****

****

**基于分布式Web的教学仿真交互式演示平台**

专 业： 物联网工程 学 号：6106213001

学生姓名： 周磊 指导教师： 虞贵财

摘要

标题：宋体，四号，两端对齐，1.35倍行距

内容：中文宋体，外文字符Times New Roman，小四，两端对齐，1.35倍行距

关键词：“关键词”三字加粗，关键词用“；”分隔

宽禁带III－Ⅴ族氮化物半导体材料在短波长高亮度发光器件、短波长激光器、光探测器以及高频和大功率电子器件等方面有着广泛的应用前景。自1994年日本日亚化学工业公司率先在国际上突破了GaN基蓝光LED外延材料生长技术以来，美、日等国十余家公司相继报导掌握了这项关键技术，并分别实现了批量或小批量生产GaN基LED。尽管如此，这项高技术仍处于高度保密状态，材料生长的关键思想及核心技术仍未公开，还无法从参考文献及专利公报中获取最重要的材料生长信息。本论文就是在这种情况下立题的，旨在研究GaN基材料生长中的物理及化学问题，为生长可商品化的高亮度GaN基LED外延材料提供科学依据。

本文在自制常压MOCVD和英国进口MOCVD系统上对III－Ⅴ族氮化物的生长机理进行了研究，对材料的性能进行了表征。通过设计并优化外延片多层结构，生长的蓝光LED外延片质量达到了目前国际上商品化的中高档水平。并获得了如下有创新和有意义的研究结果：

1．首次提出了采用偏离化学计量比的缓冲层在大晶格失配的衬底上生长单晶膜的思想，并在GaN外延生长上得以实现。采用这种缓冲层，显著改善了GaN外延膜的结晶性能，使GaN基蓝光LED器件整体性能大幅度提高，大大降低了GaN基蓝光LED的反向漏电流，降低了正向工作电压，提高了光输出功率。

…………

本文得到了国家863计划、国家自然科学基金以及教育部发光材料与器件工程研究中心项目的资助。

**关键词：**氮化物； MOCVD； LED；卢瑟福背散射沟道；光致发光；光透射谱

**Study on MOCVD growth and properties of III-Ⅴ**

页眉：外文Times New Roman，五号，居中

Times New Roman，小二号，居中

**nitrides and high brightness blue LED wafers**

Abstract

GaN based Ⅲ-Ⅴ nitrides have potential applications on high brightness LEDs, short wavelength lasers, ultraviolet detectors, high temperature and high power electronic devices. Study on physics issues and technologies of nitrides open a new area of 3th generation semiconductor.

标题：Times New Roman，四号，两端对齐，1.35倍行距

内容：Times New Roman，小四，两端对齐，1.35倍行距

关键词：“Keyword”三字加粗，关键词用“；”分隔

More than ten companies in America and Japan reported to have developed the nitrides growth technology since Nichia company in Japan first realized the commercialization of GaN based blue LED in 1994.

In this thesis,GaN and its ternary were grown by a home-made atmosphere pressure metalorganic chemical vapor deposition (MOCVD) and Thomas Swan 6×2” MOCVD systems. High bright blue LED wafers were obtained by optimizing the nitrides growth technology and wafer structure. Some encouraging results are following as:

1. We present the idea of using a buffer layer of deviation from stoichiometry for materials growth on large lattice mismatch substrates. This idea was realized in nitrides growth in this thesis. The epilayer crystalline quality was improved and the dislocation density was decreased by using GaN low and high temperature buffer layers of deviation from stoichiometry. The RBS/channeling spectra exhibited that the minimum yield χmin of GaN layers was just only 1.5%. The leak electric current of GaN based LED was obviously decreased and lower than 1μA at 5 volt reverse voltage by using this new buffer technology.

…………

This work was supported by 863 program in China.

**Keyword:** Nitrides；MOCVD；LED；Photoluminescence；RBS/channeling；Optical absorption

目录

[第一章 功能架构 与 技术架构 5](#_Toc482566399)

[1.1 simulay 系统简介 5](#_Toc482566400)

[1.2 simulay整体功能架构 5](#_Toc482566401)

[1.3 simulay技术架构 7](#_Toc482566402)

[1.3.1在线仿真系统 7](#_Toc482566403)

[1.3.2数据API平台 8](#_Toc482566404)

[1.3.3容器管理系统 8](#_Toc482566405)

[第二章 容器化与微服务 8](#_Toc482566406)

[2.1概念 8](#_Toc482566407)

[2.2容器化使用的是目前的主流的docker技术 9](#_Toc482566408)

[2.3安装与镜像制作 10](#_Toc482566409)

[2.4 DockerAPI 10](#_Toc482566410)

[2.5微服务架构 11](#_Toc482566411)

[第三章 WebTerminal设计与实践 12](#_Toc482566412)

[3.1 WebTerminal简介 12](#_Toc482566413)

[3.2 TTY 与 SSH 12](#_Toc482566414)

[3.3 WebTerminal原理框图 13](#_Toc482566415)

[3.4 WebTerminal工作流程 14](#_Toc482566416)

[第四章 前端架构设计 15](#_Toc482566417)

[4.1架构简介 15](#_Toc482566418)

[4.2前端的工程 15](#_Toc482566419)

[4.2.1模块化 15](#_Toc482566420)

[4.2.2组件化 15](#_Toc482566421)

[4.2.3规范化 15](#_Toc482566422)

[4.3自动化构建 16](#_Toc482566423)

[4.4自动化部署 16](#_Toc482566424)

[4.4.1自动化shell脚本 16](#_Toc482566425)

[4.4.2服务端实现 17](#_Toc482566426)

[第五章 后台架构设计 19](#_Toc482566427)

[5.1 Nignx (负载均衡,动静分离) 19](#_Toc482566428)

[5.1.1负载均衡 20](#_Toc482566429)

[5.1.2动静分离 20](#_Toc482566430)

[5.2基于nodeJS express 的后台管理系统 20](#_Toc482566431)

[5.3 websocket 通信 21](#_Toc482566432)

[第六章 存储系统NoSQL在simulay 中的使用 23](#_Toc482566433)

[6.1 web后台 MongoDB 23](#_Toc482566434)

[6.2代码存储 couchDB 24](#_Toc482566435)

[6.2.1 API操作接口 24](#_Toc482566436)

[6.2.2 示例代码 25](#_Toc482566437)

[第七章 API 设计 26](#_Toc482566438)

[7.1 RESTful API 设计简介 26](#_Toc482566439)

[7.2数据读取API 26](#_Toc482566440)

[7.2.1 Get runner 27](#_Toc482566441)

[7.2.2 Example response data 27](#_Toc482566442)

[7.3仿真runnerAPI 28](#_Toc482566443)

[7.3.1 Run code 28](#_Toc482566444)

[7.3.2 Simple example 28](#_Toc482566445)

[第八章 系统优化以及缓存 29](#_Toc482566446)

[8.1仿真结果缓存 29](#_Toc482566447)

[8.2 SVG 矢量图在系统的运用 30](#_Toc482566448)

[8.3 redis缓存系统 31](#_Toc482566449)

