三化物联网平台

概要设计

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 版本修订记录 | | | | |
| 编号 | 日期 | 版本 | 修订人 | 修 订 内 容 |
| 1 | 2018.4.9 | 1.0 | 陈承 | 统稿 |
| 2 | 2018.4.20 | 1.1 | 王洪润 | 修改物接入模块设计架构（第一页）。  物接入模块说明修改（第三页）。  将rpc控制设备放入物接入模块；增加App Actor，Tenant Actor和Device Actor，将session Actor把数据入库改为由Device Actor入库，并增加将数据存入消息队列的方法。 |
| 3 | 2018.7.13 | 2.0 | 李尤 | 更新各个模块 |
| 4 | 2018.7.14 | 2.1 | 李尤 | 添加3D概要设计 |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

目录

[1. 物接入模块概要设计 1](#_Toc519341694)

[1.1 模块简介 1](#_Toc519341695)

[1.2 模块架构图 1](#_Toc519341696)

[1.3 模块要点说明 2](#_Toc519341697)

[1.3.1 协议选择： 2](#_Toc519341698)

[1.3.2 通信框架选择 2](#_Toc519341699)

[1.3.3 Akka 3](#_Toc519341700)

[1.3.4 Spring 4](#_Toc519341701)

[1.4 数据库设计 4](#_Toc519341702)

[2. 规则管理模块概要设计 6](#_Toc519341703)

[2.1 模块简介 6](#_Toc519341704)

[2.2 模块架构图 7](#_Toc519341705)

[2.3 模块要点说明 7](#_Toc519341706)

[2.3.1 AppActor 7](#_Toc519341707)

[2.3.2 TenantActor 8](#_Toc519341708)

[2.3.3 RuleActor与RuleActorMessageProcessor 8](#_Toc519341709)

[2.3.4 规则处理流程 8](#_Toc519341710)

[2.4 数据库设计 9](#_Toc519341711)

[3. 插件管理模块概要设计 10](#_Toc519341712)

[3.1 模块简介 10](#_Toc519341713)

[3.2 模块架构图 10](#_Toc519341714)

[3.3 模块要点说明 10](#_Toc519341715)

[3.3.1 Controller-Data-Service 10](#_Toc519341716)

[3.3.2 Aop切面编程 11](#_Toc519341717)

[3.3.3 PluginRegistry 11](#_Toc519341718)

[3.3.4 Metrics 11](#_Toc519341719)

[3.3.5 插件处理流程 12](#_Toc519341720)

[3.4 数据库设计 12](#_Toc519341721)

[4. 服务管理模块概要设计 13](#_Toc519341722)

[4.1 模块简介 13](#_Toc519341723)

[4.2 模块架构图 13](#_Toc519341724)

[4.3 模块要点说明 14](#_Toc519341725)

[4.3.1 对外接口层 14](#_Toc519341726)

[4.3.2 ServiceContext 14](#_Toc519341727)

[4.3.3 Manufacturer、DeviceType、Model 14](#_Toc519341728)

[4.3.4 精确查询流程 14](#_Toc519341729)

[4.3.5 模糊查找流程 14](#_Toc519341730)

[4.4 数据库设计 14](#_Toc519341731)

[5. 消息中间件概要设计 16](#_Toc519341732)

[5.1 模块简介 16](#_Toc519341733)

[5.2 模块架构图 16](#_Toc519341734)

[5.3 模块要点说明 17](#_Toc519341735)

[5.3.1 创建主题 17](#_Toc519341736)

[5.3.2 发送消息 17](#_Toc519341737)

[5.3.3 启动消费者 17](#_Toc519341738)

[5.3.4 设置多代理集群 17](#_Toc519341739)

[5.3.5 使用Kafka Connect导入/导出数据 18](#_Toc519341740)

[5.3.6 使用Kafka Streams处理数据 18](#_Toc519341741)

[5.4 数据库设计 18](#_Toc519341742)

[6. 统一日志中心概要设计 19](#_Toc519341743)

[6.1 模块简介 19](#_Toc519341744)

[6.2 模块架构图 19](#_Toc519341745)

[6.3 模块要点说明 20](#_Toc519341746)

[6.3.1 收集和写入日志 20](#_Toc519341747)

[6.3.2 查询和删除日志 20](#_Toc519341748)

[6.3.3 可视化日志数据 21](#_Toc519341749)

[6.4 数据库设计 21](#_Toc519341750)

[7. 统一认证鉴权概要设计 22](#_Toc519341751)

[7.1 模块简介 22](#_Toc519341752)

[7.2 模块架构图 22](#_Toc519341753)

[7.3 模块要点说明 24](#_Toc519341754)

[7.4 数据库设计 28](#_Toc519341755)

[8. 统一配置中心概要设计 30](#_Toc519341756)

[8.1 模块简介 30](#_Toc519341757)

[8.2 模块架构图 31](#_Toc519341758)

[8.3 模块要点说明 32](#_Toc519341759)

[8.4 数据库设计 33](#_Toc519341760)

[9. 3DwebGIS平台概要设计 34](#_Toc519341761)

[9.1 模块简介 34](#_Toc519341762)

[9.2 模块架构图 35](#_Toc519341773)

[9.2.1 整体架构图 35](#_Toc519341774)

[9.2.2 系统动态交互图 35](#_Toc519341775)

[9.3 模块要点说明 36](#_Toc519341776)

[9.3.1 webGL渲染引擎 36](#_Toc519341777)

[9.3.2 模型坐标系 38](#_Toc519341778)

[9.3.3 3D模型管理 38](#_Toc519341779)

[9.3.4 通过3D模型展示设备详细信息 39](#_Toc519341780)

[9.3.5 地图界面 40](#_Toc519341781)

[9.3.6 设备列表 41](#_Toc519341782)

[9.4 数据库设计 42](#_Toc519341783)

# 物接入模块概要设计

## 模块简介

物接入模块是设备接入并能够实现设备数据上传的模块，该模块的关键是使得模块成功与设备连接并且通过该模块获取设备数据，并将设备获得的数据成功的存入数据库。将接入平台的设备进行增、删、改、查、分组、分配等操作也是该模块的功能之一。

模块的难点在于如何在设备数量增加的情况下任然能够稳定的提供服务，基于此目标我们提出了netty+akka的物接入解决方案。

使用netty框架保证通过mqtt协议接入的设备成功的与模块连接，并且运用akka框架中的actor模型来使得其保证高并发性，通过传递消息（msg）保证actor之间的数据传递，最后在Device Actor中运用savaTsData方法将设备获取的设数据存入 NoSQL（cassandra）数据库和消息队列。Device Actor会收到从客户发过来到RPC REQUEST 请求，通过RPC Response来控制设备的一些操作。

对设备的操作采用SpringBoot，采用MVC模式进行设计，提供给前端的接口调用，包括：设备操作，设备组操作，设备分配，设备控制。

设备操作：增加设备，删除设备，查找设备，更改设备

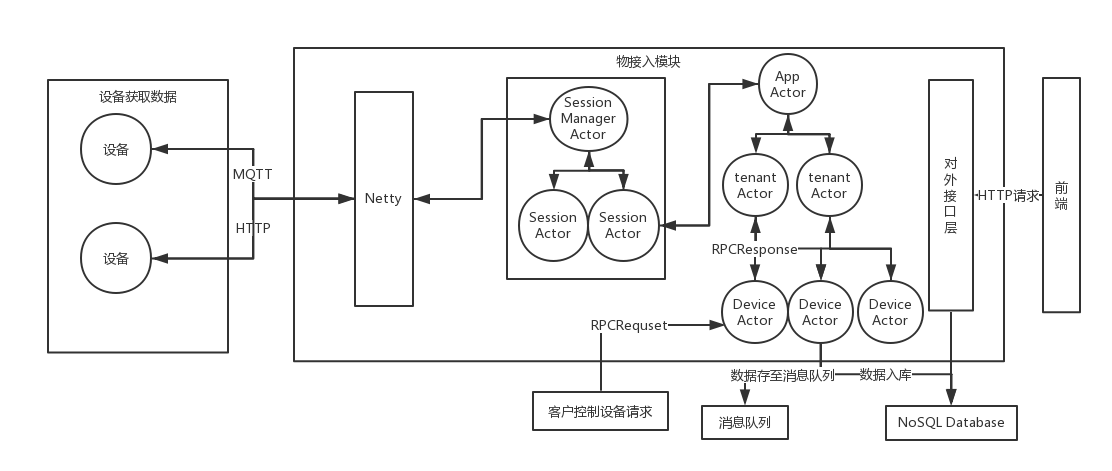
设备数据和属性操作：增加设备数据，查看设备数据，修改设备数据，删除设备数据，增加设备属性，查看设备属性，修改设备属性，删除设备属性

