Redis、Docker、计网面试

Redis

1、redis的常用数据类型有哪些

string、list、hash、set、zset
lpush命令用于将一个或多个值插入到列表的头部、lpop命令则用于从列表的头部弹出一个值

2、redis数据同步

1. 主从复制：主节点（master）将数据更改同步到一个或多个从节点（slave）。从节点会接收并应用主节点的变更，保持数据的一致性。（使用 `SYNC` 和 `PSYNC` 命令来处理数据同步过程）
2. Redis 集群：在 Redis 集群中，数据被分片存储在不同的节点上，并且主节点会将数据同步到其对应的从节点，确保高可用性和负载均衡。
3. RDB和AOF持久化：通过 RDB 快照或 AOF 日志文件保存数据状态。可以将这些文件用于数据恢复和同步。
4. 哨兵模式：哨兵模式是一种主从复制的扩展，引入了哨兵节点来监控主服务器的状态。当主服务器宕机时，哨兵会自动将从服务器中的一个提升为主服务器，保证系统的高可用性。

3、对数据一致性如何理解?

1. 主从一致性：主节点的数据变更要同步到所有从节点，从节点的状态应与主节点一致。
2. 数据持久性：确保即使在系统重启或崩溃后，数据也能恢复到一致的状态。这通过 RDB 快照和 AOF 日志来实现。
3. 强一致性与最终一致性：Redis作为分布式缓存，会存在数据的不一致，也就是Redis中的热点数据与关系型数据库的数据不一致，这种不一致性主要是针对写操作来说的。先操作数据库
   最终一致性，即数据会在一段时间后同步到所有从节点。
   强一致性，需要所有节点在任何时刻都保持相同的数据状态。

4、讲了redis和mysql的数据一致性

Redis作为分布式缓存，会存在数据的不一致，也就是Redis中的热点数据与关系型数据库的数据不一致，这种不一致性主要是针对写操作来说的。

1. 删除Redis缓存，再更新数据库：可能导致缓存穿透，增加数据库负载
   如果更新数据库失败，那么即使删除Redis成功，在重试期间，读的数据还是旧数据（即使重试成功，数据库更新了，但是Redis中被重新缓存了旧数据，导致两边的数据不一致）
   解决方案：延迟双删
   先删Redis的缓存，更新数据库中的数据，可以使用sleep方法让线程睡一会，然后再删除一个Redis的缓存
2. 先更新数据库，再删除Redis缓存：
   确保数据库数据是最新的，更新完数据库后还是会删除Redis中的缓存，最终缓存和数据库的数据是一致的，但是会有一些线程读到旧的数据
   解决方案：加锁+过期时间
   更新缓存前先加个分布式锁，确保同一时间只有一个更新缓存请求，同时给缓存加过期时间，这样即使出现缓存不一致的情况，缓存的数据也会很快过期

5、Redis中的数据会丢失么?

会，通过持久化机制来确保数据持久化和可靠性

6、redis有持久化机制吗，有几种？AOF文件会不会不停增加直到磁盘空间不足？如何解决?

三种：RDB快照、AOF日志、混合持久化同时使用 RDB 和 AOF
会

1. AOF 重写/压缩：使用 Redis 提供的 `BGREWRITEAOF` 命令手动触发或redis.conf设置 `auto-aof-rewrite-percentage` 和 `auto-aof-rewrite-min-size`参数来自动触发
   AOF文件重写，移除冗余命令，生成一个新的 AOF 文件，只包含当前数据的最小集合
2. 调整 AOF 同步策略：调整`appendfsync` 参数，选择不同的同步频率，减少不必要的磁盘IO操作，从而减缓AOF文件的增长速度
3. 定期清理和备份
4. 监控磁盘使用情况：使用监控工具和警报系统来跟踪磁盘空间的使用情况

7、讲讲AOF和RDB

1. RDB快照：定期将内存中的数据快照保存到磁盘上的 RDB 文件中
   原理：
   直接把内存中的数据保存到一个dump的文件中，通过 `redis.conf` 配置文件中的 `save` 选项来设置生成快照的频率，定时保存
   优缺点：
   文件小，恢复快，适合大规模数据恢复，但因为时周期性保持，会丢失最后一次快照后的修改数据，对数据恢复的完整性要求不高

2. AOF日志：将每个写操作追加到一个日志文件中，Redis 启动时会重放这个日志以恢复数据
   原理：
   每次写操作（如 `SET`、`DEL` 等）都会追加到 AOF 文件中，可以通过 `redis.conf` 中的 `appendfsync` 选项配置同步策略
   优缺点：
   所有的对Redis的服务器进行修改的命令都存到一个文件里，安全性高，AOF的文件体积比RDB大，需定期重写以控制文件大小，写操作对性能影响大

8、redis最新的混合持久化机制了解吗，详细介绍一下

同时使用 RDB 和 AOF 进行持久化：

1. 生成包含RDB快照和AOF日志补充的持久化文件：当开启混合持久化时，Redis在写入AOF文件时，会先将当前内存中的数据以RDB格式写入文件的开头，随后再将后续的命令以AOF格式追加到文件的末尾
2. 数据恢复过程：恢复时先加载RDB快照，然后应用AOF日志进行增量更新（由于AOF文件的前部分是以RDB格式存储的全量数据，Redis会先快速恢复这部分数据，然后再执行文件中后续的AOF命令，以实现增量数据的恢复）
   在Redis重启时，会优先检查AOF文件是否存在并开启
   优点：兼顾了性能和数据持久性
   可以在 RDB 提供快速恢复的同时，利用 AOF 提供更高的数据安全性
   会优先使用 AOF 文件来恢复数据，如果 AOF 文件不存在，才会使用 RDB 文件

9、为啥不能用redis做专门的持久化数据库存储

因为其设计初衷和特性更适合作为缓存或临时数据存储。具体原因包括：

- **内存限制**：Redis是内存数据库，所有数据存储在内存中，内存大小限制了其存储能力。
- **性能考虑**：虽然Redis支持持久化，但频繁的数据写入磁盘会引入I/O操作，降低性能。
- **数据一致性**：持久化过程中可能存在数据丢失的风险，影响数据一致性。
- **功能限制**：Redis的数据结构和查询功能相对简单，不适合处理复杂的数据关系和查询需求。

10、为什么Redis要比MySQL要快?

Redis通过内存存储、数据结构优化、单线程模型以及减少磁盘I/O操作，实现了比MySQL更快

- 内存存储：Redis将数据存储在内存中，而内存的读写速度远快于磁盘

- 数据结构优化：Redis使用高度优化的数据结构，如散列表，使得数据访问更加高效，尤其是在执行大量读写操作时。
- 单线程模型：Redis采用单线程模型处理命令，避免了多线程的线程切换和竞态条件，减少了复杂性和开销。同时，由于内存访问速度快，单线程模型并不会成为性能瓶颈。
- 无需频繁磁盘I/O：由于数据存储在内存中，Redis在正常操作中无需频繁进行磁盘I/O操作，从而减少了延迟。

11、redis为什么快

1. **内存存储**：Redis将数据存储在内存中，避免了磁盘I/O的延迟。
2. **单线程模型**：采用单线程处理所有请求，避免了多线程中的上下文切换和锁竞争。
3. **高效的数据结构**：提供高性能的数据结构，如字符串、哈希、列表、集合等，优化了各种操作。
4. **采用了非阻塞I/O多路复用机制**
5. **简化操作**：提供简单、高效的命令，减少了复杂的计算和数据处理。
6. **优化的网络协议**：使用高效的Redis协议，减少了通信开销和数据序列化时间。

12、redis数据迁移

RDB迁移：
Redis可以将内存中的数据快照保存到磁盘上的RDB文件中，然后将该文件复制到另一个Redis实例上进行恢复。这种方式适用于全量数据迁移
AOF迁移：
Redis还可以使用AOF（Append-Only File）持久化机制来迁移数据。将AOF文件复制到新的Redis实例上，并进行重放操作日志，可以将数据恢复到新的Redis实例中。这种方式适用于增量数据迁移
主从复制：
Redis支持主从复制机制，其中一个Redis实例作为主节点，其他实例作为从节点。通过配置主从关系并启动复制，数据可以从主节点同步到从节点。当数据迁移时，可以将新的Redis实例配置为从节点，使其复制主节点的数据
第三方工具

13、redis读写都很频繁的情况怎么办?

1. 读写分离，将读操作分配给从节点，写操作保留在主节点上。确保应用程序能够正确地分配读写请求，减少主节点负载
2. 配置多个从节点（副本）来提高读取性能和数据可用性
3. 优化持久化设置，平衡读写性能和持久化需求

14、redis什么情况下会变慢

1. 高负载：大量并发请求
2. 内存不足
3. 大型数据结构的复杂操作（不合理的数据结构）、阻塞命令
4. 持久化开销：aof频繁写入、长时间的rdb快照生成导致性能下降
5. 后台处理过多键过期
   解决方案：
   使用监控工具（如Redis Monitor、Redis Stats、Grafana等）实时追踪性能瓶颈。
   优化数据结构：选择适当的数据结构，避免复杂的操作和过大的数据集。
   调整配置：根据实际负载调整Redis的配置，如内存限制、持久化策略等。
   硬件升级：增加内存、提高网络带宽或使用更快的存储设备。

15、redis是单线程还是多线程

Redis 是 **单线程** 的。Redis使用单线程模型来处理客户端的请求，包括获取数据、解析请求、执行命令以及返回结果等，所有的命令和操作都是按顺序执行的，避免了多线程之间的竞争条件和锁开销，提高了访问共享数据的效率（Redis的主要操作，如键值对的读写，仍然是单线程的，为了保证原子性和不产生竞态条件）
但可以利用多个CPU核心来处理客户端的并发请求，通过在配置文件中设置`io-threads`参数可以开启多线程I/O处理（如持久化、异步删除、集群数据同步等）分散处理并发访问请求，提高处理效率。
如果应用场景需要更多的并发处理能力，可能需要考虑使用如Memcached、MemcacheDB或者其他分布式缓存解决方案。

16、非法不断地访问系统，每一次都会去mysql中进行匹配，使系统崩溃，该怎么办?

