A. 基础识记题（10 题）——难度：基础

A1（单选）——基于 SeckillApplication.java 的 main 方法

该类在系统启动时的首要任务是什么?
A. 生成订单 ID
B. 初始化日志系统
C. 扣减 Redis 库存
D. 启动工作线程池
答案：B

A2（填空）——基于 SeckillAppContext.start 方法

SeckillAppContext 在启动时会将商品库存初始化为：每个商品 __（一个整数）。
答案：100

A3（单选）——基于 SeckillAppContext.loadMaliciousUsers 方法

用于存储恶意用户 UID 的结构是什么?
A. ConcurrentHashMap
B. BloomFilterUtil
C. LinkedBlockingQueue
D. HashSet
答案：B

A4（填空）——基于 SeckillSystem.processRequest 方法

限流使用的类名是 _____。
答案：SlidingWindowRateLimiter

A5（单选）——基于 SeckillSystem.java 的 requestQueue 字段

用于承载待处理秒杀请求的数据结构是：
A. ArrayList
B. Vector
C. LinkedBlockingQueue
D. LinkedList
答案：C

A6（单选）——基于 RedisService.decreaseStockSafe 方法

扣减库存的原子性依赖于：
A. synchronized
B. Redisson
C. Lua 脚本
D. WATCH + MULTI
答案：C

A7（填空）——基于 RedisService.tryLock 方法

分布式锁的实现依赖 Redis 的 **SET NX EX** 命令，其中 NX 代表 ___。
答案：仅当 key 不存在时才设置

A8（单选）——基于 OrderManager.createOrder 方法

订单状态从 CREATED 变更到 FINISHED 的方式是：
A. 同步执行
B. 使用 ScheduledExecutorService 定时调度
C. 新线程手工 sleep
D. Redis 事件机制
答案：B

A9（填空）——基于 BloomFilterUtil.add 方法

BloomFilterUtil 使用的数据结构是 Java 的 ___。
答案：BitSet

A10（单选）——基于 LogUtil.initLogging 方法

系统日志会被写入到：
A. logs/seckill.log
B. redis.log
C. order.log
D. worker.log
答案：A

B. 逻辑分析题（8 题）——难度：中等

B1（简答）——基于 SeckillSystem.processRequest 的过滤流程

说明 processRequest 方法中请求被拒绝的四种情况。

答案：

1. **恶意用户拦截**（maliciousBloom.contains(userId)）

2. **滑动窗口限流超限**（rateLimiter.tryAcquire() 返回 false）
3. **布隆过滤器判断为重复请求**（requestBloom.contains(userId-productId)）
4. **请求队列已满，offer 返回 false**

B2（简答）——基于 SlidingWindowRateLimiter.tryAcquire 方法

阐述滑动窗口限流如何实现“在最近 1 秒内请求数不能超过 N”。

答案：

- 数据结构：ConcurrentLinkedQueue
- 每次请求记录当前时间戳
- 不断清理队列头部时间戳（小于 1 秒前的）
- 判断队列长度是否超过 maxRequests
- 若未超过，将当前时间戳加入并返回 true，否则返回 false

B3（简答）——基于 BloomFilterUtil.hash 逻辑

说明 BloomFilterUtil 使用单哈希的风险是什么？

答案：

- 单哈希会导致高误判率，某些从未出现过的 key 会被误认为存在
- 容易造成正常用户重复请求被错判为“重复”，被提前拒绝

B4（简答）——基于 RedisService.tryLock 和 releaseLockSafe 方法

解释为什么释放锁必须用 Lua 脚本，而不能直接使用 DEL。

答案：
DEL 可能误删其他线程新设置的锁（锁过期后被别人抢走）。
Lua 脚本可以保证：

- 仅当锁的 value 和当前线程的 lockValue 相等时才删除
- 保证检查与删除的原子性

B5（简答）——基于 processStockAndOrder 方法

说明该方法中分布式锁的作用是什么，若无该锁将会出现什么问题？

答案：

- 分布式锁确保同一商品的库存扣减在并发场景下是串行的
- 没有锁时，不同线程可能同时读取到相同库存 → 产生超卖

B6（简答）——基于 OrderManager 的 scheduler 调度

为什么订单状态需要使用 ScheduledExecutorService 来异步变化?

答案:

- 模拟真实支付流程（非瞬时完成）
- 避免阻塞库存扣减线程，提高处理吞吐量
- 模拟订单从 CREATED → PAID → FINISHED 的生命周期

B7（简答）——基于 RedisService.decreaseStockSafe 的 Lua 脚本

解释 Lua 脚本返回 -1 的两种情况。

答案:

1. GET 返回 null（库存 key 不存在）
2. 库存 ≤ 0，不足以扣减

B8（简答）——基于 SeckillSystem 的线程池设计

为什么采用固定大小线程池？若改为 CachedThreadPool 会发生什么?

答案:

- 固定线程池便于控制并发度、防止线程数无限增长
- CachedThreadPool 根据请求激增可能创建大量线程 → 引发线程调度压力、内存消耗、降低吞吐和稳定性

C. 代码细节题（5 题）——难度：细节/代码理解

C1（判断）——基于 BloomFilterUtil.contains 方法

布隆过滤器使用 BitSet 的 get(index) 判断元素是否存在，是一种精确判断（无误判）。
答案：错误（布隆过滤器可能误判）

C2（填空）——基于 RedisService.tryLock

tryLock 使用 Jedis 的 set 方法，其参数 NX 表示：仅当 key __ 时成功。
答案：不存在

C3（判断）——基于 SeckillSystem.processRequest

requestBloom.add(userId-productId) 在抽签之后执行。
答案：错误（在抽签之前执行）

C4（填空）——基于 Request 类

Request 中包含两个字段：___（用户 ID）和 ___（商品 ID）。
答案：userId，productId

C5（判断）——基于 OrderNode

OrderNode 内部字段包括 orderId、uid、pid、status。
答案：正确

D. 综合应用题（高难度 4 题）——并发排查 / 代码改写 / 性能分析

D1（高难度）——基于 processStockAndOrder 的分布式锁

问题：
当前 tryLock 设置超时时间为 5 秒。如果某个任务执行超过 5 秒（如 Redis 卡顿），锁会过期并被其他线程抢走。请说明：

1. 会导致什么问题？
2. 给出一种改进方式。

标准答案：

1. **问题：锁过期 → 多线程同时扣库存 → 产生超卖**
    - 原锁持有者仍在执行扣减，但是锁自动过期，其他线程获得新锁，导致并发扣库存。
2. **改进方式：**
    - 使用 Redisson 的可重入锁/自动续租机制
    - 或实现 “看门狗”（续期线程）定期延长 lock 的 TTL

