目录

一. 课程设计的目的、内容

(1) 根据课堂教学所讲授的数据库系统的设计过程和开发方法，在《数据库系统原理》课程实验 已经建立的数据库基础上，采用： GaussDB 数据库系统平台（如 openGauss, GaussDB(MySQL)，或 PostgreSQL、MySQL） Java、Python、C、C++、C#语言

(2) 现有 Web 开发框架和工具，如 Springboot + VUE、Springboot + freemarker 模板，或其它开发工具（如 Qt），设计开发具有（1）三层 B/S 结构、或（2）两层 C/S 架构的数据库应用系统——LTE 网络干扰分析系统，支持对数据库系统中 LTE 网络配置数据、KPI 指标数据和 PRB 干扰数据的查询，并通过 MRO 测量报告数据计算主邻小区间的干扰，集成网络分析软件分析网络干扰结构，对 XML 形式的 MRO 测量数据进行解析。

二. 系统需求分析

1. 系统界面

(1) 具有登录界面，可以实现用户的登录和注册功能，用户通过口令、密码登录系统。

(2) 具有一级、二级菜单。一级菜单对应系统管理、用户管理、数据管理、业务查询、业务分析五个主要功能。二级菜单项对应上述五个功能下的具体细分业务功能。

2. 用户管理

(1) 用户分为系统管理员、普通用户两大类。

(2) 管理员可以查看用户注册信息，添加删除普通用户，查看数据库信息和修改数据库参数。

(3) 普通用户可以自己注册后由管理员审批通过，或者由管理员分配账号。

3. 系统管理

(1) 可以查看数据库连接、后台数据库服务器及数据库的配置信息。

(2) 可以修改数据库连接时长、数据库缓冲区大小等参数。

4. 数据管理

(1) 可以批量导入网络配置信息、KPI 指标信息、PRB 干扰信息和 MRO 数据。
(2) 可以在用户界面上导出指定表。

5. 业务查询

(1) 小区配置信息查询：从 tbCell 表中查询小区信息。
(2) 基站 eNodeB 信息查询：从 tbCell 表中查询基站所属全部小区信息。
(3) 小区 KPI 指标信息查询：从 tbKPI 表中查询网元某个时间段（天级）某个属性值的变化情况。
(4) PRB 信息统计与查询：根据表“优化区 17 日-19 日每 PRB 干扰查询-15 分钟”，统计小时级的 PRB 干扰数据，之后根据小时级的 PRB 干扰数据表查询网元某个时间段（小时级）某个属性值的变化情况。

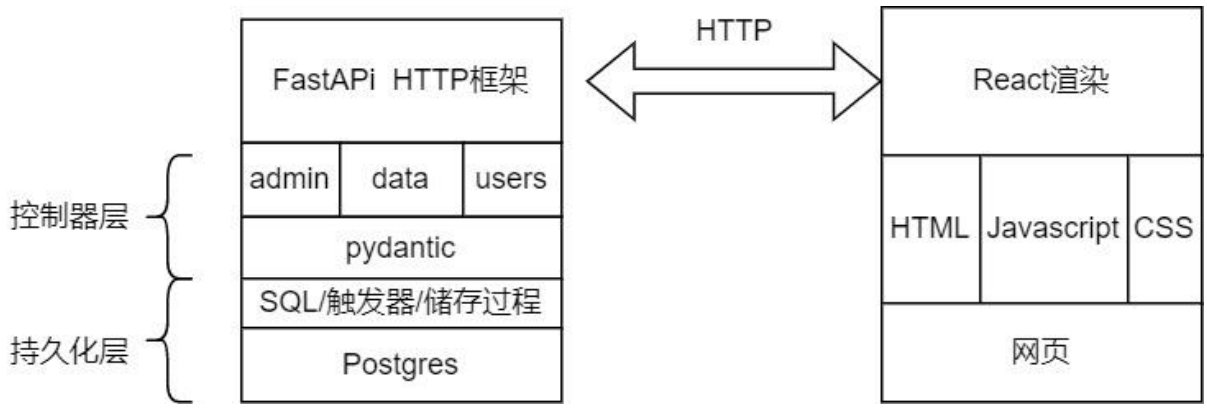
6. 业务处理

(1) 主邻小区 C2I 干扰分析：根据表 tbMROData 生成新表 tbC2Inew。
(2) 重叠覆盖干扰小区三元组分析：根据表 tbC2Inew，找出所有符合特定条件的小区三元组<a, b, c>，并生成新表 tbC2I3。
(3) MRO/MRE 测量数据解析：MRO 数据入库，XML 数据解析，表结构和索引设计。
(4) 网络干扰结构分析：干扰数据预处理，网络可视化。

三. 系统设计

1. 层次结构设计

本项目采用 Python FastApi + React 框架实现前后端分离的 B/S 架构，数据库采用的是 postgres 14。我们没有使用 ORM 框架，所有查询、删除、更改都是手写的 SQL 语句。在数据校验、序列化和反序列化上面，我们使用的与 Fastapi 配套的 pydantic 框架。



2. 环境说明

2.1 概览

安装环境：Win10/11 或 Ubuntu 20.04
运行环境：服务器运行在 Win10/11 或 Ubuntu 20.04 上，客户端使用浏览器
数据库管理系统： Postgres 11+
开发工具： Vscode, Pycharm
语言： 前端使用 Javascript(React 框架)，后端使用 Python3.8
通信协议：HTTP（前端使用 Axios 库，后端使用 Fastapi 框架,uvicorn 作为守护服务器）
数据库访问： 后端使用 asyncpg（ODBC)访问数据库，使用 pydantic 校验和转换数据库数据为 Python 对象

2.2 使用库和框架的具体说明

React
react 是 facebook 团队开发的一款前端框架。它的优势在于使用了虚拟 DOM 进行页面逻辑修改，对于 DOM 的修改和操作都只在虚拟 DOM 中进行，修改完毕之后再一次性同步到页面真实 DOM 中。国内使用 React 的公司：支付宝，淘宝

Axios
Axios，是一个基于 promise 的网络请求库，作用于 node.js 和浏览器中，它是 isomorphic 的(即同一套代码可以运行在浏览器和 node.js 中)。在服务端它使用原生 node.js http 模块，而在客户端 (浏览端) 则使用 XMLHttpRequest。

Fastapi
FastAPI 是一个现代的，快速（高性能）python web 框架。基于标准的 python 类型提示，使用 python3.6+构建 API 的 Web 框架。

Asyncpg

asyncpg 是一个专为 PostgreSQL 和 Python /asyncio 设计的数据库接口库。

Pydantic
pydantic 在运行时强制执行类型提示，并在数据无效时提供用户友好的错误。
定义数据应该如何在纯粹的、规范的 python 中；并使用 pydantic 对其进行验证。

2.3 服务器运行环境的配置

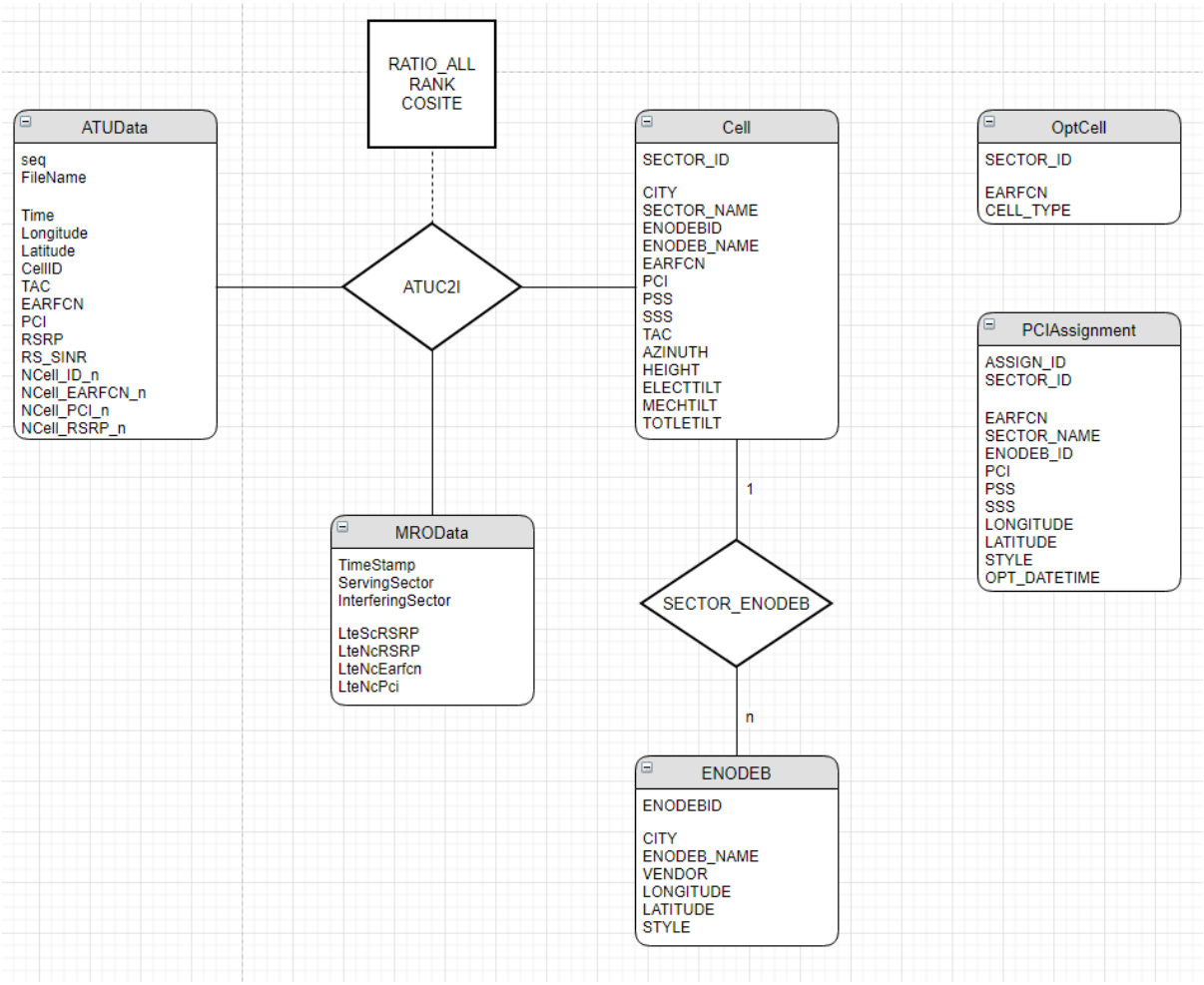
1. 安装必要的软件：Python3.6+, git ,Postgres 11+
2. 首先，使用 git 工具 clone 本项目（https://github.com/ruiqurm/bupt_database_project）
3. (optional) 安装 Python 虚拟环境
4. 打开控制台，切换到项目根目录下。
5. 输入 `pip install -r requirements.txt` ，安装必要的库
6. 输入 `createdb.exe tb` 创建数据库
7. 输入
   ```
   psql -U postgres -d tb -f "src\sql\1_create_table.sql"
   psql -U postgres -d tb -f "src\sql\2_index.sql"
   psql -U postgres -d tb -f "src\sql\3_trigger.sql"
   ```
   创建数据表和约束
8. 创建 config.json,配置 username,password 等字段，用于登录数据库。
9. `uvicorn src.main:app --reload` 在 8000 端口启动服务器

2.4 前端运行环境的配置

1. 安装必要的软件：git, NodeJs
2. git clone 前端项目（https://github.com/The3yx/lte-system）
3. 在根目录下，运行 `npm install` 安装必要库
4. `npm start` 在本地启动一个网页环境。

3. 数据库设计

3.1 E-R 图设计



3.2 关系表设计

1、小区/基站工参表 tbCell

定义：记录小区的有关信息。

范式：属于第二范式。

2、小区一阶邻区关系表 tbAdjCell：

定义：TD-LTE 网络中，如果主小区 S_SECTOR_ID 与邻小区 N_SECTOR_ID 地理上相邻、小区覆盖区域有重叠，则可以定义从主小区 S_SECTOR_ID 到邻小区 N_SECTOR_ID 的邻区关系。在 LTE 网络中，只有定义了小区间的一阶邻区关系后，移动用户才能从主小区 S_SECTOR_ID 向邻小区 N_SECTOR_ID 进行切换。该关系是单向的。

