

配置Jmeter压测计划

将Jmeter压测数据在grafana里展示

1、Docker 安装 InfluxDB

```
1 docker pull influxdb:1.8.6 # 拉取influxdb镜像
2 docker run -d -p 8086:8086 --name=jmeterdb influxdb:1.8.6 # 启动influxdb, 并命名为jmeterdb
3 docker exec -it jmeterdb bash # 进入容器
4 influx # 进入influxdb数据库
5 create database jmeter; # 创建jmeter库
6 show databases; # 显示所有数据库, 显示jmeter库就创建成功
7 use jmeter; # 进入jmeter库
8 select * from jmeter; # 查询库里面的数据, 这时数据是空的正常
```

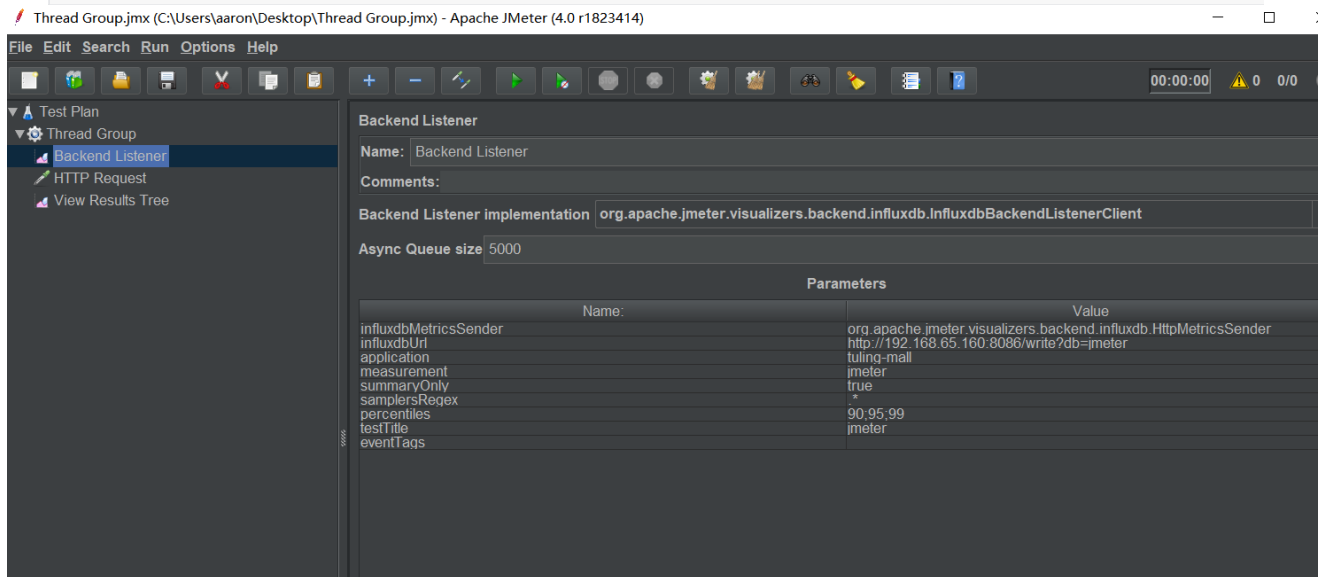
2、JMeter 配置 InfluxDB

在jmeter线程组下添加后端监听器

```
1 线程组 > 监听器 > 后端监听器
2 Thread Group > Listener > Backend Listener
```

配置参数

```
1 Backend Listener implementation:org.apache.jmeter.visualizers.backend.influxdb.InfluxdbBackendLi
stenerClient
2 influxdbUrl:http://192.168.65.160:8086/write?db=jmeter # 这里的IP输自己主机的
3 application:tuling-mall # 这里的名字自己随意定义即可
4 measurement:jmeter # 数据库的名字, jmeter为上面在influxdb中创建的jmeter库
5 testTitle:Jmeter # 这个名字也自己随意定义即可
```

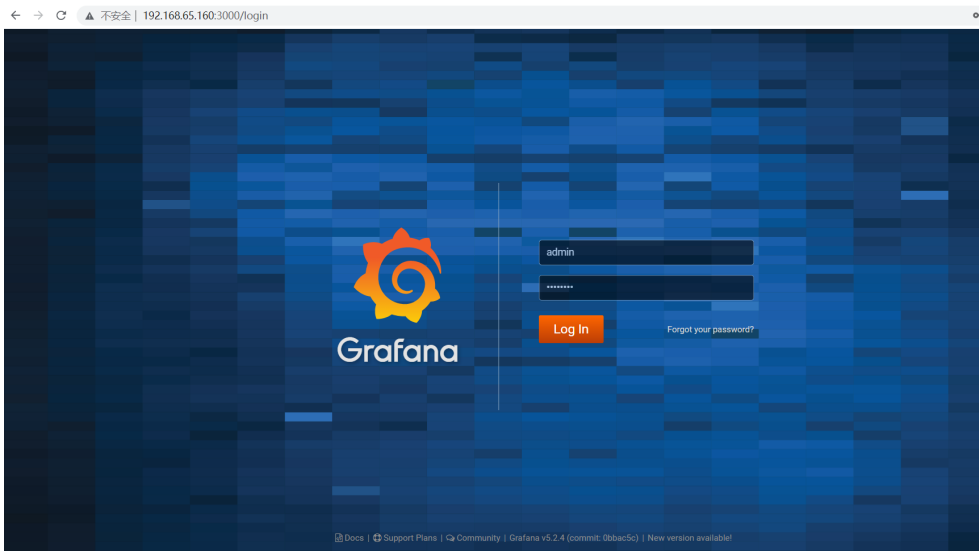


配置完之后执行一次压测脚本, 看influxdb中jmeter库里面有没有数据, 有数据就配置成功了。

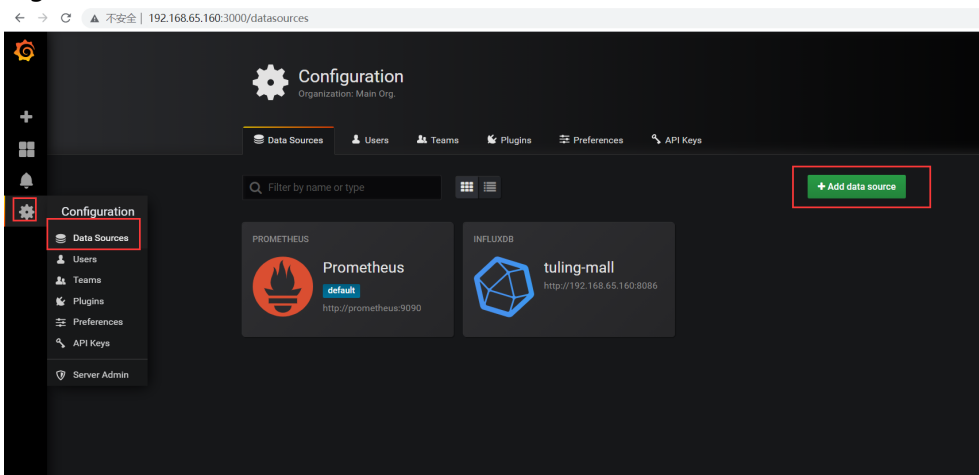
3、在Grafana中配置influxdb数据源

配置数据源

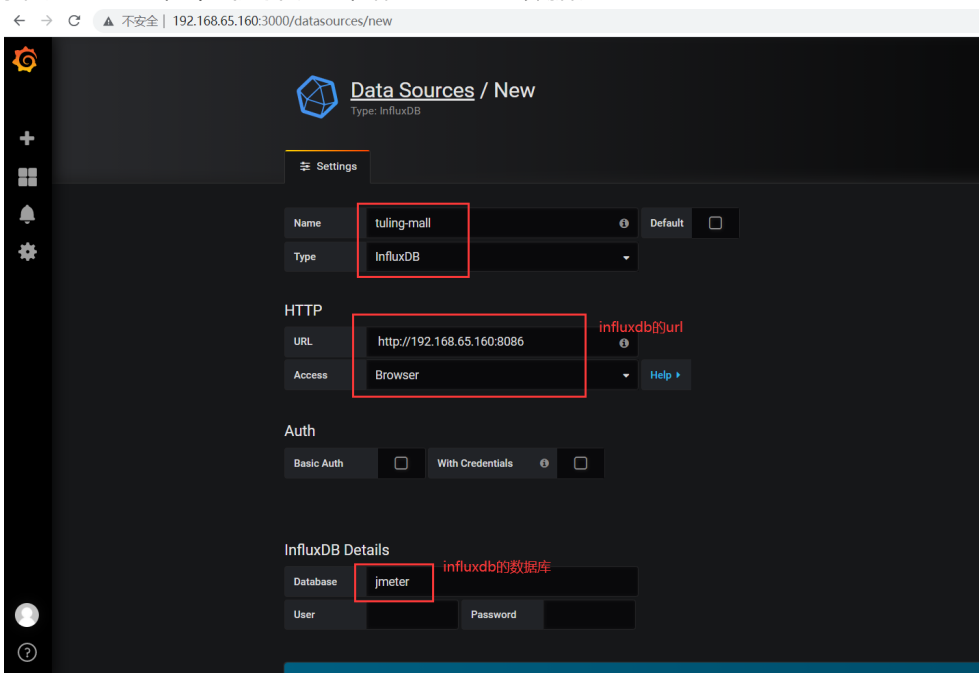
访问<http://192.168.65.160:3000/>, 进入登录页, 输入账号密码: admin/password



在grafana添加influxdb数据源，点击按钮Add data source

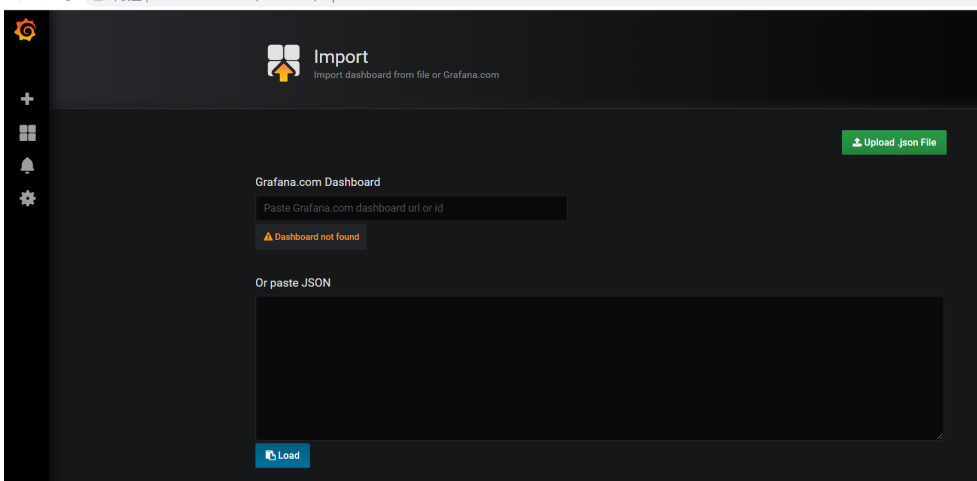
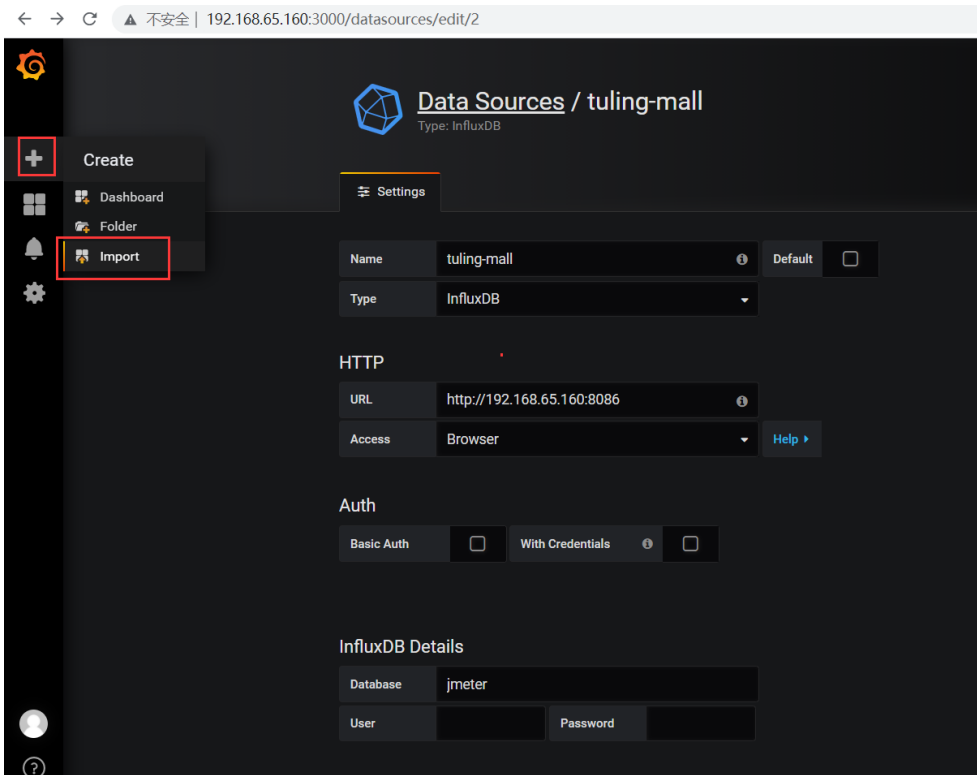


找到 influxdb，单击选择该db，配置influxdb数据源：



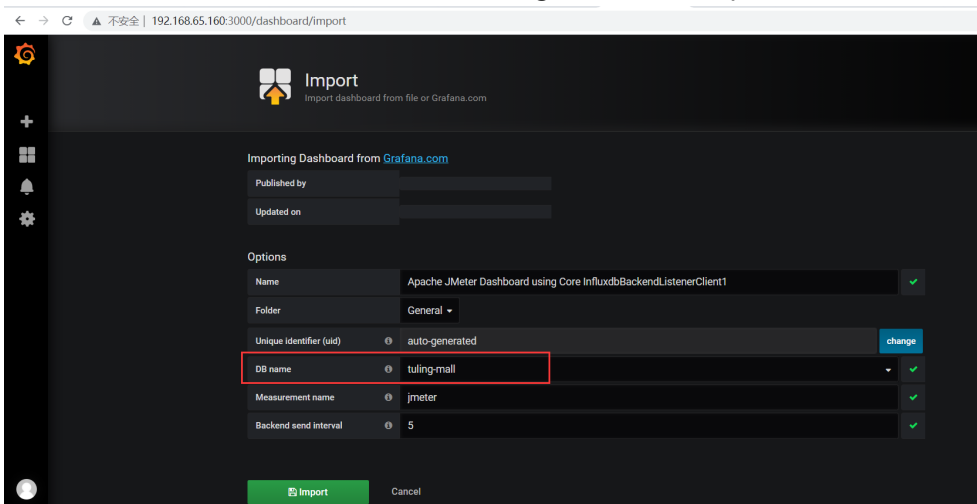
拉到页面最下面点击 Save&Test 按钮。

点击左侧加号，选择Import

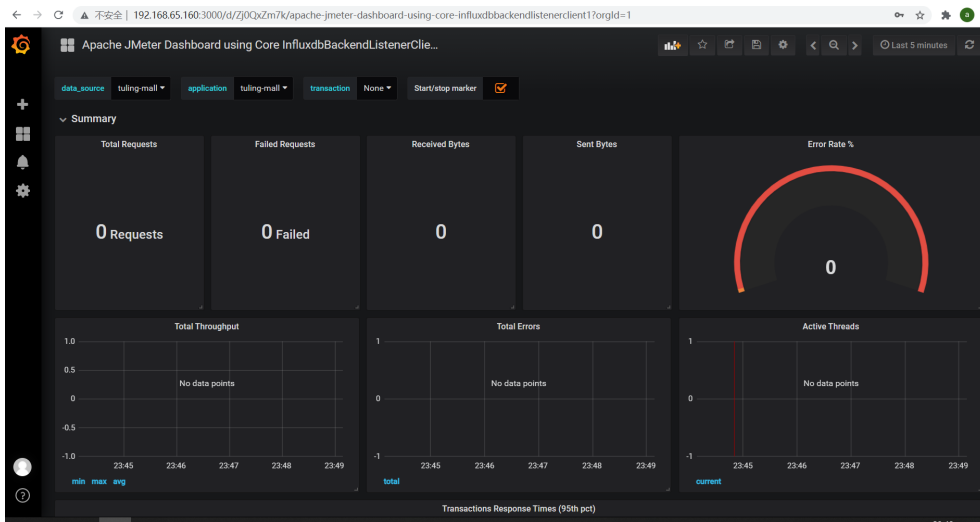


将json文本复制/粘贴到paste JSON 文本框中，单机Load按钮导入 (json文件下载地址：<https://grafana.com/api/dashboards/5496/revisions/1/download>)

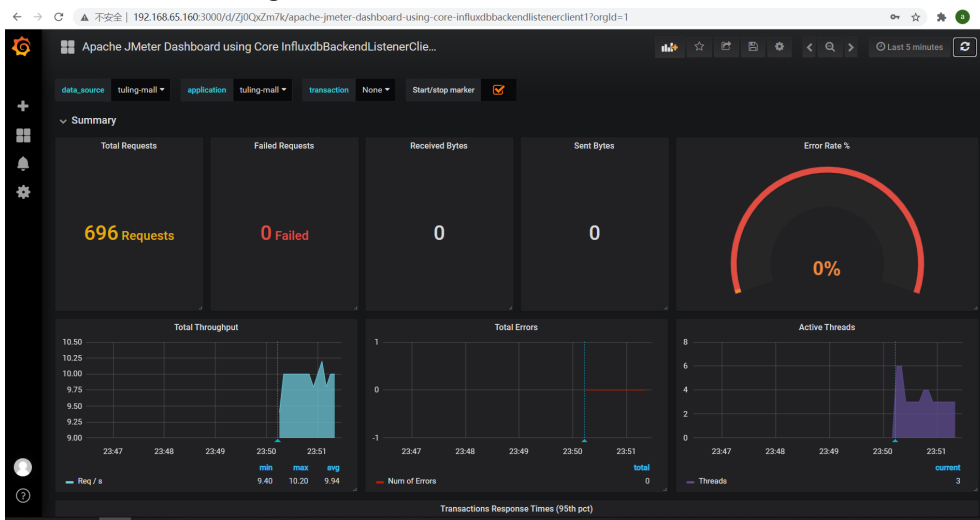
在DB name 中选择我们上面创建的数据源tuling-mall，单机 Import 按钮完成 Dashboard 导入



自动跳转至监控页面



JMeter脚本跑起来看下grafana数据!

