项目编号：T030SRIP11012

上海交通大學

暑期科研见习岗位

总结报告

岗位名称：Spring 框架在高并发场景中的应用研究

见习时间：2022-07-01 至 2022-09-01

学院(系)：电子信息与电气工程学院

专　　业：自动化

学生姓名：彭俊杰 学号：521021910404

2022 年　9 月　20 日

目录

一、见习目的

学习 Spring 基本结构和原理，学习 Spring Boot 的设计理念和特性，了解缓存技术，完成高并发应用项目的构建，巩固 Java 程序设计语言和 IDEA 使用经验，为今后学习或科研或工作夯实基础。

二、主要工作内容与任务

1.学习 Spring 基本结构和原理，了解控制反转和 AOP 思想；

2.学习 Spring Boot 的设计理念和特性，掌握基于 Spring Boot 的项目开发方法；

3.了解缓存技术，掌握分布式缓存 Redis 和 RabbitMQ 消息中间件的用法，完成高并发应用项目的构建；

4.撰写 Spring 学习总结文档。

三、见习成果与收获

1、基础知识——Spring 基本结构和原理

Spring 框架是一个开源的 Java 平台。它最初由 Rod Johnson 编写，并于 2003 年 6 月首次在 Apache 2.0 许可下发布。Spring 是企业 Java 最流行的应用程序开发框架，实际上已经成为 JavaEE 项目开发标准。Spring 大约有 20 个模块，这些组件被分别整合在核心容器(Core Container)、AOP (Aspect Oriented Programming)和设备支持(Instrumentation)、数据访问及集成 (Data Access/Integeration)、Web、报文发送 (Messaging)、Test 这 6 个模块集合中。

2、基础知识——控制反转

IoC(Inversion of Control ，控制反转）是 Spring 最核心的要点，并且贯穿始终。IoC 并不是一门技术,而是一种设计思想。在 Spring 框架中,实现控制反转的是 Spring IoC 容器，由容器来控制对象的生命周期和业务对象之间的依赖关系，而不是像传统方式（ new 对象）那样由代码来直接控制。程序中所有的对象都会在 Spring IoC 容器中登记，告诉容器该对象是什么类型、需要依赖什么，然后 IoC 容器会在系统运行到适当的时候把它要的对

象主动创建好，同时也会把该对象交给其他需要它的对象。也就是说，控制对象生存周期的不再是引用它的对象，而是由 Spring IoC 容器来控制所有对象的创建、销毁。对于某个具体的对象而言，以前是它控制其他对象，现在是所有对象都被 Spring IoC 容器所控制，所以这叫控制反转。

控制反转最直观的表达就是，IoC 容器让对象的创建不用去新建（new）了，而是由 Spring 自动生产，使用 Java 的反射机制，根据配置文件在运行时动态地去创建对象以及管理对象,并调用对象的方法。控制反转的本质是控制权由应用代码转到了外部容器(IoC 容器)。控制权的转移就是反转。控制权的转移带来的好处是降低了业务对象之间的依赖程度，即实现了解稠。

DI (Dependency Injection，依赖注入）是 IoC 的一个别名，其实两者是同一个概念，只是从不同的角度描述罢了——IoC 是一种思想，而 DI 是一种具体的技术实现手段。

3、基础知识——AOP 思想

AOP （ Aspect-OrientedProgramming ，面向切面编程）不是一种新的技术，也不是一种框架，而是一种编程思想，可以理解为 OOP(Object-Oriented Programing ，面向对象编程）的补充和完善。OOP 引入封装、继承和多态性等概念来建立一种对象层次结构，用以模拟公共行为的一个集合。 AOP 技术则恰恰相反，它利用一种称为“横切”的技术，剖解开封装的对象内部，并将那些影响了多个类的公共行为封装到一个可重用模块，并将其命名为“方面”。所谓“方面”,简单地说就是将那些与业务无关却为业务模块所共同调用的逻辑或责任封装起来，便于减少系统的重复代码，降低模块间的耦合度，并有利于未来的可操作性和可维护性。 AOP 代表的是一个横向的关系，如果说“对象”是一个空心的圆柱体，其中封装的是对象的属性和行为，那么面向方面编程的方法就像一把利刃，将这些空心圆柱体剖开，以获得其内部的消息，而剖开的切面也就是所谓的“方面”了。

4、基础知识——Spring 缓存抽象

缓存抽象的核心是将缓存应用于 Java 方法。一种通用的场景是，在每次调用方法时，如里缓存已存在，则返回缓存的结果，而不必调用实际的方法；如果缓存不存在，则执行方法的逻辑，并将方法的返回值缓存起来，以便下次调用该方法时直接返回。

与 Spring 中的其他服务一样，缓存服务是一种抽象，不是缓存实现，所以需要使用实际的存储器来存储缓存数据。也就是说，抽象不必编写缓存逻辑，但是也不提供实际的数据存储器。

org.springframework.cache.Cache、org.springframework.cache.CacheManager 两个接口代表了 Spring 的缓存抽象。

Cache 是一个缓存单位，每个 Cache 对象都有一个 name 属性，也可以称为 cacheName ，用于唯一标识一个 Cache 。每个 Cache 内部维护了 n 个 key-value 对，因

为缓存的本质就是 key-value 对，这些 key-value 对就是缓存的实际数据。

CacheManager 意为缓存管理器，用来管理多个 Cache ，其内部维护了一个 Map 结构，用于管理多个 Cache 对象。 Map 的 key 存储的是 Cache 对象的 name 属性， Map 的 value 存储的是 Cache 对象本身。

5、高并发项目搭建——秒杀系统相关组件

（1）Redis

在 Redis 的安装目录下打开 cmd 窗口，使用 redis-server.exe redis.windows.conf 命令来启动 Redis 服务

（2）Redis 客户端（可视化工具）Redis Desktop Manager

（3）Erlang

RabbitMQ 服务端代码是使用并发式语言 Erlang 编写的，安装 RabbitMQ 的前提是安装 Erlang。

（4）RabbitMQ

Erlang 和 RabbitMQ 的版本一定得是匹配的，版本对应图如下。

RabbitMQ version	Minimum required Erlang/OTP	Maximum supported Erlang/OTP	Notes
3.10.7 3.10.6 3.10.5 3.10.4 3.10.2 3.10.1 3.10.0	24.2	25.0	• Erlang 25 is the recommended series. • Erlang 23 support was discontinued on July 31st, 2022.
3.9.22	24.2	24.3	• Erlang 23 support was discontinued on July 31st, 2022.
3.9.21 3.9.20 3.9.19 3.9.18 3.9.17 3.9.16 3.9.15	23.3	24.3	• Erlang 24.3 introduces LDAP client changes that are breaking for projects compiled on earlier releases (including RabbitMQ). RabbitMQ 3.9.15 is the first release to support Erlang 24.3. • Erlang 23 support was discontinued on July 31st, 2022.
3.9.14 3.9.13 3.9.12 3.9.11 3.9.10 3.9.9 3.9.8 3.9.7 3.9.6 3.9.5 3.9.4	23.2	24.2	• Erlang/OTP 24 support announcement • Erlang 24 was released on May 12, 2021 • Some community plugins and tools may be incompatible with Erlang 24

