Redis进阶 - 性能调优：Redis性能调优详解

> Redis 的性能问题，涉及到的知识点非常广，几乎涵盖了 CPU、内存、网络、甚至磁盘的方方面面；同时还需要对上文中一些基础或底层有详细的了解。针对Redis的性能调优，这里整理分享一篇水滴与银弹（公众号）的文章，这篇文章可以帮助你构筑Redis性能调优的知识体系。@pdai

- Redis进阶 - 性能调优：Redis性能调优详解
  - 前言
  - Redis真的变慢了吗?
  - 使用复杂度过高的命令
  - 操作bigkey
  - 集中过期
  - 实例内存达到上限
  - fork耗时严重
  - 开启内存大页
  - 开启AOF
  - 绑定CPU
  - 使用Swap
  - 碎片整理
  - 网络带宽过载
  - 其他原因
  - 总结
  - 后记
  - 参考文章

前言

> Redis 作为优秀的内存数据库，其拥有非常高的性能，单个实例的 OPS 能够达到 10W 左右。但也正因此如此，当我们在使用 Redis 时，如果发现操作延迟变大的情况，就会与我们的预期不符。

你也许或多或少地，也遇到过以下这些场景：

- 在 Redis 上执行同样的命令，为什么有时响应很快，有时却很慢?
- 为什么 Redis 执行 SET、DEL 命令耗时也很久?
- 为什么我的 Redis 突然慢了一波，之后又恢复正常了?
- 为什么我的 Redis 稳定运行了很久，突然从某个时间点开始变慢了?
- ...

如果你并不清楚 Redis 内部的实现原理，那么在排查这种延迟问题时就会一头雾水。

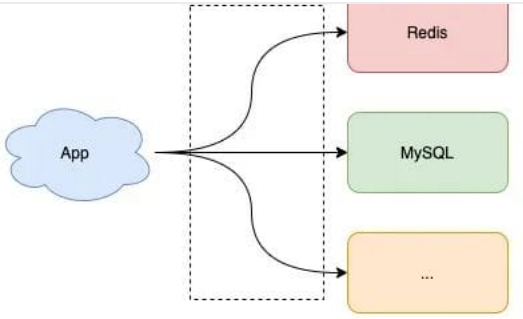
如果你也遇到了以上情况，那么，这篇文章将会给你一个「全面」的问题排查思路，并且针对这些导致变慢的场景，我还会给你一个高效的解决方案。

Redis真的变慢了吗?

> 首先，在开始之前，你需要弄清楚 Redis 是否真的变慢了?

如果你发现你的业务服务 API 响应延迟变长，首先你需要先排查服务内部，究竟是哪个环节拖慢了整个服务。

比较高效的做法是，在服务内部集成链路追踪，也就是在服务访问外部依赖的出入口，记录下每次请求外部依赖的响应延时。



如果你发现确实是操作 Redis 的这条链路耗时变长了，那么此刻你需要把焦点关注在业务服务到 Redis 这条链路上。

从你的业务服务到 Redis 这条链路变慢的原因可能也有 2 个：

- 业务服务器到 Redis 服务器之间的网络存在问题，例如网络线路质量不佳，网络数据包在传输时存在延迟、丢包等情况
- Redis 本身存在问题，需要进一步排查是什么原因导致 Redis 变慢

通常来说，第一种情况发生的概率比较小，如果是服务器之间网络存在问题，那部署在这台业务服务器上的所有服务都会发生网络延迟的情况，此时你需要联系网络运维同事，让其协助解决网络问题。

我们这篇文章，重点关注的是第二种情况。

也就是从 Redis 角度来排查，是否存在导致变慢的场景，以及都有哪些因素会导致 Redis 的延迟增加，然后针对性地进行优化。

排除网络原因，如何确认你的 Redis 是否真的变慢了？

首先，你需要对 Redis 进行基准性能测试，了解你的 Redis 在生产环境服务器上的基准性能。

什么是基准性能？

简单来讲，基准性能就是指 Redis 在一台负载正常的机器上，其最大的响应延迟和平均响应延迟分别是怎样的？

为什么要测试基准性能？我参考别人提供的响应延迟，判断自己的 Redis 是否变慢不行吗？

答案是否定的。

因为 Redis 在不同的软硬件环境下，它的性能是各不相同的。

例如，我的机器配置比较低，当延迟为 2ms 时，我就认为 Redis 变慢了，但是如果你的硬件配置比较高，那么在你的运行环境下，可能延迟是 0.5ms 时就可以认为 Redis 变慢了。

所以，你只有了解了你的 Redis 在生产环境服务器上的基准性能，才能进一步评估，当其延迟达到什么程度时，才认为 Redis 确实变慢了。

具体如何做？

为了避免业务服务器到 Redis 服务器之间的网络延迟，你需要直接在 Redis 服务器上测试实例的响应延迟情况。执行以下命令，就可以测试出这个实例 60 秒内的最大响应延迟：

```sh
$ redis-cli -h 127.0.0.1 -p 6379 --intrinsic-latency 60
Max latency so far: 1 microseconds.
Max latency so far: 15 microseconds.
Max latency so far: 17 microseconds.
Max latency so far: 18 microseconds.
Max latency so far: 31 microseconds.
Max latency so far: 32 microseconds.
Max latency so far: 59 microseconds.
Max latency so far: 72 microseconds.

1428669267 total runs (avg latency: 0.0420 microseconds / 42.00 nanoseconds per run).
Worst run took 1429x longer than the average latency.
```

从输出结果可以看到，这 60 秒内的最大响应延迟为 72 微秒（0.072毫秒）。

你还可以使用以下命令，查看一段时间内 Redis 的最小、最大、平均访问延迟：

```sh
$ redis-cli -h 127.0.0.1 -p 6379 --latency-history -i 1
min: 0, max: 1, avg: 0.13 (100 samples) -- 1.01 seconds range
min: 0, max: 1, avg: 0.12 (99 samples) -- 1.01 seconds range
```

```
min: 0, max: 1, avg: 0.08 (99 samples) -- 1.01 seconds range
...
```

以上输出结果是，每间隔 1 秒，采样 Redis 的平均操作耗时，其结果分布在 0.08 ~ 0.13 毫秒之间。

了解了基准性能测试方法，那么你就可以按照以下几步，来判断你的 Redis 是否真的变慢了：

- 在相同配置的服务器上，测试一个正常 Redis 实例的基准性能
- 找到你认为可能变慢的 Redis 实例，测试这个实例的基准性能
- 如果你观察到，这个实例的运行延迟是正常 Redis 基准性能的 2 倍以上，即可认为这个 Redis 实例确实变慢了

