史上最全 50 道 Redis 面试题

1、什么是 Redis？

Redis 本质上是一个 Key-Value 类型的内存数据库，很像 memcached，整个数据库统统加载在内存当中进行操作，定期通过异步操作把数据库数据 flush 到硬盘上进行保存。因为是纯内存操作，Redis 的性能非常出色，每秒可以处理超过 10 万次读写操作，是已知性能最快的 Key-Value DB。

Redis 的出色之处不仅仅是性能，Redis 最大的魅力是支持保存多种数据结构，此外单个 value 的最大限制是 1GB，不像 memcached 只能保存 1MB 的数据，因此 Redis 可以用来实现很多有用的功能，比方说用他的 List 来做 FIFO 双向链表，实现一个轻量级的高性 能消息队列服务，用他的 Set 可以做高性能的 tag 系统等等。另外 Redis 也可以对存入的 Key-Value 设置 expire 时间，因此也可以被当作一 个功能加强版的 memcached 来用。

Redis 的主要缺点是数据库容量受到物理内存的限制，不能用作海量数据的高性能读写，因此 Redis 适合的场景主要局限在较小数据量的高性能操作和运算上。

2、Redis 相比 memcached 有哪些优势？

(1) 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

(2) redis 的速度比 memcached 快很多

(3) redis 可以持久化其数据

3、Redis 支持哪几种数据类型？

String、List、Set、Sorted Set、hashes

4、Redis 主要消耗什么物理资源？

内存。

5、Redis 的全称是什么？

Remote Dictionary Server。

6、Redis 有哪几种数据淘汰策略？

noeviction:返回错误当内存限制达到并且客户端尝试执行会让更多内存被使用的命令（大部分的写入指令，但 DEL 和几个例外）

allkeys-lru: 尝试回收最少使用的键（LRU），使得新添加的数据有空间存放。

volatile-lru: 尝试回收最少使用的键（LRU），但仅限于在过期集合的键,使得新添加的数据有空间存放。

allkeys-random: 回收随机的键使得新添加的数据有空间存放。

volatile-random: 回收随机的键使得新添加的数据有空间存放，但仅限于在过期集合的键。

volatile-ttl: 回收在过期集合的键，并且优先回收存活时间（TTL）较短的键,使得新添加的数据有空间存放。

7、Redis 官方为什么不提供 Windows 版本？

因为目前 Linux 版本已经相当稳定，而且用户量很大，无需开发 windows 版本，反而会带来兼容性等问题。

8、一个字符串类型的值能存储最大容量是多少？

512M

9、为什么 Redis 需要把所有数据放到内存中？

Redis 为了达到最快的读写速度将数据都读到内存中，并通过异步的方式将数据写入磁盘。所以 redis 具有快速和数据持久化的特征。如果不将数据放在内存中，磁盘 I/O 速度为严重影响 redis 的性能。在内存越来越便宜的今天，redis 将会越来越受欢迎。

如果设置了最大使用的内存，则数据已有记录数达到内存限值后不能继续插入新值。

10、Redis 集群方案应该怎么做？都有哪些方案？

1.twemproxy，大概概念是，它类似于一个代理方式，使用方法和普通 redis 无任何区别，设置好它下属的多个 redis 实例后，使用时在本需要连接 redis 的地方改为连接 twemproxy，它会以一个代理的身份接收请求并使用一致性 hash 算法，将请求转接到具体 redis，将结果再返回 twemproxy。使用方式简便(相对 redis 只需修改连接端口)，对旧项目扩展的首选。 问题：twemproxy 自身单端口实例的压力，使用一致性 hash 后，对 redis 节点数量改变时候的计算值的改变，数据无法自动移动到新的节点。

2.codis，目前用的最多的集群方案，基本和 twemproxy 一致的效果，但它支持在 节点数量改变情况下，旧节点数据可恢复到新 hash 节点。

3.redis cluster3.0 自带的集群，特点在于他的分布式算法不是一致性 hash，而是 hash 槽的概念，以及自身支持节点设置从节点。具体看官方文档介绍。

4.在业务代码层实现，起几个毫无关联的 redis 实例，在代码层，对 key 进行 hash 计算，然后去对应的 redis 实例操作数据。 这种方式对 hash 层代码要求比较高，考虑部分包括，节点失效后的替代算法方案，数据震荡后的自动脚本恢复，实例的监控，等等。

11、Redis 集群方案什么情况下会导致整个集群不可用？

有 A，B，C 三个节点的集群,在没有复制模型的情况下,如果节点 B 失败了，那么整个集群就会以为缺少 5501-11000 这个范围的槽而不可用。

12、MySQL 里有 2000w 数据，redis 中只存 20w 的数据，如何保证 redis 中的数据都是热点数据？

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。

13、Redis 有哪些适合的场景？

（1）、会话缓存（Session Cache）

最常用的一种使用 Redis 的情景是会话缓存（session cache）。用 Redis 缓存会话比其他存储（如 Memcached）的优势在于：Redis 提供持久化。当维护一个不是严格要求一致性的缓存时，如果用户的购物车信息全部丢失，大部分人都会不高兴的，现在，他们还会这样吗？

幸运的是，随着 Redis 这些年的改进，很容易找到怎么恰当的使用 Redis 来缓存会话的文档。甚至广为人知的商业平台 Magento 也提供 Redis 的插件。

（2）、全页缓存（FPC）

除基本的会话 token 之外，Redis 还提供很简便的 FPC 平台。回到一致性问题，即使重启了 Redis 实例，因为有磁盘的持久化，用户也不会看到页面加载速度的下降，这是一个极大改进，类似 PHP 本地 FPC。

再次以 Magento 为例，Magento 提供一个插件来使用 Redis 作为全页缓存后端。

此外，对 WordPress 的用户来说，Pantheon 有一个非常好的插件 wp-redis，这个插件能帮助你以最快速度加载你曾浏览过的页面。

（3）、队列

Reids 在内存存储引擎领域的一大优点是提供 list 和 set 操作，这使得 Redis 能作为一个很好的消息队列平台来使用。Redis 作为队列使用的操作，就类似于本地程序语言（如 Python）对 list 的 push/pop 操作。

如果你快速的在 Google 中搜索“Redis queues”，你马上就能找到大量的开源项目，这些项目的目的就是利用 Redis 创建非常好的后端工具，以满足各种队列需求。例如，Celery 有一个后台就是使用 Redis 作为 broker，你可以从这里去查看。

（4），排行榜/计数器

Redis 在内存中对数字进行递增或递减的操作实现的非常好。集合（Set）和有序集合（Sorted Set）也使得我们在执行这些操作的时候变的非常简单，Redis 只是正好提供了这两种数据结构。所以，我们要从排序集合中获取到排名最靠前的 10 个用户 – 我们称之为“user_scores”，我们只需要像下面一样执行即可：