（5）MySQL8.0

安装后，使用客户端工具(MySQL Workbench)来连接数据库

（6）Navicat 12

管理数据库的工具

（7）IDEA

使用 2020.2 版本，更新的版本 REST 客户端被停用

6、生成唯一 ID

在高并发项目中，我们经常涉及系统唯一 ID 的生成，比如常用的订单号 ID。主要有 UUID，雪花算法生成唯一 ID 等等。我们选择学法算法生成 ID。

（1）JDK 自带的 UUID

```
public class UUIDUtil {

    public static String generateUUID(){
        UUID uuid = UUID.randomUUID();
        return uuid.toString();
    }

    public static void main(String[] args) {
        System.out.println(generateUUID());
    }
```

运行结果如图 `74ab2fd0-c076-4247-adbb-167410451a87`

该方法无法保证有序（ID 递增）

（2）雪花算法生成唯一 ID

```
public static void main(String[] args) {
    System.out.println("开始时间:"+System.currentTimeMillis());
    long startTime = System.nanoTime();
    for (int i = 0; i < 50; i++) {
        long id = SnowflakeIdWorker.generateId();
        System.out.println(id);
    }
    //计算生成ID耗时
    System.out.println("耗时: "+(System.nanoTime() - startTime) / 1000000 + "ms");
}
```

运行 main 方法，生成 50 个 ID 如下（截取片段）：

```
726400980757270528
726400980757270529
726400980757270530
726400980757270531
726400980757270532
726400980757270533
726400980757270534
726400980757270572
726400980757270573
726400980757270574
726400980757270575
726400980757270576
726400980757270577
耗时：0ms
```

可以看到，雪花算法生成的 ID 有序，可读，更符合秒杀项目的需要

7、秒杀场景实战

（1）架构思想

我们知道每年的淘宝“双十一”、12306 抢票软件都会面临超大流量，当系统流量激增时，如何让系统稳定，快速地运行是技术人员的一大难题。缓存与消息队列技术在电商系统中用得很多，因为电商系统流量多，流量多服务器压力就大，就需要借助缓存来减轻服务器压力。瞬时压力大可以用消息队列起到削峰的作用，因为消息队列的显著特点就是异步、排队，并且消息可以持久化，不用担心消息丢失。缓存服务与消息队列服务都可以搭建高可用集群。

架构思想主要有接口限流、高性能、服务降级、水平拓展、Redis 预扣库存、异步下单等。接口限流即过滤无效请求，防止渗透到数据库；高性能即将 CPU、内存等有限的系统资源分配给主要的请求；服务降级即做好服务器宕机后的备用服务。重点掌握 Redis 预扣库存、异步下单。

（2）实践演示原理

该高并发的案例主要实现的技术有：

Redis 预扣库存；

Redis 通过执行 Lua 脚本保证原子性；

使用消息队列 RabbitMQ 异步下单；

使用雪花算法生成不重复订单号；

项目整体框架采用 Spring Boot + MyBatis。

其中，预扣库存就是用户抢到商品后库存减 1 的操作，为什么要预扣库存呢？一般减库存有下单减库存和付款减库存两种方案， Redis 预扣库存是介于这两者之间的折中方案。很多请求进来都需要后台查询库存，这是一个频繁读的场景。可以使用 Redis 来缓存商品库存，具体做法是在秒杀活动开始之前，将秒杀商品的库存信息加载到 Redis 中，秒杀活动开始后，请求进来，先从 Redis 中查询商品库存，如果大于 0，则代表抢到商品，Redis 中库存减 1。这里需要注意的是，虽然 Redis 的单线程特性保证命令都是原子性操作，但是查询库存和扣除库存是两个命令，命令之间不具有原子性，如果不采取任何措施，在高并发的情况下就有可能出现超卖问题。我们可以利用 Lua 脚本将两个命令合并到一起，最后让 Redis 执行 Lua 脚本来保持原子性。另外，超卖问题的本质会导致 MySQL 的库存小于 0，因此我们可以利用如下 SQL 语句防止库存小于 0: update goods set amount - amount -1 where id =? and amount >0。由于 update 语句会给该记录加上排它锁，因此不会存在事务并发问题。还可以将数据库的 amount 字段设为无符号，如果插入负数，就会抛出异常，service 捕获异常后做容错处理。

（3）项目搭建

① 首先在 pom.xml 中引入 Spring Boot、MyBatis、Redis、RabbitMQ、commons-lang3 等相关依赖。

```xml
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.mybatis.spring.boot</groupId>
            <artifactId>mybatis-spring-boot-starter</artifactId>
            <version>2.1.3</version>
        </dependency>
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <scope>runtime</scope>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-redis</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-amqp</artifactId>
        </dependency>
        <dependency>
            <groupId>org.apache.httpcomponents</groupId>
            <artifactId>httpclient</artifactId>
        </dependency>
        <dependency>
            <groupId>org.apache.commons</groupId>
            <artifactId>commons-lang3</artifactId>
        </dependency>
    </dependencies>
</project>
```

