

南京邮电大学

毕业设计（论文）

题 目 自来水生产运行维护平台服务器设计与实现

专 业 自动化

学生姓名 杨思璇

班级学号 B13050304

指导教师 王冬生

指导单位 自动化学院

日期：2017 年 3 月 21 日至 2017 年 6 月 15 日

毕业设计（论文）原创性声明

本人郑重声明：所提交的毕业设计（论文），是本人在导师指导下，独立进行研究工作所取得的成果。除文中已注明引用的内容外，本毕业设计（论文）不包含任何其他个人或集体已经发表或撰写过的作品成果。对本研究做出过重要贡献的个人和集体，均已在文中以明确方式标明并表示了谢意。

论文作者签名：

日期： 年 月 日

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：服务器架构；均衡负载；分布式集群；容器

ABSTRACT

For the most of enterprise's software development, design and development of the server side is a necessary part of them. With the increasing business, the traditional host server solution was no longer be adopted in the industry due to the performance bottleneck and the high price. Server cluster with load balancing technology has become one of the most suitable and common solution for server design. However, the basic server cluster exist problems include strong module dependence, difficult deployment and long development cycle shortcomings, also the database can't support high concurrent access. These reasons make it still can't be widely used as a general technology to practice in the industrial sector. So based on the previous server cluster researches, our paper solve the server cluster deployment and flexible amplification problem by using Docker which is the representative of lightweight virtualization technology and the corresponding cluster management tool. In addition using the technologies of service discovery and load balance. Based on the NoSQL database and the Node.js platform, our paper provides a high availability, high-performance and highly concurrent solution for distributed cluster. And it implements the standard Restful API interface in an agile way. Finally, through the deployment and monitoring of the server, our paper shows that this solution scheme of server cluster can solve the problem of deployment difficulties better, improve the performance of the whole cluster by code test and server performance analysis.

Key words: Server framework; Load balance; Distributed swarm; Containers technology

目 录

第一章 绪论	1
1.1 课题研究现状	1
1.2 课题研究意义	3
1.3 内容安排	4
第二章 云服务器集群	5
2.1 容器技术	5
2.2 容器管理工具	6
2.3 均衡负载器	7
2.4 服务发现	8
第三章 数据库池设计	10
3.1 数据库选择	10
3.2 MongoDB 数据库	11
3.3 Redis 数据库	15
第四章 处理服务器实现	18
4.1 语言与框架	18
4.2 授权与验证	20
4.3 Restful API 接口实现	26
第五章 服务器配置与测试及性能分析	31
5.1 服务器配置与测试	31
5.2 服务器测试	34
5.3 性能分析与监控	37
结束语	41
1.总结	41
2.展望	41
致 谢	42
参考文献	43

第一章 绪论

现如今，在企业软件开发中服务器架构通常包括网页/服务器架构、客户端/服务器架构或其混合综合架构，无论在何种架构模型中，服务器作为数据的存储与处理中心都是必须的组成部分。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。

自分布式服务器与分布式计算^[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。分布式计算的模式是从实用计算、云计算和软件即服务（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，V Cardellini 等人^[7]对于 web 框架中出现服务器计算性能有限，提出了构建服务器集群的解决方案，通过均衡负载以及动态扩展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，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。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。MSA 是一种分布式服务器设计，通过将松耦合的方式将一个完成的工程剖分为更多单独的服务模块，模块之间相互分离，具有自己独立的业务逻辑，并且可以独立部署。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：容器通过 LXC 技术直接在宿主机操作系统上运行，但如一个完整的主机一样包含独立的计算机资源，包括文件系统与网络端口等，容器之间运行相互隔离，无法互相影响，也无法对宿主机环境产生改动。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]，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：

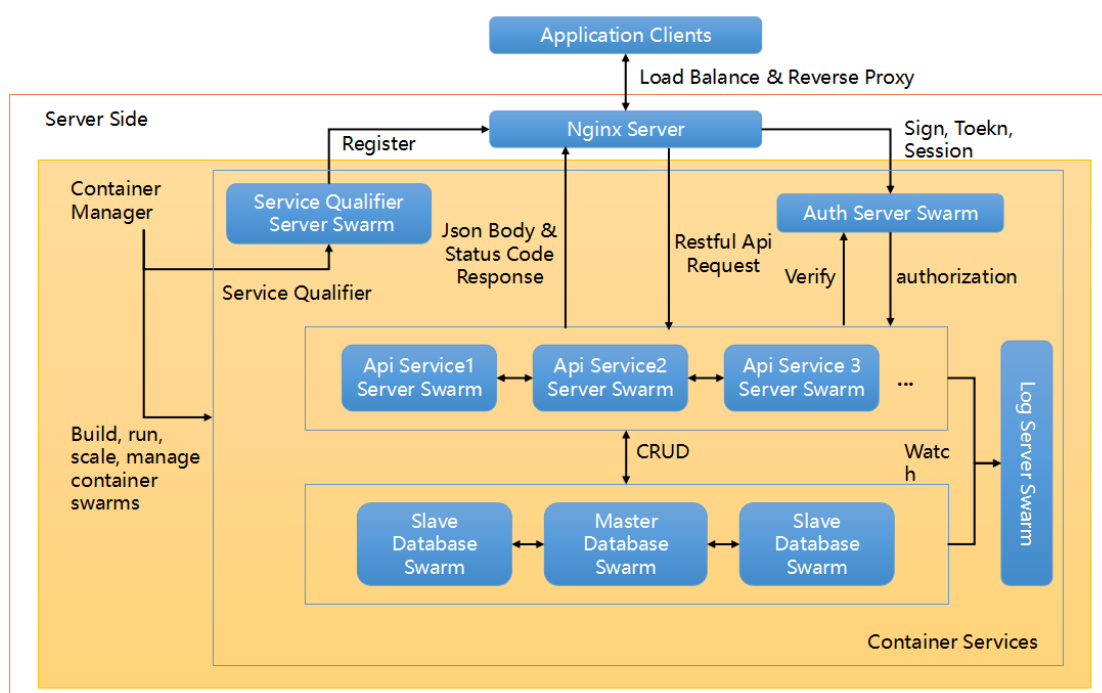


图 1-1 服务器集群架构图

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宕机情况，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，仍能响应用户业务访问，仅影响计算能力的下降。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。在现今流行的 SaaS 企业云平台更进一步解决集群部署困难的问题，也降低了企业维护的复杂度。

（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，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。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，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，解决 Server 端无法同时满足不同类型 Client 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

容器（Container）技术是一种基于 LXC 技术的虚拟化技术，也是目前实现服务器集群最佳实践方法^[18]。该技术通过隔离计算机资源，包括文件系统与网络端口等，构建一个包含环境、依赖库以及源代码的镜像，然后发布到 Docker Hub 等公开的镜像仓库中，使得用户可以在任意 Linux 系统的主机上拉取镜像然后直接运行，实现应用的虚拟化部署。容器与宿主机完全相互隔离，使其不管在任意环境的宿主机上都能直接运行。Container 技术与 VM 技术性能由于实现方式不同，因此最终性能有很大区别。Container 技术与 VM 技术相比，一个容器通常可以在很短的时间内就完成部署，而 VM 部署前需要先加载操作系统并启动，因此时间会耗费比较长，并且虚拟机需要再做一次代码解释，对系统资源会有一次额外开销。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：

