



# 课 程 设 计

设计题目 RAT 远程管理工具

学生姓名 毛冬阳

学 号 201910311127

专业班级 网络 191

指导教师 高 军

2021 年 12 月 13 日

## 摘要

本设计致力于解决对不具有公网 IP 的物联网设备进行远程管理，密切关注物联网设备的数量庞大，机能有限，流量不富裕，且设备的低可靠性。力图设计出一种，以网页的方式，借助服务器的公网 IP，以反弹 Shell 方式，便捷的在网页上实现对物联网设备的管理。对比现有的远程登录方案，ssh 需要被管设备有公网 IP，这对于庞大的物联网设备显然是不可能的。SNMP 需要设备处于同一局域网中，VPN 需要配置，并不是随时都可用的。我们的设计如果达到以下几点，就在现有解决方案中占有优势。

1. 部署配置简单，除了一个云服务器外，不需要更多的投资。
2. 通过网页就可以便捷地对设备进行管理。
3. 保证物联网设备上的可用与低成本。
4. 开源免费。

关键字： 远程管理工具 物联网 反弹 Shell

## Abstract

This design is dedicated to solving the remote management of IoT devices that do not have public IP, and pay close attention to the large number of IoT devices, limited functions, low traffic, and low reliability of the equipment. Trying to design a web page, with the help of the server's public IP, and a rebound Shell method, to conveniently implement the management of the Internet of Things devices on the web page. Compared with the existing remote login solution, ssh requires the managed device to have a public IP, which is obviously impossible for huge IoT devices. SNMP requires devices to be in the same local area network, and VPN needs to be configured, which is not always available. If our design achieves the following points, it will have an advantage in the existing solutions.

1. The deployment configuration is simple, and no more investment is required except for a cloud server.
2. The device can be easily managed through the web page.
3. guarantees availability and low cost on IoT devices.
4. Open source and free.

**Key words:** Remote management tools Internet of Things Rebound Shell

目录

<b>1</b>	<b>引言</b>	<b>1</b>
1.1	问题场景概述 . . . . .	1
<b>2</b>	<b>相关技术</b>	<b>1</b>
2.1	客户端 . . . . .	1
2.2	服务端 . . . . .	4
2.3	Web 端 . . . . .	11
2.4	项目管理方式 . . . . .	12
2.5	部署方式 . . . . .	14
<b>3</b>	<b>具体实现</b>	<b>16</b>
3.1	Client . . . . .	16
3.2	Server . . . . .	19
3.3	Web . . . . .	24
<b>4</b>	<b>测试使用</b>	<b>28</b>
<b>5</b>	<b>展望</b>	<b>32</b>

# 1 引言

## 1.1 问题场景概述

本设计致力于解决物联网问题, 物联网 (Internet of Things , IoT) 作为一种新兴网络技术和产业模式, 在业界受到广泛关注. 从国际电信联盟 (ITU) 在信息社会世界峰会上发布的《互联网报告 2005 : 物联网》中可以总结出物联网所体现的两层基本涵义:(1) 目前的三大网络, 包括互联网 (Internet)、电信网、广播电视网是物联网实现和发展的基础, 物联网是在三网基础上的延伸和扩展;(2) 用户应用终端从人与人之间的信息交互与通信扩展到了人与物、物与物、物与人之间的沟通连接, 因此, 物联网技术能够使物体变得更加智能化. 从目前的发展形势看, 最有可能率先获得智能连接功能的物体包括家居设备、电网设备、物流设备、医疗设备以及农业设备, 并基于此实现人类与自然环境的系统融合. [1] 无论物联网技术有什么样的展望, 第一步总是先要将设备连上网进行控制, 考虑到物联网设备并非图形化界面, 而是通过 Shell 进行控制. 所以建立一个有效的远程 Shell 连接就非常重要. 而本设计, 就是希望解决物联网设备没有公网 IP 仍要通过 Shell 连接的痛点. 当然也适用于所有在内网, 需要通过公网进行访问的计算机。

# 2 相关技术

## 2.1 客户端

此客户端指的是, 终端被管设备, 可以是各种架构, 运行各类操作系统. 在此场景中, 是机能受限的 IoT 设备, 通过 Websocket 的方式与服务端连接, 将 Shell 传递给服务端. 客户端开始需要注册到服务端上, 然后通过 Websocket 的进行长连接, 通过心跳包监控链路状态. 客户端保留设备信息与登录状态, 并随时等待 Web 端发送的命令. 客户端程序通过 Kotlin 语言进行编写, 利用 Kotlin 语言能够兼容 Java 的 Spring Boot 框架, 并且运行在 JVM 上, 具有 Java 的”一次编译到处运行”的特性. 可以运行在各类平台上. 此处我们用到了集成 Websocket 的 Spring 框架. 语言开源网址: <https://spring.io/projects/spring-boot>, 让我们可以方便的使用

Websocket 协议。下面我来介绍, 我们使用 Websocket 而不是 HTTP 的考量。

### 2.1.1 Websocket 通信协议

Websocket 是 html5 中的一种新协议, 它可以实现真正意义上的浏览器与服务器之间的全双工通信, 服务器可以主动实时地将消息推送至浏览器端。浏览器与服务器之间只需经过一次握手连接建立一条通道, 此后服务器就可以通过这条通道推送消息。Websocket 协议的工作过程如图2.1.1所示。[2]

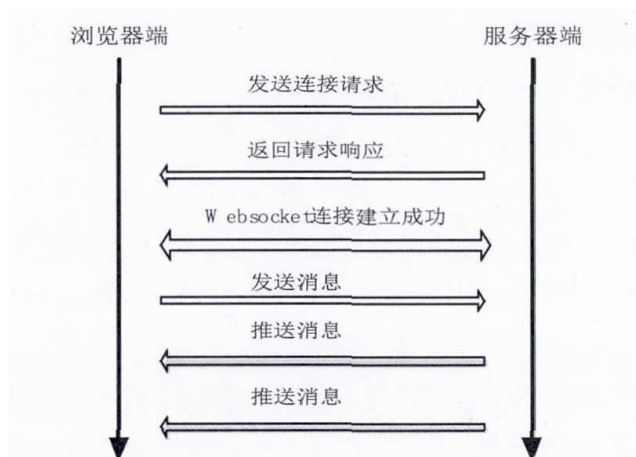


图 2.1: Websocket 协议工作流程

不难发现, Websocket 连接建立成功后, 服务端是可以主动的向客户端推送消息。不同于 http 协议的工作模式是“请求—响应”, 即用户通过浏览器客户端主动发出数据请求, 服务器处理请求后将响应发送给用户, 在这种模式下, 服务器并不能主动推送消息至客户端, 使得消息的实时推送成为困难。常见的解决方案有基于轮询技术的消息推送和基于 http 长连接的 comet。但基于轮询的消息推送是一种伪服务器实时推送, 实际上是一种客户端拖拽的方式。其实现原理是浏览器每隔一段时间刷新一次来更新网页内容, 这种方法存在着很多缺点, 如用户体验差, 刷新时间间隔的设置缺乏灵活性, 易浪费服务器资源, 推送消息的实时性差等。comet 是一种基于 http 长连接的服务器推技术, 通过 ajax 引擎来进行浏览器请求和服务器响

