缓存利器-Redis

目录

分享人：张国栋 20230602

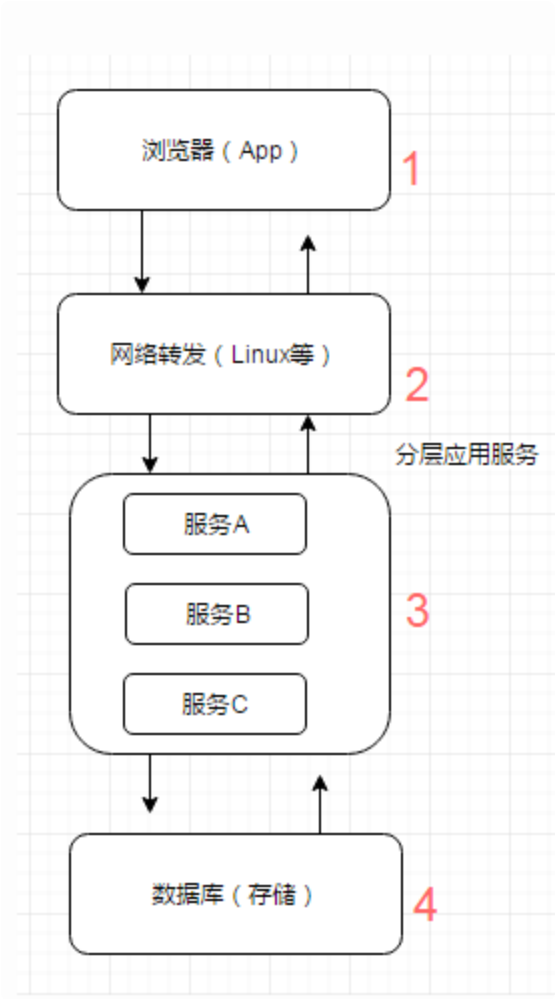
Redis

The open source, in-memory data store used by millions of developers as a database, cache, streaming engine, and message broker.

目录

1.什么是缓存

互联网应用一般流程



缓存使用范围广泛，可以在任何一个步骤使用；

缓存是一个抽象的概念，它有一些特征和关键指标：命中率；最大空间；过期策略；缓存介质

1.1 缓存常见使用目的

1.2 常见缓存方案

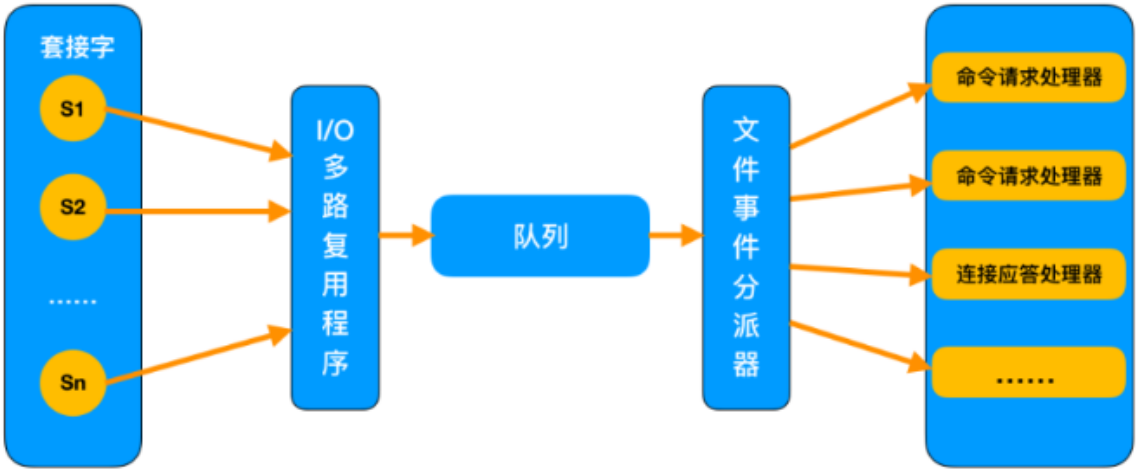
本地缓存：Guava Cache; Spring Cache 自定义使用HashMap作为缓存

分布式缓存： Redis; MemCache

比较项	GuavaCache	MemCache	Redis
支持分布式	否	半支持	支持水平拓展
是否支持持久化	否	否	支持（RDB和AOF）
缓存字段大小限制	一般无限制	一般key为250KB，value为1M	可到1GB
支持的数据类型	简单的key-value	简单的key-value	支持String，HashMap，List，Set，SortedSet，BitMap等丰富基础类型
过期策略	惰性删除和定期删除结合	惰性删除	惰性删除和定期删除结合
易用性	易用	易用	易用
多语言客户端	java	支持多种语言	支持多种语言
延时	亚毫秒级别	亚毫秒级别	亚毫秒级别

