课程介绍

缓存回顾：

使用场景：

互联网，数据越来越多，用户越来越多，并发量、吞吐量越来越大

1、使用数据库存储，分库分表，也不能满足要求，使用缓存，减轻数据库的压力

2、临时存储的数据

3、其他的场景：Session分离、分布式锁

缓存：服务器端

本地缓存（local cache）

CurrentHashMap、Ehcache、Guava Cache

缓存在应用服务器，全局变量，JVM缓存

优势：和应用程序在同一进程，访问快，没有网络开销，一般不会崩

缺点：容量小，每个JVM有一份，有数据冗余

分布式缓存

Redis、Memcached、Tair（阿里、美团）、EVCache（AWS）、Aerospike

优势：

优势：空间优势、在应用中共享（session共享）、高可用（主从）、高扩展（分区）、集群

缺点：资源、网络开销、数据一致性（AP）

前面我们已经详细的讲解了Redis，接下来我们要讲讲其他的缓存

GuavaCache、Tair、EVCache、Aerospike比较

	GuavaCache	Tair	EVCache	Aerospike
类别	本地缓存（JVM缓存）	分布式缓存	分布式缓存	分布式NoSQL数据库
应用	高并发本地缓存	阿里、美团	Netflix、AWS	互联网广告行业（国外）
性能	高	较高	很高	较高
持久化	无	有	有	有
集群	无	有	有	有

Guava Cache

Guava Cache介绍

JVM缓存

JVM 缓存，是堆缓存。其实就是创建一些全局容器，比如List、Set、Map等。

这些容器用来做数据存储。

这样做的问题：

不能按照一定的规则淘汰数据，如 `LRU，LFU，FIFO` 等。

清除数据时的回调通知

并发处理能力差，针对并发可以使用CurrentHashMap，但缓存的其他功能需要自行实现

缓存过期处理，缓存数据加载刷新等都需要手工实现

Guava Cache

Guava是Google提供的一套Java工具包，而Guava Cache是一套非常完善的本地缓存机制（JVM缓存）。

Guava cache的设计来源于CurrentHashMap，可以按照多种策略来清理存储在其中的缓存值且保持很高的并发读写性能。

Guava Cache应用场景

本地缓存的应用场景：

- 对性能有非常高的要求
- 不经常变化
- 占用内存不大
- 有访问整个集合的需求
- 数据允许不时时一致

guava cache
高并发，不需要持久化

currentHashMap
高并发

Ehcached
持久化 二级缓存

Guava Cache 的优势：

- 缓存过期和淘汰机制

  在GuavaCache中可以设置Key的过期时间，包括访问过期和创建过期

  GuavaCache在缓存容量达到指定大小时，采用LRU的方式，将不常使用的键值从Cache中删除

- 并发处理能力

  GuavaCache类似CurrentHashMap，是线程安全的。

  提供了设置并发级别的api，使得缓存支持并发的写入和读取

  采用分离锁机制，分离锁能够减小锁力度，提升并发能力

  分离锁是分拆锁定，把一个集合看分成若干partition, 每个partiton一把锁。ConcurrentHashMap就是分了16个区域，这16个区域之间是可以并发的。GuavaCache采用Segment做分区。

- 更新锁定

  一般情况下，在缓存中查询某个key，如果不存在，则查源数据，并回填缓存。（Cache Aside Pattern）

  在高并发下会出现，多次查源并重复回填缓存，可能会造成源的宕机（DB），性能下降

  GuavaCache可以在CacheLoader的load方法中加以控制，对同一个key，只让一个请求去读源并回填缓存，其他请求阻塞等待。

- 集成数据源

  一般我们在业务中操作缓存，都会操作缓存和数据源两部分

  而GuavaCache的get可以集成数据源，在从缓存中读取不到时可以从数据源中读取数据并回填缓存

- 监控缓存加载/命中情况

  统计

Guava Cache创建方式

GuavaCache有两种创建方式：

CacheLoader和Callable callback

```
public class GuavaDemo {
    public  static  void main(String args[]) throws Exception {
        LoadingCache<String,Object> cache = CacheBuilder.newBuilder()
                // 最大3个        //Cache中存储的对象,写入3秒后过期
                .maximumSize(3).expireAfterWrite(3,
                    //记录命中率     ////失效通知
                 TimeUnit.SECONDS).recordStats().removalListener(new
RemovalListener<Object, Object>() {
                    public void onRemoval(RemovalNotification<Object, Object>
notification){

 System.out.println(notification.getKey()+":"+notification.getCause());
                    }
                }
                .build(
                        new CacheLoader<String, Object>() {
                            @Override
                            public String load(String s) throws Exception {

                                return Constants.hm.get(s);
                            }
                        }

                );
        /*
            初始化cache
         */
        initCache(cache);
        System.out.println(cache.size());
        displayCache(cache);
        System.out.println("============================");
        Thread.sleep(1000);
        System.out.println(cache.getIfPresent("1"));
        Thread.sleep(2500);
```

```
        System.out.println("============================");
        displayCache(cache);


    }

    public static Object get(String key,LoadingCache cache)throws Exception{
        Object value=cache.get(key, new Callable() {
            @Override
            public Object call() throws Exception {
                Object v= Constants.hm.get(key);
                //设置回缓存
                cache.put(key,v);
                return v;
            }
        });
        return value;
    }

    public static void initCache(LoadingCache cache)throws Exception{
        /*
            前三条记录
         */
        for(int i=1;i<=3;i++){
            cache.get(String.valueOf(i));
        }
    }

    /**
     * 获得当前缓存的记录
     * @param cache
     * @throws Exception
     */
    public static void displayCache(LoadingCache cache)throws Exception{

        Iterator its=cache.asMap().entrySet().iterator();
        while(its.hasNext()){
            System.out.println(its.next().toString());
        }



    }
}
```