确认是 Redis 变慢了，那如何排查是哪里发生了问题呢？

下面跟着我的思路，我们从易到难，一步步来分析可能导致 Redis 变慢的因素。

使用复杂度过高的命令

> 首先，第一步，你需要去查看一下 Redis 的慢日志（slowlog）。

Redis 提供了慢日志命令的统计功能，它记录了有哪些命令在执行时耗时比较久。

查看 Redis 慢日志之前，你需要设置慢日志的阈值。例如，设置慢日志的阈值为 5 毫秒，并且保留最近 500 条慢日志记录：

sh
```
# 命令执行耗时超过 5 毫秒，记录慢日志
CONFIG SET slowlog-log-slower-than 5000
# 只保留最近 500 条慢日志
CONFIG SET slowlog-max-len 500
```

设置完成之后，所有执行的命令如果操作耗时超过了 5 毫秒，都会被 Redis 记录下来。

此时，你可以执行以下命令，就可以查询到最近记录的慢日志：

sh
```
127.0.0.1:6379> SLOWLOG get 5
1) 1) (integer) 32693       # 慢日志ID
   2) (integer) 1593763337  # 执行时间戳
   3) (integer) 5299        # 执行耗时(微秒)
   4) 1) "LRANGE"           # 具体执行的命令和参数
      2) "user_list:2000"
      3) "0"
      4) "-1"
2) 1) (integer) 32692
   2) (integer) 1593763337
   3) (integer) 5044
   4) 1) "GET"
      2) "user_info:1000"
...
```

通过查看慢日志，我们就可以知道在什么时间点，执行了哪些命令比较耗时。

如果你的应用程序执行的 Redis 命令有以下特点，那么有可能会导致操作延迟变大：

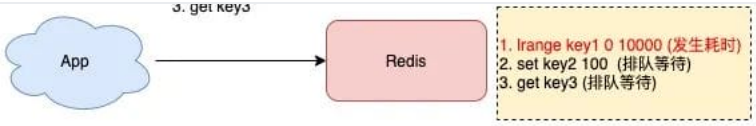
- 经常使用 O(N) 以上复杂度的命令，例如 SORT、SUNION、ZUNIONSTORE 聚合类命令
- 使用 O(N) 复杂度的命令，但 N 的值非常大

第一种情况导致变慢的原因在于，Redis 在操作内存数据时，时间复杂度过高，要花费更多的 CPU 资源。

第二种情况导致变慢的原因在于，Redis 一次需要返回给客户端的数据过多，更多时间花费在数据协议的组装和网络传输过程中。

另外，我们还可以从资源使用率层面来分析，如果你的应用程序操作 Redis 的 OPS 不是很大，但 Redis 实例的 CPU 使用率却很高，那么很有可能是使用了复杂度过高的命令导致的。

除此之外，我们都知道，Redis 是单线程处理客户端请求的，如果你经常使用以上命令，那么当 Redis 处理客户端请求时，一旦前面某个命令发生耗时，就会导致后面的请求发生排队，对于客户端来说，响应延迟也会变长。



针对这种情况如何解决呢?

答案很简单，你可以使用以下方法优化你的业务：

- 尽量不使用 O(N) 以上复杂度过高的命令，对于数据的聚合操作，放在客户端做
- 执行 O(N) 命令，保证 N 尽量的小（推荐 N <= 300），每次获取尽量少的数据，让 Redis 可以及时处理返回

操作bigkey

> 如果你查询慢日志发现，并不是复杂度过高的命令导致的，而都是 SET / DEL 这种简单命令出现在慢日志中，那么你就要怀疑你的实例否写入了 bigkey。

Redis 在写入数据时，需要为新的数据分配内存，相对应的，当从 Redis 中删除数据时，它会释放对应的内存空间。

如果一个 key 写入的 value 非常大，那么 Redis 在分配内存时就会比较耗时。同样的，当删除这个 key 时，释放内存也会比较耗时，这种类型的 key 我们一般称之为 bigkey。

此时，你需要检查你的业务代码，是否存在写入 bigkey 的情况。你需要评估写入一个 key 的数据大小，尽量避免一个 key 存入过大的数据。

如果已经写入了 bigkey，那有没有什么办法可以扫描出实例中 bigkey 的分布情况呢?

答案是可以的。

Redis 提供了扫描 bigkey 的命令，执行以下命令就可以扫描出，一个实例中 bigkey 的分布情况，输出结果是以类型维度展示的：

```sh
$ redis-cli -h 127.0.0.1 -p 6379 --bigkeys -i 0.01

...
-------- summary -------

Sampled 829675 keys in the keyspace!
Total key length in bytes is 10059825 (avg len 12.13)

Biggest string found 'key:291880' has 10 bytes
Biggest   list found 'mylist:004' has 40 items
Biggest    set found 'myset:2386' has 38 members
Biggest   hash found 'myhash:3574' has 37 fields
Biggest   zset found 'myzset:2704' has 42 members

36313 strings with 363130 bytes (04.38% of keys, avg size 10.00)
787393 lists with 896540 items (94.90% of keys, avg size 1.14)
1994 sets with 40052 members (00.24% of keys, avg size 20.09)
1990 hashs with 39632 fields (00.24% of keys, avg size 19.92)
1985 zsets with 39750 members (00.24% of keys, avg size 20.03)
```

从输出结果我们可以很清晰地看到，每种数据类型所占用的最大内存 / 拥有最多元素的 key 是哪一个，以及每种数据类型在整个实例中的占比和平均大小 / 元素数量。

其实，使用这个命令的原理，就是 Redis 在内部执行了 SCAN 命令，遍历整个实例中所有的 key，然后针对 key 的类型，分别执行 STRLEN、LLEN、HLEN、SCARD、ZCARD 命令，来获取 String 类型的长度、容器类型（List、Hash、Set、ZSet）的元素个数。

这里我需要提醒你的是，当执行这个命令时，要注意 2 个问题：

- 对线上实例进行 bigkey 扫描时，Redis 的 OPS 会突增，为了降低扫描过程中对 Redis 的影响，最好控制一下扫描的频率，指定 -i 参数即可，它表示扫描过程中每次扫描后休息的时间间隔，单位是秒
- 扫描结果中，对于容器类型（List、Hash、Set、ZSet）的 key，只能扫描出元素最多的 key。但一个 key 的元素多，不一定表示占用内存也多，你还需要根据业务情况，进一步评估内存占用情况

那针对 bigkey 导致延迟的问题，有什么好的解决方案呢?

