CEX 面经整理

1. jraft的理解/高可用方案你了解有哪些/你们的撮合服务是如何实现高可用的

官方文档：https://www.sofastack.tech/projects/sofa-jraft/overview/

JRaft 是一个基于 Raft 共识算法的高性能 Java 实现，由阿里巴巴开源。Raft 是一种分布式一致性算法，旨在简化理解和实现，同时提供与 Paxos 相当的强一致性保证。JRaft 在 Raft 的基础上进行了优化，适用于高并发、低延迟的场景。

应用场景

1. **分布式存储系统**：如分布式数据库、分布式文件系统等，需要保证数据一致性和高可用性。
2. **服务发现与配置管理**：如 ZooKeeper、etcd 等，用于服务注册与发现、配置管理等。
3. **分布式锁**：在分布式系统中实现互斥锁，保证同一时间只有一个客户端能执行关键操作。
4. **分布式任务调度**：如分布式作业调度系统，确保任务分配和执行的强一致性。
5. **消息队列**：如 Kafka、RocketMQ 等，保证消息的顺序性和一致性。

底层实现原理

1. **Leader 选举**：
   - Raft 通过 Leader 选举确保集群中只有一个 Leader 处理客户端请求。
   - 节点有三种状态：Leader、Follower、Candidate。
   - 选举过程包括超时机制和投票机制，确保只有一个 Leader 被选出。
2. **日志复制**：
   - Leader 将客户端请求作为日志条目复制到所有 Follower。
   - 当日志条目被多数节点确认后，Leader 提交该日志并应用到状态机。
   - Follower 按顺序应用日志条目，保证状态一致性。
3. **安全性**：
   - Raft 通过选举限制和日志匹配确保安全性。
   - 只有包含最新日志的节点才能成为 Leader。
   - Leader 通过日志匹配确保 Follower 的日志与自身一致。

4. **成员变更**：
    - Raft 支持动态增减集群节点。
    - 通过两阶段提交确保成员变更期间集群的可用性和一致性。
5. **性能优化**：
    - JRaft 在 Raft 基础上进行了性能优化，如批量日志复制、并行日志应用等。
    - 通过异步机制和高效的网络通信减少延迟，提升吞吐量。

总结

JRaft 是一个高效的 Raft 实现，适用于需要强一致性和高可用性的分布式系统。其核心在于 Leader 选举、日志复制和安全性机制，通过性能优化满足高并发、低延迟的需求。

2. jraft数据处理流程

https://github.com/sofastack/sofa-jraft

JRaft 的数据处理流程基于 Raft 共识算法，主要包括客户端请求处理、日志复制、状态机应用等步骤。以下是 JRaft 数据处理的核心流程：

---

1. 客户端请求处理

- 客户端向集群发送请求（如写请求）。
- 请求首先发送到当前 Leader 节点（只有 Leader 能处理写请求）。
- 如果客户端请求发送到 Follower，Follower 会拒绝并返回 Leader 的地址。

---

2. 日志追加

- Leader 将客户端请求封装为一个日志条目（Log Entry），包含以下信息：
    - 索引（Index）：日志条目的唯一标识。
    - 任期（Term）：当前 Leader 的任期号。
    - 命令（Command）：客户端请求的具体操作。
- Leader 将日志条目追加到自己的本地日志中。

---

3. 日志复制

- Leader 通过 **AppendEntries RPC** 将日志条目并行发送给所有 Follower 节点。
- Follower 收到日志条目后：

a. 检查日志条目的任期号和索引是否与本地日志匹配。
b. 如果匹配，将日志条目追加到本地日志，并返回成功响应。
c. 如果不匹配，返回失败响应，Leader 会调整日志索引并重试。

- Leader 等待大多数节点（包括自己）确认日志条目已成功复制。

---

4. 日志提交

- 当日志条目被大多数节点复制后，Leader 将该日志条目标记为已提交（Committed）。
- Leader 更新自己的提交索引（Commit Index），并通过下一次 **AppendEntries RPC** 通知 Follower 更新提交索引。

---

5. 状态机应用

- Leader 和 Follower 将已提交的日志条目应用到本地状态机（State Machine）。
- 状态机执行日志条目中的命令，完成数据更新或操作。
- 状态机应用完成后，节点更新最后应用索引（Last Applied Index）。

---

6. 响应客户端

- Leader 在状态机应用完成后，向客户端返回成功响应。
- 如果客户端请求超时或失败，客户端会重试请求。

---

7. 日志压缩与快照

- 当日志增长到一定规模时，JRaft 会触发日志压缩（Log Compaction）。
- 节点生成快照（Snapshot），保存当前状态机的状态，并删除已应用的日志条目。
- 快照通过 **InstallSnapshot RPC** 同步给落后的 Follower 节点。

---

8. 异常处理

- **Leader 故障**：如果 Leader 节点故障，集群会触发新的 Leader 选举。
- **Follower 故障**：如果 Follower 节点故障，Leader 会不断重试日志复制，直到故障节点恢复。
- **网络分区**：如果发生网络分区，少数分区的节点无法选举出新的 Leader，多数分区的节点会继续提供服务。

