目录

01.单体VS微服务

- 单体服务
  - 优点：开发简单、测试简单、部署简单、横向扩展简单、大规模更改简单
  - 缺点：
    - 过度的复杂性（代码屎山）
    - 开发速度缓慢（构建 - 运行 - 测试）
    - 部署周期长,易出问题（代码合并，牵一发而动全身，测试太慢）
    - 难以扩展和可靠性不佳（不同模块的资源需求不同，缺乏故障隔离）
  - 适用场景（小、少、短、快、雏）
- 微服务：微服务是模块化的、每个服务都拥有自己的数据库
  - 优点：持续交付和持续部署、容易维护、独立部署和独立扩展、容错性（故障隔离）
  - 缺点：
    - 服务的拆分和定义（分布式单体应用，集两者弊端于一身）
    - 分布式系统复杂性（可用可靠，事务，数据一致性，多服务查询，跨服务测试，运维复杂性）
    - 服务的依赖关系（当部署跨越多个服务的功能时需要谨慎地协调，协调团队，发布计划）

02.多数据源架构

1. `AbstractRoutingDataSource(ThreadLocal)`
2. `SqlSessionFactory(@MapperScan SqlSessionFactory)`
3. `dynamic-datasource(yml @DS("master"))`

4. 逆向工程：`mybatis-plus-generator`

前端发送请求大致流程

1. vue拦截请求，设置 token 和 memberid
2. gateway 拦截请求，jwt 校验 token
3. geteway 在请求头中添加 mermberid 后再转发(serverWebExchange)
4. 微服务在请求头中添加 memberid 向后传递(RequestContextHolder),只适合传递少量信息

任务调度：contab, xxl-job(IJobHandler), Netty时间轮算法(HashedWheelTimer, TimerTask)

03.微服务架构拆分

- 微服务拆分时机：业务规模、团队规模、技术储备、人才储备、研发效率
- 微服务拆分原则
  - 单一服务内部功能**高内聚低耦合**
  - **闭包原则**（CCP）
  - **服务自治、接口隔离**原则
  - **持续演进**原则
  - **避免影响产品的日常功能迭代**
  - **服务接口的定义要具备可扩展性**
  - **避免环形依赖与双向依赖**
  - **阶段性合并**
  - **自动化驱动**
- 功能维度拆分策略：领域驱动、数据驱动、单体拆分
  - **业务复杂度足够高：基于领域驱动拆分服务**
  - **业务复杂度较低：基于数据驱动拆分服务**
- 非功能维度拆分策略：**扩展性、复用性、高性能、高可用、安全性、异构性**
- 拆分注意的风险
  - **能否驾驭微服务的技术栈**
  - **不断纠正**
  - **不要纠结怎么拆，先拆了再说，发现不合适再重新调整**
  - **如果拆了之后发现真的不合适，在重新调整就好了**：合并后的包内服务也要满足微服务定义（DDD），以便日后拆分
- 微服务架构注意事项
  - 使用 RequestInterceptor 接口设置请求头传递 memberId
  - Skywalking + ELK

04.全链路灰度发布

- 流程：**标签路由、节点打标（K8S，Nacos）、流量染色、分布式链路追踪**
- 框架：**Discovery**（免费）：网关配置权重，微服务配置版本号；MSE（阿里云平台）

05.分布式唯一ID

- 要求：全局唯一性、单调（趋势）递增、信息安全（不能泄露订单量）
- UUID：MAC 地址泄露、无序（数据库主键无序导致B树频繁变动）
- 雪花算法：**时钟回拨**
- Mongdb ObjectID
- Seata UUID：`io.seata.common.util.IdWorker`
- MySQL：**泄露发号数量、增加数据库压力**

- **多部署几台机器，每台机器设置不同的初始值，且步长和机器数相等**
  - **水平扩展比较困难**
  - **没有了单调递增的特性（不同机器只能保证趋势递增）**
  - 每次只获取一个ID，数据库压力依然很大

- Redis：**持久化数据时存在着丢失数据的可能**
- Leaf-segment：**泄露发号数量**
  - TP999**偶尔的尖刺**：**双buffer优化**
    - **下发10%时更新下一个号段，下发完后切换到下个号段**
    - **segment长度设置为服务高峰期发号QPS的600倍（10 分钟），DB宕机时Leaf 仍能持续发号10-20 分钟**
  - **Leaf 高可用容灾（一主两从）：**DBProxy(360), Paxos(强一致)
- Leaf-snowflake
  - **使用Zookeeper配置workerid，本地缓存workerid文件**
  - **解决时钟问题**
    - 新节点启动时计算**sum(time)/nodeSize是否超过阈值**
    - 老节点每隔一段时间上报自身系统时间
    - 老节点比较zookeeper上本节点曾经的记录时间
    - 老节点比较所有运行中的节点时间
  - 出现时钟回拨
    - **直接关闭NTP同步**
    - **直接不提供服务返回ERROR_CODE，等时钟追上**
    - **做一层重试，然后上报报警系统**
    - **自动摘除本身节点并报警**
- Leaf性能：在 4C8G 的机器上 QPS 能压测到近**5万/s**，TP999 1ms

