# Redis知识点

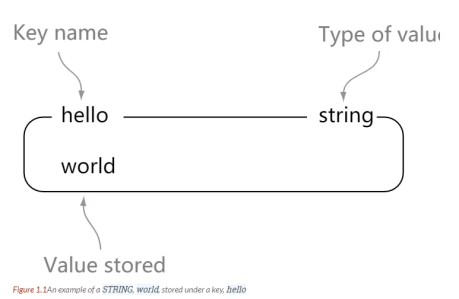
## 一、概述

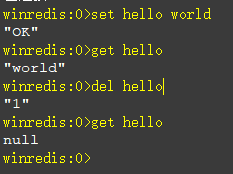
Redis是速度非常快的非关系（NoSQL）内存键值型数据库，键只允许String类型，值有五种分别是：字符串、列表、集合、散列表、有序集合。Redis支持很多特性，如内存持久化到硬盘，使用复制来扩展读性能，使用分片来扩展写性能。

## 数据类型

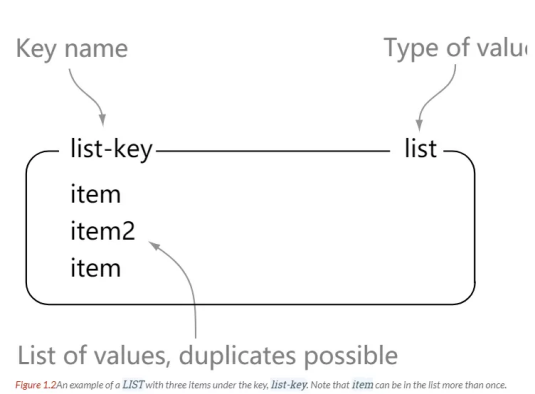
|  |  |  |
| --- | --- | --- |
| 数据类型 | 可存储值 | 操作 |
| String | 字符串、整数、点浮数 | 对整个字符串或者字符串的其中一部分执行操作  对整数和浮点数执行自增或者自减操作 |
| List | 列表 | 从两端压入或者弹出元素  对单个或者多个元素  进行修剪，只保留一个范围内的元素 |
| Set | 集合 | 添加、获取、移除单个元素  检查一个元素是否存在于集合中  计算交集、并集、差集  从集合里面随机获取元素 |
| Hash | 散列表 | 添加、获取、移除单个键值对  获取所有键值对  检查某个键是否存在 |
| ZSet | 有序集合 | 添加、获取、删除元素  根据分值范围或者成员来获取元素  计算一个键的排名 |

STRING：

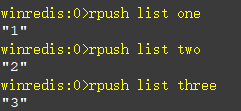




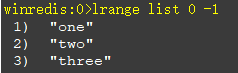
LIST：



存储



查询全部



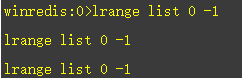
索引查询



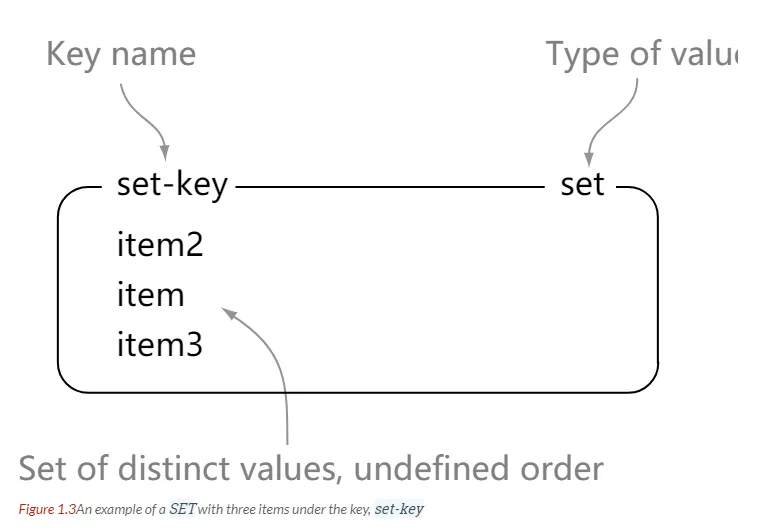
出栈



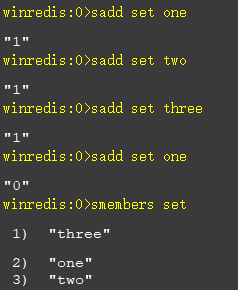
出栈后查询全部



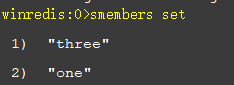
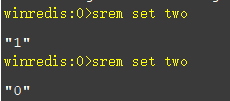
SET



