我现在有1万个问答请求，但服务器只能同时容纳1000个请求，你怎么做

1. 限流（Rate Limiting）
   防止服务器过载，确保核心服务稳定
   使用 令牌桶（Token Bucket） 或 漏桶（Leaky Bucket） 算法，限制 QPS（每秒请求数）。
   如果超出并发上限, 返回 429（请求过多），让客户端稍后重试。
   可以使用 Nginx、Traefik 或 API Gateway 来做限流。
2. 请求排队（Message Queue）
   避免请求直接被拒绝，保证请求最终被处理
   使用 RabbitMQ / Kafka / Redis 等消息队列，将请求存入队列，后端逐步处理。
   服务器空闲时从队列中拉取请求，避免因短时间内流量过大导致崩溃。
3. 降级处理（Graceful Degradation）
   保证核心功能可用，非核心功能暂时禁用
   高峰期时，优先处理重要请求，低优先级请求返回默认值或缓存结果。
   可缓存部分响应，减少数据库和计算压力，如 Redis / CDN 预缓存常用数据。
4. 负载均衡（Load Balancing）
   扩展服务器能力，提高并发吞吐量
   部署多个服务器实例，使用 Nginx / Traefik / Kubernetes 进行负载均衡。
   将 1 台服务器的 1000 并发扩展到多台服务器，如 10 台服务器可支持 1 万并发。

传入后端的问答数据，有不同的格式，如何根据格式进行分类存储，并且要保证数据的有序性。

格式识别：判断 JSON / XML / 纯文本
解析转换：XML转换为json或者统一转换为QA结构体
分类存储：存入 MySQL / Redis / MongoDB/ 消息队列
保证顺序：使用时间戳、ID进行排序

RabbitMQ对比Kafka

消息模型 队列模型（点对点、发布/订阅） 日志模型（消息分区、顺序消费）
吞吐量 中等（万级 QPS） 极高（百万级 QPS）
消息顺序 支持严格顺序 分区内有序，全局无序
消息持久化 基于磁盘存储，但会定期删除消费过的消息 日志式存储，消息保留一定时间（默认 7 天）
可靠性 支持消息确认机制，保证投递成功 高吞吐但可能丢消息（可配合 acks=all 方案提高可靠性）
事务支持 支持事务消息 默认不支持事务（只能用幂等性方式处理）
消息延迟 低（ms 级），适合实时性要求高的场景 默认批量处理，延迟较高（秒级）

扩展性 扩展性一般，队列存在性能瓶颈 天然分布式，横向扩展性强
应用场景 金融、订单系统、事务处理、低延迟场景 日志处理、流式计算、实时数据分析、大数据

哈希索引

- ✅ Memory 引擎 默认使用哈希索引
- ✅ InnoDB 主要使用 B+ 树索引，但有 自适应哈希索引（AHI）（=、in才触发）
- ✅ 哈希索引适用于 等值查询，但不支持 范围查询
- ✅ AHI 由 InnoDB 自动管理，如果影响性能可以手动关闭

秒杀库存超卖

1. Redis 分布式锁 + Lua
   秒杀开始前，将商品库存存入 Redis。
   购买时，先用 Redis SETNX 加锁，确保同一时间只有一个线程在修改库存。
   用 Lua 脚本原子性地判断库存并减少库存，避免并发问题。
2. 消息队列（MQ）
   用户请求先写入消息队列，队列按顺序消费。
   后台任务逐步减少库存，避免数据库压力过大。

IO多路复用

- 监听多个文件描述符（FD）是否就绪（如 socket 是否可读写）

1. select：
   早期的多路复用模型。
   使用数组或位图管理文件描述符（fd），有最大数量限制（默认1024）。
   每次调用都需要遍历所有的fd，性能较差，复杂度为 O(N)。
2. poll：
   改进版的select，使用链表管理fd，理论上没有数量限制。
   和select类似，仍然是遍历所有的fd，复杂度为 O(N)。
3. epoll：
   Linux特有的高效IO多路复用模型。
   使用事件驱动机制，注册fd后，内核只返回活跃的fd，避免了遍历所有fd的开销，复杂度为 O(1)。

- Redis就是基于 epoll 的单线程 Reactor

MYSQL三大日志

日志名称	日志类型	作用范围	逻辑 or 物理	主要用途
undo log	回滚日志	InnoDB（存储引擎层）	逻辑日志	回滚事务、支持 MVCC（快照读）
redo log	重做日志	InnoDB（存储引擎层）	物理日志	崩溃恢复（Crash Recovery），保证持久性
binlog	二进制日志	MySQL Server 层	逻辑日志	主从复制、增量备份、数据恢复

SQL语句执行的流程

1. 查询语句

- 先检查该语句是否有权限，如果没有权限，直接返回错误信息，如果有权限，在 MySQL8.0 版本以前，会先查询缓存，以这条 SQL 语句为 key 在内存中查询是否有结果，如果有直接缓存，如果没有，执行下一步。
- 通过分析器进行词法分析，提取 SQL 语句的关键元素，比如提取上面这个语句是查询 select，提取需要查询的表名为 tb_student，需要查询所有的列，查询条件是这个表的 id='1'。然后判断这个 SQL 语句是否有语法错误，比如关键词是否正确等等，如果检查没问题就执行下一步。
- 接下来就是优化器进行确定执行方案，上面的 SQL 语句，可以有两种执行方案：a.先查询学生表中姓名为“张三”的学生，然后判断是否年龄是 18。b.先找出学生中年龄 18 岁的学生，然后再查询姓名为“张三”的学生。那么优化器根据自己的优化算法进行选择执行效率最好的一个方案（优化器认为，有时候不一定最好）。那么确认了执行计划后就准备开始执行了。

