Notzuonotdied 调整LruCache文章的编辑细节 e8a0b9e 4 days ago

2 contributors

232 lines (186 sloc) | 7.99 KB

一、Android中的缓存策略

一般来说，缓存策略主要包含缓存的添加、获取和删除这三类操作。如何添加和获取缓存这个比较好理解，那么为什么还要删除缓存呢？这是因为不管是内存缓存还是硬盘缓存，它们的缓存大小都是有限的。当缓存满了之后，再想其添加缓存，这个时候就需要删除一些旧的缓存并添加新的缓存。

因此LRU(Least Recently Used)缓存算法便应运而生，LRU是最近最少使用的算法，它的核心思想是当缓存满时，会优先淘汰那些最近最少使用的缓存对象。采用LRU算法的缓存有两种：LrhCache和DisLruCache，分别用于实现内存缓存和硬盘缓存，其核心思想都是LRU缓存算法。

二、LruCache的使用

LruCache是Android 3.1所提供的一个缓存类，所以在Android中可以直接使用LruCache实现内存缓存。而DisLruCache目前在Android 还不是Android SDK的一部分，但Android官方文档推荐使用该算法来实现硬盘缓存。

1.LruCache的介绍

LruCache是个泛型类，主要算法原理是把最近使用的对象用强引用（即我们平常使用的对象引用方式）存储在 LinkedHashMap中。当缓存满时，把最近最少使用的对象从内存中移除，并提供了get和put方法来完成缓存的获取和添加操作。

2.LruCache的使用

LruCache的使用非常简单，我们就已图片缓存为例。

```
int maxMemory = (int) (Runtime.getRuntime().totalMemory() / 1024);
int cacheSize = maxMemory / 8;
mMemoryCache = new LruCache<String, Bitmap>(cacheSize) {
    @Override
    protected int sizeOf(String key, Bitmap value) {
        return value.getRowBytes() * value.getHeight() / 1024;
    }
};
```

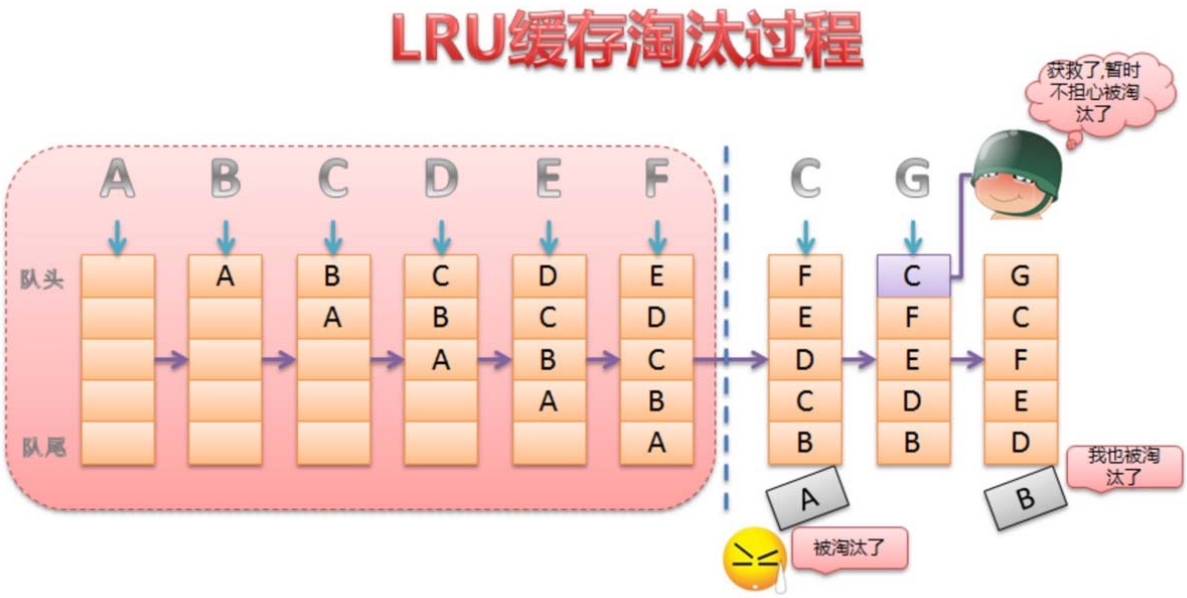
①设置LruCache缓存的大小，一般为当前进程可用容量的1/8。
②重写sizeOf方法，计算出要缓存的每张图片的大小。

注意： 缓存的总容量和每个缓存对象的大小所用单位要一致。

三、LruCache的实现原理

LruCache的核心思想很好理解，就是要维护一个缓存对象列表，其中对象列表的排列方式是按照访问顺序实现的，即一直没访问的对象，将放在队尾，即将被淘汰。而最近访问的对象将放在队头，最后被淘汰。

如下图所示：



那么这个队列到底是由谁来维护的，前面已经介绍了是由LinkedHashMap来维护。

而LinkedHashMap是由数组+双向链表的数据结构来实现的。其中双向链表的结构可以实现访问顺序和插入顺序，使得LinkedHashMap中的<key, value>对按照一定顺序排列起来。

通过下面构造函数来指定LinkedHashMap中双向链表的结构是访问顺序还是插入顺序。

```
public LinkedHashMap(int initialCapacity,
        float loadFactor,
        boolean accessOrder) {
        super(initialCapacity, loadFactor);
        this.accessOrder = accessOrder;
}
```

