Redis面试突击串讲

郭嘉

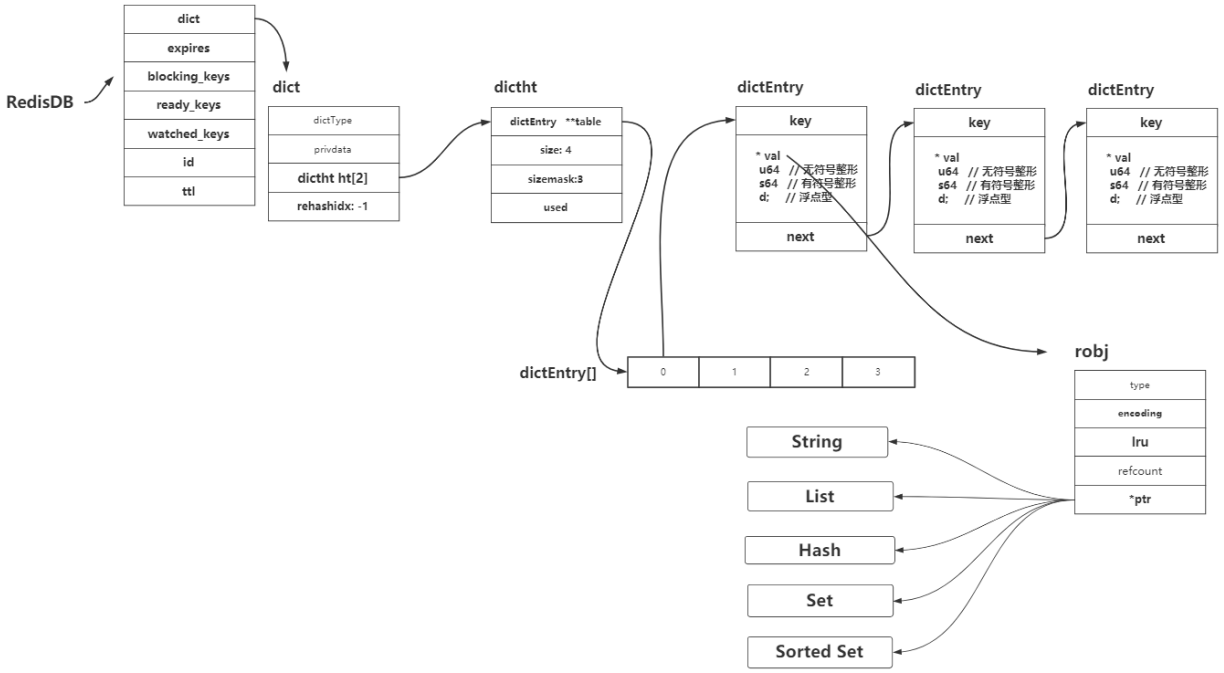
什么是Redis

Redis(Remote Dictionary Server) 是一个使用 C 语言编写的，开源的高性能非关系型（NoSQL）的**键值对**数据库。

与传统数据库不同的是 Redis 的数据是存在**内存**中的，所以读写速度非常快，因此 redis 被广泛应用于缓存方向，每秒可以处理超过 10万次读写操作。

Redis 是K-V型的数据库，整个数据库都是用字典来存储的，对Redis数据库的任何增删改查操作，实际上就是对字典中的数据进行增删改查

1. 可以存储海量数据，且可以根据键以**O(1)** 的时间复杂度取出或插入关联值
2. 键值对中键的类型可以是字符串，整型，浮点型等，且**键是唯一**的.
3. 键值对中的值类型可以是**string, hash ,list, set, sorted set**.



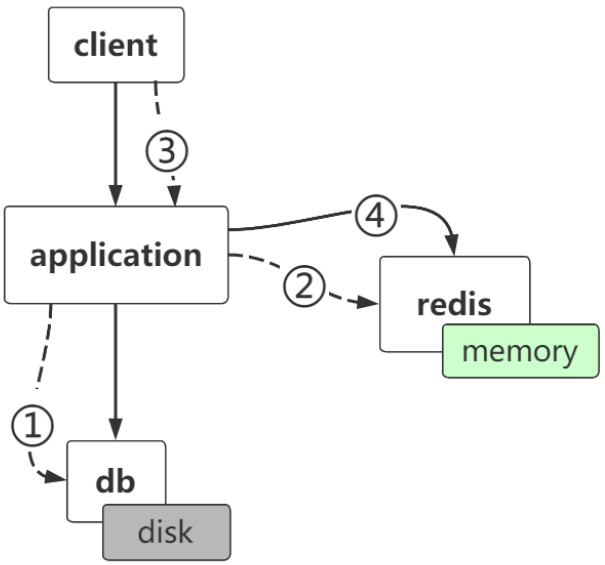
为什么要用 Redis /为什么要用缓存

假如用户

第一次访问数据库中的某些数据。这个过程会比较慢，因为是从硬盘上读取的，硬盘的寻址速度是 毫秒级的。从数据库中获取到数据后将数据存在Redis中，这样下一次再访问这些数据的时候就可以直接从Redis中获取了,Redis数据是存在内存中的，内存的寻址速度是纳秒级的，所以可以极大**提升响应速度**，同时**缓解数据库**压力 。

时间单位

```
1s=1000ms
1ms=1000us
1us=1000ns
```



为什么要用 Redis 而不用 map/guava 做缓存?

