技术选型（你为什么要用redis，不用memcached或者MongDB）？

- **数据类型方面**

redis支持更丰富的数据类型（支持更复杂的应用场景）：Redis不仅仅支持简单的k/v类型的数据，同时还提供list，set，zset，hash等数据结构的存储。

memcache仅支持简单的String类型

- **持久化方面**

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，而Memcache把数据全部存在内存中

- **集群模式**

memcache没有原生的集群模式，需要依靠客户端来实现往集群中分片写入数据；

但是redis目前支持原生的集群模式

- **memchache是多线程，采用的是非阻塞IO网络模型；redis使用的是单线程的多路IO复用模型。**

网络编程IO模型

为了保证操作系统的稳定性和安全性，一个进程的地址空间划分为 **用户空间（User space）** 和 **内核空间（Kernel space ）**。

像我们平常运行的应用程序都是运行在用户空间，只有内核空间才能进行系统态级别的资源有关的操作，比如如文件管理、进程通信、内存管理等等。也就是说，我们想要进行 IO 操作，一定是要依赖内核空间的能力。

并且，用户空间的程序不能直接访问内核空间。

当想要执行 IO 操作时，由于没有执行这些操作的权限，只能发起**系统调用**请求操作系统帮忙完成。

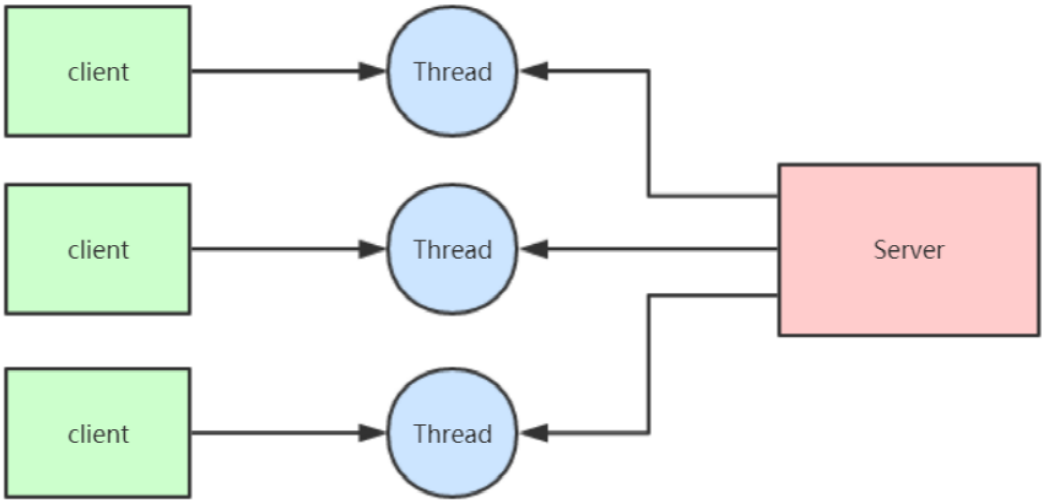
因此，用户进程想要执行 IO 操作的话，必须通过 **系统调用** 来间接访问内核空间

当应用程序发起 I/O 调用后，会经历两个步骤：

1. 内核等待 I/O 设备准备好数据
2. 内核将数据从内核空间拷贝到用户空间。

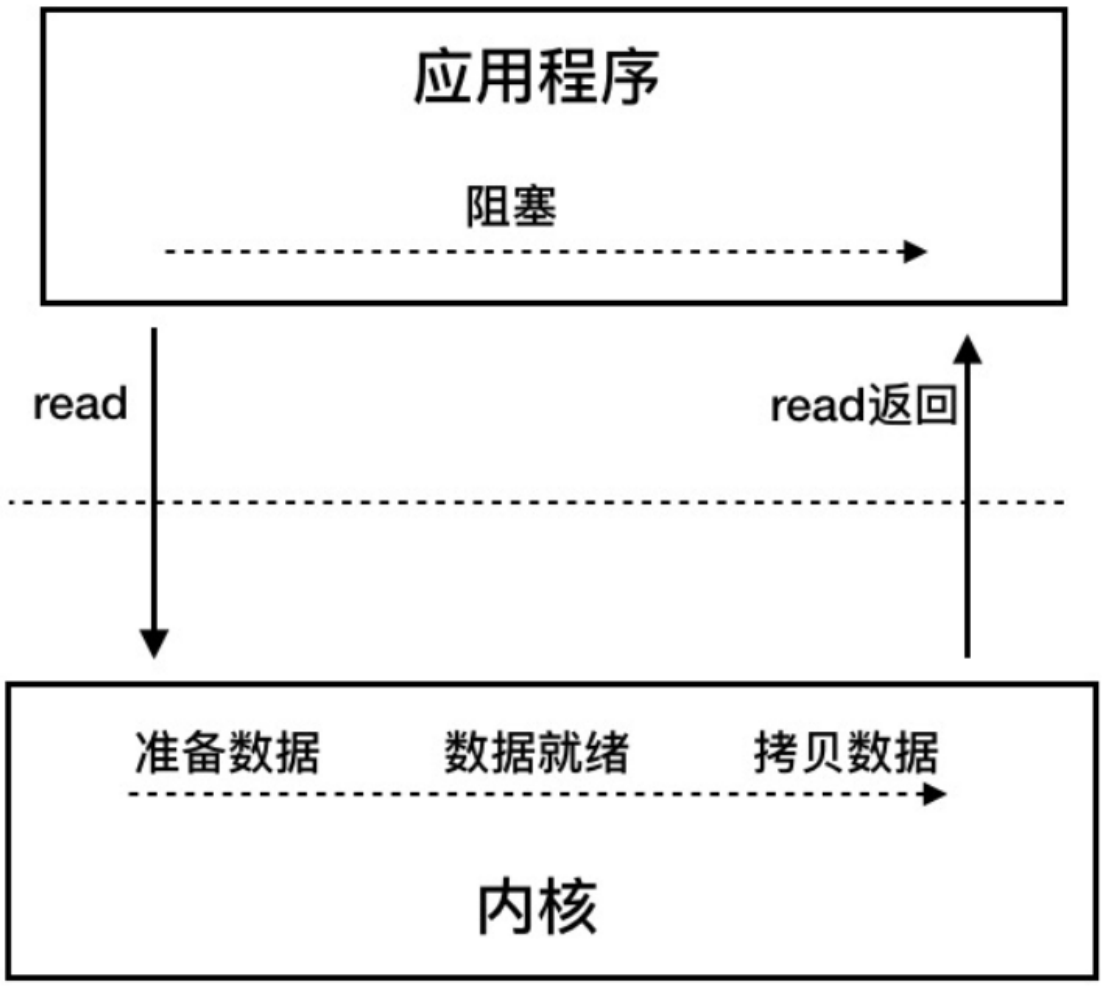
- **BIO（同步阻塞IO）**

BIO



一个客户端连接对应一个处理线程。

同步阻塞



特点：应用程序发起 read 调用后，会一直阻塞，直到在内核把数据拷贝到用户空间

缺点：

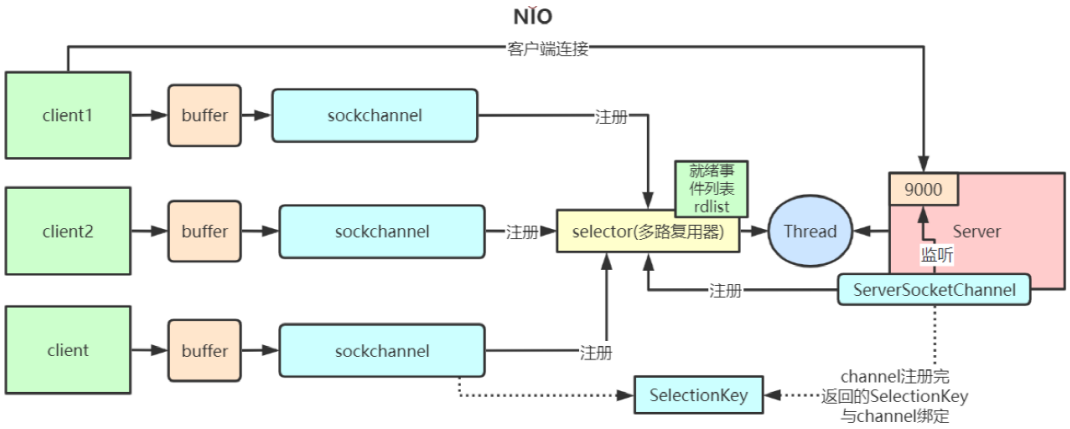
- IO代码里面的read操作是阻塞操作，如果连接不做数据的读写会导致线程阻塞，浪费资源。

