Cache Lib of iZENEsoft

Peisheng Wang

March 18, 2009

Contents

1 Document History

Date	Author	Description
2009-03-18	Peisheng Wang	Describe the current Cache Lib at iZENEsoft

2 Introduction

Nowadays, a search engines have to answer thousands of queries per second with interactive response time. In a realistic search engine, query results, inverted list, intersection of inverted list, UID related info, Content of web page, even connection (like TCP) and some computation can be cached to reduce responding time and improve the efficiency dramatically.

Our goal is to provid a cache lib that we can easily build a reusable and high scalable cache (including memory and file) which can deal with those caching above.

3 About Cache-v2

Cache-v2 is hash cache and deals with key/value pair data. It is built on izenelib. It uses memory hash and file hash of izenelib as storage container.

There are linear hash, extendible hash, cccr-string-hash-talbe, dynamic perfect hash, sdb-hash avaible in izenelib. Cccr-string-hash-table(memory) and sdb-hash(file hash, used in SDB) are the most efficient memory hash and file hash in iZeneLib . and are recommended for underlying storage container.

Cache-v2 provide 3 categories: memory based hash(MCache/MLRUCache), file based hash(CacheDB), and hybrid(MFCache).

4 Features

4.1 Key/Valur Pair

Cache deals with key/value pair, where Key is a handle for Value. Given a Key, a Value must be returned if such cache item exists, otherwise returns false. The Cache should deal with all kinds of Key-Value pair.

4.2 Policy Based

MCache, MFCache, and CacheDB all are policy-based in from of C++ template class. They are highly extendible. They all have several configurable policies, including Replacement Policy, Storage Policy , Capacity Policy, and Thread Policy. Note that, MLRUCache only use LRU for Replacement Policy.

4.2.1 Storage Policy

Theoretically, all data structures that are Access Methods instance of izenelib can be used for underlying container. Howerver, for efficiency, we only use hash data structure. The basic interfaces for storage class required are

```
bool insert(key, value);
bool del(key);
bool find(key);
```

And for file hash,

```
bool open()
```

is also required.

Cache-v2 uses template template paramter to implement storage policy. The efficient hashes in izenelib make this cache lib very efficiency.

4.2.2 Replacement Policy

Replacement Policy determines what replacement policy to use. When the cache can not store any more cache items (key/value pairs), an item must be purged out of the cache storage in order to give a room for a new item which will be more likely to be accessed.

Among others, LRU and LFU are provided for a choice.

To maximize the Hit Ratio, many replacement algorithms have been studied. But no one algorithm is superior to any other algorithm under all circumstance. Different caching algorithm should be implemented according to the different visiting pattern to cache.

And Cache-v2 can implement new replacement policy very easily and Replacement policy is a template template paramter in Cache-v2.

4.2.3 Capacity Policy

Capacity Policy determines the cache capacity, i.e. the total number of cache items the Cache can keep. A constant value is a legitimate Capacity Policy, fixing the maximum number of cache items in the system. However, the cache capacity can vary dynamically depending on the availability of memory and the data size that the upper application layer can deals with.

```
void setCacheSize(size_t cacheSize)
```

is provided in Cache-v2 to set Capacity.

4.2.4 Thread Policy

Thread Policy determines the thread-safety level of the Cache system. No Thread Policy, for example, implements no thread policy but provides higher performance. If multiple clients access to Cache at the same time, for example, inserting/removing items while searching for the same items, concurrency control is a key issue. Thread Policy implements a proper concurrency control depending on its policy.

Cache-v2 support it by template template paramentes.

4.3 Consistence memchanism

If one item resides in cache for a long time, then its content may be out of date. So it need to check periodically whether an item is overdue.

For Cache-v2, we can set the life time for the cached item and they will be purged out when expired.

4.4 Serialization

The content in memory hash may be missing when Cache failed at run time.

```
template<class Archive>void save(Archive & ar,
        const unsigned int version = 0);
template<class Archive>void load(Archive & ar,
        const unsigned int version = 0);
```

are provided for recover.

4.5 Performance monitor

Get memory usage info method(void getEfficiency(double& hitRatio, double& workload)) are added to monitor the performance of Cache.