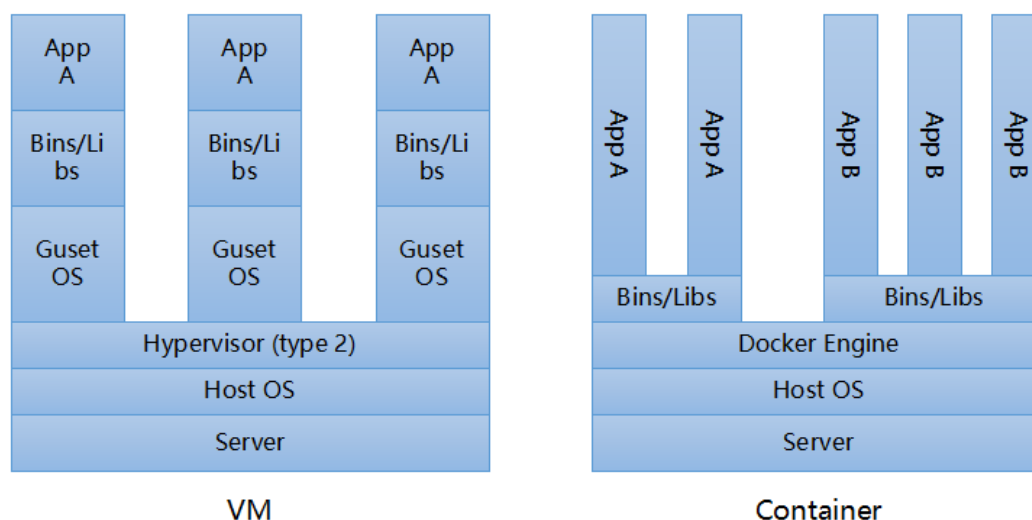


图 2-1 VM 与 Container 技术架构对比图

2.1.1 Docker 容器工具

Docker 是是目前容器技术中最有主流的代表，基于 go 语言开发的一个高级容器引擎^[19]。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。

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。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。

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、内外端口映射、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 可以轻松地在任意时刻对某个容器进行扩容。

2.2 容器管理工具

大量容器在一个集群上部署很明显会产生一系列的复杂问题，不仅会导致部署十分困难，也很难定位到故障位置，容器肯定是跟资源相匹配的。故障肯定是越快解决越好。这些问题的出现使得需要一个工具来简化整个集群容器的使用复杂度。

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。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。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。在一个分布式集群中，容器也需要是分布式的。Swarm 允许你在本地聚集 Docker 引擎。

再加上，Swarm 的角色相当于 Docker API。在它的核心内部是一个简单的系统：每台主机上通过启动一个 Swarm 客户端。管理员处理容器的操作和调度。Docker Swarm 自带了服务发现与均衡负载的功能，同时也可以接入第三方的服务发现工具，例如 Consul 或者 Zookeeper。

Docker Swarm 相比 Kubernetes 的一个显著特点是，它是 Docker 原生的一个工具，用户可以使用原生命令来操作集群的容器和数据卷。Swarm 管理员为 leader 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。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

均衡负载器（Load Balance）通过反向代理的方式，它将作为一个单一的 HTTP 访问入口点，接受所有的客户端服务 Request，然后根据预先设定的平衡策略，将请求路由到处理服务器集群上相同的处理服务器处理实例中^[20]。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：

```
FROM nginx:latest
RUN DEBIAN_FRONTEND=noninteractive \
apt-get update -qq && \
apt-get -y install curl runit unzip && \
rm -rf /var/lib/apt/lists/*
ADD nginx.service /etc/service/nginx/run
RUN rm -v /etc/nginx/conf.d/*
ADD nginx.conf /etc/consul-templates/nginx.conf
CMD ["/usr/bin/runsvdir", "/etc/service"]
```

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：

```
upstream app {
```

```

least_conn;
{{range service "development.api"}}
server {{.Address}}:{{.Port}} max_fails=3 fail_timeout=60 weight=1;
{{else}}server 127.0.0.1:65535; # force a 502{{end}}
}

server {
    listen 80 default_server;
    gzip on;
    keepalive_timeout 30;
    open_file_cache max=200000 inactive=20s;
    location / {
        proxy_cache cache;
        proxy_cache_key $scheme$proxy_host$uri$is_args$args;
        proxy_cache_valid 200 10m;
        proxy_cache_bypass $arg_nocache;
        add_header X-Cache-Status $upstream_cache_status;
        proxy_pass http://app;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
    }
}

```

2.4 服务发现

对于 Server 架构而言，服务发现功能是一个极其重要的组成部分之一。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: 应用可以利用 Consul 的键值对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, 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:

```
exec consul-template \
-consul=consul:8500 \
-template "/etc/consul-templates/nginx.conf:/etc/nginx/conf.d/app.conf:sv hup
nginx"
```

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:

```
ENV CT_URL https://releases.hashicorp.com/consul-template/0.12.2/consul-
template_0.12.2_linux_amd64.zip
```

```
ENV CT_FN consul-template.zip
```

```
RUN curl -L $CT_URL > $CT_FN && unzip $CT_FN -d /usr/local/bin && rm
$CT_FN
```


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。

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，并不是所有数据都必须实现强一致性，某些数据为了性能需求允许出现容差，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。

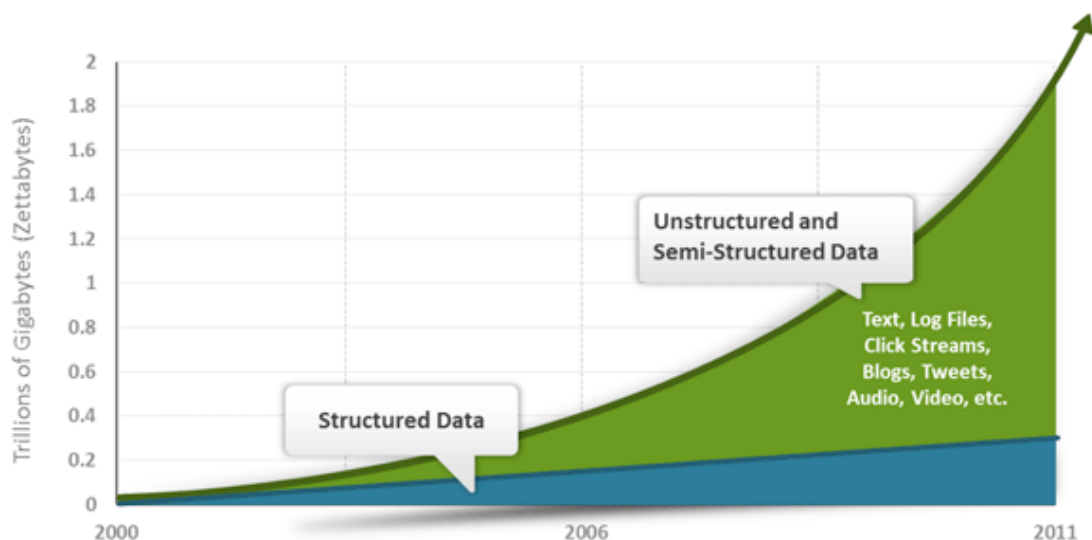


图 3-1 结构型与非结构型数据增长对比图

正是由于这种原因，非结构型数据库（NoSQL）诞生以解决该类问题，NoSQL 相比于 RDBMS 具有更松散的模式定义方式（Schema），对于模型中的每一个元素不一定具备有一个具体的数值，这种方式可以帮助 NoSQL 更加适应于未来业务的变更，而无需修改代码与做数据迁移^[21-22]。

NoSQL 与 RDBMS 的还有一类区别在于 NoSQL 可以通过更不严谨的一致性来应对更海量的数据查询与写入请求，NoSQL 解决了 RDBMS 对于海量数据无法在较短时间内进行处理的问题。实际业务中数据的存储可能并非密集的，而是

存在许多 Null 区域，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。此外，NoSQL 针对不同业务需求，通常是以下 4 种类型：

- 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^[23]。内存数据库将数据存储在内存中，与普通数据库存储在文件系统中有所区分。由于内存具备较高的传输速率与 Cache 命中，需要更少的 CPU 指令，可以直接随机访问，因此能够大量减少硬盘寻址寻道所浪费的时间。