更接近缓存穿透

1. **限流与防护**：
   限流：对访问IP地址频率进行限制，防止恶意请求过多地占用系统资源。可以使用如Redis、Nginx或专用的API网关实现
   防护：当系统检测到异常访问或请求失败率达到一定阈值，将异常或恶意IP地址加入黑名单，阻止其访问
2. **缓存策略优化**：
   引入Redis等缓存：将常见查询结果缓存到Redis等内存数据库中，减少对MySQL的查询负载
3. **数据合法性校验**：增加数据合法性校验，对于明显不合法的请求直接拒绝，不转发到数据库层。
4. **日志分析**：
   分析日志：分析访问日志和错误日志，以识别异常请求模式和潜在的攻击。
   实时日志监控：利用日志管理工具进行实时日志分析和可视化。

5. **数据库优化**：
   查询优化：对MySQL查询进行优化，使用索引、减少复杂的查询操作。

17、缓存穿透、缓存击穿、缓存雪崩怎么办？布隆过滤器怎么实现？

1. **缓存穿透**：攻击者通过请求缓存和后端存储中不存在的数据，使得所有请求落到后端存储上，导致系统瘫痪
   布隆过滤器(数据结构，用于快速检索一个元素是否存在,足够大)/黑白名单/缓存预热/如果一个查询返回的数据为空（不管是数据不存在，还是系统故障），把这个空结果进行缓存，但它的过期时间会很短，最长不超过五分钟
2. **缓存击穿**：高并发地访问，热点数据失效后，大量请求同时涌入后端存储，导致后端存储负载增大
   永不过期/缓存预热/使用互斥锁或者分布式锁来对并发请求进行控制，避免对同一资源的并发读写竞争
3. **缓存雪崩**：缓存中大量的数据同时失效或过期，导致后续请求都落到后端存储上，从而引起系统负载暴增
   设置不同的过期时间
4. **布隆过滤器**：减少对数据库的直接访问，将所有可能存在的数据哈希到一个足够大的bitmap中，一个一定不存在的数据会被这个bitmap拦截掉，从而避免了对底层存储系统的查询压力

18、redis过期键的删除策略是什么？

键过期策略 处理键的生命周期，通过定时、定期和惰性删除来管理过期键

1. **定时删除**：设置了过期时间的键在过期时被自动删除，Redis 会在每次访问时检查这些键的过期状态。
2. **定期删除**：Redis 会定期随机检查一部分键的过期时间，删除已经过期的键。这个检查过程是以一定的频率（由 `hz` 参数控制）进行的。
3. **惰性删除**：当访问某个键时，Redis 检查该键是否已过期，如果已过期则删除该键。这种策略在键未被访问时不会主动删除它们。

19、redis的淘汰策略（回收策略）

内存淘汰策略主要用于，确保 Redis 在内存使用超出配置限制时能够继续运行并处理新的请求。与键的过期时间管理无关，而是涉及如何处理超出配置内存限制的情况：

- `noeviction`：当内存达到限制时，Redis 会返回错误，而不会删除任何数据。这是默认策略。
- `allkeys-lru`：从所有键中选择最近最少使用（Least Recently Used, LRU）的键进行删除，以释放内存。

- `volatile-lru`：仅从设置了过期时间的键中选择最近最少使用的键进行删除。这意味着只有有过期时间的键才会被淘汰。
- `allkeys-random`：从所有键中随机选择一些键进行删除。
- `volatile-random`：仅从设置了过期时间的键中随机选择一些键进行删除。
- `volatile-ttl`：从设置了过期时间的键中选择那些距离过期时间最近的键进行删除。
  配置：
  要配置 Redis 的内存淘汰策略，可以在 `redis.conf` 文件中设置 `maxmemory-policy` 参数

20、内存淘汰策略和过期删除策略

这两种策略协同工作，确保 Redis 在处理高负载和大数据量时能够高效地运行：

- **内存淘汰策略**： 主要用于管理 Redis 的内存使用，确保在达到内存限制时能够继续处理新请求。
- **键过期策略**： 处理键的生命周期，通过定时、定期和惰性删除来管理过期键。

21、Redis如何实现分布式锁

- **使用SETNX命令+设置过期时间**：当指定的key不存在时，SETNX命令会创建并为其设置值，返回状态码1；若key已存在，则返回0。这可以用于尝试获取锁，只有设置成功的客户端才获得锁。同时，为避免锁被永久持有，可以为锁设置一个过期时间，使用EXPIRE命令。
  用于多进程或多线程环境中需要协调对共享资源的访问的情况
- **基于 Redlock 算法的实现**

22、redis锁、原子性、事务

1. **锁**：基于 `SETNX` 命令的锁、基于 `WATCH` 命令的锁、基于 Redlock 算法的锁，确保同一时间只有一个客户端能操作共享资源，避免数据竞争。
2. **原子性**： `SETNX`： 原子地设置 key 仅在 key 不存在时。
   `INCR`/`DECR`： 原子地增加或减少数值。
   `HINCRBY`： 原子地对哈希表中的字段进行递增操作。
   `WATCH`： 用于监视键，并在事务中提供原子性，确保事务操作的安全性。
   3.**事务**： `MULTI`：开启事务。
   `EXEC`：执行事务中的所有命令。
   `DISCARD`：取消事务。
   `WATCH`：用于监视键，并在事务中提供乐观锁机制，确保数据的一致性。
   总结：

- **Redis 锁** 用于协调对共享资源的访问，确保在同一时刻只有一个进程或线程可以持有锁。
- **原子性** 确保操作不可中断，以维护数据的一致性和完整性。
- **事务** 是一组操作的集合，提供原子性，确保所有操作要么全部成功，要么全部失败。

23、redis怎么保证原子性的?

1. **单个命令的原子性**：Redis中的许多命令，如SET、GET、INCR、DECR等，都是原子性的，即这些命令在执行过程中不会被其他命令打断。
2. **事务**：Redis支持事务操作，通过MULTI、EXEC、DISCARD和WATCH命令，可以将多个命令打包成一个事务执行，保证事务中的所有命令要么全部执行成功，要么全部不执行，从而保持原子性。
3. **分布式锁**：在分布式环境下，可以使用分布式锁（如Redlock）来保证多个客户端对同一资源的访问是原子的，即同一时间只有一个客户端能获取锁并执行操作。

24、redis高可用了解吗

高可用主要有以下几种实现方式：

1. **主从复制（Replication）**：主节点处理所有写操作，同时将这些操作复制到从节点。这样，从节点可以在主节点故障时接管工作，并且可以用于读取负载均衡。但主节点故障时需手动切换。

- **数据一致性**：主节点将写操作同步给从节点，确保从节点的数据与主节点一致。

2. **哨兵（Sentinel）**：Redis 哨兵系统负责监控 Redis 实例的状态，自动处理故障转移和通知。它监控主节点和从节点的状态，确保系统的高可用性。还提供自动故障恢复和监控。

- **数据连续性**：当主节点故障时，Sentinel 会自动选举一个从节点作为新的主节点，并将其他从节点重新配置为新的主节点的从节点。

3. **Redis 集群（Cluster）**：Redis 集群通过数据分片将数据分布在多个节点上，每个节点负责一个或多个数据槽，每个主节点都有一个或多个从节点作为备份。集群可以自动处理节点故障并重新分配数据。

- **数据一致性**：数据分片和主从复制结合使用，以实现数据的高可用性和分布式存储。故障恢复时，通过从节点接管主节点的工作来保证数据的连续性。

25、对redis的高可用机制了解吗？如何保证主从节点的数据一致？具体更新流程

Redis通过主从复制、哨兵模式和集群模式来保证高可用性。原理：基于一致性哈希算法、虚拟节点、主从复制、哨兵模式以及自动故障转移机制
主从复制机制：

1. **主节点（Master）**：负责处理所有的写操作，并将这些操作复制到从节点（Slave）。
2. **从节点（Slave）**：复制主节点的数据，用于备份和读取操作。当主节点发生故障时，从节点可以提升为主节点。
   保证主从节点数据一致性的方法主要包括：主从复制并不保证强一致性，而是最终一致性。在某些故障情况下（如网络分区或主节点崩溃），可能会存在数据不同步的情况

- **主从复制**：主节点将写操作同步给从节点，确保数据一致。
- **哨兵模式**：监控主从节点状态，自动进行故障转移，保证数据连续性。
- **集群模式**：通过数据分片和复制，实现多节点间的数据同步和故障恢复。
  具体更新流程：

