@Weifeng Bao 01/12/2016

Requirements

Implement an in-memory hashmap that put and read Key Value pairs. It should be highly scalable and safe.

Analysis

Components of HashMap

HashMap is a well known data structure that provides O(1) read/write performance. There are three main components for a hashmap design:

- hash function
- collision resolution
- storage resizing

Standard Implemenations

Java JDK provides three implemenations:

- java.util.HashMap
- java.util.Hashtable
- java.util.concurrent.ConcurrentHashMap

Issues with these implemenations

Safety

- java.util.HashMap is not thread safe. It can not be directly used in multiple threading scenario without external coordination.
- java.util.Hashtable is thread safe implemenation. It provides Object wide lock when performing put/remove.
- java.util.concurrent.ConcurrentHashMap is thread safe implementation and provides better concurrency. It takes a parameter to configure the number of Segments when constructing the object. Each Segment is concurrent accessing unit, will be locked during put/remove operations. The get operation is unblocked.

Scalability and Availability

In the environment that there are huge amount of data and highly concurrent access, the above implementations may suffer from the following perspectives:

- java.util.Hashtable may suffers from two sides
    - it uses global locker during write operations.
    - it uses dynamic resizing that doubles the memory storage whenever load factor is reached. Dynamic resizing may cause two issues:
        - it needs to copy the whole old array into a new double sized array, during the copy the whole table will be locked and not available to write. When there are millions of entries in the table, the copy operation may be costly and halt the concurrent operations.
        - memory waste. The memory size is doubled from previous size for every resizing, the bigger the data set is, the more extra memory is needed during resizing. When there are millions millions of entries in the table, there are higher chance to fail to allocate new big chunk of memory.
- java.util.concurrent.ConcurrentHashMap has a static configuration of the number of Segments. Each Segment is a sub hashmap and will also do dynamic resizing. It needs a very careful upfront planning to get optimial performance from ConcurrentHashMap.

Design

There are a lot methmatical research on how to build a high efficient evenly distributed hash function, I will go ahead borrow the mature algorithm from JDK implementation. The collision resolution algorithm I used is Linear Probing.

The focus of this design is to provide a implementation by improving the stoage resizing mechanism to meet following goals:

- dynamically scalable
- highly concurrent and thread safe
- highly available

The idea is to leverage the extendiable hashing algorithm (https://en.wikipedia.org/wiki/Extendible_hashing (https://en.wikipedia.org/wiki/Extendible_hashing)) to manage a collection of small buckets to provide:

- fine-grained parallel locking to improve parallel accessing throughput
- incremental resizing that has minimal impact to the over performance

Extendiable Hashing

Extendible hashing uses a Directory to manage a list of Buckets. A Directory consists of an array of pointers to Buckets. The array size must be in a power of 2 value. The array index maps to the lower bits of the key hashcode. The number of bits called the depth of bucket.

A particular key value pair is stored on one of the buckets as a HashEntry. One bucket is a small hashmap. When a Bucket is overflow, a Split needs to be done to resize the storage. During the split, the overflowed bucket will be locked and the existing entries will be migrated to the new buckets. Since the bucket size is fixed, the locking period is fairely small. Another impact of a split operation is that Directory may need to be double sized. Since the Directory size is in O(logN) order, the size typically is fairely small, the Directory resizing operation is also pretty fast.

The following sections describe the detail steps of put/get/remove operation, I will cover the details of Split operation in "put" operation.

Operation Steps

The following is the steps of put(K, V) operation:

```
1. get hashcode of K and apply hash function to the hashcode to ensure even distribution
2. ask Directory for the Bucket maps to hashcode
3. lock the Bucket
4. try to put K,V to the Bucket
4.1  if Bucket is overflow do the Bucket split
4.1.1  Allocate two new buckets.  Typically a bucket has small memory footprint, the allocation can be easily satisfied
4.1.2  Mark the old bucket invalid and disallow further write
4.1.3  Spread the old bucket entries into the two new buckets
4.1.4  put the new pair (K,V) into the one of the two new buckets.
4.1.5  Lock the Directory
4.1.6  If the new bucket local depth outgrow the depth of the Directory, then double the Directory size and rewire the existing bucket pointers to teh new Directory.
4.1.7  Register the two new buckets.
4.1.8  Unlock the Directory
5. unlock the Bucket
```

The following is the steps of get(K) operation:

```
1. get hashcode of K and apply hash function to the hashcode to ensure even distribution
2. ask Directory for the Bucket maps to hashcode
3. get entry of K from bucket
```