应的传送。浏览器通过 ajax 发送请求至服务器，服务器保持请求，直到有新的消息到达时才会返回响应到浏览器。浏览器收到返回响应后重新发送请求建立连接。这种方式存在着长时间保持连接浪费资源，连接数目过多时服务器负担大等缺点。Websocket 技术的出现，很好地解决了服务器推送面临的诸多困难。[2] 从多个角度对比 Websocket 协议与 Ajax 协议，如表2.1所示。两者都广泛应用于浏览器与服务器通信，却有本质不同，WebSocket 协议作为 HTML5 新推出的通信协议，适用于实时性要求严格的应用场景。服务器端监管在线的客户端状态，可以代替工业部分 HMI，为工厂可视化应用提供了基于 IT 技术的解决方案。[3]

表 2.1: 协议对比

对比项	Ajax 协议	WebSocket 协议
连接类型	UDP	TCP
生命周期	短连接	长连接
实时性	一般	优
服务器推送	不支持	支持
数据包大小	较大	较小
客户端状态	难以监管	可监管
异步请求	支持	不支持
开发难度	低	中

相较于 HTTP,Websocket 协议让服务器由被动响应, 变为主动推送, 让人们对于消息获取的实时性得到满足, 并且减少了轮询时的资源消耗。而在我们的应用场景中，恰恰需要服务端主动向客户端实时推送从 Web 端转达的 Shell 指令。考虑到如下优势。

- (1) 简单：主程序可以有效管理所有客户端连接，开发者只需专注于业务逻辑程序设计。
- (2) 稳健：为满足工业生产而构建，例如，从 2014 年正式发布以来，全球用户共同测试和优化程序，以提高服务的稳定性。
- (3) 优质：WebSocket 经过严格测试。协议库通过了符合工业标准的压力测试 (Autobahn Testsuite)。

- (4) 高性能：可配置内存大小，底层驱动基于 C 语言开发，速度快，预先为 Linux、Macos、Windows 编译好了一个以 wheel 格式针对不同操作系统和 Python 版本的安装包。[3]

基于以上理由，在整体结构设计中传递命令的链路都是使用 Websocket 协议。事实上因为 Websocket 具有部署简单，开发方便，具备稳定高效的双向数据流，适用于 IIOT 网络远距离传输的应用场景。

## 2.2 服务端

服务端代码用 Kotlin 编写，使用 PostgreSQL 数据库，并且使用 Nginx 服务器，Kotlin 的 Web 程序的开发过程中，可以方便地创建一个 Kotlin 的服务器，其中的后端业务逻辑处理具有速度快、高并发的优点，但在实践中 Kotlin 的服务器对前端静态网页文件的支持效果不好。因而实践中 Kotlin 的 Web 程序开发最好采用前后端分离的技术。页面展示采用一些前端的框架进行开发，将编译后的前端程序部署到 Nginx 服务器中，通过建立具有 RESTful 风格的 Kotlin 程序，使前后端有机地交互联系起来。在 Nginx 服务器中使用反向代理将前端程序连接到后端的 Kotlin 的 Web 程序中。因而 Kotlin 程序的前后端分离有利于提高开发的效率，同时也使程序的结构清晰，增强了程序的健壮性、可扩展性和可维护性。

### 2.2.1 Kotlin 编程语言

Kotlin 是一种在 Java 虚拟机上执行的静态类型编程语言，它也可以被编译成为 JavaScript 源代码。它主要是由俄罗斯圣彼得堡的 JetBrains 开发团队所发展出来的编程语言，虽然与 Java 语法并不兼容，但在 JVM 环境中 Kotlin 被设计成可以和 Java 代码相互运作，并可以重复使用如 Java 集合框架等的现有 Java 引用的函数库。Kotlin 除了编译成 Java 字节码运行，也可以作为脚本语言解释运行，此特性使得 Kotlin 可以以交互模式运行。交互模式是脚本语言具有的特性，解释器可以立即运行用户输入的代码，并反馈运行结果。Kotlin 兼具两重特性，这让编程更具有趣味性。同时借助 JVM 的库，功能上也非常成熟和全面。虽然现阶段不具有很大的知名度，但从使用的角度上来讲，肯定是不劣于 Java 语言的。但这还不是全部，Kotlin 还可以



被编译成 JavaScript, 在 nodejs 或者 V8 引擎上运行, 这样让我们也可以以 nodejs 构建服务端。虽然 Kotlin 具有如上有趣的特性。是一门非常有趣的现代语言。但我们使用该语言的最主要动机是, 借助 Spring Framework 的成熟解决方案实现一个更能满足我们需求的协议。虽然 Websocket 提供了一种良好的通讯方式。但是 WebSocket 协议仅仅只定义了两种类型的消息（文本和二进制），但是它们的内容没有定义。对于嵌入式设备这是足够了，但对于服务端和 Web 端的通讯这显然是不够的。于是我们需要用到 STOMP（简单的面向文本的消息传递协议）作为其子协议, 以便在 WebSocket 上使用它来定义每个消息可以发送哪些类型、格式是什么、每个消息的内容等等。子协议的使用是可选的, 但无论如何, 客户端和服务端都需要就一些定义消息内容的协议达成一致。

### 2.2.2 STOMP 协议

STOMP 协议 (即 Simple (or Streaming) Text Orientated Messaging Protocol, 简单 (流) 文本定向消息协议) 最初是为脚本语言 (如 Ruby、Python 和 Perl) 创建的, 用于连接到企业消息代理, 它被设计用于处理常用消息传递模式的最小子集, 它定义了一种机制可以用于任何可靠的双向流网络协议, 如 TCP 和 WebSocket, 虽然 STOMP 是一个面向文本的协议, 但消息 payload 可以是文本或二进制, 并且可以连续操作。允许 STOMP 客户端与任意 STOMP 消息代理 (Broker) 进行交互如图2.2

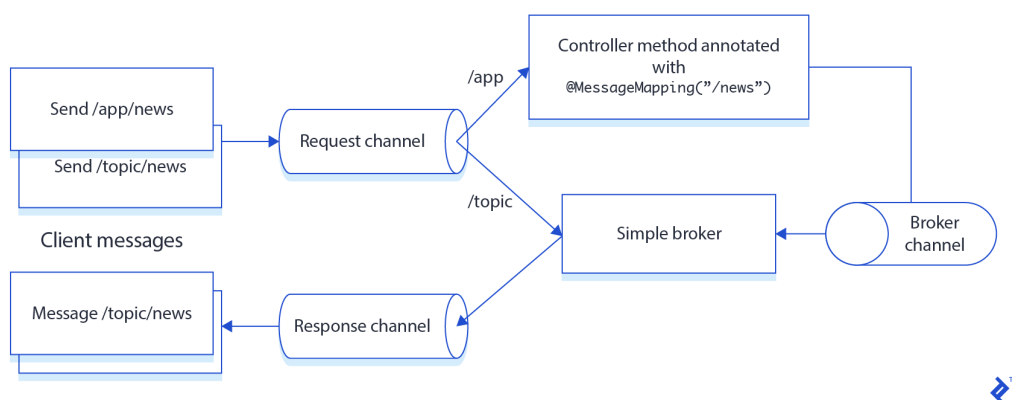


图 2.2: STOMP 的 WebSocket 实现

STOMP 协议由于设计简单,易于开发客户端,因此在多种语言和多种平台上得到广泛地应用。STOMP 是一个非常简单和容易实现的协议,其设计灵感源自于 HTTP 的简单性。尽管 STOMP 协议在服务器端的实现可能有一定的难度,但客户端的实现却很容易。例如,可以使用 Telnet 登录到任何的 STOMP 代理,并与 STOMP 代理进行交互。STOMP 协议与 2012 年 10 月 22 日发布了最新的 STOMP 1.2 规范。要查看 STOMP 1.2 规范,见: <https://stomp.github.io/stomp-specification-1.2.html>。Java 的 Stomp 就支持最新的 STOMP 1.2 规范,是一个非常优秀的开源实现。STOMP 协议与 HTTP 协议很相似,它基于 TCP 协议,使用了以下命令:

**CONNECT** STOMP 客户端通过初始化一个数据流或者 TCP 链接发送 CONNECT 帧到服务端

**CONNECTED** 如果服务端接收了链接意图,它回回复一个 CONNECTED 帧,正常链接后客户端和服务端就可以正常收发信息了。

**SEND** 客户端主动发送消息到服务器

**SUBSCRIBE** 客户端注册给定的目的地,被订阅的目的地收到的任何消息将通过 MESSAGE Frame 发送给 client。

**UNSUBSCRIBE** UNSUBSCRIBE 用来移除一个已经存在订阅,一旦一个订阅被从连接中取消,那么客户端就再也不会收到来自这个订阅的消息。

**BEGIN** BEGIN 用于开启一个事务-transaction。这种情况下的事务适用于发送消息和确认已经收到的消息。在一个事务期间,任何发送和确认的动作都会被当做事务的一个原子操作。

**COMMIT** 用来提交一个事务到处理队列中,帧中的 transaction 头是必须得,用以标示是哪个事务被提交。

**ABORT** ABORT 用于中止正在执行的事务,帧中的 transaction 头是必须得,用以标示是哪个事务被终止。

**ACK** ACK 是用来在 client 和 client-individual 模式下确认已经收到一个订阅消息的操作。在上述模式下任何订阅消息都被认为是没有被处理的，除非客户端通过回复 ACK 确认。

**NACK** NACK 是 ACK 的反向，它告诉服务端客户端没有处理该消息。服务端可以选择性的处理该消息，重新发送到另一个客户端或者丢弃它或者把他放到无效消息队列中记录。

**DISCONNECT** 客户端可以通过 DISCONNECT 帧表示正常断开链接

**MESSAGE** MESSAGE 用于传输从服务端订阅的消息到客户端。

**ERROR** 如果连接过程中出现什么错误，服务端就会发送 ERROR。在这种情况下，服务端发出 ERROR 之后必须马上断开连接。

**RECEIPT** 每当服务端收到来自客户端的需要 receipt 的帧时发送给客户端

STOMP 的客户端和服务端之间的通信是通过“帧”（Frame）实现的，每个帧由多“行”（Line）组成。第一行包含了命令，然后紧跟键值对形式的 Header 内容。第二行必须是空行。第三行开始就是 Body 内容，末尾都以空字符结尾。STOMP 的客户端和服务端之间的通信是通过 MESSAGE 帧、RECEIPT 帧或 ERROR 帧实现的，它们的格式相似。使用 STOMP 作为子协议，可以让 Spring Framework 和 Spring Security 提供比使用原始 WebSockets 更丰富的编程模型，以下是一些好处：

- 无需创建自定义消息传递协议和消息格式。
- STOMP 客户端，包含一个 Spring Framework 中的 Java 客户端。
- 你可以（可选地）使用消息代理（如 RabbitMQ、ActiveMQ 等）来管理订阅和广播消息。
- 应用程序逻辑可以组织在任意数量的 @Controller 实例中，可以基于 STOMP destination header 将消息路由到它们，而不必针对给定连接使用单个 WebSocketHandler 处理原始 WebSocket 消息。

- 你可以使用 Spring Security 来基于 STOMP destination 和消息类型保护消息。

### 2.2.3 Nginx web 服务器



图 2.3: Nginx 标志

Nginx 是异步框架的网页服务器，也可以用作反向代理、负载均衡器和 HTTP 缓存。该软件由伊戈尔·赛索耶夫创建并于 2004 年首次公开发布。Nginx 提供开箱即用的静态文件，使用的内存比 Apache 少得多，每秒可以处理大约四倍于 Apache 的请求。在低并发下 Nginx 的性能与 Apache 相当（有时候还低于），但是在高并发下 Nginx 能保持低资源低消耗高性能。Nginx 的优点还包括：高度模块化的设计，模块编写简单，以及配置文件简洁。在本案例中，Nginx 的反向代理技术是根据服务端数据渲染网页模板的工具，同时保证网络连接的高可靠性。相比于 Apache 它具有如下优点。

1. 支持高并发连接，合理优化配置 nginx+php(fastcgi) 可以承受 30000 以上并发连接数，如图1所示

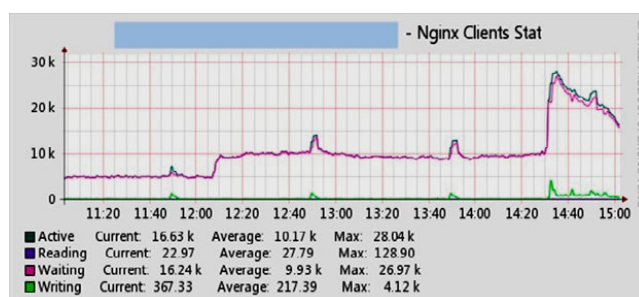


图 2.4: 金山游戏官方网站 2009 年 nginx 集群连接数

2. 内存消耗少，在实际应用环境下，单台服务器 nginx+php5(fastcgi) 处理程序能力已超过 700 次/秒，但 CPU 负载并不高，如图2所示

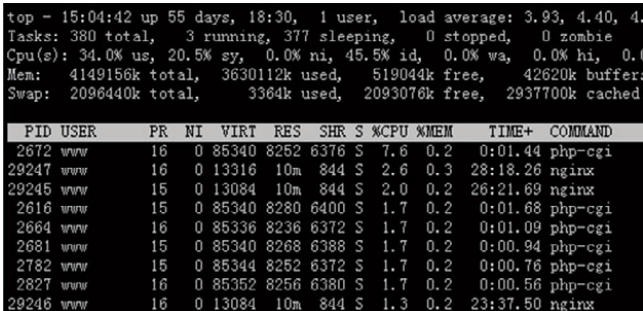


图 2.5: nginx+php 系统负载与 CPU 使用率

- 3. 成本低廉, 通过开源的 nginx 做反向代理实现负载均衡, 节省购买硬件负载均衡交换机的运维成本.
- 4. 支持热部署, 并且运行数月也无需重启, 在不间断服务的前提下对软件进行升级. [4]

2.2.4 PostgreSQL 数据库



图 2.6: PostgreSQL 标志

PostgreSQL 是最先进的、开发源码的（对象）-RDBMS，它以符合标准和可扩展为主要目标。PostgreSQL 或 Postgres 试图采用 ANSI / ISO SQL 标准进行修正。虽然这个 DBMS 不具备的 MySQL 的那样普及程度，但有许多惊人的第三方工具和库，它们设计的目的是使 PostgreSQL 使用起来简单，以致忽略了这个数据库的强大特性。现在可以通过许多操作系统的默认软件包管理器，轻松得到 PostgreSQL 的一个应用程序包。[5] PostgreSQL 经历了长时间的演变。该项目最初开始于在加利福尼亚大学伯克利分校的

Ingres 计划。Postgres 遵守 BSD 许可证发行。作为历史悠久的数据库管理系统 PostgreSQL 内置丰富的数据类型，以及复杂且强大的功能。数据类型包括：

- 任意精度的数值
- 无限制长度文本
- 几何图元
- IP 地址与 IPv6 地址
- 无类域间路由地址块，MAC 地址
- 数组
- JSON 数据
- 枚举类型
- XML 数据

