阿里巴巴Redis使用规范

规范一： 控制 key 的长度

规范二：避免使用 bigkey

规范三：使用高效序列化方法和压缩方法

规范四：使用整数对象共享池

规范五：使用 Redis 保存热数据

规范六：不同的业务数据分实例存储

规范七：在数据保存时，要设置过期时间

规范八：控制 Redis 实例的容量

规范九：线上禁用部分命令

规范十：慎用 MONITOR 命令

规范十一：慎用全量操作命令

规范十二：选择合适的数据类型

规范十三：开启 lazy-free 机制

规范十四：不使用复杂度过高的命令

规范十五：执行 O(N) 命令时，关注 N 的大小

规范十六：关注 DEL 时间复杂度

规范十七：批量命令代替单个命令

规范十八：避免集中过期 key

规范十九：使用长连接操作 Redis，合理配置连接池

规范二十：只使用 db0

规范二十一：使用物理机部署 Redis

规范二十二：关闭操作系统内存大页机制

规范二十三：合理配置主从复制参数

规范二十四：扫描线上实例时，设置休眠时间

规范二十五： 从库必须设置为 slave-read-only

规范二十六：合理配置 timeout 和 tcp-keepalive 参数

规范二十七：调整 maxmemory 时，注意主从库的调整顺序

规范二十八：Redis 安全保证

规范类别	规范内容
强制	禁用KEYS、FLUSHALL、FLUSHDB命令
推荐	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
建议	控制String类型数据的大小不超过10KB
	控制集合类型的元素个数不超过1万个
	使用Redis保存热数据
	把Redis实例的容量控制在2~6GB

规范一： 控制 key 的长度

当 key 字符串的长度增加时，SDS 中的元数据也会占用更多内存空间。我们在设置 key 的名称时，要注意控制 key 的长度。

规范二：避免使用 bigkey

我们要尽量把 String 类型的数据大小控制在 10KB 以下。
尽量把集合类型的元素个数控制在 1 万以下。
假设 Hash 集合的 hash-max-ziplist-entries 配置项是 1000，如果 Hash 集合元素个数不超过 1000，就会使用 ziplist 保存数据。

紧凑型数据结构虽然可以节省内存，但是会在一定程度上导致数据的读写性能下降。所以，如果业务应用更加需要保持高性能访问，而不是节省内存的话，在不会导致 bigkey 的前提下，你就不用刻意控制集合元素个数了。

规范三：使用高效序列化方法和压缩方法

为了节省内存，除了采用紧凑型数据结构以外，我们还可以遵循两个使用规范，分别是使用高效的序列化方法和压缩方法，这样可以减少 value 的大小。

Redis 中的字符串都是使用二进制安全的字节数组来保存的，所以，我们可以把业务数据序列化成二进制数据写入到 Redis 中。

但是，不同的序列化方法，在序列化速度和数据序列化后的占用内存空间这两个方面，效果是不一样的。比如说，protostuff 和 kryo 这两种序列化方法，就要比 Java 内置的序列化方法（java-build-in-serializer）效率更高。

此外，业务应用有时会使用字符串形式的 XML 和 JSON 格式保存数据。

这样做的好处是，这两种格式的可读性好，便于调试，不同的开发语言都支持这两种格式的解析。

缺点在于，XML 和 JSON 格式的数据占用的内存空间比较大。为了避免数据占用过大的内存空间，我建议使用压缩工具（例如 snappy 或 gzip），把数据压缩后再写入 Redis，这样就可以节省内存空间了。

规范四：使用整数对象共享池

整数是常用的数据类型，Redis 内部维护了 0 到 9999 这 1 万个整数对象，并把这些整数作为一个共享池使用。
换句话说，如果一个键值对中有 0 到 9999 范围的整数，Redis 就不会为这个键值对专门创建整数对象了，而是会复用共享池中的整数对象。
这样一来，即使大量键值对保存了 0 到 9999 范围内的整数，在 Redis 实例中，其实只保存了一份整数对象，可以节省内存空间。

