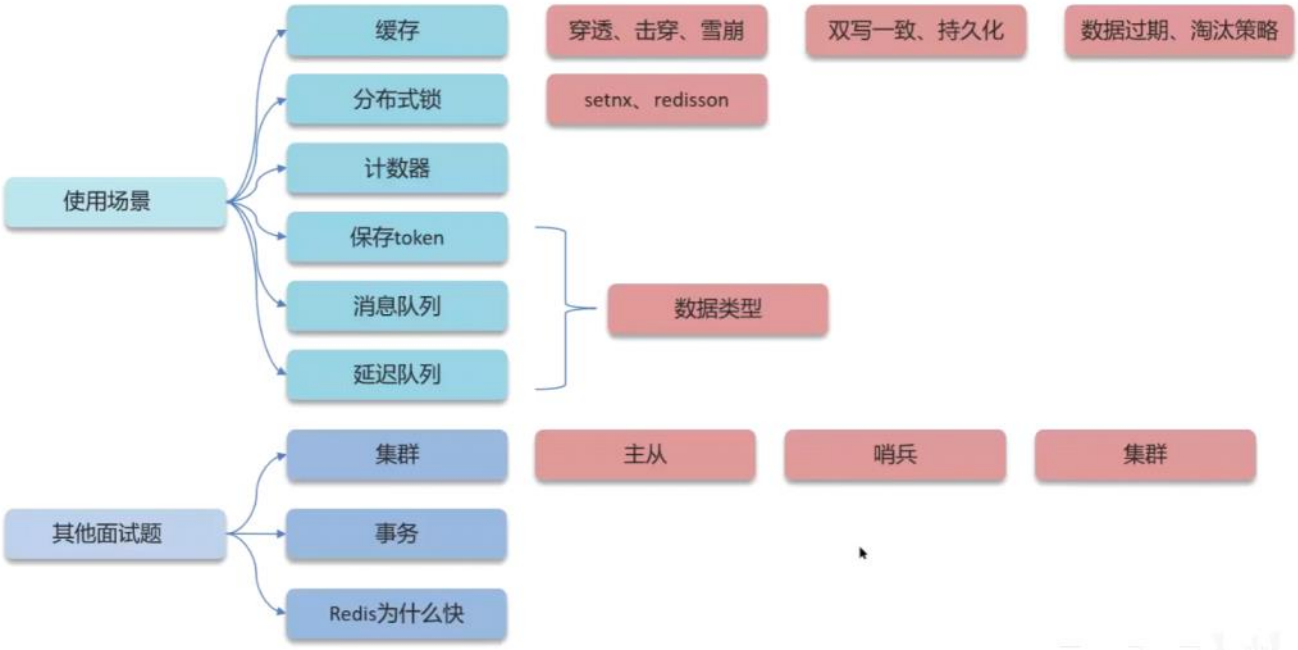
redis

2024年4月11日 12:32



我看你做的项目中，都用到了redis，你在最近的项目中哪些场景使用了redis呢?

结合项目

- 一是验证你的项目场景的真实性，二是为了作为深入发问的切入点
- 缓存　　缓存三兄弟（穿透、击穿、雪崩）、双写一致、持久化、数据过期策略，数据淘汰策略
- 分布式锁　　setnx、redisson
- 消息队列、延迟队列　　何种数据类型
- ... ...

如果发生了缓存穿透、击穿、雪崩，该如何解决?

