2020 面试题总结——Redis 篇

1、Redis 的优势。

- 速度快，因为数据存在内存中，类似于 HashMap，HashMap 的优势就是查找和操作的时间复杂度都是 O(1)。
- 支持丰富的数据类型，支持 string，list，set，zset 和 hash。
- 支持事务性。操作都是原子性，所谓的原子性就是对数据的更改要么全部执行，要么全部不执行。（这块需要注意与之前 MySQL 不同）
- 丰富的特性，可用于缓存，消息队列，按 key 设置过期时间，过期后将自动删除。

2、Redis 的数据结构都有哪些。

Redis 支持五种 Value Type，其底层实现的编码数据结构有 8 种：

- SDS - simple synamic string - 支持自动动态扩容的字节数组
- list - 平平无奇的链表
- dict - 使用双哈希表实现的, 支持平滑扩容的字典
- zskiplist - 附加了后向指针的跳跃表
- intset - 用于存储整数数值集合的自有结构
- ziplist - 一种实现上类似于 TLV, 但比 TLV 复杂的, 用于存储任意数据的有序序列的数据结构
- quicklist - 一种以 ziplist 作为结点的双链表结构
- zipmap - 一种用于在小规模场合使用的轻量级字典结构

衔接"底层数据结构"与"Value Type"的桥梁的, 则是 Redis 实现的另外一种数据结构: redisObject.。

Redis 中的 Key 与 Value 在表层都是一个 redisObject 实例, 故该结构有所谓的"类型", 即是 ValueType. 对于每一种 Value Type 类型的 redisObject, 其底层至少支持**两种**不同的底层数据结构来实现. 以应对在不同的应用场景中, Redis 的运行效率, 或内存占用.

对于具体数据结构介绍，建议看《Redis 设计与实现》第一部分，或者看这篇文章，面试官：你看过 Redis 数据结构底层实现吗？

3、Redis 的使用要注意什么。

现阶段主要从数据存储和数据获取两个方面来说明开发时的注意事项：

数据存储：因为内存空间的局限性，注定了能存储的数据量有限，如何在有限的空间内存储跟多的数据信息是我们应该关注的。Redis 内存存储的都是键值对，那么如何减少键值对所占据的内存空间就是空间优化的本质。比如可以在清晰表达业务含义的基础上尽可能缩减 key 的字符长度，也可以当 value 是图片、大文本等大数据信息时，借助压缩工具压缩之后再存入 Redis 中。

数据查询：Redis 是一种数据库，和其他数据库一样，操作时也需要有连接对象，连接对象的创建和销毁也需要耗费资源，复用连接对象很有必要，所以推荐使用连接池来进行管理（比如在自己项目中用 JedisPool 来获取 Redis 连接）。此外，对于连续多次的查询，可以使用 mget（针对 string 类型查询）/hmget（针对 Hash 类型查询），将多次请求合并为一次，提高响应速度。

4、Redis 的事务性。

Redis 通过 MULTI、EXEC、WATCH 等命令来实现事务（transaction）功能。事务提供了一种将多个命令请求打包，然后一次性、按顺序地执行多个命令的机制，并且在事务执行期间，服务器不会中断事务而改去执行其他客户端的命令请求，它会将事务中的所有命令都执行完毕后，然后才去处理其他客户端的命令请求。

事务的生命周期为：

- 事务开始：使用 MULTI 开启一个事务（自己项目中，采用 jedis.multi() 来返回一个事务 Transaction，后续可以在此事务上进行操作）
- 命令入队：在开启事务的时候，每次操作的命令将会被插入到一个队列中，同时这个命令并不会被真的执行。
- 事务执行：EXEC 命令进行提交事务

Redis 事务具有的性质：

- 单独的隔离操作：事务中所有命令都会被序列化、按顺序执行，在执行过程中不会被其他客户端发送来的命令打断。
- 没有隔离级别的概念：队列中的命令在事务没有被提交之前不会被实际执行。

