AUST 分布式系统复习

——期末真传

anyunyeyi

2026 年 2 月 16 日

前言

2025 – 2026(1) 秋季课程, 教师朱晓娟.

anyunyeyi

2026 年 2 月 16 日

目录

第一章　章节复习

1.1　第 1 章 分布式架构

理解什么是分布式系统、为什么需要它、它的挑战和演进过程.

1. 定义: 组件分布在联网计算机上, 通过消息传递进行通信和协调的系统. 关键在于对用户透明 (像一台计算机).

2. 目标与优势: 并行性 (高性能)、容错性、物理分布、安全性/隔离. 优势包括: 可靠性、可伸缩性、安全性、局部性、容错性.

3. 核心挑战 (8 大错误认知): 这是重点! 必须理解每个“错误认知”的反面就是分布式系统的现实挑战.

    - 网络不可靠、延迟不为零、带宽有限、拓扑会变、有多个管理员、传输有成本、网络异构、网络安全需额外考虑.
    - 衍生出三大难题: 并发与局部错误、性能 scaling 的复杂性、时钟与顺序问题.

4. 设计挑战:

    - 透明性: 访问、位置、并发、复制、故障、移动、性能、扩展透明性. 理解其含义.
    - 开放性、安全性、可扩展性、故障处理、并发性.

5. 架构演进 (重点! 可能出简答或填空):

   - 一体 → 应用与数据分离 → 加入缓存 → 服务器集群 (负载均衡) → 数据库读写分离 → 反向代理与 CDN → 分布式数据库/分库分表 → 业务拆分 → 微服务.
   - 理解每一步解决的核心问题 (性能、可用性、安全性).

6. 八大错误认知的具体挑战 (必考要点):

   - 网络不可靠: 请求可能丢失、延迟、重复; 节点可能宕机、网络可能分区.
   - 延迟不为零: 网络通信永远存在延迟, 且延迟时间不确定.
   - 带宽有限: 网络传输能力有上限, 不能无成本地传输大量数据.
   - 拓扑会变: 节点可能动态加入、退出或失效, 网络结构不固定.
   - 传输有成本: 数据传输消耗网络资源, 产生带宽、存储等费用.
   - 网络非统一 (异构): 网络连接在性能、带宽、可靠性上存在差异.
   - 需考虑安全: 网络可能遭受窃听、篡改、中间人攻击等安全威胁.
   - 不止一个管理员: 分布式系统通常由多个团队或组织共同管理.

1.2　第 2 章 分布式应用服务拆分

理解如何将一个大应用拆分成微服务 (领域驱动设计 DDD).

1. 拆分原因: “大泥球” 架构导致性能、可用性、扩展性差.

2. 拆分思路 - 领域驱动设计 (DDD):

   - 领域 & 子域: 业务边界.

- 限界上下文 (Bounded Context): 划分业务和技术边界的关键“刀”, 定义了特定的业务语义环境.
- 核心概念:
  - 实体: 有唯一标识, 状态可变 (如订单、用户).
  - 值对象: 无标识, 通过属性值定义 (如地址、金额).
  - 聚合 & 聚合根: 一组关联对象的集合, 聚合根是入口和管理者. 一次事务只修改一个聚合.
  - 领域事件: 聚合内部操作产生的事件, 用于跨聚合/服务间的最终一致性通信 (解耦关键).

3. 分层架构:
   - 用户接口层: 与外界交互.
   - 应用层: 协调任务, 不含核心业务逻辑, 负责服务组合、编排、事务控制.
   - 领域层: 核心业务逻辑所在, 包含实体、值对象、领域服务.
   - 基础层: 提供技术支撑 (数据库、消息队列等).
4. DDD 应用案例: 学生选课系统

   这是一个将 DDD 从业务分析到架构映射的典型例子.
   - 业务流程: 学生选课申请 → 教务处审批 → 学生上课签到 → 生成签到明细.
   - 领域对象识别: 学生、课程、申请单、审批记录、签到记录 (实体); 课程 ID、学号 (值对象).
   - 聚合与限界上下文:
     - 选课申请聚合 (聚合根: 申请单, 包含学生、课程信息)

  - 课程审批聚合 (聚合根: 审批记录)
  - 课堂签到聚合 (聚合根: 课程, 包含签到列表)
- 映射到微服务: 每个聚合/限界上下文可对应一个独立的 “选课服务”、“审批服务”、“签到服务” .

1.3　第 3 章 分布式调用

理解微服务/分布式应用间如何找到彼此、如何调用、如何保障稳定.

1. 负载均衡 (LB):
   - 类型:DNS LB(区域级, 简单但有缓存问题)、硬件 LB(性能高、贵)、软件 LB(如 Nginx, 灵活、成本低).
   - 常见算法: 轮询、加权轮询、IP 哈希、最少连接.
2. API 网关: 系统的统一入口.
   - 功能: 路由、负载均衡、认证鉴权、限流熔断、协议转换、日志记录、链式处理 (责任链模式).
   - 关键技术原理:
     - 链式处理: 网关将请求处理流程分解为多个过滤器 (如鉴权、限流、日志), 形成一个处理链, 提高可扩展性和可维护性.
     - 异步请求: 采用非阻塞 I/O(如 Netty) 处理请求, 工作线程不阻塞等待响应, 可大幅提升吞吐量, 应对高并发场景.
3. 服务注册与发现:
   - 三个角色: 服务提供者 (注册 + 心跳)、服务消费者 (查询 + 缓存)、服务注册中心 (维护列表 + 通知).
   - 常见组件:Eureka, Nacos, Consul, ZooKeeper.

4. 服务容错三剑客 (重点! 理解区别与联系):

   - 熔断: 快速失败, 防止故障扩散. 状态: 关闭 → 打开 → 半开.
   - 降级: 牺牲非核心功能或提供兜底方案, 保证核心可用.
   - 限流: 控制流量, 防止系统被压垮.
   - 关系: 限流是预防, 熔断是故障发生时的保护, 降级是熔断后的应对策略.

5. RPC(远程过程调用):

   - 目标: 让远程调用像本地调用一样简单.
   - 核心过程 (必考!):
     (a) 动态代理: 客户端调用代理 (Stub).
     (b) 序列化: 将参数、方法名等转为字节流 (如 JSON, Protobuf).
     (c) 协议编码: 给数据包加“信封”(定义边界、长度等).
     (d) 网络传输: 通过 Socket 发送.
     (e) 服务端反序列化, 调用真实方法, 再将结果序列化返回.
   - 关键点: 屏蔽网络细节, 开发者专注业务.

1.4　第 4 章 分布式协同

这是理论核心章节, 涉及同步、时钟、互斥和选举.

1. 时钟同步:

   - 物理时钟问题: 各机器时钟不同步 (偏移、漂移).NTP 协议用于同步.
   - 逻辑时钟 (Lamport Clock): 解决事件排序问题, 不关心真实时间.

  - 规则: 本地事件 +1, 发送消息时带上时间戳, 接收时取 max(本地时钟, 消息时间戳)+1.
  - 实现了 happened-before 关系, 但 $C(a) < C(b)$ 不能反推 $a \to b$.
- 向量时钟: 可以准确捕捉因果关系. 每个进程维护一个向量, 更新规则更复杂, 可以判断事件是因果相关还是并发.
- 时钟同步小结:
  - 物理时钟同步 (如 NTP): 目标是让各机器时钟与 UTC 时间尽量接近, 解决“时间准确”问题.
  - 逻辑时钟 (Lamport): 目标是定义事件发生的偏序关系, 解决“事件顺序”问题, 但不保证反向推断因果关系.
  - 向量时钟: 扩展逻辑时钟, 可准确判断两个事件是因果相关还是并发.

