一句话总结 + 掌握程度/重点程度 + 关键词 + 详细解释

Java集合

HashMap底层结构是什么？如何解决哈希冲突？

**底层结构：**

JDK1.8之前：数组+链表结构

JDK1.8之后：数组+链表+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，以优化查询效率（时间复杂度从O(n)降至O(logn)）
* 红黑树退化条件：当链表长度≤6时，红黑树会退化为链表。

**解决哈希冲突的机制：**

1. 拉链法：哈希冲突发生时，以链表形式存储在同一个数组索引
2. 红黑树优化（JDK1.8后）：当链表长度≥8且数组容量≥64时，链表转换为红黑树
   1. 扩容：通过增加数组容量，重新分配节点，减少哈希冲突，适合小容量场景。
   2. 树化：通过将链表转换为红黑树，优化查询性能，适合大容量场景。
3. 哈希函数优化：
   1. 二次扰动函数：混合高位和低位、优化索引分布、减少哈希冲突  
      由于先计算哈希值、再计算索引，所以如果哈希值的高位变化较大，但低位变化较小，直接用 (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混合，使哈希值的每一位都参与索引计算，减少冲突概率  
      步骤1：获取 key.hashCode()，记为 h。

步骤2：将 h 右移16位（h >>> 16），得到 h 的高16位。

步骤3：将 h 与 h >>> 16 进行 异或运算（^），得到最终的哈希值。

* 1. 计算索引的公式为(n-1) & hash，n是数组容量，确保索引分布均匀

1. 动态扩容机制：当元素数量超过阈值（容量×负载因子，默认0.75）时，数组扩容为原来的2倍，重新分配节点以减少冲突
   1. 负载因子过高会增加哈希冲突，过低则导致频繁扩容
   2. 数组容量为2的幂次：保证(n-1) & hash等价于取模运算，同时扩容时只需判断高位是否为1来分配新位置（无需重新计算所有哈希）

详细描述HashMap的扩容机制（触发条件、rehash过程）。

**触发条件**

1. ​元素数量超过当前容量（Capacity）与负载因子（Load Factor）的乘积。负载因子和哈希冲突成正比。
2. ​当链表长度超过阈值（默认8）但数组容量未达到64时，HashMap会优先扩容而非转换为红黑树

**扩容过程（Rehash）**

1. ​创建新数组​
2. ​重新计算索引（Rehash）​​
   1. 索引计算规则：通过 e.hash & (newCap - 1) 确定元素在新数组中的位置。由于容量是2的幂次方，扩容后高位差异可能导致元素被分配到原位置或原位置+旧容量的新位置。
   2. 链表拆分：旧链表中的节点根据哈希值的高位（如 hash & oldCap）分为两组：低位组：保留在原索引位置。高位组：迁移到 原索引 + 旧容量 的位置。
3. ​红黑树的处理​:红黑树节点同样按上述规则拆分。若拆分后的节点数≤6，则退化为链表；若仍满足红黑树条件（节点数≥8且数组容量≥64），则保持树结构。

​

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. **数据覆盖**

多个线程同时进行插入、删除或修改操作时，可能会导致数据覆盖或丢失。

例如，两个线程同时插入不同的键值对，可能会覆盖对方的操作。

1. **扩容死循环**

在JDK 1.7中，HashMap的扩容是通过头插法将链表节点迁移到新数组中的。

在并发扩容时，多个线程同时操作链表，可能会导致链表形成环形结构，从而引发死循环。

1. **树化异常**

在JDK 1.8中，链表转红黑树的操作是线程不安全的。

多个线程同时进行树化操作，可能会导致红黑树结构异常。

**如何解决HashMap线程不安全的问题？**

1. **使用 Collections.synchronizedMap**：在方法级别加锁，性能低。
2. **使用 ConcurrentHashMap**：采用分段锁（JDK 1.7）或CAS+ synchronized（JDK 1.8），性能高。

