Redis 缓存的效率

摘　要: 本实验旨在探讨 Redis 缓存在提升系统性能方面的作用。通过构建一个基于 Java 的 Web 应用，实现了对数据库查询操作的缓存处理。实验设计包括两个主要部分：首先，实现了一个标准的基于 JPA 的 RESTful API 接口来查询数据库；其次，在此基础上加入了 Redis 缓存机制以减少对后端数据库的直接访问次数。实验结果显示，使用 Redis 缓存能够显著提高 API 响应速度并降低服务器负载。

关键词: Redis 缓存；系统性能优化；RESTful API；JPA 查询；JMeter 压力测试；MySQL 数据库

问题描述：

随着互联网应用的普及和发展，用户基数不断增大，高并发访问逐渐成为一种常态。在这种背景下，后端服务面临着前所未有的挑战，特别是对于数据库的操作，频繁的读写请求不仅消耗了大量的计算资源，还可能引发数据库的性能瓶颈，导致响应时间增加、用户体验下降等问题。为了应对这一挑战，业界普遍采用缓存技术来减轻数据库的压力，提高系统的整体性能。其中，Redis 作为一种高性能的键值存储系统，因其支持多种数据结构、丰富的特性以及高效的读写速度而被广泛应用于缓存解决方案中。

本实验针对上述背景，设计并实现了一个简单的查询数据库后端服务，该服务提供了一个用于查询商品详细信息的 RESTful API 接口。在实验的第一阶段，我们基于 Java Persistence API (JPA)实现了对 MySQL 数据库的直接查询功能。然而，考虑到实际应用场景中可能存在大量重复的数据请求，这样的设计在高并发情况下可能会导致数据库过载。因此，在第二阶段，我们在原有的基础上集成了 Redis 缓存机制，用于存储那些经常被查询但变化不频繁的商品数据。通过这种方式，可以大幅度减少对数据库的直接访问次数，从而达到提升系统性能的目的。

实验设计：

1. 实验环境

- 服务器 A: Ubuntu 18.04 服务器 2 核 4G 内存虚拟机一台，图形界面，安装 JDK 17， Maven、git，Redis 6.2.4
- 服务器 B：Ubuntu 18.04 服务器 2 核 2G 内存虚拟机一台，命令行界面，安装 JDK 17，Maven、git，JMeter 5.4.1
- 服务器 C：Ubuntu 18.04 服务器 2 核 2G 内存虚拟机一台，命令行界面，安装 JDK 17，Maven、git，MySQL 9.0

2. 实验步骤

(1) 实现了一个基于 JPA 查询数据库的 RESTful API 接口

a. GET /products?name=xxxx 通过商品名称查询查询产品完整信息

(2) 在基于 JPA 查询数据库的 RESTful API 接口中实现基于 Redis 查询数据的缓存

(3) 将编写好的代码打包成 docker 镜像，部署到 OOMALL-node1 服务器上

(4) 使用华为云云监控服务 CES 监控了 MySQL 数据库服务器和 OOMALL-node1 服务器的 CPU 使用率、内存使用率和运行中的进程数

(5) 分别对没有使用 Redis 缓存的接口和使用了 Redis 缓存的接口进行测试，并对比两种方法速度差异与服务器负载

结果分析与讨论：

1.代码编写

在实验 5 实现的 JPA 查询基础上，增加 Redis 查询，在 ProductService 的实现类中定义 RedisTemplate 用于操控 redis 中的数据

```
@Autowired
private RedisTemplate<String, Object> redisTemplate;

@Override
public List<Product> getProductsByNameAndRelatedData(String name) {
    String productCacheKey = "product:" + name;
    String onSalePoCacheKey = "onSalePo:" + name;
    String allProductsKey = "allProducts:" + name;
```

对最终查到的结果（Product 的全部信息）进行缓存，如果缓存没有命中，就查询数据库获取 ProductPo 对象

```
@Override
public List<Product> getProductsByNameAndRelatedData(String name) {
    String productCacheKey = "product:" + name;
    String onSalePoCacheKey = "onSalePo:" + name;
    String allProductsKey = "allProducts:" + name;

    // 从缓存中获取数据
    List<Product> cachedProduct = (List<Product>) redisTemplate.opsForValue().get(productCacheKey);
    if (cachedProduct != null) {
        log.info("Cache hit for product: " + productCacheKey);
        return cachedProduct;
    }

    // 如果缓存中没有数据，查询数据库
    ProductPo productPo = productRepository.findByName(name);
    if (productPo == null) {
        throw new RuntimeException("Product not found with name: " + name);
    }
```

然后查询对应 product 相关的销售 onsalePo，同样先查询 redis，如果不存在就查询数据库，并将查询结果写入到 redis 中，最终还要使用 cloneFactory 将 onsalePo 对象转换为 onsale 对象

```
// 先查 redis
List<OnSalePo> cachedOnSalePos = (List<OnSalePo>) redisTemplate.opsForValue().get(onSalePoCacheKey);
if (cachedOnSalePos != null) {
    log.info("Cache hit for onsale: " + onSalePoCacheKey);
} else {
    // redis 不存在，再查数据库
    cachedOnSalePos = onSaleRepository.findByProductId(productPo.getId());

    // 添加到 redis 中
    redisTemplate.opsForValue().set(onSalePoCacheKey, cachedOnSalePos, timeout: 1, TimeUnit.HOURS);
}

List<OnSale> onSales = new ArrayList<>();
for (OnSalePo cachedOnSalePo : cachedOnSalePos) {
    onSales.add(CloneFactory.copy(new OnSale(), cachedOnSalePo));
}
```

对于查询产品相关的其他产品进行相同的操作，先查 redis，不存在再查询数据库并写入 redis，最后利用 cloneFactory 将 productPo 对象转换为 product 对象

```
// 查询相关的其他产品
List<ProductPo> allProducts = (List<ProductPo>) redisTemplate.opsForValue().get(allProductsKey);
if (allProducts != null) {
    log.info("Cache hit for allProducts: " + allProductsKey);
} else {
    // redis 中不存在，查数据库
    allProducts = productRepository.findByGoodsId(productPo.getGoodsId());
    // 存到 redis 中
    redisTemplate.opsForValue().set(allProductsKey, allProducts, timeout: 1, TimeUnit.HOURS);
}

List<Product> otherProduct = new ArrayList<>();

// 转换 ProductPo 到 Product
Product product = CloneFactory.copy(new Product(), productPo);
// 添加其他 product
for (ProductPo allProduct : allProducts) {
    if (!name.equals(allProduct.getName())) {
        otherProduct.add(CloneFactory.copy(new Product(), allProduct));
    }
}
```

最后将销售 onsale 和其他产品 otherProduct 放到查询的 product 的属性中，然后将最终的数据写入缓存，便于下次再次进行查询能够命中缓存，然后返回，业务流程结束

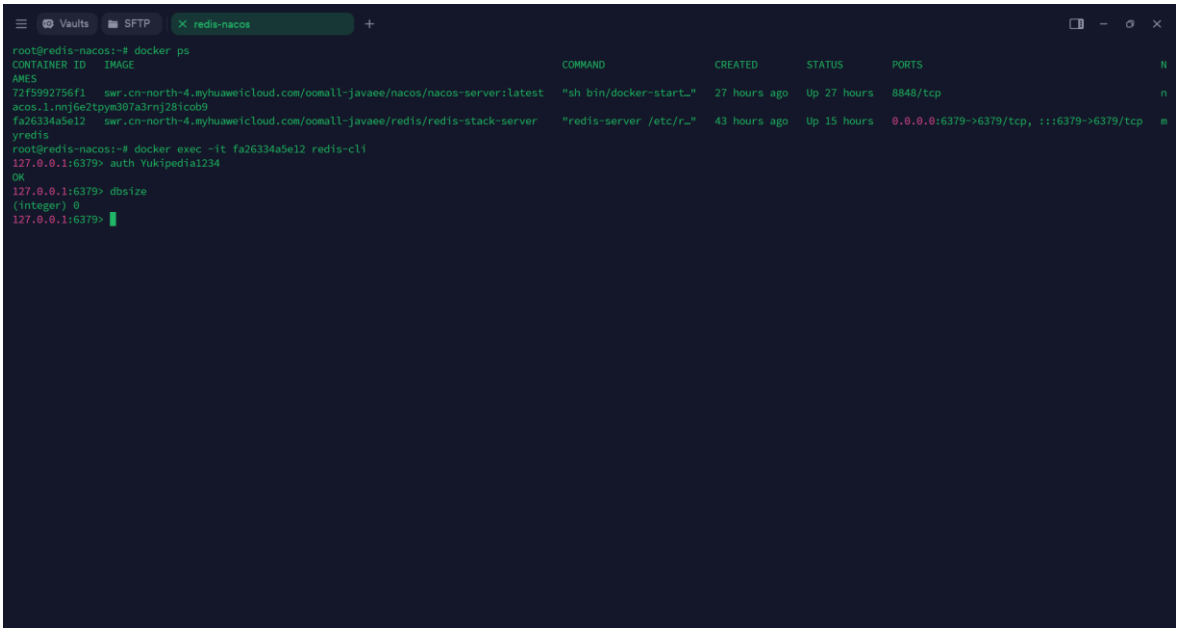
```
    product.setOnSaleList(onSales);
    product.setOtherProduct(otherProduct);

    // 最终数据存到 redis
    List<Product> productList = new ArrayList<>();
    productList.add(product);
    redisTemplate.opsForValue().set(productCacheKey, productList, timeout: 1, TimeUnit.HOURS);

    // 返回最终的数据
    return productList;
}
```

另外，还需要配置一个 RedisConfig 配置类，设置部署 redis 的服务器 ip 地址和密码，以及需要用到的 redisTemplate。其中，”configuration.setHostName”的 redis 已经映射到本地的 host 文件中，redis 映射的就是部署有 redis 的服务器