除了支持丰富的类型之外,PostgreSQL 也拥有如下优点。

- 被许多强大的开源的第三方工具所支持，这些第三方工具使设计、管理和使用这个管理系统更加方便。
- 可以像一个高级的 RDBMS 那样，可以利用存储过程以编程的方式对它进行扩展。
- 不只是一个关系型数据库管理系统，也是面向对象的，支持嵌套以及更多。[6]

如此强大丰富的功能，为精通数据库操作的开发人员提供了各种可能，但同时为初学者学习掌握工具，带来了困难。鉴于这个工具的特性，它的背后缺少普及，尽管有非常大量的部署-这可能会很容易影响人们得到支持。同时由于本身的复杂结构，当处理包含简单的大量读取操作的工作，可能表现得没有其他同行效率高，例如 MySQL。但不妨碍，其本身仍然是先进的数据库管理系统。并且在特定使用场景下，例如当数据可靠性和完整性是绝对必要不可妥协的情况下，PostgreSQL 是更好的选择。[6]

## 2.3 Web 端

### 2.3.1 React



图 2.7: React 标志

为了保证前端的美观, 我们选择使用 react 框架。React 是一个用于构建用户界面的 JAVASCRIPT 库。React 主要用于构建 UI, 很多人认为 React 是 MVC 中的 V (视图)。React 起源于 Facebook 的内部项目, 用来架设 Instagram 的网站, 并于 2013 年 5 月开源。React 拥有较高的性能, 代码逻辑非常简单, 越来越多的人已开始关注和使用它。React 具有如下特点

- (1) 声明式设计 — React 采用声明范式, 可以轻松描述应用。
- (2) 高效 — React 通过对 DOM 的模拟, 最大限度地减少与 DOM 的交互。
- (3) 灵活 — React 可以与已知的库或框架很好地配合。
- (4) JSX — JSX 是 JavaScript 语法的扩展。React 开发不一定使用 JSX, 但我们建议使用它。
- (5) 组件 — 通过 React 构建组件, 使得代码更加容易得到复用, 能够很好的应用在大项目的开发中。
- (6) 单向响应的数据流 — React 实现了单向响应的数据流, 从而减少了重复代码, 这也是它为什么比传统数据绑定更简单。

React 使创建交互式 UI 变得轻而易举。为你应用的每一个状态设计简洁的视图, 当数据变动时 React 能高效更新并渲染合适的组件。以声明式编

写 UI，可以让你的代码更加可靠，且方便调试。构建管理自身状态的封装组件，然后对其组合以构成复杂的 UI。由于组件逻辑使用 JavaScript 编写而非模板，因此你可以轻松地在应用中传递数据，并保持状态与 DOM 分离。无论你现在使用什么技术栈，在无需重写现有代码的前提下，通过引入 React 来开发新功能。React 还可以使用 Node 进行服务器渲染，或使用 React Native 开发原生移动应用。

## 2.4 项目管理方式

尽管我们这是个小项目，但我们仍然坚持使用现代化的项目管理方式。因为这样我们才能学到更多东西，并且让整个项目周期更加完整。如果这是一个合作项目，那么它就应该以一种更适宜合作的方式去开展。我们基于 GitLab 的 Pipeline 来实现代码的持续集成和持续交付部署（CI/CD），以及使用 Terraform 来实现基础设施的自动化创建，实现基础设施即代码（IaC）。在软件开发过程中，一般会有不同的部署环境，例如开发环境、集成环境、预览环境和产品环境，通过使用一套 Terraform 代码实现不同环境的基础设施创建。设计实现基于不同环境的微服务自动化持续集成和持续部署方案，微服务在部署时从基础设施中获取相应的资源信息，实现微服务的 CI/CD 完全自动化，轻松管理和控制服务代码的改动以及基础设施的变动方便地应用到不同的环境当中，极大地提高软件开发和服务部署的效率。[7]

### 2.4.1 CI/CD 自动化集成部署

GitLab 支持 DevOps\*生命周期的从计划到监控所有阶段如图2.4.1。GitLab 集成的 CI/CD 功能让企业快速地发布代码，帮助团队自动化发布应用，缩短发布的生命周期。通过在 Pipeline 中集成 CD，可以自动发布到多个环境中例如 staging 和 production 环境。

---

\*DevOps 目前并没有权威的定义，网易认为，DevOps 强调的是高效组织团队之间如何通过自动化的工具协作和沟通来完成软件的生命周期管理，从而更快、更频繁地交付更稳定的软件。



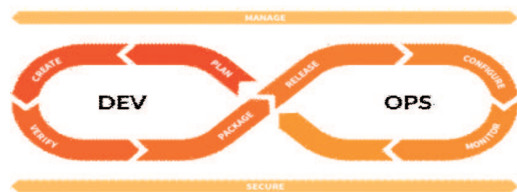


图 2.8: DevOps 生命周期

GitLab 提供持续集成服务。通过添加一个 `.gitlab-ci.yml` 文件会告诉 GitLab Runner 做什么，默认情况下它运行一个 pipeline，这个 pipeline 分为三个阶段：build、test、deploy。并不需要用到所有的阶段，没有 job 的阶段会被忽略。如果一切运行正常（没有非零的返回值），将得到与 commit 关联的漂亮的绿色标记。这使得在查看代码之前，很容易就能看出是否有一个提交导致了测试失败。大多数项目使用 GitLab CI 服务来运行测试套件，这样如果开发人员发现问题就会及时得到反馈。简而言之，CI 所需要的步骤可以归结为：(1) 添加 `.gitlab-ci.yml` 到项目的根目录；(2) 配置一个 Runner。从此刻开始，在每一次 push 到 Git 仓库的过程中，Runner 会自动开启 pipeline，pipeline 将显示在项目的 Pipeline 页面中。图2.4.1所示是一个典型的 GitLab Pipeline。[7]