- 如果线程太多，会导致服务器的线程太多，压力太大

应用场景：适用于连接数目比较小且固定的架构， 这种方式对服务器资源要求比较高

- **NIO(非阻塞IO)**

  服务器实现模式为一个线程可以处理多个请求(连接)，客户端发送的连接请求都会注册到**多路复用器selector**上，**多路复用器**轮询到连接有IO请求就进行处理



NIO调用流程：

NIO整个调用流程就是Java调用了**操作系统**的**内核函数**来创建Socket，获取到Socket的文件描述符，再创建一个Selector对象，对应操作系统的Epoll描述符，将获取到的Socket连接的文件描述符的事件绑定到Selector对应的Epoll文件描述符上，进行事件的异步通知，这样就实现了使用一条线程，并且不需要太多的无效的遍历，将事件处理交给了操作系统内核(操作系统中断程序实现)，大大提高了效率。

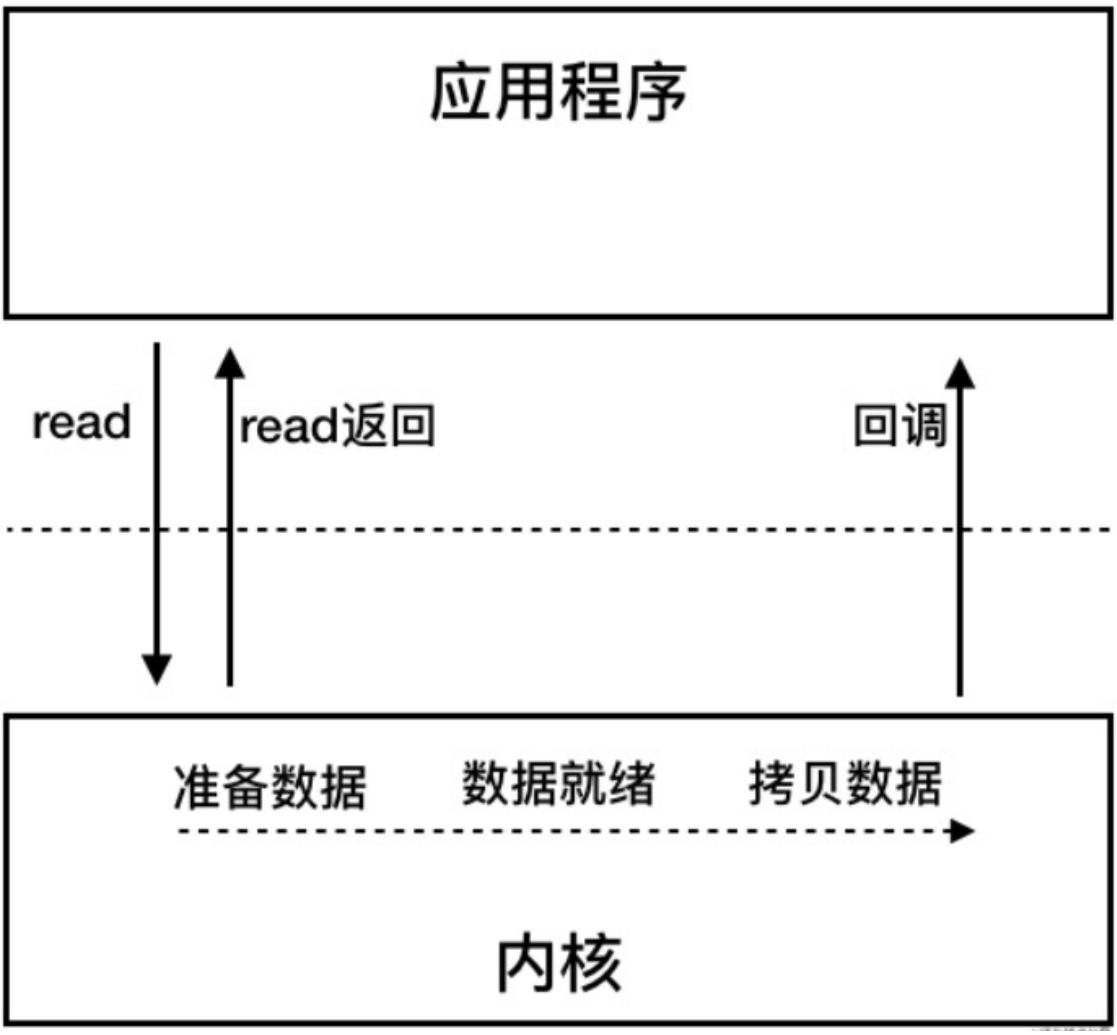
IO多路复用的内核函数（select，poll，epoll）jdk1.4之前使用的是（selct，poll）

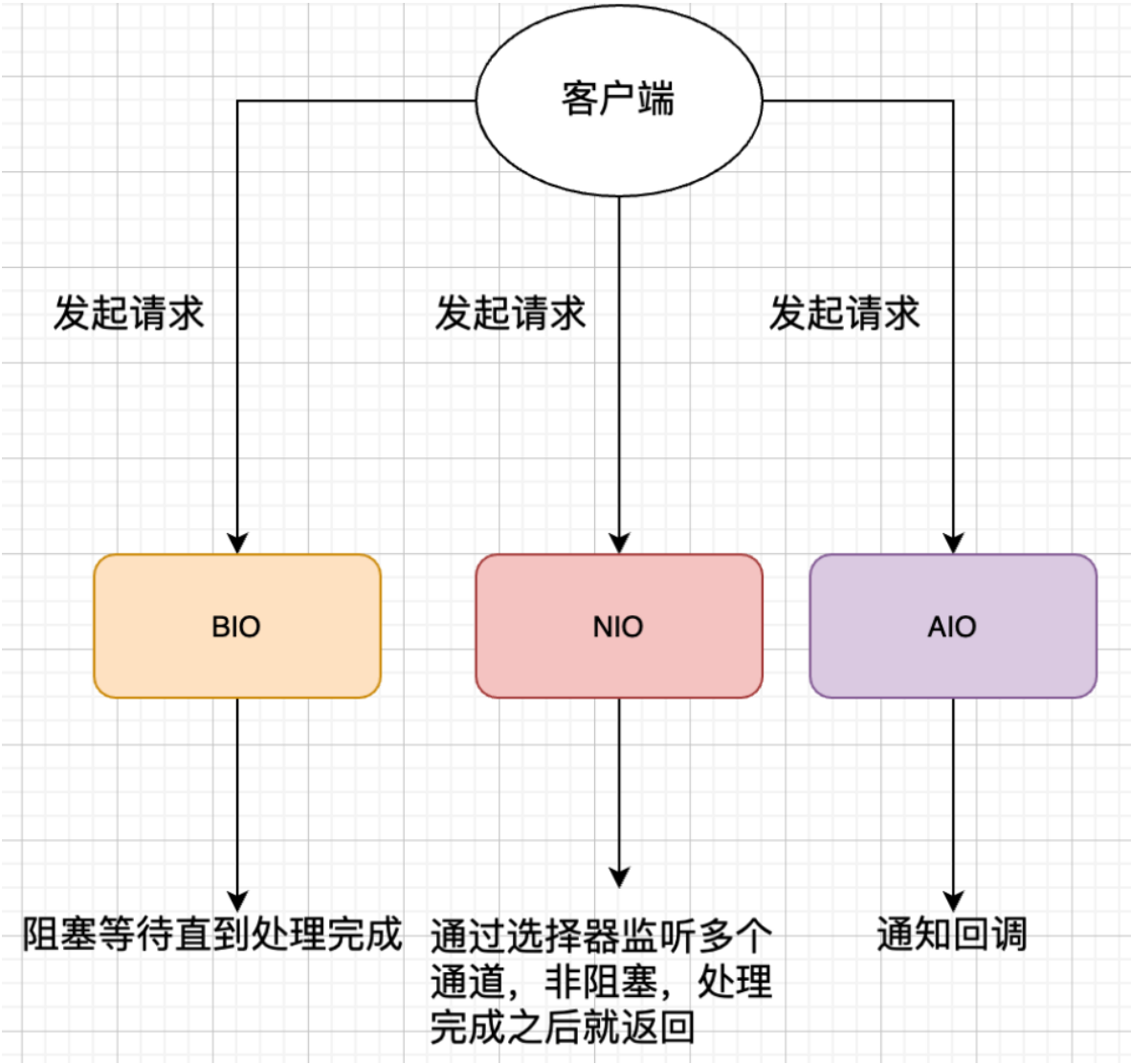
	select	poll	epoll(jdk 1.5及以上)
操作方式	遍历	遍历	回调
底层实现	数组	链表	哈希表
IO效率	每次调用都进行线性遍历，时间复杂度为O(n)	每次调用都进行线性遍历，时间复杂度为O(n)	事件通知方式，每当有IO事件就绪，系统注册的回调函数就会被调用，时间复杂度O(1)
最大连接	有上限	无上限	无上限

- **AIO**

  异步 IO 是基于**事件**和**回调机制**实现的，也就是应用操作之后会直接返回，不会堵塞在那里，当后台处理完成，操作系统会通知相应的线程进行后续的操作

异步





Redis是单线程吗？你是怎么理解redis单线程的概念？

Redis 的单线程主要是指Redis的网络IO和键值对读写操作是由一个线程来完成的，这也是 Redis 对外提供键值存储服务的主要流程。但像Redis 的其他功能，比如**持久化**、集群**数据同步**等，其实是由额外的线程执行的。

Redis是单线程为什么还这么快?

