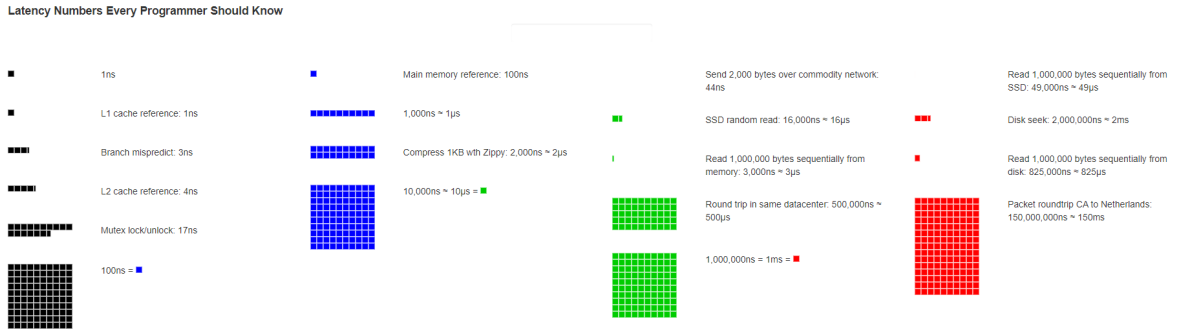
1.背景

说到缓存就不得不提Google工程师Jeff Dean在关于分布式系统PPT文档列出来的CPU内存访问速度，磁盘和网络速度，具体见

下图是一个具体展示：



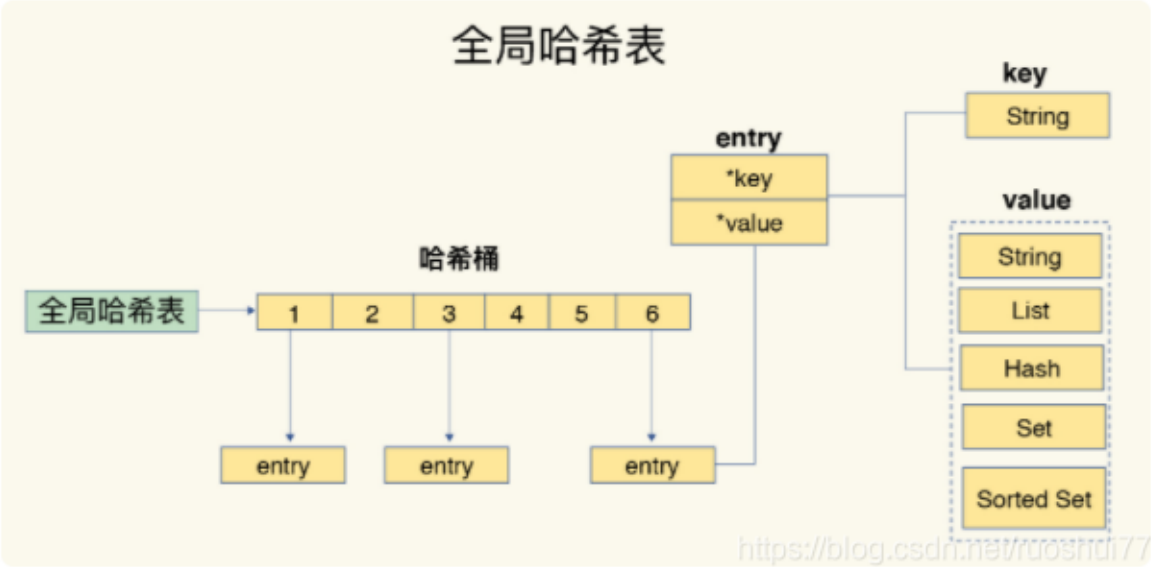
缓存的本质是解决系统各级处理速度不一致，采用以空间换时间的策略，也是微服务中常用的提高性能性能的方法—通过缓存使用频率高、读写比较大的数据，提升系统的性能。

2.常见的缓存技术

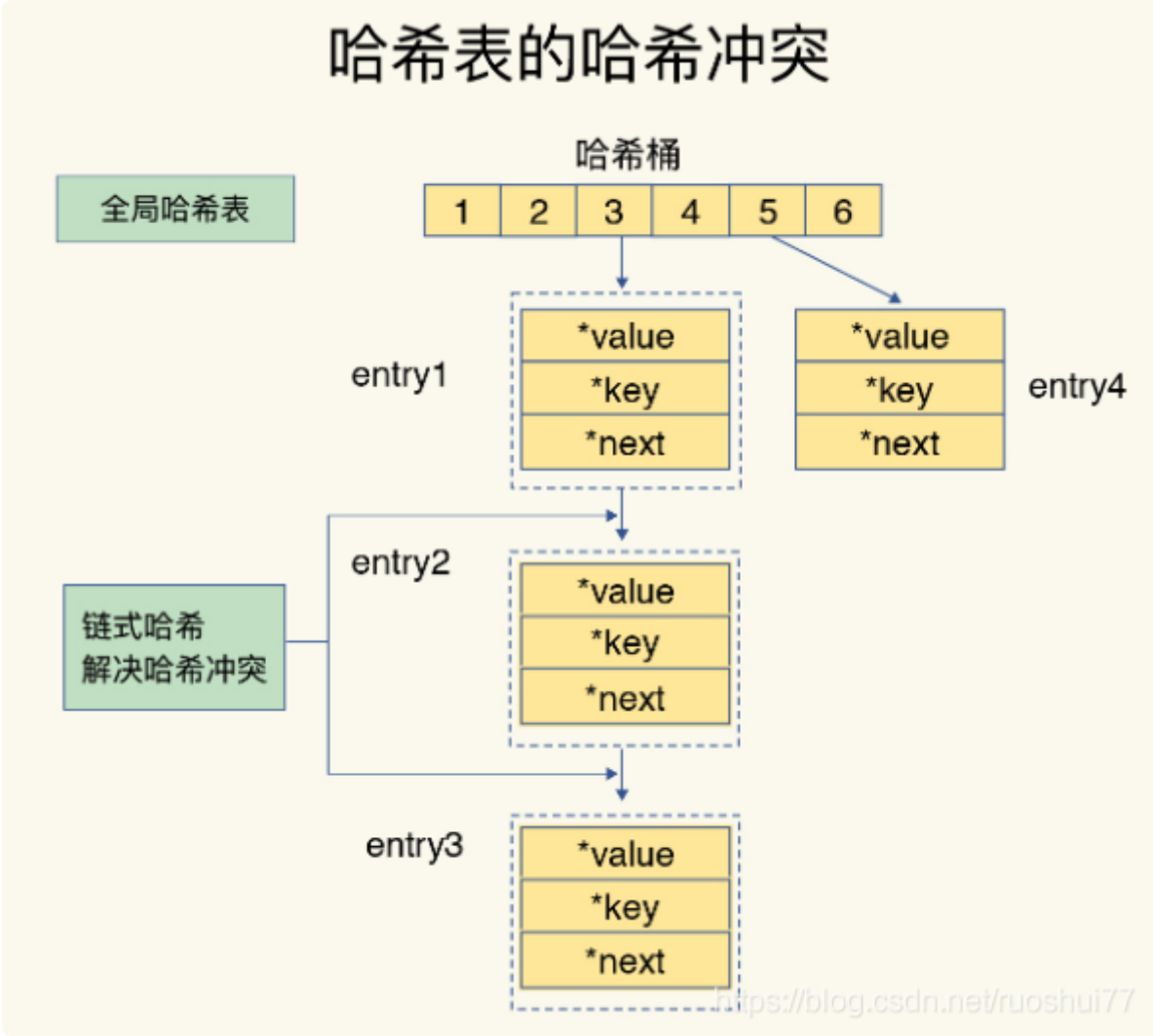
作为一个Java软件工程师，上图列出来了常用到的缓存。ORM框架Hibernate/MyBatis有两级缓存，一级缓存是session级别，二级缓存是sessionFactory级别。著名的Guava库中的Cache也非常好用，可以直接在Java程序中使用，支持设置缓存的数量，缓存失效的时间等特性，非常方便。另外Spring Cache也提供了一套Cache机制，通过注解的方式，开箱即用。Redis/Memcached是使用很多的单独的缓存服务，今天要讲的主角就是Redis。另外内存网格、内存数据库，是一个缓存发展的方向。