缓存分为**本地缓存**和**分布式缓存**。以 Java 为例，使用自带的 map 或者 guava 实现的是本地缓存，最主要的特点是轻量以及快速，生命周期随着 jvm 的销毁而结束，并且在多实例的情况下，每个实例都需要各自保存一份缓存,**造成内存浪费**，且**缓存不具有一致性**。

使用 redis 或 memcached 之类的称为**分布式缓存**，在多实例的情况下，各实例共用一份缓存数据，缓存具有一致性。

Redis 于 Memcached 的区别与选型

	是否基于内存	值类型	数据持久	过期策略	性能	虚拟内存支持
Redis	是	**string/set/hash/list/zset...**	**有**	有	高	**是**

Memcached	是	**string**	**无**	有	高	**否**

Redis是单线程的，为什么这么快

1、完全基于**内存(ns 级的访问)**，非常快速。

2、Redis 中的**数据结构**是采用类似于java中HashMap的数据结构 **Dict**，底层用数组加链表实现的哈希表，可以实现查找和操作**O(1)**时间复杂度；

3、采用**单线程**，避免了**不必要的上下文切换和竞争条件**，也不存在多进程或者多线程导致的切换而消耗 CPU，不用去考虑各种锁的问题，不存在加锁释放锁操作，没有因为可能出现死锁而导致的性能消耗；

4、使用**多路 I/O 复用模型，非阻塞 IO**；

Redis有哪些数据类型及应用场景

数据类型	可以存储的值	操作	应用场景
STRING	字符串、整数或者浮点数	1. 对整个字符串或者字符串的其中一部分执行操作 2.对整数和浮点数执行自增或者自减操作 3. 位运算	1.做简单的键值对缓存 2.计数器/分布式ID生成 3.位运算做海量数据统计（日活/月活）
LIST	列表	1.从两端压入或者弹出元素 2.对单个或者多个元素进行修剪，只保留一个范围内的元素	1. 分布式栈 2. 分布式队列/阻塞队列 3. 存储一些列表型的数据结构，类似粉丝列表、文章的评论列表之类的热点数据
SET	无序集合	1.添加、获取、移除单个元素 检查一个元素是否存在于集合中	1.两人的好友集合求交集，可以求两个人的共同好友 2.两人的好友差集，可以做可能认

SET	无序集合	中 2.计算交集、并集、差集 3.从集合里面随机获取元素	2.两人的好友差集，可以做可能认识的好友推荐。 3.并集去重可以做统计类的需求
HASH	包含键值对的无序散列表	1.添加、获取、移除单个键值对 获取所有键值对 2.检查某个键是否存在	结构化的数据，比如一个对象，缓解key数量增多产生的频繁rehash。
ZSET	有序集合	1.添加、获取、删除元素 2.根据分值范围或者成员来获取元素 3.计算一个键的排名	1.排行榜需求 2.延迟队列 3.限速器

Redis的应用场景

缓存

将热点数据放到内存中，提升访问速度，缓解DB压力。

计数器

可以对 String 进行自增自减运算，从而实现计数器功能。Redis 这种内存型数据库的读写性能非常高，很适合存储频繁读写的计数量。

分布式ID生成

利用自增特性，一次请求一个大一点的步长如 incr 2000 ,缓存在本地使用，用完再请求

海量数据统计

位图（bitmap）:存储是否参过某次活动，是否已读谋篇文章，用户是否为会员， 日活统计

```
setbit bitk offset 0/1
setbit login:0522 001 1
setbit login:0522 002 1
```

```
setbit login:0521 001 1
setbit login:0521 002 1


bitcount login:0522

gitbit act:0001 001

0 1 1 0 0 0 0 0
0 1 1 0 0 0 0 0
------------------
0 1 1 0 0 0 0 0

0 1 2 3 4 5 6 7

```

会话缓存•

可以使用 Redis 来统一存储多台应用服务器的会话信息。当应用服务器不再存储用户的会话信息，也就不再具有状态，一个用户可以请求任意一个应用服务器，从而更容易实现高可用性以及可伸缩性。