② 我们在数据库中设计两张表 book 和 book_order。创建数据库表、初始化 book 表的命令在 springboot-echcache-redis-secondskill.sql 中，如下图：

```sql
CREATE TABLE book (
  id     bigint(20) UNSIGNED NOT NULL AUTO_INCREMENT,
  name     varchar(255) CHARACTER SET utf8mb4 COLLATE utf8mb4_general_ci NULL DEFAULT NULL,
  price     decimal(10, 2) NULL DEFAULT NULL,
  amount     int(11) UNSIGNED NULL DEFAULT NULL,
  seconds_kill     int(11) NULL DEFAULT NULL,
  PRIMARY KEY (    id    ) USING BTREE
);
CREATE TABLE book_order (
  id     bigint(20) UNSIGNED NOT NULL AUTO_INCREMENT,
  order_id     varchar(255) CHARACTER SET utf8mb4 COLLATE utf8mb4_general_ci NULL DEFAULT NULL,
  book_id     bigint(20) NULL DEFAULT NULL,
  money     decimal(10, 2) NULL DEFAULT NULL,
  user_id     int(11) NULL DEFAULT NULL,
  status     int(11) NULL DEFAULT NULL,
  create_time     datetime(0) NULL DEFAULT NULL,
  PRIMARY KEY (    id    ) USING BTREE
);

INSERT INTO     book     VALUES (1, '三国演义', 200.00, 206_666, NULL);
INSERT INTO     book     VALUES (2, '水浒传', 125.90, 185_869, NULL);
INSERT INTO     book     VALUES (3, '西游记', 130.00, 8899, NULL);
INSERT INTO     book     VALUES (4, '红楼梦', 69.88, 289, NULL);
INSERT INTO     book     VALUES (5, '数据结构与算法', 79.50, 1200, NULL);
INSERT INTO     book     VALUES (6, '重新定义springcloud实战', 188.99, 55700, NULL);
INSERT INTO     book     VALUES (7, 'Hadoop权威指南', 99.00, 338, NULL);
INSERT INTO     book     VALUES (8, 'mysql技术内幕', 85.00, 1208, NULL);
INSERT INTO     book     VALUES (9, 'kafka权威指南', 120.00, 2865, NULL);
INSERT INTO     book     VALUES (10, '大话数据结构', 89.00, 6868, NULL);
INSERT INTO     book     VALUES (11, '三国演义', 69.00, 0, 1);
INSERT INTO     book     VALUES (12, '水浒传', 59.00, 0, 1);
INSERT INTO     book     VALUES (13, '西游记', 59.00, 0, 1);
INSERT INTO     book     VALUES (14, '红楼梦', 59.00, 0, 1);
```

Book 表记录在 Navicat 中如下图：

id	name	price	amount	seconds_kill
1	三国志2	200	206666	(Null)
2	水浒传	125.9	185869	(Null)
3	西游记	130	8899	(Null)
4	红楼梦	69.88	289	(Null)
5	数据结构	79.5	1200	(Null)
6	重新定义	188.99	55700	(Null)
7	Hadoop	99	338	(Null)
8	mysql技	85	1208	(Null)
9	kafka权威	120	2865	(Null)
10	大话数据	89	6868	(Null)
11	三国演义	69	0	1
12	水浒传	59	0	1
13	西游记	59	0	1
14	红楼梦	59	0	1

③ amount 字段表示库存， seconds kill 字段如果是 1，就表示秒杀商品。为了方便，这里写 7、监听 Spring 上下文刷新事件（也就是项目启动）的监听器，让秒杀商品在项目启动的时候提首载到 Redis 缓存中。监听器代码在 ApplicationEventListener . java 中。

```java
@Component
public class ApplicationEventListener {
    @Autowired
    private RedisTemplate<String,Object> redisTemplate;
    @Autowired
    private BookMapper bookMapper;
    public static final String SECONDS_BOOK_KEY = "secondsKillBooks";

    @EventListener
    public void contextRefreshedEventListener(ContextRefreshedEvent contextRefreshedEvent) {
        if(redisTemplate == null){return;}
        System.out.println("往redis中加载秒杀商品开始！");
        Book params = new Book();
        params.setSecondsKill(1);
        // 获取操作redis hash类型的操作类
        HashOperations<String, Object, Object> hashOperations = redisTemplate.opsForHash();
        // 查询秒杀商品，并通过流的方式转化为Map,Map的key为商品ID，Map的value为商品库存
        Map<String,Integer> map = bookMapper.selectBooks(params).stream().collect(
                Collectors.toMap(t-> String.valueOf(t.getId()),Book::getAmount));
        // 往redis中加载数据
        hashOperations.putAll(SECONDS_BOOK_KEY,map);

        System.out.println("往redis中加载秒杀商品成功！");
    }
}
```

④ 秒杀的核心操作是下单，定义的下单的 OrderController 代码如下：

```java
@RestController
public class OrderController {

    @Autowired
    private OrderService orderService;

    @PostMapping("/order/add")
    public String addOrder(@RequestParam(value = "bookId") Long bookId,
                           @RequestParam(value = "userId") Integer userId){
        boolean success = orderService.addOrder(bookId,userId);
        if(success){
            return "下单成功！";
        }
        return "下单失败";
    }
}
```

OrderServiceImpl 代码中有两个 addOrder 方法，Controller 调用的异步下单代码如下：

```java
@Override
public boolean addOrder(Long bookId,Integer userId) {

    DefaultRedisScript<String> redisScript = new DefaultRedisScript();
    // 设置lua脚本
    redisScript.setScriptSource(new ResourceScriptSource(new ClassPathResource("stock_simple.lua")));
    redisScript.setResultType(String.class);
    List<String> keys = new ArrayList<>();
    keys.add(ApplicationEventListener.SECONDS_BOOK_KEY);
    StringRedisSerializer serializer = new StringRedisSerializer();
    // 执行lua脚本
    String res = redisTemplate.execute(redisScript,serializer,serializer,keys, bookId.toString());
    if(!"1".equals(res)){
        return false;
    }
```

```java
    // 获取库存信息并修改库存，不具备原子性
    /*Integer amount = (Integer)(redisTemplate.opsForHash().get("secondsKillBooks", String.valueOf(bookId)));
    if(amount == null || amount <= 0){
        return false;
    }
    //redis预扣库存
    Long value = redisTemplate.opsForHash().increment("secondsKillBooks", String.valueOf(bookId), -1);
    if(value <= 0){
        // 如果扣完后库存为0，则删除当前商品
        redisTemplate.opsForHash().delete("secondsKillBooks", String.valueOf(bookId));
    }*/

    Order order = new Order();
    order.setOrderId(String.valueOf(SnowflakeIdWorker.generateId()));
    order.setBookId(bookId);
    order.setMoney(bookMapper.selectBookById(bookId).getPrice());
    order.setUserId(userId);
    order.setStatus(0);
    order.setCreateTime(new Date());
    orderProducer.sendOrder(order);
    return true;
}
```

其中 lua 脚本如下：

```
local val=redis.call('hget',KEYS[1],ARGV[1]);
-- 判断是否为空
if(val == false) then
-- 库存不存在
return '0'
elseif(tonumber(val) <= 0)
then
-- 库存为0
redis.call('hdel',KEYS[1],ARGV[1]);
return '0'
else
-- 下单成功
redis.call('hincrby',KEYS[1],ARGV[1],-1);
return '1'
end
```