在 Redis 中，事务总是具有原子性、一致性和隔离性。当 Redis 运行在某种特定的持久化模式（开启 AOF 和 RDB 服务）下时，事务也具有持久性。

着重讲一下原子性。

对于 Redis 的事务功能来说，事务队列中的命令要么就全部执行，要不就一个都不执行。从这点来说，事务具有原子性。但这个执行失败的条件是指**命令入队出错**（比如命令不存在，格式不正确等情况）而被服务器拒绝执行，而不是命令实际执行时出错。

Redis 的事务与传统的关系式数据库事务的最大区别在于，Redis 不支持事务回滚机制（rollback），即使事务队列中的某个命令在执行期间出现了错误，整个事务也会继续执行下去，直到将事务队列中的所有命令都执行完毕为止。 一定注意，Redis 的事务性与常见的关系式数据库有些不同（尤其原子性），建议直接去看《Redis 设计与实现》的 19.3 小节-事务的 ACID 性质，网上各种博客说的参差不齐。

5、当前 Redis cluster 集群有哪些方式，各自优缺点，场景。

Redis 集群是 Redis 提供的分布式数据库方案，集群通过分片来进行数据共享，并提供复制和故障转移功能。

Redis 集群使用数据分片（sharding）而非一致性哈希（consistency hashing）来实现：一个 Redis 集群包含 16384 个哈希槽(hash slot)，数据库中的每个键都属于这个 16384 个哈希槽的其中一个，集群使用公式 CRC16(key) % 16384 来计算键 key 属于哪个槽，其中 CRC16(key)语句用于计算键 key 的 CRC16 校验和。

- 数据共享：Redis 提供多个节点实例间的数据共享，也就是 Redis A、B、C、D 彼此之间的数据是同步的，同样彼此之间也可以通信，而对于客户端操作的 keys 是由 Redis 系统自行分配到各个节点中。
- 主从复制：Redis 的多个实例间通信时，一旦其中一个节点故障，那么 Redis 集群就不能继续正常工作了，所以需要一种复制机制（Master-Slave）机制，做到一旦节点 A 故障了，那么其从节点 A1 和 A2 就可以接管并继续提供与 A 同样的工作服务。
- 哈希槽值：Redis 集群中使用哈希槽来存储客户端的 keys，而在 Redis 中，目前存有 16384（2 的 14 次方）个哈希槽，它们被全部分配给所有的节点。

参考链接：Redis 集群使用总结（一）

6、Memcache 的原理，哪些数据适合放在缓存中。

Memcache 采用键值对存储方式。它本质是一个大的 hash 表，key 的最大长度为 255 个字符，最长过期时间为 30 天。它的内存模型如下：Memcache 预先将可支配的内存空间进行分区（Slab），每个分区里再分为多个块（Chunk）最大 1M，但同一个分区中块的大 小是固定的。然后，插入数据时，会根据数据大小寻找最合适的块，然后插入，当然这样也就会有部分内存浪费，但可一定程度上减少内存碎片，总体上，利大于弊。

应用场景主要是分布式应用，数据库前段缓存和服务期间数据共享等。

7、Redis 相比 memcached 有哪些优势？两者的主要区别？

- memcached 所有的值均是简单的字符串，Redis 作为其替代者，支持更为丰富的数据类型
- Redis 的速度比 memcached 快很多
- Redis 可以持久化其数据
- Redis 支持数据的备份，即 master-slave 模式的数据备份。

8、Redis 的并发竞争问题如何解决，了解 Redis 事务的 CAS 操作吗？

Redis 的并发竞争问题主要是在并发写竞争上。体现在多客户端同时并发写一个 key，修改值之后再写回去，只要顺序错了，数据就错了。

为了避免这个问题，我们可以对客户端读写 Redis 操作采用内部锁 synchronized。但解决这个问题最好的方案是 Redis 自己提供的 CAS 类的乐观锁方案。

redis 具有高级事务 CAS（乐观锁），可以被用作分布式锁。毕竟 JVM 提供的 synchronized 或者 ReentrantLock 不能应用于分布式环境下。

watch 指令在 redis 事物中提供了 CAS 的行为。为了检测被 watch 的 keys 在是否有多个 clients 同时改变引起冲突，这些 keys 将会被监控。如果至少有一个被监控的 key 在执行 exec 命令前被其他客户端修改，整个事务将会回滚，不执行任何动作，从而保证原子性操作，并且执行 exec 会得到 null 的回复。

