

用户和系统管理服务

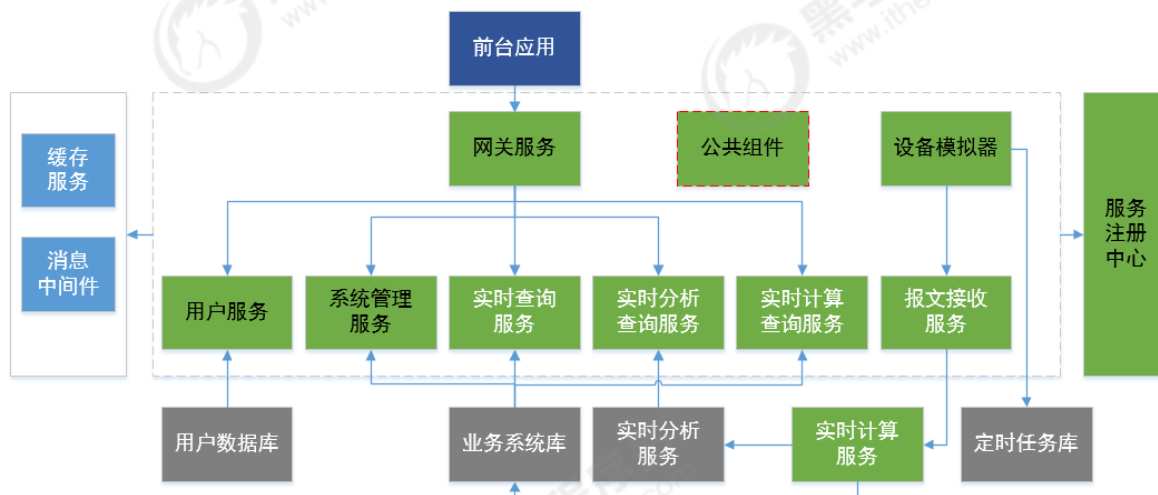
学习目标:

1. 能够掌握如何创建springboot应用
2. 能够掌握Eureka服务注册中心
3. 能够掌握API网关和spring cloud gateway
4. 能够掌握监控平台系统管理的用户和系统管理服务
5. 能够知道和了解分布式任务调度框架Quartz

项目基于spring cloud 微服务体系搭建，主要包括以下部分：

cold-chain-monitor		
└ cold-eureka	# Eureka服务注册中心	端口： 8001
└ cold-gateway	# Spring Cloud Gateway网关	端口： 8080
└ cold-user	# 用户服务	端口： 8185
└ cold-admin	# 业务管理、系统管理服务	端口： 8181
└ cold-monitor	# 实时数据查询	端口： 8183
└ cold-druid	# 历史数据查询	端口： 8182
└ cold-jobs	# 分布式任务调度（硬件模拟）	端口： 8184
└ cold-netty	# 设备报文接收服务	端口： 10010
└ cold-common	# 对象实体、公共组件	
└ cold-flink	# 实时数据处理	flink集群运行
└ cold-ui	# 前端服务	端口： 8081

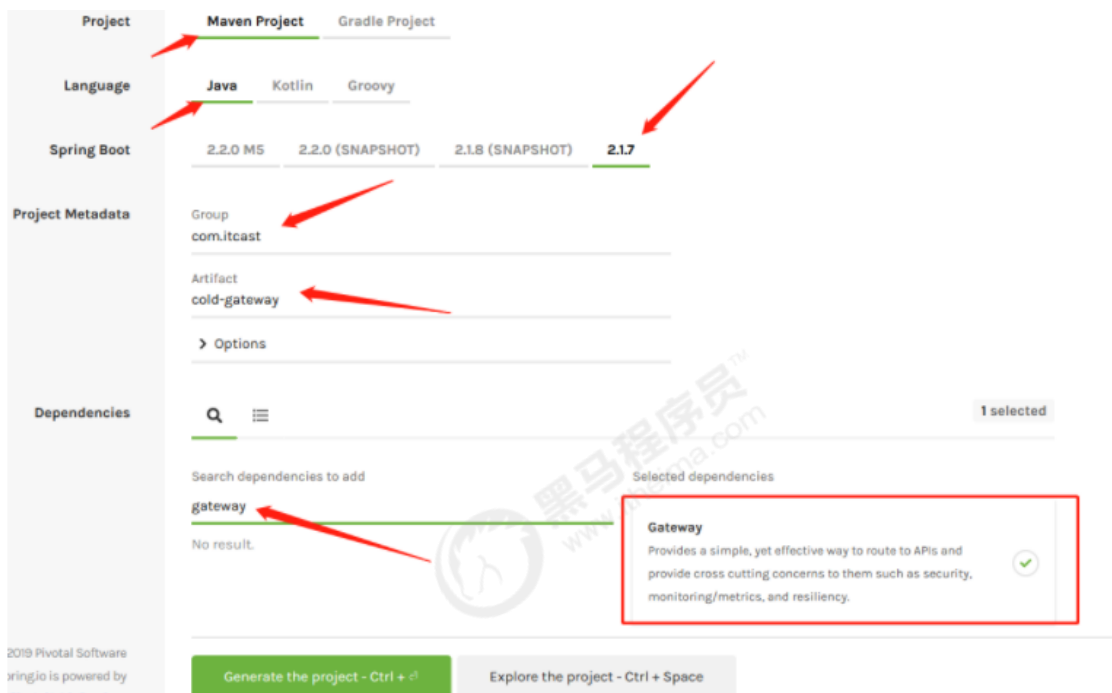
系统微服务体系结构如下：



1 快速创建spring boot应用

有很多种方式可以创建spring boot应用，比如使用idea或eclipse创建、修改pom文件创建，今天给大家介绍另外一种方式来创建spring cloud的应用。

1. 访问<https://start.spring.io>
2. 输入和选择相应的参数（在这个示例中，选择的是gateway组件）



3. 点击【Generate the project...】按钮，生成项目

4. 导入项目

此时pom.xml文件内容为：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.1.7.RELEASE</version>
    <relativePath/> <!-- lookup parent from repository -->
  </parent>
  <groupId>com.itcast</groupId>
  <artifactId>cold-gateway</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>cold-gateway</name>
  <description>Demo project for Spring Boot</description>

  <properties>
    <java.version>1.8</java.version>
    <spring-cloud.version>Greenwich.SR2</spring-cloud.version>
  </properties>

  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-starter-gateway</artifactId>
    </dependency>

    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-test</artifactId>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

```

        </dependency>
    </dependencies>

    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-dependencies</artifactId>
                <version>${spring-cloud.version}</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>

</project>

```

这样，我们就能够创建了一个添加完pom引用的空的spring boot应用了。

2 服务注册中心 (cold-eureka)

微服务为什么需要注册中心

举个现实生活中的例子，比如说，我们手机中的通讯录的两个使用场景，

- 1、当我想给张三打电话时，那我需要在通讯录中按照名字找到张三，然后就可以找到他的手机号拨打电话。
- 2、李四办了手机号，那么他把手机号告诉我，我把李四的号码存进通讯录，后续，我就可以从通讯录找到他。

另外一个例子是很多公司都有自己的公司通讯录。

上述场景就是我们在微服务架构中常常提到的：服务发现。服务注册在微服务架构中，由于每一个服务的粒度相对传统SOA来说要小的多，所以服务的数量会成倍增加。这时如果有效管理服务的注册信息就尤为重要。

在分布式系统中，我们不仅仅是需要在注册中心找到服务和地址的映射关系这么简单，我们还需要考虑更多更复杂的问题：服务注册后，如何被及时发现服务宕机后，如何及时下线服务如何有效的水平扩展服务发现时，如何进行路由服务异常时，如何进行降级注册中心如何实现自身的高可用

Eureka是Netflix开发的服务发现框架，本身是一个基于REST的服务，主要用于定位运行在AWS域中的中间层服务，以达到负载均衡和中间层服务故障转移的目的。

Eureka服务注册中心由3个角色组成：

- 1、Eureka Server

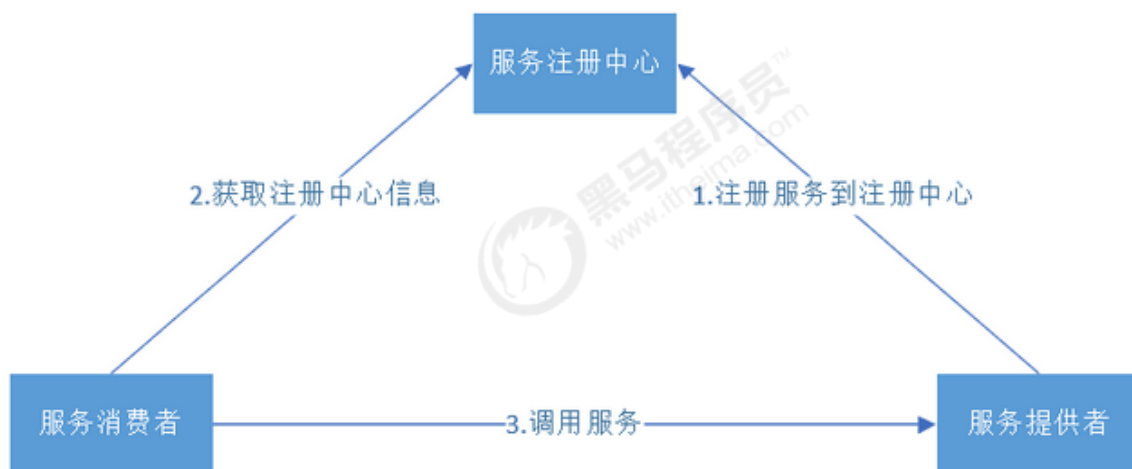
提供服务注册和发现

2、Service Provider

服务提供方将自身服务注册到Eureka，从而使服务消费方能够找到

3、Service Consumer

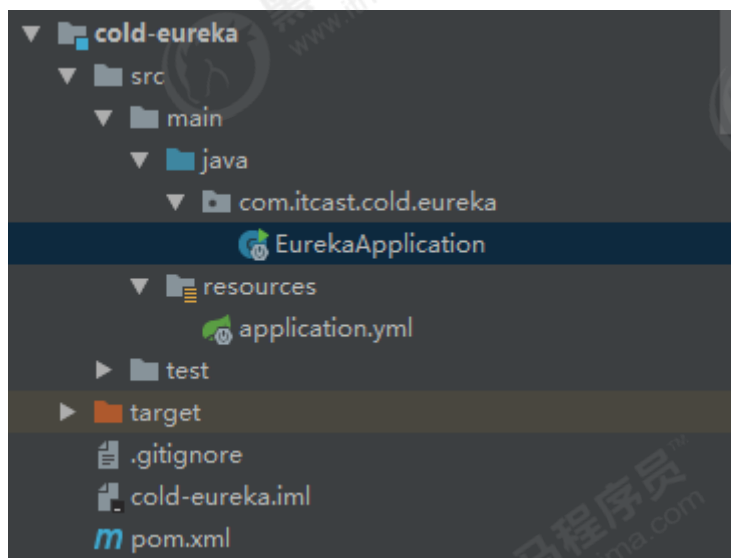
服务消费方从Eureka获取注册服务列表，从而能够消费服务



Spring Cloud将它集成在其子项目spring-cloud-netflix中，以实现SpringCloud的服务发现功能。

搭建Eureka服务注册中心

Eureka服务注册中心也是一个spring boot应用，我们可以使用上节提供的方法，快速创建Eureka服务注册中心。



pom文件：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.1.7.RELEASE</version>
```

```

    <relativePath/> <!-- lookup parent from repository -->
  </parent>
  <groupId>com.itcast</groupId>
  <artifactId>cold-eureka</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>cold-eureka</name>
  <description>Demo project for Spring Boot</description>

  <properties>
    <java.version>1.8</java.version>
    <spring-cloud.version>Greenwich.SR2</spring-cloud.version>
  </properties>

  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
    </dependency>

    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-test</artifactId>
      <scope>test</scope>
    </dependency>
  </dependencies>

  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-dependencies</artifactId>
        <version>${spring-cloud.version}</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
    </dependencies>
  </dependencyManagement>

  <build>
    <plugins>
      <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
      </plugin>
    </plugins>
  </build>
</project>

```

application.yml (快速创建的应用默认是application.properties文件,需要更改后缀名为yml) 文件内容:

