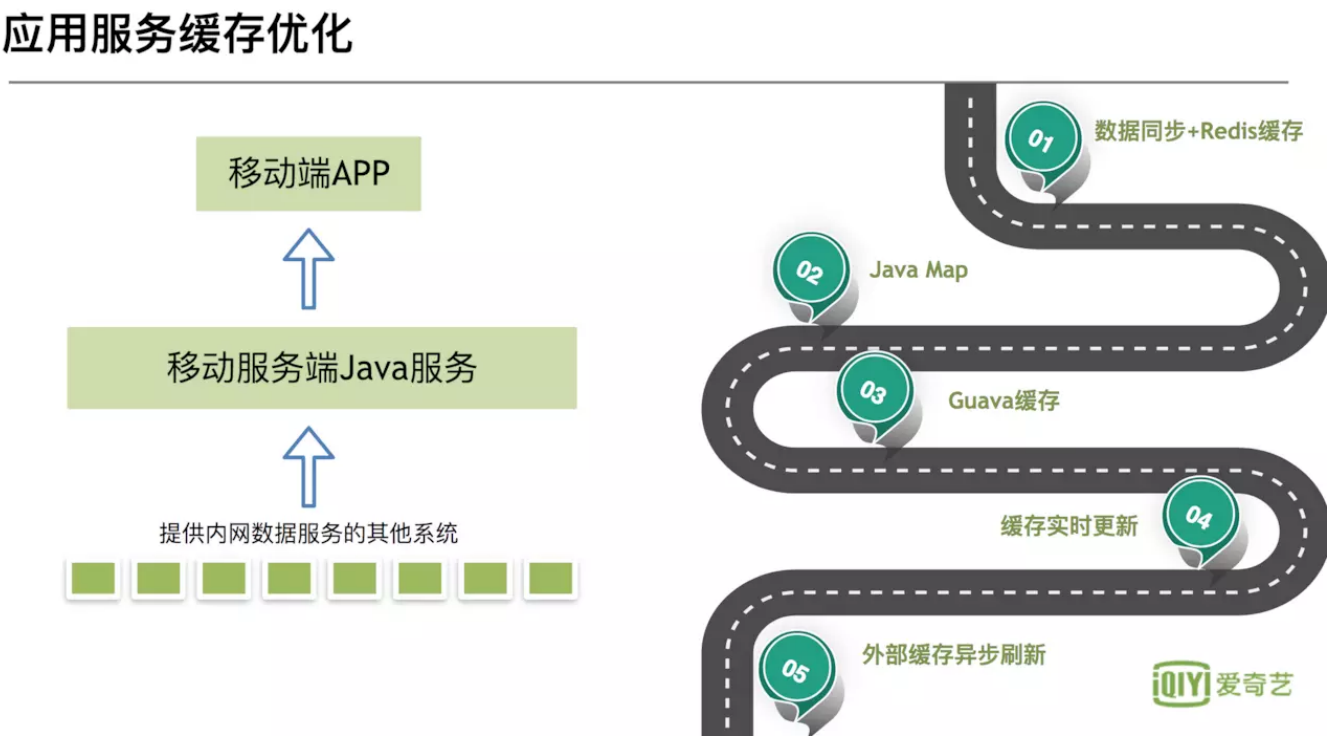
**咖啡拿铁**
2018年08月16日 阅读 5300

关注

# 你应该知道的缓存进化史

## 1.背景

本文是上周去技术沙龙听了一下爱奇艺的Java缓存之路有感写出来的。先简单介绍一下爱奇艺的java缓存道路的发展吧。




可以看见图中分为几个阶段:

- 第一阶段:数据同步加redis

通过消息队列进行数据同步至redis，然后Java应用直接去取缓存 这个阶段优点是:由于是使用的分布式缓存，所以数据更新快。缺点也比较明显:依赖Redis的稳定性，一旦redis挂了，整个缓存系统不可用，造成缓存雪崩，所有请求打到DB。

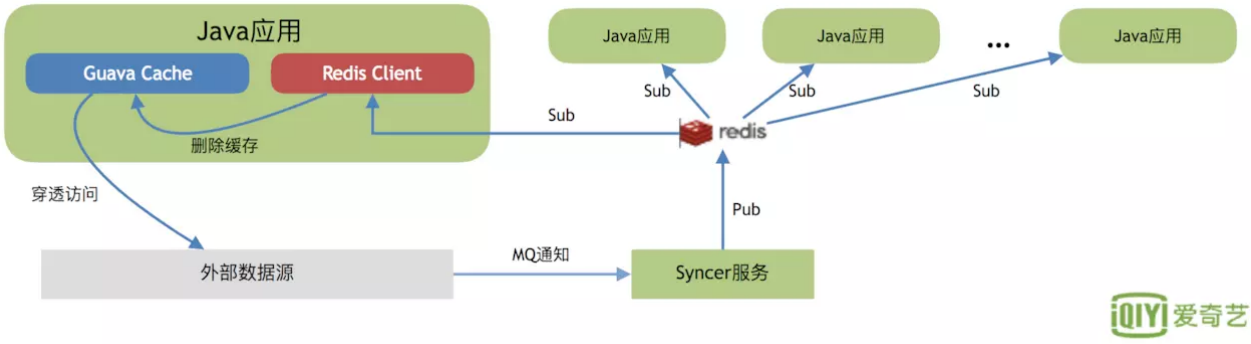
- 第二，三阶段:JavaMap到Guava cache

直大小，然后有些缓存会被淘汰，就会有命中率的问题。

- 第四阶段: Guava Cache刷新

为了解决上面的问题，利用Guava Cache可以设置写后刷新时间，进行刷新。解决了一直不更新的问题，但是依然没有解决实时刷新。

- 第五阶段: 外部缓存异步刷新




这个阶段扩展了Guava Cache,利用redis作为消息队列通知机制，通知其他java应用程序进行刷新。

这里简单介绍一下爱奇艺缓存发展的五个阶段，当然还有一些其他的优化，比如GC调优，缓存穿透，缓存覆盖的一些优化等等。有兴趣的同学可以关注公众号，联系我进行交流。

# 原始社会 - 查库

上面说的是爱奇艺的一个进化线路，但是在大家的一般开发过程中，第一步一般都没有redis，而是直接查库。

在流量不大的时候，查数据库或者读取文件是最为方便，也能完全满足我们的业务要求。

# 古代社会 - HashMap

当我们应用有一定流量之后或者查询数据库特别频繁，这个时候就可以祭出我们的java中自带的HashMap或者ConcurrentHashMap。我们可以在代码中这么写:

```
public class CustomerService {
    private HashMap<String,String> hashMap = new HashMap<>();
```

```
            if ( customer == null){
                customer = customerMapper.get(name);
                hashMap.put(name,customer);
            }
            return customer;
        }
    }
```

但是这样做就有个问题HashMap无法进行数据淘汰，内存会无限制的增长，所以hashMap很快也被淘汰了。当然并不是说他完全就没用，就像我们古代社会也不是所有的东西都是过时的，比如我们中华名族的传统美德是永不过时的，就像这个hashMap一样的可以在某些场景下作为缓存，当不需要淘汰机制的时候，比如我们利用反射，如果我们每次都通过反射去搜索Method,field，性能必定低效，这时我们用HashMap将其缓存起来，性能能提升很多。

## 近代社会 - LRUHashMap

在古代社会中难住我们的问题无法进行数据淘汰，这样会导致我们内存无限膨胀,显然我们是不可以接受的。有人就说我把一些数据给淘汰掉呗，这样不就对了，但是怎么淘汰呢？随机淘汰吗？当然不行，试想一下你刚把A装载进缓存，下一次要访问的时候就被淘汰了，那又会访问我们的数据库了，那我们要缓存干嘛呢？

