

设计系统功能性需求API:

1. 基于实际需求, 我们设计的function有:

- get_enterprise_order : 检索查看某个客户公司的所有订单, 便于分析客户需求, 为客户提供更好的服务。
- get_center_stock: 检索查看某个供应中心产品的型号, 库存, 售价和销售总额, 以便供应中心及时补货、调整售价。

2. 我们还设计了一个trigger:

- contract_type_trigger: 在向订单表orders中插入订单记录时, 根据 *estimate_delivery_date*, *lodgement_date* 和当前系统时间 *current_date* 自动补全订单状态: finished / unfinished.

3. 考虑到公司需要根据市场和行情动态调整上市产品, 我们设计了**更改或删除** product表的 trigger:

```
create trigger product_update_trigger
  before update
  on project_2.public.product
  for each row
execute procedure product_check();

create trigger product_delete_trigger
  before delete
  on project_2.public.product
  for each row
execute procedure product_delete_check();
```

- product_update_check(): 保证更改product表中产品信息时, 同步更新后信息到订单、库存等表。
- product_delete_check(): 保证删除某个product, 同时删除库存等表中的产品信息。

Index:

I. 我们尝试对orders的product_model, contract_manager 和 salesman_num列建立索引, 原因如下:

1. product 和 staff 是给定情形中的"基础表", 较少涉及更改操作且较频繁地作为查询条件。
2. orders表是staff表和product表的从表。orders表的product_model, contract_manager 和 salesman_num列是外键列。

如果从表没有包含外键列的索引, SQL 服务器需要扫描整个从表。从表越大, 删除更新等操作的时间越长。且在高并发情形下容易造成阻塞。如果主表有唯一聚集或非聚集的索引, 在从表中插入或修改时, 能利用主表的索引快速定位。

3. orders中的每一条订单, 一旦形成并添加订单信息后, 在现实情况下不易涉及更新操作。

```
--btree 对文本模糊匹配表现更好
CREATE INDEX product_index ON project_2.public.orders USING btree(product_model);
explain select * from project_2.public.orders where product_model like 'Photo%';
--hash 适用于等值匹配
CREATE INDEX salesman_index ON project_2.public.orders using hash(salesman_num);
CREATE INDEX manager_index ON project_2.public.orders using hash(contract_manager);
explain select * from project_2.public.orders where orders.salesman_num =
'11110405';
```

II. 对于stock表，建立表达式索引，更便于根据销售数量搜索产品。

```
CREATE INDEX sales_num_index ON stock ((stock.quantity-stock.current_quantity));
explain select * from stock where stock.quantity-stock.current_quantity between 0
and 10;
```

```
QUERY PLAN
1 Seq Scan on stock (cost=0.00..10.72 rows=2 width=40)
2 Filter: (((quantity - current_quantity) >= 0) AND ((quantity - current_quantity) <= 10))
```

```
QUERY PLAN
1 Bitmap Heap Scan on stock (cost=4.17..7.97 rows=2 width=40)
2 Recheck Cond: (((quantity - current_quantity) >= 0) AND ((quantity - current_quantity) <= 10))
3 -> Bitmap Index Scan on sales_num_index (cost=0.00..4.17 rows=2 width=0)
4 Index Cond: (((quantity - current_quantity) >= 0) AND ((quantity - current_quantity) <= 10))
```

```
QUERY PLAN
1 Seq Scan on orders (cost=0.00..15.61 rows=1 width=89)
2 Filter: (salesman_num = '11110405'::bpchar)
```

```
Output hash 适用于等值匹配
QUERY PLAN
1 Index Scan using salesman_index on orders (cost=0.00..8.02 rows=1 width=89)
2 Index Cond: (salesman_num = '11110405'::bpchar)
```

Role:

基于实际需求，除了superuser postgres以外，我们新建了以下三个Role

- 管理产品信息的product_manager：具有product表的所有权限
- 代替超级用户管理数据库的database_manager：具有创建数据库和创建用户的权限
- 管理xxx供应中心的管理员xxx_center_manager：具有该供应中心的产品库存视图以及该供应中心员工信息视图的所有权限

