基于 Spring Data JPA 的关联实现方案

2-6 小组

摘　要：Spring Data JPA 是 Spring Data 项目的一部分，它提供了一种简化的数据访问方式，用于与关系型数据库进行交互，它基于 Java Persistence API（JPA）标准，并提供了一套简介的 API 和注解，能够通过简单的 Java 对象来表示数据库表，并通过自动生成的 SQL 语句执行常见的 CRUD 操作。Spring Data JPA 通过封装 JPA 的复杂性，简化了数据访问层的开发工作，还提供了丰富的查询方法的定义、分页和排序支持、事务管理等功能。本文探讨了基于 Spring Data JPA 的关联实现方案，通过设计实验模拟多用户并发请求，利用负载功能测试和性能测试开源工具软件 Apache JMeter 去测试本方案的速度和服务器负载，并与第四次实验所得结果进行对比。

关键词：Spring Data JPA;JMeter;MySQL;持久化规范

1.实验描述

1.1 实验环境

服务器 A: Ubuntu 18.04 服务器 2 核 1G 内存虚拟机一台，安装 docker，Maven、 git，作为管理机，用于编译实验代码

服务器 B：Ubuntu 18.04 服务器 2 核 1G 内存虚拟机一台，安装 docker，部署 productdemoaop Docker

服务器 C：Ubuntu 18.04 服务器 2 核 1G 内存虚拟机一台，安装 docker，部署 MySQL Docker

服务器 D：Ubuntu 18.04 服务器 2 核 1G 内存虚拟机一台，安装 JMeter 5.6.3,用于测试

1.2 实验内容

采用 Spring Data JPA 可以方便的实现 MySQL 数据库的查询，仿照实验四中的方案 2,编写 Spring Data JPA 实验代码，实现 GET /products?name=xxxx 查询产品完整信息,比较 JPA 方案与实验四中三个方案的速度差异和服务器的负载。

2.实验设计

2.1 项目依赖

由于 pom.xml 没有 JPA 依赖，需要引入，故在 pom.xml 文件中新增 JPA 依赖。

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

图 1 Spring Data JPA 相关依赖引入

2.2 实体类设计

由于 JPA 是一个比较完全式的 ORM（对象-关系映射）框架，可以完全通过实体映射数据库，从而需要通过 JPA 独特的注解来定义各个实体类。

仿照 dao.bo.OnSale、Product、User 类，定义 JPA 需要用到的实体类，如图 2、3、4 所示。

@GeneratedValue(strategy = GenerationType.IDENTITY)表示主键生成策略为自增，即数据库会自动生成主键值，本次实验只用查询数据库，不用对数据库进行增删改，故而可以不需要该标识。

ProductEntity 和 OnsaleEntity 的 creator 和 modifier 字段类型为 User（dao.bo.User 类），这两个字段都被标记为@Transient，表示 creator、modifier 不会被 JPA 持久化，即不会存储到数据库中。