06.网关整合授权中心

1. 配置授权服务器（AuthorizationServerConfigurerAdapte - 授权码/密码）：DB模式、内存模式
2. SpringSecurity（WebSecurityConfigurerAdapter）
3. JWT：头部（header）、载荷（payload）与签名（signature 加盐）
   - 一次性验证、无状态认证
   - 注销续约复杂，不适合会话管理
4. JWT非对称加密（公钥私钥）
5. 扩展JWT中的存储内容（TokenEnhancer）
6. 接入网关服务（GlobalFilter）
   1. 过滤不需要认证的url
   2. 校验token（需要从授权服务获取公钥）

    1. 不能直接通过@LoadBalancer配置RestTemplate去获取公钥，因为Spring容器启动过程中@LoadBalancer还未生效
    2. 再容器启动后的事件中获取私钥，ContextRefreshedEvent
3. 校验通过后，从token中获取的用户登录信息存储到请求头中

07.订单系统设计

- 重复下单问题: **主键唯一约束**，前端会先调用生成订单号的服务得到一个订单号，用户提交订单时，在创建订单的请求中带着这个订单号。
- 订单ABA问题: **版本戳** (verison)：页面在更新数据的请求时,需要把该版本号作为更新请求的参数再带回给订单更新服务
- 读写分离（提升DB并发能力的**首选方案**）: **读写比例一般在9:1**
    - 数据不一致问题: 增加一个支付完成页面：**尽量规避更新数据后立即去从库查询刚刚更新的数据**
    - 如果数据更新后需要立刻查询，这俩步骤可以放到一个事务中，或单独指定主库查询
- 分库分表（**能小拆就小拆，最后一招**）：**如果数据量太大，就分表；如果并发请求量高，就分库**
    - **选择分片键**：MySQL**每个表中的数据不宜超过2000W**
        - **把用户ID的后几位作为订单ID的一部分。这样按订单ID查询的时候，就可以根据订单ID中的用户ID找到分片（便于用户查询）**
        - **把订单里数据同步到其他存储系统中（比如以店铺ID分片供商家查询使用）**
    - 实现方式：
        - 手工（简单业务推荐）
        - **Sharding-JDBC（推荐，代码侵入少）**
        - 代理：Atlas Sharding-Proxy Mycat（不推荐，多了一层代理且代理层也可能出现故障）
- 归档历史数据（迁移之前做好备份）：**热尾效应**
    - 尽量不要影响线上的业务（**闲时迁移**）：**每次控制在10000以下是比较合适**
    - 系统需要改动的代码非常少，基本上只有查询统计类的功能会查到历史订单
    - **不需要分布式事务**
        - 写入订单数据到MongoDB时，**同时写入当前迁入数据的最大订单ID，让这两个操作执行在同一个事务之中**
        - 在MySQL执行数据迁移时，**先去MongoDB中获得上次处理的最大OrderId，作为本次迁移的查询起始ID**
        - 写入后，删除MySQL中对应的数据（批量删除大量数据）
            - 按ID来删除，并且同样要控制住每次删除的记录条数
            - **删除语句按ID排序，每批删除的记录基本上都是ID连续的**
            - 执行删除语句后，最好能停顿一小会，因为**删除后会牵涉到大量的B+树页面分裂和合并**

08.支付超时设计

- 延迟任务（不推荐）
- RocketMQ事务消息（反向通知）：正向通知需要**订单确认的下游服务实现幂等控制**
    1. 支付宝预下单时发送事务消息：通知下游服务进行订单取消
    2. 发送消息后，会先执行本地事务。
        - 将订单ID放到Redis中，这样可以在后续进行支付状态检查时，快速找到对应的业务信息。

        - 只要下单成功，就会返回`UNKOWN`状态，这样RocketMQ会在之后进行状态回查（检查支付宝支付状态）。
    3. 然后在事务状态回查时
        - 会自行记录回查次数，超过最大次数就直接取消订单（`transactionCheckInterval`，默认回查15次，间隔60s，超过`ROLLBACK`）
        - 如果没有超过最大次数，就可以去支付宝中查询订单支付状态。
            - 如果已经支付完成，则返回`ROLLBACK`状态，消息取消，后续就不会再进行本地订单取消了。
            - 如果未支付，则记录回查次数后，返回`UNKNOWN`状态，等待下次回查。
    4. 如果订单已经超时（事务消息成功发送出去了）：下游的消费者就会完成取消本地订单，释放库存等操作
    5. 如果本地订单已经取消，而支付宝支付状态已经成功，则退款
- 兜底补偿机制：**定时任务**批量回退超时的订单
- 聚合支付平台（分布式锁）：**防止在电商进行订单超时回退后，用户再次扫码支付**