2.Redis优秀的底层设计

2.1 redis单线程处理请求

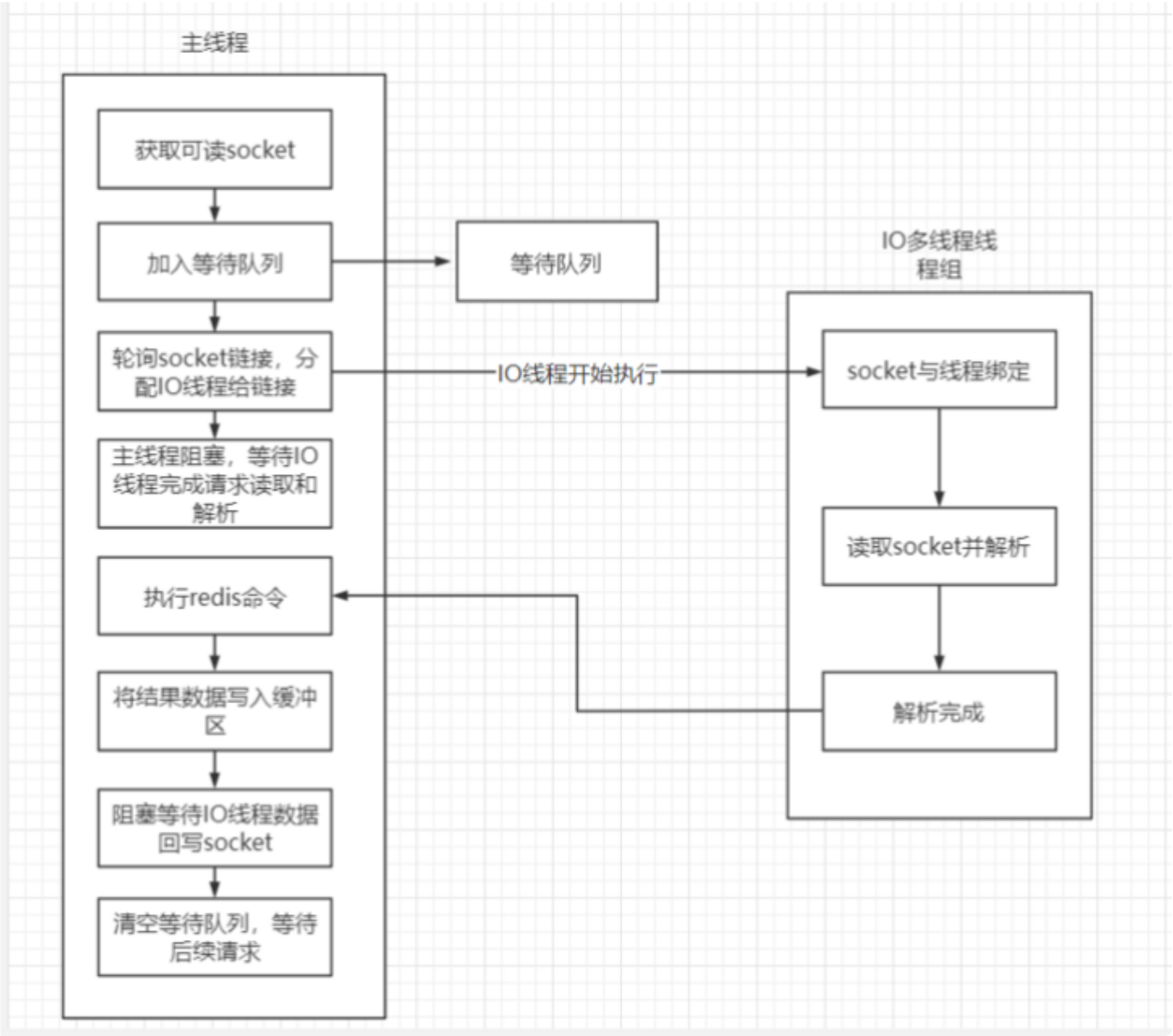


整个请求在一个线程里完成，避免处理多线程问题，redis性能瓶颈在内存，不在cpu，所以单线程已经满足

前提：Redis6.0以前版本

Redis6.0以后, 采用多个 IO 线程来处理网络请求, 后续的读写命令执行仍然是单线程

多线程能解决key或者value很大的情况导致的计算资源紧张



2.2 高效存储的数据类型

redis使用RedisObject对象来存储所有的key-value

数据类型	编码方式	转换规则
REDIS_STRING	REDIS_ENCODING_INT	如果长度小于20并且可以value可以转换为整形
	REDIS_ENCODING_EMBSTR	如果长度小于44，使用此编码
	REDIS_ENCODING_RAW	默认编码
REDIS_LIST	OBJ_ENCODING_QUICKLIST	默认编码，6.0.0版本无其他编码
REDIS_HASH	REDIS_ENCODING_ZIPLIST	初始化默认编码
	REDIS_ENCODING_HT	当元素超过配置：hash-max-ziplist-entries时使用此编码，默认值是512
REDIS_SET	REDIS_ENCODING_INTSET	如果member可以转换为整形，则使用REDIS_ENCODING_INTSET编码，否则使用REDIS_ENCODING_HT编码
	REDIS_ENCODING_HT	
REDIS_ZSET	REDIS_ENCODING_ZIPLIST	如果 zset-max-ziplist-entries 配置为0并且，元素长度小于 zset-max-ziplist-value 配置(默认64)，则使用REDIS_ENCODING_SKIPLIST编码，否则使用REDIS_ENCODING_ZIPLIST编码
	REDIS_ENCODING_SKIPLIST	

String：简单动态字符串的应用，1.单独记录了字符串长度，获取字符串长度时间复杂度o(1)，并保证了字符串的二进制安全，解决了c语言中的缓存溢出问题 2.预分配内存，惰性释放内存，减少内存碎片化，提升响应时间