2. 分布式互斥:

- 要求: 互斥性、无死锁、无饥饿、公平性.
- 三大类算法 (掌握思想与对比):
  - 集中式算法: 一个协调者, 简单但存在单点故障和瓶颈.
  - 分布式算法 (如 Ricart & Agrawala): 基于“全票通过”或“多数票通过”. 消息复杂度高 $(2 \cdot (N-1))$.
  - 令牌环算法: 令牌在逻辑环中传递, 持有者可访问资源. 公平但令牌丢失难恢复, 环变动开销大.
- 分布式锁: 是互斥的工程实现, 常用 Redis(SETNX) 或 ZooKeeper (有序临时节点) 实现.
- 性能优化: 分布式分段加锁

适用于高并发秒杀场景. 将临界资源 (如库存) 分成多个段 (如 1000 件库存分成 10 个库存段). 不同请求可以同时对不同的段进行加锁和扣减, 极大提升并发处理能力, 避免单一资源成为瓶颈.

3. 选举算法: 用于动态选出主节点 (Leader).

   - 霸道算法 (Bully):ID 大的胜出. 节点向所有 ID 高于它的节点发起选举, 无应答则自己成为 Leader.
   - 环算法 (Ring): 选举消息在环中传递, 收集所有存活节点 ID, 选出最大 ID 者.

1.5 第 5 章 分布式计算

1. MapReduce(批处理):

   - 核心思想: “分而治之” + “计算向数据靠拢”.
   - 编程模型: 只需实现 Map 和 Reduce 函数.
     - Map: $(k_1, v_1) \to list(k_2, v_2)$
     - Reduce: $(k_2, list(v_2)) \to list(k_3, v_3)$
   - 工作流程 (重点!):
     (a) Split: 输入数据被逻辑切分.
     (b) Map 阶段: 多个 Map 任务并行处理 Split.
     (c) Shuffle(关键!):Map 输出结果按照 key 进行分区、排序、合并 (Combine 可选), 然后发送给对应的 Reduce 任务.
       - Split vs Block:Split 是 MapReduce 的逻辑输入单元, 包含数据位置元信息;Block 是 HDFS 的物理存储单元 (默认 128MB). 一个 Split 可能包含多个 Block, 反之亦然.
       - Combine vs Merge:

* Combine(合并): 在 Map 端本地对相同 Key 的 Value 进行聚合 (如求和), 减少网络传输量.< “a”, 1 > 和 < “a”, 1 > 合并为 < “a”, 2 >.
* Merge(归并): 将多个已排序的文件合并成一个大文件, 不改变数据内容.< “a”, 1 > 和 < “a”, 1 > 归并为 < “a”, < $1, 1$ >>.

(d) Reduce 阶段:Reduce 任务对收到的数据进行归并计算, 输出最终结果.

- 组件:Client, JobTracker (Master), TaskTracker (Slave), Task (Map/Reduce).

2. 流计算:

- 核心理念: 数据的价值随时间流逝而降低, 需要实时处理.
- 与批处理 (Hadoop) 对比: 流处理低延迟, 批处理高吞吐.
- 处理流程: 数据实时采集 → 数据实时计算 → 实时查询服务.
- 框架代表:Storm(原生流式)、Spark Streaming(微批)、Flink(流批一体).

1.6　第 6 章 分布式存储

数据如何分布、扩展、保证可用.

1. 分布式存储系统根据数据类型选择:

- 结构化数据 (表结构): 使用分布式数据库 (如 MySQL 集群).
- 半结构化数据 (JSON/XML/Key-Value): 使用分布式缓存/键值系统 (如 Redis 集群).

- 非结构化数据 (文档、图片、视频): 使用分布式文件系统 (如 HDFS).

2. 扩展方式:

   - 垂直扩展 (Scale-up): 升级单机硬件 (CPU、内存、RAID). 有极限.
   - 水平扩展 (Scale-out): 增加机器节点 (分布式). 主流方式.

3. RAID 技术 (单机存储优化): 了解不同级别特点.

   - RAID0: 条带化, 性能好, 无冗余.
   - RAID1: 镜像, 冗余好, 容量利用率 50%.
   - RAID10/RAID01: 结合条带与镜像, 性能与冗余俱佳.

4. 数据库分库分表 (水平拆分核心):

   - 分表:
     - 水平分表: 按某个字段 (如 ID 取模、时间范围) 将数据分散到多个结构相同的表中. 需解决跨表查询、排序、事务问题.
     - 垂直分表: 将一张表的字段按业务拆分到不同表中.
   - 分库: 按业务模块、地域、数据冷热等将数据划分到不同数据库实例.

5. 主从复制与读写分离:

   - 目的: 提升读性能、做数据备份、高可用.
   - 原理: 主库写 Binlog, 从库 IO 线程拉取并写入 Relay Log, SQL 线程重放 SQL.
   - 复制方式:

  - 异步复制 (主流): 主库写完即返, 数据最终一致, 性能好.
  - 同步复制: 主库等从库写完才返, 强一致, 性能差.
- 读写分离: 写操作发往主库, 读操作发往从库 (通过中间件如 MyCat).

6. 高可用与扩容:

- 高可用方案: 主从切换 (手动/自动)、双主模式、使用 ZooKeeper 进行故障检测与选举.
- 水平扩容思路: 修改数据分片规则 (如从 id % 2 改为 id % 4), 并处理数据迁移和冗余问题.

1.7　第 7 章 分布式算法

1.7.1　基本概念

1. 核心问题: 在存在故障 (非拜占庭) 的异步网络中, 如何让多个节点就一个值达成一致?

2. CAP 定理 (几乎必考): 分布式系统无法同时满足以下三点:

- Consistency(一致性): 所有节点看到的数据相同.
- Availability(可用性): 每个请求都能得到响应.
- Partition tolerance(分区容错性): 网络分区发生时系统仍能工作.
- 结论:P 必须满足, 因此只能在 C 和 A 之间权衡.CP 系统 (如 ZooKeeper) 或 AP 系统 (如 Cassandra).

3. 一致性模型:

- 强一致性: 任何时刻读到的都是最新数据.

- 最终一致性: 一段时间后, 所有副本最终保持一致.

4. 经典算法 (理解其目标):

- Paxos: 分布式共识的基础算法, 难理解难实现.
  是分布式共识算法的基础, 其目标是让一组节点就一个值 (Value) 达成一致. 过程复杂, 包含提议者 (Proposer)、接受者 (Acceptor)、学习者 (Learner) 等角色, 通过“准备”和“接受”两个阶段, 并需要多数派 (Majority) 同意来确保一致. 以难懂著称.
- Raft: 为了可理解性而设计的共识算法, 通过选举 Leader、日志复制来达成一致. 角色:Leader, Follower, Candidate.
  为了工程实践而设计, 比 Paxos 更易理解. 它通过选举一个明确的 Leader(领导者) 来简化共识过程. 所有写请求都经过 Leader, 由 Leader 将日志复制到 Follower(跟随者), 同样需要多数派成功才能提交. 它的核心是:Leader 选举、日志复制、安全性.