所以聪明的人们就发明了几种淘汰算法，下面列举下常见的三种FIFO,LRU,LFU（还有一些ARC,MRU感兴趣的可以自行搜索）:

- FIFO:先进先出，在这种淘汰算法中，先进入缓存的会先被淘汰。这种可谓是最简单的了，但是会导致我们命中率很低。试想一下我们如果有个访问频率很高的数据是所有数据第一个访问的，而那些不是很高的是后面再访问的，那这样就会把我们的首个数据但是他的访问频率很高给挤出。
- LRU:最近最少使用算法。在这种算法中避免了上面的问题，每次访问数据都会将其放在我们的队尾，如果需要淘汰数据，就只需要淘汰队首即可。但是这个依然有个问题，如果有个数据在1个小时的前59分钟访问了1万次(可见这是个热点数据),再后一分钟没有访问这个数据，但是有其他的数据访问，就导致了我们这个热点数据被淘汰。
- LFU:最近最少频率使用。在这种算法中又对上面进行了优化，利用额外的空间记录每个数据的使用频率，然后选出频率最低进行淘汰。这样就避免了LRU不能处理时间段的问题。

上面列举了三种淘汰策略，对于这三种，实现成本是一个比一个高，同样的命中率也是一个比一个好。而我们一般来说选择的方案居中即可，即实现成本不是太高，而命中率也还行的LRU,如何实现一个LRUMap呢？我们可以通过继承LinkedHashMap，重写removeEldestEntry方法，即可完成一个简单的LRUMap。

```
    private final int max;
    private Object lock;

    public LRUMap(int max, Object lock) {
        //无需扩容
        super((int) (max * 1.4f), 0.75f, true);
        this.max = max;
        this.lock = lock;
    }

    /**
     * 重写LinkedHashMap的removeEldestEntry方法即可
     * 在Put的时候判断，如果为true，就会删除最老的
     * @param eldest
     * @return
     */
    @Override
    protected boolean removeEldestEntry(Map.Entry eldest) {
        return size() > max;
    }

    public Object getValue(Object key) {
        synchronized (lock) {
            return get(key);
        }
    }
    public void putValue(Object key, Object value) {
        synchronized (lock) {
            put(key, value);
        }
    }



    public boolean removeValue(Object key) {
        synchronized (lock) {
            return remove(key) != null;
        }
    }
    public boolean removeAll(){
        clear();
        return true;
    }
}
```

的大小特意设置到max*1.4，在下面的removeEldestEntry方法中只需要size>max就淘汰，这样我们这个map永远也走不到扩容的逻辑了，通过重写LinkedHashMap，几个简单的方法我们实现了我们的LruMap。

## 现代社会 - Guava cache

在近代社会中已经发明出来了LRUMap,用来进行缓存数据的淘汰，但是有几个问题:

- 锁竞争严重，可以看见我的代码中，Lock是全局锁，在方法级别上面的，当调用量较大时，性能必然会比较低。
- 不支持过期时间
- 不支持自动刷新

所以谷歌的大佬们对于这些问题，按捺不住了，发明了Guava cache，在Guava cache中你可以如下面的代码一样，轻松使用:

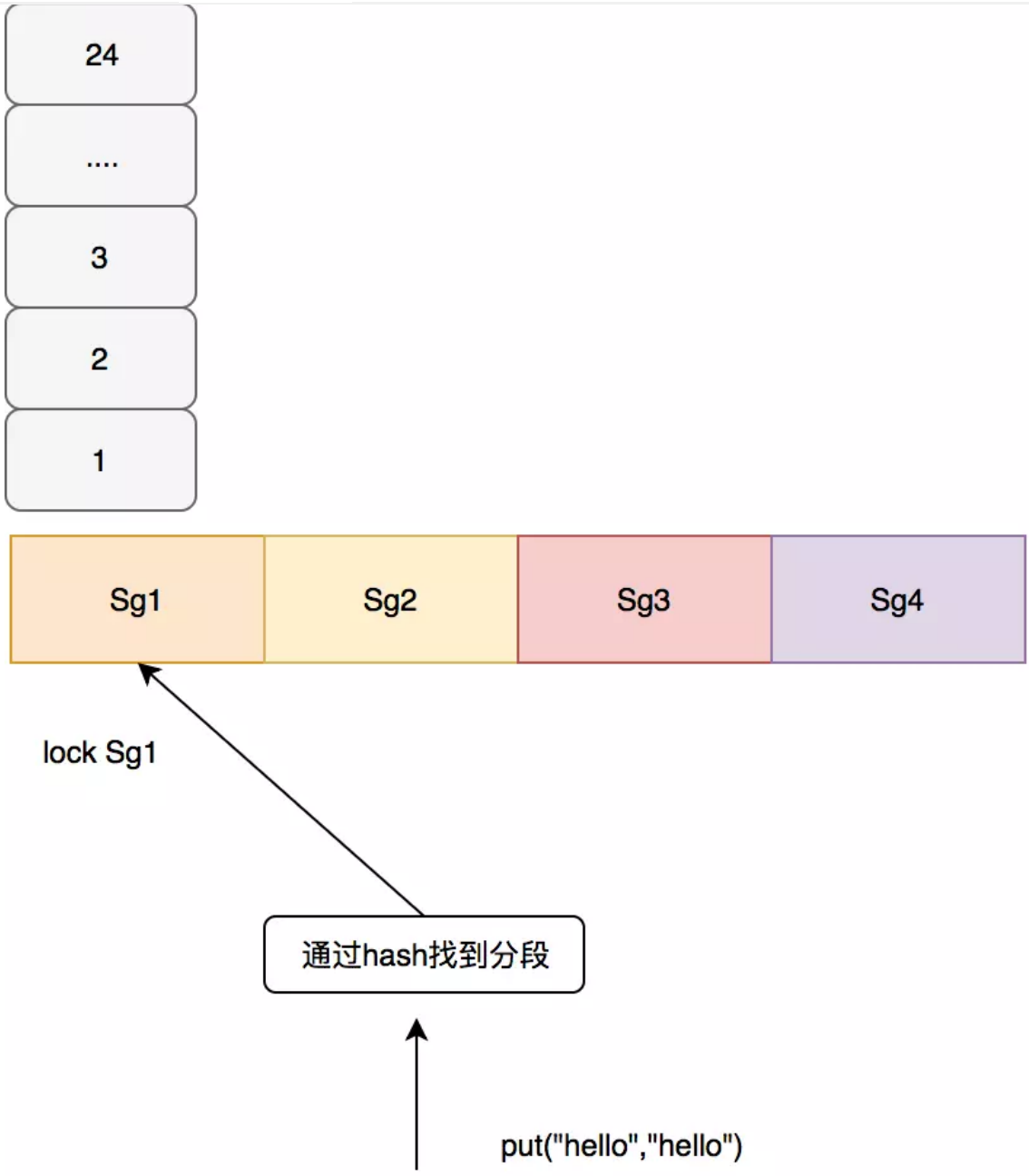
```
public static void main(String[] args) throws ExecutionException {
        LoadingCache<String, String> cache = CacheBuilder.newBuilder()
                .maximumSize(100)
                //写之后30ms过期
                .expireAfterWrite(30L, TimeUnit.MILLISECONDS)
                //访问之后30ms过期
                .expireAfterAccess(30L, TimeUnit.MILLISECONDS)
                //20ms之后刷新
                .refreshAfterWrite(20L, TimeUnit.MILLISECONDS)
                //开启weakKey key 当启动垃圾回收时，该缓存也被回收
                .weakKeys()
                .build(createCacheLoader());
        System.out.println(cache.get("hello"));
        cache.put("hello1", "我是hello1");
        System.out.println(cache.get("hello1"));
        cache.put("hello1", "我是hello2");
        System.out.println(cache.get("hello1"));
    }
    public static com.google.common.cache.CacheLoader<String, String> createCacheLoader() {
        return new com.google.common.cache.CacheLoader<String, String>() {
            @Override
            public String load(String key) throws Exception {
                return key;
            }
        };
    }
```

