#### 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，这一切都催生了新的架构设计风格：微服务架构的出现。一个优秀的服务端架构尤为重要,这样的架构不仅能够在给予用户较好的体验还能为企业节省各项成本,并且功能模块有很好的扩展性，移植性。这样就能够大大加快业务开发速度,从而提高企业利润。

本文面向交互式应用,从服务端微服务策略入手,以高稳定性、高性能、高可扩展性和高可移植性为架构的设计目标,针对基于微服务的交互式服务端设计过程中的关键技术点展开研究,设计一个微服务化的服务端架构,并且对架构中的部分关键技术模块进行设计和实现。主要研究内容如下:

## 老师评语

1. 需求 2. 创新点（2，3，设计架构 设计了实现了xxxx方案 算法之类） 3.测试验证了xx
2. 后面格式要求 文献引用，每一章一个小结，总后一个总结
3. 分析交互式应用需求，数据规模量级，服务的稳定性与容灾

2)针对服务端开发技术进行相关技术分析对比,为服务端架构设计提供基础,相关技术主要有:Restful框架、数据库技术、缓存技术。

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。在此架构对微服务模块数据存储方案、数据缓存机制等几个关键技术点进行设计。

4)根据设计方案对整体的架构以及关键技术的功能实现,进行试验及数据分析。

基于微服务的架构设计能够让研发人员只需关注上层业务逻辑的实现,通过调用稳定的微服务模块，无需关注微服务模块中功能实现和数据存储过程等较为底层的实现过程,从而降低了服务端开发难度,增加了服务端的开发效率。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

#### 第一章 绪论

#### 研究背景

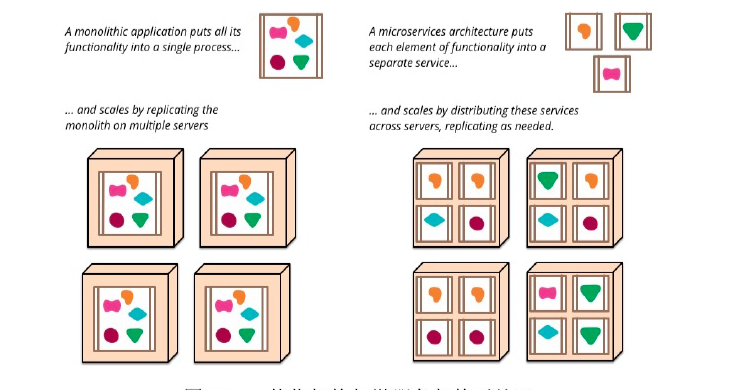
“微服务”架构是近期软件应用领域非常热门的概念。

使用传统的整体式架构(Monolithic Architecture)应用开发系统，如CRM、ERP等大型应用，随着新需求的不断增加，企业更新和修复大型整体式应用变得越来越困难；随着移动互联网的发展，企业被迫将其应用迁移至现代化UI界面架构以便能兼容移动设备，这要求企业能实现应用功能的快速上线；许多企业在SOA投资中得到的回报有限，SOA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，有时候显得力不从心。

随着应用云化的日益普及，生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（IaaS，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，一个大型复杂软件应用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，各个微服务之间是松耦合的。每个微服务仅关注于完成一件任务并很好地完成该任务。

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。

#### 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。

交互型系统是指以与用户交互为主要目的而开发的应用系统。如各种移动应用、微信、微博等等。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

本课题针对使用微服务架构进行交互式应用开发，提出了一种高可用性的服务端设计方案。

本课题针对大量实时更新的数据状态信息与数据信息，设计了基于 Redis 内存数据库与 MySQL 数据库的高可用数据管理方案，可以保证对海量、实时数据的高效存储，并且提供高可用性的保证。

本课题针对交互式应用，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。

#### 国内外研究现状

#### 各章安排

1. 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，引出了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 微服务架构关键技术研究

该章从微服务的设计原则出发，研究了针对微服务架构的服务端开发设计方法，并且对服务端所应用到的语言，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 高可用的交互式应用服务端微服务模块功能实现

该章描述了整个微服务模块的功能实现过程，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。

第二章 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，但并非所有微服务架构必须具备这些特性。

2.1.1 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，在应用架构设计中，通常将通用的模块以服务的方式抽离出来，切分成可以独立部署升级的微服务单元模块。

2.1.2 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间通讯的相关业务逻辑/智能放在组件端点侧而非放在通讯组件中，通讯机制或组件应该尽量简单及松耦合。RESTful HTTP协议和仅提供消息路由功能的轻量级异步机制是微服务架构中最常用的通讯机制。

2.1.3 无状态原则

“去中心化”数据管理(DecentralizedData Management)：微服务架构倡导采用多样性持久化(PolyglotPersistence)的方法，让每个微服务管理其自有数据库，并允许不同微服务采用不同的数据持久化技术。

2.1.4 独立部署

基础设施自动化(InfrastructureAutomation)：云化及自动化部署等技术极大地降低了微服务构建、部署和运维的难度，通过应用持续集成和持续交付等方法有助于达到加速推出市场的目的。

2.2 web服务端关键技术分析

2.2.1 Restful

REST（Representational State Transfer）描述了一个架构样式的网络系统，比如 web 应用程序。在目前主流的三种Web服务交互方案中，REST相比于SOAP（Simple Object Access protocol，简单对象访问协议）以及XML-RPC更加简单明了，无论是对URL的处理还是对Payload的编码，REST都倾向于用更加简单轻量的方法设计和实现。值得注意的是REST并没有一个明确的标准，而更像是一种设计的风格。满足这些约束条件和原则的应用程序或设计就是 RESTful。

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：例如GET用来获取资源，POST用来新建资源（也可以用于更新资源），PUT用来更新资源，DELETE用来删除资源。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

2.2.2 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。

RPC 的主要功能目标是让构建分布式计算（应用）更容易，在提供强大的远程调用能力时不损失本地调用的语义简洁性。 为实现该目标，RPC 框架需提供一种透明调用机制让使用者不必显式的区分本地调用和远程调用。 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。