3.Redis

Redis全局采用哈希表来存储Key-Value，如下图所示：



Redis 解决哈希冲突的方式，就是链式哈希。链式哈希也很容易理解，就是指同一个哈希桶中的多个元素用一个链表来保存，它们之间依次用指针连接。



3.1数据结构

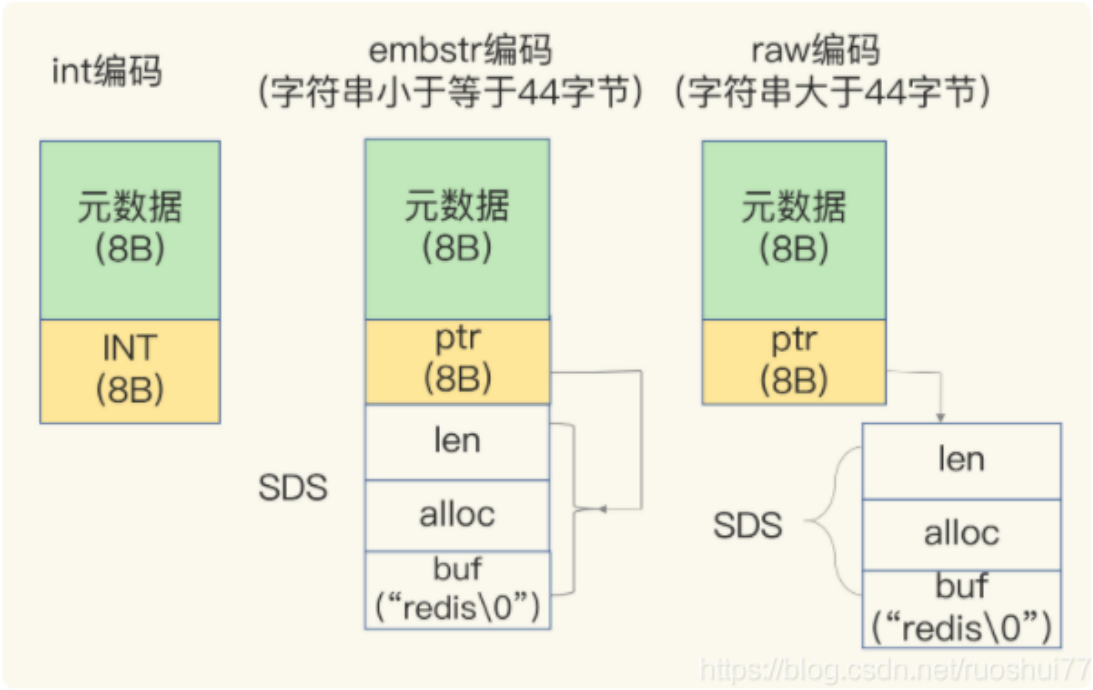
Redis提供了丰富的数据类型，包括5中基本的数据结构，包括：String、Hash、List、Set、Sorted Set，3中高级数据结构，包括：Bitmaps、Hyperloglogs、GEO。

3.1.1 String

String是Redis中最基础的数据存储类型，它在Redis中是二进制安全，这便表示String是万金油类型，可以任何格式的数据，包括：int、String、byte[]。在Redis中字符串类型的value最多可以容纳的数据长度是512M。常用操作命令如下：

Tips:

- 字符串append会使用更多的内存
- 整数共享：如果能够使用整数，就尽量使用整数
- 整数精度问题：redis大概可以保证16左右，17-18位的大整数就会丢失精度
- 字符串底层的数据结构如下：



3.1.2 Hash

Redis中Hash类型相当与Java中的Map，可以看成具有String key 和 String Value的Map容器。所以该类型非常适合于存储对象信息。如Username、 password和age。 如果Hash中包含少量的字段， 那么该类型的数据也将仅占用很少的磁盘空间。 常用操作命令如下：

3.1.1 List

在Redis中， List类型是按照插入顺序排序的字符串链表。 和数据结构中的普通链表 一样， 我们可以在其头部(Left)和尾部(Right)添加新的元素。 在插入时， 如果该键并不存在， Redis将为该键创建一个新的链表。 与此相反， 如果链表中所有的元素均被移除，那么该键也将会被从数据库中删除。 常用操作如下：

3.1.4 Set

在redis中， 可以将Set类型看作是没有排序的字符集合， 和List类型一样， 我们也可以在该类型的数值上执行添加、 删除和判断某一元素是否存在等操作。 这些操作的时间复杂度为O(1),即常量时间内完成依次操作。和List类型不同的是， Set集合中不允许出现重复的元素。常用的操作如下：

3.1.5 Sorted Set

sortedset和set极为相似， 他们都是字符串的集合， 都不允许重复的成员出现在一个set中。 他们之间的主要差别是sortedset中每一个成员都会有一个分数与之关联。 redis正是通过分数来为集合的成员进行从小到大的排序。 sortedset中分数是可以重复的 。常用的操作是：

```
zadd key score member score2 member2... : 将成员以及该成员的分数存放到sortedset中
zscore key member : 返回指定成员的分数
zcard key : 获取集合中成员数量
zrem key member [member...] : 移除集合中指定的成员， 可以指定多个成员
zrange key start end [withscores] : 获取集合中脚注为start-end的成员， [withscores]参数表明返回的成员，包含其分数
zrevrange key start stop [withscores] : 按照分数从大到小的顺序返回索引从start到stop之间的所有元素
（包含两端的元素）
zremrangebyrank key start stop : 按照排名范围删除元素
```

3.1.6 高级数据结构

- Bitmaps bitmaps不是一个真实的数据结构。 而是String类型上的一组面向bit操作的集合。 由于strings是二进制安全的blob， 并且它们的最大长度是512m， 所以bitmaps能最大设置2^32个不同的bit。
- Hyperloglogs 在redis的实现中， 您使用标准错误小于1%的估计度量结束。 这个算法的神奇在于不再
  需要与需要统计的项相对应的内存， 取而代之， 使用的内存一直恒定不变。 最坏的情况下只需要12k， 就可以计算接近2^64个不同元素的基数。
- GEO Redis的GEO特性在 Redis3.2版本中推出， 这个功能可以将用户给定的地理位置（经度和纬度） 信息储存起来， 并对这些信息进行操作。

3.2单线程快的原因

- Redis是纯内存数据库，一般都是简单的存取操作，线程占用的时间很多时间都主要花费在IO上，所以速度很快
- Redis使用的是非阻塞IO，IO多路复用，使用了单线程来轮询描述符，将数据库的开、关、读、写都转换成了事件，减少了线程切换时上下文的切换和竞争
- Redis采用了单线程的模型，保证了每个操作的原子性，也减少了线程的上下文切换和竞争
- Redis全程使用hash结构，读取速度快，还有一些特殊的数据结构，对数据存储进行了优化，如压缩表，对短数据进行压缩存储，再如，跳表，使用有序的数据结构加快读取的速度

- Redis采用自己实现的事件分离器，效率比较高，内部采用非阻塞的执行方式，吞吐能力比较大

3.3 主从复制

将一台redis服务器的数据，复制到其他的redis服务器。前者称为主节点，后者为从节点，数据的复制都是单向，只能从主节点到从节点。Master以写为主，Slave以读为主。主从复制可以数据冗余、故障恢复、负载均衡、高可用的基础。

