Implementing LRU Cache in Java

Published on: 4th April 2016

Posted By: Sain Technology Solutions

This tutorial will introduce you to characteristics of LRU Cache and explain how to implement it in Java

Abstract

Caching is often leveraged to maintain the data in form of key-value pair in main memory. This results into better performance in comparison to other options such as database storage and file storage. However since main memory is always limited, we need to define the eviction policy to ensure that cache does not grow beyond a certain size.

There are different caching algorithms based on the eviction policies that they offer. Some of these caching algorithms are LRU (Least Recently Used), LFU (Least Frequently Used) and MRU (Most Recently Used). In this tutorial, we will be focusing on LRU cache and see how we can implement it in Java.

Characteristics of LRU Cache

Here are the characteristics of LRU cache that distinguishes it from other types of caches -

1. Maintains fixed number of elements.
2. Orders the elements based on their usage pattern. Most recently used element always remains on head while least recently used element is stored at tail end.
3. Removes the element that was least recently used when cache gets full or crosses a threshold ratio (called load factor).
4. Updates the element's value if element to be inserted already exists.

LRU Cache Implementation in Java

We will be using following classes to implement LRU Cache in Java-

Recent Tutorials and Articles

Developing Apache Kafka Consum...
This tutorial provides the steps to implement a basic Apache Kafka consumer in Java

Developing Apache Kafka Produc...
This tutorial provides the steps to implement a basic Apache Kafka producer in Java

Implementing Request Response ...
This tutorial explains how to implement Request / Response paradigm with Apache Kafka usin...

Related Tutorials and Articles

Implementing LRU Cache using L...
This tutorial will introduce you to characteristics of LRU Cache and explain how to implem...

Implementing Trie Data Structu...
This tutorial will introduce you to Trie data structure along with providing its implement...

```
import java.util.HashMap;
import java.util.Map;

/**
 * This code is distributed under Apache License 2.0.
 *
 * @author Sain Technology Solutions
```



```
public class LRUCache<K, V> {

        private final int capacity;

        private Map<K, Node<K, V>> cacheElements = new HashMap<>();
        private Node<K, V> head;
        private Node<K, V> tail;

        public LRUCache(int capacity) {
                this.capacity = capacity;
        }

        /**
         * Removes the element at tail from Doubly Linked List as well as
         * cacheElements map.
         */
        private void removeTail() {
                cacheElements.remove(tail.key);

                tail = tail.previous;
                tail.next = null;
        }

        /**
         * Moves the input node to head of the doubly linked list.
         *
         * @param node
         *            Node to be moved
         */
        private void moveToHead(Node<K, V> node) {
                // If node is already at head, do nothing and simply return
                if (node == head) {
                        return;
                }

                // remove the node from LinkedList
                node.previous.next = node.next;
```

```
/**
 * Puts the input node at head of the doubly linked list.
 *
 * @param node
 *            Node to be put on head
 */
private void putAsHead(Node<K, V> node) {
        node.next = head;
        node.previous = null;

        if (head != null) {
                head.previous = node;
        }

        head = node;

        if (tail == null) {
                tail = head;
        }
}

/**
 * Returns the value corresponding to input key from Cache Map. It also
 * moves this element to head of doubly linked list since it was most
 * recently accessed.
 *
 * @param key
 *            Key for the element whose value needs to be returned
 * @return Value of the element with input key or null if no such element
 *         exists
 */
public V get(K key) {

        if (cacheElements.containsKey(key)) {
                final Node<K, V> n = cacheElements.get(key);

                moveToHead(n);

                return n.value;
        }

        return null;
}

/**
 * Put the element with input key and value in the cache. If the element
```

Recent Tutorials and Articles

Developing Apache Kafka Consum...
This tutorial provides the steps to implement a basic Apache Kafka consumer in Java

Developing Apache Kafka Produc...
This tutorial provides the steps to implement a basic Apache Kafka producer in Java

Implementing Request Response ...
This tutorial explains how to implement Request / Response paradigm with Apache Kafka usin...

```
                final Node<K, V> old = cacheElements.get(key);
                old.value = value;

                moveToHead(old);
        } else {
                final Node<K, V> created = new Node<>(key, value);

                if (cacheElements.size() >= capacity) {
                        removeTail();
                        putAsHead(created);
                } else {
                        putAsHead(created);
                }

                cacheElements.put(key, created);
        }
}

/**
 * Implementation of a Node of a Doubly Linked List.
 *
 * @author Sain Technology Solutions
 *
 * @param <K>
 * @param <V>
 */
private static class Node<K, V> {
        K key;
        V value;
        Node<K, V> next;
        Node<K, V> previous;

        private Node(K key, V value) {
                this.key = key;
                this.value = value;
        }

        @Override
        public String toString() {
                return "Node [key=" + key + ", value=" + value + "]";
        }

}

@Override
public String toString() {
```

