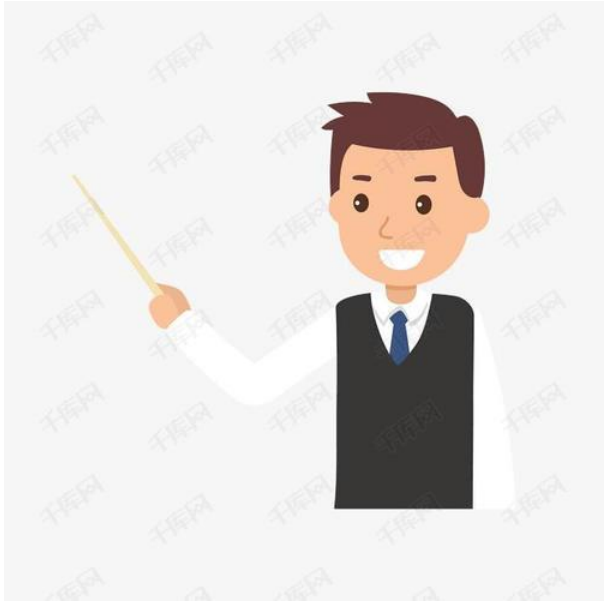
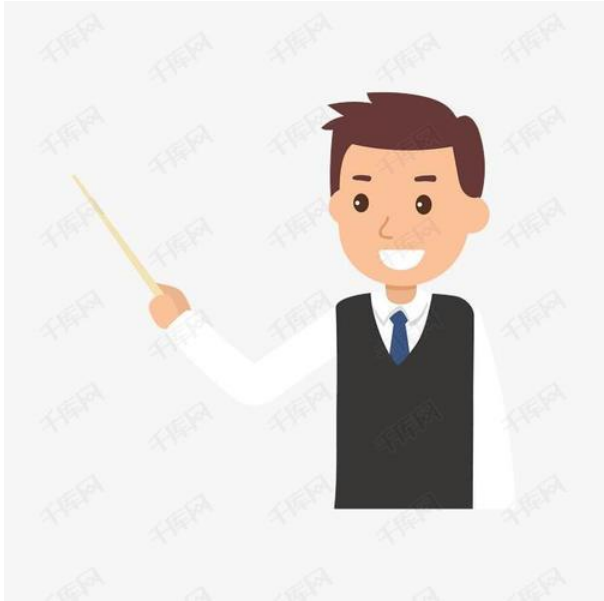
golang本地缓存选型及原理总结

(bigcache/freecache/fastcache/offheap/go-cache)

目录



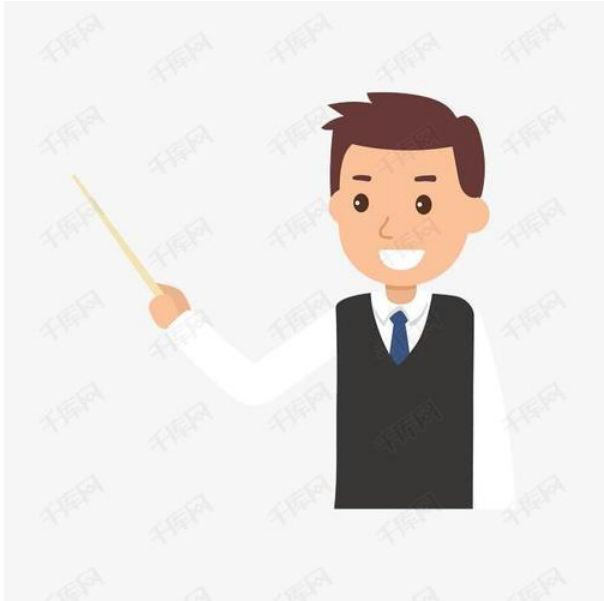
目录



需求：

- 1. 需要较高读写性能+命中率
- 2. 支持按写入时间过期
- 3. 支持淘汰策略
- 4. 需要解决gc问题，否则大量对象写入会引起stw扫描标记时间过长，cpu毛刺严重

目录



1. **freecache**：https://github.com/coocood/freecache
2. **bigcache**：https://github.com/allegro/bigcache
3. **fastcache**：https://github.com/VictoriaMetrics/fastcache
4. **offheap**：https://github.com/glycerine/offheap
5. **groupcache**：https://github.com/golang/groupcache
6. **ristretto**：https://github.com/dgraph-io/ristretto
7. **go-cache**：https://github.com/patrickmn/go-cache