数据处理流程图

复制

客户端请求

  |

  v

Leader 节点

  |

  v

日志追加（本地日志）

  |

  v

日志复制（AppendEntries RPC）

  |

  v

大多数节点确认

  |

  v

日志提交（Commit Index）

  |

  v

状态机应用

  |

  v

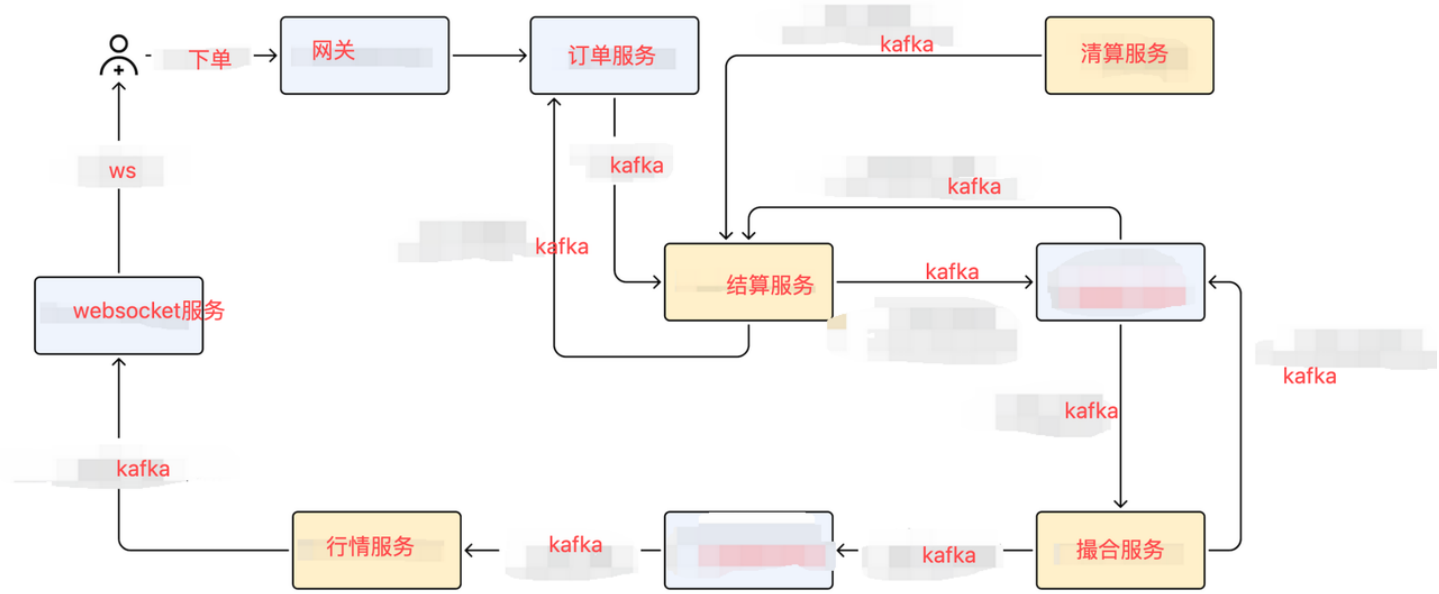
响应客户端

关键点

1. **强一致性**：JRaft 通过日志复制和提交机制，确保所有节点数据一致。
2. **高可用性**：通过 Leader 选举和日志复制，即使部分节点故障，集群仍能正常工作。
3. **性能优化**：JRaft 通过批量日志复制、异步机制等优化，提升处理性能。

JRaft 的数据处理流程清晰且高效，适用于需要强一致性和高可用性的分布式系统。

3. cex交易所架构是如何/现货交易数据流程是怎么样的



4. kafka和rocketmq的区别

Kafka 和 RocketMQ 都是流行的分布式消息队列系统，广泛应用于大数据、实时计算、异步解耦等场景。尽管它们的功能相似，但在设计理念、架构和适用场景上存在显著差异。以下是两者的主要区别：

---

1. 设计背景与目标

- **Kafka**：
  - 由 LinkedIn 开发，主要用于处理大规模的实时日志流和数据管道。
  - 设计目标是高吞吐量、低延迟，适合大数据场景。
- **RocketMQ**：
  - 由阿里巴巴开发，最初用于解决电商场景中的高并发、高可靠消息传递问题。
  - 设计目标是高可靠性、强一致性，适合金融、交易等对数据一致性要求高的场景。

---

2. 消息模型

- **Kafka**：
  - 采用 **发布-订阅模型**，支持多消费者组。
  - 消息按分区（Partition）存储，每个分区内消息有序。

- 消费者通过偏移量（Offset）管理消费进度。

- **RocketMQ**：
  - 支持 **发布-订阅模型** 和 **点对点模型**。
  - 消息按主题（Topic）和队列（Queue）存储，队列内消息有序。
  - 消费者通过消费位点（Consumer Offset）管理消费进度。

---

3. 消息存储

- **Kafka**：
  - 消息存储在磁盘上，采用顺序写和零拷贝技术，保证高吞吐量。
  - 消息按时间或大小滚动存储，支持日志压缩（Log Compaction）。
- **RocketMQ**：
  - 消息存储在磁盘上，采用混合存储结构（CommitLog + ConsumeQueue），保证高可靠性和高性能。
  - 支持消息的定时和延迟投递。

---

4. 消息可靠性

- **Kafka**：
  - 提供 **至少一次（At Least Once）** 和 **至多一次（At Most Once）** 的语义。
  - 通过副本机制（Replication）保证数据可靠性。
- **RocketMQ**：
  - 提供 **至少一次（At Least Once）** 和 **精确一次（Exactly Once）** 的语义。
  - 通过同步刷盘（Sync Flush）和主从同步（Master-Slave）机制保证数据可靠性。

---

5. 消息顺序性

- **Kafka**：
  - 保证分区内消息有序，但不保证全局有序。
  - 如果需要全局有序，可以将所有消息写入一个分区。
- **RocketMQ**：
  - 保证队列内消息有序，支持全局有序（通过单个队列实现）。
  - 支持顺序消息（Ordered Message）和普通消息（Concurrent Message）。

6. 消息延迟

- **Kafka**：
  - 不支持消息的延迟投递。
  - 可以通过外部工具或自定义逻辑实现延迟消息。
- **RocketMQ**：
  - 支持消息的延迟投递，提供多个延迟级别（如 1s、5s、10s 等）。
  - 适用于需要定时任务或延迟处理的场景。

---

7. 事务支持

- **Kafka**：
  - 支持事务消息（Transactional Messages），但实现较为复杂。
  - 适用于需要强一致性的场景。
- **RocketMQ**：
  - 提供完整的事务消息支持，包括半消息（Half Message）和事务回查机制。
  - 适用于金融、支付等对事务要求高的场景。

---

8. 生态系统与集成

- **Kafka**：
  - 生态系统非常丰富，与大数据工具（如 Hadoop、Spark、Flink）集成紧密。
  - 提供 Kafka Connect 和 Kafka Streams，支持数据集成和流处理。
- **RocketMQ**：
  - 生态系统相对较小，但与阿里云产品（如阿里云 MQ、EDAS）集成紧密。
  - 提供 RocketMQ Connect 和 RocketMQ Streams，支持数据集成和流处理。

---

9. 性能与扩展性

- **Kafka**：
  - 高吞吐量，适合大规模数据流处理。
  - 扩展性强，支持动态增加分区和节点。

- **RocketMQ**：
  - 高可靠性和低延迟，适合对一致性要求高的场景。
  - 扩展性强，支持动态增加队列和节点。

---

10. 社区与支持

- **Kafka**：
  - 社区活跃，文档丰富，商业化支持由 Confluent 提供。
- **RocketMQ**：
  - 社区活跃度较高，中文文档丰富，商业化支持由阿里巴巴提供。

