面试真题

**【美团】讲一讲Redis的缓存三剑客的触发原因和解决办法？**

**Redis缓存穿透**

穿透不存在：指查询一个不存在的数据，缓存中没有相应的记录，每次请求都会去数据库查询，造成数据库负载激增。

解决：

* 使用布隆过滤器，过滤掉不存在的请求，避免直接访问数据库。
* 对查询结果进行缓存，即使是不存在的数据，也可以缓存一个标识，以减少对数据库的请求。

**Redis缓存击穿**

击穿热点：指某个热点数据在缓存中过期，导致大量请求同时访问数据库，造成数据库负载激增。

解决：

* 使用互斥锁，确保同一时间只有一个请求可以去数据库查询并更新缓存。
* 热点数据永不过期。

**Redis 缓存雪崩**

多个雪崩：指多个缓存数据在同一时间过期，导致大量请求同时访问数据库，造成数据库负载激增。

解决：

* 采用随机过期时间策略，避免多个数据同时过期。
* 使用双缓存策略，将数据同时存储在两层缓存中，减少数据库直接请求。

**【美团】Redis的淘汰策略？**

Redis 的内存淘汰策略一共有 8 种：

* **不淘汰**
* **淘汰**
  + **基于过期时间的淘汰策略**
  + **全部数据的淘汰策略**。

**不淘汰数据（默认）**：

* noeviction：当运行内存超过最大设置内存的时候，不会淘汰数据，而是直接返回报错禁止写入

**基于过期时间的淘汰**：

* volatile-random：随机淘汰掉设置了过期时间的 key
* volatile-lru（redis3.0之前默认策略）：淘汰掉所有设置了过期时间的，然后最久未使用的 key
* volatile-lfu（redis4.0后新增）：与上面类似，不过是淘汰掉最少使用的 key
* volatile-ttl：优先淘汰掉较早过期的 key

**全部数据的淘汰**：

* allkeys-random：随机淘汰掉任意的 key
* allkeys-lru：淘汰掉缓存中最久没有使用的 key
* allkeys-lfu（redis4.0后新增）：淘汰掉缓存中最少使用的 key

**【美团】为什么要用二级缓存？Redis如果是分布式的话你害怕过期的话，把过期时间设置的长一点不就可以了吗？**

1. 内存消耗：设置过长的过期时间会导致 Redis 中的数据占用过多内存，特别是那些不再需要的数据。虽然 Redis 会定期清理过期数据，但这些数据会在清理前继续占用内存，影响性能。
2. 不必要的数据存活：过长的过期时间可能导致不再需要的数据继续占用缓存，浪费资源。
3. 缓存一致性：在分布式系统中，不同 Redis 实例可能有不同的过期时间配置，长时间存活的缓存可能导致数据与数据库不一致，影响用户体验。

【美团】了解Redis分布式部署方法吗？/ Redis集群部署有哪些方式？

1. 主从复制：
   * 一主写 多从读，增加数据冗余、提升读取性能。
2. Redis集群：
   * 至少3主3从
   * 自动分片，每个节点只负责某一部分的数据，数据按照哈希槽（slot）分配，每个键被哈希到0到16383之间的某个槽。
   * 实现水平扩展与高可用性

【美团】了解主从集群搭建吗？

* + - 配置主节点：在Redis的配置文件中设置port（端口号）和bind（绑定地址），开启appendonly持久化。
    - 配置从节点：在从节点的配置文件中设置slaveof指令，使其指向主节点。
    - 启动Redis实例
    - 验证同步：通过INFO replication命令查看主从同步状态。
    - 故障转移：若主节点宕机，可以通过手动或使用Redis Sentinel来自动将某个从节点提升为新的主节点。

【美团】论坛项目里面的维护帖子的排行的Zset如果数据量很大应该怎么办，取数据花的时间比较长，怎么办？

数据量大：

1. 定期清理：通过定期删除过期数据，保持Zset的规模较小。
2. Redis集群：将数据分片存储在多个节点上，通过水平扩展提高处理能力。

取数据慢：

1. 分区存储：将帖子按月或按分类分配到不同的Zset中，每个Zset的数据集较小
2. 分页查询：只查询所需的数据部分，而非全量数据。
3. 内存优化：**使用ziplist压缩存储、**将数据定期**RDB/AOF**持久化到磁盘
4. 缓存机制：对于不常变化的排名数据，可以将查询结果缓存一段时间，只在数据变化时更新缓存。