当然，这是假定你是根据你用户的分数做递增的排序。如果你想返回用户及用户的分数，你需要这样执行：

ZRANGE user_scores 0 10 WITHSCORES

Agora Games 就是一个很好的例子，用 Ruby 实现的，它的排行榜就是使用 Redis 来存储数据的，你可以在这里看到。

（5）、发布/订阅

最后（但肯定不是最不重要的）是 Redis 的发布/订阅功能。发布/订阅的使用场景确实非常多。我已看见人们在社交网络连接中使用，还可作为基于发布/订阅的脚本触发器，甚至用 Redis 的发布/订阅功能来建立聊天系统！（不，这是真的，你可以去核实）。

14、Redis 支持的 Java 客户端都有哪些？官方推荐用哪个？

Redisson、Jedis、lettuce 等等，官方推荐使用 Redisson。

15、Redis 和 Redisson 有什么关系？

Redisson 是一个高级的分布式协调 Redis 客服端，能帮助用户在分布式环境中轻松实现一些 Java 的对象 (Bloom filter, BitSet, Set, SetMultimap, ScoredSortedSet, SortedSet, Map, ConcurrentMap, List, ListMultimap, Queue, BlockingQueue, Deque, BlockingDeque, Semaphore, Lock,

ReadWriteLock, AtomicLong, CountDownLatch, Publish / Subscribe, HyperLogLog)。

16、Jedis 与 Redisson 对比有什么优缺点？

Jedis 是 Redis 的 Java 实现的客户端，其 API 提供了比较全面的 Redis 命令的支持；Redisson 实现了分布式和可扩展的 Java 数据结构，和 Jedis 相比，功能较为简单，不支持字符串操作，不支持排序、事务、管道、分区等 Redis 特性。Redisson 的宗旨是促进使用者对 Redis 的关注分离，从而让使用者能够将精力更集中地放在处理业务逻辑上。

17、Redis 如何设置密码及验证密码？

设置密码：config set requirepass 123456

授权密码：auth 123456

18、说说 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。

19、Redis 集群的主从复制模型是怎样的？

为了使在部分节点失败或者大部分节点无法通信的情况下集群仍然可用，所以集群使用了主从复制模型,每个节点都会有 N-1 个复制品.

20、Redis 集群会有写操作丢失吗？为什么？

Redis 并不能保证数据的强一致性，这意味这在实际中集群在特定的条件下可能会丢失写操作。

21、Redis 集群之间是如何复制的？

异步复制

22、Redis 集群最大节点个数是多少？

16384 个。

23、Redis 集群如何选择数据库？

Redis 集群目前无法做数据库选择，默认在 0 数据库。

24、怎么测试 Redis 的连通性？

ping

25、Redis 中的管道有什么用？

一次请求/响应服务器能实现处理新的请求即使旧的请求还未被响应。这样就可以将多个命令发送到服务器，而不用等待回复，最后在一个步骤中读取该答复。

这就是管道（pipelining），是一种几十年来广泛使用的技术。例如许多 POP3 协议已经实现支持这个功能，大大加快了从服务器下载新邮件的过程。

26、怎么理解 Redis 事务？

事务是一个单独的隔离操作：事务中的所有命令都会序列化、按顺序地执行。事务在执行的过程中，不会被其他客户端发送来的命令请求所打断。

事务是一个原子操作：事务中的命令要么全部被执行，要么全部都不执行。

27、Redis 事务相关的命令有哪几个？

MULTI、EXEC、DISCARD、WATCH

28、Redis key 的过期时间和永久有效分别怎么设置？

EXPIRE 和 PERSIST 命令。

29、Redis 如何做内存优化？

尽可能使用散列表（hashes），散列表（是说散列表里面存储的数少）使用的内存非常小，所以你应该尽可能的将你的数据模型抽象到一个散列表里面。比如你的 web 系统中有一个用户对象，不要为这个用户的名称，姓氏，邮箱，密码设置单独的 key,而是应该把这个用户的所有信息存储到一张散列表里面.

30、Redis 回收进程如何工作的？

1. 一个客户端运行了新的命令，添加了新的数据。
2. Redi 检查内存使用情况，如果大于 maxmemory 的限制, 则根据设定好的策略进行回收。
3. 一个新的命令被执行，等等。
4. 所以我们不断地穿越内存限制的边界，通过不断达到边界然后不断地回收回到边界以下。

如果一个命令的结果导致大量内存被使用（例如很大的集合的交集保存到一个新的键），不用多久内存限制就会被这个内存使用量超越。

31、Redis 回收使用的是什么算法？

LRU 算法

32、Redis 如何做大量数据插入？

Redis2.6 开始 redis-cli 支持一种新的被称之为 pipe mode 的新模式用于执行大量数据插入工作。

33、为什么要做 Redis 分区？

分区可以让 Redis 管理更大的内存，Redis 将可以使用所有机器的内存。如果没有分区，你最多只能使用一台机器的内存。分区使 Redis 的计算能力通过简单地增加计算机得到成倍提升,Redis 的网络带宽也会随着计算机和网卡的增加而成倍增长。

34、你知道有哪些 Redis 分区实现方案？

- 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。大多数客户端已经实现了客户端分区。
- 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 和 memcached 的一种代理实现就是 Twemproxy
- 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。

35、Redis 分区有什么缺点？

- 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）。
- 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（The partitioning granularity is the key, so it

is not possible to shard a dataset with a single huge key like a very big sorted set）.

- 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。
- 分区时动态扩容或缩容可能非常复杂。Redis 集群在运行时增加或者删除 Redis 节点，能做到最大程度对用户透明地数据再平衡，但其他一些客户端分区或者代理分区方法则不支持这种特性。然而，有一种预分片的技术也可以较好的解决这个问题。

36、Redis 持久化数据和缓存怎么做扩容？

- 如果 Redis 被当做缓存使用，使用一致性哈希实现动态扩容缩容。
- 如果 Redis 被当做一个持久化存储使用，必须使用固定的 keys-to-nodes 映射关系，节点的数量一旦确定不能变化。否则的话(即 Redis 节点需要动态变化的情况），必须使用可以在运行时进行数据再平衡的一套系统，而当前只有 Redis 集群可以做到这样。

37、分布式 Redis 是前期做还是后期规模上来了再做好？为什么？

既然 Redis 是如此的轻量（单实例只使用 1M 内存）,为防止以后的扩容，最好的办法就是一开始就启动较多实例。即便你只有一台服务器，你也可以一开始就让 Redis 以分布式的方式运行，使用分区，在同一台服务器上启动多个实例。

一开始就多设置几个 Redis 实例，例如 32 或者 64 个实例，对大多数用户来说这操作起来可能比较麻烦，但是从长久来看做这点牺牲是值得的。

这样的话，当你的数据不断增长，需要更多的 Redis 服务器时，你需要做的就是仅仅将 Redis 实例从一台服务迁移到另外一台服务器而已（而不用考虑重新分区的问题）。一旦你添加了另一台服务器，你需要将你一半的 Redis 实例从第一台机器迁移到第二台机器。