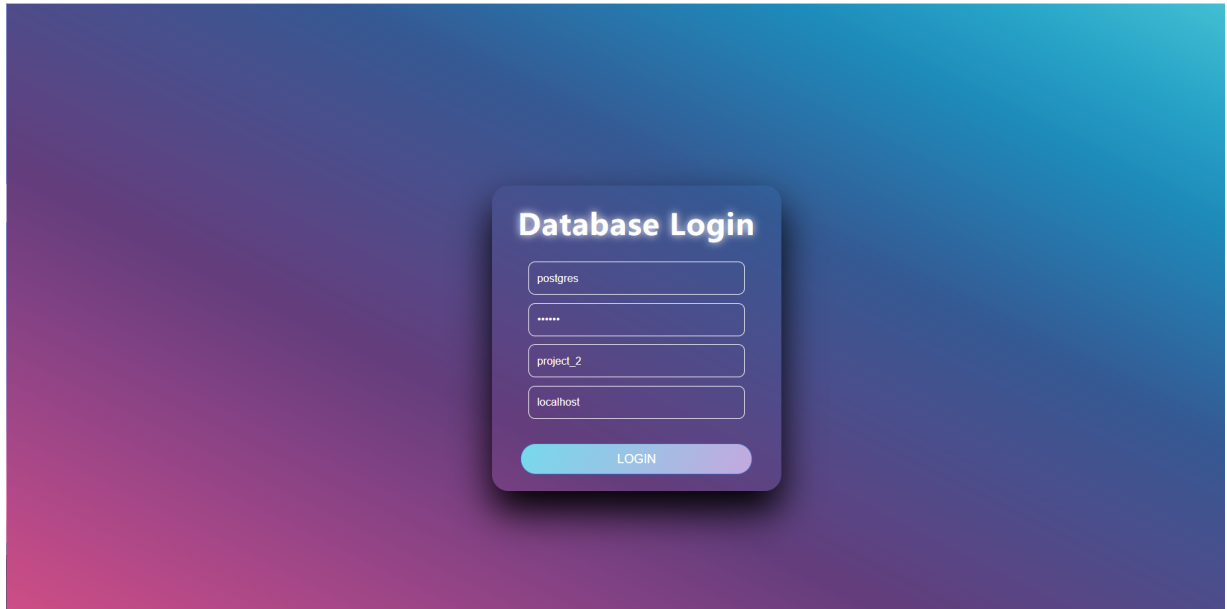
america_center_manager database_manager product_manager

前端

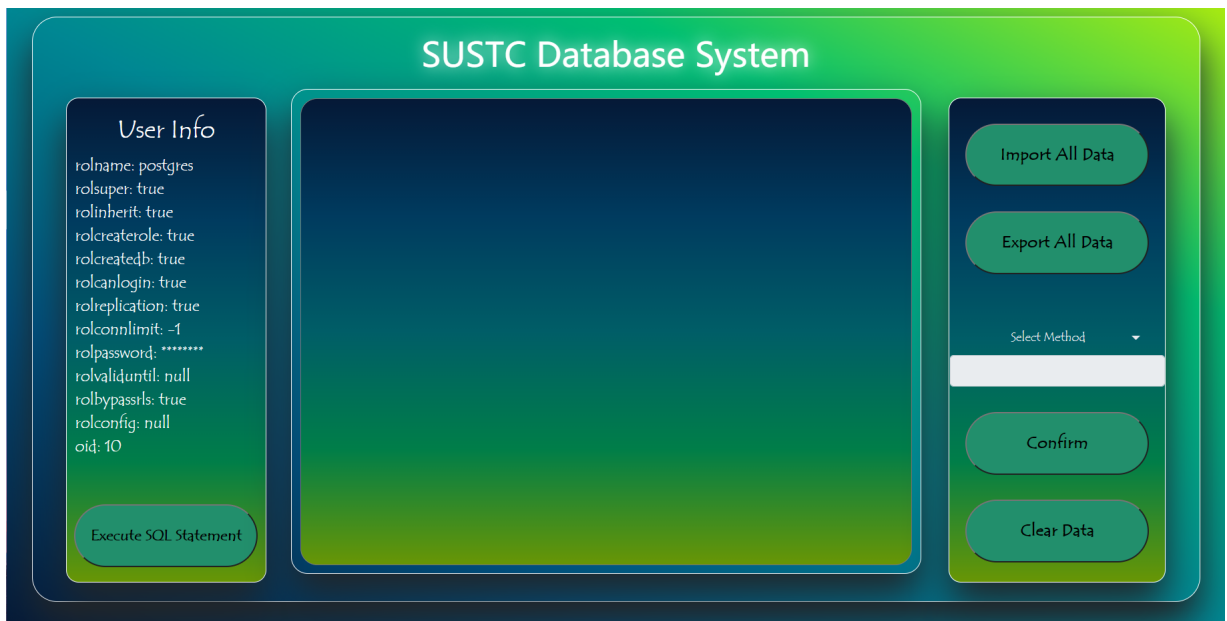
此次前端我们主要实现两个页面，一个是登录页面，用于用户验证，一个是数据库管理页面