1）纯**内存**操作

Redis **将所有数据放在内存中，所有的运算都是基于内存级别的。**内存的响应时长大约为 10 ns，这是 redis 的 QPS 过万的重要基础。

2）核心是基于**非阻塞的IO多路复用机制**

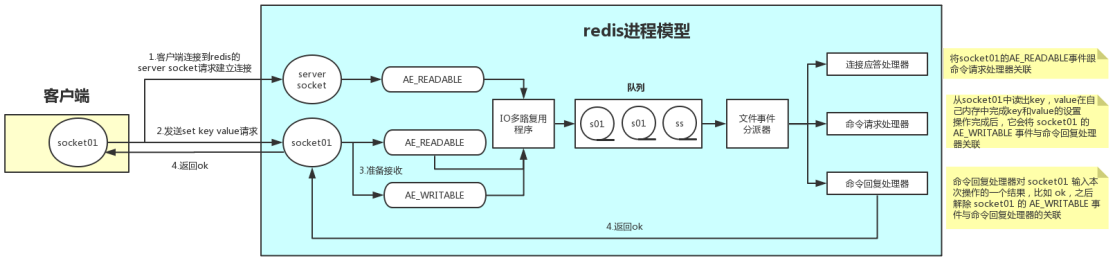
redis 需要处理多个 IO 请求，同时把每个请求的结果返回给客户端。由于 redis 是单线程模型，同一时间只能处理一个 IO 事件，于是 redis 需要在合适的时间暂停对某个 IO 事件的处理，转而去处理另一个 IO 事件，这就需要用到IO多路复用技术了， 就好比一个管理者，能够管理多个socket的IO事件，当选择了哪个socket，就处理哪个socket上的 IO 事件，其他 IO 事件就暂停处理了。

3）单线程避免了**多线程的频繁上下文切换**带来的性能问题。

- 单线程的问题：对于每个命令的执行时间是有要求的。如果某个命令执行过长，会造成其他命令的阻塞，所以 redis 适用于那些需要快速执行的场景。

【注】：由于redis是单线程的，尽量不要插入内存大的key，也要不使用一些耗时的指令（keys），这些耗时的命令也会让redis进行阻塞。

Redis单线程如何处理那么多的并发客户端连接?