The following is the steps of remove(K) operation:

```
1. get hashcode of K and apply hash function to the hashcode to ensure even distribution
2. ask Directory for the Bucket maps to hashcode
3. lock the Bucket
4. try to find the entry of K
```

```
4.1 if found, set the value of K to null
4.2 if not found, ignore the operation
5. unlock the Bucket
```

Implementation Details

This implementation packaged in joyfulmonster.zip focus on implementing the core algorithm of the extendible hashing. The org.joyfulmonster.util.ConcurrentExtendiableHashMap API definition was influenced by java.util.concurrent.ConcurrentMap API, but I didn't try to do full implementation of java.util.concurrent.ConcurrentMap

package structure

Source Code

- src\main\java\org\joyfulmonster\util\ConcurrentExtendiableHashMap.java
- src\main\java\org\joyfulmonster\util\internal\ConcurrentExtendiableHashMapImpl.java
- src\main\java\org\joyfulmonster\util\internal\Bucket.java
- src\main\java\org\joyfulmonster\util\internal\BucketFactory.java
- src\main\java\org\joyfulmonster\util\internal\BucketMetricsSupport.java
- src\main\java\org\joyfulmonster\util\internal\BucketOverflowError.java
- src\main\java\org\joyfulmonster\util\internal\Directory.java
- src\main\java\org\joyfulmonster\util\internal\HashEntry.java
- src\main\java\org\joyfulmonster\util\internal\HashStrategy.java
- src\main\java\org\joyfulmonster\util\internal\LinearProbingBucketImpl.java
- src\main\java\org\joyfulmonster\util\internal\MetricsSupport.java

Note

- ConcurrentExtendiableHashMap.java is the proxy class to the actual implementation.
- ConcurrentExtendiableHashMapImpl.java is the actual implementation entrypoint. It holds of the reference to Directory and coordinate the execution steps stated above for different operations.
- Directory.java is an AtomicReference to a AtomicReferenceArray of Buckets. So the Directory object can be shared across multiple thread. The Bucket array maybe updated atomiclly.
- Bucket.java defines the interface a Bucket, there maybe various implementations.
- LinearProbingBucketImpl.java is a hashmap implementation of Bucket, it uses Linear Probing collision resolution.
- BucketFactory.java provides the facility to manage and replace different Bucket implementation without impact Directory and hashmap implemenation.
- BucketMetricsSupport.java defines a list of methods that measure the performance

metrics of a bucket

- HashStrategy.java captures different hash functions.
- HashEntry.java represents one entry that stores in a Bucket.
- MetricsSupport.java defines a list of methods that measure the performance metrics of a hashmap

Test Code

- src\test\java\org\joyfulmonster\util\BasicTest.java
- src\test\java\org\joyfulmonster\util\ConcurrencyTest.java
- src\test\java\org\joyfulmonster\util\RandomStringSet.java

Note

- BasicTest.java is a list of basic functional test that are not parallel.
- ConcurrencyTest.java is a list of tests that spawning multiple threads to execute parallel operations.

Document

- doc\README.md -> this file
- doc\README.pdf -> the printout of this file

Build

The project is built with gradle.

- build.gradle
- gradle
- gradlew
- gradlew.bat
- settings.gradle

In order to compile and run test do the following:

> *gradlew build test*

Worst case analysis

The worst case is all the entries falls into one buckets, and the bucket acts as a Hashtable. It may happen if the bucket size is configured very big or the hashcode falls into certain pattern. In order to mitigate the worst case, the following were done in the implementation:

- apply additional scramble hash function to ensure the hashcode is evenly distributed
- minimal bucket size is 2, default bucket size is 8
- it is recommended to careful pick up bucket size for different scenarios.

Future Improvement

The following are several future improvements in my mind:

- Make the class implement java.util.concurrent.ConcurrentMap
- The LinearProbingBucketImpl uses linear probing collision resolution algorithm, it may suffer from key clustering issue. Different flavor of Bucket implementation maybe valuable in some environments or usecases.
- More testing are needed:
    - I did not find a deterministic way to discover the contention condition in highly parallel environment. What I did was to stress the parallel operations in many rounds. There certainly maybe some scenarios missing. Advises are very welcome.
    - Careful measure the performance in different kinds of work load.
- Extendible hashing scales out incrementally pretty well, however, it does not scale down. Provided there are such usecase, it is interesting to research how to scale down.