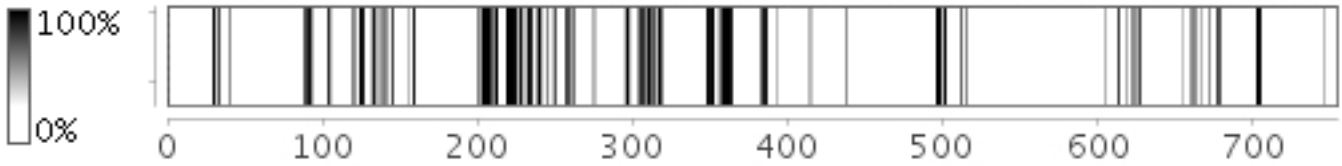
PaperPass专业版检测报告
简明打印版

比对结果（相似度）：
总　体：24 % *（总体相似度是指本地库、互联网的综合比对结果）*
本地库：11 % *（本地库相似度是指论文与学术期刊、学位论文、会议论文数据库的比对结果）*
期刊库：7 % *（期刊库相似度是指论文与学术期刊库的比对结果）*
学位库：8 % *（学位库相似度是指论文与学位论文库的比对结果）*
会议库： 1 % *（会议库相似度是指论文与会议论文库的比对结果）*
互联网：18 % *（互联网相似度是指论文与互联网资源的比对结果）*

编　号：591FE4AC3124F0CAP
版　本：专业版
标　题：自来水生产运行维护平台服务器端设计与实现
作　者：杨思璇
长　度：32036 字符(不计空格)
句子数：754句
时　间：2017-5-20 14:39:40
比对库：学术期刊、学位论文（硕博库）、会议论文、互联网资源
查真伪：http://www.paperpass.com/check

句子相似度分布图：



本地库相似资源列表（学术期刊、学位论文、会议论文）：
1. 相似度：1％篇名:《Memcached和Redis在高速缓存方面的应用》
   来源：学术期刊 《无线互联科技》 2012年9期 作者: 王心妍
2. 相似度：1％篇名:《内存数据库在点击多方通话中应用》
   来源：学术期刊 《软件》 2012年12期 作者: 刘岩 王晶 王纯
3. 相似度：1％篇名:《跨平台移动医疗应用的数据保护方案的设计与实现》
   来源：学位论文 浙江大学 2016 作者: 朱爱民
4. 相似度：1％篇名:《基于JSON的互联网异构数据整合的应用研究》
   来源：学位论文 南京邮电大学 2016 作者: 朱峰
5. 相似度：1％篇名:《传感器服务平台的设计与实现》
   来源：学位论文 中国海洋大学 2013 作者: 郭宗辉

互联网相似资源列表：

1. **相似度：3％ 标题：《从数据库管理系统选型开始 - 高山峻岭 - 博客频道...》**
   http://blog.csdn.net/gaojunling518/article/details/46516413?locationNum=5&fps=1
2. **相似度：2％ 标题：《容器集群管理工具各项对比 - V2EX》**
   http://scholar.google.com/schhp?hl=zh-CN&as_sdt=6800906a574aa9f80abcd8d4ea57e7e8
3. **相似度：1％ 标题：《addToSeturl first根据资源文档的排序获取第一个文档数...》**
   http://www.docin.com/p-1001546945-f6.html

全文简明报告：

摘 要

在绝大部分企业软件开发中，服务器的设计与开发是必要的一个组成部分，随着企业业务规模的增长， 传统的主机服务器方案由于性能瓶颈和价格高昂的原因逐渐不再被业界采纳，基于均衡负载技术的服务器集群成为一种最适用的服务器设计解决方案。 然而传统的服务器集群架构存在着模块依赖性强，部署困难，数据库无法支撑高并发访问以及开发周期过长的缺点， 使其在工业界仍然无法得到广泛与通用的技术实践。 因此本文在传统的服务器集群架构研究基础上，通过以 Docker为代表的轻量级虚拟化技术以及相应的集群管理工具， 并利用服务发现与均衡负载器，解决了服务器集群部署与弹性扩增困难的问题。 基于NoSQL数据库与Node.js平台，提供了一种高可用、高性能以及高并发的分布式集群解决方案，并以敏捷开发方式实现了标准的Restful API接口。 最后，通过对服务器的部署与监控，并基于代码测试和服务器性能分析，表明了本文所述的服务器设计方案可以较好的解决服务器集群部署困难的问题， 提高了集群整体的运算性能。

关键词： {50％： 服务器架构，均衡负载，分布式集群，容器}

第一章 绪论

{47％： 现如今，软件开发逐步发展为B/S架构与C/S架构以及其混合发展的三层架构，而不管在何种架构中，服务器都是必须的存在。} {44％： 自计算机网络出现以来，服务器就是企业软件开发中最为重要的一部分。} 随着企业用户规模的扩大，单一的主机服务器计算性能增长出现瓶颈，已经无法满足日益增长的用户请求，严重制约了企业的发展， 因此，如何开发出具备性能可弹性扩展的服务端架构，也成为业界热门的主要研究方向。

自云服务器与云计算 [2-4]的概念提出以来，便成为越来越受欢迎的企业服务器架构。 {40％： 云计算的模式是从实用计算、自主计算、网格计算和软件即服务（SaaS）[5]的概念发展而来[6]。} 本质上是一个强大的计算节点集群，通过网络的访问连接、软件和服务的组合来完成计算任务。 通过与服务器虚拟化软件相结合的方式实现分布式大型计算集群和并行处理。

1.1课题研究现状

1.1.1服务器架构研究进展

早在上个世纪90年代，服务器集群的概念在业内便被提出， {48％： V Cardellini等人[7]对于 web框架中服务器性能瓶颈提出了构建一个可动态扩展的 web服务器集群，} 通过均衡负载的方式解决计算性能的问题。 此后

，不断有人提出了新的均衡负载算法及动态扩容的方式[8-10]，如基于IP的均衡负载解决用户session问题，无状态的请求设计及基于节点性能的负载方式等。 随着业务的不断扩大，服务层代码依赖过于耦合，难以维护，2005年业界逐渐提出了面向服务架构（SOA）[11]的服务器设计概念， {100％：它可以根据需求通过网络对松散耦合的粗粒度应用组件进行分布式部署、组合和使用。} {100％：服务层是SOA的基础，可以直接被应用调用，从而有效控制系统中与软件代理交互的人为依赖性。}

2011年5月于威尼斯附近举办的一次架构师工作坊讨论中首次提出微服务架构(简称MSA)的概念。 {91％：MSA是一种分布式系统架构，它建议我们将业务切分为更加细粒度的服务，并使每个服务的责任单一，} {97％：且可独立部署，服务内部高内聚，隐含内部细节，服务之间低耦合，彼此相互隔离，} 2014年3月， Martin Fowler发表的 Microservices一文[12]真正为 MSA这一架构风格在业界正名， 也正是此文让业界对微服务有具体的认识。

然而服务器架构由于部署困难的问题，一直无法很好的用于构建服务器集群，但随着以VMWARE公司为代表的虚拟化技术的发展，业界Trieu C. Chieu[13]等人通过预先编写的虚拟机镜像，从而可以自动化地部署新的虚拟机实例，实现虚拟化的云服务器环境， 完成对服务集群的自动弹性扩展，以此便捷的构建 MSA架构的服务端。

2015年后，Red Hat公司提出以Docker为代表的容器（Container）技术[14-15]，在社区上受到热烈欢迎并且迅速发展，容器技术为应用程序提供了隔离的运行空间： {100％：每个容器内都包含一个独享的完整用户环境空间，并且一个容器内的变动不会影响其他容器的运行环境。} 通过容器彻底解决了微服务部署的问题，实现了高可扩展性的微服务架构的服务器部署[16]， {41％：并且通过容器管理技术，更为方便地实现了对服务器集群的构建与均衡负载[17]。}

1.1.2服务器基本架构

广义上的服务器集群结构基本类似，大体上由一个均衡负载器，一系列的处理服务器节点，一个数据库池组成。 对于具备动态扩容的服务器集群，通常还有一个配置子系统与服务发现子系统，并有一个定义的扩容算法所组成，见图1-1所示：

均衡负载器作为整个服务端的入口，接收用户的服务请求，并将请求按一定均衡负载算法分发给一系列的处理服务器集群节点完成请求的处理， 所有处理服务器节点共享一个数据库池，完成对数据的增删查改的需求，最后将处理结果返回给用户。 配置子系统实现了处理服务节点的自动化启动与配置，服务发现子系统完成对新启动的服务与不可用服务的探查， 并将其加入到集群网络或将其从集群网络中剔除。

1.1.3服务器集群的特点

（1）可弹性伸缩性:

服务器集群规模可以弹性扩容，根据资源需求可以随时加入或删除计算节点以扩增集群规模，增大服务集群的计算能力，以服务更大规模的用户数目，完成更大数量的服务请求。

（2）高有效性与高可用性:

服务器集群中各个集群主机不会互相影响，独立工作，即使有部分主机发生冗机情况， 中心将自动地把不可

用主机从集群网络中剔除，剩余主机的运作不会受到影响， {44%：仍能正常处理用户的服务请求，仅造成服务器计算能力的降低。} 并且集群调度中心将会尝试重新启动发生错误的主机，待其恢复功能后重新加入集群网络，以恢复集群的正常运行。

（3）服务高性价比:

服务器集群可以通过集群网络的方式将一系列不同型号、标准的廉价硬件设施组成一个高可用、高性能的计算中心， 代替了以往价格高昂的单一大型计算主机。 并且可以轻松地更新、维护集群中任意主机节点，在同等计算性能下，达到了很高的性价比。 {41%：在现今流行的SaaS企业云平台更进一步降低了服务器集群的使用与维护成本。}

（4）动态负载平衡:

