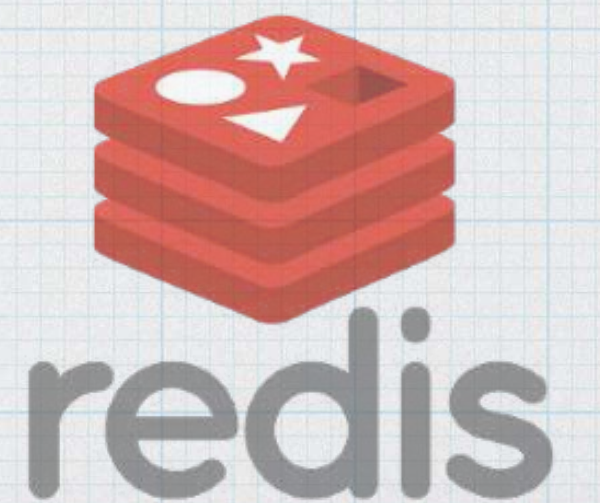


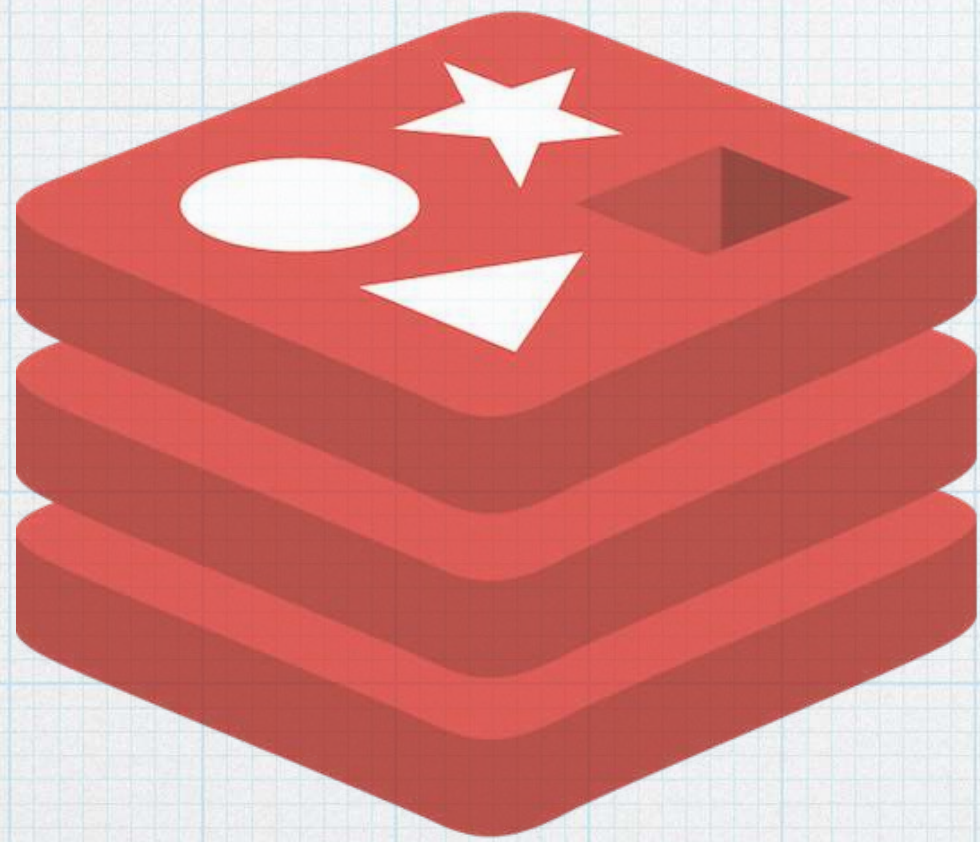
Redis经验之道

xiaorui.cc

github.com/rfyiamcool



menu



- * 数据结构
- * 底层数据结构
- * 功能点
- * 使用经验
- * 高级场景
- * Redis6

Redis数据结构

* String 字符串

* List 列表

* Hash 字典

* Set 集合

* Sorted Set 有序集合

* Stream 流

* bitmap 位图

* pubsub 发布订阅

* geo 地理位置

* hyperloglog 基数统计

Redis底层数据结构

- * SDS

- * LinkedList

- * Ziplist

- * Quicklist

- * Dict

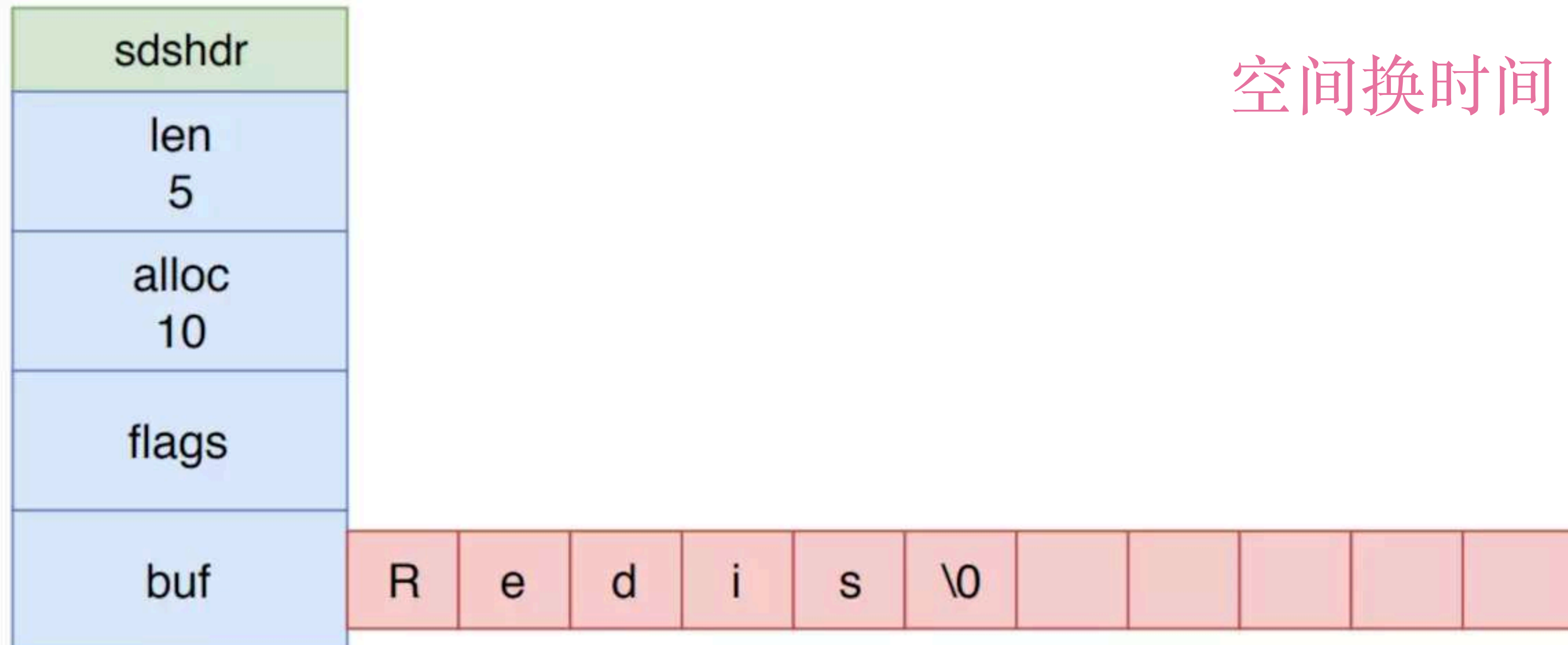
- * Inset

- * SkipTable

- * RadixTree

- * ...

