Java集合

【3/3】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

当链表长度≤6时，红黑树会退化为链表

**解决哈希冲突**：

1. **拉链法**
2. **红黑树：**优化查询时间复杂度
3. **二次扰动函数：**混合高位低位、优化索引分布、减少哈希冲突
   1. 作用：混合高位低位、优化索引分布、减少哈希冲突
   2. 原理：如果哈希值低位变化较小，用 (n-1) & hash 计算索引时，低位相同的键会映射到同一位置。通过二次扰动函数，将哈希值的高位和低位异或运算，使哈希值的每一位都参与索引计算，减少冲突概率。
4. **动态扩容机制：**
   1. 超过阈值（容量×负载因子，默认0.75）扩容2倍并重新计算索引。
   2. 负载因子过高会增加哈希冲突，过低则导致频繁扩容

【5/5】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（容量×负载因子，默认0.75）

**Java8 HashMap扩容触发条件：**

元素数量超过阈值（容量×负载因子，默认0.75）

OR 链表长度≥8但数组容量＜64

**Rehash过程：**

1. **新建一个2倍大小的数组**
2. Java8只需部分移动：if e.hash & oldCap==0
   1. 0接在loTail后面
   2. 1接在hiTail后面

举例：

数组长度是2的次方，且扩容为2倍。

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）

​

【3/3】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：**

1. 多线程同时操作导致**数据覆盖**
2. 扩容头插法迁移节点时形成**环形链表**引发死循环
3. 多线程同时**树化**导致红黑树**结构异常**

**解决方案：**

1. synchronizedMap：在方法级别加锁，性能低
2. ConcurrentHashMap：采用分段锁（Java7）或CAS+ synchronized（Java8），性能高。

【3/3】为什么Java 8将链表转红黑树的阈值设为8？

链表长度≤8时，链表**性能**足够且**内存**占用低

根据泊松分布，哈希冲突时链表长度达到8的概率极低

【3/3】HashMap的负载因子为什么默认是0.75？

0.75是**空间利用率**与**哈希冲突概率**的最佳平衡：

* 负载因子过低，导致频繁扩容
* 负载因子过高，​哈希冲突增加

【2/2】HashMap的遍历方式有哪几种？哪种效率更高？

1. ​entrySet()是效率最高的遍历方式，推荐使用。
2. forEach()+ Lambda是Java 8后的简洁语法，底层实现基于entrySet()。
3. ​keySet()返回的是所有键的集合。如果遍历时需要通过键获取值，则需要调用 map.get(key)，这会增加额外的查找操作。
4. values()返回的是所有值的集合。由于无法通过值获取键，因此适用于只需要遍历值的场景。

Java并发

【3/4】ConcurrentHashMap如何实现线程安全？JDK7和JDK8的区别？

​

**JDK7**的ConcurrentHashMap

* **分段锁**（默认16个段，因此最多支持16个线程并发操作）
* 锁粒度大（段级别），性能低

**JDK8**的ConcurrentHashMap

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

【3/5】CAS的原理是什么？存在哪些缺陷？

**CAS（Compare And Swap）无锁并发**​：

1. 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。
2. 原子性由 CPU 硬件指令（cmpxchg）保证。

**CAS 的缺陷：**

1. ​**ABA 问题**

* 值从 A 变为 B 再变回 A，CAS 无法感知中间变化。
* ​**解决**：使用版本号或时间戳（如 AtomicStampedReference引入版本号stamp）。

1. ​**自旋开销**

* 如果 CAS 失败，线程会不断自旋重试，消耗 CPU 资源。
* ​**解决**：锁机制或限制自旋次数。

1. ​**单变量限制**

* 无法直接支持多个变量的原子操作。
* ​**解决**：锁或封装为复合操作。

【3/5】AQS的核心机制是什么？举一个基于AQS实现的工具类。

AQS（AbstractQueuedSynchronizer）为构建锁和同步器提供了灵活的框架。

**AQS 的核心机制**：

1. ​通过 **state 表示资源状态**，比如锁的重入次数。
2. ​使用**​CLH 队列（**双向链表**）**​实现自旋锁的公平排队  
   线程通过 acquire() 和 release() 方法进入或离开队列。

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

1. ​支持独占锁（如 ReentrantLock）和共享锁（如 Semaphore）。

**ReentrantLock**​ 是基于 AQS 实现的独占锁：

1. 通过 state 记录锁的重入次数。
2. 实现 tryAcquire() 和 tryRelease() 方法，控制锁的获取与释放。
3. 支持公平锁和非公平锁（默认），通过 AQS 的队列机制实现公平性。

【2/3】为什么ConcurrentHashMap的size()方法需要特殊处理？

**特殊处理原因：**

1. 并发更新导致的**不准确**性​
2. 直接遍历的**性能开销**

**特殊处理方式：**

1. ​分段Segment统计（JDK7）​：  
   分别统计每个 Segment 的元素数量再累加。  
   通过多次统计，如果结果一致则返回，否则重试。
2. ​使用LongAdder计算元素数量的近似值（JDK8）​：  
   通过 baseCount 和 CounterCell 分散并发更新的竞争，减少性能开销。  
   将 baseCount 和所有 CounterCell 的值相加，得到一个总和的近似值。