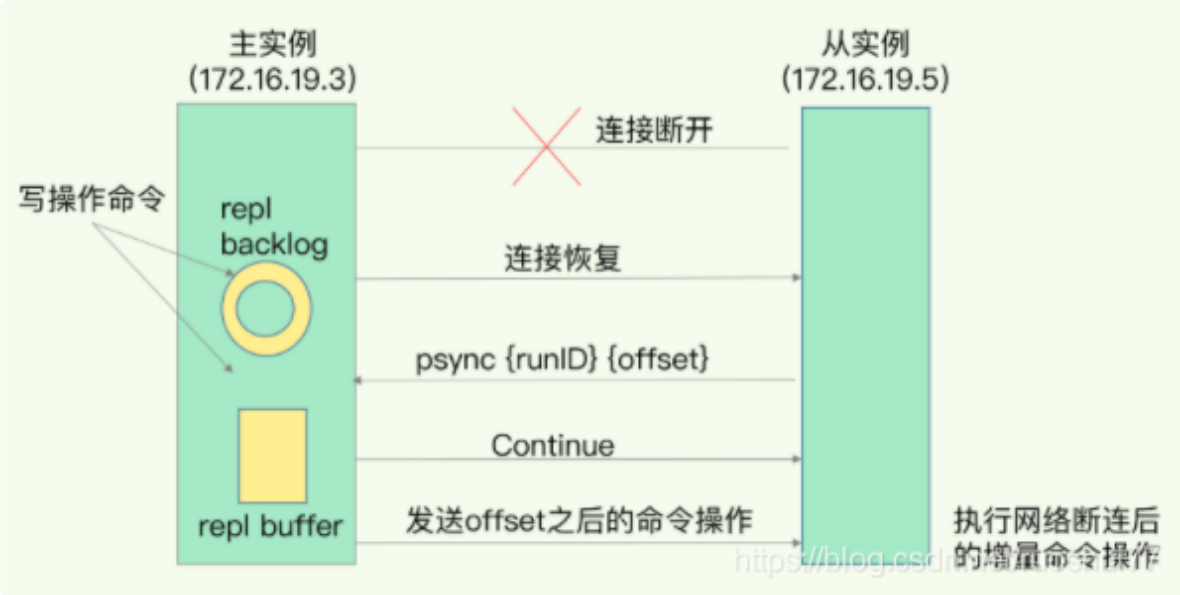
3.3.1 第一次全量同步

- 主从库间建立连接、协商同步的过程，为全量复制做准备。在这一步，从库和主库建立起连接，并告诉主库即将进行同步，主库确认回复后，主从库间就可以开始同步了。具体就是从库给主库发送psync命令，psync命令包括主库的runID和复制进度offset两个参数，主库根据这个命令的参数启动复制。

  `runID`：每个`redis`实例启动时，自动生成一个随机`ID`，用来标识这个实例。当从库和主库第一次复制时，因为不知道主库的`runID`，所以将`runID`设为`"?"`
  `offset`：此时设为`-1`，表示第一次复制

- 主库收到psync命令后，会用fullresync响应命令带上两个参数：主库runID和主库目前的复制进度offset，返回给从库
- 从库收到响应，记录这两个参数
- 主库执行bgsave命令，生成rdb文件，接着讲文件发送给从库
- 从库收到RDB文件，会先清空当前数据库，然后加载rdb文件
- 在主库将数据同步给从库的过程中，主库不会被阻塞，仍然可以接收到正常接收请求。主库将rdb文件生成后，收到的所有写操作记录到replication buffer。
- 主库会把第二阶段执行过程中新收到的写命令，再发送给从库。

3.3.2 断网同步



- repl_backlog_buffer 是一个环形缓冲区，主库会记录自己写到的位置，从库则会记录自己已经读到的位置
- 对主库来说，对应的偏移量就是 master_repl_offset；从库已复制的偏移量
- repl_backlog_buffer 是一个环形缓冲区，所以在缓冲区写满后，主库会继续写入，此时，就会覆盖掉之前写入的操作。如果从库的读取速度比较慢，就有可能导致从库还未读取的操作被主库新写的操作覆盖了，这会导致主从库间的数据不一致
- 缓冲空间的计算公式是：缓冲空间大小 = 主库写入命令速度 * 操作大小 - 主从库间网络传输命令速度 * 操作大小。repl_backlog_size = 缓冲空间大小 * 2

3.3.3 Tips

- 一个redis的实例数据库不要太大，一个实例大小在几个GB级别比较合适，减少RDB文件生成、传输、重新加载的开销
- replication buffer 是主从库在进行全量复制时，主库上用于和从库连接的客户端的 buffer，而 repl_backlog_buffer 是为了支持从库增量复制，主库上用于持续保存写操作的一块专用 buffer

3.4 sentinel

哨兵其实就是一个运行在特殊模式下的 Redis 进程，主从库实例运行的同时，它也在运行。哨兵主要负责的就是三个任务：监控、选主（选择主库）和通知。

3.4.1 监控

- 哨兵进程运行时，周期性给所有的主从库发送PING命令，检测他们是否任然在线运行。
- 若从库没有在规定时间内响应哨兵的PING命令，哨兵就会把它标记为下线状态
- 若主库在规定时间内没有响应哨兵的PING命令，哨兵就会判断主库下线，然后开始自动切换主库的流程

3.4.2 选主

主库挂了之后，哨兵就需要从很多从库里，按照一定的规则悬着一个从库实例，把它作为新的主库。

选主规则如下：

- 当前从库一定在线
- 之前的从库的网络连接状态
- 优先级最高的从库得分高，通过slave-priority配置
- 和旧主库同步成都最接近的从库得分高
- 实例库ID号小的从库得分高

3.4.3 通知

- 哨兵把新主库的连接信息发给其他从库，让他们执行replicaof命令，和新主库建立连接，并进行数据复制。
- 同时哨兵会把新主库的连接信息通知给客户端，让它们把请求操作法到新的主库上

3.4.4 哨兵集群

通常会采用多实例组成的集群模式进行部署，这也被称为哨兵集群。引入多个哨兵实例一起来判断，就可以避免单个哨兵因为自身网络状况不好，而误判主库下线的情况。同时，多个哨兵的网络同时不稳定的概率较小，由它们一起做决策，误判率也能降低。

哨兵集群是基于pub/sub机制组建的。哨兵只要和主库建立起了连接，就可以在主库上发布消息了，比如说发布它自己的连接信息（IP 和端口）。同时，它也可以从主库上订阅消息，获得其他哨兵发布的连接信息。哨兵向主库发送 INFO 命令，获取从库列表。哨兵就可以根据从库列表中的连接信息，和每个从库建立连接，并在这个连接上持续地对从库进行监控。

主从切换过程：

- 任何一个哨兵实例只要自身判断主库“主观下线"后，就给其他哨兵实例发送is-master-down-by-addr命令
- 其他哨兵实例根据自己和主库的连接情况，做出Y或N响应
- 一个哨兵获取了冲裁所需的赞成票数后，quorum后，将主库标记为“客观下线”
- 该哨兵再给其他哨兵发送Leader选举命令
- 若哨兵拿到半数以上的赞成票；并且拿到的票数大于等于哨兵配置文件的quorum值，该哨兵成为执行主从切换的Leader
- 执行主从切换

Tips：

- 要保证所有哨兵实例的配置是一致的，尤其是主观下线的判断值 down-after-milliseconds

3.5 cluster

Redis Cluster 方案采用哈希槽（Hash Slot，接下来我会直接称之为 Slot），来处理数据和实例之间的映射关系。在 Redis Cluster 方案中，一个切片集群共有 16384 个哈希槽，这些哈希槽类似于数据分区，每个键值对都会根据它的 key，被映射到一个哈希槽中。具体的映射过程分为两大步：首先根据键值对的 key，按照CRC16 算法计算一个 16 bit 的值；然后，再用这个 16bit 值对 16384 取模，得到 0~16383 范围内的模数，每个模数代表一个相应编号的哈希槽。