sds简单动态字符串

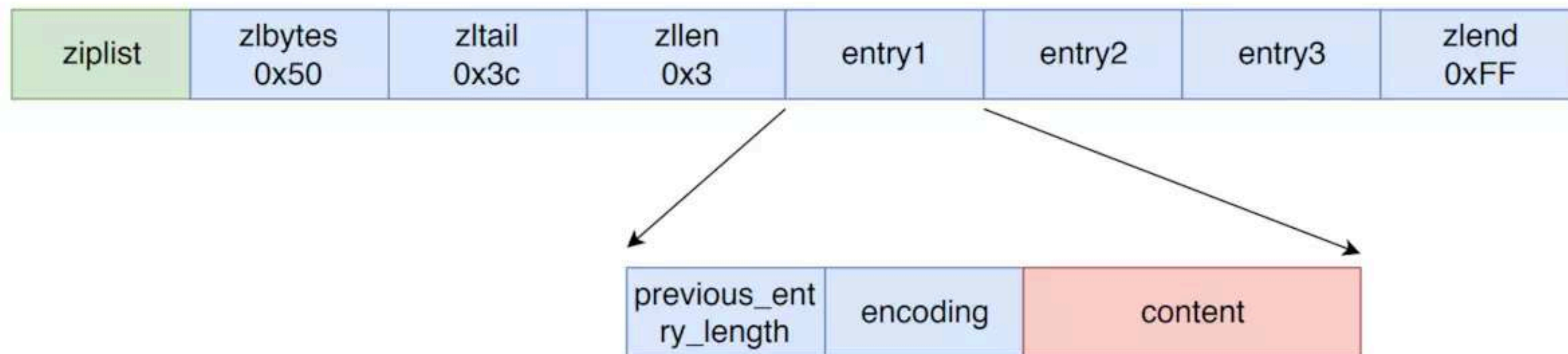


ziplist压缩列表

* 连续内存

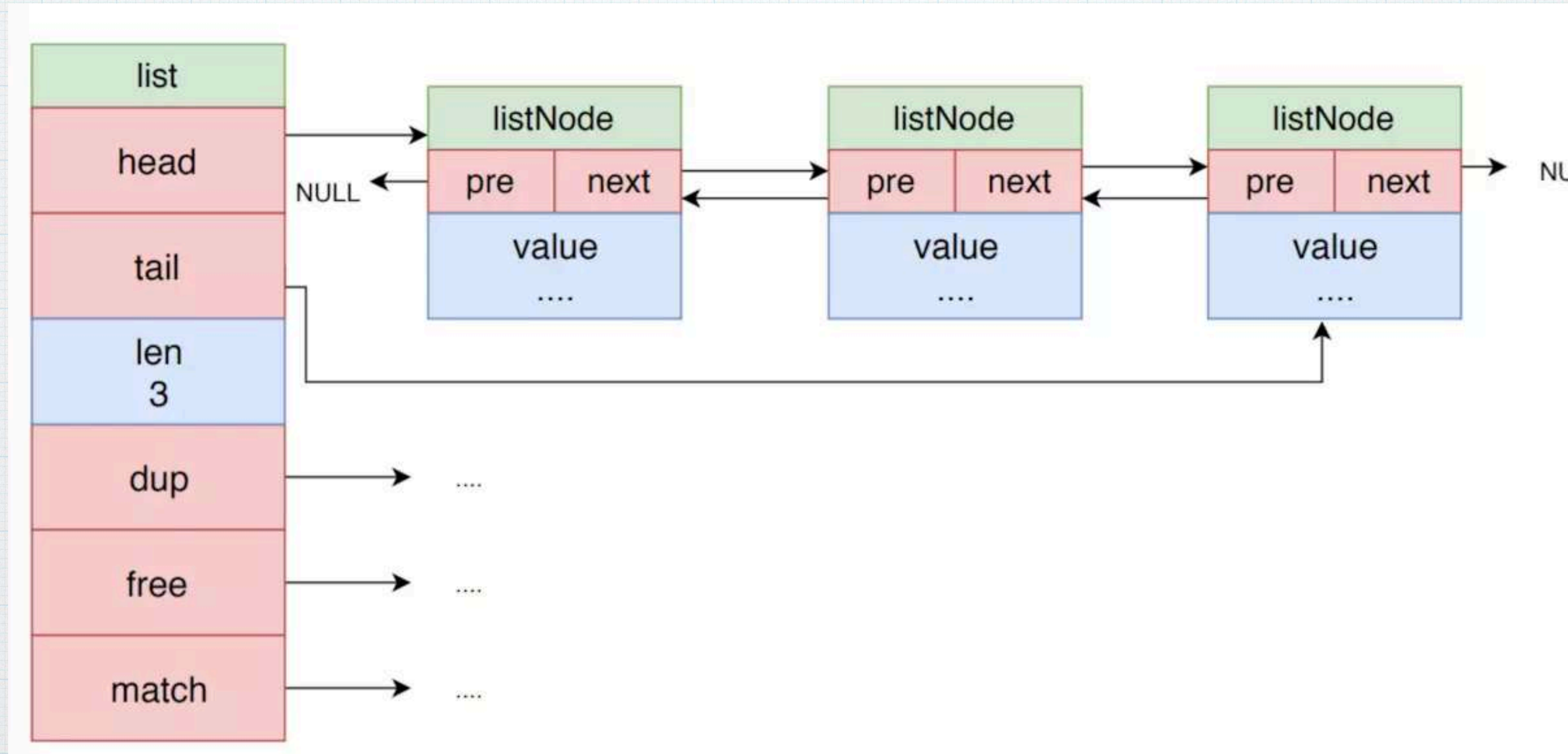
* 特殊编码

* 节省内存



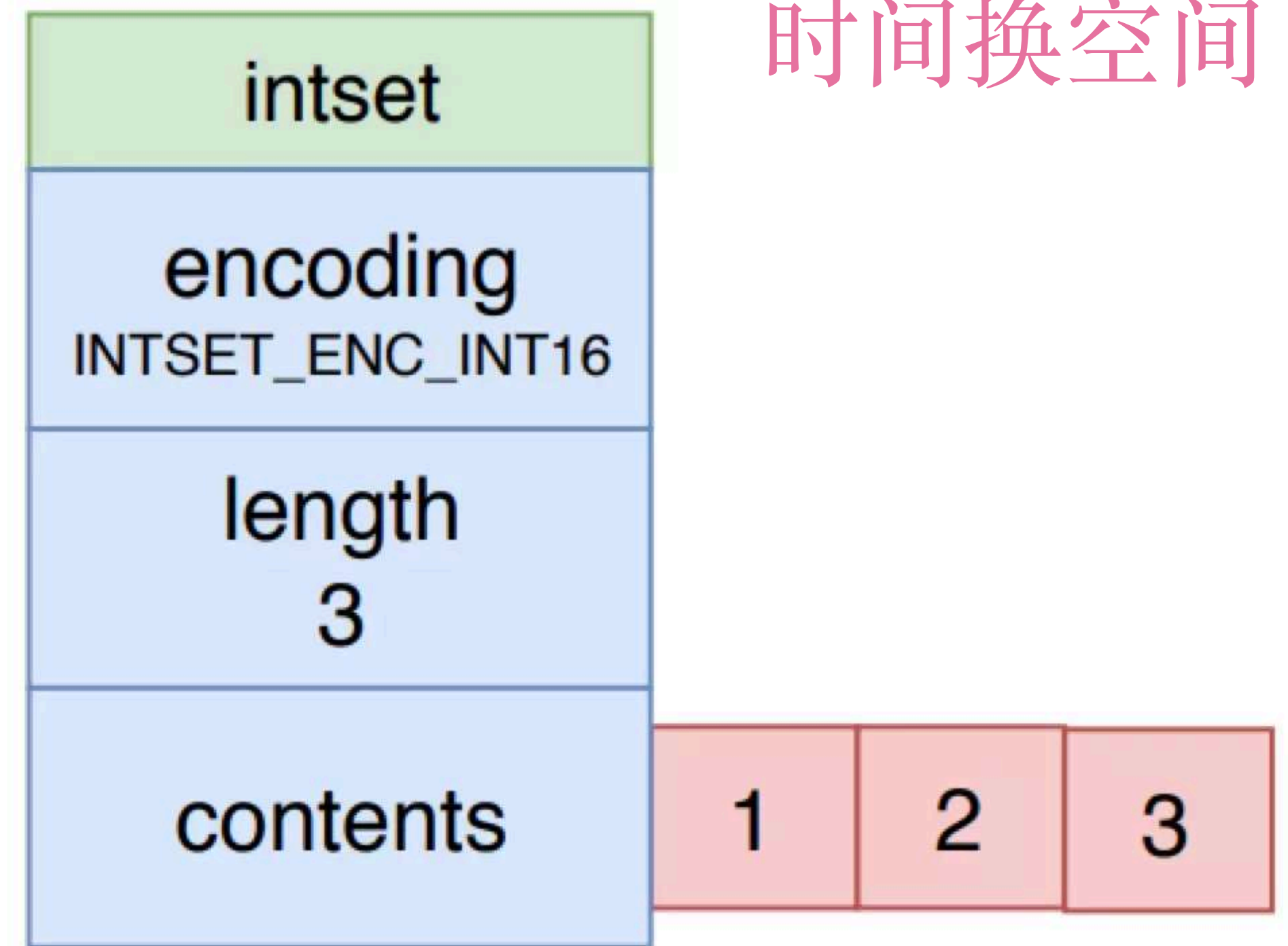
时间换空间！

linkedList (淘汰)