基于这个特点，我建议你，在满足业务数据需求的前提下，能用整数时就尽量用整数，这样可以节省实例内存。

那什么时候不能用整数对象共享池呢？主要有两种情况。

第一种情况是，如果 Redis 中设置了 maxmemory，而且启用了 LRU 策略（allkeys-lru 或 volatile-lru 策略），那么，整数对象共享池就无法使用了。这是因为，LRU 策略需要统计每个键值对的使用时间，如果不同的键值对都共享使用一个整数对象，LRU 策略就无法进行统计了。

第二种情况是，如果集合类型数据采用 ziplist 编码，而集合元素是整数，这个时候，也不能使用共享池。因为 ziplist 使用了紧凑型内存结构，判断整数对象的共享情况效率低。

规范五：使用 Redis 保存热数据

为了提供高性能访问，Redis 是把所有数据保存到内存中的。

虽然 Redis 支持使用 RDB 快照和 AOF 日志持久化保存数据，但是，这两个机制都是用来提供数据可靠性保证的，并不是用来扩充数据容量的。而且，内存成本本身就比较高，如果把业务数据都保存在 Redis 中，会带来较大的内存成本压力。

一般来说，在实际应用 Redis 时，我们会更多地把它作为缓存保存热数据，这样既可以充分利用 Redis 的高性能特性，还可以把宝贵的内存资源用在服务热数据上，就是俗话说的“好钢用在刀刃上”。

规范六：不同的业务数据分实例存储

虽然我们可以使用 key 的前缀把不同业务的数据区分开，但是，如果所有业务的数据量都很大，而且访问特征也不一样，我们把这些数据保存在同一个实例上时，这些数据的操作就会相互干扰。

你可以想象这样一个场景：假如数据采集业务使用 Redis 保存数据时，以写操作为主，而用户统计业务使用 Redis 时，是以读查询为主，如果这两个业务数据混在一起保存，读写操作相互干扰，肯定会导致业务响应变慢。

那么，我建议你把不同的业务数据放到不同的 Redis 实例中。这样一来，既可以避免单实例的内存使用量过大，也可以避免不同业务的操作相互干扰。

规范七：在数据保存时，要设置过期时间

对于 Redis 来说，内存是非常宝贵的资源，而且，Redis 通常用于保存热数据。热数据一般都有使用的时效性。

所以，在数据保存时，我建议你根据业务使用数据的时长，设置数据的过期时间。不然的话，写入 Redis 的数据会一直占用内存，如果数据持续增多，就可能达到机器的内存上限，造成内存溢出，导致服务崩溃。

规范八：控制 Redis 实例的容量

Redis 单实例的内存大小都不要太大，根据我自己的经验值，建议你设置在 2~6GB 。这样一来，无论是 RDB 快照，还是主从集群进行数据同步，都能很快完成，不会阻塞正常请求的处理。

规范九：线上禁用部分命令

Redis 是单线程处理请求操作，如果我们执行一些涉及大量操作、耗时长的命令，就会严重阻塞主线程，导致其它请求无法得到正常处理，这类命令主要有 3 种。

KEYS，按照键值对的 key 内容进行匹配，返回符合匹配条件的键值对，该命令需要对 Redis 的全局哈希表进行全表扫描，严重阻塞 Redis 主线程；

FLUSHALL，删除 Redis 实例上的所有数据，如果数据量很大，会严重阻塞 Redis 主线程；

FLUSHDB，删除当前数据库中的数据，如果数据量很大，同样会阻塞 Redis 主线程。

所以，我们在线上应用 Redis 时，就需要禁用这些命令。具体的做法是，管理员用 rename-command 命令在配置文件中对这些命令进行重命名，让客户端无法使用这些命令。

对于 KEYS 命令来说，你可以用 SCAN 命令代替 KEYS 命令，分批返回符合条件的键值对，避免造成主线程阻塞；

对于 FLUSHALL、FLUSHDB 命令来说，你可以加上 ASYNC 选项，让这两个命令使用后台线程异步删除数据，可以避免阻塞主线程。

规范十：慎用 MONITOR 命令

Redis 的 MONITOR 命令在执行后，会持续输出监测到的各个命令操作，所以，我们通常会用 MONITOR 命令返回的结果，检查命令的执行情况。

但是，MONITOR 命令会把监控到的内容持续写入输出缓冲区。如果线上命令的操作很多，输出缓冲区很快就会溢出了，这就会对 Redis 性能造成影响，甚至引起服务崩溃。

