N-Way Set Associative Cache

Implementation

A very versatile, modular and easy-to-use in-memory key-value storage.

The general data structure we're operating on is a number of "buckets" of "slots", or more formally, a Hashtable of Arrays.

The implementation of this cache is broken into the following "modules", each with their own concern. Each module listed is writen with extensibility in mind.

- `Container.java` keeps track of the Key, and it's Value. It also keeps track of cache-related metadata such as when this key-pair was last accessed, or whether the data is "dirty" (new data that only exists in the cache). If you wanted to create a new replacement strategy that wasn't based on when the container was `last_accessed`, you would extend this class.
- `ContainerSet.java` is an embodiment of a replacement strategy. `LRUContainerSet.java` and `MRUContainerSet.java` are examples of classes that extend `ContainerSet.java`. ContainerSet is an abstract class that contains the functionality of keeping track of which slots contain which key-pairs. As new key-value pairs enter the collection, it searches for empty slots, and if there are none, it defers the decision of which slot to evict to the `evict()` abstract function. This is the only function you'd need to override if you were to make a `FIFOContainerSet` for example.
- `Cache.java` a very lightweight class that executes the logic that checks what data is in the cache versus what data is in the DataStore. For example, if we try to `get(K)` an item, and it is not in the correct `ContainerSet`, we check the `DataStore` for it. If the item we're searching for is in the DataStore, we cache it.
- `DataStore.java` a very simple interface (`get` and `put`) that's used throughout the cache that you would implement to allow the Cache to communicate with any arbitary key-value store. The Cache uses the `DataStore` during any evict, or whenever `writeAllToDataStore()` is called. You could use DataStore.java to connect the Cache to a database, or even chain multiple caches across different deployments.

Usage

Creating and using a Cache:

```
Cache<String, Person> cache = new Cache(10, 3, myMongoDataStore);

cache.get("Parth") // Checks to see if "Parth" is in the cache. If not gets it from
myMongoDataStore and caches it.
```

.

Creating a cache with a custom replacement policy:

```
class RandomEvict<K, V> extends ContainerSet<K,V> { // An Aweful replacement
strategy, but a very simple one
    RandomEvict(int slots) { super(slots) }

    @Override
    int evict() {
        return (int) (super.size() * Math.random);
    }
}

Cache<String, Person> cache = new Cache(RandomEvict.class, 10, 3,
myRedisDataStore);
```

Distribution

This project is built and managed using Gradle, so upon deployment clients will be able to utilize the library simply by including something similar to:

```
compile 'com.TheTradeDesk.util:cache:1.0.0'
```

to their `build.gradle`

Included in folder

Running the test suite

Since this is a library, the only "executable" is the test suite. You can run that by invoking the gradle wrapper included in the source folder and passing the argument test:

If gradle is installed:

```
gradle run
```

`*nix`:

```
./gradlew run
```

`Windows`:

```
gradle.bat run
```

Documentation

Test reports and javadoc web pages are in the `javadoc/` and the `test_report/` folder respectively.