范式：属于第二范式。

3、二阶（同频）邻区关系表 tbSecAdjCell

定义：如果小区 B 是小区 A 的一阶邻区关系，小区 C 是小区 A 的一阶邻区关系，则 B 与 C 互为二阶邻区。

范式：属于第二范式。

4、优化小区/保护带小区表 tbOptCell

定义：本表定义了 TD-LTE 网络中需要进行优化调整（如 PCI、功率、参数调整）的 TD-LTE 优化小区的集合。该表一般是 tbCell 的子集。

范式：属于第三范式。

5、小区 PCI 优化调整结果表 tbPCIAssignment

定义：本表存放优化小区表 tbOptCell 中的各个优化小区的 PCI 优化调整结果。主键为 ASSIGN_ID, SECTOR_ID。

范式：属于第二范式。

6、路测 ATU 数据表 tbATUData

定义：在每个测试点上，测试终端会记录在当前路测点位置上收到的来自 1 个主服务小区（以 CellID 来标识）和最多 6 个邻小区的小区标识（表示为 NCell_ID_1, …, NCell_ID_6）、频点 EARFCN、PCI 和参考信号接收功率 RSRP、信噪比 SINR、测试时间、测试点经纬度等信息，组成一条针对本测试点的测量报告。

范式：属于第三范式。

7、路测 ATU C2I 干扰矩阵表 tbATUC2I

定义：该表由使用路测 ATU 数据计算得到的路测干扰组成。

范式：属于第三范式。

8、路测 ATU 切换统计矩阵 tbATUHandOver

定义：记录了在 ATU 路测数据中，统计得到的从源小区 SECTOR_ID 向目标小区 NCELL_ID 的切换的总次数。

范式：属于第三范式。

9、MRO 测量报告数据表 tbMROData：

定义：记录使用 MRO 测量报告形式测量的来自服务小区/主小区的信号强度 LteScRSRP，以及多个周边邻小区的接收信号强度 LteNcRSRP、频点 LteNcEarfcn 和物理小区标识 LteNcPci。

范式：属于第二范式。

10、基于 MR 测量报告的干扰分析表 tbC2I

定义：根据 MRO 报告，记录 C2I 干扰概率、平均值、标准差。

范式：属于第三范式。

11、小区切换统计性能表 tbHandOver

定义：主键是 SCELL，NCELL。根据源小区和目的小区 ID 记录切换尝试次数、成功次数和成功率。

范式：属于第二范式。

12、优化小区 2020/07/17-2020/07/19KPI 指标统计表 tbKPI

定义：主键是小区名称 SECTOR_NAME 和起始时间 StartTime。记录 RCC 连接建立完成次数、请求次数和成功率。

范式：属于第三范式。

13、tbPRB-表 13 优化区 17 日-19 日每 PRB 干扰查询

定义：主键是小区名称和起始时间。记录每个 PRB 接收到的干扰噪音。

范式：属于第二范式。

14、tbCell_traffic——57 个小区 2019-2020 年一年的小时级话务数据

定义：主键是 STADATE（统计日期）、STATIME（统计时间）和 SECTORID（小区标识）。记录小区的话务数据。

范式：属于第二范式。

3.3 关系表结构

tbCell:

字段名称 /属性名称	字段中文名称	数据类型	数据取值范围， 完整性/约束说明
CITY	城市/地区名称	varchar(255)	可为空
SECTOR_ID	小区 ID	varchar(50)	主键

SECTOR_NAME	小区名称	varchar(255)	not null
ENODEBID	基站 ID	int	not null，小区所属基站 eNodeB 的标识
ENODEB_NAME	基站名称	varchar(255)	not null
EARFCN	小区配置的频点编号	int	not null, 每个小区只能有 1 个频点。取值 {37900, 38098,38400,38950,39148,…}
PCI	物理小区标识(PHYCELLID)	int	取值{0,1,…,503} PCI= 3*SSS + PSS 定义本表时，加入约束 check(PHYCELLID between 0 and 503)
PSS	主同步信号标识	int	取值{0,1,2}； 可为空。数据导入时，由触发器根据 PHYCELLID 计算获得 PSS=PHYCELLID mod3
SSS	辅同步信号标识	int	取值{0,1,2,…,167}； 可为空。数据导入时，由触发器根据 PHYCELLID 计算获得
TAC	跟踪区编码	int	
VENDOR	设备厂家	varchar(255)	取值华为、中兴、诺西、爱立信、贝尔、大唐等
LONGITUDE	小区所属基站的经度	float	not null; 数值=度+分/60+秒/3600。精确到浮点型点后 5 位。正数表示东经。-180.00000 ~ 180.00000

LATITUDE	小区所属基站的纬度	float	not null; 以“度”为单位。数值=度+分/60+秒/3600。 精确到浮点型点后 5 位。正数表示北纬-90.00000 ~ 90.00000
STYLE	基站类型	varchar(255)	取值{宏站，室内，室外，…}，
AZIMUTH	小区天线方位角	float	not null，单位：度
HEIGHT	小区天线高度	float	单位：m
ELECTTILT	小区天线电下倾角	float	单位：度
MECHTILT	小区天线机械下倾角	float	单位：度
TOTLETILT	总下倾角	float	not null; TOTLETILT= ELECTTILT+ MECHTILT

tbMROData:

字段名称	字段中文名称	数据类型	数据取值范围 说明/完整性约束
TimeStamp	测量时间点	varchar(30)	主属性
ServingSector	服务小区/主小区 ID	varchar(50)	主属性
InterferingSector	干扰小区 ID	varchar(50)	主属性
LteScRSRP	服务小区参考信号接收功率 RSRP	float	
LteNcRSRP	干扰小区参考信号	float	

	接收功率 RSRP		
LteNcEarfcn	干扰小区频点	int	
LteNcPci	干扰小区 PCI	smallint	

tbKPI:

字段名称	字段中文名称	数据类型	数据取值范围 说明/完整性约束
StartTime	起始时间	date	
ENODEB_NAME	网元/基站名称	varchar(255)	外键
SECTOR_DESCRIPTION	小区描述	varchar(255)	not null
SECTOR_NAME	小区名称	varchar(255)	主键
RCCConnSUCC	RCC 连接建立完成次数	int	
RCCConnATT	RCC 连接请求次数	int	
RCCConnRATE	RCC 连接成功率	float	缺省值 null 1)当 RCCConnATT 不为零时，RCCConnRATE = RCCConnSUCC/RCCConnATT 2)当 RCCConnATT=0 时，RCCConnRATE 为 null

tbPRB:

字段名称	字段中文名称	数据类型	数据取值范围 说明/完整性约束
StartTime	起始时间	timestamp	

ENODEB_NAME	网元/基站名称	varchar(255)	外键
SECTOR_DESCRIPTION	小区描述	varchar(255)	not null
SECTOR_NAME	小区名称	varchar(255)	主键
AvgNoise0~AvgNoise99	第 0~99 个 PRB 上检测到的干扰噪声的平均值	float	

3.4 函数及触发器设计

函数设计：

在产生 tbc2i 新表时，需要根据小区对的数据进行一定的计算，得到新表的数据。新表中的均值、标准差直接通过聚集函数 avg 和 stddev 来实现：

```
command = f"""
with tmp as (
select "ServingSector" as "SCELL", "InterferingSector" as "NCELL", ("LteScRSRP"-"LteNcRSRP") as "C2I"
from tbMROData
where ("ServingSector", "InterferingSector") in (select "ServingSector", "InterferingSector"
                                                 from tbMROData
                                                 group by "ServingSector", "InterferingSector"
                                                 having count(*)>=$1) )
select "SCELL", "NCELL", avg("C2I") as "C2I_Mean", stddev("C2I") as "std"
from tmp
group by "SCELL", "NCELL";
"""
```

与正态分布相关的概率则是通过调用 scipy 库中的 norm.cdf 函数实现：

```
for i in data:
    t1 = tuple(i)
    t2 = (norm.cdf(9,t1[2],t1[3]), norm.cdf(6,t1[2],t1[3])-norm.cdf(-6,t1[2],t1[3]))
    t3 = t1+t2
    await connection.executemany(command1, [tbC2Inew.from_tuple(t3).to_tuple()])
```

从而实现函数的设计。

触发器设计：

系统中设计了触发器，用来在导入数据的时候检查要插入项的主键是否已存在，如果已存在就删除对应的项，再插入该项。相当于用 delete/insert 方式实现数据更新。下面为 tbCell 的触发器。

```
CREATE OR REPLACE FUNCTION trg_tbcell() RETURNS trigger AS $trg_tbcell$
    BEGIN
        DELETE FROM tbcell
        WHERE "SECTOR_ID"= NEW."SECTOR_ID";
        return NEW;
    END;
$trg_tbcell$ LANGUAGE plpgsql;

CREATE OR REPLACE TRIGGER tbcellBeforeInsertOrUpdate
BEFORE INSERT ON tbcell
FOR EACH ROW
EXECUTE FUNCTION trg_tbcell();
```

3.4 索引设计

现以表 tbMROData 为例，说明索引的设计，同时展示和对比不同索引的效果。

非聚集索引：

添加索引前的查询语句和执行结果如下：

```
select *
from tbmrodata
where "InterferingSector"  = '253931-2';
```

9419 行已获取 - 210ms (5ms 获取)

现通过以下语句在"InterferingSector"字段添加非聚集索引：

```
create index index1
on tbmrodata("InterferingSector");
```

再次执行之前的语句，结果如下：

9419 行已获取 - 24ms (6ms 获取)

可以看到，加了索引后执行速度比之前快了很多。

聚集索引：

postgresql 中不提供聚集索引的机制，但是可以通过 CLUSTER 命令依据索引对表中内容排序，从而 模拟聚集索引的功能。

下面根据刚才添加的 index1 对表中内容排序：

```
cluster tbmrodata using index1;
```

再次执行之前的查询语句，结果如下：

9419 行已获取 - 17ms (6ms 获取)

可以看到，速度相比于非聚集索引有所提升，但并没有很明显的变化。同时，需要注意的一点是， CLUSTER 操作是一次性的，之后如果更新表中内容，则表中内容不会自动重排。

鉴于以上原因，我们决定为系统里查询比较频繁的 4 张表（tbCell,tbKPI,tbMROData 和 tbPRB）设计非聚集索引。

tbCell：将 ENODEB_NAME, ENODEBID, SECTOR_NAME, SECTOR_ID 设置为非聚集索引。

tbKPI：将 SECTOR_NAME 设置为非聚集索引。

tbPRB: 将 ENODEB_NAME 设置为非聚集索引。
tbMROData: 将 ServingSector, InterferingSector 设置为非聚集索引。

3.5 批量导入设计

在数据的导入设计中，用户可以指定要导入的表并设置一次最大操作行 max_line（批量导入数据库的行数，默认为 50）。在导入的时候，首先读入文件，之后执行循环操作，每 max_line 行执行一次插入。在插入之前，会先从这 max_line 行中清洗不符合完整性约束的项，并把行数记录在 datalog.txt 中，之后把剩下的表项批量插入到数据库中。数据库中的触发器会检查待插入项的主键是否已存在，如果存在就删除相应的项，再插入该项。

下面是导入 tbCell 表后 datalog.txt 中的部分内容：

datalog.txt

```
1   tbcell insert error: can not insert line 2002
2   tbcell insert error: can not insert line 2003
3   tbcell insert error: can not insert line 2004
4   tbcell insert error: can not insert line 4584
5   tbcell insert error: can not insert line 5453
6   tbcell insert error: can not insert line 5454
7   tbcell insert error: can not insert line 5455
8   tbcell insert error: can not insert line 5456
9   tbcell insert error: can not insert line 5457
10  tbcell insert error: can not insert line 5458
11  tbcell insert error: can not insert line 5459
12  tbcell insert error: can not insert line 5460
13  tbcell insert error: can not insert line 5461
14  tbcell insert error: can not insert line 5462
15  tbcell insert error: can not insert line 5463
16  tbcell insert error: can not insert line 5464
17  tbcell insert error: can not insert line 5465
18  tbcell insert error: can not insert line 5466
19  tbcell insert error: can not insert line 5467
20  tbcell insert error: can not insert line 5468
21  tbcell insert error: can not insert line 5469
22  tbcell insert error: can not insert line 5470
23  tbcell insert error: can not insert line 5471
```

我们可以打开 tbCell 表，查看未被成功导入的行。

2002	隧	15603	B宝函谷隧	38400	48	0	16	18836	华为	111.97831	33.63761	宏站	0	12	3	6	3
2003	隧	15603	B宝函谷隧	38400	49	1	16	18836	华为	111.97831	33.63761	宏站	130	12	3	6	3
2004	隧	15603	B宝函谷隧	38400	50	2	16	18836	华为	111.97831	33.63761	宏站	290	12	3	6	3