3.5.1 创建集群的方法

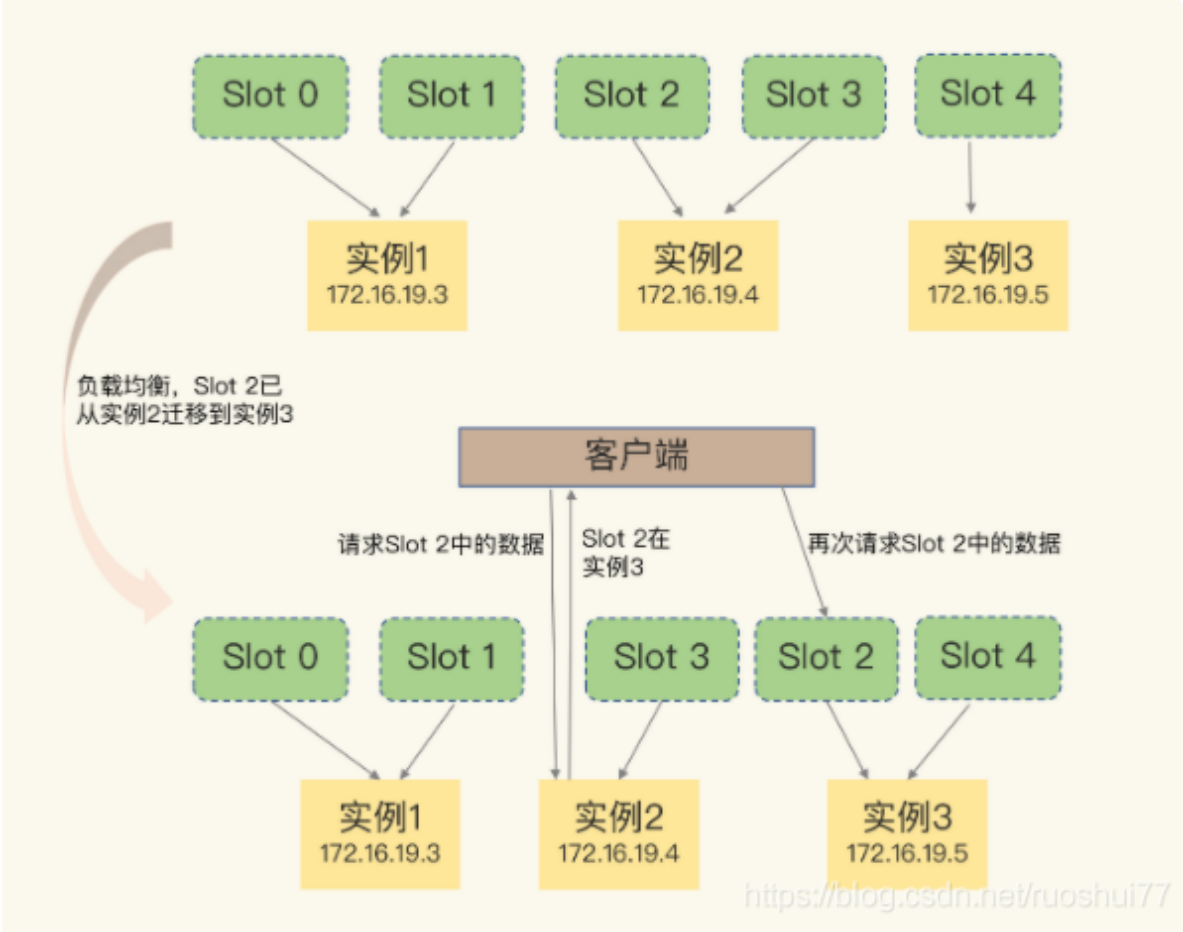
- 在部署 Redis Cluster 方案时，可以使用 cluster create 命令创建集群，此时，Redis 会自动把这些槽平均分布在集群实例上。例如，如果集群中有 N 个实例，那么，每个实例上的槽个数为 16384/N 个。
- 使用 cluster meet 命令手动建立实例间的连接，形成集群，再使用 cluster addslots 命令，指定每个实例上的哈希槽个数。

```
redis-cli -h 172.16.19.3 –p 6379 cluster addslots 0,1
redis-cli -h 172.16.19.4 –p 6379 cluster addslots 2,3
redis-cli -h 172.16.19.5 –p 6379 cluster addslots 4
```

3.5.2 客户端定位数据

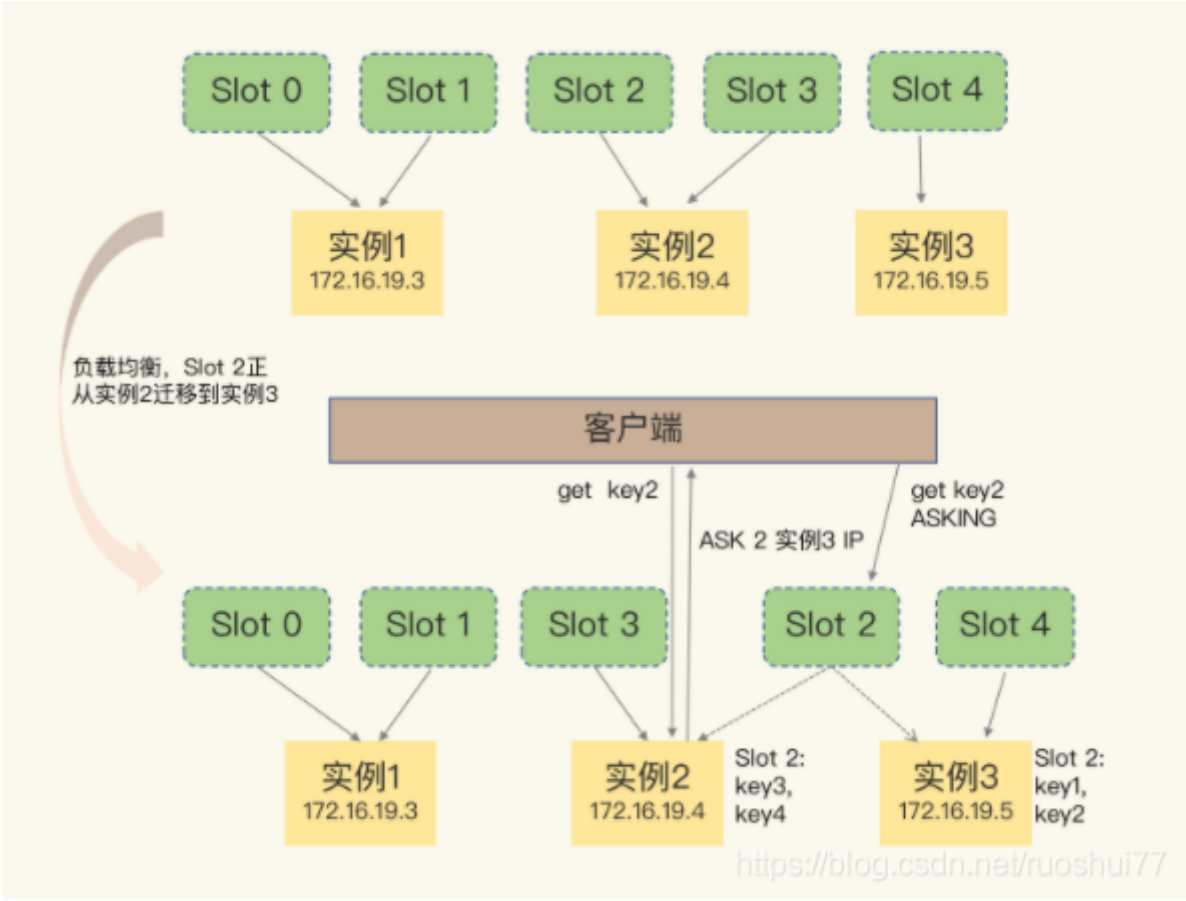
客户端和集群实例建立连接后，实例就会把哈希槽的分配信息发给客户端。客户端收到哈希槽信息后，会把哈希槽信息缓存在本地。当客户端请求键值对时，会先计算键所对应的哈希槽，然后就可以给相应的实例发送请求了。

