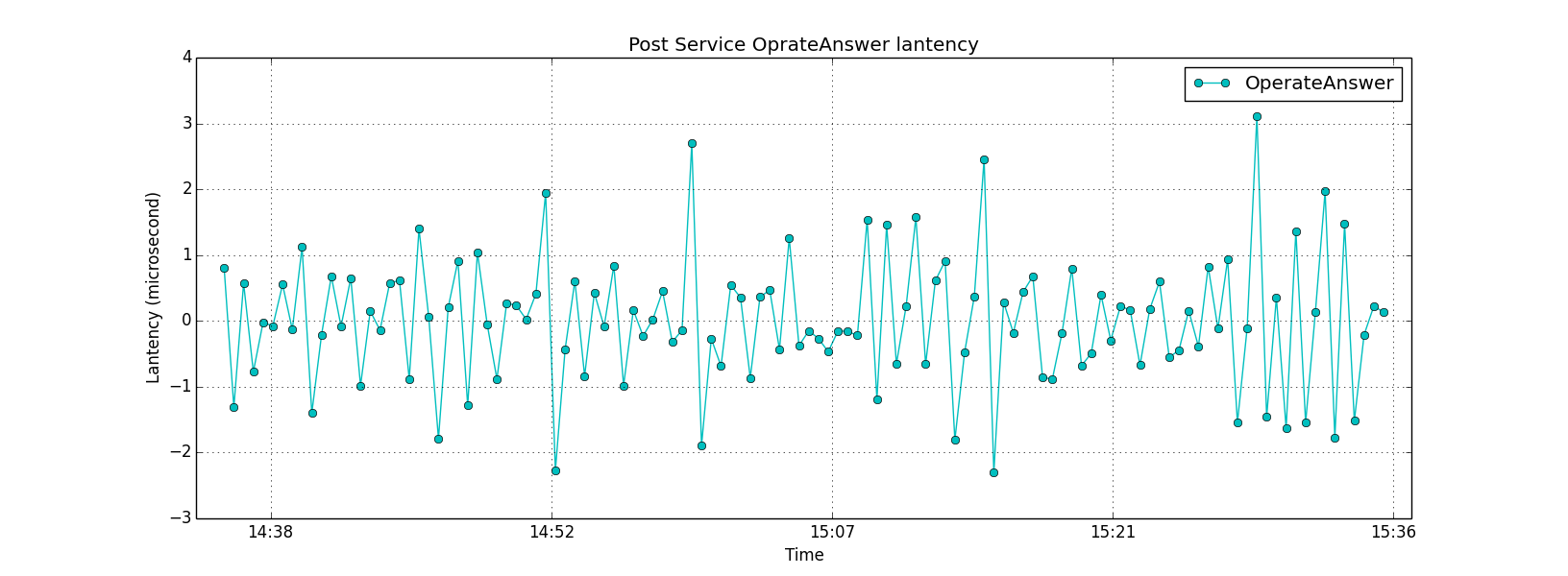


我们将一台机器的所有请求进行排序，取99%的分位点，再将所有机器求平均，得到pct99的数据。如图下所示。可见对99%的调用请求，可以99%的服务调用时延在1200毫秒一下，该小时段内的延时峰值为1800毫秒。