设备组操作：创建设备组，删除设备组，更改设备组，查找设备组，分配设备到设备组，删除设备组中设备，查找设备组中设备

设备分配操作：将设备分配给客户

设备控制操作：控制设备

## 模块架构图



## 模块要点说明

### 协议选择：

在广泛调研了现有的物联网应用层协议之后，物接入层我们拟采用mqtt协议加http协议两种实现方式，设备可通过这两种方式接入云平台。

### 通信框架选择

在通信框架的选择上，我们选择了开源nio通信框架，netty，该协议作为一个底层通信框架在许多开源框架中都有用到，是值得信赖的一款通信框架。基于netty的编程模型，我们设计了以下几个主要的业务逻辑相关类。

（1）TransportServerInitializer：

该类的功能是初始化Channel

方法：initChannel()

（2）TransportHandler类。

该类的功能是处理业务逻辑。

方法：channelRead() 读取信息操作,通过sessionmanageractor.tell()方法进入akka

ChannelReadComplete() 读完信息操作

ExceptionCaught() 异常捕获操作

（3）Decode类

该类将字节解码为消息

方法：mqttdecode() 读取字节数据将其解码为mqtt消息

（4）Encode类

该类将消息编码为字节

方法：mqttencode() 将mqtt消息编码为字节信息

（5）DeviceSessionCtx类

该类功能是控制当前会话期间的内容，如获取sessionId 判断超时、判断会话是否关闭等操作。

### Akka

akka是一款并发编程框架，提供基于异步消息处理的方式来保证高并发性。基于其actor编程模型，我们设计了SessionManagerActor和SessionActor。SessionManagerActor 管理所有的SessionActor，是akka系统的入口每一个SessionActor代表了设备与平台建立的会话。底层的netty接收到数据之后通过SessionManagerActor.tell()方法将数据传递到akka的actor系统中，接收数据后再将数据传递给SessionActor。当第一条有相应的sessionId出现的消息发送时创建SessionActor，相应的session关闭时session Actor也会关闭。

Session Manager Actor 会话管理actor，管理Session Actor。

方法：onReceive() 处理收到的消息

ForwardToSessionActor() 创建SessionActor并将消息传递给SessionActor

Session Actor 会话actor，代表一个设备与服务器的的会话，它可以是同步的，也可以是异步的。

方法：onReceive() 处理收到的消息

AppActor.tell() 发消息给AppActor

APP Actor：负责管理租户

方法：OnReceive() 处理消息

GetorCreateTenantActor() 获取或创建tenant Actor

Tenant Actor： 负责管理租户设备

方法： OnReceive() 处理消息

GetorCreateDeviceActor() 获取或创建Device Actor

Device Actor： 维护设备状态，当来自设备的第一条消息被处理时，创建一个actor，当一定时间内没有来自设备的消息时，其会关闭。

方法： OnReceive() 处理消息

ProcessRPC() PRC操作

ProcessDB() 数据库操作

ProcessKafka 实时数据存至消息队列

### Spring

Controller层：提供对外接口，前端可以调用该层的接口获取数据

Service层： 编写业务逻辑，业务层次的代码在该层有所体现，并有Controller层调用

Dao层： 数据库层面的操作，在该层实现数据库相关操作，并由Service层调用

## 数据库设计

**ts\_kv\_cf**表：

|  |  |  |
| --- | --- | --- |
| **字段** | **描述** | **数据类型** |
| **device\_Id** | 设备Id | UUID |
| **ts** | 时间戳 | bigint |
| **key** | 键（数据名称） | text |
| **bool\_v** | BOOLEAN数据值 | boolean |
| **str\_v** | STRING数据值 | text |
| **long\_v** | LONG数据值 | bigint |
| **dbl\_v** | DOUBLE数据值 | double |
| **partition** | 数据分区 | bigint |

**Attributes\_kv\_cf**表：

|  |  |  |
| --- | --- | --- |
| **字段** | **描述** | **数据类型** |
| **entity\_Id** | 设备Id | UUID |
| **last\_update\_ts** | 时间戳 | bigint |
| **attribute\_key** | 键（数据名称） | text |
| **bool\_v** | BOOLEAN数据值 | boolean |
| **str\_v** | STRING数据值 | text |
| **long\_v** | LONG数据值 | bigint |
| **dbl\_v** | DOUBLE数据值 | double |

**device**表：

|  |  |  |
| --- | --- | --- |
| **字段** | **描述** | **数据类型** |
| **id** | 设备Id | UUID |
| **tenant\_Id** | 租户Id | int |
| **customer\_Id** | 客户Id | Int |
| **name** | 设备名称 | text |
| **additional\_info** | 附加信息 | text |
| **manufacture** | 设备类型 | text |
| **device\_type** | 厂商 | text |
| **model** | 型号 | text |
| **location** | 设备位置 | text |
| **Status** | 设备状态 | tinyint |
| **Search\_text** | 搜索框 | text |
| **parent\_device\_id** | 父设备Id | text |
| **site\_id** | 站点Id | text |
| **life\_time** | 维修年限 | bigint |

**device\_credatials**表**：**

|  |  |  |
| --- | --- | --- |
| **字段** | **描述** | **数据类型** |
| **id** | Id | int |
| **device\_Id** | 设备id | int |
| **tocken** | 设备访问令牌 | varchar |

# 规则管理模块概要设计

## 模块简介

规则模块是PaaS层对系统以及设备上传消息提供处理，信息甄别以及事件响应的功能模块，针对了消息处理及响应模式，设计了以规则管理，规则处理，消息过滤，插件行为动作的一系列消息处理体系。模块中有规则获取，规则处理，并行处理，消息过滤，调用插件概念，下面做一下大致介绍。

规则获取：从数据库中获取系统内所有规则以及租户自定义规则的方法。

规则处理：生成各个规则的Actor，将消息同时丢入这些规则Actor，根据各个规则进行处理。

并行处理：各个规则Actor同时接收消息，各个Actor实现并行的消息处理方式。

消息过滤：判断该消息是否符合规则条件，符合的则继续处理。

调用插件：对符合的消息的数据，使用插件来进行处理，插件包含如警报插件，邮件插件等……

## 模块架构图



## 模块要点说明

### AppActor

按创建规则的租户建立TenantActor，并将受到设备数据按租户Id转发至各自的TenantActor：

AppActor()

初始化规则Actor的构造函数，声明tenantActors的Hashmap实例，用于将TenantActor与租户ID进行绑定。

onReceive()

akka中监听消息接受后对消息进行操作的方法，在AppActor中将消息的租户Id在HashMap中进行查找，若存在则转发至相应的TenantActor,若不存在则创建TenantActor并转发。

### TenantActor

对规则的状态进行判断，如果是激活状态的则创建并转发消息到RuleActor：

TenantActor()

初始化规则Actor的构造函数，在TenantActor创建的时候会从数据库中获取所有该租户的规则，并根据规则的状态创建RuleActor

onReceive()

akka中监听消息接受后对消息进行操作的方法，在TenantActor中将受到的消息分别转发至HashMap中存在的所有RuleActor中。

### RuleActor与RuleActorMessageProcessor

对传入的设备数据进行处理的Actor

RuleActor：

初始化规则Actor的构造函数，创建对数据进行处理的processor实例。

onReceive():

akka中监听消息接受后对消息进行操作的方法,在RuleActor中负责对数据进行处理。

nashornProcess()

对过滤器的Js代码进行解析运行，判断数据是否符合过滤器条件。

sendHTTPPOSTRequest()

通过过滤器过滤的数据想插件发送http请求完成，调用插件执行操作。

### 规则处理流程

整个规则处理流程分为2个部分，在第一部分中，从消息队列中订阅到的数据经过AppActor、TenantActor对数据的需要租户信息和规则信息进行甄别，并传到RuleActor。在第二部分的中，RuleActor对数据使用规则定义的过滤器进行过滤，通过过滤器过滤的信息，通过发送HTTP请求的方式调用插件来完成各种操作。