其他数据类型：BiteMap，HyperLogLog ，GEO

2.3 内存淘汰策略和内存过期删除策略

过期删除策略

用户主动设置过期时间，到期后自动删除

策略	流程	优缺点

定时删除	设置过期字典，设置定期删除事件到时间点后直接删除key `typedef struct redisDb { dict *dict; /* 数据库键空间，存放着所有的键值对 */ dict *expires; /* 键的过期时间 */ .... } redisDb;`	优点：对内存友好 缺点：对cpu不友好
惰性删除	在过期时间到达之后，再去访问该key，才启动删除key	优点：对cpu友好 缺点：对内存不友好
定期删除	每隔一段时间「随机」从数据库中取出一定数量的 key 进行检查，并删除其中的过期key	优点：对内存和cpu有适中处理 缺点：抽取数量不好确定，难以确定执行时长和频率

Redis 选择「惰性删除+定期删除」这两种策略配和使用

其中定期删除是一个动态策略，依赖预上一次执行删除的数量比例确定要不要循环进行定期删除

内存淘汰策略

内存使用到达redis可用的最大内存后，会启动淘汰策略删除符合条件的key

8种策略

策略	说明
noeviction	它表示当运行内存超过最大设置内存时，不淘汰任何数据，这时如果有新的数据写入，则会触发 OOM，但是如果没用数据写入的话，只是单纯的查询或者删除操作的话，还是可以正常工作。
volatile-random	随机淘汰设置了过期时间的任意键值
volatile-ttl	优先淘汰更早过期的键值。
volatile-lru	淘汰所有设置了过期时间的键值中，最久未使用的键值；

volatile-lfu	淘汰所有设置了过期时间的键值中，最少使用的键值；
allkeys-random	随机淘汰任意键值;
allkeys-lru	淘汰整个键值中最久未使用的键值；
allkeys-lfu	淘汰整个键值中最少使用的键值。

Least Recently Used 和 Least Frequently Used

2.4 持久化

持久化策略	**说明**	优缺点
RDB	bgsave命令隔一段时间会生成全量内存快照，存储为二进制文件	优点：文件体积小，回复速度快 缺点：在复制快照期间可能会丢失当时正在修改的数据
AOF	将执行更新的redis命令记录存储到文件，会涉及刷盘机制； 刷盘策略：Always， Everysec，No	优点：选择合适的刷盘机制保证数据不丢失 缺点：文件可能膨胀到比较大，根据命令行恢复数据较慢