⑤ 异步下单代码中，new 出的订单对象通过 orderProducer.sebdorder 发出的消息需要在消费者中去接收消息。

这一步要用到 RabbitMQ，需要配置交换机。在 OrderDirectExchangeConfig.java 中实现。

```
@Configuration
public class OrderDirectExchangeConfig {
    public static final String QUEUE_NAME = "order-queue";
    public static final String EXCHANGE_NAME = "order-direct-exchange";
    public static final String QUEUE_EXCHANGE_KEY = "order-queue-exchange-key";


    /**
     * 创建队列
     */
    @Bean
    public Queue orderQueue() { return new Queue(QUEUE_NAME, durable: true); }

    /**
     * 创建交换机
     */
    @Bean
    DirectExchange orderDirectExchange() { return new DirectExchange(EXCHANGE_NAME, durable: true, autoDelete: false); }
```

```
    /**
     * 将队列和直接型交换机绑定，并设置匹配键
     */
    @Bean
    Binding bindQueueExchange(){
        return BindingBuilder.bind(orderQueue())
                .to(orderDirectExchange())
                .with(QUEUE_EXCHANGE_KEY);
    }

}
```

生产者实现发送消息的代码在 OrderProducer.java 中，如下图所示：

```java
@Service
public class OrderProducer {
    @Autowired
    private RabbitTemplate rabbitTemplate;

    public boolean sendOrder(Order order){
        try {
            rabbitTemplate.convertAndSend(OrderDirectExchangeConfig.EXCHANGE_NAME,
                    OrderDirectExchangeConfig.QUEUE_EXCHANGE_KEY,new ObjectMapper().writeValueAsString(order));
        } catch (JsonProcessingException e) {
            e.printStackTrace();
        }
        return true;
    }
}
```

消费者实现收到消息的代码在 OrderConsumer.java 中，调用 OrderServiceImpl 中另一个真实下单 addOrder 方法。

```java
@Override
public boolean addOrder(Order order) {
    int result;
    result= orderMapper.insertOrder(order);
    if(result > 0){
        Book params = new Book();
        params.setId(order.getBookId());
        params.setAmount(-1);
        try {
            result = bookMapper.updateBook(params);
            if(result > 0){
                return true;
            }
        } catch (Exception e) {
            System.out.println("库存不足，无法下单!");
        }

    }
    throw new RuntimeException("更新库存失败！");
}
```

（4）项目运行

① 清空 book_order，初始化秒杀商品数目

在 Navicat 新建查询，输入代码 TRUNCATE table book_order，运行。

```
TRUNCATE table book_order
> OK
> 时间: 0.025s
```

运行成功，刷新 book_order 表，可以看到 book_order 表已经清空，如下图所示

id	order_id	book_id	money	user_id	status	create_time
(Null)	(Null)	(Null)	(Null)	(Null)	(Null)	(Null)

四件“秒杀”商品，每件数量定为 10。

11	三国演义	69	10	1
12	水浒传	59	10	1
13	西游记	59	10	1
14	红楼梦	59	10	1

② 运行 Redis、RabbitMQ、MySQL 服务

RabbitMQ	7188	RabbitMQ	正在运行
MySQL80	7004	MySQL80	正在运行
Redis	7240	Redis	正在运行

③ 在 IDEA 中连接数据库 springboot_ehcache_redis_secondskill，运行 SpringbootSecondskillApplication.java（服务端）。

运行界面如下：

```
 .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v2.2.6.RELEASE)

2022-09-12 00:50:08.020  INFO 22696 --- [           main] c.b.s.SpringbootSecondskillApplication   : Starting SpringbootSecondskillApplication on LAPTOP-F9AG32D7 with PID 22696 (C:\Users\12345\Desktop\源代码\springboot-second
2022-09-12 00:50:08.022  INFO 22696 --- [           main] c.b.s.SpringbootSecondskillApplication   : No active profile set, falling back to default profiles: default
2022-09-12 00:50:08.577  INFO 22696 --- [           main] .s.d.r.c.RepositoryConfigurationDelegate : Multiple Spring Data modules found, entering strict repository configuration mode!
2022-09-12 00:50:08.579  INFO 22696 --- [           main] .s.d.r.c.RepositoryConfigurationDelegate : Bootstrapping Spring Data Redis repositories in DEFAULT mode.
2022-09-12 00:50:08.607  INFO 22696 --- [           main] .s.d.r.c.RepositoryConfigurationDelegate : Finished Spring Data repository scanning in 13ms. Found 0 Redis repository interfaces.
2022-09-12 00:50:09.053  INFO 22696 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with port(s): 8080 (http)
2022-09-12 00:50:09.062  INFO 22696 --- [           main] o.apache.catalina.core.StandardService   : Starting service [Tomcat]
2022-09-12 00:50:09.062  INFO 22696 --- [           main] org.apache.catalina.core.StandardEngine  : Starting Servlet engine: [Apache Tomcat/9.0.33]
2022-09-12 00:50:09.435  INFO 22696 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebApplicationContext
2022-09-12 00:50:09.435  INFO 22696 --- [           main] o.s.web.context.ContextLoader            : Root WebApplicationContext: initialization completed in 1377 ms
2022-09-12 00:50:10.215  INFO 22696 --- [           main] o.s.s.concurrent.ThreadPoolTaskExecutor  : Initializing ExecutorService 'applicationTaskExecutor'
2022-09-12 00:50:10.484  INFO 22696 --- [           main] o.s.a.r.c.CachingConnectionFactory       : Attempting to connect to: [localhost:5672]
2022-09-12 00:50:10.517  INFO 22696 --- [           main] o.s.a.r.c.CachingConnectionFactory       : Created new connection: rabbitConnectionFactory#54c622a7:0/SimpleConnection@1552da16 [delegate=amqp://guest@127.0.0.1:5672/
往redis中加载秒杀商品开始!
2022-09-12 00:50:10.585  INFO 22696 --- [           main] com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Starting...
2022-09-12 00:50:10.745  INFO 22696 --- [           main] com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Start completed.
2022-09-12 00:50:10.751 DEBUG 22696 --- [           main] c.b.s.mapper.BookMapper.selectBooks      : ==>  Preparing: select * from book WHERE seconds_kill=?
2022-09-12 00:50:10.772 DEBUG 22696 --- [           main] c.b.s.mapper.BookMapper.selectBooks      : ==> Parameters: 1(Integer)
2022-09-12 00:50:10.798 DEBUG 22696 --- [           main] c.b.s.mapper.BookMapper.selectBooks      : <==      Total: 4
2022-09-12 00:50:10.987  INFO 22696 --- [           main] io.lettuce.core.EpollProvider            : Starting without optional epoll library
2022-09-12 00:50:10.988  INFO 22696 --- [           main] io.lettuce.core.KqueueProvider           : Starting without optional kqueue library
往redis中加载秒杀商品成功!
2022-09-12 00:50:11.122  INFO 22696 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8080 (http) with context path ''
2022-09-12 00:50:11.124  INFO 22696 --- [           main] c.b.s.SpringbootSecondskillApplication   : Started SpringbootSecondskillApplication in 3.438 seconds (JVM running for 4.245)
```