【美团】Redis分布式锁的实现原理

* 加锁：
  + 第一个请求才能加锁：lock:{key}和SET NX
  + 过期时间：EX
* 解锁：
  + 锁的持有者才能删除锁：GET读取锁的值
  + 保证原子性：Lua脚本

【美团】在主从集群上使用setnx分布式锁，可能会有哪些问题，怎么解决？

* **问题1：主从延迟**导致多个客户端在不同节点上同时获取锁。
  + 使用SET命令带NX（如果不存在）和PX（设置过期时间）选项来实现分布式锁。该命令可以在一个原子操作中完成加锁和设置过期时间，避免多个客户端因异步复制问题而同时获得锁。
  + 使用唯一的标识符（如UUID）确保锁只被持有的客户端释放，防止误删除。
* **问题2：Redis**单点故障，锁无法释放，导致**死锁**。
  + **Redis Sentinel**在主节点宕机时自动切换主节点。
  + **Redis Cluster**通过分片机制和故障转移保证高可用。

面向简历

Redis线程模型

* 对于读写命令来说，Redis 一直是单线程模型。
* Redis 4.0 版本之后引入了多线程来执行一些大键值对的异步删除操作
* Redis 6.0 版本之后引入了多线程来处理网络请求（提高网络 IO 读写性能）。

**Redis 单线程模型了解吗？**

Redis 基于 Reactor 模式开发了自己的文件事件处理器（file event handler）。

* 文件事件处理器使用 I/O 多路复用（multiplexing）程序来同时监听多个套接字，并根据套接字目前执行的任务来为套接字关联不同的事件处理器。
* 当被监听的套接字准备好执行连接应答（accept）、读取（read）、写入（write）、关闭（close）等操作时，与操作相对应的文件事件就会产生，这时文件事件处理器就会调用套接字之前关联好的事件处理器来处理这些事件。

**虽然文件事件处理器以单线程方式运行，但通过使用 I/O 多路复用程序来监听多个套接字**，文件事件处理器既实现了高性能的网络通信模型，又可以很好地与 Redis 服务器中其他同样以单线程方式运行的模块进行对接，这保持了 Redis 内部单线程设计的简单性。

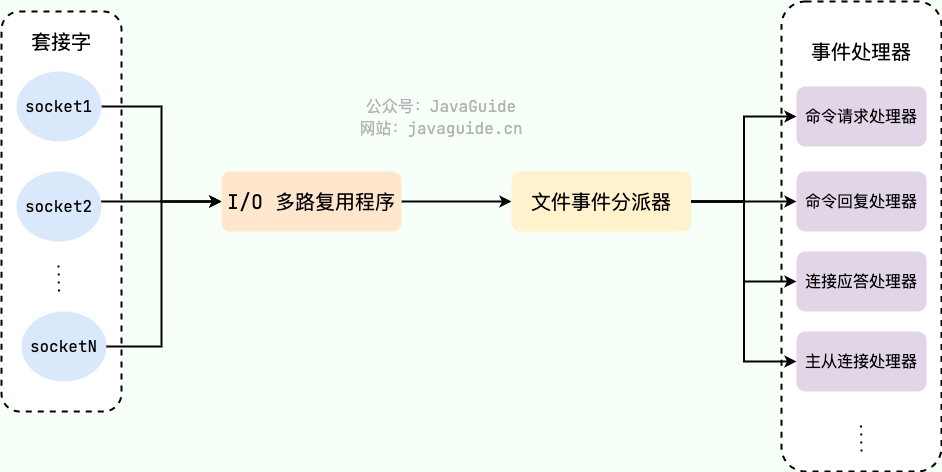
**既然是单线程，那怎么监听大量的客户端连接呢？**

Redis 通过 **IO 多路复用程序** 来监听来自客户端的大量连接（或者说是监听多个 socket），它会将感兴趣的事件及类型（读、写）注册到内核中并监听每个事件是否发生。

这样的好处非常明显：**I/O 多路复用技术的使用让 Redis 不需要额外创建多余的线程来监听客户端的大量连接，降低了资源的消耗**。

文件事件处理器（file event handler）主要是包含 4 个部分：

* 多个 socket（客户端连接）
* IO 多路复用程序（支持多个客户端连接的关键）
* 文件事件分派器（将 socket 关联到相应的事件处理器）
* 事件处理器（连接应答处理器、命令请求处理器、命令回复处理器）