所以，除非十分需要监测某些命令的执行（例如，Redis 性能突然变慢，我们想查看下客户端执行了哪些命令），你可以偶尔在短时间内使用下 MONITOR 命令，否则，我建议你不要使用 MONITOR 命令。

规范十一：慎用全量操作命令

对于集合类型的数据来说，如果想要获得集合中的所有元素，一般不建议使用全量操作的命令（例如 Hash 类型的 HGETALL、Set 类型的 SMEMBERS）。这些操作会对 Hash 和 Set 类型的底层数据结构进行全量扫描，如果集合类型数据较多的话，就会阻塞 Redis 主线程。

如果想要获得集合类型的全量数据，我给你三个小建议。

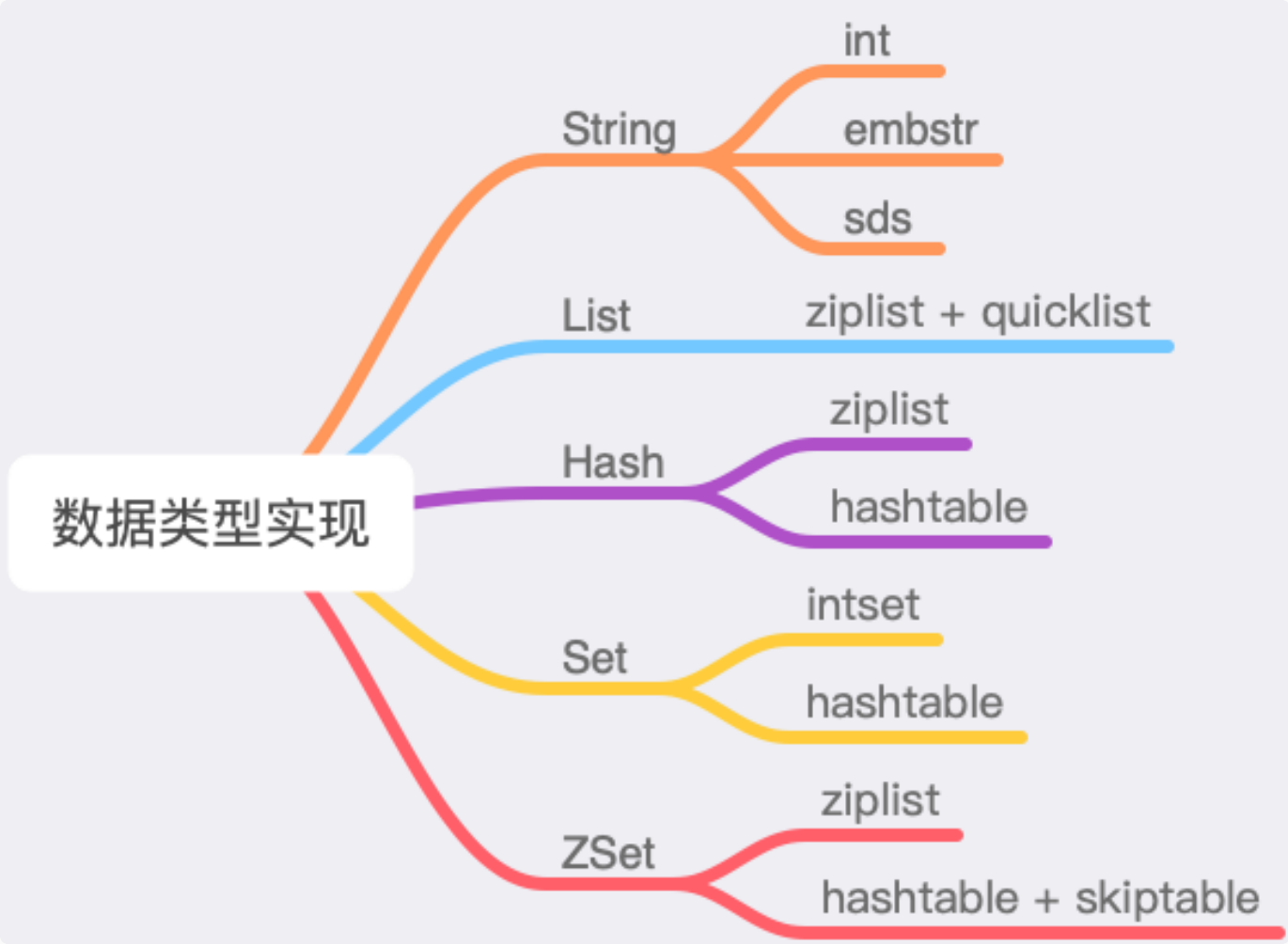
第一个建议是，你可以使用 SSCAN、HSCAN 命令分批返回集合中的数据，减少对主线程的阻塞。

