目录

【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 **渲染和边缘计算** 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：
  - **用户需求四纵领域**：**娱乐、信息、通信、商务**
  - **技术变革三横方向**：**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
- **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《华为工作法》

- **先瞄准、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》

- **主动**挖掘痛点，而非被动接受需求
- **以终为始 重疗效**，反对形式主义

阿里《阿里巴巴管理三板斧》

- **基层**：定**目标**、追**过程**、拿**结果**
- **中层**：揪**头发**、照**镜子**、闻**味道**
- **高层**：定**战略**、造**土壤**、断事**用人**

小米《小米创业思考》

- 商业本质是**创造普惠价值**，给**最多人**带来**最大化**的**美好幸福感**
- 七字诀“**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 **startPage** 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
   a) .getList(); // 当前页数据
   b) .getPageNum(); // 当前页码：2
   c) .getPageSize(); // 每页记录数：2
   d) .getTotal(); // 总记录数：6
   e) .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**
   a) 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
   b) 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);
4. 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-
2. **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
3. **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

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface AutoFill {
    OperationType value();
}
```

3. 创建**切面类** AutoFillAspect
   通过 AOP 拦截添加了 @AutoFill 注解的方法

```
@Aspect
@Component
@Slf4j
public class AutoFillAspect {

    @Pointcut("execution(* com.sky.mapper.*.*(..)) && @annotation(com.sky.annotation.AutoFill)")
    public void autoFillPointCut() {}
1
    @Before("autoFillPointCut()")
    public void autoFill(JoinPoint joinPoint) {
        log.info("开始进行公共字段自动填充...");

        // 获取方法签名
2       MethodSignature signature = (MethodSignature) joinPoint.getSignature();
        // 获取方法上的注解
        AutoFill autoFill = signature.getMethod().getAnnotation(AutoFill.class);
        // 获取操作类型
        OperationType operationType = autoFill.value();

        // 获取方法参数
        Object[] args = joinPoint.getArgs();
```

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
   a) id // 路由 ID，唯一标识这个路由规则
   b) uri // 路由目标地址，lb://表示使用服务发现（LoadBalancer）
   c) predicates // 路由断言规则，决定哪些请求匹配此路由
   - Path=/api/user/**
   d) filters // 过滤器
      i. 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
   a) value = "getUserInfo"　　// 资源名，标识要保护的服务
   b) blockHandler = "handleBlock"　　// 当触发限流、降级时的兜底处理方法
   c) 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？
  - 用户注册布隆过滤器容量设置以及碰撞率问题
