Java集合

【2/3】HashMap底层结构是什么？如何解决哈希冲突？

简要总结

**HashMap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。

**解决哈希冲突的机制：**

1. 拉链法
2. 红黑树优化
3. 二次扰动函数
4. 动态扩容机制

负载因子过高会增加哈希冲突，过低则导致频繁扩容

详细解释

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

二次扰动函数：

作用：混合高位低位、优化索引分布、减少哈希冲突  
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。

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。负载因子过高会增加哈希冲突，过低则导致频繁扩容

【4/5】详细描述HashMap的扩容机制（触发条件、rehash过程）。

简要总结

**触发条件：**

情况1：元素数量超过阈值

情况2：链表长度≥8但数组容量＜64

**Rehash过程：**

1. 创建新数组​
2. ​重新计算索引：e.hash & (newCap - 1)
3. 链表拆分、红黑树拆分/退化

详细解释

​

【1/3】为什么HashMap线程不安全？举例说明并发问题场景。

简要总结

**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. Collections.synchronizedMap
2. ConcurrentHashMap

详细解释

**HashMap线程不安全的原因**

1. **多线程同时操作导致数据覆盖**

多个线程同时操作时，可能会导致数据覆盖或丢失。

1. **扩容头插形成环形链表引发死循环**

HashMap的扩容是通过头插法将链表节点迁移到新数组中的。

多个线程同时操作链表可能导致链表形成环形结构引发死循环。

1. **多线程同时树化导致红黑树结构异常**

链表转红黑树，多个线程同时进行树化操作，可能会导致红黑树结构异常。

**解决方案：**

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

【1/3】为什么Java 8将链表转红黑树的阈值设为8？

简要总结

根据泊松分布，哈希冲突时，链表长度达到8的概率极低

链表长度≤8时，链表性能足够且内存占用低

红黑树退化为链表的阈值为6，避免787频繁转换

【1/3】HashMap的负载因子为什么默认是0.75？

简要总结

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

基于泊松分布，HashMap的查找性能接近O(1)。

【1/2】HashMap的遍历方式有哪几种？哪种效率更高？

简要总结

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，效率与entrySet()相同。
3. ​keySet()和values()适用于特定场景，效率较低。

Java并发

【1/4】ConcurrentHashMap如何实现线程安全？JDK7和JDK8的区别？

简要总结

​**JDK7**：通过分段锁实现线程安全，锁粒度大，性能低

**JDK8**：通过 CAS + synchronized 实现线程安全，锁粒度小，性能高

【1/5】CAS的原理是什么？存在哪些缺陷？

简要总结

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。

详细解释

**CAS（Compare And Swap）**是一种 ​**无锁并发**​ 机制：

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）。

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**：结合锁机制或限制自旋次数。

1. ​**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。

【1/5】AQS的核心机制是什么？举一个基于AQS实现的工具类。

简要总结

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. ​状态管理
2. ​线程排队
3. ​模板方法
4. ​独占与共享模式

**ReentrantLock**是AQS典型实现。

详细解释

**AQS 的核心机制**：

1. ​状态管理：

* 通过 volatile int state 表示资源状态，比如锁的重入次数。
* 提供 getState()、setState() 和 compareAndSetState() 方法操作状态。

1. ​线程排队：

* 使用双向链表实现的​CLH 队列​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。

1. ​模板方法：

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

1. ​独占与共享模式：

* 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，通过 AQS 的队列机制实现公平性。

【1/3】为什么ConcurrentHashMap的size()方法需要特殊处理？

简要总结

ConcurrentHashMap 的 size() 方法需要特殊处理，以解决 ​

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）
2. 使用LongAdder计算近似值（JDK8）​

详细解释

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。

1. ​使用LongAdder计算近似值（JDK8）​：

* 使用 LongAdder 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。
* 当你需要获取最终的值时，它会将 baseCount 和所有 CounterCell 的值相加，得到一个总和的近似值。

