izeneLib based on sf1lib and Ylib

Peisheng Wang

December 8, 2008

Abstract

This document is part of technical report for izenelib project. It depicts what can be going to izenelib from ylib and sf1lib, including many basical data structure and algrothm plus some useful technique.

Contents

1 Document History

Date	Author	Description
2008-12-08	Peisheng Wang	From ylib and sf1lib to izenelib

2 From ylib and sf1lib to izenelib

And some useful data structrue from ylib and sf1lib likes hashing methods and btree should be moved to izenelib, provided that it follows basic principles of izenelib. Cache and sequential-db should be built on izenelib.

3 ylib and sf1lib

3.1 KeyType and DataType

Many storages in izenelib are for key-value pair, where key and value can be of user-defined type . key is handle for value. And it is like pointer or something that help us to find the value. It is part of value and can be viewed as attribute of Value. For example, when value is a file, key is the path of file.

It is versatile to take advantage of user-defined comparison function. We also use boost::serialization and stringstream to support different DataType efficiently.

3.1.1 KeyType

KeyType can be Ystring, string, or user defined data types. Note that for btree or other storgae class providing sequential access, myKeyType must have thes following methods:

```
int compare(const myKeyType&otherkey);
```

eg.

```
typedef string myKeyType;
```

If we want to use **int** as KeyType for btree, we have to wrapper it. if KeyType is not supported by boost itself, serialize method must be provided also.

```
Struct myKeyType{
  int key;
  int compare(const myKeyType&other ){
    return key - other.key;
  }
  //for we use boost::serialization ,
  template<class Archive>
  void serialize(Archive & ar, const unsigned int version
      ){
      ar & key;
  }
}
```

3.1.2 DataType

DataType usually are user-defined data types. And it must have **key** member , and **get_key()** method. And it also should provide **serialize()** method if it uses default boost serialization method.

```
template<class T>
struct MyDataType {

 //if serialize method is private.
  friend class boost::serialization::access;
  string key;
  T data;
  const string get_key() const {
        return key;
  }
  template<class Archive>
  void serialize(Archive & ar, const unsigned int version
      ){
    ar & key;
    ar & data;
  }
};

typedef MyDataType<int> MyValueType;
```

When using SequentialDB or LinearHashFile, DataType must be converted into **DbObj**. Default Serialization Method are as follows:

```
template<class T> inline void read_image(T& dat, const
    DbObjPtr ptr) {
  stringstream istr((char*)ptr->getData());
  {
     boost::archive::text_iarchive ia(istr);
     ia & dat;
  }
}

template<class T> inline
void write_image(const T& dat, DbObjPtr ptr) {
   stringstream ostr;
   {
      boost::archive::text_oarchive oa(ostr);
      oa & dat;
   }
   ptr->setData(ostr.str().c_str(), ostr.str().size() );
}
```

However, the client can provide their serialization method for DataType. Below is an example for ystring.

```
template<> inline
void read_image<YString>(YString& dat, const DbObjPtr ptr
    )
{
    dat = (YString)((char*)ptr->getData() );
}

template<> inline
void write_image<YString>(const YString& dat, DbObjPtr
    ptr)
{
    ptr->setData(dat.c_str(), dat.size()+1);
}
```

3.2 thread safe policy

Thread Policy determines the thread-safety level of the storage component. No Thread Policy, for example, implements no thread policy but provides higher performance. If multiple clients access to storage object 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.

We implement NullLock and ReasWriteLock as follows:

```
class NullLock {
public:
        /**
         * Empty function.
         */
        inline int acquire_read_lock() {
                return 0;
        }

        inline int release_read_lock() {
                return 0;
        }

        inline int acquire_write_lock() {
                return 0;
        }

        inline int release_write_lock() {
                return 0;
        }
};

/**
 * @brief Simple Readwrite lock class using boost::
     share_mutex
 *
 */
class ReadWriteLock : private boost::noncopyable {
private:
        boost::shared_mutex rwMutex_;
public:
        /**
         * The constructor.
         */
        ReadWriteLock() {
        }
        /**
         * @brief Attempts to get the read lock.
         *
         */
        inline int acquire_read_lock() {
                rwMutex_.lock_shared();
                return 0;
        }

        /**
         * @brief Attempts to get the write lock.
         *
         */
        inline int acquire_write_lock() {
                rwMutex_.lock();
                return 0;
        }

        /**
         * @brief Attempts to release the read lock.
         *
         */
        inline int release_read_lock() {
                rwMutex_.unlock_shared();
                return 0;
        }

        inline int release_write_lock() {
                rwMutex_.unlock();
                return 0;
        }
};
```

And we can use NullLock and ReadWriteLock the way as below:

```
template<class ThreadSafePolicy>
class Storage{
ThreadSafePolicy lock_;
public:
   int foo()
   {
     lock_.acquire_read_lock();
     ....
     lock_.release_read_lock();
     return 0;
   }
   ...
}

Storage<NullLock> NotThreadSafeStorage;
Storage<ReadWriteLock> ThreadSafeStorage;
```

3.2.1 Thread-safe smart-pointer

When implementing SequentialDB, we using boost:intrusive_ptr. However, it it not thread-safe. And we provide one thread-safe one.

```
template<class ThreadSafePolicy> struct RefCount {
        int refCount;
        RefCount() :
                refCount(0) {
        }
        void refer() {
                ++refCount;
        }
        void unrefer() {
                if (--refCount == 0)
                        delete this;
        }
        virtual ~RefCount() {
        }
};

template<> struct RefCount<ReadWriteLock> {
        boost::detail::atomic_count refCount;
        //int refCount;
        RefCount() :
                refCount(0) {
        }
        void refer() {
                ++refCount;
        }
        void unrefer() {
                if (--refCount == 0)
                        delete this;
        }
        virtual ~RefCount() {
        }
};

inline void intrusive_ptr_add_ref(RefCount<NullLock> * p)
    {
        p->refer();
}
inline void intrusive_ptr_release(RefCount<NullLock> * p)
    {
        p->unrefer();
}
inline void intrusive_ptr_add_ref(RefCount<ReadWriteLock>
    * p) {
        p->refer();
}
```

```
inline void intrusive_ptr_release(RefCount<ReadWriteLock>
     * p) {
          p->unrefer();
}
```