指标	freecache	bigcache	fastcache	offheap	groupcache	ristretto
适用场景	高并发，读多写少	高并发，读多写少	高并发，读多写少	高并发，读多写少	并发读取预加载不可变的内容	高并发，读多写少
社区活跃度	fork:314 star:3.7k	fork:476 star:5.5k	fork:110 star:1.4k	fork:28 star:307	fork:1.3k starr:11.2k	fork:244 star:3.7k
存储数据限制	受初始化内存大小限制	每个shard存储数据的queue可以自动扩容	受初始化内存大小限制	Cell可以自动扩容	无参数限制	无参数限制
过期时间支持	支持	支持	不支持	不支持	不支持	支持
容量淘汰机制	近似LRU（LRU+移动次数(SET触发)）	FIFO（SET接口/定期清理）	FIFO（SET/8K周期清理）	不支持	LRU	SampledLFU、TinyLFU
GC	**Zero GC**	**Zero GC**	**Zero GC**	**Zero GC**	GO GC	GO GC
GC优化原理	Map非指针优化 通过slice内部实现map，减少指针	Map非指针优化 map[uint64]uint32	Map非指针优化 map[uint64]uint32 **堆外申请内存**	**堆外申请内存** syscall.Mmap	无优化	无优化
锁机制	分片+ 互斥锁	分片+读写锁	分片+读写锁	LockFree	读写锁	分片+读写锁
局限	每个bucket空闲分配后无法更改	组件有全局的过期时间，但无法对单条数据设置过期时间	无法设置过期时间	通过本地实现哈希表，采用探测法解决冲突，冲突严重时性能低	无过期时间能力	
是否推荐	推荐	推荐	推荐	不推荐	推荐	推荐(有GC)
总结	1.本地缓存最终都会受单机内存容量的限制 2.平均响应时间在ms级 3.**绝大部分此类组件都是基于分片+读写锁来实现功能、底层数据要么采用淘汰算法淘汰，要么采用环形队列循环使用空间**					

实现零GC方案

a.无GC：

- 分配堆外内存(Mmap)

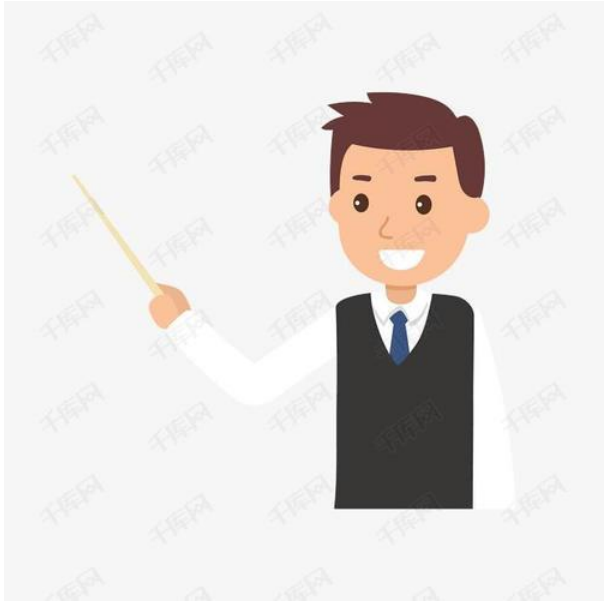
b.避免GC：

- map非指针优化(map[uint64]uint32)或者采用slice实现一套无指针的map
- 数据存入[]byte slice(可考虑底层采用环形队列封装循环使用空间)