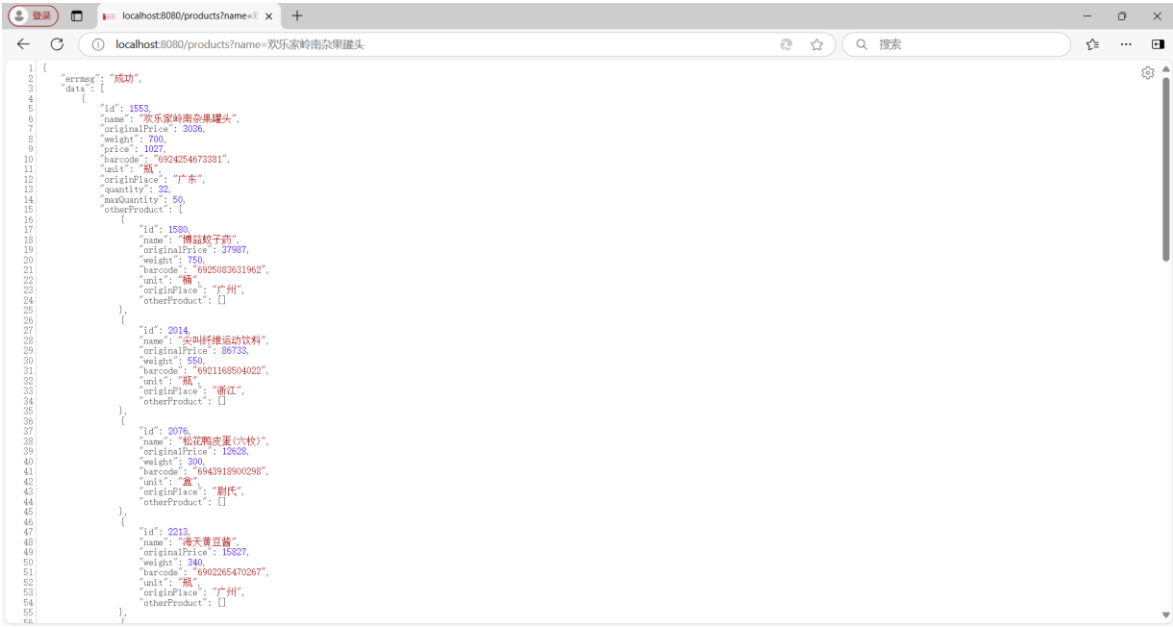
```
@Configuration
public class RedisConfig {

    // 配置 redis 连接
    @Bean
    public RedisConnectionFactory redisConnectionFactory() {
        RedisStandaloneConfiguration configuration = new RedisStandaloneConfiguration();
        configuration.setHostName("redis");
        configuration.setPort(6379);
        configuration.setPassword(RedisPassword.of( passwordAsString: "Yukipedia1234"));  // 设置密码
        return new LettuceConnectionFactory(configuration);
    }

    // 配置 RedisTemplate
    @Bean
    public RedisTemplate<String, Object> redisTemplate(RedisConnectionFactory redisConnectionFactory) {
        RedisTemplate<String, Object> redisTemplate = new RedisTemplate<>();
        redisTemplate.setConnectionFactory(redisConnectionFactory);
        return redisTemplate;
    }

}
```

2.本地测试

代码编写完成后，可以进行本地测试，初始时 redis 并没有任何缓存，下图中可以看到 dbsize 命令返回结果为 0



启动项目进行查询：

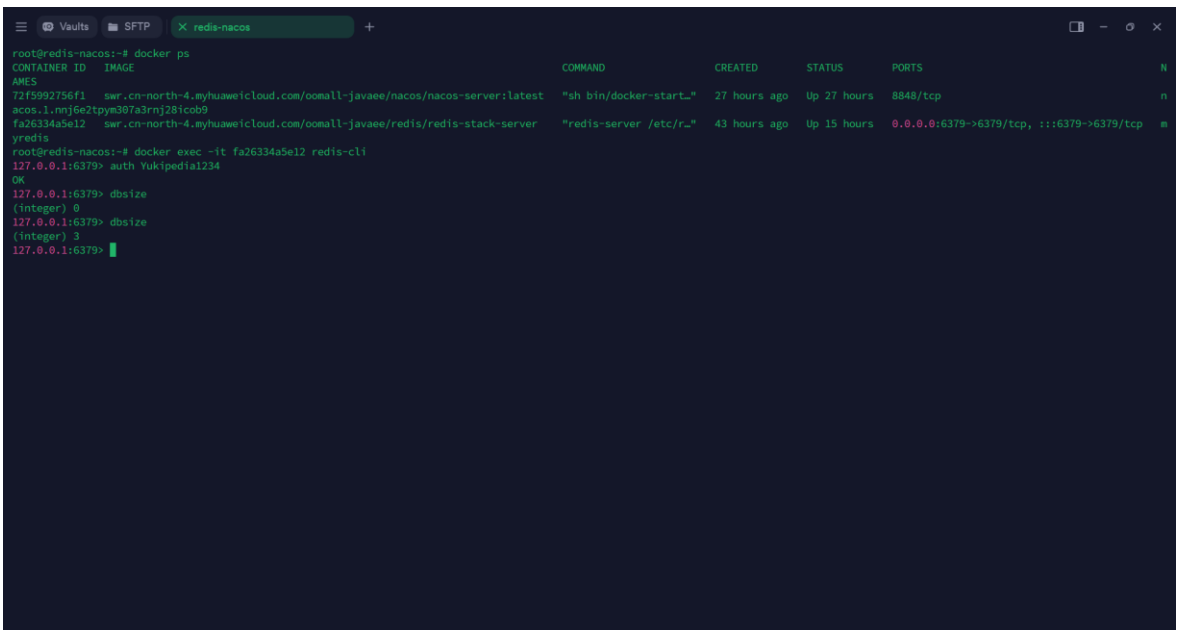



```
{
  "errmsg": "成功",
  "data": [
    {
      "id": 1553,
      "name": "欢乐家岭南杂果罐头",
      "originalPrice": 3036,
      "weight": 700,
      "price": 1027,
      "barcode": "6924254673381",
      "unit": "瓶",
      "originPlace": "广东",
      "quantity": 22,
      "maxQuantity": 50,
      "otherProduct": [
        {
          "id": 1580,
          "name": "博益蛟子药",
          "originalPrice": 37987,
          "weight": 750,
          "barcode": "6925083631962",
          "unit": "箱",
          "originPlace": "广州",
          "otherProduct": []
        },
        {
          "id": 2014,
          "name": "尖叫纤维运动饮料",
          "originalPrice": 86733,
          "weight": 550,
          "barcode": "6921168504022",
          "unit": "瓶",
          "originPlace": "浙江",
          "otherProduct": []
        },
        {
          "id": 2076,
          "name": "松花鸭皮蛋(六枚)",
          "originalPrice": 12628,
          "weight": 300,
          "barcode": "6943918900298",
          "unit": "盒",
          "originPlace": "尉氏",
          "otherProduct": []
        },
        {
          "id": 2213,
          "name": "海天黄豆酱",
          "originalPrice": 15827,
          "weight": 340,
          "barcode": "6902265470267",
          "unit": "瓶",
          "originPlace": "广州",
          "otherProduct": []
        },
```


控制台输出可以看出并没有任何缓存命中

```
2024-11-30T13:35:17.646+08:00  INFO 15452 --- [  restartedMain] o.h.e.t.j.p.i.JtaPlatformInitiator       : HHH000489: No JTA platform available (set 'hibernate.transaction.jta
2024-11-30T13:35:17.656+08:00  INFO 15452 --- [  restartedMain] j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory for persistence unit 'default'
2024-11-30T13:35:18.798+08:00  WARN 15452 --- [  restartedMain] JpaBaseConfiguration$JpaWebConfiguration : spring.jpa.open-in-view is enabled by default. Therefore, database q
2024-11-30T13:35:19.654+08:00  INFO 15452 --- [  restartedMain] o.s.b.d.a.OptionalLiveReloadServer       : LiveReload server is running on port 35729
2024-11-30T13:35:19.743+08:00  INFO 15452 --- [  restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port 8080 (http) with context path '/'
2024-11-30T13:35:19.761+08:00  INFO 15452 --- [  restartedMain] c.xmu.exp6_Redis.Exp6RedisApplication    : Started Exp6RedisApplication in 13.898 seconds (process running for
2024-11-30T13:35:44.404+08:00  INFO 15452 --- [nio-8080-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring DispatcherServlet 'dispatcherServlet'
2024-11-30T13:35:44.405+08:00  INFO 15452 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet        : Initializing Servlet 'dispatcherServlet'
2024-11-30T13:35:44.408+08:00  INFO 15452 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet        : Completed initialization in 3 ms
```

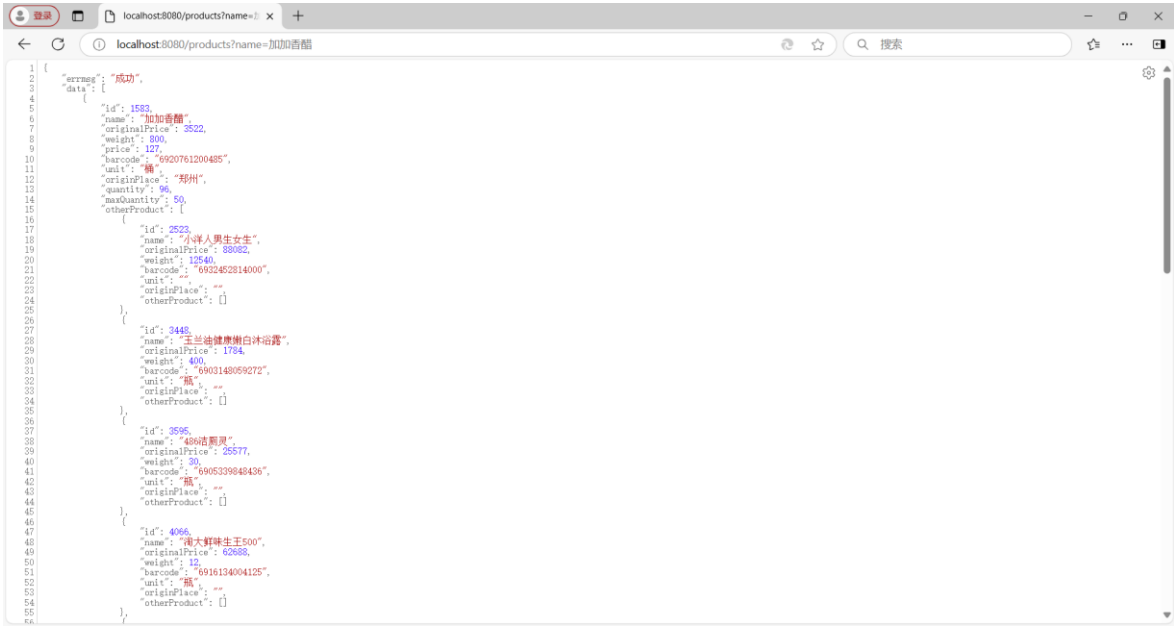
这时候我们再去查看 redis 数据库中是否有数据，可以看出我们已经把查到的 product、onsale 和 otherproduct 数据写入到了 redis 缓存中，然后我们再进行一次查询。