```

server:
  port: 8001

eureka:
  instance: #Eureka实例

```

```
hostname: eureka-8001.itcast.cn #Eureka实例所在的主机名

client:
  service-url:
    defaultZone: http://eureka-8001.itcast.cn:8001/eureka

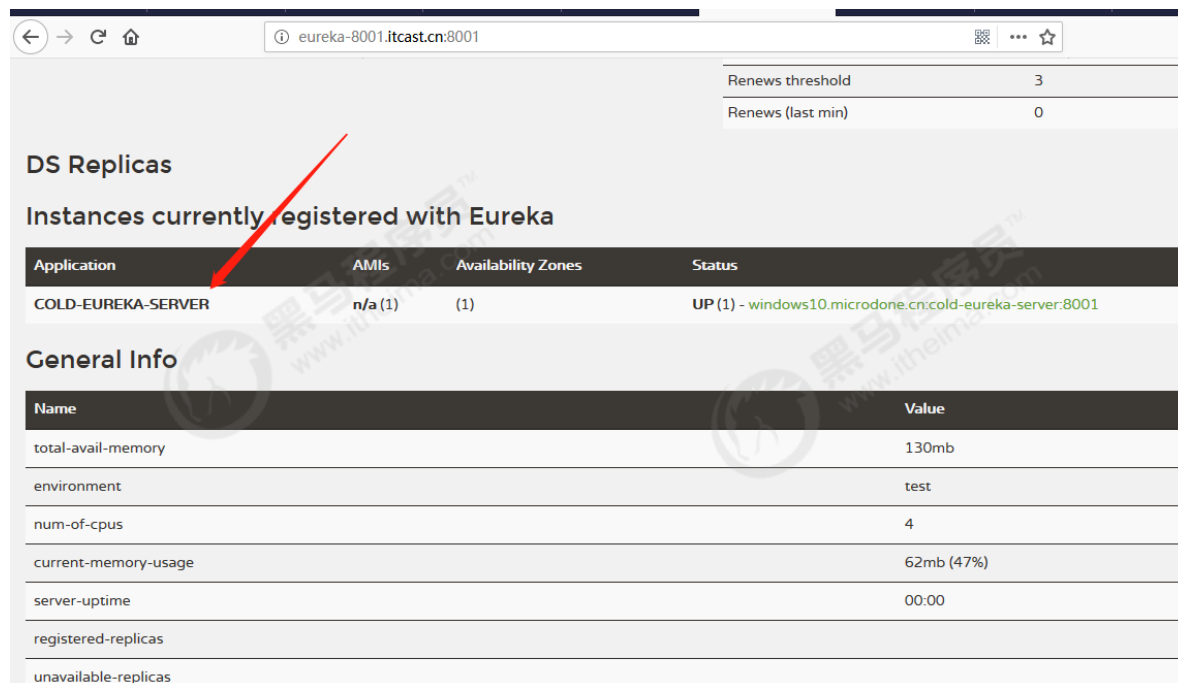
  register-with-eureka: true #表示是否将自己注册到Eureka Server上，默认为true
  fetch-registry: false      #表示是否从Eureka Server上获取注册信息，默认为true

spring:
  application:
    name: cold-eureka-server
```

有两个地方需要注意：

- 1、需要在host文件中添加：eureka-8001.itcast.cn
- 2、本例中将Eureka也作为客户端注册到了服务注册中心

通过:<http://eureka-8001.itcast.cn:8001> 访问服务注册中心：



The screenshot shows the Eureka Server web interface. At the top right, there are two metrics: 'Renews threshold' with a value of 3, and 'Renews (last min)' with a value of 0. The main section is titled 'DS Replicas' and 'Instances currently registered with Eureka'. Below this is a table with the following data:

Application	AMIs	Availability Zones	Status
COLD-EUREKA-SERVER	n/a (1)	(1)	UP (1) - windows10.microdone.cn:cold-eureka-server:8001

A red arrow points to the 'COLD-EUREKA-SERVER' application name in the table. Below the table is a 'General Info' section with a table of system metrics:

Name	Value
total-avail-memory	130mb
environment	test
num-of-cpus	4
current-memory-usage	62mb (47%)
server-uptime	00:00
registered-replicas	
unavailable-replicas	

3. 前后端分离开发规范

3.1 背景

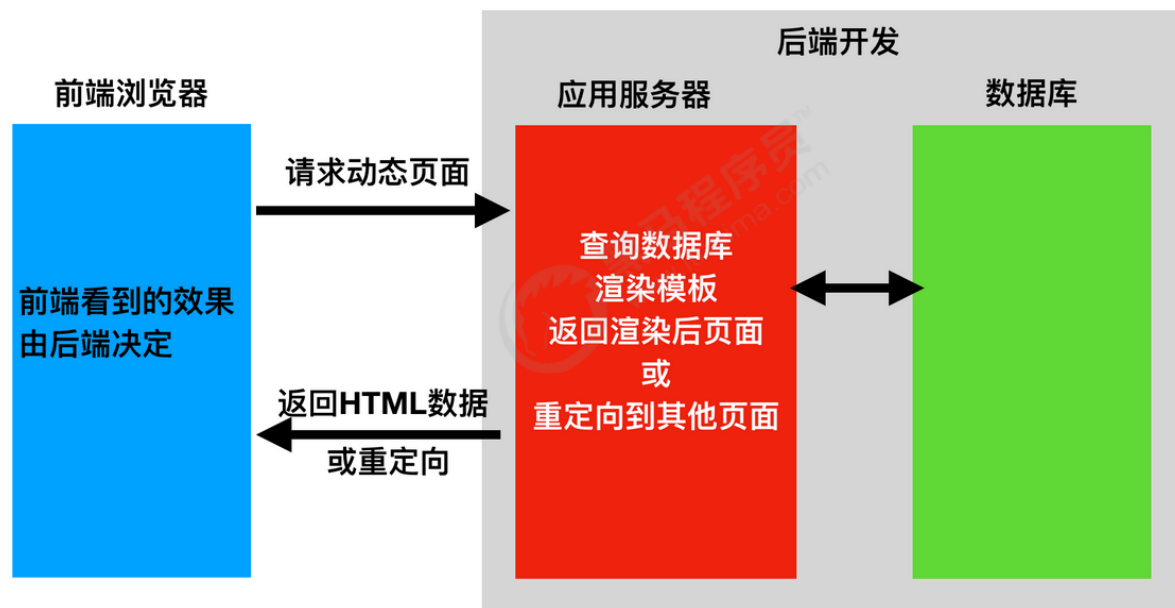
随着互联网的高速发展，前端页面的展示、交互体验越来越灵活、炫丽，响应体验也要求越来越高，后端服务的高并发、高可用、高性能、高扩展等特性的要求也愈加苛刻，从而导致前后端研发各自专注于自己擅长的领域深耕细作。

然而带来的另一个问题：前后端的对接界面双方却关注甚少，没有任何接口约定规范情况下各自干各自的，导致我们在产品项目开发过程中，前后端的接口联调对接工作量占比在30%-50%左右，甚至会更高。往往前后端接口联调对接及系统间的联调对接都是整个产品项目研发的软肋。

3.1.1 前后端不分离

在前后端不分离的应用模式中，前端页面看到的效果都是由后端控制，由后端渲染页面或重定向，也就是后端需要控制前端的展示，前端与后端的耦合度很高。这种应用模式比较适合纯网页应用，但是当后端对接App时，App可能并不需要后端返回一个HTML网页，而仅仅是数据本身，所以后端原本返回网页的接口不再适用于前端App应用，为了对接App后端还需再开发一套接口。

数据交互图如下：



前后端不分离带来几个比较典型的问题：

- 前端开发重度依赖开发环境，开发效率低
- 前后端职责依旧纠缠不清
- 对前端发挥的局限

3.1.2 前后端分离

在前后端分离的应用模式中，后端仅返回前端所需的数据，不再渲染HTML页面，不再控制前端的效果。至于前端用户看到什么效果，从后端请求的数据如何加载到前端中，都由前端自己决定，网页有网页的处理方式，App有App的处理方式，但无论哪种前端，所需的数据基本相同，后端仅需开发一套逻辑对外提供数据即可。

在前后端分离的应用模式中，前端与后端的耦合度相对较低，我们通常将后端开发的每个视图都称为一个接口，或者API，前端通过访问接口来对数据进行增删改查。

数据交互如下图：

请求方法	请求地址	后端操作
GET	/students	获取所有学生
POST	/students	增加学生
GET	/students/1	获取编号为1的学生
PUT	/students/1	修改编号为1的学生
DELETE	/students/1	删除编号为1的学生

3.3.2 域名

应该尽量将API部署在专用域名之下，如：<https://api.example.com>

如果确定API很简单，不会有进一步扩展，可以考虑放在主域名下，如：<https://example.org/api/>

3.3.3 版本

应该将API的版本号放入URL：

```
http://www.example.com/app/1.0/foo
http://www.example.com/app/1.1/foo
http://www.example.com/app/2.0/foo
```

另一种做法是，将版本号放在HTTP头信息中，但不如放入URL方便和直观。Github就采用了这种做法。因为不同的版本，可以理解成同一种资源的不同表现形式，所以应该采用同一个URL。版本号可以在HTTP请求头信息的Accept字段中进行区分：

```
Accept: vnd.example-com.foo+json; version=1.0
Accept: vnd.example-com.foo+json; version=1.1
```

3.3.4 路径

路径又称“终点”（endpoint），表示API的具体网址，每个网址代表一种资源（resource）

1. 资源作为网址，只能有名词，不能有动词，而且所用的名词往往与数据库的表名对应。举例来说，以下是不好的例子：

```
/getProducts
/listOrders
/retrieveClientByOrder?orderId=1
```

对于一个简洁结构，应该始终用名词。此外，利用的HTTP方法可以分离网址中的资源名称的操作。

```
GET /products : 将返回所有产品清单
POST /products : 将产品新建到集合
GET /products/4 : 将获取产品 4
PATCH（或）PUT /products/4 : 将更新产品 4
```

2. API中的名词应该使用复数。无论子资源或者所有资源。举例来说，获取产品的API可以这样定义

获取单个产品: <http://127.0.0.1:8080/AppName/rest/products/1>
获取所有产品: <http://127.0.0.1:8080/AppName/rest/products>

3.3.5 HTTP动词

对于资源的具体操作类型, 由HTTP动词表示。常用的HTTP动词有下面四个(括号里是对应的SQL命令)。

- GET (SELECT): 从服务器取出资源(一项或多项)。
- POST (CREATE): 在服务器新建一个资源。
- PUT (UPDATE): 在服务器更新资源(客户端提供改变后的完整资源)。
- DELETE (DELETE): 从服务器删除资源。

还有三个不常用的HTTP动词:

- PATCH (UPDATE): 在服务器更新(更新)资源(客户端提供改变的属性)。
- HEAD: 获取资源的元数据。
- OPTIONS: 获取信息, 关于资源的哪些属性是客户端可以改变的。

下面是一些例子:

- GET /zoos: 列出所有动物园
- POST /zoos: 新建一个动物园(上传文件)
- GET /zoos/ID: 获取某个指定动物园的信息
- PUT /zoos/ID: 更新某个指定动物园的信息(提供该动物园的全部信息)
- PATCH /zoos/ID: 更新某个指定动物园的信息(提供该动物园的部分信息)
- DELETE /zoos/ID: 删除某个动物园
- GET /zoos/ID/animals: 列出某个指定动物园的所有动物
- DELETE /zoos/ID/animals/ID: 删除某个指定动物园的指定动物

3.3.6 过滤信息

如果记录数量很多, 服务器不可能都将它们返回给用户。API应该提供参数, 过滤返回结果。下面是一些常见的参数。query_string 查询字符串,地址栏后面问号后面的数据,格式: name=xx&sss=xxx

?limit=10: 指定返回记录的数量
?offset=10: 指定返回记录的开始位置。
?page=2&per_page=100: 指定第几页, 以及每页的记录数。
?sortby=name&order=asc: 指定返回结果按照哪个属性排序, 以及排序顺序。
?animal_type_id=1: 指定筛选条件

参数的设计允许存在冗余, 即允许API路径和URL参数偶尔有重复。比如, GET /zoos/ID/animals 与 GET /animals?zoo_id=ID 的含义是相同的

3.3.7 状态码

服务器向用户返回的状态码和提示信息, 常见的有以下一些(方括号中是该状态码对应的HTTP动词)。

- OK - [GET]: 服务器成功返回用户请求的数据
- CREATED - [POST/PUT/PATCH]: 用户新建或修改数据成功。
- Accepted - [*]: 表示一个请求已经进入后台排队（异步任务）
- NO CONTENT - [DELETE]: 用户删除数据成功。
- INVALID REQUEST - [POST/PUT/PATCH]: 用户发出的请求有错误，服务器没有进行新建或修改数据的操作
- Unauthorized - [*]: 表示用户没有权限（令牌、用户名、密码错误）。
- Forbidden - [*] 表示用户得到授权（与401错误相对），但是访问是被禁止的。
- NOT FOUND - [*]: 用户发出的请求针对的是不存在的记录，服务器没有进行操作，该操作是幂等的。
- Not Acceptable - [GET]: 用户请求的格式不可得（比如用户请求JSON格式，但是只有XML格式）。
- Gone -[GET]: 用户请求的资源被永久删除，且不会再得到的。
- Unprocessable entity - [POST/PUT/PATCH] 当创建一个对象时，发生一个验证错误。
- INTERNAL SERVER ERROR - [*]: 服务器发生错误，用户将无法判断发出的请求是否成功。

3.3.8 错误处理

如果状态码是4xx，服务器就应该向用户返回出错信息。返回的信息中可以将code、message作为键名，code提示返回码，message展示返回信息：