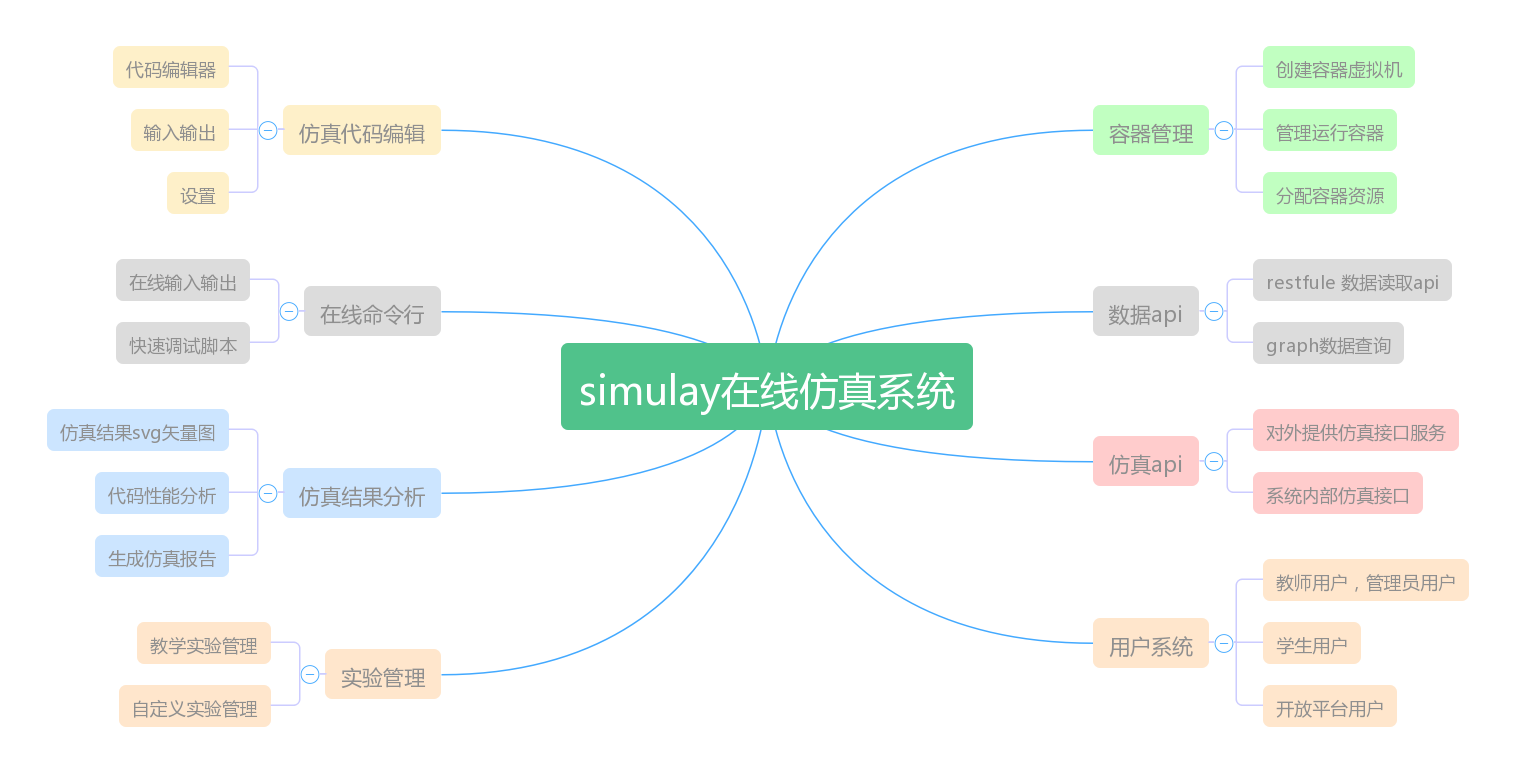
# 第一章 功能架构 与 技术架构

## 1.1 simulay 系统简介

simulay是一个基于分布式容器化web在线仿真教学交互演示系统， 其中核心的技术是基于容化的微服务运算平台，不仅有着分布式可扩展的优点， 同时在系统性能和系统安全等方面有着极大的提高。在功能上提供业界领先的 WebIDE 和WebTerminal 以供学生仿真调试使用。配合实验管理，容器管理等功能实完美的教学训练闭环。 在开放与可扩展性上 ， 提供开放的数据API，同时以API的形式开放仿真运算平台供开发者调用

## 1.2 simulay整体功能架构

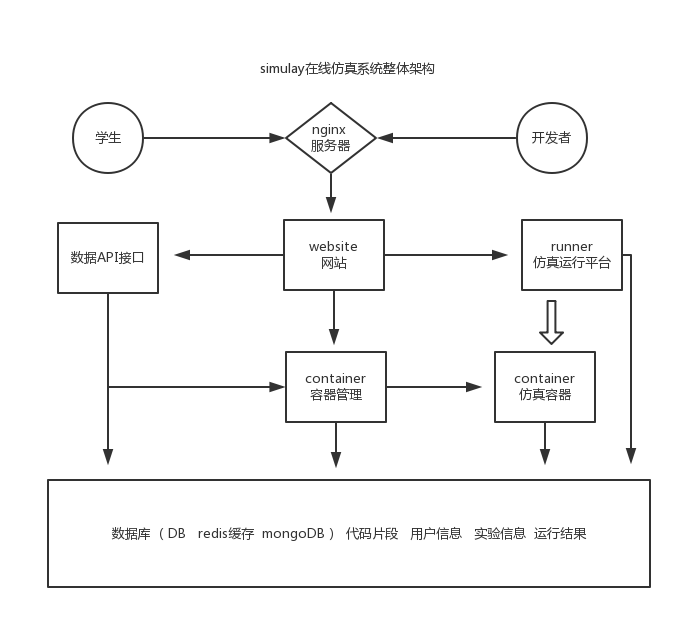
本项目着重研究基于目前最前沿的web前端与后台技术构建的可视化交互仿真系统，主要分为三大模块（子系统）： 1. 在线仿真系统：在线编辑，调试，统计， 管理实验的可视化的web前端展示系统 2. 容器管理：分布式 + 容器技术 + 仿真数据缓存，提高仿真系统的整体响应速度和系统的可扩展性 3. 开放API，数据API提供GraphQL查询接口， 运算API提供第三方调用仿真接口便于开发者使用。开放拓展系统的功能便于支持机器学习与数据挖掘。



**图1-1 simulay系统整体的功能框图**

在三大模块的基础上， 又对各个功能做了细分，划分为更小粒点的模块功能。

## 1.3 simulay技术架构



**图 1-2 simulay 在线仿真系统整体架构**

### 1.3.1在线仿真系统

系统是基于浏览器平台的，无论用户使用的是什么操作系统， 只需要一个webkit内核的浏览器，即可打开该系统。 如果把用户的使用仿真功能过程看作一个一个流的话：

1. 后台最开始接受到请求的是Nginx 负载均衡服务器（NGINX采用了异步、事件驱动的方法来处理连接，可以非常好地处理百万级规模的并发请求），
2. Nginx 服务器将请求转发到 基于NodeJS的WebSite服务程序，这个程序主要负责处理系统的业务逻辑，主要的工作就是对外接受请求， 对内调用各微服务， 最后把运算数据返回到浏览器前端。
3. webSite调用 runner仿真平台， runner调用 容器系统的接口创建一个容器，然后在容器中运行仿真代码。
4. 容器运算得出结果之后， 调用回调函数，把结果写入到底层的 数据库和缓存中， 同时更新该实验的结果数据， 最后把结果返回到浏览器前端。

### 1.3.2数据API平台

数据API接口是对外开放的开发者计划的一部分，主要是的功能就是把整个系统的各项功能包装一遍， 使用统一的接口风格，对外提供统一的API调用服务，对系统将每个API请求都转发到对应的模块中去，实验仿真数据查询， 容器管理，runner仿真运算平台等都可以进行API包装然后对外服务。

### 1.3.3容器管理系统

容器管理系统提供的在线容器创建，运行，暂停，删除等操作， 为用户提供一个轻量级的容器服务（类似虚拟机，但是在资源的占用率和创建启动速度上都完胜虚拟机），让用户以最少的资源最快的速度得到一个云服务资源。