CacheLoader

在创建cache对象时，采用CacheLoader来获取数据，当缓存不存在时能够自动加载数据到缓存中。

```
LoadingCache<String,String> cache = CacheBuilder.newBuilder()
                .maximumSize(3)
                .build(
                        new CacheLoader<String, String>() {
                            @Override
                            public String load(String s) throws Exception {
                                return Constants.hm.get(s);
                            }
                        }
                );
```

Callable Callback

```
    public static Object get(String key,LoadingCache cache)throws Exception{
        Object value=cache.get(key, new Callable() {
            @Override
            public Object call() throws Exception {
                String v= Constants.hm.get(key);
                //设置回缓存
                cache.put(key,v);
                return v;
            }
        });
        return value;
    }
```

缓存数据删除

GuavaCache的数据删除分为：被动删除和主动删除

被动删除

基于数据大小的删除

```
LoadingCache<String,Object> cache= CacheBuilder.newBuilder()
                /*
                     加附加的功能
                 */
                //最大个数
                .maximumSize(3)
                .build(new CacheLoader<String, Object>() {

            //读取数据源
            @Override
            public Object load(String key) throws Exception {
                return Constants.hm.get(key);
            }
        });

//读取缓存中的1的数据     缓存有就读取 没有就返回null
        System.out.println(cache.getIfPresent("5"));

        //读取4    读源并回写缓存   淘汰一个（LRU+FIFO）
```

```
        get("4",cache);
```

规则：LRU+FIFO

访问次数一样少的情况下，FIFO

基于过期时间的删除

隔多长时间后没有被访问过的key被删除

```
//缓存中的数据  如果3秒内没有访问则删除
                .maximumSize(3).expireAfterAccess(3, TimeUnit.SECONDS)
。。。。

        Thread.sleep(1000);
        //访问1    1被访问
        cache.getIfPresent("1");

        //歇了2.1秒
        Thread.sleep(2100);

        //最后缓存中会留下1
        System.out.println("================================");
        display(cache);
```

写入多长时间后过期

```
                    //等同于expire  ttl  缓存中对象的生命周期就是3秒
                .maximumSize(3).expireAfterWrite(3, TimeUnit.SECONDS)
                .build(new CacheLoader<String, Object>() {

            //读取数据源
            @Override
            public Object load(String key) throws Exception {
                return Constants.hm.get(key);
            }
        });

display(cache);
        Thread.sleep(1000);
        //访问1
        cache.getIfPresent("1");

        //歇了2.1秒
        Thread.sleep(2100);

        System.out.println("================================");
        display(cache);
```

基于引用的删除

可以通过weakKeys和weakValues方法指定Cache只保存对缓存记录key和value的弱引用。这样当没有其他强引用指向key和value时，key和value对象就会被垃圾回收器回收。

```
LoadingCache<String,Object> cache = CacheBuilder.newBuilder()
                //  最大3个        值的弱引用
                .maximumSize(3).weakValues()
                .build(

                );

Object value = new Object();
cache.put("1",value);
value = new Object();//原对象不再有强引用
//强制垃圾回收
System.gc();
System.out.println(cache.getIfPresent("1"));
```