这里有两点可以优化：

- 如果你使用的 Redis 是 6.0 以上版本，可以开启 lazy-free 机制（lazyfree-lazy-user-del = yes），在执行 DEL 命令时，释放内存也会放到后台线程中执行

但即便可以使用方案 2，我也不建议你在实例中存入 bigkey。

这是因为 bigkey 在很多场景下，依旧会产生性能问题。例如，bigkey 在分片集群模式下，对于数据的迁移也会有性能影响，以及我后面即将讲到的数据过期、数据淘汰、透明大页，都会受到 bigkey 的影响。

集中过期

> 如果你发现，平时在操作 Redis 时，并没有延迟很大的情况发生，但在某个时间点突然出现一波延时，其现象表现为：**变慢的时间点很有规律，例如某个整点，或者每间隔多久就会发生一波延迟。**如果是出现这种情况，那么你需要排查一下，业务代码中是否存在设置大量 key 集中过期的情况。

如果有大量的 key 在某个固定时间点集中过期，在这个时间点访问 Redis 时，就有可能导致延时变大。

为什么集中过期会导致 Redis 延迟变大?

这就需要我们了解 Redis 的过期策略是怎样的。

Redis 的过期数据采用被动过期 + 主动过期两种策略：

- **被动过期**：只有当访问某个 key 时，才判断这个 key 是否已过期，如果已过期，则从实例中删除
- **主动过期**：Redis 内部维护了一个定时任务，默认每隔 100 毫秒（1秒10次）就会从全局的过期哈希表中随机取出 20 个 key，然后删除其中过期的 key，如果过期 key 的比例超过了 25%，则继续重复此过程，直到过期 key 的比例下降到 25% 以下，或者这次任务的执行耗时超过了 25 毫秒，才会退出循环

注意，**这个主动过期 key 的定时任务，是在 Redis 主线程中执行的。**

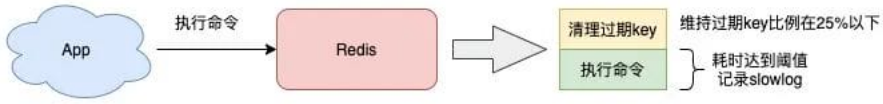
也就是说如果在执行主动过期的过程中，出现了需要大量删除过期 key 的情况，那么此时应用程序在访问 Redis 时，必须要等待这个过期任务执行结束，Redis 才可以服务这个客户端请求。

此时就会出现，应用访问 Redis 延时变大。

如果此时需要过期删除的是一个 bigkey，那么这个耗时会更久。而且，这个操作延迟的命令并不会记录在慢日志中。

因为慢日志中只记录一个命令真正操作内存数据的耗时，而 Redis 主动删除过期 key 的逻辑，是在命令真正执行之前执行的。

所以，此时你会看到，**慢日志中没有操作耗时的命令，但我们的应用程序却感知到了延迟变大，其实时间都花费在了删除过期 key 上**，这种情况我们需要尤为注意。



那遇到这种情况，如何分析和排查?

此时，你需要检查你的业务代码，是否存在集中过期 key 的逻辑。

一般集中过期使用的是 expireat / pexpireat 命令，你需要在代码中搜索这个关键字。

排查代码后，如果确实存在集中过期 key 的逻辑存在，但这种逻辑又是业务所必须的，那此时如何优化，同时又不对 Redis 有性能影响呢?

一般有两种方案来规避这个问题：

- 集中过期 key 增加一个随机过期时间，把集中过期的时间打散，降低 Redis 清理过期 key 的压力
- 如果你使用的 Redis 是 4.0 以上版本，可以开启 lazy-free 机制，当删除过期 key 时，把释放内存的操作放到后台线程中执行，避免阻塞主线程

第一种方案，在设置 key 的过期时间时，增加一个随机时间，伪代码可以这么写：

```sh
# 在过期时间点之后的 5 分钟内随机过期掉
redis.expireat(key, expire_time + random(300))
```

这样一来，Redis 在处理过期时，不会因为集中删除过多的 key 导致压力过大，从而避免阻塞主线程。

第二种方案，Redis 4.0 以上版本，开启 lazy-free 机制：

```sh
# 释放过期 key 的内存，放到后台线程执行

```

另外，除了业务层面的优化和修改配置之外，你还可以通过运维手段及时发现这种情况。

运维层面，你需要把 Redis 的各项运行状态数据监控起来，在 Redis 上执行 INFO 命令就可以拿到这个实例所有的运行状态数据。

在这里我们需要重点关注 expired_keys 这一项，它代表整个实例到目前为止，累计删除过期 key 的数量。

你需要把这个指标监控起来，**当这个指标在很短时间内出现了突增，需要及时报警出来**，然后与业务应用报慢的时间点进行对比分析，确认时间是否一致，如果一致，则可以确认确实是因为集中过期 key 导致的延迟变大。

实例内存达到上限

> 如果你的 Redis 实例设置了内存上限 maxmemory，那么也有可能导致 Redis 变慢。

当我们把 Redis 当做纯缓存使用时，通常会给这个实例设置一个内存上限 maxmemory，然后设置一个数据淘汰策略。

而当实例的内存达到了 maxmemory 后，你可能会发现，在此之后每次写入新数据，操作延迟变大了。

这是为什么？

原因在于，当 Redis 内存达到 maxmemory 后，每次写入新的数据之前，**Redis 必须先从实例中踢出一部分数据，让整个实例的内存维持在 maxmemory 之下，然后才能把新数据写进来。**

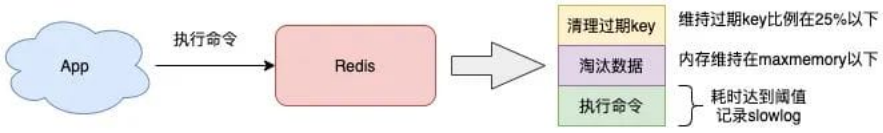
这个踢出旧数据的逻辑也是需要消耗时间的，而具体耗时的长短，要取决于你配置的淘汰策略：

- `allkeys-lru`：不管 key 是否设置了过期，淘汰最近最少访问的 key
- `volatile-lru`：只淘汰最近最少访问、并设置了过期时间的 key
- `allkeys-random`：不管 key 是否设置了过期，随机淘汰 key
- `volatile-random`：只随机淘汰设置了过期时间的 key
- `allkeys-ttl`：不管 key 是否设置了过期，淘汰即将过期的 key
- `noeviction`：不淘汰任何 key，实例内存达到 maxmeory 后，再写入新数据直接返回错误
- `allkeys-lfu`：不管 key 是否设置了过期，淘汰访问频率最低的 key（4.0+版本支持）
- `volatile-lfu`：只淘汰访问频率最低、并设置了过期时间 key（4.0+版本支持）

具体使用哪种策略，我们需要根据具体的业务场景来配置。

一般最常使用的是 allkeys-lru / volatile-lru 淘汰策略，它们的处理逻辑是，每次从实例中随机取出一批 key（这个数量可配置），然后淘汰一个最少访问的 key，之后把剩下的 key 暂存到一个池子中，继续随机取一批 key，并与之前池子中的 key 比较，再淘汰一个最少访问的 key。以此往复，直到实例内存降到 maxmemory 之下。