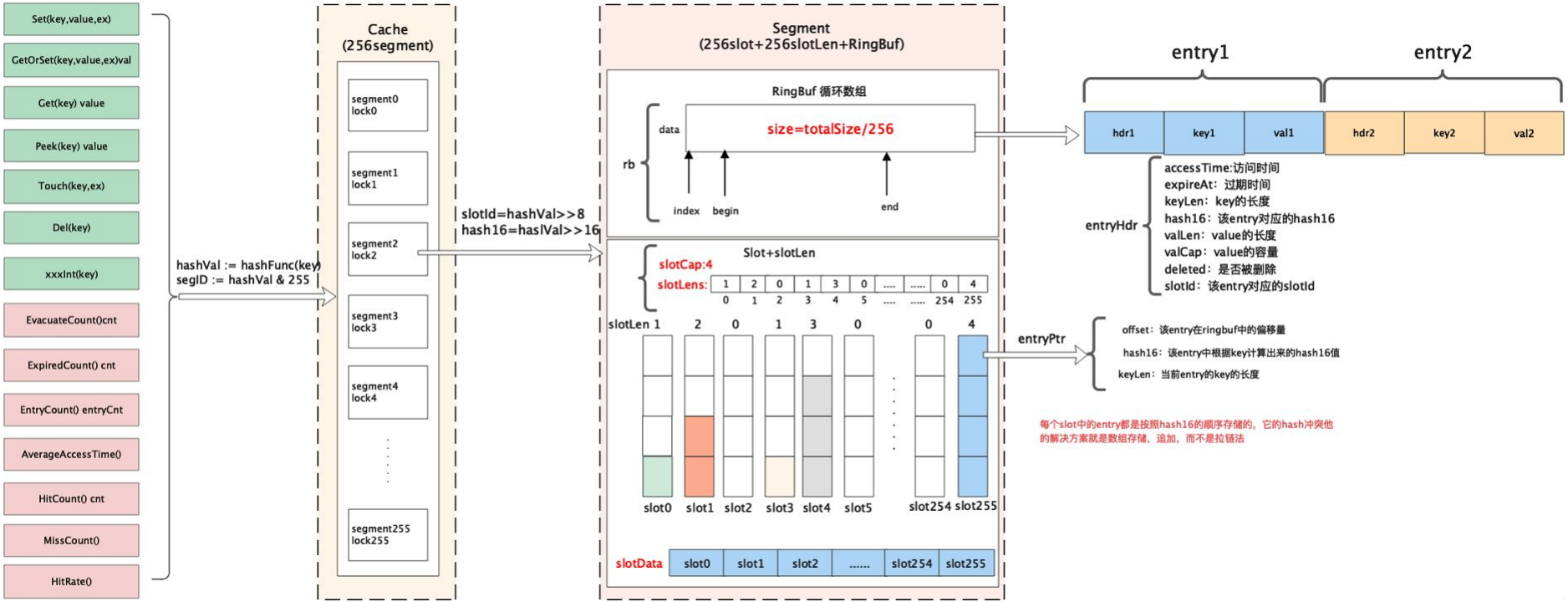
实现高并发思路

- 数据分片(降低锁的粒度)