第二个建议是，你可以化整为零，把一个大的 Hash 集合拆分成多个小的 Hash 集合。这个操作对应到业务层，就是对业务数据进行拆分，按照时间、地域、用户 ID 等属性把一个大集合的业务数据拆分成多个小集合数据。例如，当你统计用户的访问情况时，就可以按照天的粒度，把每天的数据作为一个 Hash 集合。

最后一个建议是，如果集合类型保存的是业务数据的多个属性，而每次查询时，也需要返回这些属性，那么，你可以使用 String 类型，将这些属性序列化后保存，每次直接返回 String 数据就行，不用再对集合类型做全量扫描了。

规范十二：选择合适的数据类型

Redis 提供了丰富的数据类型，这些数据类型在实现上，也对内存使用做了优化。具体来说就是，一种数据类型对应多种数据结构来实现：



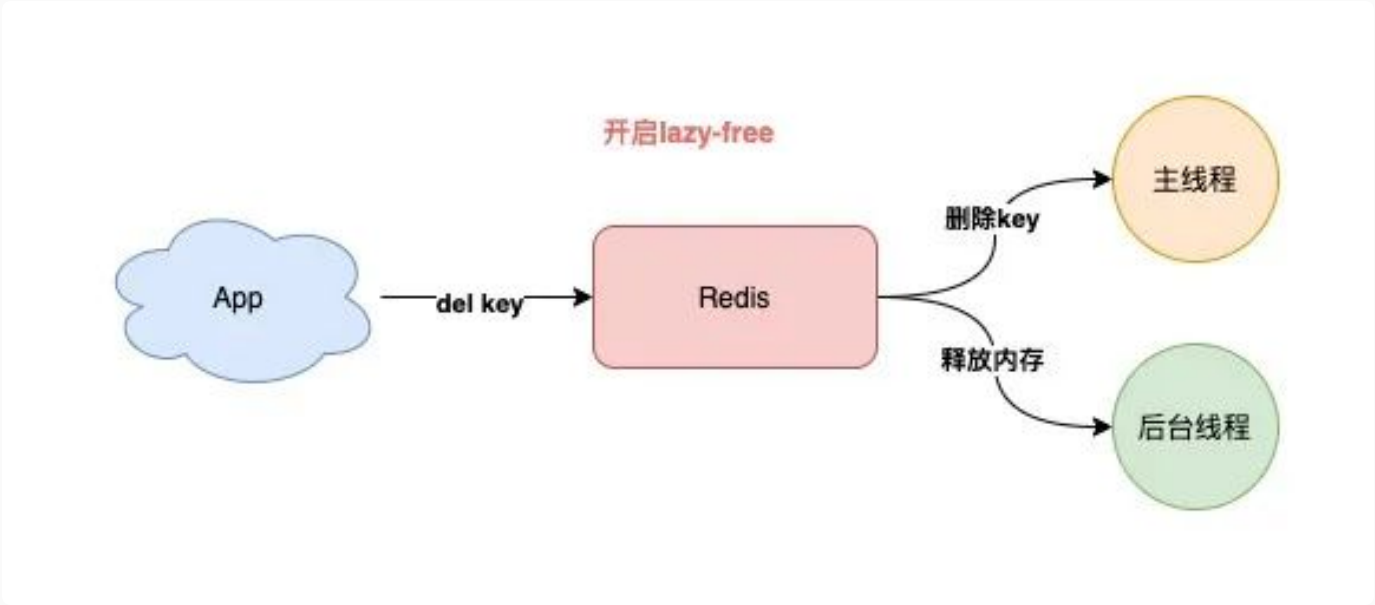
例如，String、Set 在存储 int 数据时，会采用整数编码存储。Hash、ZSet 在元素数量比较少时（可配置），会采用压缩列表（ziplist）存储，在存储比较多的数据时，才会转换为哈希表和跳表。
作者这么设计的原因，就是为了进一步节约内存资源。
那么你在存储数据时，就可以利用这些特性来优化 Redis 的内存。这里我给你的建议如下：

