参考来源

一天吃透Redis面试八股文_牛客网 (nowcoder.com)

问题

什么是 Redis?

Redis（Remote Dictionary Server）是一种**高性能的键值对数据库**，它是用 C 语言编写的非关系型数据库。Redis 最大的特点是将**数据存储在内存**中，这使得其读写速度非常快，非常适合用作**数据缓存**。Redis 同时也支持将数据持久化到磁盘，保证数据安全。此外，Redis 的操作具有原子性，保证了操作的可靠性。

Redis 的优缺点

优点：

- **速度快**：由于基于内存操作，数据的读写速度非常快。
- **数据类型丰富**：支持多种数据类型，如 String、Hash、List、Set、ZSet 等。
- **数据持久化**：支持 RDB 和 AOF 两种持久化方式，有效防止数据丢失。
- **事务支持**：所有操作都是原子性的，支持多个操作的原子性执行。
- **主从复制**：支持数据的自动同步，实现读写分离。

缺点：

- **结构化查询限制**：对复杂的结构化查询支持不足,如选择查询、联结查询。
- **内存限制**：数据库容量受限于物理内存大小，不适合大规模数据的高性能读写。
- **扩容复杂性**：在线扩容相对复杂。

为什么 Redis 那么快?

- **内存存储**：数据全部存储在内存中，避免了磁盘 I/O 开销。
- **IO 多路复用**：采用 IO 多路复用技术如epoll，高效管理网络 I/O 操作。
- **优化的数据结构**：针对各种数据类型进行了底层优化，追求高速处理。

为什么 Redis 不作为主数据库，只用作缓存?

- **事务处理局限**：Redis 只能处理简单的事务，不适合复杂事务处理。
- **数据持久性问题**：作为内存数据库，Redis 在服务器故障时可能导致数据丢失。
- **数据结构简单**：相比关系型数据库，Redis 支持的数据结构较为简单。
- **安全性问题**：缺少像主数据库那样的复杂安全机制。

Redis的线程模型

- **基于 Reactor 模式**: Redis 的网络事件处理器是基于 Reactor 模式开发的，称为文件事件处理器。
- **四个主要组成**:

    1. **多个套接字**：监听多个网络连接。
    2. **IO多路复用程序**：允许同时监听多个套接字。
    3. **文件事件分派器**：根据套接字的任务分派事件。
    4. **事件处理器**：处理不同的网络事件。
- **单线程处理**：虽然采用了多路复用技术，但文件事件处理器的消费队列是单线程的，保持了操作的简单性和高效性。

Redis 应用场景

1. **作为缓存**：用于缓存热点数据，减轻数据库负担。
2. **计数器功能**：利用原子自增操作实现，如用户点赞、访问计数等。
3. **分布式锁**：使用 SETNX 命令或 RedLock 实现，保证分布式环境下的同步。
4. **消息队列**：通过发布/订阅模式或列表实现简单的消息队列。
5. **限速器**：控制用户访问接口的频率，如秒杀场景的访问限制。
6. **社交功能**：利用集合操作实现共同好友、兴趣等功能。

Memcached 与 Redis 的区别

- **数据结构**: Redis 支持多种数据类型，而 Memcached 数据结构单一。
- **数据持久化**: Redis 支持持久化，Memcached 不支持。
- **高可用性**: Redis 支持主从同步和集群模式，Memcached 不支持原生集群。
- **性能**: Redis 通常速度更快。
- **线程模型**: Redis 使用单线程多路 IO 复用模型，Memcached 使用多线程非阻塞 IO 模型。
- **Value大小限制**: Redis 最大 512MB，Memcached 仅 1MB。

为什么选择 Redis 而不是本地 Map/Guava 缓存?