分布式队列/阻塞队列

List 是一个双向链表，可以通过 lpush/rpush 和 rpop/lpop 写入和读取消息。可以通过使用brpop/blpop 来实现阻塞队列。

分布式锁实现

在分布式场景下，无法使用基于进程的锁来对多个节点上的进程进行同步。可以使用 Redis 自带的 SETNX 命令实现分布式锁

热点数据存储

最新评论，最新文章列表，使用list 存储,ltrim取出热点数据，删除老数据

社交类需求

Set 可以实现交集，从而实现共同好友等功能，Set通过求差集，可以进行好友推荐，文章推荐。

```
sadd you a b c d
sadd me c d e f

sdiff you me : a b

```

排行榜

ZSet 可以实现有序性操作，从而实现排行榜等功能。

延迟队列

使用sorted_set，使用 【当前时间戳 + 需要延迟的时长】做score, 消息内容作为元素,调用zadd来生产消息，消费者使用zrangbyscore获取当前时间之前的数据做轮询处理。消费完再删除任务 rem key member

```

支付 30 min

zadd delay_queue new +30min task:orderId
zrangbyscore delay_queue 0 now


```

Redis持久化

Redis是基于内存的数据库，同时提供了持久化的能力，持久化就是把内存的数据写到磁盘中去，防止服务宕机了内存数据丢失。

Redis有哪些持久化方式？各自的优缺点?

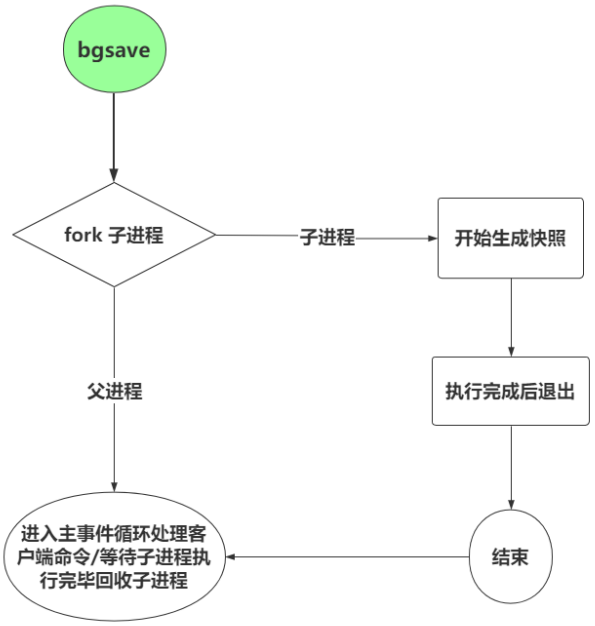
Redis 提供两种持久化机制 RDB 和 AOF 机制。
RDB（Redis DataBase）：RDB保存某一个时间点之前的快照数据。
AOF（Append-Only File）：指所有的命令行记录以 redis 命令请求协议的格式完全持久化存储保存为 aof 文件。
混合持久化（4.x）：指进行AOF重写时子进程将当前时间点的数据快照保存为RDB文件格式，而后将父进程累积命令保存为AOF格式。

RDB 快照有两种触发方式

1.为通过配置参数，如下：
通过一定的时间周期内看，命令执行的个数，超过阈值及执行快照生成

```
save 900 1
save 300 10
save 60 10000
```