2.2.3 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，相比其他语言经常使用英文关键字，其他语言的一些标点符号，它具有比其他语言更有特色语法结构。

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有上百种Web开发框架，有很多成熟的模板技术，选择Python开发Web应用，不但开发效率高，而且运行速度快。

常见的python web框架有flask，django等。

2.2.3.1 Django

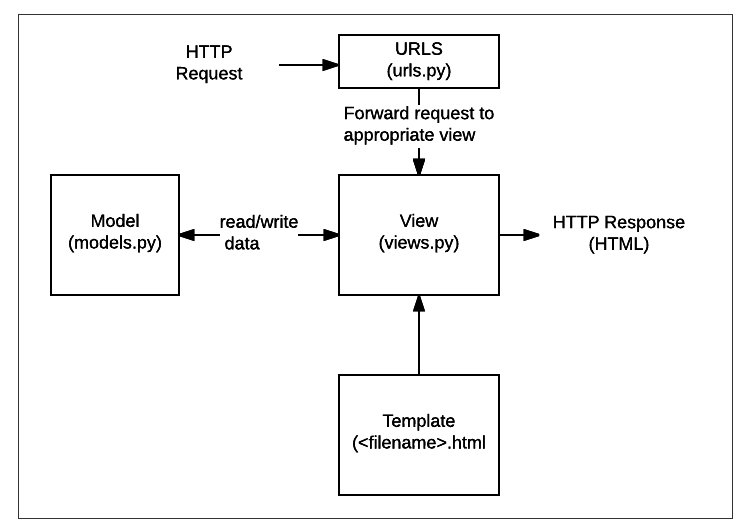
Django是一个高级的Python Web框架，可以快速开发安全和可维护的网站。使得可以专注于编写应用程序，而无需重新开发。它是免费和开源的，拥有繁荣和积极的社区，伟大的文档和许多免费和付费支持的选项。

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。

2.2.3.2 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。这并不表示我们需要把整个 Web 应用塞进单个 Python 文件（虽然确实可以），也不意味着 Flask 在功能上有所欠缺。微框架中的“微”意味着 Flask 旨在保持核心简单而易于扩展。Flask 不会替使用者做出太多决策，比如使用何种数据库。而那些 Flask 本身所选择的例如使用何种模板引擎，则很容易替换。除此之外的一切都由可由开发者掌握。

默认情况下，Flask 不包含数据库抽象层、表单验证，或是其它任何已有多种库可以胜任的功能。然而，Flask 支持用扩展来给应用添加这些功能。众多的扩展提供了数据库集成、表单验证、上传处理、各种各样的开放认证技术等功能。Flask 也许是“微小”的，但它完全可以在需求繁杂的生产环境中投入使用。

随着代码库逐渐壮大，开发者仍可自由把握项目的设计决策。Flask 不包含数据库层，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。Flask 建立了 Werkezug 和 Jinja2 的桥梁，前者实现一个合适的 WSGI 应用，后者处理模板。 Flask 也绑定了一些通用的标准库包，比如 logging 。Flask 的思想是为所有应用建立一个良好的基础，其余的一切都取决于开发者和扩展。

2.2.4 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。



2.2.4.1 MySQL

MySQL是一个[关系型数据库管理系统](https://baike.baidu.com/item/%E5%85%B3%E7%B3%BB%E5%9E%8B%E6%95%B0%E6%8D%AE%E5%BA%93%E7%AE%A1%E7%90%86%E7%B3%BB%E7%BB%9F)，由瑞典MySQL AB 公司开发，目前属于 [Oracle](https://baike.baidu.com/item/Oracle) 旗下产品。MySQL 是最流行的[关系型数据库管理系统](https://baike.baidu.com/item/%E5%85%B3%E7%B3%BB%E5%9E%8B%E6%95%B0%E6%8D%AE%E5%BA%93%E7%AE%A1%E7%90%86%E7%B3%BB%E7%BB%9F)之一，在 WEB 应用方面，MySQL是最好的 RDBMS (Relational Database Management System，关系数据库管理系统) 应用软件。

MySQL是一种关系数据库管理系统，关系数据库将数据保存在不同的表中，而不是将所有数据放在一个大仓库内，这样就增加了速度并提高了灵活性。

MySQL所使用的 SQL 语言是用于访问[数据库](https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E5%BA%93)的最常用标准化语言。MySQL 软件采用了双授权政策，分为社区版和商业版，由于其体积小、速度快、总体拥有成本低，尤其是[开放源码](https://baike.baidu.com/item/%E5%BC%80%E6%94%BE%E6%BA%90%E7%A0%81)这一特点，一般中小型网站的开发都选择 MySQL 作为网站数据库。

事务

索引

2.2.4.2 Redis

针对比较容易出现大量更新和读取操作的实时状态信息来制定其存储方案。这类实时更新的状态数据的特点是每次更新的数据量不大，并不需要很大的存储空间来处理单个更新的数据，但是其更新的频率较快而且更新的总体次数较多，所以需要一种相对较为简单并能够提供较高的数据读取性能的存储机制。同时这些数据通常是以键值对的形式进行存储与更新，一般以状态的类型为键，状态的具体数据值为值，所以我们选择键值对存储的方案。

键值对存储是一种对数据的最简单的组织方式。通常只提供对数据的获取、存储、删除等操作，不像一般的关系型数据库可以执行复杂的 SQL 查询语言，所以采用键值对存储非常适合实时更新的状态这样的不涉及过多业务逻辑的场景。而且键值对存储一般采用基于内存的存储机制，专注于数据的高效存储与索引，并且由于其结构的简单性能够带来比较好的分布式扩展和高容错性的优势，这比传统的处理大量结构数据的关系型数据库具备更好的读写性能。

基于内存的键值存储方案采用将数据和索引都缓存在内存中的机制，以计算机内存资源来换取实时数据处理的性能，利用操作系统内存高速的读写速度提供高性能的读写，并且避开了访问传统关系型数据库时磁盘 I/O 速度问题及并发量不够的问题。

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#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#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#cite_note-4)

