

# Java

## 高手笔记



你踩过的坑，  
都是你的勋章

极客时间教研部 / 著



# Java 高手笔记 | 使用指南

极客时间

极客时间

《Java 高手笔记》这本小册子，创意来自于极客时间上朱晔老师的《Java 业务开发常见错误 100 例》。课程里涉及了 100 多个案例，朱晔老师将其重新梳理成了 123 个 Java 开发常见的踩坑点，并给出了每个点对应的原因分析和解决方案。

那这样一个踩坑笔记本，有什么价值呢？

作为程序员，我们每天都在和代码打交道，除了要修复无数的 Bug，还要趟过无数的“坑”。踩坑是一件痛苦的事情，但是如果能够把这些“坑”变成经验和业务能力，踩坑反而会成为走向优秀程序员的“捷径”。

这个笔记本最大的使命就是带着你用好这条“捷径”：

- 快速了解写 Java 代码时可能会踩哪些坑，提前避坑；
- 养成及时记录和复盘的习惯，追溯问题根因、总结经验，获得更快的成长；
- 提醒自己不要犯同样的错，做一个“不贰过”的大家。

具体来说，你可以这样来实践：

极客时间

- 踩坑复盘。

打开目录页，对照列出的坑点，回想自己是否经历过。如果有，试着复盘当时的业务场景，想想自己是如何解决的，然后再翻到对应的笔记页，看老师的分析和解决方法，对比异同，并记录在右侧的空白页上。

- 记录自己新踩的坑。

如果新踩的坑包含在这 123 个坑点内，那就在对应的右侧空白页上记录下来，建议你写清楚时间、业务场景、解决思路和心得，便于复习。如果是一个全新的坑，那就记录在笔记本最后留给你的空白页里。

没踩过坑的开发人生不值得过，那就相信记录和复盘的力量，把你踩过的每一个坑，都变成你的勋章。



扫码了解这 123 个坑点  
背后的代码案例

## CONTENTS 目录

# A 代码篇

### 并发工具

02-04

- A01** 在 Web 环境中使用 ThreadLocal 出现数据错乱的坑
- A02** 使用了 ConcurrentHashMap 但还是出现了线程安全问题
- A03** 使用了 ConcurrentHashMap 但却没有发挥性能优势
- A04** 在不合适的场景下使用 CopyOnWriteArrayList 导致的性能问题

### 代码加锁

06-08

- A05** 没有理清清楚线程安全问题的所在点，导致锁无效
- A06** 加锁没有考虑锁的粒度，可能导致性能问题
- A07** 加锁没有考虑锁的场景，可能导致性能问题
- A08** 多把锁时，要格外小心死锁问题（VisualVM）

### 线程池

10-12

- A09** 使用 Executors 声明线程池导致两种类型的 OOM
- A10** 线程池线程管理策略详解：如何实现一个更激进的线程池？
- A11** 没有复用线程池，导致频繁创建线程的事故
- A12** 混用线程池，导致性能问题
- A13** CallerRunsPolicy 拒绝策略可能带来的问题

### 连接池

14-16

- A14** 你知道常见的 Client SDK 的 API，有哪 3 种形式吗？
- A15** 在多线程环境下使用 Jedis 出现的线程安全问题
- A16** 不复用 Apache CloseableHttpClient 会导致什么问题？（jstack、Isof、Wireshark）
- A17** 小心数据库连接池打满后出现的性能问题（JConsole）

极客时间

### HTTP 调用

18-22

- A18** 连接超时和读取超时的 5 个认知误区
- A19** Spring Cloud Feign 和 Ribbon 配合使用，设置超时的三个坑
- A20** Spring Cloud Ribbon 居然会自动重试我的接口？
- A21** 小心 Apache HttpClient 对并发连接数的限制

### 数据库事务

24-26

- A22** 两种错误写法导致 Spring 声明式事务未生效的两个坑
- A23** 事务即便生效也不一定能回滚的两个情况
- A24** 不出异常，事务居然也不会提交？

### 数据库索引

28-30

- A25** 聚簇索引的 3 个要点
- A26** 考虑二级索引的维护、空间和回表代价
- A27** 建了索引但是用不上的 3 种情况
- A28** 多个独立索引和联合索引如何选择的问题
- A29** 相同的 SQL 语句，有的时候能走索引有的时候不走索引的现象（MySQL optimizer trace）

## 判等问题

32-36

- A30** 什么时候不能使用 == 进行值判等 (-XX:+PrintStringTableStatistic)
- A31** 实现 equals 方法可能出现的诸多坑
- A32** equals 和 hashCode 没有配对实现的坑
- A33** equals 和 compareTo 实现逻辑不一致的坑
- A34** Lombok @EqualsAndHashCode 可能的两个坑

## 数值计算

38-40

- A35** 使用 double 进行浮点数运算的坑
- A36** 对 double 或 float 进行舍入格式化的坑
- A37** 使用 equals 对 BigDecimal 进行判等的坑
- A38** 把 BigDecimal 作为 Key 加入 HashSet 的坑
- A39** 小心数值溢出但是没有任何异常的坑

## 集合类

42-44

- A40** 使用 Arrays.asList 把数据转换为 List 的 3 个坑
- A41** 使用 List.subList 进行切片操作居然会导致 OOM
- A42** 使用数据结构需要考虑平衡时间和空间 (MAT)
- A43** LinkedList 适合什么场景呢?

## 空值处理

46-50

- A44** 注意 5 种可能出现空指针的情况 (Arthas)
- A45** POJO 字段设置默认值导致数据库中的原始值被覆盖的坑
- A46** 你见过数据库中出现 null 字符串的问题吗?

- A47** 客户端不传值以及传 null 在 POJO 中都是 null, 如何区分?

- A48** MySQL 中涉及 NULL 的 3 个容易忽略的坑

## 异常处理

52-56

- A49** 仅在框架层面粗犷捕获和处理异常, 合适吗?
- A50** 3 种错误的异常处理方式
- A51** 在 finally 中抛出异常, 会怎么样?
- A52** 异常是否可以定义为静态变量呢?
- A53** 线程池通过 execute 提交任务, 出现异常会怎么样?
- A54** 线程池通过 submit 提交任务, 出现异常会怎么样?

## 日志

58-62

- A55** 你能区分 SLF4J 的桥梁类库和绑定类库吗?
- A56** 配置不当导致日志重复记录的两个坑
- A57** 异步日志可能撑爆内存的问题
- A58** 异步日志可能出现日志丢失的问题
- A59** 异步日志还是会阻塞方法的问题
- A60** 使用 {} 占位符语法记录日志的意义, 在于什么?

## 文件 IO

64

- A61** 为什么读写文件会遇到乱码问题?
- A62** 使用 Files.lines 等返回 Stream 方法进行 IO 操作的大坑 (IsOf)
- A63** 设置缓冲区, 对读写文件的性能影响有多大?