2. 更新语句

- 先查询到张三这一条数据，不会走查询缓存，因为更新语句会导致与该表相关的查询缓存失效。
- 然后拿到查询的语句，把 age 改为 19，然后调用引擎 API 接口，写入这一行数据，InnoDB 引擎把数据保存在内存中，同时记录 redo log，此时 redo log 进入 prepare 状态，然后告诉执行器，执行完成了，随时可以提交。
- 执行器收到通知后记录 binlog，然后调用引擎接口，提交 redo log 为提交状态。
- 更新完成。

ACID分别实现的手段是什么

A：undo log C：AID I:锁和MVCC D:redo log

GMP的P的本地队列没有的话，work-stealing机制

首先他会从全局队列拿G，如果全局队列也为空的话，就从其他P中取它本地的一半的G到自己的本地队列里面

Golang执行的时候到底发生了什么

1. import -> const -> 变量 -> init() -> main()
2. 汇编入口 -> runtime.rt0_go() -> runtime.schedule/init + 其它的初始化 -> runtime.main() -> 自己定义的main()
   runtime.rt0_go()到runtime.main()之间发生的事情：

- 设置 G（Goroutine）、M（Machine，表示线程）、P（Processor，代表可执行资源）三者关系
- 初始化调度器（M 与 P 的绑定）
- 设置栈、堆内存
- 初始化垃圾回收机制
- 初始化系统线程、信号处理等等

Golang什么时候会产生死锁

只要用了sync包和channel就有可能产生死锁（除了多个G竞争资源，也有可能是主G阻塞导致所有的G都阻塞，这就是死锁）

- 同步原语中的很多数据结构使用后进行拷贝
- waitgroup中Add()和Done()不匹配
- Mutex和RWMutex加锁解锁不一致或者两个G竞争两个锁
- channel消费者和生产者都要有

分布式一致性

强一致性、弱一致性、最终一致性
强一致性：（RAFT->ETCD）三个节点，一个节点接受数据向其他节点发送数据，能得到一半以上的节点返回响应，数据更新才算完成。这种情况，任何时刻所有节点上的数据都是一致的。

- 为什么半数以上就够了：因为就算某些节点当前缺失，它稍后一定会同步过来
  弱一致性：（redis集群）第一个节点接收到数据之后，立即返回相应，然后再向其他节点同步数据。这种情况，有可能某些时候各个节点上面的数据是不一致的。

跳表和b+树

- 跳表是多级链表
- 性能相对于b+树，查询没有它快，但是插入删除比它快
- 实现起来比b+简单

手动加锁的map和sync.Map的区别

特性	手动加锁的 map	sync.Map
实现复杂度	需要自己管理锁	内置并发安全，无需手动锁管理
锁粒度	粗粒度（整个map一把锁）	细粒度（内部使用分段锁等优化）
内存占用	通常更低	通常更高（为并发优化牺牲空间）
性能特点	写多读少性能较好	读多写少性能极佳（读写分离）
API 复杂度	使用标准 map API + 锁	专用 API (Store/Load/Range 等)
键类型限制	必须可比较	接受任何类型 (`interface{}`)
零值可用性	需要 `make` 初始化	零值即可直接使用
锁的类型	悲观锁	乐观锁

gin为什么快

1. 路由匹配快 使用高性能的 前缀树（Radix Tree） 做路由匹配，查找速度非常快。
2. 零内存拷贝（Zero Allocation） 用 sync.Pool 重用上下文对象（gin.Context），大幅减少 GC 压力。
3. 编译时绑定（Handler注册阶段） 路由和 handler 关系在启动时就绑定好，避免运行时频繁计算。
4. 小巧的封装 代码简洁、模块少，避免了很多无谓的抽象。
5. 高效中间件设计 中间件链用数组存储，依次执行，无复杂跳转，性能好。

linux指令

1. 查看端口是否被占用 lsog -i :8080、netstat -tulnp | grep 8080、ss -tuln | grep 8080
2. 递归创建多层文件夹 mkdir -p a/b/c/d
3. 查看网络连接 netstat -an、ss -an

MYSQL架构

1. 连接层
2. 服务层（SQL层）：解析器、优化器、执行器
3. 存储引擎层：数据的存取
4. 存储系统层：日志文件、临时文件

docker使用

1. 进入正在运行的容器：docker exec、docker attach
2. build一个镜像：docker build -t <镜像名>:<标签> <Dockerfile所在路径>

文件上传失败怎么办？

1. 文件上传到业务服务器失败
   客户端传输失败：设置合理的 timeout 和重试机制（例如 axios、fetch 有重试插件）
   服务端处理异常：返回错误码 + 错误信息，客户端主动重试
   可以使用 上传任务记录表 或 Redis 暂存：业务服务宕机也不丢数据
2. 业务服务器上传 Ceph 失败
   使用 重试机制 + 异步任务队列（RabbitMQ）
   上传失败写入 MQ 的 “失败队列” 中，自动/手动重试
   Ceph写入成功但元数据写失败：使用 本地事务 + 分布式事务补偿 或最终一致性