链滴

Latke CodeView - 1 cache

作者：ZephyrJung
原文链接：https://ld246.com/article/1478000436656
来源网站：链滴
许可协议：署名-相同方式共享 4.0 国际 (CC BY-SA 4.0)

未注明的引用均为Java内部包

所有setter、getter、logger等逻辑无关代码均忽略不计

忽略了一些常见注解

org.b3log.latke.cache.local.memory.AbstractMemoryCache

```
import org.b3log.latke.cache.Cache;
public abstract class AbstractMemoryCache<K extends Serializable, V extends Serializable>
    implements Cache<K, V> {
    private long maxCount = Long.MAX_VALUE;
    private long hitCount;
    private long missCount;
    private long putCount;
    private long cachedCount;
    protected final void hitCountInc() {
        hitCount++;
    }
    protected final void missCountInc() {
        missCount++;
    }
    protected final void putCountInc() {
        putCount++;
    }
    protected final void cachedCountInc() {
        cachedCount++;
    }
    protected final void cachedCountDec() {
        cachedCount--;
    }
    public long getCachedBytes() {
        // TODO: getCachedBytes
        return -1;
    }
    public long getHitBytes() {
        // TODO: getHitBytes
        return -1;
    }
}
```

- **抽象类实现接口的作用是什么?**
- **为什么采用long类型，int竟然不够用?**
- **类声明处K、V的extends是什么意思**

org.b3log.latke.cache.local.memory.LruMemoryCache

```
import org.b3log.latke.cache.local.util.DoubleLinkedMap;
import org.b3log.latke.logging.Level;
import org.b3log.latke.logging.Logger;
import org.b3log.latke.util.Serializer;
public final class LruMemoryCache<K extends Serializable, V extends Serializable> extends A
```

```
stractMemoryCache<K, V> implements Serializable {
    private DoubleLinkedMap<K, byte[]> map;
    public LruMemoryCache() {
        map = new DoubleLinkedMap<K, byte[]>();
    }
    public void put(final K key, final V value) {
        //删除掉原有的，并添加到首部，即每次都将使用的放到最前面，是为LRU算法
        remove(key);
        putCountInc();
        synchronized (this) {
            if (getCachedCount() >= getMaxCount()) {
                collect();
            }
            try {
                map.addFirst(key, Serializer.serialize((Serializable) value));
            } catch (final IOException e) {
                return;
            }
            cachedCountInc();
        }
    }
    public void putAsync(final K key, final V value) {
        put(key, value);
    }
    public synchronized V get(final K key) {
        final byte[] bytes = map.get(key);
        if (bytes != null) {
            hitCountInc();
            map.makeFirst(key);
            try {
                return (V) Serializer.deserialize(bytes);
            } catch (final Exception e) {
                return null;
            }
        }
        missCountInc();
        return null;
    }
    public synchronized void remove(final K key) {
        final boolean removed = map.remove(key);
        if (removed) {
            cachedCountDec();
        }
    }
    public synchronized void remove(final Collection<K> keys) {
        for (final K key : keys) {
            remove(key);
        }
    }
    public synchronized void collect() {
        map.removeLast();
        cachedCountDec();
    }
    public synchronized void removeAll() {
```

```
        map.removeAll();
        setCachedCount(0);
        setMissCount(0);
        setHitCount(0);
    }
    public boolean contains(final K key) {
        return null != get(key); // XXX: performance issue
    }
    public long inc(final K key, final long delta) {
        V ret = get(key);
        if (null == ret || !(ret instanceof Long)) {
            final Long v = delta;
            ret = (V) v;
            put(key, ret);
        }
        if (ret instanceof Long) {
            final Long v = (Long) ret + delta;
            ret = (V) v;
            put(key, ret);
        }
        return (Long) ret;
    }
}
```

