目录

[【自我介绍】 5](#_Toc198234028)

[【微信读书】 6](#_Toc198234029)

[美团《长期有耐心》 6](#_Toc198234030)

[《华为工作法》 6](#_Toc198234031)

[腾讯《不止于培训》 6](#_Toc198234032)

[《阿里巴巴管理三板斧》 6](#_Toc198234033)

[《小米创业思考》 6](#_Toc198234034)

[【竞赛收获】 7](#_Toc198234035)

[中国软件杯 7](#_Toc198234036)

[数学建模竞赛 7](#_Toc198234037)

[【吃了么外卖平台】 8](#_Toc198234038)

[分页查询 9](#_Toc198234039)

[为什么要分页 9](#_Toc198234040)

[MyBatis + PageHelper实现 9](#_Toc198234041)

[深度分页查询 9](#_Toc198234042)

[订单状态定时处理 10](#_Toc198234043)

[Spring Task实现定时任务 10](#_Toc198234044)

[Spring Task的问题和解决 10](#_Toc198234045)

[公共字段自动填充 11](#_Toc198234046)

[自定义注解+ AOP面向切面实现 11](#_Toc198234047)

[SSM 12](#_Toc198234048)

[MVC 12](#_Toc198234049)

[IoC 12](#_Toc198234050)

[AOP 13](#_Toc198234051)

[【12306 铁路购票系统】 14](#_Toc198234052)

[12306主要模块 15](#_Toc198234053)

[微服务框架组件 15](#_Toc198234054)

[（1）Spring Cloud Gateway 实现统一网关入口 15](#_Toc198234055)

[（2）Nacos 作为注册中心和配置中心 15](#_Toc198234056)

[（3）Sentinel实现熔断限流 16](#_Toc198234057)

[用户注册缓存穿透 17](#_Toc198234058)

[缓存三剑客 17](#_Toc198234059)

[缓存穿透解决方案 17](#_Toc198234060)

[缓存击穿解决方案——双重判定锁 17](#_Toc198234061)

[令牌限流+分布式锁余票扣减 18](#_Toc198234062)

[余票扣减核心流程 18](#_Toc198234063)

[0、查询车票 18](#_Toc198234064)

[1、接口幂等：HTTP防重复提交 19](#_Toc198234065)

[2、责任链验证提交参数 20](#_Toc198234066)

[3、分布式锁 21](#_Toc198234067)

[3、令牌限流 21](#_Toc198234068)

[Binlog监听 + 消息队列 = 数据最终一致性 23](#_Toc198234069)

[明显不可取的方案 23](#_Toc198234070)

[可根据实际情况选用的方案 23](#_Toc198234071)

[核心流程 23](#_Toc198234072)

[Canal如何抓取Binlog？ 23](#_Toc198234073)

[Binlog过多，消息堆积了怎么办？ 23](#_Toc198234074)

[延时消息取消超时未支付订单 25](#_Toc198234075)

[订单延时关闭功能技术选型 25](#_Toc198234076)

[RocketMQ延时消息实现 25](#_Toc198234077)

[消息队列幂等：防止消息重复消费 26](#_Toc198234078)

[订单数据分库分表 = 雪花算法 + 基因法 28](#_Toc198234079)

[为什么分库分表？如何分库分表？ 28](#_Toc198234080)

[订单分库分表 28](#_Toc198234081)

[雪花算法生成ID（有序、唯一、性能） 29](#_Toc198234082)

[【简历八股】 30](#_Toc198234083)

[MySQL 30](#_Toc198234084)

[一条SQL的执行流程是怎样的？ 30](#_Toc198234085)

[慢SQL优化 31](#_Toc198234086)

[事务隔离级别和MVCC的实现原理？ 32](#_Toc198234087)

[链表 红黑树 B树 B+树 33](#_Toc198234088)

[MySQL 三层 B+ 树能存多少数据？ 34](#_Toc198234089)

[行锁、表锁、意向锁 35](#_Toc198234090)

[分库分表后全局ID生成方案有哪些？ 36](#_Toc198234091)

[主从同步机制 37](#_Toc198234092)

[详细讲一下binlog 37](#_Toc198234093)

[主从同步延迟的解决方案？ 37](#_Toc198234094)

[死锁的产生条件及排查方法 38](#_Toc198234095)

[覆盖索引和索引下推的区别？ 39](#_Toc198234096)

[MySQL日志？Redo Log和Undo Log？ 39](#_Toc198234097)

[miChange Buffer对写操作的优化原理？ 40](#_Toc198234098)

[在高并发、大数据量场景下，为什么要分库分表？分库分表后怎么保证性能？TPS、QPS指标怎么看？ 40](#_Toc198234099)

[MySQL中的B+树索引结构是怎样的？为什么使用B+树而不是其他结构？ 40](#_Toc198234100)

[聚簇索引和二级索引的区别？ 40](#_Toc198234101)

[为什么需要二级索引？ 41](#_Toc198234102)

[什么是覆盖索引？覆盖索引有什么问题？请举个例子。 41](#_Toc198234103)

[什么字段适合当做主键？ 41](#_Toc198234104)

[Redis 42](#_Toc198234105)

[Redis单线程模型为什么能高效处理请求？ 42](#_Toc198234106)

[缓存穿透/击穿/雪崩的解决方案？ 42](#_Toc198234107)

[RDB和AOF的优缺点及混合持久化机制？ 42](#_Toc198234108)

[Redis Cluster的slot分配算法？ 43](#_Toc198234109)

[大Key问题 43](#_Toc198234110)

[热Key问题 43](#_Toc198234111)

[布隆过滤器 44](#_Toc198234112)

[Redis事务和Lua脚本的原子性区别？ 44](#_Toc198234113)

[Redis Cluster 45](#_Toc198234114)

[阻塞、非阻塞、多路复用、异步I/O 45](#_Toc198234115)

[Java并发 46](#_Toc198234116)

[CAS的原理是什么？存在哪些缺陷？ 46](#_Toc198234117)

[AQS的核心机制是什么？举一个基于AQS实现的工具类。 46](#_Toc198234118)

[ConcurrentHashMap如何实现线程安全？Java7和Java8的区别？ 47](#_Toc198234119)

[synchronized和锁（ReentrantLock）的区别？ 47](#_Toc198234120)

[Synchronized实现 47](#_Toc198234121)

[SpringBoot/SSM 48](#_Toc198234122)

[Spring AOP的实现原理？JDK动态代理和CGLIB的区别？ 48](#_Toc198234123)

[如何解决循环依赖问题？三级缓存机制详解。 48](#_Toc198234124)

[Spring Bean的生命周期和作用域有哪些？ 49](#_Toc198234125)

[SpringMVC请求处理流程？ 50](#_Toc198234126)

[SpringBoot自动配置的实现？ 50](#_Toc198234127)

[Spring事务失效的常见场景有哪些？ 51](#_Toc198234128)

[MyBatis的#和$的区别及SQL注入问题。 51](#_Toc198234129)

[SpringBoot如何集成Tomcat容器？ 51](#_Toc198234130)

[计算机网络 52](#_Toc198234131)

[TCP三次握手的详细过程和状态变化？ 52](#_Toc198234132)

[TCP快速重传和超时重传的区别？ 53](#_Toc198234133)

[TCP拥塞控制？ 53](#_Toc198234134)

[HTTPS和HTTP有哪些区别？ 54](#_Toc198234135)

[HTTPS工作原理是什么？它是如何实现数据加密的？ 54](#_Toc198234136)

[HTTP1.1新特性？ 55](#_Toc198234137)

[HTTP2新特性？ 55](#_Toc198234138)

[HTTP3新特性？ 55](#_Toc198234139)

[操作系统 56](#_Toc198234140)

[进程和线程的区别？协程的优势是什么？ 56](#_Toc198234141)

[Select、poll、epoll的区别？ 56](#_Toc198234142)

[用户态和内核态切换的开销来源？ 56](#_Toc198234143)

[进程间通信 共享内存和消息队列的对比？ 57](#_Toc198234144)

[零拷贝技术（sendfile）的实现原理？ 57](#_Toc198234145)

[内存映射（memory map, mmap）系统调用的实现原理？ 57](#_Toc198234146)

[虚拟内存的作用？ 57](#_Toc198234147)

[Java集合 58](#_Toc198234148)

[详细描述HashMap的扩容机制（触发条件、rehash过程）。 58](#_Toc198234149)

[HashMap底层结构是什么？如何解决哈希冲突？ 58](#_Toc198234150)

[为什么HashMap线程不安全？举例说明并发问题场景。 59](#_Toc198234151)

[HashMap的遍历方式有哪几种？哪种效率更高？ 59](#_Toc198234152)

[Java基础 60](#_Toc198234153)

[Java的8个基本数据类型 60](#_Toc198234154)

[介绍一下反射的底层原理？怎么获取的？能做哪些处理？ 60](#_Toc198234155)

[强引用、软引用、弱引用和虚引用 60](#_Toc198234156)

[JVM 62](#_Toc198234157)

[JVM内存模型核心区域？ 62](#_Toc198234158)

[讲一下你熟悉垃圾回收器？ 62](#_Toc198234159)

[新生代和老年代的占比有了解吗？ 63](#_Toc198234160)

[类加载的过程分为哪几个阶段？ 63](#_Toc198234161)

[双亲委派模型的工作机制是什么？有什么优缺点？ 63](#_Toc198234162)

[在项目中遇到过 Full GC 或内存泄漏的情况吗？如何排查和解决的？ 63](#_Toc198234163)

[Linux命令 65](#_Toc198234164)

[如何用grep查找包含"error"的日志并统计次数？ 65](#_Toc198234165)

[awk如何实现按列求和？ 65](#_Toc198234166)

[如何用sed批量替换文件中的字符串？ 65](#_Toc198234167)

[strace和perf工具的作用及使用场景？ 65](#_Toc198234168)

[如何用tcpdump抓取指定端口的SYN包？ 65](#_Toc198234169)

[【背调环节】 66](#_Toc198234170)

[入职时间？实习多久？ 66](#_Toc198234171)

[最新技术趋势？ 66](#_Toc198234172)

[AI工具？大模型？智能体？ 66](#_Toc198234173)

[如何快速学习一门新技术？遇到不会的技术领域问题怎么解决？ 66](#_Toc198234174)

[未来三年职业规划？ 66](#_Toc198234175)

[【反问环节】 67](#_Toc198234176)

【自我介绍】

**面试官**您好，我叫**胡景峰**。

**本科**就读于 合肥工业大学 计算机学院

**硕士**就读于 东南大学 软件学院（明年6月毕业）

本科和硕士阶段的 **课程成绩排名** 均为前5%

硕士组内是以 **虚拟现实与人机交互** 为大方向

我的科研方向主要是 **渲染+边缘计算** 做一些优化

个人技术栈以**Java**为主，熟悉Spring、MySQL、Redis等

平时有写文档或博客**反思总结**的习惯

相信**长期主义**，有意培养小而正确的习惯，希望借助复利效应变得更好

* 睡眠时间早起时间、英语单词打卡
* 声动早咖啡播客、CSDN极客日报
* 俯卧撑、核心训练

Hello, my name is Jingfeng Hu.

I **completed undergraduate studies** in Computer Science at Hefei University of Technology. I’m currently **pursuing Master's degree** in Software Engineering at Southeast University. （and will graduate in 2026.）

I have earned **several honors**, including:

* National Scholarship
* Title of Excellent Graduate

and **prizes from competitions** like

* China Software Cup
* Mathematical Modeling Contest

I

* + - **value teamwork**,
    - **enjoy challenges**,
    - and am committed to **continuous learning and improvement**.

I believe my **technical skills and positive attitude** will allow me to **quickly adapt and contribute effectively to our team**.

【微信读书】

美团《长期有耐心》

* 王兴的“四纵三横”理论：
* 用户需求四个领域：娱乐、信息、通信、商务
* 技术变革三个方向：搜索、社交网络、移动互联网
* **学会取舍和做减法，聚焦关键战场：  
  千团大战中，因为资源有限，美团舍弃经济排名靠后的城市，因“这些城市不会影响整体胜负”。**
* 持续迭代、长期主义：  
  文档“三遍检查流程”：自我审查、细节校对、换位思考

《华为工作法》

* **“先瞄准目标，再开枪”：盲目行动只会消耗资源。**
* “把时间留给少数重要的事”：二八法则
* “‘差不多’是误差的放大器”：量化标准抵御人性惰性

腾讯《不止于培训》

* **“需求没搞清楚，就不要开始后面的工作”：  
  问题导向思维，挖掘业务痛点，而非被动接受需求。**
* 方案设计需“以终为始，重疗效”，**反对“花哨”的形式主义**。通过“线上线下结合”“资源多元化”等策略，**将培训转化为可量化的业务成果**。

《阿里巴巴管理三板斧》

* **基层**管理三板斧：**定目标、追过程、拿结果**将目标拆解为可量化的行为标准
* **中层**管理三板斧：**揪头发、照镜子、闻味道**管理者不要困于事务性工作、跳出执行层思考战略
* **高层**管理三板斧：**定战略、造土壤、断事用人**

《小米创业思考》

* **商业**本质是**创造普惠价值**，给最多的人带来最大化的美好幸福感。
* 七字“**专注、极致、口碑、快**”

【竞赛收获】

中国软件杯

* **在未知领域中快速学习与落地**：基于**金蝶云低代码平台**开发**图书管理系统**
* **协调资源、带领团队**的组织能力

数学建模竞赛

* 与队友**连续作业三天三夜**，体会到**持续专注地推进复杂任务的成就感**
* **在高压下保持思维清晰、推进任务**的能力

I’m especially good at

* + - **learning new technologies quickly**
    - and **staying focused under pressure**.