1.7.2 Paxos 共识协议

Paxos 是一个分布式一致性算法, 目的是让一个分布式系统中的多个节点 (可能发生故障) 对一个值 (Value) 达成一致.

1. 三种核心角色

- Proposer(提案者): 提出议案 (Value).Value 可以是任何操作, 比如“设置变量 X=1”. 不同 Proposer 可以提出不同的 Value.
- Acceptor(批准者): 负责“投票”批准提案. 一个 Value 必须获得超过半数 (N/2+1)Acceptor 的批准才能被通过.
- Learner(学习者): 学习被批准的 Value. 它们从 Acceptor 那里读取结果, 一旦某个 Value 被多数派接受, Learner 就学习到这个最终值.

- 核心: 算法主要描述 Proposer 和 Acceptor 的交互过程.

2. 两个阶段 & 四个步骤 Paxos 通过两个阶段来确保一致性, 即使有节点故障或网络延迟也能工作.

   Phase 1: 准备阶段 (Prepare Phase)

   - P1a: Proposer 发送 Prepare 请求
     - Proposer 生成一个全局唯一且递增的提案编号 (ProposalID).
     - 向所有 Acceptor 发送 Prepare(N) 请求（只带编号 N, 不带具体 Value).
   - P1b: Acceptor 回复 Promise
     - Acceptor 收到后, 仅当收到的 ProposalID(N) 大于它之前承诺过的任何编号时, 它才会回复.
     - 回复一个 Promise(N, V) 消息, 其中 V 是它已接受的、编号最大的那个提案的 Value(如果还没接受过提案, V 为空).
     - 同时, Acceptor 承诺: 不再接受任何编号小于 N 的提案.

   Phase 2: 批准阶段 (Accept Phase)

   - P2a: Proposer 发送 Accept 请求
     * Proposer 等待, 直到收到超过半数 Acceptor 的 Promise 回复.
     * 关键决策点:

     (a) 如果所有回复的 Value 都为空, Proposer 就可以自由决定自己的 Value.

     (b) 如果有 Acceptor 已经接受过某个 Value(回复中 V 不为空), Proposer 必须采纳那个编号最大的 Value 作为自己的提案内容 (这是 Paxos 保证一致性的精髓).

    - Proposer 向 Acceptor 发送 Accept(N, V) 请求.
  - P2b: Acceptor 接受提案
    - Acceptor 收到 Accept 请求后, 仅当该请求的编号 N >= 它承诺过的最小编号 (即 Phase1 中的 N) 时, 它才会接受这个提案.
    - 它将 (N, V) 持久化存储, 并回复 Accepted(N, V).
  - P2c: Proposer 确认提交
    - 如果 Proposer 收到超过半数 Acceptor 的 Accepted 回复, 就意味着 Value 被正式选定 (Chosen).
    - Proposer 可以广播通知所有 Learner 这个被选定的 Value.

3. 核心思想与关键保证 (面试/简答重点)

   (a) “后者认同前者” 原则: 新的 Proposer(高编号) 在提出提案前, 必须学习可能已经被选定的旧值. 如果旧值已经形成 (被多数派接受), 新提案就必须沿用这个旧值; 如果旧值尚未形成, 新提案才能提出自己的值. 这保证了一旦一个值被选定, 后续所有被选定的值都将是同一个.

   (b) 多数派 (Quorum) 机制: 任何 “通过” 都需要超过半数的 Acceptor 同意. 这保证了即使部分节点故障, 系统仍能推进; 同时, 任意两个多数派集合之间必然有交集, 这个交集确保了信息 (如已接受的 Value) 的传递和一致性.

   (c) 提案编号的抢占与活性:Proposer 通过不断生成更大的提案编号来 “抢占” 提议权. 如果一个 Proposer 卡住或失败, 其他 Proposer 可以用更高的编号重启流程, 从而避免了死锁, 保证了算法的活性 (Liveness).

第二章　学习通作业

2.1　平时测验

题目 1. [CAP 定理的核心地位] 为什么 CAP 定理是分布式系统设计的核心理论?

解答.　CAP 定理指出, 在分布式系统中, 一致性 (Consistency)、可用性 (Availability) 和分区容错性 (Partition tolerance) 三者不可兼得, 最多只能同时满足两个. 它是分布式系统设计的核心理论, 原因如下:

1. 揭示了根本性权衡: 分布式系统必然面临网络分区 (P) 风险, 因此在设计时必须在一致性 (C) 和可用性 (A) 之间做出选择. 这为系统架构提供了根本性的设计指导.

2. 指导系统分类与选型:

   - CP 系统 (如 ZooKeeper、HBase): 保证强一致性, 在网络分区时可能拒绝服务.

   - AP 系统 (如 Cassandra、Dynamo): 保证高可用性, 在网络分区时可能返回旧数据.

   - CA 系统: 通常单机系统, 不适用于真正的分布式环境.

3. 帮助设计者明确业务优先级: 例如, 金融交易系统需优先保证 CP, 而社交网络 feed 流可接受 AP.CAP 定理迫使设计者思考业务对一致性

和可用性的容忍度.

题目 2. [服务拆分的利弊与粒度平衡] 论述分布式系统中服务拆分的利弊, 并结合实际场景说明如何平衡服务粒度.

解答. 利:

1. 技术异构: 不同服务可使用最适合的技术栈.
2. 独立部署与扩展: 可按需伸缩特定服务, 资源利用率高.
3. 高内聚低耦合: 服务边界清晰, 易于团队自治与维护.
4. 容错性提升: 单个服务故障不易引发系统整体崩溃.

弊:

1. 复杂度上升: 服务间通信、协调、监控、测试难度增加.
2. 网络延迟与可靠性: 依赖网络, 需处理超时、重试、熔断等.
3. 数据一致性挑战: 跨服务事务难以实现, 通常需最终一致性.
4. 运维成本增加: 需要服务发现、配置管理、链路追踪等基础设施.

如何平衡服务粒度:

1. 基于领域驱动设计 (DDD): 以限界上下文为边界划分服务. 例如, 电商系统中, “订单”和“库存”是不同限界上下文, 应拆分为独立服务.
2. 考虑团队结构: 两个披萨团队原则, 一个服务应由一个小团队 (5-9 人) 独立负责.
3. 评估通信开销: 服务间调用频率过高可能意味着拆分过细, 应考虑合并为聚合.

4. 实际场景示例: 在“在线教育平台”中, “课程服务”、“用户服务”、“支付服务”可按业务边界拆分; 而“课程”内部的“视频转码”和“元数据管理”若频繁交互, 可先作为一个服务, 待规模扩大再拆分.

题目 3. [Bully 算法选举流程与问题] 简述分布式系统中 Bully 算法的选举流程及其潜在问题.

解答. 选举流程:

1. 当任一节点发现协调者 (主节点) 失效时, 发起选举.
2. 节点向所有 ID 比自己大的节点发送 ELECTION 消息.
3. 若无任何更高 ID 节点响应, 则该节点获胜, 成为新协调者, 并向所有更低 ID 节点发送 COORDINATOR 消息.
4. 若有更高 ID 节点响应, 则该节点退出选举, 由响应者接管选举过程.

潜在问题:

1. 消息风暴: 多个节点同时发起选举会产生大量消息.
2. 假设网络可靠: 算法假设消息不会丢失, 但在不可靠网络中可能导致多个协调者.
3. 依赖于节点 ID 大小:ID 最大的节点不一定是性能最优或最合适的协调者.
4. 恢复节点问题: 故障恢复的旧协调者可能干扰新协调者.