## 序列化

66-70

- A64 使用 RedisTemplate 保存数据，Redis 中出现乱码问题
- A65 使用 RedisTemplate 出现 ClassCastException 的问题
- A66 枚举作为 API 参数或返回值的两个大坑
- A67 注意 Jackson 反序列化对额外字段的处理
- A68 自定义 ObjectMapper 可能会覆盖 Spring Bean 的问题
- A69 注意反序列化可能不会调用自定义的构造方法的坑

## 日期时间

72-76

- A70 初始化日期时间的各种坑
- A71 SimpleDateFormat 的 3 个坑
- A72 时区问题导致日期时间错乱的诸多坑
- A73 使用时间戳进行日期时间计算的坑
- A74 Java 8 的日期时间类型

## OOM

78-80

- A75 太多份相同的对象导致的 OOM 坑 (MAT)
- A76 使用 WeakHashMap 居然也会出现 OOM ?
- A77 Tomcat 参数不合理导致的 OOM 血案 (MAT)

## Java 高级特性

82-84

- A78 通过反射调用方法，是传参决定方法重载吗？
- A79 泛型经过类型擦除覆写失败的坑
- A80 泛型经过类型擦除会多出桥接方法的坑 (javap、jclasslib)

- A81 注解即使加上了 @Inherited，也无法从接口和方法继承的问题

极客时间

## Spring 框架

86-88

- A82 注意 Bean 默认单例的问题
- A83 单例的 Bean 注入多例的 Bean，不仅仅是设置 Scope 这么简单
- A84 自定义 Aspect 因为顺序问题导致 Spring 事务失效的坑
- A85 Feign AOP 切不到的诡异案例
- A86 配置文件中的配置不生效的问题

极客时间

# B 设计篇

## 代码重复

92-94

- B01 利用工厂模式 + 模板方法模式，消除 if...else 和大量重复代码的例子
- B02 利用注解 + 反射消除重复代码的例子
- B03 利用属性拷贝工具消除重复代码的例子

## 接口设计

96-98

- B04 接口的响应要明确表示接口的处理结果
- B05 要考虑接口变迁的版本控制策略
- B06 接口同步 / 异步处理方式要明确

## 缓存设计

100–102

- B07** 把 Redis 当成数据库，会有什么问题？
- B08** 小心缓存雪崩问题
- B09** 小心缓存击穿问题
- B10** 小心缓存穿透问题
- B11** 应该先更新缓存还是先更新数据库？

## 生产就绪

104

- B12** 如何写一个正确的健康检测接口？
- B13** 如何对外暴露应用内部重要组件的状态？
- B14** 为什么指标 Metrics 对于问题定位这么重要？

## 异步处理

106–108

- B15** 为什么异步处理中消息补偿这么重要？
- B16** 为什么异步处理需要实现操作幂等？
- B17** 消息模式配置错误导致消息重复或遗漏的坑
- B18** 消息队列被死信堵塞的坑

## 数据存储

110–112

- B19** 与 MySQL 相比，Redis 更擅长什么不擅长什么？
- B20** 与 MySQL 相比，Elasticsearch 更擅长什么不擅长什么？
- B21** 与 MySQL 相比，InfluxDB 更擅长什么不擅长什么？
- B22** 结合 NoSQL 和 RDBMS 可以应对大并发的存储方案

# C 安全篇

## 数据源头

116–118

- C01** 客户端的计算不可信
- C02** 客户端提交的参数需要校验
- C03** 不能信任请求头里的任何内容
- C04** 用户标识不能从客户端获取

## 安全兜底

120–122

- C05** 如何防止平台资源被刷？
- C06** 如何防止虚拟资产无限发放？
- C07** 如何防止重复的资金？

## 数据和代码

124–126

- C08** SQL 注入如何实现拖库？（sqlmap）
- C09** 小心代码注入问题
- C10** 全方位堵漏 XSS 的四招

## 敏感数据

128–132

- C11** 使用 MD5 摘要保存密码是否安全？
- C12** MD5 摘要加盐的错误方式
- C13** 了解 BCryptPasswordEncoder
- C14** 注意使用安全的对称加密算法
- C15** 使用 AES-256-GCM 和单独的加密服务来加密敏感数据的例子



# 代码篇



## 并发工具

### A01

#### 在 Web 环境中使用 ThreadLocal 出现数据错乱的坑

**原因：**线程可能重用，导致 ThreadLocal 中的数据会串

**解决：**用完及时清空数据，比如可以自定义 HandlerInterceptorAdapter，在 preHandle 的时候去设置 ThreadLocal，在 afterCompletion 时去 remove

### A02

#### 使用了 ConcurrentHashMap 但还是出现了线程安全问题

**原因：**ConcurrentHashMap 只能保证提供的原子性读写操作（比如 putIfAbsent、computeIfAbsent、replace、compute）是线程安全的

**解决：**如果需要确保多个原子性操作整体线程安全，需要自己加锁解决

**补充：**诸如 size、isEmpty 和 containsValue 等聚合方法，在并发情况下可能会反映 ConcurrentHashMap 的中间状态。因此在并发情况下，这些方法的返回值只能用作参考，而不能用于流程控制

### A03

使用了 ConcurrentHashMap 但却没有发挥性能优势

**原因：**仍然像 HashMap 那样使用加锁的方式，来使用 ConcurrentHashMap

**解决：**考虑使用 computeIfAbsent、putIfAbsent、getOrDefault 等 API 来提升性能

### A04

在不合适的场景下使用 CopyOnWriteArrayList 导致的性能问题

**原因：**CopyOnWriteArrayList 每次修改复制一份数据

**解决：**读多写少的场景才考虑 CopyOnWriteArrayList，写多的场景考虑 ArrayList

# 代码加锁

## A05

没有理清楚线程安全问题的所在点，导致锁无效

**原因 1：**没有识别线程安全问题的原因胡乱加锁

**原因 2：**锁是实例级别的，资源是类级别的，无法有效保护

**解决：**明确锁和要保护的资源的关系和范围

## A06

加锁没有考虑锁的粒度，可能导致性能问题

**原因：**加锁粒度太大，使得大段代码整体串行执行，出现性能问题

**解决：**尽可能降低锁的粒度，仅对必要的代码块甚至是需要保护的资源本身加锁

极客时间

极客时间 / 轻松学习 · 高效学习

极客时间

极客时间

极客时间

## A07

### 加锁没有考虑锁的场景，可能导致性能问题

**原因：**千篇一律使用写锁，可以根据场景更细化地使用高级锁

**解决 1：**考虑使用 StampedLock 的乐观读的特性，进一步提高性能

**解决 2：**对于读写比例差异明显的场景，使用 ReentrantReadWriteLock 细化区分读写锁

**解决 3：**在没有明确需求的情况下，不要轻易开启公平锁特性

## A08

### 多把锁时，要格外小心死锁问题（VisualVM）

**原因：**多把锁相互等待对方释放，导致死锁

**解决：**加锁的时候考虑顺序，按顺序加锁不易死锁

**工具：**使用 VisualVM 的线程 Dump，查看死锁问题并分析死锁原因

# 线程池

## A09

使用 Executors 声明线程池导致两种类型的 OOM

**原因 1:** newFixedThreadPool 使用无界队列，队列堆积太多数据导致 OOM

**原因 2:** newCachedThreadPool 不限制最大线程数并且使用没有任何容量的 SynchronousQueue 作为队列，容易开启太多线程导致 OOM

**解决:** 手动 new ThreadPoolExecutor，根据需求设置合适的核心线程数、最大线程数、线程回策略、队列、拒绝策略，并对线程进行明确的命名以方便排查问题

## A10

线程池线程管理策略详解：如何实现一个更激进的线程池？

**原因:** Java 的线程池倾向于优先使用队列，队列满了再开启更多线程

**解决:** 重写队列的 offer 方法直接返回 false，数据不入队列，并且自定义 RejectedExecutionHandler，触发拒绝策略的时候再把任务加入队列；参考 Tomcat 的 ThreadPoolExecutor 和 TaskQueue 类

## A11

## 没有复用线程池，导致频繁创建线程的事故

**原因：**获取线程池的方法每次都返回一个 `newCachedThreadPool`，好在 `newCachedThreadPool` 可以闲置回收

**解决：**使用静态字段定义线程池，线程池务必重用

极客时间

## A12

## 混用线程池，导致性能问题

**原因：**IO 绑定操作和 CPU 绑定操作混用一个线程池，前者因为负担重，线程长期处于忙的状态，导致 CPU 操作吞吐受到影响

**解决：**根据任务的类型声明合适的线程池，不同类型的任务考虑使用独立线程池

极客时间

**扩展：**Java 8 的 `ParallelStream` 背后是一个公共线程池，别把 IO 任务使用 `ParallelStream` 来处理

## A13

## CallerRunsPolicy 拒绝策略可能带来的问题

**原因：**如果设置 `CallerRunsPolicy`，那么被拒绝的任务会由提交任务的线程运行，可能会在线程池满载的情况下直接拖垮整个应用

**解决：**对于 Web 和 Netty 场景，要仔细考虑把任务提交到线程池异步执行使用的拒绝策略，除非有明确的需求，否则不考虑使用 `CallerRunsPolicy` 拒绝策略

## 连接池

### A14

你知道常见的 Client SDK 的 API, 有哪 3 种形式吗?