```
{
  code: "50x"
  message: "Invalid API key"
}
```

3.3.9 返回结果

针对不同操作，服务器向用户返回的结果应该符合以下规范。

- GET /collection: 返回资源对象的列表（数组）
- GET /collection/ID: 返回单个资源对象(json)
- POST /collection: 返回新生成的资源对象(json)
- PUT /collection/ID: 返回完整的资源对象(json)
- DELETE /collection/ID: 返回一个空文档(空字符串)

3.3.10 超媒体

RESTful API最好做到Hypermedia，即返回结果中提供链接，连向其他API方法。使得用户不查文档，也知道下一步应该做什么。

比如，Github的API就是这种设计，访问api.github.com会得到一个所有可用API的网址列表。

```
{
  "current_user_url": "https://api.github.com/user",
  "authorizations_url": "https://api.github.com/authorizations",
  // ...
}
```

从上面可以看到，如果想获取当前用户的信息，应该去访问api.github.com/user，然后就得到了下面结果：

```
{
  "message": "Requires authentication",
  "documentation_url": "https://developer.github.com/v3"
}
```

上面代码表示，服务器给出了提示信息，以及文档的网址。

3.3.11 其他

服务器返回的数据格式，应该尽量使用JSON，避免使用XML。

注意：以上规范只是一种开发约定，并非强制遵守。

3.4 开发规范

3.4.1 规范原则

- 接口返回数据即显示：前端仅做渲染逻辑处理；
- 渲染逻辑禁止跨多个接口调用；
- 前端关注交互、渲染逻辑，尽量避免业务逻辑处理的出现；
- 请求响应传输数据格式：JSON，JSON数据尽量简单轻量，避免多级JSON的出现；

3.4.2 基本格式

请求基本格式

GET请求、POST请求等信息必须包含key为body的入参，所有请求数据包装为JSON格式，并存放入参body中，示例如下：

GET请求：

```
xxx/login?body={
  "username": "admin",
  "password": "123456",
  "captcha": "scfd",
  "rememberMe": 1
}
```

POST请求：



响应基本格式

code 返回请求的处理状态：

- 200：请求处理成功
- 500：请求处理失败
- 401：请求未认证，跳转登录页
- 406：请求未授权，跳转未授权提示页
- ...

data.message: 请求处理消息:

- code=200 且 data.message="success": 请求处理成功
- code=200 且 data.message!="success": 请求处理成功，普通消息提示：message内容
- code=500：请求处理失败，警告消息提示：message内容
- ...

3.4.3 响应实体信息

data.entity: 响应返回的实体数据

```
{
  code: 200,
  data: {
    message: "success",
    entity: {
      id: 1,
      name: "xxx",
      code: "xxx"
    }
  }
}
```

3.4.4 响应列表格式

data.list: 响应返回的列表数据

```
{
  code: 200,
  message: "success",
  data: {
    list{
      entity: {
        id: 1,
        name: "xxx",
        code: "xxx"
      },
      entity: {
        id: 2,
        name: "xxx",
        code: "xxx"
      },
    }
  }
}
```

3.4.5 响应分页格式

```
{
  code: 200,
  message: "success",
  recordCount: 2,
  totalCount: 2,
  pageNo: 1,
  pageSize: 10,
  data: {
    list: [
      {
        id: 1,
        name: "xxx",
        code: "H001"
      },
      {
        id: 2,
        name: "xxx",

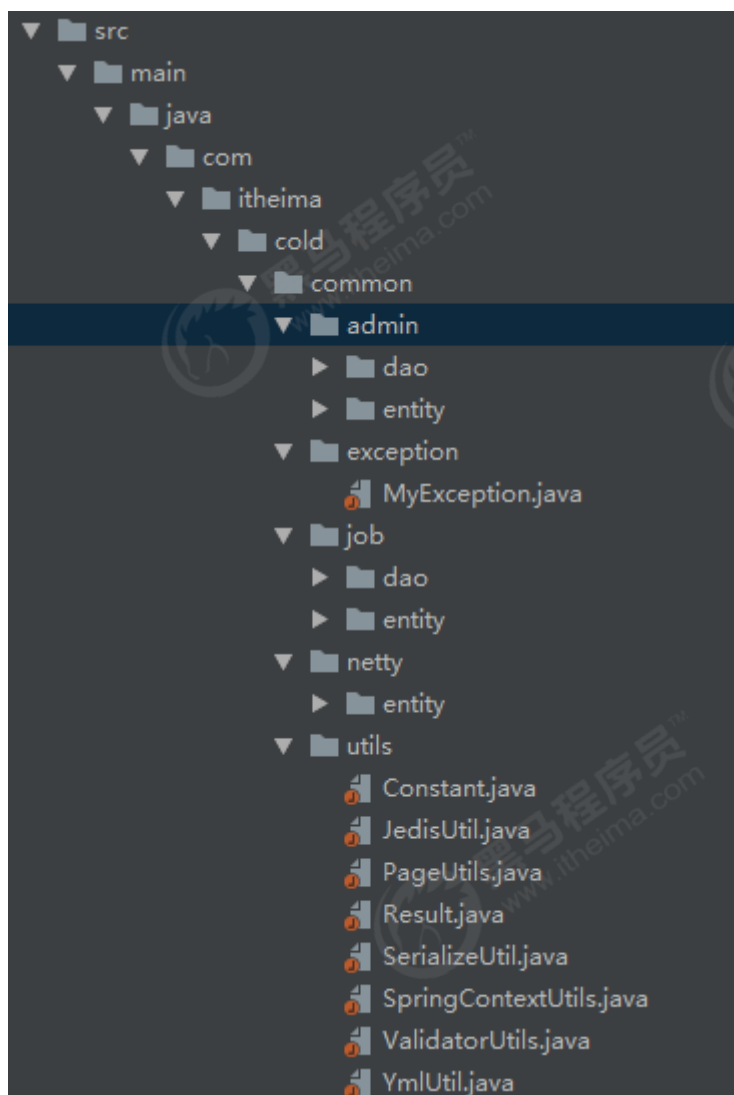
```

```
        code: "H001"
    }
    ],
    totalPage: 1
}
}
```

- data.recordCount: 当前页记录数
- data.totalCount: 总记录数
- data.pageNo: 当前页码
- data.pageSize: 每页大小
- data.totalPage: 总页数

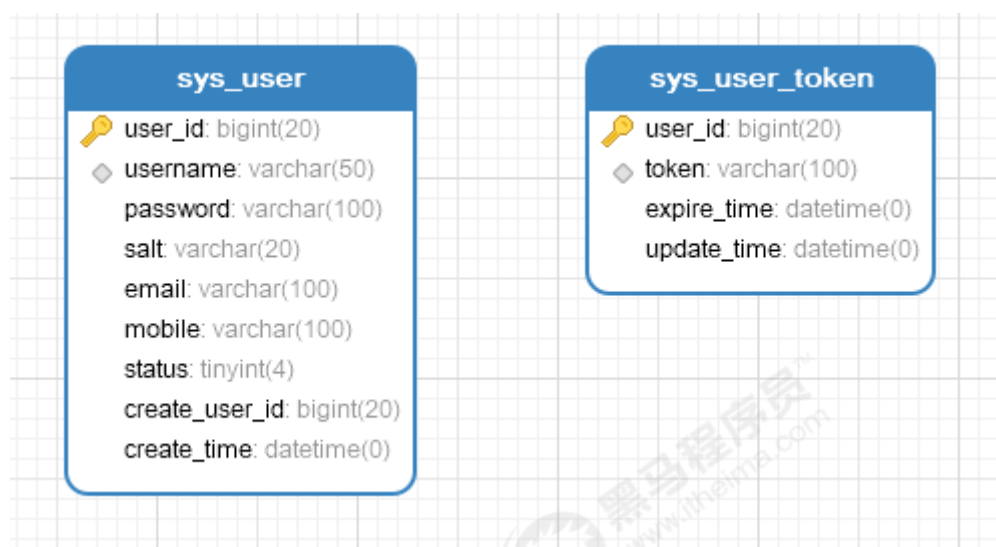
4 公共组件 (cold-common)

公共组件工程是一个普通的java工程，实体层、DAO层代码和一些工具类、公共类放在这个工程中，以便于其他项目引用。工程代码结构如下：



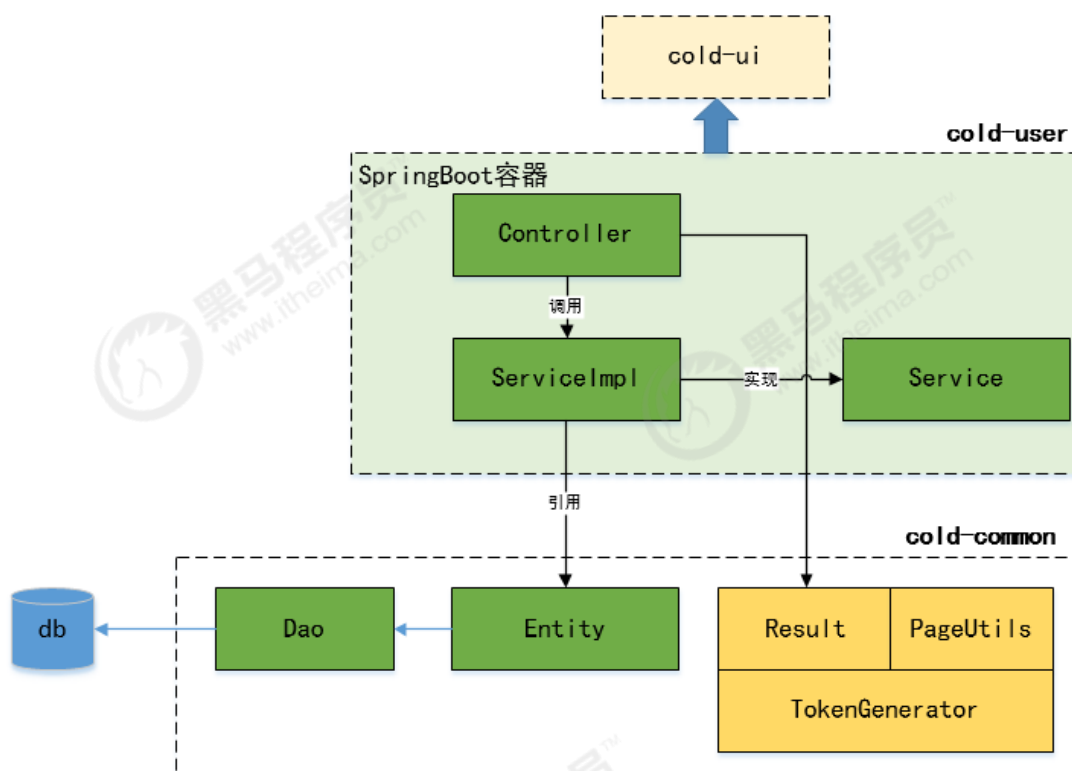
5 用户服务 (cold-user)

5.1 数据模型



5.2 代码实现

5.2.1 代码结构



5.2.2 登录接口

1. controller