具体工作机制：watch 命令会监视给定的每一个 key，当 exec 时如果监视的任一个 key 自从调用 watch 后发生过变化，则整个事务会回滚，不执行任何动作。注意 watch 的 key 是对整个连接有效的，事务也一样。如果连接断开，监视和事务都会被自动清除。

参考连接：redis 的高级事务 CAS(乐观锁) 和 Redis 实现 CAS 的乐观锁

9、Redis 适合于哪些场景。

- Session 共享（单点登陆）
- 页面缓存
- 队列（比如项目中用到的异步队列）
- 排行榜/计数器
- 发布/订阅（实现消息流）

10、Redis 持久化的机制，AOF 和 RDB 的区别。

Redis 提供两种方式进行持久化，一种是 RDB 持久化（会按照配置的指定时间将内存中的数据快照到磁盘中，创建一个 dump.rdb 文件，Redis 启动时再恢复到内存中），另一种是 AOF 持久化（以日志的形式记录每个写操作（读操作不

记录），只需追加文件但不可以改写文件，Redis 启动时会根据日志从头到尾全部执行一遍以完成数据的恢复工作）。

两者的区别在于：

- RDB 持久化是指在指定的时间间隔内将内存中的数据集快照写入磁盘，实际操作过程是 fork 一个子进程，先将数据集写入临时文件，写入成功后，再替换之前的文件，用二进制压缩存储。
- AOF 持久化以日志的形式记录服务器所处理的每一个写、删除操作，查询操作不会记录，以文本的方式记录，可以打开文件看到详细的操作记录。

需要注意的是，默认情况下，Redis 选用的是快照 RDB 的持久化方式，将内存中的数据以快照的方式写入二进制文件中，默认的文件名为 dump.rdb。

RDB 方式不能完全保证数据持久化，因为是定时保存，所以当 redis 服务 down 掉，就会丢失一部分数据，而且数据量大，写操作多的情况下，会引起大量的磁盘 IO 操作，会影响性能。所以，当 RDB 和 AOF 方式都开启的情况下，服务器会优先使用 AOF 文件来还原数据库状态，当然，AOF 恢复数据速度要慢一些。

还一点需要注意，服务器在载入 RDB 文件期间，会一直处于阻塞状态，直到载入工作完成为止。

11、Redis 对象的内存回收。

Redis 在自己的对象系统中构建了一个引用计数技术实现的内存回收机制，通过这一机制，程序可以通过跟踪对象的引用计数信息，在适当的时候自动释放对象并进行内存回收。

每个对象的引用计数信息是由 redisObject 结构的 refcount 属性记录。与 jvm 中的应用计数法很相似，不在赘述。

Redis 对象在整个生命周期可以划分为创建对象、操作对象、释放对象三个阶段。

12、知道哪些 Redis 的优化操作。

- 使用简短的 key。
- 大的数据压缩后再存入 value。
- 设置 key 有效期。
- 选择回收策略。当 Redis 的实例空间被填满了之后，将会尝试回收一部分 key。在 Redis 中，允许用户设置最大使用内存大小 server.maxmemory，当 Redis 内存数据集大小上升到一定大小的时候，就会施行数据淘汰策略，有很多不同的回收策略。

- 在服务器端使用 Lua 脚本。（没看过 Lua 就算了，比如我自己）

13、Redis 的主从复制机制原理。

主从的意义：

- redis 要达到高可用、高并发，只有单个 redis 是不够的，单个 redis 也就只能支持几万的 QPS，所以必须以集群的形式提供服务，而集群中又以多个主从组成。
- 主从是以多个 redis 集合在一起，以一个 master 多个 slave 为模式对外提供服务，master 主要以写为主，slave 提供读，即是读写分离的情况，以读多写少为准，如果写比较多的情况一般就以异步的形式提供服务。