为了保证系统中所有资源可以得到充分的利用， {41%：服务器集群通常具备一个均衡负载器将用户请求根据一定的均衡算法将请求平衡地转发给集群中的节点处理，} 从而尽可能地让不同性能的计算节点都能最大程度的利用其性能资源。 {50%：并且可以通过监控服务集群中所有处理服务节点的负载健康状态，动态地改变调度情况。}

1.2课题研究意义

由于服务器需要处理大量的用户服务访问请求，因此如何设计实现一个符合自身业务需求的服务器架构是一个最根本的问题。 相较于传统的单一主机服务器计算性能有效和价格昂贵的特性，云服务器通过大量主机构成集群网络， {41%：实现可伸缩性、高有效性、高可用性，通过均衡负载完成对系统硬件资源最为有效的利用。} 使用物理主机构建集群由于环境原因十分困难，并且资源受到限制，采用基于虚拟机的虚拟化技术则对资源开销过大， 使用容器技术对 MSA架构的服务器集群进行部署在业内则属于刚刚兴起的热门研究方向。

通过对服务端架构的研究，开展自来水生产运行维护平台服务器端架构设计，从而满足服务器数据处理、存储平台规模可扩展， 服务端集群计算能力能随着业务量的扩张而弹性扩展的需求，实现对大量并发访问请求的处理， {46%：解决服务器无法同时满足不同类型客户端请求的问题。} 合适的服务器架构可以降低企业的维护成本，方便地满足企业业务，适应企业用户规模的增长，因此，对服务器架构的研究具有重要的意义。

1.3内容安排

本课题其它内容安排如下：

（1）第二章介绍云服务器集群技术，主要包括容器技术、容器管理工具、服务发现及均衡负载等；

（2）第三章介绍数据库池设计，主要包括设计MongoDB数据库集群、Redis缓存；

（3）第四章介绍处理服务器实现，主要包括介绍基于Node.js实现OAuth2.0协议和Restful 接口；

（4）第五章介绍集群配置与性能分析，主要包括服务器集群和数据库集群的容器配置和集群负载性能分析；

（5）最后几部分是结束语、致谢、参考文献和附录。

第二章 云服务器集群

2.1 容器技术

{58％： 容器技术是目前实现微服务架构服务器最佳的选择之一。} {75％： 通过将应用（微服务）以及依赖包发布到一个可移植的容器中，然后Pull到任何主流的 Linux 机器上实现服务的虚拟化。} {83％： 容器完全使用沙盒机制，不依赖于任何语言、框架以及操作系统。} {84％： 基于容器封装应用（微服务），并与经过优化的强大基础架构相结合，使经过认证的应用能够在裸机系统、虚拟机和私有或公共云之间轻松部署。} {100％： 容器技术和虚拟机的功能有些相似，但性能相差很大。} {96％： 与虚拟机相比，容器技术通常可以在1秒内启动，而虚拟机启动加上应用系统的启动时间要长得多，且资源利用率高。} {83％： 另外，容器性能开销小，虚拟机通常需要额外的 CPU和内存来完成 OS的功能，这一部分占据了额外的计算资源，} 对于服务器集群来说不具备高性价比，以下图2-1为传统的 VM技术与容器技术的区别， 左为 VM技术架构，右为容器技术架构：

图1 VM与Container技术架构对比图

2.1.1 Docker容器工具

{75％： Docker是 Dot Cloud公司基于 go语言开发的一个基于 LXC的高级容器引擎，源代码托管在 Github上，并遵从 Apache2.0协议开源，} 也是目前容器技术中最有主流的代表，用户可以从 Docker Hub上 Pull经过认证的或其他个人 Push的公有容器镜像直接使用， 也可以自己通过编写 Docker file文件构建其自身的容器镜像。 使用 Docker file构建容器可以通过FROM命令在一个公有容器基础上构建自己的容器。 每个容器具备自己的计算机文件、进程、端口资源与接口，通过VOLUMES命令挂载本地文件与配置。

{57％： Docker解决了运行环境依赖问题，不再有更换运行环境后应用无法正常启动的问题。} {100％： 如果说LXC着眼点在于提供轻量级的虚拟技术，扎根在虚拟机，那Docker则定位于应用。} {100％： Docker所为人称道的portability、application-centric、versioning等等超越传统虚拟技术的优点都跟它的封装性密不可分。}

2.1.2 Docker Machine主机管理工具

Docker Machine是一种可以让用户在虚拟主机上安装Docker实例，并且通过Docker Machine命令管理主机的工具。 用户可以通过Docker Machine可以在不同平台上（不管是本地Mac或者Windows或者Azure、 Rackspace、OpenStack、Google等云平台）创建Docker主机。 利用Docker Machine命令，用户可以方便的启动、监控、重启Docker主机，配置Docker客户端与实例。 Docker Machine结合VMWARE可以作为一个方便的虚拟机管理工具，使得用户可以方便的创建以及管理虚拟机，并可通过创建一个集群桥接网络，实现不同主机间的通信。

2.1.3 Docker Compose容器编排工具

Docker Compose是一个编排并启动多个容器应用的工具。 通过Compose，用户可以编写Compose file来配置自己的应用服务，然后利用一些简单的命令，便可以创建、开始、停止、重启所有配置中的服务。 Compose如一个CI工作流一般，对于开发、测试及运维来说都是十分方便的。 用户可以在 Compose file中定义容器的镜像或者Build环境、内外端口映像、启动方式、启动命令，并且可以通过 Links命令将不同容器的地址写入相应容器的DNS中， 同时通过配置 Overlay网络，可以让容器在一个共享的网络中通信。 利用Scale命令，Docker Compose可

以轻松的在任意时刻对某个容器进行扩容。

2.2容器管理工具

{100%：很明显，容器在创建和交付应用程序的过程中有着新发展。} {100%：然而，大范围控制容器部署也会有一些并发症。} 容器肯定是跟资源相匹配的。 故障肯定是越快解决越好。 {100%：这些挑战会导致集群管理和编排的并发需求。}

{100%：集群管理工具是一个通过图形界面或者通过命令行来帮助你管理一组集群的软件程序。} {100%：有了这个工具，你就可以监控集群里的节点，配置 services，管理整个集群服务器。} {93%：集群管理可以从像发送工作到集群的低投入活动，到像均衡负载和可得性的高介入工作。} 主流的容器管理工具有 Google公司开发的工具 Kubernetes，与 Docker原生的 Docker Swarm，由于 Kubernetes相对学习成本较高， 命令与 Docker差别很大，更适用于管理容器数量巨大的引用，因此直接采用原生的 Docker Swarm来管理容器对于服务数量复杂度不高的应用是最佳的选择。

Docker Swarm是一个原生的Docker集群工具，让一系列的 Docker主机集群转化为一个单一的虚拟的Docker主机。 {100%：在一个分布式应用程序环境中，计算元素必须也是可以被分布的。} Swarm 允许你在本地聚集 Docker 引擎。 {100%：有了单个引擎，应用程序可以被扩展得更快，更有效率。} {100%：Swarm 能够扩容到 50000 个容器， 1000 个节点，同时当容器添加到集群的时候一点都不影响性能。}

再加上， Swarm 的角色相当于 Docker API。 {96%：任意可以操作Docker Daemon 的工具都可以运用Docker Swarm的力量在很多主机上进行扩容，包括了像 Flynn，Compose，Jenkins和Drone之类不同的主机。}

{66%：Swarm 遵循“swap， plug， play”的原则，意味着也可以在后端运行 Mesos 或者 Kubernetes 的时候，被用来作为前端 Docker 客户端。} Swarm 在它的核心内部是一个简单的系统： {100%：每个主机运行一个 Swarm 代理与管理员。} 管理员处理容器的操作和调度。 {100%：你可以在高可用状态下运行，它使用的是 Consul，ZooKeeper或者etcd来发送容错 events到后端系统。}

{100%：Docker Swarm的一个优点就是，它是一个本地解决办法——你可以用Docker命令来实施Docker网络，插件和数据卷。} Swarm 管理员为 leader 选举创建一些 master 和特定的规定。 这些条例实施在初级 master 故障的 event 里。 {100%：Swarm调度器以各种各样的过滤包为特色，也包括紧密性和节点标签。} {100%：过滤包能够附加容器到底层节点，资源得到更好的利用，性能得到提升。}

在网络上创建Swarm节点的第一步是Pull Docker Swarm镜像。 然后，利用Docker配置Swarm Manager与其余的节点来启动Docker Swarm。 这需要用户做到：

1、对于每个节点启动一个TCP端口用于与Swarm Manager通信；

2、安装Docker在每个节点上

3、创建与管理TLS信任证书来保证集群的安全

最佳的方式便是利用Docker Machine来完成这些，Docker Machine已经安装了Docker实例，并且可以自动地创建信任证书与其余的Docker Machine实例组成安全的集群网络。

2.3 均衡负载器

{43%：均衡负载器，英文名为 Load Balance，是一个 Apache HTTP均衡负载器，通过反向代理的方式，} {42%：它将作为一个单一的 web请求入口点，接受所有的客户端服务请求，然后根据一定的均衡算法，} {47%：将请求路由到处理服务器集群上相同的 web服务器处理实例中。} 需要说明的是： {74%：均衡负载器并不是必须的基础网络设备，而是一种性能优化设备。} {100%：对于网络应用而言，并不是一开始就需要负载均衡，当网络应用的访问量不断增长，} {98%：单个处理单元无法满足负载需求时，网络应用流量将要出现瓶颈时，负载均衡才会起到作用。} 常用的均衡负载器有Nginx，通过Nginx可以十分容器地实现多个请求路径的反向代理与均衡负载的功能，通过location的匹配实现路径的匹配，同时也可以用于静态资源的缓存。 Nginx还可以实现HTTP Response的缓存与压缩等诸多功能。