1. **全量同步**：
   - 从节点连接到主节点并发送 `SYNC` 命令请求数据同步。
   - 主节点执行 `BGSAVE` 命令生成 RDB 快照文件。`BGSAVE` 会在后台创建一个快照文件（RDB 文件），用于捕获当前的数据状态。
   - 主节点将 RDB 文件发送给从节点。
   - 从节点接收到 RDB 文件后，加载文件中的数据到内存中，完成全量同步。
2. **增量同步**：
   - 在全量同步完成后，主节点将新的写操作（增量数据）记录到复制缓冲区。
   - 主节点将这些增量写命令发送给从节点。
   - 从节点接收到这些增量写命令后，按顺序执行，保持数据与主节点一致。

26、主从复制的步骤是怎么做的

主从复制的步骤如下：

1. 从节点连接主节点，并发送 SYNC 命令请求复制数据。
2. 主节点接收到 SYNC 命令后，开始进行复制数据。
3. 主节点将数据发送给从节点进行复制。
4. 从节点接收到数据后，将其保存到本地数据库中。
5. 主节点将增量数据发送给从节点进行同步。
6. 从节点接收到增量数据后，将其保存到本地数据库中。
7. 主从复制完成。
   和上述更新流程类似

27、redis主从怎么搭建

1. **环境准备**：确保所有节点（主节点和从节点）已安装Redis，并关闭防火墙等可能影响通信的服务。
2. **修改配置文件**：
   - **主节点**：通常不需要特别配置，确保Redis服务正常运行即可。
   - **从节点**：需要配置`replicaof`（Redis 5.0及以后版本使用`replicaof`，之前版本使用`slaveof`）指令，指定主节点的IP地址和端口。同时，可以配置`replica-read-only yes`以确保从节点只读。
3. **启动Redis服务**：在所有节点上启动Redis服务。
4. **验证配置**：通过`info replication`命令在主节点上查看从节点信息，确认主从复制关系已正确建立。同时，可以在从节点上执行读操作，验证数据同步是否正常。
5. **测试**：在主节点上写入数据，检查从节点是否能及时同步这些更改。

28、redis 多路复用(I/O多路复用)

主要基于非阻塞I/O和事件驱动机制，如epoll（Linux特有）、select或poll等，其中Redis默认使用epoll。
Redis 使用 `select`、`epoll` 或 `kqueue` 等系统调用来监听多个客户端的I/O事件，这种机制允许 Redis 高效地管理大量的连接和请求，从而提高整体性能和响应速度。
在实际应用中，这种多路复用机制使 Redis 能够处理高并发的操作，主要应用：高并发环境、数据缓存、实时分析、消息队列、分布式锁

29、redis常见性能问题

1. **内存溢出问题**：
   - 当数据量过大或存储的key较多时，可能导致内存溢出，影响Redis性能。
   - 解决方案包括使用Redis的内存淘汰策略、优化数据结构、分片或使用集群模式等。
2. **高延迟和慢查询**：
   - 执行大量复杂命令或处理大数据量时，可能导致高延迟。
   - 解决方案包括避免使用高复杂度的命令、使用更合理的数据模型和查询方式等。
3. **IO瓶颈**：
   - 较大的数据处理操作可能会阻塞Redis主线程，导致性能下降。
   - 解决方案包括合理利用异步操作、使用多线程架构、优化I/O操作等。

30、redis如何实现高并发

- **单线程事件循环**：Redis采用单线程架构，避免了多线程环境下的锁竞争和上下文切换，简化了并发问题。

- **I/O多路复用**：该模型允许单个线程同时监听多个套接字和管道，并高效地处理传入的事件，快速响应客户端请求。
- **无锁数据结构**：Redis使用无锁数据结构来管理数据，消除了锁争用和死锁的风险，提高了并发性能。
- **管道化**：允许客户端将多个请求打包成一个请求发送给服务器，减少网络开销和服务器端处理时间。
- **集群模式**：通过集群模式，Redis可以将负载分布到多个服务器上，提升整体并发处理能力。

31、redis部署在项目中会有哪些好处

1. **性能提升**：Redis以内存为基础，提供极快的数据读取和写入速度。
2. **缓存机制**：它可以作为缓存层，减少对数据库的直接访问，显著降低延迟。
3. **数据结构支持**：Redis支持多种数据结构，如字符串、哈希、列表、集合等，适用于不同的数据需求。
4. **持久化选项**：提供不同级别的数据持久化选项，确保数据的可靠性。
5. **高可用性**：支持主从复制和高可用部署，增强系统的可靠性和可扩展性。
   redis快：基于内存，高效数据结构，io多路复用等；数据稳定恢复：redis事务，持久化RDB和AOF，主从复制结构，哨兵监听重新选举等

32、消息队列用来干什么

消息队列（MQ）主要用于解决不同进程与应用程序之间的通讯问题，作为一种分布式消息容器或中间件。应用场景主要包括：

- **异步处理**：将非核心或耗时的任务异步处理，提高系统响应速度和并发能力。例如，用户注册后，发送注册邮件和短信可以异步进行，减少用户等待时间。
- **应用解耦**：通过消息队列，不同系统或服务之间可以实现松耦合，降低系统间的直接依赖。例如，订单系统和库存系统通过消息队列通信，避免直接调用接口带来的耦合问题。
- **流量削峰**：在并发访问高峰期，通过消息队列缓冲请求，平滑处理突发流量，减轻数据库等后端服务的压力。

33、redis延迟队列怎么做的?

利用 Redis 的数据结构和功能来存储和管理任务，确保它们在指定时间后被处理
使用有序集合（Sorted Set）
使用 Redis Key 过期功能
使用 Redis Streams

34、项目中的秒杀超卖如何解决的?

- **库存预扣减**：在用户下单时，系统先进行库存预扣减，即暂时将库存数量减去用户想要购买的数量。如果预扣减成功，则用户可以继续支付流程；如果库存不足，则直接提示用户商品已售罄，防止超卖。
- **使用分布式锁**：确保同一时间只有一个客户端能修改库存，避免并发情况下的超卖。
- **数据库乐观锁**：在更新库存前检查库存版本号，只有在版本号未变更的情况下才执行更新，避免并发更新导致的超卖。
- **动态库存检查**：在用户付款环节再次检查库存，确保库存数量的准确性。

35、用redis做过stream异步队列

1. **创建Stream**：首先，在Redis中创建一个Stream类型的消息队列，用于存储消息。
2. **生产者发送消息**：生产者使用XADD命令向Stream中添加消息。消息可以包含多个字段和值，每个消息都有一个唯一的ID。
3. **消费者读取消息**：消费者使用XREAD或XREADGROUP命令从Stream中读取消息。XREADGROUP命令支持消费者组的概念，可以实现消息的负载均衡和消息确认机制。
4. **处理消息**：消费者读取到消息后，根据业务需求进行处理。
5. **消息确认**：处理完消息后，消费者需要向Redis发送XACK命令来确认消息已被处理，这样Redis就可以将这条消息从消费者的待处理列表中移除。

36、利用redis实现计数统计和用户积分排行

1.**计数统计**的基本思路是：使用Redis的字符串（String）来实现

- 将每个计数项作为Redis的一个键（key），其值（value）为计数器的当前值。
- 使用`INCR`命令对计数器进行增加操作，每次调用都会使计数器的值增加1。
- 如果需要减少计数器的值，可以使用`DECR`命令。
- 使用`GET`命令可以获取计数器的当前值。
  2.**用户积分**排行的基本思路是：使用Redis的有序集合（Sorted Set）来实现
- 将用户的ID作为有序集合的成员，用户的积分作为该成员的分数。
- 使用`ZADD`命令向有序集合中添加成员及其分数，如果该成员已经存在，则会更新其分数。
- 使用`ZRANGE`命令可以获取有序集合中指定排名范围内的成员列表，以及他们的分数。通过指定不同的排名范围，可以实现不同的查询需求，比如查询积分最高的前10名用户。
- 如果需要更新用户的积分，可以再次使用`ZADD`命令，Redis会根据新的分数重新排序。

37、redis访问数据？数据库有十万数据，如何查询

查询redis，不存在去查数据库，将该数据存到redis

38、redis

特点： Redis数据库完全在内存中，使用磁盘仅用于持久性
相比许多键值数据存储，Redis拥有一套较为丰富的数据类型
Redis可以将数据复制到任意数量的从服务服务中

39、Redis处理客户端连接、指令执行的流程

客户端连接流程
1.建立连接：
客户端通过TCP/IP协议（默认端口6379）或Unix Domain Socket（UDS）向Redis服务器发送连接请求。
Redis服务器在接收到连接请求后，会创建一个新的客户端连接对象，并将其加入连接列表。
2.身份验证：
如果Redis服务器配置了密码，客户端在建立连接后需要发送认证命令（AUTH）进行身份验证。
验证通过后，客户端才能获得访问数据库的权限
指令执行流程
1.命令发送：
客户端根据用户输入创建相应的命令，命令通常由命令名和参数组成。
客户端将命令序列化为二进制格式，通常使用RESP（Redis Serialization Protocol）格式。
客户端通过Socket将序列化后的命令发送给Redis服务器3。
2.命令接收与解析：
Redis服务器接收客户端发来的命令请求，将其缓存到客户端输入缓冲区中。
对输入缓冲区中的命令进行分析，提取命令参数和参数个数，分别保存到客户端状态的argv属性和argc属性中。
调用命令执行器，根据argv参数在命令表中查找指定的命令，并保存到客户端状态的cmd属性中。
3.命令执行准备：
进行一系列预备操作，如命令校验、参数校验、权限校验、内存检测等。
如果服务器打开了监视器功能，将执行的命令和参数等信息发送给监视器。
4.命令执行：
调用命令的实现函数（redisCommand结构的proc属性，它是一个函数指针）来执行最终的命令。
命令执行完毕后，将命令回复保存到客户端的输出缓冲区中。
5.命令回复发送：
当客户端套接字变为可写状态时，服务器执行命令回复处理器，将保存在客户端输出缓

冲区中的命令回复发送给客户端。
回复处理器清空客户端输出缓冲区，为下一个命令请求做好准备

40、Redis哨兵机制主节点故障发现和集群选主是如何做的

41、热点数据还可以使用什么别的数据结构，数据较大怎么办

Docker

1、Docker file了解吗?