## 数据库设计

Rule表：

|  |  |  |
| --- | --- | --- |
| **字段** | **描述** | **数据类型** |
| **ruleId** | 规则ID | INT |
| **tenantId** | 规则所属租户ID | INT |
| **addition\_info** | 备注信息 | Varchar |
| **name** | 规则名 | Varchar |
| **state** | 规则当前状态 | Varchar |

Filter表

|  |  |  |
| --- | --- | --- |
| **字段** | **描述** | **数据类型** |
| **filterId** | 过滤器ID | INT |
| **Type** | 过滤器类型 | Varchar |
| **jsCode** | Js代码 | Varchar |
| **name** | 过滤器名称 | Varchar |

rule2Filter\_Relation表

|  |  |  |
| --- | --- | --- |
| **字段** | **描述** | **数据类型** |
| **filterId** | 过滤器ID | INT |
| **ruleId** | 规则ID | INT |

Transform表

|  |  |  |
| --- | --- | --- |
| **字段** | **描述** | **数据类型** |
| **transformId** | 插件的ID | INT |
| **requestBody** | 请求体 | Varchar |
| **url** | 请求的URL | Varchar |
| **Name** | 插件名称 | Varchar |
| **Method** | 请求方法 | Varchar |

rule2TransForm\_Relation表

|  |  |  |
| --- | --- | --- |
| **字段** | **描述** | **数据类型** |
| **filterId** | 过滤器ID | INT |
| **transformId** | 插件ID | INT |

# 插件管理模块概要设计

## 模块简介

插件模块是PaaS层对系统实现提供基础服务功能支持的具有高扩展性的模块。针对现代物联网系统中的基础功能以及功能实现方式，本方案中插件以微服务的形式，各个插件之间相互独立，插件与插件之间毫无关联，对一个插件的操作不会影响到其他插件模块的运行。

插件包含的基本功能有以下4点：

插件信息获取：插件信息在插件部署的时候在zookeeper进行注册，将插件的名字，地址，描述等信息进行注册。 管理模块通过从zookeeper的注册节点获取插件信息。

插件管理：插件具有基本激活暂停的管理操作功能，可以通过运维人员部署服务以及移除服务的方式来新增和移除插件。

插件调用：通过调用插件的Restful接口，发送HTTP请求的方式调用插件。

插件处理：执行基础服务功能，例如发邮件，发警报等功能。

## 模块架构图



## 模块要点说明

### Controller-Data-Service

用于执行插件模块基础处理工作的流程

Controller

提供被其他模块进行调用的接口，接受传入的数据。

Data

将传入的数据进行打包解析工作

Service

对数据进行插件功能操作的部分，例如根据数据发送指定地址、主题和内容的邮件操作等。

### Aop切面编程

用于在插件插件被调用之前先行判断插件当前状态。

@ConfirmActive

通过ConfirmActive注解在插件被调用前进入切面编程。

before()

通过切面编程的before方法，在插件接口被调用前先判断当前插件的状态是否处于激活状态。若插件状态为暂停，则插件接口不可被调用。

### PluginRegistry

用于将插件的信息注册进入zookeeper的部分。

setApplicationContext()

得到所有RpcService注解的SpringBean，从注解中获取插件的注册信息，并建立插件注册实例。

PluginRegistry()

插件注册的构造函数，在此构造函数中完成模块与zookeerper的连接。

register()

将插件信息注册进入zookeeper的Registry节点中。

### Metrics

用于对插件服务的使用情况信息进行统计，可查看插件被调用的次数等信息。

setMetrics()

设置metrics的基本参数属性，对metrics进行初始化。

pendingJobs.inc()

Metrics中的计数器，当插件模块被调用一次后，则计数器+1

### 插件处理流程

插件模块的处理流程主要分为初始化部分和信息处理部分。当一个插件服务在服务器中部署时候，插件模块与zookeeper建立连接，并将插件信息注册进zookeeper，其余模块通过zookeeper获取插件的注册信息包括插件的地址和接口，这是插件的初始化部分处理。在处理部分中，其他模块通过插件的地址和接口调用插件。Aop切面编程检查当前插件状态是否激活，若插件处于激活状态，则Metrics中的计数器+1，并执行插件操作，如发送邮件，发送HTTP请求等。

## 数据库设计

Update\_message（推送消息插件表）

|  |  |  |
| --- | --- | --- |
| **字段** | **描述** | **数据类型** |
| **id** | 唯一自增Id | int |
| **tenant\_id** | 所属租户Id | int |
| **message** | 消息内容 | Varchar |
| **messageType** | 消息类型 | Varchar |
| **ts** | 时间戳 | bigint |

# 服务管理模块概要设计

## 模块简介

服务管理模块是PaaS层对系统设备所能提供的服务进行抽象集中管理的模块，针对物联网设备提供服务的特点，设计了以（厂商、设备类型、设备型号）三元组为核心的服务管理系统。系统中有服务组、服务、精确查询、模糊查询等概念，下面做一个大致介绍。

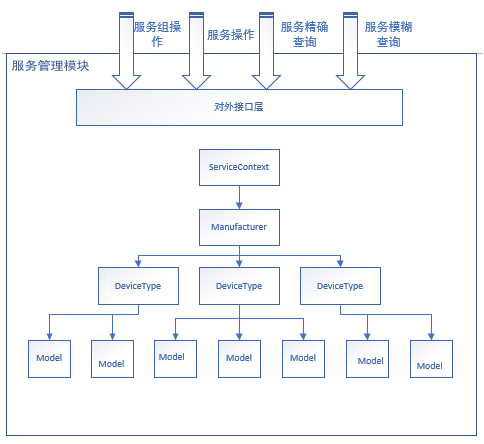
服务组由一系列服务组合而成，一个服务组与一个（厂商、设备类型、设备型号）三元组一一对应。

服务是服务组的组成单位，代表了由（厂商、设备类型、设备型号）三元组所定位到的一种具体设备所能支持的某中操作。

精确查询：指定一个三元组的，返回其所能支持的操作

模糊查询：指定关键字，返回该关键字下的一系列设备。

## 模块架构图



## 模块要点说明

### 对外接口层

拟采用springboot技术对外提供restful接口以支撑各种操作。持久化层拟采用msql数据库，将服务相关信息存储在数据库中

### ServiceContext

服务系统的对外接口，具体到代码层面表现为一个接口，对外提供一些基本的方法，全局保证一个唯一的实例。对外接口层调用此实例的方法访问服务系统提供的功能，比如服务组的增删改查，服务的增删改查，服务查询等等。

### Manufacturer、DeviceType、Model

三层构成一个树型层级结构，每一层中的节点包含下一层的引用，利用hash算法加速查找，使得总体的时间复杂度为常数。Model节点包含具体的挂载在该服务组下的设备信息以供实时查询。

### 精确查询流程

指定设备三元组以及设备唯一标识符，通过serviceContext接口进入服务系统，在树形结构中逐层查找，定位到具体设备。

### 模糊查找流程

利用全文检索开源实现elasticsearch，将一个服务组作为一个文档，文档中所有字段的关键字作为检索关键字来实现模糊查询。模糊查询分为两个阶段，第一个阶段为服务组定位阶段，第二个阶段为设备定位阶段

## 数据库设计

manufacture**r**表：

|  |  |  |
| --- | --- | --- |
| 字段 | 描述 | 数据类型 |
| manufacturer\_id | 唯一自增Id | Int |
| manufacturer\_name | 厂商名称 | Varchar |

deviceType表：

|  |  |  |
| --- | --- | --- |
| 字段 | 描述 | 数据类型 |
| deviceType\_id | 唯一自增Id | Int |
| manufacturer\_id | 设备i类型所属厂商Id | Int |
| deviceType\_name | 设备类型名称 | Varchar |

model表：

|  |  |  |
| --- | --- | --- |
| 字段 | 描述 | 数据类型 |
| model\_id | 唯一自增Id | Int |
| deviceType\_id | 型号所属设备Id | Int |
| manufacturer\_id | 设备类型所属厂商Id | Int |
| model\_name | 型号名称 | Varchar |

service表：

|  |  |  |
| --- | --- | --- |
| 字段 | 描述 | 数据类型 |
| service\_id | 唯一自增Id | Int |
| model\_id | 服务组Id | Int |
| service\_des | 服务描述 | Varchar |