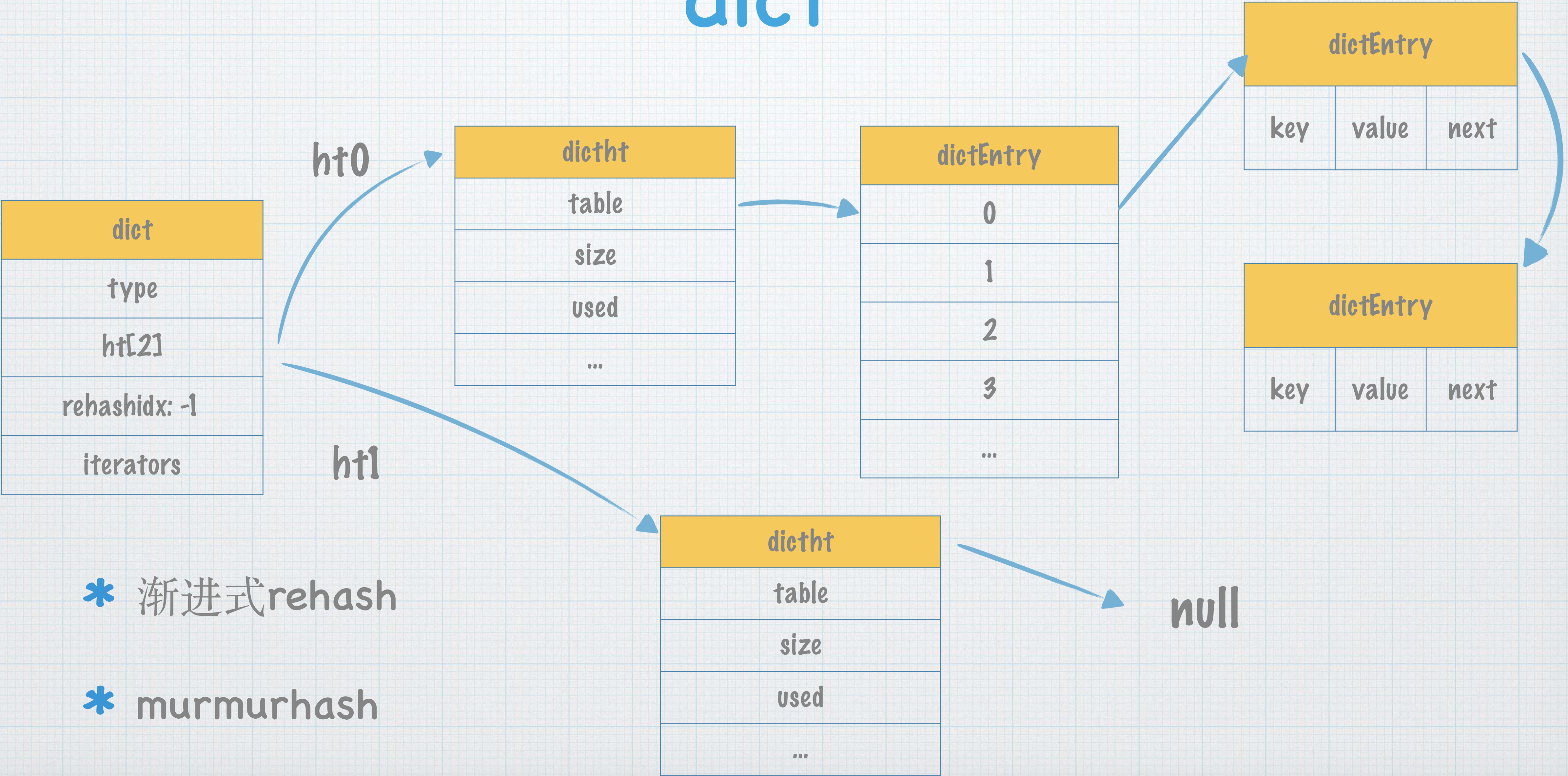


intset整数集合

- * 当value是数字
- * 当size没有超过阈值
 - * 数组项从小到大排序
 - * 二分查找



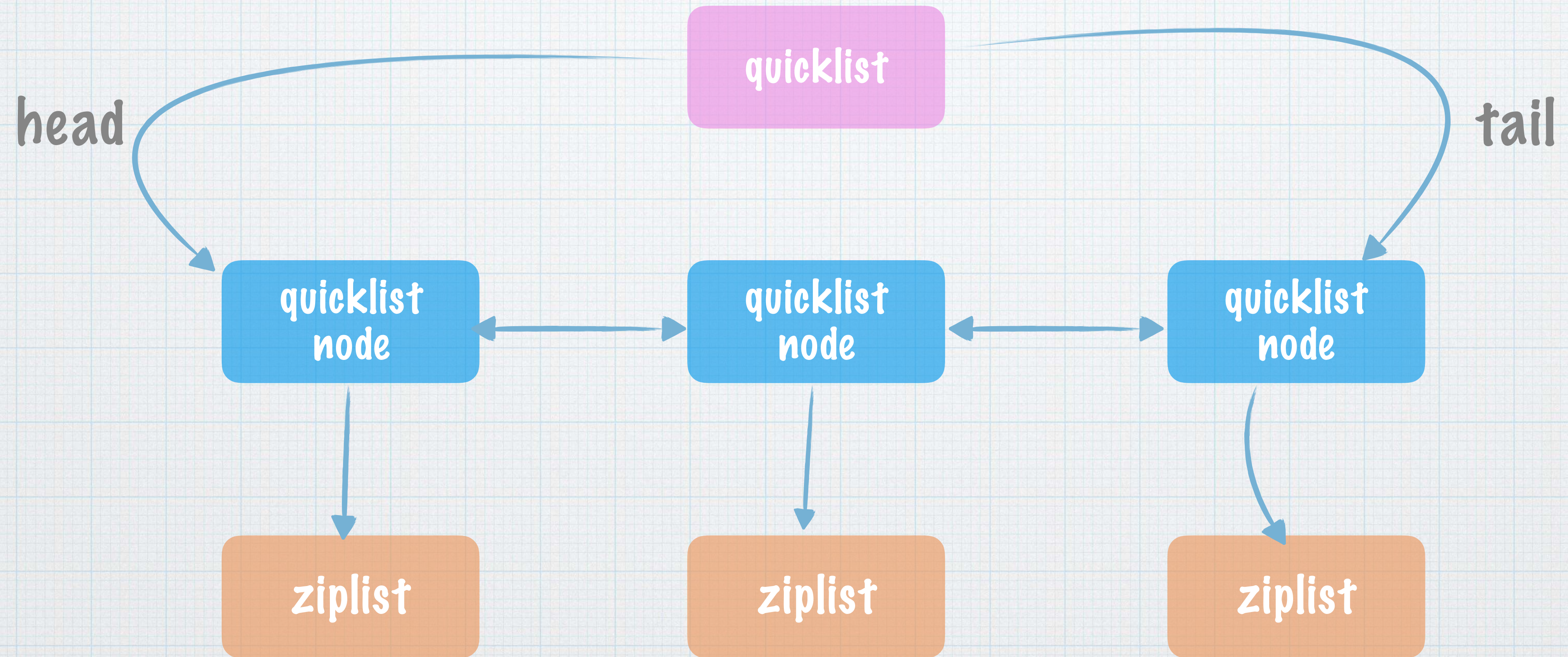
dict



* 渐进式rehash

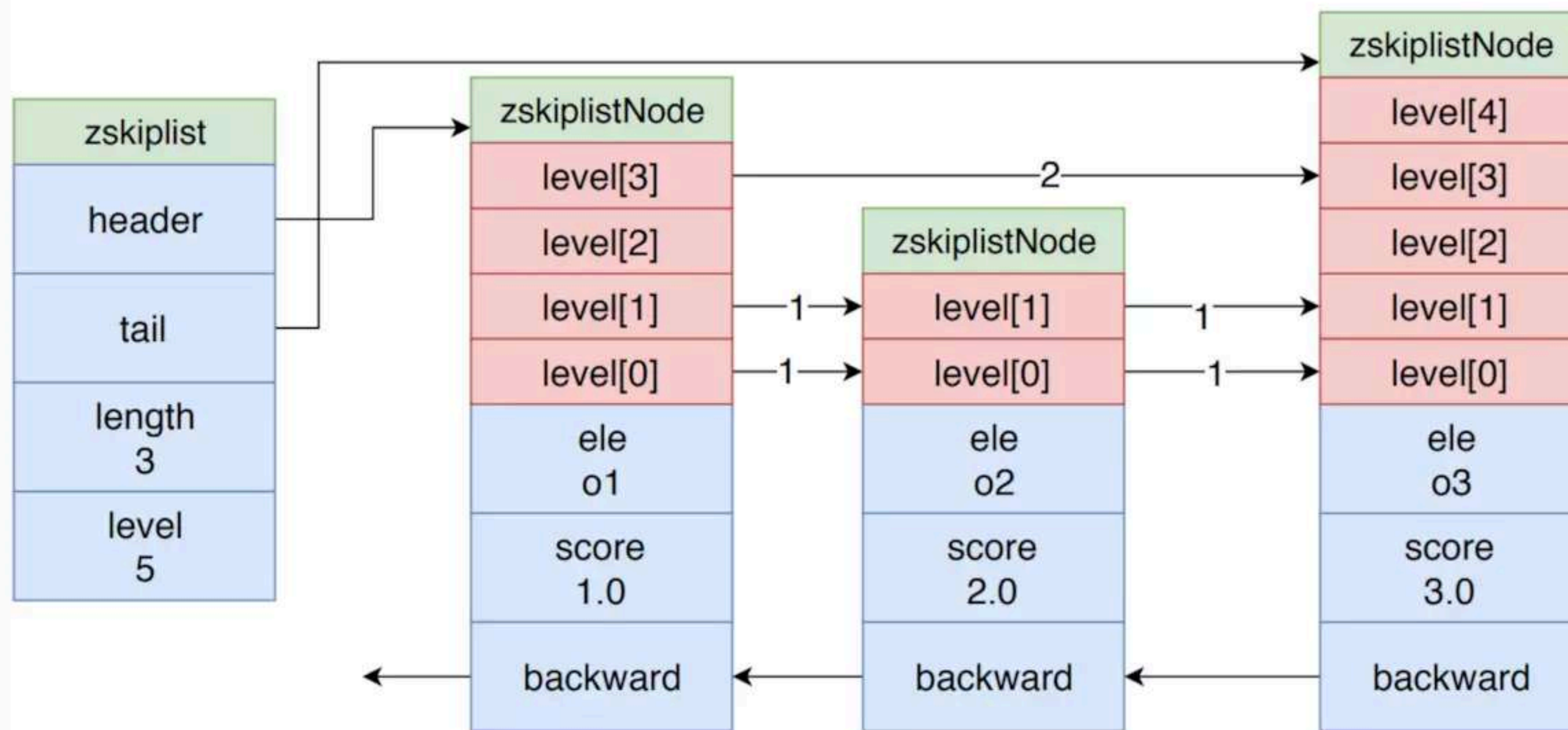
* murmurhash

quicklist



链表和ziplist的组合！

skiptable跳跃表



Redis数据结构组成

- * String

- * sds

- * Hash

- * ziplist

- * dict

- * Set

- * inset

- * dict

- * List

- * ziplist

- * quicklist

- * Sorted Set

- * hash + skiptable

- * Stream

- * radix-tree

key的规范

- * 每个业务一个database (cluster除外)
- * 加入项目的前缀
- * 一级key不要超过千万
- * 尽量都加入TTL
- * 使用hashtag来绑定节点亲和性

Value的规范

- * 选择合适的数据结构
- * 长字符压缩存取 (snappy, msgpack, more...)
- * 避免big key (删除和迁移时阻塞)
- * 避免hot key (单点性能)

ziplist更省内存

* hash (order_id)

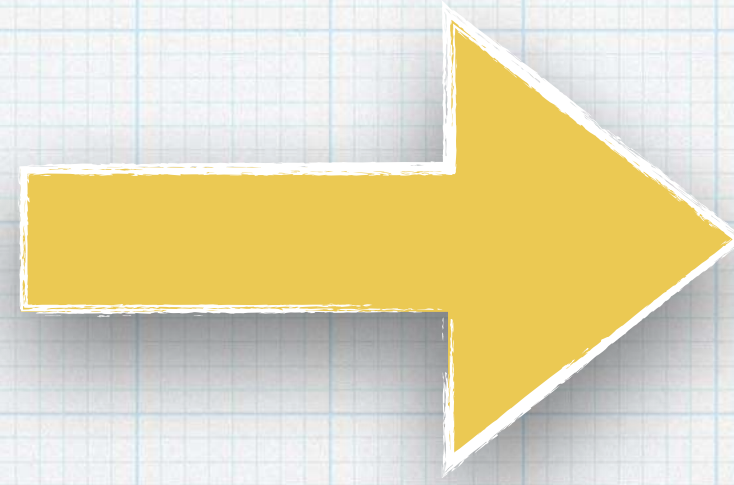
* oid_1000

* oid_1002

* oid_1333

* oid_2111

* oid_2333



* hash (order_id_1000)

* oid_1000

* oid_1002

* oid_1333

* hash (order_id_2000)

* oid_2111

* oid_2333