09.分布式事务

- Seata架构：TC、TM、RM
    1. TM 请求 TC 开启一个全局事务。TC 会生成一个 XID 作为该全局事务的编号。XID会在微服务的调用链路中传播，保证将多个微服务的子事务关联在一起。
    2. RM 请求 TC 将本地事务注册为全局事务的分支事务，通过全局事务的 XID 进行关联。
    3. TM 请求 TC 告诉 XID 对应的全局事务是进行提交还是回滚。
    4. TC 驱动 RM 们将 XID 对应的自己的本地事务进行提交还是回滚。
- 分库分表全局事务（ShardingSphere）
    1. **配置seata.conf**：`client{application.id=order \n transaction.service.group=order-group}`
    2. **开启全局事务配置**：`@ShardingTransactionType(TransactionType.BASE)`，`@Transactional`
    3. **关闭Seata数据源自动代理**：`seata.enable-auto-data-source-proxy=false`
- 柔性事务：可靠消息最终一致性
    - 本地消息表方案
        1. **定时任务扫描日志**: 启动独立的线程，定时对消息日志表中的消息进行扫描并发送至消息中间件，在消息中间件反馈发送成功后删除该消息日志，否则等待定时任务下一周期重试。
        2. **消费消息（幂等控制）**: 使用MQ的ack（即消息确认）机制，消费者监听MQ，如果消费者接收到消息并且业务处理完成后向MQ发送ack（即消息确认），此时说明消费者正常消费消息完成，MQ将不再向消费者推送消息，否则消费者会不断重试向消费者来发送消息。
    - RocketMQ 事务消息: 解决Producer端的**消息发送与本地事务执行的原子性**问题
        1. Producer发送事务消息: Producer（MQ发送方）发送事务消息至MQ Server，MQ Server将消息状态标记为Prepared（预览状态），注意此时这条消息消费者（MQ订阅方）是无法消费到的。
        2. MQ Server回应消息发送成功: MQ Server接收到Producer发送的消息则回应发送成功表示MQ已接收到消息。
        3. Producer执行本地事务
        4. 消息投递
            - 若Producer本地事务执行成功则自动向MQ Server发送commit消息

    - MQ Server接收到commit消息后将"该消息"状态标记为可消费，此时MQ订阅方即正常消费消息；
    - 若Producer 本地事务执行失败则自动向MQ Server发送rollback消息，MQ Server接收到rollback消息后将删除"该消息"。
    - MQ订阅方消费消息，消费成功则向MQ回应ack，否则将重复接收消息。这里ack默认自动回应，即程序执行正常则自动回应ack。
5. 事务回查: 如果执行Producer端本地事务过程中，执行端挂掉，或者超时，MQ Server将会不停的询问同组的其他Producer来获取事务执行状态，这个过程叫事务回查。MQ Server会根据事务回查结果来决定是否投递消息。
6. **需要分别实现本地事务执行以及本地事务回查方法**: `RocketMQLocalTransactionListener`

10.高并发缓存

缓存一定是离用户越近越好，适当缓存预热

- Redis数据一致性
    - 先更新缓存，再更新数据库（更新DB异常）
    - 先更新数据库，再更新缓存（更新缓存失败）
    - 先删除缓存，后更新数据库（更新DB之前被其他进程更新了缓存：异步延时双删）
        - 主从复制问题：更新缓存的查询指定为主库
    - **先更新数据库，后删除缓存（写入缓存之前被其他进程更新了DB：异步延时双删）**
        - 低概率：要求写库操作快于读库
    - **使用Canal监听binlog日志，更新缓存**
- 本地缓存：**设置双缓存**
    - 正式缓存是最后一次写入后经过固定时间过期
    - 备份缓存是设置最后一次访问后经过固定时间过期
    - 异步刷新：正式缓存只有无效才会被重新写入，备份缓存无论是否无效都会重新写入
- 大厂不用Redis Cluster构建集群（去中心化不适合大集群）：代理模式
    - **把代理服务的寻址功能前移到客户端中**：ZooKeeper保存元数据
- 缓存更新设计模式：Cache Aside、Read/Write Through、Read Through、Write Through、Write Behind Caching

11.数据高可用

- Canal：**把自己伪装成一个MySOL的从节点**
    1. **先开启Binlog写入功能，配置binlog-format为ROW模式**
    2. **给Canal设置一个用来复制数据的MySQL账号**
    3. **修改canal.properties文件，比较关键的是canal.destinations**
    4. **修改instance.properties**
        - **MySQL 主服务的连接配置**
        - **要对哪些相关的业务表进行监视**
    - 跨系统实时数据同步：**息队列来解耦上下游**
- 不停机更换数据库（**保证每一步都是可逆的**）

1. **把旧库的数据全部复制到新库中，保证新旧两个库的数据是实时同步的**：这一步不需要回滚，停掉同步程序就可以了
2. 改造DAO：**支持双写和读取新旧两个库，并预留热切换开关**
3. **只读写旧库，验证新旧两个库中的数据是否保持一致**
4. **开启双写开关，停掉同步程序**
    - **先写旧库，再写新库，并且以旧库的结果为准**
    - **如果旧库写成功，新库写失败，则返回成功**
    - **如果旧库写失败，则直接返回失败**，同时也不再写新库了
    - **不能让新库影响到现有业务的可用性和数据准确性**
    - 数据不一致：**比对和补偿的程序**
5. **灰度发布的方式把读请求逐步切换到新库上**
6. **全部读请求都切换到新库**：停掉比对程序，把服务的写状态改为只写新库。至此，旧库就可以下线了（只有这个步骤是不可逆的）
    - 过渡状态：从双写以旧库为准过渡到双写以新库为准。然后把比对和补偿程序反过来，用新库的数据补偿旧库的数据（成本太高）
7. **在下一次升级服务版本时下线双写功能**