3.2 MongoDB 数据库

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^[24]。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。MongoDB 内部为一个键值对的对象文档，类似于 JSON 对象，对象格式满足 JSON 标准格式，包括可以存储字符串、整数、浮点数和数组。

MongoDB 也支持大数据处理聚合操作，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

MongoDB 副本集是通过构建集群的方式，将一个实例中的存储数据实时同步给集群中的其他实例。副本集为所有存储的数据进行备份，并副本集群中的每个实例中同步副本，增强了数据库的可用性，提高了数据库的安全性。当发生错误和故障时，副本集可以无缝切换到任意一台实例上从而恢复工作。

MongoDB 的副本集常见的搭配方式为：一主一从或者一主多从，至少需要两个节点。选取一个 Leader 实例，接收处理服务器的数据操作请求，其他的作为 Secondary 从数据库，定期轮询 Leader 实例获得操作日志，再对本机副本执行日志行为，从而保证主从实例的数据一致。MongoDB 副本集结构图如下图 3-2 所示：

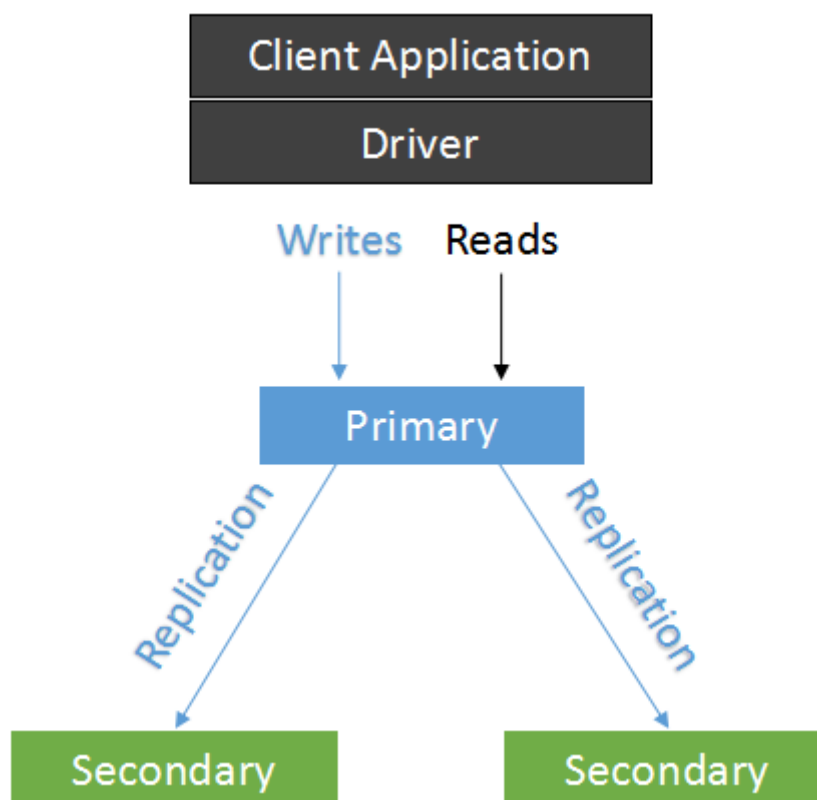


图 3-2 MongoDB 副本集结构示意图

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。Replica Sets 的结构类似于一个 NoSQL 集群，当其中任意一个实例错误或故障发生中断，副本集将会重新选取一个新的实例作为 Leader，并无缝切换到该实例继续工作。

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：

mongo-1-2:

container_name: "mongo-1-2"

image: mongo

ports:

- "30012:27017"

command: mongod --replSet rs1 --shardsvr --port 27017

restart: always

mongo-1-3:

container_name: "mongo-1-3"

image: mongo

ports:

- "30013:27017"

```

command: mongod --replSet rs1 --shardsvr --port 27017
restart: always
mongo-1-1:
  container_name: "mongo-1-1"
  image: mongo
  ports:
    - "30011:27017"
  command: mongod --replSet rs1 --shardsvr --port 27017
  links:
    - mongo-1-2:mongo-1-2
    - mongo-1-3:mongo-1-3
  restart: always

```

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：

```

mongo --host ${mongodb1}:${port} <<EOF
var cfg = { "_id": "${RS}", "members": [
  { "_id": 0, "host": "${mongodb1}:${port}" },
  { "_id": 1, "host": "${mongodb2}:${port}" },
  { "_id": 2, "host": "${mongodb3}:${port}" }]
};
rs.initiate(cfg, { force: true });
rs.reconfig(cfg, { force: true });
EOF

```

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

在 MongoDB 里面存在另一种集群，就是分片技术，可以满足 MongoDB 数据大量增长的需求。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，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^[25]。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。

分片是某一集合中负责某一子集的一台或多台服务器。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。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 3-3 所示：

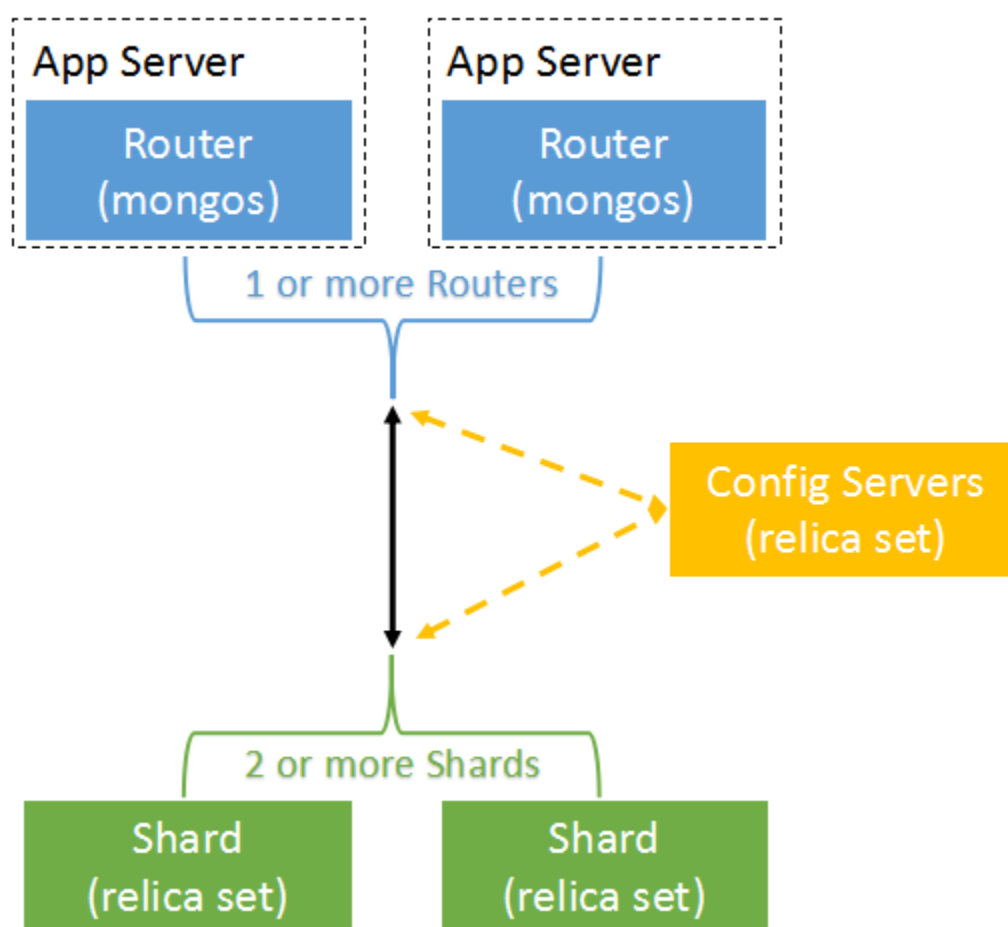


图 3-3 MongoDB 分片结构示意图

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：

```
mongo --host ${mongodb1}:${port} <<EOF
var cfg = { "_id": "${RS}", configsvr: true, "members": [
    { "_id": 0, "host": "${mongodb1}:${port}" },
    { "_id": 1, "host": "${mongodb2}:${port}" },
    { "_id": 2, "host": "${mongodb3}:${port}" } ]
};
rs.initiate(cfg, { force: true });
rs.reconfig(cfg, { force: true });
EOF
```