图 2 ProductEntity 定义

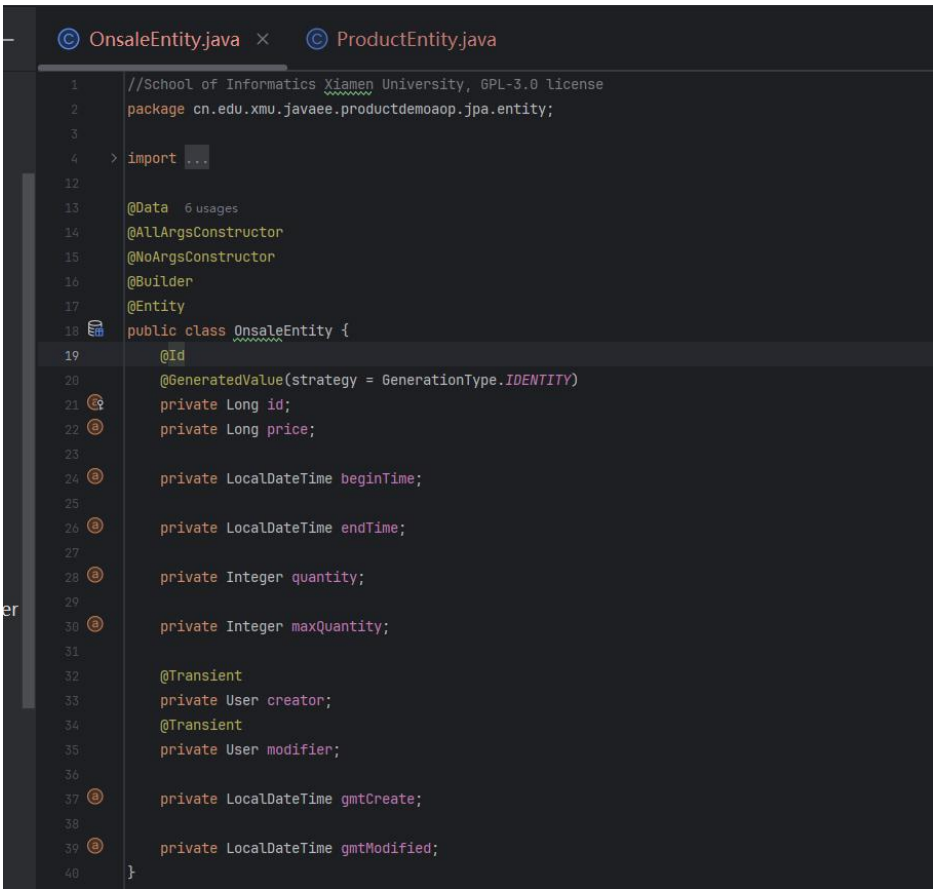

图 3 OnsaleEntity 定义

2.3 Repository 接口设计

若只是对简单的对表进行 CRUD 操作，只需要继承 JpaRepository 接口，传递了两个参数：①实体类，②实体类中的主键类型。

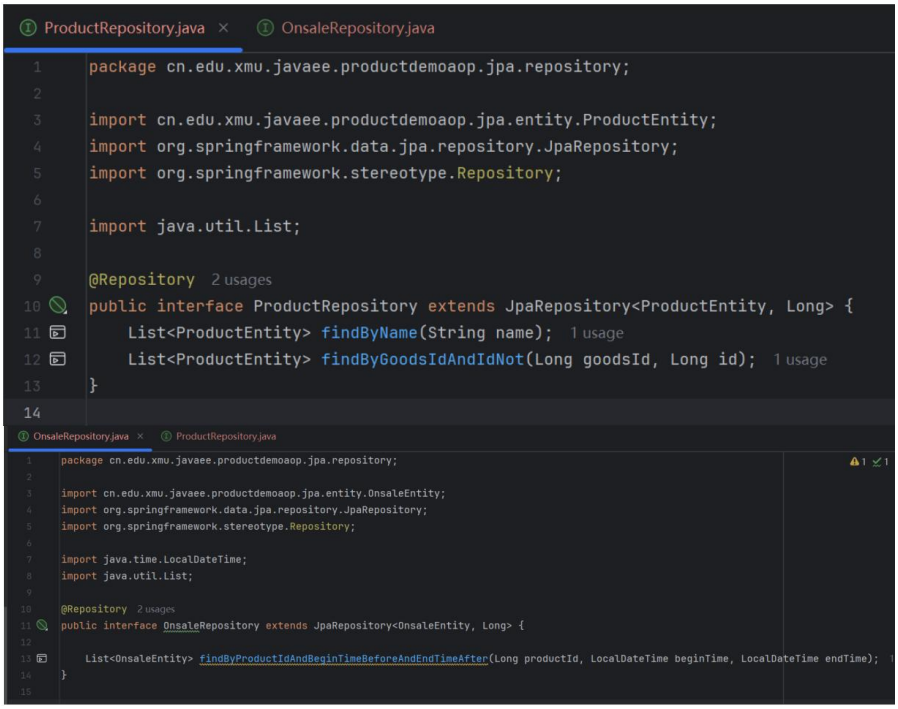

图 4　Repository 接口定义

ProductRepository 继承 JpaRepository 接口，泛型参数指定了实体类型 ProductEntity 和主键类型 Long。OnSaleRepository 同理。这些接口主要用于处理各实体类相关的数据库操作，利用 Spring Data JPA 提供的自动生成的查询方式，实现增删改查（本实验只需要查）。

2.4 相关文件设计

在 ServiceDao 文件里，添加 JPA 具体的查询操作。

ProductDao.java × OnsaleRepository.java ProductRepository.java

```
39      public class ProductDao {
198
199         @Resource  2 usages
200         private ProductRepository productRepository;
201         @Resource  1 usage
202         private OnsaleRepository onsaleRepository;
203
204
205         public List<ProductEntity> findByName_jpa(String name) {  1 usage  new *
206             LocalDateTime time = LocalDateTime.now();
207             List<ProductEntity> productList = productRepository.findByName(name);
208             for(ProductEntity product : productList){
209                 Long id = product.getId();
210                 List<OnsaleEntity> onSaleList = onsaleRepository
211                         .findByProductIdAndBeginTimeBeforeAndEndTimeAfter(id,time,time);
212                 product.setOnSaleList(onSaleList);
213                 Long goods_id = product.getGoodsId();
214                 List<ProductEntity> otherProduct = productRepository.findByGoodsIdAndIdNot(goods_id,id);
215                 product.setOtherProduct(otherProduct);
216             }
217             return productList;
218         }
219
220     }
```

图 5 ProductDao Jpa 查询方法设计

在 ProductService 文件里，添加使用 JPA 查询的方法，并 ProductController 文件里，添加 jpa 分支以调用 JPA 方法。

serviceDao

```
@Transactional  1 usage  new *
public List<ProductEntity> findProductByName_jpa(String name){
    return productDao.findByName_jpa(name);
}
```

serviceController

```
@GetMapping("/") mingqiu@xmu.edu.cn <mingqiu@xmu.edu.cn> *
public ReturnObject searchProductByName(@RequestParam String name, @RequestParam(required = false, defaultValue = "auto") String type) {
    List<ProductDto> data =null;
        ReturnObject retObj = null;
    List<Product> productList = null;
    if (null != type && "manual".equals(type)){
        productList = productService.findProductByName_manual(name);
        data = productList.stream().map(o->CloneFactory.copy(new ProductDto(),o)).collect(Collectors.toList());

    } else if (null != type && "jpa".equals(type)) {
        List<ProductEntity> productByNameManualJpa = productService.findProductByName_jpa(name);
        data = productByNameManualJpa.stream().map(o->CloneFactory.copy(new ProductDto(),o)).collect(Collectors.toList());
    } else {
        productList = productService.retrieveProductByName(name,  all: true);
        data = productList.stream().map(o->CloneFactory.copy(new ProductDto(),o)).collect(Collectors.toList());
    }
    retObj = new ReturnObject(data);
    return  retObj;
}
```

