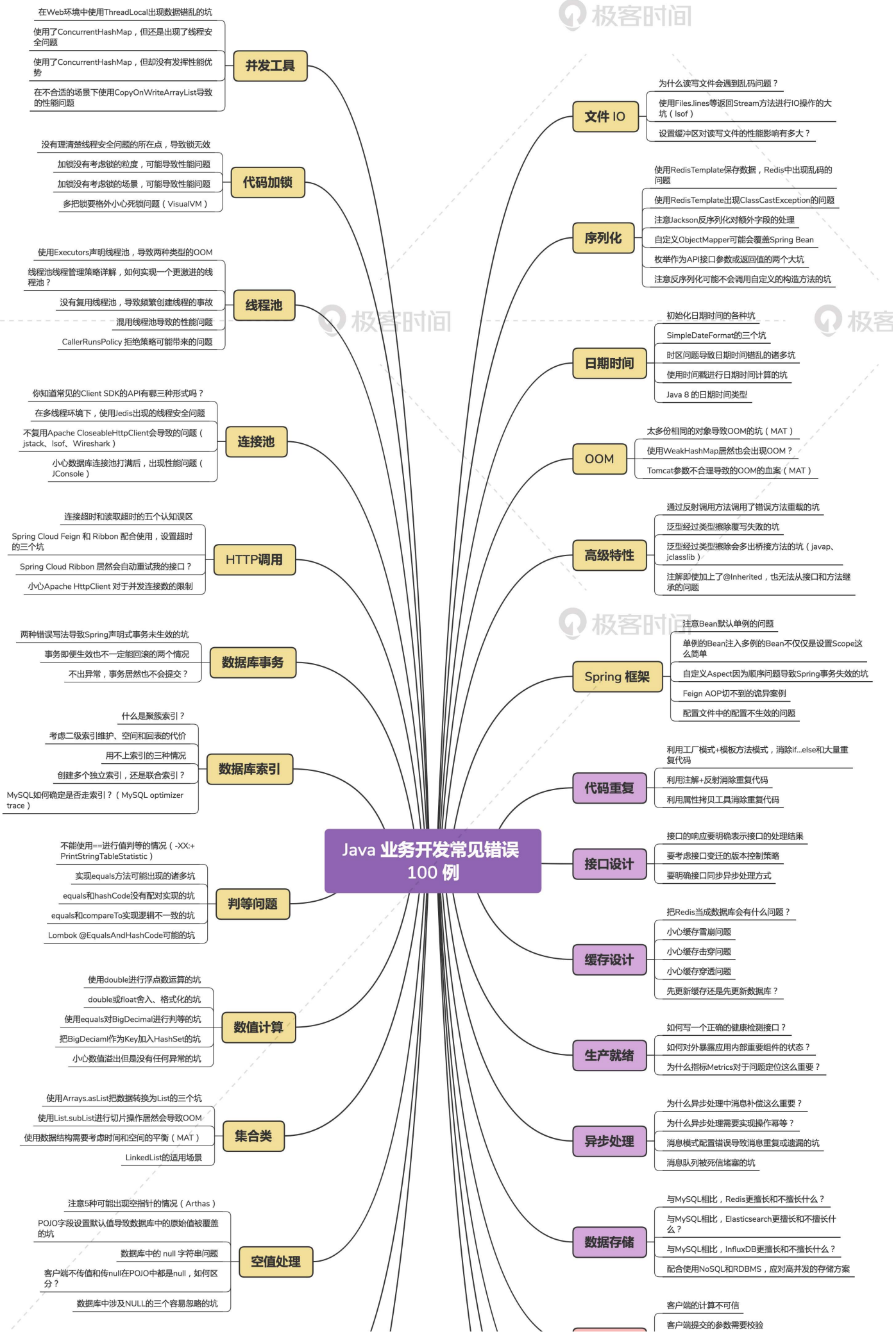