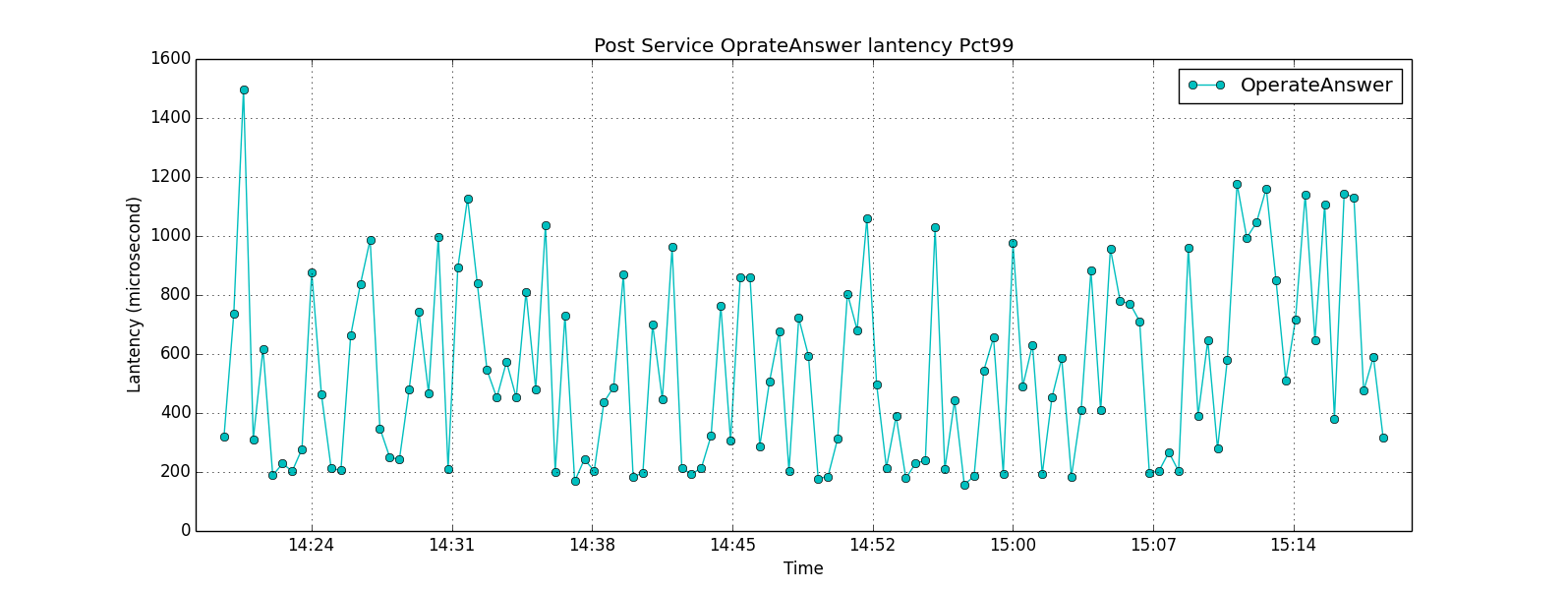
该时间段内的请求量如下，平均qps为每秒0.5的请求量

提交服务：

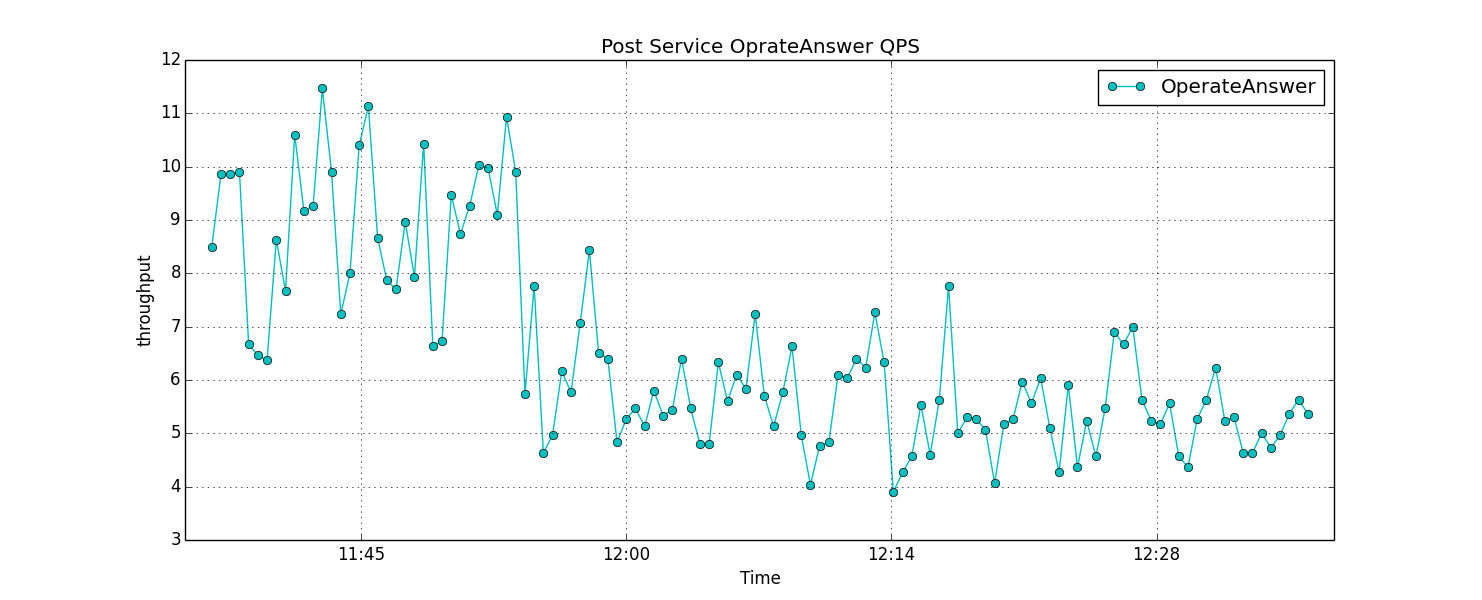
用户提交类的信息行为也相对随机，所以选取了随机某小时的数据。下图是用户修改答案操作的服务接口调用延时。



