**Digu前端系统1.3.1-1.3.2的技术文档**

[**运行依赖环境**](#_mljz0xhw0a5) **2**

Docker安装 2

Windows安装 2

非容器开发环境 2

[**统架构更新点**](#_ug9m4syqljbc) **3**

[容器化](#_38hxfrqplqds) 3

[前后端分离](#_w95qxctb0pul) 4

[pm2守护进程](#_m561c4bvcckk) 4

[相关pm2配置文件](#_qkvjc45nd93n) 4

[**项目目录结构**](#_z97tx1q8vyg7) **5**

[**Docker架构流程图**](#_pqo5yxlusy22) **7**

[**开发使用工具命令**](#_3jpipjhus252) **8**

[非Docker开发环境，安装yarn](#_ee4h2ngrghj1) 8

[**日常提测与部署流程**](#_d70bud7u6kke) **9**

[提测流程](#_temunga1d35h) 9

[部署流程](#_fa08f28v7k4m) 10

[代码托管交付](#_i9wxtytltjah) 10

[**设计说明**](#_r3pc8y1y33ni) **11**

[选型与升级](#_r1cc9t83cn5h) 11

[主要的样式文件](#_u0phbk8b3yv4) **11**

[反向代理/node中间层设计](#_7guvduepbupl) 11

[中间件依赖](#_va7krf5kt9og) 12

[组件设计与提炼](#_z6yhfanvmz19) 12

[音频播放器](#_km4tsxyx5sj7) 12

[列表匹配模式](#_1woparcktri2) 13

[列表匹配模式主要模块功能](#_7mx4lw92rvw5) 15

[列表匹配模式与话术编辑器PostMessage通讯](#_n5ldx5rx4kmh) 18

[**会话列表1.3.1更新**](#_ivmess9677gx) **18**

[**话术列表1.3.1更新**](#_dzxkms4rwwpe) **18**

**知识库**[**主要更新**](#_hozlxxx3hv8n) **19**

[**外呼设置**](#_z7rzkix6w3) **19**

[**话术编辑器1.3.2更新**](#_xuxq8nv53ke3) **19**

# 运行依赖环境

#### Docker开发与生产环境：Window/Linux/Mac平台 Docker18.06+环境以上,安装方式参考：[史上最全（全平台）docker安装方法！](https://zhuanlan.zhihu.com/p/54147784)。

#### window环境docker需要开启内存虚拟化：相关教程：[Windows 10下Docker使用经验谈- dax.net](https://www.cnblogs.com/daxnet/p/7719574.html)

#### Docker环境适用与开发环境，**生产环境目前必须要用Docker环境运行**。

#### 非容器化开发环境：Window/Linux/Mac平台 node.js10+ Yarn1.22.0+，

# 

# 统架构更新点

### 

## 容器化

考虑到目前项目的运行环境不固定，每次安装部署前端环境流程较多，在初始化服务器时需要安装过多的前端相关依赖，前端容器化后可以进行**回滚**，**扩容**等，生产开发测试环境**版本一致性**，生产无缝版本升级等。

而在开发环境，存在初始化开发环境困难，开发环境容器化后可以使前后端协作变得容器，不同前端工程师始终保持**环境保持统一**，整个开发 -> 提测 -> 部署可以使用同个编译包完成，降低线上线下不同环境前端编译而造成的出错可能性。

### 

## 前后端分离

1.3.1版本前的项目前端与后端Java环境耦一起，且基本通过本机127.0.0.1访问后端接口进行开发，或者通过webpack server进行反向代理，生产环境前端完全依赖Java环境把前端代码打入jar包内部署。

1.3.1后使**前后端分离**模式，便于前后端开发人员的协作，前端版本可以**独立迭代**与发布，分清职责。

## pm2守护进程

前端生产环境服务启动时使用了pm2作为守护进程，[PM2 守护进程工具介绍及使用](https://www.blog8090.com/pm2-start/)，生产环境启动前端服务时会在容器中启动2个以上的实例，做到多实例负载均衡，crash自动重启等特性。