# 第二章 容器化与微服务

## 2.1概念

#### 2.1.1容器化

容器为应用程序提供了隔离的运行空间，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，可以理解为轻量化的虚拟机，但是有虚拟机无法企及的创建速度和极小的资源占用，同时在部署方面十分灵活， 把传统的 dev（开发） 和 ops（运维） 结合一体 组成 devops的 开发模式。

#### 2.1.2微服务

微服务是一种架构风格，一个大型复杂软件应用由一个或多个微服务组成。系统中的各个微服务可被独立部署，各个微服务之间是松耦合的。每个微服务仅关注于完成一件任务并很好地完成该任务。在所有情况下，每个任务代表着一个小的业务能力。结合分布式的系统， 可以实现去“去中心化”数据管理(DecentralizedData Management)

## 2.2容器化使用的是目前的主流的docker技术

Docker对使用者来讲是一个C/S模式的架构，而Docker的后端是一个非常松耦合的架构。 用户是使用Docker Client与Docker Daemon建立通信，并发送请求给后者。 Docker 主要的模块有：Docker Client、Docker Daemon、Docker Registry、Graph、Driver、libcontainer以及Docker container。

在simulay系统中， 我们只需要关心一下三个部分即可：

Docker Client： 在simulay仿真系统中， 容器管理系统就是一个 Docker Client，Docker Client可以通过以下三种方式和Docker Daemon建立通信：tcp://host:port，unix://path\_to\_socket和fd://socketfd。Docker Daemon接受并处理请求。

Docker Daemon ： 是Docker架构中一个常驻在后台的系统进程，功能是：接受并处理Docker Client发送的请求。该守护进程在后台启动了一个Server，Server负责接受Docker Client发送的请求；接受请求后，Server通过路由与分发调度，找到相应的Handler来执行请求。

Docker container： Docker按照用户的需求与指令，订制相应的Docker容器：

用户通过指定容器镜像，使得Docker容器可以自定义rootfs等文件系统； 用户通过指定计算资源的配额，使得Docker容器使用指定的计算资源； 用户通过配置网络及其安全策略，使得Docker容器拥有独立且安全的网络环境； 用户通过指定运行的命令，使得Docker容器执行指定的工作。

参考文献： http://www.infoq.com/cn/articles/docker-source-code-analysis-part1 https://en.wikipedia.org/wiki/Docker\_(software)

## 2.3安装与镜像制作

Docker 官方为了简化安装流程，提供了一套安装脚本，Ubuntu 和 Debian 系统可以使用这套脚本安装：

//安装docker curl -sSL https://get.docker.com/ | sh

// 启动docker daemon sudo service docker start

// 下载ubuntu系统镜像 sudo docker pull ubuntu:14.04

// 启动一个基于ubuntu镜像的容器服务器 docker run -it --rm ubuntu:14.04 bash

// 安装 octave 仿真软件 apt-get update apt-get install octave // 退出容器 exit

// 提交镜像 sudo docker commit containerID outshien／octave：1.0 sudo docker push

至此， 仿真的镜像制作完成， 后续只需要运行一下命令即可进入octave命令行界面： sudo docker run -it outshine/octave:1.0 octave --no-gui –silent

## 2.4 DockerAPI

docker Daemon提供的 API封装，我的容器管理系统中 ，只需要实现类似的 http请求操作的封装， 即可实现容器的管理

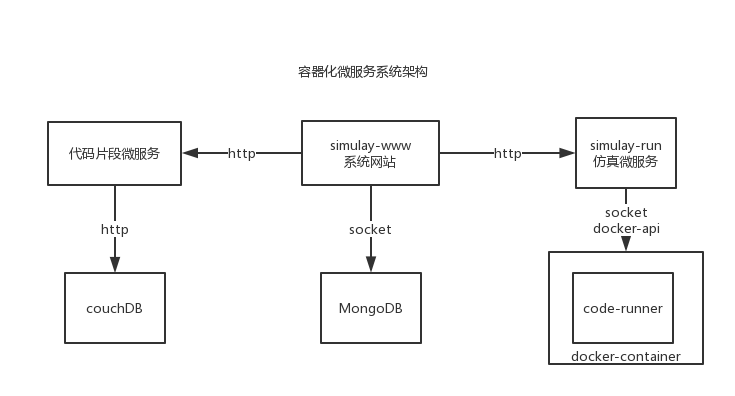
$ curl --unix-socket /var/run/docker.sock -H "Content-Type: application/json" \  
 -d '{"Image": "alpine", "Cmd": ["echo", "hello world"]}' \  
 -X POST http:/v1.24/containers/create  
 {"Id":"1c6594faf5","Warnings":null}  
  
 $ curl --unix-socket /var/run/docker.sock -X POST http:/v1.24/containers/1c6594faf5/start  
  
 $ curl --unix-socket /var/run/docker.sock -X POST http:/v1.24/containers/1c6594faf5/wait  
 {"StatusCode":0}  
  
 $ curl --unix-socket /var/run/docker.sock "http:/v1.24/containers/1c6594faf5/logs?stdout=1"  
 hello world

## 2.5微服务架构

整个simulay的系统都是基于微服务的架构的， 得益于 容器的容易创建， 每个子模块都运行在容器服务中，这里的一个原则就是 “容器即服务”。 罗列一下simulay系统中的微服务：

1. nginx 负载均衡服务器
2. mongoDB 数据服务
3. redis 服务
4. nodeJS 网站cgi 及 API接口 服务
5. 仿真运算runner 服务
6. 容器管理系统服务

以上的各个服务都是运行在独立的容器中， 互相之间通过rpc 调用， 在系统的耦合度上面做到低耦合高内聚



**图 2-1容器与微服务架构图**

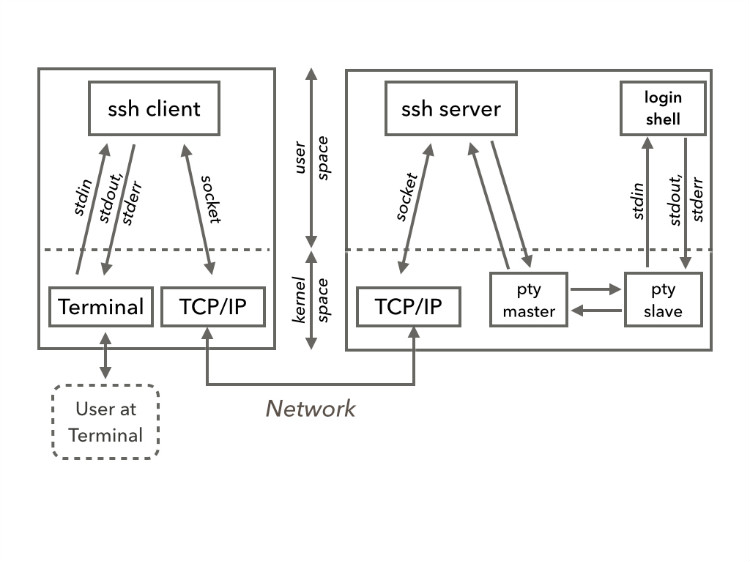
# 第三章 WebTerminal设计与实践

## 3.1 WebTerminal简介

WebTerminal 是一个基于hterm前端终端， websocket网页实时通信， 容器化运行shell的 实时高效安全的在线命令行，毫秒级创建运行容器并建立安全的socket连接，提供完美的用户体验，让在线版的octave 软件可以在任意一台电脑上无需安装即可使用。