Redis的外围由一个键、值映射的[字典](https://zh.wikipedia.org/wiki/%E5%85%B3%E8%81%94%E6%95%B0%E7%BB%84)构成。与其他[非关系型数据库](https://zh.wikipedia.org/wiki/%E9%9D%9E%E5%85%B3%E7%B3%BB%E5%9E%8B%E6%95%B0%E6%8D%AE%E5%BA%93)主要不同在于：Redis中值的类型不仅限于[字符串](https://zh.wikipedia.org/wiki/%E5%AD%97%E7%AC%A6%E4%B8%B2)。值的类型决定了值本身支持的操作。Redis支持不同无序、有序的[列表](https://zh.wikipedia.org/wiki/%E4%B8%B2%E5%88%97_(%E6%8A%BD%E8%B1%A1%E8%B3%87%E6%96%99%E5%9E%8B%E5%88%A5))，无序、有序的[集合](https://zh.wikipedia.org/w/index.php?title=%E9%9B%86%E5%90%88_(%E6%8A%BD%E8%B1%A1%E8%B3%87%E6%96%99%E5%9E%8B%E5%88%A5)&action=edit&redlink=1)间的交集、并集等高级服务器端原子操作。

Redis的持久化

所谓数据持久化就是就是将数据放到断电后数据不会丢失的设备中,Redis的数据持久化有两种方式,第一种方式是快照方式(Snapshotting),第二种是追加命令方式(Append Only File)。快照方式的主要原理就是在某个时间点把内存中的所有数据的快照保存一份到磁盘上。在条件达到时通过Fork—个子进程把内存中的数据写到一个临时文件中来实现保存数据快照。在所有数据写完后再把这个临时文件用原子函数rename(2)重命名为目标rdb文件。这种实现方式充分利用Fork的copy on write。追加命令方式的主要原理是进程每次将收到的写操作命令以追加的方式写到同一个文件AOF中。而当加载的时候,通过重新播放即可得到数据集原有的状态。AOF以增长方式进行存储的,所以每次写的信息比快照少很多,只与单位时间内的写操作数目成正比,Redis主进程是在将写操作命令放到通往AOF文件的缓冲区之后,才对客户端的请求做出回应。

2.2.5 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）。

Redis 支持内存存储中的各种类型的 value，有字符串型（String）、链表型（List）、集合型 (Set)和有序集合型 (Zset)，多种数据类型是其重要的优势地位。各种数据类型具有 push 功能、pop 功能、add 功能 remove 功能更可以交集并集和差集乃至更丰富的功能组合。在数据类型的基础上，Redis 支持多种多样的排序方式[4]。与 Memcached 排序方式相同，Redis 为了保证效率，将数据全部存储在内存中缓存（备份、快照会存到硬盘中）。Redis 按周期将新的数据写入磁盘或把变更数据写入追加的记录文件中，在这个基础上实现了主从同步。

Redis 是 key-value 开源的缓存数据库（基于内存）。官方性能测试结果：set操作每秒 110000 次，get 操作每秒 81000 次。

* + 1. 消息队列

nsq kafka 为啥用kafka

使用分布式消息队列的优点，主要包括以下几个方面：

1）解耦

在项目启动之初来预测将来项目会碰到什么需求，是极其困难的。消息系统在处理过程中间插入了一个隐含的、基于数据的接口层，两边的处理过程都要实现这一接口。这允许你独立的扩展或修改两边的处理过程，只要确保它们遵守同样的接口约束。

2）冗余

有些情况下，处理数据的过程会失败。除非数据被持久化，否则将造成丢失。消息队列把数据进行持久化直到它们已经被完全处理，通过这一方式规避了数据丢失风险。许多消息队列所采用的"插入-获取-删除"范式中，在把一个消息从队列中删除之前，需要你的处理系统明确的指出该消息已经被处理完毕，从而确保你的数据被安全的保存直到你使用完毕。

3）扩展性

因为消息队列解耦了你的处理过程，所以增大消息入队和处理的频率是很容易的，只要另外增加处理过程即可。不需要改变代码、不需要调节参数。扩展就像调大电力按钮一样简单。

4）灵活性 & 峰值处理能力

在访问量剧增的情况下，应用仍然需要继续发挥作用，但是这样的突发流量并不常见；如果为以能处理这类峰值访问为标准来投入资源随时待命无疑是巨大的浪费。使用消息队列能够使关键组件顶住突发的访问压力，而不会因为突发的超负荷的请求而完全崩溃。

5）可恢复性

系统的一部分组件失效时，不会影响到整个系统。消息队列降低了进程间的耦合度，所以即使一个处理消息的进程挂掉，加入队列中的消息仍然可以在系统恢复后被处理。

6）顺序保证

在大多使用场景下，数据处理的顺序都很重要。大部分消息队列本来就是排序的，并且能保证数据会按照特定的顺序来处理。［Kafka保证一个Partition内的消息的有序性；nsq不保证消息的顺序性］

7）缓冲

在任何重要的系统中，都会有需要不同的处理时间的元素。例如，加载一张图片比应用过滤器花费更少的时间。消息队列通过一个缓冲层来帮助任务最高效率的执行———写入队列的处理会尽可能的快速。该缓冲有助于控制和优化数据流经过系统的速度。

8）异步通信  
 很多时候，用户不想也不需要立即处理消息。消息队列提供了异步处理机制，允许用户把一个消息放入队列，但并不立即处理它。想向队列中放入多少消息就放多少，然后在需要的时候再去处理它们。

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。

第三章 微服务架构及模块总体设计

**3.1 服务端架构设计需求分析：**

1)高承载能力

服务端的高承载能为即为承载多人同时在线的能力,也就是服务端处理并发事务的效率,为了达到高承载能力的要求服务端架构就需要具有高效的进程间通信、高效的逻辑处理和高效的存储效率。

2)可扩展性

在架构设计中为了达到可拓展性的目的,核心的工作是需要设计通用性较强的各类通用模块，如：不同类型微服务功能模块，数据存储模块和数据缓存模块等一系列能够支撑大部分业务的通用模块,这样的设计能够大大减少重复开发工作。