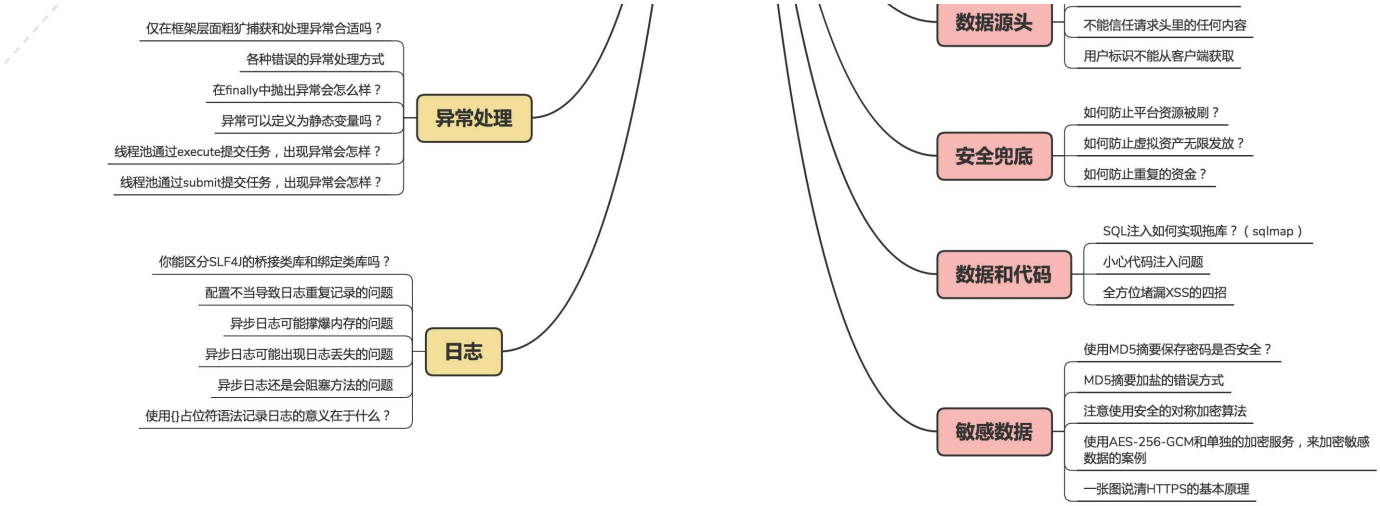
# Java业务开发常见错误100例-2