D2（高难度）——滑动窗口限流瓶颈分析

问题：
SlidingWindowRateLimiter 使用 ConcurrentLinkedQueue 存时间戳，当 QPS 达到 10 万级别时会出现怎样的性能问题？如何优化？

标准答案：
瓶颈：

- 队列不断增长、频繁清理旧时间戳

- size() 或 while(peek < threshold) 会造成大量 CAS 竞争
- GC 压力增加

优化方式：

- 使用“固定桶计数”算法（数组长度为窗口大小 / 精度）
- 使用 LongAdder 统计
- 或使用令牌桶算法降低写放大

D3（高难度）——布隆过滤器重复请求误判问题

问题：
requestBloom 可能误判用户重复提交，从而“错误拒绝”有效请求。请设计一种改进方案（不增加 Redis）。

标准答案：
可选方案：

1. **多哈希布隆过滤器**：使用 3~5 个哈希函数，降低误判
2. **增加 BitSet 大小**：例如从 10 万位扩展到百万级
3. **使用 Caffeine 本地缓存（存在即拒绝）**：过期时间短，误判率为 0
4. **双层布隆过滤器**：第一层布隆判定后再查 LRU map 验证

D4（高难度）——队列满导致请求丢弃问题排查

假设在高峰期大量用户被“队列已满”拒绝，请分析根本原因，并给出一种改进方法。

标准答案：
根本原因：

- requestQueue 固定容量（10000），生产速度 > 消费速度
- worker 线程处理时间受 Redis 性能、锁竞争、订单调度影响

改进方式：

- 增加 worker 线程数
- 使用 Disruptor/RingBuffer 替代队列
- 前端削峰：改为“预减库存 + MQ 异步下单”
- 加入“排队提示”而非直接丢弃

电商秒杀系统问答题套卷及参考答案

一、基础识记题（8-10 题，单选 / 填空）

> **特点**：考察类作用、方法参数意义、常量定义、数据结构选择、基本执行步骤。

[1] （SeckillApplication.java）单选题（难度：低）

`SeckillApplication` 的 `main` 方法的主要功能是：
A. 启动 Redis 服务并初始化库存
B. 初始化日志、启动应用上下文并模拟秒杀
C. 直接执行库存扣减逻辑
D. 只做恶意用户过滤

答案：B

---

[2] （SeckillAppContext.java）填空题（难度：低）

`SeckillAppContext.start()` 方法中，通过循环 `for (int itemId = 1; itemId <= 5; itemId++)` 调用 `system.getRedisService().setStock(...)` 的主要目的是 ____。

答案：初始化 5 个商品的库存到 Redis，每个库存数量为 100。

---

[3] （RedisService.java）单选题（难度：低）

`RedisService` 的 `decreaseStockSafe` 方法使用 Lua 脚本的主要目的是什么？
A. 提高库存查询速度
B. 记录每次库存变更日志
C. 防止高并发情况下的超卖问题，确保扣减原子性
D. 模拟延迟支付

答案：C

---

[4] （OrderManager.java）填空题（难度：低）

`OrderManager` 中使用的 `ConcurrentHashMap<Long, OrderNode>` 的作用是 ____。

答案：存储并管理订单对象，支持并发访问和状态更新。

---

[5] （BloomFilterUtil.java）单选题（难度：低）

`BloomFilterUtil` 进行存储时底层主要使用的数据结构是：
A. `ArrayList`
B. `BitSet`
C. `HashMap`
D. `LinkedList`

答案：B

---

[6] （SlidingWindowRateLimiter.java）填空题（难度：低）

`SlidingWindowRateLimiter` 中的 `maxRequestsPerSecond` 表示 ___。

答案：在滑动窗口算法中，允许一秒钟内的最大请求数。

---

[7] （SeckillSystem.java）单选题（难度：低）

`SeckillSystem.processRequest()` 中，`lotteryWinRate` 的作用是：
A. 决定用户是否进入支付环节
B. 决定用户是否进入库存扣减流程
C. 决定用户是否有资格进入秒杀队列
D. 限制每天参与秒杀的次数

答案：C

---

[8] （LogUtil.java）填空题（难度：中）

`LogUtil.initLogging()` 方法中，为了自定义日志输出格式，使用了匿名内部类重写了 ___ 的 `format` 方法。

答案：`java.util.logging.Formatter`

---

[9] （OrderStatus.java）单选题（难度：低）

系统中订单有几个状态？它们分别是：
A. 2 个：`CREATED`、`FINISHED`
B. 3 个：`CREATED`、`PAID`、`FINISHED`
C. 4 个：`CREATED`、`PAID`、`CANCELLED`、`FINISHED`
D. 2 个：`CREATED`、`CANCELLED`

答案：B

二、逻辑分析题（6-8 题，简答题）

> **特点**：需要分析类间依赖、调用流程、技术细节。

[10] （SeckillAppContext.java）简答题（难度：中）

在 `simulateRequests()` 中，每轮秒杀开始前调用 `system.clearRequestBloom()` 的目的是什么？

答案：`requestBloom` 用于本轮秒杀中防止用户重复请求。如果不在每轮开始前清空，则上一轮的数据会影响新一轮的用户请求去重，导致合法请求被错误拦截。

---

[11] （SeckillSystem.java）简答题（难度：中）

`processRequest` 方法依次做了哪些拦截和过滤步骤？请按顺序说明。

答案：

1. 恶意用户过滤：检查 `maliciousBloom` 是否包含该 `userId`
2. 滑动窗口限流：调用 `rateLimiter.tryAcquire()` 判断一秒内请求数是否超限
3. 请求去重：检查 `requestBloom` 是否包含 `userId-productId` 键
4. 抽签概率判断：`Math.random()` 是否小于 `lotteryWinRate`
5. 请求队列容量判断：是否可以 `offer` 到 `requestQueue`

---

[12] （RedisService.java）简答题（难度：中）

Lua 脚本 `"DECREASE_STOCK_SCRIPT"` 的逻辑步骤是什么？

答案：

1. 从 Redis 获取库存值（`GET`）并转换为数字
2. 如果库存不存在或小于等于 0，则返回 -1（失败）
3. 否则调用 `DECR` 将库存减 1，并返回剩余库存数

---

[13] （OrderManager.java）简答题（难度：中）

`createOrder` 中是如何模拟订单状态流转的？

答案：

1. 创建订单并将状态设为 `CREATED`
2. 将订单放入 `orderMap` 存储
3. 通过 `scheduler.schedule` 延迟 1~3 秒调用 `updateStatus` 将状态改为 `PAID`
4. 再延迟 1~2 秒调用 `updateStatus` 将状态改为 `FINISHED`

---

[14] （SlidingWindowRateLimiter.java）简答题（难度：中）

滑动窗口限流算法是如何判断是否允许新请求的？

答案：

1. 获取当前时间戳，计算一秒前的时间点
2. 从 `requestTimestamps` 队列中移除所有早于一秒前的时间戳
3. 如果队列大小小于 `maxRequestsPerSecond`，允许请求并记录当前时间戳；否则拒绝请求

---

[15] （BloomFilterUtil.java）简答题（难度：中）

`BloomFilterUtil` 的哈希值是如何计算的?