2.为通过手动执行bgsave，显示触发生成快照



优点：

1.性能最大化，fork 子进程来完成写操作，让**主进程继续处理命令**。使用单独子进程来进行持久化，保证了 redis 的高性能

2.当重启恢复数据的时候，数据量比较大时，Redis直接解析**RDB二进制文件**，生成对应的数据存储在内存中，比 AOF 的**启动效率更高**。

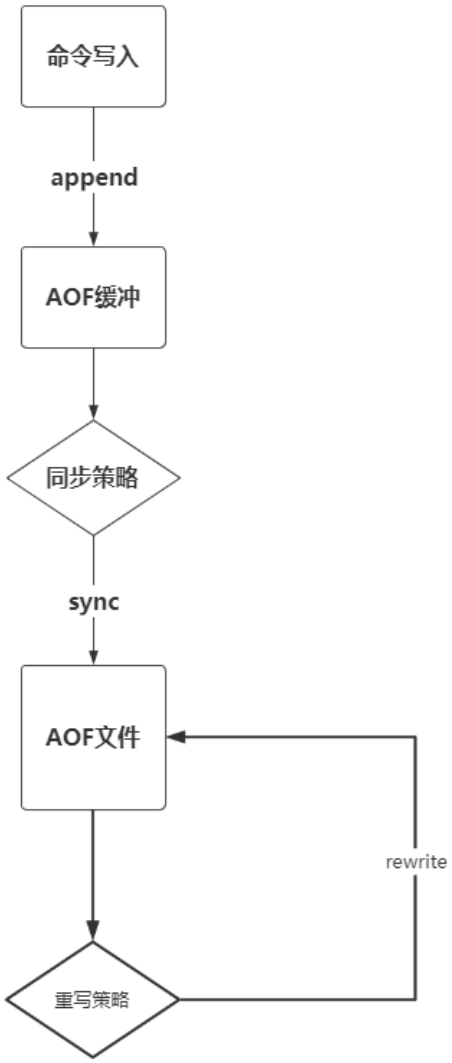
缺点：

1.数据安全性低。RDB 是**间隔一段时间**进行持久化，如果持久化之间 redis 发生故障，会发生数据丢失。所以这种方式更适合数据要求不严谨的时候)

2.RDB方式数据没办法做到实时持久化/秒级持久化。因为bgsave每次运行都要执行**fork**操作创建子进程，属于重量级操作，会消耗比较大的内存空间。

AOF持久化执行流程

通过 appendonly yes 开启



Redis使用单线程响应命令，如果每次写AOF文件命令都追加到磁盘，会极大影响处理性能，所以Redis先写入aof缓冲区，根据用户配置的同步硬盘策略写入aof文件中，可以通过 appendfsync 参数配置同步策略：值得含义如下

```
no：表示等操作系统进行数据缓存同步到磁盘
（快速响应客户端，不对AOF做数据同步，同步文件由操作系统负责，通常同步周期最长30s）
always：表示每次更新操作后手动调用 fsync() 将数据写到磁盘（每次都写磁盘，响应客户端慢，数据最安全）
everysec：表示每秒同步一次（折中，默认值）
```

AOF 重写机制

随着命令得不断写入AOF，文件会越来越大，为了解决这个问题Redis引入了AOF重写机制压缩文件体积。AOF文件重写是把Redis进程内的数据转化为写命令同步到新AOF文件的过程。AOF重写机制可以通过手动触发和自动触发

手动触发：bgrewriteaof命令

自动触发：auto-aof-rewrite-min-size和auto-aof-rewrite-percentage参数确定自动触发时机

auto-aof-rewrite-min-size：表示运行AOF重写时文件最小体积，默认为64MB。

auto-aof-rewrite-percentage：代表当前AOF文件空间（aof_current_size）和上一次重写后AOF文件空间（aof_base_size）的比值。

自动触发时机

```
aof_current_size>auto-aof-rewrite-minsize&&（aof_current_size-aof_base_size）/aof_base_size>=auto-aof-rewritepercentage
```

aof_current_size和aof_base_size可以在info Persistence统计信息中查看

优点：数据安全，aof 持久化可以配置 appendfsync 属性，有 always，每进行一次 命令操作就记录到 aof 文件中一次。

缺点：数据集大的时候，比 rdb 启动效率低。

混合持久化：

可以通过设置 aof-use-rdb-preamble yes 开启

加载时，首先会识别AOF文件是否以REDIS字符串开头，如果是，就按RDB格式加载，加载完RDB后继续按AOF格式加载剩余部分。

混合式持久化方案兼顾了RDB的速度，和aof的安全性。

Redis 是如何处理过期数据的?

过期淘汰

对于已经过期的数据，Redis 将使用两种策略来删除这些过期键，它们分别是**惰性删除**和**定期删除**

惰性删除是指 Redis 服务器不主动删除过期的键值，而是当访问键值时，再检查当前的键值是否过期，如果过期则执行删除并返回 null 给客户端；如果没过期则正常返回值信息给客户端。
它的优点是简单，不需要对过期数据做额外的处理，只是在每次访问时才检查键值是否过期。缺点是删除过期键不及时，造成了一定的空间浪费。

定期删除是指 Redis 服务器每隔一段时间会检查一下数据库，看看是否有过期键可以被清除。
默认情况下 Redis 定期检查的频率是每秒扫描 10 次，用于定期清除过期键。当然此值还可以通过配置文件进行设置，在 redis.conf 中修改配置“hz”即可，默认的值为“hz 10”。定期删除的扫描并不是遍历所有的键值对，这样的话比较费时且太消耗系统资源。Redis 服务器采用的是**随机抽取形式**，每次从过期字典中，取出 20 个键进行过期检测，过期字典中存储的是所有设置了过期时间的键值对。如果这批随机检查的数据中有 **25%** 的比例过期，那么会再抽取 20 个随机键值进行检测和删除，并且会循环执行这个流程，直到抽取的这批数据中过期键值小于 **25%**，此次检测才算完成。Redis 服务器为了保证过期删除策略不会导致线程卡死，会给过期扫描增加了最大执行时间为 25ms，及每次扫描不会超过25ms。

当内存不够用时 Redis 又是如何处理的?