#### 相关pm2配置文件

测试调试： /pm2.config.dev.json  
生产环境： /pm2.config.prod.json

# 项目目录结构

|-- build 前端构建脚本

| |-- config 通用的webpack配置文件

| |-- globalVariables 全局变量的注入

| |-- dev.js 测试环境全局变量的注入

| |-- prod.js 生产环境全局变量的注入

|

|-- common-static 通用的静态资源

|

|-- front-source 前台代码

| |-- flow 前台话术编辑器

| |-- src 前台vue界面

| |-- static 前台通用静态资源

| |-- index.html 前端页面主入口

|

|-- manage-source 管理后台代码

| |-- flow 管理后台编辑器

| |-- src 管理后台vue界面

| |-- static 管理后台通用静态资源

| |-- index.html 管理后台页面主入口

|

|-- node\_modules library root

|

|-- server nodejs用于反向代理服务器

| |-- config 配置文件，定义后端服务器地址和端口

| |-- controllers 控制器

| |-- services 服务层，目前没有逻辑实现

| |-- staticFile 可存放一些静态资源（xls）用于下载

| |-- util 工具代码

| |-- router.js 路由定义

|

|-- .babelrc bable 编译配置文件

|

|-- .dockerignore docker忽略文件

|

|-- .gitignore git忽略文件

|

|-- .postcssrc.js postcss配置文件

|

|-- .prettierrc 代码风格配置文件

|

|-- dev.backend.Dockerfile 开发环境单独启动管理后台的dockerfile

|

|-- dev.Dockerfile 开发环境dockerfile（前+后台）

|

|-- dev.front.Dockerfile 开发环境前台dockerfile

|

|-- Dockerfile 生产环境前台dockefile

|

|-- manage.Dockerfile 生产环境后台dockerfile

|

|-- nodemon.json 开发环境修改代码后自动重启nodeserver

|

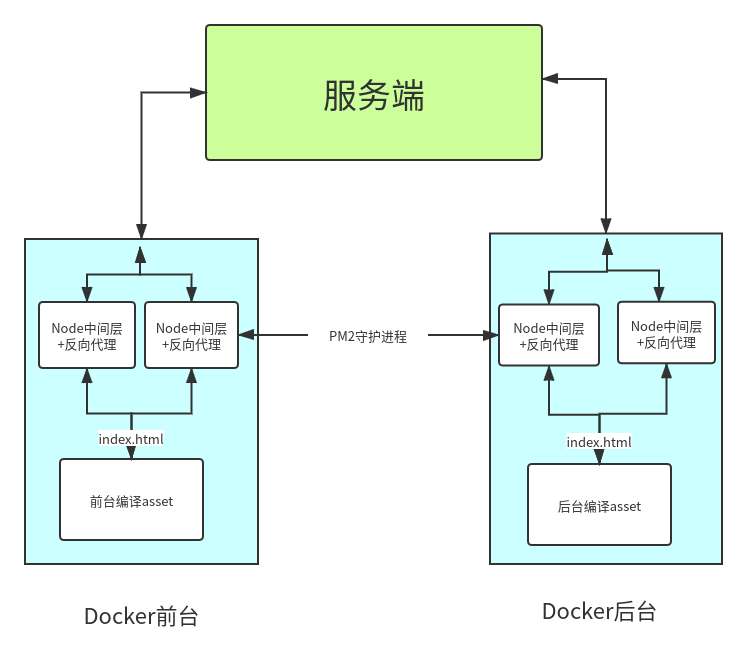
|-- pm2.config.dev.json 开发环境测试pm2配置文件

|

|-- pm2.config.prod.json 生产环境pm2配置文件

# Docker架构流程图

# 



测试环境如果使用docker，docker只作为runtime环境使用，相关开发的代码文件实质挂载到开发机器的硬盘中。开发时代码修改更新修改无需重新打包docker。

# 开发使用工具命令

### 

### 非Docker开发环境，安装yarn

安装依赖：**yarn install**

前台 flow 安装依赖：**cd front-source/flow & yarn install**

启动前台开发Watch:  **yarn dev**

启动后台开发Watch: **yarn dev:manage**

Docker开发环境（可安装yarn/npm，或直接执行package.json中命令）

构建**前台+反向代理**代理镜像:  **npm dev:docker-build**

启动**前台+反向代理**容器: **npm dev:docker-run**

Daemon方式启动**前台+反向代理**: **npm dev:docker-run-d**

停止**前台+反向代理**镜像:  **npm dev:docker-build**

构建反向代理服务镜像:  **npm dev:docker-build-backend**

启动反向代理服务镜像:  **npm dev:docker-run-backend**

Daemon方式单独启动反向代理:  **npm dev:docker-run-d-backend**

关闭反向代理服务:  **npm dev:docker-stop-backend**

构建单独构建前台前端:  **npm dev:dev:docker-build-front**

启动前台前端容器:  **npm dev:docker-run-front**

Daemon方式启动前台前端容器:  **npm dev:docker-run-d-front**停止前台前端容器:  **npm dev:dev:docker-stop-front**