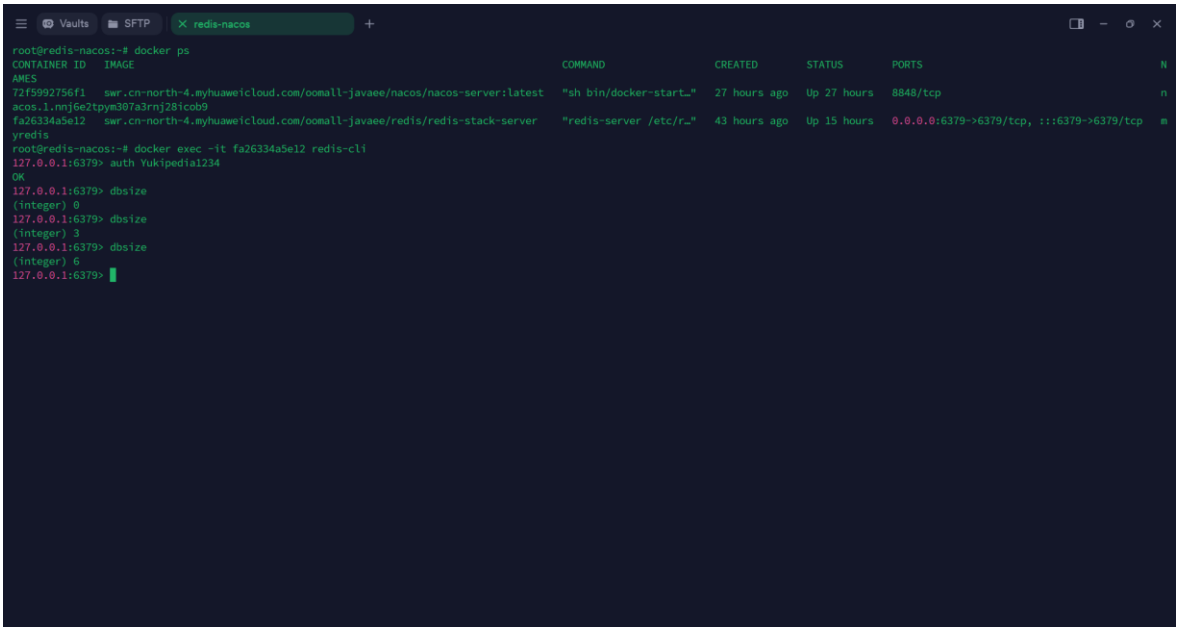
```
root@redis-nacos:~# docker ps
CONTAINER ID   IMAGE                                                  COMMAND                  CREATED        STATUS        PORTS                                       N
AMES
72f5992756f1   swr.cn-north-4.myhuaweicloud.com/oomall-javaee/nacos/nacos-server:latest   "sh bin/docker-start…"   27 hours ago   Up 27 hours   8848/tcp                                    n
acos.1.nnj6e2tpym307a3rnj28icob9
fa26334a5e12   swr.cn-north-4.myhuaweicloud.com/oomall-javaee/redis/redis-stack-server   "redis-server /etc/r…"   43 hours ago   Up 15 hours   0.0.0.0:6379->6379/tcp, :::6379->6379/tcp   m
yredis
root@redis-nacos:~# docker exec -it fa26334a5e12 redis-cli
127.0.0.1:6379> auth Yukipedia1234
OK
127.0.0.1:6379> dbsize
(integer) 0
127.0.0.1:6379> dbsize
(integer) 3
127.0.0.1:6379>
```

控制台日志输出已经命中缓存，命中的是 name 为“欢乐家岭南杂果罐头”的 product 的所有数据

然后我们换一个产品测试，再看控制台输出



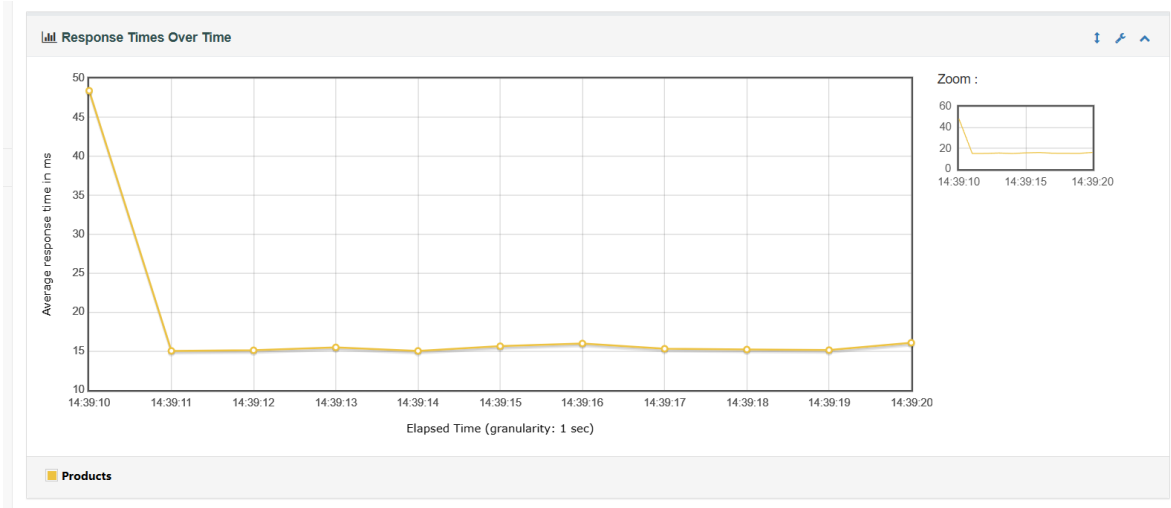
并没有缓存命中的日志记录，然后我们再看 redis 数据库数据个数，可以看到已经有了 6 个缓存数据，说明此次缓存也成功写入。然后再查一次就会有缓存命中日志输出。



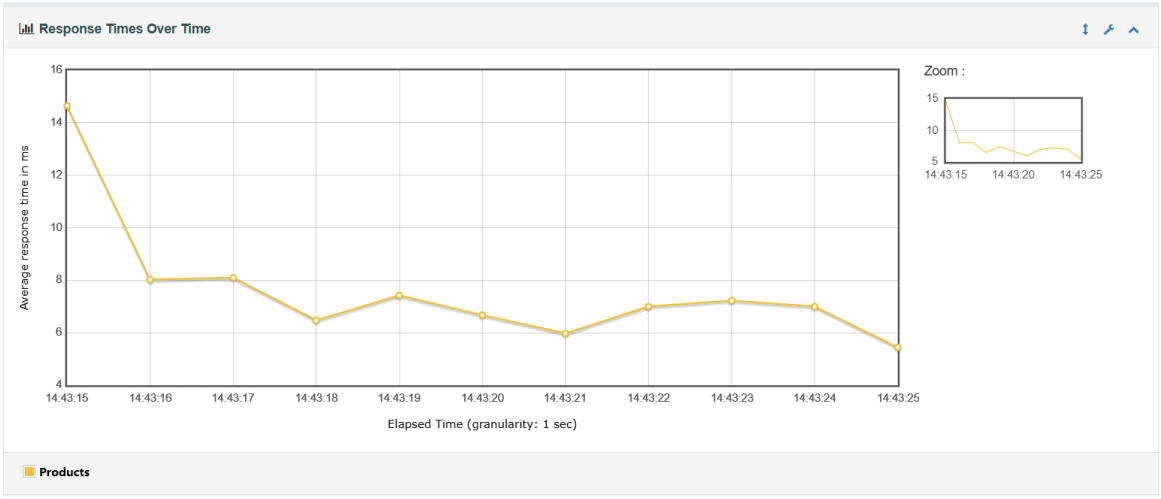
3.本地测试没有问题，我们将代码部署到服务器后再使用 jmeter 进行测试，并通过华为云云监控服务进行监控，主要监控指标为内存使用率、CPU 使用率、阻塞进程数和 1 分钟平均负载情况

(1).测试数据：thread：400、rampTime：10、loop：1

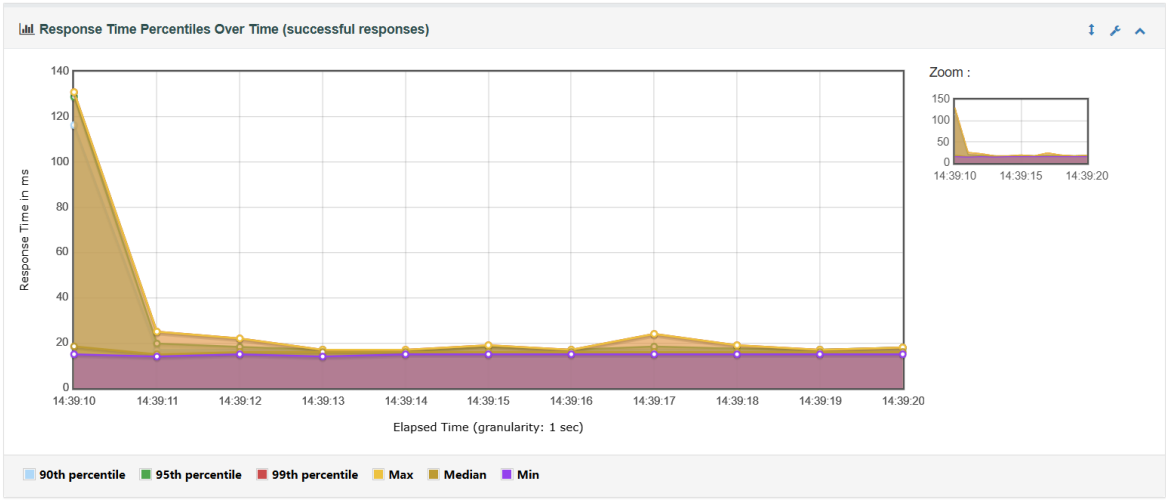
未使用 redis 接口的响应时间，可以看出平均在 15ms 上下



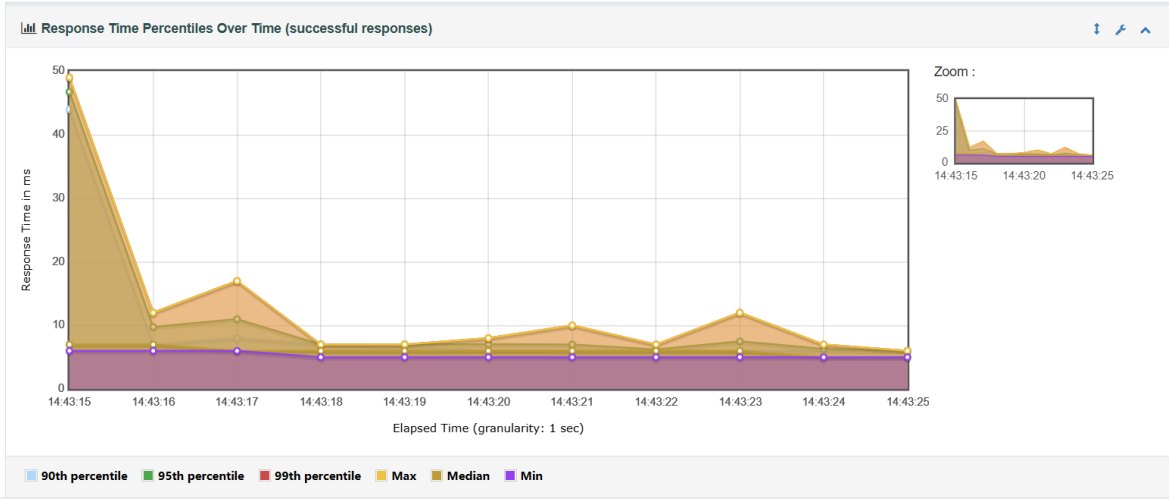
下面是使用 redis 的响应时间，可以看出基本都在 8ms 以下，显然使用 redis 后响应时间变快了



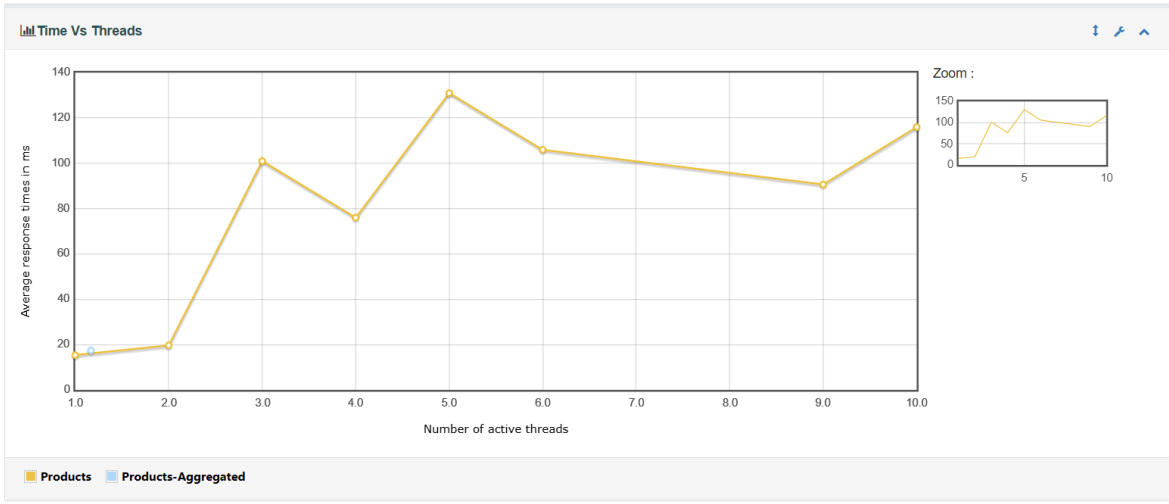
下面是未使用 redis 的**成功请求**响应时间的**百分位数**数据，大部分响应时间都接近 20ms