该容器由config/consul文件夹下的Docker file基于官方认证的nginx容器所构建，代码为：

{40%：通过挂载本地的nginx.conf实现对nginx的配置，完成反向代理与均衡负载的功能，其中均衡负载与反向代理的关键代码为：}

2.4 服务发现

{44%：对于微服务架构的服务器而言，服务发现功能是一个极其重要的组成部分。} 由于服务不可能是预先定义好不变的，随着服务运行中的弹性扩展，重启，转移，服务的地址与端口都可能发生改变， 原有依赖于这些服务的服务将会因此发生错误，而运行时创建的服务，也需要通过服务发现，将其地址加到均衡负载器的 DNS列表， 然后加入到均衡负载器的负载流中。 对于需要依赖这些动态创建服务的服务，也需要服务发现功能，来动态的获取可用服务的DNS列表。 常用的服务发现有Consul， ZooKeeper或者etcd以及Docker Swarm自带的服务发现功能。 本文采用Consul作为服务发现工具。

Consul具备多个组件，总体而言，它是一个用于服务发现与配置的工具，它提供以下关键的特性：

1. 服务发现： Consul客户端可以将一个服务注册，其他Consul客户端来发现提供者的服务，利用DNS或者HTTP，应用可以轻松地利用这些服务。

2. 健康检查： Consul客户端提供一定数量的健康检查，通过测试给定服务是否返回200 OK或者本地节点内存是否低于90%。 这些信息可以提供给集群监控健康，并被服务发现组件避免路由到不健康的主机

3. KV存储： {42%：应用可以利用Consul的key/value数据库存储一定目标数据，包括动态配置，特性标志等。}

4. 多数据中心： Consul支持多个数据中心，意味着用户不需要担心构建额外的抽象层来增加多个局部实例。

Consul是一个分布式的、高可用的系统，每一个节点提供了运行Consul Agent的服务，Agent就像节点自己一样负责节点服务的健康检查。 所有节点通知一个或多个的Consul服务端，Consul服务端用于存储与备份，它们会选取一个leader，通过主从备份防止数据丢失。 当用户需要用到服务发现时，可以通过任意Agent或Server查询，Agent会自动地将查询转发给Server。

通过绑定 Docker端口的 Registrator可以自动地将 Docker运行中的所有服务及相应的端口注册到 Consul Server中，{43%：随后通过在均衡负载器中启动一个 Consul Client，在有新的处理服务器加入到集群中时，} 便可动态地将新的服务地址与端口注册到均衡负载器，从而实现弹性扩容。 当有服务关闭时，健康检查功能也将该服务从均衡负载器中剔除。

编写在Nginx中运行的Consul Client的服务发现template服务，使得处理服务器集群增加或删除时，订阅通知更新Nginx的配置文件，重新配置均衡负载流：

在Nginx的Docker file中下载Consul Client Template并启动template服务以接收Consul Server对于新启动容器的通知：

第三章 数据库池设计

3.1数据库选择

在过去的企业服务端架构中，传统的关系型数据库（RDBMS）占据了市场上的绝对主流地位，例如MySQL，Oracle，SQL Server等。 由于关系型数据库基于集合代数的原理设计，用户通过SQL语言可以很方便的实现联表查询，能够满足十分复杂的业务查询功能，提供了数据的存储、访问以及保护能力。 {66%：通过对ACID原则的全面支持，即交易的原子性、一致性、隔离性与持久性，使得RDBMS具备很强的实用性与可靠性。}

然而随着互联网时代的到来，用户数目与数据容量的爆炸式增长使得传统的关系型数据库显的愈发力不从心。基于 Web的大量数据处理与分析需要要求极高的数据库查询速率，而在企业的实际应用中发现，{40%：并不是所有数据都必须实现强一致性，即时偶尔有些非重要数据发生错误也不影响系统的正常使用，} {97%：非强一致性模型以及更小的处理消耗更适合快速变化的动态环境。} 由下图3-1可以了解到，近年来非结构化的数据增长原快于结构化的数据增长，对于传统的关系型数据库而言处理这些非结构型数据是不合适的。

{65%：在这种背景需求下，非结构型的数据库（NoSQL）应运而生，NoSQL数据库相比于关系型数据库提供了更灵活的数据库模式（Schema），} {100%：其中每一个数据元素不需要存在于每一个数据实体当中。} {100%：定义更松散的数据结构会随着时间的推移而进化，因此在一些特定场景下NoSQL数据库会是更加实际的解决方案。}

{100%：NoSQL与关系型数据库的另外一个不同就是数据一致性的提供方式。} {100%：关系型数据库可以确保存储的数据永远保持一致性，而大多数NoSQL数据库产品提供了更松散的一致性方式。} {100%：事实上，关系型数据库产品已经可以提供不同级别的数据库锁、一致性与隔离性，而一些NoSQL则提供了多种一致性模型，可以支持完整的ACID。}

{100%：NoSQL解决了一些关系型数据库不能解决的问题，针对海量数据的处理更得心应手。} {89%：实际生产中数据被认为是稀疏的，不是所有元素都被填充，在实际值中还有很多的“空白空间”。} NoSQL通过基于稀疏矩阵的处理方式使得数据库处理这些数据的能力大幅增强。

{100%：尽管NoSQL在特定的数据类型上有着一定优势，但与关系型数据库相比它的劣势也是非常明显的。} {100%：比如，交易完整性、灵活索引以及查询易用性的缺失等。} {100%：此外，NoSQL还包含了四个不同的

类别，用来支持不同的应用：}

- 键值型数据库（Key-value）
- 文档型数据库（Document）
- 列式数据库（Columnstore）
- 图型数据库（Graph）

同时对于一些实时性极强，查询频率高但是数据量小的数据模型，基于文件系统的关系型数据库也不再适合，内存数据库解决了这一问题。 {100％：内存数据库有时也称为主内存数据库。} {100％：一个内存数据库主要通过内存来存储数据，这与基于磁盘的存储有所不同。} {100％：内存数据库的主要应用场景就是改善性能。} {100％：数据存储在内存介质当中，I/O延迟将得到大大削减。} {100％：因为机械硬盘的转动、寻道时间以及传输到缓存器的动作在内存中都被省去了。} {100％：内存数据库主要针对内存数据访问进行了优化，而传统数据库则是针对磁盘进行的数据访问优化。} {100％：内存数据库产品还可以减少开销，因为其内部算法通常更加简单，需要更少的CPU指令。}

3.2MongoDB数据库

{95％：MongoDB是由C++语言编写的，是一个基于分布式文件存储的开源数据库系统。} {93％：在高负载的情况下，可以添加更多的节点，可以保证服务器性能。} {100％：MongoDB旨在为WEB应用提供可扩展的高性能数据存储解决方案。} 它可以构建一个NoSQL集群，应对当前大数据量，大规模运算以及高并发的需求。{98％：MongoDB将数据存储为一个文档，数据结构由键值(key=]value)对组成。} MongoDB文档类似于JSON对象。 {100％：字段值可以包含其他文档，数组及文档数组。} {82％：Mongodb也支持Map/reduce操作，主要是用来对数据进行批量处理和聚合操作。} {97％：Map函数调用emit(key，value)遍历集合中所有的记录，将key与value传给Reduce函数进行处理。} Map函数和Reduce函数是使用Javascript编写的，并可以通过db.runCommand或mapreduce命令来执行MapReduce操作。

3.2.1副本集

{88％：MongoDB副本集是将数据同步在多个服务器的过程。} {100％：复制提供了数据的冗余备份，并在多个服务器上存储数据副本，提高了数据的可用性，并可以保证数据的安全性。} {100％：复制还允许您从硬件故障和服务中断中恢复数据。}

mongodb的副本集至少需要两个节点。 {100％：其中一个是主节点，负责处理客户端请求，其余的都是从节点，负责复制主节点上的数据。} mongodb各个节点常见的搭配方式为： 一主一从、一主多从。 {100％：主节点记录在其上的所有操作oplog，从节点定期轮询主节点获取这些操作，然后对自己的数据副本执行这些操作，从而保证从节点的数据与主节点一致。} MongoDB复制结构图如下图3-2所示：

其中每一个副本集是一个由Primary主数据库以及1个或多个Secondary从数据库组成的一个集合，集合中每个数据库保存相同的数据， {73％：并由主数据库负责实际的业务读写，客户端主节点读取数据，在客户端写入数据到主节点时，主节点与从节点进行数据交互保障数据的一致性，} 从数据库作为容灾备份。 {92％：Replica Sets的结构类似一个集群，完全可以把它当成一个集群，因为它确实与集群实现的作用是一样的：} {97％：如果

其中一个节点出现故障，其他节点马上会将业务接管过来而无须停机操作。}

一个由一台主节点和两个副节点的副本集Compose编排如下，通过mongod --replSet rs1 --shardsvr --port 27017命令将三个节点设为同一副本集：

在所有节点启动后，连接主数据库节点配置副本集，配置命令如下：

至此，完成了一个 MongoDB副本集集群的配置，用户可以通过任意一个节点，对数据库进行访问， 不管任意节点宕机时，副本集将会路由到剩余的可用节点访问数据。

3.2.2通过分片实现均衡负载

{100％：在Mongodb里面存在另一种集群，就是分片技术，可以满足MongoDB数据量大量增长的需求。} {98％：当 MongoDB存储海量的数据时，一台机器可能不足以存储数据，也可能不足以提供可接受的读写吞吐量，} {80％：并且由于复制所有的写入操作到主节点，延迟的敏感数据会在主节点查询，单个副本集限制了12个节点，} 当请求量巨大时会出现内存不足，本地磁盘不足，垂直扩展价格高昂等原因，单一的副本集已无法支持业务的增长。 {100％：这时，我们就可以通过在多台机器上分割数据，使得数据库系统能存储和处理更多的数据。}