- 手摸手之实现敏感信息加密存储
  - 核心技术文档-如何防止用户敏感数据泄露
- 手摸手之用户敏感信息展示脱敏
- 如何防止用户敏感数据泄露

- 缓存击穿之双重判定锁如何优化性能？（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
4. **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

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

3. 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
4. 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

a) 空
b) **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
3. **查询站点 station 以及城市 region 两张表**，加载城市站点数据
   a) 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）
   b) 缓存中存放键值对，保存**站点**与**城市**的关联关系
   c) 通过站点关联到城市，通过城市查询列车
   d) 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）
   e) 单独存储余票信息，因为余票数据是实时变更的
4. 查询**列车余票信息（座次、价格）**
5. 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。

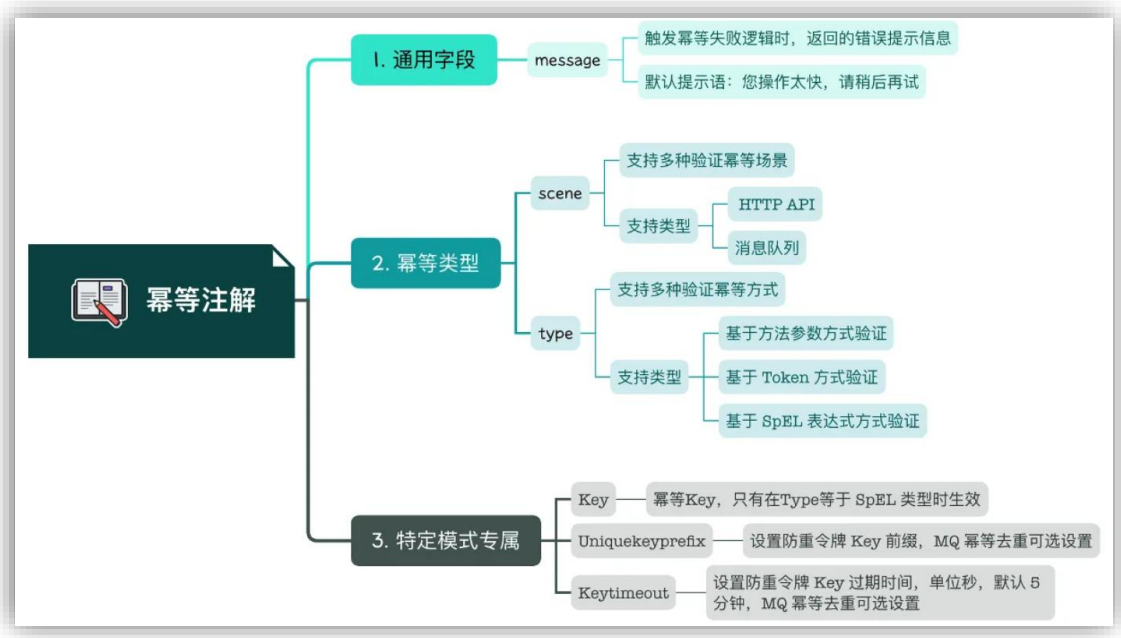
- 存在对应记录，表示请求已经执行过，直接返回而不重复操作
- 不存在对应记录，执行相应业务逻辑，在处理完成后将**请求的唯一标识（如请求 ID 或标识）**添加到 Redis 去重表中

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
   a) 处理逻辑 handler
   b) 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
   a) 提供一个 HashMap 存储责任链业务标识-责任链组件实现
   b) 创建责任链上下文 AbstractChainContext
   c) 调用 AbstractChainContext 的方法执行责任链中的处理器
   d) 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),

```
# 获取 KEYS[1] 对应的 Val
local cliVal = redis.call('get', KEYS[1])
# 判断 KEYS[1] 与 ARGV[1] 是否保持一致
if(cliVal == ARGV[1]) then
  # 删除 KEYS[1]
  redis.call('del', KEYS[1])
  return 'OK'
else
  return nil
end
```

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
   a) 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
   b) 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；
   c) 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
   a) 解析消息的 table、type、data
   b) 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）**
   a) consumeMode = ConsumeMode.CONCURRENTLY 并发消费
   b) 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
- 创建订单并支付后延时关闭订单消息怎么办？
  - 手摸手之消息队列正确使用姿势（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
   a) Redis 5.0 前如果宕机或者重启，这个消息不会重复投递
   b) 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

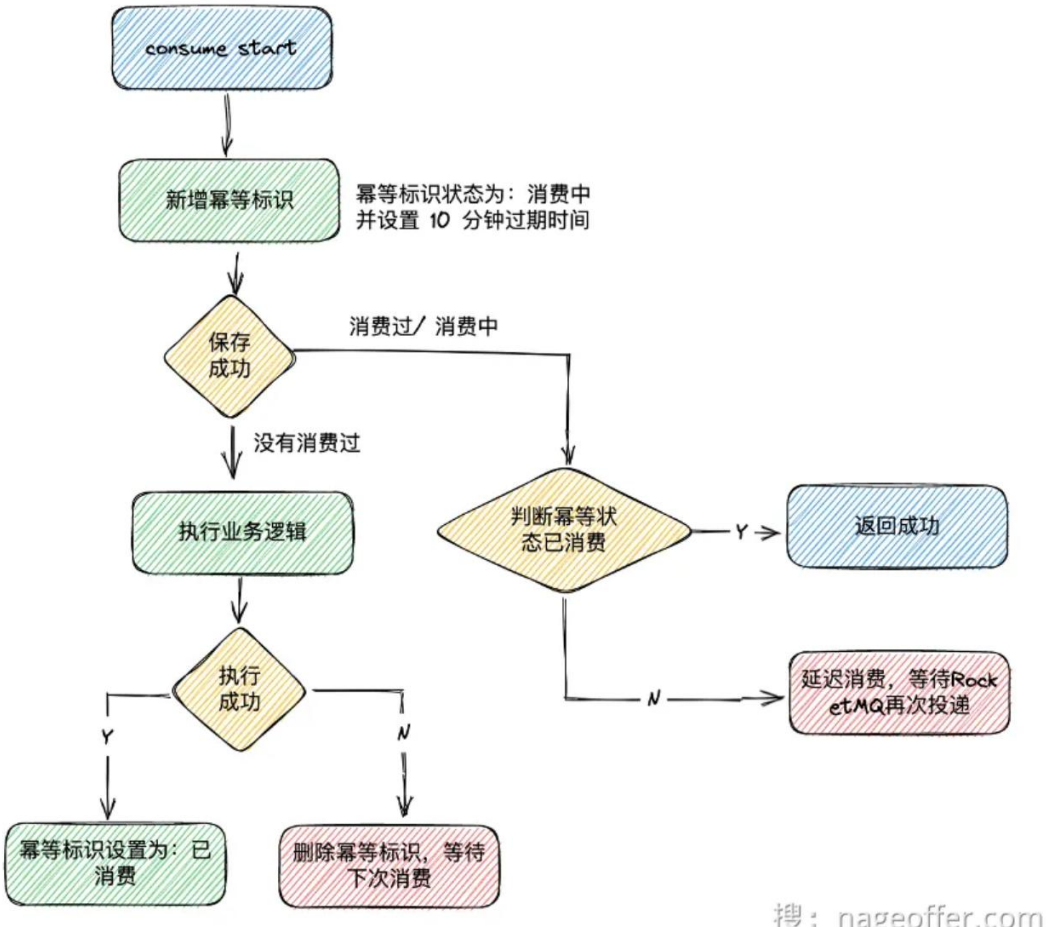
一个 60s 的时间轮，时间轮上会有一个指向当前时间的指针定时地移动到下一个时间（秒级）
在每个时间节点增加一个 round 字段，记录时间轮转动的圈数，比如对于延时 130s 的任务，round 就是 2，放在第 10 个时间刻度的链表中。

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？


consume start
新增幂等标识
幂等标识状态为：消费中
并设置 10 分钟过期时间
保存成功
消费过/ 消费中
没有消费过
执行业务逻辑
判断幂等状态已消费
Y
返回成功
执行成功
N
延迟消费，等待RocketMQ再次投递
Y
N
幂等标识设置为：已消费
删除幂等标识，等待下次消费
搜：nageoffer.com

订单数据分库分表 = 雪花算法 + 基因法