**Redis6.0 之前为什么不使用多线程？**

虽然说 Redis 是单线程模型，但实际上，**Redis 在 4.0 之后的版本中就已经加入了对多线程的支持。**

不过，Redis 4.0 增加的多线程主要是针对一些大键值对的删除操作的命令，使用这些命令就会使用主线程之外的其他线程来“异步处理”，从而减少对主线程的影响。

为此，Redis 4.0 之后新增了几个异步命令：

* UNLINK：可以看作是 DEL 命令的异步版本。
* FLUSHDB ASYNC：用于清空当前 SELECT 数据库中的所有键。
* FLUSHALL ASYNC：用于清空所有数据库的所有键，不限于当前 SELECT 的数据库。

总的来说，直到 Redis 6.0 之前，Redis 的主要操作仍然是单线程处理的。

**那 Redis6.0 之前为什么不使用多线程？** 我觉得主要原因有 3 点：

* 单线程编程容易并且更容易维护；
* Redis 的性能瓶颈不在 CPU ，主要在内存和网络；
* 多线程就会存在死锁、线程上下文切换等问题，甚至会影响性能。

**Redis6.0 之后为何引入了多线程？**

**Redis6.0 引入多线程主要是为了提高网络 IO 读写性能**，因为这个算是 Redis 中的一个性能瓶颈（Redis 的瓶颈主要受限于内存和网络）。

虽然，Redis6.0 引入了多线程，但是 Redis 的多线程只是在网络数据的读写这类耗时操作上使用了，执行命令仍然是单线程顺序执行。因此，你也不需要担心线程安全问题。

Redis6.0 的多线程默认是禁用的，只使用主线程。如需开启需要设置 IO 线程数 > 1，需要修改 redis 配置文件 redis.conf：

io-threads 4 #设置1的话只会开启主线程，官网建议4核的机器建议设置为2或3个线程，8核的建议设置为6个线程

另外：

* io-threads 的个数一旦设置，不能通过 config 动态设置。
* 当设置 ssl 后，io-threads 将不工作。

开启多线程后，默认只会使用多线程进行 IO 写入 writes，即发送数据给客户端，如果需要开启多线程 IO 读取 reads，同样需要修改 redis 配置文件 redis.conf :

io-threads-do-reads yes

但是官网描述开启多线程读并不能有太大提升，因此一般情况下并不建议开启

**Redis 后台线程了解吗？**

我们虽然经常说 Redis 是单线程模型（主要逻辑是单线程完成的），但实际还有一些后台线程用于执行一些比较耗时的操作：

* 通过 bio\_close\_file 后台线程来释放 AOF / RDB 等过程中产生的临时文件资源。
* 通过 bio\_aof\_fsync 后台线程调用 fsync 函数将系统内核缓冲区还未同步到磁盘的数据强制刷到磁盘（ AOF 文件）。
* 通过 bio\_lazy\_free后台线程释放大对象（已删除）占用的内存空间.

Redis 的持久化机制有哪些？

**RDB（Redis Database Backup）快照**：

* RDB 是通过生成某一时刻的数据快照来实现持久化的，可以在特定时间间隔内保存数据的快照。
* 适合灾难恢复和备份，能生成紧凑的二进制文件，但可能会在崩溃时丢失最后一次快照之后的数据。