主动删除

单独删除

```
 //将key=1 删除
 cache.invalidate("1");
```

批量删除

```
//将key=1和2的删除
cache.invalidateAll(Arrays.asList("1","2"));
```

清空所有数据

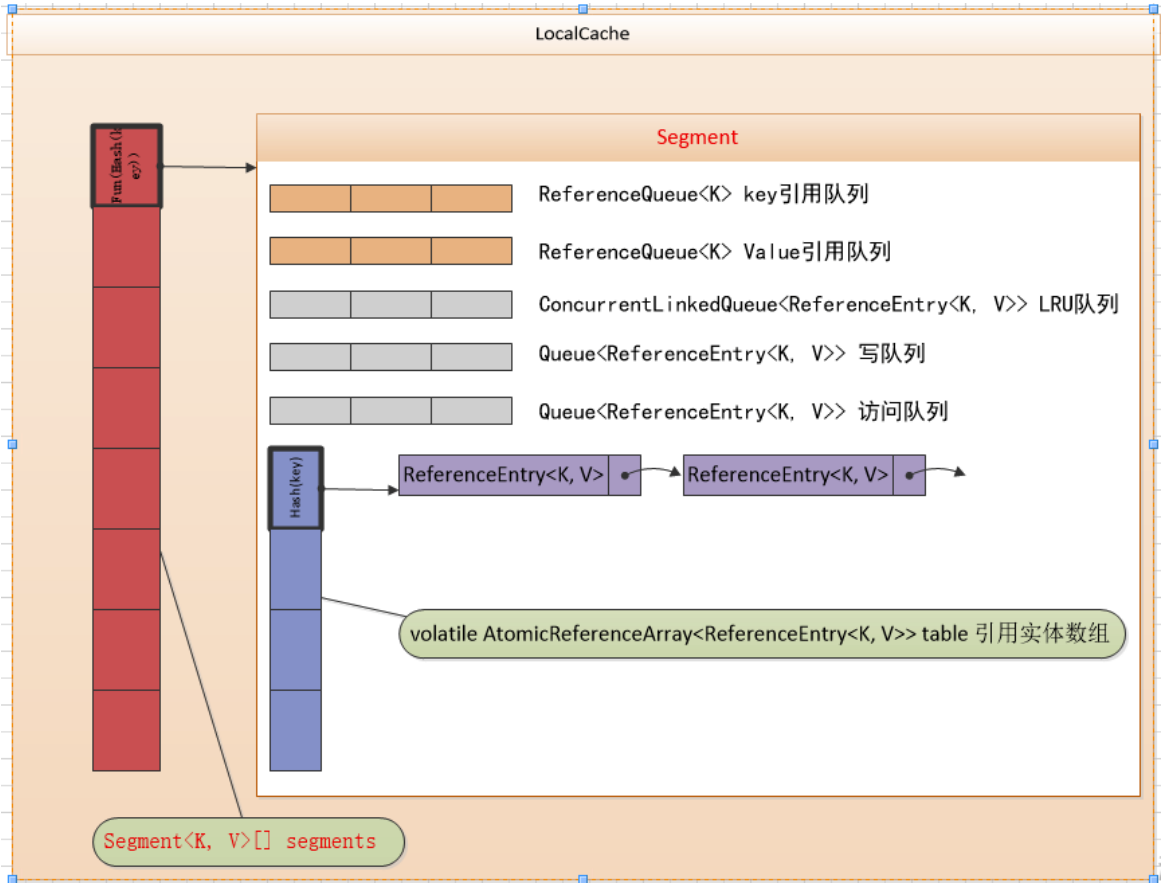
```
//清空缓存
cache.invalidateAll();
```

Guava Cache原理

GuavaCache核心原理之数据结构

Guava Cache的数据结构跟ConcurrentHashMap类似，但也不完全一样。最基本的区别是ConcurrentMap会一直保存所有添加的元素，直到显式地移除。

相对地，Guava Cache为了限制内存占用，通常都设定为自动回收元素。其数据结构图如下：



- LocalCache为Guava Cache的核心类，包含一个Segment数组组成
- Segement数组的长度决定了cache的并发数
- 每一个Segment使用了单独的锁，其实每个Segment继承了ReentrantLock，对Segment的写操作需要先拿到锁
- 每个Segment由一个table和5个队列组成
- 5个队列：

  ReferenceQueue keyReferenceQueue ： 已经被GC，需要内部清理的键引用队列

  ReferenceQueue valueReferenceQueue： 已经被GC，需要内部清理的值引用队列

  ConcurrentlinkedQueue<ReferenceEntry<k,v>> recencyQueue : LRU队列，当segment上达到临界值发生写操作时该队列会移除数据

  Queue<ReferenceEntry<K, V>> writeQueue：写队列，按照写入时间进行排序的元素队列，写入一个元素时会把它加入到队列尾部

  Queue<ReferenceEntry<K, V>> accessQueue：访问队列，按照访问时间进行排序的元素队列，访问(包括写入)一个元素时会把它加入到队列尾部

- 1个table：

  AtomicReferenceArray<ReferenceEntry<K, V>> table：AtomicReferenceArray可以用原子方式更新其元素的对象引用数组

- ReferenceEntry<k,v>

  ReferenceEntry是Guava Cache中对一个键值对节点的抽象，每个ReferenceEntry数组项都是一条ReferenceEntry链。并且一个ReferenceEntry包含key、hash、valueReference、next字段（单链）

  Guava Cache使用ReferenceEntry接口来封装一个键值对，而用ValueReference来封装Value值

GuavaCache核心原理之回收机制

Guava Cache提供了三种基本的缓存回收方式：

- 基于容量回收

  在缓存项的数目达到限定值之前，采用LRU的回收方式

- 定时回收

  expireAfterAccess：缓存项在给定时间内没有被读/写访问，则回收。回收顺序和基于大小回收一样（LRU）

  expireAfterWrite：缓存项在给定时间内没有被写访问（创建或覆盖），则回收

- 基于引用回收

  通过使用弱引用的键、或弱引用的值、或软引用的值，Guava Cache可以垃圾回收

除了以上三种还有主动删除，采用命令，这个前面讲了

GuavaCache构建的缓存不会"自动"执行清理和回收工作，也不会在某个缓存项过期后马上清理，也没有诸如此类的清理机制。

GuavaCache是在每次进行缓存操作的时候，惰性删除 如get()或者put()的时候，判断缓存是否过期

GuavaCache核心原理之Segment定位

先通过key做hash定位到所在的Segment

通过位运算找首地址的偏移量 SegmentCount>=并发数且为2的n次方

```
V get(K key, CacheLoader<? super K, V> loader) throws ExecutionException {
        // 注意，key不可为空
        int hash = hash(checkNotNull(key));
        // 通过hash定位到segment数组的某个Segment元素，然后调用其get方法
        return segmentFor(hash).get(key, hash, loader);
    }
```

