1、redis 和传统数据库关系(mysql)有啥区别？

1）Redis 是一种 **key-value 类型数据库**(NoSQL 一种)，mysql 是**关系数据库**；
2）Redis 数据操作**主要在内存**，而 mysql 主要**存储在磁盘**；
3）Redis 在某些场景使用中要明显优于 mysql，比如计数器、排行榜等方面；
4）Redis 通常用于一些特定场景，需与 Mysql **一起配合使用**，两者并不是相互替换和竞争关系，而是**共用和配合使用**。
5）redis 优势：
1>**性能极高**，**读写速度**都非常快；
2>redis **数据类型丰富**，不仅仅支持简单的 key-value 类型，同时还提供 **list\set\zset\hash** 等数据结构的存储；
3>redis 支持**数据的持久化**，可以将内存中的数据保持在磁盘中，重启的时候可以再次加载进行使用；
4>redis 支持**主从模式的数据备份**。

2、Redis 通用命令

1）**KEYS**：查看**符合模板的所有 key**；
H?llo→**?** 只匹配**一个**字符；
H*llo→*匹配 **0 个或多个**任意字符；
H[ae]llo→**[ ]**只能匹配**括号内**的字符；
H[^e]llo→**[^]**表示**只排除** e，其他都匹配；
H[a-e]llo→**[-]**匹配 **a 到 e，包括** a 和 e；
2）**DEL**：删除一个或多个指定的 key；
3）**EXISTS**：判断 key 是否存在；
4）**EXPIRE**：给一个 **key 设置有效期**，有效期**到期时**该 key **会被自动删除**；
5）**TTL**：**查看**一个 KEY 的**剩余有效期**。

3、String 数据类型命令

Value 是字符串时，根据**字符串的格式**不同，底层**编码方式不同**，但都是**字节数组形式**存储。对于**整数**和**浮点**类型的数字都会变成**二进制的形式**存储，而**字符串**转成**字节码**存储。
1）**SET**：添加或者修改已经存在的一个 String 类型的键值对；
2）**GET**：根据 key 获取 String 类型的 value；
3）**MSET**：批量添加多个 String 类型键值对；
4）**MGET**：根据多个 key 获取多个 String 类型的 value；
5）**INCR**：让一个整型的 key **自增 1**；
6）**INCRBY**:让一个整型的 **key** 自增并**指定步长**，例如：incrby num 2 让 num 值自增 2；
7）**INCRBYFLOAT**：让一个**浮点**类型的数字**自增**并**指定步长**；（必须指定步长，不常用）
8）**SETNX**：**添加**一个**之前 key 不存在**的 String 类型的键值对，若这个 key 存在，则不执行；
9）**SETEX**：**添加**一个 String 类型的键值对，并且**指定有效期**。（将 set 和 expire 合并了）

4、Key 的层级结构

多个单词之间**用':'隔开**，形成层级结构
例如 user 相关的 key：heima:user:1
如果 Value 是一个 **Java 对象**，例如一个 User 对象，则可以将**对象序列化为 JSON 字符串**后存储。(但修改某个字段不方便，不如 hash)

KEY	VALUE
heima:user:1	{"id":1, "name": "Jack", "age": 21}
heima:product:1	{"id":1, "name": "小米11", "price": 4999}

5、Hash 数据类型命令

Hash 结构可以将对象中每个字段独立存储，可以针对单个字段处理；
1）**HSET key field value**：添加或者修改 hash 类型 key 的 field 的值；
2）**HGET key field**：获取一个 hash 类型 key 的 field 的值；
3）**HMSET**：批量添加多个 hash 类型 key 的 field 的值；
4）**HMGET**：批量获取多个 hash 类型 key 的 field 的值；
5）**HGETALL**：获取一个 hash 类型的 **key 中的所有的 field 和 value**；
6）**HKEYS**：获取一个 hash 类型的 **key 中的所有的 field**；
7）**HINCRBY**:让一个 hash 类型 key 的**字段值自增**并**指定步长**；
8）**HSETNX**：添加一个 hash 类型的 key 的 field 值，若这个 field 存在，则不执行。

