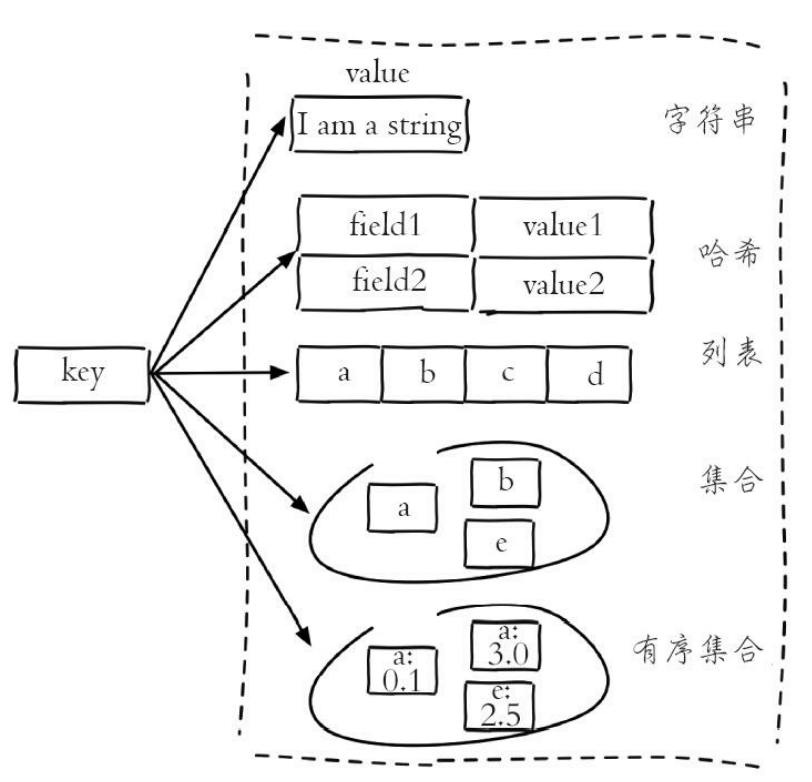
**Redis 基础数据结构**

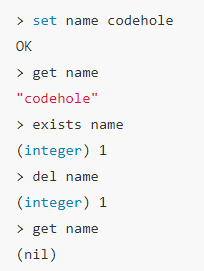
Redis 有 5 种基础数据结构，分别为：string (字符串)、list (列表)、set (集合)、hash (哈希) 和 zset (有序集合)。



**string (字符串)**

字符串 string 是 Redis 最简单的数据结构。Redis 所有的数据结构都是以唯一的 key 字符串作为名称，然后通过这个唯一 key 值来获取相应的 value 数据。不同类型的数据结构的差异就在于 value 的结构不一样。字符串结构使用非常广泛，一个常见的用途就是缓存用户信息。我们将用户信息结构体使用 JSON 序列化成字符串，然后将序列化后的字符串塞进 Redis 来缓存。同样，取用户信息会经过一次反序列化的过程。

**键值对**



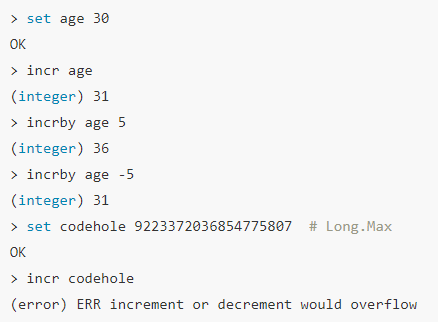
**批量键值对：**可以批量对多个字符串进行读写，节省网络耗时开销



**过期和 set 命令扩展：**可以对 key 设置过期时间，到点自动删除，这个功能常用来控制缓存的失效时间



**原子计数：**如果 value 值是一个整数，还可以对它进行自增操作。自增是有范围的，它的范围是 signed long 的最大最小值，超过了这个值，Redis 会报错



**list (列表)**

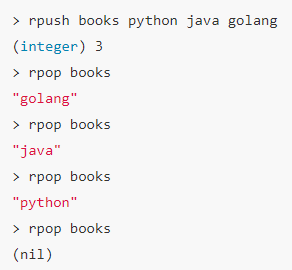
Redis 的列表相当于 Java 语言里面的 LinkedList，注意它是链表而不是数组。这意味着 list 的插入和删除操作非常快，时间复杂度为 O(1)，但是索引定位很慢，时间复杂度为 O(n)，这点让人非常意外。 当列表弹出了最后一个元素之后，该数据结构自动被删除，内存被回收。