3.5.2.1 MOVED重定向



由于负载均衡，Slot 2 中的数据已经从实例 2 迁移到了实例 3，但是，客户端缓存仍然记录着“Slot 2 在实例 2”的信息，所以会给实例 2 发送命令。实例 2 给客户端返回一条 MOVED 命令，把 Slot 2 的最新位置（也就是在实例 3 上），返回给客户端，客户端就会再次向实例 3 发送请求，同时还会更新本地缓存，把 Slot 2 与实例的对应关系更新过来。

3.5.2.2 ASK重定向



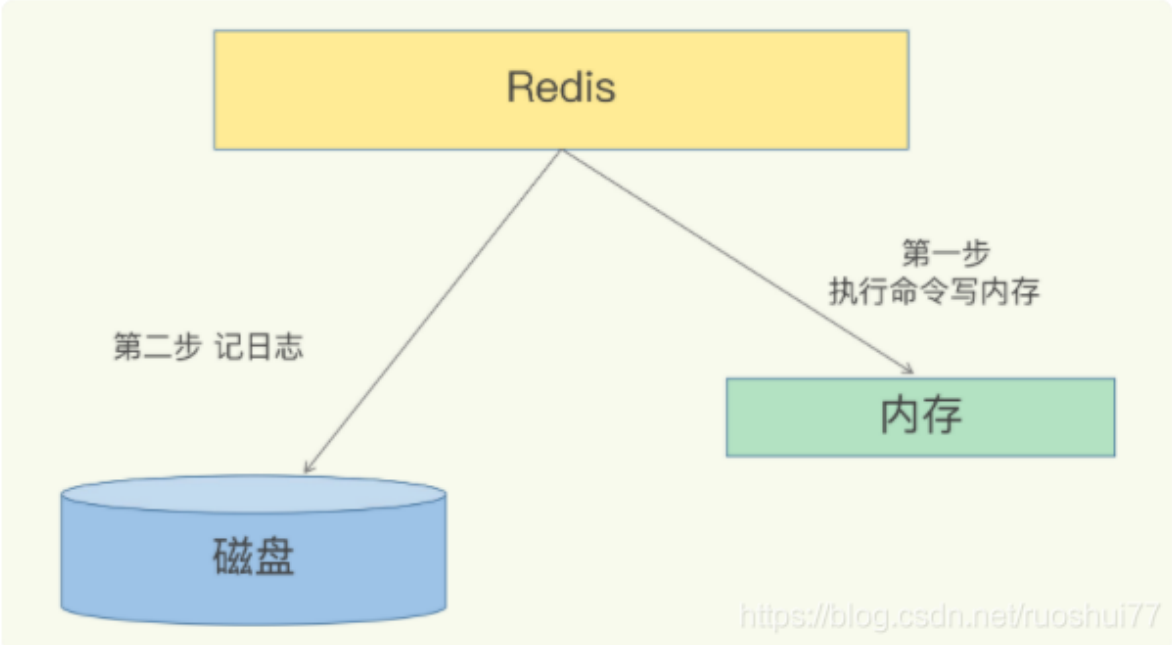
Slot 2 正在从实例 2 往实例 3 迁移，key1 和 key2 已经迁移过去，key3 和 key4 还在实例 2。客户端向实例 2 请求 key2 后，就会收到实例 2 返回的 ASK 命令。

3.5.3 Tips

在手动分配哈希槽时，需要把 16384 个槽都分配完，否则 Redis 集群无法正常工作。

3.6 持久化

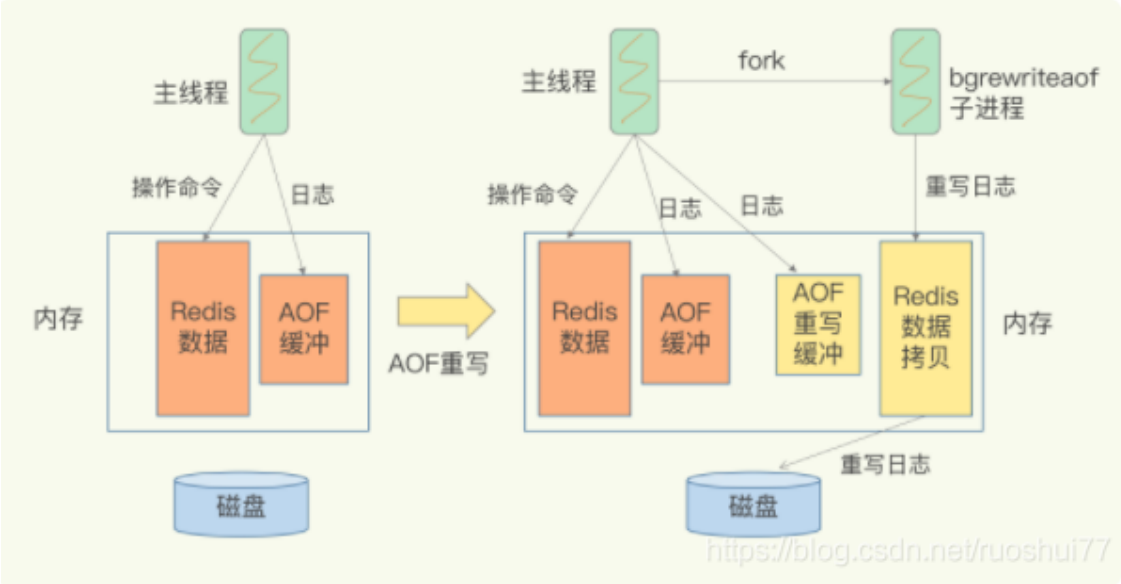
3.6.1 AOF



AOF 机制给我们提供了三个选择，也就是 AOF 配置项 appendfsync 的三个可选值：

- Always，同步写回：每个写命令执行完，立马同步地将日志写回磁盘；

- Everysec，每秒写回：每个写命令执行完，只是先把日志写到 AOF 文件的内存缓冲区，每隔一秒把缓冲区中的内容写入磁盘；
- No，操作系统控制的写回：每个写命令执行完，只是先把日志写到 AOF 文件的内存缓冲区，由操作系统决定何时将缓冲区内容写回磁盘。

