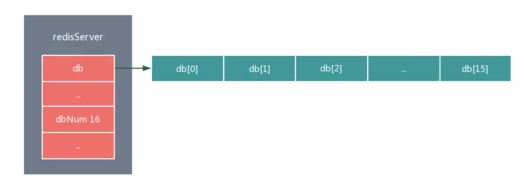
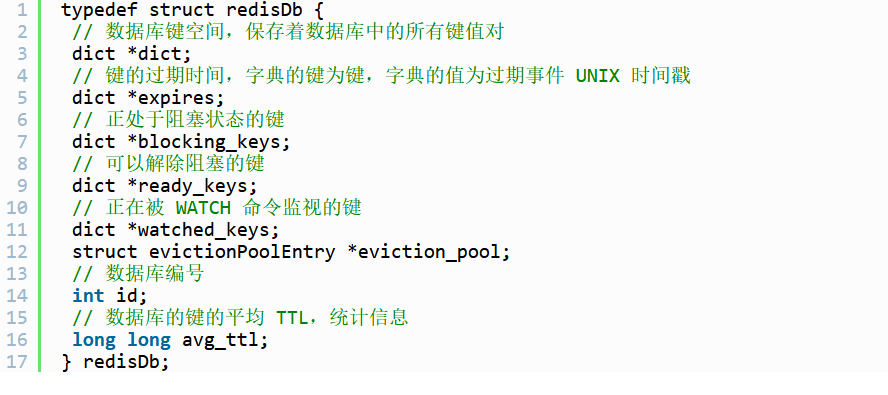
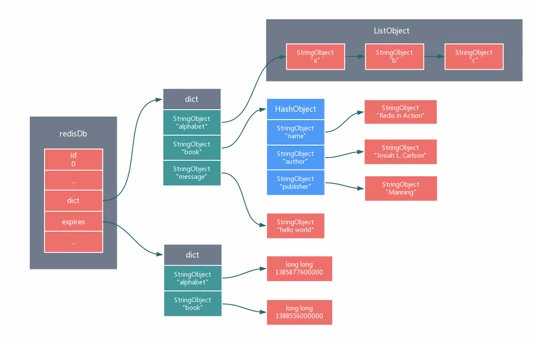
1. **Redis数据结构**
2. Redis 是内存数据库，数据都是存储在内存中，为了避免进程退出导致数据的永久丢失，需要定期将 Redis 中的数据以数据或命令的形式从内存保存到本地磁盘。当下次 Redis 重启时，利用持久化文件进行数据恢复。Redis 提供了 RDB 和 AOF 两种持久化机制，前者将当前的数据保存到磁盘，后者则是将每次执行的写命令保存到磁盘（类似于 MySQL 的 Binlog）。RDB是redis的默认持久化机制。
3. 一个单机的 Redis 服务器默认情况下有 16 个数据库（0-15号），默认使用的是 0 号数据库，可以使用 SELECT 命令切换数据库。



1. Redis 中的每个数据库都由一个 redis.h/redisDb 结构表示，它记录了单个 Redis 数据库的键空间、所有键的过期时间、处于阻塞状态和就绪状态的键、数据库编号等等。



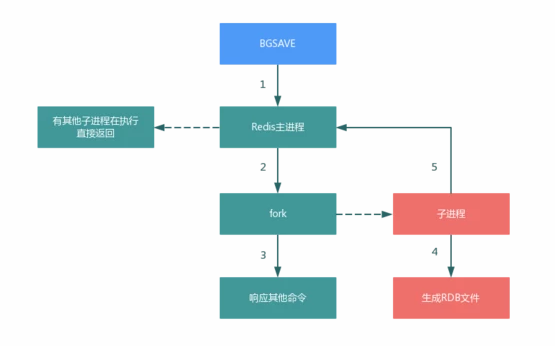
1. 由于 Redis 是一个键值对数据库（key-value pairs database）， 所以它的数据库本身也是一个字典，对应的结构正是 redisDb。其中，dict 指向的是一个记录键值对数据的字典，它的键是一个字符串对象，它的值则可以是字符串、列表、哈希表、集合和有序集合在内的任意一种 Redis 类型对象。 expires 指向的是一个用于记录键的过期时间的字典，它的键为 dict 中的数据库键，它的值为这个数据库键的过期时间戳，这个值以 long long 类型表示。