```
/**
 * 登录
 */
@PostMapping("/login")
public Map<String, Object> login(@RequestBody SysLoginForm form)throws
IOException {
    //用户信息
    SysUserEntity user =
    sysUserService.queryByUsername(form.getUsername());
}
```

```

        //账号不存在、密码错误
        if(user == null || !user.getPassword().equals(new
        Sha256Hash(form.getPassword(), user.getSalt()).toHex())) {
            return R.error("账号或密码不正确");
        }

        //生成token, 并保存到数据库
        R r = sysUserTokenService.createToken(user.getUserId());
        return r;
    }

    /**
     * 退出
     */
    @PostMapping("/logout")
    public R logout(@RequestHeader(value="token") String token) {
        sysUserTokenService.logout(token);
        return R.ok();
    }
}

```

2. service

```

@Service("sysUserService")
public class SysUserServiceImpl extends ServiceImpl<SysUserDao,
SysUserEntity> implements SysUserService {

    @Override
    public SysUserEntity queryByUsername(String username) {
        return baseMapper.queryByUsername(username);
    }

}

```

3. dao

```

@Mapper
public interface SysUserDao extends BaseMapper<SysUserEntity> {
    /**
     * 根据用户名, 查询系统用户
     */
    SysUserEntity queryByUsername(String username);
}

```

```

<select id="queryByUsername"
resultType="com.itheima.cold.user.entity.SysUserEntity">
    select * from sys_user where username = #{username}
</select>

```

4. entity

```

@Data
@TableName("sys_user")
public class SysUserEntity implements Serializable {
    private static final long serialVersionUID = 1L;
}

```



```
/**
 * 用户ID
 */
@TableId
private Long userId;

/**
 * 用户名
 */
@NotBlank(message="用户名不能为空")
private String username;

/**
 * 密码
 */
@NotBlank(message="密码不能为空")
private String password;

/**
 * 盐
 */
private String salt;

/**
 * 邮箱
 */
private String email;

/**
 * 手机号
 */
private String mobile;

/**
 * 状态 0: 禁用 1: 正常
 */
private Integer status;

/**
 * 创建者ID
 */
private Long createUserId;

/**
 * 创建时间
 */
private Date createTime;
}
```

6 网关服务 (cold-gateway)

6.1 API网关

API 网关出现的原因是微服务架构的出现，不同的微服务一般会有不同的网络地址，而外部客户端可能需要调用多个服务的接口才能完成一个业务需求，如果让客户端直接与各个微服务通信，会有以下的问题：


- 客户端会多次请求不同的微服务，增加了客户端的复杂性。
- 存在跨域请求，在一定场景下处理相对复杂。
- 认证复杂，每个服务都需要独立认证。
- 难以重构，随着项目的迭代，可能需要重新划分微服务。例如，可能将多个服务合并成一个或者将一个服务拆分成多个。如果客户端直接与微服务通信，那么重构将会很难实施。
- 某些微服务可能使用了防火墙 / 浏览器不友好的协议，直接访问会有一些困难。

以上这些问题可以借助 API 网关解决。API 网关是介于客户端和服务端之间的中间层，所有的外部请求都会先经过 API 网关这一层。也就是说，API 的实现方面更多的考虑业务逻辑，而安全、性能、监控可以交由 API 网关来做，这样既提高业务灵活性又不缺安全性，典型的架构图如图所示：


1565918533701

使用 API 网关后的优点如下：

- 易于监控。可以在网关收集监控数据并将其推送到外部系统进行分析。
- 易于认证。可以在网关上进行认证，然后再将请求转发到后端的微服务，而无须在每个微服务中进行认证。
- 减少了客户端与各个微服务之间的交互次数。


1565921512272

6.2 常用API网关

1565919244569

6.3 搭建网关

1. 访问<https://start.spring.io>
2. 输入和选择相应的参数（在这个示例中，选择的是gateway组件）

1566804460798

3. 点击【Generate the project...】按钮，生成项目
4. 导入项目

此时pom.xml文件内容为：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.1.7.RELEASE</version>
    <relativePath/> <!-- lookup parent from repository -->
  </parent>
  <groupId>com.itcast</groupId>
  <artifactId>cold-gateway</artifactId>
```

```

<version>0.0.1-SNAPSHOT</version>
<name>cold-gateway</name>
<description>Demo project for Spring Boot</description>

<properties>
    <java.version>1.8</java.version>
    <spring-cloud.version>Greenwich.SR2</spring-cloud.version>
</properties>

<dependencies>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-gateway</artifactId>
    </dependency>

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>


<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>${spring-cloud.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>

<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>


</project>

```

导入Idea后，工程如下：

 1566804557539

Spring cloud gateway作为spring boot应用，同时也是一个Eureka服务的客户端，只需要添加Eureka客户端的注解：

 1566804717444

Spring cloud gateway作为API网关，作为微服务架构体系的路由器作用，同时需要解决前端请求跨域问题。

6.3.1 API鉴权

```

public class TokenFilter implements GlobalFilter, Ordered {
    Logger logger= LoggerFactory.getLogger( TokenFilter.class );

    @Override
    public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain
chain) {

        if(!exchange.getRequest().getPath().value().equals("/cold/sys/user/login")){
            String token = exchange.getRequest().getHeaders().getFirst("token");
            if (token == null || token.isEmpty()) {
                logger.info( "token为空..." );
                exchange.getResponse().setStatusCode(HttpStatus.UNAUTHORIZED);
                return exchange.getResponse().setComplete();
            }
        }

        return chain.filter(exchange);
    }

    @Override
    public int getOrder() {
        return -100;
    }
}

```

6.3.2 跨域问题

跨域是指从一个域名的网页去请求另一个域名的资源。

跨域的严格一点的定义是：只要 协议、域名、端口有任何一个的不同，就被当作是跨域。

为什么会存储跨域问题：如果一个网页可以随意地访问另外一个网站的资源，那么就有可能在客户完全不知情的情况下出现安全问题。

如何解决跨域问题：JSONP或CROS，本例中，我们使用cros过滤器解决：

```

package com.itcast.cold.gateway.common;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.cors.CorsConfiguration;
import org.springframework.web.cors.reactive.CorsWebFilter;
import org.springframework.web.cors.reactive.UrlBasedCorsConfigurationSource;
import org.springframework.web.util.pattern.PathPatternParser;

@Configuration
public class CorsConfig {
    @Bean
    public CorsWebFilter corsFilter() {
        CorsConfiguration config = new CorsConfiguration();
        config.addAllowedMethod("*");
        config.addAllowedOrigin("*");
        config.addAllowedHeader("*");
        config.setAllowCredentials(true);    //允许发送cookie的内容
        UrlBasedCorsConfigurationSource source = new
        UrlBasedCorsConfigurationSource(new PathPatternParser());
        source.registerCorsConfiguration("/**", config);
    }
}

```

```
        return new CorsWebFilter(source);
    }
}
```

6.3.2 路由配置

可以通过配置或开发bean来进行网关配置，本例中，我们的网关主要承担着服务请求路由的功能，我们使用配置方式（application.yml）：

```
server:
  tomcat:
    uri-encoding: UTF-8
    max-threads: 1000
    min-spare-threads: 30
  port: 8080
  connection-timeout: 5000ms
  servlet:
    context-path: /cold

eureka:
  client:
    service-url:
      defaultZone: http://eureka.itheima.com:8001/eureka

spring:
  application:
    name: cold-gateway

cloud:
  gateway:
    discovery:
      locator:
        enabled: true
    routes:
      - id: cold-admin-router
        uri: lb://cold-admin
        predicates:
          - Path=/cold/admin/**

      - id: cold-monitor-router
        uri: lb://cold-monitor
        predicates:
          - Path=/cold/device/monitor/**

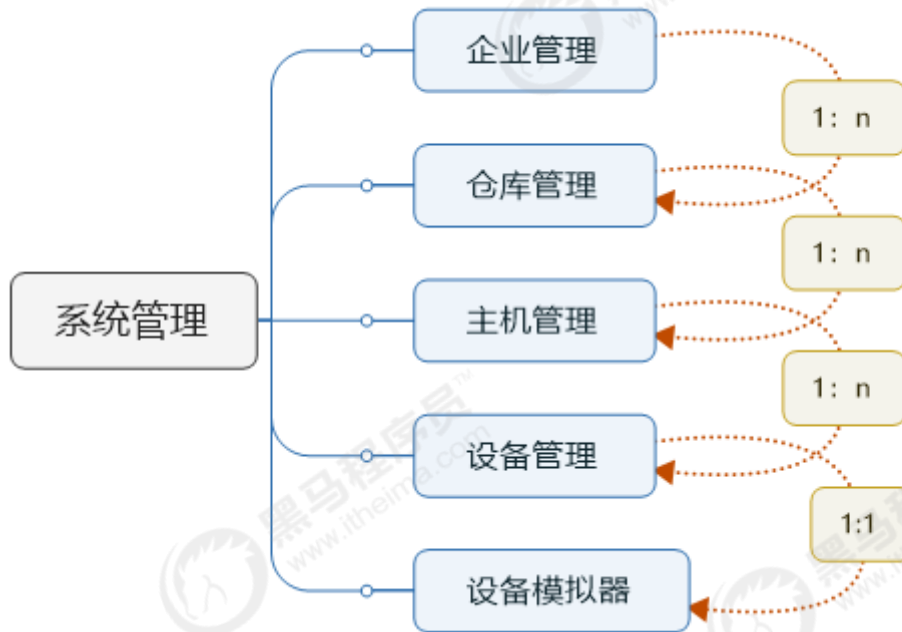
      - id: cold-jobs-router
        uri: lb://cold-jobs
        predicates:
          - Path=/cold/system/schedule/**

      - id: cold-druid-router
        uri: lb://cold-druid
        predicates:
          - Path=/cold/apache-druid/query/**
```

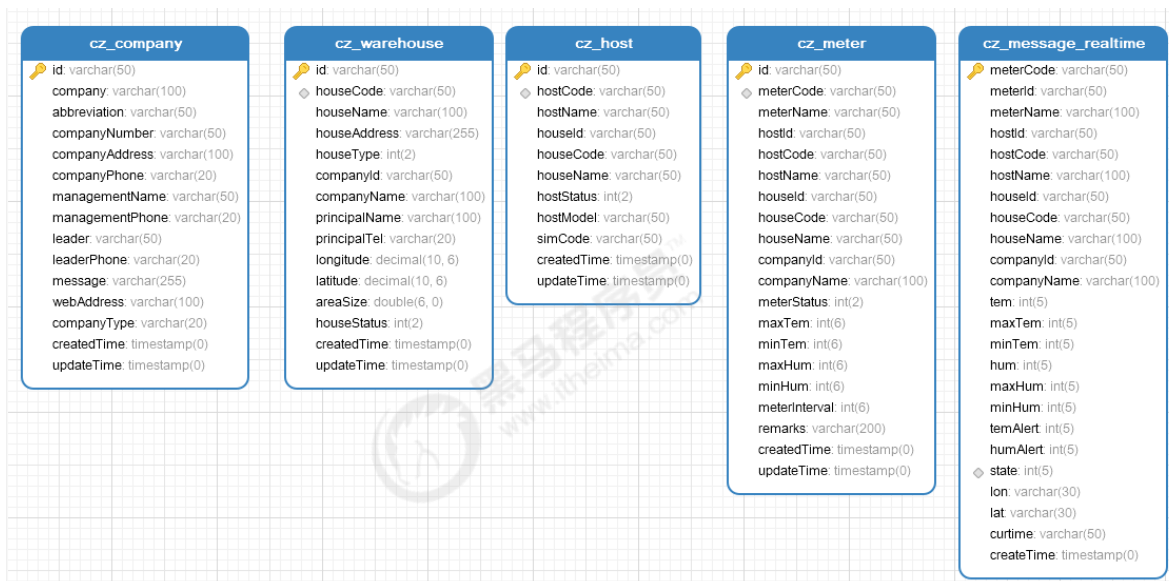
- id: cold-admin-router 路由的唯一id
- uri: lb://cold-admin 指向注册中心的服务，使用lb:// 加上ServiceName，当然也可以通过http://localhost:8090指向，lb是能够进行负载均衡
- predicates: 表示要进行的断言
 - Path= /cold/admin/** 表示path地址，根据url，以 /cold/admin/ 开头的会被转发到cold-admin服务，需要注意的是后面/** 和/* 的区别