可以看到，第 2002 到 2004 行都是最后 3 列不满足“TOTLETILT= ELECTTILT+ MECHTILT”的约束。

5451	东	7423	E宝焦村东	38400	126	0	42	14355	华为	111.78915	33.51611	宏站	105	28	3	2	5
5452	东	7423	E宝焦村东	38400	127	1	42	14355	华为	111.78915	33.51611	宏站	255	28	3	2	5
5453	F-	11317	F阳上庄F-	38400	373	1	124	14400	华为	112.98	33.6814	宏站	10				
5454	F-	11317	F阳上庄F-	38544	372	0	124	14400	华为	112.98	33.6814	宏站	140				
5455	F-	11317	F阳上庄F-	38544	374	2	124	14400	华为	112.98	33.6814	宏站	250				
5456	F-	11429	G安玉梅F-	38400	468	0	156	18740	华为	113.024	33.72	宏站	340				
5457	F-	11429	G安玉梅F-	38400	470	2	156	18740	华为	113.024	33.72	宏站	130				
5458	F-	11429	G安玉梅F-	38400	469	1	156	18740	华为	113.024	33.72	宏站	260				
5459	营	12162	F阳盐镇营	39148	69	0	23	14369	华为	113	33.6692	室分	0				
5460	F-	236141	G安刘扬F-	38400	455	2	151	14369	华为	113.019	33.7006	宏站	10				
5461	F-	236141	G安刘扬F-	38400	453	0	151	14369	华为	113.019	33.7006	宏站	150				
5462	F-	236141	G安刘扬F-	38400	454	1	151	14369	华为	113.019	33.7006	宏站	230				
5463	厂	246333	G安水泥厂	38400	414	0	138	18740	华为	113.034	33.7364	宏站	30				
5464	厂	246333	G安水泥厂	38400	415	1	138	18740	华为	113.034	33.7364	宏站	105				
5465	厂	246333	G安水泥厂	38400	416	2	138	18740	华为	113.034	33.7364	宏站	240				

可以看到，第 5343 行及之后的行最后 4 列不存在，不满足 TOTLETILT 非空的约束。

下面是 tbCell 表中定义的最后 3 列内容（摘自“《TD-LTE 网络配置数据库》课程实验背景资料及数据建模”）

ELECTTILT	小区天线电下倾角	float	单位：度
MECHTILT	小区天线机械下倾角	float	单位：度
TOTLETILT	总下倾角	float	not null; TOTLETILT= ELECTTILT+ MECHTILT

4. 软件模块设计

4.1 概览

模块	作用
routers/admin.py	提供管理员接口
routers/users.py	提供用户接口
routers/data.py	提供其余业务的接口
settings.py	全局配置文件
model.py	各表的序列化数据类
user.py	用户类

user_token.py,dependency.py	用户登录 token
pylouvain.py	聚类分析
util.py	SQL 查询等工具函数
main.py	入口模块

业务部分重点集中在 admin.py，users.py 和 data.py。因此下面逐一介绍

4.2 用户和管理员部分

```python
class User(BaseModel):
    """能看到的用户信息
    """
    id: int
    username: str
    is_active: Optional[bool] = None
    is_admin: bool = False


You, 3个月前 | 1 author (You)
class UserInDB(User):
    """数据库中的用户信息
    """
    password: str
```

用户对象主要由 id,username,is_active,is_admin,password 组成。

4.2.1 用户注册

用户注册时，需要提供一个账号和一个密码。二者长度均需大于等于 8（可配置）。用户注册后，处于未激活状态，即使登录密码正确也会提示登录失败。需要管理员激活手动激活才能成功登录。

4.2.2 用户登录

用户登录时，提供账号和密码。如果都正确，且已激活，则返回一个 token。后续

访问其他接口时，需携带 token 进行验证。

我们使用 JWT(Json Web Token)对用户进行验证。JWT 是一种非对称加密，服务器上不保存 token 本身。Token 上记录着用户信息和过期时间，通过服务器自身签发的秘钥加密。服务器通过解密可以重新获取这些信息，并判断用户是否有权限。

因为 JWT 采用的是秘钥形式的加密，因此后端不需要保存签发的 Token，节省了存储空间。

4.2.3 管理用户

我们提供了一下接口供管理员管理用户：

HTTP 方法	uri	作用	备注
GET	/api/admin/users	获取全部用户	
POST/DELETE	/api/administrator/{user_id}	授予/移除用户管理员权限	post 授予，delete 移除
POST/DELETE	/api/active/{user_id}	激活/移除用户	同上

管理员获取用户时，只能看到除密码外的信息。

4.2.4 管理数据库

管理员可以通过/admin/postgres/database 接口获取数据库内各表信息。信息包括：表名，表占用物理空间，表所在物理位置，表当前行数。

管理员还可以修改数据库的一些配置，比如 shared_buffers 大小，wal_buffers 大小，effective_cache_size 大小，最大 TCP 连接数量 max_connections，连接超时时长 tcp_keepalives_idle 等等。

需要注意的是，因为 postgres 的特性，修改这些属性不会立即生效（虽然前端显示更改了）。需要等待数据库重启后才会生效。当然，这些属性会被写在一个日志里面，防止因为异常退出导致的修改失败。

4.3 数据上传、下载部分

4.3.1 数据上传

数据上传一共分为以下几个步骤：

1. csv 文件通过 HTTP 送到后端服务器上。服务器缓存文件到临时目录下。
2. 返回 200 成功，同时启动一个后台任务
3. 后台任务每次从临时文件中读取 50 行，通过字段校验检查各个字段是否正确，如果正确，同时插入 50 条。
4. 插入时，会触发一个插入触发器。如果存在相同的记录，就删掉原来的记录。

更新上传记录的结构体。

对于客户端，可以通过访问接口获知当前插入到第几条了。同时，如果完成或者出现错误了，也会在结构体上反映出来。

详细地说，上传主要由两个函数组成：一个负责处理 I/O 和请求，另一个负责解析和上传。

处理请求的函数为 upload_data()。这个函数接收数据库名，文件和文件格式。

```
#将datalog中的内容清零
with open("datalog.txt", 'r+') as f:
    f.truncate(0)

id = uuid.uuid4().hex
new_filepath = os.path.join(Settings.TEMPDIR, id)
try:
    with open(new_filepath, "wb") as buffer:
        shutil.copyfileobj(file.file, buffer)
finally:
    file.file.close()
try:
    f = open(new_filepath, "r", encoding=encoding)
except Exception:
    raise fastapi.HTTPException(status_code=400, detail="文件编码错误")

__upload_dict[id] = UploadTask(id=id)
background_tasks.add_task(upload_data_background,
                          name, id, f, new_filepath, str2Model[name.name], max_line)

return {"id": id, "url": f"{Settings.DATA_ROUTER_PREFIX}/upload/status?id={id}"}
```

日志刷新

保存文件，生成一个id来命名

保存到状态结构体中

启动后台任务

负责解析函数主体如下：

```
counter = 0
connection = await get_connection()
reader = csv.reader(file, delimiter=',', quotechar='"')
next(reader)  # 跳过标题
try:
    command = model.get_insert_command()
    async with connection.transaction():
        for fifty_rows in batch(iter(reader), max_line):
            list = []
            # 数据清洗：从要插入的记录中删除不满足约束条件的项
            for i in fifty_rows:
                counter += 1
                r = model.from_tuple(i)
                if not r.constraints():
                    logger = Logger('datalog.txt')
                    logger.write(name+" insert error: "+"can not insert line "+str(counter)+"\n")
                else:
                    list.append(r.to_tuple())

            await connection.executemany(command, list)
            __upload_dict[id].current_row = counter
except Exception as e:
    __upload_dict[id].msg = str(e)
    __upload_dict[id].failed = True
finally:
    __upload_dict[id].done = True
    file.close()
    os.remove(new_filepath)
if isinstance(model,tbPRB):
    await connection.execute("CALL update_tbprb_new();")
```

读取CSV文件

获取插入数据库的模板命令

每次获取max_line(50)行数据

把数据转为model，并检查它的约束

如果不满足约束，写入到日志里面

一次性插入50行

更新状态字典

其中，model 是我们给每个表定义的一个 Python 类，主要是借助 pydantic 进行类型校验，用于数据库类型转 Python 类型和 Python 类型转数据库类型。以 tbcell 为例，它的模型如下：

```
class tbCell(pydantic.BaseModel,getInsertCmdMixin):
    CITY:str
    SECTOR_ID:str
    SECTOR_NAME:str
    ENODEBID:int
    ENODEB_NAME:str
    EARFCN:int      You, 2个月前 • ORM like
    PCI:int
    PSS:Optional[int]
    SSS:Optional[int]
    TAC:int
    VENDOR:str
    LONGITUDE:float
    LATITUDE:float
    STYLE:str
    AZIMUTH:float
    HEIGHT:Optional[float]
    ELECTTILT:Optional[float]
    MECHTILT:Optional[float]
    TOTLETILT:Optional[float]
```

上传失败生成的日志是在运行目录下面的 datalog.txt，例如：

datalog.txt

```
tbcell insert error: can not insert line 2002
tbcell insert error: can not insert line 2003
tbcell insert error: can not insert line 2004
tbcell insert error: can not insert line 4584
tbcell insert error: can not insert line 5453
tbcell insert error: can not insert line 5454
tbcell insert error: can not insert line 5455
tbcell insert error: can not insert line 5456
tbcell insert error: can not insert line 5457
tbcell insert error: can not insert line 5458
tbcell insert error: can not insert line 5459
tbcell insert error: can not insert line 5460
tbcell insert error: can not insert line 5461
tbcell insert error: can not insert line 5462
tbcell insert error: can not insert line 5463
```

而插入状态是一个用于前端检查当前插入到多少行的结构体，这是一个全局的变量。
其内部的数据格式如下：

```python
class UploadTask(BaseModel):
    id: str
    done: bool = False
    failed: bool = False
    msg: str = ""
    current_row: int = 0


__upload_dict = dict()
```

数据格式

id 是上传任务的编号，done 表示任务是否已经完成，failed 表示任务是否失败，msg 表示失败的原因，current_row 表示当前完成的行数。前端可以利用这个机制绘制状态条。

4.3.2 数据下载

1. 下载的流程和上传类似。
2. 首先，客户端需要提供下载表的表名，然后通过 COUNT(*)获取表内到底有多少条记录。如果记录数量超过限制，会分批处理。
3. 对于每一批，使用 postgres 提供的 COPY 指令，可以快速把数据导出为 csv 临时文件。
4. 最后返回一个列表，包含一个或若干个链接，等待客户端下载。

客户端取到链接后，可以用 get 访问链接来下载文件。
同样，下载也分为两个函数，一个函数负责处理请求和生成相应的 csv 文件，另一个函数负责提供下载的接口，防止一次性传输过多文件，造成服务器或者客户端瘫痪。
具体的实现如下：

下面分别是处理请求和生成 csv 的函数和下载的函数

```python
@data_router.get("/download")
async def download_table(table: ValidTableName):
    """请求准备下载
    """
    table_name = table.name
    connection = await get_connection()
    if table_name in __download_dict:
        for file in __download_dict[table_name]:
            os.remove(f"{Settings.TEMPDIR}/{file}")
    __download_dict[table_name] = []
    async with connection.transaction():
        count = await connection.fetchrow(f'SELECT COUNT(*) FROM {table_name}')
        count = count["count"]
        max_row = Settings.MAX_ROW_PER_FILE
        for i in range(0, count, max_row):
            id = uuid.uuid4().hex + ".csv"
            command = f'COPY (select * from {table_name} LIMIT {max_row} OFFSET {i}) TO \'{Settings.TEMPDIR}/{id}\' WITH (FORMAT CSV, HEADER);'
            # print(command)
            await connection.execute(command)
            __download_dict[table_name].append(id)
    return ["/data/download/file?id={}".format(file) for file in __download_dict[table_name]]
```

删除之前的缓存（不删除也可以，删除节省空间）

判断行数，如果行数超过max_row，就分段进行导出

使用postgres把数据导出为csv格式

更新状态表

```python
@data_router.get("/download/file")
async def download_table_file(id: str):
    file = os.path.join(Settings.TEMPDIR, id)
    return FileResponse(path=file, filename=id)
```

其中__download_dict 和上文的上传状态一样，维护了一个下载的信息。下载表内保存的是表名-下载链接的一个组合。

4.4 业务查询

4.4.1 小区配置信息查询

传入 SECTOR_NAME 或者 SECTOR_ID，查询相关的 tbcell 小区信息：

```
SELECT * From tbCell WHERE "SECTOR_NAME" = $1
- or
SELECT * From tbCell WHERE "SECTOR_ID" = $1;
```

这里有一个辅助函数，用于获取全部的 tbcell，这样客户端就能进行选取，而不是输入一个 id：

```python
@data_router.get("/sector", response_model=List[Dict[str, str]])
async def get_sector_detail(choice: GetSectorEnocdeChoice):
    """
    获取全部小区id或者名称
    """
    if choice == GetSectorEnocdeChoice.name:
        command = 'SELECT "SECTOR_NAME" From tbCell;'
    elif choice == GetSectorEnocdeChoice.id:
        command = 'SELECT "SECTOR_ID" From tbCell;'
    return await fetch_all(command)


@data_router.get("/sector/detail", response_model=tbCell)
async def get_sector_detail(name_or_id: str, choice: GetSectorEnocdeChoice):
    """
    输入(或下拉列表)小区id或名称，返回sector全部信息
    """
    if choice == GetSectorEnocdeChoice.name:
        command = 'SELECT * From tbCell WHERE "SECTOR_NAME" = $1;'
    elif choice == GetSectorEnocdeChoice.id:
        command = 'SELECT * From tbCell WHERE "SECTOR_ID" = $1;'
    return await fetch_one_then_wrap_model(command, tbCell, name_or_id)
```