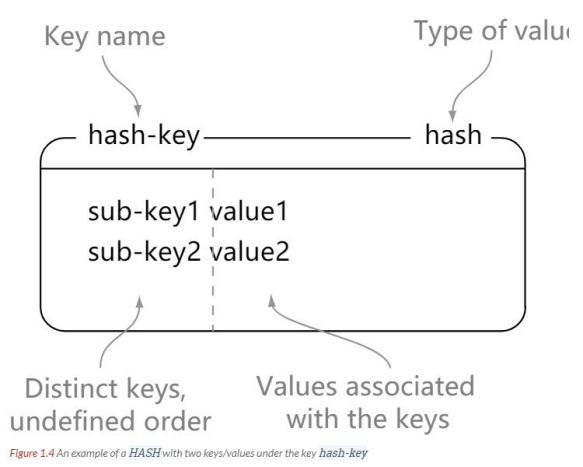
存储和查询全部

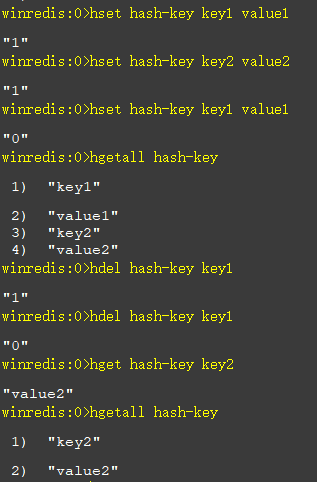


取出数据后查询

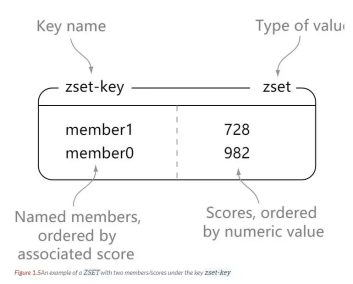


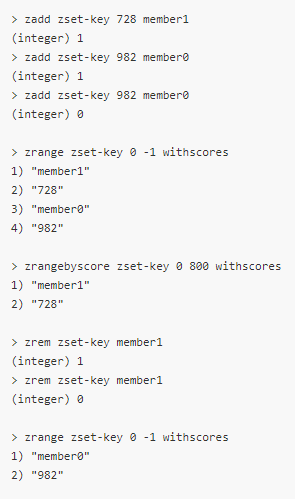
HASH





ZSET





## 使用场景

1. **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基于内存存储读写性能很高，很适合频繁存取的计数量。

1. **缓存：**

将热点数据存储于缓存之中，设置缓存最大使用量，以及淘汰策略来保证缓存命中率。

1. **查找表：**

例如DNS记录很适合用Redis进行存储

查找表和缓存类似，也可以利用Redis查找速率快的特性，但是查找表的数据不能失效， 而缓存内容可以失效，因为缓存不做为可靠数据来源

1. **消息队列：**

List是一个双向链表，可以通过lpush和rpop读写消息。

不过建议使用Kafka、Rabbit MQ的消息队列实现

1. **分布式锁实现：**

在分布式场景下，无法使用单机环境下的锁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，也可以使用官方的RedLock分布式锁 实现