{81％：分片是某一集合中负责某一子集的一台或多台服务器。} {41％：MongoDB会自动地将数据均匀分布在分片上，同时最小化需要被移动的数据量。} MongoDB在集群入口会启动Mongos服务来隐藏分片之间复杂性向用户提供一个服务接口，并作为一个路由节点接受用户的所有读写。 {44％：同时MongoDB还会存在一个配置服务器保存集群的配置信息。} 整个MongoDB集群组成部件结构如下图：

{45％：其中多个路由节点可以起到均衡负载与容灾的作用，} 防止出现路由节点宕机的情况，配置服务器负责将单个 MongoDB中的数据分片到多个副本集的实例上， 配置服务器的个数与一个副本集中的节点个数一致，启动配置服务器的命令与启动普通副本集的 MongoDB实例类似， 启动后通过以下命令配置：

最后在路由实例中，通过mongos将配置服务器端口加入路由列表启动，命令如下：

最后连接路由实例将副本集端口加入即可，命令如下：

{42％：由此，便成功创建了由6个Mongo副本集存储实例，3个配置实例，1个路由实例组成的MongoDB数据库集群。}

3.2.3创建数据集合

{87％：MongoDB 是一种面向文档(document-oriented)的数据库，其内存储的是一种 JSON-like 结构化数据。} 尽管拥有和关系型数据库 Database/Table 类似的的 DB/Collection 概念，但同一 Collection 内的 Document 可以拥有不同的属性。 {100％：集合存在于数据库中，集合没有固定的结构，这意味着你在对集合可以插入不同格式和类型的数据，但通常情况下我们插入集合的数据都会有一定的关联性。} MongoDB的元数据支持 String，Integer，Boolean，Double，Min/Max keys，Arrays，Timestamp，Date，Object ID，Binary Data，Code，Regular expression等数据类型。 同时在MongoDB里对于模型之间关系的表示，即可以用Link，又可以用Embedded，Link主要用于表示多对多的关系，Embedded主要表示包含的关系，基于此，我们通过JSON文档的方式，以fildName: {options}的格式，为所有数据模型创建MongoDB集合。

对于水厂监控的服务器而言，由于需要查询实时历史数据与最近的历史统计数据，因此查询数据需要根据数据插入时间进行排序， MongoDB提供了自动创建时间戳的方式，为每个文档建立创建与更新的时间戳，以用于排序， 方法只要在创建 Schema时 option使用{ versionKey: false， timestamps: { createdAt: ’ createTime ’， updatedAt: ’ updateTime ’ }}

同时对createTime字段创建索引，MongoDB索引方式与传统RDBMS一样采用B+树的索引方式，因此使用方法与RDBMS类似。 MongoDB支持单字段索引，复合字段索引以及数组索引的方式，我们通过ensureIndex({createTime: -1})命令，为CreateTime字段创建倒序的单字段索引。

3.3Redis数据库

redis是一个key-value存储系统。 和Memcached类似，它支持存储的value类型相对更多，包括string(字符串)、list(链表)、set(集合)、zset(sortedset--有序集合)和hash（哈希类型）。 {100％：这些数据类型都支持push/pop、add/remove及取交集并集和差集及更丰富的操作，而且这些操作都是原子性的。} 在此基础上，redis支持各种不同方式的排序。 {100％：与memcached一样，为了保证效率，数据都是缓存在内存中。} {100％：区别的是redis会周期性的把更新的数据写入磁盘或者把修改操作写入追加的记录文件，并且在此基础上实现了master-slave(主从)同步。}

Redis是一个高性能的key-value数据库。 {99％：redis的出现，很大程度补偿了memcached这类key/value存储的不足，在部分场合可以对关系数据库起到很好的补充作用。} 它提供了Python，Ruby，Erlang，PHP客户端，使用很方便。 [1]

Redis支持主从同步。 {100％：数据可以从主服务器向任意数量的从服务器上同步，从服务器可以是关联其他从服务器的主服务器。} 这使得Redis可执行单层树复制。 {100％：从盘可以有意无意的对数据进行写操作。} {100％：由于完全实现了发布/订阅机制，使得从数据库在任何地方同步树时，可订阅一个频道并接收主服务器完整的消息发布记录。} {100％：同步对读取操作的可扩展性和数据冗余很有帮助。}

3.3.1 缓存Session

{100％：Session 是面向连接的状态信息，是对 Http 无状态协议的补充。} {100％：Session数据保留在服务端，而为了标识具体 Session信息指向哪个连接，需要客户端传递向服务端发送一个连接标识，} {100％：比如存在 Cookies中的 session_ id值（也可以通过 URL的 QueryString传递），服务端根据这个 id存取状态信息。} {100％：随着网站规模（访问量/复杂度/数据量）的扩容，针对单机的方案将成为性能的瓶颈，分布式应用在所难免。} 所以，有必要研究一下 Session 的分布式存储。

{100％：如前述， Session使用的标识其实是客户端传递的 session_id，在分布式方案中，一般会针对这个值进行哈希，以确定其在 hashing ring 的存储位置。} {100％：hashing ring就是一个分布式结点的回路（取值范围：} 0到232-1，在在零点重合）： {100％：Session 应用场景中，它根据 session_id 的哈希值，按顺时针方向就近安排一个小于其值的结点进行存储。} {100％：在 Session 处理的事务中，最重要的环节莫过于 客户端与服务端 关于 session 标识的传递过程：}

{92％：1、服务端查询客户端Cookies 中是否存在 session_id}

2、有session_id，是否过期？ 过期了需要重新生成； 没有过期则延长过期

3、没有 session_id，生成一个，并写入客户端的 Set-Cookie 的 Header，这样下一次客户端发起请求时，就会在 Request Header 的 Cookies带着这个session_id。

在Node.js中，通过Express Session模块，以及Redis的connect-redis模块，可以方便的搭建基于Redis的分布式缓存，node.js实现代码如下：

3.3.2 缓存查询

对于水厂处理系统服务端而言，绝大多数的查询都是对实时数据与最近历史数据的查询，因此我们可以通过Redis构建查询缓存，降低数据库访问压力，加快请求处理相应。

Redis支持 String， List， Set， Hash类型的数据存储格式，对于水质处理流程模型的查询结果，我们可以选用List格式进行存储， 由于大部分查询都是根据 createTime倒序排序查询一定数量的数据，因此我们可以用collectionName的键存储缓存结果， 当有新的数据插入时，通过 lpush(key， String)的方式往该 collectionName的list插入更新数据的 Json字符串， 然后使用 rpop(key)命令将所有尾部元素删除。 在查询时，如果查询的排序方式是倒序排序 createTime，首先判断查询请求的 limit是否超过 list缓存的长度， 可以通过 llen(key)命令查询，如果小于则直接通过 lrange(key， skip， limit)命令直接从 Redis中取出， 否则话，先判断当前节点的可用内存大小，然后判断新的 limit是否足够，足够的话， 从 MongoDB中查询数据，然后更新到 List中。

由于用户请问访问时，都需要先获得 Token的授权，然后每次请求需要经过 Token的验证， 因此直接使用MongoDB存储 User与 Token，需要每次请求前额外访问一次数据库查询，而 User与 Token表通常数据量不大， 因此可以将整个表预先从 MongoDB存入 Redis作缓存查询。 Redis不支持直接存储Object对象，但是可以通过Hash数据结构存储Object。 通过命令hmset(key， Object)命令，将一个Object对象存入Redis，其中key是数据表名加上数据主键，然后可以通过hgetall(key)命令查询需要的数据。 对于无法直接根据主键搜索的数据，需要额外的用set(key， String)命令建立需要搜索的key与主键的关联，然后再进去查询。 当需要删除该Object时，使用hdel(key，fields)命令来删除所有Hash值。

四章 处理服务器实现

4.1语言与框架

传统的企业服务器通常采用 JAVA语言，基于 Spring， Spring MVC， Mybaits三大框架实现对 HTTP请求的处理与响应， 其中 JAVA是一种面向对象的强类型语言，然而由于 JAVA处理异步任务只有通过新的线程或者线程池实现， 缺乏 await语义，因此总有线程因为 IO或者其他原因处在 sleep状态，其结果就是没办法（很难）实现单服务器高并发的承载能力， 并且 SSM框架仍然是基于传统的 SOA思想设计，对于分布式系统的实现十分复杂，对 Restful API的支持仍有许多局限。

{92％： 为了能适应高并发的情况（成千上万的连接），服务器需要采用异步非阻塞模式。 } {100％： 可能已经在IO操作中实现了这种方式。 } {92％： 但问题是，如果服务器代码的任何部分可能产生阻塞，你都需要开启一个线程。 } {100％： 在这种级别的并发下，你不能去为每个连接创建线程。 } {100％： 所以整个代码路径都需要异步非阻塞式的， 不仅仅在输入输出层。 } 这就是 Node擅长的地方，以下图4-1是 JAVA这类多线程服务器与 Node. js的对比，因此对于一个并发数巨大但是业务逻辑基本较为简单的企业应用， Node. js是一个快速实现处

理服务器的最佳选择之一。

{51%： 图4-1 多线程服务器与Node.js服务器对比图}

{51%： Node.js是基于弱类型的解释性脚本语言JavaScript实现的可以快速构建网络服务及应用的平台，} {80%： 在Chrome JavaScript V8 Runtime建立的平台，用于方便地搭建响应速度快、易于扩展的网络应用。} {100%： Node.js使用事件驱动，非阻塞I/O模型而得以轻量和高效，非常适合在分布式设备上运行的数据密集型的实时应用。} {100%： V8引擎执行Javascript的速度非常快，性能非常好。} {100%： Node对一些特殊用例进行了优化，提供了替代的API，使得V8在非浏览器环境下运行得更好。}

{100%： Node.js使用Module模块去划分不同的功能，以简化应用的开发。} {100%： Modules模块有点象C语言中的类库。} {100%： 每一个Node.js的类库都包含了十分丰富的各类函数，比如http模块就包含了和http功能相关的很多函数，} {100%： 可以帮助开发者很容易地对比如http，tcp/udp等进行操作，还可以很容易的创建http和tcp/udp的服务器。}