In National Software Cup, I

* **self-learned a low-code platform**
* and led my team to build a **Book Management System**.

In Mathematical Modeling Contest, I

* + - **improved my logical thinking**
    - during **a high-pressure seventy-two-hour challenge**.

【吃了么外卖平台】

**网络教程项目**

面向**后台管理**而非用户

主要想通过项目实战 **理解并应用 Spring IoC、AOP**

分页查询

为什么要分页

1. 一次性获取所有数据会消耗大量的**内存**和**时间**
2. 方便用户**浏览和定位**

MyBatis + PageHelper实现

1. 调用**startPag**e将 当前页码、每页记录数 存储在ThreadLocal
2. @Intercepts拦截Executor.query方法，加上 LIMIT 实现分页查询  
   **LIMIT子句**：LIMIT 10 OFFSET 9990;  
   跳过前9990条，只取接下来的10条，等于是你想看第1000页
3. 结果封装到**PageInfo**中
   1. .getList(); // 当前页数据
   2. .getPageNum(); // 当前页码：2
   3. .getPageSize(); // 每页记录数：2
   4. .getTotal(); // 总记录数：6
   5. .getPages(); // 总页数：3

深度分页查询

当分页到比如1k页以上时，导致**深度分页查询**

**问题：**

1. 需要**扫描大量的数据页**才能定位到指定页的数据
2. **为了排序 筛选 分页，临时结果集占用过多内存**，使用磁盘存储数据，慢

**优化方案：**

1. **覆盖索引，避免回表，减少I/O：**  
   如果查询只需要 id、name 列，且这两列在一个联合索引中，那么可以创建覆盖索引来优化查询。
2. **游标分页，WHERE子句过滤数据：**WHERE id > ? LIMIT 10 替代 OFFSET  
   使用上一页的最后一条记录的主键作为下一页查询的条件。
3. **优化数据库的缓冲池大小，高频访问数据常驻内存**  
   增加 innodb\_buffer\_pool\_size 参数，例如设置为物理内存的 60~70%
4. **热门商品前10页缓存在Redis**  
   redisKey = "user:list:page:1:size:20"
5. **分库分表减少单表数据量**
   1. 构建辅助索引表，预先知道第 N 页应该访问哪个表的哪段数据
   2. ShardingSphere中间件自动解析 SQL，决定访问哪些表

订单状态定时处理

* **超时未支付订单自动取消**未支付 && 下单时间 < LocalDateTime.now().minusMinutes(15)  
  @Scheduled(cron = "0 \* \* \* \* ?")
* **长时间处于派送中的订单自动完成**（比如tb用户已收货但未确认收货）  
  派送中 && 派送开始时间 < 当前时间-60min  
  @Scheduled(cron = "0 0 1 \* \* ?")  
  凌晨1点是为了避开日志、备份等系统维护任务

Spring Task实现定时任务

1. 启动类@EnableScheduling开启定时任务
2. 方法添加@Scheduled，结合Cron表达式定义执行时间  
   秒 分 时 日 月 周 [年]
3. 在OrderMapper添加方法查询特定状态和时间条件的订单，并批量更新  
   @Select("SELECT \* FROM orders WHERE status = #{status} AND order\_time < #{orderTime}")

List<Orders> getByStatusAndOrderTimeLT(@Param("status") Integer status, @Param("orderTime") LocalDateTime orderTime);

1. OrderMapper.xml中，添加updateBatch方法的 SQL 实现

Spring Task的问题和解决

1. **问题：默认单线程，任务阻塞导致其他任务无法执行**解决：配置线程池ThreadPoolTaskScheduler允许多个任务并发执行  
   spring.task.scheduling.pool.size=10

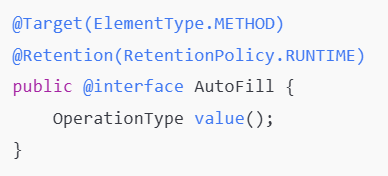
spring.task.scheduling.thread-name-prefix=scheduled-task-

1. **问题：抛出未捕获的异常导致任务停止**解决1：try-catch 捕获所有异常Exception e  
   解决2：实现 ErrorHandler 接口，统一处理定时任务中的异常
2. **问题：集群环境中部署多个应用实例时，导致任务重复执行**解决1：基于 Redis的分布式锁RedisLockRegistry  
   在配置类中定义 RedisLockRegistry Bean  
   在定时任务方法中获取锁Lock lock = redisLockRegistry.obtain("myTaskLock");  
   解决2：分布式任务调度框架Quartz   
   添加依赖  
   配置属性  
   创建一个实现了 Job 接口的类，execute定义任务逻辑  
   在配置类中定义 JobDetail 和 Trigger（cronSchedule）：

公共字段自动填充

目标：自动填充**创建时间、更新时间**等字段

自定义注解+ AOP面向切面实现

1. 定义枚举类 OperationType，用于标识INSERT还是UPDATE
2. 自定义注解 @AutoFill，用于标识需要进行公共字段自动填充的方法  
   
3. 创建**切面类**AutoFillAspect  
   通过 AOP 拦截添加了 @AutoFill 注解的方法  
   
4. 在Mapper 方法上**添加** @AutoFill并指定操作类型INSERT还是UPDATE

SSM

MVC

1. **请求到达DispatcherServlet**
2. HandlerInterceptor#**preHandle()拦截**：用于权限、日志
3. **HandlerMapping映射**：根据请求的 URL、HTTP 方法映射合适的Controller
4. **HandlerAdapter适配**：调用Controller处理
5. 返回**ModelAndView 对象或 JSON 对象**
6. HandlerInterceptor#**postHandle()拦截**
7. ViewResolver**解析**：逻辑视图解析为实际视图、对象序列化为JSON**、**
8. 视图渲染引擎**渲染**：根据 Model 中的数据渲染HTML页面
9. 响应给客户端

IoC

#### 依赖注入3种方式

控制反转IoC是**思想，**依赖注入DI是实现方式

1. **构造器注入（推荐）**：适用于强制依赖的注入
2. **Setter注入**：适用于个人开发模块的注入
3. **@Autowired字段注入（不推荐）**：无法通过构造器或Setter方法修改依赖关系，降低了可维护性、可测试性。

#### Bean 的作用域

1. **singleton（默认）：** 单例bean
2. **prototype：**适用于验证码这种短期使用的、有状态、非线程安全的对象
3. request： 每个请求（如表单数据处理）创建一个实例
4. session： 每个会话（如用户信息缓存）
5. application/global-session： 在Web应用启动时（如统计数据）
6. websocket： 每个WebSocket会话（连接状态管理）

#### Bean 的生命周期

1. 容器 通过反射 根据配置文件 **实例化 Bean**
2. 依赖注入
3. 初始化
   1. Aware回调注入应用：使Bean获取上下文信息、其它 Bean，用于日志标识
   2. BeanPostProcessor#postProcess**BeforeInitialization**  
      应用：创建动态代理实现AOP
   3. **Initialization**  
      应用：缓存预热、数据库连接池初始化
      * + @PostConstruct
        + InitializingBean#afterPropertiesSet
        + initMethod
   4. BeanPostProcessor#postProcess**AfterInitialization**
4. **使用**
5. **销毁** @PreDestroy调用DisposableBean#destroy

AOP

#### 切面 = 切点 + 通知

* **切面：**封装了权限校验、日志记录通用功能
* **切点：**要织入的方法
* **通知：**拦截后要执行的操作

通过代理的方式，拦截穿插，再调用真正方法

#### 动态代理：JDK vs CGLIB

**JDK动态代理（SpringFramework）**

* 基于接口，通过**反射**机制实现
* 注意没有接口的话会有报错

**CGLIB动态代理（SpringBoot2）**

* 基于类继承，通过ASM 字节码生成工具生成子类
* 注意不能代理 final 类和 final 方法

#### 反射

运行时动态创建对象、访问字段、调用方法，无需在编译时知道类的具体信息

Class.forName("com.xxx")

* .getConstructor().newInstance()
* .getField("xxx").setAccessible(true)
* .getMethod("xxx", String.class).invoke(obj, "param")

**注意：**

* **避免过度依赖反射**，因为动态解析和方法调用导致性能开销比直接调用大

**缓存**反射获取的类、方法、字段，减少操作频率

【12306 铁路购票系统】

GitHub找的开源项目二开，SpringBoot3 + JDK17 = 1w行左右的源码

大规模的在线火车票订购平台

主要学习**数据一致性**、对**高并发**有一个实践和应对

12306主要模块

1. **网关模块gateway-service**：JWT令牌校验、请求路由转发
2. **用户模块user-service**
   * 用户注册缓存穿透
3. **购票模块ticket-service**
   * 数据最终一致性 = Binlog监听 + 消息队列
   * 令牌限流余票扣减（幂等性、\*责任链）
4. **订单模块order-service**
   * 延时消息取消超时未支付订单
   * 订单数据分库分表 = 雪花算法 + 基因法
5. **支付模块pay-service**

？？？微服务框架组件？业界常用架构？

Spring Cloud Alibaba微服务架构

项目依赖**Nacos、Redis、RocketMQ**

部署在**云服务器**上（GitHub开发者提供）

通过 **VM 参数**指定这些中间件的**域名**

（1）Spring Cloud Gateway 实现统一网关入口

根据**URL、请求方法、请求头**路由到相应的微服务实例上

1. 引入gateway依赖
2. 配置
   1. id // 路由ID，唯一标识这个路由规则
   2. uri // 路由目标地址，lb://表示使用服务发现（LoadBalancer）
   3. predicates // 路由断言规则，决定哪些请求匹配此路由  
      - Path=/api/user/\*\*
   4. filters // 过滤器
      1. blackPathPre列出了一些需要**跳过过滤的路径前缀**

（2）Nacos 作为注册中心和配置中心

**工作原理：**

* 应用启动时，Nacos 客户端会向服务端拉取配置并定期轮询；
* 服务端变更配置时会基于长轮询推送变更通知，客户端收到后@RefreshScope刷新配置 Bean；
  + 长轮询：客户端发一次请求，服务端如果暂时没变化，会阻塞一段时间直到有结果或超时，节省大量无效请求

**如果 Nacos 集群发生故障，如何保障服务的正常运行：**

1. 采用3节点以上的集群部署、使用MySQL做配置持久化，避免单点故障。
2. Nacos 客户端本地有配置文件快照，能从本地加载配置，保障服务运行。
3. 使用负载均衡机制（如 lb://）自动选择可用服务实例，如果服务注册失败或注册中心短时不可用，服务本身不会立即挂掉。
4. 配置 Prometheus + Grafana 对 Nacos 节点健康状态做监控告警。

**实操：**

1. 引入nacos-discovery依赖  
   引入nacos-config依赖
2. 配置文件指定server-addr，但VM 参数的优先级更高
3. 通过@FeignClient指定name调用服务
4. 通过@RefreshScope使用动态配置和自动刷新

（3）Sentinel实现熔断限流

**熔断**：防止故障扩散，暂时切断对该服务的调用

* 主从复制：主节点负责写操作，多个从节点处理读操作
* 通过 Sentinel 在主节点故障时自动将一个从节点提升为新的主节点

**限流：**

* 令牌桶算法：桶里放令牌，请求获取到令牌才能被处理
* 漏桶算法：桶里放请求，以固定的速率取出请求处理

1. 引入sentinel依赖
2. 使用@SentinelResource
   1. value = "getUserInfo" // 资源名，标识要保护的服务
   2. blockHandler = "handleBlock" // 当触发限流、降级时的兜底处理方法
   3. fallback //抛出异常时的处理方法（可选）