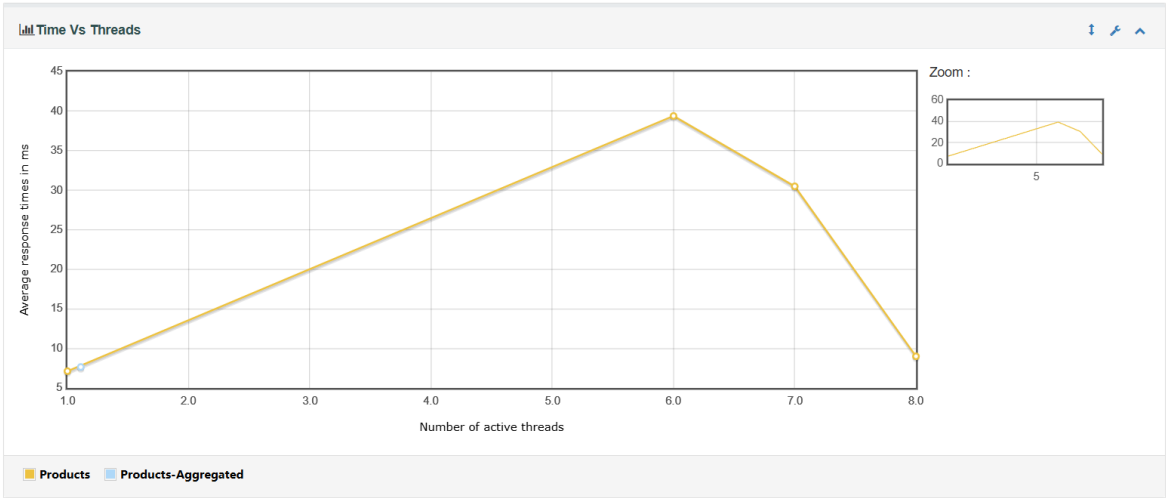
下面是使用了 redis 的**成功请求**响应时间的**百分位数**数据，大部分响应时间都接近 8ms，redis 提速较明显



下图是在多线程并发的条件下的响应时间，没有使用到 redis

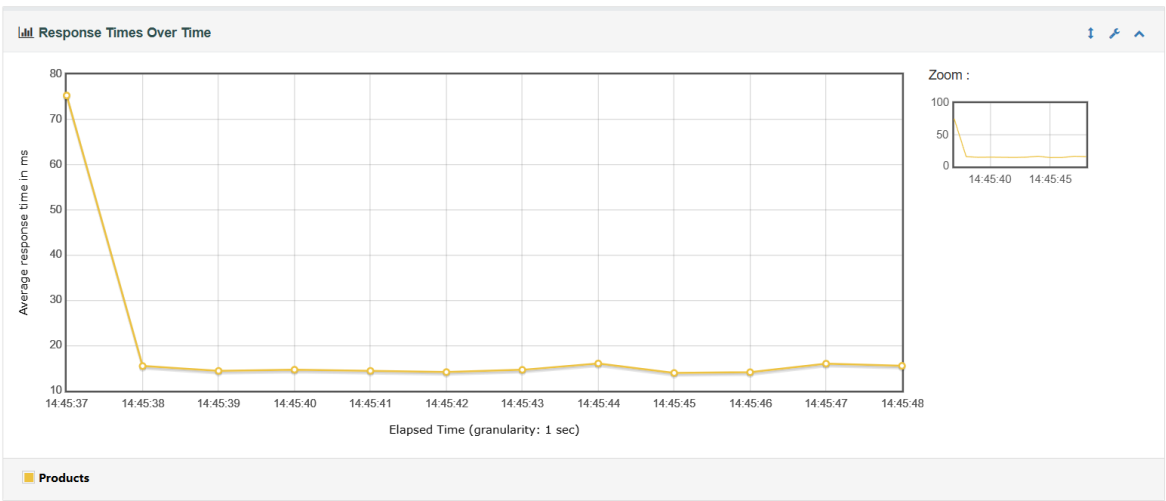


下图是在多线程并发的条件下的响应时间，使用到了 redis，可以看出 redis 的响应时间比未使用到的时间块接近 100ms

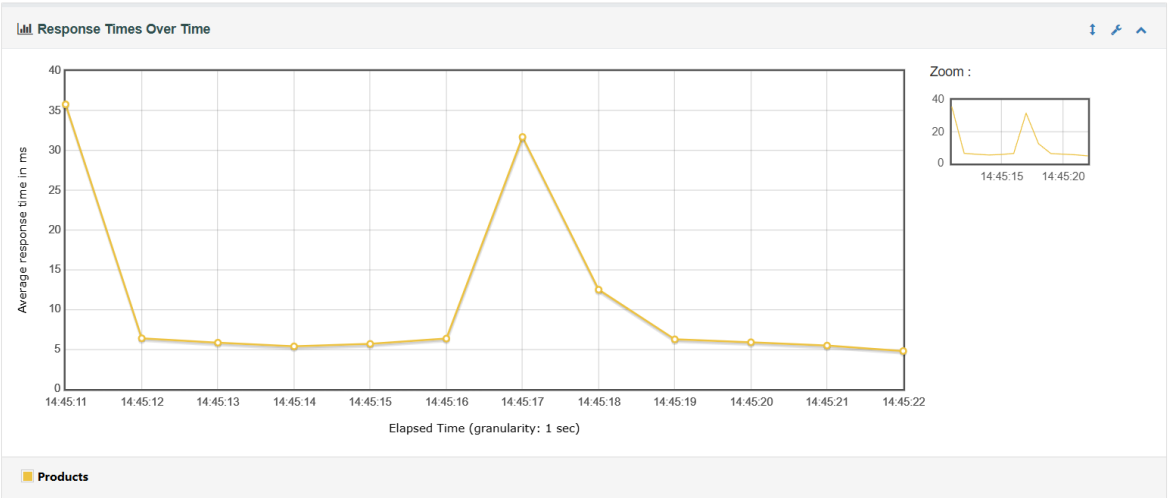


(2).测试数据：thread：800、rampTime：10、loop：1

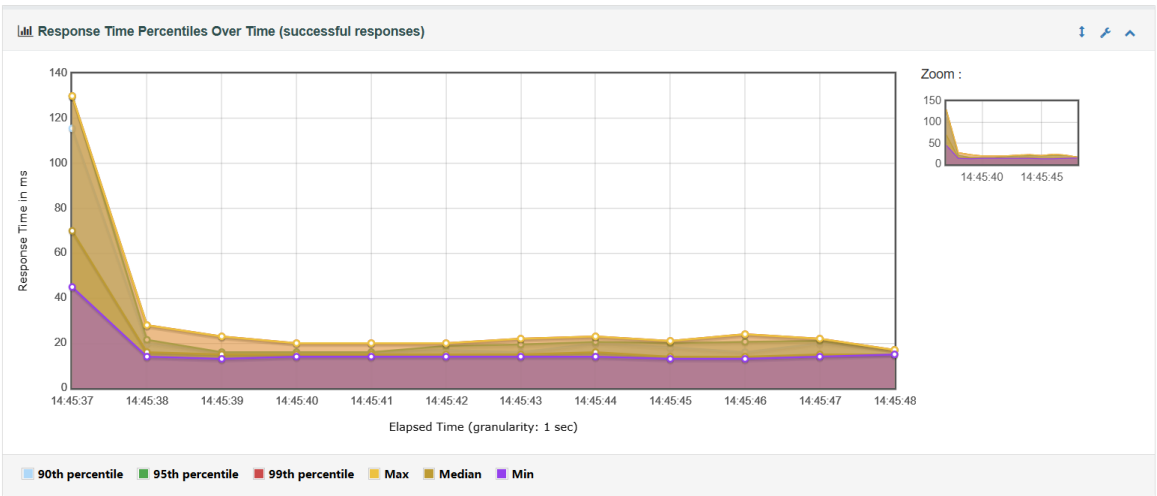
未使用 redis 接口的响应时间，可以看出平均在 15ms 上下



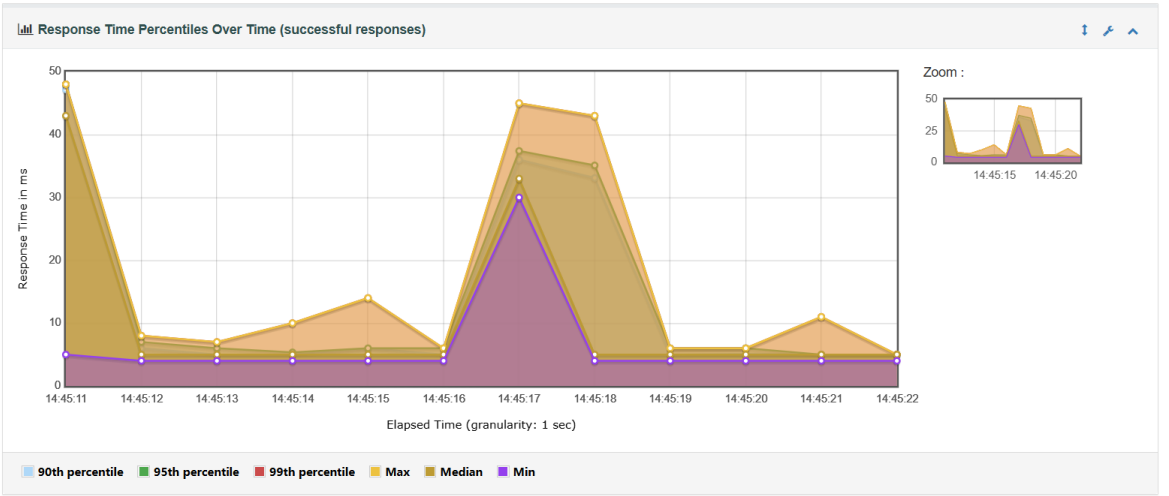
使用了 redis 接口的响应时间，可以看出平均在 5ms 上下，其中有一个响应时间较高的数据是缓存没有命中导致的