- String、Set：尽可能存储 int 类型数据
- Hash、ZSet：存储的元素数量控制在转换阈值之下，以压缩列表存储，节约内存

规范十三：开启 lazy-free 机制

如果你无法避免存储 bigkey，那么我建议你开启 Redis 的 lazy-free 机制。（4.0+版本支持）

当开启这个机制后，Redis 在删除一个 bigkey 时，释放内存的耗时操作，将会放到后台线程中去执行，这样可以在最大程度上，避免对主线程的影响。



规范十四：不使用复杂度过高的命令

Redis 是单线程模型处理请求，除了操作 bigkey 会导致后面请求发生排队之外，在执行复杂度过高的命令时，也会发生这种情况。

因为执行复杂度过高的命令，会消耗更多的 CPU 资源，主线程中的其它请求只能等待，这时也会发生排队延迟。

所以，你需要避免执行例如 SORT、SINTER、SINTERSTORE、ZUNIONSTORE、ZINTERSTORE 等聚合类命令。

对于这种聚合类操作，我建议你把它放到客户端来执行，不要让 Redis 承担太多的计算工作。

规范十五：执行 O(N) 命令时，关注 N 的大小

当你在执行 O(N) 命令时，同样需要注意 N 的大小。
如果一次性查询过多的数据，也会在网络传输过程中耗时过长，操作延迟变大。
所以，对于容器类型（List/Hash/Set/ZSet），在元素数量未知的情况下，一定不要无脑执行 LRANGE key 0 –1 / HGETALL / SMEMBERS / ZRANGE key 0 –1。

在查询数据时，你要遵循以下原则：

1. 先查询数据元素的数量（LLEN/HLEN/SCARD/ZCARD）
2. 元素数量较少，可一次性查询全量数据
3. 元素数量非常多，分批查询数据（LRANGE/HASCAN/SSCAN/ZSCAN）

规范十六：关注 DEL 时间复杂度

你没看错，在删除一个 key 时，如果姿势不对，也有可能影响到 Redis 性能。

删除一个 key，我们通常使用的是 DEL 命令，回想一下，你觉得 DEL 的时间复杂度是多少？

O(1)？其实不一定。

当你删除的是一个 String 类型 key 时，时间复杂度确实是 O(1)。

但当你要删除的 key 是 List/Hash/Set/ZSet 类型，它的复杂度其实为 O(N)，N 代表元素个数。

也就是说，删除一个 key，其元素数量越多，执行 DEL 也就越慢！

原因在于，删除大量元素时，需要依次回收每个元素的内存，元素越多，花费的时间也就越久！

而且，这个过程默认是在主线程中执行的，这势必会阻塞主线程，产生性能问题。

那删除这种元素比较多的 key，如何处理呢？

我给你的建议是，分批删除：

- List类型：执行多次 LPOP/RPOP，直到所有元素都删除完成
- Hash/Set/ZSet类型：先执行 HSCAN/SSCAN/SCAN 查询元素，再执行 HDEL/SREM/ZREM 依次删除每个元素