4.4.2 基站 eNodeB 信息查询

传入 ENODEB_NAME 或者 ENODEBID，查询相关的 tbcell 小区信息。

```
SELECT * From tbCell WHERE "ENODEB_NAME" = $1;
- or
SELECT * From tbCell WHERE "ENODEBID" = $1;
```

4.4.3 小区 KPI 指标信息查询

传入小区名，开始时间和结束时间，查询 KPI 指标

```
SELECT "StartTime","RCCConnSUCC" as "Data" From tbKPI WHERE "SECTOR_NAME" =
'{$name}' AND "StartTime" BETWEEN $1 AND $2;
```

返回值是一个列表，列表里是一个对象，包含开始时间，成功率。

其代码也比较简单，这里就不再赘述。

```python
class KPIChoice(str, Enum):
    RCCConnSUCC = "RCCConnSUCC"
    RCCConnATT = "RCCConnATT"
    RCCConnRATE = "RCCConnRATE"
```

约束网页请求选项

```python
You, 3周前 | 1 author (You)
class kpi_detail(pydantic.BaseModel):
    StartTime: datetime.date
    Data: Union[int, float, str]
```

约束输出格式

```python
@data_router.get("/kpi/detail")
async def get_kpi_detail(name: str, choice: KPIChoice, start_time: datetime.date, end_time: datetime.date):
    # command = """
    # SELECT "StartTime","{}" as "Data" From tbKPI WHERE "ENODEB_NAME" = (select "ENODEB_NAME" from tbKPI where "SECTOR_NAME"='{}' limi
    #     choice.value,name)
    command = """
    SELECT "StartTime","{}" as "Data" From tbKPI WHERE "SECTOR_NAME" = '{}' AND "StartTime" BETWEEN $1 AND $2;""".format(
        choice.value,name)
    return await fetch_all(command,start_time, end_time)
```

查询KPI信息

读取数据库，返回

4.4.4 PRB 信息统计与查询

这一部分我们采用存储过程的方法进行生成。在每次更新 PRB 表的时候，都会执行下面的函数：

```
CREATE OR REPLACE PROCEDURE update_tbprb_new()
LANGUAGE SQL
AS $$
    DROP VIEW if exists tbPRBnew;
    CREATE VIEW tbPRBnew AS
        SELECT (timestamp '2000-1-1' + interval '1 year' * (year-2000) + interval
'1 month' * (month-1) + interval '1 day' * (day-1)+interval '1 hour' * hour) as
"StartTime","ENODEB_NAME","AvgNoise0","AvgNoise1","AvgNoise2",..... (忽略中间部
分),"AvgNoise99"
        FROM
        (
        SELECT
        extract(year from "StartTime") as year,
        extract(month from "StartTime") as month,
        extract(day from "StartTime") as day,
        extract(hour from "StartTime") as hour,
        "ENODEB_NAME",
         avg("AvgNoise0") as "AvgNoise0",.......(忽略中间部分),avg("AvgNoise99")
as "AvgNoise99"
        from tbPRB
        GROUP BY year, month, day, hour,"ENODEB_NAME") as TMP
$$;
```

这是一个存储过程，**在每次上传 PRB 数据的时候进行调用**，生成一个叫 tbprbnew 的只读视图。查询的基本流程如下：

1. 从 tbPRB 中按年，月，日，时和 ENODEB_NAME 进行分组，选择年，月，日，时，

ENODEB_NAME 和平均噪音作为数据。假设这个临时查询为 TMP

2. 查找 TMP，将年，月，日，时重新拼成时间字符串，同时保留 ENODEB_NAME 和平均噪音，这些值作为视图。

这样就得到了按小时分组的数据。每次查询的时候，只需要简单地查询这个视图即可。

```
SELECT "StartTime","AvgNoise{prbindex}" as "AvgNoise"
FROM  tbPRBnew
WHERE "ENODEB_NAME" = '{name}' AND "StartTime" BETWEEN $1 AND $2
- $1 是开始时间 ,$2 是结束时间
```

同时，我们还做了 15 分钟级别的查询。这里不再赘述。

```
@data_router.get("/prb/detail")
async def get_avg_prb_line_chart(name: str, granularity: GranularityChoice, prbindex: int, start_time: datetime.datetime, end_time: da
    """
    输入网元，选择第i个PRB，选择时间区间和粒度，返回干扰噪声平均值折线图
    granularity : 粒度
    prbindex: 第几个prb
    name: 小区名称
    """
    if granularity == GranularityChoice.a15min:
        command = f"""
        SELECT "StartTime","AvgNoise{prbindex}" as "AvgNoise"
        FROM  tbPRB
        WHERE "ENODEB_NAME" = '{name}' AND "StartTime" BETWEEN $1 AND $2
        """
        return await fetch_all(command, start_time, end_time)

    else:
        connection = await get_connection()
        command = f"""
        SELECT "StartTime","AvgNoise{prbindex}" as "AvgNoise"
        FROM  tbPRBnew
        WHERE "ENODEB_NAME" = '{name}' AND "StartTime" BETWEEN $1 AND $2
        """
        result = await fetch_all(command,start_time, end_time, connection=connection)
        await connection.close()
        return result
```

查询15分钟级别

查询1小时级别

4.4.5 主邻小区 C2I 干扰分析

generate_tbC2Inew 负责生成新表 tbC2Inew。首先用 sql 语句从 tbMROData 中获取 tbC2Inew 的前 4 个属性（主小区 ID、邻小区 ID、主邻小区 RSRP 差值的均值、主邻小区 RSRP 差值的标准差），之后根据前面的属性用 python 语句计算出后两个属性（主邻小区 RSRP 差值小于 9 的概率、主邻小区 RSRP 差值绝对值小于 6 的概率），与前 4 个属性共同构成完整的记录，插入数据库。

4.4.6 重叠覆盖干扰小区三元组分析

generate_tbC2I3：负责根据表 tbC2Inew，找出所有的小区三元组<a, b, c>，其中 a、b、c 互为邻小区，并生成新表 tbC2I3。首先筛选出所有符合用户定义范围内的数据，对筛选后 的进行查询获得所有小区名，遍历小区名，先得到 a 小区与 b 小区。再在小区名中选择与 a， b 不重名的小区。在表中查询 c 是否和 a、b 在表中存在数据。如果存在则判断是否与之前查询出的结果有重复。如果没有则加入结果中。

4.4.7 MRO/MRE 测量数据解析

这个部分可以分为以下几个步骤：

1. 接收文件，保存为临时文件。
2. 解压压缩包，多线程遍历每个文件
3. 对于每个文件，解析 XML 数据。这里的数据需要联查 Tbcell，为了加快速度，我们的做法是先将 tbcell 中的 PCI 映射到 ID 的字典生成出来， 然后使用时直接哈希表映射。
4. 插入的时候，以文件为单位插入，提高插入的效率。

为了方便体现结果，我们将这部分数据保存在一个新的数据表(mrodataexternal)里。客户端可以直接下载；直接插入到 mrodata 中也是可以的。

具体到实现上面，我们使用的是协程。协程是一种编程语言层面无栈线程，其线程切换速度要快于传统的线程（因为是用户态的），非常适合 I/O 密集的场景。这这个例子中，I/O 处理（插入数据库）的时间要多于判断约束的时间，因此比较适合协程。

对于上面的操作，我们同样写了两个函数。
首先是 mro 函数。mro 函数主要负责处理请求和下载文件、解压数据。

```
@data_router.post("/mro_parse")
async def mro(file: UploadFile,encoding:str="utf-8"):
    zipfilename = uuid.uuid4().hex
    filename = "".join(file.filename.split(".")[:-1])
    zipfilepath = os.path.join(Settings.TEMPDIR,zipfilename)
    try:
        with open(zipfilepath, "wb") as buffer:
            shutil.copyfileobj(file.file, buffer)
    finally:
        file.file.close()
    folder_path = os.path.join(Settings.TEMPDIR,uuid.uuid4().hex)
    os.mkdir(folder_path)
    try:
        with support_gbk(ZipFile(zipfilepath)) as zfp:
            zfp.extractall(folder_path)
    finally:
        os.remove(zipfilepath)
```

下载和解压zip文件到临时目录

```
    connection = await get_connection()
    pci2id = {i["PCI"]:i["SECTOR_ID"] for i in await fetch_all('SELECT "SECTOR_ID","PCI" From tbCell;',connection=connection)}
    connection.close()
    try:
        unzip_path = os.path.join(folder_path, filename)
        files = []
        for r, d, f in os.walk(unzip_path):
            for file in f:
                if file.endswith(".xml") and file.startswith("TD-LTE_MRO_"):
                    files.append(os.path.join(r, file))
        await asyncio.gather(*[async_parse_mro(f,pci2id,encoding) for f in files])
    except Exception as e:
        return f"failed when unzipping {str(e)}"
    finally:
        shutil.rmtree(folder_path)
    return "ok"
```

获取全部的PCI-SECTOR id信息，用于转换

遍历临时文件目录的每一个文件，判断后缀

调用后台函数async_parse_mro处理每一个文件

删除临时目录

MRO 的解析部分比较复杂，主要需要摘取的数据如下：

```
<measurement>
  <smr>MR.LteScRSRP MR.LteNcRSRP MR.LteScRSRQ MR.LteNcRSRQ MR.LteScEarfcn MR.LteScPci
  MR.LteNcEarfcn MR.LteNcPci MR.LteScTadv MR.LteScPHR MR.LteScRIP MR.LteScAOA
  MR.LteScPlrULQci1 MR.LteScPlrULQci2 MR.LteScPlrULQci3 MR.LteScPlrULQci4
  MR.LteScPlrULQci5 MR.LteScPlrULQci6 MR.LteScPlrULQci7 MR.LteScPlrULQci8
  MR.LteScPlrULQci9 MR.LteScPlrDLQci1 MR.LteScPlrDLQci2 MR.LteScPlrDLQci3
  MR.LteScPlrDLQci4 MR.LteScPlrDLQci5 MR.LteScPlrDLQci6 MR.LteScPlrDLQci7
  MR.LteScPlrDLQci8 MR.LteScPlrDLQci9 MR.LteScSinrUL MR.LteScRI1 MR.LteScRI2
  MR.LteScRI4 MR.LteScRI8 MR.LteScPUSCHPRBNum MR.LteScPDSCHPRBNum MR.LteScBSR
  MR.LteSceNBRxTxTimeDiff MR.GsmNcellBcch MR.GsmNcellCarrierRSSI MR.GsmNcellNcc
  MR.GsmNcellBcc MR.TdsPccpchRSCP MR.TdsNcellUarfcn MR.TdsCellParameterId</smr>
  <object MmeCode="52" MmeGroupId="1034" MmeUeS1apId="230738078" TimeStamp="
  2016-06-25T13:00:01.920" id="5682-128:38400:2">
    <v>65 NIL 25 NIL 38400 85 NIL NIL NIL NIL 73 NIL NIL NIL NIL NIL NIL NIL NIL NIL
    NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL
    NIL NIL NIL NIL NIL</v>
  </object>
  <object MmeCode="50" MmeGroupId="1034" MmeUeS1apId="226555669" TimeStamp="
  2016-06-25T13:00:01.920" id="5682-128:38400:2">
    <v>74 NIL 25 NIL 38400 85 NIL NIL NIL NIL 73 NIL NIL NIL NIL NIL NIL NIL NIL NIL
    NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL
    NIL NIL NIL NIL NIL</v>
  </object>
```

其中 PCI 可以通过 tbcell 的数据转成 sector_id。

解析的函数如下：

```python
async def async_parse_mro(f:str,pci2id,encoding="utf-8"):
    # result = []
    connection = await get_connection()
    command = tbMRODataExternal.get_insert_command()
    with open(f, "r",encoding=encoding) as file:
        tree = ET.parse(file)
        root = tree.getroot()
        assert len(root) == 2, "文件不符合要求"
        # header = root[0]
        smr = root[1][0][0]
        FIELD = ("MR.LteScRSRP", "MR.LteNcRSRP", "MR.LteScEarfcn", "MR.LteNcPci")
        smr = smr.text.split()
        index = [smr.index(i) for i in FIELD]   # 查找上面字段对应的位置

        result = []
        for data in root[1][0][1:]:
            timeStamp = data.attrib["TimeStamp"]
            for object in data:
                object = object.text.split()
                try:
                    scid = pci2id[int(object[5])]   # 转PCI为sector_id
                    ncid = pci2id[int(object[7])]
                except ValueError:
                    continue                        # 校验和检查数据
                d =tbMRODataExternal.from_tuple((timeStamp, scid,ncid, object[index[0]], object[index[1]], object[index[2]], object[ir
                if d is not None:                   # 为None表示这个数据无效
                    result.append(d.to_tuple())
        try:
            await connection.executemany(command,result)
        except Exception as e:
            print(e)
    print("Done")
```