- 数据表的变更
    - 如果只是新增表：直接回退到旧版本程序
    - 如果牵涉到**表字段的变化**：**双写新旧表并设计热切换开关。**

- 比对和补偿程序
    1. **发现不一致的情况后，直接用旧库的订单数据覆盖新库的订单数据就可以了**
    2. **每次从旧库中读取一个更新时间窗口内的数据，到新库中查找具有相同主键的数据进行比对**：如果数据没带时间戳信息，那就只能从旧库中读取Binlog
        - **如果新库数据的更新时间晚于旧库数据**，那么很可能是比对期间数据发生了变化，这种情况暂时不要补偿，**放到下个时间窗口继续进行比对**即可。
        - **时间窗口的结束时间不要选取当前时间**，而是要比当前时间早一点，比如1分钟之前，这样就可以**避免比对正在写入的数据**了。
- 数据备份和恢复：通过定期的**全量备份配合Binlog**，我们可以把数据恢复到任意一个时间点
    - **指定的起始时间可以比全量备份的时间稍微提前一点儿**：为了确保回放的幂等性，需要将Binlog的格式设置为ROW格式
    - **不要把所有的鸡蛋放在同一个篮子中**

12.秒杀系统设计

- 秒杀隔离: 业务隔离(提报系统)、系统隔离、数据隔离(一主多从)
- 页面静态化（Nginx）：商详页静态化(freemarker)
    - **直接让Nginx(OpenResty - lua)访问Redis**
    - **Redis从库和Nginx在部署在同一台服务器**：Unix Domain Socket
    - 本地Redis宕机时, 回源到微服务中查询主Redis或者数据库
- 流量管控
    - 前期（**预约系统**）
        - **固定时段、人数上限（即时熔断）**
        - **预约关系表：分库分表&历史归档&MQ异步写入**
        - **预约商品表：Redis一主多从&本地缓存**
        - **页面预约人数：本地缓存累加，批量写入Redis**
    - 事中（服务平稳处理&节省机器成本）

        - **验证码和问答题（可以独立为一个微服务）**
            - **从session共享角度来说，验证码应该放入Redis**
            - Nginx Lua读时:**读从Redis未果则读主Redis；**或休眠后再读
        - **消息队列**接受：库存为0后在缓存里设置占位符，让后续请求快速失败
    - **限流（逐级限流，分层过滤 - 令牌桶和漏桶）**
        - Nginx 限流（**HttpLimitzone 和 HttpLimitReqest**）：**1 个IP 1 秒内只能访问 1 次**
        - 应用/服务层限流：线程池限流（最大连接数）；API限流（QPS- RateLimiter，**Sentinel**）
        - 自定义限流：**本地预先存放一批订单ID**，通过定时任务刷新
        - 分层过滤（秒杀5分钟，前2分钟抢完了，则**后3分钟为无效请求**）
            - **Nginx本地缓存**库存信息（商详页直接返回秒杀结束）
            - 服务层：**秒杀确认单和秒杀订单都会进行库存检查**：在实际下单和扣减库存前中断用户的请求链路
    - **限购（用户提单时，通过限购请求，再去做真实库存扣减）**
        - 商品维度限制：不同地区的库存不同
        - 个人维度限制：手机号、收货地、设备IP
- 库存扣减（先查询库存，再库存扣减）：超卖（查询和扣减不是原子操作；并发引起的请求无序）
    - 数据库方案（**简单安全，性能较差，**适合请求量较小的场景）
        - 行锁：查询库存的时候使用`for update`；`where`条件，保证库存不会被减到0以下
        - 乐观锁：查询库存时除了库存值还有版本号
            - 每次扣减库存时带上这个版本号进行扣减
            - 扣减失败，则需要重新查询，重新扣减（加重数据库的负担）
        - 将库存字段设为无符号整数：值小于零时直接报错
    - 分布式锁方案（可以实现，但不建议）：锁的有效期问题（时间太长太短都对业务有不利影响）
    - 高并发扣减方案（单个商品的高并发读和高并发写问题）
        - **写服务降级**（牺牲数据一致性获取更高的性能）
            - **同步写缓存（Lua脚本）、MQ异步写数据库（定时任务比对）**
            - **宁可少买，不可超卖**，少卖可以做返场
        - **读服务降级**（故障场景下紧急降级快速止损）
            - **建设多级缓存**（Redis, MongoDB, ES, 多副本）
            - 缓存全部故障时，切换到 DB 暂抗
        - **简化系统功能**（舍弃非核心功能）：只对流量非常高的SKU进行降级
            - 交互少，数据小，链路短，离用户近
- 热点数据（单商品SKU落在的那个分片上性能总会达到上限）
    - 读热点
        - **增加Redis从副本数**
        - **在服务内部做热点数据的本地缓存**（但本地缓存有数据延迟）
        - **不支持该活动的SKU直接短路返回**（不通用，和具体商品有关）
    - 写热点
        - **本地累加，延迟批量提交到Redis**
        - **一个热 key拆解成多个key**（复杂）
        - **对单SKU的库存直接在Redis单分片上进行扣减，进行单独限流**：扣减库存在秒杀链路的末端，流量有限
- 防刷
    - Nginx限流
    - Token机制：**对于有先后顺序的接口调用，我们要求进入下个接口之前，要在上个接口获得令牌**
    - 黑名单（本地黑名单和集群黑名单）：从外部导入（风控或别的渠道）或自己生成