题目 4. [分布式系统优势与特征迁移] 分布式系统相比集中式系统有哪些核心优势? 在迁移过程中, 如何体现分布式系统的“自治性”和“透明性”特征?

解答. 核心优势:

1. 可扩展性: 可通过增加节点水平扩展性能与容量.
2. 高可用性: 通过冗余和副本, 容忍单点故障.
3. 性能: 并行处理与数据局部性可降低延迟、提高吞吐量.
4. 地理分布: 可部署在靠近用户的位置, 减少延迟.

迁移过程中的体现:

1. 自治性: 每个节点/服务可独立管理自己的资源、部署和升级. 例如, 在迁移中, 可以先独立迁移“用户服务”而不影响“订单服务”.
2. 透明性:
   - 访问透明: 迁移后, 用户仍通过相同的 API 访问服务, 无需感知服务已分布式部署.
   - 位置透明: 用户无需知道服务具体运行在哪台服务器上, 通过服务发现机制自动路由.
   - 故障透明: 当某个节点故障时, 系统能自动将请求转移到健康节点, 用户无感知.

题目 5. [微服务拆分原则与数据一致性] 某在线教育平台需要将单体应用拆分为微服务, 以支持独立部署和扩展.

(1) 服务拆分应遵循哪些原则? 请结合领域驱动设计 (DDD) 说明.

(2) 拆分后如何解决服务间通信的“数据一致性”问题?

解答.

(1) (a) 基于限界上下文拆分: 识别核心业务领域, 如“课程管理”、“用户中心”、“订单支付”、“学习进度”等, 每个上下文对应一个服务.

(b) 单一职责与高内聚: 每个服务只负责一个明确的业务能力, 例如“课程搜索” 和 “课程推荐” 若逻辑紧密, 可属于同一 “课程服务” .

(c) 围绕领域模型设计: 识别聚合根 (如 “课程”、“用户”), 确保聚合内的强一致性, 聚合间通过领域事件实现最终一致性.

(d) 团队自治对齐: 服务边界应与团队组织架构对齐, 便于独立开发、部署和运维.

(2) (a) 最终一致性模式: 放弃分布式强一致事务, 采用以下模式:

- Saga 模式: 将长事务拆分为多个本地事务, 每个事务触发下一个, 失败时触发补偿事务.
- 事件驱动架构: 服务 A 完成操作后发布领域事件, 服务 B 订阅该事件并更新自身状态. 例如, “订单支付成功” 事件触发“课程开通” 服务.

(b) 使用消息队列: 确保事件可靠传递, 实现异步解耦.

(c) 幂等性设计: 消费者对重复消息的处理结果保持一致, 防止重复操作.

题目 6. [Bully 算法选举中的并发与分区问题] 一个分布式日志系统需要选举一个主节点 (Leader) 来协调日志写入.

(1) 如果采用 Bully 算法进行选举, 当多个节点同时发起选举时会发生什么问题? 如何解决?

(2) 如果网络分区导致部分节点无法通信, 选举算法需满足哪些条件才能保证一致性?

解答.

(1) 问题: 会产生消息风暴, 每个节点都向更高 ID 节点发送选举消息, 网络流量激增, 且可能产生多个协调者 (如果消息延迟导致部分节点未收到更高 ID 的响应).

解决: 引入随机等待时间 (随机退避) 再发起选举, 减少冲突; 或采用更稳定的选举算法 (如 Raft).

(2) 必须保证分区内最多只有一个领导者. 在 CAP 定理下, 网络分区时我们通常选择 CP(保证一致性), 即分区中多数派 (超过半数节点) 的一侧才能成功选举出领导者, 另一分区无法提供服务 (不可用). 这需要算法具有多数派投票机制 (如 Raft、Paxos), 确保不会出现 "脑裂" (两个分区各选一个主节点).

题目 7. [分布式锁实现] 一个分布式缓存系统需要确保多个节点对共享资源的互斥访问.

(1) 在分布式环境中, 为什么简单的 "文件锁" 无法满足互斥需求?

(2) 如何利用 Redis 实现分布式锁? 请描述关键步骤.

解答.

(1) 因为文件锁通常只在单机文件系统上有效. 在分布式环境中, 多个节点可能在不同机器上, 无法通过同一个本地文件系统进行锁同步. 即使使用网络文件系统 (NFS), 也会遇到性能、可靠性和网络分区等问题, 无法保证强一致的互斥.

(2) (a) 加锁: 使用 SET key random_value NX PX 30000 命令.NX 表示仅当 key 不存在时设置, PX 设置过期时间 (防止死锁).

random_value 应为全局唯一标识 (如 UUID), 用于安全释放锁.

(b) 业务操作: 获取锁成功后, 执行临界区业务逻辑.

(c) 释放锁: 使用 Lua 脚本, 先比较 key 对应的值是否与加锁时设置的 random_value 相等, 相等则删除 key. 必须保证原子性, 防止误删其他客户端的锁.

(d) 锁续期 (可选): 如果业务执行时间可能超过锁过期时间, 需使用守护线程定时续期.

题目 8. [CAP 权衡与场景分析] 某分布式系统采用 CAP 定理指导设计, 需在一致性 (C)、可用性 (A)、分区容错性 (P) 之间权衡.

(1) 如果系统优先保证 “一致性” 和 “分区容错性” , 需牺牲什么? 请举例说明适用场景.

(2) 如果系统优先保证 “可用性” 和 “分区容错性” , 需牺牲什么? 请举例说明适用场景.

解答.

(1) 牺牲可用性 (A). 当发生网络分区时, 为了保证数据一致性, 系统可能拒绝写操作或返回错误, 直到分区恢复.

适用场景: 金融核心交易系统 (如转账)、火车票售票系统. 例如, 银行跨行转账时, 必须保证两边账户金额准确一致, 宁愿暂时拒绝服务也不能出现差错.

(2) 牺牲强一致性 (C), 接受最终一致性. 当发生网络分区时, 系统继续提供服务, 但不同分区可能返回新旧不同的数据.

适用场景: 社交网络、电商商品详情页、实时评论系统. 例如, 微博 Feed 流, 容忍短暂看到旧数据, 但系统始终保持高可用.

2.2　作业 2

题目 9. 物联网数据分库分表与读写分离设计

案例背景: 某城市部署了 10 万 + 物联网设备, 实时上传环境监测数据 (如温湿度、PM2.5 等), 日均产生 TB 级数据, 需通过分库分表与读写分离实现高效存储与查询.

(1) 简述分库分表的基本原理, 并说明垂直分库与水平分表在物联网数据场景中的适用性.

(2) 若采用水平分表策略, 如何设计分片键 (Sharding Key) 以平衡数据分布与查询效率? 请结合设备 ID 或地理位置举例说明.

解答.

(1) 基本原理: 通过将数据分散到多个数据库或表中, 以突破单机存储和性能瓶颈.

垂直分库: 按业务维度将不同表拆分到不同数据库. 在物联网场景中, 可将设备元数据 (静态信息) 和时序数据 (动态传感器数据) 分库存储, 因为两者访问模式不同.

水平分表: 按某个键将同一表的数据分散到多个结构相同的表中. 在物联网场景中, 对海量的时序数据表进行水平分表是必须的, 例如按时间范围 (每月一张表) 或设备 ID 哈希进行分片.