需要注意的是，Redis 的淘汰数据的逻辑与删除过期 key 的一样，也是在命令真正执行之前执行的，也就是说它也会增加我们操作 Redis 的延迟，而且，写 OPS 越高，延迟也会越明显。



另外，如果此时你的 Redis 实例中还存储了 bigkey，那么在淘汰删除 bigkey 释放内存时，也会耗时比较久。

看到了么？bigkey 的危害到处都是，这也是前面我提醒你尽量不存储 bigkey 的原因。

针对这种情况，如何解决呢？

我给你 4 个方面的优化建议：

- 避免存储 bigkey，降低释放内存的耗时
- 淘汰策略改为随机淘汰，随机淘汰比 LRU 要快很多（视业务情况调整）
- 拆分实例，把淘汰 key 的压力分摊到多个实例上
- 如果使用的是 Redis 4.0 以上版本，开启 layz-free 机制，把淘汰 key 释放内存的操作放到后台线程中执行（配置 lazyfree-lazy-eviction = yes）

fork耗时严重

当 Redis 开启了后台 RDB 和 AOF rewrite 后，在执行时，它们都需要主进程创建出一个子进程进行数据的持久化。

主进程创建子进程，会调用操作系统提供的 fork 函数。

而 fork 在执行过程中，主进程需要拷贝自己的内存页表给子进程，如果这个实例很大，那么这个拷贝的过程也会比较耗时。

而且这个 fork 过程会消耗大量的 CPU 资源，在完成 fork 之前，整个 Redis 实例会被阻塞住，无法处理任何客户端请求。

如果此时你的 CPU 资源本来就很紧张，那么 fork 的耗时会更长，甚至达到秒级，这会严重影响 Redis 的性能。

那如何确认确实是因为 fork 耗时导致的 Redis 延迟变大呢?

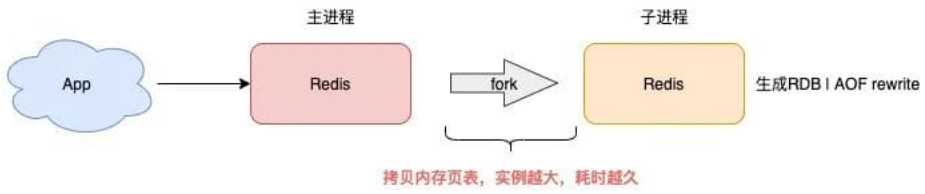
你可以在 Redis 上执行 INFO 命令，查看 latest_fork_usec 项，单位微秒。

```sh
# 上一次 fork 耗时，单位微秒
latest_fork_usec:59477
```

这个时间就是主进程在 fork 子进程期间，整个实例阻塞无法处理客户端请求的时间。

如果你发现这个耗时很久，就要警惕起来了，这意味在这期间，你的整个 Redis 实例都处于不可用的状态。

除了数据持久化会生成 RDB 之外，当主从节点第一次建立数据同步时，主节点也创建子进程生成 RDB，然后发给从节点进行一次全量同步，所以，这个过程也会对 Redis 产生性能影响。



要想避免这种情况，你可以采取以下方案进行优化：

- 控制 Redis 实例的内存：尽量在 10G 以下，执行 fork 的耗时与实例大小有关，实例越大，耗时越久
- 合理配置数据持久化策略：在 slave 节点执行 RDB 备份，推荐在低峰期执行，而对于丢失数据不敏感的业务（例如把 Redis 当做纯缓存使用），可以关闭 AOF 和 AOF rewrite
- Redis 实例不要部署在虚拟机上：fork 的耗时也与系统也有关，虚拟机比物理机耗时更久
- 降低主从库全量同步的概率：适当调大 repl-backlog-size 参数，避免主从全量同步

开启内存大页

> 除了上面讲到的子进程 RDB 和 AOF rewrite 期间，fork 耗时导致的延时变大之外，这里还有一个方面也会导致性能问题，这就是操作系统是否开启了内存大页机制。

什么是内存大页?

我们都知道，应用程序向操作系统申请内存时，是按内存页进行申请的，而常规的内存页大小是 4KB。

Linux 内核从 2.6.38 开始，支持了内存大页机制，该机制允许应用程序以 2MB 大小为单位，向操作系统申请内存。

应用程序每次向操作系统申请的内存单位变大了，但这也意味着申请内存的耗时变长。

这对 Redis 会有什么影响呢?

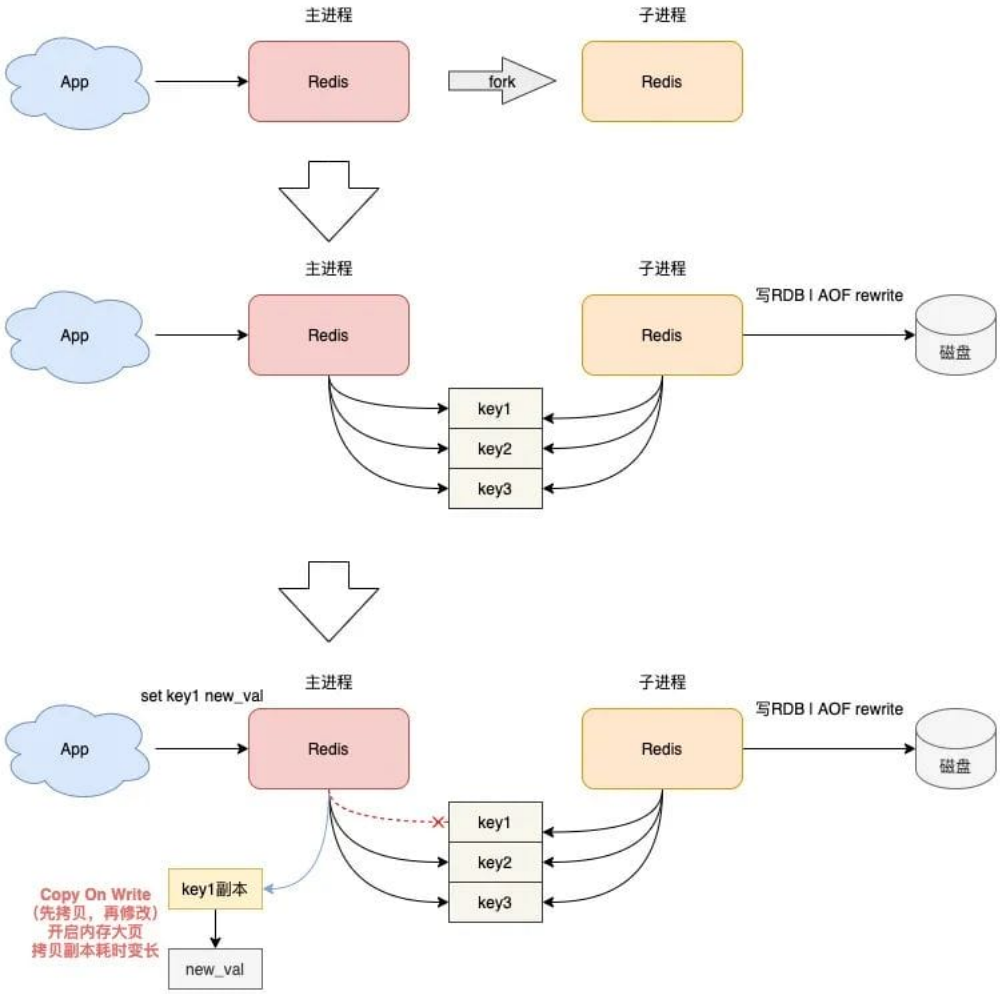
当 Redis 在执行后台 RDB 和 AOF rewrite 时，采用 fork 子进程的方式来处理。但主进程 fork 子进程后，此时的主进程依旧是可以接收写请求的，而进来的写请求，会采用 Copy On Write（写时复制）的方式操作内存数据。