【3/4】自旋锁和互斥锁在CAS中的协同作用？

1. ​​线程首先通过 CAS 尝试获取资源
2. ​若CAS失败，线程自旋重试 避免上下文切换开销，适合低竞争或短任务。
3. ​自旋仍失败，线程进入阻塞状态 释放 CPU 资源，适合高竞争或长任务。

【1/4】synchronized和ReentrantLock的性能对比？

**synchronized**：JVM 内置，自动获取释放，适合低竞争、简单锁。

**ReentrantLock**：JDK实现，手动获取释放，适合高竞争、复杂锁。

| **​特性​** | **​synchronized​** | **​ReentrantLock​** |
| --- | --- | --- |
| ​**实现机制**​ | JVM 内置 基于 Monitor 锁 | JDK 实现 基于 AQS |
| ​**锁获取方式**​ | 自动获取和释放锁 | 手动调用 lock() 和 unlock() |
| ​**锁类型**​ | 默认非公平锁 | 支持公平锁和非公平锁 |
| ​**功能扩展**​ |  | 支持高级功能： - 可中断锁 - 超时锁 - 条件变量 |
| ​**适用场景**​ | 适合低竞争、简单锁 | 适合高竞争、复杂锁 |

【2/3】线程池中的Worker类是如何利用AQS的？

Worker 类使用 AQS 的 **独占模式**，是线程池高效运行的关键设计，  
通过 state 表示线程状态，0表示空闲，1表示占用，  
结合 lock() 和 unlock() 获取释放锁，通过 tryLock() 支持对中断操作。

SpringBoot/SSM框架

【4/5】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，支持两种方式：

* **JDK 动态代理**​ 适用于目标类**实现了接口**的场景，性能好但限制多。
* **CGLIB 代理**​ 适用于目标类**未实现接口**的场景，灵活但性能低。

| **​特性​** | **​JDK 动态代理​** | **​CGLIB 代理​** |
| --- | --- | --- |
| ​**实现方式**​ | 基于接口 通过 **Proxy 类**生成**代理对象** | 基于继承 通过**字节码**生成**子类**代理 |
| ​**适用范围**​ | 目标类必须实现接口 | 目标类无需实现接口 |
| ​**限制**​ | 无法代理未实现接口的类 | 无法代理**final 类或方法** |

【4/5】Spring Bean的生命周期和作用域有哪些？

**Bean生命周期：**

1. ​**实例化**：  
   通过构造函数或工厂方法创建Bean实例。
2. ​**属性赋值**：  
   通过依赖注入进行属性赋值。
3. ​**初始化**：

* ​**前置处理：**postProcessBeforeInitialization()。
* ​**初始化方法**：
  + 执行@PostConstruct注解方法
  + InitializingBean的afterPropertiesSet()
  + 自定义init-method。
* ​**后置处理：**postProcessAfterInitialization()

1. ​**使用**
2. ​**销毁**：
   * 执行@PreDestroy注解方法
   * DisposableBean的destroy()
   * 自定义destroy-method。

**Bean的作用域：**

| **​作用域​** | **​描述​** |
| --- | --- |
| ​**Singleton**​ | 默认，每个Spring容器中只有一个Bean实例。 |
| ​**Prototype**​ | 每次请求时都会创建一个新的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 |

【2/4】SpringBoot自动配置的实现机制（@EnableAutoConfiguration）？

1. ​**启动扫描@EnableAutoConfiguration：**Spring Boot启动时扫描META-INF/spring.factories，加载自动配置类。
2. ​**条件检查@Conditional：**  
   根据类路径、配置文件、Bean是否存在判断是否加载配置类。
   1. @ConditionalOnClass
   2. @ConditionalOnMissingBean
   3. @ConditionalOnProperty
3. ​**Bean注册**：  
   符合条件的配置类注册Bean到Spring容器。
4. ​**属性绑定**@ConfigurationProperties：  
   将配置文件，如application.properties，中的属性注入Bean。

【4/5】如何解决循环依赖问题？三级缓存机制详解。

**循环依赖：**

是指两个或多个Bean相互依赖，导致无法完成初始化。

例如，Bean A依赖Bean B，而Bean B又依赖Bean A。

**三级缓存：**

1. ​**一级缓存​**：存储完全初始化后的单例Bean。
2. ​**二级缓存​**：存储提前暴露的未完全初始化的Bean（仅完成实例化，未完成属性注入和初始化）。
3. ​**三级缓存​**：存储Bean的ObjectFactory，用于生成提前暴露的Bean。