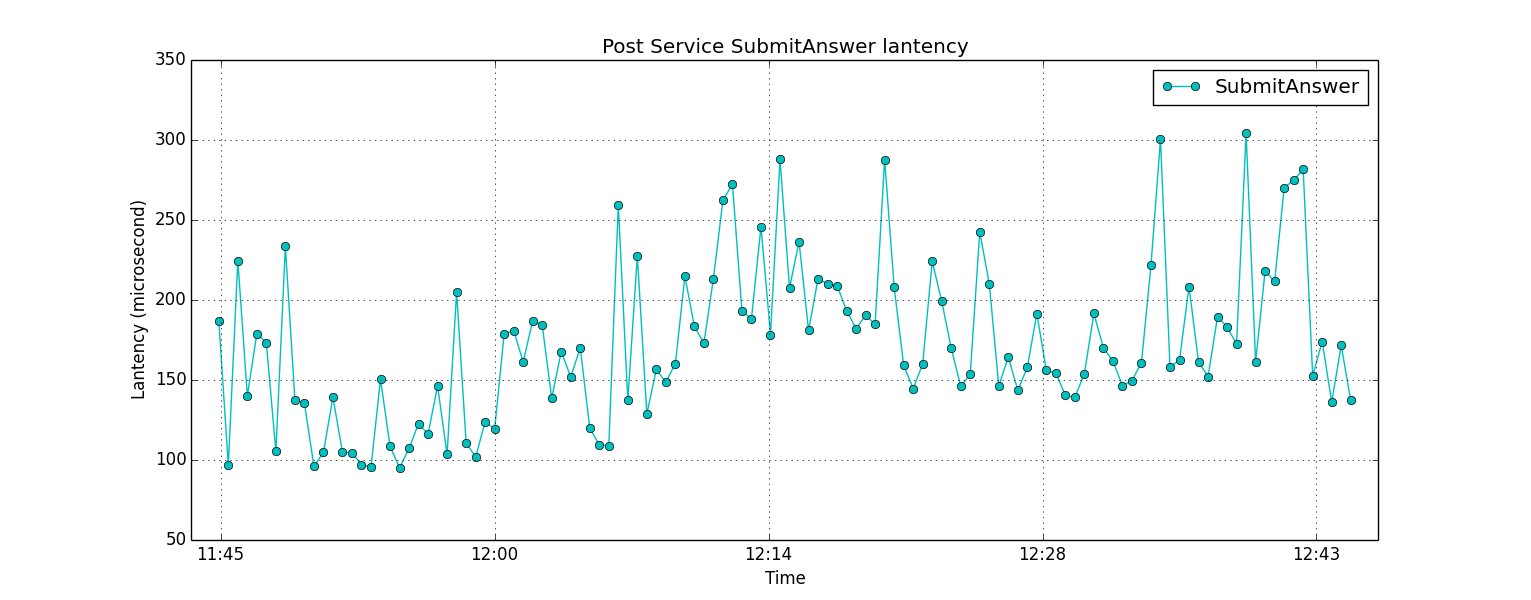
同时，我们选取数据的99%分位点，获得更覆盖绝大部分请求的延时结果。99%的请求延时在1000毫秒以下，峰值在1500毫秒。