没想到吧？一个小小的删除操作，稍微不小心，也有可能引发性能问题，你在操作时需要格外注意。

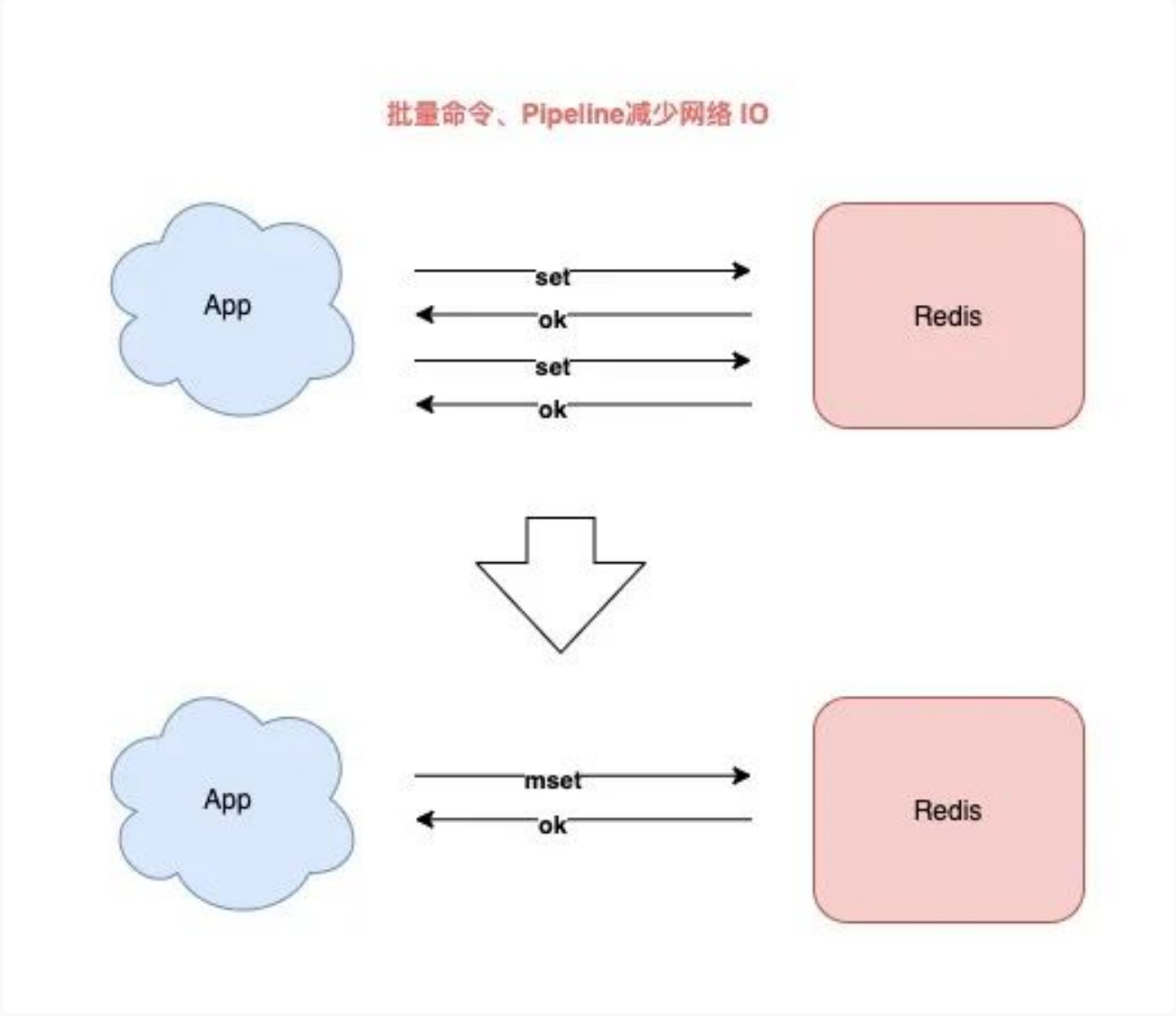
规范十七：批量命令代替单个命令

当你需要一次性操作多个 key 时，你应该使用批量命令来处理。

批量操作相比于多次单个操作的优势在于，可以显著减少客户端、服务端的来回网络 IO 次数。

所以我给你的建议是：

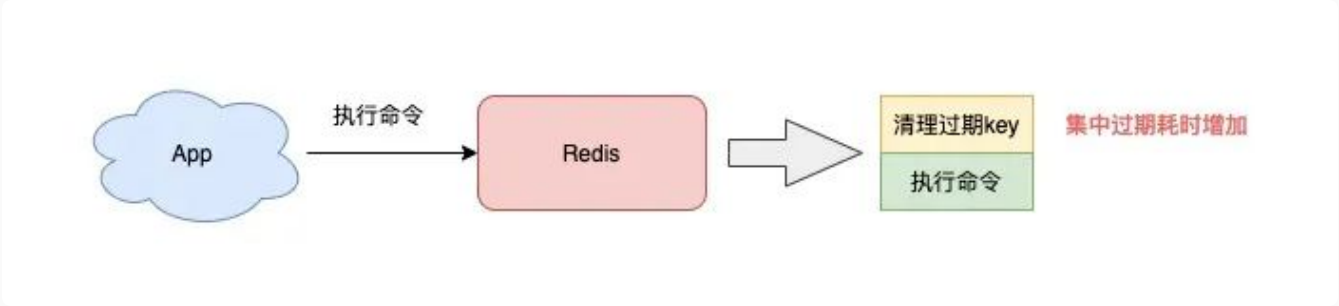
- String / Hash 使用 MGET/MSET 替代 GET/SET，HMGET/HMSET 替代 HGET/HSET
- 其它数据类型使用 Pipeline，打包一次性发送多个命令到服务端执行



规范十八：避免集中过期 key

Redis 清理过期 key 是采用定时 + 懒惰的方式来做的，而且这个过程都是在主线程中执行。

如果你的业务存在大量 key 集中过期的情况，那么 Redis 在清理过期 key 时，也会有阻塞主线程的风险。



想要避免这种情况发生，你可以在设置过期时间时，增加一个随机时间，把这些 key 的过期时间打散，从而降低集中过期对主线程的影响。

规范十九：使用长连接操作 Redis，合理配置连接池

你的业务应该使用长连接操作 Redis，避免短连接。
当使用短连接操作 Redis 时，每次都需要经过 TCP 三次握手、四次挥手，这个过程也会增加操作耗时。
同时，你的客户端应该使用连接池的方式访问 Redis，并设置合理的参数，长时间不操作 Redis 时，需及时释放连接资源。

规范二十：只使用 db0

尽管 Redis 提供了 16 个 db，但我只建议你使用 db0。
为什么呢？我总结了以下 3 点原因：

1. 在一个连接上操作多个 db 数据时，每次都需要先执行 SELECT，这会给 Redis 带来额外的压力