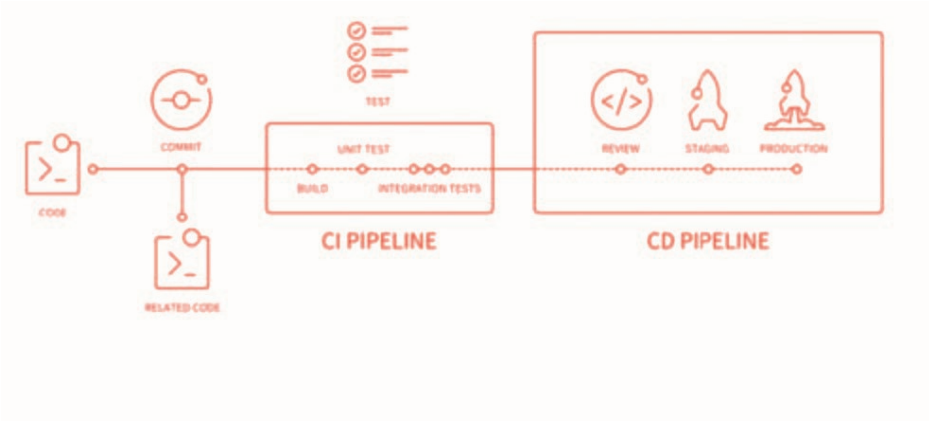


图 2.9: GitLab Pipeline

## 2.5 部署方式

### 2.5.1 Docker

在测试的时候, 我们发现源程序并不方便部署。于是采用 Docker 来进行部署, 这样无论用户使用还是测试都很方便。并且制作成镜像进行发布。



图 2.10: Docker 标志

Docker 是一个开放源代码软件, 是一个开放平台, 用于开发应用、交付 (shipping) 应用、运行应用。Docker 允许用户将基础设施 (Infrastructure) 中的应用单独分割出来, 形成更小的颗粒 (容器), 从而提高交付软件的速度。Docker 容器与虚拟机类似, 但二者在原理上不同。容器是将操作系统层虚拟化, 虚拟机则是虚拟化硬件, 因此容器更具有便携性、高效地利用服务器。容器更多的用于表示软件的一个标准化单元。由于容器的标准化, 因此它可以无视基础设施 (Infrastructure) 的差异, 部署到任何一个地方。另外, Docker 也为容器提供更强的业界的隔离兼容。Docker 是有能力打包应用程序及其虚拟容器, 可以在任何 Linux 服务器上执行的依赖性工具, 这有助于实现灵活性和便携性, 应用程序在任何地方都可以执行, 无论是公用云端服务器、私有云端服务器、单机等。由于服务端分 Web 服务器以及 PostgreSQL 数据库两部分, 所以我们会使用 Docker Compose, Compose 是用于定义和运行多个容器 Docker 应用程序的工具。通过 Compose, 你可以使用 YAML 文件来配置应用程序需要的所有服务, 然后通过使用一个命令, 就可以创建并启动所有服务。对应的命令为 `docker-compose`。例如我们的 `docker-compose.yml`

```
1  version: "3"
3  services:
```

```
5  ratcat-web:
    image: imbytecat/ratcat-react:latest
    container_name: ratcat-web
7  ports:
    - 20080:80
9  depends_on:
    - ratcat-server
11
13  ratcat-db:
    image: postgres:latest
    container_name: ratcat-db
15  restart: always
    environment:
17  POSTGRES_PASSWORD: postgres
    POSTGRES_DB: ratcat_db
19  ports:
    - 25432:5432
21  volumes:
    - ratcat_data:/var/lib/postgresql/data
23
25  volumes:
    ratcat_data:
    external: false
```

### 3 具体实现

#### 3.1 Client

STOMP 客户端设置, 我们需要设置, 心跳包的频率, 以及允许的连接延时. 相关代码如下

```
1  const stompClient = new Client({
2      brokerURL: "",
3      reconnectDelay: 5000,
4      heartbeatIncoming: 4000,
5      heartbeatOutgoing: 4000,
6  });
```

同时我们需要定义传送数据的格式, 初步拟定以下条目3.1

表 3.1: Clent 数据包

关键字	含义
nodeId	节点 ID
hostname	主机名
os	操作系统
cpu	CPU 型号
cpuLoad	CPU 负载
gpus	GPU 型号
memoryUsed	已使用内存
memoryTotal	总内存
ipv4	IPv4 地址
ipv6	IPv6 地址
rx	接收信息的总量
tx	传送信息的总量
rxSec	接收速率
txSec	发送速率
uptime	运行时间

传送数据包的格式定义代码

```
const systemInfo = {  
  2   nodeId,  
      hostname: osInfo.hostname,  
  4   os: `${osInfo.distro} ${osInfo.release} ${osInfo.arch}`,  
      cpu: `${cpuInfo.manufacturer} ${cpuInfo.brand} (${cpuInfo.  
        cores}) @ ${cpuInfo.speed}GHz`,  
  6   cpuLoad: `${Math.round(currentLoadInfo.currentLoad)}%`,  
      gpus: gpuList.toString(),  
  8   memoryUsed: `${Math.round(usedMem)}%`,  
      memoryTotal: `${Math.round(memInfo.total)}%`,  
 10   ipv4,  
      ipv6,  
 12   rx: networkStatsInfo[0].rx_bytes,  
      tx: networkStatsInfo[0].tx_bytes,  
 14   rxSec: `${Math.round(networkStatsInfo[0].rx_sec)}%`,  
      txSec: `${Math.round(networkStatsInfo[0].tx_sec)}%`,  
 16   uptime: timeInfo.uptime,  
};
```

将以上的数据包定时进行传递, 时间间隔为 5000 微秒, 便于服务端与外部端更新数据

```
1   const stompSuccessCallback = (frame: any) => {  
      console.log(`STOMP: Connection successful\n${frame}\n`);  
  3   setInterval(async () => {  
      stompClient.publish({  
  5       destination: callDestination,  
          body: JSON.stringify(await getSystemInfo()),  
  7       });  
      }, 5000);  
  9   }
```

以上传送的是更新信息, 定时传送约定好的信息。客户端订阅服务端的主要目的是传送 Web 发给服务端的命令, 订阅即使用 STOMP 的 subscribe

的关键字2.2.2, 用 JavaScript 封装的 stompClient.subscribe 函数即可完成

```
1  stompClient.subscribe(simpleBroker, async (payload: any) => {  
    ...  
3  }
```

然后是接受到各个关键词的具体实现, 就写在上面函数的函数体里面。

```
1  case "exec": {  
    const cmd = spawn(task.taskCommand, [], { shell: true });  
3  console.log(  
    `[TASK ${task.taskId}] child process pid ${cmd.pid} started`  
5  );
```

比如这行代码, 就是用来执行传送给 client 端的命令, 并且返回 console.log 的结果。同时还有强制停止命令的关键字

```
1  case "exec-stop": {  
    try {  
3      kill(childProcessTasks.get(task.taskCommand));  
    } catch (error) {  
5      console.log("kill 子进程发生错误");  
    }  
7    break;  
  }
```

因为 client 端的字符编码不同, 所以我们还需要解决支付编码不统一的问题, 这里我们首先用 GBK 尝试解码, 与 utf8 解码进行对比, 保证输出正确结果。

```
    cmd.stdout.on("data", (data) => {  
2      // TODO 优雅地解决 WIndows 上的编码问题  
      const gbkData = iconv.decode(data, "gbk");  
4      stompClient.publish({  
        destination: taskDestination,  
6      body: JSON.stringify({
```

```
8         ...task,  
        ...{ taskResult: gbkData === data ? `${data}` : `${  
            gbkData}` },  
        }),  
10    });  
    });
```

在此例举 stdout 的结果 stderr 类似

## 3.2 Server

如 docker-compose2.5.1所示, 服务端被拆分成两个部分, 放置于 Docker 的两个 Container, 分别是业务部分与数据库。这样分离运行有很多好处, 可以保证数据不易被损坏。维护起来也更加方便, 数据部分与业务部分可以分开维护升级。在业务部分, 由于我们利用框架通过注解<sup>†</sup>完成了很多工作, 所以并不是所有细节都可以在我们的源代码中看到。但是是能够实现功能的。

### 3.2.1 业务部分

在 model 层我们定义表结构, 构造实体, 便于使用 orm 框架, 对数据库进行初始化。以及之后对于数据库的操作。

```
1  @Entity  
   data class Call(  
3  @Id  
   @GeneratedValue  
5  val id: Long? = null,  
   val nodeId: String = "",  
7  var time: Date = Date(),  
   val hostname: String = "",  
9  val os: String = "",  
   val cpu: String = "",  
11 val cpuLoad: Long = 0, // 百分比的大数
```

<sup>†</sup>也就是所有代码前 @ 的部分, 通过这种方式可以自动完成模板代码, 实现功能。

```
13 //    val gpus: List<String> = listOf(), // GPU 列表
14 val gpus: String = "", // GPU 列表
15 val memoryUsed: Long = 0, // bytes
16 val memoryTotal: Long = 0, // bytes
17 val internalIpv4: String = "",
18 val internalIpv6: String = "",
19 val publicIpv4: String = "",
20 val publicIpv6: String = "",
21 val rx: Long = 0, // bytes
22 val tx: Long = 0, // bytes
23 val rxSec: Long = 0, // bytes
24 val txSec: Long = 0, // bytes
25 val uptime: Long = 0, // second
    )
```