它是一个文本文件，包含了一系列的指令和命令，用于自动化构建 Docker 镜像。创建 Dockerfile 后，docker build -t * (构建Docker镜像的名称和标签)
FROM: 指定基础镜像
RUN: 执行命令并在镜像中创建一个新的层
COPY: 将文件或目录从主机复制到镜像中
ADD: 与 `COPY` 类似，但还可以解压归档文件并下载远程文件
WORKDIR: 设置工作目录，之后的指令都会在这个目录下执行
CMD: 指定容器启动时要执行的命令。可以被 `docker run` 命令覆盖
ENTRYPOINT: 定义容器启动时的默认执行程序，不能被 `docker run` 命令覆盖
EXPOSE: 声明容器监听的端口
ENV: 设置环境变量
VOLUME: 创建一个挂载点，用于持久化数据（容器的目录，无法从 Dockerfile 中挂载主机目录）

2、解释一下dockerfile 的ONBUILD 指令?

`ONBUILD` 是定义在父镜像的Dockerfile中，但在子镜像构建时才执行。
`ONBUILD` 指令在父镜像的 Dockerfile 中定义，但其实际执行是在基于该镜像的子镜像 Dockerfile 中进行构建时。父镜像中的 `ONBUILD` 命令不会在父镜像构建时执行，而是在每次子镜像构建时触发

3、DockerFile 中的命令COPY 和ADD 命令有什么区别?

- `COPY` 是专门用于文件和目录复制的命令
- `ADD` 额外支持解压归档文件和支持从 URL 下载

4、什么是docker 镜像?

作为容器运行的软件集合，包含一组指令来创建可在 Docker 平台上运行的容器。镜像是不可变的，如需更改则需要构建新的镜像。
Docker 镜像是一个轻量级、可执行的独立软件包，它包含了运行应用程序代码、依赖库、环境变量以及配置文件。
镜像是容器的模板，通过镜像可以创建和运行 Docker 容器。

- **不可变性**：镜像一旦创建，不会被修改。任何修改都需要创建一个新的镜像。
- **层次结构**：镜像由多个层组成，每一层都是基于前一层的增量更新，优化了存储和重复利用。
- **构建**：通过 Dockerfile 定义构建镜像的步骤，从而自动化创建镜像的过程。
- **分发**：镜像可以通过 仓库进行共享和分发，使得应用程序能够在不同环境中一致地运行。

5、==构建docker 镜像应该遵循哪些原则？==

1. **保持镜像小巧**：减少镜像的体积可以加快下载和启动速度。使用较小的基础镜像，并尽量只包含应用程序运行所需的最小依赖。
2. **单一职责原则**：避免在一个镜像中部署多个应用或服务，提高镜像的可重用性和可维护性
3. **利用缓存**：利用Dockerfile构建过程中的缓存机制，将变化频率较低的指令放在前面，变化频繁的指令放在后面
4. **层合理规划**：尽量减少镜像层的数量，通过合并相关操作减少冗余层。比如将多个 `RUN` 命令合并成一个，使用`&&`连接命令。
5. **清理临时文件**：在镜像构建过程中删除不再需要的临时文件和缓存，以减少镜像体积。例如，删除包管理器缓存或编译产生的临时文件。
6. **安全性**：使用安全镜像作为基础镜像。尽量避免在镜像中使用不安全的应用或配置。
7. **使用 `.dockerignore` 文件**：排除不必要的文件和目录，减少上下文传输时间和构建镜像所需空间。
8. **明确和文档化**：使用明确的标签和版本号标记镜像，确保可追溯性。编写清晰的 Dockerfile 注释，帮助理解每个步骤的目的和作用。
9. **多阶段构建**：使用多阶段构建来分离构建和运行环境，只将最终需要的构建成果复制到最终镜像中，进一步减小镜像体积。

6、容器的理解

容器即服务 (CaaS) 或容器服务是一种用于管理容器生命周期的托管式云技术服务。它可以帮助编排（启动、停止、扩展）容器运行，简化、加速并实现应用开发与部署生命周期自动化。
容器是一种轻量级的虚拟化技术，用于运行单个或多个应用程序。
容器技术是将应用程序及其所有依赖项打包到可移植的容器中，这样可以在不同的环境中快速部署和运行应用程序，无需担心环境差异。
核心特点是其轻量级和可移植性，使得容器能够在同一物理机上运行多个容器，每个容器都有独立的环境。

1. **隔离性**：容器提供了进程和文件系统的隔离，使得不同容器中的应用程序相互独立，不会互相干扰。

2. **轻量级**：因为容器共享宿主操作系统的内核，启动速度快，占用资源少，比虚拟机更为高效。
3. **便携性**：容器可以在任何支持容器运行的环境中保持一致性，包括开发、测试和生产环境，简化了应用的迁移和部署。
4. **可移植性**：容器将应用程序及其所有依赖打包在一起，使得应用可以在不同的计算环境中无缝运行。
5. **可管理性**：使用容器编排工具（如 Docker Compose、Kubernetes）可以高效地管理和调度容器，支持自动扩展、负载均衡等功能。
6. **快速部署**：容器化应用启动迅速，支持快速开发、测试和发布周期。

7、什么是docker 容器?

Docker的容器是镜像创建出来的运行实例。是一个轻量级、独立的运行环境，用于封装和隔离应用程序及其依赖项。
它与虚拟机不同，容器不需要包括整个操作系统，共享宿主机的操作系统内核，包含应用程序和其运行所需的库、配置文件等。

1. **隔离性**：容器提供了进程和文件系统的隔离，使得不同容器中的应用程序相互独立，不会互相干扰。
2. **轻量级**：因为容器共享宿主操作系统的内核，启动速度快，占用资源少，比虚拟机更为高效。
3. **便携性**：容器可以在任何支持容器运行的环境中保持一致性，包括开发、测试和生产环境，简化了应用的迁移和部署。
4. **可移植性**：容器将应用程序及其所有依赖打包在一起，使得应用可以在不同的计算环境中无缝运行。
5. **快速部署**：容器化应用启动迅速，支持快速开发、测试和发布周期。

8、docker 容器有几种状态?

Created（已创建）Restarting（重启中）Running（运行中）Paused（已暂停）Exiting（退出中）Exited（退出）Dead（已停止）

9、==为什么容器之间不会相互影响 ==

1. **命名空间**：Docker 利用 Linux 命名空间功能来实现隔离。每个容器都运行在独立的命名空间中，防止进程和资源之间的相互干扰。
2. **控制组（cgroups）**：控制组功能用于限制和管理容器的资源使用。通过 cgroups，Docker 可以对容器的 CPU、内存、I/O 等资源进行限制和监控，确保容器之间不会因为资源争用而互相影响。
3. **网络隔离**：容器通常在独立的网络命名空间中运行，拥有自己的网络接口和 IP 地址。容器间的网络流量是隔离的，只有通过显式的配置才能实现相互访问。

4. **文件系统隔离**：Docker 使用分层的文件系统，每个容器有自己的文件系统视图，包含了它的应用程序和数据。容器的文件系统是基于 Docker 镜像构建的，容器之间的文件系统层是分开的，不会相互干扰。

10、docker是怎样实现资源隔离和资源限制的

1. **命名空间 (Namespaces)**：Docker 使用 Linux 命名空间来提供进程、网络、文件系统等的隔离。例如，`PID` 命名空间隔离进程 ID，`NET` 命名空间隔离网络资源等。
2. **控制组 (cgroups)**：控制组用于限制和监控容器的资源使用，包括 CPU、内存、磁盘 I/O 等。它们确保每个容器的资源使用不会超过预设的限制。
3. **文件系统**：Docker 使用分层的文件系统，使得每个容器拥有自己的文件系统视图。

11、容器与主机之间的数据拷贝命令?

- 从主机拷贝文件到容器：`docker cp /path/on/host container_id:/path/in/container`
- 从容器拷贝文件到主机：`docker cp container_id:/path/in/container /path/on/host`
  `container_id` 是容器的 ID 名称。

12、docker与虚拟机的区别

- 内核：虚拟机隔离性更好，因为虚拟机有单独的系统内核，Docker与宿主机共享系统内核，虚拟机相当于物理层面的隔离，Docker相当于应用层面的额隔离。
- 大小：Docker镜像一般在几十M到几百M，比较轻量，虚拟机一般在在几G，比较笨重。
- 速度：Docker共享宿主机内核一般秒级启动，虚拟机时分钟级别的，需要启动完整的操作系统环境，包括启动一个完整的内核。
- 资源：Docker占用更少的资源，虚拟机有完整的系统所以占用资源比较多。

13、如何在生产中监控docker?