Recent Tutorials and Articles

Developing Apache Kafka Consum...
This tutorial provides the steps to implement a basic Apache Kafka consumer in Java

Developing Apache Kafka Produc...
This tutorial provides the steps to implement a basic Apache Kafka producer in Java

Implementing Request Response ...
This tutorial explains how to implement Request / Response paradigm with Apache Kafka usin...

```
                cache.put(2, 1); // Will add an element with key as 2 and value as 1
                cache.put(3, 2); // Will add an element with key as 3 and value as 2

                // Will add an element with key as 4 and value as 3. Also will remove
                // the element with key 2 as it was added least recently and cache can
                // just have two elements at a time
                cache.put(4, 3);

                // Since element with key 2 was removed, it will return null
                System.out.println(cache.get(2));

                // It will return value 2 and move the element with key 3 to the head.
                // After this point, element with key 4 will be least recently accessed
                System.out.println(cache.get(3));

                // Will add an element with key as 5 and value as 4. Also will remove
                // the element with key 4 as it was accessed least recently and cache
                // can just have two elements at a time
                cache.put(5, 4);

                // Since element with key 2 was removed, it will return null
                System.out.println(cache.get(4));
        }
}
```

Usage of LRU Cache

Some of the use cases of LRU cache are as follows –

1. **General Caching -** LRU Cache can be used to cache the objects where we want to avoid the calls to database. Using LRU, we always ensure that only the objects that we have used recently remain in cache while getting rid of objects those were not used recently.
2. **Caching Bitmaps in Android -** Rendering bitmaps images in Android views can be slow if we load bitmap every time we want to use it. On the other hand, we can't maintain all the images in memory if there are too many images to render in different views. Therefore we need to keep removing the bitmaps out of memory to avoid out of memory crashes. LRU Cache works very well in this case as it automatically removes the bitmaps that were used least recently.

Recent Tutorials and Articles

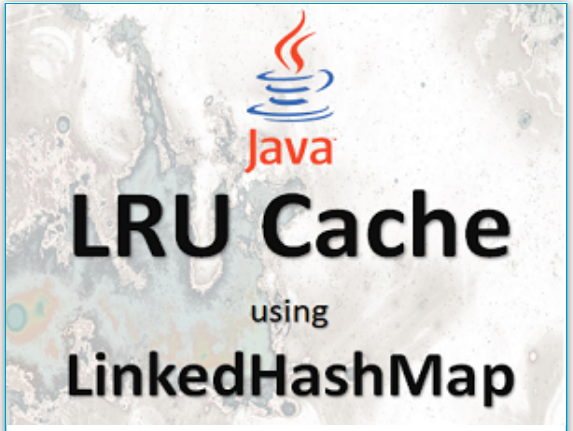
Developing Apache Kafka Consum...
This tutorial provides the steps to implement a basic Apache Kafka consumer in Java

Developing Apache Kafka Produc...
This tutorial provides the steps to implement a basic Apache Kafka producer in Java

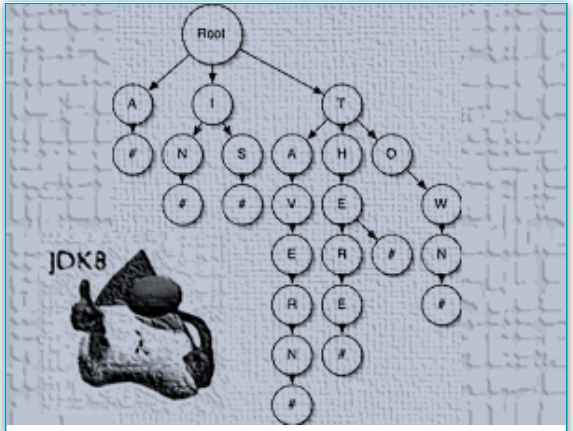
Implementing Request Response ...
This tutorial explains how to implement Request / Response paradigm with Apache Kafka usin...

Posted By: Sain Technology Solutions
Published on: 4th April 2016

You may also like...



Implementing LRU Cache using LinkedHashMap in



Implementing Trie Data Structure in Java

Comments

Name

email

Recent Tutorials and Articles

Developing Apache Kafka Consum...
This tutorial provides the steps to implement a basic Apache Kafka consumer in Java

Developing Apache Kafka Produc...
This tutorial provides the steps to implement a basic Apache Kafka producer in Java

Implementing Request Response ...
This tutorial explains how to implement Request / Response paradigm with Apache Kafka usin...

No one has commented yet. Be the first!

Recent Tutorials and Articles

Developing Apache Kafka Consum...
This tutorial provides the steps to implement a basic Apache Kafka consumer in Java

Developing Apache Kafka Produc...
This tutorial provides the steps to implement a basic Apache Kafka producer in Java

Implementing Request Response ...
This tutorial explains how to implement Request / Response paradigm with Apache Kafka usin...