主从复制功能分为两个阶段：

- 同步操作：将从服务器的数据库状态更新至主服务器当前所处的数据库状态。
- 命令传播：用于在主服务器的数据库状态被修改，导致主从服务器的数据库状态出现不一致时，让主从服务器的数据库重新回到一致状态。

Redis2.8 版本之前同步操作采用 SYNC 命令，只有全量同步，效率比较低。2.8 版本之后使用 PSYNC 命令代替 SYNC 命令来执行复制时的同步操作，自行判断是全量同步还是增量同步（通过复制偏移量、复制积压缓冲区（其实就是一个 FIFO 的队列）和服务器运行 ID 三个部分来实现），效率较高。

在命令传播阶段，从服务器默认会以每秒一次的频率，向主服务器发送命令：REPLCONF ACK <replication_offset>; 其中 replication_offset 是从服务器当前的复制偏移量。

心跳检测对主从服务器有三个作用：

- 检测主从服务器的网络连接状态。
- 辅助实现 min-slaves 选项。
- 检测命令丢失，通过对比主从服务器的复制偏移量。

参考来源：《Redis 设计与实现》的 15 章-复制

14、Redis 的线程模型是什么。

Redis 基于 Rector 模型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，但通过与 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。

15、Redis 中 set 和 zset 的区别。

set 是 Redis 下的无序集合对象，是通过 intset 或者 hashtable 编码实现。

zset 是 Redis 下的有序集合对象，是通过 ziplist 或者 skiplist 编码实现。

ziplist 编码的有序集合对象使用压缩列表作为底层实现，每个集合元素使用两个紧挨在一起的压缩列表节点来保存，第一个节点保存元素的成员，第二个节点则保存元素的分值。对于 skiplist 编码实现，它同时内部包含一个字典和跳跃表，程序都可以用 O(log(N))的复杂度往集合中添加成员，并可以用 O(1)的复杂度查找给定成员的分值。

16、分布式使用场景（存储 session）

还没遇到过

17、怎么保证缓存和数据库的一致性。

只要用缓存，就可能会涉及到缓存与数据库双存储双写，只要是双写，就一定会有数据一致性的问题。

一般来说，如果允许缓存可以稍微的跟数据库偶尔有不一致的情况，也就是说如果你的系统不是严格要求 “缓存+数据库” 必须保持一致性的话，最好不要做这个方案，即：读请求和写请求串行化，串到一个内存队列里去。

串行化可以保证一定不会出现不一致的情况，但是它也会导致系统的吞吐量大幅度降低，用比正常情况下多几倍的机器去支撑线上请求。

最经典的缓存+数据库读写的模式，就是 Cache Aside Pattern。

- 读的时候，先读缓存，缓存没有的话，就读数据库，然后取出数据后返回缓存，同时返回响应。
- 更新的时候，先更新数据库，然后再删除缓存。

为什么是删除缓存，而不是更新缓存呢。原因有二，一是在复杂缓存场景中，缓存不单单是数据库中直接取出来的值；二是更新缓存的代价是很高的。举个

例子，一个缓存涉及的表的字段，在 1 分钟内 就修改了 20 次，或者是 100 次，那么缓存更新 20 次、100 次；但是这个缓存在 1 分钟内只被读取了 1 次，有大量的冷数据。实际上，如果你只是删除缓存的话，那么在 1 分钟内，这个缓存不过就重新计算一次而已，开销大幅度降低，用到缓存才去算缓存。