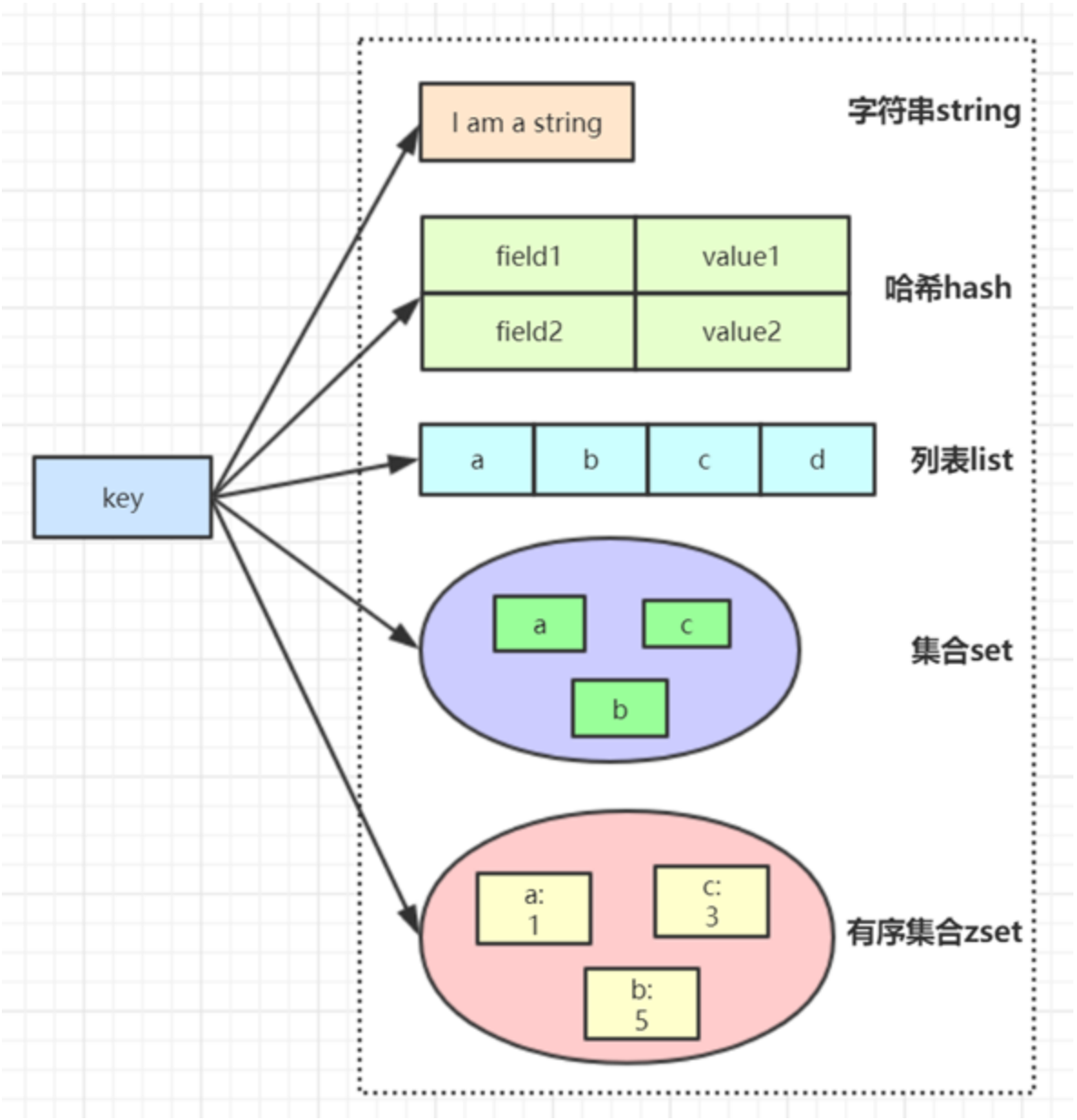


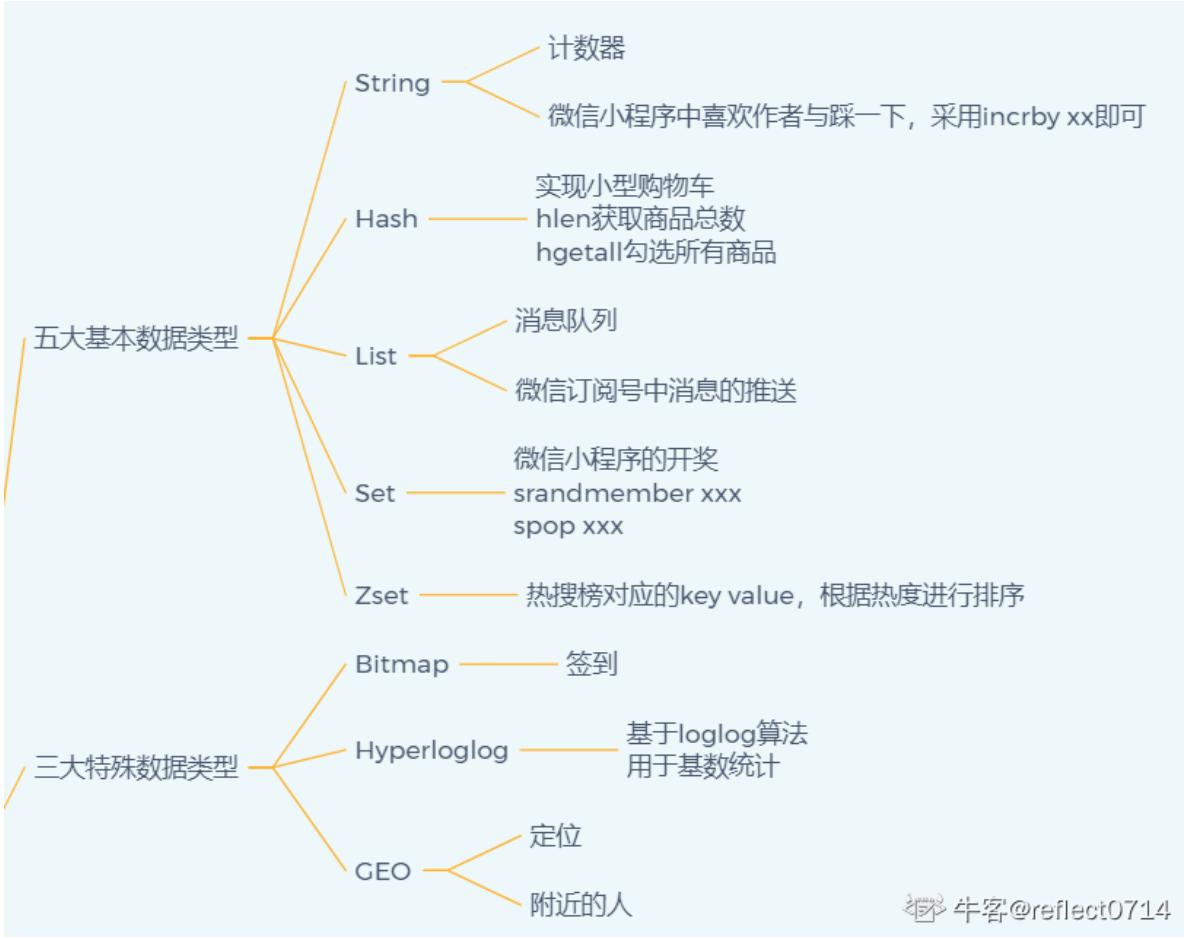
[redis的进程通信的细节]https://www.jianshu.com/p/8f2fb61097b8

原理：redis内部有一个文件事件处理器（多个socket，IO多路复用程序，文件事件分配器，事件处理器），这个文件事件处理器是单线程的。它采用内核函数epoll实现IO多路复用机制，当客户端的多个 socket 会并发产生不同的操作，每个操作对应不同的文件事件，此时IO多路复用程序会监听多个 socket，会将 socket 产生的事件放入**队列**中排队，事件分配器每次从队列中依次取出一个事件，把该事件交给对应的事件处理器进行处理，最终把事件响应的结果返回给客户端。

【注】：事件处理器（连接应答处理器，命令请求处理器，命令回复处理器）

Redis数据结构


key
I am a string
字符串string
field1
value1
field2
value2
哈希hash
a
b
c
d
列表list
a
c
b
集合set
a:
1
c:
3
b:
5
有序集合zset



String

底层数据结构：

String被称为（SDS）简单动态字符串，内部维护着一个字符数组，并且在其内部预分配了一定的空间，以减少内存的频繁分配。

内存分配的分配是这样的：当字符串的长度小于 1MB时，每次扩容都是加倍现有的空间。如果字符串长度超过 1MB时，每次扩容时只会扩展 1MB 的空间。这样既保证了内存空间够用，还不至于造成内存的浪费，字符串最大长度为 512MB。

为什么不同c语言的String?

- 求字符串长度
  C语言的字符串，求字符串的长度只能遍历，时间复杂度是O(N)，**单线程的Redis**表示鸭梨山大，但是现在引入了一个字段来存储字符串的实际长度，时间复杂度瞬间降低成了O(1)。
- 二进制安全
  在C语言中，读取字符串遵循的是“遇零则止”，即，读取字符串，当读取到“\0”，就认为已经读到了结尾，哪怕后面还有字符串也不会读取了，像图片、音频等二进制数据，经常会穿插“\0”在其中，好端端的图片、音频就毁了。但是现在有了一个字段来存储字符串的**实际长度**，读取字符串的时候，先看下这个字符串的长度是多少，然后往后读多少位就可以了。
- 缓冲区溢出
  字符串拼接是开发中常见的操作，C语言的字符串是不记录字符串长度的，一旦我们调用了拼接函数，而没有提前计算好内存，就会产生缓冲区溢出的情况，但是现在引入了free字段，来记录剩余的空间，做拼接操作之前，先去看下还有多少剩余空间，如果够，那就放心的做拼接操作，**不够，就进行扩容。**
- 减少内存重分配次数

1. 空间预分配：当对字符串进行拼接操作的时候，Redis会很贴心的分配一定的剩余空间，这块剩余空间现在看起来是有点浪费，但是我们如果继续拼接，这块剩余空间的作用就出来了。

2. 惰性空间释放：当我们做了字符串缩减的操作，Redis并不会马上回收空间，因为你可能即将又要做字符串的拼接操作，如果你再次操作，还是没有用到这部分空间，Redis也会去回收这部分空间。

应用场景

- 单值缓存

  set key value

  get key

- 对象缓存

  set user:1 value

  mset user:1:name zhuge

- 分布式锁

  setnx product:1 true//返回1，代表获取分布式锁成功

  //执行完业务释放锁

  del product:1

  set product:1 true ex 10 nx //防止程序意外终止导致死锁，设置过期时间

- 计数器

  微信公众号里面文章的阅读量

  incr article:readcount:{文章id}

- Web集群session共享

  spring session + redis实现session

  在我的项目中，我将用户信息用redis进行了缓存，实现了分布式session的问题。

Hash

- **购物车**

(hset key field value)

Hash可以用来做电商购物车，我们可以将用户id为key，商品id作为filed，商品数量作为value

(添加商品：hset cart:1001 1099 1)

(增加数量 hincrby cart:1001 1099 1)

(商品总数 hlen cart:1001)

(删除商品 hdel cart:1001 1099)

list

- **用作消息队列**

  stack（lpush+lpop）

  queue(lpush+rpop)

  BlockingQueue(lpush+Bpop)

- **微博消息和公众号消息**

比如说你关注了那几个公众号，他们一发消息，就往redis里面进行操作

lpush msg:{自己-id} 消息id

查看公众号的最新消息：

lrange msg:{自己-id} 0 -1

set

- **去重**

先把点击了抽奖那个按钮的用户加入到set中

（微信抽奖）sadd key {userId}

srandmember key count 抽奖（重复抽取）

spop key 抽几等奖 ——会先弹出抽完奖的用户，后面在中完奖的用户里面继续抽取

- **点赞，收藏，便签**

点赞：sadd key value

取消点赞：srem key

点赞的用户列表：semembers key

点赞的用户数：scard key

- **社交的关注模型**

共同关注的人

sinter wgl js （交集）

我关注的人也关注他

sismember zou jsSet（存在）

我可能认识的人

sdiff wglSet jsSet（差集）

zset （排序）

底层实现：跳表

跳表全称为跳跃列表，它允许快速查询，插入和删除一个有序连续元素的数据链表。跳跃列表的平均查找和插入时间复杂度都是O(logn)。快速查询是通过维护一个多层次的链表，且每一层链表中的元素是前一层链表元素的子集。一开始时，算法在最稀疏的层次进行搜索，直至需要查找的元素在该层两个相邻的元素中间。这时，算法将跳转到下一个层次，重复刚才的搜索，直到找到需要查找的元素为止。

跳表具有以下性质(**每两个元素提取一个元素作为上一级的索引**)：

（1）跳表是可以实现二分查找的有序链表；

（2）每个元素插入时随机生成它的level；

（3）最低层包含所有的元素；

（4）如果一个元素出现在level(x)，那么它肯定出现在x以下的level中；

（5）每个索引节点包含两个指针，一个向下，一个向右；

（6）跳表查询、插入、删除的时间复杂度为O(log n)，空间复杂度为O(n)，与平衡二叉树接近；

为什么redis选择用跳表而不是使用红黑树来实现有序集合?