以下是两个页面的截图

login界面：



database界面：



在登录验证时，用户需要输入四项信息，分别是：用户名，密码，数据库以及主机地址，在用户输入正确的信息之后，后端将会连上数据库并初始化数据库连接池，以便之后使用

前端支持的操作有：

1. 一键导入操作
2. 一键导出操作（将查询结果保存为txt文件）
3. Q6-Q13的查询操作
4. Q12-Q13的自动补全（输入一部分字符可以自动补全其余字符）
5. 面板清空操作（只是清空前端显示，不会清空后端的查询结果）

5. 手动输入SQL语句，并将执行结果打印到工作台中（如果是select语句的话）

数据库连接池

本次数据库我们所使用的的连接池是python的DBUtils包下的PooledDB

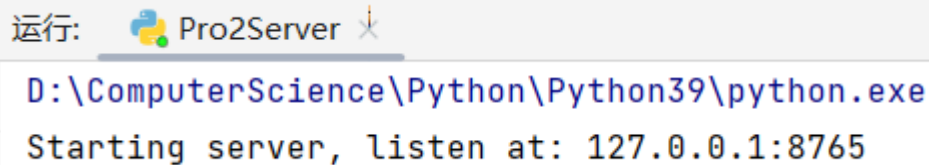
```
pool = PooledDB(  
    creator=pg,  
    mincached=1,  
    maxcached=20,  
    blocking=True,  
    port=5432,  
    database='project_2',  
    user='postgres',  
    password='123456',  
    host='localhost',  
    ping=0  
)
```

初始连接设置为1，最大连接设置为20，blocking设置为True

后端

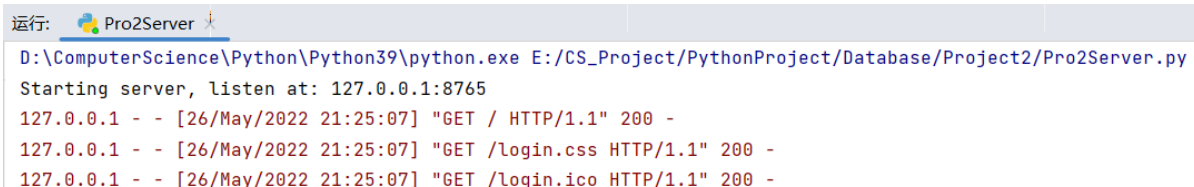
后端支持http连接和RESTful服务，http连接用于前后端交互，RESTful服务用于一些复杂查询，然后后端利用psycopg2来连接数据库

运行成功的画面是这样：



```
运行: Pro2Server  
D:\ComputerScience\Python\Python39\python.exe  
Starting server, listen at: 127.0.0.1:8765
```

遇到请求时处理结果如下：



```
运行: Pro2Server  
D:\ComputerScience\Python\Python39\python.exe E:/CS_Project/PythonProject/Database/Project2/Pro2Server.py  
Starting server, listen at: 127.0.0.1:8765  
127.0.0.1 - - [26/May/2022 21:25:07] "GET / HTTP/1.1" 200 -  
127.0.0.1 - - [26/May/2022 21:25:07] "GET /login.css HTTP/1.1" 200 -  
127.0.0.1 - - [26/May/2022 21:25:07] "GET /login.ico HTTP/1.1" 200 -
```

分布式设计：

1. 使用PostgreSQL数据库的分布式中间件Citus

我们首先考虑使用Citus解决PostgreSQL横向扩展问题，以支持更大的数据量、更大的写入和查询性能。

我们分别在两台主机的Linux系统(wsl)上部署citus（安装citus扩展），并将添加结点位本机IP。

在各个节点上，开放5432端口。

```

njl@LAPTOP-09F731HD:~$ sudo -i -u postgres psql -c "SELECT * FROM citus_get_active_worker_nodes();"
node_name | node_port
-----+-----
(0 rows)

njl@LAPTOP-09F731HD:~$ sudo -i -u postgres psql -c "SELECT * from citus_add_node('localhost', 5432);"
[sudo] password for njl:
citus_add_node
-----+-----
1
(1 row)

njl@LAPTOP-09F731HD:~$ sudo -i -u postgres psql -c "SELECT * FROM citus_get_active_worker_nodes();"
node_name | node_port
-----+-----
localhost |      5432
(1 row)

njl@LAPTOP-09F731HD:~$ sudo -i -u postgres psql -c "SELECT * from citus_add_node('172.23.64.1', 5432);"
WARNING: could not establish connection after 30000 ms
ERROR: connection to the remote node 172.23.64.1:5432 failed
njl@LAPTOP-09F731HD:~$

h1814071380@黄柯睿的电脑:~$ sudo -i -u postgres psql -c "SELECT * FROM citus_get_active_worker_nodes();"
node_name | node_port
-----+-----
(0 rows)