38、Twemproxy 是什么？

Twemproxy 是 Twitter 维护的（缓存）代理系统，代理 Memcached 的 ASCII 协议和 Redis 协议。它是单线程程序，使用 c 语言编写，运行起来非常快。它是采用 Apache 2.0 license 的开源软件。 Twemproxy 支持自动分区，如果其代理的其中一个 Redis 节点不可用时，会自动将该节点排除（这将改变原来的 keys-instances 的映射关系，所以你应该仅在把 Redis 当缓存时使用 Twemproxy)。 Twemproxy 本身不存在单点问题，因为你可以启动多个 Twemproxy 实例，然后让你的客户端去连接任意一个 Twemproxy 实例。

Twemproxy 是 Redis 客户端和服务器端的一个中间层，由它来处理分区功能应该不算复杂，并且应该算比较可靠的。

39、支持一致性哈希的客户端有哪些？

Redis-rb、Predis 等。

40、Redis 与其他 key-value 存储有什么不同？

1. Redis 有着更为复杂的数据结构并且提供对他们的原子性操作，这是一个不同于其他数据库的进化路径。Redis 的数据类型都是基于基本数据结构的同时对程序员透明，无需进行额外的抽象。
2. Redis 运行在内存中但是可以持久化到磁盘，所以在对不同数据集进行高速读写时需要权衡内存，应为数据量不能大于硬件内存。在内存数据库方面的另一个优点是， 相比在磁盘上相同的复杂的数据结构，在内存中操作起来非常简单，这样 Redis 可以做很多内部复杂性很强的事情。 同时，在磁盘格式方面他们是紧凑的以追加的方式产生的，因为他们并不需要进行随机访问。

41、Redis 的内存占用情况怎么样？

给你举个例子： 100 万个键值对（键是 0 到 999999 值是字符串“hello world”）在我的 32 位的 Mac 笔记本上 用了 100MB。同样的数据放到一个 key 里只需要 16MB， 这是因为键值有一个很大的开销。 在 Memcached 上执行也是类似的结果，但是相对 Redis 的开销要小一点点，因为 Redis 会记录类型信息引用计数等等。

当然，大键值对时两者的比例要好很多。

64 位的系统比 32 位的需要更多的内存开销，尤其是键值对都较小时，这是因为 64 位的系统里指针占用了 8 个字节。 但是，当然，64 位系统支持更大的内存，所以为了运行大型的 Redis 服务器或多或少的需要使用 64 位的系统。

42、都有哪些办法可以降低 Redis 的内存使用情况呢？

如果你使用的是 32 位的 Redis 实例，可以好好利用 Hash,list,sorted set,set 等集合类型数据，因为通常情况下很多小的 Key-Value 可以用更紧凑的方式存放到一起。

43、查看 Redis 使用情况及状态信息用什么命令？

info

44、Redis 的内存用完了会发生什么？

如果达到设置的上限，Redis 的写命令会返回错误信息（但是读命令还可以正常返回。）或者你可以将 Redis 当缓存来使用配置淘汰机制，当 Redis 达到内存上限时会冲刷掉旧的内容。

45、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）。

46、一个 Redis 实例最多能存放多少的 keys？List、Set、Sorted Set 他们最多能存放多少元素？

理论上 Redis 可以处理多达 232 的 keys，并且在实际中进行了测试，每个实例至少存放了 2 亿 5 千万的 keys。我们正在测试一些较大的值。

任何 list、set、和 sorted set 都可以放 232 个元素。

换句话说，Redis 的存储极限是系统中的可用内存值。

47、Redis 常见性能问题和解决方案？

(1) Master 最好不要做任何持久化工作，如 RDB 内存快照和 AOF 日志文件

(2) 如果数据比较重要，某个 Slave 开启 AOF 备份数据，策略设置为每秒同步一次

(3) 为了主从复制的速度和连接的稳定性，Master 和 Slave 最好在同一个局域网内

(4) 尽量避免在压力很大的主库上增加从库

(5) 主从复制不要用图状结构，用单向链表结构更为稳定，即：Master <- Slave1 <- Slave2 <- Slave3...

这样的结构方便解决单点故障问题，实现 Slave 对 Master 的替换。如果 Master 挂了，可以立刻启用 Slave1 做 Master，其他不变。

48、Redis 提供了哪几种持久化方式？

1. RDB 持久化方式能够在指定的时间间隔能对你的数据进行快照存储.
2. AOF 持久化方式记录每次对服务器写的操作,当服务器重启的时候会重新执行这些命令来恢复原始的数据,AOF 命令以 redis 协议追

加保存每次写的操作到文件末尾.Redis 还能对 AOF 文件进行后台重写,使得 AOF 文件的体积不至于过大.

3. 如果你只希望你的数据在服务器运行的时候存在,你也可以不使用任何持久化方式.
4. 你也可以同时开启两种持久化方式，在这种情况下，当 redis 重启的时候会优先载入 AOF 文件来恢复原始的数据,因为在通常情况下 AOF 文件保存的数据集要比 RDB 文件保存的数据集要完整.
5. 最重要的事情是了解 RDB 和 AOF 持久化方式的不同,让我们以 RDB 持久化方式开始。

49、如何选择合适的持久化方式？

一般来说， 如果想达到足以媲美 PostgreSQL 的数据安全性， 你应该同时使用两种持久化功能。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，除此之外， 使用 RDB 还可以避免之前提到的 AOF 程序的 bug。

50、修改配置不重启 Redis 会实时生效吗？

针对运行实例，有许多配置选项可以通过 CONFIG SET 命令进行修改，而无需执行任何形式的重启。 从 Redis 2.2 开始，可以从 AOF 切换到 RDB 的快照持久性或其他方式而不需要重启 Redis。检索 ‘CONFIG GET *’ 命令获取更多信息。

但偶尔重新启动是必须的，如为升级 Redis 程序到新的版本，或者当你需要修改某些目前 CONFIG 命令还不支持的配置参数的时候。

Redis 21 问，你接得住不？

1.什么是 redis?