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：

```
mongos --configdb cnf-serv/mongo-cnf-1:27017,mongo-cnf-2:27017,mongo-cnf-3:27017
```

最后连接路由实例将副本集端口加入即可，命令如下：

```
mongo --host ${mongodb1}:${port} <<EOF
sh.addShard( "${RS1}/${mongodb1}:${PORT1},${mongodb12}:${PORT2},${mongodb13}:${PORT3}" );
sh.addShard( "${RS2}/${mongodb21}:${PORT1},${mongodb22}:${PORT2},${mongodb23}:${PORT3}" );
EOF
```

由此，便成功创建了由 2 个副本集各三台实际数据存储实例，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

MongoDB 是一种面向文档的数据库，以类似 JSON 结构化数据的方式保存数据。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。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 `fieldName: {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。

3.3 Redis 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）。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^[26]。与其他内存数据库一样，为了保证数据的存取速率，所有的数据都是直接在内存中操作。但有一点不同的是 Redis 会在一个固定周期内向文件系统同步内存的数据存储实现数据的持久化。

Redis 是一个高性能的 key-value 数据库。它提供了 Python, Ruby, Erlang, PHP 客户端，使用很方便。相比其他内存数据库，Redis 实现了主从（master-slave）模式。数据可以从主数据库向集群中其他的从数据库同步数据，同时从数据库可以作为其他链接的从数据库的主服务器。由于 Redis 的消息机制通过观察者模式，这使得 Redis 可执行单层树复制。

3.3.1 缓存 Session

Session 是服务器保存每个连接相关状态信息的一个存储实体，是针对无状态的 HTTP 协议而扩展的一个标准^[27]。Session 数据存储在服务端，需要一个键值来关联具体的数据地址与连接。通常服务器会在 Cookie 中记录一个 SessionId 的值。每次连接时，服务端在 Cookie 中获取 SessionId 然后在数据库中查询相关的状态数据。随着企业用户规模的扩增，由于每次一个客户端的连接都要去数据库中查询 Session，因此传统数据库的查询将会导致大量的磁盘 I/O 操作造成线程阻塞。

基于这种背景，本文使用 Redis 将 Session 数据存储在内中从而实现 Session 的分布式存储。在关于 Session 的使用流程中，Server 与 Client 对于 Session 的获取和传递过程如下：

- 1、Server 检查 Client 的 Cookie 中是否具有 SessionId
- 2、有 SessionId，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 SessionId，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 SessionId。

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：

```
server.use(session({
  store: new RedisStore(options),
  secret: 'NANJINGYOUDIAN',
  resave: true,
  saveUninitialized: true
}));
```

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。

第四章 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^[28]，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。

服务器为了应付同一时间段内大量客户端请求并发访问的情况，通常会编写异步非阻塞的 IO 模型。但可能会导致一个问题，那就是每个访问请求到来时，服务器都需要创建一个新的线程来处理该请求。否则当某线程阻塞时，便无法继续处理其他的请求。当成千上万的超高并发下，服务器是无法创造相同数量的线程的，因为每个线程还是会额外消耗系统资源的，这会造成过多的资源被浪费。因此对于整个代码逻辑都需要采用回调的异步逻辑来编写，而这就是 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。

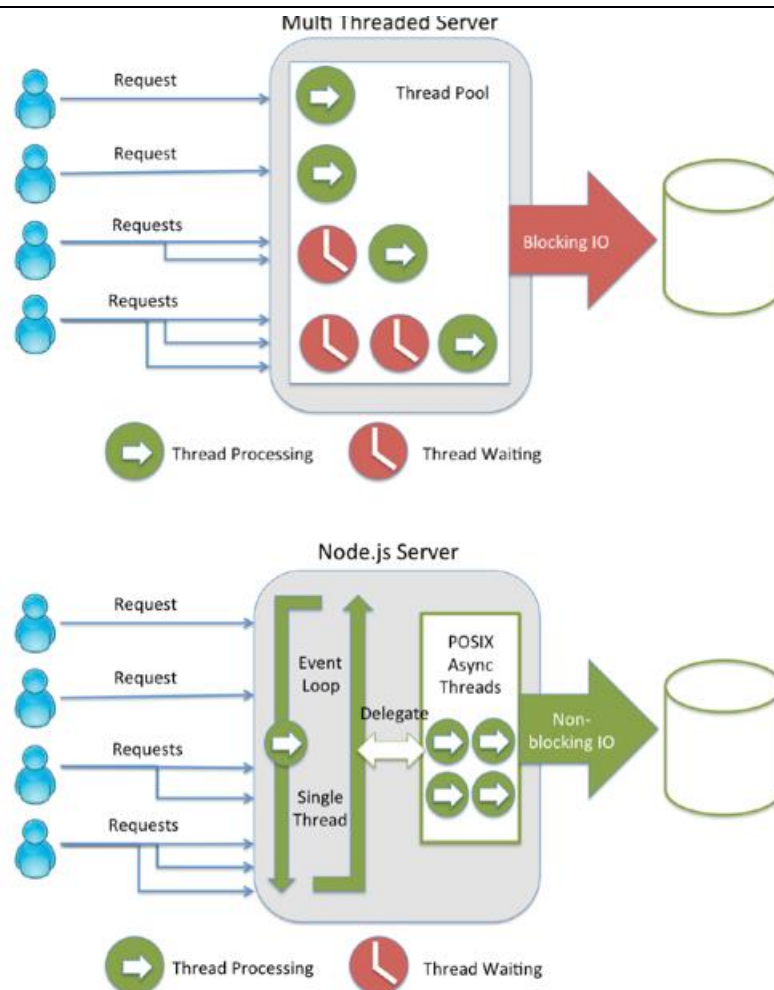


图 4-1 多线程服务器与 Node.js 服务器对比图

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，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^[29]。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。V8 引擎执行 Javascript 的速度非常快，性能非常好。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。

Node.js 通过模块来区分不同功能的组件，以中间件的方式简化应用的开发流程。每一个 Node.js 的模块基于 Node.js 一些基本的函数所构建，包含一些独有的功能。NPM 是 Node.js 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：

表 4-1 本文服务器依赖的 Node.js 模块

作用	模块名	备注
实现服务端	express	构建 Restful 服务器基本框架

Restful API 接口	express-session	储存与处理 Session 信息
	cookie-parser	解析 Cookie 数据
	body-parser	解析 Http Body 数据
	node-restful	构建 Restful API 接口
简化异步逻辑	bluebird	实现 Promise Lambda 处理异步逻辑
用户登录模块实现	connect-ensure-login	Session 中标记用户登录状态
	bcrypt-nodejs	密码加密
OAuth2.0 协议实现	passport	Passport 授权与验证基础框架
	passport-local	基于 Passport 实现用户账号密码登录验证
	passport-http-bearer	基于 Passport 实现 Bearer Token 验证
	passport-oauth2-client-password	基于 Passport 实现用户密码模式授权
	passport-http	基于 Passport 实现可信客户端模式授权
	oauth2orize	基于 Passport 实现授权码模式授权
	uuid	创建唯一性 Token
MongoDB 数据库	connect-mongo	连接 MongoDB 数据库
	mongoose	调用 MongoDB 的 API 完成 CRUD
	jsonschema	构建 Mongo Schem
Redis 数据库	connect-redis	连接 Redis 数据库
	hiredis	实现 Redis 的分布式 Session 存储
	redis	调用 Redis 数据库 API 完成 CRUD
测试与监控	mocha	Node.js 测试框架
	chai	断言工具
	newrelic	性能监控工具