总结对比

适用场景

- **Kafka**：适合大数据流处理、日志收集、实时分析等场景。
- **RocketMQ**：适合金融、电商、交易等对可靠性和一致性要求高的场景。

根据具体需求选择合适的消息队列系统，可以更好地满足业务需求。

5. 你对aqs的理解

AQS（AbstractQueuedSynchronizer） 是 Java 并发包（`java.util.concurrent.locks`）中的一个核心框架，用于构建锁和其他同步器（如 Semaphore、CountDownLatch 等）。AQS 提供了一个灵活的底层同步机制，开发者可以基于它实现自定义的同步组件。

---

1. AQS 的核心思想

AQS 的核心思想是通过一个 **双向队列（CLH 队列）** 来管理线程的排队和唤醒，同时通过一个 **状态变量（state）** 来表示同步状态。AQS 将线程的竞争和排队逻辑抽象出来，具体的同步语义（如独占锁、共享锁）由子类实现。

---

2. AQS 的核心组件

1. **状态变量（state）**：
   - 一个 `volatile int` 类型的变量，表示同步状态。
   - 子类可以通过 `getState()`、`setState()` 和 `compareAndSetState()` 方法操作状态。
   - 状态的具体含义由子类定义（如 ReentrantLock 中表示锁的重入次数）。
2. **双向队列（CLH 队列）**：
   - 一个 FIFO 双向队列，用于存储等待获取锁的线程。
   - 每个节点（Node）代表一个等待线程，包含线程引用、等待状态等信息。
3. **独占模式和共享模式**：
   - AQS 支持两种模式：
     - **独占模式**：同一时刻只有一个线程能获取锁（如 ReentrantLock）。
     - **共享模式**：多个线程可以同时获取锁（如 Semaphore、CountDownLatch）。

---

3. AQS 的工作流程

独占模式（以 ReentrantLock 为例）

1. **获取锁**：
   - 线程调用 `acquire(int arg)` 方法尝试获取锁。
   - 如果成功（通过 `tryAcquire(int arg)` 方法），线程继续执行。
   - 如果失败，线程被封装为节点加入队列，并进入阻塞状态。
2. **释放锁**：
   - 线程调用 `release(int arg)` 方法释放锁。
   - 如果成功（通过 `tryRelease(int arg)` 方法），唤醒队列中的下一个线程。

共享模式（以 Semaphore 为例）

1. **获取许可**：
   - 线程调用 `acquireShared(int arg)` 方法尝试获取许可。
   - 如果成功（通过 `tryAcquireShared(int arg)` 方法），线程继续执行。
   - 如果失败，线程被封装为节点加入队列，并进入阻塞状态。
2. **释放许可**：
   - 线程调用 `releaseShared(int arg)` 方法释放许可。
   - 如果成功（通过 `tryReleaseShared(int arg)` 方法），唤醒队列中的后续线程。

4. AQS 的关键方法

1. **模板方法**：
   - `acquire(int arg)`：独占模式获取锁。
   - `release(int arg)`：独占模式释放锁。
   - `acquireShared(int arg)`：共享模式获取锁。
   - `releaseShared(int arg)`：共享模式释放锁。
2. **钩子方法**（由子类实现）：
   - `tryAcquire(int arg)`：尝试获取锁（独占模式）。
   - `tryRelease(int arg)`：尝试释放锁（独占模式）。
   - `tryAcquireShared(int arg)`：尝试获取锁（共享模式）。
   - `tryReleaseShared(int arg)`：尝试释放锁（共享模式）。
   - `isHeldExclusively()`：判断当前线程是否独占锁。

---

5. AQS 的应用

AQS 是 Java 并发包中许多同步组件的基础，包括：

1. **ReentrantLock**：
   - 基于 AQS 实现的可重入独占锁。
   - 支持公平锁和非公平锁。
2. **ReentrantReadWriteLock**：
   - 基于 AQS 实现的读写锁。
   - 读锁是共享模式，写锁是独占模式。
3. **Semaphore**：
   - 基于 AQS 实现的信号量。
   - 控制同时访问资源的线程数。
4. **CountDownLatch**：
   - 基于 AQS 实现的倒计时门闩。
   - 用于等待多个线程完成任务。
5. **CyclicBarrier**：
   - 基于 AQS 实现的循环屏障。

- 用于多个线程相互等待，达到屏障后继续执行。

---

6. AQS 的优点

1. **灵活性**：
   - AQS 提供了底层的同步机制，开发者可以基于它实现各种自定义同步器。
2. **高性能**：
   - AQS 通过 CAS 操作和队列管理，减少了线程上下文切换的开销。
3. **可扩展性**：
   - 支持独占模式和共享模式，适用于多种同步场景。

---

7. AQS 的源码分析

AQS 的核心源码包括：

1. **状态管理**：
   - 通过 `volatile int state` 和 CAS 操作实现状态的高效管理。
2. **队列管理**：
   - 使用双向链表实现 CLH 队列，管理等待线程。
3. **线程阻塞与唤醒**：
   - 通过 `LockSupport.park()` 和 `LockSupport.unpark()` 实现线程的阻塞和唤醒。

---

8. 总结

AQS 是 Java 并发编程的核心框架之一，通过状态管理和队列机制实现了高效的线程同步。它的设计灵活且高效，是许多同步组件的基础。理解 AQS 的原理和实现，对于掌握 Java 并发编程和设计高性能并发工具具有重要意义。

6. 你对jvm的理解

在面试中回答“你对 JVM 的理解”时，可以从 **理论深度**、**实践经验** 和 **问题解决能力** 三个维度展开，展示你对 JVM 的全面理解和实际应用能力。以下是一个结构化回答模板：

---

1. 概述 JVM 的核心作用

首先简要说明 JVM 的作用，展示你对 JVM 的全局认识：

> JVM（Java 虚拟机）是 Java 程序的运行环境，它负责将 Java 字节码翻译成机器码并执行。JVM 的核心作用包括 **内存管理**、**垃圾回收**、**字节码解释与编译** 以及 **线程管理** 等。它是 Java 跨平台特性的基础，也是 Java 高性能的关键。