红黑树在HashMap中的具体应用条件是什么？

红黑树用于优化高哈希冲突场景，将查找时间复杂度从O(n)降至O(log n)：

1. ​**链表长度达到阈值（默认8）​**​
2. ​**数组容量达到阈值（默认64）​**​：若容量不足64，优先扩容数组。

**退化条件**​：  
当红黑树节点数≤6时，红黑树退化为链表，以节省内存。

为什么Java 8将链表转红黑树的阈值设为8？

1. ​根据泊松分布，哈希冲突时链表长度达到8的概率极低，约为千万分之一
2. ​链表长度≤8时，链表性能足够且内存占用更低；  
   长度>8时，红黑树显著提升查找效率。
3. ​红黑树退化为链表的阈值为6，避免频繁转换（如7→8→7）

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。基于泊松分布，HashMap的查找性能接近O(1)。

* ​**负载因子过高（如1.0）​**哈希冲突概率增加，查找性能下降。
* ​**负载因子过低（如0.5）​**频繁扩容，增加内存开销。

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。

forEach()+ Lambda是Java 8后的简洁语法，效率与entrySet()相同。

1. ​keySet()和values()适用于特定场景，效率较低。

Java并发（ConcurrentHashMap、CAS）

ConcurrentHashMap如何实现线程安全？JDK7和JDK8的区别？

* ​**JDK7**：通过分段锁实现线程安全，锁粒度较大，性能较好但不如 JDK8。
* ​**JDK8**：通过 CAS + synchronized 实现线程安全，锁粒度更小，性能更高，数据结构更优化。

CAS的原理是什么？存在哪些缺陷？

CAS 通过**硬件指令**实现**无锁并发**，性能高效，但存在 ​**ABA 问题**、**自旋开销**​ 和 ​**单变量限制**​ 等缺陷，需结合场景优化使用。

**CAS 的原理**

CAS（Compare And Swap）是一种 ​**无锁并发**​ 机制，通过 ​**硬件指令**​ 实现原子操作。其核心原理是：

1. ​**比较并交换**：
   * 比较当前值（V）与预期值（E），如果相等，则将新值（N）写入；否则不操作。
   * 操作是原子的，由 CPU 硬件指令（如 cmpxchg）保证。
2. ​**伪代码**：

java

if (V == E) {

V = N;

return true;

} else {

return false;

}

**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）。
2. ​**自旋开销**：
   * 如果 CAS 失败，线程会不断重试（自旋），消耗 CPU 资源。
   * ​**解决**：结合锁机制或限制自旋次数。
3. ​**只能保证单个变量的原子性**：
   * 无法直接支持多个变量的原子操作。
   * ​**解决**：使用锁或封装为复合操作。

AQS的核心机制是什么？举一个基于AQS实现的工具类。

AQS 通过 ​**状态管理**、**线程排队**​ 和 ​**模板方法**​ 机制，为构建锁和同步器提供了灵活的基础框架。ReentrantLock 是其典型实现，广泛应用于并发编程。

**AQS 的核心机制**

AQS（AbstractQueuedSynchronizer）是 Java 并发包的核心框架，用于构建锁和同步器。其核心机制包括：

1. ​**状态管理**：
   * 通过 volatile int state 表示资源状态（如锁的占用数）。
   * 提供 getState()、setState() 和 compareAndSetState() 方法操作状态。
2. ​**线程排队**：
   * 使用 ​**CLH 队列**​（双向链表）管理等待线程，实现公平性。
   * 线程通过 acquire() 和 release() 方法进入或离开队列。
3. ​**模板方法**：
   * 提供 tryAcquire() 和 tryRelease() 等模板方法，由子类实现具体逻辑。
4. ​**独占与共享模式**：
   * 支持独占锁（如 ReentrantLock）和共享锁（如 Semaphore）。

**基于 AQS 实现的工具类**

**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，通过 AQS 的队列机制实现公平性。

为什么ConcurrentHashMap的size()方法需要特殊处理？

ConcurrentHashMap 的 size() 方法需要特殊处理，以解决 ​**并发更新导致的不准确性**​ 和 ​**直接遍历的性能开销**。通过分段统计或基于计数器的优化，实现高效且近似准确的元素数量统计。

**ConcurrentHashMap 的 size() 方法为什么需要特殊处理？**

**原因**：ConcurrentHashMap 的高并发设计导致直接计算元素数量（size）存在挑战：

1. ​**并发更新**：
   * 多个线程同时插入或删除元素，直接遍历统计会导致结果不准确。
2. ​**性能开销**：
   * 遍历整个哈希表计算 size 在高并发场景下性能低下。