NPM是JavaScript的包管理器，也是世界上最大的软件仓库，用户通过NPM来安装，分享与发布代码模块，Node.js通过编写package.json文件构建模块依赖，然后通过npm install命令一键执行安装文件中的所有模块。 对于水质处理工厂服务端，我们创建package.json文件并加入如表4-1所示的依赖模块：

作用模块名备注

基于以上模块，我们运行npm install便可构建处理服务器的基本框架，通过require命令在JavaScript代码中编写处理服务器的具体代码。 在以下章节将介绍处理服务器的具体实现，主要包括授权与验证，Restful Api接口。

4.2 授权与验证

对于任何的服务端而言，安全往往是容易受到忽视的一个重要部分，许多早期的公司由于为了方便接口完全暴露在公网，因此成为了黑客与爬虫的目标，被窃取大量机密的公司业务数据，造成了大量的损失。 因此业界一直也在研究有关服务器授权与验证的问题。 如何验证请求的合法性，保障接口与数据的安全，是开发处理服务器的首要目标。 对于服务器而言，授权与验证过程既要保证安全性，同时也要满足高效性，避免大量服务器计算性能消耗在授权与验证过程中，造成了请求过高的延迟。

对于Node.js平台，Passport库是实现多种不同方式授权与验证的很好选择。 Passport是一个Node的身份认证中间件，它被设计用于服务端请求认证的目的。 当编写模块时，封装是一种优点，因此Passport将所有其他功能委托给应用程序。 这种原则可以隔离代码依赖，使其保持整洁和具备可维护性，并且使Passport非常容易集成到应用程序中。

{53%： 在现代web应用程序中，身份验证可以采用多种形式。} {59%： 传统中用户通过提供用户名和密码登录服务器。} 随着社交网络的兴起，使用OAuth提供商(例如Facebook或Twitter)单点登录已经成为一个受欢迎的身份验证方法，而公开API的服务通常也需要基于Token的凭证来保护访问。 {72%： 其中OAuth是一种认证与授权的协议，OAuth在”客户端”与”服务提供商”之间，设置了一个授权层（authorization layer）。} {90%： ”客户端”不能直接登录”服务提供商”，只能登录授权层，以此将用户与客户端区分开来。} {94%： ”客户端”登录授权层所用的令牌（token），与用户的密码不同。} {100%： 用户可以在登录的时候，指定授权层令牌的权限范围和有效期。} {88%： ”客户端”登录授权层以后，”服务提供商”根据令牌的权限范围和有效期

，向”客户端”开放用户储存的资料。｝ OAuth 2.0定义了四种授权方式。

1.授权码模式（authorization code）

2.简化模式（implicit）

3.密码模式（resource owner password credentials）

4.客户端模式（client credentials）

｛49％：Passport认为每个应用程序都有独特的身份验证需求。｝｛40％：通过身份验证机制，也被称为策略，Passport将其打包为单个模块。｝ 应用程序可以选择使用哪种策略，而不需要创建不必要的依赖关系。 尽管在身份验证中涉及到复杂的问题，但代码并不一定要复杂。 因此通过模块的封装，基于Passport可以很方便的实现登录验证过程以及需要的OAuth2.0授权方式。

4.2.1用户登录认证

用户账号密码登录是每个服务器需要实现的基本功能之一，对于一个服务器而言，用户信息是其最基本最重要的数据信息，｛43％：服务器需要通过用户登录过程，验证用户是否是合法的已注册用户，具备何种的权限。｝并且所有以用户用户名作为外键的表都需要先经过登录过程，才能完成业务的正常进行。 在登录成功后，因为HTTP是无状态的协议，即这个协议是无法记录用户访问状态的，其每次请求都是独立的无关联的， 而服务器需要知道用户的状态信息，尤其是登录状态信息，因此服务器需要将该用户信息记录到请求的 Cookie中标记登录状态，｛49％：再为该用户的登录会话创建 Session记录服务器的用户信息。｝

基于Passport的用户登录模式是加入passport-local的模块，而后在代码中编写本地的用户密码授权策略，便可以接受local模式的认证请求，具体代码如下：

｛40％：其中，使用 Redis数据库查询 HTTP请求中的用户名，检查 HTTP请求中密码经过 MD5加盐后的散列值是否与数据库中存储的散列值一样，｝ 如果是的话则认证成功。

而后在Node.js的Express框架中，编写登录与注销的Post请求路由路径，如下所示：

其中登录通过调用passport的authenticate命令，查询local策略，调用上述的local登录策略代码，成功或失败后复位向到相应的链接完成登录。 ｛48％：注销通过在request中清除用户的Cookie信息，完成注销过程。｝

4.2.2授权码模式授权

在授权码模式授权的作用是为处理服务器提供一个公共的授权接口，提供给不可信任的第三方客户端使用，从而避免第三方应用获取用户的账户密码信息。 ｛41％：在授权码模式流程中，客户端不会直接向资源服务器请求Token授权，而是把资源服务器导向 Auth服务器要求许可，｝｛49％：Auth服务器再通过 Redirect URI转址来告诉客户端授权许可码（code）。｝｛42％：在复位向回去之前，Auth服务器会先认证资源服务器并取得权限。｝｛43％：因为资源服务器只跟Auth服务器认证所以客户端便绝对无法拿到用户的帐号密码。｝

在这种流程中授权码会以一个散列的字符串存储在服务器数据库，然后转发给客户端。 作为授权的许可条件

。 在获取授权码之后。 此时是仍然没有获取Access Token的，客户端需要自己将授权码发送给Auth服务器请求Access Token的授权。 整个的流程具体如下图4-2所示：

（A）客户端把用户的 User-Agent转发到Auth服务器启动流程。 客户端会传输Client ID， 申请的 scopes，内部state，Redirection URI作为转址地址接受Auth服务器授权结果。

{41％：（B）Auth服务器通过 User-Agent验证用户是否合法，并确认资源拥有者许可或者驳回客户端的授权请求;}

（C）假设资源拥有者许可了授权请求，Auth服务器会把User-Agent 复位向回先前指定的Redirection URI其中包含了： Authorization Code，许可的 scopes，先前提供的内部 state;

（D）Client向Auth服务器发送Token请求，传送时附带： 先前取得的 Authorization Code以及Client的认证资料

（E）Auth服务器认证Client与Authorization Code，符合的话回传随机生成的散列Access Token与Refresh Token。

4.2.3密码模式授权

密码模式授权通常用于可信任的客户端授权，由于授权码模式授权需要经过一个授权码中转流程较为繁琐，而对于服务器可以信任的客户端请求， 如同一公司开发的或者经过认证的客户端，则可以通过密码模式授权简化流程。 在密码模式授权流程中，用户自身的帐号密码以及可信客户端的 Client信息将直接当作授权许可， 传输给Auth服务器获取 Access Token，这种模式要求客户端开发过程中禁止存储用户的帐号密码，只在授权时使用一次，用来获取 Access Token，随后存储长效的 Access Token或 Refresh Token用以认证，最后拿到的除了 Access Token之外， 还会拿到 Refresh Token，整个授权的流程图如下图4-3所示：

图4-3 密码模式授权流程图

{72％：（A）用户向客户端提供真正的帐号密码；}

{40％：（B）客户端使用客户的帐号密码与自身的Client信息，向Auth服务器申请Access Token认证;}

{45％：（C）Auth服务器认证Client信息与用户的帐号密码后，如果正确则核发Access Token给客户端}

4.2.4 Refresh Token换发授权

换发(Refreshing) Access Token，指的是处于安全考虑， Access Token存在于一个 expire时限， 超过时将会失去认证效应，导致目前的 Access Token过期或者权限不足，而需要取得新的 Token。 允许换发的前提是 Auth服务器之前有向客户端许可过Refresh Token。 如果没有的话则不行。 Client通过捕获Response的502状态码Unauthorized异常，自动完成换发Access Token。

{43％：Refresh Token通常具备较长的有效期，被用作获取新的Access Token，因此客户端需要存储Refresh Token在本地。} 在换发新的Access Token的时候，可以一起授权新的 Refresh Token，而后客户端将新的Refresh

Token替换旧的Refresh Token，这样子的话客户端必须把旧的Refresh Token。 同时，Auth服务器也许要删除旧的Refresh Token。 新的Refresh Token其Scope也要与旧的保持一致。

4.2.4 Bearer Token认证

OAuth 2.0 (RFC 6749)定义了客户端如何获取Access Token的方法，通过Access Token获取Protected Resource。OAuth 2.0定义Access Token是资源服务器用来认证的唯一方式，基于Token的验证方式资源服务器就不需要再提供其他认证方式，例如用户的账号密码。

然而在 RFC 6749里面只定义了抽象的概念，细节如Access Token的格式，如何传输给资源服务器以及无效的处理方法都没有进行定义，所以在RFC 6750标准中定义了Bearer Token的概念与用法。 所以在 RFC 6750 另外定义了 Bearer Token 的用法。 Bearer Token是一种 Access Token的类型，由 Auth服务器在资源拥有者允许下核发给客户端，{45%：资源服务器只要认证 Token合法就可以认定客户端已经由资源拥有者许可，} 不需要再通过密码来验证 Token的真伪。

Bearer Token的格式为Bearer XXXXXXXX，其中 XXXXXXXX 的格式为 b64token，b64token的定义：

b64token = 1*(ALPHA / DIGIT / "-" / "." / "_" / "~" / "+" / "/") *"="

写成 Regular Expression 即是：

/[A-Za-z0-9\-\._~\+\/]+=*/

{52%：客户端向资源服务器校验Access Token的方式有三种：}

(1) 放在HTTP Header里面，Header键规定为Authorization，值规定为Bearer加上Token实际值，Auth服务器必须支持这种方式，也是最为安全的方式。