缓存穿透

例：

一个get请求：api/news/getById/1

缓存穿透：查询一个**不存在**的数据，mysql查询不到数据也不会直接写入缓存，就会导致每次请求都查数据库

解决方案一：缓存空数据，查询返回的数据为空，仍把这个空结果进行缓存　{key:1,value:null}

优点：简单

缺点：消耗内存，可能会发生不一致的问题

缓存穿透

例：

一个get请求：api/news/getById/1

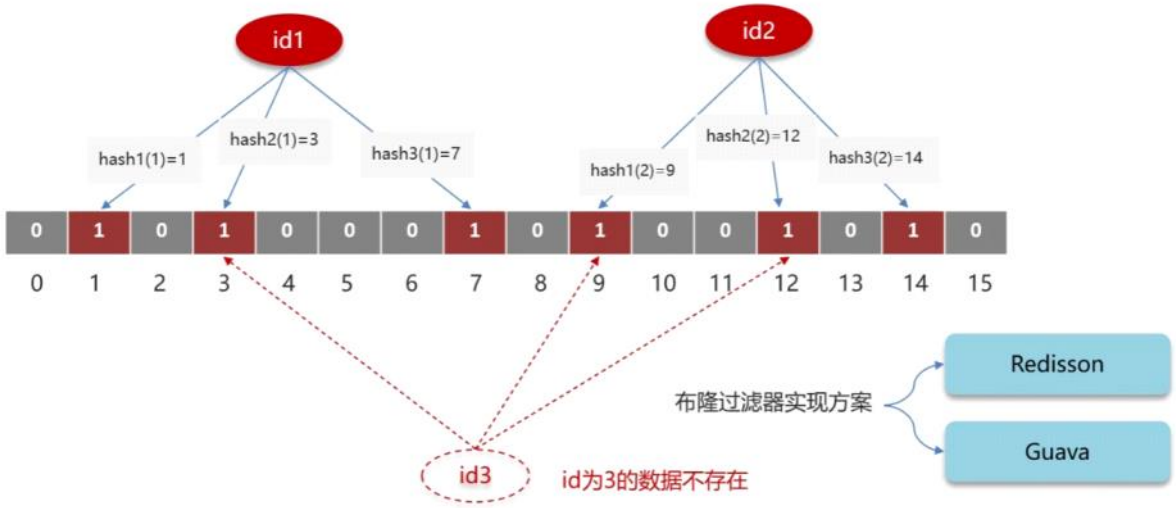
解决方案二：布隆过滤器

布隆过滤器

bitmap（位图）：相当于是一个以（bit）位为单位的数组，数组中每个单元只能存储二进制数0或1

布隆过滤器作用：布隆过滤器可以用于检索一个元素是否在一个集合中。

布隆过滤器



误判率：数组越小误判率就越大，数组越大误判率就越小，但是同时带来了更多的内存消耗。

缓存穿透

例：

一个get请求：api/news/getById/1

解决方案：

方案二：布隆过滤器

优点：内存占用较少，没有多余key

缺点：实现复杂，存在误判

1. Redis的使用场景

- 根据自己简历上的业务进行回答
- 缓存　　穿透、击穿、雪崩、双写一致、持久化、数据过期、淘汰策略
- 分布式锁　setnx、redisson