删除所有系统中镜像： **npm dev:docker-remove-all-image**

*注：目前由于端口重复问题，无法同时启动前台和后台*

# 日常提测与部署流程

目前 *测试环境* 与 *生产环境* 没有自动化**CI/CD**流程，所以之前的提册与部署都是通过Docker镜像打包后导出的方式提供给QA人员进行测试，提交给后端人员进服务行部署。

后期有了 **CI/CD** 流程后，就可以把Docker带来的便捷发挥更大。

## 

## 提测流程

制作前台制品包：**docker build -t digu-prod --build-arg ACTION\_COMMAND=prod .**

制作管理后台制品包：**docker build -t digu-prod-manage --build-arg ACTION\_COMMAND=prod -f manage.Dockerfile .**

导出前台镜像：**docker save -o ~/digu-prod.tar digu-prod**

导出管理后台镜像：**docker save -o ~/digu-prod-manage.tar digu-prod**

测试/后端人员导入镜像到测试机：**docker load < ~/digu-prod.tar**

测试/后端人员导入管理后台镜像到测试机：**docker load < ~/digu-prod-manager.tar**

启动前台容器**：docker run -p 4001:4001 digu-prod**（指定4001端口访问测试）

启动管理容器**：docker run -p 4002:4002 digu-prod-manager**（指定4002端口访问测试）

## 部署流程

为了保证编译的一致性，可以使用**提测的制品包**，直接部署在生产环境，降低未知的风险。

测试/后端人员导入镜像到生产机：**docker load < ~/digu-prod.tar**

测试/后端人员导入管理后台镜像到生产机：**docker load < ~/digu-prod-manager.tar**

前台生产启动**： docker run -p 4001:4001 --env BACKEND\_URL=http://www.xxxx.com digu-prod**

管理后台生产启动**： docker run -p 4002:4002 --env BACKEND\_URL=http://www.xxxx.com digu-prod-manager**

**BACKEND\_URL：**注入指定服务访问的生产后台API的地址，默认值定义在/server/config.json文件中，外部如果传BACKEND\_URL则使用默认地址。

## 代码托管交付

目前托管在腾讯开发者平台私有项目，如果需要可以把项目权限彻底移交。

# 设计说明

## 

## 选型与升级

**管理后台**基于1.3.1之前版本，仍然使用vue.js2.0 的桌面组件库。访问地址：[ElementUI](http://element.eleme.io/#/zh-CN/component/layout)。

**flow话术编辑器**使用[mxgraph](https://jgraph.github.io/mxgraph/) + vue，1.3.1之前话术编辑器是不走前端编译，1.3.1之后使用yarn去管理前端依赖，并使用[parcel](https://parceljs.org/)进行前端编译。

1.3.1 后新代码不再使用JQuery作为ajax client，选型使用**axios**。

由于项目逻辑比较繁杂，异步代码全面使用**async / await function**。

## 主要的样式文件

1.3.1UI使用了

/front-source/src/static/css//element-ui/lib/theme-chalk/**index.css** 作为默认主题  
 /front-source/src/static/css/**main-blue.less** 与

/front-source/src/static/css/**override-element-ui.less** 作为css基础皮肤样式，

其他冗余多余基础样式**没有在使用，**后期可以清理。

## 反向代理/node中间层设计

**反向代理服务/node中间层**使用 Koa2.7+ 作web服务，目前**反向代理服务**中的业务逻辑代码较少，后期可以将一些与业务核心逻辑无关的业务代码实现在**node中间层**。

目前中间层的功能如下：

1. **开发环境**支持跨域。（route.js:29）
2. **开发环境**flow前端页面的访问的承载，包含前台和管理后台。（route.js:49）
3. **开发环境**全局变量的动态注入。（route.js:28）
4. **生产环境**dist静态资源的转发。（route.js:69）
5. 静态资源下载服务，如word/excel等。（route.js:94）
6. 后端api的proxy。（route.js:97）
7. **controllers/services** 层目前没有太多的逻辑，后期可以考虑将登陆态或其他非核心功功能实现在nodejs中。

### 中间件依赖

1. koa-better-http-proxy，koa-proxy —— 后端api的代理/开发环境dev server代理。
2. koa-send —— dist 静态资源返回
3. koa-router —— pathname路由
4. koa-cors —— 开发环境跨域实现
5. koaService —— service注入的中间件，可以在controller中ctx.service.TestService.homeData()形式调用Service方法
6. allowedMethods —— http method的控制， 若只监听了post方法，则用GET方法不被允许。