(2) 放在Request Body里面（Form之类的），以键值对发送，前提是Header 要有 Content-Type: application/ x-www- form- urlencoded， Body格式要符合 W3 C HTML4.01定义 application/ x- www- form- urlencoded， Body要只有一个 part（不可以是 multipart）， Body要编码成只有 ASCII chars的内容， Request method必须是一种有使用 request-body的， 也就是说不能用 GET。 Auth服务器可以但不一定要支持这个方式。

(3) 放在 URI 里面的一个 Query Parameter，这种方式由于完全将Token暴露出来，因此是不建议的方式。

Auth服务器向客户端返回认证失败的情况，例如没给 Access Token或是给了但不合法（如空号、过期、资源拥有者没许可客户端拿取此资料）， 则 Auth服务器必须在回应里包含 WWW- Authenticate的 header来提示错误。这个 header 定义在 RFC 2617 Section 3.2.1。 WWW-Authenticate 的值，使用的auth-scheme 是 Bearer，随后一个空格，接着要有至少一个auth-param。

如果客户端出示了 Access Token 但认证失败，则最好加上 error 这个 auth-param，用来告诉客户为何认证失败。此外还可以加上 error_description 用自然语言来告诉开发者为什么错误，但这个不该给使用者看到。 此外也可以加上 error_uri 用来提供一个网址，里面用自然语言解释错误讯息。 这三个auth-param 都只能最多出现一次。 如果客户端没有出示 Access Token （例如客户端不知道需要认证，或是使用了不支持的认证方式（例如不支持 URI

parameter）），则 response 不应该带 error 或任何错误讯息。

4.3 Restful API接口实现

{43％： Restful(REST)或RESTful Web服务是在Internet上提供计算机系统互操作性的一种方式。} {42％：符合rest风格的Web服务允许使用一套统一标准表示对系统访问和操作的请求，并且预定义Web资源的操作是无状态的。} 其他形式的Web服务标准，会暴露本身的一系列操作，如WSDL和SOAP[1]。 2000年，Roy Fielding在他的博士论文中提出了“表征状态”的概念[2][3]。 使用REST来设计HTTP 1.1和统一资源标识符(URI)[4][5][6]。 Rest这个概念是为了传达如何设计一个良好的Web应用程序行为： 它是一个网络中的网络资源（或者虚拟状态机），用户在应用程序中通过选择要访问链接， 如/ user/ tom，来进行 GET或 DELETE等操作(状态转换)，并将下一个资源(代表应用程序的下一个状态)传输给用户使用。

互联网起初将“网络资源”定义为经过URLS区分的文档或文件，但是现今无论任何事物或实体，在网上它们都有一个更通用和更抽象的定义，可以是唯一识别，命名或地址等任何方式。 在基于Rest的Web服务中，对资源URI的请求将获得可能是XML、HTML、JSON或其他已定义格式的响应。 响应可能会确认已对存储资源进行了一些更改，并可能提供相关资源或资源集合的超文本链接。 最常见的是使用HTTP协议，可用的操作包括使用HTTP中GET、POST、PUT、DELETE等预定义的操作方法。

{65％： REST描述了一个架构样式的互联系统（如 Web应用程序），通过使用无状态的协议和标准行为，} 一个 REST系统将高性能与高可靠性，同时获得动态弹性增长的能力。 通过重用组件，可以不影响系统运行下进行资源的管理和更新，尽量它们作为一个整体正在运行。 {100％： REST 约束条件作为一个整体应用时，将生成一个简单、可扩展、有效、安全、可靠的架构。} {100％：由于它简便、轻量级以及通过 HTTP 直接传输数据的特性，RESTful Web 服务成为基于 SOAP 服务的一个最有前途的替代方案。} {100％：用于 web 服务和动态 Web 应用程序的多层架构可以实现可重用性、简单性、可扩展性和组件可响应性的清晰分离。} {100％：开发人员可以轻松使用 Ajax 和 RESTful Web 服务一起创建丰富的界面。}

4.3.1基于Express框架开发

{100％： Express 是一个基于 Node.js 平台的极简、灵活的 web 应用开发框架，它提供一系列强大的特性，帮助你创建各种 Web 和移动设备应用。} {100％：丰富的 HTTP 快捷方法和任意排列组合的 Connect 中间件，让你创建健壮、友好的 API 变得既快速又简单。} {96％： Express 不对 Node.js 已有的特性进行二次抽象，本文只是在它之上扩展了 Web 应用所需的基本功能。}

Express 的优点是线性逻辑： 路由和中间件完美融合，通过中间件形式把业务逻辑细分，简化，一个请求进来经过一系列中间件处理后再响应给用户，再复杂的业务也是线性了，清晰明了。 中间件模式就是把嵌套的异步逻辑拉平了，但它也只能是从较宏观的层面解耦顺序执行的异步业务， 它无法实现精细的异步组合控制，比如并发的异步逻辑，比如有相对复杂条件控制的异步逻辑。 开发通常会要借助 async、bluebird 等异步库。 但即便有了这类异步库，当涉及到共享状态数据时，仍然不得不写出嵌套异步逻辑。

本文先通过代码 express()声明一个新的 Express实例的 server，通过以下代码首先为 server添加必要的中间件，包括 body与 cookie的解析， session的解析存储， passport中间件的使用，关键代码如下：

{45％：其中HTTPS监听是建立在 TLS/SSL 之上的 HTTP 协议，即HTTP下加入SSL层，HTTPS的安全基础是SSL，因此加密的详细内容就需要SSL。} {91％： 它是一个URI scheme（抽象标识符体系），句法类同http:} 体

系。 用于安全的HTTP数据传输。 {81%：SSL（Secure Sockets Layer，安全套接层），及其继任者TLS（Transport Layer Security，传输层安全）是为网络通信提供安全及数据完整性的一种安全协议。} {40%：TLS与SSL在传输层对网络连接进行加密，在创建HTTPS监听之前，需要先生成服务器自己的HTTPS证书，步骤如下：}

1、为CA生成私钥： openssl genrsa -out ca-key.pem -des 1024

2、通过CA私钥生成CSR： openssl req -new -key ca-key.pem -out ca-csr.pem

3、通过CSR文件和私钥生成CA证书： openssl x509 -req -in ca-csr.pem -signkey ca-key.pem -out ca-cert.pem

4、为服务器生成私钥： openssl genrsa -out server-key.pem 1024

{48%：5、利用服务器私钥文件服务器生成CSR：} openssl req -new -key server-key.pem -config openssl.cnf -out server-csr.pem

{50%：6、通过服务器私钥文件和CSR文件生成服务器证书：} openssl x509 -req -CA ca-cert.pem -CAkey ca-key.pem -CAcreateserial -in server-csr.pem -out server-cert.pem -extensions v3_req -extfile openssl.cnf

最后在node-js中为server监听80与443端口分别作为http与http的请求端口，代码如下：

4.3.2使用node-restful开发标准Restful接口

node-restful是Express框架下的一个网络中间件，通过配合Mongoose Schema模型层资源，自动地构建标准Restful API接口与路由，其中node-restful支持的路由类型有：

node-restful提供一些方法在用户注册Mongoose Schema资源后，供用户生成接口，其中通过命令methods([...]) 可以声明一个允许资源操作的方法列表。 当用户不希望该资源对外暴露某些接口时，只需要将该方法从参数中删除即可，例如通过Resource.methods([’get’, ’post’, ’put’])可以禁用Delete操作。

同时 node- restful当然也允许注册其他的路由请求，例如通过代码 Resource. route(’ recommend ’, function(req, res, next)便可以注册一个设置在/ resources/ recommend路径的路由操作， 这个方法将会被应用到所有的方法列表中，当然也可以通过例如 Resource. route(’ recommend. get ’, function(req, res, next)的代码覆写某个已注册的方法。

通过 node- restful，我们也可以通过 node- restful提供的 before(method, function)为路由方法设置 Filter拦截器，例如对于用户登录的路由方法而言，我们可以通过以下代码，使用 bcrypt模块将用户密码明文加盐后加密， 再存储进 Mongo数据库中：

由于所有业务数据的请求必须经过OAuth2.0协议的验证，因此再所有的接口前，我们需要加上对Bearer Token的验证的before方法，关键代码如下：

对于水质处理流程监控数据而言，我们需要先从 Redis缓存中查询需要的缓存数据，对于命中的查询结果直接返回 Response， 因此需要在所有模型层的 get请求前添加 getCache方法，关键代码如下：

随后 node- restful将会根据 Mongoose Schema自动地向 Mongo数据库查询、插入、更新或者删除数据，并返回结果到 Response中， 对于服务端而言，我们在数据库操作完成后，对于 get请求，将过于大量的数据压缩与筛选，对于 post请求， 需要将新插入的数据更新到 Redis缓存中， node- restful提供了 after(method， function)方法，将返回的查询数据存储在 res. locals中， 其中 res. locals. statusCode是请求的返回状态吗， res. locals. bundle则是查询的结果，基于以下代码， 我们为 post请求处理后更新 redis缓存结果，关键代码如下：

node-restful同时也提供丰富与标准的符合Restful的查询过滤条件，基于此，我们可以开发出可用性极高的Restful标准API，其中基于node-restful实现的过滤条件包括表4-1：

过滤条件查询请求示例描述

最后我们先通过代码const router = new express.Router()声明一个新的路由，再通过代码Resource.register(router，’/resources’)将Resource模型注册到/resources路径上。 在 node的主函数 server. js中，为 server添加先前声明的 API与 AUTH的路由，代码分别为 server. use(’/’， authRouter)与 server. use(’/ api’， apiRouter)，通过路由，我们可以将 authRouter里注册的路径仍然路由在根路径下， 而 apiRouter里注册的路径，重新路由到/ api后，例如 GET/ api/ Resource/: id，至此我们便完成了处理服务器的开发。