Redis 的列表结构常用来做异步队列使用。将需要延后处理的任务结构体序列化成字符串塞进 Redis 的列表，另一个线程从这个列表中轮询数据进行处理。

**右边进左边出：队列**



**右边进右边出：栈**



**hash (字典)**

Redis 的字典相当于 Java 语言里面的 HashMap，它是无序字典。内部实现结构上同 Java 的 HashMap 也是一致的，同样的数组 + 链表二维结构。第一维 hash 的数组位置碰撞时，就会将碰撞的元素使用链表串接起来。

hash 结构也可以用来存储用户信息，不同于字符串一次性需要全部序列化整个对象，hash 可以对 用户结构中的每个字段单独存储。这样当我们需要获取用户信息时可以进行部分获取。而以整个字符串的形式去保存用户信息的话就只能一次性全部读取，这样就会比较浪费网络流量。 hash 也有缺点，hash 结构的存储消耗要高于单个字符串，到底该使用 hash 还是字符串，需要根据实际情况再三权衡。



**set (集合)**

Redis 的集合相当于 Java 语言里面的 HashSet，它内部的键值对是无序的唯一的。它的内部实现相当于一个特殊的字典，字典中所有的 value 都是一个值NULL。 当集合中最后一个元素移除之后，数据结构自动删除，内存被回收。