有序集合支持插入，删除，查找，有序输出所有元素，查找区间内的所有元素。前四项红黑树都可以完成且时间复杂度与跳表一致。但是，最后一项，红黑树的效率就没有跳表高了。

在跳表中，要查找区间的元素，我们只要定位到两个区间端点在最低层级的位置，然后按顺序遍历元素就可以了，非常高效。而红黑树只能定位到端点后，再从首位置开始每次都要查找后继节点，相对来说是比较耗时的。此外，跳表实现起来很容易且易读，红黑树实现起来相对困难，所以Redis选择使用跳表来实现有序集合。

- **微博热搜**——根据热度进行排行

zincrby key score member

zincrby hotNews:20210307 1 全国两会

展示当日排行前十

zreverse hotNews:20210307 0 10 withsores

BitMap

- **用户签到**

很多网站都提供了签到功能，并且需要展示最近一个月的签到情况，这种情况可以使用 BitMap 来实现。
根据日期 offset = （今天是一年中的第几天） % （今年的天数），key = 年份：用户id。

如果需要将用户的详细签到信息入库的话，可以考虑使用一个一步线程来完成。

- **2. 统计活跃用户（用户登陆情况）**

使用日期作为 key，然后用户 id 为 offset，如果当日活跃过就设置为1。具体怎么样才算活跃这个标准大家可以自己指定。

使用bitcount统计为1的用户，就代表签到的用户。

假如 20201009 活跃用户情况是：[1，0，1，1，0]
20201010 活跃用户情况是：[1，1，0，1，0]

统计连续两天活跃的用户总数：连续两天bittop and login_and login1 login2；至少有一天bittop or login_or login1 login2

HyperLoglog

- HyperLogLog是一种算法，并非redis独有
- 目的是做基数统计，故不是集合，不会保存元数据，只记录数量而不是数值。

- 耗空间极小，支持输入非常体积的数据量
- 核心是基数估算算法，主要表现为计算时内存的使用和数据合并的处理。最终数值存在一定误差
- 使用一般集合或数据结构来处理，当数据量一大就崩了

应用场景：

- 基数不大，数据量不大就用不上，会有点大材小用浪费空间
- 有局限性，就是只能统计基数数量，而没办法去知道具体的内容是什么
- 和bitmap相比，属于两种特定统计情况，简单来说，HyperLogLog 去重比 bitmap 方便很多
- 一般可以bitmap和hyperloglog配合使用，bitmap标识哪些用户活跃（DAU），hyperloglog计数(UV)

GEO

美团，微信中附近的人的功能，我的位置都可以使用该数据结构实现。

geoadd key m1 经度 维度

geopos key m1 查询坐标信息

geodist key m1 m2 查询两个位置之间的距离

georadius 查询周围信息

Redis持久化

有三种持久化机制

- **RDB快照持久化方式**

  save 60 1000 //60秒内有1000个键值改动就可以执行持久化

  原理：在指定的时间内，将内存中的数据快照写入到磁盘中，这个过程会fork一个子进程来将数据写入到一个临时文件中，写入成功后，替换掉之前的dump.rdb文件。

  【注】：配置自动生成的rdb文件后台使用的是bgsave命令

  何时会触发持久化

  - 正常关闭，执行shutdown
  - 默认配置save
  - 执行save/bgsave(save用主进程来持久化，bgsave用子进程持久化)
  - 执行flushall命令
- **AOF持久化**

  原理：redis将修改的每一条指令记录进文件appendonly.aof中(先写入os cache，每隔一段时间fsync到磁盘中)

  appendonly yes 开启aof持久化命令

  appendsync everysec/always/no

  何时会触发aof持久化?

  - 自动重写：auto-aof-rewrite-min-size 64mb auto-aof-rewrite-precentage 100 aof文件指导要达到64m，并且要增长100%
  - 手动重写：bgrewriteaof

  aof重启时也是fork一个子进程
- **混合持久化**

  为什么要有混合持久化?

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。

开启了混合持久化

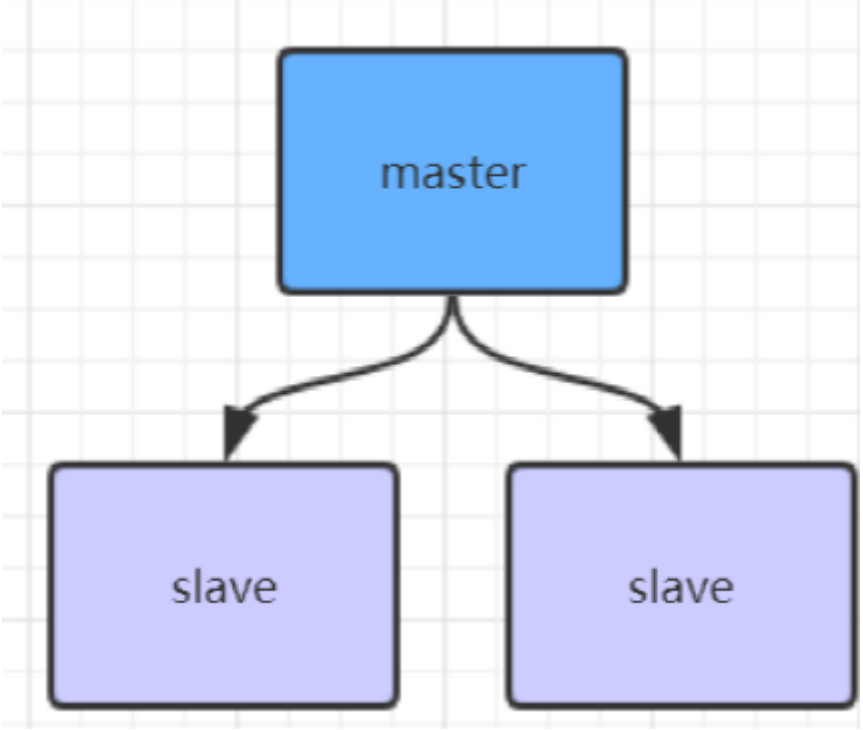
如果开启了混合持久化，AOF在重写时，redis不再是单纯将内存数据转换为追加命令写入AOF文件，而是将重写这一刻之前的内存做RDB快照处理，并且将RDB快照内容和增量的AOF修改内存数据的命令存在一起，都写入新的AOF文件，新的文件一开始不叫appendonly.aof，等到重写完新的AOF文件才会进行改名，覆盖原有的AOF文件，完成新旧两个AOF文件的替换。

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，因此重启效率大幅得到提升

Redis主从架构

- **采用主从架构的原因**：

如果系统的QPS超过10W+，甚至是百万以上的访问，单机的Redis支撑不了过大的并发，如果大量的访问过来，单机Redis被流量打死，则系统的瓶颈就卡到Redis上了。所以我们采用**主从架构+读写分离**，来支撑10w+的qps。



主从复制的原理：

- **全量复制（2.8版本之前）**

如果master配置slave，不管这个slave是否是第一次连接master，slave都会发送也给SYNC命令给master，master收到命令后，自己执行bgsave命令，生成最新的rdb快照，（而这持久化期间master会继续接收客户端的请求），持久化完成后master将rdb文件发送给slave，并将在rdb之后的新数据的缓存也发送给slave。

当master与slave之间的连接由于某些原因而断开时，slave能够自动重连Master，如果master收到了多个slave并发连接请求，它只会进行一次持久化，而不是一个连接一次，然后再把这一份持久化的数据发送给多个并发连接的slave。

- **部分复制（2.8）版本之后**

redis改用支持部分数据复制的命令PSYNC去master同步数据，slave与master能够在网络连接断开重连后只进行部分数据复制。master会在其内存中创建一个复制数据用的缓存队列，master和所有的slave都维护了该数据的offset和master id。当网络断开后，slave重新连接后，如果master的id发生了变化或者节点数据下标太旧，已经不在缓存里面了，slave将会进行一次全量复制。