**解决循环依赖的流程：**

1. ​**Bean实例化**：

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

1. ​**属性注入**：

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

1. ​**Bean初始化**：

* 完成属性注入和初始化。
* 将Bean从二级缓存移除，放入一级缓存。

【1/4】Spring事务失效的常见场景有哪些？

1. ​**非Public方法**：  
   @Transactional注解只能作用于Public方法
2. ​**异常类型不匹配**：  
   默认只回滚RuntimeException和Error  
   如果抛出比如IOException且未配置rollbackFor，不会回滚。
3. ​**异常被捕获**：  
   如果在方法内捕获异常且未重新抛出，事务管理器无法感知异常不会回滚。
4. ​**非代理调用**：  
   同一个类中，方法A调用方法B（B有@Transactional），实际调用的是原始对象的方法，而不是代理对象的方法，而事务管理是通过代理机制实现的，B的事务不会生效。
5. ​**事务传播行为配置错误**：  
   PROPAGATION\_NOT\_SUPPORTED会挂起当前事务，导致事务失效。
6. ​**数据源未配置事务管理器**：  
   如果未正确配置DataSourceTransactionManager，事务无法生效。
7. ​**多线程调用**：  
   事务是基于ThreadLocal实现的，多线程环境下事务无法传递。
8. ​**非Spring管理Bean**：  
   如果Bean未由Spring容器管理（如手动new对象），事务注解无效。

【3/4】MyBatis的#和$的区别及SQL注入问题。

* #（推荐）：安全、预编译，适用于参数值替换。
* $：直接拼接、存在SQL注入风险，适用于动态表名、排序字段。

【2/5】SpringMVC请求处理流程中的HandlerAdapter作用？

**作用：**

1. ​**适配处理器**：根据处理器类型（如Controller、HttpRequestHandler），选择合适的适配器执行。
2. ​**调用处理方法**：通过handle()方法执行处理器的业务逻辑。
3. ​**处理返回值**：将处理器返回的ModelAndView转换为对象给视图解析器。

**常见实现：**

1. ​**RequestMapping**HandlerAdapter：用于处理@Controller和@RequestMapping注解的处理器。
2. ​**HttpRequest**HandlerAdapter：用于处理HttpRequestHandler接口的实现类。
3. ​**SimpleController**HandlerAdapter：用于处理Controller接口的实现类。

**SpringMVC流程：**

1. DispatcherServlet 接收请求：
2. HandlerMapping 查找处理器：根据请求的URL找到对应的处理器（如Controller）。
3. **HandlerAdapter** 适配并调用处理器：选择合适的适配器执行、用处理器的业务逻辑方法（如handle()）。
4. Controller 执行业务逻辑：控制器（Controller）执行业务逻辑，并返回结果（如ModelAndView或数据对象）。
5. **HandlerAdapter** 处理返回值：HandlerAdapter将处理器返回的结果（如ModelAndView）转换为适合视图解析器的格式。
6. DispatcherServlet 返回响应：DispatcherServlet将处理后的结果返回给客户端（如渲染视图或返回JSON数据）。

【1/3】SpringBoot如何集成Tomcat容器？

1. **默认集成**：通过spring-boot-starter-tomcat默认内嵌Tomcat容器。
2. **自定义配置**：通过配置文件调整Tomcat参数。
3. **切换容器**：排除Tomcat依赖并引入其他容器（如Jetty）。
4. **外部部署**：修改打包方式为war，排除内嵌Tomcat，并继承SpringBootServletInitializer。

**Tomcat默认参数：**

* **端口**：8080
* **上下文路径**：空（即/）
* **最大连接数**：10000
* **最大线程数**：200
* **最小工作线程数**：10
* **连接超时时间**：60秒
* **POST请求体大小限制**：2MB

MySQL

【2/5】B+树索引的查询过程？为什么不用哈希索引？

**B+树索引的查询过程：​**

**从根节点开始**

**逐层比较**确定下一层子节点的位置

最终定位到叶子节点（叶子节点存储了实际的数据或指向数据的指针）

如果是范围查询，叶子节点通过链表连接，可以高效地遍历满足条件的数据。

**为什么不用哈希索引？**

1. ​哈希索引只能处理等值查询，无法支持范围查询
2. ​哈希函数可能导致冲突，影响查询性能
3. ​哈希索引在数据频繁插入或删除时，维护成本较高，而B+树在动态数据环境下表现更稳定。
4. ​ B+树的叶子节点有序排列，适合顺序访问和范围扫描，而哈希索引的数据分布是随机的。

【2/4】事务隔离级别和MVCC的实现原理？

**事务隔离级别：​**

1. ​**读未提交（Read Uncommitted）​**：事务可以读取未提交的数据，可能导致脏读。
2. ​**读已提交（Read Committed）​**：事务只能读取已提交的数据，可能出现不可重复读。
3. ​**可重复读（Repeatable Read）​**：事务期间多次读取同一数据结果一致，可能出现幻读。
4. ​**串行化（Serializable）​**：最高隔离级别，事务完全串行执行，避免所有并发问题，但性能最低。

**MVCC实现原理：​**

1. ​**多版本控制**：每条记录维护多个版本，每个版本带有创建和删除的时间戳（事务ID）。
2. ​**可见性判断**：事务根据自身ID和记录的版本时间戳判断哪些数据可见：
3. ​**无锁并发**：读操作不会阻塞写操作，写操作生成新版本，旧版本仍可被其他事务读取。
4. ​**垃圾回收**：定期清理不再被任何事务引用的旧版本数据，释放空间。