用户注册缓存穿透

* [手摸手之注册用户如何防止缓存穿透？](https://www.yuque.com/magestack/12306/go6vg8whk9g1lyhp) 
  + [用户注册布隆过滤器容量设置以及碰撞率问题](https://www.yuque.com/magestack/12306/fr3zurztaq3xwfkc)
* [手摸手之实现敏感信息加密存储](https://www.yuque.com/magestack/12306/vhf4i3c604t2qex7)
  + [核心技术文档-如何防止用户敏感数据泄露](https://www.yuque.com/magestack/12306/cd9zbuugg663qsu4)
* [手摸手之用户敏感信息展示脱敏](https://www.yuque.com/magestack/12306/myl4gqx84bxyxmay)
* [如何防止用户敏感数据泄露](https://www.yuque.com/magestack/12306/cd9zbuugg663qsu4)
* [缓存击穿之双重判定锁如何优化性能？](https://www.yuque.com/magestack/12306/xrtg5mibquardvvi)（逻辑清晰、简明易懂）

缓存三剑客

* **穿透** 不存在：Null标识、互斥锁、布隆过滤器
* **击穿** 热点：热点数据永不过期、互斥锁
* 多个 **雪崩**：随机过期时间、双缓存

缓存穿透解决方案

1. 缓存不存在的 Key并把值设为Null，设置短过期时间如60秒  
   **缺点：**尝试但没注册一个不存在的用户名，该值60秒内都不可被注册
2. 互斥锁 保证 只有一个线程访问数据库  
   **缺点：**其他用户注册请求缓慢或超时
3. **布隆过滤器** 存已注册用户名
   * 布隆过滤器不存在 = 数据库没有 = 可用
   * 布隆过滤器中有，再查缓存或数据库

缺点：不能删除元素，注销用户名无法再次使用

1. **布隆过滤器 + Redis Set缓存注销用户名**

* 布隆过滤器不存在 = 数据库没有 = 可用
* Redis Set缓存存在 = 已注销 = 可用
* 查询数据库到底有没有

缓存击穿解决方案——双重判定锁

* **在获取锁之前**查缓存，因为缓存命中无需获取锁
* **在获取锁之后**查缓存，因为等待锁时可能有其他线程回写缓存

令牌限流+分布式锁余票扣减

* [**手摸手之车票搜索为什么用Redis而不是ES？**](https://www.yuque.com/magestack/12306/zd9wok8w0dn8eig5) 1
* [**手摸手之如何完成列车数据检索**](https://www.yuque.com/magestack/12306/tygc8hs113al2c2z) 1
* [**节假日高并发购票Redis能扛得住么？**](https://www.yuque.com/magestack/12306/eq6v9p9dfre117mg) 1
* [**手摸手之实现列车购票流程**](https://www.yuque.com/magestack/12306/nmmgqkgbfxb2bwl0) 1
  + [**手摸手之实现用户购票责任链验证**](https://www.yuque.com/magestack/12306/ggg2txzbfgfqp6tm) 1
  + [核心技术文档-从根上理解Redis分布式锁演进架构](https://www.yuque.com/magestack/12306/ag5pffwexihshe2s) 1
* [**手摸手之实现v2版本列车购票流程**](https://www.yuque.com/magestack/12306/ov3u6lpgartx0mst)（Lua脚本不太会）
  + [**缓存击穿之双重判定锁如何优化性能？**](https://www.yuque.com/magestack/12306/xrtg5mibquardvvi)（逻辑清晰、简明易懂）
  + [**高并发库存扣减为什么需要令牌限流？**](https://www.yuque.com/magestack/12306/gdqnbhp10xx9g9kv)
* [**购买列车中间站点余票如何更新？**](https://www.yuque.com/magestack/12306/efhhtogdr8ouz6t2)
  + [**缓存与数据库一致性如何解决？**](https://www.yuque.com/magestack/12306/wocbrht50ctg14nv)（加星，很详尽）
  + [**手摸手之列车余票如何保障缓存数据库一致性**](https://www.yuque.com/magestack/12306/glv5e0785b2d7oag)
* [**余票Binlog更新延迟问题如何解决？**](https://www.yuque.com/magestack/12306/kxuug8l6zslfyz21)
  + [**购买列车余票如何防止库存超卖？**](https://www.yuque.com/magestack/12306/pemih6p8h7xaw2b3)

余票扣减核心流程

**查询车票**

1. **HTTP防止重复提交**
2. **责任链验证提交参数**

**购买车票**

1. **Redisson分布式锁：  
   同一列车、同一时间、单个用户可以进行座位分配以及创建订单行为**
2. **座位分配、创建订单**

注意：

问题：假如一趟列车有几十万人抢票，但是真正能购票的用户可能也就几千人。也就意味着哪怕几十万人都去请求这个分布式锁，最终也就几十万人中的几千人是有效的，其它都是无效获取分布式锁的行为。

目标：让少量用户去请求获取分布式锁

方法：令牌桶算法以固定速率生成令牌限流，而12306中将没有出售的座位当作令牌放到一个容器中，令牌是有限的，获取后数量会相应减少，防止超卖。

0、查询车票

TicketController的pageListTicketQuery方法

1. **责任链验证数据**
   1. **空**
   2. **出发日期≥当前日期、出发地和目的地存在、出发站和到达站存在**
2. **获取分布式锁**，避免缓存击穿

**双重判定锁**再查一次缓存

1. **查询站点station以及城市region两张表**，加载城市站点数据
   1. 搜索南京-徐州，列出南京-滁州-蚌埠-徐州  
      （通过批量查询方式获取出发站点和到达站点对应的城市集合）
   2. 缓存中存放键值对，保存**站点**与**城市**的关联关系
   3. 通过站点关联到城市，通过城市查询列车
   4. 缓存中列车的数据结构  
      Key：起始城市\_终点城市\_日期  
      Value：  
       Key：列车ID\_起始站\_终点站  
       Val：列车详细信息（出发到达时间）
   5. 单独存储余票信息，因为余票数据是实时变更的
2. 查询**列车余票信息（座次、价格）**
3. 通过**构建者模式**构建列车查询返回数据  
   return TicketPageQueryRespDTO.builder()

.trainList(seatResults)

.departureStationList(buildDepartureStationList(seatResults))

.arrivalStationList(buildArrivalStationList(seatResults))

.trainBrandList(buildTrainBrandList(seatResults))

.seatClassTypeList(buildSeatClassList(seatResults))

.build();

#### 查询车票为什么用Redis不用ElasticSearch？

通过F12网络抓取发现

* 只有点击查询时吗，后端根据**出发地、目的地、日期**一次查询
* 页面上的搜索条件（车次类型、席别、出发站）大多是前端筛选，采用构建者模式构建
* 不涉及全文搜索和模糊搜索

1. **实时性**：Redis内存存储、单线程模型 + 非阻塞I/O

2. **部署成本：**Redis轻量、易扩展，而ElasticSearch非常消耗资源

？？？0、购票分段问题？余票是怎么存储的？数据库里？缓存里？

1、接口幂等：HTTP防重复提交

**Redis去重表**记录已经处理过的请求，防止重复执行。

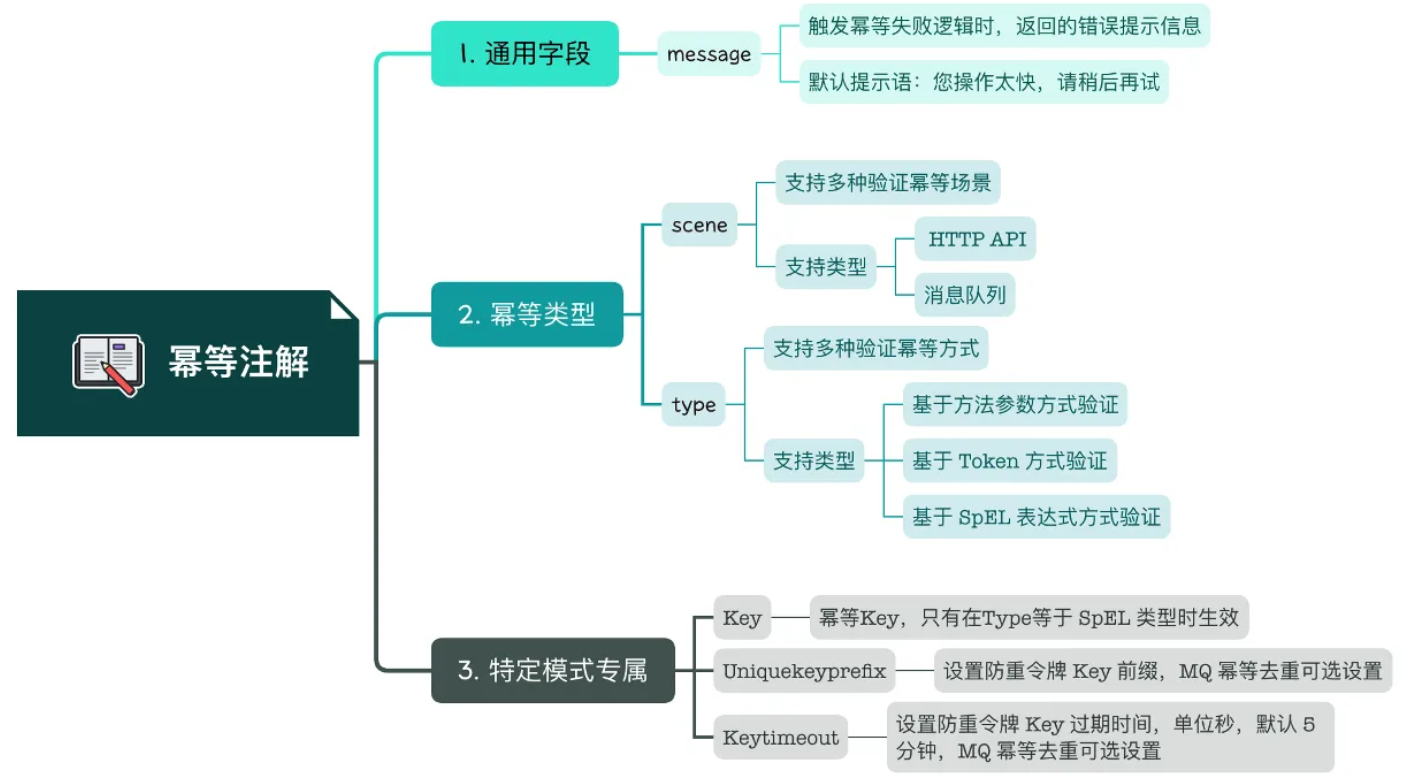
* 存在对应记录，表示请求已经执行过，直接返回而不重复操作
* 不存在对应记录，执行相应业务逻辑，在处理完成后将请求的唯一标识（如请求 ID 或标识）添加到Redis去重表中

**幂等注解@Idempotent**

1. scene：接口幂等 or 消息队列幂等
2. type：TOKEN or PARAM or SPEL

**幂等 AOP**

1. 根据scene和type获取幂等处理器
2. 执行幂等逻辑
3. 通过 instance.postProcessing() 进行分布式锁的解锁等行为



2、责任链验证提交参数

责任链模式思想的体现：

* Java异常发生时，系统会按照 catch 块的顺序依次尝试处理
* Java Web 的FilterChain接口，决定是否继续传递请求
* Spring MVC 的HandlerInterceptor拦截器在请求处理前后执行逻辑

1. 定义过滤接口ChainFilter
2. 一条业务线内，可能有多个 Handler 执行各种各样的校验任务，编写对应 Handler 实现类：NotNullChainHandler、VerifyChainHandler
   1. 处理逻辑handler
   2. 执行顺序getOrder
3. 多个实现需要一个容器组织起来，编写 Context 类容纳若干实现类，并
   1. 提供一个 HashMap存储责任链业务标识-责任链组件实现
   2. 创建责任链上下文AbstractChainContext
   3. 调用AbstractChainContext的方法执行责任链中的处理器
   4. 通过handler传入 mark 获取到本次需要执行的责任链组件，依次执行

3、分布式锁

**Old 1、setIfAbsent**向Redis中添加锁标志位lockKey，添加成功才能继续执行

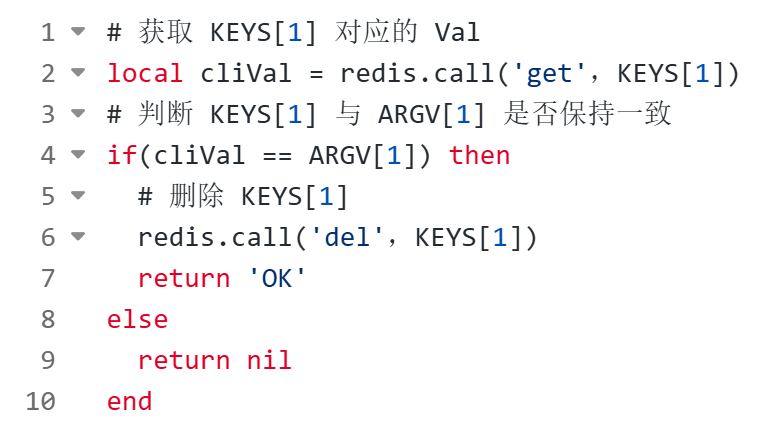
问题：异常结束/服务宕机没有释放锁，会产生死锁

**Old 2、加过期时间cache.expire(lockKey, 5,TimeUnit.SECONDS);**

**问题：加锁成功后、设置过期时间前，宕机**

* **cache.set向Redis中添加锁标志位lockKey**：用UUID识别身份防止释放别人的锁 **cache.set(lockKey, "lock", "nx"，"ex"，5);**
* **Lua脚本**：防止CPU调度，在识别身份后、释放前的时间窗口，释放了别人的锁