**AOF（Append Only File）日志**：

* AOF 通过将每个写操作追加到日志文件中实现持久化，支持将所有写操作记录下来以便恢复。
* 数据恢复更为精确，但文件体积较大，重写时可能会消耗更多资源。

Redis 4.0 新增了 RDB 和 AOF 的混合持久化机制。

Redis 缓存穿透、击穿、雪崩解决方案

**Redis缓存穿透**

穿透不存在：指查询一个不存在的数据，缓存中没有相应的记录，每次请求都会去数据库查询，造成数据库负载激增。

解决：

* 使用布隆过滤器，过滤掉不存在的请求，避免直接访问数据库。
* 对查询结果进行缓存，即使是不存在的数据，也可以缓存一个标识，以减少对数据库的请求。

**Redis缓存击穿**

击穿热点：指某个热点数据在缓存中过期，导致大量请求同时访问数据库，造成数据库负载激增。

解决：

* 使用互斥锁，确保同一时间只有一个请求可以去数据库查询并更新缓存。
* 热点数据永不过期。

**Redis 缓存雪崩**

多个雪崩：指多个缓存数据在同一时间过期，导致大量请求同时访问数据库，造成数据库负载激增。

解决：

* 采用随机过期时间策略，避免多个数据同时过期。
* 使用双缓存策略，将数据同时存储在两层缓存中，减少数据库直接请求。

知识框架

Redis 基础

Redis 通常应用于哪些场景？（简单）

Redis 和 Memcached 有哪些区别？（中等）

Redis 中常见的数据类型有哪些？（简单）

Redis 的 hash 是什么？（中等）

Redis 中跳表的实现原理是什么？（困难）

Redis 的 Lua 脚本功能是什么？如何使用？（中等）

Redis的数据结构有哪些，怎么实现的

字哈列集序位H地

字符串（String）： Simple Dynamic Strings实现，允许动态地扩展缩小长度

哈希表（Hash）：通过哈希值将数据分布到不同的槽位上，以提高查找效率。

列表（List）：双向链表结构实现。在两端执行插入和删除操作的效率高。

集合（Set）：哈希表结构实现，支持高效的添加、删除和判断元素是否存在的操作。

有序集合（Sorted Set）：跳表和哈希表结合。跳表提供有序性，哈希表提供快速查找。

位图（Bitmap）

HyperLogLog：用于估计基数（集合中不重复元素的数量）

地理空间索引（Geospatial Index）： ZSET的分数表示地理位置的坐标。

Redis为什么快？

内单非数持协

内存存储： Redis主要将数据存储在内存中，内存的读写速度远高于磁盘

单线程模型： Redis采用单线程模型，通过事件循环来处理多个客户端请求。避免了多线程之间的锁竞争，减少了上下文切换的开销。

非阻塞式 I/O： Redis使用非阻塞式的I/O操作，使得在进行网络通信和文件操作时，能够更好地利用系统资源，提高吞吐量。

高效的数据结构

持久化选项： Redis支持多种持久化选项，包括快照和追加式文件（AOF）。

优化的网络协议： Redis使用自定义的协议进行客户端和服务器之间的通信。这种协议是文本协议，易于理解和调试。同时，它也是二进制安全的，支持存储和传输任意类型的数据。

详细讲一讲IO多路复用

注册监听通知操作

Select poll epoll kqueue

1.定义：

IO多路复用它允许单一的线程同时监听多个IO通道上的事件。通过这种方式，一个线程能够有效地管理多个IO通道，从而减少线程的数量，提高系统的性能和资源利用率。如果一个IO通道上的IO操作是阻塞的，可能会影响整个IO多路复用的性能，因此通常建议使用非阻塞IO。

2.常见的IO多路复用机制：

select：通过select系统调用，允许程序监视一组文件描述符，一旦其中一个文件描述符准备好进行IO操作，就通知程序进行相应的读写操作。

poll：与select类似，但采用链表，避免了select中的文件描述符数量限制。

epoll：是Linux下效率最高的IO多路复用机制，使用红黑树实现高效的事件查找。epoll的性能随着文件描述符的增加而近乎线性增长，适用于高并发的场景。

kqueue：是BSD和macOS中的IO多路复用机制，与epoll类似，通过红黑树实现，具有高性能和可扩展性。

3. 工作流程：

程序通过IO多路复用调用（如select、epoll）将一组文件描述符注册到内核中。

内核监视这些文件描述符上的事件，当有IO事件发生时，通知应用程序。

程序通过遍历通知，找到准备好的文件描述符，进行相应的IO操作。

这样，一个线程就能够同时管理多个IO通道，而不需要为每个IO通道创建一个独立的线程。

Redis 消息队列

消息队列的选型，Kafka和RabbitMQ的区别，各自的应用场景

Kafka更稳定大规模

RabbitMQ更灵活易用

Kafka:

设计理念：

Kafka设计为分布式、高可用、持久化的消息系统。它的主要目标是提供高吞吐量、低延迟和容错性。

数据模型：

Kafka采用发布-订阅的模型，消息以主题（Topic）为单位组织。消息存储在分布式日志中，允许多个消费者并行消费。

可靠性：

Kafka强调消息的持久性，一旦消息被写入，就会被持久存储。它支持多副本复制，确保数据可靠性。

性能：

Kafka在高吞吐量和低延迟方面表现出色，特别适合大规模数据流处理。

适用场景：

日志收集与分析、事件溯源、大数据处理、实时数据流处理等。

RabbitMQ:

设计理念：

RabbitMQ设计为一个灵活、易用的消息队列系统。它提供多种消息传递模式，包括点对点和发布-订阅。

数据模型：

RabbitMQ支持多种消息传递模式，包括点对点、发布-订阅和路由。消息存储在队列中，每个队列有一个或多个消费者。

可靠性：

RabbitMQ提供可靠性传递，支持持久化消息。它可以通过不同的交换机类型和队列设置来实现不同的消息路由策略。

性能：

RabbitMQ在一般情况下表现良好，但相对于Kafka，它在处理大量数据和高吞吐量方面可能稍显逊色。

适用场景：

任务队列、点对点通信、发布-订阅、应用解耦等。

如何选择：

数据处理特点：

如果需要处理大规模的实时数据流、日志等，Kafka可能是更合适的选择。

如果强调消息传递的可靠性和灵活性，以及更广泛的消息模型，RabbitMQ可能更适用。

复杂性和易用性：

RabbitMQ通常被认为更易于入门和使用，适用于相对简单的消息场景。

可靠性需求：

如果对消息的持久性和数据不丢失有较高要求，Kafka可能更符合需求。

Redis 线程模型

* 对于读写命令来说，Redis 一直是单线程模型。
* Redis 4.0 版本之后引入了多线程来执行一些大键值对的异步删除操作
* Redis 6.0 版本之后引入了多线程来处理网络请求（提高网络 IO 读写性能）。

Redis 单线程模型了解吗？

Redis 基于 Reactor 模式开发了自己的文件事件处理器（file event handler）。

* 文件事件处理器使用 I/O 多路复用（multiplexing）程序来同时监听多个套接字，并根据套接字目前执行的任务来为套接字关联不同的事件处理器。
* 当被监听的套接字准备好执行连接应答（accept）、读取（read）、写入（write）、关闭（close）等操作时，与操作相对应的文件事件就会产生，这时文件事件处理器就会调用套接字之前关联好的事件处理器来处理这些事件。

**虽然文件事件处理器以单线程方式运行，但通过使用 I/O 多路复用程序来监听多个套接字**，文件事件处理器既实现了高性能的网络通信模型，又可以很好地与 Redis 服务器中其他同样以单线程方式运行的模块进行对接，这保持了 Redis 内部单线程设计的简单性。

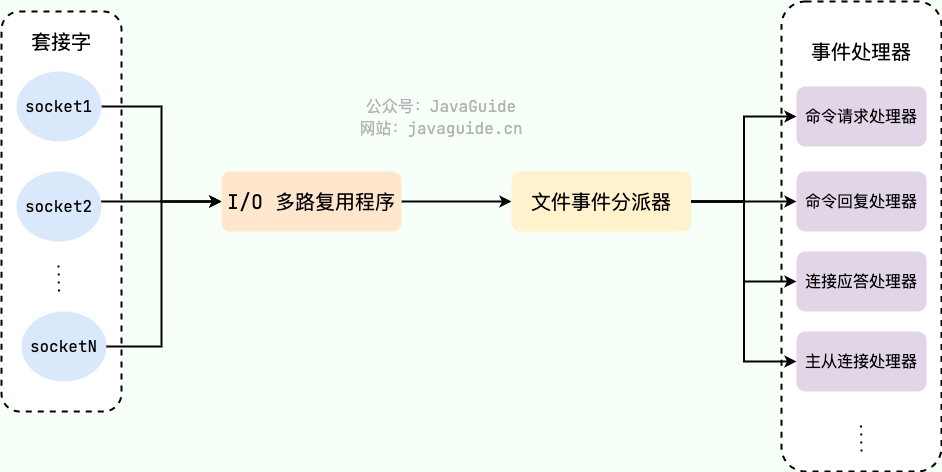
既然是单线程，那怎么监听大量的客户端连接呢？

Redis 通过 **IO 多路复用程序** 来监听来自客户端的大量连接（或者说是监听多个 socket），它会将感兴趣的事件及类型（读、写）注册到内核中并监听每个事件是否发生。

这样的好处非常明显：**I/O 多路复用技术的使用让 Redis 不需要额外创建多余的线程来监听客户端的大量连接，降低了资源的消耗**。

文件事件处理器（file event handler）主要是包含 4 个部分：

* 多个 socket（客户端连接）
* IO 多路复用程序（支持多个客户端连接的关键）
* 文件事件分派器（将 socket 关联到相应的事件处理器）
* 事件处理器（连接应答处理器、命令请求处理器、命令回复处理器）