- **生命周期**: 本地缓存如 map 或 guava 随 JVM 销毁而结束。
- **缓存一致性**: 在多实例环境下，本地缓存各自独立，无法保持一致性。而 Redis 等分布式缓存在多实例间共享，保持数据一致。
- **适用场景**: 分布式缓存适合大规模、分布式的应用环境，提供更强的数据共享和管理能力。

Redis 数据类型

基本数据类型：

1. **String**：可以存储字符串、数字或二进制数据，最大限制为512MB。
2. **Hash**：键值对集合，类似于 Java 的 HashMap。
3. **Set**：无序且唯一的集合，支持交集、并集等操作，适用于共同好友等功能。
4. **List**：有序可重复的集合，基于双向链表实现。
5. **SortedSet**：有序且唯一的集合，每个元素关联一个score，用于排行榜等场景。

特殊数据类型：

1. **Bitmap**：位图，以位为单位的数组，每位只能是0或1。
2. **Hyperloglog**：用于基数统计的算法，占用空间小。
3. **Geospatial**：存储地理位置信息，用于定位、附近的人等功能。

SortedSet 和 List 的异同点

相同点：

- 都是有序的。
- 可以获取指定范围内的元素。

不同点：

- **List** 基于链表实现，两端操作快，中间访问慢。
- **SortedSet** 基于散列表和跳表实现，中间元素访问时间复杂度为 O(logN)。
- **List** 不能简单调整元素位置；SortedSet 可以通过改变分数调整。
- **SortedSet** 相对更占内存。

Redis 内存用完时的行为

- 达到内存上限时，写命令会报错，读命令正常。
- 可配置内存淘汰机制，淘汰旧数据。

Redis 的内存优化策略

- 尽量使用集合类型（Hash, List, Set, SortedSet）来存储数据。
- 使用散列表来存储具有多个属性的对象，例如将用户的各个信息存储在一张散列表中。
- 散列表在存储少量数据时非常节省内存。

Keys 命令的问题和 Scan 命令

- **Keys 命令问题**：会阻塞 Redis 线程，影响性能。
- **Scan 命令**：渐进式遍历，避免长时间阻塞，但可能会有重复或遗漏的情况，不能保证完整遍历所有键。

Redis 事务

基本概念

Redis 事务提供了一种将多个命令打包然后连续执行的机制。在 Redis 中，事务的执行是连续的，无法被其他命令或客户端打断。

事务的生命周期

1. **开启事务**：
   - 使用 `MULTI` 命令开启一个事务。
2. **命令入队**：
   - 在事务开启后，执行的命令不会立即被执行，而是被放入一个队列中。
   - 此时，命令只是被排队，但并未实际执行。
3. **执行事务**：
   - 使用 `EXEC` 命令提交事务，之前入队的命令会被依次执行。

事务中的错误处理

- 在一个事务中，即使某个命令执行失败，其他命令仍然会被执行。
- 不保证原子性：如果事务中有命令执行失败，不会回滚其他已执行的命令。

示例

```
127.0.0.1:6379> multi
OK
127.0.0.1:6379> set a 1
QUEUED
127.0.0.1:6379> set b 1 2    # 错误的命令
QUEUED
127.0.0.1:6379> set c 3
QUEUED
127.0.0.1:6379> exec
1) OK
2) (error) ERR syntax error  # 命令执行失败
3) OK
```

WATCH 命令

- `WATCH` 命令用于监控一个或多个键，如果在执行事务前这些键被改变，事务将不会执行。
- 类似于乐观锁，用于处理并发场景。
- 执行 `EXEC` 后，所有的 `WATCH` 锁将自动取消。

示例

```
127.0.0.1:6379> watch name
OK
127.0.0.1:6379> set name 1
OK
127.0.0.1:6379> multi
OK
127.0.0.1:6379> set name 2
QUEUED
127.0.0.1:6379> set gender 1
QUEUED
127.0.0.1:6379> exec
(nil)  # 由于 name 被修改，事务未执行
127.0.0.1:6379> get gender
(nil)  # 事务中的命令没有执行
```