答案：取 `value.hashCode()` 乘以常量 `hashSeed`，取绝对值后对 `size` 取模，得到 BitSet 下标。

三、代码细节题（4-5 题，判断 / 填空）

> **特点**：从源码提取具体细节，判断正误或填空。

[16] （SeckillSystem.java）判断题（难度：中）

在 `processStockAndOrder` 方法中，如果分布式锁 `tryLock` 获取失败，系统会直接返回而不尝试扣库存。
答案：正确

---

[17] （OrderNode.java）填空题（难度：低）

`OrderNode` 的 `toString()` 方法返回的对象格式为：

```
Order{orderId=___, userId=___, productId=___, status=___}
```

答案：
orderId = 订单 ID
userId = 用户 ID
productId = 商品 ID
status = 订单状态

---

[18] （RedisService.java）判断题（难度：中）

`releaseLockSafe` 方法能确保只有分布式锁的持有者才能释放锁。
答案：正确

---

[19] （LogUtil.java）填空题（难度：低）

`LogUtil` 在初始化时会在项目目录下创建一个名为 ___ 的文件夹，用于存放日志文件。

答案：`logs`

四、综合应用题（3-4 题，代码改写 / 问题排查）

> **特点**：结合并发、高可用场景，考察优化能力、排错能力。

[20] （综合）改写题（难度：高）

目前 `lotteryWinRate` 在代码中固定为 1.0，请改写 `SeckillSystem`，让该值可以在 `SeckillAppContext.start()` 中从外部参数传入并生效。

参考答案思路：

1. 在 `SeckillSystem` 中添加构造函数 `public SeckillSystem(double lotteryWinRate)`
2. 去掉旧的 `lotteryWinRate` 固定定义
3. `SeckillAppContext` 启动时传入参数，例如 `new SeckillSystem(0.5)`

---

[21] （综合）问题排查（难度：高）

如果 Redis 服务短暂中断，`RedisService.tryLock` 方法可能出现什么异常？如何优化？

答案：

- 异常：`JedisConnectionException`，会导致请求直接失败
- 优化：增加重试机制，或提前检测 Redis 连接可用性

---

[22] （综合）性能瓶颈分析（难度：高）

在 10 万用户同时请求的压测中，系统 QPS 无法提升，分析可能的三个瓶颈点。

答案：

1. **滑动窗口限流器**参数太小限制了最大 QPS
2. `requestQueue` 容量不足导致大量请求被拒绝
3. Worker 线程数 `executor` 太少、单线程等待 Redis 阻塞

---

[23] （综合）并发安全（难度：高）

如果去掉分布式锁，系统可能发生什么问题？

答案：

- 高并发下多个线程同时扣减库存，由于非原子操作会造成**超卖**现象
- 可能出现库存负数，导致不合法的订单产生

Gemini -Java 电商秒杀系统核心源码 — 问答考核题库

第一部分：基础识记题（共 10 题）

[基础 - 填空] 1. (SeckillApplication.java)
在 `SeckillApplication` 的 `main` 方法中，第一行调用的 `LogUtil.initLogging()` 方法的作用是 **初始化日志系统，配置日志格式、级别和输出目标（控制台和文件）**。

[基础 - 填空] 2. (SeckillAppContext.java)
在 `start` 方法中，系统为 5 个商品（ID 从 1 到 5）在 Redis 中设定的初始库存量均为 **100**。

[基础 - 单选] 3. (SeckillSystem.java)
用于存储待处理秒杀请求的核心数据结构 `requestQueue` 是 Java并发包中的哪一个类的实例？ **(B)**
A. `ArrayBlockingQueue`
B. `LinkedBlockingQueue`
C. `ConcurrentLinkedQueue`
D. `PriorityBlockingQueue`

[基础 - 填空] 4. (OrderManager.java)
`OrderManager` 类使用 `ScheduledExecutorService` 类型的线程池来异步模拟订单状态从 `PAID` 到 `FINISHED` 的流转。

[基础 - 填空] 5. (RedisService.java)
在 `tryLock` 方法中，为了保证"设置键值"和"设置过期时间"的原子性，`jedis.set` 方法使用了 `SetParams` 参数，其核心配置是 `nx()` **(不存在时设置)** 和 `ex(expireSeconds)` **(设置秒级过期时间)**。

[基础 - 填空] 6. (BloomFilterUtil.java)
`BloomFilterUtil` 类内部用于存储位信息的核心数据结构是 Java Util 包中的 `BitSet` 类。

[基础 - 单选] 7. (SlidingWindowRateLimiter.java)
`SlidingWindowRateLimiter` 类中，`requestTimestamps` 队列存储的数据类型是？ **(C)**
A. `Integer` (秒)
B. `Date`
C. `Long` (毫秒时间戳)
D. `String`

[基础 - 填空] 8. (LogUtil.java)
`LogUtil` 类中的静态布尔变量 `initialized` 的作用是 **防止日志系统被重复初始化**。

[基础 - 填空] 9. (OrderStatus.java)
`OrderStatus` 枚举类定义了订单生命周期中的三个状态，按顺序分别是：`CREATED`、`PAID` 和 `FINISHED`。

[基础 - 填空] 10. (Request.java)
`Request` 模型类封装了一次秒杀请求的两个核心参数，分别是 `userId` **(用户ID)** 和 `productId` **(商品ID)**。

第二部分：逻辑分析题（共 7 题）

[中等 - 简答] 11. (SeckillSystem.java)
在 `processRequest` 方法中，系统对一个请求依次执行了 5 个核心步骤（恶意用户过滤 -> 限流 -> 重复请求去重 -> 抽签 -> 入队）。请说明为什么要把**恶意用户过滤**和**限流**放在最前面?
答案:
将恶意用户过滤和限流放在最前面是出于性能和资源优化的考虑：

1. **尽早拒绝无效流量**：恶意用户和超出流量限制的请求是典型的无效流量。在处理流程的早期就将它们拒绝，可以避免它们消耗后续更昂贵的计算资源，如布らなかった过滤器计算、随机数生成和入队操作。
2. **保护下游系统**：限流是保护整个系统的第一道防线，确保系统不会因瞬间流量洪峰而崩溃。恶意用户过滤则可以减轻业务层面的安全压力。将这两者前置，能最大化地保护系统稳定性和资源利用率。

