

# Python与JavaScript在定制项目中的应用

---

## Python与JavaScript在定制项目中的应用

前言

应用

永杰铝业大屏与二级系统

场景

V1.0

思路

实现

V2.0

V3.0

V4.0

与平台第三方接口操作的互动（正荣集团BPM）

与平台ShellExecute()外部函数的互动

结语

## 前言

---

在定制项目开发的过程中，我们经常会遇到如下场景

- 部分业务模块或者整个业务系统需要与其他系统互联互通
- 系统内部之间需要进行复杂的逻辑处理与数据交互

平台的目的是保证定制系统70%的功能不需要进行二次开发。但是剩余的30%，由于平台的通用性，是无法向定制项目倾斜的，这时候就需要开发人员结合系统自身特点，进行代码层面的开发。由于Python和JavaScript的易用性与友好性，本文强烈推荐设计人员结合业务与系统，去了解并熟悉这两门语言。同时，本文也会使用已上线项目的例子，使设计人员初步了解这两门开发语言能在定制项目开发过程中起的作用。

本文推荐知识链接：

[stack overflow](#)

## 应用

---

### 永杰铝业大屏与二级系统

#### 场景

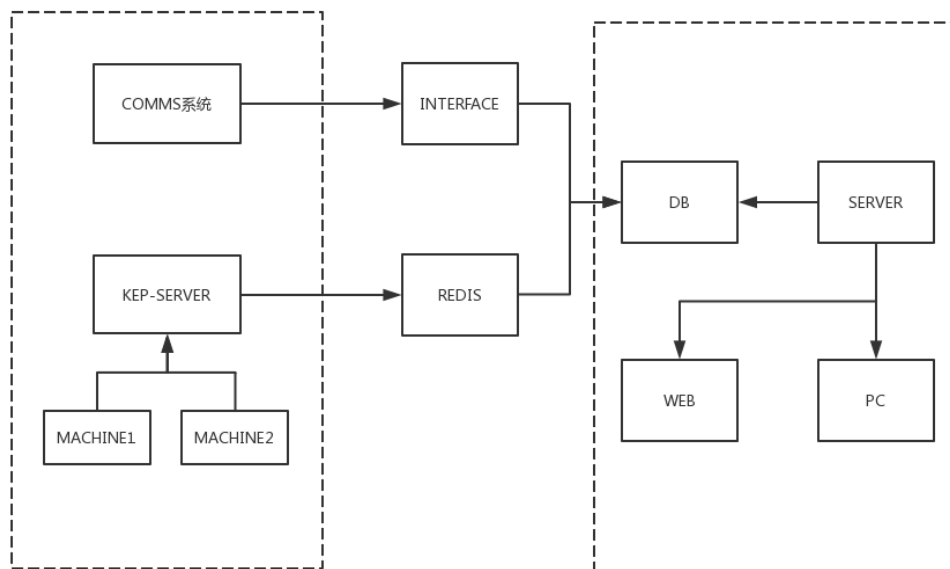
永杰铝业是一家以铝类为主要原料的生产型企业。用户有提出如下需求

- 系统以大屏的形式，展示各个车间，各个部门的生产与运营数据
- 以原有的二级系统为蓝本，将生产车间的设备运行数据进行采集，整理并发布，主要包括实时数据和历史数据

在用户工厂实地进行调研后，我们决定将百度的Echarts图表作为大屏展示的主力，以用户现有的KEP-SERVER系统为中转运站，采集设备数据。在设计实现之初到最终上线，我们更迭了几个版本，每个版本都有着比较大的技术拐点。接下来将——介绍

## V1.0

在第一个版本实现时，我们考虑的是系统的初步可见。此版本的数据流通情况如下图所示



在这个版本中，我们实现了与COMMS系统的数据连通。COMMS系统是用户内部所使用的一套报表汇集管理系统，它是我们大屏的数据来源。在我们和对方开发人员互相沟通后，决定以访问对方数据库的方式进行数据的对接。接下来，我们以INTERFACE和COMMS为范例，着重介绍Python在接口（数据库）开发中的应用。

### 思路

用户所需要的展示效果是接近于实时的，因此这个程序应该是一个自动循环运行的服务，而不是只跑一遍；

由于COMMS系统的机能限制，无法满足实时的数据交换，因此这个循环时间间隔需要能交给用户自己去调整；

由于数据展示的需要，需要在数据库外部进行数据的预处理后再写入数据库；

以“重点产品数据汇总表”为例：

整个接口分为3部分：读COMMS的数据；预处理数据；数据写入MySQL

### 实现

```
#读COMMS的数据
execute_sql = 'exec sp_API_Chenksoft ' + '@Type=' + "'" + '重点产品数据汇总表' +
"""
logging.debug(execute_sql)
while True:
    try:
        connect = pymssql.connect('192.168.50.5', 'Yongjie', '123', 'YJCommon')
        if connect:
            print('连接成功')
            logging.debug('数据库连接成功')
            break
```

