一、Java 核心基础

1. **Java 17 与 Java 8 相比，有哪些重要新特性?**
   答：密封类（Sealed Classes）限制继承范围；Record 类简化不可变数据载体；switch 表达式增强（支持模式匹配）；ZGC/Shenandoah 等低延迟垃圾收集器成熟；模块化系统（Module）进一步完善；文本块（Text Blocks）简化多行字符串处理。
2. **什么是值类型（Value Types）？为什么它解决了 Java 的什么问题?**
   答：Java 16 + 实验性特性，用于定义无标识、不可变的轻量级数据类型（如`int`的对象版），解决传统对象因引用带来的内存开销和 GC 压力，提升高频访问数据的性能。
3. **CompletableFuture 与传统线程池相比，优势在哪里?**
   答：支持链式异步操作（`thenApply`/`thenCombine`），简化多任务依赖逻辑；内置异常处理（`exceptionally`）；可组合合多个异步结果，避免手动线程同步，更适合复杂异步场景。
4. **JVM 内存模型（JMM）如何保证可见性、原子性和有序性?**
   答：可见性通过`volatileatile`（强制内存刷新）和`synchronized`（加锁时刷新）保证；原子性依赖`synchronized`或 CAS 操作；有序性通过`volatile`（禁止指令重排）和`happens-before`规则约束。

二、集合与数据结构

1. **ArrayList 扩容机制在 Java 17 中是否有优化?**
   答：基本逻辑不变（默认初始容量 10，扩容为原容量 1.5 倍），但内部实现引入`Arrays.copyOf`的优化版本，减少极端情况下的内存拷贝开销，并发场景仍需手动加锁或使用`CopyOnWriteArrayList`。
2. **HashMap 与 ConcurrentHashMap 的实现原理及线程安全差异?**
   答：HashMap 基于数组 + 链表 / 红黑树，非线程安全（扩容时可能出现环）；ConcurrentHashMap 在 JDK 8 + 用 CAS+synchronized 实现分段锁，支持高并发读写，`size()`操作通过累加段内计数实现。
3. **LinkedHashMap 如何实现 LRU 缓存?**
   答：通过`accessOrder=true`启用访问顺序维护，每次 get/put 操作将节点移至链表尾部；重写`removeEldestEntry`方法定义淘汰规则（如超过容量时移除头部节点）。

三、框架与生态

1. **Spring Boot 3.x 的核心改进?**
   答：基于 Jakarta EE 9+（包名从`javax`迁移至`jakarta`）；默认支持 GraalVM 原生镜像（AOT 编译，启动速度提升 10 倍 +）；简化 Observability（可观测性）集成；依赖管理更灵活。
2. **Spring Cloud 与 Kubernetes 的融合趋势?**
   答：Spring Cloud Kubernetes 逐渐替代传统 Netflix 组件，通过服务发现（K8s Service）、配置中心（ConfigMap/Secret）、负载均衡（K8s 原生）实现云原生部署，减少中间件依赖。
3. **MyBatis-Plus 相比 MyBatis 的核心优势?**
   答：内置 CRUD 通用接口（`BaseMapper`），减少重复 SQL；支持 Lambda 表达式构建查询条件，避免字符串拼写错误；分页插件、乐观锁等功能开箱即用，简化开发。
4. **为什么说 Quarkus 是“为容器而生”的 Java 框架?**
   答：基于 GraalVM AOT 编译，启动快、内存占用低；采用“即时启动”模式，适合 Serverless 场景；与容器化工具（Docker/K8s）深度集成，优化容器镜像大小。