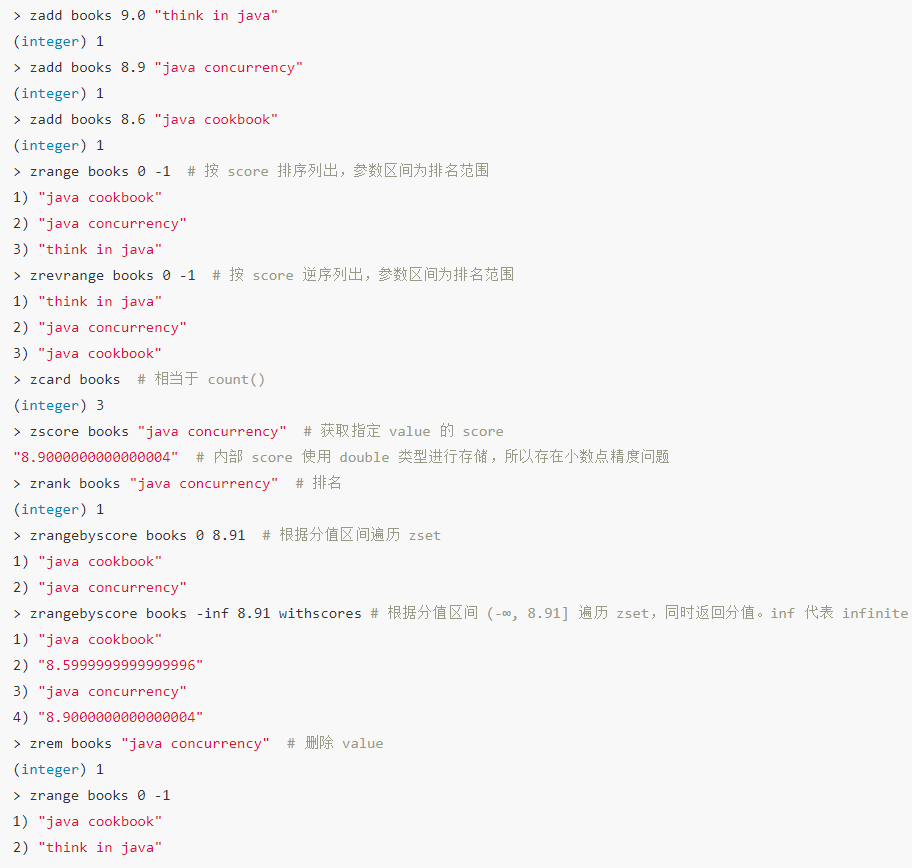


**zset (有序集合)**

zset 似于 Java 的 SortedSet 和 HashMap 的结合体，一方面它是一个 set，保证了内部 value 的唯一性，另一方面它可以给每个 value 赋予一个 score，代表这个 value 的排序权重。