图 6 ProductService、ProductController Jpa 查询设计

2.5 API 验证

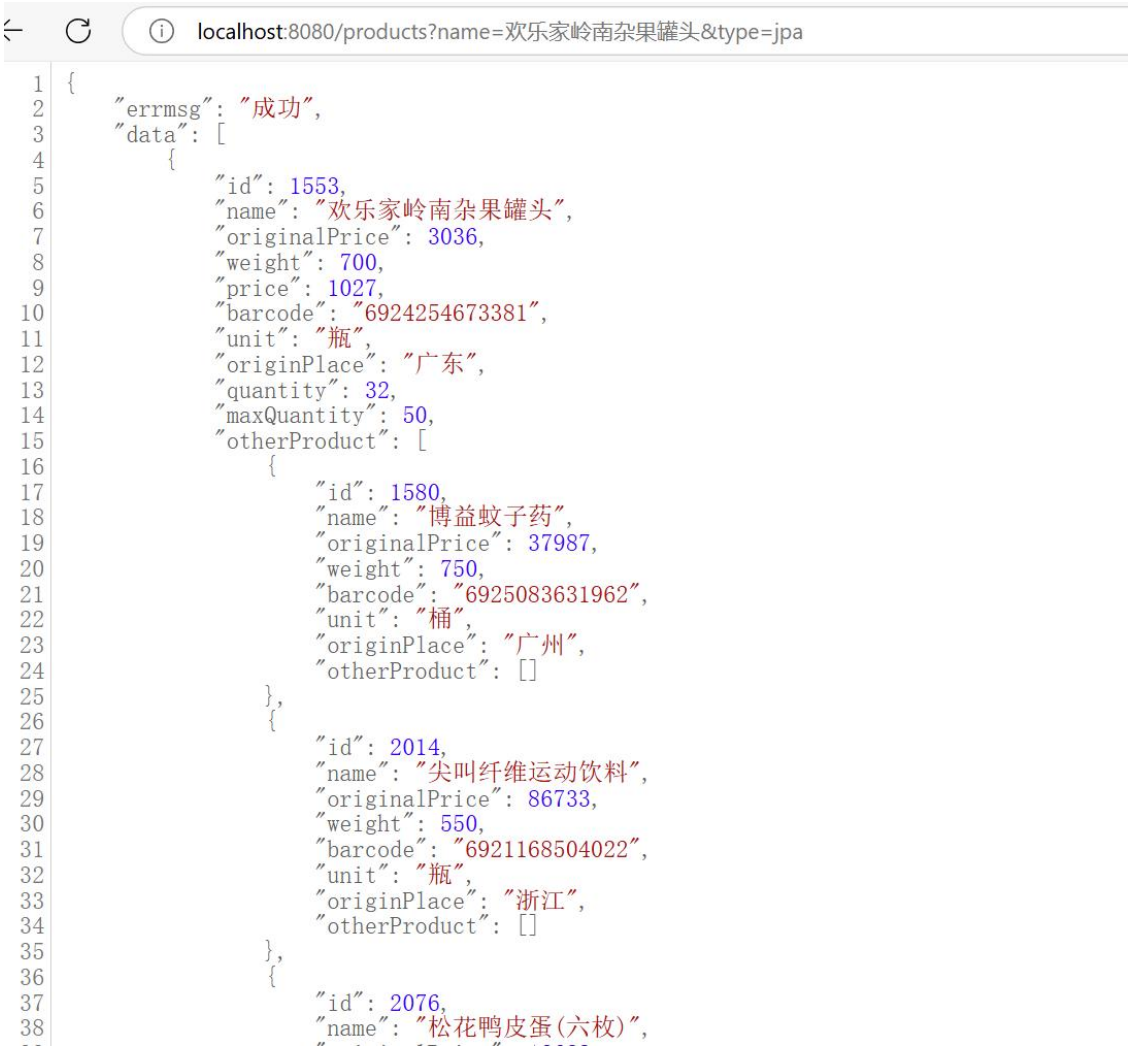

图 7 Jpa 查询 API 测试

2.6 jpa 生成的 SQL 语句

```
Hibernate:
    /* <criteria> */ select
        pe1_0.id,
        pe1_0.barcode,
        pe1_0.commission_ratio,
        pe1_0.free_threshold,
        pe1_0.gmt_create,
        pe1_0.gmt_modified,
        pe1_0.goods_id,
        pe1_0.name,
        pe1_0.origin_place,
        pe1_0.original_price,
        pe1_0.sku_sn,
        pe1_0.status,
        pe1_0.unit,
        pe1_0.weight
    from
        goods_product pe1_0
    where
        pe1_0.name=?
```

```
Hibernate:
    /* <criteria> */ select
        oe1_0.id,
        oe1_0.begin_time,
        oe1_0.end_time,
        oe1_0.gmt_create,
        oe1_0.gmt_modified,
        oe1_0.max_quantity,
        oe1_0.price,
        oe1_0.product_id,
        oe1_0.quantity
    from
        goods_onsale oe1_0
    where
        oe1_0.product_id=?
        and oe1_0.begin_time<?
        and oe1_0.end_time>?
```

```
Hibernate:
    /* <criteria> */ select
        pe1_0.id,
        pe1_0.barcode,
        pe1_0.commission_ratio,
        pe1_0.free_threshold,
        pe1_0.gmt_create,
        pe1_0.gmt_modified,
        pe1_0.goods_id,
        pe1_0.name,
        pe1_0.origin_place,
        pe1_0.original_price,
        pe1_0.sku_sn,
        pe1_0.status,
        pe1_0.unit,
        pe1_0.weight
    from
        goods_product pe1_0
    where
        pe1_0.goods_id=?
        and pe1_0.id<>?
```

图 8 jpa 查询生成的 SQL 语句

3. 实验结果描述与分析

为了使用 Spring Data JPA 代码实现 GET /products?name=xxxx&type=jpa 查询产品完整信息，并与实验四中的 auto 和 munual 进行速度和服务器负载的对比。

本次实验分为低负载测试与高负载测试。在 10 秒测试中，用 700-10-2 来与 auto、manual 对比，由于 750-10-2 已达到 auto 瓶颈，因而后续只用比较 jpa 与 manual 方案。