## 锁竞争

guava cache采用了类似ConcurrentHashMap的思想，分段加锁，在每个段里面各自负责自己的淘汰的事情。在Guava根据一定的算法进行分段，这里要说明的是，如果段太少那竞争依然很严重，如果段太多会容易出现随机淘汰，比如大小为100的，给他分100个段，那也就是让每个数据都独占一个段，而每个段会自己处理淘汰的过程，所以会出现随机淘汰。在guava cache中通过如下代码，计算出应该如何分段。

```
int segmentShift = 0;
int segmentCount = 1;
while (segmentCount < concurrencyLevel && (!evictsBySize() || segmentCount * 20 <= maxWeight))
  ++segmentShift;
  segmentCount <<= 1;
}
```

上面segmentCount就是我们最后的分段数，其保证了每个段至少10个Entry。如果没有设置concurrencyLevel这个参数，那么默认就会是4，最后分段数也最多为4，例如我们size为100，会分为4段，每段最大的size是25。 在guava cache中对于写操作直接加锁，对于读操作，如果读取的数据没有过期，且已经加载就绪，不需要进行加锁，如果没有读到会再次加锁进行二次读，如果还没有需要进行缓存加载，也就是通过我们配置的CacheLoader，我这里配置的是直接返回Key，在业务中通常配置从数据库中查询。 如下图所示:




## 过期时间

相比于LRUMap多了两种过期时间，一个是写后多久过期expireAfterWrite，一个是读后多久过期expireAfterAccess。很有意思的事情是，在guava cache中对于过期的Entry并没有马上过期(也就是并没有后台线程一直在扫)，而是通过进行读写操作的时候进行过期处理，这样做的好处是避免后台线程扫描的时候进行全局加锁。看下面的代码:

```
public static void main(String[] args) throws ExecutionException, InterruptedException {
        Cache<String, String> cache = CacheBuilder.newBuilder()
```

```
                .concurrencyLevel(1)
                .build();
        cache.put("hello1", "我是hello1");
        cache.put("hello2", "我是hello2");
        cache.put("hello3", "我是hello3");
        cache.put("hello4", "我是hello4");
        //至少睡眠5ms
        Thread.sleep(5);
        System.out.println(cache.size());
        cache.put("hello5", "我是hello5");
        System.out.println(cache.size());
    }
输出:
4
1
```

从这个结果中我们知道，在put的时候才进行的过期处理。特别注意的是我上面concurrencyLevel(1)我这里将分段最大设置为1，不然不会出现这个实验效果的，在上面一节中已经说过，我们是以段位单位进行过期处理。在每个Segment中维护了两个队列:

```
    final Queue<ReferenceEntry<K, V>> writeQueue;

    final Queue<ReferenceEntry<K, V>> accessQueue;
```

writeQueue维护了写队列，队头代表着写得早的数据，队尾代表写得晚的数据。 accessQueue维护了访问队列,和LRU一样，用来我们进行访问时间的淘汰，如果当这个Segment超过最大容量，比如我们上面所说的25，超过之后，就会把accessQueue这个队列的第一个元素进行淘汰。

```
void expireEntries(long now) {
      drainRecencyQueue();

      ReferenceEntry<K, V> e;
      while ((e = writeQueue.peek()) != null && map.isExpired(e, now)) {
        if (!removeEntry(e, e.getHash(), RemovalCause.EXPIRED)) {
          throw new AssertionError();
        }
      }
      while ((e = accessQueue.peek()) != null && map.isExpired(e, now)) {
        if (!removeEntry(e, e.getHash(), RemovalCause.EXPIRED)) {
```

```
    }
```

上面就是guava cache处理过期Entries的过程，会对两个队列一次进行peek操作，如果过期就进行删除。一般处理过期Entries可以在我们的put操作的前后，或者读取数据时发现过期了，然后进行整个Segment的过期处理，又或者进行二次读lockedGetOrLoad操作的时候调用。

```
void evictEntries(ReferenceEntry<K, V> newest) {
      ///... 省略无用代码

      while (totalWeight > maxSegmentWeight) {
        ReferenceEntry<K, V> e = getNextEvictable();
        if (!removeEntry(e, e.getHash(), RemovalCause.SIZE)) {
          throw new AssertionError();
        }
      }
    }
/**
**返回accessQueue的entry
**/
ReferenceEntry<K, V> getNextEvictable() {
      for (ReferenceEntry<K, V> e : accessQueue) {
        int weight = e.getValueReference().getWeight();
        if (weight > 0) {
          return e;
        }
      }
      throw new AssertionError();
    }
```

上面是我们驱逐Entry的时候的代码，可以看见访问的是accessQueue对其队头进行驱逐。而驱逐策略一般是在对segment中的元素发生变化时进行调用，比如插入操作，更新操作，加载数据操作。

## 自动刷新

自动刷新操作，在guava cache中实现相对比较简单，直接通过查询，判断其是否满足刷新条件，进行刷新。

## 其他特性

### 虚引用

在Guava cache中，key和value都能进行虚引用的设定，在Segment中的有两个引用队列:

```
final @Nullable ReferenceQueue<K> keyReferenceQueue;

final @Nullable ReferenceQueue<V> valueReferenceQueue;
```

这两个队列用来记录被回收的引用，其中每个队列记录了每个被回收的Entry的hash，这样回收了之后通过这个队列中的hash值就能把以前的Entry进行删除。

### 删除监听器

在guava cache中，当有数据被淘汰时，但是你不知道他到底是过期，还是被驱逐，还是因为虚引用的对象被回收？这个时候你可以调用这个方法removalListener(RemovalListener listener)添加监听器进行数据淘汰的监听，可以打日志或者一些其他处理，可以用来进行数据淘汰分析。

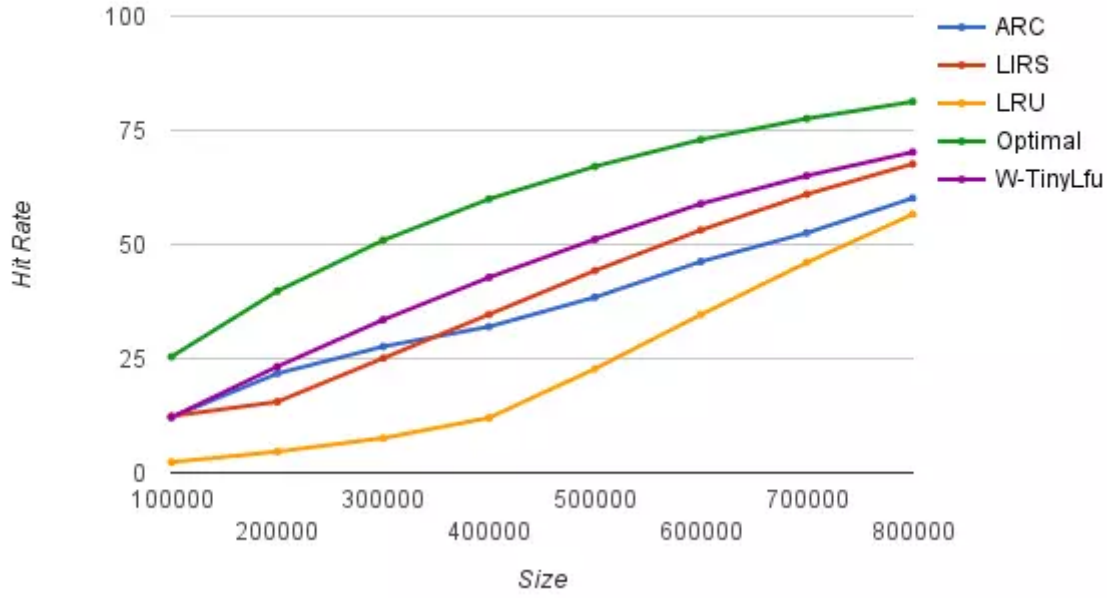
> 在RemovalCause记录了所有被淘汰的原因:被用户删除，被用户替代，过期，驱逐收集，由于大小淘汰。