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。

在现代 web 应用程序中，身份验证可以采用多种形式。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，其中 OAuth 是一种认证与授权的协议^[30]。OAuth2 在 Client 与 Server 之间，规定了一个授权层（authorization layer）。Client 不能直接通过请求在 Server 端获取数据，只能先在授权层获取授权，基于将 User 与 Client 作区分使得应用不会获取用户的密码。服务端授权后分发令牌（token）给客户端，以供客户端保存并以此请求数据。服务端需要在用户请求令牌时，设定 Token 的权限范围和过期时间。Client 登录授权层以后，Server 收到令牌后，先检验请求是否超时以及是否在授权范围内，而后接受 Client 相应的数据请求并作出相应。OAuth 2.0 定义了四种授权方式^[30]：

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）

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。

Passport 认为每个应用程序都有独特的身份验证需求。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，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，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：

```
passport.use(new LocalStrategy(function(username, password, done) {
  redis.hgetall('user:' + username, function(err, user) {
    if (err) { return done(err); }
    if (!user) { return done(null, false, { message: 'Unknow User' }); }
    if (!utils.validEncrypt(password, user.password)) { return done(null, false); }
    return done(null, user);
  });
}));
```

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，编写登录与登出的 Post 请求路由路径，如下所示：

```
router.route('/login').post(passport.authenticate('local', { successReturnToOrRedirect:
'/login/status/success', failureRedirect: '/login/status/failure' }));
router.route('/logout').get(function(req, res) { req.logout(); res.status(200).json('logout
success');});
```

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，成功或失败后重定向到相应的链接完成登录。登出通过在 request 中清除用户的 Cookie 信息，完成登出过程。

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。在授权码模式流程中，Client 端不会直接向数据服务器请求 Token 授权，而是把应用导向 Authorization 服务器的授权层请求许可，Authorization 服务器再通过 Redirect URI 重定向来颁发 Client 端授权许可码（code）。在重定向之后，Authorization 服务器会先认证授权码并颁发令牌给客户端以授权，因为处理服务器只与 Authorization 服务器认证所以 Client 端不能获取到 User 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 Authorization 服务器请求 Access Token 的授权。整个的流程具体如下图 4-2 所示：

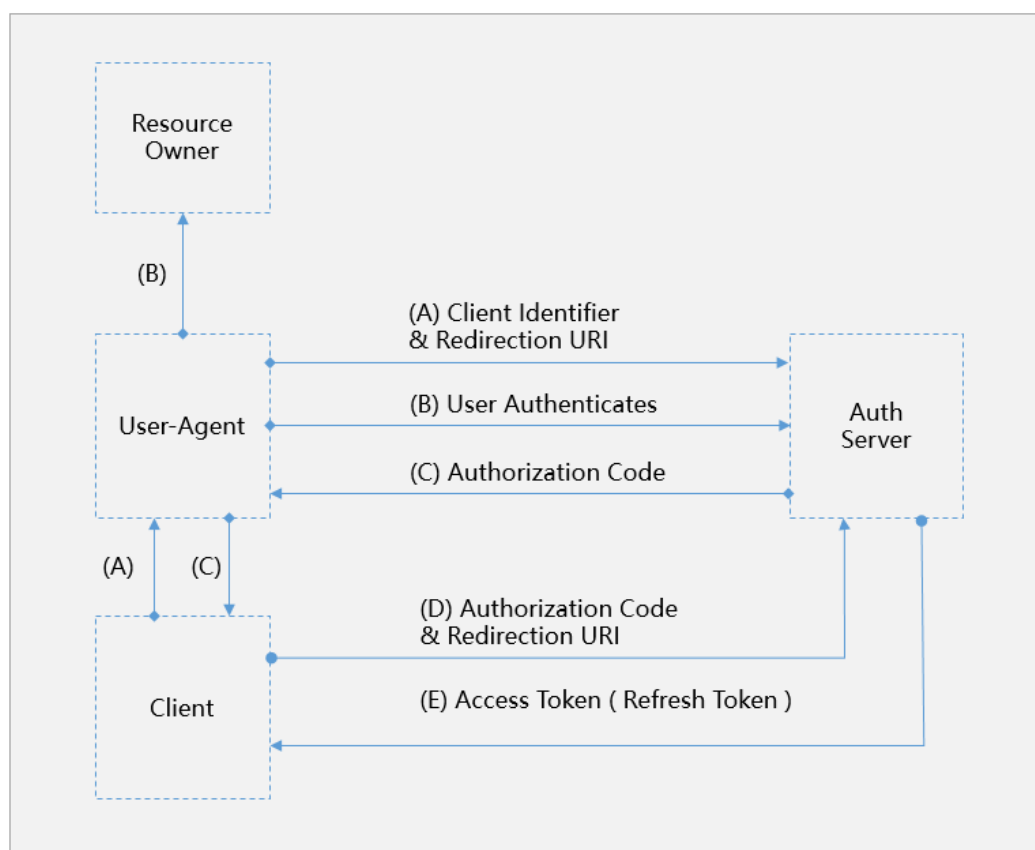


图 4-2 授权码模式授权流程图

(A) 客户端把用户的 User-Agent 转发到 Authorization 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 Authorization 服务器授权结果。

(B) Authorization 服务器通过 User-Agent 验证用户是否合法，并确认处理服务器许可或者驳回客户端的授权请求；

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

(D) Client 向 Authorization 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) Authorization 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 Authorization 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：

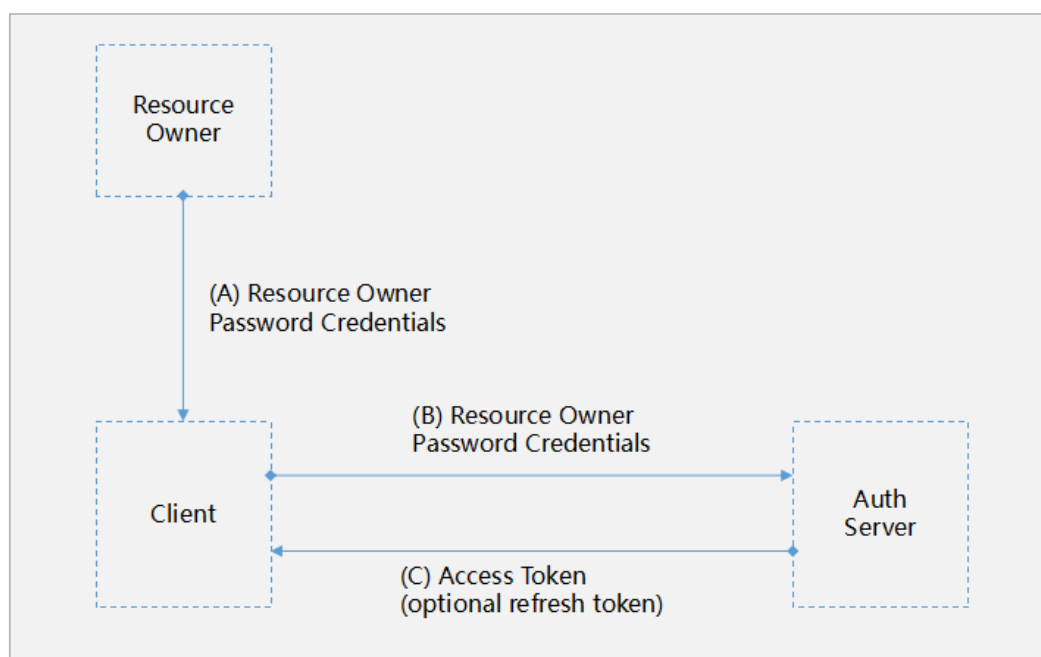


图 4-3 密码模式授权流程图

（A）用户向 Client 直接输入真正的帐号与密码；

（B）Client 使用 User 的帐号密码与自身 Client 的 ID 和 Secret，向 Authorization 服务器申请 Access Token 认证；