再找到segment中的Entry链数组，通过key的hash定位到某个Entry节点

```
 V get(K key, int hash, CacheLoader<? super K, V> loader) throws
ExecutionException {
      checkNotNull(key);
      checkNotNull(loader);
      try {
        if (count != 0) { // read-volatile
          // 内部也是通过找Entry链数组定位到某个Entry节点
          ReferenceEntry<K, V> e = getEntry(key, hash);
          ......
```

Guava Cache高级实战

GuavaCache高级实战之并发操作

并发设置

GuavaCache通过设置 concurrencyLevel 使得缓存支持并发的写入和读取

```
LoadingCache<String,Object> cache = CacheBuilder.newBuilder()
// 最大3个        同时支持CPU核数线程写缓存
.maximumSize(3).concurrencyLevel(Runtime.getRuntime().availableProcessors()).bui
ld();
```

concurrencyLevel=Segment数组的长度

同ConcurrentHashMap类似Guava cache的并发也是通过分离锁实现

```
 V get(K key, CacheLoader<? super K, V> loader) throws ExecutionException {
        int hash = this.hash(Preconditions.checkNotNull(key));
        //通过hash值确定该key位于哪一个segment上，并获取该segment
        return this.segmentFor(hash).get(key, hash, loader);
    }
```

LoadingCache采用了类似ConcurrentHashMap的方式，将映射表分为多个segment。segment之间可以并发访问，这样可以大大提高并发的效率，使得并发冲突的可能性降低了。

更新锁定

GuavaCache提供了一个refreshAfterWrite定时刷新数据的配置项

如果经过一定时间没有更新或覆盖，则会在下一次获取该值的时候，会在后台异步去刷新缓存

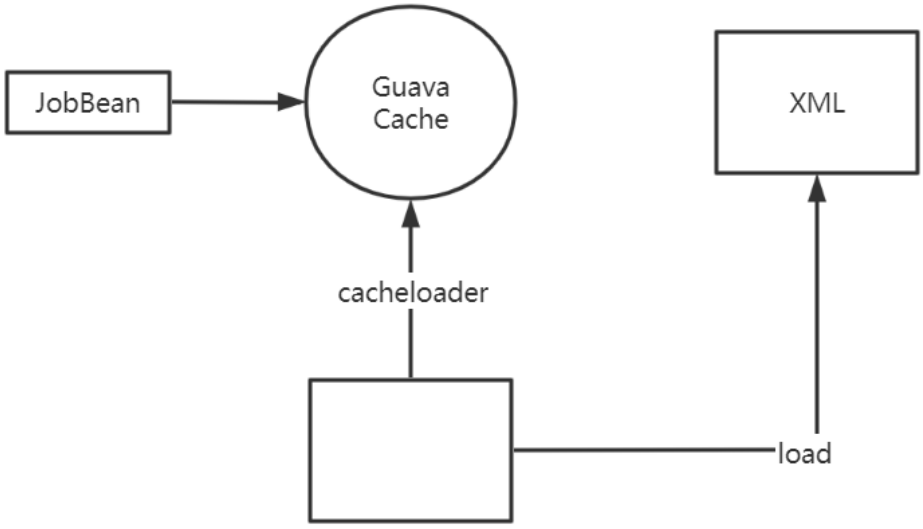
刷新时只有一个请求回源取数据，其他请求会阻塞（block）在一个固定时间段，如果在该时间段内没有获得新值则返回旧值。

```
LoadingCache<String,Object> cache = CacheBuilder.newBuilder()
                // 最大3个          同时支持CPU核数线程写缓存
.maximumSize(3).concurrencyLevel(Runtime.getRuntime().availableProcessors()).
    //3秒内阻塞会返回旧数据
    refreshAfterWrite(3,TimeUnit.SECONDS).build();
```