## guava cache的总结

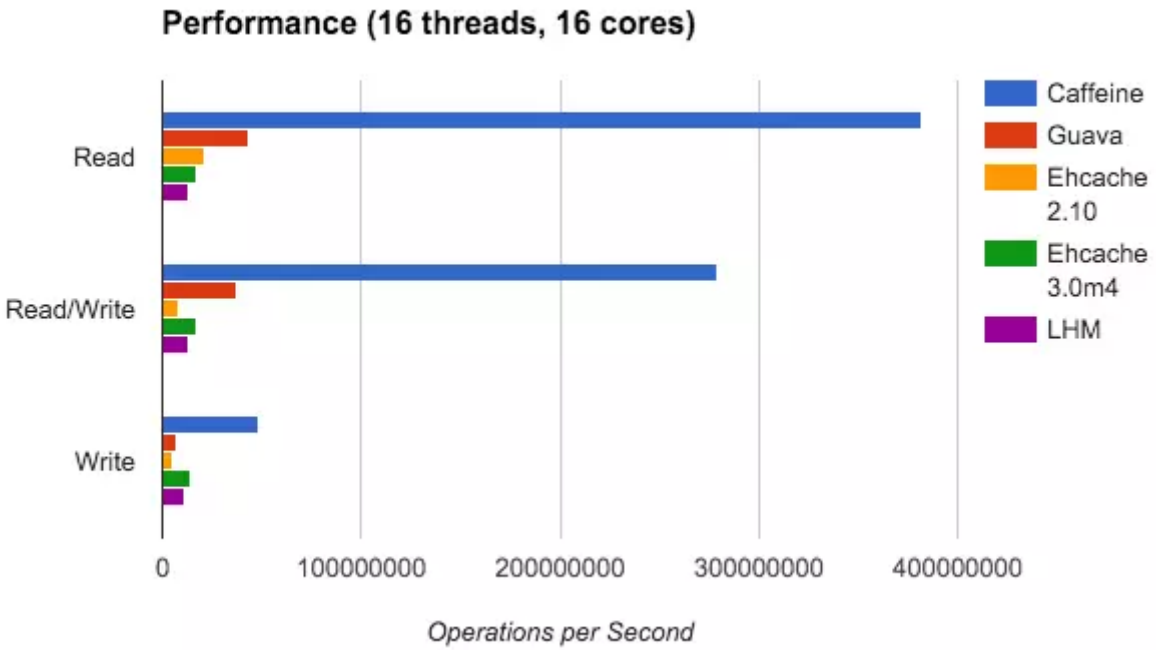
细细品读guava cache的源码总结下来，其实就是一个性能不错的，api丰富的LRU Map。爱奇艺的缓存的发展也是基于此之上，通过对guava cache的二次开发，让其可以进行java应用服务之间的缓存更新。

# 走向未来-caffeine

guava cache的功能的确是很强大，满足了绝大多数的人的需求，但是其本质上还是LRU的一层封装，所以在众多其他较为优良的淘汰算法中就相形见绌了。而caffeine cache实现了W-TinyLFU(LFU+LRU算法的变种)。下面是不同算法的命中率的比较:




其中Optimal是最理想的命中率，LRU和其他算法相比的确是个弟弟。而我们的W-TinyLFU 是最接近理想命中率的。当然不仅仅是命中率caffeine优于了guava cache，在读写吞吐量上面也是完爆guava cache。




这个时候你肯定会好奇为啥这么caffeine这么牛逼呢？别着急下面慢慢给你道来。

## W-TinyLFU

下来，这部新剧在这几天大概访问了几亿次，这个访问频率也在我们的LFU中记录了几亿次。但是新剧总会过气的，比如一个月之后这个新剧的前几集其实已经过气了，但是他的访问量的确是太高了，其他的电视剧根本无法淘汰这个新剧，所以在这种模式下是有局限性。所以各种LFU的变种出现了，基于时间周期进行衰减，或者在最近某个时间段内的频率。同样的LFU也会使用额外空间记录每一个数据访问的频率，即使数据没有在缓存中也需要记录，所以需要维护的额外空间很大。

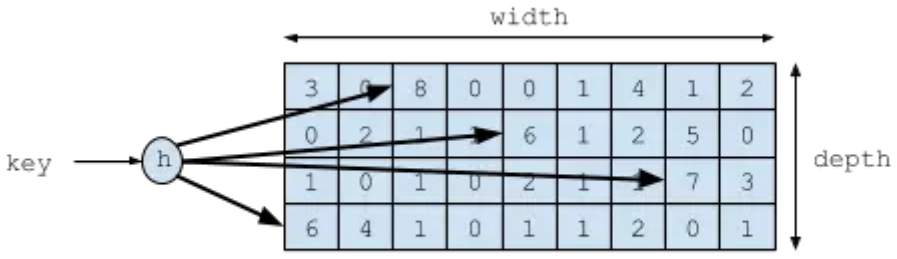
> 可以试想我们对这个维护空间建立一个hashMap，每个数据项都会存在这个hashMap中，当数据量特别大的时候，这个hashMap也会特别大。