3.1 低负载测试

3.1.1 固定时间 10s，发送 700 个线程，循环 2 次实验结果

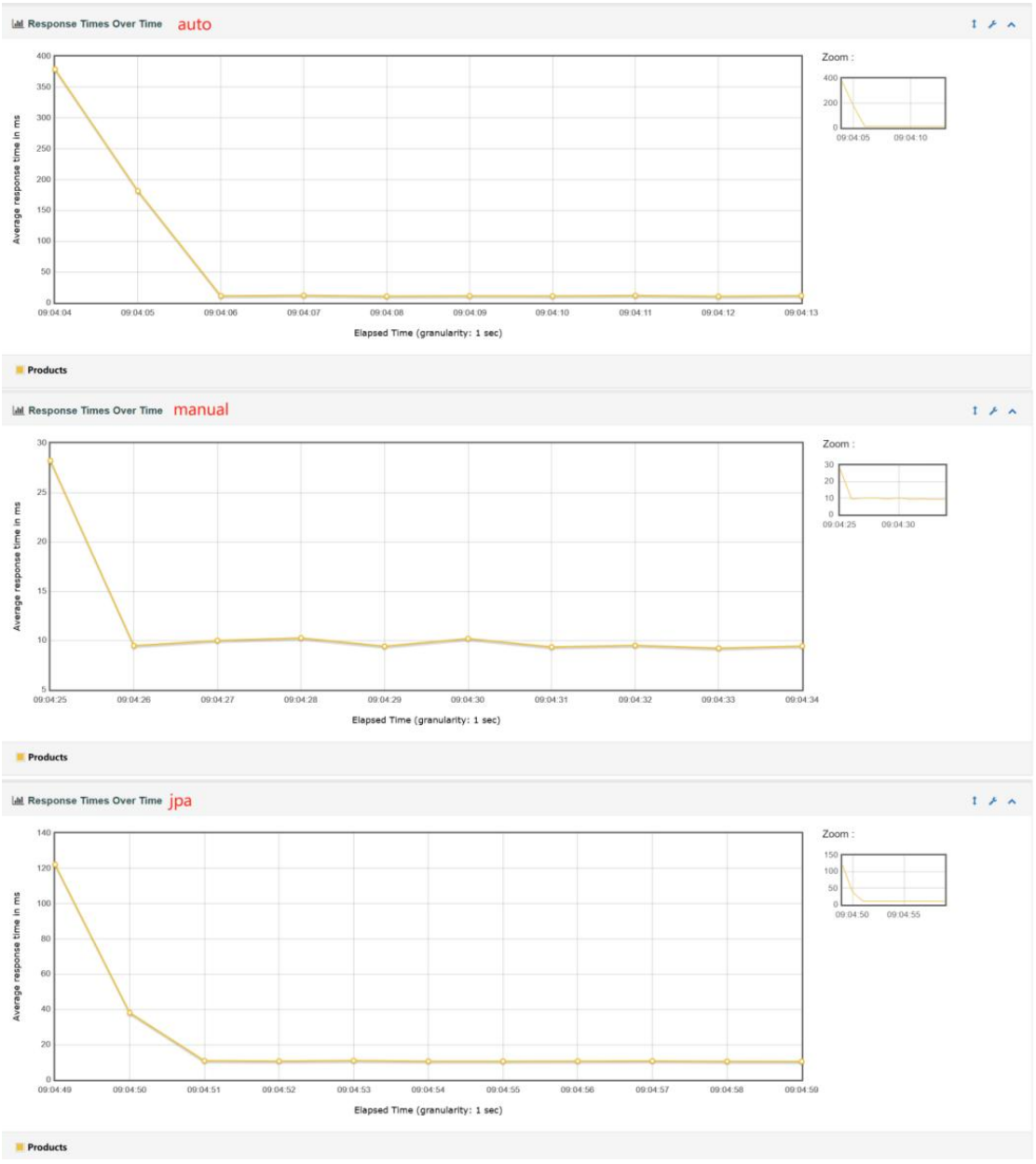

图 9 700-10-2 测试下各方案响应时间

从图 9 可以看出，auto 方案下系统大部分平均响应时间稳定且保持在 12ms 左右，但前 2 秒内响应时间达到 350ms，说明系统响应较慢，manual 方案下系统大部分平均响应时间稳定且保持在 10ms 左右，jpa 方案下大部分保持在 11ms 左右。

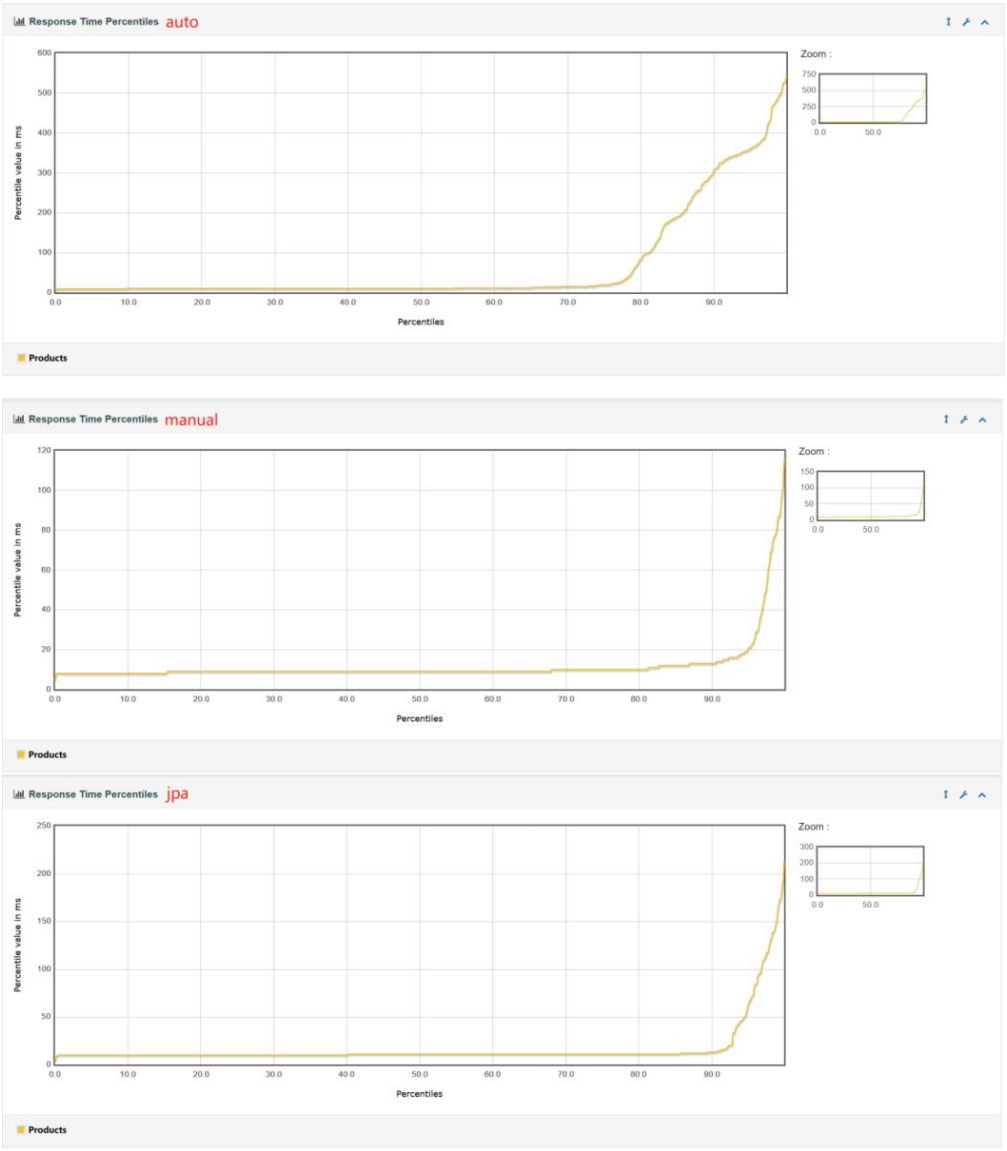

图 10 700-10-2 测试下各方案响应时间百分比

由图 10 可以看出，auto 方案下仅有 65%的请求响应时间在 13ms 内；而 munaul 和 jpa 两个方案下都有 90&的请求响应时间在 13ms 内；

由于测试时间过短，不便观察服务器的监控负载，故本组实验只通过对比各方案响应时间和响应时间百分比初步得出结论，JPA 方案性能远比 auto 方案查询性能要高，与 manual 方案基本一致。

3.1.2 固定时间 10s，发送 750 个线程，循环 2 次实验结果

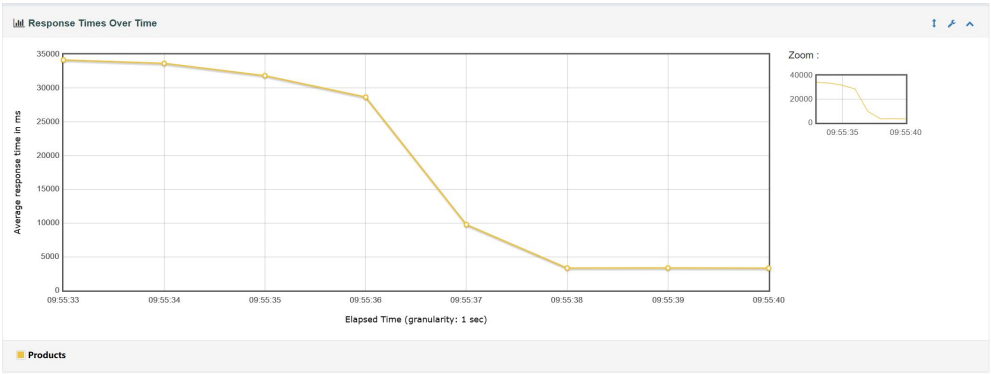

图 11 750-10-2 测试下 auto 响应时间

从图 11 可以看出，auto 在 750-10-2 测试条件下，最低响应时间达到 3300ms，最大响应时间高达 34189ms，说明 auto 方案下服务器已经达到瓶颈，说明 auto 方案远比 jpa 方案差，故而后续测试中只用比较 munual 和 jpa 方案即可。