#### 形式 1:

极客时间

内部带有连接池的 API: SDK 内部会先自动通过连接池获取连接 (几乎所有的数据库连接池, 都是这一类)

#### 形式 2:

连接池和连接分离的 API: 使用者先通过连接池获取连接, 再使用连接执行操作 (Jedis)

极客时间

#### 形式 3:

非连接池的 API: 非线程安全, 需要使用者自己封装连接池

极客时间

### A15

在多线程环境下使用 Jedis 出现的线程安全问题


**原因:** Jedis 是连接池和连接分离的 API, Jedis 类代表连接, 不能多线程环境下使用

**解决:** 每次使用 JedisPool 先获取到一个 Jedis, 然后再调用 Jedis 的 API, 通过 addShutdownHook 来关闭 JedisPool

**A16**

不复用 Apache CloseableHttpClient 会导致什么问题？（jstack、Isof、Wireshark）

连接池如果不复用，代价可能会比每次创建单个连接还要大：

- 连接池可能每次都会创建一定数量的初始连接 
- 连接池可能会有些管理模块，需要创建单独的线程来管理

工具：

- 使用 jstack 观察到没有复用连接池，会出现大量的 Connection evictor 线程 
- 使用 Isof 观察到没有复用连接池，会出现大量 TCP 连接
- 如果复用 CloseableHttpClient，使用 Wireshark 观察 HTTP 请求重用 TCP 连接的过程 

**A17**

小心数据库连接池打满后出现的性能问题（JConsole）

**原因：**数据库连接池最大连接数设置得太小，很可能会因为获取连接的等待时间太长，导致吞吐量低下甚至超时无法获取连接

**解决：**对类似数据库连接池的重要资源进行持续检测，并设置一半的使用量作为报警阈值，出现预警后及时扩容

**工具：**使用 JConsole 来观察 Hikari 连接池的 MBean，监控活跃连接、等待线程等数据

**扩展：**修改配置参数务必验证是否生效，并且在监控系统中确认参数是否生效、是否合理，避免明明使用的是 Hikari 连接池，却还在调整 Druid 连接池的参数情况



# HTTP 调用

## A18

### 连接超时和读取超时的 5 个认知误区

#### 误区 1：连接超时配置得特别长

极客时间

分析：连接超时很可能是因为防火墙等原因彻底连接不上，不太会出现连接特别慢的现象，不用设置太长

#### 误区 2：排查连接超时问题，却没理清连的是哪里

分析：很可能客户端连接的是 Nginx，不是实际的后端服务，从后端服务层面排查问题没用

极客时间

#### 误区 3：认为出现了读取超时，服务端的执行就会中断

分析：客户端读取超时，服务端业务逻辑还会持续运行，不能随意假设服务端处理是失败的

极客时间

误区 4：认为读取超时只是 Socket 网络层面的概念，是数据传输的最长耗时，故将其配置得非常短

分析：读取超时并不是数据在网络传输的时间，需要包含服务端业务逻辑执行的时间

误区 5：认为超时时间越长任务接口成功率就越高，将读取超时参数配置得太长

分析：读取超时设置太长，可能导致上游服务被下游拖垮，应该根据 SLA 设置合适的读取超时。有些时候，快速失败或熔断不是一件坏事儿

**A19**

Spring Cloud Feign 和 Ribbon 配合使用，设置超时的 3 个坑

**坑 1**

**原因：**默认情况下 Feign 的读取超时是 1 秒，这个时间过于短了

**解决：**根据自己的需要设置长一点

**坑 2**

**原因：**如果要配置 Feign 的读取超时，就必须同时配置连接超时，才能生效

**解决：**同时配置 `readTimeout` 和 `connectTimeout`

**坑 3**

**原因：**Ribbon 配置连接超时和读取超时的参数大小写和 Feign 略有不同

**解决：**Ribbon 使用 `ReadTimeout` 和 `ConnectTimeout`，注意大小写

**A20****Spring Cloud Ribbon 居然会自动重试我的接口?**

**原因:** Ribbon 对于 HTTP Get 请求在一个服务器调用失败后, 会自动到下一个节点重试一次

**解决:** 修改参数, 设置 `ribbon.MaxAutoRetriesNextServer=0`; 并且对于 Get 请求需要确保接口幂等

**A21****小心 Apache HttpClient 对并发连接数的限制**

**原因:** HttpClient 对连接数有限制, 默认一个域名 2 个并发, 所有域名 20 个并发

**解决:** 通过 `HttpClients.custom().setMaxConnPerRoute(10).setMaxConnTotal(20)` 来设置合适的值

## 数据库事务

### A22

两种错误写法导致 Spring 声明式事务未生效的  
两个坑

#### 坑 1

**原因：**因为 private 方法无法代理，所以为 private 方法设置 @Transactional 注解无法生效事务

**解决：**除非使用 AspectJ 做静态织入，否则需要确保只有 public 方法才设置 @Transactional 注解

#### 坑 2

**原因：**因为通过 this 自调用方法不走 Spring 的代理类，所以无法生效事务

**解决：**确保事务性方法从外部通过代理类调用。如果一定要从内部调用，就要重新注入当前类调用

## A23

### 事务即便生效也不一定能回滚的两个情况

#### 情况 1

**原因：**只有异常传播出了标记了 `@Transactional` 注解的方法，事务才能回滚

**解决：**避免 catch 住异常，或者通过 `TransactionAspectSupport.currentTransactionStatus().setRollbackOnly()` 手动回滚

#### 情况 2

**原因：**默认情况下，出现 `RuntimeException`（非受检异常）或 `Error` 的时候，Spring 才会回滚事务

**解决：**设置 `@Transactional(rollbackFor = Exception.class)`，来突破默认不回滚受检异常的限制

## A24

### 不出异常，事务居然也不会提交？

**原因：**默认事务传播策略是 `REQUIRED`，子方法会复用当前事务，子方法出异常后回滚当前事务，导致父方法也无法提交事务

**解决：**设置 `REQUIRES_NEW` 方式的事务传播策略，让子方法运行在独立事务中

# 数据库索引

## A25

### 聚簇索引的 3 个要点

#### 要点 1:

B+ 树，既是索引也是数据

#### 要点 2:

自动创建，只能有一个

#### 要点 3:

可以加速主键搜索和查询

极客时间

极客时间

极客时间

## A26

### 考虑二级索引的维护、空间和回表代价

#### 原因

**维护：**需要独立维护一棵 B+ 树，用来加速数据查询和排序  
考虑数据页的合并和分裂

**空间：**额外的存储空间

**回表：**二级索引不保存完整数据，只保存索引键和主键字段

#### 解决方案

按需创建

考虑联合索引

针对轻量级字段创建

极客时间

## A27

建了索引但是用不上的 3 种情况

情况 1: 查询数据内容不走左匹配

情况 2: 查询字段使用函数运算

情况 3: 查询没有使用联合索引最左边的列

极客时间

## A28

多个独立索引和联合索引如何选择的问题

考虑: 多个字段会在一个条件中查询, 并且更有可能走索引覆盖

不考虑: 永远只是单一列的查询

极客时间

极客时间

## A29

相同的 SQL 语句, 有的时候能走索引有的时候不走索引的现象 (MySQL optimizer trace)

原因: MySQL 根据不同查询方案的成本来决定是否走索引, 索引过滤效果不好的时候, 可能走全表扫描更划算

解决: 使用 EXPLAIN 来观察查询是否会走索引, 开启 optimizer trace 来了解具体原因和成本明细

## 判等问题

### A30

什么时候不能使用 == 进行值判等  
(-XX:+PrintStringTableStatistic)