## 3.2 TTY 与 SSH

Terminal 是键盘和显示器的组合，也称为 TTY（电传打字机的缩写）, Terminal 作为真实的物理设备已经不复存在了，但是为了和面向终端的程序（比如Bash）进行通信，于是就了发明了 pty（Pseudoterminal，伪终端）。当 Terminal Emulator 作为一个非面向终端的程序不直接与 pty slave 通讯，而是通过文件读写流与 pty master 通讯，pty master 再将字符输入经过线路规程的转换传送给 slave，slave 进一步传递给 bash。 关于命令行我们平时最常见就是 使用 ssh 远程登录服务器

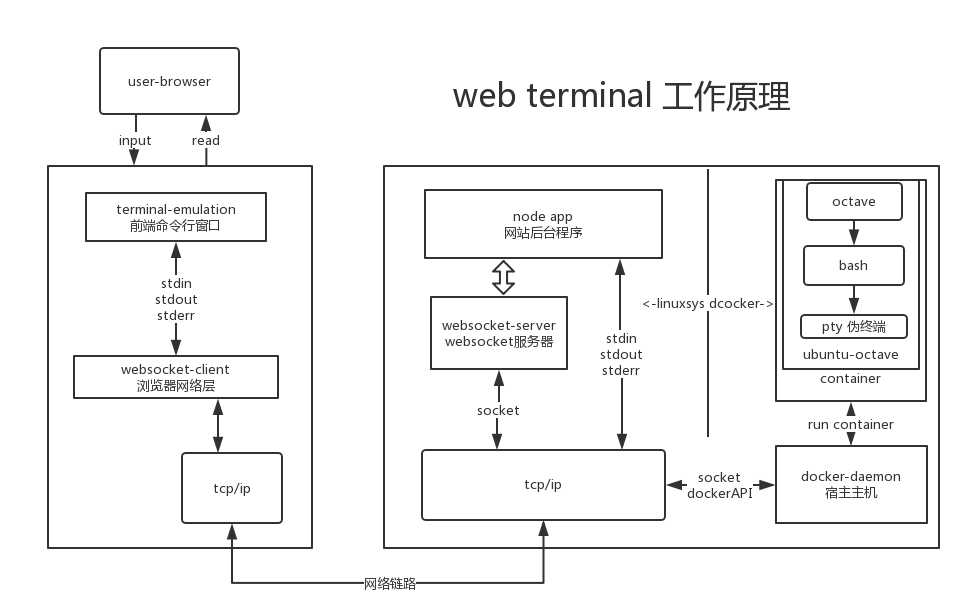


**图 3-1 SSH 的原理图**

SSH 是一个基于的 server-client 的架构，用户通过终端将字符流传递给 SSH client。SSH client 和 SSH server 之间通过 TCP/IP 协议进行通讯。远端的 server 创建一对 pty，并且fork+exec 一个 bash 进程，server 进程通过 pty 对与 bash 进行交互。

## 3.3 WebTerminal原理框图

基于 SSH 的工作原理，设计了 Web Terminal，见下图



**图3-2 Web Terminal工作原理**

## 3.4 WebTerminal工作流程

1. 用户在浏览器中打开终端页面，系统自动建立连接
2. 调用 浏览器 websocket client （js调用的是 socket.io)
3. tcp/ip 三次握手, 与服务端建立 socket 链接,socket 服务器开始接受前端传送的数据
4. node APP 通过 docker daemon 的 api 接口创建 容器
5. 容器启动, 载入 ubuntu-octave 镜像 , 并运行 octave 命令
6. octave 是输出通过到系统的 io中, 容器捕捉到系统的输出之后, 通过 docker daemon 的 httpserver 发送到 node 后台,
7. node 把 容器传输的运行结果数据 调用 socket 的 emit() 方法发送到浏览器前端.
8. 前端接收到数据 , 传入hterm 前端的 Terminal Emulator 展示

# 第四章 前端架构设计

## 4.1架构简介

simulay 浏览器前端系统采用当前业界最先进的前端开发模式, 采用前后端分离的架构, 用 web app 的概念取代以前的网页的概念, 整个 simulay 系统就是一个 spa(单页应用). 一次加载, 页面内切换如同原生 app 一般快速流畅. 前端的底层使用 vuw 框架, 前端的界面使用 iview 的前端组件库作为基础组件, 使用 webpack 作为前端打包工具. 使用一套现代化的前端开发模式.

## 4.2前端的工程

simulay 的 前端子项目使用 vue-cli 创建前端的工程项目, 是一个现代化前端工程化的项目结构,主要分为以下几个方面:

### 4.2.1模块化

整个单页应用分为不同的模块, view(页面层)就是将不同页面拆成不同的文件. 简单来说，模块化就是将一个大文件拆分成相互依赖的小文件，再进行统一的拼装和加载。

### 4.2.2组件化

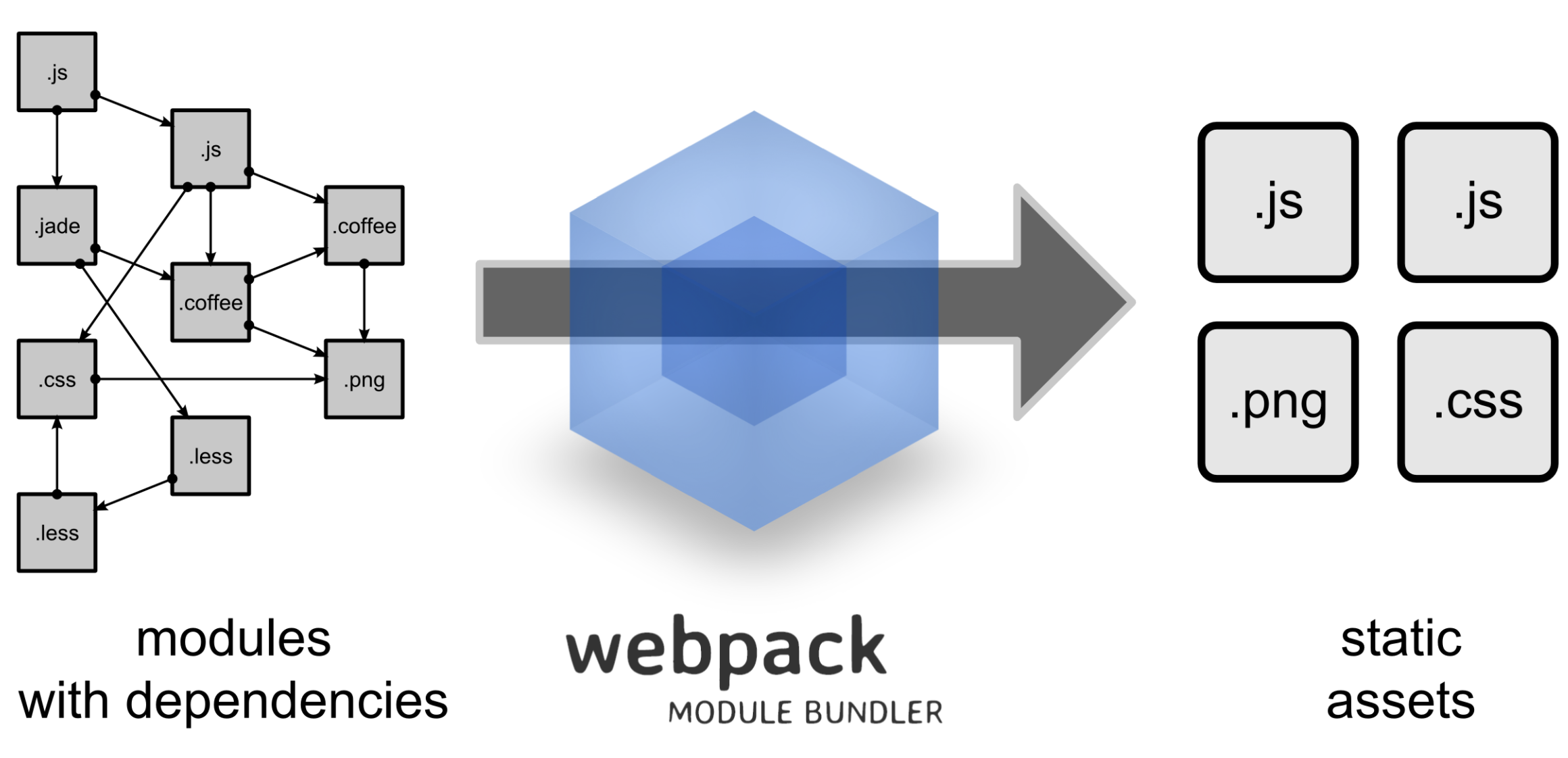
组件化是基于模块化，在设计层面上，对UI（用户界面）的拆分。从UI拆分下来的每个包含模板(HTML)+样式(CSS)+逻辑(JS)功能完备的结构单元称之为组件。其实，组件化更重要的是一种分治思想。页面是个大型组件，可以拆成若干个中型组件，然后中型组件还可以再拆，拆成若干个小型组件，小型组件也可以再拆，直到拆成DOM元素为止。DOM元素可以看成是浏览器自身的组件，作为组件的基本单元。  
  