3.1.3 固定时间 10s，发送 900 个进程，循环 2 次实验结果

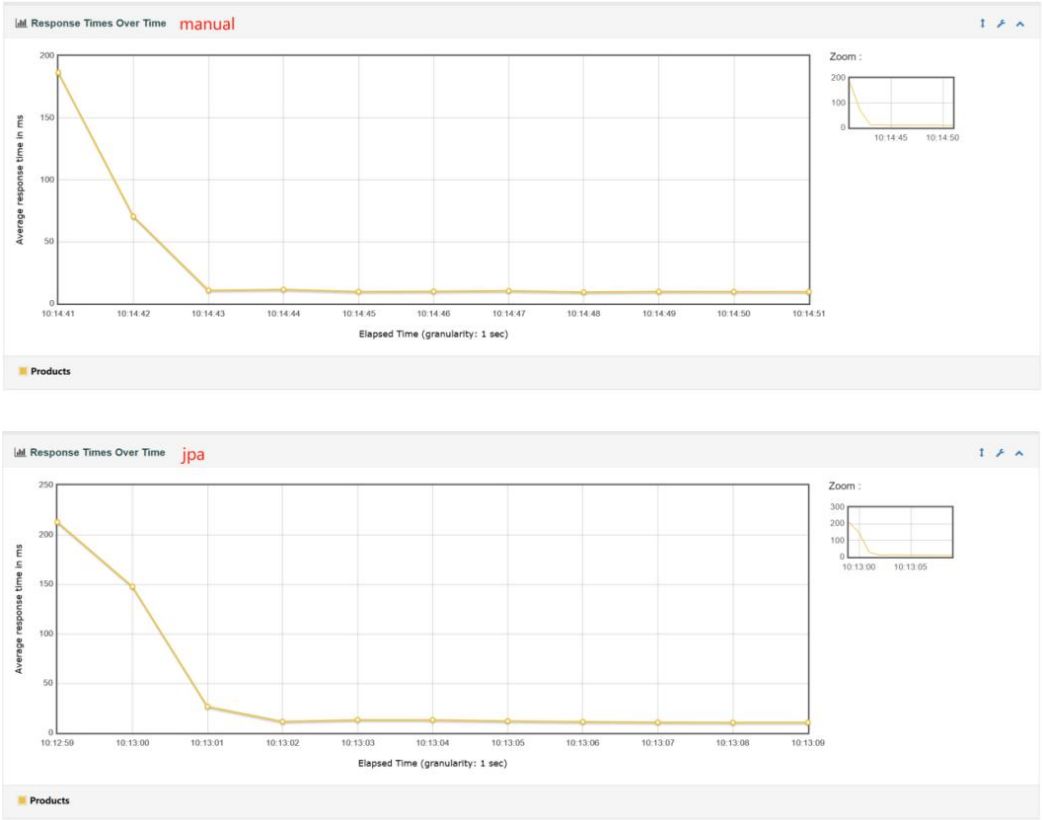

图 12 900-10-2 测试下各方案响应时间

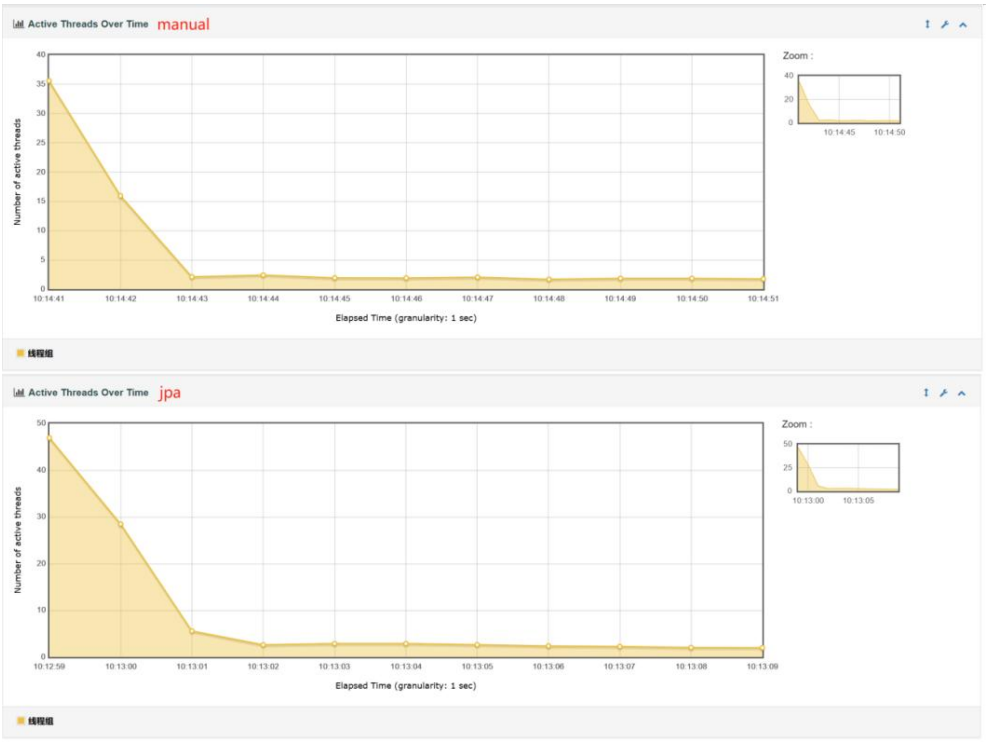

图 13 900-10-2 测试下各方案活跃线程数

由图 12 可以看出，在前两秒内，manual 方案下的响应时间最高达到 187ms，而 jpa 方案下最高响应时间达到 213ms。但是在后 8 秒内，manual 方案和 jpa 方案都稳定在 11ms 左右。

由图 13 可以看出，在前 2 秒内，jpa 方案每秒活跃线程数比 manual 方案要高，因而可以浅层次地推测出，造成前 2 秒二者响应时间不同的原因可能是因为系统每秒处理的线程请求数不同，当然也不排除网络因素的影响。

通过本组实验结果可以看出，jpa 方案和 manual 方案对服务器性能的影响基本一致，查询速度也基本相同。

3.1.4 固定时间 300s，发送 1600 进程，循环 10 次实验结果

图 14 1600-300-10 测试下各方案响应时间

图 15 1600-300-10 测试下各方案响应时间百分比

从图 14、15 可以看出，manual 方案大部分平均响应时间保持在 9ms 左右，jpa 方案大部分平均响应时间保持在 11ms 左右，且二者都有 90%的请求响应时间在 12ms 内。