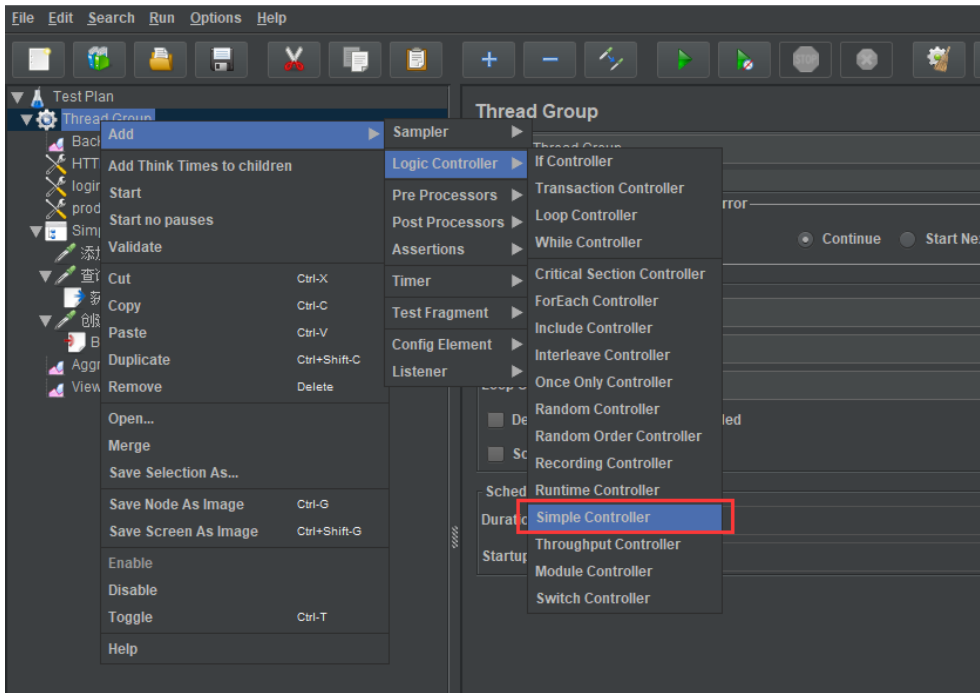


这样我们在用jmeter压测时就能通过grafana来看实时的压测数据了!

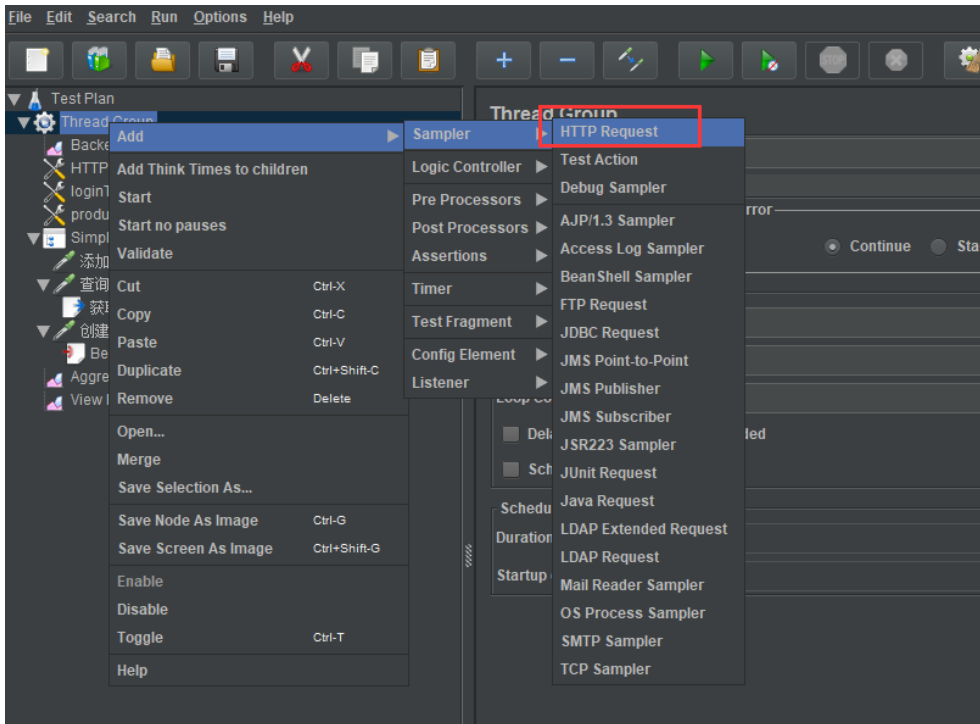
配置电商项目Jmeter压测计划(详细请参考视频)

1、创建压测请求

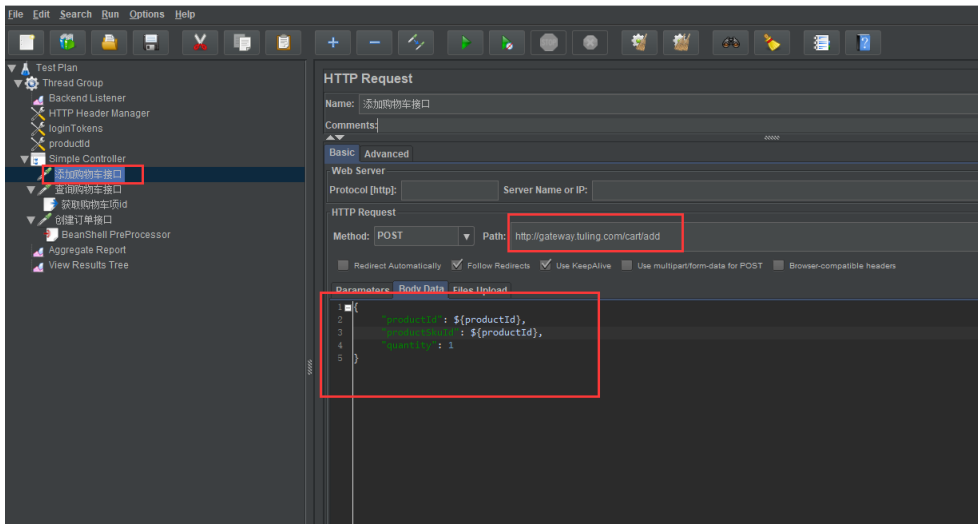
在上面新建的线程组下面新增Simple Controller, 把我们要压测的接口全部放这下面



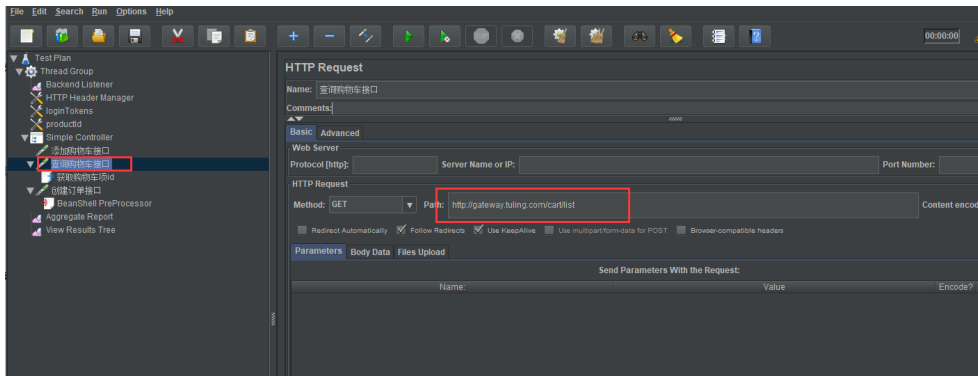
在Simple Controller下面添加我们要压测的接口(添加购物车接口, 查询购物车接口, 创建订单接口)



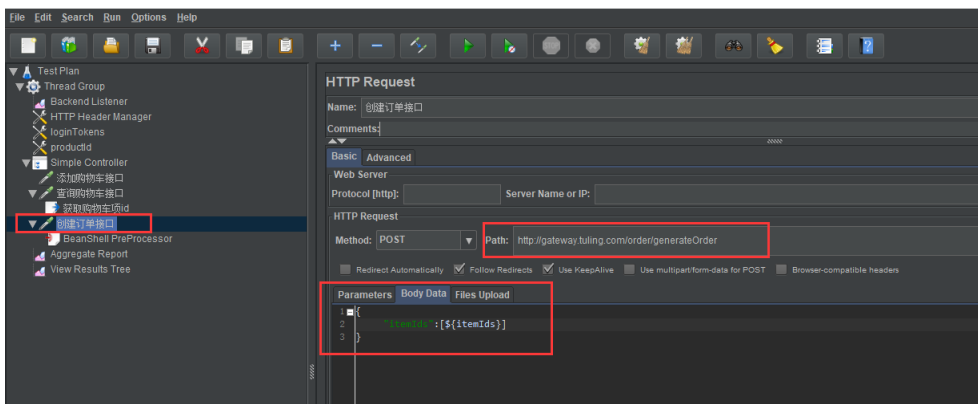
添加购物车接口配置如下图所示, $\${productId}$ 这是压测接口参数, 会从csv数据文件里取, 后面会详细说:



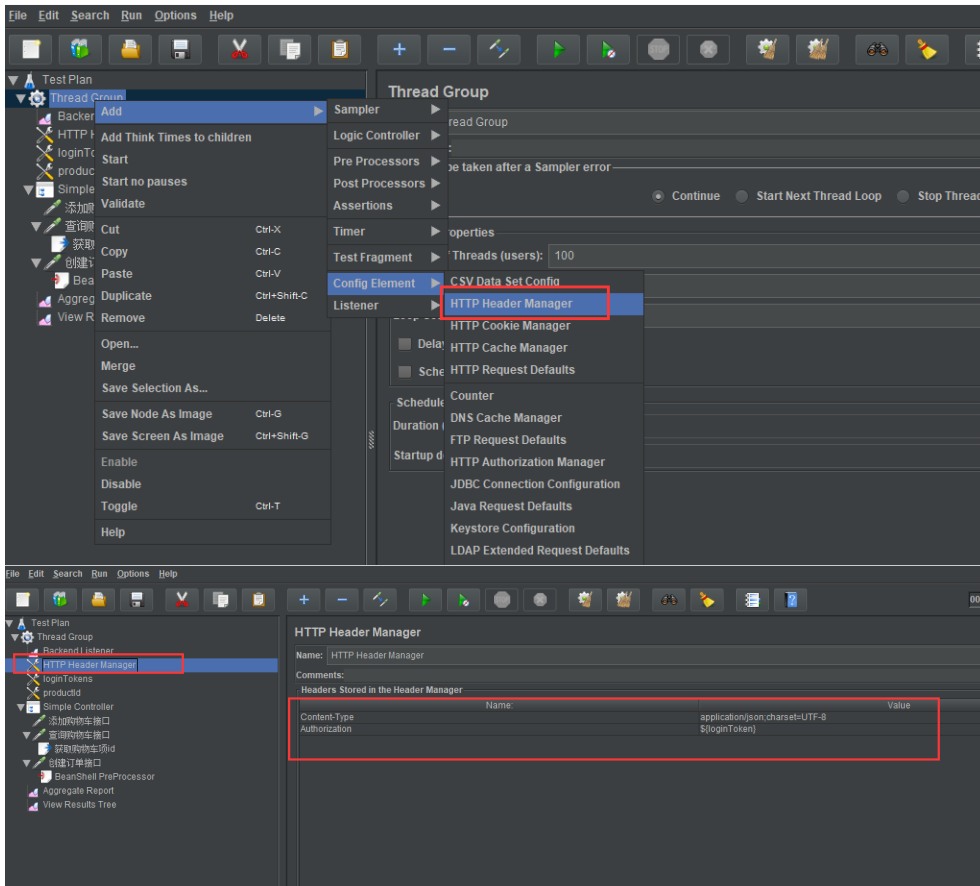
查询购物车接口配置如下图所示



创建订单接口配置如下图所示，\${itemIds}这是压测接口参数，后面会详细说如何取值：



配置所有接口的请求header，添加http header manager，\${loginToken}会从csv数据文件里取，后面会详细说：



我们来准备下压测数据\${productId}和\${loginToken}:

我们在D盘根目录创建一个txt的记事本文件，在里面放入1000个不同的商品id

product.txt - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

```
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
```

然后写一个简单的test程序用1000个不同的用户登录生成1000个登录成功的token，并且放入D盘根目录的loginTokens.txt的记事本文件里

```
1 package com.tuling.tulingmall;
2
3 import org.apache.http.HttpEntity;
4 import org.apache.http.HttpResponse;
5 import org.apache.http.NameValuePair;
6 import org.apache.http.client.HttpClient;
7 import org.apache.http.client.entity.UrlEncodedFormEntity;
8 import org.apache.http.client.methods.HttpPost;
9 import org.apache.http.impl.client.DefaultHttpClient;
10 import org.apache.http.message.BasicNameValuePair;
11 import org.apache.http.util.EntityUtils;
12 import org.json.JSONException;
13 import org.json.JSONObject;
14 import org.junit.Test;
```

```

15 import org.junit.runner.RunWith;
16 import org.springframework.boot.test.context.SpringBootTest;
17 import org.springframework.test.context.junit4.SpringRunner;
18
19 import java.io.BufferedWriter;
20 import java.io.FileOutputStream;
21 import java.io.IOException;
22 import java.io.OutputStreamWriter;
23 import java.util.ArrayList;
24 import java.util.List;
25
26 @RunWith(SpringRunner.class)
27 @SpringBootTest
28 public class GenLoginTokensTest {
29
30     @Test
31     public void testGenLoginTokens() throws IOException, JSONException {
32         for (int i = 10; i <= 1000; i++) {
33             login(i);
34         }
35     }
36
37     private void login(int userId) throws IOException, JSONException {
38         List<NameValuePair> formparams = new ArrayList<NameValuePair>();
39         formparams.add(new BasicNameValuePair("username", "zhuge" + userId));
40         formparams.add(new BasicNameValuePair("password", "123456"));
41         HttpEntity reqEntity = new UrlEncodedFormEntity(formparams, "utf-8");
42
43         HttpClient client = new DefaultHttpClient();
44         HttpPost post = new HttpPost("http://gateway.tuling.com/sso/login");
45         post.setEntity(reqEntity);
46         HttpResponse response = client.execute(post);
47
48         if (response.getStatusLine().getStatusCode() == 200) {
49             HttpEntity resEntity = response.getEntity();
50             String message = EntityUtils.toString(resEntity, "utf-8");
51             JSONObject jsonObj = new JSONObject(message);
52             String data = jsonObj.get("data").toString();
53             JSONObject jsonObj2 = new JSONObject(data);
54             //System.out.println(jsonObj2.get("token"));
55             writeFile("d:/loginTokens.txt", "bearer " + jsonObj2.get("token").toString());
56         } else {
57             System.out.println("请求失败");
58         }
59     }
60
61     public static synchronized void writeFile(String file, String content) throws IOException {
62         BufferedWriter out = null;
63         try {
64             out = new BufferedWriter(new OutputStreamWriter(
65                 new FileOutputStream(file, true)));

```

```

66 out.write(conent + "\r\n");
67 } catch (Exception e) {
68 e.printStackTrace();
69 } finally {
70 out.close();
71 }
72 }
73 }

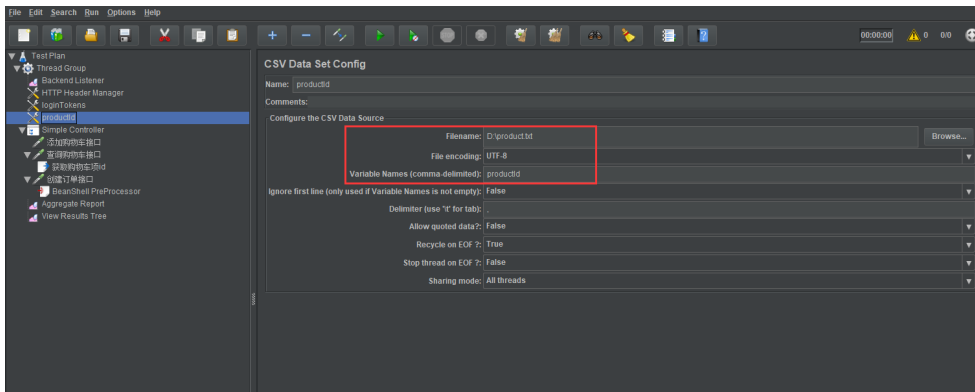
```



在jmeter里添加csv data数据供接口参数使用

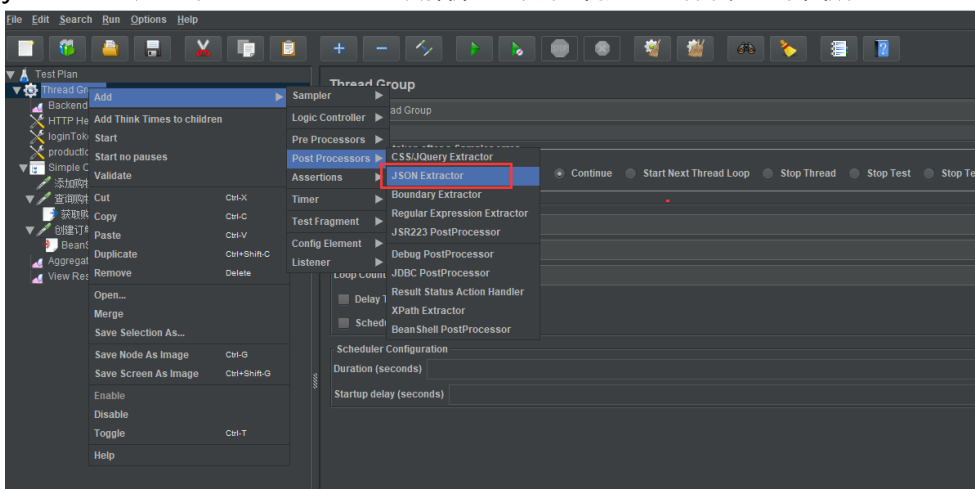
The screenshot shows the JMeter interface with the 'CSV Data Set Config' element selected in the 'Config Element' menu. The configuration details are as follows:

- Name:** loginTokens
- Comments:**
- Configure the CSV Data Source:**
 - Filename: D:\loginTokens.txt
 - File encoding: UTF-8
 - Variable Names (comma delimited): loginToken
 - Ignore first line (only used if Variable Names is not empty): False
 - Delimiter (use "\t" for tabs):
 - Allow quoted data?: False
 - Recycle on EOF?: True
 - Stop thread on EOF?: False
 - Sharing mode: All threads