![](img/组件区分图.png)

### 4.2.3规范化

规范化模块化和组件化确定了开发模型，而这些东西的实现就需要规范去落实。规范化其实是工程化中很重要的一个部分，项目初期规范制定的好坏会直接影响到后期的开发质量。其中编码规范最好采取ESLint和StyleLint等强制措施，因为人是靠不住的，比如可以Lint通不过不能提交代码或者直接编译失败报错.

## 4.3自动化构建

前端目前的代码已经不是传统的引用几个 js 脚本了, 而是一个规范的软件工程项目, 所有的代码与样式都要通过前端的编译打包工具才能发布出去.

构建工具任务流 

**图 4-1构建工具任务流**

得益于先进的打包工具, 在开发中我们用工程化和模块化组件化的思路去开发, 一方面提高了项目的规范性,另增加了项目结构的清晰程度. 开发的时候无需考虑部署到浏览器中问题, 因为有 webpack 打包编译流程, 自动化的把项目各种资源分类打包输出到目录中.

## 4.4自动化部署

自动化部署是基于 github 的 webhook 功能实现的 主要的步骤如下

### 4.4.1自动化shell脚本

在服务器创建一个 auto\_build.sh脚本:

#! /bin/bash  
SITE\_PATH='https://github.com/outshineamaze/simulay.git'  
USER='admin'  
USERGROUP='admin'  
cd $SITE\_PATH  
git reset --hard origin/master  
git clean -f  
git pull  
git checkout master  
chown -R $USER:$USERGROUP $SITE\_PATH

### 4.4.2服务端实现

Github webhooks需要跟我们的服务器进行通信，确保是可以推送到我们的服务器，所以会发送一个带有X-Hub-Signature的POST请求，为了方便我们直接用第三方的库github-webhook-handler来接收参数并且做监听事件的处理等工作。

var http = require('http');  
var spawn = require('child\_process').spawn;  
var createHandler = require('github-webhook-handler');  
// 下面填写的myscrect跟github webhooks配置一样；path是我们访问的路径  
var handler = createHandler({ path: '/auto\_build', secret: '' });  
http.createServer(function (req, res) {  
 handler(req, res, function (err) {  
 res.statusCode = 404;  
 res.end('no such location');  
 })  
}).listen(6666);  
handler.on('error', function (err) {  
 console.error('Error:', err.message)  
});  
// 监听到push事件的时候执行我们的自动化脚本  
handler.on('push', function (event) {  
 console.log('Received a push event for %s to %s',  
 event.payload.repository.name,  
 event.payload.ref);  
 runCommand('sh', ['./auto\_build.sh'], function( txt ){  
 console.log(txt);  
 });  
});  
function rumCommand( cmd, args, callback ){  
 var child = spawn( cmd, args );  
 var response = '';  
 child.stdout.on('data', function( buffer ){ resp += buffer.toString(); });  
 child.stdout.on('end', function(){ callback( resp ) });  
}  
});

到这一步服务已经跑起来了，但是对外网并不能直接访问到，所以还需要配置一下Nginx做一下反向代理：

server {  
 listen 80;  
 server\_name simulay.outshine.me;  
 location /auto\_build {  
 proxy\_pass http://127.0.0.1:6666;  
 }  
}

4.4.3 配置github webhooks

进入 https://github.com/outshineamaze/simulay/settings/hooks ,配置 webhook, 我们可以在我们的Github上面最右边有一个Settings的Tab，找到Webhooks & services

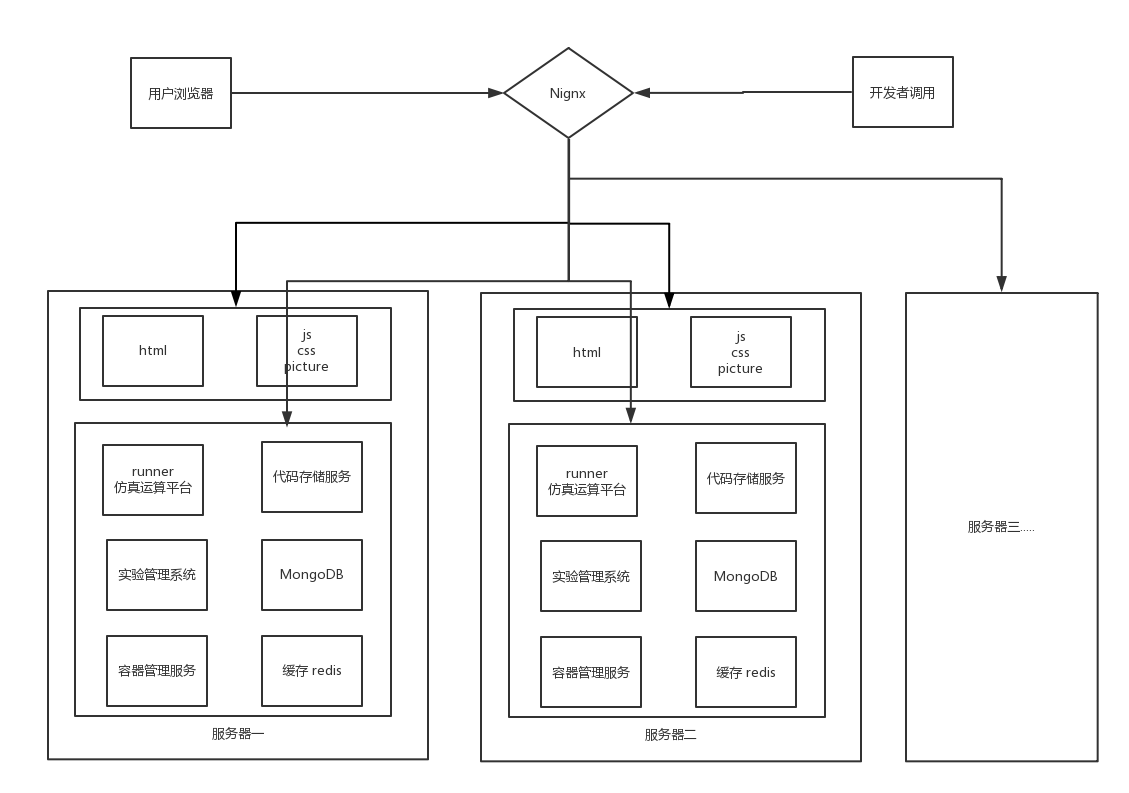
整个自动化部署配置完成之后, 我们发布代码的模式就是

1. 本地开发-测试
2. git commit && git -push
3. github 收到 push 请求, 触发 webhook , 发一个请求到我们的生产服务器
4. 生产服务器接收github 请求之后 自动运行 auto\_build.sh
5. 部署成功