1. **RDB持久化**
2. RDB 持久化（也称作快照持久化）是指将内存中的数据生成快照保存到磁盘里面，保存的文件后缀是 .rdb。rdb 文件是一个经过压缩的二进制文件，当 Redis 重新启动时，可以读取 rdb 快照文件恢复数据。RDB 功能最核心的是 rdbSave 和 rdbLoad 两个函数， 前者用于生成 RDB 文件并保存到磁盘，而后者则用于将 RDB 文件中的数据重新载入到内存中：



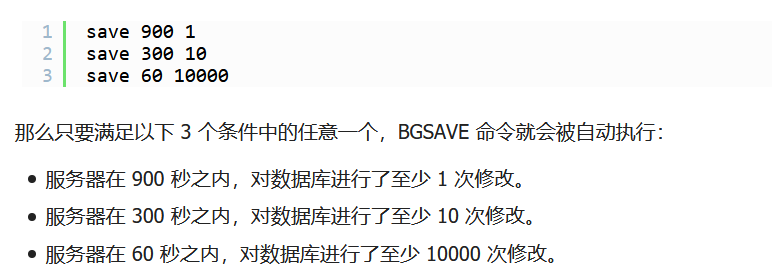
1. RDB 文件是一个单文件的全量数据，很适合数据的容灾备份与恢复，通过 RDB 文件恢复数据库耗时较短，通常 1G 的快照文件载入内存只需 20s 左右。Redis 提供了手动触发保存、自动保存间隔两种 RDB 文件的生成方式。
2. Redis 提供了两个用于生成 RDB 文件的命令，一个是 SAVE，另一个是 BGSAVE。而触发 Redis 进行 RDB 备份的方式有两种，一种是通过 SAVE 命令、BGSAVE 命令手动触发快照生成的方式，另一种是配置保存时间和写入次数，由 Redis 根据条件自动触发保存操作。
3. SAVE 是一个同步式的命令，它会阻塞 Redis 服务器进程，直到 RDB 文件创建完成为止。在服务器进程阻塞期间，服务器不能处理任何其他命令请求，这个操作期间会一直阻止redis客户端的请求处理。
4. BGSAVE 是一个异步式的命令，和 SAVE 命令直接阻塞服务器进程的做法不同，BGSAVE 命令会派生出一个子进程，由子进程负责创建 RDB 文件，服务器进程（父进程）继续处理客户的命令。在整个过程中，服务器进程只会消耗少量时间在创建子进程和处理子进程信号量上面，其余时间都是待命状态。BGSAVE 是触发 RDB 持久化的主流方式，下面给出 BGSAVE 命令生成快照的流程：



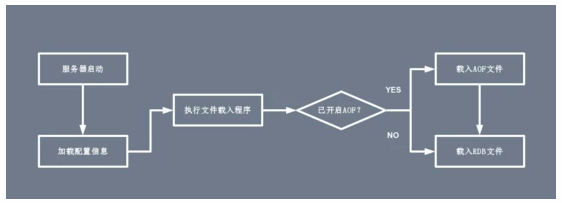
1. 客户端发起 BGSAVE 命令，Redis 主进程判断当前是否存在正在执行备份的子进程，如果存在则直接返回。
2. 父进程 fork 一个子进程 （fork 的过程中会造成阻塞的情况），这个过程可以使用 info stats 命令查看 latest\_fork\_usec 选项，查看最近一次 fork 操作消耗的时间，单位是微秒。
3. 父进程 fork 完成之后，则会返回 Background saving started 的信息提示，此时 fork 阻塞解除。
4. fork 创建的子进程开始根据父进程的内存数据生成临时的快照文件，然后替换原文件。
5. 子进程备份完毕后向父进程发送完成信息，父进程更新统计信息。