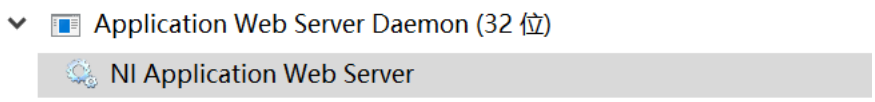
若出现报错“Web server failed to start. Port 8080 was already in use.”键盘输入“win+r”，打开 cmd 命令窗口，输入 netstat -ano|findstr "8080"，回车，如下图：

```
C:\Users\12345>netstat -ano|findstr "8080"
  TCP    0.0.0.0:8080           0.0.0.0:0              LISTENING       12232
  TCP    10.183.23.243:2171     125.39.196.233:8080    ESTABLISHED     17788

C:\Users\12345>
```

找到第一个为占用端口的进程，输入 netstat -ano|findstr "12232"，回车，如下图显示：

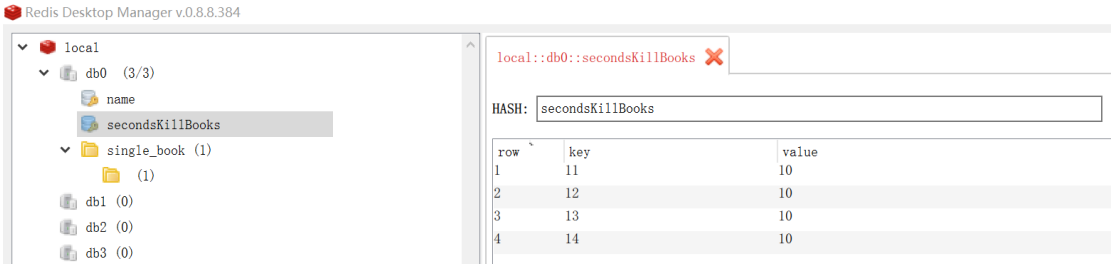
```
C:\Users\12345>netstat -ano|findstr "8080"
  TCP    0.0.0.0:8080           0.0.0.0:0              LISTENING       12232
  TCP    10.183.23.243:2171     125.39.196.233:8080    ESTABLISHED     17788

C:\Users\12345>tasklist|findstr "12232"
ApplicationWebServer.exe     12232 Services                   0      3,196 K

C:\Users\12345>
```

打开“任务管理器”，找到该进程，然后结束进程：



重新运行 SpringbootSecondskillApplication.java，项目启动成功。

④ 在 Redis 图形界面客户端——Redis Desktop Manager 中，可监测到“秒杀”商品实时数量：



该 HASH 类型里面，key 对应 book 的 id，value 对应数量 amount。

id	name	price	amount	seconds_kill
11	三国演义	69	10	1
12	水浒传	59	10	1
13	西游记	59	10	1
14	红楼梦	59	10	1

⑤ 在 SecondsKillClient.java 中，模拟客户端发出大量请求。请求数量设置成 10000，运行该客户端。

```
public class SecondsKillClient {
    // 请求数量
    private static final int HTTP_REQUEST_COUNT = 10000;
```