# 第五章 后台架构设计

## 5.1 Nignx (负载均衡,动静分离)

Nginx("engine x")是一款是由俄罗斯的程序设计师Igor Sysoev所开发高性能的 Web和 反向代理 服务器 在本系统中主要负责 负载均衡,动静分离



**图 5-1 Nginx 负载均衡反向代理动静分离**

### 5.1.1负载均衡

nginx反向代理其实主要通过配置proxy\_pass参数即可代理到某个服务器 添加如下配置即可：

vim /usr/local/nginx/conf/nginx.conf  
location /api {   
 // 将不同路径的请求代理到不同的微服务中去.  
 proxy\_pass http://127.0.0.1:8080  
 }

### 5.1.2动静分离

动静分离将网站静态资源（HTML，JavaScript，CSS，img等文件）与后台应用分开部署，提高用户访问静态代码的速度，降低对后台应用访问。这里我们将静态资源放到nginx中，动态资源转发到各个容器服务中。 在动静分离的过程中, 只需要将静态资源剥离出来即可:

//静态资源   
 location /dist/ {   
 //静态资源到nginx服务器下dist (前端编译出的目录文件)  
 alias /data/release/simulay-front/dist/  
 }

## 5.2基于nodeJS express 的后台管理系统

Express 是一种保持最低程度规模的灵活 Node.js Web 应用程序框架，为 Web 和移动应用程序提供一组强大的功能。

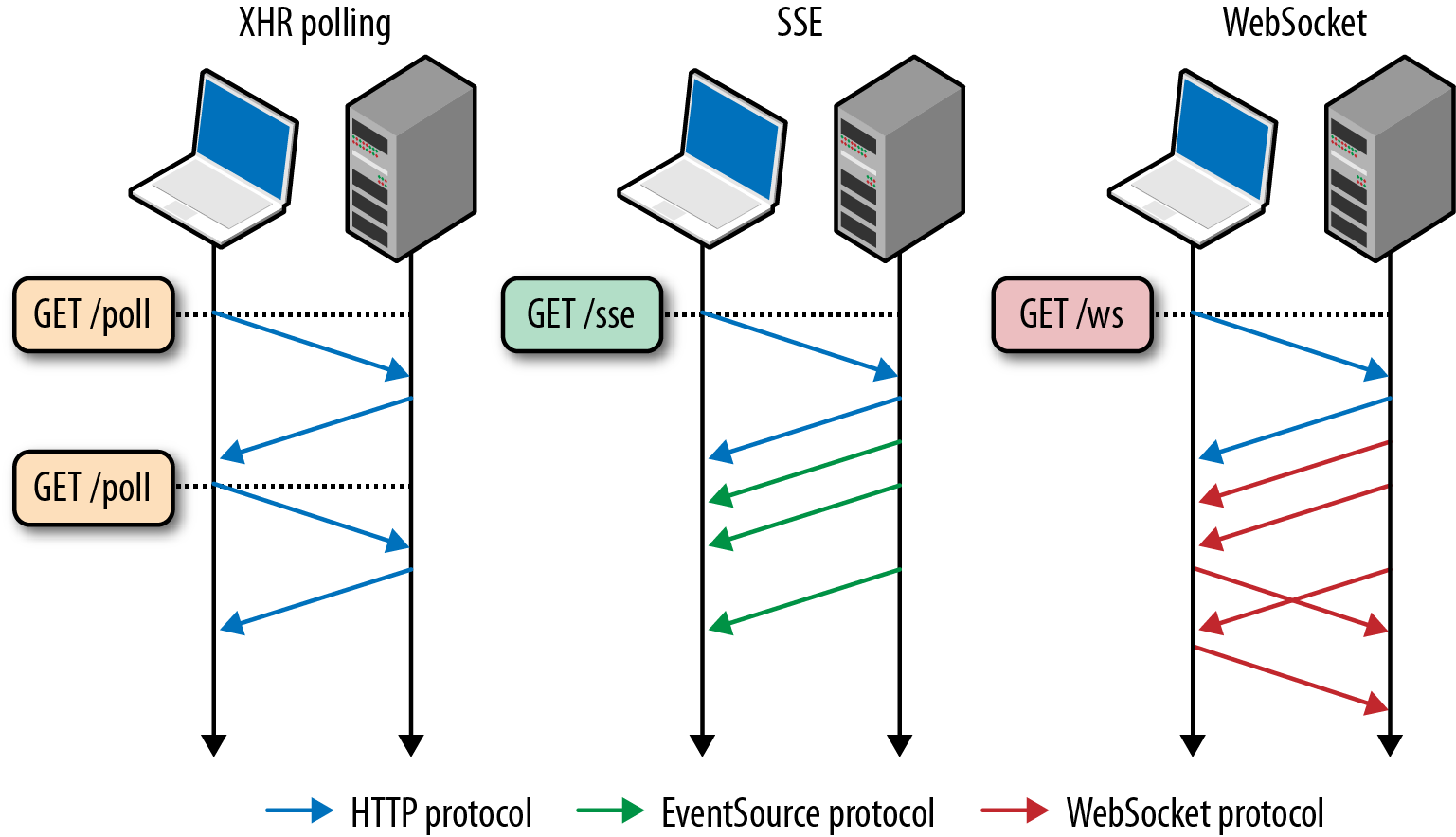
与传统的后台 架构不同的是, simulay 整个系统都是基于前后端分离模式的, 后端只需要做一件事: 以 api 的形式对外提供服务能力. 一般的后端管理系统都是一个 mvc 设计模式, 分为三层: view(视图层)-controller(业务逻辑)-model(数据模型层), 而在simulay 的系统后台中, 没有 view 层,取而代之的是 router(路由层), controller 从 model 取得数据之后,直接 encode 成 json 的数据包返回到前端的 spa(单页应用)中去. 在 simulay 后台中, 主要分为 router - controller- model 。

1. router: 路由层, 顾名思义,将不同地址的请求导航到 不同的 controller 中处理请求
2. controller: 控制层, 又名服务层, 主要 处理当前请求的业务逻辑, 负责读取请求参数, 发起数据查询请求,筛选组装查询结果, 封装 response 返回包.
3. model: 模型层, 底层的数据库都是由不同的表组成的,比如(用户表, 实验表, 仿真记录表等), 如何读取这些数据呢, 如果是用sql 语句的话效率很低, 这里使用了 mongoose 这个 ORM 对象模型连接器, 也即是数据库中的数据映射到 程序中的对象, 直接调用对象方法获取数据即可,把数据库连接 sql 查询等操作做一步封装. 简化代码.