Redis6.0 之前为什么不使用多线程？

虽然说 Redis 是单线程模型，但实际上，**Redis 在 4.0 之后的版本中就已经加入了对多线程的支持。**

不过，Redis 4.0 增加的多线程主要是针对一些大键值对的删除操作的命令，使用这些命令就会使用主线程之外的其他线程来“异步处理”，从而减少对主线程的影响。

为此，Redis 4.0 之后新增了几个异步命令：

* UNLINK：可以看作是 DEL 命令的异步版本。
* FLUSHDB ASYNC：用于清空当前 SELECT 数据库中的所有键。
* FLUSHALL ASYNC：用于清空所有数据库的所有键，不限于当前 SELECT 的数据库。

总的来说，直到 Redis 6.0 之前，Redis 的主要操作仍然是单线程处理的。

**那 Redis6.0 之前为什么不使用多线程？** 我觉得主要原因有 3 点：

* 单线程编程容易并且更容易维护；
* Redis 的性能瓶颈不在 CPU ，主要在内存和网络；
* 多线程就会存在死锁、线程上下文切换等问题，甚至会影响性能。

Redis6.0 之后为何引入了多线程？

**Redis6.0 引入多线程主要是为了提高网络 IO 读写性能**，因为这个算是 Redis 中的一个性能瓶颈（Redis 的瓶颈主要受限于内存和网络）。

虽然，Redis6.0 引入了多线程，但是 Redis 的多线程只是在网络数据的读写这类耗时操作上使用了，执行命令仍然是单线程顺序执行。因此，你也不需要担心线程安全问题。

Redis6.0 的多线程默认是禁用的，只使用主线程。如需开启需要设置 IO 线程数 > 1，需要修改 redis 配置文件 redis.conf：

io-threads 4 #设置1的话只会开启主线程，官网建议4核的机器建议设置为2或3个线程，8核的建议设置为6个线程

另外：

* io-threads 的个数一旦设置，不能通过 config 动态设置。
* 当设置 ssl 后，io-threads 将不工作。

开启多线程后，默认只会使用多线程进行 IO 写入 writes，即发送数据给客户端，如果需要开启多线程 IO 读取 reads，同样需要修改 redis 配置文件 redis.conf :

io-threads-do-reads yes

但是官网描述开启多线程读并不能有太大提升，因此一般情况下并不建议开启

Redis 后台线程了解吗？

我们虽然经常说 Redis 是单线程模型（主要逻辑是单线程完成的），但实际还有一些后台线程用于执行一些比较耗时的操作：

* 通过 bio\_close\_file 后台线程来释放 AOF / RDB 等过程中产生的临时文件资源。
* 通过 bio\_aof\_fsync 后台线程调用 fsync 函数将系统内核缓冲区还未同步到磁盘的数据强制刷到磁盘（ AOF 文件）。
* 通过 bio\_lazy\_free后台线程释放大对象（已删除）占用的内存空间.

Redis 持久化

Redis 的持久化机制有哪些？

**RDB（Redis Database Backup）快照**：

* RDB 是通过生成某一时刻的数据快照来实现持久化的，可以在特定时间间隔内保存数据的快照。
* 适合灾难恢复和备份，能生成紧凑的二进制文件，但可能会在崩溃时丢失最后一次快照之后的数据。

**AOF（Append Only File）日志**：

* AOF 通过将每个写操作追加到日志文件中实现持久化，支持将所有写操作记录下来以便恢复。
* 数据恢复更为精确，但文件体积较大，重写时可能会消耗更多资源。

Redis 4.0 新增了 RDB 和 AOF 的混合持久化机制。

Redis 在生成 RDB 文件时如何处理请求？

在 Redis 生成 RDB 文件时是异步的（使用 bgsave 命令），**采用了 fork 子进程的方式来进行快照操作**。生成 RDB 文件的过程由子进程执行，主进程继续处理客户端请求，所以可以保证 Redis 在生成快照的过程中依然对外提供服务，不会影响正常请求。

Redis 数据过期后的删除策略是什么？

* **定期删除**：Redis 每隔一定时间（默认是 100 毫秒）会随机检查一定数量的键，如果发现过期键，则将其删除。这种方式能够在后台持续清理过期数据，防止内存膨胀。
* **惰性删除**：在每次访问键时，Redis 检查该键是否已过期，如果已过期，则将其删除。这种策略保证了在使用过程中只删除不再需要的数据，但在不访问过期键时不会立即清除。