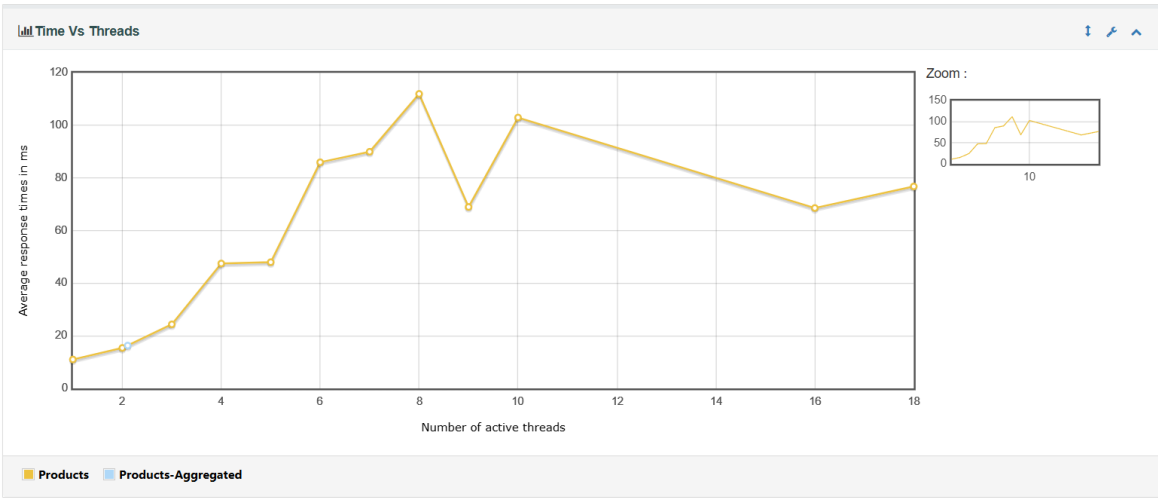
下面是未使用 redis 的成功请求响应时间的百分位数数据，大部分响应时间都接近 20ms



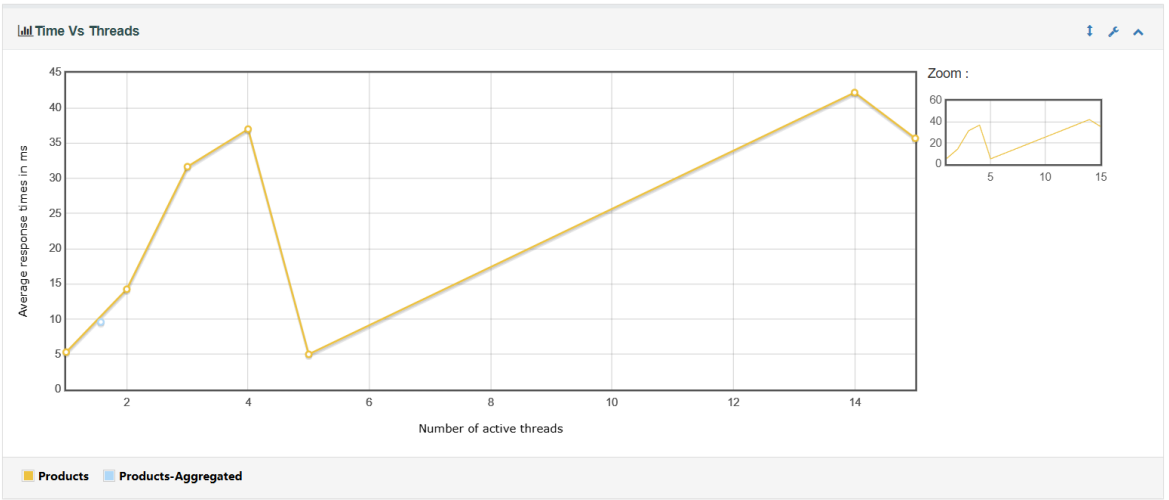
下面是使用 redis 的成功请求响应时间的百分位数数据，大部分响应时间都接近 4ms，有一个异常数据是因为缓存没有命中