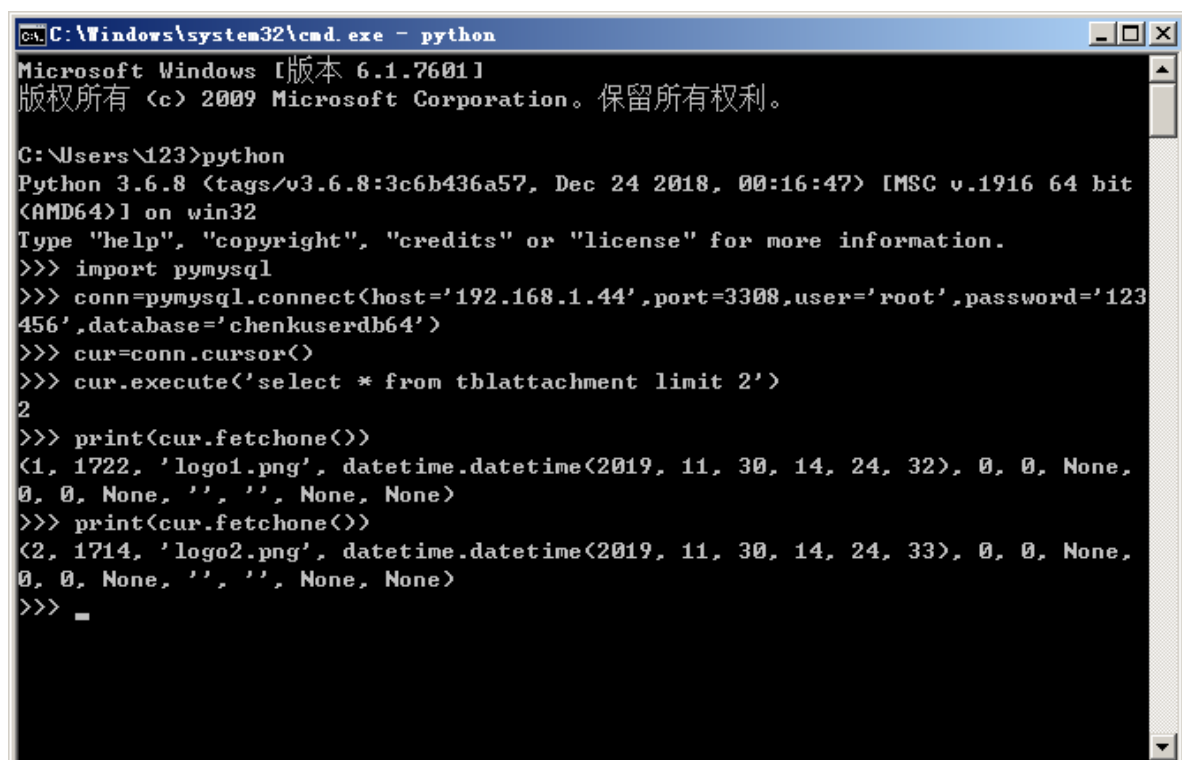
```

else:
    print('连接失败,重新连接')
    logging.debug('数据库连接失败,重新连接')
    continue
except Exception as e:
    print('异常,重新连接')
    logging.debug('连接异常,重新连接')
    continue
cursor = connect.cursor()
cursor.execute(execute_sql)
result = cursor.fetchall()

```

由于对方所使用的是SQL SERVER数据库，所以在连接操作对方数据库时，我们使用了Python的第三方库，遗憾的是作者宣布12月份后将不再对pymssql进行[更新与维护](#)。

pymssql提供了一系列友好的API供我们使用，例如上述代码中的 `pymssql.connect()`，是连接数据库的方法，用户需要传入HOST,DBNAME,USER,PWD参数；**开发人员需要一个变量用以接受连接方法的返回值，并以此判断连接是否成功，并判断接下来的操作。** `cursor=connect.cursor` 表示在此定义了一个**游标对象**，游标对象不唯一，可通 `connect.cursor()` 方法继续定义。游标的主要作用是用于执行数据库操作包括语句，存储过程，视图，函数等。代码里，定义了 `execute_sql`，同时也将 `execute_sql` 放在了 `cursor.execute()` 方法中去执行。执行返回的结果通过 `cursor.fetchone()` 或者 `cursor.fetchall()` 去获取，这两个方法通常用在查询的环境中，`fetchone()` 方法是游标获取完当前记录后指向下一条，`fetchall()` 方法是游标获取返回的所有记录后指向下一条，记录通常以元组形式存储。**需要注意的是，无论是 `fetchone` 还是 `fetchall`，由于其游标的特点，在获取完记录后，游标会指向下一条记录，上一条记录如果没有及时保存，就会丢失。**举例说明



```

C:\Windows\system32\cmd.exe - python
Microsoft Windows [版本 6.1.7601]
版权所有 (c) 2009 Microsoft Corporation。保留所有权利。

C:\Users\123>python
Python 3.6.8 (tags/v3.6.8:3c6b436a57, Dec 24 2018, 00:16:47) [MSC v.1916 64 bit
(AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import pymysql
>>> conn=pymysql.connect(host='192.168.1.44',port=3308,user='root',password='123
456',database='chenkuserdb64')
>>> cur=conn.cursor()
>>> cur.execute('select * from tblattachment limit 2')
2
>>> print(cur.fetchone())
<1, 1722, 'logo1.png', datetime.datetime(2019, 11, 30, 14, 24, 32), 0, 0, None,
0, 0, None, '', '', None, None>
>>> print(cur.fetchone())
<2, 1714, 'logo2.png', datetime.datetime(2019, 11, 30, 14, 24, 33), 0, 0, None,
0, 0, None, '', '', None, None>
>>>

```

可以看到，在执行完 `select * from tblattachment limit 2` 后，游标返回了两条记录，当使用 `fetchone` 方法时，第一次返回了第一条记录，第二次就返回了第二条记录了，如果没有及时记录，那么第一条记录就会丢失。在代码中，可以看到通过定义 `result` 去接收了返回的结果集。

当获取完数据后，需要对数据进行预处理，在这个场景中，我们需要判断每个数据项是否为空，如果为空，那么就需要一个空的字符串去替代；如果绝对值大于1，则取整；如果绝对值小于1，则保留小数

```
def empty(inputArg):
    if inputArg == None or abs(float(inputArg))==0:
        return 0

    if inputArg !=None and abs(float(inputArg))<1:
        return float(inputArg)
    if inputArg !=None and abs(float(inputArg))>=1:
        return round(float(inputArg))
```

```
for information in result:
    logging.debug(information)
    RECID = information[0]
    PrdtTypeKeyID = information[1]
    PrdtTypeKey = information[2]
    PrdtTypeKeyFZ = information[3]
    PrdtTypeKeySort = information[4]
    PrdtTypeKeyMB = empty(information[5])
    PrdtTypeKeyZR = empty(information[6])
```