- putCount和cachedCount是什么关系？我以为put就是当前存放的容量，但是看代码使用cachedCount和maxCount相比的？

读Cache接口发现注释：putCount是指存放次数，cachedCount是指当前存放的数量。

- **put和putAsync看起来没什么区别，为何多加一个putAsync？**
- **inc方法是什么意思？作用是什么？**

org.b3log.latke.cache.local.util.DoubleLinkedMap

```
public final class DoubleLinkedMap<K, V> implements Serializable {
    private static final long serialVersionUID = 1L;
    private int size = 0;
    private DoubleLinkedMapNode<K, V> first;
    private DoubleLinkedMapNode<K, V> last;
    public synchronized boolean remove(final K key) {
        final DoubleLinkedMapNode<K, V> node = getNode(key);
        if (node != null) {
            removeNode(node);
            return true;
        }
        return false;
    }
    public synchronized V get(final K key) {
        if (first == null) {
            return null;
        } else {
            DoubleLinkedMapNode<K, V> current = first;
            while (current != null) {
                if (current.getKey().equals(key)) {
```

```
                return current.getValue();
            } else {
                current = current.getNext();
            }
        }
    }
    return null;
}
public synchronized void addLast(final K key, final V value) {
    final DoubleLinkedMapNode<K, V> node = new DoubleLinkedMapNode<K, V>(key, val
e);
    addLastNode(node);
}
public synchronized void addFirst(final K key, final V value) {
    if (null == key) {
        throw new IllegalArgumentException("Key is null!");
    }
    final DoubleLinkedMapNode<K, V> node = new DoubleLinkedMapNode<K, V>(key, val
e);
    addFirstNode(node);
}
public synchronized void makeFirst(final K key) {
    final DoubleLinkedMapNode<K, V> node = getNode(key);
    if (node.getPrev() == null) {
        return;
    }
    //将当前节点的前一节点的下一节点改为当前节点的下一节点
    node.getPrev().setNext(node.getNext());
    if (node.getNext() == null) {
        //如果当前节点是最后一个节点，则将前一节点的下一节点设置为null
        last = node.getPrev();
        last.setNext(null);
    } else {
        node.getNext().setPrev(node.getPrev());
    }
    first.setPrev(node);
    node.setNext(first);
    node.setPrev(null);
    first = node;
}
public synchronized void removeAll() {
    //此处似乎直观的体现了Java不用做垃圾处理，如果是C/C++，应有回收空间代码
    for (DoubleLinkedMapNode<K, V> me = first; me != null;) {
        if (me.getPrev() != null) {
            me.setPrev(null);
        }
        final DoubleLinkedMapNode<K, V> next = me.getNext();
        me = next;
    }
    first = null;
    last = null;
    size = 0;
}
public synchronized V removeLast() {
```

```
    final DoubleLinkedMapNode<K, V> lastNode = removeLastNode();
    if (null != lastNode) {
        return lastNode.getValue();
    }
    return null;
}
public synchronized int size() {
    return size;
}
private synchronized DoubleLinkedMapNode<K, V> removeLastNode() {
    final DoubleLinkedMapNode<K, V> ret = last;
    if (last != null) {
        removeNode(last);
    }
    return ret;
}
private synchronized DoubleLinkedMapNode<K, V> getNode(final K key) {
    //遍历
    if (first == null) {
        return null;
    } else {
        DoubleLinkedMapNode<K, V> current = first;
        while (current != null) {
            if (current.getKey().equals(key)) {
                return current;
            } else {
                current = current.getNext();
            }
        }
    }
    return null;
}
private synchronized void removeNode(final DoubleLinkedMapNode<K, V> node) {
    if (node.getNext() == null) {
        if (node.getPrev() == null) {
            //删除双链表的唯一节点
            if (node == first && node == last) {
                first = null;
                last = null;
            }
        } else {
            //删除尾节点
            last = node.getPrev();
            last.setNext(null);
            node.setPrev(null);
        }
    } else if (node.getPrev() == null) {
        //删除头节点
        first = node.getNext();
        first.setPrev(null);
        node.setNext(null);
    } else {
        //删除中间的节点
        node.getPrev().setNext(node.getNext());
```

```
            node.getNext().setPrev(node.getPrev());
            node.setPrev(null);
            node.setNext(null);
        }
        size--;
    }
    private synchronized void addLastNode(final DoubleLinkedMapNode<K, V> node) {
        if (first == null) {
            first = node;
        } else {
            last.setNext(node);
            node.setPrev(last);
        }
        last = node;
        size++;
    }
    private synchronized void addFirstNode(final DoubleLinkedMapNode<K, V> node) {
        if (last == null) {
            last = node;
        } else {
            first.setPrev(node);
            node.setNext(first);
        }
        first = node;
        size++;
    }
}
final class DoubleLinkedMapNode<K, V> implements Serializable {
    private static final long serialVersionUID = -5617593667027497669L;
    private V value;
    private K key;
    private DoubleLinkedMapNode<K, V> prev;
    private DoubleLinkedMapNode<K, V> next;
    DoubleLinkedMapNode(final K key, final V value) {
        this.key = key;
        this.value = value;
    }
    public K getKey() {
        return key;
    }
    public V getValue() {
        return value;
    }
    protected DoubleLinkedMapNode<K, V> getNext() {
        return next;
    }
    protected void setNext(final DoubleLinkedMapNode<K, V> next) {
        //传递对象其实就是传递引用，如此说来与指针颇为类似
        this.next = next;
    }
    protected DoubleLinkedMapNode<K, V> getPrev() {
        return prev;
    }
    protected void setPrev(final DoubleLinkedMapNode<K, V> prev) {
```

```
        this.prev = prev;
    }
}
```