【2/2】Redo Log和Undo Log的作用及写入机制？

**Redo Log（重做日志）：​**

1. ​**作用**：确保事务的持久性，记录“做了什么”事务对数据的修改，用于崩溃恢复时重放操作。
2. ​**写入机制**：

* 事务提交时，先将修改写入Redo Log Buffer。
* 每秒或事务提交时将Redo Log Buffer刷新到磁盘的Redo Log文件。
* 采用顺序写，性能高。

**Undo Log（回滚日志）：​**

1. ​**作用**：确保事务的原子性和一致性，记录“如何撤销”事务修改前的数据，用于回滚和MVCC。
2. ​**写入机制**：

* 事务修改数据前，先将旧数据写入Undo Log。
* Undo Log存储在专门的段中，支持回滚和并发读。
* 事务提交后，Undo Log不会立即删除，可能被MVCC或其他事务引用。

【1/3】覆盖索引和索引下推的区别？

**覆盖索引（Covering Index）：​**

1. ​**定义**：查询所需的所有字段都包含在索引中，无需回表查询主键或数据行。
2. ​**优点**：减少I/O操作，提升查询性能。
3. ​**场景**：适用于SELECT语句中只查询索引字段的情况。

**索引下推（Index Condition Pushdown, ICP）：​**

1. ​**定义**：在存储引擎层利用索引过滤数据，减少回表次数。
2. ​**优点**：减少回表操作，降低查询开销。
3. ​**场景**：适用于WHERE条件中包含索引字段和非索引字段的查询。

【2/4】死锁的产生条件及排查方法（show engine innodb status）。

**死锁的产生四大条件：​**

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

**排查方法（SHOW ENGINE INNODB STATUS）：​**

1. ​**执行命令**：在MySQL中运行SHOW ENGINE INNODB STATUS;。
2. ​**查找死锁信息**：在输出中查找LATEST DETECTED DEADLOCK部分。
3. ​**分析死锁日志**：

* 查看涉及的事务ID（TRANSACTION）。
* 分析事务持有的锁（HOLDS THE LOCK(S)）和等待的锁（WAITING FOR THIS LOCK TO BE GRANTED）。
* 定位造成循环等待的SQL语句。

1. ​**解决死锁**：根据日志调整事务逻辑，如调整锁顺序、减少事务粒度或重试机制。

【1/2】Change Buffer对写操作的优化原理？

1. ​**定义**：Change Buffer是InnoDB的一种优化机制
2. ​**优化原理**：

* ​**延迟写入**：当修改 非唯一 二级索引时，不直接写入磁盘，而是先将修改记录到Change Buffer中。
* ​**批量合并**：当相关索引页被加载到内存时，再将Change Buffer中的修改合并到索引页，减少随机I/O。
* ​**减少磁盘操作**：通过延迟写入和批量合并，显著降低对磁盘的频繁写入，提升写性能。

1. ​**适用场景**：适用于写多读少、非唯一二级索引的场景。

【2/5】分库分表后全局ID生成方案有哪些？

**分库分表后全局ID生成方案：​**

1. ​**UUID**：

* ​**优点**：简单，本地生成，无需中心化服务。
* ​**缺点**：无序，存储空间大，索引效率低。

1. ​**数据库自增ID**：

* ​**方案**：每个库/表设置不同的自增步长和起始值（如库1: 1, 3, 5；库2: 2, 4, 6）。
* ​**优点**：简单，有序。
* ​**缺点**：扩展性差，依赖数据库。

1. ​**Snowflake算法**：

* ​**原理**：64位ID = 时间戳 + 机器ID + 序列号。
* ​**优点**：分布式生成，高性能，有序。
* ​**缺点**：依赖机器时钟，时钟回拨可能引发问题。

1. ​**Redis自增ID**：

* ​**原理**：利用Redis的原子操作（如INCR）生成全局ID。
* ​**优点**：高性能，简单易用。
* ​**缺点**：依赖Redis，存在单点故障风险。

1. ​**号段模式（Segment）​**：

* ​**原理**：从数据库预分配一批ID号段，本地缓存使用。
* ​**优点**：减少数据库访问，性能高。
* ​**缺点**：号段用尽需重新申请，可能短暂阻塞。

