Concurrent Cache

Version 3

Sameer Punjal

4/20/2018

Overview

This file serves as documentation for the accompanying implementation of a tiered concurrent cache with a configurable cache replacement policy. Information regarding the design of the implementation and how it is to be used can be found here.

This cache implementation is thread-safe by default (see note in "Using Alternative SubCache Implementations" section). This implementation could be a good choice when a user wants compromise between using an effective cache replacement features, and allowing for concurrent reads and writes.

Requirements

1. The cache should exist entirely within memory, without directly communicating with backing store
2. The cache should allow for keys and values of arbitrary types, with type safety enforced, so that all keys and all values must be of the same type
3. The implementation should be distributed as a library, and the end users should be able to use the library without access to the library's source code
4. The implementation should include LRU and MRU replacement algorithms for the subcaches
5. The user should be able to provide an arbitrary replacement algorithms that they can implement themselves for the subcaches

Building and Usage

This project was written in Java version 1.8. It uses JUnit version 4.8.2 as a unit testing framework, and Maven version 3.5.2 to manage the project's build. This project can be built using any Java IDE that integrates with Maven, or through the command line by issuing the command `$ mvn clean install` in the `ConcurrentCache` directory. This command will remove output from previous builds (as per the `clean` command), compile the source code and tests, produce a `.jar` file (located in the target directory), and run the unit tests located in `ConcurrentCache/src/test/java/com/sameer/concurrentcache`

To include the cache in another Java program, include the import statement `import concurrentcache.ConcurrentCache;` or `import concurrentcache.*;` Before compiling the program, build the ConcurrentCache project using Maven, and add `ConcurrentCache-1.0-SNAPSHOT.jar` (located in the `target` directory) to your project's classpath.

Public Methods

```
public ConcurrentCache(int n, int subcapacity)
```

Constructor - initializes empty cache with `n` subcaches, each with a maximum capacity of `subcapacity`. Should be used as `new ConcurrentCache<K,V>(n,subcapacity)` where `K` is the type of the keys, and `V` is the type of the values

```
public int size()
```

Returns the number of key value pairs currently stored in the cache

```
public containsKey(K key)
```

Returns true if key is contained in the cache and false if not

```
public V get(K key)
```

Returns the value associated with key. If key is not associated with a value, the return value will be null

```
public void put(K key, V value)
```

Inserts the key value pair (`key`,`value`) into the cache. Another key value pair may be evicted if necessary

```
public void remove(K key)
```

Removes the key value pair associated with key, if such a pair exists

```
public void clear()
```

Removes all key value pairs from the cache.

Using Alternative SubCache implementations

(new in this version)

By default, this implementation implements the N sets as LRU caches. However, users have the option of incorporating their own caching policies and algorithms in one of two ways:

1) **Using the Evictor interface**

The Evictor interface was designed to allow for users to create their own data structures to manage key eviction for subcaches. In the implementation of the ConcurrentCache, instances of the Evictor class are used in combination with instances of the EvictorCache class.

The EvictorCache is a synchronized Cache that stores key value pairs, and when an EvictorCache is instantiated, it takes in a capacity parameter (`int`), and an instance of Evictor. When any add, access, remove, or clear operation is performed on the cache, a corresponding call is made to the cache's Evictor. When the cache is at capacity and a new key is to be added to the cache, the EvictorCache consults the Evictor's `chooseEvict` method to determine which key value pair should be removed from the cache.

When a ConcurrentCache is initialized, N EvictorCache objects are created, and each EvictorCache is initialized with a new instance of LRUEvictor, which implements an LRU replacement policy. Using Java's Anonymous Inner Classes, a user can substitute in a custom Evictor class by overriding the following function:

```
protected Evictor<K> createEvictor()
```

This method will be called N times to construct N Evictors corresponding to the N EvictorCaches that will serve as subcaches. The following is an example that shows how another Evictor (in this case, the supplied MRUEvictor) can be substituted in:

```
ConcurrentCache<Integer,String> example =

new ConcurrentCache<Integer,String>(4,16){

@Override
protected Evictor<Integer> createEvictor(){
      return new MRUEvictor<Integer>();
}

};
```