2. 使用多个 db 的目的是，按不同业务线存储数据，那为何不拆分多个实例存储呢？拆分多个实例部署，多个业务线不会互相影响，还能提高 Redis 的访问性能
3. Redis Cluster 只支持 db0，如果后期你想要迁移到 Redis Cluster，迁移成本高

规范二十一：使用物理机部署 Redis

Redis 在做数据持久化时，采用创建子进程的方式进行。
而创建子进程会调用操作系统的 fork 系统调用，这个系统调用的执行耗时，与系统环境有关。
虚拟机环境执行 fork 的耗时，要比物理机慢得多，所以你的 Redis 应该尽可能部署在物理机上。

规范二十二：关闭操作系统内存大页机制

Linux 操作系统提供了内存大页机制，其特点在于，每次应用程序向操作系统申请内存时，申请单位由之前的 4KB 变为了 2MB。

这会导致什么问题呢？

当 Redis 在做数据持久化时，会先 fork 一个子进程，此时主进程和子进程共享相同的内存地址空间。

当主进程需要修改现有数据时，会采用写时复制（Copy On Write）的方式进行操作，在这个过程中，需要重新申请内存。

如果申请内存单位变为了 2MB，那么势必会增加内存申请的耗时，如果此时主进程有大量写操作，需要修改原有的数据，那么在此期间，操作延迟就会变大。

规范二十三：合理配置主从复制参数

在部署主从集群时，如果参数配置不合理，也有可能导致主从复制发生问题：

- 主从复制中断
- 从库发起全量复制，主库性能受到影响

在这方面我给你的建议有以下 2 点：

1. 设置合理的 repl-backlog 参数：过小的 repl-backlog 在写流量比较大的场景下，主从复制中断会引发全量复制数据的风险
2. 设置合理的 slave client-output-buffer-limit：当从库复制发生问题时，过小的 buffer 会导致从库缓冲区溢出，从而导致复制中断