6、List 数据类型命令

双向链表结构，有序，元素可重复，插入删除快，查询速度一般。
1）**LPUSH key element** ... ：向列表**左侧插入**一个或多个元素；
2）**LPOP key**：**移除并返回**列表左侧的第一个元素，没有则返回 nil；
3）**RPUSH key element** ... ：向列表**右侧插入**一个或多个元素；
4）**RPOP key**：**移除并返回**列表右侧的第一个元素，没有则返回 nil；
5）**LRANGE key star end**：返回一段**角标范围内的**所有元素
6）**BLPOP** 和 **BRPOP**：与 LPOP 和 RPOP 类似，只不过在**没有元素时等待指定时间**，而不是直接返回 nil。

7、Set 数据类型命令

无序、元素不重复、查找快、支持交集、并集、差集操作。
1）**SADD** key member ... ：向 set 中添加一个或多个元素；
2）**SREM** key member ... ：移除 set 中的指定元素；
3）**SCARD** key： 返回 **set 中元素的个数**；
4）**SISMEMBER** key member：判断一个元素**是否存在 set 中**；
5）**SMEMBERS**：获取 **set 中的所有元素**；
6）**SINTER** key1 key2 ... ：求 key1 与 key2 的**交集**；
7）**SDIFF** key1 key2 ... ：求 key1 与 key2 的**差集**；
8）**SUNION** key1 key2 ..：求 key1 和 key2 的**并集**；

8、SortedSet 数据类型命令（排行榜）

可排序的 set 集合，SortedSet 中的**每一个元素**都带有一个 **score 属性**，可以**基于 score 属性对元素排序**，底层实现是一个**跳表**和 **hash 表**。
1）**ZADD key score member**：添加一个或多个元素到 sorted set，如果**已经存在则更新其 score 值**；
2）**ZREM** key member：删除 sorted set 中的一个指定元素；
3）**ZSCORE** key member ：获取 sorted set 中的**指定元素的 score 值**；
4）**ZRANK** key member：获取 sorted set 中的**指定元素的排名**；
5）**ZCARD** key：取 **sorted set 中**的**元素个数**；
6）**ZCOUNT** key min max：统计**指定 *score* 范围内**的**所有元素的个数**；
7）**ZINCRBY** key increment member：让 sorted set 中的**指定元素自增**，**步长**为 increment 值；
8）**ZRANGE** key min max：按照 **score 排序后**，获取**指定*排名*范围内**的元素；
9）**ZRANGEBYSCORE** key min max：按照 **score 排序后**，获取**指定 *score* 范围内**的元素；
10）**ZDIFF.ZINTER.ZUNION**：求差集.交集.并集。
【注意】排名**默认 score 升序**，Zrev~降序。

9、Bitmap（位图）

Bitmap 存储的是**连续的二进制数字(0 和 1)**，通过 Bitmap，只需要**一个 bit 位**来**表示某个元素对应的值或状态**，**key** 就是对应**元素本身**。

命令	介绍
SETBIT key offset value	设置指定 offset 位置的值
GETBIT key offset	获取指定 offset 位置的值
BITCOUNT key start end	获取 start 和 end 之前值为 1 的元素个数
BITOP operation destkey key1 key2 ...	对一个或多个 Bitmap 进行运算，可用运算符有 AND, OR, XOR 以及 NOT

【应用场景】
需要**保存状态信息**（0/1 即可表示）的场景；
举例：用户**签到**情况、活跃用户情况、用户行为统计（比如**是否点赞**过某个视频）。

10、HyperLogLog（基数统计）

基数计数概率算法为了**节省内存**并不会直接存储元数据，而是通过一定的概率统计方法**预估基数值**（集合中**包含元素的个数**）
【应用场景】数量量巨大（百万、千万级别以上）的计数场景；
举例：热门网站每日/每周/每月访问 ip 数统计、热门帖子 uv 统计。

11、Geospatial (地理位置)

主要用于**存储地理位置信息**，基于 Sorted Set 实现。通过 GEO 可以轻松实现**两个位置距离的计算**、获取**指定位置附近的元素**等功能。

命令	介绍
GEOADD key longitude1 latitude1 member1 ...	添加一个或多个元素对应的经纬度信息到 GEO 中
GEOPOS key member1 member2 ...	返回给定元素的经纬度信息
GEODIST key member1 member2 M/KM/FT/MI	返回两个给定元素之间的距离
GEORADIUS key longitude latitude radius distance	获取指定位置附近 distance 范围内的其他元素，支持 ASC(由近到远)、DESC（由远到近）、Count(数量) 等参数
GEORADIUSBYMEMBER key member radius distance	类似于 GEORADIUS 命令，只是参照的中心点是 GEO 中的元素