[中等 - 简答] 12. (SeckillAppContext.java)
在 `simulateRequests` 方法的 for 循环内部，每次开始新一轮秒杀前都会调用 `system.clearRequestBloom()`。请解释这样做的目的。
答案:
这样做的目的是为了**实现多轮独立的秒杀活动**。请求去重布隆过滤器（`requestBloom`）的作用是防止同一个用户在**单轮**秒杀中对同一商品重复提交请求。如果不清空，那么第一轮成功抢购的用户在第二轮中会被误判为“重复请求”而被拦截，无法参与新一轮的秒-杀。因此，在每轮开始前必须重置布隆过滤器，确保所有用户在新一轮中都有公平的参与机会。

[中等 - 简答] 13. (RedisService.java)
请解释 `decreaseStockSafe` 方法为什么要使用 Lua 脚本，而不是在 Java 代码中先 `GET` 库存，判断后再 `DECR` 库存?
答案:
使用 Lua 脚本是为了**保证库存扣减操作的原子性**。如果在 Java 代码中执行“读-改-写”操作（`GET` -> 判断 -> `DECR`），在高并发场景下，多个线程可能同时读取到相同的库存值（如 1），然后都通过了库存大于 0 的判断，最后都执行 `DECR`，导致库存被减到负数（超卖）。而 Redis 执行 Lua 脚本是原子性的，脚本在执行期间不会被其他命令中断，从而确保了“读取-判断-扣减”这一系列操作作为一个不可分割的整体执行，彻底避免了并发竞争条件下的超卖问题。

[中等 - 简答] 14. (SeckillSystem.java)
`SeckillSystem` 中使用了 `maliciousBloom` 和 `requestBloom` 两个布隆过滤器。请说明它们各自的作用以及为什么需要两个独立的过滤器。
答案:

1. `maliciousBloom` **(恶意用户布隆过滤器)**：用于存储一个**全局的、预加载的黑名单用户列表**。它的生命周期是整个应用运行期间，作用是快速过滤掉已知的恶意账户，防止其参与任何秒杀活动。
2. `requestBloom` **(请求去重布隆过滤器)**：用于记录**本轮秒杀**中“用户ID-商品ID”的组合。它的生命周期仅限于一轮秒杀（每轮会重置），作用是防止正常用户在**单次活动**中重复点击按钮，发送多个相同请求，造成资源浪费。

需要两个独立的过滤器是因为它们解决的问题维度和生命周期完全不同：一个是全局、持久的身份风控；另一个是局部、临时的请求去重。

[中等 - 简答] 15. (SlidingWindowRateLimiter.java)
请简述 `tryAcquire` 方法是如何通过 `ConcurrentLinkedQueue` 实现滑动窗口限流逻辑的。
答案:
`tryAcquire` 方法通过以下步骤实现滑动窗口限流：

1. **记录当前请求时间**：获取当前时间的毫秒时间戳 `now`。
2. **定义窗口边界**：计算出1秒钟前的时间点 `oneSecondAgo`，这就是滑动窗口的左边界。
3. **滑动窗口（移除过期时间戳）**：通过 `while` 循环检查队列头部的元素（最早的请求时间戳），如果该时间戳早于 `oneSecondAgo`，说明它已经不在当前时间窗口内，就将其从队列中 `poll()` 移除。这个过程就是"滑动"。
4. **检查窗口内请求数**：移除所有过期时间戳后，队列的 `size()` 就代表了当前 1 秒窗口内的请求总数。
5. **判断是否允许通过**：如果 `size()` 小于设定的 `maxRequestsPerSecond`，则将当前请求的时间戳 `now` 加入队列尾部，并返回 `true`（允许通过）。否则，直接返回 `false`（拒绝请求）。

[中等 - 简答] 16. (OrderManager.java / SeckillSystem.java)
`processStockAndOrder` 方法中的库存扣减和订单创建被一个分布式锁保护。订单状态的更新（`PAID`，`FINISHED`）却是异步执行的。请分析这样设计的优缺点。
答案：
这样设计的优缺点如下：

- **优点**：
    1. **提升秒杀接口性能**：将耗时较长、非核心的订单状态流转（模拟支付、发货等）操作异步化，使得持有分布式锁的关键代码块执行时间极短。这大大减少了锁的占用时间，提高了系统的吞吐量和并发能力。
    2. **用户体验更佳**：用户在秒杀成功后能迅速得到"下单成功"的反馈，而后续的支付状态变化则通过后台任务处理，符合现代电商系统的设计模式。
- **缺点**：
    1. **数据一致性延迟**：订单的最终状态不是实时的，存在一个短暂的延迟窗口。
    2. **系统复杂性增加**：引入了 `ScheduledExecutorService` 线程池和异步任务，增加了系统的复杂性和维护成本，需要考虑线程池关闭、任务异常处理等问题。

[中等 - 简答] 17. (RedisService.java)
`releaseLockSafe` 方法也使用 Lua 脚本实现。请解释脚本中 `if redis.call('get', KEYS[1]) == ARGV[1]` 这句判断的必要性。
答案：
这句判断是为了**确保锁的释放操作是安全的，防止误删他人的锁**。在高并发场景下，可能发生以下情况：

1. 线程 A 获取了锁，但由于业务执行时间过长或 GC 停顿，导致锁因超时而自动过期。
2. 此时，线程 B 成功获取了同一个锁。
3. 随后，线程 A 的业务执行完毕，它去执行释放锁的操作。

如果没有这句判断，线程 A 会直接删除锁，而这个锁实际上是线程 B 持有的，这就造成了锁的错误释放，可能导致线程 B 的临界区代码不再安全。通过判断锁的 `value` (一个唯一的随机字符串) 是否与自己当初设置的一致，可以保证"只释放自己加的锁"，避免了这种风险。

第三部分：代码细节题（共 4 题）

[高等 - 判断] 18. (SeckillSystem.java)
在 `processStockAndOrder` 方法中，`finally` 代码块确保了无论 `try` 块中是否发生异常，`redisService.releaseLockSafe()` 都会被执行，因此该方法不可能导致分布式锁死锁。
答案： 正确。

`finally` 块的设计正是为了保证资源的确定性释放。即使在 `createOrder` 时抛出异常，锁也会被释放。同时，`tryLock` 方法设置了过期时间，即使应用崩溃，锁最终也会被 Redis 自动删除。因此，死锁的风险被有效规避。

[高等 - 填空] 19. (SlidingWindowRateLimiter.java)
`requestTimestamps` 变量选用了 `ConcurrentLinkedQueue` 而不是 `LinkedList`。这是因为 `ConcurrentLinkedQueue` 是 **线程安全的**，支持高并发环境下的无锁 `add` 和 `poll` 操作，避免了在 `tryAcquire` 方法中额外加锁的开销。