1. **会话缓存**：

可以使用Redis来缓存多台应用服务器的会话信息。

当服务器不在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及可伸缩性。

1. **其他：**

Set 可以实现交集、并集等操作，从而实现共同好友等功能。

ZSet 可以实现有序性操作，从而实现排行榜等功能。

## Redis与memcached

1. **数据类型**

Memcached 仅支持字符串类型，而 Redis 支持五种不同的数据类型，可以更灵活地解 决问题。

1. **数据持久化**

Redis 支持两种持久化策略：RDB 快照和 AOF 日志，而 Memcached 不支持持久化。

1. **分布式**

Memcached 不支持分布式，只能通过在客户端使用一致性哈希来实现分布式存储，这种方式在存储和查询时都需要先在客户端计算一次数据所在的节点。

Redis Cluster 实现了分布式的支持。

1. **内存管理机制**

在 Redis 中，并不是所有数据都一直存储在内存中，可以将一些很久没用的 value 交换到磁盘，而 Memcached 的数据则会一直在内存中。

Memcached 将内存分割成特定长度的块来存储数据，以完全解决内存碎片的问题。但是这种方式会使得内存的利用率不高，例如块的大小为 128 bytes，只存储 100 bytes 的数据，那么剩下的 28 bytes 就浪费掉了。

## 键过期时间

Redis 可以为每个键设置过期时间，当键过期时，会自动删除该键。

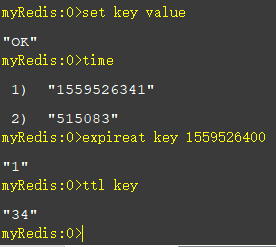
对于散列表这种容器，只能为整个键设置过期时间（整个散列表），而不能为键里面的单个元素设置过期时间。

四种不同设置：

expire <key> <ttl> :key是需要设置时间的的键，ttl是存活时间单位是秒

pexpire <key> <ttl> :ttl是存活时间单位是毫秒

expireat <key> <ttl>:使用时间戳单位是秒 如下图：



pexpireat <key> <ttl>:单位毫秒

## 数据淘汰策略

可以设置内存最大使用量，当内存使用量超出时，会施行数据淘汰策略。

Redis具有六种淘汰策略：

|  |  |
| --- | --- |
| 策略 | 描述 |
| volatile-lru | 从设置过期时间的数据集中挑选最近最少使用的数据淘汰 |
| volatile-ttl | 从设置过期时间的数据集中挑选出将要过期的数据淘汰 |
| volatile-random | 从设置过期时间的数据集中任意挑选数据淘汰 |
| allkeys-lru | 从全部数据集中挑选最近最少使用的数据淘汰 |
| allkeys-random | 从所有数据集中随机挑选数据淘汰 |
| noeviction | 禁止驱逐数据 |

作为内存数据库，出于对性能和内存消耗的考虑，Redis 的淘汰算法实际实现上并非针对所有 key，而是抽样一小部分并且从中选出被淘汰的 key。

使用 Redis 缓存数据时，为了提高缓存命中率，需要保证缓存数据都是热点数据。可以将内存最大使用量设置为热点数据占用的内存量，然后启用 allkeys-lru 淘汰策略，将最近最少使用的数据淘汰。

Redis 4.0 引入了 volatile-lfu 和 allkeys-lfu 淘汰策略，LFU 策略通过统计访问频率，将访问频率最少的键值对淘汰。

## 持久化

Redis 是内存型数据库，为了保证数据在断电后不会丢失，需要将内存中的数据持久化到硬盘上。

**1.持久过过程：**

* 客户端向服务端发送写操作（数据在客户端的内存中）。
* 数据库服务端接收到写请求的数据（数据在服务端的内存中）。
* 服务端调用write这个系统调用，将数据往磁盘上写（数据在系统内存的缓冲区中）。
* 操作系统将缓冲区中的数据转移到磁盘控制器上（数据在磁盘缓存中）。
* 磁盘控制器将数据写到磁盘的物理介质中（数据真正落到磁盘上）。

**2.故障分析：**

* 当数据库系统故障时，这时候系统内核还是完好的。那么此时只要我们执行完了第3步，那么数据就是安全的，因为后续操作系统会来完成后面几步，保证数据最终会落到磁盘上。
* 当系统断电时，这时候上面5项中提到的所有缓存都会失效，并且数据库和操作系统都会停止工作。**所以只有当数据在完成第5步后，才能保证在断电后数据不丢失**。

**3.需要清楚问题：**

* 数据库多长时间调用一次write，将数据写到内核缓冲区？
* 内核多长时间会将系统缓冲区中的数据写到磁盘控制器？
* 磁盘控制器又在什么时候把缓存中的数据写到物理介质上？