The Evictor interface can be found at the following path:
`ConcurrentCache/src/main/java/com/sameer/concurrentcache/Evictor.java`

2) Using the Cache interface

The Evictor interface was provided to allow for a user to simply specify a key replacement algorithm, while leaving the specifics of subcache storage to be handled by an Evictor's corresponding EvictorCache instance. If a user is interested in creating their own subcaches that handle storage and retrieval in a particular way, it can be done by creating a new class that implements the Cache interface, which can be found at
`ConcurrentCache/src/main/java/com/sameer/concurrentcache/Evictor.java`

A user's alternative Cache implementation can be incorporated into a ConcurrentCache instance by overriding the following method:

```
protected Cache<Integer,String> createSubCache(int subcapacity)
```

When an instance of ConcurrentCache is constructed, `createSubCache` is called N times, and the N output caches serve as the subcaches. By default, `createSubCache` returns a new instance of EvictorCache with capacity `subcapacity`, and with an instance of Evictor returned from the `createEvictor` method as mentioned in the previous section. When `createSubCache` is overridden, it should return a Cache object with a fixed capacity of `subcapacity`. The `createSubCache` method can be overridden as shown below, where `MyCache` is an example class that implements Cache.

```
ConcurrentCache<Integer,String> example =

new ConcurrentCache<Integer,String>(4,16){

@Override
protected Cache<Integer,String> createSubCache(int subcapacity){
      return new MyCache<Integer,String>(subcapacity);
}

};
```

Note: EvictorCache is a thread-safe Cache. To guarantee thread-safety in a ConcurrentCache, custom Cache's used by the ConcurrentCache must also be thread-safe. Custom evictors, however, are NOT required to be thread-safe.

The Cache interface can be found at
`ConcurrentCache/src/main/java/com/sameer/concurrentcache/Cache.java`

Implementation Details

ConcurrentCache

The ConcurrentCache maintains an ArrayList of N Caches (implemented as EvictorCaches with LRUEvictors by default). A `key value` pair is added to the ConcurrentCache by placing the pair in the subcache with index `key.hashCode()` modulo N. Assuming evenly distributed `hashCode()` values, it is likely that the key value pairs will be evenly distributed through the N subcaches. All of the major cache functionalities are handled through the functionalities of the subcaches - when a put, removal, or a check for a key is made, the corresponding check is made for the correct subcache for the input key.

The `createEvictor` method and the `createSubCache` method were both included to allow users to configure their own cache replacement policies. By default, when an instance of ConcurrentCache is constructed, `createSubCache` is called N times to create the subcaches. `createSubCache` returns a new instance of EvictorCache using an instance of Evictor returned from `createEvictor` (which returns an instance of LRUEvictor by default).

EvictorCache

An EvictorCache is initialized with a specified capacity and an instance of Evictor. Internally, key value pairs are stored using a HashMap. When any key addition, removal, or access is made, or when the EvictorCache is cleared, calls to the corresponding Evictor methods are made to pass this information on. When the EvictorCache has reached capacity and a new key value pair is to be added to the cache, the Evictor's `chooseEvict` method is called before the new pair is added, and the returned key is removed from the cache. The methods that can change the state of an EvictorCache or its Evictor are all synchronized in order to guarantee that when a ConcurrentCache uses an EvictorCache, the ConcurrentCache will be thread-safe.

LRUEvictor

The LRUEvictor is implemented using a LinkedHashSet that stores all of the keys of its corresponding EvictorCache. Java's LinkedHashSet stores elements in the order they were inserted. When an access or a reassignment is made to the EvictorCache, the corresponding key is reinserted to the LinkedHashSet, ensuring that the set maintains access order. Then, when `chooseEvict` is called, the first element of the cache is removed from the cache and returned.

MRUEvictor

The MRUEvictor simply keeps track of the last key either accessed or added to the cache. When `chooseEvict` is called, the key is returned. Nothing is done when `keyRemoved` is called, as in any case, the cache is either unchanged (the input key was not in the cache to begin with, so nothing was done), or the cache is below capacity, and `chooseEvict` will not be called until another key addition is made, thereby updating the tracked key.