- 用户分库分表
- 乘车人分库分表
- 订单分库分表
- 分布式雪花算法
- 手摸手实现分布式 ID 组件库

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；**
   a) 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；**
   a) 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**
   a) 子查询会先执行一遍，把结果生成出来，再带入外部查询
   b) 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
3. 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
  a) Using index 表示使用覆盖索引
  b) Using where 表示使用 WHERE 过滤、索引下推
  c) Using temporary 表示使用临时表，临时表开销大
  d) 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 MVCC、间隙锁（Next-Key Lock）解决幻读
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

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。
- **min_trx_id：**当前活跃 ID 之中的最小值。
- **max_trx_id：**生成 readView 时即将分给下一个事务的 ID
- **m_ids：**生成 readView 时**已启动但还未提交**的事务 ID 列表。

从新到旧判断可见，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，**可见**
- **< min_trx_id**
  说明修改这条数据的事务在生成 readView 的时候已提交，所以**可见**
- **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生成 readView 的时候还**未启动**，所以**不可见**

链表 红黑树 B 树 B+树

链表

链表查询复杂度 **O(n)**

二叉搜索树、红黑树

保证了二叉搜索树不会退化成链表

- **左根右**：左子树节点值<=根节点值，右子树节点值>=根节点值
- **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路径上黑色节点的数量相同

B 树、B+树

1. **查找性能**：B+树查找、插入、删除等操作的时间复杂度为 **O(log n)**
2. **树高增长**：B+树**非叶节点**存储**索引**和**页面指针**
   a) 可以在内存中存放更多索引页，**减少磁盘 IO**
   b) B+树查询时间更**平均、稳定（黑路同）**
3. **范围查询**：B+ 树**叶子**节点通过**链表**链接，便于**范围查询**

MySQL 三层 B+ 树能存多少数据？

三层 B+ 树的 2000 万条记录：

- 第三层**叶子节点页**：存储 16KB ÷ 1KB = 16 条数据记录
  - **默认页大小** 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）
- 第二层**中间节点页**：指向 16 * 1024 ÷（6+8） = 1170 个叶子节点
  - 索引键（BIGINT）8 B + 每个页地址指针 6 B
- 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 万是个性能瓶颈，单表数据量超过 1000 万~2000 万

- **层数增加**，查询路径变长
- **页分裂频繁**，写入开销变大
  - 申请新页、移动数据、修改索引指针，增加 CPU 和 I/O 开销
  - 产生的新页可能不连续、在磁盘上分布零散，增加磁盘寻道时间
- **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？

- 加**行锁**（锁的是叶子节点对应的**索引项**）
- 加**意向锁**（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：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：并发高、细粒度控制

- 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
- **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

1. 行锁与行锁的兼容性

当前锁 \ 请求锁	共享锁（S）	排他锁（X）
共享锁（S）	✅ 兼容	❌ 冲突
排他锁（X）	❌ 冲突	❌ 冲突

2. 表锁与意向锁的兼容性

当前锁 \ 请求锁	表级共享锁（S）	表级排他锁（X）
意向共享锁（IS）	✅ 兼容	❌ 冲突
意向排他锁（IX）	✅ 兼容	❌ 冲突
表级共享锁（S）	✅ 兼容	❌ 冲突
表级排他锁（X）	❌ 冲突	❌ 冲突

意向锁是表锁的前置标记：
意向锁本身是表级锁，但不阻止任何操作，仅用于声明“表中某些行已被锁定”。

意向锁与行锁自动关联：
当事务获取行锁时，数据库自动在表级别添加对应的意向锁

意向锁的核心作用：协调表锁与行锁的冲突