通过 call 进行举例, 其余部分参见源码, 此处略去。

在 confid 文件中我们定义了建立 WebSocket 的常连接

```
1 class WebsocketConfig : WebSocketMessageBrokerConfigurer {
2     /**
3      * 配置 WebSocket 进入点, 及开启使用 SockJS, 这些配置主要用配置连接端点, 用于 WebSocket 连接
4      *
5      * @param registry STOMP 端点
6      */
7     override fun registerStompEndpoints(registry:
8         StompEndpointRegistry) {
9         registry.addEndpoint("/myws")
10            .withSockJS() // sockjs 可以在不支持 ws 的浏览器中降级为轮训
11     }
12
13     /**
14      * 配置消息代理选项
15      *
16      * @param registry 消息代理注册配置
17      */
18 }
```



```

17     override fun configureMessageBroker(registry:
        MessageBrokerRegistry) {
        // 设置一个或者多个代理前缀，在 Controller 类中的方法里面
        // 发生的消息，会首先转发到代理从而发送到对应广播或者队列
        // 中。
19         registry.enableSimpleBroker("/topic")
        // 配置客户端发送请求消息的一个或多个前缀，该前缀会筛选消
        // 息目标转发到 Controller 类中注解对应的方法里
21         registry.setApplicationDestinationPrefixes("/app")
        }
23     }

```

在 controller 层接受每次心跳包信息 Call，同时还有 Client 和 task 的信息，现不列出。

```

1     @Controller
    class CallController {
3         @Autowired
        private val callService: CallService? = null
5
        // 接受客户端发来的心跳信息
7         @PostMapping("/call/{machineId}")
        fun sendTopicMessage(@DestinationVariable machineId: String,
            @Payload call: Call) {
9             callService?.saveCall(call)
        }
11    }

```

在 repository 我们定义，在数据库中读取信息的方式，对于每一个节点我们读取数据库中的最新信息。

```

1     @Repository
    interface CallRepository : JpaRepository<Call?, Int?> {
3         @Query(
            value = ""select *

```

```
5      from (select *, row_number() over (partition by node_id
      order by time desc) rn from call) as t
      where rn = 1""",
7      nativeQuery = true
      )
9      fun findLatestGroupByNodeId(): List<Call>

11     @Query(
      value = ""select *
13     from (select * from call where node_id = :nodeId order by
      time desc limit 24) as base
      order by time""", nativeQuery = true
15     )
      fun findSomeLatestByNodeIdOrderByTimeDesc(nodeId: String):
          List<Call>
17 }
```

在 service 层我们启动服务以列表的形式返回信息。

```
1 @Service
      class CallService {
3         @Autowired
          private val callRepository: CallRepository? = null
5
          fun saveCall(call: Call) {
7              callRepository?.save(call)
          }
9
          fun getClientCalls(): List<Call>? {
11              return callRepository?.findLatestGroupByNodeId()
          }
13
          fun getSomeLatestCalls(nodeId: String): List<Call>? {
15              return callRepository?.
                  findSomeLatestByNodeIdOrderByTimeDesc(nodeId)
          }
17 }
```

---

最后通过根目录下的 RatcatApplication 启动服务

```
1 @SpringBootApplication
   class RatcatApplication
3
5 fun main(args: Array<String>) {
   runApplication<RatcatApplication>(*args)
}
```

### 3.2.2 数据库

我们采用 PostgreSQL 数据库<sup>2.2.4</sup>进行数据存储，没有采用复杂的表结构仅仅只是以数据库的形式把数据存下来。并且由于使用 ORM 框架<sup>‡</sup>进行数据库的创建。由于表结构与字段是自动创建的，并且也是通过函数的方式进行调用而非 SQL 语句，数据库结构比较简单。

在 ratcat 数据库中，最主要的 call 表用来存储各个节点的所有信息，并且根据心跳包的数据进行更新。表结构如下。

---

<sup>‡</sup>对象关系映射（Object Relational Mapping，简称 ORM）广义上，ORM 指的是面向对象的对象模型和关系型数据库的数据结构之间的相互转换。狭义上，ORM 可以被认为是，基于关系型数据库的数据存储，实现一个虚拟的面向对象的数据访问接口。理想情况下，基于这样一个面向对象的接口，持久化一个 OO 对象应该不需要要了解任何关系型数据库存储数据的实现细节。

表 3.2: call 数据字典

属性名	存储代码	类型	长度	是否为空
节点 ID	id	bigint		not null
CPU 型号	cpu	character	255	
CPU 负载	cpu_load	bigint		not null
GPU 型号	gpus	character	255	
主机名	hostname	character	255	
内部 IPv4 地址	internal_ipv4	character	255	
内部 IPv6 地址	internal_ipv6	character	255	
总内存	memory_total	bigint		not null
已使用内存	memory_used	bigint		not null
节点 ID	node_id	character	255	
操作系统	os	character	255	
外部 IPv4 地址	public_ipv4	character	255	
外部 IPv6 地址	public_ipv6	character	255	
接收信息的总量	rx	bigint		not null
接收速率	rx_sec	bigint		not null
最近通信时间	time	timestamp		
传送信息的总量	tx	bigint		not null
发送速率	tx_sec	bigint		not null
运行时间	uptmes	bigint		not null

3.3 Web

由于使用了前后端分离的设计思想，Nginx 前端服务器作为 Web 服务器独立于 Server 运行，同样也是在一个单独的 Container 里面，详细部署信息见2.5.1。其主要完成对于从 Server 接收到 json 的信息后网页模板的渲染。主要工作内容为用 TypeScript 借助 React 框架展现一个漂亮的网页界面。由于前端代码看起来比较繁杂，现在展现部分。

绘制 Home 页面节点卡片的 NodeCard.tsx

```
const NodeCard = (props: any) => {
```

```
2    const editName = (nodeId: string, nodeName: string) => {
      const myHeaders = new Headers();
4      myHeaders.append("Content-Type", "application/x-www-form-
        urlencoded");

6      const urlencoded = new URLSearchParams();
      urlencoded.append("name", nodeName);

8      const requestOptions: RequestInit = {
10         method: "POST",
          headers: myHeaders,
12         body: urlencoded,
          redirect: "follow",
14       };

16       fetch("/api/clients/" + nodeId, requestOptions)
        .then((response) => response.text())
18       .then((result) => console.log(result))
        .catch((error) => console.log("error", error));
20     };
    return (...)
```

我们设置了两个主要页面 Home 与 Detail, Home 用来展现所有设备的卡片如4通过卡片可以跳转到 Detail 页面查看特定设备的详细信息并且控制设备如4

Home 页面通过”/api/client/”路由获得信息然后返回页面。

```
1    const Home = () => {
      // Effect Hook, 组件渲染后执行
3      React.useEffect(() => {
        // 为了立即执行
5        getCalls() && setInterval(getCalls, 3000);
      }, []); // [小技巧] 注意这里加了一个空参数以只执行一次
7      const [info, setInfo] = useState(Object);

9      const getCalls = () => {
        fetch("/api/clients/")
```

```
11     .then((response) => response.json())
    .then((result) => setInfo(result))
13     .catch((error) => console.log("error: ", error));
    return true;
15 };

17 return (...)
```