redis较好的策略是混合使用RDB和AOF： aof-use-rdb-preamble ： yes

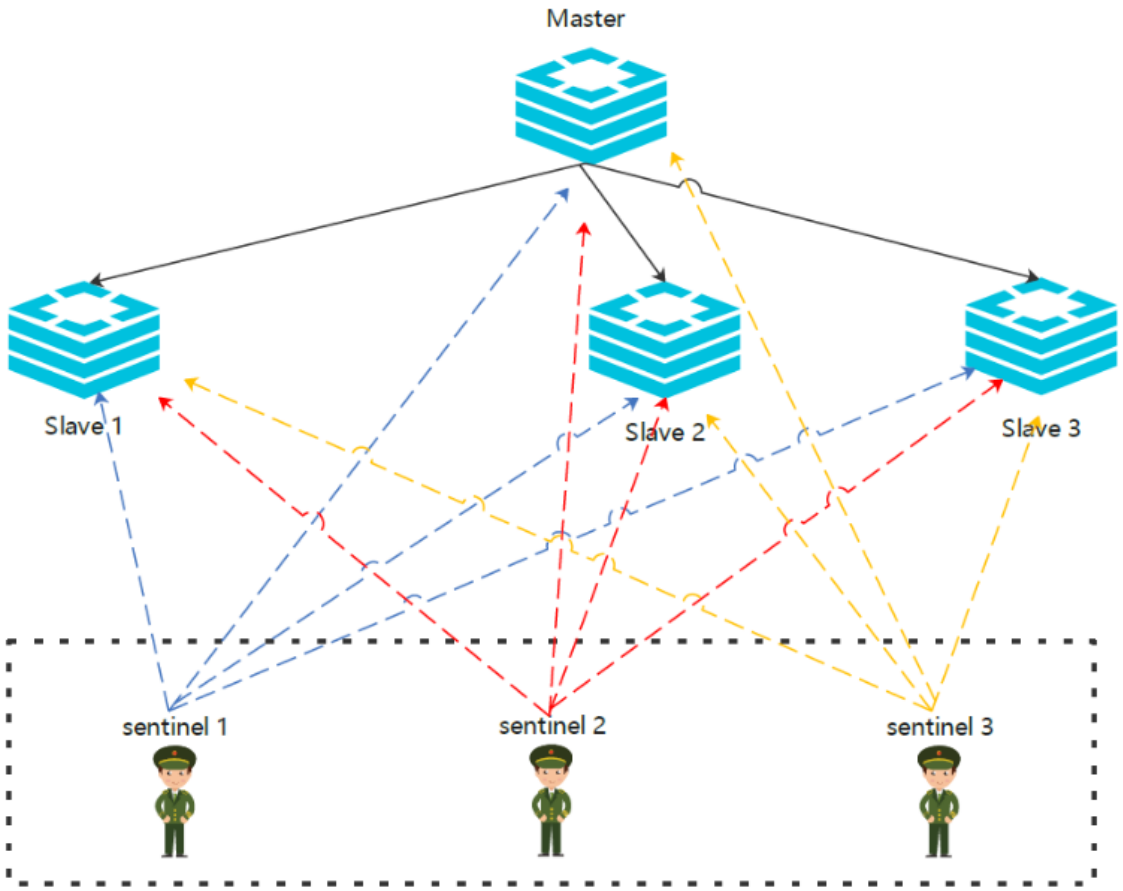
通过RDB恢复数据后，再通过近期的AOF进行调整，达到更少地丢失数据

2.5 高可用部署方案

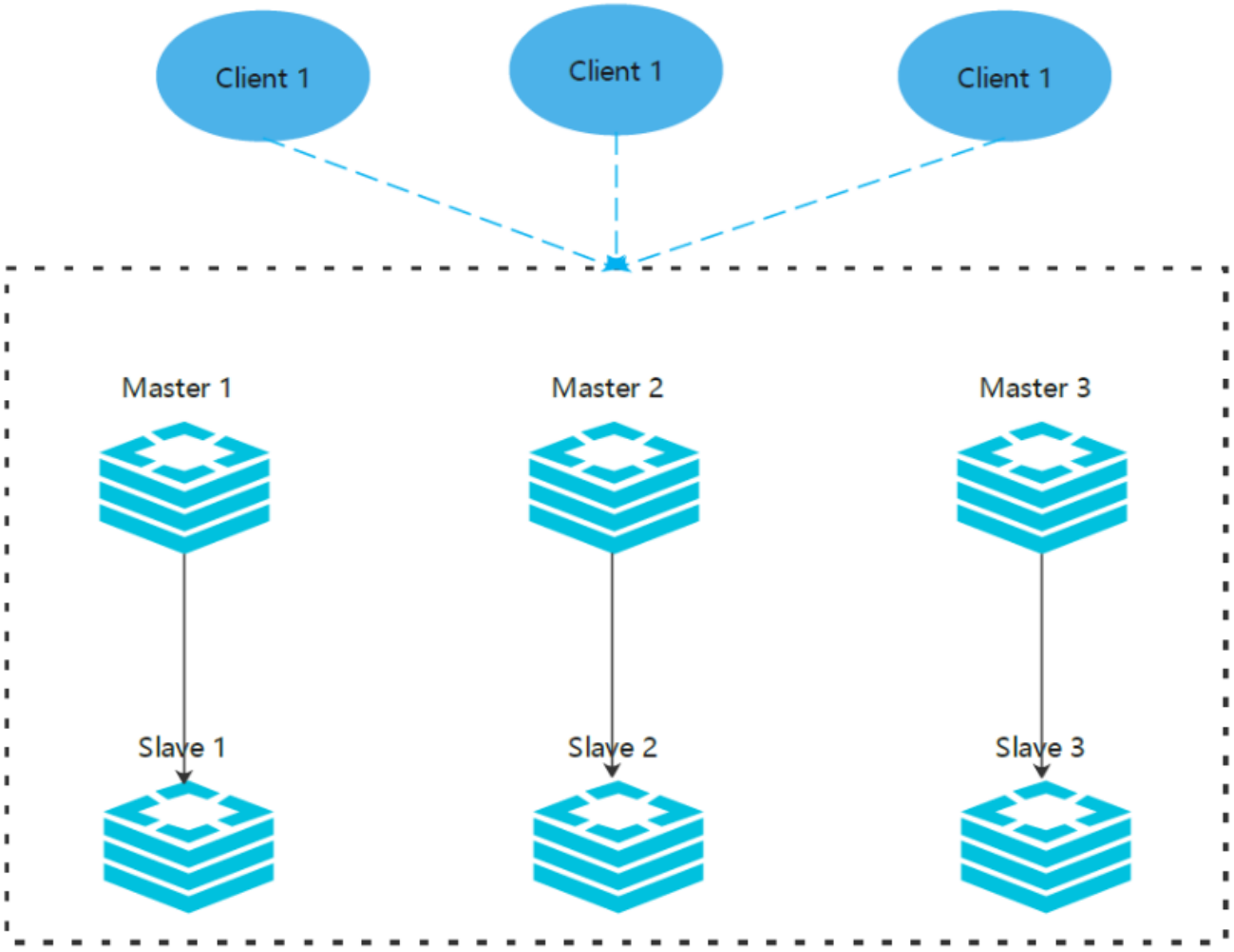
方案	说明	解决问题	遗留问题
单机	一般不用于生成环境		

主从	和mysql类似，主节点向从节点同步数据	解决单点故障问题	不具备自动容错和恢复功能； 在线扩容难
主从+哨兵	部署多个哨兵和redis，哨兵向所有redis发生检测命令，发现redis节点异常后自动踢掉节点和选举主节点	可以自动容错切换	还是存在扩容难问题； 内存使用率也不高
cluster	使用slots 插槽，将key存储在不同的哈希槽，每个节点分到一部分哈希槽	内存使用率高于主从模式 可用性高	Cluster 是无中心节点的集群架构，依靠 Goss 协议(谣言传播)协同自动化修复集群的状态