7 系统管理 (cold-admin)

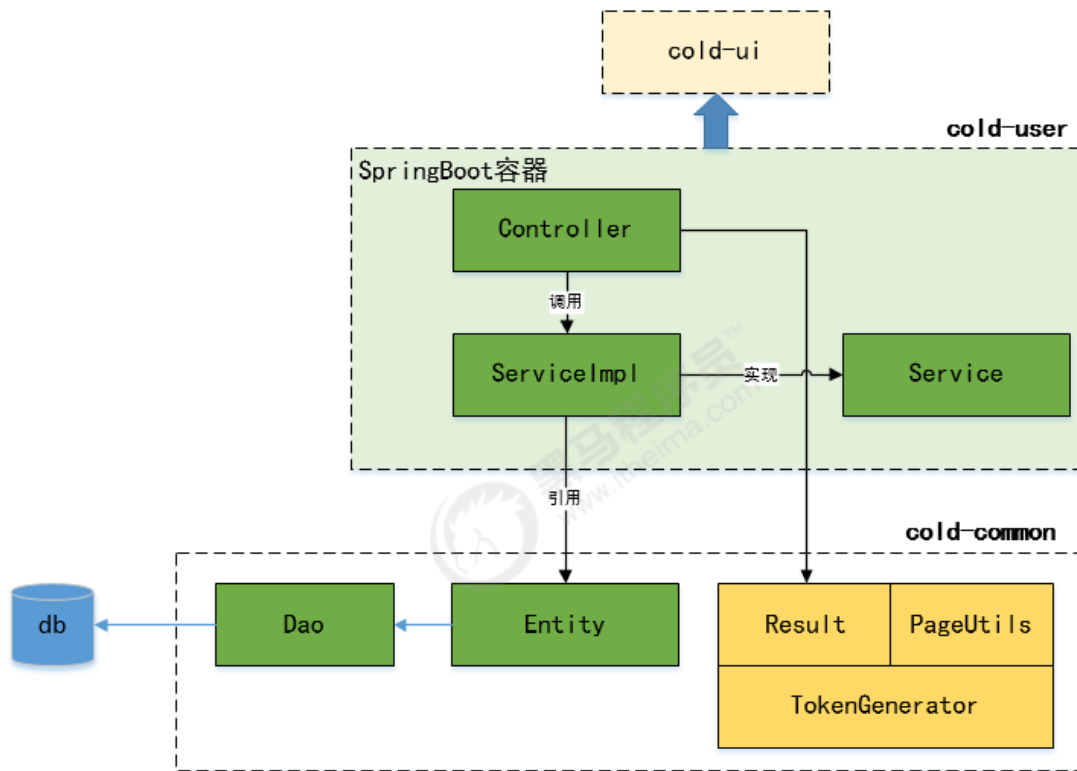
系统管理服务 (cold-admin) 包含如下功能：



数据模型：



代码结构：



实战步骤：

1. 创建数据库，执行导入cold-admin.sql文件，该文件在项目的doc文件夹下
2. 创建spring boot工程，可以使用start.spring.io创建，选择spring boot版本、eureka客户端依赖包
3. pom文件中添加cold-common依赖
4. cold-common工程中根据数据库表添加实体entity、dao
5. 添加controller、service代码
6. 运行调试工程，使用postman调用测试每一个接口
7. 完成以上步骤后，在网关服务的配置中添加cold-admin工程的路由配置

7.1 项目实战 - 企业管理

7.1.1 需求分析

企业管理的业务逻辑比较简单，是针对企业信息的增删改查。

7.1.2 具体实现

```

CREATE TABLE `cold`.`cz_company` (
  `id` varchar(50) CHARACTER SET utf8 COLLATE utf8_general_ci NOT NULL COMMENT '主键',
  `company` varchar(100) CHARACTER SET utf8 COLLATE utf8_general_ci NOT NULL COMMENT '公司名称',
  `abbreviation` varchar(50) CHARACTER SET utf8 COLLATE utf8_general_ci NULL DEFAULT NULL COMMENT '公司简称',
  `companyNumber` varchar(50) CHARACTER SET utf8 COLLATE utf8_general_ci NULL DEFAULT NULL COMMENT '企业编号',
  `companyAddress` varchar(100) CHARACTER SET utf8 COLLATE utf8_general_ci NULL DEFAULT NULL COMMENT '企业地址',
  `companyPhone` varchar(20) CHARACTER SET utf8 COLLATE utf8_general_ci NULL DEFAULT NULL COMMENT '公司电话',
  `managementName` varchar(50) CHARACTER SET utf8 COLLATE utf8_general_ci NULL DEFAULT NULL COMMENT '质量管理员',

```

```

`managementPhone` varchar(20) CHARACTER SET utf8 COLLATE utf8_general_ci NULL
DEFAULT NULL COMMENT '联系电话',
`leader` varchar(50) CHARACTER SET utf8 COLLATE utf8_general_ci NULL DEFAULT
NULL COMMENT '负责人姓名',
`leaderPhone` varchar(20) CHARACTER SET utf8 COLLATE utf8_general_ci NULL
DEFAULT NULL COMMENT '负责人电话',
`message` varchar(255) CHARACTER SET utf8 COLLATE utf8_general_ci NULL DEFAULT
NULL COMMENT '备注',
`webAddress` varchar(100) CHARACTER SET utf8 COLLATE utf8_general_ci NULL
DEFAULT NULL COMMENT '网站',
`companyType` varchar(20) CHARACTER SET utf8 COLLATE utf8_general_ci NULL
DEFAULT NULL COMMENT '企业类型',
`createTime` timestamp(0) NOT NULL DEFAULT CURRENT_TIMESTAMP COMMENT '创建时
间',
`updateTime` timestamp(0) NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP(0) COMMENT '修改时间',
PRIMARY KEY (`id`) USING BTREE
) ENGINE = InnoDB CHARACTER SET = utf8 COLLATE = utf8_general_ci COMMENT = '企业
信息表' ROW_FORMAT = Dynamic;

```

这个功能是一个单表的增删改查功能，代码结构如下：

前端代码：

```

cold-ui
├─ view
│   ├── admin
│   │   └─ company.vue
│   └─ api
│       └─ admin.ts

```

后端代码：

```

# 业务服务，引用cold-common工程
cold-admin
├─ controller
│   ├── CompanyController.java
│   └─ service
│       ├── CompanyService.java
│       └─ impl
│           └─ CompanyServiceImpl.java

# 公共组件，包括实体类、工具类等
cold-common
├─ admin
│   ├── dao
│   │   └─ CompanyDao.java
│   └─ entity
│       └─ CompanyEntity.java

```

实体类：CompanyEntity：

```

package com.itcast.cold.common.admin.entity;

import com.baomidou.mybatisplus.annotation.TableId;
import com.baomidou.mybatisplus.annotation.TableName;

```



```

import java.io.Serializable;
import java.util.Date;
import lombok.Data;

/**
 * 企业信息表
 *
 */
@Data
@TableName("cz_company")
public class CompanyEntity implements Serializable {
    private static final long serialVersionUID = 1L;

    /**
     * 主键
     */
    @TableId
    private String id;
    /**
     * 企业名称
     */
    private String company;
    /**
     * 企业简称
     */
    private String abbreviation;
    /**
     * 企业编号
     */
    private String companynumber;
    /**
     * 公司地址
     */
    private String companyaddress;
    /**
     * 公司电话
     */
    private String companyphone;

    /**
     * 质量管理员
     */
    private String managementname;

    /**
     * 质管员电话
     */
    private String managementphone;

    /**
     * 负责人姓名
     */
    private String leader;

    /**
     * 负责人电话
     */

```

```

private String leaderphone;

/**
 * 网站
 */
private String webaddress;

/**
 * 备注
 */
private String message;

/**
 * 企业类型
 */
private String companytype;

/**
 * 创建时间
 */
private Date createdtime;

/**
 * 修改时间
 */
private Date updatetime;
}

```

注意使用了mybatis plus的注解：`@TableName("cz_company")`，此注解将实体与数据库表关联起来了。

dao (Mapper接口)：

```

package com.itcast.cold.common.admin.dao;

import com.baomidou.mybatisplus.core.mapper.BaseMapper;
import com.itcast.cold.common.admin.entity.CompanyEntity;
import org.apache.ibatis.annotations.Mapper;

/**
 * 企业信息表
 *
 */
@Mapper
public interface CompanyDao extends BaseMapper<CompanyEntity> {

}

```

增删改查操作甚至不需要创建xml文件：

查询分页操作：

```

package com.itcast.cold.admin.service.impl;

import com.baomidou.mybatisplus.core.conditions.query.QueryWrapper;
import com.baomidou.mybatisplus.core.metadata.IPage;

```

```

import com.baomidou.mybatisplus.extension.plugins.pagination.Page;
import com.baomidou.mybatisplus.extension.service.impl.ServiceImpl;
import com.itcast.cold.admin.service.CompanyService;
import com.itcast.cold.common.admin.dao.CompanyDao;
import com.itcast.cold.common.admin.entity.CompanyEntity;
import com.itcast.cold.common.utils.PageUtils;
import org.springframework.stereotype.Service;

import java.util.Map;

@Service("companyService")
public class CompanyServiceImpl extends ServiceImpl<CompanyDao, CompanyEntity>
implements CompanyService {

    @Override
    public PageUtils queryPage(Map<String, Object> params) {
        //分页参数
        int current = params.get("page")== null? 1 :
Integer.valueOf(params.get("page").toString());
        int size = params.get("pagesize") == null? 10 :
Integer.valueOf(params.get("pagesize").toString());

        Page<CompanyEntity> page = new Page<>(current, size);

        //查询条件
        String company = params.get("company")==null ? "":
params.get("company").toString();
        Querywrapper<CompanyEntity> wrapper = new Querywrapper<>();
        wrapper.lambda().like(CompanyEntity::getCompany, company);

        //执行查询
        IPage<CompanyEntity> result = this.page(page, wrapper);

        //返回结构化数据
        return new PageUtils(result);
    }
}

```

当前端页面通过REST接口调用此方法后，返回数据格式如下：

```

{
  "msg": "success",
  "total": 5,
  "code": 0,
  "page": 1,
  "items": [{
    "id": "51282d3e-f499-4f91-b315-c49a1b28fbac",
    "company": "凡越运输",
    "abbreviation": "传智播客",
    "companynumber": null,
    "companyaddress": "北京市南城区一号大院",
    "companyphone": "4007382000",
    "managementname": "王五",
    "managementphone": "13846485047",
    "leader": "张磊",
    "leaderphone": "17398768686",
    "webaddress": "",

```

```
"message": "",
"companytype": "国有",
"createdtime": "2019-07-24 10:52:00",
"updatetime": "2019-07-24 10:52:00"
}, {
  "id": "7636e231-403a-4a8b-b898-80d4a8f56c36",
  "company": "速威运输",
  "abbreviation": "速威运输",
  "companynumber": null,
  "companyaddress": "速威运输",
  "companyphone": "4003729937",
  "managementname": "赵柳",
  "managementphone": "13846485047",
  "leader": "李城建",
  "leaderphone": "17398768686",
  "webaddress": "",
  "message": "",
  "companytype": "私营",
  "createdtime": "2019-07-24 10:39:15",
  "updatetime": "2019-07-24 10:39:15"
}, {
  "id": "98478163-54e4-4124-ab05-83f9e9b6f4c6",
  "company": "申达运输",
  "abbreviation": "申达运输",
  "companynumber": "张三",
  "companyaddress": "申达运输股份有限公司",
  "companyphone": "4007382000",
  "managementname": "李逍遥",
  "managementphone": "18511281822",
  "leader": "风清扬",
  "leaderphone": "18511281822",
  "webaddress": "",
  "message": "",
  "companytype": "私营",
  "createdtime": "2019-07-09 15:19:04",
  "updatetime": "2019-07-23 19:28:21"
}, {
  "id": "9d0cd756-2ee5-4ea5-9405-07864f7c5b46",
  "company": "好万运输",
  "abbreviation": "好万运输",
  "companynumber": "李四",
  "companyaddress": "好万运输",
  "companyphone": "4003729937",
  "managementname": "赵老三",
  "managementphone": "13846485047",
  "leader": "乔峰",
  "leaderphone": "13511281922",
  "webaddress": "",
  "message": "",
  "companytype": "外企",
  "createdtime": "2019-07-22 11:24:23",
  "updatetime": "2019-07-23 19:28:32"
}, {
  "id": "a4130a4e-2706-41fb-87c7-c4521d08e9a8",
  "company": "顺捷运输",
  "abbreviation": "顺捷运输",
  "companynumber": null,
  "companyaddress": "顺捷运输",
```

```

"companyphone": "408201123",
"managementname": "郭靖",
"managementphone": "13911290121",
"leader": "黄蓉",
"leaderphone": "12911290120",
"webaddress": "",
"message": "",
"companytype": "私企",
"createdtime": "2019-07-24 10:31:52",
"updatetime": "2019-07-24 10:31:52"
}
}
}