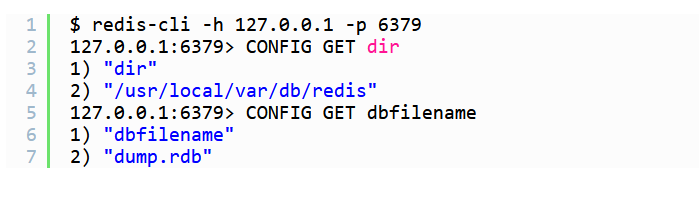
1. 因为 BGSAVE 命令可以在不阻塞服务器进程的情况下执行，所以 Redis 的配置文件 redis.conf 提供了一个 save 选项，让服务器每隔一段时间自动执行一次 BGSAVE 命令。用户可以通过 save 选项设置多个保存条件，只要其中任意一个条件被满足，服务器就会执行 BGSAVE 命令。 Redis 服务器会周期性地操作 serverCron 函数，这个函数每隔 100 毫秒就会执行一次，它的一项任务就是检查 save 选项所设置的保存条件是否满足，如果满足的话，就自动执行 BGSAVE 命令。Redis 配置文件 redis.conf 默认配置了以下 3 个保存条件：

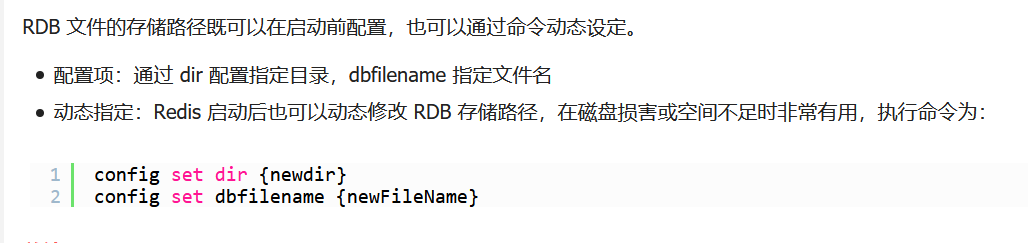


1. 和使用 SAVE 和 BGSAVE 命令创建 RDB 文件不同，Redis 没有专门提供用于载入 RDB 文件的命令，RDB 文件的载入过程是在 Redis 服务器启动时自动完成的。启动时只要在指定目录检测到 RDB 文件的存在，Redis 就会通过 rdbLoad 函数自动载入 RDB 文件。由于 AOF 文件属于增量的写入命令备份，RDB 文件属于全量的数据备份，所以更新频率比 RDB 文件的更新频率高。所以如果 Redis 服务器开启了 AOF 持久化功能，那么服务器会优先使用 AOF 文件来还原数据库状态；只有在 AOF 的持久化功能处于关闭状态时，服务器才会使用优先使用 RDB 文件还原数据库状态。

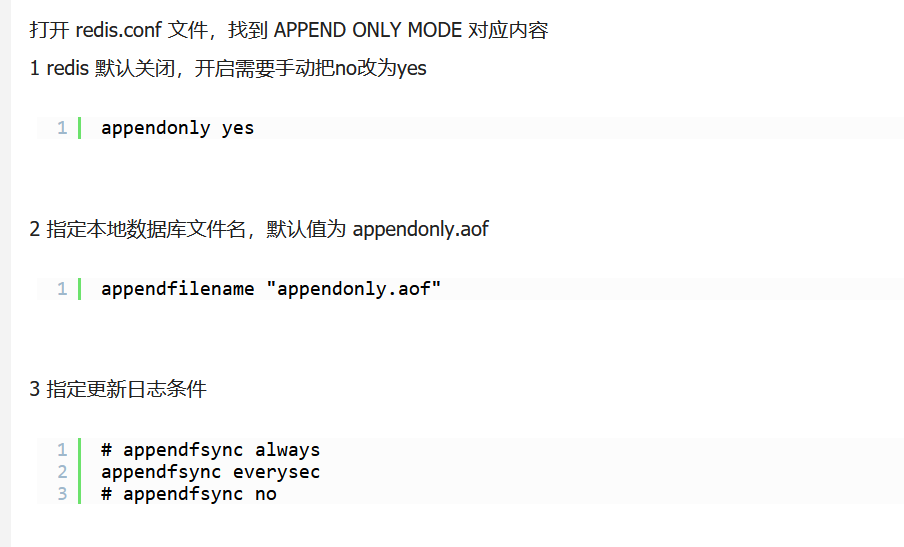


1. SAVE 命令和 BGSAVE 命令都只会备份当前数据库，备份文件名默认为 dump.rdb，可通过配置文件修改备份文件名 dbfilename xxx.rdb。可以通过以下命令查看备份文件目录和 RDB 文件名称：





1. **AOF持久化**
2. Redis 默认不开启。它的出现是为了弥补RDB的不足（数据的不一致性），所以它采用日志的形式来记录每个写操作，并追加到文件中。Redis 重启的会根据日志文件的内容将写指令从前到后执行一次以完成数据的恢复工作。
3. 配置文件：

