LRU Cache Implementation by LinkedHashMap - Leetcode

21 November 2013

Currently **LRU Cache** is among the top three least acceptance problems on leetcode, after **Word Ladder II** and **Valid Number**. It is described as follows:

> "Design and implement a data structure for Least Recently Used (LRU) cache. It should support the following operations: get and set.
> get(key) - Get the value (will always be positive) of the key if the key exists in the cache, otherwise return -1.
> set(key, value) - Set or insert the value if the key is not already present. When the cache reached its capacity, it should invalidate the least recently used item before inserting a new item."

In designing the data structure, a few ideas come to mind:

1. The get(key) and set(key) should support O(1) simplicity, otherwise it fails with a Time Limit Exceeded error. Therefore it is natural to utilize a hashtable, or a HashMap in Java.
2. To envict the least recently used entry, it is handy to utilize a doubly linked list for convenient removal of the last node. Also, to put a new <key, value> pair, doubly linked list makes it easier to add a first node.

Here first node represents the most recently visited entry, and the last node represents the least recently visited entry.

Considering both, [LinkedHashMap] (http://docs.oracle.com/javase/7/docs/api/java/util/LinkedHashMap.html) will be an ideal implementation of the LRU cache. *get()* and *set()* become straightforward. To *get* a key, we first retrieves the value if it exists, then remove the entry to add it before the first node. Similarly, to *set* a <key, value>, we first *get()* the key. Doing this has two effects: (1) if the key does not exist, nothing happens; (2) if the key already exists, it is swapped to the first node. Whichever is the case, we use a put(key, value) afterwards to add a new <key, value> or to replace the value of the existing key. Note the <K,V> is indeed <Integer, Integer>. Autoboxing takes care of the wrapping/unwrapping between int and Integer:

```
public int get(int key) {
    if(map == null || map.get(key) == null) return -1;
    int value = map.get(key);
    map.remove(key);
    map.put(key, value);
    return value;
}

public void set(int key, int value) {
    if(map == null) return;
    get(key);
    map.put(key, value);
}
```

LinkedHashMap is exactly invented to do this piece of work, isn't it? It's almost too good to be true, except it is. What is left behind, oh the *capacity*. Cache has limited size, we must remove the least recently used entry in *set()* if the *size()* is alarming. The bad news is LinkedHashMap only implements the Map interface but not the List interface. It is not a LinkedList, either. Therefore there is no removeLast() or any method like that. We can

remove the last element by brutal force but it will kill the O(1) complexity we tried so hard to accomplish.

I just saved the good news till the last: scrolling down the [documentation](http://docs.oracle.com/javase/7/docs/api/java/util/LinkedHashMap.html#removeEldestEntry(java.util.Map.Entry) there appears a *removeEldestEntry* method.

> protected boolean removeEldestEntry(Map.Entry<K,V> eldest)
> Returns true if this map should remove its eldest entry. This method is invoked by put and putAll after inserting a new entry into the map. It provides the implementor with the opportunity to remove the eldest entry each time a new one is added. This is useful if the map represents a cache: it allows the map to reduce memory consumption by deleting stale entries.

It's sometimes (always) rewarding to read the documentation. What it says is basicly the method is invoked by put() or putAll() internally as needed. The eldest (the last) entry is automatically removed when put() or putall() refers to removeEldestEntery and if it returns true. See, the method is *protected*. You are not supposed to do the removal manually. Via this trick we can implement a fixed size flavor of LinkedHashMap by extending it.

```
import java.util.LinkedHashMap;

class LRUMap<K,V> extends LinkedHashMap<K,V>{
    private final int MAX_NUM ;

    public LRUMap( int capacity){
        super(capacity);
        MAX_NUM = capacity;
    }

    @Override
```

```
    protected boolean removeEldestEntry(Map.Entry eldest){
        #last entry will automatically be removed when size() exceeds capacity
        return size() > MAX_NUM ;
    }
}
```

With this internal *removeEldestEntry* mechanism, our *set()* works without modification. The complete [code](https://github.com/fengs/leetcode/blob/master/LRUCache/LRUCache.java) clarifies itself:

```
import java.util.LinkedHashMap;

class LRUMap<K,V> extends LinkedHashMap<K,V>{
    private final int MAX_NUM;

    public LRUMap(int capacity){
        super(capacity);
        MAX_NUM = capacity;
    }

    @Override
    protected boolean removeEldestEntry(Map.Entry eldest){
        return size() > MAX_NUM;
    }
}

public class LRUCache {
    LRUMap<Integer, Integer> map;
```

```
    public LRUCache(int capacity) {
        map = new LRUMap<Integer, Integer>(capacity);
    }

    public int get(int key) {
        if(map == null || map.get(key) == null) return -1;
        int value = map.get(key);
        map.remove(key);
        map.put(key, value);
        return value;
    }

    public void set(int key, int value) {
        if(map == null) return;
        get(key);
        map.put(key, value);
    }
}
```

Online judge says yes. Thanks to the wheels that have already been invented, the solution is only 37 lines in total, not too long to read in one breath. Now you can take a good, deep breath. I'm still holding mine.

2

leetcode [1] java [1] LRU cache [1] algorithm [1]

← Previous | Archive | Next →

blog comments powered by Disqus