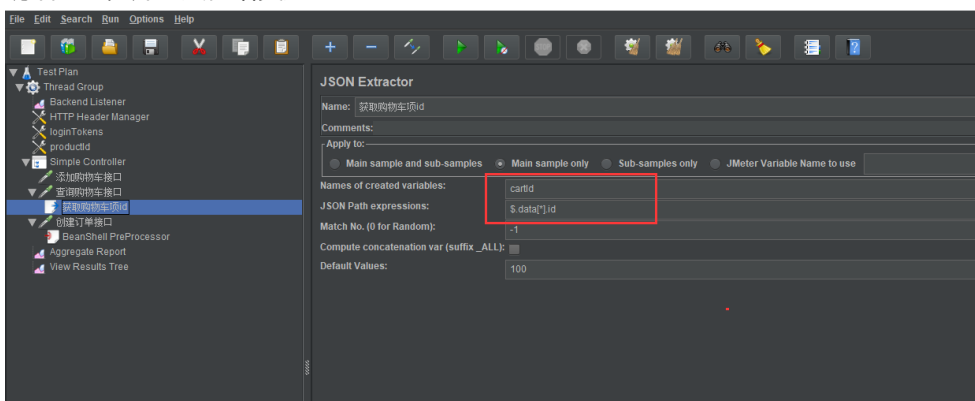


这样配置完成后，这几个接口每次用一次线程压测时就会从loginTokens.txt和product.txt文件里分别那一条数据放入上面的几个接口参数里

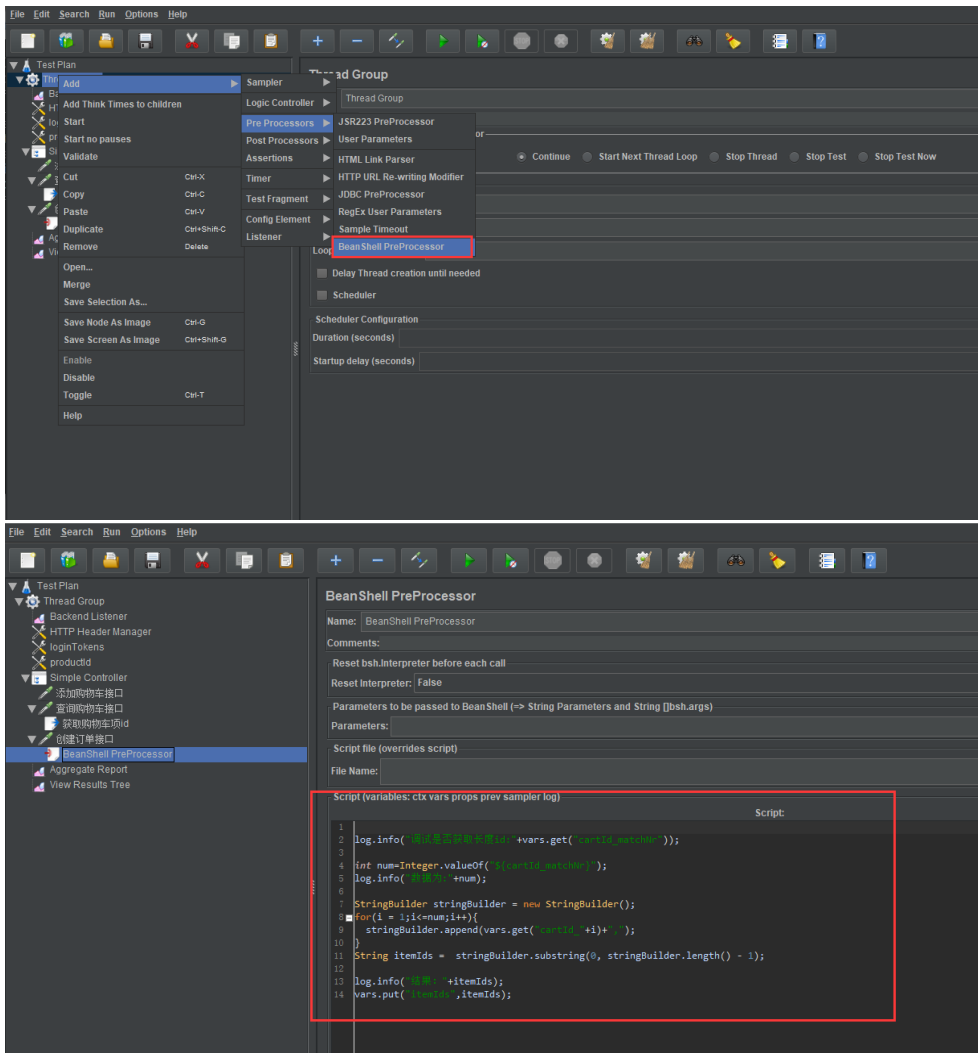
因为创建订单接口依赖到查询购物车接口，需要从查询购物车接口的结果里拿出购物车明细id，所以我们先在jmeter里创建一个JSON Extractor来解析查询购物车接口的结果，如下图所示：



这个JSON Extractor需要放在查询购物车接口的下面，下面的配置意思就是从查询购物车接口结果里解析出购物车明细的id，并且赋值给变量cartId



然后在创建订单接口的下面添加一个BeanShell PreProcessor，这个处理器会在创建订单接口调用之前执行，我们需要在这个处理器内部写一点解析参数的代码，对之前的cartId变量做处理，最终传入创建订单接口的itemIds参数里去，如下图所示：



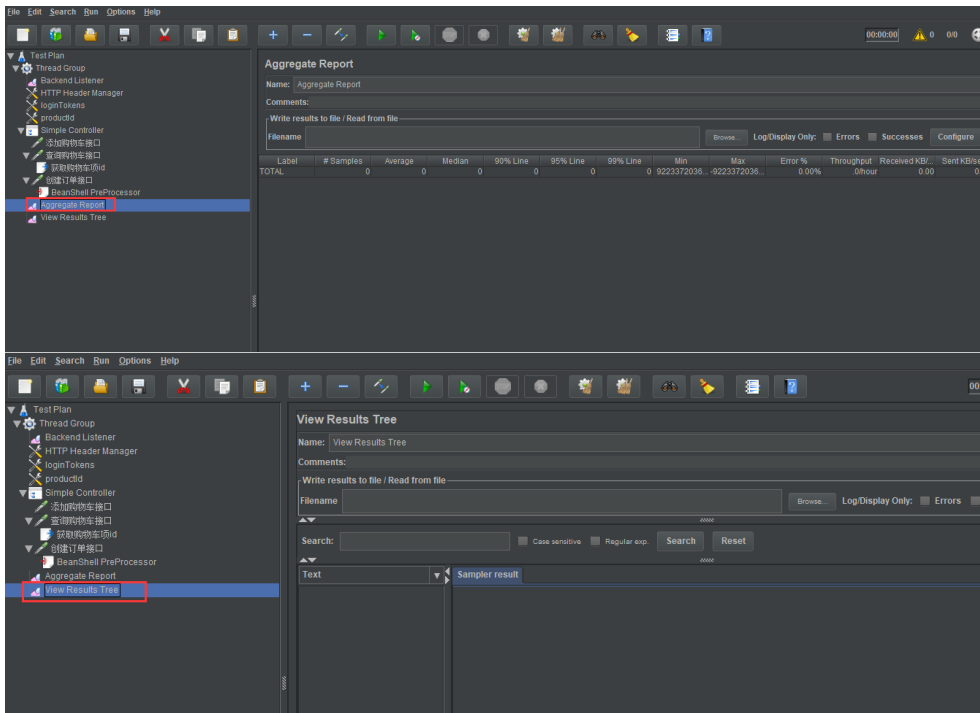
上图详细代码如下：

```

1  log.info("调试是否获取长度id:"+vars.get("cartId_matchNr"));
2
3  int num=Integer.valueOf("${cartId_matchNr}");
4  log.info("数据为:"+num);
5
6  StringBuilder stringBuilder = new StringBuilder();
7  for(i = 1;i<=num;i++){
8    stringBuilder.append(vars.get("cartId_"+i)+",");
9  }
10 String itemIds = stringBuilder.substring(0, stringBuilder.length() - 1);
11
12 log.info("结果: "+itemIds);
13 vars.put("itemIds",itemIds);

```

上面这些配置做完就可以开始压测了，只需在线程组上配置你自己想压测的线程数就可以了，如果想看压测结果可以打开压测结果报告Aggregate Report和View Results Tree查看压测结果情况



压测环境说明

总共4台高配服务器，配置见下图，云盘容量显示不准确，实际每台机器都是100G以上：

192.168.65.160：部署了K8S的master节点

192.168.65.203：部署了K8S的node1节点，里面部署了order和product服务

192.168.65.210：部署了K8S的node2节点，里面部署了authcenter、gateway和member服务

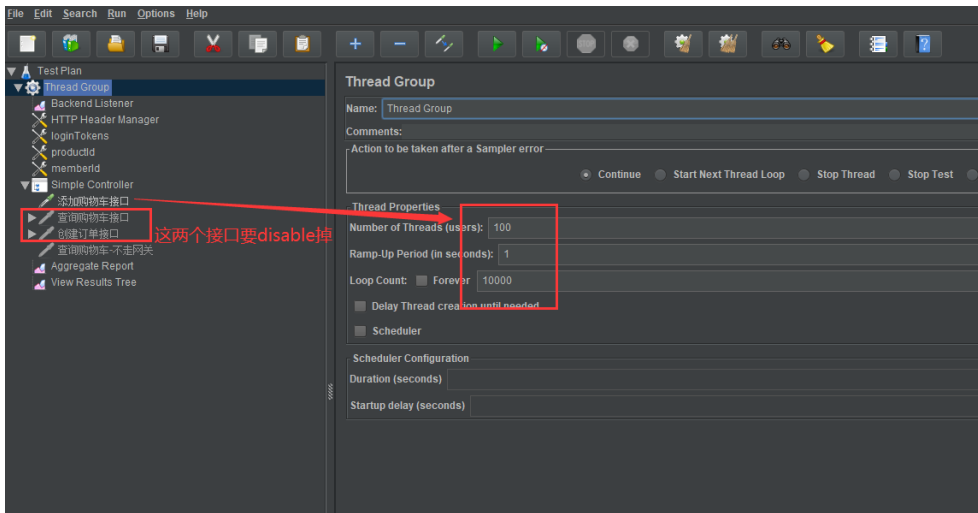
192.168.65.42：用docker部署了电商项目所有依赖的中间件和数据库

名称 ^	标签(管理员)	CPU	内存	云盘真实容量	操作系统	默认IP	默认MAC地址	启用状态(全部)
K8S集群-诸葛老师-1	勿动-授课机	8	16 GB	5.67 GB	Linux	192.168.65.160	fa:69:0f:b6:94:00	运行中
K8S集群-诸葛老师-2	勿动-授课机	8	12 GB	47.21 GB	Linux	192.168.65.203	fa:7b:42:43:1d:...	运行中
K8S集群-诸葛老师-3	勿动-授课机	8	12 GB	47.84 GB	Linux	192.168.65.210	fa:8d:81:19:03:...	运行中
K8S集群-诸葛老师-4	勿动-授课机	12	24 GB	19.23 GB	Linux	192.168.65.42	fa:24:27:62:2a:00	运行中

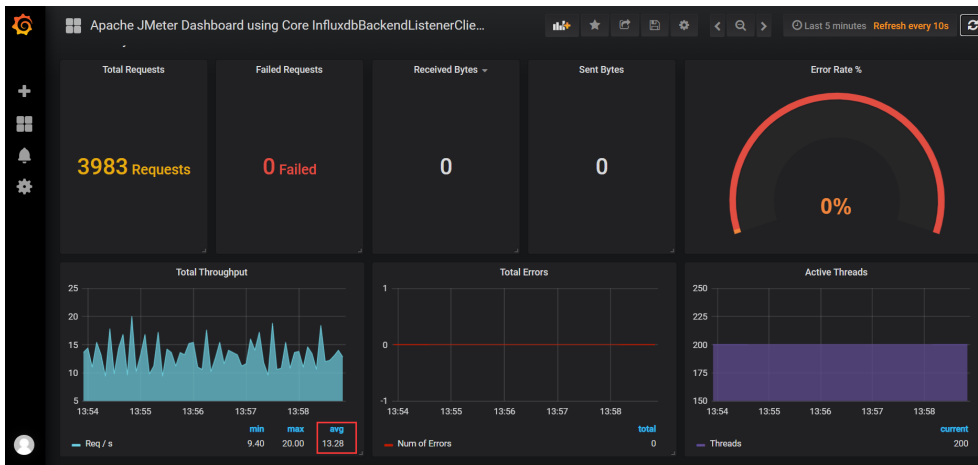
```
[root@k8s-master tuling-mall]# kubectl get all -o wide
NAME                 READY   STATUS    RESTARTS   AGE   IP              NODE             NOMINATED NODE   READINESS GATES
pod/tulingmall-authcenter-deployment-7f5f4cfb6f-5wp9m        1/1     Running   0           39m   10.244.169.186   k8s-node2       <none>            <none>
pod/tulingmall-gateway-deployment-74566bf964-cn4s2          1/1     Running   0           93m   10.244.169.181   k8s-node2       <none>            <none>
pod/tulingmall-member-deployment-84c7595c76-vn6rc           1/1     Running   0           24m   10.244.169.165   k8s-node2       <none>            <none>
pod/tulingmall-order-deployment-99c449867-m25vj             1/1     Running   0           94m   10.244.36.72    k8s-node1       <none>            <none>
pod/tulingmall-product-deployment-69f579996-pzn8j           1/1     Running   0           94m   10.244.36.69    k8s-node1       <none>            <none>
```

用Jmeter开始压测

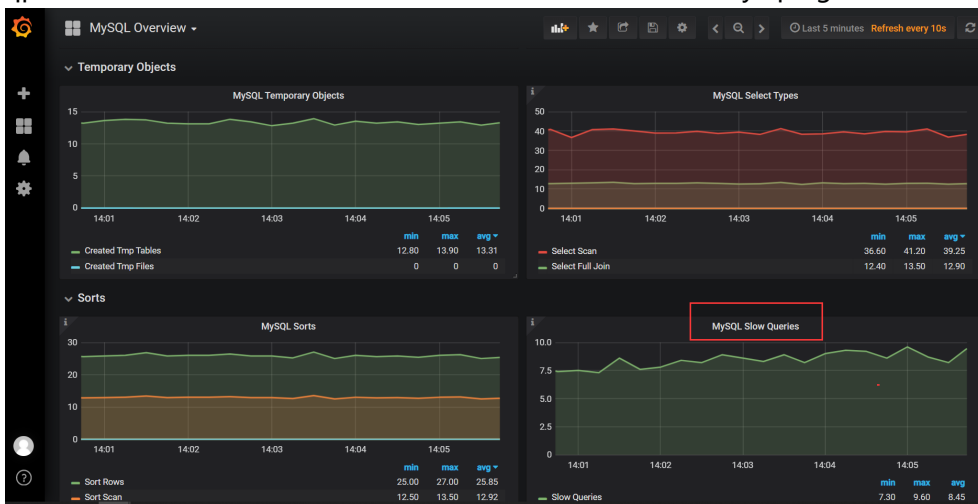
压测添加购物车接口



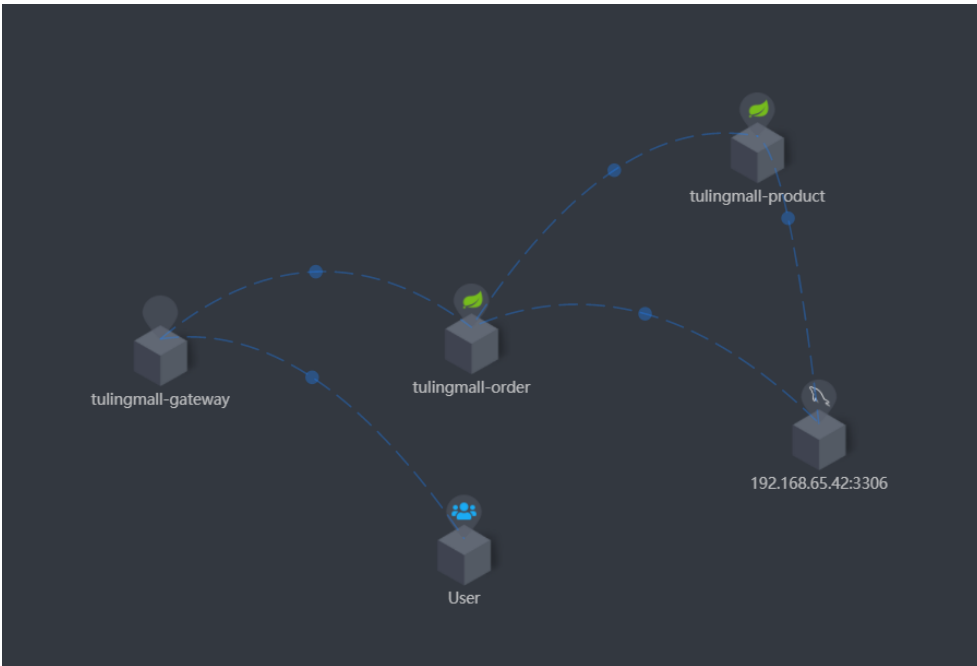
看下grafana的jmeter压测数据:



qps只有13, 这个肯定是太低了, 我们需要定位原因, 我们打开mysql的grafana监控图, 发现有不少慢查询



我们再看下skywalking观察下整个调用链路是慢在哪一步, 可以看到加购车的调用链路如下



再看下skywalking的加购物车接口的调用整个链路时长：

Method	Start Time	Exec(ms)	Exec(%)	Self(ms)	API	Service
/cart/add	2021-07-20 14:07:31	37238		2	spring-webflux	tulingmall-gateway
SpringCloudGateway/RoutingFilter	2021-07-20 14:07:31	37236		0	spring-cloud-gat...	tulingmall-gateway
(POST)/cart/add	2021-07-20 14:07:31	37234		2	spring-cloud-gat...	tulingmall-gateway
/cart/add	2021-07-20 14:01:09	35944		35555	SpringMVC	tulingmall-order
SpringCloudGateway/RoutingFilter	2021-07-20 14:01:09	35943		0	spring-cloud-gat...	tulingmall-order
(POST)/cart/add	2021-07-20 14:01:09	35942		0	Feign	tulingmall-order
/pms/cartProduct/1102	2021-07-20 14:01:09	1621		0	Feign	tulingmall-product
(GET)/pms/cartProduct/productId	2021-07-20 14:07:45	1621		1621	SpringMVC	tulingmall-product
MySQL/JDBI/PreparedStatement/execute	2021-07-20 14:07:45	1621		4	mysql-connector...	tulingmall-order
MySQL/JDBI/PreparedStatement/execute	2021-07-20 14:07:47	4		2	mysql-connector...	tulingmall-order
MySQL/JDBI/PreparedStatement/execute	2021-07-20 14:07:47	7		7	mysql-connector...	tulingmall-order

我们发现有一个post请求竟然执行30多秒，还有一个mysql操作也执行了近2秒，我们现在看下这个mysql查询是什么

商品信息

服务: tulingmall-product
 端点: MySQL/JDBI/PreparedStatement/execute
 健康类型: Exit
 组件: mysql-connector-java
 Peer: 192.168.65.42:3306
 失败: false
 db.type: sql
 db.instance: micromall

db.statement: `SELECT p.id id, p.`name` name, p.sub_title subTitle, p.price price, p.pic pic, p.product_attribute_category_id productAttributeCategoryId, p.stock stock, pa.id attr_id, pa.`name` attr_name, pa.type attr_type, ps.id sku_id, ps.sku_code sku_code, ps.price sku_price, ps.sp1 sku_sp1, ps.sp2 sku_sp2, ps.sp3 sku_sp3, ps.stock sku_stock, ps.pic sku_pic FROM pms_product p LEFT JOIN pms_product_attribute pa ON p.product_attribute_category_id = pa.product_attribute_category_id LEFT JOIN pms_sku_stock ps ON pa.id=ps-product_id WHERE p.id = ? AND pa.type = 0 ORDER BY pa.sort desc`

拉出sql看下执行计划是否合理的走了索引

```

1 EXPLAIN SELECT
2 p.id id,
3 p.`name` NAME,
4 p.sub_title subTitle,
5 p.price price,
6 p.pic pic,
7 p.product_attribute_category_id productAttributeCategoryId,
8 p.stock stock,
9 pa.id attr_id,
10 pa.`name` attr_name,
11 pa.type attr_type,
12 ps.id sku_id,
13 ps.sku_code sku_code,
14 ps.price sku_price,
15 ps.sp1 sku_sp1,
16 ps.sp2 sku_sp2,
17 ps.sp3 sku_sp3,
18 ps.stock sku_stock,
19 ps.pic sku_pic
20 FROM
21 pms_product p
22 LEFT JOIN pms_product_attribute pa ON p.product_attribute_category_id = pa.product_attribute_category_id
23 LEFT JOIN pms_sku_stock ps ON p.id = ps.product_id
24 WHERE
25 p.id = 200
26 AND pa.type = 0
27 ORDER BY
28 pa.sort DESC;

```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	p	(Null)	const	PRIMARY	PRIMARY	8	const	1	100	Using temporary; Using filesort
1	SIMPLE	pa	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	28	3.57	Using where
1	SIMPLE	ps	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	990884	100	Using where; Using join buffer (Block Nested Loop)