Docker 自带的工具、集中化日志管理、性能监控工具、报警和通知，Docker提供**docker stats**和**docker events 等工具**来监控生产中的Docker。
1.Docker命令行工具
- `docker ps`：查看运行中的容器。
- `docker stats`：实时查看容器资源使用情况（CPU、内存、网络、磁盘）。
- `docker events`：监控Docker daemon的事件。
- `docker logs`：查看容器日志。
2.Elasticsearch, Logstash, Kibana

3.Prometheus 和 Grafana
4.Prometheus、Grafana基于资源使用情况、日志内容或其他指标设置报警规则，当指标超出阈值时，发送通知

14、常用docker命令

15、容器退出后，通过docker ps命令查看不到，数据会丢失么？

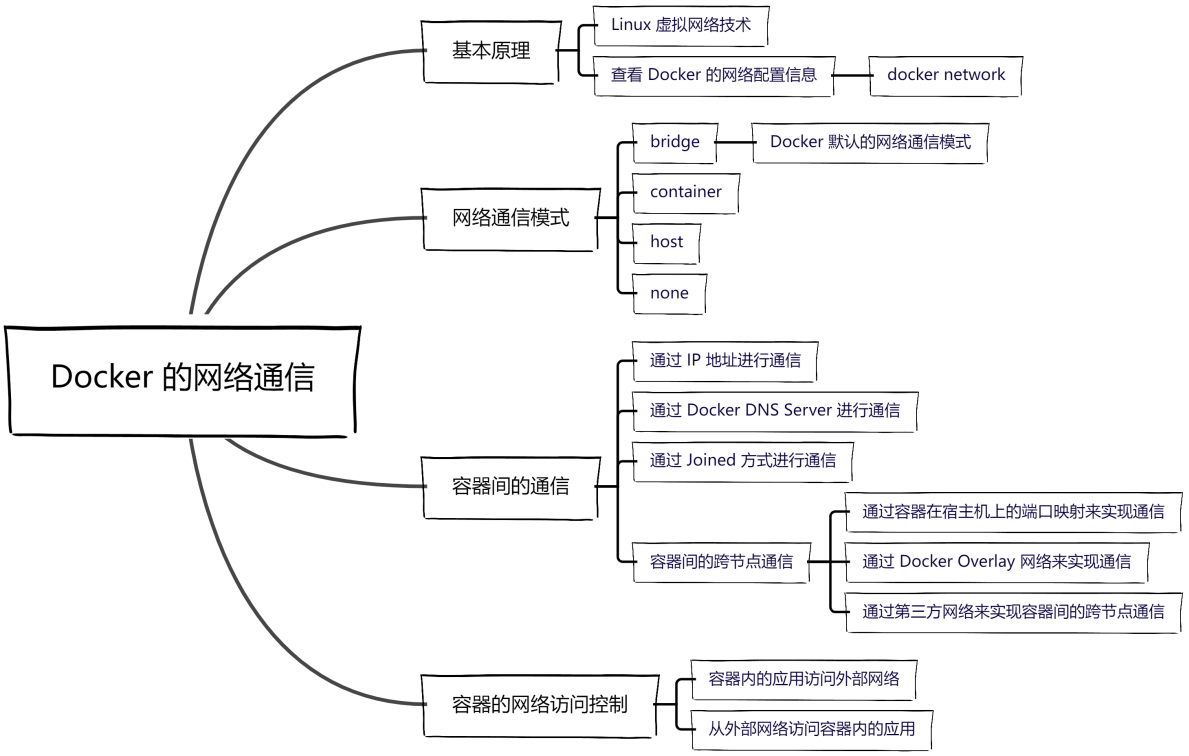
容器退出后会处于终止（exited）状态，此时可以通过 docker ps -a 查看，其中数据不会丢失，还可以通过 docker start 来启动，只有删除容器才会清除数据。

16、docker如何实现相互通信

A
从容器和容器之间、容器和宿主机之间、容器和宿主机之外这几方面
1.**容器和容器之间**：容器之间可通过 IP，Docker DNS Server 和joined 容器三种方式通信。

- 自定义网络：将容器连接到同一个自定义网络（如 `bridge` 网络），容器可以通过其名称或服务名直接进行通信。
- 默认桥接网络：如果容器使用 Docker 的默认 `bridge` 网络，可以通过容器的 IP 地址进行通信，但不建议使用容器名称进行通信，除非配置了额外的 DNS 解析。
  2.**容器和宿主机之间**
- 端口映射：通过 `-p` 在启动容器时映射宿主机和容器的端口。例如，`-p 8080:80` 将宿主机的 8080 端口映射到容器的 80 端口。宿主机可以通过 `localhost:8080` 访问容器中的服务。
- 主机网络模式：使用 `--network host` 选项，容器与宿主机共享网络堆栈，容器可以直接使用宿主机的 IP 地址进行通信。
  3.**容器和宿主机之外**
- 端口映射：将容器端口映射到宿主机端口，然后通过宿主机的公共 IP 地址进行访问。外部流量首先到达宿主机，再通过 `iptables` 规则转发到容器。
- 自定义网络驱动：对于需要更复杂网络架构的情况，可以使用 Docker 的 `overlay` 网络驱动，适用于多宿主机环境，使容器能够在不同宿主机之间进行通信。
  B
  先网络模式
  Docker的默认网络模式包括bridge、none、host和container
  再容器间
  可通过 IP，Docker DNS Server 和joined 容器三种方式通信
  最后容器访问外部网络，以及外部网络访问容器的应用
  容器内访问外部网络时，需要宿主机进行转发，开启 IP 数据包转发功能
  外部网络访问器内部的应用，Docker 采用宿主机与容器端口绑定，即利用 Linux

的 iptable 表将宿主机上的端口映射到容器内的端口。



17、==docker的网络分发 ==

Docker 的网络分发主要涉及在不同 Docker 主机或节点上实现网络连接和数据传输。Overlay 网络、通过容器在宿主机上的端口映射来实现通信、通过第三方网络来实现容器间的跨节点通信

18、讲一下docker容器的路径映射

路径映射（path mapping）是指在容器内部的文件路径和宿主机的文件路径进行映射，通过使用`docker run` `-v`或`--mount`实现。
有两种主要方式来实现路径映射：
数据卷挂载，在容器停止或删除后保留，适用于需要持久化数据的场景 `docker run -v volume_name:/container/path image_name`
绑定挂载，用于直接将宿主机的路径映射到容器内，实现双向同步。适合开发和调试时需要实时数据更新的场景
`docker run --mount source=my_volume,target=/container/path,readonly image_name`

19、Docker部署的映射端口有几种方式

主机端口映射：使用 `-p` 或 `--publish` 参数，例如 `-p 8080:80`，将容器的端口映射到主机端口，外部访问主机的 8080 端口，从而访问容器的 80 端口。
主机网络模式：使用 `--network host`，使容器共享主机的网络栈，这样容器直接使用主机的 IP 地址和端口，无需端口映射。

端口暴露：使用 `EXPOSE` 指令在 Dockerfile 中声明容器内部的端口，例如 `EXPOSE 80`，供 Docker 容器网络使用，不会映射到主机端口。需配合 `-p` 进行实际的端口映射

20、docker创建容器为什么要有端口映射

将容器内部的端口暴露到宿主机上，从而使容器内的服务能够被外部访问，通过端口映射，可以实现容器服务的访问、集成和管理。

21、docker compose有什么用

Docker Compose是一个工具，用于定义和运行多容器Docker应用程序，它通过 `docker-compose.yml` 文件来配置服务、网络和卷，通过命令启动、停止和管理整个应用程序的多个容器。
通过 `docker-compose up` 启动所有服务，使用 `docker-compose down` 停止和移除所有服务

22、docker重启策略

docker run --restart always my-image， `--restart` 选项支持以下几种策略：no、always、unless-stopped、on-failure
在 `docker-compose.yml` 文件中配置重启策略：

```
version: '3'
services:
  my-service:
    image: my-image
    restart: always
```

23、docker部署项目的过程、项目启动的命令

1. **编写Dockerfile**: 创建一个 `Dockerfile` 文件，定义镜像的构建过程，包括基础镜像、依赖项和配置。
2. **构建镜像**: 使用 `docker build` 命令构建 Docker 镜像：docker build -t my-image
3. **运行容器**: 使用 `docker run` 命令基于构建的镜像运行容器，并配置适当的参数，如端口映射、环境变量和重启策略：docker run --name my-container -p 80:80 -d my-image
4. **检查运行状态**: 使用 `docker ps` 确认容器是否正在运行：docker ps
5. **查看日志**: 使用 `docker logs` 命令查看容器的输出日志，以确保应用正常运行：docker logs my-container

24、项目用docker部署，那docker部署和普通部署有什么区别（docker在项目中实现的效果）

1.环境一致性：通过容器确保开发、测试和生产环境的一致性。所有依赖和环境配置都在 Docker 镜像中定义。
2.隔离性：每个应用在隔离的容器中运行，避免了应用间的冲突。
3.可移植性：镜像可以在任何支持 Docker 的平台上运行，便于跨环境迁移。
4.资源管理：容器化应用可以更高效地共享主机资源，并可以限制资源使用（如内存和 CPU）。
5.扩展性：容器可以快速启动和停止，便于水平扩展和负载均衡。
6.版本控制：镜像版本化清晰，可以轻松回滚到先前的版本。
Docker 部署简化了环境配置、提高了应用的可移植性和可管理性。

Linux&计网

1、linux热重启命令

热重启（即重新启动系统而不关闭电源）

- **reboot命令**：这是最常用的重启命令，它会发出一个重启信号，操作系统会立即开始关机和重启过程。使用方法如下：`sudo reboot`。
- **systemctl reboot命令**：使用systemd服务管理器来重启系统。使用方法如下：`sudo systemctl reboot`。
- **init 6命令**：将系统置于运行级别6，导致系统重启。使用方法如下：`sudo init 6`。
- **shutdown -r命令**：将系统关机并自动重新启动。使用方法如下：`sudo shutdown -r now`。