4.4.8 网络干扰分析

这部分我们基本上是按照指导书上的做的。

1. 首先获取 tbC2I 的全部数据
2. 处理得到节点和边，送入开源模块进行处理
3. 对于每个节点的经纬度坐标，我们通过查询 tbCell 得到
4. 最后使用 networkx 进行绘图，得到一张图片，返回给客户端

我们主要调整了边的阈值（原为 40），以及图片的大小，用于加速处理的过程。

4.5 其他辅助类

4.5.1 序列化模型

我们使用 pydantic 库来简化序列化检验部分。对于每一个需要序列化的表，都有一个对应的类。例如，对于 tbKPI 表：

```
CREATE TABLE IF NOT EXISTS tbKPI (
"StartTime" date,
"ENODEB_NAME" varchar(255),
"SECTOR_DESCRIPTION" varchar(255) NOT NULL,
"SECTOR_NAME" varchar(255),
"RCCConnSUCC" INT,
"RCCConnATT" INT,
"RCCConnRATE" FLOAT,
PRIMARY KEY("StartTime","SECTOR_NAME")
);
```

对应的类是：

```
class tbKPI(pydantic.BaseModel,getInsertCmdMixin):
    StartTime:datetime.date
    ENODEB_NAME:str
    SECTOR_DESCRIPTION:str
    SECTOR_NAME:str
    RCCConnSUCC:int
    RCCConnATT:int
    RCCConnRATE:float
```

我们手动实现了类似 ORM 的效果。以上传为例，我们从 csv 中读取到的数据是一个元组，类似于：(value1,value2,value3....)。我们对这个数据进行检验（使用上面的类），然后插入数据库。下载时也是类似。SQL 查询得到的结果是一个字典(key:value 键对)。我们用字典初始化这个类，然后序列化为 JSON 对象，最后返回给前端。

4.5.2 日志

Fastapi 自身提供了控制台的日志。我们又独立使用 python Logger 库重写了一个针对上传的日志。这个日志写入数据到 datalog.txt。当上传出现错误时，会在日志中写入“insert error”错误。

5. 数据库物理文件设计

通过查阅资料，postgresql 的数据和配置文件默认存放在集簇的数据目录里， 通常用 PGDATA 来引用。数据库一般有 3 个配置文件（postgresql.conf、pg_hba.conf 和 pg_ident.conf）。可以通过设置以下参数来改变各文件的存放位置。

data_directory (string)

指定用于数据存储的目录。这个选项只能在服务器启动时设置。

config_file (string)

指定主服务器配置文件（通常叫postgresql.conf）。这个参数只能在postgres命令行上设置。

hba_file (string)

指定基于主机认证配置文件（通常叫pg_hba.conf）。这个参数只能在服务器启动的时候设置。

ident_file (string)

指定用于用户名称映射的配置文件（通常叫pg_ident.conf）。这个参数只能在服务器启动的时候设置。另见第 20.2 节。

原来的 data_direcotry
我们将 data_direcotry 移动到了 D:/pgdata。

```
tb=# show data_directory;
 D:/pgdata
```

同时，对于索引文件，我们使用 postgres 的 TABLESPACE，将索引文件放到了其他盘上。
我们创建了新的 tablespace: E:/pgdata:

```
tb=# \db
 mytbsp     | postgres | E:\pgdata
 pg_default | postgres |
 pg_global  | postgres |
```

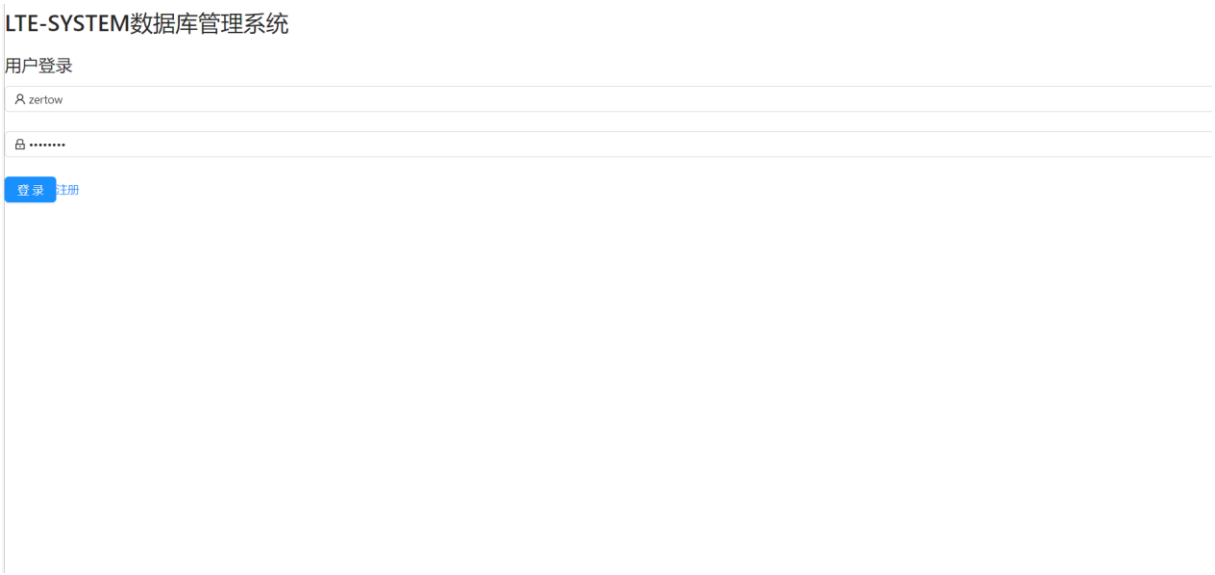
然后创建索引的时候，设置索引保存的 tablespace 为 mytbsp，即可将索引与数据分盘保存。

五. 系统运行示例

1. 登录界面

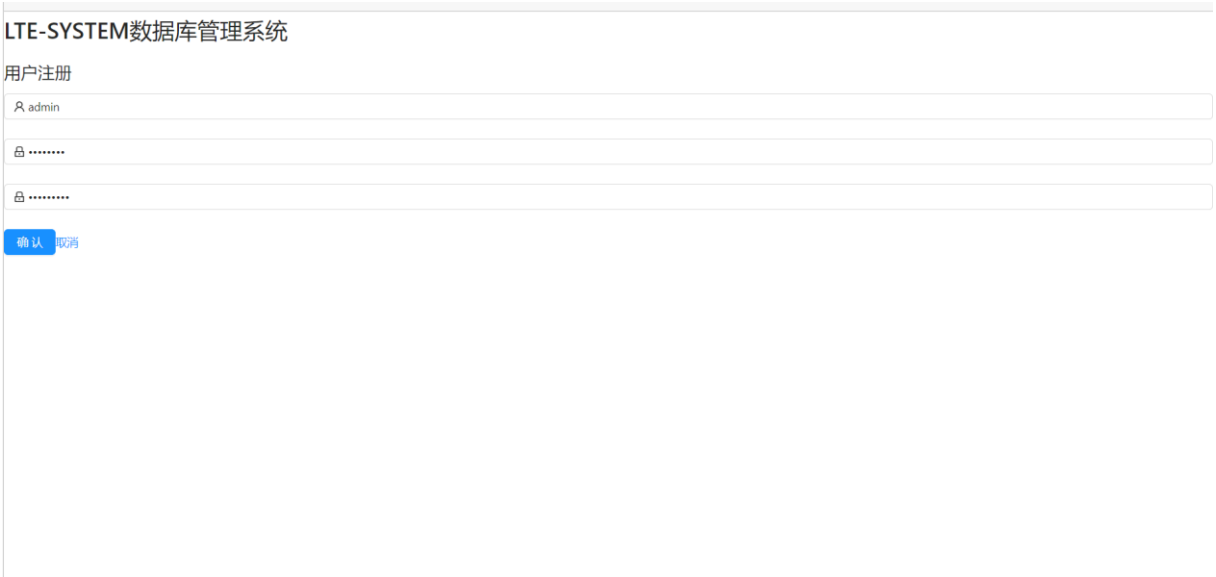
a. 用户登录

输入用户账号以及密码，前端将查询对应 api 判断账号正确性，并且登录到主界面登陆的同时会将用户的 token 存入本地，以便下次免登录直接进入系统



b. 用户注册

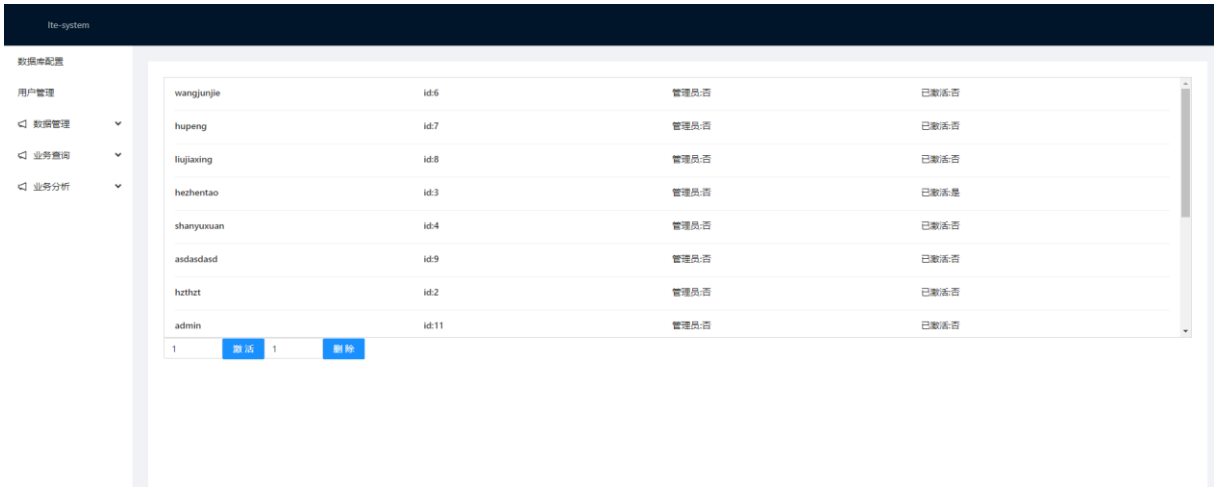
注册用户，并且确认密码，如果注册成功，会有 alert 提示注册成功。但是由于账号未激活，需要管理员激活账号后才能成功登录。