四、性能优化与并发

1. **JVM 调优中，如何平衡吞吐量与延迟?**
   答：吞吐量优先选 Parallel GC（多线程回收，停顿较长）；延迟优先选 ZGC/Shenandoah（亚毫秒级停顿，CPU 开销较高）；通过`-XX:MaxGCPauseMillis`设置目标停顿时间，JVM 自动调整策略。
2. **线程池参数如何合理配置?**
   答：CPU 密集型任务（如计算）：核心线程数 = CPU 核心数 ±1；IO 密集型任务（如网络请求）：核心线程数 = CPU 核心数 ×2；结合任务队列长度（避免 OOM）和拒绝策略（如`CallerRunsPolicy`）。
3. **什么是虚拟线程（Virtual Threads）？它解决了什么问题?**
   答：Java 19 + 正式特性，轻量级线程（由 JVM 管理，非 OS 线程），创建成本低（百万级线程无压力），解决传统线程因 OS 调度开销导致的高并发瓶颈，适合 IO 密集型场景。
4. **如何诊断 JVM 内存泄漏?**
   答：通过`jmap` dump 堆内存，结合 MAT 工具分析对象引用链；使用`jstack`查看线程状态，排查长期持有的对象（如静态集合未清理、连接池未释放）；监控`Old Gen`内存增长趋势。

五、分布式与微服务

1. **分布式事务的主流解决方案及适用场景?**
   答：2PC（强一致性，性能差，适合金融核心场景）；TCC（补偿事务，侵入业务代码，适合高并发）；SAGA（长事务拆分，最终一致性，适合订单流程）；本地消息表（低耦合，适合非核心业务）。
2. **微服务链路追踪的实现原理?**
   答：通过 TraceID 标识全局请求，SpanID 标识服务间调用；请求入口生成 TraceID，通过 HTTP 头 / RPC 元数据传递；各服务记录调用耗时并上报至追踪系统（如 SkyWalking），最终聚合为调用链路。
3. **Redis 缓存穿透、击穿、雪崩的区别及解决方案?**
   答：穿透（查询不存在的数据）：布隆过滤器拦截；击穿（热点 key 失效瞬间高并发）：互斥锁或永不过期；雪崩（大量 key 同时失效）：过期时间加随机值，多级缓存，熔断降级。

六、设计模式与架构

1. **DDD（领域驱动设计）在微服务中的实践?**
   答：按业务领域划分限界上下文（Bounded Context），每个上下文对应独立微服务；通过聚合根（Aggregate Root）管理领域对象；领域事件（Domain Event）实现跨上下文通信，避免紧耦合。
2. **如何用观察者模式实现事件驱动架构?**
   答：定义事件发布者（Subject）和订阅者（Observer）接口；发布者维护订阅者列表，事件发生时触发所有订阅者的回调方法；结合消息队列（如 Kafka）可实现跨服务的异步事件通知。
3. **单例模式的线程安全实现方式?**
   答：饿汉式（类加载时初始化，简单但可能浪费资源）；懒汉式 + 双重校验锁（`volatile`防止指令重排）；静态内部类（延迟加载且线程安全，推荐）。

七、JVM 深度剖析（20K 岗位必问）

1. **G1 和 ZGC 的垃圾回收原理及适用场景?**
   答：G1 采用 “区域化分代式”，将堆分为多个 Region，通过 Remembered Set 追踪跨 Region 引用，兼顾吞吐量和延迟，适合堆内存较大（4GB+）的应用；ZGC 是低延迟收集器，利用 Colored Pointer 和 Load Barrier 实现并发整理，停顿时间控制在 10ms 内，适合对延迟敏感的场景（如金融交易）。
2. **类加载的双亲委派模型被打破的场景有哪些?**
   答：① Tomcat 为隔离 Web 应用，自定义类加载器优先加载自身目录下的类；② OSGi 允许

Equinox 允许平级类加载器，允许模块自行决定类加载顺序；③ SPI 机制（如 JDBC）中，Bootstrap ClassLoader 委托父类加载器加载第三方实现类。

3. **JVM 内存泄漏中的 “永久代” 和 “元空间” 的区别?**
   答：永久代存在于堆内存，大小固定（易 OOM），存储类元信息、常量池等；元空间存在于本地内存，大小动态扩展，默认不限制（受物理内存限制），避免了永久代 OOM 问题（Java 8 + 取代永久代）。
4. **如何通过 JVM 参数优化 Full GC 频率?**
   答：① 增大新生代内存（`-Xmn`），减少对象进入老年代；② 调整老年代代入队阈值（`-XX:MaxTenuringThreshold`），延长对象在新生代存活时间；③ 选用 CMS/G1 等并发收集器，降低 Full 停顿；④ 避免大对象直接进入老年代（`-XX:PretenureSizeThreshold`）。