```

前端页面根据返回的数据，展示公司列表详情：

序号	企业全称	企业简称	质量管理员	联系电话	操作
1	凡越运输	传智播客	王五	13846485047	库房 修改 删除
2	速威运输	速威运输	赵柳	13846485047	库房 修改 删除
3	申达运输	申达运输	李遇遥	18511281822	库房 修改 删除
4	好万运输	好万运输	赵老三	13846485047	库房 修改 删除
5	顺捷运输	顺捷运输	郭靖	13911290121	库房 修改 删除

7.2 项目实战 - 库房管理

7.2.1 需求分析

库房是存储物品的仓库，向下又可细化为库位、货架等更细粒度的仓库计量单位。

7.2.2 表结构

```

CREATE TABLE `cold`.`cz_warehouse` (
  `id` varchar(50) CHARACTER SET utf8 COLLATE utf8_general_ci NOT NULL COMMENT '主键',
  `houseCode` varchar(50) CHARACTER SET utf8 COLLATE utf8_general_ci NOT NULL COMMENT '仓库编码',
  `houseName` varchar(100) CHARACTER SET utf8 COLLATE utf8_general_ci NULL DEFAULT NULL COMMENT '仓库名称',
  `houseAddress` varchar(255) CHARACTER SET utf8 COLLATE utf8_general_ci NULL DEFAULT NULL COMMENT '仓库地址',
  `houseType` int(2) NULL DEFAULT NULL COMMENT '库房类型: 1-冷库, 2-恒温库',
  `companyId` varchar(50) CHARACTER SET utf8 COLLATE utf8_general_ci NULL DEFAULT NULL COMMENT '所属CompanyId',
  `companyName` varchar(100) CHARACTER SET utf8 COLLATE utf8_general_ci NULL DEFAULT NULL COMMENT '公司名称',
  `principalName` varchar(100) CHARACTER SET utf8 COLLATE utf8_general_ci NULL DEFAULT NULL COMMENT '负责人',
  `principalTel` varchar(20) CHARACTER SET utf8 COLLATE utf8_general_ci NULL DEFAULT NULL COMMENT '负责人电话',
  `longitude` decimal(10, 6) NULL DEFAULT NULL COMMENT '经度',
  `latitude` decimal(10, 6) NULL DEFAULT NULL COMMENT '纬度',

```

```

`areaSize` double(6, 0) NULL DEFAULT NULL COMMENT '库房面积',
`houseStatus` int(2) NULL DEFAULT 1 COMMENT '状态: 1-正常,0-空库',
`createTime` timestamp(0) NOT NULL DEFAULT CURRENT_TIMESTAMP COMMENT '创建时间',
`updateTime` timestamp(0) NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP(0) COMMENT '修改时间',
PRIMARY KEY (`id`) USING BTREE,
UNIQUE INDEX `idxCode` (`houseCode`) USING BTREE
) ENGINE = InnoDB CHARACTER SET = utf8 COLLATE = utf8_general_ci COMMENT = '仓库表' ROW_FORMAT = Dynamic;

```

7.2.3 具体实现

参考企业管理

序号	库房类型	库房编号	库房名称	所属公司	负责人	联系电话	库房地址	面积	状态	操作
1	恒温仓	WH1011001	昌平一号仓	凡越运输	张晓七	13612912010		30000	正常	修改 删除
2	冷库	WH1020001	昌平二号仓	好万运输	武王	13622129089		20000	正常	修改 删除
3	恒温仓	WH1030001	昌平三号仓	好万运输	周乾	18922123210		40000	正常	修改 删除
4	冷库	WH1041001	朝阳一号仓	顺捷运输	孙森	13811291120		10000	正常	修改 删除
5	恒温仓	WH3012001	朝阳二号仓	申达运输	吴越	15619210201		3500	正常	修改 删除

7.3 项目实战 - 主机管理

7.3.1 需求分析

主机可以理解为一台电脑，硬件设备可以将数据汇集到主机中，向服务中心提交数据。

7.3.2 表结构

```

CREATE TABLE `cold`.`cz_host` (
  `id` varchar(50) CHARACTER SET utf8 COLLATE utf8_general_ci NOT NULL COMMENT '主键',
  `hostCode` varchar(50) CHARACTER SET utf8 COLLATE utf8_general_ci NOT NULL COMMENT '主机编码',
  `hostName` varchar(50) CHARACTER SET utf8 COLLATE utf8_general_ci NULL DEFAULT NULL COMMENT '主机名称',
  `houseId` varchar(50) CHARACTER SET utf8 COLLATE utf8_general_ci NULL DEFAULT NULL COMMENT '仓库Id',
  `houseCode` varchar(50) CHARACTER SET utf8 COLLATE utf8_general_ci NULL DEFAULT NULL COMMENT '仓库编码',
  `houseName` varchar(50) CHARACTER SET utf8 COLLATE utf8_general_ci NULL DEFAULT NULL COMMENT '仓库名称',
  `hostStatus` int(2) NULL DEFAULT 1 COMMENT '主机状态: 1-正常, 0-停用',
  `hostModel` varchar(50) CHARACTER SET utf8 COLLATE utf8_general_ci NULL DEFAULT NULL COMMENT '设备型号',
  `simCode` varchar(50) CHARACTER SET utf8 COLLATE utf8_general_ci NULL DEFAULT NULL COMMENT 'sim卡号',

```

```

`createTime` timestamp(0) NOT NULL DEFAULT CURRENT_TIMESTAMP COMMENT '创建时间',
`updateTime` timestamp(0) NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP(0) COMMENT '修改时间',
PRIMARY KEY (`id`) USING BTREE,
UNIQUE INDEX `idxCode`(`hostCode`) USING BTREE
) ENGINE = InnoDB CHARACTER SET = utf8 COLLATE = utf8_general_ci COMMENT = '主机表' ROW_FORMAT = Dynamic;

```

7.3.3 具体实现

参考2.1企业管理

序号	主机编号	主机名称	所属仓库	状态	设备型号	sim卡号	操作
1	H100110001	昌平一号仓01	昌平一号仓	正常	DG02931132	63846484	仪表管理 修改 停用 删除 重启
2	H100110002	昌平一号仓02	昌平一号仓	正常	NH08922123	84649472	仪表管理 修改 停用 删除 重启
3	H100110003	昌平一号仓03	昌平一号仓	正常	HD20130212	28129321	仪表管理 修改 停用 删除 重启
4	H100330001	昌平三号仓01	昌平三号仓	正常	HO92831209	87216932	仪表管理 修改 停用 删除 重启
5	H100220001	昌平二号仓01	昌平二号仓	正常	DE23112321	98223123	仪表管理 修改 停用 删除 重启
6	H100220002	昌平二号仓02	昌平二号仓	正常	KL021322121	89213214	仪表管理 修改 停用 删除 重启
7	H100220003	昌平二号仓03	昌平二号仓	正常	HO82120921	53218901	仪表管理 修改 停用 删除 重启

7.4 项目实战 - 仪表管理

7.4.1 需求分析

仪表是具体的硬件设备，可以收集温度、湿度、电量、电压、速度、经纬度等各种类型的数据。

在本项目中，我们假定收集的是温度和湿度，其他类型的指标操作方式类似。在这里我们初步估算一下数据量：

假设100个仓库，每个仓库有10个主机，每个主机连接着10个设备，那么将会有 $100 * 10 * 10 = 10000$ 个仪表设备，如果平均每个仪表设备每分钟提交一次报文数据，那么每天将会有： $60 * 24 * 1万 = 14400$ 万条数据，可以想象仪表设备采集的数据量非常大。

7.4.2 表结构

```

CREATE TABLE `cold`.`cz_meter` (
  `id` varchar(50) CHARACTER SET utf8 COLLATE utf8_general_ci NOT NULL COMMENT '主键ID',
  `meterCode` varchar(50) CHARACTER SET utf8 COLLATE utf8_general_ci NOT NULL COMMENT '设备编码',
  `meterName` varchar(50) CHARACTER SET utf8 COLLATE utf8_general_ci NULL DEFAULT NULL COMMENT '设备名称',
  `hostId` varchar(50) CHARACTER SET utf8 COLLATE utf8_general_ci NULL DEFAULT NULL COMMENT '主机ID',
  `hostCode` varchar(50) CHARACTER SET utf8 COLLATE utf8_general_ci NULL DEFAULT NULL COMMENT '主机编码',
  `hostName` varchar(50) CHARACTER SET utf8 COLLATE utf8_general_ci NULL DEFAULT NULL COMMENT '主机名称',
  `houseId` varchar(50) CHARACTER SET utf8 COLLATE utf8_general_ci NULL DEFAULT NULL COMMENT '仓库Id',

```

```

`houseCode` varchar(50) CHARACTER SET utf8 COLLATE utf8_general_ci NULL
DEFAULT NULL COMMENT '仓库编码',
`houseName` varchar(50) CHARACTER SET utf8 COLLATE utf8_general_ci NULL
DEFAULT NULL COMMENT '仓库名称',
`companyId` varchar(50) CHARACTER SET utf8 COLLATE utf8_general_ci NULL
DEFAULT NULL COMMENT 'companyId',
`companyName` varchar(100) CHARACTER SET utf8 COLLATE utf8_general_ci NULL
DEFAULT NULL COMMENT '公司名称',
`meterStatus` int(2) NULL DEFAULT 1 COMMENT '仪表状态(1-在用,0-停用,2-异常)',
`maxTem` int(6) NULL DEFAULT NULL COMMENT '温度上限',
`minTem` int(6) NULL DEFAULT NULL COMMENT '温度下限',
`maxHum` int(6) NULL DEFAULT NULL COMMENT '湿度上限',
`minHum` int(6) NULL DEFAULT NULL COMMENT '湿度下限',
`meterInterval` int(6) NULL DEFAULT NULL COMMENT '采集间隔',
`remarks` varchar(200) CHARACTER SET utf8 COLLATE utf8_general_ci NULL DEFAULT
NULL COMMENT '备注信息',
`createTime` timestamp(0) NOT NULL DEFAULT CURRENT_TIMESTAMP COMMENT '创建时
间',
`updateTime` timestamp(0) NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP(0) COMMENT '修改时间',
PRIMARY KEY (`id`) USING BTREE,
UNIQUE INDEX `idxCode`(`meterCode`) USING BTREE
) ENGINE = InnoDB CHARACTER SET = utf8 COLLATE = utf8_general_ci ROW_FORMAT =
Dynamic;

```

7.4.3 具体实现

仪表设备的增删改查参考2.1、企业管理。

需要注意的是，在业务场景中，我们在实时计算环节，需要从redis缓存中获得仪表设备的配置信息。

所以在仪表的添加、修改、删除时，需要更新redis数据：

```

/*
 * com.itcast.cold.admin.controller.MeterController
 */

/**
 * 保存仪表
 */
@RequestMapping("/save")
public R save(@RequestBody MeterEntity meter) throws Exception{
    String uuid = UUID.randomUUID().toString();
    meter.setId(uuid);
    meterService.save(meter);

    JedisUtil.Strings strings=JedisUtil.getInstance().new Strings();
    strings.set(meter.getMetercode(), SerializeUtil.serialize(meter));

    return R.ok();
}

/**
 * 修改仪表
 */
@RequestMapping("/update")
public R update(@RequestBody MeterEntity meter) throws Exception{