#### 基本类型之间

只能通过 == 判等

#### 引用类型之间

结论：不能通过 == 判等，需要使用 equals 方法

原因：== 是判断指针相等、判断对象实例是否是同一个，不代表对象内容是否相同

为什么有的时候 String 使用 == 判等会奏效？

原因：直接使用双引号声明出来的两个 String 对象指向常量池中的相同字符串

工具：使用 -XX:+PrintStringTableStatistic 查看字符串常量表

#### 包装类型之间

Integer 有的时候使用 == 判等会奏效？

原因：Integer 内部其实做了缓存，默认缓存 [-128, 127]，在这个区间 == 奏效



### A31

#### 实现 equals 方法可能出现的诸多坑

**原因：**考虑性能先进行指针判等、需要对另一方进行判空、确保类型相同的情况下再进行类型强制转换

**解决：**使用 IDE 帮我们生成代码

极客时间

### A32

#### equals 和 hashCode 没有配对实现的坑

**原因：**对象存入哈希表的行为不可测

**解决：**务必配对实现，使用 IDE 帮我们生成代码

极客时间

### A33

#### equals 和 compareTo 实现逻辑不一致的坑

**原因：**ArrayList.indexOf 使用 equals 判断对象是否相等，而 Collections.binarySearch 使用 compareTo 方法来比较对象以实现搜索

**解决：**对于自定义的类型，如果实现 Comparable，记得 equals、hashCode、compareTo 三者逻辑一致

**A34****Lombok @EqualsAndHashCode 可能的两个坑****坑 1**

**原因：**Lombok 的 @EqualsAndHashCode 注解实现 equals 和 hashCode 的时候，默认使用类的所有非 static、非 transient 的字段

**解决：**使用 @EqualsAndHashCode.Exclude 排除一些字段

**坑 2**

**原因：**Lombok 的 @EqualsAndHashCode 注解实现 equals 和 hashCode 的时候，默认不考虑父类

**解决：**设置 callSuper = true

## 数值计算

### A35

使用 double 进行浮点数运算的坑

原因：浮点数无法精确存储，计算会损失精度

极客时间

解决：

- 使用 BigDecimal 表示和计算浮点数，且务必使用字符串的构造方法来初始化 BigDecimal
- 如果一定要用 Double 来初始化 BigDecimal 的话，可以使用 BigDecimal.valueOf 方法

极客时间

### A36

对 double 或 float 进行舍入格式化的坑

原因：使用 double 或 float 配合 String.format 或 DecimalFormat 进行格式化舍入，也会有精度问题

极客时间

解决：还是需要使用 BigDecimal，配合 setScale 进行舍入

### A37

#### 使用 equals 对 BigDecimal 进行判等的坑

**原因：**使用 equals 比较 BigDecimal 会同时比较 value 和 scale，1.0 不等于 1，和我们想的不一样

**解决：**如果只比较 BigDecimal 的 value，可以使用 compareTo 方法

### A38

#### 把 BigDecimal 作为 Key 加入 HashSet 的坑

**原因：**BigDecimal 的 equals 和 hashCode 方法会同时考虑 value 和 scale，BigDecimal 的值 1.0 和 1，虽然 value 相同但是 scale 不同

**解决：**使用 TreeSet 替换 HashSet，或者先使用 stripTrailingZeros 方法去掉尾部的零

### A39

#### 小心数值溢出但是没有任何异常的坑

**原因：**大数值计算，数值可能默默溢出，没有任何异常


**解决：**

- 使用 Math 类的 addExact、subtractExact 等 xxExact 方法进行数值运算，在溢出时能抛出异常
- 使用大数类 BigInteger，需要转到 Long 时使用 longValueExact，在溢出时能抛出异常

## 集合类


### A40

使用 Arrays.asList 把数据转换为 List 的 3 个坑


**坑 1:** 不能直接使用 Arrays.asList 来转换基本类型数组 

**原因:** 只能是把 int 装箱为 Integer，不可能把 int 数组装箱为 Integer 数组，int 数组整体作为了一个对象成为了泛型类型 T

**解决:** 使用 Arrays.stream 或传入 Integer[]

**坑 2:** Arrays.asList 返回的 List 不支持增删操作 

**原因:** Arrays.asList 返回的 List 并不是我们期望的 java.util.ArrayList，而是 Arrays 的内部类 ArrayList

**解决:** 重新 new 一个 ArrayList 初始化 Arrays.asList 返回的 List 

**坑 3:** 对原始数组的修改会影响到我们获得的那个 List

**原因:** 内部 ArrayList 其实是直接使用了原始的数组

**解决:** 重新 new 一个 ArrayList 初始化 Arrays.asList 返回的 List

## A41

使用 List.subList 进行切片操作居然会导致 OOM

**原因：** List.subList 返回的是内部类 SubList 会引用原始 List，SubList 有强引用时会导致原来的 List 也无法 GC

**解决：**

- 不直接使用 subList 方法返回的 SubList，而是使用 new ArrayList 来重新构建一个普通的 ArrayList
- 对于 Java 8 使用 Stream 的 skip 和 limit API 来跳过流中的元素，以及限制流中元素的个数

极客时间

极客时间

极客时间

## A42

使用数据结构需要考虑平衡时间和空间（MAT）

**时间：** 要实现快速查询元素，可以考虑使用 HashMap 替换 ArrayList

**空间：** 但是 HashMap 数据结构存储利用率相比 ArrayList 会低很多

**工具：** 使用 MAT 观察数据结构内存占用

极客时间

## A43

LinkedList 适合什么场景呢？

- 在各种常用场景下，LinkedList 几乎都不能在性能上胜出 ArrayList
- 使用任何数据结构最好根据自己的使用场景来评估和测试，以数据说话

## 空值处理

### A44

注意 5 种可能出现空指针的情况（Arthas）

#### 情况 1

极客时间

**原因：**参数值是 Integer 等包装类型，使用时因为自动拆箱出现了空指针异常

**解决：**对于 Integer 的判空，可以使用 Optional.ofNullable 来构造一个 Optional<Integer>，然后使用 orElse(0) 把 null 替换为默认值再进行操作

#### 情况 2

**原因：**字符串比较出现空指针异常

**解决：**对于 String 和字面量的比较，可以把字面量放在前面，比如 "OK".equals(s)

#### 情况 3

**原因：**诸如 ConcurrentHashMap 这样的容器不支持 Key 和 Value 为 null，强行 put null 的 Key 或 Value 会出现空指针异常

**解决：**不要存 null

#### 情况 4

**原因：**A 对象包含了 B，在通过 A 对象的字段获得 B 之后，没有对字段判空就级联调用 B 的方法会出现空指针异常

**解决：**使用 Optional.ofNullable 配合 map 和 ifPresent 方法

极客时间

## 情况 5

**原因：**方法或远程服务返回的 List 不是空而是 null，没有进行判空就直接调用 List 的方法，出现空指针异常

**解决：**使用 Optional.ofNullable 包装一下返回值，然后通过 .orElse(Collections.emptyList())，实现在 List 为 null 的时候获得一个空的 List

极客时间

## 工具

使用 Arthas 的 watch 命令观察方法入参，定位 null 参数，使用 stack 命令观察调用栈定位调用路径

极客时间

极客时间

## A45

POJO 字段设置默认值导致数据库中的原始值被覆盖的坑

极客时间

**原因：**POJO 中的字段有默认值，如果客户端不传值，就会赋值为默认值，导致每次都把默认值更新到了数据库中

**解决：**POJO 字段不要设置默认值，如果怕没值可以让数据库设置默认值

## A46

你见过数据库中出现 null 字符串的问题吗？

**原因：**字符串格式化时，可能会把 null 值格式化为 null 字符串

**解决：**使用 Optional 的 orElse 方法一键把空转换为空字符串



## A47

客户端不传值以及传 null 在 POJO 中都是 null，如何区分？

**原因：**对于 JSON 到 DTO 的反序列化过程，null 的表达是有歧义的，客户端不传某个属性，或者传 null，这个属性在 DTO 中都是 null

极客时间

**解决：**使用 Optional 来包装，以区分客户端不传数据还是故意传 null

## A48

MySQL 中涉及 NULL 的 3 个容易忽略的坑

### 坑 1

**原因：**MySQL 中 sum 函数没统计到任何记录时，会返回 null 而不是 0

**解决：**可以使用 IFNULL 函数把 NULL 转换为 0

### 坑 2

**原因：**MySQL 中 count 字段不统计 NULL 值

**解决：**可以使用 IFNULL 函数把 NULL 转换为 0

### 坑 3

**原因：**MySQL 中使用诸如 =、<、> 这样的算数比较操作符比较 NULL 的结果总是 NULL，这种比较就显得没有任何意义

**解决：**对 NULL 进行判断只能使用 IS NULL、IS NOT NULL 或者 ISNULL() 函数

## 异常处理

### A49

仅在框架层面粗犷捕获和处理异常，合适吗？

- 不建议在框架层面进行异常的自动、统一处理，三层架构每一层对异常的处理原则都不同
- 不过框架可以做兜底工作，如果异常上升到最上层逻辑还是无法处理的话，可以用统一的方式进行异常转换