2. 用户管理

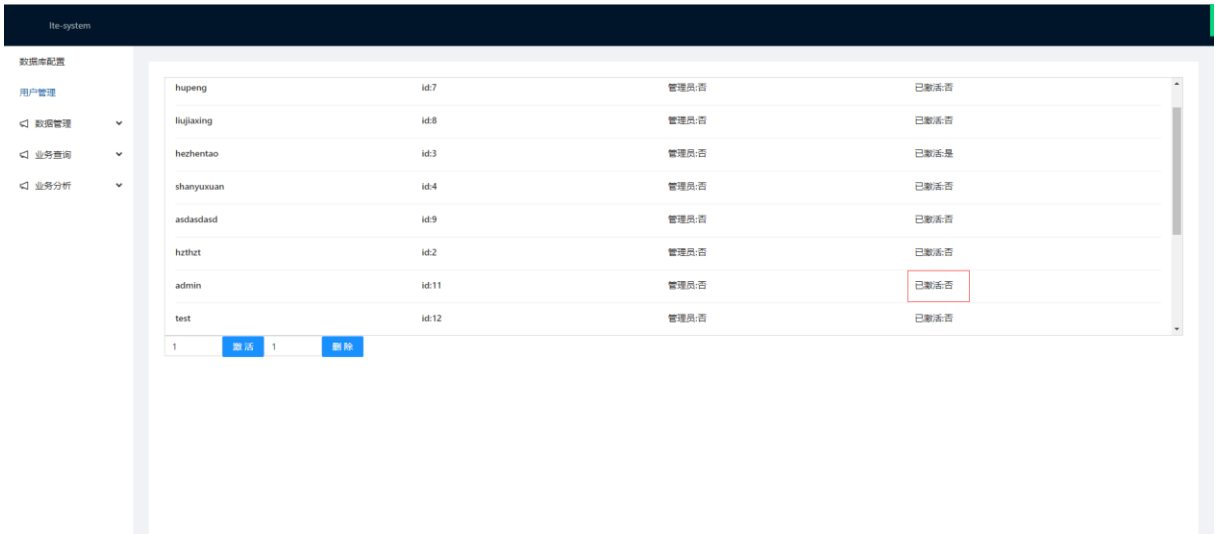
a. 管理页面

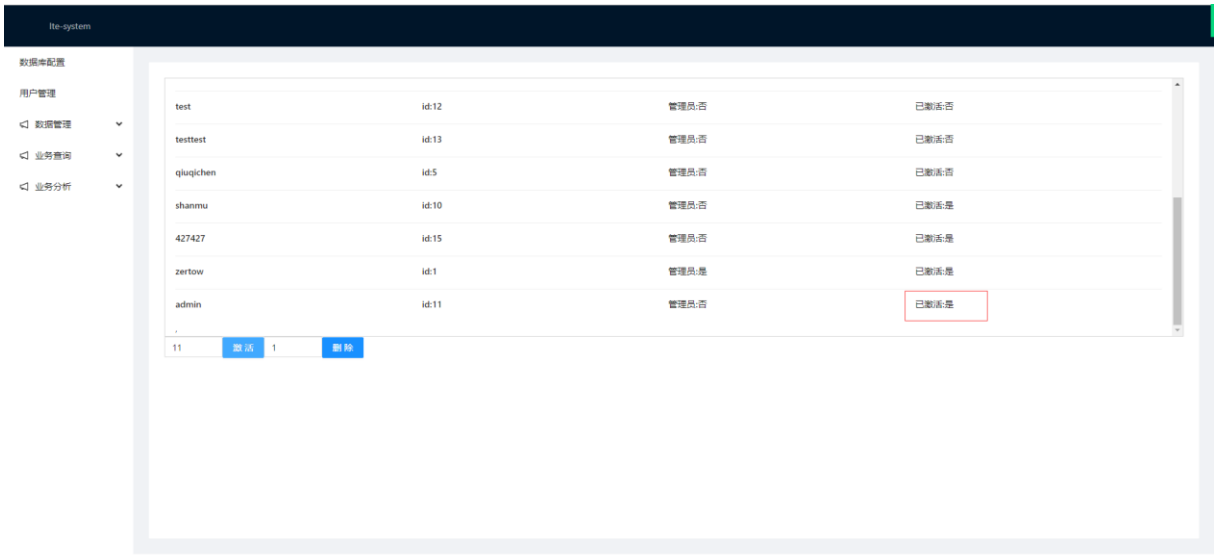
这是管理员独有的页面，普通用户无法访问。



b. 用户激活

在下方输入框输入对应所需激活 id，点击激活按钮就会将对应 id 激活，可以正常登录

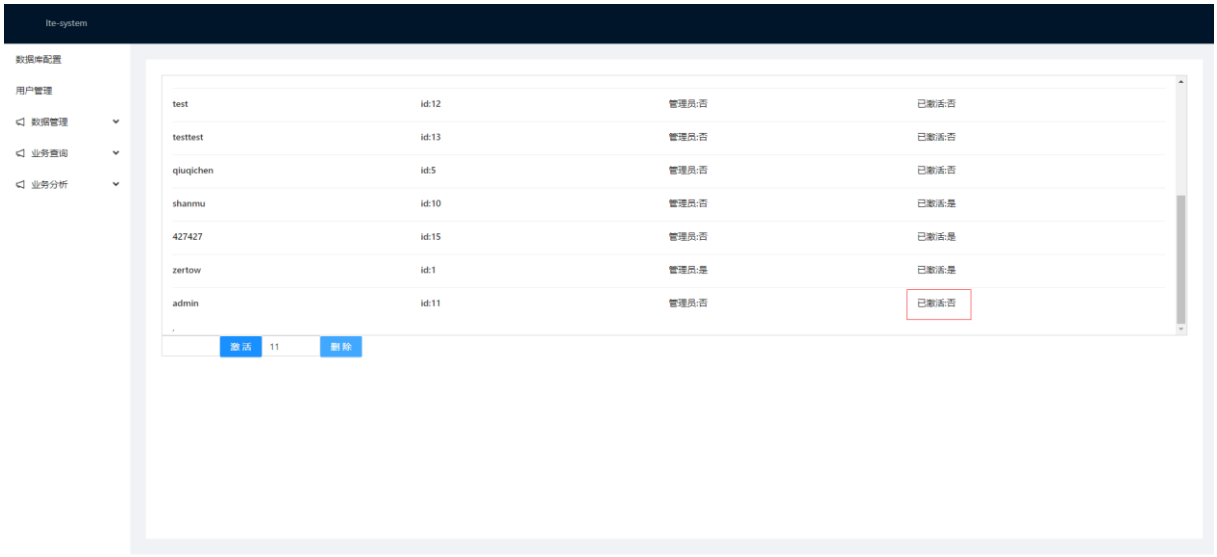




可以发现 11 号已经激活

用户反激活

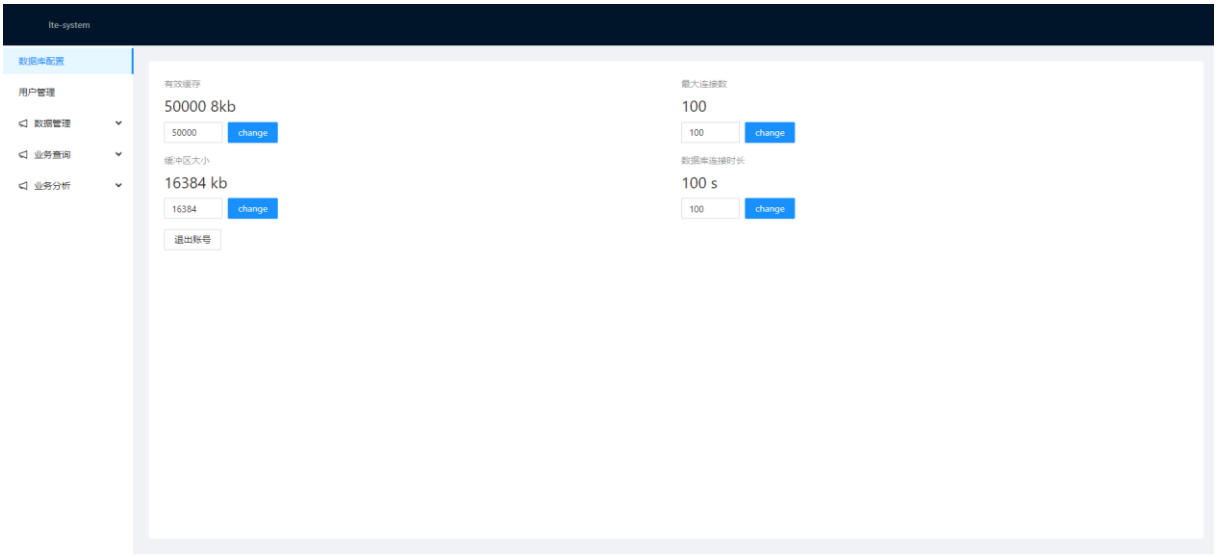
在下方输入框输入对应所需反激活 id，点击激活按钮就会将对应 id 注销，目标用户无法登录



3. 系统管理

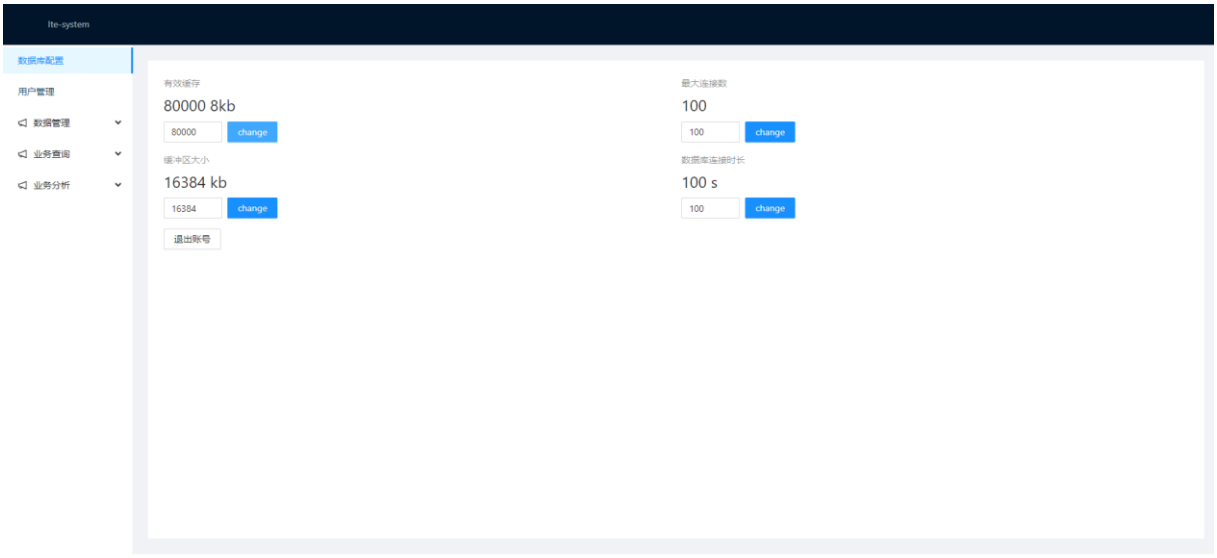
数据库配置

这里能修改数据库配置，但是每次修改后不会立即生效，需要在后端服务器重启后，该配置才会生效。



数据库修改

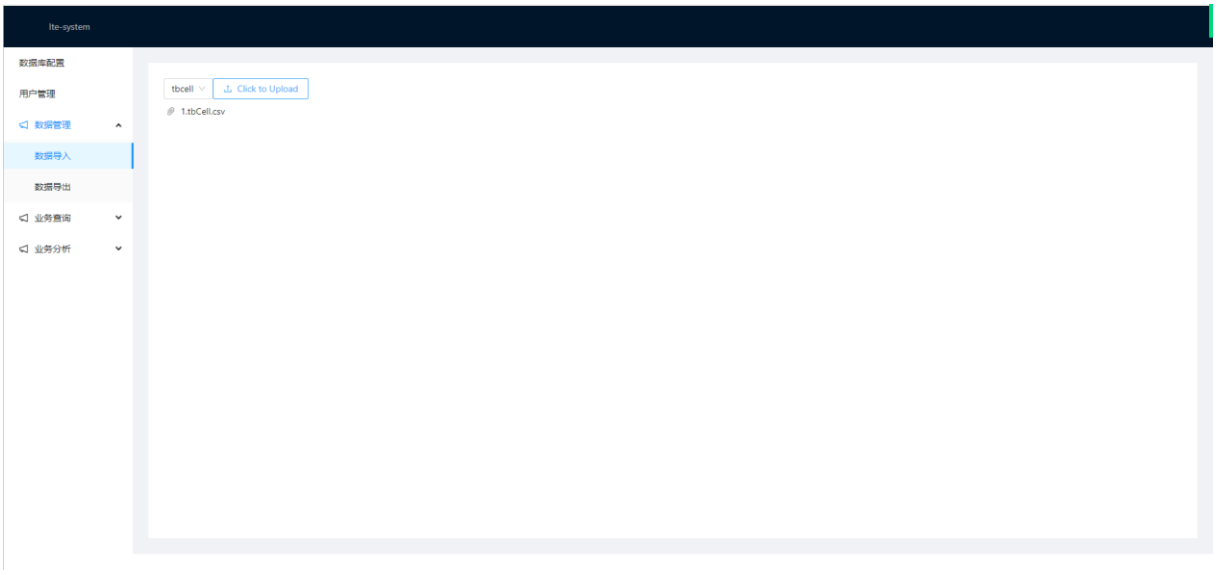
将有效缓存修改为 80000/8kb



4. 数据管理

导入数据表

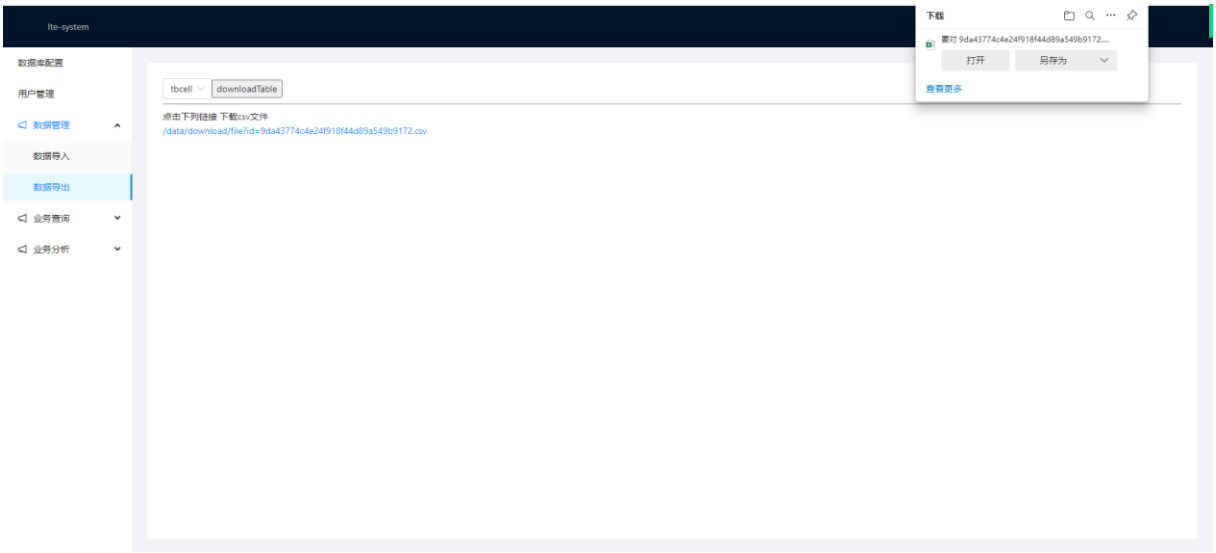
可以在左侧选择栏中选择所需导入数据库，然后在右侧可以选择本地 excel 文件以上传到后端服务器中。