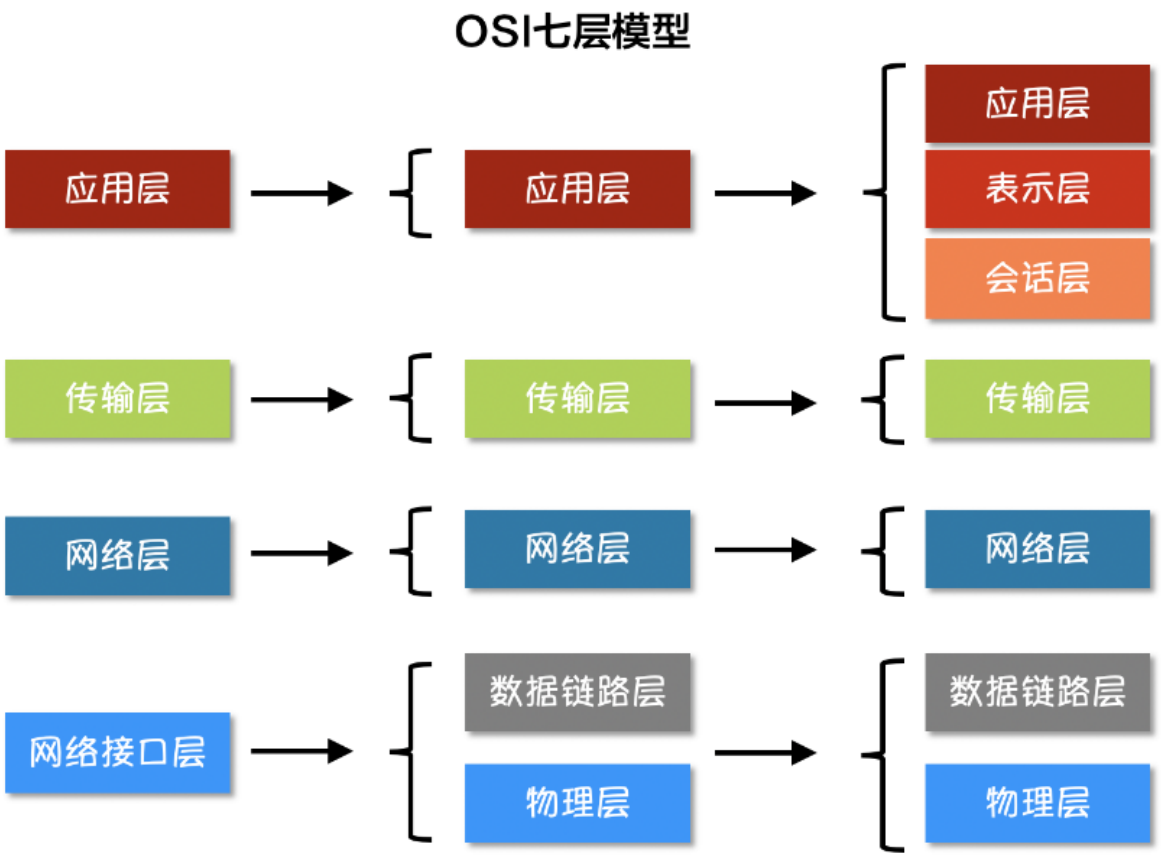
2、在普通用户下把一个文件解压复制到根目录下的bin目录

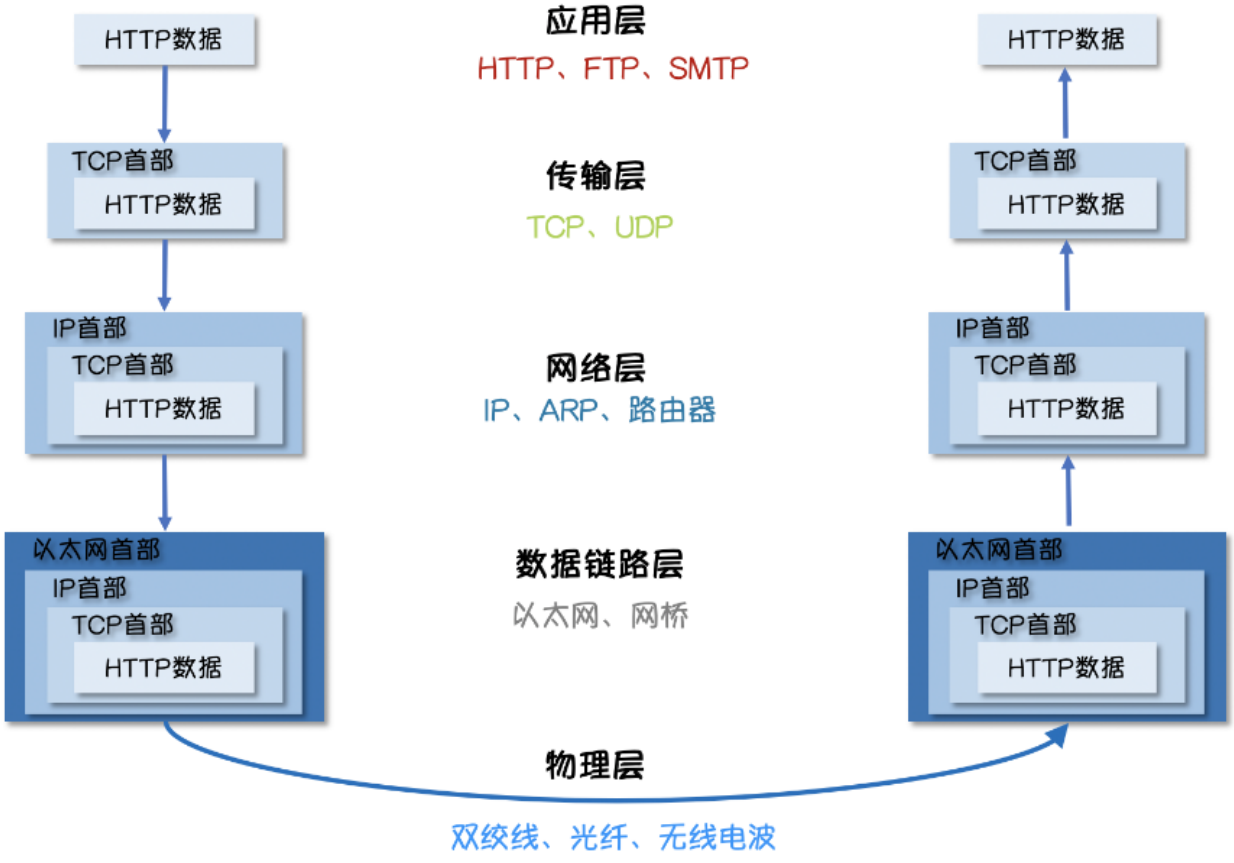
sudo
`sudo tar -xzvf your_file.tar.gz -C /bin/`
`sudo cp your_file /bin/`
压缩
`tar -czvf myfile.tar.gz /path/to/your/file`
`sudo cp myfile.tar.gz /bin/`

3、查看端口占用、CPU负载、内存占用，如何发送信号给一个进程

- **查看端口占用**：可以使用`lsof -i:端口号`或`netstat -tunlp`命令来查看某个端口是否被占用。
- **查看CPU负载**：可以使用`top`或`uptime`命令来查看CPU的负载情况。
- **查看内存占用**：可以使用`free -h`、`vmstat -s`或`top`命令来查看内存的使用情况
- `kill -Signal pid`，其中`Signal`是发送给进程的信号，`pid`是进程号

4、7层模型





5、GET和POST的区别

- **GET**用于获取数据，不会改变服务器状态，数据附加在URL中，并且对 URL 长度有要求，容易被缓存和记录。
- **POST**用于提交数据以更改服务器状态，数据放在请求体中，适合发送大量或敏感信息，不易被缓存。

6、TCP在哪一层，Http在哪一层

tcp传输层，http应用层
ARP（地址解析协议）和ICMP（Internet控制消息协议）等，用于网络层的控制和通信
应用层包括HTTP协议、FTP协议、SMTP协议、modbus、fins

7、TCP与UDP的区别联系

1. **连接性**：TCP是面向连接的协议，需要在数据传输前建立连接；UDP是无连接的协议，发送数据前不需要建立连接。
2. **可靠性**：TCP提供可靠的数据传输服务，通过重传机制确保数据不丢失；UDP不提供可靠性保证，可能会出现数据丢失。
3. **传输效率**：TCP传输速度相对较慢，因为需要建立连接和进行确认；UDP传输速度较快，因为无需建立连接和进行确认。
4. **资源占用**：TCP对系统资源要求较多；UDP对系统资源要求较少。
5. **通信模式**：TCP只能是一对一通信；UDP支持一对一、一对多、多对一和多对多通信。

8、TCP三次握手，四次挥手

- **三次握手**是TCP连接的建立过程。首先，客户端向服务端发送一个带有自身初始序号的同步报文，进入SYN-SENT状态。其次，服务端收到请求后，发送一个带有自身初始序号的同步确认报文，进入SYN-RECEIVED状态。最后，客户端收到应答后，向服务端发送确认报文，进入ESTABLISHED状态，连接建立成功。
- **四次挥手**是TCP连接的释放过程。首先，客户端向服务端发送一个带有FIN标记的报文，请求断开连接。其次，服务端收到请求后，发送一个确认报文，但可能还有数据需要发送，所以连接并未立即断开。当服务端数据发送完毕后，再次向客户端发送一个带有FIN标记的报文。最后，客户端收到报文后，发送确认报文，并等待一段时间后断开连接。服务端在收到确认报文后也断开连接