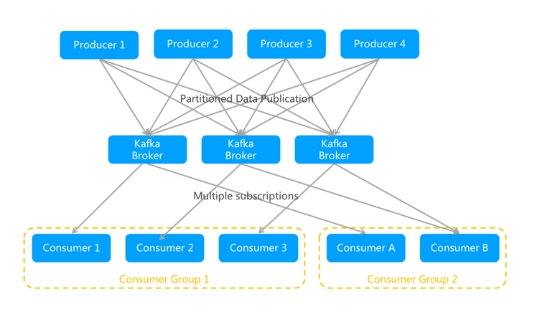
# 消息中间件概要设计

## 模块简介

消息中间件是一种由消息传送机制或消息队列模式组成的中间件技术，包含发布消息接口和订阅消息接口，支持使用发布/订阅模式实现消息的传送和处理。我们选择Kafka作为本项目的消息队列（MQ）。物联网项目的数据量大，对吞吐量的要求高，作为一种追求高吞吐量的分布式发布/订阅消息系统，Kafka适用于产生大量数据的互联网服务的数据收集业务。除此之外，Kafka分布式架构的高可用性和便捷的扩展性更加符合要求。

在Kafka系统中，消息的生产者（Producer）可以发送消息到Kafka的消息处理服务器（Broker，包含一个或者多个服务器的Kafka集群），消息的消费者（Consumer）可以从服务器中获取消息。Kafka可以通过Java客户端进行访问。同样，本模块基本具备这两项功能，可以让其他模块利用本模块的API创建生产者和消费者进行消息的发布或获取，并通过实体类对象封装消息。

## 模块架构图



## 模块要点说明

### 创建主题

可以使用分区和副本创建主题，创建完成后可以通过运行list topic命令查看该主题。另一种方法是将代理配置为在发布不存在的主题时自动创建主题，而不是手动创建主题。

### 发送消息

Kafka带有一个命令行客户端，可以从文件或标准输入中获取输入，并将其作为消息发送到Kafka集群。在默认情况下，每行将作为单独的消息发送。

可以通过运行生产者并在控制台中输入语句将消息发送到服务器。

### 启动消费者

Kafka也有一个命令行消费者，将消息转储到标准输出。

将每个命令在不同的终端中运行，能够将消息键入生产者终端，并将它们显示在消费者终端中。

所有的命令行工具都有其他选项，在没有参数的情况下运行该命令将显示更详细地记录它们的使用信息。

### 设置多代理集群

可以将集群扩展为多个节点，但仍然全部在本机上。

首先为每个broker创建一个配置文件，编辑这些新文件并设置属性。broker.id属性是集群中每个节点的唯一且永久的名称。我们只需要覆盖端口和日志目录，因为这些端口和日志目录在同一台机器上运行，并希望避免所有broker都试图在同一个端口上注册或覆盖彼此的数据。

在启动新节点并创建新主题后，为查看每个broker的行为，可以运行“describe topics”命令，输出的第一行给出了所有分区的摘要，每个附加行提供了有关一个分区的信息。“leader（领导者）”是负责给定分区的所有读写操作的节点，每个节点将成为分区随机选择部分的领导者；“replicas（副本）”是复制此分区的日志的节点列表，无论它们是领导者还是目前都正在运行；“isr”是一组“同步”副本，是副本列表的子集，目前正在运行并被引导到领导者身边。

### 使用Kafka Connect导入/导出数据

从控制台写入数据并将其写回控制台是一个方便的开始，但可能需要使用其他来源的数据或将数据从Kafka导出到其他系统。对于许多系统，可以使用Kafka Connect导入或导出数据，而不是编写自定义集成代码。

Kafka Connect是Kafka附带的一个工具，可以将数据导入和导出到Kafka。它是一个可扩展的工具，运行连接器实现与外部系统交互的自定义​​逻辑。这些连接器将数据从文件导入Kafka主题，并将数据从Kafka主题导出到文件。

### 使用Kafka Streams处理数据

Kafka Streams是一个用于构建关键任务实时应用和微服务的客户端库，输入和/或输出数据存储在Kafka集群中。Kafka Streams结合了在客户端编写和部署标准Java和Scala应用的简单性以及Kafka服务器端集群技术的优势，使这些应用具有高度的可伸缩性、弹性、容错性、分布式等特性。

## 数据库设计

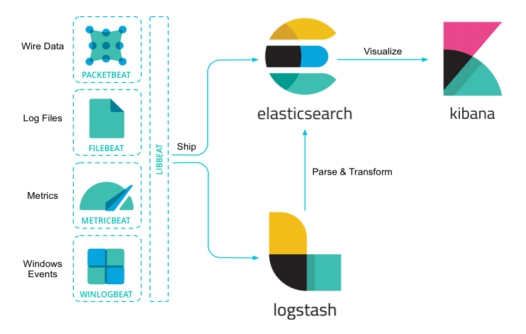
# 统一日志中心概要设计

## 模块简介

统一日志管理模块是对系统平台的日志文档进行集中管理的模块，该模块可以收集并统一存储各业务模块及下属的终端设备产生的日志文件，供用户需要时查阅。我们采用Elastic提供的ELK Stack实现统一日志管理模块。ELK Stack是一个开源日志记录平台，其中ELK是Elasticsearch、Logstash和Kibana三个开源项目的首字母缩写。Elasticsearch是一个分布式的RESTful搜索和分析引擎，允许快速并接近实时地存储、搜索和分析大量数据。Logstash是一款具有实时流水线功能的开源数据收集引擎，可以动态统一来自不同来源的数据，并将数据归一化到特定的目的地。Kibana是一个开源的分析和可视化平台，可以搜索和查看存储在Elasticsearch索引中的数据并与其进行交互。

在本系统平台中，为收集日志并查找数据、汇总异常情况，我们可以使用Logstash来收集和汇总日志，然后使用Logstash将此数据提供给Elasticsearch。一旦数据在Elasticsearch中，即可搜索和挖掘所需的任何信息。

## 模块架构图



## 模块要点说明

### 收集和写入日志

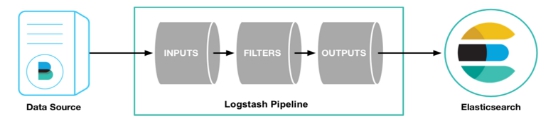
#### Beats

Beats是作为代理安装在服务器上的开源数据发布者，用于向Elasticsearch发送不同类型的操作数据。Beats可以将数据直接发送到Elasticsearch或通过可用于解析和转换数据的Logstash发送到Elasticsearch。

Beats的几个例子包括Packetbeat、Filebeat、Metricbeat和Winlogbeat。Packetbeat是一个网络数据包分析器，传输在应用服务器之间交换的事务的信息。Filebeat从服务器发送日志文件。Metricbeat是一个服务器监控代理，定期从服务器上运行的操作系统和服务收集指标。Winlogbeat发布Windows事件日志。

#### Logstash

Logstash事件处理管道有两个必需元素输入、输出以及一个可选元素过滤器。输入插件消费来自数据源的数据，过滤器插件修改指定的数据，输出插件将数据写入目标。其结构图如下所示：



默认的Logstash安装包含Beats输入插件，这一插件使Logstash能够从Elastic Beats框架接收事件。为此，可以创建一个将Beats插件作为输入的Logstash管道，使用过滤器插件解析进入管道的日志，从日志中创建特定的命名字段，将非结构化日志数据解析为结构化和可查询的内容（如JSON表示形式），并将解析的数据写入Elasticsearch群集。

### 查询和删除日志

Elasticsearch是分布式的对象存储，可以将复杂的数据结构序列化为JSON文档加以存储，并索引每个文档的内容使之可被检索。在Elasticsearch中， 每个字段的所有数据都是默认被索引的。一旦一个对象被存储在Elasticsearch中，它就可以被集群中的任意节点检索到。

为实现对文档的检索，一个文档不仅仅包含它的数据，也包含元数据，即有关文档的信息。通过元数据元素，可以存储文档到Elasticsearch中并通过ID检索它，也即使用Elasticsearch作为文档的存储介质。