第五章 服务器配置与测试及性能分析

5.1服务器配置与测试

{46％： 5.1.1单台主机配置多进程处理服务器实例}

由于处理服务器是基于 node. js平台实现的，而 node. js使用的 Chrome V8引擎是一个单线程的进程， 因此对于 node. js应用而言单个进程无法很好的利用服务器多核心的计算优势，从而造成一定的计算资源浪费。

正是基于这样的背景需求，PM2工具应运而生。 PM2是一个nodejs应用的进程管理工具，通过对CPU核心的绑定实现单台物理或虚拟主机上的资源均衡负载，以多进程的方式最大效率利用多核CPU的特性。 同时，它通过自动重启的方法保持node.js进程即使发生错误仍然可以重启来保证服务器的可用性。

配置通过 PM2启动 nodej. s应用的方法是编写 PM2的 process. yml配置文件，来配置 node. js应用启动的参数、环境行为、变量以及 log路径等， 对于本文的处理服务器，我们编写以下 process. yml文件来启动：

其中exec_mode: ’cluster’指的是开启 PM2的 cluster启动模式，它允许根据可用的 CPU数量来弹性地启动相应数量的 node. js应用进程实例， 而不需要做其他的一些额外代码修改，然后自动地将路由到该台物理主机的请求均衡负载到相应的进程实例中。 而watch： [’server/’]将会监视server文件夹下的所有代码文件，当有文件修改时，无需额外的操作与等待便可立即热部署最新的代码。 {40％：因此对于多核CPU的服务器而言，它可以极大的提升应用的性能与可用性。}

5.1.2配置处理服务器集群

对于多台物理或虚拟主机构成的集群，我们需要通过 Docker及其相关的 Docker组件编写一系列的脚本配置文件， {41％：以支持在集群中的每台物理主机上自动地启动处理服务器实例，从而最大的利用集群的计算资源。

} 并且在集群的性能出现瓶颈，无法满足用户的服务需求快速增长时，可以弹性的将新的物理主机加入集群中并自动启动新的处理服务器实例。

首先，我们在Dokcer Compose的配置文件中，编写处理服务器的编排：

其中，通过command: pm2- docker-- watch./ config/ process. yml命令用 PM2启动多进程处理服务器实例，利用当前目录的 Docker File来 Build Docker镜像， 并绑定到 Docker Hub的 gjscut/ water_ watcher_ api_ server远端镜像上。 然后添加Redis与MongoDB数据库容器的链接，以加入自身容器的DNS列表来访问。 {46％：通过Volumes将本地文件的代码挂载在容器上，实现容器中代码的热更新。} 最后将主机的80端口与443端口与容器的相应端口映射，以实现Http与Https的访问。

对于多台物理主机而言，首先需要通过Docker Swarm为所有物理主机搭建集群环境建立通信以进行管理，搭建集群的方法是：

1、选取任意一台（建议采用性能最佳）的主机作为集群的 Manager，首选获取该台主机的 IP地址， 然后在任意一台具备 Docker Swarm的主机上使用 docker swarm init-- advertise- addr[MANAGER- IP]命令建立集群;

2、输入docker swarm join-token worker获取其他主机加入该Manager集群网络的命令;

3、 输入docker swarm join --token SWMTKN-1-[TOKEN] [MANAGER-IP]与Manager建立TLS安全通信，则完成了将该台物理主机加入集群中。

其中建立TLS安全通信的原因是为了避免中间人伪造Docker通信的数据包从而启动一些外部的不安全容器造成中间人攻击。

在将所有的物理或虚拟主机都加入集群中后，先使用 docker node list命令，查询当前集群中所有节点的数量，然后使用 docker stack deploy – compose- file docker- compose. yml命令，启动 Docker Compose文件所编排的所有容器， 自动地均衡启动在集群中的节点中，再使用 docker stack services命令查询处理服务器的 Service名字，最后使用 docker service scale[SERVICE- NAME]=[NODE- AMOUNT]命令， 将处理服务器自动地扩增到集群中的每台节点中启动，并在启动时通过 PM2启动多进程的 node. js实例绑定到每台主机的 CPU上， {41％：至此完成了处理服务器的部署，并最大限度的利用了集群的计算性能。}

当集群的性能出现瓶颈，已经无法支撑用户的访问，需要增加新的主机时，在新的主机中运行以下脚本， 便可以在集群运行中的任意时刻将一台主机加入集群中并自动地部署处理服务器：

5.1.3服务发现与均衡负载器配置

对于Consul服务发现的配置，通过在Docker Compose配置文件中编写Consul与Registrator两个容器的编排：

其中consul容器使用官方最新版本的consul-server镜像，通过--bootstrap命令自动启动，并暴露8500一个HTTP端口来与外部的容器通信。 registrator容器使用同一开发者的 registrator容器镜像，并添加对consul容器的链接，然后通过命令-internal -resync 600 consul: //consul: 8500使得自己每隔600毫秒向consul server同步监听到的当前集群内所有Docker容器以及它们的IP地址与端口，VOLUMES挂载的目的是为了在registrator容器中获取Docker的所有容器列表。

最后编排好的容器会在docker stack deploy –compose-file docker-compose.yml命令中自动地部署在某台主机上。

5.2服务器测试

{48%：在软件开发中，对于一个服务器而言，良好的测试是一个十分重要的组成部分。} {44%：测试可以帮助开发人员筛查代码中出现的错误，是保证软件质量中必不可少的一个环节。} 通过测试，在服务器开发过程中开发人员可以随时的检验代码的质量，发现潜在的问题，对于代码的修改以及新特性的编写可以保证向前兼容。{43%：对于服务器端而言，通常测试分为代码单元测试与接口测试。}

5.2.1单元测试

其中单元测试通常遵循测试驱动开发（Test Drive Development， TDD）的理念进行，它是一种代码设计准则、测试准则或沟通工具， 开发人员通过预先定义好模型与业务代码的接口，然后依据测试接口再开发代码并在开发过程中自动调用预先完成的测试完成开发过程。 TDD的好处有以下几：

{79%：1、测试代码都是从需求出发的，不是从实现出发的，更关注于对外部的接口;}

{88%：2、软件的需求都被测试代码描叙得很清楚，可以减少很多不必要的文档;}

{74%：3、每次实现都是很小的步骤，这样可以集中注意解决一个问题;}

4、可以优化设计。 为了实现更简便的单元测试，会让开发者逆向的被迫面向接口编程和使用一些设计模式，自然提高代码设计的灵活性，降低耦合度。

基于node.js平台的服务器我们可以采用Mocha和Chai这两个测试框架实现本文服务端的TDD测试，Mocha是一个单元测试框架，和其他的javascript单元测试框架不同的是，他没有assertion库。 但是，Mocha允许使用第三方的断言库，因此我们使用Chai对测试结果进行断言，下面我们给出一个使用Mocha和Chai完成Redis查询缓存读取的测试示范：

{100%：describe方法是用来创建一组测试的，并且可以给这一组测试一个描述。} 一个测试就用一个it方法。 it方法的第一个参数是一个描述。 {100%：第二个参数是一个包含一个或者多个assertion的方法。} 该范例通过测试getSearchResultCache方法返回的数组结果长度是否与要求一致来测试该函数的正确性，至于结果是否正确则需要通过后面的接口测试来完成。

{100%：运行测试只需要在项目的根目录运行命令行：} mocha tests ——recursive ——watch。 {100%：recursive指明会找到根目录下的子目录的测试代码并运行。} {100%：watch则表示Mocha会监视源代码和测试代码的更改，每次更改之后重新测试。} {41%：在该函数未实现前，显然测试会报错误，当正确完成时，该测试示例的运行结果如图5-1所示：}

5.2.2接口测试

对于一个API服务器而言，需要保证业务逻辑的正确性，在客户端请求格式满足预先设计的HTTP接口规范的

情况下，｛44％：确保客户端发送的请求服务端可以正确的解析处理并返回正确无误的需求结果。｝在这种需求下，单纯的单元测试很难完整的验证整个HTTP接口的正确性，特别是无法模拟HTTP的请求，也无法验证响应的状态码与Body是否正确。｛41％：因此需要借助第三方的HTTP接口测试软件对服务器的接口进行测试。｝

Postman是一个具备许多强大测试特性的 API测试工具，它提供一个易用性较好的 GUI来构造 HTTP Request与读取和测试 HTTP Response，同时 Postman支持将接口测试分享给团队中其他人，或者与团队协同编写接口的测试。Postman也可以将写好的接口测试生成API文档，分享给客户端开发人员作为接口调用与返回数据的定义规范。

使用Postman时，首先整个团队需要下载Postman for Linux的桌面版，并注册自己的帐号再加入一个Team中。然后根据分类的需求创建API测试的文件夹，

然后，便可以新建一个窗口来构造HTTP的Request。Postman支持用户构造接口的 URL，请求的 Method，HTTP Header以及 Body，其中 Postman支持以{{}}的方式引用环境变量来构造动态的请求，环境变量可以通过设置全局变量的方式，也可以在每个请求的 Pre- request Script里使用 JS语言设置。对于接口的测试通过对每个接口的 Tests里使用 JS语言编写完成，使用 tests数组来定义断言的结果从而完成测试，当然用户也可以使用该接口测试返回的 Response数据来设置环境变量，以用作后续接口的测试，实现前后接口的上下文通信，例如，通过以下 Request请求 Token;

将返回的Token结果存为环境变量，在后续的API请求中通过在请求中使用--header ’authorization: {{token}}’设置HTTP Header从而解决Token需要动态授权不方便测试的问题。