9、TCP三次握手和四次挥手后进入什么状态

- **三次握手后进入ESTABLISHED状态**：在TCP三次握手过程中，客户端和服务器通过交换SYN和ACK报文来建立连接。当客户端收到服务器的最后一个ACK报文后，双方进入ESTABLISHED状态，表示连接已成功建立，可以进行数据传输。
- **四次挥手后进入CLOSED状态**：在TCP四次挥手过程中，客户端和服务器通过交换FIN和ACK报文来释放连接。当客户端收到服务器的最后一个ACK报文后，经过

一定的等待时间（TIME-WAIT状态），客户端进入CLOSED状态。服务器在发送完最后一个ACK报文后也进入CLOSED状态，表示连接已完全释放。

10、如果客户端发送了两次SYN包，会建立几个tcp连接

如果客户端发送了两次SYN包，**在正常情况下，会尝试建立两个TCP连接。**每次发送SYN包都是一个新的连接请求，除非服务器端的TCP实现有特殊的处理逻辑（如合并连接请求），否则每个SYN包都会被独立处理，并尝试建立一个独立的TCP连接。
TCP连接的建立是通过三次握手过程完成的，包括客户端发送SYN包、服务器回复SYN/ACK包、客户端再回复ACK包。因此，每个SYN包都会触发这个三次握手过程，除非连接请求被拒绝或超时。

11、TCP怎么保证可靠性

确认重传机制：TCP通过序列号对每个报文段进行编号，接收方收到后发送确认应答（ACK）。若发送方未收到ACK，则重传报文段，确保数据不丢失。

12、TCP四次挥手TIME WAIT状态是干嘛的，如果服务器产生过多TIME WAIT，怎么处理

TIME_WAIT状态用于确保TCP连接的可靠关闭，并防止旧连接的数据干扰新连接。
如果服务器产生过多TIME_WAIT状态，可以采取以下措施处理：

- **调整系统内核参数**：如增加TIME_WAIT套接字的重用和快速回收，减少TIME_WAIT的持续时间。
- **使用长连接**：减少连接建立和关闭的次数，从而降低TIME_WAIT状态的产生。
- **增加端口范围**：通过修改系统配置，增加可用的本地端口范围，以支持更多的并发连接。

13、为什么要设置CLOSE WAIT状态。

它表示在一个TCP连接中，一方已经发送了关闭连接的请求（FIN），但是另一方还没有完全关闭连接，仍在等待对方的关闭请求。这个状态的存在是为了处理双向关闭连接的情况，确保两个方向上都要关闭连接。

14、TCP如果大量丢包会怎样

TCP大量丢包会导致网络性能下降、传输延迟增加以及连接稳定性下降

15、如果TCP每一次丢包重新发送会阻塞网络吗

通常情况下，TCP的重传机制不会阻塞网络。TCP协议设计有拥塞控制机制，当网络出现拥塞时，TCP会降低数据传输速率，以避免网络进一步恶化。

16、假如一段数据包被TCP分为12345部分，1一直丢包，2345怎么样

当TCP检测到数据包1丢失时，它会暂停发送后续的数据包（2345），并启动重传计时器。只有当数据包1成功传输并得到确认后，TCP才会继续发送数据包2。

17、TCP滑动窗口概念

TCP滑动窗口，用于**控制和管理发送方和接收方之间的数据传输，是TCP实现流量控制和拥塞控制的基础。**其工作原理如下：

- 在TCP中，发送方维护一个发送窗口（swnd），接收方维护一个接收窗口（rwnd）。这两个窗口表示了发送方可以发送的数据范围大小，由窗口的起始字节和窗口大小两个参数定义。
- 发送方将数据分成多个数据段，按顺序发送到接收方。每个数据段都包含一个序列号，标识其在发送方发送窗口中的位置。
- 接收方使用ack确认应答报文来通知发送方已成功接收数据，随后发送方通过ack报文进行窗口滑动。

18、TCP中syn flood怎么解决

SYN Flood攻击是一种网络攻击方式，通过发送大量的伪造SYN请求来占满目标服务器的连接队列，从而使合法用户无法建立连接。这种攻击利用了TCP三次握手过程中的SYN阶段，造成服务器资源被耗尽。

1. **SYN Cookies**：一种防御机制，通过不占用资源来响应SYN请求，直到三次握手完成后才分配资源。
2. **调整SYN队列长度**：增加SYN队列的大小，以处理更多的半连接请求。
3. **使用防火墙或入侵防御系统**：可以设置规则来识别并阻止异常流量。
4. **启用TCP连接限速**：限制每个IP的连接速率，以防止滥用。

19、从浏览器输入URL到页面得到展示的过程。

1. **URL解析**：浏览器解析输入的URL，提取出协议、主机名和路径等信息。
2. **DNS解析**：浏览器将主机名转换为对应的IP地址。这个过程称为DNS解析，包括客户机本地的递归查询和服务器的迭代查询。
3. **建立TCP连接**：浏览器通过IP地址和端口号与服务器建立TCP连接，确保双方能够正常通信。
4. **发送HTTP请求**：浏览器向服务器发送HTTP请求，请求页面或资源。
5. **服务器处理请求并返回响应**：服务器接收到请求后，进行相应的处理，并将处理结果封装成HTTP响应返回给浏览器。
6. **浏览器接收响应并渲染页面**：浏览器解析响应内容，根据HTML文档和相关资源，将页面内容呈现在用户界面上。

20、通过网址访问百度，描述一下整个过程。

DNS解析：浏览器首先检查缓存中是否有www.baidu.com的IP地址，若无，则向DNS服务器发起请求，通过多级DNS服务器查询，最终获得百度的IP地址。
建立TCP连接：浏览器通过三次握手与百度的服务器建立TCP连接，确保双方通信的可靠性。
发送HTTP请求：浏览器通过建立的TCP连接向百度服务器发送HTTP请求，请求获取百度首页的内容。
服务器处理并返回响应：百度服务器处理请求，生成HTTP响应，包含状态码、响应头和响应体（即网页内容），并通过TCP连接发送给浏览器。
浏览器渲染页面：浏览器接收并解析HTTP响应，渲染页面内容，最终展示给用户。
连接关闭和资源释放：页面加载完毕后，浏览器关闭与百度的TCP连接，释放相关资源。

21、访问页面时，和服务器是一次交互，还是多次交互?

访问页面时，通常会进行多次交互。
初次请求时，客户端与服务器之间会进行一次TCP连接建立，之后可能进行多次请求以获取页面上的不同资源（如图片、CSS、JavaScript等）。这些交互包括TCP握手、数据传输和TCP连接关闭等。

22、客户端多个并发请求，造成什么影响

1. **服务器负载增加**：更多的请求会消耗服务器资源，如CPU和内存。
2. **网络带宽消耗**：并发请求会占用更多的网络带宽，可能导致网络拥堵。
3. **响应时间延长**：服务器处理并发请求时，可能会增加每个请求的响应时间。

23、说说HTTP1.0、1.1、2.0的区别。

1. **HTTP/1.0**：
   - **连接**：每个请求-响应对使用一个新的 TCP 连接，效率较低。
   - **请求**：不支持持久连接，每个请求必须重新建立连接。
   - **缓存**：支持基本的缓存机制（如 `Expires` 头）。
2. **HTTP/1.1**：
   - **连接**：引入持久连接（`Keep-Alive`），可以在一个连接上处理多个请求-响应对，减少延迟。
   - **请求**：支持管道化（pipelining），多个请求可以排队在同一连接上，但仍然按顺序处理。
   - **缓存**：改进了缓存机制，增加了 `Cache-Control` 头，更灵活的缓存控制。
   - **其他特性**：支持分块传输编码（Chunked Transfer Encoding）和更多的状态码。
3. **HTTP/2.0**：

- **连接**：使用单一的连接处理多个请求和响应，通过多路复用（Multiplexing），大幅提高效率。
- **请求**：支持并发请求和响应在同一个连接中，解决了 HTTP/1.1 的队头阻塞问题。
- **压缩**：头部压缩（HPACK），减少了请求和响应的头部开销。
- **其他特性**：支持流（Streams）和优先级（Prioritization），优化了性能和延迟。

24、HTTP长连接和短连接的区别是什么?

- **连接持续时间**：长连接在一段时间内保持连接状态，可以多次使用同一连接进行请求和响应；而短连接则每次请求-响应交互都会建立一个新的连接，并在完成后立即关闭。
- **资源利用率**：长连接减少了TCP连接建立和拆除的开销，提高了性能和效率，适用于处理大量请求的场景；短连接则在网络资源利用上更为灵活，适用于处理少量请求或并发量大的情况，但可能增加网络开销和响应时间

25、HTTP大文件上传

26、http会复用底层tcp吗?