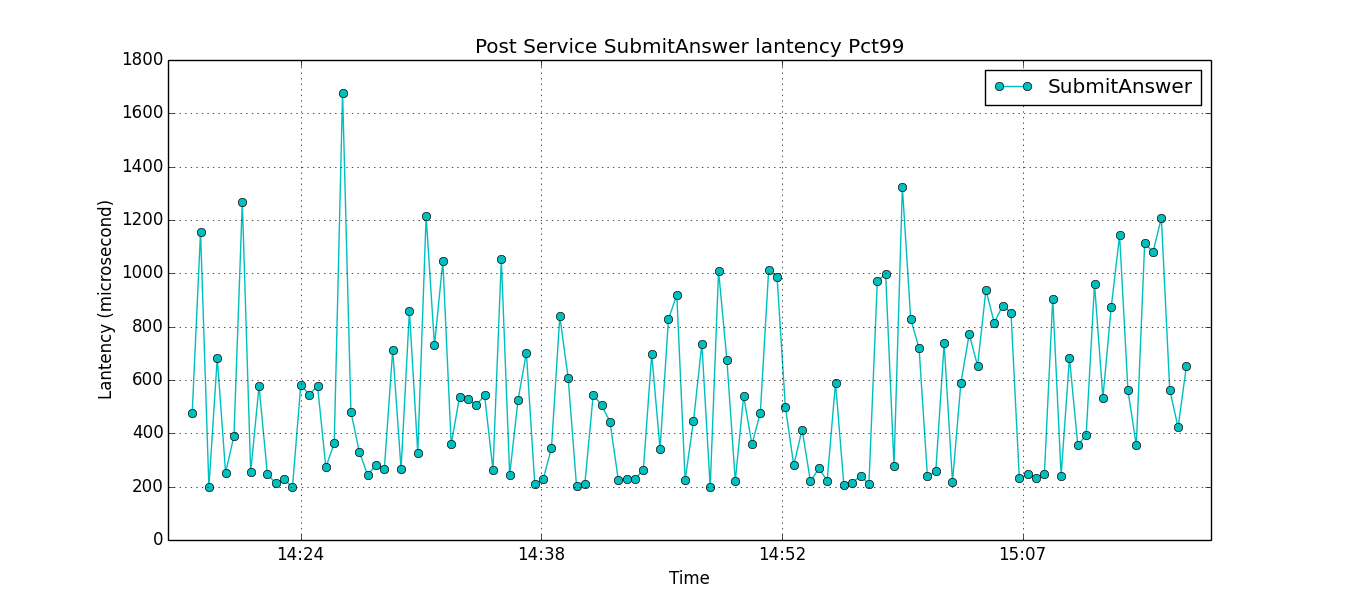
该时间段内的请求量如下，平均qps为每秒8的请求量。

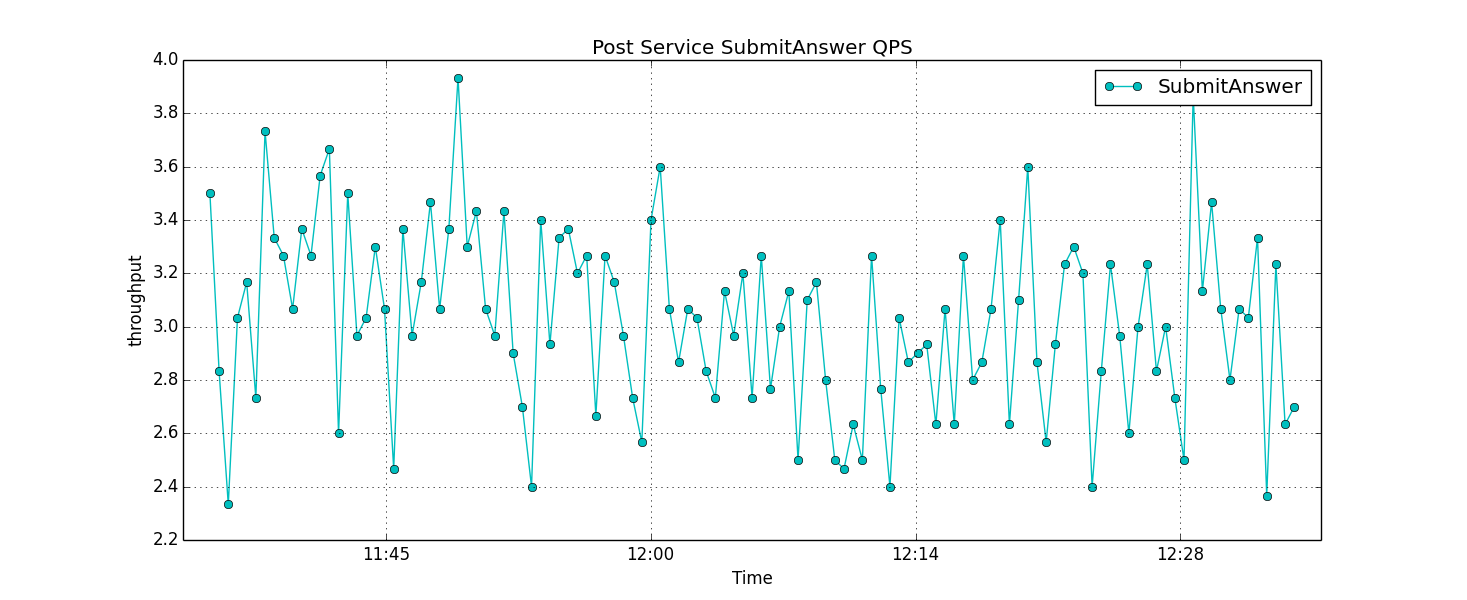


下图是用户提交答案操作的服务接口调用延时，平均延