由 admin创建, 最后修改于大约1分钟以前

Java高手笔记之业务开发常见错误100例 - 简书 (jianshu.com)

Java高手笔记之业务开发常见错误100例_java 业务开发常见错误100例_Apple_Web的博客-CSDN博客


Java 业务开发常见错误 100 例
并发工具
在Web环境中使用ThreadLocal出现数据错乱的坑
使用了ConcurrentHashMap，但还是出现了线程安全问题
使用了ConcurrentHashMap，但却没有发挥性能优势
在不合适的场景下使用CopyOnWriteArrayList导致的性能问题
代码加锁
没有理清楚线程安全问题的所在点，导致锁无效
加锁没有考虑锁的粒度，可能导致性能问题
加锁没有考虑锁的场景，可能导致性能问题
多把锁要格外小心死锁问题（VisualVM）
线程池
使用Executors声明线程池，导致两种类型的OOM
线程池线程管理策略详解，如何实现一个更激进的线程池？
没有复用线程池，导致频繁创建线程的事故
混用线程池导致的性能问题
CallerRunsPolicy 拒绝策略可能带来的问题
连接池
你知道常见的Client SDK的API有哪三种形式吗？
在多线程环境下，使用Jedis出现的线程安全问题
不复用Apache CloseableHttpClient会导致的问题（jstack、lsof、Wireshark）
小心数据库连接池打满后，出现性能问题（JConsole）
HTTP调用
连接超时和读取超时的五个认知误区
Spring Cloud Feign 和 Ribbon 配合使用，设置超时的三个坑
Spring Cloud Ribbon 居然会自动重试我的接口？
小心Apache HttpClient 对于并发连接数的限制
数据库事务
两种错误写法导致Spring声明式事务未生效的坑
事务即便生效也不一定能回滚的两个情况
不出异常，事务居然也不会提交？
数据库索引
什么是聚簇索引？
考虑二级索引维护、空间和回表的代价
用不上索引的三种情况
创建多个独立索引，还是联合索引？
MySQL如何确定是否走索引？（MySQL optimizer trace）
判等问题
不能使用==进行值判等的情况（-XX:+PrintStringTableStatistic）
实现equals方法可能出现的诸多坑
equals和hashCode没有配对实现的坑
equals和compareTo实现逻辑不一致的坑
Lombok @EqualsAndHashCode可能的坑
数值计算
使用double进行浮点数运算的坑
double或float舍入、格式化的坑
使用equals对BigDecimal进行判等的坑
把BigDeciaml作为Key加入HashSet的坑
小心数值溢出但是没有任何异常的坑
集合类
使用Arrays.asList把数据转换为List的三个坑
使用List.subList进行切片操作居然会导致OOM
使用数据结构需要考虑时间和空间的平衡（MAT）
LinkedList的适用场景
空值处理
注意5种可能出现空指针的情况（Arthas）
POJO字段设置默认值导致数据库中的原始值被覆盖的坑
数据库中的 null 字符串问题
客户端不传值和传null在POJO中都是null，如何区分？
数据库中涉及NULL的三个容易忽略的坑
文件 IO
为什么读写文件会遇到乱码问题？
使用Files.lines等返回Stream方法进行IO操作的大坑（lsof）
设置缓冲区对读写文件的性能影响有多大？
序列化
使用RedisTemplate保存数据，Redis中出现乱码的问题
使用RedisTemplate出现ClassCastException的问题
注意Jackson反序列化对额外字段的处理
自定义ObjectMapper可能会覆盖Spring Bean
枚举作为API接口参数或返回值的两个大坑
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
初始化日期时间的各种坑
SimpleDateFormat的三个坑
时区问题导致日期时间错乱的诸多坑
使用时间戳进行日期时间计算的坑
Java 8 的日期时间类型
OOM
太多份相同的对象导致OOM的坑（MAT）
使用WeakHashMap居然也会出现OOM？
Tomcat参数不合理导致的OOM的血案（MAT）
高级特性
通过反射调用方法调用了错误方法重载的坑
泛型经过类型擦除覆写失败的坑
泛型经过类型擦除会多出桥接方法的坑（javap、jclasslib）
注解即使加上了@Inherited，也无法从接口和方法继承的问题
Spring 框架
注意Bean默认单例的问题
单例的Bean注入多例的Bean不仅仅是设置Scope这么简单
自定义Aspect因为顺序问题导致Spring事务失效的坑
Feign AOP切不到的诡异案例
配置文件中的配置不生效的问题
代码重复
利用工厂模式+模板方法模式，消除if...else和大量重复代码
利用注解+反射消除重复代码
利用属性拷贝工具消除重复代码
接口设计
接口的响应要明确表示接口的处理结果
要考虑接口变迁的版本控制策略
要明确接口同步异步处理方式
缓存设计
把Redis当成数据库会有什么问题？
小心缓存雪崩问题
小心缓存击穿问题
小心缓存穿透问题
先更新缓存还是先更新数据库？
生产就绪
如何写一个正确的健康检测接口？
如何对外暴露应用内部重要组件的状态？
为什么指标Metrics对于问题定位这么重要？
异步处理
为什么异步处理中消息补偿这么重要？
为什么异步处理需要实现操作幂等？
消息模式配置错误导致消息重复或遗漏的坑
消息队列被死信堵塞的坑
数据存储
与MySQL相比，Redis更擅长和不擅长什么？
与MySQL相比，Elasticsearch更擅长和不擅长什么？
与MySQL相比，InfluxDB更擅长和不擅长什么？
配合使用NoSQL和RDBMS，应对高并发的存储方案
客户端的计算不可信
客户端提交的参数需要校验




Java高手笔记之业务开发常见错误100例

# 并发工具

## 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

## 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

## 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

## 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

## 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

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

### 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

### 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

### 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

### 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

### 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

### 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

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

### 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

## 你知道常见的Client SDK的API，有哪3种形式吗?

形式1:

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)