下图是在多线程并发的条件下的响应时间，没有使用到 redis

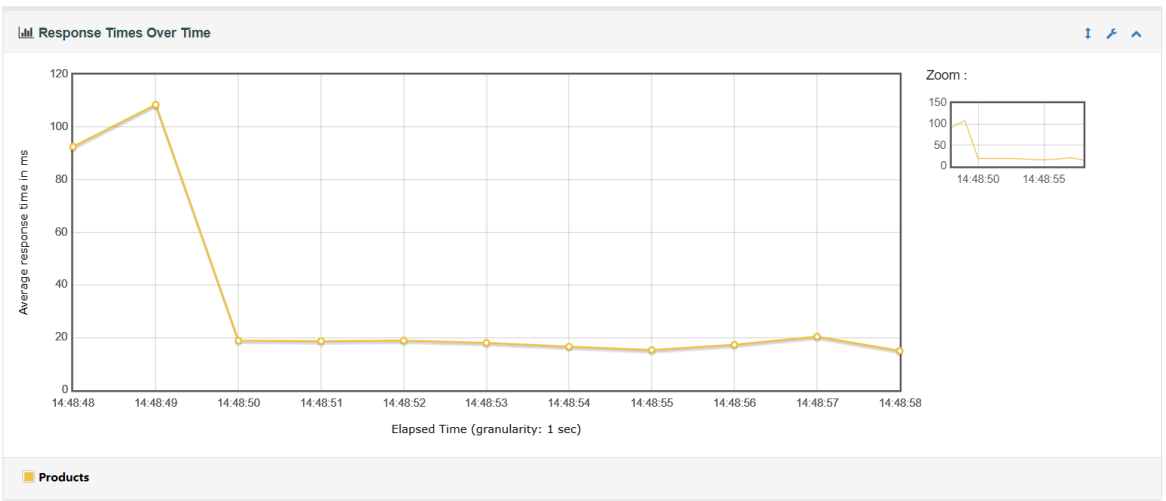


下图是在多线程并发的条件下的响应时间，使用到了 redis，可以看出响应时间大幅提升，普遍在 40ms 以下

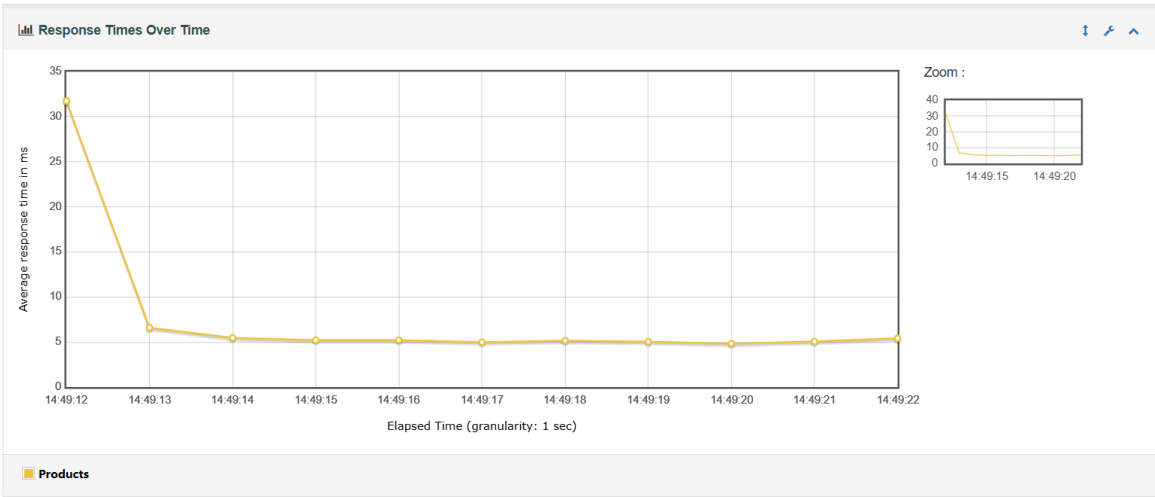


(3).测试数据：thread：1600、rampTime：10、loop：1

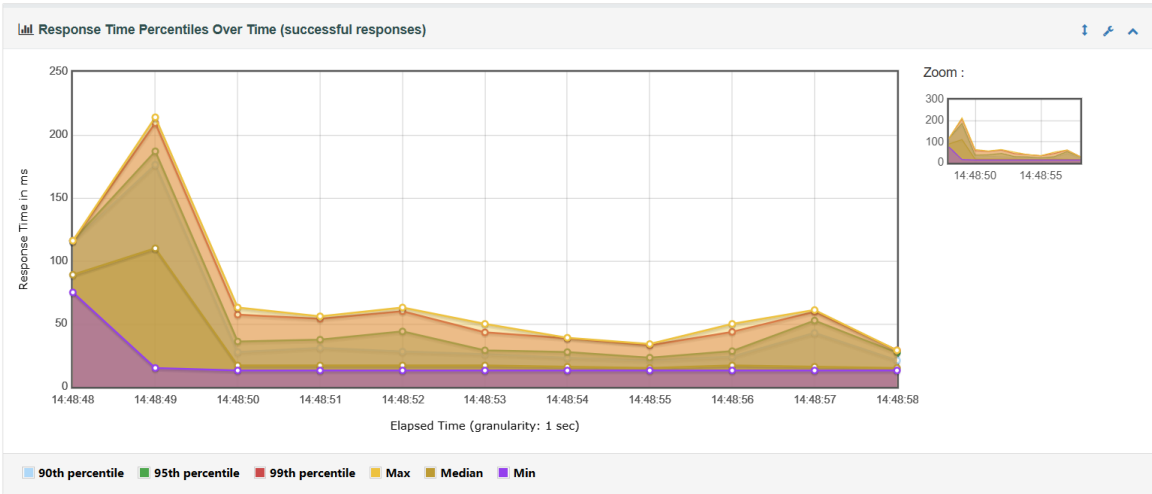
未使用 redis 接口的响应时间，可以看出平均在 20ms 上下



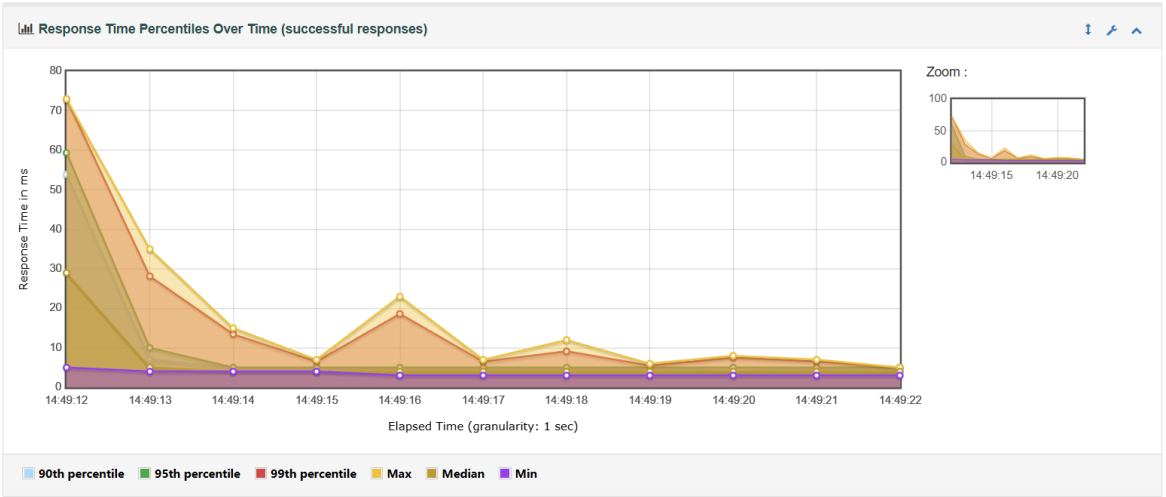
使用 redis 接口的响应时间，可以看出平均在 5ms 上下



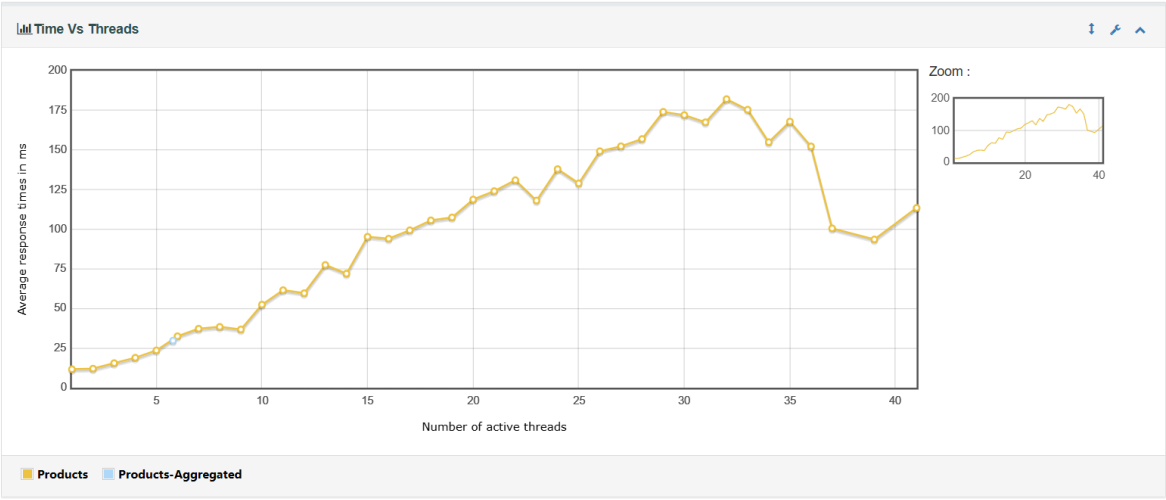
下面是未使用 redis 的成功请求响应时间的百分位数数据，大部分响应时间都接近 50ms



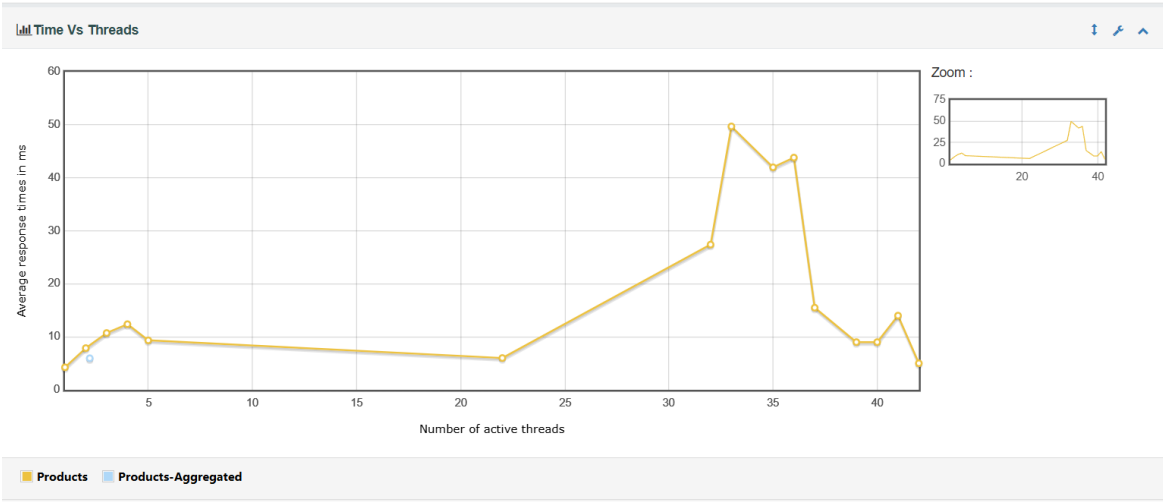
下面是使用 redis 的成功请求响应时间的百分位数数据，大部分响应时间都接近 10ms



下图是在多线程并发的条件下的响应时间，没有使用到 redis

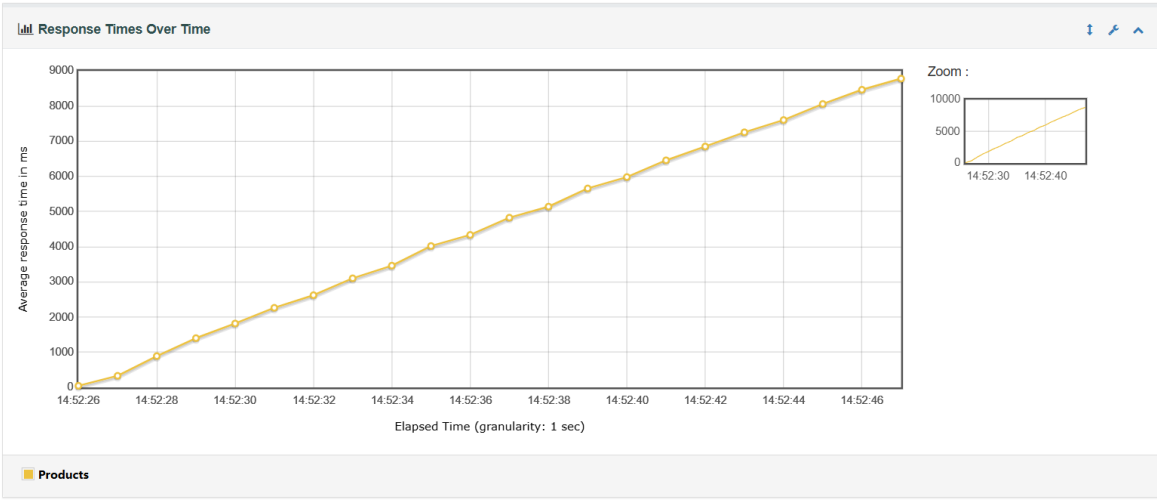


下图是在多线程并发的条件下的响应时间，使用到了 redis，性能提升了 130ms 左右

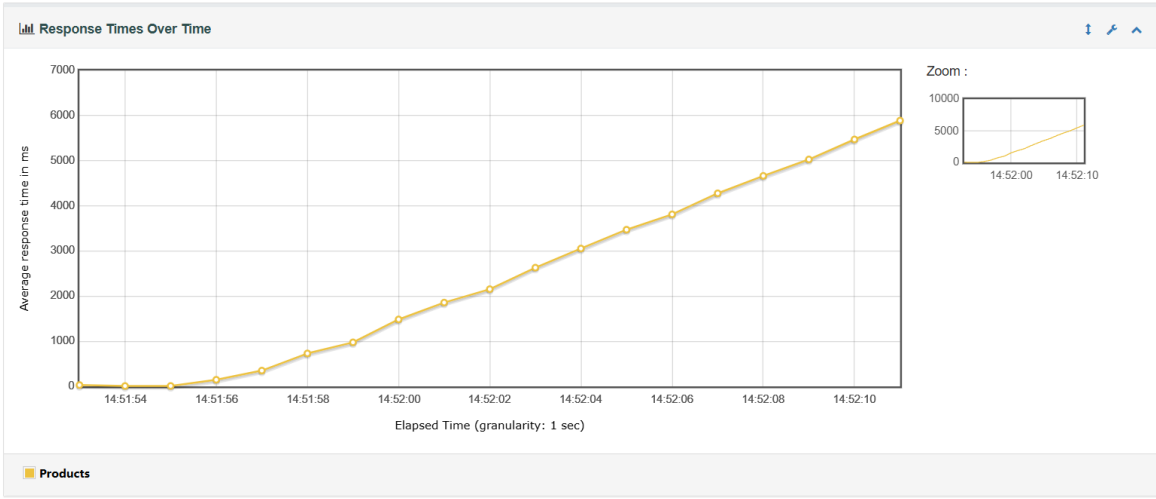


(4).测试数据：thread：4000、rampTime：10、loop：1

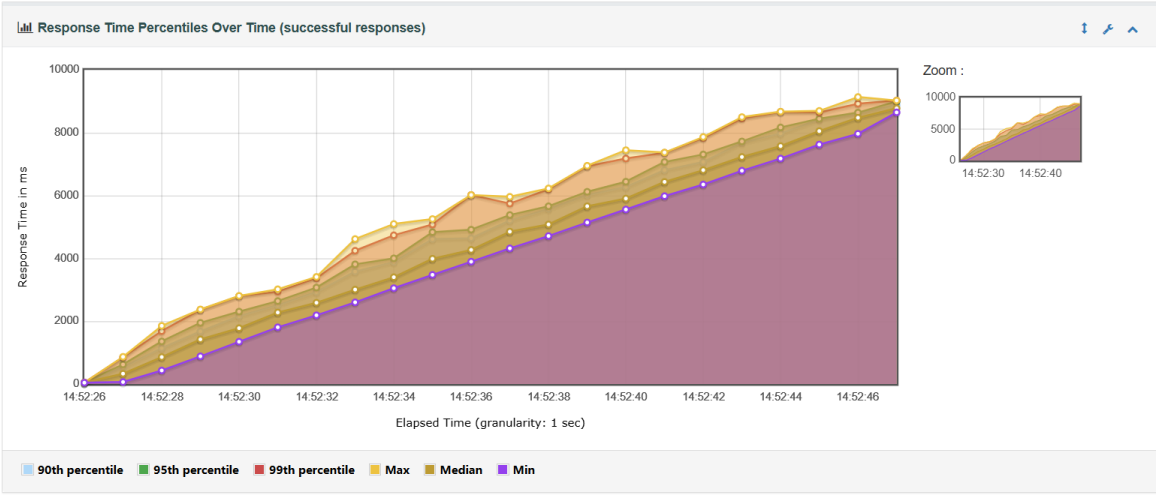
未使用 redis 接口的响应时间，可以看出此时响应时间慢慢变长了，线性增长到接近 9000ms



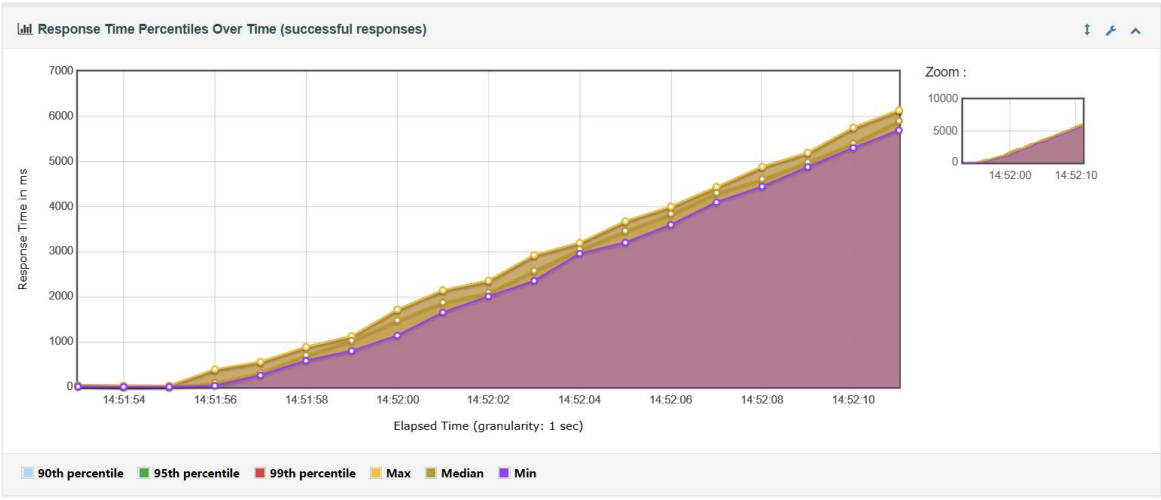
使用 redis 接口的响应时间，可以看出此时响应时间也慢慢变长了，但时间总体上小于未使用 redis