【1/4】自旋锁和互斥锁在CAS中的协同作用？

简要总结

1. ​**自旋锁**：低竞争或短任务时，线程通过自旋重试，避免上下文切换开销。
2. ​**互斥锁**：高竞争或长任务时，线程进入阻塞状态，释放 CPU 资源。
3. **协同机制**：
4. ​**CAS 尝试**：线程首先通过 CAS 尝试获取资源
5. ​**自旋等待**：若 CAS 失败，线程进入自旋状态，短暂重试多次。
6. ​**互斥阻塞**：若自旋仍失败，线程进入阻塞状态，等待唤醒。

【1/4】synchronized和ReentrantLock的性能对比？

简要总结

**synchronized**：简单，JVM 内置优化，适合低竞争。

**ReentrantLock**：强大，性能更优，适合高竞争。

详细解释

| **​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**​ | 非公平锁（默认） | 支持公平锁和非公平锁 |
| ​**功能扩展**​ | 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 |
| ​**适用场景**​ | 适合低竞争、简单锁场景 | 适合高竞争、复杂锁场景 |

【1/3】线程池中的Worker类是如何利用AQS的？

简要总结

Worker 类继承 AQS，利用独占锁，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。

详细解释

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。（0空闲，1占用）
2. ​**锁的获取与释放**：

* ​线程池从队列中获取任务后，通过 Worker 的 runWorker() 执行任务。
* lock()**获取锁**：在任务执行前调用，将 state 改为 1表示线程占用。
* ​unlock() **释放锁**：在任务执行后调用，将 state改为 0表示线程空闲。

1. ​**中断控制**：  
   通过 AQS 的 tryLock() 方法，支持对 Worker 的中断操作。

SpringBoot/SSM框架

Spring AOP的实现原理？JDK动态代理和CGLIB的区别？

简要总结

掌握/重点

记忆关键

详细解释

**Spring AOP**​ 通过动态代理实现，支持 JDK 动态代理和 CGLIB 两种方式。

**JDK 动态代理**​ 适用于目标类实现了接口的场景，性能较好但限制较多。

**CGLIB 代理**​ 适用于目标类未实现接口的场景，灵活性更高但性能略低。

**Spring AOP的实现原理**

Spring AOP（面向切面编程）通过动态代理实现，核心原理如下：

1. ​**代理模式**：Spring AOP 使用代理模式在运行时动态生成代理对象，代理对象拦截目标方法，执行切面逻辑。
2. ​**切面织入**：Spring 通过 ​**Pointcut**​ 定义切入点，**Advice**​ 定义增强逻辑，**Aspect**​ 将切面逻辑织入目标方法。
3. ​**代理类型**：Spring AOP 支持两种代理方式：
   * ​**JDK 动态代理**：基于接口，目标类必须实现接口。
   * ​**CGLIB 代理**：基于继承，目标类无需实现接口。

**JDK动态代理和CGLIB的区别**

| **​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 |
| ​**性能**​ | 调用速度较快，但创建代理对象较慢 | 创建代理对象较快，但调用速度稍慢 |
| ​**依赖**​ | 无需额外依赖 | 依赖 CGLIB 库 |
| ​**限制**​ | 无法代理未实现接口的类 | 无法代理 final 类或方法 |

**总结**

* ​**Spring AOP**​ 通过动态代理实现，支持 JDK 动态代理和 CGLIB 两种方式。
* ​**JDK 动态代理**​ 适用于目标类实现了接口的场景，性能较好但限制较多。
* ​**CGLIB 代理**​ 适用于目标类未实现接口的场景，灵活性更高但性能略低。

通过这种清晰、简洁的回答，面试官可以快速理解核心原理和区别，展现你的逻辑性和条理性。

Spring Bean的生命周期和作用域有哪些？

简要总结

掌握/重点

记忆关键

详细解释

​生命周期：实例化 → 属性赋值 → 初始化 → 使用 → 销毁。

​作用域：Singleton、Prototype、Request、Session、Application、WebSocket。

**Spring Bean的生命周期**

Spring Bean的生命周期包括以下关键阶段：

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（DI）为Bean的属性赋值。
3. ​**初始化**：
   * ​**BeanPostProcessor前置处理**：调用postProcessBeforeInitialization方法。
   * ​**初始化方法**：执行@PostConstruct注解方法、InitializingBean的afterPropertiesSet方法或自定义init-method。
   * ​**BeanPostProcessor后置处理**：调用postProcessAfterInitialization方法。
4. ​**使用**：Bean在应用上下文中被使用。
5. ​**销毁**：
   * ​**销毁方法**：执行@PreDestroy注解方法、DisposableBean的destroy方法或自定义destroy-method。