3)可移植性

一个优秀的服务端架构不仅仅要能满足单一项目的特殊需求,并且还能适用于同一领域甚至不同领域的系统需求,从而实现开发效率的提高和开发成本的降低。

4)低耦合

交互式应用的服务端是一系列功能模块构成的集合体,各个模块之间如果耦合度按高将会导致后期开发难度增大,所以服务端架构各模块之间低耦合将会提高架构的性能和开发效率

本文低耦合的设计目标主要是通过对应用的业务进行归类和模块划分,各个模块之间尽量减少业务交叉和重复,模块之间的通过远程调用通信模块进行通信,不同微服务模块享有不同的数据存储设计。

3.1.1 微服务拆分设计

微服务拆分设计有以下几个原则：

**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。

3.2 整体架构分析

3.2.1 交互式应用微服务架构设计

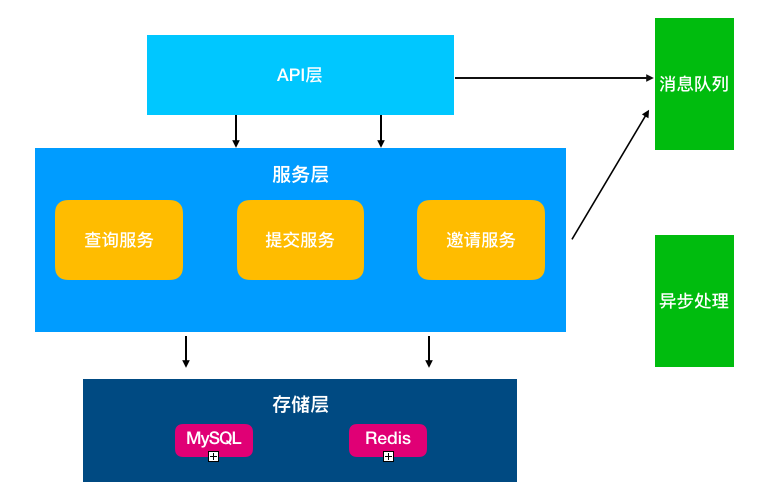


图1 服务端架构设计

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。

微服务模块划分

3.2.2 存储设计

应用场景，什么情况下会查询哪些？表？数据？索引？

什么情况下会先查缓存

图！

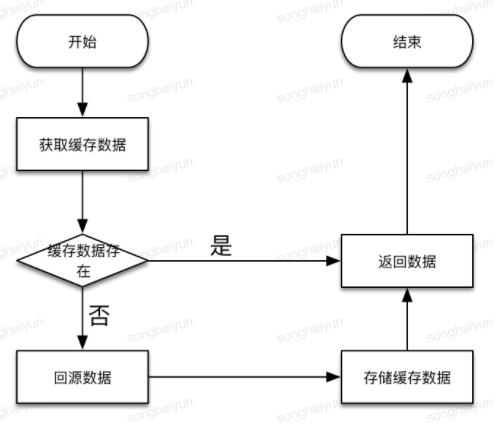
3.2.2.1mysql做数据存储

其中在大访问量的db查询前，我们会增加一层redis缓存。

缓存回源逻辑

主库mysql作为数据提交，同时同步到从库，删除缓存。缓存从从库去回源数据。

缓存是源数据的子集及其聚合，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。

****

3.2.2.2 redis做数据的实时性存储

1.Key 命名规则

Key 采用字母+数字的形式 避免多个应用使用同一个 key 的时候互相覆盖，主项目中的 key 采用 GJ#[UID] 例:GJ12345678

2.字段命名规则

主项目中的 key 内包含多个字段 代表多个应用 目前给每个应用预留的字段取值范围为 100(实际使用中字段不应该超过 10 个左右不然使用哈希的意义不大了)。

3.2.3 异步消息队列

**消息的异步处理**

**我们向kafka发送事件，由worker去接事件处理请求。**

**如果发送kafka失败，会向redis里面添加记录，并轮询redis，尝试重新插入kafka**

**3.3 微服务模块详细设计**

**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。**

**设计原则**

**1. 限界上下文：一个由显式边界限定的特定职责**

任何一个给定的领域都包含多个限界上下文，每个限界上下文中的东西分成两部分，一部分不需要与外部通信，另一部分则需要。每个上下文都有明确的接口，该接口定义了它会暴露哪些模型给其他的上下文。如果你需要从一个限界上下文获取信息，或者向其发起请求，需要使用模型和它的显式边界进行通信。

一般来讲，**微服务应该清晰地和限界上下文保持一致**。

由上可知，明确限界上下文是划分出微服务的关键所在。而**为服务找到合适的限界上下文就要求你必须了解一个领域**。服务的界限划分错误，可能会导致不得不频繁地更改服务间的协作，而这种更改成本很高，最后反而得不偿失。所以花时间了解系统是做什么的，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的。

#### ****隐藏内部实现细节****

**为了使一个服务独立于其他服务，最大化独自演化的能力，隐藏实现细节至关重要**。这一点实际和“围绕业务建模”息息相关。但另一点值得注意的是，**服务还应该隐藏它们的数据库，以避免陷入数据库耦合**。这对于单块系统来说，也通常是一个大的耦合点。就此延伸，我们可能需要考虑对数据库进行分离，并且要关注一个非常严肃的问题就是事务。事务可以保证一些事情要么都发生，要么都不发生。一个事务可以帮助我们的系统从一个一致的状态迁移到另一个一致的状态。在使用单块表结构时，所有的创建或者更新操作都可以在一个事务边界内完成。分离之后，这种好处就没有了，所以必须考虑如何“换个思路”保证系统状态的一致性。以下是几种思路：