**cache.eval(script，Lists.newArrayList(lockKey)，**



3、令牌限流

**目标：**防止大多数无效购票请求访问数据库

根据所选择的**乘车人数量以及座位类型**获取令牌takeTokenFromBucket

1. 令牌容器失效则二次校验赋值（为什么不直接用余票缓存限流？）
2. 令牌容器结构（按TrainId获取令牌）
   1. Key：TrainId
   2. Value
      1. Key：出发站\_终点站\_座位类型
      2. Value 余票
3. 准备数据
   1. actualHashKey：令牌的Key
   2. luaScriptKey：出发站\_到达站
   3. seatTypeCountMap：需要扣减的座位类型和对应数量
   4. takeoutRouteDTOList：要扣减的中间站点
4. 执行Lua 脚本获取令牌，看余票是否充足，充足则扣减

#### 为什么不直接用余票缓存限流？

* **传统秒杀架构：**在用户购买车票时，系统会先从余票缓存中扣减库存，然后再更新数据库，完成整个流程。
* **极端情况：**扣减缓存成功后宕机，导致数据库未能成功更新。可能出现前端展示余票为 0，但数据库中实际还有库存的情况。
* **令牌容器：**发现没有剩余可用令牌，可以做二次检查：触发一个请求去比对数据库是否还有值，如果有的话，那么就把令牌容器缓存删除，下个用户再购票时，重新加载即可
* （1）如果余票为0了，前端就不能发起购票请求。（2）在刷新时加锁限制

？？？3、购票是怎么实现高并发效果？令牌限流用什么开源框架了吗？

？？？3、你有看过美团、淘宝关于并发处理实现的技术吗？

？？？3、优化后有没有测试并发量

Binlog监听 + 消息队列 = 数据最终一致性

明显不可取的方案

**方案1：写缓存 → 写数据库（×）**

**方案2：写数据库 → 再写缓存（×）**

**方案3：删缓存 → 写数据库（×、**苍穹外卖**）**

可根据实际情况选用的方案

**方案4：写数据库 -> 删缓存**

* **适用：**一致性要求低，比如**社媒内容的再编辑**

**方案5：（延时双删）删缓存 → 写数据库 → 延迟再删缓存**

* **适用：**一致性要求高、读多写少，比如**商品详情页**

**方案6：写数据库 → Binlog异步更新缓存**

* **原因：**高并发场景删除缓存可能导致击穿并不合适
* **适用：**一致性要求高、高并发，比如**电商库存扣减**

核心流程

1. 写请求扣减数据库余票
2. 通过Canal监听Binlog，推送到消息队列
3. 拉取队列消息更新Redis缓存

Canal如何抓取Binlog？

1. MySQL开启Binlog并设置为 ROW 模式
   1. STATEMENT：记录每一条修改数据的 SQL。动态函数问题：比如你用了 uuid 或now 这些函数，你在主库上执行的结果并不是你在从库执行的结果，导致复制的数据不一致
   2. ROW：记录行数据最终被修改成什么样。binlog文件过大问题：update 批量更新多少行数据就会产生多少条记录，而在STATEMENT 格式下只会记录一个 update 语句而已；
   3. MIXED：根据不同的情况自动使用 ROW 模式和 STATEMENT 模式
2. Canal配置要监听的数据库信息、设置消息队列相关参数
3. 实现RocketMQListener接口的onMessage方法
   1. 解析消息的table、type、data
   2. 处理

Binlog过多，消息堆积了怎么办？

**原因：**

1. **数据写入量大：**高并发下 MySQL 产生大量 Binlog
2. **消息消费慢：**消费者处理能力不足
3. **Canal** 吞吐不高、批次过小
4. **RocketMQ** 消息未被及时消费，积压在 Broker 队列中

**解决：**

1. **MySQL -> Canal 优化：限制只监听真正需要同步的表，减少不必要的 Binlog 捕获**  
   canal.instance.filter.regex = yourdb.order, yourdb.user
2. **Canal -> RocketMQ 优化：扁平化消息体，消费者处理更轻量**  
   canal.mq.flatMessage = true
3. **RocketMQ 消费者侧优化（最关键）**
   1. consumeMode = ConsumeMode.CONCURRENTLY并发消费
   2. 批量消费 FlatMessage：一次处理多条记录，减少 MQ 收发次数。

替代方案：RocketMQ5.0基于时间轮的定时任务

延时消息取消超时未支付订单

* [**订单延时关闭功能技术选型**](https://www.yuque.com/magestack/12306/sazb1bn2a1okwtlu)
* [**创建订单并支付后延时关闭订单消息怎么办？**](https://www.yuque.com/magestack/12306/ldw8nxp96yfg7cgx)
  + [手摸手之消息队列正确使用姿势](https://www.yuque.com/magestack/12306/dhm3lr598gtt23od)（看不大懂，要补）

订单延时关闭功能技术选型

#### **定时任务（xxl-job、PowerJob、shardingsphere-elasticjob）**

* + **延时精度问题：**调度器的轮询 + 分发延迟导致的
  + **高并发问题：**大量定时任务同时执行 系统负载过大
  + **分库分表问题：**因为根据创建时间查询扫描一批订单进行关闭没有携带分片键，存在读扩散问题。  
    **读扩散问题：**在分库分表场景下，当查询条件不是分片键时，无法定位到具体要查询的分表，就需要对所有分表都执行查询语句。

#### **Redis 过期监听 / Redisson的RDelayedQueue**

1. **延时精度问题：**Redis 的过期监听通过定时器实现
2. **宕机重启问题：**设置了过期时间&&未过期的订单无法正确关闭
   1. Redis 5.0前如果宕机或者重启，这个消息不会重复投递
   2. Redis 5.0推出了 Stream 功能，有了持久化功能

#### **RabbitMQ**

* **延时精度：**RabbitMQ基于消息的Time-To-Live实现延时消息，消息过期检测是惰性触发的，等下一次投递消息、消费消息才“顺便”检查
* **高并发问题：**消息堆积
* **重复消息问题：**由于网络原因或其他不可预知的因素，可能会导致消息重复发送到订单队列。如果没有处理好消息的幂等性（idempotent），可能会导致订单重复关闭的问题，从而造成数据不一致或其他异常情况。
* **部署难度：**Rabbitmq是一个大型中间件，光docker部署就上1个G了，而且还封闭，性能比不过Rocketmq

RocketMQ延时消息实现

* RocketMQ 的延时消息是指 Producer 发送消息后，Consumer 不会立即消费，而是需要等待固定的时间才能消费。
  + Producer 把消息发送到 Broker 后，
  + Broker 判断到是延时消息，把消息投递到延时队列
  + 线程池会有 18 个线程来对延时队列进行调度，每个线程调度一个延时级别delayTimeLevel，调度任务把延时消息再投递到原始队列，   
    "1s 5s 10s 30s 1m 2m 3m 4m 5m 6m 7m 8m 9m 10m 20m 30m 1h 2h"  
    message.setDelayTimeLevel(14);
  + 这样 Consumer 就可以拉取到了，消息监听器接收到订单关闭消息时，如果仍为未支付，则取消并释放库存。
* 延迟关闭订单消息的消费者放在了购票服务：因为购票服务调用订单服务创建订单，而延时取消订单需要同时需要操作购票服务和订单服务：
  + 修改订单相关状态
  + 解锁座位状态
  + 回滚缓存余票数量
  + 令牌限流数量。
* 延时消息一旦发出去，就一定会执行。所以从业务逻辑判断，避免已支付的订单再被延时关闭。

#### RocketMQ定时消息

RocketMQ 5.0基于时间轮算法引入了定时消息

一个 60s 的时间轮，时间轮上会有一个指向当前时间的指针定时地移动到下一个时间（秒级）

在每个时间节点增加一个 round 字段，记录时间轮转动的圈数，比如对于延时 130s 的任务，round 就是 2，放在第 10 个时间刻度的链表中。

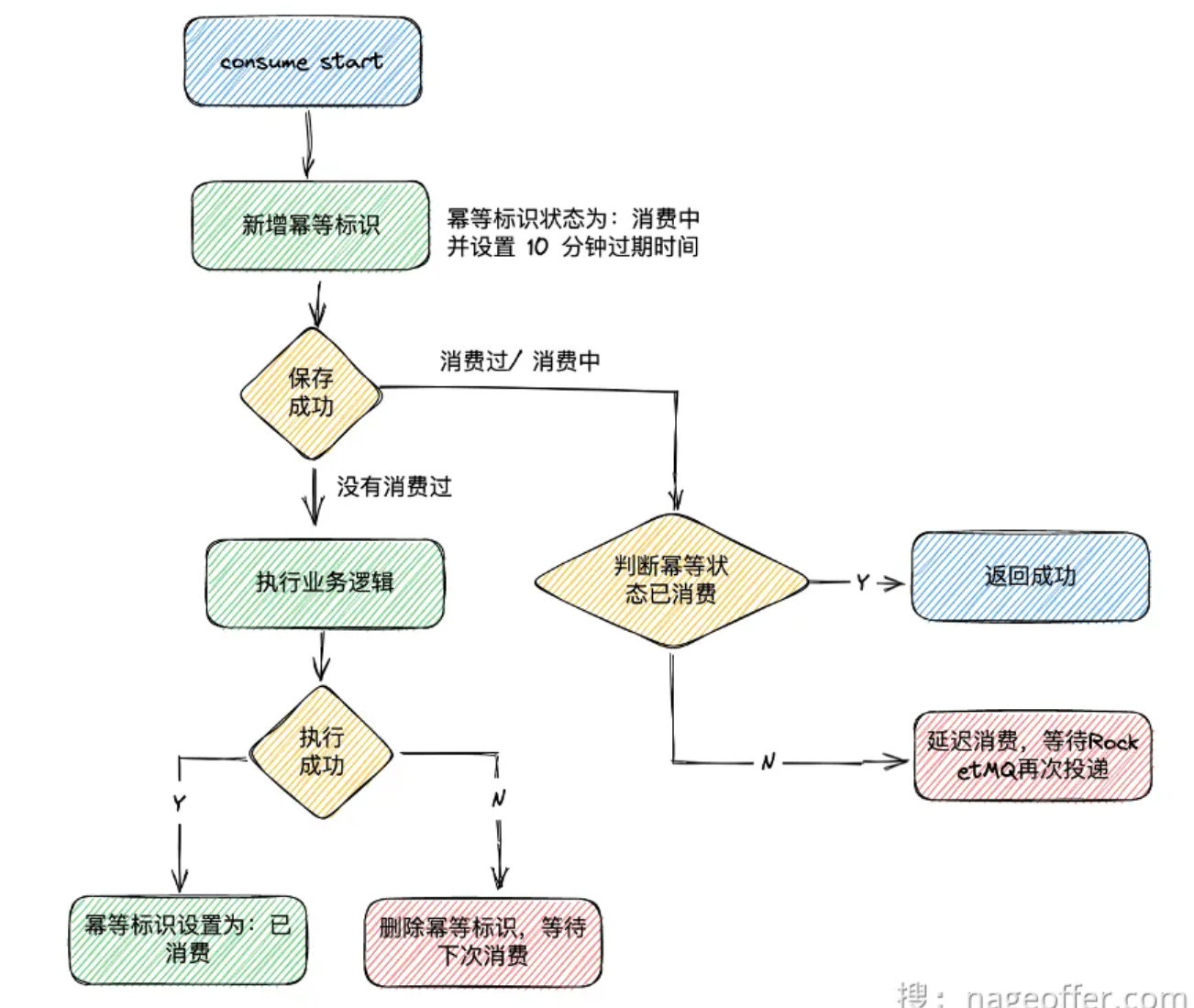
优势：不用去遍历所有的任务，每一个时间节点上的任务用链表串起来，当时间轮上的指针移动到当前的时间时，这个时间节点上的全部任务都执行。

当时间轮转到一个节点，执行节点上的任务时，首先判断 round 是否等于 0，如果等于 0，则把这个任务从任务链表中移出交给异步线程执行，否则将 round 减 1 继续检查后面的任务。