HTTP协议是构建在TCP协议之上的应用层协议，利用TCP提供的可靠、顺序的数据传输服务。HTTP通过TCP连接传输请求和响应消息，利用TCP的机制来保证数据的完整性和可靠性。

- **HTTP/1.1** 引入了 keep-alive，实现了连接复用的功能，从而达到了“长连接”，一个TCP连接可以复用多个HTTP请求和响应。
- **HTTP/2** 在一个TCP连接上使用多个流，实现了多路复用（Multiplexing），允许多个HTTP请求和响应在同一个连接上并发传输。

27、HTTPS的加密流程

客户端发起HTTPS请求：客户端向服务器发起HTTPS连接请求。
服务器证书传输：服务器将包含公钥的数字证书发送给客户端。
客户端验证证书：客户端验证证书的合法性、有效期及签名等，确保服务器身份可信。
会话密钥生成与加密：客户端生成会话密钥，并使用服务器的公钥对会话密钥进行加密后发送给服务器。
服务器解密会话密钥：服务器使用自己的私钥解密会话密钥。
加密通信：客户端和服务器使用会话密钥进行对称加密和解密，确保数据传输的安全性

客户端向服务器发起HTTPS连接请求，服务器将包含公钥的数字证书发送给客户端，客户端验证证书的合法性、有效期及签名等，生成会话密钥，并使用服务器的公钥对会话

密钥进行加密后发送给服务器，服务器使用自己的私钥解密会话密钥。通信的时候，客户端和服务器使用密钥进行加密和解密。

28、HTTPS流程，tls/ssl过程

1. **ClientHello**：客户端发起请求，包含支持的协议版本、加密套件列表、随机数等信息。
2. **ServerHello及证书交换**：服务器响应客户端，确认协议版本、加密套件，并发送服务器证书和随机数。
3. **证书验证**：客户端验证服务器证书的合法性，包括证书链的可信性、证书是否吊销、有效期及域名匹配等。
4. **密钥交换**：客户端生成预主密钥（Pre-master secret），用服务器公钥加密后发送给服务器。双方根据预主密钥和之前的随机数生成会话密钥。
5. **加密通信**：双方使用会话密钥进行加密通信，确保数据传输的安全性。

29、https是http加了什么，TLS，讲一下TLS协商密钥的过程。用户的协商用的key会被发到信道上吗，用的什么加密方式

TLS协商密钥的过程主要包括：

1. **客户端发起请求**：包含支持的协议版本、加密套件列表、随机数（rand1）等信息。
2. **服务器响应**：确认协议版本、加密套件，发送服务器证书、随机数（rand2）和选定的加密算法。
3. **客户端验证证书**：包括证书所有者、有效期、颁发机构等信息，并生成一个新的随机值（预主密钥），使用服务器的公钥加密后发送给服务器。
4. **双方生成会话密钥**：根据预主密钥和之前的随机数生成会话密钥，用于加密通信
用户的协商用的key**不会**直接发到信道上，而是使用**非对称加密**方式加密后传输，确保密钥的安全交换

30、http怎么确定包的边界，传一个很大的文件，怎么切

1. **Content-Length字段**：在HTTP头部显式设置Content-Length字段，表示消息体的长度，从而确定包的边界。这种方式适用于知道文件总大小的情况。
2. **Transfer-Encoding: chunked**：当不知道文件总大小时，可以使用分块传输编码。这种方式将文件分成多个块（chunk），每个块前面都带有表示该块大小的头部，以CRLF（回车换行符）结尾，块内容之后也紧跟CRLF，表示该块的结束。最后用一个长度为0的块表示整个消息的结束。这种方式特别适用于大文件传输。
对于传输大文件，通常会采用分块传输的方式，将大文件切割成多个小块进行传输，每块的大小可以根据网络状况动态调整，以提高传输效率。接收方在收到所有块后，再按照顺序重新组装成完整的文件。

31、http协议中，如何判断该报文已经传送完所有的数据并结束?

1. **Content-Length头**：服务器在响应中包含`Content-Length`头，指明了响应体的字节长度。客户端接收到指定字节数的数据后，知道报文结束。
2. **分块传输编码**（Chunked Transfer Encoding）：当`Transfer-Encoding: chunked`被使用时，数据被分块传送。每个块以其长度开始，以`\r\n`结束，最后一个块的长度为零，表示数据传送完毕。

32、HTTP的method有哪些

- **GET**：用于从服务器获取资源，通常用于请求数据，不会对服务器状态产生影响。
- **POST**：用于向服务器提交数据，常用于提交表单数据、上传文件等操作，可能会对服务器状态产生影响。
- **PUT**：用于向服务器上传数据，要求指定上传位置，用新的资源替换掉指定位置的资源。
- **DELETE**：请求服务器删除指定的资源。
- **PATCH**：部分更新资源，用于对资源进行局部更新。
- **HEAD**：与GET类似，但服务器只返回头部信息，不返回实际内容，常用于检查资源的状态或获取头部信息。
- **OPTIONS**：请求服务器告知支持的请求方法、支持的头部信息等，用于查看服务器支持的功能。
- **TRACE**：用于追踪请求在传输链路上的传输情况，主要用于诊断。
- **CONNECT**：用于告知服务器连接到目标地址，通常在使用SSL/TLS加密通信时使用。

33、HTTP的状态码有哪几种，是什么含义，列举几个你熟悉的状态码

1. **1xx（信息性状态码）**：
   - **100 Continue**：客户端应继续发送请求的其余部分。
   - **101 Switching Protocols**：服务器正在切换协议。
2. **2xx（成功状态码）**：
   - **200 OK**：请求成功，返回请求的资源。
   - **201 Created**：请求成功，资源已创建。
   - **204 No Content**：请求成功，但没有返回内容。
3. **3xx（重定向状态码）**：
   - **301 Moved Permanently**：请求的资源已被永久移动到新位置。
   - **302 Found**：请求的资源临时移动到其他位置。
   - **304 Not Modified**：资源未修改，可以使用缓存的版本。
4. **4xx（客户端错误状态码）**：
   - **400 Bad Request**：请求无效或格式错误。

- **401 Unauthorized**：未授权，需提供身份验证。
- **403 Forbidden**：服务器理解请求但拒绝执行。
- **404 Not Found**：请求的资源未找到。

5. **5xx（服务器错误状态码）**：
   - **500 Internal Server Error**：服务器遇到错误，无法完成请求。
   - **502 Bad Gateway**：服务器作为网关或代理时收到无效响应。
   - **503 Service Unavailable**：服务器暂时无法处理请求（通常是过载或维护）

34、HTTP状态码401和403的区别

401身份验证
403请求拒绝

35、HTTP GET和HEAD请求的区别

- **GET 请求**：
  - 用于请求指定的资源。
  - 服务器返回请求的资源及其内容（包括响应体）。
- **HEAD 请求**：
  - 用于获取资源的元数据，如头部信息，但不返回资源的内容。
  - 响应中只包含头部信息，正文部分为空。
  HEAD 请求通常用于检查资源的状态或验证资源是否存在，而 GET 请求用于实际获取资源及其内容。

36、HTTP keep-alive机制

Keep-Alive 是一种机制，用于优化HTTP连接的性能。它使得客户端和服务器之间的连接在完成一次HTTP请求/响应之后不会立即关闭，而是保持开放状态，以便进行后续的请求和响应。这可以减少建立和断开连接的开销，提高数据传输的效率。

37、HTTP是否可以在一次连接中发送多次请求而不等待后端返回

在 HTTP/1.1及更高版本中，可以在一次连接中发送多次请求而不等待后端返回，这得益于长连接（keep-alive）的特性。
在 HTTP/1.0 中，每次请求都需要单独的连接。

38、如果要做负载均衡，那么应该在哪一层上做文章?

- **应用层（第7层）**：可以基于应用层协议（如HTTP）来负载，能根据URL、浏览器类别、语言等决定是否进行负载均衡；适合需要根据内容或应用逻辑进行负载均衡的场景。

- **传输层（第4层）**：通过修改数据包的地址信息（IP+端口号）将流量转发到应用服务器；适合需要高效、低延迟的负载均衡，支持多种协议。
- **网络层（第3层）**：用于处理大规模网络流量的负载均衡。
- **数据链路层（第2层）**：适用于局域网中的负载均衡。
  选择在哪一层上做负载均衡，取决于具体的应用场景和需求。传输层负载均衡效率更高，但应用范围有限；应用层负载均衡功能更强，但资源损耗更多

39、DNS的实现细节

DNS解析的过程涉及多个步骤：

1. **用户查询**：用户在浏览器中输入一个域名，例如`www.example.com`。
2. **本地DNS缓存**：浏览器首先检查本地缓存。如果缓存中已有该域名的IP地址，则直接使用。
3. **递归查询**：
   - **本地DNS服务器**（也称为递归解析器）：如果本地缓存中没有结果，它会向更高级别的DNS服务器发起查询。
   - **根DNS服务器**：本地DNS服务器会向根DNS服务器查询，根DNS服务器提供指向顶级域DNS服务器的引用。
   - **顶级域DNS服务器**：根DNS服务器的回答将是顶级域DNS服务器的地址。顶级域DNS服务器知道对应的权威DNS服务器的地址。
   - **权威DNS服务器**：最终，本地DNS服务器向权威DNS服务器查询，权威DNS服务器提供请求域名的IP地址。
4. **结果返回**：本地DNS服务器将结果缓存一段时间（TTL），然后返回给用户的浏览器。

40、为什么要有多级dns服务器

提高网络的可靠性和灵活性
多级DNS服务器的主要作用包括提高解析效率、降低延迟、提供备份和容错机制。根DNS服务器知道顶级域名（如.com）的权威DNS服务器的位置，权威DNS服务器则知道具体域名的IP地址。这种分级结构使得查询效率更高，同时根服务器和其他各级服务器可以相互备份，提高网络的可靠性。