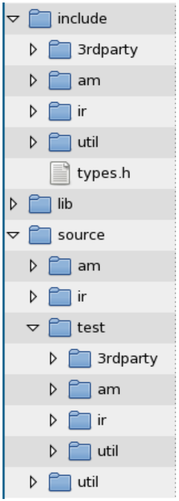
The Design and Structure of iZENELib

Yingfeng Zhang

December 12, 2008

Overview of Directory Structure

- It is recommended to put most of the source code under directory *include*.
- It is recommended to provide test programs for each component, which is put under the directory *test* accordingly. There are three kinds of test programs:
    1. Unit test, which is required to adopt *boost::test* library.
    2. Demonstration, which shows how to use the according component.
    3. Performance test.



```
include
    3rdparty
    am
    ir
    util
    types.h
lib
source
    am
    ir
    test
        3rdparty
        am
        ir
        util
    util
```

Overview of Directory Structure

- Directory *am* contains access methods including Linear Hashing,B-Tree,SkipList,etc.
- Directory *ir* contains components that are used for information retrieval and machine learning.
- Directory *util* contains common components.
- Some third party components that are required by iZENELib would be put under *3rdparty*.

Policy Based Design

Each component is required to designed based on policy. Take *am* as an example,We require all components under *am* implementing the interface *AccessMethod*:

```
template<typename KeyType, typename ValueType, typename LockType=NullLock,
        typename Alloc=std::allocator<DataType<KeyType,ValueType> > >
class AccessMethod{
public:
        virtual bool insert(const KeyType& key, const ValueType& value);
        virtual bool insert(const DataType<KeyType,ValueType>& data) = 0;
        virtual bool update(const KeyType& key, const ValueType& value);
        virtual bool update(const DataType<KeyType,ValueType>& data) = 0;
        virtual ValueType* find(const KeyType& key) = 0;
        virtual bool del(const KeyType& key) = 0;
}
```

We could change the policy of LockType and Alloc to satisfy different kinds of environment.

Concept Check

One of the most important aspect for adding components to iZENELib is to deciding which concepts to use. All concepts are required to be added to the *concept* sub directory of each component, and also, it is recommended to add concept check for each concept.For example:
We require the template parameter *KeyType* of AccessMethod to have implemented the method *compare*:

```
template<typename KeyType>
struct KeyTypeConcept{
        void constraints(){
                KeyType k1;
                KeyType k2;
                k1.compare(k2);
        }
}
BOOST_CONCEPT_ASSERT((KeyTypeConcept<KeyType>));
```

Namespace Management

It is required for all component to include the head file of 'types.h'

```
#define NS_IZENELIB_BEGIN namespace izenelib{
#define NS_IZENELIB_END }
#define NS_IZENELIB_AM_BEGIN namespace izenelib{ namespace am{
#define NS_IZENELIB_AM_END }}
#define NS_IZENELIB_IR_BEGIN namespace izenelib{ namespace ir{
#define NS_IZENELIB_IR_END }}
#define NS_IZENELIB_UTIL_BEGIN namespace izenelib{ namespace util{
#define NS_IZENELIB_UTIL_END }}
```

Namespace in each component should abey the rule, take *am* as an example:

```
NS_IZENELIB_AM_BEGIN

class AccessMethod{};
...

NS_IZENELIB_AM_END
```

Integer Type

It is required to include 'types.h' for using standard form of integers:

- int8_t
- int16_t
- int32_t
- int64_t

- uint8_t
- uint16_t
- uint32_t
- uint64_t

Exception

- If there exists any exception to throw within a certain component of iZENELib, it is recommended to inherit from *IZENELIBException* lying in *izenelib/include/util/Exception.h*
- *IZENELIBException* has already provided common exception definitions:
  1. Unknown error
  2. Generic error
  3. Missing parameter
  4. Bad parameter
  5. File I/O error
  6. Rumtime error
  7. Out of memory
  8. Illegal argument
  9. Unsupported operation
  10. Out of range

Logging

- It is recommended to adopt *boost::logging* utility for logging usage. It might be included by *boost* in future, and does not need to link other libraries when using, while other similiar logging utilities such as *Log4CPP*, *Log4CXX*, *glog* require that.
- It has been put under *izenelib/include/3rdparty/boost* already.