当数据预处理完后，就需要写入MySQL当中，写入的方法有两种

- 通过[PyMySQL](#)第三方库，对数据库直接进行操作
- MySQL暴露数据库插入接口，外部通过网络将数据写入

当MySQL可以直接访问时，推荐使用第一种方法。当数据库无法直接访问时，可通过平台生成接口，程序通过[requests](#)库去访问URL链接，同时将数据当做参数一并传入。下面是两个例子

```
while True:
    try:
        url = 'http://192.168.30.38:8080/ckapi/api/1/v2/VipProductTableAna.jsp?'
        params=
        {'token': 'chenksoft!@!', 'RECID': RECID, 'PrdtTypeKeyID': PrdtTypeKeyID, 'PrdtTypeKey': PrdtTypeKey, 'PrdtTypeKeyFZ': PrdtTypeKeyFZ, 'PrdtTypeKeySort': PrdtTypeKeySort, 'PrdtTypeKeyMB': PrdtTypeKeyMB, 'PrdtTypeKeyZR': PrdtTypeKeyZR, 'PrdtTypeKeyZJ': PrdtTypeKeyZJ, 'PrdtTypeKeyRate': PrdtTypeKeyRate, 'PrdtTypeKeyFH': PrdtTypeKeyFH, 'PrdtTypeKeyWJ': PrdtTypeKeyWJ, 'PrdtTypeKeyBYWJ': PrdtTypeKeyBYWJ, 'PrdtTypeKeyXYWJ': PrdtTypeKeyXYWJ, 'PrdtTypeKeyBLWJ': PrdtTypeKeyBLWJ, 'PrdtTypeKeyZTWJ': PrdtTypeKeyZTWJ, 'PrdtTypeKeyZZWJ': PrdtTypeKeyZZWJ, 'PrdtTypeKeyXDWJ': PrdtTypeKeyXDWJ}
        r = requests.get(url, params=params).json()
        print('写入成功')
        logging.debug(r)
        logging.debug('写入成功')
        break
    except Exception as e:
        print(e)
        logging.debug(e)
        print('写入失败，重新写入')
        logging.debug('写入失败，重新写入')
        continue
```

可以看到，`url` 变量存储了平台生成的接口链接，`params` 存储了参数，通过 `requests.get()` 方法去调用链接，开发者需要注意的是，当URL请求无法访问时，需要捕获异常，并根据异常决定是否继续访问；另外由于GET有长度限制，当数据字节过大时，请使用 `requests.post()` 方法进行调用，平台接口也支持POST方法。当数据量过大时，通过接口操作数据库就会显得比较吃力，例如在INSERT时，接口一次只能INSERT 一条，如果外部数据有几万条，那么就会耗时非常长，这个时候，就可以通过 `pymysql` 库直接对数据库进行操作

```
count=[]
for information in result:
    logging.debug(information)
    OrdNum = information[2]
    DeviceName = information[4]
    ClientOrdNo = information[5]
    PurposeName = information[7]
    PrdtID = information[8]
    MinInputWeight = information[9]
    MatPlanWeight = str(empty(information[11]))
    InStockWeight = str(empty(information[12]))
    YieldRate = information[13]
    count.append(
        (
            OrdNum,DeviceName,ClientOrdNo,PurposeName,
            PrdtID,MinInputWeight,MatPlanWeight,InStockWeight,YieldRate,
            information[6]
        ),
    )
sql='INSERT INTO
SKT202(SKT202.SKf2516,SKT202.SKf2517,SKT202.SKf2518,SKT202.SKf2519,SKT202.SKf252
0,SKT202.SKf2521,SKT202.SKf2522,SKT202.SKf2523,SKT202.SKf2524,SKT202.SKf3132)
VALUES(%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s)'
cur.executemany(sql, count)
conn.commit()
cur.close()
conn.close()
```

首先定义一个列表 `count`，每次循环内，将各个数据拼合成一个元组 (`data1,data2,data3,data4`)，然后使用列表的 `append()` 方法，将每一个元组加入到列表中。循环完成后，这个列表就存储了以元组为单位的所有数据。在外部定义好sql语句后，使用 `executemany(sql,list)` 方法，`sql`为数据库语句，`list`为数据列表，需要注意的是，sql语句中的每个%s，都会与元组中的data——对应起来。最终执行的sql语句将会是 `insert into skt202(skt202.skf1,skt202.skf2)values(1,2),(3,4),(5,6),(7,9)...`

COMMS系统的接口总共有15个，通过复用上述的代码，完全可以实现与COMMS系统的数据互通。

## V2.0

在2.0版本中，我们遇到最大的一个问题是平台的Echarts图表库不够强大，不仅包括图表种类，还包括图表的各项属性无法在设计中定义。由于图表的复杂性，平台无法对所有图表进行兼容。所以我们决定将图表写成一个HTML页面，通过容器控件去加载。

以Echarts中的动态曲线图为例

首先定义布局以及相应的JS文件

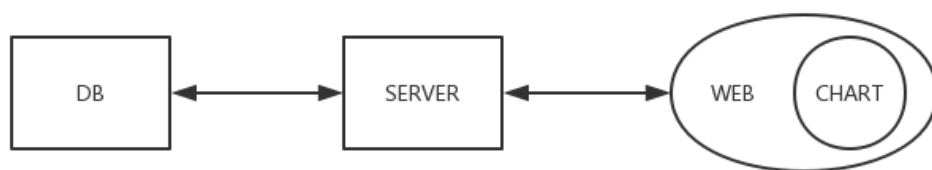
```
<div id="container" style="height: 100%"></div>
<script
  type="text/javascript"src="https://cdn.jsdelivr.net/npm/echarts/dist/echarts.mi
n.js"></script>
```

### 初始化图表实例

```
var dom = document.getElementById("container");
var myChart = echarts.init(dom);
var data=[]    //定义全局变量data
option = {
  title: {
    text: '动态数据 + 时间坐标轴'
  },
  xAxis: {
    type: 'time',
    splitLine: {
      show: false
    }
  },
  yAxis: {
    type: 'value',
    boundaryGap: [0, '100%'],
    splitLine: {
      show: false
    }
  },
  series: [{
    name: '模拟数据',
    type: 'line',
    showSymbol: false,
    hoverAnimation: false,
    data: data
  }]
};

myChart.setOption(option, true);
```

全局变量 `data` 中存放了需要展示的曲线数据，其数据结构为 `[{time1,[time1,value1]},{time2,[time2,value2]}...]`。通过外部函数 `ExecuteJS()` 将客户端查询到的数据传输给HTML；数据流转如下图



在一个时间循环内，WEB向SERVER发起数据库查询请求，SERVER将数据返回给WEB，WEB通过 `ExecuteJS()` 将数据传输给图表，图表最后显示数据。也就是说，图表的HTML文件需要有一个函数入口去承载 `ExecuteJS()` 的调用。

```
function timecall(time,value){
    data.push({time,value});
    myChart.setOption({
        series:[{
            data:data
        }]
    })
}
```