---

2. 深入 JVM 的架构

接下来，深入 JVM 的架构，展示你对 JVM 内部机制的了解：

> JVM 的架构主要包括以下几个部分：
>
> 1. **类加载器（Class Loader）**：
>   - 负责加载字节码文件到内存，包括加载、链接和初始化三个阶段。
>   - 类加载器采用双亲委派模型，确保类的唯一性和安全性。
> 2. **运行时数据区（Runtime Data Areas）**：
>   - 包括方法区、堆、栈、本地方法栈和程序计数器。
>   - 方法区存储类信息、常量、静态变量等；堆是对象分配的主要区域；栈用于存储方法调用的局部变量和操作数栈。
> 3. **执行引擎（Execution Engine）**：
>   - 负责解释和执行字节码，包括解释器、即时编译器（JIT）和垃圾回收器。
>   - JIT 将热点代码编译为本地机器码，提升执行效率。
> 4. **本地方法接口（JNI）**：
>   - 提供 Java 调用本地方法（如 C/C++ 代码）的能力。
> 5. **垃圾回收器（Garbage Collector）**：
>   - 负责自动回收不再使用的对象，管理堆内存。

---

3. 结合实际经验谈 JVM 调优

展示你在实际工作中对 JVM 的应用和调优经验：

> 在实际项目中，我经常需要对 JVM 进行调优，以解决性能问题和内存泄漏。以下是我的一些经验：
>
> 1. **内存分配调优**：
>   - 通过调整堆大小（`-Xms` 和 `-Xmx`）和方法区大小（`-XX:MetaspaceSize`），优化内存使用。
>   - 使用 `-XX:NewRatio` 和 `-XX:SurvivorRatio` 调整新生代和老年代的比例。
> 2. **垃圾回收器选择**：

- 根据应用场景选择合适的垃圾回收器，如 CMS、G1 或 ZGC。
- 例如，在高吞吐量场景下使用 Parallel GC，在低延迟场景下使用 G1 或 ZGC。

3. **监控与诊断**：
   - 使用工具如 JVisualVM、JConsole 或 Arthas 监控 JVM 的运行状态。
   - 通过分析 GC 日志（`-Xloggc`）和堆转储（`-XX:+HeapDumpOnOutOfMemoryError`）定位内存泄漏和性能瓶颈。

---

4. 解决实际问题的案例

分享一个你解决过的 JVM 相关问题，展示你的问题解决能力：

> 在一次线上项目中，我们遇到了频繁的 Full GC 问题，导致系统响应变慢。通过分析 GC 日志，我发现老年代内存占用过高，原因是缓存设计不合理，导致大量对象无法被回收。我通过以下步骤解决了问题：
>
> 1. 使用 `jmap` 和 `jhat` 分析堆转储文件，定位到缓存对象的引用链。
> 2. 优化缓存设计，引入 LRU 淘汰策略，限制缓存大小。
> 3. 调整 JVM 参数，增加老年代大小并优化 GC 策略。
>
> 最终，Full GC 的频率显著降低，系统性能恢复正常。

---

5. 对 JVM 未来发展的看法

展示你对技术的深入思考和前瞻性：

> 随着 Java 的不断发展，JVM 也在持续进化。例如：
>
> 1. **新垃圾回收器**：如 ZGC 和 Shenandoah，致力于将停顿时间控制在毫秒级以下。
> 2. **GraalVM**：支持多语言运行和原生镜像生成，进一步提升了 Java 的性能和灵活性。
> 3. **云原生支持**：JVM 正在更好地适应容器化环境，如通过 `-XX:+UseContainerSupport` 参数优化容器内的资源使用。
>
> 我认为，未来 JVM 会继续在高性能、低延迟和云原生方面发力，成为更强大的运行时平台。

---

6. 总结

最后用一句话总结你对 JVM 的理解：

> 总的来说，JVM 是 Java 生态的核心，理解其原理和机制对于编写高性能、稳定的 Java 应用至关重要。通过不断学习和实践，我能够熟练运用 JVM 调优和问题诊断技术，解决实际生产环境中的复杂问题。

回答要点

1. **结构化表达**：从理论到实践，层层深入。
2. **结合实际经验**：通过具体案例展示你的能力。
3. **展示技术深度**：提到 JVM 的高级特性和未来发展方向。
4. **自信与简洁**：语言简洁明了，语气自信。

通过这样的回答，你不仅能展示对 JVM 的深刻理解，还能体现你的实际经验和问题解决能力，给面试官留下深刻印象。

7. spring中的IOC和AOP实现原理

Spring 框架的两大核心特性是 **IOC（控制反转）** 和 **AOP（面向切面编程）**。它们分别解决了对象管理和横切关注点的问题。以下是它们的原理和实现细节：

---

1. IOC（控制反转）

1.1 IOC 的核心思想

IOC 是一种设计原则，将对象的创建和依赖注入的控制权从应用程序代码转移到框架（如 Spring）中。它的核心思想是：

- **控制反转**：对象的生命周期和依赖关系由 Spring 容器管理，而不是由开发者手动管理。
- **依赖注入（DI）**：通过构造函数、Setter 方法或字段注入，将依赖对象注入到目标对象中。

1.2 IOC 的实现原理

Spring 的 IOC 容器（如 `ApplicationContext`）通过以下步骤实现 IOC：

1. **Bean 的定义**：
   - 开发者通过 XML 配置文件、Java 注解（如 `@Component`、`@Service`）或 Java 配置类（如 `@Configuration`）定义 Bean。
2. **Bean 的加载**：
   - Spring 容器启动时，会扫描配置文件或注解，解析 Bean 的定义，并将其注册到 BeanFactory 中。
3. **Bean 的实例化**：
   - 当需要某个 Bean 时，Spring 容器会根据 Bean 的定义实例化对象。
   - 如果 Bean 是单例（Singleton），容器会缓存实例，后续请求直接返回缓存对象。

4. **依赖注入**：
   - Spring 容器根据 Bean 的依赖关系，自动注入所需的依赖对象。
   - 注入方式包括：
     - 构造函数注入（Constructor Injection）
     - Setter 方法注入（Setter Injection）
     - 字段注入（Field Injection）
5. **Bean 的生命周期管理**：
   - Spring 容器负责管理 Bean 的初始化（如 `@PostConstruct`）和销毁（如 `@PreDestroy`）。

1.3 IOC 的优点

- **解耦**：将对象的创建和依赖关系与业务逻辑分离，降低代码耦合度。
- **可测试性**：依赖注入使得单元测试更容易，可以通过 Mock 对象替换依赖。
- **灵活性**：通过配置文件或注解，可以动态调整 Bean 的定义和依赖关系。