12、SpringDataRedis 中 RedisTemplate

将**不同数据类型的操作** API 封装到了**不同的类型中**：**redisTemplate.opsFor~**

redisTemplate.opsForValue()：String 类型
redisTemplate.opsForHash()：Hash 类型
redisTemplate.opsForList()：List 类型
redisTemplate.opsForSet()：Set 类型
redisTemplate.opsForZSet()：SortedSet 类型

1）配置 redis 和 common-pool 依赖

```xml
<!--redis依赖-->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
<!--common-pool-->
<dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-pool2</artifactId>
</dependency>
```

2）在 application.yml 配置 Redis 信息

```yaml
spring:
  redis:
    host: 192.168.150.101
    port: 6379
    password: 123321
    lettuce:
      pool:
        max-active: 8  #最大连接
        max-idle: 8   #最大空闲连接
        min-idle: 0   #最小空闲连接
        max-wait: 100ms #连接等待时间
```

3）注入 RedisTemplate

```java
@Autowired
private RedisTemplate<String, Object> redisTemplate;

@Test
void testString() {
    // 写入一条String数据
    redisTemplate.opsForValue().set("name", "虎哥");
    // 获取string数据
    Object name = redisTemplate.opsForValue().get("name");
    System.out.println("name = " + name);
}
```

【注意】1）RedisTemplate 可以接收任意 Object 作为值写入 Redis；但是 RedisTemplate 内部默认采用 **JDK 序列化**，将 **Object 对象*序列化***为**字节形式**。（可读性差，占内存）
2）自定义 RedisTemplate 的序列化方式为 **JSON 序列化时**，能将 **Java 对象**自动的***序列化***为 **JSON 字符串**，并且查询时能自动把 **JSON *反序列化*为 Java 对象**；但是记录了序列化时**对应的 class 名称**（为了查询时**实现自动反序列化**），额外内存开销。

13、RedisTemplate 子类 StringRedisTemplate

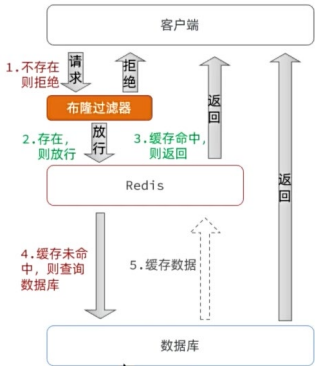
它的 key 和 value 的序列化方式默认就是 String 方式。我们只需要手动序列化将对象转成 json，然后读取的时候，手动反序列化将 json 转成对象。

```java
@Autowired
private StringRedisTemplate stringRedisTemplate;

@Test
void testString() {
    // 写入一条String数据
    stringRedisTemplate.opsForValue().set("verify:phone:13600527634", "124143");
    // 获取string数据
    Object name = stringRedisTemplate.opsForValue().get("name");
    System.out.println("name = " + name);
}

private static final ObjectMapper mapper = new ObjectMapper();

@Test
void testSaveUser() throws JsonProcessingException {
    // 创建对象
    User user = new User("虎哥", 21);
    // 手动序列化
    String json = mapper.writeValueAsString(user);
    // 写入数据
    stringRedisTemplate.opsForValue().set("user:200", json);

    // 获取数据
    String jsonUser = stringRedisTemplate.opsForValue().get("user:200");
    // 手动反序列化
    User user1 = mapper.readValue(jsonUser, User.class);
    System.out.println("user1 = " + user1);
}
```

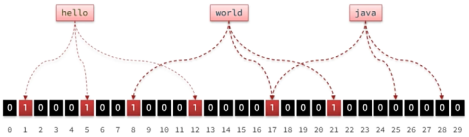
14、主从同步原理（高可用、高并发读）
1）当主从第一次同步连接或断开重连时，从节点都会**发送 psync 请求并携带 replID 和 offset 值**，尝试数据同步；
2）master 节点判断 slave 节点**是否是第一次同步**（**replid 是否一致**）或**offset 是否被覆盖**；
3）是；**全量同步**；则 master 把**发送 RDB 文件**给 slave；（执行 **bgsave 后台执行命令**，它会开启一个独立进程，将 **redis 在内存中的所有数据**都***持久化***到**硬盘**中，**生成 RDB 文件**）
4）否；**增量同步**；则 master 把 **slave 错过的数据命令**发送给 slave；（**第一次建立同步之后**，用 **repl_backlog** 记录**所有写操作命令**，主从节点**各自记录 offset 值**，**二者之差**即为需要**增量同步的数据命令**）
5）主从**保持连接状态**时，master 写数据时，都将命令传播给 slave，保持**实时同步**；
【replicationID】每个 master 节点都有唯一 id，并且所有 slave 节点与其保持一致；
【offset】repl_backlog 中写过的数据长度；主从 offset 一致代表数据一致。
【注意】
repl_backlog 设计为一个**环形数组**；**正常**情况，slave 会落后于 master 一点，**slave 同步过的数据都可以被覆盖继续写**；**异常**情况，**slave 宕机时间较长**，master 写了一圈就会**覆盖之前未同步的数据**，那么做***全量同步***。