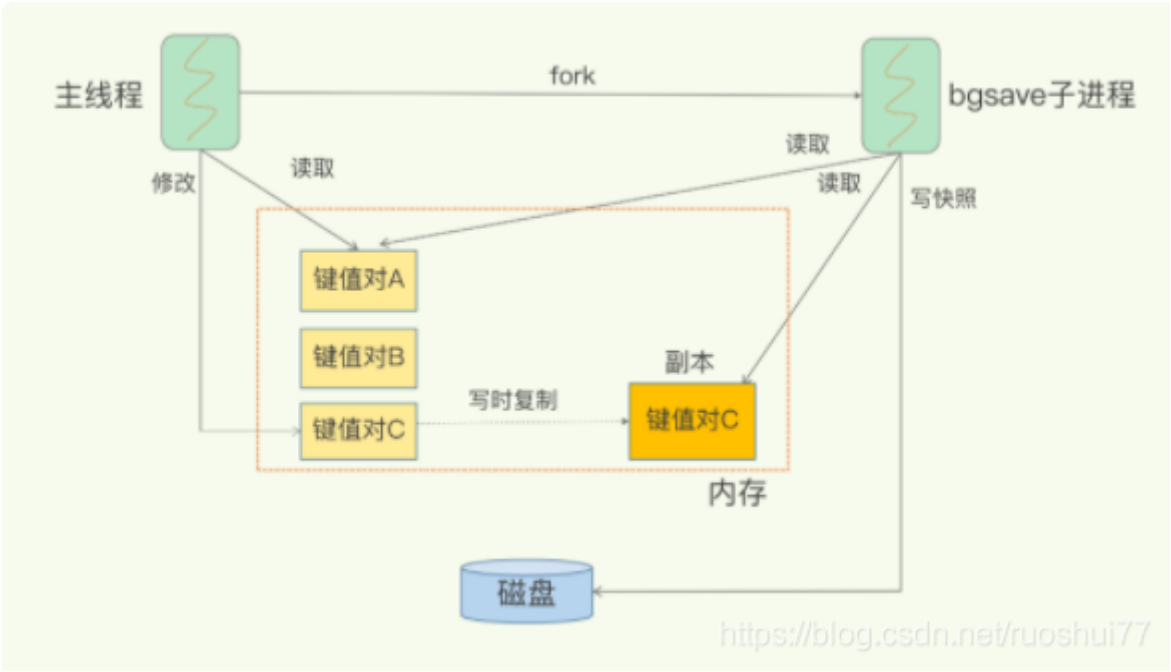


和 AOF 日志由主线程写回不同，重写过程是由后台子进程 bgrewriteaof 来完成的，这也是为了避免阻塞主线程，导致数据库性能下降。

每次执行重写时，主线程 fork 出后台的 bgrewriteaof 子进程。此时，fork 会把主线程的内存拷贝一份给 bgrewriteaof 子进程，这里面就包含了数据库的最新数据。然后，bgrewriteaof 子进程就可以在不影响主线程的情况下，逐一把拷贝的数据写成操作，记入重写日志。

因为主线程未阻塞，仍然可以处理新来的操作。此时，如果有写操作，第一处日志就是指正在使用的 AOF 日志，Redis 会把这个操作写到它的缓冲区。这样一来，即使宕机了，这个 AOF 日志的操作仍然是齐全的，可以用于恢复。而第二处日志，就是指新的 AOF 重写日志。这个操作也会被写到重写日志的缓冲区。这样，重写日志也不会丢失最新的操作。等到拷贝数据的所有操作记录重写完成后，重写日志记录的这些最新操作也会写入新的 AOF 文件，以保证数据库最新状态的记录。此时，我们就可以用新的 AOF 文件替代旧文件了。

3.6.2 RDB

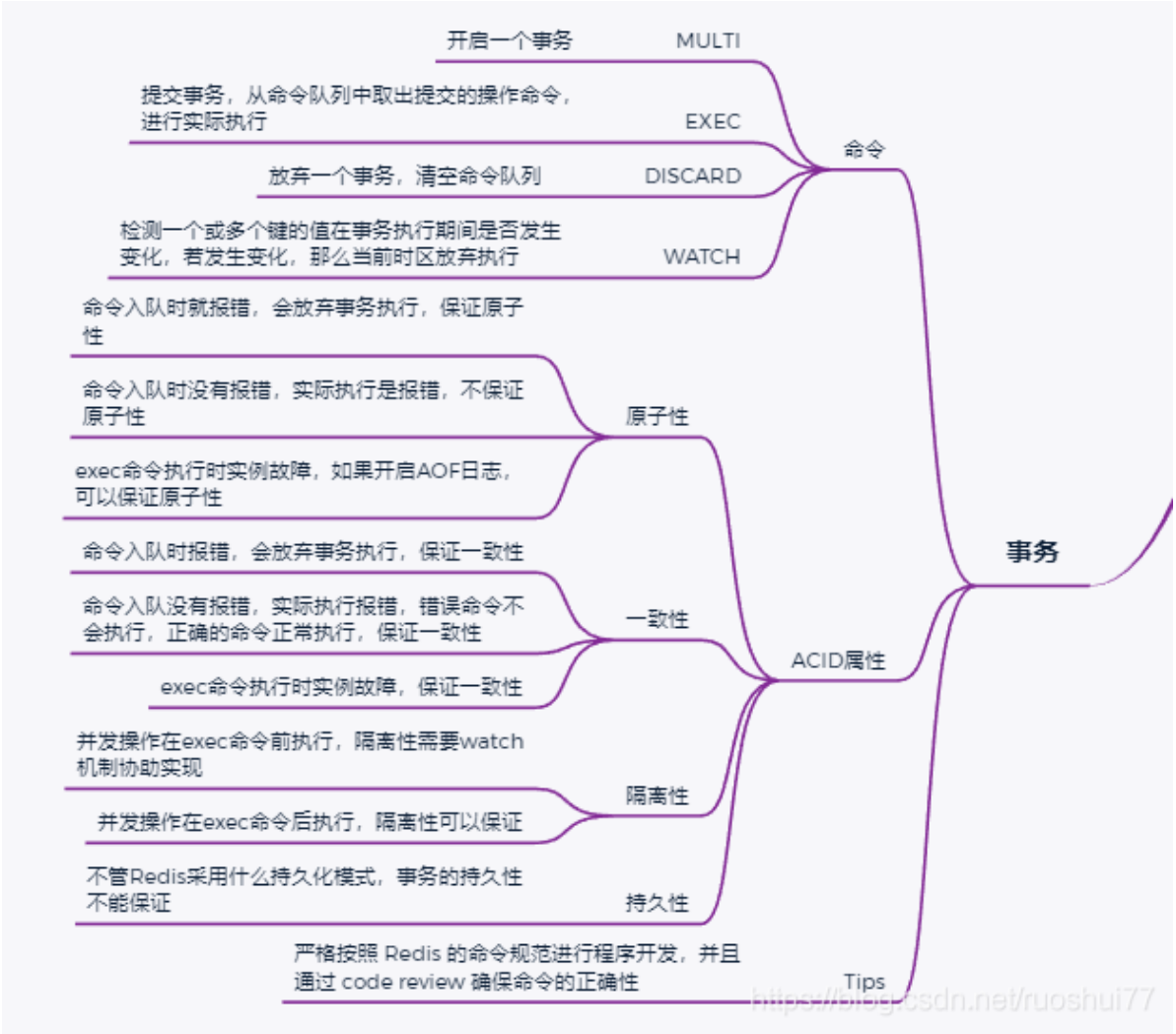


如果主线程对这些数据也都是读操作（例如图中的键值对 A），那么，主线程和 bgsave 子进程相互不影响。但是，如果主线程要修改一块数据（例如图中的键值对 C），那么，这块数据就会被复制一份，生成该数据的副本。然后，bgsave 子进程会把这个副本数据写入 RDB 文件，而在这个过程中，主线程仍然可以直接修改原来的数据。

3.7 事务

Redis 事务的本质：一组命令的集合！一个事务中的所有命令都会被序列化，在事务执行过程中，会按照顺序执行。Redis 事务没有隔离级别的概念。Redis 单条命令是保证原子性的，但是事务不保证原子性。