那么客户端的设计就需要变为 `ExecuteJS(T,timecall(time,value))` 其中，T为容器控件，`timecall(time,value)`需要设计人员提前拼接好。由于是定时读取数据，所以整个业务窗口需要套上一层定时控件。

在这个版本中，我们使用了容器控件来完成客户端无法实现的界面和功能。

### V3.0

在3.0版本中，我们遇到了一个比较大的问题：由于客户要求设备实时运行数据以毫秒级去展示，这就需要定时控件的时间间隔比较小。起初，界面的设计操作如下

1. 查询设备速度和厚度曲线数据
2. ExcuteJS给HTML传输数据
3. 查询设备运行的其他数据到界面

由于之前说到要在毫秒级，所以这三个操作被我们压缩在了2S之内，可是实际运行下来，却发现这三个操作的运行时长远远大于2S；同时由于WEB端的JS定时器机制，一旦在两秒内这三个操作没法执行完，就会立即开始下一个2S，最后导致的结果就是数据完全错乱。

我们决定整个实时数据部分从两个角度去考虑优化

- 将图表部分与平台剥离，以JS直接去访问数据库后台，不再通过 `ExecuteJS()` 传输数据。
- 建立垃圾数据删除机制

由于 `ExecuteJS` 的交互效率太低，我们在图表的JS部分加入了数据库访问机制，直接通过接口去获取数据。代码如下：

```
function get_speed_data(){
    var inputNum=[];
    var inputTime=[];
    $.ajax({
        type: 'get',
        url: 'http://127.0.0.1:8088/ckapi/api/1/v2/speeddata.jsp?',
        data: {'token': 'chenksft!@!', 'pointTime': pointTime, 'machineid': '4'},
        cache: false,
        async: false,
        success: function(getdata){
            console.log(getdata);
            if (getdata['data'].length!=0){
                pointTime=getdata['data'][getdata['data'].length-1]['time']
                flagTime=getdata['data'][0]['time'];
                console.log('in putdata');
                for (var i = 0; i < getdata['data'].length; i++) {
                    inputTime.push(getdata['data'][i]['time']);
                    inputNum.push(getdata['data'][i]['speed']);
                }
                putData(inputTime,inputNum);
            }
        }
    })
}
```

```

    }
});
    setTimeout("get_speed_data()", 500);
}

function putData(inputTime,inputNum){
    var time=[];
    var num=[];
    for (let i=0;i<inputTime.length;i++){
        var a=inputTime[i].substring((inputTime[i].lastIndexOf(':'))+1);
        var b=inputTime[i].substring(0,inputTime[i].lastIndexOf(':'));
        time.push(b+'.'+a);
    }
    for (let j=0;j<inputNum.length;j++){
        num.push(inputNum[j]);
    }
    for(var i=0; i<time.length; i++){
        var obj = {
            name:time[i],
            value:[time[i],num[i]]
        }
        data.push(obj)
    }
}

function paint(){
    if (data.length>0){
        data_run.push(data[0]);
        data.shift();
        myChart.setOption({
            yAxis: [
                {
                    min: ymin,
                    max: ymax,
                }
            ],
            series: [
                {
                    data: data_run
                }
            ]
        });
    };
    setTimeout("paint()", 200);
}

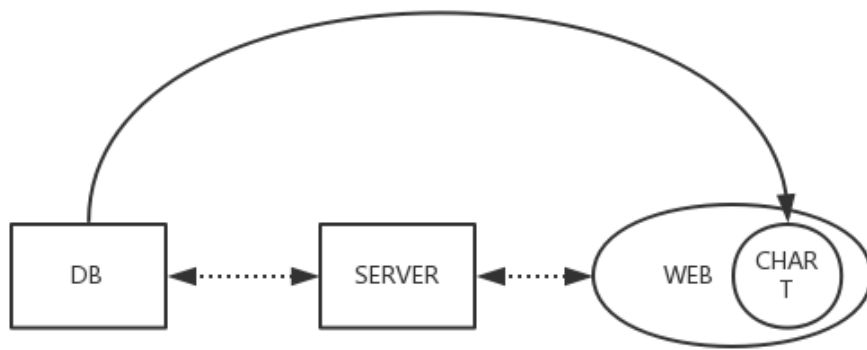
```

整个JS部分包括3个函数

1. `get_speed_data()` ,获取数据库的数据,定时500毫秒为周期执行
2. `putData()` ,数据预处理函数
3. `paint()` , 绘制曲线函数 , 定时200毫秒为周期执行

在 `get_speed_data()` 中 , 我们使用了 [AJAX](#) 来进行后台数据库的数据的获取 , 并交给 `putData()` 去处理数据 , 处理完成之后 , 将数据值传入全局变量中 , 之后在 `paint()` 中就可以直接调用。通过这种处理方式 , 数据的流转方式变得更为高效。





但是在这个过程中，我们发现，随着时间的增长，实时数据会线性累积，一天一个数据项的数据量大概在10W条，这样没过几天，数据库的数据量就会达到百万级，这还只是一个数据项。我们考虑，由于这些都是实时数据，在采集10分钟后，删掉这些对我们的业务没有任何影响。因此我们决定，每隔10分钟使用定时存储过程去执行DELETE操作，可是在实际情况中，我们发现，由于DELETE操作会对表进行锁定，所以如果没法再2毫秒内删除所有数据，就会对数据写入部分造成影响，为了保证速度，我们使用了sql中的TRUNCATE命令，该命令能以比DELETE更高的效率去删除数据，但是存储过程中无法去执行TRUNCATE，所以我们又写了个外部程序，并且定时10分钟去执行。

```
import pymysql
import time

if __name__=="__main__":
    while True:
        conn=pymysql.connect

        (host='127.0.0.1',port=3308,user='root',password='',database='chenkuserdb1')
        cur=conn.cursor()
        cur.execute('truncate table skt145')
        cur.execute('truncate table skt144')
        cur.execute('truncate table skt146')
        cur.execute('truncate table skt147')

        #总共超过20张数据表需要turncate

        cur.close()
        conn.close()
        print('next in 600')
        time.sleep(600)
```