2. 什么是缓存穿透，怎么解决

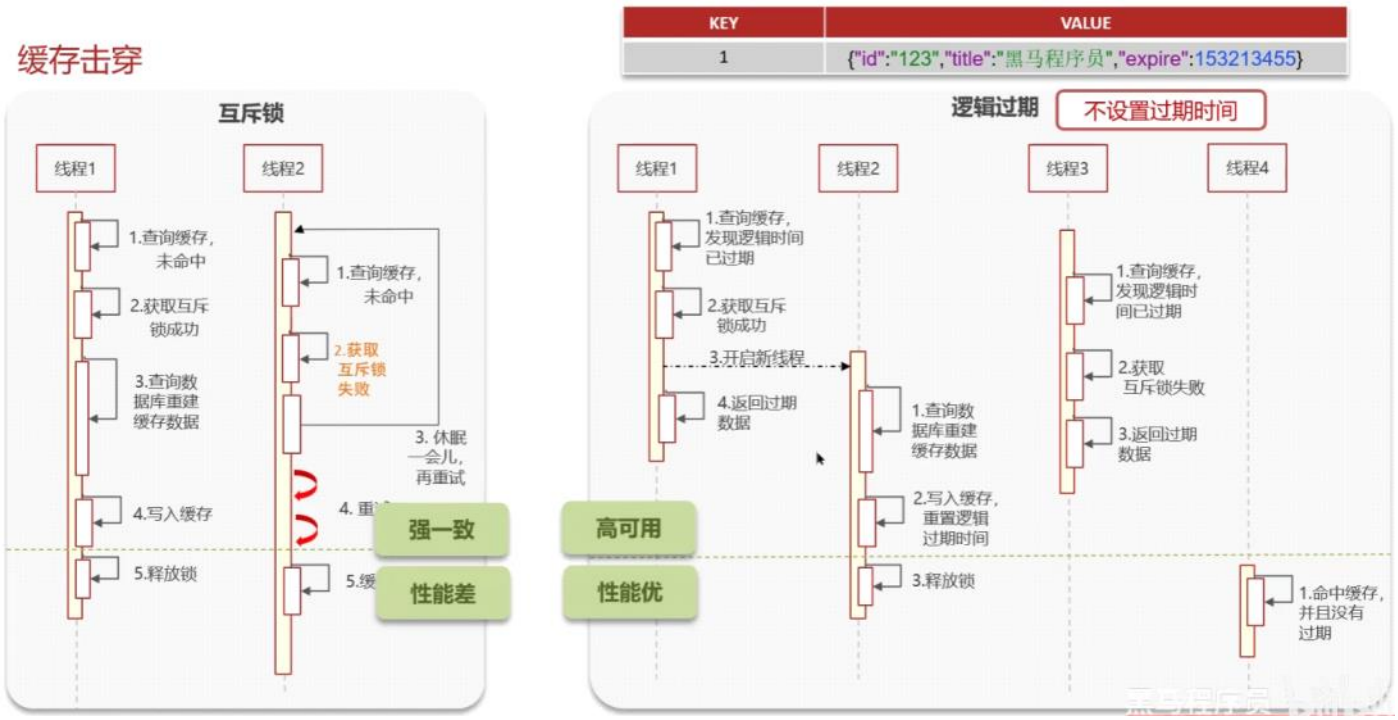
- 缓存穿透：查询一个**不存在**的数据，mysql查询不到数据也不会直接写入缓存，就会导致每次请求都查数据库
- 解决方案一：缓存空数据
- 解决方案二：布隆过滤器

缓存击穿

缓存击穿：给某一个key设置了过期时间，当key过期的时候，恰好这时间点对这个key有大量的并发请求过来，这些并发的请求可能会瞬间把DB压垮

解决方案一：互斥锁

解决方案二：逻辑过期



缓存击穿

- **缓存击穿：**给某一个key设置了过期时间，当key过期的时候，恰好这时间点对这个key有大量的并发请求过来，这些并发的请求可能会瞬间把DB压垮
- **解决方案一**：互斥锁，强一致，性能差
- **解决方案二**：逻辑过期，高可用，性能优，不能保证数据绝对一致

缓存雪崩

缓存雪崩是指在同一时段大量的缓存key同时失效或者Redis服务宕机，导致大量请求到达数据库，带来巨大压力。

解决方案：

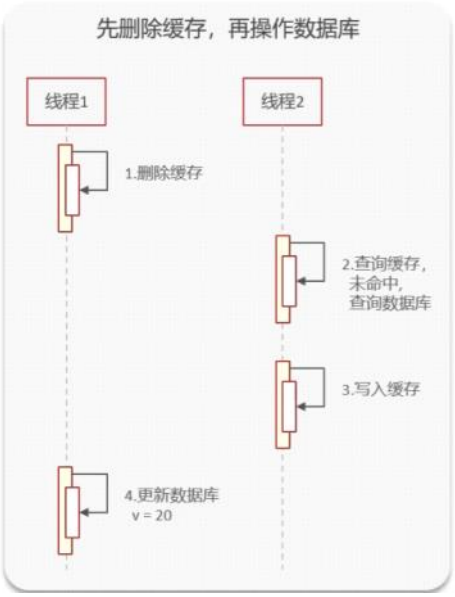
- 给不同的Key的TTL添加随机值
- 利用Redis集群提高服务的可用性　哨兵模式、集群模式
- 给缓存业务添加降级限流策略　ngxin或spring cloud gateway
- 给业务添加多级缓存