三个必须的元数据元素如下：

(1) \_index：表示文档的存放位置。一个索引应该是因共同的特性被分组到一起的文档集合。

(2) \_type：表示文档的对象类别。索引中的数据可能只是松散的组合，但通常需要明确定义一些数据中的子分区。

(3) \_id：文档的唯一标识。ID 是一个字符串，可以与\_index 以及\_type 组合从而唯一确定Elasticsearch中的一个文档。

考虑到使用者的实际需求，为实现对日志文档的查询，我们执行一个HTTP GET请求，并在请求中指定日志文档的来源、时间戳等地址信息。若需要查询符合一定条件的特定日志，例如日志的不同级别，可以使用查询字符串（query-string）搜索、匹配（match）搜索等方式，或使用支持高效地执行一个结构化查询的过滤器（filter）等。

为删除陈旧日志，可以使用DELETE请求并指定相应元数据加以操作。当需要批量地删除大量日志数据时，可以仅仅指定索引名、多个索引名甚至全部索引。删除文档不会立即将文档从磁盘中删除，只是将文档标记为已删除状态。随着索引数据的增加，Elasticsearch将会在后台清理标记为已删除的文档。

### 可视化日志数据

为提升使用日志管理模块的用户体验度，可以将Kibana与Elasticsearch相连接，向用户提供直观的显示界面。为此，需要告知Kibana想要浏览的相应索引，并定义与一个或多个索引名称匹配的索引模式。

在加载数据集之前，需要为日志字段设置映射。映射将索引中的文档分成逻辑组，并指定字段的特征，例如字段的可搜索性，是否被标记化，或分解为单独的单词。之后使用Elasticsearch bulk API加载数据集。数据集加载完毕后，即可在Kibana界面中查看到相应的日志数据。

## 数据库设计

# 统一认证鉴权概要设计

## 模块简介

在一套完整的物联网系统中，总是需要用户通过账户登录系统才能使用物联网系统的相应的功能。而用户数量肯定是不止一个，所以账户系统便是用来管理这些账户的一个体系。

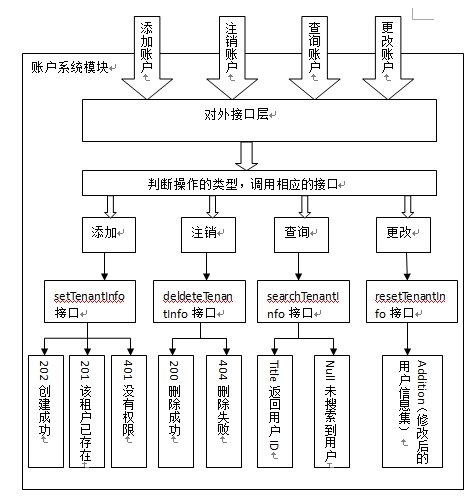
在微服务架构下，一个应用会被拆分成若干个微应用，每个微应用都需要对访问进行鉴权，每个微应用都需要明确当前访问用户以及其权限。尤其当访问来源不只是浏览器，还包括其他服务的调用时，单体应用架构下的鉴权方式就不是特别合适了。在微服务架构下，要考虑外部应用接入的场景、用户–服务的鉴权、服务–服务的鉴权等多种鉴权场景。

如用户A访问账户系统Account Service，A如果未登录，则首先需要登录，请求获取授权token。获取token之后，A将携带着token去请求访问某个文件，这样就需要对A的身份进行校验，确定A是否可以访问该文件。

对于用户来说，进行操作之前首先要明确自己的用户权限，只有在自己的权限范围之内才可以对相应的账户进行操作，因此用户登录之后的首要工作就是进行鉴权。为了方便整个系统的使用方便，把账户系统和认证鉴权模块合并。同时也可以通过调用鉴权模块的接口为其他业务service提供服务。

## 模块架构图

**账户**



在账户系统中，当用户发出对账户系统的操作（添加，删除，修改，查询）时，首先会经过对外接口层，然后对外接口层判断用户的操作类型，进而再调用相应的接口。再调用接口执行操作之后再返回相应的结果。

**认证鉴权**



认证鉴权采用Spring Security框架，请求会首先经过拦截器栈，主要用来防止恶意攻击以及粗略的权限拦截。之后在访问方法前进行Pre拦截，方法访问结束后进行Post拦截。Pre和Post进行真正的权限拦截，它们以注解的形式添加在一个类或者一个方法前。

## 模块要点说明

**账户**

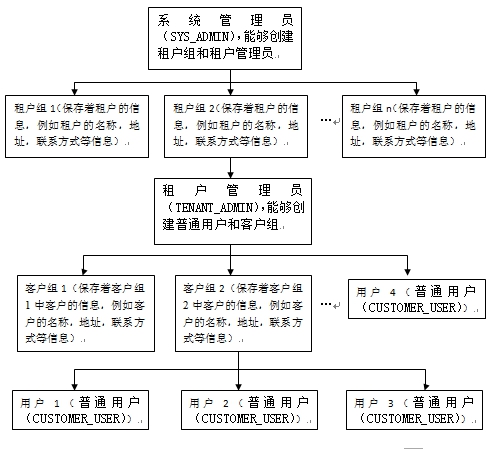
账户系统包括三个部分：租户组（Tenant），客户组（Customer）以及用户（User）。

租户组保存着租户的信息，例如租户的名称，地址，联系方式等信息，是一个类似部门的概念，同时一个租户下包含多个客户以及用户。

客户组保存着客户的信息，例如客户的名称，地址，联系方式等信息，是一个类似部门的概念，同时一个客户下包含多个用户。

用户则是物联网平台的使用者，包含名称，用户权限等属性。其中用户权限又分为系统管理员（SYS\_ADMIN）、租户管理员（TENANT\_ADMIN）和普通用户（CUSTOMER\_USER）。系统管理员不属于任何租户和客户，租户管理员只属于某一个租户而不属于任何客户，普通用户属于某一个租户以及该租户下的某个客户。

**账户系统层级关系图**



**认证与鉴权**

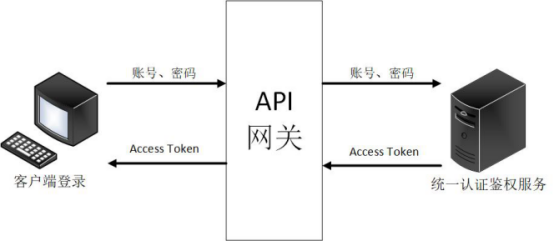
用户发出的请求主要有两种：

1. 用户尚未登录。客户端发起登录请求，网关对于登录请求直接转发到auth服务，auth服务对用户身份信息进行校验。
2. 用户已登录，请求其他服务。这种情况，客户端的请求到达网关，网关会将请求转发到对应的微服务中。由这个微服务确定用户能否调用该接口。

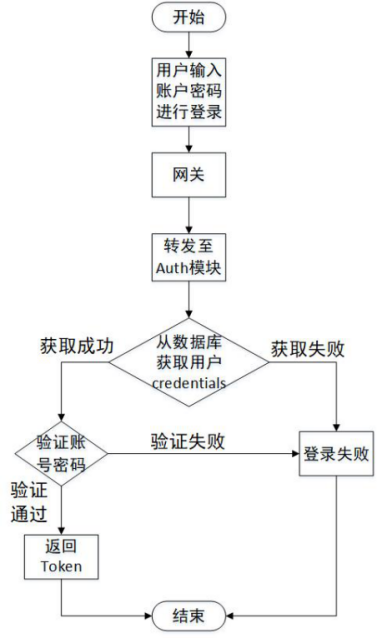
**用户认证**

用户认证部分，主要采用基于OAuth2 Token认证的方式。随着 Restful API、微服务的兴起，基于Token的认证现在已经越来越普遍。Token并非只是一个 key。Token 一般会包含用户的相关信息，通过验证 Token 就可以完成身份校验。

用户输入登录信息，发送到统一认证鉴权服务进行认证。统一认证鉴权服务验证登录信息是否正确，正确则返回包含用户基础信息、权限、有效时间等信息的Access Token。