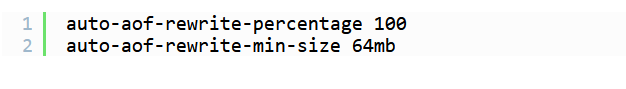


always：同步持久化，每次发生数据变化会立刻写入到磁盘中。性能较差当数据完整性比较好（慢，安全）

everysec：出厂默认推荐，每秒异步记录一次（默认值）

no：不同步

1. 正常情况下，将appendonly.aof 文件拷贝到redis的安装目录的bin目录下，重启redis服务即可。但在实际开发中，可能因为某些原因导致appendonly.aof 文件格式异常，从而导致数据还原失败，可以通过命令redis-check-aof --fix appendonly.aof 进行修复 。
2. AOF的工作原理是将写操作追加到文件中，文件的冗余内容会越来越多。所以聪明的 Redis 新增了重写机制。当AOF文件的大小超过所设定的阈值时，Redis就会对AOF文件的内容压缩。
3. 重写的原理：Redis 会fork出一条新进程，读取内存中的数据，并重新写到一个临时文件中，最后替换旧的aof文件。
4. 触发机制：当AOF文件大小是上次rewrite后大小的一倍且文件大于64M时触发。这里的“一倍”和“64M” 可以通过配置文件修改。



1. **AOF的优缺点：优点->数据的完整性和一致性高；缺点->**因为AOF记录的内容多，文件会越来越大，数据恢复也会越来越慢。
2. **Redis清除策略：redis对于过期键有三种清除策略**
3. 被动删除：只有key被操作时(如GET)，REDIS才会被动检查该key是否过期，如果过期则删除之并且返回NIL。
4. 这种删除策略对CPU是友好的，删除操作只有在不得不的情况下才会进行，不会对其他的expire key上浪费无谓的CPU时间。
5. 这种策略对内存不友好，一个key已经过期，但是在它被操作之前不会被删除，仍然占据内存空间。如果有大量的过期键存在但是又很少被访问到，那会造成大量的内存空间浪费。可能存在一些key永远不会被再次访问到，这些设置了过期时间的key也是需要在过期后被删除的，甚至可以将这种情况看作是一种内存泄露—-无用的垃圾数据占用了大量的内存，而服务器却不会自己去释放它们，这对于运行状态非常依赖于内存的Redis服务器来说，肯定不是一个好消息。
6. 主动删除：Redis会周期性的随机测试一批设置了过期时间的key并进行处理，测试到的已过期的key将被删除。
7. 典型的方式为,Redis每秒做10次如下的步骤：随机测试100个设置了过期时间的key；删除所有发现的已过期的key；若删除的key超过25个则重复步骤1。
8. Redis对每次淘汰任务执行的最大时长也有一个限定，这样保证了每次主动淘汰不会过多阻塞应用请求，以下是这个限定计算公式（单位毫秒）：

timelimit = 1000000\*ACTIVE\_EXPIRE\_CYCLE\_SLOW\_TIME\_PERC/server.hz/100;

1. 当REDIS运行在主从模式时，只有主结点才会执行上述这两种过期删除策略，然后把删除操作”del key”同步到从结点。
2. 内存限制删除（**maxmemory**）：当前已用内存超过maxmemory限定时，对于所有的读写请求，都会触发主动清理策略。
3. 注意这个清理过程是阻塞的，直到清理出足够的内存空间。所以如果在达到maxmemory并且调用方还在不断写入的情况下，可能会反复触发主动清理策略，导致请求会有一定的延迟。
4. 通过maxmemory\_policy确定redis是否释放内存以及如何释放内存。redis提供了8种内存超过限制之后的响应措施，分别如下：

--volatile-lru(least recently used):最近最少使用算法，从设置了过期时间的键中选择空转时间最长的键值对清除掉；

--volatile-lfu(least frequently used):最近最不经常使用算法，从设置了过期时间的键中选择某段时间之内使用频次最小的键值对清除掉；