　　对于第一个问题，通常数据库层面会进行全面控制。

　　而对第二个问题，操作系统有其默认的策略，但是我们也可以通过POSIX API提供的fsync系列命令强制操作系统将数据从内核区写到磁盘控制器上。

　　对于第三个问题，好像数据库已经无法触及，但实际上，大多数情况下磁盘缓存是被设置关闭的，或者是只开启为读缓存，也就是说写操作不会进行缓存，直接写到磁盘。

**建议的做法是仅仅当你的磁盘设备有备用电池时才开启写缓存**。

**4.数据损坏：**

所谓数据损坏，就是数据无法恢复，上面我们讲的都是如何保证数据是确实写到磁盘上去，但是写到磁盘上可能并不意味着数据不会损坏。比如我们可能一次写请求会进行两次不同的写操作，当意外发生时，可能会导致一次写操作安全完成，但是另一次还没有进行。如果数据库的数据文件结构组织不合理，可能就会导致数据完全不能恢复的状况出现。

**5.防止数据损坏：**

* 第一种是最粗糙的处理，就是不通过数据的组织形式保证数据的可恢复性。而是通过配置数据同步备份的方式，在数据文件损坏后通过数据备份来进行恢复。实际上MongoDB在不开启操作日志，通过配置Replica Sets时就是这种情况。
* 另一种是在上面基础上添加一个操作日志，每次操作时记一下操作的行为，这样我们可以通过操作日志来进行数据恢复。因为操作日志是顺序追加的方式写的，所以不会出现操作日志也无法恢复的情况。这也类似于MongoDB开启了操作日志的情况。
* 更保险的做法是数据库不进行旧数据的修改，只是以追加方式去完成写操作，这样数据本身就是一份日志，这样就永远不会出现数据无法恢复的情况了。实际上CouchDB就是此做法的优秀范例。

**6.两种持久化类型：**

1. **RDB持久化**

将某个时间点的所有数据都存放到硬盘上，

可以将快照复制到其他服务器从而具有相同数据的服务器副本。

如果系统发生故障，将会丢失最后一次创建快照之后的数据。

如果数据量大，保存快照的时间会很长。

默认生成文件dump.rdb

Redis默认模式是RDB触发时间如下：

save 900 1 #900秒内如果超过1个key被修改，则发起快照保存

save 300 10 #300秒内容如超过10个key被修改，则发起快照保存

save 60 10000

手动触发：save命令（同步方法save方法是同步的，没有写入完成前不执行后面 的代码）

bgsave命令（异步方法如果数据量非常大，要保存的内容很多，建议使 用bgsave，如果内容少则可以使用save方法）

1. **AOF持久化**

将写命令添加到AOF文件（append only file）的末尾。

使用AOF需要设置同步选项，从而确保写命令什么时候会同步到磁盘文件上。这是英文对文件不写入不会马上同步到磁盘上，而是先存到缓冲区，然后有操作系统决定什么时候同步到磁盘。同步选项如下：

|  |  |
| --- | --- |
| 选项 | 同步频率 |
| always | 每个命令都会同步 |
| everysec | 每一秒同步一次 |
| no | 由操作系统决定何时同步 |