其实删除缓存，而不是更新缓存，就是一个 lazy 计算的思想，不要每次都重新做复杂的计算，不管它会不会用到，而是让它到需要被使用的时候再重新计算。

在这个读写模式下，也会出现数据不一致问题。

最初级的数据不一致问题分析

问题：先修改数据库，再删除缓存。如果删除缓存失败了，那么会导致数据库中是新数据，缓存中是旧数据，数据就出现了不一致。

解决方案：先删除缓存，再修改数据库。如果数据库修改失败了，那么数据库中是旧数据，缓存中是空的，那么数据不会不一致。因为读的时候缓存没有，则读数据库中旧数据，然后更新到缓存中。

比较复杂的数据不一致问题分析

问题：数据发生了变更，先删除了缓存，然后要去修改数据库，此时还没修改。一个请求过来，去读缓存，发现缓存空了，去查询数据库，查到了修改前的旧数据，放到了缓存中。随后数据变更的程序完成了数据库的修改。在这个场景下，数据库和缓存中的数据不一样了。

针对这个方法有一种解决方案，叫做**延时双删策略**。伪代码如下：

```
public void write(String key,Object data){
redis.delKey(key);
db.updateData(data);
Thread.sleep(1000);
redis.delKey(key);
}
```

转换为中文描述为：

- 先淘汰缓存。
- 再写数据库（这两步与原来一样）。
- 休眠一秒，再次淘汰缓存。这样做可以将 1 秒内所造成的缓存脏数据再次删除。

其他主要的队列解决方案：

- 更新数据的时候，根据数据的唯一标识，将数据操作请求，发送到一个 jvm 内部队列中。读取数据的时候，如果发现数据不在缓存中，那么将重新读取数据+更新缓存的操作，根据唯一标识路由之后，也发送同一个 jvm 内部队列中。

- 一个队列对应一个工作线程，每个工作线程串行拿到对应的操作，然后一条一条的执行。这样的话，一个数据变更的操作，先删除缓存，然后再去更新数据库，但是还没完成更新。此时如果一个读请求过来，读到了空的缓存，那么可以先将缓存更新的请求发送到队列中，此时会在队列中积压，然后同步等待缓存更新完成。
- 这里有一个优化点，一个队列中，其实多个更新缓存请求串在一起是没意义的，因此可以做过滤，如果发现队列中已经有一个更新缓存的请求了，那么就不用再放个更新请求操作进去了，直接等待前面的更新操作请求完成即可。

在这个方法下，需要注意读缓存请求超时问题，每个读请求必须在超时时间内返回。如果超时还未读到更新后缓存的话，则直接从数据库读旧的值。

另外，保持读写一致性还可以用其他方法来做。比如 **MySQL 读写分离，或者 Redis 分布式锁**。

参考文章：一个高频面试题：怎么保证缓存与数据库的双写一致性？

18、Redis 为什么用 skiplist 而不用平衡树。

Redis 里面使用跳跃表 skiplist 实现 sorted set 有序集合（实际上还有散列表），而没有采用红黑树、平衡树这种结构。原因主要有以下几点：

- 跳表使用空间换时间的设计思路，通过构建多级索引来提高查询的效率，实现了基于链表的“二分查找”。
- 跳表是一种动态数据结构，支持快速的插入、删除、查找操作，时间复杂度都是 O(logn)。并且范围查找比红黑树高效。跳表的空间复杂度是 O(n)。不过，
- 跳表的实现非常灵活，可以通过改变索引构建策略，有效平衡执行效率和内存消耗。虽然跳表的代码实现并不简单，但是作为一种动态数据结构，比起红黑树来说，实现要简单多了。所以很多时候，我们为了代码的简单、易读，比起红黑树，我们更倾向用跳表。

参考文章：跳表：为什么 Redis 一定要用跳表来实现有序集合？

19、Redis 分布式锁的实现方式。

Redis 分布式锁实现方式有以下几种。

第一种，使用 redis 的 watch 命令进行实现。

watch 指令在 redis 事物中提供了 CAS 的行为。为了检测被 watch 的 keys 在是否有多个 clients 同时改变引起冲突，这些 keys 将会被监控。如果至少有一个被监控的 key 在执行 exec 命令前被其他客户端修改，整个事务将会回滚，不执行任何动作，从而保证原子性操作，并且执行 exec 会得到 null 的回复。