优点：

主从模式下，当某一节点损坏时，因为其会将数据备份到其它 Redis 实例上，这样做在很大程度上可以恢复丢失的数据。

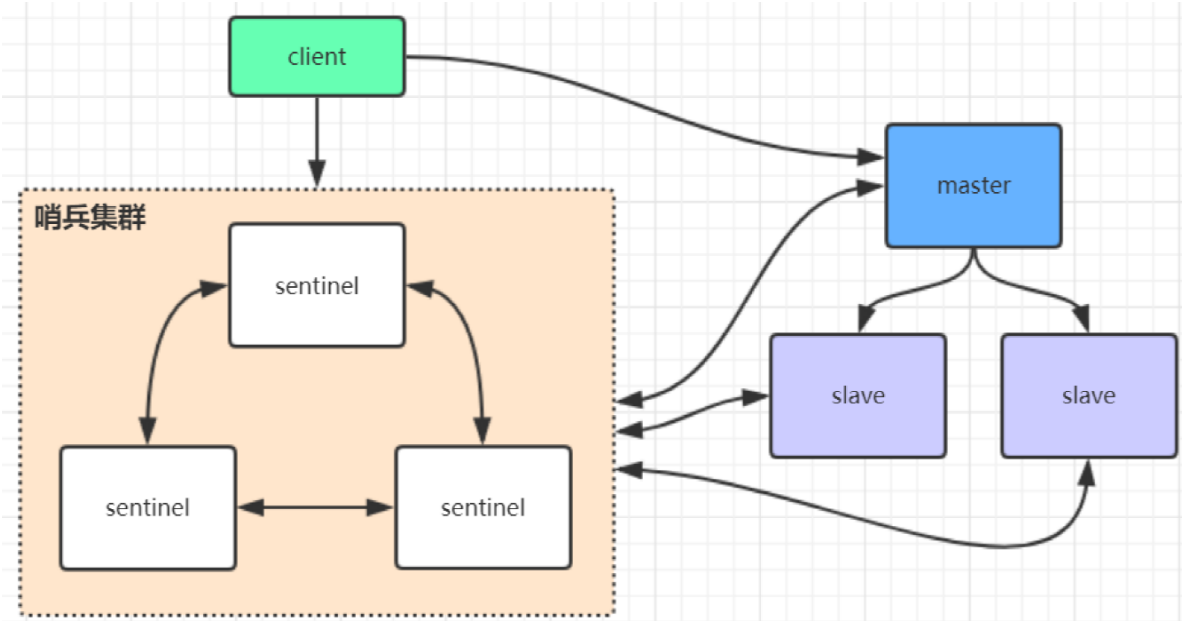
主从模式下，主节点和从节点是读写分离的。使用者不仅可以从主节点上读取数据，还可以很方便的从从节点上读取到数据，这在一定程度上缓解了主机的压力。

从节点也是能够支持写入数据的，只不过从从节点写入的数据不会同步到主节点以及其它的从节点下。

缺点：

Redis 在主从模式下，**只有主节点提供服务必须保证主节点不会宕机**——一旦主节点宕机，其它节点不会竞争称为主节点，此时，Redis将丧失写的能力。这点在生产环境中，是致命的。

Redis哨兵集群



什么是哨兵模式：

哨兵模式是一种特殊的模式，首先Redis提供了哨兵的命令，哨兵是一个独立的进程，作为进程，它会独立运行。其原理是**哨兵通过发送命令，等待Redis服务器响应，从而监控运行的多个Redis实例。**然而一个哨兵进程对Redis服务器进行监控，可能会出现问题，为此，我们可以使用多个哨兵进行监控。各个哨兵之间还会进行监控，这样就形成了多哨兵模式。

【注】：哨兵架构下client端第一次从哨兵找出redis的主节点，后续就直接访问redis的主节点，不会每次都通过 sentinel代理访问redis的主节点。除非master节点宕机会重写选举出新的master节点。

哨兵的作用：

- 通过发送命令，让Redis服务器返回监控其运行状态，包括主服务器和从服务器。
- 当哨兵监测到master宕机，会自动将slave切换成master，然后通过**发布订阅模式**通知其他的从服务器，修改配置文件，让它们切换主机。

哨兵是如何知道某个节点是不可用的？

每个哨兵节点每秒通过 **ping** 去进行心跳监测（包括所有redis实例和sentinel同伴），并根据回复判断节点是否在线。

如果某个 sentinel 线程发现master没有在给定时间内响应这个PING，则这个 sentinel 线程认为master是不可用的，这种情况叫主观宕机。

master宕机后：会出现一个故障切换的过程（failover）

具体过程：假设主服务器宕机，哨兵1先检测到这个结果，系统并不会马上进行failover过程，仅仅是哨兵1主观的认为master不可用，认为该master已经**主观宕机**。当后面的哨兵也检测到master不可用，并且数量达到一定值时，此时认为该master已经**客观宕机**。那么哨兵之间就会进行一次投票，选举出一个哨兵leader执行故障转移的操作，进行failover操作。故障切换成功后，哨兵leader就会通过**发布订阅**模式，让各个哨兵把自己监控的从服务器实现切换主机。

Redis集群（哨兵集群）为什么至少需要三个master节点，并且推荐节点数为奇数?

因为新master的选举需要大于**半数**的集群master节点同意才能选举成功，如果只有两个master节点，当其中一个挂了，是达不到选举新master的条件的。

奇数个master节点可以在满足选举该条件的基础上节省一个节点，比如三个master节点和四个master节点的集群相比，大家如果都挂了一个master节点都能选举新master节点，如果都挂了两个master节点都没法选举新master节点了，所以奇数的master节点更多的是从**节省机器资源**角度出发说的。

哨兵模式的优缺点：

优点：

- 监控：它会监听主服务器和从服务器之间是否在正常工作。
- 通知：它能够通过API告诉系统管理员或者程序，集群中某个实例出了问题。
- 故障转移：它在主节点出了问题的情况下，会在所有的从节点中竞选出一个节点，并将其作为新的主节点。
- 提供主服务器地址：它还能够向使用者提供当前主节点的地址。这在故障转移后，使用者不用做任何修改就可以知道当前主节点地址。

缺点：

- 哨兵的配置略微复杂，并且性能和高可用性等各方面表现一般，特别是在**主从切换**的瞬间存在**访问瞬断**的情况。
- 哨兵模式只有一个**主节点**对外提供服务，没法支持很高的并发。
- 且单个主节点内存也不宜设置得过大，否则会导致持久化文件过大，影响数据恢复或主从同步的效率。

Redis高可用集群

什么是redis高可用集群?

Redis 集群是一个由**多个主从节点群组成**的分布式服务器群，需要将每个节点设置成集群模式。

优点：

- 它具有复制、高可用和分片特性
- Redis集群不需要 sentinel 哨兵也能完成节点移除和故障转移的功能
- 这种集群模式没有中心节点，可水平扩展，据官方文档称可以线性扩展到上万个节点(官方推荐不超过1000个节点)
- redis集群的性能和高可用性均优于之前版本的哨兵模式，且集群配置非常简单

为什么redis集群的最大槽数是16348（2^14）个?

- 在redis节点发送心跳包时需要把所有的槽放到这个心跳包里，以便让节点知道当前**集群信息**，16384=16k，在发送心跳包时使用`char`进行bitmap压缩后是2k（`2 * 8 (8 bit) * 1024(1k) = 16K`），也就是说使用2k的空间创建了16k的槽数。
- 虽然使用CRC16算法最多可以分配65535（2^16-1）个槽位，65535=65k，压缩后就是8k（`8 * 8 (8 bit) * 1024(1k) =65K`），也就是说需要需要8k的心跳包，这个ping的消息头太大了，浪费带宽；并且一般情况下一个redis集群不会有超过1000个master节点，超过1000个节点，会导致网络拥堵，所以16k的槽位对于1000个master节点已经够用了，没必要再扩展到65535。

Redis-Cluster的原理

- Redis Cluster 将所有数据根据 key 划分到 16384 个槽位(slots)中，每个节点（小集主从群）负责其中一部分槽位。槽位的信息存储于每个节点中。
- 当 Redis Cluster 的客户端来连接集群时（原则上只要连接到一个节点即可），它也会得到一份集群的**槽位配置信息**并将其缓存在**客户端本地**。这样当客户端要查找某个key 时，可以直接定位到目标节点，避免了到服务器再进行计算与寻找的资源消耗。
- 若发生扩容缩容等，槽位的信息可能会存在客户端与服务器不一致的情况，还需要纠正机制来实现槽位信息的校验调整。
- 当主节点挂掉时，会发起选举让从节点成为主节点。

Redis集群槽位定位算法

Cluster 默认会对 key 值使用crc16算法进行hash得到一个整数值，然后用这个整数值对 16384 进行取模来得到具体槽位。

Redis集群跳转重定向

当客户端向一个错误的节点发出了指令（常发生于扩容缩容后），该节点会发现指令的 key 所在的槽位并不归自己管理，这时它会向客户端发送一个携带**目标操作节点地址**的特殊跳转指令：

1. 告诉客户端去连这个节点去获取数据，客户端收到指令后除了跳转到正确的节点上去操作
2. 同步更新纠正本地的**槽位映射表缓存**，后续所有 key 将使用新的槽位映射表

Redis集群master的选举策略?