- **为什么addFirst里判断了key是否为null，而addLast没有?**
-

node.getPrev().setNext(node.getNext());

if (node.getNext() == null) {

last = node.getPrev();

last.setNext(null);

} else {

node.getNext().setPrev(node.getPrev());

}

这个逻辑看起来有点重复，是不是可以改为：

```java
node.getPrev().setNext(node.getNext());
if (node.getNext() != null) {
    node.getNext().setPrev(node.getPrev());
}
```

- **removeLast和removeLastNode看起来也没什么区别，用意何在?**

org.b3log.latke.cache.Cache

```java
//接口定义，在AbstractMemoryCache中均已涉及
public interface Cache<K extends Serializable, V extends Serializable> {
    boolean contains(final K key);
    void put(final K key, final V value);
    void putAsync(final K key, final V value);
    V get(final K key);
    long inc(final K key, final long delta);
    void remove(final K key);
    void remove(final Collection<K> keys);
    void removeAll();
    void setMaxCount(final long maxCount);
    long getMaxCount();
    long getHitCount();
    long getMissCount();
    long getPutCount();
    long getCachedCount();
    long getCachedBytes();
    long getHitBytes();
    void collect();
}
```

org.b3log.latke.cache.CacheFactory

```
import org.b3log.latke.Latkes;
import org.b3log.latke.RuntimeEnv;
public final class CacheFactory {
    private static final Map<String, Cache<String, ?>> CACHES = Collections.synchronizedMap
new HashMap<String, Cache<String, ?>>());
    public static synchronized void removeAll() {
        RuntimeEnv runtime = Latkes.getRuntime("cache");
        if (RuntimeEnv.LOCAL == Latkes.getRuntimeEnv()) {
            runtime = RuntimeEnv.LOCAL;
        }
        switch (runtime) {
            case LOCAL:
                for (final Map.Entry<String, Cache<String, ?>> entry : CACHES.entrySet()) {
                    final Cache<String, ?> cache = entry.getValue();
                    cache.removeAll();
                }
                break;
            default:
                throw new RuntimeException("Latke runs in the hell.... Please set the enviornment c
rrectly");
        }
    }
    public static synchronized Cache<String, ? extends Serializable> getCache(final String cach
Name) {
        Cache<String, ?> ret = CACHES.get(cacheName);
        try {
            if (null == ret) {
                switch (Latkes.getRuntime("cache")) {
                    //通过反射获取到LruMemoryCache并得到一个实例
                    case LOCAL:
                        final Class<Cache<String, ?>> localLruCache = (Class<Cache<String, ?>>) Clas
.forName(
                                "org.b3log.latke.cache.local.memory.LruMemoryCache");
                        ret = localLruCache.newInstance();
                        break;
                    default:
                        throw new RuntimeException("Latke runs in the hell.... Please set the enviornm
nt correctly");
                }
                CACHES.put(cacheName, ret);
            }
        } catch (final Exception e) {
            throw new RuntimeException("Can not get cache: " + e.getMessage(), e);
        }
        return (Cache<String, Serializable>) ret;
    }
    private CacheFactory() {
    }
}
```

- Collections.synchronizedMap 看起来是个挺高端的玩意儿，有待bing
- **不太明白通过反射获取实例和直接import该类实例化有什么区别**
- 这个cacheFactory看起来是在Latke处调用的，有待观察其调用过程

org.b3log.latke.cache.NoCache

```
public final class NoCache<K extends Serializable, V extends Serializable> implements Cache
K, V> {
    private String name;
    public NoCache(final String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }
    public boolean contains(final K key) {
        return false;
    }
}
//methods implements from Cache
```

- 该类继承了Cache接口并且所有方法均为空实现
- **这个类的用意何在?**