---

2. AOP（面向切面编程）

2.1 AOP 的核心思想

AOP 是一种编程范式，用于将横切关注点（如日志、事务、权限校验）从业务逻辑中分离出来，通过切面（Aspect）统一管理。它的核心思想是：

- **横切关注点**：将分散在多个模块中的公共功能（如日志记录）提取到切面中。
- **动态代理**：通过代理模式，在运行时动态地将切面逻辑织入目标方法。

2.2 AOP 的实现原理

Spring AOP 基于动态代理实现，具体步骤如下：

1. **切面定义**：
   - 开发者通过注解（如 `@Aspect`、`@Before`、`@After`）或 XML 配置定义切面。
   - 切面包括：
     - 切点（Pointcut）：定义哪些方法需要被增强。
     - 通知（Advice）：定义增强逻辑（如前置通知、后置通知、环绕通知等）。
2. **代理创建**：
   - Spring AOP 使用 JDK 动态代理或 CGLIB 动态代理创建目标对象的代理。
     - 如果目标对象实现了接口，使用 JDK 动态代理。

    - 如果目标对象没有实现接口，使用 CGLIB 动态代理。
3. **切面织入**：
    - 在运行时，当调用目标方法时，代理对象会拦截方法调用，并执行切面逻辑。
    - 例如：
        - 前置通知（`@Before`）：在目标方法执行前执行。
        - 后置通知（`@After`）：在目标方法执行后执行。
        - 环绕通知（`@Around`）：在目标方法执行前后执行，并可以控制是否执行目标方法。
4. **AOP 的应用场景**：
    - 日志记录
    - 事务管理
    - 权限校验
    - 性能监控

2.3 AOP 的优点

- **解耦**：将横切关注点与业务逻辑分离，提高代码的可维护性。
- **复用性**：通过切面复用公共逻辑，减少重复代码。
- **灵活性**：通过配置动态调整切面逻辑，无需修改业务代码。

---

3. IOC 和 AOP 的关系

- **IOC 是基础**：AOP 依赖于 IOC，因为切面本身也是由 Spring 容器管理的 Bean。
- **AOP 是扩展**：AOP 通过动态代理增强了 IOC 管理的 Bean 的功能，实现了横切关注点的统一管理。

---

4. 实际应用案例

4.1 IOC 案例

java

复制

@Servicepublic class UserService {@Autowiredprivate UserRepository userRepository;public User getUserById(Long id) {return userRepository.findById(id);}}

- `UserService` 依赖 `UserRepository`，Spring 容器会自动注入 `UserRepository` 的实例。

4.2 AOP 案例

java

复制

@Aspect@Componentpublic class LoggingAspect {@Before("execution(* com.example.service.*.*(..))")public void logBefore(JoinPoint joinPoint) {System.out.println("Method called: " + joinPoint.getSignature().getName());}}

- 在 `com.example.service` 包下的所有方法调用前，打印日志。

---

5. 总结

- **IOC** 通过控制反转和依赖注入，实现了对象的解耦和灵活管理。
- **AOP** 通过动态代理和切面编程，实现了横切关注点的统一管理。
- 两者结合，使得 Spring 框架能够高效地管理对象和处理横切关注点，极大地提升了开发效率和代码质量。

通过深入理解 IOC 和 AOP 的原理，并结合实际应用场景，可以更好地利用 Spring 框架构建高质量的应用程序。

8. java中的设计模式

在面试中回答“Java 中的设计模式”时，可以从 **理论深度**、**实际应用** 和 **问题解决能力** 三个维度展开，展示你对设计模式的全面理解和实际应用能力。以下是一个结构化回答模板：

---

1. 概述设计模式的核心作用

首先简要说明设计模式的作用，展示你对设计模式的全局认识：

> 设计模式是解决软件设计中常见问题的经验总结，它提供了一套可复用的解决方案。设计模式的核心作用是提高代码的 **可维护性**、**可扩展性** 和 **复用性**。Java 中常用的设计模式包括创建型、结构型和行为型三大类。

---

2. 分类介绍设计模式

接下来，分类介绍设计模式，展示你对设计模式的系统理解：

> Java 中的设计模式可以分为三大类：
>
> 1. **创建型模式**：
>   - 解决对象创建的问题，包括：

- **单例模式（Singleton）**：确保一个类只有一个实例。
- **工厂模式（Factory）**：将对象的创建与使用分离。
- **建造者模式（Builder）**：分步构建复杂对象。

2. **结构型模式**：
   - 解决类或对象组合的问题，包括：
     - **适配器模式（Adapter）**：使不兼容的接口能够协同工作。
     - **装饰器模式（Decorator）**：动态地为对象添加功能。
     - **代理模式（Proxy）**：为对象提供代理以控制访问。
3. **行为型模式**：
   - 解决对象之间的职责分配和通信问题，包括：
     - **观察者模式（Observer）**：定义对象间的一对多依赖关系。
     - **策略模式（Strategy）**：定义一系列算法并使其可互换。
     - **模板方法模式（Template Method）**：定义算法的框架并允许子类重写步骤。

---

3. 结合实际经验谈设计模式的应用

展示你在实际工作中对设计模式的应用经验：

在实际项目中，我经常使用设计模式来解决复杂问题。以下是我的一些经验：

1. **单例模式**：
   - 在配置管理类中使用单例模式，确保全局唯一配置实例。
   - 例如，使用双重检查锁定（Double-Checked Locking）实现线程安全的单例。
2. **工厂模式**：
   - 在支付系统中使用工厂模式，根据支付类型创建不同的支付处理器。
   - 例如，定义一个 `PaymentFactory`，根据传入的支付类型返回 `AlipayProcessor` 或 `WechatPayProcessor`。
3. **观察者模式**：
   - 在消息通知系统中使用观察者模式，实现发布-订阅机制。
   - 例如，定义 `MessagePublisher` 和 `MessageSubscriber`，当消息发布时，所有订阅者都会收到通知。