## 

## 组件设计与提炼

### 音频播放器

1.3.1重写了**音频播放器**，代码位置/front-source/src/ng-Audio，可作为单独的组件独立调用，适用与比较多的场景。

组件支持功能：

1. 音量的提升/降低 (updateVolume)。
2. 音频文件的直接下载(download)。
3. 暂停(pause)，播放(play)，停止(pause)，循环播放的开关(changeLoop)。
4. 设置直接跳转到目标的播放的进度(setPosition)。
5. 可以在播放中将当前播放的音频进行热替换(asyncFile)。
6. 支持多个文件的连续播放（asyncFileList）。

### 列表匹配模式

1.3.1的**列表匹配模式**的需求比较庞大，所以单独作为一个spa应用进行维护，代码位置/front-source/src/ng-flow。

由于**列表匹配模式**要与**话术编辑器**进行同步，所以应用了将**列表匹配模式** 作为Iframe嵌入在**话术编辑器**中，使用**postMessageBridge.js**进行通信桥接与数据同步。话术编辑器的处理文件位置/front-source/js/postMessageBridge.js

富音频编辑器使用组件模式，可以拓展音频文本框内的组件与交互逻辑，代码位置/ng-flow/SpeechRichEditor/elements 。

前端使用**xlsx**库，实现前端的**xlsx**的动态的导入与导出/ng-flow/lib/excel.js。

话术数据与音频库会轮训检查数据是否有变动（App.vue:93），十分钟自动保存与话术编辑器同步（App.vue:88）。

列表匹配模式的代码完全基于vue组件分层设计，结构参考如下：

├── App.vue vue root

│

├── components 列表配对模式组件文件夹

│ ├── Drawer

│ │ └── index.vue 右侧的抽屉，主要用于访问音频库

│ │

│ ├── Header

│ │ └── index.vue 头部组件

│ │

│ ├── Main

│ │ └── index.vue 页面的main容器

│ │

│ ├── NodeList

│ │ └── index.vue 话术左侧节点列表

│ │

│ ├── NodeTable

│ │ ├── index.vue 话术表格

│ │ └── TipsControl.vue 鼠标点击单元格后添加和删除的气泡

│ │

│ ├── SpeechCraftModal

│ │ └── index.vue 话术详细信息修改的弹窗

│ │

│ └── SpeechRichEditor

│ ├── elements 富音频话术编辑器中的元素组件

│ │ ├── AudioTag.vue 音频节点组件

│ │ └── TextTag.vue 文本节点组件

│ ├── index.vue 富音频话术编辑器

│ └── utils.js 富音频话术编辑器工具函数

│

├── const.js 列表匹配模式内常量的定义

│

├── images

│ └── icons

│ ├── flow.png

│ └── list.png

│

├── lib 代码工具库

│ ├── excel.js 表格的导入和导出

│ ├── nodeHandler.js 话术编辑器xml格式与列表配对模式json格式相互转

│ ├── postMessageBridge.js postMessage，用于和话术编辑器的通讯

│ └── utils.js

│

├── main.js 主入口

│

└── mockData 模拟mock数据

├── mockData2.js

└── mockData.js

#### 列表匹配模式主要模块功能

**Drawer**

主要功能：右侧的抽屉，主要是用来检索音频库功能，支持网左侧talbe中进行音频的托拽加入。

主要功能函数:

fetchAudioGroupList() 拉取话术音频的分类

audioFilesListLoad() 加载音频文件列表

audioFileDragStart() 鼠标拖拽的消息传递

音频节点element被设置为可拖拽元素（draggable="true"），拖拽事件生成时，会把当前的音频的 id/fileName/soundPath/soundContent/duration 等信息json后传入拖拽事件的dataTransfer。在富音频文本编辑器中（SpeechRichEditor/index.vue） @dragover="dragOver"来接收此拖拽事件，实现拖拽音频节点添加。

抽屉的展开与关闭由 **drawerClosed** 变量控制。

**Header**

主要功能：列表匹配模式的顶部的头组件，主要是展示话术名称，承载话术信息修改的modal，展示最经保存的时间。

**NodeList**

主要功能：话术编辑器中的每一个节点，会在nodelist中列表展示，点击nodelist中的节点名称，列换左侧的话术表格。

**NodeTable**

主要功能：列表配对模式表格的主体，包含了当前节点下，话术流程的几种状态分话术的配置，可以动态**添加**与**修改**，保存后可以话术编辑器的xml数据同步。

主要的子组件：

<SpeechRichEditor>：富音频文本编辑器

<Audio>：1.3.1新音频播放器

<TipsControl>：动态添加删除分话术的气泡

主要功能函数:

asyncAudioHandler：异步的音频加载

getTestListenFiles：获取试听文件的地址  
speechRichEditorInput：当前编辑器修改输入时，暂停音频播放  
clickTipsControl：点击弹出TipsControl

deleteSpeech：删除分话术

appendSpeech：新增分话术

**SpeechCraftModal**

主要功能：点击Header组件中的话术信息修改按钮弹出的弹窗组件，修改话术名称，修改方案类型，复制appKey，在这里的修改操作会联动改变流程图视图中的数据。

**SpeechRichEditor**

主要功能：

富音频文本编辑器的文本框，文本框内遍历分析上层传来的data.value **动态创建子组件**，子组件可以理解为单个话术语句中的节点组件，放置在./elements文件夹中，目前有<AudioTag> 音频组件与<TextTag> 文本组件。

通过dropHandler() 接收**Drawer**传递来的拖拽事件，动态加入音频节点。

主要功能函数:

dropHandler/dragLeave/dragOver：处理拖拽事件

pastedHandler：处理黏贴剪贴板事件，主要是过滤html元素，为了防止把其他html元素直接复制在富文本编辑框中。

appendAudioTag：向编辑器推入音频节点  
appendTextTag：向编辑器中推入文本节点

#### 列表匹配模式与话术编辑器PostMessage通讯

主要Api:

flow.set()/flow.get()：从话术编辑器设置与获取话术数据。

backToFlow()：回退到话术编辑器试图。

handleArtificialChange()：处理话术允许挂机转人工（）  
handleArtificialIntentionChange()：切换人工意图类型

mxResources.get：mxResources的获取

# 会话列表1.3.1更新

代码位置/front-source/src/page/Dialogue/

.

├── DialogueDesc.vue 会话详细信息

└── Dialogue.vue 会话列表

会话列表详情页使用了新开发的音频播放器组件，实现了双通道的实时切换，channel.seat与channel.client的enable控制。

# 话术列表1.3.1更新

代码位置/front-source/src/page/Dialogue/

.

├── components

│ └── MySpeechCopyModal.vue 复制话术的弹窗逻辑

├── rules.js 表单的验证规则

└── Speech.vue 话术列表vue主文件

# 知识库主要更新

音频匹配弹窗：/src/page/base/AudioMatchModal.vue

批量修改知识库问答：/src/page/base/BatchEditFiles.vue

批量上传知识库问答：/src/page/base/BatchFiles.vue

知识库列表UI更新：/src/page/base/RepositoryList.vue

知识库UI更新右侧抽屉打开详情：/src/page/base/KnowledgeConfig.vue

# 外呼设置

代码位置/front-source/src/page/Task/

.

├── TaskCheck.vue 任务质检

├── TaskConfig.vue 外呼设置，新增了重呼策略的复杂配置

├── TaskDesc.vue 外呼详情

├── TaskResultRecognition.vue 识别详情

└── Task.vue 外呼列表

其中**TaskConfig.vue**为外呼设置，主要更新了外呼设置新增了话术生效范围等功能，

新增了重呼策略界面。

# 话术编辑器1.3.2更新

话术编辑器（flow）主要有两部分组成

* 流程图，使用mxgraph实现。
* 编辑表单和右侧抽屉，使用vue实现。

1.3.2 对vue相关代码做了重构，为了更好的扩展性，加入依赖管理和前端编译。主要变化有：

* 开发环境，启动dev server，地址为 localhost:1235，由上层node服务代理转发。命令为 parcel index.html —port 1235
* 生产环境，编译后被放入dist目录，与 mxgraph 相关代码类似，命令为 parcel build index.html

以上命令已经封装进 npm script（/front-source/flow/package.json） 中，开发者无需了解细节即可进行开发。

本次迭代涉及重构的代码在 /front-source/flow/neojs 目录

.

├── App.vue // 抽屉表单集合

├── Topbar.vue // 顶部悬浮组件

├── components

│ ├── DgCheckboxCollapseItem.vue // 折叠组件

│ ├── DgFormItem.vue

│ ├── IntentIonEditor // 意图编辑器

│ ├── KnowledgeForm // 知识库设置表单

│ ├── SpeechEditor // 话术编辑器

│ ├── StrategyField // 知识库返回策略

│ ├── TopbarTool // 顶部工具栏

│ ├── cascader // 联级选择器

│ ├── cascader-panel

│ ├── scrollbar // 自定义滚动条

│ └── tag // 标签

└── index.js // 入口文件，vue 初始化