登录流程图如下所示：

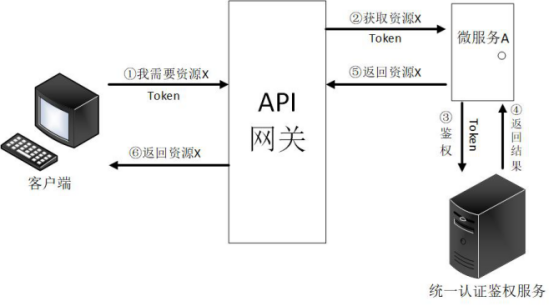


**API级别的权限控制**

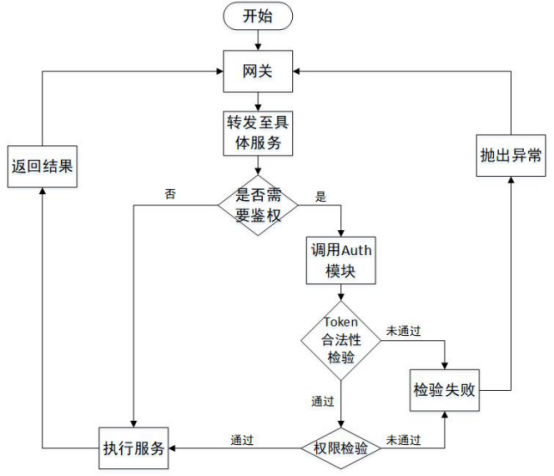
对于操作权限的控制，主要是基于Spring Security框架进行实现。

当鉴定完用户身份合法之后，对于该认证用户，我们还需要对他所进行的操作进行鉴权，以确定该用户是否拥有调用相关API的权限。

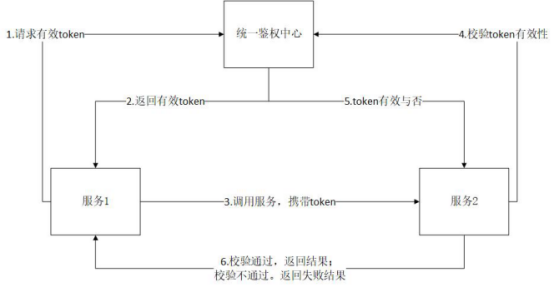
每当用户发出请求时，请求头部都会携带一个Token，该Token会被转发到统一认证鉴权模块进行合法性验证（确定该Token是系统签发）以及鉴权（确定该用户的权限），并将结果返回。各个微服务会根据返回结果来确定该请求的权限是否符合要求。符合要求则执行该服务，否则拒绝执行。



对于部分不需要权限就可以调用的接口（例如用户认证中所调用的登录接口），不需要鉴权就可以直接调用。以下是鉴权的流程图：



当服务间进行接口调用时，也需要进行鉴权。调用服务需要向鉴权中心请求token，并且携带这个token向被调用服务进行请求。被调用服务会进一步向鉴权中心请求鉴权，判定该请求是否应该被响应。



OAuth2根据使用场景不同，分成了4种模式：授权码模式（authorization code）、简化模式（implicit）、密码模式（resource owner password credentials）和客户端模式（client credentials）。对于外部请求，采用密码模式，用户提供正确的账号与密码，以及当前客户端的client id和client secret，验证成功后即可以获得access token，并且携带这个access token进行接口的请求。对于系统内部各个微服务之间的请求，采用客户端模式，发出请求的服务只需要提供client id和client secret，验证成功后就可以获得token并且携带token进行请求。

## 数据库设计

user表

|  |  |  |
| --- | --- | --- |
| 字段 | 描述 | 数据类型 |
| id | 唯一自增Id | Int |
| tenant\_id | 租户Id | Int |
| customer\_id | 客户idId | Int |
| email | 邮箱 | Varchar |
| authority | 权限 | varcahr |
| name | 名字 | varcahr |
| additional\_info | 备注 | varchar |

user\_creadentials（表）

|  |  |  |
| --- | --- | --- |
| 字段 | 描述 | 数据类型 |
| id | 唯一自增Id | Int |
| user\_id | 用户id | Int |
| password | 用户密码 | Varchar |

tenant表

|  |  |  |
| --- | --- | --- |
| 字段 | 描述 | 数据类型 |
| id | 唯一自增id | Int |
| title | 客户组名称 | Int |
| address | 地址 | Varchar |
| email | 邮箱 | varcahr |
| phone | 手机号 | varcahr |
| additional\_info | 备注 | varchar |

customer表

|  |  |  |
| --- | --- | --- |
| 字段 | 描述 | 数据类型 |
| id | 唯一自增id | Int |
| tenant\_id | 租户id | Int |
| title | 客户组名称 | Int |
| address | 地址 | Varchar |
| email | 邮箱 | varcahr |
| phone | 手机号 | varcahr |
| additional\_info | 备注 | varchar |

# 统一配置中心概要设计

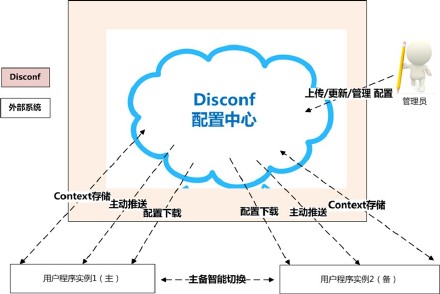
## 模块简介

统一配置中心是一套完整的基于zookeeper的分布式配置统一解决方案。他需要完成以下的一些功能需求

* 支持配置（配置项+配置文件）的分布式化管理
  + 配置发布统一化
  + 配置发布、更新统一化（云端存储、发布）:配置存储在云端系统，用户统一在平台上进行发布、更新配置。
  + 配置更新自动化：用户在平台更新配置，使用该配置的系统会自动发现该情况，并应用新配置。特殊地，如果用户为此配置定义了回调函数类，则此函数类会被自动调用。
* 配置异构系统管理
* 异构包部署统一化：这里的异构系统是指一个系统部署多个实例时，由于配置不同，从而需要多个部署包（jar或war）的情况（下同）。使用Disconf后，异构系统的部署只需要一个部署包，不同实例的配置会自动分配。特别地，在业界大量使用部署虚拟化（如JPAAS系统，SAE，BAE）的情况下，同一个系统使用同一个部署包的情景会越来越多，Disconf可以很自然地与他天然契合。 异构主备自动切换：如果一个异构系统存在主备机，主机发生挂机时，备机可以自动获取主机配置从而变成主机。
* 异构主备机Context共享工具：异构系统下，主备机切换时可能需要共享Context。可以使用Context共享工具来共享主备的Context。
* 注解式编程，极简的使用方式：我们追求的是极简的、用户编程体验良好的编程方式。通过简单的标注+极简单的代码撰写，即可完成复杂的配置分布式化。
* 需要Spring编程环境
* 简单，用户体验良好：
  + 摒弃了打散化配置的管理方式[2,3]，仍旧采用基于配置文件的编程方式，这和程序员以前的编程习惯（配置都是放在配置文件里）一致。特别的，为了支持较为小众的打散化配置功能，还特别支持了配置项。
  + 采用了基于XML无代码侵入编程方式：只需要几行XML配置，即可实现配置文件发布更新统一化、自动化。
  + 采用了基于注解式的弱代码侵入编程方式：通过编程规范，一个配置文件一个配置类，代码结构简单易懂。XML几乎没有任何更改，与原springXML配置一样。真正编程时，几乎感觉不到配置已经分布式化
* 可以托管任何类型的配置文件，这与[2,3]只能支持KV结构的功能有较大的改进。
* 配置更新实时推送
* 提供界面良好Web管理功能，可以非常方便的查看配置被哪些实例使用了。

## 模块架构图

disconf服务集群模式：



disconf的模块架构图：



## 模块要点说明

针对以上四个模块做简单介绍：

* Disconf-core
  + 分布式通知模块：支持配置更新的实时化通知
  + 路径管理模块：统一管理内部配置路径URL
* Disconf-client
  + 配置仓库容器模块：统一管理用户实例中本地配置文件和配置项的内存数据存储
  + 配置reload模块：监控本地配置文件的变动，并自动reload到指定bean
  + 扫描模块：支持扫描所有disconf注解的类和域
  + 下载模块：restful风格的下载配置文件和配置项
  + watch模块：监控远程配置文件和配置项的变化
  + 主备分配模块：主备竞争结束后，统一管理主备分配与主备监控控制
  + 主备竞争模块：支持分布式环境下的主备竞争
* Disconf-web
  + 配置存储模块：管理所有配置的存储和读取
  + 配置管理模块：支持配置的上传、下载、更新
  + 通知模块：当配置更新后，实时通知使用这些配置的所有实例
  + 配置自检监控模块：自动定时校验实例本地配置与中心配置是否一致
  + 权限控制：web的简单权限控制
* Disconf-tools
  + context共享模块：提供多实例间context的共享