15、主从集群优化
1）在 master 中配置 **repl-diskless-sync**：yes；启用**无磁盘复制**，即 master **通过网络**把数据直接传给 slave，避免全量同步时的磁盘 IO；
2）**redis 单节点上的内存占用不要太大**，减少 RDB 导致的过多磁盘 IO；
3）适当**提高 repl_backlog 的大小**；
4）master 主要用于写，太多 slave 压力大，可以采用**主-从-从**，但数据同步**时效性降低**。
16、哨兵原理
Redis 提供**哨兵机制**来实现主从集群的**自动故障恢复**。具体作用如下：
1）服务状态监控：哨兵基于**心跳机制**监测服务状态，**每隔 1 秒**向集群每个实例**发送 ping**；
【**主观下线**】如果**某哨兵节点**发现某实例在**规定时间内未响应**，则认为主观下线；
【**客观下线**】如果**超过指定数量**（quorum）的哨兵都认为该实例主观下线，即为**客观下线**；quorum 值最好超过哨兵数量的一半。
2）自动故障转移：一旦发现 master 故障，则需要在 slave 中选举新的 master；依据：
1>首先判断 slave 与 master **断开的时间长短**；（大于指定值则排除该 slave）
2>然后判断 **slave 的优先级**；（越小优先级越高）
3>如果优先级一样，则判断***slave 的 offset 值***；（**越大**说明**数据越新**，选举优先级越高）
4>offset 一样，最后判断 **slave 的运行 id 大小**；（越小，选举优先级越高）
选中后，哨兵会给选中的 slave **发送 slaveof no one 命令**，让该节点**成为 master**；
给其他 slave **发送 slaveof ip port**；让这些 slave 成为新 master 的**从节点，并同步数据**；
给故障节点**标记为 slave**，恢复后为从节点。
17、分片集群（海量数据存储、高并发写）
集群中有**多个 master**，每个 master **保存不同数据（散列插槽）**，并且可也有**多个 slave 节点**，**master 之间通过 ping 检测**彼此健康状态。
18、分片集群怎么分配数据和读数据？
在 redis 集群中，**共有 16384 个 hash slots**，集群中每一个 master 节点都会分配一定数量的 hash slots。【具体来说】**redis 数据**不是和节点绑定，而是**和插槽 slot 绑定**；当读写数据时，redis ***基于 CRC16 算法***对 **key 做哈希运算**，得到的结果**和 16384 取余**，计算出这个 key 的 **slot 值**。最后到 slot 所在 redis 节点读写操作。
19、redis 计算 key 的 hash 值有 2 种方式：
1）key 中**有{}**，根据**{}内字符串**计算 hash slot；
2）key 中**没有{}**，根据**整个 key 的字符串**；
20、redis 为什么这么快？
1）**基于内存**，内存访问速度比磁盘块；
2）内置了多种**优化后的数据结构**；（skip table）
3）**单线程事件循环**和 **IO 多路复用**。
【为啥不当做主数据库？】
内存成本太高；redis 的**数据持久化有风险**。
21、redis 和 memcached 的异同
【相同点】
1）**都基于内存**，当**做缓存**使用；
2）都有**过期策略**；
3）两者**性能**都非常高。
【不同点】
1）数据类型：redis 支持**更丰富的数据类型**；而 memcached **只支持 k-v** 数据类型；
2）数据持久化：redis 支持**持久化数据到磁盘**，供后续恢复；而 memcached **只存在内存中**；
3）集群模式：redis 支持，memcached 不支持；
4）特性支持：redis 有**事务等**，memcached 无；
5）过期数据删除：redis 支持**惰性删除**和**定期删除**；memcached **只支持惰性删除**。
22、常见的缓存读写策略（缓存一致性）
1）旁路缓存模式（读多写少）
由业务开发者在更新数据库的同时更新缓存；
读：
1>先从 **cache** 中读取数据，**读到直接返回**；
2>**读不到**就到**数据库中读取数据返回**；
3>最后把数据**放到 cache 中**；
写：
1>先**更新/删除**数据库数据，再**直接删除 cache**；
2>**新增**操作，等到**查询**该数据时**才加到 cache**；
【先删 cache 后更新数据库会数据不一致】
请求 1 把 cache 中的 A 数据删除了→请求 2 从数据库中读取数据→请求 1 再更新数据库。（更新数据库之前，读取数据库旧数据）
【先更新数据库后删 cache 也会数据不一致】
当请求 1 从数据库读数据 A→然后请求 2 更新数据库中数据 A→请求 1 将旧数据 A 写入 cache。（更新数据库之前，把旧数据带入了 cache，概率很小）
【缺陷 1】**首次请求数据**一定**不在 cache** 问题：可以将**热点数据提前**放在 cache 中。
【缺陷 2】**写多**会**频繁删除 cache**，**命中率低**：数据库和缓存强一致；更新 db **同时更新 cache**；短暂允许不一致；给缓存加短的 ttl；
2）读写穿透（少见）
缓存和数据库整合为一个服务，**cache 服务**负责**将此数据读取和写入 db**。
读：
1>先从 **cache** 中读取数据，**读到直接返回**；
2>**读不到**就到数据库中***加载数据到 cache***；
3>最后***从 cache 中返回***；
写：
1>先查 cache，**cache 中不存在**，直接**更新 db**；
2>**cache 中存在**，则**先更新 cache**，然后 **cache 服务自己更新 db**。