Redis 是一个基于内存的高性能 key-value 数据库。

2.Reids 的特点

Redis 本质上是一个 Key-Value 类型的内存数据库，很像 memcached，整个数据库统统加载在内存当中进行操作，定期通过异步操作把数据库数据 flush 到硬

盘上进行保存。因为是纯内存操作，Redis 的性能非常出色，每秒可以处理超过 10 万次读写操作，是已知性能最快的 Key-Value DB。

Redis 的出色之处不仅仅是性能，Redis 最大的魅力是支持保存多种数据结构，此外单个 value 的最大限制是 1GB，不像 memcached 只能保存 1MB 的数据，因此 Redis 可以用来实现很多有用的功能，比方说用他的 List 来做 FIFO 双向链表，实现一个轻量级的高性 能消息队列服务，用他的 Set 可以做高性能的 tag 系统等等。另外 Redis 也可以对存入的 Key-Value 设置 expire 时间，因此也可以被当作一 个功能加强版的 memcached 来用。

Redis 的主要缺点是数据库容量受到物理内存的限制，不能用作海量数据的高性能读写，因此 Redis 适合的场景主要局限在较小数据量的高性能操作和运算上。

3.使用 redis 有哪些好处？

1.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)

2.支持丰富数据类型，支持 string，list，set，sorted set，hash

1）String

常用命令：set/get/decr/incr/mget 等；

应用场景：String 是最常用的一种数据类型，普通的 key/value 存储都可以归为此类；

实现方式：String 在 redis 内部存储默认就是一个字符串，被 redisObject 所引用，当遇到 incr、decr 等操作时会转成数值型进行计算，此时 redisObject 的 encoding 字段为 int。

2）Hash

常用命令：hget/hset/hgetall 等

应用场景：我们要存储一个用户信息对象数据，其中包括用户 ID、用户姓名、年龄和生日，通过用户 ID 我们希望获取该用户的姓名或者年龄或者生日；

实现方式：Redis 的 Hash 实际是内部存储的 Value 为一个 HashMap，并提供了直接存取这个 Map 成员的接口。Key 是用户 ID, value 是一个 Map。这个 Map 的 key 是成员的属性名，value 是属性值。这样对数据的修改和存取都可以直接通过其内部 Map 的 Key(Redis 里称内部 Map 的 key 为 field), 也就是通过 key(用户 ID) + field(属性标签) 就可以操作对应属性数据。

当前 HashMap 的实现有两种方式：当 HashMap 的成员比较少时 Redis 为了节省内存会采用类似一维数组的方式来紧凑存储，而不会采用真正的 HashMap 结构，这时对应的 value 的 redisObject 的 encoding 为 zipmap，当成员数量增大时会自动转成真正的 HashMap,此时 encoding 为 ht。

3）List

常用命令：lpush/rpush/lpop/rpop/lrange 等；

应用场景：Redis list 的应用场景非常多，也是 Redis 最重要的数据结构之一，比如 twitter 的关注列表，粉丝列表等都可以用 Redis 的 list 结构来实现；

实现方式：Redis list 的实现为一个双向链表，即可以支持反向查找和遍历，更方便操作，不过带来了部分额外的内存开销，Redis 内部的很多实现，包括发送缓冲队列等也都是用的这个数据结构。

4）Set

常用命令：sadd/spop/smembers/sunion 等；

应用场景：Redis set 对外提供的功能与 list 类似是一个列表的功能，特殊之处在于 set 是可以自动排重的，当你需要存储一个列表数据，又不希望出现重复数据时，set 是一个很好的选择，并且 set 提供了判断某个成员是否在一个 set 集合内的重要接口，这个也是 list 所不能提供的；

实现方式：set 的内部实现是一个 value 永远为 null 的 HashMap，实际就是通过计算 hash 的方式来快速排重的，这也是 set 能提供判断一个成员是否在集合内的原因。

5）Sorted Set

常用命令：zadd/zrange/zrem/zcard 等；

应用场景：Redis sorted set 的使用场景与 set 类似，区别是 set 不是自动有序的，而 sorted set 可以通过用户额外提供一个优先级(score)的参数来为成员排序，并且是插入有序的，即自动排序。当你需要一个有序的并且不重复的集合列表，那么可以选择 sorted set 数据结构，比如 twitter 的 public timeline 可以以发表时间作为 score 来存储，这样获取时就是自动按时间排好序的。

实现方式：Redis sorted set 的内部使用 HashMap 和跳跃表(SkipList)来保证数据的存储和有序，HashMap 里放的是成员到 score 的映射，而跳跃表里存放的是所有的成员，排序依据是 HashMap 里存的 score,使用跳跃表的结构可以获得比较高的查找效率，并且在实现上比较简单。

3. 支持事务，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

4. 丰富的特性：可用于缓存，消息，按 key 设置过期时间，过期后将会自动删除

4.redis 相比 memcached 有哪些优势？

- 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 (3) redis 可以持久化其数据

5.Memcache 与 Redis 的区别都有哪些？

- 存储方式 Memecache 把数据全部存在内存之中，断电后会挂掉，数据不能超过内存大小。Redis 有部份存在硬盘上，这样能保证数据的持久性。
- 数据支持类型 Memcache 对数据类型支持相对简单。Redis 有复杂的数据类型。
- 使用底层模型不同 它们之间底层实现方式 以及与客户端之间通信的应用协议不一样。Redis 直接自己构建了 VM 机制 ，因为一般的系统调用系统函数的话，会浪费一定的时间去移动和请求。

6.redis 适用于的场景?

Redis 最适合所有数据 in-momory 的场景，如：

1. 会话缓存（Session Cache）

最常用的一种使用 Redis 的情景是会话缓存（session cache）。用 Redis 缓存会话比其他存储（如 Memcached）的优势在于：Redis 提供持久化。

2. 全页缓存（FPC）

除基本的会话 token 之外，Redis 还提供很简便的 FPC 平台。回到一致性问题，即使重启了 Redis 实例，因为有磁盘的持久化，用户也不会看到页面加载速度的下降，这是一个极大改进，类似 PHP 本地 FPC。

3. 队列

Reids 在内存存储引擎领域的一大优点是提供 list 和 set 操作，这使得 Redis 能作为一个很好的消息队列平台来使用。Redis 作为队列使用的操作，就类似于本地程序语言（如 Python）对 list 的 push/pop 操作。

如果你快速的在 Google 中搜索“Redis queues”，你马上就能找到大量的开源项目，这些项目的目的就是利用 Redis 创建非常好的后端工具，以满足各种队列需求。例如，Celery 有一个后台就是使用 Redis 作为 broker，你可以从这里去查看。

4.排行榜/计数器

Redis 在内存中对数字进行递增或递减的操作实现的非常好。集合（Set）和有序集合（Sorted Set）也使得我们在执行这些操作的时候变的非常简单，Redis 只是正好提供了这两种数据结构。所以，我们要从排序集合中获取到排名最靠前的 10 个用户 – 我们称之为“user_scores”，我们只需要像下面一样执行即可：

当然，这是假定你是根据你用户的分数做递增的排序。如果你想返回用户及用户的分数，你需要这样执行：

```
ZRANGE user_scores 0 10 WITHSCORES
```