再回到LRU，我们的LRU也不是那么一无是处，LRU可以很好的应对突发流量的情况，因为他不需要累计数据频率。

所以W-TinyLFU结合了LRU和LFU，以及其他的算法的一些特点。

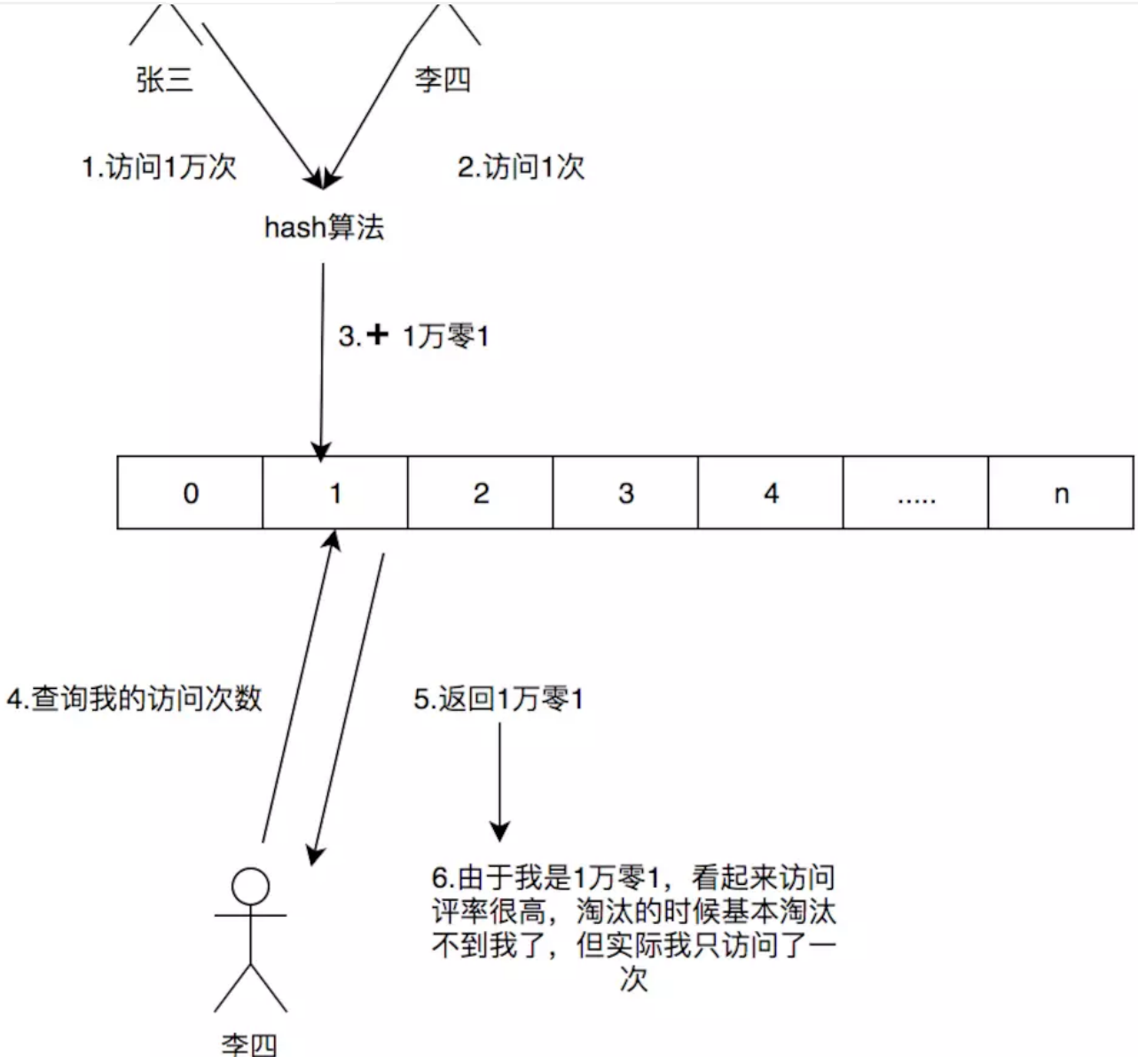
## 频率记录

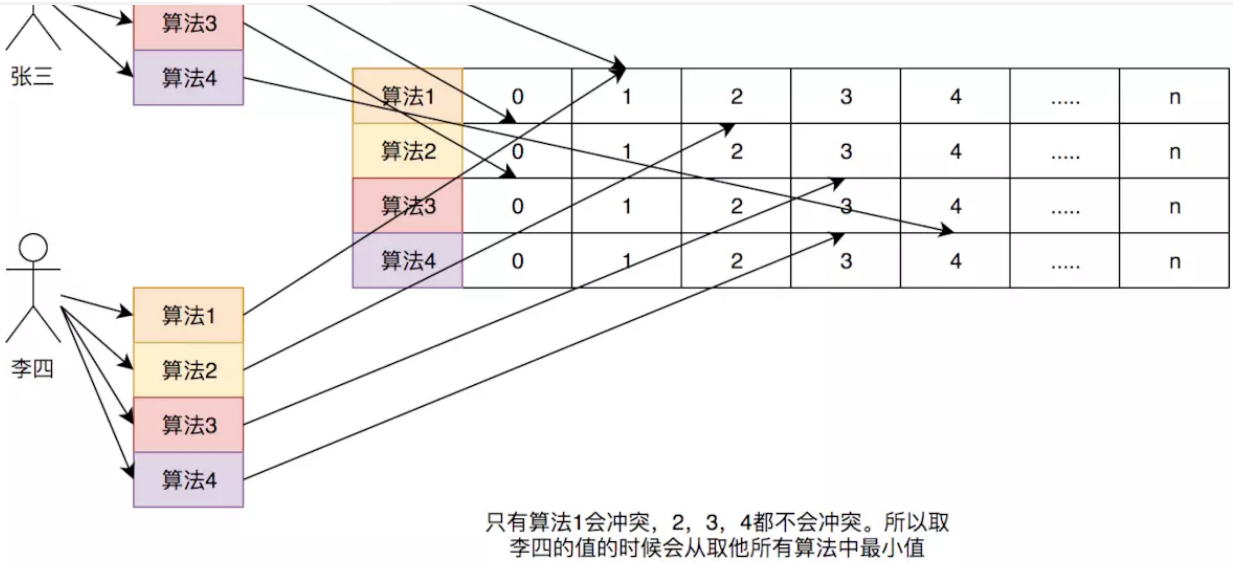
首先要说到的就是频率记录的问题，我们要实现的目标是利用有限的空间可以记录随时间变化的访问频率。在W-TinyLFU中使用Count-Min Sketch记录我们的访问频率，而这个也是布隆过滤器的一种变种。如下图所示:




**CountMin Sketch**

如果需要记录一个值，那我们需要通过多种Hash算法对其进行处理hash，然后在对应的hash算法的记录中+1，为什么需要多种hash算法呢？由于这是一个压缩算法必定会出现冲突，比如我们建立一个Long的数组，通过计算出每个数据的hash的位置。比如张三和李四，他们两有可能hash值都是相同，比如都是1那Long[1]这个位置就会增加相应的频率，张三访问1万次，李四访问1次那Long[1]这个位置就是1万零1，如果取李四的访问评率的时候就会取出是1万零1，但是李四命名只访问了1次啊，为了解决这个问题，所以用了多个hash算法可以理解为long[][]二维数组的一个概念，比如在第一个算法张三和李四冲突了，但是在第二个，第三个中很大的概率不冲突，比如一个算法大概有[illegible]概率冲突，那四个算法一起冲突的概率是1%的四次方。通过这个模式我们取李四的访问率的时候[illegible]所有算法中，李四访问最低频率的次数。所以他的名字叫Count-Min Sketch。

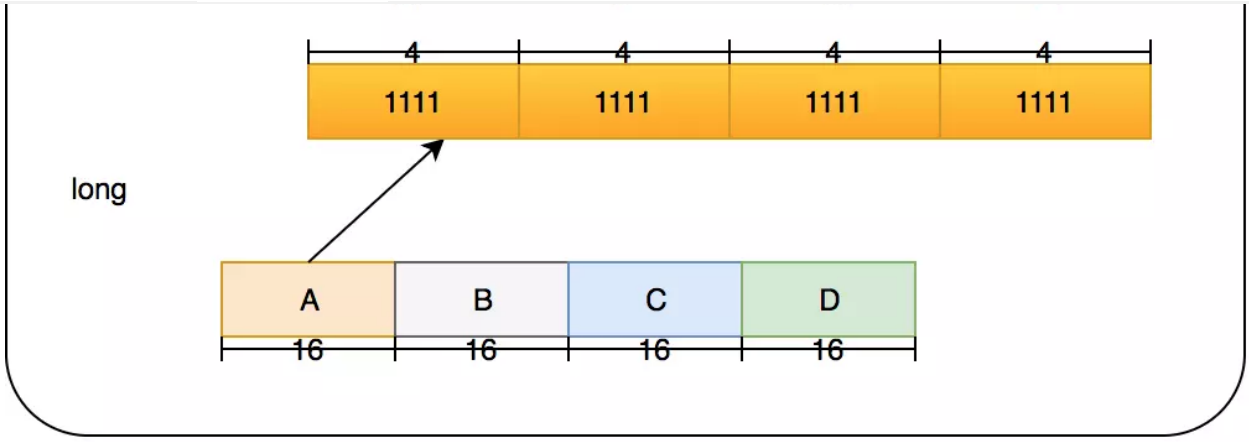

张三
李四
1.访问1万次
2.访问1次
hash算法
3.+ 1万零1
0
1
2
3
4
.....
n
4.查询我的访问次数
5.返回1万零1
6.由于我是1万零1，看起来访问评率很高，淘汰的时候基本淘汰不到我了，但实际我只访问了一次
李四