- 事务 A 对 users 表的 id=1 行加行锁（排他 X）。
- 事务 B 尝试对整张表加表锁（如 LOCK TABLES users WRITE）。
- 若无意向锁，事务 B 需遍历全量数据检查是否存在行锁，效率极低。
- 事务 A 加行锁时，表级已有意向排他锁（IX）。
- 事务 B 请求表锁时，直接检查到 IX 锁，立即知道“表中存在行锁”，无需遍历数据，避免死锁或长时间等待。

分库分表后全局 ID 生成方案有哪些？

1. **UUID**：
   无序
   UUID.randomUUID().toString(); e3e70682-f3a1-11e9-8f0b-362b9e155667
2. **Snowflake**：
   时钟回拨问题
   0 | 41-bit 时间戳 | 10-bit 机器 ID | 12-bit 序列号
3. **数据库自增 ID**：
   SET @@auto_increment_**offset** 起始值
   SET @@auto_increment_**increment** 步长
4. **Redis 自增 ID**：
   单点故障风险
   jedis.incr("global:order_id")
5. **号段模式**：
   从数据库预分配一批 ID，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

1. **写入 Binlog**：先写入 binlog、再提交事务 更新数据
2. **同步 Binlog**：连接主库的 **log dump** 线程、接收 binlog 写入 **relay log**
3. **回放 Binlog**：回放 binlog 更新数据

主从同步延迟的解决方案

- **二次查询**：如果从库查不到数据，则再去主库查一遍
  如果有不法分子故意查询必定查不到的查询，对主库产生冲击
- **写后立读**走主库
- **关键业务**走主库（用户注册）、**非关键**读写分离
- **主库写入后同步到缓存**
  引入了数据一致性问题

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';
3. **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';
4. **二进制**日志：用于**主从复制、增量备份、数据恢复**
   binlog_format 分类：
   a) **STATEMENT**：记录 SQL 语句，空间小但不安全
   b) **ROW**：记录每行变更，空间大但更精确
   c) **MIXED**：自动切换
5. **中继**日志（**从库**）：接收主库 binlog 后保存本地，依赖它进行回放
6. **事务**日志（InnoDB **Redo/Undo Log**）
   a) **Redo Log** 保证持久性、崩溃恢复：将修改 写 Buffer 刷磁盘
   b) **Undo Log** 保证原子性、MVCC 回滚：将旧数据 写段 刷磁盘

详细讲一下 binlog

- binlog **记修改**、不记查询
- 更新操作后 **Server 层**生成一条 binlog
- **事务提交前**会将所有 binlog 统一写入文件

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；
3. **MIXED**：根据不同的情况自动使用

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**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**
3. 调整**锁顺序、事务粒度、重试机制**

Change 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**：
  磁盘—内存（修改）—磁盘
- 当在**写多读少**的情况下，修改**非唯一 二级索引**时：
  先将修改记录到 Change Buffer 中，**索引页被加载**到内存时，再将 Change Buffer 中的修改合并到索引页

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？

分库分表原因

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**

性能保障

1. **合理的分片策略**（比如 hash 取模、范围分片）
2. 通过**分布式事务**保持**事务处理正确性**
3. **负载均衡、读写分离、连接池管理**

TPS、QPS

- TPS（**Transactions** Per Second）：　每秒完成的**事务数**
- QPS（**Queries** Per Second）：　每秒完成的**查询数**

聚簇索引和二级索引的区别？

聚簇索引（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（辅助索引）：叶子节点存**主键值**，需要**回表查询**

什么字段适合当做主键？

唯一、非空、自增（无序引发页分裂）

为什么需要二级索引？

1. **主键不一定能满足**所有查询需求
2. 通过**二级索引加速**非主键字段的查询（比如**根据邮箱查用户**）

覆盖索引和索引下推的区别？

覆盖索引：

查询字段**都包含**在二级索引中，**无需回表**查聚簇索引
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