【1/5】主从同步延迟的解决方案？

1. ​**优化主库性能**：

* 提升主库硬件配置（如CPU、内存、磁盘）。
* 优化SQL语句，减少锁争用和大事务。

1. ​**优化从库性能**：

* 开启并行复制（如MySQL的MTS），提升从库处理能力。
* 提升从库硬件配置，减少性能瓶颈。

1. ​**减少网络延迟**：

* 主从库部署在同一机房或低延迟网络环境。
* 使用高性能网络设备。

1. ​**调整复制策略**：

* 使用半同步复制，确保主库提交时至少一个从库已接收日志。
* 异步复制场景下，监控延迟并优化。

1. ​**分库分表**：

* 将大库拆分为多个小库，分散主库写入压力。
* 减少单库数据量，降低同步负担。

1. ​**读写分离**：

* 将延迟敏感的读操作路由到主库。
* 对非实时性要求的读操作，容忍一定延迟。

Redis

【2/5】Redis单线程模型为什么能高效处理请求？

1. ​**纯内存操作**：Redis数据存储在内存中，读写速度极快。
2. ​**非阻塞I/O**：Redis使用多路复用技术（如epoll、kqueue），单线程可以同时处理多个客户端请求，避免了线程切换的开销。
3. ​**避免锁竞争**：单线程模型无需考虑多线程并发控制，减少了锁竞争和上下文切换的开销。
4. **数据结构优化**：Redis内置高效的数据结构（如哈希表、跳表）
5. ​**事件驱动**：Redis基于事件驱动模型，高效处理网络事件和定时任务。

【3/5】缓存雪崩/穿透/击穿的解决方案？

Redis缓存穿透

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

解决：

* 使用布隆过滤器，过滤掉不存在的请求，避免直接访问数据库。
* 对查询结果进行缓存，即使是不存在的数据，也可以缓存一个标识，以减少对数据库的请求。

Redis缓存击穿

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

解决：

* 使用互斥锁，确保同一时间只有一个请求可以去数据库查询并更新缓存。
* 热点数据永不过期。

Redis 缓存雪崩

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

解决：

* 采用随机过期时间策略，避免多个数据同时过期。
* 使用双缓存策略，将数据同时存储在两层缓存中，减少数据库直接请求。
  + 步骤1：读取数据时的逻辑
    - 首先尝试从主缓存中读取数据。
    - 如果主缓存中没有数据，则尝试从备份缓存中读取数据。
    - 如果备份缓存也没有数据，则从数据库中读取数据，并同时更新主缓存和备份缓存。
  + 步骤2：写入数据时的逻辑
    - 当有新的数据写入缓存时，同时更新主缓存和备份缓存。
    - 为主缓存设置一个短的过期时间，为备份缓存设置一个较长的过期时间。
    - 当主缓存失效时，备份缓存可以继续提供数据，减少对数据库的直接访问。

【1/5】RDB和AOF的优缺点及混合持久化机制？

**RDB（快照持久化）：**

**优点**：

* ​**高性能：生成快照时对性能影响小，适合大规模数据恢复。**
* **​文件紧凑：生成的RDB文件体积小，便于备份和传输。**
* **​恢复速度快：直接加载快照文件，恢复效率高。**

**缺点**：

* ​**数据丢失风险**：定期生成快照，可能导致最后一次快照后的数据丢失。
* ​**频繁操作开销大**：数据量大时，生成快照可能耗时较长。

**AOF（追加日志持久化）：**

**优点**：

* ​**数据完整性高：记录每次写操作，数据丢失风险低。**
* **​可读性强：AOF文件是文本格式，便于人工检查和修复。**

**缺点**：

* ​**文件体积大：记录所有写操作，文件可能比RDB大。**
* **​恢复速度慢：需要逐条重放日志，恢复效率较低。**
* **​性能开销：频繁写日志可能影响性能。**

**混合持久化机制：**

**原理**：结合RDB和AOF的优点，先使用RDB生成快照，然后在快照的基础上追加AOF日志。

**优点**：

* ​**恢复速度快：优先加载RDB快照，再重放少量AOF日志。**
* **​数据完整性高：AOF日志补充了RDB快照后的数据变化。**

**适用场景**：适合对数据完整性和恢复速度要求高的场景。

【1/5】Redis Cluster的slot分配算法？

Redis Cluster采用**哈希槽（Hash Slot）​**机制，将整个键空间划分为**16384个槽（slot）​**，每个槽对应一个哈希值范围（0-16383）。

**分配算法：​**

1. ​**键的映射**：对每个键计算CRC16哈希值，然后对16384取模，得到对应的slot编号。
2. ​**slot分配**：Redis Cluster将这些slot均匀分配到集群中的各个节点，每个节点负责一部分slot。
3. ​**动态调整**：当集群节点增减时，slot可以动态重新分配，确保负载均衡。

**亮点：​**

* ​**高扩展性**：通过slot机制，Redis Cluster可以轻松扩展节点数量。
* ​**负载均衡**：slot均匀分配，避免单点压力过大。
* ​**数据迁移**：slot可以在节点间迁移，支持动态扩容和缩容。

【1/5】大Key问题如何定位和解决？

**大Key问题定位：​**

1. ​**监控工具**：使用Redis自带的MEMORY USAGE命令或监控工具（如RedisInsight、Redis监控平台）分析Key的内存占用。
2. ​**扫描命令**：通过SCAN命令遍历所有Key，结合STRLEN、HLEN、LLEN等命令，统计Key的大小。
3. ​**日志分析**：通过慢查询日志（slowlog）定位访问频率高或耗时的Key。