服务器 CPU 使用率和内存使用率如下图所示：

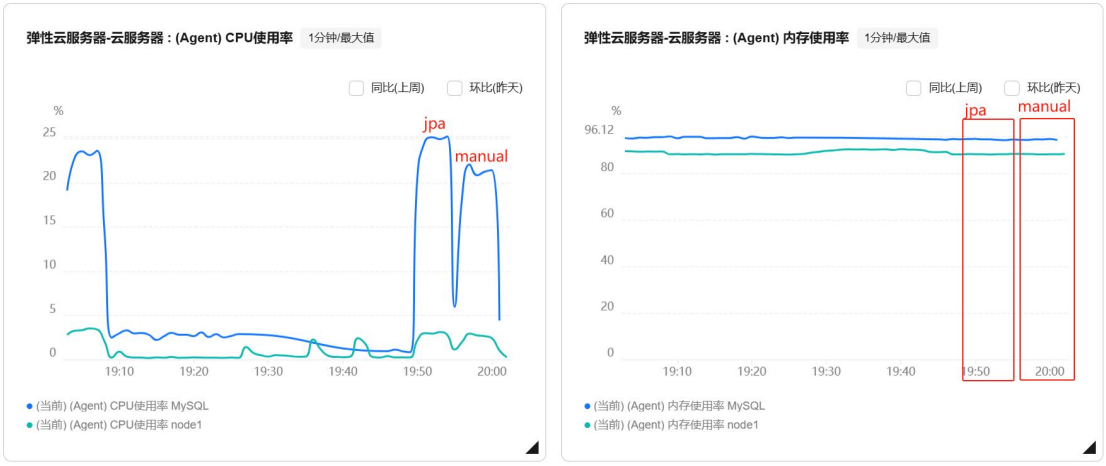

图 16 1600-300-10 测试下服务器 CPU 使用率与内存使用率

从图 16 得知，manual 测试下 CPU 使用率维持在 21%左右，jpa 测试下 CPU 使用率达 25%左右，内存使用率在两种方案下几乎一致。

通过本组实验得出结论，jpa 方案和 manual 方案下，CPU 使用率差距不大，内存使用率也基本一致，故而二者对服务器的负载压力基本一致，查询速度也基本相同。

3.2 高负载测试

3.2.1 固定时间 120s，发送 1600 个线程，循环 15 次实验结果

在本组测试下，如果系统响应请求平均且稳定，则平均每秒响应 200 个请求。

图 17 1600-120-15 测试下各方案响应时间

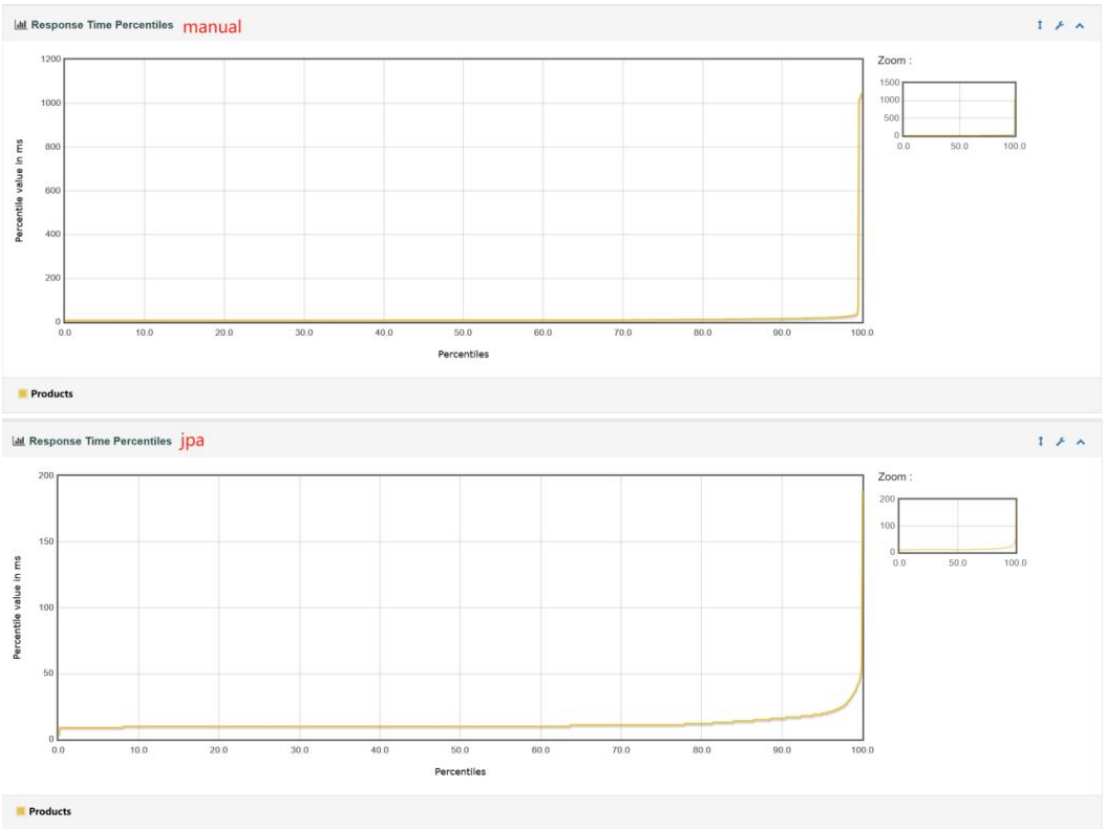

图 18 1600-120-15 测试下各方案响应时间百分比

从图 17、18 可以看出，manual 方案下系统大多数平均响应时间保持在 10ms 左右，但也有较多数响应时间不稳定，最高响应时间为 46ms，有 90%的响应请求时间在 15ms 内；jpa 方案下系统大部分平均响应时间保持在 14ms 左右，相比与 manual 方案响应请求稳定且平均，有 90%的响应请求时间在 16ms 内。