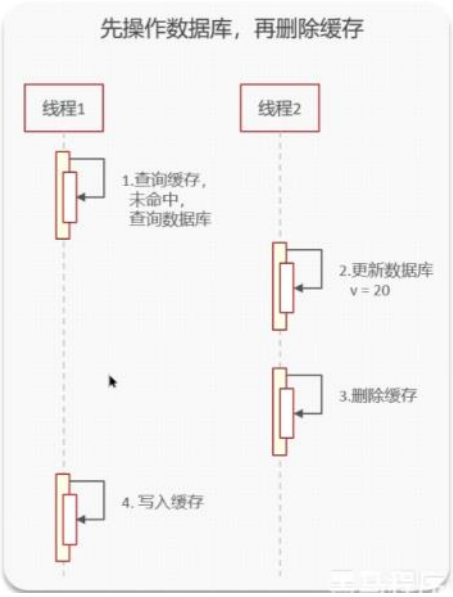
缓存雪崩

1. **缓存雪崩**是指在同一时段大量的缓存key同时失效或者Redis服务宕机，导致大量请求到达数据库，带来巨大压力。
2. **解决方案：**

- 给不同的Key的TTL添加随机值
- 利用Redis集群提高服务的可用性
- 给缓存业务添加降级限流策略　降级可做为系统的保底策略，适用于穿透、击穿、雪崩
- 给业务添加多级缓存

先删除缓存，还是先修改数据库

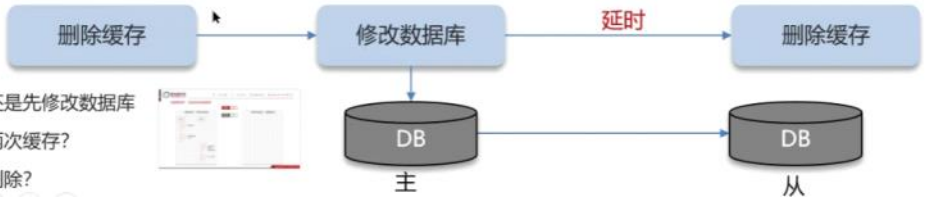




双写一致

双写一致性：当修改了数据库的数据也要同时更新缓存的数据，缓存和数据库的数据要保持一致

- 读操作：缓存命中，直接返回；缓存未命中查询数据库，写入缓存，设定超时时间
- 写操作：延迟双删



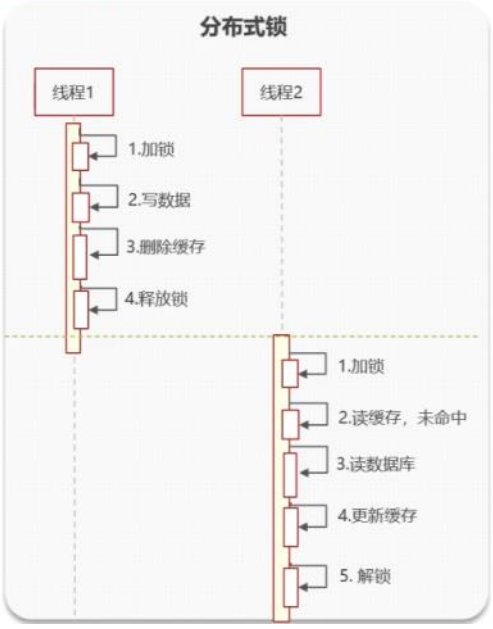
1.先删除缓存，还是先修改数据库

2.为什么要删除两次缓存?

3.为什么要延时删除?