        - 根据Nginx的流量进行分析，利用Lua共享缓存，**统计1秒内这个用户或者IP的请求频率，**放入本地缓存黑名单，但是如果黑产将请求频率控制在1*Nginx机器数以内，就不会被抓到
        - **借助Redis实现集群黑名单**（影响接口响应性能）
- 风控（不断完善用户画像）：需要大量数据和复杂的实际业务场景
- 容灾（同城双活，异地多活）：
    - 同城双活：**单向时延2ms以内（物理专线至少两根以上）**
    - 异地多活：成本太高，设计复杂

13.Sentinel限流

- 限流
    - 前端限流、接入层nginx限流
    - 网关限流
        - 基于redis+lua脚本限流（RequestRateLimiter，令牌桶）
        - sentinel限流
            - route维度限流：针对请求属性进行流控
            - API维度限流
            - 规则持久化：改造Sentinel控制台（指定端口和nacos配置中心地址）
                - 网关规则实体转换： RuleEntity#toRule；ApiDefinitionEntity#toApiDefinition；GatewayFlowRuleEntity#toGatewayFlowRule
                - json解析丢失数据：重写实体类MyApiDefinition,再转换为ApiDefinition
    - 应用层限流（微服务接入sentinel）
        - 匀速排队限流：处理间隔性突发的流量（漏桶算法）：暂时不支持 QPS > 1000 的场景
        - 热点参数限流（`@SentinelResource("resourceName")`）：参数必须是7种基本数据类型才会生效
- 降级
    - 自动化运维：自动开关降级（超时、失败次数、故障、限流）、手动开关降级
    - 功能维度：读服务降级、写服务降级
    - 系统层次维度：JS降级开关、接入层降级开关、应用层降级开关（OpenFeign整合Sentinel）
- 拒绝服务
    - 系统资源过载保护、Nginx过载保护
    - Sentinel自适应限流：系统规则持久化（system-rules.nacos）
        - Load 自适应（Linux/Unixlike）：参考值 CPU cores * 2.5
        - CPU usage：比较灵敏
        - 平均 RT：单位是毫秒
        - 并发线程数
        - 入口QPS

14.高并发读写

高并发读：加缓存/读副本；并发读；重写轻读；读写分离

- 加缓存/读副本

    - 本地缓存或集中式缓存（主动更新/被动更新）：高可用问题
        - 不回源策略：缓存为空时直接返回空，不设置缓存的过期时间
        - 回源策略：缓存为空时查库更新缓存（缓存穿透/缓存击穿/热Key过期/缓存雪崩）
    - MySQL的Master/Slave
    - CDN/静态文件加速/动静分离
- 并发读
    - 异步RPC
    - 冗余请求：100台机器每台延迟率为1%，则C端延迟率为 (1-99%100)=63.4%
        - 客户端同时向多台服务器发送请求，哪个返回得快就用哪个（但是会调用量翻倍）
        - 对冲请求：可以仅用2%的额外请求将系统99.9%的请求响应时间从1800ms降低到74ms
            - 如果客户端在一定的时间内（内部服务95%请求的响应时间）没有收到服务端的响应，则马上给另一台（或多台）服务器发送同样的请求
        - 捆绑请求
            - 当一个上游系统向下游系统分发请求时，将请求分发给任务队列较短，负载较轻的服务器，因为服务抖动一个常见的来源是请求被执行前在服务端的排队延迟
            - 但是探测负载后服务器负载会发生变化，而且多个服务探测到一个低负载服务时，会使该服务瞬间过热
                - 上游系统同时发送两个一样的请求给下游多个服务器，当下游服务器处理完成后，通知其他服务器不用再处理该请求。
                - 为了防止由于任务队列为空而导致的所有服务器同时处理任务的情况，上游系统需要在发给下游多个系统的两个请求之间引入一个延迟，该延迟时间要足够第一个系统处理完任务并通知其他系统。
        - 不要在底层设置太长的任务队列，而由上层基于任务的优先级进行动态的下发
            - 自己维护一个队列允许服务器跳过那些更早到来的非延迟敏感的批处理操作，优先往下传那些高优先级的交互请求
        - 将需要长时间运行的任务拆成多个短时间运行的任务
        - 控制好定时任务和后台运行的任务
            - 将后台的大任务拆分成一系列的小任务，在后台任务运行的时候首先确认系统的负载，如果负载太高，则延迟运行后台任务等
- 重写轻读
    - 微博Feeds流：不是查询的时候再去聚合，而是提前为每个user_id准备一个Feeds流，或者叫收件箱，每个用户都有一个发件箱和收件箱
        - 写扩散：发布1条微博后，只写入自己的发件箱就返回成功。然后后台异步地把这条微博推送到1000个粉丝的收件箱
        - 限制数量：Redis最多保存2000条，2000条以外的丢弃。2000个以前的数据持久化存储并且支持分页查询
            - DB分库分表同时按user_id和时间范围进行分片
            - DB二级索引：记录`<user_id,月份,count>`，快速地定位到offset = 5000的微博发生在哪个月份，也就是哪个数据库的分片
        - 推拉结合：大V的粉丝太多了
            - 对于粉丝数多的用户，只推送给在线的粉丝们，离线用户上线时主动去拉
    - 多表的关联查询：宽表与搜索引擎
        - 分库分表下的关联查询：提前把关联数据计算好，存在一个地方，读的时候直接去读聚合好的数据，而不是读取的时候再去做Join
        - 可以另外准备一张宽表：把要关联的表的数据算好后保存在宽表里
            - 可以定时算，也可能任何一张原始表发生变化之后就触发一次宽表数据的计算