我们发现有两个查询没走索引，pa表因为是商品属性表，本身数据量很少，关系不大，但是ps表因为是商品库存表，数据量较大，我们看到扫描了近百万行数据，这必须得加索引优化了，通过sql分析，我们决定给pms_sku_stock表的product_id字段加上索引，加完索引我们再看下查询计划

```

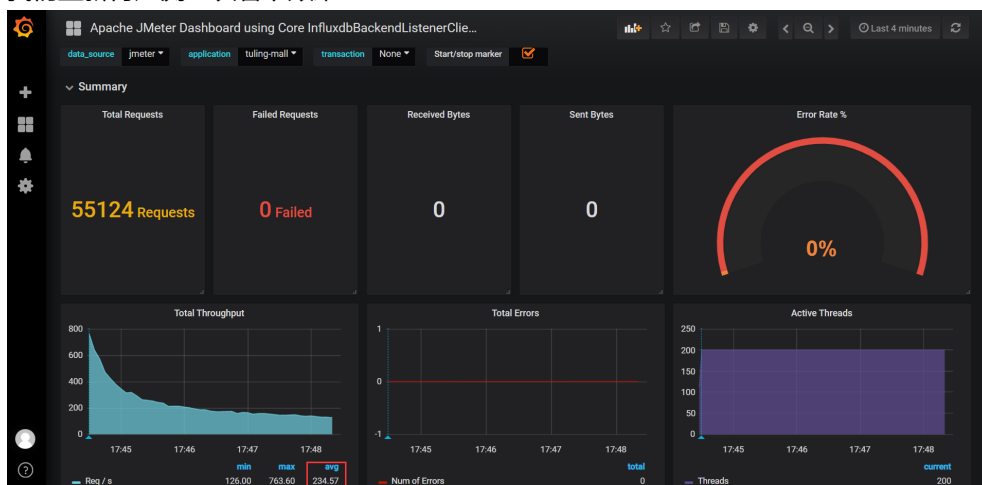
1 EXPLAIN SELECT
2 p.id id,
3 p.`name` NAME,
4 p.sub_title subTitle,
5 p.price price,
6 p.pic pic,
7 p.product_attribute_category_id productAttributeCategoryId,
8 p.stock stock,
9 pa.id attr_id,
10 pa.`name` attr_name,
11 pa.type attr_type,
12 ps.id sku_id,
13 ps.sku_code sku_code,
14 ps.price sku_price,
15 ps.sp1 sku_sp1,
16 ps.sp2 sku_sp2,
17 ps.sp3 sku_sp3,
18 ps.stock sku_stock,
19 ps.pic sku_pic
20 FROM
21 pms_product p
22 LEFT JOIN pms_product_attribute pa ON p.product_attribute_category_id = pa.product_attribute_category_id
23 LEFT JOIN pms_sku_stock ps ON p.id = ps.product_id
24 WHERE
25 p.id = 200
26 AND pa.type = 0
27 ORDER BY
28 pa.sort DESC;

```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	p	(Null)	const	PRIMARY	PRIMARY	8	const	1	100	Using filesort
1	SIMPLE	pa	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	28	3.57	Using where
1	SIMPLE	ps	(Null)	ref	idx_product_id	idx_product_id	9	const	1	100	(Null)

优化效果很明显。

我们重新再压测一次看下效果



qps超过了两百多，最高的时候有四五百多，好像优化有效果，继续一直压测下去，会发现qps一直在慢慢往下掉，这是为什么了？

我们看了下mysql机器的cpu，发现已经严重超载了，肯定是有sql执行比较慢了

```

top - 12:53:49 up 8 days, 17:37, 1 user, load average: 4.30, 1.48, 1.93
Tasks: 332 total, 1 running, 331 sleeping, 0 stopped, 0 zombie
%Cpu(s): 99.3 us, 0.3 sy, 0.0 ni, 0.2 id, 0.0 wa, 0.0 hi, 0.3 si, 0.0 st
KiB Mem : 24521464 total, 4947028 free, 14461876 used, 5112560 buff/cache
KiB Swap: 12386300 total, 12382460 free, 3840 used, 9651040 avail Mem

  PID USER      PR  NI  VIRT  RES  SHR  S  %CPU  %MEM    TIME+  COMMAND
16421 ods      20   0 6507520 589560 13136 S  1189  2.4   4544:38  mysqld
17867 root      20   0  12.0g  5.6g 14960 S   2.7  23.8  175:37.41  java
16079 1000     20   0 8040440 927120 14836 S   1.3   3.8   47:56.79  java
16973 root      20   0 8994280 943424 13652 S   1.3   3.8   58:29.03  java
16976 1000     20   0 1352436 196696 13556 S   1.0   0.8   27:09.20  node
30874 nfsnobo+ 20   0 720444 30604 6004 S   1.0   0.1   44:47.81  node_exporter
16414 ods      20   0 278264 72388 8924 S   0.7   0.3   39:15.64  mongod
18226 ods      20   0 7503436 105388 4984 S   0.7   0.4   47:04.25  beam.smp
29733 root      20   0 0 0 0 S   0.7   0.0   0:00.29  kworker/0:0
825 root      20  -20 0 0 0 S   0.3   0.0   0:51.18  kworker/11:1H
885 root      20   0 21684 1352 984 S   0.3   0.0   1:09.79  imbalance

```

再看下skywalking, 之前那条sql的执行时间已经到了几毫秒了

Method	Start Time	Exec(ms)	Exec(%)	Self(ms)	API	Service
/cart/add	2021-07-20 14:44:56	1951		1	spring-webflux	tulingmall-gateway
SpringCloudGateway/RoutingFilter	2021-07-20 14:44:56	1950		0	spring-cloud-gat...	tulingmall-gateway
SpringCloudGateway/sendRequest	2021-07-20 14:44:56	1950		0	spring-cloud-gat...	tulingmall-gateway
(POST)/cart/add	2021-07-20 14:44:56	1950		1651	SpringMVC	tulingmall-order
MySQL/DB/PreparedStatement/execute	2021-07-20 14:44:58	192		192	mysql-connector...	tulingmall-order
Balance/pms/cartProduct/productId	2021-07-20 14:44:58	3		0	Feign	tulingmall-order
/pms/cartProduct/1649	2021-07-20 14:44:58	3		0	Feign	tulingmall-order
(GET)/pms/cartProduct/productId	2021-07-20 14:44:58	3		2	SpringMVC	tulingmall-product
MySQL/DB/PreparedStatement/ex...	2021-07-20 14:44:58	1		1	mysql-connector...	tulingmall-product
MySQL/DB/PreparedStatement/execute	2021-07-20 14:44:58	8		8	mysql-connector...	tulingmall-order
MySQL/DB/PreparedStatement/execute	2021-07-20 14:44:58	2		2	mysql-connector...	tulingmall-order
MySQL/DB/Connection/commit	2021-07-20 14:44:58	8		8	mysql-connector...	tulingmall-order

不过好像又多了一条mysql操作有一百多毫秒, 我们拉出sql看下查询计划

跨度信息

服务: tulingmall-order
 端点: MySQL/DB/PreparedStatement/execute
 跨度类型: Exit
 组件: mysql-connector-java
 Peer: 192.168.65.42:3306
 失败: false
 db.type: sql
 db.instance: micromall
 db.statement: `select id, product_id, product_sku_id, member_id, quantity, price, sp1, sp2, sp3, product_pk, product_name, product_sub_title, product_sku_code, member_nickname, create_date, modify_date, delete_status, product_category_id, product_brand, product_sn, product_attr from oms_cart_item WHERE (member_id = ? and product_id = ? and delete_status = ?)`

```

1 EXPLAIN SELECT
2 id,
3 product_id,
4 product_sku_id,
5 member_id,
6 quantity,
7 price,
8 sp1,
9 sp2,
10 sp3,
11 product_pic,
12 product_name,
13 product_sub_title,
14 product_sku_code,
15 member_nickname,
16 create_date,
17 modify_date,
18 delete_status,
19 product_category_id,
20 product_brand,
21 product_sn,
22 product_attr
23 FROM
24 oms_cart_item
25 WHERE
26 (
27 member_id = 200
28 AND product_id = 200
29 AND delete_status = 0
30 )

```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	oms_cart_item	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	118358	0.1	Using where

这是一条查询购物车的sql，也没有走索引，之前这条sql执行较快可能是因为数据量不大，随着我们压测的进行，数据量越来越大，这条有问题的sql就暴露出来了，这会不会是上面那个qps下降的原因了？

我们试着优化下这条sql，优化完再重新压测下，我们给oms_cart_item表加上联合索引(product_id,member_id)，再看下查询计划

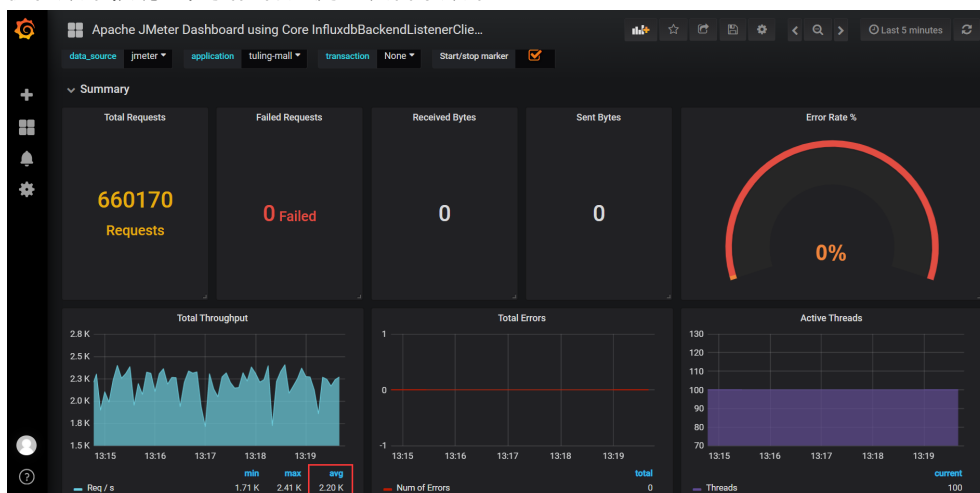
```

1 EXPLAIN SELECT
2 id,
3 product_id,
4 product_sku_id,
5 member_id,
6 quantity,
7 price,
8 sp1,
9 sp2,
10 sp3,
11 product_pic,
12 product_name,
13 product_sub_title,
14 product_sku_code,
15 member_nickname,
16 create_date,
17 modify_date,
18 delete_status,
19 product_category_id,
20 product_brand,
21 product_sn,
22 product_attr
23 FROM
24 oms_cart_item
25 WHERE
26 (
27 member_id = 200
28 AND product_id = 200
29 AND delete_status = 0
30 )

```

id	select_type	table	partitions	type	possible keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	oms_cart_item	(Null)	ref	idx_member_id_and_idx_member_id_and_product_id	const,const		const,const	1		10 Using where

优化效果很明显，我们再压测一次看下效果



```

top - 13:22:00 up 8 days, 18:05, 1 user, load average: 0.69, 1.16, 1.70
Tasks: 332 total, 1 running, 331 sleeping, 0 stopped, 0 zombie
%Cpu(s): 7.7 us, 4.4 sy, 0.0 ni, 84.4 id, 0.9 wa, 0.0 hi, 2.4 si, 0.1 st
KiB Mem : 24521464 total, 4931824 free, 14470304 used, 5119336 buff/cache
KiB Swap: 12386300 total, 12382460 free, 3840 used, 9642620 avail Mem

  PID USER      PR  NI  VIRT  RES  SHR  %CPU  %MEM    TIME+  COMMAND
 16421 ods      20   0 6507520 589568 13136 S 163.8  2.4   4593:45 mysqlld
 17867 root       20   0  12.0g  5.6g 14960 S   4.7 23.8   176:45.38 java
 16079 1000      20   0 8040440 927120 14836 S   2.3  3.8   48:16.37 java
 16414 ods      20   0 278264 72388  8924 S   1.3  0.3   39:30.73 mongod
 16976 1000      20   0 1346292 196576 13556 S   1.3  0.8   27:20.51 node
 18226 ods      20   0 7503436 105388 4984 S   1.3  0.4   47:19.77 beam.smp
 30874 nfsnobo+  20   0 720444 30604  6004 S   1.3  0.1   45:07.31 node_exporter
   343 root       20   0  70644 18516 1468 S   1.0  0.1   90:58.47 plymouthd
   825 root       0 -20    0     0     0 S   1.0  0.0    0:57.71 kworker/11:1H

```

qps稳定在两千多，mysql的cpu占用也下降了很多，优化效果相当明显，提升了一个数量级，可见平时工作中对数据库的sql优化是非常重要的，一条慢sql在高并发场景下甚至可以拖垮整个数据库

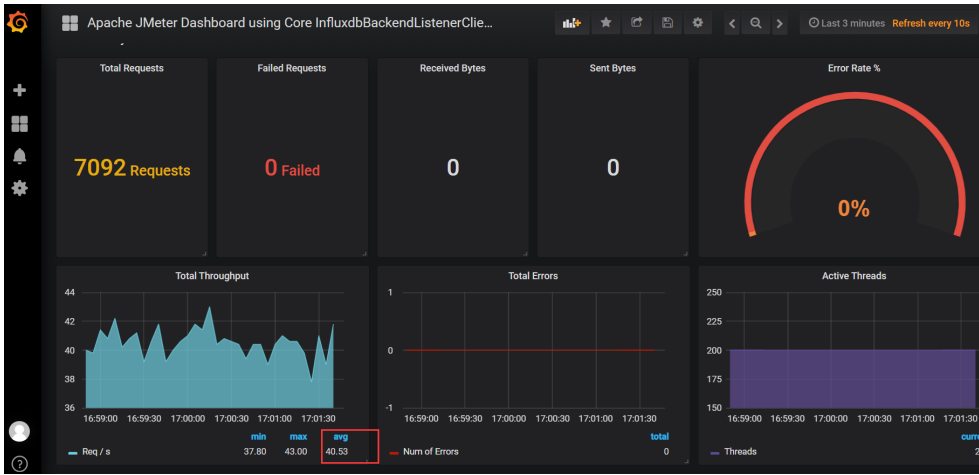
Method	Start Time	Exec(ms)	Exec(%)	Self(ms)	API	Service
/cart/add	2021-07-27 13:01:59	131	9	2	spring-webflux	tulingmall-gateway
SpringCloudGateway/RoutingFilter	2021-07-27 13:02:00	122	2	2	spring-cloud-gat...	tulingmall-gateway
SpringCloudGateway/sendRequest	2021-07-27 13:02:00	120	22	22	spring-cloud-gat...	tulingmall-gateway
SpringCloudGateway/RoutingFilter	2021-07-27 13:01:59	54	92	92	SpringMVC	tulingmall-order
IPOST/cart/add	2021-07-27 13:01:59	22	1	1	mysql-connector...	tulingmall-order
MySQL/DB/PreparedStatement/execute	2021-07-27 13:02:00	1	1	1	mysql-connector...	tulingmall-order
MySQL/DB/PreparedStatement/execute	2021-07-27 13:02:00	2	2	2	mysql-connector...	tulingmall-order
MySQL/DB/Connection/commit	2021-07-27 13:02:00	3	3	3	mysql-connector...	tulingmall-order

再看一下skywalking这条sql的执行时间也到了几毫秒了，对应的post请求的总执行时间也下降了很多，当然还有100多毫秒，这个应该算正常，在高并发情况下，请求之间争用系统资源会有等待，导致链路调用过程中会有损耗

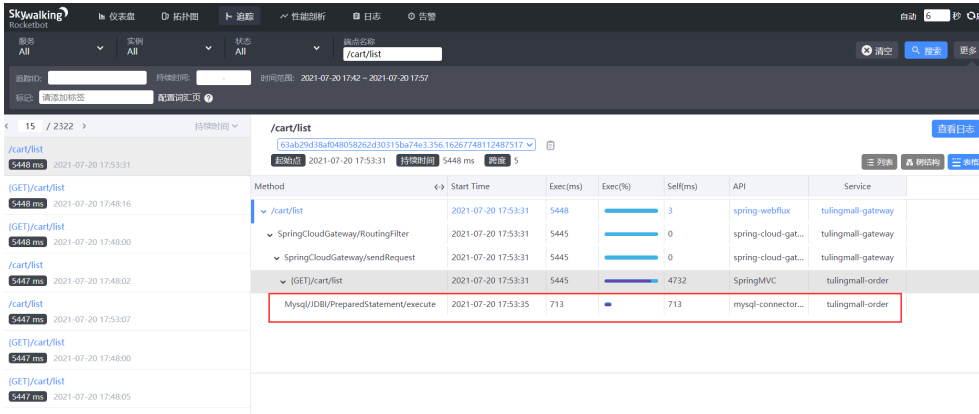
压测查询购物车接口

Thread Group Configuration:

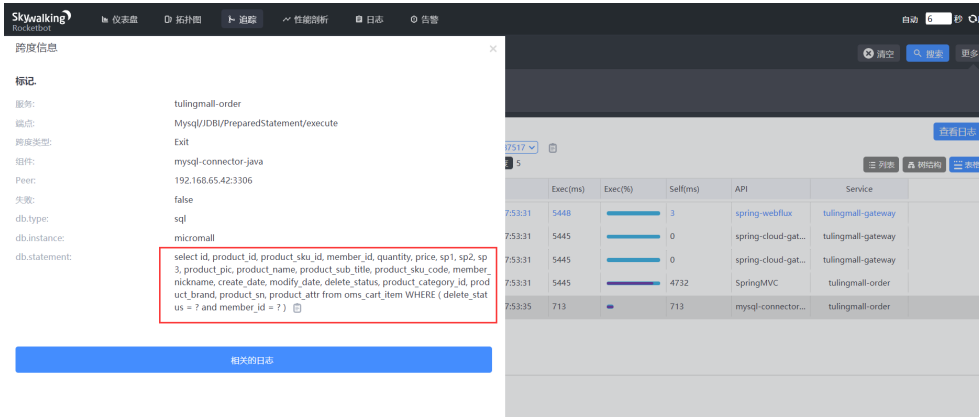
- Name: Thread Group
- Thread Properties:
 - Number of Threads (users): 100
 - Ramp-Up Period (seconds): 1
 - Loop Count: Forever 10000
- Delay Thread creation until needed:
- Scheduler:
- Scheduler Configuration:
 - Duration (seconds):
 - Startup delay (seconds):



qps只有40左右，就一个查询操作，这个qps还是有点低的，需要优化了
我们看下skywalking这个接口的执行链路跟踪



这里有一个mysql操作耗时较多



拉出sql看下执行计划，发现又是一个全表扫描



我们来对这条sql做下索引优化，我们上面对oms_cart_item表加过一个联合索引(product_id,member_id)，这条sql语句没有用上，因为不满足左前缀原则，我们把索引联合索引字段排序改成(member_id, product_id)，再看下执行计划