```



```

        meterService.updateById(meter);

        JedisUtil.getInstance().getJedis().del(meter.getMetercode());

        JedisUtil.Strings strings=JedisUtil.getInstance().new Strings();
        strings.set(meter.getMetercode(), SerializeUtil.serialize(meter));

        return R.ok();
    }

    /**
     * 删除仪表
     */
    @RequestMapping("/delete")
    public R delete(String id){
        meterService.removeById(id);

        JedisUtil.getInstance().getJedis().del(meterService.getById(id).getMetercode());

        return R.ok();
    }
}

```

8 分布式任务 (cold-jobs)

任务调度是指基于给定的时间点，给定的时间间隔或者给定执行次数自动的执行任务。

任务调度是操作系统的重要组成部分，而对于实时的操作系统，任务调度直接影响着操作系统的实时性能。任务调度涉及到多线程并发、运行时间规则定制及解析、线程池的维护等诸多方面的工作。

WEB服务器在接受请求时，会创建一个新的线程服务。但是资源有限，必须对资源进行控制，首先就是限制服务线程的最大数目，其次考虑以线程池共享服务的线程资源，降低频繁创建、销毁线程的消耗；然后任务调度信息的存储包括运行次数、调度规则以及运行数据等。一个合适的任务调度框架对于项目的整体性能来说显得尤为重要。

我们在实际的开发工作中，或多或少的都会用到任务调度这个功能。常见的分布式任务调度框架有：quartz、cronsun、Elastic-job、saturn、Its、TBSchedule、xxl-job等，下面介绍一下quartz框架。

8.1 简介

任务调度框架“Quartz”是一个完全由Java编写的开源作业调度框架，为在Java应用程序中进行作业调度提供了简单却强大的机制。Quartz允许开发人员根据时间间隔来调度作业。它实现了作业和触发器的多对多的关系，还能把多个作业与不同的触发器关联。此外，quartz调度器还支持JTA事务和集群。

8.2 核心概念

- **Job** 表示一个工作，要执行的具体内容。此接口中只有一个方法，如下：

```
void execute(JobExecutionContext context)
```

- **JobDetail** 表示一个具体的可执行的调度程序，Job 是这个可执行调度程序所要执行的内容，另外 JobDetail 还包含了这个任务调度的方案和策略。
- **Trigger** 代表一个调度参数的配置，什么时候去调。
- **Scheduler** 代表一个调度容器，一个调度容器中可以注册多个 JobDetail 和 Trigger。当 Trigger 与 JobDetail 组合，就可以被 Scheduler 容器调用了。

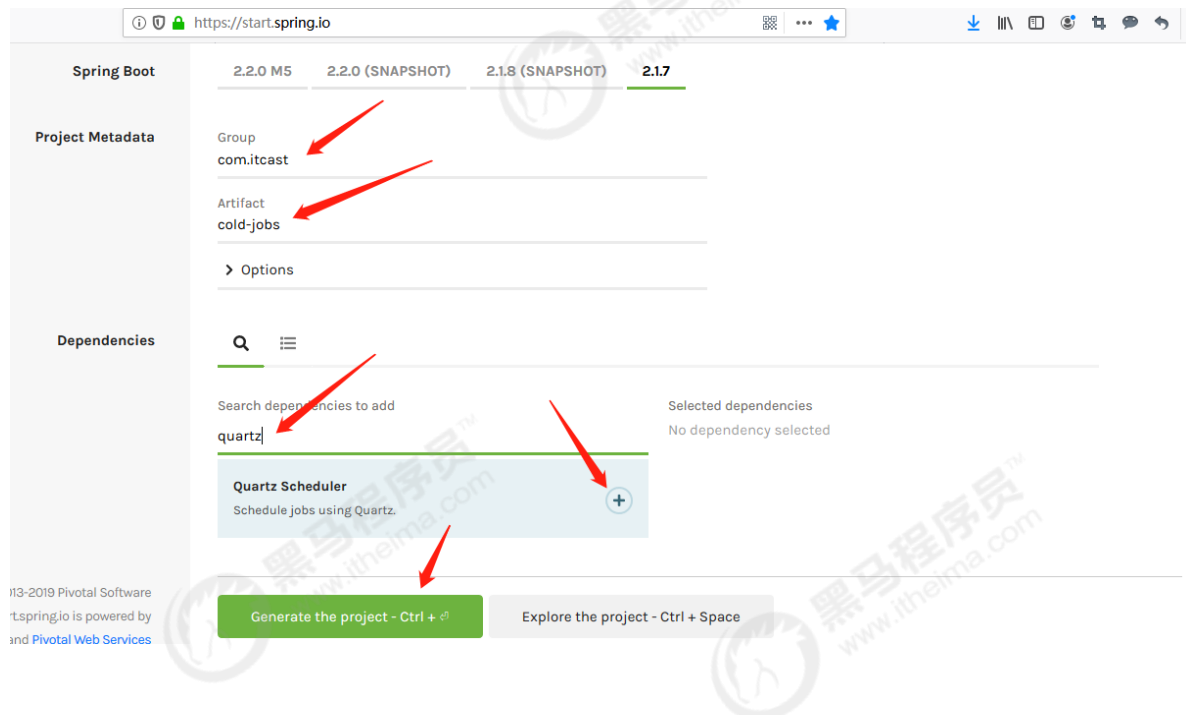
8.3 运行环境

- Quartz 可以运行嵌入在另一个独立式应用程序。
- Quartz 可以在应用程序服务器(或 servlet 容器)内被实例化, 并且参与 XA 事务。
- Quartz 可以作为一个独立的程序运行(其自己的 Java 虚拟机内), 可以通过 RMI 使用。
- Quartz 可以被实例化, 作为独立的项目集群(负载平衡和故障转移功能), 用于作业的执行。

8.4 示例

8.4.1 创建项目

使用Spring Initializr <https://start.spring.io/> 生成项目



8.4.2 pom文件

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
      https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.1.7.RELEASE</version>
    <relativePath/> <!-- lookup parent from repository -->
  </parent>
  <groupId>com.itcast</groupId>
  <artifactId>cold-jobs</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>cold-jobs</name>
  <description>Demo project for Spring Boot</description>

  <properties>
    <java.version>1.8</java.version>
  </properties>
```

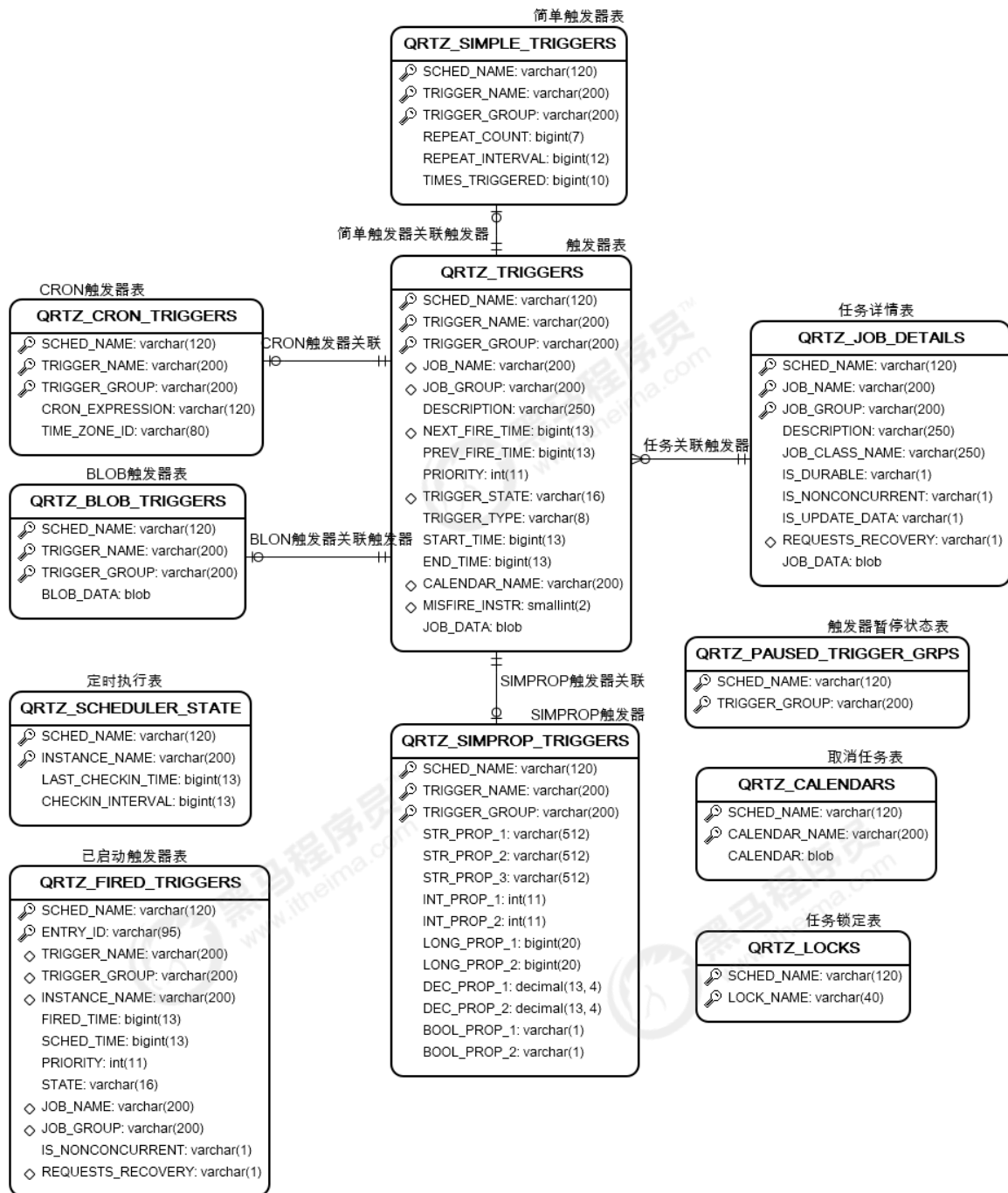
```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-quartz</artifactId>
  </dependency>

  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>

<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>

</project>
```

8.4.3 数据结构



sql文件在: 工程目录/doc/sql/cold-jobs.sql

8.4.4 核心类

配置类：

```
package com.itcast.cold.jobs.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.scheduling.quartz.SchedulerFactoryBean;

import javax.sql.DataSource;
import java.util.Properties;

/**
 * 定时任务配置
 */
```

```

*
*/
@Configuration
public class ScheduleConfig {

    @Bean
    public SchedulerFactoryBean schedulerFactoryBean(DataSource dataSource) {
        SchedulerFactoryBean factory = new SchedulerFactoryBean();
        factory.setDataSource(dataSource);

        //quartz参数
        Properties prop = new Properties();
        prop.put("org.quartz.scheduler.instanceName", "ItcastScheduler");
        prop.put("org.quartz.scheduler.instanceId", "AUTO");
        //线程池配置
        prop.put("org.quartz.threadPool.class",
"org.quartz.simpl.SimpleThreadPool");
        prop.put("org.quartz.threadPool.threadCount", "20");
        prop.put("org.quartz.threadPool.threadPriority", "5");
        //JobStore配置
        prop.put("org.quartz.jobStore.class",
"org.quartz.impl.jdbcjobstore.JobStoreTX");
        //集群配置
        prop.put("org.quartz.jobStore.isClustered", "true");
        prop.put("org.quartz.jobStore.clusterCheckinInterval", "15000");
        prop.put("org.quartz.jobStore.maxMisfiresToHandleAtATime", "1");

        prop.put("org.quartz.jobStore.misfireThreshold", "12000");
        prop.put("org.quartz.jobStore.tablePrefix", "QRTZ_");
        prop.put("org.quartz.jobStore.selectWithLocksSQL", "SELECT * FROM
{0}LOCKS UPDLOCK WHERE LOCK_NAME = ?");

        factory.setQuartzProperties(prop);

        factory.setSchedulerName("ItcastScheduler");
        //延时启动
        factory.setStartupDelay(30);

        factory.setApplicationContextSchedulerContextKey("applicationContextKey");
        //可选，QuartzScheduler 启动时更新已存在的Job，这样就不用每次修改targetObject后
        删除qrtz_job_details表对应记录了
        factory.setOverwriteExistingJobs(true);
        //设置自动启动，默认为true
        factory.setAutoStartup(true);

        return factory;
    }
}

```

定时任务类：

```

package com.itcast.cold.jobs.utils;

import com.itcast.cold.common.job.entity.ScheduleJobEntity;
import com.itcast.cold.common.job.entity.ScheduleJobLogEntity;
import com.itcast.cold.common.utils.SpringContextUtils;
import com.itcast.cold.jobs.service.ScheduleJobLogService;

```