上述是一个简单的10分钟循环，删除数据程序。

在完成了图表与客户端分离以及垃圾数据删除机制后，我们发现整个系统的前台展示过程非常流畅，所有的数据能在2S内返回结果并显示。

## V4.0

在4.0中，我们遇到的比较棘手的问题还是数据量大的问题，在3.0中我们解决了实时数据量大的问题。但是客户希望能保存所有历史数据并且前台能查询。这也就意味着，需要一套机制，将采集到的数据进行合理的存储。前文中提到一个点每天的数据量在10W，历史数据包括6个点，也就是6张表。一年的数据量为21900W，两台机器就是43800W，也就是43亿条数据。这43亿条数据放在6张表里是不合适也是不合理的。我们考虑进行分表操作，也就是说每天为6个点建立6张表，保证每张表的数据在10W条左右，这样子一年也仅仅是2000张表的增量，完全在可承受范围之内。

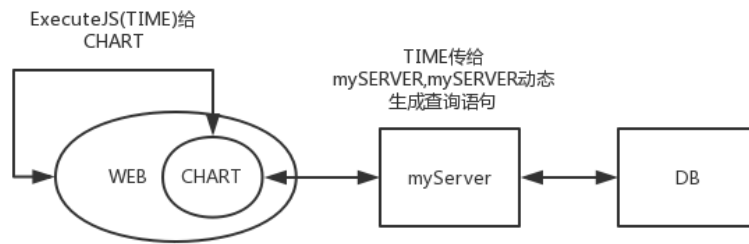
#以一个点为例子

```
def timecall():
    now=datetime.datetime.now()
    Y_M_D=str(now.strftime('%Y_%m_%d'))
    finish_length={}
    finish_length=r_redis.hgetall('Number4.YJ_PLC400.line008')
    count=[]
    #合金
    alloy={}
    alloy=r_redis.hgetall('Number4.YJ_PLC400.line053')
    if alloy!={}:
        sql_create_alloy='create table if not exists alloy_%s (num bigint(20)
not null auto_increment,time varchar(255) default NULL, value varchar(255)
default NULL, machineid varchar(255) default NULL,PRIMARY KEY (num),KEY(time))'
        %(Y_M_D)#建表操作
        cur.execute(sql_create_alloy)
        conn.commit()
        alloy_table='alloy_'+Y_M_D
        for key,value in alloy.items():
            count.append((str(key),str(value.split(';')[0]),str(value.split(';')
[1])),)
        sql_insert_alloy='insert into '+alloy_table+' (time,value,machineid)
values(%s,%s,%s)'#数据插入
        cur.executemany(sql_insert_alloy,count)
        conn.commit()#记住要commit
        count=[]

if __name__=="__main__":
    while True:
        timecall()
        time.sleep(2)
```

在上述代码运行时，首先会连接MYSQL数据库以及REDIS，然后每隔2S执行timecall()函数，在timecall()中，我们在对数据alloy进行非空判断后，马上就进行了建表操作，该语句执行完后，无论成功与否，数据库中都会有该点当天的数据表，随后程序就开始写入数据。这样子，43亿条数据被我们以分表的形式“瓜分”掉。

在以日期进行分表操作后，也给前台数据查询留下了窗口。由于查询都是以日期时间为查询条件，所以要查询的表名需要动态生成，但平台不支持表名动态传入，因此我们后台搭建了个服务，去接收前台传入的时间参数，在后台进行动态查询，并返回数据给前台。



在其中，myServer服务起了中转的作用，在Python语言中，我们经常使用Flask框架来作为服务的承载，亦如JAVA中的Struts框架等。服务需要有个端口去监听到来的访问，并接受访问的参数，同时要将结果集返回。举个简单的例子：

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def hello_world():
    return 'Hello World!'

if __name__ == '__main__':
    app.run('0.0.0.0', port=80)
```

当程序运行，并且用户访问这台机器的80端口时，他的浏览器上就会弹出“HELLO,WORD”。现在我们来完善上述代码。

#### 1. 接收参数

```
endTime = request.args.get('endTime')
machineid=request.args.get('machineid')
calltype=request.args.get('calltype')
speed_table_name='speed_'+str(startTime.split(' ')[0]).replace('-', '_')
```

#### 2. 生成动态查询语句并且访问数据库

```

conn=pymysql.connect(host='127.0.0.1',port=3308,user='root',password='',data
base='chenkuserdb1')
cur=conn.cursor()
sql_select_speed="select time, value from %s where machineid=%s and
time>='%s' and time<='%s' order by time asc " %
(speed_table_name,machineid,startTime,endTime)
cur.execute(sql_select_speed)
for information in result:
    speed_inf={}
    speed_inf['time']=information[0]
    speed_inf['value']=information[1]
    result_list.append(speed_inf)
cur.close()
conn.close()

```

### 3.返回数据集

```

return {'data':result_list} #FLASK不支持以列表形式返回，需要做成字典形式

```

一个简单的动态生成查询语句的后台服务就完成了。在服务启动后，前台就可以访问并且获取数据了。

```

function get_data(startTime,endTime,machineid){
    console.log(startTime);
    console.log(endTime);
    $.ajax({
        url:"http://127.0.0.1:9000/get_data?",
        type:'get',
        data:
        {'startTime':startTime,'endTime':endTime,'machineid':machineid,"calltype":"1"},
        async:false,
        cache:false,
        success:function(getdata){
            console.log(getdata);
            for (var i=0;i<getdata['data'].length;i++){
                tempTime.push(getdata['data'][i]['time']);
                tempValue.push(getdata['data'][i]['value']);
            }
            putData2(tempTime,tempvalue);
        }
    })
}