UNWATCH 命令

- 使用 `UNWATCH` 可以取消对所有键的监控。

你的笔记主要涉及 Redis 的两种持久化机制：RDB 和 AOF。我将对你的笔记进行整理，并对其中的一些信息进行补充。

持久化机制

持久化是将内存中的数据写入磁盘的过程，以防止服务宕机导致数据丢失。Redis 支持两种持久化方式：RDB 和 AOF，通常结合使用以达到最佳效果。

RDB 持久化

- **定义与原理**：RDB 是 Redis 的默认持久化方案，它会定期将内存中的数据存储到硬盘上的 `dump.rdb` 文件中。Redis 重启时会通过读取这个文件来恢复数据。
- **持久化过程**：
  1. 执行 `BGSAVE` 命令。
  2. 父进程判断是否有执行中的子进程，若有，则 `BGSAVE` 命令返回。
  3. 父进程通过 fork 操作创建子进程，并阻塞直至 fork 完成。
  4. 子进程将内存数据写入硬盘的临时文件，完成后替换旧的 RDB 文件。
- **触发方式**：
  - **手动触发**：使用 `SAVE` 或 `BGSAVE` 命令，其中 `SAVE` 会阻塞所有客户端请求，而 `BGSAVE` 允许后台异步操作。
  - **被动触发**：基于配置规则的自动快照（如 `SAVE 100 10`），全量复制时主节点自动执行 `BGSAVE`，以及默认的 `shutdown` 命令行为。
- **优缺点**：
  - 优点：快速加载数据，使用子进程避免主进程进行 IO 操作。
  - 缺点：无法实时持久化，频繁的 fork 操作成本高，且 RDB 文件格式可能因版本升级而不兼容。

AOF 持久化

- **定义与原理**：AOF（append only file）记录每次写命令，并在 Redis 重启时重新执行这些命令以恢复数据。AOF 解决了实时性的问题。
- **启用与配置**：默认关闭，可通过 `appendonly yes` 启用。使用 `appendfsync` 参数来设置同步策略，例如 `always`、`everysec` 或 `no`。
- **持久化过程**：
  1. 所有写命令追加到 AOF 缓冲区。
  2. 缓冲区根据策略同步到硬盘。
  3. 定期对 AOF 文件重写，压缩体积。
  4. 服务器重启时，通过加载 AOF 文件恢复数据。
- **优缺点**：
  - 优点：更好的数据保护，高性能的追加写入。
  - 缺点：AOF 文件比 RDB 大，数据恢复速度较慢。

RDB 和 AOF 的选择

- **数据非敏感**：如果数据可以从其他地方重新生成，且对数据的即时性要求不高，可以关闭持久化。
- **数据重要但可承受短暂丢失**：如缓存数据，使用 RDB 即可。RDB 对性能的影响较小，但数据恢复不是实时的。
- **内存数据持久化**：建议同时开启 RDB 和 AOF，以获得数据安全性和可靠性的平衡。
- **只使用 AOF**：优先考虑 `everysec` 配置，因为它平衡了可靠性和性能。
- **同时使用 RDB 和 AOF**：Redis 会优先使用 AOF 恢复数据，因为 AOF 提供了更完整的数据记录。

Redis 部署方案

- **单机版**：
  - 适用于小规模数据处理。
  - 问题：内存和处理能力有限，无法实现高可用。
- **主从模式**：
  - 主节点负责写，从节点负责读。
  - 适用于读取需求高的场景。
  - 问题：主节点故障需要手动处理，可用性不高。
- **哨兵模式**：
  - 解决主从模式的自动故障转移问题。
  - 适用于需要自动容错和灾难恢复的中等规模集群。
  - 问题：每个节点数据相同，可能存在内存浪费。
- **Redis Cluster**：
  - 采用分片技术，适用于海量数据、高并发、高可用场景。
  - 适用于数据量大、需求高并发的环境。
  - 所有主节点的容量总和代表可用的数据容量。

主从架构

- **作用**：支持读高并发，主要用于缓存场景。
- **原理**：一主多从，主节点负责写入，从节点负责读取。主节点将数据复制到从节点。
- **复制机制**：使用 PSYNC 命令进行数据同步。全量复制（初次连接或断连重连）和部分复制（断线后的增量数据同步）。

哨兵模式（Sentinel）

- **解决问题**：自动故障转移，提高可用性。
- **工作原理**：
  - Sentinel 实例定期检查主从节点的状态。
  - 当主节点主观下线后，经过多个 Sentinel 节点的确认，被标记为客观下线。
  - Sentinel 节点通过选举机制选出新的主节点，并通知其他节点更新配置。
- **客户端行为**：首先连接 Sentinel 确认主节点地址，然后连接主节点进行操作。