[高等 - 填空] 20. (BloomFilterUtil.java)
`add` 方法中的哈希计算 `Math.abs(value.hashCode() * hashSeed) % size` 中，`% size` 操作的目的是**将计算出的哈希值映射到 `BitSet` 的有效索引范围 [0, size-1] 内。**

[高等 - 判断] 21. (OrderManager.java)
`OrderManager` 中的 `orderMap` 使用了 `ConcurrentHashMap`，`orderIdGenerator` 使用了 `AtomicLong`，这保证了 `createOrder` 方法是完全线程安全的，无需任何 `synchronized` 块。
答案： 正确。
`AtomicLong` 的 `incrementAndGet` 操作是原子的，保证了订单 ID 生成的唯一性。
`ConcurrentHashMap` 的 `put` 操作是线程安全的。由于旧的链表逻辑已被移除，该方法的核心操作都由线程安全的组件完成，因此不再需要 `synchronized` 关键字。

第四部分：综合应用题（共 3 题）

[高等 - 问题排查] 22. (SeckillSystem.java)
假设在 `processStockAndOrder` 方法的 `try` 块中，`redisService.decreaseStockSafe(stockKey)` 执行成功，但随后的 `orderManager.createOrder(...)` 因为数据库连接池耗尽而抛出异常。请问此时系统会出现什么问题？如何优化代码来解决或缓解这个问题？
答案：

- **出现的问题**：
  系统会出现**数据不一致**的问题。具体表现为：Redis 中的库存已经成功扣减，但数据库中并未生成对应的订单。这会导致“库存减少但订单未创建”的“幽灵库存”现象，最终导致卖出的商品少于初始库存。
- **优化方案**：
  1. **引入消息队列（MQ）**：这是业界标准解法。`decreaseStockSafe` 成功后，不直接调用 `createOrder`，而是将包含 `userId` 和 `productId` 的消息发送到可靠的消息队列（如 RocketMQ, Kafka）。然后由一个专门的订单服务消费者去异步、可靠地创建订单。如果创建失败，MQ 的重试机制可以保证最终订单被创建，实现最终一致性。
  2. **事务补偿机制**：如果 `createOrder` 失败，在 `catch` 块中执行一个补偿操作，即调用 Redis 的 `INCR` 命令将之前扣减的库存加回去。但这会增加逻辑复杂性，且补偿操作自身也可能失败。
  3. **完善的重试与监控**：在 `catch` 块中加入有限次数的重试逻辑，并记录失败日志。通过监控系统发现“库存扣减成功但订单创建失败”的日志，进行人工或自动补偿。

[高等 - 代码改写] 23. (BloomFilterUtil.java)
当前的 `BloomFilterUtil` 只使用了一个简单的哈希函数，这会增加哈希冲突和误判率。请改写 `add` 和 `contains` 方法的逻辑，使其支持多个哈希函数（例如，使用 3 个不同的 `hashSeed`）。你无需实现新的哈希算法，只需展示如何应用多个种子即可。
答案：
首先，在类中增加一个哈希种子数组：

```
// private final int hashSeed = 7; // 原有代码
private final int[] hashSeeds = {7, 11, 13}; // 改为种子数组
```

然后改写 `add` 和 `contains` 方法：

```
public void add(String value) {
    for (int seed : hashSeeds) {
        int hash = Math.abs(value.hashCode() * seed) % size;
        bitSet.set(hash);
    }
}

public boolean contains(String value) {
    for (int seed : hashSeeds) {
        int hash = Math.abs(value.hashCode() * seed) % size;
        if (!bitSet.get(hash)) {
            return false; // 只要有一个位为0，就一定不存在
        }
    }
    return true; // 所有位都为1，可能存在
}
```

[高等 - 性能分析] 24. (SeckillSystem.java)
假设在一次大型秒杀活动中，日志显示大量请求因"滑动窗口限流"被拒绝，但此时 CPU 和 Redis 的负载都很低。请分析可能导致此现象的两个原因，并提出相应的调优建议。
答案：

- **可能原因一：限流阈值设置过于保守**
  `SlidingWindowRateLimiter` 的 `maxRequestsPerSecond` 值（当前为 20000）可能远低于系统的实际处理能力。系统硬件和 Redis 服务本身可以处理更高的 QPS，但被人为地限制住了。
  调优建议：通过压力测试，逐步提高 `maxRequestsPerSecond` 的值，找到系统在不出现性能显著下降（如响应时间飙升）或资源瓶颈（CPU、内存、网络IO）时的最优吞吐量，并将其设置为限流阈值。
- **可能原因二：流量突刺过于集中**
  秒杀请求可能在 1 秒内的某个极短时间窗口（如前 100 毫秒）内集中爆发，瞬间填满了限流器的计数。即使在这一秒的后续时间里系统完全空闲，后续的请求也会因为窗口已满而被拒绝。滑动窗口算法本身无法完全抚平这种微观时间尺度上的流量毛刺。
  调优建议：
  1. **结合令牌桶算法**：引入令牌桶算法（如 Guava 的 RateLimiter），它允许一定程度的流量突发（消耗积攒的令牌），对于平滑流量尖峰比滑动窗口效果更好。
  2. **前端引导**：在客户端增加随机延迟或引导用户分批进入，从源头上打散请求，使其在时间上更均匀地分布。
  3. **增加队列容量和Worker数量**：适当增加 `LinkedBlockingQueue` 的容量和 `ExecutorService` 的线程数，以吸收和处理更多的瞬时流量，但要注意这会将压力后移到消费者端。

Kimi

电商秒杀系统源代码完整面试题集

一、基础识记题（共5题）

题型说明：聚焦类作用、方法参数、常量定义、数据结构类型等基础知识点，答案可直接从代码中定位。

题目1（单选）：`SeckillApplication.java` 中 `main` 方法执行时，最终通过哪个类的实例调用 `simulateRequests` 方法来完成请求模拟？
难度：基础
答案：`SeckillAppContext` 类。在 `SeckillApplication.main` 中通过 `SeckillAppContext app = new SeckillAppContext();` 创建实例，随后调用 `app.simulateRequests(10000)`。

题目2（填空）：`SeckillAppContext.java` 的 `start` 方法中，初始化了商品ID从 到 的库存，每个商品的初始库存量为 。
难度：基础
答案：1；5；100。代码：`for (int itemId = 1; itemId <= 5; itemId++) { system.getRedisService().setStock("item:" + itemId, 100); }`