极客时间

极客时间

### A50

3 种错误的异常处理方式

3 种错误方式：

- 捕获了异常后直接生吞
- 丢弃异常的原始信息
- 抛出异常时不指定任何消息

正确：

处理异常应该杜绝生吞，并确保异常栈信息得到保留。如果需要重新抛出异常的话，请使用具有意义的异常类型和异常消息

极客时间

## A51

在 finally 中抛出异常，会怎么样？

**问题：**try 中的逻辑出现了异常，但可能被 finally 中的异常覆盖

**解决：**finally 代码块自己负责异常捕获和处理，或者抛出 try 中异常（主异常）的同时，使用 addSuppressed 把 finally 中的异常关联到主异常

**扩展：**使用 try...finally 释放资源同样会有这个问题，可以使用 try-with-resources 模式

极客时间

极客时间

## A52

异常是否可以定义为静态变量呢？

**答案：**不能，异常定义为静态变量会导致异常信息固化

**解决：**需要每次 new 一个异常出来

极客时间

### A53

线程池通过 `execute` 提交任务，出现异常会怎么样？

**答案：**线程退出，向标准错误输出打印了出现异常的线程名称和异常信息

**解决：**以 `execute` 方法提交到线程池的异步任务，最好在任务内部做好异常处理，并设置自定义的异常处理程序作为保底

### A54

线程池通过 `submit` 提交任务，出现异常会怎么样？


**答案：**线程不退出，但是异常被生吞

**解决：**通过 `submit` 提交的任务，保存 `Future`，通过 `get` 方法获取返回内容以及可能出现的异常

## 日志

### A55

你能区分 SLF4J 的桥接类库和绑定类库吗？

**桥接：**把其他日志框架的 API 记录的日志桥接到 SLF4J  极客时间

**绑定：**使其他日志框架可以兼容 SLF4J 标准

**两者结合：**统一应用中的日志框架

 极客时间

 极客时间

### A56

配置不当导致日志重复记录的两个坑

#### 坑 1

 极客时间

**原因：**Appender 同时挂载到了两个 Logger 上，一个是 `<logger>`，一个是 `<root>`。由于我们定义的 `<logger>` 继承自 `<root>`，所以同一条日志既会通过 logger 记录，也会发送到 root 记录

**解决：**如果只是希望修改日志级别，不用单独设置 `appender-ref`；如果希望输出到其他 appender，需要设置 `additivity` 属性为 `false`

#### 坑 2

**原因：**LevelFilter 仅仅配置 `level` 是无法真正起作用的

**解决：**配置 `onMatch` 和 `onMismatch` 属性

**A57****异步日志可能撑爆内存的问题**

**原因：**设置了太大的 queueSize，日志量大的时候会 OOM

**解决：**设置合适的 queueSize，设置合适的 discardingThreshold，宁愿丢一些日志

 极客时间**A58****异步日志可能出现日志丢失的问题**

**原因：**queueSize 设置得比较小（默认值就非常小），且 discardingThreshold 设置为大于 0 的值（或者为默认值），队列剩余容量少于 discardingThreshold 的配置就会丢弃  $\leq$  INFO 的日志

 极客时间

**解决：**设置 discardingThreshold 为 0，即使是  $\leq$  INFO 的级别日志也不会丢，但最好把 queueSize 设置大一点

**A59****异步日志还是会阻塞方法的问题**

**原因：**neverBlock 默认为 false，意味着总可能会出现阻塞。如果 discardingThreshold 为 0，那么队列满时再有日志写入就会阻塞；如果 discardingThreshold 不为 0，也只会丢弃  $\leq$  INFO 级别的日志，那么出现大量错误日志时，还是会阻塞程序

**解决：**设置 neverBlock 为 true，永不阻塞，但可能会丢数据。如果希望兼顾性能和丢数据问题，可以丢弃不重要的日志，把 queueSize 设置大一点，再设置一个合理的 discardingThreshold

极客时间

极客时间

**A60****使用 {} 占位符语法记录日志的意义，在于什么？**

**原因：**使用 {} 占位符语法，不能通过延迟参数值获取来解决日志数据获取的性能问题，只是减少了日志参数对象 toString() 和字符串拼接的耗时

**解决：**事先判断日志级别，或通过 Lambda 表达式进行延迟参数内容获取

极客时间

## 文件 IO

### A61

为什么读写文件会遇到乱码问题？

**原因：**FileReader 是以当前机器的默认字符集来读取文件的

**解决：**如果希望指定字符集的话，需要直接使用  
InputStreamReader 和 FileInputStream

### A62

使用 Files.lines 等返回 Stream 方法进行 IO 操作的大坑 (IsOf)

**原因：**使用 Files 类的一些流式处理操作，不会自动关闭底层句柄

**解决：**使用 try-with-resources 方式来配合，确保流的 close 方法可以调用释放资源

**工具：**使用 IsOf 观察进程打开的文件数

### A63

设置缓冲区，对读写文件的性能影响有多大？

- 每次读写一个字节的方式完全不可接受，在进行文件 IO 处理的时候，使用合适的缓冲区可以明显提高性能
- 即使读写的时候使用字节数组作为缓冲区，也建议使用 BufferedInputStream 和 BufferedOutputStream
- 如果希望有更高的性能，可以使用 FileChannel



## 序列化

### A64

使用 RedisTemplate 保存数据，Redis 中出现乱码问题

极客时间

**原因：**默认情况下 RedisTemplate 针对 Key 和 Value 使用了 JDK 序列化，所谓的一串乱码其实就是字符串 redisTemplate 经过 JDK 序列化后的结果

**解决：**使用 StringRedisTemplate，或者设置 Key 的序列化方式为字符串 `setKeySerializer(RedisSerializer.string());`

极客时间

### A65

使用 RedisTemplate 出现 ClassCastException 的问题

极客时间

**原因：**如果自定义 RedisTemplate 序列化 Value 的时候没有把类型信息一起序列化，那么获取到的 Value 不是 User 类型的，而是 LinkedHashMap 类型的。强制类型转换，可能出现 ClassCastException

**解决：**

- 可以启用 Jackson ObjectMapper 的 `activateDefaultTyping` 方法
- 更简单的方式是，直接使用 `RedisSerializer.json()` 快捷方法来获取序列化器

## A66

### 枚举作为 API 参数或返回值的两个大坑

#### 坑 1

**原因：**服务端定义的枚举客户端没有，客户端反序列化的时候出现异常

极客时间

**解决：**对于 Jackson，开启 `read_unknown_enum_values_using_default_value=true`，并且通过 `@JsonEnumDefaultValue` 注解来设置默认枚举项

#### 坑 2

极客时间

**原因：**Jackson 的最新版本 2.10 至今都存在 Bug，在 `int` 类型枚举字段上标记 `@JsonValue` 只能用于序列化，对反序列化无效

**解决：**

- 设置 `@JsonCreator` 来强制反序列化时使用自定义的工厂方法
- 或者，自定义一个 `JsonDeserializer`

极客时间

## A67

### 注意 Jackson 反序列化对额外字段的处理

**原因：**默认情况下，反序列化的时候遇到多出来的字段会抛出 `UnrecognizedPropertyException`

**解决：**配置 `ObjectMapper`，全局关闭 `DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES` 特性，或者是为类型加上 `@JsonIgnoreProperties` 注解，开启 `ignoreUnknown` 属性