--volatile-ttl:从设置了过期时间的键中选择过期时间最早的键值对清除；

--volatile-random:从设置了过期时间的键中，随机选择键进行清除；

--allkeys-lru:最近最少使用算法，从所有的键中选择空转时间最长的键值对清除；

--allkeys-lfu:最近最不经常使用算法，从所有的键中选择某段时间之内使用频次最少的键值对清除；

--allkeys-random:所有的键中，随机选择键进行删除；

--noeviction:不做任何的清理工作，在redis的内存超过限制之后，所有的写入操作都会返回错误；但是读操作都能正常的进行;

1. volatile-前缀的策略代表从redisDb中的expire字典中选择键进行清除；allkeys-开头的策略代表从dict字典中选择键进行清除。
2. 尽量不要触发maxmemory，最好在mem\_used内存占用达到maxmemory的一定比例后，需要考虑调大hz以加快淘汰，或者进行集群扩容。
3. 如果能够控制住内存，则可以不用修改maxmemory-samples配置；如果Redis本身就作为LRU cache服务（这种服务一般长时间处于maxmemory状态，由Redis自动做LRU淘汰），可以适当调大maxmemory-samples。
4. **缓存穿透、缓存雪崩、缓存击穿案例分析**
5. **缓存穿透**：缓存穿透是指查询一个一定不存在的数据，由于缓存是不命中时被动写的，并且出于容错考虑，如果从存储层查不到数据则不写入缓存，这将导致这个不存在的数据每次请求都要到存储层去查询，失去了缓存的意义。在流量大时，可能DB就挂掉了，要是有人利用不存在的key频繁攻击我们的应用，这就是漏洞。

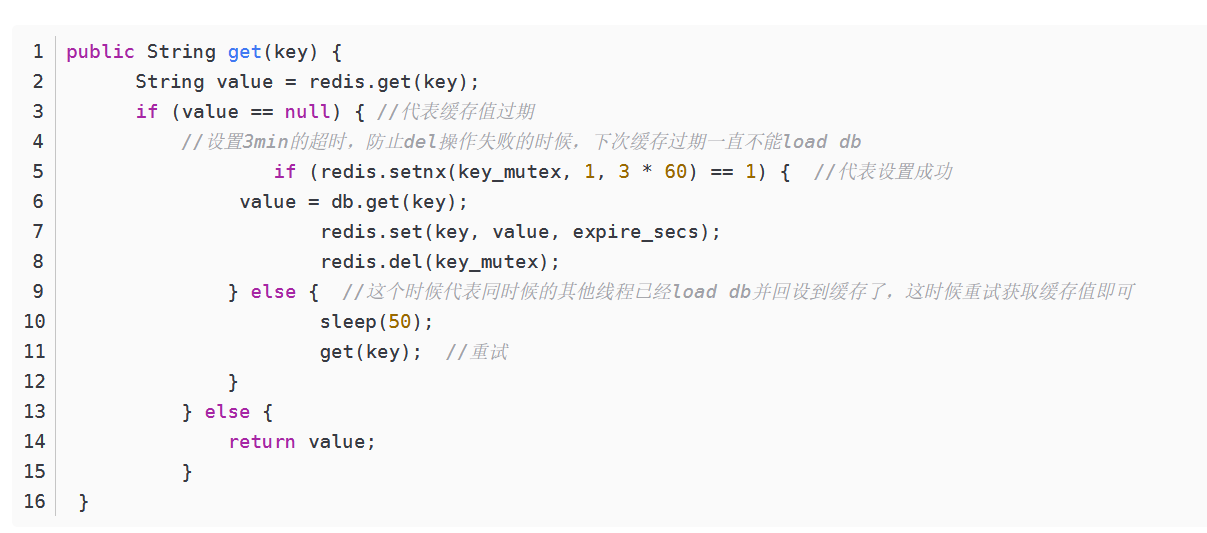
**解决方案**：有很多种方法可以有效地解决缓存穿透问题，最常见的则是采用布隆过滤器，将所有可能存在的数据哈希到一个足够大的bitmap中，一个一定不存在的数据会被 这个bitmap拦截掉，从而避免了对底层存储系统的查询压力。另外也有一个更为简单粗暴的方法（我们采用的就是这种），如果一个查询返回的数据为空（不管是数 据不存在，还是系统故障），我们仍然把这个空结果进行缓存，但它的过期时间会很短，最长不超过五分钟。