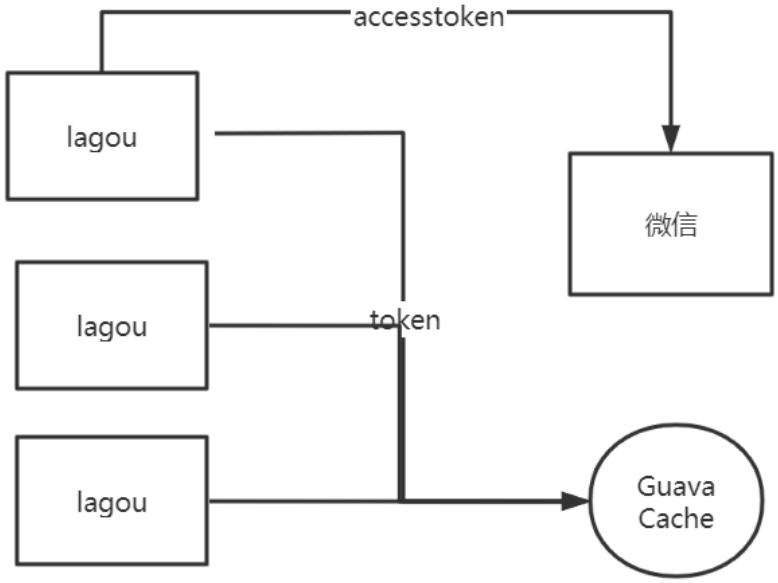
案例分享

1、拉勾首页：职位栏目

```
public class JobBean {
    private int id;
    private String name;
    private int pid;
    private String type;
    private String url;
    private int isHot;
}
```



2、更新锁定：accesstoken token失效 从公网拿token 采用更新锁定



GuavaCache高级实战之动态加载

动态加载行为发生在获取不到数据或者是数据已经过期的时间点，Guava动态加载使用回调模式

用户自定义加载方式，然后Guava cache在需要加载新数据时会回调用户的自定义加载方式

```
segmentFor(hash).get(key, hash, loader)
```

loader即为用户自定义的数据加载方式，当某一线程get不到数据会去回调该自定义加载方式去加载数据

GuavaCache高级实战之自定义LRU算法

```
public class LinkedHashLRUcache<k, v> {
    /**
     * LinkedHashMap（自身实现了ＬＲＵ算法）
     * 有序
```

```
     * 每次访问一个元素，都会加到尾部
     */


      int limit;
      LRUcache<k, v> internalLRUcache;


    public LinkedHashLRUcache(int limit) {

        this.limit = limit;
        this.internalLRUcache = new LRUcache(limit);
    }



    public void put(k key, v value) {
        this.internalLRUcache.put(key, value);
    }

    public v get(k key) {
        return this.internalLRUcache.get(key);
    }



    public static void main(String[] args) {
        LinkedHashLRUcache lru=new LinkedHashLRUcache(3);
        lru.put(1,"zhangfei1");
        lru.put(2,"zhangfei2");
        lru.put(3,"zhangfei3");
        lru.get(1);
        lru.put(4,"zhangfei4");
        for(Object o:lru.internalLRUcache.values()){
            System.out.println(o.toString());
        }
    }
}

public class LRUcache<k, v> extends LinkedHashMap<k, v> {
    private final int limit;

    public LRUcache(int limit) {
        //初始化 accessOrder ：  true 改变尾结点
        super(16, 0.75f, true);
        this.limit = limit;
    }

    //是否删除最老的数据
    @Override
    protected boolean removeEldestEntry(Map.Entry<k, v> eldest) {
        return size() > limit;
    }


}
```

GuavaCache高级实战之疑难问题

GuavaCache会oom（内存溢出）吗

会，当我们设置缓存永不过期（或者很长），缓存的对象不限个数（或者很大）时，比如：

```
Cache<String, String> cache = CacheBuilder.newBuilder()
        .expireAfterWrite(100000, TimeUnit.SECONDS)
        .build();
```

不断向GuavaCache加入大字符串，最终将会oom

解决方案：缓存时间设置相对小些，使用弱引用方式存储对象

```
Cache<String, String> cache = CacheBuilder.newBuilder()
                .expireAfterWrite(1, TimeUnit.SECONDS)
                .weakValues().build();
```

GuavaCache缓存到期就会立即清除吗

不是的，GuavaCache是在每次进行缓存操作的时候，如get()或者put()的时候，判断缓存是否过期

```
void evictEntries(ReferenceEntry<K, V> e) {
  drainRecencyQueue();

  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)) {
      throw new AssertionError();
    }
  }
}
```