Agora Games 就是一个很好的例子，用 Ruby 实现的，它的排行榜就是使用 Redis 来存储数据的，你可以在这里看到。

5.发布/订阅

最后（但肯定不是最不重要的）是 Redis 的发布/订阅功能。发布/订阅的使用场景确实非常多。推荐阅读：Redis 的 8 大应用场景。

7、redis 的缓存失效策略和主键失效机制

作为缓存系统都要定期清理无效数据，就需要一个主键失效和淘汰策略. 在 Redis 当中，有生存期的 key 被称为 volatile。在创建缓存时，要为给定的 key 设置生存期，当 key 过期的时候（生存期为 0），它可能会被删除。 **1、影响生存时间的一些操作**

生存时间可以通过使用 DEL 命令来删除整个 key 来移除，或者被 SET 和 GETSET 命令覆盖原来的数据，也就是说，修改 key 对应的 value 和使用另外相同的 key 和 value 来覆盖以后，当前数据的生存时间不同。

比如说，对一个 key 执行 INCR 命令，对一个列表进行 LPUSH 命令，或者对一个哈希表执行 HSET 命令，这类操作都不会修改 key 本身的生存时间。另一方

面，如果使用 RENAME 对一个 key 进行改名，那么改名后的 key 的生存时间和改名前一样。

RENAME 命令的另一种可能是，尝试将一个带生存时间的 key 改名成另一个带生存时间的 another_key ，这时旧的 another_key (以及它的生存时间)会被删除，然后旧的 key 会改名为 another_key ，因此，新的 another_key 的生存时间也和原本的 key 一样。使用 PERSIST 命令可以在不删除 key 的情况下，移除 key 的生存时间，让 key 重新成为一个 persistent key 。

2、如何更新生存时间

可以对一个已经带有生存时间的 key 执行 EXPIRE 命令，新指定的生存时间会取代旧的生存时间。过期时间的精度已经被控制在 1ms 之内，主键失效的时间复杂度是 0（1），EXPIRE 和 TTL 命令搭配使用，TTL 可以查看 key 的当前生存时间。设置成功返回 1；当 key 不存在或者不能为 key 设置生存时间时，返回 0 。

最大缓存配置，在 redis 中，允许用户设置最大使用内存大小

`server.maxmemory`默认为 0，没有指定最大缓存，如果有新的数据添加，超过最大内存，则会使 redis 崩溃，所以一定要设置。redis 内存数据集大小上升到一定大小的时候，就会实行数据淘汰策略。 redis 提供 6 种数据淘汰策略：

- **volatile-lru：**从已设置过期时间的数据集（server.db[i].expires）中挑选最近最少使用的数据淘汰
- **volatile-ttl：**从已设置过期时间的数据集（server.db[i].expires）中挑选将要过期的数据淘汰
- **volatile-random：**从已设置过期时间的数据集（server.db[i].expires）中任意选择数据淘汰
- **allkeys-lru：**从数据集（server.db[i].dict）中挑选最近最少使用的数据淘汰
- **allkeys-random：**从数据集（server.db[i].dict）中任意选择数据淘汰
- **no-enviction（驱逐）：**禁止驱逐数据

注意这里的 6 种机制，volatile 和 allkeys 规定了是对已设置过期时间的数据集淘汰数据还是从全部数据集淘汰数据，后面的 lru、ttl 以及 random 是三种不同的淘汰策略，再加上一种 no-enviction 永不回收的策略。 使用策略规则：

- 如果数据呈现幂律分布，也就是一部分数据访问频率高，一部分数据访问频率低，则使用 allkeys-lru
- 如果数据呈现平等分布，也就是所有的数据访问频率都相同，则使用 allkeys-random

三种数据淘汰策略：

ttl 和 random 比较容易理解，实现也会比较简单。主要是 Lru 最近最少使用淘汰策略，设计上会对 key 按失效时间排序，然后取最先失效的 key 进行淘汰

8.为什么 redis 需要把所有数据放到内存中?

Redis 为了达到最快的读写速度将数据都读到内存中，并通过异步的方式将数据写入磁盘。所以 redis 具有快速和数据持久化的特征。如果不将数据放在内存中，磁盘 I/O 速度为严重影响 redis 的性能。在内存越来越便宜的今天，redis 将会越来越受欢迎。

如果设置了最大使用的内存，则数据已有记录数达到内存限值后不能继续插入新值。

9.Redis 是单进程单线程的

redis 利用队列技术将并发访问变为串行访问，消除了传统数据库串行控制的开销

10.redis 的并发竞争问题如何解决?

Redis 为单进程单线程模式，采用队列模式将并发访问变为串行访问。Redis 本身没有锁的概念，Redis 对于多个客户端连接并不存在竞争，但是在 Jedis 客户端对 Redis 进行并发访问时会发生连接超时、数据转换错误、阻塞、客户端关闭连接等问题，这些问题均是由于客户端连接混乱造成。对此有 2 种解决方法：

1. 客户端角度，为保证每个客户端间正常有序与 Redis 进行通信，对连接进行池化，同时对客户端读写 Redis 操作采用内部锁 synchronized。
2. 服务器角度，利用 setnx 实现锁。

注：对于第一种，需要应用程序自己处理资源的同步，可以使用的方法比较通俗，可以使用 synchronized 也可以使用 lock；第二种需要用到 Redis 的 setnx 命令，但是需要注意一些问题。

11、redis 常见性能问题和解决方案：

1.Master 写内存快照，save 命令调度 rdbSave 函数，会阻塞主线程的工作，当快照比较大时对性能影响是非常大的，会间断性暂停服务，所以 Master 最好不要写内存快照。

2.Master AOF 持久化，如果不重写 AOF 文件，这个持久化方式对性能的影响是最小的，但是 AOF 文件会不断增大，AOF 文件过大会影响 Master 重启的恢复速度。Master 最好不要做任何持久化工作，包括内存快照和 AOF 日志文件，特别是不要启用内存快照做持久化,如果数据比较关键，某个 Slave 开启 AOF 备份数据，策略为每秒同步一次。

3.Master 调用 BGREWRITEAOF 重写 AOF 文件，AOF 在重写的时候会占大量的 CPU 和内存资源，导致服务 load 过高，出现短暂服务暂停现象。

4.Redis 主从复制的性能问题，为了主从复制的速度和连接的稳定性，Slave 和 Master 最好在同一个局域网内。

12.redis 事物的了解 CAS(check-and-set 操作实现乐观锁)?