Detail 源码过长，我们摘录了关键部分，以下代码是折线图的绘制。通过以下代码，就可以直观的表现设备的负载情况，并且数据随时间进行更新。

```
1  const convertCallsToData = (calls: any) => {
    const dataTemp: { time: string; type: string; usage: any }[]
      = [];
3  calls.forEach((element: any) => {
    dataTemp.push({
5      time: moment(element.time).format("HH:mm:ss"),
      type: "cpu",
7      usage: element.cpuLoad,
    });
    dataTemp.push({
9      time: moment(element.time).format("HH:mm:ss"),
      type: "memory",
11     usage: Math.round((element.memoryUsed / element.
        memoryTotal) * 100),
13     });
    });
15     setData(dataTemp);
    });
};
```

配置 STOMP 客户端，实现连接与订阅功能。

```
function stompConnect() {
2  // 设置 SOCKET
    var socket = new SockJS(wsEndpoint);
```

```
4 // 配置 STOMP 客户端
    stompClient = Stomp.over(socket);
6 // STOMP 客户端连接
    stompClient.connect({}, () => {
8         console.log("stomp连接成功");
        stompSubscribe();
10    });
}

12 function stompSubscribe() {
14     stompClient.subscribe(simpleBroker, (resp) => {
        const resObj = JSON.parse(resp.body);
16         const newObj = createTableData(
            `[${resObj.taskId}] ${resObj.taskType}(${resObj.
                taskCommand})${
18             resObj.taskResult ? "=>" + resObj.taskResult : ""
            }`
20         );
        setTableData((tableData) => [...tableData, newObj]);
22    });
}
```

所有输入的命令都是通过 STOMP 进行发送给被控设备进行执行，命令类型通过 `taskType` 进行传递，具体命令通过 `taskCommand` 进行传递。

```
1 function stompSendMsg(taskType: string, taskCommand: string) {
    const msgObj = {
3        nodeId,
        taskId: guid(),
5        taskType,
        taskCommand,
7    };
    stompClient.send(taskDestination, {}, JSON.stringify(msgObj)
        );
9 }
```

# 4 测试使用

我们用 GNS3 模拟网络环境进行测试，用 docker compose 把服务端部署在 Debian 服务器上，然后在 windows7 通过浏览器进行访问。

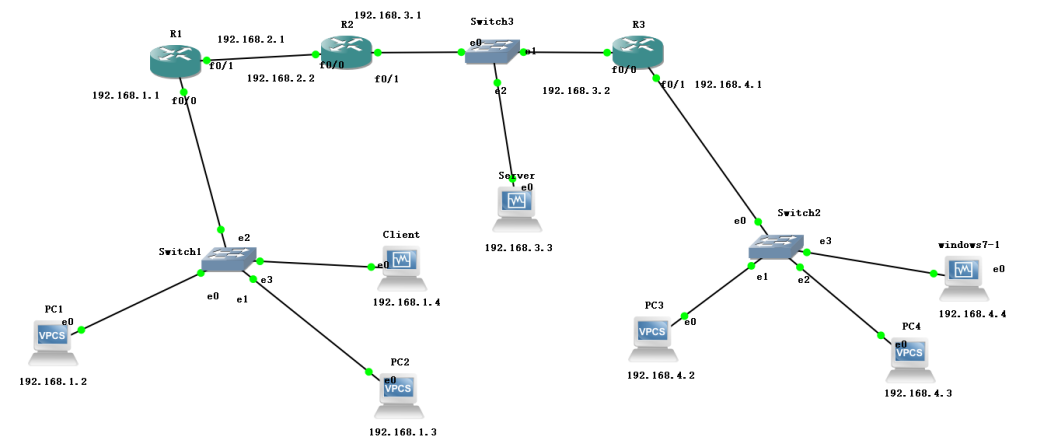


图 4.1: GNS3 测试环境

测试机状态如下

表 4.1: 测试机状态			
	被管设备	服务器	客户端
操作系统	Debian	Debian	Windows7
IP 地址	192.168.1.4	192.168.3.3	192.168.4.4
系统类型	Virtualbox 模拟 x64 的处理器		

主页4显示被管设备的状态信息，通过卡片的方式进行排布。



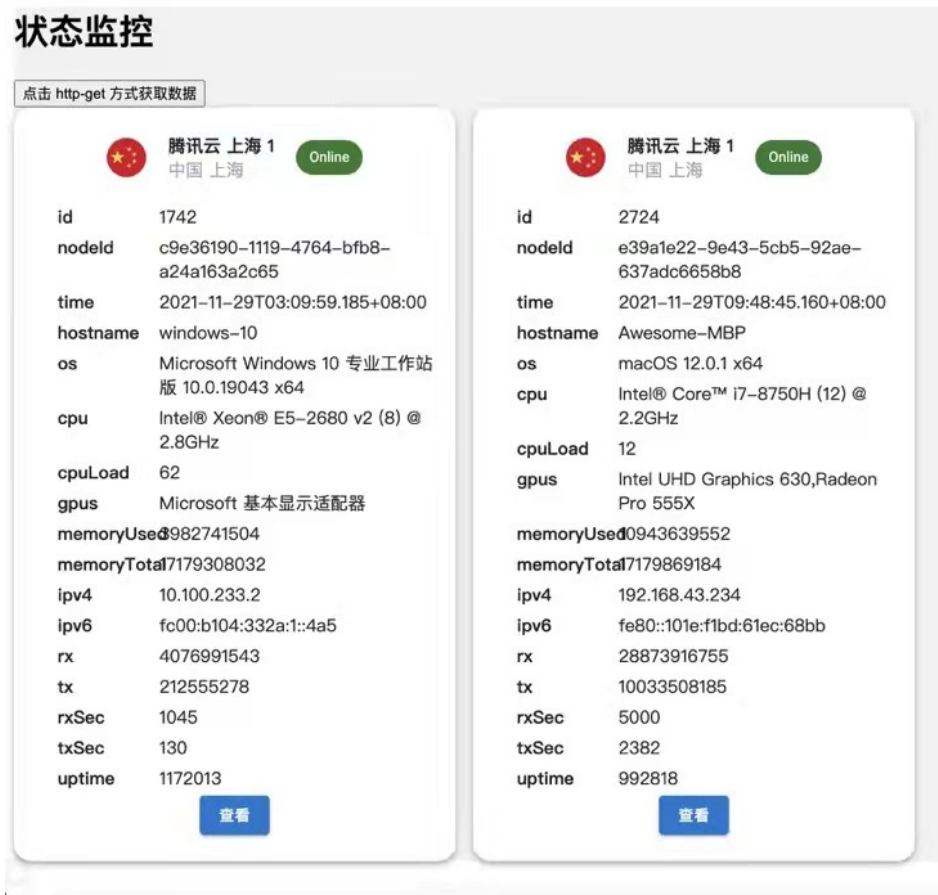



图 4.2: 主页显示

卡片内的详细内容和 Client 端获取的实时更新数据3.1一一对应。



device

中国 上海

Online

节点ID	50c3107e195e4e139c360ba180891d98
更新时间	2021/12/13 14:19:21
主机名	rdCA0
系统	Arch Linux unknown x64
CPU	Common KVM processor (10) @ 2.5GHz
CPU占用	4%
GPU	QXL paravirtual graphic card
已用内存	5.27 GiB
总内存	15.63 GiB
内网IPv4	10.100.70.102
内网IPv6	fc00:b104:332a:1:f8b1:f3ff:fe1
公网IPv4	108.61.176.68
公网IPv6	
下载总量	3.36 GiB
上传总量	0.27 GiB
下载速率	0.01 MiB/s
上传速率	0.00 MiB/s
运行时长	7.10 天