规范二十四：扫描线上实例时，设置休眠时间

不管你是使用 SCAN 扫描线上实例，还是对实例做 bigkey 统计分析，我建议你在扫描时一定记得设置休眠时间。
防止在扫描过程中，实例 OPS 过高对 Redis 产生性能抖动。

规范二十五： 从库必须设置为 slave-read-only

你的从库必须设置为 slave-read-only 状态，避免从库写入数据，导致主从数据不一致。
除此之外，从库如果是非 read-only 状态，如果你使用的是 4.0 以下的 Redis，它存在这样的 Bug：

从库写入了有过期时间的数据，不会做定时清理和释放内存。
这会造成从库的内存泄露！这个问题直到 4.0 版本才修复，你在配置从库时需要格外注意。

规范二十六：合理配置 timeout 和 tcp-keepalive 参数

如果因为网络原因，导致你的大量客户端连接与 Redis 意外中断，恰好你的 Redis 配置的 maxclients 参数比较小，此时有可能导致客户端无法与服务端建立新的连接（服务端认为超过了 maxclients）。

造成这个问题原因在于，客户端与服务端每建立一个连接，Redis 都会给这个客户端分配了一个 client fd。

当客户端与服务端网络发生问题时，服务端并不会立即释放这个 client fd。

什么时候释放呢?
Redis 内部有一个定时任务，会定时检测所有 client 的空闲时间是否超过配置的 timeout 值。
如果 Redis 没有开启 tcp-keepalive 的话，服务端直到配置的 timeout 时间后，才会清理释放这个 client fd。

在没有清理之前，如果还有大量新连接进来，就有可能导致 Redis 服务端内部持有的 client fd 超过了 maxclients，这时新连接就会被拒绝。

针对这种情况，我给你的优化建议是：

1. 不要配置过高的 timeout：让服务端尽快把无效的 client fd 清理掉
2. Redis 开启 tcp-keepalive：这样服务端会定时给客户端发送 TCP 心跳包，检测连接连通性，当网络异常时，可以尽快清理僵尸 client fd

规范二十七：调整 maxmemory 时，注意主从库的调整顺序

Redis 5.0 以下版本存在这样一个问题：**从库内存如果超过了 maxmemory，也会触发数据淘汰。**
在某些场景下，从库是可能优先主库达到 maxmemory 的（例如在从库执行 MONITOR 命令，输出缓冲区占用大量内存），那么此时从库开始淘汰数据，主从库就会产生不一致。
要想避免此问题，在调整 maxmemory 时，一定要注意主从库的修改顺序：

- 调大 maxmemory：先修改从库，再修改主库
- 调小 maxmemory：先修改主库，再修改从库

直到 Redis 5.0，Redis 才增加了一个配置 replica-ignore-maxmemory，默认从库超过 maxmemory 不会淘汰数据，才解决了此问题。

规范二十八：Redis 安全保证

1. 不要把 Redis 部署在公网可访问的服务器上
2. 部署时不使用默认端口 6379
3. 以普通用户启动 Redis 进程，禁止 root 用户启动
4. 限制 Redis 配置文件的目录访问权限
5. 推荐开启密码认证
6. 禁用/重命名危险命令（KEYS/FLUSHALL/FLUSHDB/CONFIG/EVAL）
7. 提前做好容量规划，主库机器预留一半内存资源，防止主从机器网络故障，引发大面积全量同步，导致主库机器内存不足的问题
8. 保证机器有足够的 CPU、内存、带宽、磁盘资源
9. 做好机器 CPU、内存、带宽、磁盘监控，资源不足时及时报警，任意资源不足都会影响 Redis 性能
10. 设置合理的 slowlog 阈值，并对其进行监控，slowlog 过多及时报警
11. 监控组件采集 Redis INFO 信息时，采用长连接，避免频繁的短连接
12. 做好实例运行时监控，重点关注 expired_keys、evicted_keys、latest_fork_usec 指标，这些指标短时突增可能会有阻塞风险