有脏数据风险

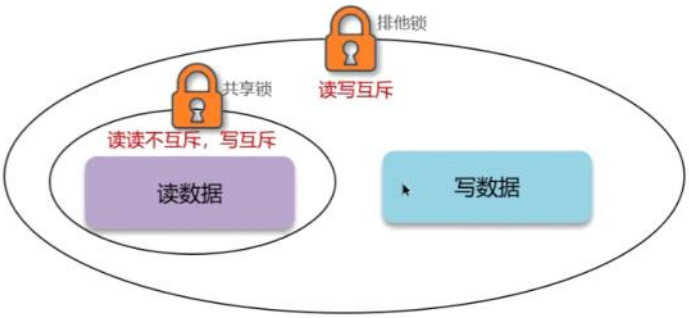
双写一致



读多写少

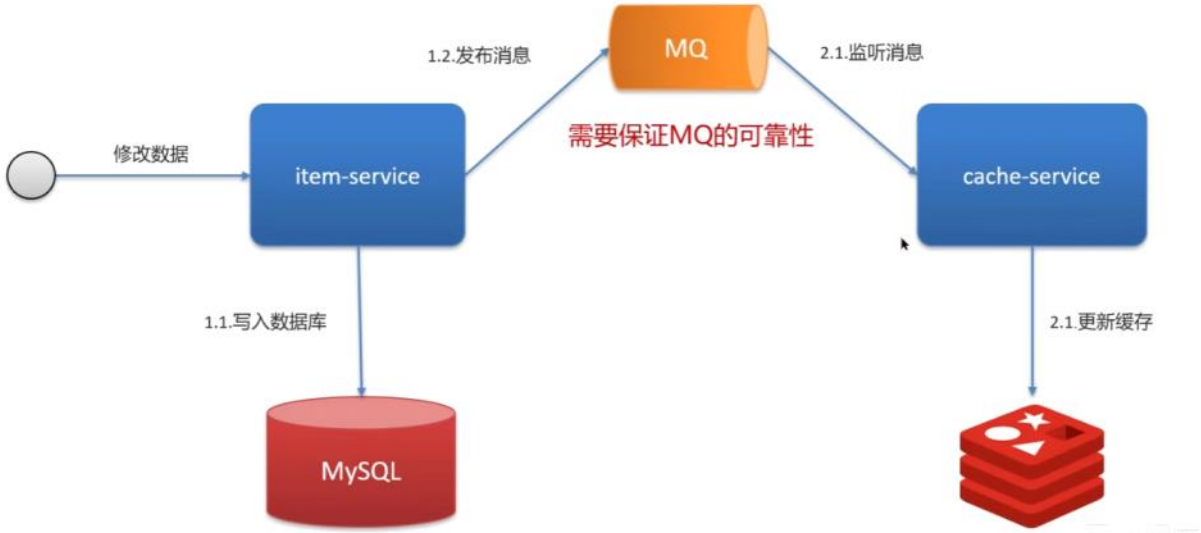
共享锁：读锁readLock，加锁之后，其他线程可以共享读操作

排他锁：独占锁writeLock也叫，加锁之后，阻塞其他线程读写操作



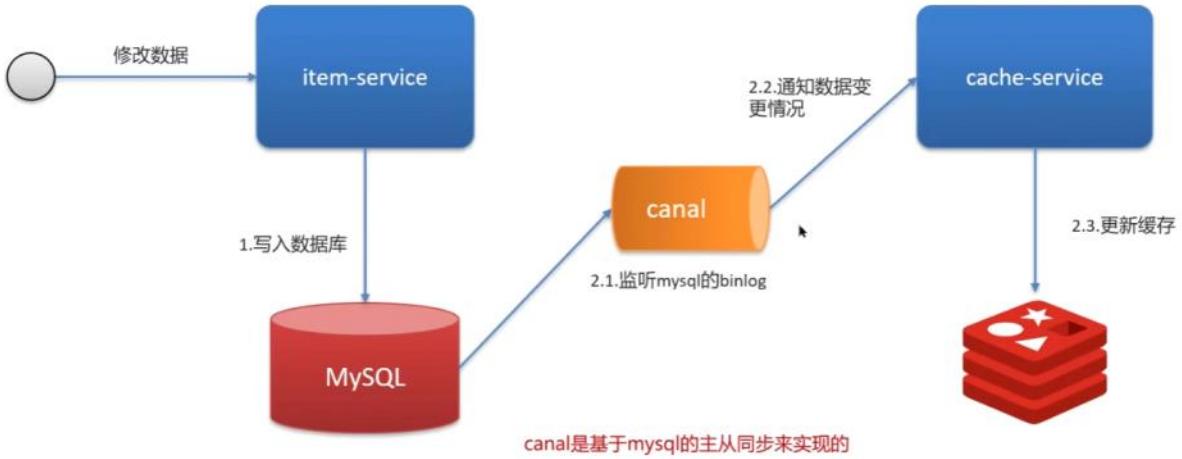
双写一致

异步通知保证数据的最终一致性



双写一致

基于Canal的异步通知：



二进制日志（BINLOG）记录了所有的 DDL（数据定义语言）语句和 DML（数据操纵语言）语句，但不包括数据查询（SELECT、SHOW）语句。

redis做为缓存，mysql的数据如何与redis进行同步呢？（双写一致性）

1. 介绍自己简历上的业务，我们当时是把文章的热点数据存入到了缓存中，虽然是热点数据，但是实时要求性并没有那么高，所以，我们当时采用的是异步的方案同步的数据
2. 我们当时是把抢券的库存存入到了缓存中，这个需要实时的进行数据同步，为了保证数据的强一致，我们当时采用的是redisson提供的读写锁来保证数据的同步