一个如果一个对象放入缓存以后，不在有任何缓存操作（包括对缓存其他key的操作），那么该缓存不会主动过期的。

GuavaCache如何找出最久未使用的数据

用accessQueue，这个队列是按照LRU的顺序存放的缓存对象（ReferenceEntry）的。会把访问过的对象放到队列的最后。

并且可以很方便的更新和删除链表中的节点，因为每次访问的时候都可能需要更新该链表，放入到链表的**尾部。**

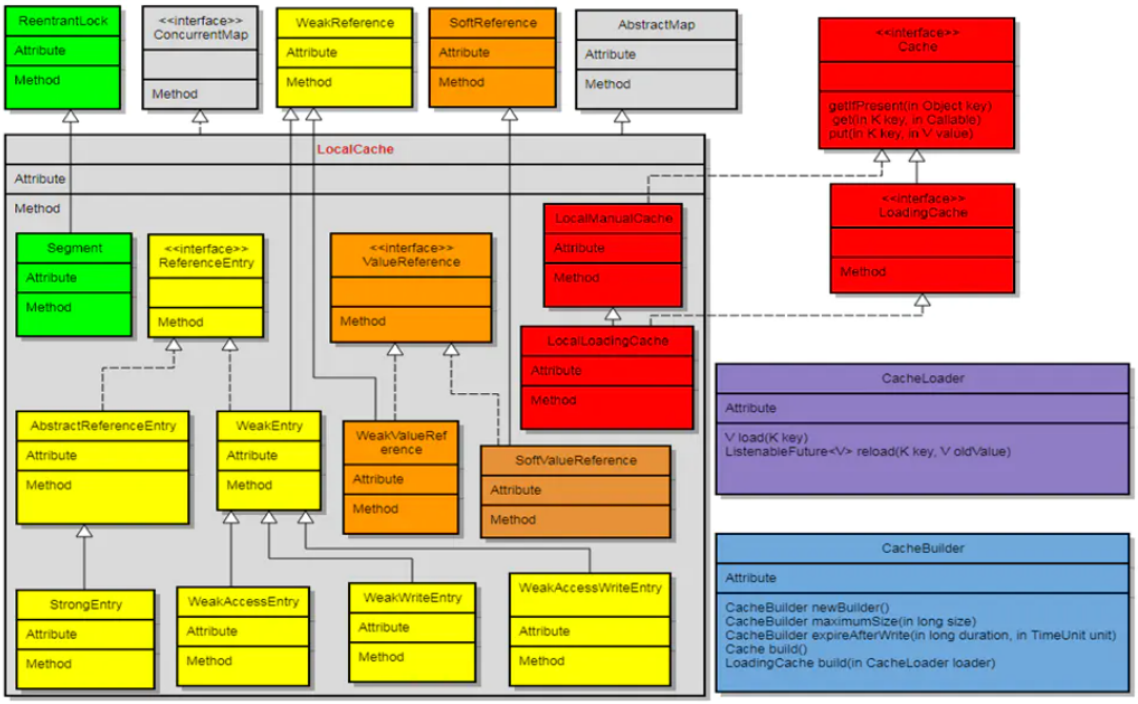
这样，每次从access中拿出的**头节点**就是最久未使用的。

对应的writeQueue用来保存最久未更新的缓存队列，实现方式和accessQueue一样。

Guava Cache源码剖析

GuavaCache源码剖析之实现框架

GuavaCache体系类图：



- CacheBuilder：类，缓存构建器。构建缓存的入口，指定缓存配置参数并初始化本地缓存。

  CacheBuilder在build方法中，会把前面设置的参数，全部传递给LocalCache，它自己实际不参与任何计算

- CacheLoader：抽象类。用于从数据源加载数据，定义load、reload、loadAll等操作
- Cache：接口，定义get、put、invalidate等操作，这里只有缓存增删改的操作，没有数据加载的操作
- LoadingCache：接口，继承自Cache。定义get、getUnchecked、getAll等操作，这些操作都会从数据源load数据
- LocalCache：类。整个guava cache的核心类，包含了guava cache的数据结构以及基本的缓存的操作方法
- LocalManualCache：LocalCache内部静态类，实现Cache接口。其内部的增删改缓存操作全部调用成员变量localCache（LocalCache类型）的相应方法
- LocalLoadingCache：LocalCache内部静态类，继承自LocalManualCache类，实现LoadingCache接口。其所有操作也是调用成员变量localCache（LocalCache类型）的相应方法

LocalCache

LoadingCache这些类表示获取Cache的方式，可以有多种方式，但是它们的方法最终调用到LocalCache的方法，LocalCache是Guava Cache的核心类。

```
class LocalCache<K, V> extends AbstractMap<K, V> implements ConcurrentMap<K, V>
```

LocalCache为Guava Cache的核心类 LocalCache的数据结构与ConcurrentHashMap很相似，都由多个segment组成，且各segment相对独立，互不影响，所以能支持并行操作。

```
//Map的数组
```

```
 final Segment<K, V>[] segments;
//并发量，即segments数组的大小
 final int concurrencyLevel;
 ...
 //访问后的过期时间，设置了expireAfterAccess就有
 final long expireAfterAccessNanos;
 //写入后的过期时间，设置了expireAfterWrite就有
 final long expireAfterWriteNa就有nos;
 //刷新时间，设置了refreshAfterWrite就有
 final long refreshNanos;
 //removal的事件队列，缓存过期后先放到该队列
 final Queue<RemovalNotification<K, V>> removalNotificationQueue;
 //设置的removalListener
 final RemovalListener<K, V> removalListener;
 ...
```