题目3（单选）：`OrderStatus.java` 枚举类定义了订单的三种状态，以下哪个状态是订单创建后的初始状态？
难度：基础
答案：`CREATED`。在 `OrderManager.createOrder` 方法中，新建 `OrderNode` 时传入 `OrderStatus.CREATED`。

题目4（填空）：`RedisService.java` 中，分布式锁的Lua脚本 `RELEASE_LOCK_SCRIPT` 在执行时会比较 和 是否相等，以确保只有锁持有者才能释放锁。
难度：基础
答案：`redis.call('get', KEYS[1])`（当前锁的value）；`ARGV[1]`（传入的lockValue）。脚本：`"if redis.call('get', KEYS[1]) == ARGV[1] then ..."`

题目5（单选）：`BloomFilterUtil.java` 使用 `java.util.BitSet` 作为底层数据结构，其核心目的是 。
难度：基础
答案：高效存储和查询元素的哈希值，实现去重判断。`BitSet` 用于以位（bit）为单位存储哈希结果，节省内存。

二、逻辑分析题（共11题）

题型说明：需分析类间调用流程、技术实现原理、关键代码逻辑，要求结合多行代码或跨类协作进行解释。

题目6（简答）：基于 `SeckillAppContext.java` 的 `simulateRequests` 方法，描述每一轮秒杀请求的完整处理流程（从恶意用户名单加载到请求去重），并说明 `clearRequestBloom` 的作用。
难度：中等
答案：

- **流程**：每轮循环中，① 调用 `system.clearRequestBloom()` 重置正常请求布隆过滤器；② 循环 `totalUsers` 次，为每个用户调用 `system.processRequest(i, targetItemId)` 提交请求；③ 线程休眠0.5秒；④ 额外插入3个恶意用户请求（UID=2,5,8）进行重复测试。
- `clearRequestBloom` **作用**：每轮秒杀开始前重置 `requestBloom`，确保每轮请求的重复判断独立，避免上一轮请求影响当前轮的重复性检测。

题目7（简答）：`SeckillSystem.java` 的 `processRequest` 方法包含5层防御机制，请按顺序列出每层判断的逻辑、使用的组件及被拦截时的日志输出格式。
难度：中等
答案：

1. **恶意用户过滤**：`maliciousBloom.contains(String.valueOf(userId))`，拦截日志：`"恶意用户拦截！UID=" + userId`
2. **滑动窗口限流**：`rateLimiter.tryAcquire()`，拦截日志：`"滑动窗口限流：请求超速 UID=" + userId`
3. **重复请求去重**：`requestBloom.contains(userId + "-" + productId)`，拦截日志：`"重复请求被拦截 UID=" + userId`
4. **随机抽签资格**：`Math.random() > lotteryWinRate`（当前 `lotteryWinRate=1.0`，实际不会拦截），拦截日志：`"用户未中签，UID=" + userId`
5. **队列容量限制**：`!requestQueue.offer(new Request(userId, productId))`，拦截日志：`"请求队列已满，UID=" + userId`

题目8（简答）：`RedisService.java` 中 `tryLock` 方法如何确保分布式锁的原子性创建？请结合 `SetParams` 参数说明。
难度：中等
答案：

- **原子性保障**：通过 Redis 的 `SET key value NX EX seconds` 命令实现，该命令是原子操作。
- **SetParams 配置**：`params.nx().ex(expireSeconds)` 同时设置 `NX`（仅当key不存在时才设置）和 `EX`（设置过期时间，单位秒），避免锁被永久持有。
- **返回值判断**：若返回 `"OK"` 表示加锁成功，否则失败。

题目9（简答）：`OrderManager.java` 的 `createOrder` 方法中，订单状态如何从 `CREATED` 自动流转到 `PAID` 和 `FINISHED`？请说明调度机制和时间延迟计算方式。
难度：中等
答案：

- **调度机制**：使用 `ScheduledExecutorService`（`scheduler`）异步调度状态更新任务。
- **时间延迟**：`payDelay` 在1~3秒随机（`1 + (int)(Math.random() * 3)`），`finishDelay` 在1~2秒随机（`1 + (int)(Math.random() * 2)`）。
- **流转流程**：创建订单后立即调度一个任务，在 `payDelay` 秒后执行 `updateStatus(orderId, OrderStatus.PAID)`；然后在 `finishDelay` 秒后执行 `updateStatus(orderId, OrderStatus.FINISHED)`。

题目10（简答）：`SlidingWindowRateLimiter.java` 的 `tryAcquire` 方法如何实现滑动窗口限流？请描述 `requestTimestamps` 队列的清理逻辑和限流判断依据。
难度：中等
答案：

- **核心逻辑**：维护一个 `ConcurrentLinkedQueue<Long>` 存储请求时间戳（毫秒）。
- **清理逻辑**：每次调用时，计算 `oneSecondAgo = now - 1000`，循环移除队列头部所有小于 `oneSecondAgo` 的过期时间戳，确保队列只保留最近1秒内的请求。
- **限流判断**：清理后若队列大小 `< maxRequestsPerSecond`，则允许请求并将当前时间戳入队；否则拒绝。

题目11（简答）：`BloomFilterUtil.java` 的 `add` 和 `contains` 方法采用相同的哈希计算逻辑，请写出该哈希公式并解释为何 `Math.abs` 和取模操作能确保不越界。
难度：中等
答案：

- **哈希公式**：`hash = Math.abs(value.hashCode() * hashSeed) % size`（`hashSeed=7`）。
- **不越界保障**：`Math.abs` 确保哈希值为非负数；`% size` 将结果映射到 `[0, size-1]` 范围内，保证不会访问 `BitSet` 越界。
- **潜在问题**：当 `value.hashCode() * 7` 发生整数溢出时，`Math.abs` 可能返回负数（如 `Integer.MIN_VALUE * 7`），但取模后仍在范围内，实际不影响功能。

题目12（简答）：`SeckillSystem.java` 中 `processStockAndOrder` 方法在执行库存扣减和订单创建时，为何需要先获取分布式锁？请说明锁的key、value设计目的及 `finally` 块的作用。
难度：中等
答案：

- **加锁目的**：防止多个worker线程同时扣减同一商品库存导致超卖。锁的key为 `"lock:" + stockKey`（如 `lock:item:1`），确保同一商品串行处理。
- **value设计**：`lockValue = UUID.randomUUID().toString()`，保证每个线程持有的锁值唯一，防止误释放其他线程的锁。
- `finally` **作用**：无论业务逻辑是否成功或抛出异常，都在 `finally` 中调用 `redisService.releaseLockSafe(lockKey, lockValue)` 释放锁，避免死锁。