(2) 方案一: 按设备 ID 哈希取模.shard = hash(device_id) % N. 优点是数据分布均匀, 避免热点. 但按时间范围查询时, 需跨所有分片查询, 效率低.

方案二: 按时间范围分区. 例如每月一张表. 优点是对按时间范围的查询非常高效 (直接定位到单表). 但可能导致数据分布不均 (新设备数据集中在新表), 且单个设备的数据散落在多张表, 查询单个设备历史需跨表.

综合方案: 联合分区键. 使用 (device_id, timestamp) 作为复合键, 先按设备 ID 哈希分库/分表, 再在表内按时间排序. 或使用时间前缀 (如

年月) 作为分表依据, 再按设备 ID 哈希进行二次分区. 需要根据查询模式权衡.

题目 10. 电商秒杀系统的分布式计算与负载均衡

案例背景: 某电商平台在“双十一”期间需处理百万级并发秒杀请求.

(1) 解释 MapReduce 框架如何将秒杀订单分片处理, 并通过 Reduce 阶段聚合库存扣减结果.

(2) 比较静态负载均衡 (如轮询) 与动态负载均衡 (如最小连接数) 在秒杀场景中的适用性, 并提出优化策略.

(3) 设计一个支持弹性扩容的秒杀任务队列, 要求说明如何通过消息中间件 (如 Kafka) 实现削峰填谷, 并结合 Redis 缓存预扣库存.

解答.

(1) Map 阶段: 多个 Map 任务并行处理来自不同渠道的秒杀请求日志/消息. 每个 Map 任务读取一部分请求数据, 输出键值对 (商品 ID, 扣减数量). 例如, 对于每个请求, Map 输出 (SKU123, 1).

Shuffle 阶段: 将相同商品 ID 的扣减数量发送到同一个 Reduce 任务.

Reduce 阶段: 每个 Reduce 任务接收一个商品 ID 的所有扣减请求, 进行累加, 得到该商品总扣减量. 然后与初始库存比较, 判断是否超卖, 并输出最终扣减结果 (商品 ID, 成功扣减数量).

(2) 轮询: 简单, 但无法考虑服务器当前负载, 可能导致部分服务器过载.

最小连接数: 更优, 能将新请求导向当前压力最小的服务器, 更适应秒杀场景中请求突发、负载不均的特点.

优化策略:

(a) 权重动态调整: 根据服务器实时 CPU、内存、网络 IO 动态调整权重.

(b) 基于业务分片: 将商品库存进行分段加锁 (如 1000 件库存分成 10 段), 不同段路由到不同服务器处理, 提升并行度.

(c) 队列缓冲: 在负载均衡器后设置队列, 平滑突发流量, 配合限流机制.

(3) (a) 削峰填谷: 所有秒杀请求先进入 Kafka 消息队列, Kafka 高吞吐、可持久化, 能承受瞬间洪峰. 后端服务作为消费者, 按自身处理能力从队列中拉取请求, 实现异步处理.

(b) 预扣库存: 使用 Redis 缓存商品库存. 消费者从 Kafka 取出订单后, 通过 Redis 原子操作 (DECRBY 或 Lua 脚本) 预扣减库存. 扣减成功后再生成订单, 写入数据库. 扣减失败则直接返回秒杀失败.

(c) 弹性扩容:Kafka 分区数量可预先设置较多, 消费者组可以动态增加实例, 以提高并发处理能力. 结合容器化技术 (如 K8s), 可根据队列堆积长度自动扩容消费者实例.

题目 11. 某云服务商需为 AI 训练任务动态分配 GPU 资源.

(1) 列举三种关键资源指标, 并说明其对 GPU 调度策略的影响.

(2) 比较容器化 (Docker) 与虚拟机 (VM) 在资源隔离与启动速度上的差异, 分析 AI 训练场景的优劣势.

(3) 设计一个基于优先级队列的抢占式调度算法, 要求说明如何平衡突发的高优先级任务与已分配的低优先级任务.

解答.

(1) (a) GPU 显存占用:AI 训练任务对显存需求大, 调度器需确保节点有足够显存, 否则任务会失败. 影响装箱 (bin packing) 策略.

(b) GPU 算力利用率 (CUDA 核心使用率): 高算力任务可能需要独占 GPU, 低算力任务可共享 (如 MIG 技术). 影响资源共享与隔离策略.

(c) 任务预计运行时长: 短任务可快速周转资源, 长任务需考虑资源预留. 影响调度算法 (如是否支持抢占).

(2) 资源隔离:VM 通过 Hypervisor 虚拟化硬件, 隔离性更强、更安全;

Docker 共享主机内核, 隔离性较弱 (但可通过命名空间、cgroups 限制).

启动速度:Docker 容器秒级启动, VM 需要分钟级启动.

AI 训练场景分析:

- 优势:Docker 启动快, 更适合快速部署和弹性伸缩; 镜像更轻量, 便于分发 AI 环境.
- 劣势: 若对安全隔离要求极高 (如多租户), VM 更合适;GPU 直通或虚拟化技术在 VM 中更成熟.
- 结论: 通常 AI 训练平台采用容器化 (Docker/Kubernetes), 配合 GPU 设备插件和调度器 (如 K8s Device Plugin), 实现高效资源管理和快速启动.

(3) (a) 优先级定义: 任务有优先级标签 (如高、中、低), 高优先级任务可抢占低优先级任务资源.

(b) 抢占流程:

- 当高优先级任务到达且无空闲资源时, 调度器选择运行中的低优先级任务进行驱逐 (先驱逐完成度低的, 或已运行时间长的).
- 被抢占的任务保存检查点 (checkpoint) 到持久存储, 释放 GPU 资源.

- 高优先级任务获得资源后开始执行.
- 资源再次空闲时, 被抢占的任务从检查点恢复执行.

(c) 平衡策略: 设置抢占成本阈值, 若预计的保存/恢复检查点时间过长, 可能放弃抢占; 或为低优先级任务设置最小保证运行时间, 避免饿死.

题目 12. 某银行在全国有多个分行, 采用分布式系统处理跨分行转账业务.

(1) 当客户 A 在分行 1 发起转账至客户 B 在分行 2 的账户时, 如何通过两阶段提交 (2PC) 保证事务一致性? 请描述协议流程.

(2) 若分行 1 与分行 2 间网络分区, 系统优先保证一致性 (C) 还是可用性 (A)? 请结合 CAP 定理分析决策依据及对用户体验的影响.

(3) 假设需支持实时余额查询功能, 请设计一种基于版本向量的跨分行余额同步机制, 并说明如何通过最终一致性解决查询延迟问题.

解答.

(1) (a) 准备阶段 (投票): 事务协调者 (TC) 向所有参与者 (分行 1、分行 2) 发送“准备请求”. 每个参与者执行本地事务 (扣款/加款) 但不提交, 写入 Undo/Redo 日志, 然后向 TC 返回“同意”或“中止”.

(b) 提交阶段 (执行):

- 若所有参与者都同意, TC 发送“提交请求”, 参与者正式提交事务, 释放锁.
- 若有任一参与者反对或超时, TC 发送“回滚请求”, 参与者利用日志回滚.

(c) 问题:2PC 是阻塞协议, 在协调者或参与者故障时可能造成资源长时间锁定.

(2) 银行转账系统必须优先保证一致性 (CP). 依据 CAP 定理, 在网络分区发生时, 为了保证数据的强一致性 (账户余额准确), 系统应选择牺牲可用性, 即暂停跨分区转账服务, 直到网络恢复. 否则可能出现两边余额不一致 (如 A 扣款 B 未到账) 的严重错误.