**特殊处理方式**

1. ​**分段统计（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. ​**基于计数器的优化（JDK8）​**：
   * 使用 LongAdder 或类似机制维护元素数量的近似值。
   * 通过 baseCount 和 CounterCell 分散并发更新的竞争，减少性能开销。

自旋锁和互斥锁在CAS中的协同作用？

自旋锁和互斥锁在 CAS 中协同作用，通过 ​**CAS 尝试 → 自旋等待 → 互斥阻塞**​ 的分级策略，平衡 ​**响应速度**​ 和 ​**资源消耗**，适应不同竞争强度的并发场景。

自旋锁和互斥锁在 CAS 中协同工作，分别针对 ​**低竞争**​ 和 ​**高竞争**​ 场景，优化并发性能：

1. ​**自旋锁的作用**：
   * ​**场景**：当资源竞争较低时，线程通过 CAS 不断重试（自旋），避免上下文切换的开销。
   * ​**优点**：响应速度快，适合短任务或低竞争场景。
   * ​**缺点**：自旋会消耗 CPU 资源，不适合高竞争或长任务。
2. ​**互斥锁的作用**：
   * ​**场景**：当资源竞争较高或自旋失败时，线程进入阻塞状态（互斥锁），释放 CPU 资源。
   * ​**优点**：避免 CPU 资源浪费，适合高竞争或长任务。
   * ​**缺点**：上下文切换带来额外开销。
3. ​**协同机制**：
   * ​**CAS 尝试**：线程首先通过 CAS 尝试获取资源，若成功则直接执行。
   * ​**自旋等待**：若 CAS 失败，线程进入自旋状态，短暂重试多次。
   * ​**互斥阻塞**：若自旋仍失败，线程进入阻塞状态，等待唤醒。

如何用CAS实现一个无锁队列？

通过 CAS 实现无锁队列，核心是利用 CAS 的原子性更新 head 和 tail，结合循环重试机制，确保线程安全的入队和出队操作，适用于高并发场景。

**关键点**

1. ​**CAS 更新**：通过 CAS 确保 head 和 tail 的原子性更新。
2. ​**循环重试**：如果 CAS 失败，则重试，直到成功。
3. ​**无锁设计**：避免使用锁，减少线程阻塞和上下文切换。

synchronized和ReentrantLock的性能对比？

* **synchronized**：简单易用，JVM 内置优化，适合低竞争场景。
* ​**ReentrantLock**：功能强大，性能更优，适合高竞争或需要高级功能的场景。
* ​**选择依据**：根据并发强度、功能需求和代码复杂度权衡选择。

**synchronized 和 ReentrantLock 的性能对比**

| **​特性​** | **​synchronized​** | **​ReentrantLock​** |
| --- | --- | --- |
| ​**实现机制**​ | JVM 内置，基于 Monitor 锁 | JDK 实现，基于 AQS（AbstractQueuedSynchronizer） |
| ​**锁获取方式**​ | 自动获取和释放锁 | 手动获取和释放锁（需调用 lock() 和 unlock()） |
| ​**锁类型**​ | 非公平锁（默认） | 支持公平锁和非公平锁（可配置） |
| ​**性能**​ | 低竞争场景下性能较好（JVM 优化） | 高竞争场景下性能更优（灵活控制） |
| ​**功能扩展**​ | 功能简单，仅支持基本的锁操作 | 支持高级功能： - 可中断锁 - 超时锁 - 条件变量（Condition） |
| ​**代码复杂度**​ | 简单易用，无需手动管理锁 | 复杂，需手动管理锁，易遗漏 unlock() |
| ​**适用场景**​ | 适合低竞争、简单锁场景 | 适合高竞争、复杂锁场景 |

线程池中的Worker类是如何利用AQS的？

Worker 类通过继承 AQS，利用其独占锁机制，实现了任务执行的线程安全控制。通过 state 管理线程状态，结合 lock() 和 unlock() 方法，确保任务执行的原子性和中断能力，是线程池高效运行的关键设计。