哨兵模式



集群模式



3.常见的Redis缓存使用问题

缓存雪崩，缓存击穿，缓存穿透，缓存与数据库不一致问题

问题	原因	解决方案
缓存雪崩	大量缓存同时失效或者redis宕机	1.均匀设置过期时间 2.设置互斥锁去读取数据库 3.缓存不设置过期时间，通过后台去更新或者删除缓存 3.采用哨兵模式或者集群模式的redis架构

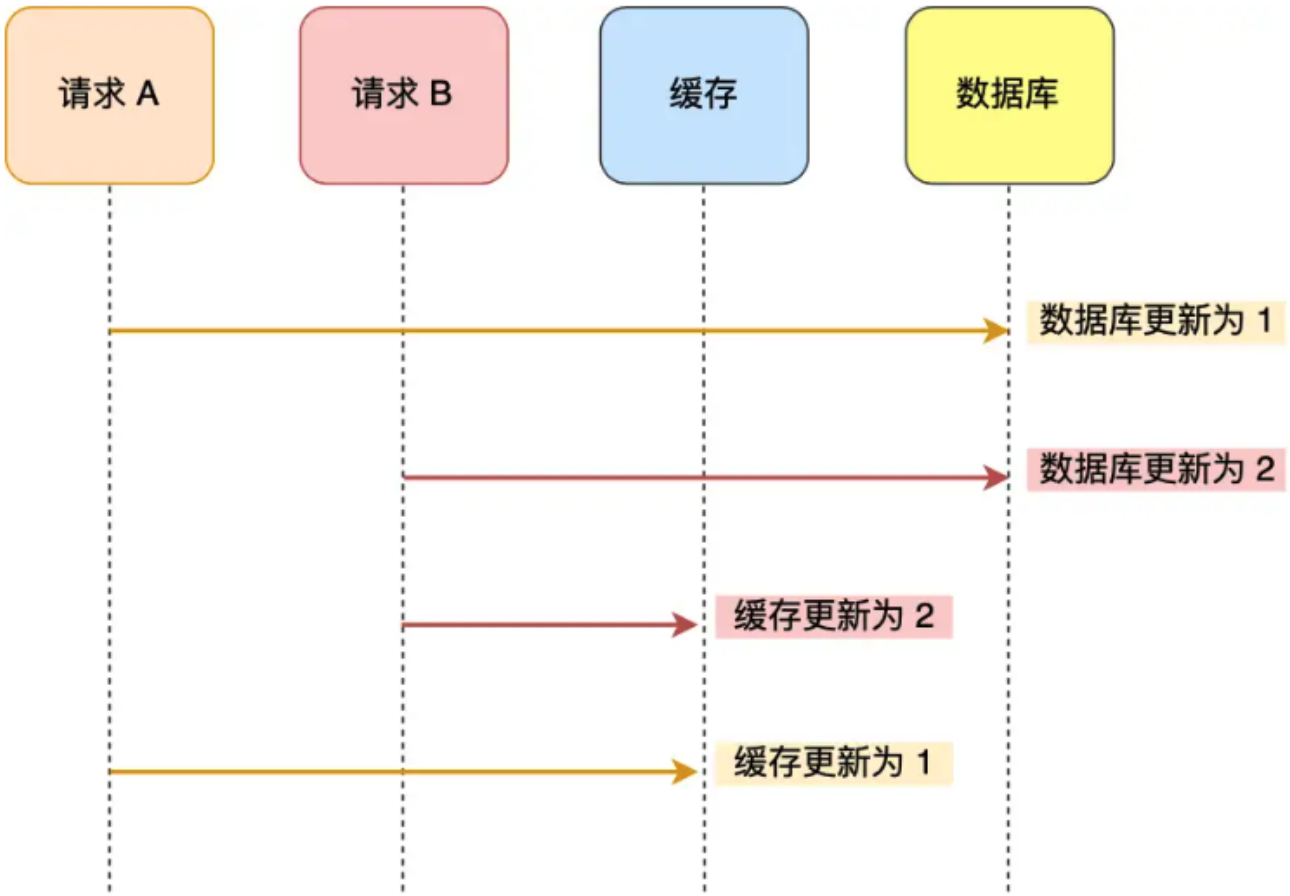
缓存击穿	热点缓存数据突然过期失效，导致大量请求直接到数据库层	1.互斥锁方案 2.不设置过期时间，采用后台程序进行缓存更新
缓存穿透	大量请求命中缓存和数据库中不存在的数据，一般是业务误操作或者黑客攻击	1.缓存空值或者默认值 2.布隆过滤器（可以对不存在的数据快速判定）
缓存和DB不一致问题	数据库更新后，由于程序异常中断或者多线程竞争原因导致缓存还是历史数据	数据更新的时候设计合理的数据更新和缓存更新策略

什么是合理的数据库更新和缓存更新策略?