其中accessOrder设置为true则为访问顺序，为false，则为插入顺序。

以具体例子解释： 当设置为true时

```
public static final void main(String[] args) {
        LinkedHashMap<Integer, Integer> map = new LinkedHashMap<>(0, 0.75f, true);
        map.put(0, 0);
        map.put(1, 1);
        map.put(2, 2);
        map.put(3, 3);
        map.put(4, 4);
        map.put(5, 5);
        map.put(6, 6);
        map.get(1);
        map.get(2);

        for (Map.Entry<Integer, Integer> entry : map.entrySet()) {
                System.out.println(entry.getKey() + ":" + entry.getValue());

        }
}
```

输出结果：

```
0:0
3:3
4:4
5:5
6:6
1:1
2:2
```

即最近访问的最后输出，那么这就正好满足的LRU缓存算法的思想。**可见LruCache巧妙实现，就是利用了LinkedHashMap的这种数据结构。**

下面我们在LruCache源码中具体看看，怎么应用LinkedHashMap来实现缓存的添加，获得和删除的。

```
public LruCache(int maxSize) {
        if (maxSize <= 0) {
                throw new IllegalArgumentException("maxSize <= 0");
        }
        this.maxSize = maxSize;
        this.map = new LinkedHashMap<K, V>(0, 0.75f, true);
}
```

从LruCache的构造函数中可以看到正是用了LinkedHashMap的访问顺序。

put()方法

```
public final V put(K key, V value) {
         //不可为空，否则抛出异常
        if (key == null || value == null) {
                throw new NullPointerException("key == null || value == null");
        }
        V previous;
        synchronized (this) {
            //插入的缓存对象值加1
                putCount++;
            //增加已有缓存的大小
                size += safeSizeOf(key, value);
           //向map中加入缓存对象
                previous = map.put(key, value);
            //如果已有缓存对象，则缓存大小恢复到之前
                if (previous != null) {
                        size -= safeSizeOf(key, previous);
                }
        }
        //entryRemoved()是个空方法，可以自行实现
        if (previous != null) {
                entryRemoved(false, key, previous, value);
        }
        //调整缓存大小(关键方法)
        trimToSize(maxSize);
        return previous;
}
```

可以看到put()方法并没有什么难点，重要的就是在添加过缓存对象后，调用 trimToSize()方法，来判断缓存是否已满，如果满了就要删除近期最少使用的算法。

trimToSize()方法

```
public void trimToSize(int maxSize) {
    //死循环
        while (true) {
                K key;
                V value;
                synchronized (this) {
            //如果map为空并且缓存size不等于0或者缓存size小于0，抛出异常
                        if (size < 0 || (map.isEmpty() && size != 0)) {
                                throw new IllegalStateException(getClass().getName()
                                        + ".sizeOf() is reporting inconsistent results!");
                        }
            //如果缓存大小size小于最大缓存，或者map为空，不需要再删除缓存对象，跳出循环
                        if (size <= maxSize || map.isEmpty()) {
                                break;
                        }
            //迭代器获取第一个对象，即队尾的元素，近期最少访问的元素
                        Map.Entry<K, V> toEvict = map.entrySet().iterator().next();
                        key = toEvict.getKey();
                        value = toEvict.getValue();
            //删除该对象，并更新缓存大小
                        map.remove(key);
                        size -= safeSizeOf(key, value);
                        evictionCount++;
                }
```

```
                entryRemoved(true, key, value, null);
        }
}
```

trimToSize()方法不断地删除LinkedHashMap中队尾的元素，即近期最少访问的，直到缓存大小小于最大值。

当调用LruCache的get()方法获取集合中的缓存对象时，就代表访问了一次该元素，将会更新队列，保持整个队列是按照访问顺序排序。这个更新过程就是在LinkedHashMap中的get()方法中完成的。

先看LruCache的get()方法

get()方法

```
public final V get(K key) {
        //key为空抛出异常
        if (key == null) {
                throw new NullPointerException("key == null");
        }

        V mapValue;
        synchronized (this) {
            //获取对应的缓存对象
            //get()方法会实现将访问的元素更新到队列头部的功能
                mapValue = map.get(key);
                if (mapValue != null) {
                        hitCount++;
                        return mapValue;
                }
                missCount++;
        }
```

其中LinkedHashMap的get()方法如下：

```
public V get(Object key) {
        LinkedHashMapEntry<K,V> e = (LinkedHashMapEntry<K,V>)getEntry(key);
        if (e == null)
                return null;
    //实现排序的关键方法
        e.recordAccess(this);
        return e.value;
}
```

调用recordAccess()方法如下：

```
void recordAccess(HashMap<K,V> m) {
        LinkedHashMap<K,V> lm = (LinkedHashMap<K,V>)m;
    //判断是否是访问排序
        if (lm.accessOrder) {
                lm.modCount++;
        //删除此元素
                remove();
        //将此元素移动到队列的头部
                addBefore(lm.header);
        }
}
```

由此可见LruCache中维护了一个集合LinkedHashMap，该LinkedHashMap是以访问顺序排序的。当调用put()方法时，就会在结合中添加元素，并调用trimToSize()判断缓存是否已满，如果满了就用LinkedHashMap的迭代器删除队尾元素，即近期最少访问的元素。当调用get()方法访问缓存对象时，就会调用LinkedHashMap的get()方法获得对应集合元素，同时会更新该元素到队头。