索引下推（适用于 WHERE 包含索引和非索引的查询）：

引擎层用索引**过滤**数据，符合条件才传回 **Server 层**，**减少**回表

- `name`、`age`是索引字段，`city`不是索引字段。
- 如果**不开启索引下推**：
  - 引擎层只能用`name='Alice' and age=25`筛选出一批**可能符合条件的**记录（注意 city 这一条件引擎层不会管），
  - 这些记录会**传回Server层**，然后Server层再根据`city='Shanghai'`这个条件进一步过滤。
- 如果**开启索引下推（ICP）**：
  - 引擎层在走`idx_name_age`索引扫描的时候，就直接把`city='Shanghai'`这个条件也一起应用了！
  - 只有符合所有条件的数据才传回Server层，减少了回表和数据传输。

Redis

你还了解哪些其他 NoSQL 数据库？

1. **键值存储 Memcached**：内存缓存数据库，读写速度快，但无法持久化
2. **文档存储 MongoDB**：以 BSON（类 JSON）格式存储数据，支持丰富的查询语法和索引，适合存储日志、用户信息
3. **列存储 Cassandra**：源于 Facebook，适合海量数据分布式存储（日志）
4. **图存储 Neo4j**：擅长处理复杂关联关系（如社交网络、知识图谱）

Redis 除了数据存储还能做什么？

1. **缓存**热点数据（如用户信息、商品详情）
2. **分布式锁** SET key value NX PX（如库存扣减）
3. 轻量级**消息队列**
   - 基于 List 数据结构（如 LPUSH/RPOP）实现简单队列，支持生产者 - 消费者模型，适用于异步任务处理（如日志收集、订单异步通知）。
   - 配合 Stream 数据结构（Redis 5.0+）可实现更完善的消息队列功能，支持消息持久化、分组消费和回溯。
4. 利用 INCR 命令实现**计数器**（如点赞数、接口调用量统计），保证原子性。
5. 基于 ZSET 有序集合实现实时**排行榜**

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**：查询不存在的数据，缓存没有记录，每次都去数据库查询

- 缓存不存在标识
- **布隆过滤器**

缓存**击穿热点**：热点数据在缓存中过期，导致大量请求同时访问数据库

- 热点数据永不过期
- **互斥锁**

缓存**雪崩多个**：多个缓存同一时间过期，导致大量请求同时访问数据库

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

RDB 和 AOF 的优缺点及混合持久化机制？

RDB 快照： **速度快** 体积**小**，便于传输
AOF 追加写： **完整** **文本**格式，便于人工检查修复
混合持久化（Redis7）：先用 RDB 生成快照，然后追加 AOF 日志

大 Key 问题

1. **拆分**
2. gzip **压缩**
3. 冷数据存储到**磁盘**

热 Key 问题

1. **拆分**：通过**哈希算法**或引入**随机前缀**
2. CDN 或本地**缓存**
3. **读写分离**
4. **限流降级**

布隆过滤器

m 位数组、k 个哈希函数、n 个元素
查询元素时任意位置为 0 一定不存在

布隆过滤器的误差率，即误判率，其计算公式为：$p \approx (1 - e^{-kn/m})^k$，其中$k$是哈希函数的个数，$n$是插入元素的个数，$m$是布隆过滤器的位数组大小。

例如，假设有一个布隆过滤器，位数组大小$m = 1000$，插入了$n = 100$个元素，使用$k = 5$个哈希函数。

首先计算$e^{-kn/m}$，即$e^{-5\times100/1000} = e^{-0.5} \approx 0.6065$。

然后计算$1 - e^{-kn/m}$，即$1 - 0.6065 = 0.3935$。

再计算$(1 - e^{-kn/m})^k$，即$0.3935^5 \approx 0.009$。

所以，这个布隆过滤器的误差率约为0.9%。

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

阿里云云数据库 Redis 版：
每个分片的 QPS 大概在 10 万，记住是每个分片，不是整个集群！

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

阻塞、非阻塞、多路复用、异步 I/O

I/O 分为两个过程：

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**）

1. **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。
2. **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（用户空间）**，这个过程你是得等待的。
3. **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。
4. **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）**CPU 硬件指令**保证**原子性无锁并发**：
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**： 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