* 对关联或失败的操作在之后进行尝试或重试，以在未来的某个时间达到**最终一致性**；
* 拒绝整个操作，做手动回退。如果已经提交了的事务，则考虑发起**补偿事务**来抵消之前的操作。但如何该补偿事务还是失败了，在这种情况下，要么再次重试，要么使用一些后台任务来清除这些不一致的状态；
* 还有一种方法是使用**分布式事务**。分布式事务会横跨多个事务，然后使用一个叫作事务管理器的工具来统一编配其他底层系统中运行的事务。一个分布式事务会保证整个系统处于一致的状态。唯一不同的是，这里的事务会运行在不同系统的不同进程中，通常它们之间使用网络进行通信。处理分布式事务常用的算法是两阶段提交。在这种方式中，首先是投票阶段。在这个阶段，每个参与者会告诉事务管理器它是否应该继续。如果事务管理器收到的所有投票都是成功，则会告知它们进行提交操作。只要收到一个否定的投票，事务管理器就会让所有的参与者回退。

需要对业务进行**拆分**。当业务量大了以后，特别是当不同的功能耦合在一起的时候，任何一个地方的改动都是非常困难的，必须对业务进行拆分，拆分的策略有两种：

* 横向拆分。按照不同的业务域进行拆分，例如订单、商品、库存、号卡资源等。形成独立的业务领域微服务集群。
* 纵向拆分。把一个业务功能里的不同模块或者组件进行拆分。例如把公共组件拆分成独立的原子服务，下沉到底层，形成相对独立的原子服务层。这样一纵一横，就可以实现业务的服务化拆分。

企业想要实施微服务架构，经常问到的第一个问题是，怎么拆？如何从单体到服务化的结构？第二个问题是拆完后业务变了增加了怎么办？另外，我们想要改变的系统往往已经成功上线，并有着活跃的用户。那么对其拆分还需要考虑现有的系统运行，如何以安全最快最低成本的方式拆分也是在这个过程中需要回答的问题。

**用户场景分析**

**普通浏览用户 浏览问题，赞同或者关心此问题，去查看回答**

**有生产内容习惯的潜在种子用户 浏览问题，赞同回答，或者关心此问题或者回答问题 –》继续浏览其他问题-》回答其他问题**

**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。**

交互式应用应有的交互功能，包括提交功能，查询功能，用户功能。

为了服务之间相互解耦合，独立，将交互式服务端主要服务划分为分3个微服务类型

基础服务： 提供查询服务

提交服务：提供提交服务

用户服务：提供邀请服务

微服务模块之间的关系如下



3.3.1查询服务详细设计

3.3.1.1用户场景分析

查询服务主要功能

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。

文章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（feed主页/个人主页）

2. 根据回答ansid列表，查询回答信息（feed主页/个人主页）

3.3.1.2 架构设计

存储方案

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。

**缓存设计**

增加一层cache，缓存问题信息，降低对db的请求量。

3.3.2 提交服务的详细设计

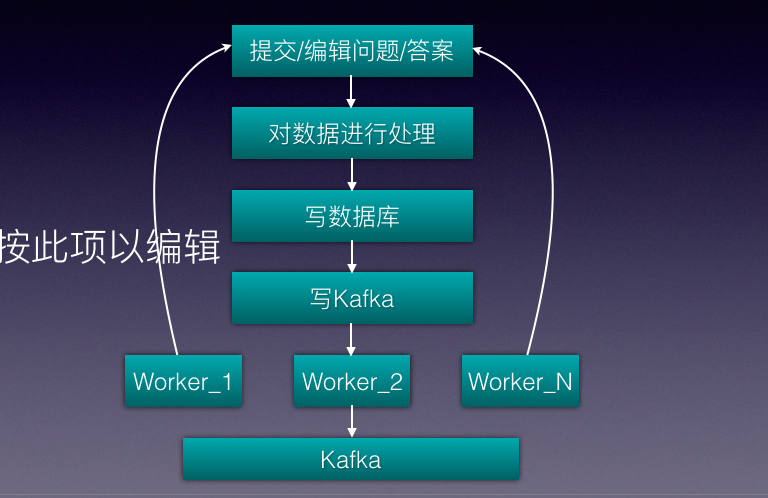
3.3.2.1用户场景分析

用户创建问题 & 用户创建答案

用户编辑问题 & 用户编辑答案

运营创建问题 & 运营编辑答案

3.3.2.2 架构设计



详细设计

创建答案



**根据userid获取用户详情（RPC调用）**

**转码，将内容转成utf8编码**

**内容过滤**

**检查参数有无及合法性**

**检查问题状态，检查回答频率，检查用户状态**

**提交数据，写answer表，**

**删除问题及用户相关的缓存**

**向kafka发送提交事件**

**如果发送失败，向redis添加记录，事后重复提交，轮询redis，尝试重新插入kafka**

3.3.3 邀请服务的详细设计

3.3.3.1用户场景分析

**邀请服务详细设计**

**用户场景分析**

**用户邀请用户，单一邀请**

**运营邀请用户，单一/批量邀请**

3.3.3.2 架构设计

**整体架构**

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/运营邀请只能邀请一次

  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**

**详细设计**

**获取用户信息，进行强用户校验：**

# 用户非封禁 非黑名单

# 每个用户每个问题最多邀请10人

# 每个用户每个问题最多被邀请10次

# 邀请用户之前没回答过这个问题

检查问题状态

写information/result表，发送kafka事件

worker接收事件，进行异步处理，发送邀请。

3.4 模块对外提供接口

3.4.1 json 服务返回数据

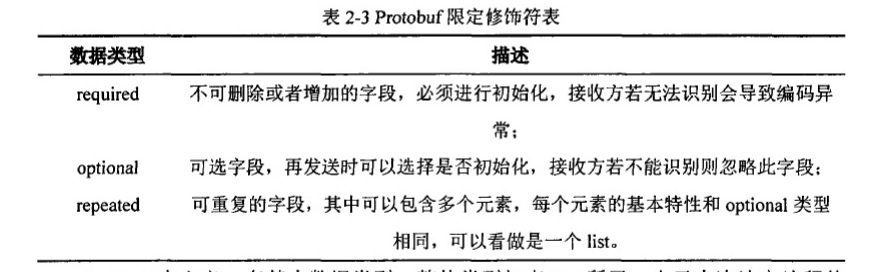
[JSON](https://baike.baidu.com/item/JSON)([JavaScript](https://baike.baidu.com/item/JavaScript) Object Notation, JS 对象标记) 是一种轻量级的数据交换格式。它基于 [ECMAScript](https://baike.baidu.com/item/ECMAScript) (w3c制定的js规范)的一个子集，采用完全独立于编程语言的文本格式来存储和表示数据。简洁和清晰的层次结构使得 JSON 成为理想的数据交换语言。 易于人阅读和编写，同时也易于机器解析和生成，并有效地提升网络传输效率。