Redis 中有哪些内存淘汰策略？

Redis 的内存淘汰策略一共有 8 种：

* **不开启数据淘汰**
* **开启数据淘汰**
  + **基于过期时间的淘汰策略**
  + **全部数据的淘汰策略**。

**不淘汰数据（默认）**：

* noeviction：当运行内存超过最大设置内存的时候，不会淘汰数据，而是直接返回报错禁止写入

**基于过期时间的淘汰**：

* volatile-random：随机淘汰掉设置了过期时间的 key
* volatile-lru（redis3.0之前默认策略）：淘汰掉所有设置了过期时间的，然后最久未使用的 key
* volatile-lfu（redis4.0后新增）：与上面类似，不过是淘汰掉最少使用的 key
* volatile-ttl：优先淘汰掉较早过期的 key

**全部数据的淘汰**：

* allkeys-random：随机淘汰掉任意的 key
* allkeys-lru：淘汰掉缓存中最久没有使用的 key
* allkeys-lfu（redis4.0后新增）：淘汰掉缓存中最少使用的 key

Redis 缓存

讲一讲Redis的缓存三剑客的各种触发原因和解决办法？

Redis缓存穿透

穿透不存在：指查询一个不存在的数据，缓存中没有相应的记录，每次请求都会去数据库查询，造成数据库负载激增。

解决：

* 使用布隆过滤器，过滤掉不存在的请求，避免直接访问数据库。
* 对查询结果进行缓存，即使是不存在的数据，也可以缓存一个标识，以减少对数据库的请求。

Redis缓存击穿

击穿热点：指某个热点数据在缓存中过期，导致大量请求同时访问数据库，造成数据库负载激增。

解决：

* 使用互斥锁，确保同一时间只有一个请求可以去数据库查询并更新缓存。
* 热点数据永不过期。

Redis 缓存雪崩

多个雪崩：指多个缓存数据在同一时间过期，导致大量请求同时访问数据库，造成数据库负载激增。

解决：

* 采用随机过期时间策略，避免多个数据同时过期。
* 使用双缓存策略，将数据同时存储在两层缓存中，减少数据库直接请求。

Redis的淘汰策略？

Redis 的内存淘汰策略一共有 8 种：

* **不开启数据淘汰**
* **开启数据淘汰**
  + **基于过期时间的淘汰策略**
  + **全部数据的淘汰策略**。

**不淘汰数据（默认）**：

* noeviction：当运行内存超过最大设置内存的时候，不会淘汰数据，而是直接返回报错禁止写入

**基于过期时间的淘汰**：

* volatile-random：随机淘汰掉设置了过期时间的 key
* volatile-lru（redis3.0之前默认策略）：淘汰掉所有设置了过期时间的，然后最久未使用的 key
* volatile-lfu（redis4.0后新增）：与上面类似，不过是淘汰掉最少使用的 key
* volatile-ttl：优先淘汰掉较早过期的 key

**全部数据的淘汰**：

* allkeys-random：随机淘汰掉任意的 key
* allkeys-lru：淘汰掉缓存中最久没有使用的 key
* allkeys-lfu（redis4.0后新增）：淘汰掉缓存中最少使用的 key

为什么要用二级缓存？Redis如果是分布式的话你害怕过期的话，把过期时间设置的长一点不就可以了吗？

内存消耗、过期数据无法即时清理、不必要的数据存活：如果将过期时间设置得过长，那么数据会在 Redis 中存活更长时间，这会导致 Redis 中存储的数据量过大，增加内存压力。长时间存在的数据有可能并不是当前请求所需要的数据，从而浪费内存资源。

Redis 会定期进行过期键的清理工作（如使用惰性删除和定期删除两种方式），但这些过期数据并不会立刻被清理。如果你设置过长的过期时间，过期的数据会继续占用内存，直到 Redis 自动进行清理，可能会影响性能，尤其是在内存较小的 Redis 实例中。

对于很多应用场景来说，数据的有效性通常有一定的生命周期，如果只是通过设置较长的过期时间来解决问题，可能会导致一些不再需要的数据仍然占用缓存资源，而这部分数据可能早已不再有意义。

缓存一致性：在分布式系统中，多个 Redis 实例可能会有不同的过期时间配置。如果你的数据在多个节点之间共享，长过期时间可能会导致一致性问题，尤其是在数据更新频繁的场景中。长时间的缓存存活会导致缓存与数据库之间的数据不一致，进而影响最终用户的体验。

加了过期时间的LRU缓存，缓存里的内容x秒后会过期，如何实现

实现带有过期时间的LRU（Least Recently Used）缓存通常需要结合两个主要概念：LRU算法和定时过期。以下是一个简单的Python示例，使用collections.OrderedDict来实现LRU缓存，并通过定时任务来处理过期时间：