也就是说，主进程一旦有数据需要修改，Redis 并不会直接修改现有内存中的数据，而是先将这块内存数据拷贝出来，再修改这块新内存的数据，这就是所谓的「写时复制」。

写时复制你也可以理解成，谁需要发生写操作，谁就需要先拷贝，再修改。

但是请注意，**主进程在拷贝内存数据时，这个阶段就涉及到新内存的申请，如果此时操作系统开启了内存大页，那么在此期间，客户端即便只修改 10B 的数据，Redis 在申请内存时也会以 2MB 为单位向操作系统申请，申请内存的耗时变长，进而导致每个写请求的延迟增加，影响到 Redis 性能。**

同样地，如果这个写请求操作的是一个 bigkey，那主进程在拷贝这个 bigkey 内存块时，一次申请的内存会更大，时间也会更久。可见，bigkey 在这里又一次影响到了性能。



那如何解决这个问题?

很简单，你只需要关闭内存大页机制就可以了。

首先，你需要查看 Redis 机器是否开启了内存大页：

```sh
$ cat /sys/kernel/mm/transparent_hugepage/enabled
[always] madvise never
```

如果输出选项是 always，就表示目前开启了内存大页机制，我们需要关掉它：

```sh
$ echo never > /sys/kernel/mm/transparent_hugepage/enabled
```

其实，操作系统提供的内存大页机制，其优势是，可以在一定程序上降低应用程序申请内存的次数。

但是对于 Redis 这种对性能和延迟极其敏感的数据库来说，我们希望 Redis 在每次申请内存时，耗时尽量短，所以我不建议你在 Redis 机器上开启这个机制。

开启AOF

> 前面我们分析了 RDB 和 AOF rewrite 对 Redis 性能的影响，主要关注点在 fork 上。

当 Redis 开启 AOF 后，其工作原理如下：

- Redis 执行写命令后，把这个命令写入到 AOF 文件内存中（write 系统调用）
- Redis 根据配置的 AOF 刷盘策略，把 AOF 内存数据刷到磁盘上（fsync 系统调用）

为了保证 AOF 文件数据的安全性，Redis 提供了 3 种刷盘机制：

- appendfsync always：主线程每次执行写操作后立即刷盘，此方案会占用比较大的磁盘 IO 资源，但数据安全性最高
- appendfsync no：主线程每次写操作只写内存就返回，内存数据什么时候刷到磁盘，交由操作系统决定，此方案对性能影响最小，但数据安全性也最低，Redis 宕机时丢失的数据取决于操作系统刷盘时机
- appendfsync everysec：主线程每次写操作只写内存就返回，然后由后台线程每隔 1 秒执行一次刷盘操作（触发fsync系统调用），此方案对性能影响相对较小，但当 Redis 宕机时会丢失 1 秒的数据

下面我们依次来分析，这几个机制对性能的影响。

如果你的 AOF 配置为 appendfsync always，那么 Redis 每处理一次写操作，都会把这个命令写入到磁盘中才返回，整个过程都是在主线程执行的，这个过程必然会加重 Redis 写负担。

原因也很简单，操作磁盘要比操作内存慢几百倍，采用这个配置会严重拖慢 Redis 的性能，因此我不建议你把 AOF 刷盘方式配置为 always。

我们接着来看 appendfsync no 配置项。

在这种配置下，Redis 每次写操作只写内存，什么时候把内存中的数据刷到磁盘，交给操作系统决定，此方案对 Redis 的性能影响最小，但当 Redis 宕机时，会丢失一部分数据，为了数据的安全性，一般我们也不采取这种配置。

> 如果你的 Redis 只用作纯缓存，对于数据丢失不敏感，采用配置 appendfsync no 也是可以的。

看到这里，我猜你肯定和大多数人的想法一样，选比较折中的方案 appendfsync everysec 就没问题了吧？

这个方案优势在于，Redis 主线程写完内存后就返回，具体的刷盘操作是放到后台线程中执行的，后台线程每隔 1 秒把内存中的数据刷到磁盘中。

这种方案既兼顾了性能，又尽可能地保证了数据安全，是不是觉得很完美？

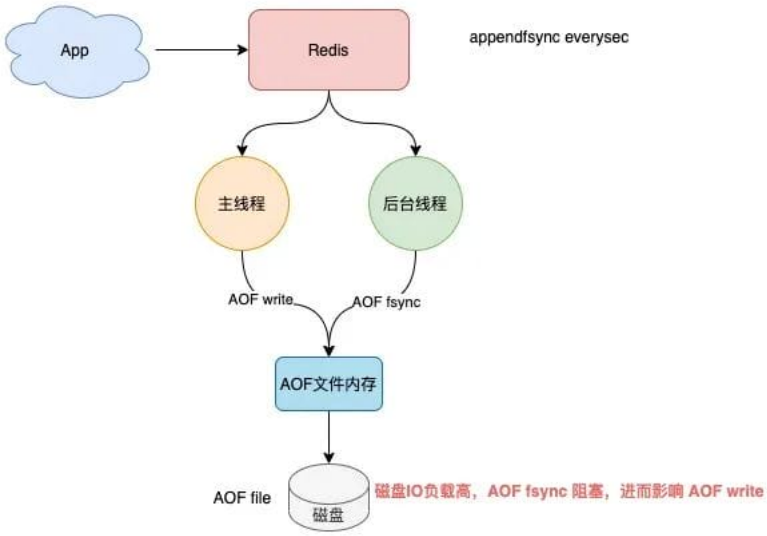
但是，这里我要给你泼一盆冷水了，采用这种方案你也要警惕一下，因为这种方案还是存在导致 Redis 延迟变大的情况发生，甚至会阻塞整个 Redis。