* hash-max-ziplist-entries = 1000

* hash-max-ziplist-value = 128

省内存

- * 连续内存, 紧凑的编码, 减少了碎片, 减少了指针引用
- * ziplist支持的数据结构
 - * hash-max-ziplist-entries && hash-max-ziplist-value
 - * list-max-ziplist-size
 - * zset-max-ziplist-entries && zset-max-ziplist-value

持久化

- * RDB

- * AOF

- * always

- * every sec

- * 混合模式 RDB + AOF

- * 加载顺序

- * 先 AOF

- * 后 RDB

注意事项

- * 避免使用 $O(n)$ 的指令 (`keys *`, `hgetall`, `smembers`, `sunion ...`)
 - * 使用`scan`, `hscan`, `sscan`, `zscan`
 - * 使用`unlink`异步删除key
 - * 业务层规避这类设计

提高吞吐

- * 使用pipeline批量传输, 减少网络RTT
- * 使用多值指令 (mset, hmset)
- * 使用script lua
- * 干掉aof ?

(big key) or (hot key)

- * big key

- * scan / small range get

- * unlink (redis 4.0 async del)

- * hash shard

- * hot key

- * hash shard

redis lua

- * 减少RTT消耗
- * 保证多指令原子性
- * 自定义指令
- * 注意阻塞问题
- * 逻辑封装

redis module

- * 自定义注册新指令
- * 性能比redis lua更强劲
- * redis4.0 以上

- * RedisJson
- * RedisBloom
- * RedisTimeSeries
- * more ...