题目13（简答）：`SeckillAppContext.java` 的 `stop` 方法中，为何需要 `Thread.sleep(5000)` 再关闭资源？ `system.shutdown()` 具体关闭了哪些组件？
难度：中等
答案：

- **休眠目的**：等待 `OrderManager` 中已调度的订单状态更新任务（PAID/FINISHED）异步执行完成，防止任务被中断。
- `system.shutdown()` **内容** ：① `executor.shutdown()` 停止8个worker线程；② `orderManager.shutdownScheduler()` 关闭订单状态调度线程池（4个线程），释放资源。

题目14（简答）：`OrderManager.java` 使用 `ConcurrentHashMap<Long, OrderNode>` 存储订单，但代码注释提到"无需链表结构"。请解释为何原本可能需要链表，以及当前设计如何保证线程安全的订单存储和查询。
难度：中等
答案：

- **链表结构场景**：若需按创建顺序遍历订单（如批量处理），可用链表维护顺序；但当前仅需通过 `orderId` 快速查询，链表非必需。
- **线程安全**：`ConcurrentHashMap` 本身支持高并发读写，`orderMap.put(orderId, node)` 和 `orderMap.get(orderId)` 均为原子操作，无需额外同步。
- **ID生成**：`AtomicLong orderIdGenerator` 保证订单ID全局唯一且线程安全。

题目15（简答）：`LogUtil.java` 的 `initLogging` 方法中，如何确保日志输出到文件和控制台时格式统一？请说明 `Formatter` 的实现逻辑和日志文件名。
难度：中等
答案：

- **统一格式**：创建 `simpleFormatter` 实例（匿名内部类），在 `format` 方法中自定义格式：`yyyy-MM-dd HH:mm:ss [LEVEL] LoggerName - Message`，并同时设置给 `fileHandler` 和 `consoleHandler`。
- **日志文件**：`logs/seckill.log`，通过 `FileHandler fileHandler = new FileHandler("logs/seckill.log", true)` 创建，`true` 表示追加模式。
- **编码**：两者均设置 `setEncoding("UTF-8")` 防止中文乱码。

三、代码细节题（共5题）

题型说明：针对具体代码行、返回值、常量值、边界条件等细节进行判断或填空，要求精确到代码字面。

题目16（判断）：`SeckillSystem.java` 中 `lotteryWinRate` 被设置为 `1.0`，这意味着所有通过前三层过滤的请求都一定会被判定为"中签"并进入队列。（）
难度：基础
答案：✓ 正确。代码 `if(Math.random() > lotteryWinRate)` 中，`Math.random()` 返回 `[0.0, 1.0)` 的随机数，由于 `lotteryWinRate=1.0`，条件永不成立，所有请求都会通过抽签环节。

题目17（填空）：`RedisService.java` 的 `decreaseStockSafe` 方法中，Lua脚本返回 `-1` 表示 或；若扣减成功则返回 。
难度：基础
答案：库存未初始化（key不存在）；库存已耗尽（`stock <= 0`）；扣减后的剩余库存量（`DECR` 结

果）。脚本逻辑：`if not stock then return -1; end; if stock <= 0 then return -1; end; return redis.call('DECR', KEYS[1]);`

题目18（判断）：`OrderManager.java` 中 `orderIdGenerator` 被声明为 `AtomicLong` 但未初始化，因此首次调用 `incrementAndGet()` 时会从0开始。（）
难度：基础
答案：✗ 错误。`AtomicLong` 虽未显式赋值，但成员变量有默认值 `0L`，因此 `incrementAndGet()` 首次返回 `1`，订单ID从1开始。

题目19（填空）：`BloomFilterUtil.java` 中 `hashSeed` 的值为，该值的作用是对 `hashCode()` 结果进行，以降低哈希冲突概率。
难度：基础
答案：7；乘数扰动（或二次哈希）。通过 `value.hashCode() * hashSeed` 扩大哈希分布范围，减少不同字符串映射到同一位置的概率。

题目20（判断）：`SlidingWindowRateLimiter.java` 中，`requestTimestamps` 使用 `ConcurrentLinkedQueue` 是为了保证多线程下时间戳入队和出队的线程安全，但 `size()` 方法在判断限流时可能存在瞬时数据不一致风险。（）
难度：高等
答案：✓ 正确。`ConcurrentLinkedQueue` 的 `size()` 需要遍历队列，非O(1)且非精确同步（弱一致性），在高并发下 `size() < maxRequestsPerSecond` 判断可能基于过期数据，导致瞬时限流失效。建议使用 `AtomicInteger` 计数器优化。

四、综合应用题（共4题）

题型说明：需结合并发场景排查潜在问题、优化代码逻辑或推导异常行为，考查深度理解和工程实践能力。

题目21（问题排查）：基于 `RedisService.java` 的 `decreaseStockSafe` 方法和 `SeckillSystem.java` 的 `processStockAndOrder` 方法，若某商品库存为1，同时有10个worker线程并发扣减，分析是否可能发生超卖？请列出所有可能的风险点及代码中的防护机制。
难度：高等
答案：

- **结论**：**不会发生超卖。**
- **防护机制**：
    1. **分布式锁**：同一商品（`stockKey`）的扣减操作由 `tryLock(lockKey, lockValue, 5)` 串行化，确保同一时间只有一个线程执行Lua脚本。
    2. **Lua原子性**：`decreaseStockSafe` 中Lua脚本在Redis服务端单线程执行，判断库存和扣减是原子操作，返回 `-1` 或新库存。
    3. **双重校验**：Worker线程先获取锁，再执行Lua脚本，即使锁过期（5秒），Lua脚本的判断也能防止超卖。
- **潜在风险点**：
    - **锁超时**：若业务逻辑（订单创建）耗时超过5秒，锁提前释放可能导致下一个线程进入，但Lua脚本的库存检查仍是最终防线。

- **网络分区**：Redis连接中断可能导致锁无法释放，但Lua脚本执行是原子的，已扣减的库存不会回滚。

题目22（代码改写）：`BloomFilterUtil.java` 当前实现存在哈希冲突率较高的问题。若要求将误判率控制在1%以内，容量为10万，请改写该类以支持 **多个哈希函数**（提示：可使用Guava的Hashing或自定义2-3个哈希函数），并说明如何计算最优哈希函数个数。
难度：高等
答案：

```
// 改写后的 BloomFilterUtil.java
public class BloomFilterUtil {
    private final BitSet bitSet;
    private final int size;
    private final int hashFunctionCount; // 哈希函数数量

    public BloomFilterUtil(int capacity, double falsePositiveRate) {
        // 计算最优size和hashFunctionCount
        this.size = (int) (-capacity * Math.log(falsePositiveRate) / (Math.log(2)
* Math.log(2)));
        this.hashFunctionCount = (int) (size / capacity * Math.log(2));
        this.bitSet = new BitSet(size);
    }

    public void add(String value) {
        for (int i = 0; i < hashFunctionCount; i++) {
            int hash = hash(value, i);
            bitSet.set(hash);
        }
    }

    public boolean contains(String value) {
        for (int i = 0; i < hashFunctionCount; i++) {
            int hash = hash(value, i);
            if (!bitSet.get(hash)) return false;
        }
        return true;
    }

    private int hash(String value, int seed) {
        // 使用双重哈希策略：hash1 + seed * hash2
        int hash1 = value.hashCode();
        int hash2 = Integer.rotateLeft(hash1, seed * 5); // 简单扰动
        return Math.abs((hash1 + seed * hash2) % size);
    }
}
```