这里和以前的做个对比，简单的举个例子:如果一个hashMap来记录这个频率，如果我有100个数据，那这个HashMap就得存储100个这个数据的访问频率。哪怕我这个缓存的容量是1，因为Lfu的规则我必须全部记录这个100个数据的访问频率。如果有更多的数据我就有记录更多的。

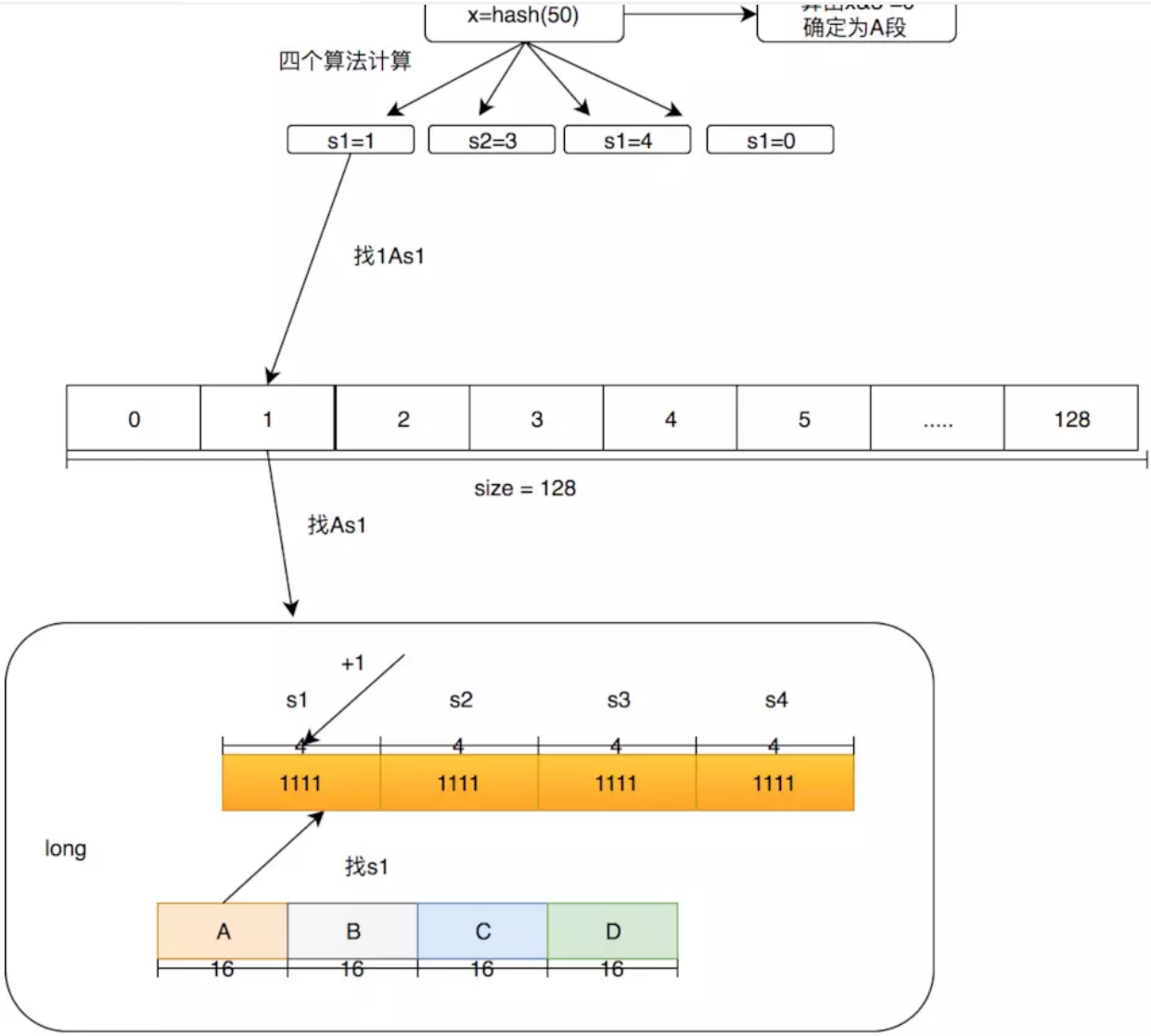
在Count-Min Sketch中，我这里直接说caffeine中的实现吧(在FrequencySketch这个类中),如果你的缓存大小是100，他会生成一个long数组大小是和100最接近的2的幂的数，也就是128。而这个数组将会记录我们的访问频率。在caffeine中他规则频率最大为15，15的二进制位1111，总共是4位，而Long型是64位。所以每个Long型可以放16种算法，但是caffeine并没有这么做，只用了四种hash算法，每个Long型被分为四段，每段里面保存的是四个算法的频率。这样做的好处是可以进一步减少Hash冲突，原先128大小的hash，就变成了128X4。

一个Long的结构如下:




我们的4个段分为A,B,C,D，在后面我也会这么叫它们。而每个段里面的四个算法我叫他s1,s2,s3,s4。下面举个例子如果要添加一个访问50的数字频率应该怎么做？我们这里用size=100来举例。

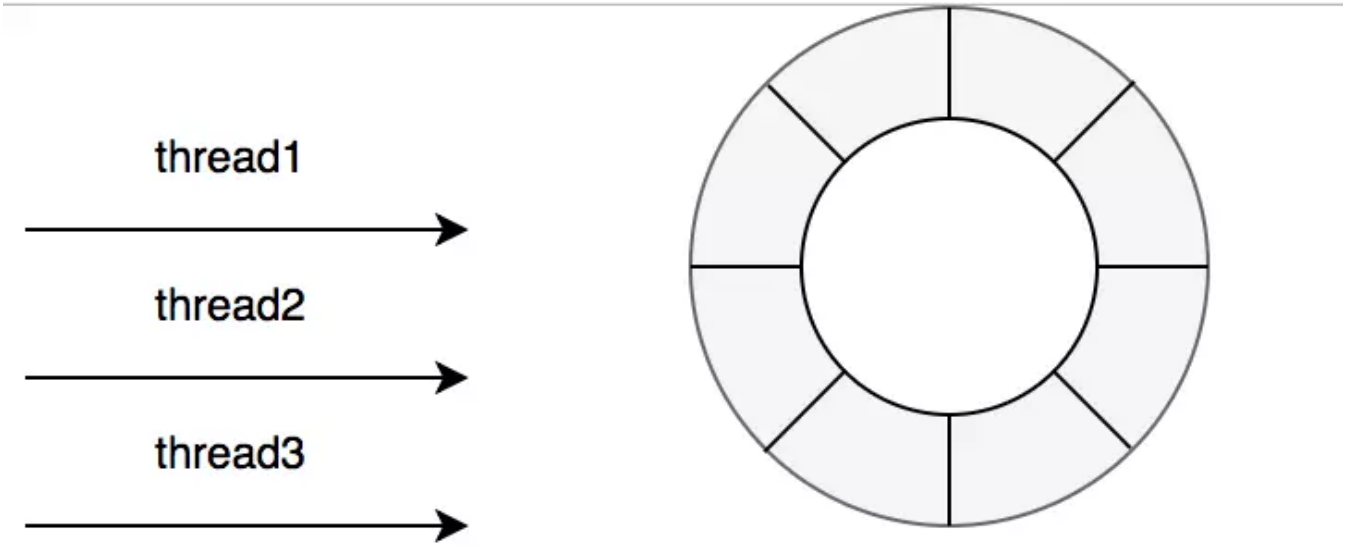
1. 首先确定50这个hash是在哪个段里面，通过hash & 3必定能获得小于4的数字，假设hash & 3=0，那就在A段。
2. 对50的hash再用其他hash算法再做一次hash，得到long数组的位置。假设用s1算法得到1，s2算法得到3，s3算法得到4，s4算法得到0。
3. 然后在long[1]的A段里面的s1位置进行+1,简称1As1加1，然后在3As2加1，在4As3加1，在0As4加1。