```

由于wsl的IP是本地内IP，本地的IP又属于校园网内网IP，不是静态IP，因此要考虑部署NAT网络地址转换和端口映射。使得两台主机（服务器）能够互相通信，从而实现分布式。具体实现可以在docker中进行配置，此处我们了解概念后没有具体实现。

2. 手动实现简单分布式

由于上面的那一种方式需要公网IP，或者云端服务器，并且由于我们使用的wsl所在网段是我们电脑的子网，不在校园网的子网下，因此并不能绑定校园网的IP，并且我们的校园网服务器给我们分配的ip是随时间动态改变的，切换的时候会很不方便。因此我们最终放弃了上面的那种分布式设计，我们准备手动实现我们自己的分布式。

我们的想法是另开一个proxy.py作为代理服务器，里面维护一个服务器列表，当有查询结果的时候，从服务器列表中选择一个服务器来进行查询，并返回查询结果，当需要做出改变的时候（如update，delete等），所有的服务器都需要改变。这样做的好处就是减少高并发带来的服务器压力，并且避免单个服务器宕机引起查询失败。代理服务器绑定的IP就是校园网IP，这样，只要设备和代理服务器处于同一个子网下（也就是校园网），就都可以拿来当服务器。

压力测试：

压力测试这部分我们使用的是Apache下的Jmeter软件来进行压力测试。软件是基于java编写的。

我们最后选择测试的线程数是：

1server（无分布式）：400线程、600线程、800线程、1000线程

2server：1000线程、1200线程、1400线程、1600线程、1800线程

1server时结果如下：

400线程：

线程组.jmx (C:\Users\john\Desktop\线程组.jmx) - Apache JMeter (5.4.3)

文件 编辑 查找 运行 选项 工具 帮助

Test Plan

- 线程组
 - HTTP请求默认值
 - HTTP请求
 - 响应断言
 - 察看结果树
 - 汇总报告

汇总报告

名称: 汇总报告

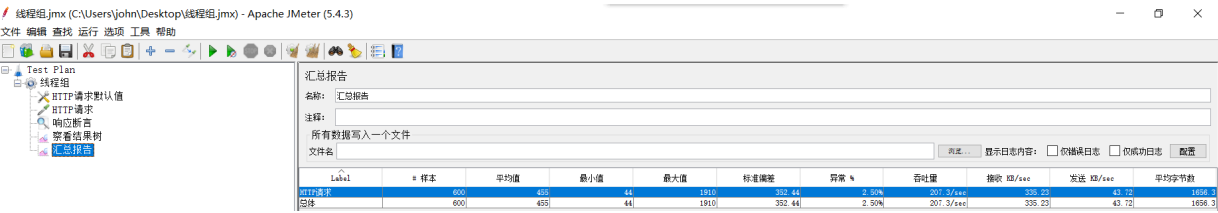
注释:

所有数据写入一个文件

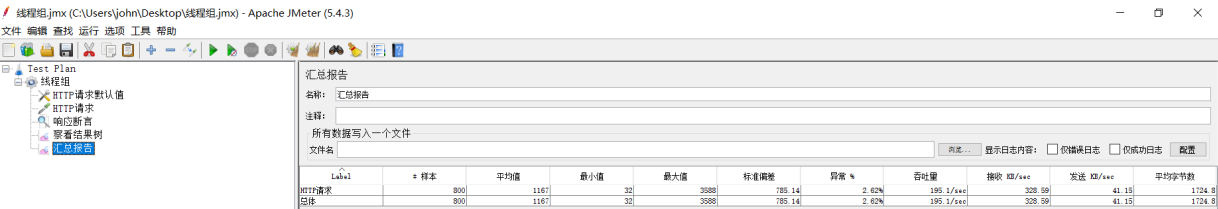
文件名: [浏览...] 显示日志内容: ☐ 仅错误日志 ☐ 仅成功日志 配置

Label	样本数	平均值	最小值	最大值	标准偏差	异常 %	吞吐量	接收 KB/sec	发送 KB/sec	平均字节数
HTTP请求	400	119	23	1012	245.84	0.00%	181.2/sec	50.62	38.23	286.0
等待	400	119	23	1012	245.84	0.00%	181.2/sec	50.62	38.23	286.0