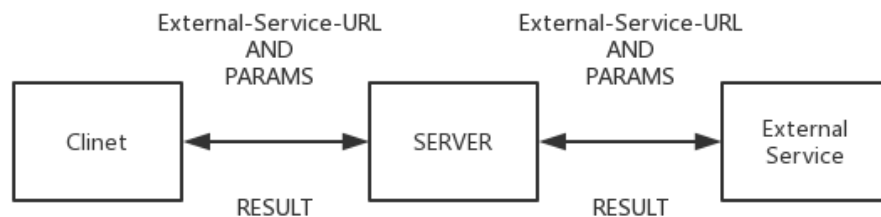
```

在分表以及动态查询的加持下，用户在前台看到图像的延迟在2S内，完全符合用户以及体验的要求。

在永杰这个项目中，我们以平台服务为中心，使用了大量的微服务来增强用户的体验，并且从数据库结构进行了优化，成功接受了大数据量的考验。

## 与平台第三方接口操作的互动（正荣集团BPM）

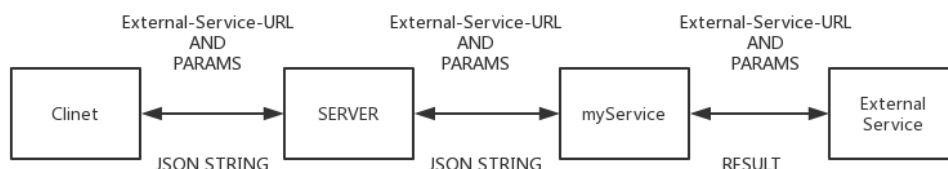
在平台中，有可以对外部接口直接调用的操作——调用第三方接口操作。这个操作能在平台内直接把控件的相关值当做参数传入，直接调用外部的接口。



该通路为一条环线：

1. Client向SERVER发起请求，告诉SERVER访问哪个链接，并将参数一并传给SERVER
2. SERVER接收到请求和参数后，向最终的External\_Service发起调用
3. SERVER接收到回传的RESULT并解析，返回客户端对应绑定的控件中

但是这条环线有一个缺点，SERVER只能以JSON字符串的形式发送给External\_Service,并且也只能解析规定格式的回传JSON字符串。如果当External不接受JSON字符串或者回传的不是JSON字符串时，环路就会断开。因此我们需要有一个外部服务将数据进行二次包装，包装成符合External\_Service格式的数据，在接收到回传数据后，也需要将数据包装成JSON字符串返回给SERVER。



我们以正荣集团外部BPM接口为例，详细介绍外部服务如何与第三方接口操作进行互动

正荣集团的接口传输格式以XML字符串格式为主，完全不符合平台的解析标准，因此一定需要再添加个外部服务来进行数据解析。我们以上一章提到的FLASK框架为例。

首先，设置监听路由

```
@app.route('/requestlist',methods = ["POST"])
def request_list():
```

例：当SERVER以POST形式访问127.0.0.1:80/requestlist?时，将会执行request\_list函数。

然后，获取传过来的数据

```
requestList=eval(bytes.decode(request.get_data()))  
print(requestList)  
Num=requestList['requestList']
```

由于SERVER将数据放在了BODY中，所以需要将BODY中的数据解析出来。上两步操作在客户端中对应的操作如下

第三方接口信息

名称:

requestlist

调用名:

requestlist

调用地址:

http://192.168.1.90:60/

描述:

参数:

ID	名称	参数类型	描述
2	requestList	字符型	

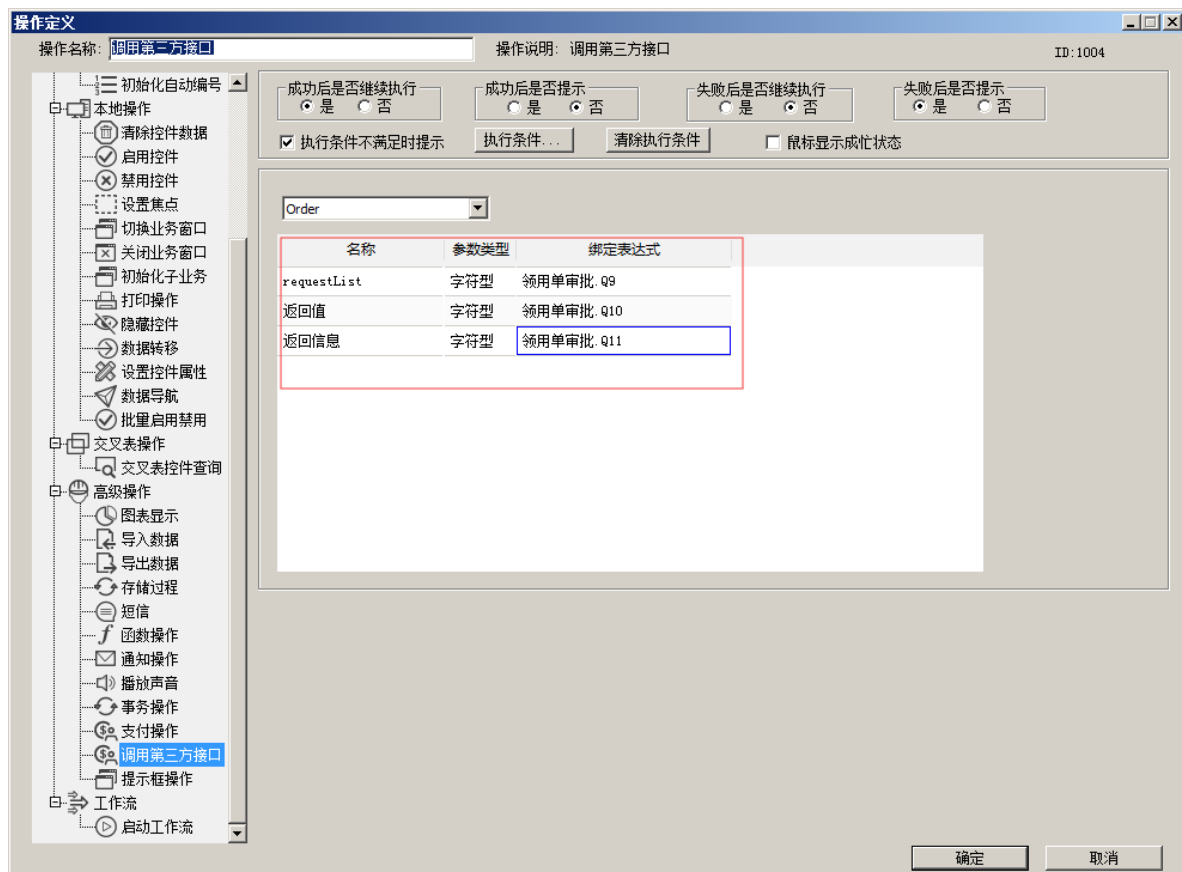
新增

修改

删除

确定

取消



在接收到数据后，我们可以根据具体的业务逻辑，或对数据进行二次包装，或根据传入的数据去查询其他数据。在BPM流程中，客户端仅仅需要传输一个编号，我们需要根据这个编号去数据库进行查询相关信息，并进行拼接。由于数据处理部分需要结合实际情况，所以BPM的数据处理部分代码不再赘述。

在数据处理完毕后，我们向最终的URL发起调用请求，接收到结果并处理后，我们要么将数据返回给SERVER,由SERVER返回给客户端，要么将数据写入数据库中，并将成功调用的消息返回给SERVER，由SERVER返回给客户端。