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

## 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

## 不复用Apache CloseableHttpClient会导致什么问题?(jstack、lsof、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:

1、连接池可能每次都会创建一定数量的初始连接

2、连接池可能会有一些管理模块，需要创建单独的线程来管理

工具:
1、使用jstack观察到没有复用连接池，会出现大量的Connection evictor线程

2、使用lsof观察到没有复用连接池，会出现大量TCP连接

3、如果复用CloseableHttpClient，使用wireshark 观察HTTP请求重用一个TCP连接的过程

## 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连接池的参数的情况

# HTTP调用

## 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

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

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

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

### Spring Cloud Feign和 Ribbon配合使用，设置超时的三个坑

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略微不同

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

### Spring Cloud Ribbon居然会自动重试我的接?

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=O;并且对于Get请求需要确保接幂等

### 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

### 两种错误写法导致Spring声明式事务未生效的两个坑

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

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

### 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

### 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

## 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

## 考虑二级索引的维护、空间和回表代价

原因
1、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

2、空间:额外的存储空间

3、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
1、按需创建

2、考虑联合索引

3、针对轻量级字段创建

## 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

## 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

## 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optimizertrace来了解具体原因和成本明细

# 判等问题

## 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

### 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

### 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

### 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，如果要实现Comparable，记得equals、hashCode、compareTo三者逻辑一致

### 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

### 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

解决:
1、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
2、如果一定要用 Double来初始化 BigDecimal的话,可以使用BigDecimal valueOf方法

### 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

解决:还是需要使用 BigDecima,配合 setscale进行舍入

### 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,1.00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

### 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

### 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:

1、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
2、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

### 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

### 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 :
1、不直接使用subList方法返回的SubList，而是使用newArrayList来重新构建一个普通的ArrayList
2、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

### 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

### LinkedList适合什么场景呢?

1、在各种常用场景下,LinkedList几乎都不能在性能上胜出ArrayList

2、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

### 注意5种可能出现空指针的情况 (Arthas)

情况1
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，然后使用orElse(O)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

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

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

### POJO字段设置默认值导致数据库中的原始值被覆盖的坑

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

### 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

### 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

解决：使用Optional来包装，以区分客户端不传数据还是故意传null

### 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O
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

### 仅在框架层面粗犷捕获和处理异常，合适吗?

1、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
2、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

### 3种错误的异常处理方式

3种错误方式

1、捕获了异常后直接生吞·

2、丢弃异常的原始信息
3、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

### 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

### 异常是否可以定义为静态变量呢?

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

### 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

### 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

### 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

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

### 配置不当导致日志重复记录的两个坑

坑1
原因: Appender同时挂载到了两个Logger 上,一个是，一个是。由于我们定义的继承自，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

### 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

### 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<=INFO的日志

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

### 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<=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

### 使用子占位符语法记录日志的意义，在于什么?

原因:使用子占位符语法，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

## 文件IO

### 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

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

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lsof观察进程打开的文件数

### 设置缓冲区，对读写文件的性能影响有多大?

1、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

2、即使读写的时候使用字节数组作为缓冲区，也建议使用BufferedIInputStream和 BufferedOutputStream

3、如果希望有更高的性能，可以使用FileChannel

# 序列化

## 使用RedisTemplate保存数据，Redis 中出现乱码问题

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());

## 使用RedisTemplate出现ClassCastException的问题

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:
1、可以启用Jackson ObjectMapper的activateDefaultTyping方法

2、更简单的方式是，直接使用RedisSerializer.json()快捷方法来获取序列化器

## 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

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
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：
1、设置@JsonCreator来强制反序列化时使用自定义的工厂方法

2、或者，自定义一个JsonDeserializer

## 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

## 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：
1、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

2、自定义Jackson20bjectMapperBuilderCustomizer来实现额外的ObjectMapper 特性配置

## 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

## 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O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

## 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

## 时区问题导致日期时间错乱的诸多坑

原因:
1、一个字符串只能是一个时间表示,无法对应某个时间点

2、时区设置不正确

解决：

1、Date这种UTC时间戳才能作为时间点

2、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

3、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

4、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

## 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:
1、使用Calendar类的add方法

2、更推荐使用Java 8的日期时间类型:

一可以使用各种 minus和plus方法直接对日期进行加减操作

一还可以通过with方法进行快捷时间调节

一可以直接使用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

## 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

## 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会占用不必要的资源，在并发量大的时候因为资源大量分配导致OOM

# Java高级特性

## 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(Integerage)

## 泛型经过类型擦除覆写失败的坑

原因:

1、子类没有指定String泛型参数，父类的泛型方法 setValue(Tvalue)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

2、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

## 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

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

## 注解即使加上了@Inherited，也无法从接口和方法继承的问题

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

## 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

## 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

## 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

## 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HttpClientFeignLoadBalancedConfguration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

## 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ConfgurationPropertySourcesPropertySource 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”

无标签