always会严重降低服务器性能。

everysec选项比较合适，可以保证系统崩溃时只会丢失一秒左右的数据，并且 Redis 每秒执行一次同步对服务器性能几乎没有任何影响；

no选项并不能给服务器性能带来多大的提升，而且也会增加系统崩溃时数据丢失的数量。

随着服务器写请求的增多，AOF 文件会越来越大。Redis 提供了一种将 AOF 重写的特性，能够去除 AOF 文件中的冗余写命令。

默认生成文件appendonly.aof

默认情况是关闭的：appendonly：no

同步模式是第二种：appendfsync everysec

# appendfsync always

appendfsync everysec

# appendfsync no

## 事务

一个事务包含了多个命令，服务器在执行事务期间，不会改去执行其它客户端的命令请求。

事务中的多个命令被一次性发送给服务器，而不是一条一条发送，这种方式被称为流水线，它可以减少客户端与服务器之间的网络通信次数从而提升性能。

Redis 最简单的事务实现方式是使用 MULTI 和 EXEC 命令将事务操作包围起来。

## 数据一致性

造成的原因：在高并发情况下，读写操作同时发生时，无论是先删缓存再存数据，还是先存数据再删缓存都容易产生数据库与redis中的数据无法保持一致。

案例及解决方法：

案例一：在更新库存商品时，当库存100需要更新成99，先更新在删除缓存，当删除缓存失败时，数据库是99，缓存是100 就会导致不一致。

解决方法：先删除缓存再存数据库，当更新缓存失败时就不对数据库进行更新，这时读到的数据是数据库的旧数据，但是缓存与数据库的数据是一致的。

案例二：在高并发情况下先删除了缓存，然后还没对数据库完成更新时，进来一个读操作就又会把数据库中100库存重新读取进缓存，等数据库修改完成后数据库是99就会产生数据不一致情况。

解决方法：可以使用队列来解决，创建几个队列，如创建10个队列，然后通过商品ID做hash然后取模于队列个数，当数据请求过来时先丢进队列里去，当更新后删除，对于上面的场景，当请求读数据进来时先差了缓存发现没有数据，就先查看队列是否有相同Id商品在更新，如果有就放入队列然后同步等待缓存更新。

这里有一个优化点，如果发现队列里有一个查询请求了，那么就不要放新的查询操作进去了，用一个while（true）循环去查询缓存，循环个200MS左右，如果缓存里还没有则直接取数据库的旧数据，一般情况下是可以取到的。

在高并发下解决场景二要注意的问题

（1）读请求时长阻塞

 由于读请求进行了非常轻度的异步化，所以一定要注意读超时的问题，每个读请求必须在超时间内返回，该解决方案最大的风险在于可能数据更新很频繁，导致队列中挤压了大量的更新操作在里面，然后读请求会发生大量的超时，最后导致大量的请求直接走数据库，像遇到这种情况，一般要做好足够的压力测试，如果压力过大，需要根据实际情况添加机器。

（2）请求并发量过高

 这里还是要做好压力测试，多模拟真实场景，并发量在最高的时候QPS多少，扛不住就要多加机器，还有就是做好读写比例是多少

（3）多服务实例部署的请求路由

可能这个服务部署了多个实例，那么必须保证说，执行数据更新操作，以及执行缓存更新操作的请求，都通过nginx服务器路由到相同的服务实例上

（4）热点商品的路由问题，导致请求的倾斜

某些商品的读请求特别高，全部打到了相同的机器的相同丢列里了，可能造成某台服务器压力过大，因为只有在商品数据更新的时候才会清空缓存，然后才会导致读写并发，所以更新频率不是太高的话，这个问题的影响并不是很大，但是确实有可能某些服务器的负载会高一些。

其他方法：<https://blog.kido.site/2018/11/24/db-and-cache-preface/>

## 十、Redis为什么速度那么快

1.操作对象时内存，不需要加载到内存之中，

2.单线程操作，（没有CPU竞争时间，线程切换时间）

3.采用I/O多路复用技术，一个线程管理多个IO通道

## 十一、Redis缓存穿透，雪崩，击穿

**缓存击穿：**

缓存穿透是指查询一个一定不存在的数据，由于缓存是不命中时被动写的，并且出于容错考虑，如果从存储层查不到数据则不写入缓存，这将导致这个不存在的数据每次请求都要到存储层去查询，失去了缓存的意义。在流量大时，可能DB就挂掉了，要是有人利用不存在的key频繁攻击我们的应用，这就是漏洞。

**解决方法：**

1. **使用互斥锁：**该方法是比较普遍的做法，即，在根据key获得的value值为空时，先锁上，再从数据库加载，加载完毕，释放锁。若其他线程发现获取锁失败，则睡眠50ms后重试。至于锁的类型，单机环境用并发包的Lock类型就行，集群环境则使用分布式锁( redis的setnx)

**优点:**

思路简单

保证一致性

**缺点：**

代码复杂度增大

存在死锁的风险

1. **异步构建缓存：**在这种方案下，构建缓存采取异步策略，会从线程池中取线程来异步构建缓存，从而不会让所有的请求直接怼到数据库上。该方案redis自己维护一个timeout，当timeout小于System.currentTimeMillis()时，则进行缓存更新，否则直接返回value值。