## A68

### 自定义 ObjectMapper 可能会覆盖 Spring Bean 的问题

**原因：**如果在项目中重新使用 @Bean 自定义 ObjectMapper，注意它可能会覆盖 Spring Boot 默认的 ObjectMapper 及相关配置

**解决：**

- 不要自定义 ObjectMapper，而是直接在配置文件设置相关参数，来修改 Spring 默认的 ObjectMapper 的功能，比如 `spring.jackson.serialization.write_enums_using_index`
- 自定义 `Jackson2ObjectMapperBuilderCustomizer` 来实现额外的 ObjectMapper 特性配置

## A69

### 注意反序列化可能不会调用自定义的构造方法的坑

**原因：**默认情况下，在反序列化的时候，Jackson 框架只会调用无参构造方法创建对象

**解决：**可以使用 `@JsonCreator` 注解配合 `@JsonProperty` 注解，实现走自定义的构造方法

## 日期时间

### A70

#### 初始化日期时间的各种坑

**原因：**Date 设置的年应该是和 1900 的差值，使用 Calendar 设置月份是从 0 开始的

**解决：**使用 Calendar 设置月份使用枚举，更推荐使用 Java 8 的日期时间 API，清晰明了

### A71

#### SimpleDateFormat 的 3 个坑

##### 坑 1

**原因：**YYYY 是 week year，可能出现提前跨年问题

**解决：**使用 yyyy

##### 坑 2

**原因：**类非线程安全

**解决：**使用 ThreadLocal 包装

##### 坑 3

**原因：**解析字符串比较宽容。比如，我们期望使用 yyyyMM 来解析 20160901 字符串，会把 0901 当做月份处理

**解决：**自行进行数据合法性判断

##### 更推荐

使用 Java 8 的 DateTimeFormatterBuilder 来初始化 DateTimeFormatter，没有这 3 个问题

## A72

### 时区问题导致日期时间错乱的诸多坑

#### 原因：

- 一个字符串只能是一个时间表示，无法对应某个时间点
- 时区设置不正确

极客时间

#### 解决：

- Date 这种 UTC 时间戳才能作为时间点
- 如果要把一个时间表示解析为时间戳，需要设置合适的时区
- 把 UTC 时间戳格式化为时间表示的时候，也需要确保时区正确
- 更推荐使用 Java 8 的 ZonedDateTime 保存时间，然后使用设置了 ZoneId 的 DateTimeFormatter 配合 ZonedDateTime 进行时间格式化，来得到本地时间表示

极客时间

极客时间

极客时间

## A73

### 使用时间戳进行日期时间计算的坑

原因：手动计算时间戳来加减日期出现 int 溢出问题

#### 解决：

- 使用 Calendar 类的 add 方法
- 更推荐使用 Java 8 的日期时间类型：
  - 可以使用各种 minus 和 plus 方法直接对日期进行加减操作
  - 还可以通过 with 方法进行快捷时间调节
  - 可以直接使用 Lambda 表达式进行自定义的时间调整

## Java 8 的日期时间类型


**LocalDate**: 表示日期, 无时区属性

**LocalTime**: 表示时间, 无时区属性

**LocalDateTime**: LocalDate+LocalTime, 无时区属性 


**Instant**: 时间戳, 类似 Date

**ZonedDateTime**: 表示日期和实践, 有时区属性, 类似  
GregorianCalendar (Calendar 的一种)

**Duration**: 一段时间 

**Period**: 一段日期, 基于日历系统的日期区间

**DateTimeFormatter**: 可以格式化日期时间, 线程安全

**ZoneId/ZoneOffset**: 时区 

# OOM

## A75

### 太多份相同的对象导致的 OOM 坑 (MAT)

**原因：**不管是程序的实现不合理，还是因为各种框架对数据的重复处理、加工和转换，相同的数据在内存中不一定只占用一份空间

**解决：**针对内存量使用超大的业务逻辑，比如缓存逻辑、文件上传下载和导出逻辑，我们在做容量评估时，可能还需要实际做一下 Dump，而不是简单地假设数据只有一份

## A76

### 使用 WeakHashMap 居然也会出现 OOM？

**原因：**WeakHashMap 的 Key 虽然是弱引用，但是其 Value 持有 Key 中对象的强引用会导致 Key 无法回收，无限往 WeakHashMap 加入数据同样会 OOM

**解决：**使用 Spring 提供的 ConcurrentReferenceHashMap，或使用 WeakReference 来包装 Value

## Tomcat 参数不合理导致的 OOM 血案 (MAT)

**原因：**设置不合理的 `server.max-http-header-size=10M`，每个请求需要占用 20M 内存，并发大的时候导致 OOM

**解决：**要根据实际需求来修改参数配置，可以考虑预留 2 到 5 倍的量。容量类的参数背后往往代表了资源，设置超大的参数就有可能占用不必要的资源，在并发量大的时候因为资源大量分配导致 OOM



## Java 高级特性

### A78

通过反射调用方法，是传参决定方法重载吗？

**原因：**反射调用方法，是以反射获取方法时传入的方法名称和参数类型而不是调用方法时的参数类型，来确定调用方法

**解决：**

`getDeclaredMethod("age", Integer.TYPE)` 对应 `age(int age)`

`getDeclaredMethod("age", Integer.class)` 对应 `age(Integer age)`

极客时间

### A79

泛型经过类型擦除覆写失败的坑

极客时间

**原因：**

- 子类没有指定 `String` 泛型参数，父类的泛型方法 `setValue(T value)` 在泛型擦除后是 `setValue(Object value)`，子类中入参是 `String` 的 `setValue` 方法被当作了新方法
- 子类的 `setValue` 方法没有增加 `@Override` 注解，因此编译器没能检测到重写失败的问题

**解决：**方法重写一定要标记 `@Override` 注解

**A80**

泛型经过类型擦除会多出桥接方法的坑  
(javap、jclasslib)

**原因：**泛型类型擦除后会生成桥接方法，使用反射按照方法名称查询方法列表会得到重复方法

极客时间

**解决：**按照方法签名查询单一方法，或者查询方法列表的时候用 `!method.isBridge()` 过滤桥接方法

**工具：**使用 `jclasslib` 和 `javap` 查看和反编译字节码

极客时间

极客时间

**A81**

注解即使加上了 `@Inherited`，也无法从接口和方法继承的问题

极客时间

**原因：**`@Inherited` 只能实现类上的注解继承，无法实现方法上注解的继承

**解决：**使用 Spring 的 `AnnotatedElementUtils.findMergedAnnotation`

# Spring 框架

## A82

### 注意 Bean 默认单例的问题

**原因：**父类有状态，子类标记了 `@Service`，成为了单例，造成 OOM

**解决：**在为类标记上 `@Service` 注解把类型交由容器管理前，首先评估一下类是否有状态，然后为 Bean 设置合适的 Scope

极客时间

## A83

### 单例的 Bean 注入多例的 Bean，不仅仅是设置 Scope 这么简单

**原因：**Bean 默认是单例的，所以单例的 Controller 注入的 Service 也是一次性创建的，即使 Service 本身标识了 prototype 的范围也没用

**解决：**设置 `proxyMode = ScopedProxyMode.TARGET_CLASS` 走代理方式

极客时间

极客时间

## A84

### 自定义 Aspect 因为顺序问题导致 Spring 事务失效的坑

**原因：**自定义的切面先捕获了异常，使得 Spring 声明式事务无法回滚

**解决：**使用 `@Order` 为自定义的切面设置优先级

## A85

### Feign AOP 切不到的诡异案例

#### 案例 1

**原因：**虽然 LoadBalancerFeignClient 和 ApacheHttpClient 都是 feign.Client 接口的实现，但是 HttpClientFeignLoadBalancedConfiguration 的自动配置只是把前者定义为 Bean，后者是 new 出来的，不是 Bean 无法 AOP