1.数据库更新

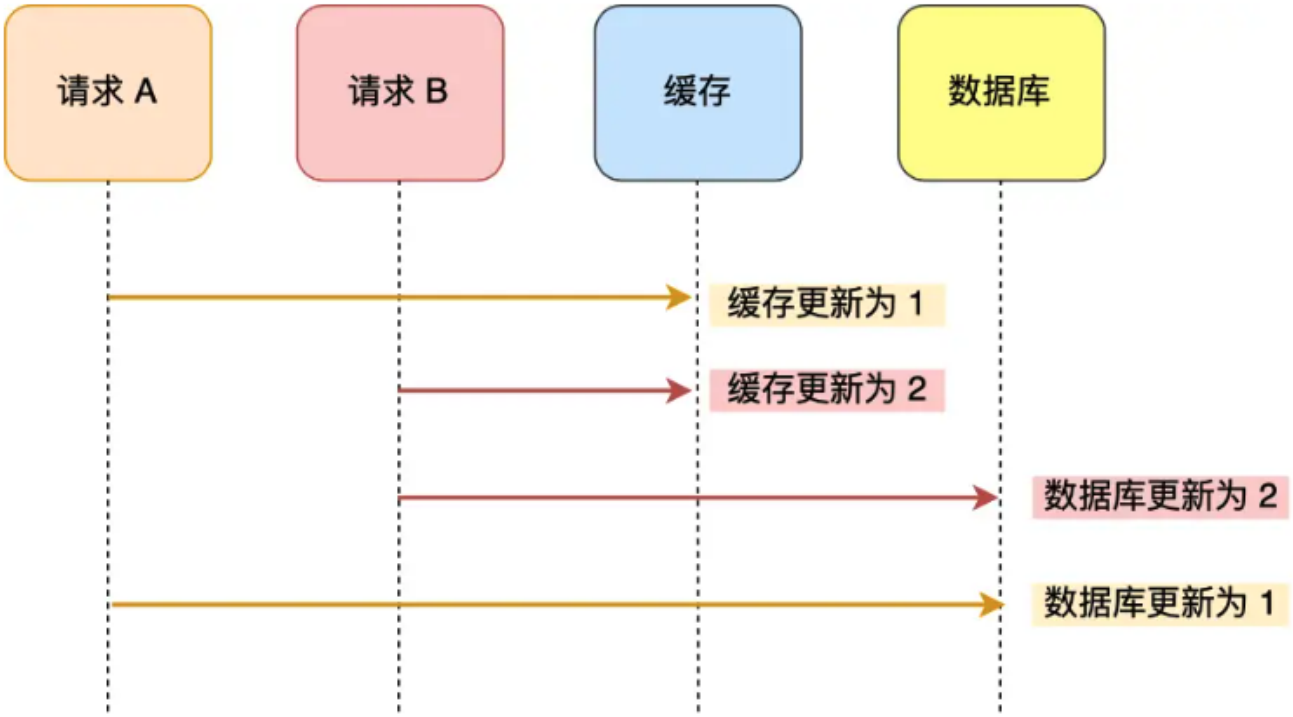
2.缓存更新

第一种方案：先更新数据库，再更新缓存



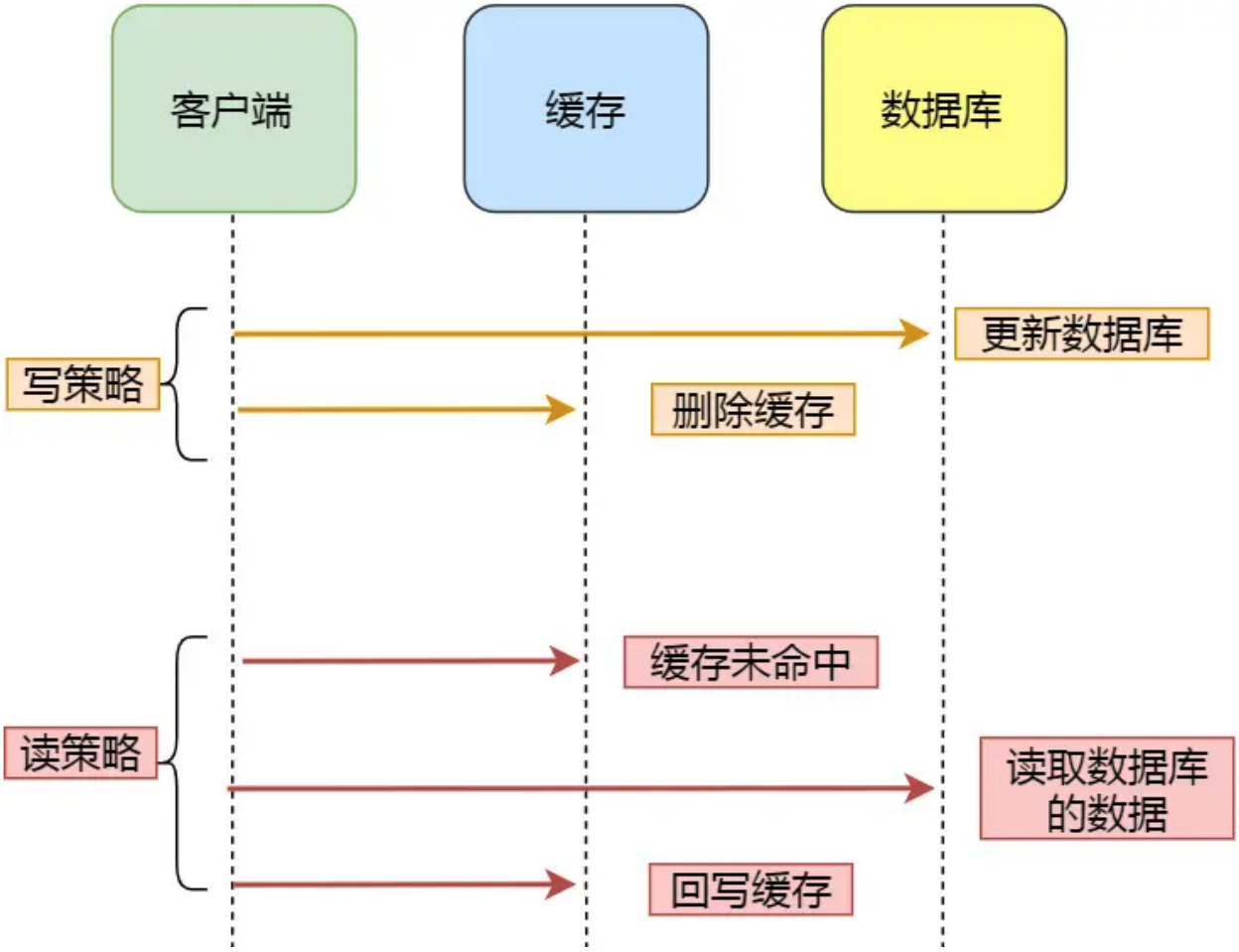
出现不一致

第二种方案：先更新缓存，再更新数据库

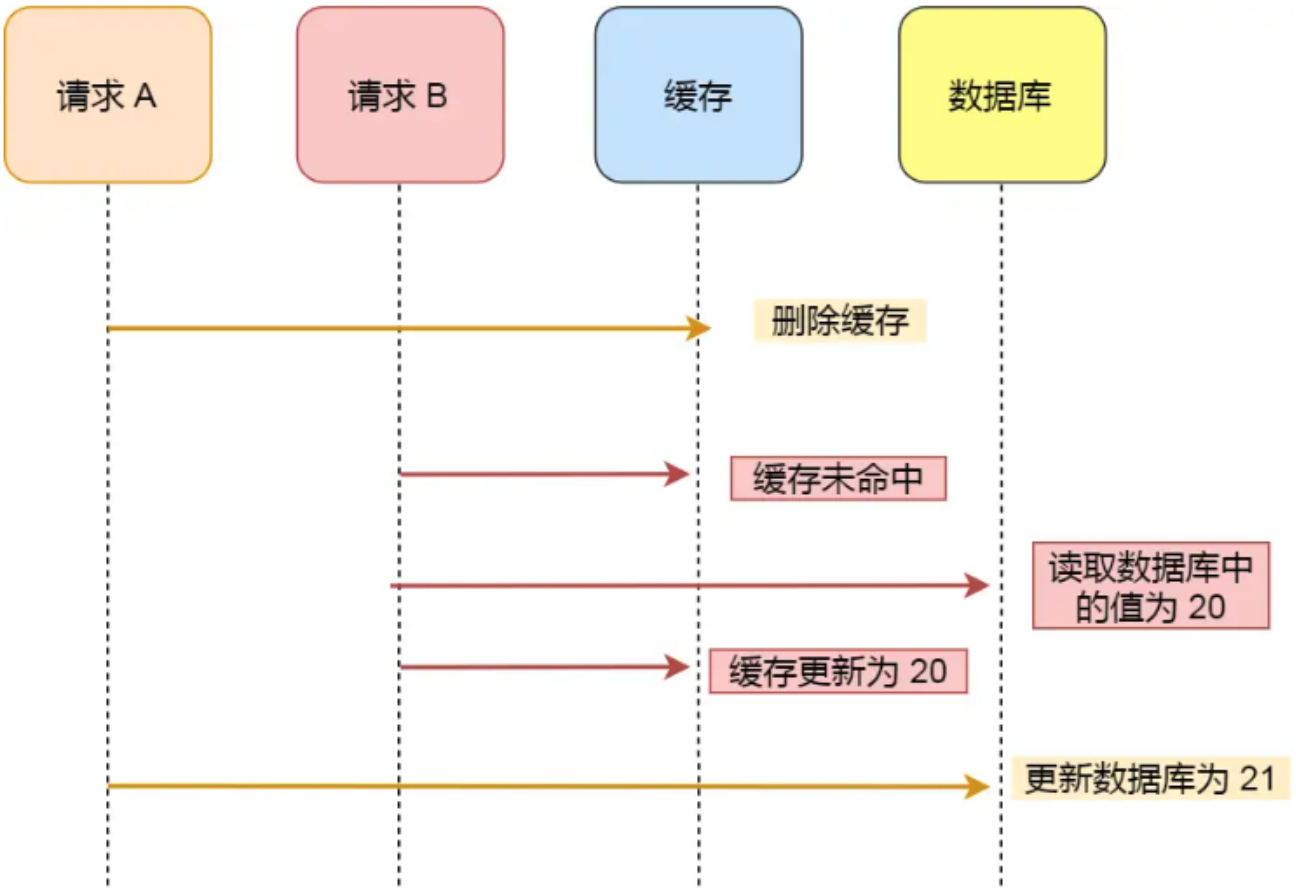


出现不一致

第三种方案：**Cache Aside 旁路缓存**

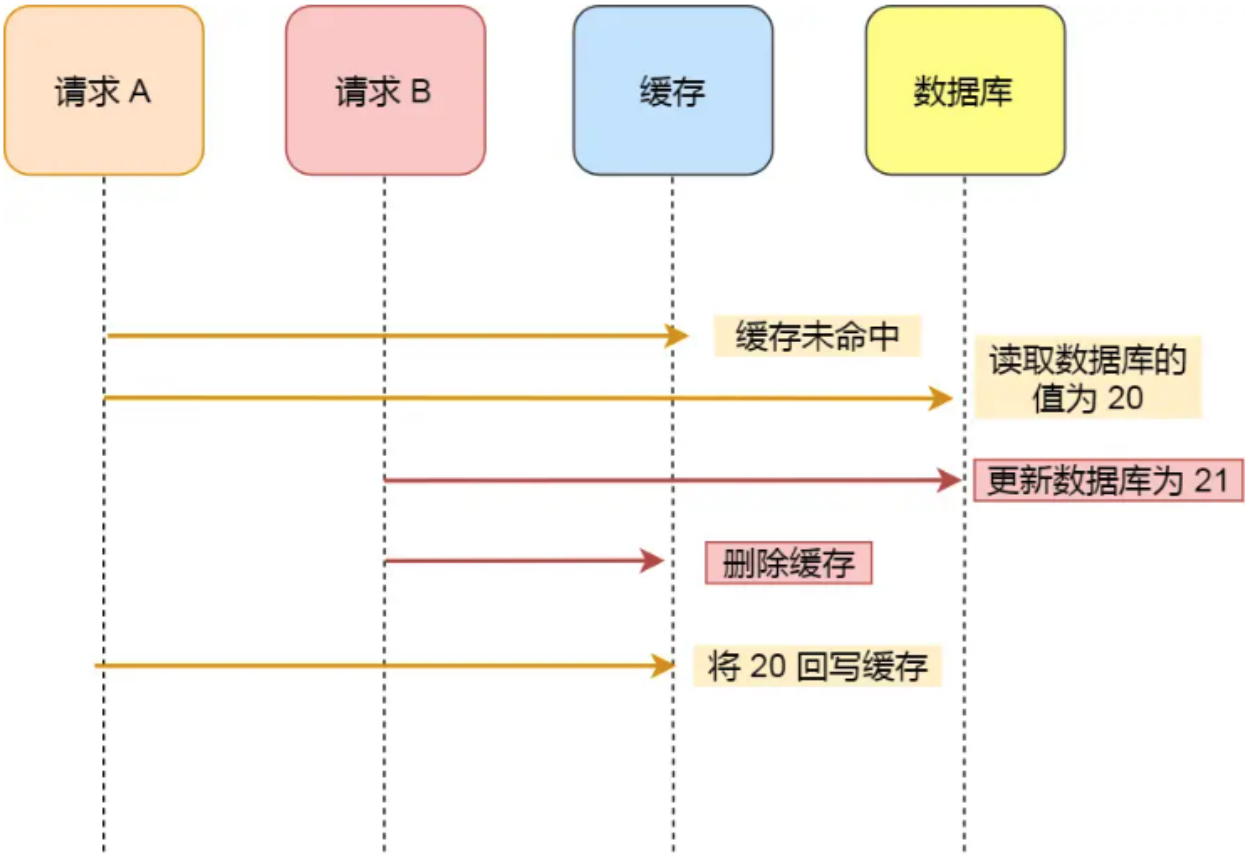


先删除缓存，再更新数据库



产生了不一致脏数据

先更新数据库，再删除缓存



也可能会出现数据不一致问题，但是出现不一致的情况是A请求的写缓存比B的更新数据库和删除缓存都要慢，现实情况这种概率很难出现