3.4.2 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,主要应用领域是数据结构的序列化和反序列A[45""6]。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[481。

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

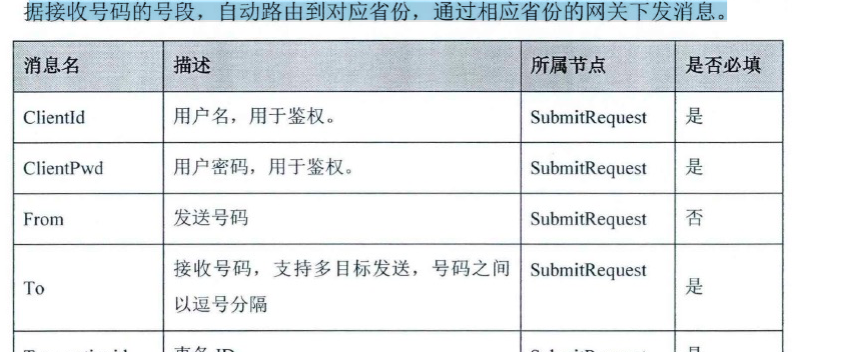


第四章 重要模块及接口具体设计

基于微服务的接口设计.

接口描述

客户端可以将短消息通过消息分发系统下发,下发时指定发送方号码、接收方号码、发送内容。发送失败时,消息分发系统返回状态码。该接口可以根据接收号码的号段,自动路由到对应省份,通过相应省份的网关下发消息。



实现方案

1)客户端发送POST请求到达消息分发系统,以XML格式发送请求。

2)消息分发系统解析消息后,系统根据用户名和密码进行登录,如果登录不成功,终止操作。否则进行下一步。

3)调用消息指定的通道组或系统配置的发送通道采用同步发送方法进行下发,采用多目标方发送方式,只有一个号码时,放到To字段中,多个号码时放到Bcc中。

4)消息分发系统根据发送结果构造HTTP响应消息返回给客户端,以状态码标示消息发送结果,同时包含发送的号码数量、失败号码数量等相关信息。

4.1 提交服务

目的

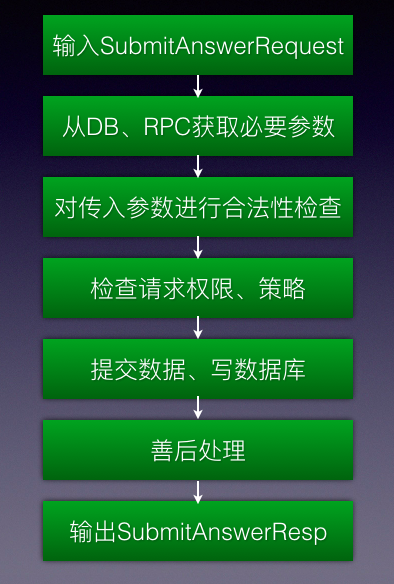
梳理逻辑、问题答案的状态转变。

接口功能清晰，代码逻辑简单维护性好。

明确事件系统的定义，整理目前所有的异步事件，确定事件定义的原则。

解决对kafka的容灾问题，避免kafka事件发送失败影响业务。

4.1.1 整体架构



4.1.2 事件类型定义

事件系统的定义

异步事件用于通知关心问答数据变更的下游以及完成部分的异步调用逻辑等，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）。

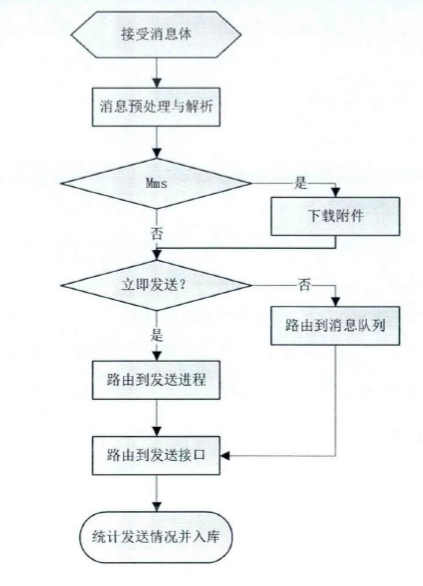
事实上，消息消费者关心的并不是事件本身，而是某个字段或者某几个字段发生了怎样的改动。一次数据库状态的变更，一定可以用一个三元组来表示（old\_value, new\_value, extra\_info），其中，old\_value表示状态变更前，答案或者问题在数据库中的各种状态属性的值，new\_value表示状态变更后，答案或者问题在数据库中的各种状态属性的值，extra\_info则表示状态变更的其他额外信息，如事件发生的时间，事件操作的来源等等。