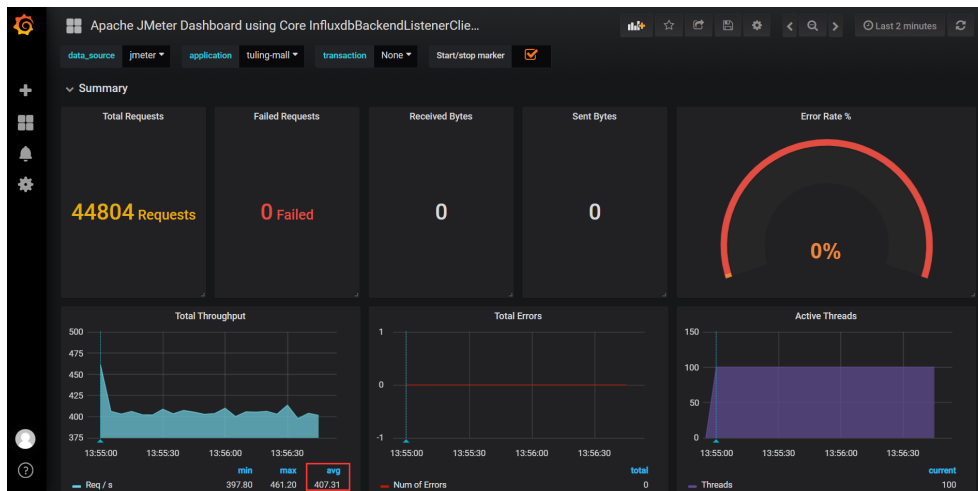
```

1  EXPLAIN SELECT
2  id,
3  product_id,
4  product_sku_id,
5  member_id,
6  quantity,
7  price,
8  sp1,
9  sp2,
10 sp3,
11 product_pic,
12 product_name,
13 product_sub_title,
14 product_sku_code,
15 member_nickname,
16 create_date,
17 modify_date,
18 delete_status,
19 product_category_id,
20 product_brand,
21 product_sn,
22 product_attr
23 FROM
24 oms_cart_item
25 WHERE
26 (
27   delete_status = 0
28   AND member_id = 200
29 )

```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	oms_cart_item	(Null)	ref	idx_member_id_and_product_idx_member_id_and_product_id		9	const	204	10	Using where

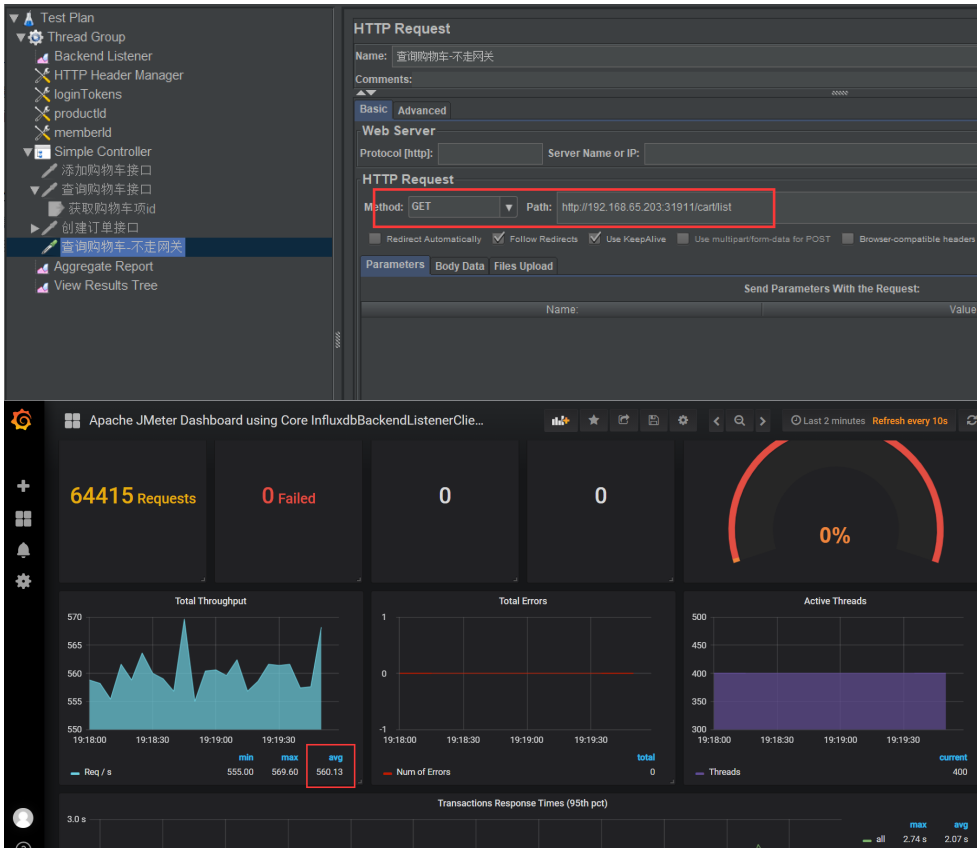
已经走了索引，不过这里一个用户的购物车记录有两百多条，这个是因为我们反复压测加购物车接口，但是总共就1000个用户，所以一个用户购物车里有很多商品，这里其实还要注意，这条sql最好要做下分页，我们再重新压测看下效果



qps能接近400了，我们再看下skywalking链路追踪，之前那条sql执行已经下降到十几毫秒了

Method	Start Time	Exec(ms)	Exec(%)	Self(ms)	API	Service
/cart/list	2021-07-20 18:45:29	1509	1229		spring-webflux	tulingmall-gateway
SpringCloudGateway/RoutingFilter	2021-07-20 18:45:29	280	0		spring-cloud-gat...	tulingmall-gateway
SpringCloudGateway/SendRequest	2021-07-20 18:45:29	280	1		spring-cloud-gat...	tulingmall-gateway
(GET)/cart/list	2021-07-20 18:45:29	279	264		SpringMVC	tulingmall-order
MySQL/DB/PreparedStatement/execute	2021-07-20 18:45:29	15	1	15	mysql-connector...	tulingmall-order

不过我们从上图发现gateway网关的请求时间依然很长，我们试着优化下，我们尝试着跳过网关直接压下查询购物车接口看下qps是否有提升



我们看到qps接近翻倍了，看来网关gateway应用应该要查下是否有优化的地方了

优化网关gateway应用

因为gateway是java应用，我们可以进入容器用arthas和jdk自带的一些命令分析下jvm情况
在压测的时候进入gateway容器

```
1 kubectl exec -it tulingmall-gateway-deployment-ddc85c775-gkn2s -- sh
```

在容器里下载arthas

```
1 wget https://arthas.gitee.io/arthas-boot.jar
2 java -jar arthas-boot.jar
```

通过arthas的一些常用命令来分析下jvm情况

```
1 dashboard
```

ID	NAME	GROUP	PRIORITY	STATE	%CPU	DELTA_TIME	TIME	INTERRUPTED	DAEMON
125	reactor-http-epoll-4	main	5	RUNNABLE	21.55	1.4377	0:45.750	false	true
127	reactor-http-epoll-6	main	5	RUNNABLE	20.96	1.048	0:49.772	false	true
66	reactor-http-epoll-1	main	5	RUNNABLE	18.98	0.949	0:49.883	false	true
129	reactor-http-epoll-8	main	5	RUNNABLE	18.97	0.948	0:50.989	false	true
124	reactor-http-epoll-3	main	5	RUNNABLE	17.82	0.899	0:49.514	false	true
112	reactor-http-epoll-2	main	5	RUNNABLE	17.12	0.856	0:53.502	false	true
128	reactor-http-epoll-7	main	5	RUNNABLE	16.63	0.831	0:47.335	false	true
126	reactor-http-epoll-5	main	5	RUNNABLE	15.54	0.777	0:49.320	false	true
117	sentinel-time-tick-thread	main	5	TIMED_WAITING	2.17	0.108	0:6.170	false	true
-1	C1 CompilerThread0	-	-1	-	1.44	0.072	0:12.412	false	true
-1	C2 CompilerThread0	-	-1	-	0.64	0.032	0:27.648	false	true

Memory	used	total	max	usage	GC
heap	283M	427M	2635M	10.75%	gc-ps_scavenge.count 78
ps Eden space	177M	265M	956M	13.53%	gc-ps_scavenge.time(ms) 1158
ps survivor space	11M	12M	12M	99.23%	gc-ps_marksweep.count 3
ps old gen	94M	149M	1976M	4.77%	gc-ps_marksweep.time(ms) 295
nonheap	143M	148M	-1	96.95%	
code cache	46M	46M	248M	19.19%	
metaspace	86M	90M	-1	96.05%	
compressed class space	11M	11M	1024M	1.09%	
direct	513K	513K	-	100.00%	


```
Runtime
os.name Linux
os.version 3.10.0-1160.25.1.el7.x86_64
java.version 1.8.0_111
java.home /usr/lib/jvm/java-8-openjdk-amd64/jre
systemload.average 2.53
processors 8
timestamp/uptime Sat Jul 24 15:33:12 CST 2021/444s
```

dashboard还算正常，线程和堆占用的资源没有特别的异常情况，我们再执行下thread命令查看下更多的线程执行情况

```
1 thread
```

ID	NAME	GROUP	PRIORITY	STATE	PCPU	DELTA TIME	TIME	INTERRUPTED	DAEMON
102	reactor-http-epoll-4	main	5	RUNNABLE	62.78	0.126	6:46.983	false	true
106	reactor-http-epoll-8	main	5	RUNNABLE	62.27	0.125	6:49.866	false	true
105	reactor-http-epoll-7	main	5	RUNNABLE	31.86	0.064	6:43.690	false	true
101	reactor-http-epoll-3	main	5	RUNNABLE	19.13	0.038	6:50.975	false	true
103	reactor-http-epoll-5	main	5	RUNNABLE	12.8	0.025	6:46.081	false	true
104	reactor-http-epoll-6	main	5	RUNNABLE	12.33	0.024	6:43.846	false	true
88	reactor-http-epoll-2	main	5	RUNNABLE	12.04	0.024	6:46.635	false	true
27	grpc-nio-worker-ELC-1-3	main	5	RUNNABLE	10.7	0.021	1:28.398	false	true
-1	C1 CompilerThread3	-	-1	-	8.16	0.016	0:16.972	false	true
54	reactor-http-epoll-1	main	5	RUNNABLE	8.02	0.016	6:47.261	false	true
6	DataCarrier_DEFAULT_Consumer_0_Thread	main	5	TIMED_WAITING	3.81	0.007	0:38.949	false	true
94	sentinel-time-tick-thread	main	5	TIMED_WAITING	2.54	0.005	1:1.990	false	true
692	grpc-default-executor-11	main	5	TIMED_WAITING	1.5	0.003	0:2.018	false	true
752	arthas-command-execute	system	5	RUNNABLE	0.6	0.001	0:0.013	false	true
38	com.alibaba.nacos.client.Worker.fixed-192	main	5	TIMED_WAITING	0.3	0.000	0:12.303	false	true
15	DataCarrier_gRPC_log_Consumer_0_Thread	main	5	TIMED_WAITING	0.22	0.000	0:6.226	false	true
-1	VM Periodic Task Thread	-	-1	-	0.2	0.000	0:2.399	false	true
691	grpc-default-executor-10	main	5	TIMED_WAITING	0.17	0.000	0:1.928	false	true
-1	C2 CompilerThread1	-	-1	-	0.05	0.000	0:41.924	false	true
-1	C2 CompilerThread0	-	-1	-	0.04	0.000	0:42.224	false	true
-1	C2 CompilerThread2	-	-1	-	0.02	0.000	0:40.800	false	true
5	SkyWalkingAgent-1-LogFileWriter-0	main	5	TIMED_WAITING	0.02	0.000	0:0.208	false	true
2	Reference Handler	-	10	WAITING	0.0	0.000	0:0.067	false	true
3	Finalizer	system	8	WAITING	0.0	0.000	0:0.111	false	true
4	Signal Dispatcher	system	9	RUNNABLE	0.0	0.000	0:0.001	false	true
739	Attach Listener	system	0	RUNNABLE	0.0	0.000	0:0.033	false	true

没有被阻塞的线程，还算正常，不过我们看到gateway内部的线程池很繁忙，占用cpu很高

1 thread -n 3 #最繁忙的3个线程（占用cpu最多的前3个），输出栈信息

```
[arthas@1]$ thread -n 3
"reactor-http-epoll-1" Id=65 cpuUsage=23.44% deltaTime=47ms time=724179ms RUNNABLE (in native)
  at io.netty.channel.epoll.Native.epollWait0(Native Method)
  at io.netty.channel.epoll.Native.epollWait(Native.java:96)
  at io.netty.channel.epoll.EpollEventLoop.epollWait(EpollEventLoop.java:276)
  at io.netty.channel.epoll.EpollEventLoop.run(EpollEventLoop.java:305)
  at io.netty.util.concurrent.SingleThreadEventExecutor$5.run(SingleThreadEventExecutor.java:918)
  at io.netty.util.internal.ThreadExecutorMap$2.run(ThreadExecutorMap.java:74)
  at io.netty.util.concurrent.FastThreadLocalRunnable.run(FastThreadLocalRunnable.java:30)
  at java.lang.Thread.run(Thread.java:745)

"reactor-http-epoll-5" Id=94 cpuUsage=23.0% deltaTime=46ms time=725930ms RUNNABLE
  at io.netty.channel.epoll.Native.epollWait0(Native Method)
  at io.netty.channel.epoll.Native.epollWait(Native.java:96)
  at io.netty.channel.epoll.EpollEventLoop.epollWait(EpollEventLoop.java:276)
  at io.netty.channel.epoll.EpollEventLoop.run(EpollEventLoop.java:305)
  at io.netty.util.concurrent.SingleThreadEventExecutor$5.run(SingleThreadEventExecutor.java:918)
  at io.netty.util.internal.ThreadExecutorMap$2.run(ThreadExecutorMap.java:74)
  at io.netty.util.concurrent.FastThreadLocalRunnable.run(FastThreadLocalRunnable.java:30)
  at java.lang.Thread.run(Thread.java:745)

"reactor-http-epoll-6" Id=95 cpuUsage=22.9% deltaTime=46ms time=718855ms RUNNABLE
  at org.springframework.util.MimeType.<init>(MimeType.java:175)
  at org.springframework.util.MimeTypeUtils.parseMimeType(MimeTypeUtils.java:242)
  at org.springframework.http.MediaType.parseMediaType(MediaType.java:532)
  at org.springframework.web.reactive.result.condition.AbstractMediaTypeExpression.<init>(AbstractMediaTypeExpression.java:53)
  at org.springframework.web.reactive.result.condition.ProducesRequestCondition.ProducesRequestCondition.<init>(ProducesRequestCondition.java:53)
  at org.springframework.web.reactive.result.condition.ProducesRequestCondition.<init>(ProducesRequestCondition.java:53)
```

都是gateway的netty线程，也没有太多问题

gateway可以调节netty线程池的大小，有一个参数叫 reactor.netty.ioWorkerCount，但是一般不建议调整，因为这个线程池的内部就是按照cpu核数(我这个gateway部署的机器是8核的)来确定的，调大意义也不是很大，当然我们可以试着来调试下，在gateway的deployment文件里加上这个 -Dreactor.netty.ioWorkerCount=16 JVM参数即可，我们重启再来看下gateway服务内部的线程执行情况

ID	NAME	GROUP	PRIORITY	STATE	PCPU	DELTA TIME	TIME	INTERRUPTED	DAEMON
108	reactor-http-epoll-2	main	5	RUNNABLE	27.87	0.057	1:44.156	false	true
121	reactor-http-epoll-15	main	5	RUNNABLE	22.63	0.046	1:40.651	false	true
109	reactor-http-epoll-3	main	5	RUNNABLE	19.65	0.040	1:43.484	false	true
112	reactor-http-epoll-6	main	5	RUNNABLE	18.05	0.037	1:46.052	false	true
122	reactor-http-epoll-16	main	5	RUNNABLE	17.84	0.036	1:42.216	false	true
120	reactor-http-epoll-14	main	5	RUNNABLE	13.45	0.027	1:42.905	false	true
110	reactor-http-epoll-4	main	5	RUNNABLE	13.03	0.026	1:42.021	false	true
117	reactor-http-epoll-11	main	5	RUNNABLE	12.87	0.026	1:40.653	false	true
115	reactor-http-epoll-9	main	5	RUNNABLE	10.39	0.021	1:40.523	false	true
113	reactor-http-epoll-7	main	5	RUNNABLE	10.2	0.021	1:41.183	false	true
27	grpc-nio-worker-ELC-1-3	main	5	RUNNABLE	10.18	0.021	0:30.649	false	true
119	reactor-http-epoll-13	main	5	RUNNABLE	9.38	0.019	1:38.695	false	true
118	reactor-http-epoll-12	main	5	RUNNABLE	9.38	0.019	1:40.828	false	true
54	reactor-http-epoll-1	main	5	RUNNABLE	8.15	0.016	1:45.854	false	true
114	reactor-http-epoll-8	main	5	RUNNABLE	7.81	0.016	1:44.223	false	true
116	reactor-http-epoll-10	main	5	RUNNABLE	6.76	0.013	1:42.425	false	true
111	reactor-http-epoll-5	main	5	RUNNABLE	5.2	0.010	1:44.935	false	true
6	DataCarrier_DEFAULT_Consumer_0_Thread	main	5	TIMED_WAITING	3.0	0.006	0:12.837	false	true
253	arthas-command-execute	system	5	RUNNABLE	2.87	0.005	0:0.018	false	true
127	sentinel-time-tick-thread	main	5	TIMED_WAITING	2.66	0.005	0:14.295	false	true
137	grpc-default-executor-1	main	5	TIMED_WAITING	1.37	0.002	0:1.743	false	true
-1	C1 CompilerThread3	-	-1	-	0.9	0.001	0:13.038	false	true
39	com.alibaba.nacos.client.Worker.fixed-192	main	5	TIMED_WAITING	0.41	0.000	0:3.699	false	true
129	sentinel-metrics-record-task-thread-1	main	5	TIMED_WAITING	0.39	0.000	0:0.333	false	true
138	grpc-default-executor-2	main	5	TIMED_WAITING	0.33	0.000	0:2.124	false	true
15	DataCarrier_gRPC_log_Consumer_0_Thread	main	5	TIMED_WAITING	0.22	0.000	0:1.622	false	true

虽然线程数增加了，但是压测结果并没有显著增加，因为cpu核数只有8个，所以对于gateway这种应用我们机器要尽量把cpu的配置加大一点，它对cpu的消耗很大，继续查下其它原因

1 thread -b #输出阻塞的线程栈信息，如果响应慢，阻塞状态的线程比较多，我们需要重点关注

```
[arthas@1]$ thread -b
No most blocking thread found!
```

我们再用jvm命令看下有没有死锁

1 jvm

```

-----
THREAD
-----
COUNT                91
DAEMON-COUNT          87
PEAK-COUNT            92
STARTED-COUNT         2323
DEADLOCK-COUNT        0
-----
FILE-DESCRIPTOR

```

如果有死锁，可以通过thread -b命令找出死锁线程堆栈，类似下图

```

[arthas@5395]$ thread -b
"Thread-1" Id=9 BLOCKED on java.lang.Object@2d77936d owned by "Thread-2" Id=10
  at com.tuling.jvm.ArthasTest.lambda$deadThread$2(ArthasTest.java:63)
  - blocked on java.lang.Object@2d77936d
  - locked java.lang.Object@12c518c5 <---- but blocks 1 other threads!
  at com.tuling.jvm.ArthasTest.lambda$2/303563356.run(Unknown Source)
  at java.lang.Thread.run(Thread.java:748)