八、并发编程进阶

1. **synchronized 的锁升级过程（偏向锁→轻量级锁→重量级锁）?**
   答：① 偏向锁：无竞争时，线程获取锁后在对象头标记线程 ID，避免 CAS 操作；② 轻量级锁：多线程交替执行时，通过 CAS 竞争锁，失败时自旋重试；③ 重量级锁：自旋失败后升级为 OS 互斥锁，线程进入阻塞态，适用于长耗时操作。
2. **ThreadLocal 的实现原理及内存泄漏问题?**
   答：底层通过 Thread 的 ThreadLocalMap 存储键值对（key 为 ThreadLocal 实例，value 为数据），实现线程隔离；内存泄漏原因：ThreadLocalMap 的 Entry 是弱引用（key），若 ThreadLocal 实例被回收，value 仍可能被线程引用（线程未销毁时），解决方案：使用后调用 `remove()`。
3. **Java 9 的 Flow API 与 RxJava 的异同?**
   答：均基于响应式编程模型（Publisher-Subscriber）；Flow 是 JDK 原生 API，轻量但功能简单；RxJava 提供更丰富的操作符（如 map/flatMap）、线程调度和背压策略，适合复杂异步场景。
4. **线程池的拒绝策略在高并发场景下如何选择?**
   答：① `AbortPolicy`（默认）：直接抛异常，适合不允许丢失任务的场景；② `CallerRunsPolicy`：让提交任务的线程执行任务，缓解流量峰值（自我调节）；③ `DiscardOldestPolicy`：丢弃队列中最旧任务，适合实时性要求高的场景（如日志收集）。

九、Spring 全家桶深度

1. **Spring IoC 容器初始化的核心流程?**
   答：① 资源定位（读取 XML / 注解配置）；② BeanDefinition 加载（解析配置为 BeanDefinition 对象）；③ BeanDefinition 注册（存入 DefaultListableBeanFactory 的 Map 中）；④ BeanFactory 初始化（BeanFactoryPostProcessor 处理）；⑤ Bean 实例化（依赖注入、初始化方法调用）。
2. **Spring 事务传播机制中，REQUIRES_NEW 与 NESTED 的区别?**
   答：REQUIRES_NEW：创建新事务，若当前存在事务则暂停，两个事务独立提交 / 回滚；NESTED：在当前事务中创建子事务（savepoint），子事务回滚不影响父事务，父事务回滚会连带子事务回滚。
3. **Spring AOP 的切面执行顺序如何控制?**
   答：① 同一切面内：按通知定义顺序（@Before→@Around→@After→@AfterReturning/@AfterThrowing）；② 不同切面：通过 `@Order` 注解指定优先级（值越小越先执行），或实现 Ordered 接口。
4. **Spring Boot 的 Starter 机制原理?**
   答：Starter 是依赖描述符，通过 `spring.factories` 文件配置 AutoConfiguration 类；Spring Boot 启动时扫描 classpath 下的 AutoConfiguration，根据条件注解（@Conditional）自动配置 Bean，简化依赖管理（如 `spring-boot-starter-web` 自动引入 Tomcat、Spring MVC）。

十、数据库进阶

1. **MySQL 的 InnoDB 引擎中，聚簇索引与非聚簇索引的区别?**
   答：聚簇索引（主键索引）：叶子节点存储完整数据行，一张表仅一个，查询效率高；非聚簇索引（二级索引）：叶子节点存储主键值，需回表查询完整数据（覆盖索引可避免回表）。
2. **MySQL 的 MVCC（多版本并发控制）实现原理?**
   答：通过 undo log 保存数据历史版本，事务可见性由 Read View（活跃事务列表）判断：① 事务开始时生成 Read View；② 数据行的隐藏列（trx_id、roll_ptr）记录最后修改的事务 ID 和 undo log 指针；③ 对比数据行 trx_id 与 Read View，决定是否可见。
3. **分库分表后，如何解决分布式事务和跨表查询问题?**
   答：分布式事务：采用 TCC 或 SAGA 模式（避免 2PC 的性能问题）；跨表查询：① 按业务拆分避免跨表；② 全局表（字典表冗余到各库）；③ 借助中间件（Sharding-JDBC 的广播表、绑定表）；④ 搜索引擎（Elasticsearch）聚合结果。
4. **MySQL 的慢查询优化步骤?**
   答：① 开启慢查询日志（`slow_query_log=1`），设置阈值（`long_query_time=1`）；② 通过`explain`分析 SQL 执行计划，查看是否使用索引、rows 扫描行数；③ 优化索引（添加缺失索引、删除冗余索引）；④ 改写 SQL（避免`select *`、子查询改 join）；⑤ 分表分库或读写分离。