那你来介绍一下异步的方案（你来介绍一下redisson读写锁的这种方案）

- 允许延时一致的业务，采用异步通知

① 使用MQ中间中间件，更新数据之后，通知缓存删除

② 利用canal中间件，不需要修改业务代码，伪装为mysql的一个从节点，canal通过读取binlog数据更新缓存

- 强一致性的，采用Redisson提供的读写锁

① 共享锁：读锁readLock，加锁之后，其他线程可以共享读操作

② 排他锁：独占锁writeLock也叫，加锁之后，阻塞其他线程读写操作

Redis持久化

RDB全称Redis Database Backup file（Redis数据备份文件），也被叫做Redis数据快照。简单来说就是把内存中的所有数据都记录到磁盘中。当Redis实例故障重启后，从磁盘读取快照文件，恢复数据

```
[root@localhost ~]# redis-cli
127.0.0.1:6379> save    #由Redis主进程来执行RDB，会阻塞所有命令
ok
127.0.0.1:6379> bgsave    #开启子进程执行RDB，避免主进程受到影响
Background saving started
```

主动备份

Redis内部有触发RDB的机制，可以在redis.conf文件中找到，格式如下：

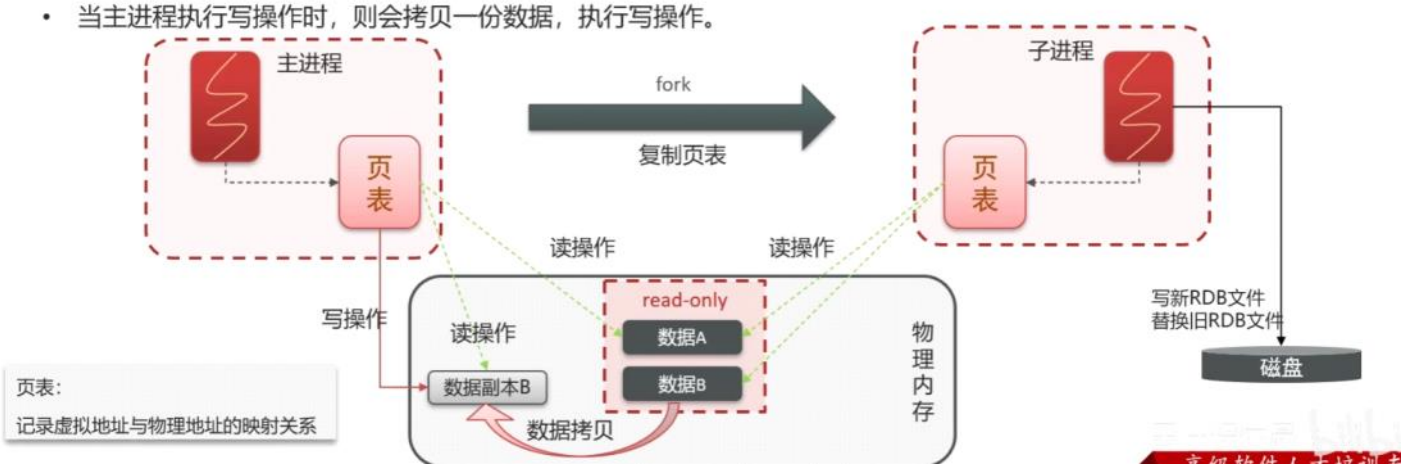
```
# 900秒内，如果至少有1个key被修改，则执行bgsave
save 900 1
save 300 10
save 60 10000
```

RDB的执行原理?