对用户体验的影响: 用户转账请求会失败或超时, 体验较差, 但资金安全得到保障.

(3) 版本向量: 每个分行的账户副本维护一个版本向量, 记录自己以及其他分行副本的更新版本号.

同步机制:

(a) 当分行 1 更新账户 A 余额时, 增加本地版本号, 并将更新 (新余额、新版本向量) 异步发送给其他分行.

(b) 分行 2 收到同步消息时, 比较版本向量: 若分行 2 的版本低于分行 1, 则接受更新; 若存在冲突 (如双方版本都高于对方在某些分行的记录), 则需要业务逻辑解决冲突 (如保留最新时间戳的更新, 或人工干预).

最终一致性解决查询延迟: 客户查询余额时, 可能读到稍旧的数据 (例如从本地副本读取, 而最新更新尚未同步过来). 系统可通过以下方式缓解:

- 读己之写: 保证用户总能读到自己的最新更新 (通过路由到同一分行).
- 版本标记: 返回数据时附带版本号, 应用可判断数据新鲜度, 必要时可等待同步.

2.3　作业 3

题目 13. [Paxos 算法] 在分布式系统中, 多个节点需要就某个值 (例如配置参数或日志序号) 达成一致.Paxos 是一种经典的共识算法. 假设有一个由 5 个节点组成的集群 (编号为 A、B、C、D、E), 它们正在使用 Paxos 算法对一个值进行决议. 当前没有网络分区, 所有节点均可互相通信. 请回答以下问题:

(1) Paxos 算法的主要目标是什么? 在什么条件下能保证安全性?

(2) Paxos 中有哪两类主要角色? 请写出它们的名称, 并简要说明各自的作用.

(3) 假设 Proposer A 发起提案, 编号为 10. 它向所有节点发送 Prepare 请求. 如果它收到了来自 B、C、D 的 Promise 回复 (共 3 个), 且这些回复中都没有已接受的值, 那么 A 接下来可以做什么? 它是否已经成功让整个集群就该值达成共识? 为什么?

(4) 为什么 Paxos 要求 “多数派 (majority)” 同意, 而不是 “全部节点” 同意? 这样做有什么好处?

解答.

(1) Paxos 算法的主要目标是让分布式系统中的多个节点对一个值 (value) 达成一致 (共识).

安全性保证条件: 在异步网络模型中, 只要超过半数 (多数派) 的节点存活且能互相通信, Paxos 就能保证安全性 (即一旦一个值被选定, 就不会改变, 且所有节点最终都能学习到这个值).

(2)
- Proposer(提案者): 提出提案 (包括唯一递增的提案编号和值). 目标是让提案被批准.

- Acceptor(批准者): 对提案进行投票. 一个提案必须获得超过半数 Acceptor 的接受才能被选定.Acceptor 需遵守承诺: 不再接受编号小于已承诺编号的提案.

(3) A 接下来可以进入批准阶段 (Accept Phase), 向所有节点发送 Accept(10, v) 请求 (v 可由 A 自由选择, 因为回复中无已接受值).

尚未成功达成共识. 因为共识达成的标志是提案获得超过半数 Acceptor 的接受 (accepted). 目前只收到 Promise 回复, 仅表示 A 获得了提交资格, 还需等待 Accept 请求的确认.

(4)
- 好处 1: 容忍节点故障——只要多数派存活, 系统仍可工作.
- 好处 2: 防止脑裂——网络分区时, 最多只有一个分区能形成多数派, 避免不一致决策.
- 好处 3: 降低延迟——无需等待所有节点响应, 提高效率.

题目 14. [分布式资源调度] 某大学实验室搭建了一个由 5 台服务器组成的分布式计算集群, 用于运行学生的数据分析作业. 每台服务器有 8 个 CPU 核心和 16 GB 内存. 系统采用一个简单的集中式调度器来分配任务. 现在有三个作业等待调度:

- 作业 A: 需要 4 核 + 8 GB 内存, 运行时间约 10 分钟;
- 作业 B: 需要 2 核 + 4 GB 内存, 运行时间约 30 分钟;
- 作业 C: 需要 6 核 + 12 GB 内存, 运行时间约 5 分钟.

假设当前所有服务器都空闲. 请回答以下问题:

(1) 什么是分布式资源调度? 它的主要目标是什么?

(2) 如果调度器采用 “先来先服务” (FCFS) 策略, 且作业到达顺序为 A → B → C, 那么作业 C 能否被立即调度? 为什么?

(3) 如果改用 “最小资源占用优先” (即优先调度所需资源较少的作业), 调度顺序会变成怎样? 这种策略可能带来什么好处和问题?

解答.

(1) 分布式资源调度是在多机集群中, 根据任务需求和资源状况, 将任务分配到合适机器的过程.

主要目标: 提高资源利用率、系统吞吐量, 保证公平性, 满足任务资源需求.

(2) 不能立即调度. 因为 FCFS 按到达顺序调度, 作业 C 必须等待作业 A 和 B 先被分配. 虽然集群资源充足 (5 台服务器, 每台 8 核 16GB), 但调度器会依次尝试分配 A、B, 然后才轮到 C. 因此作业 C 需要等待.

(3)
- 调度顺序:B(2 核 4GB)→ A(4 核 8GB)→ C(6 核 12GB).
- 好处: 提高资源利用率, 减少小作业等待时间, 降低平均周转时间.
- 问题: 可能导致大作业饥饿 (若持续有小作业到达), 资源碎片化可能影响大作业调度.

题目 15. [秒杀系统] 某电商平台计划在 “618” 期间上线一个秒杀活动:

- 商品: 限量 100 件 iPhone;
- 开始时间:2026 年 6 月 18 日 10:00:00;
- 预计瞬时并发用户数:10 万 +;
- 系统架构包括:Web 前端、应用服务器、MySQL 数据库 (库存字段 stock 初始值为 100).

开发团队最初采用如下简单逻辑处理下单请求: – 伪代码

```
BEGIN;
SELECT stock FROM products WHERE id = 1;
-- 假设 iPhone 的 id=1
IF stock > 0 THEN
    UPDATE products SET stock = stock - 1 WHERE id = 1;
    INSERT INTO orders (...) VALUES (...);
    COMMIT;
ELSE
    ROLLBACK;
END IF;
```

但在压力测试中发现: 实际生成订单数远超 100 单, 出现 “超卖” 问题. 请回答以下问题:

(1) 简述 “超卖” 现象在此场景中产生的根本原因.

(2) 为防止超卖, 可对上述数据库操作做哪些改进? 请写出一条能保证原子性的 SQL 语句 (或事务逻辑).

(3) 仅靠数据库行锁 (如使用 UPDATE ... WHERE stock $>$ 0) 在 10 万 QPS 下可能面临什么瓶颈? 请指出两个主要问题.

(4) 为应对高并发秒杀, 常采用 “缓存 + 异步队列” 架构. 请简要说明:

  (a) 为什么先用 Redis 缓存库存?

  (b) 为什么将真实下单请求放入消息队列异步处理?

解答.

(1) 根本原因: 多个并发事务同时执行 “读取库存 $\rightarrow$ 判断 $>0\rightarrow$ 扣减库存” 流程, 由于该流程非原子操作, 事务交错执行可能导致多个事务读到同一库存值且都认为可扣减, 造成库存扣减次数超过实际库存量.