Redis 内存淘汰策略

当 Redis 的内存超过最大允许的内存之后，Redis 会触发内存淘汰策略
当 Redis 内存不够用时，Redis 服务器会根据服务器设置的淘汰策略，删除一些不常用的数据，以保证 Redis 服务器的顺利运行。

在 4.0 版本之前 Redis 的内存淘汰策略有以下 6 种。

noeviction：不淘汰任何数据，当内存不足时，执行缓存新增操作会报错，它是 Redis 默认内存淘汰策略。
allkeys-lru：淘汰整个键值中最久未使用的键值。
allkeys-random：随机淘汰任意键值。
volatile-lru：淘汰所有设置了过期时间的键值中最久未使用的键值。
volatile-random：随机淘汰设置了过期时间的任意键值。
volatile-ttl：优先淘汰更早过期的键值。
而在 Redis 4.0 版本中又新增了 2 种淘汰策略：
volatile-lfu，淘汰所有设置了过期时间的键值中最少使用的键值；
allkeys-lfu，淘汰整个键值中最少使用的键值。

内存淘汰策略可以通过配置文件来修改，redis.conf 对应的配置项是“maxmemory-policy noeviction”，只需要把它修改成我们需要设置的类型即可

如何保证缓存与数据库双写时的数据一致性?

你只要用缓存，就可能会涉及到缓存与数据库双存储双写，你只要是双写，就一定会有数据一致性的问题，比如更新数据库后，写缓存失败，那么你如何解决一致性问题?

一般来说，要完全保证数据库和缓存的一致性，需要将请求同步串行化，这样往往性能上是不可接受的，缓存可以稍微的跟数据库偶尔有不一致的情况，最好不要做这个方案。**串行化之后**，就会导致系统的吞吐量会大幅度的降低。

还有一种方式就是可能会暂时产生不一致的情况，但是发生的几率特别小，就是**先更新数据库，然后再更新缓存。**缓存更新如果失败后，可以进行重试，如果重试失败，则将失败的 key 写入**消息队列**中，待缓存访问恢复后，将这些 key 从缓存删除。这些 key 在再次被查询时，重新从 DB 加载，从而保证数据的一致性。也可以设置较短的过期时间，缩短数据不一致的时间。

问题场景	描述	解决
先写缓存，再写数据库，缓存写成功，数据库写失败	缓存写成功，但写数据库失败或者响应延迟，则下次读取（并发读）缓存时，就出现脏读	这个写缓存的方式，缓存置为失效；读取再写缓存
先写数据库，再写缓存，数据库写成功，缓存写失败	写数据库成功，但写缓存失败，则下次读取（并发读）缓存时，则读不到数据	缓存使用时，假如读现

Redis 主从复制的原理是怎样的?

为什么需要主从复制功能呢？有两个作用

1）读写分离，单台服务器能支撑的QPS是有上限的，我们可以部署一台主服务器、多台从服务器，主服务器只处理写请求，从服务器通过复制功能同步主服务器数据，只处理读请求，以此提升Redis服务能力；

2）数据容灾，任何服务器都有宕机的可能，我们同样可以通过主从复制功能提升Redis服务的可靠性；由于从服务器与主服务器数据保持同步，一旦主服务器宕机，可以立即将请求切换到从服务器，