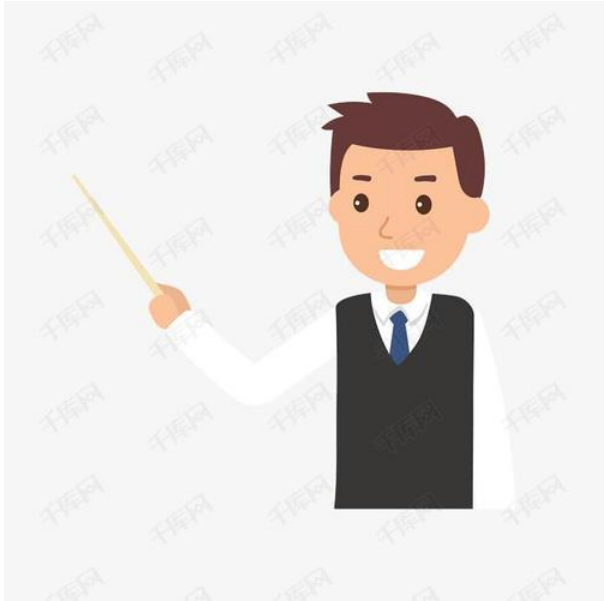
目录



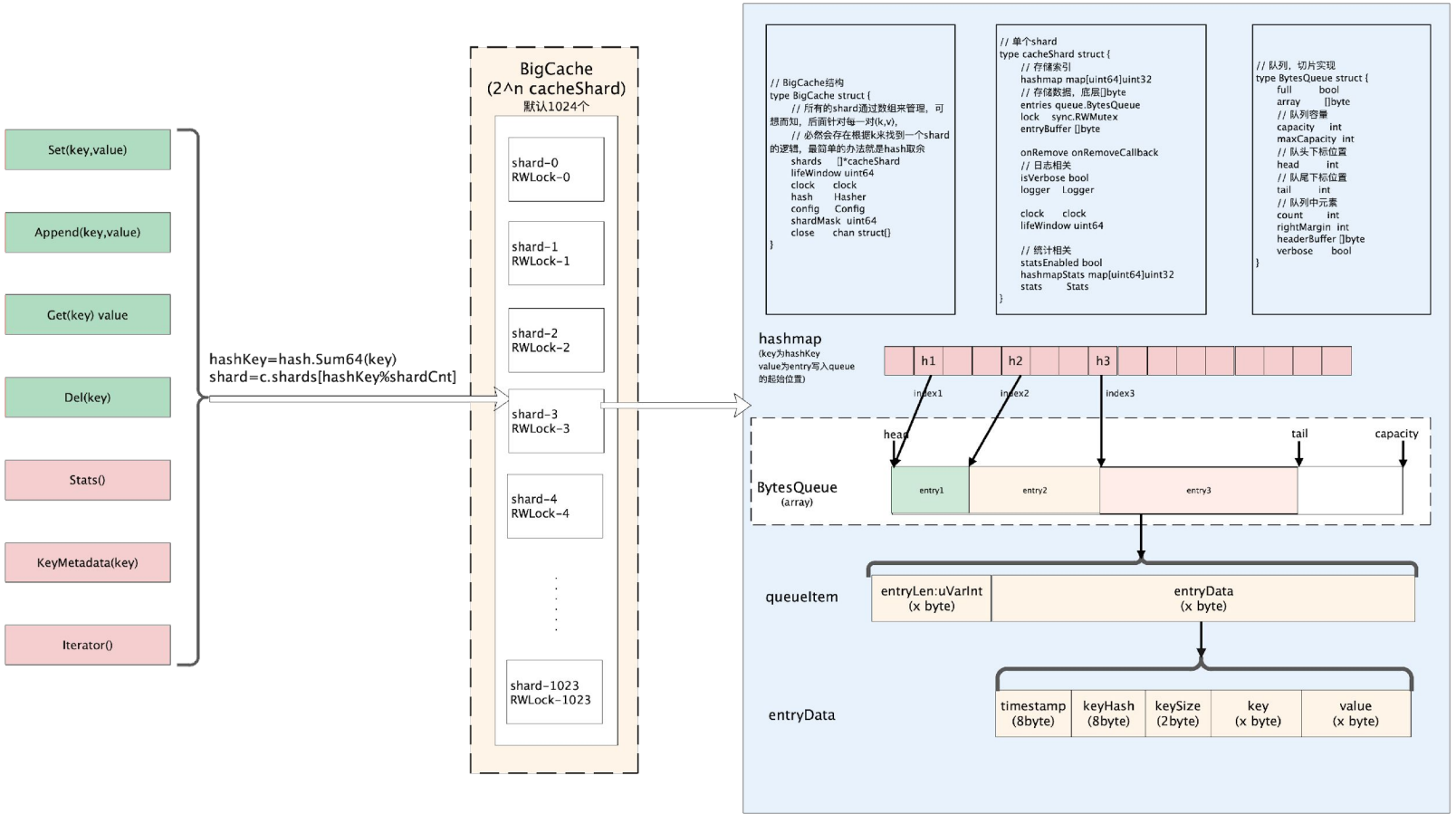
FreeCache原理



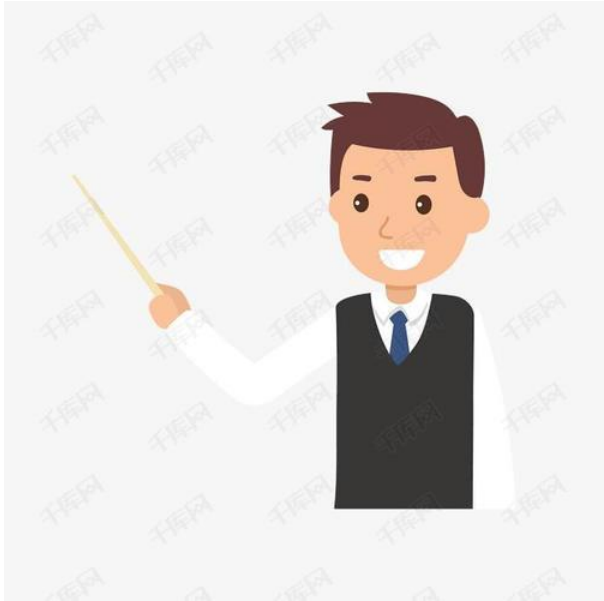
目录



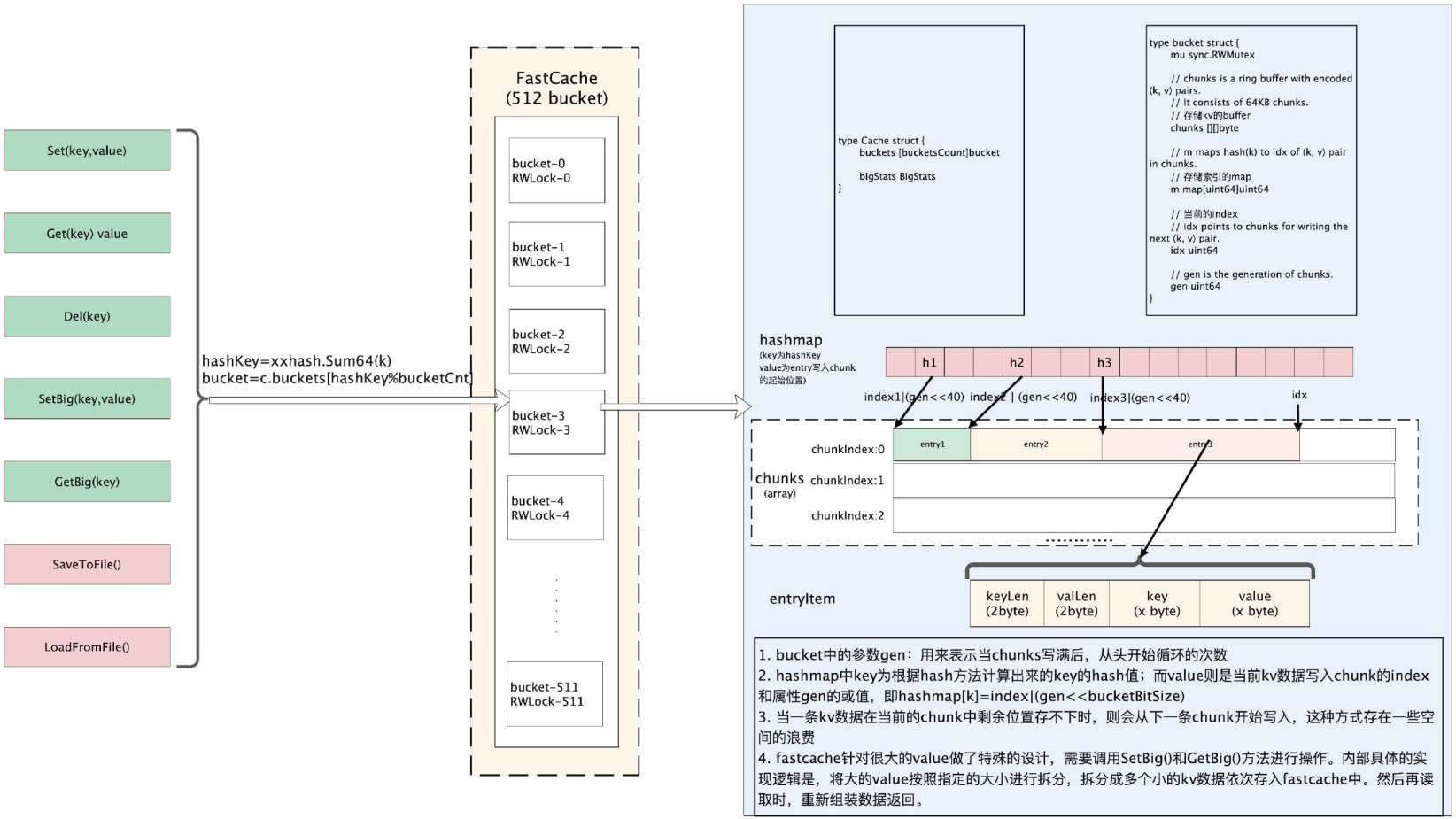
BigCache原理


Set(key,value)
Append(key,value)
Get(key) value
Del(key)
Stats()
KeyMetadata(key)
Iterator()
hashKey=hash.Sum64(key)
shard=c.shards[hashKey%shardCnt]
BigCache
(2^n cacheShard)
默认1024个
shard-0
RWLock-0
shard-1
RWLock-1
shard-2
RWLock-2
shard-3
RWLock-3
shard-4
RWLock-4
shard-1023
RWLock-1023
// BigCache结构