zset 可以用来存粉丝列表，value 值是粉丝的用户 ID，score 是关注时间。我们可以对粉丝列表按关注时间进行排序。

zset 还可以用来存储学生的成绩，value 值是学生的 ID，score 是他的考试成绩。我们可以对成绩按分数进行排序就可以得到他的名次。

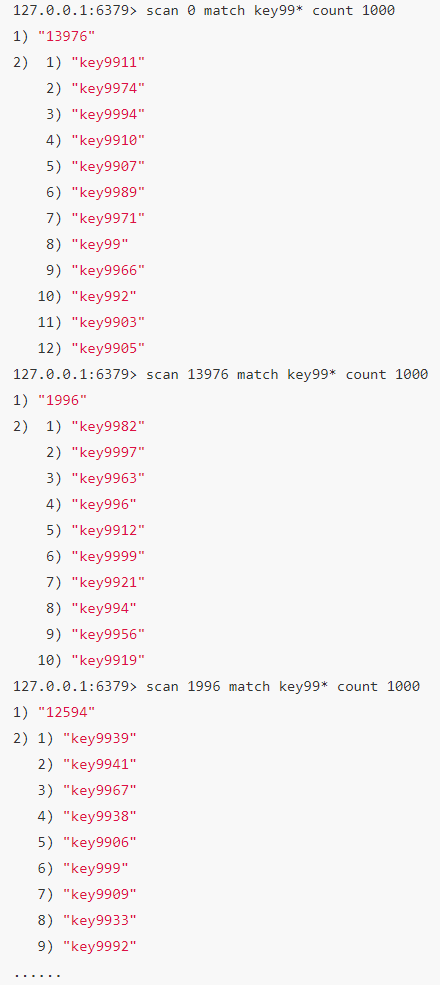


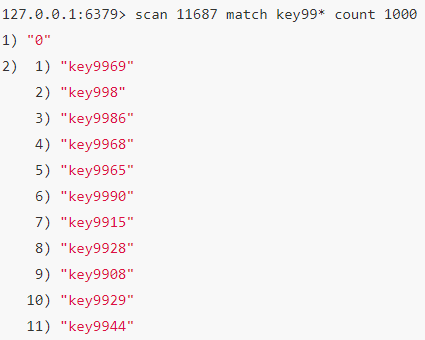
**其他高级命令**

**keys：全量遍历键**，用来列出所有满足特定正则字符串规则的key，当redis数据量比较大时，性能比较差，要避免使用

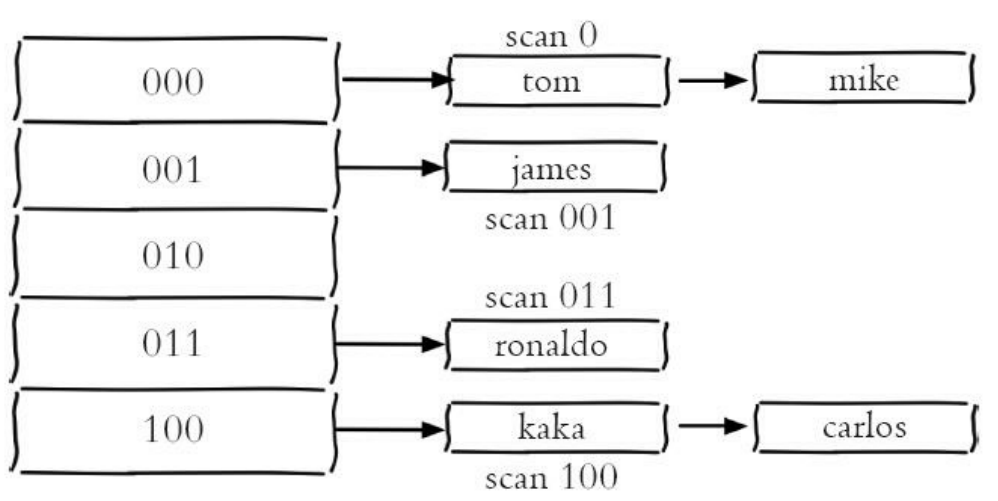


**scan：渐进式遍历键，**scan 参数提供了三个参数，第一个是 cursor 整数值，第二个是 key 的正则模式，第三个是遍历的 limit hint。第一次遍历时，cursor 值为 0，然后将返回结果中第一个整数值作为下一次遍历的 cursor。一直遍历到返回的 cursor 值为 0 时结束。