从而避免Redis服务中断。

复制原理：

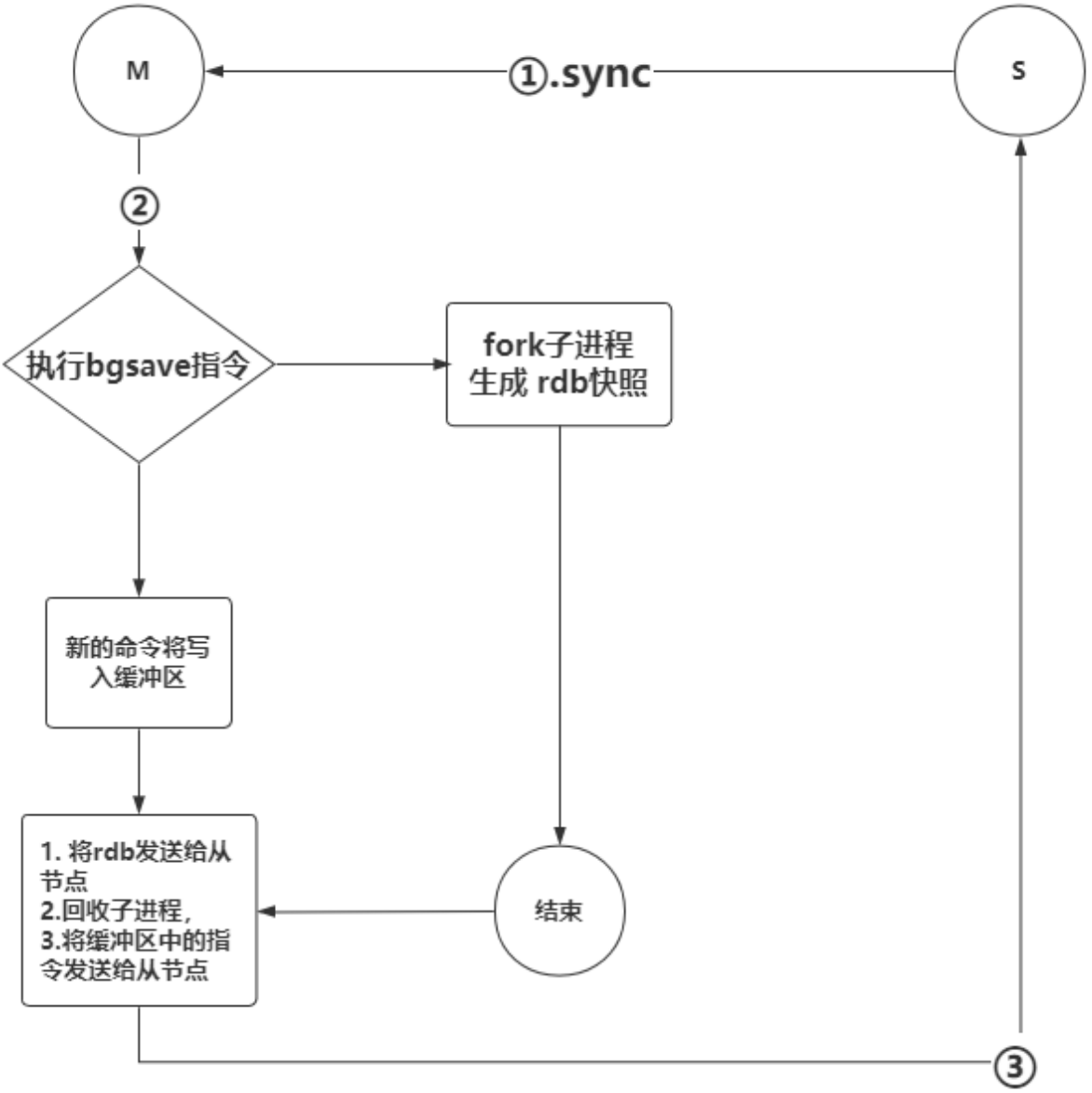
1. 从节点向主节点发送 sync 命令， 请求同步数据

2. 主节点收到sync命令，开始执行bgsave持久化数据到一个 rdb文件，并且在持久化期间会将所有新执行的写入命令都保存到一个缓冲区

3.当持久化数据执行完毕后，主节点将该RDB文件发送给从服务器，从服务器接收该文件，并加载到内存中。

4.主节点将缓冲区中的指令发送给从节点

5.每当主节点接收到写命令时，都会将该指令按照Redis协议发送给从节点，从节点接收并处理主节点发过来的命令。



上述流程以及可以完成主从复制的基本功能，但是由于bgsave是一个重量级的操作，如果复制过程中发现了网络问题，从节点重写连到主节点时，又执行了sync 请求，如果这个重连的时间很短的化，主节点数据没有发生很大的变化，这时是没必要重新生成快照的，所以 Redis2.8 以后提出了新的主从复制解决方案，从服务器会记录已经从主节点接收到的数据量（复制偏移量），Redis主节点会维护一个复制缓冲区，记录自己已执行且待发送给从服务器的命令请求，同时还需要记录缓冲区第一个字节的**复制偏移量**，从节点同步请求改为了 **psync**，

当从节点连接到主节点是会 发送 **psync 同时带上已经接收到的复制偏移量**，如果该复制偏移量在主节点的复制缓冲区区间内，则不需要执行持久化操作，主节点直接发送复制缓冲区中的指令即可。这就是部分重同步。