排查问题

- * 外部

- * redis-cli monitor

- * 内部

- * keyspace

- * slow log

- * 内存碎片

- * -- bigkeys

- * string, bytes空间

- * set, list, zset, hash, 元素个数

- * redis-rdb-tool

- * 分析内存分布

- * `memory usage key_name`

- * `memory stats`

- * `memory purge`

- * 阻塞及延迟

- * `redis-cli --intrinsic-latency 10`

- * `redis-cli --latency-history`

- * `info -> instantaneous_ops_per_sec`

- * `info -> used_memory_human`

- * `connected_clients`

单线程？

* 主线程

* aof日志

* bio线程

```
Thread 4 (Thread 0x7f4aaf96a700 (LWP 10485)):  
#0  0x00007f4ab647d68c in pthread_cond_wait@@GLIBC_2.3.2 () from /lib64/libpthread.so.0  
#1  0x000000000048ccbd in bioProcessBackgroundJobs ()  
#2  0x00007f4ab6479aa1 in start_thread () from /lib64/libpthread.so.0  
#3  0x00007f4ab61c6bcd in clone () from /lib64/libc.so.6  
Thread 3 (Thread 0x7f4aaef69700 (LWP 10486)):  
#0  0x00007f4ab647d68c in pthread_cond_wait@@GLIBC_2.3.2 () from /lib64/libpthread.so.0  
#1  0x000000000048ccbd in bioProcessBackgroundJobs ()  
#2  0x00007f4ab6479aa1 in start_thread () from /lib64/libpthread.so.0  
#3  0x00007f4ab61c6bcd in clone () from /lib64/libc.so.6  
Thread 2 (Thread 0x7f4aae568700 (LWP 10487)):  
#0  0x00007f4ab647d68c in pthread_cond_wait@@GLIBC_2.3.2 () from /lib64/libpthread.so.0  
#1  0x000000000048ccbd in bioProcessBackgroundJobs ()  
#2  0x00007f4ab6479aa1 in start_thread () from /lib64/libpthread.so.0  
#3  0x00007f4ab61c6bcd in clone () from /lib64/libc.so.6  
Thread 1 (Thread 0x7f4ab6f30f40 (LWP 10482)):  
#0  0x00007f4ab61c71c3 in epoll_wait () from /lib64/libc.so.6  
#1  0x000000000042b3de in aeProcessEvents ()  
#2  0x000000000042b88b in aeMain ()  
#3  0x00000000004342b9 in main ()
```


过期key的实现

* 惰性删除

* 定时删除

- ◆ 默认每100ms进行一次
- ◆ 从过期字典中随机取出 20 个键
- ◆ 删除这 20 个键中过期的键
- ◆ 如果过期键的比例超过 25%，重复步骤 1 和 2
- ◆ 直到25ms最长执行时间退出

* 触发MaxMemory时尝试删除

```
void activeExpireCycle(int type) {
    // 最多允许25%的CPU时间用于过期Key清理
    // 若hz=10, 则一次activeExpireCycle最多只能执行25ms
    timelimit = 1000000*ACTIVE_EXPIRE_CYCLE_SLOW_TIME_PERC/server.hz/100;

    for (j = 0; j < dbs_per_call; j++) {
        int expired;
        redisDb *db = server.db+(current_db % server.dnum);

        current_db++;
        do {
            ...
            // 一次取20个Key, 判断是否过期
            if (num > ACTIVE_EXPIRE_CYCLE_LOOKUPS_PER_LOOP)
                num = ACTIVE_EXPIRE_CYCLE_LOOKUPS_PER_LOOP;

            while (num--) {
                dictEntry *de;
                long long ttl;

                if ((de = dictGetRandomKey(db->expires)) == NULL) break;
                ttl = dictGetSignedIntegerVal(de)-now;
                if (activeExpireCycleTryExpire(db,de,now)) expired++;
            }

            if ((iteration & 0xf) == 0) { /* check once every 16 iterations.
                long long elapsed = ustime()-start;
                latencyAddSampleIfNeeded("expire-cycle",elapsed/1000);
                if (elapsed > timelimit) timelimit_exit = 1;
            }

            // 超过25ms退出
            if (timelimit_exit) return;
            // 超时的key超过了25%, 继续清理
        } while (expired > ACTIVE_EXPIRE_CYCLE_LOOKUPS_PER_LOOP/4);
    }
}
```


缓存淘汰

- * volatile-lru (默认)

- * 从设置过期数据集里查找最近最少使用

- * volatile-ttl

- * 从设置过期的数据集里清理已经过期的key.

- * volatile-random

- * 从设置过期的数据集中任意选择数据淘汰

- * allkeys-lru

- * 从数据集中挑选最近最少使用的数据淘汰

- * allkeys-random

- * 从数据集中任意选择数据淘汰

- * no-eviction

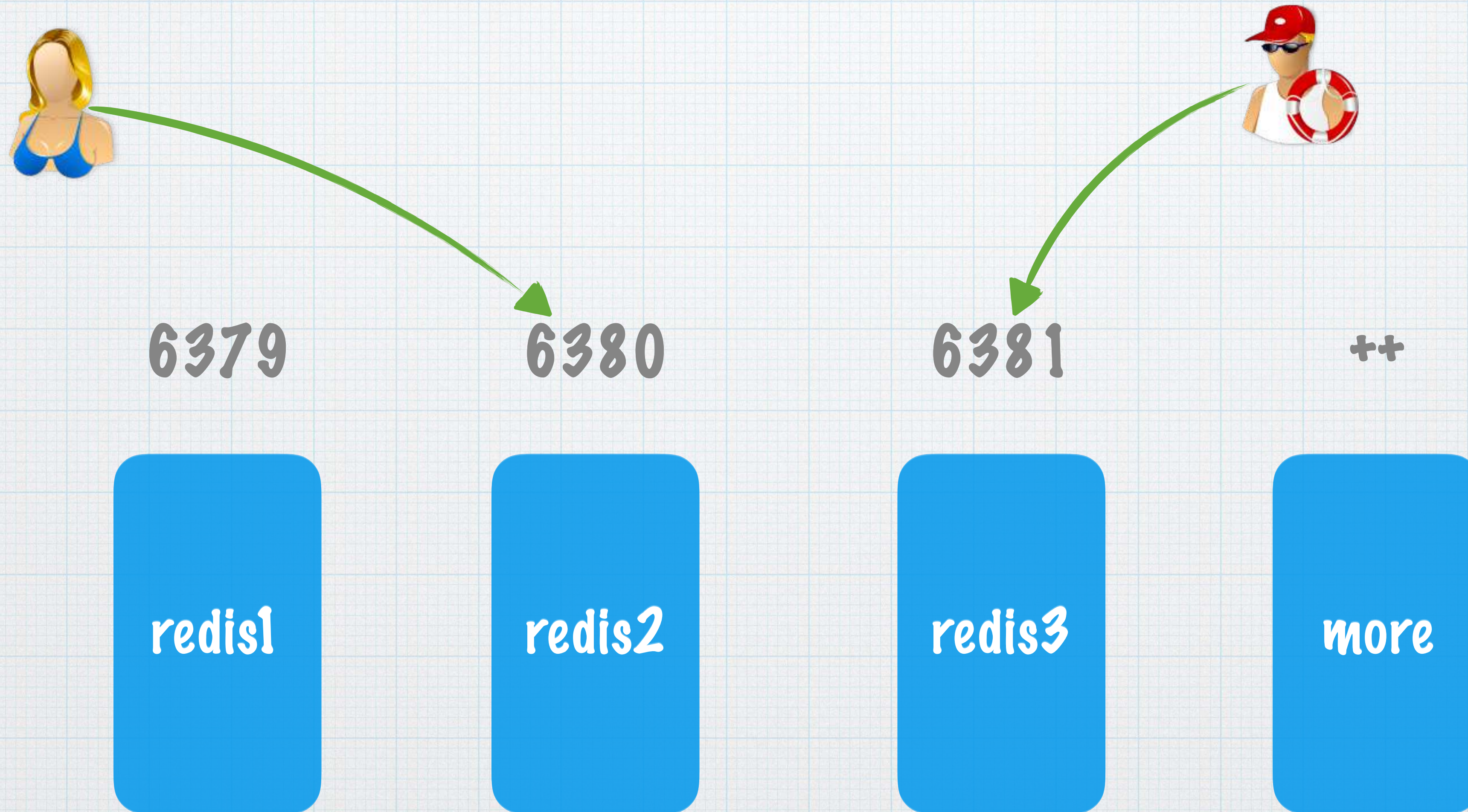
- * 不清理

单机进化到多实例

- * 什么是多实例
- * 为什么要多实例化
- * 多实例化需要注意什么？

单机多实例

what 多实例



why 多实例

- * 最大程度的使用内存
- * 避免单实例RDB Write时
 - * 被kernel oom
 - * 使用swap造成阻塞.
- * 单实例启动太慢
- * 扩展, 迁移, 内存随便整理
- * copy on write will block
- * 绕开redis单工作线程的问题
 - * 阻塞指令
 - * 系统调用消耗
 - * busy event
 - * hashcrc
 - * more ...