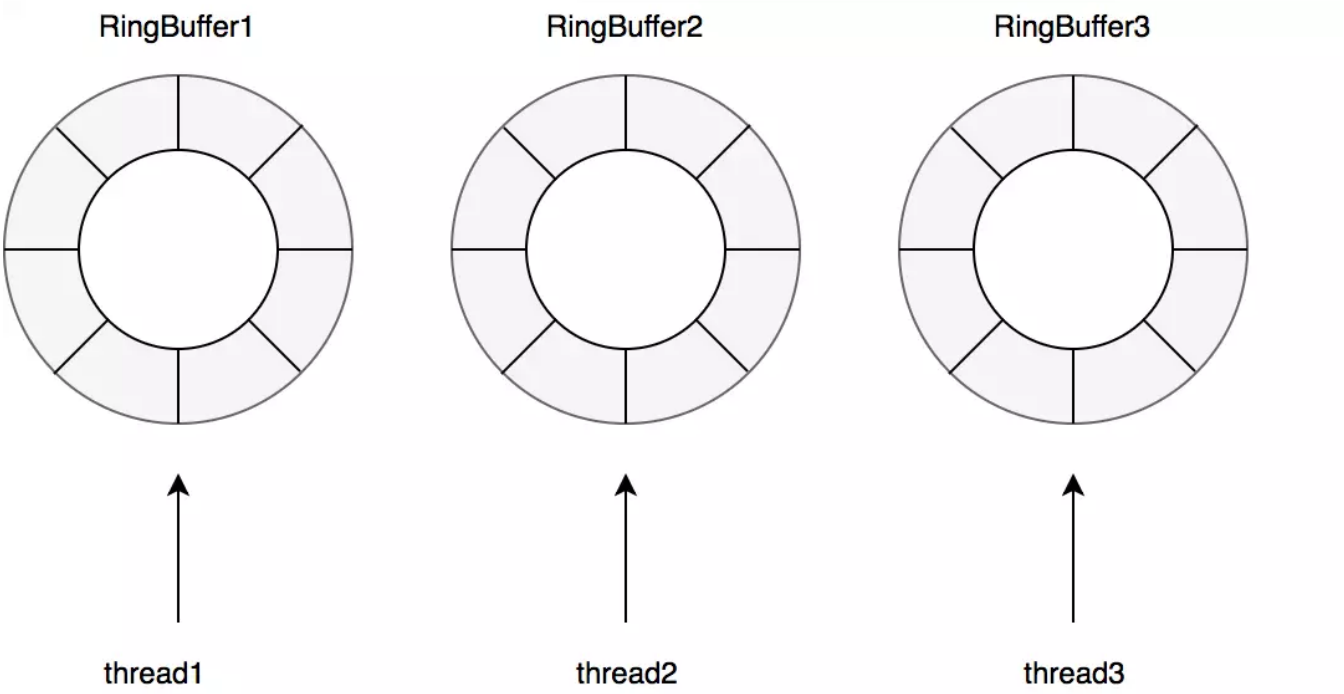

这个时候有人会质疑频率最大为15的这个是否太小？没关系在这个算法中，比如size等于100，如果他全局提升了1000次就会全局除以2衰减，衰减之后也可以继续增加，这个算法再W-TinyLFU的论文中证明了其可以较好的适应时间段的访问频率。

## 读写性能

在guava cache中我们说过其读写操作中夹杂着过期时间的处理，也就是你在一次Put操作中有可能还会做淘汰操作，所以其读写性能会受到一定影响，可以看上面的图中，caffeine的确在读写操作上面完爆guava cache。主要是因为在caffeine，对这些事件的操作是通过异步操作，他将事件提交至队列，这里的队列的数据结构是RingBuffer,不清楚的可以看看这篇文章,你应该知道的高性能无锁队列Disruptor。然后通过会通过默认的ForkJoinPool.commonPool()，或者自己配置线程池，进行队列操作，然后在进行后续的淘汰，过期操作。