bgsave开始时会fork主进程得到子进程，子进程共享主进程的内存数据。完成fork后读取内存数据并写入 RDB 文件。

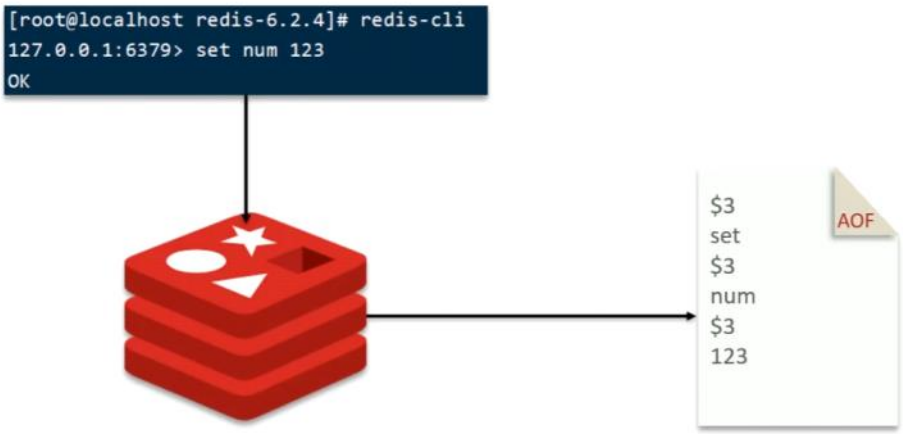
fork采用的是copy-on-write技术：

- 当主进程执行读操作时，访问共享内存；
- 当主进程执行写操作时，则会拷贝一份数据，执行写操作。



AOF

AOF全称为Append Only File（追加文件）。Redis处理的每一个写命令都会记录在AOF文件，可以看做是命令日志文件。



AOF

AOF默认是关闭的，需要修改redis.conf配置文件来开启AOF：

```
# 是否开启AOF功能，默认是no
appendonly yes
# AOF文件的名称
appendfilename "appendonly.aof"
```

AOF的命令记录的频率也可以通过redis.conf文件来配：

```
# 表示每执行一次写命令，立即记录到AOF文件
appendfsync always
# 写命令执行完先放入AOF缓冲区，然后表示每隔1秒将缓冲区数据写到AOF文件，是默认方案
appendfsync everysec
# 写命令执行完先放入AOF缓冲区，由操作系统决定何时将缓冲区内容写回磁盘
appendfsync no
```

配置项	刷盘时机	优点	缺点
Always	同步刷盘	可靠性高，几乎不丢数据	性能影响大
everysec	每秒刷盘	性能适中	最多丢失1秒数据
no	操作系统控制	性能最好	可靠性较差，可能丢失大量数据

AOF

因为是记录命令，AOF文件会比RDB文件大的多。而且AOF会记录对同一个key的多次写操作，但只有最后一次写操作才有意义。通过执行bgrewriteaof命令，可以让AOF文件执行重写功能，用最少的命令达到相同效果。

Redis也会在触发阈值时自动去重写AOF文件。阈值也可以在redis.conf中配置：

```
# AOF文件比上次文件 增长超过多少百分比则触发重写
auto-aof-rewrite-percentage 100
# AOF文件体积最小多大以上才触发重写
auto-aof-rewrite-min-size 64mb
```

RDB与AOF对比

RDB和AOF各有自己的优缺点，如果对数据安全性要求较高，在实际开发中往往会**结合**两者来使用。

	RDB	AOF
持久化方式	定时对整个内存做快照	记录每一次执行的命令
数据完整性	不完整，两次备份之间会丢失	相对完整，取决于刷盘策略
文件大小	会有压缩，文件体积小	记录命令，文件体积很大
宕机恢复速度	很快	慢
数据恢复优先级	低，因为数据完整性不如AOF	高，因为数据完整性更高
系统资源占用	高，大量CPU和内存消耗	低，主要是磁盘IO资源 但AOF重写时会占用大量CPU和内存资源
使用场景	可以容忍数分钟的数据丢失，追求更快的启动速度	对数据安全性要求较高常见