此时任务管理器显示 CPU 利用率极高。

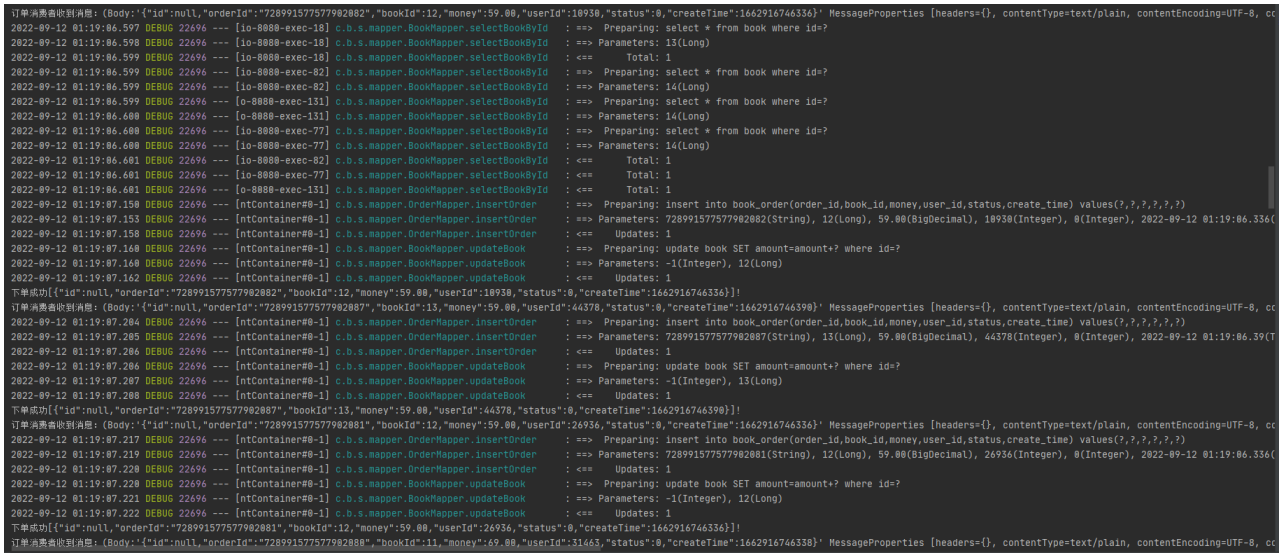
数秒之后， SecondsKillClient 线程发起请求，客户端请求成功，节选截图如下：

```
Apache-HttpClient/4.5.12 (Java/1.8.0_131)[\r][\n]"
01:19:24.408 [线程1200号! ] DEBUG org.apache.http.wire - http-outgoing-9997 >> "Accept-Encoding: gzip,deflate[\r][\n]"
01:19:24.408 [线程1200号! ] DEBUG org.apache.http.wire - http-outgoing-9997 >> "[\r][\n]"
01:19:24.408 [线程1200号! ] DEBUG org.apache.http.wire - http-outgoing-9997 >> "bookId=14&userId=63302"
01:19:24.324 [线程7911号! ] DEBUG org.apache.http.headers - http-outgoing-4789 << HTTP/1.1 200
01:19:24.408 [线程7911号! ] DEBUG org.apache.http.headers - http-outgoing-4789 << Content-Type: text/plain;charset=UTF-8
01:19:24.408 [线程7911号! ] DEBUG org.apache.http.headers - http-outgoing-4789 << Content-Length: 12
01:19:24.408 [线程7911号! ] DEBUG org.apache.http.headers - http-outgoing-4789 << Date: Sun, 11 Sep 2022 17:19:23 GMT
01:19:24.408 [线程7911号! ] DEBUG org.apache.http.headers - http-outgoing-4789 << Keep-Alive: timeout=60
01:19:24.408 [线程7911号! ] DEBUG org.apache.http.headers - http-outgoing-4789 << Connection: keep-alive
01:19:24.408 [线程7911号! ] DEBUG org.apache.http.impl.execchain.MainClientExec - Connection can be kept alive for 60000 MILLISECONDS
01:19:24.408 [线程7911号! ] DEBUG org.apache.http.impl.conn.PoolingHttpClientConnectionManager - Connection manager is shutting down
01:19:24.408 [线程7911号! ] DEBUG org.apache.http.impl.conn.DefaultManagedHttpClientConnection - http-outgoing-4789: Shutdown connection
01:19:24.408 [线程7911号! ] DEBUG org.apache.http.impl.conn.PoolingHttpClientConnectionManager - Connection manager shut down
01:19:24.409 [线程6723号! ] DEBUG org.apache.http.wire - http-outgoing-9570 << "HTTP/1.1 200 [\r][\n]"
01:19:24.409 [线程6723号! ] DEBUG org.apache.http.wire - http-outgoing-9570 << "Content-Type: text/plain;charset=UTF-8[\r][\n]"
01:19:24.409 [线程6723号! ] DEBUG org.apache.http.wire - http-outgoing-9570 << "Content-Length: 12[\r][\n]"
01:19:24.409 [线程6723号! ] DEBUG org.apache.http.wire - http-outgoing-9570 << "Date: Sun, 11 Sep 2022 17:19:24 GMT[\r][\n]"
01:19:24.409 [线程6723号! ] DEBUG org.apache.http.wire - http-outgoing-9570 << "Keep-Alive: timeout=60[\r][\n]"
01:19:24.409 [线程6723号! ] DEBUG org.apache.http.wire - http-outgoing-9570 << "Connection: keep-alive[\r][\n]"
01:19:24.409 [线程6723号! ] DEBUG org.apache.http.wire - http-outgoing-9570 << "[\r][\n]"
01:19:24.409 [线程6723号! ] DEBUG org.apache.http.wire - http-outgoing-9570 << "[0xe4][0xb8][0x8b][0xe5][0x8d][0x95][0xe5][0xa4][0xb1][0xe8][0xb4][0xa5]"
01:19:24.409 [线程6723号! ] DEBUG org.apache.http.headers - http-outgoing-9570 << HTTP/1.1 200
01:19:24.409 [线程6723号! ] DEBUG org.apache.http.headers - http-outgoing-9570 << Content-Type: text/plain;charset=UTF-8
01:19:24.409 [线程6723号! ] DEBUG org.apache.http.headers - http-outgoing-9570 << Content-Length: 12
01:19:24.409 [线程6723号! ] DEBUG org.apache.http.headers - http-outgoing-9570 << Date: Sun, 11 Sep 2022 17:19:24 GMT
01:19:24.409 [线程6723号! ] DEBUG org.apache.http.headers - http-outgoing-9570 << Keep-Alive: timeout=60
01:19:24.409 [线程6723号! ] DEBUG org.apache.http.headers - http-outgoing-9570 << Connection: keep-alive
01:19:24.409 [线程6723号! ] DEBUG org.apache.http.impl.execchain.MainClientExec - Connection can be kept alive for 60000 MILLISECONDS
01:19:24.409 [线程6723号! ] DEBUG org.apache.http.impl.conn.PoolingHttpClientConnectionManager - Connection manager is shutting down
01:19:24.409 [线程6723号! ] DEBUG org.apache.http.impl.conn.DefaultManagedHttpClientConnection - http-outgoing-9570: Shutdown connection
```

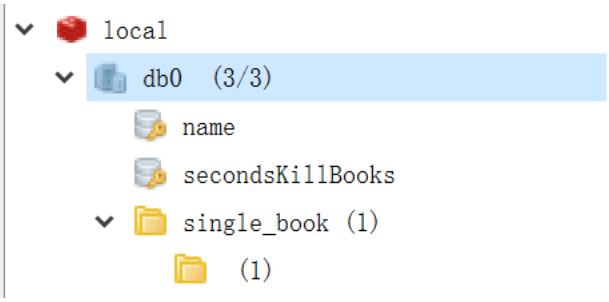
```
01:19:24.577 [线程6044号! ] DEBUG org.apache.http.wire - http-outgoing-7558 << "HTTP/1.1 200 [\r][\n]"
01:19:24.577 [线程6044号! ] DEBUG org.apache.http.wire - http-outgoing-7558 << "Content-Type: text/plain;charset=UTF-8[\r][\n]"
01:19:24.577 [线程6044号! ] DEBUG org.apache.http.wire - http-outgoing-7558 << "Content-Length: 12[\r][\n]"
01:19:24.577 [线程6044号! ] DEBUG org.apache.http.wire - http-outgoing-7558 << "Date: Sun, 11 Sep 2022 17:19:24 GMT[\r][\n]"
01:19:24.577 [线程6044号! ] DEBUG org.apache.http.wire - http-outgoing-7558 << "Keep-Alive: timeout=60[\r][\n]"
01:19:24.577 [线程6044号! ] DEBUG org.apache.http.wire - http-outgoing-7558 << "Connection: keep-alive[\r][\n]"
01:19:24.577 [线程6044号! ] DEBUG org.apache.http.wire - http-outgoing-7558 << "[\r][\n]"
01:19:24.577 [线程6044号! ] DEBUG org.apache.http.wire - http-outgoing-7558 << "[0xe4][0xb8][0x8b][0xe5][0x8d][0x95][0xe5][0xa4][0xb1][0xe8][0xb4][0xa5]"
01:19:24.577 [线程6044号! ] DEBUG org.apache.http.headers - http-outgoing-7558 << HTTP/1.1 200
01:19:24.577 [线程6044号! ] DEBUG org.apache.http.headers - http-outgoing-7558 << Content-Type: text/plain;charset=UTF-8
01:19:24.577 [线程6044号! ] DEBUG org.apache.http.headers - http-outgoing-7558 << Content-Length: 12
01:19:24.577 [线程6044号! ] DEBUG org.apache.http.headers - http-outgoing-7558 << Date: Sun, 11 Sep 2022 17:19:24 GMT
01:19:24.577 [线程6044号! ] DEBUG org.apache.http.headers - http-outgoing-7558 << Keep-Alive: timeout=60
01:19:24.577 [线程6044号! ] DEBUG org.apache.http.headers - http-outgoing-7558 << Connection: keep-alive
01:19:24.577 [线程6044号! ] DEBUG org.apache.http.impl.execchain.MainClientExec - Connection can be kept alive for 60000 MILLISECONDS
01:19:24.577 [线程6044号! ] DEBUG org.apache.http.impl.conn.PoolingHttpClientConnectionManager - Connection manager is shutting down
01:19:24.577 [线程6044号! ] DEBUG org.apache.http.impl.conn.DefaultManagedHttpClientConnection - http-outgoing-7558: Shutdown connection
01:19:24.577 [线程6044号! ] DEBUG org.apache.http.impl.conn.PoolingHttpClientConnectionManager - Connection manager shut down

进程已结束,退出代码0
```