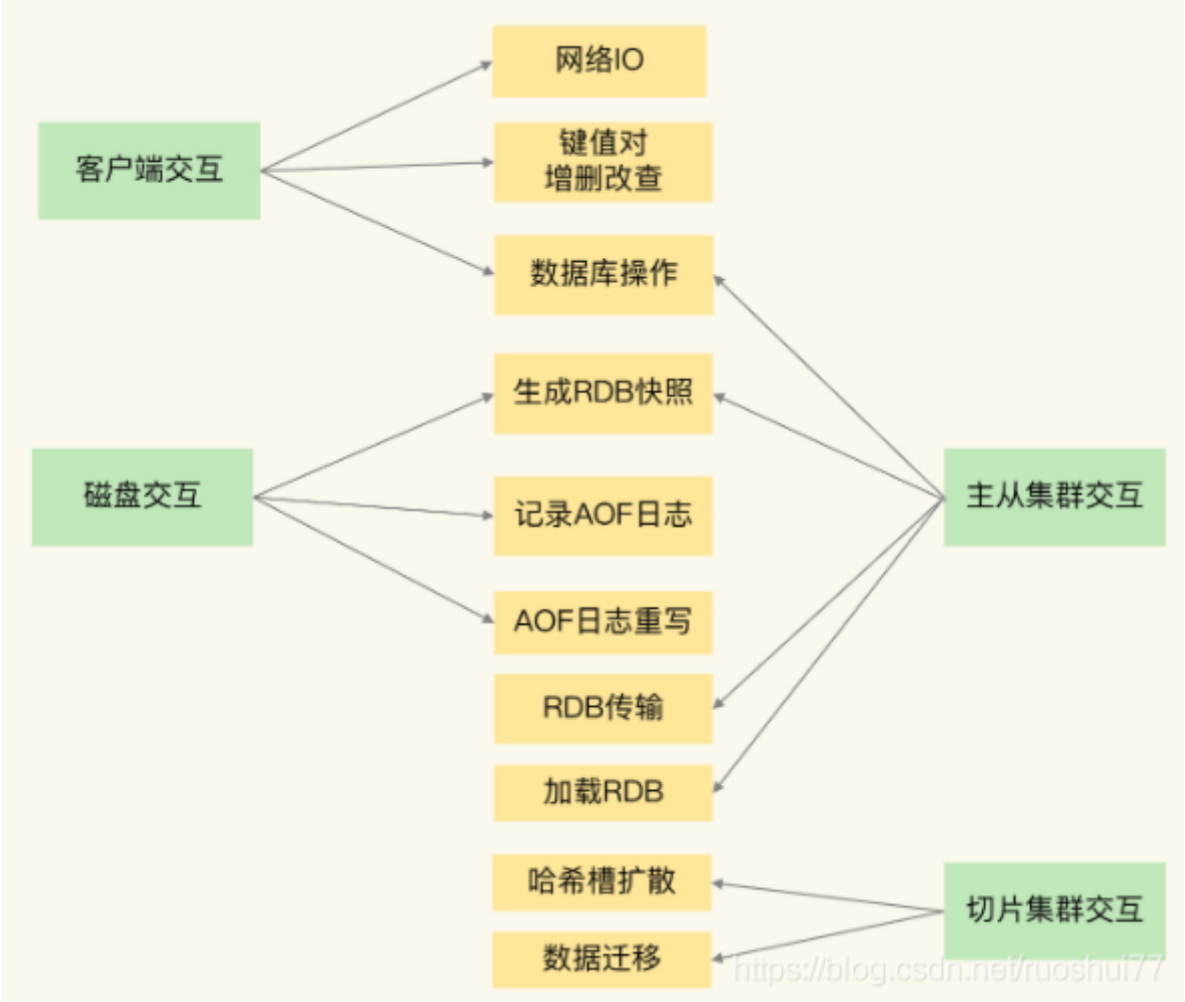
```
#开启事务
127.0.0.1:6379> MULTIOK
#将a:stock减1，
127.0.0.1:6379> DECR a:stock
QUEUED
#将b:stock减1
127.0.0.1:6379> DECR b:stock
QUEUED
#实际执行事务
127.0.0.1:6379> EXEC
1) (integer) 4
2) (integer) 9
```



3.8 使用业务场景

3.9 慢可能的原因和排查方法

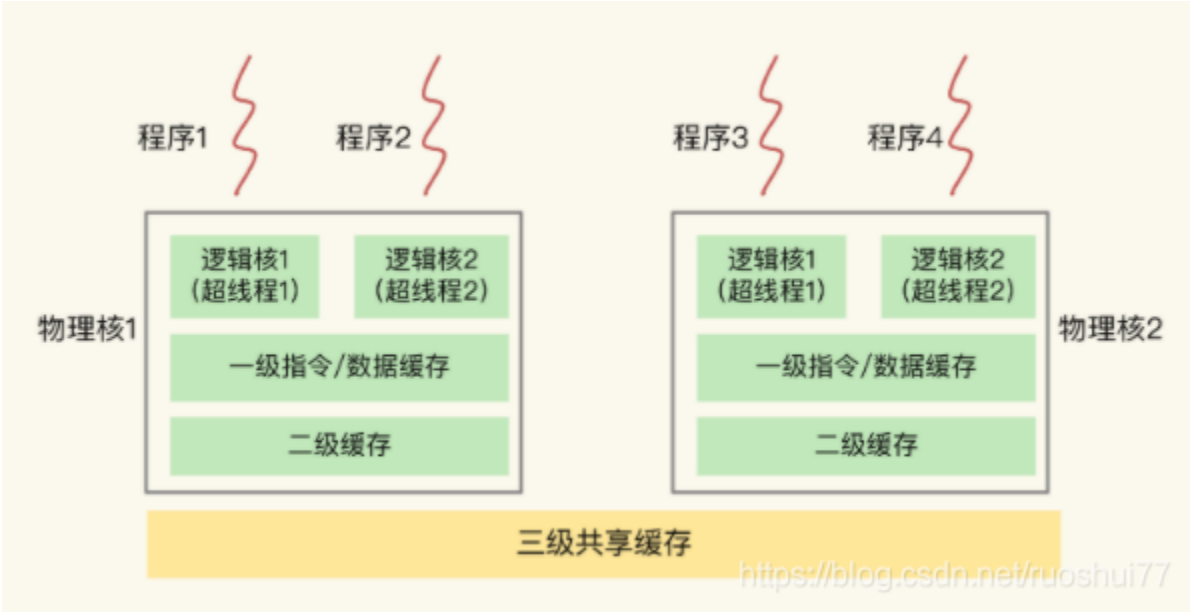
3.9.1 Redis内部阻塞操作



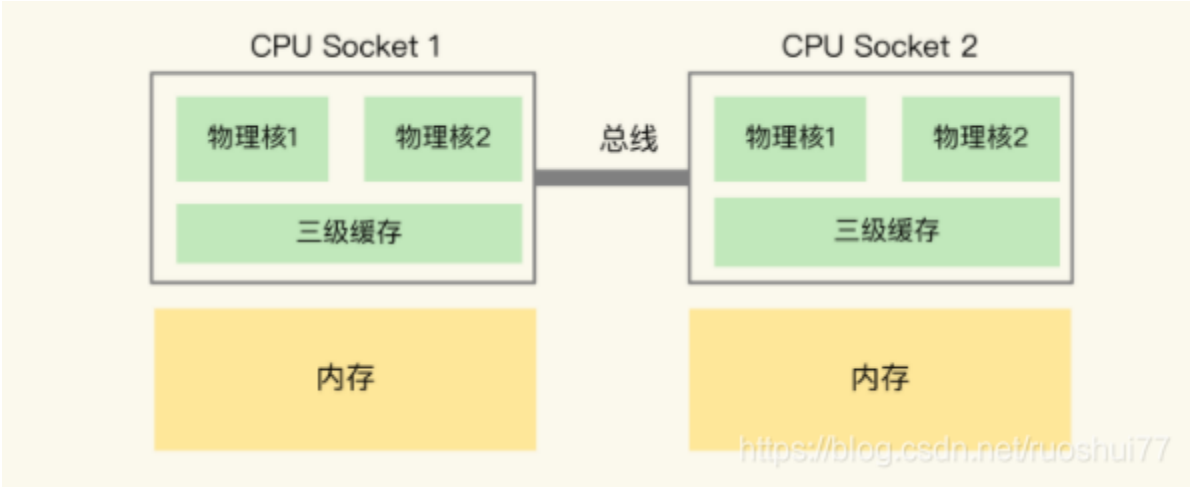
Tips:

- 集合全量查询和聚合操作：可以使用SCAN命令，分批读取数据，再在客户端进行聚合计算
- 从库加载RDB文件：主库把数据量大小控制在2-4GB左右，保证RDB文件能以较快的速度加载

3.9.2 CPU核核核NUMA架构的影响



在多核 CPU 的场景下，一旦应用程序需要在一个新的 CPU 核上运行，那么，运行时信息就需要重新加载到新的 CPU 核上。而且，新的 CPU 核的 L1、L2 缓存也需要重新加载数据和指令，这会导致程序的运行时间增加。