```
r=requests.post(url,headers=headers,data=finalString.encode("utf-8")).text
print(r)
returnResult={"code":"0","data":"成功","msg":"成功"}
return json.dumps(returnResult)
```

上述的过程以代码形式表现

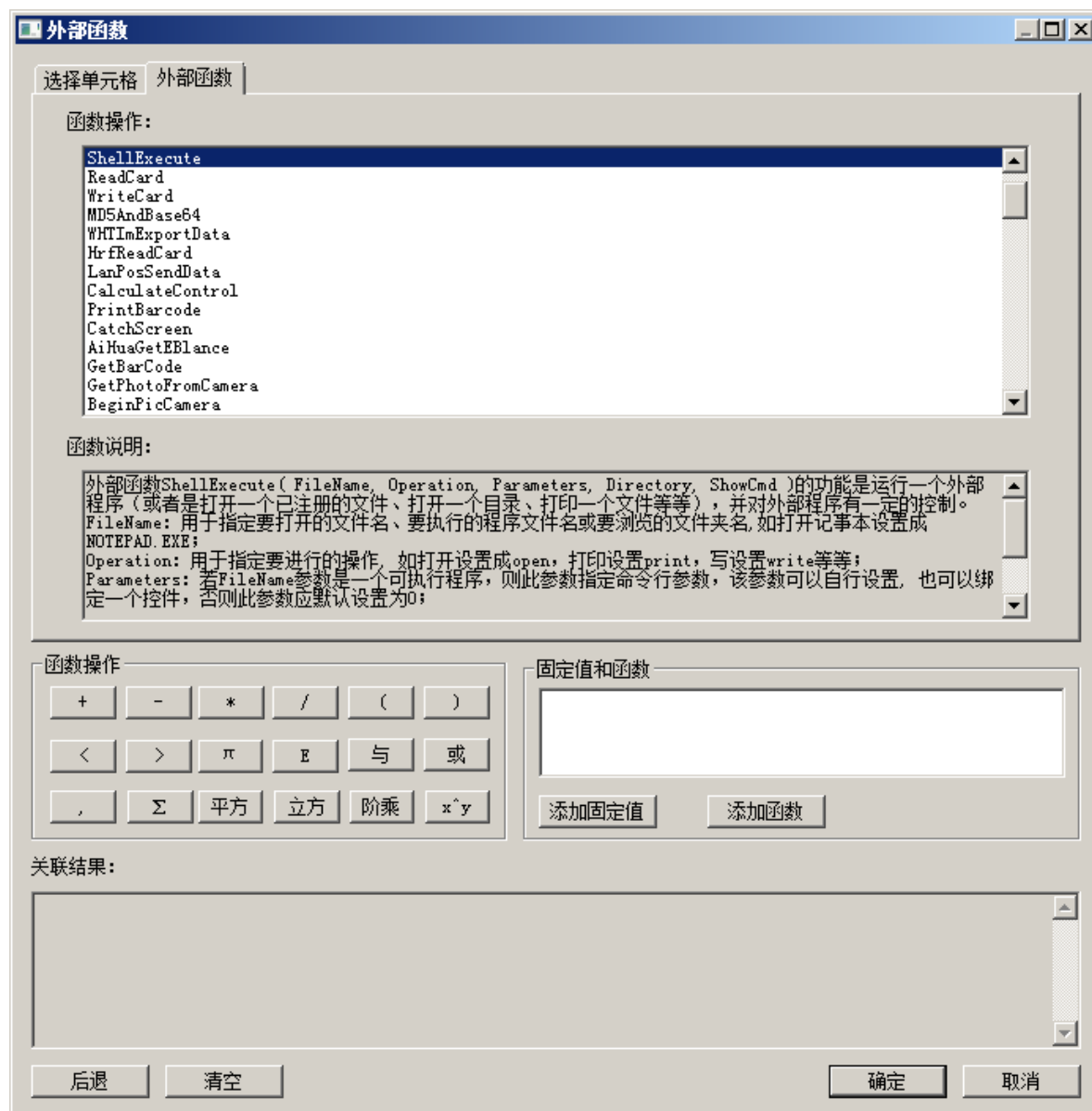
```
@app.route('/requestlist',methods = ["POST"])
def request_list():
    #该部分为数据接收部分
    requestList=eval(bytes.decode(request.get_data()))
    print(requestList)
    Num=requestList['requestList']
    #该部分为数据处理部分（略）