(2) 使用原子更新语句, 并在事务中判断影响行数:

```
UPDATE products SET stock = stock - 1
WHERE id = 1
AND stock > 0;
```

若该语句影响行数为 1, 则扣减成功, 可插入订单; 否则失败.

(3) (a) 数据库连接与性能瓶颈: 单机 MySQL 难以承受 10 万 QPS, 连接数可能耗尽, 响应延迟激增.

(b) 行锁竞争激烈: 所有请求竞争同一行锁, 导致大量事务阻塞串行化, 吞吐量急剧下降.

(4)
- Redis 基于内存, 读写性能极高 (十万级 QPS), 可将库存检查与扣减操作前置, 避免数据库被压垮.
- 削峰填谷: 队列缓冲瞬时高并发请求, 后端按处理能力消费; 解耦与快速响应: 用户请求入队后立即返回, 后端异步处理订单, 提升用户体验和系统可伸缩性.

题目 16. [RPC 实现] 某同学在学习分布式系统时, 尝试使用 RPC 实现一个简单的“天气查询服务”. 客户端运行在机器 A 上, 服务器运行在机器 B 上. 他使用某种 RPC 框架 (如 gRPC 或自定义简易 RPC) 实现如下功能:

- 客户端调用 get_temperature(city: string) → float;
- 服务器根据城市名返回当前温度 (模拟值);
- 网络通信基于 TCP, 参数和返回值通过序列化传输.
- 但在测试过程中, 他遇到了以下问题:

- 客户端调用后长时间无响应;
- 有时收到错误: “无法解析返回值”;
- 当服务器宕机后重启, 客户端再次调用仍然失败.

请回答以下问题:

(1) 简述远程过程调用 (RPC) 的基本思想, 并说明它如何“隐藏”网络通信的复杂性.

(2) RPC 调用过程中通常涉及哪几个关键步骤? 请按顺序列出至少 4 个.

(3) 针对上述三个问题, 分别分析其最可能的原因 (每个问题限 1 条原因).

解答.

(1) RPC 的基本思想是使远程函数调用像本地调用一样简单. 它通过客户端存根 (stub) 和服务器存根隐藏复杂性: 客户端存根将调用序列化并发送, 服务器存根接收并反序列化、调用实际函数, 再将结果序列化返回. 程序员无需关心网络传输、序列化等细节.

(2) (a) 客户端调用本地存根方法.

(b) 客户端存根将方法名、参数序列化成字节流.

(c) 通过网络将字节流发送到服务器.

(d) 服务器存根反序列化请求, 调用真实服务方法.

(后续步骤: 服务器序列化结果、返回、客户端反序列化结果.)

(3)
- 长时间无响应: 网络超时或服务器处理超时, 客户端未设置合理超时时间.
- “无法解析返回值”: 客户端与服务器序列化/反序列化协议不一致 (如数据结构版本不匹配).

- 服务器重启后客户端调用仍失败: 客户端存根未检测连接断开并重连 (如连接池维持旧连接).

题目 17. [MySQL 主从复制实验] 某同学在单台 Linux 服务器上进行 MySQL 主从复制实验, 主库监听端口 3306, 从库监听端口 3307. 他完成了以下操作:

(1) 在主库配置文件中添加:

```
[mysqld]
server-id = 1
log-bin = mysql-bin
```

(2) 创建复制用户:

```
CREATE USER 'repl'@'localhost' IDENTIFIED
BY '123456';
GRANT REPLICATION SLAVE ON . TO 'repl'
@'localhost';
执行 SHOW MASTER STATUS; 得到:
File: mysql-bin.000002
Position: 156
```

(3) 修改从库的配置文件 my.cnf

```
————————(1)————————
————————(2)————————
```

(4) 在从库执行:

```
CHANGE MASTER TO
    MASTER_HOST=____(3)______,
    MASTER_PORT=____(4)______,
    MASTER_USER='repl',
    MASTER_PASSWORD='123456',
    MASTER_LOG_FILE=____(5)_____,
    MASTER_LOG_POS=156;
    START SLAVE;
```

(5) 如果从库在连接主库 (3306) 时, 使用 127.0.0.1, 会出现什么结果? 应该如何修改?

解答.

(3)

```
(1) server-id = 2
(2) log-bin = mysql-bin
# 可选，通常从库也开启以便级联复制
```

(4)

```
MASTER_HOST = (3) 127.0.0.1,
# 或 localhost
MASTER_PORT = (4) 3306,
MASTER_LOG_FILE = (5) mysql-bin.000002,
```

(5)

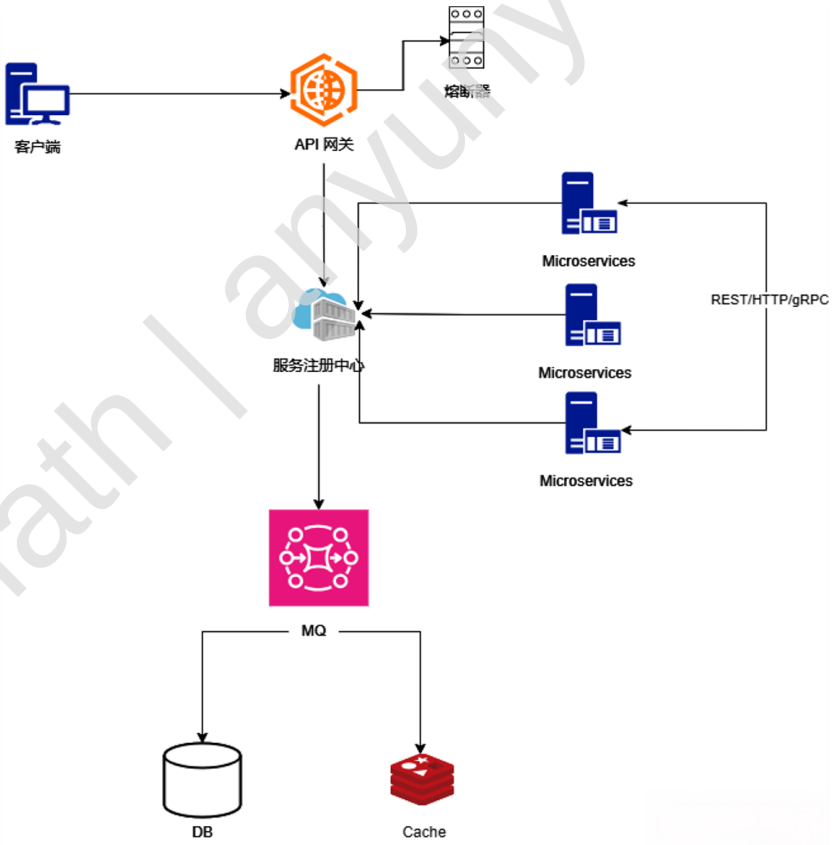
- 结果: 正常情况下可以连接, 因为主从在同一机器, 且主库监听 3306、从库监听 3307, 端口不冲突.

- 修改: 若连接失败, 需检查:
  (a) 主库 bind-address 是否绑定了 127.0.0.1;
  (b) 复制用户'repl'@'localhost' 是否允许从 127.0.0.1 连接 (或改为'repl'@'127.0.0.1');
  (c) 防火墙是否放行 3306 端口.