可以看到 tbcell 导入成功

导出数据表

可以在左侧选择栏中选择所需导出数据库，点击右侧按钮后，会在下方生成下载文件地址，点击即可下载。

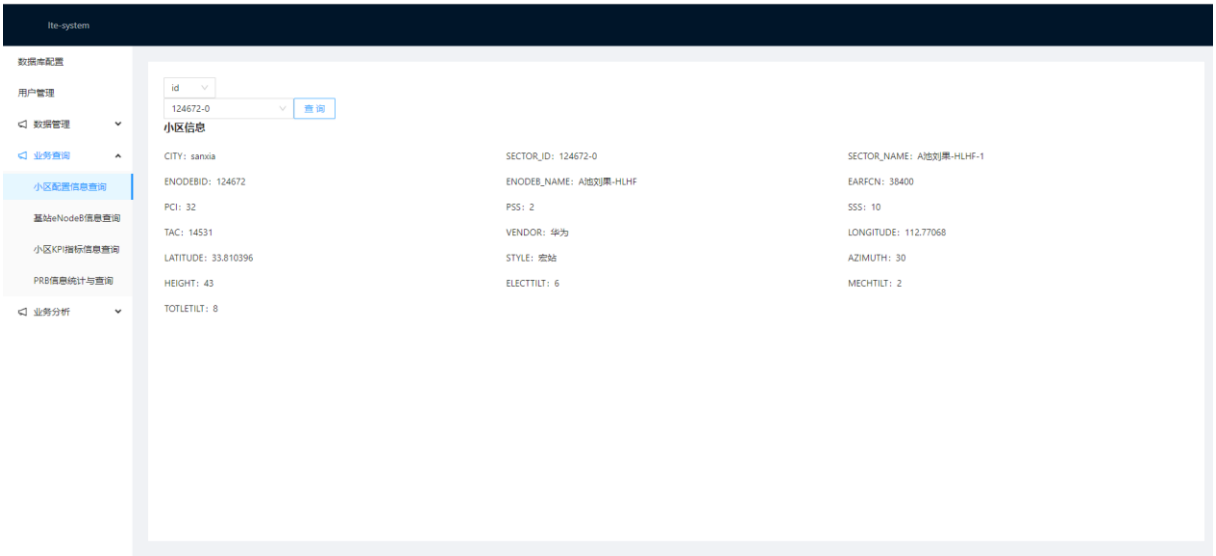


可以看到数据表 tbcell 可以下载

5. 业务查询

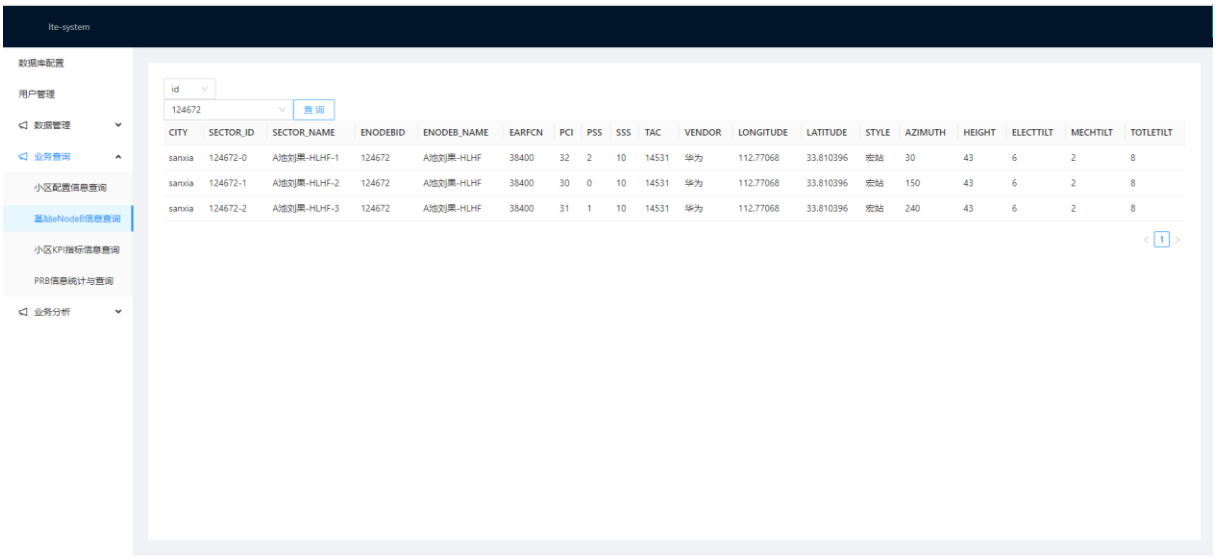
a. 小区配置信息查询

顶部选择栏选择查询方式，id 或者 name，下侧为所需查询的数据，这些数据是可查询，并且会自动提示可能对的查询数据。查询后会在下方将对应的数据详细展示。



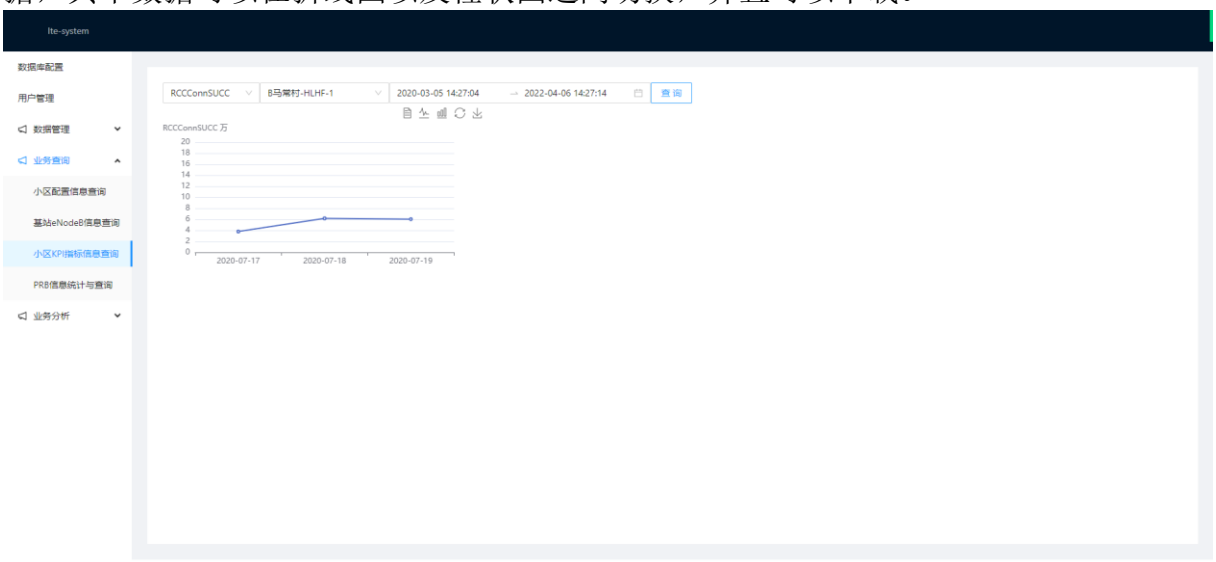
b. 基站 eNodeB 信息查询

eNodeB 的与小区信息类似，但是有多个数据，因此选择使用 list 来展示数据。



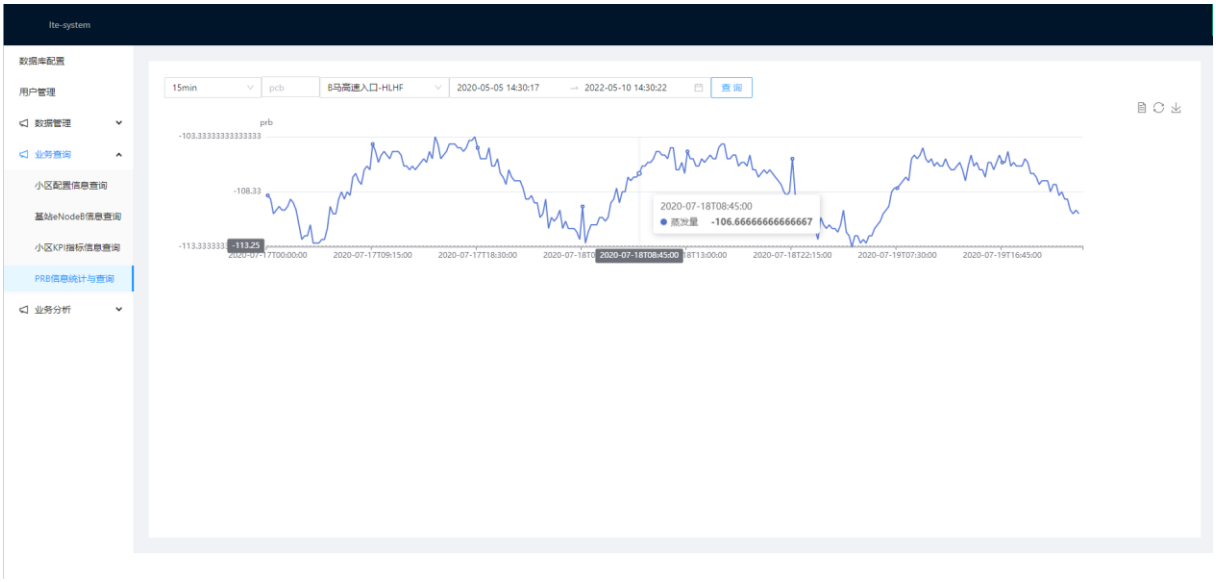
c. 小区 KPI 指标信息查询

这里的数据使用 echart 来展示，并且根据参数的上下限选择了适合的展示方式。通过选择 RUCC 类型，小区名以及时间范围，最后通过 echart 来展示数据，其中数据可以在折线图以及柱状图之间切换，并且可以下载。



d. PRB 信息统计与查询

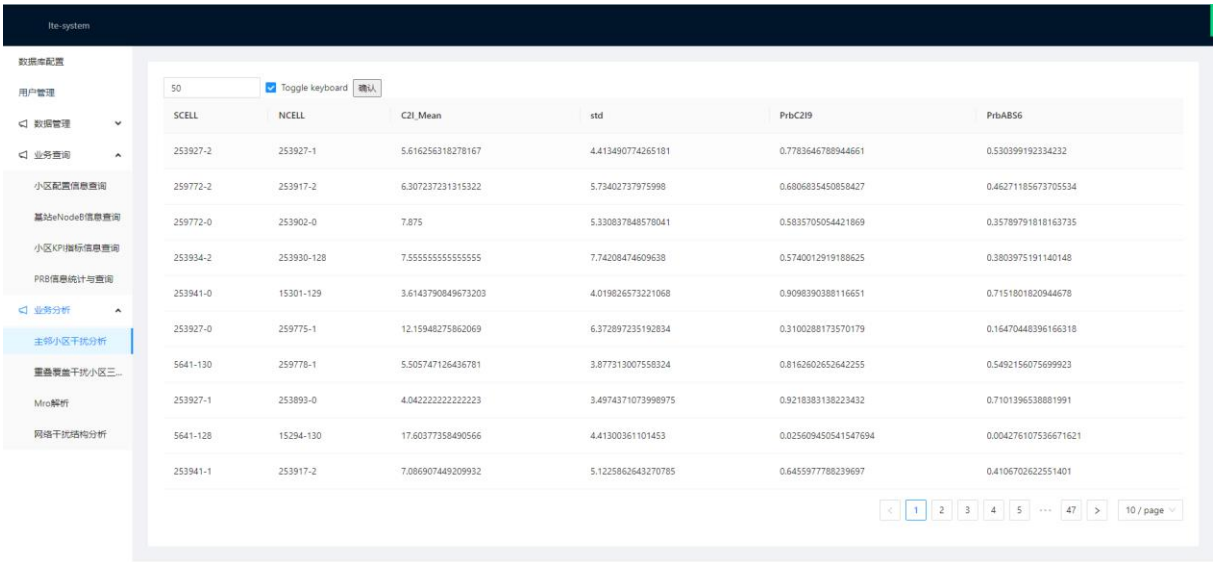
这里的数据使用 echart 来展示，根据数据进行了自适应。通过选择细粒度，小区名以及时间范围，最后通过 echart 来展示数据，其中数据可以下载。