    #该部分为数据返回部分
    r=requests.post(url,headers=headers,data=finalString.encode("utf-8")).text
    print(r)
    returnResult={"code":"0","data":"成功","msg":"成功"}
    return json.dumps(returnResult)
```

所有的第三方服务都可以通过对上述代码进行扩展，来达到需求。

## 与平台ShellExecute()外部函数的互动

在平台中，有个外部函数ShellExecute(),它的作用是执行客户端本地的应用程序



在Python中，有一个第三方叫做[Pyinstaller](#),它的作用是将Python代码打包成为Windows下的可执行程序(EXE)，将这两个结合起来，我们就可以对ShellExecute进行自定义扩展。

以上海卫星工程研究所项目为例

用户以SURFACE为主要设备进行离线巡检，以PC为服务器进行数据存储。PC和SURFACE各有一套服务端+客户端。用户希望在巡检完成后，在SURFACE上将巡检数据上传至PC，包括附件。其中有个数据难点，那就是无法将附件一步传到位。比如SURFACE中该记录的附件ID为6，待上传数据和附件后，记录中的附件ID仍然为6，但是附件表中该附件的ID有可能不为6，这是由于附件表ID自增引起的。因此，数据和附件需要分开传，并且先上传附件，将回传的附件ID，作为记录中的新附件ID传过去。

- 将表中所有上传标记为0的数据查询出来
- 查询每一条记录中的附件ID
- 根据附件ID，去查询附件表中的源文件名
- 先调用文件上传接口，将文件传输过去，并得到返回的附件ID
- 将数据中的旧附件ID改为新的附件ID
- 将数据传输过去
- 将整个程序打包为EXE，设计中设置个按钮调用ShellExecute操作，执行外部的EXE



外部函数

选择单元格

外部函数

当前模块：

点检管理

当前业务：

点检记录

1

2

3

4

5

6

7

8

9

10

11

12

13

14

A

B

C

D

E

F

G

H

I

J

K

L

M

N

点检记录

下载查询所有点检记录

点检编号

点检人员

点检时间

序号

详情

上传标记

点检编号

函数操作

+

-

\*

/

(

)

<

>

π

E

与

或

,

Σ

平方

立方

阶乘

x^y

固定值和函数

添加固定值

添加函数

关联结果：

ShellExecute ( sateline.exe , open , -1 )

后退

清空

确定

取消

晨科软件

上海宇航系统工程研究所PAD ( 805所 )

Admin

点检管理

点检记录

点检编号

点检人员

点检时间

点检时间

设备编号

设备名称

上传附件1成功

上传附件2成功

上传附件3成功

上传附件4成功

上传附件5成功

上传附件6成功

上传附件7成功

上传附件8成功

上传附件9成功

上传附件10成功

上传附件11成功

上传附件12成功

上传附件13成功

上传附件14成功

上传附件15成功

上传附件16成功

上传附件17成功

上传附件18成功

上传附件19成功

上传附件20成功

上传附件21成功

上传附件22成功

上传附件23成功

上传附件24成功

上传附件25成功

上传附件26成功

上传附件27成功

上传附件28成功

上传附件29成功

上传附件30成功

上传附件31成功

上传附件32成功

上传附件33成功

上传附件34成功

上传附件35成功

上传附件36成功

上传附件37成功

上传附件38成功

上传附件39成功

上传附件40成功

上传附件41成功

上传附件42成功

上传附件43成功

上传附件44成功

上传附件45成功

上传附件46成功

上传附件47成功

上传附件48成功

上传附件49成功

上传附件50成功

上传附件51成功

上传附件52成功

上传附件53成功

上传附件54成功

上传附件55成功

上传附件56成功

上传附件57成功

上传附件58成功

上传附件59成功

上传附件60成功

上传附件61成功

上传附件62成功

上传附件63成功

上传附件64成功

上传附件65成功

上传附件66成功

上传附件67成功

上传附件68成功

上传附件69成功

上传附件70成功

上传附件71成功

上传附件72成功

上传附件73成功

上传附件74成功

上传附件75成功

上传附件76成功

上传附件77成功

上传附件78成功

上传附件79成功

上传附件80成功

上传附件81成功

上传附件82成功

上传附件83成功

上传附件84成功

上传附件85成功

上传附件86成功

上传附件87成功

上传附件88成功

上传附件89成功

上传附件90成功

上传附件91成功

上传附件92成功

上传附件93成功

上传附件94成功

上传附件95成功

上传附件96成功

上传附件97成功

上传附件98成功

上传附件99成功

上传附件100成功

设备名称

型号规格

设备类型

存放位置

入库日期

备注

窗口

控件

描述

原始SQL

SQL

1

事件

3017940

耗时: 0 毫秒

点检管理

点检记录

上传记录

数据单点

2

操作

2405

耗时: 0 毫秒

执行操作: 点检操作

其他代码不再赘述，介绍文件上传接口代码

```
def uploadFile(path,fileList):
```

```

#定义空的LIST存放返回后的附件ID
#returnList=[]
#读取fileList循环上传文件
returnList={"pic1":"","pic2":"","pic3":"","pic4":"","pic5":"","pic6":""}
for i in range(1,len(fileList)+1):
    if fileList[i-1]==None or fileList[i-1]=='':
        #returnList["pic"+str(i)]=""
        continue
    filename=fileList[i-1]
    print(path+str(filename))
    #判断文件是否存在
    if not (os.path.exists(path+filename)):
        print('附件目录下无该附件')
        logging.debug('无附件'+str(filename))
        continue
    else:
        #查询ORIFILENAME
        cur.execute('select orifilename from tblattachment where id=%s'%(filename))
        orifilename=cur.fetchone()[0]
        #拷贝文件
        #当前目录下创建tempFile文件夹
        if not (os.path.exists('tempFile')):
            os.mkdir('tempFile')
        shutil.copy(path+filename,'tempFile/'+orifilename)
        #开始上传附件
        data={'domainid':domainid}
        f=open('tempFile/'+orifilename,'rb')
        files={orifilename:f}
        print(files)
        k=0

        while k<10:
            try:
                r=requests.post(upload_url,data=data,files=files).json()
                print(r)

                returnList["pic"+str(i)]=r['data']
                #print(r)
                logging.debug(r)
                print('上传附件'+str(filename)+'成功')
                break
            except Exception as e:
                logging.debug(e)
                print(e)
                k=k+1
        #删除tempFile下的附件
        f.close()
        os.remove('tempFile/'+orifilename)
print(returnList)
return returnList

```

代码完成后，我们可以通过Pyinstaller将代码打包成EXE程序。在控制台输入 `pyinstaller -F --onefile example.py` 即可在当前目录的dist文件下看到生成的exe文件，将exe放在客户端目录下，即可调用。

## 结语

---

本文从项目实际需求出发，分析并找到合适的方法来解决。本文提供了一种新的设计思路，就是以平台为主干，以各种微服务为分叉，有的做中间服务，有的做数据整理，有的做优化数据库等等。最主要的是，开发语言不是重点，在我们知晓相关逻辑后，可以通过各种语言去实现，不仅仅包括Python和JavaScript。

本文的编辑器为[Typora](#)，该编辑器为Markdown编辑器。

永杰项目代码 <https://git.chenksoft.com/customize/yongjie>

正荣项目代码<https://git.chenksoft.com/customize/ZhengRong>

宇航工程研究所代码 <https://git.chenksoft.com/zhuhongjie/customize/tree/master/sateline>

希望各项目组能在项目完成后，将需求和设计思路相结合，以文档的形式记录。

下一篇：[OAUTH协议下的单点登录与组织架构同步实现](#)(未完成)