查看

图 4.3: 卡片详细内容

卡片所展现内容清晰直观, 尽可能的显示被管主机的所有信息, 最上方的标题栏, 为设备的别名, 可由用户自主编辑。编辑后的结果, 储存在服务端的数据库中, 可以实现同步更改。点击下方蓝色的查看按钮, 可以跳转到命令页面4

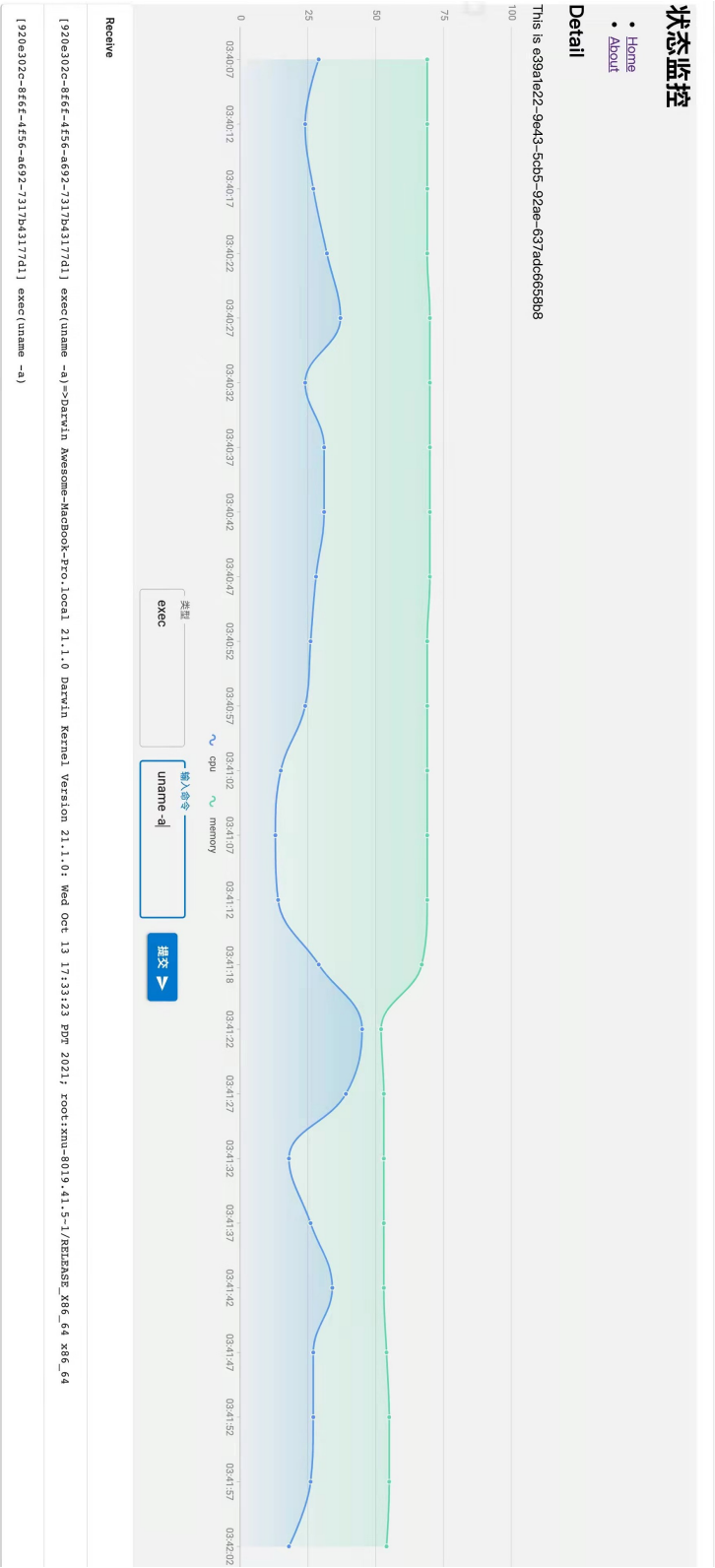


图 4.4: 命令页面

在迷你页面中,我们以曲线图的形式,显示机器负载。绿线是内存负载,蓝线是 CPU 负载。状态下方的两个文本框,可以输入命令。下方的只读文本是回显命令。命令会被终端机进行执行,并且回显状态。这样通过网页就可以实现对于远程设备的管理。

## 5 展望

由于这是我们初次用相关技术解决该领域问题,我们要把主要的精力投入到学习新的技术上面。而新学的技术,只能保证会永远不能保证精通。这对于程序工艺有很多影响,没有经过优化的程序,它的运行效率实在不能说很高。我相信如果摆脱 Nodejs 客户端运行环境的限制,还可以把运行效率更上一层楼。并且更适合嵌入式设备使用。对于主机来说,我们可以实现很方便的部署,但运行负载来说相对不小。

还有很多有用的功能,没有被实现。就比如互动式 Shell,这需要对 Linux 的 Shell 有比较深入的了解。并且我们并没有实现对于 Windows 系统的远程操作方式。这受制于上游的包并没有针对 Windows 的功能。但状态监控是可以针对全平台的。

同时多用户登录,以及用户权限功能有待完善,我们希望建立如数据库管理系统一样的完善的角色系统。这样才方便,在真实的生产环境中分配工作任务,划定管理权限。

永远有更好的程序,受时间限制,事务繁杂,程序只能到此提交。但学习到的技术,会支持我们完成更好的事。

## 插图

2.1	Websocket 协议工作流程 . . . . .	2
2.2	STOMP 的 WebSocket 实现 . . . . .	5
2.3	Nginx 标志 . . . . .	8
2.4	金山游戏官方网站 2009 年 nginx 集群连接数 . . . . .	8
2.5	nginx+php 系统负载与 CPU 使用率 . . . . .	9
2.6	PostgreSQL 标志 . . . . .	9
2.7	React 标志 . . . . .	11
2.8	DevOps 生命周期 . . . . .	13
2.9	GitLab Pipeline . . . . .	13
2.10	Docker 标志 . . . . .	14
4.1	GNS3 测试环境 . . . . .	28
4.2	主页显示 . . . . .	29
4.3	卡片详细内容 . . . . .	30
4.4	命令页面 . . . . .	31

表格

2.1	协议对比 . . . . .	3
3.1	Clent 数据包 . . . . .	16
3.2	call 数据字典 . . . . .	24
4.1	测试机状态 . . . . .	28

## 参考文献

- [1] 钱志鸿; 王义君;. 物联网技术与应用研究. 电子学报, 40, 2012.
- [2] 齐华; 李佳; 刘军;. 基于 websocket 的消息实时推送设计与实现. 微处理机, 37, 2016.
- [3] 李垚; 张宇; 张惠樑;. websocket 协议在 iiot 领域的应用优势. 自动化应用, 2021.
- [4] 凌质亿; 刘哲星; 曹蕾;. 高并发环境下 apache 与 nginx 的 i/o 性能比较. 计算机系统应用, 22, 2013.
- [5] 唐敏; 宋杰;. 嵌入式数据库 sqlite 的原理与应用. 电脑知识与技术, 2008.
- [6] 董纪英; 燕志伟; 梁正玉;. sqlite、mysql、postgresql 关系型数据库管理系统比较. 电脑编程技巧与维护, 2014.
- [7] 刘万里;. 多环境下的 ci/cd 自动化集成部署设计. 现代计算机 (专业版), 2019.