2. **自旋开销**：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);
3. **单变量限制**：无法直接支持多变量原子操作
   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 记录重入次数）**
2. 在 **CLH 队列**中，线程通过 **acquire() 和 release() 按请求顺序排队，**
   后请求的线程**自旋等待**，直到前面的线程**释放锁**，保证公平性。

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，适合**低竞争**
  隐式锁　自动获取释放
- **ReentrantLock**：
  基于 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**：
    一个线程第一次获取锁时，标记为偏向锁，后续再获取几乎没有开销
  - **轻量级锁**：
    另一线程尝试获取偏向锁，升级为轻量级锁，用 **CAS** 来减少开销
  - **重量级锁**：
    CAS 失败升级为重量级锁，**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：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，但未属性注入和未初始化**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. **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。
2. **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。
3. **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**
   - postProcess**BeforeInitialization()**
   - **Initialization()**
     - @PostConstruct 注解方法
     - InitializingBean 的 afterPropertiesSet()
     - init-method
   - postProcess**AfterInitialization**()
4. **使用**
5. **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. Handler**Mapping**　　根据请求 URL 找到对应 Controller
3. Handler**Adapter**　　**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. Handler**Adapter**　　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**
    1. @SpringBootConfiguration
    2. @EnableAutoConfiguration
    3. @ComponentScan：　ImportSelector 加载 spring.factories 中的自动配置类
2. **条件匹配@Conditional**
    1. @Conditional**OnClass**
    2. @Conditional**OnMissingBean**

Spring 事务失效的常见场景有哪些？

1. **异常未抛出**
2. **异常不匹配**
   默认只回滚 **RuntimeException 和 Error**
   如果抛出 *IOException* 且未配置 *rollbackFor*，不会回滚
3. **同类方法调用**
   同一个类的方法 A 调用有@Transactional 的方法 B
   实际调用的是原始对象的方法，而不是代理对象的方法
   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. 因为事务基于 **ThreadLocal** 实现，所以**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
- $：**直接拼接**、存在 **SQL 注入风险**、**动态表名、排序字段必须用$**：
  - 为了严格控制用户输入，**只有值可以用 # 作为参数占位符**
  - **表名、字段名、ORDER BY 排序字段不能使用 ？作为参数绑定**
  - 确保只能传递合法的表名和字段名
    - **白名单**校验
    - **Enum** 限制可选字段

SpringBoot 如何集成 Tomcat 容器？

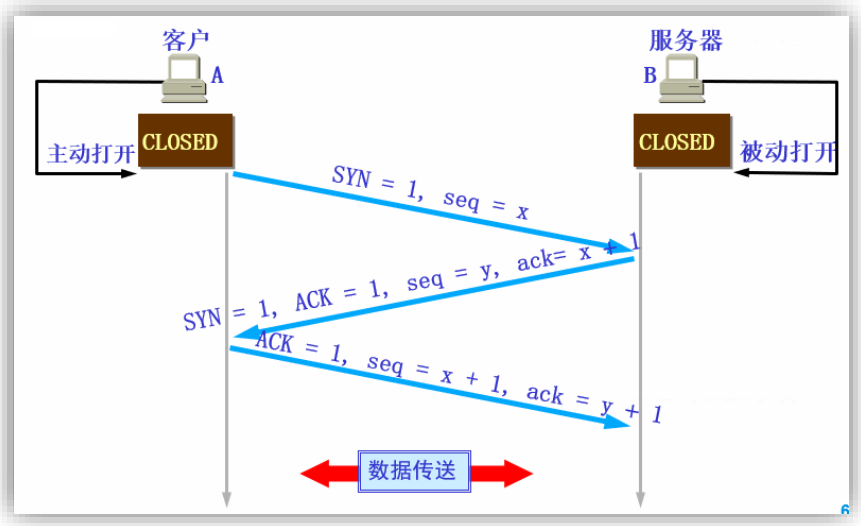
默认内嵌 Tomcat 容器

- **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

TCP 三次握手的详细过程和状态变化？

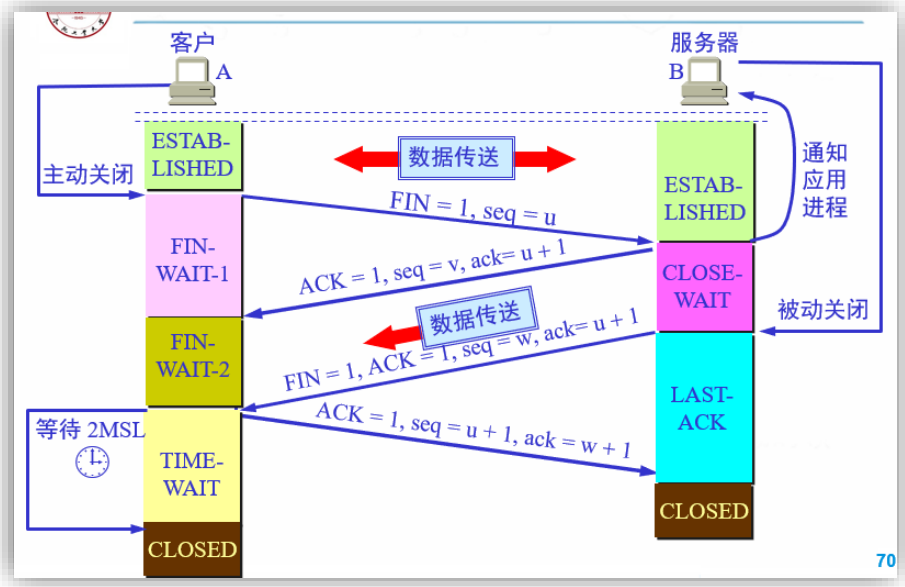
三次握手

- 第一次: 客户端发出**同步**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**旧连接的数据包**干扰新连接
2. **保证对端收到最后的 ACK**，避免关闭异常