- **最优哈希函数个数公式**：`k = (m/n) * ln2`，其中 `m` 为位数组大小，`n` 为预期容量。`m = -n * ln(p) / (ln2)^2`，代入 `n=100000, p=0.01` 得 `m ≈ 958505` 位，`k ≈ 7` 个哈希函数。

题目23（问题排查）：`SlidingWindowRateLimiter.java` 的 `tryAcquire` 方法在高并发（如10万QPS）下可能成为性能瓶颈。请分析其时间复杂度和并发问题，并提出 **两种优化方案**（要求：线程安全且O(1)时间复杂度）。
难度：高等
答案：

- **性能瓶颈分析**：
  - **时间复杂度**：`while (!requestTimestamps.isEmpty() && requestTimestamps.peek() < oneSecondAgo)` 循环清理 + `size()` 遍历统计，最坏O(n)，n为每秒请求数。
  - **并发问题**：多线程同时清理和入队，`ConcurrentLinkedQueue` 的 `size()` 和 `poll()` 存在CAS竞争，高并发下CPU飙升。
- **优化方案**：
  方案1：令牌桶算法

```
public class TokenBucketRateLimiter {
    private final AtomicInteger tokens;
    private final int capacity;
    private final long refillInterval = 1000; // 1秒
    private volatile long lastRefillTime = System.currentTimeMillis();

    public boolean tryAcquire() {
        refill();
        int current = tokens.get();
        if (current > 0) {
            return tokens.compareAndSet(current, current - 1);
        }
        return false;
    }

    private void refill() {
        long now = System.currentTimeMillis();
        if (now - lastRefillTime >= refillInterval) {
            tokens.set(capacity); // 重置令牌
            lastRefillTime = now;
        }
    }
}
```

  方案2：时间窗口计数器（Atomic + 数组）

```
public class ArraySlidingWindowRateLimiter {
    private final AtomicInteger[] windows = new AtomicInteger[60]; // 60个槽位，每100ms一个
    private final AtomicLong lastWindow = new AtomicLong(0);

    public boolean tryAcquire() {
        long now = System.currentTimeMillis();
        int index = (int) ((now / 100) % 60);
        long currentWindow = now / 100;
        if (lastWindow.get() != currentWindow) {
            windows[index].set(0);
            lastWindow.set(currentWindow);
        }
```

```
            return windows[index].incrementAndGet() <= maxRequestsPerSecond / 60;
        }
    }
```

题目24（代码改写）：当前 `SeckillSystem.java` 使用固定大小线程池（8个worker）和容量为10000的 `LinkedBlockingQueue` 处理请求。若秒杀活动持续60秒，总请求量达100万，请改写 `startWorkers` 和 `processRequest` 方法，实现 **动态扩容/缩容** 线程池（基于队列长度）和 **拒绝策略**（当队列满时返回自定义JSON响应而非简单日志）。
难度：高等
答案：

```
// 改写后的 SeckillSystem.java 关键部分
private final ThreadPoolExecutor executor;
private final BlockingQueue<Request> requestQueue;

public SeckillSystem() {
    // 核心线程8，最大线程32，空闲30秒回收，队列容量10000
    this.requestQueue = new LinkedBlockingQueue<>(10000);
    this.executor = new ThreadPoolExecutor(
        8, 32, 30, TimeUnit.SECONDS,
        new LinkedBlockingQueue<>(10000),
        new ThreadPoolExecutor.CallerRunsPolicy() // 队列满时由主线程执行
    );
    // 动态监控线程池

 Executors.newSingleThreadScheduledExecutor().scheduleAtFixedRate(this::adjustThr
eadPool, 10, 10, TimeUnit.SECONDS);
}

private void adjustThreadPool() {
    int queueSize = requestQueue.size();
    int poolSize = executor.getPoolSize();
    if (queueSize > 8000 && poolSize < executor.getMaximumPoolSize()) {
        executor.setCorePoolSize(Math.min(poolSize + 4,
executor.getMaximumPoolSize()));
    } else if (queueSize < 2000 && poolSize > executor.getCorePoolSize()) {
        executor.setCorePoolSize(Math.max(poolSize - 2,
executor.getCorePoolSize()));
    }
}

public String processRequest(long userId, int productId) {
    // ... 前面过滤逻辑不变 ...
    // Step5: 队列容量限制（改写为返回JSON）
    if (!requestQueue.offer(new Request(userId, productId))) {
        return "{\"code\": 503, \"msg\": \"系统繁忙，请重试\", \"userId\": " +
userId + "}";
    }
    return "{\"code\": 202, \"msg\": \"请求已接收\", \"userId\": " + userId + "}";
}

public void shutdown() {
    // 优雅关闭：先不接受新请求，再等待任务完成
```

```
    executor.shutdown();
    try {
        if (!executor.awaitTermination(60, TimeUnit.SECONDS)) {
            executor.shutdownNow();
        }
    } catch (InterruptedException e) {
        executor.shutdownNow();
    }
    orderManager.shutdownScheduler();
}
```

- **动态扩容策略**：每10秒检查队列长度，>8000时增加核心线程，<2000时减少核心线程，避免资源浪费。
- **拒绝策略**：队列满时不丢弃，由 `CallerRunsPolicy` 让主线程执行，确保请求不丢失，同时返回JSON提升客户端体验。

题目覆盖统计

- **所有类均覆盖**：`SeckillApplication` (1)、`SeckillAppContext` (2,6,13)、`SeckillSystem` (7,12,16,20,24)、`RedisService` (4,8,17,21)、`OrderManager` (9,14,18)、`BloomFilterUtil` (5,11,19,22)、`SlidingWindowRateLimiter` (10,15,20,23)、`LogUtil` (15)、`OrderNode/OrderStatus/Request` (3,7)。
- **关键技术全覆盖**：库存初始化、恶意用户拦截、滑动窗口限流、布隆过滤器去重、Lua脚本防超卖、分布式锁、订单状态流转、日志初始化、线程池管理。