```

总体看来没有太大问题，我们再用jstat看下gc情况

```
1 jstat -gcutil 1 1000 1000
```

S0	S1	E	O	M	CCS	YGC	YGCT	FGC	FGCT	GCT
75.00	0.00	69.48	88.50	95.96	94.18	2236	25.044	9	2.436	27.480
0.00	94.79	27.20	88.66	95.96	94.18	2237	25.056	9	2.436	27.492
0.00	94.79	83.25	88.66	95.96	94.18	2237	25.056	9	2.436	27.492
100.00	0.00	39.69	89.20	95.96	94.18	2238	25.065	9	2.436	27.501
100.00	0.00	96.95	89.20	95.96	94.18	2238	25.065	9	2.436	27.501
0.00	73.21	62.90	89.28	95.96	94.18	2239	25.075	9	2.436	27.511
90.00	0.00	21.68	89.46	95.96	94.18	2240	25.084	9	2.436	27.521
90.00	0.00	84.26	89.46	95.96	94.18	2240	25.084	9	2.436	27.521
0.00	93.75	49.92	89.55	95.96	94.18	2241	25.092	9	2.436	27.528
92.50	0.00	15.49	89.64	95.96	94.18	2242	25.100	9	2.436	27.536
92.50	0.00	78.91	89.64	95.96	94.18	2242	25.100	9	2.436	27.536
0.00	86.25	45.26	89.71	95.96	94.18	2243	25.107	9	2.436	27.544
100.00	0.00	13.97	89.88	95.96	94.18	2244	25.115	9	2.436	27.551
100.00	0.00	81.61	89.88	95.96	94.18	2244	25.115	9	2.436	27.551
0.00	85.94	54.24	89.97	95.96	94.18	2245	25.123	9	2.436	27.559
87.50	0.00	24.79	90.05	95.96	94.18	2246	25.133	9	2.436	27.570
87.50	0.00	98.16	90.05	95.96	94.18	2246	25.133	9	2.436	27.570
0.00	95.31	73.43	90.13	95.96	94.18	2247	25.143	9	2.436	27.580
100.00	0.00	48.93	90.34	95.96	94.18	2248	25.153	9	2.436	27.589
0.00	86.25	22.34	90.49	95.96	94.18	2249	25.165	9	2.436	27.601
100.00	0.00	0.00	90.74	95.96	94.18	2250	25.173	9	2.436	27.609

我们观察一段时间发现，FGC还算正常，但是YGC几乎每秒1次，感觉有点频繁，可能是年轻代太小了，我们看下堆内存情况，当然如果FGC比较频繁我们可以通过jmap查看下对象占用内存的情况

```
1 jmap -histo 1 | head -20
```

num	#instances	#bytes	class name
1:	2949803	70795272	java.util.LinkedList\$Node
2:	428041	66603344	[C
3:	2911082	46577312	org.apache.skywalking.apm.agent.core.context.trace.NoopSpan
4:	42870	20812512	[I
5:	33144	15865032	[B
6:	342133	8211192	java.lang.String
7:	82427	8051384	[Ljava.lang.Object;
8:	108976	4359040	java.util.LinkedHashMap\$Entry
9:	46412	3301736	[Ljava.util.HashMap\$Node;
10:	50672	2837632	java.util.LinkedHashMap
11:	30975	2516400	[Lio.netty.util.Recycler\$DefaultHandle;
12:	94523	2268552	ch.qos.logback.classic.spi.StackTraceElementProxy
13:	69842	2234944	java.util.concurrent.ConcurrentHashMap\$Node
14:	18512	2040080	java.lang.Class
15:	62809	2009888	java.lang.StackTraceElement

结果是比较正常的，skywalking的对象多也是正常的，因为每条请求都生成skywalking相关对象

我们看下当前堆内存情况

```
1 jinfo -flags 1
```

```

Attaching to process ID 1, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 25.111-b14
Non-default VM flags: -XX:CICompilerCount=4 -XX:InitialHeapSize=195035136 -XX:MaxHeapSize=3107979264 -XX:MaxNewSize=1035993088 -XX:MinHeapDeltaBytes=524288 -XX:NewSize=65011712 -XX:OldSize=130823424 -XX:+UseCompressedClassPointers -XX:+UseCompressedOops -XX:+UseParallelGC

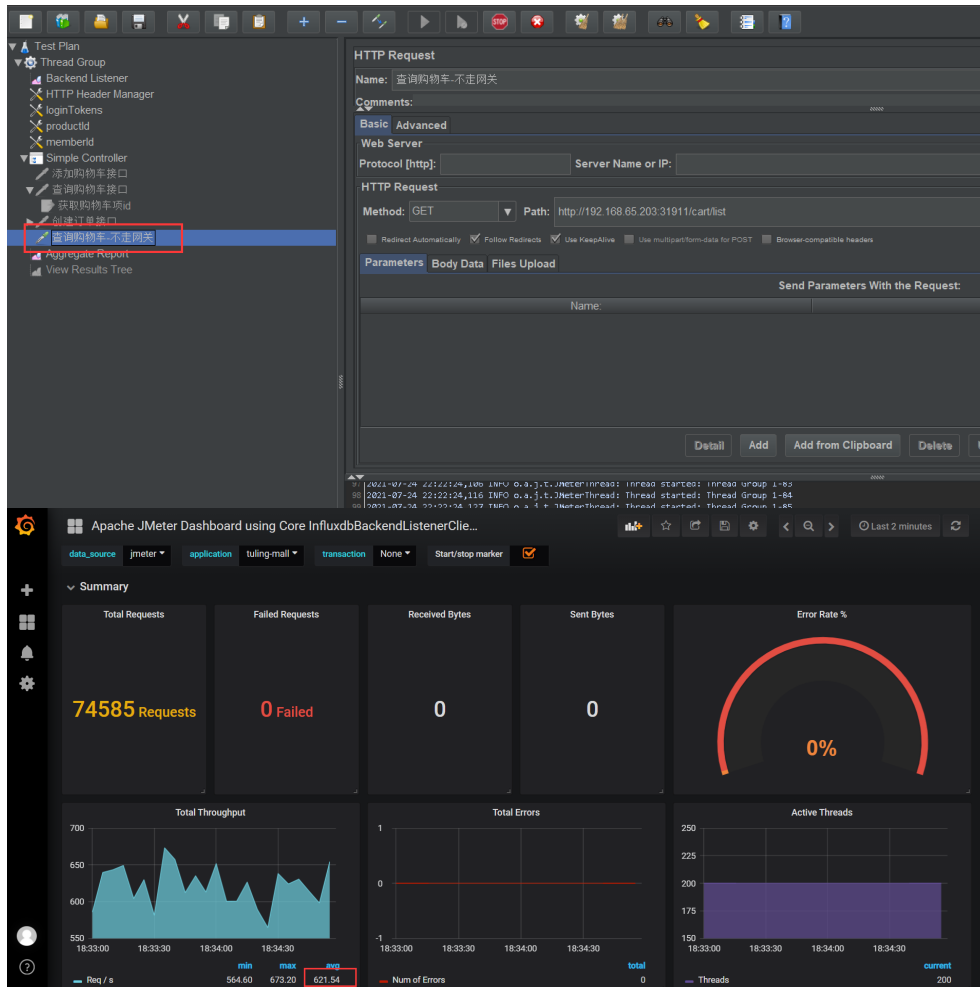
```

因为我们没有设置JVM参数，所以默认初始内存是比较小的，而且默认的jvm堆设置会导致程序运行的过程中JVM堆的经常性扩容，也会影响程序性能，我们尝试着设置下把堆内存设置到1G看下效果，我们给gateway应用的JVM启动参数增加了如下参数

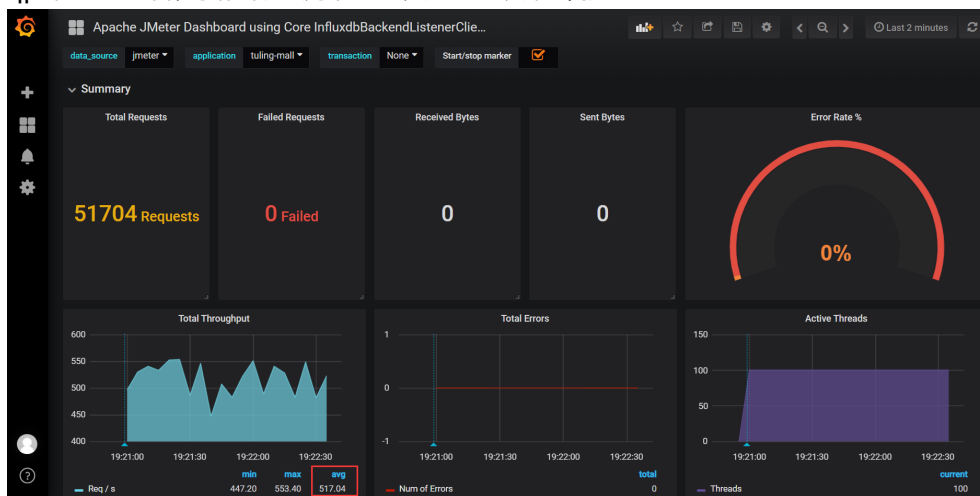
```
1 -Xms1G -Xmx1G -Xmn512M -Xss512K -XX:MetaspaceSize=256M -XX:MaxMetaspaceSize=256M
```

重新测下不走网关的查询购物车接口和走网关的查询购物车接口的对比情况，看下调整了gateway的JVM参数后的走网关接口的qps是否还会降低一半

我们先压测下不走网关的查询购物车接口：



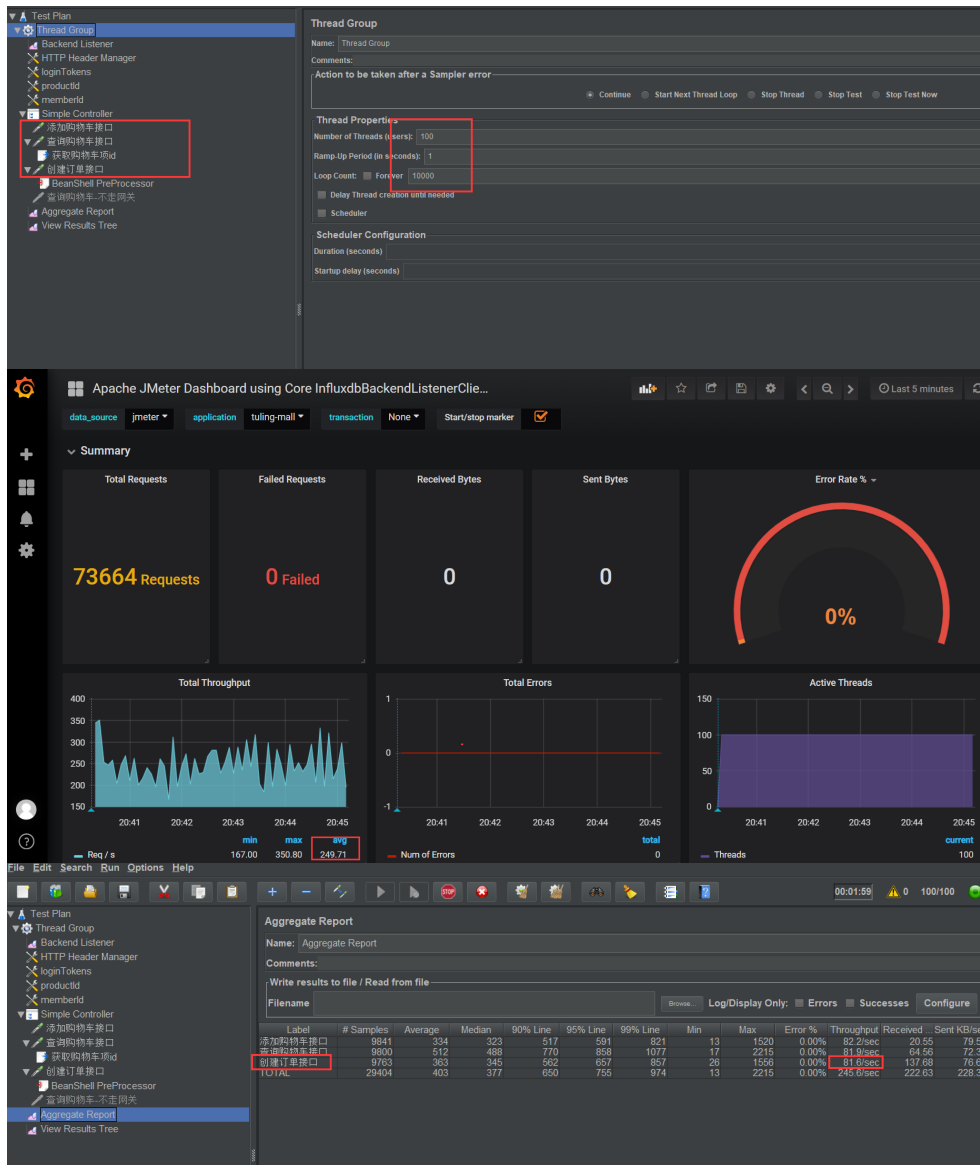
qps在620左右，我们再压测下走网关的查询购物车接口：



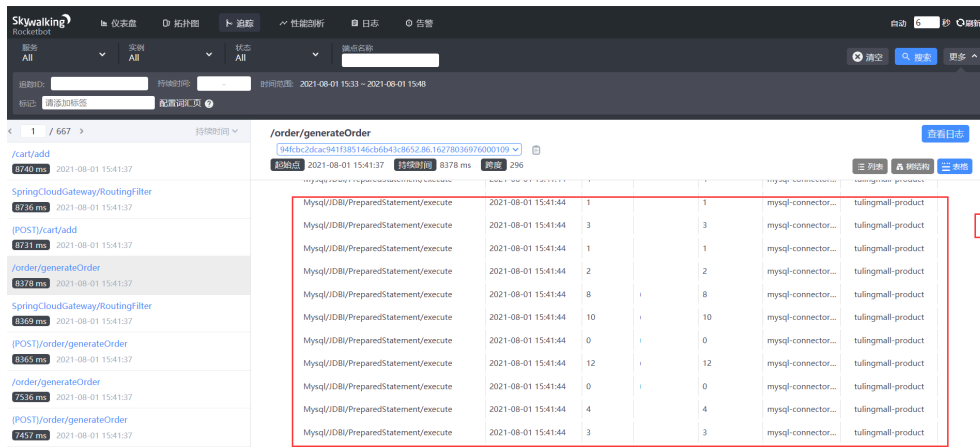
这时走网关的查询购物车接口qps也接近了500左右，有一点提升，从经验上来看微服务架构里加了网关这个服务性能会有一些的损耗，但是这个损耗依然有点大，优化效果不是很明显。

因为网关gateway对cpu性能要求较高，我怀疑是不是这两台机器的性能有些区别，我把两台node1和node2上部署的服务对调了后再压测发现经过网关和不经过的差别已经不是很大了。有的时候压测结果跟机器性能会有较大关系。

压测创建订单接口



我们发现创建订单接口的qps不到100，凭经验来说应该是有优化余地的，注意这里grafana里展示的是3个接口的总qps



我们查看了下skywalking的创建订单接口的链路，发现有一部分请求链路非常长，里面包含上百条查询商品的执行sql，这条sql本身执行比较快，问题不大，不过为什么一次创建订单会执行上百次的查询商品的sql了，从代码定位问题，发现由于我们之前压测时因为压测用户数据不多，所以一个用户下面加入了上百个商品，下单时会查询购物车里每个商品相关的信息，显然这里是优化一下的，查询购物车子项数据我们上面讲过最好做下分页，那么我们就可以针对分页数据用 in 这样的sql批量查询商品信息。

其次我们发现一次创建订单请求时间较长，但是里面每一步的具体操作时间很正常，这可能是由于这些操作有锁等待的情况，我们可以进入order服务后台用arthas来分析下

然后看了下应用和mysql的机器使用率，我们发现order和gateway服务占用cpu较高，因为现在主要就在压测order服务

```
top - 19:47:02 up 2 days, 7:23, 1 user, load average: 12.06, 4.80, 5.89
Tasks: 280 total, 2 running, 278 sleeping, 0 stopped, 0 zombie
%Cpu(s): 46.9 us, 9.9 sy, 0.0 ni, 33.0 id, 0.0 wa, 0.0 hi, 9.7 si, 0.5 st
KiB Mem: 12136788 total, 3484400 free, 4319956 used, 4332432 buff/cache
KiB Swap: 0 total, 0 free, 0 used, 7424140 avail Mem

  PID USER      PR  NI   VIRT   RES   SHR  S  %CPU  %MEM     TIME+ COMMAND
 23530 root      20   0 9149972  1.3g 16192 S 341.4 11.1 360:49.23 java
 24427 root      20   0 8922312  1.4g 16012 S 158.3 12.2  73:12.65 java
  1398 root      20   0 1071960  78152 8880 S 13.6  0.6 134:35.15 dockerd
 23511 root      20   0 108808   4012 3016 S  8.3  0.0  2:45.45 containerd-shim
   832 root      20   0 2177244 64768 20424 S  5.0  0.5 111:09.02 kubelet
 4230 root      20   0 1954008 47444 19048 S  3.3  0.4 101:16.96 calico-node
   14 root      20   0 0         0     0 S  1.0  0.0  5:48.33 ksoftirqd/1
   19 root      20   0 0         0     0 S  1.0  0.0  5:03.55 ksoftirqd/2
   29 root      20   0 0         0     0 S  1.0  0.0  5:50.23 ksoftirqd/4

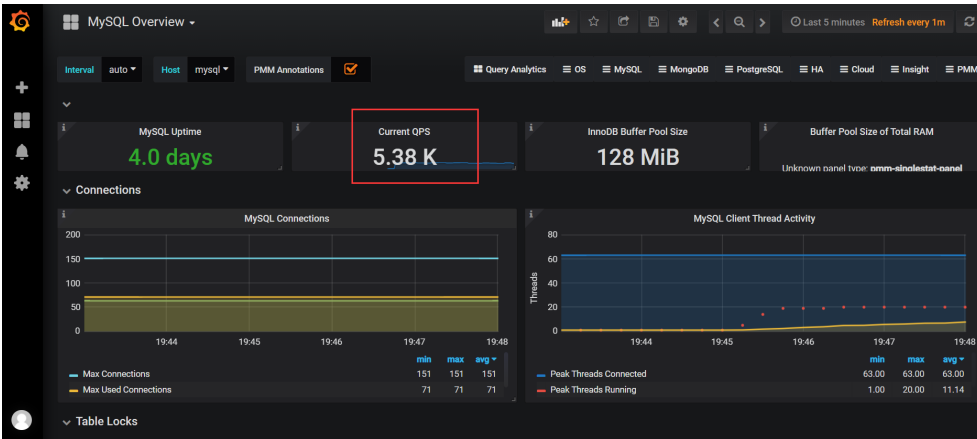
  PID USER      PR  NI   VIRT   RES   SHR  S  %CPU  %MEM     TIME+ COMMAND
   360 root      20   0 9028456  1.4g 12396 S 133.9 11.9  40:24.22 java
 25295 root      20   0 8921076 722604 9080 S  35.2  6.0  20:19.53 java
  1518 root      20   0 1112876 72716 7100 S  32.6  0.6 113:55.22 dockerd
   338 root      20   0 110216  3804 2872 S 16.9  0.0  4:20.78 containerd-shim
 3833 root      20   0 108808  8516 2032 S  6.0  0.1 25:28.95 containerd-shim
 22255 33        20   0 384756 60360 1452 S  5.0  0.5  5:02.40 nginx
 22167 33        20   0 384756 60360 1452 S  4.0  0.5  5:04.38 nginx
 22165 33        20   0 384884 60568 1468 S  3.7  0.5  4:50.71 nginx
 22166 33        20   0 384756 60388 1476 S  3.3  0.5  5:00.68 nginx
 22169 33        20   0 384756 60352 1452 S  3.3  0.5  4:54.51 nginx
```