Redis Cluster

- **解决问题**：扩大写能力和容量，实现分布式存储。
- **配置**：至少6个节点（3主3从）。
- **数据分区**：基于虚拟槽（0-16383槽）的分区，使用 CRC16 算法进行键的分配。
- **节点通信**：使用两个端口号，一个用于客户端通信，另一个用于节点间通信（cluster bus）。
- **特点**：
  - 支持动态扩容和高可用。
  - 不保证强一致性，支持有限的事务操作。
  - 只能使用一个数据库空间（0号数据库）。

哈希分区算法

- **节点取余分区**：简单，适合翻倍扩容。
- **一致性哈希分区**：添加或删除节点时，只影响相邻节点，减少数据迁移。
- **虚拟槽分区**：Redis Cluster 使用，依据 CRC16 哈希值分配键到不同槽。

过期键的删除策略

1. **被动删除**：在访问某个 key 时，如果发现该 key 已经过期，则将其删除。
2. **主动删除**：Redis 定时清理过期键。每次会遍历所有数据库，从每个数据库中随机选出20个 key，若有过期则删除。如果其中超过5个 key 过期，则继续清理该数据库；否则，开始清理下一个数据库。
3. **内存不够时清理**：当使用的内存超过通过 `maxmemory` 参数设置的最大内存限制时，Redis 会根据配置的淘汰策略进行内存清理。

内存淘汰策略

- **Redis v4.0 之前的策略**：
  - `volatile-lru`：只对设置了过期时间的 key 使用 LRU 算法进行淘汰。
  - `allkeys-lru`：从所有 key 中使用 LRU 算法淘汰。
  - `volatile-ttl`：从设置了过期时间的 key 中挑选即将过期的 key 进行淘汰。
  - `volatile-random`：随机淘汰设置了过期时间的 key。
  - `allkeys-random`：从所有 key 中随机淘汰。
  - `no-eviction`：不进行任何淘汰，新写入操作会报错。
- **Redis v4.0 及之后增加的策略**：
  - `volatile-lfu`：从设置了过期时间的 key 中挑选最不常用的 key 进行淘汰。
  - `allkeys-lfu`：从所有 key 中挑选最不常用的 key 进行淘汰。
- 内存淘汰策略可以通过配置文件的 `maxmemory-policy` 选项来修改，默认配置为 `noeviction`。

缓存

1. **先删除缓存再更新数据库**：先删除缓存，然后更新数据库。问题是在这个过程中，如果有新的读请求，可能会读取并缓存旧数据，导致数据不一致。
2. **先更新数据库再删除缓存**：先更新数据库，然后删除缓存。这种方法在更新过程中读取的是缓存中的旧数据，但更新完成后可以恢复一致。
3. **异步更新缓存**：更新数据库后，将更新操作作为消息发送到消息队列，由 Redis 异步消费更新数据。这种方法可以保证操作的顺序一致性，但也存在延迟问题。

缓存穿透

- **问题描述**：查询不存在的数据，导致每次请求都需要访问数据库。
- **解决方案**：
  - 缓存空值：对查询不到的数据设置一个短暂的缓存。
  - 布隆过滤器：通过多个哈希函数将可能存在的数据映射到一个位数组中，拦截不存在的数据请求。

缓存雪崩

- **问题描述**：大量缓存同时过期，导致对数据库的请求急剧增加。
- **解决方案**：
  - 过期时间随机化：为缓存设置随机的过期时间，避免同时过期。
  - 加锁排队：使用锁或队列来控制对数据库的访问。
  - 二级缓存：设置一个备用的缓存层，如本地缓存。

缓存击穿

- **问题描述**：热点 key 过期，导致大量请求落到数据库。
- **解决方案**：
  - 加互斥锁：使用分布式锁控制对热点 key 的访问。
  - 热点数据永不过期：将热点数据设置为永不过期，并通过定时任务异步更新。

缓存预热

- **定义**：系统上线后预先加载热点数据到缓存中。
- **实现方式**：
  - 手动操作：通过特定的缓存刷新界面。
  - 自动加载：项目启动时自动预热缓存。
  - 定时任务：定期刷新缓存中的数据。