**优点:**

性价最佳，用户无需等待

**缺点：**

无法保证缓存一致性

1. **布隆过滤法：**有很多种方法可以有效地解决缓存穿透问题，最常见的则是采用布隆过滤器，将所有可能存在的数据哈希到一个足够大的bitmap中，一个一定不存在的数据会被这个bitmap拦截掉，从而避免了对底层存储系统的查询压力。

**优点:**

思路简单

保证一致性

性能强

**缺点：**

代码复杂度增大

需要另外维护一个集合来存放缓存的Key

布隆过滤器不支持删值操作

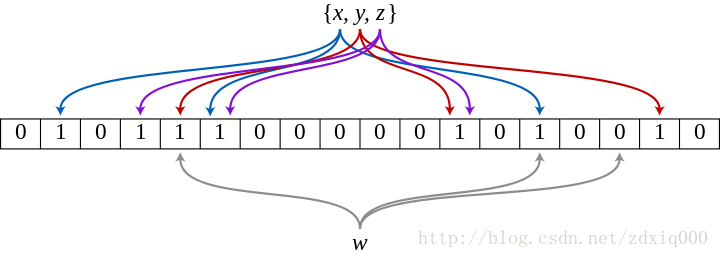
**布隆过滤：**

就是哈希算法的产物。

布隆过滤器（Bloom Filter）的核心实现是一个超大的位数组（或者叫位向量）和几个哈希函数。假设位数组的长度为m，哈希函数的个数为k

假设插入3个元素{x,y,z}集合，哈希函数有3个。

1. 初始化数组，每个位置都为零
2. 3个元素分别通过三次的哈希函数计算映射到数组上，数组标记为1
3. 查找一个数w是否在此集合之中，通过第2步的，映射到三个点上
4. 如果三个点都是1则该元素有可能在集合中，反之有一个不在就证明不在集合中。三点都为一不能说明一定因为布隆过滤有一定的错误率。其内部维护一个全为0的bit数组，需要说明的是，布隆过滤器有一个误判率的概念，误判率越低，则数组越长，所占空间越大。误判率越高则数组越小，所占的空间越小。错误率为0.03是维护数组是7298440，错误率0.01维护数组9585058



1. **暴力法**：如果一个查询返回的数据为空（不管是数据不存在，还是系统故障），我们仍然把这个空结果进行缓存，但它的过期时间会很短，最长不超过五分钟。

**缓存雪崩：**

缓存雪崩是指在我们设置缓存时采用了相同的过期时间，导致缓存在某一时刻同时失效，请求全部转发到DB，DB瞬时压力过重雪崩。

**解决方法：**

缓存失效时的雪崩效应对底层系统的冲击非常可怕。大多数系统设计者考虑用加锁或者队列的方式保证缓存的单线 程（进程）写，从而避免失效时大量的并发请求落到底层存储系统上。这里分享一个简单方案就是将缓存失效时间分散开，比如我们可以在原有的失效时间基础上增加一个随机值，比如1-5分钟随机，这样每一个缓存的过期时间的重复率就会降低，就很难引发集体失效的事件。

**缓存击穿：**

缓存击穿，是指一个key非常热点，在不停的扛着大并发，大并发集中对这一个点进行访问，当这个key在失效的瞬间，持续的大并发就穿破缓存，直接请求数据库，就像在一个屏障上凿开了一个洞。

**解决方法：**

1、使用锁，单机用synchronized,lock等，分布式用分布式锁，互斥锁。

2、缓存过期时间不设置，而是设置在key对应的value里。如果检测到存的时间超过过期时间则异步更新缓存。

3、在value设置一个比过期时间t0小的过期时间值t1，当t1过期的时候，延长t1并做更新缓存操作。

## 十二、集群

一、读多写少用缓存，写多读少用缓冲！

二、通信机制：网络通信

三、数据存储于内存，客户端-服务端的形式

四、缓存使用三个阶段：

1. 会使用----0.5年
2. 高阶使用----1（主从集群、哨兵高可用、cluster）
3. 深度认知-----3（例如在什么情况下会丢数据，如高性能）

五、为什么redis不直接用HTTP进行交互？

原因：在Redis设计中是保持高性能，网络传输HTTP需要传输很多东西，东西多服务器解析速度慢，导致性能慢。

底层通讯原理：通过TCP进行连接，RESP协议