然后看了下mysql机器192.168.65.42，cpu使用率非常高

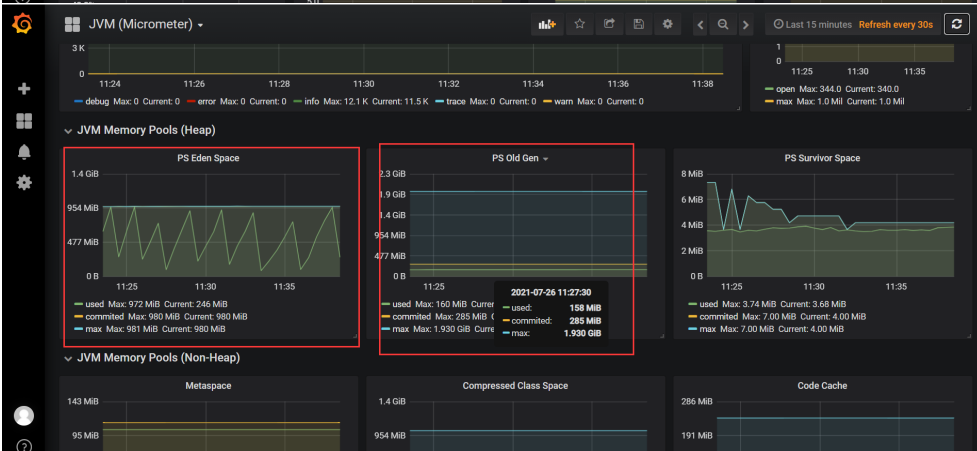
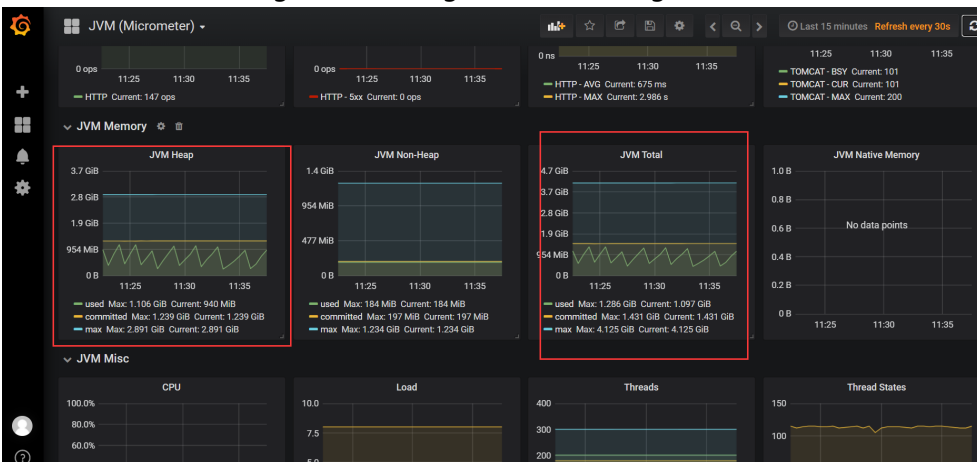
```
top - 19:47:33 up 9 days, 31 min, 1 user, load average: 6.88, 3.54, 3.54
Tasks: 331 total, 1 running, 330 sleeping, 0 stopped, 0 zombie
%Cpu(s): 40.8 us, 8.0 sy, 0.0 ni, 46.1 id, 1.1 wa, 0.0 hi, 3.6 si, 0.4 st
KiB Mem: 24521464 total, 4322404 free, 14548212 used, 5650848 buff/cache
KiB Swap: 12386300 total, 12382460 free, 3840 used, 9564712 avail Mem

  PID USER      PR  NI   VIRT   RES   SHR  S  %CPU  %MEM     TIME+ COMMAND
 16421 ods      20   0 6507520 600220 13136 S 620.9  2.4 4953:36 mysqld
 17867 root      20   0  12.0g  5.6g 14960 S  6.3 23.8 191:31.13 java
 16298 ods      20   0  61688 15296 1836 S  4.0  0.1 12:34.39 redis-server
 16079 1000     20   0 8040440 927376 14836 S  2.0  3.8 52:10.71 java
 16973 root      20   0 8996336 944384 13652 S  2.0  3.9 63:25.56 java
 16414 ods      20   0 278264 72396 8924 S  1.7  0.3 42:45.30 mongod
   343 root      20   0  70644 18516 1468 S  1.0  0.1 93:53.54 plymouthd
 16976 1000     20   0 1353460 183272 13556 S  1.0  0.7 29:32.69 node
   14 root      20   0 0         0     0 S  0.7  0.0  1:08.75 ksoftirqd/1
   383 root      0 -20 0         0     0 S  0.7  0.0  0:57.17 kworker/4:1H
   492 root      0 -20 0         0     0 S  0.7  0.0  0:56.89 kworker/2:1H
   633 root      0 -20 0         0     0 S  0.7  0.0  0:55.43 kworker/7:1H
   756 root      0 -20 0         0     0 S  0.7  0.0  0:29.41 kworker/9:1H
```

我们再观察下mysql的grafana，发现qps非常高，说明mysql正在大量的执行sql，所以cpu高还算正常



我们再看下order服务的grafana, 发现gc还算比较正常, gc主要也是发生在年轻代



我们再进入下order服务的容器, 用artha1s观察下是否有锁等待的情况

我们执行下dashboard命令观察下order服务的整体情况, 发现没特别异常的情况

ID	NAME	GROUP	PRIORITY	STATE	%CPU	DELTA TIME	TIME	INTERRUPTED	DAEMON
25	grpc-nio-worker-ELG-1-2	main	5	RUNNABLE	6.42	0.321	0:48.848	false	true
-1	C2 CompilerThread1	-	-1	-	3.4	0.170	1:12.793	false	true
6	DataCarrier.DEFAULT.Consumer.0.Thread	main	5	TIMED_WAITING	2.82	0.141	3:35.296	false	true
180	http-nio-8844-ClientPoller	main	5	RUNNABLE	2.45	0.187	5:52.149	false	true
77	letuce-epollEventLoop-4-1	main	5	RUNNABLE	1.98	0.899	3:44.823	false	true
2799	http-nio-8844-exec-1162	main	5	WAITING	1.7	0.085	0:15.189	false	true
89	http-nio-8844-BlockPoller	main	5	RUNNABLE	1.68	0.084	5:46.943	false	true
2671	http-nio-8844-exec-1069	main	5	WAITING	1.67	0.083	0:16.464	false	true
2763	http-nio-8844-exec-1126	main	5	RUNNABLE	1.66	0.083	0:14.784	false	true
2744	http-nio-8844-exec-1107	main	5	TIMED_WAITING	1.57	0.078	0:15.070	false	true
-1	C2 CompilerThread0	-	-1	-	1.55	0.077	1:11.526	false	true

Memory	used	total	max	usage	GC
heap	225M	607M	2635M	8.57%	gc_ps_scavenge.count 4922
ps_survivor_space	3M	249M	977M	5.24%	gc_ps_scavenge.time(ms) 68258
ps_survivor_space	3M	3M	3M	93.75%	gc_ps_marksweep.count 24
ps_old_gen	171M	354M	1976M	8.67%	gc_ps_marksweep.time(ms) 7532
nonheap	193M	203M	-1	95.05%	
code_cache	67M	68M	240M	28.28%	
metaspace	112M	119M	-1	93.54%	
compressed_class_space	13M	15M	1024M	1.34%	
direct	1M	1M	-	100.00%	


```

Runtime
os.name Linux
os.version 3.10.0-1160.25.1.el7.x86_64
java.version 1.8.0_111
java.home /usr/lib/jvm/java-8-openjdk-amd64/jre
systemload.average 4.43
processors 8
timestamp/uptime Tue Jul 27 20:51:42 CST 2021/126885
[artha@i15 ~]

```



```

11 exclusions: "*.js,*.gif,*.jpg,*.png,*.css,*.ico,/druid/*" #不统计这些请求数据
12 stat-view-servlet: #访问监控网页的登录用户名和密码
13 login-username: druid
14 login-password: druid

```

我们调整到如下值

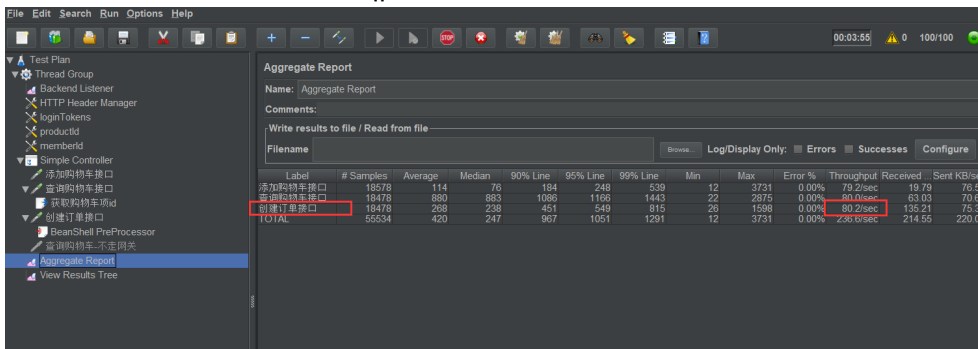
```

1 spring:
2   datasource:
3     url: jdbc:mysql://192.168.65.42:3306/micromall?
serverTimezone=UTC&useSSL=false&useUnicode=true&characterEncoding=UTF-8
4     username: root
5     password: root
6   druid:
7     initial-size: 10 #连接池初始化大小
8     min-idle: 20 #最小空闲连接数
9     max-active: 100 #最大连接数
10  web-stat-filter:
11    exclusions: "*.js,*.gif,*.jpg,*.png,*.css,*.ico,/druid/*" #不统计这些请求数据
12    stat-view-servlet: #访问监控网页的登录用户名和密码
13    login-username: druid
14    login-password: druid

```

重启下pod服务，再压测试下，重启pod可以用直接删除的方式，k8s会自动再启动一个pod

我们再压测看下情况，遗憾的是qps并没有什么变化

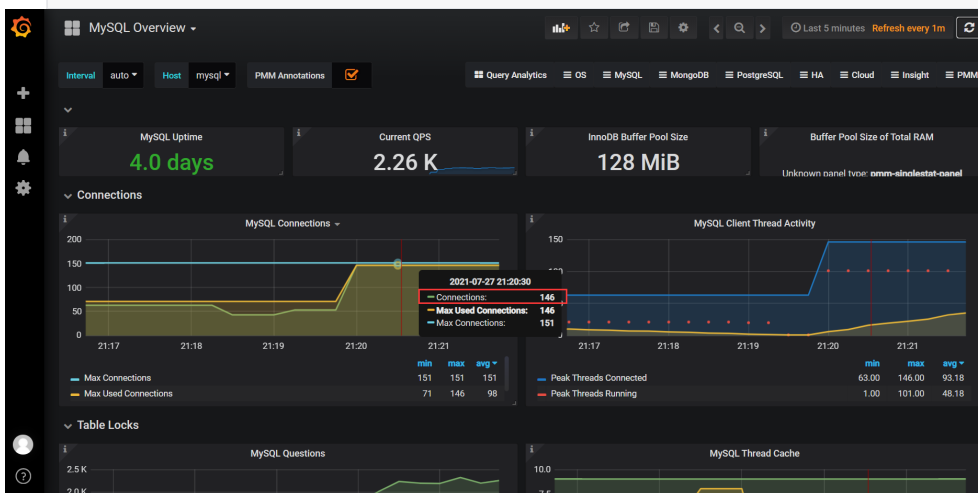


数据库连接倒是用上来了，不过mysql连接已经用满了，我们需要增大点mysql服务端的总连接数

```

1 set global max_connections=500; # 这样修改mysql重启后会恢复之前值，要彻底修改需要修改mysql的配置文件并重启

```



waiting的线程也减少了很多，等待数据库连接的线程已经几乎没有了，但是整体感觉线程都比较忙

```

[arthas@]# thread
Threads Total: 227, New: 0, RUNNABLE: 125, BLOCKED: 0, WAITING: 36, TIMED_WAITING: 51, TERMINATED: 0, Internal threads: 15

```

ID	NAME	GROUP	PRIORITY	STATE	%CPU	DELTA_TIME	TIME	INTERRUPTED	DAEMON
391	http-nio-8844-exec-158	main	5	RUNNABLE	3.44	0.006	0:5.805	false	true
356	http-nio-8844-exec-123	main	5	RUNNABLE	3.27	0.005	0:5.731	false	true
384	http-nio-8844-exec-151	main	5	RUNNABLE	3.0	0.006	0:5.913	false	true
430	http-nio-8844-exec-197	main	5	RUNNABLE	2.92	0.005	0:5.782	false	true
418	http-nio-8844-exec-185	main	5	RUNNABLE	2.74	0.005	0:5.821	false	true
364	http-nio-8844-exec-131	main	5	RUNNABLE	2.7	0.005	0:5.671	false	true
351	http-nio-8844-exec-128	main	5	RUNNABLE	2.69	0.005	0:5.808	false	true
24	grpc-nio-worker-ELG-1-1	main	5	RUNNABLE	2.55	0.005	0:5.960	false	true
344	http-nio-8844-exec-111	main	5	RUNNABLE	2.5	0.005	0:5.901	false	true
402	http-nio-8844-exec-169	main	5	RUNNABLE	2.47	0.005	0:5.890	false	true
342	http-nio-8844-exec-109	main	5	RUNNABLE	2.42	0.004	0:5.839	false	true
335	http-nio-8844-exec-152	main	5	RUNNABLE	2.41	0.004	0:5.765	false	true
424	http-nio-8844-exec-191	main	5	RUNNABLE	2.4	0.004	0:5.678	false	true
414	http-nio-8844-exec-181	main	5	RUNNABLE	2.34	0.004	0:5.809	false	true
380	http-nio-8844-exec-147	main	5	RUNNABLE	2.33	0.004	0:5.782	false	true
374	http-nio-8844-exec-141	main	5	RUNNABLE	2.26	0.004	0:5.668	false	true
407	http-nio-8844-exec-174	main	5	RUNNABLE	2.26	0.004	0:5.858	false	true
428	http-nio-8844-exec-195	main	5	TIMED_WAITING	2.23	0.004	0:5.689	false	true
426	http-nio-8844-exec-193	main	5	RUNNABLE	2.22	0.004	0:5.894	false	true
377	http-nio-8844-exec-144	main	5	RUNNABLE	2.21	0.004	0:5.914	false	true
421	http-nio-8844-exec-188	main	5	RUNNABLE	2.16	0.004	0:5.797	false	true
404	http-nio-8844-exec-171	main	5	RUNNABLE	2.16	0.004	0:5.705	false	true
77	lettuce-epollEventloop-4-1	main	5	RUNNABLE	2.11	0.004	0:12.361	false	true
353	http-nio-8844-exec-120	main	5	RUNNABLE	2.11	0.004	0:5.740	false	true
400	http-nio-8844-exec-167	main	5	RUNNABLE	2.09	0.004	0:5.762	false	true
413	http-nio-8844-exec-180	main	5	RUNNABLE	1.98	0.004	0:5.708	false	true
420	http-nio-8844-exec-189	main	5	RUNNABLE	1.92	0.003	0:5.764	false	true

我们试着调大tomcat的处理线程数再压测试下有没有效果

- server:
- port: 8844
- tomcat:
- accept-count: 200 # accept队列的长度, 当accept队列中连接的个数达到acceptCount时, 新进来的请求一律被拒绝。默认值是100
- threads:
- max: 300 # 线程池中最大活跃线程数, 默认值200
- min-spare: 20 # 线程池中保持的最小线程数
- max-connections: 1000 # Tomcat最多能并发处理的请求(连接)

重启order服务后再次压测发现下单qps几乎没什么变化, 注意, 线程数并不是越大越好的, 太大了会导致cpu争用, 让线程响应时间变长, 所以线程数具体设置多少合适, 需要通过压测具体的项目不断调整到较合理的值, 比如, 如果我们期望系统单台机器下单达到300并发, 最大线程数设置为100或300都能达到, 那我们就可以设置为100会更合适点。

既然加大tomcat线程数不行, 我们再回到skywalking看下下单接口的执行链路

Method	Start Time	Exec(ms)	Exec(%)	Self(ms)	API	Service
/order/generateOrder	2021-07-27 22:12:50	89	12		spring-we...	tulingmall-gateway
SpringCloudGateway/RoutingFilter	2021-07-27 22:12:50	77	0		spring-cloud-gat...	tulingmall-gateway
SpringCloudGateway/sendRequest	2021-07-27 22:12:50	77	0		spring-cloud-gat...	tulingmall-gateway
(POST)/order/generateOrder	2021-07-27 22:12:50	68	6		SpringMVC	tulingmall-order
Balancer/member/center/getMemberInfo	2021-07-27 22:12:50	3	0		Feign	tulingmall-order
/member/center/getMemberInfo	2021-07-27 22:12:50	3	3		Feign	tulingmall-order
MySQL/DB/PreparedStatement/execute	2021-07-27 22:12:50	1	1		mysql-connector...	tulingmall-order
Balancer/pms/getPromotionProductList	2021-07-27 22:12:50	9	0		Feign	tulingmall-order
/pms/getPromotionProductList	2021-07-27 22:12:50	9	1		Feign	tulingmall-order
(GET)/pms/getPromotionProductList	2021-07-27 22:12:50	8	2		SpringMVC	tulingmall-product
MySQL/DB/PreparedStatement/execute	2021-07-27 22:12:50	6	6		mysql-connector...	tulingmall-product
Balancer/stock/lockStock	2021-07-27 22:12:50	36	0		Feign	tulingmall-order