3）异步缓存写入
（少见，消息队列中消息异步写入磁盘）
直接基于缓存操作，不对数据库直接操作，而是**其他线程异步**将缓存**持久化到数据库**。
23、RDB 持久化
通过快照获取内存数据在某时间点上的副本。
1）主从模式全量同步数据；2）重启时加载；
24、RDB 创建快照时会阻塞主线程嘛？
save：同步保存操作；会阻塞 redis 主线程。
bgsave：独立子进程后台执行，不阻塞（默认）
25、AOF 持久化（实时好，缓冲区有风险）
没执行一条会更新 redis 中数据的命令，都会将该命令写入 AOF 缓冲区中，再写入 AOF 文件（还在系统内核缓冲区），最后根据持久化方式（fsync 策略）同步到磁盘中。
26、AOF 工作基本流程
1）命令追加：所有写命令追加到 **AOF 缓冲区**；
2）文件写入：调用 **write 系统函数**将 AOF 缓冲区数据**写入*系统内核缓冲区*的 AOF 文件**中；
3）文件同步：调用 **fsnc 系统函数**强制将 AOF 缓冲区数据**同步到磁盘中**；（fsnc 完前**会阻塞**）
4）文件重写：**定期对 AOF 文件重写**，以**压缩**；
5）重启加载：redis 重启，可加载 AOF 恢复。
27、AOF 持久化方式有哪些？（刷盘时机）
1）appendfsnc **always**：**write 后，后台线程*立即***调用 fsnc 函数**刷盘同步** AOF 文件（性能低）
2）appendfsnc **everysec**：write 后，后台线程***每秒***调用 fsnc 函数刷盘***同步一次*** AOF 文件；
3）appendfsnc **no**：write 后，后台线程让***操作系统决定何时***刷盘同步数据；（Linux30 秒）
28、AOF 为什么在执行完命令之后记录日志？
1）避免**额外的检查开销**，AOF 记录日志不会对命令进行**语法检查**；
2）在命令执行完后再记录，**不会阻塞当前的命令执行**。（**Redis 主线程**进行 AOF 记录日志）
风险：1）执行完命令**宕机**了，**少一条命令**；
2）**阻塞后续命令**执行。
29、AOF 重写机制
1）构建**新 AOF 文件**和 **AOF 重写缓冲区**；
2）**读取**所有**缓存的键值对数据**，并为每个键值对***生成一条最新命令***，**写入新的 AOF 文件**；（此过程很**耗时**，使用**后台子进程** bgwruteaof）
3）重写期间，**新的写命令**加入 **AOF 缓冲区**（防止重写过程宕机，可用于***恢复数据***）和 **AOF 重写缓冲区**（保证**数据一致性**）；
4）**AOF 重写缓冲区**写入**新的 AOF 文件**中，保证新 AOF 文件**和数据库状态一致**；
5）最后新的 AOF 文件**覆盖**旧 AOF 文件。
30、AOF 校验机制
Redis 启动时会**检查 AOF 文件是否完整**（有无**损坏**或**丢失数据**）。
检查**“校验和”的数字**：通过对**整个 AOF 文件内容**进行 **CRC64 算法**计**算出的数字**；并将其**保存在文件末尾**，加载时**验证是否一致**。
31、如何选择 RDB 和 AOF？
1）RDB 好于 AOF：
1>redis 丢失一些数据不影响的话，用 RDB；RDB 用**二进制存储**，保存某个时间点的数据，**文件很小**；AOF **追加命令**，**文件很大**，重写后会压缩，但是**重写期间会写入磁盘两次**。
2>RDB 文件恢复**直接解析**即可；而 AOF 需要**逐行执行命令**。
2）AOF 好于 RDB：
AOF **实时性好**，**更安全**；RDB 以特定二进制保存，**版本演进**过程**可能不兼容**。
32、Redis 是如何判断数据是否过期的？
***expires 指针**通过**过期字典（hash 表）**来保存数据过期的时间，过期字典的键是 **redis 中的 key**，过期字典的值是**过期时间戳**。
查数据时先看该 key **是否在过期字典**中，**不在则直接返回数据**，有则判断是否过期，**过期则删除返回 null**。
33、Redis 中过期 key 删除策略
Redis **不会实时监测** key 的过期时间，**到点即删需要为每个键加定时器**，对 **CPU 压力大**。
Redis 采用
1）惰性删除：当**有命令操作该 key** 时，检查该 key **是否过期**，过期即删；
2）周期删除：通过**一个定时任务**，**周期性抽样部分**有 TTL 的 key，过期即删。
【每次**随机抽查数量**】expire.c **默认为 20**；
【如何控制**定期删除的执行频率**】server.hz **默认为每秒 10 次**。dynamic-hz 则开启自适应。
【为什么**定期删除不是删除所有过期 key**】为了**平衡内存和性能**；key 太多是**会很耗时**。
34、大量 Key 其中过期怎么办？
可能会使 Redis 的**请求延迟变高**。
1）设置键的**过期时间**时，**尽量随机一点**；
2）**后台异步删除过期 key**，不阻塞主线程。
35、内存淘汰策略
Redis **每次处理客户端命令时**都会**判断内存使用**情况，当 redis 内存使用**达到设置的阈值**时，redis 会**主动挑选*部分 key* 删除**以释放更多内存。
1）**不淘汰**；（默认）内存满时不写入数据；
2）比较 TTL 的剩余时间，**TTL 小的淘汰**；
3）全部 key，**随机**淘汰；
4）设置 TTL 的 key，随机淘汰；
5）全部 key，基于**最近最少使用**(LRU)算法；
6）设置 TTL 的 key，基于 LRU 算法；
7）全部 key，基于**最少频率使用**(LFU)算法；
8）设置 TTL 的 key，基于 LFU 算法；
【注意】RedisObject 结构会保存数据的 lru（最近一次访问时间）