和众多其它数据库一样，Redis 作为 NoSQL 数据库也同样提供了事务机制。在 Redis 中，MULTI/EXEC/DISCARD/WATCH 这四个命令是我们实现事务的基石。相信对有关系型数据库开发经验的开发者而言这一概念并不陌生，即便如此，我们还是会简要的列出 Redis 中事务的实现特征：

1). 在事务中的所有命令都将会被串行化的顺序执行，事务执行期间，Redis 不会再为其它客户端的请求提供任何服务，从而保证了事物中的所有命令被原子的执行。

2). 和关系型数据库中的事务相比，在 Redis 事务中如果有某一条命令执行失败，其后的命令仍然会被继续执行。

3). 我们可以通过 MULTI 命令开启一个事务，有关系型数据库开发经验的人可以将其理解为"BEGIN TRANSACTION"语句。在该语句之后执行的命令都将被视为事务之内的操作，最后我们可以通过执行 EXEC/DISCARD 命令来提交/回滚该事务内的所有操作。这两个 Redis 命令可被视为等同于关系型数据库中的 COMMIT/ROLLBACK 语句。

4). 在事务开启之前，如果客户端与服务器之间出现通讯故障并导致网络断开，其后所有待执行的语句都将不会被服务器执行。然而如果网络中断事件是发生在客户端执行 EXEC 命令之后，那么该事务中的所有命令都会被服务器执行。

5). 当使用 Append-Only 模式时，Redis 会通过调用系统函数 write 将该事务内的所有写操作在本次调用中全部写入磁盘。然而如果在写入的过程中出现系统崩溃，如电源故障导致的宕机，那么此时也许只有部分数据被写入到磁盘，而另外一部分数据却已经丢失。Redis 服务器会在重新启动时执行一系列必要的一致性检测，一旦发现类似问题，就会立即退出并给出相应的错误提示。此

时，我们就要充分利用 Redis 工具包中提供的 redis-check-aof 工具，该工具可以帮助我们定位到数据不一致的错误，并将已经写入的部分数据进行回滚。修复之后我们就可以再次重新启动 Redis 服务器了。

13.WATCH 命令和基于 CAS 的乐观锁?

在 Redis 的事务中，WATCH 命令可用于提供 CAS(check-and-set)功能。假设我们通过 WATCH 命令在事务执行之前监控了多个 Keys，倘若在 WATCH 之后有任何 Key 的值发生了变化，EXEC 命令执行的事务都将被放弃，同时返回 Null multi-bulk 应答以通知调用者事务执行失败。例如，我们再次假设 Redis 中并未提供 incr 命令来完成键值的原子性递增，如果要实现该功能，我们只能自行编写相应的代码。

其伪码如下：

```
val = GET mykey

val = val + 1

SET mykey $val
```

以上代码只有在单连接的情况下才可以保证执行结果是正确的，因为如果在同一时刻有多个客户端在同时执行该段代码，那么就会出现多线程程序中经常出现的一种错误场景--竞态争用(race condition)。

比如，客户端 A 和 B 都在同一时刻读取了 mykey 的原有值，假设该值为 10，此后两个客户端又均将该值加一后 set 回 Redis 服务器，这样就会导致 mykey 的结果为 11，而不是我们认为的 12。为了解决类似的问题，我们需要借助 WATCH 命令的帮助，见如下代码：

```
WATCH mykey

val = GET mykey

val = val + 1

MULTI

SET mykey $val

EXEC
```

和此前代码不同的是，新代码在获取 mykey 的值之前先通过 WATCH 命令监控了该键，此后又将 set 命令包围在事务中，这样就可以有效的保证每个连接在执

行 EXEC 之前，如果当前连接获取的 mykey 的值被其它连接的客户端修改，那么当前连接的 EXEC 命令将执行失败。这样调用者在判断返回值后就可以获悉 val 是否被重新设置成功。

14.使用过 Redis 分布式锁么，它是什么回事？

先拿 setnx 来争抢锁，抢到之后，再用 expire 给锁加一个过期时间防止锁忘记了释放。

这时候对方会告诉你说你回答得不错，然后接着问如果在 setnx 之后执行 expire 之前进程意外 crash 或者要重启维护了，那会怎么样？

这时候你要给予惊讶的反馈：唉，是喔，这个锁就永远得不到释放了。紧接着你需要抓一抓自己得脑袋，故作思考片刻，好像接下来的结果是你主动思考出来的，然后回答：我记得 set 指令有非常复杂的参数，这个应该是可以同时把 setnx 和 expire 合成一条指令来用的！对方这时会显露笑容，心里开始默念：摁，这小子还不错。

15.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。

对方接着追问：如果这个 redis 正在给线上的业务提供服务，那使用 keys 指令会有什么问题？

这个时候你要回答 redis 关键的一个特性：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。

16.使用过 Redis 做异步队列么，你是怎么用的？

一般使用 list 结构作为队列，rpush 生产消息，lpop 消费消息。当 lpop 没有消息的时候，要适当 sleep 一会再重试。

如果对方追问可不可以不用 sleep 呢？ list 还有个指令叫 blpop，在没有消息的时候，它会阻塞住直到消息到来。

如果对方追问能不能生产一次消费多次呢？使用 pub/sub 主题订阅者模式，可以实现 1:N 的消息队列。

如果对方追问 pub/sub 有什么缺点？在消费者下线的情况下，生产的消息会丢失，得使用专业的消息队列如 rabbitmq 等。

如果对方追问 redis 如何实现延时队列？我估计现在你很想把面试官一棒打死如果你手上有一根棒球棍的话，怎么问的这么详细。但是你很克制，然后神态自若的回答道：使用 sortedset，拿时间戳作为 score，消息内容作为 key 调用 zadd 来生产消息，消费者用 zrangebyscore 指令获取 N 秒之前的数据轮询进行处理。

到这里，面试官暗地里已经对你竖起了大拇指。但是他不知道的是此刻你却竖起了中指，在椅子背后。

17.如果有大量的 key 需要设置同一时间过期，一般需要注意什么？

如果大量的 key 过期时间设置的过于集中，到过期的那个时间点，redis 可能会出现短暂的卡顿现象。一般需要在时间上加一个随机值，使得过期时间分散一些。

18.Redis 如何做持久化的？

bgsave 做镜像全量持久化，aof 做增量持久化。因为 bgsave 会耗费较长时间，不够实时，在停机的时候会导致大量丢失数据，所以需要 aof 来配合使用。在 redis 实例重启时，会使用 bgsave 持久化文件重新构建内存，再使用 aof 重放近期的操作指令来实现完整恢复重启之前的状态。

对方追问那如果突然机器掉电会怎样？取决于 aof 日志 sync 属性的配置，如果不要求性能，在每条写指令时都 sync 一下磁盘，就不会丢失数据。但是在高性能的要求下每次都 sync 是不现实的，一般都使用定时 sync，比如 1s1 次，这个时候最多就会丢失 1s 的数据。

对方追问 bgsave 的原理是什么？你给出两个词汇就可以了，fork 和 cow。fork 是指 redis 通过创建子进程来进行 bgsave 操作，cow 指的是 copy on write，子进程创建后，父子进程共享数据段，父进程继续提供读写服务，写脏的页面数据会逐渐和子进程分离开来。

19.Pipeline 有什么好处，为什么要用 pipeline？

可以将多次 IO 往返的时间缩减为一次，前提是 pipeline 执行的指令之间没有因果相关性。使用 redis-benchmark 进行压测的时候可以发现影响 redis 的 QPS 峰值的一个重要因素是 pipeline 批次指令的数目。