TIME_WAIT 过多的解决方案：

1. 增加**可用端口**：sudo sysctl -w net.ipv4.ip_**local_port_range="32768 60999"**
2. 开启 TCP **连接复用**：sudo sysctl -w net.ipv4.**tcp_tw_reuse=1**
3. 使用**连接池复用连接**

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？

1) 慢启动

- 初始拥塞窗口 cwnd 为 1 个最大报文段大小(Maximum Segment Size, MSS)
- 每次 ACK 后 cwnd * 2，直到达到慢启动阈值(ssthresh, slow start threshold)或网络拥塞

2) 拥塞避免

当拥塞窗口达到 ssthresh 后，每个 RTT(往返时间）+1

3) 快速重传

收到三个重复的 ACK 后，立即重传被认为丢失的报文段，而无需等待超时

4) 快速恢复(Fast Recovery)

在快速重传后，不进入慢启动，而是

- cwnd / 2
- ssthresh = new cwnd
- cwnd + 1，快速恢复到丢包前的传输速率

HTTPS 和 HTTP 有哪些区别？

- **HTTPS**：443（**HTTP**：80）
- **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 + 非对称 + 哈希

1. 客户端 Hello：**SSL/TLS 版本、加密算法、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、数字证书**
4. 客户端：**验证证书、生成会话密钥后用公钥加密发送**
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. **keep-alive 持久连接**：复用同一连接（串行请求响应）
2. **请求管道化**：同时发送多个请求
3. **Cache-Control 头部**：更细粒度的缓存控制
   - no-cache：验证后才能缓存
   - no-store：不许缓存
   - max-age=60：缓存内容 60 秒有效
   - public：可以被任何缓存（比如代理）保存
   - private：只能被单个用户缓存，不能被代理保存
4. **Host 头部**：允许同一 IP 托管多个域名

即使同一个服务器只有一个 IP（比如 example.com 和 test.com 都指向同一个 IP），它也可以区分请求、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**：通过并行请求响应在**应用层**消除队头阻塞
   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
2. **独立流量控制**（1.1 依赖 TCP 拥塞控制）
3. **二进制协议** 以帧传输（1.1 基于文本）
4. HPACK 算法**头部压缩**（1.1 请求头冗长（如 Cookies））
5. 服务器**主动推送**

HTTP3 新特性？

1. **基于 QUIC 协议：传输层**消除队头阻塞（之前依赖 TCP 协议）
   - 基于 UDP、采用多路复用 + 独立流量控制 + 快速重传
   - Quick UDP Internet Connections 是 Google 提出，建立在 UDP 上
2. **零 RTT**：首次请求立即开始数据传输（之前基于 TCP 三次握手）
   - 往返时延（Round-Trip Time, RTT）
3. **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
- 创切开销大：**独立的代码、数据、文件**
- 进程间独立，需**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
- 创切开销小：**独立的 PC 和堆栈**
- 直接读写内存，需要**同步机制**

协程：
- 程序自身调度（**用户态**）
- 适合**高并发 I/O** 任务（日志分析、动画渲染）

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)**
2. **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)**
3. **epoll：**
   - **事件驱动机制**只关注活跃的文件描述符，效率高
   - 支持边缘触发和水平触发。时间复杂度：**O(1)**
     边缘触发只在**状态变化**时通知，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. **系统调用**需要内核验证
4. **缓存**失效
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
- **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
- **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？

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）系统调用的实现原理？

mmap 将**文件或设备**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