终极办法：最终一致性方案--延迟双删

可以在更新完数据库后，等待一段时间再进行缓存删除，但是会带来了一定的复杂度，等待多久删除也是比较难评估，还得保证第二次删除必须成功

没有最好的办法，只有最符合当前场景的办法！

使用Redis作为缓存常见使用规范：

1.使用的key通常以业务名或者数据库表名作为前缀，冒号分割后加上对应的key

2.保证语义使用的key尽量短

3.value尽量合理设计大小，一般小于10KB，大key的存在可能会导致分片不均、网络阻塞等问题

4.列表等数据中尽量控制数量大小，因为HGETALL等命令时间复杂度与个数有关，频繁执行会带来较大性能消耗

5.禁止使用keys，尽量不要使用hgetall、lrange、smembers、zrange、sinter等操作

4.其他--Redis作为分布式锁

redis的另一大经典使用场景--分布式锁

一般常见方案：

1.自行封装工具类，利用setnx保证加锁的原子性，利用lua脚本保证释放锁的正确性

2.使用Redisson框架，里面封装好了加锁释放锁的api，还有自动续期功能可以使用

3.RedLock方案，解决redis加锁存在的单点问题，是更加严格的分布式锁方案，过于依赖资源，一般场景不使用

5.最后

一本我最喜欢的Redis书籍《Redis深度历险》-钱文品

一场经典辩论：redis作者antirez和Kleppmann

http://antirez.com/news/101

https://martin.kleppmann.com/2016/02/08/how-to-do-distributed-locking.html

中文版：https://juejin.cn/post/6976538149904678925