后台服务的日志节选如下，可看到下单成功的日志。

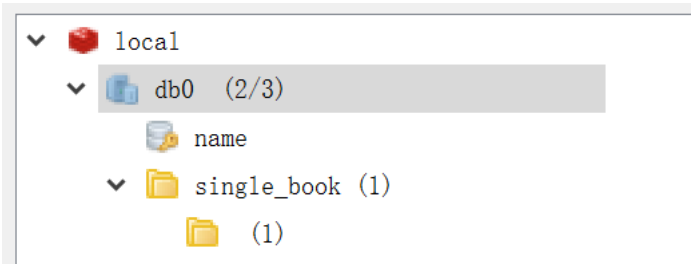
```
2022-09-12 01:19:06.082  INFO 22696 --- [o-8080-exec-159] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring DispatcherServlet 'dispatcherServlet'
2022-09-12 01:19:06.082  INFO 22696 --- [o-8080-exec-159] o.s.web.servlet.DispatcherServlet        : Initializing Servlet 'dispatcherServlet'
2022-09-12 01:19:06.099  INFO 22696 --- [o-8080-exec-159] o.s.web.servlet.DispatcherServlet        : Completed initialization in 17 ms
2022-09-12 01:19:06.333 DEBUG 22696 --- [io-8080-exec-98] c.b.s.mapper.BookMapper.selectBookById   : ==>  Preparing: select * from book where id=?
2022-09-12 01:19:06.333 DEBUG 22696 --- [io-8080-exec-82] c.b.s.mapper.BookMapper.selectBookById   : ==>  Preparing: select * from book where id=?
2022-09-12 01:19:06.333 DEBUG 22696 --- [io-8080-exec-20] c.b.s.mapper.BookMapper.selectBookById   : ==>  Preparing: select * from book where id=?
2022-09-12 01:19:06.333 DEBUG 22696 --- [io-8080-exec-51] c.b.s.mapper.BookMapper.selectBookById   : ==>  Preparing: select * from book where id=?
2022-09-12 01:19:06.333 DEBUG 22696 --- [o-8080-exec-132] c.b.s.mapper.BookMapper.selectBookById   : ==>  Preparing: select * from book where id=?
2022-09-12 01:19:06.333 DEBUG 22696 --- [o-8080-exec-180] c.b.s.mapper.BookMapper.selectBookById   : ==>  Preparing: select * from book where id=?
2022-09-12 01:19:06.333 DEBUG 22696 --- [o-8080-exec-162] c.b.s.mapper.BookMapper.selectBookById   : ==>  Preparing: select * from book where id=?
2022-09-12 01:19:06.333 DEBUG 22696 --- [o-8080-exec-142] c.b.s.mapper.BookMapper.selectBookById   : ==>  Preparing: select * from book where id=?
2022-09-12 01:19:06.333 DEBUG 22696 --- [io-8080-exec-20] c.b.s.mapper.BookMapper.selectBookById   : ==> Parameters: 13(Long)
2022-09-12 01:19:06.333 DEBUG 22696 --- [o-8080-exec-180] c.b.s.mapper.BookMapper.selectBookById   : ==> Parameters: 12(Long)
2022-09-12 01:19:06.333 DEBUG 22696 --- [o-8080-exec-162] c.b.s.mapper.BookMapper.selectBookById   : ==> Parameters: 12(Long)
2022-09-12 01:19:06.333 DEBUG 22696 --- [o-8080-exec-142] c.b.s.mapper.BookMapper.selectBookById   : ==> Parameters: 13(Long)
2022-09-12 01:19:06.333 DEBUG 22696 --- [o-8080-exec-185] c.b.s.mapper.BookMapper.selectBookById   : ==>  Preparing: select * from book where id=?
2022-09-12 01:19:06.333 DEBUG 22696 --- [io-8080-exec-51] c.b.s.mapper.BookMapper.selectBookById   : ==> Parameters: 12(Long)
2022-09-12 01:19:06.333 DEBUG 22696 --- [o-8080-exec-185] c.b.s.mapper.BookMapper.selectBookById   : ==> Parameters: 11(Long)
2022-09-12 01:19:06.335 DEBUG 22696 --- [io-8080-exec-51] c.b.s.mapper.BookMapper.selectBookById   : <==      Total: 1
2022-09-12 01:19:06.335 DEBUG 22696 --- [io-8080-exec-82] c.b.s.mapper.BookMapper.selectBookById   : ==> Parameters: 14(Long)
2022-09-12 01:19:06.333 DEBUG 22696 --- [o-8080-exec-194] c.b.s.mapper.BookMapper.selectBookById   : ==>  Preparing: select * from book where id=?
2022-09-12 01:19:06.335 DEBUG 22696 --- [o-8080-exec-194] c.b.s.mapper.BookMapper.selectBookById   : ==> Parameters: 14(Long)
2022-09-12 01:19:06.333 DEBUG 22696 --- [io-8080-exec-98] c.b.s.mapper.BookMapper.selectBookById   : ==> Parameters: 12(Long)
2022-09-12 01:19:06.336 DEBUG 22696 --- [o-8080-exec-162] c.b.s.mapper.BookMapper.selectBookById   : <==      Total: 1
2022-09-12 01:19:06.336 DEBUG 22696 --- [o-8080-exec-180] c.b.s.mapper.BookMapper.selectBookById   : <==      Total: 1
2022-09-12 01:19:06.337 DEBUG 22696 --- [o-8080-exec-142] c.b.s.mapper.BookMapper.selectBookById   : <==      Total: 1
2022-09-12 01:19:06.333 DEBUG 22696 --- [o-8080-exec-132] c.b.s.mapper.BookMapper.selectBookById   : ==> Parameters: 13(Long)
2022-09-12 01:19:06.335 DEBUG 22696 --- [io-8080-exec-20] c.b.s.mapper.BookMapper.selectBookById   : <==      Total: 1
2022-09-12 01:19:06.335 DEBUG 22696 --- [o-8080-exec-185] c.b.s.mapper.BookMapper.selectBookById   : <==      Total: 1
2022-09-12 01:19:06.337 DEBUG 22696 --- [io-8080-exec-82] c.b.s.mapper.BookMapper.selectBookById   : <==      Total: 1
2022-09-12 01:19:06.390 DEBUG 22696 --- [o-8080-exec-132] c.b.s.mapper.BookMapper.selectBookById   : <==      Total: 1
```