这个示例中，LRUCacheWithExpiration类继承自OrderedDict，通过使用OrderedDict的有序字典来实现LRU缓存，同时通过定时任务检查过期时间并清理过期的缓存项。注意，这里使用了一个简单的线程来定期清理过期项，你可能需要根据实际需求进行调整，例如使用定时器、协程等机制。

Redis 性能与优化

为什么 Redis 设计为单线程？6.0 版本为何引入多线程？（中等）

Redis 的 Pipeline 功能是什么？（中等）

Redis 中的 Big Key 问题是什么？如何解决？（中等）

如何解决 Redis 中的热点 key 问题？（中等）

Redis 集群（Cluster）

了解Redis分布式的部署方法吗？了解主从集群搭建吗？

Redis集群部署有哪些方式？

Redis 主从复制的实现原理是什么？

Redis 的主从复制是指一个 Redis 实例（主节点）可以将数据复制到一个或多个从节点（从节点），从节点从主节点获取数据并保持同步。

**复制流程**：

* **连接**：从节点通过向主节点发送 PSYNC 命令建立连接。
* **全量复制**：如果是第一次连接或之前的连接失效，从节点会请求全量复制，主节点将当前数据快照（RDB文件）发送给从节点。
* **增量复制**：全量复制完毕后，主从之间会保持一个长连接，主节点会通过这个连接将后续的写操作传递给从节点执行，来保证数据的一致。

Redis 集群的实现原理是什么？

简单来说，集群就是通过多台机器分担单台机器上的压力。

Redis 集群（Redis cluster）是通过多个 Redis 实例组成的，每个实例存储部分的数据（即每个实例之间的数据是不重复的）。

具体是采用哈希槽（Hash Slot）机制来分配数据，将整个键空间划分为 16384 个槽（slots）。每个 Redis 实例负责一定范围的哈希槽，数据的 key 经过哈希函数计算后对 16384 取余即可定位到对应的节点。

客户端在发送请求时，会通过集群的任意节点进行连接，如果该节点存储了对应的数据则直接返回，反之该节点会根据请求的键值计算哈希槽并路由到正确的节点。

Redis 集群会出现脑裂问题吗？

Redis 集群存在脑裂问题风险，特别是在**网络分区**的情况下，可能会导致同一集群内出现多个主节点，导致数据不一致。

在 Redis 集群中，如何根据键定位到对应的节点？

Redis 集群将数据分布到 16384 个哈希槽（slots） 中，每个键通过CRC16 哈希函数计算键的哈希值，然后对 16384 取模，得到哈希槽编号（范围是 0 到 16383）。

为什么需要 Redis Cluster？解决了什么问题？有什么优势？

Redis Cluster 是如何分片的？

为什么 Redis Cluster 的哈希槽是 16384 个?

如何确定给定 key 的应该分布到哪个哈希槽中？

Redis Cluster 支持重新分配哈希槽吗？

Redis Cluster 扩容缩容期间可以提供服务吗？

Redis Cluster 中的节点是怎么进行通信的？

Redis 哨兵（Sentinel）

Redis 的哨兵机制是什么？

**Redis 的哨兵机制（Sentinel）** 是一种高可用解决方案

主要功能包括：

* **监控**：哨兵不断监控 Redis 主节点和从节点的运行状态，定期发送 PING 请求检查节点是否正常。
* **自动故障转移**：当主节点发生故障时，哨兵会选举一个从节点提升为新的主节点，并通知客户端更新主节点的地址，从而实现高可用。
* **通知**：哨兵可以向系统管理员或其他服务发送通知，以便快速处理 Redis 实例的状态变化。

什么是 Sentinel？ 有什么用？

Sentinel 如何检测节点是否下线？主观下线与客观下线的区别?

Sentinel 是如何实现故障转移的？

为什么建议部署多个 sentinel 节点（哨兵集群）？

Sentinel 如何选择出新的 master（选举机制）?

如何从 Sentinel 集群中选择出 Leader ？

Sentinel 可以防止脑裂吗？

Redis 事务与分布式锁

论坛项目里面的维护帖子的排行的Zset如果数据量很大应该怎么办，取数据花的时间比较长，怎么办？

Redis分布式锁的实现原理

在主从集群上使用setnx分布式锁，可能会有哪些问题，怎么解决？

Redis 支持事务吗？如何实现？