（C）Authorization 服务器确认 Client 信息与用户的帐号密码后，如果正确则核发 Access Token 给 Client

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。

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^[31]。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，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 XXXXXXXXX，其中 XXXXXXXXX 的格式为 b64token，b64token 的定义：

$$\text{b64token} = 1 * (\text{ALPHA} / \text{DIGIT} / \text{"-"} / \text{"."} / \text{"_"} / \text{"~"} / \text{"+"} / \text{" /"}) * "="$$

写成 Regular Expression 即是： $/[A-Za-z0-9\-\._\~\+\/]+=*/$

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 W3C HTML 4.01 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援这个方式。

(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，或是使用了不支援的认证方式（例如不支援 URI parameter）），则 response 不应该带 error 或任何错误讯息。

4.3 Restful API 接口实现

Restful(REST)或 RESTful API 接口是在 Internet 上提供数据资源操作的一种方式^[32]。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。2000 年, Roy Fielding 在他的博士论文中提出了“表征状态”的概念。使用 REST 来设计 HTTP 1.1 和统一资源标识符 (URI)。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。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。

4.3.1 基于 Express 框架开发

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^[33]。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。

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：

```
const server = express()
server.use(bodyParser.urlencoded({extended: true}));
server.use(bodyParser.json());
server.use(cookieParser());
server.use(passport.initialize());
server.use(passport.session());
const httpServer = http.createServer(server);
const httpsServer = https.createServer(options, server);
httpServer.listen(80,null);
httpsServer.listen(443,null);
```

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。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。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。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`
- 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`
- 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 `https` 的请求端口，代码如下：

```
var options = {
  pfx: fs.readFileSync('../keys/server.pfx'),
  passphrase: 'guojun@123'
};
```



```
const httpServer = http.createServer(server);
const httpsServer = https.createServer(options, server);
httpServer.listen(80, null);
httpsServer.listen(403, null);
```

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：

```
GET /resources
GET /resources/:id
POST /resources
PUT /resources/:id
DELETE /resources/:id
```

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：

```
Resource.before('post', hash_password).before('put', hash_password);
function hash_password(req, res, next) {
  req.body.password = bcrypt.hashSync(req.body.password,
    bcrypt.genSaltSync(), null); next();
}
```

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：

```
Resource.before('get', passport.authenticate('bearer', { session: false }))
.before('post', passport.authenticate('bearer', { session: false }))
.before('put', passport.authenticate('bearer', { session: false })))
```

```
.before('delete', passport.authenticate('bearer', { session: false }));
```

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：

```
Resource.before('get', function(req, res, next) {
  if (req.query.sort === "-createTime" && req.query.limit <
    redis.llen('Resource')){
    var cache = redis.lrange('Resource', req.query.skip, req.query.limit);
    if (cache) return res.status(200).json(cache);
    else next()
  }
})
```

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 是请求的返回状态码，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：

```
Resource.after('post', function(req, res, next) {
  var cache = redis.lrange('Resource',0,redis.llen('Resource'))
  cache.pop();
  cache.unshift(res.locals.bundle.toJSON())
}
```

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：

表 4-2 本文服务器依赖的 Node.js 模块

过滤条件	查询请求	示例	描述
limit	limit	/users?limit=5	返回前 5 个用户数据
skip	skip	/users?limit=5&&skip=5	返回第 5 到到第 10 个用户数据
sort list by field	sort	/users?sort="-age"	返回根据年龄倒序排序的所有用户
equal	equals	/users?gender=male	返回所有男性用户
not equal	ne	/users?gender_ne=male	返回所有不是男性的用户
greater than	gt	/users?age_gt=18	返回年龄大于 18 的用户
greater than or equal to	gte	/users?age_gte=18	返回所有年龄大于等于 18 的用户

less than	lt	/users?age_lt=30	返回年龄小于 30 的用户
less than or equal to	lte	/users?age_lte=30	返回年龄小于等于 30 的用户
in	in	/users?gender_in=female,male	返回所有的男性或女性用户

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

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 node.js 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：

```
apps:
  - script : './server/server.js'
    name   : 'API Server'
    exec_mode: 'cluster'
    instances: 'max'
    watch  : ['server/']
```

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。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，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：

api:

command: pm2-docker --watch ./config/process.yml

build .

image: gjscut/water_watcher_api_server

environment:

SERVICE_NAME: api

SERVICE_TAGS: development

NODE_ENV: development

NODE_CONFIG_DIR: ./config

NODE_APP_INSTANCE: "

links:

- redis:redis

- mongo-router:mongodb

ports:

- "80:80"

- "443:443"

volumes:

- ./docker-node-express-boilerplate

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。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，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：

```
RUN docker swarm join --token SWMTKN-1- $\{TOKEN\}$   $\{MANAGER-IP\}$ 
COUNTER= 1
WHILE [  $\$`docker node list`$  ]
DO COUNTER='expr $COUNTER+1' DONE
RUN docker service scale api=$COUNTER
```

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：

```
consul:
  image: gliderlabs/consul-server:latest
  command: -server -bootstrap
  ports:
    - "8500:8500"
  restart: always

registrator:
  image: gliderlabs/registrator:master
  links:
    - consul
  command: -internal -resync 600 consul://consul:8500
  volumes:
    - "/var/run/docker.sock:/tmp/docker.sock"
  restart: always
```

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

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。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。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：

- 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；
- 2、软件的需求都被测试代码描述得很清楚，可以减少很多不必要的文档；
- 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。

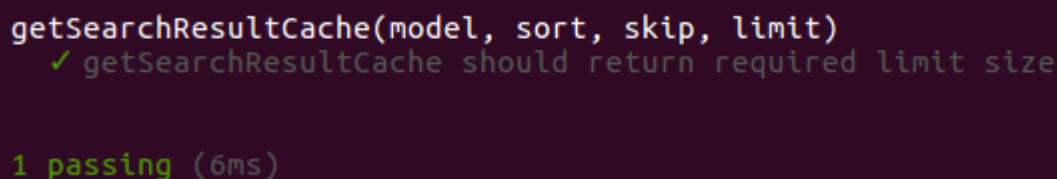
基于 `node.js` 平台的服务器我们可以采用 `Mocha` 和 `Chai` 这两个测试框架实现本文服务端的 `TDD` 测试，`Mocha` 是一个单元测试框架，和其他的 `javascript` 单元测试框架不同的是，他没有 `assertion` 库。但是，`Mocha` 允许使用第三方的断言库，因此我们使用 `Chai` 对测试结果进行断言，下面我们给出一个使用 `Mocha` 和 `Chai` 完成 `Redis` 查询缓存读取的测试示范：

```
describe('getSearchResultCache(model, sort, skip, limit)', function() {
  it('getSearchResultCache should return required limit size', function() {
    var model = 'pumpRoomFirst', sort = '-createTime', skip = 0, limit = 360;
    var results = getSearchResultCache(model, sort, skip, limit);
    expect(results.length).to.equal(limit);
    sort = 'createTime';
    results = getSearchResultCache(model, sort, skip, limit);
```

```
expect(results.length).to.equal(0);  
limit = 100000;  
results = getSearchResultCache(model, sort, skip, limit);  
expect(results.length).to.equal(0);  
});  
});
```

`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。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。

运行测试只需要在项目的根目录运行命令行：`mocha tests --recursive --watch`。`recursive` 指明会找到根目录下的子目录的测试代码并运行。`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。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：



```
getSearchResultCache(model, sort, skip, limit)  
✓ getSearchResultCache should return required limit size  
  
1 passing (6ms)
```