服务器 CPU 使用率和内存使用率如下图所示：

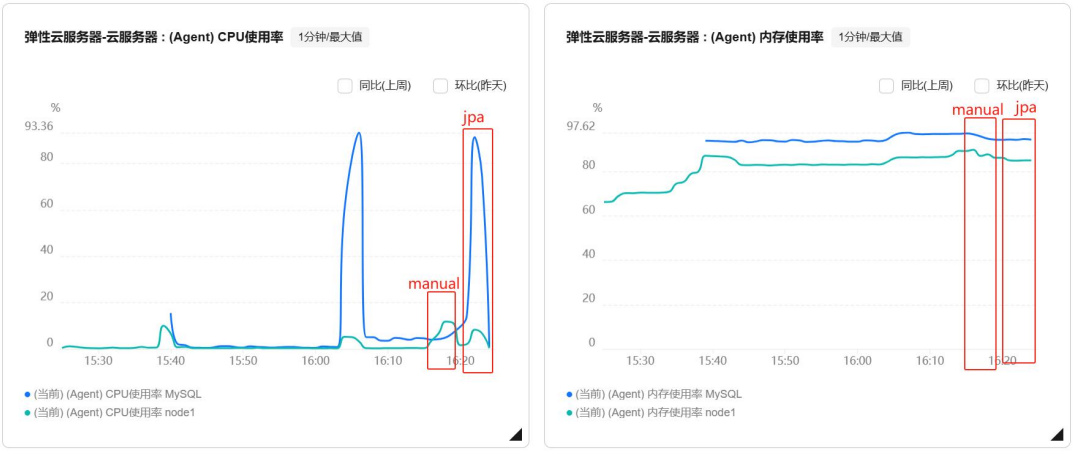

图 19 1600-120-15 测试下服务器 CPU 使用率与内存使用率

由图 19 可知，manual 方案下 MySQL 服务器 CPU 使用率仅仅为 8%左右，node1 服务器 CPU 使用率为 11%左右；jpa 方案下，MySQL 服务器 CPU 使用率达到 91%左右，node1 服务器 CPU 使用率为 8%左右。在两个方案下，MySQL 和 node1 服务器的内存使用率基本一致。

本组结果可以初步说明，在高并发请求条件下，jpa 方案对服务器负载的压力要高于 manual 方案。

3.2.2 固定时间 300s，发送 1600 个线程，循环 40 次实验结果

在本组测试下，如果系统响应请求平均且稳定，则平均每秒响应 213 个请求。

图 20 1600-300-40 测试下各方案响应时间

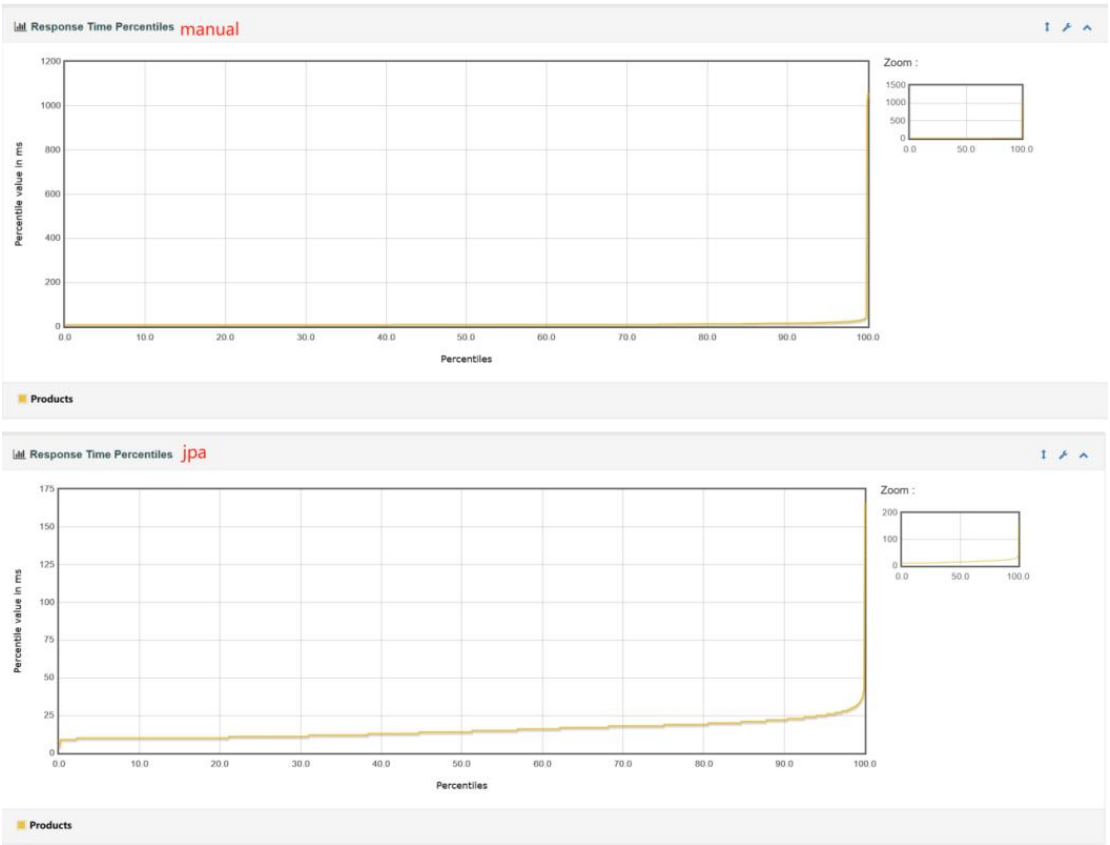

图 21 1600-300-40 测试下各方案响应时间百分比

由图 20、21 可以看出，manual 方案下前一分半时间内系统响应时间出现波动起伏，但最高响应时间也

才只是 34ms，后面四分半内趋于稳定，维持在 10ms 左右；jpa 方案下在整段时间内，系统响应时间都出现波动起伏。manual 方案有 90%的响应请求时间在 15ms 内，而 jpa 方案仅有 60%的响应请求时间在 15ms 内。服务器 CPU 使用率和内存使用率如下图所示：

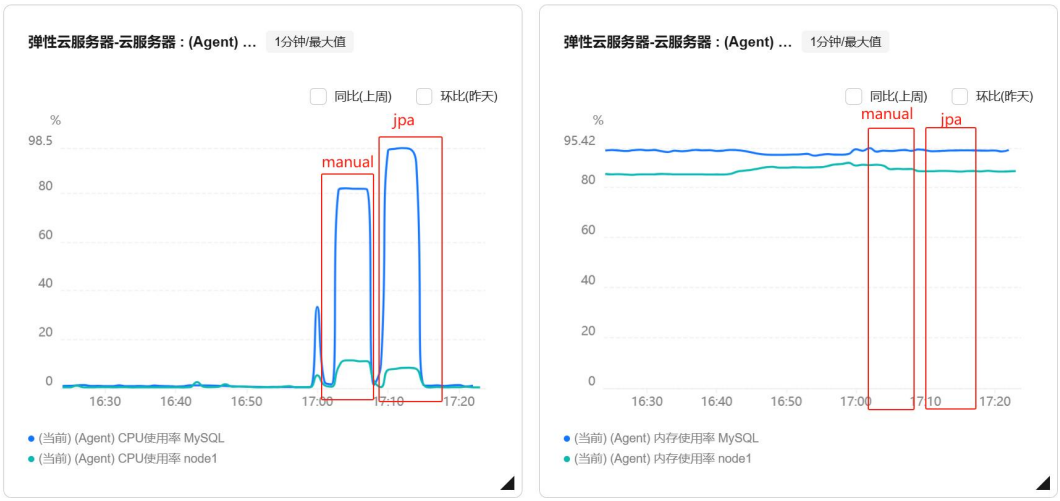

图 22 1600-300-40 测试下服务器 CPU 使用率与内存使用率

由图 22 可以看出，manual 方案下 MySQL 服务器 CPU 使用率达到 80%，jpa 方案下 MySQL 服务器 CPU 使用率达到 98%，二者 MySQL 服务器和 node1 服务器的内存使用率基本相同。