**解决：**去掉 Ribbon 模块依赖，让 ApacheHttpClient 直接成为一个 Bean

#### 案例 2

**原因：**Spring Boot 2.x 默认使用 CGLIB 的方式，但通过继承实现代理有个问题是，无法继承 final 的类，而因为 ApacheHttpClient 类就是定义为了 final，无法直接切入

**解决：**把配置参数 proxy-target-class 的值修改为 false，以切换到使用 JDK 动态代理的方式

## A86

### 配置文件中的配置不生效的问题

**原因：**PropertySource 配置优先级：系统属性 -> 环境变量 -> 配置文件

**解决：**注意优先级问题，在配置文件自定义配置的时候避免因为相同的 Key 而出现配置冲突

**扩展：**SpringBoot 2.0 因为需要实现 Relaxed Binding 2.0，通过自定义 ConfigurationPropertySourcesPropertySource 并且把它作为配置源的第一个，实现了对 PropertySourcesPropertyResolver 中遍历逻辑的“劫持”



R

设计篇

极客时间

极客时间

极客时间

极客时间

## 代码重复

### B01

利用工厂模式 + 模板方法模式，消除 if...else 和大量重复代码的例子

极客时间

**原因：**有多个并行的类实现相似逻辑时造成的代码重复

**解决：**

- 考虑提取相同逻辑在父类中实现，差异逻辑通过抽象方法留给子类实现。使用类似的模板方法把相同的流程和逻辑固定成模板，保留差异的同时尽可能避免代码重复
- 同时，可以使用 Spring 的 IoC 特性注入相应的子类，来避免实例化子类时的大量 if...else 代码

极客时间

极客时间

### B02

利用注解 + 反射消除重复代码的例子

**原因：**使用硬编码的方式重复实现相同的数据处理算法造成的代码重复

**解决：**考虑把规则转换为自定义注解，作为元数据对类或对字段、方法进行描述，然后通过反射动态读取这些元数据、字段或调用方法，实现规则参数和规则定义的分离，把变化的部分也就是规则的参数放入注解，规则的定义统一处理

## B03

### 利用属性拷贝工具消除重复代码的例子

**原因：**业务代码中常见的 DO、DTO、VO 转换时大量字段的手动赋值，这种转换可能出现多次，字段多的时候非常容易出错

**解决：**

- 不要手动进行赋值，考虑使用 Bean 映射工具进行
- 还可以考虑采用单元测试，对所有字段进行赋值正确性校验

极客时间

极客时间

极客时间

极客时间

## 接口设计

### B04

接口的响应要明确表示接口的处理结果

原因：

- 接口透传了下游服务的错误码
- 接口响应体中的字段意义不明确

解决：

- 服务应该转换和收敛下游服务的错误码
- 可以考虑以“接口设计”来设计响应体，在 HTTP 状态码是 200 的时候去解析接口响应，在响应中 success 为 true 的时候去解析 data 获取业务数据，最后通过查看 data 中的业务状态来真正获得业务处理结果



## B05

### 要考虑接口变迁的版本控制策略

**原因：**接口不可能一成不变，涉及到参数定义的变化或是参数废弃，导致接口无法向前兼容，这个时候接口可能就需要有版本的概念

**解决：**

- 版本策略最好一开始就考虑
- 版本实现方式要统一，比如可以自定义

RequestMappingHandlerMapping 通过注解实现统一的、基于 URL 的版本控制策略

极客时间

极客时间

极客时间

## B06

### 接口同步 / 异步处理方式要明确

**原因：**如果接口表面是同步返回结果的同步接口，内部却是异步处理，那么接口的处理逻辑很可能不可预测，最好要么同步要么异步


**解决：**对于同步接口，由调用方控制超时；对于异步逻辑，则返回调用方任务 ID，让调用方根据任务 ID 查询结果

极客时间

# 缓存设计

## B07


把 Redis 当成数据库，会有什么问题？

**问题：**无法超出内存限制保存数据，没有数据库那么可靠 

**解决：**客户端要处理从缓存获取不到数据的情况，服务端配置合理的 maxmemory + 淘汰策略

## B08

小心缓存雪崩问题

**原因：**缓存 Key 同时大规模失效需要回源，导致数据库压力激增问题 

**解决：**差异化缓存过期时间，或者初始化缓存数据的时候设置缓存永不过期，然后启动一个后台线程定时把所有数据更新到缓存

## B09

小心缓存击穿问题

**原因：**在某些 Key 属于极端热点数据，且并发量很大的情况下，如果这个 Key 过期，可能会在某个瞬间出现大量的并发请求同时回源

**解决：**使用进程内的锁进行限制，这样每一个节点都可以以一个并发回源数据库；或者不使用锁进行限制，而是使用类似 Semaphore 的工具限制并发数

## B10

## 小心缓存穿透问题

**原因：**如果原始数据不存在，导致缓存中的数据不存在，客户端每次访问都会到数据库回源

**解决：**对于不存在的数据，同样设置一个特殊的 Value 到缓存中；或者使用布隆过滤器做前置过滤

## B11

## 应该先更新缓存还是先更新数据库？

数据库和缓存同步更新不太合理。更好的方案是，先更新数据库再删除缓存（如果删除缓存失败，则考虑进行一定的重试），访问的时候按需加载数据到缓存

## 生产就绪

### B12

如何写一个正确的健康检测接口？

- 启用 Spring Boot Actuator，并且实现自定义的 HealthIndicator 来触达关键内部组件
- 也可以实现 CompositeHealthContributor 聚合多个 HealthIndicator

极客时间

### B13

如何对外暴露应用内部重要组件的状态？

实现自定义的 InfoContributor

极客时间

极客时间

### B14

为什么指标 Metrics 对于问题定位这么重要？

- 通过收集指标然后在图形工具展现为曲线图、饼图等图表，可以帮助我们快速定位分析问题，机器擅长从一堆数据（比如原始日志）中分析出问题和规律，而人更擅长看图找规律
- 可以定义总业务指标来反映业务监控状态，以判断是否有问题
- 可以定义各种应用指标，监控上游、内部和下游的处理能力和处理性能，来分析哪个环节可能出问题
- 可以收集 JVM 以及应用关键组件的指标，来定位哪个组件可能出现问题
- 实现：Micrometer + InfluxDB / Prometheus

极客时间

## 异步处理

### B15

为什么异步处理中消息补偿这么重要？

**原因：**消息发送、传输、处理等环节都可能发生消息丢失，而且任何 MQ 中间件都无法确保 100% 可靠

**解决：**必须考虑补偿或者说建立主备双活流程，最好备线达到主线的处理能力，也就是 MQ 不工作的时候，备线通过补偿实现业务正常运作

极客时间

极客时间

### B16

为什么异步处理需要实现操作幂等？

极客时间

**原因：**MQ 重复发消息、自动补偿重复处理、人工后台补偿重复处理

**解决：**判断业务操作是否已处理过

**B17****消息模式配置错误导致消息重复或遗漏的坑**

**原因：**应当配置为队列的消息配置为了广播，导致消息重复处理；  
应当配置为广播的消息配置为了队列，导致消息漏处理

**解决：**不同的 MQ 配置消息模式的方式都不同，RabbitMQ 使用不同的交换器，RocketMQ 使用 ConsumerGroup 的概念

**B18****消息队列被死信堵塞的坑**

**原因：**始终无法处理的死信消息可能会让 MQ 无限堆积消息，  
最终出现 OOM，也影响正常消息处理

**解决：**指数退避重试，最后还是处理失败则加入死信队列进行最后处理

## 数据存储

### B19

与 MySQL 相比, Redis 更擅长什么不擅长什么?

擅长: 针对单一 Key 的操作

极客时间

不擅长: 搜索 Key, 使用 keys 命令搜索 Key 可能会阻塞整个数据库

### B20

与 MySQL 相比, Elasticsearch 更擅长什么不擅长什么?

擅长: 全文搜索

极客时间

不擅长: 频繁数据更新

### B21

与 MySQL 相比, InfluxDB 更擅长什么不擅长什么?