**大Key问题解决：​**

1. ​**拆分**：将大Key拆分为多个小Key（如Hash拆分为多个子Hash）。
2. ​**压缩**：对Value进行压缩（如使用gzip或自定义压缩算法）。
3. ​**冷热分离**：将不频繁访问的数据迁移到其他存储（如冷数据存储到磁盘）。
4. ​**缓存优化**：避免存储过大的Value，改用分页或分段加载。
5. ​**过期策略**：为大Key设置合理的过期时间，避免长期占用内存。

**亮点：​**

* ​**高效定位**：结合工具和命令快速定位大Key。
* ​**灵活解决**：通过拆分、压缩、冷热分离等多种手段针对性解决。
* ​**预防为主**：优化设计，避免大Key的产生。

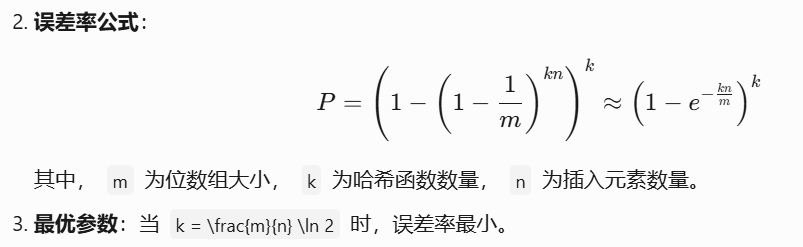
【2/4】布隆过滤器的实现原理及误差率计算？

**实现原理：​**

1. ​**数据结构**：布隆过滤器是一个由m个二进制位组成的位数组，初始值全为0。
2. ​**哈希函数**：使用k个独立的哈希函数，将元素映射到位数组的k个位置。
3. ​**添加元素**：将元素通过k个哈希函数计算，将对应的位数组位置置为1。
4. ​**查询元素**：通过k个哈希函数计算，若所有对应位置均为1，则元素可能存在；若有任意位置为0，则元素一定不存在。

**误差率计算：​**

1. ​**误差来源**：哈希冲突导致误判，即不同元素可能映射到相同的位数组位置。
2. ​**误差率公式**：



【1/4】Redis事务和Lua脚本的原子性区别？

**Redis事务的原子性：​**

1. ​**实现方式**：通过MULTI、EXEC、DISCARD命令实现事务。
2. ​**原子性特点**：

* ​**批量执行**：EXEC时，事务中的所有命令按顺序一次性执行。
* ​**非隔离性**：事务执行期间，其他客户端命令可能会插入执行。
* ​**无回滚机制**：如果某条命令失败，后续命令仍会执行，不支持回滚。

**Lua脚本的原子性：​**

1. ​**实现方式**：通过EVAL或EVALSHA执行Lua脚本。
2. ​**原子性特点**：

* ​**完全原子**：脚本执行期间，Redis不会执行其他命令，保证脚本内的所有操作原子性。
* ​**隔离性**：脚本执行时，其他客户端命令会被阻塞，直到脚本执行完毕。
* ​**灵活性**：支持复杂逻辑和条件判断，且脚本可以复用。

**区别总结：​**

1. ​**原子性级别**：Lua脚本是真正的原子操作，而Redis事务是批量执行，不具备完全原子性。
2. ​**隔离性**：Lua脚本执行期间完全隔离，事务则可能被其他命令插入。
3. ​**回滚机制**：Lua脚本不支持回滚，但可以通过逻辑控制实现类似效果；事务不支持回滚。

【1/5】热Key问题的解决方案（本地缓存/分片）？

**热Key问题定义：​**  
热Key是指访问频率极高的Key，导致单个Redis节点或实例负载过高，影响性能和稳定性。

**解决方案：​**

1. ​**本地缓存（Local Cache）：​**

* ​**实现方式**：在应用层（如JVM）缓存热Key的值，减少对Redis的直接访问。
* ​**优点**：降低Redis压力，提升响应速度。
* ​**缺点**：数据一致性难以保证，需设置合理的过期时间或更新策略。

1. ​**分片（Sharding）：​**

* ​**实现方式**：将热Key分散到多个Redis实例或节点，通过哈希算法或手动分片均衡负载。
* ​**优点**：彻底解决单点压力问题，提升整体性能。
* ​**缺点**：实现复杂，需维护分片逻辑，且可能增加网络开销。

**补充方案：​**

* ​**读写分离**：将读请求分散到从节点，减轻主节点压力。
* ​**缓存预热**：提前加载热Key到缓存，避免高峰期的集中访问。
* ​**限流降级**：对热Key的访问进行限流，防止系统过载。