How 多实例

- * 比如, 128G 内存.
- * 11G 为一个实例, 启动个10实例
- * 后台脚本来控制bgsave.
- * 启动时也是一个一个的启动

简约集群

* vip多线程版 twemproxy

* codis

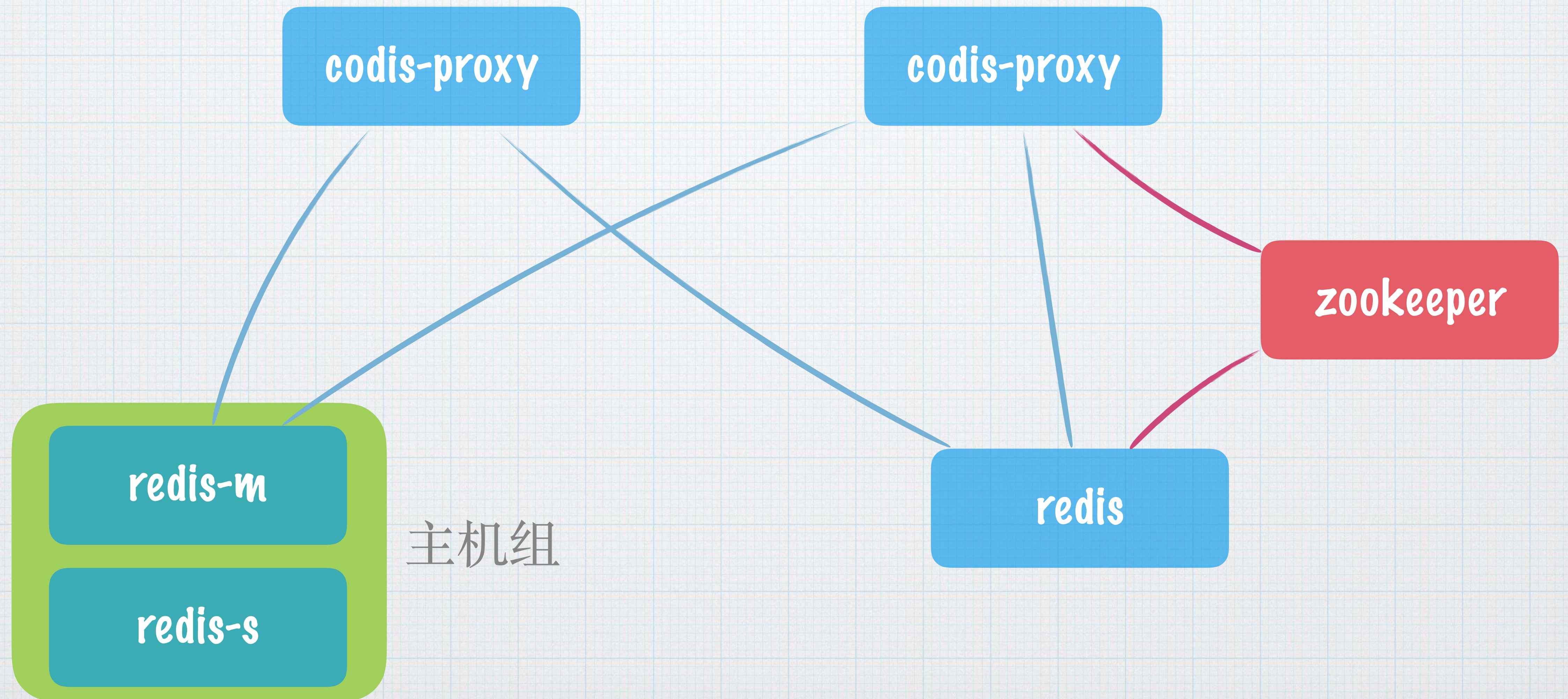
* redis cluster

集群

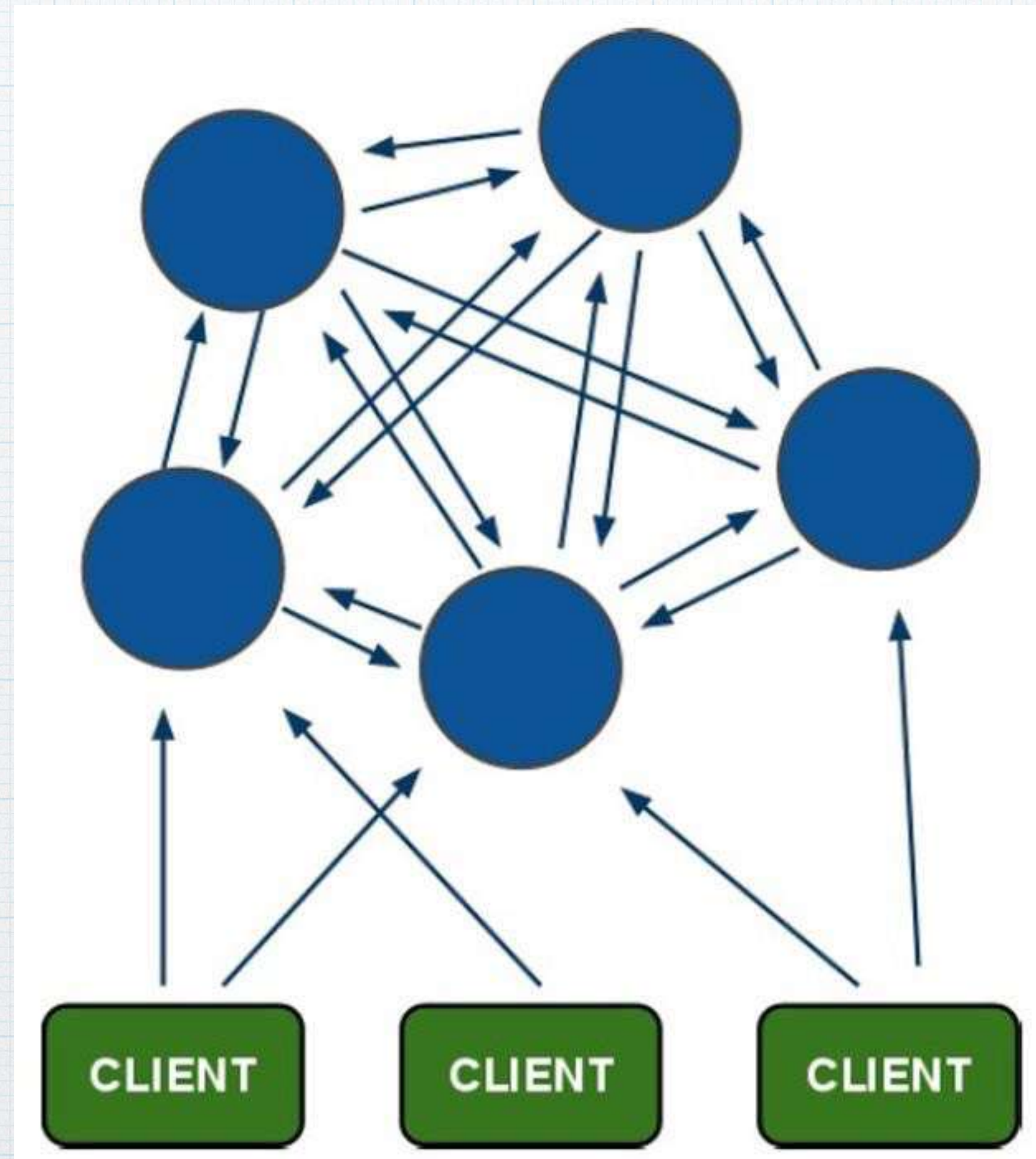
codis vs redis cluster

	cluster	codis
hash_tag	y	y
design	中心化	去中心化
pipeline	client move order	支持
slot	y	y
多租户	y	y
性能	high	this < cluster
code	复杂	简单
范围	广	也有不少大厂