---

4. 解决实际问题的案例

分享一个你使用设计模式解决实际问题的案例，展示你的问题解决能力：

> 在一次电商项目中，我们需要实现一个促销活动的折扣计算功能。由于折扣规则复杂且可能频繁变化，我使用了 **策略模式** 来解决这个问题：
>
> 1. 定义一个 `DiscountStrategy` 接口，包含 `calculateDiscount` 方法。
> 2. 实现多个具体的策略类，如 `PercentageDiscountStrategy`、`FixedAmountDiscountStrategy` 等。
> 3. 在促销活动中，根据用户选择的折扣类型动态切换策略。
>
> 通过策略模式，我们实现了折扣规则的灵活扩展和动态切换，极大地提高了代码的可维护性。

---

5. 对设计模式未来发展的看法

展示你对技术的深入思考和前瞻性：

> 随着软件开发的复杂度不断增加，设计模式的重要性也越来越突出。未来，设计模式可能会在以下方面发展：
>
> 1. **与函数式编程结合**：如使用 Lambda 表达式简化观察者模式、策略模式等。
> 2. **在微服务架构中的应用**：如使用代理模式实现服务网关，使用装饰器模式实现服务熔断等。
> 3. **与新兴技术的结合**：如使用设计模式优化 AI 模型的管理和调用。

---

6. 总结

最后用一句话总结你对设计模式的理解：

> 总的来说，设计模式是软件开发中的重要工具，它帮助我们编写高质量、可维护的代码。通过不断学习和实践，我能够熟练运用各种设计模式，解决实际生产环境中的复杂问题。

回答要点

1. **结构化表达**：从理论到实践，层层深入。
2. **结合实际经验**：通过具体案例展示你的能力。
3. **展示技术深度**：提到设计模式的高级应用和未来发展方向。
4. **自信与简洁**：语言简洁明了，语气自信。

通过这样的回答，你不仅能展示对设计模式的深刻理解，还能体现你的实际经验和问题解决能力，给面试官留下深刻印象。

9. 你们服务下撤单接口qps能达到多少

压测最高达到8-9k

10. 你们交易对如何分布在撮合服务的

共4台撮合服务，每台撮合服务负责20多个交易对撮合，交易对的热点币对分散到不同撮合服务的

11. 数据库死锁发生的场景

数据库死锁通常发生在多个事务并发访问和修改相同资源时，由于资源竞争和锁的持有顺序不当，导致事务相互等待，形成死锁。以下是常见的死锁场景及其原因分析：

---

1. 不同顺序访问相同资源

场景描述：

- 事务 A 先访问资源 X，再访问资源 Y。
- 事务 B 先访问资源 Y，再访问资源 X。
- 如果事务 A 和事务 B 同时执行，可能会形成死锁。

示例：

sql

复制

-- 事务 ABEGIN;UPDATE table1 SET column1 = 'value1' WHERE id = 1; -- 锁定资源 XUPDATE table2 SET column2 = 'value2' WHERE id = 2; -- 尝试锁定资源 Y-- 事务 BBEGIN;UPDATE table2 SET column2 = 'value2' WHERE id = 2; -- 锁定资源 YUPDATE table1 SET column1 = 'value1' WHERE id = 1; -- 尝试锁定资源 X

原因：

- 事务 A 和事务 B 以不同的顺序访问资源 X 和 Y，导致相互等待。

---

2. 批量更新或删除

场景描述：

- 事务 A 和事务 B 同时执行批量更新或删除操作，涉及多行数据。
- 如果事务 A 和事务 B 以不同的顺序锁定行，可能会形成死锁。

示例：

sql

复制

-- 事务 ABEGIN;UPDATE table1 SET column1 = 'value1' WHERE id IN (1, 2, 3); -- 锁定行 1, 2, 3-- 事务 BBEGIN;UPDATE table1 SET column1 = 'value1' WHERE id IN (3, 2, 1); -- 锁定行 3, 2, 1

原因：

- 事务 A 和事务 B 以不同的顺序锁定相同的行，导致相互等待。

---

3. 索引竞争

场景描述：

- 事务 A 和事务 B 同时操作同一张表，但使用不同的索引。
- 如果事务 A 和事务 B 以不同的顺序锁定索引，可能会形成死锁。

示例：

sql

复制

-- 事务 ABEGIN;UPDATE table1 SET column1 = 'value1' WHERE index_column = 'A'; -- 锁定索引 A-- 事务 BBEGIN;UPDATE table1 SET column1 = 'value1' WHERE index_column = 'B'; -- 锁定索引 B

原因：

- 事务 A 和事务 B 操作不同的索引，但涉及相同的行，导致相互等待。

---

4. 外键约束

场景描述：

- 事务 A 和事务 B 同时操作具有外键关系的表。
- 如果事务 A 和事务 B 以不同的顺序锁定主表和从表，可能会形成死锁。

示例：

sql

复制

-- 事务 ABEGIN;UPDATE parent_table SET column1 = 'value1' WHERE id = 1; -- 锁定主表UPDATE child_table SET column2 = 'value2' WHERE parent_id = 1; -- 尝试锁定从表-- 事务 BBEGIN;UPDATE

child_table SET column2 = 'value2' WHERE parent_id = 1; -- 锁定从表UPDATE parent_table SET column1 = 'value1' WHERE id = 1; -- 尝试锁定主表

原因：

- 事务 A 和事务 B 以不同的顺序锁定主表和从表，导致相互等待。

---

5. 嵌套事务

场景描述：

- 事务 A 和事务 B 同时执行嵌套事务（如存储过程调用）。
- 如果事务 A 和事务 B 以不同的顺序锁定资源，可能会形成死锁。

示例：

sql

复制

-- 事务 ABEGIN;CALL nested_procedure(); -- 嵌套事务锁定资源 X 和 Y-- 事务 BBEGIN;CALL nested_procedure(); -- 嵌套事务锁定资源 Y 和 X

原因：

- 事务 A 和事务 B 在嵌套事务中以不同的顺序锁定资源，导致相互等待。

---

6. 分布式事务

场景描述：

- 在分布式系统中，多个事务同时操作多个数据库。
- 如果事务 A 和事务 B 以不同的顺序锁定跨数据库的资源，可能会形成死锁。

示例：

sql

复制

-- 事务 ABEGIN;UPDATE db1.table1 SET column1 = 'value1' WHERE id = 1; -- 锁定数据库 1 的资源 UPDATE db2.table2 SET column2 = 'value2' WHERE id = 2; -- 尝试锁定数据库 2 的资源-- 事务 BBEGIN;UPDATE db2.table2 SET column2 = 'value2' WHERE id = 2; -- 锁定数据库 2 的资源UPDATE db1.table1 SET column1 = 'value1' WHERE id = 1; -- 尝试锁定数据库 1 的资源