## 数据库设计

app表

|  |  |  |
| --- | --- | --- |
| 字段 | 描述 | 数据类型 |
| app\_id | 唯一自增Id | Int |
| name | app名 | Varchar |
| description | 介绍 | Varchar |
| create\_time | 生成时间 | Varchar |
| update\_time | 修改时间 | Varchar |
| emails | 邮箱 | Varchar |

config表

|  |  |  |
| --- | --- | --- |
| 字段 | 描述 | 数据类型 |
| config\_id | 唯一自增Id | Int |
| type | 配置文件/配置项 | tinyint |
| status | 状态（1正常，0异常） | tinyint |
| name | 键名称 | Varchar |
| value | 键值 | Text |
| app\_id | 应用id | Int |
| version | 版本 | Varchar |
| env\_id | 环境id | Int |
| create\_time | 生成时间 | Varchar |
| update\_time | 修改时间 | Varchar |

config\_history

|  |  |  |
| --- | --- | --- |
| 字段 | 描述 | 数据类型 |
| id | 唯一自增Id | Int |
| config\_id | 配置id | Int |
| old\_value | 旧值 | Text |
| new\_value | 新值 | Text |
| create\_time | 创建时间 | Varcahr |
| update\_by | 修改者 | Int |

env表

|  |  |  |
| --- | --- | --- |
| 字段 | 描述 | 数据类型 |
| env\_id | 唯一自增Id | Int |
| name | 环境名 | Varchar |

# 3DwebGIS平台概要设计

## 模块简介

物联网3D-BIM平台是整个系统SaaS层的一个应用和验证。它基于系统的物接入模块、服务管理模块和3D压缩引擎模块，在此基础上，实现系统的3D可视化浏览，管理和操作。3D平台拥有场景展示和设备展示的能力，能够和实际场景对应起来，把实际场景的情形在浏览器中渲染出来，设备的属性信息、遥测信息、相对位置等一目了然。同时系统中的监控器，能够把异常信息送到3D平台，在3D场景中可以查看到异常设备的位置和异常信息。

主要组成部分有前端3D渲染，设备数据显示与标记，3D模型文件上传与管理，监控提醒等。用户上传3D模型到服务器端，经过压缩后，把原始模型保存在OSS中，服务器磁盘保留压缩后的模型，用户加载时请求服务器磁盘的资源。

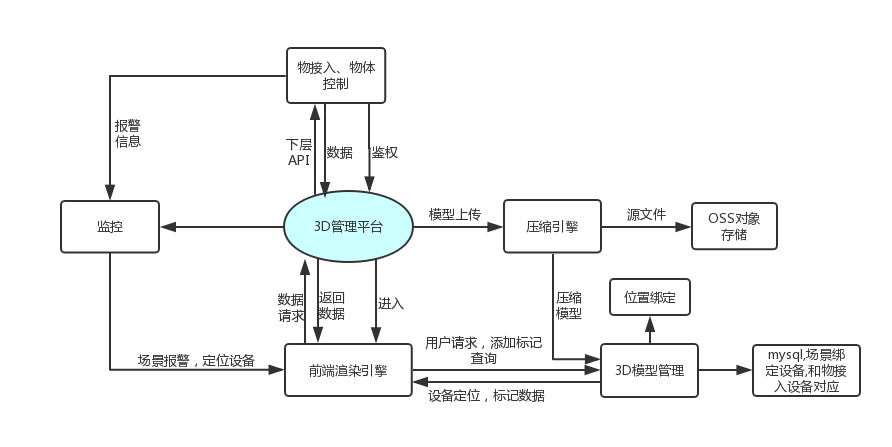


## 模块架构图

### 整体架构图



### 系统动态交互图



## 模块要点说明

### webGL渲染引擎

基本用例01：场景模型加载渲染

|  |  |  |
| --- | --- | --- |
| **系统事件名称** | **用例** | **参数说明** |
| **1.loadSceneModel(sceneModelUrl)** | 加载场景模型 | 1. **sceneModelUrl：压缩后的模型文件路径** |
| **2. processOn(xhr)** | 模型加载进度 | 1.xhr:浏览器资源请求 |
| **3. processFinsh** | 加载完成 |  |

基本用例02：设备模型加载渲染

|  |  |  |
| --- | --- | --- |
| **系统事件名称** | **用例** | **参数说明** |
| **1.loadDeviceModel(sceneModelUrl)** | 加载设备模型 | 1. **sceneModelUrl：压缩后的模型文件路径** |
| **2. processOn(xhr)** | 模型加载进度 | 1.xhr:浏览器资源请求 |
| **3. processFinsh** | 加载完成 |  |

基本用例03：设备数据信息显示

|  |  |  |
| --- | --- | --- |
| **系统事件名称** | **用例** | **参数说明** |
| **1.rayCaster（event.x,event.y）** | 识别场景中模型 | 1. **event.x,event.y:鼠标点击坐标** |
| **2. getDeviceInfo(deviceId)** | 获取设备属性信息 | 1.deviceId:设备唯一ID |
| **3. getDeviceData(deviceId)** | 获取设备遥测数据 | 1.deviceId:设备唯一ID |
| **4. getDeviceLabel(deviceId)** | 获取设备标记数据 | 1.deviceId:设备唯一ID |
| **5. getDeviceHistroycalData(deviceId)** | 获取设备历史数据 | 1.deviceId:设备唯一ID |

基本用例04：消除模型

|  |  |  |
| --- | --- | --- |
| **系统事件名称** | **用例** | **参数说明** |
| **1.removeModel** | 删除设备模型 |  |

基本用例05：设备定位

|  |  |  |
| --- | --- | --- |
| **系统事件名称** | **用例** | **参数说明** |
| **1.locateDevice(deviceLocation)** | 设备定位 | 1.deviceLocation:设备位置信息 |
| **2.lookAtdeviceLocation)** | 视角设备 | 1.deviceLocation:设备位置信息 |

基本用例06：添加设备模型

|  |  |  |
| --- | --- | --- |
| **系统事件名称** | **用例** | **参数说明** |
| **1.uploadDeviceModel（obj）** | 上传设备模型 | Obj为常用3d文件 |
| **2.defaultModel(Location)** | 使用默认模型添加 | 1.Location:设备位置信息，双击场景中一个位置，获取位置信息，也可以填入3D变换信息 |

基本用例07：设备模型3D变换

|  |  |  |
| --- | --- | --- |
| **系统事件名称** | **用例** | **参数说明** |
| **1.translate(interserted)** | 对设备模型进行3D变换 | 右键选中的模型，会出现变换工具，用Q,W,E切换变换方式 |
| **2.updateLocation** | 更新设备模型3d变换信息 | 在右侧详情框点击更新位置即可 |
| **3.escape** | 退出变换 | 按下ESC键 |

基本用例08：场景模型3D变换

|  |  |  |
| --- | --- | --- |
| **系统事件名称** | **用例** | **参数说明** |
| **1.translate(sceneModel)** | 对场景模型进行3D变换 | 在右上角的菜单中，可以对场景模型的曝光度，透明度，平移，旋转，缩放数据进行调整 |
| **2.updateLocation** | 更新场景模型3d变换信息 | 在右上角菜单中点击保存场景设置即可 |

基本用例09：场景漫游

|  |  |  |
| --- | --- | --- |
| **系统事件名称** | **用例** | **参数说明** |
| **1.lockPoint** | 对场景以漫游的方式查看 | 点击开始漫游按钮，可以用WASD控制前进，鼠标控制方向，对场景以第一人称视角查看 |
| **2.escape** | 退出漫游 | 按下ESC键 |

### 模型坐标系

基本用例01：点击获取当前坐标信息

|  |  |  |
| --- | --- | --- |
| **系统事件名称** | **用例** | **参数说明** |
| **1.listenClick（window）** | 监听鼠标点击 | 1. **window:浏览器窗口** |
| **2.get**[**Coordinate**](https://www.baidu.com/link?url=YizAMUSBCYrYsGy9hxdZ0rGs5pxuZJXD-7AvxCohYy4JmLq2DslNuCSNBTY-OVbTZKaqOwn3K8l62gF08YV2o5fGrzllJ_cR-LVX--9CKMS&wd=&eqid=eda2ddd8000402d9000000025ac858d2)**(window.x,window.y)** | 得到基于场景的坐标信息 | 1. **window.x,window.y** ：当前鼠标位置 |