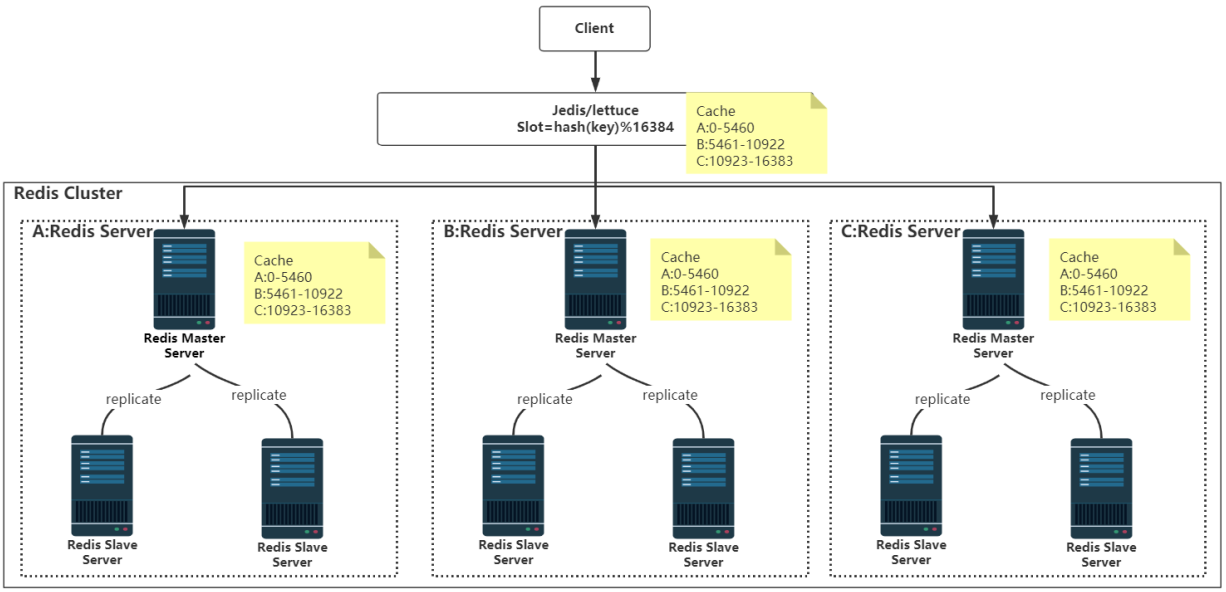
你知道有哪些Redis分区实现方案?

1. **客户端分区**就是在客户端就已经决定数据会被存储到哪个redis节点或者从哪个redis节点读取。
    a. 范围切分
    b. hash切分：一致性哈希/普通哈希

2. **代理分区**意味着客户端将请求发送给代理，然后代理决定去哪个节点写数据或者读数据。代理根据分区规则决定请求哪些Redis实例，然后根据Redis的响应结果返回给客户端。redis的代理分区实现如Twemproxy。

3. **查询路由**(Query routing) 的意思是客户端随机地请求任意一个redis实例，然后由Redis将请求转发给正确的Redis节点。Redis Cluster实现了一种混合形式的查询路由，但并不是直接将请求从一个redis节点转发到另一个redis节点，而是在客户端的帮助下直接redirected到正确的redis节点。

为什么要做Redis分片？redis 集群模式的工作原理能说一下么?



redis集群是一个由多个主从节点群组成的分布式服务器群，它具有复制、高可用和分片特性。Redis集群不需要sentinel哨兵也能完成节点移除和故障转移的功能。Redis集群模式没有中心节点，可水平扩展，据官方文档称可以线性扩展到上万个节点(官方推荐不超过1000个节点)。

高并发：支持大并发，通过横向拓展实现，通过 --cluster add-node 添加新机器到集群执行 reshard 命令，重新分配slot ，将相对均摊了 slot 的分布，缓冲了 其他机器的并发压力，从而应对 百万，甚至上千万的并发。

复制：每个小集群都是一个主从复制的架构，从而保证了 主节点挂掉的时候，不至于丢失全部数据，当选举产生新的master节点后，继续对外进行服务,在主备切换过程中，部分key会有影响，但是其他分片上的key不会有任何影响，从而保证了高可用的场景。

分片：每个不同的主从架构小集群，数据是不一致的，客户端通过哈希函数，将数据路由到不同的数据节点，从而实现了数据的分片。这样技术内存不够用的时候，只需要添加新的集群节点进来，重新分配一下slot 就可以了。

方案说明

1. 通过哈希的方式，将数据分片，每个节点均分存储一定哈希槽(哈希值)区间的数据，默认分配了16384 个槽位
2. 每份数据分片会存储在多个互为主从的多节点上
3. 数据写入先写主节点，再同步到从节点

4. 同一分片多个节点间的数据不保持一致性
5. 读取数据时，当客户端操作的key没有分配在该节点上时，redis会返回转向指令，指向正确的节点

在集群模式下，redis 的 key 是如何寻址的?

每个服务器都会维护一个 slot表，对应了每个 slot 真实的物理节点，当服务器收到命令时，会对 key进行 crc16 进行哈希，得到哈希值 后，对 16384 进行取模，取模的值就是key 对应的slot，如果 这个slot 是由当前服务器处理，则直接继续执行命令，如果不是由当前节点处理，则返回该slot 对应的服务器节点地址，由客户端重写请求对应的地址。

Redis分片有什么缺点?