## 5.3 websocket 通信

在 webTerminal 中 提到多次 websocket 这个概念. 首先解释一下它是什么,

WebSocket一种在单个 TCP 连接上进行全双工通讯的协议。WebSocket通信协议于2011年被IETF定为标准RFC 6455，并被RFC7936所补充规范。WebSocket API也被W3C定为标准。  
  
它使得客户端和服务器之间的数据交换变得更加简单，允许服务端主动向客户端推送数据。在 WebSocket API 中，浏览器和服务器只需要完成一次握手，两者之间就直接可以创建持久性的连接，并进行双向数据传输。 ![websocket](https://zh.wikipedia.org/zh-hans/WebSocket)



#### 5.3.1 WebSocket有以下特点

是真正的全双工方式，建立连接后客户端与服务器端是完全平等的，可以互相主动请求。而HTTP长连接基于HTTP，是传统的客户端对服务器发起请求的模式。  
HTTP长连接中，每次数据交换除了真正的数据部分外，服务器和客户端还要大量交换HTTP header，信息交换效率很低。Websocket协议通过第一个request建立了TCP连接之后，之后交换的数据都不需要发送 HTTP header就能交换数据，这显然和原有的HTTP协议有区别所以它需要对服务器和客户端都进行升级才能实现（主流浏览器都已支持HTML5）。此外还有 multiplexing、不同的URL可以复用同一个WebSocket连接等功能。这些都是HTTP长连接不能做到的。

#### 5.3.2性能测试

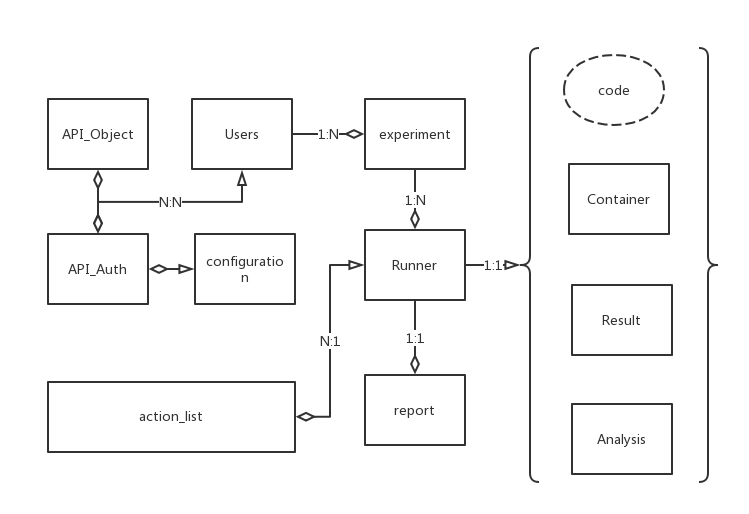
在 WebTerminal 中, 如果使用 XHR Polling (传统浏览器中 AJAX 请求,目前大部分网站前端的请求都是以这种形式发起)的话, 将平均带来几百倍的请求量消耗而且这大部门的请求都是无效的 ping 请求, 具体的区别可以看下面的测试图: 

其中Ops/sec 是指请求次数与总耗时之比放大1000倍,可以理解为平均请求时间, 可以看出来websocket 比传统的 xhr 请求快了三倍,同时, 只建立一条 tcp/ip 连接, 对网络资源占用是 xhr 1/N(实验次数)。

# 第六章 存储系统NoSQL在simulay 中的使用

simulay 系统没有使用传统的 关系型数据库, 而是使用当前最为前沿的 NoSql, NoSQL是对不同于传统的关系数据库的数据库管理系统的统称。

## 6.1 web后台 MongoDB

在 simulay 后台系统中使用了 MongoDB 作为底层的数据库,MongoDB是一种文档导向数据库管理系统, 在后台中, simulay 系统结合了 mongoose 的 orm 模型对象封装, 整体的 DB 模型如下: 

**图 6-1 整体 DB 模型图**

## 6.2代码存储 couchDB

CouchDB（Couch是 cluster of unreliable commodity hardware的首字母缩写） 特性:REST API 所有的数据都有一个唯一的通过HTTP暴露出来的URI。REST使用HTTP方法 POST，GET，PUT和DELETE来操作对应的四个基本CRUD(Create，Read，Update，Delete）操作来操作所有的资源。

基于以上的特性, simulay 系统把所有的代码片段(snippets)存储独立出来, 放在在 couchDB 中, 然后通过 http请求从 couchDB 中读取代码,同时, 这个 couchDB 也可以通过 API 平台转发为第三方提供服务.

### 6.2.1 API操作接口

|  |  |  |  |
| --- | --- | --- | --- |
| Action | Method | Route | Requires token |
| [List snippets](list_snippets.md) | GET | /snippets | Optional |
| [Create snippet](create_snippet.md) | POST | /snippets | Optional |
| [Get snippet](get_snippet.md) | GET | /snippets/:id | No |
| [Update snippet](update_snippet.md) | PUT | /snippets/:id | Yes |
| [Delete snippet](delete_snippet.md) | DELETE | /snippets/:id | Yes |

### 6.2.2 示例代码

##### Create snippet  
 curl --request POST \  
 --header 'Content-type: application/json' \  
 --data '{"language": "octave", "title": "test", "public": true, "files": [{"name": "main.m", "content": "print(42)"}]}' \  
 --url 'https://simulay.outshine.me/snippets'  
##### Update snippet (requires api token)  
 curl --request PUT \  
 --header 'Authorization: Token 0123456-789a-bcde-f012-3456789abcde' \  
 --header 'Content-type: application/json' \  
 --data '{"language": "octave", "test - updated": "test", "public": false, "files": [{"name": "main.m", "content": "print(42)"}]}' \  
 --url 'https://simulay.outshine.me/snippets/e374yxuw49'  
##### Get snippet  
 curl --request GET \  
 --url 'https://simulay.outshine.me/snippets/e2tx9nh4fh'  
##### List your snippets on page 5 with 3 snippets per page  
 curl --request GET \  
 --header 'Authorization: Token 0123456-789a-bcde-f012-3456789abcde' \  
 --url 'https://simulay.outshine.me/snippets?page=5&per\_page=3'

# 第七章 API 设计

## 7.1 RESTful API 设计简介

simulay 不仅仅是一个在线仿真系统, 同时也有对外提供接口服务的能力, 整套 API 的设计参考 REST架构风格的 API 设计规范, REST（英文：Representational State Transfer，又称具象状态传输）是Roy Thomas Fielding博士于2000年在他的博士论文 [ Fielding, Roy Thomas. Chapter 5: Representational State Transfer (REST). Architectural Styles and the Design of Network-based Software Architectures (Ph.D.). University of California, Irvine. 2000. This chapter introduced the Representational State Transfer (REST) architectural style for distributed hypermedia systems. REST provides a set of architectural constraints that, when applied as a whole, emphasizes scalability of component interactions, generality of interfaces, independent deployment of components, and intermediary components to reduce interaction latency, enforce security, and encapsulate legacy systems.]

中提出来的一种万维网软件架构风格，目的是便于不同软件/程序在网络（例如互联网）中互相传递信息。

主要有三大特点

1. 资源是由URI来指定。
2. 对资源的操作包括获取、创建、修改和删除资源，这些操作正好对应HTTP协议提供的GET、POST、PUT和DELETE方法。
3. 通过操作资源的表现形式来操作资源。

## 7.2数据读取API

|  |  |  |  |
| --- | --- | --- | --- |
| Action | Method | Route | Requires token |
| [List runner](list_runner.md) | GET | /runner | Optional |
| [Get runner](get_runner.md) | GET | /runner/:id | No |
| [Update runner](update_runner.md) | PUT | /runner/:id | Yes |
| [Delete runner](delete_runner.md) | DELETE | /runner/:id | Yes |
| [Get result](get_runner.md) | GET | /runner/:id/result | No |
| [Get analysis](get_runner.md) | GET | /runner/:id/analysis | No |

### 7.2.1 Get runner

curl --request GET \  
 --url 'HTTPS://simulay.outshine.me/runner/e2tx9nh4fh'