最后构造编写完所有接口的请求与测试后，设置测试次数，运行Postman中的接口集合测试，测试结果如下图：

则表示服务端通过了所有的已设计接口测试，能够满足客户端正常的业务请求。

5.3性能分析与监控

5.3.1访问负载压力分析

服务器的根本职责是接受并解析客户端的访问请求，对数据处理后返回请求所需的数据或状态，在HTTP通信的每一个阶段都需要一定的计算与内存资源来支持。而客户端的数目会随着企业业务规模的扩展而迅速扩增，因此在同一时间内可能会有大量的客户端请求同时发向服务端，｛45％：造成服务器计算资源或内存资源耗尽，使得服务器的 CPU无法及时响应每个客户端的｝｛42％：请求，甚至会导致服务器主机整个操作系统因为计算资源枯竭而崩溃造成宕机。｝｛42％：正因如此，对于一个服务器而言，通常用户人员会对服务器的访问负载压力进行测试与分析，｝从而预先检验服务器当前的计算性能是否能够满足企业实际生产中的用户访问负载需求。｛50％：对于服务器压力测试主要包括以下指标:｝

1、吞吐率（Requests per second）。概念：｛100％：服务器并发处理能力的量化描述，单位是reqs/s，指的是某个并发用户数下单位时间内处理的请求数。｝｛100％：某个并发用户数下单位时间内能处理的最大请求数，称之为最大吞吐率。｝

计算公式：｛96％：总请求数 / 处理完成这些请求数所花费的时间;｝

2、并发连接数（The number of concurrent connections）。 概念： {96％：某个时刻服务器所接受的请求数目，简单的讲，就是一个会话;}

3、并发用户数（The number of concurrent users，Concurrency Level）。 概念： {98％：要注意区分这个概念和并发连接数之间的区别，一个用户可能同时会产生多个会话，也即连接数;}

4、用户平均请求等待时间（Time per request）。 计算公式： {97％：处理完成所有请求数所花费的时间/（总请求数/并发用户数）;}

5、服务器平均请求等待时间（Time per request: across all concurrent requests）。 计算公式： {61％：处理完成所有请求数所花费的时间/总请求数，可以看到，它是吞吐率的倒数。} {96％：同时，它也=用户平均请求等待时间/并发用户数。}

{56％：对于服务器的访问负载压力测试，通常会选用Apache Bench（ab）工具，ab是Apache超文本传输协议(HTTP)的性能测试工具。} {100％：ab非常实用，它不仅可以对apache服务器进行网站访问压力测试，也可以对或其它类型的服务器进行压力测试。} 比如nginx、tomcat、IIS等，当然也可以对Node.js平台的 服务器进行测试。本文中我们首先在 CPU为 Intel(R) Xeon(R) CPU E5-2650 v2@ 2.60 GHz的单主机服务器环境与 CPU为 Intel(R) Core(TM) i5- 6600 CPU@3.30 GHz4虚拟主机集群环境上部署本文的服务器， 再分别使用 ab工具，对服务器进行测试10并发，100并发，1000并发条件下，持续10秒的负载请求， 测试结果如下表5-1所示：

{61％：通过分析以上测试数据，我们可以得到以下结论：}

1、在相同的服务器环境中，当同一时间的并发访问数越多，服务器对于请求的吞吐率越低， {50％：服务器平均处理时间越长，传输速率越低。}

2、Node.js构建的服务器对于超高并发的请求可以很好的处理，1000并发与10并发的压力测试结果并不会相差太多，服务器仍然可以很好的响应。

{46％：3、 CPU E5-2650单核单虚拟主机环境每秒可以处理大约370个用户请求，} CPU i5-6600单核四主机集群环境每秒可以处理大约2400个用户请求， CPU性能的提升与集群中计算节点数量的提高可以显着增强服务器的计算性能，以满足更高的用户访问负载压力。

5.3.2实时请求监控与分析

对于已经完成部署的服务器而言，仍然可能存在潜在的错误导致客户端对接口的请求无法得到正确的响应，或者在某段时间由于某些特殊原因导致出现大量的请求给服务器端带来负载压力需要开发人员弹性扩增新的计算节点， 也有可能某些接口因为代码编写原因导致需要长时间的处理才能响应客户端或者存在内存泄漏的情况。 因此在服务器部署之后，开发人员仍然需要监控服务器的实时运行状态、虚拟机CPU和内存信息以及请求的访问状态，了解哪些接口耗费大量计算机资源。

对于服务器的监控，可以自己通过 Log日志的方式记录关键的信息然后整理发布到一个相应的接口分析， 但是这种方法由于需要写入大量的 log方法，影响代码的可读性，增加编程的繁琐程度，同时日志的方式也不够直观， 并且对 HTTP接口无法实现很直观的监控，因此本文采用第三方的应用性能监控（Application Performance

Management, {46%: APM) New Relic对服务器端进行实时的监控与性能分析。}

{100%: New Relic是一款基于SaaS的云端应用监测与管理平台，可以监测和管理云端、网络端及移动端的应用，} {100%：能让开发者以终端用户、服务器端或应用代码端的视角来监控自己的应用。} {95%：利用了AOP（面向切面编程）的编程思想，把应用所有的监控逻辑抽象出来，} {100%：让用户专注写自己的业务逻辑，在系统启动的时候，通过相应的技术手段把应用监控的逻辑代码再织入到用户的应用里面，} 从而添加应用监控功能。 通过传输软件实时的web或非web app的性能数据， New Relic可以绘制可读性强的数据图表， {42%：并且分析接口与数据库的CPU占用时间来帮助开发人员监控与分析服务器的性能状态。}

使用New Relic的方法很简单，通过npm install newrelic – save的方式安装node的newrelic探针模块，然后将node_ modules/ newrelic目录下的newrelic. js文件复制到工程的根目录， 再在New Relic官方网站上申请社区免费版帐号以及license_ key，填入newrelic. js文件中，最后在node. js的主方法文件的第一行加入require(’ newrelic ’); 变成成功完成node.js平台的New Relic探针的安装。 重启服务器后New Relic探针将会定时的将数据发送给New Relic的server，用户即可以在官方网站上直接实时监控服务器，首先监控概括图如下图所示。 该图可以反映服务器整体实时响应时间、各个接口的实时响应时间，服务器负载压力、以及请求响应错误率，对于响应错误还能进行错误查询与分析。

对于服务器各个接口的详细监控，反映了各个接口占请求的百分比以及各个时间段接口的调用次数，如下图所示：

New Relic还提供了对单个接口性能分析的功能，如下图所示：

其中， New Relic可以分析每个接口调用过程中经过了哪些中间件，回调与数据库访问，并统计出各个阶段的耗时与百分比， 帮助开发人员定位代码的性能问题所在，从而改善服务器处理效率。

结束语

1.总结

本系统基于自动化容器部署、分布式集群、弹性扩容、 NoSQL存储、查询缓存、 OAuth2授权等技术， {42%：使得实现的水质监控服务器端具备了高计算性能、高可用性、易部署性以及安全性等特点，} 能够以极高的性价比满足大量客户端的并发数据请求，并且具备随业务增长而增强集群计算规模的能力， 并且通过集群技术避免物理主机宕机造成服务器不可使用的情况，通过数据库的主从模式保证数据可以超长时间的存储， 从而满足业务数据的存储需求。

{55%：在做毕业设计的过程中，我遇到过许多困难。} 最开始。 回想起这整个过程，我感触颇多。

（1）这次毕设使我认识到自己的相关专业知识掌握得还不够牢固，知识体系不太全面，主要是以前课堂上学习到的内容没有深入地理解，不够透彻。 {41%：因此，我会在日后的工作、生活中不断充电，活到老，学到老；}

（2）毕设不仅检测学生的书本知识，更是对学生的分析问题解决问题、动手能力最开始的时候觉得题目很难， 无从下手，在自己的努力和老师同学的帮助下，我对题目越来越理解，思路也逐渐清晰， 个人的能力得到了极大的提高；

{44%：（3）在这次毕业设计中，我体会到了同学之间深厚的感情。} 每当我遇到挫折，都会有热心的同学帮助我解决问题，度过难关； {43%：或者是给我提出一些宝贵的建议，激发我对毕设的不断完善和改进，所以在这里非常感谢帮助他们。}

2.展望

关于本系统，我认为有下几个方面可以进行改进：

（1）改进数据库表设计，减少数据的冗余存储以减少硬盘存储容量占用，同时降低HTTP传输历史数据时数据量过大导致传输时间较长的问题。 在请求中也可以增加一个数据请求数的url参数，在服务器端过滤掉多余的数据，以降低数据传输的时间;

（2）增加模型层的丰富度，对用户表写入更多字段以支持其他功能，在服务器端加入值班人员表， 数据监控表等模型，以更灵活的支持客户端的功能需求;

（3）简化数据库集群的部署过程以及简化弹性扩容的部署流程，编写脚本命令使得用户可以直接在容器启动的过程中部署数据库集群和完成集群的扩容;

（4）编写更加丰富的测试与文档，增加测试的代码覆盖率来加强服务端的健壮性，更好的方便客户端开发人员使用客户端接口。

致 谢

{42%：这次毕业设计能够顺利完成离不开老师和同学们的帮助，是他们的支持和鼓励让我不断前进，坚持不懈。} {44%：在毕设即将结束之际，我要对给予我大量关心和督促的王冬生老师表示衷心的谢意。} {46%：王老师工作认真、治学严谨、学富五车，他的学术研究精神深深地感染和激励了我。} {42%：此外，王老师人格高尚，待人和善，在毕设期间不仅教授了我相关学术知识和科研方法，还教会了我许多做人的道理，这些都将使我终生受益。} 当然，在论文的写作过程中，王老师也给了我许多意见和建议，帮助我不断完善论文，提高质量。

同时，我也要向身边的同学表示感谢，当我遇到困难时，他们给予了我很大的帮助和鼓励，使最终我能走到这一步。

{44%：最后要感谢的是大学四年来所有的任课老师，没有他们的辛勤劳动、谆谆教导就没有我今天的成果。}

在这即将离开学校、进入社会之际，我祝愿所有曾经关心、帮助和支持过我的人都能拥有一个绚烂缤纷的美好明天。

参考文献