每个segment由一个table和若干队列组成。缓存数据存储在table中，其类型为AtomicReferenceArray。

```
static class Segment<K, V> extends ReentrantLock{

         /**
         * segments 维护一个entry列表的table，确保一致性状态。所以可以不加锁去读。节点的next field是不可修改的final，因为所有list的增加操作
         */
        final LocalCache<K, V> map;

        /**
         * 该segment区域内所有存活的元素个数
         */
        volatile int count;

        /**
         * 改变table大小size的更新次数。这个在批量读取方法期间保证它们可以看到一致性的快照：
         * 如果modCount在我们遍历段加载大小或者核对containsValue期间被改变了，然后我们会看到一个不一致的状态视图，以至于必须去重试。
         * count+modCount 保证内存一致性
         *
         * 感觉这里有点像是版本控制，比如数据库里的version字段来控制数据一致性
         */
        int modCount;

        /**
         * 每个段表，使用乐观锁的Array来保存entry The per-segment table.
         */
        volatile AtomicReferenceArray<ReferenceEntry<K, V>> table; // 这里和concurrentHashMap不一致，原因是这边元素是引用，直接使用不会线程安全
        /**
         * A queue of elements currently in the map, ordered by write time. Elements are added to the tail of the queue
         * on write.
         */
        @GuardedBy("Segment.this")
        final Queue<ReferenceEntry<K, V>> writeQueue;

        /**
```

```
         * A queue of elements currently in the map, ordered by access time.
Elements are added to the tail of the queue
         * on access (note that writes count as accesses).
         */
        @GuardedBy("Segment.this")
        final Queue<ReferenceEntry<K, V>> accessQueue;
}

interface ReferenceEntry<K, V> {
    /**
     * Returns the value reference from this entry.
     */
    ValueReference<K, V> getValueReference();

    /**
     * Sets the value reference for this entry.
     */
    void setValueReference(ValueReference<K, V> valueReference);

    /**
     * Returns the next entry in the chain.
     */
    @Nullable
    ReferenceEntry<K, V> getNext();

    /**
     * Returns the entry's hash.
     */
    int getHash();

    /**
     * Returns the key for this entry.
     */
    @Nullable
    K getKey();

    /*
     * Used by entries that use access order. Access entries are maintained in a
doubly-linked list.
     * New entries are added at the tail of the list at write time; stale
entries are expired from
     * the head of the list.
     */

    /**
     * Returns the time that this entry was last accessed, in ns.
     */
    long getAccessTime();

    /**
     * Sets the entry access time in ns.
     */
    void setAccessTime(long time);
}
```

GuavaCache源码剖析之CacheBuilder

缓存构建器。构建缓存的入口，指定缓存配置参数并初始化本地缓存。
主要采用builder的模式，CacheBuilder的每一个方法都返回这个CacheBuilder知道build方法的调用。
注意build方法有重载，带有参数的为构建一个具有数据加载功能的缓存，不带参数的构建一个没有数据加载功能的缓存。

```
LocalLoadingCache(
        CacheBuilder<? super K, ? super V> builder, CacheLoader<? super K, V>
loader) {
      super(new LocalCache<K, V>(builder,
checkNotNull(loader)));//LocalLoadingCache构造函数需要一个LocalCache作为参数
    }
  //构造LocalCache
  LocalCache(
      CacheBuilder<? super K, ? super V> builder, @Nullable CacheLoader<? super
K, V> loader) {
    concurrencyLevel = Math.min(builder.getConcurrencyLevel(), MAX_SEGMENTS);//
默认并发水平是4

keyStrength = builder.getKeyStrength();//key的强引用
valueStrength = builder.getValueStrength();

keyEquivalence = builder.getKeyEquivalence();//key比较器
valueEquivalence = builder.getValueEquivalence();

maxWeight = builder.getMaximumWeight();
weigher = builder.getWeigher();
expireAfterAccessNanos = builder.getExpireAfterAccessNanos();//读写后有效期，超时重
载
expireAfterWriteNanos = builder.getExpireAfterWriteNanos();//写后有效期，超时重载
refreshNanos = builder.getRefreshNanos();

removalListener = builder.getRemovalListener();//缓存触发失效 或者 GC回收软/弱引用，
触发监听器
removalNotificationQueue =//移除通知队列
    (removalListener == NullListener.INSTANCE)
        ? LocalCache.<RemovalNotification<K, V>>discardingQueue()
        : new ConcurrentLinkedQueue<RemovalNotification<K, V>>();

ticker = builder.getTicker(recordsTime());
entryFactory = EntryFactory.getFactory(keyStrength, usesAccessEntries(),
usesWriteEntries());
globalStatsCounter = builder.getStatsCounterSupplier().get();
defaultLoader = loader;//缓存加载器

int initialCapacity = Math.min(builder.getInitialCapacity(), MAXIMUM_CAPACITY);
if (evictsBySize() && !customWeigher()) {
  initialCapacity = Math.min(initialCapacity, (int) maxWeight);
}
```