Redis存储键值对实际使用的是hashtable的数据结构



**Info：**查看redis服务运行信息，分为 9 大块，每个块都有非常多的参数，这 9 个块分别是:

Server 服务器运行的环境参数

Clients 客户端相关信息

Memory 服务器运行内存统计数据

Persistence 持久化信息

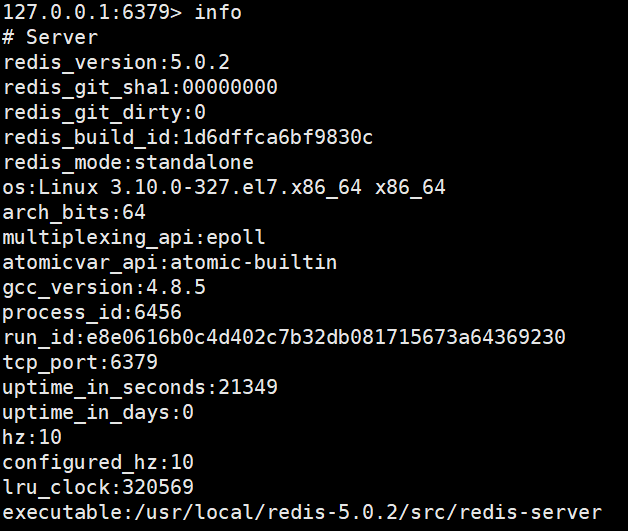
Stats 通用统计数据

Replication 主从复制相关信息

CPU CPU 使用情况

Cluster 集群信息

KeySpace 键值对统计数量信息



**核心原理**

**Redis的单线程和高性能**

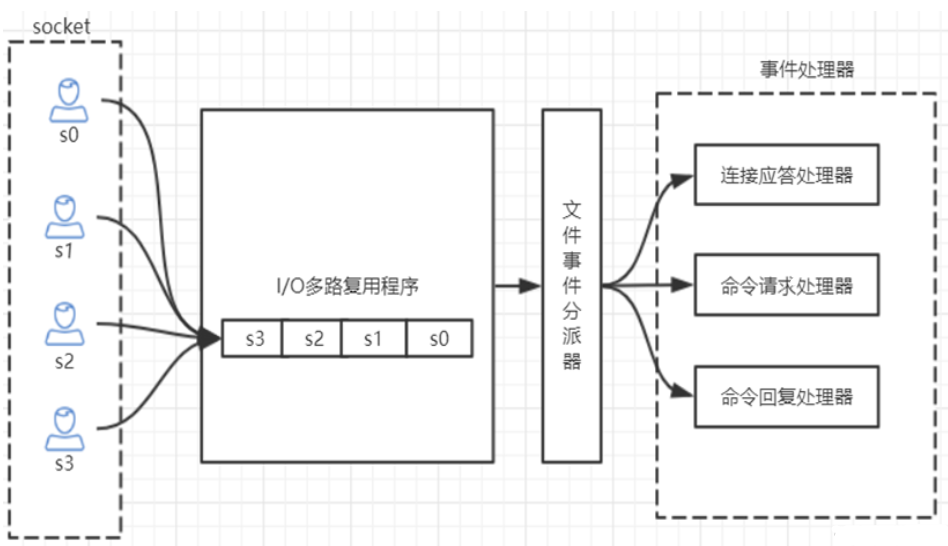
**Redis 单线程为什么还能这么快？**

因为它所有的数据都在内存中，所有的运算都是内存级别的运算，而且单线程避免了多线程的切换性能损耗问题。正因为 Redis 是单线程，所以要小心使用 Redis 指令，对于那些耗时的指令(比如keys)，一定要谨慎使用，一不小心就可能会导致 Redis 卡顿。

**Redis 单线程如何处理那么多的并发客户端连接？**

Redis的**IO多路复用**：redis利用epoll来实现IO多路复用，将连接信息和事件放到队列中，依次放到文件事件分派器，事件分派器将事件分发给事件处理器。

Nginx也是采用IO多路复用原理解决C10K问题



**持久化**

**RDB快照（snapshot）**

在默认情况下， Redis 将内存数据库快照保存在名字为 dump.rdb 的二进制文件中。

你可以对 Redis 进行设置， 让它在“ N 秒内数据集至少有 M 个改动”这一条件被满足时， 自动保存一次数据集。

比如说， 以下设置会让 Redis 在满足“ 60 秒内有至少有 1000 个键被改动”这一条件时， 自动保存一次数据集：

# save 60 1000

**AOF（append-only file）**

快照功能并不是非常耐久（durable）： 如果 Redis 因为某些原因而造成故障停机， 那么服务器将丢失最近写入、且仍未保存到快照中的那些数据。从 1.1 版本开始， Redis 增加了一种完全耐久的持久化方式： AOF 持久化，将修改的每一条指令记录进文件

你可以通过修改配置文件来打开 AOF 功能：

# appendonly yes

从现在开始， 每当 Redis 执行一个改变数据集的命令时（比如 [SET](http://redisdoc.com/string/set.html#set)）， 这个命令就会被追加到 AOF 文件的末尾。

这样的话， 当 Redis 重新启时， 程序就可以通过重新执行 AOF 文件中的命令来达到重建数据集的目的。

你可以配置 Redis 多久才将数据 fsync 到磁盘一次。

有三个选项：

每次有新命令追加到 AOF 文件时就执行一次 fsync ：非常慢，也非常安全。

每秒 fsync 一次：足够快（和使用 RDB 持久化差不多），并且在故障时只会丢失 1 秒钟的数据。

从不 fsync ：将数据交给操作系统来处理。更快，也更不安全的选择。

推荐（并且也是默认）的措施为每秒 fsync 一次， 这种 fsync 策略可以兼顾速度和安全性。

**RDB 和 AOF ，我应该用哪一个？**

如果你非常关心你的数据， 但仍然可以承受数分钟以内的数据丢失， 那么你可以只使用 RDB 持久化。

有很多用户都只使用 AOF 持久化， 但我们并不推荐这种方式： 因为定时生成 RDB 快照（snapshot）非常便于进行数据库备份， 并且 RDB 恢复数据集的速度也要比 AOF 恢复的速度要快。