**Spring Bean的作用域**

Spring Bean的作用域定义了Bean的创建和生命周期范围，常见作用域包括：

| **​作用域​** | **​描述​** |
| --- | --- |
| ​**Singleton**​ | 默认作用域，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，仅适用于Web应用。 |
| ​**Session**​ | 每个HTTP会话创建一个Bean实例，仅适用于Web应用。 |
| ​**Application**​ | 每个ServletContext生命周期内创建一个Bean实例，仅适用于Web应用。 |
| ​**WebSocket**​ | 每个WebSocket会话创建一个Bean实例，仅适用于Web应用。 |

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

简要总结

掌握/重点

记忆关键

详细解释

* ​**核心注解**：@EnableAutoConfiguration。
* ​**条件化配置**：通过@Conditional注解实现按需加载。
* ​**自动配置类**：通过META-INF/spring.factories定义，结合@Configuration注册Bean。
* ​**属性绑定**：通过@ConfigurationProperties实现配置文件与Bean的绑定。

**Spring Boot自动配置的实现机制（@EnableAutoConfiguration）**

Spring Boot的自动配置通过@EnableAutoConfiguration实现，核心机制如下：

1. ​**启动类注解**：
   * @SpringBootApplication包含@EnableAutoConfiguration，用于启用自动配置。
2. ​**条件化配置**：
   * ​\*\*@Conditional注解\*\*：根据条件（如类路径、配置文件、Bean是否存在）决定是否加载配置类。
   * 常见条件注解：@ConditionalOnClass、@ConditionalOnMissingBean、@ConditionalOnProperty等。
3. ​**自动配置类**：
   * ​**META-INF/spring.factories**：定义自动配置类的全限定名，Spring Boot启动时加载这些类。
   * 自动配置类通过@Configuration定义Bean，结合条件注解实现按需加载。
4. ​**配置属性绑定**：
   * 通过@ConfigurationProperties将配置文件（如application.properties）中的属性绑定到Bean。

**核心流程**

1. ​**启动扫描**：Spring Boot启动时扫描META-INF/spring.factories，加载自动配置类。
2. ​**条件检查**：根据条件注解判断是否加载配置类。
3. ​**Bean注册**：符合条件的配置类注册Bean到Spring容器。
4. ​**属性绑定**：将配置文件中的属性注入Bean。

如何解决循环依赖问题？三级缓存机制详解。

简要总结

掌握/重点

记忆关键

详细解释

* ​三级缓存：一级缓存（完全初始化Bean）、二级缓存（未完全初始化Bean）、三级缓存（Bean工厂）。
* ​解决流程：实例化 → 属性注入（通过三级缓存获取依赖） → 初始化 → 完成缓存迁移。
* ​适用场景：仅适用于单例Bean的循环依赖，原型Bean不支持。

**如何解决循环依赖问题？三级缓存机制详解**

**1. ​循环依赖问题**

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。

**2. ​Spring的解决方案：三级缓存机制**

Spring通过**三级缓存**解决循环依赖问题，核心思想是提前暴露未完全初始化的Bean。

**三级缓存机制详解**

Spring的三级缓存分为以下三个层次：

1. ​**一级缓存（Singleton Objects）​**：
   * 存储完全初始化后的单例Bean。
   * 通过singletonObjects Map实现。
2. ​**二级缓存（Early Singleton Objects）​**：
   * 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）。
   * 通过earlySingletonObjects Map实现。
3. ​**三级缓存（Singleton Factories）​**：
   * 存储Bean的工厂对象（ObjectFactory），用于生成提前暴露的Bean。
   * 通过singletonFactories Map实现。