Iterator

- For those components that could provide sequential access, an iterator is necessary. It is recommended to provide the iterator which is inherited from *boost::iterator_facade*, for example:

```
template<typename KeyType, typename ValueType=void,
        typename EnableValue=void>
class DataType{};
template<template<typename , typename, typename> class DataType,
typename KeyType, typename ValueType=void>
class am_iterator :public boost::iterator_facade<
                am_iterator<DataType,KeyType,ValueType>
                ,DataType<KeyType,ValueType,void>
                ,boost::forward_traversal_tag
        >
{
public:
        am_iterator():pData_(0){}

        explicit am_iterator(DataType<KeyType,ValueType,void>* p)
        :pData_(p)
        {}
private:
        DataType<KeyType,ValueType,void>* pData_;
};
```

Memory Management

- Traditional memory management including *new&delete*, *std::allocator*.It has the following shortcomings:
    1. They are simple encapsulation of *malloc* and *free*, which is optimized for large trunk memory while not for small trunks.
    2. Frequently used traditional approaches will cause memory fragment.
    3. It is easy to cause memory leak.Although reference count is an approach, but it has shortcomings:
        - Not all of the C++ programmers like smart pointers, and not all of the C++ programmers like the SAME smart pointer. You have to convert between them.Then things become complex and difficult to control.
        - Having a risk of Circular Reference.
        - Tracking down memory leaks is more difficult

Memory Management

- An improved allocator is recommended to use-*boost::memory*, although it has not been included by *boost* yet.
- It is based on memory pool and garbage collection, and has conquered the limitations of traditional approaches:
  1. The speed of object construction and destroy has been accelerated to about 60 times.
  2. Memory fragment can never be caused.
  3. Users does not need to care the object destroy within a certain scope, thus memory leak is avoided.

Memory Management

- *boost::memory* still has some problems,yet, the unstable part of it has already been removed and the link library dependency has also been removed for easier usage. What's more, a memory hole has also been fixed.
- It is included under *izenelib/include/3rdparth/boost*, the usage and performance can be shown by *izenelib/source/test/3rdparth/boost/memory/**:

```
#include <boost/memory.hpp>
boost::memory::block_pool recycle;
boost::scoped_alloc alloc(recycle);
int* intObj = BOOST_NEW(alloc, int);
int* intObjWithArg = BOOST_NEW(alloc, int)(10);
int* intArray = BOOST_NEW_ARRAY(alloc, int, 100);
...

boost::scoped_alloc alloc;
std::set<int, std::less<int>, stl_allocator<int> >
                s(std::less<int>(), alloc);
for (int i = 0; i < Count; ++i)
        s.insert(i);
```

Memory Management

- Although it is easy to use, it has some issues to notice:
  1. The library dependency has been removed partialy by moving the singleton global objects from *.cpp to *.h, although it is safe because they do not lie in the interface head files, it may cause problems if it has been used in multi dynamic libraries.
  2. We could avoid the above limitation with two approaches:
     - Use -fPIC flag to g++ compile flags, which could create position independent shared library. With this, the global variables in different libraries can share the same address if these libraries are used by one program. Therefore the singelton semantic can still be reserved.
     - Singleton semantic of *boost::memory* exists only in this following usage:

       ```
       boost::scoped_alloc alloc;
       ```

       If we adopted another way, then no singleton semantic is required, shown later:

       ```
       boost::memory::block_pool recycle;
       boost::scoped_alloc alloc(recycle);
       ```

Memory Management

- A file based allocator is going to be developed. These allocators could be used as the template parameter, which could change the storage policy from memory to file seamlessly.

String Utility

- The shortcomings of *std::string*
  1. It is based on refence count, which performs terribly under multi-thread environment. Reference count has been proven unnecessary.
  2. Its allocator will cause memory fragment.
  3. It performs bad when treating large string.
- String utility to be added to iZENELib should conquer the above limitations, while providing the similar interface as *std::string*.