计算机网络

TCP三次握手的详细过程和状态变化？

HTTP/1.1到HTTP/2的核心改进有哪些？

TIME\_WAIT状态的作用及过多时的解决方案？

TCP快速重传和超时重传的区别？

HTTPS的TLS握手过程（ECDHE算法）？

QUIC协议如何解决队头阻塞问题？

TCP拥塞控制中的BBR算法原理？

WebSocket协议如何实现全双工通信？

操作系统

进程和线程的区别？协程的优势是什么？

虚拟内存的作用及页面置换算法（LRU实现）？

select/poll/epoll的区别及边缘触发模式？

用户态和内核态切换的开销来源？

自旋锁在单核CPU下是否有意义？

mmap系统调用的实现原理？

进程间通信共享内存和消息队列的对比？

零拷贝技术（sendfile）的实现原理？

Linux命令

如何用grep查找包含"error"的日志并统计次数？

awk如何实现按列求和（{sum+=$3} END{print sum}）？

如何用sed批量替换文件中的字符串？

strace和perf工具的作用及使用场景？

lsof命令如何查看被删除但未释放的文件？

如何用tcpdump抓取指定端口的SYN包？

使用dd命令测试磁盘IO性能的方法？

如何通过/proc文件系统查看进程内存映射？

Nginx

【1/4】Nginx 的反向代理是如何工作的？

Nginx反向代理通过以下步骤将客户端请求透明地转发到后端服务器，隐藏后端细节，提升安全性和性能：

1. 接收请求
2. **转发请求：**通过proxy\_pass指令将请求转发到指定的后端服务器
3. **修改请求头：**使用proxy\_set\_header修改请求头

* 日志记录：后端需要记录客户端IP，而非Nginx的IP。
* 域名路由：后端服务根据Host头区分不同站点。
* 协议识别：后端需要知道客户端是否使用HTTPS。

1. 返回响应

【1/3】Nginx反向代理和负载均衡配置示例？

* ​**反向代理**：通过**proxy\_pass**转发请求到后端服务器。
* ​**负载均衡**：使用**upstream**模块配置多台服务器，支持
  + 轮询（默认）：依次分发请求。
  + 权重（weight）：根据权重分配请求，权重越高分配越多。
  + 备份（backup）：仅在主服务器不可用时使用。
  + IP哈希策略（ip\_hash）：根据客户端IP固定分发到同一服务器。

【1/2】Nginx热更新配置文件的原理（信号机制）？

Nginx通过HUP（**Hang Up**）信号实现热更新，无需重启服务即可加载新配置，保证服务零停机，高效可靠。

1. ​​**验证配置**：使用**nginx -t**验证配置文件语法正确性。
2. ​**发送信号**：向主进程发送HUP信号（**kill -HUP <主进程PID>**）。
3. ​**重新加载**：主进程接收HUP信号后重新加载配置文件并启动新的工作进程。
4. ​**平滑过渡**：旧工作进程处理完当前请求后退出，新工作进程接管请求。

【1/3】Nginx 如何优化静态资源的加载速度？

1. ​**启用Gzip压缩：**压缩静态资源（如CSS、JS、HTML），减少传输体积。
2. **静态资源分离**：将静态资源部署到独立的域名或CDN，减少主域名请求压力。
3. **设置缓存头：**通过**expires**指令设置静态资源的缓存时间，减少重复请求。
4. **启用文件缓存：**使用open\_file\_cache缓存文件描述符，减少磁盘I/O。
5. **启用Sendfile：**直接在**内核中传输**文件，减少用户态和内核态的切换。
6. **启用HTTP/2**：使用HTTP/2协议，支持多路复用，提升加载效率。

Docker

【/5】Docker镜像分层机制及构建优化方法？

**Docker镜像分层机制：​**

1. ​**分层结构**：Docker镜像由多个只读层（Layer）组成，每个层代表文件系统的一部分。层之间通过联合文件系统（UnionFS）叠加，形成最终的镜像。
2. ​**共享与复用**：层是共享的，多个镜像可以复用相同的层，减少存储空间占用。
3. ​**写时复制（Copy-on-Write）​**：容器启动时，基于镜像层创建可写层，修改操作仅在可写层进行，不影响底层镜像。

**构建优化方法：​**

1. ​**精简基础镜像**：选择轻量级基础镜像（如Alpine），减少不必要的依赖。
2. ​**合并指令**：将多个RUN指令合并为一条，减少层数。
3. ​**利用缓存**：将不常变化的指令（如安装依赖）放在前面，充分利用缓存加速构建。
4. ​**多阶段构建**：使用多阶段构建（Multi-stage Build），在最终镜像中只保留必要的文件，减少镜像体积。
5. ​**清理无用文件**：在构建过程中及时删除临时文件，避免冗余。

【1/4】Docker网络模式（bridge/host/none）的区别？

1. ​**Bridge模式（默认）​**：

* 容器通过虚拟网桥（docker0）与主机通信。
* 每个容器分配独立IP，容器间通过IP通信。
* 容器与外部网络通过NAT转发通信。
* ​**适用场景**：隔离性好，适合通用场景，如微服务架构。