6. 业务处理

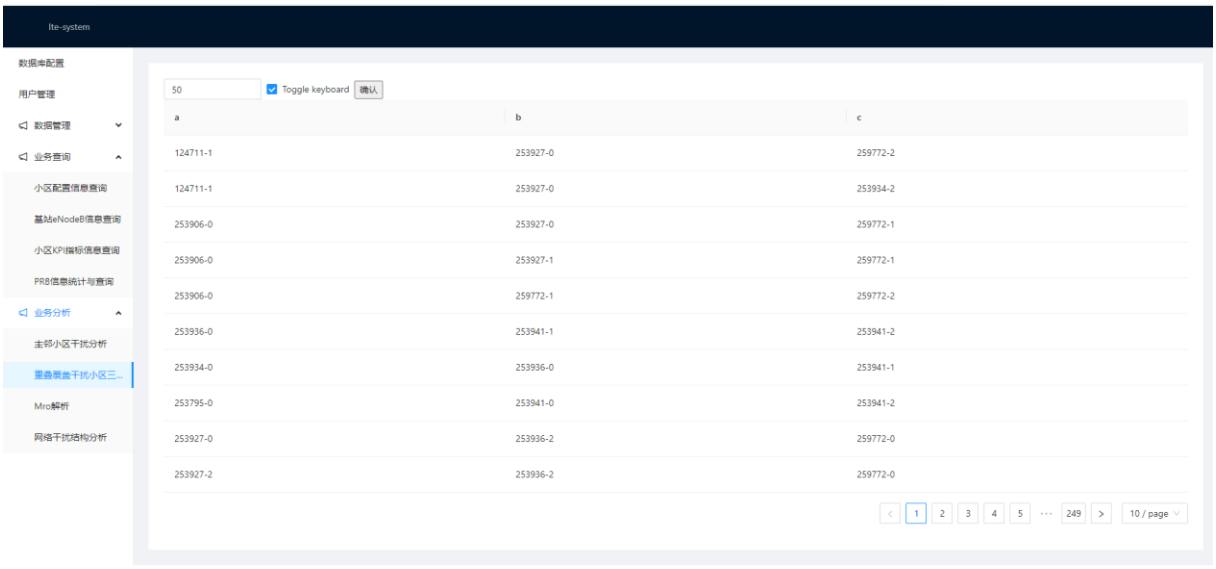
a. 主邻小区 C2I 干扰分析

C2I 干扰分析需要输入重叠主邻小区对数的阈值，用于后端接口的请求，返回值为生成的 c2inew 表的链接，需要再次访问文件流得到 csv 文件，再使用 parseCsv 的方法，将 csv 文件解析为可用的格式



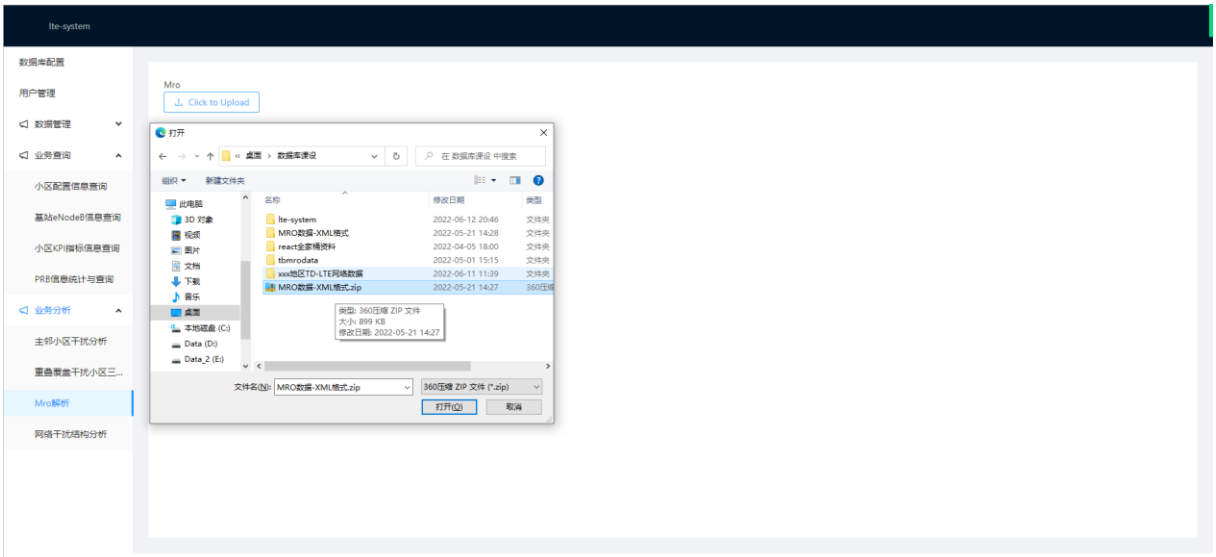
b. 重叠覆盖干扰小区三元组分析

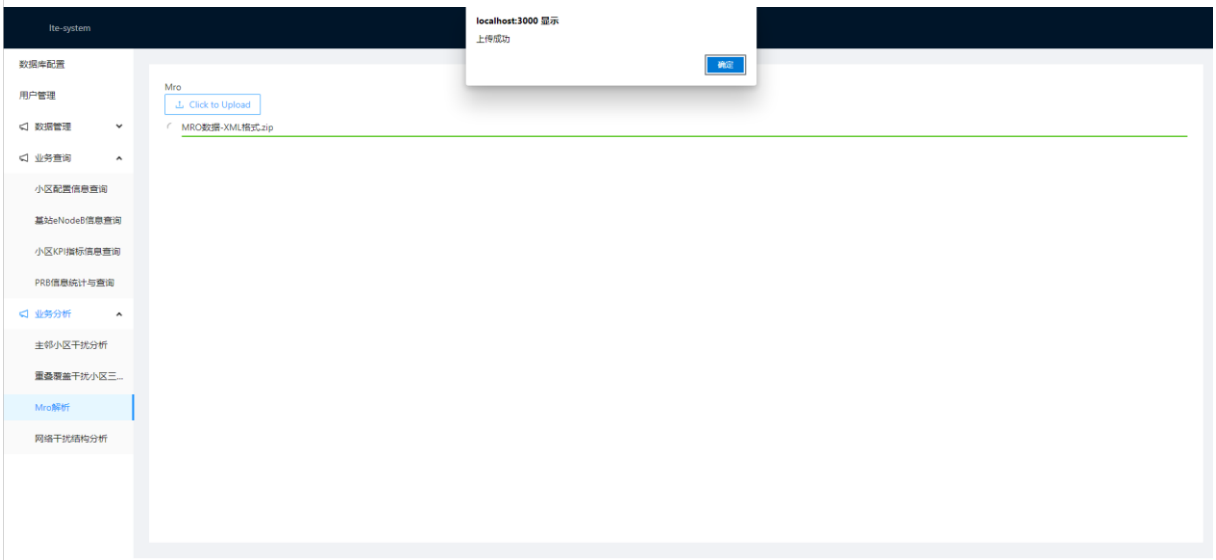
重叠覆盖干扰小区三元组分析需要输入阈值 x，用于后端接口的请求，返回值为生成的 c2i3 表的链接，需要再次访问文件流得到 csv 文件，再使用 parseCsv 的方法，将 csv 文件解析为可用的格式



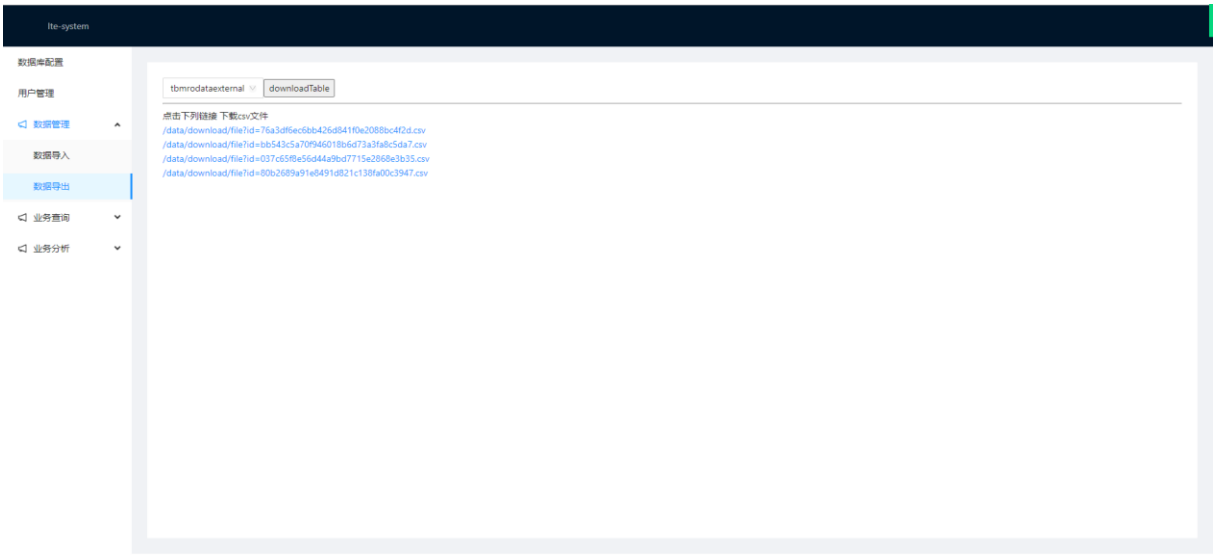
c. MRO/MRE 测量数据解析

上传 xml 格式文件，进度条显示上传进度，上传成功弹窗提示



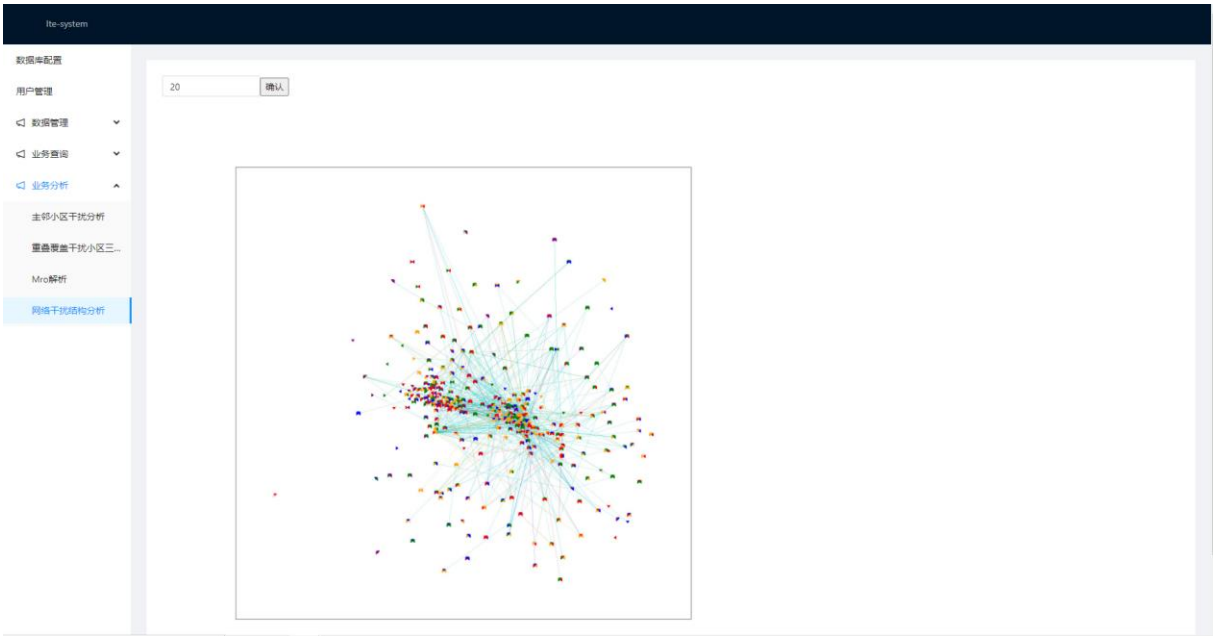


再去数据导出可查到四张 tbmrodataexternal 表



d. 网络干扰分析

设置阈值为 20，得到图片如下



六. 总结

本次实验使用前后端分离的开发模式进行编程。在开发过程中我们实现了高效的协同编程。后端开发与前端开发分别在不同机器上进行，这极大的提升了开发效率。在开发的过程中，面对需要处理的业务。我们首先分析业务需求、了解需要处理的方向，之后研究多种解决方案，并根据性能、复杂性、实现难度多方面综合考虑，最后制定出解决方案。在编程的过程中，我们也在不断遇到问题、分析问题、解决问题。这极大的提升了我们解决问题的能力，并且对 React、FastAPI 等开发框架有了更加深刻的理解。