```

import org.apache.commons.lang.StringUtils;
import org.quartz.JobExecutionContext;
import org.quartz.JobExecutionException;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.scheduling.quartz.QuartzJobBean;

import java.lang.reflect.Method;
import java.util.Date;

/**
 * 定时任务
 *
 */
public class ScheduleJob extends QuartzJobBean {
    private Logger logger = LoggerFactory.getLogger(getClass());

    @Override
    protected void executeInternal(JobExecutionContext context) throws
    JobExecutionException {
        ScheduleJobEntity scheduleJob = (ScheduleJobEntity)
        context.getMergedJobDataMap()
            .get(ScheduleJobEntity.JOB_PARAM_KEY);

        //获取spring bean
        ScheduleJobLogService scheduleJobLogService = (ScheduleJobLogService)
        SpringContextUtils.getBean("scheduleJobLogService");

        //数据库保存执行记录
        ScheduleJobLogEntity log = new ScheduleJobLogEntity();
        log.setJobId(scheduleJob.getJobId());
        log.setBeanName(scheduleJob.getBeanName());
        log.setParams(scheduleJob.getParams());
        log.setCreateTime(new Date());

        //任务开始时间
        long startTime = System.currentTimeMillis();

        try {
            //执行任务
            logger.debug("任务准备执行, 任务ID: " + scheduleJob.getJobId());

            Object target =
            SpringContextUtils.getBean(scheduleJob.getBeanName());
            Method method = target.getClass().getDeclaredMethod("run",
            String.class);
            method.invoke(target, scheduleJob.getParams());

            //任务执行总时长
            long times = System.currentTimeMillis() - startTime;
            log.setTimes((int)times);
            //任务状态    0: 成功    1: 失败
            log.setStatus(0);

            logger.debug("任务执行完毕, 任务ID: " + scheduleJob.getJobId() + " 总共
            耗时: " + times + "毫秒");
        }
    }
}

```

```

    } catch (Exception e) {
        logger.error("任务执行失败, 任务ID: " + scheduleJob.getJobId(), e);

        //任务执行总时长
        long times = System.currentTimeMillis() - startTime;
        log.setTimes((int)times);

        //任务状态    0: 成功    1: 失败
        log.setStatus(1);
        log.setError(StringUtils.substring(e.toString(), 0, 2000));
    } finally {
        scheduleJobLogService.save(log);
    }
}
}
}

```

定时任务的工具类：

```

package com.itcast.cold.jobs.utils;

import com.itcast.cold.common.exception.RRException;
import com.itcast.cold.common.job.entity.ScheduleJobEntity;
import com.itcast.cold.common.utils.Constant;
import org.quartz.*;

/**
 * 定时任务工具类
 *
 */
public class ScheduleUtils {
    private final static String JOB_NAME = "TASK_";

    /**
     * 获取触发器key
     */
    public static TriggerKey getTriggerKey(Long jobId) {
        return TriggerKey.triggerKey(JOB_NAME + jobId);
    }

    /**
     * 获取jobkey
     */
    public static JobKey getJobKey(Long jobId) {
        return JobKey.jobKey(JOB_NAME + jobId);
    }

    /**
     * 获取表达式触发器
     */
    public static CronTrigger getCronTrigger(Scheduler scheduler, Long jobId) {
        try {
            return (CronTrigger) scheduler.getTrigger(getTriggerKey(jobId));
        } catch (SchedulerException e) {
            throw new RRException("获取定时任务CronTrigger出现异常", e);
        }
    }
}

```

```

/**
 * 创建定时任务
 */
public static void createScheduleJob(Scheduler scheduler, ScheduleJobEntity
scheduleJob) {
    try {
        //构建job信息
        JobDetail jobDetail =
JobBuilder.newJob(ScheduleJob.class).withIdentity(getJobKey(scheduleJob.getJobId
()))).build();

        //表达式调度构建器
        CronScheduleBuilder scheduleBuilder =
CronScheduleBuilder.cronSchedule(scheduleJob.getCronExpression())
                .withMisfireHandlingInstructionDoNothing();

        //按新的cronExpression表达式构建一个新的trigger
        CronTrigger trigger =
TriggerBuilder.newTrigger().withIdentity(getTriggerKey(scheduleJob.getJobId()))
                .withSchedule(scheduleBuilder).build();

        //放入参数，运行时的方法可以获取
        jobDetail.getJobDataMap().put(ScheduleJobEntity.JOB_PARAM_KEY,
scheduleJob);

        scheduler.scheduleJob(jobDetail, trigger);

        //暂停任务
        if(scheduleJob.getStatus() ==
Constant.ScheduleStatus.PAUSE.getValue()){
            pauseJob(scheduler, scheduleJob.getJobId());
        }
    } catch (SchedulerException e) {
        throw new RRException("创建定时任务失败", e);
    }
}

/**
 * 更新定时任务
 */
public static void updateScheduleJob(Scheduler scheduler, ScheduleJobEntity
scheduleJob) {
    try {
        TriggerKey triggerKey = getTriggerKey(scheduleJob.getJobId());

        //表达式调度构建器
        CronScheduleBuilder scheduleBuilder =
CronScheduleBuilder.cronSchedule(scheduleJob.getCronExpression())
                .withMisfireHandlingInstructionDoNothing();

        CronTrigger trigger = getCronTrigger(scheduler,
scheduleJob.getJobId());

        //按新的cronExpression表达式重新构建trigger
        trigger =
trigger.getTriggerBuilder().withIdentity(triggerKey).withSchedule(scheduleBuilde
r).build();

```



```

        //参数
        trigger.getJobDataMap().put(ScheduleJobEntity.JOB_PARAM_KEY,
scheduleJob);

        scheduler.rescheduleJob(triggerKey, trigger);

        //暂停任务
        if(scheduleJob.getStatus() ==
Constant.ScheduleStatus.PAUSE.getValue()){
            pauseJob(scheduler, scheduleJob.getJobId());
        }

    } catch (SchedulerException e) {
        throw new RRException("更新定时任务失败", e);
    }
}

/**
 * 立即执行任务
 */
public static void run(Scheduler scheduler, ScheduleJobEntity scheduleJob) {
    try {
        //参数
        JobDataMap dataMap = new JobDataMap();
        dataMap.put(ScheduleJobEntity.JOB_PARAM_KEY, scheduleJob);

        scheduler.triggerJob(getJobKey(scheduleJob.getJobId()), dataMap);
    } catch (SchedulerException e) {
        throw new RRException("立即执行定时任务失败", e);
    }
}

/**
 * 暂停任务
 */
public static void pauseJob(Scheduler scheduler, Long jobId) {
    try {
        scheduler.pauseJob(getJobKey(jobId));
    } catch (SchedulerException e) {
        throw new RRException("暂停定时任务失败", e);
    }
}

/**
 * 恢复任务
 */
public static void resumeJob(Scheduler scheduler, Long jobId) {
    try {
        scheduler.resumeJob(getJobKey(jobId));
    } catch (SchedulerException e) {
        throw new RRException("暂停定时任务失败", e);
    }
}

/**
 * 删除定时任务
 */

```

```

public static void deleteScheduleJob(Scheduler scheduler, Long jobId) {
    try {
        scheduler.deleteJob(getJobKey(jobId));
    } catch (SchedulerException e) {
        throw new RRException("删除定时任务失败", e);
    }
}
}

```

8.4.5 任务接口

任务接口：

```

package com.itcast.cold.jobs.task;

/**
 * 定时任务接口，所有定时任务都要实现该接口
 *
 *
 */
public interface ITask {

    /**
     * 执行定时任务接口
     *
     * @param params 参数，多参数使用JSON数据
     */
    void run(String params);
}

```

业务功能上提供了定时任务的增删改查和任务执行的日志：

ID	bean名称	方法名称	参数	cron表达式	备注	状态	操作
52	deviceM402003001		M402003001	/20 * * * * *	每20秒执行一次	正常	修改 删除 暂停 恢复 立即执行
51	deviceM402002002		M402002002	/55 * * * * *	每55秒执行一次	正常	修改 删除 暂停 恢复 立即执行
50	deviceM402002001		M402002001	0 /1 * * * * *	每分钟执行一次	正常	修改 删除 暂停 恢复 立即执行
49	deviceM402001002		M402001002	/30 * * * * *	每30秒执行一次	正常	修改 删除 暂停 恢复 立即执行
48	deviceM402001001		M402001001	/40 * * * * *	每40秒执行一次	正常	修改 删除 暂停

注意quartz有多种触发器，我们使用的是cron触发器：

* bean名称

deviceM101001002

* 方法名称

run

参数

M101001002

* cron表达式

0 /1 * * * ? *

备注

每1分钟执行一次

取消

确定

8.4.6 CRON表达式

Cron表达式是一个**字符串**，字符串以5或6个空格隔开，分开 6或*7个域，每一个域代表一个含义,Cron有如下两种语法格式：

Seconds Minutes Hours DayofMonth Month DayofWeek Year

或

Seconds Minutes Hours DayofMonth Month DayofWeek

每一个域可出现的字符如下：

Seconds	可出现： - * / ，四个字符，有效范围为0-59的整数
Minutes	可出现： - * / ，四个字符，有效范围为0-59的整数
Hours	可出现： - * / ，四个字符，有效范围为0-23的整数
DayofMonth	可出现： - * / ， ? L W C八个字符，有效范围为1-31的整数
Month	可出现： - * / ，四个字符，有效范围为1-12的整数或JAN-DEC
Dayofweek	可出现： - * / ， ? L C #四个字符，有效范围为1-7的整数或SUN-SAT两个范围。1表示星期天，2表示星期一， 依次类推

Year 可出现： - * / ，四个字符，有效范围为1970-2099年

字段	允许值	允许的特殊字符
秒	0-59	, - * /
分	0-59	, - * /
小时	0-23	, - * /
日期	1-31	, - * ? / L W C
月份	1-12 或者 JAN-DEC	, - * /
星期	1-7 或者 SUN-SAT	, - * / ? L C #
年（可选）	留空， 1970-2099	, - * /

示例

0 0 10,14,16 * * ?

每天上午10点，下午2点，4点

0 0/30 9-17 * * ?

朝九晚五工作时间内每半小时

0 0 12 ? * WED	表示每个星期三中午12点
0 0 12 * * ?	每天中午12点触发
0 15 10 ? * *	每天上午10:15触发
0 15 10 * * ?	每天上午10:15触发
0 15 10 * * ? *	每天上午10:15触发
0 15 10 * * ? 2005	2005年的每天上午10:15触发
0 * 14 * * ?	在每天下午2点到下午2:59期间的每1分钟触发
0 0/5 14 * * ?	在每天下午2点到下午2:55期间的每5分钟触发
0 0/5 14,18 * * ?	在每天下午2点到2:55期间和下午6点到6:55期间的每5分钟触发
0 0-5 14 * * ?	在每天下午2点到下午2:05期间的每1分钟触发
0 10,44 14 ? 3 WED	每年三月的星期三的下午2:10和2:44触发
0 15 10 ? * MON-FRI	周一至周五的上午10:15触发
0 15 10 15 * ?	每月15日上午10:15触发
0 15 10 L * ?	每月最后一日的上午10:15触发
0 15 10 ? * 6L	每月的最后一个星期五上午10:15触发
0 15 10 ? * 6L 2002-2005	2002年至2005年的每月的最后一个星期五上午10:15触发
0 15 10 ? * 6#3	每月的第三个星期五上午10:15触发

9 总结

今天我们学习了以下几块内容：

- 1、如何快速搭建spring boot应用
- 2、基于Eureka服务注册中心
- 3、spring cloud gateway网关服务
- 4、前后端分离的开发规范
- 5、用户和系统管理服务
- 6、了解了分布式任务调度框架和重点介绍了quartz框架