这是为什么？我把 AOF 最耗时的刷盘操作，放到后台线程中也会影响到 Redis 主线程？

你试想这样一种情况：当 Redis 后台线程在执行 AOF 文件刷盘时，如果此时磁盘的 IO 负载很高，那这个后台线程在执行刷盘操作（fsync系统调用）时就会被阻塞住。

此时的主线程依旧会接收写请求，紧接着，主线程又需要把数据写到文件内存中（write 系统调用），**但此时的后台子线程由于磁盘负载过高，导致 fsync 发生阻塞，迟迟不能返回，那主线程在执行 write 系统调用时，也会被阻塞住，直到后台线程 fsync 执行完成后，主线程执行 write 才能成功返回。**

看到了么？在这个过程中，主线程依旧有阻塞的风险。



所以，尽管你的 AOF 配置为 appendfsync everysec，也不能掉以轻心，要警惕磁盘压力过大导致的 Redis 有性能问题。

- 子进程正在执行 AOF rewrite，这个过程会占用大量的磁盘 IO 资源
- 有其他应用程序在执行大量的写文件操作，也会占用磁盘 IO 资源

对于情况1，说白了就是，Redis 的 AOF 后台子线程刷盘操作，撞上了子进程 AOF rewrite！

这怎么办？难道要关闭 AOF rewrite 才行？

幸运的是，Redis 提供了一个配置项，当子进程在 AOF rewrite 期间，可以让后台子线程不执行刷盘（不触发 fsync 系统调用）操作。

这相当于在 AOF rewrite 期间，临时把 appendfsync 设置为了 none，配置如下：

```sh
# AOF rewrite 期间，AOF 后台子线程不进行刷盘操作
# 相当于在这期间，临时把 appendfsync 设置为了 none
no-appendfsync-on-rewrite yes
```

当然，开启这个配置项，在 AOF rewrite 期间，如果实例发生宕机，那么此时会丢失更多的数据，性能和数据安全性，你需要权衡后进行选择。

如果占用磁盘资源的是其他应用程序，那就比较简单了，你需要定位到是哪个应用程序在大量写磁盘，然后把这个应用程序迁移到其他机器上执行就好了，避免对 Redis 产生影响。

当然，如果你对 Redis 的性能和数据安全都有很高的要求，那么我建议从硬件层面来优化，更换为 SSD 磁盘，提高磁盘的 IO 能力，保证 AOF 期间有充足的磁盘资源可以使用。

绑定CPU

> 很多时候，我们在部署服务时，为了提高服务性能，降低应用程序在多个 CPU 核心之间的上下文切换带来的性能损耗，通常采用的方案是进程绑定 CPU 的方式提高性能。

但在部署 Redis 时，如果你需要绑定 CPU 来提高其性能，我建议你仔细斟酌后再做操作。

为什么？

因为 Redis 在绑定 CPU 时，是有很多考究的，如果你不了解 Redis 的运行原理，随意绑定 CPU 不仅不会提高性能，甚至有可能会带来相反的效果。

我们都知道，一般现代的服务器会有多个 CPU，而每个 CPU 又包含多个物理核心，每个物理核心又分为多个逻辑核心，每个物理核下的逻辑核共用 L1/L2 Cache。

而 Redis Server 除了主线程服务客户端请求之外，还会创建子进程、子线程。

其中子进程用于数据持久化，而子线程用于执行一些比较耗时操作，例如异步释放 fd、异步 AOF 刷盘、异步 lazy-free 等等。

如果你把 Redis 进程只绑定了一个 CPU 逻辑核心上，那么当 Redis 在进行数据持久化时，fork 出的子进程会继承父进程的 CPU 使用偏好。

而此时的子进程会消耗大量的 CPU 资源进行数据持久化（把实例数据全部扫描出来需要耗费CPU），这就会导致子进程会与主进程发生 CPU 争抢，进而影响到主进程服务客户端请求，访问延迟变大。

这就是 Redis 绑定 CPU 带来的性能问题。

那如何解决这个问题呢？

如果你确实想要绑定 CPU，可以优化的方案是，不要让 Redis 进程只绑定在一个 CPU 逻辑核上，而是绑定在多个逻辑核心上，而且，绑定的多个逻辑核心最好是同一个物理核心，这样它们还可以共用 L1/L2 Cache。

当然，即便我们把 Redis 绑定在多个逻辑核心上，也只能在一定程度上缓解主线程、子进程、后台线程在 CPU 资源上的竞争。

因为这些子进程、子线程还是会在这多个逻辑核心上进行切换，存在性能损耗。

如何再进一步优化？

可能你已经想到了，我们是否可以让主线程、子进程、后台线程，分别绑定在固定的 CPU 核心上，不让它们来回切换，这样一来，他们各自使用的 CPU 资源互不影响。

其实，这个方案 Redis 官方已经想到了。

Redis 在 6.0 版本已经推出了这个功能，我们可以通过以下配置，对主线程、后台线程、后台 RDB 进程、AOF rewrite 进程，绑定固定的 CPU 逻辑核心：

```sh
# Redis Server 和 IO 线程绑定到 CPU核心 0,2,4,6
server_cpulist 0-7:2

```

```
# 后台 AOF rewrite 进程绑定到 CPU 核心 8,9,10,11
aof_rewrite_cpulist 8-11

# 后台 RDB 进程绑定到 CPU 核心 1,10,11
# bgsave_cpulist 1,10-1
```

如果你使用的正好是 Redis 6.0 版本，就可以通过以上配置，来进一步提高 Redis 性能。

这里我需要提醒你的是，一般来说，Redis 的性能已经足够优秀，除非你对 Redis 的性能有更加严苛的要求，否则不建议你绑定 CPU。

从上面的分析你也能看出，绑定 CPU 需要你对计算机体系结构有非常清晰的了解，否则谨慎操作。

我们继续分析还有什么场景会导致 Redis 变慢。

使用Swap

如果你发现 Redis 突然变得非常慢，**每次的操作耗时都达到了几百毫秒甚至秒级**，那此时你就需要检查 Redis 是否使用到了 Swap，在这种情况下 Redis 基本上已经无法提供高性能的服务了。

什么是 Swap？为什么使用 Swap 会导致 Redis 的性能下降？

如果你对操作系统有些了解，就会知道操作系统为了缓解内存不足对应用程序的影响，允许把一部分内存中的数据换到磁盘上，以达到应用程序对内存使用的缓冲，这些内存数据被换到磁盘上的区域，就是 Swap。