1. ​**Host模式**：

* 容器直接使用主机的网络栈，与主机共享IP和端口。
* 无独立IP，性能更高，但隔离性差。
* ​**适用场景**：性能高，适合低延迟需求。

1. ​**None模式**：

* 容器无网络接口，完全隔离。
* 仅支持本地进程间通信（IPC）。
* ​**适用场景**：完全隔离，适合极端安全要求或自定义网络配置。

【1/4】如何用Docker Compose编排多容器应用？

1. ​**定义服务**：在docker-compose.yml中定义每个容器为服务，指定镜像、端口、环境变量等配置。
2. ​**配置网络**：默认创建bridge网络，服务间通过服务名通信；可自定义网络实现更复杂的拓扑。
3. ​**数据持久化**：使用volumes挂载主机目录或命名卷，确保数据持久化。
4. ​**依赖管理**：通过depends\_on指定服务启动顺序，确保依赖服务先启动。
5. ​**一键启停**：使用docker-compose up启动所有服务，docker-compose down停止并清理。

【1/5】Kubernetes中Pod和Deployment的关系？

Pod是运行容器的基本单元，Deployment是管理Pod的高级抽象

**​**

**Pod**：

* Kubernetes的最小调度单元，包含一个或多个容器。
* 容器共享网络和存储，协同工作。
* 生命周期短暂，不直接用于生产部署。

**Deployment**：

* 用于管理Pod的声明式对象，确保指定数量的Pod副本始终运行。
* 提供滚动更新、回滚和扩缩容功能。
* 通过定义Pod模板（template）创建和管理Pod。

**关系**：

1. ​**创建**：Deployment根据Pod模板创建Pod。
2. ​**管理**：Deployment监控Pod状态，确保副本数符合预期。
3. ​**更新**：Deployment通过滚动更新策略替换旧Pod为新Pod。
4. ​**扩展**：Deployment可动态调整Pod副本数。

【1/4】容器与虚拟机的性能差异来源？

1. ​**虚拟化层级**：

* ​**虚拟机**：通过Hypervisor虚拟化硬件，运行完整的操作系统（Guest OS），开销较大。
* ​**容器**：共享主机操作系统内核，无额外操作系统开销，性能更高。

1. ​**启动速度**：

* ​**虚拟机：**启动需要加载完整操作系统，速度较慢。
* **​容器：**直接启动应用进程，秒级启动。

1. ​**资源占用**：

* ​**虚拟机：**每个虚拟机独占操作系统资源，内存和存储占用较高。
* **​容器：**共享主机内核，资源占用更少，支持更高密度部署。

1. ​**隔离性**：

* ​**虚拟机：**硬件级隔离，安全性更高。
* **​容器：**进程级隔离，依赖内核特性，安全性相对较低。

【1/4】如何通过cgroups限制容器资源使用？

1. ​**CPU限制**：使用--cpus参数限制容器可使用的CPU核心数。

* 示例：docker run --cpus="1.5" <image>，限制容器使用1.5个CPU核心。

1. ​**内存限制**：使用--memory参数限制容器可使用的内存大小。

* 示例：docker run --memory="512m" <image>，限制容器使用512MB内存。

1. ​**内存交换限制**：使用--memory-swap参数限制容器内存和交换空间的总大小。

* 示例：docker run --memory="512m" --memory-swap="1g" <image>，限制容器使用512MB内存和1GB交换空间。

1. ​**磁盘I/O限制**：使用--device-read-bps和--device-write-bps限制磁盘读写速率。

* 示例：docker run --device-read-bps="/dev/sda:1mb" <image>，限制容器对/dev/sda的读取速率为1MB/s。

1. ​**进程数限制**：使用--pids-limit参数限制容器内的最大进程数。

* 示例：docker run --pids-limit="100" <image>，限制容器内最多运行100个进程。

【0/1】Docker 的核心组件有哪些？它们的作用是什么？

1. Docker **Engine**：运行容器的**引擎**
2. Docker **Images**：容器运行所需的**文件和依赖**
3. Docker **Containers**：运行的**容器**实例
4. Docker **Hub**：**公共镜像仓库**

【1/1】什么是 Dockerfile？列出常见指令及其用途？

Dockerfile 用于**构建镜像**的文件

常见指令：

* FROM：指定基础镜像
* RUN：运行命令
* CMD：容器启动时执行的命令

容器化技术

【1/1】容器与虚拟机的区别是什么？

容器：轻量级的，直接共享宿主机的操作系统内核。

虚拟机：需要完整的操作系统，开销较大。

【1/1】Kubernetes 如何管理容器化应用？

Kubernetes 提供容器编排功能，包括：

* 自动化部署
* 扩展
* 负载均衡
* 服务发现

【1/1】容器化技术的优点有哪些？

* 提高开发效率
* 易于迁移
* 更好的资源利用和隔离性

Devops

【1/1】什么是 CI/CD？它在 DevOps 中的作用是什么？

* CI（持续集成）：自动化构建和测试代码
* CD（持续部署）：将通过测试的代码自动部署到生产环境

【0/1】DevOps 常用工具有哪些？

* CI/CD工具：Jenkins、GitLab CI、Circle CI
* 配置管理：Ansible、Chef、Puppet
* 监控：Prometheus、Grafana

【1/1】什么是基础设施即代码（IaC）？

使用代码定义和管理基础设施（如 YAML、Terraform），实现自动化、可重复性、可版本化。