我们发现下单接口执行已经很快了, 不到100ms, 而且里面每步操作耗费的时间也很少
再次看下几个服务器的资源使用情况,

```

1 192.168.65.160 2 192.168.65.160 3 192.168.65.203 4 192.168.65.210 5 192.168.65.42
top - 22:41:55 up 2 days, 10:18, 1 user, load average: 13.27, 9.35, 5.55
Tasks: 282 total, 1 running, 281 sleeping, 0 stopped, 0 zombie
%Cpu(s): 14.8 us, 3.6 sy, 0.0 ni, 78.7 id, 0.0 wa, 0.0 hi, 2.9 si, 0.0 st
KiB Mem : 12136788 total, 3324160 free, 4529696 used, 4282932 buff/cache
KiB Swap: 0 total, 0 free, 0 used, 7206240 avail Mem

  PID USER      PR  NI  VIRT  RES  SHR  S  %CPU  %MEM    TIME+  COMMAND
 6504 root        20   0 9152156 1.5g 16164 S  98.0 12.6 25:05.72 java
24427 root        20   0 9002496 1.4g 16016 S  45.2 12.3 147:25.10 java
 1398 root        20   0 1071960 78360 8884 S   5.3  0.6 143:19.75 dockerd
 4230 root        20   0 1954008 47560 19084 S   3.3  0.4 106:19.00 calico-node
   832 root        20   0 2177244 65152 20552 S   3.0  0.5 117:00.52 kubelet
 6484 root        20   0 1088808 7880 3108 S   1.7  0.1  0:20.08 containerd-shim
 1196 root        20   0 1496640 57796 16320 S   1.0  0.5 27:19.54 containerd
 2958 33          20   0 136908 21948 6216 S   0.7  0.2 25:49.89 nginx-ingress-c
3221 gnome-i+  20   0 430912 21672 3736 S   0.7  0.2  8:28.21 gvfs-udisks2-vo
   9 root        20   0 0 0 0 S   0.3  0.0  5:05.84 rcu_sched
  24 root        20   0 0 0 0 S   0.3  0.0  6:04.20 ksoftirqd/3
  29 root        20   0 0 0 0 S   0.3  0.0  6:15.46 ksoftirqd/4
  315 root        20   0 67776 15836 1484 S   0.3  0.1 26:33.63 plymouthd
 3183 gnome-i+  20   0 463720 3972 3104 S   0.3  0.0 4:41.52 gsd-housekeepin
 3776 root        20   0 110216 10768 2600 S   0.3  0.1  5:26.04 containerd-shim
 4358 root        20   0 812 376 196 S   0.3  0.0  0:47.88 bird
10264 33          20   0 384372 60504 1860 S   0.3  0.5  0:01.19 nginx
10268 33          20   0 384372 60492 1856 S   0.3  0.5  0:01.19 nginx
31205 root        20   0 0 0 0 S   0.3  0.0  0:16.65 kworker/3:2
   1 root        20   0 191568 4092 2192 S   0.0  0.0  1:57.64 systemd
  20 root        20   0 0 0 0 S   0.0  0.0  0:00.13 kthreadd

1 192.168.65.160 2 192.168.65.160 3 192.168.65.203 4 192.168.65.210 5 192.168.65.42
top - 22:42:54 up 2 days, 4:35, 1 user, load average: 0.58, 1.17, 1.12
Tasks: 285 total, 1 running, 284 sleeping, 0 stopped, 0 zombie
%Cpu(s): 8.5 us, 3.6 sy, 0.0 ni, 86.3 id, 0.0 wa, 0.0 hi, 1.5 si, 0.0 st
KiB Mem : 12136788 total, 322784 free, 4739984 used, 7074020 buff/cache
KiB Swap: 0 total, 0 free, 0 used, 6864484 avail Mem

  PID USER      PR  NI  VIRT  RES  SHR  S  %CPU  %MEM    TIME+  COMMAND
10097 root        20   0 9029464 913648 16116 S  55.8  7.5 21:31.52 java
25295 root        20   0 8954984 1.0g 8644 S  18.3  9.0 42:43.90 java
 1518 root        20   0 1112876 72860 7188 S  11.3  0.6 136:46.69 dockerd
10080 root        20   0 1088808 3996 3024 S   5.0  0.0  1:43.22 containerd-shim
 3368 root        20   0 1953752 47340 18852 S   3.0  0.4 95:31.99 calico-node
  797 root        20   0 1955792 63652 17944 S   2.7  0.5 104:26.77 kubelet
 3833 root        20   0 1088808 8480 2028 S   2.0  0.1 29:44.21 containerd-shim
10195 33          20   0 384628 60904 2044 S   2.0  0.5  0:27.05 nginx
10193 33          20   0 384628 60936 2072 S   1.7  0.5  0:27.69 nginx
10194 33          20   0 384756 61020 2044 S   1.7  0.5  0:27.40 nginx
10195 33          20   0 384628 60904 2044 S   1.7  0.5  0:27.05 nginx

1 192.168.65.160 2 192.168.65.160 3 192.168.65.203 4 192.168.65.210 5 192.168.65.42
top - 22:43:20 up 9 days, 3:26, 1 user, load average: 11.09, 9.67, 6.13
Tasks: 332 total, 1 running, 331 sleeping, 0 stopped, 0 zombie
%Cpu(s): 36.1 us, 8.9 sy, 0.0 ni, 53.4 id, 0.3 wa, 0.0 hi, 1.2 si, 0.2 st
KiB Mem : 24521464 total, 3462408 free, 14619496 used, 6439560 buff/cache
KiB Swap: 12386300 total, 12382460 free, 3840 used, 9493428 avail Mem

  PID USER      PR  NI  VIRT  RES  SHR  S  %CPU  %MEM    TIME+  COMMAND
16421 ods        20   0 6859676 619820 13236 S  565.4  2.5 5521:35 mysqld
17867 root        20   0 12.0g  5.6g 14960 S   5.3 23.8 199:16.95 java
16973 root        20   0 9003532 944572 13652 S   1.7  3.9 65:43.58 java
18226 ods        20   0 7503436 105388 4984 S   1.3  0.4 52:39.44 beam.smp
  343 root        20   0 70644 18516 1468 S   1.0  0.1 95:13.57 plymouthd
16414 ods        20   0 278264 72396 8924 S   1.0  0.3 44:24.66 mongod
  934 root        20  -20 0 0 0 S   0.7  0.0  1:26.52 kworker/6:1H
16079 1000       20   0 8040440 927376 14836 S   0.7  3.8 54:14.54 java
16298 ods        20   0 61688 15264 1836 S   0.7  0.1 14:24.91 redis-server
16976 1000       20   0 1352948 203864 13556 S   0.7  0.8 30:36.16 node
23367 root        20   0 164316 2532 1620 R   0.7  0.0  0:02.66 top
   9 root        20   0 0 0 0 S   0.3  0.0 11:44.76 rcu_sched
  20 root        20   0 0 0 0 S   0.3  0.0  1:20.26 ksoftirqd/4

```

我们发现我们调整完各种连接池之后，虽然qps没有增加，但是应用的cpu使用率明显降下来了，之前都是百分之两三百，但是mysql的cpu使用率依然很高，说明，mysql这边压力依然很大，但是创建订单接口链路里的sql执行都挺快，我们来看下mysql的慢查询有没有我们没有注意到的sql，慢查询如何监测可以参考mysql课程，如下是一些关键sql

```

1 -- 慢查询
2 show variables like '%slow_query_log%'; -- 查看是否开启慢查询
3 set global slow_query_log=1; -- 开启慢查询
4
5 show variables like 'long_query_time%'; -- 查看慢查询阈值
6 set global long_query_time=0.1; -- 设置慢查询阈值(单位s)，设置完需要重新开启session才能查看生效
7 show global variables like 'long_query_time'; -- 查看慢查询阈值
8
9 show variables like '%log_output%'; -- 查看慢查询写入文件还是数据表
10 set global log_output='TABLE'; -- 设置慢查询写入数据表，默认是写入文件

```

```

11
12 select sleep(1); -- 执行一条慢查询
13
14 select * from mysql.slow_log; -- 查看慢查询记录表
15 TRUNCATE mysql.slow_log;

```

通过慢查询我们发现竟然还有不少sql执行时间超过100ms的

```

26 select * from mysql.slow_log; -- 查看慢查询记录表
27 TRUNCATE mysql.slow_log;
28
29 show variables like "max_connections*"; -- 查看mysql最大连接数
30 set global max_connections=500;

```

信息	结果1	概况	状态
start_time	user_host	query_time	lock_time
2021-07-27 14:47:37.6964	root[root]@	00:00:00.246553	00:00:00.000065
2021-07-27 14:47:37.6964	root[root]@	00:00:00.233029	00:00:00.000044
2021-07-27 14:47:37.7369	root[root]@	00:00:00.273992	00:00:00.000070
2021-07-27 14:47:37.7380	root[root]@	00:00:00.274036	00:00:00.000047
2021-07-27 14:47:37.7431	root[root]@	00:00:00.280772	00:00:00.000057
2021-07-27 14:47:38.5025	root[root]@	00:00:00.996998	00:00:00.000089
2021-07-27 14:47:38.5043	root[root]@	00:00:01.056703	00:00:00.000090
2021-07-27 14:47:38.5080	root[root]@	00:00:01.028060	00:00:00.000093
2021-07-27 14:47:38.5261	root[root]@	00:00:01.018907	00:00:00.000094

我们看了一些执行时间较长的sql，发现基本都是下面这条sql，那我们就来优化下

查询创建工具 查询编辑器

```

1 EXPLAIN SELECT
2 id,
3 product_id,
4 product_sku_id,
5 member_id,
6 quantity,
7 price,
8 sp1,
9 sp2,
10 sp3,
11 product_pic,
12 product_name,
13 product_sub_title,
14 product_sku_code,
15 member_nickname,
16 create_date,
17 modify_date,
18 delete_status,
19 product_category_id,
20 product_brand,
21 product_sn,
22 product_attr
23 FROM
24 oms_cart_item
25 WHERE
26 (
27 delete_status = 0
28 AND member_id = 453
29 )

```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	oms_cart	(Null)	ref	idx_member_id_and_product_id	idx_member_id_and_product_id	9	const	1453		10 Using where

我们发现这条sql走了索引，但是扫描数据有一千多条，更奇怪的是当我们执行这条sql的时候发现查询的结果竟然为空。

查询创建工具 查询编辑器

```

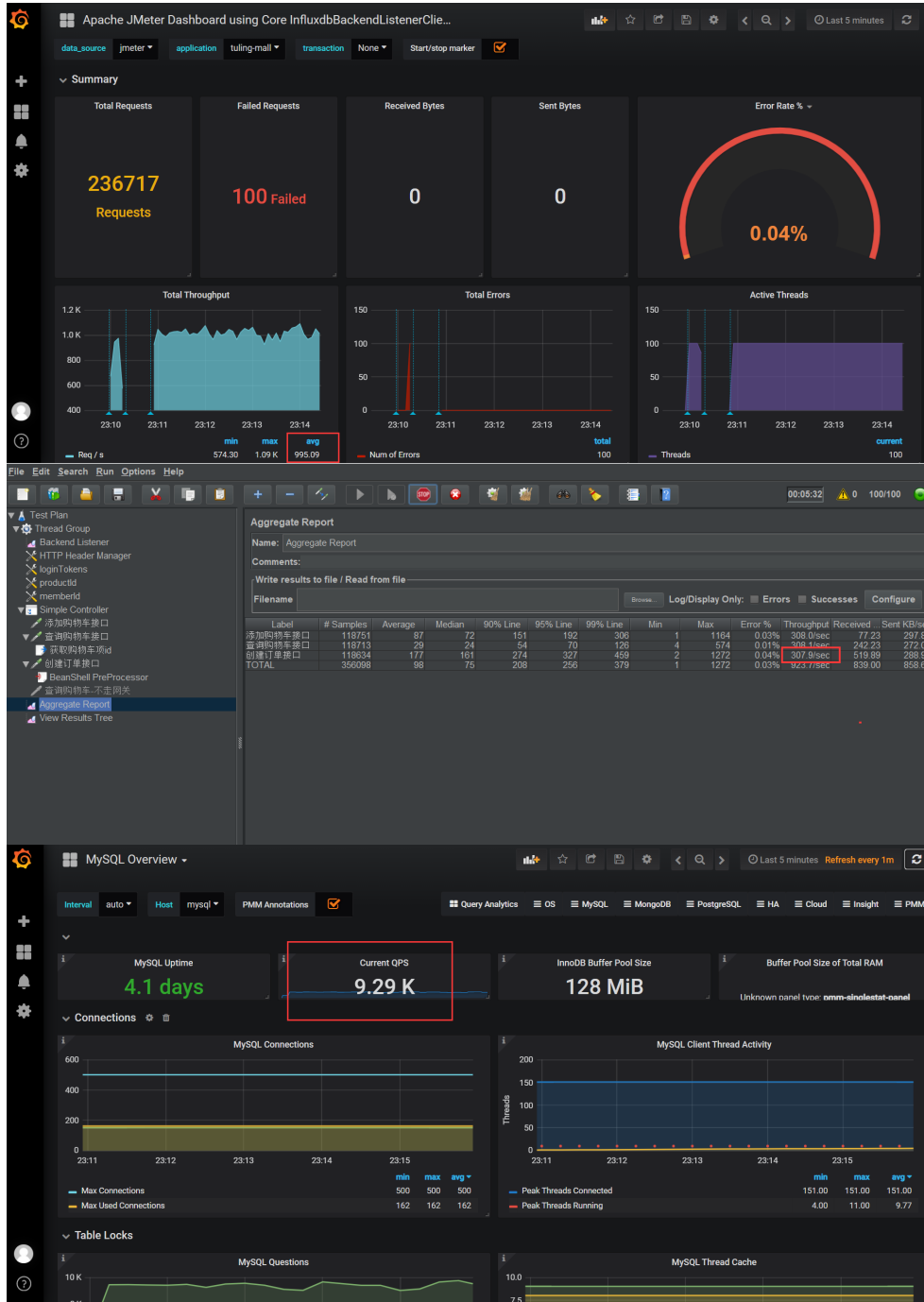
1 SELECT
2 id,
3 product_id,
4 product_sku_id,
5 member_id,
6 quantity,
7 price,
8 sp1,
9 sp2,
10 sp3,
11 product_pic,
12 product_name,
13 product_sub_title,
14 product_sku_code,
15 member_nickname,
16 create_date,
17 modify_date,
18 delete_status,
19 product_category_id,
20 product_brand,
21 product_sn,
22 product_attr
23 FROM
24 oms_cart_item
25 WHERE
26 (
27 delete_status = 0
28 AND member_id = 453
29 )

```

id	product_id	product_sku_id	member_id	quantity	price	sp1	sp2	sp3	product_pic	product_name	product_sub_title	product_sku_code	member_nickname
(Null)	(Null)	(Null)	(Null)	(Null)	(Null)	(Null)	(Null)	(Null)	(Null)	(Null)	(Null)	(Null)	(Null)

而且这样的慢sql还挺多，并且查询结果都是空，那我们就需要结合业务代码分析下这个sql是怎么回事了，原来目前系统里实现的生成订单业务在生成完订单后并没有删除购物车记录，而且修改了记录的删除状态delete_status的值为1，所以这条sql是查询的用户购物车里的还没生成的购物项数据，可能后面要生成订单了，因为一个用户可能买过很多商品，所以这个用户对应的购物项数据表里可能有很多delete_status值为1的数据，虽然上面走了一个索引，但是key_len的值我们可以发现只走了member_id的索引，所以每次会扫描出几千条数据，我们之前在mysql课里讲过对于delete_status的字段，只有两个值，一般没必要建索引，但是这里是可以尝试建下索引的，因为表里的delete_status字段的值大多数都是1，而我们查询的sql更多是用0来过滤的，所以理论上建完索引，会大大减少mysql的查询扫描次数，二话不说，我们在oms_cart_item表里再建一个联合索引(member_id, delete_status)，

其实，仔细分析下oms_cart_item表，其实里面的两个联合索引是可以合并成一个的(member_id, delete_status, product_id)，重新再压测看下效果



```

top - 23:15:36 up 9 days, 3:59, 1 user, load average: 3.08, 2.22, 2.57
Tasks: 332 total, 1 running, 331 sleeping, 0 stopped, 0 zombie
%cpu(s): 12.9 us, 6.3 sy, 0.0 ni, 74.8 id, 1.6 wa, 0.0 hi, 4.0 si, 0.3 st
KiB Mem : 24521464 total, 3458712 free, 14630468 used, 6432284 buff/cache
KiB Swap: 12386300 total, 12382460 free, 3840 used, 9482456 avail Mem

  PID USER      PR  NI   VIRT   RES   SHR  S  %CPU  %MEM     TIME+ COMMAND
16421 ods        20   0 6859676 624800 13236 S 266.9  2.5   5579:15 mysqlld
17867 root        20   0  12.0g   5.6g  14960 S   5.8  25.8 200:30.91 java
16298 ods        20   0  61688  15276  1836 S   2.3  0.1  14:38.86 redis-server
 934 root        0 -20   0       0       0 S   2.0  0.0   1:37.55 kworker/6:1H
30874 nfsnobo+  20   0 720444 30604  6004 S   2.0  0.1  51:36.65 node_exporter
16973 root        20   0 9003532 944576 13652 S   1.7  3.9  66:07.79 java
  34 root        20   0   0       0       0 S   1.0  0.0   1:36.90 ksoftirqd/5
16414 ods        20   0 278264 72396  8924 S   1.0  0.3  44:41.73 mongod
18226 ods        20   0 7503436 105388 4984 S   1.0  0.4  52:58.14 beam.smp
  19 root        20   0   0       0       0 S   0.7  0.0   1:38.00 ksoftirqd/2
  24 root        20   0   0       0       0 S   0.7  0.0   1:30.18 ksoftirqd/3
  29 root        20   0   0       0       0 S   0.7  0.0   1:32.48 ksoftirqd/4
  44 root        20   0   0       0       0 S   0.7  0.0   1:46.26 ksoftirqd/7
  54 root        20   0   0       0       0 S   0.7  0.0   1:26.72 ksoftirqd/9
  64 root        20   0   0       0       0 S   0.7  0.0   1:20.42 ksoftirqd/11
 343 root        20   0  70644  18516  1468 S   0.7  0.1  95:28.31 plymouthd
 877 root        20   0  90568   3180  2320 S   0.7  0.0   5:26.88 rngd
16976 1000       20   0 1352948 204308 13556 S   0.7  0.8  30:47.64 node
17659 root        20   0  10.0g   1.9g  13868 S   0.7  8.1  15:06.25 java
21107 root        20   0   0       0       0 S   0.7  0.0   0:18.83 kworker/6:2
   6 root        20   0   0       0       0 S   0.3  0.0   1:39.00 ksoftirqd/0
   9 root        20   0   0       0       0 S   0.3  0.0  11:47.35 rcu_sched
  14 root        20   0   0       0       0 S   0.3  0.0   1:28.77 ksoftirqd/1

```

我们看到优化效果非常明显，下单接口qps超过了300，mysql的cpu使用率明显降低了很多，并且mysql的qps快上万了，再看下mysql的慢查询日志表里也没有再产生新的慢查询了，目前购物车与订单相关表里的数据都已经过百万了，对于这种复杂的接口在这个数据量级和目前这个配置的机器上差不多也就这个水平了当然，性能优化是无止尽的，我们要权衡优化的代价和给我们系统性能带来的提升是否相匹配！

对于JVM内部的一些内存问题其实还需要长时间的压测，比如一些内存泄漏之类的问题，一般这种短时间的压测是比较难发现的，这块大家可以在课后试着长时间的压测下，看下系统中是不是有些内存泄漏的问题，结合我们JVM课程里讲的方法去调优下，大家可以参考下面JVM参数配置

JVM配置模板

如果内存不大，比如4核8G的机器，可以用默认的Parallel垃圾收集器，如果对停顿时间有一定要求，Jdk 1.8版本可以使用ParNew+CMS垃圾收集器组合，比如下面配置(参考JVM调优课程)，如果是大内存的服务，对单机并发要求非常高，那么一般可以用G1垃圾收集器了

```

1 -Xms3072M -Xmx3072M -Xmn1536M -Xss1M -XX:MetaspaceSize=256M -XX:MaxMetaspaceSize=256M -XX:SurvivorRatio=6
2 -XX:MaxTenuringThreshold=5 -XX:PretenureSizeThreshold=1M -XX:+UseParNewGC -XX:+UseConcMarkSweepGC
3 -XX:CMSInitiatingOccupancyFraction=92 -XX:+UseCMSCompactAtFullCollection -XX:CMSFullGCsBeforeCompaction=0

```

优化结论:

- 1、我们发现大多数时候我们系统的瓶颈往往出现在数据库上，所以我们秒杀课里的优化方案是尽可能的让各种操作被缓存以及其它各种中间件拦截，让他们尽量少的到达mysql数据库。
- 2、我们之前在mysql系列课里讲过的尽量不要搞太多表关联的sql查询，因为不好优化索引，所以我们建议对于一些多表的操作能用java做的可以尽量用java做，哪怕java实现可能费时间更多点，但是java应用扩容是很方便的，数据库扩容是比较麻烦的。

批量删除deployment

```
1 kubectl get deployment | awk '{print $1}' | xargs kubectl delete deployment
```

批量删除Evicted状态的pod

```
1 kubectl get pods | grep Evicted | awk '{print $1}' | xargs kubectl delete pod
```

1 文档: 电商项目性能优化实战.note

2 链接: <http://note.youdao.com/noteshare?id=63f923a3eb6cfd6628c2f9b1c10d81e5&sub=1E3E031CFD6941A88423DBCF7AAD79E1>