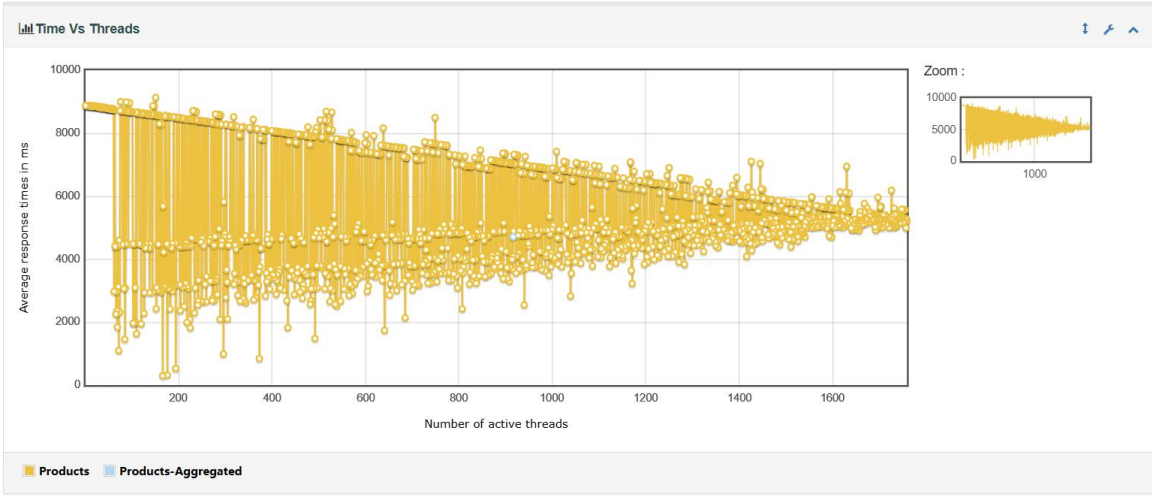
下面是未使用 redis 的成功请求响应时间的百分位数数据，同样是线性增长，最终会接近 9000ms



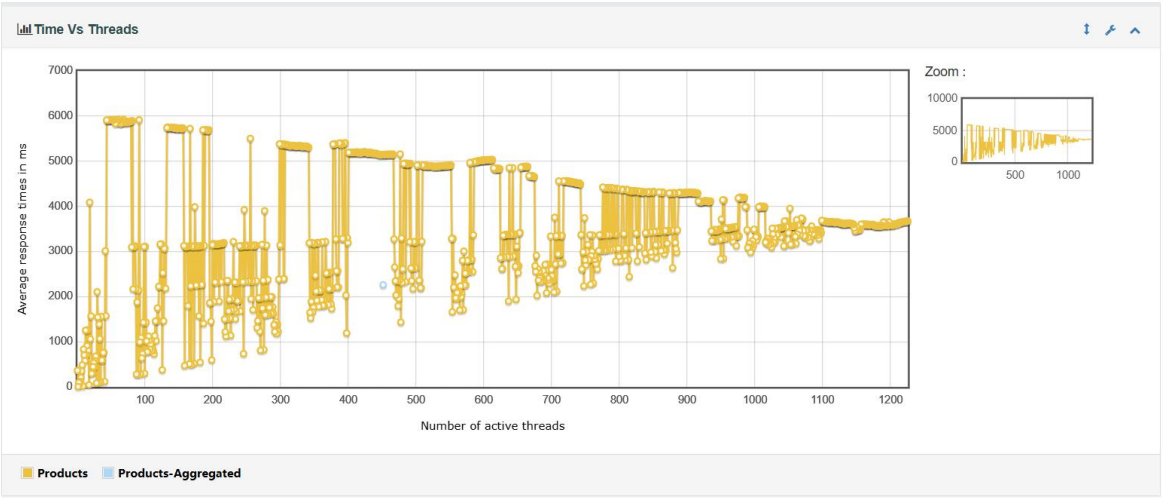
下面是使用 redis 的成功请求响应时间的百分位数数据，同样是线性增长，总体时间仍然小于未使用 redis 的时间



下图是在多线程并发的条件下的响应时间，没有使用到 redis

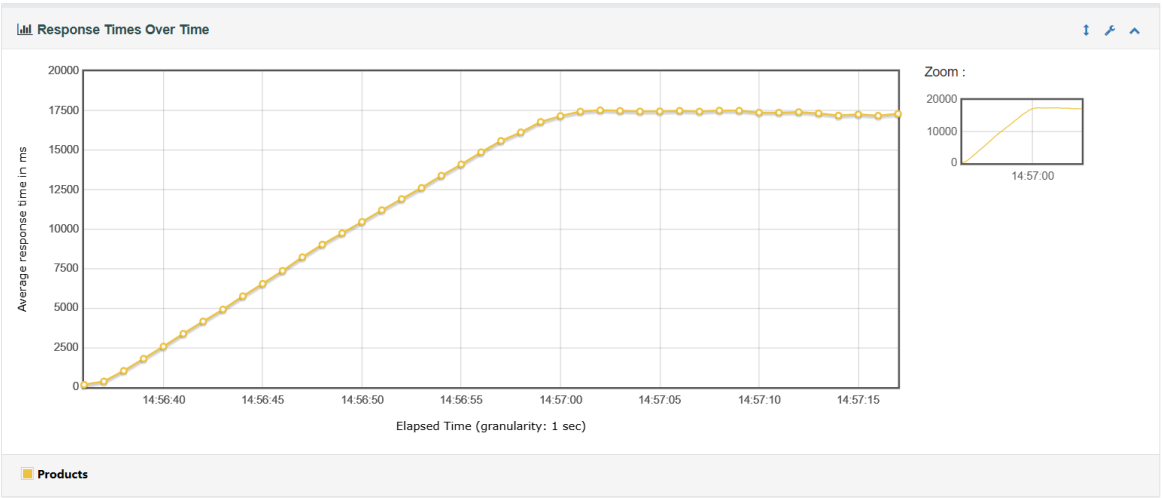


下图是在多线程并发的条件下的响应时间，使用到了 redis，显然在高并发条件下 redis 的效果更好

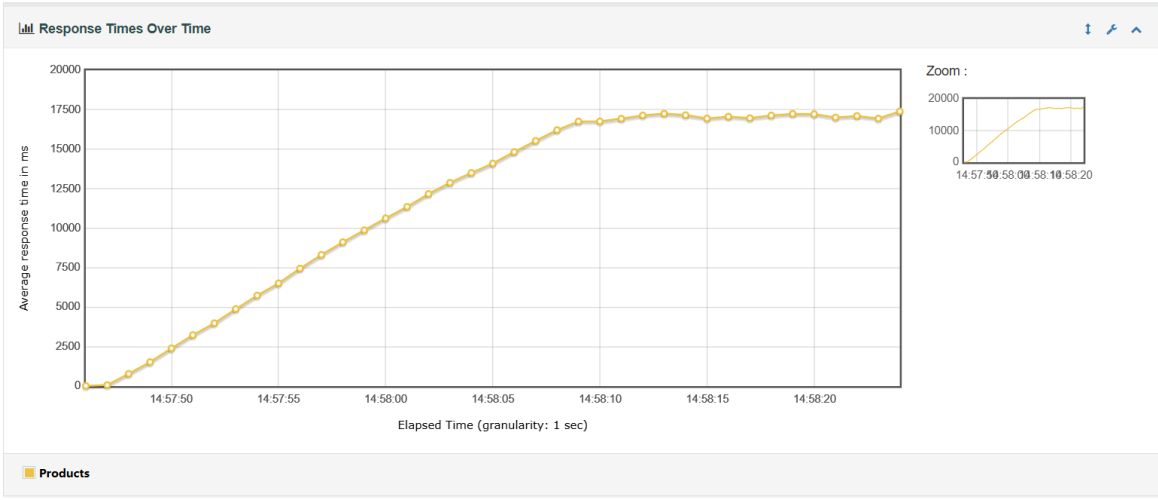


(5).测试数据：thread：8000、rampTime：10、loop：1

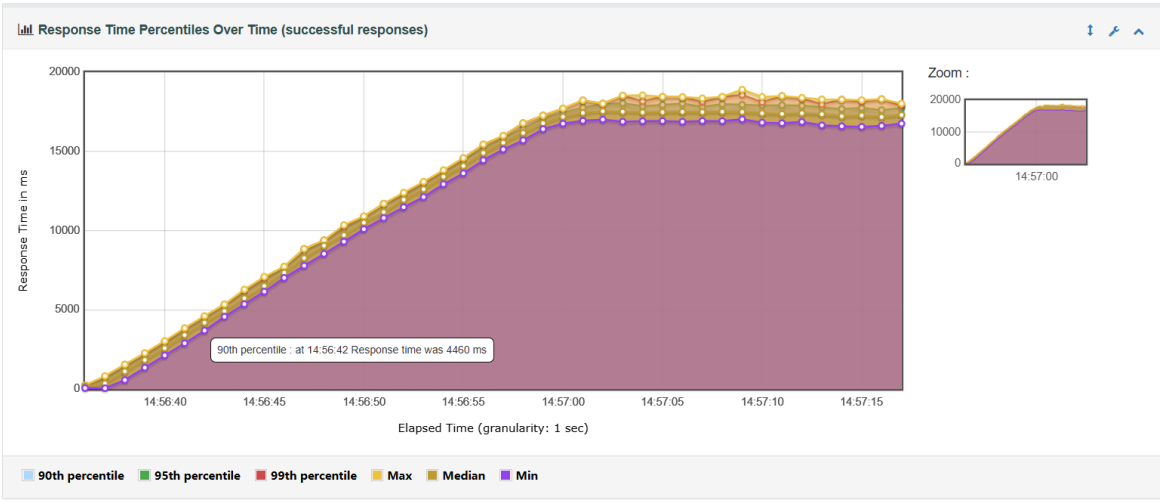
未使用 redis 接口的响应时间，可以看出响应时间已经基本稳定，可能是受到服务器资源的影响