        - ES搜索引擎：把多张表的Join结果做成一个个的文档
- 读写分离（CQRS 架构）
    - 分别为读和写设计不同的数据结构
    - 写-分库分表；读-缓存；join宽表存ES
    - 读和写的串联
        - 定时任务定期把业务数据库中的数据转换成适合高并发读的数据结构
        - 写的一端把数据的变更发送到消息中间件，然后读的一端消费消息
        - 直接监听业务数据库中的 Binlog，监听数据库的变化来更新读的一端的数据
    - 读比写有延迟（最终一致性）
        - 但是对对于用户自己的数据，自己写自己读，在用户体验上肯定要保证自己修改的数据马上能看到
            - 读和写完全同步
            - 异步的，但要控制读比写的延迟非常小,用户感知不到
        - 保证数据不能丢失、不能乱序

高并发写：数据分片；异步化；批量写

- 数据分片：对要处理的数据或请求分成多份并行处理
    - 分库分表
    - ES分布式索引：并行地在n个索引上查询，再把查询结果进行合并
- 异步化：凡是不阻碍主流程的业务逻辑，都可以异步化，放到后台去做
    - 短信验证码注册或登录：60s之后没有收到短信，用户会再次点击
    - 电商的订单系统拆单（支付完成后，服务器会立即返回成功，而不是等1个拆成3个之后再返回成功）
    - 广告计费系统
        - 在扣费之前其实还有一系列的业务逻辑要处理，比如判断是否为机器人在刷单
        - 实际上用户的点击请求或浏览请求首先会以日志的形式进行落盘。一般是支持持久化的消息队列
        - 落盘之后，立即给客户端返回数据。后续的所有处理，当然也包括扣费，全部是异步化的，而且会使用流式计算模型完成后续的所有工作。
    - 写内存 + Write-Ahead日志：MySQL中为了提高磁盘IO的写性能，使用了Write-Ahead日志，也就是Redo Log。
        - 高并发地扣减 MySOL 中的账户余额，或者电商系统中扣库存，如果直接在数据库中扣，数据库会扛不住，则可以在Redis中扣，同时落一条日志（日志可以在一个高可靠的消息中间件或数据库中插入一条条的日志）。当Redis宕机，把所有的日志重放完毕，再用数据库中的数据初始化Redis 中的数据。
- 批量写
    - 广告计费系统的合并扣费
        - 扣费模块一次性地从持久化消息队列中取多条消息，对多条消息按广告主的账号ID进行分组，同一个组内的消息的扣费金额累加合并，然后从数据库里扣除
    - MySQL 的小事务合并机制
        - 比如扣库存，对同一个SKU，本来是扣10次、每次扣1个，也就是10个事务；在MySQL内核里面合并成1次扣10个，也就是10个事务变成了1个事务。
        - 我们同样可以借鉴这一点，比如我们的Canal同步代码中，首页广告内容的更新就采用同样的机制
        - 在多机房的数据库多活（跨数据中心的数据库复制）场景中，事务合并也是加速数据库复制的一个重要策略

RockDB（LSM-Tree 如何兼顾读写性能）：虽然RockDB性能还不如Redis，但是已经可以算是在同一个量级水平了，毕竟一个是内存操作，另一个是磁盘IO操作。RocksDB 的数据结构，可以保证写入磁盘的绝大多数操作都是顺序写入的

1. 操作命令写入磁盘的 WAL 日志中（顺序写）：唯一作用就是从故障中恢复系统数据
2. 写入内存的MemTable中（按照Key组织的跳表，MemTable 太大会导致读写性能下降，一般是32MB）
    1. 直接将Key 写入 MemTable，而不会预先查看MemTable中是否已经存在该Key
    2. 写入MemTable之后就可以返回写入成功
3. MemTable写满之后，就会转换成Immutable(不可变的)MemTable，然后再创建一个空的MemTable继续写
    1. Immutable MemTable也不能在内存中无限地占地方，而是会有一个后台线程，不停地把Immutable MemTable复制到磁盘文件中，然后释放内存空间。
    2. 每个Immutable MemTable 对应于一个磁盘文件，这些文件就是SSTable（这些SSTable文件中的Key是有序的,但是文件之间却是完全无序的，所以还是无法查找）
4. SSTable 采用分层合并机制，SSTable被分为很多层，每一层的容量都有一个固定的上限。下一层的容量是上一层的10倍。
    1. 当某一层写满时，就会触发后台线程往下一层合并
    2. 数据合并到下一层之后，本层的SSTable文件就可以删除了
    3. 合并的过程也是排序的过程，除了Level 0以外，每层中的文件都是有序的
5. LSM-Tree 查找的过程也是分层查找
    1. 先在内存中的 MemTable 和Immutable MemTable 中查找，然后再按照顺序依次在磁盘的每层 SSTable 文件中查找，一旦找到了就直接返回
    2. 越是经常被读写到的热数据，它在这个分层结构中就越靠上，对这样的Key 查找就越快
    3. 在内存中缓存SSTable文件的Key用布隆过滤器避免无谓的查找等，以加速查找过程
    4. 使用多个memtable并在immutbale memtable提前进行数据合并
    5. 为存储的数据逻辑分族，独特的filter对读优化