对于读操作比写操作更加频繁，进一步减少竞争，其为每个线程配备了一个RingBuffer：



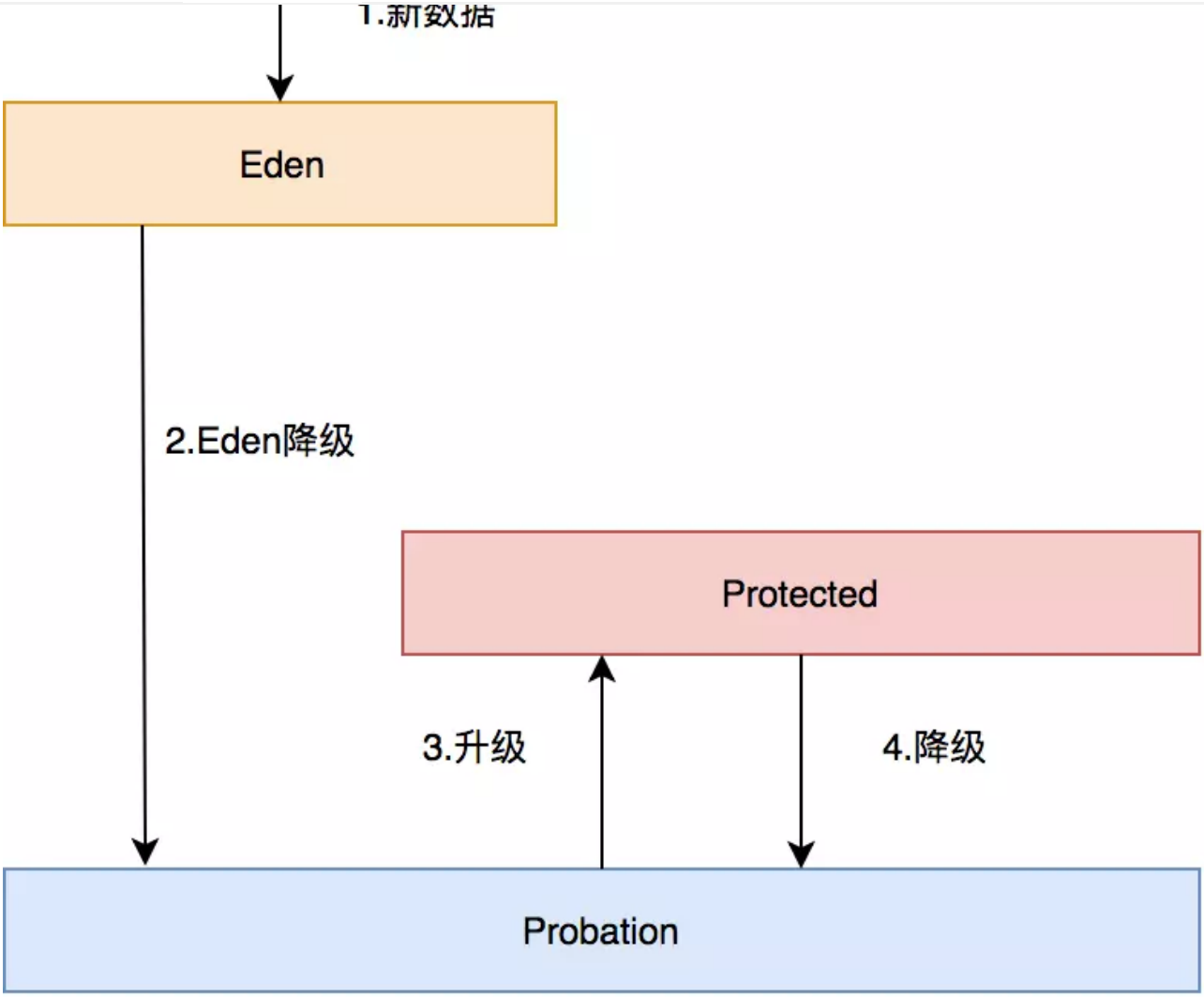

## 数据淘汰策略

- Eden队列:在caffeine中规定只能为缓存容量的%1,如果size=100,那这个队列的有效大小就等于1。这个队列中记录的是新到的数据，防止突发流量由于之前没有访问频率，而导致被淘汰。比如有一部新剧上线，在最开始其实是没有访问频率的，防止上线之后被其他缓存淘汰出去，而加入这个区域。伊甸区，最舒服最安逸的区域，在这里很难被其他数据淘汰。

- Probation队列:叫做缓刑队列，在这个队列就代表你的数据相对比较冷，马上就要被淘汰了。这个有效大小为size减去eden减去protected。

- Protected队列:在这个队列中，可以稍微放心一下了，你暂时不会被淘汰，但是别急，如果Probation队列没有数据了或者Protected数据满了，你也将会被面临淘汰的尴尬局面。当然想要变成这个队列，需要把Probation访问一次之后，就会提升为Protected队列。这个有效大小为(size减去eden) X 80% 如果size =100，就会是79。

这三个队列关系如下:




1. 所有的新数据都会进入Eden。
2. Eden满了，淘汰进入Probation。
3. 如果在Probation中访问了其中某个数据，则这个数据升级为Protected。
4. 如果Protected满了又会继续降级为Probation。

对于发生数据淘汰的时候，会从Probation中进行淘汰，会把这个队列中的数据队头称为受害者，这个队头肯定是最早进入的，按照LRU队列的算法的话那他其实他就应该被淘汰，但是在这里只能叫他受害者，这个队列是缓刑队列，代表马上要给他行刑了。这里会取出队尾叫候选者，也叫攻击者。这里受害者会和攻击者做PK，通过我们的Count-Min Sketch中的记录的频率数据有以下几个判断:

- 如果攻击者大于受害者，那么受害者就直接被淘汰。
- 如果攻击者<=5，那么直接淘汰攻击者。这个逻辑在他的注释中有解释:

```
    // exploits that a hot candidate is rejected in favor of a hot victim. The threshold of a warm
    // candidate reduces the number of random acceptances to minimize the impact on the hit rate.
    return false;
  }
  int random = ThreadLocalRandom.current().nextInt();
  return ((random & 127) == 0);
```

他认为设置一个预热的门槛会让整体命中率更高。

- 其他情况，随机淘汰。

## 如何使用

对于熟悉Guava的玩家来说如果担心有切换成本，那么你完全就多虑了，caffeine的api借鉴了Guava的api，可以发现其基本一模一样。

```
public static void main(String[] args) {
        Cache<String, String> cache = Caffeine.newBuilder()
                .expireAfterWrite(1, TimeUnit.SECONDS)
                .expireAfterAccess(1,TimeUnit.SECONDS)
                .maximumSize(10)
                .build();
        cache.put("hello","hello");
    }
```

顺便一提的是，越来越多的开源框架都放弃了Guava cache，比如Spring5。在业务上我也自己曾经比较过Guava cache和caffeine最终选择了caffeine，在线上也有不错的效果。所以不用担心caffeine不成熟，没人使用。

# 最后

本文主要讲了爱奇艺的缓存之路和本地缓存的一个发展历史(从古至今到未来），以及每一种缓存的实现基本原理。当然要使用好缓存光是这些仅仅不够，比如本地缓存如何在其他地方更改了之后同步更新，分布式缓存，多级缓存等等。后面也会专门写一节介绍这个如何用好缓存。对于Guava cache和caffeine的原理后面也会专门抽出时间写这两个的源码分析，如果感兴趣的朋友可以关注公众号第一时间查阅更新文章。

最后这篇文章被我收录于JGrowing，一个全面，优秀，由社区一起共建的Java学习路线，如果您[illegible]与开源项目的维护，可以一起共建，github地址为:github.com/javagrowing... 麻烦给个小星星[illegible]


**关注下面的标签，发现更多相似文章**

Redis 后端 Java 算法

**咖啡拿铁** 后端开发 公众号:【咖啡拿铁】 @ 猿辅导
获得点赞 4,733 次 · 文章被阅读 119,421 次

关注



## 评论

输入评论...

雨季不再来 JAVA开发

本地缓存更多的还是一些数据不太变化的场景的应用，能使用到缓存来抵抗db，肯定服务的qps也是比较大的，如果本地缓存比较多，也是会对服务的gc有压力，还有每次服务发布都需要预热数据吧

7月前 回复

7月前

咖啡拿铁 (作者) 后端开发 公众号:...
回复 咖啡拿铁 (作者): 但是如果特别多变的话 就不适合 了
7月前

加载更多

linkerlin
多级缓存比单级大缓存更有效。
7月前 回复

咖啡拿铁 (作者) 后端开发 公众号:...
回复 linkerlin: 是的
7月前

xkenmon
赞，写得非常清晰^_^
7月前 回复

星_星hh
赞，最近正在用分布式缓存
7月前 回复

leetomlee123
网页链接
7月前 回复

咖啡拿铁 (作者) 后端开发 公众号:...
回复 leetomlee123: 这都能有广告
7月前

## 相关推荐

专栏 · 陈续渊 · 1天前 · 后端 / Spring Cloud
**[Spring Cloud Tutorial翻译系列]微服务-定义、原则、好处**
2 5

专栏 · 埃里克拓荒 · 14小时前 · 算法

专栏 · 吾乃上将军邢道荣 · 14小时前 · 后端

**SpringBoot踩坑日记-一个非空校验引发的bug**

2

专栏 · lijiangtao · 17小时前 · Java

**Java并发 之 线程组 ThreadGroup 介绍**

2

热 · 专栏 · 石杉的架构笔记 · 2天前 · Java

**尴尬的面试现场：说说你们系统有多大QPS？系统到底怎么抗住高并发的？【石杉的架构笔记】**

148 37

专栏 · Sophie May · 16小时前 · Java

**Java多线程之Callable,Future,FutureTask**

1

专栏 · 深夜里的程序猿 · 22小时前 · Redis

**Redis的正确使用姿势**

27

专栏 · 架构师修行之路 · 18小时前 · 算法

**程序猿修仙之路--算法之选择排序**

1

专栏 · 金空空 · 19小时前 · Java

**Java填坑系列之LinkedList**

专栏 · 掘金翻译计划 · 16小时前 · 后端 / 掘金翻译计划

**[译] 多线程简介：一步一步来接近多线程的世界**

19 1