问题就在于，当内存中的数据被换到磁盘上后，Redis 再访问这些数据时，就需要从磁盘上读取，访问磁盘的速度要比访问内存慢几百倍！

尤其是针对 Redis 这种对性能要求极高、性能极其敏感的数据库来说，这个操作延时是无法接受的。

此时，你需要检查 Redis 机器的内存使用情况，确认是否存在使用了 Swap。

你可以通过以下方式来查看 Redis 进程是否使用到了 Swap：

```sh
# 先找到 Redis 的进程 ID
$ ps -aux | grep redis-server

# 查看 Redis Swap 使用情况
$ cat /proc/$pid/smaps | egrep '^(Swap|Size)'
```

输出结果如下：

```sh
Size:              1256 kB
Swap:                 0 kB
Size:                 4 kB
Swap:                 0 kB
Size:               132 kB
Swap:                 0 kB
Size:             63488 kB
Swap:                 0 kB
Size:               132 kB
Swap:                 0 kB
Size:             65404 kB
Swap:                 0 kB
Size:           1921024 kB
Swap:                 0 kB
...
```

这个结果会列出 Redis 进程的内存使用情况。

每一行 Size 表示 Redis 所用的一块内存大小，Size 下面的 Swap 就表示这块 Size 大小的内存，有多少数据已经被换到磁盘上了，如果这两个值相等，说明这块内存的数据都已经完全被换到磁盘上了。

如果只是少量数据被换到磁盘上，例如每一块 Swap 占对应 Size 的比例很小，那影响并不是很大。**如果是几百兆甚至上 GB 的内存被换到了磁盘上，那么你就需要警惕了，这种情况 Redis 的性能肯定会急剧下降。**

此时的解决方案是：

- 增加机器的内存，让 Redis 有足够的内存可以使用
- 整理内存空间，释放出足够的内存供 Redis 使用，然后释放 Redis 的 Swap，让 Redis 重新使用内存

可见，当 Redis 使用到 Swap 后，此时的 Redis 性能基本已达不到高性能的要求（你可以理解为武功被废），所以你也需要提前预防这种情况。

预防的办法就是，你需要对 Redis 机器的内存和 Swap 使用情况进行监控，在内存不足或使用到 Swap 时报警出来，及时处理。

碎片整理

> Redis 的数据都存储在内存中，当我们的应用程序频繁修改 Redis 中的数据时，就有可能会导致 Redis 产生内存碎片。

内存碎片会降低 Redis 的内存使用率，我们可以通过执行 INFO 命令，得到这个实例的内存碎片率：

```sh
# Memory
used_memory:5709194824
used_memory_human:5.32G
used_memory_rss:8264855552
used_memory_rss_human:7.70G
...
mem_fragmentation_ratio:1.45
```

这个内存碎片率是怎么计算的？

很简单，mem_fragmentation_ratio = used_memory_rss / used_memory。

其中 used_memory 表示 Redis 存储数据的内存大小，而 used_memory_rss 表示操作系统实际分配给 Redis 进程的大小。

如果 mem_fragmentation_ratio > 1.5，说明内存碎片率已经超过了 50%，这时我们就需要采取一些措施来降低内存碎片了。

解决的方案一般如下：

- 如果你使用的是 Redis 4.0 以下版本，只能通过重启实例来解决
- 如果你使用的是 Redis 4.0 版本，它正好提供了自动碎片整理的功能，可以通过配置开启碎片自动整理

但是，开启内存碎片整理，它也有可能会导致 Redis 性能下降。

原因在于，Redis 的碎片整理工作是也在主线程中执行的，当其进行碎片整理时，必然会消耗 CPU 资源，产生更多的耗时，从而影响到客户端的请求。

所以，当你需要开启这个功能时，最好提前测试评估它对 Redis 的影响。

Redis 碎片整理的参数配置如下：

```sh
# 开启自动内存碎片整理（总开关）
activedefrag yes

# 内存使用 100MB 以下，不进行碎片整理
active-defrag-ignore-bytes 100mb

# 内存碎片率超过 10%，开始碎片整理
active-defrag-threshold-lower 10
# 内存碎片率超过 100%，尽最大努力碎片整理
active-defrag-threshold-upper 100

# 内存碎片整理占用 CPU 资源最小百分比
active-defrag-cycle-min 1
# 内存碎片整理占用 CPU 资源最大百分比
active-defrag-cycle-max 25

# 碎片整理期间，对于 List/Set/Hash/ZSet 类型元素一次 Scan 的数量
active-defrag-max-scan-fields 1000
```

你需要结合 Redis 机器的负载情况，以及应用程序可接受的延迟范围进行评估，合理调整碎片整理的参数，尽可能降低碎片整理期间对 Redis 的影响。

网络带宽过载

> 如果以上产生性能问题的场景，你都规避掉了，而且 Redis 也稳定运行了很长时间，但在某个时间点之后开始，操作 Redis 突然开始变慢了，而且一直持续下去，这种情况又是什么原因导致？

Redis 的高性能，除了操作内存之外，就在于网络 IO 了，如果网络 IO 存在瓶颈，那么也会严重影响 Redis 的性能。

如果确实出现这种情况，你需要及时确认占满网络带宽 Redis 实例，如果属于正常的业务访问，那就需要及时扩容或迁移实例了，避免因为这个实例流量过大，影响这个机器的其他实例。

运维层面，你需要对 Redis 机器的各项指标增加监控，包括网络流量，在网络流量达到一定阈值时提前报警，及时确认和扩容。

其他原因

好了，以上这些方面就是如何排查 Redis 延迟问题的思路和路径。

除了以上这些，还有一些比较小的点，你也需要注意一下：

- **频繁短连接**

你的业务应用，应该使用长连接操作 Redis，避免频繁的短连接。

频繁的短连接会导致 Redis 大量时间耗费在连接的建立和释放上，TCP 的三次握手和四次挥手同样也会增加访问延迟。

- **运维监控**

前面我也提到了，要想提前预知 Redis 变慢的情况发生，必不可少的就是做好完善的监控。

监控其实就是对采集 Redis 的各项运行时指标，通常的做法是监控程序定时采集 Redis 的 INFO 信息，然后根据 INFO 信息中的状态数据做数据展示和报警。

这里我需要提醒你的是，在写一些监控脚本，或使用开源的监控组件时，也不能掉以轻心。

在写监控脚本访问 Redis 时，尽量采用长连接的方式采集状态信息，避免频繁短连接。同时，你还要注意控制访问 Redis 的频率，避免影响到业务请求。

在使用一些开源的监控组件时，最好了解一下这些组件的实现原理，以及正确配置这些组件，防止出现监控组件发生 Bug，导致短时大量操作 Redis，影响 Redis 性能的情况发生。