**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。
2. ​**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。
3. ​**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。

Spring事务失效的常见场景有哪些？

简要总结

掌握/重点

记忆关键

详细解释

非Public方法、异常类型不匹配、异常被捕获、非代理调用、传播行为错误、数据源未配置事务管理器、多线程调用、非Spring管理Bean。

1. ​**非Public方法**：
   * @Transactional注解只能作用于Public方法，Private、Protected或默认修饰符的方法不会生效。
2. ​**异常类型不匹配**：
   * 默认只回滚RuntimeException和Error，如果抛出其他异常（如IOException）且未配置rollbackFor，事务不会回滚。
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，事务不会回滚。
4. ​**非代理调用**：
   * 同一个类中，方法A调用方法B（B有@Transactional），由于代理机制，B的事务不会生效。
5. ​**事务传播行为配置错误**：
   * 例如，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。

MyBatis的#{}和${}的区别及SQL注入问题。

简要总结

掌握/重点

记忆关键

详细解释

* \*\*#{}\*\*：安全、预编译，适用于参数值替换。
* \*\*`${}``\*\*：直接拼接，存在SQL注入风险，适用于SQL片段替换。
* **最佳实践**：优先使用#{}，仅在必要时（如动态表名、排序字段）使用${}，并严格校验输入

**MyBatis的#{}和${}的区别及SQL注入问题**

**1. ​\*\*#{} 和 ${} 的区别\*\***

| **​特性​** | **​\*\*#{}\*\*​** | **​\*\*${}\*\*​** |
| --- | --- | --- |
| ​**处理方式**​ | 预编译处理，使用PreparedStatement | 直接替换，使用Statement |
| ​**安全性**​ | 防止SQL注入，安全 | 存在SQL注入风险，不安全 |
| ​**使用场景**​ | 参数值替换（如WHERE id = #{id}） | SQL片段替换（如ORDER BY ${column}） |
| ​**数据类型处理**​ | 自动处理类型转换 | 直接拼接字符串，需手动处理类型 |

**2. ​SQL注入问题**

* ​\*\*#{}\*\*：通过预编译机制，将参数值作为占位符传入，防止SQL注入。
* ​\*\*${}\*\*：直接拼接SQL字符串，如果用户输入恶意内容（如1 OR 1=1），会导致SQL注入。

SpringMVC请求处理流程中的HandlerAdapter作用？

简要总结

掌握/重点

记忆关键

详细解释

HandlerAdapter在Spring MVC中的作用是：

* ​**适配处理器**：根据处理器类型选择合适的适配器。
* ​**执行业务逻辑**：调用处理器的处理方法。
* ​**处理返回值**：将结果转换为ModelAndView。

**Spring MVC请求处理流程中的HandlerAdapter作用**

**1. ​核心作用**

HandlerAdapter是Spring MVC框架中的核心组件，负责**适配并执行处理器（Handler）​**，将请求映射到具体的处理方法。

**2. ​主要职责**

* ​**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。
* ​**调用处理方法**：通过handle()方法执行处理器的业务逻辑。
* ​**处理返回值**：将处理器返回的结果（如ModelAndView）转换为ModelAndView对象，供视图解析器使用。

**3. ​常见实现**

* ​\*\*RequestMappingHandlerAdapter\*\*：用于处理@Controller和@RequestMapping注解的处理器。
* ​\*\*HttpRequestHandlerAdapter\*\*：用于处理HttpRequestHandler接口的实现类。
* ​\*\*SimpleControllerHandlerAdapter\*\*：用于处理Controller接口的实现类。

SpringBoot如何集成Tomcat容器？

简要总结

掌握/重点

记忆关键

详细解释

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

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

【1/4】Nginx 的反向代理是如何工作的？

简要总结

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？

简要总结

* ​**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）？

简要总结

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？

简要总结

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

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？

【0/1】Docker 的核心组件有哪些？它们的作用是什么？

简要总结

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？

简要总结

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？

简要总结

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？

简要总结

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？

简要总结

* 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？

简要总结

* 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？

简要总结

* 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）？

简要总结

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。