具体工作机制：watch 命令会监视给定的每一个 key，当 exec 时如果监视的任一个 key 自从调用 watch 后发生过变化，则整个事务会回滚，不执行任何动作。注意 watch 的 key 是对整个连接有效的，事务也一样。如果连接断开，监视和事务都会被自动清除。

第二种，使用 redis 的 setnx 命令进行实现。

先看一下这个相关的命令。

```
SETNX key value
```

如果 key 不存在，就设置 key 对应字符串 value。在这种情况下，该命令和 SET 一样。当 key 已经存在时，就不做任何操作。SETNX 是“SET if Not eXists”。

```
expire KEY seconds
```

设置 key 的过期时间。如果 key 已过期，将会被自动删除。

```
del KEY
```

删除 key

由于当某个 key 不存在的时候，SETNX 才会设置该 key。且由于 Redis 采用单进行单线程模型，所以，不需要担心并发问题。那么，就可以利用 SETNX 的特性维护一个 key，存在的时候，即锁被某个线程持有；不存在的时候，没有线程持有锁。

并且还可以设置 key 的过期时间当作锁的超时时间，释放锁就直接可以将 key 删除即可。

20、Redis 遇到的问题和缺点。

- Redis 不具备自动容错和恢复功能，主机从机的宕机都会导致前端部分读写请求失败，需要等待机器重启或者切换数据库才能恢复。
- Redis 主从复制过程中，第一个步骤是同步，需要采用全量复制，复制过程中主机会 fork 出一个子进程对内存做一份快照，并将子进程的内存快照保存为文件发送给从机，这一过程需要确保主机有足够多的空余内存。若快照文件较大，对集群的服务能力会产生较大的影响。
- Redis 作为缓存的话，还会出现缓存和数据库双写一致性的问题。

21、Redis 各个数据类型的使用场景。

Redis 支持物种数据类型：string（字符串）、hash（哈希）、list（列表）、set（集合）及 zset（有序集合）。

使用场景分别为：

- **String：**是简单的 key-value 类型，value 其实不仅可以是 String，也可以是数字。常规 key-value 缓存应用；常规计数：微博数，粉丝数等。
- **hash：**是一个 string 类型的 field 和 value 的映射表，hash 特别适合用于存储对象。存储部分变更的信息，比如说用户信息等。
- **list：**是一个链表。可以轻松实现最新消息排行榜等功能。另外一个应用就是可以实现一个消息队列。可以利用 List 的 PUSH 操作，将任务存在 List 中，然后工作线程再用 POP 操作将任务取出进行执行。也可以通过 zset 构建有优先级的队列系统。此外，还可以将 redis 用作日志收集器，实际上还是一个队列，多个端点将日志信息写入 redis，然后一个 worker 统一将所有日志写到磁盘。
- **set：**是一个没有重复值得集合。可以存储一些集合性的数据。在微博应用中，可以将一个用户所有的关注人存在一个集合中，将其所有粉丝存在一个集合中。Redis 还为集合提供了求交集、并集、差集等操作，可以非常方便的实现如共同关注、共同喜好等功能。
- **zset：**相比 set，zset 增加了一个权重参数 score，使得集合中的元素能够按 score 进行有序排列。比如一个存储全班同学成绩的 sorted set，其集合 value 可以是同学的学号，而 score 就可以是其考试得分，这样在数据插入集合的时候，就已经进行了天然的排序。也可以利用 zset 设计带有优先级的队列。另外，还可以做排行榜应用，取 TOP N 操作。

22、Redis 数据淘汰策略。

Redis 提供了五种数据淘汰策略：

- volatile-lru：使用 LRU 算法进行数据淘汰（淘汰上次使用时间最早的，且使用次数最少的 key），只淘汰设定了有效期的
- keyallkeys-lru：使用 LRU 算法进行数据淘汰，所有的 key 都可以被淘汰
- volatile-random：随机淘汰数据，只淘汰设定了有效期的 key
- allkeys-random：随机淘汰数据，所有的 key 都可以被淘汰
- volatile-ttl：淘汰剩余有效期最短的 key