15.海量数据存储

- 存储系统的技术选型（在线业务系统? 分析系统?）
    - **数据量.** 不必对未来做过多的预留, 最多三年来估计就足够了
        1. 1GB以下或数据量在千万以下：都可以
        2. 1-10G或数据量在一亿以内：单机存储系统的上限
        3. **超过10GB或数据量超过一亿：分布式存储，数据分片**
    - **总体拥有成本：**团队是否熟悉该产品；是否简单、易于学习和使用；运维成本
- 在线业务系统（**增删改查**）：MySOL一般是首选（写性能、毫秒级、并发、强大的查询能力）
    - 存储选型
        - GB级：MySQL；超过GB级：ES，Hbase、Cassandra、ClickHouse
        - 超过TB级：HDFS；只能事先对数据做聚合计算，然后再在聚合计算的结果上进行实时查询
    - **商品系统（高并发、SKU数据规模）**
        - 小规模的电商：用数据库表存储
        - **分而治之**：把商品系统需要存储的数据，按照特点分别进行存储
            - **可以在数据库中建一张表来保存商品的基本信息，在数据库前面加一个缓存**
            - **保留商品数据的每一个历史版本：订单中关联的商品数据，必须是下单那个时刻的商品数据**

        - **使用 MongoDB 保存商品参数（参数信息数据量大、数据结构不统一，不需要对商品参数进行事务和多表联查）**
        - **使用对象存储保存商品图片和视频（在数据库中只保存图片和视频的ID或URL）**
        - **商品介绍静态化+CDN：CMS（内容管理系统）**
    - **购物车系统（展示、结算）**：存储SKUID(商品ID)、数量、加购时间和勾选状态。商品的价格和总价，以及商品介绍等信息，可以从电商的其他系统中实时获取，不需要购物车系统专门保存
        - **暂存购物车（用户未登录时加入的购物车）**：Cookie（小型电商系统）、LocalStorage（节省带宽）
        - **用户购物车（保持同步）**
            - 复杂查询：MySQL
            - 高性能高并发：Redis（对可靠性要求不高），异步写入DB
    - 选择方案：复杂度、性能、系统可用性、数据可靠性、可扩展性
        - **这些因素中的每一个都是可以根据业务适当牺牲的**
        - **如果性能提升带来的收益远大于丢失少量数据所付出的代价，那么这个选择就是合理的**
- 离线分析系统 （海量数据的存储和计算）
    - 前端海量数据（**前端埋点数据、监控数据和日志数据**）：**先存储再计算**
        - **一般都会选择Kafka/RocketMQ来存储**
        - 如果是需要长时间(几个月到几年)保存的海量数据，适合用**HDFS之类的分布式文件系统来存储**
        - 时序数据库**InfluxDB：专用于存储类似于监控之类的有时间特征，并且数据内容都是数值的数据**
    - **提高查询速度：流计算或批计算，几十秒甚至几分钟都算是快速的**
        - **把计算结果保存到另外一个存储系统**
        - **对于写入性能和响应时延，要求一般不高，也不要求高并发**
        - 数据库选型
            - GB级以下：MySQL
            - 超过MySQL极限：**Hbase、Cassandra、ClickHouse**（10GB量级的数据查询基本上可以做到秒级返回）
            - **ES：支持结构化数据的存储和查询，也支持分布式并行查询**（缺点：需要具有大内存的服务器）
            - 超过TB级：**HDFS、Spark、Hive**
    - **根据查询选择存储系统**：不同的存储系统之间并没有本质的差异，只有在它所擅长的那些领域或场景下，才会有很好的性能表现
    - **京东智能的补货系统：根据历史的物流数据，对未来的趋势做出预测，为全国的每个仓库补货（几十亿的量级）**
        - **按照区域做分片，再汇总**
        - **站点和快递人员的时效达成情况：事先按照站点和快递人员把数据汇总好,存放到分布式KV存储中**
        - **物流规划：把数据放到Hive表中，按照时间进行分片**
            - 物流规划人员可以在上面执行一些分析类的查询任务，这样的查询任务即使是花上数小时的时间，用于验证一个新的规划算法，也是可以接受的

16.ES数据迁移

不同方法的迁移步骤：

- elasticsearch-dump：适合数据量不大，迁移索引个数不多的场景
- snapshot：适用数据量大的场景，原理就是从源ES集群创建数据快照，然后在目标ES集群中进行恢复
    - 创建快照前必须先创建repository仓库，一个repository仓库可以包含多份快照文件
    - 源ES集群中创建snapshot
    - 目标ES集群中创建repository
    - 移动源ES集群snapshot至目标ES集群的仓库
    - 从快照恢复
- reindex：跨索引、跨集群的数据迁移，应用于大量集群数据的迁移
    - 配置reindex.remote.whitelist参数，指明能够reindex的远程集群的白名单
    - 调用reindex api
- logstash