### 3D模型管理

基本用例01：设备信息与模型绑定

|  |  |  |
| --- | --- | --- |
| **系统事件名称** | **用例** | **参数说明** |
| **1.bindDeviceModel(deviceId,mesh)** | 模型中嵌入设备信息 | 1.deviceId:设备唯一ID |

基本用例02：模型上传

|  |  |  |
| --- | --- | --- |
| **系统事件名称** | **用例** | **参数说明** |
| **1.getLocalModel(url)** | 从本地读取模型 | 1.url:本地模型Url |
| **2. postModelToEcs（xhr）** | 向服务器上传模型文件 | 1.xhr:上传的资源 |
| **3. processOn(xhr)** | 上传进度 | 1.xhr:上传的资源 |
| **4. showHint(modelStatus,compressStatus)** | 提示模型上传和压缩状态 | 1.modelStatus:模型上传状态  2.compressStatus:模型压缩状态 |

基本用例03：场景与设备绑定

|  |  |  |
| --- | --- | --- |
| **系统事件名称** | **用例** | **参数说明** |
| **1.bindDeviceofScene (deviceIds,sceneID)** | 绑定设备模型到场景中 | 1.deviceIds:一个或多个设备唯一ID  2. **sceneID**:场景ID |
| **2.cancelDeviceofScene (deviceId,sceneID)** | 取消绑定模型到场景 | 1.deviceId:设备唯一ID  2. **sceneID**:场景ID |

基本用例04：本地加载模型

### 通过3D模型展示设备详细信息

基本用例01：点击模型获取设备属性信息

|  |  |  |
| --- | --- | --- |
| **系统事件名称** | **用例** | **参数说明** |
| **1.getDevcieAttr(deviceId,mesh)** | 获取模型对应设备的属性信息 | 1.deviceId:设备唯一ID  2.mesh：和设备对应的3D模型 |
| **2.showDetail()** | 展示设备详细信息 | 包括所有属性值，最新遥测数据，历史数据和控制面板 |

基本用例02：鼠标滑过模型显示标签，点击出现设备面板

|  |  |  |
| --- | --- | --- |
| **系统事件名称** | **用例** | **参数说明** |
| **1.showTip(mesh)** | 显示标签，标签展示设备名 | 1.mesh：和设备对应的3D模型 |
| **2.clickTip(deviceInfo)** | 点击标签出现设备面板 | 1.deviceInfo:设备基本信息  在设备面板上可以清除模型，更新位置，定位模型 |

基本用例03：设备属性信息搜索键值

|  |  |  |
| --- | --- | --- |
| **系统事件名称** | **用例** | **参数说明** |
| **1.searchKey(key)** | 搜索设备键值 | 1.key ，搜索内容 |

基本用例04：查看历史数据

|  |  |  |
| --- | --- | --- |
| **系统事件名称** | **用例** | **参数说明** |
| **1.getHistoryData(startTime,endTime)** | 根据起止时间查看历史数据 | 1.startTime:起始时间  2.endTime:结束时间 |

基本用例05：控制设备

|  |  |  |
| --- | --- | --- |
| **系统事件名称** | **用例** | **参数说明** |
| **1.sendControl(uid,status)** | 如果有对应的服务，则会获得控制面板 | 1.uid:设备的Uid  2.status：设备的状态 |

基本用例06：搜索设备

|  |  |  |
| --- | --- | --- |
| **系统事件名称** | **用例** | **参数说明** |
| **1.searchDevice(textSearch)** | 在当前场景下搜索该站点的设备 | 1. **textSearch**:搜索关键字 |

基本用例07：设备列表

|  |  |  |
| --- | --- | --- |
| **系统事件名称** | **用例** | **参数说明** |
| **1.deviceList()** | 该站点被分配的所有设备列表 |  |
| **2.showDetail()** | 点击一列显示设备详情 |  |

### 地图界面

基本用例01：在地图上添加站点

|  |  |  |
| --- | --- | --- |
| **系统事件名称** | **用例** | **参数说明** |
| **1.addSite(longtitude,lantitude,siteName)** | 在鼠标点击位置添加站点 | 1.longtitude,lantitude:当前点击位置经纬度  2.siteName：站点名称 |
| **2.addSiteByLocation()** | 输入经纬度添加站点 | 同上 |

基本用例02：框选搜索

|  |  |  |
| --- | --- | --- |
| **系统事件名称** | **用例** | **参数说明** |
| **1.searchByDrag()** | 在地图上拖拽一个框，显示框中的站点 |  |
| **2.showSiteDevice()** | 点击一个站点，展示这个站点被分配的所有设备 |  |

基本用例03：设备搜索

|  |  |  |
| --- | --- | --- |
| **系统事件名称** | **用例** | **参数说明** |
| **1.searchDevice(****textSearch)** | 根据设备名称搜索当前租户的设备，展示信息和所属站点，并可进入相应站点 | 1. **textSearch**:搜索关键字 |

基本用例04：修改站点

|  |  |  |
| --- | --- | --- |
| **系统事件名称** | **用例** | **参数说明** |
| **1.reName(newName)** | 对当前站点重命名 | 1. **newName**:站点新名称 |

基本用例05：进入场景

|  |  |  |
| --- | --- | --- |
| **系统事件名称** | **用例** | **参数说明** |
| **1.goToScene(siteId)** | 进入站点对应的3D场景 | 1. **siteId**:站点ID |

### 设备列表

基本用例01：搜索设备

|  |  |  |
| --- | --- | --- |
| **系统事件名称** | **用例** | **参数说明** |
| **1.searchDevice(textSearch)** | 根据设备名称搜索设备 | 1.textSearch:设备名称 |

基本用例02：设备展示

|  |  |  |
| --- | --- | --- |
| **系统事件名称** | **用例** | **参数说明** |
| **1.allDevice (tenantId)** | 当前租户下所有设备分页展示 | 1.tenantId:租户ID |

基本用例03：分配/取消分配到站点，定位到站点操作

|  |  |  |
| --- | --- | --- |
| **系统事件名称** | **用例** | **参数说明** |
| **1.assignDevice (deviceId,siteId)** | 把某个设备分配给一个站点 | 1.deviceId:被分配的设备Id  2,siteID:分配的站点ID |
| **2.flytoSite ()** | 常看某一设备所属站点在地图上的位置 |  |
| **3.cancelAssign (deviceId)** | 取消分配某个设备到站点 | 1.deviceId:被取消分配的设备Id |

## 数据库设计

tenantOwnModel表:

|  |  |  |
| --- | --- | --- |
| **字段** | **描述** | **数据类型** |
| **tenantId(key)** | 租户Id | Varchar |
| **sceneModelNum** | 租户拥有的场景模型数量 | Int |
| **deviceModelNum** | 租户拥有的设备模型数量 | Int |

Sites表:

|  |  |  |
| --- | --- | --- |
| **字段** | **描述** | **数据类型** |
| **id(key)** | 站点ID | Int |
| **tenantId** | 租户Id | Varchar |
| **name** | 站点名称 | Varchar |
| **longtitude** | 站点的经度 | Double |
| **lantitude** | 站点的纬度 | Double |
| **sceneUrl** | 场景模型的路径 | Varchar |
| **compressStatus** | 模型压缩状态 | Int |
| **ossStatus** | 源文件是否存入OSS | Boolean |
| **devicesModelCount** | 场景中设备模型数量 | BIGInt |
| **sceneModelLoca** | 场景模型3D变换信息 | Varchar |

devicesMode表l:

|  |  |  |
| --- | --- | --- |
| **字段** | **描述** | **数据类型** |
| **id** | 设备模型ID | INT |
| **tenantId** | 租户Id | Varchar |
| **name** | 设备名称 | Varchar |
| **deviceId** | 模型对应的设备Id | Varchar |
| **siteId** | 模型所属站点ID | Int |
| **deviceModelUrl** | 设备模型的路径 | Varchar |
| **compressStatus** | 模型压缩状态 | Int |
| **ossStatus** | 源文件是否存入OSS | Boolean |
| **location** | 设备模型在当前场景下的位置信息 | Varchar |
| **label** | 模型标签 | Text |