图 5-1 单元测试示例测试结果图

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，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。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;

```
curl --request POST \
  --url http://{{address}}:{{port}}/oauth/token
  --header 'content-type: application/x-www-form-urlencoded'
  --data
'grant_type=password&client_id=admin&client_secret=123456&username=admin&password=admin'
```

然后，在 Tests 中，通过;

```
postman.setEnvironmentVariable("token", data.token_type + " " +
data.access_token);

postman.setEnvironmentVariable("refreshToken", data.refresh_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 5-2:

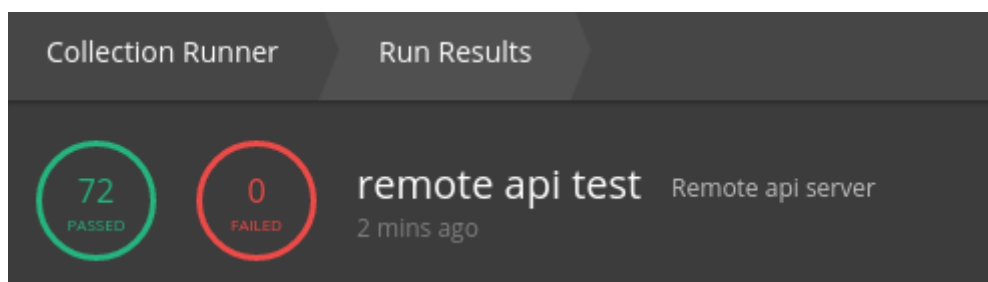


图 5-2 接口测试结果图

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 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, APM）New Relic 对服务器端进行实时的监控与性能分析。

New Relic 是一款基于 SaaS 的远端服务器监控与分析平台^[34]，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代码抽象出来，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，并且分析接口的 CPU 占用时间来帮助开发人员监控与分析应用的性能状态。

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 5-3 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。

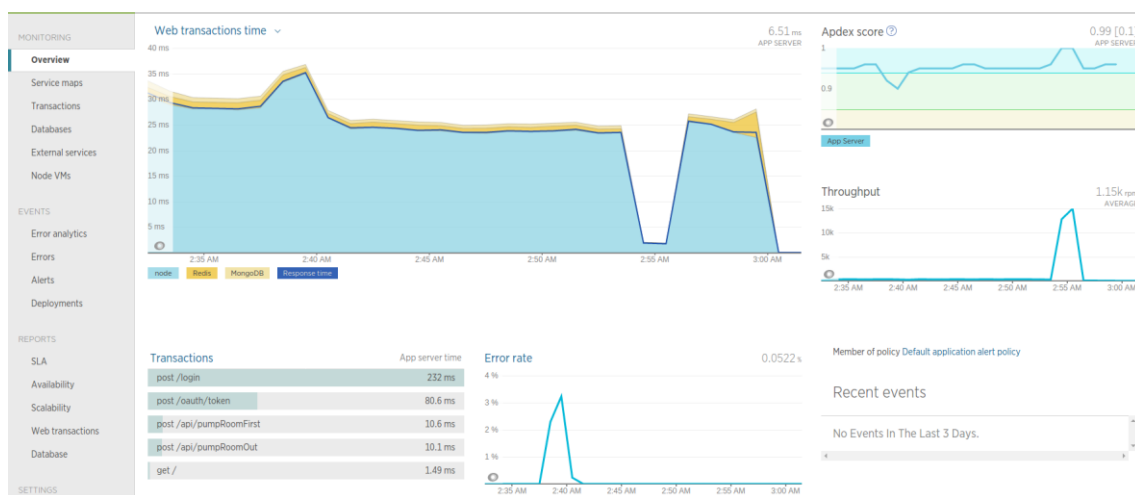


图 5-3 New Relic 总体监控概括图

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。此外 New Relic 还提供了对单个接口性能分析的功能，如下图 5-4 所示：

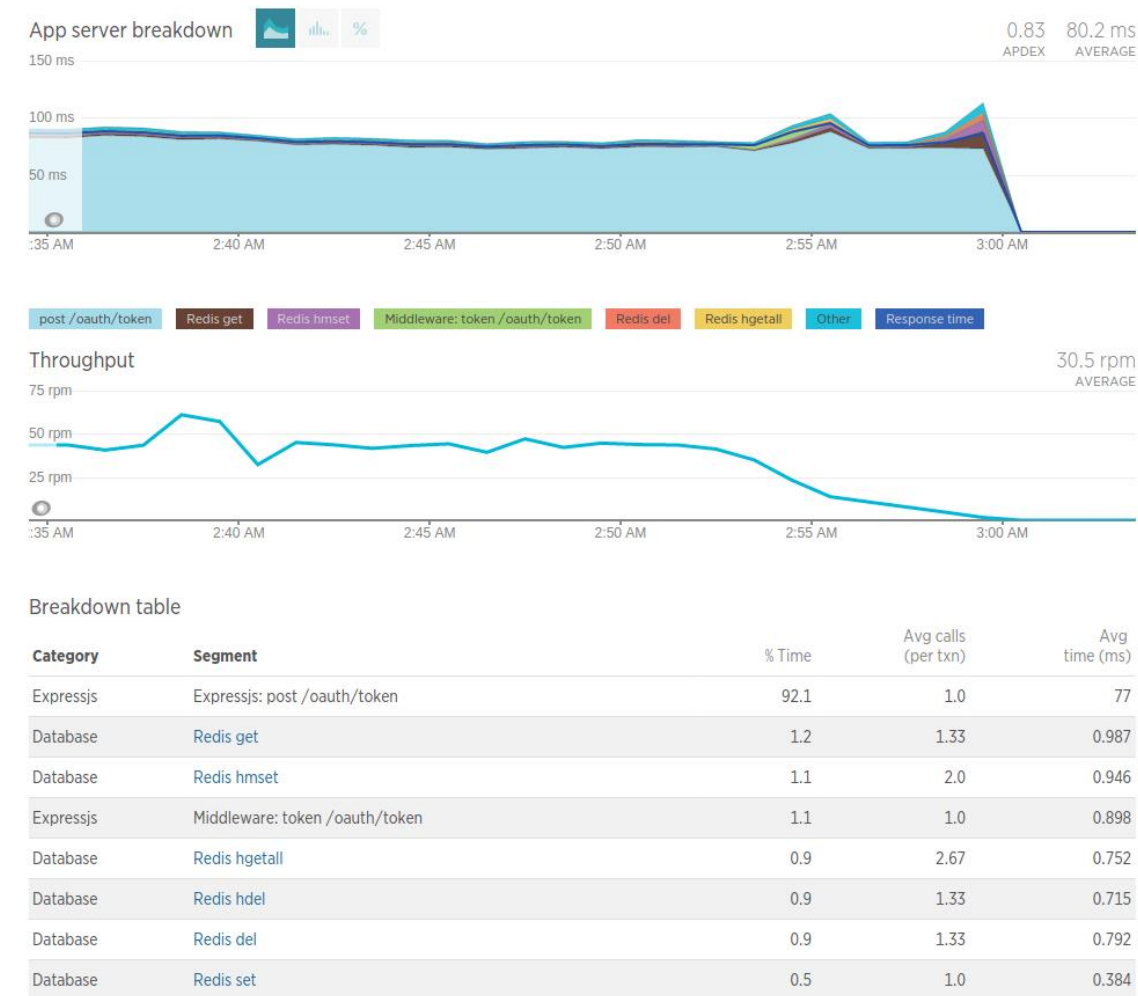


图 5-4 New Relic 接口性能分析图

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。

5.3.2 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，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。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。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：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。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：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）。这个指标代表某个时刻 Server 端同时接受到的并发请求数量；

3、并发用户数（The number of concurrent users, Concurrency Level）。概念：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：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：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。同时，它也=用户平均请求等待时间/并发用户数。

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。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 @ 2.60GHz 的单主机服务器环境与 CPU 为 Intel(R) Core(TM) i5-6600 CPU @ 3.30GHz 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：

表 5-1 访问负载压力测试数据结果表

测试环境	CPU E5-2650 单核单虚拟主机			CPU i5-6600 单核四主机集群		
并发连接数	10	100	100	10	100	1000
吞吐率 (req/s)	372.80	370.50	357.23	2451.59	2357.29	2163.82
用户平均请求等待时 (ms)	26.824	269.904	2799.287	4.079	42.422	462.145
服务器平均请求等待时间 (ms)	2.682	2.699	2.799	0.408	0.424	0.462
请求时间分布 (ms)	50% 2.41 66% 2.43 80% 2.82 90% 3.94 95% 4.15 98% 4.67 99% 5.94 100% 10.61	50% 2.56 66% 2.58 80% 2.64 90% 2.78 95% 3.14 98% 3.62 99% 3.88 100% 3.93	50% 2.56 66% 2.69 80% 2.74 90% 2.99 95% 3.41 98% 3.86 99% 3.95 100% 4.03	50% 0.30 66% 0.42 80% 0.47 90% 0.53 95% 0.61 98% 0.64 99% 1.24 100% 1.63	50% 0.40 66% 0.43 80% 0.53 90% 0.57 95% 0.63 98% 0.69 99% 1.39 100% 2.51	50% 0.41 66% 0.44 80% 0.56 90% 0.61 95% 0.69 98% 0.88 99% 1.72 100% 6.73
传输速率 (Kb/sec)	190.04	188.87	182.11	1249.74	1201.66	1158.24

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，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。
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著增强服务器的计算性能，以满足更高的用户访问负载压力。

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，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。

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。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；

（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；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

这次毕业设计能够顺利完成离不开王冬生老师以及负责客户端开发的同学们的讨论与帮助，正是基于他们的支持才能让我在困难的道路上坚持不懈，勇攀高峰。在本科毕设完成之际，我要对给予我大量帮助与建议的王冬生老师表示衷心的感谢。王老师在整个指导我们毕设的过程中，明确的给我们所有人定下了立项目的与方案，帮助我们确定了技术路线，并组织我们开展多次讨论最终确定了实现方案。同时王老师渊博的知识与认真的工作态度也感染了我们，让我们受益匪浅，明白了许多相关学术知识和科研方法。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。

还要感谢的是大学四年来所有的任课老师，是他们的辛勤劳动、谆谆教导造就了我今天毕设的成果。