缓存降级

- **目的**：在高负载或服务故障时保持核心服务的可用性。
- **实现方式**：
  - 自动降级：基于关键性能指标自动降级。
  - 人工降级：根据预设的规则进行人工降级。
  - 默认值返回：Redis 故障时，直接返回默认值或固定内容。

Redis 实现消息队列

1. **使用 List 类型**
   - 生产消息：使用 `rpush`。
   - 消费消息：使用 `lpop`（阻塞模式可以使用 `blpop`）。
2. **使用 Pub/Sub 模式**
   - 实现一个生产者和多个消费者。
   - 缺点：消费者下线时，生产的消息会丢失。

例子：

假设有一个电商平台，需要处理用户的订单。当用户下单时，系统需要将订单信息发送到一个处理队列中，后台的处理服务（消费者）会从这个队列中获取订单并进行处理。

使用 List 类型实现消息队列

生产消息（生产者）

假设一个用户下了一个订单，订单 ID 是 `order123`。

```
RPUSH orders_queue "order123"
```

这里，`orders_queue` 是作为消息队列的 List，`"order123"` 是被推送到队列的订单 ID。

消费消息（消费者）

后台的订单处理服务会定期检查队列。

- **非阻塞消费**：

  ```
  LPOP orders_queue
  ```

  这个命令会从队列 `orders_queue` 的头部移除并返回一个订单 ID，该订单接下来会被处理。

- **阻塞消费**：

  ```
  BLPOP orders_queue 10
  ```

  如果队列为空，这个命令会阻塞最多 10 秒直到队列中有新的订单 ID 可以处理。

使用 Pub/Sub 模式实现消息队列

发布消息（生产者）

当用户下单时，系统可以发布一条消息到一个通道，比如 `new_order`。

```
PUBLISH new_order "order123"
```

订阅消息（消费者）

订单处理服务需要订阅 `new_order` 通道来接收新订单的通知。

```
SUBSCRIBE new_order
```

当 `new_order` 通道收到消息时，订阅它的服务就会收到通知，并可以开始处理该订单。

缺点

在 Pub/Sub 模式下，如果订单处理服务在消息发送时未在线，它将错过这些订单，因为消息不会在 Redis 中持久化。

总结

- 使用 List 类型实现的消息队列适合于需要确保所有消息都被处理的场景，如订单处理。
- 使用 Pub/Sub 模式实现的消息队列适合于实时通知，但不适合于需要保证消息持久化和可靠交付的场景。

Redis 实现延时队列

使用 Sorted Set

- 以时间戳为 score，消息内容为 key。
- 生产消息：使用 `zadd`。
- 消费消息：使用 `zrangebyscore` 指令获取一定时间之前的数据进行处理。

Pipeline 的作用

- **优化命令执行**

    - 批量请求和批量返回结果，提高执行速度。
- **注意事项**
    - 命令数量不宜过多，避免增加等待时间和网络阻塞。

LUA 脚本应用

- **原子性**
    - 脚本运行期间，不执行其他脚本或命令。
- **减少网络开销**
    - 将多条命令一次性打包执行。
- **应用场景**
    - 例如，限制接口访问频率，通过 Lua 脚本实现原子性操作。

接口限流的脚本实现

- **Lua 脚本例子**
    - 用于接口访问次数的统计和限制。
- **注意**
    - 此接口限流实现较简单，实际应用中更常用的是令牌桶算法或漏桶算法。

RedLock 机制

- **目的**：在 Redis 环境中实现更安全的分布式锁。
- **特性**：
    - 互斥访问：确保同一时间只有一个客户端持有锁。
    - 避免死锁：即使锁的持有者崩溃或失去连接，其他客户端也能获取锁。
    - 容错性：只要大多数 Redis 节点运行，锁就能正常工作。

处理 Redis 大 Key

- **定义**：
    - STRING 类型值超过 5MB。
    - 集合类型（ZSET、Hash、List、Set）成员超过 1W。
- **处理方法**：
    - 对于 STRING 类型：使用序列化、压缩或拆分。
    - 对于集合类型：进行数据分片。