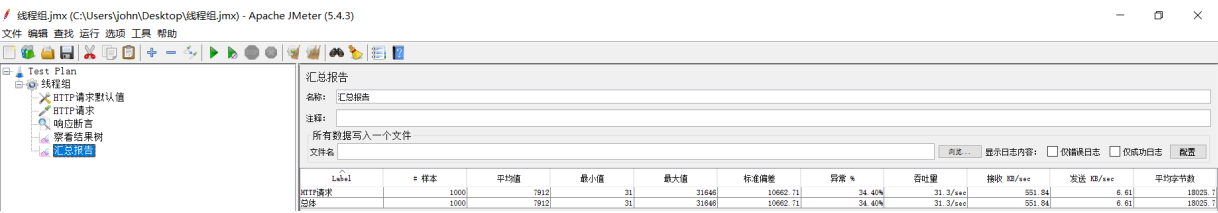
600线程：



800线程：



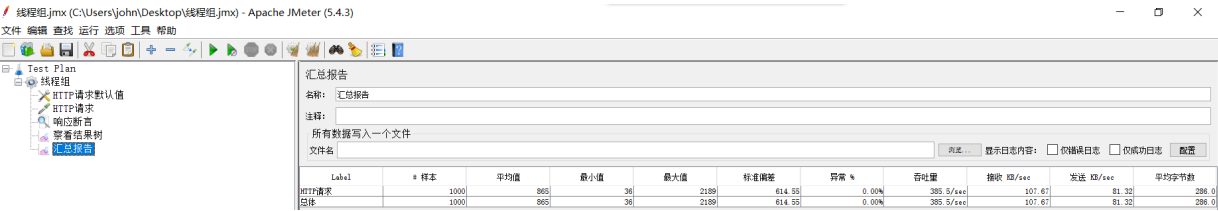
1000线程：



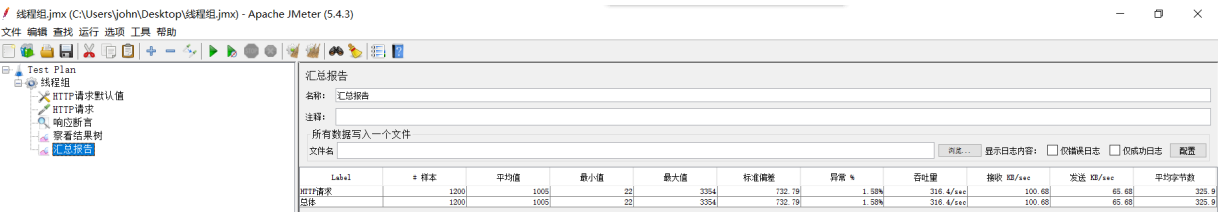
可以看出，从600线程开始，服务器的处理能力就开始下降了，开始出现丢包的情况，到1000线程时，服务器已经非常卡顿，最终有34%的异常

2server时结果如下：

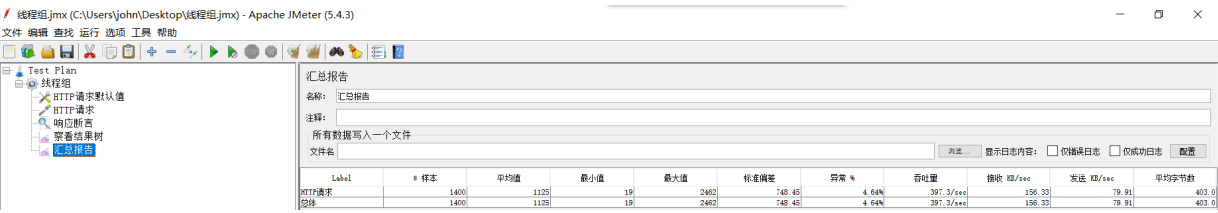
1000线程：



1200线程：



1400线程：



1600线程：

线程组.jmx (C:\Users\john\Desktop\线程组.jmx) - Apache JMeter (5.4.3)