type BigCache struct {
// 所有的shard通过数组来管理，可想而知，后面针对每一对(k,v),
// 必然会存在根据k来找到一个shard的逻辑，最简单的办法就是hash取余
shards []*cacheShard
lifeWindow uint64
clock clock
hash Hasher
config Config
shardMask uint64
close chan struct{}
}
// 单个shard
type cacheShard struct {
// 存储索引
hashmap map[uint64]uint32
// 存储数据，底层[]byte
entries queue.BytesQueue
lock sync.RWMutex
entryBuffer []byte
onRemove onRemoveCallback
// 日志相关
isVerbose bool
logger Logger
clock clock
lifeWindow uint64
// 统计相关
statsEnabled bool
hashmapStats map[uint64]uint32
stats Stats
}
// 队列，切片实现
type BytesQueue struct {
full bool
array []byte
// 队列容量
capacity int
maxCapacity int
// 队头下标位置
head int
// 队尾下标位置
tail int
// 队列中元素
count int
rightMargin int
headerBuffer []byte
verbose bool
}
hashmap
(key为hashKey
value为entry写入queue
的起始位置)
h1
h2
h3
index1
index2
index3
head
tail
capacity
BytesQueue
(array)
entry1
entry2
entry3
queueItem
entryLen:uVarInt
(x byte)
entryData
(x byte)
entryData
timestamp
(8byte)
keyHash
(8byte)
keySize
(2byte)
key
(x byte)
value
(x byte)