1. **缓存雪崩**：缓存雪崩是指在我们设置缓存时采用了相同的过期时间，导致缓存在某一时刻同时失效，请求全部转发到DB，DB瞬时压力过重雪崩。

**解决方案**：缓存失效时的雪崩效应对底层系统的冲击非常可怕。大多数系统设计者考虑用加锁或者队列的方式保证缓存的单线 程（进程）写，从而避免失效时大量的并发请求落到底层存储系统上。这里分享一个简单方案就是将缓存失效时间分散开，比如我们可以在原有的失效时间基础上增加一个随机值，比如1-5分钟随机，这样每一个缓存的过期时间的重复率就会降低，就很难引发集体失效的事件。

1. **缓存击穿**：对于一些设置了过期时间的key，如果这些key可能会在某些时间点被超高并发地访问，是一种非常“热点”的数据。这个时候，需要考虑一个问题：缓存被“击穿”的问题，这个和缓存雪崩的区别在于这里针对某一key缓存，前者则是很多key。缓存在某个时间点过期的时候，恰好在这个时间点对这个Key有大量的并发请求过来，这些请求发现缓存过期一般都会从后端DB加载数据并回设到缓存，这个时候大并发的请求可能会瞬间把后端DB压垮。

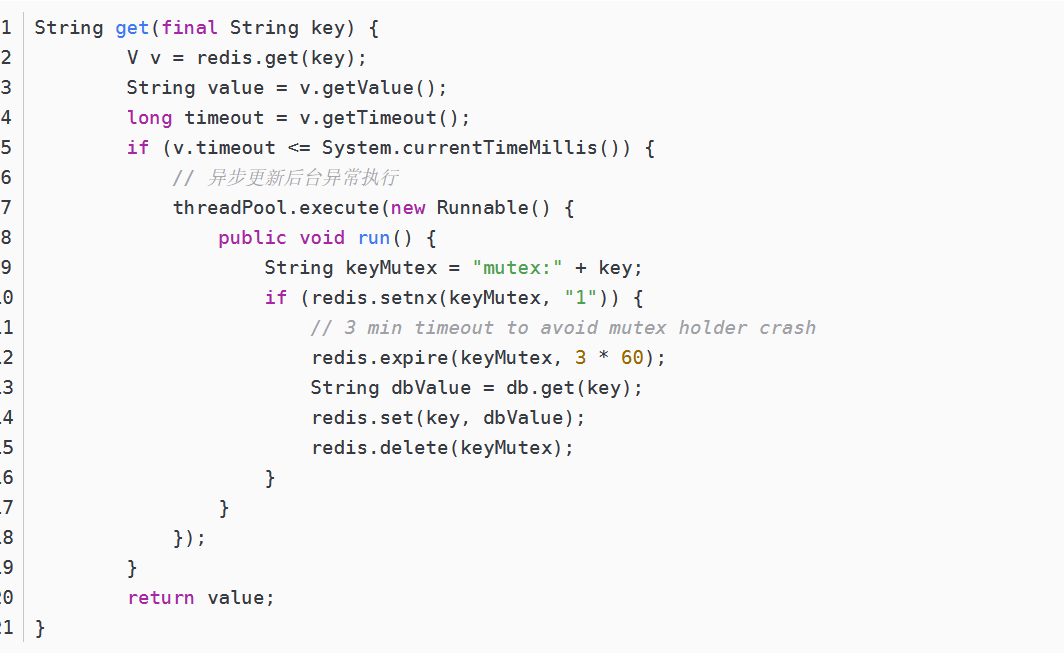
**解决方案1（使用互斥锁）**：业界比较常用的做法，是使用mutex。简单地来说，就是在缓存失效的时候（判断拿出来的值为空），不是立即去load db，而是先使用缓存工具的某些带成功操作返回值的操作（比如Redis的SETNX或者Memcache的ADD）去set一个mutex key，当操作返回成功时，再进行load db的操作并回设缓存；否则，就重试整个get缓存的方法。