？？？消息队列幂等：防止消息重复消费

本次请求的唯一幂等 Key =

？？？RocketMQ的其他应用场景？什么情况下需要消息队列？



订单数据分库分表 = 雪花算法 + 基因法

* [用户分库分表](https://www.yuque.com/magestack/12306/pb98neetmww1rr9y)
* [乘车人分库分表](https://www.yuque.com/magestack/12306/zhsauz6ksng8wvgf)
* [订单分库分表](https://www.yuque.com/magestack/12306/dyr1d4r3me19gg7l)
* [分布式雪花算法](https://www.yuque.com/magestack/12306/ciigw9ctq0v90u3w)
* [手摸手实现分布式ID组件库](https://www.yuque.com/magestack/12306/lc2yb8gxtvfdt7rp)

为什么分库分表？如何分库分表？

1. 单表数据量过大
2. 表结构复杂：字段过多、索引过多
3. 12306节假日购票并发访问量过高

* 一种分法是，保持表结构相同，水平拆分数据行
* 另一种分法，根据业务，垂直拆分字段

订单分库分表

需求：

* 用户要能查看自己的订单
* 支持订单号精准查询

#### 复合分片键：用户ID、订单ID

1. **保证用户查询自己所有订单时能精准路由到一个分片；**
   1. 也就是明确提供user\_id，用 idx\_userid\_orderid走覆盖索引
2. **支持订单号的精准定位；**
   1. 也就是只有order\_id，提取后六位实现路由，再走idx\_orderid索引
3. 兼容历史遗留数据（旧逻辑生成的订单ID），因此仍保留用户 ID 为分片键中的一部分，提高系统健壮性。

分片算法需要根据分片键的值来操作，例如，范围分片、哈希分片等，定位具体的表，避免读请求扩散（同一用户所有订单都在一个分片里，不用再到其他分片里查找），当定位到具体的表之后，还是需要根据索引来查找具体的记录。

虽然还是拿两个字段作为了分片键，但是由于**自定义了分片算法**，所以无论 SQL 中只有用户 ID，还是只有带用户后六位的订单号，都能找到对应的分片

1. 如果你只设置订单号为分片键，以**用户ID**查询时无法精确路由
2. 如果某些老订单是由旧逻辑生成的订单ID，或者部分测试数据、历史数据中并不满足基因法冗余，那么不能完全依赖订单ID做路由。保留**用户ID** 为分片键更加健壮。

#### **基因法**把 用户 ID 后六位 冗余到 订单ID 里（tb订单）

每次查询时需要带着用户和订单两个字段，非常不方便：

1. 如果SQL包含用户 ID直接取用户 ID 后六位
2. 不包含就从订单号中获取后六位，也就是用户 ID 后六位

#### 其他查询场景（比如查某一时间段内的订单）？

废案：如果采用冗余，那么新的查询维度就要新增一个表

将数据冗余到ES做查询：支持同步和查询、不同视角的索引

雪花算法生成ID（有序、唯一、性能）

64位：

* 41 时间戳：毫秒级（41 位的长度可以使用 69 年）
* 10 机器 ID：1024 个节点
* 12序列号：每个节点每毫秒4096个

#### 会不会超过Long最大值？

雪花算法生成的 ID 是一个 **64 位的long**

String.valueOf()字符串拼接形成最终的订单 ID

美团 Leaf：https://github.com/Meituan-Dianping/Leaf

百度 Uid：https://github.com/baidu/uid-generator

【简历八股】

MySQL

一条SQL的执行流程是怎样的？

1. 连接器校验**权限**
2. 分析器进行**词法**分析、**语法**分析、构建**AST抽象语法树**
3. 优化器选择合适的**索引**、**表连接**顺序
4. 执行器调用引擎层，按**B+树**索引检索、**加锁（行锁、表锁）**
5. 返回结果集

慢SQL优化

#### SQL本身

1. 避免 SELECT \*，只选择需要的列
2. **子查询 ->**  JOIN连接查询
   1. 子查询会先执行一遍，把结果生成出来，再带入外部查询
   2. JOIN选择最佳 JOIN 策略，结合多表上的索引进行优化

Hash Join适合无序大表

Merge Join 适合有序大表

1. 避免在索引列上使用函数或表达式，可能导致索引失效

#### 索引优化

**EXPLAIN分析**执行计划

* select\_type：查询类型：简单查询、主查询、子查询
* type：访问类型：范围扫描、索引扫描、全表扫描
* key：实际用到的索引
* rows：扫描行数越小越好
* Extra：额外信息
  1. Using index表示使用覆盖索引
  2. Using where表示使用WHERE过滤、索引下推
  3. Using temporary表示使用临时表，临时表开销大
  4. Using filesort表示额外排序，数据量大时很不好

#### 数据库配置

1. **增加缓冲池大小**，使更多的数据和索引可以缓存在内存中
2. 将**数据文件和日志文件**分布在不同的磁盘上，减少磁盘 I/O 冲突

#### 数据库服务器性能

1. **读写分离**
2. **集群**负载均衡

事务隔离级别

**事务隔离级别：​**

1. 读未提交**：**
2. 读已提交**：**解决脏读
   * 脏读：一个事务读取到另一个事务未提交（后来回滚）的数据
3. 可重复读（默认）：解决不可重复读
   * 不可重复读：同一事务读取两次，结果不同
   * InnoDB通过间隙锁（Next-Key Lock）解决幻读：  
     不仅锁住已有数据，还锁住数据之间的间隙，以防止插入新数据。
4. 串行化：解决幻读
   * 幻读：同一事务读取两次，结果集记录数量不同

MVCC的实现原理

MVCC多版本并发控制，允许多个事务同时读写数据库，而无需互相等待

原理：

数据修改时生成新版本记录，每个记录保留版本号或时间戳

不同时刻启动的事务可以无锁地读取不同版本的数据

写操作可以继续写，无非就是会创建新的数据版本

举例：

insert (1,XX)不仅存储 ID 1、name XX，还有：

* **trx\_id**：当前事务ID
* **roll\_pointer**：指向UndoLog的指针

readView用于判断哪个版本对当前事务可见：

* **creator\_trx\_id**，当前事务ID。
* **m\_ids**，生成 readView 时**已启动但还未提交**的事务ID列表。
  + **min\_trx\_id**，当前活跃ID之中的最小值。
  + **max\_trx\_id**，生成 readView 时 InnoDB 将分配给下一个事务的 ID 的值（事务 ID 是递增分配的，越后面申请的事务ID越大）

对于可见版本的判断是从最新版本开始沿着版本链逐渐寻找老的版本，如果遇到符合条件的版本就返回，判断条件如下：

* **如果当前数据版本的 trx\_id** == creator\_trx\_id  
  说明修改这条数据的事务就是当前事务，所以可见。
* **如果当前数据版本的 trx\_id** < min\_trx\_id  
  说明修改这条数据的事务在当前事务生成 readView 的时候已提交，所以可见
* **如果当前数据版本的 trx\_id 大小在 min\_trx\_id 和 max\_trx\_id 之间**
  + trx\_id若在 m\_ids 中，说明修改数据的事务还未提交，所以不可见
  + trx\_id若不在 m\_ids 中，表明事务已经提交，可见。
* 如果当前数据版本的 trx\_id >= max\_trx\_id  
  说明修改这条数据的事务在当前事务生成 readView 的时候还**未启动**，所以不可见(结合事务ID递增来看)。

链表 红黑树 B树 B+树

#### 链表

链表查询是 O(n) 的时间复杂度

#### 红黑树

保证了树不会退化成链表

* **左根右‌：**左子树节点的值小于等于根节点的值，右子树节点的值大于等于根节点的值。
* ‌**根叶黑‌：**根节点是黑色的，所有叶子节点（空节点）也是黑色的。
* ‌**不红红‌：**不存在两个相邻的红节点。
* ‌**黑路同‌：**从根节点到任一叶节点的所有简单路径上黑色节点的数量相同‌

#### B树

1. B 树**每个节点**都存储了**完整的数据**  
   B+ 树**非叶子节点**存储**主键/索引值**和**页面指针**，**叶子节点**存储**完整数据**。
2. B+ 树**查询时间更平均、稳定**（从根到叶）
3. B+ 树可以在**内存**中存放**更多索引页**，**减少磁盘IO**
4. B+ 树叶子组成了**链表**，便于范围查询

#### B+树

**1）高效的查找性能**

* B+ 树是一种**自平衡树**，每个**叶子**节点到**根**节点的**路径长度相同**
* 插入和删除节点时会进行**分裂和合并**操作，以保持树的平衡，但它又会有一定的**冗余**节点，使得删除的时候树结构的变化小，更高效。
* 查找、插入、删除等操作的时间复杂度为 O(log n)

**2）树的高度增长不会过快**

* **红黑树**数据越多，树的高度增长就越快。
* B+ 树是**多叉树**，非叶子节点仅保存**主键/索引值**和**页面指针**，使得每一页能容纳更多的记录，因此内存中就能存放更多索引，容易命中缓存，使得查询磁盘的 I/O 次数减少。

**3）范围查询能力强**

* B+ 树适合范围查询。因为**叶子节点通过链表链接**，从根节点定位到叶子节点**查找到范围的起点**之后，只需要**顺序扫描**链表即可遍历后续的数据。

B+树索引的查询过程：​

从**根**节点自上而下逐层比较确定下一层子节点的位置，最终定位到**叶子**节点

如果是范围查询，叶子节点通过链表连接，可以高效地遍历满足条件的数据。

#### 为什么不用哈希索引？

B+树**有序 范围查询**、哈希索引**随机分布 等值查询**

MySQL 三层 B+ 树能存多少数据？

* B+树默认**数据页大小** = 16KB（可以通过innodb\_page\_size修改）
* 每个数据记录大小 = 1KB（一般会比这个小，这里取整方便计算）
* 索引键（BIGINT）8 B + 每个指针6 B（页地址）

**三层 B+ 树的存储计算：**

* 第三层**叶子节点页：**存储 16KB ÷ 1KB = 16 条数据记录
* 第二层**中间节点页**：指向 16 \* 1024 ÷（6+8） = 1170 个叶子节点
* 第一层**根节点**：指向 1170 个中间节点
* 数据总量 = 1170 \* 1170 \* 16 = 2190 2400（大约 2000 万条记录）

#### 这道题其实也是在问，什么时候需要考虑分表？

2000万条记录，2000**万**是个性能瓶颈，单表数据量超过 1000万~2000万

* B+树层数会增加，查询路径变长；
* 页分裂/合并频繁，写入开销变大；
  + 插入新记录时，如果聚簇索引页满，会触发页分裂
  + 页分裂频繁会产生大量碎片
* Buffer Pool命中率降低，I/O压力增加；

#### 三层B+树，加行锁时会发生什么？是否还会加其他锁？

* 三层B+树说明数据量较大
* 加行锁时，锁的是叶子节点对应的索引项。

同时可能还隐式加意向锁（表级别的意向锁 IS 或 IX），用于标识自己想加行锁。

* 如果事务冲突严重、锁升级机制触发，可能会升级为表锁。

行锁、表锁、意向锁

**表锁：**

* 适合批量操作、大事务。
* 如果没有命中索引、全表扫描，可能会从行锁自动升级为表锁。

**行锁：**

* 锁的是索引项，而不是物理行。需要走索引才能加行锁，否则是锁全表。
* 适合并发高、细粒度控制。
* **临键锁 = 记录锁 + 间隙锁：**锁定某一行记录和间隙，防止幻读。
* **插入意向锁：**多个插入操作并发时用

**意向锁：**

* **场景：**当事务想加行锁时，需要检查是否已经被其他事务加上了表锁。
* **解决：**通过意向锁状态，无需逐行检查、快速判断是否可以安全地加行锁



分库分表后全局ID生成方案有哪些？

1. ​UUID：无序。  
   String uuid = UUID.randomUUID().toString();  
   Generated UUID: e3e70682-f3a1-11e9-8f0b-362b9e155667
2. ​Snowflake：时间戳 + 机器ID + 序列号。 时钟回拨可能引发问题。  
   0 | 41-bit 时间戳 | 10-bit 机器 ID | 12-bit 序列号
3. 数据库自增ID：依赖数据库。  
   
4. ​Redis自增ID：存在单点故障风险。  
   
5. ​号段模式​：从数据库预分配一批ID号段，本地缓存使用。号段用尽需重新申请，可能短暂阻塞。

主从同步机制

复制的过程就是将 binlog 中的数据从主库传输到从库上

1. **写入 Binlog：**会先写入 binlog，再提交事务，更新数据。
2. **同步 Binlog：**创建一个专门的 I/O 线程，连接主库的 log dump 线程，来接收主库的 binlog 日志，再把 binlog 信息写入 relay log 的中继日志里
3. 回放 Binlog：回放 binlog，并更新存储引擎中的数据。

MySQL日志？Redo Log和Undo Log？

1. 错误日志：服务器崩溃、表损坏信息
2. 查询日志：记录所有查询、更新SQL 语句，严重影响性能、不在线上开启  
   SET global general\_log = 1;