此外，如果不设置如上策略的话，还有一种 noeviction 策略，当内存限制达到，谁也不删除，返回错误。

23、Redis 哈希槽的概念。

Redis 集群没有使用一致性 hash,而是引入了哈希槽的概念，Redis 集群有 16384 个哈希槽，每个 key 通过 CRC16 校验后对 16384 取模来决定放置哪个槽，集群的每个节点负责一部分 hash 槽。

24、Redis 的缓存雪崩。

缓存雪崩，简单的理解就是：由于原有缓存失效（或者数据未加载到缓存中），新缓存未到期间（缓存正常从 Redis 中获取），所有原本应该访问缓存的请求都去查询数据库了，而对数据库 CPU 和内存造成巨大压力，严重的会造成数据库宕机，造成系统的崩溃。对此，基本的解决思路有：

- 考虑采用加锁或者队列的方式保证不会同时有大量的线程对数据库一次性进行读写，避免缓存失效时对数据库造成太大的压力，虽然能够一定的程度上缓解了数据库的压力，但是也降低了系统的吞吐量。
- 分析用户的行为，不同的 key 设置不同的过期时间，尽量让缓存失效的时间均匀分布。
- 做二级缓存，或者双缓存策略。A1 为原始缓存，A2 为拷贝缓存，A1 失效时，可以访问 A2，A1 缓存失效时间设置为短期，A2 设置为长期。

具体方法如下：

- 事发前：实现 Redis 的高可用(主从架构+Sentinel 或者 Redis Cluster)，尽量避免 Redis 挂掉这种情况发生。
- 事发中：万一 Redis 真的挂了，我们可以设置本地缓存(ehcache)+限流(hystrix)，尽量避免我们的数据库被干掉(起码能保证我们的服务还是能正常工作的)
- 事发后：redis 持久化，重启后自动从磁盘上加载数据，快速恢复缓存数据。

25、Redis 的缓存穿透。

缓存穿透是指用户查询数据时，在数据库中没有，自然在缓存中也不会有。这样就导致用户查询的时候，在缓存中找不到，每次都要去数据库中查询。查询一个必然不存在的数据。比如文章表，查询一个不存在的 id，每次都会访问 DB，如果有人恶意破坏，很可能直接对 DB 造成影响。对此，基本的解决思路有：

- 如果查询数据库为空，直接设置一个默认值存放到缓存中，这样第二次缓存中获取就有值了，而不会继续访问数据库，这种方法最简单粗暴。
- 根据缓存数据 key 的规则进行过滤，比如说缓存 Key 为 mac 地址。这就要求 key 必须有以顶的规则，这种方法可以缓解一部分的压力，但是无法根治。
- 采用布隆过滤器，将所有可能存在的数据哈希到一个足够大的 BitSet 中，不存在的数据将会被拦截掉，从而避免了对底层存储系统的查询压力。对于布隆过滤器，可以用 BitSet 来构建。

具体方法如下：

由于请求的参数是不合法的(每次都请求不存在的参数)，于是我们可以使用布隆过滤器(BloomFilter)或者压缩 filter 提前拦截，不合法就不让这个请求到数据库层！

当我们从数据库找不到的时候，我们也将这个空对象设置到缓存里边去。下次再请求的时候，就可以从缓存里边获取了。这种情况我们一般会将空对象设置一个较短的过期时间。

26、Redis 的 SDS 相比 char[]的优点。

SDS 结构中拥有 len（字符串长度）、free（未使用字节的数量）和 buf 数组（保存字符串）。具有以下几点优势。

- 常数复杂度获取字符串长度。
- 杜绝缓冲区溢出。
- 减少修改字符串时带来的内存重分配次数，因为有空间预分配和惰性空间释放。