N-Way Set-Associative Cache

Usage

Quick Start

To create a LRU Cache,

```
// This line is to create a 4-way (by default) set-associative LRU cache with 64 total cache entries.
Cache<Integer, Integer> lruCache = CacheFactory.createCache(64);
```

To put and get a key-value pair,

```
lruCache.put(1, 22);
lruCache.get(1);
```

Cache Size, Associativity and Bucket

While Cache Entry is the basic key-value store pair, Cache size is the total number of cache entry, i.e. the max number of key-value the cache is able to keep. Cache is consist of a fixed number of bucket, which is a group of cache entries. The associativity is the number of cache entries in bucket.

So a 4-way cache of size 64 has 64/4 = 8 cache buckets.

CacheFactory.createCache(int size) by default creates 4-way associative LRU cache. To customize the associativity,

```
// In this situation, it will create a LRU Cache with associativity of 8.
Cache<Integer, Integer> lruCache = CacheFactory.createCache(64, 8);
```

Cache Eviction Strategy

When a new key-value pair is being inserted into cache but the bucket it belongs to is full, cache eviction strategy is used to evict a record in this bucket before inserting new one.

By default, CacheFactory creates cache using LRU strategy,

```
CacheFactory.createCache(size, associativity) is equivalent to,
CacheFactory.createCache(new LRUCacheStrategy<K, V>(), size, associativity);
```

And to use MRU strategy,

```
CacheFactory.createCache(new MRUCacheStrategy<K, V>(), size, associativity);
```

However this can be customized by

```
CacheFactory.createCache(CacheStrategy<K, V> strategy, int size, int associativity);
```

Design

Data Structures

As an n-way associative cache is consist of a fixed number of cache buckets, we can think of cache as an array of CacheBucket. Given a key, CacheBucket is located by key.hashCode % numOfBuckets.

CacheBucket is the critical role that manages the records, including storing, indexing, and eviction strategy. As the records are being added and deleted frequently, array is not a very efficient data structure to store records. Map structure is good for looking up and add/delete keys however it doesn't have any orderness, which means it's difficult to implement strategy like LRU on top of that. With these considerations, I chose doubly-linked list as the backend data structure. It's called CacheEntryList.

CacheEntryList

CacheEntryList represents the cache entries in the cache bucket. It's implemented in the way of doubly-linked list so that it can add an entry, delete an entry or move an entry in the middle of list to head or tail at O(1). This list can be seen as a ranking of entries. And various types of cache strategy (algorithm such as LRU or MRU) may require the list to order in different ways.

CacheIndex

CacheIndex is a mapping from key to the reference of CacheEntry in CacheEntryList. As an index built on CacheEntryList, CacheIndex is introduced to speed up key lookup. If solely using CacheEntryList, key lookup takes O(n) because it has to iterate all the entries of bucket. With CacheIndex, we can see significant speed up when associativity is large. CacheIndex can be implemented in HashMap.

Customize Eviction Strategy

To create a Cache with alternative eviction algorithm, like LRC (least recent created) strategy, you can implement CacheStrategy with the following interfaces,

```
void postGet(CacheEntryList<K, V> cacheEntryList, CacheEntry<K, V> target);

K evict(CacheIndex<K, V> cacheIndex, CacheEntryList<K, V> cacheEntryList);

void doPut(CacheIndex<K, V> cacheIndex, CacheEntryList<K, V> cacheEntryList, CacheEntry<K, V> entry);
```

To illustrate how these interface help to build cache strategy, I can take the LRU Cache Strategy which already implemented in the project as an example.

First, postGet method will move the data entry to the tail of the cacheEntryList after each time using the entry, this operation can been seen as an update to the list so that the newly used data entry will always be at the tail of the list, while the least recently used data entry will always be at the head.

Second, evict method will remove the data entry to the head of cacheEntryList when the capacity of cache is met, the data entry will also be removed from the map. This operation ensures the cache size within the capacity, and the least recently used entry be removed from the cache, thus it is no longer able to be used.

The doPut method will put the newly added data entry (key-value pair) into the cache by adding the entry to the tail of the list and the map.

Implementing these methods in a different way (i.e. move newly used entry to a different location, remove entry in a different location, or put newly added entry to a different location, in the list of entries) will result in a new ranking of the entries, thus creating a new cache strategy.