虚拟内存的作用？

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

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
   a) 0 接在 loTail 后面
   b) 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
- 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、红黑树**
2. **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**
3. **动态扩容机制：**
   a) 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。
   b) 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. **Concurrent**HashMap：
   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
  - == **比地址**
  - equals() 比值

- **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() **创建对象**
- .getField("xxx").setAccessible(true) **访问字段**
- .getMethod("xxx", String.class).invoke(obj, "param") **调用方法**

注意避免过度依赖反射：

- **原因**：动态解析性能开销比直接调用大
- **方案**：**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**：垃圾回收器**永不回收**强引用指向的对象，即使系统内存紧张，宁愿抛出 OutOfMemoryError
2. **软引用**：当系统**内存不足时回收**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**：**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**：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. **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）。
3. **方法区（元空间）**：存储类元信息、常量池（JDK8 后由元空间实现）。
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
2. 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
2. **Parallel GC** 多线程 GC，是 JDK8 默认垃圾回收器
   缺点：停顿时间较长
3. **Concurrent Mark Sweep 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）
4. **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
5. **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。

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：

- JVM 堆使用率长期在 90% 以上
- 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

解决：

1. 及时释放对象
2. 缓存过期策略
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、优化 GC 策略（G1 替代 CMS）

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）

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 -p <pid>：跟踪进程的**系统调用**
perf top：查看占用 CPU 最高的函数、分析 **CPU 性能瓶颈**

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**
   a) Deepseek 清华大学 104 页 PPT
      i. 推理模型（DeepSeek-R1）：数学推导、逻辑分析
      ii. 通用模型（GPT-4）：文本生成、创意写作
   b) 知识掌握得比人多、需求掌握得比人少（自我判断）
      i. 调试思路
      ii. 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）
   a) 学习了错误数据——数据清洗
   b) 基于概率生成，即使它们不正确——检索增强生成（RAG）
   c) 提示词不够具体，导致模型脑补——提示工程
   d) 缺乏真实世界反馈——强化学习 / Man-in-the-loop
3. **应用**
   a) 腾讯云黑客松 Copilot
   b) Cursor
   c) Coze
   d) 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？