原因：

- 事务 A 和事务 B 以不同的顺序锁定跨数据库的资源，导致相互等待。

---

7. 锁升级

场景描述：

- 事务 A 和事务 B 同时操作同一张表，事务 A 从行锁升级为表锁，事务 B 从表锁升级为行锁。
- 如果事务 A 和事务 B 以不同的顺序锁定资源，可能会形成死锁。

示例：

sql

复制

-- 事务 ABEGIN;SELECT * FROM table1 WHERE id = 1 FOR UPDATE; -- 行锁ALTER TABLE table1 ADD COLUMN new_column INT; -- 尝试升级为表锁-- 事务 BBEGIN;LOCK TABLE table1 WRITE; -- 表锁UPDATE table1 SET column1 = 'value1' WHERE id = 1; -- 尝试升级为行锁

原因：

- 事务 A 和事务 B 以不同的顺序锁定资源，导致相互等待。

---

8. 总结

数据库死锁通常发生在以下场景：

1. **不同顺序访问相同资源**。
2. **批量更新或删除**。
3. **索引竞争**。
4. **外键约束**。
5. **嵌套事务**。
6. **分布式事务**。
7. **锁升级**。

通过合理设计事务、统一资源访问顺序和使用监控工具，可以有效减少死锁的发生，提升系统的稳定性和性能。

12. Update sql 执行时间过长 如何优化

当 MySQL 中的 `UPDATE` 语句执行时间过长时，可能是由于数据量大、索引不合理、锁竞争或硬件资源不足等原因导致的。以下是详细的优化方法和步骤：

---

1. 分析问题原因

在优化之前，首先需要分析 `UPDATE` 语句执行时间过长的原因。可以通过以下工具和方法进行诊断：

1. **使用 `EXPLAIN` 分析查询计划**：
   - 查看 `UPDATE` 语句的执行计划，了解是否使用了索引、扫描了多少行等。
   - 示例：
   - sql
   - 复制
   - EXPLAIN UPDATE users SET status = 'active' WHERE created_at < '2023-01-01';
   - 关注 `type`、`key`、`rows` 和 `Extra` 字段。
2. **检查慢查询日志**：
   - 启用慢查询日志（`slow_query_log`），记录执行时间超过阈值的 SQL 语句。
   - 示例：
   - sql
   - 复制
   - SET GLOBAL slow_query_log = 'ON';SET GLOBAL long_query_time = 1; -- 记录执行时间超过 1 秒的查询
3. **监控锁竞争**：
   - 使用 `SHOW ENGINE INNODB STATUS` 或 `information_schema.INNODB_TRX` 查看当前事务和锁的状态。
   - 示例：
   - sql
   - 复制
   - SHOW ENGINE INNODB STATUS;SELECT * FROM information_schema.INNODB_TRX;

---

2. 优化 UPDATE 语句

2.1 减少更新范围

- 如果 `UPDATE` 语句更新的数据量过大，可以分批更新，减少单次更新的行数。
- 示例：
- sql
- 复制
- -- 分批更新UPDATE users SET status = 'active' WHERE created_at < '2023-01-01' LIMIT 1000;

2.2 使用索引

- 确保 `WHERE` 条件中的字段有索引，避免全表扫描。
- 示例：
- sql
- 复制
- CREATE INDEX idx_users_created_at ON users(created_at);

2.3 避免更新不必要的字段

- 只更新需要修改的字段，减少写操作的开销。
- 示例：
- sql
- 复制
- -- 不推荐UPDATE users SET status = 'active', last_login = NOW() WHERE id = 1;-- 推荐UPDATE users SET status = 'active' WHERE id = 1;

2.4 使用 JOIN 优化复杂更新

- 对于涉及多表的 `UPDATE` 语句，使用 `JOIN` 代替子查询。
- 示例：
- sql
- 复制
- -- 不推荐UPDATE orders SET status = 'completed' WHERE user_id IN (SELECT id FROM users WHERE status = 'active');-- 推荐UPDATE orders o JOIN users u ON o.user_id = u.id SET o.status = 'completed' WHERE u.status = 'active';

---

3. 优化索引

3.1 创建合适的索引

- 为 `WHERE` 条件中的字段创建索引，加速查询。

- 示例：
- sql
- 复制
- CREATE INDEX idx_users_status ON users(status);

3.2 避免索引失效

- 避免在索引字段上使用函数或表达式。
- 示例：
- sql
- 复制
- -- 不推荐UPDATE users SET status = 'active' WHERE YEAR(created_at) = 2023;-- 推荐UPDATE users SET status = 'active' WHERE created_at BETWEEN '2023-01-01' AND '2023-12-31';

3.3 删除不必要的索引

- 过多的索引会增加写操作的开销，删除不常用的索引。

---

4. 减少锁竞争

4.1 降低隔离级别

- 将事务隔离级别从 `REPEATABLE READ` 降低到 `READ COMMITTED`，减少锁的范围。
- 示例：
- sql
- 复制
- SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;

4.2 使用行锁

- 确保 `UPDATE` 语句使用行锁（如 InnoDB 引擎），而不是表锁。
- 示例：
- sql
- 复制
- -- 确保表使用 InnoDB 引擎ALTER TABLE users ENGINE=InnoDB;

4.3 分批提交

- 对于大事务，分批提交以减少锁的持有时间。

- 示例：
- sql
- 复制
- BEGIN;UPDATE users SET status = 'active' WHERE created_at < '2023-01-01' LIMIT 1000;COMMIT;BEGIN;UPDATE users SET status = 'active' WHERE created_at < '2023-01-01' LIMIT 1000 OFFSET 1000;COMMIT;

---

5. 优化表结构

5.1 分区表

- 对大表进行分区，减少单次更新的数据量。
- 示例：
- sql
- 复制
- CREATE TABLE users (
  id INT PRIMARY KEY,
  name VARCHAR(100),
  created_at DATE)PARTITION BY RANGE (YEAR(created_at)) (PARTITION p0 VALUES LESS THAN (2020),PARTITION p1 VALUES LESS THAN (2021),PARTITION p2 VALUES LESS THAN (2022));

5.2 归档历史数据

- 将历史数据归档到其他表或数据库，减少主表的数据量。
- 示例：
- sql
- 复制
- INSERT INTO users_archive SELECT * FROM users WHERE created_at < '2020-01-01';DELETE FROM users WHERE created_at < '2020-01-01';