20.Redis 的同步机制了解么？

Redis 可以使用主从同步，从从同步。第一次同步时，主节点做一次 bgsave，并同时将后续修改操作记录到内存 buffer，待完成后将 rdb 文件全量同步到复制节点，复制节点接受完成后将 rdb 镜像加载到内存。加载完成后，再通知主节点将期间修改的操作记录同步到复制节点进行重放就完成了同步过程。

21.是否使用过 Redis 集群，集群的原理是什么？

Redis Sentinal 着眼于高可用，在 master 宕机时会自动将 slave 提升为 master，继续提供服务。

Redis Cluster 着眼于扩展性，在单个 redis 内存不足时，使用 Cluster 进行分片存储。

为什么 Redis 单线程却能支撑高并发？

最近在看 UNIX 网络编程并研究了一下 Redis 的实现，感觉 Redis 的源代码十分适合阅读和分析，其中 I/O 多路复用（mutiplexing）部分的实现非常干净和优雅，在这里想对这部分的内容进行简单的整理。

几种 I/O 模型

为什么 Redis 中要使用 I/O 多路复用这种技术呢？

首先，Redis 是跑在单线程中的，所有的操作都是按照顺序线性执行的，但是由于读写操作等待用户输入或输出都是阻塞的，所以 I/O 操作在一般情况下往往不能直接返回，这会导致某一文件的 I/O 阻塞导致整个进程无法对其它客户提供服务，而 **I/O 多路复用** 就是为了解决这个问题而出现的。

Blocking I/O

先来看一下传统的阻塞 I/O 模型到底是如何工作的：当使用 `read` 或者 `write` 对某一个 **文件描述符（File Descriptor 以下简称 FD）** 进行读写时，如果当前 FD 不可读或不可写，整个 Redis 服务就不会对其它的操作作出响应，导致整个服务不可用。

这也就是传统意义上的，也就是我们在编程中使用最多的阻塞模型：

阻塞模型虽然开发中非常常见也非常易于理解，但是由于它会影响其他 FD 对应的服务，所以在需要处理多个客户端任务的时候，往往都不会使用阻塞模型。

I/O 多路复用

虽然还有很多其它的 I/O 模型，但是在这里都不会具体介绍。

阻塞式的 I/O 模型并不能满足这里的需求，我们需要一种效率更高的 I/O 模型来支撑 Redis 的多个客户（redis-cli），这里涉及的就是 I/O 多路复用模型了：

在 I/O 多路复用模型中，最重要的函数调用就是 `select`，该方法的能够同时监控多个文件描述符的可读可写情况，当其中的某些文件描述符可读或者可写时，`select` 方法就会返回可读以及可写的文件描述符个数。

关于 `select` 的具体使用方法，在网络上资料很多，这里就不过多展开介绍了；

与此同时也有其它的 I/O 多路复用函数 `epoll/kqueue/evport`，它们相比 `select` 性能更优秀，同时也能支撑更多的服务。

Reactor 设计模式

Redis 服务采用 Reactor 的方式来实现文件事件处理器（每一个网络连接其实都对应一个文件描述符）

文件事件处理器使用 I/O 多路复用模块同时监听多个 FD，当 `accept`、`read`、`write` 和 `close` 文件事件产生时，文件事件处理器就会回调 FD 绑定的事件处理器。

虽然整个文件事件处理器是在单线程上运行的，但是通过 I/O 多路复用模块的引入，实现了同时对多个 FD 读写的监控，提高了网络通信模型的性能，同时也可以保证整个 Redis 服务实现的简单。

I/O 多路复用模块

I/O 多路复用模块封装了底层的 `select`、`epoll`、`avport` 以及 `kqueue` 这些 I/O 多路复用函数，为上层提供了相同的接口。

在这里我们简单介绍 Redis 是如何包装 `select` 和 `epoll` 的，简要了解该模块的功能，整个 I/O 多路复用模块抹平了不同平台上 I/O 多路复用函数的差异性，提供了相同的接口：

- `static int aeApiCreate(aeEventLoop *eventLoop)`
- `static int aeApiResize(aeEventLoop *eventLoop, int setsize)`
- `static void aeApiFree(aeEventLoop *eventLoop)`
- `static int aeApiAddEvent(aeEventLoop *eventLoop, int fd, int mask)`
- `static void aeApiDelEvent(aeEventLoop *eventLoop, int fd, int mask)`
- `static int aeApiPoll(aeEventLoop *eventLoop, struct timeval *tvp)`

同时，因为各个函数所需要的参数不同，我们在每一个子模块内部通过一个 `aeApiState` 来存储需要的上下文信息：

```
// select

typedef struct aeApiState {

    fd_set rfds, wfds;

    fd_set _rfds, _wfds;
```

```
} aeApiState;



// epoll

typedef struct aeApiState {

    int epfd;

    struct epoll_event *events;

} aeApiState;
```

这些上下文信息会存储在 `eventLoop` 的 `void *state` 中，不会暴露到上层，只在当前子模块中使用。

封装 select 函数

`select` 可以监控 FD 的可读、可写以及出现错误的情况。

在介绍 I/O 多路复用模块如何对 `select` 函数封装之前，先来看一下 `select` 函数使用的大致流程：

```
int fd = /* file descriptor */



fd_set rfds;

FD_ZERO(&rfds);

FD_SET(fd, &rfds)



for ( ; ; ) {

    select(fd+1, &rfds, NULL, NULL, NULL);

    if (FD_ISSET(fd, &rfds)) {

        /* file descriptor `fd` becomes readable */
```

```
    }

}
```

1. 初始化一个可读的 `fd_set` 集合，保存需要监控可读性的 FD；
2. 使用 `FD_SET` 将 `fd` 加入 `rfds`；
3. 调用 `select` 方法监控 `rfds` 中的 FD 是否可读；
4. 当 `select` 返回时，检查 FD 的状态并完成对应的操作。

而在 Redis 的 `ae_select` 文件中代码的组织顺序也是差不多的，首先在 `aeApiCreate` 函数中初始化 `rfds` 和 `wfds`：

```
static int aeApiCreate(aeEventLoop *eventLoop) {

    aeApiState *state = zmalloc(sizeof(aeApiState));

    if (!state) return -1;

    FD_ZERO(&state->rfds);

    FD_ZERO(&state->wfds);

    eventLoop->apidata = state;

    return 0;

}
```

而 `aeApiAddEvent` 和 `aeApiDelEvent` 会通过 `FD_SET` 和 `FD_CLR` 修改 `fd_set` 中对应 FD 的标志位：

```
static int aeApiAddEvent(aeEventLoop *eventLoop, int fd, int ma
sk) {

    aeApiState *state = eventLoop->apidata;

    if (mask & AE_READABLE) FD_SET(fd,&state->rfds);

    if (mask & AE_WRITABLE) FD_SET(fd,&state->wfds);

    return 0;

}
```