SET global general\_log\_file = '/var/log/mysql/mysql.log';

1. 慢查询日志：记录执行时间超过10s的SQL  
   SET global slow\_query\_log = 1;

SET global long\_query\_time = 2; -- 超过2秒算慢查询

SET global slow\_query\_log\_file = '/var/log/mysql/mysql-slow.log';

1. 二进制日志（Binlog）：用于主从复制、增量备份、数据恢复，binlog\_format分类：
   1. STATEMENT：记录 SQL 语句（空间小但不安全）
   2. ROW：记录每行变更（空间大但更精确）
   3. MIXED：自动切换
2. 中继日志（仅从库）：接收主库 binlog 后保存本地，依赖它进行回放
3. 事务日志（InnoDB Redo/Undo Log）
   1. Redo Log持久性、崩溃恢复：将修改写Buffer刷磁盘
   2. Undo Log原子性、MVCC读一致性、回滚：将旧数据写段中刷磁盘

详细讲一下binlog

* binlog 文件记录修改，不会记录查询。
* MySQL在完成一条更新操作后，Server层生成一条binlog
* 事务提交前会将所有 binlog 统一写入 binlog 文件

binlog 有 3 种格式类型

1. STATEMENT：·记录每一条修改数据的 SQL。  
   动态函数问题：比如你用了 uuid 或now 这些函数，你在主库上执行的结果并不是你在从库执行的结果，导致复制的数据不一致
2. ROW：记录行数据最终被修改成什么样。  
   binlog文件过大问题：update 批量更新多少行数据就会产生多少条记录，而在STATEMENT 格式下只会记录一个 update 语句而已；
3. MIXED：根据不同的情况自动使用 ROW 模式和 STATEMENT 模式

主从同步延迟的解决方案？



死锁的产生条件及排查方法

死锁产生四大条件：​

1. ​互斥：资源一次只能被一个事务占用。
2. ​持等：事务持有资源的同时，等待其他资源。
3. ​不夺：已分配的资源不能被强制剥夺，只能由持有者释放。
4. ​循等：事务之间形成资源等待的环路。

**排查方法：​**

1. ​SHOW ENGINE INNODB STATUS;
2. ​查看LATEST DETECTED DEADLOCK：

* 查看涉事务ID
* 分析事务持有的锁和等待的锁
* 定位造成循环等待的SQL语句

1. ​调整锁顺序、事务粒度、重试机制

覆盖索引和索引下推的区别？

覆盖索引**：​**查询字段都包含在索引中，无需回表

索引下推**：​**在引擎层利用索引过滤数据，减少回表，适用于WHERE包含索引和非索引的查询



miChange Buffer对写操作的优化原理？

如果数据库直接修改磁盘上的索引页，那么需要：将索引页从磁盘加载到内存，在内存中修改数据，最后写回磁盘。

当修改非唯一 二级索引时，先将修改记录到Change Buffer中，当相关索引页被加载到内存时，再将Change Buffer中的修改合并到索引页。

适用于写多读少

在高并发、大数据量场景下，为什么要分库分表？分库分表后怎么保证性能？TPS、QPS指标怎么看？

**在系统规模扩大、单库单表无法支撑时，需要分库分表：**

* **分库：将不同的数据分配到不同的数据库上，减轻单个数据库压力，提升读写能力。**
* **分表：将单个表的数据拆成多个表，比如按时间、用户ID取模，提升查询性能，减少单表热点。**

**TPS（Transactions Per Second）：每秒完成的事务数**

**QPS（Queries Per Second）：每秒完成的查询次数**

**性能保障措施：**

* **负载均衡分散压力**
* **保持分库分表后的事务处理正确性（如通过分布式事务/最终一致性）**
* **读写分离（主从同步）**
* **连接池管理，防止连接爆炸**
* **合理设计分片策略（比如hash取模、范围分片）**

MySQL中的B+树索引结构是怎样的？为什么使用B+树而不是其他结构？

**B+树是MySQL InnoDB引擎默认采用的数据结构。**

**它是多路平衡搜索树，所有数据存在叶子节点上，叶子节点形成有序链表，方便区间查询。**

**原因：**

* **减少IO次数：磁盘IO很慢，B+树通过高阶分支（一个节点可以挂载上百子节点）大大减少访问层数。**
* **范围查询友好：叶子节点链表顺序存储，适合范围检索。**
* **稳定性好：增删改查时树高变化不大，维护成本低。**

聚簇索引和二级索引的区别？

**聚簇索引（Clustered Index）：主键索引，数据和索引存储在一起，叶子节点存放的是整行数据。**

**二级索引（Secondary Index）：又称辅助索引，叶子节点存储的是主键值，需要回表查询。节省空间，查询时如果二级索引列不能满足查询，需要通过主键再回到聚簇索引查原数据。**

为什么需要二级索引？

**主键不一定能满足所有查询需求。**

**通过二级索引，可以加速根据非主键字段的查询，比如根据邮箱查用户。**

什么是覆盖索引？覆盖索引有什么问题？请举个例子。

**覆盖索引：查询的列全部在二级索引中就能被满足，不需要回表查聚簇索引。**

**优点：**

* **减少IO开销（避免回表）**
* **查询速度快**

**问题：**

* **如果二级索引列很多，占用空间大**
* **如果列变动频繁，维护成本高**

**SELECT email FROM users WHERE email = 'xxx@xxx.com';**

**假设 email 上有二级索引，这条查询就是覆盖索引，不需要回表。**

什么字段适合当做主键？

* 唯一性，且不能为空，自增字段
* 如果字段的值是随机无序的，可能会引发页分裂的问题，造型性能影响
* 分布式 id

Redis

？？？其他NoSQL数据库

？？？你只用Redis做一些数据存储相关工作吗

Redis数据监听

Redis单线程模型为什么能高效处理请求？

1. ​**内存**
2. ​**跳表**
3. **单线程 事件驱动模型 + I/O多路复用**

缓存穿透/击穿/雪崩的解决方案？

缓存穿透不存在：查询一个不存在的数据，缓存中没有相应的记录，每次请求都会去数据库查询，造成数据库负载激增。

* 布隆过滤器
* 缓存不存在标识

Redis缓存击穿热点：某个热点数据在缓存中过期，导致大量请求同时访问数据库，造成数据库负载激增。

* 互斥锁
* 热点数据永不过期

Redis 缓存雪崩多个：多个缓存数据在同一时间过期，导致大量请求同时访问数据库，造成数据库负载激增。

* 双缓存策略：主缓存设置短过期时间，备份缓存设置较长过期时间
* 随机过期时间

RDB和AOF的优缺点及混合持久化机制？

RDB快照：更快、文件体积小（便于传输）

AOF追加写：更完整、文本格式（便于人工检查修复）

Redis7混合持久化机制：先用RDB生成快照，然后追加AOF日志

Redis Cluster的slot分配算法？

Redis Cluster将整个键空间划分为16384个哈希槽，每个槽对应一个范围。

分配算法**：​**

1. ​对每个键计算CRC16哈希值，然后对16384取模，得到对应的slot编号。
   1. **适配65536的1/4，让哈希值均匀映射到槽**
   2. 优化心跳包的网络传输（16384b大约2KB）在节点之间传递关于哈希槽分配的信息时，用较少的比特位就可以表示一个哈希槽的状态
2. ​每个节点负责一部分slot，节点增减时重新分配，确保负载均衡。

大Key问题

1. **拆分**
   1. **Q：现在redis中有一个大key，如何平滑拆分？**
   2. **A：双写：大的redis key正常写入，同时还要写一个拆分后的。在某一个时机两边数据能够对齐的情况下，从大 Key迁移到新Key上。**
2. gzip**压缩**
3. 冷数据存储到**磁盘**

热Key问题

1. ​**拆分**到多个节点：通过哈希算法或引入随机前缀。
2. **缓存**：CDN或本地缓存
3. ​**读写分离**：将读请求分散到从节点，减轻主节点压力。
4. ​**限流降级**

布隆过滤器

布隆过滤器是一个初始全0的m位数组、k个哈希函数、n个元素映射置1。

查询元素时任意位置为0一定不存在



Redis事务和Lua脚本的原子性区别？

Redis事务

* 通过**MULTI、EXEC、DISCARD**批量执行
* 事务执行期间其他客户端命令可能插入执行
* 如果某条命令失败后续命令仍会执行，**不支持回滚**

Lua脚本​

* 通过EVAL或EVALSHA真正完全原子
* 执行期间其他客户端命令会被阻塞
* 如果Lua 脚本中有错误直接抛出错误不会执行，**隐式“回滚”**

Redis Cluster

Redis Cluster适合中小规模 Redis集群（几十个节点），不适合超大规模集群。

**原因：**

采用去中心化的**流言协议Gossip**传播集群配置的变化，规模越大、速度越慢。

**解决1：**

在客户端和 Redis 节点之间，还需要增加一层代理服务（开源的 Redis 集群方案 twemproxy 和 Codis）

1. 转发请求和响应
2. 监控所有节点，及时进行主从切换
3. 维护集群的元数据（主从信息、槽和节点关系映射表）

**解决2：**把寻址功能前移到客户端中去，客户端在发起请求之前，先去查询元数据，就可以知道要访问的是哪个分片和哪个节点，然后直连对应的 Redis 节点访问数据。（JAVA 客户端 Jedis）

阿里云云数据库 Redis 版：

每个分片的 QPS 大概在 8-10万，记住是每个分片，不是整个集群！

阻塞、非阻塞、多路复用、异步I/O

I/O 是分为两个过程的：

1. 数据准备的过程（端出菜来）
2. 数据从内核空间拷贝到用户进程缓冲区的过程（打进饭盒）
3. **阻塞 I/O** 好比，你去饭堂吃饭，但是饭堂的菜还没做好，然后你就一直在那里等啊等，等了好长一段时间终于等到饭堂阿姨把菜端了出来（数据准备的过程），但是你还得继续等阿姨把菜（内核空间）打到你的饭盒里（用户空间），经历完这两个过程，你才可以离开。
4. **非阻塞 I/O** 好比，你去了饭堂，问阿姨菜做好了没有，阿姨告诉你没，你就离开了，过几十分钟，你又来饭堂问阿姨，阿姨说做好了，于是阿姨帮你把菜打到你的饭盒里，这个过程你是得等待的。
5. **基于非阻塞的 I/O 多路复用**好比，你去饭堂吃饭，发现有一排窗口，饭堂阿姨告诉你这些窗口都还没做好菜，等做好了再通知你，于是等啊等（select 调用中），过了一会阿姨通知你菜做好了，但是不知道哪个窗口的菜做好了，你自己看吧。于是你只能一个一个窗口去确认，后面发现 5 号窗口菜做好了，于是你让 5 号窗口的阿姨帮你打菜到饭盒里，这个**打菜的过程你是要等待**的，虽然时间不长。打完菜后，你自然就可以离开了。
6. **异步 I/O** 好比，你让饭堂阿姨将菜做好并把菜打到饭盒里后，把饭盒送到你面前，整个过程你都不需要任何等待。

Java并发

CAS的原理是什么？存在哪些缺陷？

**CAS（Compare And Swap）**由 CPU 硬件指令保证原子性的无锁并发​：

比较当前值（V）与预期值（E），相等则将新值（N）写入，否则不操作。

1. ​ABA 问题：值从 A 变为 B 再变回 A，CAS 无法感知中间变化

* ​解决：版本号或时间戳。  
  AtomicStampedReference<Integer> atomicRef  
  Integer value = atomicRef.get(stampHolder);获取值和版本号  
  atomicRef.compareAndSet

1. ​自旋开销：如果 CAS 失败，线程会不断自旋重试、消耗 CPU

* ​解决：限制自旋次数。  
  while (!updated && retries < MAX\_RETRIES)  
  updated = counter.compareAndSet(expected, newValue);

1. ​单变量限制：无法直接支持多个变量的原子操作。

* ​解决：封装为Pair对象。

AQS的核心机制是什么？举一个基于AQS实现的工具类。

AQS（Abstract Queued Synchronizer）抽象队列锁

1. ​通过 state表示**资源状态（ReentrantLock通过 state 记录锁的重入次数）**
2. 在​CLH 队列中，线程按请求顺序排队，后面请求锁的线程会自旋等待，直到它前面的线程释放锁，保证公平性。线程通过 acquire() 和 release() 方法进入或离开队列。提供 tryAcquire() 和 tryRelease() 等模板方法，由子类实现具体逻辑。