Redis数据删除策略-惰性删除

惰性删除：设置该key过期时间后，我们不去管它，当需要该key时，我们在检查其是否过期，如果过期，我们就删掉它，反之返回该key

例子

```
set name zhangsan 10
get name  //发现name过期了，直接删除key
```

优点：对CPU友好，只会在使用该key时才会进行过期检查，对于很多用不到的key不用浪费时间进行过期检查

缺点：对内存不友好，如果一个key已经过期，但是一直没有使用，那么该key就会一直存在内存中，内存永远不会释放

Redis数据删除策略-定期删除

定期删除：每隔一段时间，我们就对一些key进行检查，删除里面过期的key(从一定数量的数据库中取出一定数量的随机key进行检查，并删除其中的过期key)。

定期清理有两种模式：

- SLOW模式是定时任务，执行频率默认为10hz，每次不超过25ms，以通过修改配置文件redis.conf 的hz 选项来调整这个次数
- FAST模式执行频率不固定，但两次间隔不低于2ms，每次耗时不超过1ms

优点：可以通过限制删除操作执行的时长和频率来减少删除操作对 CPU 的影响。另外定期删除，也能有效释放过期键占用的内存。

缺点：难以确定删除操作执行的时长和频率。

Redis的过期删除策略：惰性删除 + 定期删除两种策略进行配合使用

数据淘汰策略

数据的淘汰策略：当Redis中的内存不够用时，此时在向Redis中添加新的key，那么Redis就会按照某一种规则将内存中的数据删除掉，这种数据的删除规则被称之为内存的淘汰策略。

Redis支持8种不同策略来选择要删除的key：

```
#
# The default is:
#
# maxmemory-policy noeviction
```

- noeviction： 不淘汰任何key，但是内存满时不允许写入新数据，默认就是这种策略。
- volatile-ttl： 对设置了TTL的key，比较key的剩余TTL值，TTL越小越先被淘汰
- allkeys-random：对全体key ，随机进行淘汰。
- volatile-random：对设置了TTL的key ，随机进行淘汰。
- allkeys-lru： 对全体key，基于LRU算法进行淘汰
- volatile-lru： 对设置了TTL的key，基于LRU算法进行淘汰
- allkeys-lfu： 对全体key，基于LFU算法进行淘汰
- volatile-lfu： 对设置了TTL的key，基于LFU算法进行淘汰

key1是在3s之前访问的, key2是在9s之前访问的，删除的就是key2

LRU（**L**east **R**ecently **U**sed）最近最少使用。用当前时间减去最后一次访问时间，这个值越大则淘汰优先级越高。
LFU（**L**east **F**requently **U**sed）最少频率使用。会统计每个key的访问频率，值越小淘汰优先级越高。

key1最近5s访问了4次, key2最近5s访问了9次， 删除的就是key1

数据淘汰策略-使用建议

1. 优先使用 allkeys-lru 策略。充分利用 LRU 算法的优势，把最近最常访问的数据留在缓存中。如果业务有明显的冷热数据区分，建议使用。
2. 如果业务中数据访问频率差别不大，没有明显冷热数据区分，建议使用 allkeys-random，随机选择淘汰。
3. 如果业务中有置顶的需求，可以使用 volatile-lru 策略，同时置顶数据不设置过期时间，这些数据就一直不被删除，会淘汰其他设置过期时间的数据。
4. 如果业务中有短时高频访问的数据，可以使用 allkeys-lfu 或 volatile-lfu 策略。

关于数据淘汰策略其他的面试问题

1. 数据库有1000万数据 ,Redis只能缓存20w数据, 如何保证Redis中的数据都是热点数据 ?

   使用allkeys-lru(挑选最近最少使用的数据淘汰)淘汰策略，留下来的都是经常访问的热点数据

2. Redis的内存用完了会发生什么?

   主要看数据淘汰策略是什么? 如果是默认的配置 （ noeviction ） ， 会直接报错

数据淘汰策略

1. Redis提供了8种不同的数据淘汰策略，默认是noeviction不删除任何数据，内存不足直接报错
2. LRU：最少最近使用。用当前时间减去最后一次访问时间，这个值越大则淘汰优先级越高。
3. LFU：最少频率使用。会统计每个key的访问频率，值越小淘汰优先级越高

平时开发过程中用的比较多的就是allkeys-lru（结合自己的业务场景）