**解决方案2（提前使用互斥锁）**：在value内部设置1个超时值(timeout1), timeout1比实际的memcache timeout(timeout2)小。当从cache读取到timeout1发现它已经过期时候，马上延长timeout1并重新设置到cache。然后再从数据库加载数据并设置到cache中。

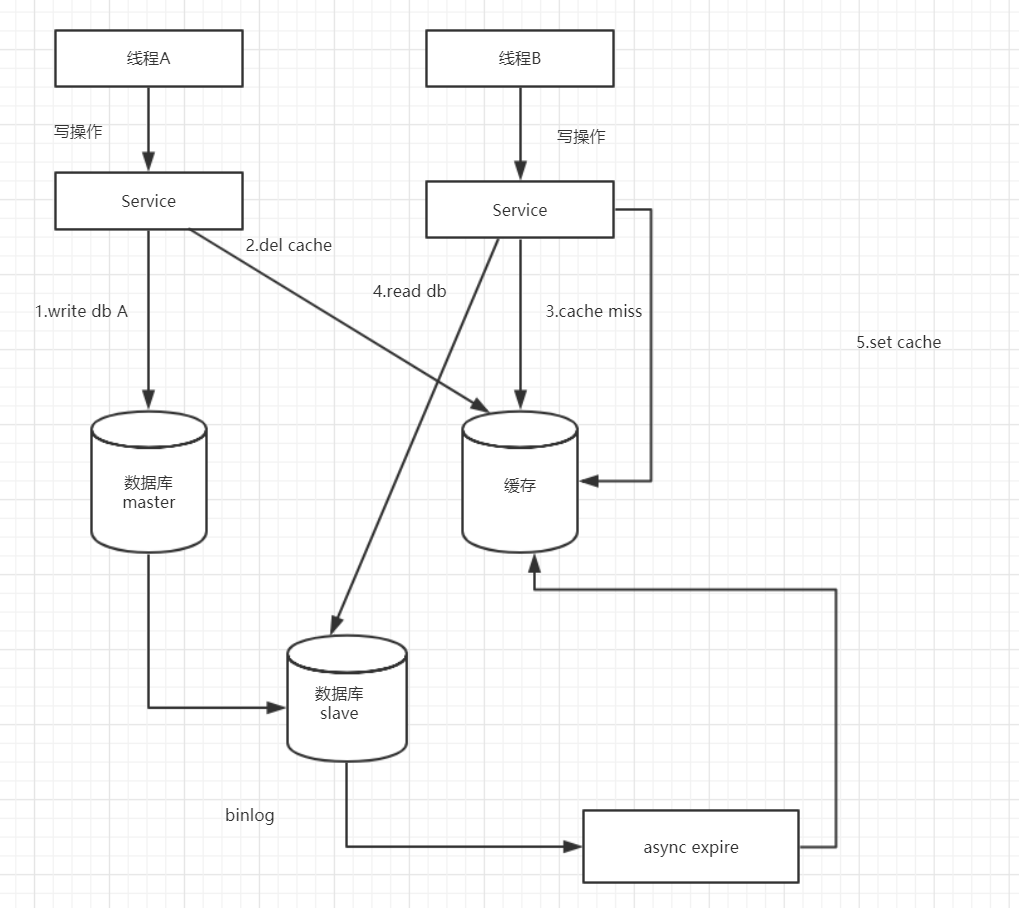


**解决方案3（永不过期）:** (1) 从redis上看，确实没有设置过期时间，这就保证了，不会出现热点key过期问题，也就是“物理”不过期。(2) 从功能上看，如果不过期，那不就成静态的了吗？所以我们把过期时间存在key对应的value里，如果发现要过期了，通过一个后台的异步线程进行缓存的构建，也就是“逻辑”过期。从实战看，这种方法对于性能非常友好，唯一不足的就是构建缓存时候，其余线程(非构建缓存的线程)可能访问的是老数据，但是对于一般的互联网功能来说这个还是可以忍受。





1. **并发环境下缓存的一致性：**
2. 未读写分离的数据库：先更新数据在，清除缓存，清除缓存失败，事务回滚（比较暴力）。
3. 读写分离的数据库：使用binlog异步删除，如果是主从库，binlog取自从库；如果是一主多从，消费端取到最后一台的binlog后再删除



1. 地点