ConcurrentHashMap如何实现线程安全？Java7和Java8的区别？

Java7 分段锁（默认16个段）：锁粒度大（段级别），性能低

Java8 CAS + synchronized：锁粒度小（桶级别），性能高

synchronized和锁（ReentrantLock）的区别？

**synchronized**：

JVM内置优化，适合低竞争

隐式锁 自动获取释放

**ReentrantLock**：

基于抽象队列锁AQS，适合高竞争

显式锁 手动获取释放

支持可中断锁、尝试锁等特性，

提供 tryAcquire() 和 tryRelease() 等模板方法，由子类实现具体逻辑。

Synchronized实现

依赖于监视器锁Monitor和 对象头Object Header

* Mark Word：
* 未锁定
* 偏向锁
  + 当一个线程第一次获取锁时，JVM 会将该线程标记为“偏向”状态，后续若该线程再获取该锁，几乎没有开销。
* 轻量级锁
  + 当另一个线程尝试获取已经被偏向的锁时，锁会升级为轻量级锁，使用 CAS 操作来减少锁竞争的开销。
* 重量级锁
  + 当 CAS 失败无法获取锁，锁会升级为重量级锁，线程会被挂起，直到锁被释放。
* Klass Pointer：指向对象的类型元数据，帮助 JVM 确定对象的类型信息。

**synchronized 修饰方法**：会增加一个 ACC\_SYNCHRONIZED 标志。每当一个线程访问该方法时，会检查方法的访问标志。如果包含标志，线程必须先获得该方法对应的对象锁，然后才能执行该方法，从而保证方法的同步性。

**synchronized 修饰代码块**：插入 monitorenter 和 monitorexit 字节码指令加锁解锁

SpringBoot/SSM

Spring AOP的实现原理？JDK动态代理和CGLIB的区别？

Spring AOP​通过动态代理实现

* JDK动态代理​
  + 基于接口、通过 Proxy 类生成代理对象
  + 无法代理未实现接口的类
* CGLIB代理​
  + 基于继承、通过字节码生成子类代理
  + 无法代理 final 类或方法

如何解决循环依赖问题？三级缓存机制详解。

循环依赖**：**两个或多个Bean相互依赖，导致无法完成初始化。

三级缓存**：**

1. ​一级缓存（Singleton Objects Map 单例对象映射表）：  
   存放已经实例化、属性注入、初始化的 Bean。
2. 二级缓存（Early Singleton Objects Map 早期单例对象映射表）：  
   存放已实例化，但未完成属性注入和初始化的 Bean。  
   防止 AOP每次调用ObjectFactory.getObject()都产生新代理对象。
3. 三级缓存（Singleton Factories Map 单例工厂映射表）：  
   ObjectFactory的getObject()调用getEarlyBeanReference()创建早期Bean。

解决循环依赖的流程**：**

1. ​**Bean实例化**：

* 创建Bean实例，但未注入属性和初始化。
* 将Bean工厂对象（ObjectFactory）放入三级缓存。

1. ​**属性注入**：

* 发现依赖的Bean，尝试从一级缓存获取。
* 如果一级缓存不存在，从二级缓存获取。
* 如果二级缓存不存在，从三级缓存获取工厂对象，将这个 ObjectFactory 从三级缓存中移除，生成的Bean并放入二级缓存。

1. ​**Bean初始化**：

* 完成属性注入和初始化。
* 将Bean从二级缓存移除，放入一级缓存。

Spring Bean的生命周期和作用域有哪些？

**Bean生命周期：**

1. ​实例化
2. ​属性赋值
3. ​初始化

* ​**前置处理**postProcessBeforeInitialization()。
* ​Initialization**()**：
  + @PostConstruct注解方法
  + InitializingBean的afterPropertiesSet()
  + init-method
* ​**后置处理**postProcessAfterInitialization()

1. ​使用
2. ​销毁

* @PreDestroy注解方法
* DisposableBean的destroy()
* destroy-method

**Bean的作用域：**

Singleton 默认单例

Prototype 每次请求创建一个新的Bean实例。

Request 每个HTTP请求创建一个Bean实例

Session 每个HTTP会话创建一个Bean实例

Application 每个ServletContext生命周期内创建一个Bean实例

WebSocket 每个WebSocket会话创建一个Bean实例

SpringMVC请求处理流程？

1. DispatcherServlet接收请求
2. HandlerMapping根据请求URL找到对应Controller
3. HandlerAdapter适配并调用处理器、通过handle()执行业务逻辑
4. Controller 执行业务逻辑，返回ModelAndView或数据对象
5. HandlerAdapter将ModelAndView或数据对象转换为适合的格式。
6. ViewResolver视图解析器解析
7. DispatcherServlet将渲染视图或JSON数据返回给客户端。

SpringBoot自动配置的实现？

1. ​@SpringBootApplication =   
    @SpringBootConfiguration

+ @EnableAutoConfiguration // 关键：开启自动配置

+ @ComponentScan  
Spring Boot启动时ImportSelector加载spring.factories 中的自动配置类

1. ​条件匹配@Conditional：  
   根据类路径、Bean是否存在判断是否加载配置类。
   1. @ConditionalOnClass
   2. @ConditionalOnMissingBean

Spring事务失效的常见场景有哪些？

1. ​异常不匹配  
   默认只回滚RuntimeException和Error，如果抛出比如IOException且未配置rollbackFor，不会回滚。
2. ​异常未抛出
3. ​同类方法调用  
   同一个类中，方法A调用方法B（B有@Transactional），实际调用的是原始对象的方法，而不是代理对象的方法，而事务管理是通过代理机制实现的，B的事务不会生效。
4. ​非public方法、final方法、static 方法
5. ​事务基于ThreadLocal实现，多线程事务无法传递。
6. ​其他
   * **事务传播行为**PROPAGATION\_NOT\_SUPPORTED会挂起事务
   * 未正确配置数据源DataSourceTransactionManager
   * MyISAM引擎本身不支持事务

MyBatis的#和$的区别及SQL注入问题。

* #：预编译、更安全
* $：直接拼接、存在SQL注入风险

动态表名、排序字段必须用$：

* 因为表名、字段名、ORDER BY 排序字段不能使用 ? 作为参数绑定。
* 只有值可以用 # 作为参数占位符。

严格控制用户输入，确保只能传递合法的表名和字段名

* 白名单校验
* 使用 Enum 限制可选字段

SpringBoot如何集成Tomcat容器？

默认内嵌Tomcat容器

* **端口**：8080
* **最大1w连接**：有1w客户端可以TCP连接到服务器
* **最大200线程：**服务器可以同时处理200个请求

计算机网络

TCP三次握手的详细过程和状态变化？

三次握手

* 第一次: 客户端发出同步请求报文。
* 第二次：服务器发出同步确认报文。
* 第三次: 客户端发出确认报文。



四次挥手

* 第一次：客户端发出释放报文。
* 第二次：服务器发出确认报文。
* 第三次：服务器发出释放确认报文。
* 第四次：客户端发出确认报文。



**TIME\_WAIT状态的作用：​**

1. 防止旧连接的延迟数据包干扰新连接。
2. 保证对端收到最后的ACK，避免连接关闭异常。

**TIME\_WAIT过多的解决方案：​**

1. 开启TCP 连接复用：sudo sysctl -w net.ipv4.tcp\_tw\_reuse=1
2. 增加可用端口：sudo sysctl -w net.ipv4.ip\_local\_port\_range="32768 60999"
3. 使用连接池复用连接
4. 减少短连接的使用

Sudo: Superuser Do

Sysctl: System Control

TCP快速重传和超时重传的区别？

* 快速重传：收到3个重复ACK快速重传，适用部分包丢失但网络较好
* 超时重传：未收到ACK且计时器超时重传，适用严重丢包或连接中断

TCP拥塞控制？



MSS：**Maximum Segment Size**

Ssthresh：**slow start threshold**

HTTPS和HTTP有哪些区别？

**1**. 端口号

* **HTTP**：80
* **HTTPS**：443

2. 加密

* **HTTP**
* **HTTPS**：通过SSL/TLS协议加密

HTTPS工作原理是什么？它是如何实现数据加密的？

SSL/TLS: 对称加密、非对称加密、哈希算法的结合

1. 客户端Hello：SSL/TLS版本、**加密算法列表**、**随机数**
2. 服务器Hello：SSL/TLS版本、**加密算法、随机数**
3. 服务器发送**公钥和数字证书**
4. 客户端验证证书，并生成会话密钥使用服务器公钥加密发送
5. 服务器使用私钥解密获得**会话密钥**。
6. 交换握手消息摘要、开始加密通信。

HTTP1.1新特性？

1. 持久连接keep-alive复用同一连接（串行请求响应）

2. 支持请求管道化：同时发送多个请求

3. 更细粒度的缓存控制：新增Cache-Control头部

* no-cache：必须向服务器验证后才能使用缓存。
* no-store：不允许缓存（敏感数据）。
* max-age=60：缓存内容在 60 秒内有效。
* public：响应可以被任何缓存（比如代理服务器）保存。
* private：响应只能被单个用户缓存，不能被共享缓存（代理）保存。

4. Host头部：允许同一IP托管多个域名  
这样即使同一个服务器只有一个 IP 地址，它也可以根据 Host 头部区分请求，托管多个不同的网站（比如 example.com 和 test.com 都指向同一个 IP，但服务器能正确响应不同内容）。

5. 新增错误代码：410资源已永久删除

HTTP2新特性？

1. 多路复用 应用层消除队头阻塞（并行请求响应）

2. 二进制协议 以帧传输

* **1.1基于文本**

3. 独立流量控制

* **1.1依赖TCP的拥塞控制**

4. HPACK算法头部压缩

* **1.1请求头冗长，尤其对于重复的请求头，如Cookies等**

1. 服务器主动推送

HTTP3新特性？

1. 基于QUIC协议

* **1和 2**依赖TCP协议
* Quick UDP Internet Connections是Google提出，建立在UDP上

2. 零RTT连接建立

* 往返时延（Round-Trip Time, RTT）
* **1和2**基于TCP的三次握手
* **3**首次请求时可以立即开始数据传输

3. 传输层 消除队头阻塞

* **2**虽然支持多路复用，但仍然受TCP队头阻塞的影响：当一个流的数据包丢失时，整个连接的延迟会增加，影响其他流的传输。
* QUIC基于UDP、采用多路复用 + 独立流量控制和快速重传

4.内建TLS加密

操作系统

进程和线程的区别？协程的优势是什么？

进程：

* 资源分配基本单位
* 进程创建和切换开销大：有独立的代码、数据、文件等，
* 进程之间是相互独立的，需要**进程间通信机制**（管道、消息队列、共享内存、套接字）

线程：

* CPU调度基本单位
* 线程创建和切换开销小：组内共享资源，但有独立的PC和堆栈，
* 线程直接读写内存即可，但需要**同步机制**

Select、poll、epoll的区别？

1. ​**select**：

* 使用 file descriptor set ，支持的文件描述符数量1024。
* 每次调用需要遍历所有文件描述符，效率低。时间复杂度：O(n)。

1. ​**poll**：

* 使用 ​pollfd 数组，支持的文件描述符数量无限。
* 每次调用需要遍历所有文件描述符，效率低。时间复杂度：O(n)。

1. ​**epoll**：

* 使用事件驱动机制，只关注活跃的文件描述符，效率高。
* 支持边缘触发（ET）和水平触发（LT）模式。时间复杂度：O(1)。  
  边缘触发模式只在状态变化时通知，需一次性处理所有数据，否则会丢失事件。适用高并发网络编程。

用户态和内核态切换的开销来源？

1. ​寄存器、程序计数器状态
2. ​权限**切换**
3. ​缓存**失效**
4. ​系统调用需要内核验证
5. ​切换可能伴随中断处理

进程间通信 共享内存和消息队列的对比？

**消息队列**：

* ​内核维护消息队列，进程发送/接收消息通信
* 自带同步机制
* ​有数据拷贝 速度慢，适用小数据量异步通信

**共享内存**：

* ​多个进程映射同一块物理内存
* 需要同步机制（如信号量）避免竞争。
* ​无数据拷贝 速度快，适用高性能大数据量通信。

零拷贝技术（sendfile）的实现原理？

零拷贝技术（sendfile）：在文件传输中，让内核直接将文件数据从磁盘读取到网络缓冲区，无需经过用户态，避免数据在用户态和内核态之间的拷贝，提升性能。

* 传统方式：磁盘 -> 内核缓冲区 -> 用户缓冲区 -> 网络缓冲区
* sendfile ：磁盘 -> 网络缓冲区

内存映射（memory map, mmap）系统调用的实现原理？

mmap系统调用将文件/设备映射到进程的虚拟地址空间，实现文件读写或共享内存。减少用户态和内核态的数据拷贝，提高性能。

* ​**创建映射**：内核在进程的虚拟地址空间中分配一段区域，建立虚拟内存与文件/设备的映射关系。
* ​**延迟加载**：映射时不加载文件内容，通过缺页中断按需加载。
* ​**读写操作**：进程通过指针直接访问映射区域，内核负责将数据同步到文件或设备。