时在250毫秒左右，最高平均延时达到300毫秒。



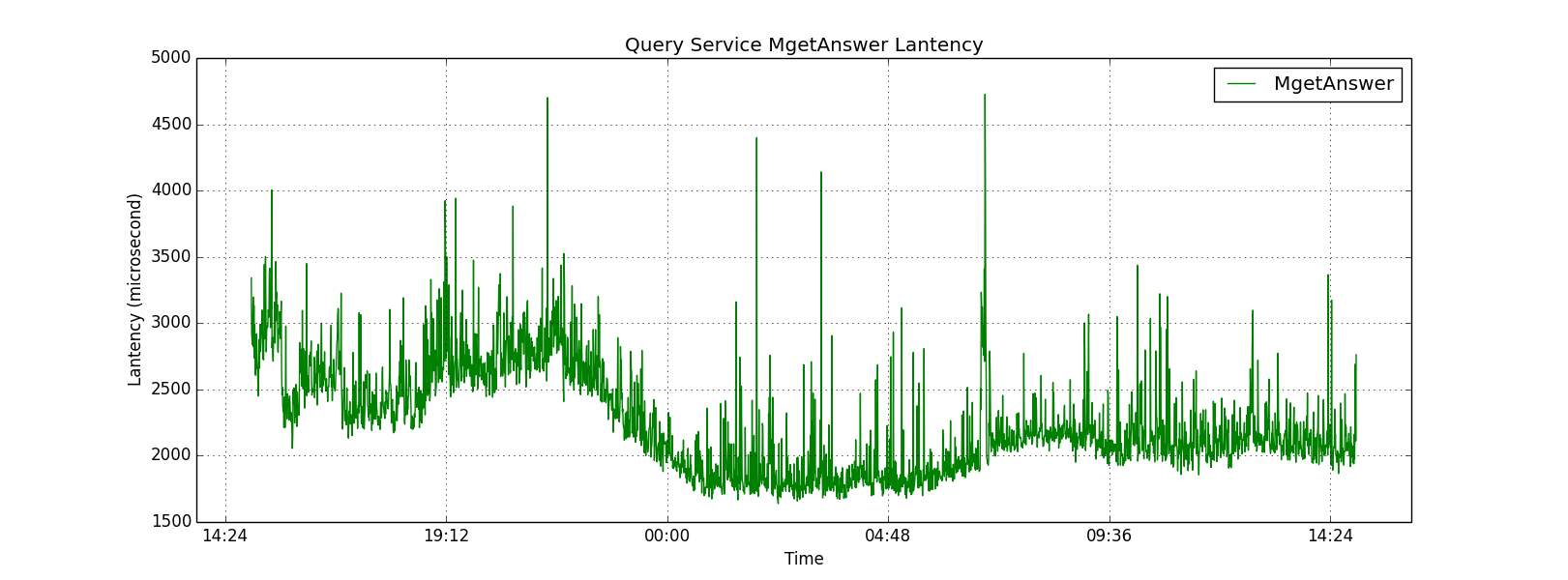
同时，我们选取数据的99%分位点，获得更覆盖绝大部分请求的延时结果。99%的请求延时在1200毫秒以下，峰值在1700毫秒。