最后我要感谢的是我男朋友郭同学对我技术上和学术上的指导，在他的帮助下帮我解决了许多论文中的困惑与技术难点，才能最终完成所有代码。期望我与他能够在人生道路上继续互相扶持，携手共进。

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

- [1] 张友生, 陈松乔. C/S 与 B/S 混合软件体系结构模型[J]. 计算机工程与应用, 2002, 38(23):138-140.
- [2] G. Gruman, "What cloud computing really means", InfoWorld, Jan. 2009.
- [3] R. Buyya, Y. S. Chee, and V. Srikumar, "Market-Oriented Cloud Computing: Vision, Hype, and Reality for Delivering IT Services as Computing Utilities", Department of Computer Science and Software Engineering, University of Melbourne, Australia, July 2008, pp. 9.
- [4] D. Chappell, "A Short Introduction to Cloud Platforms", David Chappell & Associates, August 2008.
- [5] Cusumano M. Cloud computing and SaaS as new computing platforms[J]. 2010, 53(4):27-29.
- [6] E. Knorr, "Software as a service: The next big thing", InfoWorld, March 2006.
- [7] Cardellini V, Colajanni M, Yu P S. Dynamic load balancing on Web-server systems[J]. IEEE Internet Computing, 1999, 3(3):28-39.
- [8] Choi E. Performance test and analysis for an adaptive load balancing mechanism on distributed server cluster systems[J]. Future Generation Computer Systems, 2004, 20(2):237-247.
- [9] Brendel J, Kring C J, Liu Z, et al. World-wide-web server with delayed resource-binding for resource-based load balancing on a distributed resource multi-node network: US, US5774660[P]. 1998.
- [10] Bowman-Amuah M K. Load balancer in environment services patterns: US, US6578068[P]. 2003.
- [11] Newcomer E, Lomow G. Understanding SOA with Web Services (Independent Technology Guides)[J]. 2005, 25(4):72.
- [12] Martin Fowler. Microservices a definition of this new architectural term [EB/OL]. (2014-03-14).https://martinfowler.com/articles/microservices.html?utm_source=tuicool&utm_medium=referral
- [13] Chieu T C, Mohindra A, Karve A A, et al. Dynamic Scaling of Web Applications in a Virtualized Cloud Computing Environment[C]// IEEE International Conference on E-Business Engineering. IEEE Xplore, 2009:281-286.
- [14] 孙海洪. 微服务架构和容器技术应用[J]. 金融电子化, 2016(5):63-64.
- [15] 李忠民, 齐占新. 业务架构的微应用化与技术架构的微服务化——兼谈微服务架构的实施实践[J]. 科技创新与应用, 2016(35):95-96.
- [16] 张宁溪, 朱晓民. 基于 Docker、Swarm、Consul 与 Nginx 构建高可用和可扩展 Web 服务框架的方法[J]. 电信技术, 2016(11):21-25.
- [17] 卢胜林, 倪明, 张翰博. 基于 Docker Swarm 集群的调度策略优化[J]. 信息技术, 2016(7):147-

151.

- [18] 杜军. 基于 Kubernetes 的云端资源调度器改进[D]. 浙江大学, 2016.
- [19] 陈清金, 陈存香, 张岩. Docker 技术实现分析[J]. 信息通信技术, 2015(2):37-40.
- [20] Decandia G, Hastorun D, Jampani M, et al. Dynamo: amazon's highly available key-value store[J]. ACM SIGOPS Operating Systems Review, 2007, 41(6):205-202.
- [21] 张征, 李小丽. NoSQL 数据库数据模型以及系统介绍[J]. 科学与财富, 2015, 7(36):227-227.
- [22] Moniruzzaman A B M, Hossain S A. NoSQL Database: New Era of Databases for Big data Analytics - Classification, Characteristics and Comparison[J]. International Journal of Database Theory & Application, 2013, 6.
- [23] 王珊, 肖艳芹, 刘大为,等. 内存数据库关键技术研究[J]. 计算机应用, 2007, 27(10):2353-2357.
- [24] Chodorow K, Dirolf M. MongoDB: The Definitive Guide[J]. 2010.
- [25] 王胜, 杨超, 崔蔚,等. 基于 MongoDB 的分布式缓存[J]. 计算机系统应用, 2016, 25(4):97-101.
- [26] Carlson J L. Redis in Action[J]. Media.johnwiley.com.au, 2013.
- [27] Lahtinen E J A. Using Session Replication in Web Services[J]. Mikkonen Tommi, 2015.
- [28] 胡启敏, 薛锦云, 钟林辉. 基于 Spring 框架的轻量级 J2EE 架构与应用[J]. 计算机工程与应用, 2008, 44(5):115-118.
- [29] Tilkov S, Vinoski S. Node.js: Using JavaScript to Build High-Performance Network Programs[J]. IEEE Internet Computing, 2010, 14(6):80-83.
- [30] D. Hardt E. The OAuth 2.0 Authorization Framework[J]. Ietf Rfc, 2012.
- [31] T. Lodderstedt E, McGloin M, Hunt P. OAuth 2.0 Threat Model and Security Considerations[J]. 2013.
- [32] Pautasso C. RESTful Web service composition with BPEL for REST[J]. Data & Knowledge Engineering, 2009, 68(9):851-866.
- [33] Archer R. ExpressJS: Web App Development with Node.js Framework[M]. CreateSpace Independent Publishing Platform, 2015.
- [34] in. New Relic Security Overview[J]. 2015.