**线程池中的 Worker 类如何利用 AQS**

Worker 类是线程池中执行任务的核心单元，它通过继承 ​**AQS（AbstractQueuedSynchronizer）​**​ 实现任务的独占锁机制，确保线程安全地执行任务。具体利用方式如下：

1. ​**独占锁机制**：
   * Worker 类使用 AQS 的 ​**独占模式**，通过 state 表示线程是否正在执行任务。
   * state = 0：线程空闲；state = 1：线程正在执行任务。
2. ​**锁的获取与释放**：
   * ​**获取锁**：在任务执行前调用 lock() 方法，将 state 从 0 改为 1，表示线程占用。
   * ​**释放锁**：在任务执行后调用 unlock() 方法，将 state 从 1 改为 0，表示线程空闲。
3. ​**任务执行流程**：
   * 线程池从队列中获取任务后，通过 Worker 的 runWorker() 方法执行任务。
   * 在任务执行前后，分别调用 lock() 和 unlock()，确保任务执行的原子性。
4. ​**中断控制**：
   * 通过 AQS 的 tryLock() 方法，支持对 Worker 的中断操作，用于线程池的关闭或任务取消。

SpringBoot/SSM框架

Spring AOP的实现原理？JDK动态代理和CGLIB的区别？

Spring Bean的生命周期和作用域有哪些？

SpringBoot自动配置的实现机制（@EnableAutoConfiguration）？

如何解决循环依赖问题？三级缓存机制详解。

Spring事务失效的常见场景有哪些？

MyBatis的#{}和${}的区别及SQL注入问题。

SpringMVC请求处理流程中的HandlerAdapter作用？

SpringBoot如何集成Tomcat容器？

MySQL

B+树索引的查询过程？为什么不用哈希索引？

事务隔离级别和MVCC的实现原理？

Redo Log和Undo Log的作用及写入机制？

覆盖索引和索引下推的区别？

死锁的产生条件及排查方法（show engine innodb status）。

Change Buffer对写操作的优化原理？

分库分表后全局ID生成方案有哪些？

主从同步延迟的解决方案？

Redis

Redis单线程模型为什么能高效处理请求？

缓存雪崩/穿透/击穿的解决方案？

RDB和AOF的优缺点及混合持久化机制？

Redis Cluster的slot分配算法？

大Key问题如何定位和解决？

布隆过滤器的实现原理及误差率计算？

Redis事务和Lua脚本的原子性区别？

热Key问题的解决方案（本地缓存/分片）？

网络协议（TCP/IP）

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

Nginx 的反向代理是如何工作的？

Nginx反向代理通过：

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修改或添加请求头，确保后端服务器获取正确的客户端信息。
   * 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。
4. 返回响应

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。

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。

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Nginx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。

Nginx 如何优化静态资源的加载速度？

​启用Gzip压缩：压缩静态资源（如CSS、JS、HTML），减少传输体积。

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。

启用文件缓存：使用open\_file\_cache缓存文件描述符，减少磁盘I/O。

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。

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。

启用HTTP/2：使用HTTP/2协议，支持多路复用，提升加载效率。

Docker

Docker镜像分层机制及构建优化方法？

Docker网络模式（bridge/host/none）的区别？

如何用Docker Compose编排多容器应用？

Kubernetes中Pod和Deployment的关系？

容器与虚拟机的性能差异来源？

如何通过cgroups限制容器资源使用？

Docker 的核心组件有哪些？它们的作用是什么？

* Docker Engine：运行容器的引擎
* Docker Images：容器运行所需的文件和依赖
* Docker Containers：运行的容器实例
* Docker Hub：公共镜像仓库

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

Docker 的数据卷是什么？如何使用？

Docker数据卷是持久化存储的数据，用于在容器之间共享数据。

创建：docker volume create my\_volume

挂载：docker run -v my\_volume:/data my\_container

容器化技术

容器与虚拟机的区别是什么？

容器是轻量级的，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。

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

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

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/交付）：将通过测试的代码自动部署到生产环境

DevOps 常用工具有哪些？

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

什么是基础设施即代码（IaC）？

使用代码（如 YAML、Terraform）定义和管理基础设施

优点：自动化、可重复性、可版本化