⑥ 在 Redis 图形界面客户端刷新 Redis，刷新之前如下图：



刷新后，secondsKillBooks 已经被删除。



⑦ Navicat 中刷新 book 表，表中“秒杀”商品数量均为 0，表明都已卖完且未超卖。

id	name	price	amount	seconds_kill
11	三国演义	69	0	1
12	水浒传	59	0	1
13	西游记	59	0	1
14	红楼梦	59	0	1

刷新 book_order 表，有 40 个订单数据如下图所示，项目搭建完成。

id	order_id	book_id	money	user_id	status	create_time
1	728991577577902082	12	59	10930	0	2022-09-12 01:19:06
2	728991577577902087	13	59	44378	0	2022-09-12 01:19:06
3	728991577577902081	12	59	26936	0	2022-09-12 01:19:06
4	728991577577902080	11	69	31463	0	2022-09-12 01:19:06
5	728991577577902083	13	59	99464	0	2022-09-12 01:19:06
6	728991577577902088	12	59	9699	0	2022-09-12 01:19:06
7	728991577577902086	14	59	72707	0	2022-09-12 01:19:06
8	728991577573707776	14	59	98008	0	2022-09-12 01:19:06
9	728991577577902085	13	59	77015	0	2022-09-12 01:19:06
10	728991578181881858	14	59	35188	0	2022-09-12 01:19:06
11	728991578177687552	11	69	79602	0	2022-09-12 01:19:06
12	728991578181881857	14	59	56757	0	2022-09-12 01:19:06
13	728991578181881856	12	59	80702	0	2022-09-12 01:19:06
14	728991577577902084	12	59	96461	0	2022-09-12 01:19:06
15	728991578244796416	14	59	77953	0	2022-09-12 01:19:06
16	728991578253185024	13	59	50538	0	2022-09-12 01:19:06
17	728991578303516672	13	59	30524	0	2022-09-12 01:19:07
18	728991578332876800	12	59	90810	0	2022-09-12 01:19:07
19	728991578500648960	11	69	64269	0	2022-09-12 01:19:07
20	728991578496454656	12	59	90752	0	2022-09-12 01:19:07
21	728991578496454657	11	69	10065	0	2022-09-12 01:19:07
22	728991578290933760	13	59	33369	0	2022-09-12 01:19:07
23	728991578584535040	11	69	14918	0	2022-09-12 01:19:07
24	728991578513231872	14	59	73970	0	2022-09-12 01:19:07
25	728991578546786304	11	69	39353	0	2022-09-12 01:19:07
26	728991578584535041	12	59	86761	0	2022-09-12 01:19:07
27	728991578236407808	12	59	46211	0	2022-09-12 01:19:07
28	728991578546786305	13	59	1593	0	2022-09-12 01:19:07
29	728991578580340736	12	59	48943	0	2022-09-12 01:19:07
30	728991578622283776	13	59	15482	0	2022-09-12 01:19:07
31	728991578626478080	11	69	69837	0	2022-09-12 01:19:07
32	728991578639060993	11	69	51605	0	2022-09-12 01:19:07
33	728991578639060992	11	69	96325	0	2022-09-12 01:19:07
34	728991578651643904	13	59	86031	0	2022-09-12 01:19:07
35	728991578664226817	11	69	66134	0	2022-09-12 01:19:07
36	728991578664226816	14	59	6047	0	2022-09-12 01:19:07
37	728991578701975552	13	59	99330	0	2022-09-12 01:19:07
38	728991578710364160	14	59	47298	0	2022-09-12 01:19:07
39	728991578710364162	14	59	70034	0	2022-09-12 01:19:07
40	728991578710364161	14	59	68479	0	2022-09-12 01:19:07

四、总结

“Spring 框架在高并发场景中的应用研究”是非常有意义的一次搭建项目的研究。如今 Spring boot 框架受到越来越多开发者的青睐，广泛运用在企业开发中。本次高并发应用项目——电商“秒杀”系统的构建在项目开发中也十分常见，能够在暑期进行一次相关项目的实战必然是有收获的。

在开始本次科研实习前，笔者甚至还没有接触过 Java 程序。如今 Java 语言在程序开发中广泛使用，借本次暑期科研实习的机会，我结合之前学习 C++语言的知识，也学习了一点 Java 语言，也是为今后的学习打基础。笔者也没有使用 IDEA 的经验。虽然之前接触过 Visual Studio 等软件，但也只作为运行程序的工具，没有开发经验。除此之外，数据库的使用也是一大难题。对于毫无经验的小白来说，想要正确配置、连接数据库，需要经历无数的试错，投入大量时间和精力。包括一开始的下载并安装程序，也经历了各种各样的波折，出现各种问题。

笔者解决问题主要依靠的是 CSDN 社区中的博客。遇到一个问题就去浏览搜索解决方案，经过不断的搜索和自我摸索，最终还是在零基础的条件下独立完成了高并发案例的搭建。这次经历可谓大大提高了自学能力尤其是搜索资源的能力，同时也让自己对 IDEA 的使用、数据库的连接有了一定的了解。能在短暂的时间里收获知识，笔者这次暑期科研的经历是宝贵、颇具意义的。

最后，衷心感谢项目老师和同学们的支持与帮助，感谢所有提供参考解决方案的博客。愿本次科研实习项目结束后，新的一切都顺利。

指导教师评价意见及成绩评定

见习成绩：☐通过　　☐不通过

☐ 同意学生获得 1 学分

指导教师（签名）：

年　月　日

院（系）意见

负责人（签字）：

院（系）公章：

年　月　日