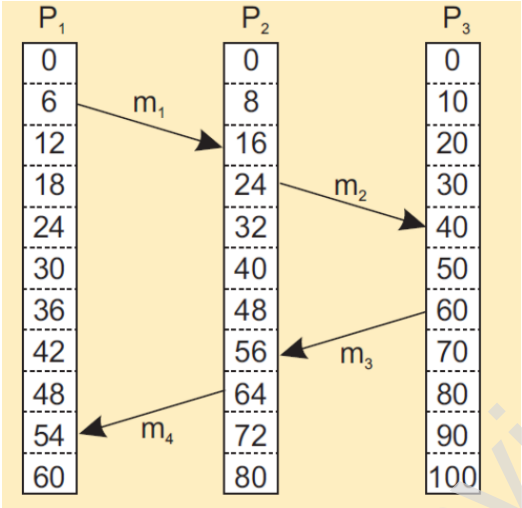
2.4　重修作业

题目 18.　画一个分布式系统架构图, 标注服务注册、网关、熔断.

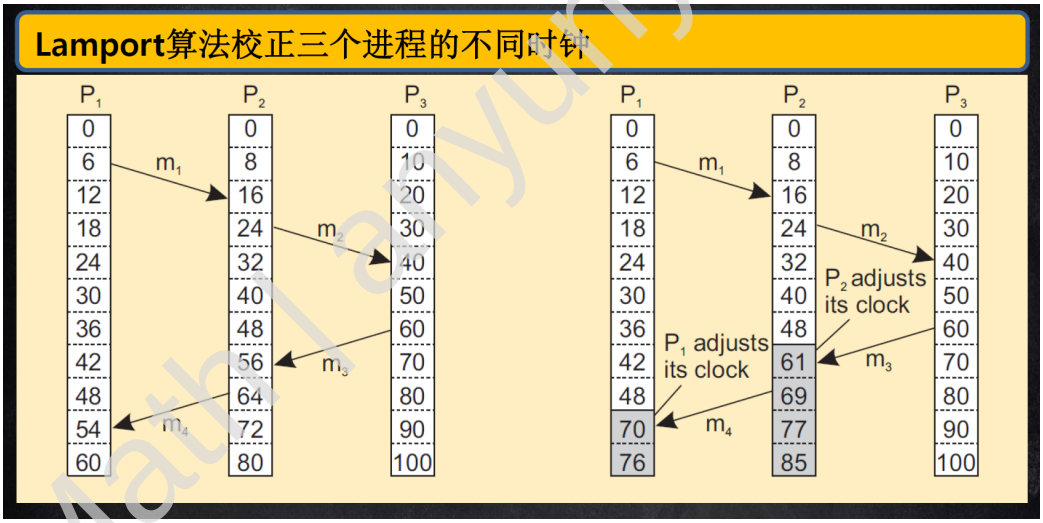
解答.



题目 19.　假设如下图所示三个进程需要进行时间同步, 请简述 Lamport 算法校正三个进程的不同时钟过程, 并画出校正后的进程计数.



解答.



第三章　复习策略

3.1　作业问题

1. 问题 1 (关于 CAP 定理): 你的课件未直接讲述 CAP 定理. 因此, 如果考试出现, 这可能是老师补充的内容. 你可以在答案中结合 “分布式系统的挑战” (网络不可靠、分区) 和 “一致性需求” 来阐述类似思想. 核心是理解 “网络分区发生时, 需要在一致性和可用性之间权衡” .

2. 问题 3、问题 6(1): 关于 Bully 算法, 你的课件 (第 4 章 PDF) 中有详细描述和图示. 请重点复习那个流程.

3. 问题 5(2)、问题 7、问题 12(1): 关于数据一致性和事务, 你的课件提到了:

   - 分布式事务: 通过 “事务协调器” 分两阶段处理 (第 1 章).
   - 最终一致性: 通过 “领域事件” 实现 (第 2 章).
   - 分布式锁: 用 Redis 或 ZooKeeper 实现 (第 4 章 PPT).
   - 回答时应紧扣这些知识点, 而不是大谈 Paxos.

4. 问题 8: 这道题明确关于 CAP 定理. 既然课件未讲, 备考时你需要记住这个基本结论:P(分区容错) 必须保证, 因此只能在 C 和 A 之间选.CP 系统 (如 ZooKeeper) 保证一致性, AP 系统 (如很多 NoSQL) 保证可

用性. 结合课件中“错误认知: 网络是可靠的”来理解“分区”的必然性.

5. 问题 12(2): 这是一个典型的 CAP 场景. 答案就按 CP(银行系统必须保证一致性, 牺牲可用性) 来回答.

3.2　要点精析

1. 关于 CAP 定理 (题 1, 8, 12(2)):
   - 现状: 课件未直接讲授, 但作业和复习资料中频繁出现, 考试概率极高.
   - 策略: 必须掌握核心结论:P(分区容错) 是分布式系统基础, 因此实际在 C 和 A 间权衡.
     - CP 系统 (如 ZooKeeper): 保证强一致性, 分区时可能拒绝服务. 适用场景: 金融交易、账号余额.
     - AP 系统 (如 Cassandra): 保证高可用性, 分区时可能返回旧数据. 适用场景: 社交 Feed、商品缓存.
2. 关于 Bully 选举算法 (题 3, 6(1)):
   - 复习依据: 对照课件第 4 章 PDF 第 47-48 页的流程图.
   - 必答要点: 流程 (谁发起、向谁发 ELECTION、如何胜出、如何宣布), 以及缺点 (消息风暴、依赖 ID 大小、网络不可靠假设).
3. 关于数据一致性与事务 (题 5(2), 7, 12(1)):
   - 紧扣课件: 回答时必须使用课件中的概念:
     - 分布式事务: 参考第 1 章“事务协调器”两阶段处理思想.
     - 最终一致性: 参考第 2 章通过“领域事件”异步通信实现.

  - 分布式锁: 参考第 4 章 PPT, 用 Redis(SETNX) 或 ZooKeeper(有序节点) 实现.

4. 关于 RPC 过程 (题 16):

   - 这是绝对核心: 必须能默写动态代理 → 序列化 → 协议编码 → 网络传输 → 反序列化 → 返回结果的全过程. 作业中的问题 (超时、解析错误、连接失败) 都是对此过程各环节的考察.

3.3　备考要点

3.3.1　第一级: 必考且必会

1. 第 3 章分布式调用:RPC 全过程、服务治理三剑客 (熔断/降级/限流) 的区别与联系、API 网关核心功能.

2. 第 4 章分布式协同:Lamport 逻辑时钟规则、三大互斥算法思想与对比、Bully 选举算法流程.

3. 第 1 章分布式架构: 八大错误认知 (现实挑战)、架构演进历程与每一步的目的.

4. 第 6 章分布式存储: 分库分表 (垂直/水平) 策略、主从复制 (异步) 原理与读写分离架构.

3.3.2　高频理解与应用

1. 第 2 章服务拆分: 能用 DDD 概念 (实体/值对象/聚合根/限界上下文) 分析简单案例 (如学生选课).

2. 第 5 章分布式计算:MapReduce 的 Shuffle 过程、流计算与批处理的核心理念对比.

3. 跨章节综合: 能将 “服务拆分 → 服务调用 → 容错” 串联, 或将 “数据分片 → 复制 → 一致性挑战” 串联.

3.3.3 第三级: 了解与记忆

1. RAID 级别 (0, 1, 10)、向量时钟作用、负载均衡算法类型.