文件 编辑 查找 运行 选项 工具 帮助

Test Plan
线程组
HTTP请求默认值
HTTP请求
响应断言
察看结果树
汇总报告

汇总报告
名称: 汇总报告
注释:
所有数据写入一个文件
文件名: 浏览... 显示日志内容: ☐ 仅错误日志 ☐ 仅成功日志 配置

Label	# 样本	平均值	最小值	最大值	标准偏差	异常 %	吞吐量	接收 KB/sec	发送 KB/sec	平均字节数
HTTP请求	1800	1185	14	3336	785.67	9.38%	382.9/sec	195.23	73.19	522.3
总计	1800	1185	14	3336	785.67	9.38%	382.9/sec	195.23	73.19	522.3

1800线程：

线程组.jmx (C:\Users\john\Desktop\线程组.jmx) - Apache JMeter (5.4.3)

文件 编辑 查找 运行 选项 工具 帮助

Test Plan
线程组
HTTP请求默认值
HTTP请求
响应断言
察看结果树
汇总报告

汇总报告
名称: 汇总报告
注释:
所有数据写入一个文件
文件名: 浏览... 显示日志内容: ☐ 仅错误日志 ☐ 仅成功日志 配置

Label	# 样本	平均值	最小值	最大值	标准偏差	异常 %	吞吐量	接收 KB/sec	发送 KB/sec	平均字节数
HTTP请求	1800	1262	18	4903	830.97	15.00%	305.2/sec	197.85	54.72	663.9
总计	1800	1262	18	4903	830.97	15.00%	305.2/sec	197.85	54.72	663.9

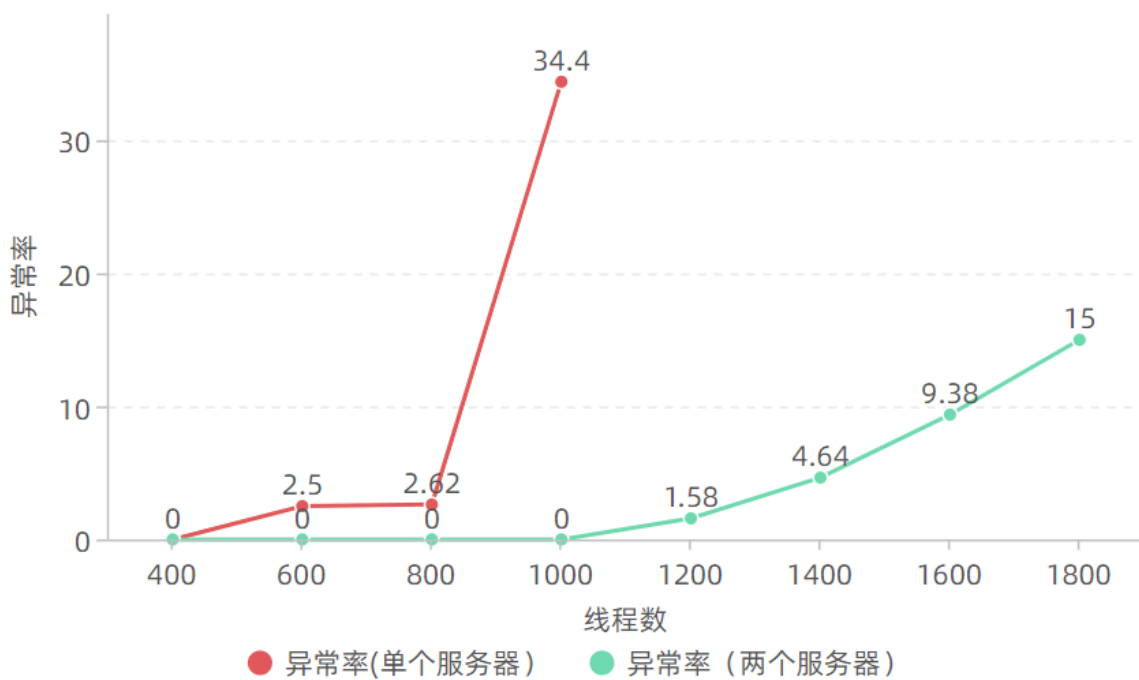
可以看出，使用分布式之后，仅仅只是两个server，已经可以轻松处理1000线程不报错了。

在1200线程的时候才开始出现丢包的情况，并且在1800线程的时候也仅有15%左右的异常，并且没有明显卡顿。可以看出分布式对于查询的优化效率是成倍的增长的

以下是根据异常率画的图：

压力测试

单位：%



数据来源：自测