特点：

1. elasticsearch-dump和logstash做跨集群数据迁移时，都要求用于执行迁移任务的机器可以同时访问到两个集群，不然网络无法连通的情况下就无法实现迁移。而使用snapshot的方式没有这个限制，因为snapshot方式是完全离线的。因此elasticsearch-dump和logstash迁移方式更适合于源ES集群和目标ES集群处于同一网络的情况下进行迁移，而需要跨云厂商的迁移，比如从阿里云ES集群迁移至腾讯云ES集群，可以选择使用snapshot的方式进行迁移，当然也可以通过打通网络实现集群互通，但是成本较高。
2. elasticsearchdump工具和mysql数据库用于做数据备份的工具mysqldump工具类似，都是逻辑备份，需要将数据一条一条导出后再执行导入，所以适合数据量小的场景下进行迁移；
3. snapshot的方式适合数据量大的场景下进行迁移。

17.架构师能力提升

业务驱动技术：需求来自何处？真需求还是伪需求？产品手段vs技术手段？需求的优先级？什么是业务：闭环明明是业务问题，却想用技术手段解决；明明是技术问题，由于技术无法实现，反过来将就业务；可能既不是业务问题，也不是技术问题,而是组织架构问题，是部门利益问题，是公司的盈利模式问题，实践中只有时刻意识到我们面对的是业务问题，还是技术问题，或是其他的更高层次的问题，才可能在一个正确的层面上去解决，**“业务架构”不是“技术架构”，是“技术架构”外面的东西**

业务架构

- **业务架构既关乎组织架构，也关乎技术架构**
    - **“康威定律”：一个组织产生的系统设计等同于组织之内、组织之间的沟通结构。**这也意味着：如果组织架构不合理，则会约束业务架构，也约束技术架构的发展。而组织架构的调整涉及部门利益的重新分配，所以往往也只能由高层来推动
    - 技术的思考会反过来影响业务，重新思考团队的组织方式，团队的组织方式又会变,接下来又会影响业务的发展方式
- **警惕“伪”分层**：底层调用上层；同层之间，服务之间各种双向调用；层之间没有隔离，参数层层透传，一直穿透到最底层，导致底层系统经常变动；聚合层特别多，为了满足客户端需求，各种拼装（业务服务层太薄，纯粹从技术角度拆分了业务。而不是从业务角度让服务成为一个完整的闭环，或者说一个领域）
    - 越底层的系统越单一、越简单、越固化；越上层的系统花样越多、越容易变化。要做到这一点，需要**层与层之间有很好的隔离和抽象。**
    - **层与层之间严格地遵守上层调用下层的准则**
- **边界思维**：优雅的接口，龌龊的实现；把混乱的逻辑约束在一个小范围内，而不会扩散到所有系统

    - **单一职责原则；接口的设计往往比接口的实现更重要；把复杂留给自己，把简单留给用户**；组织的各个部门之间边界清晰
    - **边界思维的重点在于“约束”，是一个“负方法”的思维方式**：要看的不是它能做什么，而是它不能做什么
        - **架构强调的不是系统能支持什么，而是系统的“约束”是什么**
        - 没有“约束”，就没有架构；如果“无所不能”，则意味着“一无所能”
- **系统化思维**
    - **把不同的“东西”串在一起考虑，而不是割裂后分开来看**
    - **刨根问底**：单独去看每一个业务的每一个系统，都没有问题，但要系统性地把这五大类业务串在一起来看，可能库存的账目是对不齐的。**所以库存往往是单独的微服务，实际问题在于库存不是一个简单的数字，而是一个复杂的数据模型，有自己单独的领域**
- **利益相关者分析**：首先要确定的是系统为哪几类人服务，同哪几个外部系统交互，也就确定了系统的边界
    - 把系统当作一个**黑盒子**，看为哪几类人服务。也就**定义了整个系统的边界，定义了整个系统做什么和不做什么。**
    - 业务具有“闭环”的特点。**利益相关者就是一个最好的看待业务的视角。**
    - **每个利益相关者代表了一个视角**
    - **根据不同的利益相关者，可以划分成不同的系统和业务。**
- **非功能性需求：并发性；可用性；一致性；稳定性和可靠性；可维护性；可扩展性；可重用性；**
    - 对于C端应用，会更关注高并发和高可用，然后有的业务（比如支付）对一致性要求非常高；
    - 对于B端业务，会更关注系统的可维护性、可扩展性性
- **抽象**：分析和分解各种概念、实体、系统，然后又造出一些新的概念、框架的过程
    - 计算机中的抽象：存储的抽象；计算的抽象是顺序、选择、循环；面向对象的方法学；面向接口的方法学；
    - 怎么做抽象：**分解（找出差异和共性）；归纳（造词）；**
    - 抽象带来的问题：抽象的好处就是找出共性、简化事物
        - 抽象造成意义模糊；抽象错误；抽象造成关键特征丢失；抽象过度

技术管理

- **能力模型**：格局；技术历史；抽象能力（从上到下）；深入思考能力；落地能力（执行力&项目管理）；
- **影响力的塑造**：关键时候能顶上；老板思维；空杯心态；持续改进；建言献策；
- **团队能力提升**
    - 不确定性与风险把控：需求；技术；人员；组织；历史遗留问题；
    - 以价值为中心的管理：技术；非功能性需求；业务价值；公司角度；“战略性投入”项目；研究性普惠性技术；避免陷入“无效忙碌”的状态
    - 团队培养
        - 技术能力（识人与培养）：初级、中级、高级、资深
        - 独立意识：每个人在自己所处的层次都是可“托付”的
        - 思维能力：通过一次次的讨论来言传身教；每个人在职场上工作，都是要养家糊口的，多去赋能他人、成就他人