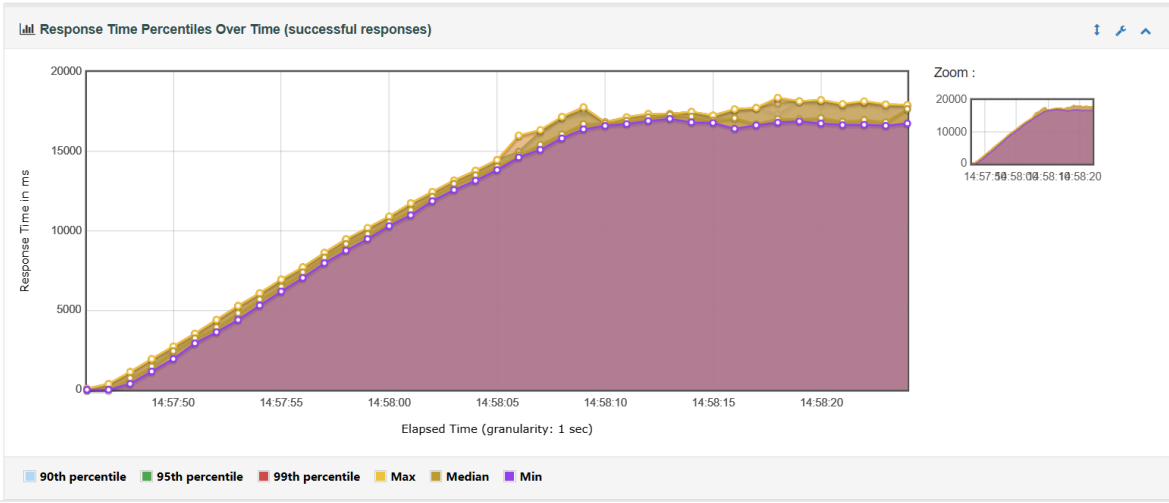
同样的，使用 redis 时由于线程数太多，也会出现响应时间稳定的情况，但到达瓶颈前，redis 的响应时间仍然小于未使用 redis 的时间



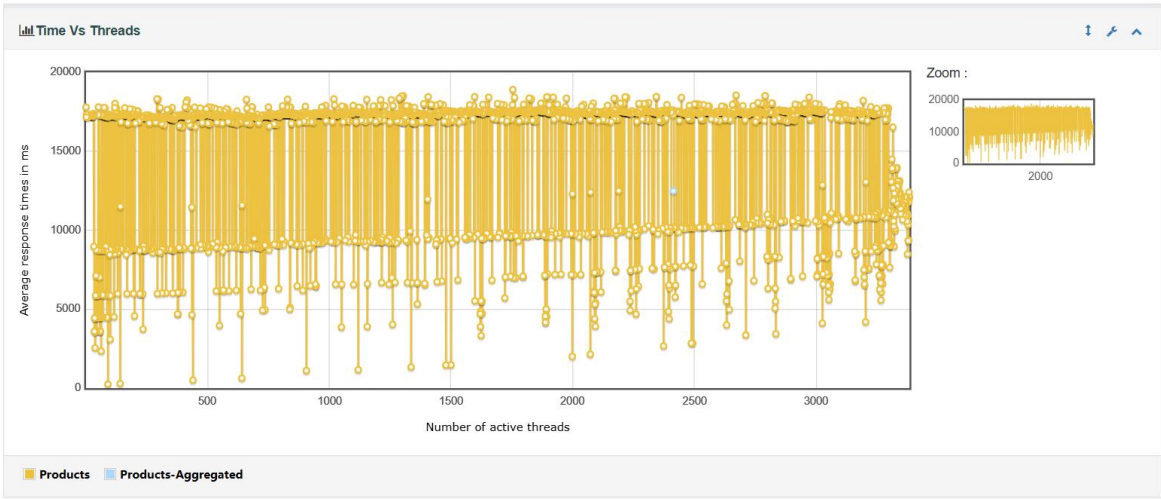
下面是未使用 redis 的成功请求响应时间的百分位数数据，也出现了瓶颈



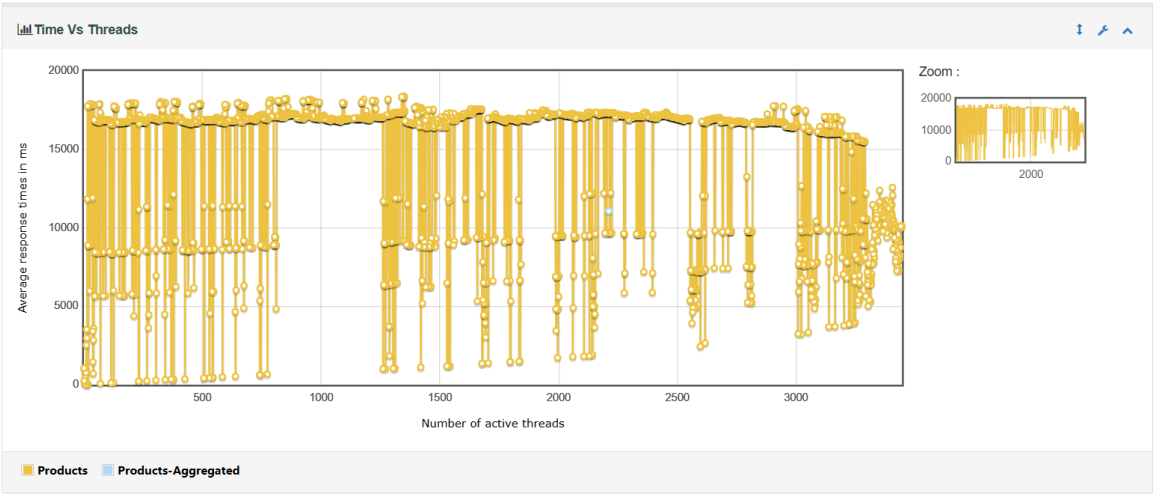
下面是使用 redis 的成功请求响应时间的百分位数数据，同样出现了瓶颈，理由和上述相同



下图是在多线程并发的条件下的响应时间，没有使用到 redis



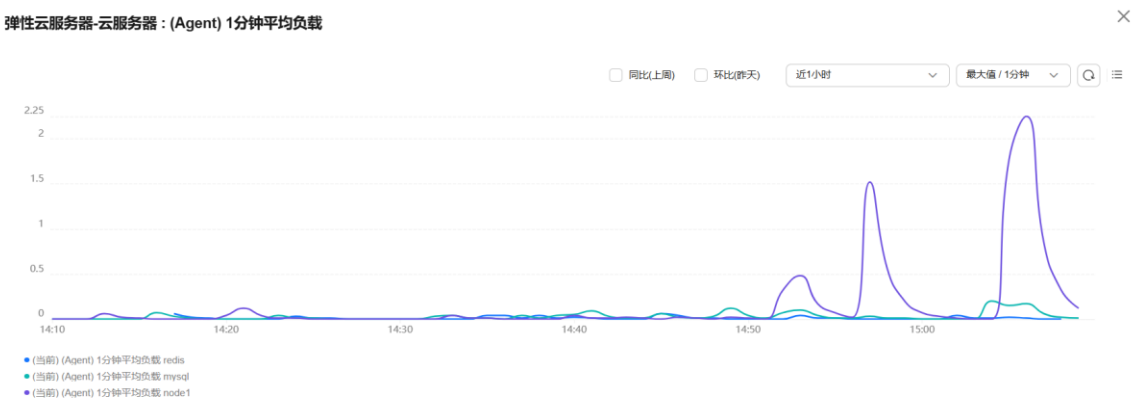
下图是在多线程并发的条件下的响应时间，使用到了 redis，但并发条件下使用 redis 时响应时间较小的线程数更多，仍会有优化



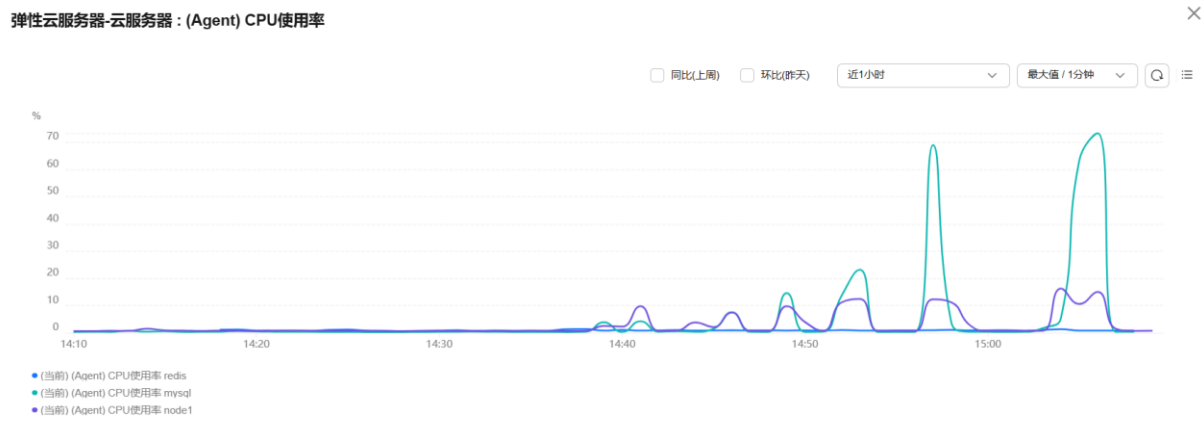
4.华为云云监控结果

(1).1 分钟平均负载

有了 redis 后，数据库的压力明显下降，主要的查询压力放到了 redis 层

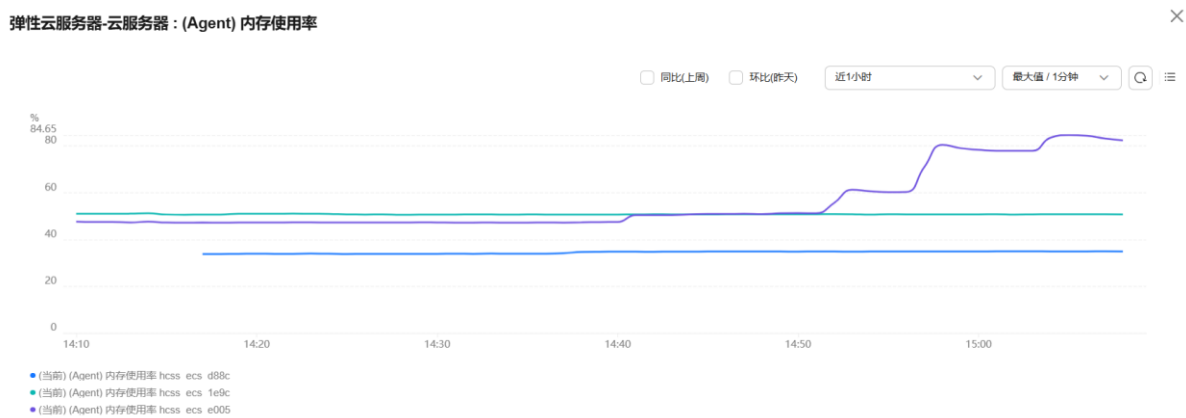


(2).CPU 使用率



(3).内存使用率

由于 redis 缓存是基于内存实现的，所以部署了 redis 的服务器内存使用率很高



5.总结

使用 Redis 缓存能够显著提高系统的性能，减少数据库负载，降低响应时间，提升吞吐量，缓存的使用适合查询频繁、数据变化不大的场景，对于高并发读取操作尤为有效。对于大规模数据和高频率写操作的系统，需要合理设计缓存失效策略和更新机制，以保证缓存的有效性和一致性。在生产环境中，需要结合具体的业务需求和系统架构，合理选择缓存策略，避免缓存穿透、缓存击穿等问题。

References:

[1] 实验代码仓 https://github.com/YUK1PEDIA/XMU-JavaEE-exp6-Redis.git

[2] JMeter 下载与使用文档：https://jmeter.apache.org/download_jmeter.cgi

[3] Spring Data JPA 官方文档：https://docs.spring.io/spring-data/jpa/docs/current/reference/html/

[4] MyBatis 官方文档：https://mybatis.org/mybatis-3/

[5] Redis 官方文档：redis 中文文档