---

6. 硬件和配置优化

6.1 增加内存

- 增加 MySQL 的内存配置（如 `innodb_buffer_pool_size`），减少磁盘 I/O。
- 示例：

- sql
- 复制
- SET GLOBAL innodb_buffer_pool_size = 2G;

6.2 使用 SSD

- 使用 SSD 代替机械硬盘，提升磁盘 I/O 性能。

6.3 优化配置参数

- 调整 MySQL 的配置参数（如 `innodb_flush_log_at_trx_commit`、`sync_binlog`），平衡性能和数据安全性。
- 示例：
- sql
- 复制
- SET GLOBAL innodb_flush_log_at_trx_commit = 2;SET GLOBAL sync_binlog = 100;

---

7. 总结

优化 `UPDATE` 语句的执行时间需要从多个方面入手：

1. **分析问题原因**：使用 `EXPLAIN`、慢查询日志和锁监控工具。
2. **优化 SQL 语句**：减少更新范围、使用索引、避免更新不必要的字段。
3. **优化索引**：创建合适的索引，避免索引失效。
4. **减少锁竞争**：降低隔离级别、使用行锁、分批提交。
5. **优化表结构**：分区表、归档历史数据。
6. **硬件和配置优化**：增加内存、使用 SSD、调整配置参数。

通过以上方法，可以显著提升 `UPDATE` 语句的执行效率，减少数据库的负载和响应时间。

13. B+tree 和 二叉树的区别

B+ 树和二叉树是两种不同的树形数据结构，它们在设计目标、结构特点和应用场景上有显著差异。以下是它们的详细对比：

---

1. 设计目标

- **二叉树**：

- 设计目标是高效地支持查找、插入和删除操作。
- 适用于内存中的数据存储和操作。

- **B+ 树**：
  - 设计目标是优化磁盘 I/O 操作，适合大规模数据存储。
  - 广泛应用于数据库和文件系统中。

---

2. 结构特点

2.1 二叉树

- **节点结构**：
  - 每个节点最多有两个子节点（左子节点和右子节点）。
- **平衡性**：
  - 普通二叉树可能退化为链表，导致性能下降。
  - 平衡二叉树（如 AVL 树、红黑树）通过旋转操作保持平衡。
- **高度**：
  - 高度为 O(log n)，其中 n 是节点数。

2.2 B+ 树

- **节点结构**：
  - 每个节点可以有多个子节点（通常远大于 2）。
  - 内部节点只存储键值，不存储数据；数据全部存储在叶子节点。
- **平衡性**：
  - B+ 树通过分裂和合并操作保持平衡。
- **高度**：
  - 高度为 O(log m n)，其中 m 是节点的最大子节点数，n 是节点数。
  - 由于 m 通常较大，B+ 树的高度比二叉树低。

---

3. 存储方式

- **二叉树**：
  - 通常存储在内存中，适合处理小规模数据。
- **B+ 树**：

- 通常存储在磁盘上，适合处理大规模数据。
- 通过减少树的高度，减少磁盘 I/O 操作。

---

4. 查找性能

- **二叉树**：
  - 查找时间复杂度为 O(log n)。
  - 适合内存中的高效查找。
- **B+ 树**：
  - 查找时间复杂度为 O(log m n)。
  - 由于节点大小通常与磁盘块大小匹配，B+ 树的查找性能在磁盘上更优。

---

5. 插入和删除

- **二叉树**：
  - 插入和删除操作可能导致树的不平衡，需要通过旋转操作重新平衡。
  - 时间复杂度为 O(log n)。
- **B+ 树**：
  - 插入和删除操作通过分裂和合并节点保持平衡。
  - 时间复杂度为 O(log m n)。

---

6. 应用场景

- **二叉树**：
  - 适用于内存中的数据存储和操作，如：
    - 二叉搜索树（BST）用于高效查找。
    - 堆用于优先队列。
    - 平衡二叉树（如 AVL 树、红黑树）用于需要严格平衡的场景。
- **B+ 树**：
  - 适用于磁盘上的大规模数据存储，如：
    - 数据库索引（如 MySQL 的 InnoDB 存储引擎）。
    - 文件系统（如 NTFS、ReiserFS）。

---

7. 数据存储方式

- **二叉树**：
  - 数据存储在任意节点。
- **B+ 树**：
  - 数据仅存储在叶子节点，内部节点只存储键值。
  - 叶子节点通过指针连接，支持高效的范围查询。

---

8. 范围查询

- **二叉树**：
  - 范围查询效率较低，需要遍历多个节点。
- **B+ 树**：
  - 范围查询效率高，叶子节点通过指针连接，可以快速遍历。

---

9. 总结对比

特性	二叉树	B+ 树
设计目标	内存中的高效操作	磁盘上的大规模数据存储
节点子节点数	最多 2 个	多个（通常远大于 2）
平衡性	可能不平衡，需旋转操作	通过分裂和合并保持平衡
高度	O(log n)	O(log m n)
存储方式	内存	磁盘
查找性能	O(log n)	O(log m n)
插入和删除	O(log n)，可能需旋转	O(log m n)，通过分裂和合并保持平衡
应用场景	内存中的数据操作	数据库索引、文件系统
数据存储	任意节点	仅叶子节点
范围查询	效率较低	效率高

---

10. 实际应用示例

10.1 二叉树的应用

- **二叉搜索树（BST）：**
- python
- 复制

- class TreeNode:def init(self, value):
  self.value = value
  self.left = None
  self.right = Noneclass BinarySearchTree:def init(self):
  self.root = Nonedef insert(self, value):if not self.root:
  self.root = TreeNode(value)else:
  self._insert(self.root, value)def _insert(self, node, value):if value < node.value:if node.left:
  self._insert(node.left, value)else:
  node.left = TreeNode(value)else:if node.right:
  self._insert(node.right, value)else:
  node.right = TreeNode(value)

10.2 B+ 树的应用

- **数据库索引**：
  - MySQL 的 InnoDB 存储引擎使用 B+ 树作为索引结构。
  - 支持高效的点查询和范围查询。

---

11. 总结

- **二叉树** 适合内存中的高效操作，但在大规模数据存储和磁盘 I/O 优化方面表现不足。
- **B+ 树** 通过多子节点和分层存储优化了磁盘 I/O，适合数据库和文件系统等大规模数据存储场景。

根据具体需求选择合适的数据结构，可以显著提升系统性能。