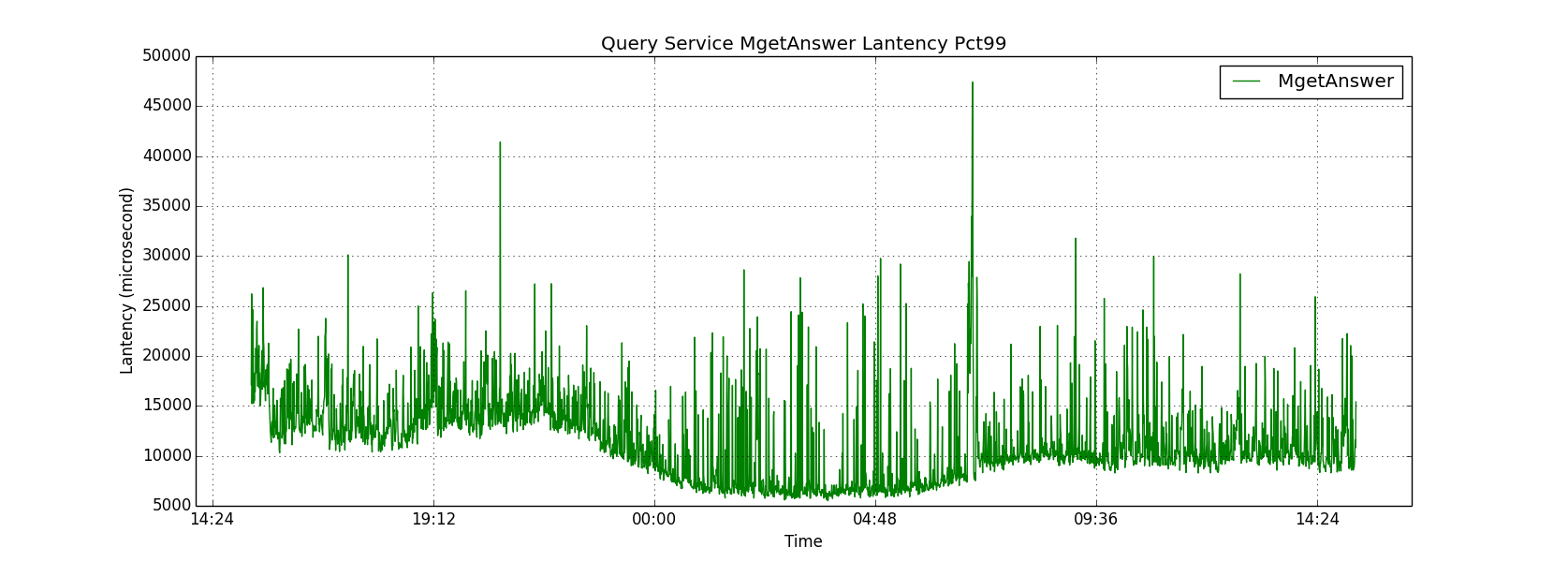
该时间段内的请求量如下，平均qps为每秒3.6的请求量

查询服务：

查询答案接口成功率延时如下。对于高并发的查询请求调用，我们选取24小时时间段，来体现服务的的性能情况。如下图可见，查询答案服务接口的调用延迟平均在2500毫秒左右，最高达到4500毫秒。



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。99%的请求延时在10000毫秒以下，峰值在4500毫秒。



该时间段内的请求量如下，平均qps为每秒8000的请求量，峰值请求量为12000.

如下图可见，查询问题服务接口的调用延迟平均在1200毫秒左右，最高达到2700毫秒。

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。99%的请求延时在5000毫秒以下，峰值在25000毫秒。



该时间段内的请求量如下，平均qps为每秒3000的请求量，峰值请求量为5000.

1. 总结

上述各章阐述了在微服务的架构下，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，包括数据持久化和存储设计，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，问题表和回答表都可以支持万级的高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] 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

[2] 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

[3] 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

[4] 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

[5] 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

[6] 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

[7] 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

[8] 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

[9] 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

[10] 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

[11] 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

[12] 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

[12] 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

[13] 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

[14] 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

[15] 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

[14] 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

[15] 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

[16] 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

[17] 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

[18] 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

[19] 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

[20] 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

[21] 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

[22] 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

[23] 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

[24] http://martinfowler.com/articles/microservices.html