当 slave 发现自己的 master 变为 FAIL 状态时，便尝试进行 Failover，以期成为新的 master。由于挂掉的master可能会有多个slave，从而存在多个slave竞争成为master节点的过程， 其过程如下：

- slave 发现自己的 master 变为 FAIL，然后将自己记录的集群周期 currentEpoch 加一，并广播 FAILOVER_AUTH_REQUEST 信息，表示请求成为新 Master。
- 其他节点收到该信息，只有 master 响应，判断请求者的合法性，并发送FAILOVER_AUTH_ACK，对每一个epoch只发送一次ack。
- 尝试 Failover 的 slave 收集FAILOVER_AUTH_ACK。
- 超过半数收到ack的 Slave 变成新 Master，然后广播通知其他集群节点

【注】：因此每个集群至少要3个slave，因为只有2个slave时，投票率=50%；另外，选举也是需要时间的，所以选举过程中可能造成数据丢失，但极少数据丢失redis是允许的。

> 两点注意：
> 为了避免网络抖动造成的假死，master 无响应后会有**一定时间**的等待。
> 每个slave广播REQUEST并不是同时的，而是有一个延迟时间。DELAY = 500ms + random(0 ~ 500ms) + SLAVE_RANK * 1000ms。

> 其中，**SLAVE_RANK** 表示此slave已经从master复制数据的总量的rank。Rank越小代表已复制的数据越新。这种方式下，持有最新数据的slave将会首先发起选举（理论上）

网络抖动问题：

真实世界的机房网络往往并不是风平浪静的，它们经常会发生各种各样的小问题。比如网络抖动就是非常常见的一种现象，突然之间部分连接变得不可访问，然后很快又恢复正常。

为解决这种问题，RedisCluster 提供了一种选项**cluster-node-timeout**，表示当某个节点持续 timeout 的时间失联时，才可以认定该节点出现故障，需要进行主从切换。如果没有这个选项，网络抖动会导致主从频繁切换(数据的重新复制)。

集群（主从）中脑裂数据丢失的问题

redis的**集群脑裂**是指因为网络问题，导致redis master节点跟redis slave节点处于不同的网络分区，此时在集群中因为slave节点无法感知到master的存在，所以将slave节点提升为master节点。此时存在两个不同的master节点。

集群脑裂问题中，如果客户端还在基于原来的master节点继续写入数据，那么新的master节点将无法同步这些数据，当网络问题解决之后，集群将原先的master节点降为slave节点，此时再从新的master中同步数据，将会造成大量的数据丢失。

> 解决方案：min-slaves-to-write 1 //本例集群数是3，写数据成功最少同步的slave数量，这个数量可以模仿大于半数机制配置，比如集群总共三个节点可以配置1，加上leader就是2，超过了半数

异步复制数据导致数据丢失的问题

因为master->slave的数据同步是异步的，所以可能存在部分数据还没有同步到slave，master就宕机了，此时这部分数据就丢失了。

> 解决方案：min-slaves-to-write 1 min-slaves=max-lag 10 至少有一个salve，数据复制和同步的延迟不能超过10s，如果不符合这个条件，那么master将不会接受任何请求。

有了min-slaves-max-lag这个配置，就可以确保，一旦slave复制数据和ack延时太长，就认为master宕机后损失的数据太多了，那么就拒绝写请求，这样可以把master宕机时由于部分数据未同步到slave导致的数据丢失降低到可控范围内

分布式锁

dutianxu.dtx@mybank.cn 蚂蚁金服网上银行

MySQL实现分布式锁

- 基于数据库实现排他锁

  将增加一个字段并将其设为unique key，对该键做了唯一性约束，这里如果有多个请求同时提交到数据库的话，数据库会保证只有一个操作可以成功。

  insert into table(uniqueKey) values 1;

- 基于数据库实现乐观锁

  将增加一个字段version，先获取锁select version from table where id=1;

  再占有锁，进行更新version，update table set version = version+1 where id=1 and version =1

  如果没有更新到一行数据，则说明这个资源已经被别人占用了。

MySQL实现分布式锁的缺点：

Redis实现分布式锁

使用redis自带命令：setnx

```
String client=UUID.randomUUID().toString();
String lockKey="product";
try{
 boolean
lock=redisTemplate.opsForValue().expire(lockKey,client,1,TimeUnit.SECONDS);//给锁设置过期时间,并判断是否获取锁成功
}

    if(lock){
    执行业务代码
    }else{
        log.info("获取锁失败");
    }finally{

    if(lock.equals(redisTemplate.opsForValue().get(client))){
        redisTemplate.delete(client);//释放锁(在这里可以使得，自己加的锁由自己释放)
    }

}
```

使用redisson实现分布式锁

```
RLock redissonLock=redisson.getLock(lockKey);
//加锁,
redissonLock.lock();//这里有锁续命的过程，具体时间每过设置时间的1/3就重新设置过期时间。

finally{
    redisson.unLock();
}
//没有加锁成功的线程，一直尝试自旋进行加锁
```

redis实现分布式锁的缺点：

在这种场景（主从结构）中存在明显的竞态:
 客户端A从master获取到锁，在master将锁同步到slave之前，master宕掉了。slave节点被晋级为master节点，客户端B取得了同一个资源被客户端A已经获取到的另外一个锁。**安全失效**!

zookeeper实现分布式锁

zookeeper的数据结构就像一棵树，这棵树由节点组成，这种节点叫做Znode。

Znode分为四种类型：

```
1.持久节点：zk客户端与服务端断开连接之后，该节点依然存在。
2.持久顺序节点：根据创建的时间顺序给该节点名称进行编号
3.临时节点：zk客户端与服务端断开连接之后，该节点会删除。
4.临时顺序节点：根据创建的时间进行编号，断开连接之后会删除。
```

获得锁的过程（监听前一个节点）：首先，在Zookeeper当中创建一个持久节点ParentLock。当第一个客户端想要获得锁时，需要在ParentLock这个节点下面创建一个**临时顺序节点** Lock1，然后查找根节点下面的所有临时节点并排序，如果当前节点是最小的，则成功获得锁；当后面有一个客户端前来获取锁，则在根节点下面再创建一个临时序号节点，同理查找当前节点是否是最小的，如果不是，则注册

一个Watcher用于监听比排序比他考前的一个节点的存活。client2枪锁失败，进入等待状态。

羊群效应：在等待锁获得期间，所有等待节点都在监听同一个Lock节点，一但lock节点变更，则所有等待节点都会被触发，然后在同时反查Lock 子节点。如果等待队列过大会使用Zookeeper承受非常大的流量压力。

zk实现分布式锁的缺点：

性能上可能并没有缓存服务那么高。因为每次在创建锁和释放锁的过程中，都要动态创建、销毁瞬时节点来实现锁功能。ZK中创建和删除节点只能通过Leader服务器来执行，然后将数据同不到所有的Follower机器上。

> 其实，使用Zookeeper也有可能带来并发问题，只是并不常见而已。考虑这样的情况，由于网络抖动，客户端可ZK集群的session连接断了，那么zk以为客户端挂了，就会删除临时节点，这时候其他客户端就可以获取到分布式锁了。就可能产生并发问题。这个问题不常见是因为zk有重试机制，一旦zk集群检测不到客户端的心跳，就会重试，Curator客户端支持多种重试策略。多次重试之后还不行的话才会删除临时节点。（所以，选择一个合适的重试策略也比较重要，要在锁的粒度和并发之间找一个平衡。）

Redis事务

redis事务的三个阶段

- 开始事务
- 命令入队
- 执行事务

Redis事务相关命令：

watch key1 key2 ... : 监视一或多个key,如果在事务执行之前，被监视的key被其他（事务）命令改动，则事务被打断 （ 类似乐观锁 ）

multi : 标记一个事务块的开始（ queued ）

exec : 执行所有事务块的命令 （ 一旦执行exec后，之前加的监控锁都会被取消掉 ）

discard : 取消事务，放弃事务块中的所有命令

unwatch : 取消watch对所有key的监控

【注】：若在事务队列中存在命令性错误（类似于Java编译性错误get22 key），则执行exec命令时，所有命令都不会执行

若在事务队列中存在语法性错误（incr k1），则执行命令时，其他命令都会执行，错误命令抛出异常。

Redis缓存淘汰策略

现象：当Redis内存超出物理内存限制后，内存的数据会开始和磁盘产生频繁的交换，交换会使得redis的性能急剧下降，而在生产环境中是不允许出现redis的交换行为的。redis提供了配置参数maxmemerory(一般设置为最大物理内存的3/4)；当超出maxmem后，redis将会执行缓存淘汰策略。

淘汰策略有哪些

3.0

1. volatile-lru：从设置过期时间的数据集（server.db[i].expires）中挑选出最近最少使用的数据淘汰。没有设置过期时间的key不会被淘汰，这样就可以在增加内存空间的同时保证需要持久化的数据不会丢失。