整个 `ae_select` 子模块中最重要的函数就是 `aeApiPoll`，它是实际调用 `select` 函数的部分，其作用就是在 I/O 多路复用函数返回时，将对应的 FD 加入 `aeEventLoop` 的 `fired` 数组中，并返回事件的个数：

```
static int aeApiPoll(aeEventLoop *eventLoop, struct timeval *tv
p) {

    aeApiState *state = eventLoop->apidata;

    int retval, j, numevents = 0;



    memcpy(&state->_rfds,&state->rfds,sizeof(fd_set));

    memcpy(&state->_wfds,&state->wfds,sizeof(fd_set));



    retval = select(eventLoop->maxfd+1,

                &state->_rfds,&state->_wfds,NULL,tvp);

    if (retval > 0) {

        for (j = 0; j <= eventLoop->maxfd; j++) {

            int mask = 0;

            aeFileEvent *fe = &eventLoop->events[j];



            if (fe->mask == AE_NONE) continue;

            if (fe->mask & AE_READABLE && FD_ISSET(j,&state->_rf
ds))

                mask |= AE_READABLE;

            if (fe->mask & AE_WRITABLE && FD_ISSET(j,&state->_wf
ds))

                mask |= AE_WRITABLE;
```

```
            eventLoop->fired[numevents].fd = j;

            eventLoop->fired[numevents].mask = mask;

            numevents++;

        }

    }

    return numevents;

}
```

封装 epoll 函数

Redis 对 `epoll` 的封装其实也是类似的，使用 `epoll_create` 创建 `epoll` 中使用的 `epfd`：

```
static int aeApiCreate(aeEventLoop *eventLoop) {

    aeApiState *state = zmalloc(sizeof(aeApiState));



    if (!state) return -1;

    state->events = zmalloc(sizeof(struct epoll_event)*eventLoo
p->setsize);

    if (!state->events) {

        zfree(state);

        return -1;

    }

    state->epfd = epoll_create(1024); /* 1024 is just a hint for
 the kernel */

    if (state->epfd == -1) {

        zfree(state->events);
```

```
        zfree(state);

        return -1;

    }

    eventLoop->apidata = state;

    return 0;

}
```

在 `aeApiAddEvent` 中使用 `epoll_ctl` 向 `epfd` 中添加需要监控的 FD 以及监听的事件：

```
static int aeApiAddEvent(aeEventLoop *eventLoop, int fd, int ma
sk) {

    aeApiState *state = eventLoop->apidata;

    struct epoll_event ee = {0}; /* avoid valgrind warning */

    /* If the fd was already monitored for some event, we need a
 MOD

     * operation. Otherwise we need an ADD operation. */

    int op = eventLoop->events[fd].mask == AE_NONE ?

            EPOLL_CTL_ADD : EPOLL_CTL_MOD;



    ee.events = 0;

    mask |= eventLoop->events[fd].mask; /* Merge old events */

    if (mask & AE_READABLE) ee.events |= EPOLLIN;

    if (mask & AE_WRITABLE) ee.events |= EPOLLOUT;

    ee.data.fd = fd;

    if (epoll_ctl(state->epfd,op,fd,&ee) == -1) return -1;
```

```
    return 0;

}
```

由于 `epoll` 相比 `select` 机制略有不同，在 `epoll_wait` 函数返回时并不需要遍历所有的 FD 查看读写情况；在 `epoll_wait` 函数返回时会提供一个 `epoll_event` 数组：

```
typedef union epoll_data {

    void    *ptr;

    int      fd; /* 文件描述符 */

    uint32_t u32;

    uint64_t u64;

} epoll_data_t;



struct epoll_event {

    uint32_t     events; /* Epoll 事件 */

    epoll_data_t data;

};
```

其中保存了发生的 `epoll` 事件（`EPOLLIN`、`EPOLLOUT`、`EPOLLERR` 和 `EPOLLHUP`）以及发生该事件的 FD。

`aeApiPoll` 函数只需要将 `epoll_event` 数组中存储的信息加入 `eventLoop` 的 `fired` 数组中，将信息传递给上层模块：

```
static int aeApiPoll(aeEventLoop *eventLoop, struct timeval *tv
p) {

    aeApiState *state = eventLoop->apidata;

    int retval, numevents = 0;
```

```
    retval = epoll_wait(state->epfd,state->events,eventLoop->se
tsize,

            tvp ? (tvp->tv_sec*1000 + tvp->tv_usec/1000) : -1);

    if (retval > 0) {

        int j;



        numevents = retval;

        for (j = 0; j < numevents; j++) {

            int mask = 0;

            struct epoll_event *e = state->events+j;



            if (e->events & EPOLLIN) mask |= AE_READABLE;

            if (e->events & EPOLLOUT) mask |= AE_WRITABLE;

            if (e->events & EPOLLERR) mask |= AE_WRITABLE;

            if (e->events & EPOLLHUP) mask |= AE_WRITABLE;

            eventLoop->fired[j].fd = e->data.fd;

            eventLoop->fired[j].mask = mask;

        }

    }

    return numevents;

}
```

子模块的选择

因为 Redis 需要在多个平台上运行，同时为了最大化执行的效率与性能，所以会根据编译平台的不同选择不同的 I/O 多路复用函数作为子模块，提供给上层统一的接口；在 Redis 中，我们通过宏定义的使用，合理的选择不同的子模块：

```
#ifdef HAVE_EVPORT

#include "ae_evport.c"

#else

    #ifdef HAVE_EPOLL

    #include "ae_epoll.c"

    #else

        #ifdef HAVE_KQUEUE

        #include "ae_kqueue.c"

        #else

        #include "ae_select.c"

        #endif

    #endif

#endif
```

因为 `select` 函数是作为 POSIX 标准中的系统调用，在不同版本的操作系统上都会实现，所以将其作为保底方案：

Redis 会优先选择时间复杂度为 $O(1)$ 的 I/O 多路复用函数作为底层实现，包括 Solaries 10 中的 `evport`、Linux 中的 `epoll` 和 macOS/FreeBSD 中的 `kqueue`，上述的这些函数都使用了内核内部的结构，并且能够服务几十万的文件描述符。

但是如果当前编译环境没有上述函数，就会选择 `select` 作为备选方案，由于其在使用时会扫描全部监听的描述符，所以其时间复杂度较差 $O(n)$，并且只能同时服务 1024 个文件描述符，所以一般并不会以 `select` 作为第一方案使用。

总结

Redis 对于 I/O 多路复用模块的设计非常简洁，通过宏保证了 I/O 多路复用模块在不同平台上都有着优异的性能，将不同的 I/O 多路复用函数封装成相同的 API 提供给上层使用。

整个模块使 Redis 能以单进程运行的同时服务成千上万个文件描述符，避免了由于多进程应用的引入导致代码实现复杂度的提升，减少了出错的可能性。