目录

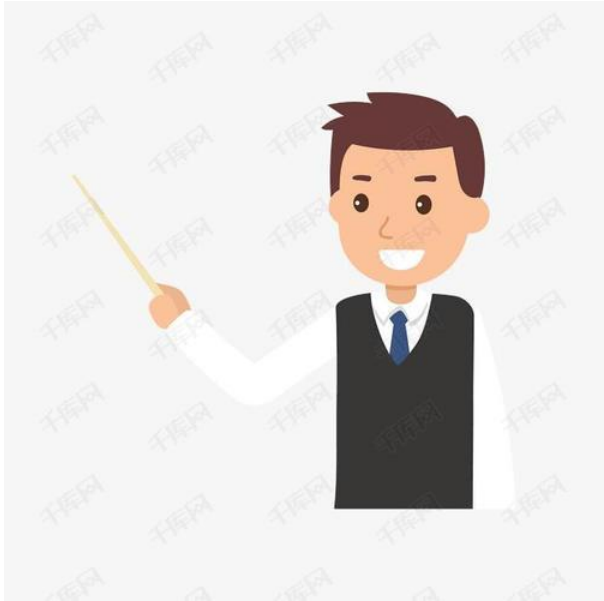


FastCache原理

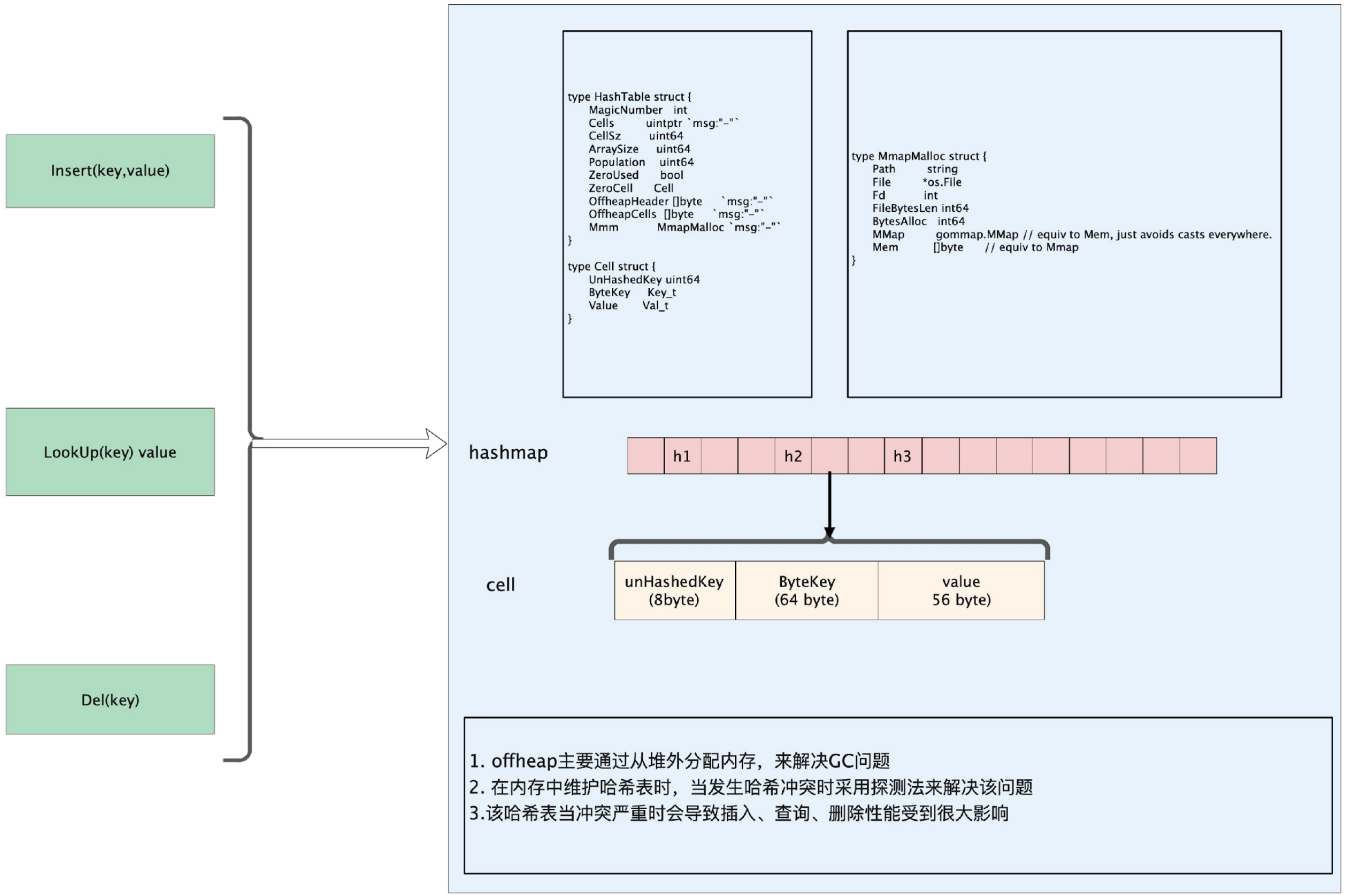


1. bucket中的参数gen：用来表示当chunks写满后，从头开始循环的次数
2. hashmap中key为根据hash方法计算出来的key的hash值；而value则是当前kv数据写入chunk的index和属性gen的或值，即hashmap[k]=index|(gen<<bucketBitSize)
3. 当一条kv数据在当前的chunk中剩余位置存不下时，则会从下一条chunk开始写入，这种方式存在一些空间的浪费
4. fastcache针对很大的value做了特殊的设计，需要调用SetBig()和GetBig()方法进行操作。内部具体的实现逻辑是，将大的value按照指定的大小进行拆分，拆分成多个小的kv数据依次存入fastcache中。然后再读取时，重新组装数据返回。

目录



Offheap原理


Insert(key,value)
LookUp(key) value
Del(key)
type HashTable struct {
MagicNumber int
Cells uintptr `msg:"-"`
CellSz uint64
ArraySize uint64
Population uint64
ZeroUsed bool
ZeroCell Cell
OffheapHeader []byte `msg:"-"`
OffheapCells []byte `msg:"-"`
Mmm MmapMalloc `msg:"-"`
}
type Cell struct {
UnHashedKey uint64
ByteKey Key_t
Value Val_t
}
type MmapMalloc struct {
Path string
File *os.File
Fd int
FileBytesLen int64
BytesAlloc int64
MMap gommap.MMap // equiv to Mem, just avoids casts everywhere.
Mem []byte // equiv to Mmap
}
hashmap
h1
h2
h3
cell
unHashedKey (8byte)
ByteKey (64 byte)
value 56 byte)
1. offheap主要通过从堆外分配内存，来解决GC问题
2. 在内存中维护哈希表时，当发生哈希冲突时采用探测法来解决该问题
3.该哈希表当冲突严重时会导致插入、查询、删除性能受到很大影响

总结

目录