codis



redis cluster



常用场景

- * 缓存

- * 缓存一致性

- * 缓存穿透

- * 缓存击穿

- * 缓存雪崩

- * 分布式锁

- * redlock

缓存一致性

- * write cache → write db

- * write db → write cache

- * **evict cache → write db**

- * write db → evict cache

- * evit cache → write db →
evit cache

缓存一致性

- * write cache -> write db

- * 更新db失败

- * write db -> write cache

- * 更新cache失败

- * 并发引起脏数据

- * client1更新了DB

- * client2更新了DB

- * client2更新cache

- * 但client1覆盖了client2的cache

缓存一致性

- * evict cache -> write db

- * 延迟引起脏数据

- * client1 先删除缓存

- * client2 查询发现缓存不存在

- * client2 数据库查询得到旧值

- * client2 将旧值写入缓存

- * client1 将新值写入数据库

- * write db -> evict cache

- * client1 查询数据库, 得一个旧值

- * client2 将新值写入数据库

- * client2 删除缓存

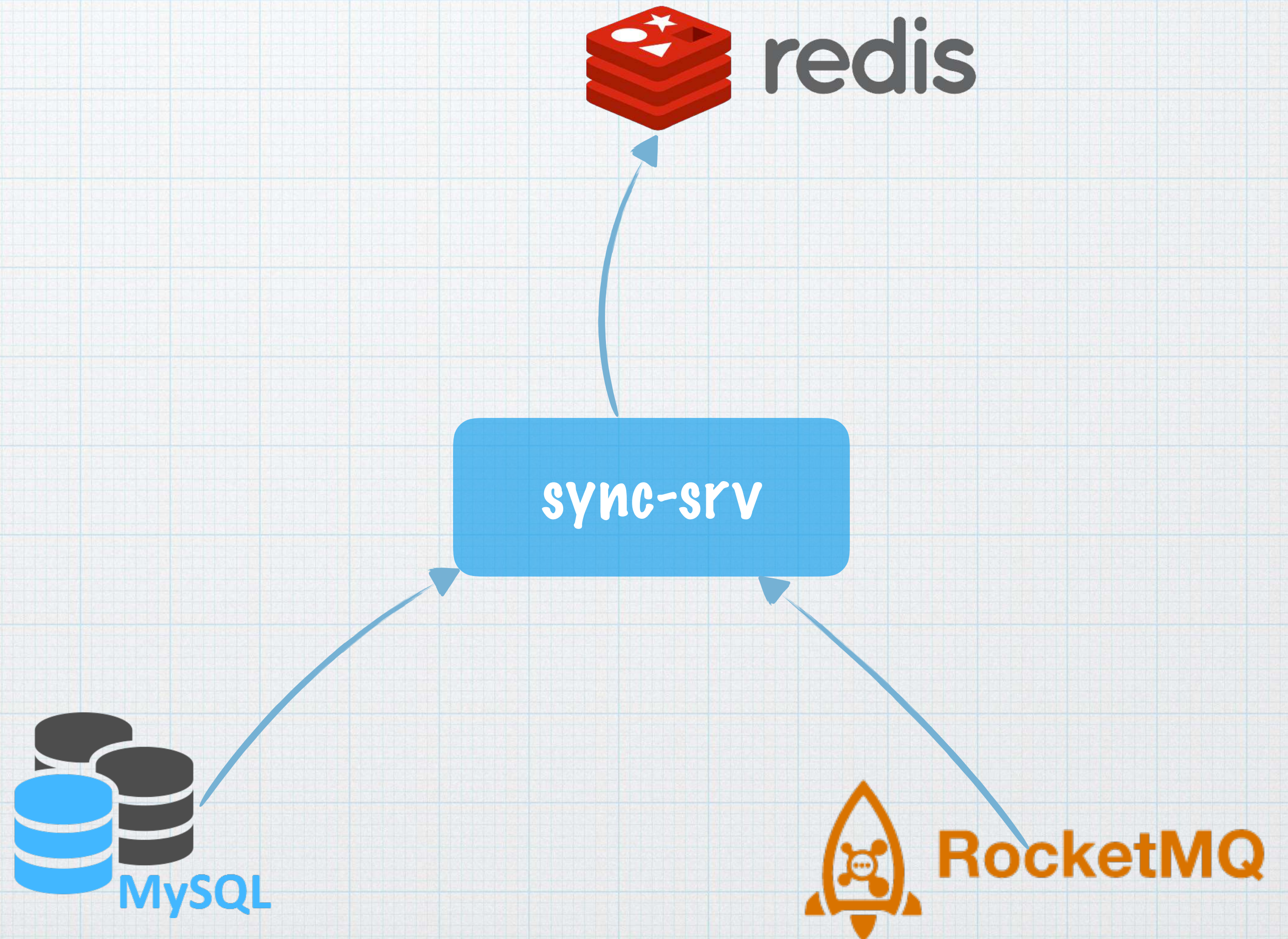
- * client1 将查到的旧值写入缓存

- * evit cache -> write db -> evit cache

- * 脏数据概率更小

规避脏数据

- * TTL
- * 定时更新
- * Binlog订阅更新
- * Delay Queue



缓存一致性

- * 在更新策略上, 难以保证强一致性
- * 但可以最终一致性
- * 拼概率, 减少产生脏数据的可能

- * `evct cache; write db; binlog or ttl` 双写

- * 多数公司的选择

- * `write db; binlog`更新

- * didi, iqiyi

- * `write db; evct cache` 也是个好选择！

- * facebook

穿透&雪崩

- * 穿透 (访问一个不存在的key)

- * 在缓存中加入该key的null值

- * 雪崩 (大量key的失效)

- * 不主动配置TTL

- * 后台同步缓存时, 加入jitter ttl

击穿

- * 击穿 (大量请求未缓存的key)
 - * 实现redis分段锁, 同样的请求争夺一把锁
 - * 拿到锁的去数据库查询
 - * 未拿到锁的等待, 再尝试访问缓存
 - * 缓存中还没有数据, 尝试数据库拿取