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）。

由post服务保证事件中的每个字段都是可信的，非可信字段直接给null。

具体每个核心事件对应的数据库修改内容如下：

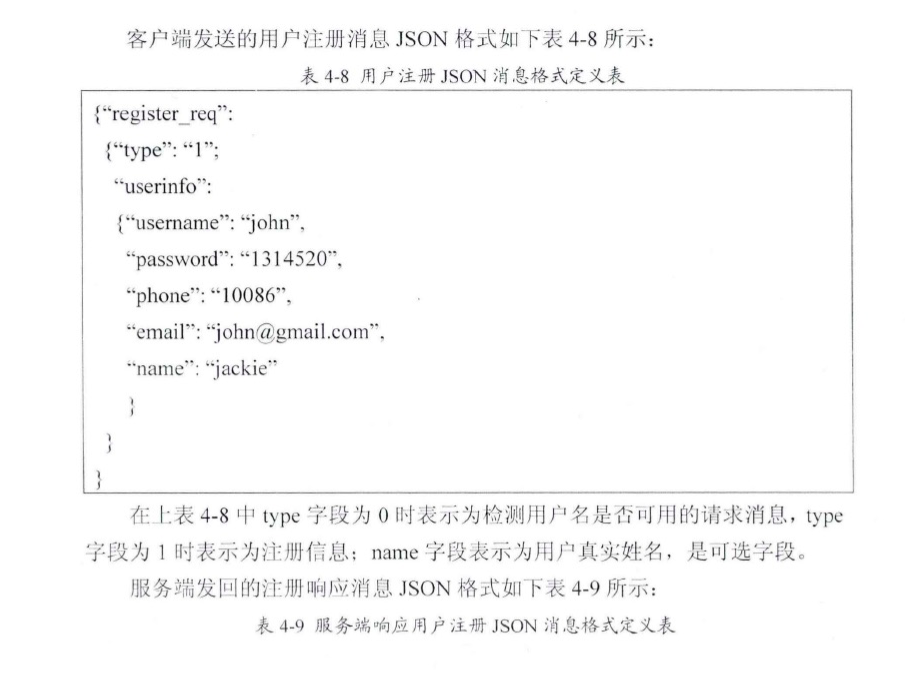
| 事件名称 | 对应的数据库变动 |
| --- | --- |
| CreateAnswer | answer表中插入了新数据 |
| CreateQuestion | 问题的display\_status状态变为放出状态（0）， |
| DeleteAnswer | 删除答案，答案的status变为0 |



4.1.4 数据库写表关系

4.1.5 缓存处理

用户与推送注册消息定义



提交服务流程



4.2 查询服务

4.2.1 整体架构

4.2.2 事件类型定义

4.2.3 异步处理

4.2.4 数据库写表关系

4.2.5 缓存处理

4.2.6 灾备设计

事件灾备系统的设计

利用多个消息中间件

提交服务采用类似于方案2的做法，kafka + redis的方案

即消息发送给kafka失败后发送redis，由worker（backup\_worker）不断的取redis，重新发送kafka

4.3 邀请服务

4.3.1 整体架构

4.3.2 事件类型定义

4.3.3 异步处理

4.3.4 数据库写表关系

4.3.5 缓存处理

第五章 测试

5.1测试环境

本次测试均在项目所使用的云服务器上进行,在程序中模拟玩家在线并发送测试所需的数据包,服务端性能数据监控记录工具使用nmon,该工具主要用于查看cpu的使用率、巧存利用率、网卡IO速率和磁盘读写10速率。

测试虚拟机的配置如下:

系统发行版本:Centosrelease6.8

CPU:Intel(R)Xeon(R)CPUE5-26xxv,2394MHz

内存:4G

网卡速度:lOOOMb/s

5.2测试方法

5.2.1 功能测试

5.2.2 qos测试

本文定义的 QoS 属性主要考虑那些从服务提供商端和服务用户端能够截取到的客观服务质量。因此质量属性的可度量和可计算性是本文选择的 QoS 属性的关键。从用户角度而言,主要替用户解决其所关心的如何能够取得一个综合评价值最好的服务。从服务端角度而言,对服务的信息直接进行拦截能够防止单纯通过静态扩展WSDL 文件而造成弄虚作假的情形。而且激烈的排名制度也有利于服务的优胜劣汰从而促进服务质量不断提高。

如今,不同的组织如 ITU,ETSI,ITIF 等对于 QoS 给出了不同的定义[27]。如 IETF主要所指的 QoS 实际上是指网络层(IP 层的 QoS),相应的,其所指的 QoS 性能参数也是指 IP 层的 QoS 性能参数,包括丢包率,吞吐量,时延变化等。而 ITU 则更倾向于实施推荐标准,如各个行为主体之间签订 SLA 协议,并根据该协议来约束服务

质量等。

在文献[28]中,罗列了 Web service 的非功能性属性中可能出现的质量标准:

系统级 QoS:可用性,可靠性,队列长度,处理容量,带宽,错误率,吞吐量,内存容量

任务级 QoS:达到成功完成的可能性,等待时间,响应时间,完成任务所需要的内存吞吐量环境的 QoS:成功完成的可能性,端到端响应时间,可感知的下载速度,用户站点的友好性

商业的 QoS:交易丢失率,Web 站点的收益,每页的收益,每次交易金额,访问设备的成本。

针对如此众多的服务质量,有些服务质量属性带有很大的主观性,比如用户站点的友好性等等,不同用户具有不同的感知性。为此本文主要针对系统应用级的服务QoS,选用几个具有代表性的 QoS 属性进行阐述,不考虑服务的领域特性。

(1). 性能(Performance):根据 W3C 的定义,Web service 的性能代表了一个服务请求能在多快的时间被完成,其中吞吐量被定义为在一个给定时间段内完成的服务请求数[29]。在本文所描述的客观的服务评价中,性能是最重要的指标。就数据采集而言,性能可以分成与时间相关的性能和与调用次数相关性能。时间相关的性能包括它可以从以下几个指标来衡量:响应时间,延迟,等待时间等,它衡量了从服务请求者发送请求开始,至得到响应的过程中的经历。

延迟定义为请求从客户端发送服务端之间的时间差,与用户端和服务段之间的网络因素相关。

响应时间定义为 Web 服务执行系列操作的时间,是 Web 服务对请求进行处理的时间。

等待时间定义为服务请求到达容器后至服务请求得到响应之间的时间差,该时间和服务调用请求多少有很大关系,服务请求时间是本文第三章服务预测的基础。

服务请求数定义为单位时间内服务端接受到的客户端请求数,单位时间内服务请求数量巨大,服务可能来不及响应,导致服务请求失效。服务请求数与服务等待时间相关较大。在服务端并发处理的服务数量有限时,过大的服务请求数量直接导致请求更长的等待时间。