### 7.2.2 Example response data

{  
 "id": "e2tx9nh4fh",  
 "url": "HTTPS://simulay.outshine.me/runner/e2tx9nh4fh",  
 "created": "2017-04-23T22:03:11Z",  
 "modified": "2017-04-23T22:03:11Z",  
 "hash": "9bdd2b79fafbf81313a79b1df1be5c2671422307",  
 "language": "octave",  
 "title": "main",  
 "public": true,  
 "owner": "anonymous",  
 "files": [  
 {  
 "name": "main.m",  
 "content": "2+2  
 }  
 ],  
 result\_id: '23423423',  
 analysis\_id: '23424251',  
}

## 7.3仿真runnerAPI

|  |  |  |  |
| --- | --- | --- | --- |
| Action | Method | Route | Requires token |
| [List languages](list_languages.md) | GET | /languages | No |
| [List versions](list_versions.md) | GET | /languages/:language | No |
| [Run code](run.md) | POST | /languages/:language/:version | Yes |

### 7.3.1 Run code

curl --request POST \  
 --header 'Authorization: Token 0123456-789a-bcde-f012-3456789abcde' \  
 --header 'Content-type: application/json' \  
 --data '{"files": [{"name": "main.m", "content": "2+2"}]}' \  
 --url 'https://simulay.outshine.me/languages/octave/latest'

### 7.3.2 Simple example

Request

{  
 "files": [  
 {  
 "name": "main.m",  
 "content": "2+2"  
 }  
 ]  
}

##### Response

{  
 "stdout": "4",  
 "stderr": "",  
 "error": ""  
}

# 第八章 系统优化以及缓存

simulay 为了提升系统整体的加载数据, 特别从以下的几个方面做了性能优化

1. 前端文件压缩打包, 同时以单页应用的方式开发 整个系统
2. WebTerminal 复用 tcp/IP, 减少 TCP三次握手的请求时长
3. 仿真结果缓存, 对于每一次实验在系统内部都是一个 runner 的对象, 假如每次把仿真的对象缓存到 DB 中, 那么下次如果有相同代码和输入的仿真请求,就可以直接从 DB 中返回仿真结果了
4. 仿真结果的图片以 jpg,png, svg 矢量图的等多元格式返回到前端. 用最快的速度,最小的资源占用,加载出显示效果最好的图片
5. 使用基于内存的的 key-value 缓存数据库 redis

## 8.1仿真结果缓存

1. 用户发起一个仿真请求
2. 后台接受仿真请求,并计算(代码 + 输入) 的hash 值
3. 在 redis 查询 hash 值为 key 的 仿真结果缓存对象
4. 如果查询到结果直接返回, 未查询到结果, 创建一个 runner 对象
5. 仿真结束之后, 数据DB 写入一份, redis 缓存写入一份, 再返回到前端一份

//仿真结果缓存的伪代码

// 用户请求  
var request = {  
 name: ‘main.m’,  
 content: ‘2+2’  
}  
if (var result = redis.get(MD5(name + content))) {  
 return result;  
} else {  
 runner = new Runner(content);  
 runner.save(); // 存储到数据库  
 redis.set(MD5(name + content), runner.reusult) // 存储到 redis 缓存  
 return runner.result;  
}

使用仿真结果缓存带来的是每次仿真都有计算一次(代码 + 输入) 的hash值, 计算 hash 使用的是 MD5消息摘要算法（英语：MD5 Message-Digest Algorithm），一种被广泛使用的密码散列函数，可以产生出一个128位（16字节）的散列值（hash value），用于确保信息传输完整一致。MD5由罗纳德·李维斯特设计，于1992年公开，用以取代MD4算法。这套算法的程序在 RFC 1321[https://tools.ietf.org/html/rfc1321] 中被加以规范。

## 8.2 SVG 矢量图在系统的运用

SVG是一种可缩放矢量图形（英语：Scalable Vector Graphics，SVG）是基于可扩展标记语言（XML），用于描述二维矢量图形的图形格式。SVG[https://www.w3.org/TR/SVG/]由W3C制定，是一个开放标准。 特点: SVG提供的功能集涵盖了嵌套转换、裁剪路径、Alpha通道、滤镜效果等能力，它还具备了传统图片没有的矢量功能，在任何高清设备都很高清

svg 格式的矢量图大小取决于图片内容的丰富程度, 图片内容越丰富,大小会剧增加, 但是换来的是无限放大都无损的特性, 而 使用 png 格式的图片

在 simulay 系统主要是使用了 svg 和 jpg(有损压缩算法) 在生成仿真结果时, 会同时生成 svg 和 jpg 的图片, 后台会先返回 jpg 图片到端显示结果, 当用点击大图的时候加载 svg 矢量图

生成的图片取决于用户的实际需求,是需要更快的加载速度, 还是跟清晰的仿真结果图, 所以在仿真界面设置了一个 输出图片格式的选项, 当用户选择自动时, 会默认显示压缩过jpg, 当 svg 加载成功之后, 自动把 jpg 替换掉.

## 8.3 redis缓存系统

Redis是一个开源（BSD许可）的，内存中的数据结构存储系统，它可以用作数据库、缓存和消息中间件。 它支持多种类型的数据结构，如 字符串（strings）， 散列（hashes）， 列表（lists）， 集合（sets）， 有序集合（sorted sets） 与范围查询， bitmaps， hyperloglogs 和 地理空间（geospatial） 索引半径查询。 Redis 内置了 复制（replication），LUA脚本（Lua scripting）， LRU驱动事件（LRU eviction），事务（transactions） 和不同级别的 磁盘持久化（persistence）， 并通过 Redis哨兵（Sentinel）和自动 分区（Cluster）提供高可用性（high availability）。

redis 在系统中主要充当 缓存的作用

1. 普通 DB 查询请求的缓存, 比如在 api 中, 必然有大量一样的查询,每次只需从 reds 中取数据即可, 更新数据时刷新 redis 中的数据缓存即可
2. 利用其有序集合(sorted sets), 可以做一个排名列表, 比如最受欢迎的实验,最耗时的实验,调用次数最多的 api等,无需每次都从 DB中耗费大量时间查询.
3. 利用其列表（lists)对象, 做任务队列,后期如果系统访问量剧增的时候, 比如1秒内有1000多次仿真, 这时候就会进入任务队列,并放回自动把当前的仿真任务添加到任务队列中(大部分时候系统负载较小无法触发这种情况,但是simulay也考虑到, 并提供好方案处理)

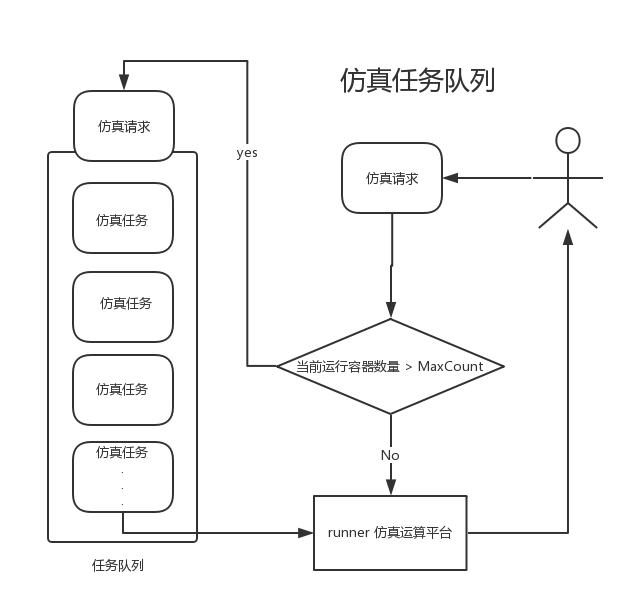


图 8-1 仿真任务队列