十一、中间件实战

1. **Redis 的 Cluster 集群如何实现数据分片与故障转移?**
   答：数据分片：采用哈希槽（16384 个），每个节点负责部分槽，`CRC16(key) % 16384`计算槽位；故障转移：① 哨兵（Sentinel）监测主节点心跳；② 主节点宕机后，哨兵选举新主节点（从节点晋升）；③ 重新分配槽位，更新集群配置。
2. **Kafka 的高吞吐量原理?**
   答：① 顺序写磁盘（避免随机 IO）；② 分区并行处理（多分区同时读写）；③ 批量发送与拉取（减少网络请求）；④ 零拷贝（`sendfile`系统调用，数据直接从磁盘到网络，跳过用户态）。
3. **RabbitMQ 如何保证消息不丢失?**
   答：① 生产者：开启确认机制（`publisher-confirms`），失败重试；② Broker：队列持久化（durable=true）+ 消息持久化（delivery_mode=2）；③ 消费者：关闭自动 ACK（autoAck=false），处理完成后手动 ACK。
4. **Elasticsearch 的倒排索引原理及优化方案?**
   答：倒排索引：将文档拆分为词条（Term），建立 “词条→文档 ID 列表” 的映射，支持快速全文检索；优化：① 合理设置分片数（避免过多分片导致资源浪费）；② 调整刷新间隔（`refresh_interval`，平衡实时性与性能）；③ 使用索引别名实现零停机升级。

十二、架构设计与性能优化

1. **微服务架构下的服务降级与熔断策略?**
   答：降级：非核心服务故障时，返回默认值（如缓存数据），保障核心流程可用（如电商下单时，推荐服务降级）；熔断：当服务错误率超过阈值，暂时停止调用，避免级联失败（如 Sentinel 的 RT / 异常比例阈值），熔断后通过半开状态试探恢复。
2. **如何设计一个高并发的秒杀系统?**
   答：① 前端限流（按钮置灰、验证码）；② 接口层：Nginx 限流、网关过滤无效请求；③ 应用层：Redis 预减库存、消息队列异步下单；④ 数据层：分库分表、热点数据缓存、库存行锁；⑤ 兜底：库存不足时快速失败。
3. **DDD 中聚合根与限界上下文的设计原则?**
   答：聚合根：① 作为聚合的入口，统一管理聚合内对象；② 维护聚合内数据一致性；③ 仅通过聚合根与外部交互（如订单聚合根包含订单项，外部只能通过订单操作订单项）；限界上下文：① 按业务领域划分，内部高内聚；② 通过上下文映射（Context Mapping）定义跨域交互规则（如共享内核、防腐层）。

4. **Java 应用的性能瓶颈分析工具链?**
   答：① 监控：Prometheus+Grafana（指标监控）、SkyWalking（链路追踪）；② 诊断：Arthas（在线诊断，查看线程、内存、方法耗时）、JProfiler（性能剖析）；③ 日志：ELK（日志收集分析）；④ 压测：JMeter/Gatling（模拟高并发场景）。

复习建议：

1. **源码优先**：Spring、JDK 并发包（`ConcurrentHashMap`、`ThreadPoolExecutor`）、MyBatis 的核心源码至少过 1-2 遍，理解底层设计。
2. **实战结合**：针对分布式事务、缓存问题等，动手搭建 Demo 验证（如用 Seata 实现 TCC，用 Redis 模拟缓存穿透）。
3. **场景化表达**：回答时结合项目经历（如 “我们在秒杀系统中用 Redis+RabbitMQ 解决了 XX 问题，峰值 TPS 达到 XX”）。