吞吐量定义为服务在单位时间内处理的客户端请求数量。

(2). 可靠性(Reliability):可靠性经美国标准化研究所定义为产品在规定条件和规定时间内完成规定功能软件不引起系统失效的概率,该概率是系统输入和系统使用的函数,也是软件中存在错误的函数[30]。根据该定义我们可以看到它主要集中于时间和条件。可以从下列几方面来衡量:

可靠度定义为在时间 t 内的函数 R(t)=P{在时间[0,t]内按规定条件运行成功},其中 p 为一概率。

平均失效时间定义为软件在规定的条件下随即使用时,在规定的时间内,软件产品的寿命单位总数与失效总次数之比。

平均修复时间:发生失效后修复软件系统所需要的平均时间。

可用性是指 Web 服务是否存在或是否能正常被使用,表示服务可用的可能性大小。服务可用性的值高表示其能被正常使用的概率大,服务可用性的值较小,则反应了服务可能经常处于不稳定状态。对于服务的可用性可以由公式:qav s = Ta S /θ进行计算,Ta S 指服务 s 在最近连续时间θ内能够被访问的总时间

有效率定义为对这个服务发出的调用请求中,得到服务器响应的请求数目占请求综述的比例,在服务端处理服务的过程中,服务处理请求所需要的分布种类将会影响到服务请求排队的数量。至于服务如何处理排队中的请求,不是本文考虑的重点。

(3). 安全性通过验证涉及到的各方、对消息加密以及提供访问控制来提供机密性和不可抵赖性[31]。由于 Web 服务调用是发生在公共的因特网上,安全性的重要性已经有所增加。根据服务请求者的不同,服务提供者可以用不同的方法来提供安全性,所提供的安全性也可以有不同的级别。

(4). 费用(Cost):某服务的服务价格就是服务请求者调用这次操作所必须支付的费用,一般由服务提供者直接指定。

机器数

对于服务质量 QoS 数据,我们选择了:响应时间,等待时间,处理时间,吞吐量,有效率,费用,安全性,成功调用的服务次数。



QoS 信息的采集和反馈

打点.. 日志.. xxx还真的不知道

QoS 信息采集是用于采集 Web service 在运行时与 QoS 相关的信息,QoS 信息采集分为请求端与运行端质量信息采集。请求端是指在使用 Web 服务的用户端进行质量信息采集,而运行端质量信息采集是指在运行 Web 服务的应用服务器平台上进行信息采集。在请求端与运行端进行信息采集有利于全面获得Web 服务运行状态信息

http流图…

QoS 信息汇聚与存储

这个不知道啊 .. 问下架构 or 看文档

上述是服务评价和推荐体系的基础。主要构造了服务评价的质量模型体系,该服务质量模型能很好的适应于服务质量属性的扩展,并且本文在该模型的基础上给出了服务质量的指标的定义,并规范了取得的服务质量数据的预处理过程。

qps大约多少

反应时间多少

高峰期多少

并针对分析…

第六章 总结：

微服务架构的优点：

每个服务都比较简单，只关注于一个业务功能。

微服务架构方式是松耦合的，可以提供更高的灵活性。

微服务可通过最佳及最合适的不同的编程语言与工具进行开发，能够做到有的放矢地解决针对性问题。

每个微服务可由不同团队独立开发，互不影响，加快推出市场的速度。

微服务架构是持续交付(CD)的巨大推动力，允许在频繁发布不同服务的同时保持系统其他部分的可用性和稳定性。

微服务架构的缺点：

 微服务的一些想法在实践上是好的，但当整体实现时也会呈现出其复杂性。

运维开销及成本增加：整体应用可能只需部署至一小片应用服务区集群，而微服务架构可能变成需要构建/测试/部署/运行数十个独立的服务，并可能需要支持多种语言和环境。这导致一个整体式系统如果由20个微服务组成，可能需要40~60个进程。

必须有坚实的DevOps开发运维一体化技能：开发人员需要熟知运维与投产环境，开发人员也需要掌握必要的数据存储技术如NoSQL，具有较强DevOps技能的人员比较稀缺，会带来招聘人才方面的挑战。

隐式接口及接口匹配问题：把系统分为多个协作组件后会产生新的接口，这意味着简单的交叉变化可能需要改变许多组件，并需协调一起发布。在实际环境中，一个新品发布可能被迫同时发布大量服务，由于集成点的大量增加，微服务架构会有更高的发布风险。

代码重复：某些底层功能需要被多个服务所用，为了避免将“同步耦合引入到系统中”，有时需要向不同服务添加一些代码，这就会导致代码重复。

分布式系统的复杂性：作为一种分布式系统，微服务引入了复杂性和其他若干问题，例如网络延迟、容错性、消息序列化、不可靠的网络、异步机制、版本化、差异化的工作负载等，开发人员需要考虑以上的分布式系统问题。

异步机制：微服务往往使用异步编程、消息与并行机制，如果应用存在跨微服务的事务性处理，其实现机制会变得复杂化。

可测性的挑战：在动态环境下服务间的交互会产生非常微妙的行为，难以可视化及全面测试。经典微服务往往不太重视测试，更多的是通过监控发现生产环境的异常，进而快速回滚或采取其他必要的行动。但对于特别在意风险规避监管或投产环境错误会产生显著影响的场景下需要特别注意。

参考文献

1. <http://martinfowler.com/articles/microservices.html)>

致谢

在论文完成之际,谨向我尊敬的导师张洪欣老师、吕英华老师、张金玲老师致以衷心的感谢!感谢张洪欣老师读研期间给我的谆谆教诲和指导，给我们创造了良好的科研和生活条件,为我的课题研究付出了诸多心血和不遗余力的指导,使得论文最终得以顺利完成。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。感谢夏翼老师，夏老师敏锐的洞察力、严谨的做事风格、高尚的为人，都将对我产生深刻的影响，并将使我受益终生。

最后我要特别感谢我的家人长期以来对我学业的支持,是他们的深切关怀热切希望支持我和鼓励着我,使我能够面对挑战、刻苦钻研。我想要是没有他们,我就不会取得今天的成绩。