擅长: 时间序列数据聚合

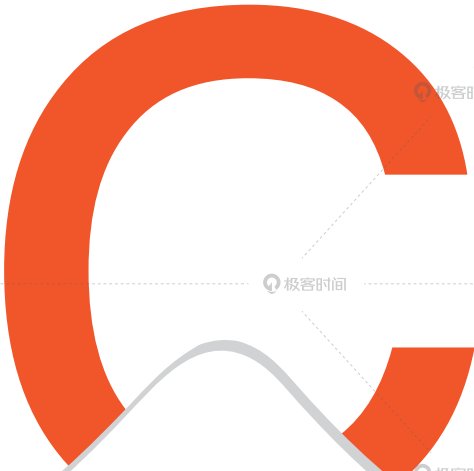
不擅长: 保存明细数据, 保存大量具有不同值的 tag, 可能会 OOM

## B22

### 结合 NoSQL 和 RDBMS 可以应对大并发的存储方案

- 主数据由两种 MySQL 数据表构成，索引表承担简单条件的搜索以得到主键，Sharding 表承担大并发的主键查询。主数据由同步写服务写入，写入后发出 MQ 消息
- 辅助数据可以根据自己的需要选用合适的 NoSQL，由单独的一个或多个异步写服务监听 MQ 后异步写入
- 由统一的查询服务，对接所有查询需求，根据不同的查询需求路由查询到合适的存储，确保每一个存储系统可以根据场景发挥所长，同时按照场景分散各数据库系统的查询压力





# 安全篇

极客时间

极客时间

极客时间

极客时间

## 数据源头

### C01

#### 客户端的计算不可信

**原因：**客户端进行计算可以改善交互体验，但因为数据可以篡改，只能用于客户端呈现

**解决：**客户端只能提供原始数据（比如商品 ID 和购买份数），计算数据需要通过服务端进行（比如订单价格）

极客时间

极客时间

### C02

#### 客户端提交的参数需要校验

**原因：**即使客户端的数据原本来自服务端，客户端也可以随意修改数据

**解决：**服务端需要判断参数合法性

极客时间

### C03

#### 不能信任请求头里的任何内容

**原因：**请求头里的任何数据都可以随便模拟修改

**解决：**用户进行登录或三方授权登录（比如微信），拿到用户标识来做唯一性判断

极客时间

### C04

#### 用户标识不能从客户端获取

**原因：**可以随意篡改用户标识，有的时候不是不知道这点，而是遗留了用于调试的参数

**解决：**用户标识只能在服务端从登录后用户的会话中获取

极客时间

## 安全兜底

### C05

#### 如何防止平台资源被刷？

**原因：**代码本身涉及有偿使用的三方服务。如果因为代码本身缺乏授权、用量控制而被利用导致大量调用，势必会消耗大量的钱，给公司造成损失

**解决：**

- 对请求合法性进行校验（比如校验请求头，校验请求是否经过前置流程）
- 进行总量和频次控制
- 通过触发验证码等人机识别方案，来过滤不合理的使用

### C06

#### 如何防止虚拟资产无限发放？

**原因：**代码涉及虚拟资产的发放，比如积分、优惠券等。虽然说虚拟资产不直接对应货币，但一般可以在平台兑换具有真实价值的资产。从某种意义上说，虚拟资产就是具有一定价值的钱，但因为不直接涉及钱和外部资金通道，所以容易产生随意性发放导致漏洞

**解决：**

- 把优惠券等虚拟资产考虑为一种资源，其生产不是凭空的，而需要事先申请
- 申请需要有流程，每一次申请作为一个批次，有数量控制
- 批次中的每一个优惠券都有 ID 可跟踪、可注销

## 如何防止重复的资金？

**原因：**代码涉及真实钱的进出。比如，对用户扣款，如果出现非正常的多次重复扣款，小则用户投诉、用户流失，大则被相关管理机构要求停业整改，影响业务

极客时间

**解决：**任何资金操作都需要在平台侧生成业务属性的订单，可以是优惠券发放订单，可以是返现订单，可以是借款订单，一定是先有订单再去做资金操作；一定要做好防重，也就是实现幂等处理，并且幂等处理必须是全链路的

极客时间

极客时间

极客时间

## 数据和代码

### C08

#### SQL 注入如何实现拖库？（sqlmap）

**原因：**数据查询参数被当做代码拼接到了 SQL，黑客构造不同的代码实现注入

**误区：**

- 认为 SQL 注入问题只可能发生于 Http Get 请求，也就是通过 URL 传入的参数才可能产生注入点

- 认为不返回数据的接口，不可能存在注入问题

- 认为 SQL 注入的影响范围，只是通过短路实现突破登录，只需要登录操作加强预防即可

**解决：**使用参数化查询，对于 MyBatis 避免使用 \${} 占位符

**工具：**演示使用 sqlmap 进行错误注入、布尔盲注和时间盲注进行拖库

### C09

#### 小心代码注入问题

**原因：**通过 ScriptEngine 动态执行脚本，把参数拼接到脚本代码中，参数被当作了代码执行

**解决：**

- 通过 SimpleBindings 来绑定参数

- 使用 SecurityManager 配合 AccessControlContext，来构建一个脚本运行的沙箱环境

## C10

## 全方位堵漏 XSS 的四招

**原因：**原本应该是让用户输入正常数据的地方被黑客替换为了 JavaScript 脚本，然后页面在显示这个数据或把数据存入数据库的时候没有对 HTML 进行转义，最后 JavaScript 被浏览器当做了代码执行


**解决：**

- 实现 Filter 来转义请求参数
- 实现 JsonSerializer 和 JsonSerializer 来转义 JSON 参数
- 确保模板引擎转义后在页面呈现，比如对于 thymeleaf 使用 `th:text`
- 为 Cookie 开启 HttpOnly 属性，来防止脚本读取 Cookie

## 敏感数据

### C11

使用 MD5 摘要保存密码是否安全？

不安全，使用彩虹表可以轻易“解密”10 位以内的摘要 

### C12


MD5 摘要加盐的错误方式

 极客时间

 极客时间

盐的作用是防止彩虹表破解，本身并不需要加密保存

**错误：**固定盐、太短的盐、使用用户数据的一部分作为盐

**正确：**全球唯一的、具有一定长度的、随机的盐 

### C13

了解 BCryptPasswordEncoder

- 可以设置不同的代价因子来控制哈希操作的速度，代价因子为 10 的时候比 MD5 慢 100 倍
- 把盐作为算法的一部分，强制我们遵从安全保存密码的最佳实践



## C14

### 注意使用安全的对称加密算法

- DES 不够安全，在 1999 年的 DES 挑战赛 3 中，DES 密码破解耗时不到一天，3DES 又太慢
- AES 算法有各种模式，其中 ECB 模式最简单，但是不安全，可以通过密文来操纵明文
- 比较推荐的是 CTR/GCM 模式

极客时间

极客时间

## C15

### 使用 AES-256-GCM 和单独的加密服务来加密敏感数据的例子

- 密文、脱敏数据和用户数据保存在一起
- 独立的加密服务做加密操作，分离保存密钥、初始化向量
- 可以从外部由用户提供 AAD（附加认证数据）进一步确保安全

极客时间

极客时间

极客时间

极客时间

极客时间

极客时间

极客时间

极客时间

极客时间

极客时间

极客时间

极客时间

极客时间

极客时间

极客时间

极客时间

极客时间

极客时间

极客时间

极客时间

极客时间

 极客时间

 极客时间

 极客时间

 极客时间

## Java 高手笔记

内容来源：《Java 业务开发常见错误 100 例》 作者：朱晔

\* 极客时间原创出品，版权所有，侵权必究

极客时间官网：<https://time.geekbang.org/>

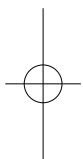
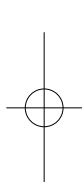
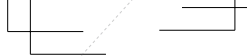
微博 / 公众号：@ 极客时间

视觉：Daisy

编者：极客时间教研部



扫码了解更多课程



极客时间

极客时间

极客时间

极客时间