2. volatile-ttl：除了淘汰机制采用LRU，策略基本上与volatile-lru相似，从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，ttl值越大越优先被淘汰。
3. 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。当内存达到限制无法写入非过期时间的数据集时，可以通过该淘汰策略在主键空间中随机移除某个key。
4. allkeys-lru：从数据集（server.db[i].dict）中挑选最近最少使用的数据淘汰，该策略要淘汰的key面向的是全体key集合，而非过期的key集合。
5. allkeys-random：从数据集(server.db[i].dict）中选择任意数据淘汰。
6. no-enviction：禁止驱逐数据，也就是当内存不足以容纳新入数据时，新写入操作就会报错，请求可以继续进行，线上任务也不能持续进行，采用no-enviction策略可以保证数据不被丢失，这也是**系统默认**的一种淘汰策略。

4.0版本

7.allkeys-lfu 最近访问次数最少的被淘汰

8.volatile-lfu

怎么选择淘汰策略

1. 在Redis中，数据有一部分访问频率较高，其余部分访问频率较低，或者无法预测数据的使用频率时，设置allkeys-lru是比较合适的。
2. 如果所有数据**访问概率大致相等**时，可以选择allkeys-random。
3. 如果研发者需要通过**设置不同的ttl**来判断数据过期的先后顺序，此时可以选择volatile-ttl策略。
4. 如果希望一些数据能**长期被保存**，而一些数据可以被淘汰掉时，选择volatile-lru或volatile-random都是比较不错的。
5. 由于设置expire会消耗额外的内存，如果计划避免Redis内存在此项上的浪费，可以选用allkeys-lru 策略，这样就可以不再设置过期时间，高效利用内存了。

手写LRU

采用双向链表加HashMap

```
public class LRUCache{

    private int capacity;
    private Map<Integer, ListNode> map; //key->node
    private ListNode head;  // dummy head
    private ListNode tail;  // dummy tail

    public LRUCache(int capacity) {
        this.capacity = capacity;
        map = new HashMap<>();
        head = new ListNode(-1, -1);
        tail = new ListNode(-1, -1);
        head.next = tail;
        tail.pre = head;
    }

    public int get(int key) {
        if (!map.containsKey(key)) {
            return -1;
        }
        ListNode node = map.get(key);
```

```
        // 先删除该节点，再接到尾部
        node.pre.next = node.next;
        node.next.pre = node.pre;
        moveToTail(node);

        return node.val;
    }

    public void put(int key, int value) {
        // 直接调用这边的get方法，如果存在，它会在get内部被移动到尾巴，不用再移动一遍,直接修改值即可
        if (get(key) != -1) {
            map.get(key).val = value;
            return;
        }
        // 若不存在，new一个出来,如果超出容量，把头去掉
        ListNode node = new ListNode(key, value);
        map.put(key, node);
        moveToTail(node);

        if (map.size() > capacity) {
            map.remove(head.next.key);
            head.next = head.next.next;
            head.next.pre = head;
        }
    }

    // 把节点移动到尾巴
    private void moveToTail(ListNode node) {
        node.pre = tail.pre;
        tail.pre = node;
        node.pre.next = node;
        node.next = tail;
    }

    // 定义双向链表节点
    private class ListNode {
        int key;
        int val;
        ListNode pre;
        ListNode next;

        public ListNode(int key, int val) {
            this.key = key;
            this.val = val;
            pre = null;
            next = null;
        }
    }
}
```

为什么要采用双向链表?

我们涉及到删除元素的操作，链表删除元素除了自己本身的指针信息，还需要记录前驱节点的指针信息，我们要求我们的时间复杂度是O(1)，当我们采用双向链表，所以我们能直接获取到前驱节点的指针。

为什么hash表中已经存储了key，那么链表中为什么还要存储 key 和 value 呢，只存入 value 不就行了?

我们删除链表中的节点，需要借助双向链表来实现O（1），删除过后，需要在hash表中进行删除，而删除hash表中对应的key，只能从链表中获取key。

Redis常见的性能问题

缓存雪崩

面试管你好，我了解的，目前**电商**首页以及**热点数据**都会去做缓存，一般缓存都是定时任务去刷新，或者是查不到之后去更新的，定时任务刷新就有一个问题。

举一个简单的例子：如果所有首页的Key失效时间都是12小时，中午12点刷新的，我零点有个秒杀活动大量，此时用户涌入，假设当时每秒 6000 个请求，本来缓存在可以扛住每秒 5000 个请求，但是缓存当时所有的Key都失效了。此时 1 秒 6000 个请求全部落数据库，数据库必然扛不住，它会报一下警，真实情况可能DBA都没反应过来就直接挂了。此时，如果没用什么特别的方案来处理这个故障，DBA 很着急，重启数据库，但是数据库立马又被新的流量给打死了。这就是我理解的缓存雪崩。

同一时间大面积失效，那一瞬间Redis跟没有一样，那这个数量级别的请求直接打到数据库几乎是灾难性的，如果打挂的是一个用户服务的库，那其他依赖他的库所有的接口几乎都会报错，你怎么重启用户都会把你打挂，等你能重启的时候，用户早就睡觉去了，并且对你的产品失去了信心，什么垃圾产品。

缓存雪崩解决方案：

- 在批量往Redis存数据的时候，把每个Key的**失效时间都加个随机值**就好了，这样可以保证数据不会在同一时间大面积失效。
- 或者设置**热点数据永远不过期**，有更新操作就更新缓存就好了（比如运维更新了首页商品，那你刷下缓存就完事了，不要设置过期时间），电商首页的数据也可以用这个操作，保险。
- 使用主从模式或集群模式来尽量保证缓存服务的高可用。

缓存穿透

我理解的缓存穿透是指缓存和数据库中**都没有的数据**，而用户不断发起请求，我们数据库的 id 都是1开始自增上去的，如发起为id值为 -1 的数据或 id 为特别大，不存在的数据。这时的用户很可能是攻击者，攻击会导致数据库压力过大，严重会击垮数据库。

缓存穿透解决方案：

- 缓存穿透我会在**接口层**增加校验，比如用**户鉴权校验**，参数做校验，不合法的参数直接代码Return，比如：id 做基础校验，id <=0的直接**拦截**等。
- 采用布隆过滤器，使用布隆过滤器需要把所有数据提前放入布隆过滤器，并且在增加数据时也要往布隆过滤器里放。

  布隆过滤器就是一个大型的**位数组**和几个不一样的**无偏 hash 函数**。所谓无偏就是能够把元素的 hash 值算得比较均匀。向布隆过滤器中添加 key 时，会使用多个 hash 函数对 key 进行 hash，算得一个整数索引值然后对位数组长度进行**取模**运算得到一个位置，每个 hash 函数都会算得一个不同的位置。再把位数组的这几个位置都置为 1 就完成了 add 操作。

  向布隆过滤器询问 key 是否存在时，跟 add 一样，也会把 hash 的几个位置都算出来，看看位数组中这几个位置是否都为 1，只要有一个位为 0，那么说明布隆过滤器中这个key 不存在。如果都是 1，这并不能说明这个key 就一定存在，只是极有可能存在，因为这些位被置为 1 可能是因为其它的 key 存在所致。

  如果布隆过滤器中key存在，再去缓存中去查询。

缓存击穿（失效）

我理解的缓存穿透首先是，这个跟缓存雪崩有点像，但是又有一点不一样，缓存雪崩是因为**大面积的缓存**失效，打崩了DB，而缓存击穿不同的是缓存击穿是指**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，就像在一个完好无损的桶上凿开了一个洞。

- 当前key是一个热点的key（例如一个热门的娱乐新闻），并发量非常大
- 重建缓存不能在短时间内完成，可能是一个复杂计算，例如复杂的sql，复杂的IO，多个依赖

要解决的问题主要是避免大量线程同时重建缓存

缓存击穿解法方案：

- 一是设置热点数据永不过期。
- 可以利用**互斥锁**来解决，此方法只允许一个线程重建缓存，其他线程等待缓存的线程执行完，重新从缓存中获取数据即可。

缓存与数据库双写不一致

1.双写不一致：线程1先写（10），此时线程2后写（6），线程2并更新了缓存（6），然后线程1更新缓存（10）。

2.读写不一致：线程1进行写数据（10），并删除缓存，此时线程3读数据库（10），然后线程2写数据库（6），删除缓存，最后线程3更新缓存（10）。

缓存不一致的解决方案：

- 对于**并发几率很小**的数据(如个人维度的订单数据、用户数据等)，这种几乎不用考虑这个问题，很少会发生缓存不一致，可以给缓存数据加上过期时间，每隔一段时间触发读的主动更新即可。
- 就算**并发很高**，如果业务上能容忍短时间的缓存数据不一致(如商品名称，商品分类菜单等)，缓存加上过期时间依然可以解决大部分业务对于缓存的要求。
- 如果不能容忍缓存数据不一致，可以通过加**读写锁**保证并发读写或写写的时候按顺序排好队，读读的时候相当于无锁。