在多 CPU 架构下，一个应用程序访问所在 Socket 的本地内存和访问远端内存的延迟并不一致。

在 CPU 的 NUMA 架构下，对 CPU 核的编号规则，并不是先把一个 CPU Socket 中的所有逻辑核编完，再对下一个 CPU Socket 中的逻辑核编码，而是先给每个 CPU Socket 中每个物理核的第一个逻辑核依次编号，再给每个 CPU Socket 中的物理核的第二个逻辑核依次编号。

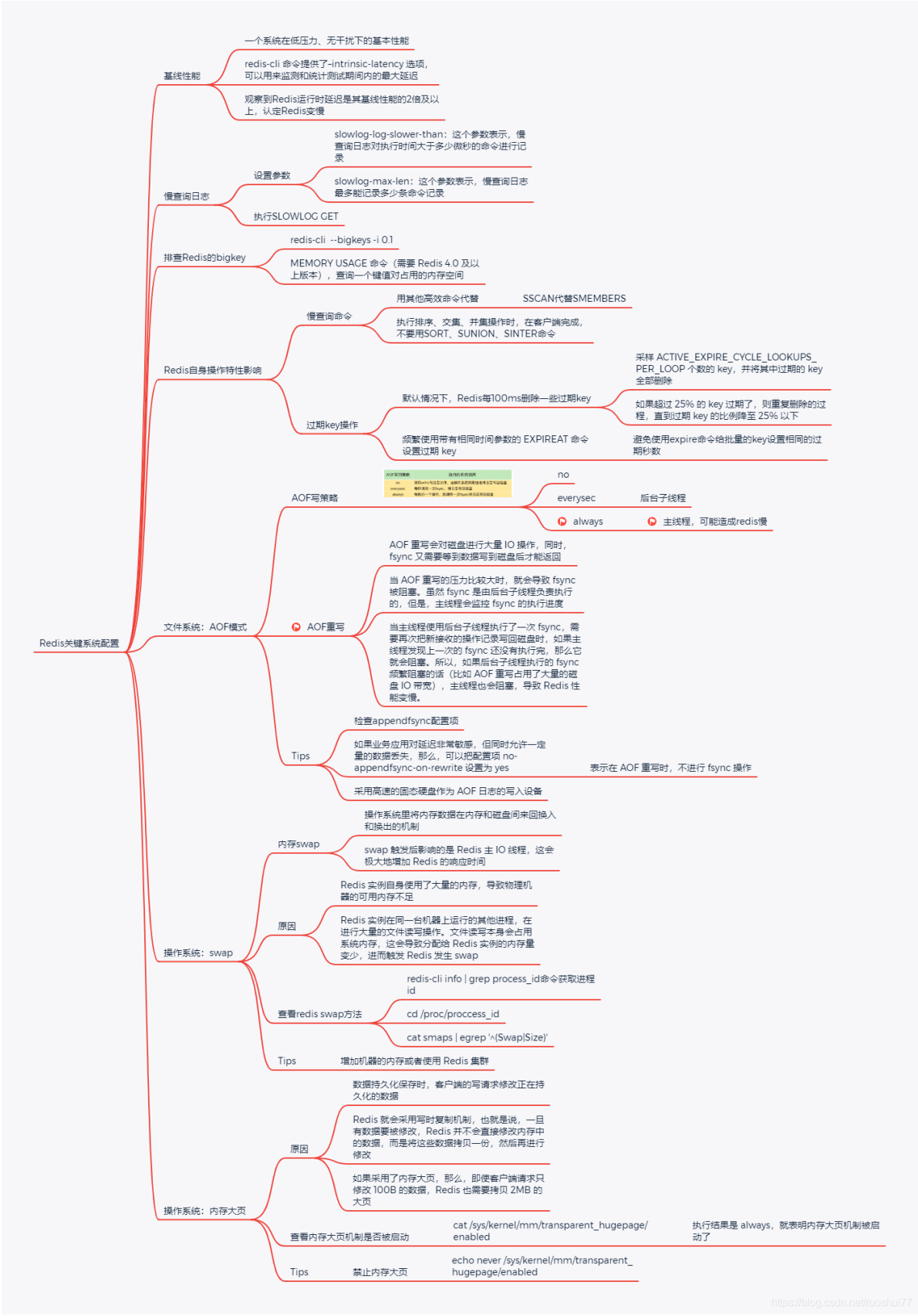
Tips:

- 多核CPU，使用taskset命令把一个程序绑定在一个物理核上运行
- NUMA架构，把网络中断程序和Redis实例绑在同一个CPU Socket上

3.9.3 Redis内存碎片

3.9.4 Redis缓冲区

3.9.6 Redis关键系统配置



3.9.7 Redis变慢CheckList

- 使用复杂度过高的命令或一次查询全量数据；
- 操作 bigkey；
- 大量 key 集中过期；
- 内存达到 maxmemory；
- 客户端使用短连接和 Redis 相连；
- 当 Redis 实例的数据量大时，无论是生成 RDB，还是 AOF 重写，都会导致 fork 耗时严重；
- AOF 的写回策略为 always，导致每个操作都要同步刷回磁盘；

- Redis 实例运行机器的内存不足，导致 swap 发生，Redis 需要到 swap 分区读取数据；
- 进程绑定 CPU 不合理；
- Redis 实例运行机器上开启了透明内存大页机制；
- 网卡压力过大

3.10 常见问题

3.10.1 缓存穿透

原因： 大量并发查询不存在的KEY， 导致都直接将压力透传到数据库

解决办法：

- 缓存空值的KEY， 这样第一次不存在也会被加载会记录， 下次拿到有这个KEY。
- 完全以缓存为准， 使用 延迟异步加载 的策略， 这样就不会触发更新
- Bloom过滤或RoaringBitmap 判断KEY是否存在

3.10.2 缓存击穿

原因： 某个KEY失效的时候， 正好有大量并发请求访问这个KEY

解决办法：

- KEY的更新操作添加全局互斥锁
- 完全以缓存为准， 使用 延迟异步加载 的策略， 这样就不会触

3.10.3 缓存雪崩

原因： 当某一时刻发生大规模的缓存失效的情况， 会有大量的请求进来直接打到数据库， 导致数据库压力过大升值宕机

解决办法：

- 更新策略在时间上做到比较均匀
- 使用的热数据尽量分散到不同的机器上
- 多台机器做主从复制或者多副本， 实现高可用
- 实现熔断限流机制， 对系统进行负载能力控制

3.11 使用建议

- 线上禁止部分命令，包括：KEYS、FLUSHALL、FLUSHDB；具体做法是管理员用rename-command命令在配置文件对这些命令进行重命名，让客户端无法使用这些命令
- 使用业务名做key的前缀，并使用缩写形式
- 控制key的长度
- 使用高效序列化方法和压缩方法
- 使用整数对象共享池
- 不同的业务数据保存到不同实例
- 数据保存时设置过期时间
- 慎用MONITOR命令
- 慎用全量操作命令
- 控制String类型操作的大小不超过10KB
- 控制集合类型的元素不超过10000个
- 使用Redis保存热数据
  上做到比较均匀
- 使用的热数据尽量分散到不同的机器上
- 多台机器做主从复制或者多副本， 实现高可用

- 实现熔断限流机制， 对系统进行负载能力控制

3.11 使用建议

- 线上禁止部分命令，包括：KEYS、FLUSHALL、FLUSHDB；具体做法是管理员用rename-command命令在配置文件对这些命令进行重命名，让客户端无法使用这些命令
- 使用业务名做key的前缀，并使用缩写形式
- 控制key的长度
- 使用高效序列化方法和压缩方法
- 使用整数对象共享池
- 不同的业务数据保存到不同实例
- 数据保存时设置过期时间
- 慎用MONITOR命令
- 慎用全量操作命令
- 控制String类型操作的大小不超过10KB
- 控制集合类型的元素不超过10000个
- 使用Redis保存热数据
- 把Redis实例的容量控制在2-6GB