分布式锁

- * 安全可靠

- * say no

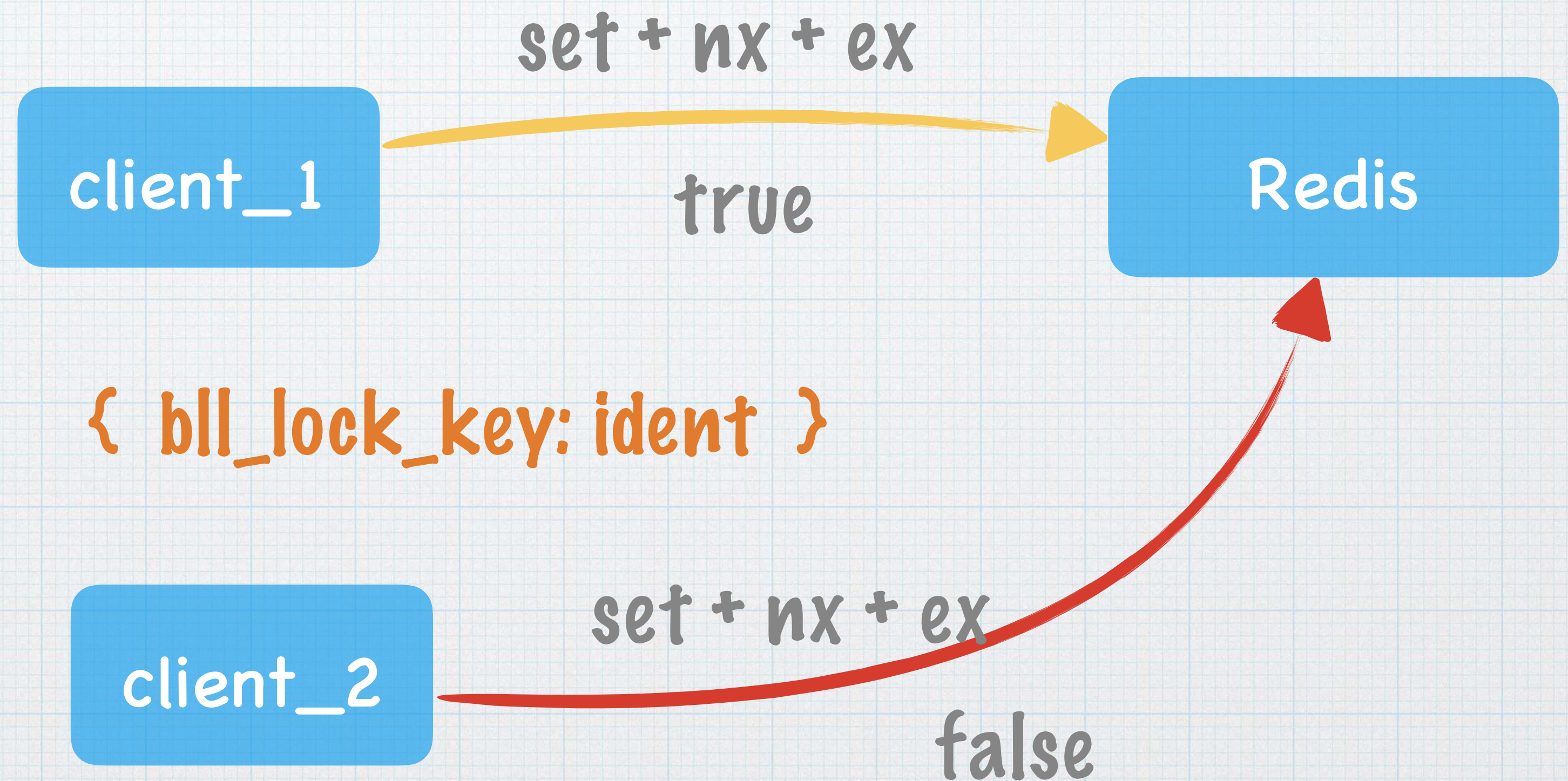
- * 可重入锁

- * say yes

- * 公平调度

- * say hard

lua make (compare and set) !!!



redlock

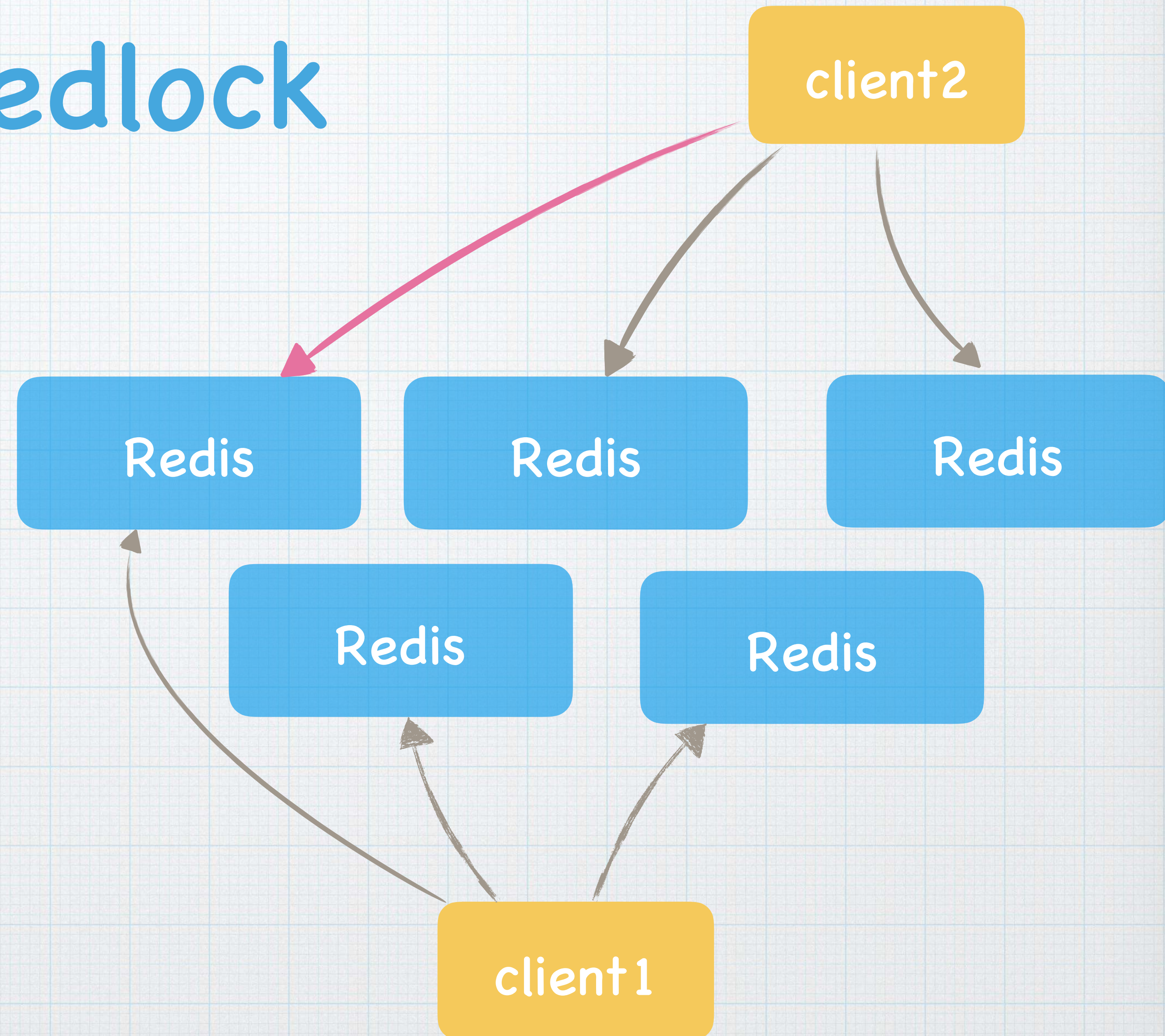
- * $2/n + 1$

- * 推荐

- * 最少5个实例

- * 3个及以上拿到锁

- * 未拿到锁的, 释放锁



经历过的性能指标

- * 1w 的稳定长连接
- * 10w TPS
- * 队列千万级别
- * 百万数量key

单节点

经历过的容量

- * 300个redis实例
- * 30台服务器 (混部)
- * 每个实例10G
- * 约 3T 内存

集群

redis 6

- * 新增的resp3加入缓存特性

 - * 返回key的属性, 比如频率

 - * 更新范围

- * acl用户权限控制

 - * 控制命令及key

- * redis cluster proxy

 - * 兼容各类sdk

- * io多线程

 - * io线程负责read, decode, encode, write

 - * 操作内存还是主线程

other

- * redis cluster那些事儿

- * https://github.com/rfyiamcool/share_ppt/blob/master/redis_cluster.pdf

- * redis之高级应用

- * https://github.com/rfyiamcool/share_ppt/blob/master/redis_advance.pdf

- * 大话redis设计与实现

- * https://github.com/rfyiamcool/share_ppt/blob/master/rediscode.pdf

“Q & A!”

一峰云就她了