我们当时就发生过，DBA 在使用一些开源组件时，因为配置和使用问题，导致监控程序频繁地与 Redis 建立和断开连接，导致 Redis 响应变慢。

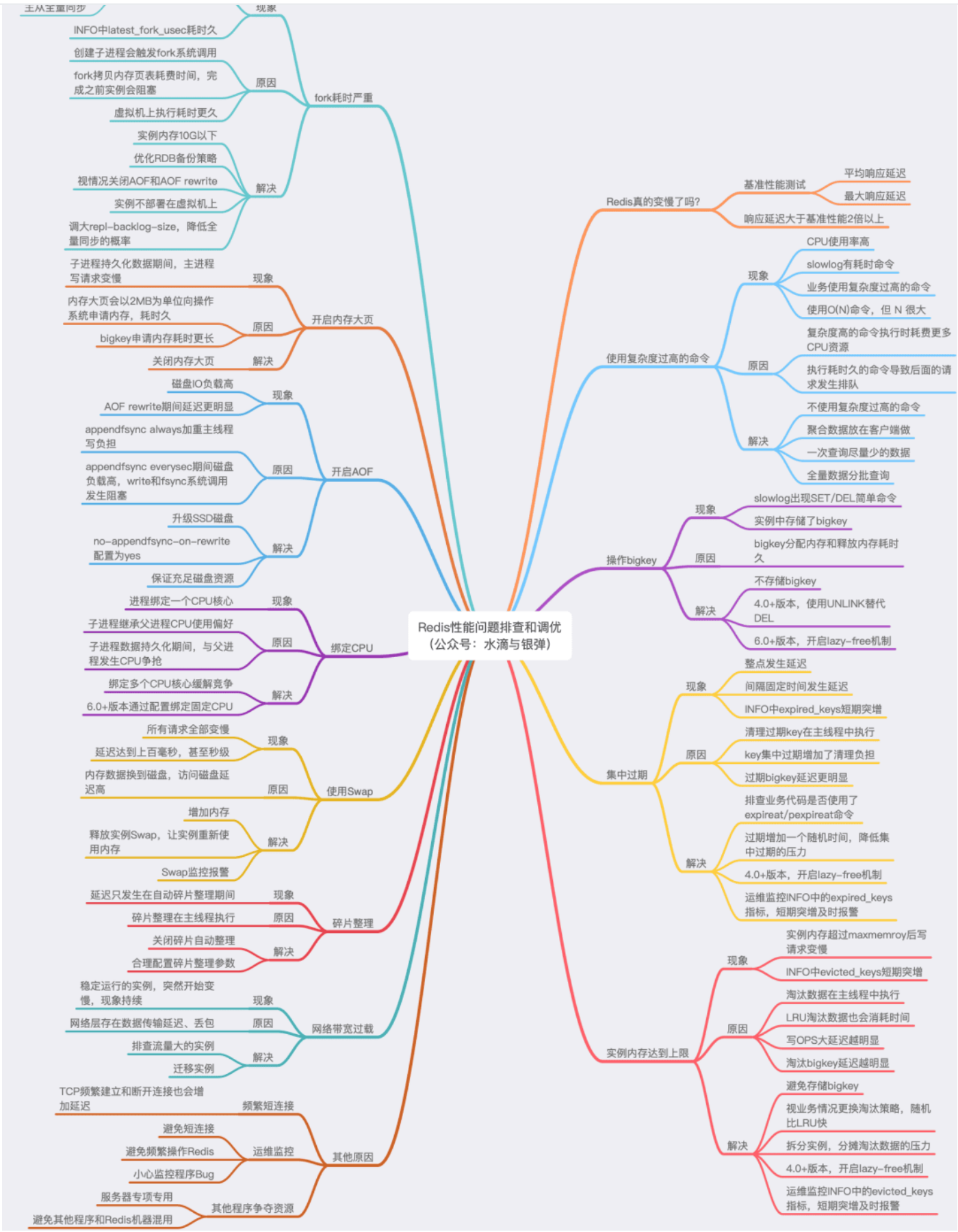
- **其它程序争抢资源**

最后需要提醒你的是，你的 Redis 机器最好专项专用，只用来部署 Redis 实例，不要部署其他应用程序，尽量给 Redis 提供一个相对「安静」的环境，避免其它程序占用 CPU、内存、磁盘资源，导致分配给 Redis 的资源不足而受到影响。

总结

> 好了，以上就是我总结的在使用 Redis 过程中，常见的可能导致延迟、甚至阻塞的问题场景，以及如何快速定位和分析这些问题，并且针对性地提供了解决方案。

这里我也汇总成了思维导图，方便你在排查 Redis 性能问题时，快速地去分析和定位。



这里再简单总结一下，Redis 的性能问题，既涉及到了业务开发人员的使用方面，也涉及到了 DBA 的运维方面。

作为业务开发人员，我们需要了解 Redis 的基本原理，例如各个命令执行的时间复杂度、数据过期策略、数据淘汰策略等，从而更合理地使用 Redis 命令，并且结合业务场景进行优化。

同时，DBA 在部署 Redis 时，需要提前对进行容量规划，预留足够的机器资源，还要对 Redis 机器和实例做好完善的监控，这样才能尽可能地保证 Redis 的稳定运行。

后记

> 如果你能耐心地看到这里，想必你肯定已经对 Redis 的性能调优有了很大的收获。

你应该也发现了，Redis 的性能问题，涉及到的知识点非常广，几乎涵盖了 CPU、内存、网络、甚至磁盘的方方面面，同时，你还需要了解计算机的体系结构，以及操作系统的各种机制。

从资源使用角度来看，包含的知识点如下：

- **CPU 相关**：使用复杂度过高命令、数据的持久化，都与耗费过多的 CPU 资源有关
- **内存相关**：bigkey 内存的申请和释放、数据过期、数据淘汰、碎片整理、内存大页、内存写时复制都与内存息息相关
- **磁盘相关**：数据持久化、AOF 刷盘策略，也会受到磁盘的影响
- **网络相关**：短连接、实例流量过载、网络流量过载，也会降低 Redis 性能
- **计算机系统**：CPU 结构、内存分配，都属于最基础的计算机系统知识
- **操作系统**：写时复制、内存大页、Swap、CPU 绑定，都属于操作系统层面的知识

没想到吧？Redis 为了把性能做到极致，涉及到了这么多项优化。

参考文章

- 公众号：水滴与银弹
- 原文：Redis为什么变慢了？一文讲透如何排查Redis性能问题
- 本文为「水滴与银弹」的原创文章，遵循 CC 4.0 BY-SA 版权协议，转载请附上原文出处链接及本声明

← Redis进阶 - 运维监控：Redis的监控详解

Redis大厂经验 - 微博：万亿级日访问量下，Redis在微博的9年优化历程 →