虚拟内存的作用？

虚拟内存：扩展、隔离、管理、共享

* ​允许程序使用比物理内存更大的地址空间
* ​每个进程拥有独立的虚拟地址空间，提高安全性
* ​简化内存分配和回收，支持动态内存需求
* ​多个进程可以共享同一块内存区域

Java集合

详细描述HashMap的扩容机制（触发条件、rehash过程）。

**Java7扩容条件：**

元素数量超过阈值 && 发生哈希冲突

**Java8扩容条件：**

1. 情况一：元素数量超过阈值
2. 情况二：链表长度≥8 && 数组长度＜64，优先扩容而不是转为红黑树

Rehash过程**：**

1. 新建一个2倍大小的数组
2. 只需部分移动：if e.hash & oldCap==0
   1. 0接在loTail后面
   2. 1接在hiTail后面

基础：

数组长度是2的次方，且扩容为2倍。

oldCap：

位运算代替取模提高效率、保证hash均匀分布：hash & (数组长度-1)

旧长16即010000，16-1=15即001111

新长32即100000，32-1=31即011111

rehashing时用**旧长16**即010000作为一个Mask看对应该位是否为1

源码**if e.hash & oldCap==0**（0接在loTail后面、1接在hiTail后面）

HashMap底层结构是什么？如何解决哈希冲突？

**HashMap底层结构**是数组+链表，Java8后引入红黑树。

当链表长度≥8且 数组容量≥64时，链表会转换为红黑树  
（链表长度≤8时，链表性能足够且内存占用低）

当红黑树节点数≤6时，红黑树会退化为链表

**解决哈希冲突**：

1. **拉链法**
2. **红黑树**
3. **二次扰动函数：**如果哈希值低位变化较小，计算索引时，低位相同的键会映射到同一位置。**将哈希值高位低位异或、优化索引分布、减少哈希冲突**
4. **动态扩容机制：**
   1. 超过阈值（容量×负载因子，默认0.75）扩容2倍并重新计算索引。
   2. 0.75是空间利用与哈希冲突的最佳平衡  
      负载因子过高会增加哈希冲突  
      **负载因子**过低则导致频繁扩容

为什么HashMap线程不安全？举例说明并发问题场景。

**HashMap线程不安全的原因：**

数组数据覆盖、头插环形链表、红黑树树化结构异常

**解决方案：**

1. Collections.synchronizedMap(new HashMap<>())：在方法级别加锁，性能低
2. ConcurrentHashMap：采用分段锁（7）或CAS+ synchronized（8），性能高

HashMap的遍历方式有哪几种？哪种效率更高？

1. ​entrySet() 效率最高、推荐使用。
2. forEach()+ Lambda 底层实现基于entrySet()。
3. ​keySet() 返回所有键的集合，通过 map.get(key)获取值
4. values() 返回所有值的集合。由于无法通过值获取键

for (Map.Entry<String, Integer> entry : map.entrySet()) {

String key = entry.getKey();

Integer value = entry.getValue();

System.out.println(key + ": " + value);

}

map.forEach((key, value) -> {

System.out.println(key + ": " + value);

});

Java基础

Java的8个基本数据类型

1. byte： 1 字节
2. short： 2 字节
3. int： 4 字节
4. long： 8 字节
5. float： 4 字节
6. double： 8 字节
7. char： 2 字节
8. boolean： 逻辑1位 / 1字节

* 包装类型是类，存储在堆中
  + == 比较的是对象的内存地址，
  + equals() 比较的是对象的值
* **局部变量**在**栈**上
* **成员变量**在**堆**上
* **静态**字段在**方法区**

介绍一下反射的底层原理？怎么获取的？能做哪些处理？

运行时动态创建对象、访问字段、调用方法，无需在编译时知道类的具体信息

Class.forName("com.xxx")

* .getConstructor().newInstance()
* .getField("xxx").setAccessible(true)
* .getMethod("xxx", String.class).invoke(obj, "param")

**注意：**

* **避免过度依赖反射**，因为动态解析和方法调用导致性能开销比直接调用大

**缓存**反射获取的类、方法、字段，减少操作频率

强引用、软引用、弱引用和虚引用

1. **强引用**：只要一个对象有强引用指向它，垃圾回收器永远不会回收该对象，即使系统内存紧张，宁愿抛出 OutOfMemoryError
2. **软引用**：有用但并非必需的对象，当系统内存不足时，垃圾回收器会回收软引用指向的对象，避免内存溢出。通常用于实现缓存机制。
3. **弱引用**：比软引用更弱，无论内存是否充足该对象会立即被回收。常用于防止内存泄漏。
4. **虚引用**：虚引用是最弱。仅有虚引用，那么它与没有引用一样，随时会被垃圾回收。作用是跟踪对象的垃圾回收状态。在对象被回收时，虚引用会被放入一个 ReferenceQueue，我们可以通过这个队列来执行一些清理或其他后续操作。

JVM

JVM内存模型核心区域？

JVM内存分为：

1. 堆：线程共享，存放对象实例，由GC管理，可能发生OOM。
2. 方法区（元空间）：存储类元信息、常量池（JDK8后由元空间实现）。
3. 虚拟机栈：线程私有，存储方法调用的栈帧（局部变量、操作数栈）。
4. 本地方法栈：服务于Native方法。
5. 程序计数器：记录当前线程执行指令的地址，无OOM。

#### 堆和栈的区别是什么？

堆：存对象实例、对象存活到GC回收、线程共享、堆异常OutOfMemory

栈：存方法调用和局部变量、随方法结束销毁、线程私有、栈异常StackOverflowError

#### Java什么时候会把对象分配在堆内存以外的地方

1. 逃逸分析与标量替换：  
   -XX:+DoEscapeAnalysis逃逸分析发现：  
   （1）对象未逃出方法作用域  
   （2）对象内部字段被拆分为局部变量

效果：  
（1）对象分配在栈上  
（2）不创建对象结构

1. Netty框架显式分配**堆外内存**来规避GC开销

#### 临时变量都是存在栈上的吗

基本数据类型作为局部变量——在栈上

逃逸分析对象不会逃逸出方法，进行标量替换，拆解成基本类型，放在栈上

对象引用（数组、类实例）——引用地址在栈上，实际内容在都堆中

ThreadLocalMap，键是ThreadLocal对象的弱引用，值则是用户设置的对象，这些键值对同样存于堆中

讲一下你熟悉垃圾回收器？

1. Serial GC 单线程GC，是JVM Client模式默认垃圾回收器

**缺点：**全程StopTheWorld

1. Parallel GC 多线程GC，是JDK8默认垃圾回收器

**缺点：**停顿时间较长。

1. ConcurrentMarkSweep GC 最短停顿时间

初始标记、并发标记、重新标记、并发清除

缺点：内存碎片化、CPU敏感  
适用老年代，对CPU敏感（已废弃）。

1. Garbage 1st GC 停顿时间可控、压缩避免碎片，JDK9+用G1替代CMS

划分堆为2048个1~32MB区域，采用Mixed GC模式，通过-XX:MaxGCPauseMillis停顿时间可控  
缺点：内存占用高

1. **ZGC**：亚毫秒级停顿（<10ms），基于颜色指针和读屏障，适用大堆低延迟场景（如金融交易）。

新生代和老年代的占比有了解吗？

新生代**占**堆内存的 1/3**：Eden区、Survivor区 (S0和S1)**

老年代**占**堆内存的 2/3

#### 新生代和老年代占比的调整

-XX:NewRatio=2 表示老年代占总堆内存的 **2/3**，新生代占 **1/3**

#### 如何根据应用场景调整新生代与老年代的比例？

对于短生命周期对象较多的，如Web应用，**增**加新**生代的内存**。

对于长生命周期对象较多的，如大数据处理，**增**加老**年代的内存**。

类加载的过程分为哪几个阶段？

1. **加载**：读取字节码到方法区，生成Class对象。
2. **验证**：检查字节码合法性（如魔数、语法）。
3. **准备**：为静态变量分配内存并赋初始值（0或null）。
4. **解析**：将符号引用转为直接引用。
5. **初始化**：执行静态代码块和变量赋值（触发<clinit>）。

双亲委派模型的工作机制是什么？有什么优缺点？

**机制**：类加载器收到请求后，先委派给父类加载器（Bootstrap→Extension→Application），父类无法完成时自己加载。  
**优点**：避免重复加载核心类，保证安全性（如自定义String类不会被加载）。  
**缺点**：无法解决基础类调用用户代码的场景（如JDBC需用线程上下文类加载器打破委派）。

在项目中遇到过 Full GC 或内存泄漏的情况吗？如何排查和解决的？

**现象：**

* 频繁 Full GC，延迟飙高，JVM 堆使用率长期在 90% 以上；
* 通过日志观察到 Full GC 次数异常（每分钟多次）。

**排查方式：**

1. 打开 GC 日志，使用 GCViewer / GCEasy 分析回收频率和耗时；
2. 使用 jmap -heap 观察内存占用；jmap -histo查看对象数量分布。
3. 使用 jstat, jstack, arthas 等工具进一步分析堆、线程、锁等情况；
4. 使用 MAT 或 VisualVM 分析 dump 的堆文件，定位大对象或泄漏类。

**原因：**

* 线程池中缓存了大量大对象未释放；
* 缓存系统（Guava Cache）使用不当，未配置 TTL 导致内存占满。

**解决：**

1. 优化对象生命周期，及时释放；
2. 对缓存添加合理过期策略；
3. 对频繁创建对象的代码重构为对象池复用；
4. 调整 JVM 参数，增大堆大小并优化 GC 策略（G1 替代 CMS）。

Linux命令

如何用grep查找包含"error"的日志并统计次数？

* grep "error" logfile.log：查找 logfile.log 中包含 "error" 的所有行
* grep -c "error" logfile.log：统计 logfile.log 中包含 "error" 的行数

awk如何实现按列求和？

awk '{sum+=$3} END{print sum}' filename

* {sum+=$3}：逐行累加第3列的值。
* END{print sum}：处理完所有行后，输出累加结果。

如何用sed批量替换文件中的字符串？

sed -i 's/old\_string/new\_string/g' filename

* sed：Stream Editor 流编辑器
* -i：in-place，直接原地修改文件、不输出到终端
* s：**替换命令**，格式为 s/查找内容/替换内容/
* g ：全局替换

strace和perf工具的作用及使用场景？

strace：跟踪进程系统调用

strace -p <pid> 监视进程 <pid> 的所有系统调用。

perf：分析 CPU 性能瓶颈、函数调用链、硬件事件

perf top 实时查看占用 CPU 最高的函数

如何用tcpdump抓取指定端口的SYN包？

tcpdump -i eth0 'tcp[tcpflags] & tcp-syn != 0 and port 80'

* -i eth0：指定网卡接口
* tcp[tcpflags] & tcp-syn != 0：匹配 SYN标志位
* port 80：指定端口

【背调环节】

入职时间？实习多久？

**6月初**能够到岗Landing，至少可以实习**3个月**，可以根据具体情况实习更久

最新技术趋势？

CSDN极客日报

声动早咖啡播客

谷歌学术

AI工具？大模型？智能体？

1. **学习**
   1. Deepseek清华大学104页PPT
      1. 推理模型（DeepSeek-R1）：数学推导、逻辑分析
      2. 通用模型（GPT-4）：文本生成、创意写作
   2. 知识掌握得比人多、需求掌握得比人少（自我判断）
      1. 调试思路
      2. 文档归纳
2. **问题**  
   大模型幻觉：虚构信息、捏造细节（医疗、法律领域敏感）
   1. 学习了错误数据——数据清洗
   2. 基于概率生成，即使它们不正确——检索增强生成（RAG）
   3. 提示词不够具体，导致模型脑补——提示工程
   4. 缺乏真实世界反馈——强化学习 / Man-in-the-loop
3. **应用**
   1. 腾讯云黑客松Copilot
   2. Cursor
   3. Coze
   4. MCP

如何快速学习一门新技术？遇到不会的技术领域问题怎么解决？

1. 明确**紧急程度、**制定**学习目标**
2. 搜罗**书籍 教程**，关注**文档 社区**
3. **渐进式实践学习**
4. **请教**他人、总结**复盘**

未来三年职业规划？

1. **快速Landing、熟悉业务**：理清已有代码职能、熟悉模块调用关系
2. 保持嗅觉、**学习探索改良**
3. 理解把握上级的**工作思路与战略方向**，  
   在团队中发挥**桥梁作用**：对齐目标、沟通协同、执行效率

【反问环节】

* 能谈一谈您对部门**对公司的感受**？
* 我觉得我这次表现一般/不是很好，您可以给一些**评价或建议**吗？
* 这个岗位为什么还在招人，有**实习生离职**还是**缺人手**还是**提供更多机会**？
* 实习生主要工作内容？