36、缓存穿透（缓存和数据库都不存在）
大量请求的不合理数据在数据库中根本不存在，从而导致请求穿透缓存，直接打到数据库。
【缓存空对象，并设置 TTL】
实现简单，但有**额外消耗**。
【布隆过滤器】
请求先通过布隆过滤器判断 key 是否存在，不存在拒绝该请求。

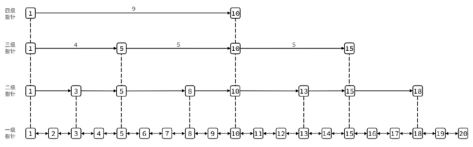


【补充**布隆过滤器**】
是一种**数据统计的算法**，用于检索一个元素是否存在一个集合中，但**无需存储元素**，而是把元素**映射到一个很长的二进制数位上**。
1）需要一个**很长的二进制数**，默认全为 0；
2）需要 **N 个不同算法的哈希函数**；
3）将集合中每个元素根据 **N 个哈希函数运算**，得到 N 个数字，再将 N 个数字对应 bit 位**置 1**
4）判断**元素是否存在**，只需要运算哈希值，看对应 **bit 位是否为 1** 即可。
【注意】只能**判断一定不存在**，不能判断一定存在，因为可能**不同元素*有部分重叠 bit 位***。