1.涉及多个key的操作通常不会被支持。例如你不能对两个集合求交集，因为他们可能被存储到不同的Redis实例（实际上这种情况也有办法，但是不能直接使用交集指令）。

2.同时操作多个key,则不能使用Redis事务.

- 分区使用的粒度是key，不能使用一个非常长的排序key存储一个数据集
- 当使用分区的时候，数据处理会非常复杂，例如为了备份你必须从不同的Redis实例和主机同时收集RDB / AOF文件。

Redis是单线程的，如何提高多核CPU的利用率?

可以在同一个服务器部署多个Redis的实例，并把他们当作不同的服务器来使用，在某些时候，无论如何一个服务器是不够的， 所以，如果你想使用多个CPU，你可以考虑一下分片（shard）。

假如Redis里面有1亿个key，其中有10w个key是以某个固定的已知的前缀开头的，如果将它们全部找出来?

使用keys指令可以扫出指定模式的key列表。由于redis的单线程的。keys指令会导致线程阻塞一段时间，线上服务会停顿，直到指令执行完毕，服务才能恢复。这个时候可以使用scan指令，scan指令可以无阻塞的提取出指定模式的key列表，但是会有一定的**重复概率**，在客户端做一次去重就可以了，但是整体所花费的时间会比直接用keys指令长。

什么是缓存失效？如何解决

在写缓存时，我们一般会根据业务的访问特点，给每种业务数据设置一个过期时间，让缓存数据在这个**固定的过期时间**后被淘汰。一般情况下，因为缓存数据是

逐步写入的，所以也是逐步过期被淘汰的。但在某些场景，如缓冲预热时，一大批数据会被系统主动或被动从 DB 批量加载，然后写入缓存。这些数据写入缓存时，**由于使用相同的过期时间**，在经历这个过期时间之后，这批数据就会一起到期，从而被缓存淘汰。此时，对这批数据的所有请求，都会出现缓存失效，从而都穿透到 DB，DB 由于查询量太大，就很容易压力大增，请求变慢。

解决方案

对于批量 key 缓存失效的问题，原因既然是预置的固定过期时间，那在设计缓存的过期时间时，可以使用：**过期时间=固定时间+随机时间**。即相同业务数据写缓存时，在基础过期时间之上，再加一个随机的过期时间，让数据在未来一段时间内慢慢过期，避免瞬时全部过期，对 DB 造成过大压力，

什么是缓存穿透，如何解决

大量的请求访问一个不存在的Key， 由于缓冲数据不存在，则会穿透到数据库，这个时候，大量请求同时访问数据库，容易造成数据库奔溃，从而使系统对外不可用，这就是缓冲穿透。缓存穿透存在的原因，就是因为我们在系统设计时，更多考虑的是正常访问路径，对特殊访问路径、异常访问路径考虑相对欠缺。如果由大量的请求被认为构造，则对系统的伤害时致命性的。

解决方案

1. 查询这些不存在的数据时，第一次查 DB，虽然没查到结果返回 NULL，仍然记录这个 key 到缓存，只是这个 key 对应的 value 是一个特殊设置的值，如 empty,且设置一个过期时间。但是当key 过多的时候，也会操作内存的浪费。

2. 构建一个 BloomFilter 缓存过滤器，记录全量数据，这样访问数据时，可以直接通过 BloomFilter 判断这个 key 是否存在，如果不存在直接返回即可，根本无需查缓存和 DB

什么是缓存雪崩

在流量洪峰到达时，大量的正常请求导致了，缓存服务器宕机，所有请求到达db，导致了db服务不可用，就时缓冲雪崩。

解决方案：
1. 对缓存进行实时监控，当请求访问的慢速比超过阈值，及时报警，通过自动故障转移，服务降级，停止部分非核心接口访问
2. 对大热key，缓存在本地，缓解redis压力。

分布式锁

1.过期时间设置问题
2.保证原子操作问题
3.客户端误删问题
4.释放锁的原子操作问题
5.主备架构，故障转移数据同步问题

RedLock

1.获取**当前时间**
2.按顺序依次向N个Redis节点获取锁，为确保某个Redis节点失败不影响算法继续进行，获取锁还需要设置一个超时时间，Redis获取锁失败，立即尝试下一个节点
3.计算加锁过程的耗时时间， 当前时间减第一步获取锁的时间如果小于锁的有效时间，且客户端从大多数Redis节点都加锁成功，则认为加锁成功，否则认为加锁失败

4.如果最终获取锁的操作成功，锁的有效时间应该重新计算，锁的有效时间=设置的有效时间-加锁消耗的时间
5.如果加锁失败了，则客户端应该释放所有节点对应得锁

Redis线程模型原理?

1.非阻塞IO
2.事件轮询（多路复用）