5 Cache Components

The Cache lib consists of these classes below:

```
class MFCache; //it resides both in memory and file.
class MCache;  //it only resides only in memory.
class MLRUCache;  //It is a MCache using LRU replacement
    policy, more fast and tight.
class CacheDB; //File based with MCache/MLRUCache.
```

To implement different replacement policies. MFCache and MCache use RB tree as container to use store CacheInfo of cached items.

5.1 MFCache

MFCache combine memory and file cache. It can set the proportion of memory hash size and file hash size. Items in memory have higher priority than in file. It also provied dump option for replacement policy.

```
template <class KeyType, class DataType, class
    ReplacementPolicy, class MemoryHash, Class
FileHash, class LockType=ReadWriteLock> class MFCache;
```

5.2 MCache

When MFCache has only memory hash, it becomes MCache.

```
template <class KeyType, class DataType, class
    ReplacementPolicy, class Hash,  class LockType=
    ReadWriteLock> class MCache;
```

5.2.1 MLRUCache

It is inefficient to use MCache(RB-tree) for replacement policy. For MLRU-Cache, we use std::list to store all the keys. Everytime, we insert an item into the cache, if it hits, we just delete it from the list, and insert it into the end of the list, and if not hits, we just insert it into the end of of the list. To save memory usage and achieve fast access of list items,

To save memory usage, we wrap cached items with iterator of the list.

```
typedef list<KeyType> CacheInfoList;
typedef typename list<KeyType> :: iterator LIT;

template<class K, class D>
struct _CachedData {
   D data;
   LIT lit;
   const K& get_key() const{
       return (const K&)data.get_key();
   }
};
typedef _CachedData<KeyType, DataType> CachedData;
typedef LinearHashTable<KeyType, CachedData, NullLock>
    linHash;
```

5.3 CacheDB

It is persistent that it stores all the key/value pairs in file hash(sdb_hash, linear hash file...), Plus, it also caches most of the items being used in memory such that it provides fast lookup to most of the retrieval calls. It has all the replacement policy, storage policy, and others. CacheDB can be used as our base DB for key-value pairs. Our version supports efficient caching explicitly (instead of relying on OS virtual memory/swapping system). It also support multi-threads and locking/concurrency.

It uses MCache or MLRUCache to implement its memory cache.

```
template <class KeyType, class ValueType, class
    ReplacementPolicy, class MCache,
                class DataHash, class ThreadSafeLock =
                    NullLock> class CacheDB
```

6 How to extend Cache?

6.1 New Replacement policy

We use CacheInfo to store the necessary info for Caching Algorithm.

`struct CacheInfo`

CacheInfo provides necessary information for Replacement Algorithm. Any item in CacheHash has only one corresponding CacheInfo, and vice verse. It contains these members below:

```
KeyType key;
size_t  docSize;
bool    isHit;
time_t LastAccessTime;
time_t FirstAccessTime;
time_t TimeToLive;
unsigned int iCount;
```

Through these info in CacheInfo above, we can implement all kinds of Replacement algorithm, including LRU, LFU, SLRU and so on. To implement different algorithms, we only need to provide different comparisons object between CacheInfo objects.

For example, for SLRU, the comparison is

```
template <class KeyType>
struct slruCmp{
//The item hits has higher priority than the item not
    hits.
bool operator() (const CacheInfo<KeyType> &lhs, const
    CacheInfo<KeyType> &rhs) const
{
    return (lhs.isHit < rhs.isHit)
        || ( (lhs.isHit == rhs.isHit) && (lhs.
            LastAccessTime < rhs.LastAccessTime) )
        || ( (lhs.isHit == rhs.isHit) && (lhs.
            LastAccessTime == rhs.LastAccessTime) && (lhs.
            iCount < rhs.iCount) )
        || ( (lhs.isHit == rhs.isHit) && (lhs.
            LastAccessTime == rhs.LastAccessTime) && (lhs.
            iCount == rhs.iCount) && (lhs.key < rhs.key) )
            ;
    }
};
```

6.2 Distributed Cache

Make it distributed.