3.2.2 multi-thread testing suite

All hasing methods have the following methods:

```
  DataType* find(const KeyType& key);

  const DataType* find(const KeyType& key);

  bool insert(const DataType& data);

  bool del(const KeyType& key) ;
  };
```

We also provide multi-thread testing framework for hashing methods.

```
template<class T>
struct run_thread_insert{
  run_thread_insert(char *str_,  T& cm_ ): cm(&cm_) {
          strcpy(str, str_);
 }
 void operator()() {
   ifstream inf(str);
         YString ystr;
         while (inf>>ystr ) {
                 sum++;
                 if (trace) {
                 //       boost::mutex::scoped_lock lock(
                     io_mutex);
                          cout<<str<<":␣insertValue:␣value=
                             "<<ystr<<endl;
                          cout<< "t1␣numItem␣=␣"<<cm->
                             num_items()<<endl;
                 }
                 if (cm->insert(ystr)){...}
         }
         /cout<< "t1␣numItem␣=␣"<<cm->num_items()<<endl;
 }
 char str[100];
 T*   cm;
};

template<class T>
struct run_thread_get{
 run_thread_get(char *str_,  T& cm_):cm(&cm_) {
         strcpy(str, str_);
 }
 void operator()() {
         ifstream inf(str);
         YString ystr;
         while (inf>>ystr ) {
                 if (trace) {
                 //boost::mutex::scoped_lock lock(io_mutex
                     );
                 cout<<str<<":␣getValue:␣key="<<ystr<<endl
                     ;
                 cout<< "t2␣numItem␣=␣"<<cm->num_items()<<
                     endl;
                 //cm.display();
                 }
                 if (cm->find(ystr.get_key()) ){;}
         }
         cout<< "t2␣numItem␣=␣"<<cm->num_items()<<endl;
         }
         char str[100];
         T * cm;
};

template<class T>
struct run_thread_del {
  run_thread_del(char *str_, T& cm_):cm(&cm_) {
         strcpy(str, str_);
  }
  void operator()() {
         ifstream inf(str);
         YString ystr;
         while (inf>>ystr) {
                 if (trace) {
                 //      boost::mutex::scoped_lock lock(
                     io_mutex);
                 cout<<str<<":␣del␣key:␣key="<<ystr<<endl;
                 cout<< "t4␣numItem␣=␣"<<cm->num_items()<<
                     endl;
                 //cm.display();
                 }
                 cm->del(ystr);
         }
         cout<< "t4␣numItem␣=␣"<<cm->num_items()<<endl;
         }
         char str[100];
         T* cm;
};

template<class T>
void run(T &cm) {

         boost::thread_group threads;
         int n = 1;
         for(int i=1; i<=n; i++)
         {
                 char fileName[1000];
                 sprintf(fileName, "dat/wordlist_%d.dat",
                     1);
                 threads.create_thread(run_thread_insert<T
                     >(fileName, cm) );

                 sprintf(fileName, "dat/wordlist_%d.dat",
                     2);
                 threads.create_thread(run_thread_insert<T
                     >(fileName, cm) );

                 sprintf(fileName, "dat/wordlist_%d.dat",
                     3);
                 threads.create_thread(run_thread_insert<T
                     >(fileName, cm) );

                 sprintf(fileName, "dat/wordlist_%d.dat",
                     4);
                 threads.create_thread(run_thread_get<T>(
                     fileName, cm) );

                 sprintf(fileName, "dat/wordlist_%d.dat",
                     5);
                 threads.create_thread(run_thread_get<T>(
                     fileName, cm) );

                 sprintf(fileName, "dat/wordlist_%d.dat",
                     6);
                 threads.create_thread(run_thread_get<T>(
                     fileName, cm) );

                 sprintf(fileName, "dat/wordlist_%d.dat",
                     7);
                 threads.create_thread(run_thread_del<T>(
                     fileName, cm) );

                 sprintf(fileName, "dat/wordlist_%d.dat"
                     ,8);
                 threads.create_thread(run_thread_del<T>(
                     fileName, cm) );
         }
         threads.join_all();
}
```

3.3 Serialization for memory storage

We provide serialization for memory storage likes **LinearHashTable ExtendibleHash** when using as underlying storage for **Cache**. and it has the following interfaces:

```
class LinearHashTable{
...
public:
/**
 *  save and load methods are compatible with boost::
    serization.
 */
template<class Archive> void save(Archive & ar,
                        const unsigned int version = 0)
                            const

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

/**
 *  dump is to shift data to other AccessMethos object.
 */
void dump(AccessMethods<KeyType,DataType> & am) const;
...
}
```

4 To be extracted

1. ProcMemInfo.h,it reads the current process' memory usage status, including real memory usage and virtual memory usage.
2. BTreeFile.h, file version of btree. However, its interface should be adjusted to be compatiable with AccessMethods.
3. Compressor.h, now it is only for order-preserving compressing, other compressing schema like Huffman algorithm can also be added.
4. ...

5 To be added

1. Smart pointer with threadsafe policy
2. .....

6 How to use izenelib