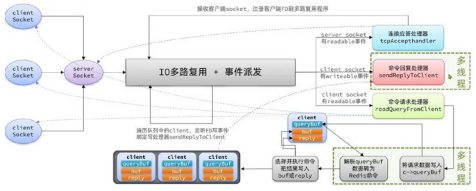


37、缓存雪崩（**大量 key** 缓存无，数据库有）
同一时段***大量的缓存 key 同时失效***或***Redis 宕机***，导致大量请求**到达数据库**。
【大量 key 同时失效----设计**随机失效时间**】
例如在**固定过期时间**的基础上**加上一个随机值**，避免大面积缓存同时到期。
【redis 宕机----**redis 集群**】主从或分片集群；
【redis 宕机----**限流操作**】降低对数据库压力；
【redis 宕机----**多级缓存**】
本地缓存+redis 缓存的二级缓存组合。
38、缓存击穿（**hotkey** 缓存无，数据库有）
对于***高并发访问的 hotkey*** 突然失效，无数请求也会瞬间到达数据库。
【提前预热】针对**热点数据提前预热**，将其存入缓存并**设置合理的过期时间**。
39、Redis 常见阻塞原因
1）【**O(n)的命令**】例如 **key ***；
有**遍历需求**可以使用 **SCAN 命令代替**；
2）【**save 命令**】
创建 **RDB 快照**；（bgsave 不会阻塞）
3）【**AOF**】
日志记录、**刷盘**、**重写**都会阻塞；
4）【**大 Key**】
（string 类型大于 1Mb，其他>1000 元素）
一个 key 对应的 **value 值占用内存过大**。
1>Redis 单线程处理，操作大 key **耗时**；
2>**网络传输**大 key 的**流量高**；
3>**删除**大 key 时**会阻塞**；（分批或异步删除）
5）【**CPU 竞争**】
redis 是典型的 ***CPU 密集型***应用，不建议和其他多核 CPU 密集型服务部署在一起，**争不过**其他的时候会**性能严重下降**。
6）【**网络问题**】**网络延迟**，**网络中断**等。
40、Redis 内存碎片（不可用的空闲内存）
1）redis 存储数据时，向 OS **申请的内存空间**可能会**大于实际需要的空间**；
2）**频繁修改** redis 中的数据也会产生碎片。
【清理 redis 内存】
1）通过 **config set 命令**将 **activedefrag** 配置为 yes 即可；
2）**重启节点**也会整理内存碎片。

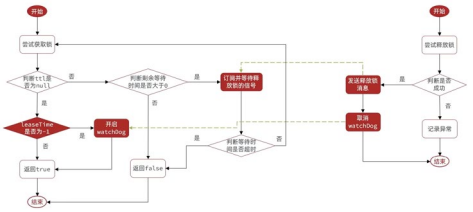
41、Redis 性能优化
1）使用批量操作减少网络传输；
一个 redis 命令执行有 4 步：发送命令→命令排队→命令执行→返回结果；发送命令和返回结果都是**数据网络传输的时间**，故使用***批量操作***可以**减少网络传输次数**。（***原生批操作***，***pipeline 封装***一组命令）
2）大量 Key 集中过期
1>随机过期时间 **2>惰性删除**避免阻塞主线程
3）大 Key 消耗内存和带宽，降低性能；
4）热 Key 占用大量的 **CPU 和带宽**，**影响其他请求**正常处理；（**1>读写分离**：主写从读；**2>分片集群**：热 key 分散在多个节点上）
5）慢查询命令（复杂度高的命令）
42、skiplist 跳表
1）第 1 层**有序的双向链表**；其他层是单向；
2）每个节点包含**多层指针**，层数是 1-32 层；
3）**层级越高**的指针，到下一节点的**跨度越大**；