Redis 性能问题及解决方案

1. Master最好不要做任何持久化工作，包括内存快照和AOF日志文件，特别是不要启用内存快照做持久化。
2. 如果数据比较关键，某个Slave开启AOF备份数据，策略为每秒同步一次。
3. 为了主从复制的速度和连接的稳定性，Slave和Master最好在同一个局域网内。
4. 尽量避免在压力较大的主库上增加从库

5. Master调用BGREWRITEAOF重写AOF文件，AOF在重写的时候会占大量的CPU和内存资源，导致服务load过高，出现短暂服务暂停现象。
6. 为了Master的稳定性，主从复制不要用图状结构，用单向链表结构更稳定，即主从关系为：Master<–Slave1<–Slave2<–Slave3…，这样的结构也方便解决单点故障问题，实现Slave对Master的替换，也即，如果Master挂了，可以立马启用Slave1做Master，其他不变。

Redis 过期键未释放内存的原因

- **覆盖导致过期时间改变**：由于操作错误，原有键的过期时间被更改。
- **惰性删除策略**：键实际过期但未被立即删除，只有在下次访问时才检查和删除。
- **定时删除策略**：每隔一定时间清理一部分过期键，可能导致某些过期键暂时未被删除。

Redis 性能突然变慢的原因

1. **存在 Big Key**：
   - 例子：假设你在 Redis 中存储了一个非常大的列表或哈希表，例如一个包含数百万个元素的列表。当你尝试删除或操作这个大键时，Redis 需要花费相当多的时间来处理这个操作，这期间可能会阻塞其他命令的执行，导致性能下降。
2. **内存上限设置（maxmemory）**：
   - 例子：如果你的 Redis 实例配置了内存上限，当存储的数据接近这个限制时，Redis 需要运行淘汰算法来腾出空间给新的写操作。这个过程会消耗额外的计算资源，从而影响性能。
3. **开启内存大页（Transparent Huge Pages, THP）**：
   - 例子：假如你在服务器上启用了内存大页功能，这可能会在 Redis 执行快照（RDB）或日志重写（AOF）操作时增加内存分配的耗时，因为大页内存的分配通常比标准页更慢。
4. **使用 Swap 空间**：
   - 例子：当你的 Redis 实例使用的内存超过了物理内存的限制，操作系统会开始将部分数据换出到磁盘上的 Swap 空间。由于磁盘的读写速度远低于内存，这会显著降低数据访问速度。
5. **网络带宽过载**：
   - 例子：如果 Redis 服务器与客户端之间的网络带宽被大量数据传输占满，例如大量的大键值传输，这可能导致网络延迟增加，进而影响 Redis 的响应时间。
6. **频繁短连接**：
   - 例子：在某些场景下，客户端可能频繁地创建与 Redis 的短暂连接，然后立即关闭。每个 TCP 连接的建立和释放都需要时间和资源，如果这种情况非常频繁，就会导致大量的资源被用于处理连接，而不是实际的数据操作。

Redis 集群最大槽数为 16384 的原因

1. **心跳包消息头占用空间考虑**：
   - 在 Redis 集群中，节点之间定期发送心跳包以检查彼此的状态。每个心跳包包含有关节点负责的槽位信息。如果槽位数量增加，比如到 65536 个，心跳包的大小也会增加，因此占用更多的网络带宽。选择 16384 个槽位是一个折衷，它保证了信息足够详细，同时不会过度占用网络资源。
2. **节点数量限制**：
   - 虽然理论上 Redis 集群可以支持多达 1000 个 master 节点，但实际上大多数集群的节点数量远少于此。每个节点都会负责一部分槽位。如果槽位太多，而节点数量相对较少，那么每个节点负责的槽位数就会很大，这可能在某些情况下导致不必要的复杂性和资源消耗。
3. **哈希槽压缩效率**：

- Redis 集群中，节点的哈希槽分布通常是以 bitmap 的形式保存的，即一个二进制位映射一个槽位。更少的槽位意味着这个 bitmap 更小，更容易处理和传输。此外，随着槽位数量的增加，bitmap 的压缩效率可能会降低。