GuavaCache源码剖析之Put流程

1、上锁

2、清除队列元素

清理的是keyReferenceQueue和valueReferenceQueue这两个队列，这两个队列是引用队列

如果发现key或者value被GC了，那么会在put的时候触发清理

3、setvalue方法了，它做的是将value写入Entry

```
V put(K key, int hash, V value, boolean onlyIfAbsent) {
      //保证线程安全，加锁
      lock();
      try {
        //获取当前的时间
        long now = map.ticker.read();
        //清除队列中的元素
        preWriteCleanup(now);
        ...
        //获取当前Entry中的HashTable的Entry数组
        AtomicReferenceArray<ReferenceEntry<K, V>> table = this.table;
        //定位
        int index = hash & (table.length() - 1);
        //获取第一个元素
        ReferenceEntry<K, V> first = table.get(index);
        //遍历整个Entry链表
        // Look for an existing entry.
        for (ReferenceEntry<K, V> e = first; e != null; e = e.getNext()) {
          K entryKey = e.getKey();
          if (e.getHash() == hash
              && entryKey != null
              && map.keyEquivalence.equivalent(key, entryKey)) {
            // We found an existing entry.
            //如果找到相应的元素
            ValueReference<K, V> valueReference = e.getValueReference();
            //获取value
            V entryValue = valueReference.get();
            //如果entry的value为null，可能被GC掉了
            if (entryValue == null) {
              ++modCount;
              if (valueReference.isActive()) {
                enqueueNotification( //减小锁时间的开销
                    key, hash, entryValue, valueReference.getWeight(),
RemovalCause.COLLECTED);
                //利用原来的key并且刷新value
                  //存储数据，并且将新增加的元素写入两个队列中，一个Write队列、一个Access
队列
                setValue(e, key, value, now);
                newCount = this.count; // count remains unchanged
              } else {
                setValue(e, key, value, now);//存储数据，并且将新增加的元素写入两个队列
中
                newCount = this.count + 1;
              }
              this.count = newCount; // write-volatile，保证内存可见性
              //淘汰缓存
              evictEntries(e);
              return null;
```

```
            } else if (onlyIfAbsent) {//原来的Entry中包含指定key的元素，所以读取一次，
读取操作需要更新Access队列
    .......
                setValue(e, key, value, now);//存储数据，并且将新增加的元素写入两个队列中
                //数据的淘汰
                evictEntries(e);
                return entryValue;
              }
            }
          }
          //如果目标的entry不存在，那么新建entry
          // Create a new entry.
          ++modCount;
          ReferenceEntry<K, V> newEntry = newEntry(key, hash, first);
          setValue(newEntry, key, value, now);
 ......
        } finally {
          //解锁
          unlock();
          //处理刚刚的remove Cause
          postWriteCleanup();
        }
      }
```

GuavaCache源码剖析之Get流程

1. 获取对象引用（引用可能是非alive的，比如是需要失效的、比如是loading的）；
2. 判断对象引用是否是alive的（如果entry是非法的、部分回收的、loading状态、需要失效的，则认为不是alive）。
3. 如果对象是alive的，如果设置refresh，则异步刷新查询value，然后等待返回最新value。
4. 针对不是alive的，但却是在loading的，等待loading完成（阻塞等待）。
5. 这里如果value还没有拿到，则查询loader方法获取对应的值（阻塞获取）。

```
// LoadingCache methods
    //local cache的代理


    V get(K key, CacheLoader<? super K, V> loader) throws ExecutionException {
    int hash = hash(checkNotNull(key));//hash-->rehash
    return segmentFor(hash).get(key, hash, loader);
  }

  // loading
    //进行指定key对应的value的获取，读取不加锁
    V get(K key, int hash, CacheLoader<? super K, V> loader) throws
ExecutionException {
   ....
      try {
        if (count != 0) { // read-volatile  volatile读会刷新缓存，尽量保证可见性,如果
为0那么直接load
          // don't call getLiveEntry, which would ignore loading values
          ReferenceEntry<K, V> e = getEntry(key, hash);
          //如果对应的Entry不为Null,证明值还在
          if (e != null) {
```

```
            long now = map.ticker.read();//获取当前的时间，根据当前的时间进行Live的数据的读取
            V value = getLiveValue(e, now);  //  判断是否为alive（此处是懒失效，在每次get时才检查是否达到失效时机）
    ......
          }
        }
        //如果取不到值，那么进行统一的加锁get
        // at this point e is either null or expired;     此处或者为null，或者已经被失效。
        return lockedGetOrLoad(key, hash, loader);
      } catch (ExecutionException ee) {
        Throwable cause = ee.getCause();
        if (cause instanceof Error) {
          throw new ExecutionError((Error) cause);
        } else if (cause instanceof RuntimeException) {
          throw new UncheckedExecutionException(cause);
        }
        throw ee;
      } finally {
        postReadCleanup();//每次Put和get之后都要进行一次Clean
      }
    }
```

GuavaCache源码剖析之过期重载

数据过期不会自动重载，而是通过get操作时执行过期重载。具体就是CacheBuilder构造的LocalLoadingCache

```
static class LocalLoadingCache<K, V> extends LocalManualCache<K, V>
      implements LoadingCache<K, V> {
LocalLoadingCache(
    CacheBuilder<? super K, ? super V> builder, CacheLoader<? super K, V>
loader) {
  super(new LocalCache<K, V>(builder, checkNotNull(loader)));
}

// LoadingCache methods

@Override
public V get(K key) throws ExecutionException {
  return localCache.getOrLoad(key);
}

@Override
public V getUnchecked(K key) {
  try {
    return get(key);
  } catch (ExecutionException e) {
    throw new UncheckedExecutionException(e.getCause());
  }
}
```

```
@Override
public ImmutableMap<K, V> getAll(Iterable<? extends K> keys) throws
ExecutionException {
  return localCache.getAll(keys);
}

@Override
public void refresh(K key) {
  localCache.refresh(key);
}

@Override
public final V apply(K key) {
  return getUnchecked(key);
}

// Serialization Support

private static final long serialVersionUID = 1;

@Override
Object writeReplace() {
  return new LoadingSerializationProxy<K, V>(localCache);
}
}
```