43、SortedSet 底层数据结构是怎样的
1）SortedSet 需要**存储 score 和 member 值**，而且要快速**根据 member 查 score**，所以底层要有**哈希表**，以 member 为键，score 为值；
2）SortedSet 需要**根据 score 排序**，所以底层有**跳表存储 score 和 member**，并 **score 排序**；
3）根据 *member 查询 score* 时，基于**哈希表**；
4）根据 *score 查 member* 时，基于**跳表**；
5）根据 member 查排名时，先**哈希表**再**跳表**；
44、Redis 时单线程还是多线程？
1）对于 redis **核心业务处理部分**（**命令处理**），都是使用**单线程**；
2）而对于**整个 redis** 来说，是**多线程**的；
在 redis**4.0 版本**中，引入**多线程异步处理**一些**耗时**的任务（如删除命令 unlink）；
在 redis**6.0 版本**中，在**核心网络模型**中引入**多线程**进一步提高对多核 CPU 的利用率，主要体现在 1>**命令请求处理器**中利用多线程**并行解析多个请求**当中的数据；2>**命令回复处理器**利用多线程将队列中的客户端**结果写出**。
45、为什么 Redis 选择单线程？
抛开持久化不谈，redis 是**纯内存**操作，所以它的**性能瓶颈是网络延迟**而不是执行速度，所以多线程并无很大的性能提升；
2）多线程导致**过多的上下文切换**，**额外开销**；
3）多线程要考虑**线程安全问题**，**锁影响**性能。
46、Redis 单线程网络模型的整个流程
1）severSocket 的 **FD 注册**到 aeEventLoop 上，
2）aeApiPoll **等待就绪**；
3）当有 **clientSocket 连上来**（即 severSocket 就绪），就会**调用专门处理 severSocket 读事件**的***连接应答处理器***，接收 clientSocket 的请求并将 clientSocket 的 **FD 注册**到 aeEventLoop 上；（**此时 FD** 不仅有服务端也有客户端）
4）若 **clientSocket 可读**，则调用专门处理 clientSocket 读事件的**命令请求处理器**；
5）获取**当前客户端**，客户端中有 querybuffer 缓冲区用来读写，所以命令请求处理器会将**请求写入**客户端的 **querybuffer 缓冲区**；
6）**解析** querybuffer **缓冲区**中的**数据*转为***redis **命令**，并***写入***客户端 reply（大）/buffer **缓冲区**；
7）将要写出的**客户端放入队列**，排队等输出；
8）beforesleep 遍历队列中的客户端，**监听 FD 写事件**，调用专门处理 clientSocket 写事件**命令回复处理器**，写出到对应客户端。



47、为什么需要分布式锁？
分布式系统下，**不同的服务/客户端**通常运行在**独立的 JVM 进程**上。如果**多个 JVM 进程共享同一份资源**的话，使用**本地锁无法**实现资源的**互斥访问**。
48、如何基于 Redis 实现简易的分布式锁？
1）***SETNX*** **获取锁**：***SETNX*** 可以**实现互斥**，如果 key 不存在，才会设置 key 的值。
2）**释放锁 del 命令删除**对应的 key；使用 Lua 脚本判断 key 对应的 value 是否相等，保证原子操作的同时，防止误删其他锁。
3）怎么保证操作结束后，锁一定会释放？
避免锁无法释放，1）给锁**设置一个过期时间**；
2）设置一个专门用来监控和续期锁的**看门狗 WatchDog** 实现**自动续期**，**保证锁持续到结束**。
【注意】1）一定要**保证设置 key 值**和**过期时间**是一个**原子操作**，否则可能还没加上时间。
2）默认情况下，每过 10 秒，看门狗**检查一次线程是否持有锁**，有则**执行续期操作**，默认续期 30 秒；
3）只有**未指定锁超时时间**，**才使用 Watch Dog** 自动续期机制。
49、如何实现可重入锁？
手动的话可以使用 **hash 数据结构**设计锁，存储**持有锁的线程 ID** 和**计数器值**；计数器>0，锁被占有，通过**判断线程 ID** 确定**是不是同一个线程**，再将计数器++/--。
50、基于 Redis 实现分布式锁
1）尝试获取锁；
2）**判断 ttl** 是否为 null；
3）null 则能获取锁，并**设置锁的时间**；
如果没有设置锁时间，开启 WatchDog；
5）ttl 不为 null，**判断剩余等待时间**是否>0；
6）<0 则不能超时重试了
7）>0 则**订阅并等待别人释放锁**的信号；
8）再次判断是否超过等待时间；
9）未超时，则**再次重试获取锁**。



51、redis 如何解决集群中分布式锁的可靠性？
【问题】**主从同步有异步延迟**；master 节点获**取锁后宕机**了，slave 节点中**对应锁并未被获取**，升级为 master 后导致**误判**。
【解决】Redisson 的 **multiLock**：
多个独立的 Redis 主节点，必须在所有主节点**都获取重入锁**，才算获取锁成功。
当某个 master 宕机，它的 slave 升级为 master，但还没同步，则获取锁失败。