此结果可以初步说明，在高并发请求条件下，jpa 方案对服务器负载的压力高于 manual 方案，jpa 方案的稳定性比 manual 方案的稳定性差，jpa 方案查询效率比 manual 方案低。

3.2.3 固定时间 10s，发送 1200 个线程，循环 2 次实验结果

在本组测试下，如果系统响应请求平均且稳定，则每秒响应 240 个请求。

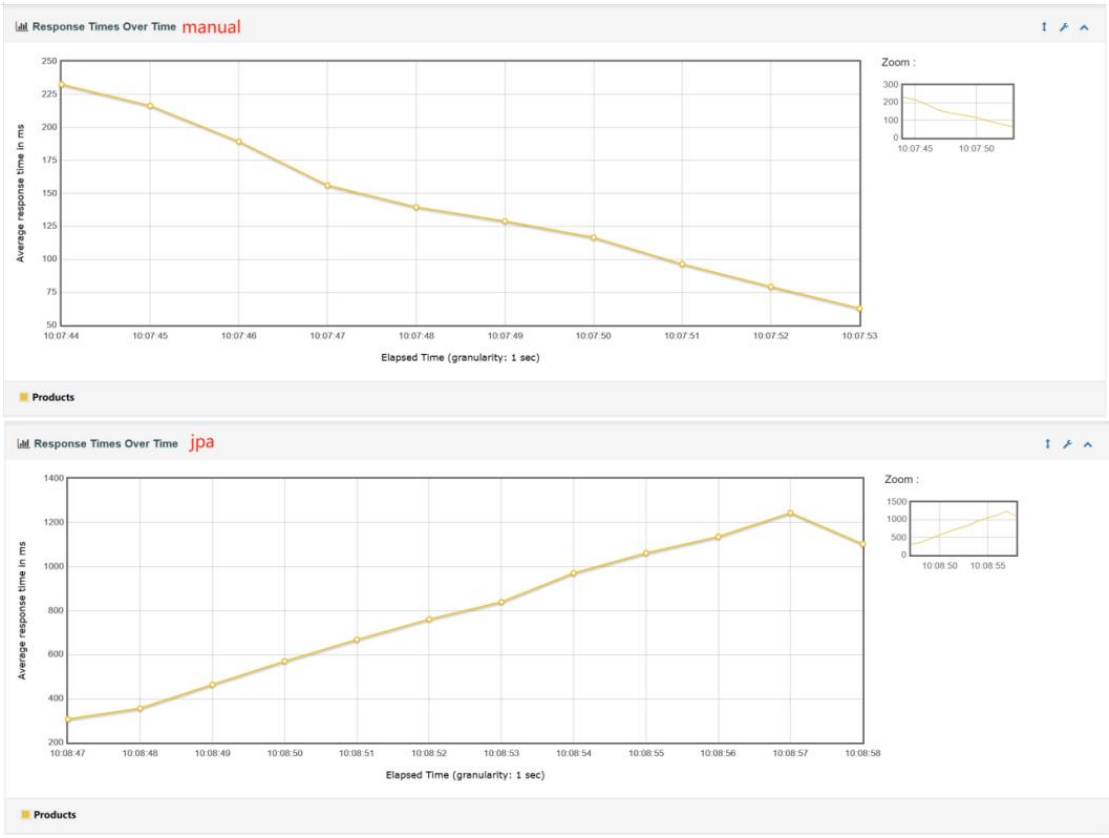

图 23 1200-10-2 测试下各方案响应时间

由图 23 可知，manual 方案的响应时间以直线型下降，最高响应时间才达到 250ms；而 jpa 方案的响应

时间以直线型上升，最低响应时间就已经达到了 307ms。说明 jpa 方案在平均每秒并发 240 个请求下已经到了服务器瓶颈，而 manual 方案在该测试条件下还能较好的响应请求。

由本组实验可以得出结论，在高并发请求条件下，manual 方案能较好且稳定的响应请求，而 jpa 方案响应请求能力大大减弱。

3.3 实验小结

由上述各组实验结果可以得出，在低负载或中等负载下，jpa 方案与 manual 方案的性能基本一致。而在高负载情况下，jpa 方案对服务器负载压力远远高于 manual 方案，jpa 方案的稳定性也低于 manual 方案。

4.总结

Spring Data JPA 性能分析：Spring Data JPA 通过简化 JPA 操作封装了常见的 CRUD 功能，使得查询代码较为简洁，减少了手动配置 SQL 的需求，但这种封装可能在高并发场景下增加了额外的性能开销。在 10 秒和中等并发下，JPA 方案与 manual 即 MyBatis 关联性能相似，表明在低到中等负载下，Spring Data JPA 是一个性能相对稳定且维护简便的选择。但是，由于它生成的 SQL 可能不如手动优化的 SQL 精确，因此在一些复杂查询或者性能敏感的场景下，可能会存在性能损失。

manual 的性能优势：在高并发请求条件下，manual 方案的性能比 jpa 方案性能高，manual 方案下系统响应稳定性高于 jpa 方案。可能其手动管理的 SQL 语句可根据具体需求进行更精确的优化，而 Spring Data JPA 在高并发下会出现一定的性能劣势。这种差距主要是因为 manual 即 MyBatis 映射对 SQL 优化提供了更高的控制权，而 JPA 封装的 CRUD 操作在底层多了一层间接开销。

Spring Data JPA 在开发便捷性和中等负载下的性能稳定性方面具有一定优势，是适合快速开发和数据量适中场景的有效选择。然而，在高并发和高性能需求的场景中，MyBatis 表现出更高的性能和稳定性，尤其在处理复杂查询和大规模数据访问时更为优越。因此，在数据库设计和技术选型时，需结合系统负载和性能需求综合考虑：对于开发效率和代码维护性有较高需求且并发量适中的应用，Spring Data JPA 方案是合适选择；对于查询性能要求严苛、并发量较大的系统，MyBatis 能够在高负载下提供更优的响应时间和系统稳定性。本实验结果显示，合理选择持久化技术并进行有效的数据库索引优化，对系统整体性能具有关键性作用。

参考文献：

[1] 2-6.productdemoaop 工程 https://github.com/xixi115/2-6-JavaEEPlatform/tree/main/Exp5