**Redis 4.0 混合持久化**

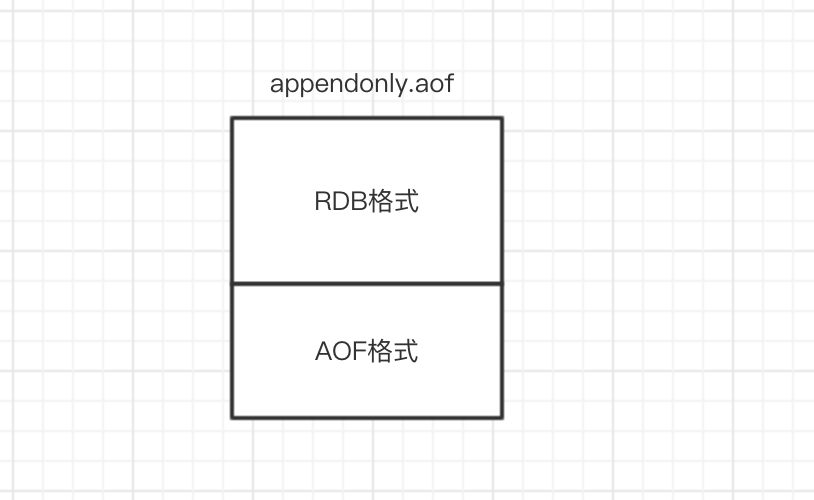
重启 Redis 时，我们很少使用 rdb 来恢复内存状态，因为会丢失大量数据。我们通常使用 AOF 日志重放，但是重放 AOF 日志性能相对 rdb 来说要慢很多，这样在 Redis 实例很大的情况下，启动需要花费很长的时间。 Redis 4.0 为了解决这个问题，带来了一个新的持久化选项——混合持久化。**AOF在重写**(aof文件里可能有太多没用指令，所以aof会定期根据内存的最新数据生成aof文件)**时将重写这一刻之前的内存rdb快照文件的内容和增量的 AOF修改内存数据的命令日志文件存在一起，都写入新的aof文件，新的文件一开始不叫appendonly.aof，等到重写完新的AOF文件才会进行改名，原子的覆盖原有的AOF文件，完成新旧两个AOF文件的替换；**

AOF根据配置规则在后台自动重写，也可以人为执行命令bgrewriteaof重写AOF。 于是在 Redis 重启的时候，可以先加载 rdb 的内容，然后再重放增量 AOF 日志就可以完全替代之前的 AOF 全量文件重放，重启效率因此大幅得到提升。

开启混合持久化：

# aof-use-rdb-preamble yes

混合持久化aof文件结构



**缓存淘汰策略**

当 Redis 内存超出物理内存限制时，内存的数据会开始和磁盘产生频繁的交换 (swap)。交换会让 Redis 的性能急剧下降，对于访问量比较频繁的 Redis 来说，这样龟速的存取效率基本上等于不可用。

在生产环境中我们是不允许 Redis 出现交换行为的，为了限制最大使用内存，Redis 提供了配置参数 maxmemory 来限制内存超出期望大小。

当实际内存超出 maxmemory 时，Redis 提供了几种可选策略 (maxmemory-policy) 来让用户自己决定该如何腾出新的空间以继续提供读写服务。

**noeviction** 不会继续服务写请求 (DEL 请求可以继续服务)，读请求可以继续进行。这样可以保证不会丢失数据，但是会让线上的业务不能持续进行。这是默认的淘汰策略。

**volatile-lru** 尝试淘汰设置了过期时间的 key，最少使用的 key 优先被淘汰。没有设置过期时间的 key 不会被淘汰，这样可以保证需要持久化的数据不会突然丢失。

**volatile-ttl** 跟上面一样，除了淘汰的策略不是 LRU，而是 key 的剩余寿命 ttl 的值，ttl 越小越优先被淘汰。

**volatile-random** 跟上面一样，不过淘汰的 key 是过期 key 集合中随机的 key。

**allkeys-lru** 区别于 volatile-lru，这个策略要淘汰的 key 对象是全体的 key 集合，而不只是过期的 key 集合。这意味着没有设置过期时间的 key 也会被淘汰。

**allkeys-random** 跟上面一样，不过淘汰的策略是随机的 key。

volatile-xxx 策略只会针对带过期时间的 key 进行淘汰，allkeys-xxx 策略会对所有的 key 进行淘汰。如果你只是拿 Redis 做缓存，那应该使用 allkeys-xxx，客户端写缓存时不必携带过期时间。如果你还想同时使用 Redis 的持久化功能，那就使用 volatile-xxx 策略，这样可以保留没有设置过期时间的 key，它们是永久的 key 不会被 LRU 算法淘汰。